

Kotlin 官方文档

中文版

目录

关于本书 1.1

文档 1.1.1

入门 1.2

Kotlin 之旅 1.3

概述 1.4

Kotlin 多平台 1.4.1

Kotlin 用于服务器端开发 1.4.2

Kotlin 用于 Android 开发 1.4.3

Kotlin Wasm 1.4.4

Kotlin 原生 1.4.5

Kotlin 用于 JavaScript 开发 1.4.6

Kotlin 用于数据分析 1.4.7

Kotlin 用于 AI 驱动的应用开发 1.4.8

Kotlin 用于竞技程序设计 1.4.9

Kotlin 的新特性 1.5

Kotlin 2.1.20 1.5.1

Kotlin 2.1.0 1.5.2

Kotlin 2.2.0-RC 1.5.3

早期版本 1.5.4

Kotlin 2.0.20 1.5.4.1

Kotlin 2.0.0 1.5.4.2

Kotlin 1.9.20 1.5.4.3

Kotlin 1.9.0 1.5.4.4

Kotlin 1.8.20 1.5.4.5

Kotlin 1.8.0 1.5.4.6

Kotlin 1.7.20 1.5.4.7

Kotlin 1.7.0 1.5.4.8

Kotlin 1.6.20 1.5.4.9

Kotlin 1.6.0 1.5.4.10

Kotlin 1.5.30 1.5.4.11

Kotlin 1.5.20 1.5.4.12
Kotlin 1.5.0 1.5.4.13
Kotlin 1.4.30 1.5.4.14
Kotlin 1.4.20 1.5.4.15
Kotlin 1.4.0 1.5.4.16
Kotlin 1.3 1.5.4.17
Kotlin 1.2 1.5.4.18
Kotlin 1.1 1.5.4.19
版本发布与路线图 1.6
Kotlin 路线图 1.6.1
Kotlin 语言特性与提案 1.6.2
Kotlin 演进原则 1.6.3
Kotlin 各组件的稳定性 1.6.4
Kotlin 版本发布 1.6.5
基础 1.7
基本语法 1.7.1
习惯用法 1.7.2
例学 Kotlin↗ 1.7.3
编码规范 1.7.4
概念 1.8
类型 1.8.1
基本类型 1.8.1.1
概述 1.8.1.1.1
数字 1.8.1.1.2
无符号整型 1.8.1.1.3
布尔 1.8.1.1.4
字符 1.8.1.1.5
字符串 1.8.1.1.6
数组 1.8.1.1.7
类型检测与类型转换 1.8.1.2
控制流程 1.8.2
条件与循环 1.8.2.1
返回与跳转 1.8.2.2
异常 1.8.2.3

包与导入 1.8.3
类与对象 1.8.4
类 1.8.4.1
继承 1.8.4.2
属性 1.8.4.3
接口 1.8.4.4
函数式（SAM）接口 1.8.4.5
可见性修饰符 1.8.4.6
扩展 1.8.4.7
数据类 1.8.4.8
密封类与密封接口 1.8.4.9
泛型：in、out、where 1.8.4.10
嵌套类 1.8.4.11
枚举类 1.8.4.12
内联类 1.8.4.13
对象表达式与对象声明 1.8.4.14
委托 1.8.4.15
属性委托 1.8.4.16
类型别名 1.8.4.17
函数 1.8.5
函数 1.8.5.1
lambda 表达式 1.8.5.2
内联函数 1.8.5.3
操作符重载 1.8.5.4
构建器 1.8.5.5
类型安全的构建器 1.8.5.5.1
使用带有构建器类型推断的构建器 1.8.5.5.2
空安全 1.8.6
相等性 1.8.7
this 表达式 1.8.8
异步程序设计技术 1.8.9
协程 1.8.10
注解 1.8.11

解构声明 1.8.12
反射 1.8.13
数据分析 1.9
Kotlin Notebook 入门 1.9.1
Kotlin Notebook 简介 1.9.1.1
1、环境搭建 1.9.1.2
2、创建第一个 Kotlin Notebook 1.9.1.3
3、将依赖添加到 Kotlin Notebook 1.9.1.4
分享 Kotlin Notebook 1.9.2
Kotlin Notebook 的输出格式 1.9.3
使用数据源 1.9.4
从文件中取数据 1.9.4.1
从 web 源与 API 取数据 1.9.4.2
链接数据库并从中取数据 1.9.4.3
Kotlin Notebook 中数据可视化 1.9.5
用于数据分析的库 1.9.6
平台 1.10
JVM 1.10.1
Kotlin/JVM 入门 1.10.1.1
与 Java 比较 1.10.1.2
在 Kotlin 中调用 Java 1.10.1.3
在 Java 中调用 Kotlin 1.10.1.4
Spring 1.10.1.5
使用 Spring Boot 创建用到数据库的 RESTful web 服务 1.10.1.5.1
Spring Boot 与 Kotlin 入门 1.10.1.5.1.1
创建使用 Kotlin 的 Spring Boot 项目 1.10.1.5.1.2
向 Spring Boot 项目添加数据类 1.10.1.5.1.3
为 Spring Boot 项目添加数据库支持 1.10.1.5.1.4
使用 Spring Data CrudRepository 进行数据库访问 1.10.1.5.1.5
构建一个使用 Spring AI 基于 Qdrant 中存储的文档来回答问题的 Kotlin 应用——教程 1.10.1.5.2
Spring 框架 Kotlin 文档↗ 1.10.1.5.3
使用 Spring Boot 与 Kotlin 构建 web 应用程序——教程↗ 1.10.1.5.4
使用 Kotlin 协程与 RSocket 创建聊天应用程序——教程↗ 1.10.1.5.5

在 JVM 平台中用 JUnit 测试代码——教程 1.10.1.6
在项目中混用 Java 与 Kotlin——教程 1.10.1.7
在 Kotlin 中使用 Java 记录类型 1.10.1.8
从 Java 到 Kotlin 迁移指南 1.10.1.9
字符串 1.10.1.9.1
集合 1.10.1.9.2
可空性 1.10.1.9.3
标准输入 1.10.1.9.4
原生 1.10.2
Kotlin/Native 入门——在 IntelliJ IDEA 中 1.10.2.1
定义文件 1.10.2.2
与 C 语言互操作 1.10.2.3
与 C 语言互操作性 1.10.2.3.1
映射来自 C 语言的原始数据类型——教程 1.10.2.3.2
映射来自 C 语言的结构与联合类型——教程 1.10.2.3.3
映射来自 C 语言的函数指针——教程 1.10.2.3.4
映射来自 C 语言的字符串——教程 1.10.2.3.5
Kotlin/Native 开发动态库——教程 1.10.2.3.6
创建使用 C 语言互操作与 libcurl 的应用——教程 1.10.2.3.7
与 Objective-C 互操作性 1.10.2.4
与 Swift/Objective-C 互操作性 1.10.2.4.1
Kotlin/Native 开发 Apple framework——教程 1.10.2.4.2
Kotlin/Native 库 1.10.2.5
平台库 1.10.2.6
内存管理器 1.10.2.7
Kotlin/Native 内存管理 1.10.2.7.1
与 ARC 集成 1.10.2.7.2
迁移到新版内存管理器 1.10.2.7.3
调试 Kotlin/Native 1.10.2.8
符号化 iOS 崩溃报告 1.10.2.9
参考与提示 1.10.2.10
目标平台支持 1.10.2.10.1
改进编译时间 1.10.2.10.2
许可证文件 1.10.2.10.3

Kotlin/Native FAQ 1.10.2.10.4
WebAssembly（Wasm） 1.10.3
在 IntelliJ IDEA 中入门 Kotlin/Wasm 1.10.3.1
Kotlin/Wasm 与 WASI 入门 1.10.3.2
调试 Kotlin/Wasm 代码 1.10.3.3
与 JavaScript 的互操作 1.10.3.4
故障排查 1.10.3.5
JavaScript 1.10.4
搭建 Kotlin/JS 项目 1.10.4.1
运行 Kotlin/JS 1.10.4.2
开发服务器与持续编译 1.10.4.3
调试 Kotlin/JS 代码 1.10.4.4
在 Kotlin/JS 平台中运行测试 1.10.4.5
Kotlin/JS 无用代码消除 1.10.4.6
Kotlin/JS IR 编译器 1.10.4.7
Kotlin 用于 JS 平台 1.10.4.8
浏览器与 DOM API 1.10.4.8.1
在 Kotlin 中使用 JavaScript 代码 1.10.4.8.2
动态类型 1.10.4.8.3
使用来自 npm 的依赖 1.10.4.8.4
在 JavaScript 中使用 Kotlin 代码 1.10.4.8.5
JavaScript 模块 1.10.4.8.6
Kotlin/JS 反射 1.10.4.8.7
类型安全的 HTML DSL 1.10.4.8.8
脚本 1.10.5
Kotlin 自定义脚本入门——教程 1.10.5.1
Kotlin 多平台开发↗ 1.11
标准库 1.12
集合 1.12.1
集合概述 1.12.1.1
构造集合 1.12.1.2
迭代器 1.12.1.3
区间与数列 1.12.1.4

序列 1.12.1.5
集合操作概述 1.12.1.6
集合转换操作 1.12.1.7
过滤集合 1.12.1.8
加减操作符 1.12.1.9
分组 1.12.1.10
取集合的一部分 1.12.1.11
取单个元素 1.12.1.12
排序 1.12.1.13
聚合操作 1.12.1.14
集合写操作 1.12.1.15
List 相关操作 1.12.1.16
Set 相关操作 1.12.1.17
Map 相关操作 1.12.1.18
读取标准输入 1.12.2
选择加入要求 1.12.3
作用域函数 1.12.4
时间度量 1.12.5
官方库 1.13
协程（kotlinx.coroutines） 1.13.1
协程指南 1.13.1.1
协程基础 1.13.1.2
Kotlin 协程与通道介绍↗ 1.13.1.3
取消与超时 1.13.1.4
组合挂起函数 1.13.1.5
协程上下文与调度器 1.13.1.6
异步流 1.13.1.7
通道 1.13.1.8
协程异常处理 1.13.1.9
共享的可变状态与并发 1.13.1.10
select 表达式（实验性的） 1.13.1.11
使用 IntelliJ IDEA 调试协程——教程 1.13.1.12
使用 IntelliJ IDEA 调试 Kotlin Flow——教程 1.13.1.13
序列化（kotlinx.serialization） 1.13.2

Kotlin Metadata JVM（kotlin-metadata-jvm） 1.13.3
Lincheck（kotlinx.lincheck） 1.13.4
Lincheck 指南 1.13.4.1
用 Lincheck 编写第一个测试 1.13.4.2
测试策略 1.13.4.3
操作参数 1.13.4.4
数据结构约束 1.13.4.5
进度保证 1.13.4.6
时序规范 1.13.4.7
Ktor↗ 1.13.5
API 参考 1.14
标准库（stdlib）↗ 1.14.1
测试库（kotlin.test）↗ 1.14.2
协程（kotlinx.coroutines）↗ 1.14.3
序列化（kotlinx.serialization）↗ 1.14.4
Kotlin I/O 库（kotlinx-io）↗ 1.14.5
日期与时间（kotlinx-datetime）↗ 1.14.6
JVM 元数据（kotlinx-metadata-jvm）↗ 1.14.7
Ktor↗ 1.14.8
Kotlin Gradle 插件（kotlin-gradle-plugin）↗ 1.14.9
语言参考 1.15
关键字与操作符 1.15.1
语法↗ 1.15.2
语言规范↗ 1.15.3
工具 1.16
构建工具 1.16.1
Gradle 1.16.1.1
Gradle 概述 1.16.1.1.1
Gradle 入门——教程 1.16.1.1.2
配置 Gradle 项目 1.16.1.1.3
Gradle 最佳实践 1.16.1.1.4
Kotlin Gradle 插件中的编译器选项 1.16.1.1.5
Kotlin Gradle 插件中的编译项与缓存 1.16.1.1.6

支持 Gradle 插件变体 1.16.1.1.7
Maven 1.16.1.2
Ant 1.16.1.3
Dokka 1.16.2
简介 1.16.2.1
Dokka 入门 1.16.2.2
运行 Dokka 1.16.2.3
Gradle 1.16.2.3.1
Maven 1.16.2.3.2
CLI 1.16.2.3.3
输出格式 1.16.2.4
HTML 1.16.2.4.1
Markdown 1.16.2.4.2
Javadoc 1.16.2.4.3
Dokka 插件 1.16.2.5
模块文档 1.16.2.6
用于 Kotlin 开发的 IDE 1.16.3
使用 IntelliJ IDEA 迁移到 Kotlin 代码风格 1.16.4
Kotlin Notebook 1.16.5
用于 Kotlin 的 Lets-Plot 1.16.6
运行代码片段 1.16.7
Kotlin 与 TeamCity 的持续集成 1.16.8
编写代码文档：KDoc 1.16.9
Kotlin 与 OSGi 1.16.10
编译器与插件 1.17
编译器 1.17.1
K2 编译器迁移指南 1.17.1.1
Kotlin 命令行编译器 1.17.1.2
Kotlin 编译器选项 1.17.1.3
编译器插件 1.17.2
全开放编译器插件 1.17.2.1
No-arg 编译器插件 1.17.2.2
带有接收者的 SAM 编译器插件 1.17.2.3
kapt 编译器插件 1.17.2.4

Lombok 编译器插件 1.17.2.5
Power-assert 1.17.2.6
Compose 编译器 1.17.3
Compose 编译器迁移指南 1.17.3.1
Compose 编译器选项 DSL 1.17.3.2
Kotlin 符号处理（KSP）API 1.17.4
KSP 概述 1.17.4.1
KSP 快速入门 1.17.4.2
为什么选用 KSP 1.17.4.3
KSP 示例 1.17.4.4
KSP 如何为 Kotlin 代码建模 1.17.4.5
Java 注解处理对应到 KSP 参考 1.17.4.6
增量处理 1.17.4.7
多轮次处理 1.17.4.8
KSP 与 Kotlin 多平台 1.17.4.9
在命令行运行 KSP 1.17.4.10
常见问题 1.17.4.11
学习资料 1.18
学习资料概述 1.18.1
例学 Kotlin↗ 1.18.2
Kotlin 心印 1.18.3
Kotlin 核心课程↗ 1.18.4
Kotlin 动手实践 1.18.5
Kotlin 小窍门 1.18.6
图书 1.18.7
Advent of Code 谜题 1.18.8
在 IDE 中学习（JetBrains 学院） 1.18.9
学习 Kotlin 1.18.9.1
讲授 Kotlin 1.18.9.2
库作者指南 1.18.10
简介 1.18.10.1
最小化心理复杂性 1.18.10.2
概述 1.18.10.2.1

简单 1.18.10.2.2
可读性 1.18.10.2.3
一致性 1.18.10.2.4
可预测性 1.18.10.2.5
可调试性 1.18.10.2.6
可测试性 1.18.10.2.7
反向兼容性 1.18.10.3
内容翔实的文档 1.18.10.4
构建多平台库 1.18.10.5
早期访问计划（EAP） 1.19
参与 Kotlin 早期访问计划 1.19.1
配置构建采用 EAP 1.19.2
其他资源 1.20
FAQ 1.20.1
Kotlin 兼容性指南 1.20.2
Kotlin 2.1 的兼容性指南 1.20.2.1
Kotlin 2.0 的兼容性指南 1.20.2.2
Kotlin 1.9 的兼容性指南 1.20.2.3
Kotlin 1.8 的兼容性指南 1.20.2.4
Kotlin 1.7.20 的兼容性指南 1.20.2.5
Kotlin 1.7 的兼容性指南 1.20.2.6
Kotlin 1.6 的兼容性指南 1.20.2.7
Kotlin 1.5 的兼容性指南 1.20.2.8
Kotlin 1.4 的兼容性指南 1.20.2.9
Kotlin 1.3 的兼容性指南 1.20.2.10
兼容模式 1.20.2.11
Kotlin 基金会 1.20.3
Kotlin 基金会↗ 1.20.3.1
语言委员会准则↗ 1.20.3.2
提交不兼容变更指南↗ 1.20.3.3
Kotlin 品牌用途准则↗ 1.20.3.4
Kotlin 基金会 FAQ↗ 1.20.3.5
Google Summer of Code 1.20.4
概述 1.20.4.1

Google Summer of Code with Kotlin 2025↗ 1.20.4.2
Google Summer of Code with Kotlin 2024↗ 1.20.4.3
Google Summer of Code with Kotlin 2023↗ 1.20.4.4
安全 1.20.5
社区 1.20.6
贡献力量 1.20.6.1
KUG 准则 1.20.6.2
Kotlin 之夜准则 1.20.6.3
Kotlin Slack 的行为准则与指南 1.20.6.4
Kotlin 品牌素材 1.20.7
宣传资料↗ 1.20.8

Kotlin 官方文档 中文版

本书是 Kotlin 语言官方文档的中文翻译，内容来自 Kotlin 中文站项目。

本书会与 Kotlin 中文站及 JetBrains 的 Kotlin 官方站准同步更新。 所以请时常来更新，或者阅读在线版本：https://book.kotlincn.net/。

2017-05-13，Kotlin 中文站的参考文档已翻译校对完毕且与官方站同步。 随着官网新增内容越来越多，尤其是 2021 年版本库结构大改还导致翻译停滞了一段时间，已经有不少内容尚未翻译。 目前在不断完善翻译中，有些贡献者基于旧版本翻译的内容也会逐步解决冲突合并进来。 当然，今后还会持续与官网保持同步更新并继续翻译，请随时关注。 欢迎大家一起来翻译/改进，翻译前请阅读翻译指南草稿。

目前已更新至 2.1.21。最近一次更新：2025-06-17。

本书采用 Apache License 2.0 许可发布，因内容来源采用该许可。

下载离线版本： PDF | EPUB | MOBI

有任何问题或建议，可在 Kotlin 中文站反馈。

欢迎关注我的博客《灰蓝时光》（ https://hltj.me/ ）以及公众号与微博：

公众号	微博

Kotlin 文档

最新稳定版：2.1.21

Kotlin 入门

在 IDE 中（IntelliJ IDEA 或 Android Studio）为所选平台创建第一个 Kotlin 项目

在线试用 **Kotlin**

直接在浏览器中编写、编辑、运行与共享 Kotlin 代码

第一步

基础语法

Kotlin 语法的快速简介：关键字、操作符、程序结构

Kotlin 之旅

Kotlin 编程语言的基础知识导览

Kotlin 多平台

Kotlin 多平台概述

探索 Kotlin 多平台技术来在应用程序之间共享代码

Kotlin 多平台向导

快速创建并下载多平台项目模板

Kotlin 多平台入门

创建一个在 Android 与 iOS 上都能运行的移动应用

Compose 多平台入门

使用 Compose 多平台来实现跨移动端、桌面端和 web 端的统一用户界面

精选主题

Kotlin 2.1.20 的新特性

新的实验性特性、性能改进与问题修复

Kotlin 版本发布

最新的 Kotlin 版本及其更新说明

标准库 **API** 参考

Kotlin 日常工作必备：IO、 文件、 线程、 集合等等

基本类型

Kotlin 类型系统：数字、字符串、数组与其他内置类型

集合

集合：list、set 与 map

协程

并发：协程、流、通道

Kotlin 入门

最新 Kotlin 版本： **2.1.21**

Kotlin 是一门现代但已成熟的编程语言，旨在让开发人员更幸福快乐。 它简洁、安全、可与 Java 及其他语言互操作，并提供了多种方式在多个平台间复用代码，以实现高效编程。

首先，何不参加 Kotlin 之旅[1]呢？ 该教程涵盖了 Kotlin 编程语言的基础知识并且可以完全在浏览器中完成。



安装 Kotlin

Kotlin 已包含在每个 IntelliJ IDEA 与 Android Studio 版本中了。

可下载并安装这两个 IDE 之一来开始使用 Kotlin。

选择你的 Kotlin 使用场景

> 【控制台】

在这里可学到如何使用 Kotlin 开发控制台应用程序并创建单元测试。

1. 使用 **IntelliJ IDEA** 项目向导创建一个基本的 **JVM** 应用程序.
2. 编写第一个单元测试.

> 【后端】

在这里可学到如何使用 Kotlin 服务端开发后端应用程序。

1. 创建第一个后端应用程序：

    - 创建使用 Spring Boot 的 RESTful Web 服务
    - 创建使用 Ktor 的 HTTP API
2. 了解如何在应用程序中混合使用 **Kotlin** 与 **Java** 代码.

> 【跨平台】

在此可以了解到如何使用 Kotlin 多平台开发跨平台应用程序。

1. 搭建用于跨平台开发的环境.
2. 创建第一个用于 **iOS** 与 **Android** 应用程序：
    - 从头开始创建跨平台应用程序，并：
        - 共享业务逻辑，同时保持 UI 原生
        - 共享业务逻辑与 UI
    - 让现有的 Android 应用程序能在 iOS 上运行
    - 创建使用 Ktor 与 SQLdelight 的跨平台应用程序
3. 探索样例项目.

> 【Android】

- 如需开始使用 Kotlin 用于 Android 开发，请阅读 谷歌对 Android 上 Kotlin 入门的建议。

> 【数据分析】

无论是构建数据流水线，还是将机器学习模型投入生产，Kotlin 都是处理数据并充分发挥其价值的绝佳选择。

1. 在 **IDE** 中无缝创建并编辑 **notebook**：
    - Kotlin Notebook 入门
2. 探索并试验数据：
    - DataFrame – 用于数据分析与处理的库。
    - Kandy – 用于数据可视化的绘图工具。
3. 在 **Twitter** 上关注 **Kotlin** 用于数据分析： KotlinForData.

加入 **Kotlin** 社区

持续关注 Kotlin 生态系统的最新动态并分享你的经验。

- 加入我们：
  - Slack：获取邀请.
  - StackOverflow：订阅“kotlin”标签。
- 在 Youtube、 Twitter、 Bluesky 与 Reddit 上关注 Kotlin。
- 订阅 Kotlin 资讯。

如果遇到任何困难或问题，请在我们的问题跟踪器中报告问题。

还缺少什么？

如果本页有任何遗漏或令人困惑之处，请提交反馈。

> [1]. 译注：此处双关，“tour of Kotlin”字面意为“科特林岛之旅”，此处意为“Kotlin（入门）教程”。 ↩

欢迎参加 Kotlin 之旅！

> 这些教程可以完全在浏览器中完成。不需要安装。
>
> 

通过这些导览快速掌握 Kotlin 编程语言的精髓，将你从入门提升到中级水平。 每一章都包含：

- 理论：通过示例介绍语言的关键概念。
- 实践：通过练习来检验对所学内容的理解。
- 答案：参考答案。

从我们的初学者之旅开始掌握基础知识：

Start beginner Kotlin tour →

- 变量
- 基本类型
- 集合
- 控制流程
- 函数
- 类
- 空安全

如果已准备将对 Kotlin 的理解提升到更高水平，请参加我们的进阶之旅：

Start intermediate Kotlin tour →

- 扩展函数
- 作用域函数
- 带接收者的 lambda 表达式
- 类与接口
- 对象
- 开放类与特殊类
- 属性
- 空安全
- 库与 API

概述

- Kotlin 多平台
- Kotlin 用于服务器端开发
- Kotlin 用于 Android 开发
- Kotlin Wasm
- Kotlin 原生
- Kotlin 用于 JavaScript 开发
- Kotlin 用于数据分析
- Kotlin 用于 AI 驱动的应用开发
- Kotlin 用于竞技程序设计

Kotlin 多平台

简化跨平台项目的开发，并减少为不同平台编写和维护相同代码所花费的时间。

Kotlin 多平台快速入门

搭建并运行第一个 KMP 项目

Compose 多平台

探索 JetBrains 开发的基于 Kotlin 的声明式多平台 UI 框架

热门使用场景

共享逻辑但保持 **UI** 原生

了解 Kotlin 多平台并在 Android Studio 中创建在 Android 与 iOS 上都能运行的移动应用

共享逻辑与 **UI**

使用 Compose 多平台创建一个在 Android、iOS 与桌面端共享 UI 的应用程序

共享一段逻辑

创建一个在 Android 与 iOS 之间共享数据访问层的多平台应用

让应用支持多平台

让 Android 应用程序能在 iOS 上运行

Kotlin 多平台技术

项目结构的关键概念

了解基础知识：公共代码与平台专用代码、目标以及源代码集

预期与实际声明

使用预期声明与实际声明来访问平台特定 API

分层项目结构

为共享公共代码安排源代码集的分层结构

设置 **iOS** 集成

将 Kotlin 多平台共享模块集成到 iOS 应用中

教程与样例

搭建并运行第一个 **KMP** 项目

搭建并运行第一个 KMP 项目

让应用支持多平台

让 Android 应用程序能在 iOS 上运行

测试多平台应用

解如何创建并测试 Kotlin 多平台应用程序

参见精选项目列表

Kotlin 多平台样例

在 **GitHub** 上查找更多项目

GitHub“kotlin-multiplatform”主题

工具

Kotlin 多平台 **IDE** 插件

用于 IntelliJ IDEA 与 Android Studio 的 Kotlin 多平台插件

Compose 热加载

使用 Compose 热加载快速迭代 UI

Gradle DSL 参考

为 Kotlin 多平台项目配置 Gradle 构建脚本

库搜索平台

由 JetBrains 开发的 Kotlin 多平台库实验性搜索平台

新建项目向导

快速生成并下载多平台项目模板

Amper 配置工具

使用 Amper 配置项目

社区与帮助

加入 **Kotlin** 官方的 **Slack** 频道

Kotlin Slack

在 **StackOverflow** 上订阅 **Kotlin** 多平台标签

订阅 kotlin-multiplatform 标签

在 **YouTrack** 中报告问题

Kotlin 问题追踪系统

Kotlin 用于服务器端开发

Kotlin 非常适合开发服务器端应用程序。它可以让你编写简明且表现力强的代码， 同时保持与现有基于 Java 的技术栈的完全兼容性以及平滑的学习曲线：

- 表现力：Kotlin 的革新式语言功能，例如支持类型安全的构建器和委托属性，有助于构建强大而易于使用的抽象。
- 可伸缩性：Kotlin 对协程的支持有助于构建服务器端应用程序， 伸缩到适度的硬件要求以应对大量的客户端。
- 互操作性：Kotlin 与所有基于 Java 的框架完全兼容，因此可以使用你熟悉的技术栈，同时获得更现代化语言的优势。
- 迁移：Kotlin 支持大型代码库从 Java 到 Kotlin 逐步迁移。你可以开始用 Kotlin 编写新代码，同时系统中较旧部分继续用 Java。
- 工具：除了很棒的 IDE 支持之外，Kotlin 还为 IntelliJ IDEA Ultimate 的插件提供了框架特定的工具（例如 Spring 与 Ktor）。
- 学习曲线：对于 Java 开发人员，Kotlin 入门很容易。包含在 Kotlin 插件中的自动 Java-to-Kotlin 的转换器有助于迈出第一步。Kotlin 心印 通过一系列互动练习提供了语言主要功能的指南。 像 Ktor 这样的 Kotlin 专用框架提供了简单直接的方式，而没有大型框架隐藏的复杂性。

使用 Kotlin 进行服务器端开发的框架

以下是 Kotlin 服务器端框架的一些示例：

- Spring 利用 Kotlin 的语言功能提供更简洁的 API， 从版本 5.0 开始。在线项目生成器可以让你用 Kotlin 快速生成一个新项目。
- Ktor 是 JetBrains 为在 Kotlin 中创建 Web 应用程序而构建的框架，利用协程实现高可伸缩性，并提供易于使用且合乎惯用法的 API。
- Quarkus 为使用 Kotlin 提供了头等支持。 它是由 Red Hat 维护的开源框架。 Quarkus 是为 Kubernetes 全新构建、并利用数百个（还在增加）最佳库提供内聚的全栈框架。
- Vert.x 是在 JVM 上构建反应式 Web 应用程序的框架， 为 Kotlin 提供了专门支持，包括完整的文档。
- kotlinx.html 是可在 Web 应用程序中用于构建 HTML 的 DSL。 它可以作为传统模板系统（如JSP和FreeMarker）的替代品。
- Micronaut 是基于 JVM 的现代全栈框架，用于构建模块化、易于测试的微服务与无服务器应用程序。它带有许多有用的内置特性。

- http4k是一个纯 Kotlin 编写、占用空间很小的用于 Kotlin HTTP 应用程序的函数式工具包。 该库基于 Twitter 的论文《你的服务器即函数》（Your Server as a Function），并将 HTTP 服务器与客户端都建模为可以组合起来的简单 Kotlin 函数。
- Javalin 是用于 Kotlin 与 Java 的非常轻量级的 Web 框架，支持 WebSockets、HTTP2 与异步请求。
- 通过相应 Java 驱动程序进行持久化的可用选项包括直接 JDBC 访问、JPA 以及使用 NoSQL 数据库。 对于 JPA，kotlin-jpa 编译器插件使 Kotlin 编译的类适应框架的要求。

> 可以在 https://kotlin.link/ 找到更多框架。
>
> 

部署 Kotlin 服务器端应用程序

Kotlin 应用程序可以部署到支持 Java Web 应用程序的任何主机，包括 Amazon Web Services、Google Cloud Platform 等。

要在 Heroku 上部署 Kotlin 应用程序，可以按照 Heroku 官方教程来做。

AWS Labs 提供了一个示例项目，展示了 Kotlin 编写 AWS Lambda 函数的使用。

谷歌云平台（Google Cloud Platform）提供了一系列将 Kotlin 应用程序部署到 GCP 的教程，包括 Ktor 与 App Engine 应用及 Spring 与 App engine 应用。此外， 还有一个交互式代码实验室（interactive code lab）用于部署 Kotlin Spring 应用程序。

Kotlin 用于服务器端的产品

Corda 是一个开源的分布式分类帐平台，由各大银行提供支持 ，完全由 Kotlin 构建。

JetBrains 账户，负责 JetBrains 整个许可证销售和验证过程的系统 100% 由 Kotlin 编写，自 2015 年生产运行以来，一直没有重大问题。

chess.com 是一个专注于国际象棋的网站，服务于全球数百万热爱这项游戏的玩家。它使用 Ktor 来无缝配置多个 HTTP 客户端。

Adobe 的工程师使用 Kotlin 进行服务器端应用开发，并使用 Ktor 在 Adobe 体验平台中进行原型设计，该平台使组织能在应用数据科学与机器学习之前集中并标准化客户数据。

下一步

- 关于更深入的介绍，请查看本站的 Kotlin 文档及 Kotlin 心印。

- 探索如何使用 Ktor 构建异步服务器应用程序，Ktor 是一个采用 Kotlin 协程的框架。
- 观看网络研讨会“Micronaut for microservices with Kotlin” 并浏览详细指南， 了解如何在 Micronaut 框架中使用 Kotlin 扩展函数。
- http4k 提供了生成完整项目的 CLI（译注：命令行界面），以及通过单条 bash 命令使用 GitHub、Travis 与 Heroku 生成整套 CD（译注：持续交付）流水线的 starter 仓库。
- 想要从 Java 迁移到 Kotlin 吗？了解下在 Java 与 Kotlin 中如何处理字符串的典型任务。

Kotlin 用于 Android 开发

自 2019 年 Google I/O 以来，Kotlin 就成为了 Android 移动开发的首选。

超过 50% 的专业 Android 开发者使用 Kotlin 作为主要语言，而只有 30% 使用 Java 作为主要语言。 70% 以 Kotlin 为主要语言的开发人员表示，Kotlin 提高了他们的工作效率。

使用 Kotlin 进行 Android 开发，可以得益于：

- 代码更少、可读性更强。 花更少的时间来编写代码与理解他人的代码 。
- 常见错误更少。 根据 Google的内部数据，用 Kotlin 构建的应用崩溃的可能性低 20%。
- **Jetpack** 库中的 **Kotlin** 支持。 Jetpack Compose 是 Android 推荐的现代工具包，用于在 Kotlin 中构建原生 UI。 KTX 扩展为现有的 Android 库添加了 Kotlin 语言特性，如协程、扩展函数、 lambda 表达式与命名参数。
- 支持多平台开发。 Kotlin 多平台不仅支持 Android 开发，还支持 iOS、 后端以及 web 应用程序开发。 一些 Jetpack 库已经是多平台的了。 Compose 多平台是 JetBrains 基于 Kotlin 与 Jetpack Compose 的声明式 UI 框架， 能够跨平台（iOS、Android、桌面与 web）共享 UI。
- 成熟的语言与环境。 自 2011 年创建以来，Kotlin 不仅通过语言而且通过强大的工具在整个生态系统中不断发展。 现在，它已无缝集成到 Android Studio 中， 并被许多公司积极用于开发 Android 应用程序。
- 与 **Java** 的互操作性。 可以在应用程序中将 Kotlin 与 Java 编程语言一起使用， 而无需将所有代码迁移到 Kotlin。
- 易学易用。Kotlin 非常易于学习，尤其是对于 Java 开发人员而言。
- 大社区。 Kotlin 得到了社区的大力支持与许多贡献，该社区在全世界范围内都在增长 。排名前一千的 Android 应用中超过 95% 使用 Kotlin。

许多初创公司与财富 500 强公司已经使用 Kotlin 开发了 Android 应用程序，参见面向 Android 开发者的谷歌网站上的列表。

开始将 Kotlin 用于：

- Android 开发，请阅读 Google 关于用 Kotlin 开发 Android 应用的文档。
- 开发跨平台移动应用程序，请参见创建具有共享逻辑和原生 UI 的应用程序。

Kotlin/Wasm

> Kotlin/Wasm 处于 Alpha 阶段。 可能会随时更改。可以在生产环境之前使用。欢迎在 YouTrack 上提供反馈。
>
> 加入 Kotlin/Wasm 社区。
>
> 

Kotlin/Wasm has the power to compile your Kotlin code into WebAssembly (Wasm) format. With Kotlin/Wasm, you can create applications that run on different environments and devices, which support Wasm and meet Kotlin's requirements.

Wasm is a binary instruction format for a stack-based virtual machine. This format is platform-independent because it runs on its own virtual machine. Wasm provides Kotlin and other languages with a compilation target.

You can use Kotlin/Wasm in different target environments, such as browsers, for developing web applications built with Compose Multiplatform, or outside the browser in standalone Wasm virtual machines. In the outside-of-browser case, WebAssembly System Interface (WASI) provides access to platform APIs, which you can also utilize.

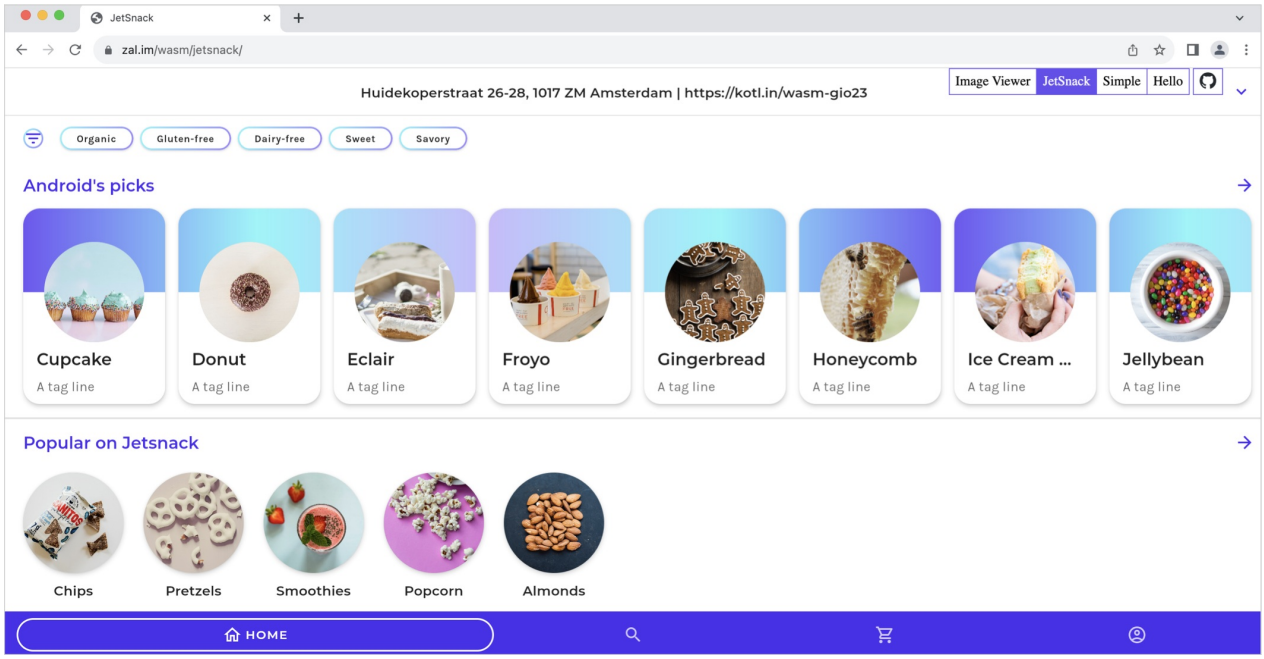
Kotlin/Wasm 与 Compose 多平台

With Kotlin, you have the power to build applications and reuse mobile and desktop user interfaces (UIs) in your web projects through Compose Multiplatform and Kotlin/Wasm.

Compose Multiplatform is a declarative framework based on Kotlin and Jetpack Compose that allows you to implement the UI once and share it across all the platforms you target.

For web platforms, Compose Multiplatform uses Kotlin/Wasm as its compilation target. Applications built with Kotlin/Wasm and Compose Multiplatform use a `wasm-js` target and run in browsers.

Explore our online demo of an application built with Compose Multiplatform and Kotlin/Wasm



> To run applications built with Kotlin/Wasm in a browser, you need a browser version that supports the new garbage collection and legacy exception handling proposals. To check the browser support status, see the WebAssembly roadmap.

Additionally, you can use the most popular Kotlin libraries in Kotlin/Wasm out of the box. Like in other Kotlin and Multiplatform projects, you can include dependency declarations in the build script. For more information, see Adding dependencies on multiplatform libraries.

Would you like to try it yourself?

Get started with Kotlin/Wasm →

Kotlin/Wasm 与 WASI

Kotlin/Wasm uses the WebAssembly System Interface (WASI) for server-side applications. Applications built with Kotlin/Wasm and WASI use a Wasm-WASI target, allowing you to call the WASI API and run applications outside the browser environment.

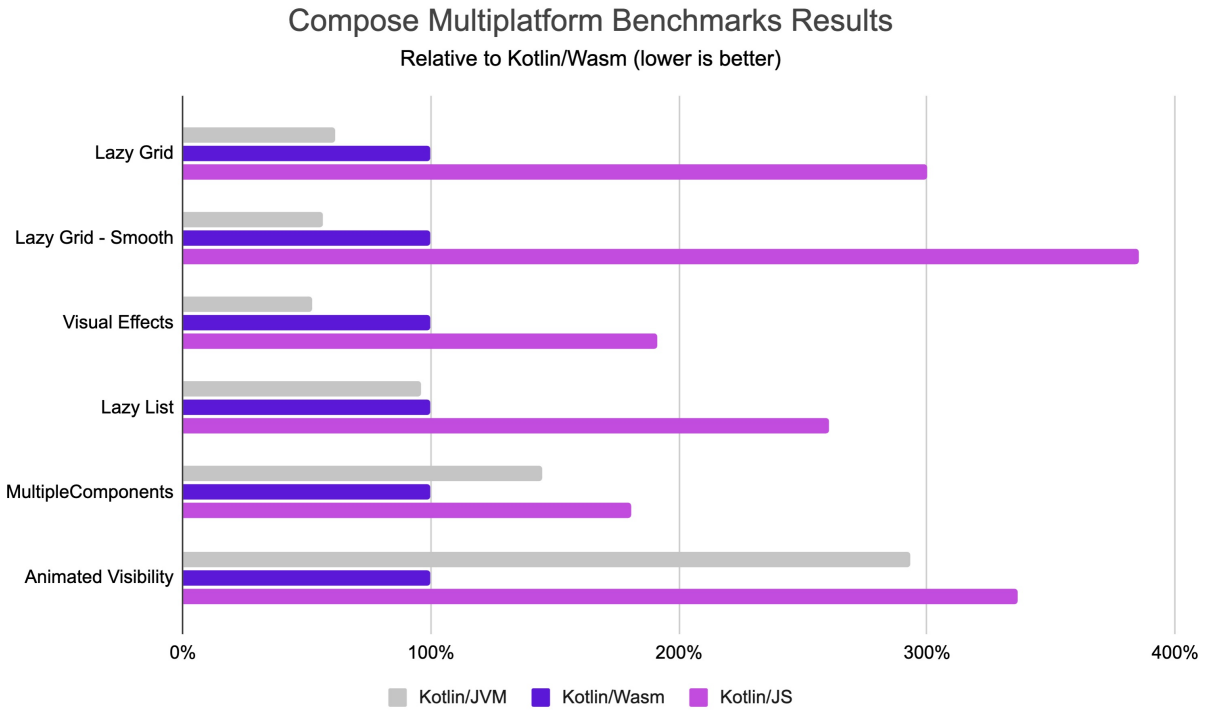
Kotlin/Wasm leverages WASI to abstract away platform-specific details, allowing the same Kotlin code to run across diverse platforms. This expands the reach of Kotlin/Wasm beyond web applications without requiring custom handling for each runtime.

WASI provides a secure standard interface for running Kotlin applications compiled to WebAssembly across different environments.

> To see Kotlin/Wasm and WASI in action, check the Get started with Kotlin/Wasm and WASI tutorial.

Kotlin/Wasm 性能

Although Kotlin/Wasm is still in Alpha, Compose Multiplatform running on Kotlin/Wasm already shows encouraging performance traits. You can see that its execution speed outperforms JavaScript and is approaching that of the JVM:



We regularly run benchmarks on Kotlin/Wasm, and these results come from our testing in a recent version of Google Chrome.

浏览器 API 支持

The Kotlin/Wasm standard library provides declarations for browser APIs, including the DOM API. With these declarations, you can directly use the Kotlin API to access and utilize various browser functionalities. For example, in your Kotlin/Wasm applications, you can use manipulation with DOM elements or fetch the API without defining these declarations from scratch. To learn more, see our Kotlin/Wasm browser example.

The declarations for browser API support are defined using JavaScript interoperability capabilities. You can use the same capabilities to define your own declarations. In addition, Kotlin/Wasm–JavaScript interoperability allows you to use Kotlin code from JavaScript. For more information, see Use Kotlin code in JavaScript.

敬请反馈

Kotlin/Wasm 反馈

- Slack: Get a Slack invite and provide your feedback directly to developers in our #webassembly channel.
- Report any issues in YouTrack.

Compose 多平台反馈

- Slack: provide your feedback in the #compose-web public channel.
- Report any issues in GitHub.

了解更多

- Learn more about Kotlin/Wasm in this YouTube playlist.
- Explore the Kotlin/Wasm examples in our GitHub repository.

Kotlin/Native

Kotlin/Native 是一种将 Kotlin 代码编译为无需虚拟机就可运行的原生二进制文件的技术。Kotlin/Native 包含一个基于 LLVM 的 Kotlin 编译器后端以及Kotlin 标准库的原生实现。

为什么选用 Kotlin/Native？

Kotlin/Native 的主要设计目标是让 Kotlin 可以为不希望或者不可能使用虚拟机的平台（例如嵌入式设备或者 iOS）编译。它非常适合当你需要生成无需额外运行时或虚拟机的自包含程序的情况。

It's easy to include compiled Kotlin code in existing projects written in C, C++, Swift, Objective-C, and other languages. You can also use existing native code, static or dynamic C libraries, Swift/Objective-C frameworks, graphical engines, and anything else directly from Kotlin/Native.

Get started with Kotlin/Native

目标平台

Kotlin/Native 支持以下平台：

- Linux
- Windows（通过 MinGW）
- Android NDK
- Apple 目标平台，面向 macOS、 iOS、 tvOS 与 watchOS

> 如需编译苹果目标，需要安装 Xcode 及其命令行工具。
>
> 

参见所支持目标的完整列表。

互操作

Kotlin/Native 支持与不同操作系统的原生编程语言的双向互操作。 The compiler can create executables for many platforms, static or dynamic C libraries, and Swift/Objective-C frameworks.

与 C 语言互操作

Kotlin/Native provides interoperability with C. You can use existing C libraries directly from Kotlin code.

To learn more, complete the following tutorials:

- Create a dynamic library with C headers for C/C++ projects
- Learn how C types are mapped into Kotlin
- Create a native HTTP client using C interop and libcurl

与 Swift/Objective-C 互操作

Kotlin/Native provides interoperability with Swift through Objective-C. You can use Kotlin code directly from Swift/Objective-C applications on macOS and iOS.

To learn more, complete the Kotlin/Native as an Apple framework tutorial.

在多个平台之间共享代码

Kotlin/Native includes a set of prebuilt platform libraries that help share Kotlin code between projects. POSIX, gzip, OpenGL, Metal, Foundation, and many other popular libraries and Apple frameworks are pre-imported and included as Kotlin/Native libraries in the compiler package.

Kotlin/Native is a part of the Kotlin Multiplatform technology that helps share common code across multiple platforms, including Android, iOS, JVM, web, and native. Multiplatform libraries provide the necessary APIs for common Kotlin code and allow writing shared parts of projects in Kotlin all in one place.

内存管理器

Kotlin/Native uses an automatic memory manager that is similar to the JVM and Go. It has its own tracing garbage collector, which is also integrated with Swift/Objective-C's ARC.

The memory consumption is controlled by a custom memory allocator. It optimizes memory usage and helps prevent sudden surges in memory allocation.

Kotlin 用于 JavaScript 开发

Kotlin/JS 提供了转换 Kotlin 代码、Kotlin 标准库的能力，并且兼容 JavaScript 的任何依赖项。Kotlin/JS 的当前实现以 ES5 为目标。

使用 Kotlin/JS 的推荐方法是通过 `kotlin.multiplatform` Gradle 插件。它提供了一种集中且便捷的方式来设置与控制以 JavaScript 为目标的 Kotlin 项目。 这包括基本特性， 例如控制应用程序的捆绑，直接从 npm 添加 JavaScript 依赖项等等。要获得可用选项的概述，请查看搭建 Kotlin/JS 项目。

Kotlin/JS IR 编译器

Kotlin/JS IR 编译器相对于旧版默认编译器进行了许多改进。 例如，通过消除死代码来减小生成的可执行文件的体积， 并提供了与 JavaScript 生态系统及其工具更加流畅的互操作性。

> 自 Kotlin 1.8.0 版起，旧编译器已弃用。
>
> 

通过从 Kotlin 代码生成 TypeScript 声明文件（`d.ts`），IR 编译器使创建混合 TypeScript 与 Kotlin 代码的“混合” 应用程序变得更加容易，并利用 Kotlin 多平台代码共享功能。

如需了解关于 Kotlin/JS IR 编译器中可用特性的更多信息，以及如何在项目中尝试使用它，请访问 Kotlin/JS IR 编译器文档页及其迁移指南。

Kotlin/JS 框架

Modern web development benefits significantly from frameworks that simplify building web applications. Here are a few examples of popular web frameworks for Kotlin/JS written by different authors:

Kobweb

Kobweb is an opinionated Kotlin framework for creating websites and web apps. It leverages Compose HTML and live-reloading for fast development. Inspired by Next.js, Kobweb promotes a standard structure for adding widgets, layouts, and pages.

Out of the box, Kobweb provides page routing, light/dark mode, CSS styling, Markdown support, backend APIs, and more features. It also includes a UI library called Silk, a set of versatile widgets for modern UIs.

Kobweb also supports site export, generating page snapshots for SEO and automatic search indexing. Additionally, Kobweb makes it easy to create DOM-based UIs that efficiently update in response to state changes.

Visit the Kobweb site for documentation and examples.

For updates and discussions about the framework, join the #kobweb and #compose-web channels in the Kotlin Slack.

KVision

KVision is an object-oriented web framework that makes it possible to write applications in Kotlin/JS with ready-to-use components that can be used as building blocks for your application's user interface. You can use both reactive and imperative programming models to build your frontend, use connectors for Ktor, Spring Boot, and other frameworks to integrate it with your server-side applications, and share code using Kotlin Multiplatform.

Visit KVision site for documentation, tutorials, and examples.

For updates and discussions about the framework, join the #kvision and #javascript channels in the Kotlin Slack.

fritz2

fritz2 is a standalone framework for building reactive web user interfaces. It provides its own type-safe DSL for building and rendering HTML elements, and it makes use of Kotlin's coroutines and flows to express components and their data bindings. It provides state management, validation, routing, and more out of the box, and integrates with Kotlin Multiplatform projects.

Visit fritz2 site for documentation, tutorials, and examples.

For updates and discussions about the framework, join the #fritz2 and #javascript channels in the Kotlin Slack.

Doodle

Doodle is a vector-based UI framework for Kotlin/JS. Doodle applications use the browser's graphics capabilities to draw user interfaces instead of relying on DOM, CSS, or Javascript. By using this approach, Doodle gives you precise control over the rendering of arbitrary UI elements, vector shapes, gradients, and custom visualizations.

Visit Doodle site for documentation, tutorials, and examples.

For updates and discussions about the framework, join the #doodle and #javascript channels in the Kotlin Slack.

加入 Kotlin/JS 社区

可以在官方 Kotlin Slack 中加入 #javascript 频道， 同社区与团队交谈。

Kotlin 用于数据分析

Exploring and analyzing data is something you may not do every day, but it's a crucial skill you need as a software developer.

Let's think about software development duties where data analysis is key: analyzing what's actually inside collections when debugging, digging into memory dumps or databases, or receiving JSON files with large amounts of data when working with REST APIs, to mention some.

With Kotlin's Exploratory Data Analysis (EDA) tools, such as Kotlin notebooks, Kotlin DataFrame, and Kandy, you have at your disposal a rich set of capabilities to enhance your analytics skills and support you across different scenarios:

- **Load, transform, and visualize data in various formats:** with our Kotlin EDA tools, you can perform tasks like filtering, sorting, and aggregating data. Our tools can seamlessly read data right in the IDE from different file formats, including CSV, JSON, and TXT.

  Kandy, our plotting tool, allows you to create a wide range of charts to visualize and gain insights from the dataset.

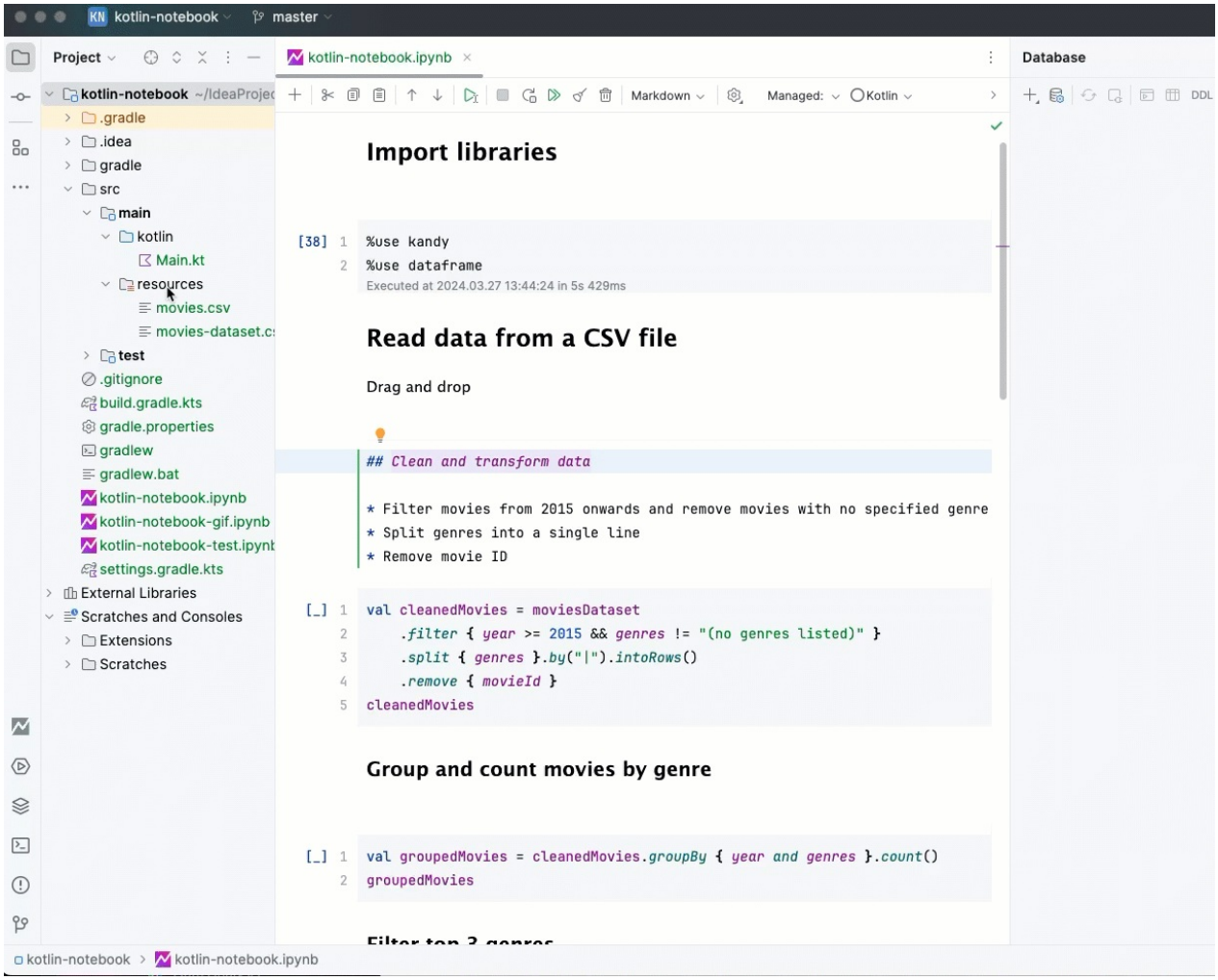
- **Efficiently analyze data stored in relational databases:** Kotlin DataFrame seamlessly integrates with databases and provides capabilities similar to SQL queries. You can retrieve, manipulate, and visualize data directly from various databases.

- **Fetch and analyze real-time and dynamic datasets from web APIs:** the EDA tools' flexibility allows integration with external APIs via protocols like OpenAPI. This feature helps you fetch data from web APIs, to then clean and transform the data to your needs.

Would you like to try our Kotlin tools for data analysis?

Get started with Kotlin Notebook

Our Kotlin data analysis tools let you smoothly handle your data from start to finish. Effortlessly retrieve your data with simple drag-and-drop functionality in our Kotlin Notebook. Clean, transform, and visualize it with just a few lines of code. Additionally, export your output charts in a matter of clicks.



Notebook

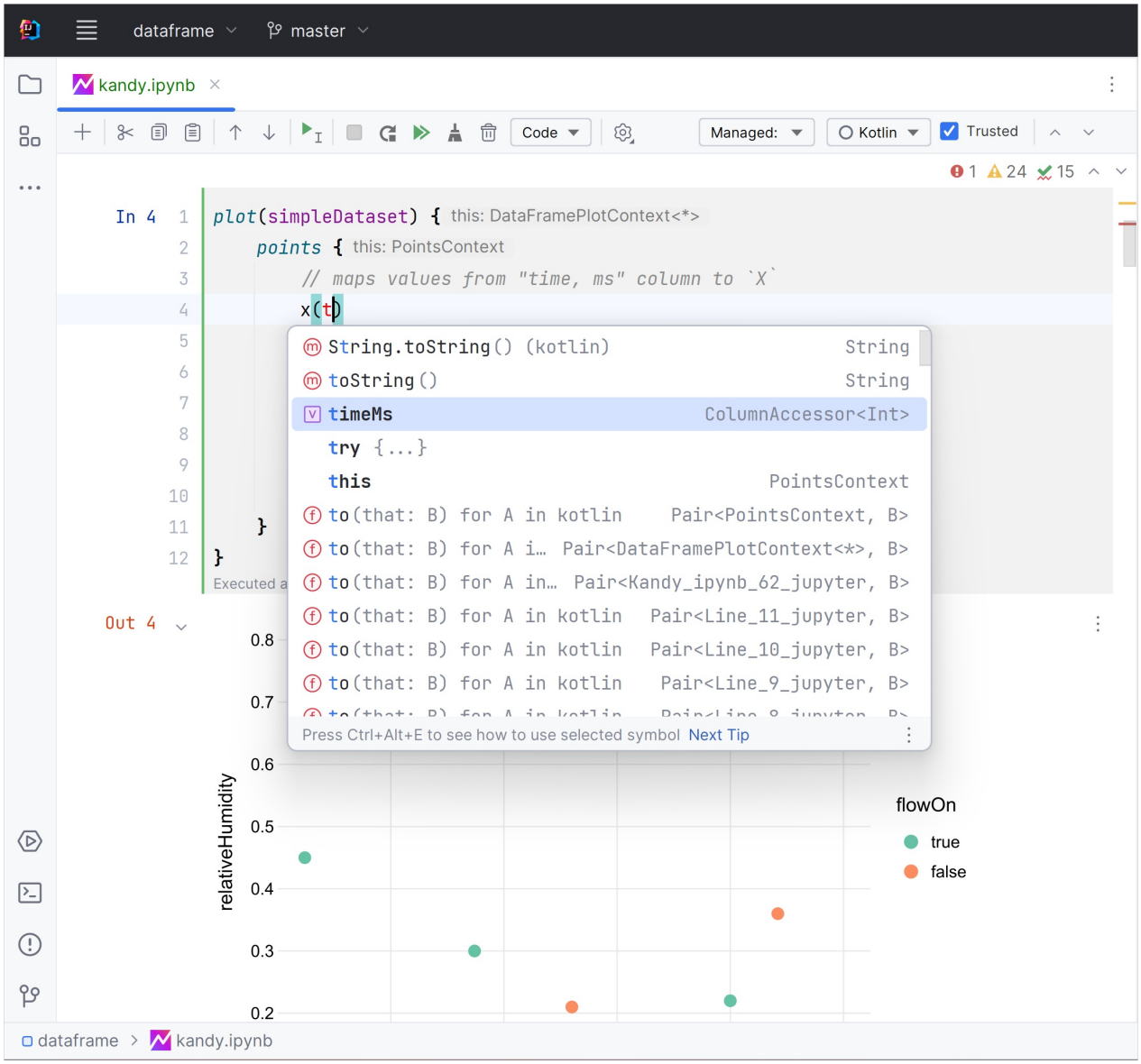
Notebooks are interactive editors that integrate code, graphics, and text in a single environment. When using a notebook, you can run code cells and immediately see the output.

Kotlin offers different notebook solutions, such as Kotlin Notebook, Datalore, and Kotlin-Jupyter Notebook, providing convenient features for data retrieving, transformation, exploration, modeling, and more. These Kotlin notebook solutions are based on our Kotlin Kernel.

You can seamlessly share your code among Kotlin Notebook, Datalore, and Kotlin-Jupyter Notebook. Create a project in one of our Kotlin notebooks and continue working in another notebook without compatibility issues.

Benefit from the features of our powerful Kotlin notebooks and the perks of coding with Kotlin. Kotlin integrates with these notebooks to help you manage data and share your findings with colleagues while building up your data science and machine learning skills.

Discover the features of our different Kotlin notebook solutions and choose the one that best aligns with your project requirements.



Kotlin Notebook

The Kotlin Notebook is a plugin for IntelliJ IDEA that allows you to create notebooks in Kotlin. It provides our IDE experience with all common IDE features, offering real-time code insights and project integration.

Datalore 中的 Kotlin notebook

With Datalore, you can use Kotlin in the browser straight out of the box without additional installation. You can also share your notebooks and run them remotely, collaborate with other Kotlin notebooks in real-time, receive smart coding assistance as you write code, and export results through interactive or static reports.

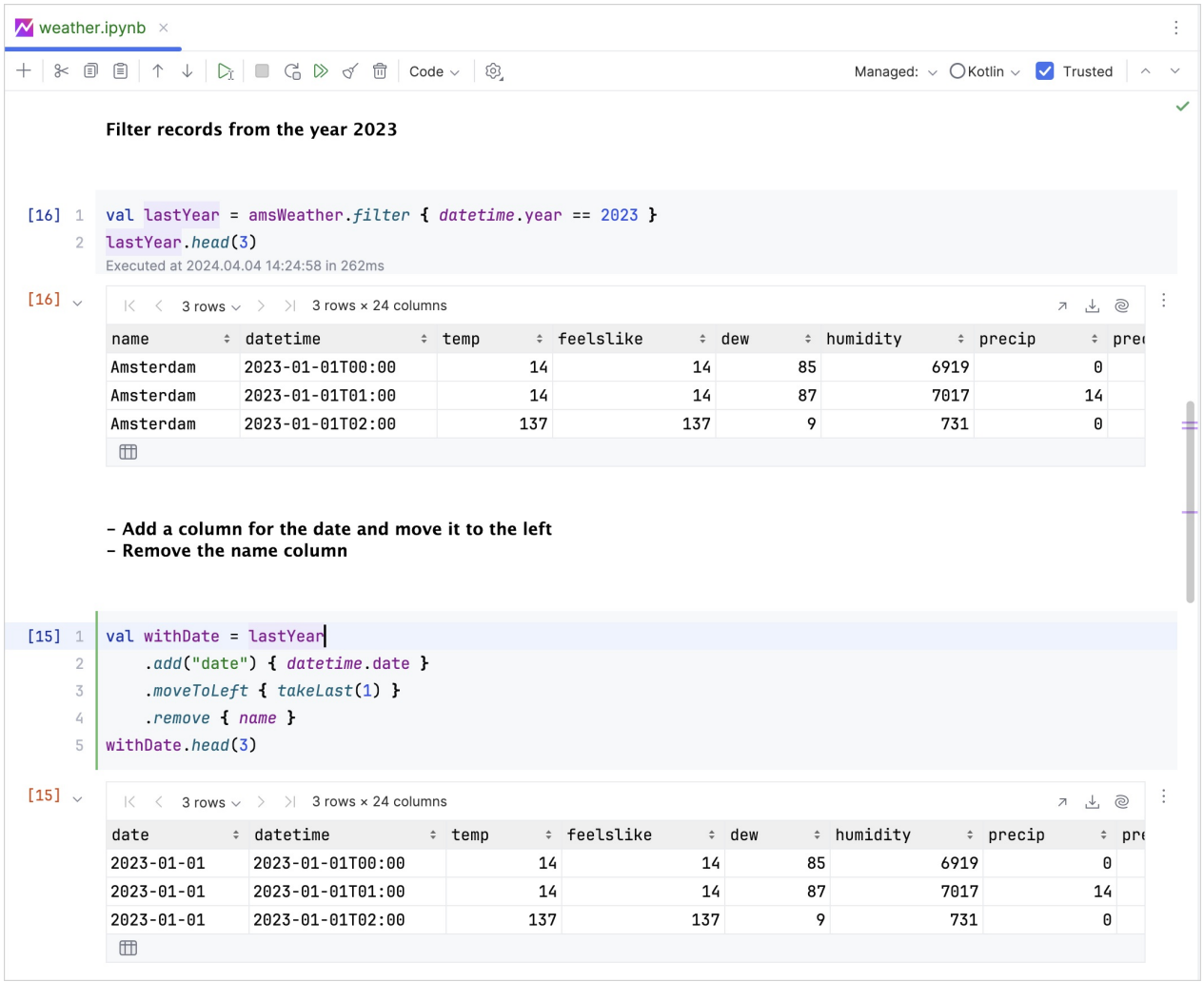
带 Kotlin 内核的 Jupyter Notebook

Jupyter Notebook is an open-source web application that allows you to create and share documents containing code, visualizations, and Markdown text. Kotlin-Jupyter is an open-source project that brings Kotlin support to Jupyter Notebook to harness Kotlin's power within the Jupyter environment.

Kotlin DataFrame

The Kotlin DataFrame library lets you manipulate structured data in your Kotlin projects. From data creation and cleaning to in-depth analysis and feature engineering, this library has you covered.

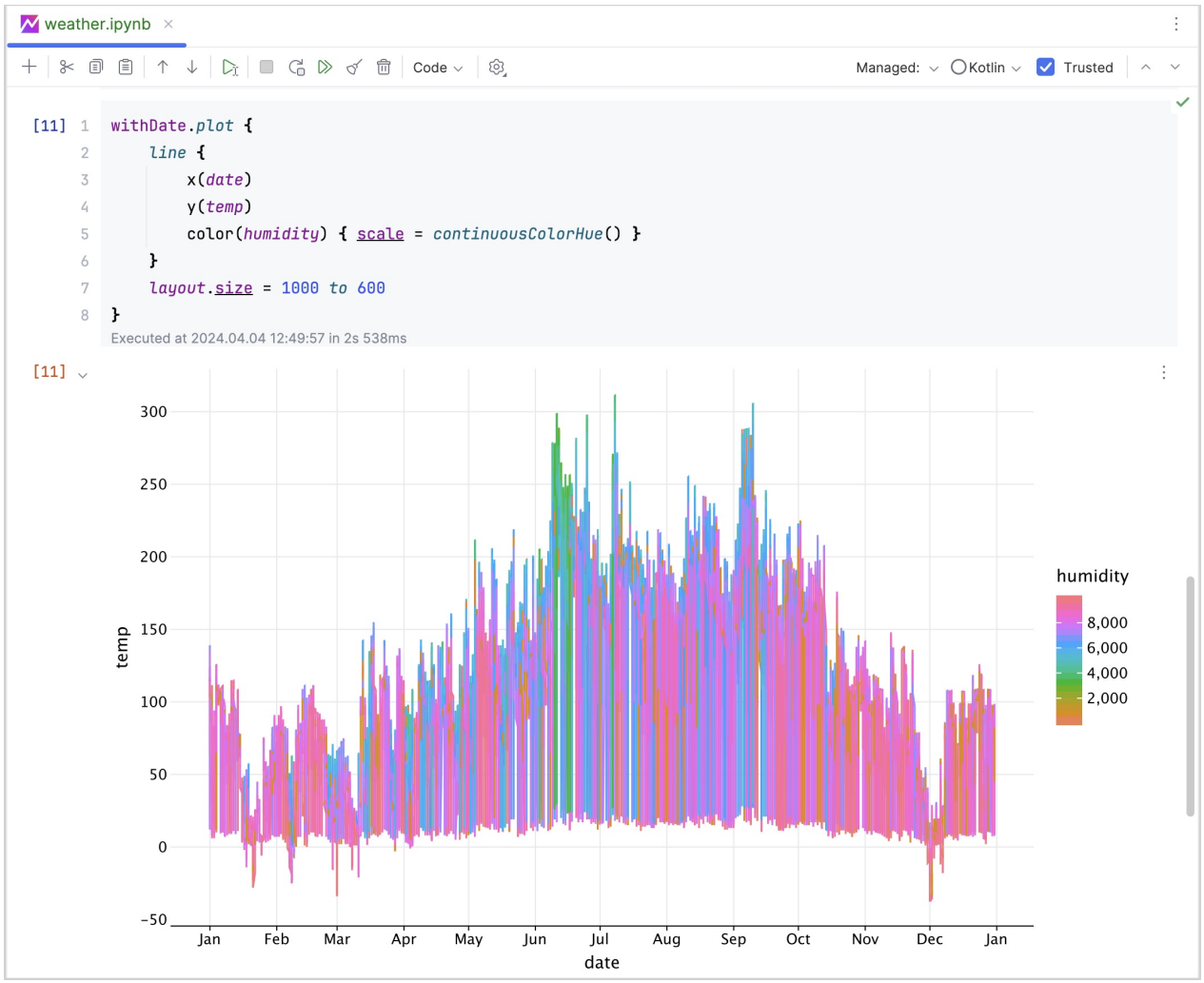
With the Kotlin DataFrame library, you can work with different file formats, including CSV, JSON, XLS, and XLSX. This library also facilitates the data retrieval process with its ability to connect with SQL databases or APIs.



Kandy

Kandy is an open-source Kotlin library that provides a powerful and flexible DSL for plotting charts of various types. This library is a simple, idiomatic, readable, and type-safe tool to visualize data.

Kandy has seamless integration with Kotlin Notebook, Datalore, and Kotlin-Jupyter Notebook. You can also easily combine the Kandy and Kotlin DataFrame libraries to complete different data-related tasks.



下一步做什么

- Get started with Kotlin Notebook
- Retrieve and transform data using the Kotlin DataFrame library
- Visualize data using the Kandy library
- Learn more about Kotlin and Java libraries for data analysis

Kotlin 用于 AI 驱动的应用开发

Kotlin 为 AI 驱动的应用程序提供了现代且实用的基础。

它可以跨平台使用，与现有的 AI 框架完美集成，并支持常见的 AI 开发模式。

> 本页通过 Kotlin-AI-Examples 版本库中的工作示例介绍了如何在现实世界的 AI 场景中使用 Kotlin。
>
> 

Kotlin AI 代理化框架——Koog

Koog 是一个基于 Kotlin 的框架，用于在本地创建及运行 AI 代理而需外部服务。 Koog 是 JetBrains 的创新开源代理框架，支持开发者在 JVM 生态系统中构建 AI 代理。 它为构建能够与工具交互、处理复杂工作流以及与用户沟通的智能代理提供了纯 Kotlin 的实现。

更多使用场景

Kotlin 可以助力 AI 开发的其他使用场景还有很多。 从将语言模型集成到后端服务，到构建 AI 驱动的用户界面， 这些示例展示了 Kotlin 在各种 AI 应用程序中的多功能性。

检索增强生成

使用 Kotlin 构建检索增强生成（RAG）流水线，将语言模型连接到外部源，如文档、向量存储或 API。 例如：

- `springAI-demo`：一个 Spring Boot 应用，将 Kotlin 标准库文档加载到矢量存储中并支持基于文档的问答。
- `langchain4j-spring-boot`：使用 LangChain4j 的一个最小 RAG 示例。

基于代理的应用程序

使用 Kotlin 构建 AI 代理，使用语言模型与工具进行推理、规划与行动。 例如：

- `koog`：展示如何使用 Kotlin 代理框架 Koog 构建 AI 代理。
- `langchain4j-spring-boot`：包含了一个使用 LangChain4j 构建简单的工具使用代理。

思维链提示

实现结构化的提示技术，通过多步推理来指导语言模型。 例如：

- `LangChain4j_Overview.ipynb`：演示思维链与结构化输出的 Kotlin Notebook。

后端服务中的 LLM

使用 Kotlin 与 Spring 将 LLM 集成到业务逻辑或 REST API 中。 例如：

- `spring-ai-examples`：包含分类、聊天与总结示例。
- `springAI-demo`：演示 LLM 响应与应用程序逻辑的完整集成。

具有 AI 的多平台用户界面

使用 Compose 多平台在 Kotlin 中构建交互式 AI 驱动的 UI。 例如：

- `mcp-demo`：连接到 Claude 与 OpenAI 并使用 Compose 多平台呈现响应的桌面 UI。

探索示例

可以在 Kotlin-AI-Examples 版本库中探索并运行示例。 每个项目都是独立的。可以将每个项目用作构建基于 Kotlin 的 AI 应用程序的参考或模板。

下一步做什么

- 完成构建一个使用 Spring AI 根据 Qdrant 中存储的文档回答问题的 Kotlin 应用教程，以了解关于在 IntelliJ IDEA 中结合使用 Spring AI 与 Kotlin 的更多信息
- 加入 Kotlin 社区，与其他使用 Kotlin 构建 AI 应用程序的开发者建立联系

Kotlin 用于竞技程序设计

本教程适用于之前未使用 Kotlin 的竞技程序员，也适用于之前从未参与过任何竞技性程序设计活动的 Kotlin 开发人员。 本教程假定读者具有相应的编程技能。

竞技性程序设计是一项智力运动，参赛选手在严格的限制条件下编写程序精确地解决指定的算法问题。问题可以简单到任何软件开发人员都能解题、只需很少代码就能得到正确答案，也可以复杂到需要特殊的算法、数据结构知识以及大量实践。虽然 Kotlin 不是专为竞技性编程而设计的，但是它恰好适合这一领域，显著减少了程序员所需编写与阅读的样板代码量，这样几乎可以像动态脚本语言一样编写代码，同时又有静态类型语言的工具与性能支持。

关于如何搭建 Kotlin 开发环境，请参见 Kotlin/JVM 入门。在竞技程序设计中，通常会创建单个项目，而每个问题的答案写在单个源文件中。

简单示例：可达数问题

我们来看一个具体的示例。

Codeforces 第 555 轮第 3 次分赛已于 4 月 26 日举行，意味着它有适合任何开发者尝试的问题。 可以打开这个链接来阅读问题。 这组问题中最简单的是问题 A：可达数。 它要求实现问题陈述中所描述的简单算法。

我们会通过创建一个任意名称的 Kotlin 源文件来解这个问题。 `A.kt` 就挺好。 首先，需要实现问题陈述中指定的函数，如下：

我们以这样的方式来表示函数 f(x)：将 x 加 1，然后，如果得到的数至少以一个零结尾， 就去掉这个零。

Kotlin 是一门实用且不拘一格的语言，既支持命令式也支持函数式编程风格， 而不会将开发人员推向任何一种风格。可以按函数式风格实现函数 `f`，使用像尾递归这样的 Kotlin 特性：

```
tailrec fun removeZeroes(x: Int): Int =
    if (x % 10 == 0) removeZeroes(x / 10) else x

fun f(x: Int) = removeZeroes(x + 1)
```

也可以编写函数 `f` 的命令式实现，使用传统的 while 循环与可变变量（在 Kotlin 中用 var 表示）：

```
fun f(x: Int): Int {
    var cur = x + 1
    while (cur % 10 == 0) cur /= 10
```

```
    return cur
}
```

由于普遍使用类型推断，在 Kotlin 中很多地方的类型都是可选的，不过每个声明仍然具有编译期已知的明确定义的静态类型。

现在就只剩编写读取输入的主函数并实现问题陈述所要求算法的其余部分——计算在对标准输入所给出的初始数 `n` 重复应用函数 `f` 时所产生的不同整数的个数。

默认情况下，Kotlin 在 JVM 上运行，可以直接访问丰富且高效的集合库，其中包含通用的集合与数据结构，如动态大小的数组（`ArrayList`）、基于哈希的 map 与 set（`HashMap` / `HashSet`）、基于树的 map 与 set（`TreeMap` / `TreeSet`）。使用整数哈希 set 来跟踪应用函数 `f` 时已达到的值，该问题解法的一个简单命令式版本可以这样编写：

> 【Kotlin 1.6.0 及更高版本】

```
fun main() {
    var n = readln().toInt() // 读取输入的整数
    val reached = HashSet<Int>() // 可变的哈希 set
    while (reached.add(n)) n = f(n) // 迭代函数 f
    println(reached.size) // 输出答案
}
```

在竞技程序设计中无需处理输入格式错误的情况。竞技程序设计中的输入格式向来都是精确指定的，并且实际输入不能偏离问题陈述中的输入规范。That's why you can use Kotlin's `readln()` function. 它断言输入的字符串存在，如不存在则抛出异常。同样，如果输入不是整数，那么 `String.toInt()` 函数会抛出异常。

> 【早期版本】

```
fun main() {
    var n = readLine()!!.toInt() // 读取输入的整数
    val reached = HashSet<Int>() // 可变的哈希 set
    while (reached.add(n)) n = f(n) // 迭代函数 f
    println(reached.size) // 输出答案
}
```

请注意 readLine() 函数调用之后的空断言操作符 `!!` 的使用。Kotlin 的 `readLine()` 函数定义为返回可空类型 `String?` 并且在输入结束时返回 `null`，这样显式迫使开发人员处理缺少输入的情况。

在竞技程序设计中无需处理输入格式错误的情况。 在竞技程序设计中，输入格式总是精确指定并且实际输入不能偏离问题陈述中的输入规范。 这就是空断言操作符 `!!` 本质上所做的——它断言输入字符串存在否则抛出异常。 同理， String.toInt() 也是这样。

所有在线竞技程序设计活动都允许使用预编写代码，因此可以定义自己的面向竞技程序设计的工具函数库，以使实际解题代码更易于读写。然后，可以使用该代码作为解题模板。例如，可以定义以下辅助函数来读取竞技程序设计中的输入：

> 【Kotlin 1.6.0 及更高版本】

```
private fun readStr() = readln() // 一行字符串
private fun readInt() = readStr().toInt() // 单个整数
// 用于在解题中会用到的其他类型的类似声明等
```

> 【早期版本】

```
private fun readStr() = readLine()!! // 一行字符串
private fun readInt() = readStr().toInt() // 单个整数
// 用于在解题中会用到的其他类型的类似声明等
```

请注意这里使用了 `private`（私有）可见修饰符。 虽然可见性修饰符的概念与竞技程序设计并无瓜葛， 但是它让你能够将基于相同模板的多个解题文件放在同一包中，而不会出现公有声明冲突的报错。

函数式操作示例：长数问题

对于更复杂的问题，Kotlin 丰富的集合函数式操作库就派上用场了， 可以大幅减少模板代码，并将代码写成从上到下、从左到右的流式数据转换流水线。例如问题 B：长数问题用一个简单的贪心算法实现，可以采用这种风格编写而无需任何可变变量：

> 【Kotlin 1.6.0 及更高版本】

```
fun main() {
    // 读取输入
```

```
    val n = readln().toInt()
    val s = readln()
    val fl = readln().split(" ").map { it.toInt() }
    // 定义局部函数 f
    fun f(c: Char) = '0' + fl[c - '1']
    // 贪婪查找第一个与最后一个索引
    val i = s.indexOfFirst { c -> f(c) > c }
        .takeIf { it >= 0 } ?: s.length
    val j = s.withIndex().indexOfFirst { (j, c) -> j > i && f(c) < c }
        .takeIf { it >= 0 } ?: s.length
    // 组合并写出答案
    val ans =
        s.substring(0, i) +
        s.substring(i, j).map { c -> f(c) }.joinToString("") +
        s.substring(j)
    println(ans)
}
```

> 【早期版本】

```
fun main() {
    // 读取输入
    val n = readLine()!!.toInt()
    val s = readLine()!!
    val fl = readLine()!!.split(" ").map { it.toInt() }
    // 定义局部函数 f
    fun f(c: Char) = '0' + fl[c - '1']
    // 贪婪查找第一个与最后一个索引
    val i = s.indexOfFirst { c -> f(c) > c }
        .takeIf { it >= 0 } ?: s.length
    val j = s.withIndex().indexOfFirst { (j, c) -> j > i && f(c) < c }
        .takeIf { it >= 0 } ?: s.length
    // 组合并写出答案
    val ans =
        s.substring(0, i) +
        s.substring(i, j).map { c -> f(c) }.joinToString("") +
        s.substring(j)
    println(ans)
}
```

在这段密集的代码中，除了集合转换之外，还可以看到像局部函数以及 elvis 操作符 `?:` 这样灵便的 Kotlin 特性， 通过 elvis 操作符，可以用简洁易读的表达式如 `.takeIf { it >= 0 } ?: s.length` 来表达类似“如果是正数就取其值，否则取长度”的惯用法， 当然，也完全可以使用 Kotlin 创建额外的可变变量并以命令式风格表达相同的代码。

为了让这种竞技程序设计任务中读取输入更简洁， 可以使用以下输入读取辅助函数列表：

> 【Kotlin 1.6.0 及更高版本】

```
private fun readStr() = readln() // 一行字符串
private fun readInt() = readStr().toInt() // 单个整数
private fun readStrings() = readStr().split(" ") // 字符串列表
private fun readInts() = readStrings().map { it.toInt() } // 整数列表
```

> 【早期版本】

```
private fun readStr() = readLine()!! // 一行字符串
private fun readInt() = readStr().toInt() // 单个整数
private fun readStrings() = readStr().split(" ") // 字符串列表
private fun readInts() = readStrings().map { it.toInt() } // 整数列表
```

有了这些辅助函数，读取输入的代码部分变得更简单，一行行地严格遵循问题陈述中的输入规范：

```
// 读取输入
val n = readInt()
val s = readStr()
val fl = readInts()
```

请注意，在竞技程序设计中，习惯给变量取比通常在工业编程实践中更短的名称，因为代码只需编写一次，以后就不用支持了。 当然，这些名称通常仍然是助记手段——数组用 `a`，索引用 `i`、`j`，表格的行列号用 `r`、`c`，坐标用 `x`、`y` 等。 输入数据的名称保持与问题陈述中所给出的名称相同也更容易。 当然，越复杂的问题就越需要更多的代码来解，这导致更长、具有自解释性的变量名与函数名。

更多提示与技巧

竞技程序设计通常有这样的输入：

输入的第一行包含两个整数 `n` 与 `k`

在 Kotlin 中，这一行可以通过使用对整型列表进行解构声明的下列语句简明地解析：

```
val (n, k) = readInts()
```

很多人习惯使用 JVM 的 `java.util.Scanner` 类来解析结构较少的输入格式。Kotlin 已设计成能与 JVM 库很好互操作，因此在 Kotlin 中使用它们会很自然。然而请注意，`java.util.Scanner` 极其慢。事实上，速度慢得以至用它解析 10^5 个或更多整数时，很可能不满足典型的 2 秒限制，而这是一个简单的 Kotlin `split(" ").map { it.toInt() }` 就能做到的。

在 Kotlin 中写输出通常很简单，调用 println(...) 以及使用 Kotlin 的字符串模板。然而，当输出包含大约 10^5 或更多行时，必须小心。调用这么多次 `println` 太慢了，因为 Kotlin 中的该输出会在每行后自动刷新写缓冲。 从数组或 list 中写多行的更快方式是使用 joinToString() 函数以 `"\n"` 作为分隔符，如下所示：

```
println(a.joinToString("\n")) // 数组/list 中的每个元素占一行
```

学习 Kotlin

Kotlin 很易学，尤其对于已经了解 Java 的人。 对于软件开发人员来说，关于 Kotlin 基本语法的简短介绍可以直接在以基本语法开始的网站参考部分中直接找到。

IDEA 已内置 Java-to-Kotlin 转换器。 熟悉 Java 的人可以用它来学习相应的 Kotlin 语法结构，但它并不完美，并且你仍然需要自己熟悉 Kotlin 并学习 Kotlin 惯用法。

学习 Kotlin 语法以及 Kotlin 标准库 API 的一个很好的资源是 Kotlin 心印。

Kotlin 的新特性

- Kotlin 2.1.20
- Kotlin 2.1.0
- Kotlin 2.2.0-RC
- 早期版本
  - Kotlin 2.0.20
  - Kotlin 2.0.0
  - Kotlin 1.9.20
  - Kotlin 1.9.0
  - Kotlin 1.8.20
  - Kotlin 1.8.0
  - Kotlin 1.7.20
  - Kotlin 1.7.0
  - Kotlin 1.6.20
  - Kotlin 1.6.0
  - Kotlin 1.5.30
  - Kotlin 1.5.20
  - Kotlin 1.5.0
  - Kotlin 1.4.30
  - Kotlin 1.4.20
  - Kotlin 1.4.0
  - Kotlin 1.3
  - Kotlin 1.2
  - Kotlin 1.1

Kotlin 2.1.20 的新特性

发布于：*2025-03-20*

The Kotlin 2.1.20 release is here! Here are the main highlights:

- **K2 compiler updates**: updates to the new kapt and Lombok plugins
- **Kotlin Multiplatform**: new DSL to replace Gradle's Application plugin
- **Kotlin/Native**: support for Xcode 16.3 and a new inlining optimization
- **Kotlin/Wasm**: default custom formatters, support for DWARF, and migration to Provider API
- **Gradle support**: compatibility with Gradle's Isolated Projects and custom publication variants
- **Standard library**: common atomic types, improved UUID support, and new time-tracking functionality
- **Compose compiler**: relaxed restrictions on `@Composable` functions and other updates
- **Documentation**: notable improvements to the Kotlin documentation.

IDE support

The Kotlin plugins that support 2.1.20 are bundled in the latest IntelliJ IDEA and Android Studio. You don't need to update the Kotlin plugin in your IDE. All you need to do is to change the Kotlin version to 2.1.20 in your build scripts.

See Update to a new release for details.

Download sources for Kotlin artifacts in projects with OSGi support

Sources of all dependencies of the `kotlin-osgi-bundle` library are now included in its distribution. This allows IntelliJ IDEA to download these sources to provide documentation for Kotlin symbols and improve the debugging experience.

Kotlin K2 compiler

We're continuing to improve plugin support for the new Kotlin K2 compiler. This release brings updates to the new kapt and Lombok plugins.

New default kapt plugin

Starting with Kotlin 2.1.20, the K2 implementation of the kapt compiler plugin is enabled by default for all the projects.

The JetBrains team launched the new implementation of the kapt plugin with the K2 compiler back in Kotlin 1.9.20. Since then, we have further developed the internal implementation of K2 kapt and made its behavior similar to that of the K1 version, while significantly improving its performance as well.

If you encounter any issues when using kapt with the K2 compiler, you can temporarily revert to the previous plugin implementation.

To do this, add the following option to the `gradle.properties` file of your project:

```
kapt.use.k2=false
```

Please report any issues to our issue tracker.

Lombok compiler plugin: support for `@SuperBuilder` and updates on `@Builder`

The Kotlin Lombok compiler plugin now supports the `@SuperBuilder` annotation, making it easier to create builders for class hierarchies. Previously, developers using Lombok in Kotlin had to manually define builders when working with inheritance. With `@SuperBuilder`, the builder automatically inherits superclass fields, allowing you to initialize them when constructing an object.

Additionally, this update includes several improvements and bug fixes:

- The `@Builder` annotation now works on constructors, allowing more flexible object creation. For more details, see the corresponding YouTrack issue.
- Several issues related to Lombok's code generation in Kotlin have been resolved, improving overall compatibility. For more details, see the GitHub changelog.

For more information about the `@SuperBuilder` annotation, see the official Lombok documentation.

Kotlin Multiplatform: new DSL to replace Gradle's Application plugin

Starting with Gradle 8.7, the Application plugin is no longer compatible with the Kotlin Multiplatform Gradle plugin. Kotlin 2.1.20 introduces an Experimental DSL to achieve similar functionality. The new `executable {}` block configures execution tasks and Gradle distributions for JVM targets.

Before the `executable {}` block in your build script, add the following `@OptIn` annotation:

```
@OptIn(ExperimentalKotlinGradlePluginApi::class)
```

For example:

```
kotlin {
    jvm {
        @OptIn(ExperimentalKotlinGradlePluginApi::class)
        binaries {
            // Configures a JavaExec task named "runJvm" and a Gradle distribution for
 the "main" compilation in this target
            executable {
                mainClass.set("foo.MainKt")
            }

            // Configures a JavaExec task named "runJvmAnother" and a Gradle distribut
ion for the "main" compilation
            executable(KotlinCompilation.MAIN_COMPILATION_NAME, "another") {
                // Set a different class
                mainClass.set("foo.MainAnotherKt")
            }

            // Configures a JavaExec task named "runJvmTest" and a Gradle distribution
 for the "test" compilation
            executable(KotlinCompilation.TEST_COMPILATION_NAME) {
                mainClass.set("foo.MainTestKt")
            }

            // Configures a JavaExec task named "runJvmTestAnother" and a Gradle distr
ibution for the "test" compilation
            executable(KotlinCompilation.TEST_COMPILATION_NAME, "another") {
                mainClass.set("foo.MainAnotherTestKt")
            }
        }
    }
}
```

In this example, Gradle's Distribution plugin is applied on the first `executable {}` block.

If you run into any issues, report them in our issue tracker or let us know in our public Slack channel.

Kotlin/Native

Support for Xcode 16.3

Starting with Kotlin **2.1.21**, the Kotlin/Native compiler supports Xcode 16.3 – the latest stable version of Xcode. Feel free to update your Xcode and continue working on your Kotlin projects for Apple operating systems.

The 2.1.21 release also fixes the related cinterop issue that caused compilation failures in Kotlin Multiplatform projects.

New inlining optimization

Kotlin 2.1.20 introduces a new inlining optimization pass, which comes before the actual code generation phase.

The new inlining pass in the Kotlin/Native compiler should perform better than the standard LLVM inliner and improve the runtime performance of the generated code.

The new inlining pass is currently Experimental. To try it out, use the following compiler option:

```
-Xbinary=preCodegenInlineThreshold=40
```

Our experiments show that setting the threshold to 40 tokens (code units parsed by the compiler) provides a reasonable compromise for compilation optimization. According to our benchmarks, this gives an overall performance improvement of 9.5%. Of course, you can try out other values, too.

If you experience increased binary size or compilation time, please report such issues via YouTrack.

Kotlin/Wasm

This release improves Kotlin/Wasm debugging and property usage. Custom formatters now work out of the box in development builds, while DWARF debugging facilitates code inspection. Additionally, the Provider API simplifies property usage in Kotlin/Wasm and Kotlin/JS.

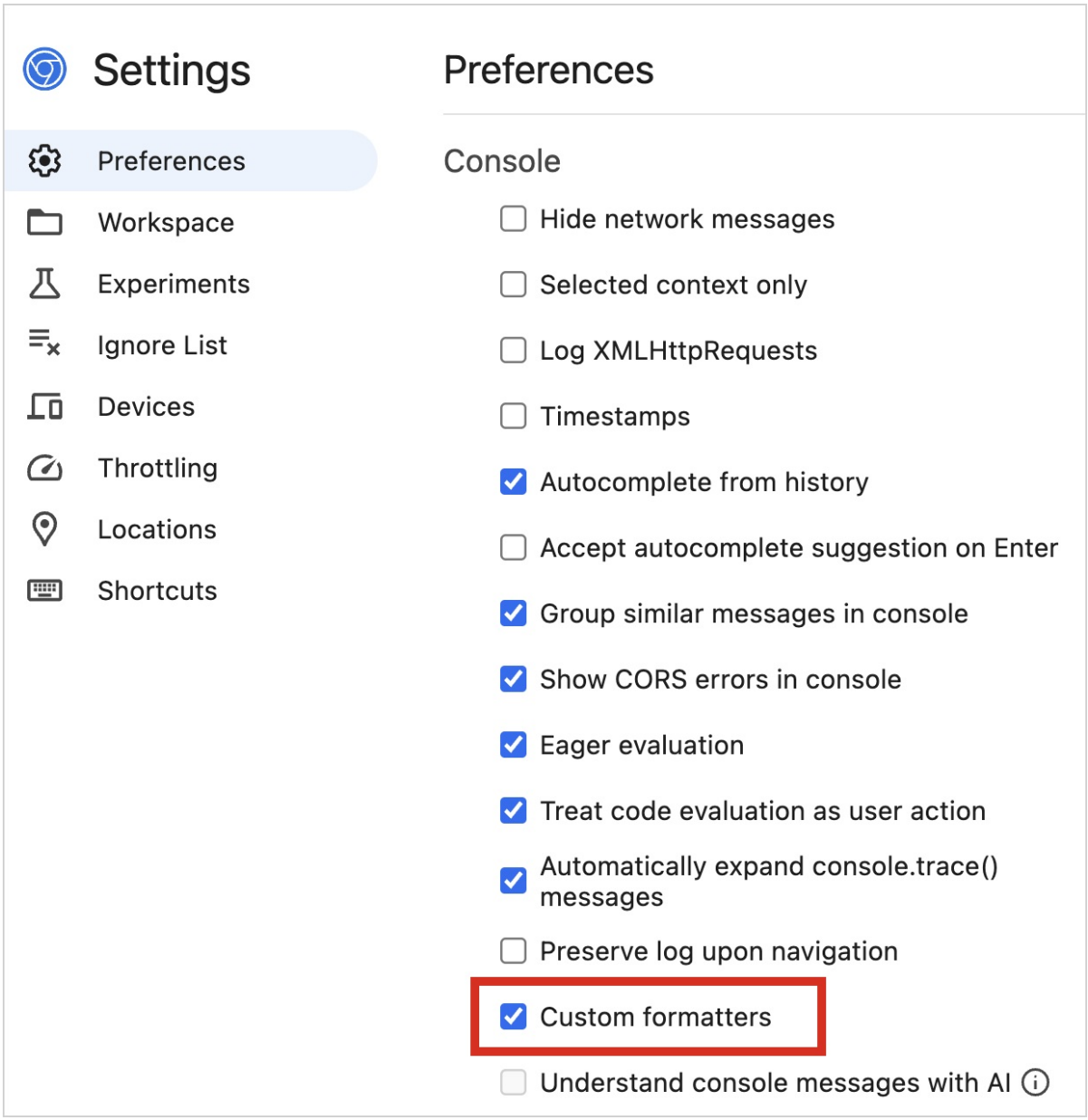
Custom formatters enabled by default

Before, you had to manually configure custom formatters to improve debugging in web browsers when working with Kotlin/Wasm code.

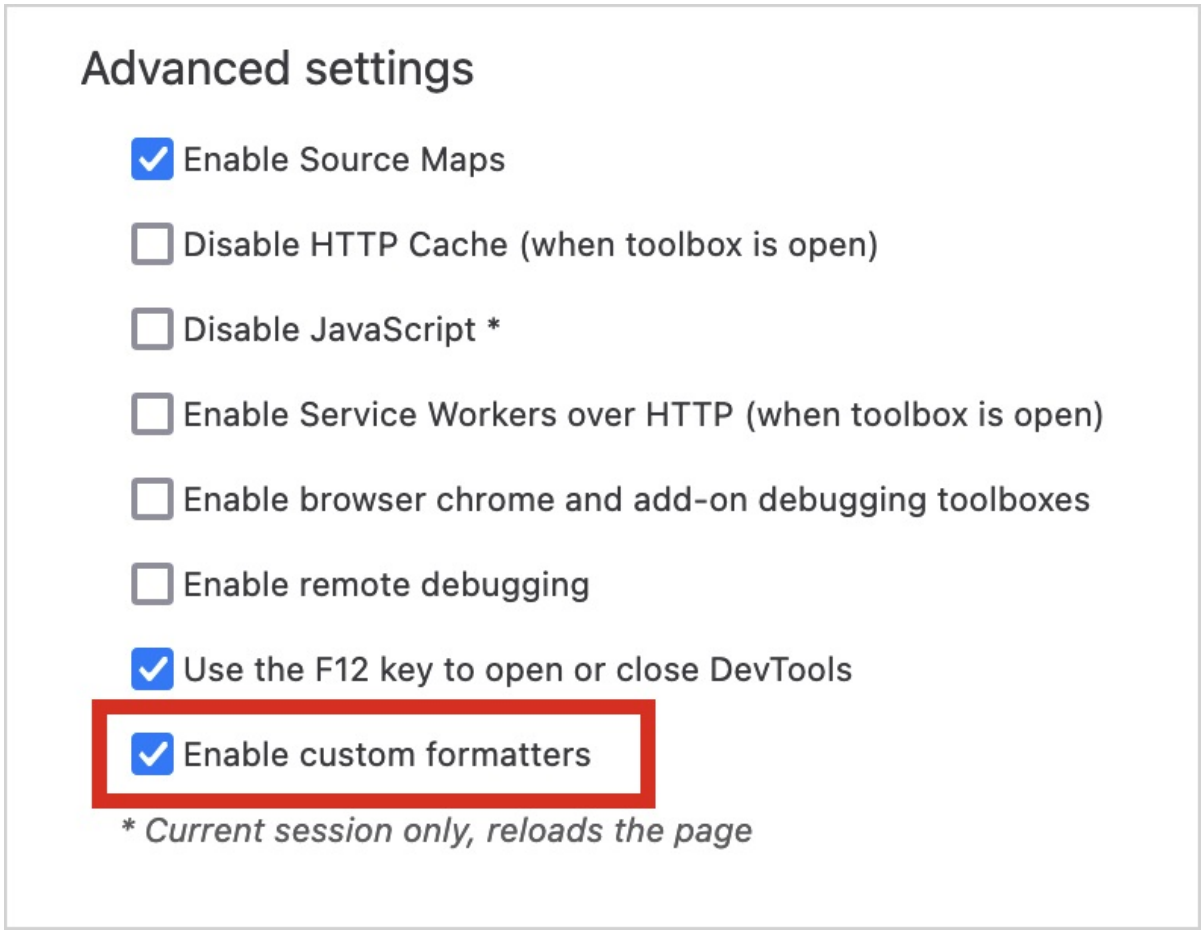
In this release, custom formatters are enabled by default in development builds, so you don't need additional Gradle configurations.

To use this feature, you only need to ensure that custom formatters are enabled in your browser's developer tools:

- In Chrome DevTools, find the custom formatters checkbox in **Settings | Preferences | Console**:



- In Firefox DevTools, find the custom formatters checkbox in **Settings | Advanced settings**:



This change primarily affects Kotlin/Wasm development builds. If you have specific requirements for production builds, you need to adjust your Gradle configuration accordingly. To do so, add the following compiler option to the `wasmJs {}` block:

```
// build.gradle.kts
kotlin {
    wasmJs {
        // ...

        compilerOptions {
            freeCompilerArgs.add("-Xwasm-debugger-custom-formatters")
        }
    }
}
```

Support for DWARF to debug Kotlin/Wasm code

Kotlin 2.1.20 introduces support for DWARF (debugging with arbitrary record format) in Kotlin/Wasm.

With this change, the Kotlin/Wasm compiler is able to embed DWARF data into the generated WebAssembly (Wasm) binary. Many debuggers and virtual machines can read this data to provide insights into the compiled code.

DWARF is mainly useful for debugging Kotlin/Wasm applications inside standalone Wasm virtual machines (VMs). To use this feature, the Wasm VM and debugger must support DWARF.

With DWARF support, you can step through Kotlin/Wasm applications, inspect variables, and gain code insights. To enable this feature, use the following compiler option:

```
-Xwasm-generate-dwarf
```

Migration to Provider API for Kotlin/Wasm and Kotlin/JS properties

Previously, properties in Kotlin/Wasm and Kotlin/JS extensions were mutable (`var`) and assigned directly in build scripts:

```
the<NodeJsExtension>().version = "2.0.0"
```

Now, properties are exposed through the Provider API, and you must use the `.set()` function to assign values:

```
the<NodeJsEnvSpec>().version.set("2.0.0")
```

The Provider API ensures that values are lazily computed and properly integrated with task dependencies, improving build performance.

With this change, direct property assignments are deprecated in favor of `*EnvSpec` classes, such as `NodeJsEnvSpec` and `YarnRootEnvSpec` .

Additionally, several alias tasks have been removed to avoid confusion:

Deprecated task	Replacement
`wasmJsRun`	`wasmJsBrowserDevelopmentRun`
`wasmJsBrowserRun`	`wasmJsBrowserDevelopmentRun`
`wasmJsNodeRun`	`wasmJsNodeDevelopmentRun`
`wasmJsBrowserWebpack`	`wasmJsBrowserProductionWebpack` or `wasmJsBrowserDistribution`
`jsRun`	`jsBrowserDevelopmentRun`
`jsBrowserRun`	`jsBrowserDevelopmentRun`
`jsNodeRun`	`jsNodeDevelopmentRun`
`jsBrowserWebpack`	`jsBrowserProductionWebpack` or `jsBrowserDistribution`

If you only use Kotlin/JS or Kotlin/Wasm in build scripts, no action is required as Gradle automatically handles assignments.

However, if you maintain a plugin based on the Kotlin Gradle Plugin, and your plugin does not apply `kotlin-dsl`, you must update property assignments to use the `.set()` function.

Gradle

Kotlin 2.1.20 is fully compatible with Gradle 7.6.3 through 8.11. You can also use Gradle versions up to the latest Gradle release. However, be aware that doing so may result in deprecation warnings, and some new Gradle features might not work.

This version of Kotlin includes compatibility of Kotlin Gradle plugins with Gradle's Isolated Projects as well as support for custom Gradle publication variants.

Kotlin Gradle plugins compatible with Gradle's Isolated Projects

> This feature is currently in a pre-Alpha state in Gradle. JS and Wasm targets are not supported at the moment. Use it only with Gradle version 8.10 or higher and solely for evaluation purposes.
>
> 

Since Kotlin 2.1.0, you've been able to preview Gradle's Isolated Projects feature in your projects.

Previously, you had to configure the Kotlin Gradle plugin to make your project compatible with the Isolated Projects feature before you could try it out. In Kotlin 2.1.20, this additional step is no longer necessary.

Now, to enable the Isolated Projects feature, you only need to set the system property.

Gradle's Isolated Projects feature is supported in Kotlin Gradle plugins for both multiplatform projects and projects that contain only the JVM or Android target.

Specifically for multiplatform projects, if you notice problems with your Gradle build after upgrading, you can opt out of the new Kotlin Gradle plugin behavior by adding:

```
kotlin.kmp.isolated-projects.support=disable
```

However, if you use this Gradle property in your multiplatform project, you can't use the Isolated Projects feature.

Let us know about your experience with this feature in YouTrack.

Support for adding custom Gradle publication variants

Kotlin 2.1.20 introduces support for adding custom Gradle publication variants. This feature is available for multiplatform projects and projects targeting the JVM.

> You cannot modify existing Gradle variants with this feature.
>
> 

This feature is Experimental. To opt in, use the `@OptIn(ExperimentalKotlinGradlePluginApi::class)` annotation.

To add a custom Gradle publication variant, invoke the `adhocSoftwareComponent()` function, which returns an instance of `AdhocComponentWithVariants` that you can configure in the Kotlin DSL:

```
plugins {
    // Only JVM and Multiplatform are supported
    kotlin("jvm")
    // or
    kotlin("multiplatform")
}


kotlin {
    @OptIn(ExperimentalKotlinGradlePluginApi::class)
    publishing {
        // Returns an instance of AdhocSoftwareComponent
        adhocSoftwareComponent()
        // Alternatively, you can configure AdhocSoftwareComponent in the DSL block as
 follows
        adhocSoftwareComponent {
            // Add your custom variants here using the AdhocSoftwareComponent API
        }
    }
}
```

> For more information on variants, see Gradle's Customizing publishing guide.
>
> 

Standard library

This release brings new Experimental features to the standard library: common atomic types, improved support for UUIDs, and new time-tracking functionality.

Common atomic types

In Kotlin 2.1.20, we are introducing common atomic types in the standard library's `kotlin.concurrent.atomics` package, enabling shared, platform-independent code for thread-safe operations. This simplifies development for Kotlin Multiplatform projects by removing the need to duplicate atomic-dependent logic across source sets.

The `kotlin.concurrent.atomics` package and its properties are Experimental. To opt in, use the `@OptIn(ExperimentalAtomicApi::class)` annotation or the compiler option `-opt-in=kotlin.ExperimentalAtomicApi`.

Here's an example that shows how you can use `AtomicInt` to safely count processed items across multiple threads:

```
// Imports the necessary libraries
import kotlin.concurrent.atomics.*
import kotlinx.coroutines.*

//sampleStart
@OptIn(ExperimentalAtomicApi::class)
suspend fun main() {
    // Initializes the atomic counter for processed items
    var processedItems = AtomicInt(0)
    val totalItems = 100
    val items = List(totalItems) { "item$it" }
    // Splits the items into chunks for processing by multiple coroutines
    val chunkSize = 20
    val itemChunks = items.chunked(chunkSize)
    coroutineScope {
        for (chunk in itemChunks) {
            launch {
                for (item in chunk) {
                    println("Processing $item in thread ${Thread.currentThread()}")
                    processedItems += 1 // Increment counter atomically
                }
            }
         }
    }
//sampleEnd
    // Prints the total number of processed items
    println("Total processed items: ${processedItems.load()}")
}
```

{validate="false" kotlin-runnable="true" kotlin-min-compiler-version="2.1.20"}

To enable seamless interoperability between Kotlin's atomic types and Java's `java.util.concurrent.atomic` atomic types, the API provides the `.asJavaAtomic()` and `.asKotlinAtomic()` extension functions. On the JVM, Kotlin atomics and Java atomics are the same types in runtime, so you can transform Java atomics into Kotlin atomics and vice versa without any overhead.

Here's an example that shows how Kotlin and Java atomic types can work together:

```
// Imports the necessary libraries
import kotlin.concurrent.atomics.*
import java.util.concurrent.atomic.*

//sampleStart
@OptIn(ExperimentalAtomicApi::class)
fun main() {
    // Converts Kotlin AtomicInt to Java's AtomicInteger
    val kotlinAtomic = AtomicInt(42)
    val javaAtomic: AtomicInteger = kotlinAtomic.asJavaAtomic()
    println("Java atomic value: ${javaAtomic.get()}")
    // Java atomic value: 42

    // Converts Java's AtomicInteger back to Kotlin's AtomicInt
    val kotlinAgain: AtomicInt = javaAtomic.asKotlinAtomic()
    println("Kotlin atomic value: ${kotlinAgain.load()}")
    // Kotlin atomic value: 42
}
//sampleEnd
```

{validate="false" kotlin-runnable="true" kotlin-min-compiler-version="2.1.20"}

Changes in UUID parsing, formatting, and comparability

The JetBrains team continues to improve the support for UUIDs introduced to the standard library in 2.0.20.

Previously, the `parse()` function only accepted UUIDs in the hex-and-dash format. With Kotlin 2.1.20, you can use `parse()` for *both* the hex-and-dash and the plain hexadecimal (without dashes) formats.

We've also introduced functions specific to operations with the hex-and-dash format in this release:

- `parseHexDash()` parses UUIDs from the hex-and-dash format.
- `toHexDashString()` converts a `Uuid` into a `String` in the hex-and-dash format (mirroring the functionality of `toString()`).

These functions work similarly to `parseHex()` and `toHexString()`, which were introduced earlier for the hexadecimal format. Explicit naming for parsing and formatting functionality should improve code clarity and your overall experience with UUIDs.

UUIDs in Kotlin are now `Comparable`. Starting with Kotlin 2.1.20, you can directly compare and sort values of the `Uuid` type. This enables the use of the `<` and `>` operators and standard library extensions available exclusively for `Comparable` types or their collections (such as `sorted()`), and it also allows passing UUIDs to any functions or APIs that require the `Comparable` interface.

Remember that the UUID support in the standard library is still Experimental. To opt in, use the `@OptIn(ExperimentalUuidApi::class)` annotation or the compiler option `-opt-in=kotlin.uuid.ExperimentalUuidApi`:

```
import kotlin.uuid.ExperimentalUuidApi
import kotlin.uuid.Uuid

//sampleStart
@OptIn(ExperimentalUuidApi::class)
fun main() {
    // parse() accepts a UUID in a plain hexadecimal format
    val uuid = Uuid.parse("550e8400e29b41d4a716446655440000")

    // Converts it to the hex-and-dash format
    val hexDashFormat = uuid.toHexDashString()

    // Outputs the UUID in the hex-and-dash format
    println(hexDashFormat)

    // Outputs UUIDs in ascending order
    println(
        listOf(
            uuid,
            Uuid.parse("780e8400e29b41d4a716446655440005"),
            Uuid.parse("5ab88400e29b41d4a716446655440076")
        ).sorted()
    )
   }
//sampleEnd
```

{validate="false" kotlin-runnable="true" kotlin-min-compiler-version="2.1.20"}

New time tracking functionality

Starting with Kotlin 2.1.20, the standard library provides the ability to represent a moment in time. This functionality was previously only available in `kotlinx-datetime`, an official Kotlin library.

The `kotlinx.datetime.Clock` interface is introduced to the standard library as `kotlin.time.Clock` and the `kotlinx.datetime.Instant` class as `kotlin.time.Instant`. These concepts naturally align with the `time` package in the standard library because they're only concerned with moments in time compared to a more complex calendar and timezone functionality that remains in `kotlinx-datetime`.

`Instant` and `Clock` are useful when you need precise time tracking without considering time zones or dates. For example, you can use them to log events with timestamps, measure durations between two points in time, and obtain the current moment for system processes.

To provide interoperability with other languages, additional converter functions are available:

- `.toKotlinInstant()` converts a time value to a `kotlin.time.Instant` instance.
- `.toJavaInstant()` converts the `kotlin.time.Instant` value to a `java.time.Instant` value.
- `Instant.toJSDate()` converts the `kotlin.time.Instant` value to an instance of the JS `Date` class. This conversion is not precise; JS uses millisecond precision to represent dates, while Kotlin allows for nanosecond resolution.

The new time features of the standard library are still Experimental. To opt in, use the `@OptIn(ExperimentalTime::class)` annotation:

```
import kotlin.time.*

@OptIn(ExperimentalTime::class)
fun main() {

    // Get the current moment in time
    val currentInstant = Clock.System.now()
    println("Current time: $currentInstant")

    // Find the difference between two moments in time
    val pastInstant = Instant.parse("2023-01-01T00:00:00Z")
    val duration = currentInstant - pastInstant

    println("Time elapsed since 2023-01-01: $duration")
}
```

{validate="false" kotlin-runnable="true" kotlin-min-compiler-version="2.1.20"}

For more information on the implementation, see this KEEP proposal.

Compose compiler

In 2.1.20, the Compose compiler relaxes some restrictions on `@Composable` functions introduced in previous releases. In addition, the Compose compiler Gradle plugin is set to include source information by default, aligning the behavior on all platforms with Android.

Support for default arguments in open `@Composable` functions

The compiler previously restricted default arguments in open `@Composable` functions due to incorrect compiler output, which would result in crashes at runtime. The underlying issue is now resolved, and default arguments are fully supported when used with Kotlin 2.1.20 or newer.

Compose compiler allowed default arguments in open functions before version 1.5.8, so the support depends on project configuration:

- If an open composable function is compiled with Kotlin version 2.1.20 or newer, the compiler generates correct wrappers for default arguments. This includes wrappers compatible with pre-1.5.8 binaries, meaning that downstream libraries will also be able to use this open function.
- If the open composable function is compiled with Kotlin older than 2.1.20, Compose uses a compatibility mode, which might result in runtime crashes. When using the compatibility mode, the compiler emits a warning to highlight potential problems.

Final overridden functions are allowed to be restartable

Virtual functions (overrides of `open` and `abstract`, including interfaces) were forced to be non-restartable with the 2.1.0 release. This restriction is now relaxed for functions that are members of final classes or are `final` themselves – they will be restarted or skipped as usual.

You might observe some behavior changes in affected functions after upgrading to Kotlin 2.1.20. To force non-restartable logic from the previous version, apply the `@NonRestartableComposable` annotation to the function.

`ComposableSingletons` removed from public API

`ComposableSingletons` is a class created by the Compose compiler when optimizing `@Composable` lambdas. Lambdas that do not capture any parameters are allocated once and cached in a property of the class, saving allocations during runtime. The class is generated with internal visibility and is only intended for optimizing lambdas inside a compilation unit (usually a file).

However, this optimization was also applied to `inline` function bodies, which resulted in singleton lambda instances leaking into the public API. To fix this problem, starting with 2.1.20, `@Composable` lambdas are no longer optimized into singletons inside inline functions. At the same time, the Compose compiler will continue generating singleton classes and lambdas for inline functions to support binary compatibility for modules that were compiled under the previous model.

Source information included by default

The Compose compiler Gradle plugin already has the including source information feature enabled by default on Android. Starting with Kotlin 2.1.20, this feature will be enabled by default on all platforms.

Remember to check if you set this option using `freeCompilerArgs`. This method can cause the build to fail when used alongside the plugin, due to an option being effectively set twice.

Breaking changes and deprecations

- To align Kotlin Multiplatform with upcoming changes in Gradle, we are phasing out the `withJava()` function. Java source sets are now created by default. If you use the Java test fixtures Gradle plugin, upgrade directly to Kotlin 2.1.21 to avoid compatibility issues.
- The JetBrains team is proceeding with the deprecation of the `kotlin-android-extensions` plugin. If you try to use it in your project, you'll now get a configuration error, and no plugin code will be executed.
- The legacy `kotlin.incremental.classpath.snapshot.enabled` property has been removed from the Kotlin Gradle plugin. The property used to provide an opportunity to fall back to a built-in ABI snapshot on the JVM. The plugin now uses other methods to detect and avoid unnecessary recompilations, making the property obsolete.

Documentation updates

The Kotlin documentation has received some notable changes:

Revamped and new pages

- Kotlin roadmap – see the updated list of Kotlin's priorities on language and ecosystem evolution.
- Gradle best practices page – learn essential best practices for optimizing your Gradle builds and improving performance.
- Compose Multiplatform and Jetpack Compose – an overview of the relation between the two UI frameworks.
- Kotlin Multiplatform and Flutter – see the comparison of two popular cross-platform frameworks.
- Interoperability with C – explore the details of Kotlin's interoperability with C.
- Numbers – learn about different Kotlin types for representing numbers.

New and updated tutorials

- Publish your library to Maven Central – learn how to publish KMP library artifacts to the most popular Maven repository.
- Kotlin/Native as a dynamic library – create a dynamic Kotlin library.
- Kotlin/Native as an Apple framework – create your own framework and use Kotlin/Native code from Swift/Objective-C applications on macOS and iOS.

How to update to Kotlin 2.1.20

Starting from IntelliJ IDEA 2023.3 and Android Studio Iguana (2023.2.1) Canary 15, the Kotlin plugin is distributed as a bundled plugin included in your IDE. This means that you can't install the plugin from JetBrains Marketplace anymore.

To update to the new Kotlin version, change the Kotlin version to 2.1.20 in your build scripts.

Kotlin 2.1.0 的新特性

发布于：*2024-11-27*

The Kotlin 2.1.0 release is here! Here are the main highlights:

- **New language features in preview**: Guard conditions in `when` with a subject, non-local `break` and `continue`, and multi-dollar string interpolation.
- **K2 compiler updates**: More flexibility around compiler checks and improvements to the kapt implementation.
- **Kotlin Multiplatform**: Introduced basic support for Swift export, stable Gradle DSL for compiler options, and more.
- **Kotlin/Native**: Improved support for `iosArm64` and other updates.
- **Kotlin/Wasm**: Multiple updates, including support for incremental compilation.
- **Gradle support**: Improved compatibility with newer versions of Gradle and the Android Gradle plugin, along with updates to the Kotlin Gradle plugin API.
- **Documentation**: Significant improvements to the Kotlin documentation.

IDE support

The Kotlin plugins that support 2.1.0 are bundled in the latest IntelliJ IDEA and Android Studio. You don't need to update the Kotlin plugin in your IDE. All you need to do is change the Kotlin version to 2.1.0 in your build scripts.

See Update to a new Kotlin version for details.

Language

After the Kotlin 2.0.0 release with the K2 compiler, the JetBrains team is focusing on improving the language with new features. In this release, we are excited to announce several new language design improvements.

These features are available in preview, and we encourage you to try them and share your feedback:

- Guard conditions in `when` with a subject
- Non-local `break` and `continue`
- Multi-dollar interpolation: improved handling of `$` in string literals

> All the features have IDE support in the latest 2024.3 version of IntelliJ IDEA with K2 mode enabled.

> Learn more in the IntelliJ IDEA 2024.3 blog post.
>
> 

See the full list of Kotlin language design features and proposals.

This release also brings the following language updates:

-
-
-

Guard conditions in when with a subject

> This feature is In preview, and opt-in is required (see details below).
>
> We would appreciate your feedback in YouTrack.
>
> 

Starting from 2.1.0, you can use guard conditions in `when` expressions or statements with subjects.

Guard conditions allow you to include more than one condition for the branches of a `when` expression, making complex control flows more explicit and concise as well as flattening the code structure.

To include a guard condition in a branch, place it after the primary condition, separated by `if`:

```
sealed interface Animal {
    data class Cat(val mouseHunter: Boolean) : Animal {
        fun feedCat() {}
    }

    data class Dog(val breed: String) : Animal {
        fun feedDog() {}
    }
}

fun feedAnimal(animal: Animal) {
    when (animal) {
        // Branch with only the primary condition. Calls `feedDog()` when `animal` is
`Dog`
        is Animal.Dog -> animal.feedDog()
        // Branch with both primary and guard conditions. Calls `feedCat()` when `anim
al` is `Cat` and is not `mouseHunter`
        is Animal.Cat if !animal.mouseHunter -> animal.feedCat()
        // Prints "Unknown animal" if none of the above conditions match
        else -> println("Unknown animal")
    }
}
```

In a single `when` expression, you can combine branches with and without guard conditions. The code in a branch with a guard condition runs only if both the primary condition and the guard condition are `true`. If the primary condition does not match, the guard condition is not evaluated. Additionally, guard conditions support `else if`.

To enable guard conditions in your project, use the following compiler option in the command line:

```
kotlinc -Xwhen-guards main.kt
```

Or add it to the `compilerOptions {}` block of your Gradle build file:

```
// build.gradle.kts
kotlin {
    compilerOptions {
        freeCompilerArgs.add("-Xwhen-guards")
    }
}
```

Non-local break and continue

> This feature is In preview, and opt-in is required (see details below).
>
> We would appreciate your feedback in YouTrack.
>
> 

Kotlin 2.1.0 adds a preview of another long-awaited feature, the ability to use non-local `break` and `continue`. This feature expands the toolset you can use in the scope of inline functions and reduces boilerplate code in your project.

Previously, you could use only non-local returns. Now, Kotlin also supports `break` and `continue` jump expressions non-locally. This means that you can apply them within lambdas passed as arguments to an inline function that encloses the loop:

```
fun processList(elements: List<Int>): Boolean {
    for (element in elements) {
        val variable = element.nullableMethod() ?: run {
            log.warning("Element is null or invalid, continuing...")
            continue
        }
        if (variable == 0) return true // If variable is zero, return true
    }
    return false
}
```

To try the feature in your project, use the `-Xnon-local-break-continue` compiler option in the command line:

```
kotlinc -Xnon-local-break-continue main.kt
```

Or add it in the `compilerOptions {}` block of your Gradle build file:

```
// build.gradle.kts
kotlin {
    compilerOptions {
        freeCompilerArgs.add("-Xnon-local-break-continue")
    }
}
```

We're planning to make this feature Stable in future Kotlin releases. If you encounter any issues when using non-local `break` and `continue`, please report them to our issue tracker.

Multi-dollar string interpolation

> The feature is In preview and opt-in is required (see details below).
>
> We would appreciate your feedback in YouTrack.
>
> 

Kotlin 2.1.0 introduces support for multi-dollar string interpolation, improving how the dollar sign (`$`) is handled within string literals. This feature is helpful in contexts that require multiple dollar signs, such as templating engines, JSON schemas, or other data formats.

String interpolation in Kotlin uses a single dollar sign. However, using a literal dollar sign in a string, which is common in financial data and templating systems, required workarounds such as `${'$'}`. With the multi-dollar interpolation feature enabled, you can configure how many dollar signs trigger interpolation, with fewer dollar signs being treated as string literals.

Here's an example of how to generate an JSON schema multi-line string with placeholders using `$`:

```
val KClass<*>.jsonSchema : String
    get() = $$"""
    {
      "$schema": "https://json-schema.org/draft/2020-12/schema",
      "$id": "https://example.com/product.schema.json",
      "$dynamicAnchor": "meta"
      "title": "$${simpleName ?: qualifiedName ?: "unknown"}",
      "type": "object"
    }
    """
```

In this example, the initial `$$` means that you need **two dollar signs** (`$$`) to trigger interpolation. It prevents `$schema` , `$id` , and `$dynamicAnchor` from being interpreted as interpolation markers.

This approach is especially helpful when working with systems that use dollar signs for placeholder syntax.

To enable the feature, use the following compiler option in the command line:

```
kotlinc -Xmulti-dollar-interpolation main.kt
```

Alternatively, update the `compilerOptions {}` block of your Gradle build file:

```
// build.gradle.kts
kotlin {
    compilerOptions {
        freeCompilerArgs.add("-Xmulti-dollar-interpolation")
    }
}
```

If your code already uses standard string interpolation with a single dollar sign, no changes are needed. You can use `$$` whenever you need literal dollar signs in your strings.

Support for requiring opt-in to extend APIs

Kotlin 2.1.0 introduces the `@SubclassOptInRequired` annotation, which allows library authors to require an explicit opt-in before users can implement experimental interfaces or extend experimental classes.

This feature can be useful when a library API is stable enough to use but might evolve with new abstract functions, making it unstable for inheritance.

To add the opt-in requirement to an API element, use the `@SubclassOptInRequired` annotation with a reference to the annotation class:

```
@RequiresOptIn(
level = RequiresOptIn.Level.WARNING,
message = "Interfaces in this library are experimental"
)
annotation class UnstableApi()

@SubclassOptInRequired(UnstableApi::class)
interface CoreLibraryApi
```

In this example, the `CoreLibraryApi` interface requires users to opt in before they can implement it. A user can opt in like this:

```
@OptIn(UnstableApi::class)
interface MyImplementation: CoreLibraryApi
```

> When you use the `@SubclassOptInRequired` annotation to require opt-in, the requirement is not propagated to any inner or nested classes.
>
> 

For a real-world example of how to use the `@SubclassOptInRequired` annotation in your API, check out the `SharedFlow` interface in the `kotlinx.coroutines` library.

Improved overload resolution for functions with generic types

Previously, if you had a number of overloads for a function where some had value parameters of a generic type and others had function types in the same position, the resolution behavior could sometimes be inconsistent.

This led to different behavior depending on whether your overloads were member functions or extension functions. For example:

```
class KeyValueStore<K, V> {
    fun store(key: K, value: V) {} // 1
    fun store(key: K, lazyValue: () -> V) {} // 2
}

fun <K, V> KeyValueStore<K, V>.storeExtension(key: K, value: V) {} // 1
fun <K, V> KeyValueStore<K, V>.storeExtension(key: K, lazyValue: () -> V) {} // 2

fun test(kvs: KeyValueStore<String, Int>) {
    // Member functions
    kvs.store("", 1)    // Resolves to 1
    kvs.store("") { 1 } // Resolves to 2

    // Extension functions
    kvs.storeExtension("", 1)    // Resolves to 1
    kvs.storeExtension("") { 1 } // Doesn't resolve
}
```

In this example, the `KeyValueStore` class has two overloads for the `store()` function, where one overload has function parameters with generic types `K` and `V`, and another has a lambda function that returns a generic type `V`. Similarly, there are two overloads for the extension function: `storeExtension()`.

When the `store()` function was called with and without a lambda function, the compiler successfully resolved the correct overloads. However, when the extension function `storeExtension()` was called with a lambda function, the compiler didn't resolve the correct overload because it incorrectly considered both overloads to be applicable.

To fix this problem, we've introduced a new heuristic so that the compiler can discard a possible overload when a function parameter with a generic type can't accept a lambda function based on information from a different argument. This change makes the behavior of member functions and extension functions consistent, and it is enabled by default in Kotlin 2.1.0.

Improved exhaustiveness checks for when expressions with sealed classes

In previous versions of Kotlin, the compiler required an `else` branch in `when` expressions for type parameters with sealed upper bounds, even when all cases in the `sealed class` hierarchy were covered. This behavior is addressed and improved in Kotlin 2.1.0, making exhaustiveness checks more powerful and allowing you to remove redundant `else` branches, keeping `when` expressions cleaner and more intuitive.

Here's an example demonstrating the change:

```
sealed class Result
object Error: Result()
class Success(val value: String): Result()

fun <T : Result> render(result: T) = when (result) {
    Error -> "Error!"
    is Success -> result.value
    // Requires no else branch
}
```

Kotlin K2 compiler

With Kotlin 2.1.0, the K2 compiler now provides more flexibility when working with compiler checks and warnings, as well as improved support for the kapt plugin.

Extra compiler checks

With Kotlin 2.1.0, you can now enable additional checks in the K2 compiler. These are extra declaration, expression, and type checks that are usually not crucial for compilation but can still be useful if you want to validate the following cases:

Check type	Comment

`REDUNDANT_NULLABLE`	`Boolean??` is used instead of `Bool`
`PLATFORM_CLASS_MAPPED_TO_KOTLIN`	`java.lang.String` is used instead `kotlin.String`
`ARRAY_EQUALITY_OPERATOR_CAN_BE_REPLACED_WITH_EQUALS`	`arrayOf("") == arrayOf("")` is us instead of `arrayOf("").contentEquals(arrayO`
`REDUNDANT_CALL_OF_CONVERSION_METHOD`	`42.toInt()` is used instead of `42`
`USELESS_CALL_ON_NOT_NULL`	`"".orEmpty()` is used instead of `"`
`REDUNDANT_SINGLE_EXPRESSION_STRING_TEMPLATE`	`"$string"` is used instead of `stri`
`UNUSED_ANONYMOUS_PARAMETER`	A parameter is passed in the lambda expression but never used
`REDUNDANT_VISIBILITY_MODIFIER`	`public class Klass` is used instea `class Klass`
`REDUNDANT_MODALITY_MODIFIER`	`final class Klass` is used instead `class Klass`
`REDUNDANT_SETTER_PARAMETER_TYPE`	`set(value: Int)` is used instead of `set(value)`
`CAN_BE_VAL`	`var local = 0` is defined but neve reassigned, can be `val local = 42`
`ASSIGNED_VALUE_IS_NEVER_READ`	`val local = 42` is defined but nev afterward in the code
`UNUSED_VARIABLE`	`val local = 0` is defined but neve in the code
`REDUNDANT_RETURN_UNIT_TYPE`	`fun foo(): Unit {}` is used instea `fun foo() {}`
`UNREACHABLE_CODE`	Code statement is present but can ne executed

If the check is true, you'll receive a compiler warning with a suggestion on how to fix the problem.

Extra checks are disabled by default. To enable them, use the `-Wextra` compiler option in the command line or specify `extraWarnings` in the `compilerOptions {}` block of your Gradle build file:

```
// build.gradle.kts
kotlin {
    compilerOptions {
        extraWarnings.set(true)
    }
}
```

For more information on how to define and use compiler options, see Compiler options in the Kotlin Gradle plugin.

Global warning suppression

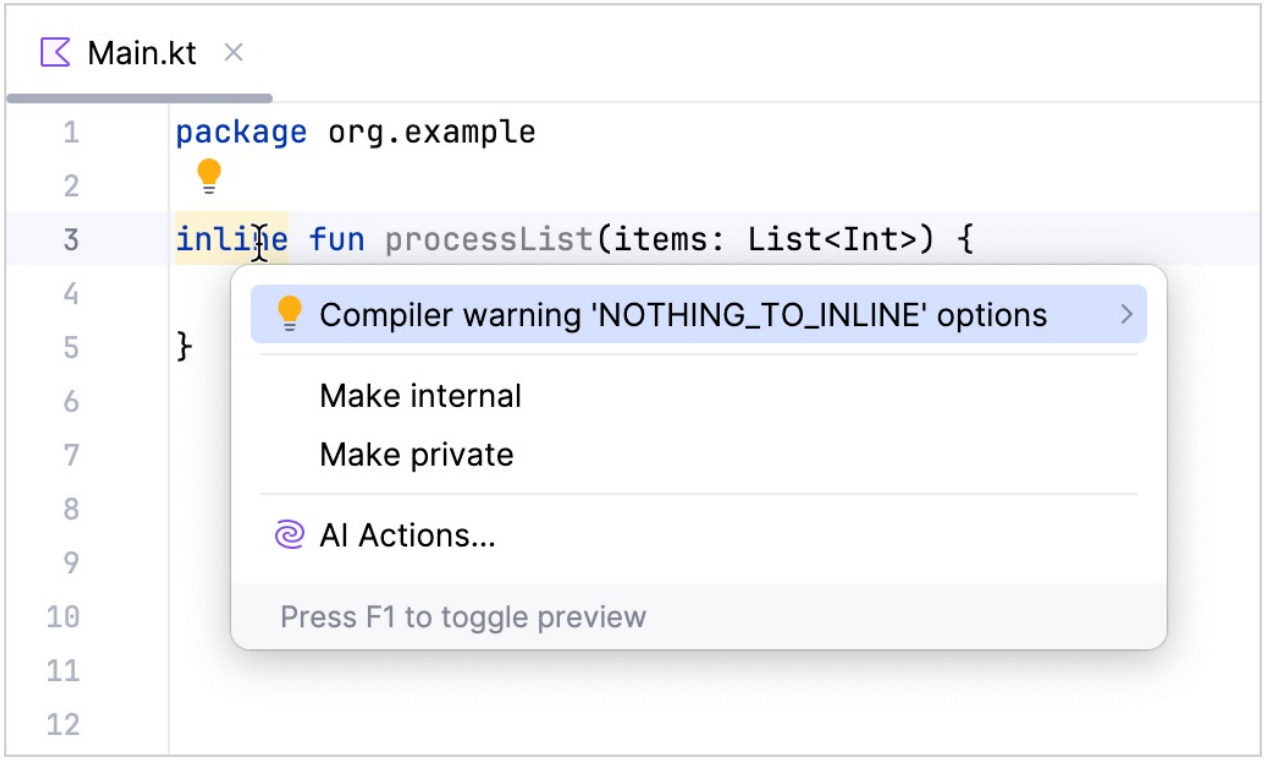
In 2.1.0, the Kotlin compiler has received a highly requested feature – the ability to suppress warnings globally.

You can now suppress specific warnings in the whole project by using the `-Xsuppress-warning=WARNING_NAME` syntax in the command line or the `freeCompilerArgs` attribute in the `compilerOptions {}` block of your build file.

For example, if you have extra compiler checks enabled in your project but want to suppress one of them, use:

```
// build.gradle.kts
kotlin {
    compilerOptions {
        extraWarnings.set(true)
        freeCompilerArgs.add("-Xsuppress-warning=CAN_BE_VAL")
    }
}
```

If you want to suppress a warning but don't know its name, select the element and click the light bulb icon (or use Cmd + Enter/Alt + Enter):



The new compiler option is currently Experimental. The following details are also worth noting:

- Error suppression is not allowed.
- If you pass an unknown warning name, compilation will result in an error.
- You can specify several warnings at once:

【Command line】

```
  kotlinc -Xsuppress-warning=NOTHING_TO_INLINE -Xsuppress-warning=NO_TAIL_CALLS_FOUND
main.kt
```

【Build file】

```
  // build.gradle.kts
  kotlin {
      compilerOptions {
          freeCompilerArgs.addAll(
              listOf(
                  "-Xsuppress-warning=NOTHING_TO_INLINE",
                  "-Xsuppress-warning=NO_TAIL_CALLS_FOUND"
              )
          )
      }
  }
```

Improved K2 kapt implementation

> The kapt plugin for the K2 compiler (K2 kapt) is in Alpha. It may be changed at any time.
>
> We would appreciate your feedback in YouTrack.
>
> 

Currently, projects using the kapt plugin work with the K1 compiler by default, supporting Kotlin versions up to 1.9.

In Kotlin 1.9.20, we launched an experimental implementation of the kapt plugin with the K2 compiler (K2 kapt). We have now improved K2 kapt's internal implementation to mitigate technical and performance issues.

While the new K2 kapt implementation doesn't introduce new features, its performance has significantly improved compared to the previous K2 kapt implementation. Additionally, the K2 kapt plugin's behavior is now much closer to that of K1 kapt.

To use the new K2 kapt plugin implementation, enable it just like you did the previous K2 kapt plugin. Add the following option to the `gradle.properties` file of your project:

```
kapt.use.k2=true
```

In upcoming releases, the K2 kapt implementation will be enabled by default instead of K1 kapt, so you will no longer need to enable it manually.

We highly appreciate your feedback before the new implementation is stabilized.

Resolution for overload conflicts between unsigned and non-primitive types

This release addresses the issue of resolution for overload conflicts that could occur in previous versions when functions were overloaded for unsigned and non-primitive types, as in the following examples:

Overloaded extension functions

```
fun Any.doStuff() = "Any"
fun UByte.doStuff() = "UByte"

fun main() {
    val uByte: UByte = UByte.MIN_VALUE
    uByte.doStuff() // Overload resolution ambiguity before Kotlin 2.1.0
}
```

In earlier versions, calling `uByte.doStuff()` led to ambiguity because both the `Any` and `UByte` extensions were applicable.

Overloaded top-level functions

```
fun doStuff(value: Any) = "Any"
fun doStuff(value: UByte) = "UByte"

fun main() {
    val uByte: UByte = UByte.MIN_VALUE
    doStuff(uByte) // Overload resolution ambiguity before Kotlin 2.1.0
}
```

Similarly, the call to `doStuff(uByte)` was ambiguous because the compiler couldn't decide whether to use the `Any` or `UByte` version. With 2.1.0, the compiler now handles these cases correctly, resolving the ambiguity by giving precedence to the more specific type, in this case `UByte`.

Kotlin/JVM

Starting with version 2.1.0, the compiler can generate classes containing Java 23 bytecode.

Change of JSpecify nullability mismatch diagnostics severity to strict

Kotlin 2.1.0 enforces strict handling of nullability annotations from `org.jspecify.annotations`, improving type safety for Java interoperability.

The following nullability annotations are affected:

- `org.jspecify.annotations.Nullable`
- `org.jspecify.annotations.NonNull`
- `org.jspecify.annotations.NullMarked`
- Legacy annotations in `org.jspecify.nullness` (JSpecify 0.2 and earlier)

Starting from Kotlin 2.1.0, nullability mismatches are raised from warnings to errors by default. This ensures that annotations like `@NonNull` and `@Nullable` are enforced during type checks, preventing unexpected nullability issues at runtime.

The `@NullMarked` annotation also affects the nullability of all members within its scope, making the behavior more predictable when you're working with annotated Java code.

Here's an example demonstrating the new default behavior:

```
// Java
import org.jspecify.annotations.*;
public class SomeJavaClass {
    @NonNull
    public String foo() { //...
    }

    @Nullable
    public String bar() { //...
    }
}
```

```
// Kotlin
fun test(sjc: SomeJavaClass) {
```

```
    // Accesses a non-null result, which is allowed
    sjc.foo().length

    // Raises an error in the default strict mode because the result is nullable
    // To avoid the error, use ?.length instead
    sjc.bar().length
}
```

You can manually control the severity of diagnostics for these annotations. To do so, use the `-Xnullability-annotations` compiler option to choose a mode:

- `ignore` : Ignore nullability mismatches.
- `warning` : Report warnings for nullability mismatches.
- `strict` : Report errors for nullability mismatches (default mode).

For more information, see Nullability annotations.

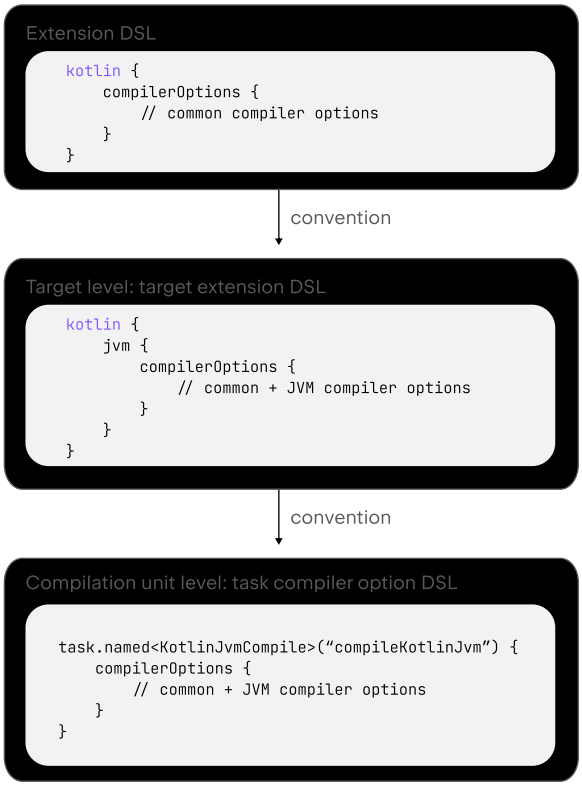
Kotlin Multiplatform

Kotlin 2.1.0 introduces basic support for Swift export and makes publishing Kotlin Multiplatform libraries easier. It also focuses on improvements around Gradle that stabilize the new DSL for configuring compiler options and bring a preview of the Isolated Projects feature.

New Gradle DSL for compiler options in multiplatform projects promoted to Stable

In Kotlin 2.0.0, we introduced a new Experimental Gradle DSL to simplify the configuration of compiler options across your multiplatform projects. In Kotlin 2.1.0, this DSL has been promoted to Stable.

The overall project configuration now has three layers. The highest is the extension level, then the target level, and the lowest is the compilation unit (which is usually a compilation task):



To learn more about the different levels and how compiler options can be configured between them, see Compiler options.

Preview Gradle's Isolated Projects in Kotlin Multiplatform

> This feature is Experimental and is currently in a pre-Alpha state in Gradle. Use it only with Gradle version 8.10 and solely for evaluation purposes. The feature may be dropped or changed at any time.
>
> We would appreciate your feedback on it in YouTrack. Opt-in is required (see details below).
>
> 

In Kotlin 2.1.0, you can preview Gradle's Isolated Projects feature in your multiplatform projects.

The Isolated Projects feature in Gradle improves build performance by "isolating" configuration of individual Gradle projects from each other. Each project's build logic is restricted from directly accessing the mutable state of other projects, allowing them to safely run in parallel. To support this feature, we made some changes to the Kotlin Gradle plugin's model, and we are interested in hearing about your experiences during this preview phase.

There are two ways to enable the Kotlin Gradle plugin's new model:

- Option 1: **Testing compatibility without enabling Isolated Projects** – To check compatibility with the Kotlin Gradle plugin's new model without enabling the Isolated Projects feature, add the following Gradle property in the `gradle.properties` file of your project:

  ```
  # gradle.properties
  kotlin.kmp.isolated-projects.support=enable
  ```

- Option 2: **Testing with Isolated Projects enabled** – Enabling the Isolated Projects feature in Gradle automatically configures the Kotlin Gradle plugin to use the new model. To enable the Isolated Projects feature, set the system property. In this case, you don't need to add the Gradle property for the Kotlin Gradle plugin to your project.

Basic support for Swift export

> This feature is currently in the early stages of development. It may be dropped or changed at any time. Opt-in is required (see the details below), and you should use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack.
>
> 

Version 2.1.0 takes the first step towards providing support for Swift export in Kotlin, allowing you to export Kotlin sources directly to the Swift interface without using Objective-C headers. This should make multiplatform development for Apple targets easier.

The current basic support includes the ability to:

- Export multiple Gradle modules from Kotlin directly to Swift.
- Define custom Swift module names with the `moduleName` property.
- Set collapse rules for the package structure with the `flattenPackage` property.

You can use the following build file in your project as a starting point for setting up Swift export:

```
// build.gradle.kts
kotlin {

    iosX64()
    iosArm64()
    iosSimulatorArm64()

    @OptIn(ExperimentalSwiftExportDsl::class)
    swiftExport {
        // Root module name
        moduleName = "Shared"

        // Collapse rule
        // Removes package prefix from generated Swift code
```

```
            flattenPackage = "com.example.sandbox"

            // Export external modules
            export(project(":subproject")) {
                // Exported module name
                moduleName = "Subproject"
                // Collapse exported dependency rule
                flattenPackage = "com.subproject.library"
            }
        }
    }
```

You can also clone our public sample with Swift export already set up.

The compiler automatically generates all the necessary files (including `swiftmodule` files, static `a` library, and header and `modulemap` files) and copies them into the app's build directory, which you can access from Xcode.

How to enable Swift export

Keep in mind that the feature is currently only in the early stages of development.

Swift export currently works in projects that use direct integration to connect the iOS framework to the Xcode project. This is a standard configuration for Kotlin Multiplatform projects created in Android Studio or through the web wizard.

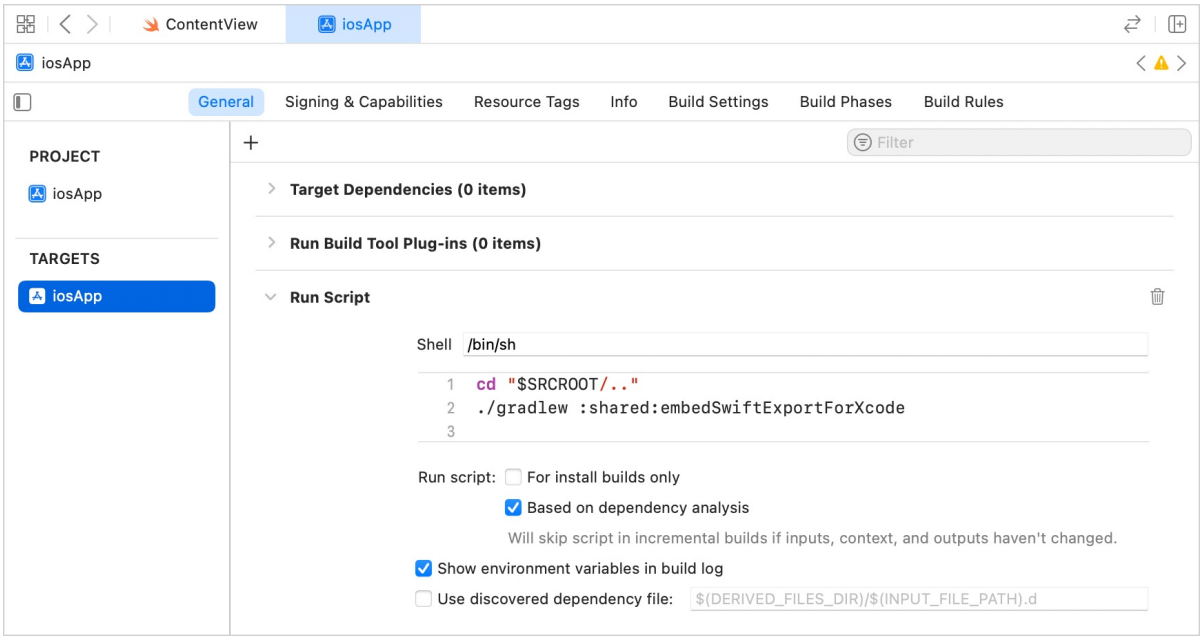
To try out Swift export in your project:

1. Add the following Gradle option to your `gradle.properties` file:

   ```
   # gradle.properties
   kotlin.experimental.swift-export.enabled=true
   ```

2. In Xcode, open the project settings.
3. On the **Build Phases** tab, locate the **Run Script** phase with the `embedAndSignAppleFrameworkForXcode` task.
4. Adjust the script to feature the `embedSwiftExportForXcode` task instead in the run script phase:

   ```
   ./gradlew :<Shared module name>:embedSwiftExportForXcode
   ```



Leave feedback on Swift export

We're planning to expand and stabilize Swift export support in future Kotlin releases. Please leave your feedback in this YouTrack issue.

Ability to publish Kotlin libraries from any host

> This feature is currently Experimental. Opt-in is required (see the details below), and you should use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack.
>
> 

The Kotlin compiler produces `.klib` artifacts for publishing Kotlin libraries. Previously, you could get the necessary artifacts from any host, except for Apple platform targets that required a Mac machine. That put a special restraint on Kotlin Multiplatform projects that targeted iOS, macOS, tvOS, and watchOS targets.

Kotlin 2.1.0 lifts this restriction, adding support for cross-compilation. Now you can use any host to produce `.klib` artifacts, which should greatly simplify the publishing process for Kotlin and Kotlin Multiplatform libraries.

How to enable publishing libraries from any host

To try cross-compilation out in your project, add the following binary option to your `gradle.properties` file:

```
# gradle.properties
kotlin.native.enableKlibsCrossCompilation=true
```

This feature is currently Experimental and has some limitations. You still need to use a Mac machine if:

- Your library has a cinterop dependency.
- You have CocoaPods integration set up in your project.
- You need to build or test final binaries for Apple targets.

Leave feedback on publishing libraries from any host

We're planning to stabilize this feature and further improve library publication in future Kotlin releases. Please leave your feedback in our issue tracker YouTrack.

For more information, see Publishing multiplatform libraries.

Support for non-packed klibs

Kotlin 2.1.0 makes it possible to generate non-packed `.klib` file artifacts. This gives you the option to configure dependencies on klibs directly rather than unpack them first.

This change can also improve performance, decreasing compilation and linking time in your Kotlin/Wasm, Kotlin/JS, and Kotlin/Native projects.

For example, our benchmark shows a performance improvement of roughly 3% in total build time on the project with 1 linking and 10 compilation tasks (the project builds a single native executable binary that depends on 9 simplified projects). However, the actual impact on build time depends on both the number of subprojects and their respective sizes.

How to set up your project

By default, Kotlin compilation and linking tasks are now configured to use the new non-packed artifacts.

If you have set up custom build logic for resolving klibs and want to use the new unpacked artifacts, you need to explicitly specify the preferred variant of klib package resolution in your Gradle build file:

```
// build.gradle.kts
import org.jetbrains.kotlin.gradle.plugin.attributes.KlibPackaging
// ...
val resolvableConfiguration = configurations.resolvable("resolvable") {

    // For the new non-packed configuration:
    attributes.attribute(KlibPackaging.ATTRIBUTE, project.objects.named(KlibPackaging.
NON_PACKED))
```

```
    // For the previous packed configuration:
    attributes.attribute(KlibPackaging.ATTRIBUTE, project.objects.named(KlibPackaging.
PACKED))
}
```

Non-packed `.klib` files are generated at the same path in your project's build directory as the packed ones previously were. In turn, packed klibs are now located in the `build/libs` directory.

If no attribute is specified, the packed variant is used. You can check the list of available attributes and variants with the following console command:

```
./gradlew outgoingVariants
```

We would appreciate your feedback on this feature in YouTrack.

Further deprecation of old `android` target

In Kotlin 2.1.0, the deprecation warning for the old `android` target name has been raised to an error.

Currently, we recommend using the `androidTarget` option in your Kotlin Multiplatform projects targeting Android. This is a temporary solution that is necessary to free the `android` name for the upcoming Android/KMP plugin from Google.

We'll provide further migration instructions when the new plugin is available. The new DSL from Google will be the preferred option for Android target support in Kotlin Multiplatform.

For more information, see the Kotlin Multiplatform compatibility guide.

Dropped support for declaring multiple targets of the same type

Before Kotlin 2.1.0, you could declare multiple targets of the same type in your multiplatform projects. However, this made it challenging to distinguish between targets and to support shared source sets effectively. In most cases, a simpler setup, such as using separate Gradle projects, works better. For detailed guidance and an example of how to migrate, see Declaring several similar targets in the Kotlin Multiplatform compatibility guide.

Kotlin 1.9.20 triggered a deprecation warning if you declared multiple targets of the same type in your multiplatform projects. In Kotlin 2.1.0, this deprecation warning is now an error for all targets except Kotlin/JS ones. To learn more about why Kotlin/JS targets are exempt, see this issue in YouTrack.

Kotlin/Native

Kotlin 2.1.0 includes an upgrade for the `iosArm64` target support, improved cinterop caching process, and other updates.

iosArm64 promoted to Tier 1

The `iosArm64` target, which is crucial for Kotlin Multiplatform development, has been promoted to Tier 1. This is the highest level of support in the Kotlin/Native compiler.

This means the target is regularly tested on the CI pipeline to ensure that it's able to compile and run. We also provide source and binary compatibility between compiler releases for the target.

For more information on target tiers, see Kotlin/Native target support.

LLVM update from 11.1.0 to 16.0.0

In Kotlin 2.1.0, we updated LLVM from version 11.1.0 to 16.0.0. The new version includes bug fixes and security updates. In certain cases, it also provides compiler optimizations and faster compilation.

If you have Linux targets in your project, take note that the Kotlin/Native compiler now uses the `lld` linker by default for all Linux targets.

This update shouldn't affect your code, but if you encounter any issues, please report them to our issue tracker.

Changes to caching in cinterop

In Kotlin 2.1.0, we're making changes to the cinterop caching process. It no longer has the `CacheableTask` annotation type. The new recommended approach is to use the `cacheIf` output type to cache the results of the task.

This should resolve issues where `UP-TO-DATE` checks failed to detect changes to header files specified in the definition file, preventing the build system from recompiling the code.

Deprecation of the mimalloc memory allocator

Back in Kotlin 1.9.0, we introduced the new memory allocator, and then we enabled it by default in Kotlin 1.9.20. The new allocator has been designed to make garbage collection more efficient and improve the Kotlin/Native memory manager's runtime performance.

The new memory allocator replaced the previous default allocator, mimalloc. Now, it's time to deprecate mimalloc in the Kotlin/Native compiler.

You can now remove the `-Xallocator=mimalloc` compiler option from your build scripts. If you encounter any issues, please report them to our issue tracker.

For more information on the memory allocator and garbage collection in Kotlin, see Kotlin/Native memory management.

Kotlin/Wasm

Kotlin/Wasm received multiple updates along with support for incremental compilation.

Support for incremental compilation

Previously, when you changed something in your Kotlin code, the Kotlin/Wasm toolchain had to recompile the entire codebase.

Starting from 2.1.0, incremental compilation is supported for Wasm targets. In development tasks, the compiler now recompiles only files relevant to changes from the last compilation, which noticeably reduces the compilation time.

This change currently doubles the compilation speed, and there are plans to improve it further in future releases.

In the current setup, incremental compilation for Wasm targets is disabled by default. To enable incremental compilation, add the following line to your project's `local.properties` or `gradle.properties` file:

```
# gradle.properties
kotlin.incremental.wasm=true
```

Try out Kotlin/Wasm incremental compilation and share your feedback. Your insights will help make this feature Stable and enabled by default sooner.

Browser APIs moved to the kotlinx-browser stand-alone library

Previously, the declarations for web APIs and related target utilities were part of the Kotlin/Wasm standard library.

In this release, the `org.w3c.*` declarations have been moved from the Kotlin/Wasm standard library to the new kotlinx-browser library. This library also includes other web-related packages, such as `org.khronos.webgl`, `kotlin.dom`, and `kotlinx.browser`.

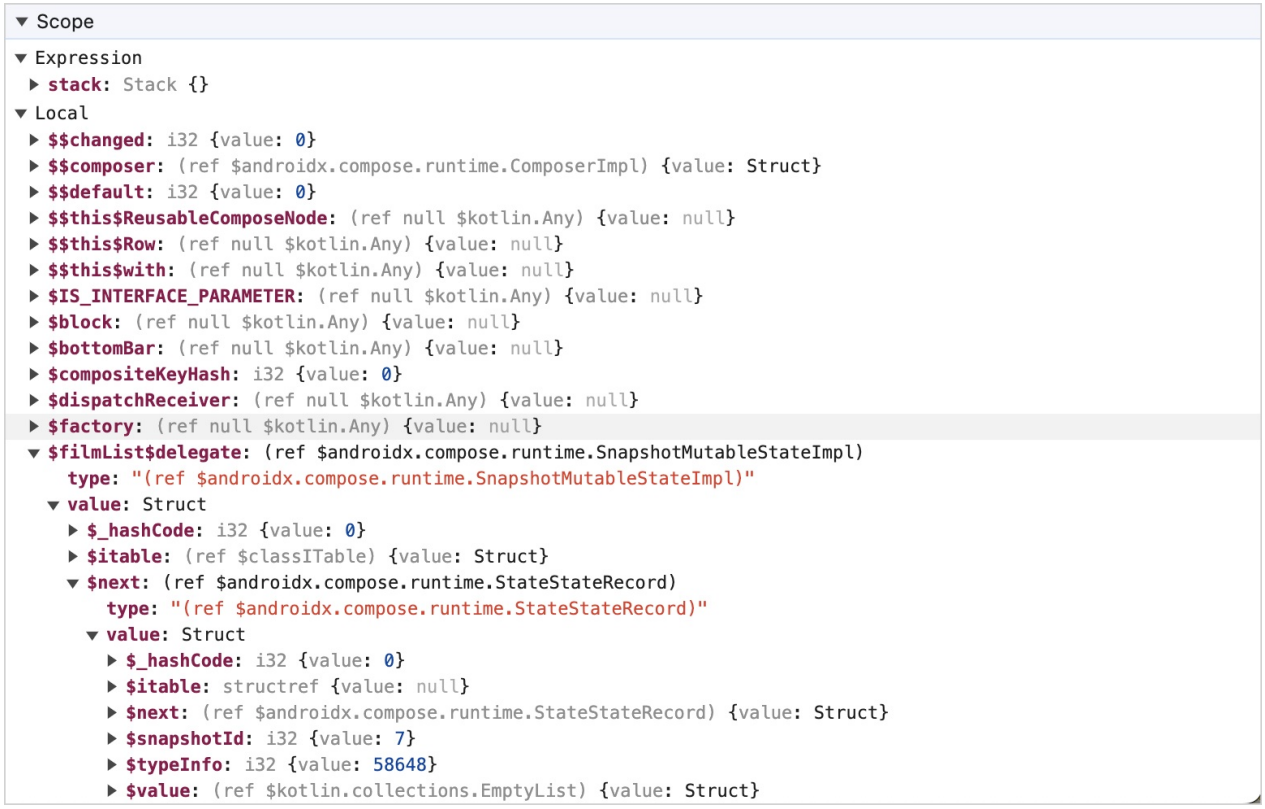
This separation provides modularity, enabling independent updates for web-related APIs outside of Kotlin's release cycle. Additionally, the Kotlin/Wasm standard library now contains only declarations available in any JavaScript environment.

To use the declarations from the moved packages, you need to add the `kotlinx-browser` dependency to your project's build configuration file:

```
// build.gradle.kts
val wasmJsMain by getting {
    dependencies {
        implementation("org.jetbrains.kotlinx:kotlinx-browser:0.3")
    }
}
```

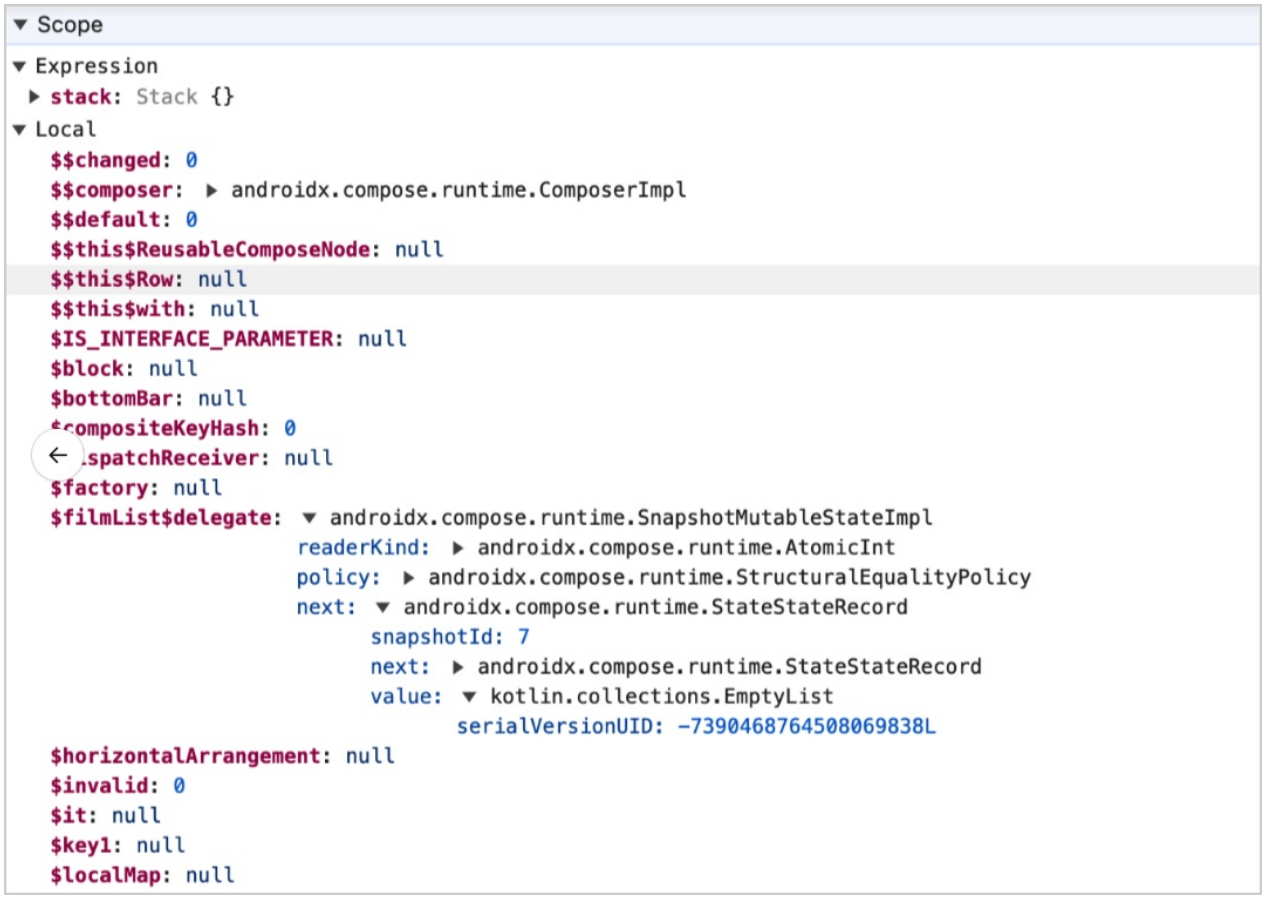
Improved debugging experience for Kotlin/Wasm

Previously, when debugging Kotlin/Wasm code in web browsers, you might have encountered a low-level representation of variable values in the debugging interface. This often made it challenging to track the current state of the application.



To improve this experience, custom formatters have been added in the variable view. The implementation uses the custom formatters API, which is supported across major browsers like Firefox and Chromium-based ones.

With this change, you can now display and locate variable values in a more user-friendly and comprehensible manner.



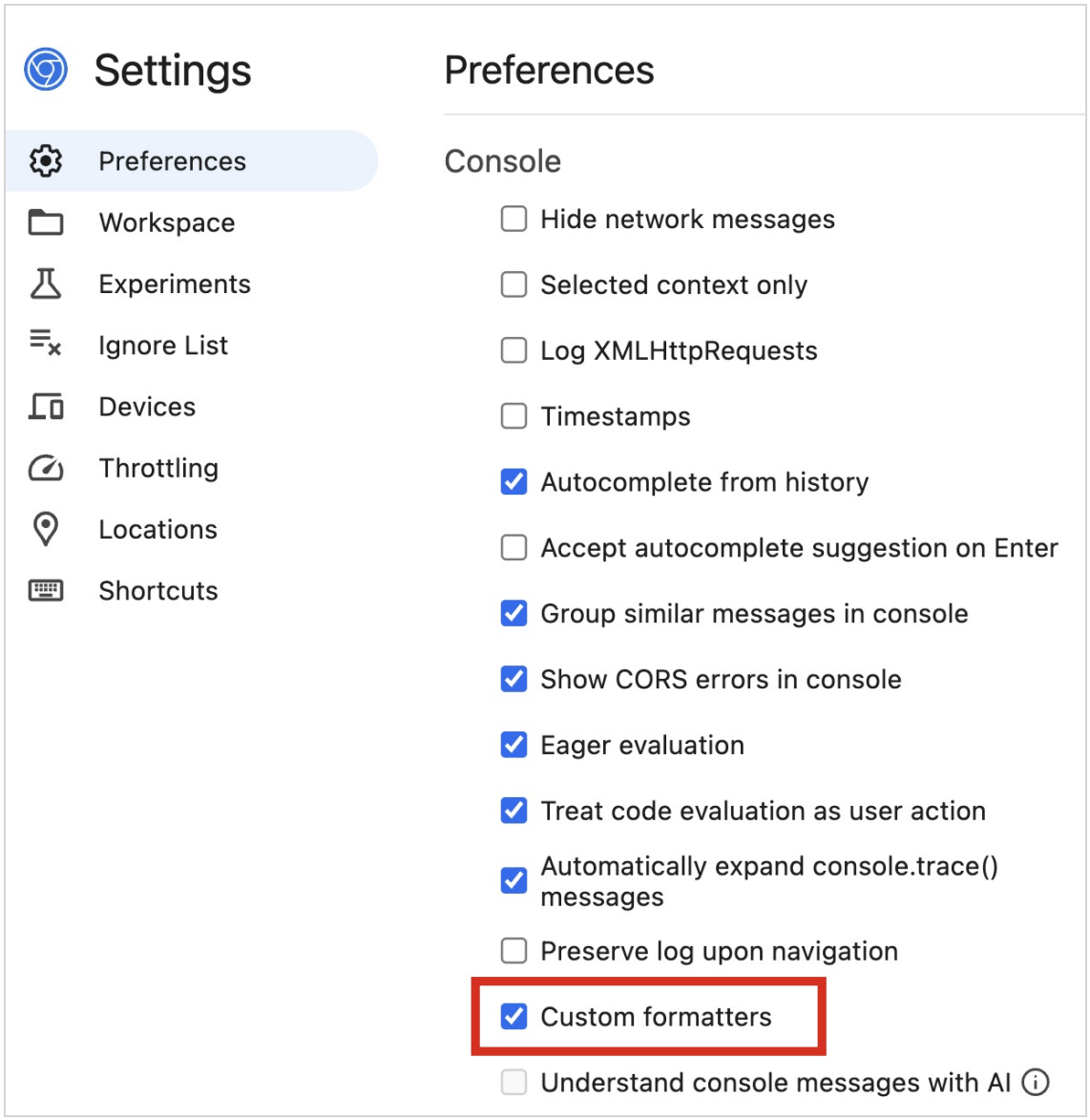
To try the new debugging experience:

1. Add the following compiler option to the `wasmJs {}` compiler options:

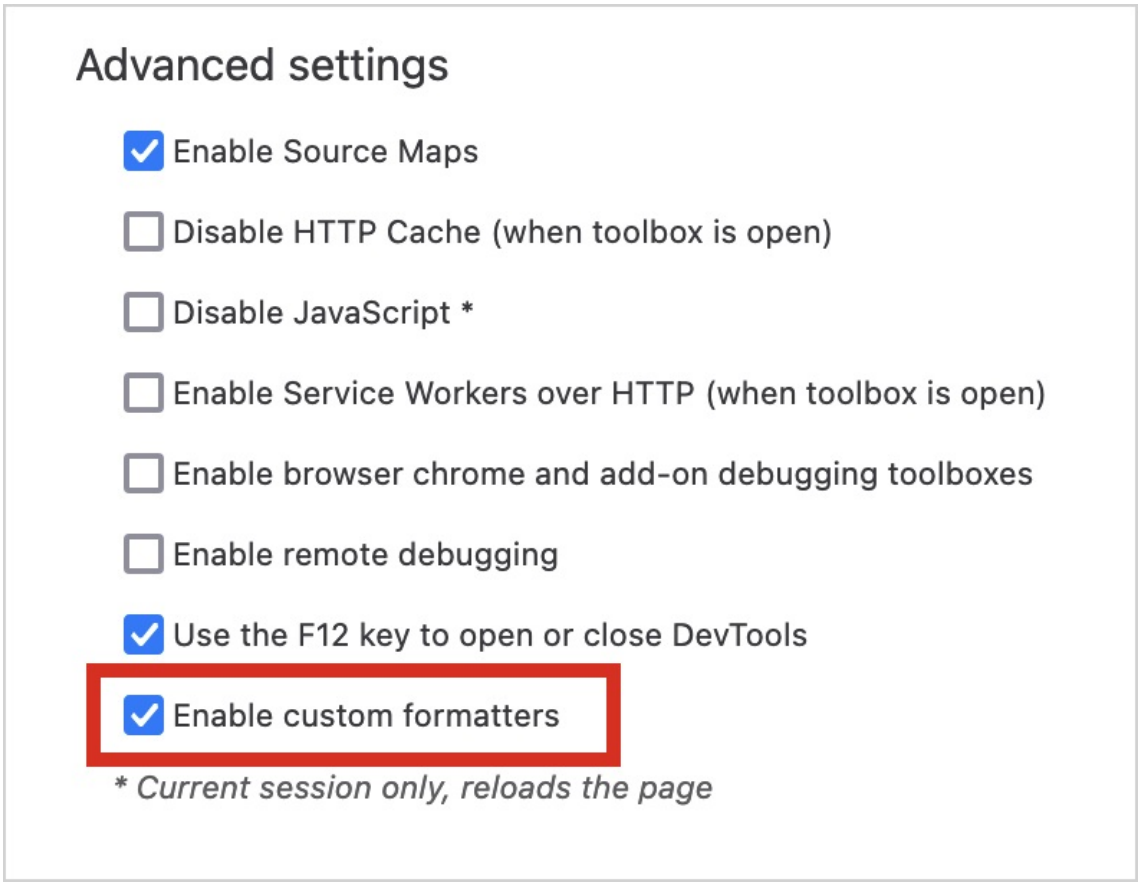
```
// build.gradle.kts
kotlin {
    wasmJs {
        // ...

        compilerOptions {
            freeCompilerArgs.add("-Xwasm-debugger-custom-formatters")
        }
    }
}
```

2. Enable custom formatters in your browser:

    - In Chrome DevTools, it's available via **Settings | Preferences | Console**:



- In Firefox DevTools, it's available via **Settings | Advanced settings**:



Reduced size of Kotlin/Wasm binaries

The size of your Wasm binaries produced by production builds will be reduced by up to 30%, and you may see some performance improvements. This is because the `--closed-world`, `--type-ssa`, and `--type-merging` Binaryen options are now considered safe to use for all Kotlin/Wasm projects and are enabled by default.

Improved JavaScript array interoperability in Kotlin/Wasm

While Kotlin/Wasm's standard library provides the `JsArray<T>` type for JavaScript arrays, there was no direct method to transform `JsArray<T>` into Kotlin's native `Array` or `List` types.

This gap required creating custom functions for array transformations, complicating interoperability between Kotlin and JavaScript code.

This release introduces an adapter function that automatically converts `JsArray<T>` to `Array<T>` and vice versa, simplifying array operations.

Here's an example of conversion between generic types: Kotlin `List<T>` and `Array<T>` to JavaScript `JsArray<T>`.

```
val list: List<JsString> =
    listOf("Kotlin", "Wasm").map { it.toJsString() }
```

```
// Uses .toJsArray() to convert List or Array to JsArray
val jsArray: JsArray<JsString> = list.toJsArray()

// Uses .toArray() and .toList() to convert it back to Kotlin types
val kotlinArray: Array<JsString> = jsArray.toArray()
val kotlinList: List<JsString> = jsArray.toList()
```

Similar methods are available for converting typed arrays to their Kotlin equivalents (for example, `IntArray` and `Int32Array`). For detailed information and implementation, see the `kotlinx-browser` repository.

Here's an example of conversion between typed arrays: Kotlin `IntArray` to JavaScript `Int32Array`.

```
import org.khronos.webgl.*

    // ...

    val intArray: IntArray = intArrayOf(1, 2, 3)

    // Uses .toInt32Array() to convert Kotlin IntArray to JavaScript Int32Array
    val jsInt32Array: Int32Array = intArray.toInt32Array()

    // Uses toIntArray() to convert JavaScript Int32Array back to Kotlin IntArray
    val kotlinIntArray: IntArray = jsInt32Array.toIntArray()
```

Support for accessing JavaScript exception details in Kotlin/Wasm

Previously, when a JavaScript exception occurred in Kotlin/Wasm, the `JsException` type provided only a generic message without details from the original JavaScript error.

Starting from Kotlin 2.1.0, you can configure `JsException` to include the original error message and stack trace by enabling a specific compiler option. This provides more context to help diagnose issues originating from JavaScript.

This behavior depends on the `WebAssembly.JSTag` API, which is available only in certain browsers:

- **Chrome**: Supported from version 115
- **Firefox**: Supported from version 129
- **Safari**: Not yet supported

To enable this feature, which is disabled by default, add the following compiler option to your `build.gradle.kts` file:

```
// build.gradle.kts
```

```
kotlin {
    wasmJs {
        compilerOptions {
            freeCompilerArgs.add("-Xwasm-attach-js-exception")
        }
    }
}
```

Here's an example demonstrating the new behavior:

```
external object JSON {
    fun <T: JsAny> parse(json: String): T
}

fun main() {
    try {
        JSON.parse("an invalid JSON")
    } catch (e: JsException) {
        println("Thrown value is: ${e.thrownValue}")
        // SyntaxError: Unexpected token 'a', "an invalid JSON" is not valid JSON

        println("Message: ${e.message}")
        // Message: Unexpected token 'a', "an invalid JSON" is not valid JSON

        println("Stacktrace:")
        // Stacktrace:

        // Prints the full JavaScript stack trace
        e.printStackTrace()
    }
}
```

With the `-Xwasm-attach-js-exception` option enabled, `JsException` provides specific details from the JavaScript error. Without the option, `JsException` includes only a generic message stating that an exception was thrown while running JavaScript code.

Deprecation of default exports

As part of the migration to named exports, an error was previously printed to the console when a default import was used for Kotlin/Wasm exports in JavaScript.

In 2.1.0, default imports have been completely removed to fully support named exports.

When coding in JavaScript for the Kotlin/Wasm target, you now need to use the corresponding named imports instead of default imports.

This change marks the last phase of a deprecation cycle to migrate to named exports:

In version 2.0.0: A warning message was printed to the console, explaining that exporting entities via default exports is deprecated.

In version 2.0.20: An error occurred, requesting the use of the corresponding named import.

In version 2.1.0: The use of default imports has been completely removed.

Subproject-specific Node.js settings

You can configure Node.js settings for your project by defining properties of the `NodeJsRootPlugin` class for `rootProject`. In 2.1.0, you can configure these settings for each subproject using a new class, `NodeJsPlugin`. Here's an example demonstrating how to set a specific Node.js version for a subproject:

```
// build.gradle.kts
project.plugins.withType<org.jetbrains.kotlin.gradle.targets.js.nodejs.NodeJsPlugin> {
    project.the<org.jetbrains.kotlin.gradle.targets.js.nodejs.NodeJsEnvSpec>().version
 = "22.0.0"
}
```

To use the new class for the entire project, add the same code in the `allprojects {}` block:

```
// build.gradle.kts
allprojects {
    project.plugins.withType<org.jetbrains.kotlin.gradle.targets.js.nodejs.NodeJsPlugi
n> {
        project.the<org.jetbrains.kotlin.gradle.targets.js.nodejs.NodeJsEnvSpec>().ver
sion = "your Node.js version"
    }
}
```

You can also use Gradle convention plugins to apply the settings to a particular set of subprojects.

Kotlin/JS

Support for non-identifier characters in properties

Kotlin/JS previously did not allow using names for test methods with spaces enclosed in backticks.

Similarly, it was not possible to access JavaScript object properties that contained characters not permitted in Kotlin identifiers, such as hyphens or spaces:

```
external interface Headers {
    var accept: String?
```

```
    // Invalid Kotlin identifier due to hyphen
    var `content-length`: String?
}

val headers: Headers = TODO("value provided by a JS library")
val accept = headers.accept
// Causes error due to the hyphen in property name
val length = headers.`content-length`
```

This behavior differed from JavaScript and TypeScript, which allow such properties to be accessed using non-identifier characters.

Starting from Kotlin 2.1.0, this feature is enabled by default. Kotlin/JS now allows you to use the backticks (`` `) and the `` @JsName` annotation to interact with JavaScript properties containing non-identifier characters and to use names for test methods.

Additionally, you can use the `@JsName` and `@JsQualifier` annotations to map Kotlin property names to JavaScript equivalents:

```
object Bar {
    val `property example`: String = "bar"
}

@JsQualifier("fooNamespace")
external object Foo {
    val `property example`: String
}

@JsExport
object Baz {
    val `property example`: String = "bar"
}

fun main() {
    // In JavaScript, this is compiled into Bar.property_example_HASH
    println(Bar.`property example`)
    // In JavaScript, this is compiled into fooNamespace["property example"]
    println(Foo.`property example`)
    // In JavaScript, this is compiled into Baz["property example"]
    println(Baz.`property example`)
}
```

Support for generating ES2015 arrow functions

In Kotlin 2.1.0, Kotlin/JS introduces support for generating ES2015 arrow functions, such as `(a, b) => expression`, instead of anonymous functions.

Using arrow functions can reduce the bundle size of your project, especially when using the experimental `-Xir-generate-inline-anonymous-functions` mode. This also makes the generated code more aligned with modern JS.

This feature is enabled by default when targeting ES2015. Alternatively, you can enable it by using the `-Xes-arrow-functions` command line argument.

Learn more about ES2015 (ECMAScript 2015, ES6) in the official documentation.

Gradle improvements

Kotlin 2.1.0 is fully compatible with Gradle 7.6.3 through 8.6. Gradle versions 8.7 to 8.10 are also supported, with only one exception. If you use the Kotlin Multiplatform Gradle plugin, you may see deprecation warnings in your multiplatform projects calling the `withJava()` function in the JVM target. We plan to fix this issue as soon as possible.

For more information, see the related issue in YouTrack.

You can also use Gradle versions up to the latest Gradle release, but if you do, keep in mind that you might encounter deprecation warnings or some new Gradle features might not work.

Minimum supported AGP version bumped to 7.3.1

Starting with Kotlin 2.1.0, the minimum supported Android Gradle plugin version is 7.3.1.

Minimum supported Gradle version bumped to 7.6.3

Starting with Kotlin 2.1.0, the minimum supported Gradle version is 7.6.3.

New API for Kotlin Gradle plugin extensions

Kotlin 2.1.0 introduces a new API to make it easier to create your own plugins for configuring the Kotlin Gradle plugin. This change deprecates the `KotlinTopLevelExtension` and `KotlinTopLevelExtensionConfig` interfaces and introduces the following interfaces for plugin authors:

Name	Description
`KotlinBaseExtension`	A plugin DSL extension type for configuring common Kotlin JVM, Android, and Multiplatform plugin options for the entire project: • `org.jetbrains.kotlin.jvm` • `org.jetbrains.kotlin.android` • `org.jetbrains.kotlin.multiplatform`

`KotlinJvmExtension`	A plugin DSL extension type for configuring Kotlin **JVM** plugin options for the entire project.
`KotlinAndroidExtension`	A plugin DSL extension type for configuring Kotlin **Android** plugin options for the entire project.

For example, if you want to configure compiler options for both JVM and Android projects, use `KotlinBaseExtension`:

```
configure<KotlinBaseExtension> {
    if (this is HasConfigurableKotlinCompilerOptions<*>) {
        with(compilerOptions) {
            if (this is KotlinJvmCompilerOptions) {
                jvmTarget.set(JvmTarget.JVM_17)
            }
        }
    }
}
```

This configures the JVM target to 17 for both JVM and Android projects.

To configure compiler options specifically for JVM projects, use `KotlinJvmExtension`:

```
configure<KotlinJvmExtension> {
    compilerOptions {
        jvmTarget.set(JvmTarget.JVM_17)
    }

    target.mavenPublication {
        groupId = "com.example"
        artifactId = "example-project"
        version = "1.0-SNAPSHOT"
    }
}
```

This example similarly configures the JVM target to 17 for JVM projects. It also configures a Maven publication for the project so that its output is published to a Maven repository.

You can use the `KotlinAndroidExtension` in exactly the same way.

Compiler symbols hidden from the Kotlin Gradle plugin API

Previously, KGP included `org.jetbrains.kotlin:kotlin-compiler-embeddable` in its runtime dependencies, making internal compiler symbols available in the build script classpath. These symbols were intended for internal use only.

Starting with Kotlin 2.1.0, KGP bundles a subset of `org.jetbrains.kotlin:kotlin-compiler-embeddable` class files in its JAR file and progressively removes them. This change aims to prevent compatibility issues and simplify KGP maintenance.

If other parts of your build logic, such as plugins like `kotlinter`, depend on a different version of `org.jetbrains.kotlin:kotlin-compiler-embeddable` than the one bundled with KGP, it can lead to conflicts and runtime exceptions.

To prevent such issues, KGP now shows a warning if `org.jetbrains.kotlin:kotlin-compiler-embeddable` is present in the build classpath alongside KGP.

As a long-term solution, if you're a plugin author using `org.jetbrains.kotlin:kotlin-compiler-embeddable` classes, we recommend running them in an isolated classloader. For example, you can achieve it using the Gradle Workers API with classloader or process isolation.

Using the Gradle Workers API

This example demonstrates how to safely use the Kotlin compiler in a project producing a Gradle plugin. First, add a compile-only dependency in your build script. This makes the symbol available at compile time only:

```
// build.gradle.kts
dependencies {
    compileOnly("org.jetbrains.kotlin:kotlin-compiler-embeddable:2.1.21")
}
```

Next, define a Gradle work action to print the Kotlin compiler version:

```
import org.gradle.workers.WorkAction
import org.gradle.workers.WorkParameters
import org.jetbrains.kotlin.config.KotlinCompilerVersion
abstract class ActionUsingKotlinCompiler : WorkAction<WorkParameters.None> {
    override fun execute() {
        println("Kotlin compiler version: ${KotlinCompilerVersion.getVersion()}")
    }
}
```

Now create a task that submits this action to the worker executor using classloader isolation:

```
import org.gradle.api.DefaultTask
import org.gradle.api.file.ConfigurableFileCollection
import org.gradle.api.tasks.Classpath
import org.gradle.api.tasks.TaskAction
import org.gradle.workers.WorkerExecutor
import javax.inject.Inject
```

```
abstract class TaskUsingKotlinCompiler: DefaultTask() {
    @get:Inject
    abstract val executor: WorkerExecutor

    @get:Classpath
    abstract val kotlinCompiler: ConfigurableFileCollection

    @TaskAction
    fun compile() {
        val workQueue = executor.classLoaderIsolation {
            classpath.from(kotlinCompiler)
        }
        workQueue.submit(ActionUsingKotlinCompiler::class.java) {}
    }
}
```

Finally, configure the Kotlin compiler classpath in your Gradle plugin:

```
import org.gradle.api.Plugin
import org.gradle.api.Project
abstract class MyPlugin: Plugin<Project> {
    override fun apply(target: Project) {
        val myDependencyScope = target.configurations.create("myDependencyScope")
        target.dependencies.add(myDependencyScope.name, "$KOTLIN_COMPILER_EMBEDDABLE:$
KOTLIN_COMPILER_VERSION")
        val myResolvableConfiguration = target.configurations.create("myResolvable") {
            extendsFrom(myDependencyScope)
        }
        target.tasks.register("myTask", TaskUsingKotlinCompiler::class.java) {
            kotlinCompiler.from(myResolvableConfiguration)
        }
    }

    companion object {
        const val KOTLIN_COMPILER_EMBEDDABLE = "org.jetbrains.kotlin:kotlin-compiler-e
mbeddable"
        const val KOTLIN_COMPILER_VERSION = "2.1.21"
    }
}
```

Compose compiler updates

Support for multiple stability configuration files

The Compose compiler can interpret multiple stability configuration files, but the `stabilityConfigurationFile` option of the Compose Compiler Gradle plugin previously allowed for only a single file to be specified. In Kotlin 2.1.0, this functionality was reworked to allow you to

use several stability configuration files for a single module:

- The `stabilityConfigurationFile` option is deprecated.
- There is a new option, `stabilityConfigurationFiles`, with the type `ListProperty<RegularFile>`.

Here's how to pass several files to the Compose compiler using the new option:

```
// build.gradle.kt
composeCompiler {
    stabilityConfigurationFiles.addAll(
        project.layout.projectDirectory.file("configuration-file1.conf"),
        project.layout.projectDirectory.file("configuration-file2.conf"),
    )
}
```

Pausable composition

Pausable composition is a new Experimental feature that changes how the compiler generates skippable functions. With this feature enabled, composition can be suspended on skipping points during runtime, allowing long-running composition processes to be split across multiple frames. Pausable composition is used in lazy lists and other performance-intensive components for prefetching content that might cause frames to drop when executed in a blocking manner.

To try out pausable composition, add the following feature flag in the Gradle configuration for the Compose compiler:

```
// build.gradle.kts
composeCompiler {
    featureFlags = setOf(
        ComposeFeatureFlag.PausableComposition
    )
}
```

> Runtime support for this feature was added in the 1.8.0-alpha02 version of `androidx.compose.runtime`. The feature flag has no effect when used with older runtime versions.
>
> 

Changes to open and overridden @Composable functions

Virtual (open, abstract, and overridden) `@Composable` functions can no longer be restartable. The codegen for restartable groups was generating calls that did not work correctly with inheritance, resulting in runtime crashes.

This means that virtual functions won't be restarted or skipped: whenever their state is invalidated, runtime will recompose their parent composable instead. If your code is sensitive to recompositions, you may notice changes in runtime behavior.

Performance improvements

The Compose compiler used to create a full copy of module's IR to transform `@Composable` types. Apart from increased memory consumption when copying elements that were not related to Compose, this behavior was also breaking downstream compiler plugins in certain edge cases.

This copy operation was removed, resulting in potentially faster compilation times.

Standard library

Changes to the deprecation severity of standard library APIs

In Kotlin 2.1.0, we are raising the deprecation severity level of several standard library APIs from warning to error. If your code relies on these APIs, you need to update it to ensure compatibility. The most notable changes include:

- **Locale-sensitive case conversion functions for `Char` and `String` are deprecated:** Functions like `Char.toLowerCase()`, `Char.toUpperCase()`, `String.toUpperCase()`, and `String.toLowerCase()` are now deprecated, and using them results in an error. Replace them with locale-agnostic function alternatives or other case conversion mechanisms. If you want to continue using the default locale, replace calls like `String.toLowerCase()` with `String.lowercase(Locale.getDefault())`, explicitly specifying the locale. For a locale-agnostic conversion, replace them with `String.lowercase()`, which uses the invariant locale by default.
- **Kotlin/Native freezing API is deprecated:** Using the freezing-related declarations previously marked with the `@FreezingIsDeprecated` annotation now results in an error. This change reflects the transition from the legacy memory manager in Kotlin/Native, which required freezing objects to share them between threads. To learn how to migrate from freezing-related APIs in the new memory model, see the Kotlin/Native migration guide. For more information, see the announcement about the deprecation of freezing.

- **`appendln()` is deprecated in favor of `appendLine()`:** The `StringBuilder.appendln()` and `Appendable.appendln()` functions are now deprecated, and using them results in an error. To replace them, use the `StringBuilder.appendLine()` or `Appendable.appendLine()` functions instead. The `appendln()` function is deprecated because, on Kotlin/JVM, it uses the `line.separator` system property, which has a different default value on each OS. On Kotlin/JVM, this property defaults to `\r\n` (CR LF) on Windows and `\n` (LF) on other systems. On the other hand, the `appendLine()` function consistently uses `\n` (LF) as the line separator, ensuring consistent behavior across platforms.

For a complete list of affected APIs in this release, see the KT-71628 YouTrack issue.

Stable file tree traversal extensions for java.nio.file.Path

Kotlin 1.7.20 introduced Experimental extension functions for the `java.nio.file.Path` class, which allows you to walk through a file tree. In Kotlin 2.1.0, the following file tree traversal extensions are now Stable:

- `walk()` lazily traverses the file tree rooted at the specified path.
- `fileVisitor()` makes it possible to create a `FileVisitor` separately. `FileVisitor` specifies the actions to be performed on directories and files during traversal.
- `visitFileTree(fileVisitor: FileVisitor, ...)` traverses through a file tree, invoking the specified `FileVisitor` on each encountered entry, and it uses the `java.nio.file.Files.walkFileTree()` function under the hood.
- `visitFileTree(..., builderAction: FileVisitorBuilder.() -> Unit)` creates a `FileVisitor` with the provided `builderAction` and calls the `visitFileTree(fileVisitor, ...)` function.
- `sealed interface FileVisitorBuilder` allows you to define a custom `FileVisitor` implementation.
- `enum class PathWalkOption` provides traversal options for the `Path.walk()` function.

The examples below demonstrate how to use these file traversal APIs to create custom `FileVisitor` behaviors, which allows you to define specific actions for visiting files and directories.

For instance, you can explicitly create a `FileVisitor` and use it later:

```
val cleanVisitor = fileVisitor {
    onPreVisitDirectory { directory, attributes ->
        // Placeholder: Add logic on visiting directories
        FileVisitResult.CONTINUE
    }

    onVisitFile { file, attributes ->
        // Placeholder: Add logic on visiting files
        FileVisitResult.CONTINUE
    }
```

```
}

// Placeholder: Add logic here for general setup before traversal
projectDirectory.visitFileTree(cleanVisitor)
```

You can also create a `FileVisitor` with the `builderAction` and use it immediately for the traversal:

```
projectDirectory.visitFileTree {
    // Defines the builderAction:
    onPreVisitDirectory { directory, attributes ->
        // Some logic on visiting directories
        FileVisitResult.CONTINUE
    }

    onVisitFile { file, attributes ->
        // Some logic on visiting files
        FileVisitResult.CONTINUE
    }
}
```

Additionally, you can traverse a file tree rooted at the specified path with the `walk()` function:

```
fun traverseFileTree() {
    val cleanVisitor = fileVisitor {
        onPreVisitDirectory { directory, _ ->
            if (directory.name == "build") {
                directory.toFile().deleteRecursively()
                FileVisitResult.SKIP_SUBTREE
            } else {
                FileVisitResult.CONTINUE
            }
        }

        // Deletes files with the .class extension
        onVisitFile { file, _ ->
            if (file.extension == "class") {
                file.deleteExisting()
            }
            FileVisitResult.CONTINUE
        }
    }

    // Sets up the root directory and files
    val rootDirectory = createTempDirectory("Project")

    // Creates the src directory with A.kt and A.class files
    rootDirectory.resolve("src").let { srcDirectory ->
        srcDirectory.createDirectory()
        srcDirectory.resolve("A.kt").createFile()
```

```
        srcDirectory.resolve("A.class").createFile()
    }

    // Creates the build directory with a Project.jar file
    rootDirectory.resolve("build").let { buildDirectory ->
        buildDirectory.createDirectory()
        buildDirectory.resolve("Project.jar").createFile()
    }

    // Uses the walk() function:
    val directoryStructure = rootDirectory.walk(PathWalkOption.INCLUDE_DIRECTORIES)
        .map { it.relativeTo(rootDirectory).toString() }
        .toList().sorted()
    println(directoryStructure)
    // "[, build, build/Project.jar, src, src/A.class, src/A.kt]"

    // Traverses the file tree with cleanVisitor, applying the rootDirectory.visitFile
Tree(cleanVisitor) cleanup rules
    val directoryStructureAfterClean = rootDirectory.walk(PathWalkOption.INCLUDE_DIREC
TORIES)
        .map { it.relativeTo(rootDirectory).toString() }
        .toList().sorted()
    println(directoryStructureAfterClean)
    // "[, src, src/A.kt]"
}
```

Documentation updates

The Kotlin documentation has received some notable changes:

Language concepts

- Improved Null safety page – Learn how to handle `null` values safely in your code.
- Improved Objects declarations and expressions page – Learn how to define a class and create an instance in a single step.
- Improved When expressions and statements section – Learn about the `when` conditional and how you can use it.
- Updated Kotlin roadmap, Kotlin evolution principles, and Kotlin language features and proposals pages – Learn about Kotlin's plans, ongoing developments, and guiding principles.

Compose compiler

- Compose compiler documentation now located in the Compiler and plugins section – Learn about the Compose compiler, the compiler options, and the steps to migrate.

API references

- New Kotlin Gradle plugins API reference – Explore the API references for the Kotlin Gradle plugin and the Compose compiler Gradle plugin.

Multiplatform development

- New Building a Kotlin library for multiplatform page – Learn how to design your Kotlin libraries for Kotlin Multiplatform.
- New Introduction to Kotlin Multiplatform page – Learn about Kotlin Multiplatform's key concepts, dependencies, libraries, and more.
- Updated Kotlin Multiplatform overview page – Navigate through the essentials of Kotlin Multiplatform and popular use cases.
- New iOS integration section – Learn how to integrate a Kotlin Multiplatform shared module into your iOS app.
- New Kotlin/Native's definition file page – Learn how to create a definition file to consume C and Objective-C libraries.
- Get started with WASI – Learn how to run a simple Kotlin/Wasm application using WASI in various WebAssembly virtual machines.

Tooling

- New Dokka migration guide – Learn how to migrate to Dokka Gradle plugin v2.

Compatibility guide for Kotlin 2.1.0

Kotlin 2.1.0 is a feature release and can, therefore, bring changes that are incompatible with your code written for earlier versions of the language. Find the detailed list of these changes in the Compatibility guide for Kotlin 2.1.0.

Install Kotlin 2.1.0

Starting from IntelliJ IDEA 2023.3 and Android Studio Iguana (2023.2.1) Canary 15, the Kotlin plugin is distributed as a bundled plugin included in your IDE. This means that you can't install the plugin from JetBrains Marketplace anymore.

To update to the new Kotlin version, change the Kotlin version to 2.1.0 in your build scripts.

What's new in Kotlin 2.2.0-RC3

Released: 2025-06-16

> This document doesn't cover all of the features of the Early Access Preview (EAP) release, but it highlights some major improvements.
>
> See the full list of changes in the GitHub changelog.
>
> 

The Kotlin 2.2.0-RC3 release is out! Here are some details of this EAP release:

- Language: preview of context parameters
- Kotlin compiler: unified management of compiler warnings
- Kotlin/JVM: changes to default method generation for interface functions
- Gradle: integration of Problems API within KGP diagnostics and KGP compatibility with '--warning-mode'

IDE support

The Kotlin plugins that support 2.2.0-RC3 are bundled in the latest IntelliJ IDEA and Android Studio. You don't need to update the Kotlin plugin in your IDE. All you need to do is to change the Kotlin version to 2.2.0-RC3 in your build scripts.

See Update to a new release for details.

Language

This release promotes some language features as stable and brings context parameters in preview.

Stable features: guard conditions, non-local break and continue, and multi-dollar interpolation

In Kotlin 2.1.0, several new language features were introduced in preview. We're happy to announce that these language features became Stable in this release:

- Guard conditions in `when` with a subject
- Non-local `break` and `continue`
- Multi-dollar interpolation: improved handling of `$` in string literals

See the full list of Kotlin language design features and proposals.

Preview of context parameters

In this release, context parameters are introduced in preview. Context parameters allow functions and properties to declare dependencies that are implicitly available in the surrounding context.

This feature replaces an older experimental feature called context receivers. To migrate from context receivers to context parameters, you can use assisted support in IntelliJ IDEA, as described in the blog post.

How to declare context parameters

You can declare context parameters for properties and functions using the `context` keyword followed by a list of parameters, each of the form `name: Type`. Here is an example with a dependency on the `UserService` interface:

```
// `UserService` defines the dependency required in context
interface UserService {
    fun log(message: String)
    fun findUserById(id: Int): String
}

// Declares a function with a context parameter
context(users: UserService)
fun outputMessage(message: String) {
    // Uses `log` from the context
    users.log("Log: $message")
}

// Declares a property with a context parameter
context(users: UserService)
val firstUser: String
    // Uses `findUserById` from the context
    get() = users.findUserById(1)
```

You can use `_` as a context parameter name. In this case, the parameter's value is available for resolution but is not accessible by name inside the block:

```
// Uses `_` as context parameter name
context(_: UserService)
fun logWelcome() {
    // Resolution still finds the appropriate `log` function from UserService
    outputMessage("Welcome!")
}
```

Context parameters resolution

Kotlin resolves context parameters at the call site by searching for matching context values in the current scope. Kotlin matches them by their type. If multiple compatible values exist at the same scope level, the compiler reports an ambiguity:

```
// `UserService` defines the dependency required in context
interface UserService {
    fun log(message: String)
}

// Declares a function with a context parameter
context(users: UserService)
fun outputMessage(message: String) {
    users.log("Log: $message")
}

fun main() {
    // Implements `UserService`
    val serviceA = object : UserService {
        override fun log(message: String) = println("A: $message")
    }

    // Implements `UserService`
    val serviceB = object : UserService {
        override fun log(message: String) = println("B: $message")
    }

    // Both `serviceA` and `serviceB` match the expected `UserService` type at the cal
l site
    context(serviceA, serviceB) {
        outputMessage("This will not compile")
        // Ambiguity error
    }
}
```

Restrictions

Context parameters are in continuous improvement; some of the current restrictions are:

- Constructors cannot declare context parameters
- Properties with context parameters can't have backing fields or initializers
- Properties with context parameters can't use delegation

However, context parameters in Kotlin represent a significant improvement in managing dependencies through simplified dependency injection, improved DSL design, and scoped operations. For more information, see the feature's KEEP.

How to enable context parameters

To enable context parameters in your project, use the following compiler option in the command line:

```
-Xcontext-parameters
```

Or add it to the `compilerOptions {}` block of your Gradle build file:

```
// build.gradle.kts
kotlin {
    compilerOptions {
        freeCompilerArgs.add("-Xcontext-parameters")
    }
}
```

> Specifying both `-Xcontext-receivers` and `-Xcontext-parameters` compiler options simultaneously leads to an error.
>
> 

Leave your feedback

This feature is planned to be stabilized and improved in future Kotlin releases. We would appreciate your feedback in our issue tracker YouTrack.

Kotlin compiler: unified management of compiler warnings

Kotlin 2.2.0-RC3 introduces a new compiler option, `-Xwarning-level`. It's designed to provide a unified way of managing compiler warnings in Kotlin projects.

Previously, you could only apply general module-wide rules, like disabling all warnings with `-nowarn`, turning all warnings to compilation errors with `-Werror`, or enabling additional compiler checks with `-Wextra`. The only option to adjust them for specific warnings was the `-Xsuppress-warning` option.

With the new solution, you can override general rules and exclude specific diagnostics in a consistent way.

How to apply

The new compiler option has the following syntax:

```
-Xwarning-level=DIAGNOSTIC_NAME:(error|warning|disabled)
```

- `error` : raises the specified warning to an error.
- `warning` : emits a warning and is enabled by default.
- `disabled` : completely suppresses the specified warning module-wide.

Keep in mind that you can only configure the severity level of *warnings* with the new compiler option.

Use cases

With the new solution, you can better fine-tune warning reporting in your project by combining general rules with specific ones. Choose your use case:

Suppress warnings

Command	Description
`-nowarn`	Suppresses all warnings during compilation.
`-Xwarning-level=DIAGNOSTIC_NAME:disabled`	Suppresses only specified warnings. Works the same as `-Xsuppress-warning` .
`-nowarn -Xwarning-level=DIAGNOSTIC_NAME:warning`	Suppresses all warnings except for the specified ones.

Raise warnings to errors

Command	Description
`-Werror`	Raises all warnings to compilation errors.
`-Xwarning-level=DIAGNOSTIC_NAME:error`	Raises only specified warnings to errors.
`-Werror -Xwarning-level=DIAGNOSTIC_NAME:warning`	Raises all warnings to errors except for the specified ones.

Enable additional compiler warnings

Command	Description
`-Wextra`	Enables all additional declaration, expression, and type compiler checks that emit warnings if true.
`-Xwarning-level=DIAGNOSTIC_NAME:warning`	Enables only specified additional compiler checks.
`-Wextra -Xwarning-level=DIAGNOSTIC_NAME:disabled`	Enables all additional checks except for the specified ones.

Warning lists

In case you have many warnings you want to exclude from general rules, you can list them in a separate file through `@argfile`.

Leave feedback

The new compiler option is still Experimental. Please report any problems to our issue tracker, YouTrack.

Kotlin/JVM

Changes to default method generation for interface functions

Starting from Kotlin 2.2.0-RC3, functions declared in interfaces are compiled to JVM default methods unless configured otherwise. This change affects how Kotlin's interface functions with implementations are compiled to bytecode. This behavior is controlled by the new stable compiler option `-jvm-default`, replacing the deprecated `-Xjvm-default` option.

You can control the behavior of the `-jvm-default` option using the following values:

- `enable` (default): generates default implementations in interfaces and includes bridge functions in subclasses and `DefaultImpls` classes. Use this mode to maintain binary compatibility with older Kotlin versions.
- `no-compatibility`: generates only default implementations in interfaces. This mode skips compatibility bridges and `DefaultImpls` classes, making it suitable for new code.
- `disable`: disables default implementations in interfaces. Only bridge functions and `DefaultImpls` classes are generated, matching the behavior before Kotlin 2.2.0-RC3.

To configure the `-jvm-default` compiler option, set the `jvmDefault` property in your Gradle Kotlin DSL:

```
kotlin {
  compilerOptions {
    jvmDefault = JvmDefaultMode.NO_COMPATIBILITY
  }
}
```

Support for reading and writing annotations in Kotlin metadata

Previously, you had to read annotations from compiled JVM class files using reflection or bytecode analysis and manually match them to metadata entries based on signatures. This process was error-prone, especially for overloaded functions.

Now, in Kotlin 2.2.0-RC3, the Kotlin Metadata JVM library introduces support for reading annotations stored in Kotlin metadata.

To make annotations available in the metadata for your compiled files, add the following compiler option:

```
-Xannotations-in-metadata
```

Alternatively, add it to the `compilerOptions {}` block of your Gradle build file:

```
// build.gradle.kts
kotlin {
    compilerOptions {
        freeCompilerArgs.add("-Xannotations-in-metadata")
    }
}
```

With this option enabled, the Kotlin compiler writes annotations into metadata alongside the JVM bytecode, making them accessible to the `kotlin-metadata-jvm` library.

The library provides the following APIs for accessing annotations:

- `KmClass.annotations`
- `KmFunction.annotations`
- `KmProperty.annotations`
- `KmConstructor.annotations`
- `KmPropertyAccessorAttributes.annotations`
- `KmValueParameter.annotations`
- `KmFunction.extensionReceiverAnnotations`
- `KmProperty.extensionReceiverAnnotations`
- `KmProperty.backingFieldAnnotations`
- `KmProperty.delegateFieldAnnotations`
- `KmEnumEntry.annotations`

These APIs are Experimental. To opt in, use the `@OptIn(ExperimentalAnnotationsInMetadata::class)` annotation.

Here's an example of reading annotations from Kotlin metadata:

```
@file:OptIn(ExperimentalAnnotationsInMetadata::class)
```

```
import kotlin.metadata.ExperimentalAnnotationsInMetadata
import kotlin.metadata.jvm.KotlinClassMetadata

annotation class Label(val value: String)

@Label("Message class")
class Message

fun main() {
    val metadata = Message::class.java.getAnnotation(Metadata::class.java)
    val kmClass = (KotlinClassMetadata.readStrict(metadata) as KotlinClassMetadata.Cla
ss).kmClass
    println(kmClass.annotations)
    // [@Label(value = StringValue("Message class"))]
}
```

> If you use the `kotlin-metadata-jvm` library in your projects, we recommend testing and updating your code to support annotations. Otherwise, when annotations in metadata become enabled by default in a future Kotlin version, your projects may produce invalid or incomplete metadata.
>
> If you experience any problems, please report them in our issue tracker.
>
> 

Kotlin/Native

Per-object memory allocation

Kotlin/Native's memory allocator can now reserve memory on a per-object basis. In some cases, it may help you avoid strict memory limitations or high memory consumption on the application's startup.

The new feature is designed to replace the `-Xallocator=std` compiler option that enabled the system memory allocator instead of the default one. Now you can disable buffering (paging of allocations) without switching memory allocators.

The feature is currently Experimental. To enable it, set the following option in your `gradle.properties` file:

```
kotlin.native.binary.pagedAllocator=false
```

Please report any problems to our issue tracker YouTrack.

LLVM update from 16 to 19

In Kotlin 2.2.0-RC3, we updated LLVM from version 16 to 19. The new version includes performance improvements, bug fixes, and security updates.

This update shouldn't affect your code, but if you encounter any issues, please report them to our issue tracker.

Kotlin/Wasm: wasmJs target separated from js target

Before, the `wasmJs` target shared the same infrastructure as the `js` target. As a result, both targets were hosted in the same directory (`build/js`) and used the same NPM tasks and configurations.

Now, the `wasmJs` target has its own infrastructure separate from the `js` target. This allows the Wasm tasks and types to be distinct from the JavaScript ones, enabling independent configuration.

Additionally, the Wasm-related project files and NPM dependencies are now in a separate `build/wasm` directory.

New NPM-related tasks have been introduced for Wasm, while existing JavaScript tasks are now dedicated only to JavaScript:

Wasm tasks	JavaScript tasks
`kotlinWasmNpmInstall`	`kotlinNpmInstall`
`wasmRootPackageJson`	`rootPackageJson`

Similarly, new Wasm-specific declarations are introduced:

Wasm declarations	JavaScript declarations
`WasmNodeJsRootPlugin`	`NodeJsRootPlugin`
`WasmNodeJsPlugin`	`NodeJsPlugin`
`WasmYarnPlugin`	`YarnPlugin`
`WasmNodeJsRootExtension`	`NodeJsRootExtension`
`WasmNodeJsEnvSpec`	`NodeJsEnvSpec`
`WasmYarnRootEnvSpec`	`YarnRootEnvSpec`

You can now work with the Wasm target independently of the JavaScript target, which simplifies the configuration.

This change is enabled by default and requires no additional configuration.

Kotlin/JS

Fix for copy() in @JsPlainObject interfaces

Kotlin/JS has an experimental plugin called `js-plain-objects`, which introduced a `copy()` function for interfaces annotated with `@JsPlainObject`. You can use the `copy()` function to manipulate objects.

However, the initial implementation of `copy()` was not compatible with inheritance, and this caused issues when a `@JsPlainObject` interface extended other interfaces.

To avoid limitations on plain objects, the `copy()` function has been moved from the object itself to its companion object:

```
@JsPlainObject
external interface User {
    val name: String
    val age: Int
}

fun main() {
    val user = User(name = "SomeUser", age = 21)
    // This syntax is not valid anymore
    val copy = user.copy(age = 35)
    // This is the correct syntax
    val copy = User.copy(user, age = 35)
}
```

This change resolves conflicts in the inheritance hierarchy and eliminates ambiguity. It is enabled by default starting from Kotlin 2.2.0-RC3.

Support for typealiases in files with @JsModule annotation

Previously, files annotated with `@JsModule` to import declarations from JavaScript modules were restricted to external declarations only. Meaning, you couldn't declare a `typealias` in such files.

Starting with Kotlin 2.2.0-RC3, you can declare typealiases inside files marked with `@JsModule`:

```
@file:JsModule("somepackage")
package somepackage
typealias SomeClass = Any
```

This change reduces an aspect of Kotlin/JS interoperability limitations, and more improvements are planned for future releases.

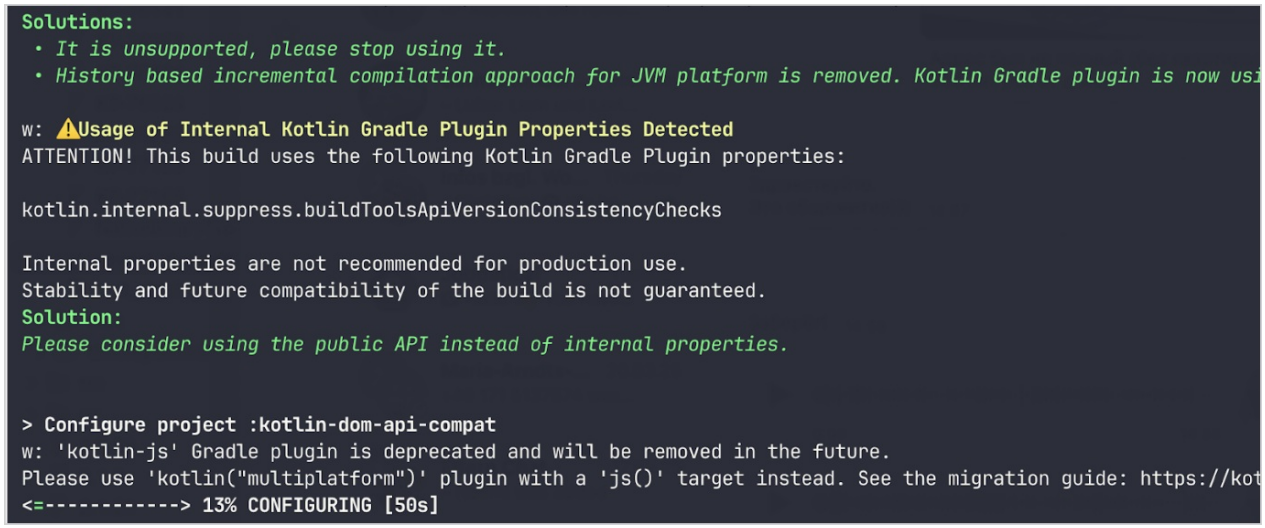
Support for typealiases in files with `@JsModule` is enabled by default.

Gradle

Kotlin 2.2.0-RC3 is fully compatible with Gradle 7.6.3 through 8.14. You can also use Gradle versions up to the latest Gradle release. However, be aware that doing so may result in deprecation warnings, and some new Gradle features might not work.

Support for rich output in console for Kotlin Gradle plugin

In Kotlin 2.2.0-RC3, we support color and other rich output in the console during the Gradle build process, making it easier to read and understand the reported diagnostics. Rich output is available in supported terminal emulators for Linux and macOS. We're working on adding support for Windows.



This feature is enabled by default, but if you want to override it, add the following Gradle property to your `gradle.properties` file:

```
org.gradle.console=plain
```

For more information about this property and its options, see Gradle's documentation on Customizing log format.

Integration of Problems API within KGP diagnostics

Previously, the Kotlin Gradle Plugin (KGP) reported diagnostics—such as warnings and errors—only as plain text output to the console or logs.

Starting with 2.2.0-RC3, KGP introduces an additional reporting mechanism: it now uses Gradle's Problems API, a standardized way to report rich, structured problem information during the build process.

KGP diagnostics are now easier to read and more consistently displayed across different interfaces like the Gradle CLI and IntelliJ IDEA.

This integration is enabled by default, starting with Gradle 8.6 or later. As the API is still evolving, use the most recent Gradle version to benefit from the latest improvements.

KGP compatibility with '--warning-mode'

The Kotlin Gradle Plugin (KGP) diagnostics reported issues using fixed severity levels, meaning Gradle's `--warning-mode` command-line option had no effect on how KGP displayed errors.

Now, KGP diagnostics are compatible with the `--warning-mode` option, providing more flexibility. For example, you can convert all warnings into errors or disable warnings entirely.

With this change, KGP diagnostics adjust the output based on the selected warning mode:

- When you set `--warning-mode=fail`, diagnostics with `Severity.Warning` are now elevated to `Severity.Error`.
- When you set `--warning-mode=none`, diagnostics with `Severity.Warning` are not logged.

This behavior is enabled by default starting with 2.2.0-RC3.

To ignore the `--warning-mode` option, set `kotlin.internal.diagnostics.ignoreWarningMode=true` in your Gradle properties.

Kotlin standard library: Stable Base64 and HexFormat APIs

In Kotlin 2.2.0-RC3, the `Base64` API and `HexFormat` API are now Stable.

Base64 encoding and decoding

Kotlin 1.8.20 introduced Experimental support for Base64 encoding and decoding. In Kotlin 2.2.0-RC3, the Base64 API is now Stable and includes four encoding schemes, with the new `Base64.Pem` added in this release:

- `Base64.Default` uses the standard Base64 encoding scheme.

> The `Base64.Default` is the companion object of the `Base64` class. As a result, you can call its functions with `Base64.encode()` and `Base64.decode()` instead of `Base64.Default.encode()` and `Base64.Default.decode()`.
>
> 

- `Base64.UrlSafe` uses the "URL and Filename safe" encoding scheme.
- `Base64.Mime` uses the MIME encoding scheme, inserting a line separator every 76 characters during encoding and skipping illegal characters during decoding.
- `Base64.Pem` encodes data like `Base64.Mime` but limits the line length to 64 characters.

You can use the Base64 API to encode binary data into a Base64 string and decode it back into bytes.

Here's an example:

```
val foBytes = "fo".map { it.code.toByte() }.toByteArray()
Base64.Default.encode(foBytes) // "Zm8="
// Alternatively:
// Base64.encode(foBytes)

val foobarBytes = "foobar".map { it.code.toByte() }.toByteArray()
Base64.UrlSafe.encode(foobarBytes) // "Zm9vYmFy"

Base64.Default.decode("Zm8=") // foBytes
// Alternatively:
// Base64.decode("Zm8=")

Base64.UrlSafe.decode("Zm9vYmFy") // foobarBytes
```

On the JVM, use the `.encodingWith()` and `.decodingWith()` extension functions to encode and decode Base64 with input and output streams:

```
import kotlin.io.encoding.*
import java.io.ByteArrayOutputStream

fun main() {
    val output = ByteArrayOutputStream()
    val base64Output = output.encodingWith(Base64.Default)

    base64Output.use { stream ->
        stream.write("Hello World!!".encodeToByteArray())
    }

    println(output.toString())
    // SGVsbG8gV29ybGQhIQ==
}
```

Hexadecimal parsing and formatting with the HexFormat API

The `HexFormat` API introduced in Kotlin 1.9.0 is now Stable. You can use it to convert between numerical values and hexadecimal strings.

For example:

```
fun main() {
    //sampleStart
    println(93.toHexString())
    //sampleEnd
}
```

For more information, see New HexFormat class to format and parse hexadecimals.

早期版本

- Kotlin 2.0.20
- Kotlin 2.0.0
- Kotlin 1.9.20
- Kotlin 1.9.0
- Kotlin 1.8.20
- Kotlin 1.8.0
- Kotlin 1.7.20
- Kotlin 1.7.0
- Kotlin 1.6.20
- Kotlin 1.6.0
- Kotlin 1.5.30
- Kotlin 1.5.20
- Kotlin 1.5.0
- Kotlin 1.4.30
- Kotlin 1.4.20
- Kotlin 1.4.0
- Kotlin 1.3
- Kotlin 1.2
- Kotlin 1.1

Kotlin 2.0.20 的新特性

发布于：*2024-08-22*

The Kotlin 2.0.20 release is out! This version includes performance improvements and bug fixes for Kotlin 2.0.0, where we announced the Kotlin K2 compiler as Stable. Here are some additional highlights from this release:

- The data class copy function to have the same visibility as the constructor
- Static accessors for source sets from the default target hierarchy are now available in multiplatform projects
- Concurrent marking for Kotlin/Native has been made possible in the garbage collector
- The `@ExperimentalWasmDsl` annotation in Kotlin/Wasm has a new location
- Support has been added for Gradle versions 8.6–8.8
- A new option allows sharing JVM artifacts between Gradle projects as class files
- The Compose compiler has been updated
- Support for UUIDs has been added to the common Kotlin standard library

IDE support

The Kotlin plugins that support 2.0.20 are bundled in the latest IntelliJ IDEA and Android Studio. You don't need to update the Kotlin plugin in your IDE. All you need to do is to change the Kotlin version to 2.0.20 in your build scripts.

See Update to a new release for details.

Language

Kotlin 2.0.20 begins to introduce changes to improve consistency in data classes and replace the Experimental context receivers feature.

Data class copy function to have the same visibility as constructor

Currently, if you create a data class using a `private` constructor, the automatically generated `copy()` function doesn't have the same visibility. This can cause problems later in your code. In future Kotlin releases, we will introduce the behavior that the default visibility of the `copy()` function is the same as the constructor. This change will be introduced gradually to help you migrate your code as smoothly as possible.

Our migration plan starts with Kotlin 2.0.20, which issues warnings in your code where the visibility will change in the future. For example:

```
// Triggers a warning in 2.0.20
data class PositiveInteger private constructor(val number: Int) {
    companion object {
        fun create(number: Int): PositiveInteger? = if (number > 0) PositiveInteger(nu
mber) else null
    }
}

fun main() {
    val positiveNumber = PositiveInteger.create(42) ?: return
    // Triggers a warning in 2.0.20
    val negativeNumber = positiveNumber.copy(number = -1)
    // Warning: Non-public primary constructor is exposed via the generated 'copy()' m
ethod of the 'data' class.
    // The generated 'copy()' will change its visibility in future releases.
}
```

For the latest information about our migration plan, see the corresponding issue in YouTrack.

To give you more control over this behavior, in Kotlin 2.0.20 we've introduced two annotations:

- `@ConsistentCopyVisibility` to opt in to the behavior now before we make it the default in a later release.
- `@ExposedCopyVisibility` to opt out of the behavior and suppress warnings at the declaration site. Note that even with this annotation, the compiler still reports warnings when the `copy()` function is called.

If you want to opt in to the new behavior already in 2.0.20 for a whole module rather than in individual classes, you can use the `-Xconsistent-data-class-copy-visibility` compiler option. This option has the same effect as adding the `@ConsistentCopyVisibility` annotation to all data classes in a module.

Phased replacement of context receivers with context parameters

In Kotlin 1.6.20, we introduced context receivers as an Experimental feature. After listening to community feedback, we've decided not to continue with this approach and will take a different direction.

In future Kotlin releases, context receivers will be replaced by context parameters. Context parameters are still in the design phase, and you can find the proposal in the KEEP.

Since the implementation of context parameters requires significant changes to the compiler, we've decided not to support context receivers and context parameters simultaneously. This decision greatly simplifies the implementation and minimizes the risk of unstable behavior.

We understand that context receivers are already being used by a large number of developers. Therefore, we will begin gradually removing support for context receivers. Our migration plan starts with Kotlin 2.0.20, where warnings are issued in your code when context receivers are used with the `-Xcontext-receivers` compiler option. For example:

```
class MyContext

context(MyContext)
// Warning: Experimental context receivers are deprecated and will be superseded by co
ntext parameters.
// Please don't use context receivers. You can either pass parameters explicitly or us
e members with extensions.
fun someFunction() {
}
```

This warning will become an error in future Kotlin releases.

If you use context receivers in your code, we recommend that you migrate your code to use either of the following:

- Explicit parameters.

Before	After
```kotlin context(ContextReceiverType) fun someFunction() { contextReceiverMember() } ```	```kotlin fun someFunction(explicitContext: ContextReceiverType) { explicitContext.contextReceiverMember() } ```

- Extension member functions (if possible).

Before	After
```kotlin context(ContextReceiverType) fun contextReceiverMember() = TODO() context(ContextReceiverType) fun someFunction() { contextReceiverMember() } ```	```kotlin class ContextReceiverType { fun contextReceiverMember() = TODO() } fun ContextReceiverType.someFunction() { contextReceiverMember() } ```

Alternatively, you can wait until the Kotlin release where context parameters are supported in the compiler. Note that context parameters will initially be introduced as an Experimental feature.

Kotlin Multiplatform

Kotlin 2.0.20 brings improvements to source set management in multiplatform projects as well as deprecates compatibility with some Gradle Java plugins due to recent changes in Gradle.

Static accessors for source sets from the default target hierarchy

Since Kotlin 1.9.20, the default hierarchy template is automatically applied to all Kotlin Multiplatform projects. And for all of the source sets from the default hierarchy template, the Kotlin Gradle plugin provided type-safe accessors. That way, you could finally access source sets for all the specified targets without having to use `by getting` or `by creating` constructs.

Kotlin 2.0.20 aims to improve your IDE experience even further. It now provides static accessors in the `sourceSets {}` block for all the source sets from the default hierarchy template. We believe this change will make accessing source sets by name easier and more predictable.

Each such source set now has a detailed KDoc comment with a sample and a diagnostic message with a warning in case you try to access the source set without declaring the corresponding target first:

```
kotlin {
    jvm()
    linuxX64()
    linuxArm64()
    mingwX64()

    sourceSets {
        commonMain.languageSettings {
            progressiveMode = true
        }


        jvmMain { }
        linuxX64Main { }
        linuxArm64Main { }
        // Warning: accessing source set without registering the target
        iosX64Main { }
    }
}
```

```
public abstract val NamedDomainObjectContainer<KotlinSourceSet>.iosX64Main:
NamedDomainObjectProvider<KotlinSourceSet>
```

Static accessor for the main Kotlin Source Set of iosX64 target. Declare iosX64 target to access this source set. If iosX64 target wasn't declared, accessing this source set will cause a runtime error during configuration time.

Sample:

```
kotlin {
   iosX64() // Target is declared, iosX64Main source set is created

   sourceSets {
     iosX64Main.dependencies {
         // Add iosX64Main dependencies here
     }
   }
}
```

Since: 2.0.20

Learn more about the hierarchical project structure in Kotlin Multiplatform.

Deprecated compatibility with Kotlin Multiplatform Gradle plugin and Gradle Java plugins

In Kotlin 2.0.20, we introduce a deprecation warning when you apply the Kotlin Multiplatform Gradle plugin and any of the following Gradle Java plugins to the same project: Java, Java Library, and Application. The warning also appears when another Gradle plugin in your multiplatform project applies a Gradle Java plugin. For example, the Spring Boot Gradle Plugin automatically applies the Application plugin.

We've added this deprecation warning due to fundamental compatibility issues between Kotlin Multiplatform's project model and Gradle's Java ecosystem plugins. Gradle's Java ecosystem plugins currently don't take into account that other plugins may:

- Also publish or compile for the JVM target in a different way than the Java ecosystem plugins.
- Have two different JVM targets in the same project, such as JVM and Android.
- Have a complex multiplatform project structure with potentially multiple non-JVM targets.

Unfortunately, Gradle doesn't currently provide any API to address these issues.

We previously used some workarounds in Kotlin Multiplatform to help with the integration of Java ecosystem plugins. However, these workarounds never truly solved the compatibility issues, and since the release of Gradle 8.8, these workarounds are no longer possible. For more information, see our YouTrack issue.

While we don't yet know exactly how to resolve this compatibility problem, we are committed to continuing support for some form of Java source compilation in your Kotlin Multiplatform projects. At a minimum, we will support the compilation of Java sources and using Gradle's `java-base` plugin within your multiplatform projects.

In the meantime, if you see this deprecation warning in your multiplatform project, we recommend that you:

1. Determine whether you actually need the Gradle Java plugin in your project. If not, consider removing it.
2. Check if the Gradle Java plugin is only used for a single task. If so, you might be able to remove the plugin without much effort. For example, if the task uses a Gradle Java plugin to create a Javadoc JAR file, you can define the Javadoc task manually instead.

Otherwise, if you want to use both the Kotlin Multiplatform Gradle plugin and these Gradle plugins for Java in your multiplatform project, we recommend that you:

1. Create a separate subproject in your multiplatform project.
2. In the separate subproject, apply the Gradle plugin for Java.
3. In the separate subproject, add a dependency on your parent multiplatform project.

> The separate subproject must **not** be a multiplatform project, and you must only use it to set up a dependency on your multiplatform project.
>
> 

For example, you have a multiplatform project called `my-main-project` and you want to use the Application Gradle plugin to run a JVM application.

Once you've created a subproject, let's call it `subproject-A`, your parent project structure should look like this:

```
.
├── build.gradle.kts
├── settings.gradle
├── subproject-A
    └── build.gradle.kts
    └── src
        └── Main.java
```

In your subproject's `build.gradle.kts` file, apply the Application plugin in the `plugins {}` block:

【Kotlin】

```
plugins {
    id("application")
}
```

【Groovy】

```
plugins {
    id('application')
}
```

In your subproject's `build.gradle.kts` file, add a dependency on your parent multiplatform project:

【Kotlin】

```
dependencies {
    implementation(project(":my-main-project")) // The name of your parent multiplatfo
rm project
}
```

【Groovy】

```
dependencies {
    implementation project(':my-main-project') // The name of your parent multiplatfor
m project
}
```

Your parent project is now set up to work with both plugins.

Kotlin/Native

Kotlin/Native receives improvements in the garbage collector and for calling Kotlin suspending functions from Swift/Objective-C.

Concurrent marking in garbage collector

In Kotlin 2.0.20, the JetBrains team takes another step toward improving Kotlin/Native runtime performance. We've added experimental support for concurrent marking in the garbage collector (GC).

By default, application threads must be paused when GC is marking objects in the heap. This greatly affects the duration of the GC pause time, which is important for the performance of latency-critical applications, such as UI applications built with Compose Multiplatform.

Now, the marking phase of the garbage collection can be run simultaneously with application threads. This should significantly shorten the GC pause time and help improve app responsiveness.

How to enable

The feature is currently Experimental. To enable it, set the following option in your `gradle.properties` file:

```
kotlin.native.binary.gc=cms
```

Please report any problems to our issue tracker YouTrack.

Support for bitcode embedding removed

Starting with Kotlin 2.0.20, the Kotlin/Native compiler no longer supports bitcode embedding. Bitcode embedding was deprecated in Xcode 14 and removed in Xcode 15 for all Apple targets.

Now, the `embedBitcode` parameter for the framework configuration, as well as the `-Xembed-bitcode` and `-Xembed-bitcode-marker` command line arguments are deprecated.

If you still use earlier versions of Xcode but want to upgrade to Kotlin 2.0.20, disable bitcode embedding in your Xcode projects.

Changes to GC performance monitoring with signposts

Kotlin 2.0.0 made it possible to monitor the performance of the Kotlin/Native garbage collector (GC) through Xcode Instruments. Instruments include the signposts tool, which can show GC pauses as events. This comes in handy when checking GC-related freezes in your iOS apps.

The feature was enabled by default, but unfortunately, it sometimes led to crashes when the application was run simultaneously with Xcode Instruments. Starting with Kotlin 2.0.20, it requires an explicit opt-in with the following compiler option:

```
-Xbinary=enableSafepointSignposts=true
```

Learn more about GC performance analysis in the documentation.

Ability to call Kotlin suspending functions from Swift/Objective-C on non-main threads

Previously, Kotlin/Native had a default restriction, limiting the ability to call Kotlin suspending functions from Swift and Objective-C to only the main thread. Kotlin 2.0.20 lifts that limitation, allowing you to run Kotlin `suspend` functions from Swift/Objective-C on any thread.

If you've previously switched the default behavior for non-main threads with the `kotlin.native.binary.objcExportSuspendFunctionLaunchThreadRestriction=none` binary option, you can now remove it from your `gradle.properties` file.

Kotlin/Wasm

In Kotlin 2.0.20, Kotlin/Wasm continues the migration towards named exports and relocates the `@ExperimentalWasmDsl` annotation.

Error in default export usage

As part of the migration towards named exports, a warning message was previously printed to the console when using a default import for Kotlin/Wasm exports in JavaScript.

To fully support named exports, this warning has now been upgraded to an error. If you use a default import, you encounter the following error message:

```
Do not use default import. Use the corresponding named import instead.
```

This change is part of a deprecation cycle to migrate towards named exports. Here's what you can expect during each phase:

- **In version 2.0.0**: A warning message is printed to the console, explaining that exporting entities via default exports is deprecated.
- **In version 2.0.20**: An error occurs, requesting the use of the corresponding named import.
- **In version 2.1.0**: The use of default imports is completely removed.

New location of ExperimentalWasmDsl annotation

Previously, the `@ExperimentalWasmDsl` annotation for WebAssembly (Wasm) features was placed in this location within the Kotlin Gradle plugin:

```
org.jetbrains.kotlin.gradle.targets.js.dsl.ExperimentalWasmDsl
```

In 2.0.20, the `@ExperimentalWasmDsl` annotation has been relocated to:

```
org.jetbrains.kotlin.gradle.ExperimentalWasmDsl
```

The previous location is now deprecated and might lead to build failures with unresolved references.

To reflect the new location of the `@ExperimentalWasmDsl` annotation, update the import statement in your Gradle build scripts. Use an explicit import for the new `@ExperimentalWasmDsl` location:

```
import org.jetbrains.kotlin.gradle.ExperimentalWasmDsl
```

Alternatively, remove this star import statement from the old package:

```
import org.jetbrains.kotlin.gradle.targets.js.dsl.*
```

Kotlin/JS

Kotlin/JS introduces some Experimental features to support static members in JavaScript and to create Kotlin collections from JavaScript.

Support for using Kotlin static members in JavaScript

> This feature is Experimental. It may be dropped or changed at any time. Use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack.
>
> 

Starting with Kotlin 2.0.20, you can use the `@JsStatic` annotation. It works similarly to @JvmStatic and instructs the compiler to generate additional static methods for the target declaration. This helps you use static members from your Kotlin code directly in JavaScript.

You can use the `@JsStatic` annotation for functions defined in named objects, as well as in companion objects declared inside classes and interfaces. The compiler generates both a static method of the object and an instance method in the object itself. For example:

```
class C {
    companion object {
        @JsStatic
        fun callStatic() {}
        fun callNonStatic() {}
    }
}
```

Now, `callStatic()` is static in JavaScript while `callNonStatic()` is not:

```
C.callStatic();              // Works, accessing the static function
C.callNonStatic();           // Error, not a static function in the generated JavaScri
pt
C.Companion.callStatic();    // Instance method remains
C.Companion.callNonStatic(); // The only way it works
```

It's also possible to apply the `@JsStatic` annotation to a property of an object or a companion object, making its getter and setter methods static members in that object or the class containing the companion object.

Ability to create Kotlin collections from JavaScript

> This feature is Experimental. It may be dropped or changed at any time. Use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack.
>
> 

Kotlin 2.0.0 introduced the ability to export Kotlin collections to JavaScript (and TypeScript). Now, the JetBrains team is taking another step to improve collection interoperability. Starting with Kotlin 2.0.20, it's possible to create Kotlin collections directly from the JavaScript/TypeScript side.

You can create Kotlin collections from JavaScript and pass them as arguments to the exported constructors or functions. As soon as you mention a collection inside an exported declaration, Kotlin generates a factory for the collection that is available in JavaScript/TypeScript.

Take a look at the following exported function:

```
// Kotlin
@JsExport
fun consumeMutableMap(map: MutableMap<String, Int>)
```

Since the `MutableMap` collection is mentioned, Kotlin generates an object with a factory method available from JavaScript/TypeScript. This factory method then creates a `MutableMap` from a JavaScript `Map`:

```
// JavaScript
import { consumeMutableMap } from "an-awesome-kotlin-module"
import { KtMutableMap } from "an-awesome-kotlin-module/kotlin-kotlin-stdlib"

consumeMutableMap(
    KtMutableMap.fromJsMap(new Map([["First", 1], ["Second", 2]]))
)
```

This feature is available for the `Set`, `Map`, and `List` Kotlin collection types and their mutable counterparts.

Gradle

Kotlin 2.0.20 is fully compatible with Gradle 6.8.3 through 8.6. Gradle 8.7 and 8.8 are also supported, with only one exception: If you use the Kotlin Multiplatform Gradle plugin, you may see deprecation warnings in your multiplatform projects calling the `withJava()` function in the JVM target. We plan to fix this issue as soon as possible.

For more information, see the issue in YouTrack.

You can also use Gradle versions up to the latest Gradle release, but if you do, keep in mind that you might encounter deprecation warnings or some new Gradle features might not work.

This version brings changes such as beginning the deprecation process for the old incremental compilation approach based on JVM history files, as well as a new way of sharing JVM artifacts between projects.

Deprecated incremental compilation based on JVM history files

In Kotlin 2.0.20, the incremental compilation approach based on JVM history files is deprecated in favor of the new incremental compilation approach that has been enabled by default since Kotlin 1.8.20.

The incremental compilation approach based on JVM history files suffered from limitations, such as not working with Gradle's build cache and not supporting compilation avoidance. In contrast, the new incremental compilation approach overcomes these limitations and has performed well since its introduction.

Given that the new incremental compilation approach has been used by default for the last two major Kotlin releases, the `kotlin.incremental.useClasspathSnapshot` Gradle property is deprecated in Kotlin 2.0.20. Therefore, if you use it to opt out, you will see a deprecation warning.

Option to share JVM artifacts between projects as class files

> This feature is Experimental. It may be dropped or changed at any time. Use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack. Opt-in is required (see details below).
>
> 

In Kotlin 2.0.20, we introduce a new approach that changes the way the outputs of Kotlin/JVM compilations, such as JAR files, are shared between projects. With this approach, Gradle's `apiElements` configuration now has a secondary variant that provides access to the directory containing compiled `.class` files. When configured, your project uses this directory instead of requesting the compressed JAR artifact during compilation. This reduces the number of times JAR files are compressed and decompressed, especially for incremental builds.

Our testing shows that this new approach can provide build performance improvements for Linux and macOS hosts. However, on Windows hosts, we have seen a degradation in performance due to how Windows handles I/O operations when working with files.

To try this new approach, add the following property to your `gradle.properties` file:

```
kotlin.jvm.addClassesVariant=true
```

By default, this property is set to `false` and the `apiElements` variant in Gradle requests the compressed JAR artifact.

> Gradle has a related property that you can use in your Java-only projects to only expose the compressed JAR artifact during compilation **instead** of the directories containing compiled `.class` files:
>
> ```
> org.gradle.java.compile-classpath-packaging=true
> ```
>
> For more information on this property and its purpose, see Gradle's documentation on the Significant build performance drop on Windows for huge multi-projects.
>
> 

We would appreciate your feedback on this new approach. Have you noticed any performance improvements while using it? Let us know by adding a comment in YouTrack.

Aligned dependency behavior of Kotlin Gradle plugin with java-test-fixtures plugin

Prior to Kotlin 2.0.20, if you used the `java-test-fixtures` plugin in your project, there was a difference between Gradle and the Kotlin Gradle plugin in how dependencies were propagated.

The Kotlin Gradle plugin propagated dependencies:

- From the `java-test-fixtures` plugin's `implementation` and `api` dependency types to the `test` source set compilation classpath.
- From the main source set's `implementation` and `api` dependency types to the `java-test-`

`fixtures` plugin's source set compilation classpath.

However, Gradle only propagated dependencies in the `api` dependency types.

This difference in behavior led to some projects finding resource files multiple times in the classpath.

As of Kotlin 2.0.20, the Kotlin Gradle plugin's behavior is aligned with Gradle's `java-test-fixtures` plugin so this problem no longer occurs for this or other Gradle plugins.

As a result of this change, some dependencies in the `test` and `testFixtures` source sets may no longer be accessible. If this happens, either change the dependency declaration type from `implementation` to `api` or add a new dependency declaration on the affected source set.

Added task dependency for rare cases when the compile task lacks one on an artifact

Prior to 2.0.20, we found that there were scenarios where a compile task was missing a task dependency for one of its artifact inputs. This meant that the result of the dependent compile task was unstable, as sometimes the artifact had been generated in time, but sometimes, it hadn't.

To fix this issue, the Kotlin Gradle plugin now automatically adds the required task dependency in these scenarios.

In very rare cases, we've found that this new behavior can cause a circular dependency error. For example, if you have multiple compilations where one compilation can see all internal declarations of the other, and the generated artifact relies on the output of both compilation tasks, you could see an error like:

```
FAILURE: Build failed with an exception.

What went wrong:
Circular dependency between the following tasks:
:lib:compileKotlinJvm
--- :lib:jvmJar
     \--- :lib:compileKotlinJvm (*)
(*) - details omitted (listed previously)
```

To fix this circular dependency error, we've added a Gradle property: `archivesTaskOutputAsFriendModule`.

By default, this property is set to `true` to track the task dependency. To disable the use of the artifact in the compilation task, so that no task dependency is required, add the following in your `gradle.properties` file:

```
kotlin.build.archivesTaskOutputAsFriendModule=false
```

For more information, see the issue in YouTrack.

Compose compiler

In Kotlin 2.0.20, the Compose compiler gets a few improvements.

Fix for the unnecessary recompositions issue introduced in 2.0.0

Compose compiler 2.0.0 has an issue where it sometimes incorrectly infers the stability of types in multiplatform projects with non-JVM targets. This can lead to unnecessary (or even endless) recompositions. We strongly recommended updating your Compose apps made for Kotlin 2.0.0 to version 2.0.10 or newer.

If your app is built with Compose compiler 2.0.10 or newer but uses dependencies built with version 2.0.0, these older dependencies may still cause recomposition issues. To prevent this, update your dependencies to versions built with the same Compose compiler as your app.

New way to configure compiler options

We've introduced a new option configuration mechanism to avoid the churn of top-level parameters. It's harder for the Compose compiler team to test things out by creating or removing top-level entries for the `composeCompiler {}` block. So, options such as strong skipping mode and non-skipping group optimizations are now enabled through the `featureFlags` property. This property will be used to test new Compose compiler options that will eventually become default.

This change has also been applied to the Compose compiler Gradle plugin. To configure feature flags going forward, use the following syntax (this code will flip all of the default values):

```
composeCompiler {
    featureFlags = setOf(
        ComposeFeatureFlag.IntrinsicRemember.disabled(),
        ComposeFeatureFlag.OptimizeNonSkippingGroups,
        ComposeFeatureFlag.StrongSkipping.disabled()
    )
}
```

Or, if you are configuring the Compose compiler directly, use the following syntax:

```
-P plugin:androidx.compose.compiler.plugins.kotlin:featureFlag=IntrinsicRemember
```

The `enableIntrinsicRemember`, `enableNonSkippingGroupOptimization`, and `enableStrongSkippingMode` properties have been therefore deprecated.

We would appreciate any feedback you have on this new approach in YouTrack.

Strong skipping mode enabled by default

Strong skipping mode for the Compose compiler is now enabled by default.

Strong skipping mode is a Compose compiler configuration option that changes the rules for what composables can be skipped. With strong skipping mode enabled, composables with unstable parameters can now also be skipped. Strong skipping mode also automatically remembers lambdas used in composable functions, so you should no longer need to wrap your lambdas with `remember` to avoid recomposition.

For more details, see the strong skipping mode documentation.

Composition trace markers enabled by default

The `includeTraceMarkers` option is now set to `true` by default in the Compose compiler Gradle plugin to match the default value in the compiler plugin. This allows you to see composable functions in the Android Studio system trace profiler. For details about composition tracing, see this Android Developers blog post.

Non-skipping group optimizations

This release includes a new compiler option: when enabled, non-skippable and non-restartable composable functions will no longer generate a group around the body of the composable. This leads to fewer allocations and thus to improved performance. This option is experimental and disabled by default but can be enabled with the feature flag `OptimizeNonSkippingGroups` as shown above.

This feature flag is now ready for wider testing. Any issues found when enabling the feature can be filed on the Google issue tracker.

Support for default parameters in abstract composable functions

You can now add default parameters to abstract composable functions.

Previously, the Compose compiler would report an error when attempting to do this even though it is valid Kotlin code. We've now added support for this in the Compose compiler, and the restriction has been removed. This is especially useful for including default `Modifier` values:

```
abstract class Composables {
    @Composable
    abstract fun Composable(modifier: Modifier = Modifier)
}
```

Default parameters for open composable functions are still restricted in 2.0.20. This restriction will be addressed in future releases.

Standard library

The standard library now supports universally unique identifiers as an Experimental feature and includes some changes to Base64 decoding.

Support for UUIDs in the common Kotlin standard library

> This feature is Experimental. To opt in, use the `@ExperimentalUuidApi` annotation or the compiler option `-opt-in=kotlin.uuid.ExperimentalUuidApi`.
>
> 

Kotlin 2.0.20 introduces a class for representing UUIDs (universally unique identifiers) in the common Kotlin standard library to address the challenge of uniquely identifying items.

Additionally, this feature provides APIs for the following UUID-related operations:

- Generating UUIDs.
- Parsing UUIDs from and formatting them to their string representations.
- Creating UUIDs from specified 128-bit values.
- Accessing the 128 bits of a UUID.

The following code example demonstrates these operations:

```
// Constructs a byte array for UUID creation
val byteArray = byteArrayOf(
    0x55, 0x0E, 0x84.toByte(), 0x00, 0xE2.toByte(), 0x9B.toByte(), 0x41, 0xD4.toByte()
,
    0xA7.toByte(), 0x16, 0x44, 0x66, 0x55, 0x44, 0x00, 0x00
)

val uuid1 = Uuid.fromByteArray(byteArray)
val uuid2 = Uuid.fromULongs(0x550E8400E29B41D4uL, 0xA716446655440000uL)
val uuid3 = Uuid.parse("550e8400-e29b-41d4-a716-446655440000")

println(uuid1)
// 550e8400-e29b-41d4-a716-446655440000
println(uuid1 == uuid2)
// true
println(uuid2 == uuid3)
// true

// Accesses UUID bits
val version = uuid1.toLongs { mostSignificantBits, _ ->
```

```
    ((mostSignificantBits shr 12) and 0xF).toInt()
}
println(version)
// 4

// Generates a random UUID
val randomUuid = Uuid.random()

println(uuid1 == randomUuid)
// false
```

To maintain compatibility with APIs that use `java.util.UUID`, there are two extension functions in Kotlin/JVM for converting between `java.util.UUID` and `kotlin.uuid.Uuid`: `.toJavaUuid()` and `.toKotlinUuid()`. For example:

```
val kotlinUuid = Uuid.parseHex("550e8400e29b41d4a716446655440000")
// Converts Kotlin UUID to java.util.UUID
val javaUuid = kotlinUuid.toJavaUuid()

val javaUuid = java.util.UUID.fromString("550e8400-e29b-41d4-a716-446655440000")
// Converts Java UUID to kotlin.uuid.Uuid
val kotlinUuid = javaUuid.toKotlinUuid()
```

This feature and the provided APIs simplify multiplatform software development by allowing code sharing among multiple platforms. UUIDs are also ideal in environments where generating unique identifiers is difficult.

Some example use cases involving UUIDs include:

- Assigning unique IDs to database records.
- Generating web session identifiers.
- Any scenario requiring unique identification or tracking.

Support for minLength in HexFormat

> The `HexFormat` class and its properties are Experimental. To opt in, use the `@OptIn(ExperimentalStdlibApi::class)` annotation or the compiler option `-opt-in=kotlin.ExperimentalStdlibApi`.
>
> 

Kotlin 2.0.20 adds a new `minLength` property to the `NumberHexFormat` class, accessed through `HexFormat.number`. This property lets you specify the minimum number of digits in hexadecimal representations of numeric values, enabling padding with zeros to meet the required length. Additionally, leading zeros can be trimmed using the `removeLeadingZeros` property:

```
fun main() {
    println(93.toHexString(HexFormat {
        number.minLength = 4
        number.removeLeadingZeros = true
    }))
    // "005d"
}
```

The `minLength` property does not affect parsing. However, parsing now allows hex strings to have more digits than the type's width if the extra leading digits are zeros.

Changes to the Base64's decoder behavior

> The `Base64` class and its related features are Experimental. To opt in, use the `@OptIn(ExperimentalEncodingApi::class)` annotation or the compiler option `-opt-in=kotlin.io.encoding.ExperimentalEncodingApi`.
>
> 

Two changes were introduced to the Base64 decoder's behavior in Kotlin 2.0.20:

- The Base64 decoder now requires padding
- A `withPadding` function has been added for padding configuration

The Base64 decoder now requires padding

The Base64 encoder now adds padding by default, and the decoder requires padding and prohibits non-zero pad bits when decoding.

withPadding function for padding configuration

A new `.withPadding()` function has been introduced to give users control over the padding behavior of Base64 encoding and decoding:

```
val base64 = Base64.UrlSafe.withPadding(Base64.PaddingOption.ABSENT_OPTIONAL)
```

This function enables the creation of `Base64` instances with different padding options:

`PaddingOption`	On encode	On decode
`PRESENT`	Add padding	Padding is required
`ABSENT`	Omit padding	No padding allowed
`PRESENT_OPTIONAL`	Add padding	Padding is optional

`ABSENT_OPTIONAL`	Omit padding	Padding is optional

You can create `Base64` instances with different padding options and use them to encode and decode data:

```
import kotlin.io.encoding.Base64
import kotlin.io.encoding.ExperimentalEncodingApi

@OptIn(ExperimentalEncodingApi::class)
fun main() {
    // Example data to encode
    val data = "fooba".toByteArray()

    // Creates a Base64 instance with URL-safe alphabet and PRESENT padding
    val base64Present = Base64.UrlSafe.withPadding(Base64.PaddingOption.PRESENT)
    val encodedDataPresent = base64Present.encode(data)
    println("Encoded data with PRESENT padding: $encodedDataPresent")
    // Encoded data with PRESENT padding: Zm9vYmE=

    // Creates a Base64 instance with URL-safe alphabet and ABSENT padding
    val base64Absent = Base64.UrlSafe.withPadding(Base64.PaddingOption.ABSENT)
    val encodedDataAbsent = base64Absent.encode(data)
    println("Encoded data with ABSENT padding: $encodedDataAbsent")
    // Encoded data with ABSENT padding: Zm9vYmE

    // Decodes the data back
    val decodedDataPresent = base64Present.decode(encodedDataPresent)
    println("Decoded data with PRESENT padding: ${String(decodedDataPresent)}")
    // Decoded data with PRESENT padding: fooba

    val decodedDataAbsent = base64Absent.decode(encodedDataAbsent)
    println("Decoded data with ABSENT padding: ${String(decodedDataAbsent)}")
    // Decoded data with ABSENT padding: fooba
}
```

Documentation updates

The Kotlin documentation has received some notable changes:

- Improved Standard input page - Learn how to use Java Scanner and `readln()`.
- Improved K2 compiler migration guide - Learn about performance improvements, compatibility with Kotlin libraries and what to do with your custom compiler plugins.
- Improved Exceptions page - Learn about exceptions, how to throw and catch them.
- Improved Test code using JUnit in JVM - tutorial - Learn how to create tests using JUnit.
- Improved Interoperability with Swift/Objective-C page - Learn how to use Kotlin declarations in Swift/Objective-C code and Objective-C declarations in Kotlin code.
- Improved Swift package export setup page - Learn how to set up Kotlin/Native output that can

be consumed by a Swift package manager dependency.

Install Kotlin 2.0.20

Starting from IntelliJ IDEA 2023.3 and Android Studio Iguana (2023.2.1) Canary 15, the Kotlin plugin is distributed as a bundled plugin included in your IDE. This means that you can't install the plugin from JetBrains Marketplace anymore.

To update to the new Kotlin version, change the Kotlin version to 2.0.20 in your build scripts.

Kotlin 2.0.0 的新特性

发布于：*2024-05-21*

The Kotlin 2.0.0 release is out and the new Kotlin K2 compiler is Stable! Additionally, here are some other highlights:

- New Compose compiler Gradle plugin
- Generation of lambda functions using invokedynamic
- The kotlinx-metadata-jvm library is now Stable
- Monitoring GC performance in Kotlin/Native with signposts on Apple platforms
- Resolving conflicts in Kotlin/Native with Objective-C methods
- Support for named export in Kotlin/Wasm
- Support for unsigned primitive types in functions with @JsExport in Kotlin/Wasm
- Optimize production builds by default using Binaryen
- New Gradle DSL for compiler options in multiplatform projects
- Stable replacement of the enum class values generic function
- Stable AutoCloseable interface

Kotlin 2.0 is a huge milestone for the JetBrains team. This release was the center of KotlinConf 2024. Check out the opening keynote, where we announced exciting updates and addressed the recent work on the Kotlin language:

YouTube 视频：KotlinConf'24 - Keynote

IDE support

The Kotlin plugins that support Kotlin 2.0.0 are bundled in the latest IntelliJ IDEA and Android Studio. You don't need to update the Kotlin plugin in your IDE. All you need to do is to change the Kotlin version to Kotlin 2.0.0 in your build scripts.

- For details about IntelliJ IDEA's support for the Kotlin K2 compiler, see Support in IDEs.
- For more details about IntelliJ IDEA's support for Kotlin, see Kotlin releases.

Kotlin K2 compiler

The road to the K2 compiler has been a long one, but now the JetBrains team is finally ready to announce its stabilization. In Kotlin 2.0.0, the new Kotlin K2 compiler is used by default and it is Stable for all target platforms: JVM, Native, Wasm, and JS. The new compiler brings major performance improvements, speeds up new language feature development, unifies all platforms that Kotlin supports, and provides a better architecture for multiplatform projects.

The JetBrains team has ensured the quality of the new compiler by successfully compiling 10 million lines of code from selected user and internal projects. 18,000 developers were involved in the stabilization process, testing the new K2 compiler across a total of 80,000 projects and reporting any problems they found.

To help make the migration process to the new compiler as smooth as possible, we've created a K2 compiler migration guide. This guide explains the many benefits of the compiler, highlights any changes you might encounter, and describes how to roll back to the previous version if necessary.

In a blog post, we explored the performance of the K2 compiler in different projects. Check it out if you'd like to see real data on how the K2 compiler performs and find instructions on how to collect performance benchmarks from your own projects.

You can also watch this talk from KotlinConf 2024, where Michail Zarečenskij, the lead language designer, discusses the feature evolution in Kotlin and the K2 compiler:

YouTube 视频：Kotlin Language Features in 2.0 and Beyond

Current K2 compiler limitations

Enabling K2 in your Gradle project comes with certain limitations that can affect projects using Gradle versions below 8.3 in the following cases:

- Compilation of source code from `buildSrc`.
- Compilation of Gradle plugins in included builds.
- Compilation of other Gradle plugins if they are used in projects with Gradle versions below 8.3.
- Building Gradle plugin dependencies.

If you encounter any of the problems mentioned above, you can take the following steps to address them:

- Set the language version for `buildSrc`, any Gradle plugins, and their dependencies:

```
kotlin {
    compilerOptions {
        languageVersion.set(org.jetbrains.kotlin.gradle.dsl.KotlinVersion.KOTLIN_1
_9)
        apiVersion.set(org.jetbrains.kotlin.gradle.dsl.KotlinVersion.KOTLIN_1_9)
    }
}
```

> If you configure language and API versions for specific tasks, these values will override the values set by the `compilerOptions` extension. In this case, language and API versions should not be higher than 1.9.
>
> 

- Update the Gradle version in your project to 8.3 or later.

Smart cast improvements

The Kotlin compiler can automatically cast an object to a type in specific cases, saving you the trouble of having to explicitly cast it yourself. This is called smart casting. The Kotlin K2 compiler now performs smart casts in even more scenarios than before.

In Kotlin 2.0.0, we've made improvements related to smart casts in the following areas:

- Local variables and further scopes
- Type checks with logical `or` operator
- Inline functions
- Properties with function types
- Exception handling
- Increment and decrement operators

Local variables and further scopes

Previously, if a variable was evaluated as not `null` within an `if` condition, the variable would be smart-cast. Information about this variable would then be shared further within the scope of the `if` block.

However, if you declared the variable **outside** the `if` condition, no information about the variable would be available within the `if` condition, so it couldn't be smart-cast. This behavior was also seen with `when` expressions and `while` loops.

From Kotlin 2.0.0, if you declare a variable before using it in your `if`, `when`, or `while` condition, then any information collected by the compiler about the variable will be accessible in the corresponding block for smart-casting.

This can be useful when you want to do things like extract boolean conditions into variables. Then, you can give the variable a meaningful name, which will improve your code readability and make it possible to reuse the variable later in your code. For example:

```
class Cat {
    fun purr() {
        println("Purr purr")
    }
}

fun petAnimal(animal: Any) {
    val isCat = animal is Cat
    if (isCat) {
        // In Kotlin 2.0.0, the compiler can access
        // information about isCat, so it knows that
        // animal was smart-cast to the type Cat.
        // Therefore, the purr() function can be called.
        // In Kotlin 1.9.20, the compiler doesn't know
        // about the smart cast, so calling the purr()
        // function triggers an error.
        animal.purr()
    }
}

fun main() {
    val kitty = Cat()
    petAnimal(kitty)
    // Purr purr
}
```

Type checks with logical or operator

In Kotlin 2.0.0, if you combine type checks for objects with an `or` operator (`||`), a smart cast is made to their closest common supertype. Before this change, a smart cast was always made to the `Any` type.

In this case, you still had to manually check the object type afterward before you could access any of its properties or call its functions. For example:

```
interface Status {
    fun signal() {}
}

interface Ok : Status
```

```
interface Postponed : Status
interface Declined : Status

fun signalCheck(signalStatus: Any) {
    if (signalStatus is Postponed || signalStatus is Declined) {
        // signalStatus is smart-cast to a common supertype Status
        signalStatus.signal()
        // Prior to Kotlin 2.0.0, signalStatus is smart cast
        // to type Any, so calling the signal() function triggered an
        // Unresolved reference error. The signal() function can only
        // be called successfully after another type check:

        // check(signalStatus is Status)
        // signalStatus.signal()
    }
}
```

> The common supertype is an **approximation** of a union type. Union types are not supported in Kotlin.
>
> 

Inline functions

In Kotlin 2.0.0, the K2 compiler treats inline functions differently, allowing it to determine in combination with other compiler analyses whether it's safe to smart-cast.

Specifically, inline functions are now treated as having an implicit `callsInPlace` contract. This means that any lambda functions passed to an inline function are called in place. Since lambda functions are called in place, the compiler knows that a lambda function can't leak references to any variables contained within its function body.

The compiler uses this knowledge along with other compiler analyses to decide whether it's safe to smart-cast any of the captured variables. For example:

```
interface Processor {
    fun process()
}

inline fun inlineAction(f: () -> Unit) = f()

fun nextProcessor(): Processor? = null

fun runProcessor(): Processor? {
    var processor: Processor? = null
    inlineAction {
        // In Kotlin 2.0.0, the compiler knows that processor
        // is a local variable, and inlineAction() is an inline function, so
        // references to processor can't be leaked. Therefore, it's safe
```

```
        // to smart-cast processor.

        // If processor isn't null, processor is smart-cast
        if (processor != null) {
            // The compiler knows that processor isn't null, so no safe call
            // is needed
            processor.process()

            // In Kotlin 1.9.20, you have to perform a safe call:
            // processor?.process()
        }

        processor = nextProcessor()
    }

    return processor
}
```

Properties with function types

In previous versions of Kotlin, there was a bug that meant that class properties with a function type weren't smart-cast. We fixed this behavior in Kotlin 2.0.0 and the K2 compiler. For example:

```
class Holder(val provider: (() -> Unit)?) {
    fun process() {
        // In Kotlin 2.0.0, if provider isn't null, then
        // provider is smart-cast
        if (provider != null) {
            // The compiler knows that provider isn't null
            provider()

            // In 1.9.20, the compiler doesn't know that provider isn't
            // null, so it triggers an error:
            // Reference has a nullable type '(() -> Unit)?', use explicit '?.invoke()
' to make a function-like call instead
        }
    }
}
```

This change also applies if you overload your `invoke` operator. For example:

```
interface Provider {
    operator fun invoke()
}

interface Processor : () -> String

class Holder(val provider: Provider?, val processor: Processor?) {
    fun process() {
```

```
        if (provider != null) {
            provider()
            // In 1.9.20, the compiler triggers an error:
            // Reference has a nullable type 'Provider?' use explicit '?.invoke()' to
make a function-like call instead
        }
    }
}
```

Exception handling

In Kotlin 2.0.0, we've made improvements to exception handling so that smart cast information can be passed on to `catch` and `finally` blocks. This change makes your code safer as the compiler keeps track of whether your object has a nullable type. For example:

```
//sampleStart
fun testString() {
    var stringInput: String? = null
    // stringInput is smart-cast to String type
    stringInput = ""
    try {
        // The compiler knows that stringInput isn't null
        println(stringInput.length)
        // 0

        // The compiler rejects previous smart cast information for
        // stringInput. Now stringInput has the String? type.
        stringInput = null

        // Trigger an exception
        if (2 > 1) throw Exception()
        stringInput = ""
    } catch (exception: Exception) {
        // In Kotlin 2.0.0, the compiler knows stringInput
        // can be null, so stringInput stays nullable.
        println(stringInput?.length)
        // null

        // In Kotlin 1.9.20, the compiler says that a safe call isn't
        // needed, but this is incorrect.
    }
}

//sampleEnd
fun main() {
    testString()
}
```

Increment and decrement operators

Prior to Kotlin 2.0.0, the compiler didn't understand that the type of an object can change after using an increment or decrement operator. As the compiler couldn't accurately track the object type, your code could lead to unresolved reference errors. In Kotlin 2.0.0, this has been fixed:

```
interface Rho {
    operator fun inc(): Sigma = TODO()
}

interface Sigma : Rho {
    fun sigma() = Unit
}

interface Tau {
    fun tau() = Unit
}

fun main(input: Rho) {
    var unknownObject: Rho = input

    // Check if unknownObject inherits from the Tau interface
    // Note, it's possible that unknownObject inherits from both
    // Rho and Tau interfaces.
    if (unknownObject is Tau) {

        // Use the overloaded inc() operator from interface Rho.
        // In Kotlin 2.0.0, the type of unknownObject is smart-cast to
        // Sigma.
        ++unknownObject

        // In Kotlin 2.0.0, the compiler knows unknownObject has type
        // Sigma, so the sigma() function can be called successfully.
        unknownObject.sigma()

        // In Kotlin 1.9.20, the compiler doesn't perform a smart cast
        // when inc() is called so the compiler still thinks that
        // unknownObject has type Tau. Calling the sigma() function
        // throws a compile-time error.

        // In Kotlin 2.0.0, the compiler knows unknownObject has type
        // Sigma, so calling the tau() function throws a compile-time
        // error.
        unknownObject.tau()
        // Unresolved reference 'tau'

        // In Kotlin 1.9.20, since the compiler mistakenly thinks that
        // unknownObject has type Tau, the tau() function can be called,
        // but it throws a ClassCastException.
    }
}
```

Kotlin Multiplatform improvements

In Kotlin 2.0.0, we've made improvements in the K2 compiler related to Kotlin Multiplatform in the following areas:

- Separation of common and platform sources during compilation
- Different visibility levels of expected and actual declarations

Separation of common and platform sources during compilation

Previously, the design of the Kotlin compiler prevented it from keeping common and platform source sets separate at compile time. As a consequence, common code could access platform code, which resulted in different behavior between platforms. In addition, some compiler settings and dependencies from common code used to leak into platform code.

In Kotlin 2.0.0, our implementation of the new Kotlin K2 compiler included a redesign of the compilation scheme to ensure strict separation between common and platform source sets. This change is most noticeable when you use expected and actual functions. Previously, it was possible for a function call in your common code to resolve to a function in platform code. For example:

Common code	Platform code
```kotlin fun foo(x: Any) = println("common foo") fun exampleFunction() { foo(42) } ```	```kotlin // JVM fun foo(x: Int) = println("platform foo") // JavaScript // There is no foo() function overload // on the JavaScript platform ```

In this example, the common code has different behavior depending on which platform it is run on:

- On the JVM platform, calling the `foo()` function in the common code results in the `foo()` function from the platform code being called as `platform foo`.
- On the JavaScript platform, calling the `foo()` function in the common code results in the `foo()` function from the common code being called as `common foo`, as there is no such function available in the platform code.

In Kotlin 2.0.0, common code doesn't have access to platform code, so both platforms successfully resolve the `foo()` function to the `foo()` function in the common code: `common foo`.

In addition to the improved consistency of behavior across platforms, we also worked hard to fix cases where there was conflicting behavior between IntelliJ IDEA or Android Studio and the compiler. For instance, when you used expected and actual classes, the following would happen:

Common code	Platform code
``kotlin expect class Identity { fun confirmIdentity(): String } fun common() { // Before 2.0.0, // it triggers an IDE-only error	```kotlin actual class Identity { actual fun

common() { // Before 2.0.0, // it triggers an IDE-only error Identity().confirmIdentity() // RESOLUTION_TO_CLASSIFIER : Expected class // Identity has no default constructor. } ```	confirmIdentity() = "expect class fun: jvm" } ```

In this example, the expected class `Identity` has no default constructor, so it can't be called successfully in common code. Previously, an error was only reported by the IDE, but the code still compiled successfully on the JVM. However, now the compiler correctly reports an error:

```
Expected class 'expect class Identity : Any' does not have default constructor
```

When resolution behavior doesn't change

We're still in the process of migrating to the new compilation scheme, so the resolution behavior is still the same when you call functions that aren't within the same source set. You'll notice this difference mainly when you use overloads from a multiplatform library in your common code.

Suppose you have a library, which has two `whichFun()` functions with different signatures:

```
// Example library

// MODULE: common
fun whichFun(x: Any) = println("common function")

// MODULE: JVM
fun whichFun(x: Int) = println("platform function")
```

If you call the `whichFun()` function in your common code, the function that has the most relevant argument type in the library is resolved:

```
// A project that uses the example library for the JVM target

// MODULE: common
fun main() {
    whichFun(2)
    // platform function
}
```

In comparison, if you declare the overloads for `whichFun()` within the same source set, the function from the common code will be resolved because your code doesn't have access to the platform-specific version:

```
// Example library isn't used

// MODULE: common
fun whichFun(x: Any) = println("common function")
```

```
fun main() {
    whichFun(2)
    // common function
}

// MODULE: JVM
fun whichFun(x: Int) = println("platform function")
```

Similar to multiplatform libraries, since the `commonTest` module is in a separate source set, it also still has access to platform-specific code. Therefore, the resolution of calls to functions in the `commonTest` module exhibits the same behavior as in the old compilation scheme.

In the future, these remaining cases will be more consistent with the new compilation scheme.

Different visibility levels of expected and actual declarations

Before Kotlin 2.0.0, if you used expected and actual declarations in your Kotlin Multiplatform project, they had to have the same visibility level. Kotlin 2.0.0 now also supports different visibility levels but **only** if the actual declaration is *more* permissive than the expected declaration. For example:

```
expect internal class Attribute // Visibility is internal
actual class Attribute          // Visibility is public by default,
                                // which is more permissive
```

Similarly, if you are using a type alias in your actual declaration, the visibility of the **underlying type** should be the same or more permissive than the expected declaration. For example:

```
expect internal class Attribute                 // Visibility is internal
internal actual typealias Attribute = Expanded

class Expanded                                  // Visibility is public by default,
                                                // which is more permissive
```

Compiler plugins support

Currently, the Kotlin K2 compiler supports the following Kotlin compiler plugins:

- `all-open`
- AtomicFU
- `jvm-abi-gen`
- `js-plain-objects`
- kapt
- Lombok

- Lombok
- `no-arg`
- Parcelize
- SAM with receiver
- serialization
- Power-assert

In addition, the Kotlin K2 compiler supports:

- The Jetpack Compose compiler plugin 2.0.0, which was moved into the Kotlin repository.
- The Kotlin Symbol Processing (KSP) plugin since KSP2.

> If you use any additional compiler plugins, check their documentation to see if they are compatible with K2.
>
> 

Experimental Kotlin Power-assert compiler plugin

> The Kotlin Power-assert plugin is Experimental. It may be changed at any time.
>
> 

Kotlin 2.0.0 introduces an experimental Power-assert compiler plugin. This plugin improves the experience of writing tests by including contextual information in failure messages, making debugging easier and more efficient.

Developers often need to use complex assertion libraries to write effective tests. The Power-assert plugin simplifies this process by automatically generating failure messages that include intermediate values of the assertion expression. This helps developers quickly understand why a test failed.

When an assertion fails in a test, the improved error message shows the values of all variables and sub-expressions within the assertion, making it clear which part of the condition caused the failure. This is particularly useful for complex assertions where multiple conditions are checked.

To enable the plugin in your project, configure it in your `build.gradle(.kts)` file:

> **【Kotlin】**

```
plugins {
    kotlin("multiplatform") version "2.0.0"
    kotlin("plugin.power-assert") version "2.0.0"
}

powerAssert {
```

```
}
```

【Groovy】

```
plugins {
    id 'org.jetbrains.kotlin.multiplatform' version '2.0.0'
    id 'org.jetbrains.kotlin.plugin.power-assert' version '2.0.0'
}

powerAssert {
    functions = ["kotlin.assert", "kotlin.test.assertTrue"]
}
```

Learn more about the Kotlin Power-assert plugin in the documentation.

How to enable the Kotlin K2 compiler

Starting with Kotlin 2.0.0, the Kotlin K2 compiler is enabled by default. No additional actions are required.

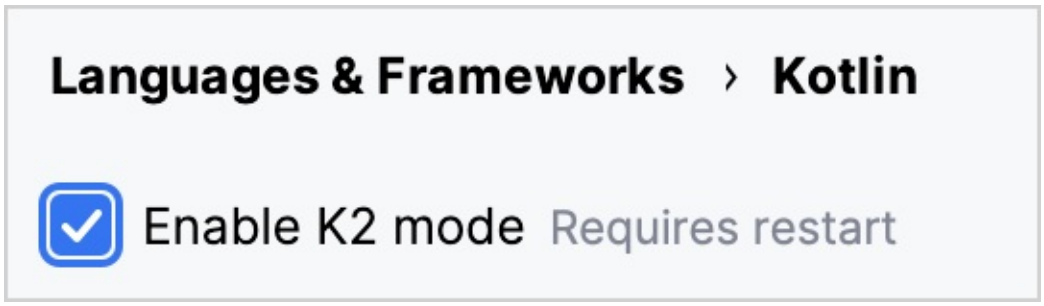
Try the Kotlin K2 compiler in Kotlin Playground

Kotlin Playground supports the 2.0.0 release. Check it out!

Support in IDEs

By default, IntelliJ IDEA and Android Studio still use the previous compiler for code analysis, code completion, highlighting, and other IDE-related features. To get the full Kotlin 2.0 experience in your IDE, enable K2 mode.

In your IDE, go to **Settings** | **Languages & Frameworks** | **Kotlin** and select the **Enable K2 mode** option. The IDE will analyze your code using its K2 mode.



After enabling K2 mode, you may notice differences in IDE analysis due to changes in compiler behavior. Learn how the new K2 compiler differs from the previous one in our migration guide.

- Learn more about K2 mode in our blog.
- We are actively collecting feedback about K2 mode, so please share your thoughts in our public Slack channel.

Leave your feedback on the new K2 compiler

We would appreciate any feedback you may have!

- Report any problems you face with the new K2 compiler in our issue tracker.
- Enable the "Send usage statistics" option to allow JetBrains to collect anonymous data about K2 usage.

Kotlin/JVM

Starting with version 2.0.0, the compiler can generate classes containing Java 22 bytecode. This version also brings the following changes:

- Generation of lambda functions using invokedynamic
- The kotlinx-metadata-jvm library is now Stable

Generation of lambda functions using invokedynamic

Kotlin 2.0.0 introduces a new default method for generating lambda functions using `invokedynamic`. This change reduces the binary sizes of applications compared to the traditional anonymous class generation.

Since the first version, Kotlin has generated lambdas as anonymous classes. However, starting from Kotlin 1.5.0, the option for `invokedynamic` generation has been available by using the `-Xlambdas=indy` compiler option. In Kotlin 2.0.0, `invokedynamic` has become the default method for lambda generation. This method produces lighter binaries and aligns Kotlin with JVM optimizations, ensuring applications benefit from ongoing and future improvements in JVM performance.

Currently, it has three limitations compared to ordinary lambda compilation:

- A lambda compiled into `invokedynamic` is not serializable.
- Experimental `reflect()` API does not support lambdas generated by `invokedynamic`.
- Calling `.toString()` on such a lambda produces a less readable string representation:

```
fun main() {
    println({})
```

```
    // With Kotlin 1.9.24 and reflection, returns
    // () -> kotlin.Unit

    // With Kotlin 2.0.0, returns
    // FileKt$$Lambda$13/0x00007f88a0004608@506e1b77
}
```

To retain the legacy behavior of generating lambda functions, you can either:

- Annotate specific lambdas with `@JvmSerializableLambda`.
- Use the compiler option `-Xlambdas=class` to generate all lambdas in a module using the legacy method.

The kotlinx-metadata-jvm library is Stable

In Kotlin 2.0.0, the `kotlinx-metadata-jvm` library became Stable. Now that the library has changed to the `kotlin` package and coordinates, you can find it as `kotlin-metadata-jvm` (without the "x").

Previously, the `kotlinx-metadata-jvm` library had its own publishing scheme and version. Now, we will build and publish the `kotlin-metadata-jvm` updates as part of the Kotlin release cycle, with the same backward compatibility guarantees as the Kotlin standard library.

The `kotlin-metadata-jvm` library provides an API to read and modify metadata of binary files generated by the Kotlin/JVM compiler.

Kotlin/Native

This version brings the following changes:

- Monitoring GC performance with signposts
- Resolving conflicts with Objective-C methods
- Changed log level for compiler arguments in Kotlin/Native
- Explicitly added standard library and platform dependencies to Kotlin/Native
- Tasks error in Gradle configuration cache

Monitoring GC performance with signposts on Apple platforms

Previously, it was only possible to monitor the performance of Kotlin/Native's garbage collector (GC) by looking into logs. However, these logs were not integrated with Xcode Instruments, a popular toolkit for investigating issues with iOS apps' performance.

Since Kotlin 2.0.0, GC reports pauses with signposts that are available in Instruments. Signposts allow for custom logging within your app, so now, when debugging iOS app performance, you can check if a GC pause corresponds to the application freeze.

Learn more about GC performance analysis in the documentation.

Resolving conflicts with Objective-C methods

Objective-C methods can have different names, but the same number and types of parameters. For example, `locationManager:didEnterRegion:` and `locationManager:didExitRegion:`. In Kotlin, these methods have the same signature, so an attempt to use them triggers a conflicting overloads error.

Previously, you had to manually suppress conflicting overloads to avoid this compilation error. To improve Kotlin interoperability with Objective-C, the Kotlin 2.0.0 introduces the new `@ObjCSignatureOverride` annotation.

The annotation instructs the Kotlin compiler to ignore conflicting overloads, in case several functions with the same argument types but different argument names are inherited from the Objective-C class.

Applying this annotation is also safer than general error suppression. This annotation can only be used in the case of overriding Objective-C methods, which are supported and tested, while general suppression may hide important errors and lead to silently broken code.

Changed log level for compiler arguments

In this release, the log level for compiler arguments in Kotlin/Native Gradle tasks, such as `compile`, `link`, and `cinterop`, has changed from `info` to `debug`.

With `debug` as its default value, the log level is consistent with other Gradle compilation tasks and provides detailed debugging information, including all compiler arguments.

Explicitly added standard library and platform dependencies to Kotlin/Native

Previously, the Kotlin/Native compiler resolved standard library and platform dependencies implicitly, which caused inconsistencies in the way the Kotlin Gradle plugin worked across Kotlin targets.

Now, each Kotlin/Native Gradle compilation explicitly includes standard library and platform dependencies in its compile-time library path via the `compileDependencyFiles` compilation parameter.

Tasks error in Gradle configuration cache

Since Kotlin 2.0.0, you may encounter a configuration cache error with messages indicating: `invocation of Task.project at execution time is unsupported`.

This error appears in tasks such as `NativeDistributionCommonizerTask` and `KotlinNativeCompile`.

However, this is a false-positive error. The underlying issue is the presence of tasks that are not compatible with the Gradle configuration cache, like the `publish*` task.

This discrepancy may not be immediately apparent, as the error message suggests a different root cause.

As the precise cause isn't explicitly stated in the error report, the Gradle team is already addressing the issue to fix reports.

Kotlin/Wasm

Kotlin 2.0.0 improves performance and interoperability with JavaScript:

- Optimized production builds by default using Binaryen
- Support for named export
- Support for unsigned primitive types in functions with `@JsExport`
- Generation of TypeScript declaration files in Kotlin/Wasm
- Support for catching JavaScript exceptions
- New exception handling proposal is now supported as an option
- The `withWasm()` function is split into JS and WASI variants

Optimized production builds by default using Binaryen

The Kotlin/Wasm toolchain now applies the Binaryen tool during production compilation to all projects, as opposed to the previous manual setup approach. By our estimations, it should improve runtime performance and reduce the binary size for your project.

> This change only affects production compilation. The development compilation process stays the same.
>
> 

Support for named export

Previously, all exported declarations from Kotlin/Wasm were imported into JavaScript using default export:

```
//JavaScript:
import Module from "./index.mjs"

Module.add()
```

Now, you can import each Kotlin declaration marked with `@JsExport` by name:

```
// Kotlin:
@JsExport
fun add(a: Int, b: Int) = a + b
```

```
//JavaScript:
import { add } from "./index.mjs"
```

Named exports make it easier to share code between Kotlin and JavaScript modules. They improve readability and help you manage dependencies between modules.

Support for unsigned primitive types in functions with @JsExport

Starting from Kotlin 2.0.0, you can use unsigned primitive types inside external declarations and functions with the `@JsExport` annotation that makes Kotlin/Wasm functions available in JavaScript code.

This helps to mitigate the previous limitation that prevented the unsigned primitives from being used directly inside exported and external declarations. Now you can export functions with unsigned primitives as a return or parameter type and consume external declarations that return or consume unsigned primitives.

For more information on Kotlin/Wasm interoperability with JavaScript, see the documentation.

Generation of TypeScript declaration files in Kotlin/Wasm

> Generating TypeScript declaration files in Kotlin/Wasm is Experimental. It may be dropped or changed at any time.
>
> 

In Kotlin 2.0.0, the Kotlin/Wasm compiler is now capable of generating TypeScript definitions from any `@JsExport` declarations in your Kotlin code. These definitions can be used by IDEs and JavaScript tools to provide code autocompletion, help with type checks, and make it easier to include Kotlin code in JavaScript.

The Kotlin/Wasm compiler collects any top-level functions marked with `@JsExport` and automatically generates TypeScript definitions in a `.d.ts` file.

To generate TypeScript definitions, in your `build.gradle(.kts)` file in the `wasmJs {}` block, add the `generateTypeScriptDefinitions()` function:

```
kotlin {
    wasmJs {
        binaries.executable()
        browser {
        }
        generateTypeScriptDefinitions()
    }
}
```

Support for catching JavaScript exceptions

Previously, Kotlin/Wasm code could not catch JavaScript exceptions, making it difficult to handle errors originating from the JavaScript side of the program.

In Kotlin 2.0.0, we have implemented support for catching JavaScript exceptions within Kotlin/Wasm. This implementation allows you to use `try-catch` blocks, with specific types like `Throwable` or `JsException`, to handle these errors properly.

Additionally, `finally` blocks, which help execute code regardless of exceptions, also work correctly. While we're introducing support for catching JavaScript exceptions, no additional information is provided when a JavaScript exception, like a call stack, occurs. However, we are working on these implementations.

New exception handling proposal is now supported as an option

In this release, we introduce support for the new version of WebAssembly's exception handling proposal within Kotlin/Wasm.

This update ensures the new proposal aligns with Kotlin requirements, enabling the use of Kotlin/Wasm on virtual machines that only support the latest version of the proposal.

Activate the new exception handling proposal by using the `-Xwasm-use-new-exception-proposal` compiler option, which is turned off by default.

The withWasm() function is split into JS and WASI variants

The `withWasm()` function, which used to provide Wasm targets for hierarchy templates, is deprecated in favor of specialized `withWasmJs()` and `withWasmWasi()` functions.

Now you can separate the WASI and JS targets between different groups in the tree definition.

Kotlin/JS

Among other changes, this version brings modern JS compilation to Kotlin, supporting more features from the ES2015 standard:

- New compilation target
- Suspend functions as ES2015 generators
- Passing arguments to the main function
- Per-file compilation for Kotlin/JS projects
- Improved collection interoperability
- Support for createInstance()
- Support for type-safe plain JavaScript objects
- Support for npm package manager
- Changes to compilation tasks
- Discontinuing legacy Kotlin/JS JAR artifacts

New compilation target

In Kotlin 2.0.0, we're adding a new compilation target to Kotlin/JS, `es2015`. This is a new way for you to enable all the ES2015 features supported in Kotlin at once.

You can set it up in your `build.gradle(.kts)` file like this:

```
kotlin {
    js {
        compilerOptions {
            target.set("es2015")
        }
    }
}
```

The new target automatically turns on ES classes and modules and the newly supported ES generators.

Suspend functions as ES2015 generators

This release introduces Experimental support for ES2015 generators for compiling suspend functions.

Using generators instead of state machines should improve the final bundle size of your project. For example, the JetBrains team managed to decrease the bundle size of its Space project by 20% by using the ES2015 generators.

Learn more about ES2015 (ECMAScript 2015, ES6) in the official documentation.

Passing arguments to the main function

Starting with Kotlin 2.0.0, you can specify a source of your `args` for the `main()` function. This feature makes it easier to work with the command line and pass the arguments.

To do this, define the `js {}` block with the new `passAsArgumentToMainFunction()` function, which returns an array of strings:

```
kotlin {
    js {
        binary.executable()
        passAsArgumentToMainFunction("Deno.args")
    }
}
```

The function is executed at runtime. It takes the JavaScript expression and uses it as the `args: Array<String>` argument instead of the `main()` function call.

Also, if you use the Node.js runtime, you can take advantage of a special alias. It allows you to pass `process.argv` to the `args` parameter once instead of adding it manually every time:

```
kotlin {
    js {
        binary.executable()
        nodejs {
            passProcessArgvToMainFunction()
        }
    }
}
```

Per-file compilation for Kotlin/JS projects

Kotlin 2.0.0 introduces a new granularity option for the Kotlin/JS project output. You can now set up a per-file compilation that generates one JavaScript file for each Kotlin file. It helps to significantly optimize the size of the final bundle and improve the loading time of the program.

Previously, there were only two output options. The Kotlin/JS compiler could generate a single `.js` file for the whole project. However, this file might be too large and inconvenient to use. Whenever you wanted to use a function from your project, you had to include the entire JavaScript file as a

dependency. Alternatively, you could configure a compilation of a separate `.js` file for each project module. This is still the default option.

Since module files could also be too large, with Kotlin 2.0.0, we add a more granular output that generates one (or two, if the file contains exported declarations) JavaScript file per each Kotlin file. To enable the per-file compilation mode:

1. Add the `useEsModules()` function to your build file to support ECMAScript modules:

   ```
   // build.gradle.kts
   kotlin {
       js(IR) {
           useEsModules() // Enables ES2015 modules
           browser()
       }
   }
   ```

   You can also use the new `es2015` compilation target for that.

2. Apply the `-Xir-per-file` compiler option or update your `gradle.properties` file with:

   ```
   # gradle.properties
   kotlin.js.ir.output.granularity=per-file // `per-module` is the default
   ```

Improved collection interoperability

Starting with Kotlin 2.0.0, it's possible to export declarations with a Kotlin collection type inside the signature to JavaScript (and TypeScript). This applies to `Set`, `Map`, and `List` collection types and their mutable counterparts.

To use Kotlin collections in JavaScript, first mark the necessary declarations with `@JsExport` annotation:

```
// Kotlin
@JsExport
data class User(
    val name: String,
    val friends: List<User> = emptyList()
)

@JsExport
val me = User(
    name = "Me",
    friends = listOf(User(name = "Kodee"))
)
```

You can then consume them from JavaScript as regular JavaScript arrays:

```
// JavaScript
import { User, me, KtList } from "my-module"

const allMyFriendNames = me.friends
    .asJsReadonlyArrayView()
    .map(x => x.name) // ['Kodee']
```

> Unfortunately, creating Kotlin collections from JavaScript is still unavailable. We're planning to add this functionality in Kotlin 2.0.20.
>
> 

Support for createInstance()

Starting with Kotlin 2.0.0, you can use the `createInstance()` function from the Kotlin/JS target. Previously, it was only available on the JVM.

This function from the KClass interface creates a new instance of the specified class, which is useful for getting the runtime reference to a Kotlin class.

Support for type-safe plain JavaScript objects

> The `js-plain-objects` plugin is Experimental. It may be dropped or changed at any time. The `js-plain-objects` plugin **only** supports the K2 compiler.
>
> 

To make it easier to work with JavaScript APIs, in Kotlin 2.0.0, we provide a new plugin: `js-plain-objects`, which you can use to create type-safe plain JavaScript objects. The plugin checks your code for any external interfaces that have a `@JsPlainObject` annotation and adds:

- An inline `invoke` operator function inside the companion object that you can use as a constructor.
- A `.copy()` function that you can use to create a copy of your object while adjusting some of its properties.

For example:

```
import kotlinx.js.JsPlainObject

@JsPlainObject
external interface User {
    var name: String
    val age: Int
```

```
    val email: String?
}

fun main() {
    // Creates a JavaScript object
    val user = User(name = "Name", age = 10)
    // Copies the object and adds an email
    val copy = user.copy(age = 11, email = "some@user.com")

    println(JSON.stringify(user))
    // { "name": "Name", "age": 10 }
    println(JSON.stringify(copy))
    // { "name": "Name", "age": 11, "email": "some@user.com" }
}
```

Any JavaScript objects created with this approach are safer because instead of only seeing errors at runtime, you can see them at compile time or even highlighted by your IDE.

Consider this example, which uses a `fetch()` function to interact with a JavaScript API using external interfaces to describe the shape of the JavaScript objects:

```
import kotlinx.js.JsPlainObject

@JsPlainObject
external interface FetchOptions {
    val body: String?
    val method: String
}

// A wrapper for Window.fetch
suspend fun fetch(url: String, options: FetchOptions? = null) = TODO("Add your custom
behavior here")

// A compile-time error is triggered as "metod" is not recognized
// as method
fetch("https://google.com", options = FetchOptions(metod = "POST"))
// A compile-time error is triggered as method is required
fetch("https://google.com", options = FetchOptions(body = "SOME STRING"))
```

In comparison, if you use the `js()` function instead to create your JavaScript objects, errors are only found at runtime or aren't triggered at all:

```
suspend fun fetch(url: String, options: FetchOptions? = null) = TODO("Add your custom
behavior here")

// No error is triggered. As "metod" is not recognized, the wrong method
// (GET) is used.
fetch("https://google.com", options = js("{ metod: 'POST' }"))
```

```
// By default, the GET method is used. A runtime error is triggered as
// body shouldn't be present.
fetch("https://google.com", options = js("{ body: 'SOME STRING' }"))
// TypeError: Window.fetch: HEAD or GET Request cannot have a body
```

To use the `js-plain-objects` plugin, add the following to your `build.gradle(.kts)` file:

> 【Kotlin】

```
plugins {
    kotlin("plugin.js-plain-objects") version "2.0.0"
}
```

> 【Groovy】

```
plugins {
    id "org.jetbrains.kotlin.plugin.js-plain-objects" version "2.0.0"
}
```

Support for npm package manager

Previously, it was only possible for the Kotlin Multiplatform Gradle plugin to use Yarn as a package manager to download and install npm dependencies. From Kotlin 2.0.0, you can use npm as your package manager instead. Using npm as a package manager means that you have one less tool to manage during your setup.

For backward compatibility, Yarn is still the default package manager. To use npm as your package manager, set the following property in your `gradle.properties` file:

```
kotlin.js.yarn = false
```

Changes to compilation tasks

Previously, the `webpack` and `distributeResources` compilation tasks both targeted the same directories. Moreover, the `distribution` task declared the `dist` as its output directory as well. This resulted in overlapping outputs and produced a compilation warning.

So, starting with Kotlin 2.0.0, we've implemented the following changes:

- The `webpack` task now targets a separate folder.
- The `distributeResources` task has been completely removed.
- The `distribution` task now has the `Copy` type and targets the `dist` folder.

Discontinuing legacy Kotlin/JS JAR artifacts

Starting with Kotlin 2.0.0, the Kotlin distribution no longer contains legacy Kotlin/JS artifacts with the `.jar` extension. Legacy artifacts were used in the unsupported old Kotlin/JS compiler and unnecessary for the IR compiler, which uses the `klib` format.

Gradle improvements

Kotlin 2.0.0 is fully compatible with Gradle 6.8.3 through 8.5. You can also use Gradle versions up to the latest Gradle release, but if you do, keep in mind that you might encounter deprecation warnings or some new Gradle features might not work.

This version brings the following changes:

- New Gradle DSL for compiler options in multiplatform projects
- New Compose compiler Gradle plugin
- New attribute to distinguish JVM and Android published libraries
- Improved Gradle dependency handling for CInteropProcess in Kotlin/Native
- Visibility changes in Gradle
- New directory for Kotlin data in Gradle projects
- Kotlin/Native compiler downloaded when needed
- Deprecating old ways of defining compiler options
- Bumped minimum AGP supported version
- New Gradle property for trying the latest language version
- New JSON output format for build reports
- kapt configurations inherit annotation processors from superconfigurations
- Kotlin Gradle plugin no longer uses deprecated Gradle conventions

New Gradle DSL for compiler options in multiplatform projects

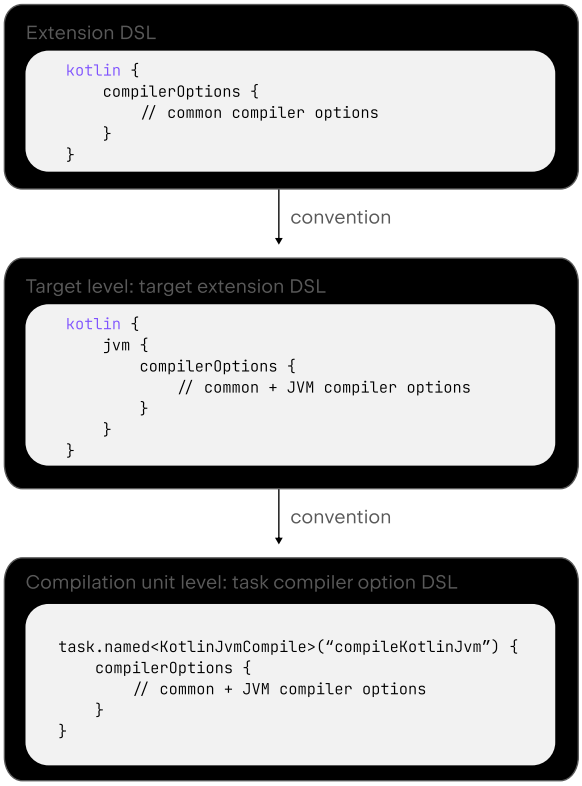
> This feature is Experimental. It may be dropped or changed at any time. Use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack.
>
> 

Prior to Kotlin 2.0.0, configuring compiler options in a multiplatform project with Gradle was only possible at a low level, such as per task, compilation, or source set. To make it easier to configure compiler options more generally in your projects, Kotlin 2.0.0 comes with a new Gradle DSL.

With this new DSL, you can configure compiler options at the extension level for all the targets and shared source sets like `commonMain` and at a target level for a specific target:

```
kotlin {
    compilerOptions {
        // Extension-level common compiler options that are used as defaults
        // for all targets and shared source sets
        allWarningsAsErrors.set(true)
    }
    jvm {
        compilerOptions {
            // Target-level JVM compiler options that are used as defaults
            // for all compilations in this target
            noJdk.set(true)
        }
    }
}
```

The overall project configuration now has three layers. The highest is the extension level, then the target level and the lowest is the compilation unit (which is usually a compilation task):



The settings at a higher level are used as a convention (default) for a lower level:

- The values of extension compiler options are the default for target compiler options, including shared source sets, like `commonMain`, `nativeMain`, and `commonTest`.
- The values of target compiler options are used as the default for compilation unit (task) compiler

options, for example, `compileKotlinJvm` and `compileTestKotlinJvm` tasks.

In turn, configurations made at a lower level override related settings at a higher level:

- Task-level compiler options override related configurations at the target or the extension level.
- Target-level compiler options override related configurations at the extension level.

When configuring your project, keep in mind that some old ways of setting up compiler options have been deprecated.

We encourage you to try the new DSL out in your multiplatform projects and leave feedback in YouTrack, as we plan to make this DSL the recommended approach for configuring compiler options.

New Compose compiler Gradle plugin

The Jetpack Compose compiler, which translates composables into Kotlin code, has now been merged into the Kotlin repository. This will help transition Compose projects to Kotlin 2.0.0, as the Compose compiler will always ship simultaneously with Kotlin. This also bumps the Compose compiler version to 2.0.0.

To use the new Compose compiler in your projects, apply the `org.jetbrains.kotlin.plugin.compose` Gradle plugin in your `build.gradle(.kts)` file and set its version equal to Kotlin 2.0.0.

To learn more about this change and see the migration instructions, see the Compose compiler documentation.

New attribute to distinguish JVM and Android-published libraries

Starting with Kotlin 2.0.0, the `org.gradle.jvm.environment` Gradle attribute is published by default with all Kotlin variants.

The attribute helps distinguish JVM and Android variants of Kotlin Multiplatform libraries. It indicates that a certain library variant is better suited for a certain JVM environment. The target environment could be "android", "standard-jvm", or "no-jvm".

Publishing this attribute should make consuming Kotlin Multiplatform libraries with JVM and Android targets more robust from non-multiplatform clients as well, such as Java-only projects.

If necessary, you can disable attribute publication. To do that, add the following Gradle option to your `gradle.properties` file:

```
kotlin.publishJvmEnvironmentAttribute=false
```

Improved Gradle dependency handling for CInteropProcess in Kotlin/Native

In this release, we enhanced the handling of the `defFile` property to ensure better Gradle task dependency management in Kotlin/Native projects.

Before this update, Gradle builds could fail if the `defFile` property was designated as an output of another task that hadn't been executed yet. The workaround for this issue was to add a dependency on this task:

```
kotlin {
    macosArm64("native") {
        compilations.getByName("main") {
            cinterops {
                val cinterop by creating {
                    defFileProperty.set(createDefFileTask.flatMap { it.defFile.asFile
})
                    project.tasks.named(interopProcessingTaskName).configure {
                        dependsOn(createDefFileTask)
                    }
                }
            }
        }
    }
}
```

To fix this, there is a new `RegularFileProperty` property called `definitionFile`. Now, Gradle lazily verifies the presence of the `definitionFile` property after the connected task has run later in the build process. This new approach eliminates the need for additional dependencies.

The `CInteropProcess` task and the `CInteropSettings` class use the `definitionFile` property instead of `defFile` and `defFileProperty`:

> 【Kotlin】

```
kotlin {
    macosArm64("native") {
        compilations.getByName("main") {
            cinterops {
                val cinterop by creating {
                    definitionFile.set(project.file("def-file.def"))
                }
            }
        }
    }
}
```

【Groovy】

```
kotlin {
    macosArm64("native") {
        compilations.main {
            cinterops {
                cinterop {
                    definitionFile.set(project.file("def-file.def"))
                }
            }
        }
    }
}
```

`defFile` and `defFileProperty` parameters are deprecated.



Visibility changes in Gradle

This change impacts only Kotlin DSL users.



In Kotlin 2.0.0, we've modified the Kotlin Gradle Plugin for better control and safety in your build scripts. Previously, certain Kotlin DSL functions and properties intended for a specific DSL context would inadvertently leak into other DSL contexts. This leakage could lead to the use of incorrect compiler options, settings being applied multiple times, and other misconfigurations:

```
kotlin {
    // Target DSL couldn't access methods and properties defined in the
    // kotlin{} extension DSL
    jvm {
        // Compilation DSL couldn't access methods and properties defined
        // in the kotlin{} extension DSL and Kotlin jvm{} target DSL
        compilations.configureEach {
            // Compilation task DSLs couldn't access methods and
            // properties defined in the kotlin{} extension, Kotlin jvm{}
            // target or Kotlin compilation DSL
            compileTaskProvider.configure {
                // For example:
                explicitApi()
                // ERROR as it is defined in the kotlin{} extension DSL
                mavenPublication {}
```

```
                // ERROR as it is defined in the Kotlin jvm{} target DSL
                defaultSourceSet {}
                // ERROR as it is defined in the Kotlin compilation DSL
            }
        }
    }
}
```

To fix this issue, we've added the `@KotlinGradlePluginDsl` annotation, preventing the exposure of the Kotlin Gradle plugin DSL functions and properties to levels where they are not intended to be available. The following levels are separated from each other:

- Kotlin extension
- Kotlin target
- Kotlin compilation
- Kotlin compilation task

For the most popular cases, we've added compiler warnings with suggestions on how to fix them if your build script is configured incorrectly. For example:

```
kotlin {
    jvm {
        sourceSets.getByName("jvmMain").dependencies {
            implementation("org.jetbrains.kotlinx:kotlinx-coroutines-core-jvm:1.7.3")
        }
    }
}
```

In this case, the warning message for `sourceSets` is:

```
[DEPRECATION] 'sourceSets: NamedDomainObjectContainer<KotlinSourceSet>' is deprecated.
Accessing 'sourceSets' container on the Kotlin target level DSL is deprecated. Conside
r configuring 'sourceSets' on the Kotlin extension level.
```

We would appreciate your feedback on this change! Share your comments directly to Kotlin developers in our #gradle Slack channel. Get a Slack invite.

New directory for Kotlin data in Gradle projects

> Do not commit the `.kotlin` directory to version control. For example, if you are using Git, add `.kotlin` to your project's `.gitignore` file.
>
> 

In Kotlin 1.8.20, the Kotlin Gradle plugin switched to storing its data in the Gradle project cache directory: `<project-root-directory>/.gradle/kotlin`. However, the `.gradle` directory is reserved for Gradle only, and as a result it's not future-proof.

To solve this, as of Kotlin 2.0.0, we will store Kotlin data in your `<project-root-directory>/.kotlin` by default. We will continue to store some data in the `.gradle/kotlin` directory for backward compatibility.

The new Gradle properties you can configure are:

Gradle property	Description
`kotlin.project.persistent.dir`	Configures the location where your project-level data is stored. Default: `<project-root-directory>/.kotlin`
`kotlin.project.persistent.dir.gradle.disableWrite`	A boolean value that controls whether writing Kotlin data to the `.gradle` directory is disabled. Default: `false`

Add these properties to the `gradle.properties` file in your projects for them to take effect.

Kotlin/Native compiler downloaded when needed

Before Kotlin 2.0.0, if you had a Kotlin/Native target configured in the Gradle build script of your multiplatform project, Gradle would always download the Kotlin/Native compiler in the configuration phase.

This happened even if there was no task to compile code for a Kotlin/Native target that was due to run in the execution phase. Downloading the Kotlin/Native compiler in this way was particularly inefficient for users who only wanted to check the JVM or JavaScript code in their projects. For example, to perform tests or checks with their Kotlin project as part of a CI process.

In Kotlin 2.0.0, we changed this behavior in the Kotlin Gradle plugin so that the Kotlin/Native compiler is downloaded in the execution phase and **only** when a compilation is requested for a Kotlin/Native target.

In turn, the Kotlin/Native compiler's dependencies are now downloaded not as a part of the compiler, but in the execution phase as well.

If you encounter any issues with the new behavior, you can temporarily switch back to the previous behavior by adding the following Gradle property to your `gradle.properties` file:

```
kotlin.native.toolchain.enabled=false
```

Starting with Kotlin 1.9.20-Beta, the Kotlin/Native distribution is published to Maven Central along with the CDN.

This allowed us to change how Kotlin looks for and downloads the necessary artifacts. Instead of the CDN, by default, it now uses the Maven repositories that you specified in the `repositories {}` block of your project.

You can temporarily switch this behavior back by setting the following Gradle property in your `gradle.properties` file:

```
kotlin.native.distribution.downloadFromMaven=false
```

Please report any problems to our issue tracker YouTrack. Both of these Gradle properties that change the default behavior are temporary and will be removed in future releases.

Deprecated old ways of defining compiler options

In this release, we continue to refine how you can set up compiler options. It should resolve ambiguity between different ways and make the project configuration more straightforward.

Since Kotlin 2.0.0, the following DSLs for specifying compiler options are deprecated:

- The `kotlinOptions` DSL from the `KotlinCompile` interface that implements all Kotlin compilation tasks. Use `KotlinCompilationTask<CompilerOptions>` instead.
- The `compilerOptions` property with the `HasCompilerOptions` type from the `KotlinCompilation` interface. This DSL was inconsistent with other DSLs and configured the same `KotlinCommonCompilerOptions` object as `compilerOptions` inside the `KotlinCompilation.compileTaskProvider` compilation task, which was confusing.

  Instead, we recommend using the `compilerOptions` property from the Kotlin compilation task:

  ```
  kotlinCompilation.compileTaskProvider.configure {
      compilerOptions { ... }
  }
  ```

  For example:

  ```
  kotlin {
      js(IR) {
          compilations.all {
              compileTaskProvider.configure {
                  compilerOptions.freeCompilerArgs.add("-Xir-minimized-member-names=
  false")
              }
          }
  ```

```
    }
}
```

- The `kotlinOptions` DSL from the `KotlinCompilation` interface.
- The `kotlinOptions` DSL from the `KotlinNativeArtifactConfig` interface, the `KotlinNativeLink` class, and the `KotlinNativeLinkArtifactTask` class. Use the `toolOptions` DSL instead.
- The `dceOptions` DSL from the `KotlinJsDce` interface. Use the `toolOptions` DSL instead.

For more information on how to specify compiler options in the Kotlin Gradle plugin, see How to define options.

Bumped minimum supported AGP version

Starting with Kotlin 2.0.0, the minimum supported Android Gradle plugin version is 7.1.3.

New Gradle property for trying the latest language version

Prior to Kotlin 2.0.0, we had the following Gradle property to try out the new K2 compiler: `kotlin.experimental.tryK2`. Now that the K2 compiler is enabled by default in Kotlin 2.0.0, we decided to evolve this property into a new form that you can use to try the latest language version in your projects: `kotlin.experimental.tryNext`. When you use this property in your `gradle.properties` file, the Kotlin Gradle plugin increments the language version to one above the default value for your Kotlin version. For example, in Kotlin 2.0.0, the default language version is 2.0, so the property configures language version 2.1.

This new Gradle property produces similar metrics in build reports as before with `kotlin.experimental.tryK2`. The language version configured is included in the output. For example:

```
##### 'kotlin.experimental.tryNext' results #####
:app:compileKotlin: 2.1 language version
:lib:compileKotlin: 2.1 language version
##### 100% (2/2) tasks have been compiled with Kotlin 2.1 #####
```

To learn more about how to enable build reports and their content, see Build reports.

New JSON output format for build reports

In Kotlin 1.7.0, we introduced build reports to help track compiler performance. Over time, we've added more metrics to make these reports even more detailed and helpful when investigating performance issues. Previously, the only output format for a local file was the `*.txt` format. In

Kotlin 2.0.0, we support the JSON output format to make it even easier to analyze using other tools.

To configure JSON output format for your build reports, declare the following properties in your `gradle.properties` file:

```
kotlin.build.report.output=json

// The directory to store your build reports
kotlin.build.report.json.directory=my/directory/path
```

Alternatively, you can run the following command:

```
./gradlew assemble -Pkotlin.build.report.output=json -Pkotlin.build.report.json.direct
ory="my/directory/path"
```

Once configured, Gradle generates your build reports in the directory that you specify with the name: `${project_name}-date-time-<sequence_number>.json`.

Here's an example snippet from a build report with JSON output format that contains build metrics and aggregated metrics:

```
"buildOperationRecord": [
    {
     "path": ":lib:compileKotlin",
      "classFqName": "org.jetbrains.kotlin.gradle.tasks.KotlinCompile_Decorated",
      "startTimeMs": 1714730820601,
      "totalTimeMs": 2724,
      "buildMetrics": {
        "buildTimes": {
          "buildTimesNs": {
            "CLEAR_OUTPUT": 713417,
            "SHRINK_AND_SAVE_CURRENT_CLASSPATH_SNAPSHOT_AFTER_COMPILATION": 19699333,
            "IR_TRANSLATION": 281000000,
            "NON_INCREMENTAL_LOAD_CURRENT_CLASSPATH_SNAPSHOT": 14088042,
            "CALCULATE_OUTPUT_SIZE": 1301500,
            "GRADLE_TASK": 2724000000,
            "COMPILER_INITIALIZATION": 263000000,
            "IR_GENERATION": 74000000,
...
          }
        }
...
 "aggregatedMetrics": {
    "buildTimes": {
      "buildTimesNs": {
        "CLEAR_OUTPUT": 782667,
        "SHRINK_AND_SAVE_CURRENT_CLASSPATH_SNAPSHOT_AFTER_COMPILATION": 22031833,
        "IR_TRANSLATION": 333000000,
```

```
        "NON_INCREMENTAL_LOAD_CURRENT_CLASSPATH_SNAPSHOT": 14890292,
        "CALCULATE_OUTPUT_SIZE": 2370750,
        "GRADLE_TASK": 3234000000,
        "COMPILER_INITIALIZATION": 292000000,
        "IR_GENERATION": 89000000,
...
      }
    }
```

kapt configurations inherit annotation processors from superconfigurations

Prior to Kotlin 2.0.0, if you wanted to define a common set of annotation processors in a separate Gradle configuration and extend this configuration in kapt-specific configurations for your subprojects, kapt would skip annotation processing because it couldn't find any annotation processors. In Kotlin 2.0.0, kapt can successfully detect that there are indirect dependencies on your annotation processors.

As an example, for a subproject using Dagger, in your `build.gradle(.kts)` file, use the following configuration:

```
val commonAnnotationProcessors by configurations.creating
configurations.named("kapt") { extendsFrom(commonAnnotationProcessors) }

dependencies {
    implementation("com.google.dagger:dagger:2.48.1")
    commonAnnotationProcessors("com.google.dagger:dagger-compiler:2.48.1")
}
```

In this example, the `commonAnnotationProcessors` Gradle configuration is your common configuration for annotation processing that you want to be used for all your projects. You use the `extendsFrom()` method to add `commonAnnotationProcessors` as a superconfiguration. kapt sees that the `commonAnnotationProcessors` Gradle configuration has a dependency on the Dagger annotation processor. Therefore, kapt includes the Dagger annotation processor in its configuration for annotation processing.

Thanks to Christoph Loy for the implementation!

Kotlin Gradle plugin no longer uses deprecated Gradle conventions

Prior to Kotlin 2.0.0, if you used Gradle 8.2 or higher, the Kotlin Gradle plugin incorrectly used Gradle conventions that had been deprecated in Gradle 8.2. This led to Gradle reporting build deprecations. In Kotlin 2.0.0, the Kotlin Gradle plugin has been updated to no longer trigger these

deprecation warnings when you use Gradle 8.2 or higher.

Standard library

This release brings further stability to the Kotlin standard library and makes even more existing functions common for all platforms:

- Stable replacement of the enum class values generic function
- Stable AutoCloseable interface
- Common protected property AbstractMutableList.modCount
- Common protected function AbstractMutableList.removeRange
- Common String.toCharArray(destination)

Stable replacement of the enum class values generic function

In Kotlin 2.0.0, the `enumEntries<T>()` function becomes Stable. The `enumEntries<T>()` function is a replacement for the generic `enumValues<T>()` function. The new function returns a list of all enum entries for the given enum type `T`. The `entries` property for enum classes was previously introduced and also stabilized to replace the synthetic `values()` function. For more information about the entries property, see What's new in Kotlin 1.8.20.

> The `enumValues<T>()` function is still supported, but we recommend that you use the `enumEntries<T>()` function instead because it has less of a performance impact. Every time you call `enumValues<T>()`, a new array is created, whereas whenever you call `enumEntries<T>()`, the same list is returned each time, which is far more efficient.
>
> 

For example:

```
enum class RGB { RED, GREEN, BLUE }

inline fun <reified T : Enum<T>> printAllValues() {
    print(enumEntries<T>().joinToString { it.name })
}

printAllValues<RGB>()
// RED, GREEN, BLUE
```

Stable AutoCloseable interface

In Kotlin 2.0.0, the common `AutoCloseable` interface becomes Stable. It allows you to easily close resources and includes a couple of useful functions:

- The `use()` extension function, which executes a given block function on the selected resource and then closes it down correctly, whether an exception is thrown or not.
- The `AutoCloseable()` constructor function, which creates instances of the `AutoCloseable` interface.

In the example below, we define the `XMLWriter` interface and assume that there is a resource that implements it. For example, this resource could be a class that opens a file, writes XML content, and then closes it:

```
interface XMLWriter {
    fun document(encoding: String, version: String, content: XMLWriter.() -> Unit)
    fun element(name: String, content: XMLWriter.() -> Unit)
    fun attribute(name: String, value: String)
    fun text(value: String)

    fun flushAndClose()
}

fun writeBooksTo(writer: XMLWriter) {
    val autoCloseable = AutoCloseable { writer.flushAndClose() }
    autoCloseable.use {
        writer.document(encoding = "UTF-8", version = "1.0") {
            element("bookstore") {
                element("book") {
                    attribute("category", "fiction")
                    element("title") { text("Harry Potter and the Prisoner of Azkaban"
) }
                    element("author") { text("J. K. Rowling") }
                    element("year") { text("1999") }
                    element("price") { text("29.99") }
                }
                element("book") {
                    attribute("category", "programming")
                    element("title") { text("Kotlin in Action") }
                    element("author") { text("Dmitry Jemerov") }
                    element("author") { text("Svetlana Isakova") }
                    element("year") { text("2017") }
                    element("price") { text("25.19") }
                }
            }
        }
    }
}
```

Common protected property AbstractMutableList.modCount

In this release, the `modCount` `protected` property of the `AbstractMutableList` interface becomes common. Previously, the `modCount` property was available on each platform but not for the common target. Now, you can create custom implementations of `AbstractMutableList` and access the property in common code.

The property keeps track of the number of structural modifications made to the collection. This includes operations that change the collection size or alter the list in a way that may cause iterations in progress to return incorrect results.

You can use the `modCount` property to register and detect concurrent modifications when implementing a custom list.

Common protected function AbstractMutableList.removeRange

In this release, the `removeRange()` `protected` function of the `AbstractMutableList` interface becomes common. Previously, it was available on each platform but not for the common target. Now, you can create custom implementations of `AbstractMutableList` and override the function in common code.

The function removes elements from this list following the specified range. By overriding this function, you can take advantage of the custom implementations and improve the performance of the list operation.

Common String.toCharArray(destination) function

This release introduces a common `String.toCharArray(destination)` function. Previously, it was only available on the JVM.

Let's compare it with the existing `String.toCharArray()` function. It creates a new `CharArray` that contains characters from the specified string. The new common `String.toCharArray(destination)` function, however, moves `String` characters into an existing destination `CharArray`. This is useful if you already have a buffer that you want to fill:

```
fun main() {
    val myString = "Kotlin is awesome!"
    val destinationArray = CharArray(myString.length)

    // Convert the string and store it in the destinationArray:
    myString.toCharArray(destinationArray)

    for (char in destinationArray) {
        print("$char ")
        // K o t l i n  i s  a w e s o m e !
    }
}
```

Install Kotlin 2.0.0

Starting from IntelliJ IDEA 2023.3 and Android Studio Iguana (2023.2.1) Canary 15, the Kotlin plugin is distributed as a bundled plugin included in your IDE. This means that you can't install the plugin from JetBrains Marketplace anymore.

To update to the new Kotlin version, change the Kotlin version to 2.0.0 in your build scripts.

Kotlin 1.9.20 的新特性

发布于：*2023-11-01*

The Kotlin 1.9.20 release is out, the K2 compiler for all the targets is now in Beta, and Kotlin Multiplatform is now Stable. Additionally, here are some of the main highlights:

- New default hierarchy template for setting up multiplatform projects
- Full support for the Gradle configuration cache in Kotlin Multiplatform
- Custom memory allocator enabled by default in Kotlin/Native
- Performance improvements for the garbage collector in Kotlin/Native
- New and renamed targets in Kotlin/Wasm
- Support for the WASI API in the standard library for Kotlin/Wasm

You can also find a short overview of the updates in this video:

YouTube 视频：What's new in Kotlin 1.9.20

IDE support

The Kotlin plugins that support 1.9.20 are available for:

IDE	Supported versions
IntelliJ IDEA	2023.1.x, 2023.2.x, 2023.x
Android Studio	Hedgehog (2023.1.1), Iguana (2023.2.1)

> Starting from IntelliJ IDEA 2023.3.x and Android Studio Iguana (2023.2.1) Canary 15, the Kotlin plugin is automatically included and updated. All you need to do is update the Kotlin version in your projects.
>
> 

New Kotlin K2 compiler updates

The Kotlin team at JetBrains is continuing to stabilize the new K2 compiler, which will bring major performance improvements, speed up new language feature development, unify all the platforms that Kotlin supports, and provide a better architecture for multiplatform projects.

K2 is currently in **Beta** for all targets. Read more in the release blog post

Support for Kotlin/Wasm

Since this release, the Kotlin/Wasm supports the new K2 compiler. Learn how to enable it in your project.

Preview kapt compiler plugin with K2

> Support for K2 in the kapt compiler plugin is Experimental. Opt-in is required (see details below), and you should use it only for evaluation purposes.
>
> 

In 1.9.20, you can try using the kapt compiler plugin with the K2 compiler. To use the K2 compiler in your project, add the following options to your `gradle.properties` file:

```
kotlin.experimental.tryK2=true
kapt.use.k2=true
```

Alternatively, you can enable K2 for kapt by completing the following steps:

1. In your `build.gradle.kts` file, set the language version to `2.0`.
2. In your `gradle.properties` file, add `kapt.use.k2=true`.

If you encounter any issues when using kapt with the K2 compiler, please report them to our issue tracker.

How to enable the Kotlin K2 compiler

Enable K2 in Gradle

To enable and test the Kotlin K2 compiler, use the new language version with the following compiler option:

```
-language-version 2.0
```

You can specify it in your `build.gradle.kts` file:

```
kotlin {
    sourceSets.all {
        languageSettings {
            languageVersion = "2.0"
        }
    }
}
```

Enable K2 in Maven

To enable and test the Kotlin K2 compiler, update the `<project/>` section of your `pom.xml` file:

```
<properties>
    <kotlin.compiler.languageVersion>2.0</kotlin.compiler.languageVersion>
</properties>
```

Enable K2 in IntelliJ IDEA

To enable and test the Kotlin K2 compiler in IntelliJ IDEA, go to **Settings** | **Build, Execution, Deployment** | **Compiler** | **Kotlin Compiler** and update the **Language Version** field to `2.0 (experimental)`.

Leave your feedback on the new K2 compiler

We would appreciate any feedback you may have!

- Provide your feedback directly to K2 developers on Kotlin Slack – get an invite and join the #k2-early-adopters channel.
- Report any problems you faced with the new K2 compiler on our issue tracker.
- Enable the Send usage statistics option to allow JetBrains to collect anonymous data about K2 usage.

Kotlin/JVM

Starting with version 1.9.20, the compiler can generate classes containing Java 21 bytecode.

Kotlin/Native

Kotlin 1.9.20 includes a Stable memory manager with the new memory allocator enabled by default, performance improvements for the garbage collector, and other updates:

- Custom memory allocator enabled by default
- Performance improvements for the garbage collector
- Incremental compilation of `klib` artifacts
- Managing library linkage issues
- Companion object initialization on class constructor calls
- Opt-in requirement for all cinterop declarations
- Custom message for linker errors
- Removal of the legacy memory manager
- Change to our target tiers policy

Custom memory allocator enabled by default

Kotlin 1.9.20 comes with the new memory allocator enabled by default. It's designed to replace the previous default allocator, `mimalloc`, to make garbage collection more efficient and improve the runtime performance of the Kotlin/Native memory manager.

The new custom allocator divides system memory into pages, allowing independent sweeping in consecutive order. Each allocation becomes a memory block within a page, and the page keeps track of block sizes. Different page types are optimized for various allocation sizes. The consecutive arrangement of memory blocks ensures efficient iteration through all allocated blocks.

When a thread allocates memory, it searches for a suitable page based on the allocation size. Threads maintain a set of pages for different size categories. Typically, the current page for a given size can accommodate the allocation. If not, the thread requests a different page from the shared allocation space. This page may already be available, require sweeping, or have to be created first.

The new allocator allows for multiple independent allocation spaces simultaneously, which will enable the Kotlin team to experiment with different page layouts to improve performance even further.

How to enable the custom memory allocator

Starting with Kotlin 1.9.20, the new memory allocator is the default. No additional setup is required.

If you experience high memory consumption, you can switch back to `mimalloc` or the system allocator with `-Xallocator=mimalloc` or `-Xallocator=std` in your Gradle build script. Please report such issues in YouTrack to help us improve the new memory allocator.

For the technical details of the new allocator's design, see this README.

Performance improvements for the garbage collector

The Kotlin team continues to improve the performance and stability of the new Kotlin/Native memory manager. This release brings a number of significant changes to the garbage collector (GC), including the following 1.9.20 highlights:

-
-

Full parallel mark to reduce the pause time for the GC

Previously, the default garbage collector performed only a partial parallel mark. When the mutator thread was paused, it would mark the GC's start from its own roots, like thread–local variables and the call stack. Meanwhile, a separate GC thread was responsible for marking the start from global roots, as well as the roots of all mutators that were actively running the native code and therefore not paused.

This approach worked well in cases where there were a limited number of global objects and the mutator threads spent a considerable amount of time in a runnable state executing Kotlin code. However, this is not the case for typical iOS applications.

Now the GC uses a full parallel mark that combines paused mutators, the GC thread, and optional marker threads to process the mark queue. By default, the marking process is performed by:

- Paused mutators. Instead of processing their own roots and then being idle while not actively executing code, they contribute to the whole marking process.
- The GC thread. This ensures that at least one thread will perform marking.

This new approach makes the marking process more efficient, reducing the pause time of the GC.

Tracking memory in big chunks to improve the allocation performance

Previously, the GC scheduler tracked the allocation of each object individually. However, neither the new default custom allocator nor the `mimalloc` memory allocator allocates separate storage for each object; they allocate large areas for several objects at once.

In Kotlin 1.9.20, the GC tracks areas instead of individual objects. This speeds up the allocation of small objects by reducing the number of tasks performed on each allocation and, therefore, helps to minimize the garbage collector's memory usage.

Incremental compilation of klib artifacts

> This feature is Experimental. It may be dropped or changed at any time. Opt-in is required (see details below). Use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack.
>
> 

Kotlin 1.9.20 introduces a new compilation time optimization for Kotlin/Native. The compilation of `klib` artifacts into native code is now partially incremental.

When compiling Kotlin source code into native binary in debug mode, the compilation goes through two stages:

1. Source code is compiled into `klib` artifacts.
2. `klib` artifacts, along with dependencies, are compiled into a binary.

To optimize the compilation time in the second stage, the team has already implemented compiler caches for dependencies. They are compiled into native code only once, and the result is reused every time a binary is compiled. But `klib` artifacts built from project sources were always fully recompiled into native code at every project change.

With the new incremental compilation, if the project module change causes only a partial recompilation of source code into `klib` artifacts, just a part of the `klib` is further recompiled into a binary.

To enable incremental compilation, add the following option to your `gradle.properties` file:

```
kotlin.incremental.native=true
```

If you face any issues, report such cases to YouTrack.

Managing library linkage issues

This release improves the way the Kotlin/Native compiler handles linkage issues in Kotlin libraries. Error messages now include more readable declarations as they use signature names instead of hashes, helping you find and fix the issue more easily. Here's an example:

```
No function found for symbol 'org.samples/MyClass.removedFunction|removedFunction(kotl
in.Int;kotlin.String){}[0]'
```

The Kotlin/Native compiler detects linkage issues between third-party Kotlin libraries and reports errors at runtime. You might face such issues if the author of one third-party Kotlin library makes an incompatible change in experimental APIs that another third-party Kotlin library consumes.

Starting with Kotlin 1.9.20, the compiler detects linkage issues in silent mode by default. You can adjust this setting in your projects:

- If you want to record these issues in your compilation logs, enable warnings with the `-Xpartial-linkage-loglevel=WARNING` compiler option.
- It's also possible to raise the severity of reported warnings to compilation errors with `-Xpartial-linkage-loglevel=ERROR`. In this case, the compilation fails, and you get all the errors in the compilation log. Use this option to examine the linkage issues more closely.

```
// An example of passing compiler options in a Gradle build file:
kotlin {
    macosX64("native") {
        binaries.executable()

        compilations.configureEach {
            compilerOptions.configure {
                // To report linkage issues as warnings:
                freeCompilerArgs.add("-Xpartial-linkage-loglevel=WARNING")

                // To raise linkage warnings to errors:
                freeCompilerArgs.add("-Xpartial-linkage-loglevel=ERROR")
            }
        }
    }
}
```

If you face unexpected problems with this feature, you can always opt out with the `-Xpartial-linkage=disable` compiler option. Don't hesitate to report such cases to our issue tracker.

Companion object initialization on class constructor calls

Starting with Kotlin 1.9.20, the Kotlin/Native backend calls static initializers for companion objects in class constructors:

```
class Greeting {
    companion object {
        init {
            print("Hello, Kotlin!")
        }
    }
}

fun main() {
    val start = Greeting() // Prints "Hello, Kotlin!"
}
```

The behavior is now unified with Kotlin/JVM, where a companion object is initialized when the corresponding class matching the semantics of a Java static initializer is loaded (resolved).

Now that the implementation of this feature is more consistent between platforms, it's easier to share code in Kotlin Multiplatform projects.

Opt-in requirement for all cinterop declarations

Starting with Kotlin 1.9.20, all Kotlin declarations generated by the `cinterop` tool from C and Objective-C libraries, like libcurl and libxml, are marked with `@ExperimentalForeignApi`. If the opt-in annotation is missing, your code won't compile.

This requirement reflects the Experimental status of the import of C and Objective-C libraries. We recommend that you confine its use to specific areas in your projects. This will make your migration easier once we begin stabilizing the import.

> As for native platform libraries shipped with Kotlin/Native (like Foundation, UIKit, and POSIX), only some of their APIs need an opt-in with `@ExperimentalForeignApi`. In such cases, you get a warning with an opt-in requirement.
>
> 

Custom message for linker errors

If you're a library author, you can now help your users resolve linker errors with custom messages.

If your Kotlin library depends on C or Objective-C libraries, for example, using the CocoaPods integration, its users need to have these dependent libraries locally on the machine or configure them explicitly in the project build script. If this was not the case, users used to get a confusing "Framework not found" message.

You can now provide a specific instruction or a link in the compilation failure message. To do that, pass the `-Xuser-setup-hint` compiler option to `cinterop` or add a `userSetupHint=message` property to your `.def` file.

Removal of the legacy memory manager

The new memory manager was introduced in Kotlin 1.6.20 and became the default in 1.7.20. Since then, it has been receiving further updates and performance improvements and has become Stable.

The time has come to complete the deprecation cycle and remove the legacy memory manager. If you're still using it, remove the `kotlin.native.binary.memoryModel=strict` option from your `gradle.properties` and follow our Migration guide to make the necessary changes.

Change to our target tiers policy

We've decided to upgrade the requirements for tier 1 support. The Kotlin team is now committed to providing source and binary compatibility between compiler releases for targets eligible for tier 1. They must also be regularly tested with CI tools to be able to compile and run. Currently, tier 1 includes the following targets for macOS hosts:

- `macosX64`
- `macosArm64`
- `iosSimulatorArm64`
- `iosX64`

In Kotlin 1.9.20, we've also removed a number of previously deprecated targets, namely:

- `iosArm32`
- `watchosX86`
- `wasm32`
- `mingwX86`
- `linuxMips32`
- `linuxMipsel32`

See the full list of currently supported targets.

Kotlin Multiplatform

Kotlin 1.9.20 focuses on the stabilization of Kotlin Multiplatform and makes new steps in improving developer experience with the new project wizards and other notable features:

- Kotlin Multiplatform is Stable
- Template for configuring multiplatform projects
- New project wizard
- Full support for the Gradle Configuration cache
- Easier configuration of new standard library versions in Gradle
- Default support for third-party cinterop libraries
- Support for Kotlin/Native compilation caches in Compose Multiplatform projects
- Compatibility guidelines

Kotlin Multiplatform is Stable

The 1.9.20 release marks an important milestone in the evolution of Kotlin: Kotlin Multiplatform has finally become Stable. This means that the technology is safe to use in your projects and 100% ready for production. It also means that further development of Kotlin Multiplatform will continue

according to our strict backward compatibility rules.

Please note that some advanced features of Kotlin Multiplatform are still evolving. When using them, you'll receive a warning that describes the current stability status of the feature you're using. Before using any experimental functionality in IntelliJ IDEA,

you'll need to enable it explicitly in **Settings** | **Advanced Settings** | **Kotlin** | **Experimental Multiplatform**.

- Visit the Kotlin blog to learn more about the Kotlin Multiplatform stabilization and future plans.
- Check out the Multiplatform compatibility guide to see what significant changes were made on the way to stabilization.
- Read about the mechanism of expected and actual declarations, an important part of Kotlin Multiplatform that was also partially stabilized in this release.

Template for configuring multiplatform projects

Starting with Kotlin 1.9.20, the Kotlin Gradle plugin automatically creates shared source sets for popular multiplatform scenarios. If your project setup is one of them, you don't need to configure the source set hierarchy manually. Just explicitly specify the targets necessary for your project.

Setup is now easier thanks to the default hierarchy template, a new feature of the Kotlin Gradle plugin. It's a predefined template of a source set hierarchy built into the plugin. It includes intermediate source sets that Kotlin automatically creates for the targets you declared. See the full template.

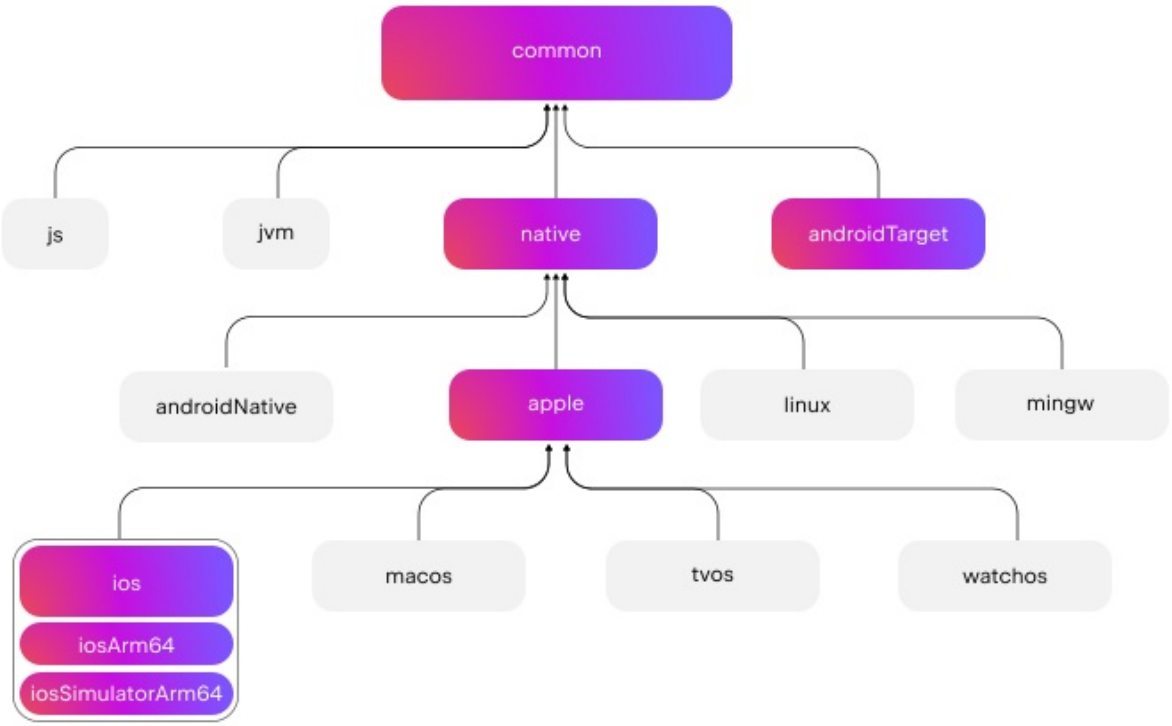
Create your project easier

Consider a multiplatform project that targets both Android and iPhone devices and is developed on an Apple silicon MacBook. Compare how this project is set up between different versions of Kotlin:

Kotlin 1.9.0 and earlier (a standard setup)	Kotlin 1.9.20
```kotlin kotlin { androidTarget() iosArm64() iosSimulatorArm64() sourceSets { val commonMain by getting val iosMain by creating { dependsOn(commonMain) } val iosArm64Main by getting { dependsOn(iosMain) } val iosSimulatorArm64Main by getting { dependsOn(iosMain) } } } ```	```kotlin kotlin { androidTarget() iosArm64() iosSimulatorArm64() // The iosMain source set is created automatically } ```

Notice how the use of the default hierarchy template considerably reduces the amount of boilerplate code needed to set up your project.

When you declare the `androidTarget`, `iosArm64`, and `iosSimulatorArm64` targets in your code, the Kotlin Gradle plugin finds suitable shared source sets from the template and creates them for you. The resulting hierarchy looks like this:

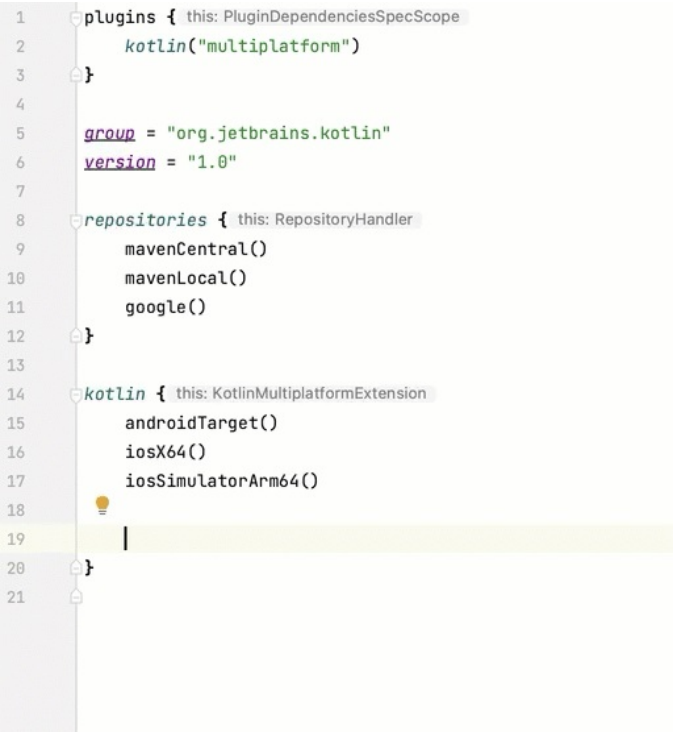


{thumbnail="true" width="350" thumbnail-same-file="true"}

Green source sets are actually created and included in the project, while gray ones from the default template are ignored.

Use completion for source sets

To make it easier to work with the created project structure, IntelliJ IDEA now provides completion for source sets created with the default hierarchy template:



Kotlin also warns you if you attempt to access a source set that doesn't exist because you haven't declared the respective target. In the example below, there is no JVM target (only `androidTarget`, which is not the same). But let's try to use the `jvmMain` source set and see what happens:

```
kotlin {
    androidTarget()
    iosArm64()
    iosSimulatorArm64()

    sourceSets {
        jvmMain {
        }
    }
}
```

In this case, Kotlin reports a warning in the build log:

```
w: Accessed 'source set jvmMain' without registering the jvm target:
  kotlin {
      jvm() /* <- register the 'jvm' target */

      sourceSets.jvmMain.dependencies {

      }
  }
```

Set up the target hierarchy

Starting with Kotlin 1.9.20, the default hierarchy template is automatically enabled. In most cases, no additional setup is required.

However, if you're migrating existing projects created before 1.9.20, you might encounter a warning if you had previously introduced intermediate sources manually with `dependsOn()` calls. To solve this issue, do the following:

- If your intermediate source sets are currently covered by the default hierarchy template, remove all manual `dependsOn()` calls and source sets created with `by creating` constructions.

  To check the list of all default source sets, see the full hierarchy template.

- If you want to have additional source sets that the default hierarchy template doesn't provide, for example, one that shares code between a macOS and a JVM target, adjust the hierarchy by reapplying the template explicitly with `applyDefaultHierarchyTemplate()` and configuring additional source sets manually as usual with `dependsOn()`:

  ```
  kotlin {
      jvm()
      macosArm64()
      iosArm64()
      iosSimulatorArm64()

      // Apply the default hierarchy explicitly. It'll create, for example, the iosM
  ain source set:
      applyDefaultHierarchyTemplate()

      sourceSets {
          // Create an additional jvmAndMacos source set
          val jvmAndMacos by creating {
              dependsOn(commonMain.get())
          }

          macosArm64Main.get().dependsOn(jvmAndMacos)
          jvmMain.get().dependsOn(jvmAndMacos)
      }
  }
  ```

- If there are already source sets in your project that have the exact same names as those generated by the template but that are shared among different sets of targets, there's currently no way to modify the default `dependsOn` relations between the template's source sets.

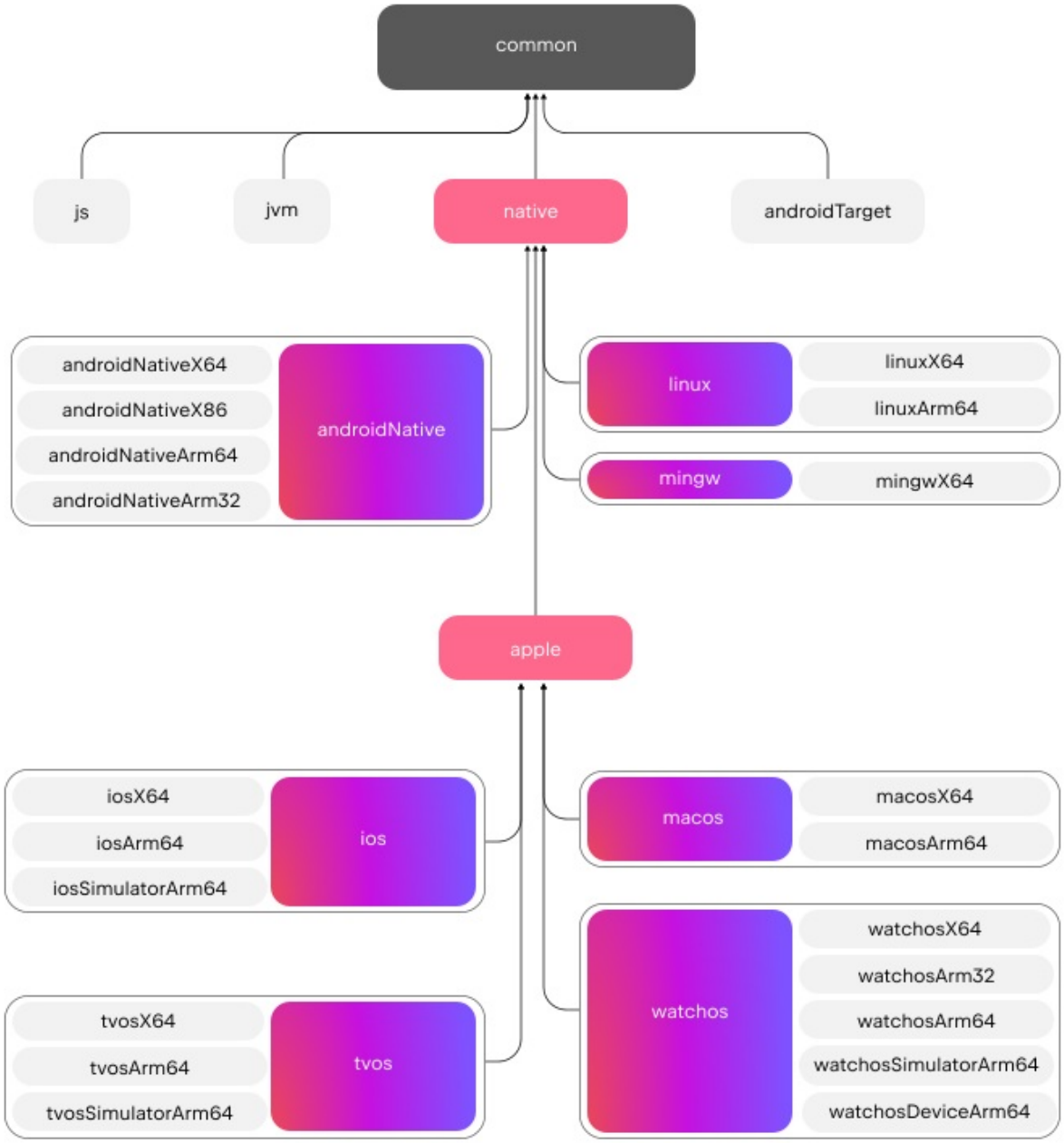
  One option you have here is to find different source sets for your purposes, either in the default hierarchy template or ones that have been manually created. Another is to opt out of the template completely.

To opt out, add `kotlin.mpp.applyDefaultHierarchyTemplate=false` to your `gradle.properties` and configure all other source sets manually.

We're currently working on an API for creating your own hierarchy templates to simplify the setup process in such cases.

See the full hierarchy template

When you declare the targets to which your project compiles, the plugin picks the shared source sets from the template accordingly and creates them in your project.



> This example only shows the production part of the project, omitting the `Main` suffix (for example, using `common` instead of `commonMain`). However, everything is the same for `*Test` sources as well.
>
> 

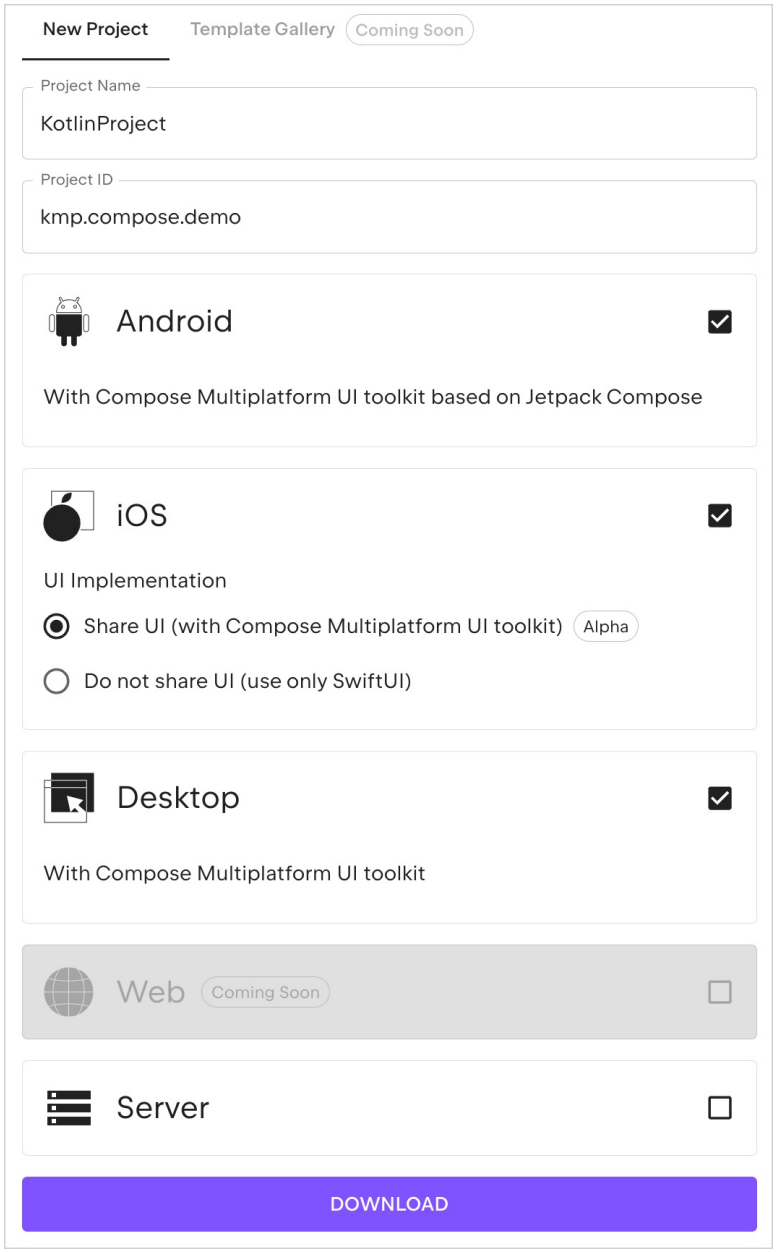
New project wizard

The JetBrains team is introducing a new way of creating cross–platform projects – the Kotlin Multiplatform web wizard.

This first implementation of the new Kotlin Multiplatform wizard covers the most popular Kotlin Multiplatform use cases. It incorporates all the feedback about previous project templates and makes the architecture as robust and reliable as possible.

The new wizard has a distributed architecture that allows us to have a unified backend and different frontends, with the web version being the first step. We're considering both implementing an IDE version and creating a command-line tool in the future. On the web, you always get the latest version of the wizard, while in IDEs you'll need to wait for the next release.

With the new wizard, project setup is easier than ever. You can tailor your projects to your needs by choosing the target platforms for mobile, server, and desktop development. We also plan to add web development in future releases.



The new project wizard is now the preferred way to create cross–platform projects with Kotlin. Since 1.9.20, the Kotlin plugin no longer provides a **Kotlin Multiplatform** project wizard in IntelliJ IDEA.

The new wizard will guide you easily through the initial setup, making the onboarding process much smoother. If you encounter any issues, please report them to YouTrack to help us improve your experience with the wizard.

Create project →

Full support for the Gradle configuration cache in Kotlin Multiplatform

Previously, we introduced a preview of the Gradle configuration cache, which was available for Kotlin multiplatform libraries. With 1.9.20, the Kotlin Multiplatform plugin takes a step further.

It now supports the Gradle configuration cache in the Kotlin CocoaPods Gradle plugin, as well as in the integration tasks that are necessary for Xcode builds, like `embedAndSignAppleFrameworkForXcode`.

Now all multiplatform projects can take advantage of the improved build time. The Gradle configuration cache speeds up the build process by reusing the results of the configuration phase for subsequent builds. For more details and setup instructions, see the Gradle documentation.

Easier configuration of new standard library versions in Gradle

When you create a multiplatform project, a dependency for the standard library (`stdlib`) is added automatically to each source set. This is the easiest way to get started with your multiplatform projects.

Previously, if you wanted to configure a dependency on the standard library manually, you needed to configure it for each source set individually. From `kotlin-stdlib:1.9.20` onward, you only need to configure the dependency **once** in the `commonMain` root source set:

Standard library version 1.9.10 and earlier	Standard library version 1.9.20
```kotlin kotlin { sourceSets { // For the common source set val commonMain by getting { dependencies { implementation("org.jetbrains.kotlin:kotlin-stdlib-common:1.9.10") } } // For the JVM source set val jvmMain by getting { dependencies { implementation("org.jetbrains.kotlin:kotlin-stdlib:1.9.10") } } // For the JS source set val jsMain by getting { dependencies { implementation("org.jetbrains.kotlin:kotlin-stdlib-js:1.9.10") } } } } ```	```kotlin kotlin { sourceSets { commonMain { dependencies { implementation("org.jetbrains.kotlin:kotlin-stdlib:1.9.20") } } } } ```

This change was made possible by including new information in the Gradle metadata of the standard library. This allows Gradle to automatically resolve the correct standard library artifacts for the other source sets.

Default support for third-party cinterop libraries

Kotlin 1.9.20 adds default support (rather than support by opt-in) for all cinterop dependencies in projects that have the Kotlin CocoaPods Gradle plugin applied.

This means you can now share more native code without being limited by platform–specific dependencies. For example, you can add dependencies on Pod libraries to the `iosMain` shared source set.

Previously, this only worked with platform-specific libraries shipped with a Kotlin/Native distribution (like Foundation, UIKit, and POSIX). All third-party Pod libraries are now available in shared source sets by default. You no longer need to specify a separate Gradle property to support them.

Support for Kotlin/Native compilation caches in Compose Multiplatform projects

This release resolves a compatibility issue with the Compose Multiplatform compiler plugin, which mostly affected Compose Multiplatform projects for iOS.

To work around this issue, you had to disable caching by using the `kotlin.native.cacheKind=none` Gradle property. However, this workaround came at a performance cost: It slowed down compilation time as caching didn't work in the Kotlin/Native compiler.

Now that the issue is fixed, you can remove `kotlin.native.cacheKind=none` from your `gradle.properties` file and enjoy the improved compilation times in your Compose Multiplatform projects.

For more tips on improving compilation times, see the Kotlin/Native documentation.

Compatibility guidelines

When configuring your projects, check the Kotlin Multiplatform Gradle plugin's compatibility with the available Gradle, Xcode, and Android Gradle plugin (AGP) versions:

Kotlin Multiplatform Gradle plugin	Gradle	Android Gradle plugin	Xcode
1.9.20	7.5 and later	7.4.2–8.2	15.0. See details below

As of this release, the recommended version of Xcode is 15.0. Libraries delivered with Xcode 15.0 are fully supported, and you can access them from anywhere in your Kotlin code.

However, XCode 14.3 should still work in the majority of cases. Keep in mind that if you use version 14.3 on your local machine, libraries delivered with Xcode 15 will be visible but not accessible.

Kotlin/Wasm

In 1.9.20, Kotlin Wasm reached the Alpha level of stability.

- Compatibility with Wasm GC phase 4 and final opcodes
- New `wasm-wasi` target, and the renaming of the `wasm` target to `wasm-js`
- Support for the WASI API in standard library

- Kotlin/Wasm API improvements

> Kotlin Wasm is Alpha. It is subject to change at any time. Use it only for evaluation purposes.
>
> We would appreciate your feedback on it in YouTrack.
>
> 

Compatibility with Wasm GC phase 4 and final opcodes

Wasm GC moves to the final phase and it requires updates of opcodes – constant numbers used in the binary representation. Kotlin 1.9.20 supports the latest opcodes, so we strongly recommend that you update your Wasm projects to the latest version of Kotlin. We also recommend using the latest versions of browsers with the Wasm environment:

- Version 119 or newer for Chrome and Chromium–based browsers.
- Version 119 or newer for Firefox. Note that in Firefox 119, you need to turn on Wasm GC manually.

New wasm-wasi target, and the renaming of the wasm target to wasm-js

In this release, we're introducing a new target for Kotlin/Wasm – `wasm-wasi`. We're also renaming the `wasm` target to `wasm-js`. In the Gradle DSL, these targets are available as `wasmWasi {}` and `wasmJs {}`, respectively.

To use these targets in your project, update the `build.gradle.kts` file:

```
kotlin {
    wasmWasi {
        // ...
    }
    wasmJs {
        // ...
    }
}
```

The previously introduced `wasm {}` block has been deprecated in favor of `wasmJs {}`.

To migrate your existing Kotlin/Wasm project, do the following:

- In the `build.gradle.kts` file, rename the `wasm {}` block to `wasmJs {}`.
- In your project structure, rename the `wasmMain` directory to `wasmJsMain`.

Support for the WASI API in the standard library

In this release, we have included support for WASI, a system interface for the Wasm platform. WASI support makes it easier for you to use Kotlin/Wasm outside of browsers, for example in server–side applications, by offering a standardized set of APIs for accessing system resources. In addition, WASI provides capability–based security – another layer of security when accessing external resources.

To run Kotlin/Wasm applications, you need a VM that supports Wasm Garbage Collection (GC), for example, Node.js or Deno. Wasmtime, WasmEdge, and others are still working towards full Wasm GC support.

To import a WASI function, use the `@WasmImport` annotation:

```
import kotlin.wasm.WasmImport

@WasmImport("wasi_snapshot_preview1", "clock_time_get")
private external fun wasiRawClockTimeGet(clockId: Int, precision: Long, resultPtr: Int)
: Int
```

You can find a full example in our GitHub repository.

> It isn't possible to use interoperability with JavaScript, while targeting `wasmWasi`.

Kotlin/Wasm API improvements

This release delivers several quality-of-life improvements to the Kotlin/Wasm API. For example, you're no longer required to return a value for DOM event listeners:

Before 1.9.20	In 1.9.20
```kotlin fun main() { window.onload = { document.body?.sayHello() null } } ```	```kotlin fun main() { window.onload = { document.body?.sayHello() } } ```

Gradle

Kotlin 1.9.20 is fully compatible with Gradle 6.8.3 through 8.1. You can also use Gradle versions up to the latest Gradle release, but if you do, keep in mind that you might encounter deprecation warnings or some new Gradle features might not work.

This version brings the following changes:

- Support for test fixtures to access internal declarations
- New property to configure paths to Konan directories
- New build report metrics for Kotlin/Native tasks

Support for test fixtures to access internal declarations

In Kotlin 1.9.20, if you use Gradle's `java-test-fixtures` plugin, then your test fixtures now have access to `internal` declarations within main source set classes. In addition, any test sources can also see any `internal` declarations within test fixtures classes.

New property to configure paths to Konan directories

In Kotlin 1.9.20, the `kotlin.data.dir` Gradle property is available to customize your path to the `~/.konan` directory so that you don't have to configure it through the environment variable `KONAN_DATA_DIR`.

Alternatively, you can use the `-Xkonan-data-dir` compiler option to configure your custom path to the `~/.konan` directory via the `cinterop` and `konanc` tools.

New build report metrics for Kotlin/Native tasks

In Kotlin 1.9.20, Gradle build reports now include metrics for Kotlin/Native tasks. Here is an example of a build report containing these metrics:

```
Total time for Kotlin tasks: 20.81 s (93.1 % of all tasks time)
Time   |% of Kotlin time|Task
15.24 s|73.2 %          |:compileCommonMainKotlinMetadata
5.57 s |26.8 %          |:compileNativeMainKotlinMetadata

Task ':compileCommonMainKotlinMetadata' finished in 15.24 s
Task info:
  Kotlin language version: 2.0
Time metrics:
  Total Gradle task time: 15.24 s
  Spent time before task action: 0.16 s
  Task action before worker execution: 0.21 s
  Run native in process: 2.70 s
    Run entry point: 2.64 s
Size metrics:
  Start time of task action: 2023-07-27T11:04:17

Task ':compileNativeMainKotlinMetadata' finished in 5.57 s
Task info:
  Kotlin language version: 2.0
Time metrics:
  Total Gradle task time: 5.57 s
  Spent time before task action: 0.04 s
  Task action before worker execution: 0.02 s
  Run native in process: 1.48 s
    Run entry point: 1.47 s
Size metrics:
  Start time of task action: 2023-07-27T11:04:32
```

In addition, the `kotlin.experimental.tryK2` build report now includes any Kotlin/Native tasks that were compiled and lists the language version used:

```
##### 'kotlin.experimental.tryK2' results #####
:lib:compileCommonMainKotlinMetadata: 2.0 language version
:lib:compileKotlinJvm: 2.0 language version
:lib:compileKotlinIosArm64: 2.0 language version
:lib:compileKotlinIosSimulatorArm64: 2.0 language version
:lib:compileKotlinLinuxX64: 2.0 language version
:lib:compileTestKotlinJvm: 2.0 language version
:lib:compileTestKotlinIosSimulatorArm64: 2.0 language version
:lib:compileTestKotlinLinuxX64: 2.0 language version
##### 100% (8/8) tasks have been compiled with Kotlin 2.0 #####
```

> If you use Gradle 8.0, you might come across some problems with build reports, especially when Gradle configuration caching is enabled. This is a known issue, which is fixed in Gradle 8.1 and later.
>
> 

Standard library

In Kotlin 1.9.20, the Kotlin/Native standard library becomes Stable, and there are some new features:

- Replacement of the Enum class values generic function
- Improved performance of HashMap operations in Kotlin/JS

Replacement of the Enum class values generic function

> This feature is Experimental. It may be dropped or changed at any time. Opt-in is required (see details below). Use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack.
>
> 

In Kotlin 1.9.0, the `entries` property for enum classes became Stable. The `entries` property is a modern and performant replacement for the synthetic `values()` function. As part of Kotlin 1.9.20, there is a replacement for the generic `enumValues<T>()` function: `enumEntries<T>()`.

> The `enumValues<T>()` function is still supported, but we recommend that you use the `enumEntries<T>()` function instead because it has less performance impact. Every time you call `enumValues<T>()`, a new array is created, whereas whenever you call `enumEntries<T>()`, the same list is returned each time, which is far more efficient.



For example:

```
enum class RGB { RED, GREEN, BLUE }

@OptIn(ExperimentalStdlibApi::class)
inline fun <reified T : Enum<T>> printAllValues() {
    print(enumEntries<T>().joinToString { it.name })
}

printAllValues<RGB>()
// RED, GREEN, BLUE
```

How to enable the enumEntries function

To try this feature, opt in with `@OptIn(ExperimentalStdlibApi)` and use language version 1.9 or later. If you use the latest version of the Kotlin Gradle plugin, you don't need to specify the language version to test the feature.

The Kotlin/Native standard library becomes Stable

In Kotlin 1.9.0, we explained the actions we've taken to bring the Kotlin/Native standard library closer to our goal of stabilization. In Kotlin 1.9.20, we finally conclude this work and make the Kotlin/Native standard library Stable. Here are some highlights from this release:

- The `Vector128` class was moved from the `kotlin.native` package to the `kotlinx.cinterop` package.
- The opt-in requirement level for `ExperimentalNativeApi` and `NativeRuntimeApi` annotations, which were introduced as part of Kotlin 1.9.0, has been raised from `WARNING` to `ERROR`.
- Kotlin/Native collections now detect concurrent modifications, for example, in the `ArrayList` and `HashMap` collections.
- The `printStackTrace()` function from the `Throwable` class now prints to `STDERR` instead of `STDOUT`.

  > The output format of `printStackTrace()` isn't Stable and is subject to change.
  >
  > 

Improvements to the Atomics API

In Kotlin 1.9.0, we said that the Atomics API would be ready to become Stable when the Kotlin/Native standard library becomes Stable. Kotlin 1.9.20 includes the following additional changes:

- Experimental `AtomicIntArray`, `AtomicLongArray`, and `AtomicArray<T>` classes are introduced. These new classes are designed specifically to be consistent with Java's atomic arrays so that in the future, they can be included in the common standard library.

  > The `AtomicIntArray`, `AtomicLongArray`, and `AtomicArray<T>` classes are Experimental. They may be dropped or changed at any time. To try them, opt in with `@OptIn(ExperimentalStdlibApi)`. Use them only for evaluation purposes. We would appreciate your feedback in YouTrack.
  >
  > 

- In the `kotlin.native.concurrent` package, the Atomics API that was deprecated in Kotlin 1.9.0 with deprecation level `WARNING` has had its deprecation level raised to `ERROR`.
- In the `kotlin.concurrent` package, member functions of the `AtomicInt` and `AtomicLong` classes that had deprecation level: `ERROR`, have been removed.
- All member functions of the `AtomicReference` class now use atomic intrinsic functions.

For more information on all of the changes in Kotlin 1.9.20, see our YouTrack ticket.

Improved performance of HashMap operations in Kotlin/JS

Kotlin 1.9.20 improves the performance of `HashMap` operations and reduces their memory footprint in Kotlin/JS. Internally, Kotlin/JS has changed its internal implementation to open addressing. This means that you should see performance improvements when you:

- Insert new elements into a `HashMap`.
- Search for existing elements in a `HashMap`.
- Iterate through keys or values in a `HashMap`.

Documentation updates

The Kotlin documentation has received some notable changes:

- The JVM Metadata API reference – Explore how you can parse metadata with Kotlin/JVM.
- Time measurement guide – Learn how to calculate and measure time in Kotlin.
- Improved Collections chapter in the tour of Kotlin – Learn the fundamentals of the Kotlin programming language with chapters including both theory and practice.
- Definitely non-nullable types – Learn about definitely non-nullable generic types.
- Improved Arrays page – Learn about arrays and when to use them.
- Expected and actual declarations in Kotlin Multiplatform – Learn about the Kotlin mechanism of expected and actual declarations in Kotlin Multiplatform.

Install Kotlin 1.9.20

Check the IDE version

IntelliJ IDEA 2023.1.x and 2023.2.x automatically suggest updating the Kotlin plugin to version 1.9.20. IntelliJ IDEA 2023.3 will include the Kotlin 1.9.20 plugin.

Android Studio Hedgehog (231) and Iguana (232) will support Kotlin 1.9.20 in their upcoming releases.

The new command–line compiler is available for download on the GitHub release page.

Configure Gradle settings

To download Kotlin artifacts and dependencies, update your `settings.gradle(.kts)` file to use the Maven Central repository:

```
pluginManagement {
    repositories {
        mavenCentral()
        gradlePluginPortal()
    }
}
```

If the repository is not specified, Gradle uses the sunset JCenter repository, which could lead to issues with Kotlin artifacts.

Kotlin 1.9.0 的新特性

发布于：*2023-07-06*

The Kotlin 1.9.0 release is out and the K2 compiler for the JVM is now in **Beta**. Additionally, here are some of the main highlights:

- 新版 Kotlin K2 编译器更新
- 稳定版枚举类的 values 函数替代
- 稳定版用于前闭后开区间的 `..<` 操作符
- 新增按名称获取正则表达式捕获组的公共函数
- 新增用于创建父目录路径的实用程序
- Kotlin 多平台 Gradle 配置缓存预览版
- Kotlin 多平台 Android 目标平台支持的变更
- Kotlin/Native 自定义内存分配器预览版
- Kotlin/Native 的库链接
- Kotlin/Wasm 尺寸相关的优化

You can also find a short overview of the updates in this video:

YouTube 视频：What's new in Kotlin 1.9.0

IDE 支持

The Kotlin plugins that support 1.9.0 are available for:

IDE	Supported versions
IntelliJ IDEA	2022.3.x, 2023.1.x
Android Studio	Giraffe (223), Hedgehog (231)*

*The Kotlin 1.9.0 plugin will be included with Android Studio Giraffe (223) and Hedgehog (231) in their upcoming releases.

The Kotlin 1.9.0 plugin will be included with IntelliJ IDEA 2023.2 in the upcoming releases.

> To download Kotlin artifacts and dependencies, configure your Gradle settings to use the Maven Central Repository.
>
> 

新版 Kotlin K2 编译器更新

The Kotlin team at JetBrains continues to stabilize the K2 compiler, and the 1.9.0 release introduces further advancements. The K2 compiler for the JVM is now in **Beta**.

There's now also basic support for Kotlin/Native and multiplatform projects.

kapt 编译器插件与 K2 编译器的兼容性

You can use the kapt plugin in your project along with the K2 compiler, but with some restrictions. Despite setting `languageVersion` to `2.0`, the kapt compiler plugin still utilizes the old compiler.

If you execute the kapt compiler plugin within a project where `languageVersion` is set to `2.0`, kapt will automatically switch to `1.9` and disable specific version compatibility checks. This behavior is equivalent to including the following command arguments:

- `-Xskip-metadata-version-check`
- `-Xskip-prerelease-check`
- `-Xallow-unstable-dependencies`

These checks are exclusively disabled for kapt tasks. All other compilation tasks will continue to utilize the new K2 compiler.

If you encounter any issues when using kapt with the K2 compiler, please report them to our issue tracker.

在项目中试用 K2 编译器

Starting with 1.9.0 and until the release of Kotlin 2.0, you can easily test the K2 compiler by adding the `kotlin.experimental.tryK2=true` Gradle property to your `gradle.properties` file. You can also run the following command:

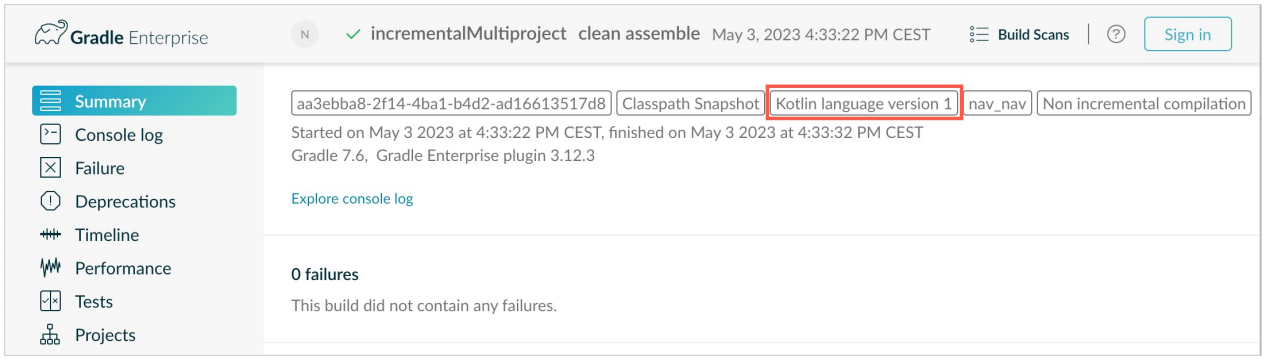
```
./gradlew assemble -Pkotlin.experimental.tryK2=true
```

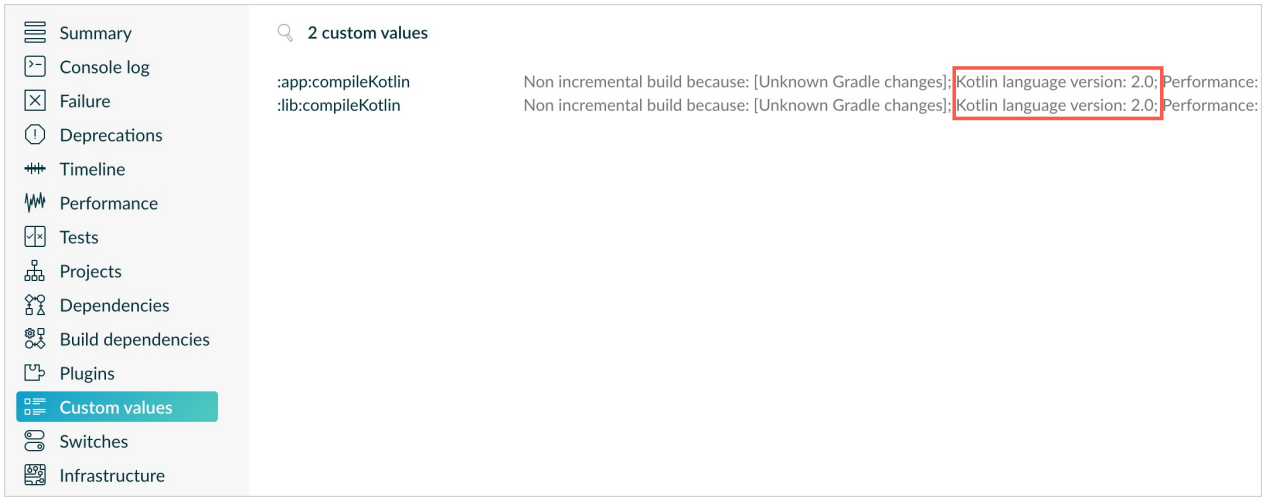
This Gradle property automatically sets the language version to 2.0 and updates the build report with the number of Kotlin tasks compiled using the K2 compiler compared to the current compiler:

```
##### 'kotlin.experimental.tryK2' results (Kotlin/Native not checked) #####
:lib:compileKotlin: 2.0 language version
:app:compileKotlin: 2.0 language version
##### 100% (2/2) tasks have been compiled with Kotlin 2.0 #####
```

Gradle 构建报告

Gradle build reports now show whether the current or the K2 compiler was used to compile the code. In Kotlin 1.9.0, you can see this information in your Gradle build scans:





You can also find the Kotlin version used in the project right in the build report:

```
Task info:
  Kotlin language version: 1.9
```

> If you use Gradle 8.0, you might come across some problems with build reports, especially when Gradle configuration caching is enabled. This is a known issue, fixed in Gradle 8.1 and later.
>
> 

当前 K2 编译器的限制

Enabling K2 in your Gradle project comes with certain limitations that can affect projects using Gradle versions below 8.3 in the following cases:

- Compilation of source code from `buildSrc`.
- Compilation of Gradle plugins in included builds.
- Compilation of other Gradle plugins if they are used in projects with Gradle versions below 8.3.
- Building Gradle plugin dependencies.

If you encounter any of the problems mentioned above, you can take the following steps to address them:

- Set the language version for `buildSrc`, any Gradle plugins, and their dependencies:

```
kotlin {
    compilerOptions {
        languageVersion.set(org.jetbrains.kotlin.gradle.dsl.KotlinVersion.KOTLIN_1_9)
        apiVersion.set(org.jetbrains.kotlin.gradle.dsl.KotlinVersion.KOTLIN_1_9)
    }
}
```

- Update the Gradle version in your project to 8.3 when it becomes available.

请反馈关于新版 K2 编译器的问题

We'd appreciate any feedback you may have!

- Provide your feedback directly to K2 developers Kotlin's Slack – get an invite and join the #k2-early-adopters channel.
- Report any problems you've faced with the new K2 compiler on our issue tracker.
- Enable the **Send usage statistics** option to allow JetBrains to collect anonymous data about K2 usage.

语言

In Kotlin 1.9.0, we're stabilizing some new language features that were introduced earlier:

- 枚举类的 values 函数替代
- 与数据类相称的数据对象
- 在内联类中支持有主体的次构造函数

稳定版枚举类的 values 函数替代

In 1.8.20, the `entries` property for enum classes was introduced as an Experimental feature. The `entries` property is a modern and performant replacement for the synthetic `values()` function. In 1.9.0, the `entries` property is Stable.

> The `values()` function is still supported, but we recommend that you use the `entries` property instead.
>
> 

```
enum class Color(val colorName: String, val rgb: String) {
    RED("Red", "#FF0000"),
    ORANGE("Orange", "#FF7F00"),
    YELLOW("Yellow", "#FFFF00")
}

fun findByRgb(rgb: String): Color? = Color.entries.find { it.rgb == rgb }
```

For more information about the `entries` property for enum classes, see What's new in Kotlin 1.8.20.

稳定版与数据类相称的数据对象

Data object declarations, which were introduced in Kotlin 1.8.20, are now Stable. This includes the functions added for symmetry with data classes: `toString()`, `equals()`, and `hashCode()`.

This feature is particularly useful with `sealed` hierarchies (like a `sealed class` or `sealed interface` hierarchy), because `data object` declarations can be used conveniently alongside `data class` declarations. In this example, declaring `EndOfFile` as a `data object` instead of a plain `object` means that it automatically has a `toString()` function without the need to override it manually. This maintains symmetry with the accompanying data class definitions.

```
sealed interface ReadResult
data class Number(val number: Int) : ReadResult
data class Text(val text: String) : ReadResult
data object EndOfFile : ReadResult

fun main() {
    println(Number(7)) // Number(number=7)
    println(EndOfFile) // EndOfFile
}
```

For more information, see What's new in Kotlin 1.8.20.

在内联类中支持有主体的次构造函数

Starting with Kotlin 1.9.0, the use of secondary constructors with bodies in inline value classes is available by default:

```
@JvmInline
value class Person(private val fullName: String) {
    // Allowed since Kotlin 1.4.30:
    init {
        check(fullName.isNotBlank()) {
            "Full name shouldn't be empty"
        }
    }
    // Allowed by default since Kotlin 1.9.0:
    constructor(name: String, lastName: String) : this("$name $lastName") {
        check(lastName.isNotBlank()) {
            "Last name shouldn't be empty"
        }
    }
}
```

Previously, Kotlin allowed only public primary constructors in inline classes. As a result, it was impossible to encapsulate underlying values or create an inline class that would represent some constrained values.

As Kotlin developed, these issues were fixed. Kotlin 1.4.30 lifted restrictions on `init` blocks and then Kotlin 1.8.20 came with a preview of secondary constructors with bodies. They are now available by default. Learn more about the development of Kotlin inline classes in this KEEP.

Kotlin/JVM

Starting with version 1.9.0, the compiler can generate classes with a bytecode version corresponding to JVM 20. In addition, the deprecation of the `JvmDefault` annotation and legacy `-Xjvm-default` modes continues.

弃用 JvmDefault 注解与旧版 -Xjvm-default 模式

Starting from Kotlin 1.5, the usage of the `JvmDefault` annotation has been deprecated in favor of the newer `-Xjvm-default` modes: `all` and `all-compatibility`. With the introduction of `JvmDefaultWithoutCompatibility` in Kotlin 1.4 and `JvmDefaultWithCompatibility` in Kotlin 1.6, these modes offer comprehensive control over the generation of `DefaultImpls` classes, ensuring seamless compatibility with older Kotlin code.

Consequently in Kotlin 1.9.0, the `JvmDefault` annotation no longer holds any significance and has been marked as deprecated, resulting in an error. It will eventually be removed from Kotlin.

Kotlin/Native

Among other improvements, this release brings further advancements to the Kotlin/Native memory manager that should enhance its robustness and performance:

- 自定义内存分配器预览版
- 主线程上的 Objective-C 或 Swift 对象释放钩子
- 在 Kotlin/Native 中访问常量值时不初始化对象
- 能够为 iOS 模拟器测试配置独立模式
- Kotlin/Native 的库链接

自定义内存分配器预览版

Kotlin 1.9.0 introduces the preview of a custom memory allocator. Its allocation system improves the runtime performance of the Kotlin/Native memory manager.

The current object allocation system in Kotlin/Native uses a general-purpose allocator that doesn't have the functionality for efficient garbage collection. To compensate, it maintains thread-local linked lists of all allocated objects before the garbage collector (GC) merges them into a single list, which can be iterated during sweeping. This approach comes with several performance downsides:

- The sweeping order lacks memory locality and often results in scattered memory access patterns, leading to potential performance issues.
- Linked lists require additional memory for each object, increasing memory usage, particularly when dealing with many small objects.
- The single list of allocated objects makes it challenging to parallelize sweeping, which can cause memory usage problems when mutator threads allocate objects faster than the GC thread can collect them.

To address these issues, Kotlin 1.9.0 introduces a preview of the custom allocator. It divides system memory into pages, allowing independent sweeping in consecutive order. Each allocation becomes a memory block within a page, and the page keeps track of block sizes. Different page types are optimized for various allocation sizes. The consecutive arrangement of memory blocks ensures efficient iteration through all allocated blocks.

When a thread allocates memory, it searches for a suitable page based on the allocation size. Threads maintain a set of pages for different size categories. Typically, the current page for a given size can accommodate the allocation. If not, the thread requests a different page from the shared allocation space. This page may already be available, require sweeping, or should be created first.

The new allocator allows having multiple independent allocation spaces simultaneously, which will allow the Kotlin team to experiment with different page layouts to improve performance even further.

For more information on the design of the new allocator, see this README.

如何启用

Add the `-Xallocator=custom` compiler option:

```
kotlin {
    macosX64("native") {
        binaries.executable()

        compilations.configureEach {
            compilerOptions.configure {
                freeCompilerArgs.add("-Xallocator=custom")
            }
        }
    }
}
```

敬请反馈

We would appreciate your feedback in YouTrack to improve the custom allocator.

主线程上的 Objective-C 或 Swift 对象释放钩子

Starting with Kotlin 1.9.0, the Objective-C or Swift object deallocation hook is called on the main thread if the object is passed to Kotlin there. The way the Kotlin/Native memory manager previously handled references to Objective-C objects could lead to memory leaks. We believe the new behavior should improve the robustness of the memory manager.

Consider an Objective-C object that is referenced in Kotlin code, for example, when passed as an argument, returned by a function, or retrieved from a collection. In this case, Kotlin creates its own object that holds the reference to the Objective-C object. When the Kotlin object gets deallocated, the Kotlin/Native runtime calls the `objc_release` function that releases that Objective-C reference.

Previously, the Kotlin/Native memory manager ran `objc_release` on a special GC thread. If it's the last object reference, the object gets deallocated. Issues could come up when Objective-C objects have custom deallocation hooks like the `dealloc` method in Objective-C or the `deinit` block in Swift, and these hooks expect to be called on a specific thread.

Since hooks for objects on the main thread usually expect to be called there, Kotlin/Native runtime now calls `objc_release` on the main thread as well. It should cover the cases when the Objective-C object was passed to Kotlin on the main thread, creating a Kotlin peer object there. This only works if

the main dispatch queue is processed, which is the case for regular UI applications. When it's not the main queue or the object was passed to Kotlin on a thread other than main, the `objc_release` is called on a special GC thread as before.

如何选择退出

In case you face issues, you can disable this behavior in your `gradle.properties` file with the following option:

```
kotlin.native.binary.objcDisposeOnMain=false
```

Don't hesitate to report such cases to our issue tracker.

在 Kotlin/Native 中访问常量值时不初始化对象

Starting with Kotlin 1.9.0, the Kotlin/Native backend doesn't initialize objects when accessing `const val` fields:

```
object MyObject {
    init {
        println("side effect!")
    }

    const val y = 1
}

fun main() {
    println(MyObject.y) // No initialization at first
    val x = MyObject    // Initialization occurs
    println(x.y)
}
```

The behavior is now unified with Kotlin/JVM, where the implementation is consistent with Java and objects are never initialized in this case. You can also expect some performance improvements in your Kotlin/Native projects thanks to this change.

在 Kotlin/Native 中能够为 iOS 模拟器测试配置独立模式

By default, when running iOS simulator tests for Kotlin/Native, the `--standalone` flag is used to avoid manual simulator booting and shutdown. In 1.9.0, you can now configure whether this flag is used in a Gradle task via the `standalone` property. By default, the `--standalone` flag is used so standalone mode is enabled.

Here is an example of how to disable standalone mode in your `build.gradle.kts` file:

```
tasks.withType<org.jetbrains.kotlin.gradle.targets.native.tasks.KotlinNativeSimulatorTest>().configureEach {
    standalone.set(false)
}
```

> If you disable standalone mode, you must boot the simulator manually. To boot your simulator from CLI, you can use the following command:
>
> ```
> /usr/bin/xcrun simctl boot <DeviceId>
> ```
>
> 

Kotlin/Native 的库链接

Starting with Kotlin 1.9.0, the Kotlin/Native compiler treats linkage issues in Kotlin libraries the same way as Kotlin/JVM. You might face such issues if the author of one third-party Kotlin library makes an incompatible change in experimental APIs that another third-party Kotlin library consumes.

Now builds don't fail during compilation in case of linkage issues between third-party Kotlin libraries. Instead, you'll only encounter these errors in run time, exactly as on the JVM.

The Kotlin/Native compiler reports warnings every time it detects issues with library linkage. You can find such warnings in your compilation logs, for example:

```
No function found for symbol 'org.samples/MyRemovedClass.doSomething|365763277190985851[0]'

Can not get instance of singleton 'MyEnumClass.REMOVED_ENTRY': No enum entry found for symbol 'org.samples/MyEnumClass.REMOVED_ENTRY|null[0]'

Function 'getMyRemovedClass' can not be called: Function uses unlinked class symbol 'org.samples/MyRemovedClass|null[0]'
```

You can further configure or even disable this behavior in your projects:

- If you don't want to see these warnings in your compilation logs, suppress them with the `-Xpartial-linkage-loglevel=INFO` compiler option.
- It's also possible to raise the severity of reported warnings to compilation errors with `-Xpartial-linkage-loglevel=ERROR`. In this case, the compilation fails and you'll see all the errors in the compilation log. Use this option to examine the linkage issues more closely.
- If you face unexpected problems with this feature, you can always opt out with the `-Xpartial-linkage=disable` compiler option. Don't hesitate to report such cases to our issue tracker.

```
// An example of passing compiler options via Gradle build file.
kotlin {
    macosX64("native") {
        binaries.executable()

        compilations.configureEach {
            compilerOptions.configure {
                // To suppress linkage warnings:
                freeCompilerArgs.add("-Xpartial-linkage-loglevel=INFO")

                // To raise linkage warnings to errors:
                freeCompilerArgs.add("-Xpartial-linkage-loglevel=ERROR")

                // To disable the feature completely:
                freeCompilerArgs.add("-Xpartial-linkage=disable")
            }
        }
    }
}
```

用于 C 语言互操作隐式整数转换的编译器选项

We have introduced a compiler option for C interop that allows you to use implicit integer conversions. After careful consideration, we've introduced this compiler option to prevent unintentional use as this feature still has room for improvement and our aim is to have an API of the highest quality.

In this code sample an implicit integer conversion allows `options = 0` even though `options` has unsigned type `UInt` and `0` is signed.

```
val today = NSDate()
val tomorrow = NSCalendar.currentCalendar.dateByAddingUnit(
    unit = NSCalendarUnitDay,
    value = 1,
    toDate = today,
    options = 0
)
```

To use implicit conversions with native interop libraries, use the `-XXLanguage:+ImplicitSignedToUnsignedIntegerConversion` compiler option.

You can configure this in your Gradle `build.gradle.kts` file:

```
tasks.withType<org.jetbrains.kotlin.gradle.tasks.KotlinNativeCompile>().configureEach
{
    compilerOptions.freeCompilerArgs.addAll(
        "-XXLanguage:+ImplicitSignedToUnsignedIntegerConversion"
```

```
    )
}
```

Kotlin 多平台

Kotlin Multiplatform has received some notable updates in 1.9.0 designed to improve your developer experience:

- Android 目标平台支持的变更
- 默认启用新版 Android 源代码集布局
- 多平台项目 Gradle 配置缓存预览版

Android 目标平台支持的变更

We continue our efforts to stabilize Kotlin Multiplatform. An essential step is to provide first-class support for the Android target. We're excited to announce that in the future, the Android team from Google will provide its own Gradle plugin to support Android in Kotlin Multiplatform.

To open the way for this new solution from Google, we're renaming the `android` block in the current Kotlin DSL in 1.9.0. Please change all the occurrences of the `android` block to `androidTarget` in your build scripts. This is a temporary change that is necessary to free the `android` name for the upcoming DSL from Google.

The Google plugin will be the preferred way of working with Android in multiplatform projects. When it's ready, we'll provide the necessary migration instructions so that you'll be able to use the short `android` name as before.

默认启用新版 Android 源代码集布局

Starting with Kotlin 1.9.0, the new Android source set layout is the default. It replaced the previous naming schema for directories, which was confusing in multiple ways. The new layout has a number of advantages:

- Simplified type semantics – The new Android source layout provides clear and consistent naming conventions that help to distinguish between different types of source sets.
- Improved source directory layout – With the new layout, the `SourceDirectories` arrangement becomes more coherent, making it easier to organize code and locate source files.
- Clear naming schema for Gradle configurations – The schema is now more consistent and predictable in both `KotlinSourceSets` and `AndroidSourceSets`.

The new layout requires the Android Gradle plugin version 7.0 or later and is supported in Android Studio 2022.3 and later. See our migration guide to make the necessary changes in your `build.gradle(.kts)` file.

Gradle 配置缓存预览版

Kotlin 1.9.0 comes with support for the Gradle configuration cache in multiplatform libraries. If you're a library author, you can already benefit from the improved build performance.

The Gradle configuration cache speeds up the build process by reusing the results of the configuration phase for subsequent builds. The feature has become Stable since Gradle 8.1. To enable it, follow the instructions in the Gradle documentation.

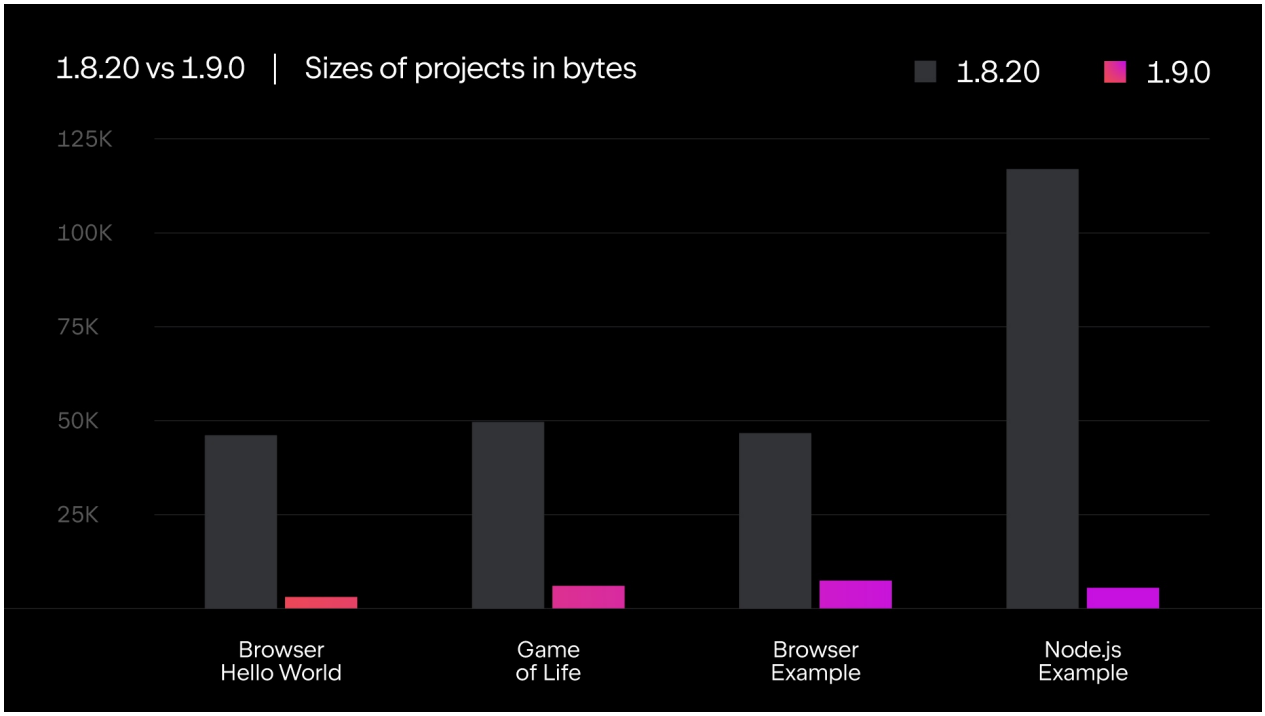
> The Kotlin Multiplatform plugin still doesn't support the Gradle configuration cache with Xcode integration tasks or the Kotlin CocoaPods Gradle plugin. We expect to add this feature in future Kotlin releases.
>
> 

Kotlin/Wasm

The Kotlin team continues to experiment with the new Kotlin/Wasm target. This release introduces several performance and size-related optimizations, along with updates in JavaScript interop.

尺寸相关的优化

Kotlin 1.9.0 introduces significant size improvements for WebAssembly (Wasm) projects. Comparing two "Hello World" projects, the code footprint for Wasm in Kotlin 1.9.0 is now over 10 times smaller than in Kotlin 1.8.20.



These size optimizations result in more efficient resource utilization and improved performance when targeting Wasm platforms with Kotlin code.

JavaScript 互操作更新

This Kotlin update introduces changes to the interoperability between Kotlin and JavaScript for Kotlin/Wasm. As Kotlin/Wasm is an Experimental feature, certain limitations apply to its interoperability.

动态类型的限制

Starting with version 1.9.0, Kotlin no longer supports the use of `Dynamic` types in Kotlin/Wasm. This is now deprecated in favor of the new universal `JsAny` type, which facilitates JavaScript interoperability.

For more details, see the Kotlin/Wasm interoperability with JavaScript documentation.

非外部类型的限制

Kotlin/Wasm supports conversions for specific Kotlin static types when passing values to and from JavaScript. These supported types include:

- Primitives, such as signed numbers, `Boolean`, and `Char`.
- `String`.
- Function types.

Other types were passed without conversion as opaque references, leading to inconsistencies between JavaScript and Kotlin subtyping.

To address this, Kotlin restricts JavaScript interop to a well-supported set of types. Starting from Kotlin 1.9.0, only external, primitive, string, and function types are supported in Kotlin/Wasm JavaScript interop. Furthermore, a separate explicit type called `JsReference` has been introduced to represent handles to Kotlin/Wasm objects that can be used in JavaScript interop.

For more details, refer to the Kotlin/Wasm interoperability with JavaScript documentation.

Kotlin Playground 中的 Kotlin/Wasm

Kotlin Playground supports the Kotlin/Wasm target. You can write, run, and share your Kotlin code that targets the Kotlin/Wasm. Check it out!

> Using Kotlin/Wasm requires enabling experimental features in your browser.
>
> Learn more about how to enable these features.
>
> 

```
import kotlin.time.*
import kotlin.time.measureTime

fun main() {
    println("Hello from Kotlin/Wasm!")
    computeAck(3, 10)
}

tailrec fun ack(m: Int, n: Int): Int = when {
    m == 0 -> n + 1
    n == 0 -> ack(m - 1, 1)
    else -> ack(m - 1, ack(m, n - 1))
}

fun computeAck(m: Int, n: Int) {
    var res = 0
    val t = measureTime {
        res = ack(m, n)
    }
    println()
    println("ack($m, $n) = ${res}")
    println("duration: ${t.inWholeNanoseconds / 1e6} ms")
}
```

Kotlin/JS

This release introduces updates for Kotlin/JS, including the removal of the old Kotlin/JS compiler, Kotlin/JS Gradle plugin deprecation and Experimental support for ES2015:

- 删除旧版 Kotlin/JS 编译器
- 弃用 Kotlin/JS Gradle 插件
- 弃用外部枚举
- 对 ES2015 类与模块的实验性支持
- 变更了 JS 生产分发的默认位置
- 从 stdlib-js 中提出 org.w3c 声明

> Starting from version 1.9.0, partial library linkage is also enabled for Kotlin/JS.
>
> 

删除旧版 Kotlin/JS 编译器

In Kotlin 1.8.0, we announced that the IR-based backend became Stable. Since then, not specifying the compiler has become an error, and using the old compiler leads to warnings.

In Kotlin 1.9.0, using the old backend results in an error. Please migrate to the IR compiler by following our migration guide.

弃用 Kotlin/JS Gradle 插件

Starting with Kotlin 1.9.0, the `kotlin-js` Gradle plugin is deprecated. We encourage you to use the `kotlin-multiplatform` Gradle plugin with the `js()` target instead.

The functionality of the Kotlin/JS Gradle plugin essentially duplicated the `kotlin-multiplatform` plugin and shared the same implementation under the hood. This overlap created confusion and increased maintenance load on the Kotlin team.

Refer to our Compatibility guide for Kotlin Multiplatform for migration instructions. If you find any issues that aren't covered in the guide, please report them to our issue tracker.

弃用外部枚举

In Kotlin 1.9.0, the use of external enums will be deprecated due to issues with static enum members like `entries`, that can't exist outside Kotlin. We recommend using an external sealed class with object subclasses instead:

```
// Before
external enum class ExternalEnum { A, B }

// After
external sealed class ExternalEnum {
```

```
    object A: ExternalEnum
    object B: ExternalEnum
}
```

By switching to an external sealed class with object subclasses, you can achieve similar functionality to external enums while avoiding the problems associated with default methods.

Starting from Kotlin 1.9.0, the use of external enums will be marked as deprecated. We encourage you to update your code to utilize the suggested external sealed class implementation for compatibility and future maintenance.

对 ES2015 类与模块的实验性支持

This release introduces Experimental support for ES2015 modules and generation of ES2015 classes:

- Modules offer a way to simplify your codebase and improve maintainability.
- Classes allow you to incorporate object-oriented programming (OOP) principles, resulting in cleaner and more intuitive code.

To enable these features, update your `build.gradle.kts` file accordingly:

```
// build.gradle.kts
kotlin {
    js(IR) {
        useEsModules() // Enables ES2015 modules
        browser()
    }
}

// Enables ES2015 classes generation
tasks.withType<KotlinJsCompile>().configureEach {
    kotlinOptions {
        useEsClasses = true
    }
}
```

Learn more about ES2015 (ECMAScript 2015, ES6) in the official documentation.

变更了 JS 生产分发的默认位置

Prior to Kotlin 1.9.0, the distribution target directory was `build/distributions`. However, this is a common directory for Gradle archives. To resolve this issue, we've changed the default distribution target directory in Kotlin 1.9.0 to: `build/dist/<targetName>/<binaryName>`.

For example, `productionExecutable` was in `build/distributions`. In Kotlin 1.9.0, it's in `build/dist/js/productionExecutable`.

> If you have a pipeline in place that uses the results of these builds, make sure to update the directory.
>
> 

从 stdlib-js 中提出 org.w3c 声明

Since Kotlin 1.9.0, the `stdlib-js` no longer includes `org.w3c` declarations. Instead, these declarations have been moved to a separate Gradle dependency. When you add the Kotlin Multiplatform Gradle plugin to your `build.gradle.kts` file, these declarations will be automatically included in your project, similar to the standard library.

There is no need for any manual action or migration. The necessary adjustments will be handled automatically.

Gradle

Kotlin 1.9.0 comes with new Gradle compiler options and a lot more:

- 删除了 classpath 属性
- 新版 Gradle 编译器选项
- 用于 Kotlin/JVM 的项目级编译器选项
- 用于 Kotlin/Native 模块名的编译器选项
- 官方 Kotlin 库的独立编译器插件
- 最低支持版本递增
- kapt 不会导致及早任务创建
- JVM 目标平台验证模式的编程化配置

删除了 classpath 属性

In Kotlin 1.7.0, we announced the start of a deprecation cycle for the `KotlinCompile` task's property: `classpath`. The deprecation level was raised to `ERROR` in Kotlin 1.8.0. In this release, we've finally removed the `classpath` property. All compile tasks should now use the `libraries` input for a list of libraries required for compilation.

新版编译器选项

The Kotlin Gradle plugin now provides new properties for opt-ins and the compiler's progressive mode.

- To opt in to new APIs, you can now use the `optIn` property and pass a list of strings like: `optIn.set(listOf(a, b, c))`.

- To enable progressive mode, use `progressiveMode.set(true)`.

用于 Kotlin/JVM 的项目级编译器选项

Starting with Kotlin 1.9.0, a new `compilerOptions` block is available inside the `kotlin` configuration block:

```
kotlin {
    compilerOptions {
        jvmTarget.set(JVM.Target_11)
    }
}
```

It makes configuring compiler options much easier. However, it is important to note some important details:

- This configuration only works on the project level.
- For the Android plugin, this block configures the same object as:

```
android {
    kotlinOptions {}
}
```

- The `android.kotlinOptions` and `kotlin.compilerOptions` configuration blocks override each other. The last (lowest) block in the build file always takes effect.
- If `moduleName` is configured on the project level, its value could be changed when passed to the compiler. It's not the case for the `main` compilation, but for other types, for example, test sources, the Kotlin Gradle plugin will add the `_test` suffix.
- The configuration inside the `tasks.withType<KotlinJvmCompile>().configureEach {}` (or `tasks.named<KotlinJvmCompile>("compileKotlin") { }`) overrides both `kotlin.compilerOptions` and `android.kotlinOptions`.

用于 Kotlin/Native 模块名的编译器选项

The Kotlin/Native `module-name` compiler option is now easily available in the Kotlin Gradle plugin.

This option specifies a name for the compilation module and can also be used for adding a name prefix for declarations exported to Objective-C.

You can now set the module name directly in the `compilerOptions` block of your Gradle build files:

> 【Kotlin】

```
tasks.named<org.jetbrains.kotlin.gradle.tasks.KotlinNativeCompile>("compileKotlinLinux
X64") {
    compilerOptions {
        moduleName.set("my-module-name")
    }
}
```

【Groovy】

```
tasks.named("compileKotlinLinuxX64", org.jetbrains.kotlin.gradle.tasks.KotlinNativeCom
pile.class) {
    compilerOptions {
        moduleName = "my-module-name"
    }
}
```

官方 Kotlin 库的独立编译器插件

Kotlin 1.9.0 introduces separate compiler plugins for its official libraries. Previously, compiler plugins were embedded into their corresponding Gradle plugins. This could cause compatibility issues in case the compiler plugin was compiled against a Kotlin version higher than the Gradle build's Kotlin runtime version.

Now compiler plugins are added as separate dependencies, so you'll no longer face compatibility issues with older Gradle versions. Another major advantage of the new approach is that new compiler plugins can be used with other build systems like Bazel.

Here's the list of new compiler plugins we're now publishing to Maven Central:

- kotlin-atomicfu-compiler-plugin
- kotlin-allopen-compiler-plugin
- kotlin-lombok-compiler-plugin
- kotlin-noarg-compiler-plugin
- kotlin-sam-with-receiver-compiler-plugin
- kotlinx-serialization-compiler-plugin

Every plugin has its `-embeddable` counterpart, for example, `kotlin-allopen-compiler-plugin-embeddable` is designed for working with the `kotlin-compiler-embeddable` artifact, the default option for scripting artifacts.

Gradle adds these plugins as compiler arguments. You don't need to make any changes to your existing projects.

最低支持版本递增

Starting with Kotlin 1.9.0, the minimum supported Android Gradle plugin version is 4.2.2.

See the Kotlin Gradle plugin's compatibility with available Gradle versions in our documentation.

kapt 不会在 Gradle 中及早任务创建

Prior to 1.9.0, the kapt compiler plugin caused eager task creation by requesting the configured instance of the Kotlin compilation task. This behavior has been fixed in Kotlin 1.9.0. If you use the default configuration for your `build.gradle.kts` file then your setup is not affected by this change.

> If you use a custom configuration, your setup will be adversely affected. For example, if you have modified the `KotlinJvmCompile` task using Gradle's tasks API, you must similarly modify the `KaptGenerateStubs` task in your build script.
>
> For example, if your script has the following configuration for the `KotlinJvmCompile` task:
>
> ```
> tasks.named<KotlinJvmCompile>("compileKotlin") { // Your custom configuration }
> ```
>
> In this case, you need to make sure that the same modification is included as part of the `KaptGenerateStubs` task:
>
> ```
> tasks.named<KaptGenerateStubs>("kaptGenerateStubs") { // Your custom configuratio
> n }
> ```
>
> 

For more information, see our YouTrack ticket.

JVM 目标平台验证模式的编程化配置

Before Kotlin 1.9.0, there was only one way to adjust the detection of JVM target incompatibility between Kotlin and Java. You had to set `kotlin.jvm.target.validation.mode=ERROR` in your `gradle.properties` for the whole project.

You can now also configure it on the task level in your `build.gradle.kts` file:

```
tasks.named<org.jetbrains.kotlin.gradle.tasks.KotlinJvmCompile>("compileKotlin") {
    jvmTargetValidationMode.set(org.jetbrains.kotlin.gradle.dsl.jvm.JvmTargetValidatio
nMode.WARNING)
```

```
}
```

标准库

Kotlin 1.9.0 has some great improvements for the standard library:

- `..<` 操作符与时间 API 已稳定。
- Kotlin/Native 标准库已彻底审阅并更新
- `@Volatile` 注解可以在更多平台上使用
- 有了按名称获取正则表达式捕获组的公共函数
- 引入了 `HexFormat` 类来格式化与解析十六进制

稳定版用于前闭后开区间的 ..< 操作符

The new `..<` operator for open-ended ranges that was introduced in Kotlin 1.7.20 and became Stable in 1.8.0. In 1.9.0, the standard library API for working with open-ended ranges is also Stable.

Our research shows that the new `..<` operator makes it easier to understand when an open-ended range is declared. If you use the `until` infix function, it's easy to make the mistake of assuming that the upper bound is included.

Here is an example using the `until` function:

```
fun main() {
    for (number in 2 until 10) {
        if (number % 2 == 0) {
            print("$number ")
        }
    }
    // 2 4 6 8
}
```

And here is an example using the new `..<` operator:

```
fun main() {
    for (number in 2..<10) {
        if (number % 2 == 0) {
            print("$number ")
        }
    }
    // 2 4 6 8
}
```

> From IntelliJ IDEA version 2023.1.1, a new code inspection is available that highlights when you can use the `..<` operator.
>
> 

For more information about what you can do with this operator, see What's new in Kotlin 1.7.20.

稳定版时间 API

Since 1.3.50, we have previewed a new time measurement API. The duration part of the API became Stable in 1.6.0. In 1.9.0, the remaining time measurement API is Stable.

The old time API provided the `measureTimeMillis` and `measureNanoTime` functions, which aren't intuitive to use. Although it is clear that they both measure time in different units, it isn't clear that `measureTimeMillis` uses a wall clock to measure time, whereas `measureNanoTime` uses a monotonic time source. The new time API resolves this and other issues to make the API more user friendly.

With the new time API, you can easily:

- Measure the time taken to execute some code using a monotonic time source with your desired time unit.
- Mark a moment in time.
- Compare and find the difference between two moments in time.
- Check how much time has passed since a specific moment in time.
- Check whether the current time has passed a specific moment in time.

测量代码执行时间

To measure the time taken to execute a block of code, use the `measureTime` inline function.

To measure the time taken to execute a block of code **and** return the result of the block of code, use the `measureTimedValue` inline function.

By default, both functions use a monotonic time source. However, if you want to use an elapsed real-time source, you can. For example, on Android the default time source `System.nanoTime()` only counts time while the device is active. It loses track of time when the device enters deep sleep. To keep track of time while the device is in deep sleep, you can create a time source that uses `SystemClock.elapsedRealtimeNanos()`) instead:

```
object RealtimeMonotonicTimeSource : AbstractLongTimeSource(DurationUnit.NANOSECONDS)
{
    override fun read(): Long = SystemClock.elapsedRealtimeNanos()
}
```

标记并测量时间上的差异

To mark a specific moment in time, use the `TimeSource` interface and the `markNow()` function to create a `TimeMark`. To measure differences between `TimeMarks` from the same time source, use the subtraction operator (`-`):

```
import kotlin.time.*

fun main() {
    val timeSource = TimeSource.Monotonic
    val mark1 = timeSource.markNow()
    Thread.sleep(500) // Sleep 0.5 seconds.
    val mark2 = timeSource.markNow()

    repeat(4) { n ->
        val mark3 = timeSource.markNow()
        val elapsed1 = mark3 - mark1
        val elapsed2 = mark3 - mark2

        println("Measurement 1.${n + 1}: elapsed1=$elapsed1, elapsed2=$elapsed2, diff=
${elapsed1 - elapsed2}")
    }
    // It's also possible to compare time marks with each other.
    println(mark2 > mark1) // This is true, as mark2 was captured later than mark1.
}
```

To check if a deadline has passed or a timeout has been reached, use the `hasPassedNow()` and `hasNotPassedNow()` extension functions:

```
import kotlin.time.*
import kotlin.time.Duration.Companion.seconds

fun main() {
    val timeSource = TimeSource.Monotonic
    val mark1 = timeSource.markNow()
    val fiveSeconds: Duration = 5.seconds
    val mark2 = mark1 + fiveSeconds

    // It hasn't been 5 seconds yet
    println(mark2.hasPassedNow())
    // false

    // Wait six seconds
    Thread.sleep(6000)
    println(mark2.hasPassedNow())
    // true
}
```

Kotlin/Native 标准库的稳定之旅

As our standard library for Kotlin/Native continues to grow, we decided that it was time for a complete review to ensure that it meets our high standards. As part of this, we carefully reviewed **every** existing public signature. For each signature, we considered whether it:

- Has a unique purpose.
- Is consistent with other Kotlin APIs.
- Has similar behavior to its counterpart for the JVM.
- Is future-proof.

Based on these considerations, we made one of the following decisions:

- Made it Stable.
- Made it Experimental.
- Marked it as `private`.
- Modified its behavior.
- Moved it to a different location.
- Deprecated it.
- Marked it as obsolete.

> If an existing signature has been:
>
> - Moved to another package, then the signature still exists in the original package but it's now deprecated with deprecation level: `WARNING`. IntelliJ IDEA will automatically suggest replacements upon code inspection.
> - Deprecated, then it's been deprecated with deprecation level: `WARNING`.
> - Marked as obsolete, then you can keep using it, but it will be replaced in future.
>
> 

We won't list all of the results of the review here, but here are some of the highlights:

- We stabilized the Atomics API.
- We made `kotlinx.cinterop` Experimental and now require different opt-ins for the package to be used. For more information, see Explicit C-interoperability stability guarantees.
- We marked the `Worker` class and its related APIs as obsolete.
- We marked the `BitSet` class as obsolete.
- We marked all `public` APIs in the `kotlin.native.internal` package as `private` or moved them to other packages.

显式 C 语言互操作性的稳定性保证

To maintain the high quality of our API, we decided to make `kotlinx.cinterop` Experimental. Although `kotlinx.cinterop` has been thoroughly tried and tested, there is still room for improvement before we are satisfied enough to make it Stable. We recommend that you use this API for interoperability but that you try to confine its use to specific areas in your projects. This will make your migration easier once we begin evolving this API to make it Stable.

If you want to use C-like foreign APIs such as pointers, you must opt in with `@OptIn(ExperimentalForeignApi)`, otherwise your code won't compile.

To use the remainder of `kotlinx.cinterop`, which covers Objective-C/Swift interoperability, you must opt in with `@OptIn(BetaInteropApi)`. If you try to use this API without the opt-in, your code will compile but the compiler will raise warnings that provide a clear explanation of what behavior you can expect.

For more information about these annotations, see our source code for `Annotations.kt`.

For more information on **all** of the changes as part of this review, see our YouTrack ticket.

We'd appreciate any feedback you might have! You can provide your feedback directly by commenting on the ticket.

稳定版 @Volatile 注解

If you annotate a `var` property with `@Volatile`, then the backing field is marked so that any reads or writes to this field are atomic, and writes are always made visible to other threads.

Prior to 1.8.20, the `kotlin.jvm.Volatile` annotation was available in the common standard library. However, this annotation was only effective on the JVM. If you used it on other platforms, it was ignored, which led to errors.

In 1.8.20, we introduced an experimental common annotation, `kotlin.concurrent.Volatile`, which you could preview in both the JVM and Kotlin/Native.

In 1.9.0, `kotlin.concurrent.Volatile` is Stable. If you use `kotlin.jvm.Volatile` in your multiplatform projects, we recommend that you migrate to `kotlin.concurrent.Volatile`.

新增按名称获取正则表达式捕获组的公共函数

Prior to 1.9.0, every platform had its own extension to get a regular expression capture group by its name from a regular expression match. However there was no common function. It wasn't possible to have a common function prior to Kotlin 1.8.0, because the standard library still supported JVM targets 1.6 and 1.7.

As of Kotlin 1.8.0, the standard library is compiled with JVM target 1.8. So in 1.9.0, there is now a **common** `groups` function that you can use to retrieve a group's contents by its name for a regular expression match. This is useful when you want to access the results of regular expression matches belonging to a particular capture group.

Here is an example with a regular expression containing three capture groups: `city`, `state`, and `areaCode`. You can use these group names to access the matched values:

```
fun main() {
    val regex = """\b(?<city>[A-Za-z\s]+),\s(?<state>[A-Z]{2}):\s(?<areaCode>[0-9]{3})
\b""".toRegex()
    val input = "Coordinates: Austin, TX: 123"

    val match = regex.find(input)!!
    println(match.groups["city"]?.value)
    // Austin
    println(match.groups["state"]?.value)
    // TX
    println(match.groups["areaCode"]?.value)
    // 123
}
```

新增用于创建父目录路径的实用程序

In 1.9.0 there is a new `createParentDirectories()` extension function that you can use to create a new file with all the necessary parent directories. When you provide a file path to `createParentDirectories()` it checks whether the parent directories already exist. If they do, it does nothing. However, if they do not, it creates them for you.

`createParentDirectories()` is particularly useful when you are copying files. For example, you can use it in combination with the `copyToRecursively()` function:

```
sourcePath.copyToRecursively(
    destinationPath.createParentDirectories(),
    followLinks = false
 )
```

新增 HexFormat 类用于格式化与解析十六进制

> The new `HexFormat` class and its related extension functions are Experimental, and to use them, you can opt in with `@OptIn(ExperimentalStdlibApi::class)` or the compiler argument `-opt-in=kotlin.ExperimentalStdlibApi`.
>
> 

In 1.9.0, the `HexFormat` class and its related extension functions are provided as an Experimental feature that allows you to convert between numerical values and hexadecimal strings. Specifically, you can use the extension functions to convert between hexadecimal strings and `ByteArrays` or other numeric types (`Int` , `Short` , `Long`).

For example:

```
println(93.toHexString()) // "0000005d"
```

The `HexFormat` class includes formatting options that you can configure with the `HexFormat{}` builder.

If you are working with `ByteArrays` you have the following options, which are configurable by properties:

Option	Description
`upperCase`	Whether hexadecimal digits are upper or lower case. By default, lower case is assumed. `upperCase = false` .
`bytes.bytesPerLine`	The maximum number of bytes per line.
`bytes.bytesPerGroup`	The maximum number of bytes per group.
`bytes.bytesSeparator`	The separator between bytes. Nothing by default.
`bytes.bytesPrefix`	The string that immediately precedes a two-digit hexadecimal representation of each byte, nothing by default.
`bytes.bytesSuffix`	The string that immediately succeeds a two-digit hexadecimal representation of each byte, nothing by default.

For example:

```
val macAddress = "001b638445e6".hexToByteArray()

// Use HexFormat{} builder to separate the hexadecimal string by colons
println(macAddress.toHexString(HexFormat { bytes.byteSeparator = ":" }))
// "00:1b:63:84:45:e6"

// Use HexFormat{} builder to:
// * Make the hexadecimal string uppercase
// * Group the bytes in pairs
// * Separate by periods
val threeGroupFormat = HexFormat { upperCase = true; bytes.bytesPerGroup = 2; bytes.gr
oupSeparator = "." }

println(macAddress.toHexString(threeGroupFormat))
// "001B.6384.45E6"
```

If you are working with numeric types, you have the following options, which are configurable by properties:

Option	Description
`number.prefix`	The prefix of a hexadecimal string, nothing by default.
`number.suffix`	The suffix of a hexadecimal string, nothing by default.
`number.removeLeadingZeros`	Whether to remove leading zeros in a hexadecimal string. By default, no leading zeros are removed. `number.removeLeadingZeros = false`

For example:

```
// Use HexFormat{} builder to parse a hexadecimal that has prefix: "0x".
println("0x3a".hexToInt(HexFormat { number.prefix = "0x" })) // "58"
```

文档更新

The Kotlin documentation has received some notable changes:

- The tour of Kotlin – Learn the fundamentals of the Kotlin programming language with chapters including both theory and practice.
- Android source set layout – Learn about the new Android source set layout.
- Compatibility guide for Kotlin Multiplatform – Learn about the incompatible changes you might encounter while developing projects with Kotlin Multiplatform.
- Kotlin Wasm – Learn about Kotlin/Wasm and how you can use it in your Kotlin Multiplatform projects.

安装 Kotlin 1.9.0

检查 IDE 版本

IntelliJ IDEA 2022.3.3 and 2023.1.1 automatically suggest updating the Kotlin plugin to version 1.9.0. IntelliJ IDEA 2023.2 will include the Kotlin 1.9.0 plugin.

Android Studio Giraffe (223) and Hedgehog (231) will support Kotlin 1.9.0 in their upcoming releases.

The new command-line compiler is available for download on the GitHub release page.

配置 Gradle 设置

To download Kotlin artifacts and dependencies, update your `settings.gradle(.kts)` file to use the Maven Central repository:

```
pluginManagement {
    repositories {
        mavenCentral()
        gradlePluginPortal()
    }
}
```

If the repository is not specified, Gradle uses the sunset JCenter repository, which could lead to issues with Kotlin artifacts.

Kotlin 1.9.0 的兼容性指南

Kotlin 1.9.0 is a feature release and can, therefore, bring changes that are incompatible with your code written for earlier versions of the language. Find the detailed list of these changes in the Compatibility guide for Kotlin 1.9.0.

Kotlin 1.8.20 的新特性

发布于：*2025-05-13*

The Kotlin 1.8.20 release is out and here are some of its biggest highlights:

- 新版 Kotlin K2 编译器更新
- 新增实验性 Kotlin/Wasm 目标平台
- Gradle 中默认新版 JVM 增量编译
- Kotlin/Native 目标更新
- Kotlin 多平台 Gradle 复合构建预览版
- 改进了 Xcode 中 Gradle 错误的输出
- 标准库中对 AutoCloseable 接口的实验性支持
- 标准库中对 Base64 编码的实验性支持

You can also find a short overview of the changes in this video:

YouTube 视频：What's new in Kotlin 1.8.20

IDE 支持

The Kotlin plugins that support 1.8.20 are available for:

IDE	Supported versions
IntelliJ IDEA	2022.2.x, 2022.3.x, 2023.1.x
Android Studio	Flamingo (222)

> To download Kotlin artifacts and dependencies properly, configure Gradle settings to use the Maven Central repository.
>
> 

新版 Kotlin K2 编译器更新

The Kotlin team continues to stabilize the K2 compiler. As mentioned in the Kotlin 1.7.0 announcement, it's still in **Alpha**. This release introduces further improvements on the road to K2 Beta.

Starting with this 1.8.20 release, the Kotlin K2 compiler:

- Has a preview version of the serialization plugin.
- Provides Alpha support for the JS IR compiler.
- Introduces the future release of the new language version, Kotlin 2.0.

Learn more about the new compiler and its benefits in the following videos:

- What Everyone Must Know About The NEW Kotlin K2 Compiler
- The New Kotlin K2 Compiler: Expert Review

如何启用 Kotlin K2 编译器

To enable and test the Kotlin K2 compiler, use the new language version with the following compiler option:

```
-language-version 2.0
```

You can specify it in your `build.gradle(.kts)` file:

```
kotlin {
   sourceSets.all {
       languageSettings {
           languageVersion = "2.0"
       }
   }
}
```

The previous `-Xuse-k2` compiler option has been deprecated.

> The Alpha version of the new K2 compiler only works with JVM and JS IR projects. It doesn't support Kotlin/Native or any of the multiplatform projects yet.
>
> 

反馈你对新版 K2 编译器的意见

We would appreciate any feedback you may have!

- Provide your feedback directly to K2 developers on Kotlin Slack – get an invite and join the #k2-early-adopters channel.
- Report any problems you faced with the new K2 compiler on our issue tracker.
- Enable the **Send usage statistics** option to allow JetBrains to collect anonymous data about K2 usage.

语言

As Kotlin continues to evolve, we're introducing preview versions for new language features in 1.8.20:

- 枚举类类 values 函数的现代且高性能替代
- 与数据类相称的数据对象
- 取消对内联类体中有次构造函数限制

枚举类类 values 函数的现代且高性能替代

> This feature is Experimental. It may be dropped or changed at any time. Opt-in is required (see details below). Use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack.
>
> 

Enum classes have a synthetic `values()` function, which returns an array of defined enum constants. However, using an array can lead to hidden performance issues in Kotlin and Java. In addition, most of the APIs use collections, which require eventual conversion. To fix these problems, we've introduced the `entries` property for Enum classes, which should be used instead of the `values()` function. When called, the `entries` property returns a pre-allocated immutable list of defined enum constants.

> The `values()` function is still supported, but we recommend that you use the `entries` property instead.
>
> 

```
enum class Color(val colorName: String, val rgb: String) {
    RED("Red", "#FF0000"),
    ORANGE("Orange", "#FF7F00"),
    YELLOW("Yellow", "#FFFF00")
}

@OptIn(ExperimentalStdlibApi::class)
fun findByRgb(rgb: String): Color? = Color.entries.find { it.rgb == rgb }
```

如何启用 entries 属性

To try this feature out, opt in with `@OptIn(ExperimentalStdlibApi)` and enable the `-language-version 1.9` compiler option. In a Gradle project, you can do so by adding the following to your `build.gradle(.kts)` file:

> 【Kotlin】

```
tasks
    .withType<org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTask<*>>()
    .configureEach {
        compilerOptions
            .languageVersion
            .set(
                org.jetbrains.kotlin.gradle.dsl.KotlinVersion.KOTLIN_1_9
            )
    }
```

> 【Groovy】

```
tasks
    .withType(org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTask.class)
    .configureEach {
        compilerOptions.languageVersion =
            org.jetbrains.kotlin.gradle.dsl.KotlinVersion.KOTLIN_1_9
    }
```

> Starting with IntelliJ IDEA 2023.1, if you have opted in to this feature, the appropriate IDE inspection will notify you about converting from `values()` to `entries` and offer a quick-fix.
>
> 

For more information on the proposal, see the KEEP note.

与数据类相称的数据对象预览版

Data objects allow you to declare objects with singleton semantics and a clean `toString()` representation. In this snippet, you can see how adding the `data` keyword to an object declaration improves the readability of its `toString()` output:

```
package org.example
object MyObject
data object MyDataObject

fun main() {
    println(MyObject) // org.example.MyObject@1f32e575
    println(MyDataObject) // MyDataObject
}
```

Especially for `sealed` hierarchies (like a `sealed class` or `sealed interface` hierarchy), `data objects` are an excellent fit because they can be used conveniently alongside `data class` declarations. In this snippet, declaring `EndOfFile` as a `data object` instead of a plain `object` means that it will get a pretty `toString` without the need to override it manually. This maintains symmetry with the accompanying data class definitions.

```
sealed interface ReadResult
data class Number(val number: Int) : ReadResult
data class Text(val text: String) : ReadResult
data object EndOfFile : ReadResult

fun main() {
    println(Number(7)) // Number(number=7)
    println(EndOfFile) // EndOfFile
}
```

数据对象的语义

Since their first preview version in Kotlin 1.7.20, the semantics of data objects have been refined. The compiler now automatically generates a number of convenience functions for them:

toString

The `toString()` function of a data object returns the simple name of the object:

```
data object MyDataObject {
    val x: Int = 3
}

fun main() {
    println(MyDataObject) // MyDataObject
}
```

equals 与 hashCode

The `equals()` function for a `data object` ensures that all objects that have the type of your `data object` are considered equal. In most cases, you will only have a single instance of your data object at runtime (after all, a `data object` declares a singleton). However, in the edge case where another object of the same type is generated at runtime (for example, via platform reflection through `java.lang.reflect`, or by using a JVM serialization library that uses this API under the hood), this ensures that the objects are treated as equal.

Make sure to only compare `data objects` structurally (using the `==` operator) and never by reference (the `===` operator). This helps avoid pitfalls when more than one instance of a data object exists at runtime. The following snippet illustrates this specific edge case:

```
import java.lang.reflect.Constructor

data object MySingleton

fun main() {
    val evilTwin = createInstanceViaReflection()

    println(MySingleton) // MySingleton
    println(evilTwin) // MySingleton

    // Even when a library forcefully creates a second instance of MySingleton, its `e
quals` method returns true:
    println(MySingleton == evilTwin) // true

    // Do not compare data objects via ===.
    println(MySingleton === evilTwin) // false
}

fun createInstanceViaReflection(): MySingleton {
    // Kotlin reflection does not permit the instantiation of data objects.
    // This creates a new MySingleton instance "by force" (i.e., Java platform reflect
ion)
    // Don't do this yourself!
    return (MySingleton.javaClass.declaredConstructors[0].apply { isAccessible = true
} as Constructor<MySingleton>).newInstance()
}
```

The behavior of the generated `hashCode()` function is consistent with that of the `equals()` function, so that all runtime instances of a `data object` have the same hash code.

数据对象没有 copy 与 componentN 函数

While `data object` and `data class` declarations are often used together and have some similarities, there are some functions that are not generated for a `data object`:

Because a `data object` declaration is intended to be used as a singleton object, no `copy()` function is generated. The singleton pattern restricts the instantiation of a class to a single instance, and allowing copies of the instance to be created would violate that restriction.

Also, unlike a `data class`, a `data object` does not have any data properties. Since attempting to destructure such an object would not make sense, no `componentN()` functions are generated.

We would appreciate your feedback on this feature in YouTrack.

如何启用数据对象预览版

To try this feature out, enable the `-language-version 1.9` compiler option. In a Gradle project, you can do so by adding the following to your `build.gradle(.kts)` file:

> 【Kotlin】

```
tasks
    .withType<org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTask<*>>()
    .configureEach {
        compilerOptions
            .languageVersion
            .set(
                org.jetbrains.kotlin.gradle.dsl.KotlinVersion.KOTLIN_1_9
            )
    }
```

> 【Groovy】

```
tasks
    .withType(org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTask.class)
    .configureEach {
        compilerOptions.languageVersion =
            org.jetbrains.kotlin.gradle.dsl.KotlinVersion.KOTLIN_1_9
    }
```

取消对内联类中有函数体的次构造函数限制的预览版

> This feature is Experimental. It may be dropped or changed at any time. Opt-in is required (see details below). Use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack.



Kotlin 1.8.20 lifts restrictions on the use of secondary constructors with bodies in inline classes.

Inline classes used to allow only a public primary constructor without `init` blocks or secondary constructors to have clear initialization semantics. As a result, it was impossible to encapsulate underlying values or create an inline class that would represent some constrained values.

These issues were fixed when Kotlin 1.4.30 lifted restrictions on `init` blocks. Now we're taking it a step further and allowing secondary constructors with bodies in preview mode:

```
@JvmInline
value class Person(private val fullName: String) {
    // Allowed since Kotlin 1.4.30:
    init {
        check(fullName.isNotBlank()) {
            "Full name shouldn't be empty"
        }
    }

    // Preview available since Kotlin 1.8.20:
    constructor(name: String, lastName: String) : this("$name $lastName") {
        check(lastName.isNotBlank()) {
            "Last name shouldn't be empty"
        }
    }
}
```

如何启用带函数体的次构造函数

To try this feature out, enable the `-language-version 1.9` compiler option. In a Gradle project, you can do so by adding the following to your `build.gradle(.kts)`:

【Kotlin】

```
tasks
    .withType<org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTask<*>>()
    .configureEach {
        compilerOptions
            .languageVersion
            .set(
                org.jetbrains.kotlin.gradle.dsl.KotlinVersion.KOTLIN_1_9
            )
    }
```

【Groovy】

```
tasks
    .withType(org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTask.class)
    .configureEach {
        compilerOptions.languageVersion =
            org.jetbrains.kotlin.gradle.dsl.KotlinVersion.KOTLIN_1_9
    }
```

We encourage you to try this feature out and submit all reports in YouTrack to help us make it the default in Kotlin 1.9.0.

Learn more about the development of Kotlin inline classes in this KEEP.

新增 Kotlin/Wasm 目标平台

Kotlin/Wasm (Kotlin WebAssembly) goes Experimental in this release. The Kotlin team finds WebAssembly to be a promising technology and wants to find better ways for you to use it and get all of the benefits of Kotlin.

WebAssembly binary format is independent of the platform because it runs using its own virtual machine. Almost all modern browsers already support WebAssembly 1.0. To set up the environment to run WebAssembly, you only need to enable an experimental garbage collection mode that Kotlin/Wasm targets. You can find detailed instructions here: How to enable Kotlin/Wasm.

We want to highlight the following advantages of the new Kotlin/Wasm target:

- Faster compilation speed compared to the `wasm32` Kotlin/Native target, since Kotlin/Wasm doesn't have to use LLVM.
- Easier interoperability with JS and integration with browsers compared to the `wasm32` target, thanks to the Wasm garbage collection.
- Potentially faster application startup compared to Kotlin/JS and JavaScript because Wasm has a compact and easy-to-parse bytecode.
- Improved application runtime performance compared to Kotlin/JS and JavaScript because Wasm is a statically typed language.

Starting with the 1.8.20 release, you can use Kotlin/Wasm in your experimental projects. We provide the Kotlin standard library (`stdlib`) and test library (`kotlin.test`) for Kotlin/Wasm out of the box. IDE support will be added in future releases.

Learn more about Kotlin/Wasm in this YouTube video.

如何启用 Kotlin/Wasm

To enable and test Kotlin/Wasm, update your `build.gradle.kts` file:

```
plugins {
    kotlin("multiplatform") version "1.8.20"
}

kotlin {
    wasm {
        binaries.executable()
        browser {
        }
    }
    sourceSets {
        val commonMain by getting
        val commonTest by getting {
            dependencies {
                implementation(kotlin("test"))
            }
        }
        val wasmMain by getting
        val wasmTest by getting
    }
}
```

> Check out the GitHub repository with Kotlin/Wasm examples.
>
> 

To run a Kotlin/Wasm project, you need to update the settings of the target environment:

> **【Chrome】**

- For version 109:

  Run the application with the `--js-flags=--experimental-wasm-gc` command line argument.

- For version 110 or later:

  1. Go to `chrome://flags/#enable-webassembly-garbage-collection` in your browser.
  2. Enable **WebAssembly Garbage Collection**.
  3. Relaunch your browser.

> **【Firefox】**

For version 109 or later:

1. Go to `about:config` in your browser.
2. Enable `javascript.options.wasm_function_references` and `javascript.options.wasm_gc` options.
3. Relaunch your browser.

> 【Edge】

For version 109 or later:

Run the application with the `--js-flags=--experimental-wasm-gc` command line argument.

反馈你对 Kotlin/Wasm 的意见

We would appreciate any feedback you may have!

- Provide your feedback directly to developers in Kotlin Slack – get an invite and join the #webassembly channel.
- Report any problems you faced with Kotlin/Wasm on this YouTrack issue.

Kotlin/JVM

Kotlin 1.8.20 introduces a preview of Java synthetic property references and support for the JVM IR backend in the kapt stub generating task by default.

Java 合成属性引用预览版

> This feature is Experimental. It may be dropped or changed at any time. Use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack.
>
> 

Kotlin 1.8.20 introduces the ability to create references to Java synthetic properties, for example, for such Java code:

```
public class Person {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
```

```
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }
}
```

Kotlin has always allowed you to write `person.age`, where `age` is a synthetic property. Now, you can also create references to `Person::age` and `person::age`. All the same works for `name`, as well.

```
val persons = listOf(Person("Jack", 11), Person("Sofie", 12), Person("Peter", 11))
    persons
        // Call a reference to Java synthetic property:
        .sortedBy(Person::age)
        // Call Java getter via the Kotlin property syntax:
        .forEach { person -> println(person.name) }
```

如何启用 Java 合成属性引用

To try this feature out, enable the `-language-version 1.9` compiler option. In a Gradle project, you can do so by adding the following to your `build.gradle(.kts)`:

> 【Kotlin】

```
tasks
    .withType<org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTask<*>>()
    .configureEach {
        compilerOptions
            .languageVersion
            .set(
                org.jetbrains.kotlin.gradle.dsl.KotlinVersion.KOTLIN_1_9
            )
    }
```

> 【Groovy】

```
tasks
```

```
    .withType(org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTask.class)
    .configureEach {
        compilerOptions.languageVersion =
            org.jetbrains.kotlin.gradle.dsl.KotlinVersion.KOTLIN_1_9
    }
```

在 kapt 存根生成任务中支持 JVM IR 后端

In Kotlin 1.7.20, we introduced support for the JVM IR backend in the kapt stub generating task. Starting with this release, this support works by default. You no longer need to specify `kapt.use.jvm.ir=true` in your `gradle.properties` to enable it. We would appreciate your feedback on this feature in YouTrack.

Kotlin/Native

Kotlin 1.8.20 includes changes to supported Kotlin/Native targets, interoperability with Objective-C, and improvements to the CocoaPods Gradle plugin, among other updates:

- Kotlin/Native 目标更新
- 弃用旧版内存管理器
- 使用 @import 指令支持 Objective-C 头文件
- Cocoapods Gradle 插件支持仅链接模式
- 将 Objective-C 扩展作为 UIKit 的类成员导入
- 重新实现编译器中的编译器缓存管理
- Cocoapods Gradle 插件弃用 `useLibraries()`

Kotlin/Native 目标更新

The Kotlin team decided to revisit the list of targets supported by Kotlin/Native, split them into tiers, and deprecate some of them starting with Kotlin 1.8.20. See the Kotlin/Native target support section for the full list of supported and deprecated targets.

The following targets have been deprecated with Kotlin 1.8.20 and will be removed in 1.9.20:

- `iosArm32`
- `watchosX86`
- `wasm32`
- `mingwX86`
- `linuxArm32Hfp`
- `linuxMips32`
- `linuxMipsel32`

As for the remaining targets, there are now three tiers of support depending on how well a target is supported and tested in the Kotlin/Native compiler. A target can be moved to a different tier. For example, we'll do our best to provide full support for `iosArm64` in the future, as it is important for Kotlin Multiplatform.

If you're a library author, these target tiers can help you decide which targets to test on CI tools and which ones to skip. The Kotlin team will use the same approach when developing official Kotlin libraries, like kotlinx.coroutines.

Check out our blog post to learn more about the reasons for these changes.

弃用旧版内存管理器

Starting with 1.8.20, the legacy memory manager is deprecated and will be removed in 1.9.20. The new memory manager was enabled by default in 1.7.20 and has been receiving further stability updates and performance improvements.

If you're still using the legacy memory manager, remove the `kotlin.native.binary.memoryModel=strict` option from your `gradle.properties` and follow our Migration guide to make the necessary changes.

The new memory manager doesn't support the `wasm32` target. This target is also deprecated starting with this release and will be removed in 1.9.20.

使用 @import 指令支持 Objective-C 头文件

> This feature is Experimental. It may be dropped or changed at any time. Opt-in is required (see details below). Use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack.
>
> 

Kotlin/Native can now import Objective-C headers with `@import` directives. This feature is useful for consuming Swift libraries that have auto-generated Objective-C headers or classes of CocoaPods dependencies written in Swift.

Previously, the cinterop tool failed to analyze headers that depended on Objective-C modules via the `@import` directive. The reason was that it lacked support for the `-fmodules` option.

Starting with Kotlin 1.8.20, you can use Objective-C headers with `@import`. To do so, pass the `-fmodules` option to the compiler in the definition file as `compilerOpts`. If you use CocoaPods integration, specify the cinterop option in the configuration block of the `pod()` function like this:

```
kotlin {
    ios()
```

```
    cocoapods {
        summary = "CocoaPods test library"
        homepage = "https://github.com/JetBrains/kotlin"

        ios.deploymentTarget = "13.5"

        pod("PodName") {
            extraOpts = listOf("-compiler-option", "-fmodules")
        }
    }
}
```

This was a highly awaited feature, and we welcome your feedback about it in YouTrack to help us make it the default in future releases.

Cocoapods Gradle 插件支持仅链接模式

With Kotlin 1.8.20, you can use Pod dependencies with dynamic frameworks only for linking, without generating cinterop bindings. This may come in handy when cinterop bindings are already generated.

Consider a project with 2 modules, a library and an app. The library depends on a Pod but doesn't produce a framework, only a `.klib`. The app depends on the library and produces a dynamic framework. In this case, you need to link this framework with the Pods that the library depends on, but you don't need cinterop bindings because they are already generated for the library.

To enable the feature, use the `linkOnly` option or a builder property when adding a dependency on a Pod:

```
cocoapods {
    summary = "CocoaPods test library"
    homepage = "https://github.com/JetBrains/kotlin"

    pod("Alamofire", linkOnly = true) {
        version = "5.7.0"
    }
}
```

> If you use this option with static frameworks, it will remove the Pod dependency entirely because Pods are not used for static framework linking.
>
> 

将 Objective-C 扩展作为 UIKit 的类成员导入

Since Xcode 14.1, some methods from Objective-C classes have been moved to category members. That led to the generation of a different Kotlin API, and these methods were imported as Kotlin extensions instead of methods.

You may have experienced issues resulting from this when overriding methods using UIKit. For example, it became impossible to override `drawRect()` or `layoutSubviews()` methods when subclassing a UIVIew in Kotlin.

Since 1.8.20, category members that are declared in the same headers as NSView and UIView classes are imported as members of these classes. This means that the methods subclassing from NSView and UIView can be easily overridden, like any other method.

If everything goes well, we're planning to enable this behavior by default for all of the Objective-C classes.

重新实现编译器中的编译器缓存管理

To speed up the evolution of compiler caches, we've moved compiler cache management from the Kotlin Gradle plugin to the Kotlin/Native compiler. This unblocks work on several important improvements, including those to do with compilation times and compiler cache flexibility.

If you encounter some problem and need to return to the old behavior, use the `kotlin.native.cacheOrchestration=gradle` Gradle property.

We would appreciate your feedback on this in YouTrack.

Cocoapods Gradle 插件弃用 useLibraries()

Kotlin 1.8.20 starts the deprecation cycle of the `useLibraries()` function used in the CocoaPods integration for static libraries.

We introduced the `useLibraries()` function to allow dependencies on Pods containing static libraries. With time, this case has become very rare. Most of the Pods are distributed by sources, and Objective-C frameworks or XCFrameworks are a common choice for binary distribution.

Since this function is unpopular and it creates issues that complicate the development of the Kotlin CocoaPods Gradle plugin, we've decided to deprecate it.

For more information on frameworks and XCFrameworks, see Build final native binaries.

Kotlin 多平台

Kotlin 1.8.20 strives to improve the developer experience with the following updates to Kotlin Multiplatform:

- 设置源代码集分层结构的新方式
- Kotlin 多平台支持 Gradle 复合构建预览版
- 改进了 Xcode 中 Gradle 错误的输出

源代码集分层结构的新方式

> The new approach to source set hierarchy is Experimental. It may be changed in future Kotlin releases without prior notice. Opt-in is required (see the details below). We would appreciate your feedback in YouTrack.
>
> 

Kotlin 1.8.20 offers a new way of setting up source set hierarchy in your multiplatform projects − the default target hierarchy. The new approach is intended to replace target shortcuts like `ios`, which have their design flaws.

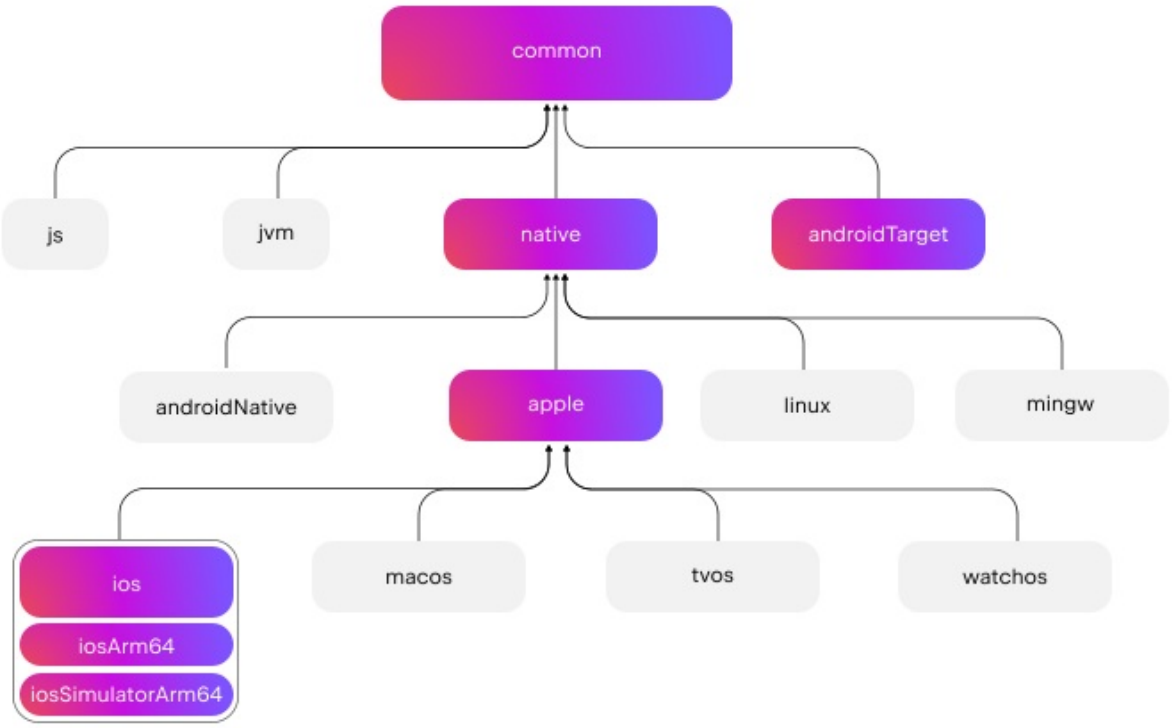
The idea behind the default target hierarchy is simple: You explicitly declare all the targets to which your project compiles, and the Kotlin Gradle plugin automatically creates shared source sets based on the specified targets.

搭建项目

Consider this example of a simple multiplatform mobile app:

```
@OptIn(ExperimentalKotlinGradlePluginApi::class)
kotlin {
    // Enable the default target hierarchy:
    targetHierarchy.default()

    android()
    iosArm64()
    iosSimulatorArm64()
}
```

You can think of the default target hierarchy as a template for all possible targets and their shared source sets. When you declare the final targets `android`, `iosArm64`, and `iosSimulatorArm64` in your code, the Kotlin Gradle plugin finds suitable shared source sets from the template and creates them for you. The resulting hierarchy looks like this:



{thumbnail="true" width="350" thumbnail-same-file="true"}

Green source sets are actually created and present in the project, while gray ones from the default template are ignored. As you can see, the Kotlin Gradle plugin hasn't created the `watchos` source set, for example, because there are no watchOS targets in the project.

If you add a watchOS target, such as `watchosArm64`, the `watchos` source set is created, and the code from the `apple`, `native`, and `common` source sets is compiled to `watchosArm64`, as well.

You can find the complete scheme for the default target hierarchy in the documentation.

> In this example, the `apple` and `native` source sets compile only to the `iosArm64` and `iosSimulatorArm64` targets. Therefore, despite their names, they have access to the full iOS API. This might be counter-intuitive for source sets like `native`, as you may expect that only APIs available on all native targets are accessible in this source set. This behavior may change in the future.
>
> 

为什么要更换快捷方式

Creating source sets hierarchies can be verbose, error-prone, and unfriendly for beginners. Our previous solution was to introduce shortcuts like `ios` that create a part of the hierarchy for you. However, working with shortcuts proved they have a big design flaw: they're difficult to change.

Take the `ios` shortcut, for example. It creates only the `iosArm64` and `iosX64` targets, which can be confusing and may lead to issues when working on an M1-based host that requires the `iosSimulatorArm64` target as well. However, adding the `iosSimulatorArm64` target can be a very disruptive change for user projects:

- All dependencies used in the `iosMain` source set have to support the `iosSimulatorArm64` target; otherwise, the dependency resolution fails.
- Some native APIs used in `iosMain` may disappear when adding a new target (though this is unlikely in the case of `iosSimulatorArm64`).
- In some cases, such as when writing a small pet project on your Intel-based MacBook, you might not even need this change.

It became clear that shortcuts didn't solve the problem of configuring hierarchies, which is why we stopped adding new shortcuts at some point.

The default target hierarchy may look similar to shortcuts at first glance, but they have a crucial distinction: **users have to explicitly specify the set of targets**. This set defines how your project is compiled and published and how it participates in dependency resolution. Since this set is fixed, changes to the default configuration from the Kotlin Gradle plugin should cause significantly less distress in the ecosystem, and it will be much easier to provide tooling-assisted migration.

如何启用默认分层结构

This new feature is Experimental. For Kotlin Gradle build scripts, you need to opt in with `@OptIn(ExperimentalKotlinGradlePluginApi::class)`.

For more information, see Hierarchical project structure.

敬请反馈

This is a significant change to multiplatform projects. We would appreciate your feedback to help make it even better.

Kotlin 多平台支持 Gradle 复合构建预览版

> This feature has been supported in Gradle builds since Kotlin Gradle Plugin 1.8.20. For IDE support, use IntelliJ IDEA 2023.1 Beta 2 (231.8109.2) or later and the Kotlin Gradle plugin 1.8.20 with any Kotlin IDE plugin.
>
> 

Starting with 1.8.20, Kotlin Multiplatform supports Gradle composite builds. Composite builds allow you to include builds of separate projects or parts of the same project into a single build.

Due to some technical challenges, using Gradle composite builds with Kotlin Multiplatform was only partially supported. Kotlin 1.8.20 contains a preview of the improved support that should work with a larger variety of projects. To try it out, add the following option to your `gradle.properties`:

```
kotlin.mpp.import.enableKgpDependencyResolution=true
```

This option enables a preview of the new import mode. Besides the support for composite builds, it provides a smoother import experience in multiplatform projects, as we've included major bug fixes and improvements to make the import more stable.

已知问题

It's still a preview version that needs further stabilization, and you might encounter some issues with import along the way. Here are some known issues we're planning to fix before the final release of Kotlin 1.8.20:

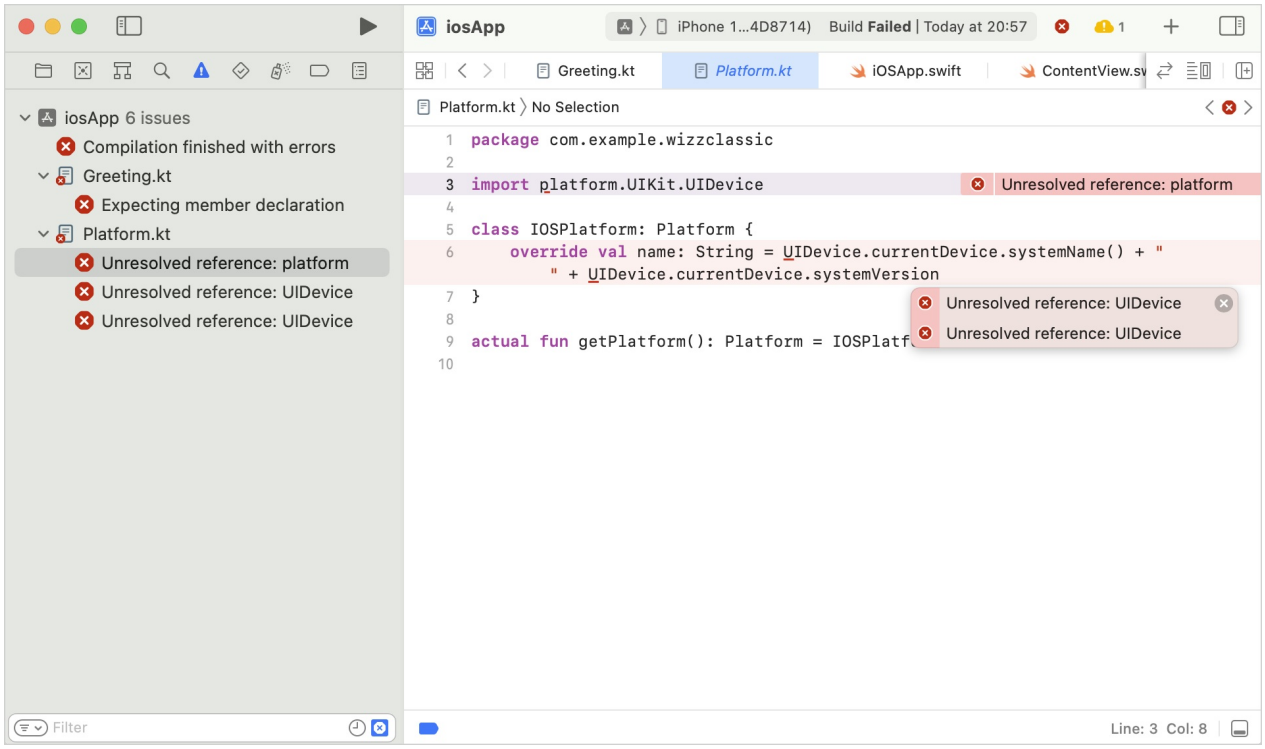
- There's no Kotlin 1.8.20 plugin available for IntelliJ IDEA 2023.1 EAP yet. Despite that, you can still set the Kotlin Gradle plugin version to 1.8.20 and try out composite builds in this IDE.
- If your projects include builds with a specified `rootProject.name`, composite builds may fail to resolve the Kotlin metadata. For the workaround and details, see this Youtrack issue.

We encourage you to try it out and submit all reports on YouTrack to help us make it the default in Kotlin 1.9.0.

改进了 Xcode 中 Gradle 错误的输出

If you had issues building your multiplatform projects in Xcode, you might have encountered a "Command PhaseScriptExecution failed with a nonzero exit code" error. This message signals that the Gradle invocation has failed, but it's not very helpful when trying to detect the problem.

Starting with Kotlin 1.8.20, Xcode can parse the output from the Kotlin/Native compiler. Furthermore, in case the Gradle build fails, you'll see an additional error message from the root cause exception in Xcode. In most cases, it'll help to identify the root problem.



The new behavior is enabled by default for the standard Gradle tasks for Xcode integration, like `embedAndSignAppleFrameworkForXcode` that can connect the iOS framework from your multiplatform project to the iOS application in Xcode. It can also be enabled (or disabled) with the `kotlin.native.useXcodeMessageStyle` Gradle property.

Kotlin/JavaScript

Kotlin 1.8.20 changes the ways TypeScript definitions can be generated. It also includes a change designed to improve your debugging experience:

- 删除了 Gradle plugin 中的 Dukat 集成
- 源代码映射中的 Kotlin 变量与函数名称
- 选择加入 TypeScript 定义文件的生成

删除了 Gradle plugin 中的 Dukat 集成

In Kotlin 1.8.20, we've removed our Experimental Dukat integration from the Kotlin/JavaScript Gradle plugin. The Dukat integration supported the automatic conversion of TypeScript declaration files (`.d.ts`) into Kotlin external declarations.

You can still convert TypeScript declaration files (`.d.ts`) into Kotlin external declarations by using our Dukat tool instead.

> The Dukat tool is Experimental. It may be dropped or changed at any time.
>
> 

源代码映射中的 Kotlin 变量与函数名称

To help with debugging, we've introduced the ability to add the names that you declared in Kotlin code for variables and functions into your source maps. Prior to 1.8.20, these weren't available in source maps, so in the debugger, you always saw the variable and function names of the generated JavaScript.

You can configure what is added by using `sourceMapNamesPolicy` in your Gradle file `build.gradle.kts`, or the `-source-map-names-policy` compiler option. The table below lists the possible settings:

Setting	Description	Example output
`simple-names`	Variable names and simple function names are added. (Default)	`main`
`fully-qualified-names`	Variable names and fully qualified function names are added.	`com.example.kjs.playground.main`
`no`	No variable or function names are added.	N/A

See below for an example configuration in a `build.gradle.kts` file:

```
tasks.withType<org.jetbrains.kotlin.gradle.tasks.Kotlin2JsCompile>().configureEach {
    compilercompileOptions.sourceMapNamesPolicy.set(org.jetbrains.kotlin.gradle.dsl.JsSourceMapNamesPolicy.SOURCE_MAP_NAMES_POLICY_FQ_NAMES) // or SOURCE_MAP_NAMES_POLICY_NO, or SOURCE_MAP_NAMES_POLICY_SIMPLE_NAMES
}
```

Debugging tools like those provided in Chromium-based browsers can pick up the original Kotlin names from your source map to improve the readability of your stack trace. Happy debugging!

> The addition of variable and function names in source maps is Experimental. It may be dropped or changed at any time.
>
> 

选择加入 TypeScript 定义文件的生成

Previously, if you had a project that produced executable files (`binaries.executable()`), the Kotlin/JS IR compiler collected any top-level declarations marked with `@JsExport` and automatically generated TypeScript definitions in a `.d.ts` file.

As this isn't useful for every project, we've changed the behavior in Kotlin 1.8.20. If you want to generate TypeScript definitions, you have to explicitly configure this in your Gradle build file. Add `generateTypeScriptDefinitions()` to your `build.gradle.kts.file` in the `js` section. For

example:

```
kotlin {
    js {
        binaries.executable()
        browser {
        }
        generateTypeScriptDefinitions()
    }
}
```

> The generation of TypeScript definitions (`d.ts`) is Experimental. It may be dropped or changed at any time.
>
> 

Gradle

Kotlin 1.8.20 is fully compatible with Gradle 6.8 through 7.6 except for some special cases in the Multiplatform plugin. You can also use Gradle versions up to the latest Gradle release, but if you do, keep in mind that you might encounter deprecation warnings or some new Gradle features might not work.

This version brings the following changes:

- 新的 Gradle 插件版本对齐
- Gradle 中默认新版 JVM 增量编译
- 精确备份编译任务的输出
- 为所有 Gradle 版本惰性创建 Kotlin/JVM 任务
- 编译任务的 destinationDirectory 支持非默认位置
- 能够选择退出向 HTTP 统计服务报告编译器参数

新的 Gradle 插件版本对齐

Gradle provides a way to ensure dependencies that must work together are always aligned in their versions. Kotlin 1.8.20 adopted this approach, too. It works by default so that you don't need to change or update your configuration to enable it. In addition, you no longer need to resort to this workaround for resolving Kotlin Gradle plugins' transitive dependencies.

We would appreciate your feedback on this feature in YouTrack.

Gradle 中默认新版 JVM 增量编译

The new approach to incremental compilation, which has been available since Kotlin 1.7.0, now works by default. You no longer need to specify `kotlin.incremental.useClasspathSnapshot=true` in your `gradle.properties` to enable it.

We would appreciate your feedback on this. You can file an issue in YouTrack.

精确备份编译任务的输出

> Precise backup of compilation tasks' outputs is Experimental. To use it, add `kotlin.compiler.preciseCompilationResultsBackup=true` to `gradle.properties`. We would appreciate your feedback on it in YouTrack.

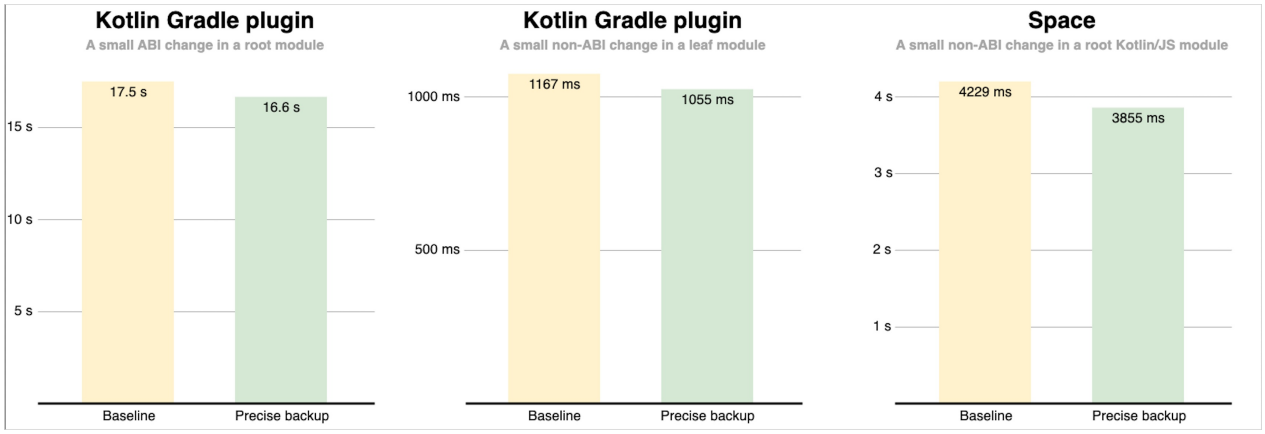
Starting with Kotlin 1.8.20, you can enable precise backup, whereby only those classes that Kotlin recompiles in the incremental compilation will be backed up. Both full and precise backups help to run builds incrementally again after compilation errors. Precise backup also saves build time compared to full backup. Full backup may take **noticeable** build time in large projects or if many tasks are making backups, especially if a project is located on a slow HDD.

This optimization is Experimental. You can enable it by adding the `kotlin.compiler.preciseCompilationResultsBackup` Gradle property to the `gradle.properties` file:

```
kotlin.compiler.preciseCompilationResultsBackup=true
```

JetBrains 精确备份使用示例

In the following charts, you can see examples of using precise backup compared to full backup:



The first and second charts show how precise backup in the Kotlin project affects building the Kotlin Gradle plugin:

1. After making a small ABI change – adding a new public method – to a module that lots of

modules depend on.
2. After making a small non-ABI change – adding a private function – to a module that no other modules depend on.

The third chart shows how precise backup in the Space project affects building a web frontend after a small non-ABI change – adding a private function – to a Kotlin/JS module that lots of modules depend on.

These measurements were performed on a computer with the Apple M1 Max CPU; different computers will yield slightly different results. The factors affecting performance include but are not limited to:

- How warm the Kotlin daemon and the Gradle daemon are.
- How fast or slow the disk is.
- The CPU model and how busy it is.
- Which modules are affected by the changes and how big these modules are.
- Whether the changes are ABI or non-ABI.

使用构建报告评估优化

To estimate the impact of the optimization on your computer for your project and your scenarios, you can use Kotlin build reports. Enable reports in the text file format by adding the following property to your `gradle.properties` file:

```
kotlin.build.report.output=file
```

Here is an example of a relevant part of the report before enabling precise backup:

```
Task ':kotlin-gradle-plugin:compileCommonKotlin' finished in 0.59 s
<...>
Time metrics:
 Total Gradle task time: 0.59 s
 Task action before worker execution: 0.24 s
  Backup output: 0.22 s // Pay attention to this number
<...>
```

And here is an example of a relevant part of the report after enabling precise backup:

```
Task ':kotlin-gradle-plugin:compileCommonKotlin' finished in 0.46 s
<...>
Time metrics:
 Total Gradle task time: 0.46 s
 Task action before worker execution: 0.07 s
  Backup output: 0.05 s // The time has reduced
```

```
  Run compilation in Gradle worker: 0.32 s
   Clear jar cache: 0.00 s
   Precise backup output: 0.00 s // Related to precise backup
   Cleaning up the backup stash: 0.00 s // Related to precise backup
<...>
```

为所有 Gradle 版本惰性创建 Kotlin/JVM 任务

For projects with the `org.jetbrains.kotlin.gradle.jvm` plugin on Gradle 7.3+, the Kotlin Gradle plugin no longer creates and configures the task `compileKotlin` eagerly. On lower Gradle versions, it simply registers all the tasks and doesn't configure them on a dry run. The same behavior is now in place when using Gradle 7.3+.

编译任务的 destinationDirectory 支持非默认位置

Update your build script with some additional code if you do one of the following:

- Override the Kotlin/JVM `KotlinJvmCompile` / `KotlinCompile` task's `destinationDirectory` location.
- Use a deprecated Kotlin/JS/Non-IR variant and override the `Kotlin2JsCompile` task's `destinationDirectory`.

You need to explicitly add `sourceSets.main.kotlin.classesDirectories` to `sourceSets.main.outputs` in your JAR file:

```
tasks.jar(type: Jar) {
    from sourceSets.main.outputs
    from sourceSets.main.kotlin.classesDirectories
}
```

能够选择退出向 HTTP 统计服务报告编译器参数

You can now control whether the Kotlin Gradle plugin should include compiler arguments in HTTP build reports. Sometimes, you might not need the plugin to report these arguments. If a project contains many modules, its compiler arguments in the report can be very heavy and not that helpful. There is now a way to disable it and thus save memory. In your `gradle.properties` or `local.properties`, use the `kotlin.build.report.include_compiler_arguments=(true|false)` property.

We would appreciate your feedback on this feature on YouTrack.

标准库

Kotlin 1.8.20 adds a variety of new features, including some that are particularly useful for Kotlin/Native development:

- 支持 AutoCloseable 接口
- 支持 Base64 编解码
- 在 Kotlin/Native 平台支持 @Volatile
- 修复在 Kotlin/Native 中使用正则表达式时的栈溢出错误

支持 AutoCloseable 接口

> The new `AutoCloseable` interface is Experimental, and to use it you need to opt in with `@OptIn(ExperimentalStdlibApi::class)` or the compiler argument `-opt-in=kotlin.ExperimentalStdlibApi`.



The `AutoCloseable` interface has been added to the common standard library so that you can use one common interface for all libraries to close resources. In Kotlin/JVM, the `AutoCloseable` interface is an alias for `java.lang.AutoClosable`.

In addition, the extension function `use()` is now included, which executes a given block function on the selected resource and then closes it down correctly, whether an exception is thrown or not.

There is no public class in the common standard library that implements the `AutoCloseable` interface. In the example below, we define the `XMLWriter` interface and assume that there is a resource that implements it. For example, this resource could be a class that opens a file, writes XML content, and then closes it.

```
interface XMLWriter : AutoCloseable {
    fun document(encoding: String, version: String, content: XMLWriter.() -> Unit)
    fun element(name: String, content: XMLWriter.() -> Unit)
    fun attribute(name: String, value: String)
    fun text(value: String)
}

fun writeBooksTo(writer: XMLWriter) {
    writer.use { xml ->
        xml.document(encoding = "UTF-8", version = "1.0") {
            element("bookstore") {
                element("book") {
                    attribute("category", "fiction")
                    element("title") { text("Harry Potter and the Prisoner of Azkaban"
) }
                    element("author") { text("J. K. Rowling") }
                    element("year") { text("1999") }
                    element("price") { text("29.99") }
                }
```

```
                element("book") {
                    attribute("category", "programming")
                    element("title") { text("Kotlin in Action") }
                    element("author") { text("Dmitry Jemerov") }
                    element("author") { text("Svetlana Isakova") }
                    element("year") { text("2017") }
                    element("price") { text("25.19") }
                }
            }
        }
    }
}
```

支持 Base64 编码

> The new encoding and decoding functionality is Experimental, and to use it, you need to opt in with `@OptIn(ExperimentalEncodingApi::class)` or the compiler argument `-opt-in=kotlin.io.encoding.ExperimentalEncodingApi`.
>
> 

We've added support for Base64 encoding and decoding. We provide 3 class instances, each using different encoding schemes and displaying different behaviors. Use the `Base64.Default` instance for the standard Base64 encoding scheme.

Use the `Base64.UrlSafe` instance for the "URL and Filename safe" encoding scheme.

Use the `Base64.Mime` instance for the MIME encoding scheme. When you use the `Base64.Mime` instance, all encoding functions insert a line separator every 76 characters. In the case of decoding, any illegal characters are skipped and don't throw an exception.

> The `Base64.Default` instance is the companion object of the `Base64` class. As a result, you can call its functions via `Base64.encode()` and `Base64.decode()` instead of `Base64.Default.encode()` and `Base64.Default.decode()`.
>
> 

```
val foBytes = "fo".map { it.code.toByte() }.toByteArray()
Base64.Default.encode(foBytes) // "Zm8="
// Alternatively:
// Base64.encode(foBytes)

val foobarBytes = "foobar".map { it.code.toByte() }.toByteArray()
Base64.UrlSafe.encode(foobarBytes) // "Zm9vYmFy"

Base64.Default.decode("Zm8=") // foBytes
// Alternatively:
// Base64.decode("Zm8=")
```

```
Base64.UrlSafe.decode("Zm9vYmFy") // foobarBytes
```

You can use additional functions to encode or decode bytes into an existing buffer, as well as to append the encoding result to a provided `Appendable` type object.

In Kotlin/JVM, we've also added the extension functions `encodingWith()` and `decodingWith()` to enable you to perform Base64 encoding and decoding with input and output streams.

在 Kotlin/Native 平台支持 @Volatile

> `@Volatile` in Kotlin/Native is Experimental. It may be dropped or changed at any time. Opt-in is required (see details below). Use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack.
>
> 

If you annotate a `var` property with `@Volatile`, then the backing field is marked so that any reads or writes to this field are atomic, and writes are always made visible to other threads.

Prior to 1.8.20, the `kotlin.jvm.Volatile` annotation was available in the common standard library. However, this annotation is only effective in the JVM. If you use it in Kotlin/Native, it is ignored, which can lead to errors.

In 1.8.20, we've introduced a common annotation, `kotlin.concurrent.Volatile`, that you can use in both the JVM and Kotlin/Native.

如何启用

To try this feature out, opt in with `@OptIn(ExperimentalStdlibApi)` and enable the `-language-version 1.9` compiler option. In a Gradle project, you can do so by adding the following to your `build.gradle(.kts)` file:

> 【Kotlin】

```
tasks
    .withType<org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTask<*>>()
    .configureEach {
        compilerOptions
            .languageVersion
            .set(
                org.jetbrains.kotlin.gradle.dsl.KotlinVersion.KOTLIN_1_9
            )
    }
```

【Groovy】

```
tasks
    .withType(org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTask.class)
    .configureEach {
        compilerOptions.languageVersion =
            org.jetbrains.kotlin.gradle.dsl.KotlinVersion.KOTLIN_1_9
    }
```

修复在 Kotlin/Native 中使用正则表达式时的栈溢出错误

In previous versions of Kotlin, a crash could occur if your regex input contained a large number of characters, even when the regex pattern was very simple. In 1.8.20, this issue has been resolved. For more information, see KT-46211.

序列化更新

Kotlin 1.8.20 comes with Alpha support for the Kotlin K2 compiler and prohibits serializer customization via companion object.

Kotlin K2 编译器的原型序列化编译器插件

> Support for the serialization compiler plugin for K2 is in Alpha. To use it, enable the Kotlin K2 compiler.
>
> 

Starting with 1.8.20, the serialization compiler plugin works with the Kotlin K2 compiler. Give it a try and share your feedback with us!

禁止通过伴生对象进行隐式序列化器定制

Currently, it is possible to declare a class as serializable with the `@Serializable` annotation and, at the same time, declare a custom serializer with the `@Serializer` annotation on its companion object.

For example:

```
import kotlinx.serialization.*
```

```
@Serializable
class Foo(val a: Int) {
    @Serializer(Foo::class)
    companion object {
        // Custom implementation of KSerializer<Foo>
    }
}
```

In this case, it's not clear from the `@Serializable` annotation which serializer is used. In actual fact, class `Foo` has a custom serializer.

To prevent this kind of confusion, in Kotlin 1.8.20 we've introduced a compiler warning for when this scenario is detected. The warning includes a possible migration path to resolve this issue.

If you use such constructs in your code, we recommend updating them to the below:

```
import kotlinx.serialization.*

@Serializable(Foo.Companion::class)
class Foo(val a: Int) {
    // Doesn't matter if you use @Serializer(Foo::class) or not
    companion object: KSerializer<Foo> {
        // Custom implementation of KSerializer<Foo>
    }
}
```

With this approach, it is clear that the `Foo` class uses the custom serializer declared in the companion object. For more information, see our YouTrack ticket.

> In Kotlin 2.0, we plan to promote the compile warning to a compiler error. We recommend that you migrate your code if you see this warning.
>
> 

文档更新

The Kotlin documentation has received some notable changes:

- Get started with Spring Boot and Kotlin – create a simple application with a database and learn more about the features of Spring Boot and Kotlin.
- Scope functions – learn how to simplify your code with useful scope functions from the standard library.
- CocoaPods integration – set up an environment to work with CocoaPods.

安装 Kotlin 1.8.20

核对 IDE 版本

IntelliJ IDEA 2022.2 and 2022.3 automatically suggest updating the Kotlin plugin to version 1.8.20. IntelliJ IDEA 2023.1 has the built-in Kotlin plugin 1.8.20.

Android Studio Flamingo (222) and Giraffe (223) will support Kotlin 1.8.20 in the next releases.

The new command-line compiler is available for download on the GitHub release page.

配置 Gradle 设置

To download Kotlin artifacts and dependencies properly, update your `settings.gradle(.kts)` file to use the Maven Central repository:

```
pluginManagement {
    repositories {
        mavenCentral()
        gradlePluginPortal()
    }
}
```

If the repository is not specified, Gradle uses the sunset JCenter repository that could lead to issues with Kotlin artifacts.

Kotlin 1.8.0 的新特性

发布于：*2022-12-28*

The Kotlin 1.8.0 release is out and here are some of its biggest highlights:

- JVM 平台新增实验性函数：递归复制或删除目录内容
- 改进了 kotlin-reflect 性能
- 新增 -Xdebug 编译器选项以提供更好的调试体验
- `kotlin-stdlib-jdk7` 与 `kotlin-stdlib-jdk8` 合并进了 `kotlin-stdlib`
- 改进了 Objective-C/Swift 互操作性
- 与 Gradle 7.3 的兼容性

IDE 支持

The Kotlin plugin that supports 1.8.0 is available for:

IDE	所支持版本
IntelliJ IDEA	2021.3、 2022.1、 2022.2
Android Studio	Electric Eel（221）、 Flamingo（222）

> You can update your projects to Kotlin 1.8.0 in IntelliJ IDEA 2022.3 without updating the IDE plugin.
>
> To migrate existing projects to Kotlin 1.8.0 in IntelliJ IDEA 2022.3, change the Kotlin version to `1.8.0` and reimport your Gradle or Maven project.

Kotlin/JVM

Starting with version 1.8.0, the compiler can generate classes with a bytecode version corresponding to JVM 19. The new language version also includes:

- 关闭 JVM 注解目标生成的编译器选项
- 用于禁用优化的新增 `-Xdebug` 编译器选项
- 删除旧版后端
- 支持 Lombok 的 @Builder 注解

不生成 TYPE_USE 与 TYPE_PARAMETER 注解目标的能力

If a Kotlin annotation has `TYPE` among its Kotlin targets, the annotation maps to `java.lang.annotation.ElementType.TYPE_USE` in its list of Java annotation targets. This is just like how the `TYPE_PARAMETER` Kotlin target maps to the `java.lang.annotation.ElementType.TYPE_PARAMETER` Java target. This is an issue for Android clients with API levels less than 26, which don't have these targets in the API.

Starting with Kotlin 1.8.0, you can use the new compiler option `-Xno-new-java-annotation-targets` to avoid generating the `TYPE_USE` and `TYPE_PARAMETER` annotation targets.

用于禁用优化的新增编译器选项

Kotlin 1.8.0 adds a new `-Xdebug` compiler option, which disables optimizations for a better debugging experience. For now, the option disables the "was optimized out" feature for coroutines. In the future, after we add more optimizations, this option will disable them, too.

The "was optimized out" feature optimizes variables when you use suspend functions. However, it is difficult to debug code with optimized variables because you don't see their values.

> **Never use this option in production**: Disabling this feature via `-Xdebug` can cause memory leaks.
>
> 

删除旧版后端

In Kotlin 1.5.0, we announced that the IR-based backend became Stable. That meant that the old backend from Kotlin 1.4.* was deprecated. In Kotlin 1.8.0, we've removed the old backend completely. By extension, we've removed the compiler option `-Xuse-old-backend` and the Gradle `useOldBackend` option.

支持 Lombok 的 @Builder 注解

The community has added so many votes for the Kotlin Lombok: Support generated builders (@Builder) YouTrack issue that we just had to support the @Builder annotation.

We don't yet have plans to support the `@SuperBuilder` or `@Tolerate` annotations, but we'll reconsider if enough people vote for the @SuperBuilder and @Tolerate issues.

Learn how to configure the Lombok compiler plugin.

Kotlin/Native

Kotlin 1.8.0 includes changes to Objective-C and Swift interoperability, support for Xcode 14.1, and improvements to the CocoaPods Gradle plugin:

- 支持 Xcode 14.1
- 改进了 Objective-C/Swift 互操作性
- CocoaPods Gradle 插件中默认动态 framework

支持 Xcode 14.1

The Kotlin/Native compiler now supports the latest stable Xcode version, 14.1. The compatibility improvements include the following changes:

- There's a new `watchosDeviceArm64` preset for the watchOS target that supports Apple watchOS on ARM64 platforms.
- The Kotlin CocoaPods Gradle plugin no longer has bitcode embedding for Apple frameworks by default.
- Platform libraries were updated to reflect the changes to Objective-C frameworks for Apple targets.

改进了 Objective-C/Swift 互操作性

To make Kotlin more interoperable with Objective-C and Swift, three new annotations were added:

- `@ObjCName` allows you to specify a more idiomatic name in Swift or Objective-C, instead of renaming the Kotlin declaration.

  The annotation instructs the Kotlin compiler to use a custom Objective-C and Swift name for this class, property, parameter, or function:

  ```
  @ObjCName(swiftName = "MySwiftArray")
  class MyKotlinArray {
      @ObjCName("index")
      fun indexOf(@ObjCName("of") element: String): Int = TODO()
  }

  // Usage with the ObjCName annotations
  let array = MySwiftArray()
  let index = array.index(of: "element")
  ```

- `@HiddenFromObjC` allows you to hide a Kotlin declaration from Objective-C.

  The annotation instructs the Kotlin compiler not to export a function or property to Objective-C and, consequently, Swift. This can make your Kotlin code more Objective-C/Swift-friendly.

- `@ShouldRefineInSwift` is useful for replacing a Kotlin declaration with a wrapper written in Swift.

  The annotation instructs the Kotlin compiler to mark a function or property as `swift_private` in the generated Objective-C API. Such declarations get the `__` prefix, which makes them invisible to Swift code.

  You can still use these declarations in your Swift code to create a Swift-friendly API, but they won't be suggested by Xcode's autocompletion, for example.

  For more information on refining Objective-C declarations in Swift, see the official Apple documentation.

> The new annotations require opt-in.
>
> 

The Kotlin team is very grateful to Rick Clephas for implementing these annotations.

CocoaPods Gradle 插件中默认动态 framework

Starting with Kotlin 1.8.0, Kotlin frameworks registered by the CocoaPods Gradle plugin are linked dynamically by default. The previous static implementation was inconsistent with the behavior of the Kotlin Gradle plugin.

```
kotlin {
    cocoapods {
        framework {
            baseName = "MyFramework"
            isStatic = false // Now dynamic by default
        }
    }
}
```

If you have an existing project with a static linking type and you upgrade to Kotlin 1.8.0 (or change the linking type explicitly), you may encounter an error with the project's execution. To fix it, close your Xcode project and run `pod install` in the Podfile directory.

For more information, see the CocoaPods Gradle plugin DSL reference.

Kotlin 多平台：新版 Android 源代码集布局

Kotlin 1.8.0 introduces a new Android source set layout that replaces the previous naming schema for directories, which is confusing in multiple ways.

Consider an example of two `androidTest` directories created in the current layout. One is for `KotlinSourceSets` and the other is for `AndroidSourceSets`:

- They have different semantics: Kotlin's `androidTest` belongs to the `unitTest` type, whereas Android's belongs to the `integrationTest` type.
- They create a confusing `SourceDirectories` layout, as `src/androidTest/kotlin` has a `UnitTest` and `src/androidTest/java` has an `InstrumentedTest`.
- Both `KotlinSourceSets` and `AndroidSourceSets` use a similar naming schema for Gradle configurations, so the resulting configurations of `androidTest` for both Kotlin's and Android's source sets are the same: `androidTestImplementation`, `androidTestApi`, `androidTestRuntimeOnly`, and `androidTestCompileOnly`.

To address these and other existing issues, we've introduced a new Android source set layout. Here are some of the key differences between the two layouts:

KotlinSourceSet 命名模式

当前源代码集布局	新版源代码集布局
`targetName` + `AndroidSourceSet.name`	`targetName` + `AndroidVariantType`

`{AndroidSourceSet.name}` maps to `{KotlinSourceSet.name}` as follows:

	当前源代码集布局	新版源代码集布局
main	androidMain	androidMain
test	androidTest	android**Unit**Test
androidTest	android**Android**Test	android**Instrumented**Test

SourceDirectories

当前源代码集布局	新版源代码集布局
The layout adds additional `/kotlin` SourceDirectories	`src/{AndroidSourceSet.name}/kotlin`, `src/{KotlinSourceSet.name}/kotlin`

`{AndroidSourceSet.name}` maps to `{SourceDirectories included}` as follows:

	当前源代码集布局	新版源代码集布局
main	src/androidMain/kotlin, src/main/kotlin, src/main/java	src/androidMain/kotlin, src/main/kotlin, src/main/java
test	src/androidTest/kotlin, src/test/kotlin, src/test/java	src/android**Unit**Test/kotlin, src/test/kotlin, src/test/java
androidTest	src/android**Android**Test/kotlin,	src/android**Instrumented**Test/kotlin,

androidTest	src/androidTest/java	src/androidTest/java, **src/androidTest/kotlin**

AndroidManifest.xml 文件的位置

当前源代码集布局	新版源代码集布局
src/{**Android**SourceSet.name}/AndroidManifest.xml	src/{**Kotlin**SourceSet.name}/AndroidMan

`{AndroidSourceSet.name}` maps to `{AndroidManifest.xml location}` as follows:

	当前源代码集布局	新版源代码集布局
main	src/main/AndroidManifest.xml	src/**android**Main/AndroidManifest.xml
debug	src/debug/AndroidManifest.xml	src/**android**Debug/AndroidManifest.xml

The relation between Android and common tests

The new Android source set layout changes the relation between Android-instrumented tests (renamed to `androidInstrumentedTest` in the new layout) and common tests.

Previously, there was a default `dependsOn` relation between `androidAndroidTest` and `commonTest`. In practice, it meant the following:

- The code in `commonTest` was available in `androidAndroidTest`.
- `expect` declarations in `commonTest` had to have corresponding `actual` implementations in `androidAndroidTest`.
- Tests declared in `commonTest` were also running as Android instrumented tests.

In the new Android source set layout, the `dependsOn` relation is not added by default. If you prefer the previous behavior, manually declare this relation in your `build.gradle.kts` file:

```
kotlin {
    // ...
    sourceSets {
        val commonTest by getting
        val androidInstrumentedTest by getting {
            dependsOn(commonTest)
        }
    }
}
```

Support for Android flavors

Previously, the Kotlin Gradle plugin eagerly created source sets that correspond to Android source sets with `debug` and `release` build types or custom flavors like `demo` and `full`. It made them accessible by constructions like `val androidDebug by getting { ... }`.

In the new Android source set layout, those source sets are created in the `afterEvaluate` phase. It makes such expressions invalid, leading to errors like `org.gradle.api.UnknownDomainObjectException: KotlinSourceSet with name 'androidDebug' not found`.

To work around that, use the new `invokeWhenCreated()` API in your `build.gradle.kts` file:

```
kotlin {
    // ...
    sourceSets.invokeWhenCreated("androidFreeDebug") {
        // ...
    }
}
```

配置与设置

The new layout will become the default in future releases. You can enable it now with the following Gradle option:

```
kotlin.mpp.androidSourceSetLayoutVersion=2
```

> The new layout requires Android Gradle plugin 7.0 or later and is supported in Android Studio 2022.3 and later.
>
> 

The usage of the previous Android-style directories is now discouraged. Kotlin 1.8.0 marks the start of the deprecation cycle, introducing a warning for the current layout. You can suppress the warning with the following Gradle property:

```
kotlin.mpp.androidSourceSetLayoutVersion1.nowarn=true
```

Kotlin/JS

Kotlin 1.8.0 stabilizes the JS IR compiler backend and brings new features to JavaScript-related Gradle build scripts:

- 稳定版 JS IR 编译器后端
- 报告 yarn.lock 已更新的新设置

- 通过 Gradle 属性添加浏览器测试目标
- 向项目添加 CSS 支持的新方式

稳定版 JS IR 编译器后端

Starting with this release, the Kotlin/JS intermediate representation (IR-based) compiler backend is Stable. It took a while to unify infrastructure for all three backends, but they now work with the same IR for Kotlin code.

As a consequence of the stable JS IR compiler backend, the old one is deprecated from now on.

Incremental compilation is enabled by default along with the stable JS IR compiler.

If you still use the old compiler, switch your project to the new backend with the help of our migration guide.

报告 yarn.lock 已更新的新设置

If you use the `yarn` package manager, there are three new special Gradle settings that could notify you if the `yarn.lock` file has been updated. You can use these settings when you want to be notified if `yarn.lock` has been changed silently during the CI build process.

These three new Gradle properties are:

- `YarnLockMismatchReport`, which specifies how changes to the `yarn.lock` file are reported. You can use one of the following values:
  - `FAIL` fails the corresponding Gradle task. This is the default.
  - `WARNING` writes the information about changes in the warning log.
  - `NONE` disables reporting.
- `reportNewYarnLock`, which reports about the recently created `yarn.lock` file explicitly. By default, this option is disabled: it's a common practice to generate a new `yarn.lock` file at the first start. You can use this option to ensure that the file has been committed to your repository.
- `yarnLockAutoReplace`, which replaces `yarn.lock` automatically every time the Gradle task is run.

To use these options, update your build script file `build.gradle.kts` as follows:

```
import org.jetbrains.kotlin.gradle.targets.js.yarn.YarnLockMismatchReport
import org.jetbrains.kotlin.gradle.targets.js.yarn.YarnRootExtension

rootProject.plugins.withType(org.jetbrains.kotlin.gradle.targets.js.yarn.YarnPlugin::c
lass.java) {
    rootProject.the<YarnRootExtension>().yarnLockMismatchReport =
        YarnLockMismatchReport.WARNING // NONE | FAIL
    rootProject.the<YarnRootExtension>().reportNewYarnLock = false // true
```

```
    rootProject.the<YarnRootExtension>().yarnLockAutoReplace = false // true
}
```

通过 Gradle 属性添加浏览器测试目标

Starting with Kotlin 1.8.0, you can set test targets for different browsers right in the Gradle properties file. Doing so shrinks the size of the build script file as you no longer need to write all targets in `build.gradle.kts`.

You can use this property to define a list of browsers for all modules, and then add specific browsers in the build scripts of particular modules.

For example, the following line in your Gradle property file will run the test in Firefox and Safari for all modules:

```
kotlin.js.browser.karma.browsers=firefox,safari
```

See the full list of available values for the property on GitHub.

The Kotlin team is very grateful to Martynas Petuška for implementing this feature.

向项目添加 CSS 支持的新方式

This release provides a new approach to adding CSS support to your project. We assume that this will affect a lot of projects, so don't forget to update your Gradle build script files as described below.

Before Kotlin 1.8.0, the `cssSupport.enabled` property was used to add CSS support:

```
browser {
    commonWebpackConfig {
        cssSupport.enabled = true
    }
}
```

Now you should use the `enabled.set()` method in the `cssSupport {}` block:

```
browser {
    commonWebpackConfig {
        cssSupport {
            enabled.set(true)
        }
    }
}
```

Gradle

Kotlin 1.8.0 **fully** supports Gradle versions 7.2 and 7.3. You can also use Gradle versions up to the latest Gradle release, but if you do, keep in mind that you might encounter deprecation warnings or some new Gradle features might not work.

This version brings lots of changes:

- 将 Kotlin 编译器选项暴露为 Gradle 惰性属性
- 提高最低支持版本
- 禁用 Kotlin 守护程序回退策略的能力
- 在传递依赖项中使用最新的 kotlin-stdlib 版本
- 强制检查相关 Kotlin 与 Java 编译任务的 JVM 目标兼容等价性
- 解决 Kotlin Gradle 插件的传递依赖项
- 弃用与删除

将 Kotlin 编译器选项暴露为 Gradle 惰性属性

To expose available Kotlin compiler options as Gradle lazy properties and to integrate them better into the Kotlin tasks, we made lots of changes:

- Compile tasks have the new `compilerOptions` input, which is similar to the existing `kotlinOptions` but uses `Property` from the Gradle Properties API as the return type:

  ```
  tasks.named("compileKotlin", org.jetbrains.kotlin.gradle.tasks.KotlinJvmCompile::c
  lass.java) {
      compilerOptions {
          useK2.set(true)
      }
  }
  ```

- The Kotlin tools tasks `KotlinJsDce` and `KotlinNativeLink` have the new `toolOptions` input, which is similar to the existing `kotlinOptions` input.

- New inputs have the `@Nested` Gradle annotation. Every property inside the inputs has a related Gradle annotation, such as `@Input` or `@Internal`.
- The Kotlin Gradle plugin API artifact has two new interfaces:
  - `org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTask`, which has the `compilerOptions` input and the `compileOptions()` method. All Kotlin compilation tasks implement this interface.
  - `org.jetbrains.kotlin.gradle.tasks.KotlinToolTask`, which has the `toolOptions` input and the `toolOptions()` method. All Kotlin tool tasks – `KotlinJsDce`, `KotlinNativeLink`, and `KotlinNativeLinkArtifactTask` – implement this interface.

- Some `compilerOptions` use the new types instead of the `String` type:
  - `JvmTarget`
  - `KotlinVersion` (for the `apiVersion` and the `languageVersion` inputs)
  - `JsMainFunctionExecutionMode`
  - `JsModuleKind`
  - `JsSourceMapEmbedMode`

  For example, you can use `compilerOptions.jvmTarget.set(JvmTarget.JVM_11)` instead of `kotlinOptions.jvmTarget = "11"`.

  The `kotlinOptions` types didn't change, and they are internally converted to `compilerOptions` types.
- The Kotlin Gradle plugin API is binary-compatible with previous releases. There are, however, some source and ABI-breaking changes in the `kotlin-gradle-plugin` artifact. Most of these changes involve additional generic parameters to some internal types. One important change is that the `KotlinNativeLink` task no longer inherits the `AbstractKotlinNativeCompile` task.
- `KotlinJsCompilerOptions.outputFile` and the related `KotlinJsOptions.outputFile` options are deprecated. Use the `Kotlin2JsCompile.outputFileProperty` task input instead.

> The Kotlin Gradle plugin still adds the `KotlinJvmOptions` DSL to the Android extension:
>
> ```
> android {
>     kotlinOptions {
>         jvmTarget = "11"
>     }
> }
> ```
>
> This will be changed in the scope of this issue, when the `compilerOptions` DSL will be added to a module level.
>
> 

限制

> The `kotlinOptions` task input and the `kotlinOptions{...}` task DSL are in support mode and will be deprecated in upcoming releases. Improvements will be made only to `compilerOptions` and `toolOptions`.
>
> 

Calling any setter or getter on `kotlinOptions` delegates to the related property in the `compilerOptions`. This introduces the following limitations:

- `compilerOptions` and `kotlinOptions` cannot be changed in the task execution phase (see one

exception in the paragraph below).

- `freeCompilerArgs` returns an immutable `List<String>`, which means that, for example, `kotlinOptions.freeCompilerArgs.remove("something")` will fail.

Several plugins, including `kotlin-dsl` and the Android Gradle plugin (AGP) with Jetpack Compose enabled, try to modify the `freeCompilerArgs` attribute in the task execution phase. We've added a workaround for them in Kotlin 1.8.0. This workaround allows any build script or plugin to modify `kotlinOptions.freeCompilerArgs` in the execution phase but produces a warning in the build log. To disable this warning, use the new Gradle property `kotlin.options.suppressFreeCompilerArgsModificationWarning=true`. Gradle is going to add fixes for the `kotlin-dsl` plugin and AGP with Jetpack Compose enabled.

提高最低支持版本

Starting with Kotlin 1.8.0, the minimum supported Gradle version is 6.8.3 and the minimum supported Android Gradle plugin version is 4.1.3.

See the Kotlin Gradle plugin compatibility with available Gradle versions in our documentation

禁用 Kotlin 守护程序回退策略的能力

There is a new Gradle property `kotlin.daemon.useFallbackStrategy`, whose default value is `true`. When the value is `false`, builds fail on problems with the daemon's startup or communication. There is also a new `useDaemonFallbackStrategy` property in Kotlin compile tasks, which takes priority over the Gradle property if you use both. If there is insufficient memory to run the compilation, you can see a message about it in the logs.

The Kotlin compiler's fallback strategy is to run a compilation outside the Kotlin daemon if the daemon somehow fails. If the Gradle daemon is on, the compiler uses the "In process" strategy. If the Gradle daemon is off, the compiler uses the "Out of process" strategy. Learn more about these execution strategies in the documentation. Note that silent fallback to another strategy can consume a lot of system resources or lead to non-deterministic builds; see this YouTrack issue for more details.

在传递依赖项中使用最新的 kotlin-stdlib 版本

If you explicitly write Kotlin version 1.8.0 or higher in your dependencies, for example: `implementation("org.jetbrains.kotlin:kotlin-stdlib:1.8.0")`, then the Kotlin Gradle Plugin will use that Kotlin version for transitive `kotlin-stdlib-jdk7` and `kotlin-stdlib-jdk8` dependencies. This is done to avoid class duplication from different stdlib versions (learn more about merging `kotlin-stdlib-jdk7` and `kotlin-stdlib-jdk8` into `kotlin-stdlib`). You can disable this behavior with the `kotlin.stdlib.jdk.variants.version.alignment` Gradle property:

```
kotlin.stdlib.jdk.variants.version.alignment=false
```

If you run into issues with version alignment, align all versions via the Kotlin BOM by declaring a platform dependency on `kotlin-bom` in your build script:

```
implementation(platform("org.jetbrains.kotlin:kotlin-bom:1.8.0"))
```

Learn about other cases and our suggested solutions in the documentation.

强制检查相关 Kotlin 与 Java 编译任务的 JVM 目标

> This section applies to your JVM project even if your source files are only in Kotlin and you don't use Java.
>
> 

Starting from this release, the default value for the `kotlin.jvm.target.validation.mode` property is `error` for projects on Gradle 8.0+ (this version of Gradle has not been released yet), and the plugin will fail the build in the event of JVM target incompatibility.

The shift of the default value from `warning` to `error` is a preparation step for a smooth migration to Gradle 8.0. **We encourage you to set this property to `error`** and configure a toolchain or align JVM versions manually.

Learn more about what can go wrong if you don't check the targets' compatibility.

解决 Kotlin Gradle 插件的传递依赖项

In Kotlin 1.7.0, we introduced support for Gradle plugin variants. Because of these plugin variants, a build classpath can have different versions of the Kotlin Gradle plugins that depend on different versions of some dependency, usually `kotlin-gradle-plugin-api`. This can lead to a resolution problem, and we would like to propose the following workaround, using the `kotlin-dsl` plugin as an example.

The `kotlin-dsl` plugin in Gradle 7.6 depends on the `org.jetbrains.kotlin.plugin.sam.with.receiver:1.7.10` plugin, which depends on `kotlin-gradle-plugin-api:1.7.10`. If you add the `org.jetbrains.kotlin.gradle.jvm:1.8.0` plugin, this `kotlin-gradle-plugin-api:1.7.10` transitive dependency may lead to a dependency resolution error because of a mismatch between the versions (`1.8.0` and `1.7.10`) and the variant attributes' `org.gradle.plugin.api-version` values. As a workaround, add this constraint to align the versions. This workaround may be needed until we implement the Kotlin Gradle Plugin libraries alignment platform, which is in the plans:

```
dependencies {
    constraints {
        implementation("org.jetbrains.kotlin:kotlin-sam-with-receiver:1.8.0")
    }
}
```

This constraint forces the `org.jetbrains.kotlin:kotlin-sam-with-receiver:1.8.0` version to be used in the build classpath for transitive dependencies. Learn more about one similar case in the Gradle issue tracker.

弃用与删除

In Kotlin 1.8.0, the deprecation cycle continues for the following properties and methods:

- In the notes for Kotlin 1.7.0 that the `KotlinCompile` task still had the deprecated Kotlin property `classpath`, which would be removed in future releases. Now, we've changed the deprecation level to `error` for the `KotlinCompile` task's `classpath` property. All compile tasks use the `libraries` input for a list of libraries required for compilation.
- We removed the `kapt.use.worker.api` property that allowed running kapt via the Gradle Workers API. By default, kapt has been using Gradle workers since Kotlin 1.3.70, and we recommend sticking to this method.
- In Kotlin 1.7.0, we announced the start of a deprecation cycle for the `kotlin.compiler.execution.strategy` property. In this release, we removed this property. Learn how to define a Kotlin compiler execution strategy in other ways.

标准库

Kotlin 1.8.0：

- 更新了 JVM 编译项目标。
- 稳定了一些函数——Java 与 Kotlin 之间的 TimeUnit 转换、 `cbrt()` 、 Java `Optional` 扩展函数。
- 提供可比较且可减去的 `TimeMark` 预览版。
- 包含 `java.nio.file.path` 的实验性扩展函数。
- 附赠改进了 kotlin-reflect 性能。

更新了 JVM 编译项目标

In Kotlin 1.8.0, the standard libraries (`kotlin-stdlib` , `kotlin-reflect` , and `kotlin-script-*`) are compiled with JVM target 1.8. Previously, the standard libraries were compiled with JVM target 1.6.

Kotlin 1.8.0 no longer supports JVM targets 1.6 and 1.7. As a result, you no longer need to declare `kotlin-stdlib-jdk7` and `kotlin-stdlib-jdk8` separately in build scripts because the contents of these artifacts have been merged into `kotlin-stdlib`.

> If you have explicitly declared `kotlin-stdlib-jdk7` and `kotlin-stdlib-jdk8` as dependencies in your build scripts, then you should replace them with `kotlin-stdlib`.
>
> 

Note that mixing different versions of stdlib artifacts could lead to class duplication or to missing classes. To avoid that, the Kotlin Gradle plugin can help you align stdlib versions.

cbrt()

The `cbrt()` function, which allows you to compute the real cube root of a `double` or `float`, is now Stable.

```
import kotlin.math.*

fun main() {
    val num = 27
    val negNum = -num

    println("The cube root of ${num.toDouble()} is: " +
            cbrt(num.toDouble()))
    println("The cube root of ${negNum.toDouble()} is: " +
            cbrt(negNum.toDouble()))
}
```

Java 与 Kotlin 之间的 TimeUnit 转换

The `toTimeUnit()` and `toDurationUnit()` functions in `kotlin.time` are now Stable. Introduced as Experimental in Kotlin 1.6.0, these functions improve interoperability between Kotlin and Java. You can now easily convert between Java `java.util.concurrent.TimeUnit` and Kotlin `kotlin.time.DurationUnit`. These functions are supported on the JVM only.

```
import kotlin.time.*

// For use from Java
fun wait(timeout: Long, unit: TimeUnit) {
    val duration: Duration = timeout.toDuration(unit.toDurationUnit())
    ...
}
```

可比较可减去的 TimeMark

> The new functionality of `TimeMarks` is Experimental, and to use it you need to opt in by using `@OptIn(ExperimentalTime::class)` or `@ExperimentalTime`.
>
> 

Before Kotlin 1.8.0, if you wanted to calculate the time difference between multiple `TimeMarks` and **now**, you could only call `elapsedNow()` on one `TimeMark` at a time. This made it difficult to compare the results because the two `elapsedNow()` function calls couldn't be executed at exactly the same time.

To solve this, in Kotlin 1.8.0 you can subtract and compare `TimeMarks` from the same time source. Now you can create a new `TimeMark` instance to represent **now** and subtract other `TimeMarks` from it. This way, the results that you collect from these calculations are guaranteed to be relative to each other.

```
import kotlin.time.*
fun main() {
//sampleStart
    val timeSource = TimeSource.Monotonic
    val mark1 = timeSource.markNow()
    Thread.sleep(500) // Sleep 0.5 seconds
    val mark2 = timeSource.markNow()

    // Before 1.8.0
    repeat(4) { n ->
        val elapsed1 = mark1.elapsedNow()
        val elapsed2 = mark2.elapsedNow()

        // Difference between elapsed1 and elapsed2 can vary depending
        // on how much time passes between the two elapsedNow() calls
        println("Measurement 1.${n + 1}: elapsed1=$elapsed1, " +
                "elapsed2=$elapsed2, diff=${elapsed1 - elapsed2}")
    }
    println()

    // Since 1.8.0
    repeat(4) { n ->
        val mark3 = timeSource.markNow()
        val elapsed1 = mark3 - mark1
        val elapsed2 = mark3 - mark2

        // Now the elapsed times are calculated relative to mark3,
        // which is a fixed value
        println("Measurement 2.${n + 1}: elapsed1=$elapsed1, " +
                "elapsed2=$elapsed2, diff=${elapsed1 - elapsed2}")
    }
    // It's also possible to compare time marks with each other
    // This is true, as mark2 was captured later than mark1
    println(mark2 > mark1)
//sampleEnd
```

```
}
```

This new functionality is particularly useful in animation calculations where you want to calculate the difference between, or compare, multiple `TimeMarks` representing different frames.

递归复制或删除目录

> These new functions for `java.nio.file.path` are Experimental. To use them, you need to opt in with `@OptIn(kotlin.io.path.ExperimentalPathApi::class)` or `@kotlin.io.path.ExperimentalPathApi`. Alternatively, you can use the compiler option `-opt-in=kotlin.io.path.ExperimentalPathApi`.
>
> 

We have introduced two new extension functions for `java.nio.file.Path`, `copyToRecursively()` and `deleteRecursively()`, which allow you to recursively:

- Copy a directory and its contents to another destination.
- Delete a directory and its contents.

These functions can be very useful as part of a backup process.

错误处理

Using `copyToRecursively()`, you can define what should happen if an exception occurs while copying, by overloading the `onError` lambda function:

```
sourceRoot.copyToRecursively(destinationRoot, followLinks = false,
    onError = { source, target, exception ->
        logger.logError(exception, "Failed to copy $source to $target")
        OnErrorResult.TERMINATE
    })
```

When you use `deleteRecursively()`, if an exception occurs while deleting a file or folder, then the file or folder is skipped. Once the deletion has completed, `deleteRecursively()` throws an `IOException` containing all the exceptions that occurred as suppressed exceptions.

文件覆盖

If `copyToRecursively()` finds that a file already exists in the destination directory, then an exception occurs. If you want to overwrite the file instead, use the overload that has `overwrite` as an argument and set it to `true`:

```
fun setUpEnvironment(projectDirectory: Path, fixtureName: String) {
    fixturesRoot.resolve(COMMON_FIXTURE_NAME)
        .copyToRecursively(projectDirectory, followLinks = false)
    fixturesRoot.resolve(fixtureName)
        .copyToRecursively(projectDirectory, followLinks = false,
            overwrite = true) // patches the common fixture
}
```

自定义复制动作

To define your own custom logic for copying, use the overload that has `copyAction` as an additional argument. By using `copyAction` you can provide a lambda function, for example, with your preferred actions:

```
sourceRoot.copyToRecursively(destinationRoot, followLinks = false) { source, target ->
    if (source.name.startsWith(".")) {
        CopyActionResult.SKIP_SUBTREE
    } else {
        source.copyToIgnoringExistingDirectory(target, followLinks = false)
        CopyActionResult.CONTINUE
    }
}
```

For more information on these extension functions, see our API reference.

Java Optional 扩展函数

The extension functions that were introduced in Kotlin 1.7.0 are now Stable. These functions simplify working with Optional classes in Java. They can be used to unwrap and convert `Optional` objects on the JVM, and to make working with Java APIs more concise. For more information, see What's new in Kotlin 1.7.0.

改进了 kotlin-reflect 性能

Taking advantage of the fact that `kotlin-reflect` is now compiled with JVM target 1.8, we migrated our internal cache mechanism to Java's `ClassValue`. Previously we only cached `KClass`, but we now also cache `KType` and `KDeclarationContainer`. These changes have led to significant performance improvements when invoking `typeOf()`.

文档更新

The Kotlin documentation has received some notable changes:

修订与新增页面

- Gradle overview – learn how to configure and build a Kotlin project with the Gradle build system, available compiler options, compilation, and caches in the Kotlin Gradle plugin.
- Nullability in Java and Kotlin – see the differences between Java's and Kotlin's approaches to handling possibly nullable variables.
- Lincheck guide – learn how to set up and use the Lincheck framework for testing concurrent algorithms on the JVM.

新增与更新教程

- Get started with Gradle and Kotlin/JVM – create a console application using IntelliJ IDEA and Gradle.
- Create a multiplatform app using Ktor and SQLDelight – create a mobile application for iOS and Android using Kotlin Multiplatform Mobile.
- Get started with Kotlin Multiplatform – learn about cross-platform mobile development with Kotlin and create an app that works on both Android and iOS.

安装 Kotlin 1.8.0

IntelliJ IDEA 2021.3, 2022.1, and 2022.2 automatically suggest updating the Kotlin plugin to version 1.8.0. IntelliJ IDEA 2022.3 will have the 1.8.0 version of the Kotlin plugin bundled in an upcoming minor update.

> To migrate existing projects to Kotlin 1.8.0 in IntelliJ IDEA 2022.3, change the Kotlin version to `1.8.0` and reimport your Gradle or Maven project.
>
> 

For Android Studio Electric Eel (221) and Flamingo (222), version 1.8.0 of the Kotlin plugin will be delivered with the upcoming Android Studios updates. The new command-line compiler is available for download on the GitHub release page.

Kotlin 1.8.0 的兼容性指南

Kotlin 1.8.0 is a feature release and can, therefore, bring changes that are incompatible with your code written for earlier versions of the language. Find the detailed list of these changes in the Compatibility guide for Kotlin 1.8.0.

Kotlin 1.7.20 的新特性

The IDE support for Kotlin 1.7.20 is available for IntelliJ IDEA 2021.3, 2022.1, and 2022.2.

发布于：*2022-09-29*

The Kotlin 1.7.20 release is out! Here are some highlights from this release:

- 新版 Kotlin K2 编译器支持 `all-open`、带有接收者的 SAM、Lombok 以及其他编译器插件
- 我们引入了用于创建前闭后开区间的 ..< 操作符预览版
- 现在默认启用新版 Kotlin/Native 内存管理器
- 我们为 JVM 引入了一个新的实验性特性：拥有泛型底层类型的内联类

You can also find a short overview of the changes in this video:

YouTube 视频：What's new in Kotlin 1.7.20

支持 Kotlin K2 编译器插件

The Kotlin team continues to stabilize the K2 compiler. K2 is still in **Alpha** (as announced in the Kotlin 1.7.0 release), but it now supports several compiler plugins. You can follow this YouTrack issue to get updates from the Kotlin team on the new compiler.

Starting with this 1.7.20 release, the Kotlin K2 compiler supports the following plugins:

- `all-open`
- `no-arg`
- 带有接收者的 SAM
- Lombok
- AtomicFU
- `jvm-abi-gen`

> The Alpha version of the new K2 compiler only works with JVM projects. It doesn't support Kotlin/JS, Kotlin/Native, or other multiplatform projects.
>
> 

Learn more about the new compiler and its benefits in the following videos:

- The Road to the New Kotlin Compiler
- K2 Compiler: a Top-Down View

如何启用 Kotlin K2 编译器

To enable the Kotlin K2 compiler and test it, use the following compiler option:

```
-Xuse-k2
```

You can specify it in your `build.gradle(.kts)` file:

> 【Kotlin】

```
tasks.withType<KotlinCompile> {
    kotlinOptions.useK2 = true
}
```

> 【Groovy】

```
compileKotlin {
    kotlinOptions.useK2 = true
}
```

Check out the performance boost on your JVM projects and compare it with the results of the old compiler.

请反馈关于新版 K2 编译器的问题

We really appreciate your feedback in any form:

- Provide your feedback directly to K2 developers in Kotlin Slack: get an invite and join the #k2-early-adopters channel.

- Report any problems you faced with the new K2 compiler to our issue tracker.
- Enable the **Send usage statistics** option to allow JetBrains collecting anonymous data about K2 usage.

语言

Kotlin 1.7.20 introduces preview versions for new language features, as well as puts restrictions on builder type inference:

- 用于创建前闭后开区间的 ..< 操作符预览版
- 新增数据对象声明
- 构建器类型推断限制

用于创建前闭后开区间的 ..< 操作符预览版

> The new operator is Experimental, and it has limited support in the IDE.
>
> 

This release introduces the new `..<` operator. Kotlin has the `..` operator to express a range of values. The new `..<` operator acts like the `until` function and helps you define the open-ended range.

YouTube 视频：New operator for open-ended ranges

Our research shows that this new operator does a better job at expressing open-ended ranges and making it clear that the upper bound is not included.

Here is an example of using the `..<` operator in a `when` expression:

```
when (value) {
    in 0.0..<0.25 -> // First quarter
    in 0.25..<0.5 -> // Second quarter
    in 0.5..<0.75 -> // Third quarter
    in 0.75..1.0 ->  // Last quarter  <- Note closed range here
}
```

标准库 API 变更

The following new types and operations will be introduced in the `kotlin.ranges` packages in the common Kotlin standard library:

新接口 OpenEndRange<T>

The new interface to represent open-ended ranges is very similar to the existing `ClosedRange<T>` interface:

```
interface OpenEndRange<T : Comparable<T>> {
    // Lower bound
    val start: T
    // Upper bound, not included in the range
    val endExclusive: T
    operator fun contains(value: T): Boolean = value >= start && value < endExclusive
    fun isEmpty(): Boolean = start >= endExclusive
}
```

在现有可迭代区间中实现 OpenEndRange

When developers need to get a range with an excluded upper bound, they currently use the `until` function to effectively produce a closed iterable range with the same values. To make these ranges acceptable in the new API that takes `OpenEndRange<T>`, we want to implement that interface in the existing iterable ranges: `IntRange`, `LongRange`, `CharRange`, `UIntRange`, and `ULongRange`. So they will simultaneously implement both the `ClosedRange<T>` and `OpenEndRange<T>` interfaces.

```
class IntRange : IntProgression(...), ClosedRange<Int>, OpenEndRange<Int> {
    override val start: Int
    override val endInclusive: Int
    override val endExclusive: Int
}
```

标准类型的 rangeUntil 运算符

The `rangeUntil` operators will be provided for the same types and combinations currently defined by the `rangeTo` operator. We provide them as extension functions for prototype purposes, but for consistency, we plan to make them members later before stabilizing the open-ended ranges API.

如何启用 ..< 操作符

To use the `..<` operator or to implement that operator convention for your own types, enable the `-language-version 1.8` compiler option.

The new API elements introduced to support the open-ended ranges of the standard types require an opt-in, as usual for an experimental stdlib API: `@OptIn(ExperimentalStdlibApi::class)`. Alternatively, you could use the `-opt-in=kotlin.ExperimentalStdlibApi` compiler option.

Read more about the new operator in this KEEP document.

通过数据对象改进了单例与密封类层次结构的字符串表示

> Data objects are Experimental, and have limited support in the IDE at the moment.
>
> 

This release introduces a new type of `object` declaration for you to use: `data object`. Data object behaves conceptually identical to a regular `object` declaration but comes with a clean `toString` representation out of the box.

YouTube 视频：Data objects in Kotlin 1.7.20

```
package org.example
object MyObject
data object MyDataObject

fun main() {
    println(MyObject) // org.example.MyObject@1f32e575
    println(MyDataObject) // MyDataObject
}
```

This makes `data object` declarations perfect for sealed class hierarchies, where you may use them alongside `data class` declarations. In this snippet, declaring `EndOfFile` as a `data object` instead of a plain `object` means that it will get a pretty `toString` without the need to override it manually, maintaining symmetry with the accompanying `data class` definitions:

```
sealed class ReadResult {
    data class Number(val value: Int) : ReadResult()
    data class Text(val value: String) : ReadResult()
    data object EndOfFile : ReadResult()
}
```

```
fun main() {
    println(ReadResult.Number(1)) // Number(value=1)
    println(ReadResult.Text("Foo")) // Text(value=Foo)
    println(ReadResult.EndOfFile) // EndOfFile
}
```

如何启用数据对象

To use data object declarations in your code, enable the `-language-version 1.9` compiler option. In a Gradle project, you can do so by adding the following to your `build.gradle(.kts)`:

> 【Kotlin】

```
tasks.withType<org.jetbrains.kotlin.gradle.tasks.KotlinCompile>().configureEach {
    // ...
    kotlinOptions.languageVersion = "1.9"
}
```

> 【Groovy】

```
compileKotlin {
    // ...
    kotlinOptions.languageVersion = '1.9'
}
```

Read more about data objects, and share your feedback on their implementation in the respective KEEP document.

新的构建器类型推断限制

Kotlin 1.7.20 places some major restrictions on the use of builder type inference that could affect your code. These restrictions apply to code containing builder lambda functions, where it's impossible to derive the parameter without analyzing the lambda itself. The parameter is used as an argument. Now, the compiler will always show an error for such code and ask you to specify the type explicitly.

This is a breaking change, but our research shows that these cases are very rare, and the restrictions shouldn't affect your code. If they do, consider the following cases:

- Builder inference with extension that hides members.

If your code contains an extension function with the same name that will be used during the builder inference, the compiler will show you an error:

```
class Data {
    fun doSmth() {} // 1
}

fun <T> T.doSmth() {} // 2

fun test() {
    buildList {
        this.add(Data())
        this.get(0).doSmth() // Resolves to 2 and leads to error
    }
}
```

To fix the code, you should specify the type explicitly:

```
class Data {
    fun doSmth() {} // 1
}

fun <T> T.doSmth() {} // 2

fun test() {
    buildList<Data> { // Type argument!
        this.add(Data())
        this.get(0).doSmth() // Resolves to 1
    }
}
```

- Builder inference with multiple lambdas and the type arguments are not specified explicitly.

  If there are two or more lambda blocks in builder inference, they affect the type. To prevent an error, the compiler requires you to specify the type:

```
fun <T: Any> buildList(
    first: MutableList<T>.() -> Unit,
    second: MutableList<T>.() -> Unit
): List<T> {
    val list = mutableListOf<T>()
    list.first()
    list.second()
    return list
}

fun main() {
    buildList(
        first = { // this: MutableList<String>
```

```
                add("")
            },
            second = { // this: MutableList<Int>
                val i: Int = get(0)
                println(i)
            }
        )
    }
```

To fix the error, you should specify the type explicitly and fix the type mismatch:

```
fun main() {
    buildList<Int>(
        first = { // this: MutableList<Int>
            add(0)
        },
        second = { // this: MutableList<Int>
            val i: Int = get(0)
            println(i)
        }
    )
}
```

If you haven't found your case mentioned above, file an issue to our team.

See this YouTrack issue for more information about this builder inference update.

Kotlin/JVM

Kotlin 1.7.20 introduces generic inline classes, adds more bytecode optimizations for delegated properties, and supports IR in the kapt stub generating task, making it possible to use all the newest Kotlin features with kapt:

- 泛型内联类
- 更多属性委托的优化场景
- 在 kapt 存根生成任务中支持 JVM IR 后端

泛型内联类

> Generic inline classes is an Experimental feature. It may be dropped or changed at any time. Opt-in is required (see details below), and you should use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack.
>
> 

Kotlin 1.7.20 allows the underlying type of JVM inline classes to be a type parameter. The compiler maps it to `Any?` or, generally, to the upper bound of the type parameter.

YouTube 视频：Generic inline classes in Kotlin 1.7.20

Consider the following example:

```
@JvmInline
value class UserId<T>(val value: T)

fun compute(s: UserId<String>) {} // Compiler generates fun compute-<hashcode>(s: Any?)
```

The function accepts the inline class as a parameter. The parameter is mapped to the upper bound, not the type argument.

To enable this feature, use the `-language-version 1.8` compiler option.

We would appreciate your feedback on this feature in YouTrack.

更多属性委托的优化场景

In Kotlin 1.6.0, we optimized the case of delegating to a property by omitting the `$delegate` field and generating immediate access to the referenced property. In 1.7.20, we've implemented this optimization for more cases. The `$delegate` field will now be omitted if a delegate is:

- A named object:

  ```
  object NamedObject {
      operator fun getValue(thisRef: Any?, property: KProperty<*>): String = ...
  }

  val s: String by NamedObject
  ```

- A final `val` property with a backing field and a default getter in the same module:

  ```
  val impl: ReadOnlyProperty<Any?, String> = ...
  ```

```
class A {
    val s: String by impl
}
```

- A constant expression, an enum entry, `this`, or `null`. Here's an example of `this`:

```
class A {
    operator fun getValue(thisRef: Any?, property: KProperty<*>) ...

    val s by this
}
```

Learn more about delegated properties.

We would appreciate your feedback on this feature in YouTrack.

在 kapt 存根生成任务中支持 JVM IR 后端

> Support for the JVM IR backend in the kapt stub generating task is an Experimental feature. It may be changed at any time. Opt-in is required (see details below), and you should use it only for evaluation purposes.
>
> 

Before 1.7.20, the kapt stub generating task used the old backend, and repeatable annotations didn't work with kapt. With Kotlin 1.7.20, we've added support for the JVM IR backend in the kapt stub generating task. This makes it possible to use all the newest Kotlin features with kapt, including repeatable annotations.

To use the IR backend in kapt, add the following option to your `gradle.properties` file:

```
kapt.use.jvm.ir=true
```

We would appreciate your feedback on this feature in YouTrack.

Kotlin/Native

Kotlin 1.7.20 comes with the new Kotlin/Native memory manager enabled by default and gives you the option to customize the `Info.plist` file:

- 新的默认内存管理器
- 自定义 Info.plist 文件

默认启用新版 Kotlin/Native 内存管理器

This release brings further stability and performance improvements to the new memory manager, allowing us to promote the new memory manager to Beta.

The previous memory manager complicated writing concurrent and asynchronous code, including issues with implementing the `kotlinx.coroutines` library. This blocked the adoption of Kotlin Multiplatform Mobile because concurrency limitations created problems with sharing Kotlin code between iOS and Android platforms. The new memory manager finally paves the way to promote Kotlin Multiplatform Mobile to Beta.

The new memory manager also supports the compiler cache that makes compilation times comparable to previous releases. For more on the benefits of the new memory manager, see our original blog post for the preview version. You can find more technical details in the documentation.

配置与设置

Starting with Kotlin 1.7.20, the new memory manager is the default. Not much additional setup is required.

If you've already turned it on manually, you can remove the `kotlin.native.binary.memoryModel=experimental` option from your `gradle.properties` or `binaryOptions["memoryModel"] = "experimental"` from the `build.gradle(.kts)` file.

If necessary, you can switch back to the legacy memory manager with the `kotlin.native.binary.memoryModel=strict` option in your `gradle.properties`. However, compiler cache support is no longer available for the legacy memory manager, so compilation times might worsen.

冻结

In the new memory manager, freezing is deprecated. Don't use it unless you need your code to work with the legacy manager (where freezing is still required). This may be helpful for library authors that need to maintain support for the legacy memory manager or developers who want to have a fallback if they encounter issues with the new memory manager.

In such cases, you can temporarily support code for both new and legacy memory managers. To ignore deprecation warnings, do one of the following:

- Annotate usages of the deprecated API with `@OptIn(FreezingIsDeprecated::class)`.
- Apply `languageSettings.optIn("kotlin.native.FreezingIsDeprecated")` to all the Kotlin source sets in Gradle.
- Pass the compiler flag `-opt-in=kotlin.native.FreezingIsDeprecated`.

在 Swift/Objective-C 中调用 Kotlin 挂起函数

The new memory manager still restricts calling Kotlin `suspend` functions from Swift and Objective-C from threads other than the main one, but you can lift it with a new Gradle option.

This restriction was originally introduced in the legacy memory manager due to cases where the code dispatched a continuation to be resumed on the original thread. If this thread didn't have a supported event loop, the task would never run, and the coroutine would never be resumed.

In certain cases, this restriction is no longer required, but a check of all the necessary conditions can't be easily implemented. Because of this, we decided to keep it in the new memory manager while introducing an option for you to disable it. For this, add the following option to your `gradle.properties`:

```
kotlin.native.binary.objcExportSuspendFunctionLaunchThreadRestriction=none
```

> Do not add this option if you use the `native-mt` version of `kotlinx.coroutines` or other libraries that have the same "dispatch to the original thread" approach.
>
> 

The Kotlin team is very grateful to Ahmed El-Helw for implementing this option.

敬请反馈

This is a significant change to our ecosystem. We would appreciate your feedback to help make it even better.

Try the new memory manager on your projects and share feedback in our issue tracker, YouTrack.

自定义 Info.plist 文件

When producing a framework, the Kotlin/Native compiler generates the information property list file, `Info.plist`. Previously, it was cumbersome to customize its contents. With Kotlin 1.7.20, you can directly set the following properties:

Property	Binary option
`CFBundleIdentifier`	`bundleId`
`CFBundleShortVersionString`	`bundleShortVersionString`
`CFBundleVersion`	`bundleVersion`

To do that, use the corresponding binary option. Pass the `-Xbinary=$option=$value` compiler flag or set the `binaryOption(option, value)` Gradle DSL for the necessary framework.

The Kotlin team is very grateful to Mads Ager for implementing this feature.

Kotlin/JS

Kotlin/JS has received some enhancements that improve the developer experience and boost performance:

- Klib generation is faster in both incremental and clean builds, thanks to efficiency improvements for the loading of dependencies.
- Incremental compilation for development binaries has been reworked, resulting in major improvements in clean build scenarios, faster incremental builds, and stability fixes.
- We've improved `.d.ts` generation for nested objects, sealed classes, and optional parameters in constructors.

Gradle

The updates for the Kotlin Gradle plugin are focused on compatibility with the new Gradle features and the latest Gradle versions.

Kotlin 1.7.20 contains changes to support Gradle 7.1. Deprecated methods and properties were removed or replaced, reducing the number of deprecation warnings produced by the Kotlin Gradle plugin and unblocking future support for Gradle 8.0.

There are, however, some potentially breaking changes that may need your attention:

目标配置

- `org.jetbrains.kotlin.gradle.dsl.SingleTargetExtension` now has a generic parameter, `SingleTargetExtension<T : KotlinTarget>`.
- The `kotlin.targets.fromPreset()` convention has been deprecated. Instead, you can still use `kotlin.targets { fromPreset() }`, but we recommend setting up targets explicitly.
- Target accessors auto-generated by Gradle are no longer available inside the `kotlin.targets { }` block. Please use the `findByName("targetName")` method instead.

  Note that such accessors are still available in the case of `kotlin.targets`, for example, `kotlin.targets.linuxX64`.

源代码目录配置

The Kotlin Gradle plugin now adds Kotlin `SourceDirectorySet` as a `kotlin` extension to Java's `SourceSet` group. This makes it possible to configure source directories in the `build.gradle.kts` file similarly to how they are configured in Java, Groovy, and Scala:

```
sourceSets {
    main {
        kotlin {
            java.setSrcDirs(listOf("src/java"))
            kotlin.setSrcDirs(listOf("src/kotlin"))
        }
    }
}
```

You no longer need to use a deprecated Gradle convention and specify the source directories for Kotlin.

Remember that you can also use the `kotlin` extension to access `KotlinSourceSet`:

```
kotlin {
    sourceSets {
        main {
        // ...
        }
    }
}
```

用于 JVM 工具链配置的新方法

This release provides a new `jvmToolchain()` method for enabling the JVM toolchain feature. If you don't need any additional configuration fields, such as `implementation` or `vendor`, you can use this method from the Kotlin extension:

```
kotlin {
    jvmToolchain(17)
}
```

This simplifies the Kotlin project setup process without any additional configuration. Before this release, you could specify the JDK version only in the following way:

```
kotlin {
    jvmToolchain {
        languageVersion.set(JavaLanguageVersion.of(17))
    }
}
```

标准库

Kotlin 1.7.20 offers new extension functions for the `java.nio.file.Path` class, which allows you to walk through a file tree:

- `walk()` lazily traverses the file tree rooted at the specified path.
- `fileVisitor()` makes it possible to create a `FileVisitor` separately. `FileVisitor` defines actions on directories and files when traversing them.
- `visitFileTree(fileVisitor: FileVisitor, ...)` consumes a ready `FileVisitor` and uses `java.nio.file.Files.walkFileTree()` under the hood.
- `visitFileTree(..., builderAction: FileVisitorBuilder.() -> Unit)` creates a `FileVisitor` with the `builderAction` and calls the `visitFileTree(fileVisitor, ...)` function.
- `FileVisitResult`, return type of `FileVisitor`, has the `CONTINUE` default value that continues the processing of the file.

> The new extension functions for `java.nio.file.Path` are Experimental. They may be changed at any time. Opt-in is required (see details below), and you should use them only for evaluation purposes.
>
> 

Here are some things you can do with these new extension functions:

- Explicitly create a `FileVisitor` and then use:

```
val cleanVisitor = fileVisitor {
    onPreVisitDirectory { directory, attributes ->
        // Some logic on visiting directories
        FileVisitResult.CONTINUE
    }

    onVisitFile { file, attributes ->
        // Some logic on visiting files
        FileVisitResult.CONTINUE
    }
}

// Some logic may go here

projectDirectory.visitFileTree(cleanVisitor)
```

- Create a `FileVisitor` with the `builderAction` and use it immediately:

```
projectDirectory.visitFileTree {
// Definition of the builderAction:
```

```
    onPreVisitDirectory { directory, attributes ->
        // Some logic on visiting directories
        FileVisitResult.CONTINUE
    }

    onVisitFile { file, attributes ->
        // Some logic on visiting files
        FileVisitResult.CONTINUE
    }
}
```

- Traverse a file tree rooted at the specified path with the `walk()` function:

  ```kotlin @OptIn(kotlin.io.path.ExperimentalPathApi::class) fun traverseFileTree() {

```
val cleanVisitor = fileVisitor {
    onPreVisitDirectory { directory, _ ->
        if (directory.name == "build") {
            directory.toFile().deleteRecursively()
            FileVisitResult.SKIP_SUBTREE
        } else {
            FileVisitResult.CONTINUE
        }
    }

    onVisitFile { file, _ ->
        if (file.extension == "class") {
            file.deleteExisting()
        }
        FileVisitResult.CONTINUE
    }
}

val rootDirectory = createTempDirectory("Project")

rootDirectory.resolve("src").let { srcDirectory ->
    srcDirectory.createDirectory()
    srcDirectory.resolve("A.kt").createFile()
    srcDirectory.resolve("A.class").createFile()
}

rootDirectory.resolve("build").let { buildDirectory ->
    buildDirectory.createDirectory()
    buildDirectory.resolve("Project.jar").createFile()
}
```

// Use walk function: val directoryStructure = rootDirectory.walk(PathWalkOption.INCLUDE_DIRECTORIES) .map { it.relativeTo(rootDirectory).toString() } .toList().sorted() assertPrints(directoryStructure, "[, build, build/Project.jar, src, src/A.class, src/A.kt]")
```

```
    rootDirectory.visitFileTree(cleanVisitor)

    val directoryStructureAfterClean = rootDirectory.walk(PathWalkOption.INCLUDE_DIRECTO
RIES)
        .map { it.relativeTo(rootDirectory).toString() }
        .toList().sorted()
    assertPrints(directoryStructureAfterClean, "[, src, src/A.kt]")
```

//sampleEnd } ```

As is usual for an experimental API, the new extensions require an opt-in: `@OptIn(kotlin.io.path.ExperimentalPathApi::class)` or `@kotlin.io.path.ExperimentalPathApi`. Alternatively, you can use a compiler option: `-opt-in=kotlin.io.path.ExperimentalPathApi`.

We would appreciate your feedback on the `walk()` function and the visit extension functions in YouTrack.

文档更新

Since the previous release, the Kotlin documentation has received some notable changes:

修订与改进的页面

- 基本类型概述 – learn about the basic types used in Kotlin: numbers, Booleans, characters, strings, arrays, and unsigned integer numbers.
- 用于 Kotlin 开发的 IDE – see the list of IDEs with official Kotlin support and tools that have community-supported plugins.

Kotlin Multiplatform 期刊中的新增文章

- 原生与跨平台应用开发：如何选择？ – check out our overview and advantages of cross-platform app development and the native approach.
- 六大最佳跨平台应用开发框架 – read about the key aspects to help you choose the right framework for your cross-platform project.

新增与更新的教程

- 多平台入门 – learn about cross-platform mobile development with Kotlin and create an app that works on both Android and iOS.
- 使用 React 与 Kotlin/JS 构建 web 应用程序 – create a browser app exploring Kotlin's DSLs and features of a typical React program.

版本发布文档中的变更

We no longer provide a list of recommended kotlinx libraries for each release. This list included only the versions recommended and tested with Kotlin itself. It didn't take into account that some libraries depend on each other and require a special kotlinx version, which may differ from the recommended Kotlin version.

We're working on finding a way to provide information on how libraries interrelate and depend on each other so that it will be clear which kotlinx library version you should use when you upgrade the Kotlin version in your project.

安装 Kotlin 1.7.20

IntelliJ IDEA 2021.3, 2022.1, and 2022.2 automatically suggest updating the Kotlin plugin to 1.7.20.

> For Android Studio Dolphin (213), Electric Eel (221), and Flamingo (222), the Kotlin plugin 1.7.20 will be delivered with upcoming Android Studios updates.
>
> 

The new command-line compiler is available for download on the GitHub release page.

Kotlin 1.7.20 的兼容性指南

Although Kotlin 1.7.20 is an incremental release, there are still incompatible changes we had to make to limit spread of the issues introduced in Kotlin 1.7.0.

Find the detailed list of such changes in the Compatibility guide for Kotlin 1.7.20.

Kotlin 1.7.0 的新特性

The IDE support for Kotlin 1.7.0 is available for IntelliJ IDEA 2021.2, 2021.3, and 2022.1.

发布于：*2022-07-09*

Kotlin 1.7.0 has been released. It unveils the Alpha version of the new Kotlin/JVM K2 compiler, stabilizes language features, and brings performance improvements for the JVM, JS, and Native platforms.

Here is a list of the major updates in this version:

- The new Kotlin K2 compiler is in Alpha now, and it offers serious performance improvements. It is available only for the JVM, and none of the compiler plugins, including kapt, work with it.
- A new approach to the incremental compilation in Gradle. Incremental compilation is now also supported for changes made inside dependent non-Kotlin modules and is compatible with Gradle.
- We've stabilized opt-in requirement annotations, definitely non-nullable types, and builder inference.
- There's now an underscore operator for type args. You can use it to automatically infer a type of argument when other types are specified.
- This release allows implementation by delegation to an inlined value of an inline class. You can now create lightweight wrappers that do not allocate memory in most cases.

You can also find a short overview of the changes in this video:

YouTube 视频：What's new in Kotlin 1.7.0

用于 JVM 的新版 Kotlin K2 编译器进入 Alpha 阶段

This Kotlin release introduces the **Alpha** version of the new Kotlin K2 compiler. The new compiler aims to speed up the development of new language features, unify all of the platforms Kotlin supports, bring performance improvements, and provide an API for compiler extensions.

We've already published some detailed explanations of our new compiler and its benefits:

- The Road to the New Kotlin Compiler
- K2 Compiler: a Top-Down View

It's important to point out that with the Alpha version of the new K2 compiler we were primarily focused on performance improvements, and it only works with JVM projects. It doesn't support Kotlin/JS, Kotlin/Native, or other multi-platform projects, and none of compiler plugins, including kapt, work with it.

Our benchmarks show some outstanding results on our internal projects:

Project	Current Kotlin compiler performance	New K2 Kotlin compiler performance	Performance boost
Kotlin	2.2 KLOC/s	4.8 KLOC/s	~ x2.2
YouTrack	1.8 KLOC/s	4.2 KLOC/s	~ x2.3
IntelliJ IDEA	1.8 KLOC/s	3.9 KLOC/s	~ x2.2
Space	1.2 KLOC/s	2.8 KLOC/s	~ x2.3

> The KLOC/s performance numbers stand for the number of thousands of lines of code that the compiler processes per second.
>
> 

You can check out the performance boost on your JVM projects and compare it with the results of the old compiler. To enable the Kotlin K2 compiler, use the following compiler option:

```
-Xuse-k2
```

Also, the K2 compiler includes a number of bugfixes. Please note that even issues with **State: Open** from this list are in fact fixed in K2.

The next Kotlin releases will improve the stability of the K2 compiler and provide more features, so stay tuned!

If you face any performance issues with the Kotlin K2 compiler, please report them to our issue tracker.

语言

Kotlin 1.7.0 introduces support for implementation by delegation and a new underscore operator for type arguments. It also stabilizes several language features introduced as previews in previous releases:

- 用委托实现内联类的内联值
- 下划线操作符用于类型参数
- 稳定版构建器推断
- 稳定版选择加入要求
- 稳定版绝对不可空类型

内联类的内联值可以通过委托来实现

If you want to create a lightweight wrapper for a value or class instance, it's necessary to implement all interface methods by hand. Implementation by delegation solves this issue, but it did not work with inline classes before 1.7.0. This restriction has been removed, so you can now create lightweight wrappers that do not allocate memory in most cases.

```
interface Bar {
    fun foo() = "foo"
}

@JvmInline
value class BarWrapper(val bar: Bar): Bar by bar

fun main() {
    val bw = BarWrapper(object: Bar {})
    println(bw.foo())
}
```

下划线操作符用于类型参数

Kotlin 1.7.0 introduces an underscore operator, `_`, for type arguments. You can use it to automatically infer a type argument when other types are specified:

```
abstract class SomeClass<T> {
    abstract fun execute(): T
}

class SomeImplementation : SomeClass<String>() {
    override fun execute(): String = "Test"
}

class OtherImplementation : SomeClass<Int>() {
    override fun execute(): Int = 42
}
```

```
object Runner {
    inline fun <reified S: SomeClass<T>, T> run(): T {
        return S::class.java.getDeclaredConstructor().newInstance().execute()
    }
}

fun main() {
    // T is inferred as String because SomeImplementation derives from SomeClass<Strin
g>
    val s = Runner.run<SomeImplementation, _>()
    assert(s == "Test")

    // T is inferred as Int because OtherImplementation derives from SomeClass<Int>
    val n = Runner.run<OtherImplementation, _>()
    assert(n == 42)
}
```

> You can use the underscore operator in any position in the variables list to infer a type argument.
>
> 

稳定版构建器推断

Builder inference is a special kind of type inference that is useful when calling generic builder functions. It helps the compiler infer the type arguments of a call using the type information about other calls inside its lambda argument.

Starting with 1.7.0, builder inference is automatically activated if a regular type inference cannot get enough information about a type without specifying the `-Xenable-builder-inference` compiler option, which was introduced in 1.6.0.

Learn how to write custom generic builders.

稳定版选择加入要求

Opt-in requirements are now Stable and do not require additional compiler configuration.

Before 1.7.0, the opt-in feature itself required the argument `-opt-in=kotlin.RequiresOptIn` to avoid a warning. It no longer requires this; however, you can still use the compiler argument `-opt-in` to opt-in for other annotations, a module.

稳定版绝对不可空类型

In Kotlin 1.7.0, definitely non-nullable types have been promoted to Stable. They provide better interoperability when extending generic Java classes and interfaces.

You can mark a generic type parameter as definitely non-nullable at the use site with the new syntax `T & Any`. The syntactic form comes from the notation for intersection types and is now limited to a type parameter with nullable upper bounds on the left side of `&` and a non-nullable `Any` on the right side:

```
fun <T> elvisLike(x: T, y: T & Any): T & Any = x ?: y

fun main() {
    // OK
    elvisLike<String>("", "").length
    // Error: 'null' cannot be a value of a non-null type
    elvisLike<String>("", null).length

    // OK
    elvisLike<String?>(null, "").length
    // Error: 'null' cannot be a value of a non-null type
    elvisLike<String?>(null, null).length
}
```

Learn more about definitely non-nullable types in this KEEP.

Kotlin/JVM

This release brings performance improvements for the Kotlin/JVM compiler and a new compiler option. Additionally, callable references to functional interface constructors have become Stable. Note that since 1.7.0, the default target version for Kotlin/JVM compilations is `1.8`.

- 编译器性能优化
- 新增编译器选项 `-Xjdk-release`
- 稳定版函数式接口构造函数的可调用引用
- 删除了 JVM 目标平台版本 1.6

编译器性能优化

Kotlin 1.7.0 introduces performance improvements for the Kotlin/JVM compiler. According to our benchmarks, compilation time has been reduced by 10% on average compared to Kotlin 1.6.0. Projects with lots of usages of inline functions, for example, projects using `kotlinx.html`, will compile faster thanks to the improvements to the bytecode postprocessing.

新增编译器选项：-Xjdk-release

Kotlin 1.7.0 presents a new compiler option, `-Xjdk-release`. This option is similar to the javac's command-line `--release` option. The `-Xjdk-release` option controls the target bytecode version and limits the API of the JDK in the classpath to the specified Java version. For example, `kotlinc -Xjdk-release=1.8` won't allow referencing `java.lang.Module` even if the JDK in the dependencies is version 9 or higher.

> This option is not guaranteed to be effective for each JDK distribution.
>
> 

Please leave your feedback on this YouTrack ticket.

稳定版函数式接口构造函数的可调用引用

Callable references to functional interface constructors are now Stable. Learn how to migrate from an interface with a constructor function to a functional interface using callable references.

Please report any issues you find in YouTrack.

删除了 JVM 目标平台版本 1.6

The default target version for Kotlin/JVM compilations is `1.8`. The `1.6` target has been removed.

Please migrate to JVM target 1.8 or above. Learn how to update the JVM target version for:

- Gradle
- Maven
- The command-line compiler

Kotlin/Native

Kotlin 1.7.0 includes changes to Objective-C and Swift interoperability and stabilizes features that were introduced in previous releases. It also brings performance improvements for the new memory manager along with other updates:

- 新版内存管理器的性能改进
- 与 JVM 及 JS IR 后端统一编译器插件 ABI
- 支持独立的 Android 可执行文件
- 与 Swift async/await 互操作：返回 `Void` 而不是 `KotlinUnit`
- 通过 Objective-C 桥接禁止未声明的异常
- 改进了 CocoaPods 集成
- 覆盖了 Kotlin/Native 编译器下载 URL

新版内存管理器的性能改进

> The new Kotlin/Native memory manager is in Alpha. It may change incompatibly and require manual migration in the future. We would appreciate your feedback in YouTrack.
>
> 

The new memory manager is still in Alpha, but it is on its way to becoming Stable. This release delivers significant performance improvements for the new memory manager, especially in garbage collection (GC). In particular, concurrent implementation of the sweep phase, introduced in 1.6.20, is now enabled by default. This helps reduce the time the application is paused for GC. The new GC scheduler is better at choosing the GC frequency, especially for larger heaps.

Also, we've specifically optimized debug binaries, ensuring that the proper optimization level and link-time optimizations are used in the implementation code of the memory manager. This helped us improve execution time by roughly 30% for debug binaries on our benchmarks.

Try using the new memory manager in your projects to see how it works, and share your feedback with us in YouTrack.

与 JVM 及 JS IR 后端统一编译器插件 ABI

Starting with Kotlin 1.7.0, the Kotlin Multiplatform Gradle plugin uses the embeddable compiler jar for Kotlin/Native by default. This feature was announced in 1.6.0 as Experimental, and now it's Stable and ready to use.

This improvement is very handy for library authors, as it improves the compiler plugin development experience. Before this release, you had to provide separate artifacts for Kotlin/Native, but now you can use the same compiler plugin artifacts for Native and other supported platforms.

> This feature might require plugin developers to take migration steps for their existing plugins.
>
> Learn how to prepare your plugin for the update in this YouTrack issue.
>
> 

支持独立的 Android 可执行文件

Kotlin 1.7.0 provides full support for generating standard executables for Android Native targets. It was introduced in 1.6.20, and now it's enabled by default.

If you want to roll back to the previous behavior when Kotlin/Native generated shared libraries, use the following setting:

```
binaryOptions["androidProgramType"] = "nativeActivity"
```

与 Swift async/await 互操作：返回 Void 而不是 KotlinUnit

Kotlin `suspend` functions now return the `Void` type instead of `KotlinUnit` in Swift. This is the result of the improved interop with Swift's `async` / `await`. This feature was introduced in 1.6.20, and this release enables this behavior by default.

You don't need to use the `kotlin.native.binary.unitSuspendFunctionObjCExport=proper` property anymore to return the proper type for such functions.

通过 Objective-C 桥接禁止未声明的异常

When you call Kotlin code from Swift/Objective-C code (or vice versa) and this code throws an exception, it should be handled by the code where the exception occurred, unless you specifically allowed the forwarding of exceptions between languages with proper conversion (for example, using the `@Throws` annotation).

Previously, Kotlin had another unintended behavior where undeclared exceptions could "leak" from one language to another in some cases. Kotlin 1.7.0 fixes that issue, and now such cases lead to program termination.

So, for example, if you have a `{ throw Exception() }` lambda in Kotlin and call it from Swift, in Kotlin 1.7.0 it will terminate as soon as the exception reaches the Swift code. In previous Kotlin versions, such an exception could leak to the Swift code.

The `@Throws` annotation continues to work as before.

改进了 CocoaPods 集成

Starting with Kotlin 1.7.0, you no longer need to install the `cocoapods-generate` plugin if you want to integrate CocoaPods in your projects.

Previously, you needed to install both the CocoaPods dependency manager and the `cocoapods-generate` plugin to use CocoaPods, for example, to handle iOS dependencies in Kotlin Multiplatform Mobile projects.

Now setting up the CocoaPods integration is easier, and we've resolved the issue when `cocoapods-generate` couldn't be installed on Ruby 3 and later. Now the newest Ruby versions that work better on Apple M1 are also supported.

See how to set up the initial CocoaPods integration.

覆盖了 Kotlin/Native 编译器下载 URL

Starting with Kotlin 1.7.0, you can customize the download URL for the Kotlin/Native compiler. This is useful when external links on the CI are forbidden.

To override the default base URL `https://download.jetbrains.com/kotlin/native/builds`, use the following Gradle property:

```
kotlin.native.distribution.baseDownloadUrl=https://example.com
```

> The downloader will append the native version and target OS to this base URL to ensure it downloads the actual compiler distribution.
>
> 

Kotlin/JS

Kotlin/JS is receiving further improvements to the JS IR compiler backend along with other updates that can make your development experience better:

- 新版 IR 后端的性能改进
- 使用 IR 时的成员名缩短
- 通过 IR 后端的 polyfill 支持旧版浏览器
- 从 js 表达式动态加载 JavaScript 模块
- 为 JavaScript 测试运行器指定环境变量

新版 IR 后端的性能改进

This release has some major updates that should improve your development experience:

- Incremental compilation performance of Kotlin/JS has been significantly improved. It takes less time to build your JS projects. Incremental rebuilds should now be roughly on par with the legacy backend in many cases now.
- The Kotlin/JS final bundle requires less space, as we have significantly reduced the size of the final artifacts. We've measured up to a 20% reduction in the production bundle size compared to the legacy backend for some large projects.
- Type checking for interfaces has been improved by orders of magnitude.
- Kotlin generates higher-quality JS code

使用 IR 时的成员名缩短

The Kotlin/JS IR compiler now uses its internal information about the relationships of your Kotlin classes and functions to apply more efficient minification, shortening the names of functions, properties, and classes. This shrinks the resulting bundled applications.

This type of minification is automatically applied when you build your Kotlin/JS application in production mode and is enabled by default. To disable member name minification, use the `-Xir-minimized-member-names` compiler flag:

```
kotlin {
    js(IR) {
        compilations.all {
            compileKotlinTask.kotlinOptions.freeCompilerArgs += listOf("-Xir-minimized
-member-names=false")
        }
    }
}
```

通过 IR 后端的 polyfill 支持旧版浏览器

The IR compiler backend for Kotlin/JS now includes the same polyfills as the legacy backend. This allows code compiled with the new compiler to run in older browsers that do not support all the methods from ES2015 used by the Kotlin standard library. Only those polyfills actually used by the project are included in the final bundle, which minimizes their potential impact on the bundle size.

This feature is enabled by default when using the IR compiler, and you don't need to configure it.

从 js 表达式动态加载 JavaScript 模块

When working with the JavaScript modules, most applications use static imports, whose use is covered with the JavaScript module integration. However, Kotlin/JS was missing a mechanism to load JavaScript modules dynamically at runtime in your applications.

Starting with Kotlin 1.7.0, the `import` statement from JavaScript is supported in `js` blocks, allowing you to dynamically bring packages into your application at runtime:

```
val myPackage = js("import('my-package')")
```

为 JavaScript 测试运行器指定环境变量

To tune Node.js package resolution or pass external information to Node.js tests, you can now specify environment variables used by the JavaScript test runners. To define an environment variable, use the `environment()` function with a key-value pair inside the `testTask` block in your build script:

```
kotlin {
    js {
        nodejs {
            testTask {
                environment("key", "value")
            }
        }
    }
}
```

标准库

In Kotlin 1.7.0, the standard library has received a range of changes and improvements. They introduce new features, stabilize experimental ones, and unify support for named capturing groups for Native, JS, and the JVM:

- 集合函数 min() 与 max() 返回不可空值
- 指定索引位置的正则表达式匹配
- 对以前语言与 API 版本的扩展支持
- 通过反射访问注解
- 稳定版深递归函数
- 默认时间源的时间标记基于内联类
- Java Optional 的新实验性扩展函数
- 在 JS 与 Native 平台支持命名捕获

集合函数 min() 与 max() 返回不可空值

In Kotlin 1.4.0, we renamed the `min()` and `max()` collection functions to `minOrNull()` and `maxOrNull()`. These new names better reflect their behavior – returning null if the receiver collection is empty. It also helped align the functions' behavior with naming conventions used throughout the Kotlin collections API.

The same was true of `minBy()`, `maxBy()`, `minWith()`, and `maxWith()`, which all got their *OrNull() synonyms in Kotlin 1.4.0. Older functions affected by this change were gradually deprecated.

Kotlin 1.7.0 reintroduces the original function names, but with a non-nullable return type. The new `min()`, `max()`, `minBy()`, `maxBy()`, `minWith()`, and `maxWith()` functions now strictly return the collection element or throw an exception.

```
fun main() {
    val numbers = listOf<Int>()
    println(numbers.maxOrNull()) // "null"
```

```
    println(numbers.max()) // "Exception in... Collection is empty."
}
```

指定索引位置的正则表达式匹配

The `Regex.matchAt()` and `Regex.matchesAt()` functions, introduced in 1.5.30, are now Stable. They provide a way to check whether a regular expression has an exact match at a particular position in a `String` or `CharSequence`.

`matchesAt()` checks for a match and returns a boolean result:

```
fun main() {
    val releaseText = "Kotlin 1.7.0 is on its way!"
    // regular expression: one digit, dot, one digit, dot, one or more digits
    val versionRegex = "\\d[.]\\d[.]\\d+".toRegex()

    println(versionRegex.matchesAt(releaseText, 0)) // "false"
    println(versionRegex.matchesAt(releaseText, 7)) // "true"
}
```

`matchAt()` returns the match if it's found, or `null` if it isn't:

```
fun main() {
    val releaseText = "Kotlin 1.7.0 is on its way!"
    val versionRegex = "\\d[.]\\d[.]\\d+".toRegex()

    println(versionRegex.matchAt(releaseText, 0)) // "null"
    println(versionRegex.matchAt(releaseText, 7)?.value) // "1.7.0"
}
```

We'd be grateful for your feedback on this YouTrack issue.

对以前语言与 API 版本的扩展支持

To support library authors developing libraries that are meant to be consumable in a wide range of previous Kotlin versions, and to address the increased frequency of major Kotlin releases, we have extended our support for previous language and API versions.

With Kotlin 1.7.0, we're supporting three previous language and API versions rather than two. This means Kotlin 1.7.0 supports the development of libraries targeting Kotlin versions down to 1.4.0. For more information on backward compatibility, see Compatibility modes.

通过反射访问注解

The `KAnnotatedElement.findAnnotations()` extension function, which was first introduced in 1.6.0, is now Stable. This reflection function returns all annotations of a given type on an element, including individually applied and repeated annotations.

```
@Repeatable
annotation class Tag(val name: String)

@Tag("First Tag")
@Tag("Second Tag")
fun taggedFunction() {
    println("I'm a tagged function!")
}

fun main() {
    val x = ::taggedFunction
    val foo = x as KAnnotatedElement
    println(foo.findAnnotations<Tag>()) // [@Tag(name=First Tag), @Tag(name=Second Tag
)]
}
```

稳定版深递归函数

Deep recursive functions have been available as an experimental feature since Kotlin 1.4.0, and they are now Stable in Kotlin 1.7.0. Using `DeepRecursiveFunction`, you can define a function that keeps its stack on the heap instead of using the actual call stack. This allows you to run very deep recursive computations. To call a deep recursive function, `invoke` it.

In this example, a deep recursive function is used to calculate the depth of a binary tree recursively. Even though this sample function calls itself recursively 100,000 times, no `StackOverflowError` is thrown:

```
class Tree(val left: Tree?, val right: Tree?)

val calculateDepth = DeepRecursiveFunction<Tree?, Int> { t ->
    if (t == null) 0 else maxOf(
        callRecursive(t.left),
        callRecursive(t.right)
    ) + 1
}

fun main() {
    // Generate a tree with a depth of 100_000
    val deepTree = generateSequence(Tree(null, null)) { prev ->
        Tree(prev, null)
    }.take(100_000).last()

    println(calculateDepth(deepTree)) // 100000
}
```

Consider using deep recursive functions in your code where your recursion depth exceeds 1000 calls.

默认时间源的时间标记基于内联类

Kotlin 1.7.0 improves the performance of time measurement functionality by changing the time marks returned by `TimeSource.Monotonic` into inline value classes. This means that calling functions like `markNow()`, `elapsedNow()`, `measureTime()`, and `measureTimedValue()` doesn't allocate wrapper classes for their `TimeMark` instances. Especially when measuring a piece of code that is part of a hot path, this can help minimize the performance impact of the measurement:

```
@OptIn(ExperimentalTime::class)
fun main() {
    val mark = TimeSource.Monotonic.markNow() // Returned `TimeMark` is inline class
    val elapsedDuration = mark.elapsedNow()
}
```

> This optimization is only available if the time source from which the `TimeMark` is obtained is statically known to be `TimeSource.Monotonic`.

Java Optional 的新实验性扩展函数

Kotlin 1.7.0 comes with new convenience functions that simplify working with `Optional` classes in Java. These new functions can be used to unwrap and convert optional objects on the JVM and help make working with Java APIs more concise.

The `getOrNull()`, `getOrDefault()`, and `getOrElse()` extension functions allow you to get the value of an `Optional` if it's present. Otherwise, you get `null`, a default value, or a value returned by a function, respectively:

```
val presentOptional = Optional.of("I'm here!")

println(presentOptional.getOrNull())
// "I'm here!"

val absentOptional = Optional.empty<String>()

println(absentOptional.getOrNull())
// null
println(absentOptional.getOrDefault("Nobody here!"))
// "Nobody here!"
println(absentOptional.getOrElse {
    println("Optional was absent!")
```

```
    "Default value!"
})
// "Optional was absent!"
// "Default value!"
```

The `toList()`, `toSet()`, and `asSequence()` extension functions convert the value of a present `Optional` to a list, set, or sequence, or return an empty collection otherwise. The `toCollection()` extension function appends the `Optional` value to an already existing destination collection:

```
val presentOptional = Optional.of("I'm here!")
val absentOptional = Optional.empty<String>()
println(presentOptional.toList() + "," + absentOptional.toList())
// ["I'm here!"], []
println(presentOptional.toSet() + "," + absentOptional.toSet())
// ["I'm here!"], []
val myCollection = mutableListOf<String>()
absentOptional.toCollection(myCollection)
println(myCollection)
// []
presentOptional.toCollection(myCollection)
println(myCollection)
// ["I'm here!"]
val list = listOf(presentOptional, absentOptional).flatMap { it.asSequence() }
println(list)
// ["I'm here!"]
```

These extension functions are being introduced as Experimental in Kotlin 1.7.0. You can learn more about `Optional` extensions in this KEEP. As always, we welcome your feedback in the Kotlin issue tracker.

在 JS 与 Native 平台支持命名捕获

Starting with Kotlin 1.7.0, named capturing groups are supported not only on the JVM, but on the JS and Native platforms as well.

To give a name to a capturing group, use the (`?<name>group`) syntax in your regular expression. To get the text matched by a group, call the newly introduced `MatchGroupCollection.get()` function and pass the group name.

按名称取回匹配分组的值

Consider this example for matching city coordinates. To get a collection of groups matched by the regular expression, use `groups`. Compare retrieving a group's contents by its number (index) and by its name using `value`:

```
fun main() {
    val regex = "\\b(?<city>[A-Za-z\\s]+),\\s(?<state>[A-Z]{2}):\\s(?<areaCode>[0-9]{3
})\\b".toRegex()
    val input = "Coordinates: Austin, TX: 123"
    val match = regex.find(input)!!
    println(match.groups["city"]?.value) // "Austin" — by name
    println(match.groups[2]?.value) // "TX" — by number
}
```

命名反向引用

You can now also use group names when backreferencing groups. Backreferences match the same text that was previously matched by a capturing group. For this, use the `\k<name>` syntax in your regular expression:

```
fun backRef() {
    val regex = "(?<title>\\w+), yes \\k<title>".toRegex()
    val match = regex.find("Do you copy? Sir, yes Sir!")!!
    println(match.value) // "Sir, yes Sir"
    println(match.groups["title"]?.value) // "Sir"
}
```

替换表达式中的命名分组

Named group references can be used with replacement expressions. Consider the `replace()` function that substitutes all occurrences of the specified regular expression in the input with a replacement expression, and the `replaceFirst()` function that swaps the first match only.

Occurrences of `${name}` in the replacement string are substituted with the subsequences corresponding to the captured groups with the specified name. You can compare replacements in group references by name and index:

```
fun dateReplace() {
    val dateRegex = Regex("(?<dd>\\d{2})-(?<mm>\\d{2})-(?<yyyy>\\d{4})")
    val input = "Date of birth: 27-04-2022"
    println(dateRegex.replace(input, "\${yyyy}-\${mm}-\${dd}")) // "Date of birth: 202
2-04-27" — by name
    println(dateRegex.replace(input, "\$3-\$2-\$1")) // "Date of birth: 2022-04-27" —
by number
}
```

Gradle

This release introduces new build reports, support for Gradle plugin variants, new statistics in kapt, and a lot more:

- 增量编译的新方式
- 用于跟踪编译器性能的新版构建报告
- Gradle 与 Android Gradle 插件最低支持版本的变更
- 支持 Gradle 插件变体
- Kotlin Gradle 插件 API 的更新
- 通过插件 API 提供 sam-with-receiver 插件
- 编译任务的变更
- kapt 中每个注解处理器生成文件的新版统计信息
- 弃用 kotlin.compiler.execution.strategy 系统属性
- 删除弃用的选项、方法与插件

增量编译的新方式

> The new approach to incremental compilation is Experimental. It may be dropped or changed at any time. Opt-in is required (see the details below). We encourage you to use it only for evaluation purposes, and we would appreciate your feedback in YouTrack.
>
> 

In Kotlin 1.7.0, we've reworked incremental compilation for cross-module changes. Now incremental compilation is also supported for changes made inside dependent non-Kotlin modules, and it is compatible with the Gradle build cache. Support for compilation avoidance has also been improved.

We expect you'll see the most significant benefit of the new approach if you use the build cache or frequently make changes in non-Kotlin Gradle modules. Our tests for the Kotlin project on the `kotlin-gradle-plugin` module show an improvement of greater than 80% for the changes after the cache hit.

To try this new approach, set the following option in your `gradle.properties`:

```
kotlin.incremental.useClasspathSnapshot=true
```

> The new approach to incremental compilation is currently available for the JVM backend in the Gradle build system only.
>
> 

Learn how the new approach to incremental compilation is implemented under the hood in this blog post.

Our plan is to stabilize this technology and add support for other backends (JS, for instance) and build systems. We'd appreciate your reports in YouTrack about any issues or strange behavior you encounter in this compilation scheme. Thank you!

The Kotlin team is very grateful to Ivan Gavrilovic, Hung Nguyen, Cédric Champeau, and other external contributors for their help.

Kotlin 编译器任务的构建报告

> Kotlin build reports are Experimental. They may be dropped or changed at any time. Opt-in is required (see details below). Use them only for evaluation purposes. We appreciate your feedback on them in YouTrack.
>
> 

Kotlin 1.7.0 introduces build reports that help track compiler performance. Reports contain the durations of different compilation phases and reasons why compilation couldn't be incremental.

Build reports come in handy when you want to investigate issues with compiler tasks, for example:

- When the Gradle build takes too much time and you want to understand the root cause of the poor performance.
- When the compilation time for the same project differs, sometimes taking seconds, sometimes taking minutes.

To enable build reports, declare where to save the build report output in `gradle.properties`:

```
kotlin.build.report.output=file
```

The following values (and their combinations) are available:

- `file` saves build reports in a local file.
- `build_scan` saves build reports in the `custom values` section of the build scan.

  > The Gradle Enterprise plugin limits the number of custom values and their length. In big projects, some values could be lost.
  >
  > 

- `http` posts build reports using HTTP(S). The POST method sends metrics in the JSON format. Data may change from version to version. You can see the current version of the sent data in the Kotlin repository.

There are two common cases that analyzing build reports for long-running compilations can help you resolve:

- The build wasn't incremental. Analyze the reasons and fix underlying problems.
- The build was incremental, but took too much time. Try to reorganize source files — split big files, save separate classes in different files, refactor large classes, declare top-level functions in different files, and so on.

Learn more about new build reports in this blog post.

You are welcome to try using build reports in your infrastructure. If you have any feedback, encounter any issues, or want to suggest improvements, please don't hesitate to report them in our issue tracker. Thank you!

最低支持版本升级

Starting with Kotlin 1.7.0, the minimum supported Gradle version is 6.7.1. We had to raise the version to support Gradle plugin variants and the new Gradle API. In the future, we should not have to raise the minimum supported version as often, thanks to the Gradle plugin variants feature.

Also, the minimal supported Android Gradle plugin version is now 3.6.4.

支持 Gradle 插件变体

Gradle 7.0 introduced a new feature for Gradle plugin authors — plugins with variants. This feature makes it easier to add support for new Gradle features while maintaining compatibility for Gradle versions below 7.1. Learn more about variant selection in Gradle.

With Gradle plugin variants, we can ship different Kotlin Gradle plugin variants for different Gradle versions. The goal is to support the base Kotlin compilation in the `main` variant, which corresponds to the oldest supported versions of Gradle. Each variant will have implementations for Gradle features from a corresponding release. The latest variant will support the widest Gradle feature set. With this approach, we can extend support for older Gradle versions with limited functionality.

Currently, there are only two variants of the Kotlin Gradle plugin:

- `main` for Gradle versions 6.7.1–6.9.3
- `gradle70` for Gradle versions 7.0 and higher

In future Kotlin releases, we may add more.

To check which variant your build uses, enable the `--info` log level and find a string in the output starting with `Using Kotlin Gradle plugin`, for example, `Using Kotlin Gradle plugin main variant`.

> Here are workarounds for some known issues with variant selection in Gradle:
>
> - ResolutionStrategy in pluginManagement is not working for plugins with multivariants

> - Plugin variants are ignored when a plugin is added as the `buildSrc` common dependency
>
> 

Leave your feedback on this YouTrack ticket.

Kotlin Gradle 插件 API 的更新

The Kotlin Gradle plugin API artifact has received several improvements:

- There are new interfaces for Kotlin/JVM and Kotlin/kapt tasks with user-configurable inputs.
- There is a new `KotlinBasePlugin` interface that all Kotlin plugins inherit from. Use this interface when you want to trigger some configuration action whenever any Kotlin Gradle plugin (JVM, JS, Multiplatform, Native, and other platforms) is applied:

  ```
  project.plugins.withType<org.jetbrains.kotlin.gradle.plugin.KotlinBasePlugin>() {
      // Configure your action here
  }
  ```

  You can leave your feedback about the `KotlinBasePlugin` in this YouTrack ticket.

- We've laid the groundwork for the Android Gradle plugin to configure Kotlin compilation within itself, meaning you won't need to add the Kotlin Android Gradle plugin to your build. Follow Android Gradle Plugin release announcements to learn about the added support and try it out!

通过插件 API 提供了 sam-with-receiver 插件

The sam-with-receiver compiler plugin is now available via the Gradle plugins DSL:

```
plugins {
    id("org.jetbrains.kotlin.plugin.sam.with.receiver") version "$kotlin_version"
}
```

编译任务的变更

Compile tasks have received lots of changes in this release:

- Kotlin compile tasks no longer inherit the Gradle `AbstractCompile` task. They inherit only the `DefaultTask`.
- The `AbstractCompile` task has the `sourceCompatibility` and `targetCompatibility` inputs. Since the `AbstractCompile` task is no longer inherited, these inputs are no longer available in Kotlin users' scripts.
- The `SourceTask.stableSources` input is no longer available, and you should use the `sources`

input. `setSource(...)` methods are still available.
- All compile tasks now use the `libraries` input for a list of libraries required for compilation. The `KotlinCompile` task still has the deprecated Kotlin property `classpath`, which will be removed in future releases.
- Compile tasks still implement the `PatternFilterable` interface, which allows the filtering of Kotlin sources. The `sourceFilesExtensions` input was removed in favor of using `PatternFilterable` methods.
- The deprecated `Gradle destinationDir: File` output was replaced with the `destinationDirectory: DirectoryProperty` output.
- The Kotlin/Native `AbstractNativeCompile` task now inherits the `AbstractKotlinCompileTool` base class. This is an initial step toward integrating Kotlin/Native build tools into all the other tools.

Please leave your feedback in this YouTrack ticket.

kapt 中每个注解处理器生成文件的统计信息

The `kotlin-kapt` Gradle plugin already reports performance statistics for each processor. Starting with Kotlin 1.7.0, it can also report statistics on the number of generated files for each annotation processor.

This is useful to track if there are unused annotation processors as a part of the build. You can use the generated report to find modules that trigger unnecessary annotation processors and update the modules to prevent that.

Enable the statistics in two steps:

- Set the `showProcessorStats` flag to `true` in your `build.gradle.kts`:

```
kapt {
    showProcessorStats = true
}
```

- Set the `kapt.verbose` Gradle property to `true` in your `gradle.properties`:

```
kapt.verbose=true
```

> You can also enable verbose output via the command line option `verbose`.
>
> 

The statistics will appear in the logs with the `info` level. You'll see the `Annotation processor stats:` line followed by statistics on the execution time of each annotation processor. After these lines, there will be the `Generated files report:` line followed by statistics on the number of generated files for each annotation processor. For example:

```
[INFO] Annotation processor stats:
[INFO] org.mapstruct.ap.MappingProcessor: total: 290 ms, init: 1 ms, 3 round(s): 289 m
s, 0 ms, 0 ms
[INFO] Generated files report:
[INFO] org.mapstruct.ap.MappingProcessor: total sources: 2, sources per round: 2, 0, 0
```

Please leave your feedback in this YouTrack ticket.

弃用 kotlin.compiler.execution.strategy 系统属性

Kotlin 1.6.20 introduced new properties for defining a Kotlin compiler execution strategy. In Kotlin 1.7.0, a deprecation cycle has started for the old system property `kotlin.compiler.execution.strategy` in favor of the new properties.

When using the `kotlin.compiler.execution.strategy` system property, you'll receive a warning. This property will be deleted in future releases. To preserve the old behavior, replace the system property with the Gradle property of the same name. You can do this in `gradle.properties`, for example:

```
kotlin.compiler.execution.strategy=out-of-process
```

You can also use the compile task property `compilerExecutionStrategy`. Learn more about this on the Gradle page.

删除弃用的选项、方法与插件

删除 useExperimentalAnnotation 方法

In Kotlin 1.7.0, we completed the deprecation cycle for the `useExperimentalAnnotation` Gradle method. Use `optIn()` instead to opt in to using an API in a module.

For example, if your Gradle module is multiplatform:

```
sourceSets {
    all {
        languageSettings.optIn("org.mylibrary.OptInAnnotation")
    }
}
```

Learn more about opt-in requirements in Kotlin.

删除弃用的编译器选项

We've completed the deprecation cycle for several compiler options:

- The `kotlinOptions.jdkHome` compiler option was deprecated in 1.5.30 and has been removed in the current release. Gradle builds now fail if they contain this option. We encourage you to use Java toolchains, which have been supported since Kotlin 1.5.30.
- The deprecated `noStdlib` compiler option has also been removed. The Gradle plugin uses the `kotlin.stdlib.default.dependency=true` property to control whether the Kotlin standard library is present.

> The compiler arguments `-jdkHome` and `-no-stdlib` are still available.
>
> 

删除弃用的插件

In Kotlin 1.4.0, the `kotlin2js` and `kotlin-dce-plugin` plugins were deprecated, and they have been removed in this release. Instead of `kotlin2js`, use the new `org.jetbrains.kotlin.js` plugin. Dead code elimination (DCE) works when the Kotlin/JS Gradle plugin is properly configured.

In Kotlin 1.6.0, we changed the deprecation level of the `KotlinGradleSubplugin` class to `ERROR`. Developers used this class for writing compiler plugins. In this release, this class has been removed. Use the `KotlinCompilerPluginSupportPlugin` class instead.

> The best practice is to use Kotlin plugins with versions 1.7.0 and higher throughout your project.
>
> 

删除弃用的协程 DSL 选项与属性

We removed the deprecated `kotlin.experimental.coroutines` Gradle DSL option and the `kotlin.coroutines` property used in `gradle.properties`. Now you can just use *suspending functions* or add the `kotlinx.coroutines` dependency to your build script.

Learn more about coroutines in the Coroutines guide.

删除工具链扩展方法中的类型转换

Before Kotlin 1.7.0, you had to do the type cast into the `JavaToolchainSpec` class when configuring the Gradle toolchain with Kotlin DSL:

```
kotlin {
    jvmToolchain {
        (this as JavaToolchainSpec).languageVersion.set(JavaLanguageVersion.of(<MAJOR_
JDK_VERSION>)
    }
}
```

Now, you can omit the `(this as JavaToolchainSpec)` part:

```
kotlin {
    jvmToolchain {
        languageVersion.set(JavaLanguageVersion.of(<MAJOR_JDK_VERSION>)
    }
}
```

迁移到 Kotlin 1.7.0

安装 Kotlin 1.7.0

IntelliJ IDEA 2022.1 and Android Studio Chipmunk (212) automatically suggest updating the Kotlin plugin to 1.7.0.

> For IntelliJ IDEA 2022.2, and Android Studio Dolphin (213) or Android Studio Electric Eel (221), the Kotlin plugin 1.7.0 will be delivered with upcoming IntelliJ IDEA and Android Studios updates.
>
> 

The new command-line compiler is available for download on the GitHub release page.

使用 Kotlin 1.7.0 迁移既有项目或启动新项目

- To migrate existing projects to Kotlin 1.7.0, change the Kotlin version to `1.7.0` and reimport your Gradle or Maven project. Learn how to update to Kotlin 1.7.0.
- To start a new project with Kotlin 1.7.0, update the Kotlin plugin and run the Project Wizard from **File** | **New** | **Project**.

Kotlin 1.7.0 的兼容性指南

Kotlin 1.7.0 is a feature release and can, therefore, bring changes that are incompatible with your code written for earlier versions of the language. Find the detailed list of such changes in the Compatibility guide for Kotlin 1.7.0.

Kotlin 1.6.20 的新特性

发布于：*2022-04-04*

Kotlin 1.6.20 reveals previews of the future language features, makes the hierarchical structure the default for multiplatform projects, and brings evolutionary improvements to other components.

You can also find a short overview of the changes in this video:

YouTube 视频：What's new in Kotlin 1.6.20

语言

In Kotlin 1.6.20, you can try two new language features:

- Kotlin/JVM 的上下文接收者原型
- 绝对不可空类型

Kotlin/JVM 的上下文接收者原型

> The feature is a prototype available only for Kotlin/JVM. With `-Xcontext-receivers` enabled, the compiler will produce pre-release binaries that cannot be used in production code. Use context receivers only in your toy projects. We appreciate your feedback in YouTrack.
>
> 

With Kotlin 1.6.20, you are no longer limited to having one receiver. If you need more, you can make functions, properties, and classes context-dependent (or *contextual*) by adding context receivers to their declaration. A contextual declaration does the following:

- It requires all declared context receivers to be present in a caller's scope as implicit receivers.
- It brings declared context receivers into its body scope as implicit receivers.

```
interface LoggingContext {
```

```
    val log: Logger // This context provides a reference to a logger
}

context(LoggingContext)
fun startBusinessOperation() {
    // You can access the log property since LoggingContext is an implicit receiver
    log.info("Operation has started")
}

fun test(loggingContext: LoggingContext) {
    with(loggingContext) {
        // You need to have LoggingContext in a scope as an implicit receiver
        // to call startBusinessOperation()
        startBusinessOperation()
    }
}
```

To enable context receivers in your project, use the `-Xcontext-receivers` compiler option. You can find a detailed description of the feature and its syntax in the KEEP.

Please note that the implementation is a prototype:

- With `-Xcontext-receivers` enabled, the compiler will produce pre-release binaries that cannot be used in production code
- The IDE support for context receivers is minimal for now

Try the feature in your toy projects and share your thoughts and experience with us in this YouTrack issue. If you run into any problems, please file a new issue.

绝对不可空类型

> Definitely non-nullable types are in Beta. They are almost stable, but migration steps may be required in the future. We'll do our best to minimize any changes you have to make.
>
> 

To provide better interoperability when extending generic Java classes and interfaces, Kotlin 1.6.20 allows you to mark a generic type parameter as definitely non-nullable on the use site with the new syntax `T & Any`. The syntactic form comes from a notation of intersection types and is now limited to a type parameter with nullable upper bounds on the left side of `&` and non-nullable `Any` on the right side:

```
fun <T> elvisLike(x: T, y: T & Any): T & Any = x ?: y

fun main() {
    // OK
    elvisLike<String>("", "").length
    // Error: 'null' cannot be a value of a non-null type
```

```
    elvisLike<String>("", null).length

    // OK
    elvisLike<String?>(null, "").length
    // Error: 'null' cannot be a value of a non-null type
    elvisLike<String?>(null, null).length
}
```

Set the language version to `1.7` to enable the feature:

> 【Kotlin】

```
kotlin {
    sourceSets.all {
        languageSettings.apply {
            languageVersion = "1.7"
        }
    }
}
```

> 【Groovy】

```
kotlin {
    sourceSets.all {
        languageSettings {
            languageVersion = '1.7'
        }
    }
}
```

Learn more about definitely non-nullable types in the KEEP.

Kotlin/JVM

Kotlin 1.6.20 introduces:

- Compatibility improvements of default methods in JVM interfaces: 接口的新 `@JvmDefaultWithCompatibility` 注解 and `-Xjvm-default` 模式的兼容性变更
- 支持 JVM 后端中单个模块的并行编译
- 支持函数式接口构造函数的可调用引用

接口的新 @JvmDefaultWithCompatibility 注解

Kotlin 1.6.20 introduces the new annotation `@JvmDefaultWithCompatibility`: use it along with the `-Xjvm-default=all` compiler option to create the default method in JVM interface for any non-abstract member in any Kotlin interface.

If there are clients that use your Kotlin interfaces compiled without the `-Xjvm-default=all` option, they may be binary-incompatible with the code compiled with this option. Before Kotlin 1.6.20, to avoid this compatibility issue, the recommended approach was to use the `-Xjvm-default=all-compatibility` mode and also the `@JvmDefaultWithoutCompatibility` annotation for interfaces that didn't need this type of compatibility.

This approach had some disadvantages:

- You could easily forget to add the annotation when a new interface was added.
- Usually there are more interfaces in non-public parts than in the public API, so you end up having this annotation in many places in your code.

Now, you can use the `-Xjvm-default=all` mode and mark interfaces with the `@JvmDefaultWithCompatibility` annotation. This allows you to add this annotation to all interfaces in the public API once, and you won't need to use any annotations for new non-public code.

Leave your feedback about this new annotation in this YouTrack ticket.

-Xjvm-default 模式的兼容性变更

Kotlin 1.6.20 adds the option to compile modules in the default mode (the `-Xjvm-default=disable` compiler option) against modules compiled with the `-Xjvm-default=all` or `-Xjvm-default=all-compatibility` modes. As before, compilations will also be successful if all modules have the `-Xjvm-default=all` or `-Xjvm-default=all-compatibility` modes. You can leave your feedback in this YouTrack issue.

Kotlin 1.6.20 deprecates the `compatibility` and `enable` modes of the compiler option `-Xjvm-default`. There are changes in other modes' descriptions regarding the compatibility, but the overall logic remains the same. You can check out the updated descriptions.

For more information about default methods in the Java interop, see the interoperability documentation and this blog post.

支持 JVM 后端中单个模块的并行编译

> Support for parallel compilation of a single module in the JVM backend is Experimental. It may be dropped or changed at any time. Opt-in is required (see details below), and you should use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack.



We are continuing our work to improve the new JVM IR backend compilation time. In Kotlin 1.6.20, we added the experimental JVM IR backend mode to compile all the files in a module in parallel. Parallel compilation can reduce the total compilation time by up to 15%.

Enable the experimental parallel backend mode with the compiler option `-Xbackend-threads`. Use the following arguments for this option:

- `N` is the number of threads you want to use. It should not be greater than your number of CPU cores; otherwise, parallelization stops being effective because of switching context between threads
- `0` to use a separate thread for each CPU core

Gradle can run tasks in parallel, but this type of parallelization doesn't help a lot when a project (or a major part of a project) is just one big task from Gradle's perspective. If you have a very big monolithic module, use parallel compilation to compile more quickly. If your project consists of lots of small modules and has a build parallelized by Gradle, adding another layer of parallelization may hurt performance because of context switching.

> Parallel compilation has some constraints:
>
> - It doesn't work with kapt because kapt disables the IR backend
> - It requires more JVM heap by design. The amount of heap is proportional to the number of threads



支持函数式接口构造函数的可调用引用

> Support for callable references to functional interface constructors is Experimental. It may be dropped or changed at any time. Opt-in is required (see details below), and you should use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack.



Support for callable references to functional interface constructors adds a source-compatible way to migrate from an interface with a constructor function to a functional interface.

Consider the following code:

```
interface Printer {
    fun print()
}

fun Printer(block: () -> Unit): Printer = object : Printer { override fun print() = bl
ock() }
```

With callable references to functional interface constructors enabled, this code can be replaced with just a functional interface declaration:

```
fun interface Printer {
    fun print()
}
```

Its constructor will be created implicitly, and any code using the `::Printer` function reference will compile. For example:

```
documentsStorage.addPrinter(::Printer)
```

Preserve the binary compatibility by marking the legacy function `Printer` with the `@Deprecated` annotation with `DeprecationLevel.HIDDEN`:

```
@Deprecated(message = "Your message about the deprecation", level = DeprecationLevel.H
IDDEN)
fun Printer(...) {...}
```

Use the compiler option `-XXLanguage:+KotlinFunInterfaceConstructorReference` to enable this feature.

Kotlin/Native

Kotlin/Native 1.6.20 marks continued development of its new components. We've taken another step toward consistent experience with Kotlin on other platforms:

- 新版内存管理器的更新
- 新版内存管理器中清除阶段的并发实现
- 注解类的实例化
- 与 Swift async/await 互操作：返回 Void 而不是 KotlinUnit
- 使用 libbacktrace 实现更佳的堆栈跟踪
- 支持独立的 Android 可执行文件
- 性能提升
- 改进了 cinterop 模块导入过程中的错误处理
- 对 Xcode 13 库的支持

新版内存管理器的更新

> The new Kotlin/Native memory manager is in Alpha. It may change incompatibly and require manual migration in the future. We would appreciate your feedback on it in YouTrack.
>
> 

With Kotlin 1.6.20, you can try the Alpha version of the new Kotlin/Native memory manager. It eliminates the differences between the JVM and Native platforms to provide a consistent developer experience in multiplatform projects. For example, you'll have a much easier time creating new cross-platform mobile applications that work on both Android and iOS.

The new Kotlin/Native memory manager lifts restrictions on object-sharing between threads. It also provides leak-free concurrent programming primitives that are safe and don't require any special management or annotations.

The new memory manager will become the default in future versions, so we encourage you to try it now. Check out our blog post to learn more about the new memory manager and explore demo projects, or jump right to the migration instructions to try it yourself.

Try using the new memory manager on your projects to see how it works and share feedback in our issue tracker, YouTrack.

新版内存管理器中清除阶段的并发实现

If you have already switched to our new memory manager, which was announced in Kotlin 1.6, you might notice a huge execution time improvement: our benchmarks show 35% improvement on average. Starting with 1.6.20, there is also a concurrent implementation for the sweep phase available for the new memory manager. This should also improve the performance and decrease the duration of garbage collector pauses.

To enable the feature for the new Kotlin/Native memory manager, pass the following compiler option:

```
-Xgc=cms
```

Feel free to share your feedback on the new memory manager performance in this YouTrack issue.

注解类的实例化

In Kotlin 1.6.0, instantiation of annotation classes became Stable for Kotlin/JVM and Kotlin/JS. The 1.6.20 version delivers support for Kotlin/Native.

Learn more about 注解类的实例化.

与 Swift async/await 互操作：返回 Void 而不是 KotlinUnit

> Concurrency interoperability with Swift async/await is Experimental. It may be dropped or changed at any time. You should use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack.
>
> 

We've continued working on the experimental interop with Swift's async/await (available since Swift 5.5). Kotlin 1.6.20 differs from previous versions in the way it works with `suspend` functions with the `Unit` return type.

Previously, such functions were presented in Swift as `async` functions returning `KotlinUnit`. However, the proper return type for them is `Void`, similar to non-suspending functions.

To avoid breaking the existing code, we're introducing a Gradle property that makes the compiler translate `Unit`-returning suspend functions to `async` Swift with the `Void` return type:

```
# gradle.properties
kotlin.native.binary.unitSuspendFunctionObjCExport=proper
```

We plan to make this behavior the default in future Kotlin releases.

使用 libbacktrace 实现更佳的堆栈跟踪

> Using libbacktrace for resolving source locations is Experimental. It may be dropped or changed at any time. You should use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack.
>
> 

Kotlin/Native is now able to produce detailed stack traces with file locations and line numbers for better debugging of `linux*` (except `linuxMips32` and `linuxMipsel32`) and `androidNative*` targets.

This feature uses the libbacktrace library under the hood. Take a look at the following code to see an example of the difference:

```
fun main() = bar()
fun bar() = baz()
inline fun baz() {
    error("")
}
```

- **Before 1.6.20:**

```
Uncaught Kotlin exception: kotlin.IllegalStateException:
```

```
   at 0   example.kexe        0x227190       kfun:kotlin.Throwable#<init>(kotlin.Strin
g?){} + 96
   at 1   example.kexe        0x221e4c       kfun:kotlin.Exception#<init>(kotlin.Strin
g?){} + 92
   at 2   example.kexe        0x221f4c       kfun:kotlin.RuntimeException#<init>(kotli
n.String?){} + 92
   at 3   example.kexe        0x22234c       kfun:kotlin.IllegalStateException#<init>(
kotlin.String?){} + 92
   at 4   example.kexe        0x25d708       kfun:#bar(){} + 104
   at 5   example.kexe        0x25d68c       kfun:#main(){} + 12
```

- **1.6.20 with libbacktrace:**

```
Uncaught Kotlin exception: kotlin.IllegalStateException:
   at 0   example.kexe        0x229550    kfun:kotlin.Throwable#<init>(kotlin.String?)
{} + 96 (/opt/buildAgent/work/c3a91df21e46e2c8/kotlin/kotlin-native/runtime/src/main/k
otlin/kotlin/Throwable.kt:24:37)
   at 1   example.kexe        0x22420c    kfun:kotlin.Exception#<init>(kotlin.String?)
{} + 92 (/opt/buildAgent/work/c3a91df21e46e2c8/kotlin/kotlin-native/runtime/src/main/k
otlin/kotlin/Exceptions.kt:23:44)
   at 2   example.kexe        0x22430c    kfun:kotlin.RuntimeException#<init>(kotlin.S
tring?){} + 92 (/opt/buildAgent/work/c3a91df21e46e2c8/kotlin/kotlin-native/runtime/src
/main/kotlin/kotlin/Exceptions.kt:34:44)
   at 3   example.kexe        0x22470c    kfun:kotlin.IllegalStateException#<init>(kot
lin.String?){} + 92 (/opt/buildAgent/work/c3a91df21e46e2c8/kotlin/kotlin-native/runtim
e/src/main/kotlin/kotlin/Exceptions.kt:70:44)
   at 4   example.kexe        0x25fac8    kfun:#bar(){} + 104 [inlined] (/opt/buildAge
nt/work/c3a91df21e46e2c8/kotlin/libraries/stdlib/src/kotlin/util/Preconditions.kt:143:
56)
   at 5   example.kexe        0x25fac8    kfun:#bar(){} + 104 [inlined] (/private/tmp/
backtrace/src/commonMain/kotlin/app.kt:4:5)
   at 6   example.kexe        0x25fac8    kfun:#bar(){} + 104 (/private/tmp/backtrace/
src/commonMain/kotlin/app.kt:2:13)
   at 7   example.kexe        0x25fa4c    kfun:#main(){} + 12 (/private/tmp/backtrace/
src/commonMain/kotlin/app.kt:1:14)
```

On Apple targets, which already had file locations and line numbers in stack traces, libbacktrace provides more details for inline function calls:

- **Before 1.6.20:**

```
Uncaught Kotlin exception: kotlin.IllegalStateException:
   at 0   example.kexe    0x10a85a8f8    kfun:kotlin.Throwable#<init>(kotlin.String?){
} + 88 (/opt/buildAgent/work/c3a91df21e46e2c8/kotlin/kotlin-native/runtime/src/main/ko
tlin/kotlin/Throwable.kt:24:37)
   at 1   example.kexe    0x10a855846    kfun:kotlin.Exception#<init>(kotlin.String?){
} + 86 (/opt/buildAgent/work/c3a91df21e46e2c8/kotlin/kotlin-native/runtime/src/main/ko
tlin/kotlin/Exceptions.kt:23:44)
   at 2   example.kexe    0x10a855936    kfun:kotlin.RuntimeException#<init>(kotlin.St
ring?){} + 86 (/opt/buildAgent/work/c3a91df21e46e2c8/kotlin/kotlin-native/runtime/src/
```

```
main/kotlin/kotlin/Exceptions.kt:34:44)
   at 3   example.kexe    0x10a855c86    kfun:kotlin.IllegalStateException#<init>(kotl
in.String?){} + 86 (/opt/buildAgent/work/c3a91df21e46e2c8/kotlin/kotlin-native/runtime
/src/main/kotlin/kotlin/Exceptions.kt:70:44)
   at 4   example.kexe    0x10a8489a5    kfun:#bar(){} + 117 (/private/tmp/backtrace/s
rc/commonMain/kotlin/app.kt:2:1)
   at 5   example.kexe    0x10a84891c    kfun:#main(){} + 12 (/private/tmp/backtrace/s
rc/commonMain/kotlin/app.kt:1:14)
...
```

- **1.6.20 with libbacktrace:**

```
Uncaught Kotlin exception: kotlin.IllegalStateException:
   at 0   example.kexe    0x10669bc88    kfun:kotlin.Throwable#<init>(kotlin.String?){
} + 88 (/opt/buildAgent/work/c3a91df21e46e2c8/kotlin/kotlin-native/runtime/src/main/ko
tlin/kotlin/Throwable.kt:24:37)
   at 1   example.kexe    0x106696bd6    kfun:kotlin.Exception#<init>(kotlin.String?){
} + 86 (/opt/buildAgent/work/c3a91df21e46e2c8/kotlin/kotlin-native/runtime/src/main/ko
tlin/kotlin/Exceptions.kt:23:44)
   at 2   example.kexe    0x106696cc6    kfun:kotlin.RuntimeException#<init>(kotlin.St
ring?){} + 86 (/opt/buildAgent/work/c3a91df21e46e2c8/kotlin/kotlin-native/runtime/src/
main/kotlin/kotlin/Exceptions.kt:34:44)
   at 3   example.kexe    0x106697016    kfun:kotlin.IllegalStateException#<init>(kotl
in.String?){} + 86 (/opt/buildAgent/work/c3a91df21e46e2c8/kotlin/kotlin-native/runtime
/src/main/kotlin/kotlin/Exceptions.kt:70:44)
   at 4   example.kexe    0x106689d35    kfun:#bar(){} + 117 [inlined] (/opt/buildAgen
t/work/c3a91df21e46e2c8/kotlin/libraries/stdlib/src/kotlin/util/Preconditions.kt:143:5
6)
>>  at 5   example.kexe    0x106689d35    kfun:#bar(){} + 117 [inlined] (/private/tmp/
backtrace/src/commonMain/kotlin/app.kt:4:5)
   at 6   example.kexe    0x106689d35    kfun:#bar(){} + 117 (/private/tmp/backtrace/s
rc/commonMain/kotlin/app.kt:2:13)
   at 7   example.kexe    0x106689cac    kfun:#main(){} + 12 (/private/tmp/backtrace/s
rc/commonMain/kotlin/app.kt:1:14)
...
```

To produce better stack traces with libbacktrace, add the following line to `gradle.properties`:

```
# gradle.properties
kotlin.native.binary.sourceInfoType=libbacktrace
```

Please tell us how debugging Kotlin/Native with libbacktrace works for you in this YouTrack issue.

支持独立的 Android 可执行文件

Previously, Android Native executables in Kotlin/Native were not actually executables but shared libraries that you could use as a NativeActivity. Now there's an option to generate standard executables for Android Native targets.

For that, in the `build.gradle(.kts)` part of your project, configure the executable block of your `androidNative` target. Add the following binary option:

```
kotlin {
    androidNativeX64("android") {
        binaries {
            executable {
                binaryOptions["androidProgramType"] = "standalone"
            }
        }
    }
}
```

Note that this feature will become the default in Kotlin 1.7.0. If you want to preserve the current behavior, use the following setting:

```
binaryOptions["androidProgramType"] = "nativeActivity"
```

Thanks to Mattia Iavarone for the implementation!

性能提升

We are working hard on Kotlin/Native to speed up the compilation process and improve your developing experience.

Kotlin 1.6.20 brings some performance updates and bug fixes that affect the LLVM IR that Kotlin generates. According to the benchmarks on our internal projects, we achieved the following performance boosts on average:

- 15% reduction in execution time
- 20% reduction in the code size of both release and debug binaries
- 26% reduction in the compilation time of release binaries

These changes also provide a 10% reduction in compilation time for a debug binary on a large internal project.

To achieve this, we've implemented static initialization for some of the compiler-generated synthetic objects, improved the way we structure LLVM IR for every function, and optimized the compiler caches.

改进了 cinterop 模块导入过程中的错误处理

This release introduces improved error handling for cases where you import an Objective-C module using the `cinterop` tool (as is typical for CocoaPods pods). Previously, if you got an error while trying to work with an Objective-C module (for instance, when dealing with a compilation error in a header), you received an uninformative error message, such as `fatal error: could not build module $name`. We expanded upon this part of the `cinterop` tool, so you'll get an error message with an extended description.

对 Xcode 13 库的支持

Libraries delivered with Xcode 13 have full support as of this release. Feel free to access them from anywhere in your Kotlin code.

Kotlin 多平台

1.6.20 brings the following notable updates to Kotlin Multiplatform:

- 所有新建多平台项目现在默认都支持分层结构
- Kotlin CocoaPods Gradle 插件为 CocoaPods 集成提供了几个有用的特性

多平台项目的分层结构支持

Kotlin 1.6.20 comes with hierarchical structure support enabled by default. Since introducing it in Kotlin 1.4.0, we've significantly improved the frontend and made IDE import stable.

Previously, there were two ways to add code in a multiplatform project. The first was to insert it in a platform-specific source set, which is limited to one target and can't be reused by other platforms. The second is to use a common source set shared across all the platforms that are currently supported by Kotlin.

Now you can share source code among several similar native targets that reuse a lot of the common logic and third-party APIs. The technology will provide the correct default dependencies and find the exact API available in the shared code. This eliminates a complex build setup and having to use workarounds to get IDE support for sharing source sets among native targets. It also helps prevent unsafe API usages meant for a different target.

The technology will come in handy for library authors, too, as a hierarchical project structure allows them to publish and consume libraries with common APIs for a subset of targets.

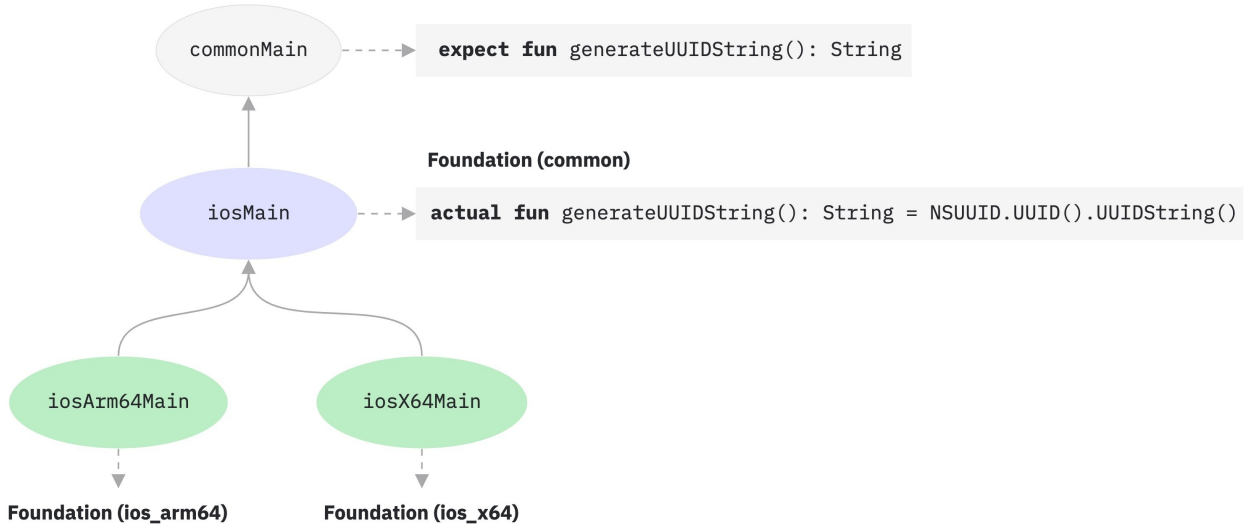
By default, libraries published with the hierarchical project structure are compatible only with hierarchical structure projects.

在项目中更好地共享代码

Without hierarchical structure support, there is no straightforward way to share code across *some* but not *all* Kotlin targets. One popular example is sharing code across all iOS targets and having access to iOS-specific dependencies, like Foundation.

Thanks to the hierarchical project structure support, you can now achieve this out of the box. In the new structure, source sets form a hierarchy. You can use platform-specific language features and dependencies available for each target that a given source set compiles to.

For example, consider a typical multiplatform project with two targets — `iosArm64` and `iosX64` for iOS devices and simulators. The Kotlin tooling understands that both targets have the same function and allows you to access that function from the intermediate source set, `iosMain`.



The Kotlin toolchain provides the correct default dependencies, like Kotlin/Native stdlib or native libraries. Moreover, Kotlin tooling will try its best to find exactly the API surface area available in the shared code. This prevents such cases as, for example, the use of a macOS-specific function in code shared for Windows.

库作者的更多机会

When a multiplatform library is published, the API of its intermediate source sets is now properly published alongside it, making it available for consumers. Again, the Kotlin toolchain will automatically figure out the API available in the consumer source set while carefully watching out for unsafe usages, like using an API meant for the JVM in JS code. Learn more about sharing code in libraries.

配置与设置

Starting with Kotlin 1.6.20, all your new multiplatform projects will have a hierarchical project structure. No additional setup is required.

- If you've already turned it on manually, you can remove the deprecated options from `gradle.properties`:

  ```
  # gradle.properties
  kotlin.mpp.enableGranularSourceSetsMetadata=true
  kotlin.native.enableDependencyPropagation=false // or 'true', depending on your pr
  evious setup
  ```

- For Kotlin 1.6.20, we recommend using Android Studio 2021.1.1 (Bumblebee) or later to get the best experience.

- You can also opt out. To disable hierarchical structure support, set the following options in `gradle.properties`:

  ```
  # gradle.properties
  kotlin.mpp.hierarchicalStructureSupport=false
  ```

敬请反馈

This is a significant change to the whole ecosystem. We would appreciate your feedback to help make it even better.

Try it now and report any difficulties you encounter to our issue tracker.

Kotlin CocoaPods Gradle 插件

To simplify CocoaPods integration, Kotlin 1.6.20 delivers the following features:

- The CocoaPods plugin now has tasks that build XCFrameworks with all registered targets and generate the Podspec file. This can be useful when you don't want to integrate with Xcode directly, but you want to build artifacts and deploy them to your local CocoaPods repository.

  Learn more about building XCFrameworks.

- If you use CocoaPods integration in your projects, you're used to specifying the required Pod version for the entire Gradle project. Now you have more options:

  - Specify the Pod version directly in the `cocoapods` block
  - Continue using a Gradle project version

  If none of these properties is configured, you'll get an error.

- You can now configure the CocoaPod name in the `cocoapods` block instead of changing the name of the whole Gradle project.
- The CocoaPods plugin introduces a new `extraSpecAttributes` property, which you can use to configure properties in a Podspec file that were previously hard-coded, like `libraries` or `vendored_frameworks`.

```
kotlin {
    cocoapods {
        version = "1.0"
        name = "MyCocoaPod"
        extraSpecAttributes["social_media_url"] = 'https://twitter.com/kotlin'
        extraSpecAttributes["vendored_frameworks"] = 'CustomFramework.xcframework'
        extraSpecAttributes["libraries"] = 'xml'
    }
}
```

See the full Kotlin CocoaPods Gradle plugin DSL reference.

Kotlin/JS

Kotlin/JS improvements in 1.6.20 mainly affect the IR compiler:

- 开发版二进制文件的增量编译（IR）
- 时顶层属性默认惰性初始化（IR）
- 模块默认采用分开的 JS 文件（IR）
- Char 类优化（IR）
- 导出改进（IR 与旧版后端）
- 异步测试的 @AfterTest 保证

使用 IR 编译器对开发版二进制文件进行增量编译

To make Kotlin/JS development with the IR compiler more efficient, we're introducing a new *incremental compilation* mode.

When building **development binaries** with the `compileDevelopmentExecutableKotlinJs` Gradle task in this mode, the compiler caches the results of previous compilations on the module level. It uses the cached compilation results for unchanged source files during subsequent compilations, making them complete more quickly, especially with small changes. Note that this improvement exclusively targets the development process (shortening the edit-build-debug cycle) and doesn't affect the building of production artifacts.

To enable incremental compilation for development binaries, add the following line to the project's `gradle.properties`:

```
# gradle.properties
kotlin.incremental.js.ir=true // false by default
```

In our test projects, the new mode made incremental compilation up to 30% faster. However, the clean build in this mode became slower because of the need to create and populate the caches.

Please tell us what you think of using incremental compilation with your Kotlin/JS projects in this YouTrack issue.

使用 IR 编译器时顶层属性默认惰性初始化

In Kotlin 1.4.30, we presented a prototype of lazy initialization of top-level properties in the JS IR compiler. By eliminating the need to initialize all properties when the application launches, lazy initialization reduces the startup time. Our measurements showed about a 10% speed-up on a real-life Kotlin/JS application.

Now, having polished and properly tested this mechanism, we're making lazy initialization the default for top-level properties in the IR compiler.

```
// lazy initialization
val a = run {
    val result = // intensive computations
        println(result)
    result
} // run is executed upon the first usage of the variable
```

If for some reason you need to initialize a property eagerly (upon the application start), mark it with the `@EagerInitialization` annotation.

使用 IR 编译器时模块默认采用分开的 JS 文件

Previously, the JS IR compiler offered an ability to generate separate `.js` files for project modules. This was an alternative to the default option – a single `.js` file for the whole project. This file might be too large and inconvenient to use, because whenever you want to use a function from your project, you have to include the entire JS file as a dependency. Having multiple files adds flexibility and decreases the size of such dependencies. This feature was available with the `-Xir-per-module` compiler option.

Starting from 1.6.20, the JS IR compiler generates separate `.js` files for project modules by default.

Compiling the project into a single `.js` file is now available with the following Gradle property:

```
# gradle.properties
kotlin.js.ir.output.granularity=whole-program // `per-module` is the default
```

In previous releases, the experimental per-module mode (available via the `-Xir-per-module=true` flag) invoked `main()` functions in each module. This is inconsistent with the regular single `.js` mode. Starting with 1.6.20, the `main()` function will be invoked in the main module only in both cases. If you do need to run some code when a module is loaded, you can use top-level properties annotated with the `@EagerInitialization` annotation. See Lazy initialization of top-level properties by default (IR).

Char 类优化

The `Char` class is now handled by the Kotlin/JS compiler without introducing boxing (similar to inline classes). This speeds up operations on chars in Kotlin/JS code.

Aside from the performance improvement, this changes the way `Char` is exported to JavaScript: it's now translated to `Number`.

导出与 TypeScript 声明生成的改进

Kotlin 1.6.20 is bringing multiple fixes and improvements to the export mechanism (the `@JsExport` annotation), including the generation of TypeScript declarations (`.d.ts`). We've added the ability to export interfaces and enums, and we've fixed the export behavior in some corner cases that were reported to us previously. For more details, see the list of export improvements in YouTrack.

Learn more about using Kotlin code from JavaScript.

异步测试的 @AfterTest 保证

Kotlin 1.6.20 makes `@AfterTest` functions work properly with asynchronous tests on Kotlin/JS. If a test function's return type is statically resolved to `Promise`, the compiler now schedules the execution of the `@AfterTest` function to the corresponding `then()` callback.

安全

Kotlin 1.6.20 introduces a couple of features to improve the security of your code:

- 在 klibs 中使用相对路径
- Kotlin/JS Gradle 项目持久化 yarn.lock

- 默认使用 `--ignore-scripts` 安装 npm 依赖

在 klibs 中使用相对路径

A library in `klib` format contains a serialized IR representation of source files, which also includes their paths for generating proper debug information. Before Kotlin 1.6.20, stored file paths were absolute. Since the library author may not want to share absolute paths, the 1.6.20 version comes with an alternative option.

If you are publishing a `klib` and want to use only relative paths of source files in the artifact, you can now pass the `-Xklib-relative-path-base` compiler option with one or multiple base paths of source files:

> 【Kotlin】

```
tasks.withType(org.jetbrains.kotlin.gradle.dsl.KotlinCompile::class).configureEach {
    // $base is a base path of source files
    kotlinOptions.freeCompilerArgs += "-Xklib-relative-path-base=$base"
}
```

> 【Groovy】

```
tasks.withType(org.jetbrains.kotlin.gradle.dsl.KotlinCompile).configureEach {
    kotlinOptions {
        // $base is a base path of source files
        freeCompilerArgs += "-Xklib-relative-path-base=$base"
    }
}
```

Kotlin/JS Gradle 项目持久化 yarn.lock

> The feature was backported to Kotlin 1.6.10.
>
> 

The Kotlin/JS Gradle plugin now provides an ability to persist the `yarn.lock` file, making it possible to lock the versions of the npm dependencies for your project without additional Gradle configuration. The feature brings changes to the default project structure by adding the auto-generated `kotlin-js-store` directory to the project root. It holds the `yarn.lock` file inside.

We strongly recommend committing the `kotlin-js-store` directory and its contents to your version control system. Committing lockfiles to your version control system is a recommended practice because it ensures your application is being built with the exact same dependency tree on all machines, regardless of whether those are development environments on other machines or CI/CD services. Lockfiles also prevent your npm dependencies from being silently updated when a project is checked out on a new machine, which is a security concern.

Tools like Dependabot can also parse the `yarn.lock` files of your Kotlin/JS projects, and provide you with warnings if any npm package you depend on is compromised.

If needed, you can change both directory and lockfile names in the build script:

【Kotlin】

```
rootProject.plugins.withType<org.jetbrains.kotlin.gradle.targets.js.yarn.YarnPlugin> {
    rootProject.the<org.jetbrains.kotlin.gradle.targets.js.yarn.YarnRootExtension>().l
ockFileDirectory =
        project.rootDir.resolve("my-kotlin-js-store")
    rootProject.the<org.jetbrains.kotlin.gradle.targets.js.yarn.YarnRootExtension>().l
ockFileName = "my-yarn.lock"
}
```

【Groovy】

```
rootProject.plugins.withType(org.jetbrains.kotlin.gradle.targets.js.yarn.YarnPlugin) {
    rootProject.extensions.getByType(org.jetbrains.kotlin.gradle.targets.js.yarn.YarnR
ootExtension).lockFileDirectory =
        file("my-kotlin-js-store")
    rootProject.extensions.getByType(org.jetbrains.kotlin.gradle.targets.js.yarn.YarnR
ootExtension).lockFileName = 'my-yarn.lock'
}
```

> Changing the name of the lockfile may cause dependency inspection tools to no longer pick up the file.



默认使用 --ignore-scripts 安装 npm 依赖

> The feature was backported to Kotlin 1.6.10.



The Kotlin/JS Gradle plugin now prevents the execution of lifecycle scripts during the installation of npm dependencies by default. The change is aimed at reducing the likelihood of executing malicious code from compromised npm packages.

To roll back to the old configuration, you can explicitly enable lifecycle scripts execution by adding the following lines to `build.gradle(.kts)`:

【Kotlin】

```
rootProject.plugins.withType<org.jetbrains.kotlin.gradle.targets.js.yarn.YarnPlugin> {
    rootProject.the<org.jetbrains.kotlin.gradle.targets.js.yarn.YarnRootExtension>().i
gnoreScripts = false
}
```

【Groovy】

```
rootProject.plugins.withType(org.jetbrains.kotlin.gradle.targets.js.yarn.YarnPlugin) {
    rootProject.extensions.getByType(org.jetbrains.kotlin.gradle.targets.js.yarn.YarnR
ootExtension).ignoreScripts = false
}
```

Learn more about npm dependencies of a Kotlin/JS Gradle project.

Gradle

Kotlin 1.6.20 brings the following changes for the Kotlin Gradle Plugin:

- 用于定义 Kotlin 编译器执行策略的新属性 `kotlin.compiler.execution.strategy` 与 `compilerExecutionStrategy`
- 弃用选项 `kapt.use.worker.api`、`kotlin.experimental.coroutines` 与 `kotlin.coroutines`
- 删除 `kotlin.parallel.tasks.in.project` 构建选项

用于定义 Kotlin 编译器执行策略的属性

Before Kotlin 1.6.20, you used the system property `-Dkotlin.compiler.execution.strategy` to define a Kotlin compiler execution strategy. This property might have been inconvenient in some cases. Kotlin 1.6.20 introduces a Gradle property with the same name, `kotlin.compiler.execution.strategy`, and the compile task property `compilerExecutionStrategy`.

The system property still works, but it will be removed in future releases.

The current priority of properties is the following:

- The task property `compilerExecutionStrategy` takes priority over the system property and the Gradle property `kotlin.compiler.execution.strategy`.
- The Gradle property takes priority over the system property.

There are three compiler execution strategies that you can assign to these properties:

Strategy	Where Kotlin compiler is executed	Incremental compilation	Other characteristics
Daemon	Inside its own daemon process	Yes	*The default strategy*. Can be shared between different Gradle daemons
In process	Inside the Gradle daemon process	No	May share the heap with the Gradle daemon
Out of process	In a separate process for each call	No	—

Accordingly, the available values for `kotlin.compiler.execution.strategy` properties (both system and Gradle's) are:

1. `daemon` (default)
2. `in-process`
3. `out-of-process`

Use the Gradle property `kotlin.compiler.execution.strategy` in `gradle.properties`:

```
# gradle.properties
kotlin.compiler.execution.strategy=out-of-process
```

The available values for the `compilerExecutionStrategy` task property are:

1. `org.jetbrains.kotlin.gradle.tasks.KotlinCompilerExecutionStrategy.DAEMON` (default)
2. `org.jetbrains.kotlin.gradle.tasks.KotlinCompilerExecutionStrategy.IN_PROCESS`
3. `org.jetbrains.kotlin.gradle.tasks.KotlinCompilerExecutionStrategy.OUT_OF_PROCESS`

Use the task property `compilerExecutionStrategy` in the `build.gradle.kts` build script:

```
import org.jetbrains.kotlin.gradle.dsl.KotlinCompile
import org.jetbrains.kotlin.gradle.tasks.KotlinCompilerExecutionStrategy

// ...

tasks.withType<KotlinCompile>().configureEach {
    compilerExecutionStrategy.set(KotlinCompilerExecutionStrategy.IN_PROCESS)
}
```

Please leave your feedback in this YouTrack task.

弃用一些 kapt 与协程的构建选项

In Kotlin 1.6.20, we changed deprecation levels of the properties:

- We deprecated the ability to run kapt via the Kotlin daemon with `kapt.use.worker.api` – now it produces a warning to Gradle's output. By default, kapt has been using Gradle workers since the 1.3.70 release, and we recommend sticking to this method.

  We are going to remove the option `kapt.use.worker.api` in future releases.

- We deprecated the `kotlin.experimental.coroutines` Gradle DSL option and the `kotlin.coroutines` property used in `gradle.properties`. Just use *suspending functions* or add the `kotlinx.coroutines` dependency to your `build.gradle(.kts)` file.

  Learn more about coroutines in the Coroutines guide.

删除 kotlin.parallel.tasks.in.project 构建选项

In Kotlin 1.5.20, we announced the deprecation of the build option `kotlin.parallel.tasks.in.project`. This option has been removed in Kotlin 1.6.20.

Depending on the project, parallel compilation in the Kotlin daemon may require more memory. To reduce memory consumption, increase the heap size for the Kotlin daemon.

Learn more about the currently supported compiler options in the Kotlin Gradle plugin.

Kotlin 1.6.0 的新特性

发布于：*2021-11-16*

Kotlin 1.6.0 引入了新的语言特性、对现有特性的优化与改进以及对 Kotlin 标准库的大量改进。

还可以在版本发布博文中找到这些变更的概述。

语言

Kotlin 1.6.0 将上一版本 1.5.30 中预览的几个语言特性提升为稳定版了：

- 稳定版对于枚举、密封类与布尔值主语穷尽 when 语句
- 稳定版挂起函数作为超类型
- 稳定版挂起转换
- 稳定版注解类实例化

还包括各种类型推断改进以及对类的类型参数上注解的支持：

- 改进了递归泛型类型的类型推断
- 构建器类型推断变更
- 对类的类型参数上注解的支持

稳定版对于枚举、密封类与布尔值主语穷尽 when 语句

An *exhaustive* `when` statement contains branches for all possible types or values of its subject, or for some types plus an `else` branch. It covers all possible cases, making your code safer.

We will soon prohibit non-exhaustive `when` statements to make the behavior consistent with `when` expressions. To ensure smooth migration, Kotlin 1.6.0 reports warnings about non-exhaustive `when` statements with an enum, sealed, or Boolean subject. These warnings will become errors in future releases.

```
sealed class Contact {
    data class PhoneCall(val number: String) : Contact()
    data class TextMessage(val number: String) : Contact()
}

fun Contact.messageCost(): Int =
    when(this) { // Error: 'when' expression must be exhaustive
        is Contact.PhoneCall -> 42
```

```
    }

fun sendMessage(contact: Contact, message: String) {
    // Starting with 1.6.0

    // Warning: Non exhaustive 'when' statements on Boolean will be
    // prohibited in 1.7, add 'false' branch or 'else' branch instead
    when(message.isEmpty()) {
        true -> return
    }
    // Warning: Non exhaustive 'when' statements on sealed class/interface will be
    // prohibited in 1.7, add 'is TextMessage' branch or 'else' branch instead
    when(contact) {
        is Contact.PhoneCall -> TODO()
    }
}
```

See this YouTrack ticket for a more detailed explanation of the change and its effects.

稳定版挂起函数作为超类型

Implementation of suspending functional types has become Stable in Kotlin 1.6.0. A preview was available in 1.5.30.

The feature can be useful when designing APIs that use Kotlin coroutines and accept suspending functional types. You can now streamline your code by enclosing the desired behavior in a separate class that implements a suspending functional type.

```
class MyClickAction : suspend () -> Unit {
    override suspend fun invoke() { TODO() }
}

fun launchOnClick(action: suspend () -> Unit) {}
```

You can use an instance of this class where only lambdas and suspending function references were allowed previously: `launchOnClick(MyClickAction())`.

There are currently two limitations coming from implementation details:

- You can't mix ordinary functional types and suspending ones in the list of supertypes.
- You can't use multiple suspending functional supertypes.

稳定版挂起转换

Kotlin 1.6.0 introduces Stable conversions from regular to suspending functional types. Starting from 1.4.0, the feature supported functional literals and callable references. With 1.6.0, it works with any form of expression. As a call argument, you can now pass any expression of a suitable regular functional type where suspending is expected. The compiler will perform an implicit conversion automatically.

```
fun getSuspending(suspending: suspend () -> Unit) {}

fun suspending() {}

fun test(regular: () -> Unit) {
    getSuspending { }           // OK
    getSuspending(::suspending) // OK
    getSuspending(regular)      // OK
}
```

稳定版注解类实例化

Kotlin 1.5.30 introduced experimental support for instantiation of annotation classes on the JVM platform. With 1.6.0, the feature is available by default both for Kotlin/JVM and Kotlin/JS.

Learn more about instantiation of annotation classes in this KEEP.

改进了递归泛型类型的类型推断

Kotlin 1.5.30 introduced an improvement to type inference for recursive generic types, which allowed their type arguments to be inferred based only on the upper bounds of the corresponding type parameters. The improvement was available with the compiler option. In version 1.6.0 and later, it is enabled by default.

```
// Before 1.5.30
val containerA = PostgreSQLContainer<Nothing>(DockerImageName.parse("postgres:13-alpin
e")).apply {
  withDatabaseName("db")
  withUsername("user")
  withPassword("password")
  withInitScript("sql/schema.sql")
}

// With compiler option in 1.5.30 or by default starting with 1.6.0
val containerB = PostgreSQLContainer(DockerImageName.parse("postgres:13-alpine"))
  .withDatabaseName("db")
  .withUsername("user")
  .withPassword("password")
  .withInitScript("sql/schema.sql")
```

构建器类型推断变更

Builder inference is a type inference flavor which is useful when calling generic builder functions. It can infer the type arguments of a call with the help of type information from calls inside its lambda argument.

We're making multiple changes that are bringing us closer to fully stable builder inference. Starting with 1.6.0:

- You can make calls returning an instance of a not yet inferred type inside a builder lambda without specifying the `-Xunrestricted-builder-inference` compiler option introduced in 1.5.30.
- With `-Xenable-builder-inference`, you can write your own builders without applying the `@BuilderInference` annotation.

  > Note that clients of these builders will need to specify the same `-Xenable-builder-inference` compiler option.
  >
  > 

- With the `-Xenable-builder-inference`, builder inference automatically activates if a regular type inference cannot get enough information about a type.

Learn how to write custom generic builders.

对类的类型参数上注解的支持

对类的类型参数上注解的支持 looks like this:

```
@Target(AnnotationTarget.TYPE_PARAMETER)
annotation class BoxContent

class Box<@BoxContent T> {}
```

Annotations on all type parameters are emitted into JVM bytecode so annotation processors are able to use them.

For the motivating use case, read this YouTrack ticket.

Learn more about annotations.

对以前版本的 API 支持更长时间

Starting with Kotlin 1.6.0, we will support development for three previous API versions instead of two, along with the current stable one. Currently, we support versions 1.3, 1.4, 1.5, and 1.6.

Kotlin/JVM

For Kotlin/JVM, starting with 1.6.0, the compiler can generate classes with a bytecode version corresponding to JVM 17. The new language version also includes optimized delegated properties and repeatable annotations, which we had on the roadmap:

- 1.8 JVM 目标平台中运行时保留的可重复注解
- 优化了在给定 KProperty 实例上调用了 get/set 的委托属性

1.8 JVM 目标平台中运行时保留的可重复注解

Java 8 introduced repeatable annotations, which can be applied multiple times to a single code element. The feature requires two declarations to be present in the Java code: the repeatable annotation itself marked with `@java.lang.annotation.Repeatable` and the containing annotation to hold its values.

Kotlin also has repeatable annotations, but requires only `@kotlin.annotation.Repeatable` to be present on an annotation declaration to make it repeatable. Before 1.6.0, the feature supported only `SOURCE` retention and was incompatible with Java's repeatable annotations. Kotlin 1.6.0 removes these limitations. `@kotlin.annotation.Repeatable` now accepts any retention and makes the annotation repeatable both in Kotlin and Java. Java's repeatable annotations are now also supported from the Kotlin side.

While you can declare a containing annotation, it's not necessary. For example:

- If an annotation `@Tag` is marked with `@kotlin.annotation.Repeatable`, the Kotlin compiler automatically generates a containing annotation class under the name of `@Tag.Container`:

```
@Repeatable
annotation class Tag(val name: String)

// The compiler generates @Tag.Container containing annotation
```

- To set a custom name for a containing annotation, apply the `@kotlin.jvm.JvmRepeatable` meta-annotation and pass the explicitly declared containing annotation class as an argument:

```
@JvmRepeatable(Tags::class)
annotation class Tag(val name: String)

annotation class Tags(val value: Array<Tag>)
```

Kotlin reflection now supports both Kotlin's and Java's repeatable annotations via a new function, `KAnnotatedElement.findAnnotations()`.

Learn more about Kotlin repeatable annotations in this KEEP.

优化了在给定 KProperty 实例上调用了 get/set 的委托属性

We optimized the generated JVM bytecode by omitting the `$delegate` field and generating immediate access to the referenced property.

For example, in the following code

```
class Box<T> {
    private var impl: T = ...

    var content: T by ::impl
}
```

Kotlin no longer generates the field `content$delegate`. Property accessors of the `content` variable invoke the `impl` variable directly, skipping the delegated property's `getValue` / `setValue` operators and thus avoiding the need for the property reference object of the `KProperty` type.

Thanks to our Google colleagues for the implementation!

Learn more about delegated properties.

Kotlin/Native

Kotlin/Native is receiving multiple improvements and component updates, some of them in the preview state:

- 新版内存管理器预览
- 对 Xcode 13 的支持
- 在任何主机上编译 Windows 目标
- LLVM 与链接器更新
- 性能提升
- 与 JVM 及 JS IR 后端统一编译器插件 ABI
- klib 链接失败的详细错误信息
- 重新设计了未处理异常的处理 API

新版内存管理器预览

> The new Kotlin/Native memory manager is Experimental. It may be dropped or changed at any time. Opt-in is required (see details below), and you should use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack.
>
> 

With Kotlin 1.6.0, you can try the development preview of the new Kotlin/Native memory manager. It moves us closer to eliminating the differences between the JVM and Native platforms to provide a consistent developer experience in multiplatform projects.

One of the notable changes is the lazy initialization of top-level properties, like in Kotlin/JVM. A top-level property gets initialized when a top-level property or function from the same file is accessed for the first time. This mode also includes global interprocedural optimization (enabled only for release binaries), which removes redundant initialization checks.

We've recently published a blog post about the new memory manager. Read it to learn about the current state of the new memory manager and find some demo projects, or jump right to the migration instructions to try it yourself. Please check how the new memory manager works on your projects and share feedback in our issue tracker, YouTrack.

对 Xcode 13 的支持

Kotlin/Native 1.6.0 supports Xcode 13 – the latest version of Xcode. Feel free to update your Xcode and continue working on your Kotlin projects for Apple operating systems.

> New libraries added in Xcode 13 aren't available for use in Kotlin 1.6.0, but we're going to add support for them in upcoming versions.
>
> 

在任何主机上编译 Windows 目标

Starting from 1.6.0, you don't need a Windows host to compile the Windows targets `mingwX64` and `mingwX86`. They can be compiled on any host that supports Kotlin/Native.

LLVM 与链接器更新

We've reworked the LLVM dependency that Kotlin/Native uses under the hood. This brings various benefits, including:

- Updated LLVM version to 11.1.0.
- Decreased dependency size. For example, on macOS it's now about 300 MB instead of 1200 MB in the previous version.
- Excluded dependency on the `ncurses5` library that isn't available in modern Linux

distributions.

In addition to the LLVM update, Kotlin/Native now uses the LLD linker (a linker from the LLVM project) for MingGW targets. It provides various benefits over the previously used ld.bfd linker, and will allow us to improve runtime performance of produced binaries and support compiler caches for MinGW targets. Note that LLD requires import libraries for DLL linkage. Learn more in this Stack Overflow thread.

性能提升

Kotlin/Native 1.6.0 delivers the following performance improvements:

- Compilation time: compiler caches are enabled by default for `linuxX64` and `iosArm64` targets. This speeds up most compilations in debug mode (except the first one). Measurements showed about a 200% speed increase on our test projects. The compiler caches have been available for these targets since Kotlin 1.5.0 with additional Gradle properties; you can remove them now.
- Runtime: iterating over arrays with `for` loops is now up to 12% faster thanks to optimizations in the produced LLVM code.

与 JVM 及 JS IR 后端统一编译器插件 ABI

> The option to use the common IR compiler plugin ABI for Kotlin/Native is Experimental. It may be dropped or changed at any time. Opt-in is required (see details below), and you should use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack.
>
> 

In previous versions, authors of compiler plugins had to provide separate artifacts for Kotlin/Native because of the differences in the ABI.

Starting from 1.6.0, the Kotlin Multiplatform Gradle plugin is able to use the embeddable compiler jar – the one used for the JVM and JS IR backends – for Kotlin/Native. This is a step toward unification of the compiler plugin development experience, as you can now use the same compiler plugin artifacts for Native and other supported platforms.

This is a preview version of such support, and it requires an opt-in. To start using generic compiler plugin artifacts for Kotlin/Native, add the following line to `gradle.properties`: `kotlin.native.useEmbeddableCompilerJar=true`.

We're planning to use the embeddable compiler jar for Kotlin/Native by default in the future, so it's vital for us to hear how the preview works for you.

If you are an author of a compiler plugin, please try this mode and check if it works for your plugin. Note that depending on your plugin's structure, migration steps may be required. See this YouTrack issue for migration instructions and leave your feedback in the comments.

klib 链接失败的详细错误信息

The Kotlin/Native compiler now provides detailed error messages for klib linkage errors. The messages now have clear error descriptions, and they also include information about possible causes and ways to fix them.

For example:

- 1.5.30:

```
  e: java.lang.IllegalStateException: IrTypeAliasSymbol expected: Unbound public s
ymbol for public kotlinx.coroutines/CancellationException|null[0]
  <stack trace>
```

- 1.6.0:

```
  e: The symbol of unexpected type encountered during IR deserialization: IrClassP
ublicSymbolImpl, kotlinx.coroutines/CancellationException|null[0].
  IrTypeAliasSymbol is expected.

  This could happen if there are two libraries, where one library was compiled aga
inst the different version of the other library than the one currently used in the
 project.
  Please check that the project configuration is correct and has consistent versio
ns of dependencies.

  The list of libraries that depend on "org.jetbrains.kotlinx:kotlinx-coroutines-c
ore (org.jetbrains.kotlinx:kotlinx-coroutines-core-macosx64)" and may lead to conf
licts:
  <list of libraries and potential version mismatches>

  Project dependencies:
  <dependencies tree>
```

重新设计了未处理异常的处理 API

We've unified the processing of unhandled exceptions throughout the Kotlin/Native runtime and exposed the default processing as the function `processUnhandledException(throwable: Throwable)` for use by custom execution environments, like `kotlinx.coroutines`. This processing is also applied to exceptions that escape operation in `Worker.executeAfter()`, but only for the new memory manager.

API improvements also affected the hooks that have been set by `setUnhandledExceptionHook()`. Previously such hooks were reset after the Kotlin/Native runtime called the hook with an unhandled exception, and the program would always terminate right after. Now these hooks may be used more than once, and if you want the program to always terminate on an unhandled exception, either do not set an unhandled exception hook (`setUnhandledExceptionHook()`), or make sure to call `terminateWithUnhandledException()` at the end of your hook. This will help you send exceptions to a third-party crash reporting service (like Firebase Crashlytics) and then terminate the program. Exceptions that escape `main()` and exceptions that cross the interop boundary will always terminate the program, even if the hook did not call `terminateWithUnhandledException()`.

Kotlin/JS

We're continuing to work on stabilizing the IR backend for the Kotlin/JS compiler. Kotlin/JS now has an 禁用下载 Node.js 与 Yarn 的选项.

可以选择使用预装的 Node.js 与 Yarn

You can now disable downloading Node.js and Yarn when building Kotlin/JS projects and use the instances already installed on the host. This is useful for building on servers without internet connectivity, such as CI servers.

To disable downloading external components, add the following lines to your `build.gradle(.kts)`:

- Yarn:

> 【Kotlin】

```
    rootProject.plugins.withType<org.jetbrains.kotlin.gradle.targets.js.yarn.YarnPlugi
n> {
        rootProject.the<org.jetbrains.kotlin.gradle.targets.js.yarn.YarnRootExtension>
().download = false // or true for default behavior
    }
```

> 【Groovy】

```
    rootProject.plugins.withType(org.jetbrains.kotlin.gradle.targets.js.yarn.YarnPlugi
n) {
        rootProject.extensions.getByType(org.jetbrains.kotlin.gradle.targets.js.yarn.Y
arnRootExtension).download = false
    }
```

- Node.js:

【Kotlin】

```
    rootProject.plugins.withType<org.jetbrains.kotlin.gradle.targets.js.nodejs.NodeJsR
ootPlugin> {
        rootProject.the<org.jetbrains.kotlin.gradle.targets.js.nodejs.NodeJsRootExtens
ion>().download = false // or true for default behavior
    }
```

【Groovy】

```
    rootProject.plugins.withType(org.jetbrains.kotlin.gradle.targets.js.nodejs.NodeJsR
ootPlugin) {
        rootProject.extensions.getByType(org.jetbrains.kotlin.gradle.targets.js.nodejs
.NodeJsRootExtension).download = false
    }
```

Kotlin Gradle 插件

In Kotlin 1.6.0, we changed the deprecation level of the `KotlinGradleSubplugin` class to 'ERROR'. This class was used for writing compiler plugins. In the following releases, we'll remove this class. Use the class `KotlinCompilerPluginSupportPlugin` instead.

We removed the `kotlin.useFallbackCompilerSearch` build option and the `noReflect` and `includeRuntime` compiler options. The `useIR` compiler option has been hidden and will be removed in upcoming releases.

Learn more about the currently supported compiler options in the Kotlin Gradle plugin.

标准库

The new 1.6.0 version of the standard library stabilizes experimental features, introduces new ones, and unifies its behavior across the platforms:

- 新版 readline 函数
- 稳定版 typeOf()
- 稳定版集合构建器
- 稳定版 Duration API
- 按 Regex 拆分为序列
- 整数的循环移位运算
- JS 平台 replace() 与 replaceFirst() 的变更
- 既有 API 的改进
- 弃用项

新版 readline 函数

Kotlin 1.6.0 offers new functions for handling standard input: `readln()` and `readlnOrNull()`.

> For now, new functions are available for the JVM and Native target platforms only.
>
> 

Earlier versions	1.6.0 alternative	Usage
`readLine()!!`	`readln()`	Reads a line from stdin and returns it, or throws a `RuntimeException` if EOF has been reached.
`readLine()`	`readlnOrNull()`	Reads a line from stdin and returns it, or returns `null` if EOF has been reached.

We believe that eliminating the need to use `!!` when reading a line will improve the experience for newcomers and simplify teaching Kotlin. To make the read-line operation name consistent with its `println()` counterpart, we've decided to shorten the names of new functions to 'ln'.

```
println("What is your nickname?")
val nickname = readln()
println("Hello, $nickname!")
```

```
fun main() {
//sampleStart
    var sum = 0
    while (true) {
        val nextLine = readlnOrNull().takeUnless {
            it.isNullOrEmpty()
        } ?: break
        sum += nextLine.toInt()
    }
    println(sum)
//sampleEnd
}
```

The existing `readLine()` function will get a lower priority than `readln()` and `readlnOrNull()` in your IDE code completion. IDE inspections will also recommend using new functions instead of the legacy `readLine()`.

We're planning to gradually deprecate the `readLine()` function in future releases.

稳定版 typeOf()

Version 1.6.0 brings a Stable `typeOf()` function, closing one of the major roadmap items.

Since 1.3.40, `typeOf()` was available on the JVM platform as an experimental API. Now you can use it in any Kotlin platform and get `KType` representation of any Kotlin type that the compiler can infer:

```
inline fun <reified T> renderType(): String {
    val type = typeOf<T>()
    return type.toString()
}

fun main() {
    val fromExplicitType = typeOf<Int>()
    val fromReifiedType = renderType<List<Int>>()
}
```

稳定版集合构建器

In Kotlin 1.6.0, collection builder functions have been promoted to Stable. Collections returned by collection builders are now serializable in their read-only state.

You can now use `buildMap()`, `buildList()`, and `buildSet()` without the opt-in annotation:

```
fun main() {
//sampleStart
    val x = listOf('b', 'c')
    val y = buildList {
        add('a')
        addAll(x)
        add('d')
    }
    println(y)  // [a, b, c, d]
//sampleEnd
}
```

稳定版 Duration API

The Duration class for representing duration amounts in different time units has been promoted to Stable. In 1.6.0, the Duration API has received the following changes:

- The first component of the `toComponents()` function that decomposes the duration into days, hours, minutes, seconds, and nanoseconds now has the `Long` type instead of `Int`. Before, if the value didn't fit into the `Int` range, it was coerced into that range. With the `Long` type, you can decompose any value in the duration range without cutting off the values that don't fit into `Int`.
- The `DurationUnit` enum is now standalone and not a type alias of `java.util.concurrent.TimeUnit` on the JVM. We haven't found any convincing cases in which having `typealias DurationUnit = TimeUnit` could be useful. Also, exposing the `TimeUnit` API through a type alias might confuse `DurationUnit` users.
- In response to community feedback, we're bringing back extension properties like `Int.seconds`. But we'd like to limit their applicability, so we put them into the companion of the `Duration` class. While the IDE can still propose extensions in completion and automatically insert an import from the companion, in the future we plan to limit this behavior to cases when the `Duration` type is expected.

```
import kotlin.time.Duration.Companion.seconds

fun main() {
//sampleStart
    val duration = 10000
    println("There are ${duration.seconds.inWholeMinutes} minutes in $duration seconds")
    // There are 166 minutes in 10000 seconds
//sampleEnd
}
```

We suggest replacing previously introduced companion functions, such as `Duration.seconds(Int)`, and deprecated top-level extensions like `Int.seconds` with new extensions in `Duration.Companion`.

> Such a replacement may cause ambiguity between old top-level extensions and new companion extensions. Be sure to use the wildcard import of the kotlin.time package – `import kotlin.time.*` – before doing automated migration.
>
> 

按 Regex 拆分为序列

The `Regex.splitToSequence(CharSequence)` and `CharSequence.splitToSequence(Regex)` functions are promoted to Stable. They split the string around matches of the given regex, but return the result as a Sequence so that all operations on this result are executed lazily:

```
fun main() {
//sampleStart
    val colorsText = "green, red, brown&blue, orange, pink&green"
    val regex = "[,\\s]+".toRegex()
    val mixedColor = regex.splitToSequence(colorsText)
    // or
    // val mixedColor = colorsText.splitToSequence(regex)
        .onEach { println(it) }
        .firstOrNull { it.contains('&') }
    println(mixedColor) // "brown&blue"
//sampleEnd
}
```

整数的循环移位运算

In Kotlin 1.6.0, the `rotateLeft()` and `rotateRight()` functions for bit manipulations became Stable. The functions rotate the binary representation of the number left or right by a specified number of bits:

```
fun main() {
//sampleStart
    val number: Short = 0b10001
    println(number
        .rotateRight(2)
        .toString(radix = 2)) // 100000000000100
    println(number
        .rotateLeft(2)
        .toString(radix = 2))  // 1000100
//sampleEnd
}
```

JS 平台 replace() 与 replaceFirst() 的变更

Before Kotlin 1.6.0, the `replace()` and `replaceFirst()` Regex functions behaved differently in Java and JS when the replacement string contained a group reference. To make the behavior consistent across all target platforms, we've changed their implementation in JS.

Occurrences of `${name}` or `$index` in the replacement string are substituted with the subsequences corresponding to the captured groups with the specified index or a name:

- `$index` – the first digit after '$' is always treated as a part of the group reference. Subsequent digits are incorporated into the `index` only if they form a valid group reference.Only digits

'0'–'9' are considered potential components of the group reference. Note that indexes of captured groups start from '1'. The group with index '0' stands for the whole match.

- `${name}` – the `name` can consist of Latin letters 'a'–'z', 'A'–'Z', or digits '0'–'9'. The first character must be a letter.

  > Named groups in replacement patterns are currently supported only on the JVM.
  >
  

- To include the succeeding character as a literal in the replacement string, use the backslash character `\`:

```
fun main() {
//sampleStart
    println(Regex("(.+)").replace("Kotlin", """\$ $1""")) // $ Kotlin
    println(Regex("(.+)").replaceFirst("1.6.0", """\\ $1""")) // \ 1.6.0
//sampleEnd
}
```

```
You can use [`Regex.escapeReplacement()`](https://kotlinlang.org/api/latest/jvm/stdlib
/kotlin.text/-regex/escape-replacement.html) if the replacement string has to be treat
ed as a literal string.
```

既有 API 的改进

- Version 1.6.0 added the infix extension function for `Comparable.compareTo()`. You can now use the infix form for comparing two objects for order:

```
class WrappedText(val text: String) : Comparable<WrappedText> {
    override fun compareTo(other: WrappedText): Int =
        this.text compareTo other.text
}
```

- `Regex.replace()` in JS is now also not inline to unify its implementation across all platforms.
- The `compareTo()` and `equals()` String functions, as well as the `isBlank()` CharSequence function now behave in JS exactly the same way they do on the JVM. Previously there were deviations when it came to non-ASCII characters.

弃用项

In Kotlin 1.6.0, we're starting the deprecation cycle with a warning for some JS-only stdlib API.

concat()、 match() 与 matches() 字符串函数

- To concatenate the string with the string representation of a given other object, use `plus()` instead of `concat()`.
- To find all occurrences of a regular expression within the input, use `findAll()` of the Regex class instead of `String.match(regex: String)`.
- To check if the regular expression matches the entire input, use `matches()` of the Regex class instead of `String.matches(regex: String)`.

采用比较函数对数组排序（sort()）

We've deprecated the `Array<out T>.sort()` function and the inline functions `ByteArray.sort()`, `ShortArray.sort()`, `IntArray.sort()`, `LongArray.sort()`, `FloatArray.sort()`, `DoubleArray.sort()`, and `CharArray.sort()`, which sorted arrays following the order passed by the comparison function. Use other standard library functions for array sorting.

See the collection ordering section for reference.

工具

Kover——Kotlin 的代码覆盖工具

> The Kover Gradle plugin is Experimental. We would appreciate your feedback on it in GitHub.

With Kotlin 1.6.0, we're introducing Kover – a Gradle plugin for the IntelliJ and JaCoCo Kotlin code coverage agents. It works with all language constructs, including inline functions.

Learn more about Kover on its GitHub repository or in this video:

YouTube 视频：Kover – The Code Coverage Plugin

Coroutines 1.6.0-RC

`kotlinx.coroutines` 1.6.0-RC is out with multiple features and improvements:

- Support for the new Kotlin/Native memory manager
- Introduction of dispatcher *views* API, which allows limiting parallelism without creating additional threads
- Migrating from Java 6 to Java 8 target
- `kotlinx-coroutines-test` with the new reworked API and multiplatform support
- Introduction of `CopyableThreadContextElement`, which gives coroutines a thread-safe write access to `ThreadLocal` variables

Learn more in the changelog.

迁移到 Kotlin 1.6.0

IntelliJ IDEA and Android Studio will suggest updating the Kotlin plugin to 1.6.0 once it is available.

To migrate existing projects to Kotlin 1.6.0, change the Kotlin version to `1.6.0` and reimport your Gradle or Maven project. Learn how to update to Kotlin 1.6.0.

To start a new project with Kotlin 1.6.0, update the Kotlin plugin and run the Project Wizard from **File** | **New** | **Project**.

The new command-line compiler is available for download on the GitHub release page.

Kotlin 1.6.0 is a feature release and can, therefore, bring changes that are incompatible with your code written for earlier versions of the language. Find the detailed list of such changes in the Compatibility Guide for Kotlin 1.6.

Kotlin 1.5.30 的新特性

发布于：*2021-08-24*

Kotlin 1.5.30 offers language updates including previews of future changes, various improvements in platform support and tooling, and new standard library functions.

Here are some major improvements:

- Language features, including experimental sealed `when` statements, changes in using opt-in requirement, and others
- Native support for Apple silicon
- Kotlin/JS IR backend reaches Beta
- Improved Gradle plugin experience

You can also find a short overview of the changes in the release blog post and this video:

YouTube 视频：Kotlin 1.5.30

语言特性

Kotlin 1.5.30 is presenting previews of future language changes and bringing improvements to the opt-in requirement mechanism and type inference:

- 对于密封类与布尔值主语穷尽 when 语句
- 挂起函数作为超类型
- 对隐式用到实验性 API 要求选择加入
- 对使用选择加入要求的注解不同目标的变更
- 递归泛型类型的类型推断改进
- 消除构建器推断限制

对于密封类与布尔值主语穷尽 when 语句

> Support for sealed (exhaustive) when statements is Experimental. It may be dropped or changed at any time. Opt-in is required (see the details below), and you should use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack.
>
> 

An *exhaustive* `when` statement contains branches for either all possible types or values of its subject, or for certain types and includes an `else` branch to cover any remaining cases.

We're planning to prohibit non-exhaustive `when` statements soon to make the behavior consistent with `when` expressions. To ensure smooth migration, you can configure the compiler to report warnings about non-exhaustive `when` statements with a sealed class or a Boolean. Such warnings will appear by default in Kotlin 1.6 and will become errors later.

> Enums already get a warning.
>
> 

```
sealed class Mode {
    object ON : Mode()
    object OFF : Mode()
}

fun main() {
    val x: Mode = Mode.ON
    when (x) {
        Mode.ON -> println("ON")
    }
// WARNING: Non exhaustive 'when' statements on sealed classes/interfaces
// will be prohibited in 1.7, add an 'OFF' or 'else' branch instead

    val y: Boolean = true
    when (y) {
        true -> println("true")
    }
// WARNING: Non exhaustive 'when' statements on Booleans will be prohibited
// in 1.7, add a 'false' or 'else' branch instead
}
```

To enable this feature in Kotlin 1.5.30, use language version `1.6`. You can also change the warnings to errors by enabling progressive mode.

> **【Kotlin】**

```
kotlin {
    sourceSets.all {
        languageSettings.apply {
```

```
            languageVersion = "1.6"
            //progressiveMode = true // false by default
        }
    }
}
```

【Groovy】

```
kotlin {
    sourceSets.all {
        languageSettings {
            languageVersion = '1.6'
            //progressiveMode = true // false by default
        }
    }
}
```

挂起函数作为超类型

> Support for suspending functions as supertypes is Experimental. It may be dropped or changed at any time. Opt-in is required (see the details below), and you should use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack.
>
> 

Kotlin 1.5.30 provides a preview of the ability to use a `suspend` functional type as a supertype with some limitations.

```
class MyClass: suspend () -> Unit {
    override suspend fun invoke() { TODO() }
}
```

Use the `-language-version 1.6` compiler option to enable the feature:

【Kotlin】

```
kotlin {
    sourceSets.all {
        languageSettings.apply {
            languageVersion = "1.6"
        }
```

```
    }
}
```

【Groovy】

```
kotlin {
    sourceSets.all {
        languageSettings {
            languageVersion = '1.6'
        }
    }
}
```

The feature has the following restrictions:

- You can't mix an ordinary functional type and a `suspend` functional type as supertype. This is because of the implementation details of `suspend` functional types in the JVM backend. They are represented in it as ordinary functional types with a marker interface. Because of the marker interface, there is no way to tell which of the superinterfaces are suspended and which are ordinary.
- You can't use multiple `suspend` functional supertypes. If there are type checks, you also can't use multiple ordinary functional supertypes.

对隐式用到实验性 API 要求选择加入

> The opt-in requirement mechanism is Experimental. It may change at any time. See how to opt-in. Use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack.
>
> 

The author of a library can mark an experimental API as requiring opt-in to inform users about its experimental state. The compiler raises a warning or error when the API is used and requires explicit consent to suppress it.

In Kotlin 1.5.30, the compiler treats any declaration that has an experimental type in the signature as experimental. Namely, it requires opt-in even for implicit usages of an experimental API. For example, if the function's return type is marked as an experimental API element, a usage of the function requires you to opt-in even if the declaration is not marked as requiring an opt-in explicitly.

```
// Library code
```

```
@RequiresOptIn(message = "This API is experimental.")
@Retention(AnnotationRetention.BINARY)
@Target(AnnotationTarget.CLASS)
annotation class MyDateTime // Opt-in requirement annotation

@MyDateTime
class DateProvider // A class requiring opt-in

// Client code

// Warning: experimental API usage
fun createDateSource(): DateProvider { /* ... */ }

fun getDate(): Date {
    val dateSource = createDateSource() // Also warning: experimental API usage
    // ...
}
```

Learn more about opt-in requirements.

对使用选择加入要求的注解不同目标的变更

> The opt-in requirement mechanism is Experimental. It may change at any time. See how to opt-in. Use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack.
>
> 

Kotlin 1.5.30 presents new rules for using and declaring opt-in requirement annotations on different targets. The compiler now reports an error for use cases that are impractical to handle at compile time. In Kotlin 1.5.30:

- Marking local variables and value parameters with opt-in requirement annotations is forbidden at the use site.
- Marking override is allowed only if its basic declaration is also marked.
- Marking backing fields and getters is forbidden. You can mark the basic property instead.
- Setting `TYPE` and `TYPE_PARAMETER` annotation targets is forbidden at the opt-in requirement annotation declaration site.

Learn more about opt-in requirements.

递归泛型类型的类型推断改进

In Kotlin and Java, you can define a recursive generic type, which references itself in its type parameters. In Kotlin 1.5.30, the Kotlin compiler can infer a type argument based only on upper bounds of the corresponding type parameter if it is a recursive generic. This makes it possible to create various patterns with recursive generic types that are often used in Java to make builder APIs.

```
// Kotlin 1.5.20
val containerA = PostgreSQLContainer<Nothing>(DockerImageName.parse("postgres:13-alpin
e")).apply {
    withDatabaseName("db")
    withUsername("user")
    withPassword("password")
    withInitScript("sql/schema.sql")
}

// Kotlin 1.5.30
val containerB = PostgreSQLContainer(DockerImageName.parse("postgres:13-alpine"))
    .withDatabaseName("db")
    .withUsername("user")
    .withPassword("password")
    .withInitScript("sql/schema.sql")
```

You can enable the improvements by passing the `-Xself-upper-bound-inference` or the `-language-version 1.6` compiler options. See other examples of newly supported use cases in this YouTrack ticket.

消除构建器推断限制

Builder inference is a special kind of type inference that allows you to infer the type arguments of a call based on type information from other calls inside its lambda argument. This can be useful when calling generic builder functions such as `buildList()` or `sequence()`: `buildList { add("string") }`.

Inside such a lambda argument, there was previously a limitation on using the type information that the builder inference tries to infer. This means you can only specify it and cannot get it. For example, you cannot call `get()` inside a lambda argument of `buildList()` without explicitly specified type arguments.

Kotlin 1.5.30 removes these limitations with the `-Xunrestricted-builder-inference` compiler option. Add this option to enable previously prohibited calls inside a lambda argument of generic builder functions:

```
@kotlin.ExperimentalStdlibApi
val list = buildList {
    add("a")
    add("b")
    set(1, null)
    val x = get(1)
    if (x != null) {
        removeAt(1)
    }
}
```

```
@kotlin.ExperimentalStdlibApi
val map = buildMap {
    put("a", 1)
    put("b", 1.1)
    put("c", 2f)
}
```

Also, you can enable this feature with the `-language-version 1.6` compiler option.

Kotlin/JVM

With Kotlin 1.5.30, Kotlin/JVM receives the following features:

- 注解类的实例化
- 改进了对可空性注解配置的支持

See the Gradle section for Kotlin Gradle plugin updates on the JVM platform.

注解类的实例化

> 注解类的实例化 is Experimental. It may be dropped or changed at any time. Opt-in is required (see the details below), and you should use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack.
>
> 

With Kotlin 1.5.30 you can now call constructors of annotation classes in arbitrary code to obtain a resulting instance. This feature covers the same use cases as the Java convention that allows the implementation of an annotation interface.

```
annotation class InfoMarker(val info: String)

fun processInfo(marker: InfoMarker) = ...

fun main(args: Array<String>) {
    if (args.size != 0)
        processInfo(getAnnotationReflective(args))
    else
        processInfo(InfoMarker("default"))
}
```

Use the `-language-version 1.6` compiler option to enable this feature. Note that all current annotation class limitations, such as restrictions to define non-`val` parameters or members different from secondary constructors, remain intact.

Learn more about instantiation of annotation classes in this KEEP

改进了对可空性注解配置的支持

The Kotlin compiler can read various types of nullability annotations to get nullability information from Java. This information allows it to report nullability mismatches in Kotlin when calling Java code.

In Kotlin 1.5.30, you can specify whether the compiler reports a nullability mismatch based on the information from specific types of nullability annotations. Just use the compiler option `-Xnullability-annotations=@<package-name>:<report-level>`. In the argument, specify the fully qualified nullability annotations package and one of these report levels:

- `ignore` to ignore nullability mismatches
- `warn` to report warnings
- `strict` to report errors.

See the full list of supported nullability annotations along with their fully qualified package names.

Here is an example showing how to enable error reporting for the newly supported RxJava 3 nullability annotations: `-Xnullability-annotations=@io.reactivex.rxjava3.annotations:strict`. Note that all such nullability mismatches are warnings by default.

Kotlin/Native

Kotlin/Native has received various changes and improvements:

- Apple silicon 支持
- 改进了用于 CocoaPods Gradle 插件的 Kotlin DSL
- 与 Swift 5.5 async/await 的实验性互操作
- 改进了对象与伴生对象的 Swift/Objective-C 映射
- 对于 MinGW 目标弃用了链接到 DLL 而未导入库的用法

Apple silicon 支持

Kotlin 1.5.30 introduces native support for Apple silicon.

Previously, the Kotlin/Native compiler and tooling required the Rosetta translation environment for working on Apple silicon hosts. In Kotlin 1.5.30, the translation environment is no longer needed – the compiler and tooling can run on Apple silicon hardware without requiring any additional actions.

We've also introduced new targets that make Kotlin code run natively on Apple silicon:

- `macosArm64`

- `iosSimulatorArm64`
- `watchosSimulatorArm64`
- `tvosSimulatorArm64`

They are available on both Intel-based and Apple silicon hosts. All existing targets are available on Apple silicon hosts as well.

Note that in 1.5.30 we provide only basic support for Apple silicon targets in the `kotlin-multiplatform` Gradle plugin. Particularly, the new simulator targets aren't included in the `ios`, `tvos`, and `watchos` target shortcuts. We will keep working to improve the user experience with the new targets.

改进了用于 CocoaPods Gradle 插件的 Kotlin DSL

用于 Kotlin/Native frameworks 的新的参数

Kotlin 1.5.30 introduces the improved CocoaPods Gradle plugin DSL for Kotlin/Native frameworks. In addition to the name of the framework, you can specify other parameters in the Pod configuration:

- Specify the dynamic or static version of the framework
- Enable export dependencies explicitly
- Enable Bitcode embedding

To use the new DSL, update your project to Kotlin 1.5.30, and specify the parameters in the `cocoapods` section of your `build.gradle(.kts)` file:

```
cocoapods {
    frameworkName = "MyFramework" // This property is deprecated
    // and will be removed in future versions
    // New DSL for framework configuration:
    framework {
        // All Framework properties are supported
        // Framework name configuration. Use this property instead of
        // deprecated 'frameworkName'
        baseName = "MyFramework"
        // Dynamic framework support
        isStatic = false
        // Dependency export
        export(project(":anotherKMMModule"))
        transitiveExport = false // This is default.
        // Bitcode embedding
        embedBitcode(BITCODE)
    }
}
```

支持 Xcode 配置的自定义名称

The Kotlin CocoaPods Gradle plugin supports custom names in the Xcode build configuration. It will also help you if you're using special names for the build configuration in Xcode, for example `Staging`.

To specify a custom name, use the `xcodeConfigurationToNativeBuildType` parameter in the `cocoapods` section of your `build.gradle(.kts)` file:

```
cocoapods {
    // Maps custom Xcode configuration to NativeBuildType
    xcodeConfigurationToNativeBuildType["CUSTOM_DEBUG"] = NativeBuildType.DEBUG
    xcodeConfigurationToNativeBuildType["CUSTOM_RELEASE"] = NativeBuildType.RELEASE
}
```

This parameter will not appear in the Podspec file. When Xcode runs the Gradle build process, the Kotlin CocoaPods Gradle plugin will select the necessary native build type.

> There's no need to declare the `Debug` and `Release` configurations because they are supported by default.
>
> 

与 Swift 5.5 async/await 的实验性互操作

> Concurrency interoperability with Swift async/await is Experimental. It may be dropped or changed at any time. You should use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack.
>
> 

We added support for calling Kotlin's suspending functions from Objective-C and Swift in 1.4.0, and now we're improving it to keep up with a new Swift 5.5 feature – concurrency with `async` and `await` modifiers.

The Kotlin/Native compiler now emits the `_Nullable_result` attribute in the generated Objective-C headers for suspending functions with nullable return types. This makes it possible to call them from Swift as `async` functions with the proper nullability.

Note that this feature is experimental and can be affected in the future by changes in both Kotlin and Swift. For now, we're offering a preview of this feature that has certain limitations, and we are eager to hear what you think. Learn more about its current state and leave your feedback in this YouTrack issue.

改进了对象与伴生对象的 Swift/Objective-C 映射

Getting objects and companion objects can now be done in a way that is more intuitive for native iOS developers. For example, if you have the following objects in Kotlin:

```
object MyObject {
    val x = "Some value"
}

class MyClass {
    companion object {
        val x = "Some value"
    }
}
```

To access them in Swift, you can use the `shared` and `companion` properties:

```
MyObject.shared
MyObject.shared.x
MyClass.companion
MyClass.Companion.shared
```

Learn more about Swift/Objective-C interoperability.

对于 MinGW 目标弃用了链接到 DLL 而未导入库的用法

LLD is a linker from the LLVM project, which we plan to start using in Kotlin/Native for MinGW targets because of its benefits over the default ld.bfd – primarily its better performance.

However, the latest stable version of LLD doesn't support direct linkage against DLL for MinGW (Windows) targets. Such linkage requires using import libraries. Although they aren't needed with Kotlin/Native 1.5.30, we're adding a warning to inform you that such usage is incompatible with LLD that will become the default linker for MinGW in the future.

Please share your thoughts and concerns about the transition to the LLD linker in this YouTrack issue.

Kotlin 多平台

1.5.30 brings the following notable updates to Kotlin Multiplatform:

- 能在共享的原生代码中使用自定义 `cinterop` 库
- 对 XCFrameworks 的支持
- Android 构件的新版默认发布设置

能在共享的原生代码中使用自定义 cinterop 库

Kotlin Multiplatform gives you an option to use platform-dependent interop libraries in shared source sets. Before 1.5.30, this worked only with platform libraries shipped with Kotlin/Native distribution. Starting from 1.5.30, you can use it with your custom `cinterop` libraries. To enable this feature, add the `kotlin.mpp.enableCInteropCommonization=true` property in your `gradle.properties`:

```
kotlin.mpp.enableGranularSourceSetsMetadata=true
kotlin.native.enableDependencyPropagation=false
kotlin.mpp.enableCInteropCommonization=true
```

对 XCFrameworks 的支持

All Kotlin Multiplatform projects can now have XCFrameworks as an output format. Apple introduced XCFrameworks as a replacement for universal (fat) frameworks. With the help of XCFrameworks you:

- Can gather logic for all the target platforms and architectures in a single bundle.
- Don't need to remove all unnecessary architectures before publishing the application to the App Store.

XCFrameworks is useful if you want to use your Kotlin framework for devices and simulators on Apple M1.

To use XCFrameworks, update your `build.gradle(.kts)` script:

> 【Kotlin】

```
import org.jetbrains.kotlin.gradle.plugin.mpp.apple.XCFramework

plugins {
    kotlin("multiplatform")
}

kotlin {
    val xcf = XCFramework()

    ios {
        binaries.framework {
            baseName = "shared"
            xcf.add(this)
        }
    }
    watchos {
        binaries.framework {
            baseName = "shared"
            xcf.add(this)
```

```
        }
    }
    tvos {
        binaries.framework {
            baseName = "shared"
            xcf.add(this)
        }
    }
}
```

【Groovy】

```
import org.jetbrains.kotlin.gradle.plugin.mpp.apple.XCFrameworkConfig

plugins {
    id 'org.jetbrains.kotlin.multiplatform'
}

kotlin {
    def xcf = new XCFrameworkConfig(project)

    ios {
        binaries.framework {
            baseName = "shared"
            xcf.add(it)
        }
    }
    watchos {
        binaries.framework {
            baseName = "shared"
            xcf.add(it)
        }
    }
    tvos {
        binaries.framework {
            baseName = "shared"
            xcf.add(it)
        }
    }
}
```

When you declare XCFrameworks, these new Gradle tasks will be registered:

- `assembleXCFramework`
- `assembleDebugXCFramework` (additionally debug artifact that contains dSYMs)
- `assembleReleaseXCFramework`

Learn more about XCFrameworks in this WWDC video.

Android 构件的新版默认发布设置

Using the `maven-publish` Gradle plugin, you can publish your multiplatform library for the Android target by specifying Android variant names in the build script. The Kotlin Gradle plugin will generate publications automatically.

Before 1.5.30, the generated publication metadata included the build type attributes for every published Android variant, making it compatible only with the same build type used by the library consumer. Kotlin 1.5.30 introduces a new default publishing setup:

- If all Android variants that the project publishes have the same build type attribute, then the published variants won't have the build type attribute and will be compatible with any build type.
- If the published variants have different build type attributes, then only those with the `release` value will be published without the build type attribute. This makes the release variants compatible with any build type on the consumer side, while non-release variants will only be compatible with the matching consumer build types.

To opt-out and keep the build type attributes for all variants, you can set this Gradle property: `kotlin.android.buildTypeAttribute.keep=true`.

Kotlin/JS

Two major improvements are coming to Kotlin/JS with 1.5.30:

- JS IR 编译器后端达到 Beta 版
- 为使用 Kotlin/JS IR 后端的应用程序提供更好的调试体验

JS IR 编译器后端达到 Beta 版

The IR-based compiler backend for Kotlin/JS, which was introduced in 1.4.0 in Alpha, has reached Beta.

Previously, we published the migration guide for the JS IR backend to help you migrate your projects to the new backend. Now we would like to present the Kotlin/JS Inspection Pack IDE plugin, which displays the required changes directly in IntelliJ IDEA.

为使用 Kotlin/JS IR 后端的应用程序提供更好的调试体验

Kotlin 1.5.30 brings JavaScript source map generation for the Kotlin/JS IR backend. This will improve the Kotlin/JS debugging experience when the IR backend is enabled, with full debugging support that includes breakpoints, stepping, and readable stack traces with proper source references.

Learn how to debug Kotlin/JS in the browser or IntelliJ IDEA Ultimate.

Gradle

As a part of our mission to improve the Kotlin Gradle plugin user experience, we've implemented the following features:

- 支持 Java toolchains, which includes an ability to specify a JDK home with the `UsesKotlinJavaToolchain` interface for older Gradle versions
- 显式指定 Kotlin 守护进程 JVM 参数的更简单方式

支持 Java toolchains

Gradle 6.7 introduced the "Java toolchains support" feature. Using this feature, you can:

- Run compilations, tests, and executables using JDKs and JREs that are different from the Gradle ones.
- Compile and test code with an unreleased language version.

With toolchains support, Gradle can autodetect local JDKs and install missing JDKs that Gradle requires for the build. Now Gradle itself can run on any JDK and still reuse the build cache feature.

The Kotlin Gradle plugin supports Java toolchains for Kotlin/JVM compilation tasks. A Java toolchain:

- Sets the `jdkHome` option available for JVM targets.

  > The ability to set the `jdkHome` option directly has been deprecated.
  >
  > 

- Sets the `kotlinOptions.jvmTarget` to the toolchain's JDK version if the user didn't set the `jvmTarget` option explicitly. If the toolchain is not configured, the `jvmTarget` field uses the default value. Learn more about JVM target compatibility.

- Affects which JDK `kapt` workers are running on.

Use the following code to set a toolchain. Replace the placeholder `<MAJOR_JDK_VERSION>` with the JDK version you would like to use:

> 【Kotlin】

```
kotlin {
    jvmToolchain {
        (this as JavaToolchainSpec).languageVersion.set(JavaLanguageVersion.of(<MAJOR_
JDK_VERSION>)) // "8"
    }
}
```

> 【Groovy】

```
kotlin {
    jvmToolchain {
        languageVersion.set(JavaLanguageVersion.of(<MAJOR_JDK_VERSION>)) // "8"
    }
}
```

Note that setting a toolchain via the `kotlin` extension will update the toolchain for Java compile tasks as well.

You can set a toolchain via the `java` extension, and Kotlin compilation tasks will use it:

```
java {
    toolchain {
        languageVersion.set(JavaLanguageVersion.of(<MAJOR_JDK_VERSION>)) // "8"
    }
}
```

For information about setting any JDK version for `KotlinCompile` tasks, look through the docs about setting the JDK version with the Task DSL.

For Gradle versions from 6.1 to 6.6, use the `UsesKotlinJavaToolchain` interface to set the JDK home.

能够使用 UsesKotlinJavaToolchain 接口指定 JDK home

All Kotlin tasks that support setting the JDK via `kotlinOptions` now implement the `UsesKotlinJavaToolchain` interface. To set the JDK home, put a path to your JDK and replace the `<JDK_VERSION>` placeholder:

> 【Kotlin】

```
project.tasks
    .withType<UsesKotlinJavaToolchain>()
    .configureEach {
        it.kotlinJavaToolchain.jdk.use(
            "/path/to/local/jdk",
            JavaVersion.<LOCAL_JDK_VERSION>
        )
    }
```

> 【Groovy】

```
project.tasks
    .withType(UsesKotlinJavaToolchain.class)
    .configureEach {
        it.kotlinJavaToolchain.jdk.use(
            '/path/to/local/jdk',
            JavaVersion.<LOCAL_JDK_VERSION>
        )
    }
```

Use the `UsesKotlinJavaToolchain` interface for Gradle versions from 6.1 to 6.6. Starting from Gradle 6.7, use the Java toolchains instead.

When using this feature, note that kapt task workers will only use process isolation mode, and the `kapt.workers.isolation` property will be ignored.

显式指定 Kotlin 守护进程 JVM 参数的更简单方式

In Kotlin 1.5.30, there's a new logic for the Kotlin daemon's JVM arguments. Each of the options in the following list overrides the ones that came before it:

- If nothing is specified, the Kotlin daemon inherits arguments from the Gradle daemon (as before). For example, in the `gradle.properties` file:

  ```
  org.gradle.jvmargs=-Xmx1500m -Xms=500m
  ```

- If the Gradle daemon's JVM arguments have the `kotlin.daemon.jvm.options` system property, use it as before:

```
org.gradle.jvmargs=-Dkotlin.daemon.jvm.options=-Xmx1500m -Xms=500m
```

- You can add the `kotlin.daemon.jvmargs` property in the `gradle.properties` file:

```
kotlin.daemon.jvmargs=-Xmx1500m -Xms=500m
```

- You can specify arguments in the `kotlin` extension:

> 【Kotlin】

```
kotlin {
    kotlinDaemonJvmArgs = listOf("-Xmx486m", "-Xms256m", "-XX:+UseParallelGC")
}
```

> 【Groovy】

```
kotlin {
    kotlinDaemonJvmArgs = ["-Xmx486m", "-Xms256m", "-XX:+UseParallelGC"]
}
```

- You can specify arguments for a specific task:

> 【Kotlin】

```
tasks
    .matching { it.name == "compileKotlin" && it is CompileUsingKotlinDaemon }
    .configureEach {
        (this as CompileUsingKotlinDaemon).kotlinDaemonJvmArguments.set(listOf("-X
mx486m", "-Xms256m", "-XX:+UseParallelGC"))
    }
```

> 【Groovy】

```
tasks
    .matching {
```

```
            it.name == "compileKotlin" && it instanceof CompileUsingKotlinDaemon
        }
        .configureEach {
            kotlinDaemonJvmArguments.set(["-Xmx1g", "-Xms512m"])
        }
```

```
> In this case a new Kotlin daemon instance can start on task execution. Learn more ab
out [the Kotlin daemon's interactions with JVM arguments](gradle-compilation-and-cache
s.md#setting-kotlin-daemons-jvm-arguments).
>
![](/images/note.png)
```

For more information about the Kotlin daemon, see the Kotlin daemon and using it with Gradle.

标准库

Kotlin 1.5.30 is bringing improvements to the standard library's `Duration` and `Regex` APIs:

- 变更 `Duration.toString()` 输出
- 由 String 解析 Duration
- 在特定位置匹配 Regex
- 按 Regex 拆分为序列

变更 Duration.toString() 输出

> The Duration API is Experimental. It may be dropped or changed at any time. Use it only for evaluation purposes. We would appreciate hearing your feedback on it in YouTrack.
>
> 

Before Kotlin 1.5.30, the `Duration.toString()` function would return a string representation of its argument expressed in the unit that yielded the most compact and readable number value. From now on, it will return a string value expressed as a combination of numeric components, each in its own unit. Each component is a number followed by the unit's abbreviated name: `d`, `h`, `m`, `s`. For example:

Example of function call	Previous output	Current output
Duration.days(45).toString()	`45.0d`	`45d`
Duration.days(1.5).toString()	`36.0h`	`1d 12h`
Duration.minutes(1230).toString()	`20.5h`	`20h 30m`

Duration.minutes(2415).toString()	`40.3h`	`1d 16h 15m`
Duration.minutes(920).toString()	`920m`	`15h 20m`
Duration.seconds(1.546).toString()	`1.55s`	`1.546s`
Duration.milliseconds(25.12).toString()	`25.1ms`	`25.12ms`

The way negative durations are represented has also been changed. A negative duration is prefixed with a minus sign (`-`), and if it consists of multiple components, it is surrounded with parentheses: `-12m` and `-(1h 30m)` .

Note that small durations of less than one second are represented as a single number with one of the subsecond units. For example, `ms` (milliseconds), `us` (microseconds), or `ns` (nanoseconds): `140.884ms` , `500us` , `24ns` . Scientific notation is no longer used to represent them.

If you want to express duration in a single unit, use the overloaded `Duration.toString(unit, decimals)` function.

> We recommend using `Duration.toIsoString()` in certain cases, including serialization and interchange. `Duration.toIsoString()` uses the stricter ISO-8601 format instead of `Duration.toString()` .
>
> 

由 String 解析 Duration

> The Duration API is Experimental. It may be dropped or changed at any time. Use it only for evaluation purposes. We would appreciate hearing your feedback on it in this issue.
>
> 

In Kotlin 1.5.30, there are new functions in the Duration API:

- `parse()` , which supports parsing the outputs of:
  - `toString()` .
  - `toString(unit, decimals)` .
  - `toIsoString()` .
- `parseIsoString()` , which only parses from the format produced by `toIsoString()` .
- `parseOrNull()` and `parseIsoStringOrNull()` , which behave like the functions above but return `null` instead of throwing `IllegalArgumentException` on invalid duration formats.

Here are some examples of `parse()` and `parseOrNull()` usages:

```
import kotlin.time.Duration
import kotlin.time.ExperimentalTime

@ExperimentalTime
```

```
fun main() {
//sampleStart
    val isoFormatString = "PT1H30M"
    val defaultFormatString = "1h 30m"
    val singleUnitFormatString = "1.5h"
    val invalidFormatString = "1 hour 30 minutes"
    println(Duration.parse(isoFormatString)) // "1h 30m"
    println(Duration.parse(defaultFormatString)) // "1h 30m"
    println(Duration.parse(singleUnitFormatString)) // "1h 30m"
    //println(Duration.parse(invalidFormatString)) // throws exception
    println(Duration.parseOrNull(invalidFormatString)) // "null"
//sampleEnd
}
```

And here are some examples of `parseIsoString()` and `parseIsoStringOrNull()` usages:

```
import kotlin.time.Duration
import kotlin.time.ExperimentalTime

@ExperimentalTime
fun main() {
//sampleStart
    val isoFormatString = "PT1H30M"
    val defaultFormatString = "1h 30m"
    println(Duration.parseIsoString(isoFormatString)) // "1h 30m"
    //println(Duration.parseIsoString(defaultFormatString)) // throws exception
    println(Duration.parseIsoStringOrNull(defaultFormatString)) // "null"
//sampleEnd
}
```

在特定位置匹配 Regex

> `Regex.matchAt()` and `Regex.matchesAt()` functions are Experimental. They may be dropped or changed at any time. Use them only for evaluation purposes. We would appreciate hearing your feedback on them in YouTrack.
>
> 

The new `Regex.matchAt()` and `Regex.matchesAt()` functions provide a way to check whether a regex has an exact match at a particular position in a `String` or `CharSequence`.

`matchesAt()` returns a boolean result:

```
fun main(){
//sampleStart
    val releaseText = "Kotlin 1.5.30 is released!"
    // regular expression: one digit, dot, one digit, dot, one or more digits
    val versionRegex = "\\d[.]\\d[.]\\d+".toRegex()
```

```
    println(versionRegex.matchesAt(releaseText, 0)) // "false"
    println(versionRegex.matchesAt(releaseText, 7)) // "true"
//sampleEnd
}
```

`matchAt()` returns the match if one is found or `null` if one isn't:

```
fun main(){
//sampleStart
    val releaseText = "Kotlin 1.5.30 is released!"
    val versionRegex = "\\d[.]\\d[.]\\d+".toRegex()
    println(versionRegex.matchAt(releaseText, 0)) // "null"
    println(versionRegex.matchAt(releaseText, 7)?.value) // "1.5.30"
//sampleEnd
}
```

按 Regex 拆分为序列

> `Regex.splitToSequence()` and `CharSequence.splitToSequence(Regex)` functions are Experimental. They may be dropped or changed at any time. Use them only for evaluation purposes. We would appreciate hearing your feedback on them in YouTrack.
>
> 

The new `Regex.splitToSequence()` function is a lazy counterpart of `split()`. It splits the string around matches of the given regex, but it returns the result as a Sequence so that all operations on this result are executed lazily.

```
fun main(){
//sampleStart
    val colorsText = "green, red , brown&blue, orange, pink&green"
    val regex = "[,\\s]+".toRegex()
    val mixedColor = regex.splitToSequence(colorsText)
        .onEach { println(it) }
        .firstOrNull { it.contains('&') }
    println(mixedColor) // "brown&blue"
//sampleEnd
}
```

A similar function was also added to `CharSequence`:

```
    val mixedColor = colorsText.splitToSequence(regex)
```

serialization 1.3.0-RC

`kotlinx.serialization` 1.3.0-RC is here with new JSON serialization capabilities:

- Java IO streams serialization
- Property-level control over default values
- An option to exclude null values from serialization
- Custom class discriminators in polymorphic serialization

Learn more in the changelog.

Kotlin 1.5.20 的新特性

发布于：*2021-06-24*

Kotlin 1.5.20 has fixes for issues discovered in the new features of 1.5.0, and it also includes various tooling improvements.

You can find an overview of the changes in the release blog post and this video:

YouTube 视频：Kotlin 1.5.20

Kotlin/JVM

Kotlin 1.5.20 is receiving the following updates on the JVM platform:

- 通过 invokedynamic 字符串连接
- JSpecify 可空性注解的支持
- 支持在 Kotlin 与 Java 代码都有的模块中调用 Java 的 Lombok 所生成代码

通过 invokedynamic 字符串连接

Kotlin 1.5.20 compiles string concatenations into dynamic invocations (`invokedynamic`) on JVM 9+ targets, thereby keeping up with modern Java versions. More precisely, it uses `StringConcatFactory.makeConcatWithConstants()` for string concatenation.

To switch back to concatenation via `StringBuilder.append()` used in previous versions, add the compiler option `-Xstring-concat=inline` .

Learn how to add compiler options in Gradle, Maven, and the command-line compiler.

JSpecify 可空性注解的支持

The Kotlin compiler can read various types of nullability annotations to pass nullability information from Java to Kotlin. Version 1.5.20 introduces support for the JSpecify project, which includes the standard unified set of Java nullness annotations.

With JSpecify, you can provide more detailed nullability information to help Kotlin keep null-safety interoperating with Java. You can set default nullability for the declaration, package, or module scope, specify parametric nullability, and more. You can find more details about this in the JSpecify user guide.

Here is the example of how Kotlin can handle JSpecify annotations:

```
// JavaClass.java
import org.jspecify.nullness.*;

@NullMarked
public class JavaClass {
  public String notNullableString() { return ""; }
  public @Nullable String nullableString() { return ""; }
}
```

```
// Test.kt
fun kotlinFun() = with(JavaClass()) {
  notNullableString().length // OK
  nullableString().length    // Warning: receiver nullability mismatch
}
```

In 1.5.20, all nullability mismatches according to the JSpecify-provided nullability information are reported as warnings. Use the `-Xjspecify-annotations=strict` and `-Xtype-enhancement-improvements-strict-mode` compiler options to enable strict mode (with error reporting) when working with JSpecify. Please note that the JSpecify project is under active development. Its API and implementation can change significantly at any time.

Learn more about null-safety and platform types.

支持在 Kotlin 与 Java 代码都有的模块中调用 Java 的 Lombok 所生成代码

> The Lombok compiler plugin is Experimental. It may be dropped or changed at any time. Use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack.
>
> 

Kotlin 1.5.20 introduces an experimental Lombok compiler plugin. This plugin makes it possible to generate and use Java's Lombok declarations within modules that have Kotlin and Java code. Lombok annotations work only in Java sources and are ignored if you use them in Kotlin code.

The plugin supports the following annotations:

- `@Getter` , `@Setter`
- `@NoArgsConstructor` , `@RequiredArgsConstructor` , and `@AllArgsConstructor`
- `@Data`
- `@With`
- `@Value`

We're continuing to work on this plugin. To find out the detailed current state, visit the Lombok compiler plugin's README.

Currently, we don't have plans to support the `@Builder` annotation. However, we can consider this if you vote for `@Builder` in YouTrack.

Learn how to configure the Lombok compiler plugin.

Kotlin/Native

Kotlin/Native 1.5.20 offers a preview of the new feature and the tooling improvements:

- 将 KDoc 注释导出到所生成的 Objective-C 头文件的选择加入项
- 编译器 bug 修复
- 提高了同一数组内 Array.copyInto() 的性能

将 KDoc 注释导出到所生成的 Objective-C 头文件的选择加入项

> The ability to export KDoc comments to generated Objective-C headers is Experimental. It may be dropped or changed at any time. Opt-in is required (see the details below), and you should use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack.
>
> 

You can now set the Kotlin/Native compiler to export the documentation comments (KDoc) from Kotlin code to the Objective-C frameworks generated from it, making them visible to the frameworks' consumers.

For example, the following Kotlin code with KDoc:

```
/**
 * Prints the sum of the arguments.
 * Properly handles the case when the sum doesn't fit in 32-bit integer.
 */
fun printSum(a: Int, b: Int) = println(a.toLong() + b)
```

produces the following Objective-C headers:

```
/**
 * Prints the sum of the arguments.
 * Properly handles the case when the sum doesn't fit in 32-bit integer.
 */
+ (void)printSumA:(int32_t)a b:(int32_t)b __attribute__((swift_name("printSum(a:b:)"))
);
```

This also works well with Swift.

To try out this ability to export KDoc comments to Objective-C headers, use the `-Xexport-kdoc` compiler option. Add the following lines to `build.gradle(.kts)` of the Gradle projects you want to export comments from:

> 【Kotlin】

```
kotlin {
    targets.withType<org.jetbrains.kotlin.gradle.plugin.mpp.KotlinNativeTarget> {
        compilations.get("main").kotlinOptions.freeCompilerArgs += "-Xexport-kdoc"
    }
}
```

> 【Groovy】

```
kotlin {
    targets.withType(org.jetbrains.kotlin.gradle.plugin.mpp.KotlinNativeTarget) {
        compilations.get("main").kotlinOptions.freeCompilerArgs += "-Xexport-kdoc"
    }
}
```

We would be very grateful if you would share your feedback with us using this YouTrack ticket.

编译器 bug 修复

The Kotlin/Native compiler has received multiple bug fixes in 1.5.20. You can find the complete list in the changelog.

There is an important bug fix that affects compatibility: in previous versions, string constants that contained incorrect UTF surrogate pairs were losing their values during compilation. Now such values are preserved. Application developers can safely update to 1.5.20 – nothing will break. However, libraries compiled with 1.5.20 are incompatible with earlier compiler versions. See this YouTrack issue for details.

提高了同一数组内 Array.copyInto() 的性能

We've improved the way `Array.copyInto()` works when its source and destination are the same array. Now such operations finish up to 20 times faster (depending on the number of objects being copied) due to memory management optimizations for this use case.

Kotlin/JS

With 1.5.20, we're publishing a guide that will help you migrate your projects to the new IR-based backend for Kotlin/JS.

JS IR 后端迁移指南

The new JS IR 后端迁移指南 identifies issues you may encounter during migration and provides solutions for them. If you find any issues that aren't covered in the guide, please report them to our issue tracker.

Gradle

Kotlin 1.5.20 introduces the following features that can improve the Gradle experience:

- 在 kapt 中缓存注解处理器的类加载器
- 弃用 `kotlin.parallel.tasks.in.project` 构建属性

在 kapt 中缓存注解处理器的类加载器

> 在 kapt 中缓存注解处理器的类加载器 is Experimental. It may be dropped or changed at any time. Use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack.
>
> 

There is now a new experimental feature that makes it possible to cache the classloaders of annotation processors in kapt. This feature can increase the speed of kapt for consecutive Gradle runs.

To enable this feature, use the following properties in your `gradle.properties` file:

```
# positive value will enable caching
# use the same value as the number of modules that use kapt
kapt.classloaders.cache.size=5

# disable for caching to work
kapt.include.compile.classpath=false
```

Learn more about kapt.

弃用 kotlin.parallel.tasks.in.project 构建属性

With this release, Kotlin parallel compilation is controlled by the Gradle parallel execution flag `--parallel`. Using this flag, Gradle executes tasks concurrently, increasing the speed of compiling tasks and utilizing the resources more efficiently.

You no longer need to use the `kotlin.parallel.tasks.in.project` property. This property has been deprecated and will be removed in the next major release.

标准库

Kotlin 1.5.20 changes the platform-specific implementations of several functions for working with characters and as a result brings unification across platforms:

- Kotlin/Native 与 Kotlin/JS 的 Char.digitToInt() 支持所有 Unicode 数字。
- 跨平台统一 Char.isLowerCase()/isUpperCase() 的实现。

Kotlin/Native 与 Kotlin/JS 的 Char.digitToInt() 支持所有 Unicode 数字

`Char.digitToInt()` returns the numeric value of the decimal digit that the character represents. Before 1.5.20, the function supported all Unicode digit characters only for Kotlin/JVM: implementations on the Native and JS platforms supported only ASCII digits.

From now, both with Kotlin/Native and Kotlin/JS, you can call `Char.digitToInt()` on any Unicode digit character and get its numeric representation.

```
fun main() {
//sampleStart
    val ten = '\u0661'.digitToInt() + '\u0039'.digitToInt() // ARABIC-INDIC DIGIT ONE
+ DIGIT NINE
    println(ten)
//sampleEnd
}
```

跨平台统一 Char.isLowerCase()/isUpperCase() 的实现

The functions `Char.isUpperCase()` and `Char.isLowerCase()` return a boolean value depending on the case of the character. For Kotlin/JVM, the implementation checks both the `General_Category` and the `Other_Uppercase` / `Other_Lowercase` Unicode properties.

Prior to 1.5.20, implementations for other platforms worked differently and considered only the general category. In 1.5.20, implementations are unified across platforms and use both properties to determine the character case:

```
fun main() {
//sampleStart
    val latinCapitalA = 'A' // has "Lu" general category
    val circledLatinCapitalA = ''  // has "Other_Uppercase" property
    println(latinCapitalA.isUpperCase() && circledLatinCapitalA.isUpperCase())
//sampleEnd
}
```

Kotlin 1.5.0 的新特性

发布于：*2021-05-05*

Kotlin 1.5.0 introduces new language features, stable IR-based JVM compiler backend, performance improvements, and evolutionary changes such as stabilizing experimental features and deprecating outdated ones.

You can also find an overview of the changes in the release blog post.

语言特性

Kotlin 1.5.0 brings stable versions of the new language features presented for preview in 1.4.30:

- JVM 记录类型支持
- 密封接口以及密封类改进
- 内联类

Detailed descriptions of these features are available in this blog post and the corresponding pages of Kotlin documentation.

JVM 记录类型支持

Java is evolving fast, and to make sure Kotlin remains interoperable with it, we've introduced support for one of its latest features – record classes.

Kotlin's support for JVM records includes bidirectional interoperability:

- In Kotlin code, you can use Java record classes like you would use typical classes with properties.
- To use a Kotlin class as a record in Java code, make it a `data` class and mark it with the `@JvmRecord` annotation.

```
@JvmRecord
data class User(val name: String, val age: Int)
```

Learn more about using JVM records in Kotlin.

YouTube 视频：Support for JVM Records in Kotlin 1.5.0

密封接口

Kotlin interfaces can now have the `sealed` modifier, which works on interfaces in the same way it works on classes: all implementations of a sealed interface are known at compile time.

```
sealed interface Polygon
```

You can rely on that fact, for example, to write exhaustive `when` expressions.

```
fun draw(polygon: Polygon) = when (polygon) {
   is Rectangle -> // ...
   is Triangle -> // ...
   // else is not needed - all possible implementations are covered
}
```

Additionally, sealed interfaces enable more flexible restricted class hierarchies because a class can directly inherit more than one sealed interface.

```
class FilledRectangle: Polygon, Fillable
```

Learn more about sealed interfaces.

YouTube 视频：Sealed Interfaces and Sealed Classes Improvements

包范围的密封类层次结构

Sealed classes can now have subclasses in all files of the same compilation unit and the same package. Previously, all subclasses had to appear in the same file.

Direct subclasses may be top-level or nested inside any number of other named classes, named interfaces, or named objects.

The subclasses of a sealed class must have a name that is properly qualified – they cannot be local or anonymous objects.

Learn more about sealed class hierarchies.

内联类

Inline classes are a subset of value-based classes that only hold values. You can use them as wrappers for a value of a certain type without the additional overhead that comes from using memory allocations.

Inline classes can be declared with the `value` modifier before the name of the class:

```
value class Password(val s: String)
```

The JVM backend also requires a special `@JvmInline` annotation:

```
@JvmInline
value class Password(val s: String)
```

The `inline` modifier is now deprecated with a warning.

Learn more about inline classes.

YouTube 视频：From Inline to Value Classes

Kotlin/JVM

Kotlin/JVM has received a number of improvements, both internal and user-facing. Here are the most notable among them:

- 稳定版 JVM IR 后端

- 新的默认 JVM 目标：1.8
- 采用 invokedynamic 的 SAM 适配器
- 采用 invokedynamic 的 lambda 表达式
- @JvmDefault 与旧版 Xjvm-default 模式的弃用
- 处理可空性注解的改进

稳定版 JVM IR 后端

The IR-based backend for the Kotlin/JVM compiler is now Stable and enabled by default.

Starting from Kotlin 1.4.0, early versions of the IR-based backend were available for preview, and it has now become the default for language version `1.5`. The old backend is still used by default for earlier language versions.

You can find more details about the benefits of the IR backend and its future development in this blog post.

If you need to use the old backend in Kotlin 1.5.0, you can add the following lines to the project's configuration file:

- In Gradle:

> 【Kotlin】

```
tasks.withType<org.jetbrains.kotlin.gradle.dsl.KotlinJvmCompile> {
  kotlinOptions.useOldBackend = true
}
```

> 【Groovy】

```
tasks.withType(org.jetbrains.kotlin.gradle.dsl.KotlinJvmCompile) {
 kotlinOptions.useOldBackend = true
}
```

- In Maven:

```
<configuration>
   <args>
       <arg>-Xuse-old-backend</arg>
   </args>
```

```
</configuration>
```

新的默认 JVM 目标：1.8

The default target version for Kotlin/JVM compilations is now `1.8`. The `1.6` target is deprecated.

If you need a build for JVM 1.6, you can still switch to this target. Learn how:

- in Gradle
- in Maven
- in the command-line compiler

采用 invokedynamic 的 SAM 适配器

Kotlin 1.5.0 now uses dynamic invocations (`invokedynamic`) for compiling SAM (Single Abstract Method) conversions:

- Over any expression if the SAM type is a Java interface
- Over lambda if the SAM type is a Kotlin functional interface

The new implementation uses `LambdaMetafactory.metafactory()` and auxiliary wrapper classes are no longer generated during compilation. This decreases the size of the application's JAR, which improves the JVM startup performance.

To roll back to the old implementation scheme based on anonymous class generation, add the compiler option `-Xsam-conversions=class` .

Learn how to add compiler options in Gradle, Maven, and the command-line compiler.

采用 invokedynamic 的 lambda 表达式

> Compiling plain Kotlin lambdas into invokedynamic is Experimental. It may be dropped or changed at any time. Opt-in is required (see details below), and you should use it only for evaluation purposes. We would appreciate hearing your feedback on it in YouTrack.
>
> 

Kotlin 1.5.0 is introducing experimental support for compiling plain Kotlin lambdas (which are not converted to an instance of a functional interface) into dynamic invocations (`invokedynamic`). The implementation produces lighter binaries by using `LambdaMetafactory.metafactory()` , which effectively generates the necessary classes at runtime. Currently, it has three limitations compared to ordinary lambda compilation:

- A lambda compiled into invokedynamic is not serializable.
- Calling `toString()` on such a lambda produces a less readable string representation.

- Experimental `reflect` API does not support lambdas created with `LambdaMetafactory`.

To try this feature, add the `-Xlambdas=indy` compiler option. We would be grateful if you could share your feedback on it using this YouTrack ticket.

Learn how to add compiler options in Gradle, Maven, and command-line compiler.

@JvmDefault 与旧版 Xjvm-default 模式的弃用

Prior to Kotlin 1.4.0, there was the `@JvmDefault` annotation along with `-Xjvm-default=enable` and `-Xjvm-default=compatibility` modes. They served to create the JVM default method for any particular non-abstract member in the Kotlin interface.

In Kotlin 1.4.0, we introduced the new `Xjvm-default` modes, which switch on default method generation for the whole project.

In Kotlin 1.5.0, we are deprecating `@JvmDefault` and the old Xjvm-default modes: `-Xjvm-default=enable` and `-Xjvm-default=compatibility`.

Learn more about default methods in the Java interop.

处理可空性注解的改进

Kotlin supports handling type nullability information from Java with nullability annotations. Kotlin 1.5.0 introduces a number of improvements for the feature:

- It reads nullability annotations on type arguments in compiled Java libraries that are used as dependencies.
- It supports nullability annotations with the `TYPE_USE` target for:
  - Arrays
  - Varargs
  - Fields
  - Type parameters and their bounds
  - Type arguments of base classes and interfaces
- If a nullability annotation has multiple targets applicable to a type, and one of these targets is `TYPE_USE`, then `TYPE_USE` is preferred. For example, the method signature `@Nullable String[] f()` becomes `fun f(): Array<String?>!` if `@Nullable` supports both `TYPE_USE` and `METHOD` as targets.

For these newly supported cases, using the wrong type nullability when calling Java from Kotlin produces warnings. Use the `-Xtype-enhancement-improvements-strict-mode` compiler option to enable strict mode for these cases (with error reporting).

Learn more about null-safety and platform types.

Kotlin/Native

Kotlin/Native is now more performant and stable. The notable changes are:

- 性能提升
- 停用内存泄漏检测器

性能提升

In 1.5.0, Kotlin/Native is receiving a set of performance improvements that speed up both compilation and execution.

Compiler caches are now supported in debug mode for `linuxX64` (only on Linux hosts) and `iosArm64` targets. With compiler caches enabled, most debug compilations complete much faster, except for the first one. Measurements showed about a 200% speed increase on our test projects.

To use compiler caches for new targets, opt in by adding the following lines to the project's `gradle.properties`:

- For `linuxX64`: `kotlin.native.cacheKind.linuxX64=static`
- For `iosArm64`: `kotlin.native.cacheKind.iosArm64=static`

If you encounter any issues after enabling the compiler caches, please report them to our issue tracker YouTrack.

Other improvements speed up the execution of Kotlin/Native code:

- Trivial property accessors are inlined.
- `trimIndent()` on string literals is evaluated during the compilation.

停用内存泄漏检测器

The built-in Kotlin/Native memory leak checker has been disabled by default.

It was initially designed for internal use, and it is able to find leaks only in a limited number of cases, not all of them. Moreover, it later turned out to have issues that can cause application crashes. So we've decided to turn off the memory leak checker.

The memory leak checker can still be useful for certain cases, for example, unit testing. For these cases, you can enable it by adding the following line of code:

```
Platform.isMemoryLeakCheckerActive = true
```

Note that enabling the checker for the application runtime is not recommended.

Kotlin/JS

Kotlin/JS is receiving evolutionary changes in 1.5.0. We're continuing our work on moving the JS IR compiler backend towards stable and shipping other updates:

- 将 webpack 升级到版本 5
- 用于 IR 编译器的框架与库

升级到 webpack 5

The Kotlin/JS Gradle plugin now uses webpack 5 for browser targets instead of webpack 4. This is a major webpack upgrade that brings incompatible changes. If you're using a custom webpack configuration, be sure to check the webpack 5 release notes.

Learn more about bundling Kotlin/JS projects with webpack.

用于 IR 编译器的框架与库

> The Kotlin/JS IR compiler is in Alpha. It may change incompatibly and require manual migration in the future. We would appreciate your feedback on it in YouTrack.
>
> 

Along with working on the IR-based backend for Kotlin/JS compiler, we encourage and help library authors to build their projects in `both` mode. This means they are able to produce artifacts for both Kotlin/JS compilers, therefore growing the ecosystem for the new compiler.

Many well-known frameworks and libraries are already available for the IR backend: KVision, fritz2, doodle, and others. If you're using them in your project, you can already build it with the IR backend and see the benefits it brings.

If you're writing your own library, compile it in the 'both' mode so that your clients can also use it with the new compiler.

Kotlin 多平台

In Kotlin 1.5.0, choosing a testing dependency for each platform has been simplified and it is now done automatically by the Gradle plugin.

A new API for getting a char category is now available in multiplatform projects.

标准库

The standard library has received a range of changes and improvements, from stabilizing experimental parts to adding new features:

- 稳定版无符号整数类型
- 稳定版用于大小写文本的区域设置无关 API
- 稳定版字符到整数转换 API
- 稳定版 Path API
- 趋负无穷截尾的除余运算
- Duration API 变更
- 用于获取字符类别的新版 API 现已对多平台代码可用
- 新的集合函数 firstNotNullOf()
- String?.toBoolean() 的严格版本

You can learn more about the standard library changes in this blog post.

YouTube 视频：New Standard Library Features

稳定版无符号整数类型

The `UInt`, `ULong`, `UByte`, `UShort` unsigned integer types are now Stable. The same goes for operations on these types, ranges, and progressions of them. Unsigned arrays and operations on them remain in Beta.

Learn more about unsigned integer types.

稳定版用于大小写文本的区域设置无关 API

This release brings a new locale-agnostic API for uppercase/lowercase text conversion. It provides an alternative to the `toLowerCase()`, `toUpperCase()`, `capitalize()`, and `decapitalize()` API functions, which are locale-sensitive. The new API helps you avoid errors due to different locale settings.

Kotlin 1.5.0 provides the following fully Stable alternatives:

- For `String` functions:

早期版本	**1.5.0** 的另一选择

`String.toUpperCase()`	`String.uppercase()`
`String.toLowerCase()`	`String.lowercase()`
`String.capitalize()`	`String.replaceFirstChar { it.uppercase() }`
`String.decapitalize()`	`String.replaceFirstChar { it.lowercase() }`

- For `Char` functions:

早期版本	1.5.0 的另一选择
`Char.toUpperCase()`	`Char.uppercaseChar(): Char` `Char.uppercase(): String`
`Char.toLowerCase()`	`Char.lowercaseChar(): Char` `Char.lowercase(): String`
`Char.toTitleCase()`	`Char.titlecaseChar(): Char` `Char.titlecase(): String`

> For Kotlin/JVM, there are also overloaded `uppercase()`, `lowercase()`, and `titlecase()` functions with an explicit `Locale` parameter.
>
> 

The old API functions are marked as deprecated and will be removed in a future release.

See the full list of changes to the text processing functions in KEEP.

稳定版字符到整数转换 API

Starting from Kotlin 1.5.0, new char-to-code and char-to-digit conversion functions are Stable. These functions replace the current API functions, which were often confused with the similar string-to-Int conversion.

The new API removes this naming confusion, making the code behavior more transparent and unambiguous.

This release introduces `Char` conversions that are divided into the following sets of clearly named functions:

- Functions to get the integer code of `Char` and to construct `Char` from the given code:

```
fun Char(code: Int): Char
fun Char(code: UShort): Char
val Char.code: Int
```

- Functions to convert `Char` to the numeric value of the digit it represents:

```
fun Char.digitToInt(radix: Int): Int
fun Char.digitToIntOrNull(radix: Int): Int?
```

- An extension function for `Int` to convert the non-negative single digit it represents to the corresponding `Char` representation:

```
fun Int.digitToChar(radix: Int): Char
```

The old conversion APIs, including `Number.toChar()` with its implementations (all except `Int.toChar()`) and `Char` extensions for conversion to a numeric type, like `Char.toInt()`, are now deprecated.

Learn more about the char-to-integer conversion API in KEEP.

稳定版 Path API

The experimental Path API with extensions for `java.nio.file.Path` is now Stable.

```
// construct path with the div (/) operator
val baseDir = Path("/base")
val subDir = baseDir / "subdirectory"

// list files in a directory
val kotlinFiles: List<Path> = Path("/home/user").listDirectoryEntries("*.kt")
```

Learn more about the Path API.

趋负无穷截尾的除余运算

New operations for modular arithmetics have been added to the standard library:

- `floorDiv()` returns the result of floored division. It is available for integer types.
- `mod()` returns the remainder of floored division (*modulus*). It is available for all numeric types.

These operations look quite similar to the existing division of integers and rem() function (or the `%` operator), but they work differently on negative numbers:

- `a.floorDiv(b)` differs from a regular `/` in that `floorDiv` rounds the result down (towards the lesser integer), whereas `/` truncates the result to the integer closer to 0.
- `a.mod(b)` is the difference between `a` and `a.floorDiv(b) * b`. It's either zero or has the same sign as `b`, while `a % b` can have a different one.

```
fun main() {
//sampleStart
    println("Floored division -5/3: ${(-5).floorDiv(3)}")
    println( "Modulus: ${(-5).mod(3)}")

    println("Truncated division -5/3: ${-5 / 3}")
    println( "Remainder: ${-5 % 3}")
//sampleEnd
}
```

Duration API 变更

> The Duration API is Experimental. It may be dropped or changed at any time. Use it only for evaluation purposes. We would appreciate hearing your feedback on it in YouTrack.

There is an experimental Duration class for representing duration amounts in different time units. In 1.5.0, the Duration API has received the following changes:

- Internal value representation now uses `Long` instead of `Double` to provide better precision.
- There is a new API for conversion to a particular time unit in `Long`. It comes to replace the old API, which operates with `Double` values and is now deprecated. For example, `Duration.inWholeMinutes` returns the value of the duration expressed as `Long` and replaces `Duration.inMinutes`.
- There are new companion functions for constructing a `Duration` from a number. For example, `Duration.seconds(Int)` creates a `Duration` object representing an integer number of seconds. Old extension properties like `Int.seconds` are now deprecated.

```
import kotlin.time.Duration
import kotlin.time.ExperimentalTime

@ExperimentalTime
fun main() {
//sampleStart
    val duration = Duration.milliseconds(120000)
    println("There are ${duration.inWholeSeconds} seconds in ${duration.inWholeMinutes}
 minutes")
//sampleEnd
}
```

用于获取字符类别的新版 API 现已对多平台代码可用

Kotlin 1.5.0 introduces the new API for getting a character's category according to Unicode in multiplatform projects. Several functions are now available in all the platforms and in the common code.

Functions for checking whether a char is a letter or a digit:

- `Char.isDigit()`
- `Char.isLetter()`
- `Char.isLetterOrDigit()`

```
fun main() {
//sampleStart
    val chars = listOf('a', '1', '+')
    val (letterOrDigitList, notLetterOrDigitList) = chars.partition { it.isLetterOrDig
it() }
    println(letterOrDigitList) // [a, 1]
    println(notLetterOrDigitList) // [+]
//sampleEnd
}
```

Functions for checking the case of a char:

- `Char.isLowerCase()`
- `Char.isUpperCase()`
- `Char.isTitleCase()`

```
fun main() {
//sampleStart
    val chars = listOf('Dž', 'Lj', 'Nj', 'Dz', '1', 'A', 'a', '+')
    val (titleCases, notTitleCases) = chars.partition { it.isTitleCase() }
    println(titleCases) // [Dž, Lj, Nj, Dz]
    println(notTitleCases) // [1, A, a, +]
//sampleEnd
}
```

Some other functions:

- `Char.isDefined()`
- `Char.isISOControl()`

The property `Char.category` and its return type enum class `CharCategory`, which indicates a char's general category according to Unicode, are now also available in multiplatform projects.

Learn more about characters.

新的集合函数 firstNotNullOf()

The new `firstNotNullOf()` and `firstNotNullOfOrNull()` functions combine `mapNotNull()` with `first()` or `firstOrNull()`. They map the original collection with the custom selector function and return the first non-null value. If there is no such value, `firstNotNullOf()` throws an exception, and `firstNotNullOfOrNull()` returns null.

```
fun main() {
//sampleStart
    val data = listOf("Kotlin", "1.5")
    println(data.firstNotNullOf(String::toDoubleOrNull))
    println(data.firstNotNullOfOrNull(String::toIntOrNull))
//sampleEnd
}
```

String?.toBoolean() 的严格版本

Two new functions introduce case-sensitive strict versions of the existing String?.toBoolean():

- `String.toBooleanStrict()` throws an exception for all inputs except the literals `true` and `false`.
- `String.toBooleanStrictOrNull()` returns null for all inputs except the literals `true` and `false`.

```
fun main() {
//sampleStart
    println("true".toBooleanStrict())
    println("1".toBooleanStrictOrNull())
    // println("1".toBooleanStrict()) // Exception
//sampleEnd
}
```

kotlin-test 库

The kotlin-test library introduces some new features:

- 简化多平台项目中的测试依赖项用法
- Kotlin/JVM 源代码集测试框架的自动选择
- 断言函数更新

简化多平台项目中的测试依赖项用法

Now you can use the `kotlin-test` dependency to add dependencies for testing in the `commonTest` source set, and the Gradle plugin will infer the corresponding platform dependencies for each test source set:

- `kotlin-test-junit` for JVM source sets, see automatic choice of a testing framework for Kotlin/JVM source sets
- `kotlin-test-js` for Kotlin/JS source sets
- `kotlin-test-common` and `kotlin-test-annotations-common` for common source sets
- No extra artifact for Kotlin/Native source sets

Additionally, you can use the `kotlin-test` dependency in any shared or platform-specific source set.

An existing kotlin-test setup with explicit dependencies will continue to work both in Gradle and in Maven.

Learn more about setting dependencies on test libraries.

Kotlin/JVM 源代码集测试框架的自动选择

The Gradle plugin now chooses and adds a dependency on a testing framework automatically. All you need to do is add the dependency `kotlin-test` in the common source set.

Gradle uses JUnit 4 by default. Therefore, the `kotlin("test")` dependency resolves to the variant for JUnit 4, namely `kotlin-test-junit`:

> 【Kotlin】

```
kotlin {
    sourceSets {
        val commonTest by getting {
            dependencies {
                implementation(kotlin("test")) // This brings the dependency
                                               // on JUnit 4 transitively
            }
        }
    }
}
```

> 【Groovy】

```
kotlin {
```

```
    sourceSets {
        commonTest {
            dependencies {
                implementation kotlin("test") // This brings the dependency
                                              // on JUnit 4 transitively
            }
        }
    }
}
```

You can choose JUnit 5 or TestNG by calling `useJUnitPlatform()` or `useTestNG()` in the test task:

```
tasks {
    test {
        // enable TestNG support
        useTestNG()
        // or
        // enable JUnit Platform (a.k.a. JUnit 5) support
        useJUnitPlatform()
    }
}
```

You can disable automatic testing framework selection by adding the line `kotlin.test.infer.jvm.variant=false` to the project's `gradle.properties`.

Learn more about setting dependencies on test libraries.

断言函数更新

This release brings new assertion functions and improves the existing ones.

The `kotlin-test` library now has the following features:

- 检测值的类型

  You can use the new `assertIs<T>` and `assertIsNot<T>` to check the type of a value:

  ```
  @Test
  fun testFunction() {
      val s: Any = "test"
      assertIs<String>(s)  // throws AssertionError mentioning the actual type of s
  if the assertion fails
      // can now print s.length because of contract in assertIs
      println("${s.length}")
  }
  ```

Because of type erasure, this assert function only checks whether the `value` is of the `List` type in the following example and doesn't check whether it's a list of the particular `String` element type: `assertIs<List<String>>(value)`.

- 比较数组、序列以及任意可迭代容器的内容

  There is a new set of overloaded `assertContentEquals()` functions for comparing content for different collections that don't implement structural equality:

```
@Test
fun test() {
    val expectedArray = arrayOf(1, 2, 3)
    val actualArray = Array(3) { it + 1 }
    assertContentEquals(expectedArray, actualArray)
}
```

- **`Double` and `Float` 数值的 `assertEquals()` 与 `assertNotEquals()` 的新的重载**

  There are new overloads for the `assertEquals()` function that make it possible to compare two `Double` or `Float` numbers with absolute precision. The precision value is specified as the third parameter of the function:

```
 @Test
fun test() {
    val x = sin(PI)

    // precision parameter
    val tolerance = 0.000001

    assertEquals(0.0, x, tolerance)
}
```

- 用于检测集合与元素内容的新函数

  You can now check whether the collection or element contains something with the `assertContains()` function. You can use it with Kotlin collections and elements that have the `contains()` operator, such as `IntRange`, `String`, and others:

```
@Test
fun test() {
    val sampleList = listOf<String>("sample", "sample2")
    val sampleString = "sample"
    assertContains(sampleList, sampleString)  // element in collection
    assertContains(sampleString, "amp")       // substring in string
}
```

- `assertTrue()`、`assertFalse()`、`expect()` 现在是内联函数

  From now on, you can use these as inline functions, so it's possible to call suspend functions inside a lambda expression:

```
@Test
fun test() = runBlocking<Unit> {
    val deferred = async { "Kotlin is nice" }
    assertTrue("Kotlin substring should be present") {
        deferred.await() .contains("Kotlin")
    }
}
```

kotlinx 库

Along with Kotlin 1.5.0, we are releasing new versions of the kotlinx libraries:

- `kotlinx.coroutines` 1.5.0-RC
- `kotlinx.serialization` 1.2.1
- `kotlinx-datetime` 0.2.0

coroutines 1.5.0-RC

`kotlinx.coroutines` 1.5.0-RC is here with:

- New channels API
- Stable reactive integrations
- And more

Starting with Kotlin 1.5.0, experimental coroutines are disabled and the `-Xcoroutines=experimental` flag is no longer supported.

Learn more in the changelog and the `kotlinx.coroutines` 1.5.0 release blog post.

YouTube 视频：kotlinx.coroutines 1.5.0

serialization 1.2.1

`kotlinx.serialization` 1.2.1 is here with:

- Improvements to JSON serialization performance
- Support for multiple names in JSON serialization
- Experimental .proto schema generation from `@Serializable` classes
- And more

Learn more in the changelog and the `kotlinx.serialization` 1.2.1 release blog post.

YouTube 视频：kotlinx.serialization 1.2.1

dateTime 0.2.0

`kotlinx-datetime` 0.2.0 is here with:

- `@Serializable` Datetime objects
- Normalized API of `DateTimePeriod` and `DatePeriod`
- And more

Learn more in the changelog and the `kotlinx-datetime` 0.2.0 release blog post.

迁移到 Kotlin 1.5.0

IntelliJ IDEA and Android Studio will suggest updating the Kotlin plugin to 1.5.0 once it is available.

To migrate existing projects to Kotlin 1.5.0, just change the Kotlin version to `1.5.0` and re-import your Gradle or Maven project. Learn how to update to Kotlin 1.5.0.

To start a new project with Kotlin 1.5.0, update the Kotlin plugin and run the Project Wizard from **File** | **New** | **Project**.

The new command-line compiler is available for downloading on the GitHub release page.

Kotlin 1.5.0 is a feature release and therefore can bring incompatible changes to the language. Find the detailed list of such changes in the Compatibility Guide for Kotlin 1.5.

Kotlin 1.4.30 的新特性

发布于：*2021-02-03*

Kotlin 1.4.30 offers preview versions of new language features, promotes the new IR backend of the Kotlin/JVM compiler to Beta, and ships various performance and functional improvements.

You can also learn about new features in this blog post.

语言特性

Kotlin 1.5.0 is going to deliver new language features – JVM 记录类型支持, sealed interfaces, and Stable inline classes. In Kotlin 1.4.30, you can try these features and improvements in preview mode. We would be very grateful if you share your feedback with us in the corresponding YouTrack tickets, as that will allow us to address it before the release of 1.5.0.

- JVM 记录类型支持
- 密封接口以及密封类改进
- 改进的内联类

To enable these language features and improvements in preview mode, you need to opt in by adding specific compiler options. See the sections below for details.

Learn more about the new features preview in this blog post.

JVM 记录类型支持

> The JVM records feature is Experimental. It may be dropped or changed at any time. Opt-in is required (see the details below), and you should use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack.
>
> 

The JDK 16 release includes plans to stabilize a new Java class type called record. To provide all the benefits of Kotlin and maintain its interoperability with Java, Kotlin is introducing experimental record class support.

You can use record classes that are declared in Java just like classes with properties in Kotlin. No additional steps are required.

Starting with 1.4.30, you can declare the record class in Kotlin using the `@JvmRecord` annotation for a data class:

```
@JvmRecord
data class User(val name: String, val age: Int)
```

To try the preview version of JVM records, add the compiler options `-Xjvm-enable-preview` and `-language-version 1.5`.

We're continuing to work on JVM records support, and we would be very grateful if you would share your feedback with us using this YouTrack ticket.

Learn more about implementation, restrictions, and the syntax in KEEP.

密封接口

> 密封接口 are Experimental. They may be dropped or changed at any time. Opt-in is required (see the details below), and you should use them only for evaluation purposes. We would appreciate your feedback on them in YouTrack.
>
> 

In Kotlin 1.4.30, we're shipping the prototype of *sealed interfaces*. They complement sealed classes and make it possible to build more flexible restricted class hierarchies.

They can serve as "internal" interfaces that cannot be implemented outside the same module. You can rely on that fact, for example, to write exhaustive `when` expressions.

```
sealed interface Polygon

class Rectangle(): Polygon
class Triangle(): Polygon

// when() is exhaustive: no other polygon implementations can appear
// after the module is compiled
fun draw(polygon: Polygon) = when (polygon) {
    is Rectangle -> // ...
    is Triangle -> // ...
}
```

Another use-case: with sealed interfaces, you can inherit a class from two or more sealed superclasses.

```
sealed interface Fillable {
   fun fill()
}
sealed interface Polygon {
   val vertices: List<Point>
}
```

```
class Rectangle(override val vertices: List<Point>): Fillable, Polygon {
   override fun fill() { /*...*/ }
}
```

To try the preview version of sealed interfaces, add the compiler option `-language-version 1.5`. Once you switch to this version, you'll be able to use the `sealed` modifier on interfaces. We would be very grateful if you would share your feedback with us using this YouTrack ticket.

Learn more about sealed interfaces.

包范围的密封类层次结构

> Package-wide hierarchies of sealed classes are Experimental. They may be dropped or changed at any time. Opt-in is required (see the details below), and you should use them only for evaluation purposes. We would appreciate your feedback on them in YouTrack.
>
> 

Sealed classes can now form more flexible hierarchies. They can have subclasses in all files of the same compilation unit and the same package. Previously, all subclasses had to appear in the same file.

Direct subclasses may be top-level or nested inside any number of other named classes, named interfaces, or named objects. The subclasses of a sealed class must have a name that is properly qualified – they cannot be local nor anonymous objects.

To try package-wide hierarchies of sealed classes, add the compiler option `-language-version 1.5`. We would be very grateful if you would share your feedback with us using this YouTrack ticket.

Learn more about package-wide hierarchies of sealed classes.

改进的内联类

> Inline value classes are in Beta. They are almost stable, but migration steps may be required in the future. We'll do our best to minimize any changes you have to make. We would appreciate your feedback on the inline classes feature in YouTrack.
>
> 

Kotlin 1.4.30 promotes inline classes to Beta and brings the following features and improvements to them:

- Since inline classes are value-based, you can define them using the `value` modifier. The `inline` and `value` modifiers are now equivalent to each other. In future Kotlin versions, we're planning to deprecate the `inline` modifier.

From now on, Kotlin requires the `@JvmInline` annotation before a class declaration for the JVM backend:

```
inline class Name(private val s: String)

value class Name(private val s: String)

// For JVM backends
@JvmInline
value class Name(private val s: String)
```

- Inline classes can have `init` blocks. You can add code to be executed right after the class is instantiated:

```
@JvmInline
value class Negative(val x: Int) {
  init {
      require(x < 0) { }
  }
}
```

- Calling functions with inline classes from Java code: before Kotlin 1.4.30, you couldn't call functions that accept inline classes from Java because of mangling. From now on, you can disable mangling manually. To call such functions from Java code, you should add the `@JvmName` annotation before the function declaration:

```
inline class UInt(val x: Int)

fun compute(x: Int) { }

@JvmName("computeUInt")
fun compute(x: UInt) { }
```

- In this release, we've changed the mangling scheme for functions to fix the incorrect behavior. These changes led to ABI changes.

  Starting with 1.4.30, the Kotlin compiler uses a new mangling scheme by default. Use the `-Xuse-14-inline-classes-mangling-scheme` compiler flag to force the compiler to use the old 1.4.0 mangling scheme and preserve binary compatibility.

Kotlin 1.4.30 promotes inline classes to Beta and we are planning to make them Stable in future releases. We'd be very grateful if you would share your feedback with us using this YouTrack ticket.

To try the preview version of inline classes, add the compiler option `-Xinline-classes` or `-language-version 1.5`.

Learn more about the mangling algorithm in KEEP.

Learn more about inline classes.

Kotlin/JVM

JVM IR 编译器后端达到 Beta 版

The IR-based compiler backend for Kotlin/JVM, which was presented in 1.4.0 in Alpha, has reached Beta. This is the last pre-stable level before the IR backend becomes the default for the Kotlin/JVM compiler.

We're now dropping the restriction on consuming binaries produced by the IR compiler. Previously, you could use code compiled by the new JVM IR backend only if you had enabled the new backend. Starting from 1.4.30, there is no such limitation, so you can use the new backend to build components for third-party use, such as libraries. Try the Beta version of the new backend and share your feedback in our issue tracker.

To enable the new JVM IR backend, add the following lines to the project's configuration file:

- In Gradle:

> 【Kotlin】

```
tasks.withType(org.jetbrains.kotlin.gradle.dsl.KotlinJvmCompile::class) {
  kotlinOptions.useIR = true
}
```

> 【Groovy】

```
tasks.withType(org.jetbrains.kotlin.gradle.dsl.KotlinJvmCompile) {
  kotlinOptions.useIR = true
}
```

- In Maven:

```
<configuration>
    <args>
        <arg>-Xuse-ir</arg>
```

```
    </args>
</configuration>
```

Learn more about the changes that the JVM IR backend brings in this blog post.

Kotlin/Native

性能提升

Kotlin/Native has received a variety of performance improvements in 1.4.30, which has resulted in faster compilation times. For example, the time required to rebuild the framework in the Networking and data storage with Kotlin Multiplatform Mobile sample has decreased from 9.5 seconds (in 1.4.10) to 4.5 seconds (in 1.4.30).

Apple watchOS 64-位模拟器目标

The x86 simulator target has been deprecated for watchOS since version 7.0. To keep up with the latest watchOS versions, Kotlin/Native has the new target `watchosX64` for running the simulator on 64-bit architecture.

对 Xcode 12.2 库的支持

We have added support for the new libraries delivered with Xcode 12.2. You can now use them from Kotlin code.

Kotlin/JS

顶层属性的延迟初始化

> 顶层属性的延迟初始化 is Experimental. It may be dropped or changed at any time. Opt-in is required (see the details below), and you should use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack.
>
> 

The IR backend for Kotlin/JS is receiving a prototype implementation of lazy initialization for top-level properties. This reduces the need to initialize all top-level properties when the application starts, and it should significantly improve application start-up times.

We'll keep working on the lazy initialization, and we ask you to try the current prototype and share your thoughts and results in this YouTrack ticket or the `#javascript` channel in the official Kotlin Slack (get an invite here).

To use the lazy initialization, add the `-Xir-property-lazy-initialization` compiler option when compiling the code with the JS IR compiler.

Gradle 项目改进

支持 Gradle 配置缓存

Starting with 1.4.30, the Kotlin Gradle plugin supports the configuration cache feature. It speeds up the build process: once you run the command, Gradle executes the configuration phase and calculates the task graph. Gradle caches the result and reuses it for subsequent builds.

To start using this feature, you can use the Gradle command or set up the IntelliJ based IDE.

标准库

用于大小写文本的区域设置无关 API

> The locale-agnostic API feature is Experimental. It may be dropped or changed at any time. Use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack.
>
> 

This release introduces the experimental locale-agnostic API for changing the case of strings and characters. The current `toLowerCase()`, `toUpperCase()`, `capitalize()`, `decapitalize()` API functions are locale-sensitive. This means that different platform locale settings can affect code behavior. For example, in the Turkish locale, when the string "kotlin" is converted using `toUpperCase`, the result is "KOTLİN", not "KOTLIN".

```
// current API
println("Needs to be capitalized".toUpperCase()) // NEEDS TO BE CAPITALIZED

// new API
println("Needs to be capitalized".uppercase()) // NEEDS TO BE CAPITALIZED
```

Kotlin 1.4.30 provides the following alternatives:

- For `String` functions:

Earlier versions	1.4.30 alternative

`String.toUpperCase()`	`String.uppercase()`
`String.toLowerCase()`	`String.lowercase()`
`String.capitalize()`	`String.replaceFirstChar { it.uppercase() }`
`String.decapitalize()`	`String.replaceFirstChar { it.lowercase() }`

- For `Char` functions:

Earlier versions	1.4.30 alternative
`Char.toUpperCase()`	`Char.uppercaseChar(): Char` `Char.uppercase(): String`
`Char.toLowerCase()`	`Char.lowercaseChar(): Char` `Char.lowercase(): String`
`Char.toTitleCase()`	`Char.titlecaseChar(): Char` `Char.titlecase(): String`

> For Kotlin/JVM, there are also overloaded `uppercase()`, `lowercase()`, and `titlecase()` functions with an explicit `Locale` parameter.



See the full list of changes to the text processing functions in KEEP.

明确字符到码值与字符到数位的转换

> The unambiguous API for the `Char` conversion feature is Experimental. It may be dropped or changed at any time. Use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack.



The current `Char` to numbers conversion functions, which return UTF-16 codes expressed in different numeric types, are often confused with the similar String-to-Int conversion, which returns the numeric value of a string:

```
"4".toInt() // returns 4
'4'.toInt() // returns 52
// and there was no common function that would return the numeric value 4 for Char '4'
```

To avoid this confusion we've decided to separate `Char` conversions into two following sets of clearly named functions:

- Functions to get the integer code of `Char` and to construct `Char` from the given code:

```
fun Char(code: Int): Char
fun Char(code: UShort): Char
val Char.code: Int
```

- Functions to convert `Char` to the numeric value of the digit it represents:

```
fun Char.digitToInt(radix: Int): Int
fun Char.digitToIntOrNull(radix: Int): Int?
```

- An extension function for `Int` to convert the non-negative single digit it represents to the corresponding `Char` representation:

```
fun Int.digitToChar(radix: Int): Char
```

See more details in KEEP.

序列化更新

Along with Kotlin 1.4.30, we are releasing `kotlinx.serialization` 1.1.0-RC, which includes some new features:

- 内联类序列化支持
- 无符号原生类型的序列化支持

内联类序列化支持

Starting with Kotlin 1.4.30, you can make inline classes serializable:

```
@Serializable
inline class Color(val rgb: Int)
```

> The feature requires the new 1.4.30 IR compiler.
>
> 

The serialization framework does not box serializable inline classes when they are used in other serializable classes.

Learn more in the `kotlinx.serialization` docs.

无符号原生类型的序列化支持

Starting from 1.4.30, you can use standard JSON serializers of kotlinx.serialization for unsigned primitive types: `UInt`, `ULong`, `UByte`, and `UShort`:

```
@Serializable
class Counter(val counted: UByte, val description: String)
fun main() {
   val counted = 239.toUByte()
   println(Json.encodeToString(Counter(counted, "tries")))
}
```

Learn more in the `kotlinx.serialization` docs.

Kotlin 1.4.20 的新特性

发布于：*2020-11-23*

Kotlin 1.4.20 offers a number of new experimental features and provides fixes and improvements for existing features, including those added in 1.4.0.

You can also learn about new features with more examples in this blog post.

Kotlin/JVM

Improvements of Kotlin/JVM are intended to keep it up with the features of modern Java versions:

- Java 15 目标
- invokedynamic 字符串连接

Java 15 目标

Now Java 15 is available as a Kotlin/JVM target.

invokedynamic 字符串连接

> `invokedynamic` string concatenation is Experimental. It may be dropped or changed at any time. Opt-in is required (see details below). Use it only for evaluation purposes. We appreciate your feedback on it in YouTrack.
>
> 

Kotlin 1.4.20 can compile string concatenations into dynamic invocations on JVM 9+ targets, therefore improving the performance.

Currently, this feature is experimental and covers the following cases:

- `String.plus` in the operator (`a + b`), explicit (`a.plus(b)`), and reference (`(a::plus)(b)`) form.
- `toString` on inline and data classes.
- string templates except for ones with a single non-constant argument (see KT-42457).

To enable `invokedynamic` string concatenation, add the `-Xstring-concat` compiler option with one of the following values:

- `indy-with-constants` to perform `invokedynamic` concatenation on strings with

StringConcatFactory.makeConcatWithConstants().

- `indy` to perform `invokedynamic` concatenation on strings with StringConcatFactory.makeConcat().
- `inline` to switch back to the classic concatenation via `StringBuilder.append()`.

Kotlin/JS

Kotlin/JS keeps evolving fast, and in 1.4.20 you can find a number experimental features and improvements:

- Gradle DSL 变更
- 新的向导模板
- 使用 IR 编译器时忽略编译错误

Gradle DSL 变更

The Gradle DSL for Kotlin/JS receives a number of updates which simplify project setup and customization. This includes webpack configuration adjustments, modifications to the auto-generated `package.json` file, and improved control over transitive dependencies.

用于 webpack 配置的单点

A new configuration block `commonWebpackConfig` is available for the browser target. Inside it, you can adjust common settings from a single point, instead of having to duplicate configurations for the `webpackTask`, `runTask`, and `testTask`.

To enable CSS support by default for all three tasks, add the following snippet in the `build.gradle(.kts)` of your project:

```
browser {
    commonWebpackConfig {
        cssSupport.enabled = true
    }
    binaries.executable()
}
```

Learn more about configuring webpack bundling.

在 Gradle 中定制 package.json

For more control over your Kotlin/JS package management and distribution, you can now add properties to the project file `package.json` via the Gradle DSL.

To add custom fields to your `package.json`, use the `customField` function in the compilation's `packageJson` block:

```
kotlin {
    js(BOTH) {
        compilations["main"].packageJson {
            customField("hello", mapOf("one" to 1, "two" to 2))
        }
    }
}
```

Learn more about `package.json` 定制.

选择性 yarn 依赖解析

> Support for selective yarn dependency resolutions is Experimental. It may be dropped or changed at any time. Use it only for evaluation purposes. We appreciate your feedback on it in YouTrack.
>
> 

Kotlin 1.4.20 provides a way of configuring Yarn's selective dependency resolutions - the mechanism for overriding dependencies of the packages you depend on.

You can use it through the `YarnRootExtension` inside the `YarnPlugin` in Gradle. To affect the resolved version of a package for your project, use the `resolution` function passing in the package name selector (as specified by Yarn) and the version to which it should resolve.

```
rootProject.plugins.withType<YarnPlugin> {
    rootProject.the<YarnRootExtension>().apply {
        resolution("react", "16.0.0")
        resolution("processor/decamelize", "3.0.0")
    }
}
```

Here, *all* of your npm dependencies which require `react` will receive version `16.0.0`, and `processor` will receive its dependency `decamelize` as version `3.0.0`.

禁用 granular workspaces

> Disabling granular workspaces is Experimental. It may be dropped or changed at any time. Use it only for evaluation purposes. We appreciate your feedback on it in YouTrack.
>
> 

To speed up build times, the Kotlin/JS Gradle plugin only installs the dependencies which are required for a particular Gradle task. For example, the `webpack-dev-server` package is only installed when you execute one of the `*Run` tasks, and not when you execute the assemble task. Such behavior can potentially bring problems when you run multiple Gradle processes in parallel. When the dependency requirements clash, the two installations of npm packages can cause errors.

To resolve this issue, Kotlin 1.4.20 includes an option to disable these so-called *granular workspaces*. This feature is currently available through the `YarnRootExtension` inside the `YarnPlugin` in Gradle. To use it, add the following snippet to your `build.gradle.kts` file:

```
rootProject.plugins.withType<YarnPlugin> {
    rootProject.the<YarnRootExtension>().disableGranularWorkspaces()
}
```

新的向导模板

To give you more convenient ways to customize your project during creation, the project wizard for Kotlin comes with new templates for Kotlin/JS applications:

- **Browser Application** - a minimal Kotlin/JS Gradle project that runs in the browser.
- **React Application** - a React app that uses the appropriate `kotlin-wrappers`. It provides options to enable integrations for style-sheets, navigational components, or state containers.
- **Node.js Application** - a minimal project for running in a Node.js runtime. It comes with the option to directly include the experimental `kotlinx-nodejs` package.

使用 IR 编译器时忽略编译错误

> *Ignore compilation errors* mode is Experimental. It may be dropped or changed at any time. Opt-in is required (see details below). Use it only for evaluation purposes. We appreciate your feedback on it in YouTrack.
>
> 

The IR compiler for Kotlin/JS comes with a new experimental mode - *compilation with errors*. In this mode, you can run you code even if it contains errors, for example, if you want to try certain things it when the whole application is not ready yet.

There are two tolerance policies for this mode:

- `SEMANTIC`: the compiler will accept code which is syntactically correct, but doesn't make sense semantically, such as `val x: String = 3`.
- `SYNTAX`: the compiler will accept any code, even if it contains syntax errors.

To allow compilation with errors, add the `-Xerror-tolerance-policy=` compiler option with one of the values listed above.

Learn more about Kotlin/JS IR compiler.

Kotlin/Native

Kotlin/Native's priorities in 1.4.20 are performance and polishing existing features. These are the notable improvements:

- 逃逸分析
- 性能提升与错误修复
- 选择加入 Objective-C 异常的包装
- CocoaPods 插件改进
- 对 Xcode 12 库的支持

逃逸分析

> The escape analysis mechanism is Experimental. It may be dropped or changed at any time. Use it only for evaluation purposes. We appreciate your feedback on it in YouTrack.
>
> 

Kotlin/Native receives a prototype of the new escape analysis mechanism. It improves the runtime performance by allocating certain objects on the stack instead of the heap. This mechanism shows a 10% average performance increase on our benchmarks, and we continue improving it so that it speeds up the program even more.

The escape analysis runs in a separate compilation phase for the release builds (with the `-opt` compiler option).

If you want to disable the escape analysis phase, use the `-Xdisable-phases=EscapeAnalysis` compiler option.

性能提升与错误修复

Kotlin/Native receives performance improvements and bug fixes in various components, including the ones added in 1.4.0, for example, the code sharing mechanism.

选择加入 Objective-C 异常的包装

> The Objective-C exception wrapping mechanism is Experimental. It may be dropped or changed at any time. Opt-in is required (see details below). Use it only for evaluation purposes. We appreciate your feedback on it in YouTrack.
>
> 

Kotlin/Native now can handle exceptions thrown from Objective-C code in runtime to avoid program crashes.

You can opt in to wrap `NSException`'s into Kotlin exceptions of type `ForeignException`. They hold the references to the original `NSException`'s. This lets you get the information about the root cause and handle it properly.

To enable wrapping of Objective-C exceptions, specify the `-Xforeign-exception-mode objc-wrap` option in the `cinterop` call or add `foreignExceptionMode = objc-wrap` property to `.def` file. If you use CocoaPods integration, specify the option in the `pod {}` build script block of a dependency like this:

```
pod("foo") {
    extraOpts = listOf("-Xforeign-exception-mode", "objc-wrap")
}
```

The default behavior remains unchanged: the program terminates when an exception is thrown from the Objective-C code.

CocoaPods 插件改进

Kotlin 1.4.20 continues the set of improvements in CocoaPods integration. Namely, you can try the following new features:

- 改进了任务执行
- 扩展了 DSL
- 更新了与 Xcode 的集成

改进了任务执行

CocoaPods plugin gets an improved task execution flow. For example, if you add a new CocoaPods dependency, existing dependencies are not rebuilt. Adding an extra target also doesn't affect rebuilding dependencies for existing ones.

扩展了 DSL

The DSL of adding CocoaPods dependencies to your Kotlin project receives new capabilites.

In addition to local Pods and Pods from the CocoaPods repository, you can add dependencies on the following types of libraries:

- A library from a custom spec repository.
- A remote library from a Git repository.
- A library from an archive (also available by arbitrary HTTP address).
- A static library.
- A library with custom cinterop options.

Learn more about adding CocoaPods dependencies in Kotlin projects. Find examples in the Kotlin with CocoaPods sample.

更新了与 Xcode 的集成

To work correctly with Xcode, Kotlin requires some Podfile changes:

- If your Kotlin Pod has any Git, HTTP, or specRepo Pod dependency, you should also specify it in the Podfile.
- When you add a library from the custom spec, you also should specify the location of specs at the beginning of your Podfile.

Now integration errors have a detailed description in IDEA. So if you have problems with your Podfile, you will immediately know how to fix them.

Learn more about creating Kotlin pods.

对 Xcode 12 库的支持

We have added support for new libraries delivered with Xcode 12. Now you can use them from the Kotlin code.

Kotlin 多平台

更新了多平台库发布的结构

Starting from Kotlin 1.4.20, there is no longer a separate metadata publication. Metadata artifacts are now included in the *root* publication which stands for the whole library and is automatically resolved to the appropriate platform-specific artifacts when added as a dependency to the common source set.

Learn more about publishing a multiplatform library.

与早期版本的兼容性

This change of structure breaks the compatibility between projects with hierarchical project structure. If a multiplatform project and a library it depends on both have the hierarchical project structure, then you need to update them to Kotlin 1.4.20 or higher simultaneously. Libraries published with Kotlin 1.4.20 are not available for using from project published with earlier versions.

Projects and libraries without the hierarchical project structure remain compatible.

标准库

The standard library of Kotlin 1.4.20 offers new extensions for working with files and a better performance.

- java.nio.file.Path 的扩展
- 改进了 String.replace 函数的性能

java.nio.file.Path 的扩展

> Extensions for `java.nio.file.Path` are Experimental. They may be dropped or changed at any time. Opt-in is required (see details below). Use them only for evaluation purposes. We appreciate your feedback on them in YouTrack.
>
> 

Now the standard library provides experimental extensions for `java.nio.file.Path`. Working with the modern JVM file API in an idiomatic Kotlin way is now similar to working with `java.io.File` extensions from the `kotlin.io` package.

```
// construct path with the div (/) operator
val baseDir = Path("/base")
val subDir = baseDir / "subdirectory"

// list files in a directory
val kotlinFiles: List<Path> = Path("/home/user").listDirectoryEntries("*.kt")
```

The extensions are available in the `kotlin.io.path` package in the `kotlin-stdlib-jdk7` module. To use the extensions, opt-in to the experimental annotation `@ExperimentalPathApi`.

改进了 String.replace 函数的性能

The new implementation of `String.replace()` speeds up the function execution. The case-sensitive variant uses a manual replacement loop based on `indexOf`, while the case-insensitive one uses regular expression matching.

Kotlin Android 扩展

In 1.4.20 the Kotlin Android Extensions plugin becomes deprecated and `Parcelable` implementation generator moves to a separate plugin.

- 弃用合成视图
- Parcelable 实现生成器的新插件

弃用合成视图

Synthetic views were presented in the Kotlin Android Extensions plugin a while ago to simplify the interaction with UI elements and reduce boilerplate. Now Google offers a native mechanism that does the same - Android Jetpack's view bindings, and we're deprecating synthetic views in favor of those.

We extract the Parcelable implementations generator from `kotlin-android-extensions` and start the deprecation cycle for the rest of it - synthetic views. For now, they will keep working with a deprecation warning. In the future, you'll need to switch your project to another solution. Here are the guidelines that will help you migrate your Android project from synthetics to view bindings.

Parcelable 实现生成器的新插件

The `Parcelable` implementation generator is now available in the new `kotlin-parcelize` plugin. Apply this plugin instead of `kotlin-android-extensions`.

> `kotlin-parcelize` and `kotlin-android-extensions` can't be applied together in one module.
>
> 

The `@Parcelize` annotation is moved to the `kotlinx.parcelize` package.

Learn more about `Parcelable` implementation generator in the Android documentation.

Kotlin 1.4.0 的新特性

发布于：*2020-08-17*

在 Kotlin 1.4.0 中，我们对其所有组件进行了大量改进，其中重点是质量与性能。 以下是 Kotlin 1.4.0 中最重要的变更列表。

语言特性与改进

Kotlin 1.4.0 中有各种不同的语言特性与改进。包括：

- Kotlin 接口的 SAM 转换
- 面向库作者的显式 API 模式
- 混用具名与位置参数
- 拖尾的逗号
- 可调用引用改进
- 循环中的 when 内部可以 break 及 continue

Kotlin 接口的 SAM 转换

Before Kotlin 1.4.0, you could apply SAM (Single Abstract Method) conversions only when working with Java methods and Java interfaces from Kotlin. From now on, you can use SAM conversions for Kotlin interfaces as well. To do so, mark a Kotlin interface explicitly as functional with the `fun` modifier.

SAM conversion applies if you pass a lambda as an argument when an interface with only one single abstract method is expected as a parameter. In this case, the compiler automatically converts the lambda to an instance of the class that implements the abstract member function.

```
fun interface IntPredicate {
    fun accept(i: Int): Boolean
}

val isEven = IntPredicate { it % 2 == 0 }

fun main() {
    println("Is 7 even? - ${isEven.accept(7)}")
}
```

Learn more about Kotlin functional interfaces and SAM conversions.

面向库作者的显式 API 模式

Kotlin compiler offers *explicit API mode* for library authors. In this mode, the compiler performs additional checks that help make the library's API clearer and more consistent. It adds the following requirements for declarations exposed to the library's public API:

- Visibility modifiers are required for declarations if the default visibility exposes them to the public API. This helps ensure that no declarations are exposed to the public API unintentionally.
- Explicit type specifications are required for properties and functions that are exposed to the public API. This guarantees that API users are aware of the types of API members they use.

Depending on your configuration, these explicit APIs can produce errors (*strict* mode) or warnings (*warning* mode). Certain kinds of declarations are excluded from such checks for the sake of readability and common sense:

- primary constructors
- properties of data classes
- property getters and setters
- `override` methods

Explicit API mode analyzes only the production sources of a module.

To compile your module in the explicit API mode, add the following lines to your Gradle build script:

> **【Kotlin】**

```
kotlin {
    // for strict mode
    explicitApi()
    // or
    explicitApi = ExplicitApiMode.Strict

    // for warning mode
    explicitApiWarning()
    // or
    explicitApi = ExplicitApiMode.Warning
}
```

> **【Groovy】**

```
kotlin {
    // for strict mode
    explicitApi()
```

```
    // or
    explicitApi = 'strict'

    // for warning mode
    explicitApiWarning()
    // or
    explicitApi = 'warning'
}
```

When using the command-line compiler, switch to explicit API mode by adding the `-Xexplicit-api` compiler option with the value `strict` or `warning`.

```
-Xexplicit-api={strict|warning}
```

Find more details about the explicit API mode in the KEEP.

混用具名与位置参数

In Kotlin 1.3, when you called a function with named arguments, you had to place all the arguments without names (positional arguments) before the first named argument. For example, you could call `f(1, y = 2)`, but you couldn't call `f(x = 1, 2)`.

It was really annoying when all the arguments were in their correct positions but you wanted to specify a name for one argument in the middle. It was especially helpful for making absolutely clear which attribute a boolean or `null` value belongs to.

In Kotlin 1.4, there is no such limitation – you can now specify a name for an argument in the middle of a set of positional arguments. Moreover, you can mix positional and named arguments any way you like, as long as they remain in the correct order.

```
fun reformat(
    str: String,
    uppercaseFirstLetter: Boolean = true,
    wordSeparator: Char = ' '
) {
    // ...
}

//Function call with a named argument in the middle
reformat("This is a String!", uppercaseFirstLetter = false , '-')
```

拖尾的逗号

With Kotlin 1.4 you can now add a trailing comma in enumerations such as argument and parameter lists, `when` entries, and components of destructuring declarations. With a trailing comma, you can add new items and change their order without adding or removing commas.

This is especially helpful if you use multi-line syntax for parameters or values. After adding a trailing comma, you can then easily swap lines with parameters or values.

```
fun reformat(
    str: String,
    uppercaseFirstLetter: Boolean = true,
    wordSeparator: Character = ' ', //尾部逗号
) {
    // ...
}
```

```
val colors = listOf(
    "red",
    "green",
    "blue", //尾部逗号
)
```

可调用引用改进

Kotlin 1.4 supports more cases for using callable references:

- 对具有默认实参值的函数的引用
- 可作为返回 `Unit` 的函数的函数引用
- 根据函数的参数数量进行调整的引用
- 可调用引用的挂起转换

对具有默认实参值的函数的引用

Now you can use callable references to functions with default argument values. If the callable reference to the function `foo` takes no arguments, the default value `0` is used.

```
fun foo(i: Int = 0): String = "$i!"

fun apply(func: () -> String): String = func()

fun main() {
    println(apply(::foo))
}
```

Previously, you had to write additional overloads for the function `apply` to use the default argument values.

```
// some new overload
fun applyInt(func: (Int) -> String): String = func(0)
```

返回 Unit 的函数的函数引用

In Kotlin 1.4, you can use callable references to functions returning any type in `Unit`-returning functions. Before Kotlin 1.4, you could only use lambda arguments in this case. Now you can use both lambda arguments and callable references.

```
fun foo(f: () -> Unit) { }
fun returnsInt(): Int = 42

fun main() {
    foo { returnsInt() } // this was the only way to do it  before 1.4
    foo(::returnsInt) // starting from 1.4, this also works
}
```

根据函数的参数数量进行调整的引用

Now you can adapt callable references to functions when passing a variable number of arguments (`vararg`) . You can pass any number of parameters of the same type at the end of the list of passed arguments.

```
fun foo(x: Int, vararg y: String) {}

fun use0(f: (Int) -> Unit) {}
fun use1(f: (Int, String) -> Unit) {}
fun use2(f: (Int, String, String) -> Unit) {}

fun test() {
    use0(::foo)
    use1(::foo)
    use2(::foo)
}
```

可调用引用的挂起转换

In addition to suspend conversion on lambdas, Kotlin now supports suspend conversion on callable references starting from version 1.4.0.

```
fun call() {}
```

```
fun takeSuspend(f: suspend () -> Unit) {}

fun test() {
    takeSuspend { call() } // OK before 1.4
    takeSuspend(::call) // In Kotlin 1.4, it also works
}
```

在循环中的 when 内部使用 break 与 continue

In Kotlin 1.3, you could not use unqualified `break` and `continue` inside `when` expressions included in loops. The reason was that these keywords were reserved for possible fall-through behavior in `when` expressions.

That's why if you wanted to use `break` and `continue` inside `when` expressions in loops, you had to label them, which became rather cumbersome.

```
fun test(xs: List<Int>) {
    LOOP@for (x in xs) {
        when (x) {
            2 -> continue@LOOP
            17 -> break@LOOP
            else -> println(x)
        }
    }
}
```

In Kotlin 1.4, you can use `break` and `continue` without labels inside `when` expressions included in loops. They behave as expected by terminating the nearest enclosing loop or proceeding to its next step.

```
fun test(xs: List<Int>) {
    for (x in xs) {
        when (x) {
            2 -> continue
            17 -> break
            else -> println(x)
        }
    }
}
```

The fall-through behavior inside `when` is subject to further design.

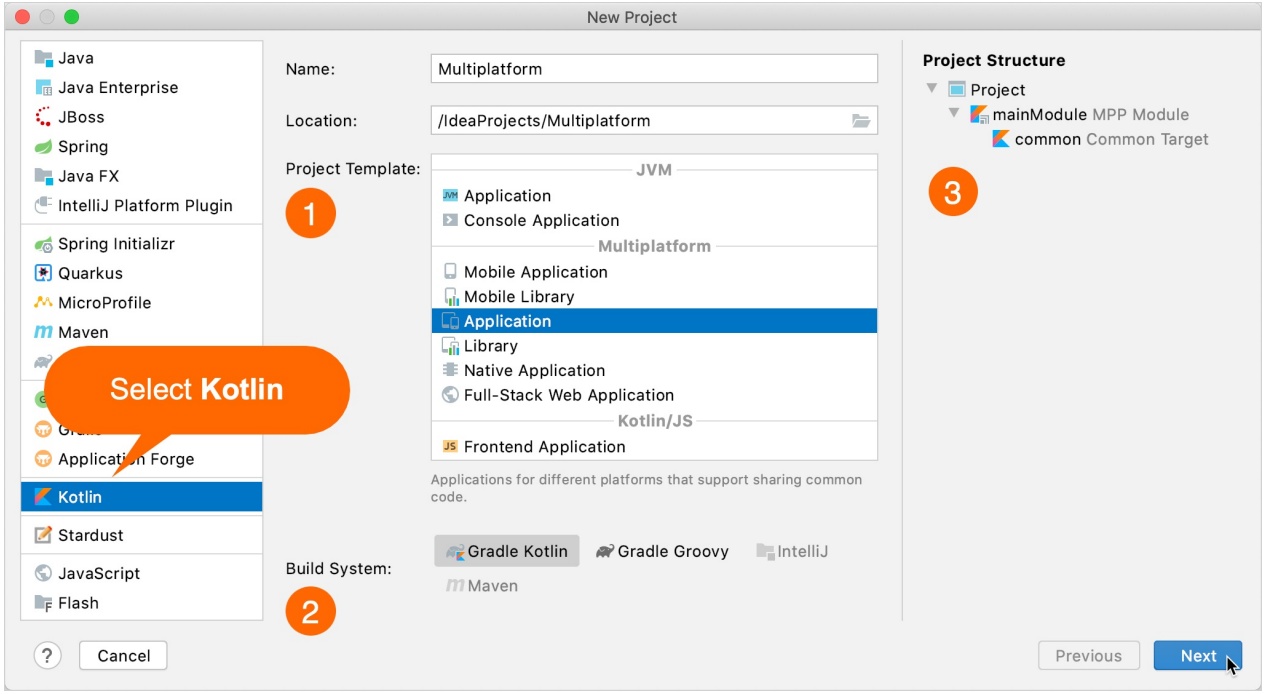
IDE 中的新工具

With Kotlin 1.4, you can use the new tools in IntelliJ IDEA to simplify Kotlin development:

- 新的灵活项目向导
- 协程调试器

新的灵活项目向导

With the flexible new Kotlin Project Wizard, you have a place to easily create and configure different types of Kotlin projects, including multiplatform projects, which can be difficult to configure without a UI.



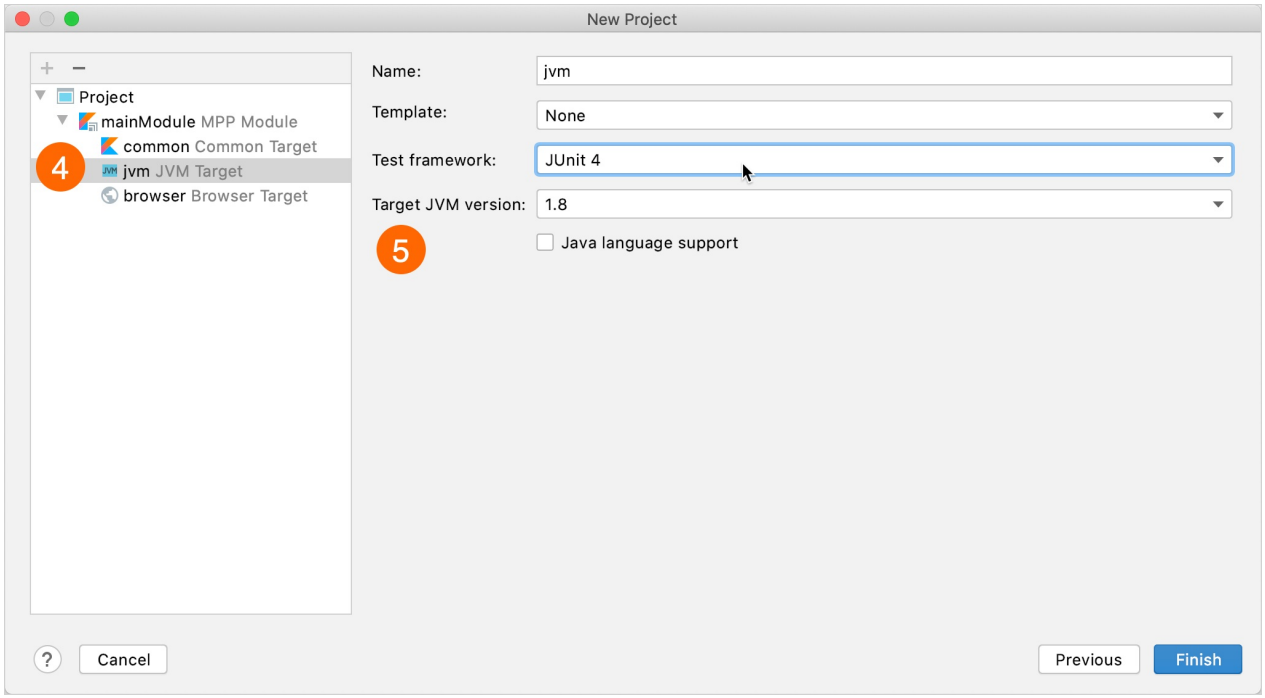
The new Kotlin Project Wizard is both simple and flexible:

1. *Select the project template*, depending on what you're trying to do. More templates will be added in the future.
2. *Select the build system* – Gradle (Kotlin or Groovy DSL), Maven, or IntelliJ IDEA.

   The Kotlin Project Wizard will only show the build systems supported on the selected project template.
3. *Preview the project structure* directly on the main screen.

Then you can finish creating your project or, optionally, *configure the project* on the next screen:

1. *Add/remove modules and targets* supported for this project template.
2. *Configure module and target settings*, for example, the target JVM version, target template, and test framework.



In the future, we are going to make the Kotlin Project Wizard even more flexible by adding more configuration options and templates.

You can try out the new Kotlin Project Wizard by working through these tutorials:

- Create a console application based on Kotlin/JVM
- Create a Kotlin/JS application for React
- Create a Kotlin/Native application

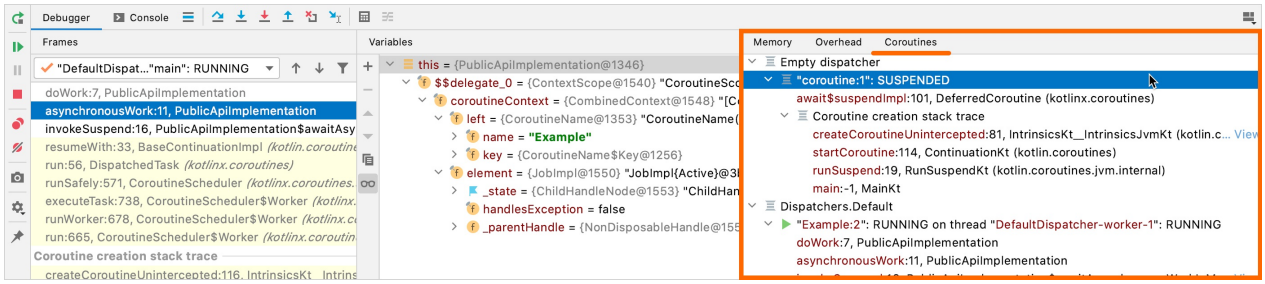
协程调试器

Many people already use coroutines for asynchronous programming. But when it came to debugging, working with coroutines before Kotlin 1.4, could be a real pain. Since coroutines jumped between threads, it was difficult to understand what a specific coroutine was doing and check its context. In some cases, tracking steps over breakpoints simply didn't work. As a result, you had to rely on logging or mental effort to debug code that used coroutines.

In Kotlin 1.4, debugging coroutines is now much more convenient with the new functionality shipped with the Kotlin plugin.

> Debugging works for versions 1.3.8 or later of `kotlinx-coroutines-core`.
>
> 

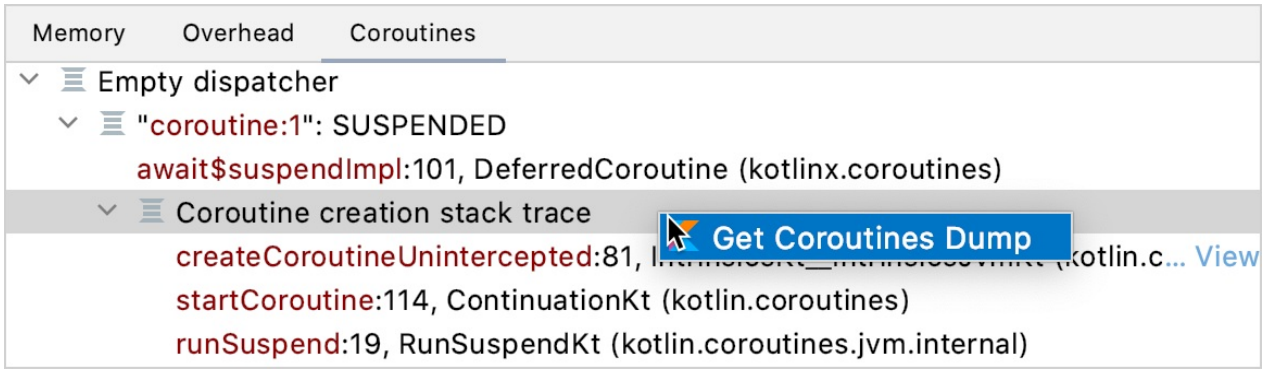
The **Debug Tool Window** now contains a new **Coroutines** tab. In this tab, you can find information about both currently running and suspended coroutines. The coroutines are grouped by the dispatcher they are running on.



Now you can:

- Easily check the state of each coroutine.
- See the values of local and captured variables for both running and suspended coroutines.
- See a full coroutine creation stack, as well as a call stack inside the coroutine. The stack includes all frames with variable values, even those that would be lost during standard debugging.

If you need a full report containing the state of each coroutine and its stack, right-click inside the **Coroutines** tab, and then click **Get Coroutines Dump**. Currently, the coroutines dump is rather simple, but we're going to make it more readable and helpful in future versions of Kotlin.



Learn more about debugging coroutines in this blog post and IntelliJ IDEA documentation.

新编译器

The new Kotlin compiler is going to be really fast; it will unify all the supported platforms and provide an API for compiler extensions. It's a long-term project, and we've already completed several steps in Kotlin 1.4.0:

- 新的、更强大的类型推断算法 is enabled by default.
- 新的 JVM 与 JS IR 后端. They will become the default once we stabilize them.

新的更强大的类型推断算法

Kotlin 1.4 uses a new, more powerful type inference algorithm. This new algorithm was already available to try in Kotlin 1.3 by specifying a compiler option, and now it's used by default. You can find the full list of issues fixed in the new algorithm in YouTrack. Here you can find some of the most

noticeable improvements:

- 会自动推断类型的更多情况
- lambda 表达式最后一个表达式的智能转换
- 可调用引用的智能转换
- 属性委托的更佳推断
- 具有不同参数的 Java 接口的 SAM 转换
- Kotlin 中的 Java SAM 接口

会自动推断类型的更多情况

The new inference algorithm infers types for many cases where the old algorithm required you to specify them explicitly. For instance, in the following example the type of the lambda parameter `it` is correctly inferred to `String?`:

```
//sampleStart
val rulesMap: Map<String, (String?) -> Boolean> = mapOf(
    "weak" to { it != null },
    "medium" to { !it.isNullOrBlank() },
    "strong" to { it != null && "^[a-zA-Z0-9]+$".toRegex().matches(it) }
)
//sampleEnd

fun main() {
    println(rulesMap.getValue("weak")("abc!"))
    println(rulesMap.getValue("strong")("abc"))
    println(rulesMap.getValue("strong")("abc!"))
}
```

In Kotlin 1.3, you needed to introduce an explicit lambda parameter or replace `to` with a `Pair` constructor with explicit generic arguments to make it work.

lambda 表达式最后一个表达式的智能转换

In Kotlin 1.3, the last expression inside a lambda wasn't smart cast unless you specified the expected type. Thus, in the following example, Kotlin 1.3 infers `String?` as the type of the `result` variable:

```
val result = run {
    var str = currentValue()
    if (str == null) {
        str = "test"
    }
    str // the Kotlin compiler knows that str is not null here
}
// The type of 'result' is String? in Kotlin 1.3 and String in Kotlin 1.4
```

In Kotlin 1.4, thanks to the new inference algorithm, the last expression inside a lambda gets smart cast, and this new, more precise type is used to infer the resulting lambda type. Thus, the type of the `result` variable becomes `String`.

In Kotlin 1.3, you often needed to add explicit casts (either `!!` or type casts like `as String`) to make such cases work, and now these casts have become unnecessary.

可调用引用的智能转换

In Kotlin 1.3, you couldn't access a member reference of a smart cast type. Now in Kotlin 1.4 you can:

```
import kotlin.reflect.KFunction

sealed class Animal
class Cat : Animal() {
    fun meow() {
        println("meow")
    }
}

class Dog : Animal() {
    fun woof() {
        println("woof")
    }
}

//sampleStart
fun perform(animal: Animal) {
    val kFunction: KFunction<*> = when (animal) {
        is Cat -> animal::meow
        is Dog -> animal::woof
    }
    kFunction.call()
}
//sampleEnd

fun main() {
    perform(Cat())
}
```

You can use different member references `animal::meow` and `animal::woof` after the animal variable has been smart cast to specific types `Cat` and `Dog`. After type checks, you can access member references corresponding to subtypes.

属性委托的更佳推断

The type of a delegated property wasn't taken into account while analyzing the delegate expression which follows the `by` keyword. For instance, the following code didn't compile before, but now the compiler correctly infers the types of the `old` and `new` parameters as `String?`:

```
import kotlin.properties.Delegates

fun main() {
    var prop: String? by Delegates.observable(null) { p, old, new ->
        println("$old → $new")
    }
    prop = "abc"
    prop = "xyz"
}
```

具有不同参数的 Java 接口的 SAM 转换

Kotlin has supported SAM conversions for Java interfaces from the beginning, but there was one case that wasn't supported, which was sometimes annoying when working with existing Java libraries. If you called a Java method that took two SAM interfaces as parameters, both arguments needed to be either lambdas or regular objects. You couldn't pass one argument as a lambda and another as an object.

The new algorithm fixes this issue, and you can pass a lambda instead of a SAM interface in any case, which is the way you'd naturally expect it to work.

```
// FILE: A.java
public class A {
    public static void foo(Runnable r1, Runnable r2) {}
}
```

```
// FILE: test.kt
fun test(r1: Runnable) {
    A.foo(r1) {}  // Works in Kotlin 1.4
}
```

Kotlin 中的 Java SAM 接口

In Kotlin 1.4, you can use Kotlin 中的 Java SAM 接口 and apply SAM conversions to them.

```
import java.lang.Runnable

fun foo(r: Runnable) {}

fun test() {
```

```
    foo { } // OK
}
```

In Kotlin 1.3, you would have had to declare the function `foo` above in Java code to perform a SAM conversion.

统一的后端与可扩展性

In Kotlin, we have three backends that generate executables: Kotlin/JVM, Kotlin/JS, and Kotlin/Native. Kotlin/JVM and Kotlin/JS don't share much code since they were developed independently of each other. Kotlin/Native is based on a new infrastructure built around an intermediate representation (IR) for Kotlin code.

We are now migrating Kotlin/JVM and Kotlin/JS to the same IR. As a result, all three backends share a lot of logic and have a unified pipeline. This allows us to implement most features, optimizations, and bug fixes only once for all platforms. Both new IR-based back-ends are in Alpha.

A common backend infrastructure also opens the door for multiplatform compiler extensions. You will be able to plug into the pipeline and add custom processing and transformations that will automatically work for all platforms.

We encourage you to use our new JVM IR and JS IR backends, which are currently in Alpha, and share your feedback with us.

Kotlin/JVM

Kotlin 1.4.0 includes a number of JVM-specific improvements, such as:

- 新的 JVM IR 后端
- 在接口中生成默认方法的新模式
- 统一用于空检测的异常类型
- 在 JVM 字节码中的类型注解

新的 JVM IR 后端

Along with Kotlin/JS, we are migrating Kotlin/JVM to the unified IR backend, which allows us to implement most features and bug fixes once for all platforms. You will also be able to benefit from this by creating multiplatform extensions that will work for all platforms.

Kotlin 1.4.0 does not provide a public API for such extensions yet, but we are working closely with our partners, including Jetpack Compose, who are already building their compiler plugins using our new backend.

We encourage you to try out the new Kotlin/JVM backend, which is currently in Alpha, and to file any issues and feature requests to our issue tracker. This will help us to unify the compiler pipelines and bring compiler extensions like Jetpack Compose to the Kotlin community more quickly.

To enable the new JVM IR backend, specify an additional compiler option in your Gradle build script:

```
kotlinOptions.useIR = true
```

> If you enable Jetpack Compose, you will automatically be opted in to the new JVM backend without needing to specify the compiler option in `kotlinOptions`.
>
> 

When using the command-line compiler, add the compiler option `-Xuse-ir`.

> You can use code compiled by the new JVM IR backend only if you've enabled the new backend. Otherwise, you will get an error. Considering this, we don't recommend that library authors switch to the new backend in production.
>
> 

生成默认方法的新模式

When compiling Kotlin code to targets JVM 1.8 and above, you could compile non-abstract methods of Kotlin interfaces into Java's `default` methods. For this purpose, there was a mechanism that includes the `@JvmDefault` annotation for marking such methods and the `-Xjvm-default` compiler option that enables processing of this annotation.

In 1.4.0, we've added a new mode for generating default methods: `-Xjvm-default=all` compiles *all* non-abstract methods of Kotlin interfaces to `default` Java methods. For compatibility with the code that uses the interfaces compiled without `default`, we also added `all-compatibility` mode.

For more information about default methods in the Java interop, see the interoperability documentation and this blog post.

统一用于空检测的异常类型

Starting from Kotlin 1.4.0, all runtime null checks will throw a `java.lang.NullPointerException` instead of `KotlinNullPointerException`, `IllegalStateException`, `IllegalArgumentException`, and `TypeCastException`. This applies to: the `!!` operator, parameter null checks in the method preamble, platform-typed expression null checks, and the `as` operator with a non-nullable type. This doesn't apply to `lateinit` null checks and explicit library function calls like `checkNotNull` or `requireNotNull`.

This change increases the number of possible null check optimizations that can be performed either by the Kotlin compiler or by various kinds of bytecode processing tools, such as the Android R8 optimizer.

Note that from a developer's perspective, things won't change that much: the Kotlin code will throw exceptions with the same error messages as before. The type of exception changes, but the information passed stays the same.

在 JVM 字节码中的类型注解

Kotlin can now generate type annotations in the JVM bytecode (target version 1.8+), so that they become available in Java reflection at runtime. To emit the type annotation in the bytecode, follow these steps:

1. Make sure that your declared annotation has a proper annotation target (Java's `ElementType.TYPE_USE` or Kotlin's `AnnotationTarget.TYPE`) and retention (`AnnotationRetention.RUNTIME`).
2. Compile the annotation class declaration to JVM bytecode target version 1.8+. You can specify it with `-jvm-target=1.8` compiler option.
3. Compile the code that uses the annotation to JVM bytecode target version 1.8+ (`-jvm-target=1.8`) and add the `-Xemit-jvm-type-annotations` compiler option.

Note that the type annotations from the standard library aren't emitted in the bytecode for now because the standard library is compiled with the target version 1.6.

So far, only the basic cases are supported:

- Type annotations on method parameters, method return types and property types;
- Invariant projections of type arguments, such as `Smth<@Ann Foo>`, `Array<@Ann Foo>`.

In the following example, the `@Foo` annotation on the `String` type can be emitted to the bytecode and then used by the library code:

```
@Target(AnnotationTarget.TYPE)
annotation class Foo

class A {
    fun foo(): @Foo String = "OK"
}
```

Kotlin/JS

On the JS platform, Kotlin 1.4.0 provides the following improvements:

- 新的 Gradle DSL
- 新的 JS IR 后端

新的 Gradle DSL

The `kotlin.js` Gradle plugin comes with an adjusted Gradle DSL, which provides a number of new configuration options and is more closely aligned to the DSL used by the `kotlin-multiplatform` plugin. Some of the most impactful changes include:

- Explicit toggles for the creation of executable files via `binaries.executable()`. Read more about the executing Kotlin/JS and its environment here.
- Configuration of webpack's CSS and style loaders from within the Gradle configuration via `cssSupport`. Read more about using CSS and style loaders here.
- Improved management for npm dependencies, with mandatory version numbers or semver version ranges, as well as support for *development*, *peer*, and *optional* npm dependencies using `devNpm`, `optionalNpm` and `peerNpm`. Read more about dependency management for npm packages directly from Gradle here.
- Stronger integrations for Dukat, the generator for Kotlin external declarations. External declarations can now be generated at build time, or can be manually generated via a Gradle task.

新的 JS IR 后端

The IR backend for Kotlin/JS, which currently has Alpha stability, provides some new functionality specific to the Kotlin/JS target which is focused around the generated code size through dead code elimination, and improved interoperation with JavaScript and TypeScript, among others.

To enable the Kotlin/JS IR backend, set the key `kotlin.js.compiler=ir` in your `gradle.properties`, or pass the `IR` compiler type to the `js` function of your Gradle build script:

```
kotlin {
    js(IR) { // or: LEGACY, BOTH
        // ...
    }
    binaries.executable()
}
```

For more detailed information about how to configure the new backend, check out the Kotlin/JS IR compiler documentation.

With the new @JsExport annotation and the ability to **generate TypeScript definitions from Kotlin code**, the Kotlin/JS IR compiler backend improves JavaScript & TypeScript interoperability. This also makes it easier to integrate Kotlin/JS code with existing tooling, to create **hybrid applications** and leverage code-sharing functionality in multiplatform projects.

Learn more about the available features in the Kotlin/JS IR compiler backend.

Kotlin/Native

In 1.4.0, Kotlin/Native got a significant number of new features and improvements, including:

- 在 Swift 与 Objective-C 中支持挂起函数
- 默认支持 Objective-C 泛型
- Objective-C/Swift 互操作中的异常处理
- 默认在苹果目标平台生成 release 版 .dSYM
- 性能提升
- 简化了 CocoaPods 依赖项的管理

在 Swift 与 Objective-C 中支持 kotlin 的挂起函数

In 1.4.0, we add the basic support for suspending functions in Swift and Objective-C. Now, when you compile a Kotlin module into an Apple framework, suspending functions are available in it as functions with callbacks (`completionHandler` in the Swift/Objective-C terminology). When you have such functions in the generated framework's header, you can call them from your Swift or Objective-C code and even override them.

For example, if you write this Kotlin function:

```
suspend fun queryData(id: Int): String = ...
```

...then you can call it from Swift like so:

```
queryData(id: 17) { result, error in
   if let e = error {
       print("ERROR: \(e)")
   } else {
       print(result!)
   }
}
```

Learn more about using suspending functions in Swift and Objective-C.

默认支持 Objective-C 泛型

Previous versions of Kotlin provided experimental support for generics in Objective-C interop. Since 1.4.0, Kotlin/Native generates Apple frameworks with generics from Kotlin code by default. In some cases, this may break existing Objective-C or Swift code calling Kotlin frameworks. To have the

framework header written without generics, add the `-Xno-objc-generics` compiler option.

```
kotlin {
    targets.withType<org.jetbrains.kotlin.gradle.plugin.mpp.KotlinNativeTarget> {
        binaries.all {
            freeCompilerArgs += "-Xno-objc-generics"
        }
    }
}
```

Please note that all specifics and limitations listed in the documentation on interoperability with Objective-C are still valid.

Objective-C/Swift 互操作中的异常处理

In 1.4.0, we slightly change the Swift API generated from Kotlin with respect to the way exceptions are translated. There is a fundamental difference in error handling between Kotlin and Swift. All Kotlin exceptions are unchecked, while Swift has only checked errors. Thus, to make Swift code aware of expected exceptions, Kotlin functions should be marked with a `@Throws` annotation specifying a list of potential exception classes.

When compiling to Swift or the Objective-C framework, functions that have or are inheriting `@Throws` annotation are represented as `NSError*`-producing methods in Objective-C and as `throws` methods in Swift.

Previously, any exceptions other than `RuntimeException` and `Error` were propagated as `NSError`. Now this behavior changes: now `NSError` is thrown only for exceptions that are instances of classes specified as parameters of `@Throws` annotation (or their subclasses). Other Kotlin exceptions that reach Swift/Objective-C are considered unhandled and cause program termination.

默认在苹果目标平台生成 release 版 .dSYM

Starting with 1.4.0, the Kotlin/Native compiler produces debug symbol files (`.dSYM`s) for release binaries on Darwin platforms by default. This can be disabled with the `-Xadd-light-debug=disable` compiler option. On other platforms, this option is disabled by default. To toggle this option in Gradle, use:

```
kotlin {
    targets.withType<org.jetbrains.kotlin.gradle.plugin.mpp.KotlinNativeTarget> {
        binaries.all {
            freeCompilerArgs += "-Xadd-light-debug={enable|disable}"
        }
    }
}
```

Learn more about crash report symbolication.

性能提升

Kotlin/Native has received a number of performance improvements that speed up both the development process and execution. Here are some examples:

- To improve the speed of object allocation, we now offer the mimalloc memory allocator as an alternative to the system allocator. mimalloc works up to two times faster on some benchmarks. Currently, the usage of mimalloc in Kotlin/Native is experimental; you can switch to it using the `-Xallocator=mimalloc` compiler option.
- We've reworked how C interop libraries are built. With the new tooling, Kotlin/Native produces interop libraries up to 4 times as fast as before, and artifacts are 25% to 30% the size they used to be.
- Overall runtime performance has improved because of optimizations in GC. This improvement will be especially apparent in projects with a large number of long-lived objects. `HashMap` and `HashSet` collections now work faster by escaping redundant boxing.
- In 1.3.70 we introduced two new features for improving the performance of Kotlin/Native compilation: caching project dependencies and running the compiler from the Gradle daemon. Since that time, we've managed to fix numerous issues and improve the overall stability of these features.

简化了 CocoaPods 依赖项的管理

Previously, once you integrated your project with the dependency manager CocoaPods, you could build an iOS, macOS, watchOS, or tvOS part of your project only in Xcode, separate from other parts of your multiplatform project. These other parts could be built in IntelliJ IDEA.

Moreover, every time you added a dependency on an Objective-C library stored in CocoaPods (Pod library), you had to switch from IntelliJ IDEA to Xcode, call `pod install`, and run the Xcode build there.

Now you can manage Pod dependencies right in IntelliJ IDEA while enjoying the benefits it provides for working with code, such as code highlighting and completion. You can also build the whole Kotlin project with Gradle, without having to switch to Xcode. This means you only have to go to Xcode when you need to write Swift/Objective-C code or run your application on a simulator or device.

Now you can also work with Pod libraries stored locally.

Depending on your needs, you can add dependencies between:

- A Kotlin project and Pod libraries stored remotely in the CocoaPods repository or stored locally

on your machine.
- A Kotlin Pod (Kotlin project used as a CocoaPods dependency) and an Xcode project with one or more targets.

Complete the initial configuration, and when you add a new dependency to `cocoapods`, just re-import the project in IntelliJ IDEA. The new dependency will be added automatically. No additional steps are required.

Learn how to add dependencies.

Kotlin 多平台

> Support for multiplatform projects is in Alpha. It may change incompatibly and require manual migration in the future. We appreciate your feedback on it in YouTrack.
>
> 

Kotlin 多平台 reduces time spent writing and maintaining the same code for different platforms while retaining the flexibility and benefits of native programming. We continue investing our effort in multiplatform features and improvements:

- 使用分层项目结构在多个目标中共享代码
- 在分层结构中利用原生库
- 只需指定一次 kotlinx 依赖项

> Multiplatform projects require Gradle 6.0 or later.
>
> 

使用分层项目结构在多个目标中共享代码

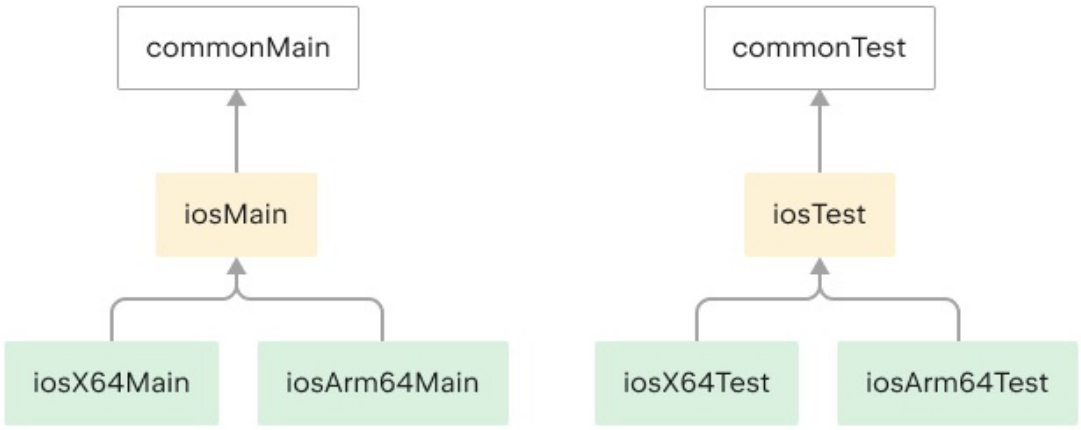
With the new hierarchical project structure support, you can share code among several platforms in a multiplatform project.

Previously, any code added to a multiplatform project could be placed either in a platform-specific source set, which is limited to one target and can't be reused by any other platform, or in a common source set, like `commonMain` or `commonTest`, which is shared across all the platforms in the project. In the common source set, you could only call a platform-specific API by using an `expect` declaration that needs platform-specific `actual` implementations.

This made it easy to share code on all platforms, but it was not so easy to share between only some of the targets, especially similar ones that could potentially reuse a lot of the common logic and third-party APIs.

For example, in a typical multiplatform project targeting iOS, there are two iOS-related targets: one for iOS ARM64 devices, and the other for the x64 simulator. They have separate platform-specific source sets, but in practice, there is rarely a need for different code for the device and simulator, and their dependencies are much alike. So iOS-specific code could be shared between them.

Apparently, in this setup, it would be desirable to have a *shared source set for two iOS targets*, with Kotlin/Native code that could still directly call any of the APIs that are common to both the iOS device and the simulator.

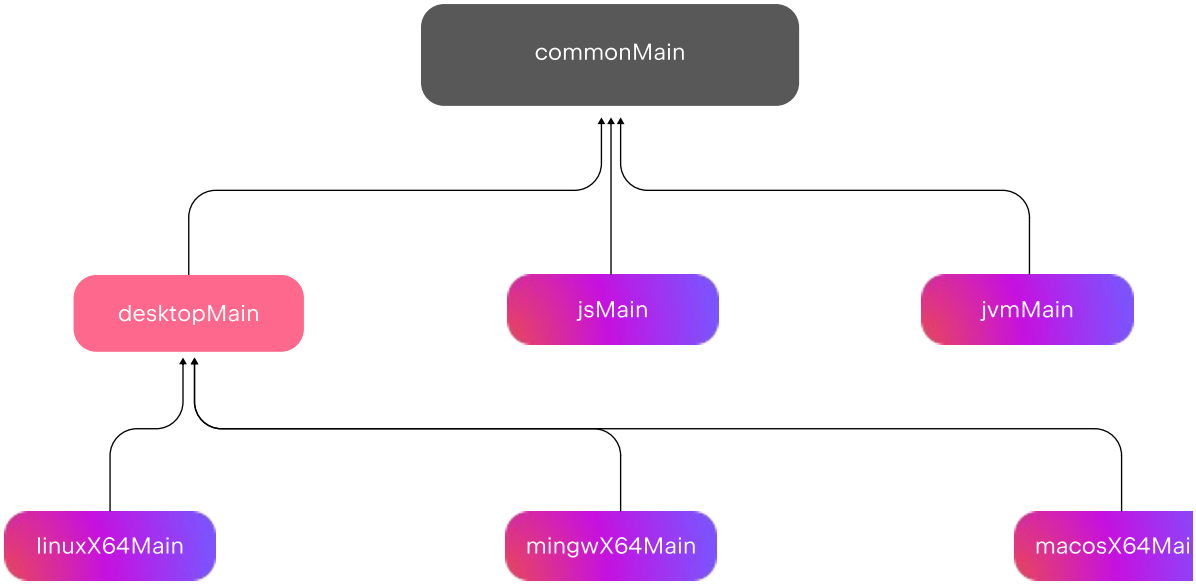


Now you can do this with the hierarchical project structure support, which infers and adapts the API and language features available in each source set based on which targets consume them.

For common combinations of targets, you can create a hierarchical structure with target shortcuts. For example, create two iOS targets and the shared source set shown above with the `ios()` shortcut:

```
kotlin {
    ios() // iOS device and simulator targets; iosMain and iosTest source sets
}
```

For other combinations of targets, create a hierarchy manually by connecting the source sets with the `dependsOn` relation.



【Kotlin】

```
kotlin{
    sourceSets {
        val desktopMain by creating {
            dependsOn(commonMain)
        }
        val linuxX64Main by getting {
            dependsOn(desktopMain)
        }
        val mingwX64Main by getting {
            dependsOn(desktopMain)
        }
        val macosX64Main by getting {
            dependsOn(desktopMain)
        }
    }
}
```

【Groovy】

```
kotlin {
    sourceSets {
        desktopMain {
            dependsOn(commonMain)
        }
        linuxX64Main {
            dependsOn(desktopMain)
```

```
        }
        mingwX64Main {
            dependsOn(desktopMain)
        }
        macosX64Main {
            dependsOn(desktopMain)
        }
    }
}
```

Thanks to the hierarchical project structure, libraries can also provide common APIs for a subset of targets. Learn more about sharing code in libraries.

在分层结构中利用原生库

You can use platform-dependent libraries, such as Foundation, UIKit, and POSIX, in source sets shared among several native targets. This can help you share more native code without being limited by platform-specific dependencies.

No additional steps are required – everything is done automatically. IntelliJ IDEA will help you detect common declarations that you can use in the shared code.

Learn more about usage of platform-dependent libraries.

只需指定一次依赖项

From now on, instead of specifying dependencies on different variants of the same library in shared and platform-specific source sets where it is used, you should specify a dependency only once in the shared source set.

> **【Kotlin】**

```
kotlin {
    sourceSets {
        val commonMain by getting {
            dependencies {
                implementation("org.jetbrains.kotlinx:kotlinx-coroutines-core:1.10.2")
            }
        }
    }
}
```

> 【Groovy】

```
kotlin {
    sourceSets {
        commonMain {
            dependencies {
                implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-core:1.10.2'
            }
        }
    }
}
```

Don't use kotlinx library artifact names with suffixes specifying the platform, such as `-common`, `-native`, or similar, as they are NOT supported anymore. Instead, use the library base artifact name, which in the example above is `kotlinx-coroutines-core`.

However, the change doesn't currently affect:

- The `stdlib` library – starting from Kotlin 1.4.0, the stdlib dependency is added automatically.
- The `kotlin.test` library – you should still use `test-common` and `test-annotations-common`. These dependencies will be addressed later.

If you need a dependency only for a specific platform, you can still use platform-specific variants of standard and kotlinx libraries with such suffixes as `-jvm` or `-js`, for example `kotlinx-coroutines-core-jvm`.

Learn more about configuring dependencies.

Gradle 项目改进

Besides Gradle project features and improvements that are specific to Kotlin 多平台, Kotlin/JVM, Kotlin/Native, and Kotlin/JS, there are several changes applicable to all Kotlin Gradle projects:

- 现在默认添加了对标准库的依赖
- Kotlin 项目需要最近版本的 Gradle
- 改进了 IDE 对 Kotlin Gradle DSL 的支持

默认添加了对标准库的依赖

You no longer need to declare a dependency on the `stdlib` library in any Kotlin Gradle project, including a multiplatform one. The dependency is added by default.

The automatically added standard library will be the same version of the Kotlin Gradle plugin, since they have the same versioning.

For platform-specific source sets, the corresponding platform-specific variant of the library is used, while a common standard library is added to the rest. The Kotlin Gradle plugin will select the appropriate JVM standard library depending on the `kotlinOptions.jvmTarget` compiler option of your Gradle build script.

Learn how to change the default behavior.

Kotlin 项目的最低 Gradle 版本

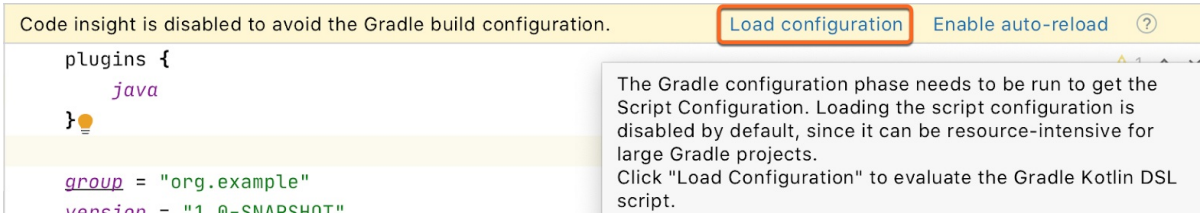
To enjoy the new features in your Kotlin projects, update Gradle to the latest version. Multiplatform projects require Gradle 6.0 or later, while other Kotlin projects work with Gradle 5.4 or later.

改进了 IDE 对 *.gradle.kts 的支持

In 1.4.0, we continued improving the IDE support for Gradle Kotlin DSL scripts (`*.gradle.kts` files). Here is what the new version brings:

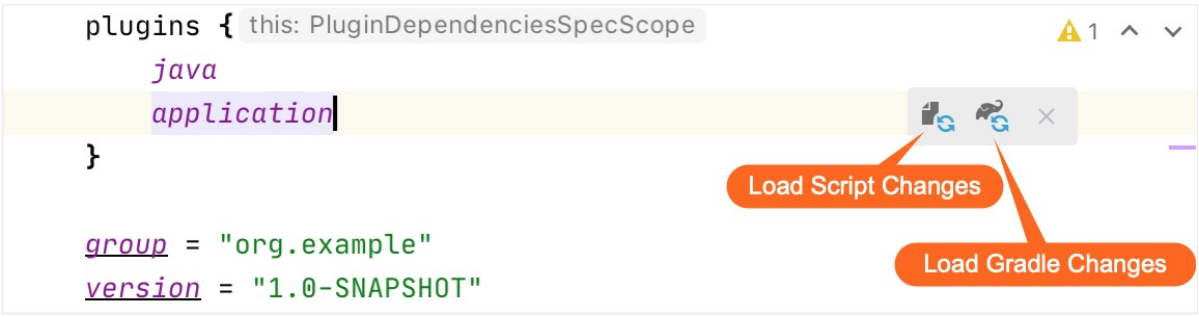
- *Explicit loading of script configurations* for better performance. Previously, the changes you make to the build script were loaded automatically in the background. To improve the performance, we've disabled the automatic loading of build script configuration in 1.4.0. Now the IDE loads the changes only when you explicitly apply them.

  In Gradle versions earlier than 6.0, you need to manually load the script configuration by clicking **Load Configuration** in the editor.

  In Gradle 6.0 and above, you can explicitly apply changes by clicking **Load Gradle Changes** or by reimporting the Gradle project.

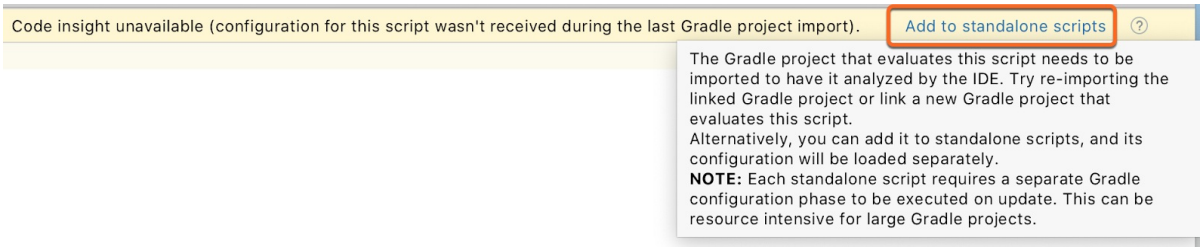
  We've added one more action in IntelliJ IDEA 2020.1 with Gradle 6.0 and above – **Load Script Configurations**, which loads changes to the script configurations without updating the whole project. This takes much less time than reimporting the whole project.



You should also **Load Script Configurations** for newly created scripts or when you open a project with new Kotlin plugin for the first time.

With Gradle 6.0 and above, you are now able to load all scripts at once as opposed to the previous implementation where they were loaded individually. Since each request requires the Gradle configuration phase to be executed, this could be resource-intensive for large Gradle projects.

Currently, such loading is limited to `build.gradle.kts` and `settings.gradle.kts` files (please vote for the related issue). To enable highlighting for `init.gradle.kts` or applied script plugins, use the old mechanism – adding them to standalone scripts. Configuration for that scripts will be loaded separately when you need it. You can also enable auto-reload for such scripts.



- *Better error reporting*. Previously you could only see errors from the Gradle Daemon in separate log files. Now the Gradle Daemon returns all the information about errors directly and shows it in the Build tool window. This saves you both time and effort.

标准库

Here is the list of the most significant changes to the Kotlin standard library in 1.4.0:

- 公共异常处理 API
- 数组与集合的新函数
- 一些字符串操作函数
- 一些位操作
- 属性委托改进
- 由 KType 转换为 Java Type
- Kotlin 反射的 Proguard 配置

- 现有 API 改进
- stdlib 构件的 module-info 描述符
- 弃用项
- 排除弃用的实验性协程

公共异常处理 API

The following API elements have been moved to the common library:

- `Throwable.stackTraceToString()` extension function, which returns the detailed description of this throwable with its stack trace, and `Throwable.printStackTrace()`, which prints this description to the standard error output.
- `Throwable.addSuppressed()` function, which lets you specify the exceptions that were suppressed in order to deliver the exception, and the `Throwable.suppressedExceptions` property, which returns a list of all the suppressed exceptions.
- `@Throws` annotation, which lists exception types that will be checked when the function is compiled to a platform method (on JVM or native platforms).

数组与集合的新函数

集合

In 1.4.0, the standard library includes a number of useful functions for working with **collections**:

- `setOfNotNull()`, which makes a set consisting of all the non-null items among the provided arguments.

```
fun main() {
//sampleStart
    val set = setOfNotNull(null, 1, 2, 0, null)
    println(set)
//sampleEnd
}
```

- `shuffled()` for sequences.

```
fun main() {
//sampleStart
    val numbers = (0 until 50).asSequence()
    val result = numbers.map { it * 2 }.shuffled().take(5)
    println(result.toList()) //five random even numbers below 100
//sampleEnd
}
```

- `*Indexed()` counterparts for `onEach()` and `flatMap()`. The operation that they apply to the collection elements has the element index as a parameter.

```
fun main() {
//sampleStart
    listOf("a", "b", "c", "d").onEachIndexed {
        index, item -> println(index.toString() + ":" + item)
    }

   val list = listOf("hello", "kot", "lin", "world")
          val kotlin = list.flatMapIndexed { index, item ->
              if (index in 1..2) item.toList() else emptyList()
          }
//sampleEnd
          println(kotlin)
}
```

- `*OrNull()` counterparts `randomOrNull()`, `reduceOrNull()`, and `reduceIndexedOrNull()`. They return `null` on empty collections.

```
fun main() {
//sampleStart
     val empty = emptyList<Int>()
     empty.reduceOrNull { a, b -> a + b }
     //empty.reduce { a, b -> a + b } // Exception: Empty collection can't be re
duced.
//sampleEnd
}
```

- `runningFold()`, its synonym `scan()`, and `runningReduce()` apply the given operation to the collection elements sequentially, similarly to `fold()` and `reduce()`; the difference is that these new functions return the whole sequence of intermediate results.

```
fun main() {
//sampleStart
    val numbers = mutableListOf(0, 1, 2, 3, 4, 5)
    val runningReduceSum = numbers.runningReduce { sum, item -> sum + item }
    val runningFoldSum = numbers.runningFold(10) { sum, item -> sum + item }
//sampleEnd
    println(runningReduceSum.toString())
    println(runningFoldSum.toString())
}
```

- `sumOf()` takes a selector function and returns a sum of its values for all elements of a collection. `sumOf()` can produce sums of the types `Int`, `Long`, `Double`, `UInt`, and `ULong`. On the JVM, `BigInteger` and `BigDecimal` are also available.

```
data class OrderItem(val name: String, val price: Double, val count: Int)

fun main() {
//sampleStart
    val order = listOf<OrderItem>(
        OrderItem("Cake", price = 10.0, count = 1),
        OrderItem("Coffee", price = 2.5, count = 3),
        OrderItem("Tea", price = 1.5, count = 2))

    val total = order.sumOf { it.price * it.count } // Double
    val count = order.sumOf { it.count } // Int
//sampleEnd
    println("You've ordered $count items that cost $total in total")
}
```

- The `min()` and `max()` functions have been renamed to `minOrNull()` and `maxOrNull()` to comply with the naming convention used across the Kotlin collections API. An `*OrNull` suffix in the function name means that it returns `null` if the receiver collection is empty. The same applies to `minBy()`, `maxBy()`, `minWith()`, `maxWith()` – in 1.4, they have `*OrNull()` synonyms.
- The new `minOf()` and `maxOf()` extension functions return the minimum and the maximum value of the given selector function on the collection items.

```
data class OrderItem(val name: String, val price: Double, val count: Int)

fun main() {
//sampleStart
    val order = listOf<OrderItem>(
        OrderItem("Cake", price = 10.0, count = 1),
        OrderItem("Coffee", price = 2.5, count = 3),
        OrderItem("Tea", price = 1.5, count = 2))
    val highestPrice = order.maxOf { it.price }
//sampleEnd
    println("The most expensive item in the order costs $highestPrice")
}
```

```
There are also `minOfWith()` and `maxOfWith()`, which take a `Comparator` as an argume
nt, and `*OrNull()` versions
```

of all four functions that return `null` on empty collections.

- New overloads for `flatMap` and `flatMapTo` let you use transformations with return types that don't match the receiver type, namely:
  - Transformations to `Sequence` on `Iterable`, `Array`, and `Map`
  - Transformations to `Iterable` on `Sequence`

```
fun main() {
//sampleStart
  val list = listOf("kot", "lin")
  val lettersList = list.flatMap { it.asSequence() }
  val lettersSeq = list.asSequence().flatMap { it.toList() }
//sampleEnd
  println(lettersList)
  println(lettersSeq.toList())
}
```

- `removeFirst()` and `removeLast()` shortcuts for removing elements from mutable lists, and `*orNull()` counterparts of these functions.

数组

To provide a consistent experience when working with different container types, we've also added new functions for **arrays**:

- `shuffle()` puts the array elements in a random order.
- `onEach()` performs the given action on each array element and returns the array itself.
- `associateWith()` and `associateWithTo()` build maps with the array elements as keys.
- `reverse()` for array subranges reverses the order of the elements in the subrange.
- `sortDescending()` for array subranges sorts the elements in the subrange in descending order.
- `sort()` and `sortWith()` for array subranges are now available in the common library.

```
fun main() {
//sampleStart
    var language = ""
    val letters = arrayOf("k", "o", "t", "l", "i", "n")
    val fileExt = letters.onEach { language += it }
       .filterNot { it in "aeuio" }.take(2)
       .joinToString(prefix = ".", separator = "")
    println(language) // "kotlin"
    println(fileExt) // ".kt"

    letters.shuffle()
    letters.reverse(0, 3)
    letters.sortDescending(2, 5)
    println(letters.contentToString()) // [k, o, t, l, i, n]
//sampleEnd
}
```

Additionally, there are new functions for conversions between `CharArray` / `ByteArray` and `String`:

- `ByteArray.decodeToString()` and `String.encodeToByteArray()`

- `CharArray.concatToString()` and `String.toCharArray()`

```
fun main() {
//sampleStart
    val str = "kotlin"
    val array = str.toCharArray()
    println(array.concatToString())
//sampleEnd
}
```

ArrayDeque

We've also added the `ArrayDeque` class – an implementation of a double-ended queue. A double-ended queue lets you add or remove elements both at the beginning or end of the queue in an amortized constant time. You can use a double-ended queue by default when you need a queue or a stack in your code.

```
fun main() {
    val deque = ArrayDeque(listOf(1, 2, 3))

    deque.addFirst(0)
    deque.addLast(4)
    println(deque) // [0, 1, 2, 3, 4]

    println(deque.first()) // 0
    println(deque.last()) // 4

    deque.removeFirst()
    deque.removeLast()
    println(deque) // [1, 2, 3]
}
```

The `ArrayDeque` implementation uses a resizable array underneath: it stores the contents in a circular buffer, an `Array`, and resizes this `Array` only when it becomes full.

一些字符串操作函数

The standard library in 1.4.0 includes a number of improvements in the API for string manipulation:

- `StringBuilder` has useful new extension functions: `set()`, `setRange()`, `deleteAt()`, `deleteRange()`, `appendRange()`, and others.

  ```
  fun main() {
  //sampleStart
      val sb = StringBuilder("Bye Kotlin 1.3.72")
      sb.deleteRange(0, 3)
  ```

```
        sb.insertRange(0, "Hello", 0 ,5)
        sb.set(15, '4')
        sb.setRange(17, 19, "0")
        print(sb.toString())
    //sampleEnd
    }
```

- Some existing functions of `StringBuilder` are available in the common library. Among them are `append()`, `insert()`, `substring()`, `setLength()`, and more.
- New functions `Appendable.appendLine()` and `StringBuilder.appendLine()` have been added to the common library. They replace the JVM-only `appendln()` functions of these classes.

```
fun main() {
//sampleStart
    println(buildString {
        appendLine("Hello,")
        appendLine("world")
    })
//sampleEnd
}
```

一些位操作

New functions for bit manipulations:

- `countOneBits()`
- `countLeadingZeroBits()`
- `countTrailingZeroBits()`
- `takeHighestOneBit()`
- `takeLowestOneBit()`
- `rotateLeft()` and `rotateRight()` (experimental)

```
fun main() {
//sampleStart
    val number = "1010000".toInt(radix = 2)
    println(number.countOneBits())
    println(number.countTrailingZeroBits())
    println(number.takeHighestOneBit().toString(2))
//sampleEnd
}
```

属性委托改进

In 1.4.0, we have added new features to improve your experience with delegated properties in Kotlin:

- Now a property can be delegated to another property.
- A new interface `PropertyDelegateProvider` helps create delegate providers in a single declaration.
- `ReadWriteProperty` now extends `ReadOnlyProperty` so you can use both of them for read-only properties.

Aside from the new API, we've made some optimizations that reduce the resulting bytecode size. These optimizations are described in this blog post.

Learn more about delegated properties.

由 KType 转换为 Java Type

A new extension property `KType.javaType` (currently experimental) in the stdlib helps you obtain a `java.lang.reflect.Type` from a Kotlin type without using the whole `kotlin-reflect` dependency.

```
import kotlin.reflect.javaType
import kotlin.reflect.typeOf

@OptIn(ExperimentalStdlibApi::class)
inline fun <reified T> accessReifiedTypeArg() {
   val kType = typeOf<T>()
   println("Kotlin type: $kType")
   println("Java type: ${kType.javaType}")
}

@OptIn(ExperimentalStdlibApi::class)
fun main() {
   accessReifiedTypeArg<String>()
   // Kotlin type: kotlin.String
   // Java type: class java.lang.String

   accessReifiedTypeArg<List<String>>()
   // Kotlin type: kotlin.collections.List<kotlin.String>
   // Java type: java.util.List<java.lang.String>
}
```

Kotlin 反射的 Proguard 配置

Starting from 1.4.0, we have embedded Proguard/R8 configurations for Kotlin Reflection in `kotlin-reflect.jar` . With this in place, most Android projects using R8 or Proguard should work with kotlin-reflect without needing any additional configuration. You no longer need to copy-paste the Proguard rules for kotlin-reflect internals. But note that you still need to explicitly list all the APIs you're going to reflect on.

现有 API 改进

- Several functions now work on null receivers, for example:
  - `toBoolean()` on strings
  - `contentEquals()`, `contentHashcode()`, `contentToString()` on arrays
- `NaN`, `NEGATIVE_INFINITY`, and `POSITIVE_INFINITY` in `Double` and `Float` are now defined as `const`, so you can use them as annotation arguments.
- New constants `SIZE_BITS` and `SIZE_BYTES` in `Double` and `Float` contain the number of bits and bytes used to represent an instance of the type in binary form.
- The `maxOf()` and `minOf()` top-level functions can accept a variable number of arguments (`vararg`).

stdlib 构件的 module-info 描述符

Kotlin 1.4.0 adds `module-info.java` module information to default standard library artifacts. This lets you use them with jlink tool, which generates custom Java runtime images containing only the platform modules that are required for your app. You could already use jlink with Kotlin standard library artifacts, but you had to use separate artifacts to do so – the ones with the "modular" classifier – and the whole setup wasn't straightforward.

In Android, make sure you use the Android Gradle plugin version 3.2 or higher, which can correctly process jar files with module-info.

弃用项

Double 与 Float 的 toShort() 与 toByte()

We've deprecated the functions `toShort()` and `toByte()` on `Double` and `Float` because they could lead to unexpected results because of the narrow value range and smaller variable size.

To convert floating-point numbers to `Byte` or `Short`, use the two-step conversion: first, convert them to `Int`, and then convert them again to the target type.

浮点数组的 contains()、 indexOf() 与 lastIndexOf()

We've deprecated the `contains()`, `indexOf()`, and `lastIndexOf()` extension functions of `FloatArray` and `DoubleArray` because they use the IEEE 754 standard equality, which contradicts the total order equality in some corner cases. See this issue for details.

min() 与 max() 集合函数

We've deprecated the `min()` and `max()` collection functions in favor of `minOrNull()` and `maxOrNull()`, which more properly reflect their behavior – returning `null` on empty collections. See this issue for details.

排除弃用的实验性协程

The `kotlin.coroutines.experimental` API was deprecated in favor of kotlin.coroutines in 1.3.0. In 1.4.0, we're completing the deprecation cycle for `kotlin.coroutines.experimental` by removing it from the standard library. For those who still use it on the JVM, we've provided a compatibility artifact `kotlin-coroutines-experimental-compat.jar` with all the experimental coroutines APIs. We've published it to Maven, and we include it in the Kotlin distribution alongside the standard library.

稳定版 JSON 序列化

With Kotlin 1.4.0, we are shipping the first stable version of kotlinx.serialization - 1.0.0-RC. Now we are pleased to declare the JSON serialization API in `kotlinx-serialization-core` (previously known as `kotlinx-serialization-runtime`) stable. Libraries for other serialization formats remain experimental, along with some advanced parts of the core library.

We have significantly reworked the API for JSON serialization to make it more consistent and easier to use. From now on, we'll continue developing the JSON serialization API in a backward-compatible manner. However, if you have used previous versions of it, you'll need to rewrite some of your code when migrating to 1.0.0-RC. To help you with this, we also offer the **Kotlin Serialization Guide** – the complete set of documentation for `kotlinx.serialization`. It will guide you through the process of using the most important features and it can help you address any issues that you might face.

> **Note**: `kotlinx-serialization` 1.0.0-RC only works with Kotlin compiler 1.4. Earlier compiler versions are not compatible.

脚本与 REPL

In 1.4.0, scripting in Kotlin benefits from a number of functional and performance improvements along with other updates. Here are some of the key changes:

- 新的依赖项解析 API
- 新的 REPL API
- 已编译脚本缓存
- 构件重命名

To help you become more familiar with scripting in Kotlin, we've prepared a project with examples. It contains examples of the standard scripts (`*.main.kts`) and examples of uses of the Kotlin Scripting API and custom script definitions. Please give it a try and share your feedback using our issue tracker.

新的依赖项解析 API

In 1.4.0, we've introduced a new API for resolving external dependencies (such as Maven artifacts), along with implementations for it. This API is published in the new artifacts `kotlin-scripting-dependencies` and `kotlin-scripting-dependencies-maven` . The previous dependency resolution functionality in `kotlin-script-util` library is now deprecated.

新的 REPL API

The new experimental REPL API is now a part of the Kotlin Scripting API. There are also several implementations of it in the published artifacts, and some have advanced functionality, such as code completion. We use this API in the Kotlin Jupyter kernel and now you can try it in your own custom shells and REPLs.

已编译脚本缓存

The Kotlin Scripting API now provides the ability to implement a compiled scripts cache, significantly speeding up subsequent executions of unchanged scripts. Our default advanced script implementation `kotlin-main-kts` already has its own cache.

构件重命名

In order to avoid confusion about artifact names, we've renamed `kotlin-scripting-jsr223-embeddable` and `kotlin-scripting-jvm-host-embeddable` to just `kotlin-scripting-jsr223` and `kotlin-scripting-jvm-host` . These artifacts depend on the `kotlin-compiler-embeddable` artifact, which shades the bundled third-party libraries to avoid usage conflicts. With this renaming, we're making the usage of `kotlin-compiler-embeddable` (which is safer in general) the default for scripting artifacts. If, for some reason, you need artifacts that depend on the unshaded `kotlin-`

`compiler`, use the artifact versions with the `-unshaded` suffix, such as `kotlin-scripting-jsr223-unshaded`. Note that this renaming affects only the scripting artifacts that are supposed to be used directly; names of other artifacts remain unchanged.

迁移到 Kotlin 1.4.0

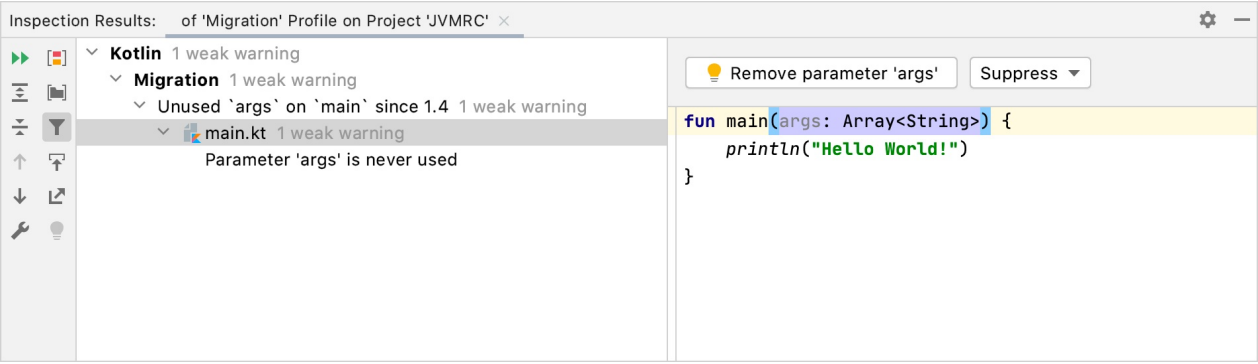
The Kotlin plugin's migration tools help you migrate your projects from earlier versions of Kotlin to 1.4.0.

Just change the Kotlin version to `1.4.0` and re-import your Gradle or Maven project. The IDE will then ask you about migration.

If you agree, it will run migration code inspections that will check your code and suggest corrections for anything that doesn't work or that is not recommended in 1.4.0.



Code inspections have different severity levels, to help you decide which suggestions to accept and which to ignore.



Kotlin 1.4.0 is a feature release and therefore can bring incompatible changes to the language. Find the detailed list of such changes in the **Compatibility Guide for Kotlin 1.4**.

Kotlin 1.3 的新特性

发布于：*2018-10-29*

协程正式发布

历经了漫长而充足的的测试，协程终于正式发布了！这意味着自 Kotlin 1.3 起，协程的语言支持与 API 已完全稳定。参见新版协程概述。

Kotlin 1.3 引入了挂起函数的可调用引用以及在反射 API 中对协程的支持。

Kotlin/Native

Kotlin 1.3 继续改进与完善原生平台。详情请参见 Kotlin/Native 概述。

多平台项目

在 1.3 中，我们完全修改了多平台项目的模型，以提高表现力与灵活性， 并使共享公共代码更加容易。此外，多平台项目现在也支持 Kotlin/Native！

与旧版模型的主要区别在于：

- 在旧版模型中，需要将公共代码与平台相关代码分别放在独立的模块中，以 `expectedBy` 依赖项链接。 现在，公共代码与平台相关代码放在相同模块的不同源根（source root）中，使项目更易于配置。
- 对于不同的已支持平台，现在有大量的预设的平台配置。
- 更改了依赖配置)； 现在为每个源根分别指定依赖项。
- 源代码集（source set）现在可以在任意平台子集之间共享（例如，在一个目标平台为 JS、Android 与 iOS 的模块中，可以有一个只在 Android 与 iOS 之间共享的源代码集）。
- 现在支持发布多平台库了。

更多相关信息，请参考多平台程序设计文档。

契约

Kotlin 编译器会做大量的静态分析工作，以提供警告并减少模版代码。其中最显著的特性之一就是智能转换——能够根据类型检测自动转换类型。

```
fun foo(s: String?) {
    if (s != null) s.length // 编译器自动将“s”转换为“String”
}
```

然而，一旦将这些检测提取到单独的函数中，所有智能转换都立即消失了：

```
fun String?.isNotNull(): Boolean = this != null

fun foo(s: String?) {
    if (s.isNotNull()) s.length // 没有智能转换 :(
}
```

为了改善在此类场景中的行为，Kotlin 1.3 引入了称为 契约 的实验性机制。

契约 让一个函数能够以编译器理解的方式显式描述其行为。 目前支持两大类场景：

- 通过声明函数调用的结果与所传参数值之间的关系来改进智能转换分析：

```
fun require(condition: Boolean) {
    // 这是一种语法格式，告诉编译器：
    // “如果这个函数成功返回，那么传入的‘condition’为 true”
    contract { returns() implies condition }
    if (!condition) throw IllegalArgumentException(...)
}

fun foo(s: String?) {
    require(s is String)
    // s 在这里智能转换为“String”，因为否则
    // “require”会抛出异常
}
```

- 在存在高阶函数的情况下改进变量初始化的分析：

```
fun synchronize(lock: Any?, block: () -> Unit) {
    // 告诉编译器：
    // “这个函数会在此时此处调用‘block’，并且刚好只调用一次”
    contract { callsInPlace(block, EXACTLY_ONCE) }
}

fun foo() {
    val x: Int
    synchronize(lock) {
        x = 42 // 编译器知道传给“synchronize”的 lambda 表达式刚好
               // 只调了一次，因此不会报重复赋值错
    }
    println(x) // 编译器知道一定会调用该 lambda 表达式而执行
               // 初始化操作，因此可以认为“x”在这里已初始化
}
```

标准库中的契约

`stdlib`（kotlin 标准库）已经利用契约带来了如上所述的对代码分析的改进。 这部分契约是稳定版的，这意味着你现在就可以从改进的代码分析中受益，而无需任何额外的 opt-ins：

```
//sampleStart
fun bar(x: String?) {
    if (!x.isNullOrEmpty()) {
        println("length of '$x' is ${x.length}") // 哇，已经智能转换为非空！
    }
}
//sampleEnd
fun main() {
    bar(null)
    bar("42")
}
```

自定义契约

可以为自己的函数声明契约，不过这个特性是实验性的，因为目前的语法尚处于早期原型状态，并且很可能还会更改。另外还要注意，目前 Kotlin 编译器并不会验证契约，因此程序员有责任编写正确合理的契约。

通过调用标准库（stdlib）函数 `contract` 来引入自定义契约，该函数提供了 DSL 作用域：

```
fun String?.isNullOrEmpty(): Boolean {
    contract {
        returns(false) implies (this@isNullOrEmpty != null)
    }
    return this == null || isEmpty()
}
```

请参见 KEEP 中关于语法与兼容性注意事项的详细信息。

将 when 主语捕获到变量中

在 Kotlin 1.3 中，可以将 `when` 表达式主语捕获到一个变量中：

```
fun Request.getBody() =
        when (val response = executeRequest()) {
            is Success -> response.body
            is HttpError -> throw HttpException(response.status)
        }
```

虽然已经可以在 `when` 表达式前面提取这个变量，但是在 `when` 中的 `val` 使其作用域刚好限制在 `when` 主体中，从而防止命名空间污染。关于 `when` 表达式的完整文档请参见这里。

接口中伴生对象的 @JvmStatic 与 @JvmField

对于 Kotlin 1.3，可以使用注解 `@JvmStatic` 与 `@JvmField` 标记接口的 `companion` 对象成员。 在类文件中会将这些成员提升到相应接口中并标记为 `static`。

例如，以下 Kotlin 代码：

```
interface Foo {
    companion object {
        @JvmField
        val answer: Int = 42

        @JvmStatic
        fun sayHello() {
            println("Hello, world!")
        }
    }
}
```

相当于这段 Java 代码：

```
interface Foo {
    public static int answer = 42;
    public static void sayHello() {
        // ……
    }
}
```

注解类中的内嵌声明

在 Kotlin 1.3 中，注解可以有内嵌的类、接口、对象与伴生对象：

```
annotation class Foo {
    enum class Direction { UP, DOWN, LEFT, RIGHT }

    annotation class Bar

    companion object {
        fun foo(): Int = 42
        val bar: Int = 42
    }
}
```

无参的 main

按照惯例，Kotlin 程序的入口点是一个签名类似于 `main(args: Array<String>)` 的函数， 其中 `args` 表示传给该程序的命令行参数。然而，并非每个应用程序都支持命令行参数， 因此这个参数往往到最后都没有用到。

Kotlin 1.3 引入了一种更简单的无参 `main` 形式。现在 Kotlin 版的 `Hello,World` 缩短了19个字符！

```
fun main() {
    println("Hello, world!")
}
```

更多元的函数

在 Kotlin 中用带有不同数量参数的泛型类来表示函数类型： `Function0<R>` 、 `Function1<P0, R>` 、 `Function2<P0, P1, R>` ……这种方式有一个问题是这个列表是有限的，目前只到 `Function22` 。

Kotlin 1.3 放宽了这一限制，并添加了对具有更多元数（参数个数）的函数的支持：

```
fun trueEnterpriseComesToKotlin(block: (Any, Any, …… /* 42 个 */, Any) -> Any) {
    block(Any(), Any(), ……, Any())
}
```

渐进模式

Kotlin 非常注重代码的稳定性与向后兼容性：Kotlin 兼容性策略提到破坏性变更（例如，会使以前编译正常的代码现在不能通过编译的变更）只能在主版本（**1.2**、**1.3** 等）中引入。

我们相信很多用户可以使用更快的周期，其中关键的编译器修复会即时生效， 从而使代码更安全、更正确。因此 Kotlin 1.3 引入了渐进编译器模式，可以通过将参数 `-progressive` 传给编译器来启用。

在渐进模式下，语言语义中的一些修复可以即时生效。所有这些修复都有以下两个重要特征：

- 保留了源代码与旧版编译器的向后兼容性，这意味着可以通过非渐进式编译器编译所有可由渐进式编译器编译的代码。

- 只是在某种意义上使代码更安全——例如，可以禁用某些不安全的智能转换，可以将所生成代码的行为更改为更可预测、更稳定，等等。

启用渐进模式可能需要重写一些代码，但不会太多——所有在渐进模式启用的修复都已经过精心挑选、通过审核并提供迁移辅助工具。 我们希望对于任何积极维护、即将快速更新到最新语言版本的代码库来说， 渐进模式都是一个不错的选择。

内联类

> Inline classes are in Alpha. They may change incompatibly and require manual migration in the future. We appreciate your feedback on it in YouTrack. See details in the reference.



Kotlin 1.3 引入了一种新的声明方式——`inline class`。内联类可以看作是普通类的受限版，尤其是内联类必须有且只有一个属性：

```
inline class Name(val s: String)
```

Kotlin 编译器会使用此限制来积极优化内联类的运行时表示， 并使用底层属性的值替换内联类的实例，其中可能会移除构造函数调用、GC 压力， 以及启用其他优化：

```
inline class Name(val s: String)
//sampleStart
fun main() {
    // 下一行不会调用构造函数，并且
    // 在运行时，“name”只包含字符串 "Kotlin"
    val name = Name("Kotlin")
    println(name.s)
}
//sampleEnd
```

详见内联类的参考文档。

无符号整型

> Unsigned integers are in Beta. Their implementation is almost stable, but migration steps may be required in the future. We'll do our best to minimize any changes you will have to make.



Kotlin 1.3 引入了无符号整型类型：

- `kotlin.UByte`：无符号 8 比特整数，范围是 0 到 255
- `kotlin.UShort`：无符号 16 比特整数，范围是 0 到 65535
- `kotlin.UInt`：无符号 32 比特整数，范围是 0 到 2^32 - 1
- `kotlin.ULong`：无符号 64 比特整数，范围是 0 到 2^64 - 1

无符号类型也支持其对应有符号类型的大多数操作：

```
fun main() {
//sampleStart
// 可以使用字面值后缀定义无符号类型:
val uint = 42u
val ulong = 42uL
val ubyte: UByte = 255u

// 通过标准库扩展可以将有符号类型转换为无符号类型，反之亦然:
val int = uint.toInt()
val byte = ubyte.toByte()
val ulong2 = byte.toULong()

// 无符号类型支持类似的操作符:
val x = 20u + 22u
val y = 1u shl 8
val z = "128".toUByte()
val range = 1u..5u
//sampleEnd
println("ubyte: $ubyte, byte: $byte, ulong2: $ulong2")
println("x: $x, y: $y, z: $z, range: $range")
}
```

详见其参考文档。

@JvmDefault

> `@JvmDefault` is Experimental. It may be dropped or changed at any time. Use it only for evaluation purposes. We appreciate your feedback on it in YouTrack.
>
> 

Kotlin 兼容很多 Java 版本，其中包括不支持默认方法的 Java 6 与 Java 7。 为了方便起见，Kotlin 编译器可以变通突破这个限制，不过这个变通方法与 Java 8 引入的 `default` 方法并不兼容。

这可能会是 Java 互操作性的一个问题，因此 Kotlin 1.3 引入了 `@JvmDefault` 注解。 以此注解标注的方法会生成为 JVM 平台的 `default` 方法：

```
interface Foo {
```

```
    // 会生成为“default”方法
    @JvmDefault
    fun foo(): Int = 42
}
```

> 警告！以 `@JvmDefault` 注解标注的 API 会对二进制兼容性产生严重影响。 在生产中使用 `@JvmDefault` 之前，请务必仔细阅读其参考文档页。
>
> 

标准库

多平台随机数

在 Kotlin 1.3 之前没有在所有平台生成随机数的统一方式——我们不得不借助平台相关的解决方案，如 JVM 平台的 `java.util.Random` 。这个版本通过引入在所有平台都可用的 `kotlin.random.Random` 类来解决这一问题。

```
import kotlin.random.Random

fun main() {
//sampleStart
    val number = Random.nextInt(42)  // 数字在区间 [0, 上限) 内
    println(number)
//sampleEnd
}
```

isNullOrEmpty 与 orEmpty 扩展

一些类型的 `isNullOrEmpty` 与 `orEmpty` 扩展已经存在于标准库中。如果接收者是 `null` 或空容器，第一个函数返回 `true` ；而如果接收者是 `null` ，第二个函数回退为空容器实例。Kotlin 1.3 为集合、映射以及对象数组提供了类似的扩展。

在两个现有数组间复制元素

为包括无符号整型数组在内的现有数组类型新增的函数 `array.copyInto(targetArray, targetOffset, startIndex, endIndex)` 使在纯 Kotlin 中实现基于数组的容器更容易。

```
fun main() {
//sampleStart
    val sourceArr = arrayOf("k", "o", "t", "l", "i", "n")
    val targetArr = sourceArr.copyInto(arrayOfNulls<String>(6), 3, startIndex = 3, end
```

```
Index = 6)
    println(targetArr.contentToString())

    sourceArr.copyInto(targetArr, startIndex = 0, endIndex = 3)
    println(targetArr.contentToString())
//sampleEnd
}
```

associateWith

一个很常见的情况是，有一个键的列表，希望通过将其中的每个键与某个值相关联来构建映射。 以前可以通过 `associate { it to getValue(it) }` 函数来实现，不过现在我们引入了一种更高效、更易读的替代方式： `keys.associateWith { getValue(it) }` 。

```
fun main() {
//sampleStart
    val keys = 'a'..'f'
    val map = keys.associateWith { it.toString().repeat(5).capitalize() }
    map.forEach { println(it) }
//sampleEnd
}
```

ifEmpty 与 ifBlank 函数

集合、映射、对象数组、字符序列以及序列现在都有一个 `ifEmpty` 函数，它可以指定一个备用值，当接收者为空容器时以该值代替接收者使用：

```
fun main() {
//sampleStart
    fun printAllUppercase(data: List<String>) {
        val result = data
        .filter { it.all { c -> c.isUpperCase() } }
            .ifEmpty { listOf("<no uppercase>") }
        result.forEach { println(it) }
    }

    printAllUppercase(listOf("foo", "Bar"))
    printAllUppercase(listOf("FOO", "BAR"))
//sampleEnd
}
```

此外，字符序列与字符串还有一个 `ifBlank` 扩展，它与 `ifEmpty` 类似，只是会检测字符串是否全部都是空白符而不只是空串。

```
fun main() {
//sampleStart
```

```
    val s = "    \n"
    println(s.ifBlank { "<blank>" })
    println(s.ifBlank { null })
//sampleEnd
}
```

反射中的密封类

我们在 `kotlin-reflect` 中添加了一个新的 API，可以列出密封（`sealed`）类的所有直接子类型，即 `KClass.sealedSubclasses`。

小改动

- `Boolean` 类型现在有伴生对象了。
- `Any?.hashCode()` 扩展函数对 `null` 返回 0。
- `Char` 现在提供了 `MIN_VALUE` 与 `MAX_VALUE` 常量。
- 原生类型伴生对象中的常量 `SIZE_BYTES` 与 `SIZE_BITS`。

工具

IDE 中的代码风格支持

Kotlin 1.3 在 IntelliJ IDEA 中引入了对推荐代码风格的支持。 其迁移指南参见这里。

kotlinx.serialization

kotlinx.serialization 是一个在 Kotlin 中为（反）序列化对象提供多平台支持的库。以前，它是一个独立项目，不过自 Kotlin 1.3 起， 它与其他编译器插件一样随 Kotlin 编译器发行版一起发行。其主要区别在于，无需人为关注序列化 IDE 插件与正在使用的 Kotlin IDE 插件是否兼容了：现在 Kotlin IDE 插件已经包含了序列化支持！

详见这里。

> 尽管 kotlinx.serialization 现在随 Kotlin 编译器发行版一起发行，但 in Kotlin 1.3 它仍是一个实验性的特性。
>
> 

脚本更新

> Scripting is Experimental. It may be dropped or changed at any time. Use it only for evaluation purposes. We appreciate your feedback on it in YouTrack.



Kotlin 1.3 继续发展与改进脚本 API，为脚本定制引入了一些实验性支持， 例如添加外部属性、提供静态或动态依赖等等。

关于其他细节，请参考 KEEP-75。

草稿文件支持

Kotlin 1.3 引入了对可运行的草稿文件（*scratch files*）的支持。草稿文件是一个扩展名为 .kts 的 kotlin 脚本文件， 可以在编辑器中直接运行并获取求值结果。

更多细节请参考通用草稿文件文档。

Kotlin 1.2 的新特性

发布于：*2017-11-28*

目录

- 多平台项目
- 其他语言特性
- 标准库
- JVM 后端
- JavaScript 后端

多平台项目（实验性的）

多平台项目是 Kotlin 1.2 中的一个新的实验性的特性，允许你在支持 Kotlin 的目标平台——JVM、JavaScript 以及（将来的）Native 之间重用代码。在多平台项目中，你有三种模块：

- 一个公共模块包含平台无关代码，以及无实现的依赖平台的 API 声明。
- 平台模块包含通用模块中的平台相关声明在指定平台的实现，以及其他平台相关代码。
- 常规模块面向指定的平台，既可以是平台模块的依赖，也可以依赖平台模块。

当你为指定平台编译多平台项目时，既会生成公共代码也会生成平台相关代码。

多平台项目支持的一个主要特点是可以通过预期声明与实际声明来表达公共代码对平台相关部分的依赖关系。 一个预期声明指定一个 API（类、接口、注解、顶层声明等）。 一个实际声明要么是该 API 的平台相关实现，要么是一个引用到在一个外部库中该 API 的一个既有实现的别名。这是一个示例：

在公共代码中：

```
// 预期平台相关 API:
expect fun hello(world: String): String

fun greet() {
    // 该预期 API 的用法:
    val greeting = hello("multiplatform world")
    println(greeting)
}

expect class URL(spec: String) {
    open fun getHost(): String
```

```
    open fun getPath(): String
}
```

在 JVM 平台代码中：

```
actual fun hello(world: String): String =
    "Hello, $world, on the JVM platform!"

// 使用既有平台相关实现：
actual typealias URL = java.net.URL
```

关于构建多平台项目的详细信息与步骤，请参见多平台程序设计文档。

其他语言特性

注解中的数组字面值

自 Kotlin 1.2 起，注解的数组参数可以通过新的数组字面值语法传入，而无需使用 `arrayOf` 函数：

```
@CacheConfig(cacheNames = ["books", "default"])
public class BookRepositoryImpl {
    // ……
}
```

该数组字面值语法仅限于注解参数。

lateinit 顶层属性与局部变量

`lateinit` 修饰符现在可以用于顶层属性与局部变量了。例如，后者可用于当一个 lambda 表达式作为构造函数参数传给一个对象时，引用另一个必须稍后定义的对象：

```
class Node<T>(val value: T, val next: () -> Node<T>)

fun main(args: Array<String>) {
    // 三个节点的环：
    lateinit var third: Node<Int>

    val second = Node(2, next = { third })
    val first = Node(1, next = { second })

    third = Node(3, next = { first })

    val nodes = generateSequence(first) { it.next() }
```

```
    println("Values in the cycle: ${nodes.take(7).joinToString { it.value.toString() }
}, ...")
}
```

检测 lateinit 变量是否已初始化

现在可以通过属性引用的 `isInitialized` 来检测该 lateinit var 是否已初始化：

```
class Foo {
    lateinit var lateinitVar: String

    fun initializationLogic() {
//sampleStart
        println("isInitialized before assignment: " + this::lateinitVar.isInitialized)
        lateinitVar = "value"
        println("isInitialized after assignment: " + this::lateinitVar.isInitialized)
//sampleEnd
    }
}

fun main(args: Array<String>) {
    Foo().initializationLogic()
}
```

内联函数带有默认函数式参数

内联函数现在允许其内联函式数参数具有默认值：

```
//sampleStart
inline fun <E> Iterable<E>.strings(transform: (E) -> String = { it.toString() }) =
    map { transform(it) }

val defaultStrings = listOf(1, 2, 3).strings()
val customStrings = listOf(1, 2, 3).strings { "($it)" }
//sampleEnd

fun main(args: Array<String>) {
    println("defaultStrings = $defaultStrings")
    println("customStrings = $customStrings")
}
```

源自显式类型转换的信息会用于类型推断

Kotlin 编译器现在可将类型转换信息用于类型推断。如果你调用一个返回类型参数 `T` 的泛型方法并将返回值转换为指定类型 `Foo`，那么编译器现在知道对于本次调用需要绑定类型为 `Foo`。

这对于 Android 开发者来说尤为重要，因为编译器现在可以正确分析 Android API 级别 26中的泛型 `findViewById` 调用：

```
val button = findViewById(R.id.button) as Button
```

智能类型转换改进

当一个变量有安全调用表达式与空检测赋值时，其智能转换现在也可以应用于安全调用接收者：

```
fun countFirst(s: Any): Int {
//sampleStart
    val firstChar = (s as? CharSequence)?.firstOrNull()
    if (firstChar != null)
    return s.count { it == firstChar } // s: Any 会智能转换为 CharSequence

    val firstItem = (s as? Iterable<*>)?.firstOrNull()
    if (firstItem != null)
    return s.count { it == firstItem } // s: Any 会智能转换为 Iterable<*>
//sampleEnd
    return -1
}

fun main(args: Array<String>) {
  val string = "abacaba"
  val countInString = countFirst(string)
  println("called on \"$string\": $countInString")

  val list = listOf(1, 2, 3, 1, 2)
  val countInList = countFirst(list)
  println("called on $list: $countInList")
}
```

智能转换现在也允许用于在 lambda 表达式中局部变量，只要这些局部变量仅在 lambda 表达式之前修改即可：

```
fun main(args: Array<String>) {
//sampleStart
    val flag = args.size == 0
    var x: String? = null
    if (flag) x = "Yahoo!"

    run {
        if (x != null) {
            println(x.length) // x 会智能转换为 String
        }
    }
//sampleEnd
```

```
}
```

支持 ::foo 作为 this::foo 的简写

现在写绑定到 `this` 成员的可调用引用可以无需显式接收者，即 `::foo` 取代 `this::foo`。这也使在引用外部接收者的成员的 lambda 表达式中使用可调用引用更加方便。

破坏性变更：try 块后可靠智能转换

Kotlin 以前将 `try` 块中的赋值语句用于块后的智能转换，这可能会破坏类型安全与空安全并引发运行时故障。这个版本修复了该问题，使智能转换更加严格，但可能会破坏一些依靠这种智能转换的代码。

如果要切换到旧版智能转换行为，请传入回退标志 `-Xlegacy-smart-cast-after-try` 作为编译器参数。该参数会在 Kotlin 1.3中弃用。

弃用：数据类弃用 copy

当从已具有签名相同的 `copy` 函数的类型派生数据类时，为数据类生成的 `copy` 实现使用超类型的默认值，这导致反直觉行为， 或者导致运行时失败，如果超类型中没有默认实参的话。

导致 `copy` 冲突的继承在 Kotlin 1.2 中已弃用并带有警告， 而在 Kotlin 1.3中将会是错误。

弃用：枚举条目中的嵌套类型

由于初始化逻辑的问题，已弃用在枚举条目内部定义一个非 `inner class` 的嵌套类。这在 Kotlin 1.2 中会引起警告，而在 Kotlin 1.3中会成为错误。

弃用：vararg 单个具名实参

为了与注解中的数组字面值保持一致，向一个具名实参形式的 vararg 参数传入单个项目的用法（ `foo(items = i)` ）已被弃用。请使用伸展操作符连同相应的数组工厂函数：

```
foo(items = *arrayOf(1))
```

在这种情况下有一项防止性能下降的优化可以消除冗余的数组创建。 单参数形式在 Kotlin 1.2 中会产生警告，而在 Kotlin 1.3中会放弃。

弃用：扩展 Throwable 的泛型类的内部类

继承自 `Throwable` 的泛型类的内部类可能会在 throw-catch 场景中违反类型安全性，因此已弃用，在 Kotlin 1.2 中会是警告，而在 Kotlin 1.3中会是错误。

弃用：修改只读属性的幕后字段

通过在自定义 getter 中赋值 `field = ……` 来修改只读属性的幕后字段的用法已被弃用， 在 Kotlin 1.2 中会是警告，而在 Kotlin 1.3中会是错误。

标准库

Kotlin 标准库构件与拆分包

Kotlin 标准库现在完全兼容 Java 9 的模块系统，它禁止拆分包（多个 jar 文件声明的类在同一包中）。为了支持这点，我们引入了新的 `kotlin-stdlib-jdk7` 与 `kotlin-stdlib-jdk8` ，它们取代了旧版的 `kotlin-stdlib-jre7` 与 `kotlin-stdlib-jre8` 。

在 Kotlin 看来新的构件中的声明在相同的包名内，而在 Java看来有不同的包名。因此，切换到新的构件无需修改任何源代码。

确保与新的模块系统兼容的另一处变更是在 `kotlin-reflect` 库中删除了 `kotlin.reflect` 包中弃用的声明。如果你正在使用它们，你需要切换到使用 `kotlin.reflect.full` 包中的声明，自 Kotlin 1.1 起就支持这个包了。

windowed、chunked、zipWithNext

用于 `Iterable<T>` 、 `Sequence<T>` 与 `CharSequence` 的新的扩展覆盖了这些应用场景：缓存或批处理（ `chunked` ）、 滑动窗口与计算滑动均值（ `windowed` ）以及处理成对的后续条目（ `zipWithNext` ）：

```
fun main(args: Array<String>) {
//sampleStart
    val items = (1..9).map { it * it }

    val chunkedIntoLists = items.chunked(4)
    val points3d = items.chunked(3) { (x, y, z) -> Triple(x, y, z) }
    val windowed = items.windowed(4)
    val slidingAverage = items.windowed(4) { it.average() }
    val pairwiseDifferences = items.zipWithNext { a, b -> b - a }
//sampleEnd

    println("items: $items\n")

    println("chunked into lists: $chunkedIntoLists")
    println("3D points: $points3d")
```

```
    println("windowed by 4: $windowed")
    println("sliding average by 4: $slidingAverage")
    println("pairwise differences: $pairwiseDifferences")
}
```

fill、replaceAll、shuffle/shuffled

添加了一些用于操作列表的扩展函数：`MutableList` 的 `fill`、`replaceAll` 与 `shuffle`，以及只读 `List` 的 `shuffled`：

```
fun main(args: Array<String>) {
//sampleStart
    val items = (1..5).toMutableList()

    items.shuffle()
    println("Shuffled items: $items")

    items.replaceAll { it * 2 }
    println("Items doubled: $items")

    items.fill(5)
    println("Items filled with 5: $items")
//sampleEnd
}
```

kotlin-stdlib 中的数学运算

为满足由来已久的需求，Kotlin 1.2 添加了 JVM 与 JS 公用的用于数学运算的 `kotlin.math` API，包含以下内容：

- 常量：`PI` 与 `E`
- 三角函数：`cos`、`sin`、`tan` 及其反函数：`acos`、`asin`、`atan`、`atan2`
- 双曲函数：`cosh`、`sinh`、`tanh` 及其反函数：`acosh`、`asinh`、`atanh`
- 指数函数：`pow`（扩展函数）、`sqrt`、`hypot`、`exp`、`expm1`
- 对数函数：`log`、`log2`、`log10`、`ln`、`ln1p`
- 取整：
  - `ceil`、`floor`、`truncate`、`round`（奇进偶舍）函数
  - `roundToInt`、`roundToLong`（四舍五入）扩展函数
- 符号与绝对值：
  - `abs` 与 `sign` 函数
  - `absoluteValue` 与 `sign` 扩展属性
  - `withSign` 扩展函数
- 两个数的最值函数：`max` 与 `min`
- 二进制表示：

- `ulp` 扩展属性
- `nextUp`、`nextDown`、`nextTowards` 扩展函数
- `toBits`、`toRawBits`、`Double.fromBits`（这些在 `kotlin` 包中）

这些函数同样也有 `Float` 参数版本（但不包括常量）。

用于 BigInteger 与 BigDecimal 的操作符与转换

Kotlin 1.2 引入了一些使用 `BigInteger` 与 `BigDecimal` 运算以及由其他数字类型创建它们的函数。具体如下：

- `toBigInteger` 用于 `Int` 与 `Long`
- `toBigDecimal` 用于 `Int`、`Long`、`Float`、`Double` 以及 `BigInteger`
- 算术与位运算操作符函数：
  - 二元操作符 `+`、`-`、`*`、`/`、`%` 以及中缀函数 `and`、`or`、`xor`、`shl`、`shr`
  - 一元操作符 `-`、`++`、`--` 以及函数 `inv`

浮点数到比特的转换

添加了用于将 `Double` 及 `Float` 与其比特表示形式相互转换的函数：

- `toBits` 与 `toRawBits` 对于 `Double` 返回 `Long` 而对于 `Float` 返回 `Int`
- `Double.fromBits` 与 `Float.fromBits` 用于有相应比特表示形式创建浮点数

正则表达式现在可序列化

`kotlin.text.Regex` 类现在已经是 `Serializable` 的了并且可用在可序列化的继承结构中。

如果可用，Closeable.use 会调用 Throwable.addSuppressed

当在其他异常之后关闭资源期间抛出一个异常，`Closeable.use` 函数会调用 `Throwable.addSuppressed`。

要启用这个行为，需要依赖项中有 `kotlin-stdlib-jdk7`。

JVM 后端

构造函数调用规范化

自 1.0 版起，Kotlin 就已支持带有复杂控制流的表达式，诸如 try-catch 表达式以及内联函数。根据 Java 虚拟机规范这样的代码是有效的。不幸的是， 当这样的表达式出现在构造函数调用的参数中时，一些字节码处理工具不能很好地处理这种代码 。

为了缓解这种字节码处理工具用户的这一问题，我们添加了一个命令行编译器选项（ `-Xnormalize-constructor-calls=模式` ），告诉编译器为这样的构造过程生成更接近 Java 的字节码 。其中 `模式` 是下列之一：

- `disable` （默认）——以与 Kotlin 1.0 即 1.1 相同的方式生成字节码。
- `enable` ——为构造函数调用生成类似 Java 的字节码。 这可能会改变类加载与初始化的顺序 。
- `preserve-class-initialization` ——为构造函数调用生成类似 Java 的字节码，并确保类初始化顺序得到保留。这可能会影响应用程序的整体性能；仅用在多个类之间共享一些复杂状态并在类初始化时更新的场景中。

“人工”解决办法是将具有控制流的子表达式的值存储在变量中，而不是直接在调用参数内对其求值。这与 `-Xnormalize-constructor-calls=enable` 类似。

Java 默认方法调用

在 Kotlin 1.2 之前，面向 JVM 1.6 的接口成员覆盖 Java 默认方法会产生一个关于超类型调用的警告： `Super calls to Java default methods are deprecated in JVM target 1.6. Recompile with '-jvm-target 1.8'` （“面向 JVM 1.6 的 Java 默认方法的超类型调用已弃用，请使用‘-jvm-target 1.8’重新编译”）。 在 Kotlin 1.2 中，这是一个错误，因此这样的代码都需要面向 JVM 1.8 编译。

破坏性变更：平台类型 x.equals(null) 的一致行为

在映射到 Java 原生类型（ `Int!` 、 `Boolean!` 、 `Short!` 、 `Long!` 、 `Float!` 、 `Double!` 、 `Char!` ）的平台类型上调用 `x.equals(null)` ，当 `x` 为 `null` 时错误地返回了 `true` 。 自 Kotlin 1.2 起，在平台类型的空值上调用 `x.equals(……)` 都会抛出 **NPE**（但 `x == ...` 不会）。

要返回到 1.2 之前的行为，请将标志 `-Xno-exception-on-explicit-equals-for-boxed-null` 传给编译器。

破坏性变更：修正平台 null 透过内联扩展接收者逃逸

在平台类型的空值上调用内联扩展函数并没有检测接收者是否为 null， 因而允许 null 逃逸到其他代码中。Kotlin 1.2 在调用处强制执行这项检测， 如果接收者为空就抛出异常。

要切换到旧版行为，请将回退标志 `-Xno-receiver-assertions` 传给编译器。

JavaScript 后端

默认启用 TypedArray 支持

将 Kotlin 原生数组（如 `IntArray`、`DoubleArray` 等）翻译为 JavaScript 有类型数组的 JS 有类型数组支持之前是选择性加入的功能，现在已默认启用。

工具

警告作为错误

编译器现在提供一个将所有警告视为错误的选项。可在命令行中使用 `-Werror`，或者在 Gradle 中使用以下代码片段：

```
compileKotlin {
  kotlinOptions.allWarningsAsErrors = true
}
```

Kotlin 1.1 的新特性

发布于：*2016-02-15*

目录

- 协程
- 其他语言特性
- 标准库
- JVM 后端
- JavaScript 后端

JavaScript

从 Kotlin 1.1 开始，JavaScript 目标平台不再当是实验性的。所有语言功能都支持， 并且有许多新的工具用于与前端开发环境集成。更详细改动列表，请参见下文 。

协程（实验性的）

Kotlin 1.1 的关键新特性是协程，它带来了 `future` / `await` 、 `yield` 以及类似的编程模式的支持。Kotlin 的设计中的关键特性是协程执行的实现是语言库的一部分， 而不是语言的一部分，所以你不必绑定任何特定的编程范式或并发库。

协程实际上是一个轻量级的线程，可以挂起并稍后恢复。 协程通过挂起函数支持：对这样的函数的调用可能会挂起协程，并启动一个新的协程，我们通常使用匿名挂起函数（即挂起 lambda 表达式）。

我们来看看在外部库 kotlinx.coroutines 中实现的 `async` / `await` ：

```
// 在后台线程池中运行该代码
fun asyncOverlay() = async(CommonPool) {
    // 启动两个异步操作
    val original = asyncLoadImage("original")
    val overlay = asyncLoadImage("overlay")
    // 然后应用叠加到两个结果
    applyOverlay(original.await(), overlay.await())
}

// 在 UI 上下文中启动新的协程
```

```
launch(UI) {
// 等待异步叠加完成
    val image = asyncOverlay().await()
// 然后在 UI 中显示
    showImage(image)
}
```

这里，`async { …… }` 启动一个协程，当我们使用 `await()` 时，挂起协程的执行，而执行正在等待的操作，并且在等待的操作完成时恢复（可能在不同的线程上）。

标准库通过 `yield` 与 `yieldAll` 函数使用协程来支持惰性生成序列。 在这样的序列中，在取回每个元素之后挂起返回序列元素的代码块， 并在请求下一个元素时恢复。这里有一个例子：

```
import kotlin.coroutines.experimental.*

fun main(args: Array<String>) {
    val seq = buildSequence {
      for (i in 1..5) {
          // 产生一个 i 的平方
          yield(i * i)
      }
      // 产生一个区间
      yieldAll(26..28)
    }

    // 输出该序列
    println(seq.toList())
}
```

运行上面的代码以查看结果。随意编辑它并再次运行！

更多信息请参见协程文档及教程。

请注意，协程目前还是一个实验性的特性，这意味着 Kotlin 团队不承诺在最终的 1.1 版本时保持该功能的向后兼容性。

其他语言特性

类型别名

类型别名允许你为现有类型定义备用名称。 这对于泛型类型（如集合）以及函数类型最有用。 这里有几个例子：

```
//sampleStart
typealias OscarWinners = Map<String, String>
```

```
fun countLaLaLand(oscarWinners: OscarWinners) =
        oscarWinners.count { it.value.contains("La La Land") }

// 请注意，类型名称（初始名与类型别名）是可互换的：
fun checkLaLaLandIsTheBestMovie(oscarWinners: Map<String, String>) =
        oscarWinners["Best picture"] == "La La Land"
//sampleEnd

fun oscarWinners(): OscarWinners {
    return mapOf(
            "Best song" to "City of Stars (La La Land)",
            "Best actress" to "Emma Stone (La La Land)",
            "Best picture" to "Moonlight" /* …… */)
}

fun main(args: Array<String>) {
    val oscarWinners = oscarWinners()

    val laLaLandAwards = countLaLaLand(oscarWinners)
    println("LaLaLandAwards = $laLaLandAwards (in our small example), but actually it'
s 6.")

    val laLaLandIsTheBestMovie = checkLaLaLandIsTheBestMovie(oscarWinners)
    println("LaLaLandIsTheBestMovie = $laLaLandIsTheBestMovie")
}
```

更详细信息请参见类型别名文档与 KEEP。

已绑定的可调用引用

现在可以使用 `::` 操作符来获取指向特定对象实例的方法或属性的成员引用。 以前这只能用 lambda 表达式表示。 这里有一个例子：

```
//sampleStart
val numberRegex = "\\d+".toRegex()
val numbers = listOf("abc", "123", "456").filter(numberRegex::matches)
//sampleEnd

fun main(args: Array<String>) {
    println("Result is $numbers")
}
```

更详细信息请参阅其文档与 KEEP。

密封类与数据类

Kotlin 1.1 删除了一些对 Kotlin 1.0 中已存在的密封类与数据类的限制。 现在你可以在同一个文件中的任何地方定义一个密封类的子类，而不只是以作为密封类嵌套类的方式。 数据类现在可以扩展其他类。 这可以用来友好且清晰地定义一个表达式类的层次结构：

```
//sampleStart
sealed class Expr

data class Const(val number: Double) : Expr()
data class Sum(val e1: Expr, val e2: Expr) : Expr()
object NotANumber : Expr()

fun eval(expr: Expr): Double = when (expr) {
    is Const -> expr.number
    is Sum -> eval(expr.e1) + eval(expr.e2)
    NotANumber -> Double.NaN
}
val e = eval(Sum(Const(1.0), Const(2.0)))
//sampleEnd

fun main(args: Array<String>) {
    println("e is $e") // 3.0
}
```

更详细信息请参阅密封类文档或者密封类及数据类的 KEEP。

lambda 表达式中的解构

现在可以使用解构声明语法来解开传递给 lambda 表达式的参数。 这里有一个示例：

```
fun main(args: Array<String>) {
//sampleStart
    val map = mapOf(1 to "one", 2 to "two")
    // 之前
    println(map.mapValues { entry ->
      val (key, value) = entry
      "$key -> $value!"
    })
    // 现在
    println(map.mapValues { (key, value) -> "$key -> $value!" })
//sampleEnd
}
```

更详细信息请参阅解构声明文档与 KEEP。

下划线用于未使用的参数

对于具有多个参数的 lambda 表达式，可以使用 `_` 字符替换不使用的参数的名称：

```
fun main(args: Array<String>) {
    val map = mapOf(1 to "one", 2 to "two")

//sampleStart
    map.forEach { _, value -> println("$value!") }
//sampleEnd
}
```

这也适用于解构声明：

```
data class Result(val value: Any, val status: String)

fun getResult() = Result(42, "ok").also { println("getResult() returns $it") }

fun main(args: Array<String>) {
//sampleStart
    val (_, status) = getResult()
//sampleEnd
    println("status is '$status'")
}
```

更详细信息请参阅其 KEEP。

数字字面值中的下划线

正如在 Java 8 中一样，Kotlin 现在允许在数字字面值中使用下划线来分隔数字分组：

```
//sampleStart
val oneMillion = 1_000_000
val hexBytes = 0xFF_EC_DE_5E
val bytes = 0b11010010_01101001_10010100_10010010
//sampleEnd

fun main(args: Array<String>) {
    println(oneMillion)
    println(hexBytes.toString(16))
    println(bytes.toString(2))
}
```

更详细信息请参阅其 KEEP。

对于属性的更短语法

对于没有自定义访问器、或者将 getter 定义为表达式主体的属性，现在可以省略属性的类型：

```
//sampleStart
```

```
    data class Person(val name: String, val age: Int) {
    val isAdult get() = age >= 20 // 属性类型推断为 “Boolean”
}
//sampleEnd
fun main(args: Array<String>) {
    val akari = Person("Akari", 26)
    println("$akari.isAdult = ${akari.isAdult}")
}
```

内联属性访问器

如果属性没有幕后字段，现在可以使用 `inline` 修饰符来标记该属性访问器。 这些访问器的编译方式与内联函数相同。

```
//sampleStart
public val <T> List<T>.lastIndex: Int
    inline get() = this.size - 1
//sampleEnd

fun main(args: Array<String>) {
    val list = listOf('a', 'b')
    // 其 getter 会内联
    println("Last index of $list is ${list.lastIndex}")
}
```

你也可以将整个属性标记为 `inline` ——这样修饰符应用于两个访问器。

更详细信息请参阅内联函数文档与 KEEP。

局部委托属性

现在可以对局部变量使用委托属性语法。 一个可能的用途是定义一个延迟求值的局部变量：

```
import java.util.Random

fun needAnswer() = Random().nextBoolean()

fun main(args: Array<String>) {
//sampleStart
    val answer by lazy {
        println("Calculating the answer...")
        42
    }
    if (needAnswer()) {                     // 返回随机值
        println("The answer is $answer.")   // 此时计算出答案
    }
    else {
        println("Sometimes no answer is the answer...")
```

```
    }
//sampleEnd
}
```

更详细信息请参阅其 KEEP。

委托属性绑定的拦截

对于委托属性，现在可以使用 `provideDelegate` 操作符拦截委托到属性之间的绑定。 例如，如果我们想要在绑定之前检测属性名称，我们可以这样写：

```
class ResourceLoader<T>(id: ResourceID<T>) {
    operator fun provideDelegate(thisRef: MyUI, property: KProperty<*>): ReadOnlyPrope
rty<MyUI, T> {
        checkProperty(thisRef, property.name)
        …… // 属性创建
    }

    private fun checkProperty(thisRef: MyUI, name: String) { …… }
}

fun <T> bindResource(id: ResourceID<T>): ResourceLoader<T> { …… }

class MyUI {
    val image by bindResource(ResourceID.image_id)
    val text by bindResource(ResourceID.text_id)
}
```

`provideDelegate` 方法在创建 `MyUI` 实例期间将会为每个属性调用，并且可以立即执行必要的验证。

更详细信息请参阅属性委托文档。

泛型枚举值访问

现在可以用泛型的方式来对枚举类的值进行枚举：

```
//sampleStart
enum class RGB { RED, GREEN, BLUE }

inline fun <reified T : Enum<T>> printAllValues() {
    print(enumValues<T>().joinToString { it.name })
}
//sampleEnd

fun main(args: Array<String>) {
    printAllValues<RGB>() // 输出 RED, GREEN, BLUE
```

```
}
```

对于 DSL 中隐式接收者的作用域控制

`@DslMarker` 注解允许限制来自 DSL 上下文中的外部作用域的接收者的使用。 考虑那个典型的 HTML 构建器示例：

```
table {
    tr {
        td { + "Text" }
    }
}
```

在 Kotlin 1.0 中，传递给 `td` 的 lambda 表达式中的代码可以访问三个隐式接收者：传递给 `table`、 `tr` 与 `td` 的。 这允许你调用在上下文中没有意义的方法——例如在 `td` 里面调用 `tr`，从而在 `<td>` 中放置一个 `<tr>` 标签。

在 Kotlin 1.1 中，你可以限制这种情况，以使只有在 `td` 的隐式接收者上定义的方法会在传给 `td` 的 lambda 表达式中可用。你可以通过定义标记有 `@DslMarker` 元注解的注解并将其应用于标记类的基类。

更详细信息请参阅类型安全的构建器文档与 KEEP。

rem 操作符

`mod` 操作符现已弃用，而使用 `rem` 取代。动机参见这个问题。

标准库

字符串到数字的转换

在 String 类中有一些新的扩展，用来将它转换为数字，而不会在无效数字上抛出异常：`String.toIntOrNull(): Int?` 、 `String.toDoubleOrNull(): Double?` 等。

```
val port = System.getenv("PORT")?.toIntOrNull() ?: 80
```

还有整数转换函数，如 `Int.toString()` 、 `String.toInt()` 、 `String.toIntOrNull()` ， 每个都有一个带有 `radix` 参数的重载，它允许指定转换的基数（2 到 36）。

onEach()

`onEach` 是一个小、但对于集合与序列很有用的扩展函数，它允许对操作链中的集合/序列的每个元素执行一些操作，可能带有副作用。 对于迭代其行为像 `forEach` 但是也进一步返回可迭代实例。 对于序列它返回一个包装序列，它在元素迭代时延迟应用给定的动作。

```
inputDir.walk()
        .filter { it.isFile && it.name.endsWith(".txt") }
        .onEach { println("Moving $it to $outputDir") }
        .forEach { moveFile(it, File(outputDir, it.toRelativeString(inputDir))) }
```

also()、takeIf() 与 takeUnless()

这些是适用于任何接收者的三个通用扩展函数。

`also` 就像 `apply`：它接受接收者、做一些动作、并返回该接收者。 二者区别是在 `apply` 内部的代码块中接收者是 `this`， 而在 `also` 内部的代码块中是 `it`（并且如果你想的话，你可以给它另一个名字）。 当你不想掩盖来自外部作用域的 `this` 时这很方便：

```
class Block {
    lateinit var content: String
}

//sampleStart
fun Block.copy() = Block().also {
    it.content = this.content
}
//sampleEnd

// 使用“apply”代替
fun Block.copy1() = Block().apply {
    this.content = this@copy1.content
}

fun main(args: Array<String>) {
    val block = Block().apply { content = "content" }
    val copy = block.copy()
    println("Testing the content was copied:")
    println(block.content == copy.content)
}
```

`takeIf` 就像单个值的 `filter`。它检测接收者是否满足该谓词，并在满足时返回该接收者否则不满足时返回 `null`。 结合 elvis 操作符（?:）与及早返回，可以编写如下结构：

```
val outDirFile = File(outputDir.path).takeIf { it.exists() } ?: return false
// 对现有的 outDirFile 做些事情
```

```
fun main(args: Array<String>) {
```

```
    val input = "Kotlin"
    val keyword = "in"

//sampleStart
    val index = input.indexOf(keyword).takeIf { it >= 0 } ?: error("keyword not found"
)
    // 对输入字符串中的关键字索引做些事情，鉴于它已找到
//sampleEnd

    println("'$keyword' was found in '$input'")
    println(input)
    println(" ".repeat(index) + "^")
}
```

`takeUnless` 与 `takeIf` 相同，只是它采用了反向谓词。当它 不 满足谓词时返回接收者，否则返回 `null` 。因此，上面的示例之一可以用 `takeUnless` 重写如下：

```
val index = input.indexOf(keyword).takeUnless { it < 0 } ?: error("keyword not found")
```

当你有一个可调用的引用而不是 lambda 时，使用也很方便：

```
private fun testTakeUnless(string: String) {
//sampleStart
    val result = string.takeUnless(String::isEmpty)
//sampleEnd

    println("string = \"$string\"; result = \"$result\"")
}

fun main(args: Array<String>) {
    testTakeUnless("")
    testTakeUnless("abc")
}
```

groupingBy()

此 API 可以用于按照键对集合进行分组，并同时折叠每个组。 例如，它可以用于计算文本中字符的频率：

```
fun main(args: Array<String>) {
    val words = "one two three four five six seven eight nine ten".split(' ')
//sampleStart
    val frequencies = words.groupingBy { it.first() }.eachCount()
//sampleEnd
    println("Counting first letters: $frequencies.")

    // 另一种方式是使用“groupBy”与“mapValues”创建一个中间的映射，
```

```
    // 而“groupingBy”的方式会即时计数。
    val groupBy = words.groupBy { it.first() }.mapValues { (_, list) -> list.size }
    println("Comparing the result with using 'groupBy': ${groupBy == frequencies}.")
}
```

Map.toMap() 与 Map.toMutableMap()

这俩函数可以用来简易复制映射：

```
class ImmutablePropertyBag(map: Map<String, Any>) {
    private val mapCopy = map.toMap()
}
```

Map.minus(key)

运算符 `plus` 提供了一种将键值对添加到只读映射中以生成新映射的方法，但是没有一种简单的方法来做相反的操作：从映射中删除一个键采用不那么直接的方式如 `Map.filter()` 或 `Map.filterKeys()` 。 现在运算符 `minus` 填补了这个空白。有 4 个可用的重载：用于删除单个键、键的集合、 键的序列与键的数组。

```
fun main(args: Array<String>) {
//sampleStart
    val map = mapOf("key" to 42)
    val emptyMap = map - "key"
//sampleEnd

    println("map: $map")
    println("emptyMap: $emptyMap")
}
```

minOf() 与 maxOf()

这些函数可用于查找两个或三个给定值中的最小与最大值，其中值是原生数字或 `Comparable` 对象。每个函数还有一个重载，它接受一个额外的 `Comparator` 实例， 如果你想比较自身不可比的对象的话。

```
fun main(args: Array<String>) {
//sampleStart
    val list1 = listOf("a", "b")
    val list2 = listOf("x", "y", "z")
    val minSize = minOf(list1.size, list2.size)
    val longestList = maxOf(list1, list2, compareBy { it.size })
//sampleEnd

    println("minSize = $minSize")
```

```
    println("longestList = $longestList")
}
```

类似数组的列表实例化函数

类似于 `Array` 构造函数，现在有创建 `List` 与 `MutableList` 实例的函数，并通过调用 lambda 表达式来初始化每个元素：

```
fun main(args: Array<String>) {
//sampleStart
    val squares = List(10) { index -> index * index }
    val mutable = MutableList(10) { 0 }
//sampleEnd

    println("squares: $squares")
    println("mutable: $mutable")
}
```

Map.getValue()

`Map` 上的这个扩展函数返回一个与给定键相对应的现有值，或者抛出一个异常，提示找不到该键。 如果该映射是用 `withDefault` 生成的，这个函数将返回默认值，而不是抛异常。

```
fun main(args: Array<String>) {
//sampleStart
    val map = mapOf("key" to 42)
    // 返回不可空 Int 值 42
    val value: Int = map.getValue("key")

    val mapWithDefault = map.withDefault { k -> k.length }
    // 返回 4
    val value2 = mapWithDefault.getValue("key2")

    // map.getValue("anotherKey") // <- 这将抛出 NoSuchElementException
//sampleEnd

    println("value is $value")
    println("value2 is $value2")
}
```

抽象集合

这些抽象类可以在实现 Kotlin 集合类时用作基类。 对于实现只读集合，有 `AbstractCollection`、 `AbstractList`、 `AbstractSet` 与 `AbstractMap`， 而对于可变集合，有 `AbstractMutableCollection`、 `AbstractMutableList`、 `AbstractMutableSet` 与

`AbstractMutableMap`。 在 JVM 上，这些抽象可变集合从 JDK 的抽象集合继承了大部分的功能。

数组处理函数

标准库现在提供了一组用于逐个元素操作数组的函数：比较（ `contentEquals` 与 `contentDeepEquals` ），哈希码计算（ `contentHashCode` 与 `contentDeepHashCode` ）， 以及转换成一个字符串（ `contentToString` 与 `contentDeepToString` ）。它们都支持 JVM （它们作为 `java.util.Arrays` 中的相应函数的别名）与 JS（在 Kotlin 标准库中提供实现）。

```
fun main(args: Array<String>) {
//sampleStart
    val array = arrayOf("a", "b", "c")
    println(array.toString())  // JVM 实现：类型及哈希乱码
    println(array.contentToString())  // 良好格式化为列表
//sampleEnd
}
```

JVM 后端

Java 8 字节码支持

Kotlin 现在可以选择生成 Java 8 字节码（命令行选项 `-jvm-target 1.8` 或者 Ant/Maven/Gradle 中的相应选项）。目前这并不改变字节码的语义（特别是，接口与 lambda 表达式中的默认方法的生成与 Kotlin 1.0 中完全一样），但我们计划在以后进一步使用它。

Java 8 标准库支持

现在有支持在 Java 7 与 8 中新添加的 JDK API 的标准库的独立版本。 如果你需要访问新的 API，请使用 `kotlin-stdlib-jre7` 与 `kotlin-stdlib-jre8` maven 构件，而不是标准的 `kotlin-stdlib`。 这些构件是在 `kotlin-stdlib` 之上的微小扩展，它们将它作为传递依赖项带到项目中。

字节码中的参数名

Kotlin 现在支持在字节码中存储参数名。这可以使用命令行选项 `-java-parameters` 启用。

常量内联

编译器现在将 `const val` 属性的值内联到使用它们的位置。

可变闭包变量

用于在 lambda 表达式中捕获可变闭包变量的装箱类不再具有 volatile 字段。 此更改提高了性能，但在一些罕见的使用情况下可能导致新的竞争条件。如果受此影响，你需要提供自己的同步机制来访问变量。

javax.scripting 支持

Kotlin 现在与javax.script API（JSR-223）集成。 其 API 允许在运行时求值代码段：

```
val engine = ScriptEngineManager().getEngineByExtension("kts")!!
engine.eval("val x = 3")
println(engine.eval("x + 2"))  // 输出 5
```

关于使用 API 的示例项目参见这里 。

kotlin.reflect.full

为 Java 9 支持准备，在 `kotlin-reflect.jar` 库中的扩展函数与属性已移动到 `kotlin.reflect.full` 包中。旧包（ `kotlin.reflect` ）中的名称已弃用，将在 Kotlin 1.2 中删除。请注意，核心反射接口（如 `KClass` ）是 Kotlin 标准库（而不是 `kotlin-reflect` ）的一部分，不受移动影响。

JavaScript 后端

统一的标准库

Kotlin 标准库的大部分目前可以从代码编译成 JavaScript 来使用。 特别是，关键类如集合（ `ArrayList` 、 `HashMap` 等）、异常（ `IllegalArgumentException` 等）以及其他几个关键类（ `StringBuilder` 、 `Comparator` ）现在都定义在 `kotlin` 包下。在 JVM 平台上，一些名称是相应 JDK 类的类型别名，而在 JS 平台上，这些类在 Kotlin 标准库中实现。

更好的代码生成

JavaScript 后端现在生成更加可静态检测的代码，这对 JS 代码处理工具（如 minifiers、optimisers、 linters 等）更加友好。

external 修饰符

如果你需要以类型安全的方式在 Kotlin 中访问 JavaScript 实现的类， 你可以使用 `external` 修饰符写一个 Kotlin 声明。（在 Kotlin 1.0 中，使用了 `@native` 注解。）与 JVM 目标平台不同，JS 平台允许对类与属性使用 external 修饰符。 例如，可以按以下方式声明 DOM `Node` 类：

```
external class Node {
    val firstChild: Node

    fun appendChild(child: Node): Node

    fun removeChild(child: Node): Node

    // 等等
}
```

改进的导入处理

现在可以更精确地描述应该从 JavaScript 模块导入的声明。 如果在外部声明上添加 `@JsModule("<模块名>")` 注解，它会在编译期间正确导入到模块系统（CommonJS或AMD）。例如，使用 CommonJS，该声明会通过 `require(……)` 函数导入。 此外，如果要将声明作为模块或全局 JavaScript 对象导入， 可以使用 `@JsNonModule` 注解。

例如，以下是将 JQuery 导入 Kotlin 模块的方法：

```
external interface JQuery {
    fun toggle(duration: Int = definedExternally): JQuery
    fun click(handler: (Event) -> Unit): JQuery
}

@JsModule("jquery")
@JsNonModule
@JsName("$")
external fun jquery(selector: String): JQuery
```

在这种情况下，JQuery 将作为名为 `jquery` 的模块导入。或者，它可以用作 $-对象， 这取决于Kotlin编译器配置使用哪个模块系统。

你可以在应用程序中使用如下所示的这些声明：

```
fun main(args: Array<String>) {
    jquery(".toggle-button").click {
        jquery(".toggle-panel").toggle(300)
    }
}
```

版本发布与路线图

- Kotlin 路线图
- Kotlin 语言特性与提案
- Kotlin 演进原则
- Kotlin 各组件的稳定性
- Kotlin 版本发布

Kotlin 路线图

最后修改时间	2025 年 2 月
下次更新时间	**2025 年 8 月**

欢迎来看 Kotlin 路线图！一窥 JetBrains 团队的工作重点。

关键优先事项

这个路线图的目标是给出全局图景。 这里列出了我们的关键关注领域——我们专注于交付的最重要的方向：

- **Language evolution**: more efficient data handling, increased abstraction, enhanced performance with clear code.
- **Kotlin Multiplatform**: release direct Kotlin to Swift Export, streamlined build setup, and simplified creation of multiplatform libraries.
- **Experience of third-party ecosystem authors**: simplified development and publication process for Kotlin libraries, tools, and frameworks.

以子系统划分的 Kotlin 路线图

If you have any questions or feedback about the roadmap or the items on it, feel free to post them to YouTrack tickets or in the #kotlin-roadmap channel of Kotlin Slack (request an invite).

子系统	当前聚焦
语言	See the full list of Kotlin language features and proposals or follow the YouTrack issue for upcoming language features
编译器	• Finalize JSpecify support • Deprecate K1 compiler • Promote Kotlin/Wasm (`wasm-js` target) to Beta • Kotlin/Wasm: Switch `wasm-wasi` target of libraries to WASI Preview 2 • Kotlin/Wasm: Support Component Model
多	• The first public release of Swift Export • Enable Concurrent Mark and Sweep (CMS) GC by default • Stabilize klib cross-compilation on different platforms • Implement the next generation distribution format of multiplatform libraries

平台	• Support declaring Kotlin Multiplatform dependencies at project-level • Unify inline semantics between all Kotlin targets • Enable incremental compilation of klib artifacts by default Kotlin Multiplatform development roadmap
工具	• Improve development experience for Kotlin/Wasm projects in IntelliJ IDEA • Improve performance of imports • Support resources in XCFrameworks • Kotlin Notebook: Smoother access and improved experience • IntelliJ IDEA K2 mode complete release • Design the Build Tools API • Kotlin Ecosystem Plugin supporting Declarative Gradle • Support Gradle project isolation • Improve integration of Kotlin/Native toolchain into Gradle • Improve Kotlin build reports • Expose stable compiler arguments in Gradle DSL • Improve Kotlin scripting and experience with `.gradle.kts`
库生态	**Library ecosystem roadmap items:** • Refine the Dokka HTML output UI • Introduce default warnings/errors for Kotlin functions that return non-unit values that are unused • New multiplatform API for the standard library: Support for Unicode and codepoints • Stabilize the `kotlinx-io` library • Improve Kotlin distribution UX: add code coverage and binary compatibility validation • Promote `kotlinx-datetime` to Beta **Ktor:** • Add gRPC support to Ktor with a generator plugin and tutorial • Make project structuring for the backend applications simple • Publish CLI generator to SNAP • Create Kubernetes Generator Plugin • Make Dependency Injection Usage Simple • HTTP/3 Support **Exposed:** • Release 1.0.0 • Add R2DBC Support

> • This roadmap is not an exhaustive list of all things the team is working on, only the biggest projects.
> • There's no commitment to delivering specific features or fixes in specific versions.
> • We will adjust our priorities as we go and update the roadmap approximately every six months.



相对 2024 年 9 月版的变化

已完成

We've **completed** the following items from the previous roadmap:

- Compiler: Support debugging inline functions on Android
- Compiler: Improve the quality of compiler diagnostics
- Multiplatform: Support Xcode 16 in Kotlin
- Multiplatform: Publish publicly available API reference for Kotlin Gradle Plugin
- Tooling: Provide out-of-the-box debugging experience for Kotlin/Wasm targets
- Library ecosystem: Implement new Dokka Gradle plugin based on Dokkatoo
- Library ecosystem: New multiplatform API for the standard library: Atomics
- Library ecosystem: Expand Library authors' guidelines

新增

We've **added** the following items to the roadmap:

- Compiler: Finalize JSpecify support
- Compiler: Deprecate K1 compiler
- Compiler: Promote Kotlin/Wasm (`wasm-js` target) to Beta
- Tooling: Improve development experience for Kotlin/Wasm projects in IntelliJ IDEA
- Tooling: Improve performance of imports
- Tooling: Support resources in XCFrameworks
- Tooling: Smoother access and improved experience in Kotlin Notebook
- Ktor: Add gRPC support to Ktor with a generator plugin and tutorial
- Ktor: Make project structuring for the backend applications simple
- Ktor: Publish CLI generator to SNAP
- Ktor: Create Kubernetes Generator Plugin
- Ktor: Make Dependency Injection Usage Simple
- Ktor: HTTP/3 Support
- Exposed: Release 1.0.0
- Exposed: Add R2DBC Support

进行中

All other previously identified roadmap items are in progress. You can check their YouTrack tickets for updates.

Kotlin 语言特性与提案

JetBrains evolves the Kotlin language according to the Kotlin language evolution principles, guided by pragmatic design.

> Language feature proposals are listed from Kotlin 1.7.0.
>
> See the explanation of language feature statuses in the Kotlin evolution principles documentation.
>
> 

【All】

Exploration and design	**Kotlin statics and static extensions** * KEEP proposal: [statics.md](https://github.com/Kotlin/KEEP/blob/statics/proposals/statics.md) * YouTrack issue: [KT-11968](https://youtrack.jetbrains.com/issue/KT-11968)
Exploration and design	**Collection literals** * KEEP proposal: Not defined * YouTrack issue: [KT-43871](https://youtrack.jetbrains.com/issue/KT-43871)
Exploration and design	**Union types for errors and exceptions** * KEEP proposal: Not defined * YouTrack issue: [KT-68296](https://youtrack.jetbrains.com/issue/KT-68296)
Exploration and design	**Name-based destructuring** * KEEP proposal: Not defined * YouTrack issue: [KT-19627](https://youtrack.jetbrains.com/issue/KT-19627)
Exploration and design	**Support immutability** * KEEP notes: [immutability](https://github.com/Kotlin/KEEP/blob/master/notes/value-classes.md#immutability-and-value-classes) * YouTrack issue: [KT-1179](https://youtrack.jetbrains.com/issue/KT-1179)
KEEP discussion	**KMP Kotlin-to-Java direct actualization** * KEEP proposal: [kmp-kotlin-to-java-direct-actualization.md](https://github.com/Kotlin/KEEP/blob/kotlin-to-java-direct-actualization/proposals/kmp-kotlin-to-java-direct-actualization.md) * YouTrack issue: [KT-67202](https://youtrack.jetbrains.com/issue/KT-67202)
KEEP discussion	**kotlin.time.Instant** * KEEP proposal: [Instant and Clock](https://github.com/dkhalanskyjb/KEEP/blob/dkhalanskyjb-instant/proposals/stdlib/instant.md)
KEEP discussion	**Common Atomics and Atomic Arrays** * KEEP proposal: [Common atomics](https://github.com/Kotlin/KEEP/blob/mvicsokolova/common-atomics/proposals/common-atomics.md) * YouTrack issue: [KT-62423](https://youtrack.jetbrains.com/issue/KT-62423)
	Improvements to annotation use-site targets on properties * KEEP proposal: [Improvements to annotation use-site targets on properties]

discussion**	(https://github.com/Kotlin/KEEP/blob/change-defaulting-rule/proposals/change-defaulting-rule.md) * YouTrack issue: [KT-19289](https://youtrack.jetbrains.com/issue/KT-19289)
KEEP discussion	**Nested (non-capturing) type aliases** * KEEP proposal: [Nested (non-capturing) type aliases](https://github.com/Kotlin/KEEP/blob/nested-typealias/proposals/nested-typealias.md) * YouTrack issue: [KT-45285](https://youtrack.jetbrains.com/issue/KT-45285)
KEEP discussion	**Streamline KDoc ambiguity links** * KEEP proposal: [streamline-KDoc-ambiguity-references.md](https://github.com/Kotlin/KEEP/blob/kdoc/Streamline-KDoc-ambiguity-references/proposals/kdoc/streamline-KDoc-ambiguity-references.md) * GitHub issues: [dokka/#3451](https://github.com/Kotlin/dokka/issues/3451), [dokka/#3179](https://github.com/Kotlin/dokka/issues/3179), [dokka/#3334](https://github.com/Kotlin/dokka/issues/3334)
KEEP discussion	**Resolution of links to extensions in KDoc** * KEEP proposal: [links-to-extensions.md](https://github.com/Kotlin/KEEP/blob/kdoc/extension-links/proposals/kdoc/links-to-extensions.md) * GitHub issue: [dokka/#3555](https://github.com/Kotlin/dokka/issues/3555)
KEEP discussion	**Uuid** * KEEP proposal: [uuid.md](https://github.com/Kotlin/KEEP/blob/uuid/proposals/stdlib/uuid.md) * YouTrack issue: [KT-31880](https://youtrack.jetbrains.com/issue/KT-31880)
KEEP discussion	**Improve resolution using expected type** * KEEP proposal: [improved-resolution-expected-type.md](https://github.com/Kotlin/KEEP/blob/improved-resolution-expected-type/proposals/improved-resolution-expected-type.md) * YouTrack issue: [KT-16768](https://youtrack.jetbrains.com/issue/KT-16768)
KEEP discussion	**Expose boxed inline value classes in JVM** * KEEP proposal: [jvm-expose-boxed.md](https://github.com/Kotlin/KEEP/blob/jvm-expose-boxed/proposals/jvm-expose-boxed.md) * YouTrack issue: [KT-28135](https://youtrack.jetbrains.com/issue/KT-28135)
KEEP discussion	**Explicit backing fields: both `public` and `private` type for the same property** * KEEP proposal: [explicit-backing-fields.md](https://github.com/Kotlin/KEEP/blob/explicit-backing-fields-re/proposals/explicit-backing-fields.md) * YouTrack issue: [KT-14663](https://youtrack.jetbrains.com/issue/KT-14663)
KEEP discussion	**Context parameters: support for context-dependent declarations** * KEEP proposal: [context-parameters.md](https://github.com/Kotlin/KEEP/blob/context-parameters/proposals/context-parameters.md) * YouTrack issue: [KT-14663](https://youtrack.jetbrains.com/issue/KT-10468)
KEEP discussion	**Java synthetic property references** * KEEP proposal: [references-to-java-synthetic-properties.md](https://github.com/Kotlin/KEEP/blob/master/proposals/references-to-java-synthetic-properties.md) * YouTrack issue: [KT-8575](https://youtrack.jetbrains.com/issue/KT-8575) * Target version: 2.2.0
**In	**Guard conditions in when-with-subject** * KEEP proposal: [guards.md](https://github.com/Kotlin/KEEP/blob/guards/proposals/guards.md) *

In preview	(https://github.com/Kotlin/KEEP/blob/guards/proposals/guards.md) * YouTrack issue: [KT-13626](https://youtrack.jetbrains.com/issue/KT-13626) * Available since: 2.1.0
Stable	**Stabilized `@SubclassOptInRequired`** * KEEP proposal: [subclass-opt-in-required.md](https://github.com/Kotlin/KEEP/blob/master/proposals/subclass-opt-in-required.md) * YouTrack issue: [KT-54617] (https://youtrack.jetbrains.com/issue/KT-54617) * Available since: 2.1.0
In preview	**Multidollar interpolation: improved handling of `$` in string literals** * KEEP proposal: [dollar-escape.md] (https://github.com/Kotlin/KEEP/blob/master/proposals/dollar-escape.md) * YouTrack issue: [KT-2425](https://youtrack.jetbrains.com/issue/KT-2425) * Available since: 2.1.0
In preview	**Non-local `break` and `continue`** * KEEP proposal: [break-continue-in-inline-lambdas.md] (https://github.com/Kotlin/KEEP/blob/master/proposals/break-continue-in-inline-lambdas.md) * YouTrack issue: [KT-1436] (https://youtrack.jetbrains.com/issue/KT-1436) * Available since: 2.1.0
Stable	**`Enum.entries`: performant replacement of the `Enum.values()`** * KEEP proposal: [enum-entries.md] (https://github.com/Kotlin/KEEP/blob/master/proposals/enum-entries.md) * YouTrack issue: [KT-48872](https://youtrack.jetbrains.com/issue/KT-48872) * Target version: 2.0.0
Stable	**Data objects** * KEEP proposal: [data-objects.md] (https://github.com/Kotlin/KEEP/blob/master/proposals/data-objects.md) * YouTrack issue: [KT-4107](https://youtrack.jetbrains.com/issue/KT-4107) * Target version: 1.9.0
Stable	**RangeUntil operator `..<`** * KEEP proposal: [open-ended-ranges.md] (https://github.com/kotlin/KEEP/blob/open-ended-ranges/proposals/open-ended-ranges.md) * YouTrack issue: [KT-15613] (https://youtrack.jetbrains.com/issue/KT-15613) * Target version: 1.7.20
Stable	**Definitely non-nullable types** * KEEP proposal: [definitely-non-nullable-types.md](https://github.com/Kotlin/KEEP/blob/master/proposals/definitely-non-nullable-types.md) * YouTrack issue: [KT-26245] (https://youtrack.jetbrains.com/issue/KT-26245) * Target version: 1.7.0
Revoked	**Context receivers** * KEEP proposal: [context-receivers.md] (https://github.com/Kotlin/KEEP/blob/master/proposals/context-receivers.md) * YouTrack issue: [KT-10468](https://youtrack.jetbrains.com/issue/KT-10468)

> **【Exploration and design】**

> **【KEEP discussion】**

> **【In preview】**

> **【Stable】**

> **【Revoked】**

Kotlin 演进原则

实用主义演进原则

> 语言的设计是坚如磐石的，
>
> 但这块石头相当柔软，
>
> 经过一番努力，我们可以后期重塑它。
>
> *Kotlin* 设计团队

Kotlin 旨在成为程序员的实用工具。在语言演进方面，它的实用主义本质遵循以下原则：

- 一直保持语言的现代性。
- 与用户保持持续的反馈循环。
- 使版本更新对用户来说简单且舒适。

由于这是理解 Kotlin 如何向前发展的关键，我们来展开来看这些原则。

保持语言现代性。我们认识到系统随着时间的推移积累了很多遗留问题。曾经是尖端 技术的东西如今可能已经无可救药地过时了。我们必须改进语言，使其与我们用户的需求保持一致、 与用户期望保持同步。这不仅包括添加新特性，还包括逐步淘汰 不再推荐用于生产并且已经成为历史的旧特性。

舒适的更新。如果没有适度谨慎地进行不兼容的变更（例如从语言中删除内容）可能会导致从一个版本到下一个版本的痛苦迁移过程。我们会始终提前公布这类变更， 将相应内容标记为已弃用并 在变更发生之前 提供自动化的迁移工具。当语言发生变更之时， 我们希望世界上绝大多数代码都已经更新，这样迁移到新版本就没有问题了。

反馈循环。 通过弃用周期需要付出很大的努力，因此我们希望最大限度地减少将来不兼容变更的数量。除了使用我们的最佳判断之外，我们相信在现实生活中试用是验证设计的最佳方法。在最终定论之前，我们希望已经实战测试过。这就是为什么我们利用每个机会在语言的生产版本中提供我们早期版设计，只是处于 稳定前状态的一种：实验性、 Alpha、 Beta。这些特性并不稳定， 可以随时更改，选择使用它们的用户明确表示已准备好了应对未来的迁移问题。这些用户提供了宝贵的反馈，而我们收集这些反馈来迭代设计并使其坚如磐石。

不兼容的变更

如果从一个版本更新到另一个版本时，一些以前工作的代码不再工作， 那么它是语言中的 不兼容的变更（有时称为“破坏性变更”）。 在一些场景中“不再工作”的确切含义可能会有争议，但是它肯定包含以下内容：

- 之前编译运行正常的代码现在（编译或链接）失败并报错。 这包括删除语言结构以及添加新的限制。
- 之前正常执行的代码现在抛异常了。

属于“灰色区域”的不太明显的情况包括以不同方式处理极端情况，抛出与以前不同类型的异常， 更改仅通过反射可以观察到的行为，修改未记录或未定义的行为，重命名二进制文件等等。 有时这些更改至关重要，并且会极大地影响迁移体验，有时影响微不足道。

绝对不是不兼容的变更的一些示例包括：

- 添加新的警告。
- 启用新的语言结构或放宽对现有语言结构的限制。
- 更改私有/内部 API 和其他实现细节。

保持语言现代化和舒适更新的原则表明，有时有必要进行不兼容的更改， 但应该详细介绍这些更改。我们的目标是使用户提前了解即将发生的更改，以便他们能够轻松地迁移代码。

理想情况下，应通过有问题的代码中报告的编译期警告（通常称为弃用警告）来宣告每个不兼容的更改，并辅以自动化迁移工具。 因此，理想的迁移工作流程如下：

- 更新到版本 A（宣布更改）
  - 查看有关即将发生的更改的警告
  - 借助工具迁移代码
- 更新到版本 B（发生更改）
  - 完全没有问题

实际上，在编译期无法准确检测到某些更改，因此不会报告任何警告， 但是至少会通过版本 A 的发行说明通知用户版本 B 中即将进行的更改。

处理编译器错误

编译器是复杂的软件，尽管开发人员尽了最大努力，但它们仍然存在 bug。 导致编译器自身编译失败、或报告虚假错误、或生成明显编译失败的代码的 bug， 虽然很烦人并且常常令人尴尬，但它们很容易修复，因为这些修复不构成不兼容的变更。 其他 bug 可能会导致编译器生成不会编译失败的错误代码： 例如，遗漏了源代码中的一些错误，或者只是生成了错误的指令。 这些 bug 的修复是技术上不兼容的更改（某些代码过去可以正常编译，但现在编译失败）， 但是我们倾向于尽快修复它们，以防止不良代码模式在用户代码中传播。 我们认为，这符合“舒适更新”的原则， 因为较少的用户有机会遇到此问题。 当然，这仅适用于在发行版本中出现后不久发现的 bug。

决策制定

JetBrains 是 Kotlin 的原始创建者， 在社区的帮助下并与 Kotlin 基金会 合作来推动其发展。

首席语言设计师（现为 Michail Zarečenskij）负责监督 Kotlin 编程语言的所有更改。 首席设计师在与语言发展有关的所有事务中拥有最终决定权。 此外，对完全稳定的组件进行不兼容的更改必须完全由Kotlin 基金会指定的语言委员会（目前由 Jeffrey van Gogh、Werner Dietl 与 Michail Zarečenskij 组成）批准。

语言委员会对将进行哪些不兼容的更改以及应采取什么确切的措施使用户尽可能无缝更新做出最终决定。 为此，它依赖一组语言委员会准则。

Language and tooling releases

类似 2.0.0 等版本的稳定版本通常被认为是对语言进行重大更改的语言版本。 通常，在语言版本之间会发布工具版本，编号为 x.x.**20**。

工具版本带来了工具方面的更新（通常包括特性），性能提升和错误修复。 我们试图使这些版本彼此兼容， 因此对编译器的更改主要是优化和添加/删除警告。 稳定前可能随时添加、删除或更改特性。

语言版本通常会添加新特性，并且可能会删除或更改以前弃用的特性。 某项特性从稳定前到稳定版的过渡也包含在语言版本中。

早期预览版本

在发布语言与工具的稳定版本之前， 我们会发布许多称为 EAP（“Early Access Preview”）的早期预览版本，这些版本使我们能够更快地进行迭代并从社区中收集反馈。 语言版本的早期预览版本通常会生成二进制文件，这些文件随后将被稳定的编译器拒绝， 以确保二进制文件中可能存在的错误只在预览期出现。 最终发布的候选版本通常没有此限制。

稳定前特性

根据上述反馈环原则，我们在语言的开放与发行版本中对设计进行迭代，其中某些特性具有稳定前状态之一并且可以更改。 这些特性可以随时被添加、更改或删除，不会发出警告。 我们尽量确保稳定前特性不会被用户意外使用。 此类特性通常需要在代码或项目配置中进行某种类型的显式选择。

A Kotlin language feature can have one of the following statuses:

- **Exploration and design**. We are considering the introduction of a new feature to the language. This involves discussing how it would integrate with existing features, gathering use cases, and assessing its potential impact. We need feedback from users on the problems this feature would solve and the use cases it addresses. Whenever possible, we try to estimate how often these use cases and problems occur would also be beneficial. Typically, ideas are documented as YouTrack issues, where the discussion continues.
- **KEEP discussion**. We are fairly certain that the feature should be added to the language. We aim to provide a motivation, use-cases, design, and other important details in a document called a *KEEP*. We expect feedback from users to focus on discussing all the information provided in the KEEP.
- **In preview**. A feature prototype is ready, and you can enable it using a feature-specific compiler option. We seek feedback on your experience with the feature, including how easily it integrates into your codebase, how it interacts with existing code, and any IDE support issues or suggestions. The feature's design may change significantly, or it could be completely revoked based on feedback. When a feature is *in preview*, it has a stability level.
- **Stable**. The language feature is now a first-class citizen in the Kotlin language. We guarantee its backward compatibility and that we'll provide tooling support.
- **Revoked**. We have revoked the proposal and will not implement the feature in the Kotlin language. We may revoke a feature that is *in preview* if it is not a good fit for Kotlin.

See the full list of Kotlin language proposals and their statuses.

不同组件的状态

了解更多关于 Kotlin 的不同组件的稳定性状态， 例如 Kotlin/JVM、JS 与 Native 编译器以及各种库。

Libraries

A language is nothing without its ecosystem, so we pay extra attention to enable smooth library evolution.

Ideally, a new version of a library can be used as a "drop-in replacement" for an older version. This means that upgrading a binary dependency should not break anything, even if the application is not recompiled (this is possible under dynamic linking).

On the one hand, to achieve this, the compiler has to provide certain *Application Binary Interface* (ABI) stability guarantees under the constraints of separate compilation. This is why every change in the language is examined from a binary compatibility standpoint.

On the other hand, a lot depends on the library authors being careful about which changes are safe to make. Thus, it's crucial that library authors understand how source changes affect compatibility and follow certain best practices to keep both APIs and ABIs of their libraries stable. Here are some assumptions that we make when considering language changes from the library evolution standpoint:

- Library code should always specify return types of public/protected functions and properties explicitly, thus never relying on type inference for public API. Subtle changes in type inference may cause return types to change inadvertently, leading to binary compatibility issues.
- Overloaded functions and properties provided by the same library should do essentially the same thing. Changes in type inference may result in more precise static types to be known at call sites, causing changes in overload resolution.

Library authors can use the `@Deprecated` and `@RequiresOptIn` annotations to control the evolution of their API surface. Note that `@Deprecated(level=HIDDEN)` can be used to preserve binary compatibility even for declarations removed from the API.

Also, by convention, packages named "internal" are not considered public API. All API residing in packages named "experimental" is considered pre-stable and can change at any moment.

We evolve the Kotlin Standard Library (`kotlin-stdlib`) for stable platforms according to the principles stated above. Changes to the contracts for its API undergo the same procedures as changes in the language itself.

Compiler options

Command line options accepted by the compiler are also a kind of public API, and they are subject to the same considerations. Supported options (those that don't have the "-X" or "-XX" prefix) can be added only in language releases and should be properly deprecated before removing them. The "-X" and "-XX" options are experimental and can be added and removed at any time.

Compatibility tools

As legacy features get removed and bugs fixed, the source language changes, and old code that has not been properly migrated may not compile anymore. The normal deprecation cycle allows a comfortable period of time for migration, and even when it's over and the change ships in a stable version, there's still a way to compile non-migrated code.

Compatibility options

We provide the `-language-version X.Y` and `-api-version X.Y` options that make a new version emulate the behavior of an old one for compatibility purposes. To give you more time for migration, we support the three previous language and API versions in addition to the latest stable one.

Actively maintained code bases can benefit from getting bug fixes as soon as possible, without waiting for a full deprecation cycle to complete. Currently, such projects can enable the `-progressive` option and get such fixes enabled even in tooling releases.

All options are available in the command line as well as in Gradle and in Maven.

Evolving the binary format

Unlike sources that can be fixed by hand in the worst case, binaries are a lot harder to migrate, and this makes backwards compatibility crucial in the case of binaries. Incompatible changes to binaries can make updates very uncomfortable and thus should be introduced with even more care than those in the source language syntax.

For fully stable versions of the compiler, the default binary compatibility protocol is the following:

- All binaries are backwards compatible; that means a newer compiler can read older binaries (for example, 1.3 understands 1.0 through 1.2).
- Older compilers reject binaries that rely on new features (for example, the 1.0 compiler rejects binaries that use coroutines).
- Preferably (but we can't guarantee it), the binary format is mostly forwards compatible with the next language release, but not later ones (in the cases when new features are not used, for example, 1.9 can understand most binaries from 2.0, but not 2.1).

This protocol is designed for comfortable updates as no project can be blocked from updating its dependencies even if it's using a slightly outdated compiler.

Note that not all target platforms have reached this level of stability, but Kotlin/JVM has.

Kotlin klib binaries

Kotlin klib binaries have reached the Stable level in Kotlin 1.9.20. However, there are some compatibility details you need to keep in mind:

- klib binaries are backwards compatible starting with Kotlin 1.9.20. For example, the 2.0.x compiler can read binaries produced by the 1.9.2x compiler.
- Forward compatibility is *not* guaranteed. For example, the 2.0.x compiler is *not* guaranteed to read binaries produced by the 2.1.x compiler.

> The Kotlin cinterop klib binaries are still in Beta. Currently, we cannot give specific compatibility guarantees between different Kotlin versions for cinterop klib binaries.



Kotlin 各组件的稳定性

The Kotlin language and toolset are divided into many components such as the compilers for the JVM, JS and Native targets, the Standard Library, various accompanying tools and so on. Many of these components were officially released as **Stable**, which means that they were evolved in a backward-compatible way following the principles of *Comfortable Updates* and *Keeping the Language Modern*.

Following the *Feedback Loop* principle, we release many things early for the community to try out, so a number of components are not yet released as **Stable**. Some of them are at a very early stage, some are more mature. We mark them as **Experimental**, **Alpha** or **Beta** depending on how quickly each component evolves and the level of risk users take on when adopting it.

Stability levels explained

Here's a quick guide to these stability levels and their meaning:

实验性的 means "try it only in toy projects":

- We are just trying out an idea and want some users to play with it and give feedback. If it doesn't work out, we may drop it any minute.

Alpha means "use at your own risk, expect migration issues":

- We intend to productize this idea, but it hasn't reached its final shape yet.

Beta means "you can use it, we'll do our best to minimize migration issues for you":

- It's almost done, user feedback is especially important now.
- Still, it's not 100% finished, so changes are possible (including ones based on your own feedback).
- Watch for deprecation warnings in advance for the best update experience.

We collectively refer to *Experimental*, *Alpha* and *Beta* as **pre-stable** levels.

已稳定 means "use it even in most conservative scenarios":

- It's done. We will be evolving it according to our strict backward compatibility rules.

Please note that stability levels do not say anything about how soon a component will be released as Stable. Similarly, they do not indicate how much a component will be changed before release. They only say how fast a component is changing and how much risk of update issues users are running.

GitHub badges for Kotlin components

The Kotlin GitHub organization hosts different Kotlin-related projects. Some of them we develop full-time, while others are side projects.

Each Kotlin project has two GitHub badges describing its stability and support status:

- **Stability** status. This shows how quickly each project is evolving and how much risk the users are taking when adopting it. The stability status completely coincides with the stability level of the Kotlin language features and its components:
  - stands for **Experimental**
  - stands for **Alpha**
  - stands for **Beta**
  - stands for **Stable**
- **Support** status. This shows our commitment to maintaining a project and helping users to solve their problems. The level of support is unified for all JetBrains products.

  See the JetBrains Open Source document for details.

Stability of subcomponents

A stable component may have an experimental subcomponent, for example:

- a stable compiler may have an experimental feature;
- a stable API may include experimental classes or functions;
- a stable command-line tool may have experimental options.

We make sure to document precisely which subcomponents are not **Stable**. We also do our best to warn users where possible and ask to opt them in explicitly to avoid the accidental use of features that have not been released as stable.

Current stability of Kotlin components

> By default, all new components have Experimental status.

Kotlin 编译器

组件	状态	状态起始版本	备注
Kotlin/JVM	已稳定	1.0.0	
Kotlin/Native	已稳定	1.9.0	
Kotlin/JS	已稳定	1.3.0	
Kotlin/Wasm	Alpha	1.9.20	
分析 API	已稳定		

核心编译器插件

组件	状态	状态起始版本	备注
All-open	已稳定	1.3.0	
No-arg	已稳定	1.3.0	
SAM-with-receiver	已稳定	1.3.0	
kapt	已稳定	1.3.0	
Lombok	实验性的	1.5.20	
Power-assert	实验性的	2.0.0	

Kotlin 库

组件	状态	状态起始版本	备注
kotlin-stdlib (JVM)	已稳定	1.0.0	
kotlinx-coroutines	已稳定	1.3.0	
kotlinx-serialization	已稳定	1.0.0	
kotlin-metadata-jvm	已稳定	2.0.0	
kotlin-reflect (JVM)	Beta	1.0.0	
kotlinx-datetime	Alpha	0.2.0	
kotlinx-io	Alpha	0.2.0	

Kotlin 多平台

		状态起始版本	

		状态起始版本	
Kotlin 多平台	已稳定	1.9.20	
用于 Android Studio 的 Kotlin 多平台插件	Beta	0.8.0	与语言版本独立

Kotlin/Native

组件	状态	状态起始版本	备注
Kotlin/Native 运行时	已稳定	1.9.20	
Kotlin/Native 与 C 语言及 Objective-C 互操作	Beta	1.3.0	
klib 二进制	已稳定	1.9.20	不包括 cinterop klibs，请参阅下文
cinterop klib 二进制	Beta	1.3.0	
CocoaPods 集成	已稳定	1.9.20	

> For details about Kotlin/Native targets support, see .

语言工具

组件	状态	状态起始版本	备注
脚本语法与语义	Alpha	1.2.0	
脚本嵌入与扩展 API	Beta	1.5.0	
脚本 IDE 支持	Beta		Available since IntelliJ IDEA 2023.1 and later
CLI 脚本	Alpha	1.2.0	

Language features and design proposals

For language features and new design proposals, see .

Kotlin 版本发布

Latest Kotlin version: **2.1.21**

See What's new in Kotlin 2.1.21 for details

Since Kotlin 2.0.0, we ship the following types of releases:

- *Language releases* (2.*x*.*0*) that bring major changes in the language and include tooling updates. Released once in 6 months.
- *Tooling releases* (2.*x*.*20*) that are shipped between language releases and include updates in the tooling, performance improvements, and bug fixes. Released in 3 months after corresponding *language release*.
- *Bug fix releases* (2.*x*.*yz*) that include bug fixes for *tooling releases*. There is no exact release schedule for these releases.

For each language and tooling release, we also ship several preview (*EAP*) versions for you to try new features before they are released. See Early Access Preview for details.

> If you want to be notified about new Kotlin releases, subscribe to the Kotlin newsletter, follow Kotlin on X, or enable the **Watch | Custom | Releases** option on the Kotlin GitHub repository.
>
> 

更新到新版 Kotlin

To upgrade your project to a new release, you need to update your build script file. For example, to update to Kotlin 2.1.21, change the version of the Kotlin Gradle plugin in your `build.gradle(.kts)` file:

> **【Kotlin】**

```
plugins {
    // Replace `<...>` with the plugin name appropriate for your target environment
    kotlin("<...>") version "2.1.21"
    // For example, if your target environment is JVM:
    // kotlin("jvm") version "2.1.21"
    // If your target is Kotlin Multiplatform:
    // kotlin("multiplatform") version "2.1.21"
}
```

【Groovy】

```
plugins {
    // Replace `<...>` with the plugin name appropriate for your target environment
    id 'org.jetbrains.kotlin.<...>' version '2.1.21'
    // For example, if your target environment is JVM:
    // id 'org.jetbrains.kotlin.jvm' version '2.1.21'
    // If your target is Kotlin Multiplatform:
    // id 'org.jetbrains.kotlin.multiplatform' version '2.1.21'
}
```

If you have projects created with earlier Kotlin versions, change the Kotlin version in your projects and update kotlinx libraries if necessary.

If you are migrating to the new language release, Kotlin plugin's migration tools will help you with the migration.

IDE 支持

Kotlin has full out-of-the-box support in IntelliJ IDEA and Android Studio with an official Kotlin plugin developed by JetBrains.

K2 mode in IntelliJ IDEA and Android Studio uses the K2 compiler to improve code analysis, code completion, and highlighting.

Starting with IntelliJ IDEA 2025.1, K2 mode is enabled by default.

In Android Studio, you can enable K2 mode starting with 2024.1 by following these steps:

1. Go to **Settings** | **Languages & Frameworks** | **Kotlin**.
2. Select the **Enable K2 mode** option.

Learn more about K2 mode in our blog.

Kotlin release compatibility

Learn more about types of Kotlin releases and their compatibility

版本发布详情

The following table lists details of the latest Kotlin releases:

> You can also try Early Access Preview (EAP) versions of Kotlin.
>
> 

版本信息	版本重点
2.1.21 Released: **May 13, 2025** Release on GitHub	A bug fix release for Kotlin 2.1.20. For more details, please refer to the changelog.
2.1.20 Released: **March 20, 2025** Release on GitHub	A tooling release for Kotlin 2.1.0 containing new experimental features, performance improvements, and bug fixes. Learn more about Kotlin 2.1.20 in What's new in Kotlin 2.1.20.
2.1.10 Released: **January 27, 2025** Release on GitHub	A bug fix release for Kotlin 2.1.0 For more details, please refer to the changelog.
2.1.0 Released: **November 27, 2024** Release on GitHub	A language release introducing new language features. Learn more about Kotlin 2.1.0 in What's new in Kotlin 2.1.0.
2.0.21 Released: **October 10, 2024** Release on GitHub	A bug fix release for Kotlin 2.0.20 For more details, please refer to the changelog.
2.0.20 Released: **August 22, 2024**	A tooling release for Kotlin 2.0.0 containing performance improvements and bug fixes. Features also include concurrent marking in Kotlin/Native's garbage collector, UUID support in Kotlin common standard library, Compose compiler updates, and support up to Gradle 8.8.

Release on GitHub	Learn more about Kotlin 2.0.20 in What's new in Kotlin 2.0.20.
2.0.10 Released: **August 6, 2024** Release on GitHub	A bug fix release for Kotlin 2.0.0. Learn more about Kotlin 2.0.0 in What's new in Kotlin 2.0.0.
2.0.0 Released: **May 21, 2024** Release on GitHub	A language release with the Stable Kotlin K2 compiler. Learn more about Kotlin 2.0.0 in What's new in Kotlin 2.0.0.
1.9.25 Released: **July 19, 2024** Release on GitHub	A bug fix release for Kotlin 1.9.20, 1.9.21, 1.9.22, 1.9.23, and 1.9.24. Learn more about Kotlin 1.9.20 in What's new in Kotlin 1.9.20.
1.9.24 Released: **May 7, 2024** Release on GitHub	A bug fix release for Kotlin 1.9.20, 1.9.21, 1.9.22, and 1.9.23. Learn more about Kotlin 1.9.20 in What's new in Kotlin 1.9.20.
1.9.23 Released: **March 7, 2024** Release on GitHub	A bug fix release for Kotlin 1.9.20, 1.9.21, and 1.9.22. Learn more about Kotlin 1.9.20 in What's new in Kotlin 1.9.20.
1.9.22 Released: **December 21, 2023** Release on GitHub	A bug fix release for Kotlin 1.9.20 and 1.9.21. Learn more about Kotlin 1.9.20 in What's new in Kotlin 1.9.20.
1.9.21	

Released: **November 23, 2023** Release on GitHub	A bug fix release for Kotlin 1.9.20. Learn more about Kotlin 1.9.20 in What's new in Kotlin 1.9.20.
1.9.20 Released: **November 1, 2023** Release on GitHub	A feature release with Kotlin K2 compiler in Beta and Stable Kotlin Multiplatform. Learn more in: • What's new in Kotlin 1.9.20
1.9.10 Released: **August 23, 2023** Release on GitHub	A bug fix release for Kotlin 1.9.0. Learn more about Kotlin 1.9.0 in What's new in Kotlin 1.9.0. For Android Studio Giraffe and Hedgehog, the Kotlin plugin 1.9.10 will be delivered with upcoming Android Studios updates.
1.9.0 Released: **July 6, 2023** Release on GitHub	A feature release with Kotlin K2 compiler updates, new enum class values function, new operator for open-ended ranges, preview of Gradle configuration cache in Kotlin Multiplatform, changes to Android target support in Kotlin Multiplatform, preview of custom memory allocator in Kotlin/Native. Learn more in: • What's new in Kotlin 1.9.0 • What's new in Kotlin YouTube video
1.8.22 Released: **June 8, 2023** Release on GitHub	A bug fix release for Kotlin 1.8.20. Learn more about Kotlin 1.8.20 in What's new in Kotlin 1.8.20.
1.8.21 Released: **April 25, 2023** Release on GitHub	A bug fix release for Kotlin 1.8.20. Learn more about Kotlin 1.8.20 in What's new in Kotlin 1.8.20. For Android Studio Flamingo and Giraffe, the Kotlin plugin 1.8.21 will be delivered with upcoming Android Studios updates.
1.8.20	A feature release with Kotlin K2 compiler updates, AutoCloseable interface and Base64 encoding in stdlib, new JVM incremental compilation enabled by default, new Kotlin/Wasm compiler backend.

Release on GitHub	Learn more in: • What's new in Kotlin 1.8.20 • What's new in Kotlin YouTube video
1.8.10 Released: **February 2, 2023** Release on GitHub	A bug fix release for Kotlin 1.8.0. Learn more about Kotlin 1.8.0. For Android Studio Electric Eel and Flamingo, the Kotlin plugin 1.8.10 will be delivered with upcoming Android Studios updates.
1.8.0 Released: **December 28, 2022** Release on GitHub	A feature release with improved kotlin-reflect performance, new recursively copy or delete directory content experimental functions for JVM, improved Objective-C/Swift interoperability. Learn more in: • What's new in Kotlin 1.8.0 • Compatibility guide for Kotlin 1.8.0
1.7.21 Released: **November 9, 2022** Release on GitHub	A bug fix release for Kotlin 1.7.20. Learn more about Kotlin 1.7.20 in What's new in Kotlin 1.7.20. For Android Studio Dolphin, Electric Eel, and Flamingo, the Kotlin plugin 1.7.21 will be delivered with upcoming Android Studios updates.
1.7.20 Released: **September 29, 2022** Release on GitHub	An incremental release with new language features, the support for several compiler plugins in the Kotlin K2 compiler, the new Kotlin/Native memory manager enabled by default, and the support for Gradle 7.1. Learn more in: • What's new in Kotlin 1.7.20 • What's new in Kotlin YouTube video • Compatibility guide for Kotlin 1.7.20 Learn more about Kotlin 1.7.20.
1.7.10 Released: **July 7, 2022** Release on GitHub	A bug fix release for Kotlin 1.7.0. Learn more about Kotlin 1.7.0. For Android Studio Dolphin (213) and Android Studio Electric Eel (221), the Kotlin plugin 1.7.10 will be delivered with upcoming Android Studios updates.
1.7.0	

Released: **June 9, 2022** Release on GitHub	Learn more in: • What's new in Kotlin 1.7.0 • What's new in Kotlin YouTube video • Compatibility guide for Kotlin 1.7.0
1.6.21 Released: **April 20, 2022** Release on GitHub	A bug fix release for Kotlin 1.6.20. Learn more about Kotlin 1.6.20.
1.6.20 Released: **April 4, 2022** Release on GitHub	An incremental release with various improvements such as: • Prototype of context receivers • Callable references to functional interface constructors • Kotlin/Native: performance improvements for the new memory manager • Multiplatform: hierarchical project structure by default • Kotlin/JS: IR compiler improvements • Gradle: compiler execution strategies Learn more about Kotlin 1.6.20.
1.6.10 Released: **December 14, 2021** Release on GitHub	A bug fix release for Kotlin 1.6.0. Learn more about Kotlin 1.6.0.
1.6.0 Released: **November 16, 2021** Release on GitHub	A feature release with new language features, performance improvements, and evolutionary changes such as stabilizing experimental APIs. Learn more in: • Release blog post • What's new in Kotlin 1.6.0 • Compatibility guide
1.5.32 Released: **November 29, 2021** Release on GitHub	A bug fix release for Kotlin 1.5.31. Learn more about Kotlin 1.5.30.

1.5.31 Released: **September 20, 2021** Release on GitHub	A bug fix release for Kotlin 1.5.30. Learn more about Kotlin 1.5.30.
1.5.30 Released: **August 23, 2021** Release on GitHub	An incremental release with various improvements such as: • Instantiation of annotation classes on JVM • Improved opt-in requirement mechanism and type inference • Kotlin/JS IR backend in Beta • Support for Apple Silicon targets • Improved CocoaPods support • Gradle: Java toolchain support and improved daemon configuration Learn more in: • Release blog post • What's new in Kotlin 1.5.30
1.5.21 Released: **July 13, 2021** Release on GitHub	A bug fix release for Kotlin 1.5.20. Learn more about Kotlin 1.5.20.
1.5.20 Released: **June 24, 2021** Release on GitHub	An incremental release with various improvements such as: • String concatenation via `invokedynamic` on JVM by default • Improved support for Lombok and support for JSpecify • Kotlin/Native: KDoc export to Objective-C headers and faster `Array.copyInto()` inside one array • Gradle: caching of annotation processors' classloaders and support for the `--parallel` Gradle property • Aligned behavior of stdlib functions across platforms Learn more in: • Release blog post • What's new in Kotlin 1.5.20
1.5.10 Released: **May 24, 2021** Release on GitHub	A bug fix release for Kotlin 1.5.0. Learn more about Kotlin 1.5.0.

Released: **May 5, 2021** Release on GitHub	Learn more in: • Release blog post • What's new in Kotlin 1.5.0 • Compatibility guide
1.4.32 Released: **March 22, 2021** Release on GitHub	A bug fix release for Kotlin 1.4.30. Learn more about Kotlin 1.4.30.
1.4.31 Released: **February 25, 2021** Release on GitHub	A bug fix release for Kotlin 1.4.30 Learn more about Kotlin 1.4.30.
1.4.30 Released: **February 3, 2021** Release on GitHub	An incremental release with various improvements such as: • New JVM backend, now in Beta • Preview of new language features • Improved Kotlin/Native performance • Standard library API improvements Learn more in: • Release blog post • What's new in Kotlin 1.4.30
1.4.21 Released: **December 7, 2020** Release on GitHub	A bug fix release for Kotlin 1.4.20 Learn more about Kotlin 1.4.20.
1.4.20 Released: **November 23, 2020** Release on GitHub	An incremental release with various improvements such as: • Supporting new JVM features, like string concatenation via `invokedynamic` • Improved performance and exception handling for Kotlin Multiplatform Mobile projects • Extensions for JDK Path: `Path("dir") / "file.txt"` Learn more in: • Release blog post • What's new in Kotlin 1.4.20

	• What's new in Kotlin 1.4.20
1.4.10 发布于：2020-09-07 GitHub 上的版本发布	A bug fix release for Kotlin 1.4.0. Learn more about Kotlin 1.4.0.
1.4.0 发布于：2020-08-17 GitHub 上的版本发布	A feature release with many features and improvements that mostly focus on quality and performance. Learn more in: • Release blog post • What's new in Kotlin 1.4.0 • Compatibility guide • Migrating to Kotlin 1.4.0
1.3.72 发布于：2020-04-15 GitHub 上的版本发布	A bug fix release for Kotlin 1.3.70. Learn more about Kotlin 1.3.70.

基础

- 基本语法
- 习惯用法
- 例学 Kotlin↗
- 编码规范

基本语法

这是一组基本语法元素及示例。在每段的末尾都有一个指向相关主题详述的链接。

也可以通过 JetBrains 学院的免费 Kotlin 核心课程学习所有 Kotlin 要领。

包的定义与导入

包的声明应处于源文件顶部：

```
package my.demo

import kotlin.text.*

// ……
```

目录与包的结构无需匹配：源代码可以在文件系统的任意位置。

参见包。

程序入口点

Kotlin 应用程序的入口点是 `main` 函数：

```
fun main() {
    println("Hello world!")
}
```

`main` 的另一种形式接受可变数量的 `String` 参数：

```
fun main(args: Array<String>) {
    println(args.contentToString())
}
```

输出打到标准输出

`print` 将其参数打到标准输出：

```
fun main() {
//sampleStart
    print("Hello ")
    print("world!")
//sampleEnd
}
```

`println` 输出其参数并添加换行符，以便接下来输出的内容出现在下一行：

```
fun main() {
//sampleStart
    println("Hello world!")
    println(42)
//sampleEnd
}
```

Read from the standard input

The `readln()` function reads from the standard input. This function reads the entire line the user enters as a string.

You can use the `println()`, `readln()`, and `print()` functions together to print messages requesting and showing user input:

```
// Prints a message to request input
println("Enter any word: ")

// Reads and stores the user input. For example: Happiness
val yourWord = readln()

// Prints a message with the input
print("You entered the word: ")
print(yourWord)
// You entered the word: Happiness
```

For more information, see Read standard input.

函数

带有两个 `Int` 参数、返回 `Int` 的函数：

```
//sampleStart
fun sum(a: Int, b: Int): Int {
```

```
    return a + b
}
//sampleEnd

fun main() {
    print("sum of 3 and 5 is ")
    println(sum(3, 5))
}
```

函数体可以是表达式。其返回类型可以推断出来：

```
//sampleStart
fun sum(a: Int, b: Int) = a + b
//sampleEnd

fun main() {
    println("sum of 19 and 23 is ${sum(19, 23)}")
}
```

返回无意义的值的函数：

```
//sampleStart
fun printSum(a: Int, b: Int): Unit {
    println("sum of $a and $b is ${a + b}")
}
//sampleEnd

fun main() {
    printSum(-1, 8)
}
```

`Unit` 返回类型可以省略：

```
//sampleStart
fun printSum(a: Int, b: Int) {
    println("sum of $a and $b is ${a + b}")
}
//sampleEnd

fun main() {
    printSum(-1, 8)
}
```

参见函数。

变量

In Kotlin, you declare a variable starting with a keyword, `val` or `var`, followed by the name of the variable.

Use the `val` keyword to declare variables that are assigned a value only once. These are immutable, read-only local variables that can't be reassigned a different value after initialization:

```
fun main() {
//sampleStart
    // Declares the variable x and initializes it with the value of 5
    val x: Int = 5
    // 5
//sampleEnd
    println(x)
}
```

Use the `var` keyword to declare variables that can be reassigned. These are mutable variables, and you can change their values after initialization:

```
fun main() {
//sampleStart
    // Declares the variable x and initializes it with the value of 5
    var x: Int = 5
    // Reassigns a new value of 6 to the variable x
    x += 1
    // 6
//sampleEnd
    println(x)
}
```

Kotlin supports type inference and automatically identifies the data type of a declared variable. When declaring a variable, you can omit the type after the variable name:

```
fun main() {
//sampleStart
    // Declares the variable x with the value of 5;`Int` type is inferred
    val x = 5
    // 5
//sampleEnd
    println(x)
}
```

You can use variables only after initializing them. You can either initialize a variable at the moment of declaration or declare a variable first and initialize it later. In the second case, you must specify the data type:

```
fun main() {
```

```
//sampleStart
    // Initializes the variable x at the moment of declaration; type is not required
    val x = 5
    // Declares the variable c without initialization; type is required
    val c: Int
    // Initializes the variable c after declaration
    c = 3
    // 5
    // 3
//sampleEnd
    println(x)
    println(c)
}
```

可以在顶层声明变量：

```
//sampleStart
val PI = 3.14
var x = 0

fun incrementX() {
    x += 1
}
// x = 0; PI = 3.14
// incrementX()
// x = 1; PI = 3.14
//sampleEnd

fun main() {
    println("x = $x; PI = $PI")
    incrementX()
    println("incrementX()")
    println("x = $x; PI = $PI")
}
```

For information about declaring properties, see Properties.

创建类与实例

使用 `class` 关键字定义类：

```
class Shape
```

类的属性可以在其声明或主体中列出：

```
class Rectangle(val height: Double, val length: Double) {
```

```
    val perimeter = (height + length) * 2
}
```

具有类声明中所列参数的默认构造函数会自动可用：

```
class Rectangle(val height: Double, val length: Double) {
    val perimeter = (height + length) * 2
}
fun main() {
    val rectangle = Rectangle(5.0, 2.0)
    println("The perimeter is ${rectangle.perimeter}")
}
```

类之间继承由冒号（ `:` ）声明。默认情况下类都是 `final` 的；如需让一个类可继承， 请将其标记为 `open` ：

```
open class Shape

class Rectangle(val height: Double, val length: Double): Shape() {
    val perimeter = (height + length) * 2
}
```

For more information about constructors and inheritance, see Classes and Objects and instances.

注释

与大多数现代语言一样，Kotlin 支持单行（或行末）与多行（块）注释：

```
// 这是一个行注释

/* 这是一个多行的
   块注释。 */
```

Kotlin 中的块注释可以嵌套：

```
/* 注释从这里开始
/* 包含嵌套的注释 */
并且在这里结束。 */
```

参见编写 Kotlin 代码文档 查看关于文档注释语法的信息。

字符串模板

```
fun main() {
//sampleStart
    var a = 1
    // 模板中的简单名称:
    val s1 = "a is $a"

    a = 2
    // 模板中的任意表达式:
    val s2 = "${s1.replace("is", "was")}, but now is $a"
//sampleEnd
    println(s2)
}
```

参见字符串模板。

条件表达式

```
//sampleStart
fun maxOf(a: Int, b: Int): Int {
    if (a > b) {
        return a
    } else {
        return b
    }
}
//sampleEnd

fun main() {
    println("max of 0 and 42 is ${maxOf(0, 42)}")
}
```

在 Kotlin 中，`if` 也可以用作表达式：

```
//sampleStart
fun maxOf(a: Int, b: Int) = if (a > b) a else b
//sampleEnd

fun main() {
    println("max of 0 and 42 is ${maxOf(0, 42)}")
}
```

参见 `if` 表达式。

for 循环

```
fun main() {
//sampleStart
    val items = listOf("apple", "banana", "kiwifruit")
    for (item in items) {
        println(item)
    }
//sampleEnd
}
```

或者：

```
fun main() {
//sampleStart
    val items = listOf("apple", "banana", "kiwifruit")
    for (index in items.indices) {
        println("item at $index is ${items[index]}")
    }
//sampleEnd
}
```

参见 for 循环。

while 循环

```
fun main() {
//sampleStart
    val items = listOf("apple", "banana", "kiwifruit")
    var index = 0
    while (index < items.size) {
        println("item at $index is ${items[index]}")
        index++
    }
//sampleEnd
}
```

参见 while 循环。

when 表达式

```
//sampleStart
fun describe(obj: Any): String =
    when (obj) {
        1          -> "One"
        "Hello"    -> "Greeting"
```

```
        is Long    -> "Long"
        !is String -> "Not a string"
        else       -> "Unknown"
    }
//sampleEnd

fun main() {
    println(describe(1))
    println(describe("Hello"))
    println(describe(1000L))
    println(describe(2))
    println(describe("other"))
}
```

See when expressions and statements.

使用区间（range）

使用 `in` 操作符来检测某个数字是否在指定区间内：

```
fun main() {
//sampleStart
    val x = 10
    val y = 9
    if (x in 1..y+1) {
        println("fits in range")
    }
//sampleEnd
}
```

检测某个数字是否在指定区间外：

```
fun main() {
//sampleStart
    val list = listOf("a", "b", "c")

    if (-1 !in 0..list.lastIndex) {
        println("-1 is out of range")
    }
    if (list.size !in list.indices) {
        println("list size is out of valid list indices range, too")
    }
//sampleEnd
}
```

区间迭代：

```
fun main() {
//sampleStart
    for (x in 1..5) {
        print(x)
    }
//sampleEnd
}
```

或数列迭代：

```
fun main() {
//sampleStart
    for (x in 1..10 step 2) {
        print(x)
    }
    println()
    for (x in 9 downTo 0 step 3) {
        print(x)
    }
//sampleEnd
}
```

参见区间与数列。

集合

对集合进行迭代：

```
fun main() {
    val items = listOf("apple", "banana", "kiwifruit")
//sampleStart
    for (item in items) {
        println(item)
    }
//sampleEnd
}
```

使用 `in` 操作符来判断集合内是否包含某实例：

```
fun main() {
    val items = setOf("apple", "banana", "kiwifruit")
//sampleStart
    when {
        "orange" in items -> println("juicy")
        "apple" in items -> println("apple is fine too")
    }
```

```
//sampleEnd
}
```

使用 lambda 表达式来过滤（filter）与映射（map）集合：

```
fun main() {
//sampleStart
    val fruits = listOf("banana", "avocado", "apple", "kiwifruit")
    fruits
      .filter { it.startsWith("a") }
      .sortedBy { it }
      .map { it.uppercase() }
      .forEach { println(it) }
//sampleEnd
}
```

参见集合概述。

空值与空检测

当可能用 `null` 值时，必须将引用显式标记为可空。可空类型名称以问号（`?`）结尾。

如果 `str` 的内容不是数字返回 `null`：

```
fun parseInt(str: String): Int? {
    // ……
}
```

使用返回可空值的函数：

```
fun parseInt(str: String): Int? {
    return str.toIntOrNull()
}

//sampleStart
fun printProduct(arg1: String, arg2: String) {
    val x = parseInt(arg1)
    val y = parseInt(arg2)

    // 直接使用 `x * y` 会导致编译错误，因为它们可能为 null
    if (x != null && y != null) {
        // 在空检测后，x 与 y 会自动转换为非空值（non-nullable）
        println(x * y)
    }
    else {
        println("'$arg1' or '$arg2' is not a number")
```

```
    }
}
//sampleEnd

fun main() {
    printProduct("6", "7")
    printProduct("a", "7")
    printProduct("a", "b")
}
```

或者：

```
fun parseInt(str: String): Int? {
    return str.toIntOrNull()
}

fun printProduct(arg1: String, arg2: String) {
    val x = parseInt(arg1)
    val y = parseInt(arg2)

//sampleStart
    // ……
    if (x == null) {
        println("Wrong number format in arg1: '$arg1'")
        return
    }
    if (y == null) {
        println("Wrong number format in arg2: '$arg2'")
        return
    }

    // 在空检测后，x 与 y 会自动转换为非空值
    println(x * y)
//sampleEnd
}

fun main() {
    printProduct("6", "7")
    printProduct("a", "7")
    printProduct("99", "b")
}
```

参见空安全。

类型检测与自动类型转换

`is` 操作符检测一个表达式是否某类型的一个实例。 如果一个不可变的局部变量或属性已经判断出为某类型，那么检测后的分支中可以直接当作该类型使用，无需显式转换：

```
//sampleStart
fun getStringLength(obj: Any): Int? {
    if (obj is String) {
        // `obj` 在该条件分支内自动转换成 `String`
        return obj.length
    }

    // 在离开类型检测分支后，`obj` 仍然是 `Any` 类型
    return null
}
//sampleEnd

fun main() {
    fun printLength(obj: Any) {
        println("Getting the length of '$obj'. Result: ${getStringLength(obj) ?: "Erro
r: The object is not a string"} ")
    }
    printLength("Incomprehensibilities")
    printLength(1000)
    printLength(listOf(Any()))
}
```

或者：

```
//sampleStart
fun getStringLength(obj: Any): Int? {
    if (obj !is String) return null

    // `obj` 在这一分支自动转换为 `String`
    return obj.length
}
//sampleEnd

fun main() {
    fun printLength(obj: Any) {
        println("Getting the length of '$obj'. Result: ${getStringLength(obj) ?: "Erro
r: The object is not a string"} ")
    }
    printLength("Incomprehensibilities")
    printLength(1000)
    printLength(listOf(Any()))
}
```

甚至：

```
//sampleStart
fun getStringLength(obj: Any): Int? {
    // `obj` 在 `&&` 右边自动转换成 `String` 类型
    if (obj is String && obj.length > 0) {
```

```
        return obj.length
    }

    return null
}
//sampleEnd

fun main() {
    fun printLength(obj: Any) {
        println("Getting the length of '$obj'. Result: ${getStringLength(obj) ?: "Erro
r: The object is not a string"} ")
    }
    printLength("Incomprehensibilities")
    printLength("")
    printLength(1000)
}
```

参见类以及类型转换。

习惯用法

一些在 Kotlin 中广泛使用的语法习惯，如果你有更喜欢的语法习惯或者风格，建一个 pull request 贡献给我们吧！

创建 DTO（POJO/POCO）

```
data class Customer(val name: String, val email: String)
```

会为 `Customer` 类提供以下功能：

- 所有属性的 getter（对于 `var` 定义的还有 setter）
- `equals()`
- `hashCode()`
- `toString()`
- `copy()`
- 所有属性的 `component1()`、`component2()`……等等（参见数据类）

函数的默认实参

```
fun foo(a: Int = 0, b: String = "") { …… }
```

过滤 list

```
val positives = list.filter { x -> x > 0 }
```

或者可以更短:

```
val positives = list.filter { it > 0 }
```

了解 Java 与 Kotlin 过滤的区别。

检测元素是否存在于集合中

```
if ("john@example.com" in emailsList) { …… }

if ("jane@example.com" !in emailsList) { …… }
```

字符串内插

```
println("Name $name")
```

了解 Java 与 Kotlin 字符串连接的区别。

Read standard input safely

```
// Reads a string and returns null if the input can't be converted into an integer. Fo
r example: Hi there!
val wrongInt = readln().toIntOrNull()
println(wrongInt)
// null

// Reads a string that can be converted into an integer and returns an integer. For ex
ample: 13
val correctInt = readln().toIntOrNull()
println(correctInt)
// 13
```

For more information, see Read standard input.

类型判断

```
when (x) {
    is Foo -> ……
    is Bar -> ……
    else   -> ……
}
```

只读 list

```
val list = listOf("a", "b", "c")
```

只读 map

```
val map = mapOf("a" to 1, "b" to 2, "c" to 3)
```

访问 map 条目

```
println(map["key"])
map["key"] = value
```

遍历 map 或者 pair 型 list

```
for ((k, v) in map) {
    println("$k -> $v")
}
```

`k` 与 `v` 可以是任何适宜的名称，例如 `name` 与 `age`。

区间迭代

```
for (i in 1..100) { …… }  // 闭区间: 包含 100
for (i in 1..< 100) { …… } // 左闭右开区间: 不包含 100
for (x in 2..10 step 2) { …… }
for (x in 10 downTo 1) { …… }
(1..10).forEach { …… }
```

延迟属性

```
val p: String by lazy { // 该值仅在首次访问时计算
    // 计算该字符串
}
```

扩展函数

```
fun String.spaceToCamelCase() { …… }

"Convert this to camelcase".spaceToCamelCase()
```

创建单例

```
object Resource {
    val name = "Name"
}
```

Use inline value classes for type-safe values

```
@JvmInline
value class EmployeeId(private val id: String)

@JvmInline
value class CustomerId(private val id: String)
```

If you accidentally mix up `EmployeeId` and `CustomerId`, a compilation error is triggered.

> The `@JvmInline` annotation is only needed for JVM backends.
>
> 

实例化一个抽象类

```
abstract class MyAbstractClass {
    abstract fun doSomething()
    abstract fun sleep()
}

fun main() {
    val myObject = object : MyAbstractClass() {
        override fun doSomething() {
            // ……
        }

        override fun sleep() { // ……
        }
    }
    myObject.doSomething()
}
```

if-not-null 缩写

```
val files = File("Test").listFiles()

println(files?.size) // 如果 files 不是 null，那么输出其大小（size）
```

if-not-null-else 缩写

```
val files = File("Test").listFiles()

// For simple fallback values:
println(files?.size ?: "empty") // 如果 files 为 null，那么输出“empty”

// 如需在代码块中计算更复杂的备用值，请使用 `run`
val filesSize = files?.size ?: run {
    val someSize = getSomeSize()
    someSize * 2
}
println(filesSize)
```

if null 执行一个语句

```
val values = ……
val email = values["email"] ?: throw IllegalStateException("Email is missing!")
```

在可能会空的集合中取第一元素

```
val emails = …… // 可能会是空集合
val mainEmail = emails.firstOrNull() ?: ""
```

了解 Java 与 Kotlin 获取第一元素的差别。

if not null 执行代码

```
val value = ……

value?.let {
    …… // 如果非空会执行这个代码块
}
```

映射可空值（如果非空的话）

```
val value = ……

val mapped = value?.let { transformValue(it) } ?: defaultValue
// 如果该值或其转换结果为空，那么返回 defaultValue。
```

返回 when 表达式

```
fun transform(color: String): Int {
    return when (color) {
        "Red" -> 0
        "Green" -> 1
        "Blue" -> 2
        else -> throw IllegalArgumentException("Invalid color param value")
    }
}
```

try-catch 表达式

```
fun test() {
    val result = try {
        count()
    } catch (e: ArithmeticException) {
        throw IllegalStateException(e)
    }

    // 使用 result
}
```

if 表达式

```
val y = if (x == 1) {
    "one"
} else if (x == 2) {
    "two"
} else {
    "other"
}
```

返回类型为 Unit 的方法的构建器风格用法

```
fun arrayOfMinusOnes(size: Int): IntArray {
    return IntArray(size).apply { fill(-1) }
}
```

单表达式函数

```
fun theAnswer() = 42
```

等价于

```
fun theAnswer(): Int {
    return 42
}
```

单表达式函数与其它惯用法一起使用能简化代码，例如和 `when` 表达式一起使用：

```
fun transform(color: String): Int = when (color) {
    "Red" -> 0
    "Green" -> 1
    "Blue" -> 2
    else -> throw IllegalArgumentException("Invalid color param value")
}
```

对一个对象实例调用多个方法（with）

```
class Turtle {
    fun penDown()
    fun penUp()
    fun turn(degrees: Double)
    fun forward(pixels: Double)
}

val myTurtle = Turtle()
with(myTurtle) { // 画一个 100 像素的正方形
    penDown()
    for (i in 1..4) {
        forward(100.0)
        turn(90.0)
    }
    penUp()
}
```

配置对象的属性（apply）

```
val myRectangle = Rectangle().apply {
    length = 4
    breadth = 5
    color = 0xFAFAFA
}
```

这对于配置未出现在对象构造函数中的属性非常有用。

Java 7 的 try-with-resources

```
val stream = Files.newInputStream(Paths.get("/some/file.txt"))
stream.buffered().reader().use { reader ->
    println(reader.readText())
}
```

需要泛型信息的泛型函数

```
//  public final class Gson {
//      ……
//      public <T> T fromJson(JsonElement json, Class<T> classOfT) throws JsonSyntaxExc
eption {
//      ……

inline fun <reified T: Any> Gson.fromJson(json: JsonElement): T = this.fromJson(json,
T::class.java)
```

交换两个变量

```
var a = 1
var b = 2
a = b.also { b = a }
```

将代码标记为不完整（TODO）

Kotlin 的标准库有一个 `TODO()` 函数，该函数总是抛出一个 `NotImplementedError` 。 其返回类型为 `Nothing` ，因此无论预期类型是什么都可以使用它。 还有一个接受原因参数的重载：

```
fun calcTaxes(): BigDecimal = TODO("Waiting for feedback from accounting")
```

IntelliJ IDEA 的 kotlin 插件理解 `TODO()` 的语言，并且会自动在 TODO 工具窗口中添加代码指示。

下一步做什么？

- 使用地道 Kotlin 风格求解 Advent of Code 谜题。
- 了解如何在 Java 与 Kotlin 中处理字符串的典型任务。
- 了解如何在 Java 与 Kotlin 中处理集合的典型任务。
- 了解如何在 Java 与 Kotlin 中处理可空性。

例学 Kotlin

例学 Kotlin

编码规范

众所周知且易于遵循的编码规范对于任何编程语言都至关重要。 在这里，我们为使用 Kotlin 的项目提供关于代码风格与代码组织的准则。

在 IDE 中配置风格

两个最流行的 Kotlin IDE——IntelliJ IDEA 与 Android Studio 都为代码风格提供了强大的支持。 可以将它们配置为根据指定的代码风格自动格式化代码。

应用风格指南

1. 转到 **Settings/Preferences | Editor | Code Style | Kotlin**。
2. 点击 **Set from...**。
3. 选择 **Kotlin style guide** 。

验证代码是否遵循风格指南

1. 转到 **Settings/Preferences | Editor | Inspections | General**。
2. 打开 **Incorrect formatting** 探查项。 默认启用验证样式指南中描述的其他问题（例如命名约定）的附加探查项。

源代码组织

目录结构

在纯 Kotlin 项目中，推荐的目录结构遵循省略了公共根包的包结构。例如，如果项目中的所有代码都位于 `org.example.kotlin` 包及其子包中，那么 `org.example.kotlin` 包的文件应该直接放在源代码根目录下，而 `org.example.kotlin.network.socket` 中的文件应该放在源代码根目录下的 `network/socket` 子目录中。

> 对于 JVM 平台：Kotlin 源文件应当与 Java 源文件位于同一源文件根目录下， 并遵循相同的目录结构（每个文件应存储在与其 package 语句对应的目录中。
>
> 

源文件名称

如果 Kotlin 文件包含单个类或接口（以及可能相关的顶层声明），那么文件名应该与该类的名称相同，并追加 `.kt` 扩展名。 这适用于所有类型的类和接口。 如果文件包含多个类或只包含顶层声明， 那么选择一个描述该文件所包含内容的名称，并以此命名该文件。 使用首字母大写的驼峰风格，其中每个单词的首字母都大写。 例如 `ProcessDeclarations.kt`。

文件的名称应该描述文件中代码的作用。因此，应避免在文件名中使用诸如 `Util` 之类的无意义词语。

多平台项目

在多平台项目中，平台特有的源代码集中包含顶层声明的文件应具有与该源代码集名称相关联的后缀。 例如：

- **jvm**Main/kotlin/Platform.**jvm**.kt
- **android**Main/kotlin/Platform.**android**.kt
- **ios**Main/kotlin/Platform.**ios**.kt

对于公共源代码集，包含顶层声明的文件不应该有后缀。 例如， `commonMain/kotlin/Platform.kt`.

技术细节

由于 JVM 的限制〔不允许顶层成员（函数、属性）〕，我们建议在多平台项目中遵循这种文件命名方案。

为了解决 JVM 的这个问题，Kotlin JVM 编译器会创建包含顶层成员声明的包装类〔所谓的“文件门面”（file facades）〕。 文件门面有一个源自文件名的内部名称。

另外，JVM 不允许多个类具有相同的完整限定名称 (FQN，fully qualified name)。 这可能会导致 Kotlin 项目无法编译到 JVM 的情况：

```
root
|- commonMain/kotlin/myPackage/Platform.kt // 包含 'fun count() { }'
|- jvmMain/kotlin/myPackage/Platform.kt // 包含 'fun multiply() { }'
```

这里两个 `Platform.kt` 文件都在同一个包中，因此 Kotlin JVM 编译器生成两个文件门面，它们都有 FQN `myPackage.PlatformKt`。 这就会产生 "Duplicate JVM classes"（“重复的 JVM 类”）错误。

避免这种情况的最简单的方式就是按照上述指南重命名其中一个文件。 这种命名方案有助于避免冲突，同时保持代码的可读性。

> 对于以下两种场景，上述建议可能看起来多余，但我们仍然建议遵循之：
>
> - 非 JVM 平台不存在重复文件门面的问题。 但是，这种命名方案有助于保持文件命名

> 的一致性。
> - 在 JVM 中，如果源文件没有顶层声明就不会生成文件门面，也就不会遇到命名冲突。
>
> 但是，这种命名方案有助于避免简单的重构或添加可能包含顶层函数并同样导致 "Duplicate JVM classes"（“重复 JVM 类”错误）的情况。



源文件组织

鼓励多个声明（类、顶级函数或者属性）放在同一个 Kotlin 源文件中， 只要这些声明在语义上彼此紧密关联，并且文件保持合理大小（不超过几百行）。

特别是在为类定义与类的所有客户都相关的扩展函数时， 请将它们放在与类自身相同的地方。而在定义仅对指定客户有意义的扩展函数时，请将它们放在紧挨该客户代码之后。避免只是为了保存 某个类的所有扩展函数而创建文件。

类布局

一个类的内容应按以下顺序排列：

1. 属性声明与初始化块
2. 次构造函数
3. 方法声明
4. 伴生对象

不要按字母顺序或者可见性对方法声明排序，也不要将常规方法与扩展方法分开。而是要把相关的东西放在一起，这样从上到下阅读类的人就能够跟进所发生事情的逻辑。选择一个顺序（高级别优先，或者相反）并坚持下去。

将嵌套类放在紧挨使用这些类的代码之后。如果打算在外部使用嵌套类，而且类中并没有引用这些类，那么把它们放到末尾，在伴生对象之后。

接口实现布局

在实现一个接口时，实现成员的顺序应该与该接口的成员顺序相同（如果需要， 还要插入用于实现的额外的私有方法）。

重载布局

在类中总是将重载放在一起。

命名规则

在 Kotlin 中，包名与类名的命名规则非常简单：

- 包的名称总是小写且不使用下划线（`org.example.project`）。 通常不鼓励使用多个词的名称，但是如果确实需要使用多个词，可以将它们连接在一起或使用驼峰风格（`org.example.myProject`）。
- 类与对象的名称使用首字母大写的驼峰风格：

```
open class DeclarationProcessor { /*……*/ }

object EmptyDeclarationProcessor : DeclarationProcessor() { /*……*/ }
```

函数名

函数、属性与局部变量的名称以小写字母开头、使用驼峰风格而不使用下划线：

```
fun processDeclarations() { /*……*/ }
var declarationCount = 1
```

例外：用于创建类实例的工厂函数可以与抽象返回类型具有相同的名称：

```
interface Foo { /*……*/ }

class FooImpl : Foo { /*……*/ }

fun Foo(): Foo { return FooImpl() }
```

测试方法的名称

当且仅当在测试中，可以使用反引号括起来的带空格的方法名。 请注意，API 级别在 30 及以上的 Android 运行时才支持这样的方法名。 测试代码中也允许方法名使用下划线。

```
class MyTestCase {
    @Test fun `ensure everything works`() { /*……*/ }

    @Test fun ensureEverythingWorks_onAndroid() { /*……*/ }
}
```

属性名

常量名称（标有 `const` 的属性，或者保存不可变数据的没有自定义 `get` 函数的顶层/对象 `val` 属性）应该使用全大写、下划线分隔的名称，遵循 screaming snake case 约定：

```
const val MAX_COUNT = 8
val USER_NAME_FIELD = "UserName"
```

保存带有行为的对象或者可变数据的顶层/对象属性的名称应该使用驼峰风格名称：

```
val mutableCollection: MutableSet<String> = HashSet()
```

保存单例对象引用的属性的名称可以使用与 `object` 声明相同的命名风格：

```
val PersonComparator: Comparator<Person> = /*……*/
```

对于枚举常量，可以使用全大写、下划线分隔的 (screaming snake case) 名称（`enum class Color { RED, GREEN }`）也可使用首字母大写的常规驼峰名称，具体取决于用途。

幕后属性的名称

如果一个类有两个概念上相同的属性，一个是公共 API 的一部分，另一个是实现细节，那么使用下划线作为私有属性名称的前缀：

```
class C {
    private val _elementList = mutableListOf<Element>()

    val elementList: List<Element>
         get() = _elementList
}
```

选择好名称

类的名称通常是用来解释类是什么的名词或者名词短语：`List`、`PersonReader`。

方法的名称通常是动词或动词短语，说明该方法做什么：`close`、`readPersons`。修改对象或者返回一个新对象的名称也应遵循建议。例如 `sort` 是对一个集合就地排序，而 `sorted` 是返回一个排序后的集合副本。

名称应该表明实体的目的是什么，所以最好避免在名称中使用无意义的单词（`Manager`、`Wrapper`）。

当使用首字母缩写作为声明名称的一部分时，请遵循以下规则：

- 对于两个字母的缩写，两字母都大写。例如，`IOStream`。

- 对于超过两个字母的缩写，只大写第一个字母。例如，`XmlFormatter` 或 `HttpInputStream`。

格式化

缩进

使用 4 个空格缩进。不要使用 tab。

对于花括号，将左花括号放在结构起始处的行尾，而将右花括号放在与左括结构横向对齐的单独一行。

```
if (elements != null) {
    for (element in elements) {
        // ……
    }
}
```

> 在 Kotlin 中，分号是可选的，因此换行很重要。语言设计采用 Java 风格的花括号格式，如果尝试使用不同的格式化风格，那么可能会遇到意外的行为。



横向空白

- 在二元操作符左右留空格（`a + b`）。例外：不要在“range to”操作符（`0..i`）左右留空格。
- 不要在一元运算符左右留空格（`a++`）。
- 在控制流关键字（`if`、`when`、`for` 以及 `while`）与相应的左括号之间留空格。
- 不要在主构造函数声明、方法声明或者方法调用的左括号之前留空格。

```
class A(val x: Int)

fun foo(x: Int) { …… }

fun bar() {
    foo(1)
}
```

- 绝不在 `(`、`[` 之后或者 `]`、`)` 之前留空格。
- 绝不在 `.` 或者 `?.` 左右留空格：`foo.bar().filter { it > 2 }.joinToString()`，`foo?.bar()`。
- 在 `//` 之后留一个空格：`// 这是一条注释`。

- 不要在用于指定类型参数的尖括号前后留空格：`class Map<K, V> { …… }`。
- 不要在 `::` 前后留空格：`Foo::class`、`String::length`。
- 不要在用于标记可空类型的 `?` 前留空格：`String?`。

作为一般规则，避免任何类型的水平对齐。将标识符重命名为不同长度的名称不应该影响声明或者任何用法的格式。

冒号

在以下场景中的 `:` 之前留一个空格：

- 当它用于分隔类型与超类型时。
- 当委托给一个超类的构造函数或者同一类的另一个构造函数时。
- 在 `object` 关键字之后。

而当分隔声明与其类型时，不要在 `:` 之前留空格。

在 `:` 之后总要留一个空格。

```
abstract class Foo<out T : Any> : IFoo {
    abstract fun foo(a: Int): T
}

class FooImpl : Foo() {
    constructor(x: String) : this(x) { /*……*/ }

    val x = object : IFoo { /*……*/ }
}
```

类头

具有少数主构造函数参数的类可以写成一行：

```
class Person(id: Int, name: String)
```

具有较长类头的类应该格式化，以使每个主构造函数参数都在带有缩进的独立的行中。 另外，右括号应该位于一个新行上。如果使用了继承，那么超类的构造函数调用或者所实现接口的列表应该与右括号位于同一行：

```
class Person(
    id: Int,
    name: String,
    surname: String
) : Human(id, name) { /*……*/ }
```

对于多个接口，应该将超类构造函数调用放在首位，然后将每个接口应放在不同的行中：

```
class Person(
    id: Int,
    name: String,
    surname: String
) : Human(id, name),
    KotlinMaker { /*……*/ }
```

对于具有很长超类型列表的类，在冒号后面换行，并横向对齐所有超类型名：

```
class MyFavouriteVeryLongClassHolder :
    MyLongHolder<MyFavouriteVeryLongClass>(),
    SomeOtherInterface,
    AndAnotherOne {

    fun foo() { /*……*/ }
}
```

为了将类头与类体分隔清楚，当类头很长时，可以在类头后放一空行（如上例所示）或者将左花括号放在独立行上：

```
class MyFavouriteVeryLongClassHolder :
    MyLongHolder<MyFavouriteVeryLongClass>(),
    SomeOtherInterface,
    AndAnotherOne
{
    fun foo() { /*……*/ }
}
```

构造函数参数使用常规缩进（4 个空格）。这确保了在主构造函数中声明的属性与 在类体中声明的属性具有相同的缩进。

修饰符

如果一个声明有多个修饰符，请始终按照以下顺序安放：

```
public / protected / private / internal
expect / actual
final / open / abstract / sealed / const
external
override
lateinit
tailrec
vararg
suspend
```

```
inner
enum / annotation / fun // 在 `fun interface` 中是修饰符
companion
inline/ value
infix
operator
data
```

将所有注解放在修饰符前：

```
@Named("Foo")
private val foo: Foo
```

除非你在编写库，否则请省略多余的修饰符（例如 `public`）。

注解

通常将注解放在单独的行上，在它们所依附的声明之前，并使用相同的缩进：

```
@Target(AnnotationTarget.PROPERTY)
annotation class JsonExclude
```

无参数的注解可以放在同一行：

```
@JsonExclude @JvmField
var x: String
```

无参数的单个注解可以与相应的声明放在同一行：

```
@Test fun foo() { /*……*/ }
```

文件注解

文件注解位于文件注释（如果有的话）之后、 `package` 语句之前， 并且用一个空白行与 `package` 分开（为了强调其针对文件而不是包）。

```
/** 授权许可、版权以及任何其他内容 */
@file:JvmName("FooBar")

package foo.bar
```

函数

如果函数签名不适合单行，请使用以下语法：

```
fun longMethodName(
    argument: ArgumentType = defaultValue,
    argument2: AnotherArgumentType,
): ReturnType {
    // 函数体
}
```

函数参数使用常规缩进（4 个空格）。有助于确保与构造函数参数一致

对于由单个表达式构成的函数体，优先使用表达式形式。

```
fun foo(): Int {     // 不良
    return 1
}

fun foo() = 1        // 良好
```

表达式函数体格式化

如果函数的表达式函数体与函数声明不适合放在同一行，那么将 `=` 留在第一， 并将表达式函数体缩进 4 个空格。

```
fun f(x: String, y: String, z: String) =
    veryLongFunctionCallWithManyWords(andLongParametersToo(), x, y, z)
```

属性格式化

对于非常简单的只读属性，请考虑单行格式：

```
val isEmpty: Boolean get() = size == 0
```

对于更复杂的属性，总是将 `get` 与 `set` 关键字放在不同的行上：

```
val foo: String
    get() { /*……*/ }
```

对于具有初始化器的属性，如果初始化器很长，那么在 `=` 号后增加一个换行并将初始化器缩进四个空格：

```
private val defaultCharset: Charset? =
    EncodingRegistry.getInstance().getDefaultCharsetForPropertiesFiles(file)
```

控制流语句

如果 `if` 或 `when` 语句的条件有多行，那么在语句体外边总是使用大括号。 将该条件的每个后续行相对于条件语句起始处缩进 4 个空格。 将该条件的右圆括号与左花括号放在单独一行：

```
if (!component.isSyncing &&
    !hasAnyKotlinRuntimeInScope(module)
) {
    return createKotlinNotConfiguredPanel(module)
}
```

这有助于对齐条件与语句体。

将 `else`、 `catch`、 `finally` 关键字以及 `do-while` 循环的 `while` 关键字与之前的花括号放在相同的行上：

```
if (condition) {
    // 主体
} else {
    // else 部分
}

try {
    // 主体
} finally {
    // 清理
}
```

在 `when` 语句中，如果一个分支不止一行，可以考虑用空行将其与相邻的分支块分开：

```
private fun parsePropertyValue(propName: String, token: Token) {
    when (token) {
        is Token.ValueToken ->
            callback.visitValue(propName, token.value)

        Token.LBRACE -> { // ……
        }
    }
}
```

将短分支放在与条件相同的行上，无需花括号。

```
when (foo) {
    true -> bar() // 良好
```

```
    false -> { baz() } // 不良
}
```

方法调用格式化

在较长参数列表的左括号后添加一个换行符。按 4 个空格缩进参数。 将密切相关的多个参数分在同一行。

```
drawSquare(
    x = 10, y = 10,
    width = 100, height = 100,
    fill = true
)
```

在分隔参数名与值的 `=` 左右留空格。

链式调用换行

当对链式调用换行时，将 `.` 字符或者 `?.` 操作符放在下一行，并带有单倍缩进：

```
val anchor = owner
    ?.firstChild!!
    .siblings(forward = true)
    .dropWhile { it is PsiComment || it is PsiWhiteSpace }
```

调用链的第一个调用通常在换行之前，当然如果能让代码更有意义也可以忽略这点。

Lambda 表达式

在 lambda 表达式中，应该在花括号左右以及分隔参数与代码体的箭头左右留空格。 如果一个调用接受单个 lambda 表达式，尽可能将其放在圆括号外边传入。

```
list.filter { it > 10 }
```

如果为 lambda 表达式分配一个标签，那么不要在该标签与左花括号之间留空格：

```
fun foo() {
    ints.forEach lit@{
        // ……
    }
}
```

在多行的 lambda 表达式中声明参数名时，将参数名放在第一行，后跟箭头与换行符：

```
appendCommaSeparated(properties) { prop ->
    val propertyValue = prop.get(obj)  // ……
}
```

如果参数列表太长而无法放在一行上，请将箭头放在单独一行：

```
foo {
   context: Context,
   environment: Env
   ->
   context.configureEnv(environment)
}
```

尾部逗号

尾部逗号是一系列元素中最后一项之后的逗号符号：

```
class Person(
    val firstName: String,
    val lastName: String,
    val age: Int, // 尾部逗号
)
```

使用尾部逗号有以下几点收益：

- 使版本控制差异更清晰——因为所有焦点都集中在变更的值上。
- 使添加及重新排序元素更容易——操作元素时无需再添加或删除逗号。
- 简化了代码生成逻辑（例如对象初始化器）。 最后一个元素也可以有逗号。

尾部逗号完全是可选的——没有它们代码仍可以照常工作。 Kotlin 风格指南鼓励在声明处使用尾部逗号，而自行决定在调用处是否使用。

如需在 IntelliJ IDEA 格式化程序中启用尾部逗号，请转到 **Settings/Preferences | Editor | Code Style | Kotlin**， 打开 **Other** 选项卡并选择 **Use trailing comma** 选项。

枚举

```
enum class Direction {
    NORTH,
    SOUTH,
    WEST,
    EAST, // 尾部逗号
}
```

值实参

```
fun shift(x: Int, y: Int) { /*……*/ }
shift(
    25,
    20, // 尾部逗号
)
val colors = listOf(
    "red",
    "green",
    "blue", // 尾部逗号
)
```

类属性与形参

```
class Customer(
    val name: String,
    val lastName: String, // 尾部逗号
)
class Customer(
    val name: String,
    lastName: String, // 尾部逗号
)
```

函数值形参

```
fun powerOf(
    number: Int,
    exponent: Int, // 尾部逗号
) { /*……*/ }
constructor(
    x: Comparable<Number>,
    y: Iterable<Number>, // 尾部逗号
) {}
fun print(
    vararg quantity: Int,
    description: String, // 尾部逗号
) {}
```

类型可选的形参（包括 setter）

```
val sum: (Int, Int, Int) -> Int = fun(
    x,
```

```
    y,
    z, // 尾部逗号
): Int {
    return x + y + x
}
println(sum(8, 8, 8))
```

索引后缀

```
class Surface {
    operator fun get(x: Int, y: Int) = 2 * x + 4 * y - 10
}
fun getZValue(mySurface: Surface, xValue: Int, yValue: Int) =
    mySurface[
        xValue,
        yValue, // 尾部逗号
    ]
```

lambda 表达式形参

```
fun main() {
    val x = {
            x: Comparable<Number>,
            y: Iterable<Number>, // 尾部逗号
        ->
        println("1")
    }
    println(x)
}
```

when 条目

```
fun isReferenceApplicable(myReference: KClass<*>) = when (myReference) {
    Comparable::class,
    Iterable::class,
    String::class, // 尾部逗号
        -> true
    else -> false
}
```

（注解中的）集合字面值

```
annotation class ApplicableFor(val services: Array<String>)
@ApplicableFor([
```

```
    "serializer",
    "balancer",
    "database",
    "inMemoryCache", // 尾部逗号
])
fun run() {}
```

类型实参

```
fun <T1, T2> foo() {}
fun main() {
    foo<
            Comparable<Number>,
            Iterable<Number>, // 尾部逗号
            >()
}
```

类型形参

```
class MyMap<
        MyKey,
        MyValue, // 尾部逗号
        > {}
```

解构声明

```
data class Car(val manufacturer: String, val model: String, val year: Int)
val myCar = Car("Tesla", "Y", 2019)
val (
    manufacturer,
    model,
    year, // 尾部逗号
) = myCar
val cars = listOf<Car>()
fun printMeanValue() {
    var meanValue: Int = 0
    for ((
        _,
        _,
        year, // 尾部逗号
    ) in cars) {
        meanValue += year
    }
    println(meanValue/cars.size)
}
printMeanValue()
```

文档注释

对于较长的文档注释，将开头 `/**` 放在一个独立行中，并且后续每行都以星号开头：

```
/**
 * 这是一条多行
 * 文档注释。
 */
```

简短注释可以放在一行内：

```
/** 这是一条简短文档注释。 */
```

通常，避免使用 `@param` 与 `@return` 标记。而是将参数与返回值的描述直接合并到文档注释中，并在提到参数的任何地方加上参数链接。 只有当需要不适合放进主文本流程的冗长描述时才应使用 `@param` 与 `@return` 。

```
// 避免这样:

/**
 * Returns the absolute value of the given number.
 * @param number The number to return the absolute value for.
 * @return The absolute value.
 */
fun abs(number: Int): Int { /*……*/ }

// 而要这样:

/**
 * Returns the absolute value of the given [number].
 */
fun abs(number: Int): Int { /*……*/ }
```

避免重复结构

一般来说，如果 Kotlin 中的某种语法结构是可选的并且被 IDE 高亮为冗余的，那么应该在代码中省略之。为了清楚起见，不要在代码中保留不必要的语法元素 。

Unit 返回类型

如果函数返回 Unit，那么应该省略返回类型：

```
fun foo() { // 这里省略了“: Unit”

}
```

分号

尽可能省略分号。

字符串模版

将简单变量传入到字符串模版中时不要使用花括号。只有用到更长表达式时才使用花括号。

```
println("$name has ${children.size} children")
```

语言特性的惯用法

不可变性

优先使用不可变（而不是可变）数据。初始化后未修改的局部变量与属性，总是将其声明为 `val` 而不是 `var` 。

总是使用不可变集合接口（ `Collection` , `List` , `Set` , `Map` ）来声明无需改变的集合。使用工厂函数创建集合实例时，尽可能选用返回不可变集合类型的函数：

```
// 不良: 使用可变集合类型作为无需改变的值
fun validateValue(actualValue: String, allowedValues: HashSet<String>) { …… }

// 良好: 使用不可变集合类型
fun validateValue(actualValue: String, allowedValues: Set<String>) { …… }

// 不良: arrayListOf() 返回 ArrayList<T>，这是一个可变集合类型
val allowedValues = arrayListOf("a", "b", "c")

// 良好: listOf() 返回 List<T>
val allowedValues = listOf("a", "b", "c")
```

默认实参值

优先声明带有默认实参的函数而不是声明重载函数。

```
// 不良
fun foo() = foo("a")
```

```
fun foo(a: String) { /*……*/ }

// 良好
fun foo(a: String = "a") { /*……*/ }
```

类型别名

如果有一个在代码库中多次用到的函数类型或者带有类型参数的类型，那么最好为它定义一个类型别名：

```
typealias MouseClickHandler = (Any, MouseEvent) -> Unit
typealias PersonIndex = Map<String, Person>
```

如果使用一个 private 或者 internal 的类型别名来避免名称冲突，请优先使用包与导入中提到的 `import …… as ……`。

Lambda 表达式参数

在简短、非嵌套的 lambda 表达式中建议使用 `it` 用法而不是显式声明参数。而在有参数的嵌套 lambda 表达式中，始终显式声明参数。

在 lambda 表达式中返回

避免在 lambda 表达式中使用多个返回到标签。请考虑重新组织这样的 lambda 表达式使其只有单一退出点。 如果这无法做到或者不够清晰，请考虑将 lambda 表达式转换为匿名函数。

不要在 lambda 表达式的最后一条语句中使用返回到标签。

具名实参

当一个方法接受多个相同的原生类型参数或者多个 `Boolean` 类型参数时，请使用具名实参语法， 除非在上下文中的所有参数的含义都已绝对清楚。

```
drawSquare(x = 10, y = 10, width = 100, height = 100, fill = true)
```

条件语句

优先使用 `try`、 `if` 与 `when` 的表达形式。

```
return if (x) foo() else bar()
```

```
return when(x) {
    0 -> "zero"
    else -> "nonzero"
}
```

优先选用上述代码而不是：

```
if (x)
    return foo()
else
    return bar()
```

```
when(x) {
    0 -> return "zero"
    else -> return "nonzero"
}
```

if 还是 when

二元条件优先使用 `if` 而不是 `when`。 例如，使用 `if` 的这种语法：

```
if (x == null) …… else ……
```

而不是使用 `when` 的这种：

```
when (x) {
    null -> // ……
    else -> // ……
}
```

如果有三个或多个选项时优先使用 `when`。

Guard conditions in when expression

Use parentheses when combining multiple boolean expressions in `when` expressions or statements with guard conditions:

```
when (status) {
    is Status.Ok if (status.info.isEmpty() || status.info.id == null) -> "no informati
on"
}
```

Instead of:

```
when (status) {
    is Status.Ok if status.info.isEmpty() || status.info.id == null -> "no information"

}
```

在条件中的可空的布尔值

如果需要在条件语句中用到可空的 `Boolean`，使用 `if (value == true)` 或 `if (value == false)` 检测。

循环

优先使用高阶函数（`filter`、`map` 等）而不是循环。例外：`forEach`（优先使用常规的 `for` 循环，除非 `forEach` 的接收者是可空的或者 `forEach` 用做长调用链的一部分。）

当在使用多个高阶函数的复杂表达式与循环之间进行选择时，请了解每种情况下所执行操作的开销并且记得考虑性能因素。

区间上循环

使用 `..<` 操作符在一个左闭右开区间上循环：

```
for (i in 0..n - 1) { /*……*/ }  // 不良
for (i in 0..<n) { /*……*/ }  // 良好
```

字符串

优先使用字符串模板而不是字符串拼接。

优先使用多行字符串而不是将 `\n` 转义序列嵌入到常规字符串字面值中。

如需在多行字符串中维护缩进，当生成的字符串不需要任何内部缩进时使用 `trimIndent`，而需要内部缩进时使用 `trimMargin`：

```
fun main() {
//sampleStart
   println("""
    Not
    trimmed
    text
    """
```

```
    )

    println("""
     Trimmed
     text
     """.trimIndent()
    )

    println()

    val a = """Trimmed to margin text:
|if(a > 1) {
|    return a
|}""".trimMargin()

    println(a)
//sampleEnd
}
```

了解 Java 与 Kotlin 多行字符串的区别。

函数还是属性

在某些场景中，不带参数的函数可与只读属性互换。 虽然语义相似，但是在某种程度上有一些风格上的约定。

底层算法优先使用属性而不是函数：

- 不会抛异常。
- 计算开销小（或者在首次运行时缓存）。
- 如果对象状态没有改变，那么多次调用都会返回相同结果。

扩展函数

放手去用扩展函数。每当你有一个主要用于某个对象的函数时，可以考虑使其成为一个以该对象为接收者的扩展函数。为了尽量减少 API 污染，尽可能地限制扩展函数的可见性。根据需要，使用局部扩展函数、成员扩展函数或者具有私有可视性的顶层扩展函数。

中缀函数

一个函数只有用于两个角色类似的对象时才将其声明为中缀（`infix`）函数。良好示例如：`and`、`to`、`zip`。 不良示例如：`add`。

如果一个方法会改动其接收者，那么不要声明为中缀（`infix`）形式。

工厂函数

如果为一个类声明一个工厂函数，那么不要让它与类自身同名。优先使用独特的名称， 该名称能表明为何该工厂函数的行为与众不同。只有当确实没有特殊的语义时， 才可以使用与该类相同的名称。

```
class Point(val x: Double, val y: Double) {
    companion object {
        fun fromPolar(angle: Double, radius: Double) = Point(……)
    }
}
```

如果一个对象有多个重载的构造函数，它们并非调用不同的超类构造函数，并且不能简化为具有默认实参值的单个构造函数，那么优先用工厂函数取代这些重载的构造函数。

平台类型

返回平台类型表达式的公有函数/方法必须显式声明其 Kotlin 类型：

```
fun apiCall(): String = MyJavaApi.getProperty("name")
```

任何使用平台类型表达式初始化的属性（包级别或类级别）必须显式声明其 Kotlin 类型：

```
class Person {
    val name: String = MyJavaApi.getProperty("name")
}
```

使用平台类型表达式初始化的局部值可以有也可以没有类型声明：

```
fun main() {
    val name = MyJavaApi.getProperty("name")
    println(name)
}
```

作用域函数 apply/with/run/also/let

Kotlin 提供了一系列用来在给定对象上下文中执行代码块的函数：`let`、`run`、`with`、`apply` 以及 `also`。 关于不同情况下选择正确作用域函数的准则，请参考作用域函数。

库的编码规范

在编写库时，建议遵循一组额外的规则以确保 API 的稳定性：

- 总是显式指定成员的可见性（以避免将声明意外暴露为公有 API ）。

- 总是显式指定函数返回类型以及属性类型（以避免当实现改变时意外更改返回类型）。
- 为所有公有成员提供 KDoc 注释，不需要任何新文档的覆盖成员除外（以支持为该库生成文档）。

在库作者指南中详细了解为库编写 API 时需要考虑的最佳实践与意见。

概念

- 类型
  - 基本类型
    - 概述
    - 数字
    - 无符号整型
    - 布尔
    - 字符
    - 字符串
    - 数组
  - 类型检测与类型转换
- 控制流程
  - 条件与循环
  - 返回与跳转
  - 异常
- 包与导入
- 类与对象
  - 类
  - 继承
  - 属性
  - 接口
  - 函数式（SAM）接口
  - 可见性修饰符
  - 扩展
  - 数据类
  - 密封类与密封接口
  - 泛型：in、out、where
  - 嵌套类
  - 枚举类
  - 内联类
  - 对象表达式与对象声明
  - 委托
  - 属性委托
  - 类型别名
- 函数
  - 函数
  - lambda 表达式

    - 内联函数
    - 操作符重载
    - 构建器
        - 类型安全的构建器
        - 使用带有构建器类型推断的构建器
- 空安全
- 相等性
- this 表达式
- 异步程序设计技术
- 协程
- 注解
- 解构声明
- 反射

类型

- 基本类型
  - 概述
  - 数字
  - 无符号整型
  - 布尔
  - 字符
  - 字符串
  - 数组
- 类型检测与类型转换

基本类型

- 概述
- 数字
- 无符号整型
- 布尔
- 字符
- 字符串
- 数组

基本类型

在 Kotlin 中，所有东西都是对象，在这个意义上讲可以在任何变量上调用成员函数与属性。
While certain types have an optimized internal representation as primitive values at runtime (such as numbers, characters, booleans and others), they appear and behave like regular classes to you.

本节会描述 Kotlin 中使用的基本类型：

- 数字及其无符号版
- 布尔
- 字符
- 字符串
- 数组

> Learn how to perform type checks and casts in Kotlin.
>
> 

To learn about other Kotlin types, such as `Nothing`, `Any`, and `Unit`, look through the Kotlin API reference:

- `Any`
- `Nothing`
- `Unit`

数字

整数类型

Kotlin 提供了一组表示数字的内置类型。 对于整数，有四种不同大小与取值范围的类型：

类型	大小（比特数）	最小值	最大值
`Byte`	8	-128	127
`Short`	16	-32768	32767
`Int`	32	-2,147,483,648 (-2^{31})	2,147,483,647 ($2^{31} - 1$)
`Long`	64	-9,223,372,036,854,775,808 (-2^{63})	9,223,372,036,854,775,807 ($2^{63} - 1$)

> In addition to signed integer types, Kotlin also provides unsigned integer types. As unsigned integers are aimed at a different set of use cases, they are covered separately. See .



当初始化一个没有显式指定类型的变量时，编译器会自动推断为自 `Int` 起足以表示该值的最小类型。 如果不超过 `Int` 的取值范围，那么类型是 `Int`。 如果超出了该范围，那么类型是 `Long`。 如需显式指定 `Long` 值，请给该值追加后缀 `L`。 To use the `Byte` or `Short` type, specify it explicitly in the declaration. 显式指定类型会触发编译器检测该值不会超出指定类型的取值范围。

```
val one = 1 // Int
val threeBillion = 3000000000 // Long
val oneLong = 1L // Long
val oneByte: Byte = 1
```

浮点类型

对于实数，Kotlin 提供了浮点类型 `Float` 与 `Double` 类型，遵循 IEEE 754 标准。 `Float` 表达 IEEE 754 单精度，而 `Double` 表达双精度。

这两个类型的大小不同，并为两种不同精度的浮点数提供存储：

类型	大小（比特数）	有效数字比特数	指数比特数	十进制位数
`Float`	32	24	8	6-7

Double	64	53	11	15-16

可以仅使用带小数部分的数字初始化 `Double` 与 `Float` 变量。 小数部分与整数部分之间用句点（ `.` ）分隔

对于以小数初始化的变量，编译器会推断为 `Double` 类型：

```
val pi = 3.14          // Double

val one: Double = 1    // Int is inferred
// 初始化类型不匹配

val oneDouble = 1.0    // Double
```

如需将一个值显式指定为 `Float` 类型，请添加 `f` 或 `F` 后缀。 如果以这种方式提供的值包含多于 7 位十进制数，那么会将其舍入：

```
val e = 2.7182818284          // Double
val eFloat = 2.7182818284f    // Float, 实际值为 2.7182817
```

与一些其他语言不同，Kotlin 中的数字没有隐式拓宽转换。 例如，具有 `Double` 参数的函数只能对 `Double` 值调用，而不能对 `Float` 、 `Int` 或者其他数字值调用：

```
fun main() {
//sampleStart
    fun printDouble(x: Double) { print(x) }

    val x = 1.0
    val xInt = 1
    val xFloat = 1.0f

    printDouble(x)

    printDouble(xInt)
    // 参数类型不匹配

    printDouble(xFloat)
    // 参数类型不匹配
//sampleEnd
}
```

如需将数值转换为不同的类型，请使用显式转换。

数字字面常量

数值常量字面值有以下几种:

- 十进制：`123`
- Long 类型，以大写 `L` 结尾：`123L`
- 十六进制：`0x0F`
- 二进制：`0b00001011`

> Kotlin 不支持八进制。
>
> 

Kotlin 同样支持浮点数的常规表示方法:

- Doubles (default when the fractional part does not end with a letter): `123.5` , `123.5e10`
- Float，以字母 `f` 或 `F` 结尾：`123.5f`

你可以使用下划线使数字常量更易读：

```
val oneMillion = 1_000_000
val creditCardNumber = 1234_5678_9012_3456L
val socialSecurityNumber = 999_99_9999L
val hexBytes = 0xFF_EC_DE_5E
val bytes = 0b11010010_01101001_10010100_10010010
val bigFractional = 1_234_567.7182818284
```

> 无符号整数字面值也有特殊后缀。
>
> 更多内容请参阅无符号整型字面值。
>
> 

Boxing and caching numbers on the Java Virtual Machine

The way the JVM stores numbers can make your code behave counterintuitively because of the cache used by default for small (byte-sized) numbers.

The JVM stores numbers as primitive types: `int` , `double` , and so on. When you use generic types or create a nullable number reference such as `Int?` , numbers are boxed in Java classes such as `Integer` or `Double` .

The JVM applies a memory optimization technique to `Integer` and other objects that represent numbers between `-128` and `127`. All nullable references to such objects refer to the same cached object. For example, nullable objects in the following code are referentially equal:

```
fun main() {
//sampleStart
    val a: Int = 100
    val boxedA: Int? = a
    val anotherBoxedA: Int? = a

    println(boxedA === anotherBoxedA) // true
//sampleEnd
}
```

For numbers outside this range, the nullable objects are different but structurally equal:

```
fun main() {
//sampleStart
    val b: Int = 10000
    val boxedB: Int? = b
    val anotherBoxedB: Int? = b

    println(boxedB === anotherBoxedB) // false
    println(boxedB == anotherBoxedB) // true
//sampleEnd
}
```

For this reason, Kotlin warns about using referential equality with boxable numbers and literals with the following message: `"Identity equality for arguments of types ... and ... is prohibited."` When comparing `Int`, `Short`, `Long`, and `Byte` types (as well as `Char` and `Boolean`), use structural equality checks to get consistent results.

显式数字转换

由于不同的表示方式，数字类型并不是彼此的子类型。 As a consequence, smaller types are *not* implicitly converted to bigger types and vice versa. For example, assigning a value of type `Byte` to an `Int` variable requires an explicit conversion:

```
fun main() {
//sampleStart
    val byte: Byte = 1
    // OK, 字面值会静态检测

    val intAssignedByte: Int = byte
    // Initializer type mismatch
```

```
    val intConvertedByte: Int = byte.toInt()

    println(intConvertedByte)
//sampleEnd
}
```

所有数字类型都支持转换为其他类型：

- `toByte(): Byte` (deprecated for Float and Double)
- `toShort(): Short`
- `toInt(): Int`
- `toLong(): Long`
- `toFloat(): Float`
- `toDouble(): Double`

很多情况都不需要显式类型转换，因为类型会从上下文推断出来， 而算术运算符会有重载自动处理类型转换。例如：

```
fun main() {
//sampleStart
    val l = 1L + 3       // Long + Int => Long
    println(l is Long)   // true
//sampleEnd
}
```

Reasoning against implicit conversions

Kotlin doesn't support implicit conversions because they can lead to unexpected behavior.

If numbers of different types were converted implicitly, we could sometimes lose equality and identity silently. For example, imagine if `Int` was a subtype of `Long`:

```
// Hypothetical code, does not actually compile:
val a: Int? = 1    // A boxed Int (java.lang.Integer)
val b: Long? = a   // Implicit conversion yields a boxed Long (java.lang.Long)
print(b == a)      // Prints "false" as Long.equals() checks not only the value but wh
ether the other number is Long as well
```

数字运算

Kotlin支持数字运算的标准集：`+`、`-`、`*`、`/`、`%`。它们已定义为相应的类成员：

```
fun main() {
```

```
//sampleStart
    println(1 + 2)
    println(2_500_000_000L - 1L)
    println(3.14 * 2.71)
    println(10.0 / 3)
//sampleEnd
}
```

可以在自定义数字类中覆盖这些操作符。 详情请参见操作符重载。

整数除法

整数间的除法总是返回整数。会丢弃任何小数部分。

```
fun main() {
//sampleStart
    val x = 5 / 2
    println(x == 2.5)
    // Operator '==' cannot be applied to 'Int' and 'Double'

    println(x == 2)
    // true
//sampleEnd
}
```

对于任何两个整数类型之间的除法来说都是如此：

```
fun main() {
//sampleStart
    val x = 5L / 2
    println (x == 2)
    // Error, as Long (x) cannot be compared to Int (2)

    println(x == 2L)
    // true
//sampleEnd
}
```

如需返回带小数部分的除法结果，请将其中的一个参数显式转换为浮点类型：

```
fun main() {
//sampleStart
    val x = 5 / 2.toDouble()
    println(x == 2.5)
//sampleEnd
}
```

位运算

Kotlin 对整数提供了一组位运算。它们直接使用数字的比特表示在二进制级别进行操作。 位运算有可以通过中缀形式调用的函数表示。只能应用于 `Int` 与 `Long`：

```
fun main() {
//sampleStart
    val x = 1
    val xShiftedLeft = (x shl 2)
    println(xShiftedLeft)
    // 4

    val xAnd = x and 0x000FF000
    println(xAnd)
    // 0
//sampleEnd
}
```

完整的位运算列表：

- `shl(bits)` – 有符号左移
- `shr(bits)` – 有符号右移
- `ushr(bits)` – 无符号右移
- `and(bits)` – 位与
- `or(bits)` – 位或
- `xor(bits)` – 位异或
- `inv()` – 位非

浮点数比较

本节讨论的浮点数操作如下：

- 相等性检测：`a == b` 与 `a != b`
- 比较操作符：`a < b`、`a > b`、`a <= b`、`a >= b`
- 区间实例以及区间检测：`a..b`、`x in a..b`、`x !in a..b`

当其中的操作数 `a` 与 `b` 都是静态已知的 `Float` 或 `Double` 或者它们对应的可空类型（声明为该类型，或者推断为该类型，或者智能类型转换的结果是该类型），两数字所形成的操作或者区间遵循 IEEE 754 浮点运算标准。

然而，为了支持泛型场景并提供全序支持，对于并非静态类型就是浮点数的情况，行为是不同的。例如是 `Any`、`Comparable<...>` 或者 `Collection<T>` 类型。 这种情况下，这些操作使用为 `Float` 与 `Double` 实现的 `equals` 与 `compareTo`。 因此：

- 认为 `NaN` 与其自身相等

- 认为 `NaN` 比包括正无穷大（`POSITIVE_INFINITY`）在内的任何其他元素都大
- 认为 `-0.0` 小于 `0.0`

以下示例显示了静态类型作为浮点数（`Double.NaN`）的操作数与静态类型并非作为浮点数的操作数（`listOf(T)`）之间的行为差异。

```
fun main() {
    //sampleStart
    // 静态类型作为浮点数的操作数
    println(Double.NaN == Double.NaN)                 // false

    // 静态类型并非作为浮点数的操作数
    // 所以 NaN 等于它本身
    println(listOf(Double.NaN) == listOf(Double.NaN)) // true

    // 静态类型作为浮点数的操作数
    println(0.0 == -0.0)                              // true

    // 静态类型并非作为浮点数的操作数
    // 所以 -0.0 小于 0.0
    println(listOf(0.0) == listOf(-0.0))              // false

    println(listOf(Double.NaN, Double.POSITIVE_INFINITY, 0.0, -0.0).sorted())
    // [-0.0, 0.0, Infinity, NaN]
    //sampleEnd
}
```

无符号整型

除了整数类型，对于无符号整数，Kotlin 还提供了以下类型：

Type	Size (bits)	Min value	Max value
`UByte`	8	0	255
`UShort`	16	0	65,535
`UInt`	32	0	4,294,967,295 (2^{32} - 1)
`ULong`	64	0	18,446,744,073,709,551,615 (2^{64} - 1)

无符号类型支持其对应有符号类型的大多数操作。

> 无符号数实现为具有单个存储属性的内联类， 该属性类型为与其宽度相同的对应有符号类型。 If you want to convert between unsigned and signed integer types, make sure you update your code so that any function calls and operations support the new type.
>
> 

无符号数组与区间

> 无符号数组及其对应操作处于 Beta 状态。 它们可能随时进行不兼容变更。 需要选择加入（详见下文）。
>
> 

与原生类型相同，每个无符号类型都有表示相应类型数组的类型：

- `UByteArray`：无符号字节数组。
- `UShortArray`：无符号短整型数组。
- `UIntArray`：无符号整型数组。
- `ULongArray`：无符号长整型数组。

与有符号整型数组一样，它们提供了类似于 `Array` 类的 API 而没有装箱开销。

When you use unsigned arrays, you receive a warning that indicates that this feature is not stable yet. To remove the warning, opt-in with the `@ExperimentalUnsignedTypes` annotation. It's up to you to decide if your clients have to explicitly opt-in into usage of your API, but keep in mind that unsigned arrays are not a stable feature, so an API that uses them can be broken by changes in the language. Learn more about opt-in requirements.

区间与数列也支持 `UInt` 与 `ULong`（通过这些类 `UIntRange`、`UIntProgression`、`ULongRange`、`ULongProgression`）。 Together with the unsigned integer types, these classes are stable.

无符号整数字面值

为使无符号整型更易用，可以在整型字面值后附加一个后缀来表示指定无符号类型（类似于 `Float` 用 `F` 或 `Long` 用 `L`）：

- `u` and `U` letters signify unsigned literals without specifying the exact type. If no expected type is provided, the compiler uses `UInt` or `ULong` depending on the size of the literal:

```
val b: UByte = 1u  // UByte，已提供预期类型
val s: UShort = 1u // UShort，已提供预期类型
val l: ULong = 1u  // ULong，已提供预期类型

val a1 = 42u // UInt：未提供预期类型，常量适于 UInt
val a2 = 0xFFFF_FFFF_FFFFu // ULong：未提供预期类型，常量不适于 UInt
```

- `uL` 与 `UL` 显式指定字面值应为无符号长整型：

```
val a = 1UL // ULong，即使未提供预期类型并且常量适于 UInt
```

使用场景

The main use case of unsigned numbers is utilizing the full bit range of an integer to represent positive values.

For example, to represent hexadecimal constants that do not fit in signed types such as color in 32-bit `AARRGGBB` format:

```
data class Color(val representation: UInt)

val yellow = Color(0xFFCC00CCu)
```

You can use unsigned numbers to initialize byte arrays without explicit `toByte()` literal casts:

```
val byteOrderMarkUtf8 = ubyteArrayOf(0xEFu, 0xBBu, 0xBFu)
```

Another use case is interoperability with native APIs. Kotlin allows representing native declarations that contain unsigned types in the signature. The mapping won't substitute unsigned integers with signed ones keeping the semantics unaltered.

非目标

While unsigned integers can only represent positive numbers and zero, it's not a goal to use them where application domain requires non-negative integers. For example, as a type of collection size or collection index value.

There are a couple of reasons:

- Using signed integers can help to detect accidental overflows and signal error conditions, such as `List.lastIndex` being -1 for an empty list.
- Unsigned integers cannot be treated as a range-limited version of signed ones because their range of values is not a subset of the signed integers range. Neither signed, nor unsigned integers are subtypes of each other.

布尔

`Boolean` 类型表示可以有 `true` 与 `false` 两个值的布尔对象。 `Boolean` has a nullable counterpart declared as `Boolean?` .

> On the JVM, booleans stored as the primitive `boolean` type typically use 8 bits.

布尔值的内置运算有：

- `||` ——析取（逻辑或）
- `&&` ——合取（逻辑与）
- `!` ——否定（逻辑非）

For example:

```
fun main() {
//sampleStart
    val myTrue: Boolean = true
    val myFalse: Boolean = false
    val boolNull: Boolean? = null

    println(myTrue || myFalse)
    // true
    println(myTrue && myFalse)
    // false
    println(!myTrue)
    // false
    println(boolNull)
    // null
//sampleEnd
}
```

The `||` and `&&` operators work lazily, which means:

- If the first operand is `true` , the `||` operator does not evaluate the second operand.
- If the first operand is `false` , the `&&` operator does not evaluate the second operand.

> 在 JVM 平台，布尔对象的可空引用是装箱的 Java 类，类似于数字。

字符

字符用 `Char` 类型表示。 字符字面值用单引号括起来: `'1'` 。

> On the JVM, a character stored as primitive type: `char` , represents a 16-bit Unicode character.
>
> 

特殊字符可以以转义反斜杠 `\` 开始。 支持这几个转义序列：

- `\t` ——制表符
- `\b` ——退格符
- `\n` ——换行（LF）
- `\r` ——回车（CR）
- `\'` ——单引号
- `\"` ——双引号
- `\\` ——反斜杠
- `\$` ——美元符

编码其他字符要用 Unicode 转义序列语法： `'\uFF00'` 。

```
fun main() {
//sampleStart
    val aChar: Char = 'a'

    println(aChar)
    println('\n') // 输出一个额外的换行符
    println('\uFF00')
//sampleEnd
}
```

如果字符变量的值是数字，那么可以使用 `digitToInt()` 函数将其显式转换为 `Int` 数字。

> 在JVM 平台，当需要可空引用时字符会是装箱的 Java 类，类似于数字。 装箱操作不保留同一性。
>
> 

字符串

Kotlin 中字符串用 `String` 类型表示。

> On the JVM, an object of `String` type in UTF-16 encoding uses approximately 2 bytes per character.



通常，字符串值是双引号 (`"`) 中的字符序列：

```
val str = "abcd 123"
```

字符串的元素——字符可以使用索引运算符访问: `s[i]`。 可以使用 `for` 循环遍历这些字符：

```
fun main() {
    val str = "abcd"
//sampleStart
    for (c in str) {
        println(c)
    }
//sampleEnd
}
```

字符串是不可变的。 一旦初始化了一个字符串，就不能改变它的值或者给它赋新值。 所有转换字符串的操作都以一个新的 `String` 对象来返回结果，而保持原始字符串不变：

```
fun main() {
//sampleStart
    val str = "abcd"

    // 创建并输出一个新的 String 对象
    println(str.uppercase())
    // ABCD

    // 原始字符串保持不变
    println(str)
    // abcd
//sampleEnd
}
```

如需连接字符串，可以用 `+` 操作符。这也适用于连接字符串与其他类型的值， 只要表达式中的第一个元素是字符串：

```
fun main() {
//sampleStart
    val s = "abc" + 1
    println(s + "def")
    // abc1def
//sampleEnd
}
```

> 在大多数情况下，优先使用字符串模板或多行字符串而不是字符串连接。
>
> 

字符串字面值

Kotlin 有两种类型的字符串字面值:

- 转义字符串
- 多行字符串

转义字符串

转义字符串可以包含转义字符。

以下是转义字符串的一个示例：

```
val s = "Hello, world!\n"
```

转义采用传统的反斜杠（ `\` ）方式。

所支持的转义序列请参见 字符 页。

多行字符串

多行字符串可以包含换行以及任意文本。 它使用三个引号（ `"""` ）分界符括起来，内部没有转义并且可以包含换行以及任何其他字符：

```
val text = """
    for (c in "foo")
```

```
        print(c)
    """
```

如需删掉多行字符串中的前导空格，请使用 `trimMargin()` 函数：

```
val text = """
|Tell me and I forget.
|Teach me and I remember.
|Involve me and I learn.
|(Benjamin Franklin)
    """.trimMargin()
```

默认以竖线 `|` 作为边界前缀，但你可以选择其他字符并作为参数传入，比如 `trimMargin(">")` 。

字符串模板

字符串字面值可以包含模板表达式——一些小段代码，会求值并把结果合并到字符串中。When a template expression is processed, Kotlin automatically calls the `.toString()` function on the expression's result to convert it into a string. 模板表达式以美元符（ `$` ）开头，要么由一个变量名构成:

```
fun main() {
//sampleStart
    val i = 10
    println("i = $i")
    // i = 10

    val letters = listOf("a","b","c","d","e")
    println("Letters: $letters")
    // Letters: [a, b, c, d, e]

//sampleEnd
}
```

要么是用花括号括起来的表达式:

```
fun main() {
//sampleStart
    val s = "abc"
    println("$s.length is ${s.length}")
    // abc.length is 3
//sampleEnd
}
```

在多行字符串及转义字符串中都可以使用模板。不过，多行字符串不支持反斜杠转义。 如需在多行字符串中允许出现在标识符（identifier）开头的任意符号之前插入美元符 `$`， 请使用以下语法：

```
val price = """
${'$'}_9.99
"""
```

> To avoid `${'$'}` sequences in strings, you can use the Experimental multi-dollar string interpolation feature.
>
> 

Multi-dollar string interpolation

> The multi-dollar string interpolation is Experimental and opt-in is required (see details below).
>
> It may be changed at any time. We would appreciate your feedback in YouTrack.
>
> 

Multi-dollar string interpolation allows you to specify how many consecutive dollar signs are required to trigger interpolation. Interpolation is the process of embedding variables or expressions directly into a string.

While you can escape literals for single-line strings, multiline strings in Kotlin don't support backslash escaping. To include dollar signs (`$`) as literal characters, you must use the `${'$'}` construct to prevent string interpolation. This approach can make code harder to read, especially when strings contain multiple dollar signs.

Multi-dollar string interpolation simplifies this by letting you treat dollar signs as literal characters in both single-line and multiline strings. For example:

```
val KClass<*>.jsonSchema : String
    get() = $$"""
    {
      "$schema": "https://json-schema.org/draft/2020-12/schema",
      "$id": "https://example.com/product.schema.json",
      "$dynamicAnchor": "meta"
      "title": "$${simpleName ?: qualifiedName ?: "unknown"}",
      "type": "object"
    }
    """
```

Here, the `$$` prefix specifies that two consecutive dollar signs are required to trigger string interpolation. Single dollar signs remain as literal characters.

You can adjust how many dollar signs trigger interpolation. For example, using three consecutive dollar signs (`$$$`) allows `$` and `$$` to remain as literals while enabling interpolation with `$$$` :

```
val productName = "carrot"
val requestedData =
    $$$"""{
      "currency": "$",
      "enteredAmount": "42.45 $$",
      "$$serviceField": "none",
      "product": "$$$productName"
    }
    """

println(requestedData)
//{
//    "currency": "$",
//    "enteredAmount": "42.45 $$",
//    "$$serviceField": "none",
//    "product": "carrot"
//}
```

Here, the `$$$` prefix allows the string to include `$` and `$$` without requiring the `${'$'}` construct for escaping.

To enable the feature, use the following compiler option in the command line:

```
kotlinc -Xmulti-dollar-interpolation main.kt
```

Alternatively, update the `compilerOptions {}` block of your Gradle build file:

```
// build.gradle.kts
kotlin {
    compilerOptions {
        freeCompilerArgs.add("-Xmulti-dollar-interpolation")
    }
}
```

This feature doesn't affect existing code that uses single-dollar string interpolation. You can continue using a single `$` as before and apply multi-dollar signs when you need to handle literal dollar signs in strings.

String formatting

> String formatting with the `String.format()` function is only available in Kotlin/JVM.



To format a string to your specific requirements, use the `String.format()` function.

The `String.format()` function accepts a format string and one or more arguments. The format string contains one placeholder (indicated by `%`) for a given argument, followed by format specifiers. Format specifiers are formatting instructions for the respective argument, consisting of flags, width, precision, and conversion type. Collectively, format specifiers shape the output's formatting. Common format specifiers include `%d` for integers, `%f` for floating-point numbers, and `%s` for strings. You can also use the `argument_index$` syntax to reference the same argument multiple times within the format string in different formats.

> For a detailed understanding and an extensive list of format specifiers, see Java's Class Formatter documentation.
>
> 

Let's look at an example:

```
fun main() {
//sampleStart
    // Formats an integer, adding leading zeroes to reach a length of seven characters
    val integerNumber = String.format("%07d", 31416)
    println(integerNumber)
    // 0031416

    // Formats a floating-point number to display with a + sign and four decimal places

    val floatNumber = String.format("%+.4f", 3.141592)
    println(floatNumber)
    // +3.1416

    // Formats two strings to uppercase, each taking one placeholder
    val helloString = String.format("%S %S", "hello", "world")
    println(helloString)
    // HELLO WORLD

    // Formats a negative number to be enclosed in parentheses, then repeats the same
number in a different format (without parentheses) using `argument_index$`.
    val negativeNumberInParentheses = String.format("%(d means %1\$d", -31416)
    println(negativeNumberInParentheses)
    //(31416) means -31416
//sampleEnd
}
```

{interpolate-variables="false" kotlin-runnable="true" kotlin-min-compiler-version="1.3"}

The `String.format()` function provides similar functionality to string templates. However, the `String.format()` function is more versatile because there are more formatting options available.

In addition, you can assign the format string from a variable. This can be useful when the format string changes, for example, in localization cases that depend on the user locale.

Be careful when using the `String.format()` function because it can be easy to mismatch the number or position of the arguments with their corresponding placeholders.

数组

数组是一种保存固定数量相同类型或其子类型的值的数据结构。 Kotlin 中最常见的数组类型是对象类型数组，由 `Array` 类表示。

> 如果在对象类型数组中使用原生类型，那么会对性能产生影响，因为原生值都装箱成了对象。 为了避免装箱开销，请改用原生类型数组。
>
> 

何时使用数组

当需要满足特殊的底层需求时，可以在 Kotlin 中使用数组。 例如，如果性能要求超出了常规应用程序的需要，或者需要构建自定义的数据结构。 如果没有这些限制，那么请改用集合。

与数组相比，集合具有以下优点：

- 集合可以是只读的，这提供了更多的控制权而支持编写具有明确意图的健壮代码。
- 易于对集合增删元素。相比之下，数组大小是固定的。 对数组增删元素的唯一方式是每次创建一个新数组，效率很低：

```
fun main() {
//sampleStart
    var riversArray = arrayOf("Nile", "Amazon", "Yangtze")

    // 使用 += 赋值操作创建了一个新的 riversArray，
    // 复制了原始元素并添加了“Mississippi”
    riversArray += "Mississippi"
    println(riversArray.joinToString())
    // Nile, Amazon, Yangtze, Mississippi
//sampleEnd
}
```

- 可以使用相等操作符（ `==` ）来检验两个集合是否在结构上相等。但不能对数组使用这个操作符。 相反，必须使用一个特殊函数，关于这点可以参阅比较数组了解更多信息。

关于集合的更多信息请参见集合概述。

创建数组

To create arrays in Kotlin, you can use:

- functions, such as `arrayOf()`, `arrayOfNulls()`) or `emptyArray()`.
- the `Array` constructor.

This example uses the `arrayOf()` function and passes item values to it:

```
fun main() {
//sampleStart
    // Creates an array with values [1, 2, 3]
    val simpleArray = arrayOf(1, 2, 3)
    println(simpleArray.joinToString())
    // 1, 2, 3
//sampleEnd
}
```

This example uses the `arrayOfNulls()`) function to create an array of a given size filled with `null` elements:

```
fun main() {
//sampleStart
    // Creates an array with values [null, null, null]
    val nullArray: Array<Int?> = arrayOfNulls(3)
    println(nullArray.joinToString())
    // null, null, null
//sampleEnd
}
```

This example uses the `emptyArray()` function to create an empty array :

```
    var exampleArray = emptyArray<String>()
```

> You can specify the type of the empty array on the left-hand or right-hand side of the assignment due to Kotlin's type inference.
>
> For example:
>
> ```
> var exampleArray = emptyArray<String>()
>
> var exampleArray: Array<String> = emptyArray()
> ```



The `Array` constructor takes the array size and a function that returns values for array elements given its index:

```
fun main() {
```

```
//sampleStart
    // Creates an Array<Int> that initializes with zeros [0, 0, 0]
    val initArray = Array<Int>(3) { 0 }
    println(initArray.joinToString())
    // 0, 0, 0

    // 创建一个 Array<String> 初始化为 ["0", "1", "4", "9", "16"]
    val asc = Array(5) { i -> (i * i).toString() }
    asc.forEach { print(it) }
    // 014916
//sampleEnd
}
```

> Like in most programming languages, indices start from 0 in Kotlin.
>
> 

嵌套数组

Arrays can be nested within each other to create multidimensional arrays:

```
fun main() {
//sampleStart
    // Creates a two-dimensional array
    val twoDArray = Array(2) { Array<Int>(2) { 0 } }
    println(twoDArray.contentDeepToString())
    // [[0, 0], [0, 0]]

    // Creates a three-dimensional array
    val threeDArray = Array(3) { Array(3) { Array<Int>(3) { 0 } } }
    println(threeDArray.contentDeepToString())
    // [[[0, 0, 0], [0, 0, 0], [0, 0, 0]], [[0, 0, 0], [0, 0, 0], [0, 0, 0]], [[0, 0,
0], [0, 0, 0], [0, 0, 0]]]
//sampleEnd
}
```

> Nested arrays don't have to be the same type or the same size.
>
> 

访问与修改元素

Arrays are always mutable. To access and modify elements in an array, use the indexed access operator `[]`:

```
fun main() {
//sampleStart
```

```
    val simpleArray = arrayOf(1, 2, 3)
    val twoDArray = Array(2) { Array<Int>(2) { 0 } }

    // Accesses the element and modifies it
    simpleArray[0] = 10
    twoDArray[0][0] = 2

    // Prints the modified element
    println(simpleArray[0].toString()) // 10
    println(twoDArray[0][0].toString()) // 2
//sampleEnd
}
```

Arrays in Kotlin are *invariant*. This means that Kotlin doesn't allow you to assign an `Array<String>` to an `Array<Any>` to prevent a possible runtime failure. Instead, you can use `Array<out Any>`. For more information, see Type Projections.

使用数组

In Kotlin, you can work with arrays by using them to pass a variable number of arguments to a function or perform operations on the arrays themselves. For example, comparing arrays, transforming their contents or converting them to collections.

向函数传入可变数量的实参

In Kotlin, you can pass a variable number of arguments to a function via the `vararg` parameter. This is useful when you don't know the number of arguments in advance, like when formatting a message or creating an SQL query.

To pass an array containing a variable number of arguments to a function, use the *spread* operator (`*`). The spread operator passes each element of the array as individual arguments to your chosen function:

```
fun main() {
    val lettersArray = arrayOf("c", "d")
    printAllStrings("a", "b", *lettersArray)
    // abcd
}

fun printAllStrings(vararg strings: String) {
    for (string in strings) {
        print(string)
    }
}
```

For more information, see Variable number of arguments (varargs).

比较数组

To compare whether two arrays have the same elements in the same order, use the `.contentEquals()` and `.contentDeepEquals()` functions:

```
fun main() {
//sampleStart
    val simpleArray = arrayOf(1, 2, 3)
    val anotherArray = arrayOf(1, 2, 3)

    // Compares contents of arrays
    println(simpleArray.contentEquals(anotherArray))
    // true

    // Using infix notation, compares contents of arrays after an element
    // is changed
    simpleArray[0] = 10
    println(simpleArray contentEquals anotherArray)
    // false
//sampleEnd
}
```

> Don't use equality (`==`) and inequality (`!=`) operators to compare the contents of arrays. These operators check whether the assigned variables point to the same object.
>
> To learn more about why arrays in Kotlin behave this way, see our blog post.



转换数组

Kotlin has many useful functions to transform arrays. This document highlights a few but this isn't an exhaustive list. For the full list of functions, see our API reference.

Sum

To return the sum of all elements in an array, use the `.sum()` function:

```
fun main() {
//sampleStart
    val sumArray = arrayOf(1, 2, 3)

    // Sums array elements
    println(sumArray.sum())
    // 6
```

```
//sampleEnd
}
```

> The `.sum()` function can only be used with arrays of numeric data types, such as `Int`.
>
> 

Shuffle

To randomly shuffle the elements in an array, use the `.shuffle()` function:

```
fun main() {
//sampleStart
    val simpleArray = arrayOf(1, 2, 3)

    // Shuffles elements [3, 2, 1]
    simpleArray.shuffle()
    println(simpleArray.joinToString())

    // Shuffles elements again [2, 3, 1]
    simpleArray.shuffle()
    println(simpleArray.joinToString())
//sampleEnd
}
```

将数组转换为集合

If you work with different APIs where some use arrays and some use collections, then you can convert your arrays to collections and vice versa.

转换为 List 或 Set

To convert an array to a `List` or `Set`, use the `.toList()` and `.toSet()` functions.

```
fun main() {
//sampleStart
    val simpleArray = arrayOf("a", "b", "c", "c")

    // Converts to a Set
    println(simpleArray.toSet())
    // [a, b, c]

    // Converts to a List
    println(simpleArray.toList())
    // [a, b, c, c]
//sampleEnd
}
```

转换为 Map

To convert an array to a `Map`, use the `.toMap()` function.

Only an array of `Pair<K,V>` can be converted to a `Map`. The first value of a `Pair` instance becomes a key, and the second becomes a value. This example uses the infix notation to call the `to` function to create tuples of `Pair`:

```
fun main() {
//sampleStart
    val pairArray = arrayOf("apple" to 120, "banana" to 150, "cherry" to 90, "apple" t
o 140)

    // Converts to a Map
    // The keys are fruits and the values are their number of calories
    // Note how keys must be unique, so the latest value of "apple"
    // overwrites the first
    println(pairArray.toMap())
    // {apple=140, banana=150, cherry=90}

//sampleEnd
}
```

原生类型数组

If you use the `Array` class with primitive values, these values are boxed into objects. As an alternative, you can use primitive-type arrays, which allow you to store primitives in an array without the side effect of boxing overhead:

Primitive-type array	Equivalent in Java
`BooleanArray`	`boolean[]`
`ByteArray`	`byte[]`
`CharArray`	`char[]`
`DoubleArray`	`double[]`
`FloatArray`	`float[]`
`IntArray`	`int[]`
`LongArray`	`long[]`
`ShortArray`	`short[]`

These classes have no inheritance relation to the `Array` class, but they have the same set of functions and properties.

This example creates an instance of the `IntArray` class:

```
fun main() {
//sampleStart
    // Creates an array of Int of size 5 with the values initialized to zero
    val exampleArray = IntArray(5)
    println(exampleArray.joinToString())
    // 0, 0, 0, 0, 0
//sampleEnd
}
```

> To convert primitive-type arrays to object-type arrays, use the `.toTypedArray()` function.
>
> To convert object-type arrays to primitive-type arrays, use `.toBooleanArray()`, `.toByteArray()`, `.toCharArray()`, and so on.
>
> 

下一步做什么？

- To learn more about why we recommend using collections for most use cases, read our Collections overview.
- Learn about other basic types.
- If you are a Java developer, read our Java to Kotlin migration guide for Collections.

类型检测与类型转换

In Kotlin, you can perform type checks to check the type of an object at runtime. Type casts enable you to convert objects to a different type.

> To learn specifically about **generics** type checks and casts, for example `List<T>`, `Map<K,V>`, see Generics type checks and casts.
>
> 

is 与 !is 操作符

如需在运行时检测对象是否符合给定类型，使用 `is` 操作符或其否定形式 `!is`：

```
if (obj is String) {
    print(obj.length)
}

if (obj !is String) { // 与 !(obj is String) 相同
    print("Not a String")
} else {
    print(obj.length)
}
```

智能转换

大多数场景都不需要使用显式转换操作符，because the compiler automatically casts objects for you. This is called smart-casting. 编译器跟踪不可变值的类型检测以及显式转换， 并在必要时自动插入隐式安全的）转换：

```
fun demo(x: Any) {
    if (x is String) {
        print(x.length) // x 自动转换为字符串
    }
}
```

编译器甚至足够聪明，能够知道如果反向检测导致返回那么该转换是安全的：

```
if (x !is String) return

print(x.length) // x 自动转换为字符串
```

Control flow

Smart casts work not only for `if` conditional expressions but also for `when` expressions and `while` loops:

```
when (x) {
    is Int -> print(x + 1)
    is String -> print(x.length + 1)
    is IntArray -> print(x.sum())
}
```

If you declare a variable of `Boolean` type before using it in your `if`, `when`, or `while` condition, then any information collected by the compiler about the variable will be accessible in the corresponding block for smart-casting.

This can be useful when you want to do things like extract boolean conditions into variables. Then, you can give the variable a meaningful name, which will improve your code readability and make it possible to reuse the variable later in your code. For example:

```
class Cat {
    fun purr() {
        println("Purr purr")
    }
}

fun petAnimal(animal: Any) {
    val isCat = animal is Cat
    if (isCat) {
        // The compiler can access information about
        // isCat, so it knows that animal was smart-cast
        // to the type Cat.
        // Therefore, the purr() function can be called.
        animal.purr()
    }
}

fun main(){
    val kitty = Cat()
    petAnimal(kitty)
    // Purr purr
}
```

Logical operators

The compiler can perform smart casts on the right-hand side of `&&` or `||` operators if there is a type check (regular or negative) on the left-hand side:

```
// `||` 右侧的 x 自动转换为 String
if (x !is String || x.length == 0) return

// `&&` 右侧的 x 自动转换为 String
if (x is String && x.length > 0) {
    print(x.length) // x 自动转换为 String
}
```

If you combine type checks for objects with an `or` operator (`||`), a smart cast is made to their closest common supertype:

```
interface Status {
    fun signal() {}
}

interface Ok : Status
interface Postponed : Status
interface Declined : Status

fun signalCheck(signalStatus: Any) {
    if (signalStatus is Postponed || signalStatus is Declined) {
        // signalStatus is smart-cast to a common supertype Status
        signalStatus.signal()
    }
}
```

> The common supertype is an **approximation** of a union type. Union types are not currently supported in Kotlin.
>
> 

Inline functions

The compiler can smart-cast variables captured within lambda functions that are passed to inline functions.

Inline functions are treated as having an implicit `callsInPlace` contract. This means that any lambda functions passed to an inline function are called in place. Since lambda functions are called in place, the compiler knows that a lambda function can't leak references to any variables contained within its function body.

The compiler uses this knowledge, along with other analyses to decide whether it's safe to smart-cast any of the captured variables. For example:

```
interface Processor {
    fun process()
}

inline fun inlineAction(f: () -> Unit) = f()

fun nextProcessor(): Processor? = null

fun runProcessor(): Processor? {
    var processor: Processor? = null
    inlineAction {
        // The compiler knows that processor is a local variable and inlineAction()
        // is an inline function, so references to processor can't be leaked.
        // Therefore, it's safe to smart-cast processor.

        // If processor isn't null, processor is smart-cast
        if (processor != null) {
            // The compiler knows that processor isn't null, so no safe call
            // is needed
            processor.process()
        }

        processor = nextProcessor()
    }

    return processor
}
```

Exception handling

Smart cast information is passed on to `catch` and `finally` blocks. This makes your code safer as the compiler tracks whether your object has a nullable type. For example:

```
//sampleStart
fun testString() {
    var stringInput: String? = null
    // stringInput is smart-cast to String type
    stringInput = ""
    try {
        // The compiler knows that stringInput isn't null
        println(stringInput.length)
        // 0

        // The compiler rejects previous smart cast information for
        // stringInput. Now stringInput has the String? type.
        stringInput = null

        // Trigger an exception
        if (2 > 1) throw Exception()
        stringInput = ""
```

```
    } catch (exception: Exception) {
        // The compiler knows stringInput can be null
        // so stringInput stays nullable.
        println(stringInput?.length)
        // null
    }
}
//sampleEnd
fun main() {
    testString()
}
```

Smart cast prerequisites

> 请注意，当编译器能保证变量在检测及其使用之间不可改变时，智能转换才有效。
>
> 

智能转换适用于以下情形：

`val` 局部变量	总是可以，局部委托属性除外。
`val` 属性	如果属性是 `private` 、 `internal` ，或者该检测在声明属性的同一模块中执行。 智能转换不能用于 `open` 的属性或者具有自定义 getter 的属性。
`var` 局部变量	如果变量在检测及其使用之间未修改、没有在会修改它的 lambda 中捕获、并且不是局部委托属性。
`var` 属性	决不可能，因为该变量可以随时被其他代码修改。

“不安全的”转换操作符

To explicitly cast an object to a non-nullable type, use the *unsafe* cast operator `as` :

```
val x: String = y as String
```

If the cast isn't possible, the compiler throws an exception. This is why it's called *unsafe*.

In the previous example, if `y` is `null` , the code above also throws an exception. This is because `null` can't be cast to `String` , as `String` isn't nullable. To make the example work for possible null values, use a nullable type on the right-hand side of the cast:

```
val x: String? = y as String?
```

“安全的”（可空）转换操作符

为了避免异常，可以使用安全转换操作符 `as?`，它可以在失败时返回 `null`：

```
val x: String? = y as? String
```

请注意，尽管事实上 `as?` 的右边是一个非空类型的 `String`，但是其转换的结果是可空的。

控制流程

- 条件与循环
- 返回与跳转
- 异常

条件与循环

If 表达式

在 Kotlin 中，`if` 是一个表达式：它会返回一个值。 因此就不需要三元运算符（`条件 ? 然后 : 否则`），因为普通的 `if` 就能胜任这个角色。

```
fun main() {
    val a = 2
    val b = 3

    //sampleStart
    var max = a
    if (a < b) max = b

    // With else
    if (a > b) {
      max = a
    } else {
      max = b
    }

  // 作为表达式
 max = if (a > b) a else b

    // You can also use `else if` in expressions:
    val maxLimit = 1
    val maxOrLimit = if (maxLimit > a) maxLimit else if (a > b) a else b

    println("max is $max")
    // max is 3
    println("maxOrLimit is $maxOrLimit")
    // maxOrLimit is 3
    //sampleEnd
}
```

`if` 表达式的分支可以是代码块，这种情况最后的表达式作为该块的值：

```
val max = if (a > b) {
    print("Choose a")
    a
} else {
    print("Choose b")
    b
}
```

If you're using `if` as an expression, for example, for returning its value or assigning it to a variable, the `else` branch is mandatory.

When expressions and statements

`when` is a conditional expression that runs code based on multiple possible values or conditions. It is similar to the `switch` statement in Java, C, and similar languages. For example:

```
fun main() {
    //sampleStart
    val x = 2
    when (x) {
        1 -> print("x == 1")
        2 -> print("x == 2")
        else -> print("x is neither 1 nor 2")
    }
    // x == 2
    //sampleEnd
}
```

`when` 将它的参数与所有的分支条件顺序比较，直到某个分支满足条件。

You can use `when` in a few different ways. Firstly, you can use `when` either as an **expression** or as a **statement**. As an expression, `when` returns a value for later use in your code. As a statement, `when` completes an action without returning anything of further use:

Expression	Statement
```kotlin // Returns a string assigned to the // text variable val text = when (x) { 1 -> "x == 1" 2 -> "x == 2" else -> "x is neither 1 nor 2" } ```	```kotlin // Returns nothing but triggers a // print statement when (x) { 1 -> print("x == 1") 2 -> print("x == 2") else -> print("x is neither 1 nor 2") } ```

Secondly, you can use `when` with or without a subject. Whether you use a subject with `when` or not, your expression or statement behaves the same. We recommend using `when` with a subject when possible, as it makes your code easier to read and maintain by clearly showing what you're checking.

With subject `x`	Without subject
```kotlin when(x) { ... } ```	```kotlin when { ... } ```

Depending on how you use `when`, there are different requirements for whether you need to cover all possible cases in your branches.

If you use `when` as a statement, you don't have to cover all possible cases. In this example, some cases aren't covered, so nothing happens. However, no error occurs:

```
fun main() {
    //sampleStart
    val x = 3
    when (x) {
        // Not all cases are covered
        1 -> print("x == 1")
        2 -> print("x == 2")
    }
    //sampleEnd
}
```

In a `when` statement, the values of individual branches are ignored. Just like with `if`, each branch can be a block, and its value is the value of the last expression in the block.

If you use `when` as an expression, you have to cover all possible cases. In other words, it must be *exhaustive*. The value of the first matching branch becomes the value of the overall expression. If you don't cover all cases, the compiler throws an error.

If your `when` expression has a subject, you can use an `else` branch to make sure that all possible cases are covered, but it isn't mandatory. For example, if your subject is a `Boolean`, `enum` class, `sealed` class, or one of their nullable counterparts, you can cover all cases without an `else` branch:

```
enum class Bit {
    ZERO, ONE
}

val numericValue = when (getRandomBit()) {
    // No else branch is needed because all cases are covered
    Bit.ZERO -> 0
    Bit.ONE -> 1
}
```

If your `when` expression **doesn't** have a subject, you **must** have an `else` branch or the compiler throws an error. The `else` branch is evaluated when none of the other branch conditions are satisfied:

```
val message = when {
    a > b -> "a is greater than b"
    a < b -> "a is less than b"
    else -> "a is equal to b"
}
```

`when` expressions and statements offer different ways to simplify your code, handle multiple conditions, and perform type checks.

You can define a common behavior for multiple cases by combining their conditions in a single line with a comma:

```
when (x) {
    0, 1 -> print("x == 0 or x == 1")
    else -> print("otherwise")
}
```

可以用任意表达式（而不只是常量）作为分支条件：

```
when (x) {
    s.toInt() -> print("s encodes x")
    else -> print("s does not encode x")
}
```

You can also check whether a value is or isn't contained in a range or collection via the `in` or `!in` keywords:

```
when (x) {
    in 1..10 -> print("x is in the range")
    in validNumbers -> print("x is valid")
    !in 10..20 -> print("x is outside the range")
    else -> print("none of the above")
}
```

Additionally, you can check that a value is or isn't a particular type via the `is` or `!is` keywords. 注意： 由于智能转换，可以访问该类型的成员函数与属性而无需任何额外的检测。

```
fun hasPrefix(x: Any) = when(x) {
    is String -> x.startsWith("prefix")
    else -> false
}
```

可以用 `when` 来取代 `if - else if` 链。 如果没有主语（subject，译注：指 `when` 所判断的表达式），所有的分支条件都是简单的布尔表达式。第一个条件为 `true` 的分支会执行：

```
when {
    x.isOdd() -> print("x is odd")
    y.isEven() -> print("y is even")
    else -> print("x+y is odd")
}
```

可以使用以下语法将主语捕获到变量中：

```
fun Request.getBody() =
    when (val response = executeRequest()) {
        is Success -> response.body
        is HttpError -> throw HttpException(response.status)
    }
```

作为主语引入的变量的作用域仅限于 `when` 表达式或语句的主体。

Guard conditions in when expressions

> Guard conditions are an experimental feature that may be changed at any time. We would appreciate your feedback in YouTrack.
>
> 

Guard conditions allow you to include more than one condition to the branches of a `when` expression, making complex control flow more explicit and concise. You can use guard conditions in `when` expressions or statements with a subject.

To include a guard condition in a branch, place it after the primary condition, separated by `if`:

```
sealed interface Animal {
    data class Cat(val mouseHunter: Boolean) : Animal
    data class Dog(val breed: String) : Animal
}

fun feedAnimal(animal: Animal) {
    when (animal) {
        // Branch with only primary condition. Calls `feedDog()` when `animal` is `Dog`

        is Animal.Dog -> feedDog()
        // Branch with both primary and guard conditions. Calls `feedCat()` when `anim
al` is `Cat` and not `mouseHunter`
        is Animal.Cat if !animal.mouseHunter -> feedCat()
        // Prints "Unknown animal" if none of the above conditions match
        else -> println("Unknown animal")
    }
}
```

In a single `when` expression, you can combine branches with and without guard conditions. The code in a branch with a guard condition runs only if both the primary condition and the guard condition evaluate to true. If the primary condition does not match, the guard condition is not evaluated.

If you use guard conditions in `when` statements without an `else` branch, and none of the conditions matches, none of the branches is executed.

Otherwise, if you use guard conditions in `when` expressions without an `else` branch, the compiler requires you to declare all the possible cases to avoid runtime errors.

Additionally, guard conditions support `else if`:

```
when (animal) {
    // Checks if `animal` is `Dog`
    is Animal.Dog -> feedDog()
    // Guard condition that checks if `animal` is `Cat` and not `mouseHunter`
    is Animal.Cat if !animal.mouseHunter -> feedCat()
    // Calls giveLettuce() if none of the above conditions match and animal.eatsPlants
 is true
    else if animal.eatsPlants -> giveLettuce()
    // Prints "Unknown animal" if none of the above conditions match
    else -> println("Unknown animal")
}
```

Combine multiple guard conditions within a single branch using the boolean operators `&&` (AND) or `||` (OR). Use parentheses around the boolean expressions to avoid confusion:

```
when (animal) {
    is Animal.Cat if (!animal.mouseHunter && animal.hungry) -> feedCat()
}
```

You can use guard conditions in any `when` expression or statement with a subject, except the case when you have multiple conditions separated by a comma. For example, `0, 1 -> print("x == 0 or x == 1")`.

> To enable guard conditions in CLI, run the following command:
>
> `kotlinc -Xwhen-guards main.kt`
>
> To enable guard conditions in Gradle, add the following line to the `build.gradle.kts` file:
>
> `kotlin.compilerOptions.freeCompilerArgs.add("-Xwhen-guards")`
>
> 

For 循环

`for` 循环可以对任何提供迭代器（iterator）的对象进行遍历，这相当于像 C# 这样的语言中的 `foreach` 循环。 `for` 的语法如下所示：

```
for (item in collection) print(item)
```

`for` 循环体可以是一个代码块。

```
for (item: Int in ints) {
    // ……
}
```

如上所述，`for` 可以循环遍历任何提供了迭代器的对象。这意味着：

- 有一个成员函数或者扩展函数 `iterator()` 返回 `Iterator<>`，其中：
  - 有一个成员函数或者扩展函数 `next()`
  - 有一个成员函数或者扩展函数 `hasNext()` 返回 `Boolean`。

这三个函数都需要标记为 `operator`。

如需在数字区间上迭代，请使用区间表达式：

```
fun main() {
//sampleStart
    for (i in 1..3) {
        print(i)
    }
    for (i in 6 downTo 0 step 2) {
        print(i)
    }
    // 1236420
//sampleEnd
}
```

对区间或者数组的 `for` 循环会被编译为并不创建迭代器的基于索引的循环。

如果你想要通过索引遍历一个数组或者一个 list，你可以这么做：

```
fun main() {
val array = arrayOf("a", "b", "c")
//sampleStart
    for (i in array.indices) {
        print(array[i])
    }
    // abc
//sampleEnd
}
```

或者你可以用库函数 `withIndex`：

```
fun main() {
    val array = arrayOf("a", "b", "c")
//sampleStart
```

```
    for ((index, value) in array.withIndex()) {
        println("the element at $index is $value")
    }
    // the element at 0 is a
    // the element at 1 is b
    // the element at 2 is c
//sampleEnd
}
```

while 循环

`while` 和 `do-while` 当循环条件满足时会持续处理它们的主体。 它们之间的区别在于条件检查的时间：

- `while` 先检查条件，如果满足，则处理主体，然后再返回到条件检查。
- `do-while` 先处理主体，然后检查条件。如果满足，则循环重复。所以 `do-while` 的主体至少运行一次，不管条件如何。

```
while (x > 0) {
    x--
}

do {
  val y = retrieveData()
} while (y != null) // y 在此处可见
```

循环中的 break 与 continue

在循环中 Kotlin 支持传统的 `break` 与 `continue` 操作符。参见返回与跳转。

返回与跳转

Kotlin 有三种结构化跳转表达式：

- `return` 默认从最直接包围它的函数或者匿名函数返回。
- `break` 终止最直接包围它的循环。
- `continue` 继续下一次最直接包围它的循环。

所有这些表达式都可以用作更大表达式的一部分：

```
val s = person.name ?: return
```

这些表达式的类型是 Nothing 类型。

Break 与 Continue 标签

在 Kotlin 中任何表达式都可以用标签来标记。 标签的格式为标识符后跟 `@` 符号，例如：`abc@` 、 `fooBar@` 。 要为一个表达式加标签，我们只要在其前加标签即可。

```
loop@ for (i in 1..100) {
    // ……
}
```

现在，可以用标签限定 `break` 或者 `continue` ：

```
loop@ for (i in 1..100) {
    for (j in 1..100) {
        if (……) break@loop
    }
}
```

标签限定的 `break` 跳转到刚好位于该标签指定的循环后面的执行点。 `continue` 继续标签指定的循环的下一次迭代。

> In some cases, you can apply `break` and `continue` *non-locally* without explicitly defining labels. Such non-local usages are valid in lambda expressions used in enclosing inline functions.

返回到标签

Kotlin 中函数可以使用函数字面量、局部函数与对象表达式实现嵌套。 标签限定的 `return` 让你可能从外层函数返回。

最重要的一个用途就是从 lambda 表达式中返回。 To return from a lambda expression, label it and qualify the `return`：

```
//sampleStart
fun foo() {
    listOf(1, 2, 3, 4, 5).forEach lit@{
        if (it == 3) return@lit // 局部返回到该 lambda 表达式的调用者—forEach 循环
        print(it)
    }
    print(" done with explicit label")
}
//sampleEnd

fun main() {
    foo()
}
```

现在，它只会从 lambda 表达式中返回。通常情况下使用隐式标签更方便，因为该标签与接受该 lambda 的函数同名。

```
//sampleStart
fun foo() {
    listOf(1, 2, 3, 4, 5).forEach {
        if (it == 3) return@forEach // 局部返回到该 lambda 表达式的调用者—forEach 循环
        print(it)
    }
    print(" done with implicit label")
}
//sampleEnd

fun main() {
    foo()
}
```

或者，我们用一个匿名函数替代 lambda 表达式。 匿名函数内部的 `return` 语句将从该匿名函数自身返回

```
//sampleStart
fun foo() {
    listOf(1, 2, 3, 4, 5).forEach(fun(value: Int) {
        if (value == 3) return  // 局部返回到匿名函数的调用者—forEach 循环
        print(value)
    })
    print(" done with anonymous function")
}
```

```
//sampleEnd

fun main() {
    foo()
}
```

请注意，前文三个示例中使用的局部返回类似于在常规循环中使用 `continue`。

并没有 `break` 的直接等价形式，不过可以通过增加另一层嵌套 lambda 表达式并从其中非局部返回来模拟：

```
//sampleStart
fun foo() {
    run loop@{
        listOf(1, 2, 3, 4, 5).forEach {
            if (it == 3) return@loop // 从传入 run 的 lambda 表达式非局部返回
            print(it)
        }
    }
    print(" done with nested loop")
}
//sampleEnd

fun main() {
    foo()
}
```

当要返一个回值的时候，解析器优先选用标签限定的返回：

```
return@a 1
```

这意味着“返回 `1` 到 `@a`”，而不是“返回一个标签标注的表达式 `(@a 1)`”。

> In some cases, you can return from a lambda expression without using labels. Such *non-local* returns are located in a lambda but exit the enclosing inline function.
>
> 

异常

Exceptions help your code run more predictably, even when runtime errors occur that could disrupt program execution. Kotlin treats all exceptions as *unchecked* by default. Unchecked exceptions simplify the exception handling process: you can catch exceptions, but you don't need to explicitly handle or declare them.

> Learn more about how Kotlin handles exceptions when interacting with Java, Swift, and Objective-C in the Exception interoperability with Java, Swift, and Objective-C section.
>
> 

Working with exceptions consists of two primary actions:

- **Throwing exceptions:** indicate when a problem occurs.
- **Catching exceptions:** handle the unexpected exception manually by resolving the issue or notifying the developer or application user.

Exceptions are represented by subclasses of the `Exception` class, which is a subclass of the `Throwable` class. For more information about the hierarchy, see the Exception hierarchy section. Since `Exception` is an `open class`, you can create custom exceptions to suit your application's specific needs.

Throw exceptions

You can manually throw exceptions with the `throw` keyword. Throwing an exception indicates that an unexpected runtime error has occurred in the code. Exceptions are objects, and throwing one creates an instance of an exception class.

You can throw an exception without any parameters:

```
throw IllegalArgumentException()
```

To better understand the source of the problem, include additional information, such as a custom message and the original cause:

```
val cause = IllegalStateException("Original cause: illegal state")

// Throws an IllegalArgumentException if userInput is negative
// Additionally, it shows the original cause, represented by the cause IllegalStateExc
eption
if (userInput < 0) {
```

```
    throw IllegalArgumentException("Input must be non-negative", cause)
}
```

In this example, an `IllegalArgumentException` is thrown when the user inputs a negative value. You can create custom error messages and keep the original cause (`cause`) of the exception, which will be included in the stack trace.

Throw exceptions with precondition functions

Kotlin offers additional ways to automatically throw exceptions using precondition functions. Precondition functions include:

Precondition function	Use case	Exception thrown
`require()`	Checks user input validity	`IllegalArgumentException`
`check()`	Checks object or variable state validity	`IllegalStateException`
`error()`	Indicates an illegal state or condition	`IllegalStateException`

These functions are suitable for situations where the program's flow cannot continue if specific conditions aren't met. This streamlines your code and makes handling these checks efficient.

require() function

Use the `require()` function to validate input arguments when they are crucial for the function's operation, and the function can't proceed if these arguments are invalid.

If the condition in `require()` is not met, it throws an `IllegalArgumentException` :

```
fun getIndices(count: Int): List<Int> {
    require(count >= 0) { "Count must be non-negative. You set count to $count." }
    return List(count) { it + 1 }
}

fun main() {
    // This fails with an IllegalArgumentException
    println(getIndices(-1))

    // Uncomment the line below to see a working example
    // println(getIndices(3))
    // [1, 2, 3]
}
```

> The `require()` function allows the compiler to perform smart casting. After a successful check, the variable is automatically cast to a non-nullable type. These functions are often used for nullability checks to ensure that the variable is not null before proceeding. For example:
>
> ```
> fun printNonNullString(str: String?) {
>     // Nullability check
>     require(str != null)
>     // After this successful check, 'str' is guaranteed to be
>     // non-null and is automatically smart cast to non-nullable String
>     println(str.length)
> }
> ```
>
> 

check() function

Use the `check()` function to validate the state of an object or variable. If the check fails, it indicates a logic error that needs to be addressed.

If the condition specified in the `check()` function is `false`, it throws an `IllegalStateException`:

```
fun main() {
    var someState: String? = null

    fun getStateValue(): String {

        val state = checkNotNull(someState) { "State must be set beforehand!" }
        check(state.isNotEmpty()) { "State must be non-empty!" }
        return state
    }
    // If you uncomment the line below then the program fails with IllegalStateExcepti
on
    // getStateValue()

    someState = ""

    // If you uncomment the line below then the program fails with IllegalStateExcepti
on
    // getStateValue()
    someState = "non-empty-state"

    // This prints "non-empty-state"
    println(getStateValue())
}
```

> The `check()` function allows the compiler to perform smart casting. After a successful check, the variable is automatically cast to a non-nullable type. These functions are often used for nullability checks to ensure that the variable is not null before proceeding. For example:
>
> ```
> fun printNonNullString(str: String?) {
>     // Nullability check
>     check(str != null)
>     // After this successful check, 'str' is guaranteed to be
>     // non-null and is automatically smart cast to non-nullable String
>     println(str.length)
> }
> ```
>
> 

error() function

The `error()` function is used to signal an illegal state or a condition in the code that logically should not occur. It's suitable for scenarios when you want to throw an exception intentionally in your code, such as when the code encounters an unexpected state. This function is particularly useful in `when` expressions, providing a clear way to handle cases that shouldn't logically happen.

In the following example, the `error()` function is used to handle an undefined user role. If the role is not one of the predefined ones, an `IllegalStateException` is thrown:

```
class User(val name: String, val role: String)

fun processUserRole(user: User) {
    when (user.role) {
        "admin" -> println("${user.name} is an admin.")
        "editor" -> println("${user.name} is an editor.")
        "viewer" -> println("${user.name} is a viewer.")
        else -> error("Undefined role: ${user.role}")
    }
}

fun main() {
    // This works as expected
    val user1 = User("Alice", "admin")
    processUserRole(user1)
    // Alice is an admin.

    // This throws an IllegalStateException
    val user2 = User("Bob", "guest")
    processUserRole(user2)
}
```

Handle exceptions using try-catch blocks

When an exception is thrown, it interrupts the normal execution of the program. You can handle exceptions gracefully with the `try` and `catch` keywords to keep your program stable. The `try` block contains the code that might throw an exception, while the `catch` block catches and handles the exception if it occurs. The exception is caught by the first `catch` block that matches its specific type or a superclass of the exception.

Here's how you can use the `try` and `catch` keywords together:

```
try {
    // Code that may throw an exception
} catch (e: SomeException) {
    // Code for handling the exception
}
```

It's a common approach to use `try-catch` as an expression, so it can return a value from either the `try` block or the `catch` block:

```
fun main() {
    val num: Int = try {

        // If count() completes successfully, its return value is assigned to num
        count()

    } catch (e: ArithmeticException) {

        // If count() throws an exception, the catch block returns -1,
        // which is assigned to num
        -1
    }
    println("Result: $num")
}

// Simulates a function that might throw ArithmeticException
fun count(): Int {

    // Change this value to return a different value to num
    val a = 0

    return 10 / a
}
```

You can use multiple `catch` handlers for the same `try` block. You can add as many `catch` blocks as needed to handle different exceptions distinctively. When you have multiple `catch` blocks, it's important to order them from the most specific to the least specific exception, following a top-to-

bottom order in your code. This ordering aligns with the program's execution flow.

Consider this example with custom exceptions:

```
open class WithdrawalException(message: String) : Exception(message)
class InsufficientFundsException(message: String) : WithdrawalException(message)

fun processWithdrawal(amount: Double, availableFunds: Double) {
    if (amount > availableFunds) {
        throw InsufficientFundsException("Insufficient funds for the withdrawal.")
    }
    if (amount < 1 || amount % 1 != 0.0) {
        throw WithdrawalException("Invalid withdrawal amount.")
    }
    println("Withdrawal processed")
}

fun main() {
    val availableFunds = 500.0

    // Change this value to test different scenarios
    val withdrawalAmount = 500.5

    try {
        processWithdrawal(withdrawalAmount.toDouble(), availableFunds)

    // The order of catch blocks is important!
    } catch (e: InsufficientFundsException) {
        println("Caught an InsufficientFundsException: ${e.message}")
    } catch (e: WithdrawalException) {
        println("Caught a WithdrawalException: ${e.message}")
    }
}
```

A general catch block handling `WithdrawalException`, catches all exceptions of its type, including specific ones like `InsufficientFundsException`, unless they are caught earlier by a more specific catch block.

The finally block

The `finally` block contains code that always executes, regardless of whether the `try` block completes successfully or throws an exception. With the `finally` block you can clean up code after the execution of `try` and `catch` blocks. This is especially important when working with resources like files or network connections, as `finally` guarantees they are properly closed or released.

Here is how you would typically use the `try-catch-finally` blocks together:

```
try {
```

```
    // Code that may throw an exception
}
catch (e: YourException) {
    // Exception handler
}
finally {
    // Code that is always executed
}
```

The returned value of a `try` expression is determined by the last executed expression in either the `try` or `catch` block. If no exceptions occur, the result comes from the `try` block; if an exception is handled, it comes from the `catch` block. The `finally` block is always executed, but it doesn't change the result of the `try-catch` block.

Let's look at an example to demonstrate:

```
fun divideOrNull(a: Int): Int {

    // The try block is always executed
    // An exception here (division by zero) causes an immediate jump to the catch block

    try {
        val b = 44 / a
        println("try block: Executing division: $b")
        return b
    }

    // The catch block is executed due to the ArithmeticException (division by zero if
 a ==0)
    catch (e: ArithmeticException) {
        println("catch block: Encountered ArithmeticException $e")
        return -1
    }
    finally {
        println("finally block: The finally block is always executed")
    }
}

fun main() {

    // Change this value to get a different result. An ArithmeticException will return
: -1
    divideOrNull(0)
}
```

> In Kotlin, the idiomatic way to manage resources that implement the `AutoClosable` interface, such as file streams like `FileInputStream` or `FileOutputStream`, is to use the `.use()` function. This function automatically closes the resource when the block of code completes,

> regardless of whether an exception is thrown, thereby eliminating the need for a `finally` block. Consequently, Kotlin does not require a special syntax like Java's try-with-resources for resource management.
>
> ```
> FileWriter("test.txt").use { writer ->
> writer.write("some text")
> // After this block, the .use function automatically calls writer.close(), simila
> r to a finally block
> }
> ```



If your code requires resource cleanup without handling exceptions, you can also use `try` with the `finally` block without `catch` blocks:

```
class MockResource {
    fun use() {
        println("Resource being used")
        // Simulate a resource being used
        // This throws an ArithmeticException if division by zero occurs
        val result = 100 / 0

        // This line is not executed if an exception is thrown
        println("Result: $result")
    }

    fun close() {
        println("Resource closed")
    }
}

fun main() {
    val resource = MockResource()
//sampleStart
    try {

        // Attempts to use the resource
        resource.use()

    } finally {

        // Ensures that the resource is always closed, even if an exception occurs
        resource.close()
    }

    // This line is not printed if an exception is thrown
    println("End of the program")
//sampleEnd
}
```

As you can see, the `finally` block guarantees that the resource is closed, regardless of whether an exception occurs.

In Kotlin, you have the flexibility to use only a `catch` block, only a `finally` block, or both, depending on your specific needs, but a `try` block must always be accompanied by at least one `catch` block or a `finally` block.

Create custom exceptions

In Kotlin, you can define custom exceptions by creating classes that extend the built-in `Exception` class. This allows you to create more specific error types tailored to your application's needs.

To create one, you can define a class that extends `Exception`:

```
class MyException: Exception("My message")
```

In this example, there is a default error message, "My message", but you can leave it blank if you want.

> Exceptions in Kotlin are stateful objects, carrying information specific to the context of their creation, referred to as the stack trace. Avoid creating exceptions using object declarations. Instead, create a new instance of the exception every time you need one. This way, you can ensure the exception's state accurately reflects the specific context.

Custom exceptions can also be a subclass of any pre-existent exception subclass, like the `ArithmeticException` subclass:

```
class NumberTooLargeException: ArithmeticException("My message")
```

> If you want to create subclasses of custom exceptions, you must declare the parent class as `open` because classes are final by default and cannot be subclassed otherwise.
>
> For example:
>
> ```
> // Declares a custom exception as an open class, making it subclassable
> open class MyCustomException(message: String): Exception(message)
>
> // Creates a subclass of the custom exception
> class SpecificCustomException: MyCustomException("Specific error message")
> ```

Custom exceptions behave just like built-in exceptions. You can throw them using the `throw` keyword, and handle them with `try-catch-finally` blocks. Let's look at an example to demonstrate:

```
class NegativeNumberException: Exception("Parameter is less than zero.")
class NonNegativeNumberException: Exception("Parameter is a non-negative number.")

fun myFunction(number: Int) {
    if (number < 0) throw NegativeNumberException()
    else if (number >= 0) throw NonNegativeNumberException()
}

fun main() {

    // Change the value in this function to a get a different exception
    myFunction(1)
}
```

In applications with diverse error scenarios, creating a hierarchy of exceptions can help making the code clearer and more specific. You can achieve this by using an abstract class or a sealed class as a base for common exception features and creating specific subclasses for detailed exception types. Additionally, custom exceptions with optional parameters offer flexibility, allowing initialization with varied messages, which enables more granular error handling.

Let's look at an example using the sealed class `AccountException` as the base for an exception hierarchy, and class `APIKeyExpiredException`, a subclass, which showcases the use of optional parameters for improved exception detail:

```
//sampleStart
// Creates a sealed class as the base for an exception hierarchy for account-related e
rrors
sealed class AccountException(message: String, cause: Throwable? = null):
Exception(message, cause)

// Creates a subclass of AccountException
class InvalidAccountCredentialsException : AccountException("Invalid account credentia
ls detected")

// Creates a subclass of AccountException, which allows the addition of custom message
s and causes
class APIKeyExpiredException(message: String = "API key expired", cause: Throwable? =
null)    : AccountException(message, cause)

// Change values of placeholder functions to get different results
fun areCredentialsValid(): Boolean = true
fun isAPIKeyExpired(): Boolean = true
//sampleEnd

// Validates account credentials and API key
fun validateAccount() {
```

```
    if (!areCredentialsValid()) throw InvalidAccountCredentialsException()
    if (isAPIKeyExpired()) {
        // Example of throwing APIKeyExpiredException with a specific cause
        val cause = RuntimeException("API key validation failed due to network error")
        throw APIKeyExpiredException(cause = cause)
    }
}

fun main() {
    try {
        validateAccount()
        println("Operation successful: Account credentials and API key are valid.")
    } catch (e: AccountException) {
        println("Error: ${e.message}")
        e.cause?.let { println("Caused by: ${it.message}") }
    }
}
```

Nothing 类型

In Kotlin, every expression has a type. The type of the expression `throw IllegalArgumentException()` is `Nothing`, a built-in type that is a subtype of all other types, also known as the bottom type. This means `Nothing` can be used as a return type or generic type where any other type is expected, without causing type errors.

`Nothing` is a special type in Kotlin used to represent functions or expressions that never complete successfully, either because they always throw an exception or enter an endless execution path like an infinite loop. You can use `Nothing` to mark functions that are not yet implemented or are designed to always throw an exception, clearly indicating your intentions to both the compiler and code readers. If the compiler infers a `Nothing` type in a function signature, it will warn you. Explicitly defining `Nothing` as the return type can eliminate this warning.

This Kotlin code demonstrates the use of the `Nothing` type, where the compiler marks the code following the function call as unreachable:

```
class Person(val name: String?)

fun fail(message: String): Nothing {
    throw IllegalArgumentException(message)
    // This function will never return successfully.
    // It will always throw an exception.
}

fun main() {
    // Creates an instance of Person with 'name' as null
    val person = Person(name = null)
```

```
    val s: String = person.name ?: fail("Name required")

    // 's' is guaranteed to be initialized at this point
    println(s)
}
```

Kotlin's `TODO()` function, which also uses the `Nothing` type, serves as a placeholder to highlight areas of the code that need future implementation:

```
fun notImplementedFunction(): Int {
    TODO("This function is not yet implemented")
}

fun main() {
    val result = notImplementedFunction()
    // This throws a NotImplementedError
    println(result)
}
```

As you can see, the `TODO()` function always throws a `NotImplementedError` exception.

Exception classes

Let's explore some common exception types found in Kotlin, which are all subclasses of the `RuntimeException` class:

- `ArithmeticException` : This exception occurs when an arithmetic operation is impossible to perform, like division by zero.

  ```
  val example = 2 / 0 // throws ArithmeticException
  ```

- `IndexOutOfBoundsException` : This exception is thrown to indicate that an index of some sort, such as an array or string is out of range.

  ```
  val myList = mutableListOf(1, 2, 3)
  myList.removeAt(3)  // throws IndexOutOfBoundsException
  ```

  > To avoid this exception, use a safer alternative, such as the `getOrNull()` function:
  >
  > ```
  > val myList = listOf(1, 2, 3)
  > // Returns null, instead of IndexOutOfBoundsException
  > val element = myList.getOrNull(3)
  > println("Element at index 3: $element")
  > ```



- `NoSuchElementException` : This exception is thrown when an element that does not exist in a particular collection is accessed. It occurs when using methods that expect a specific element, such as `first()` or `last()` .

```
val emptyList = listOf<Int>()
val firstElement = emptyList.first()  // throws NoSuchElementException
```

> To avoid this exception, use a safer alternative, such as the `firstOrNull()` function:
>
> ```
> val emptyList = listOf<Int>()
> // Returns null, instead of NoSuchElementException
> val firstElement = emptyList.firstOrNull()
> println("First element in empty list: $firstElement")
> ```
>
> 

- `NumberFormatException` : This exception occurs when attempting to convert a string to a numeric type, but the string doesn't have an appropriate format.

```
val string = "This is not a number"
val number = string.toInt() // throws NumberFormatException
```

> To avoid this exception, use a safer alternative, such as the `toIntOrNull()` function:
>
> ```
> val nonNumericString = "not a number"
> // Returns null, instead of NumberFormatException
> val number = nonNumericString.toIntOrNull()
> println("Converted number: $number")
> ```
>
> 

- `NullPointerException` : This exception is thrown when an application attempts to use an object reference that has the `null` value. Even though Kotlin's null safety features significantly reduce the risk of NullPointerExceptions, they can still occur either through deliberate use of the `!!` operator or when interacting with Java, which lacks Kotlin's null safety.

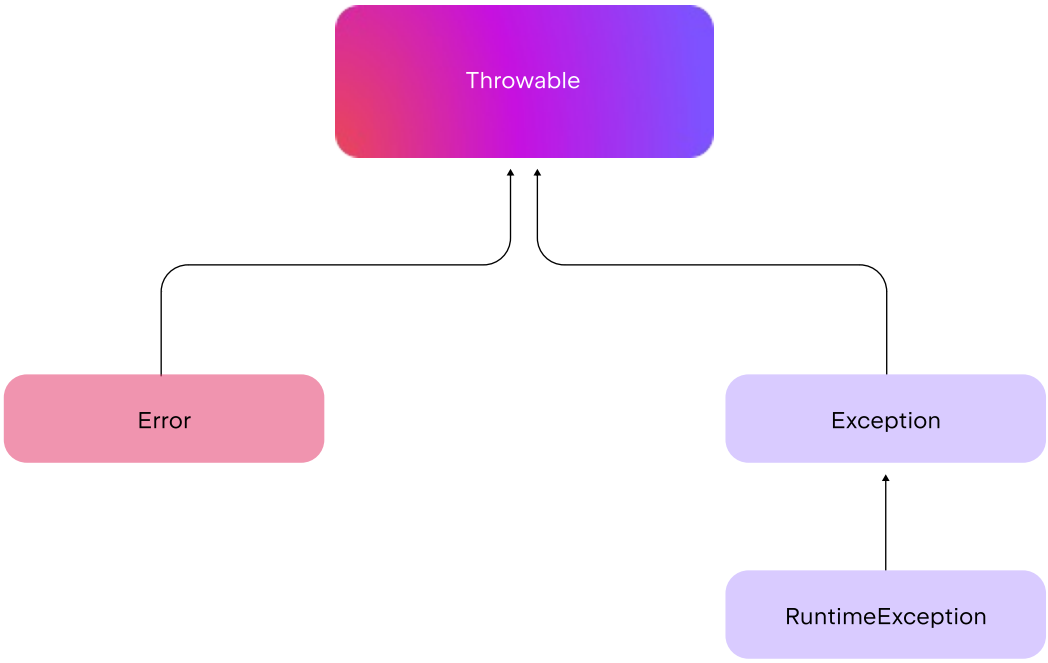
```
val text: String? = null
println(text!!.length)  // throws a NullPointerException
```

While all exceptions are unchecked in Kotlin, and you don't have to catch them explicitly, you still have the flexibility to catch them if desired.

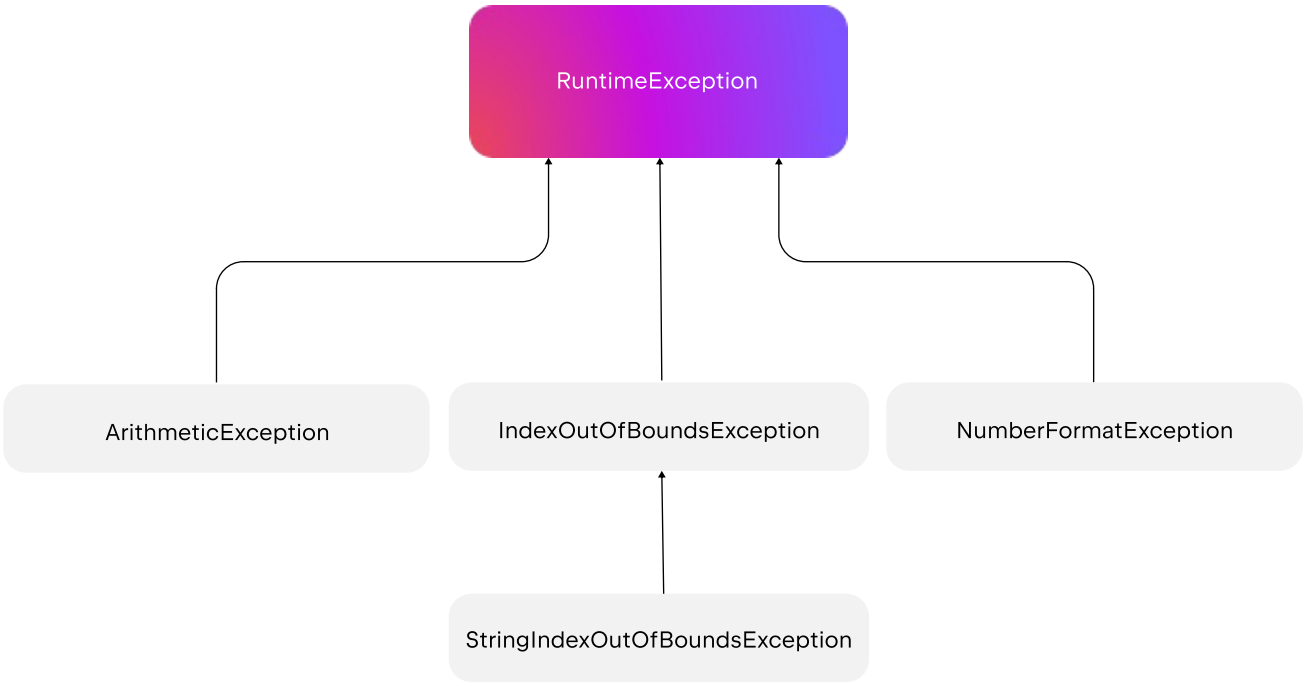
Exception hierarchy

The root of the Kotlin exception hierarchy is the `Throwable` class. It has two direct subclasses, `Error` and `Exception`:

- The `Error` subclass represents serious fundamental problems that an application might not be able to recover from by itself. These are problems that you generally would not attempt to handle, such as `OutOfMemoryError` or `StackOverflowError`.
- The `Exception` subclass is used for conditions that you might want to handle. Subtypes of the `Exception` type, such as the `RuntimeException` and `IOException` (Input/Output Exception), deal with exceptional events in applications.



`RuntimeException` is usually caused by insufficient checks in the program code and can be prevented programmatically. Kotlin helps prevent common `RuntimeExceptions` like `NullPointerException` and provides compile-time warnings for potential runtime errors, such as division by zero. The following picture demonstrates a hierarchy of subtypes descended from `RuntimeException`:



Stack trace

The *stack trace* is a report generated by the runtime environment, used for debugging. It shows the sequence of function calls leading to a specific point in the program, especially where an error or exception occurred.

Let's see an example where the stack trace is automatically printed because of an exception in a JVM environment:

```
fun main() {
//sampleStart
    throw ArithmeticException("This is an arithmetic exception!")
//sampleEnd
}
```

Running this code in a JVM environment produces the following output:

```
Exception in thread "main" java.lang.ArithmeticException: This is an arithmetic except
ion!
    at MainKt.main(Main.kt:3)
    at MainKt.main(Main.kt)
```

The first line is the exception description, which includes:

- Exception type: `java.lang.ArithmeticException`
- Thread: `main`
- Exception message: `"This is an arithmetic exception!"`

Each other line that starts with an `at` after the exception description is the stack trace. A single line is called a *stack trace element* or a *stack frame*:

- `at MainKt.main (Main.kt:3)` : This shows the method name (`MainKt.main`) and the source file and line number where the method was called (`Main.kt:3`).
- `at MainKt.main (Main.kt)` : This shows that the exception occurs in the `main()` function of the `Main.kt` file.

Exception interoperability with Java, Swift, and Objective-C

Since Kotlin treats all exceptions as unchecked, it can lead to complications when such exceptions are called from languages that distinguish between checked and unchecked exceptions. To address this disparity in exception handling between Kotlin and languages like Java, Swift, and Objective-C, you can use the `@Throws` annotation. This annotation alerts callers about possible exceptions. For more information, see Calling Kotlin from Java and Interoperability with Swift/Objective-C.

包与导入

源文件通常以包声明开头:

```
package org.example

fun printMessage() { /*……*/ }
class Message { /*……*/ }

// ……
```

源文件所有内容（无论是类还是函数）都包含在该包内。 所以上例中 `printMessage()` 的全名是 `org.example.printMessage`， 而 `Message` 的全名是 `org.example.Message`。

如果没有指明包，该文件的内容属于无名字的默认包。

默认导入

有多个包会默认导入到每个 Kotlin 文件中：

- kotlin.*
- kotlin.annotation.*
- kotlin.collections.*
- kotlin.comparisons.*
- kotlin.io.*
- kotlin.ranges.*
- kotlin.sequences.*
- kotlin.text.*

根据目标平台还会导入额外的包：

- JVM:
  - java.lang.*
  - kotlin.jvm.*
- JS:
  - kotlin.js.*

导入

除了默认导入之外，每个文件可以包含它自己的导入（`import`）指令。

可以导入一个单个名称：

```
import org.example.Message // 现在 Message 可以不用限定符访问
```

也可以导入一个作用域下的所有内容：包、类、对象等:

```
import org.example.* // “org.example”中的一切都可访问
```

如果出现名字冲突，可以使用 `as` 关键字在本地重命名冲突项来消歧义：

```
import org.example.Message // Message 可访问
import org.test.Message as TestMessage // TestMessage 代表“org.test.Message”
```

关键字 `import` 并不仅限于导入类；也可用它来导入其他声明：

- 顶层函数及属性
- 在对象声明中声明的函数和属性
- 枚举常量

顶层声明的可见性

如果顶层声明是 `private` 的，它是声明它的文件所私有的（参见 可见性修饰符）。

类与对象

- 类
- 继承
- 属性
- 接口
- 函数式（SAM）接口
- 可见性修饰符
- 扩展
- 数据类
- 密封类与密封接口
- 泛型：in、out、where
- 嵌套类
- 枚举类
- 内联类
- 对象表达式与对象声明
- 委托
- 属性委托
- 类型别名

类

Kotlin 中使用关键字 `class` 声明类

```
class Person { /*……*/ }
```

类声明由类名、类头（指定其类型参数、主构造函数等）以及由花括号包围的类体构成。类头与类体都是可选的； 如果一个类没有类体，可以省略花括号。

```
class Empty
```

构造函数

在 Kotlin 中的一个类有一个主构造函数并可能有一个或多个次构造函数。主构造函数在类头中声明，它跟在类名与可选的类型参数后。

```
class Person constructor(firstName: String) { /*……*/ }
```

如果主构造函数没有任何注解或者可见性修饰符，可以省略这个 `constructor` 关键字。

```
class Person(firstName: String) { /*……*/ }
```

The primary constructor initializes a class instance and its properties in the class header. The class header can't contain any runnable code. If you want to run some code during object creation, use *initializer blocks* inside the class body. Initializer blocks are declared with the `init` keyword followed by curly braces. Write any code that you want to run within the curly braces.

在实例初始化期间，初始化块按照它们出现在类体中的顺序执行，与属性初始化器交织在一起：

```
//sampleStart
class InitOrderDemo(name: String) {
    val firstProperty = "First property: $name".also(::println)

    init {
        println("First initializer block that prints $name")
    }

    val secondProperty = "Second property: ${name.length}".also(::println)
```

```
    init {
        println("Second initializer block that prints ${name.length}")
    }
}
//sampleEnd

fun main() {
    InitOrderDemo("hello")
}
```

主构造的参数可以在初始化块中使用。它们也可以在类体内声明的属性初始化器中使用：

```
class Customer(name: String) {
    val customerKey = name.uppercase()
}
```

Kotlin has a concise syntax for declaring properties and initializing them from the primary constructor:

```
class Person(val firstName: String, val lastName: String, var age: Int)
```

Such declarations can also include default values of the class properties:

```
class Person(val firstName: String, val lastName: String, var isEmployed: Boolean = tr
ue)
```

声明类属性时，可以使用尾部逗号：

```
class Person(
    val firstName: String,
    val lastName: String,
    var age: Int, // 尾部逗号
) { /*……*/ }
```

与普通属性一样，主构造函数中声明的属性可以是可变的（`var`）或只读的（`val`）。

如果构造函数有注解或可见性修饰符，这个 `constructor` 关键字是必需的，并且这些修饰符在它前面：

```
class Customer public @Inject constructor(name: String) { /*……*/ }
```

Learn more about visibility modifiers.

次构造函数

类也可以声明前缀有 `constructor` 的次构造函数：

```
class Person(val pets: MutableList<Pet> = mutableListOf())

class Pet {
    constructor(owner: Person) {
        owner.pets.add(this) // adds this pet to the list of its owner's pets
    }
}
```

如果类有一个主构造函数，每个次构造函数需要委托给主构造函数， 可以直接委托或者通过别的次构造函数间接委托。委托到同一个类的另一个构造函数用 `this` 关键字即可：

```
class Person(val name: String) {
    val children: MutableList<Person> = mutableListOf()
    constructor(name: String, parent: Person) : this(name) {
        parent.children.add(this)
    }
}
```

请注意，初始化块中的代码实际上会成为主构造函数的一部分。对主构造函数的委托发生在访问次构造函数的第一条语句时，因此所有初始化块与属性初始化器中的代码都会在次构造函数体之前执行。

即使该类没有主构造函数，这种委托仍会隐式发生，并且仍会执行初始化块：

```
//sampleStart
class Constructors {
    init {
        println("Init block")
    }

    constructor(i: Int) {
        println("Constructor $i")
    }
}
//sampleEnd

fun main() {
    Constructors(1)
}
```

如果一个非抽象类没有声明任何（主或次）构造函数，它会有一个生成的不带参数的主构造函数。构造函数的可见性是 public。

如果你不希望你的类有一个公有构造函数，那么声明一个带有非默认可见性的空的主构造函数：

```
class DontCreateMe private constructor() { /*……*/ }
```

> 在 JVM 上，如果主构造函数的所有的参数都有默认值，编译器会生成一个额外的无参构造函数，它将使用默认值。这使得 Kotlin 更易于使用像 Jackson 或者 JPA 这样的通过无参构造函数创建类的实例的库。
>
> ```
> class Customer(val customerName: String = "")
> ```
>
> 

创建类的实例

创建一个类的实例，只需像普通函数一样调用构造函数。You can assign the created instance to a variable:

```
val invoice = Invoice()

val customer = Customer("Joe Smith")
```

> Kotlin does not have a `new` keyword.
>
> 

创建嵌套类、内部类与匿名内部类的类实例的过程在嵌套类中有述。

类成员

类可以包含：

- 构造函数与初始化块
- 函数
- 属性
- 嵌套类与内部类
- 对象声明

继承

Classes can be derived from each other and form inheritance hierarchies. Learn more about inheritance in Kotlin.

抽象类

类以及其中的某些或全部成员可以声明为 `abstract`。 抽象成员在本类中可以不用实现。 并不需要用 `open` 标注抽象类或者函数。

```
abstract class Polygon {
    abstract fun draw()
}

class Rectangle : Polygon() {
    override fun draw() {
        // draw the rectangle
    }
}
```

可以用一个抽象成员覆盖一个非抽象的开放成员。

```
open class Polygon {
    open fun draw() {
        // some default polygon drawing method
    }
}

abstract class WildShape : Polygon() {
    // Classes that inherit WildShape need to provide their own
    // draw method instead of using the default on Polygon
    abstract override fun draw()
}
```

伴生对象

如果你需要写一个可以无需用一个类的实例来调用、但需要访问类内部的函数（例如，工厂方法），你可以把它写成该类内对象声明中的一员。

更具体地讲，如果在你的类内声明了一个伴生对象， 你就可以访问其成员，只是以类名作为限定符。

继承

在 Kotlin 中所有类都有一个共同的超类 `Any`，对于没有超类型声明的类它是默认超类：

```
class Example // 从 Any 隐式继承
```

`Any` 有三个方法：`equals()`、`hashCode()` 与 `toString()`。因此，为所有 Kotlin 类都定义了这些方法。

默认情况下，Kotlin 类是最终（final）的——它们不能被继承。 要使一个类可继承，请用 `open` 关键字标记它：

```
open class Base // 该类开放继承
```

如需声明一个显式的超类型，请在类头中把超类型放到冒号之后：

```
open class Base(p: Int)

class Derived(p: Int) : Base(p)
```

如果派生类有一个主构造函数，其基类可以（并且必须）根据其参数在该主构造函数中初始化。

如果派生类没有主构造函数，那么每个次构造函数必须使用 `super` 关键字初始化其基类型，或委托给另一个做到这点的构造函数。 请注意，在这种情况下，不同的次构造函数可以调用基类型的不同的构造函数：

```
class MyView : View {
    constructor(ctx: Context) : super(ctx)

    constructor(ctx: Context, attrs: AttributeSet) : super(ctx, attrs)
}
```

覆盖方法

Kotlin 对于可覆盖的成员以及覆盖后的成员需要显式修饰符：

```
open class Shape {
    open fun draw() { /*……*/ }
    fun fill() { /*……*/ }
```

```
}

class Circle() : Shape() {
    override fun draw() { /*……*/ }
}
```

`Circle.draw()` 函数上必须加上 `override` 修饰符。如果没写，编译器会报错。 如果函数没有标注 `open` 如 `Shape.fill()`，那么子类中不允许定义相同签名的函数， 无论加不加 `override`。将 `open` 修饰符添加到 final 类（即没有 `open` 的类）的成员上不起作用。

标记为 `override` 的成员本身是开放的，因此可以在子类中覆盖。如果你想禁止再次覆盖，使用 `final` 关键字：

```
open class Rectangle() : Shape() {
    final override fun draw() { /*……*/ }
}
```

覆盖属性

属性与方法的覆盖机制相同。在超类中声明然后在派生类中重新声明的属性必须以 `override` 开头，并且它们必须具有兼容的类型。 每个声明的属性可以由具有初始化器的属性或者具有 `get` 方法的属性覆盖：

```
open class Shape {
    open val vertexCount: Int = 0
}

class Rectangle : Shape() {
    override val vertexCount = 4
}
```

你也可以用一个 `var` 属性覆盖一个 `val` 属性，但反之则不行。 这是允许的，因为一个 `val` 属性本质上声明了一个 `get` 方法， 而将其覆盖为 `var` 只是在子类中额外声明一个 `set` 方法。

请注意，你可以在主构造函数中使用 `override` 关键字作为属性声明的一部分：

```
interface Shape {
    val vertexCount: Int
}

class Rectangle(override val vertexCount: Int = 4) : Shape // 总是有 4 个顶点

class Polygon : Shape {
    override var vertexCount: Int = 0  // 以后可以设置为任何数
```

```
}
```

派生类初始化顺序

在构造派生类的新实例的过程中，第一步完成其基类的初始化（在之前只有对基类构造函数参数的求值），这意味着它发生在派生类的初始化逻辑运行之前。

```
//sampleStart
open class Base(val name: String) {

    init { println("Initializing a base class") }

    open val size: Int =
        name.length.also { println("Initializing size in the base class: $it") }
}

class Derived(
    name: String,
    val lastName: String,
) : Base(name.replaceFirstChar { it.uppercase() }.also { println("Argument for the bas
e class: $it") }) {

    init { println("Initializing a derived class") }

    override val size: Int =
        (super.size + lastName.length).also { println("Initializing size in the derive
d class: $it") }
}
//sampleEnd

fun main() {
    println("Constructing the derived class(\"hello\", \"world\")")
    Derived("hello", "world")
}
```

这意味着，基类构造函数执行时，派生类中声明或覆盖的属性都还没有初始化。在基类初始化逻辑中（直接或者通过另一个覆盖的 `open` 成员的实现间接）使用任何一个这种属性，都可能导致不正确的行为或运行时故障。 设计一个基类时，应该避免在构造函数、属性初始化器或者 `init` 块中使用 `open` 成员。

调用超类实现

派生类中的代码可以使用 `super` 关键字调用其超类的函数与属性访问器的实现：

```
open class Rectangle {
```

```
    open fun draw() { println("Drawing a rectangle") }
    val borderColor: String get() = "black"
}

class FilledRectangle : Rectangle() {
    override fun draw() {
        super.draw()
        println("Filling the rectangle")
    }

    val fillColor: String get() = super.borderColor
}
```

在一个内部类中访问外部类的超类，可以使用由外部类名限定的 `super` 关键字来实现：`super@Outer`：

```
open class Rectangle {
    open fun draw() { println("Drawing a rectangle") }
    val borderColor: String get() = "black"
}

//sampleStart
class FilledRectangle: Rectangle() {
    override fun draw() {
        val filler = Filler()
        filler.drawAndFill()
    }

    inner class Filler {
        fun fill() { println("Filling") }
        fun drawAndFill() {
            super@FilledRectangle.draw() // 调用 Rectangle 的 draw() 实现
            fill()
            println("Drawn a filled rectangle with color ${super@FilledRectangle.borde
rColor}") // 使用 Rectangle 所实现的 borderColor 的 get()
        }
    }
}
//sampleEnd

fun main() {
    val fr = FilledRectangle()
        fr.draw()
}
```

覆盖规则

在 Kotlin 中，实现继承由下述规则规定：如果一个类从它的直接超类继承相同成员的多个实现， 它必须覆盖这个成员并提供其自己的实现（也许用继承来的其中之一）。

如需表示采用从哪个超类型继承的实现，请使用由尖括号中超类型名限定的 `super` ，如 `super<Base>`：

```
open class Rectangle {
    open fun draw() { /* …… */ }
}

interface Polygon {
    fun draw() { /* …… */ } // 接口成员默认就是“open”的
}

class Square() : Rectangle(), Polygon {
    // 编译器要求覆盖 draw():
    override fun draw() {
        super<Rectangle>.draw() // 调用 Rectangle.draw()
        super<Polygon>.draw() // 调用 Polygon.draw()
    }
}
```

可以同时继承 `Rectangle` 与 `Polygon`， 但是二者都有各自的 `draw()` 实现，所以必须在 `Square` 中覆盖 `draw()`， 并为其提供一个单独的实现以消除歧义。

属性

声明属性

Kotlin 类中的属性既可以用关键字 `var` 声明为可变的， 也可以用关键字 `val` 声明为只读的。

```
class Address {
    var name: String = "Holmes, Sherlock"
    var street: String = "Baker"
    var city: String = "London"
    var state: String? = null
    var zip: String = "123456"
}
```

使用一个属性，以其名称引用它即可：

```
fun copyAddress(address: Address): Address {
    val result = Address() // Kotlin 中没有“new”关键字
    result.name = address.name // 将调用访问器
    result.street = address.street
    // ……
    return result
}
```

Getter 与 Setter

声明一个属性的完整语法如下：

```
var <propertyName>[: <PropertyType>] [= <property_initializer>]
    [<getter>]
    [<setter>]
```

其初始器（initializer）、getter 和 setter 都是可选的。属性类型如果可以从初始器， 或其 getter 的返回值（如下文所示）中推断出来，也可以省略：

```
var initialized = 1 // 类型 Int、默认 getter 和 setter
// var allByDefault // 错误: 需要显式初始化器, 隐含默认 getter 和 setter
```

一个只读属性的语法和一个可变的属性的语法有两方面的不同： 1、只读属性的用 `val` 而不是 `var` 声明 2、只读属性不允许 setter

```
val simple: Int? // 类型 Int、默认 getter、必须在构造函数中初始化
val inferredType = 1 // 类型 Int 、默认 getter
```

可以为属性定义自定义的访问器。如果定义了一个自定义的 getter，那么每次访问该属性时都会调用它（这让可以实现计算出的属性）。以下是一个自定义 getter 的示例：

```
//sampleStart
class Rectangle(val width: Int, val height: Int) {
    val area: Int // property type is optional since it can be inferred from the gette
r's return type
        get() = this.width * this.height
}
//sampleEnd
fun main() {
    val rectangle = Rectangle(3, 4)
    println("Width=${rectangle.width}, height=${rectangle.height}, area=${rectangle.ar
ea}")
}
```

如果可以从 getter 推断出属性类型，则可以省略它：

```
val area get() = this.width * this.height
```

如果定义了一个自定义的 setter，那么每次给属性赋值时都会调用它, except its initialization. 一个自定义的 setter 如下所示：

```
var stringRepresentation: String
    get() = this.toString()
    set(value) {
        setDataFromString(value) // 解析字符串并赋值给其他属性
    }
```

按照惯例，setter 参数的名称是 `value`，但是如果你喜欢你可以选择一个不同的名称。

如果需要改变对一个访问器进行注解或者改变其可见性，但是不希望改变默认的实现， 那么可以定义访问器而不定义其实现:

```
var setterVisibility: String = "abc"
    private set // 此 setter 是私有的并且有默认实现

var setterWithAnnotation: Any? = null
    @Inject set // 用 Inject 注解此 setter
```

幕后字段

在 Kotlin 中，字段仅作为属性的一部分在内存中保存其值时使用。字段不能直接声明。 然而，当一个属性需要一个幕后字段时，Kotlin 会自动提供。这个幕后字段可以使用 `field` 标识符在访问器中引用：

```
var counter = 0 // 这个初始器直接为幕后字段赋值
    set(value) {
        if (value >= 0)
            field = value
            // counter = value // ERROR StackOverflow: Using actual name 'counter' wou
ld make setter recursive
    }
```

`field` 标识符只能用在属性的访问器内。

如果属性至少一个访问器使用默认实现， 或者自定义访问器通过 `field` 引用幕后字段，将会为该属性生成一个幕后字段。

例如，以下情况下就没有幕后字段：

```
val isEmpty: Boolean
    get() = this.size == 0
```

幕后属性

如果你的需求不符合这套隐式的幕后字段方案， 那么总可以使用 幕后属性（*backing property*）：

```
private var _table: Map<String, Int>? = null
public val table: Map<String, Int>
    get() {
        if (_table == null) {
            _table = HashMap() // 类型参数已推断出
        }
        return _table ?: throw AssertionError("Set to null by another thread")
    }
```

> 对于 JVM 平台：通过默认 getter 和 setter 访问私有属性会被优化以避免函数调用开销。

编译期常量

如果只读属性的值在编译期是已知的，那么可以使用 `const` 修饰符将其标记为编译期常量。这种属性需要满足以下要求：

- 必须位于顶层或者是 `object` 声明 或伴生对象的一个成员
- 必须以 `String` 或原生类型值初始化
- 不能有自定义 getter

The compiler will inline usages of the constant, replacing the reference to the constant with its actual value. However, the field will not be removed and therefore can be interacted with using reflection.

这些属性也可以用在注解中：

```
const val SUBSYSTEM_DEPRECATED: String = "This subsystem is deprecated"

@Deprecated(SUBSYSTEM_DEPRECATED) fun foo() { …… }
```

延迟初始化属性与变量

一般地，属性声明为非空类型必须在构造函数中初始化。 然而，这经常不方便。例如：属性可以通过依赖注入来初始化， 或者在单元测试的 setup 方法中初始化。 这种情况下，你不能在构造函数内提供一个非空初始器。 但你仍然想在类体中引用该属性时避免空检测。

为处理这种情况，你可以用 `lateinit` 修饰符标记该属性：

```
public class MyTest {
    lateinit var subject: TestSubject

    @SetUp fun setup() {
        subject = TestSubject()
    }

    @Test fun test() {
        subject.method()  // 直接解引用
    }
}
```

该修饰符只能用于在类体中的属性（不是在主构造函数中声明的 `var` 属性， 并且仅当该属性没有自定义 getter 或 setter 时），也用于顶层属性与局部变量。 该属性或变量必须为非空类型，并且不能是原生类型。

在初始化前访问一个 `lateinit` 属性会抛出一个特定异常，该异常明确标识该属性被访问及它没有初始化的事实。

检测一个 lateinit var 是否已初始化

要检测一个 `lateinit var` 是否已经初始化过，请在该属性的引用上使用 `.isInitialized`：

```
if (foo::bar.isInitialized) {
    println(foo.bar)
}
```

此检测仅对可词法级访问的属性可用，当声明位于同一个类型内、位于其中一个外围类型中或者位于相同文件的顶层的属性时。

覆盖属性

参见覆盖属性

委托属性

最常见的一类属性就是简单地从幕后字段中读取（以及可能的写入），但是使用自定义 getter 和 setter 可以实现属性的任何行为。介于最简单的第一类与多样的第二类之间，属性可做的事情有一些常见的模式。一些示例：惰性值、通过键值从映射（map）读取、访问数据库、访问时通知侦听器。

这些常见行为可以通过使用委托属性实现为库。

接口

Kotlin 的接口可以既包含抽象方法的声明也包含实现。与抽象类不同的是，接口无法保存状态。它可以有属性但必须声明为抽象或提供访问器实现。

使用关键字 `interface` 来定义接口：

```
interface MyInterface {
    fun bar()
    fun foo() {
      // 可选的方法体
    }
}
```

实现接口

一个类或者对象可以实现一个或多个接口：

```
class Child : MyInterface {
    override fun bar() {
        // 方法体
    }
}
```

接口中的属性

可以在接口中定义属性。在接口中声明的属性要么是抽象的，要么提供访问器的实现。在接口中声明的属性不能有幕后字段（backing field），因此接口中声明的访问器不能引用它们：

```
interface MyInterface {
    val prop: Int // 抽象的

    val propertyWithImplementation: String
        get() = "foo"

    fun foo() {
        print(prop)
    }
}

class Child : MyInterface {
    override val prop: Int = 29
```

```
}
```

接口继承

一个接口可以从其他接口派生，意味着既能提供基类型成员的实现也能声明新的函数与属性。很自然地，实现这样接口的类只需定义所缺少的实现：

```
interface Named {
    val name: String
}

interface Person : Named {
    val firstName: String
    val lastName: String

    override val name: String get() = "$firstName $lastName"
}

data class Employee(
    // 不必实现“name”
    override val firstName: String,
    override val lastName: String,
    val position: Position
) : Person
```

解决覆盖冲突

实现多个接口时，可能会遇到同一方法继承多个实现的问题：

```
interface A {
    fun foo() { print("A") }
    fun bar()
}

interface B {
    fun foo() { print("B") }
    fun bar() { print("bar") }
}

class C : A {
    override fun bar() { print("bar") }
}

class D : A, B {
    override fun foo() {
        super<A>.foo()
```

```
            super<B>.foo()
        }

        override fun bar() {
            super<B>.bar()
        }
    }
```

上例中，接口 *A* 和 *B* 都定义了方法 *foo()* 和 *bar()*。 两者都实现了 *foo()*, 但是只有 *B* 实现了 *bar()* (*bar()* 在 *A* 中没有标记为抽象， 因为在接口中没有方法体时默认为抽象) 。 现在，如果实现 *A* 的一个具体类 *C*，那么必须要重写 *bar()* 并实现这个抽象方法。

然而，如果从 *A* 和 *B* 派生 *D*，需要实现从多个接口继承的所有方法，并指明 *D* 应该如何实现它们。这一规则既适用于继承单个实现 (*bar()*) 的方法也适用于继承多个实现 (*foo()*) 的方法。

函数式（SAM）接口

只有一个抽象成员函数的接口称为函数式接口或 单一抽象方法（*SAM*）接口。函数式接口可以有多个非抽象成员函数，但只能有一个抽象成员函数。

可以用 `fun` 修饰符在 Kotlin 中声明一个函数式接口。

```
fun interface KRunnable {
   fun invoke()
}
```

SAM 转换

对于函数式接口，可以通过 lambda 表达式实现 SAM 转换，从而使代码更简洁、更有可读性。

使用 lambda 表达式可以替代手动创建实现函数式接口的类。 通过 SAM 转换， Kotlin can convert any lambda expression whose signature matches the signature of the interface's single method into the code, which dynamically instantiates the interface implementation.

例如，有这样一个 Kotlin 函数式接口：

```
fun interface IntPredicate {
   fun accept(i: Int): Boolean
}
```

如果不使用 SAM 转换，那么你需要像这样编写代码：

```
// 创建一个类的实例
val isEven = object : IntPredicate {
   override fun accept(i: Int): Boolean {
       return i % 2 == 0
   }
}
```

通过利用 Kotlin 的 SAM 转换，可以改为以下等效代码：

```
// 通过 lambda 表达式创建一个实例
val isEven = IntPredicate { it % 2 == 0 }
```

可以通过更短的 lambda 表达式替换所有不必要的代码。

```
fun interface IntPredicate {
   fun accept(i: Int): Boolean
}

val isEven = IntPredicate { it % 2 == 0 }

fun main() {
   println("Is 7 even? - ${isEven.accept(7)}")
}
```

你也可以使用 Java 接口上的 SAM 转换。

Migration from an interface with constructor function to a functional interface

Starting from 1.6.20, Kotlin supports callable references to functional interface constructors, which adds a source-compatible way to migrate from an interface with a constructor function to a functional interface. Consider the following code:

```
interface Printer {
    fun print()
}

fun Printer(block: () -> Unit): Printer = object : Printer { override fun print() = bl
ock() }
```

With callable references to functional interface constructors enabled, this code can be replaced with just a functional interface declaration:

```
fun interface Printer {
    fun print()
}
```

Its constructor will be created implicitly, and any code using the `::Printer` function reference will compile. For example:

```
documentsStorage.addPrinter(::Printer)
```

Preserve the binary compatibility by marking the legacy function `Printer` with the `@Deprecated` annotation with `DeprecationLevel.HIDDEN`:

```
@Deprecated(message = "Your message about the deprecation", level = DeprecationLevel.H
IDDEN)
fun Printer(...) {...}
```

函数式接口与类型别名比较

You can also simply rewrite the above using a type alias for a functional type:

```
typealias IntPredicate = (i: Int) -> Boolean

val isEven: IntPredicate = { it % 2 == 0 }

fun main() {
   println("Is 7 even? - ${isEven(7)}")
}
```

However, 函数式接口和类型别名用途并不相同。 类型别名只是现有类型的名称——它们不会创建新的类型，而函数式接口却会创建新类型。 You can provide extensions that are specific to a particular functional interface to be inapplicable for plain functions or their type aliases.

类型别名只能有一个成员，而函数式接口可以有多个非抽象成员函数以及一个抽象成员函数。 函数式接口还可以实现以及继承其他接口。

函数式接口比类型别名更灵活并且提供了更多的功能, but they can be more costly both syntactically and at runtime because they can require conversions to a specific interface. When you choose which one to use in your code, consider your needs:

- If your API needs to accept a function (any function) with some specific parameter and return types – use a simple functional type or define a type alias to give a shorter name to the corresponding functional type.
- If your API accepts a more complex entity than a function – for example, it has non-trivial contracts and/or operations on it that can't be expressed in a functional type's signature – declare a separate functional interface for it.

可见性修饰符

类、对象、接口、构造函数、方法与属性及其 setter 都可以有可见性修饰符。 getter 总是与属性有着相同的可见性。

在 Kotlin 中有这四个可见性修饰符：`private`、`protected`、`internal` 和 `public`。 默认可见性是 `public`。

在本页可以学到这些修饰符如何应用到不同类型的声明作用域。

包

函数、属性和类、对象和接口可以直接在包内的顶层声明：

```
// 文件名: example.kt
package foo

fun baz() { …… }
class Bar { …… }
```

- 如果你不使用任何可见性修饰符，默认为 `public`，这意味着你的声明将随处可见。
- 如果你声明为 `private`，它只会在声明它的文件内可见。
- 如果你声明为 `internal`，它会在相同模块内随处可见。
- `protected` 修饰符不适用于顶层声明。

> 要使用另一包中可见的顶层声明，需要将其导入进来。
>
> 

例如:

```
// 文件名: example.kt
package foo

private fun foo() { …… } // 在 example.kt 内可见

public var bar: Int = 5 // 该属性随处可见
    private set         // setter 只在 example.kt 内可见

internal val baz = 6    // 相同模块内可见
```

类成员

对于类内部声明的成员：

- `private` 意味着只该成员在这个类内部（包含其所有成员）可见；
- `protected` 意味着该成员具有与 `private` 一样的可见性，但也在子类中可见。
- `internal` 意味着能见到类声明的本模块内的任何客户端都可见其 `internal` 成员。
- `public` 意味着能见到类声明的任何客户端都可见其 `public` 成员。

> 在 Kotlin 中，外部类不能访问内部类的 private 成员。
>
> 

如果你覆盖一个 `protected` 或 `internal` 成员并且没有显式指定其可见性，该成员还会具有与原始成员相同的可见性。

例子:

```
open class Outer {
    private val a = 1
    protected open val b = 2
    internal open val c = 3
    val d = 4  // 默认 public

    protected class Nested {
        public val e: Int = 5
    }
}

class Subclass : Outer() {
    // a 不可见
    // b、c、d 可见
    // Nested 和 e 可见

    override val b = 5   // “b”为 protected
    override val c = 7   // 'c' is internal
}

class Unrelated(o: Outer) {
    // o.a、o.b 不可见
    // o.c 和 o.d 可见（相同模块）
    // Outer.Nested 不可见，Nested::e 也不可见
}
```

构造函数

使用以下语法来指定一个类的的主构造函数的可见性：

> You need to add an explicit `constructor` keyword.
>
> 

```
class C private constructor(a: Int) { …… }
```

这里的构造函数是私有（ `private` ）的。默认情况下，所有构造函数都是公开（ `public` ）的，这实际上等于类可见的地方它就可见（即 一个 `internal` 类的构造函数只能在相同模块内可见).

For sealed classes, constructors are `protected` by default. For more information, see Sealed classes.

局部声明

局部变量、函数和类不能有可见性修饰符。

模块

可见性修饰符 `internal` 意味着该成员只在相同模块内可见。更具体地说， 一个模块是编译在一起的一套 Kotlin 文件，例如：

- 一个 IntelliJ IDEA 模块
- 一个 Maven 项目
- 一个 Gradle 源代码集（例外是 `test` 源代码集可以访问 `main` 的 internal 声明）
- 一次 `<kotlinc>` Ant 任务执行所编译的一套文件

扩展

Kotlin 能够对一个类或接口扩展新功能而无需继承该类或者使用像装饰者这样的设计模式。 这通过叫做扩展的特殊声明完成。

例如，你可以为一个你不能修改的、来自第三方库中的类或接口编写一个新的函数。 这个新增的函数就像那个原始类本来就有的函数一样，可以用寻常方式调用。 这种机制称为扩展函数。此外，也有扩展属性， 允许你为一个已经存在的类添加新的属性。

扩展函数

声明一个扩展函数需用一个接收者类型也就是被扩展的类型来作为他的前缀。 下面代码为 `MutableList<Int>` 添加一个 `swap` 函数：

```
fun MutableList<Int>.swap(index1: Int, index2: Int) {
    val tmp = this[index1] // “this”对应该列表
    this[index1] = this[index2]
    this[index2] = tmp
}
```

这个 `this` 关键字在扩展函数内部对应到接收者对象（传过来的在点符号前的对象）现在，可以对任意 `MutableList<Int>` 调用该函数了：

```
val list = mutableListOf(1, 2, 3)
list.swap(0, 2) // “swap()”内部的“this”会保存“list”的值
```

这个函数对任何 `MutableList<T>` 起作用，可以泛化它：

```
fun <T> MutableList<T>.swap(index1: Int, index2: Int) {
    val tmp = this[index1] // “this”对应该列表
    this[index1] = this[index2]
    this[index2] = tmp
}
```

为了在接收者类型表达式中使用泛型，需要在函数名前声明泛型参数。 For more information about generics, 参见泛型函数。

扩展是静态解析的

扩展不能真正的修改他们所扩展的类。通过定义一个扩展，并没有在一个类中插入新成员，只不过是可以通过该类型的变量用点表达式去调用这个新函数。

Extension functions are dispatched *statically*. So which extension function is called is already known at compile time based on the receiver type. For example:

```
fun main() {
//sampleStart
    open class Shape
    class Rectangle: Shape()

    fun Shape.getName() = "Shape"
    fun Rectangle.getName() = "Rectangle"

    fun printClassName(s: Shape) {
        println(s.getName())
    }

    printClassName(Rectangle())
//sampleEnd
}
```

这个例子会输出 *Shape*，因为调用的扩展函数只取决于参数 `s` 的声明类型，该类型是 `Shape` 类。

如果一个类定义有一个成员函数与一个扩展函数，而这两个函数又有相同的接收者类型、 相同的名字，并且都适用给定的参数，这种情况总是取成员函数。 例如：

```
fun main() {
//sampleStart
    class Example {
        fun printFunctionType() { println("Class method") }
    }

    fun Example.printFunctionType() { println("Extension function") }

    Example().printFunctionType()
//sampleEnd
}
```

这段代码输出 *Class method*。

当然，扩展函数重载同样名字但不同签名成员函数也完全可以：

```
fun main() {
//sampleStart
    class Example {
        fun printFunctionType() { println("Class method") }
```

```
    }

    fun Example.printFunctionType(i: Int) { println("Extension function #$i") }

    Example().printFunctionType(1)
//sampleEnd
}
```

可空接收者

注意可以为可空的接收者类型定义扩展。这样的扩展可以在对象变量上调用， 即使其值为 null。 If the receiver is `null`, then `this` is also `null`. So when defining an extension with a nullable receiver type, we recommend performing a `this == null` check inside the function body to avoid compiler errors.

可以在没有检测 null 的时候调用 Kotlin 中的 `toString()`：检测已发生在扩展函数的内部：

```
fun Any?.toString(): String {
    if (this == null) return "null"
    // 空检测之后，“this”会自动转换为非空类型，所以下面的 toString()
    // 解析为 Any 类的成员函数
    return toString()
}
```

扩展属性

与扩展函数类似，Kotlin 支持扩展属性：

```
val <T> List<T>.lastIndex: Int
    get() = size - 1
```

> 由于扩展没有实际的将成员插入类中，因此对扩展属性来说幕后字段是无效的。这就是为什么扩展属性不能有初始化器。他们的行为只能由显式提供的 getter/setter 定义。

例如:

```
val House.number = 1 // 错误：扩展属性不能有初始化器
```

伴生对象的扩展

如果一个类定义有一个伴生对象，你也可以为伴生对象定义扩展函数与属性。就像伴生对象的常规成员一样，可以只使用类名作为限定符来调用伴生对象的扩展成员：

```
class MyClass {
    companion object { }  // 将被称为 "Companion"
}

fun MyClass.Companion.printCompanion() { println("companion") }

fun main() {
    MyClass.printCompanion()
}
```

扩展的作用域

大多数情况都在顶层定义扩展——直接在包里：

```
package org.example.declarations

fun List<String>.getLongestString() { /*……*/}
```

如需使用所定义包之外的一个扩展，只需在调用方导入它：

```
package org.example.usage

import org.example.declarations.getLongestString

fun main() {
    val list = listOf("red", "green", "blue")
    list.getLongestString()
}
```

更多信息参见导入

扩展声明为成员

可以在一个类内部为另一个类声明扩展。在这样的扩展内部，有多个隐式接收者—— 其中的对象成员可以无需通过限定符访问。扩展声明所在的类的实例称为分发接收者，扩展方法调用所在的接收者类型的实例称为扩展接收者。

```
class Host(val hostname: String) {
    fun printHostname() { print(hostname) }
}
```

```
class Connection(val host: Host, val port: Int) {
    fun printPort() { print(port) }

    fun Host.printConnectionString() {
         printHostname()   // 调用 Host.printHostname()
        print(":")
         printPort()   // 调用 Connection.printPort()
    }

    fun connect() {
         /*……*/
         host.printConnectionString()   // 调用扩展函数
    }
}

fun main() {
    Connection(Host("kotl.in"), 443).connect()
    //Host("kotl.in").printConnectionString()  // 错误，该扩展函数在 Connection 外不可用
}
```

对于分发接收者与扩展接收者的成员名字冲突的情况，扩展接收者优先。要引用分发接收者的成员你可以使用 限定的 `this` 语法。

```
class Connection {
    fun Host.getConnectionString() {
        toString()         // 调用 Host.toString()
        this@Connection.toString()  // 调用 Connection.toString()
    }
}
```

声明为成员的扩展可以声明为 `open` 并在子类中覆盖。这意味着这些函数的分发对于分发接收者类型是虚拟的，但对于扩展接收者类型是静态的。

```
open class Base { }

class Derived : Base() { }

open class BaseCaller {
    open fun Base.printFunctionInfo() {
        println("Base extension function in BaseCaller")
    }

    open fun Derived.printFunctionInfo() {
        println("Derived extension function in BaseCaller")
    }

    fun call(b: Base) {
        b.printFunctionInfo()   // 调用扩展函数
```

```
    }
}

class DerivedCaller: BaseCaller() {
    override fun Base.printFunctionInfo() {
        println("Base extension function in DerivedCaller")
    }

    override fun Derived.printFunctionInfo() {
        println("Derived extension function in DerivedCaller")
    }
}

fun main() {
    BaseCaller().call(Base())   // “Base extension function in BaseCaller”
    DerivedCaller().call(Base())  // “Base extension function in DerivedCaller”——分发接收者虚拟解析
    DerivedCaller().call(Derived())  // “Base extension function in DerivedCaller”——扩展接收者静态解析
}
```

关于可见性的说明

Extensions utilize the same visibility modifiers as regular functions declared in the same scope would. For example:

- 在文件顶层声明的扩展可以访问同一文件中的其他 `private` 顶层声明。
- 如果扩展是在其接收者类型外部声明的，那么它不能访问接收者的 `private` 或 `protected` 成员。

数据类

Data classes in Kotlin are primarily used to hold data. For each data class, the compiler automatically generates additional member functions that allow you to print an instance to readable output, compare instances, copy instances, and more. Data classes are marked with `data`:

```
data class User(val name: String, val age: Int)
```

编译器自动从主构造函数中声明的所有属性导出以下成员：

- `equals()` / `hashCode()` 对。
- `toString()` 格式是 `"User(name=John, age=42)"`。
- `componentN()` 函数 按声明顺序对应于所有属性。
- `copy()` 函数（见下文）

为了确保生成的代码的一致性以及有意义的行为，数据类必须满足以下要求：

- 主构造函数必须至少有一个参数。
- 主构造函数的所有参数必须标记为 `val` 或 `var`。
- 数据类不能是抽象、开放、密封或者内部的。

此外，数据类成员的生成遵循关于成员继承的这些规则：

- 如果在数据类体中有显式实现 `equals()`、`hashCode()` 或者 `toString()`，或者这些函数在父类中有 `final` 实现，那么不会生成这些函数，而会使用现有函数。
- 如果超类型具有 `open` 的 `componentN()` 函数并且返回兼容的类型，那么会为数据类生成相应的函数，并覆盖超类的实现。如果超类型的这些函数由于签名不兼容或者是 final 而导致无法覆盖，那么会报错。
- 不允许为 `componentN()` 以及 `copy()` 函数提供显式实现。

数据类可以扩展其他类（示例请参见密封类）。

> 在 JVM 中，如果生成的类需要含有一个无参的构造函数，那么属性必须指定默认值。（参见构造函数）：
>
> ```
> data class User(val name: String = "", val age: Int = 0)
> ```

在类体中声明的属性

请注意，对于那些自动生成的函数，编译器只使用在主构造函数内部定义的属性。 如需在生成的实现中排除一个属性，请将其声明在类体中：

```
data class Person(val name: String) {
    var age: Int = 0
}
```

In the example below, only the `name` property is used by default inside the `toString()`, `equals()`, `hashCode()`, and `copy()` implementations, and there is only one component function, `component1()`. The `age` property is declared inside the class body and is excluded. Therefore, two `Person` objects with the same `name` but different `age` values are considered equal since `equals()` only evaluates properties from the primary constructor:

```
data class Person(val name: String) {
    var age: Int = 0
}
fun main() {
//sampleStart
    val person1 = Person("John")
    val person2 = Person("John")
    person1.age = 10
    person2.age = 20

    println("person1 == person2: ${person1 == person2}")
    // person1 == person2: true

    println("person1 with age ${person1.age}: ${person1}")
    // person1 with age 10: Person(name=John)

    println("person2 with age ${person2.age}: ${person2}")
    // person2 with age 20: Person(name=John)
//sampleEnd
}
```

复制

Use the `copy()` function to copy an object, allowing you to alter *some* of its properties while keeping the rest unchanged. The implementation of this function for the `User` class above would be as follows:

```
fun copy(name: String = this.name, age: Int = this.age) = User(name, age)
```

然后可以这样写：

```
val jack = User(name = "Jack", age = 1)
val olderJack = jack.copy(age = 2)
```

数据类与解构声明

为数据类生成的 *component* 函数 使它们可在解构声明中使用：

```
val jane = User("Jane", 35)
val (name, age) = jane
println("$name, $age years of age")
// Jane, 35 years of age
```

标准数据类

标准库提供了 `Pair` 与 `Triple` 类。尽管在很多情况下具名数据类是更好的设计选择， 因为它们通过为属性提供有意义的名称使代码更具可读性。

密封类与密封接口

Sealed classes and interfaces provide controlled inheritance of your class hierarchies. All direct subclasses of a sealed class are known at compile time. No other subclasses may appear outside the module and package within which the sealed class is defined. The same logic applies to sealed interfaces and their implementations: once a module with a sealed interface is compiled, no new implementations can be created.

> Direct subclasses are classes that immediately inherit from their superclass.
>
> Indirect subclasses are classes that inherit from more than one level down from their superclass.
>
> 

When you combine sealed classes and interfaces with the `when` expression, you can cover the behavior of all possible subclasses and ensure that no new subclasses are created to affect your code adversely.

Sealed classes are best used for scenarios when:

- **Limited class inheritance is desired:** You have a predefined, finite set of subclasses that extend a class, all of which are known at compile time.
- **Type-safe design is required:** Safety and pattern matching are crucial in your project. Particularly for state management or handling complex conditional logic. For an example, check out Use sealed classes with when expressions.
- **Working with closed APIs:** You want robust and maintainable public APIs for libraries that ensure that third-party clients use the APIs as intended.

For more detailed practical applications, see Use case scenarios.

> Java 15 introduced a similar concept, where sealed classes use the `sealed` keyword along with the `permits` clause to define restricted hierarchies.
>
> 

Declare a sealed class or interface

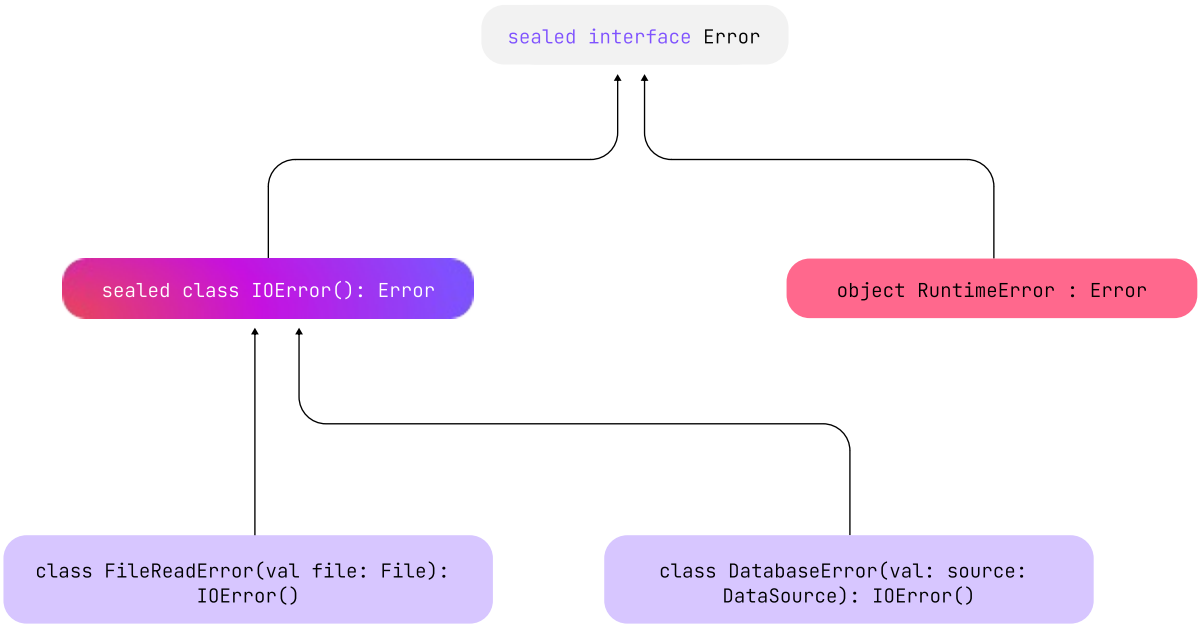
To declare a sealed class or interface, use the `sealed` modifier:

```
// Create a sealed interface
sealed interface Error

// Create a sealed class that implements sealed interface Error
```

```
sealed class IOError(): Error

// Define subclasses that extend sealed class 'IOError'
class FileReadError(val file: File): IOError()
class DatabaseError(val source: DataSource): IOError()

// Create a singleton object implementing the 'Error' sealed interface
object RuntimeError : Error
```

This example could represent a library's API that contains error classes to let library users handle errors that it can throw. If the hierarchy of such error classes includes interfaces or abstract classes visible in the public API, then nothing prevents other developers from implementing or extending them in the client code. Since the library doesn't know about errors declared outside of it, it can’t treat them consistently with its own classes. However, with a **sealed** hierarchy of error classes, library authors can be sure that they know all the possible error types and that other error types can't appear later.

The hierarchy of the example looks like this:



Constructors

A sealed class itself is always an abstract class, and as a result, can't be instantiated directly. However, it may contain or inherit constructors. These constructors aren't for creating instances of the sealed class itself but for its subclasses. Consider the following example with a sealed class called `Error`

and its several subclasses, which we instantiate:

```
sealed class Error(val message: String) {
    class NetworkError : Error("Network failure")
    class DatabaseError : Error("Database cannot be reached")
    class UnknownError : Error("An unknown error has occurred")
}

fun main() {
    val errors = listOf(Error.NetworkError(), Error.DatabaseError(), Error.UnknownErro
r())
    errors.forEach { println(it.message) }
}
// Network failure
// Database cannot be reached
// An unknown error has occurred
```

You can use `enum` classes within your sealed classes to use enum constants to represent states and provide additional detail. Each enum constant exists only as a **single** instance, while subclasses of a sealed class may have **multiple** instances. In the example, the `sealed class Error` along with its several subclasses, employs an `enum` to denote error severity. Each subclass constructor initializes the `severity` and can alter its state:

```
enum class ErrorSeverity { MINOR, MAJOR, CRITICAL }

sealed class Error(val severity: ErrorSeverity) {
    class FileReadError(val file: File): Error(ErrorSeverity.MAJOR)
    class DatabaseError(val source: DataSource): Error(ErrorSeverity.CRITICAL)
    object RuntimeError : Error(ErrorSeverity.CRITICAL)
    // Additional error types can be added here
}
```

Constructors of sealed classes can have one of two visibilities: `protected` (by default) or `private`:

```
sealed class IOError {
    // A sealed class constructor has protected visibility by default. It's visible in
side this class and its subclasses
    constructor() { /*...*/ }

    // Private constructor, visible inside this class only.
    // Using a private constructor in a sealed class allows for even stricter control
over instantiation, enabling specific initialization procedures within the class.
    private constructor(description: String): this() { /*...*/ }

    // This will raise an error because public and internal constructors are not allow
ed in sealed classes
    // public constructor(code: Int): this() {}
}
```

Inheritance

Direct subclasses of sealed classes and interfaces must be declared in the same package. They may be top-level or nested inside any number of other named classes, named interfaces, or named objects. Subclasses can have any visibility as long as they are compatible with normal inheritance rules in Kotlin.

Subclasses of sealed classes must have a properly qualified name. They can't be local or anonymous objects.

> `enum` classes can't extend a sealed class, or any other class. However, they can implement sealed interfaces:
>
> ```
> sealed interface Error
>
> // enum class extending the sealed interface Error
> enum class ErrorType : Error {
>     FILE_ERROR, DATABASE_ERROR
> }
> ```
>
> 

These restrictions don't apply to indirect subclasses. If a direct subclass of a sealed class is not marked as sealed, it can be extended in any way that its modifiers allow:

```
// Sealed interface 'Error' has implementations only in the same package and module
sealed interface Error

// Sealed class 'IOError' extends 'Error' and is extendable only within the same packa
ge
sealed class IOError(): Error

// Open class 'CustomError' extends 'Error' and can be extended anywhere it's visible
open class CustomError(): Error
```

Inheritance in multiplatform projects

There is one more inheritance restriction in multiplatform projects: direct subclasses of sealed classes must reside in the same source set. It applies to sealed classes without the expected and actual modifiers.

If a sealed class is declared as `expect` in a common source set and have `actual` implementations in platform source sets, both `expect` and `actual` versions can have subclasses in their source sets. Moreover, if you use a hierarchical structure, you can create subclasses in any source set between the `expect` and `actual` declarations.

Learn more about the hierarchical structure of multiplatform projects.

Use sealed classes with when expression

使用密封类的关键好处在于使用 `when` 表达式的时候。 The `when` expression, used with a sealed class, allows the Kotlin compiler to check exhaustively that all possible cases are covered. In such cases, you don't need to add an `else` clause:

```
// Sealed class and its subclasses
sealed class Error {
    class FileReadError(val file: String): Error()
    class DatabaseError(val source: String): Error()
    object RuntimeError : Error()
}

//sampleStart
// Function to log errors
fun log(e: Error) = when(e) {
    is Error.FileReadError -> println("Error while reading file ${e.file}")
    is Error.DatabaseError -> println("Error while reading from database ${e.source}")
    Error.RuntimeError -> println("Runtime error")
    // 不需要 `else` 子句，因为已经覆盖了所有的情况
}
//sampleEnd

// List all errors
fun main() {
    val errors = listOf(
        Error.FileReadError("example.txt"),
        Error.DatabaseError("usersDatabase"),
        Error.RuntimeError
    )

    errors.forEach { log(it) }
}
```

When using sealed classes with `when` expressions, you can also add guard conditions to include additional checks in a single branch. For more information, see Guard conditions in when expressions.

> In multiplatform projects, if you have a sealed class with a `when` expression as an expected declaration in your common code, you still need an `else` branch. This is because subclasses of `actual` platform implementations may extend sealed classes that aren't known in the common code.
>
> 

Use case scenarios

Let's explore some practical scenarios where sealed classes and interfaces can be particularly useful.

State management in UI applications

You can use sealed classes to represent different UI states in an application. This approach allows for structured and safe handling of UI changes. This example demonstrates how to manage various UI states:

```
sealed class UIState {
    data object Loading : UIState()
    data class Success(val data: String) : UIState()
    data class Error(val exception: Exception) : UIState()
}

fun updateUI(state: UIState) {
    when (state) {
        is UIState.Loading -> showLoadingIndicator()
        is UIState.Success -> showData(state.data)
        is UIState.Error -> showError(state.exception)
    }
}
```

Payment method handling

In practical business applications, handling various payment methods efficiently is a common requirement. You can use sealed classes with `when` expressions to implement such business logic. By representing different payment methods as subclasses of a sealed class, it establishes a clear and manageable structure for processing transactions:

```
sealed class Payment {
    data class CreditCard(val number: String, val expiryDate: String) : Payment()
    data class PayPal(val email: String) : Payment()
    data object Cash : Payment()
}
```

```
fun processPayment(payment: Payment) {
    when (payment) {
        is Payment.CreditCard -> processCreditCardPayment(payment.number, payment.expi
ryDate)
        is Payment.PayPal -> processPayPalPayment(payment.email)
        is Payment.Cash -> processCashPayment()
    }
}
```

`Payment` is a sealed class that represents different payment methods in an e-commerce system: `CreditCard`, `PayPal`, and `Cash`. Each subclass can have its specific properties, like `number` and `expiryDate` for `CreditCard`, and `email` for `PayPal`.

The `processPayment()` function demonstrates how to handle different payment methods. This approach ensures that all possible payment types are considered, and the system remains flexible for new payment methods to be added in the future.

API request-response handling

You can use sealed classes and sealed interfaces to implement a user authentication system that handles API requests and responses. The user authentication system has login and logout functionalities. The `ApiRequest` sealed interface defines specific request types: `LoginRequest` for login, and `LogoutRequest` for logout operations. The sealed class, `ApiResponse`, encapsulates different response scenarios: `UserSuccess` with user data, `UserNotFound` for absent users, and `Error` for any failures. The `handleRequest` function processes these requests in a type-safe manner using a `when` expression, while `getUserById` simulates user retrieval:

```
// Import necessary modules
import io.ktor.server.application.*
import io.ktor.server.resources.*

import kotlinx.serialization.*

// Define the sealed interface for API requests using Ktor resources
@Resource("api")
sealed interface ApiRequest

@Serializable
@Resource("login")
data class LoginRequest(val username: String, val password: String) : ApiRequest


@Serializable
@Resource("logout")
object LogoutRequest : ApiRequest

// Define the ApiResponse sealed class with detailed response types
```

```
sealed class ApiResponse {
    data class UserSuccess(val user: UserData) : ApiResponse()
    data object UserNotFound : ApiResponse()
    data class Error(val message: String) : ApiResponse()
}

// User data class to be used in the success response
data class UserData(val userId: String, val name: String, val email: String)

// Function to validate user credentials (for demonstration purposes)
fun isValidUser(username: String, password: String): Boolean {
    // Some validation logic (this is just a placeholder)
    return username == "validUser" && password == "validPass"
}

// Function to handle API requests with detailed responses
fun handleRequest(request: ApiRequest): ApiResponse {
    return when (request) {
        is LoginRequest -> {
            if (isValidUser(request.username, request.password)) {
                ApiResponse.UserSuccess(UserData("userId", "userName", "userEmail"))
            } else {
                ApiResponse.Error("Invalid username or password")
            }
        }
        is LogoutRequest -> {
            // Assuming logout operation always succeeds for this example
            ApiResponse.UserSuccess(UserData("userId", "userName", "userEmail")) // Fo
r demonstration
        }
    }
}

// Function to simulate a getUserById call
fun getUserById(userId: String): ApiResponse {
    return if (userId == "validUserId") {
        ApiResponse.UserSuccess(UserData("validUserId", "John Doe", "john@example.com"
))
    } else {
        ApiResponse.UserNotFound
    }
    // Error handling would also result in an Error response.
}

// Main function to demonstrate the usage
fun main() {
    val loginResponse = handleRequest(LoginRequest("user", "pass"))
    println(loginResponse)

    val logoutResponse = handleRequest(LogoutRequest)
    println(logoutResponse)

    val userResponse = getUserById("validUserId")
```

```
    println(userResponse)

    val userNotFoundResponse = getUserById("invalidId")
    println(userNotFoundResponse)
}
```

泛型：in、out、where

Kotlin 中的类可以有类型参数，与 Java 类似：

```
class Box<T>(t: T) {
    var value = t
}
```

创建这样类的实例只需提供类型参数即可：

```
val box: Box<Int> = Box<Int>(1)
```

但是如果类型参数可以推断出来，例如从构造函数的参数或者从其他途径， 就可以省略类型参数：

```
val box = Box(1) // 1 具有类型 Int，所以编译器推算出它是 Box<Int>
```

型变

Java 类型系统中最棘手的部分之一是通配符类型（参见 Java Generics FAQ）。 而 Kotlin 中没有。 相反，Kotlin 有声明处型变（declaration-site variance）与类型投影（type projections）。

Variance and wildcards in Java

我们来思考下为什么 Java 需要这些神秘的通配符。 首先，Java 中的泛型是不型变的， 这意味着 `List<String>` 并不是 `List<Object>` 的子类型。 如果 `List` 不是不型变的，它就没比 Java 的数组好到哪去，因为如下代码会通过编译但是导致运行时异常：

```
// Java
List<String> strs = new ArrayList<String>();

// Java reports a type mismatch here at compile-time.
List<Object> objs = strs;

// What if it didn't?
// We would be able to put an Integer into a list of Strings.
objs.add(1);

// And then at runtime, Java would throw
// a ClassCastException: Integer cannot be cast to String
```

```
String s = strs.get(0);
```

Java 禁止这样的事情以保证运行时的安全。但这样会有一些影响。例如，考虑 `Collection` 接口中的 `addAll()` 方法。该方法的签名应该是什么？直觉上，需要这样写：

```
// Java
interface Collection<E> …… {
    void addAll(Collection<E> items);
}
```

但随后，就无法做到以下这样（完全安全的）的事：

```
// Java

// The following would not compile with the naive declaration of addAll:
// Collection<String> is not a subtype of Collection<Object>
void copyAll(Collection<Object> to, Collection<String> from) {
    to.addAll(from);
}
```

这就是为什么 `addAll()` 的实际签名是以下这样：

```
// Java
interface Collection<E> …… {
    void addAll(Collection<? extends E> items);
}
```

通配符类型参数 `? extends E` 表示此方法接受 `E` 或者 *`E`* 的一个子类型对象的集合，而不只是 `E` 自身。这意味着我们可以安全地从其中（该集合中的元素是 E 的子类的实例）读取 `E`，但不能写入，因为我们不知道什么对象符合那个未知的 `E` 的子类型。反过来，该限制可以得到想要的行为：`Collection<String>` 表示为 `Collection<? extends Object>` 的子类型。简而言之，带 *extends* 限定（上界）的通配符类型使得类型是协变的（*covariant*）。

理解为什么这能够工作的关键相当简单：如果只能从集合中获取元素，那么使用 `String` 的集合，并且从其中读取 `Object` 也没问题。反过来，如果只能向集合中 放入 元素，就可以用 `Object` 集合并向其中放入 `String`：in Java there is `List<? super String>`, which accepts `String`s or any of its supertypes.

后者称为逆变性（*contravariance*），并且对于 `List <? super String>` 你只能调用接受 `String` 作为参数的方法（例如，你可以调用 `add(String)` 或者 `set(int, String)`），如果调用函数返回 `List<T>` 中的 `T`，你得到的并非一个 `String` 而是一个 `Object`。

Joshua Bloch 在其著作《Effective Java》第三版 中很好地解释了该问题（第 31 条：“利用有限制通配符来提升 API 的灵活性”）。 他称那些你只能从中读取的对象为生产者， 并称那些只能向其写入的对象为消费者。他建议：

> “为了灵活性最大化，在表示生产者或消费者的输入参数上使用通配符类型。”

他还提出了以下助记符：*PECS* 代表生产者-*Extends*、消费者-*Super*（*Producer-Extends, Consumer-Super*）。

> 如果你使用一个生产者对象，如 `List<? extends Foo>`，在该对象上不允许调用 `add()` 或 `set()`， 但这并不意味着它是不可变的：例如，没有什么阻止你调用 `clear()` 从列表中删除所有元素，因为 `clear()` 根本无需任何参数。
>
> 通配符（或其他类型的型变）保证的唯一的事情是类型安全。不可变性完全是另一回事。



声明处型变

假设有一个泛型接口 `Source<T>`，该接口中不存在任何以 `T` 作为参数的方法，只是方法返回 `T` 类型值：

```
// Java
interface Source<T> {
    T nextT();
}
```

那么，在 `Source <Object>` 类型的变量中存储 `Source <String>` 实例的引用是极为安全的 —— 没有消费者-方法可以调用。但是 Java 并不知道这一点，并且仍然禁止这样操作：

```
// Java
void demo(Source<String> strs) {
  Source<Object> objects = strs; // ！！！在 Java 中不允许
  // ……
}
```

为了修正这一点，我们必须声明对象的类型为 `Source<? extends Object>`。这么做毫无意义，因为我们可以像以前一样在该对象上调用所有相同的方法，所以更复杂的类型并没有带来价值。 但编译器并不知道。

在 Kotlin 中，有一种方法向编译器解释这种情况。这称为声明处型变： 可以标注 `Source` 的类型参数 `T` 来确保它仅从 `Source<T>` 成员中返回（生产），并从不被消费。 为此请使用 `out` 修饰符：

```
interface Source<out T> {
    fun nextT(): T
}

fun demo(strs: Source<String>) {
    val objects: Source<Any> = strs // 这个没问题，因为 T 是一个 out-参数
    // ……
}
```

一般原则是：当一个类 `C` 的类型参数 `T` 被声明为 `out` 时，它就只能出现在 `C` 的成员的输出-位置， 但回报是 `C<Base>` 可以安全地作为 `C<Derived>` 的超类。

简而言之，可以说类 `C` 是在参数 `T` 上是协变的，或者说 `T` 是一个协变的类型参数。 可以认为 `C` 是 `T` 的生产者，而不是 `T` 的消费者。

`out` 修饰符称为型变注解，并且由于它在类型参数声明处提供， 所以它提供了声明处型变。这与 Java 的使用处型变相反，其类型用途通配符使得类型协变。

另外除了 `out` ，Kotlin 又补充了一个型变注解： `in` 。它使得一个类型参数逆变，即只可以消费而不可以生产。逆变类型的一个很好的例子是 `Comparable` ：

```
interface Comparable<in T> {
    operator fun compareTo(other: T): Int
}

fun demo(x: Comparable<Number>) {
    x.compareTo(1.0) // 1.0 拥有类型 Double，它是 Number 的子类型
    // 因此，可以将 x 赋给类型为 Comparable <Double> 的变量
    val y: Comparable<Double> = x // OK！
}
```

in 和 *out* 两词看起来是自解释的（因为它们已经在 C# 中成功使用很长时间了）， 因此上面提到的助记符不是真正需要的。可以将其改写为更高级的抽象：

存在性（**The Existential**）变换：消费者 **in**, 生产者 **out!** :-)

类型投影

使用处型变：类型投影

将类型参数 `T` 声明为 `out` 非常简单，并且能避免使用处子类型化的麻烦， 但是有些类实际上不能限制为只返回 `T` ！一个很好的例子是 `Array` ：

```
class Array<T>(val size: Int) {
    operator fun get(index: Int): T { …… }
```

```
    operator fun set(index: Int, value: T) { …… }
}
```

该类在 `T` 上既不能是协变的也不能是逆变的。这造成了一些不灵活性。考虑下述函数：

```
fun copy(from: Array<Any>, to: Array<Any>) {
    assert(from.size == to.size)
    for (i in from.indices)
        to[i] = from[i]
}
```

这个函数应该将项目从一个数组复制到另一个数组。让我们尝试在实践中应用它：

```
val ints: Array<Int> = arrayOf(1, 2, 3)
val any = Array<Any>(3) { "" }
copy(ints, any)
//   ^ 其类型为 Array<Int> 但此处期望 Array<Any>
```

这里我们遇到同样熟悉的问题：`Array <T>` 在 `T` 上是不型变的，因此 `Array <Int>` 与 `Array <Any>` 都不是另一个的子类型。为什么？再次重复，因为 `copy` 可能有非预期行为，例如它可能尝试写一个 `String` 到 `from`， 并且如果我们实际上传递一个 `Int` 的数组，以后会抛 `ClassCastException` 异常。

To prohibit the `copy` function from *writing* to `from`, you can do the following:

```
fun copy(from: Array<out Any>, to: Array<Any>) { …… }
```

这就是类型投影：意味着 `from` 不仅仅是一个数组，而是一个受限制的（投影的）数组。 只可以调用返回类型为类型参数 `T` 的方法，如上，这意味着只能调用 `get()`。 这就是使用处型变的用法，并且是对应于 Java 的 `Array<? extends Object>`、 但更简单。

你也可以使用 `in` 投影一个类型：

```
fun fill(dest: Array<in String>, value: String) { …… }
```

`Array<in String>` 对应于 Java 的 `Array<? super String>`，也就是说，你可以传递一个 `String` 数组、 `CharSequence` 数组或 `Object` 数组给 `fill()` 函数。

星投影

有时你想说，你对类型参数一无所知，但仍然希望以安全的方式使用它。 这里的安全方式是定义泛型类型的这种投影，该泛型类型的每个具体实例化都会是该投影的子类型。

Kotlin 为此提供了所谓的星投影语法：

- 对于 `Foo <out T : TUpper>`，其中 `T` 是一个具有上界 `TUpper` 的协变类型参数，`Foo <*>` 等价于 `Foo <out TUpper>`。 意味着当 `T` 未知时，你可以安全地从 `Foo <*>` 读取 `TUpper` 的值。
- 对于 `Foo <in T>`，其中 `T` 是一个逆变类型参数，`Foo <*>` 等价于 `Foo <in Nothing>`。 意味着当 `T` 未知时， 没有什么可以以安全的方式写入 `Foo <*>`。
- 对于 `Foo <T : TUpper>`，其中 `T` 是一个具有上界 `TUpper` 的不型变类型参数，`Foo<*>` 对于读取值时等价于 `Foo<out TUpper>` 而对于写值时等价于 `Foo<in Nothing>`。

如果泛型类型具有多个类型参数，则每个类型参数都可以单独投影。 例如，如果类型被声明为 `interface Function <in T, out U>`，可以使用以下星投影：

- `Function<*, String>` 表示 `Function<in Nothing, String>`。
- `Function<Int, *>` 表示 `Function<Int, out Any?>`。
- `Function<*, *>` 表示 `Function<in Nothing, out Any?>`。

> 星投影非常像 Java 的原始类型，但是安全。



泛型函数

不仅类可以有类型参数。函数也可以有。类型参数要放在函数名称之前：

```
fun <T> singletonList(item: T): List<T> {
    // ……
}

fun <T> T.basicToString(): String { // 扩展函数
    // ……
}
```

要调用泛型函数，在调用处函数名之后指定类型参数即可：

```
val l = singletonList<Int>(1)
```

可以省略能够从上下文中推断出来的类型参数，所以以下示例同样适用：

```
val l = singletonList(1)
```

泛型约束

能够替换给定类型参数的所有可能类型的集合可以由泛型约束限制。

上界

最常见的约束类型是上界，与 Java 的 `extends` 关键字对应：

```
fun <T : Comparable<T>> sort(list: List<T>) {  …… }
```

冒号之后指定的类型是上界，表明只有 `Comparable<T>` 的子类型可以替代 `T` 。 例如：

```
sort(listOf(1, 2, 3)) // OK。Int 是 Comparable<Int> 的子类型
sort(listOf(HashMap<Int, String>())) // 错误：HashMap<Int, String> 不是 Comparable<HashM
ap<Int, String>> 的子类型
```

默认的上界（如果没有声明）是 `Any?` 。在尖括号中只能指定一个上界。 如果同一类型参数需要多个上界，需要一个单独的 *where*-子句：

```
fun <T> copyWhenGreater(list: List<T>, threshold: T): List<String>
    where T : CharSequence,
          T : Comparable<T> {
    return list.filter { it > threshold }.map { it.toString() }
}
```

所传递的类型必须同时满足 `where` 子句的所有条件。在上述示例中，类型 `T` 必须 既实现了 `CharSequence` 也实现了 `Comparable` 。

Definitely non-nullable types

To make interoperability with generic Java classes and interfaces easier, Kotlin supports declaring a generic type parameter as **definitely non-nullable**.

To declare a generic type `T` as definitely non-nullable, declare the type with `& Any` . For example: `T & Any` .

A definitely non-nullable type must have a nullable upper bound.

The most common use case for declaring definitely non-nullable types is when you want to override a Java method that contains `@NotNull` as an argument. For example, consider the `load()` method:

```
import org.jetbrains.annotations.*;

public interface Game<T> {
    public T save(T x) {}
```

```
    @NotNull
    public T load(@NotNull T x) {}
}
```

To override the `load()` method in Kotlin successfully, you need `T1` to be declared as definitely non-nullable:

```
interface ArcadeGame<T1> : Game<T1> {
    override fun save(x: T1): T1
    // T1 is definitely non-nullable
    override fun load(x: T1 & Any): T1 & Any
}
```

When working only with Kotlin, it's unlikely that you will need to declare definitely non-nullable types explicitly because Kotlin's type inference takes care of this for you.

类型擦除

Kotlin 为泛型声明用法执行的类型安全检测在编译期进行。 运行时泛型类型的实例不保留关于其类型实参的任何信息。 其类型信息称为被擦除。例如， `Foo<Bar>` 与 `Foo<Baz?>` 的实例都会被擦除为 `Foo<*>` 。

泛型类型检测与类型转换

由于类型擦除，并没有通用的方法在运行时检测一个泛型类型的实例是否通过指定类型参数所创建，并且编译器禁止这种 `is` 检测，例如 `ints is List<Int>` or `list is T` (type parameter). 当然，你可以对一个实例检测星投影的类型：

```
if (something is List<*>) {
    something.forEach { println(it) } // 每一项的类型都是 `Any?`
}
```

类似地，当已经让一个实例的类型参数（在编译期）静态检测， 就可以对涉及非泛型部分做 `is` 检测或者类型转换。请注意， 在这种情况下，会省略尖括号：

```
fun handleStrings(list: MutableList<String>) {
    if (list is ArrayList) {
        // `list` 智能转换为 `ArrayList<String>`
    }
}
```

省略类型参数的这种语法可用于不考虑类型参数的类型转换： `list as ArrayList` 。

泛型函数调用的类型参数也同样只在编译期检测。在函数体内部， 类型参数不能用于类型检测，并且类型转换为类型参数（ `foo as T` ）也是非受检的。 The only exclusion is inline functions with reified type parameters, which have their actual type arguments inlined at each call site. This enables type checks and casts for the type parameters. However, the restrictions described above still apply for instances of generic types used inside checks or casts. For example, in the type check `arg is T`, if `arg` is an instance of a generic type itself, its type arguments are still erased.

```
//sampleStart
inline fun <reified A, reified B> Pair<*, *>.asPairOf(): Pair<A, B>? {
    if (first !is A || second !is B) return null
    return first as A to second as B
}

val somePair: Pair<Any?, Any?> = "items" to listOf(1, 2, 3)


val stringToSomething = somePair.asPairOf<String, Any>()
val stringToInt = somePair.asPairOf<String, Int>()
val stringToList = somePair.asPairOf<String, List<*>>()
val stringToStringList = somePair.asPairOf<String, List<String>>() // Compiles but bre
aks type safety!
// Expand the sample for more details

//sampleEnd

fun main() {
    println("stringToSomething = " + stringToSomething)
    println("stringToInt = " + stringToInt)
    println("stringToList = " + stringToList)
    println("stringToStringList = " + stringToStringList)
    //println(stringToStringList?.second?.forEach() {it.length}) // This will throw Cl
assCastException as list items are not String
}
```

非受检类型转换

类型转换为带有具体类型参数的泛型类型，如 `foo as List<String>` 无法在运行时检测。 当高级程序逻辑隐含了类型转换的类型安全而无法直接通过编译器推断时， 可以使用这种非受检类型转换。 See the example below.

```
fun readDictionary(file: File): Map<String, *> = file.inputStream().use {
    TODO("Read a mapping of strings to arbitrary elements.")
}

// 我们已将存有一些 `Int` 的映射保存到这个文件
val intsFile = File("ints.dictionary")

// Warning: Unchecked cast: `Map<String, *>` to `Map<String, Int>`
```

```
val intsDictionary: Map<String, Int> = readDictionary(intsFile) as Map<String, Int>
```

最后一行的类型转换会出现一个警告。编译器无法在运行时完全检测该类型转换，并且不能保证映射中的值是“Int”。

为避免未受检类型转换，可以重新设计程序结构。在上例中，可以使用具有类型安全实现的不同接口 `DictionaryReader<T>` 与 `DictionaryWriter<T>`。 可以引入合理的抽象，将未受检的类型转换从调用处移动到实现细节中。 正确使用泛型型变也有帮助。

对于泛型函数，使用具体化的类型参数可以使形如 `arg as T` 这样的类型转换受检，除非 `arg` 对应类型的自身类型参数已被擦除。

可以通过在产生警告的语句或声明上用注解 `@Suppress("UNCHECKED_CAST")` 标注来禁止未受检类型转换警告：

```
inline fun <reified T> List<*>.asListOfType(): List<T>? =
    if (all { it is T })
        @Suppress("UNCHECKED_CAST")
        this as List<T> else
        null
```

> 对于 JVM 平台：数组类型（`Array<Foo>`）会保留关于其元素被擦除类型的信息，并且类型转换为一个数组类型可以部分受检：元素类型的可空性与类型实参仍然会被擦除。例如，如果 `foo` 是一个保存了任何 `List<*>`（无论可不可空）的数组的话，类型转换 `foo as Array<List<String>?>` 都会成功。

Underscore operator for type arguments

The underscore operator `_` can be used for type arguments. Use it to automatically infer a type of the argument when other types are explicitly specified:

```
abstract class SomeClass<T> {
    abstract fun execute() : T
}

class SomeImplementation : SomeClass<String>() {
    override fun execute(): String = "Test"
}

class OtherImplementation : SomeClass<Int>() {
    override fun execute(): Int = 42
}
```

```
object Runner {
    inline fun <reified S: SomeClass<T>, T> run() : T {
        return S::class.java.getDeclaredConstructor().newInstance().execute()
    }
}

fun main() {
    // T is inferred as String because SomeImplementation derives from SomeClass<Strin
g>
    val s = Runner.run<SomeImplementation, _>()
    assert(s == "Test")

    // T is inferred as Int because OtherImplementation derives from SomeClass<Int>
    val n = Runner.run<OtherImplementation, _>()
    assert(n == 42)
}
```

嵌套类与内部类

类可以嵌套在其他类中：

```
class Outer {
    private val bar: Int = 1
    class Nested {
        fun foo() = 2
    }
}

val demo = Outer.Nested().foo() // == 2
```

还可以使用带有嵌套的接口。所有类与接口的组合都是可能的：可以将接口嵌套在类中、将类嵌套在接口中、将接口嵌套在接口中。

```
interface OuterInterface {
    class InnerClass
    interface InnerInterface
}

class OuterClass {
    class InnerClass
    interface InnerInterface
}
```

内部类

标记为 `inner` 的嵌套类能够访问其外部类的成员。内部类会带有一个对外部类的对象的引用：

```
class Outer {
    private val bar: Int = 1
    inner class Inner {
        fun foo() = bar
    }
}

val demo = Outer().Inner().foo() // == 1
```

参见限定的 `this` 表达式以了解内部类中的 `this` 的消歧义用法。

匿名内部类

使用对象表达式创建匿名内部类实例：

```
window.addMouseListener(object : MouseAdapter() {

    override fun mouseClicked(e: MouseEvent) { …… }

    override fun mouseEntered(e: MouseEvent) { …… }
})
```

> 对于 JVM 平台, 如果对象是函数式 Java 接口（即具有单个抽象方法的 Java 接口）的实例，可以使用带接口类型前缀的 lambda 表达式创建它：
>
> ```
> val listener = ActionListener { println("clicked") }
> ```
>
> 

枚举类

枚举类的最基本的应用场景是实现类型安全的枚举：

```
enum class Direction {
    NORTH, SOUTH, WEST, EAST
}
```

每个枚举常量都是一个对象。枚举常量以逗号分隔。

因为每一个枚举都是枚举类的实例，所以可以这样初始化：

```
enum class Color(val rgb: Int) {
    RED(0xFF0000),
    GREEN(0x00FF00),
    BLUE(0x0000FF)
}
```

匿名类

枚举常量可以声明其带有相应方法以及覆盖了基类方法的自身匿名类 。

```
enum class ProtocolState {
    WAITING {
        override fun signal() = TALKING
    },

    TALKING {
        override fun signal() = WAITING
    };

    abstract fun signal(): ProtocolState
}
```

如果枚举类定义任何成员，那么使用分号将成员定义与常量定义分隔开。

在枚举类中实现接口

一个枚举类可以实现接口（但不能从类继承），可以为所有条目提供统一的接口成员实现，也可以在相应匿名类中为每个条目提供各自的实现。 只需将想要实现的接口添加到枚举类声明中即可，如下所示：

```
import java.util.function.BinaryOperator
import java.util.function.IntBinaryOperator

//sampleStart
enum class IntArithmetics : BinaryOperator<Int>, IntBinaryOperator {
    PLUS {
        override fun apply(t: Int, u: Int): Int = t + u
    },
    TIMES {
        override fun apply(t: Int, u: Int): Int = t * u
    };

    override fun applyAsInt(t: Int, u: Int) = apply(t, u)
}
//sampleEnd

fun main() {
    val a = 13
    val b = 31
    for (f in IntArithmetics.entries) {
        println("$f($a, $b) = ${f.apply(a, b)}")
    }
}
```

All enum classes implement the Comparable interface by default. Constants in the enum class are defined in the natural order. For more information, see Ordering.

使用枚举常量

Kotlin 中的枚举类有合成的属性与方法用于列出定义的枚举常量以及通过其名称获取枚举常量。这些方法的签名如下（假设枚举类的名称是 `EnumClass`）：

```
EnumClass.valueOf(value: String): EnumClass
EnumClass.entries: EnumEntries<EnumClass> // specialized List<EnumClass>
```

Below is an example of them in action:

```
enum class RGB { RED, GREEN, BLUE }

fun main() {
    for (color in RGB.entries) println(color.toString()) // prints RED, GREEN, BLUE
    println("The first color is: ${RGB.valueOf("RED")}") // prints "The first color is
```

```
    : RED"
}
```

如果指定的名称与类中定义的任何枚举常量均不匹配， `valueOf()` 方法会抛出 `IllegalArgumentException` 异常。

Prior to the introduction of `entries` in Kotlin 1.9.0, the `values()` function was used to retrieve an array of enum constants.

每个枚举常量也都具有这两个属性： `name` 与 `ordinal` ， 用于在枚举类声明中获取其名称与（自 0 起的）位置：

```
enum class RGB { RED, GREEN, BLUE }

fun main() {
    //sampleStart
    println(RGB.RED.name)    // prints RED
    println(RGB.RED.ordinal) // prints 0
    //sampleEnd
}
```

可以使用 `enumValues<T>()` 与 `enumValueOf<T>()` 函数以泛型的方式访问枚举类中的常量。 In Kotlin 2.0.0, the `enumEntries<T>()` function is introduced as a replacement for the `enumValues<T>()` function. The `enumEntries<T>()` function returns a list of all enum entries for the given enum type `T` .

The `enumValues<T>()` function is still supported, but we recommend that you use the `enumEntries<T>()` function instead because it has less performance impact. Every time you call `enumValues<T>()` a new array is created, whereas whenever you call `enumEntries<T>()` the same list is returned each time, which is far more efficient.

For example:

```
enum class RGB { RED, GREEN, BLUE }

inline fun <reified T : Enum<T>> printAllValues() {
    println(enumEntries<T>().joinToString { it.name })
}

printAllValues<RGB>()
// RED, GREEN, BLUE
```

> For more information about inline functions and reified type parameters, see Inline functions.
>
> 

Inline value classes

Sometimes it is useful to wrap a value in a class to create a more domain-specific type. 然而，由于额外的堆内存分配问题， 它会引入运行时的性能开销。此外，如果被包装的类型是原生类型，the performance hit is significant, 因为原生类型通常在运行时就进行了大量优化，然而他们的包装器却没有得到任何特殊的处理。

为了解决这类问题，Kotlin 引入了一种被称为内联类的特殊类。 Inline classes are a subset of value-based classes. They don't have an identity and can only hold values.

To declare an inline class, use the `value` modifier before the name of the class:

```
value class Password(private val s: String)
```

To declare an inline class for the JVM backend, use the `value` modifier along with the `@JvmInline` annotation before the class declaration:

```
// For JVM backends
@JvmInline
value class Password(private val s: String)
```

内联类必须含有唯一的一个属性在主构造函数中初始化。在运行时， 将使用这个唯一属性来表示内联类的实例（关于运行时的内部表达请参阅下文）：

```
// 不存在 'Password' 类的真实实例对象
// 在运行时，'securePassword' 仅仅包含 'String'
val securePassword = Password("Don't try this in production")
```

这就是内联类的主要特性，它灵感来源于 *inline* 这个名称：类的数据被内联到该类使用的地方（类似于内联函数中的代码被内联到该函数调用的地方）。

成员

内联类支持普通类中的一些功能。特别是，内联类可以声明属性与函数, have an `init` block and secondary constructors:

```
@JvmInline
value class Person(private val fullName: String) {
    init {
        require(fullName.isNotEmpty()) {
```

```
            "Full name shouldn't be empty"
        }
    }

    constructor(firstName: String, lastName: String) : this("$firstName $lastName") {
        require(lastName.isNotBlank()) {
            "Last name shouldn't be empty"
        }
    }

    val length: Int
        get() = fullName.length

    fun greet() {
        println("Hello, $fullName")
    }
}

fun main() {
    val name1 = Person("Kotlin", "Mascot")
    val name2 = Person("Kodee")
    name1.greet() // greet` 函数会作为一个静态方法被调用
    println(name2.length) // 属性的 getter 会作为一个静态方法被调用
}
```

Inline class properties cannot have backing fields. They can only have simple computable properties (no `lateinit` /delegated properties).

继承

内联类允许去继承接口

```
interface Printable {
    fun prettyPrint(): String
}

@JvmInline
value class Name(val s: String) : Printable {
    override fun prettyPrint(): String = "Let's $s!"
}

fun main() {
    val name = Name("Kotlin")
    println(name.prettyPrint()) // 仍然会作为一个静态方法被调用
}
```

禁止内联类参与到类的继承关系结构中。这就意味着内联类不能继承其他的类而且始终是 `final` 的。

表示方式

在生成的代码中，Kotlin 编译器为每个内联类保留一个包装器。内联类的实例可以在运行时表示为包装器或者基础类型。这就类似于 `Int` 可以表示为原生类型 `int` 或者包装器 `Integer`。

为了生成性能最优的代码，Kotlin 编译更倾向于使用基础类型而不是包装器。 然而，有时候使用包装器是必要的。一般来说，只要将内联类用作另一种类型， 它们就会被装箱。

```
interface I

@JvmInline
value class Foo(val i: Int) : I

fun asInline(f: Foo) {}
fun <T> asGeneric(x: T) {}
fun asInterface(i: I) {}
fun asNullable(i: Foo?) {}

fun <T> id(x: T): T = x

fun main() {
    val f = Foo(42)

    asInline(f)    // 拆箱操作: 用作 Foo 本身
    asGeneric(f)   // 装箱操作: 用作泛型类型 T
    asInterface(f) // 装箱操作: 用作类型 I
    asNullable(f)  // 装箱操作: 用作不同于 Foo 的可空类型 Foo?

    // 在下面这里例子中, 'f' 首先会被装箱（当它作为参数传递给 'id' 函数时）然后又被拆箱（当它从'id'
函数中被返回时）
    // 最后, 'c' 中就包含了被拆箱后的内部表达(也就是 '42'), 和 'f' 一样
    val c = id(f)
}
```

因为内联类既可以表示为基础类型有可以表示为包装器，引用相等对于内联类而言毫无意义，因此这也是被禁止的。

Inline classes can also have a generic type parameter as the underlying type. In this case, the compiler maps it to `Any?` or, generally, to the upper bound of the type parameter.

```
@JvmInline
value class UserId<T>(val value: T)

fun compute(s: UserId<String>) {} // compiler generates fun compute-<hashcode>(s: Any?)
```

名字修饰

由于内联类被编译为其基础类型，因此可能会导致各种模糊的错误，例如意想不到的平台签名冲突：

```
@JvmInline
value class UInt(val x: Int)

// 在 JVM 平台上被表示为'public final void compute(int x)'
fun compute(x: Int) { }

// 同理，在 JVM 平台上也被表示为'public final void compute(int x)'!
fun compute(x: UInt) { }
```

为了缓解这种问题，一般会通过在函数名后面拼接一些稳定的哈希码来修饰函数名。 因此，`fun compute(x: UInt)` 将会被表示为 `public final void compute-<hashcode>(int x)`，以此来解决冲突的问题。

Calling from Java code

You can call functions that accept inline classes from Java code. To do so, you should manually disable mangling: add the `@JvmName` annotation before the function declaration:

```
@JvmInline
value class UInt(val x: Int)

fun compute(x: Int) { }

@JvmName("computeUInt")
fun compute(x: UInt) { }
```

内联类与类型别名

初看起来，内联类似乎与类型别名非常相似。实际上，两者似乎都引入了一种新的类型，并且都在运行时表示为基础类型。

然而，关键的区别在于类型别名与其基础类型（以及具有相同基础类型的其他类型别名）是赋值兼容的，而内联类却不是这样。

换句话说，内联类引入了一个真实的新类型，与类型别名正好相反，类型别名仅仅是为现有的类型取了个新的替代名称（别名）：

```
typealias NameTypeAlias = String
```

```
@JvmInline
value class NameInlineClass(val s: String)

fun acceptString(s: String) {}
fun acceptNameTypeAlias(n: NameTypeAlias) {}
fun acceptNameInlineClass(p: NameInlineClass) {}

fun main() {
    val nameAlias: NameTypeAlias = ""
    val nameInlineClass: NameInlineClass = NameInlineClass("")
    val string: String = ""

    acceptString(nameAlias) // 正确：传递别名类型的实参替代函数中基础类型的形参
    acceptString(nameInlineClass) // 错误：不能传递内联类的实参替代函数中基础类型的形参

    // And vice versa:
    acceptNameTypeAlias(string) // 正确：传递基础类型的实参替代函数中别名类型的形参
    acceptNameInlineClass(string) // 错误：不能传递基础类型的实参替代函数中内联类类型的形参
}
```

Inline classes and delegation

Implementation by delegation to inlined value of inlined class is allowed with interfaces:

```
interface MyInterface {
    fun bar()
    fun foo() = "foo"
}

@JvmInline
value class MyInterfaceWrapper(val myInterface: MyInterface) : MyInterface by myInterf
ace

fun main() {
    val my = MyInterfaceWrapper(object : MyInterface {
        override fun bar() {
            // body
        }
    })
    println(my.foo()) // prints "foo"
}
```

对象声明与对象表达式

In Kotlin, objects allow you to define a class and create an instance of it in a single step. This is useful when you need either a reusable singleton instance or a one-time object. To handle these scenarios, Kotlin provides two key approaches: *object declarations* for creating singletons and *object expressions* for creating anonymous, one-time objects.

> A singleton ensures that a class has only one instance and provides a global point of access to it.
>
> 

Object declarations and object expressions are best used for scenarios when:

- **Using singletons for shared resources:** You need to ensure that only one instance of a class exists throughout the application. For example, managing a database connection pool.
- **Creating factory methods:** You need a convenient way to create instances efficiently. Companion objects allow you to define class-level functions and properties tied to a class, simplifying the creation and management of these instances.
- **Modifying existing class behavior temporarily:** You want to modify the behavior of an existing class without the need to create a new subclass. For example, adding temporary functionality to an object for a specific operation.
- **Type-safe design is required:** You require one-time implementations of interfaces or abstract classes using object expressions. This can be useful for scenarios like a button click handler.

对象声明

You can create single instances of objects in Kotlin using object declarations, which always have a name following the `object` keyword. This allows you to define a class and create an instance of it in a single step, which is useful for implementing singletons:

```
//sampleStart
// Declares a Singleton object to manage data providers
object DataProviderManager {
    private val providers = mutableListOf<DataProvider>()

    // Registers a new data provider
    fun registerDataProvider(provider: DataProvider) {
        providers.add(provider)
    }

    // Retrieves all registered data providers
    val allDataProviders: Collection<DataProvider>
```

```
        get() = providers
}
//sampleEnd

// Example data provider interface
interface DataProvider {
    fun provideData(): String
}

// Example data provider implementation
class ExampleDataProvider : DataProvider {
    override fun provideData(): String {
        return "Example data"
    }
}

fun main() {
    // Creates an instance of ExampleDataProvider
    val exampleProvider = ExampleDataProvider()

    // To refer to the object, use its name directly
    DataProviderManager.registerDataProvider(exampleProvider)

    // Retrieves and prints all data providers
    println(DataProviderManager.allDataProviders.map { it.provideData() })
    // [Example data]
}
```

> 对象声明的初始化过程是线程安全的并且在首次访问时进行。



如需引用该对象（`object`），直接使用其名称即可：

```
DataProviderManager.registerDataProvider(exampleProvider)
```

Object declarations can also have supertypes, similar to how anonymous objects can inherit from existing classes or implement interfaces:

```
object DefaultListener : MouseAdapter() {
    override fun mouseClicked(e: MouseEvent) { …… }

    override fun mouseEntered(e: MouseEvent) { …… }
}
```

Like variable declarations, object declarations are not expressions, so they cannot be used on the right-hand side of an assignment statement:

```
// Syntax error: An object expression cannot bind a name.
val myObject = object MySingleton {
    val name = "Singleton"
}
```

对象声明不能在局部作用域，这意味着它们不能直接嵌套在函数内部。 不过，它们可以嵌套在其他对象声明或非内部类中。

Data objects

When printing a plain object declaration in Kotlin, the string representation contains both its name and the hash of the `object`:

```
object MyObject

fun main() {
    println(MyObject)
    // MyObject@hashcode
}
```

However, by marking an object declaration with the `data` modifier, you can instruct the compiler to return the actual name of the object when calling `toString()`, the same way it works for data classes:

```
data object MyDataObject {
    val number: Int = 3
}

fun main() {
    println(MyDataObject)
    // MyDataObject
}
```

Additionally, the compiler generates several functions for your `data object`:

- `toString()` returns the name of the data object
- `equals()` / `hashCode()` enables equality checks and hash-based collections

> You can't provide a custom `equals` or `hashCode` implementation for a `data object`.
>
> 

The `equals()` function for a `data object` ensures that all objects that have the type of your `data object` are considered equal. In most cases, you will only have a single instance of your `data object` at runtime, since a `data object` declares a singleton. However, in the edge case where

another object of the same type is generated at runtime (for example, by using platform reflection with `java.lang.reflect` or a JVM serialization library that uses this API under the hood), this ensures that the objects are treated as being equal.

> Make sure that you only compare `data objects` structurally (using the `==` operator) and never by reference (using the `===` operator). This helps you to avoid pitfalls when more than one instance of a data object exists at runtime.
>
> 

```
import java.lang.reflect.Constructor

data object MySingleton

fun main() {
    val evilTwin = createInstanceViaReflection()

    println(MySingleton)
    // MySingleton

    println(evilTwin)
    // MySingleton

    // Even when a library forcefully creates a second instance of MySingleton,
    // its equals() function returns true:
    println(MySingleton == evilTwin)
    // true

    // Don't compare data objects using ===
    println(MySingleton === evilTwin)
    // false
}

fun createInstanceViaReflection(): MySingleton {
    // Kotlin reflection does not permit the instantiation of data objects.
    // This creates a new MySingleton instance "by force" (using Java platform reflect
ion)
    // Don't do this yourself!
    return (MySingleton.javaClass.declaredConstructors[0].apply { isAccessible = true
} as Constructor<MySingleton>).newInstance()
}
```

The generated `hashCode()` function has a behavior that is consistent with the `equals()` function, so that all runtime instances of a `data object` have the same hash code.

Differences between data objects and data classes

While `data object` and `data class` declarations are often used together and have some similarities, there are some functions that are not generated for a `data object`:

- No `copy()` function. Because a `data object` declaration is intended to be used as singletons, no `copy()` function is generated. Singletons restrict the instantiation of a class to a single instance, which would be violated by allowing copies of the instance to be created.
- No `componentN()` function. Unlike a `data class`, a `data object` does not have any data properties. Since attempting to destructure such an object without data properties wouldn't make sense, no `componentN()` functions are generated.

Use data objects with sealed hierarchies

Data object declarations are particularly useful for sealed hierarchies like sealed classes or sealed interfaces. They allow you to maintain symmetry with any data classes you may have defined alongside the object.

In this example, declaring `EndOfFile` as a `data object` instead of a plain `object` means that it will get the `toString()` function without the need to override it manually:

```
sealed interface ReadResult
data class Number(val number: Int) : ReadResult
data class Text(val text: String) : ReadResult
data object EndOfFile : ReadResult

fun main() {
    println(Number(7))
    // Number(number=7)
    println(EndOfFile)
    // EndOfFile
}
```

伴生对象

Companion objects allow you to define class-level functions and properties. This makes it easy to create factory methods, hold constants, and access shared utilities.

类内部的对象声明可以用 `companion` 关键字标记：

```
class MyClass {
    companion object Factory {
        fun create(): MyClass = MyClass()
    }
}
```

该伴生对象（`companion object`）的成员可通过只使用类名作为限定符来调用：

```
class User(val name: String) {
    // Defines a companion object that acts as a factory for creating User instances
    companion object Factory {
        fun create(name: String): User = User(name)
    }
}

fun main(){
    // Calls the companion object's factory method using the class name as the qualifi
er.
    // Creates a new User instance
    val userInstance = User.create("John Doe")
    println(userInstance.name)
    // John Doe
}
```

可以省略伴生对象的名称，在这种情况下使用名称 `Companion`：

```
class User(val name: String) {
    // Defines a companion object without a name
    companion object { }
}

// Accesses the companion object
val companionUser = User.Companion
```

Class members can access `private` members of their corresponding `companion object`:

```
class User(val name: String) {
    companion object {
        private val defaultGreeting = "Hello"
    }

    fun sayHi() {
        println(defaultGreeting)
    }
}
User("Nick").sayHi()
// Hello
```

When a class name is used by itself, it acts as a reference to the companion object of the class, regardless of whether the companion object is named or not:

```
//sampleStart
class User1 {
    // Defines a named companion object
    companion object Named {
        fun show(): String = "User1's Named Companion Object"
```

```
    }
}

// References the companion object of User1 using the class name
val reference1 = User1

class User2 {
    // Defines an unnamed companion object
    companion object {
        fun show(): String = "User2's Companion Object"
    }
}

// References the companion object of User2 using the class name
val reference2 = User2
//sampleEnd

fun main() {
    // Calls the show() function from the companion object of User1
    println(reference1.show())
    // User1's Named Companion Object

    // Calls the show() function from the companion object of User2
    println(reference2.show())
    // User2's Companion Object
}
```

虽然 Kotlin 中伴生对象的成员看起来像其他语言的静态成员，但是它们实际上是伴生对象的实例成员，这意味着它们属于对象本身。这让伴生对象可以实现接口：

```
interface Factory<T> {
    fun create(name: String): T
}

class User(val name: String) {
    // Defines a companion object that implements the Factory interface
    companion object : Factory<User> {
        override fun create(name: String): User = User(name)
    }
}

fun main() {
    // Uses the companion object as a Factory
    val userFactory: Factory<User> = User
    val newUser = userFactory.create("Example User")
    println(newUser.name)
    // Example User
}
```

当然，在 JVM 平台，如果使用 `@JvmStatic` 注解，你可以将伴生对象的成员生成为真正的静态方法和字段。更详细信息请参见Java 互操作性一节 。

对象表达式

Object expressions declare a class and create an instance of that class, but without naming either of them. These classes are useful for one-time use. They can either be created from scratch, inherit from existing classes, or implement interfaces. Instances of these classes are also called *anonymous objects* because they are defined by an expression, not a name.

Create anonymous objects from scratch

Object expressions start with the `object` keyword.

If the object doesn't extend any classes or implement interfaces, you can define an object's members directly inside curly braces after the `object` keyword:

```
fun main() {
//sampleStart
    val helloWorld = object {
        val hello = "Hello"
        val world = "World"
        // Object expressions extend the Any class, which already has a toString() fun
ction,
        // so it must be overridden
        override fun toString() = "$hello $world"
    }

    print(helloWorld)
    // Hello World
//sampleEnd
}
```

Inherit anonymous objects from supertypes

如需创建一个继承自某个（或某些）类型的匿名对象， specify this type after `object` and a colon `:` . Then implement or override the members of this class as if you were inheriting from it:

```
window.addMouseListener(object : MouseAdapter() {
    override fun mouseClicked(e: MouseEvent) { /*...*/ }

    override fun mouseEntered(e: MouseEvent) { /*...*/ }
})
```

如果超类型有一个构造函数，那么传递适当的构造函数参数给它。 可以指定多个超类型，在冒号后用逗号分隔：

```
//sampleStart
// Creates an open class BankAccount with a balance property
open class BankAccount(initialBalance: Int) {
    open val balance: Int = initialBalance
}

// Defines an interface Transaction with an execute() function
interface Transaction {
    fun execute()
}

// A function to perform a special transaction on a BankAccount
fun specialTransaction(account: BankAccount) {
    // Creates an anonymous object that inherits from the BankAccount class and implem
ents the Transaction interface
    // The balance of the provided account is passed to the BankAccount superclass con
structor
    val temporaryAccount = object : BankAccount(account.balance), Transaction {

        override val balance = account.balance + 500  // Temporary bonus

        // Implements the execute() function from the Transaction interface
        override fun execute() {
            println("Executing special transaction. New balance is $balance.")
        }
    }
    // Executes the transaction
    temporaryAccount.execute()
}
//sampleEnd
fun main() {
    // Creates a BankAccount with an initial balance of 1000
    val myAccount = BankAccount(1000)
    // Performs a special transaction on the created account
    specialTransaction(myAccount)
    // Executing special transaction. New balance is 1500.
}
```

Use anonymous objects as return and value types

When you return an anonymous object from a local or `private` function or property, all the members of that anonymous object are accessible through that function or property:

```
//sampleStart
class UserPreferences {
    private fun getPreferences() = object {
```

```
        val theme: String = "Dark"
        val fontSize: Int = 14
    }

    fun printPreferences() {
        val preferences = getPreferences()
        println("Theme: ${preferences.theme}, Font Size: ${preferences.fontSize}")
    }
}
//sampleEnd

fun main() {
    val userPreferences = UserPreferences()
    userPreferences.printPreferences()
    // Theme: Dark, Font Size: 14
}
```

This allows you to return an anonymous object with specific properties, offering a simple way to encapsulate data or behavior without creating a separate class.

If a function or property that returns an anonymous object has `public`, `protected`, or `internal` visibility, its actual type is:

- `Any` if the anonymous object doesn't have a declared supertype.
- The declared supertype of the anonymous object, if there is exactly one such type.
- The explicitly declared type if there is more than one declared supertype.

In all these cases, members added in the anonymous object are not accessible. Overridden members are accessible if they are declared in the actual type of the function or property. For example:

```
//sampleStart
interface Notification {
    // Declares notifyUser() in the Notification interface
    fun notifyUser()
}

interface DetailedNotification

class NotificationManager {
    // The return type is Any. The message property is not accessible.
    // When the return type is Any, only members of the Any class are accessible.
    fun getNotification() = object {
        val message: String = "General notification"
    }

    // The return type is Notification because the anonymous object implements only on
e interface
    // The notifyUser() function is accessible because it is part of the Notification
interface
    // The message property is not accessible because it is not declared in the Notifi
```

```
cation interface
    fun getEmailNotification() = object : Notification {
        override fun notifyUser() {
            println("Sending email notification")
        }
        val message: String = "You've got mail!"
    }

    // The return type is DetailedNotification. The notifyUser() function and the mess
age property are not accessible
    // Only members declared in the DetailedNotification interface are accessible
    fun getDetailedNotification(): DetailedNotification = object : Notification, Detai
ledNotification {
        override fun notifyUser() {
            println("Sending detailed notification")
        }
        val message: String = "Detailed message content"
    }
}
//sampleEnd
fun main() {
    // This produces no output
    val notificationManager = NotificationManager()

    // The message property is not accessible here because the return type is Any
    // This produces no output
    val notification = notificationManager.getNotification()

    // The notifyUser() function is accessible
    // The message property is not accessible here because the return type is Notifica
tion
    val emailNotification = notificationManager.getEmailNotification()
    emailNotification.notifyUser()
    // Sending email notification

    // The notifyUser() function and message property are not accessible here because
the return type is DetailedNotification
    // This produces no output
    val detailedNotification = notificationManager.getDetailedNotification()
}
```

Access variables from anonymous objects

对象表达式主体内的代码可以访问来自包含它的作用域的变量：

```
import java.awt.event.MouseAdapter
import java.awt.event.MouseEvent

fun countClicks(window: JComponent) {
    var clickCount = 0
    var enterCount = 0
```

```
    // MouseAdapter provides default implementations for mouse event functions
    // Simulates MouseAdapter handling mouse events
    window.addMouseListener(object : MouseAdapter() {
        override fun mouseClicked(e: MouseEvent) {
            clickCount++
        }

        override fun mouseEntered(e: MouseEvent) {
            enterCount++
        }
    })
    // The clickCount and enterCount variables are accessible within the object expres
sion
}
```

对象声明与表达式之间的行为差异

对象声明与对象表达式在初始化行为方面存在差异：

- 对象表达式是在使用他们的地方立即执行（及初始化）的。
- 对象声明是在第一次被访问到时延迟初始化的。
- 伴生对象的初始化是在相应的类被加载（解析）时，与 Java 静态初始化器的语义相匹配。

委托

委托模式已经证明是实现继承的一个很好的替代方式， 而 Kotlin 可以零样板代码地原生支持它。

`Derived` 类可以通过将其所有公有成员都委托给指定对象来实现一个接口 `Base`：

```
interface Base {
    fun print()
}

class BaseImpl(val x: Int) : Base {
    override fun print() { print(x) }
}

class Derived(b: Base) : Base by b

fun main() {
    val base = BaseImpl(10)
    Derived(base).print()
}
```

`Derived` 的超类型列表中的 `by`-子句表示 `b` 将会在 `Derived` 中内部存储， 并且编译器将生成转发给 `b` 的所有 `Base` 的方法。

覆盖由委托实现的接口成员

覆盖符合预期：编译器会使用 `override` 覆盖的实现而不是委托对象中的。如果将 `override fun printMessage() { print("abc") }` 添加到 `Derived`，那么当调用 `printMessage` 时程序会输出 *abc* 而不是 *10*：

```
interface Base {
    fun printMessage()
    fun printMessageLine()
}

class BaseImpl(val x: Int) : Base {
    override fun printMessage() { print(x) }
    override fun printMessageLine() { println(x) }
}

class Derived(b: Base) : Base by b {
    override fun printMessage() { print("abc") }
}
```

```
fun main() {
    val base = BaseImpl(10)
    Derived(base).printMessage()
    Derived(base).printMessageLine()
}
```

但请注意，以这种方式重写的成员不会在委托对象的成员中调用，委托对象的成员只能访问其自身对接口成员实现：

```
interface Base {
    val message: String
    fun print()
}

class BaseImpl(x: Int) : Base {
    override val message = "BaseImpl: x = $x"
    override fun print() { println(message) }
}

class Derived(b: Base) : Base by b {
    // 在 b 的 `print` 实现中不会访问到这个属性
    override val message = "Message of Derived"
}

fun main() {
    val b = BaseImpl(10)
    val derived = Derived(b)
    derived.print()
    println(derived.message)
}
```

Learn more about delegated properties.

属性委托

With some common kinds of properties, even though you can implement them manually every time you need them, it is more helpful to implement them once, add them to a library, and reuse them later. For example:

- 延迟属性（*lazy* properties）：其值只在首次访问时计算。
- 可观察属性（*observable* properties）：监听器会收到有关此属性变更的通知。
- 把多个属性储存在一个映射（*map*）中，而不是每个存在单独的字段中。

为了涵盖这些（以及其他）情况，Kotlin 支持 委托属性:

```
class Example {
    var p: String by Delegate()
}
```

语法是： `val/var <属性名>: <类型> by <表达式>` 。在 `by` 后面的表达式是该 委托， 因为属性对应的 `get()` （与 `set()` ）会被委托给它的 `getValue()` 与 `setValue()` 方法。 属性的委托不必实现接口，但是需要提供一个 `getValue()` 函数（对于 `var` 属性还有 `setValue()` ）。

例如:

```
import kotlin.reflect.KProperty

class Delegate {
    operator fun getValue(thisRef: Any?, property: KProperty<*>): String {
        return "$thisRef, thank you for delegating '${property.name}' to me!"
    }

    operator fun setValue(thisRef: Any?, property: KProperty<*>, value: String) {
        println("$value has been assigned to '${property.name}' in $thisRef.")
    }
}
```

当从委托到一个 `Delegate` 实例的 `p` 读取时，将调用 `Delegate` 中的 `getValue()` 函数。它的第一个参数是读出 `p` 的对象、第二个参数保存了对 `p` 自身的描述（例如可以取它的名称)。

```
val e = Example()
println(e.p)
```

输出结果：

```
Example@33a17727, thank you for delegating 'p' to me!
```

类似地，当我们给 `p` 赋值时，将调用 `setValue()` 函数。前两个参数相同， 第三个参数保存将要被赋予的值：

```
e.p = "NEW"
```

输出结果：

```
NEW has been assigned to 'p' in Example@33a17727.
```

委托对象的要求规范可以在下文找到。

可以在函数或代码块中声明一个委托属性；它不一定是类的成员。 你可以在下文找到其示例。

标准委托

Kotlin 标准库为几种有用的委托提供了工厂方法。

延迟属性 Lazy properties

`lazy()` 是接受一个 lambda 并返回一个 `Lazy <T>` 实例的函数，返回的实例可以作为实现延迟属性的委托。 第一次调用 `get()` 会执行已传递给 `lazy()` 的 lambda 表达式并记录结果。后续调用 `get()` 只是返回记录的结果。

```
val lazyValue: String by lazy {
    println("computed!")
    "Hello"
}

fun main() {
    println(lazyValue)
    println(lazyValue)
}
```

默认情况下，对于 lazy 属性的求值是同步锁的（*synchronized*）：该值只在一个线程中计算，但所有线程都会看到相同的值。如果初始化委托的同步锁不是必需的，这样可以让多个线程同时执行，那么将 `LazyThreadSafetyMode.PUBLICATION` 作为参数传给 `lazy()`。

如果你确定初始化将总是发生在与属性使用位于相同的线程， 那么可以使用 `LazyThreadSafetyMode.NONE` 模式。它不会有任何线程安全的保证以及相关的开销。

可观察属性 Observable properties

`Delegates.observable()` 接受两个参数：初始值与修改时处理程序（handler）。

每当我们给属性赋值时会调用该处理程序（在赋值后执行）。它有三个参数：被赋值的属性、旧值与新值：

```
import kotlin.properties.Delegates

class User {
    var name: String by Delegates.observable("<no name>") {
        prop, old, new ->
        println("$old -> $new")
    }
}

fun main() {
    val user = User()
    user.name = "first"
    user.name = "second"
}
```

如果你想截获赋值并否决它们，那么使用 `vetoable()` 取代 `observable()`。 在属性被赋新值之前会调用传递给 `vetoable` 的处理程序。

委托给另一个属性

一个属性可以把它的 getter 与 setter 委托给另一个属性。这种委托对于顶层和类的属性（成员和扩展）都可用。该委托属性可以为：

- 顶层属性
- 同一个类的成员或扩展属性
- 另一个类的成员或扩展属性

为将一个属性委托给另一个属性，应在委托名称中使用 `::` 限定符，例如， `this::delegate` 或 `MyClass::delegate`。

```
var topLevelInt: Int = 0
class ClassWithDelegate(val anotherClassInt: Int)

class MyClass(var memberInt: Int, val anotherClassInstance: ClassWithDelegate) {
    var delegatedToMember: Int by this::memberInt
    var delegatedToTopLevel: Int by ::topLevelInt

    val delegatedToAnotherClass: Int by anotherClassInstance::anotherClassInt
}
```

```
var MyClass.extDelegated: Int by ::topLevelInt
```

这是很有用的，例如，当想要以一种向后兼容的方式重命名一个属性的时候：引入一个新的属性、 使用 `@Deprecated` 注解来注解旧的属性、并委托其实现。

```
class MyClass {
   var newName: Int = 0
   @Deprecated("Use 'newName' instead", ReplaceWith("newName"))
   var oldName: Int by this::newName
}
fun main() {
   val myClass = MyClass()
   // 通知: 'oldName: Int' is deprecated.
   // Use 'newName' instead
   myClass.oldName = 42
   println(myClass.newName) // 42
}
```

将属性储存在映射中

一个常见的用例是在一个映射（map）里存储属性的值。 这经常出现在像解析 JSON 或者执行其他“动态”任务的应用中。 在这种情况下，你可以使用映射实例自身作为委托来实现委托属性。

```
class User(val map: Map<String, Any?>) {
    val name: String by map
    val age: Int     by map
}
```

在这个例子中，构造函数接受一个映射参数：

```
val user = User(mapOf(
    "name" to "John Doe",
    "age"  to 25
))
```

Delegated properties take values from this map through string keys, which are associated with the names of properties:

```
class User(val map: Map<String, Any?>) {
    val name: String by map
    val age: Int     by map
}
```

```
fun main() {
    val user = User(mapOf(
        "name" to "John Doe",
        "age"  to 25
    ))
//sampleStart
    println(user.name) // Prints "John Doe"
    println(user.age)  // Prints 25
//sampleEnd
}
```

这也适用于 `var` 属性，如果把只读的 `Map` 换成 `MutableMap` 的话：

```
class MutableUser(val map: MutableMap<String, Any?>) {
    var name: String by map
    var age: Int     by map
}
```

局部委托属性

你可以将局部变量声明为委托属性。 例如，你可以使一个局部变量惰性初始化：

```
fun example(computeFoo: () -> Foo) {
    val memoizedFoo by lazy(computeFoo)

    if (someCondition && memoizedFoo.isValid()) {
        memoizedFoo.doSomething()
    }
}
```

`memoizedFoo` 变量只会在第一次访问时计算。 如果 `someCondition` 失败，那么该变量根本不会计算。

属性委托要求

对于一个只读属性（即 `val` 声明的），委托必须提供一个操作符函数 `getValue()`，该函数具有以下参数：

- `thisRef` 必须与属性所有者类型（对于扩展属性必须是被扩展的类型）相同或者是其超类型。
- `property` 必须是类型 `KProperty<*>` 或其超类型。

`getValue()` 必须返回与属性相同的类型（或其子类型）。

```
class Resource

class Owner {
    val valResource: Resource by ResourceDelegate()
}

class ResourceDelegate {
    operator fun getValue(thisRef: Owner, property: KProperty<*>): Resource {
        return Resource()
    }
}
```

对于一个可变属性（即 `var` 声明的），委托必须额外提供一个操作符函数 `setValue()`，该函数具有以下参数：

- `thisRef` 必须与属性所有者类型（对于扩展属性必须是被扩展的类型）相同或者是其超类型。
- `property` 必须是类型 `KProperty<*>` 或其超类型。
- `value` 必须与属性类型相同（或者是其超类型）。

```
class Resource

class Owner {
    var varResource: Resource by ResourceDelegate()
}

class ResourceDelegate(private var resource: Resource = Resource()) {
    operator fun getValue(thisRef: Owner, property: KProperty<*>): Resource {
        return resource
    }
    operator fun setValue(thisRef: Owner, property: KProperty<*>, value: Any?) {
        if (value is Resource) {
            resource = value
        }
    }
}
```

`getValue()` 或/与 `setValue()` 函数可以通过委托类的成员函数提供或者由扩展函数提供。当你需要委托属性到原本未提供的这些函数的对象时后者会更便利。 两函数都需要用 `operator` 关键字来进行标记。

You can create delegates as anonymous objects without creating new classes, by using the interfaces `ReadOnlyProperty` and `ReadWriteProperty` from the Kotlin standard library. They provide the required methods: `getValue()` is declared in `ReadOnlyProperty`; `ReadWriteProperty` extends it and adds `setValue()`. This means you can pass a `ReadWriteProperty` whenever a `ReadOnlyProperty` is expected.

```
fun resourceDelegate(resource: Resource = Resource()): ReadWriteProperty<Any?, Resourc
e> =
    object : ReadWriteProperty<Any?, Resource> {
        var curValue = resource
        override fun getValue(thisRef: Any?, property: KProperty<*>): Resource = curVa
lue
        override fun setValue(thisRef: Any?, property: KProperty<*>, value: Resource)
{
            curValue = value
        }
    }

val readOnlyResource: Resource by resourceDelegate()  // ReadWriteProperty as val
var readWriteResource: Resource by resourceDelegate()
```

Translation rules for delegated properties

在底层，Kotlin 编译器会为某些类型的委托属性生成辅助属性并委托给它们。

> For optimization purposes, the compiler *does not* generate auxiliary properties in several cases. Learn about the optimization on the example of delegating to another property.
>
> 

例如，对于属性 `prop`，生成隐藏属性 `prop$delegate`，而访问器的代码只是简单地委托给这个附加属性：

```
class C {
    var prop: Type by MyDelegate()
}

// 这段是由编译器生成的相应代码：
class C {
    private val prop$delegate = MyDelegate()
    var prop: Type
        get() = prop$delegate.getValue(this, this::prop)
        set(value: Type) = prop$delegate.setValue(this, this::prop, value)
}
```

Kotlin 编译器在参数中提供了关于 `prop` 的所有必要信息：第一个参数 `this` 引用到外部类 `C` 的实例，而 `this::prop` 是 `KProperty` 类型的反射对象，该对象描述 `prop` 自身。

Optimized cases for delegated properties

The `$delegate` field will be omitted if a delegate is:

- A referenced property:

```
class C<Type> {
    private var impl: Type = ...
    var prop: Type by ::impl
}
```

- A named object:

```
object NamedObject {
    operator fun getValue(thisRef: Any?, property: KProperty<*>): String = ...
}

val s: String by NamedObject
```

- A final `val` property with a backing field and a default getter in the same module:

```
val impl: ReadOnlyProperty<Any?, String> = ...

class A {
    val s: String by impl
}
```

- A constant expression, enum entry, `this` , `null` . The example of `this` :

```
class A {
    operator fun getValue(thisRef: Any?, property: KProperty<*>) ...

    val s by this
}
```

Translation rules when delegating to another property

When delegating to another property, the Kotlin compiler generates immediate access to the referenced property. This means that the compiler doesn't generate the field `prop$delegate` . This optimization helps save memory.

Take the following code, for example:

```
class C<Type> {
    private var impl: Type = ...
    var prop: Type by ::impl
}
```

Property accessors of the `prop` variable invoke the `impl` variable directly, skipping the delegated property's `getValue` and `setValue` operators, and thus the `KProperty` reference object is not needed.

For the code above, the compiler generates the following code:

```
class C<Type> {
    private var impl: Type = ...

    var prop: Type
        get() = impl
        set(value) {
            impl = value
        }

    fun getProp$delegate(): Type = impl // This method is needed only for reflection
}
```

提供委托

通过定义 `provideDelegate` 操作符，可以扩展创建属性实现所委托对象的逻辑。 如果 `by` 右侧所使用的对象将 `provideDelegate` 定义为成员或扩展函数， 那么会调用该函数来创建属性委托实例。

One of the possible use cases of `provideDelegate` is to check the consistency of the property upon its initialization.

例如，如需在绑定之前检测属性名称，可以这样写：

```
class ResourceDelegate<T> : ReadOnlyProperty<MyUI, T> {
    override fun getValue(thisRef: MyUI, property: KProperty<*>): T { ... }
}

class ResourceLoader<T>(id: ResourceID<T>) {
    operator fun provideDelegate(
            thisRef: MyUI,
            prop: KProperty<*>
    ): ReadOnlyProperty<MyUI, T> {
        checkProperty(thisRef, prop.name)
        // 创建委托
        return ResourceDelegate()
    }

    private fun checkProperty(thisRef: MyUI, name: String) { …… }
}

class MyUI {
```

```
    fun <T> bindResource(id: ResourceID<T>): ResourceLoader<T> { …… }

    val image by bindResource(ResourceID.image_id)
    val text by bindResource(ResourceID.text_id)
}
```

`provideDelegate` 的参数与 `getValue` 的相同：

- `thisRef` 必须与 属性所有者 类型（对于扩展属性必须是被扩展的类型）相同或者是它的超类型；
- `property` 必须是类型 `KProperty<*>` 或其超类型。

在创建 `MyUI` 实例期间，为每个属性调用 `provideDelegate` 方法，并立即执行必要的验证。

如果没有这种拦截属性与其委托之间的绑定的能力，为了实现相同的功能， 你必须显式传递属性名，这不是很方便：

```
// 检测属性名称而不使用“provideDelegate”功能
class MyUI {
    val image by bindResource(ResourceID.image_id, "image")
    val text by bindResource(ResourceID.text_id, "text")
}

fun <T> MyUI.bindResource(
        id: ResourceID<T>,
        propertyName: String
): ReadOnlyProperty<MyUI, T> {
    checkProperty(this, propertyName)
   // 创建委托
}
```

在生成的代码中，会调用 `provideDelegate` 方法来初始化辅助的 `prop$delegate` 属性。 比较对于属性声明 `val prop: Type by MyDelegate()` 生成的代码与上面（当 `provideDelegate` 方法不存在时）生成的代码：

```
class C {
    var prop: Type by MyDelegate()
}

// 这段代码是当“provideDelegate”功能可用时
// 由编译器生成的代码:
class C {
    // 调用“provideDelegate”来创建额外的“delegate”属性
    private val prop$delegate = MyDelegate().provideDelegate(this, this::prop)
    var prop: Type
        get() = prop$delegate.getValue(this, this::prop)
        set(value: Type) = prop$delegate.setValue(this, this::prop, value)
}
```

请注意，`provideDelegate` 方法只影响辅助属性的创建，并不会影响为 getter 或 setter 生成的代码。

With the `PropertyDelegateProvider` interface from the standard library, you can create delegate providers without creating new classes.

```
val provider = PropertyDelegateProvider { thisRef: Any?, property ->
    ReadOnlyProperty<Any?, Int> {_, property -> 42 }
}
val delegate: Int by provider
```

类型别名

类型别名为现有类型提供替代名称。 如果类型名称太长，你可以另外引入较短的名称，并使用新的名称替代原类型名。

它有助于缩短较长的泛型类型。 例如，通常缩减集合类型是很有吸引力的：

```
typealias NodeSet = Set<Network.Node>

typealias FileTable<K> = MutableMap<K, MutableList<File>>
```

你可以为函数类型提供另外的别名：

```
typealias MyHandler = (Int, String, Any) -> Unit

typealias Predicate<T> = (T) -> Boolean
```

你可以为内部类和嵌套类创建新名称：

```
class A {
    inner class Inner
}
class B {
    inner class Inner
}

typealias AInner = A.Inner
typealias BInner = B.Inner
```

类型别名不会引入新类型。 它们等效于相应的底层类型。 当你在代码中添加 `typealias Predicate<T>` 并使用 `Predicate<Int>` 时，Kotlin 编译器总是把它扩展为 `(Int) -> Boolean`。 因此，当你需要泛型函数类型时，你可以传递该类型的变量，反之亦然：

```
typealias Predicate<T> = (T) -> Boolean

fun foo(p: Predicate<Int>) = p(42)

fun main() {
    val f: (Int) -> Boolean = { it > 0 }
    println(foo(f)) // 输出 "true"

    val p: Predicate<Int> = { it > 0 }
    println(listOf(1, -2).filter(p)) // 输出 "[1]"
}
```

函数

- 函数
- lambda 表达式
- 内联函数
- 操作符重载
- 构建器
  - 类型安全的构建器
  - 使用带有构建器类型推断的构建器

函数

Kotlin 函数使用 `fun` 关键字声明：

```
fun double(x: Int): Int {
    return 2 * x
}
```

函数用法

使用常规方式来调用函数：

```
val result = double(2)
```

调用成员函数使用点表示法：

```
Stream().read() // 创建类 Stream 实例并调用 read()
```

参数

函数参数使用 Pascal 表示法定义——*name*: *type*。参数用逗号隔开， 每个参数必须有显式类型：

```
fun powerOf(number: Int, exponent: Int): Int { /*……*/ }
```

在声明函数参数的时候，你也可以使用尾部逗号：

```
fun powerOf(
    number: Int,
    exponent: Int, // 尾部逗号
) { /*……*/ }
```

默认实参

函数参数可以有默认值，当省略相应的参数时使用默认值。这可以减少重载数量：

```
fun read(
    b: ByteArray,
```

```
    off: Int = 0,
    len: Int = b.size,
) { /*……*/ }
```

通过在类型的后面添加 `=` 符号来设置参数的默认值。

覆盖方法总是使用与基类型方法相同的默认实参值。 当覆盖一个有默认实参值的方法时，必须从签名中省略默认实参值：

```
open class A {
    open fun foo(i: Int = 10) { /*……*/ }
}

class B : A() {
    override fun foo(i: Int) { /*……*/ }  // 不能有默认值
}
```

如果一个有默认值参数在一个无默认值的参数之前，那么该默认值只能通过使用具名实参调用该函数来使用：

```
fun foo(
    bar: Int = 0,
    baz: Int,
) { /*……*/ }

foo(baz = 1) // 使用默认值 bar = 0
```

如果在默认实参之后的最后一个参数是 lambda 表达式，那么它既可以作为具名实参在括号内传入，也可以在括号外传入：

```
fun foo(
    bar: Int = 0,
    baz: Int = 1,
    qux: () -> Unit,
) { /*……*/ }

foo(1) { println("hello") }     // 使用默认值 baz = 1
foo(qux = { println("hello") }) // 使用两个默认值 bar = 0 与 baz = 1
foo { println("hello") }        // 使用两个默认值 bar = 0 与 baz = 1
```

具名实参

在调用函数的时候可以指明一个或多个参数，这对于包含多个参数的函数是十分有帮助的，尤其是在传入 `null` 或者布尔值的情况下很难联想到参数的含义。

在调用函数的过程中使用具名实参时，你可以随意修改参数的顺序。如果想要使用某些参数的默认值，只需要在传参的时候省略掉这些参数即可。

以 `reformat()` 函数为例，该函数包含 4 个有默认值的参数。

```
fun reformat(
    str: String,
    normalizeCase: Boolean = true,
    upperCaseFirstLetter: Boolean = true,
    divideByCamelHumps: Boolean = false,
    wordSeparator: Char = ' ',
) { /*……*/ }
```

当调用这个函数时，不需要让其所有参数都具名：

```
reformat(
    "String!",
    false,
    upperCaseFirstLetter = false,
    divideByCamelHumps = true,
    '_'
)
```

你也可以省略所有有默认值的参数：

```
reformat("This is a long String!")
```

除了省略掉所有有默认值的参数，你也可以选择只省略某些特定的有默认值的参数。 但是你需要在跳过第一个参数后，对后续的所有参数都使用具名实参：

```
reformat("This is a short String!", upperCaseFirstLetter = false, wordSeparator = '_')
```

可以通过具名实参与展开操作符（数组前加 `*`）来传入可变实参（`vararg`）：

```
fun foo(vararg strings: String) { /*……*/ }

foo(strings = *arrayOf("a", "b", "c"))
```

> 在 JVM 平台中调用 Java 函数时不能使用具名实参语法，因为 Java 字节码并不总是保留函数参数的名称。



返回 Unit 的函数

如果一个函数并不返回有用的值，其返回类型是 `Unit`。 `Unit` 是一种只有一个值—— `Unit` 的类型。 这个值不需要显式返回：

```
fun printHello(name: String?): Unit {
    if (name != null)
        println("Hello $name")
    else
        println("Hi there!")
    // `return Unit` 或者 `return` 是可选的
}
```

`Unit` 返回类型声明也是可选的。上面的代码等同于：

```
fun printHello(name: String?) { …… }
```

单表达式函数

当函数体由单个表达式构成时，可以省略花括号并且在 `=` 符号之后指定代码体即可：

```
fun double(x: Int): Int = x * 2
```

当返回值类型可由编译器推断时，显式声明返回类型是可选的：

```
fun double(x: Int) = x * 2
```

显式返回类型

具有块代码体的函数必须始终显式指定返回类型，除非他们旨在返回 `Unit`， 在这种情况下指定返回值类型是可选的。

Kotlin 不推断具有块代码体的函数的返回类型，因为这样的函数在代码体中可能有复杂的控制流， 并且返回类型对于读者（有时甚至对于编译器）是不明显的。

可变数量的参数（varargs）

函数的参数（通常是最后一个）可以用 `vararg` 修饰符标记：

```
fun <T> asList(vararg ts: T): List<T> {
    val result = ArrayList<T>()
    for (t in ts) // ts 是一个数组
        result.add(t)
    return result
}
```

在本例中，可以将可变数量的参数传递给函数：

```
val list = asList(1, 2, 3)
```

在函数内部，类型 `T` 的 `vararg` 参数的可见方式是作为 `T` 数组，如上例中的 `ts` 变量具有类型 `Array <out T>` 。

只有一个参数可以标注为 `vararg` 。如果 `vararg` 参数不是列表中的最后一个参数， 必须使用具名实参语法传递其后的参数的值，或者，如果参数具有函数类型，则通过在括号外部传一个 lambda。

当调用 `vararg` -函数时，可以逐个传参，例如 `asList(1, 2, 3)` 。如果已经有一个数组并希望将其内容传给该函数，那么使用展开操作符（在数组前面加 `*` ）：

```
val a = arrayOf(1, 2, 3)
val list = asList(-1, 0, *a, 4)
```

如果你想传入一个原生类型数组到 `vararg` 中，你需要先通过 `toTypedArray()` 函数将其转换为常规的类型化数组：

```
val a = intArrayOf(1, 2, 3) // IntArray 是一种原生类型数组
val list = asList(-1, 0, *a.toTypedArray(), 4)
```

中缀表示法

标有 `infix` 关键字的函数也可以使用中缀表示法（忽略该调用的点与圆括号）调用。 中缀函数必须满足以下要求：

- 它们必须是成员函数或扩展函数。
- 它们必须只有一个参数。
- 其参数不得接受可变数量的参数且不能有默认值。

```
infix fun Int.shl(x: Int): Int { …… }

// 用中缀表示法调用该函数
1 shl 2

// 等同于这样
1.shl(2)
```

> 中缀函数调用的优先级低于算术操作符、类型转换以及 `rangeTo` 操作符。 以下表达式是等价的：

> - `1 shl 2 + 3` 等价于 `1 shl (2 + 3)`
> - `0 until n * 2` 等价于 `0 until (n * 2)`
> - `xs union ys as Set<*>` 等价于 `xs union (ys as Set<*>)`
>
> 另一方面，中缀函数调用的优先级高于布尔操作符 `&&` 与 `||`、`is-` 与 `in-` 检测以及其他一些操作符。这些表达式也是等价的：
>
> - `a && b xor c` 等价于 `a && (b xor c)`
> - `a xor b in c` 等价于 `(a xor b) in c`
>
> 

请注意，中缀函数总是要求指定接收者与参数。当使用中缀表示法在当前接收者上调用方法时，需要显式使用 `this`。这是确保非模糊解析所必需的。

```
class MyStringCollection {
    infix fun add(s: String) { /*……*/ }

    fun build() {
        this add "abc"   // 正确
        add("abc")       // 正确
        //add "abc"        // 错误：必须指定接收者
    }
}
```

函数作用域

Kotlin 函数可以在文件顶层声明，这意味着你不需要像一些语言如 Java、C# 与 Scala (在 Scala 3 之后新增了顶层作用域的支持) 那样需要创建一个类来保存一个函数。此外除了顶层函数，Kotlin 中函数也可以声明在局部作用域、作为成员函数以及扩展函数。

局部函数

Kotlin 支持局部函数，即一个函数在另一个函数内部：

```
fun dfs(graph: Graph) {
    fun dfs(current: Vertex, visited: MutableSet<Vertex>) {
        if (!visited.add(current)) return
        for (v in current.neighbors)
            dfs(v, visited)
    }

    dfs(graph.vertices[0], HashSet())
}
```

局部函数可以访问外部函数（闭包）的局部变量。在上例中，`visited` 可以是局部变量：

```
fun dfs(graph: Graph) {
    val visited = HashSet<Vertex>()
    fun dfs(current: Vertex) {
        if (!visited.add(current)) return
        for (v in current.neighbors)
            dfs(v)
    }

    dfs(graph.vertices[0])
}
```

成员函数

成员函数是在类或对象内部定义的函数：

```
class Sample {
    fun foo() { print("Foo") }
}
```

成员函数以点表示法调用：

```
Sample().foo() // 创建类 Sample 实例并调用 foo
```

关于类和覆盖成员的更多信息参见类和继承。

泛型函数

函数可以有泛型参数，通过在函数名前使用尖括号指定：

```
fun <T> singletonList(item: T): List<T> { /*……*/ }
```

关于泛型函数的更多信息，请参见泛型。

尾递归函数

Kotlin 支持一种称为尾递归的函数式编程风格。 对于某些使用循环的算法，可以使用尾递归替代而不会有堆栈溢出的风险。 当一个函数用 `tailrec` 修饰符标记并满足所需的形式条件时，编译器会优化该递归， 留下一个快速而高效的基于循环的版本：

```
val eps = 1E-10 // "good enough", could be 10^-15

tailrec fun findFixPoint(x: Double = 1.0): Double =
    if (Math.abs(x - Math.cos(x)) < eps) x else findFixPoint(Math.cos(x))
```

这段代码计算余弦的不动点（`fixpoint` of cosine），这是一个数学常数。 它只是重复地从 `1.0` 开始调用 `Math.cos`， 直到结果不再改变，对于这里指定的 `eps` 精度会产生 `0.7390851332151611` 的结果。最终代码相当于这种更传统风格的代码：

```
val eps = 1E-10 // "good enough", could be 10^-15

private fun findFixPoint(): Double {
    var x = 1.0
    while (true) {
        val y = Math.cos(x)
        if (Math.abs(x - y) < eps) return x
        x = Math.cos(x)
    }
}
```

要符合 `tailrec` 修饰符的条件的话，函数必须将其自身调用作为它执行的最后一个操作。在递归调用后有更多代码时， 不能使用尾递归，不能用在 `try` / `catch` / `finally` 块中，也不能用于 open 的函数。 目前在 Kotlin for the JVM 与 Kotlin/Native 中支持尾递归。

另见：

- 内联函数
- 扩展函数
- 高阶函数与 Lambda 表达式

高阶函数与 lambda 表达式

Kotlin 函数都是头等的，这意味着它们可以存储在变量与数据结构中，并可以作为参数传给其他高阶函数以及从其他高阶函数返回。可以像操作任何其他非函数值一样对函数进行操作。

为促成这点，作为一门静态类型编程语言的 Kotlin 使用一系列函数类型来表示函数并提供一组特定的语言结构，例如 lambda 表达式。

高阶函数

高阶函数是将函数用作参数或返回值的函数。

高阶函数的一个不错的示例是集合的函数式风格的 `fold`）， 它接受一个初始累积值与一个接合函数，并通过将当前累积值与每个集合元素连续接合起来代入累积值来构建返回值：

```
fun <T, R> Collection<T>.fold(
    initial: R,
    combine: (acc: R, nextElement: T) -> R
): R {
    var accumulator: R = initial
    for (element: T in this) {
        accumulator = combine(accumulator, element)
    }
    return accumulator
}
```

在上述代码中，参数 `combine` 具有函数类型 `(R, T) -> R`，因此 `fold` 接受一个函数作为参数， 该函数接受类型分别为 `R` 与 `T` 的两个参数并返回一个 `R` 类型的值。 在 `for` 循环内部调用该函数，然后将其返回值赋值给 `accumulator`。

为了调用 `fold`，需要传给它一个函数类型的实例作为参数， 而在高阶函数调用处，（下文详述的）lambda 表达 式广泛用于此目的。

```
fun main() {
    //sampleStart
    val items = listOf(1, 2, 3, 4, 5)

    // Lambdas 表达式是花括号括起来的代码块。
    items.fold(0, {
        // 如果一个 lambda 表达式有参数，前面是参数，后跟“->”
        acc: Int, i: Int ->
        print("acc = $acc, i = $i, ")
        val result = acc + i
```

```
        println("result = $result")
        // lambda 表达式中的最后一个表达式是返回值:
        result
    })

    // lambda 表达式的参数类型是可选的，如果能够推断出来的话:
    val joinedToString = items.fold("Elements:", { acc, i -> acc + " " + i })

    // 函数引用也可以用于高阶函数调用:
    val product = items.fold(1, Int::times)
    //sampleEnd
    println("joinedToString = $joinedToString")
    println("product = $product")
}
```

函数类型

Kotlin 使用类似 `(Int) -> String` 的函数类型来处理函数的声明：`val onClick: () -> Unit = ……` 。

这些类型具有与函数签名相对应的特殊表示法——它们的参数和返回值：

- 所有函数类型都有一个圆括号括起来的参数类型列表以及一个返回类型：`(A, B) -> C` 表示接受类型分别为 `A` 与 `B` 两个参数并返回一个 `C` 类型值的函数类型。 参数类型列表可以为空，如 `() -> A` 。 `Unit` 返回类型不可省略。
- 函数类型可以有一个额外的接收者类型，它在表示法中的点之前指定：类型 `A.(B) -> C` 表示可以在 `A` 的接收者对象上以一个 `B` 类型参数来调用并返回一个 `C` 类型值的函数。 带有接收者的函数字面值通常与这些类型一起使用。
- 挂起函数属于函数类型的特殊种类，它的表示法中有一个 `suspend` 修饰符 ，例如 `suspend () -> Unit` 或者 `suspend A.(B) -> C` 。

函数类型表示法可以选择性地包含函数的参数名：`(x: Int, y: Int) -> Point` 。 这些名称可用于表明参数的含义。

如需将函数类型指定为可空，请使用圆括号，如下所示：`((Int, Int) -> Int)?` 。

函数类型还可以使用圆括号进行接合：`(Int) -> ((Int) -> Unit)` 。

> 箭头表示法是右结合的，`(Int) -> (Int) -> Unit` 与前述示例等价，但不等于 `((Int) -> (Int)) -> Unit` 。
>
> 

还可以通过使用类型别名给函数类型起一个别称：

```
typealias ClickHandler = (Button, ClickEvent) -> Unit
```

函数类型实例化

有几种方法可以获得函数类型的实例：

- 使用函数字面值的代码块，采用以下形式之一：
  - lambda 表达式：`{ a, b -> a + b }`，
  - 匿名函数：`fun(s: String): Int { return s.toIntOrNull() ?: 0 }`

  带有接收者的函数字面值可用作带有接收者的函数类型的值。
- 使用已有声明的可调用引用：
  - 顶层、局部、成员、扩展函数：`::isOdd` 、 `String::toInt`，
  - 顶层、成员、扩展属性：`List<Int>::size`，
  - 构造函数：`::Regex`

  这包括指向特定实例成员的绑定的可调用引用：`foo::toString`。
- 使用实现函数类型接口的自定义类的实例：

```
class IntTransformer: (Int) -> Int {
    override operator fun invoke(x: Int): Int = TODO()
}

val intFunction: (Int) -> Int = IntTransformer()
```

如果有足够信息，编译器可以推断变量的函数类型：

```
val a = { i: Int -> i + 1 } // 推断出的类型是 (Int) -> Int
```

带与不带接收者的函数类型非字面值可以互换，其中接收者可以替代第一个参数，反之亦然。例如，`(A, B) -> C` 类型的值可以传给或赋值给期待 `A.(B) -> C` 类型值的地方，反之亦然：

```
fun main() {
    //sampleStart
    val repeatFun: String.(Int) -> String = { times -> this.repeat(times) }
    val twoParameters: (String, Int) -> String = repeatFun // OK

    fun runTransformation(f: (String, Int) -> String): String {
        return f("hello", 3)
    }
    val result = runTransformation(repeatFun) // OK
    //sampleEnd
```

```
    println("result = $result")
}
```

> 默认情况下推断出的是没有接收者的函数类型，即使变量是通过扩展函数引用来初始化的。 如需改变这点，请显式指定变量类型。



函数类型实例调用

函数类型的值可以通过其 `invoke(……)` 操作符调用： `f.invoke(x)` 或者直接 `f(x)` 。

如果该值具有接收者类型，那么应该将接收者对象作为第一个参数传递。 调用带有接收者的函数类型值的另一个方式是在其前面加上接收者对象， 就好比该值是一个扩展函数： `1.foo(2)` 。

例如：

```
fun main() {
    //sampleStart
    val stringPlus: (String, String) -> String = String::plus
    val intPlus: Int.(Int) -> Int = Int::plus

    println(stringPlus.invoke("<-", "->"))
    println(stringPlus("Hello, ", "world!"))

    println(intPlus.invoke(1, 1))
    println(intPlus(1, 2))
    println(2.intPlus(3)) // 类扩展调用
    //sampleEnd
}
```

内联函数

有时使用内联函数可以为高阶函数提供灵活的控制流。

Lambda 表达式与匿名函数

lambda 表达式与匿名函数是函数字面值，函数字面值即没有声明而是立即做为表达式传递的函数。考虑下面的例子：

```
max(strings, { a, b -> a.length < b.length })
```

函数 `max` 是一个高阶函数，因为它接受一个函数作为第二个参数。 其第二个参数是一个表达式，它本身是一个函数，称为函数字面值，它等价于以下具名函数：

```
fun compare(a: String, b: String): Boolean = a.length < b.length
```

Lambda 表达式语法

Lambda 表达式的完整语法形式如下：

```
val sum: (Int, Int) -> Int = { x: Int, y: Int -> x + y }
```

- lambda 表达式总是括在花括号中。
- 完整语法形式的参数声明放在花括号内，并有可选的类型标注。
- 函数体跟在一个 `->` 之后。
- 如果推断出的该 lambda 的返回类型不是 `Unit`，那么该 lambda 主体中的最后一个（或可能是单个）表达式会视为返回值。

如果将所有可选标注都留下，看起来如下：

```
val sum = { x: Int, y: Int -> x + y }
```

传递末尾的 lambda 表达式

按照 Kotlin 惯例，如果函数的最后一个参数是函数，那么作为相应参数传入的 lambda 表达式可以放在圆括号之外：

```
val product = items.fold(1) { acc, e -> acc * e }
```

这种语法也称为拖尾 *lambda* 表达式。

如果该 lambda 表达式是调用时唯一的参数，那么圆括号可以完全省略：

```
run { println("...") }
```

it：单个参数的隐式名称

一个 lambda 表达式只有一个参数很常见。

If the compiler can parse the signature without any parameters, the parameter does not need to be declared and `->` can be omitted. 该参数会隐式声明为 `it`：

```
```

```
ints.filter { it > 0 } // 这个字面值是“(it: Int) -> Boolean”类型的
```

从 lambda 表达式中返回一个值

可以使用限定的返回语法从 lambda 显式返回一个值。 否则，将隐式返回最后一个表达式的值。

因此，以下两个片段是等价的：

```
ints.filter {
    val shouldFilter = it > 0
    shouldFilter
}

ints.filter {
    val shouldFilter = it > 0
    return@filter shouldFilter
}
```

这一约定连同在圆括号外传递 lambda 表达式一起支持 LINQ-风格 的代码：

```
strings.filter { it.length == 5 }.sortedBy { it }.map { it.uppercase() }
```

下划线用于未使用的变量

如果 lambda 表达式的参数未使用，那么可以用下划线取代其名称：

```
map.forEach { (_, value) -> println("$value!") }
```

在 lambda 表达式中解构

在 lambda 表达式中解构是作为解构声明的一部分描述的。

匿名函数

上文的 lambda 表达式语法缺少一个东西——指定函数的返回类型的能力。在大多数情况下，这是不必要的。因为返回类型可以自动推断出来。然而，如果确实需要显式指定， 可以使用另一种语法： 匿名函数。

```
fun(x: Int, y: Int): Int = x + y
```

匿名函数看起来非常像一个常规函数声明，除了其名称省略了。其函数体既可以是表达式（如上所示）也可以是代码块：

```
fun(x: Int, y: Int): Int {
    return x + y
}
```

参数和返回类型的指定方式与常规函数相同，除了能够从上下文推断出的参数类型可以省略：

```
ints.filter(fun(item) = item > 0)
```

匿名函数的返回类型推断机制与正常函数一样：对于具有表达式函数体的匿名函数将自动推断返回类型，但具有代码块函数体的返回类型必须显式指定（或者已假定为 `Unit`）。

> 当匿名函数作为参数传递时，需将其放在括号内。 允许将函数留在圆括号外的简写语法仅适用于 lambda 表达式。



Lambda表达式与匿名函数之间的另一个区别是非局部返回的行为。 一个不带标签的 `return` 语句总是在用 `fun` 关键字声明的函数中返回。这意味着 lambda 表达式中的 `return` 将从包含它的函数返回，而匿名函数中的 `return` 将从匿名函数自身返回。

闭包

Lambda 表达式或者匿名函数（以及局部函数和对象表达式）可以访问其闭包，其中包含在外部作用域中声明的变量。 在 lambda 表达式中可以修改闭包中捕获的变量：

```
var sum = 0
ints.filter { it > 0 }.forEach {
    sum += it
}
print(sum)
```

带有接收者的函数字面值

带有接收者的函数类型，例如 `A.(B) -> C`，可以用特殊形式的函数字面值实例化——带有接收者的函数字面值。

如上所述，Kotlin 提供了（当提供接收者对象时）调用带有接收者的函数类型实例的能力。

在这样的函数字面值内部，传给调用的接收者对象成为隐式的 `this`，以便访问接收者对象的成员而无需任何额外的限定符，亦可使用 `this` 表达式 访问接收者对象。

这种行为与扩展函数类似，扩展函数也允许在函数体内部访问接收者对象的成员。

这里有一个带有接收者的函数字面值及其类型的示例，其中在接收者对象上调用了 `plus`：

```
val sum: Int.(Int) -> Int = { other -> plus(other) }
```

匿名函数语法允许你直接指定函数字面值的接收者类型。 如果你需要使用带接收者的函数类型声明一个变量，并在之后使用它，这会非常有用。

```
val sum = fun Int.(other: Int): Int = this + other
```

当接收者类型可以从上下文推断时，lambda 表达式可以用作带接收者的函数字面值。 One of the most important examples of their usage is type-safe builders:

```
class HTML {
  fun body() { …… }
}

fun html(init: HTML.() -> Unit): HTML {
  val html = HTML()  // 创建接收者对象
  html.init()        // 将该接收者对象传给该 lambda
  return html
}

html {       // 带接收者的 lambda 由此开始
  body()   // 调用该接收者对象的一个方法
}
```

内联函数

使用高阶函数会带来一些运行时的效率损失：每一个函数都是一个对象，并且会捕获一个闭包。 闭包那些在函数体内会访问到的变量的作用域。 内存分配（对于函数对象和类）和虚拟调用会引入运行时间开销。

但是在许多情况下通过内联化 lambda 表达式可以消除这类的开销。 下述函数是这种情况的很好的例子。 `lock()` 函数可以很容易地在调用处内联。 考虑下面的情况：

```
lock(l) { foo() }
```

编译器没有为参数创建一个函数对象并生成一个调用。取而代之，编译器可以生成以下代码：

```
l.lock()
try {
    foo()
} finally {
    l.unlock()
}
```

为了让编译器这么做，需要使用 `inline` 修饰符标记 `lock()` 函数：

```
inline fun <T> lock(lock: Lock, body: () -> T): T { …… }
```

`inline` 修饰符影响函数本身和传给它的 lambda 表达式：所有这些都将内联到调用处。

内联可能导致生成的代码增加。不过如果使用得当（避免内联过大函数）， 性能上会有所提升，尤其是在循环中的“超多态（megamorphic）”调用处。

noinline

如果不希望内联所有传给内联函数的 lambda 表达式参数都内联，那么可以用 `noinline` 修饰符标记不希望内联的函数参数：

```
inline fun foo(inlined: () -> Unit, noinline notInlined: () -> Unit) { …… }
```

可以内联的 lambda 表达式只能在内联函数内部调用或者作为可内联的参数传递。 但是 `noinline` 的 lambda 表达式可以以任何喜欢的方式操作，包括存储在字段中、或者进行传递。

> 如果一个内联函数没有可内联的函数参数并且没有具体化的类型参数，编译器会产生一个警告，因为内联这样的函数很可能并无益处（如果你确认需要内联， 那么可以用 `@Suppress("NOTHING_TO_INLINE")` 注解关掉该警告）。
>
> 

Non-local jump expressions

Returns

在 Kotlin 中，只能对具名或匿名函数使用正常的、非限定的 return 来退出。 要退出一个 lambda 表达式，需要使用一个标签。 在 lambda 表达式内部禁止使用裸 `return`，因为 lambda 表达式不能使包含它的函数返回：

```
fun ordinaryFunction(block: () -> Unit) {
    println("hi!")
}
//sampleStart
fun foo() {
    ordinaryFunction {
        return // 错误：不能使 `foo` 在此处返回
    }
}
//sampleEnd
fun main() {
    foo()
}
```

但是如果 lambda 表达式传给的函数是内联的，该 return 也可以内联。因此可以这样：

```
inline fun inlined(block: () -> Unit) {
    println("hi!")
}
//sampleStart
fun foo() {
    inlined {
        return // OK：该 lambda 表达式是内联的
    }
}
//sampleEnd
fun main() {
    foo()
```

```
}
```

这种返回（位于 lambda 表达式中，但退出包含它的函数）称为非局部返回。 通常会在循环中用到这种结构，其内联函数通常包含：

```
fun hasZeros(ints: List<Int>): Boolean {
    ints.forEach {
        if (it == 0) return true // 从 hasZeros 返回
    }
    return false
}
```

请注意，一些内联函数可能调用传给它们的不是直接来自函数体、而是来自另一个执行上下文的 lambda 表达式参数，例如来自局部对象或嵌套函数。在这种情况下，该 lambda 表达式中也不允许非局部控制流。To indicate that the lambda parameter of the inline function cannot use non-local returns, mark the lambda parameter with the `crossinline` modifier:

```
inline fun f(crossinline body: () -> Unit) {
    val f = object: Runnable {
        override fun run() = body()
    }
    // ……
}
```

Break and continue

> This feature is currently In preview. We're planning to stabilize it in future releases. To opt in, use the `-Xnon-local-break-continue` compiler option. We would appreciate your feedback on it in YouTrack.

Similar to non-local `return`, you can apply `break` and `continue` jump expressions in lambdas passed as arguments to an inline function that encloses a loop:

```
fun processList(elements: List<Int>): Boolean {
    for (element in elements) {
        val variable = element.nullableMethod() ?: run {
            log.warning("Element is null or invalid, continuing...")
            continue
        }
        if (variable == 0) return true
    }
    return false
}
```

具体化的类型参数

有时候需要访问一个作为参数传递的一个类型：

```
fun <T> TreeNode.findParentOfType(clazz: Class<T>): T? {
    var p = parent
    while (p != null && !clazz.isInstance(p)) {
        p = p.parent
    }
    @Suppress("UNCHECKED_CAST")
    return p as T?
}
```

在这里向上遍历一棵树并且检测每个节点是不是特定的类型。 这都没有问题，但是调用处不是很优雅：

```
treeNode.findParentOfType(MyTreeNode::class.java)
```

更好的解决方案是只要传一个类型给该函数，可以按以下方式调用它：

```
treeNode.findParentOfType<MyTreeNode>()
```

为能够这么做，内联函数支持具体化的类型参数，于是可以这样写：

```
inline fun <reified T> TreeNode.findParentOfType(): T? {
    var p = parent
    while (p != null && p !is T) {
        p = p.parent
    }
    return p as T?
}
```

上述代码使用 `reified` 修饰符来限定类型参数使其可以在函数内部访问它， 几乎就像是一个普通的类一样。由于函数是内联的，不需要反射，正常的操作符如 `!is` 和 `as` 现在均可用。此外，还可以按照如上所示的方式调用该函数： `myTree.findParentOfType<MyTreeNodeType>()` 。

虽然在许多情况下可能不需要反射，但仍然可以对一个具体化的类型参数使用它：

```
inline fun <reified T> membersOf() = T::class.members

fun main(s: Array<String>) {
    println(membersOf<StringBuilder>().joinToString("\n"))
}
```

普通的函数（未标记为内联函数的）不能有具体化参数。 不具有运行时表示的类型（例如非具体化的类型参数或者类似于 `Nothing` 的虚构类型）不能用作具体化的类型参数的实参。

内联属性

`inline` 修饰符可用于没有幕后字段的属性的访问器。 你可以标注独立的属性访问器：

```
val foo: Foo
    inline get() = Foo()

var bar: Bar
    get() = ……
    inline set(v) { …… }
```

你也可以标注整个属性，将它的两个访问器都标记为内联（`inline`）：

```
inline var bar: Bar
    get() = ……
    set(v) { …… }
```

在调用处，内联访问器如同内联函数一样内联。

公有 API 内联函数的限制

当一个内联函数是 `public` 或 `protected` 而不是 `private` 或 `internal` 声明的一部分时，就会认为它是一个模块级的公有 API。可以在其他模块中调用它，并且也可以在调用处内联这样的调用。

这带来了一些由模块做这样变更时导致的二进制兼容的风险—— 声明一个内联函数但调用它的模块在它修改后并没有重新编译。

为了消除这种由非公有 API 变更引入的不兼容的风险，公有 API 内联函数体内不允许使用非公有声明，即，不允许使用 `private` 与 `internal` 声明以及其部件。

一个 `internal` 声明可以由 `@PublishedApi` 标注，这会允许它在公有 API 内联函数中使用。当一个 `internal` 内联函数标记有 `@PublishedApi` 时，也会像公有函数一样检测其函数体。

操作符重载

在 Kotlin 中可以为类型提供预定义的一组操作符的自定义实现。这些操作符具有预定义的符号表示（如 `+` 或 `*`）与优先级。为了实现这样的操作符，需要为相应的类型提供一个指定名称的成员函数或扩展函数。这个类型会成为二元操作符左侧的类型及一元操作符的参数类型。

To overload an operator, mark the corresponding function with the `operator` modifier:

```
interface IndexedContainer {
    operator fun get(index: Int)
}
```

When overriding your operator overloads, you can omit `operator`:

```
class OrdersList: IndexedContainer {
    override fun get(index: Int) { /*...*/ }
}
```

一元操作

一元前缀操作符

表达式	翻译为
`+a`	`a.unaryPlus()`
`-a`	`a.unaryMinus()`
`!a`	`a.not()`

这个表是说，当编译器处理例如表达式 `+a` 时，它执行以下步骤：

- 确定 `a` 的类型，令其为 `T`。
- 为接收者 `T` 查找一个带有 `operator` 修饰符的无参函数 `unaryPlus()`，即成员函数或扩展函数。
- 如果函数不存在或不明确，则导致编译错误。
- 如果函数存在且其返回类型为 `R`，那就表达式 `+a` 具有类型 `R`。

> 这些操作以及所有其他操作都针对基本类型做了优化，不会为它们引入函数调用的开销。



以下是如何重载一元减运算符的示例：

```
data class Point(val x: Int, val y: Int)

operator fun Point.unaryMinus() = Point(-x, -y)

val point = Point(10, 20)

fun main() {
   println(-point)  // 输出“Point(x=-10, y=-20)”
}
```

递增与递减

表达式	翻译为
`a++`	`a.inc()` + 见下文
`a--`	`a.dec()` + 见下文

`inc()` 和 `dec()` 函数必须返回一个值，它用于赋值给使用 `++` 或 `--` 操作的变量。它们不应该改变在其上调用 `inc()` 或 `dec()` 的对象。

编译器执行以下步骤来解析后缀形式的操作符，例如 `a++`：

- 确定 `a` 的类型，令其为 `T`。
- 查找一个适用于类型为 `T` 的接收者的、带有 `operator` 修饰符的无参数函数 `inc()`。
- 检测函数的返回类型是 `T` 的子类型。

计算表达式的步骤是：

- 把 `a` 的初始值存储到临时存储 `a0` 中。
- 把 `a0.inc()` 结果赋值给 `a`。
- 把 `a0` 作为该表达式的结果返回。

对于 `a--`，步骤是完全类似的。

对于前缀形式 `++a` 和 `--a` 以相同方式解析，其步骤是：

- 把 `a.inc()` 结果赋值给 `a`。
- 把 `a` 的新值作为该表达式结果返回。

二元操作

算术运算符

表达式	翻译为
`a + b`	`a.plus(b)`
`a - b`	`a.minus(b)`
`a * b`	`a.times(b)`
`a / b`	`a.div(b)`
`a % b`	`a.rem(b)`
`a..b`	`a.rangeTo(b)`
`a..<b`	`a.rangeUntil(b)`

对于此表中的操作，编译器只是解析成翻译为列中的表达式。

下面是一个从给定值起始的 `Counter` 类的示例，它可以使用重载的 `+` 运算符来增加计数：

```
data class Counter(val dayIndex: Int) {
    operator fun plus(increment: Int): Counter {
        return Counter(dayIndex + increment)
    }
}
```

in 操作符

表达式	翻译为
`a in b`	`b.contains(a)`
`a !in b`	`!b.contains(a)`

对于 `in` 和 `!in`，过程是相同的，但是参数的顺序是相反的。

索引访问操作符

表达式	翻译为
`a[i]`	`a.get(i)`
`a[i, j]`	`a.get(i, j)`
`a[i_1, ……, i_n]`	`a.get(i_1, ……, i_n)`
`a[i] = b`	`a.set(i, b)`
`a[i, j] = b`	`a.set(i, j, b)`
`a[i_1, ……, i_n] = b`	`a.set(i_1, ……, i_n, b)`

方括号转换为调用带有适当数量参数的 `get` 和 `set`。

invoke 操作符

表达式	翻译为
`a()`	`a.invoke()`
`a(i)`	`a.invoke(i)`
`a(i, j)`	`a.invoke(i, j)`
`a(i_1, ……, i_n)`	`a.invoke(i_1, ……, i_n)`

圆括号转换为调用带有适当数量参数的 `invoke` 。

广义赋值

表达式	翻译为
`a += b`	`a.plusAssign(b)`
`a -= b`	`a.minusAssign(b)`
`a *= b`	`a.timesAssign(b)`
`a /= b`	`a.divAssign(b)`
`a %= b`	`a.remAssign(b)`

对于赋值操作，例如 `a += b` ，编译器执行以下步骤：

- 如果右列的函数可用：
  - 如果相应的二元函数（即 `plusAssign()` 对应于 `plus()` ）也可用， `a` is a mutable variable, and the return type of `plus` is a subtype of the type of `a` , 那么报告错误（无法区分）。
  - 确保其返回类型是 `Unit` ，否则报告错误。
  - 生成 `a.plusAssign(b)` 的代码。
- 否则试着生成 `a = a + b` 的代码（这里包含类型检测： `a + b` 的类型必须是 `a` 的子类型）。

> 赋值在 Kotlin 中不是表达式。

相等与不等操作符

表达式	翻译为
`a == b`	`a?.equals(b) ?: (b === null)`
`a != b`	`!(a?.equals(b) ?: (b === null))`

这些操作符只使用函数 `equals(other: Any?): Boolean` ， 可以覆盖它来提供自定义的相等性检测实现。不会调用任何其他同名函数（如 `equals(other: Foo)` ）。

> `===` 和 `!==` （同一性检测）不可重载，因此不存在对他们的约定。



这个 `==` 操作符有些特殊：它被翻译成一个复杂的表达式，用于筛选 `null` 值。 `null == null` 总是 true，对于非空的 `x` ， `x == null` 总是 false 而不会调用 `x.equals()` 。

比较操作符

表达式	翻译为
`a > b`	`a.compareTo(b) > 0`
`a < b`	`a.compareTo(b) < 0`
`a >= b`	`a.compareTo(b) >= 0`
`a <= b`	`a.compareTo(b) <= 0`

所有的比较都转换为对 `compareTo` 的调用，这个函数需要返回 `Int` 值

属性委托操作符

`provideDelegate` 、 `getValue` 以及 `setValue` 操作符函数已在委托属性中描述。

具名函数的中缀调用

可以通过中缀函数的调用 来模拟自定义中缀操作符。

构建器

- 类型安全的构建器
- 使用带有构建器类型推断的构建器

类型安全的构建器

通过使用命名得当的函数作为构建器，结合带有接收者的函数字面值， 可以在 Kotlin 中创建类型安全、静态类型的构建器。

类型安全的构建器可以创建基于 Kotlin 的适用于采用半声明方式构建复杂层次数据结构领域专用语言（DSL）。以下是构建器的范例应用场景：

- 使用 Kotlin 代码生成标记语言，例如 HTML 或 XML
- 为 Web 服务器配置路由：Ktor

考虑下面的代码：

```
import com.example.html.* // 参见下文声明

fun result() =
    html {
        head {
            title {+"XML encoding with Kotlin"}
        }
        body {
            h1 {+"XML encoding with Kotlin"}
            p  {+"this format can be used as an alternative markup to XML"}

            // 一个具有属性和文本内容的元素
            a(href = "https://kotlinlang.org") {+"Kotlin"}

            // 混合的内容
            p {
                +"This is some"
                b {+"mixed"}
                +"text. For more see the"
                a(href = "https://kotlinlang.org") {+"Kotlin"}
                +"project"
            }
            p {+"some text"}

            // 以下代码生成的内容
            p {
                for (arg in args)
                    +arg
            }
        }
    }
```

这是完全合法的 Kotlin 代码。 可以在这里在线运行上文代码（修改它并在浏览器中运行）。

实现原理

Assume that you need to implement a type-safe builder in Kotlin. 首先，定义想要构建的模型。在本例中我们需要建模 HTML 标签。 用一些类就可以轻易完成。 例如，`HTML` 是一个描述 `<html>` 标签的类，它定义了像 `<head>` 和 `<body>` 这样的子标签。 （参见下文它的声明。）

现在，让我们回想下为什么可以在代码中这样写：

```
html {
 // ……
}
```

`html` 实际上是一个函数调用，它接受一个 lambda 表达式 作为参数。 该函数定义如下：

```
fun html(init: HTML.() -> Unit): HTML {
    val html = HTML()
    html.init()
    return html
}
```

这个函数接受一个名为 `init` 的参数，该参数本身就是一个函数。 该函数的类型是 `HTML.() -> Unit`，它是一个带接收者的函数类型。 这意味着需要向函数传递一个 HTML 类型的实例（接收者）， 并且可以在函数内部调用该实例的成员。

该接收者可以通过 `this` 关键字访问：

```
html {
    this.head { …… }
    this.body { …… }
}
```

（ `head` 和 `body` 是 `HTML` 的成员函数。）

现在，像往常一样， `this` 可以省略掉了，得到的东西看起来已经非常像一个构建器了：

```
html {
    head { …… }
    body { …… }
}
```

那么，这个调用做什么？ 让我们看看上面定义的 `html` 函数的主体。 它创建了一个 `HTML` 的新实例，然后通过调用作为参数传入的函数来初始化它（在本例中，归结为在HTML实例上调用 `head` 和 `body`），然后返回此实例。 这正是构建器所应做的。

`HTML` 类中的 `head` 和 `body` 函数的定义与 `html` 类似。 唯一的区别是，它们将构建的实例添加到包含 `HTML` 实例的 `children` 集合中：

```
fun head(init: Head.() -> Unit): Head {
    val head = Head()
    head.init()
    children.add(head)
    return head
}

fun body(init: Body.() -> Unit): Body {
    val body = Body()
    body.init()
    children.add(body)
    return body
}
```

实际上这两个函数做同样的事情，所以可以有一个泛型版本， `initTag`：

```
protected fun <T : Element> initTag(tag: T, init: T.() -> Unit): T {
    tag.init()
    children.add(tag)
    return tag
}
```

所以，现在该函数很简单：

```
fun head(init: Head.() -> Unit) = initTag(Head(), init)

fun body(init: Body.() -> Unit) = initTag(Body(), init)
```

并且可以使用它们来构建 `<head>` 和 `<body>` 标签。

这里要讨论的另一件事是如何向标签体中添加文本。在上例中这样写到：

```
html {
    head {
        title {+"XML encoding with Kotlin"}
    }
    // ……
}
```

所以基本上，只是把一个字符串放进一个标签体内部，但在它前面有一个小的 `+`， 所以它是一个函数调用，调用一个前缀 `unaryPlus()` 操作。 该操作实际上是由一个扩展函数 `unaryPlus()` 定义的，该函数是 `TagWithText` 抽象类（ `Title` 的父类）的成员：

```
operator fun String.unaryPlus() {
    children.add(TextElement(this))
}
```

所以，在这里前缀 `+` 所做的事情是把一个字符串包装到一个 `TextElement` 实例中，并将其添加到 `children` 集合中， 以使其成为标签树的一个适当的部分。

所有这些都在上面构建器示例顶部导入的包 `com.example.html` 中定义。 在最后一节中，你可以阅读这个包的完整定义。

作用域控制：@DslMarker

使用 DSL 时，可能会遇到上下文中可以调用太多函数的问题。 可以调用 lambda 表达式内部每个可用的隐式接收者的方法，因此得到一个不一致的结果， 就像在另一个 `head` 内部的 `head` 标记那样：

```
html {
    head {
        head {} // 应该禁止
    }
    // ……
}
```

在这个例子中，必须只有最近层的隐式接收者 `this@head` 的成员可用； `head()` 是外部接收者 `this@html` 的成员，所以调用它一定是非法的。

为了解决这个问题，有一种控制接收者作用域的特殊机制。

为了使编译器开始控制标记，我们只是必须用相同的标记注解来标注在 DSL 中使用的所有接收者的类型。 例如，对于 HTML 构建器，我们声明一个注解 `@HTMLTagMarker` ：

```
@DslMarker
annotation class HtmlTagMarker
```

如果一个注解类使用 `@DslMarker` 注解标注，那么该注解类称为 DSL 标记。

在我们的 DSL 中，所有标签类都扩展了相同的超类 `Tag` 。 只需使用 `@HtmlTagMarker` 来标注超类就足够了，之后，Kotlin 编译器会将所有继承的类视为已标注：

```
@HtmlTagMarker
abstract class Tag(val name: String) { …… }
```

不必用 `@HtmlTagMarker` 标注 `HTML` 或 `Head` 类，因为它们的超类已标注过：

```
class HTML() : Tag("html") { …… }

class Head() : Tag("head") { …… }
```

在添加了这个注解之后，Kotlin 编译器就知道哪些隐式接收者是同一个 DSL 的一部分，并且只允许调用最近层的接收者的成员：

```
html {
    head {
        head { } // 错误：外部接收者的成员
    }
    // ……
}
```

请注意，仍然可以调用外部接收者的成员，但是要做到这一点，你必须明确指定这个接收者：

```
html {
    head {
        this@html.head { } // 可能
    }
    // ……
}
```

You can also apply the `@DslMarker` annotation directly to function types. Simply annotate the `@DslMarker` annotation with `@Target(AnnotationTarget.TYPE)`:

```
@Target(AnnotationTarget.TYPE)
@DslMarker
annotation class HtmlTagMarker
```

As a result, the `@DslMarker` annotation can be applied to function types, most commonly to lambdas with receivers. For example:

```
fun html(init: @HtmlTagMarker HTML.() -> Unit): HTML { ... }

fun HTML.head(init: @HtmlTagMarker Head.() -> Unit): Head { ... }

fun Head.title(init: @HtmlTagMarker Title.() -> Unit): Title { ... }
```

When you call these functions, the `@DslMarker` annotation restricts access to outer receivers in the body of a lambda marked with it unless you specify them explicitly:

```
html {
    head {
        title {
            // Access to title, head or other functions of outer receivers is restrict
ed here.
        }
    }
}
```

Only the nearest receiver's members and extensions are accessible within a lambda, preventing unintended interactions between nested scopes.

com.example.html 包的完整定义

这就是 `com.example.html` 包的定义（只有上面例子中使用的元素）。 它构建一个 HTML 树。代码中大量使用了扩展函数和带有接收者的 lambda 表达式。

```
package com.example.html

interface Element {
    fun render(builder: StringBuilder, indent: String)
}

class TextElement(val text: String) : Element {
    override fun render(builder: StringBuilder, indent: String) {
        builder.append("$indent$text\n")
    }
}

@DslMarker
annotation class HtmlTagMarker

@HtmlTagMarker
abstract class Tag(val name: String) : Element {
    val children = arrayListOf<Element>()
    val attributes = hashMapOf<String, String>()

    protected fun <T : Element> initTag(tag: T, init: T.() -> Unit): T {
        tag.init()
        children.add(tag)
        return tag
    }

    override fun render(builder: StringBuilder, indent: String) {
        builder.append("$indent<$name${renderAttributes()}>\n")
        for (c in children) {
            c.render(builder, indent + "  ")
        }
        builder.append("$indent</$name>\n")
```

```
    }

    private fun renderAttributes(): String {
        val builder = StringBuilder()
        for ((attr, value) in attributes) {
            builder.append(" $attr=\"$value\"")
        }
        return builder.toString()
    }

    override fun toString(): String {
        val builder = StringBuilder()
        render(builder, "")
        return builder.toString()
    }
}

abstract class TagWithText(name: String) : Tag(name) {
    operator fun String.unaryPlus() {
        children.add(TextElement(this))
    }
}

class HTML : TagWithText("html") {
    fun head(init: Head.() -> Unit) = initTag(Head(), init)

    fun body(init: Body.() -> Unit) = initTag(Body(), init)
}

class Head : TagWithText("head") {
    fun title(init: Title.() -> Unit) = initTag(Title(), init)
}

class Title : TagWithText("title")

abstract class BodyTag(name: String) : TagWithText(name) {
    fun b(init: B.() -> Unit) = initTag(B(), init)
    fun p(init: P.() -> Unit) = initTag(P(), init)
    fun h1(init: H1.() -> Unit) = initTag(H1(), init)
    fun a(href: String, init: A.() -> Unit) {
        val a = initTag(A(), init)
        a.href = href
    }
}

class Body : BodyTag("body")
class B : BodyTag("b")
class P : BodyTag("p")
class H1 : BodyTag("h1")

class A : BodyTag("a") {
    var href: String
        get() = attributes["href"]!!
```

```
        set(value) {
            attributes["href"] = value
        }
}

fun html(init: HTML.() -> Unit): HTML {
    val html = HTML()
    html.init()
    return html
}
```

使用带有构建器类型推断的构建器

Kotlin supports *builder type inference* (or builder inference), which can come in useful when you are working with generic builders. It helps the compiler infer the type arguments of a builder call based on the type information about other calls inside its lambda argument.

Consider this example of `buildMap()` usage:

```
fun addEntryToMap(baseMap: Map<String, Number>, additionalEntry: Pair<String, Int>?) {
    val myMap = buildMap {
        putAll(baseMap)
        if (additionalEntry != null) {
            put(additionalEntry.first, additionalEntry.second)
        }
    }
}
```

There is not enough type information here to infer type arguments in a regular way, but builder inference can analyze the calls inside the lambda argument. Based on the type information about `putAll()` and `put()` calls, the compiler can automatically infer type arguments of the `buildMap()` call into `String` and `Number`. Builder inference allows to omit type arguments while using generic builders.

Writing your own builders

Requirements for enabling builder inference

> Before Kotlin 1.7.0, enabling builder inference for a builder function required `-Xenable-builder-inference` compiler option. In 1.7.0 the option is enabled by default.

To let builder inference work for your own builder, make sure its declaration has a builder lambda parameter of a function type with a receiver. There are also two requirements for the receiver type:

1. It should use the type arguments that builder inference is supposed to infer. For example:

   ```
   fun <V> buildList(builder: MutableList<V>.() -> Unit) { ... }
   ```

   > Note that passing the type parameter's type directly like `fun <T> myBuilder(builder: T.() -> Unit)` is not yet supported.



2. It should provide public members or extensions that contain the corresponding type parameters in their signature. For example:

```
class ItemHolder<T> {
    private val items = mutableListOf<T>()

    fun addItem(x: T) {
        items.add(x)
    }

    fun getLastItem(): T? = items.lastOrNull()
}

fun <T> ItemHolder<T>.addAllItems(xs: List<T>) {
    xs.forEach { addItem(it) }
}

fun <T> itemHolderBuilder(builder: ItemHolder<T>.() -> Unit): ItemHolder<T> =
    ItemHolder<T>().apply(builder)

fun test(s: String) {
    val itemHolder1 = itemHolderBuilder { // Type of itemHolder1 is ItemHolder<Str
ing>
        addItem(s)
    }
    val itemHolder2 = itemHolderBuilder { // Type of itemHolder2 is ItemHolder<Str
ing>
        addAllItems(listOf(s))
    }
    val itemHolder3 = itemHolderBuilder { // Type of itemHolder3 is ItemHolder<Str
ing?>
        val lastItem: String? = getLastItem()
        // ...
    }
}
```

Supported features

Builder inference supports:

- Inferring several type arguments

```
fun <K, V> myBuilder(builder: MutableMap<K, V>.() -> Unit): Map<K, V> { ... }
```

- Inferring type arguments of several builder lambdas within one call including interdependent ones

```
fun <K, V> myBuilder(
    listBuilder: MutableList<V>.() -> Unit,
    mapBuilder: MutableMap<K, V>.() -> Unit
): Pair<List<V>, Map<K, V>> =
    mutableListOf<V>().apply(listBuilder) to mutableMapOf<K, V>().apply(mapBuilder
)

fun main() {
    val result = myBuilder(
        { add(1) },
        { put("key", 2) }
    )
    // result has Pair<List<Int>, Map<String, Int>> type
}
```

- Inferring type arguments whose type parameters are lambda's parameter or return types

```
fun <K, V> myBuilder1(
    mapBuilder: MutableMap<K, V>.() -> K
): Map<K, V> = mutableMapOf<K, V>().apply { mapBuilder() }

fun <K, V> myBuilder2(
    mapBuilder: MutableMap<K, V>.(K) -> Unit
): Map<K, V> = mutableMapOf<K, V>().apply { mapBuilder(2 as K) }

fun main() {
    // result1 has the Map<Long, String> type inferred
    val result1 = myBuilder1 {
        put(1L, "value")
        2
    }
    val result2 = myBuilder2 {
        put(1, "value 1")
        // You can use `it` as "postponed type variable" type
        // See the details in the section below
        put(it, "value 2")
    }
}
```

How builder inference works

Postponed type variables

Builder inference works in terms of *postponed type variables*, which appear inside the builder lambda during builder inference analysis. A postponed type variable is a type argument's type, which is in the process of inferring. The compiler uses it to collect type information about the type argument.

Consider the example with `buildList()`:

```
val result = buildList {
    val x = get(0)
}
```

Here `x` has a type of postponed type variable: the `get()` call returns a value of type `E`, but `E` itself is not yet fixed. At this moment, a concrete type for `E` is unknown.

When a value of a postponed type variable gets associated with a concrete type, builder inference collects this information to infer the resulting type of the corresponding type argument at the end of the builder inference analysis. For example:

```
val result = buildList {
    val x = get(0)
    val y: String = x
} // result has the List<String> type inferred
```

After the postponed type variable gets assigned to a variable of the `String` type, builder inference gets the information that `x` is a subtype of `String`. This assignment is the last statement in the builder lambda, so the builder inference analysis ends with the result of inferring the type argument `E` into `String`.

Note that you can always call `equals()`, `hashCode()`, and `toString()` functions with a postponed type variable as a receiver.

Contributing to builder inference results

Builder inference can collect different varieties of type information that contribute to the analysis result. It considers:

- Calling methods on a lambda's receiver that use the type parameter's type

  ```
  val result = buildList {
      // Type argument is inferred into String based on the passed "value" argument
      add("value")
  } // result has the List<String> type inferred
  ```

- Specifying the expected type for calls that return the type parameter's type

  ```
  val result = buildList {
      // Type argument is inferred into Float based on the expected type
      val x: Float = get(0)
  } // result has the List<Float> type
  ```

```
class Foo<T> {
    val items = mutableListOf<T>()
}

fun <K> myBuilder(builder: Foo<K>.() -> Unit): Foo<K> = Foo<K>().apply(builder)

fun main() {
    val result = myBuilder {
        val x: List<CharSequence> = items
        // ...
    } // result has the Foo<CharSequence> type
}
```

- Passing postponed type variables' types into methods that expect concrete types

```
fun takeMyLong(x: Long) { ... }

fun String.isMoreThat3() = length > 3

fun takeListOfStrings(x: List<String>) { ... }

fun main() {
    val result1 = buildList {
        val x = get(0)
        takeMyLong(x)
    } // result1 has the List<Long> type

    val result2 = buildList {
        val x = get(0)
        val isLong = x.isMoreThat3()
    // ...
    } // result2 has the List<String> type

    val result3 = buildList {
        takeListOfStrings(this)
    } // result3 has the List<String> type
}
```

- Taking a callable reference to the lambda receiver's member

```
fun main() {
    val result = buildList {
        val x: KFunction1<Int, Float> = ::get
    } // result has the List<Float> type
}
```

```
fun takeFunction(x: KFunction1<Int, Float>) { ... }
```

```
fun main() {
    val result = buildList {
        takeFunction(::get)
    } // result has the List<Float> type
}
```

At the end of the analysis, builder inference considers all collected type information and tries to merge it into the resulting type. See the example.

```
val result = buildList { // Inferring postponed type variable E
    // Considering E is Number or a subtype of Number
    val n: Number? = getOrNull(0)
    // Considering E is Int or a supertype of Int
    add(1)
    // E gets inferred into Int
} // result has the List<Int> type
```

The resulting type is the most specific type that corresponds to the type information collected during the analysis. If the given type information is contradictory and cannot be merged, the compiler reports an error.

Note that the Kotlin compiler uses builder inference only if regular type inference cannot infer a type argument. This means you can contribute type information outside a builder lambda, and then builder inference analysis is not required. Consider the example:

```
fun someMap() = mutableMapOf<CharSequence, String>()

fun <E> MutableMap<E, String>.f(x: MutableMap<E, String>) { ... }

fun main() {
    val x: Map<in String, String> = buildMap {
        put("", "")
        f(someMap()) // Type mismatch (required String, found CharSequence)
    }
}
```

Here a type mismatch appears because the expected type of the map is specified outside the builder lambda. The compiler analyzes all the statements inside with the fixed receiver type `Map<in String, String>`.

空安全

Null safety is a Kotlin feature designed to significantly reduce the risk of null references, also known as The Billion-Dollar Mistake.

许多编程语言（包括 Java）中最常见的陷阱之一，就是访问空引用的成员会导致空引用异常。在 Java 中，这等同于 `NullPointerException` 或简称 *NPE*。

Kotlin explicitly supports nullability as part of its type system, meaning you can explicitly declare which variables or properties are allowed to be `null`. Also, when you declare non-null variables, the compiler enforces that these variables cannot hold a `null` value, preventing an NPE.

Kotlin's null safety ensures safer code by catching potential null-related issues at compile time rather than runtime. This feature improves code robustness, readability, and maintainability by explicitly expressing `null` values, making the code easier to understand and manage.

Kotlin 中 NPE 的可能的原因只可能是：

- 显式调用 `throw NullPointerException()`。
- 使用了非空断言操作符 `!!`。
- 数据在初始化期间不一致，例如当：
  - 一个在构造函数中出现的未初始化的 `this` 用于其他地方（“泄漏 `this`”）。
  - 超类的构造函数调用一个开放成员，该成员在派生类中的实现使用了未初始化的状态。
- Java 互操作：
  - 企图访问平台类型的 `null` 引用的成员。
  - 泛型类型的可空性问题。例如，一段 Java 代码可能会向 Kotlin 的 `MutableList<String>` 中添加 `null`，这需要 `MutableList<String?>` 才能正确处理。
  - 由外部 Java 代码引发的其他问题。

> Besides NPE, another exception related to null safety is `UninitializedPropertyAccessException`. Kotlin throws this exception when you try to access a property that has not been initialized, ensuring that non-nullable properties are not used until they are ready. This typically happens with `lateinit` properties.
>
> 

可空类型与非空类型

在 Kotlin 中，类型系统区分一个类型可以容纳 `null` （可空类型）还是 不能容纳（非空类型）。例如，String 类型的常规变量不能容纳 `null`：

```
fun main() {
//sampleStart
    // Assigns a non-null string to a variable
    var a: String = "abc"
    // Attempts to re-assign null to the non-nullable variable
    a = null
    print(a)
    // Null can not be a value of a non-null type String
//sampleEnd
}
```

You can safely call a method or access a property on `a`. It's guaranteed not to cause an NPE because `a` is a non-nullable variable. The compiler ensures that `a` always holds a valid `String` value, so there's no risk of accessing its properties or methods when it's `null`:

```
fun main() {
//sampleStart
    // Assigns a non-null string to a variable
    val a: String = "abc"
    // Returns the length of a non-nullable variable
    val l = a.length
    print(l)
    // 3
//sampleEnd
}
```

To allow `null` values, declare a variable with a `?` sign right after the variable type. For example, you can declare a nullable string by writing `String?`. This expression makes `String` a type that can accept `null`:

```
fun main() {
//sampleStart
    // Assigns a nullable string to a variable
    var b: String? = "abc"
    // Successfully re-assigns null to the nullable variable
    b = null
    print(b)
    // null
//sampleEnd
}
```

If you try accessing `length` directly on `b`, the compiler reports an error. This is because `b` is declared as a nullable variable and can hold `null` values. Attempting to access properties on nullables directly leads to an NPE:

```
fun main() {
//sampleStart
    // Assigns a nullable string to a variable
    var b: String? = "abc"
    // Re-assigns null to the nullable variable
    b = null
    // Tries to directly return the length of a nullable variable
    val l = b.length
    print(l)
    // Only safe (?.) or non-null asserted (!!.) calls are allowed on a nullable recei
ver of type String?
//sampleEnd
}
```

In the example above, the compiler requires you to use safe calls to check for nullability before accessing properties or performing operations. There are several ways to handle nullables:

- Check for `null` with the `if` conditional
- Safe call operator `?.`
- Elvis operator `?:`
- Not-null assertion operator `!!`
- Nullable receiver
- `let` function
- Safe casts `as?`
- Collections of a nullable type

Read the next sections for details and examples of `null` handling tools and techniques.

Check for null with the if conditional

When working with nullable types, you need to handle nullability safely to avoid an NPE. One way to handle this is checking for nullability explicitly with the `if` conditional expression.

For example, check whether `b` is `null` and then access `b.length`:

```
fun main() {
//sampleStart
    // Assigns null to a nullable variable
    val b: String? = null
    // Checks for nullability first and then accesses length
    val l = if (b != null) b.length else -1
```

```
    print(l)
    // -1
//sampleEnd
}
```

In the example above, the compiler performs a smart cast to change the type from nullable `String?` to non-nullable `String`. 它还会跟踪所执行检测的相关信息，并允许在 `if` 条件内部调用 `length` 。

同时，也支持更复杂的条件：

```
fun main() {
//sampleStart
    // Assigns a nullable string to a variable
    val b: String? = "Kotlin"

    // Checks for nullability first and then accesses length
    if (b != null && b.length > 0) {
        print("String of length ${b.length}")
        // String of length 6
    } else {
        // Provides alternative if the condition is not met
        print("Empty string")
    }
//sampleEnd
}
```

Note that the example above only works when the compiler can guarantee that `b` doesn't change between the check and its usage, same as the smart cast prerequisites.

Safe call operator

The safe call operator `?.` allows you to handle nullability safely in a shorter form. Instead of throwing an NPE, if the object is `null`, the `?.` operator simply returns `null`:

```
fun main() {
//sampleStart
    // Assigns a nullable string to a variable
    val a: String? = "Kotlin"
    // Assigns null to a nullable variable
    val b: String? = null

    // Checks for nullability and returns length or null
    println(a?.length)
    // 6
    println(b?.length)
    // null
```

```
//sampleEnd
}
```

The `b?.length` expression checks for nullability and 如果 `b` 非空，就返回 `b.length`，否则返回 `null`，这个表达式的类型是 `Int?`。

You can use the `?.` operator with both `var` and `val` variables in Kotlin:

- A nullable `var` can hold a `null` (for example, `var nullableValue: String? = null`) or a non-null value (for example, `var nullableValue: String? = "Kotlin"`). If it's a non-null value, you can change it to `null` at any point.
- A nullable `val` can hold a `null` (for example, `val nullableValue: String? = null`) or a non-null value (for example, `val nullableValue: String? = "Kotlin"`). If it's a non-null value, you cannot change it to `null` subsequently.

安全调用在链式调用中很有用。例如，一个员工 Bob 可能会（或者不会）分配给一个部门。可能有另外一个员工是该部门的负责人。获取 Bob 所在部门负责人（如果有的话）的名字，写作：

```
bob?.department?.head?.name
```

如果链中的任何属性（环节）为 `null`，这个链式调用就会返回 `null`。

也可以将安全调用放在赋值的左侧：

```
person?.department?.head = managersPool.getManager()
```

In the example above, if one of the receivers in the safe call chain is `null`, the assignment is skipped, and the expression on the right is not evaluated at all. For example, if either `person` or `person.department` is `null`, the function is not called. Here's the equivalent of the same safe call but with the `if` conditional:

```
if (person != null && person.department != null) {
    person.department.head = managersPool.getManager()
}
```

Elvis 操作符

When working with nullable types, you can check for `null` and provide an alternative value. For example, if `b` is not `null`, access `b.length`. Otherwise, return an alternative value:

```
fun main() {
```

```
//sampleStart
    // Assigns null to a nullable variable
    val b: String? = null
    // Checks for nullability. If not null, returns length. If null, returns 0
    val l: Int = if (b != null) b.length else 0
    println(l)
    // 0
//sampleEnd
}
```

除了写完整的 `if` 表达式，还可以使用 Elvis 操作符 `?:` 以更简洁的方式处理：

```
fun main() {
//sampleStart
    // Assigns null to a nullable variable
    val b: String? = null
    // Checks for nullability. If not null, returns length. If null, returns a non-nul
l value
    val l = b?.length ?: 0
    println(l)
    // 0
//sampleEnd
}
```

如果 `?:` 左侧表达式不是 `null`，Elvis 操作符就返回其左侧表达式。否则，Elvis 操作符返回右侧表达式。当且仅当左侧为 `null` 时，才会对右侧表达式求值。

因为 `throw` 和 `return` 在 Kotlin 中都是表达式，也可以将它们用在 elvis 操作符右侧。这可能会很方便，例如，检测函数参数：

```
fun foo(node: Node): String? {
    // Checks for getParent(). If not null, it's assigned to parent. If null, returns
null
    val parent = node.getParent() ?: return null
    // Checks for getName(). If not null, it's assigned to name. If null, throws excep
tion
    val name = node.getName() ?: throw IllegalArgumentException("name expected")
    // ……
}
```

非空断言操作符

非空断言操作符（ `!!` ）将任何值转换为非空类型。

When you apply the `!!` operator to a variable whose value is not `null`, it's safely handled as a non-nullable type, and the code executes normally. However, if the value is `null`, the `!!` operator forces it to be treated as non-nullable, which results in an NPE.

When `b` is not `null` and the `!!` operator makes it return its non-null value (which is a `String` in this example), it accesses `length` correctly:

```
fun main() {
//sampleStart
    // Assigns a nullable string to a variable
    val b: String? = "Kotlin"
    // Treats b as non-null and accesses its length
    val l = b!!.length
    println(l)
    // 6
//sampleEnd
}
```

When `b` is `null` and the `!!` operator makes it return its non-null value, and an NPE occurs:

```
fun main() {
//sampleStart
    // Assigns null to a nullable variable
    val b: String? = null
    // Treats b as non-null and tries to access its length
    val l = b!!.length
    println(l)
    // Exception in thread "main" java.lang.NullPointerException
//sampleEnd
}
```

The `!!` operator is particularly useful when you are confident that a value is not `null` and there's no chance of getting an NPE, but the compiler cannot guarantee this due to certain rules. In such cases, you can use the `!!` operator to explicitly tell the compiler that the value is not `null`.

Nullable receiver

You can use extension functions with a nullable receiver type, allowing these functions to be called on variables that might be `null`.

By defining an extension function on a nullable receiver type, you can handle `null` values within the function itself instead of checking for `null` at every place where you call the function.

For example, the `.toString()` extension function can be called on a nullable receiver. When invoked on a `null` value, it safely returns the string `"null"` without throwing an exception:

```
//sampleStart
fun main() {
    // Assigns null to a nullable Person object stored in the person variable
    val person: Person? = null

    // Applies .toString to the nullable person variable and prints a string
    println(person.toString())
    // null
}

// Defines a simple Person class
data class Person(val name: String)
//sampleEnd
```

In the example above, even though `person` is `null`, the `.toString()` function safely returns the string `"null"`. This can be helpful for debugging and logging.

If you expect the `.toString()` function to return a nullable string (either a string representation or `null`), use the safe-call operator `?.`. The `?.` operator calls `.toString()` only if the object is not `null`, otherwise it returns `null`:

```
//sampleStart
fun main() {
    // Assigns a nullable Person object to a variable
    val person1: Person? = null
    val person2: Person? = Person("Alice")

    // Prints "null" if person is null; otherwise prints the result of person.toString
()
    println(person1?.toString())
    // null
    println(person2?.toString())
    // Person(name=Alice)
}

// Defines a Person class
data class Person(val name: String)
//sampleEnd
```

The `?.` operator allows you to safely handle potential `null` values while still accessing properties or functions of objects that might be `null`.

Let function

To handle `null` values and perform operations only on non-null types, you can use the safe call operator `?.` together with the `let` function.

This combination is useful for evaluating an expression, check the result for `null`, and execute code only if it's not `null`, avoiding manual null checks:

```
fun main() {
//sampleStart
    // Declares a list of nullable strings
    val listWithNulls: List<String?> = listOf("Kotlin", null)

    // Iterates over each item in the list
    for (item in listWithNulls) {
        // Checks if the item is null and only prints non-null values
        item?.let { println(it) }
        //Kotlin
    }
//sampleEnd
}
```

安全的类型转换

The regular Kotlin operator for type casts is the `as` operator. However, regular casts can result in an exception if the object is not of the target type.

You can use the `as?` operator for safe casts. It tries to cast a value to the specified type and returns `null` if the value is not of that type:

```
fun main() {
//sampleStart
    // Declares a variable of type Any, which can hold any type of value
    val a: Any = "Hello, Kotlin!"

    // Safe casts to Int using the 'as?' operator
    val aInt: Int? = a as? Int
    // Safe casts to String using the 'as?' operator
    val aString: String? = a as? String

    println(aInt)
    // null
    println(aString)
    // "Hello, Kotlin!"
//sampleEnd
}
```

The code above prints `null` because `a` is not an `Int`, so the cast fails safely. It also prints `"Hello, Kotlin!"` because it matches the `String?` type, so the safe cast succeeds.

可空类型的集合

如果你有一个包含可空元素的集合，并且只想保留其中非空的元素，可使用 filterNotNull() 函数：

```
fun main() {
//sampleStart
    // Declares a list containing some null and non-null integer values
    val nullableList: List<Int?> = listOf(1, 2, null, 4)

    // Filters out null values, resulting in a list of non-null integers
    val intList: List<Int> = nullableList.filterNotNull()

    println(intList)
    // [1, 2, 4]
//sampleEnd
}
```

下一步做什么？

- 了解如何在 Java 与 Kotlin 中处理可空性。
- Learn about generic types that are definitely non-nullable.

相等性

Kotlin 中有两类相等性：

- 结构相等（ `==` ——用 `equals()` 检测）
- 引用相等（ `===` ——检测两个引用指向同一对象）

结构相等

Structural equality verifies if two objects have the same content or structure. 结构相等由 `==` 操作及其否定形式 `!=` 操作判断。 按照约定，像 `a == b` 这样的表达式会翻译成：

```
a?.equals(b) ?: (b === null)
```

如果 `a` 不是 `null` 则调用 `equals(Any?)` 函数。否则（ `a` 是 `null` ），检测 `b` 是否与 `null` 引用相等：

```
fun main() {
    var a = "hello"
    var b = "hello"
    var c = null
    var d = null
    var e = d

    println(a == b)
    // true
    println(a == c)
    // false
    println(c == e)
    // true
}
```

请注意，当与 `null` 显式比较时完全没必要优化你的代码： `a == null` 会被自动转换为 `a === null` 。

In Kotlin, the `equals()` function is inherited by all classes from the `Any` class. By default, the `equals()` function implements referential equality. However, classes in Kotlin can override the `equals()` function to provide a custom equality logic and, in this way, implement structural equality.

Value classes and data classes are two specific Kotlin types that automatically override the `equals()` function. That's why they implement structural equality by default.

However, in the case of data classes, if the `equals()` function is marked as `final` in the parent class, its behavior remains unchanged.

Distinctly, non-data classes (those not declared with the `data` modifier) do not override the `equals()` function by default. Instead, non-data classes implement referential equality behavior inherited from the `Any` class. To implement structural equality, non-data classes require a custom equality logic to override the `equals()` function.

如需提供自定义的相等检测实现，请覆盖 `equals(other: Any?): Boolean` 函数：

```
class Point(val x: Int, val y: Int) {
    override fun equals(other: Any?): Boolean {
        if (this === other) return true
        if (other !is Point) return false

        // Compares properties for structural equality
        return this.x == other.x && this.y == other.y
    }
}
```

> When overriding the equals() function, you should also override the hashCode() function to keep consistency between equality and hashing and ensure a proper behavior of these functions.
>
> 

名称相同但签名不同的函数（如 `equals(other: Foo)`）并不会影响操作符 `==` 与 `!=` 的相等性检测。

结构相等与 `Comparable<……>` 接口定义的比较无关，因此只有自定义的 `equals(Any?)` 实现可能会影响该操作符的行为。

引用相等

Referential equality verifies the memory addresses of two objects to determine if they are the same instance.

引用相等由 `===`（以及其否定形式 `!==`）操作判断。`a === b` 当且仅当 `a` 与 `b` 指向同一个对象时求值为 true：

```
fun main() {
    var a = "Hello"
    var b = a
    var c = "world"
    var d = "world"

    println(a === b)
```

```
    // true
    println(a === c)
    // false
    println(c === d)
    // true

}
```

对于运行时以原生类型表示的值（例如 `Int`），`===` 相等检测等价于 `==` 检测。

> The referential equality is implemented differently in Kotlin/JS. For more information about equality, see the Kotlin/JS documentation.
>
> 

浮点数相等性

当相等性检测的两个操作数都是静态已知的（可空或非空的）`Float` 或 `Double` 类型时，该检测遵循 IEEE 754 浮点数运算标准。

The behavior is different for operands that are not statically typed as floating-point numbers. In these cases, structural equality is implemented. As a result, checks with operands not statically typed as floating-point numbers differ from the IEEE standard. In this scenario:

- `NaN` 等于其自身
- `NaN` 比（包括 `POSITIVE_INFINITY` 在内的）任何其他元素都大
- `-0.0` 不等于 `0.0`

更多信息请参见浮点数比较。

数组相等性

如需比较两个数组是否具有相同顺序的相同元素，请使用 `contentEquals()`。

更多信息请参见比较数组。

this 表达式

表示当前的 接收者 可使用 `this` 表达式：

- 在类的成员中，`this` 指的是该类的当前对象。
- 在扩展函数或者带有接收者的函数字面值中，`this` 表示在点左侧传递的 接收者 参数。

如果 `this` 没有限定符，它指的是最内层的包含它的作用域。要引用其他作用域中的 `this`，请使用 标签限定符：

限定的 this

要访问来自外部作用域的 `this`（一个类 或者扩展函数， 或者带标签的带有接收者的函数字面值）使用 `this@label`， 其中 `@label` 是一个代指 `this` 来源的标签：

```
class A { // 隐式标签 @A
    inner class B { // 隐式标签 @B
        fun Int.foo() { // 隐式标签 @foo
            val a = this@A // A 的 this
            val b = this@B // B 的 this

            val c = this // foo() 的接收者，一个 Int
            val c1 = this@foo // foo() 的接收者，一个 Int

            val funLit = lambda@ fun String.() {
                val d = this // funLit 的接收者，一个 String
            }

            val funLit2 = { s: String ->
                // foo() 的接收者，因为它包含的 lambda 表达式
                // 没有任何接收者
                val d1 = this
            }
        }
    }
}
```

Implicit this

当对 `this` 调用成员函数时，可以省略 `this.` 部分。 但是如果有一个同名的非成员函数时，请谨慎使用，因为在某些情况下会调用同名的非成员：

```
fun main() {
    fun printLine() { println("Local function") }

  class A {
    fun printLine() { println("Member function") }

        fun invokePrintLine(omitThis: Boolean = false) {
      if (omitThis) printLine()
      else this.printLine()
    }
  }

  A().invokePrintLine() // Member function
    A().invokePrintLine(omitThis = true) // Local function
}
```

异步程序设计

几十年以来，作为开发人员，我们面临着需要解决的问题——如何防止我们的应用程序被阻塞。 当我们正在开发桌面应用，移动应用，甚至服务器端应用程序时，我们希望避免让用户等待或导致更糟糕的原因成为阻碍应用程序扩展的瓶颈。

有很多途径来解决这种问题，包括：

- 线程
- 回调
- Future、 Promise 及其他
- 反应式扩展
- 协程

在解释协程的含义之前，让我们简要回顾一些其他解决方案。

线程

到目前为止，线程可能是最常见的避免应用程序阻塞的方法。

```
fun postItem(item: Item) {
    val token = preparePost()
    val post = submitPost(token, item)
    processPost(post)
}

fun preparePost(): Token {
    // 发起请求并因此阻塞了主线程
    return token
}
```

让我们假设在上面的代码中，`preparePost` 是一个长时间运行的进程，因此会阻塞用户界面。我们可以做的是在一个单独的线程中启动它。这样就可以允许我们避免阻塞 UI。这是一种非常常见的技术，但有一系列缺点：

- 线程并非廉价的。线程需要昂贵的上下文切换。
- 线程不是无限的。可被启动的线程数受底层操作系统的限制。在服务器端应用程序中，这可能会导致严重的瓶颈。
- 线程并不总是可用。在一些平台中，比如 JavaScript 甚至不支持线程。
- 线程不容易使用。调试线程与避免竞争条件是我们在多线程编程中遇到的常见问题。

回调

使用回调，其想法是将一个函数作为参数传递给另一个函数，并在处理完成后调用此函数。

```
fun postItem(item: Item) {
    preparePostAsync { token ->
        submitPostAsync(token, item) { post ->
            processPost(post)
        }
    }
}

fun preparePostAsync(callback: (Token) -> Unit) {
    // 发起请求并立即返回
    // 设置稍后调用的回调
}
```

原则上这感觉就像一个更优雅的解决方案，但又有几个问题：

- 回调嵌套的难度。通常被用作回调的函数，经常最终需要自己的回调。这导致了一系列回调嵌套并导致出现难以理解的代码。该模式通常被称为回调地狱，or the pyramid of doom) due to the triangular shape that indentations from these deeply nested callbacks create.
- 错误处理很复杂。嵌套模型使错误处理和传播变得更加复杂。

回调在诸如 JavaScript 之类的事件循环体系结构中非常常见，但即使在那里，通常人们已经转而使用其他方法，例如 promises 或反应式扩展。

Future、 Promise 及其他

futures 或 promises 背后的想法（根据语言或平台的不同可能会使用其他术语），是当我们发起调用的时候，对我们承诺在某些时候该调用会返回一个 `Promise` 对象，然后我们可以对其进行操作。

```
fun postItem(item: Item) {
    preparePostAsync()
        .thenCompose { token ->
            submitPostAsync(token, item)
        }
        .thenAccept { post ->
            processPost(post)
        }

}

fun preparePostAsync(): Promise<Token> {
    // 发起请求并当稍后的请求完成时返回一个 promise
```

```
    return promise
}
```

这种方法需要对我们的编程方式进行一系列更改，尤其是：

- 不同的编程模型。与回调类似，编程模型从自上而下的命令式方法转变为具有链式调用的组合模型。传统的编程结构例如循环，异常处理，等等。通常在此模型中不再有效。
- 不同的 API。通常这需要学习完整的新 API 诸如 `thenCompose` 或 `thenAccept`，这也可能因平台而异。
- 具体的返回值类型。返回类型远离我们需要的实际数据，而是返回一个必须被内省的新类型“Promise”。
- 异常处理会很复杂。错误的传播和链接并不总是直截了当的。

反应式扩展

Erik Meijer) 将反应式扩展（Rx）引入了 C#. 虽然它在 .NET 平台上是毫无疑义的， 但是在 Netflix 将它移植到 Java 并取名为 RxJava 之前绝对不是主流。从那时起，反应式被移植到各种平台，包括 JavaScript（RxJS）。

Rx 背后的想法是走向所谓的“可观察流”，我们现在将数据视为流（无限量的数据），并且可以观察到这些流。 实际上，Rx 很简单， Observer Pattern 带有一系列扩展，允许我们对数据进行操作。

在方法上它与 Futures 非常相似，但是人们可以将 Future 视为一个离散元素，而 Rx 返回一个流。然而，与前面类似，它还介绍了一种全新的思考我们的编程模型的方式，著名的表述是：

```
“一切都是流，并且它是可被观察的”
```

这意味着处理问题的方式不同，并且在编写同步代码时从我们使用的方式发生了相当大的转变。与 Futures 相反的一个好处是，它被移植到这么多平台，通常我们可以找到一致的 API 体验，无论我们使用 C＃、Java、JavaScript，还是 Rx 可用的任何其他语言。

此外，Rx 确实引入了一种更好的错误处理方法。

协程

Kotlin 编写异步代码的方式是使用协程，这是一种计算可被挂起的想法。即一种函数可以在某个时刻暂停执行并稍后恢复的想法。

协程的一个好处是，当涉及到开发人员时，编写非阻塞代码与编写阻塞代码基本相同。编程模型本身并没有真正改变。

以下面的代码为例：

```
fun postItem(item: Item) {
    launch {
        val token = preparePost()
        val post = submitPost(token, item)
        processPost(post)
    }
}

suspend fun preparePost(): Token {
    // 发起请求并挂起该协程
    return suspendCoroutine { /* ... */ }
}
```

此代码将启动长时间运行的操作，而不会阻塞主线程。 `preparePost` 就是所谓的 `可挂起的函数` ，因此它含有 `suspend` 前缀。这意味着如上所述，该函数将被执行、暂停执行以及在某个时间点恢复。

- 该函数的签名保持完全相同。唯一的不同是它被添加了 `suspend` 修饰符。但是返回类型依然是我们想要的类型。
- 编写这段代码代码就好像我们正在编写同步代码，自上而下，不需要任何特殊语法，除了使用一个名为 `launch` 的函数，它实质上启动了该协程（在其他教程中介绍）。
- 编程模型和 API 保持不变。我们可以继续使用循环，异常处理等，而且不需要学习一整套新的 API。
- 它与平台无关。无论我们是面向 JVM，JavaScript 还是其他任何平台，我们编写的代码都是相同的。编译器负责将其适应每个平台。

协程并不是一个新的概念，它并不是 Kotlin 发明的。它们已经存在了几十年，并且在 Go 等其他一些编程语言中很受欢迎。但重要的是要注意就是他们在 Kotlin 中实现的方式，大部分功能都委托给了库。事实上，除了 `suspend` 关键字，没有任何其他关键字被添加到语言中。这也是与其他语言的不同之处，例如 C# 将 `async` 以及 `await` 作为语法的一部分。而在 Kotlin 中，他们都只是库函数。

For more information, see the Coroutines reference.

协程

异步或非阻塞程序设计是开发领域的重要部分。 创建服务器端应用、 桌面应用或者移动端应用时，都很重要的一点是， 提供的体验不仅是从用户角度看着流畅， 而且还能在需要时伸缩（ scalable，可扩充/缩减规模 ）。

Kotlin 以一种灵活的方式解决了这个问题，在语言层面提供了协程支持， 而将大部分功能委托给库。

除了打开了异步程序设计的大门之外，协程还提供了丰富的其他可能性， 例如并发性与参与者。

如何开始

刚接触 Kotlin？可以看看入门页。

文档

- 协程指南
- 基础
- 通道
- 协程上下文与调度器
- 共享的可变状态与并发
- 异步流

教程

- 异步程序设计技术
- 协程与通道简介
- 使用 IntelliJ IDEA 调试协程
- 使用 IntelliJ IDEA 调试 Kotlin Flow——教程
- 在 Android 上测试 Kotlin 协程

范例项目

- kotlinx.coroutines 示例与源代码
- KotlinConf 应用

注解

注解是将元数据附加到代码的方法。要声明注解，请将 `annotation` 修饰符放在类的前面：

```
annotation class Fancy
```

注解的附加属性可以通过用元注解标注注解类来指定：

- `@Target` 指定可以用该注解标注的元素的可能的类型（类、函数、属性与表达式）；
- `@Retention` 指定该注解是否存储在编译后的 class 文件中，以及它在运行时能否通过反射可见（默认都是 true）；
- `@Repeatable` 允许在单个元素上多次使用相同的该注解；
- `@MustBeDocumented` 指定该注解是公有 API 的一部分，并且应该包含在生成的 API 文档中显示的类或方法的签名中。

```
@Target(AnnotationTarget.CLASS, AnnotationTarget.FUNCTION,
        AnnotationTarget.TYPE_PARAMETER, AnnotationTarget.VALUE_PARAMETER,
        AnnotationTarget.EXPRESSION)
@Retention(AnnotationRetention.SOURCE)
@MustBeDocumented
annotation class Fancy
```

用法

```
@Fancy class Foo {
    @Fancy fun baz(@Fancy foo: Int): Int {
        return (@Fancy 1)
    }
}
```

如果需要对类的主构造函数进行标注，则需要在构造函数声明中添加 `constructor` 关键字，并将注解添加到其前面：

```
class Foo @Inject constructor(dependency: MyDependency) { …… }
```

你也可以标注属性访问器：

```
class Foo {
    var x: MyDependency? = null
```

```
        @Inject set
}
```

构造函数

注解可以有接受参数的构造函数。

```
annotation class Special(val why: String)

@Special("example") class Foo {}
```

允许的参数类型有：

- 对应于 Java 原生类型的类型（Int、 Long等）
- 字符串
- 类（`Foo::class`）
- 枚举
- 其他注解
- 上面已列类型的数组

注解参数不能有可空类型，因为 JVM 不支持将 `null` 作为注解属性的值存储。

如果注解用作另一个注解的参数，则其名称不以 `@` 字符为前缀：

```
annotation class ReplaceWith(val expression: String)

annotation class Deprecated(
        val message: String,
        val replaceWith: ReplaceWith = ReplaceWith(""))

@Deprecated("This function is deprecated, use === instead", ReplaceWith("this === othe
r"))
```

如果需要将一个类指定为注解的参数，请使用 Kotlin 类（KClass）。Kotlin 编译器会自动将其转换为 Java 类，以便 Java 代码能够正常访问该注解与参数。

```

import kotlin.reflect.KClass

annotation class Ann(val arg1: KClass<*>, val arg2: KClass<out Any>)

@Ann(String::class, Int::class) class MyClass
```

Instantiation

In Java, an annotation type is a form of an interface, so you can implement it and use an instance. As an alternative to this mechanism, Kotlin lets you call a constructor of an annotation class in arbitrary code and similarly use the resulting instance.

```
annotation class InfoMarker(val info: String)

fun processInfo(marker: InfoMarker): Unit = TODO()

fun main(args: Array<String>) {
    if (args.isNotEmpty())
        processInfo(getAnnotationReflective(args))
    else
        processInfo(InfoMarker("default"))
}
```

Learn more about instantiation of annotation classes in this KEEP.

Lambda 表达式

注解也可以用于 lambda 表达式。它们会被应用于生成 lambda 表达式体的 `invoke()` 方法上。这对于像 Quasar 这样的框架很有用， 该框架使用注解进行并发控制。

```
annotation class Suspendable

val f = @Suspendable { Fiber.sleep(10) }
```

注解使用处目标

当对属性或主构造函数参数进行标注时，从相应的 Kotlin 元素生成的 Java 元素会有多个，因此在生成的 Java 字节码中该注解有多个可能位置 。如果要指定精确地指定应该如何生成该注解，请使用以下语法：

```
class Example(@field:Ann val foo,    // 标注 Java 字段
              @get:Ann val bar,      // 标注 Java getter
              @param:Ann val quux)   // 标注 Java 构造函数参数
```

可以使用相同的语法来标注整个文件。 要做到这一点，把带有目标 `file` 的注解放在文件的顶层、package 指令之前或者在所有导入之前（如果文件在默认包中的话）：

```
@file:JvmName("Foo")

package org.jetbrains.demo
```

如果你对同一目标有多个注解，那么可以这样来避免目标重复——在目标后面添加方括号并将所有注解放在方括号内：

```
class Example {
     @set:[Inject VisibleForTesting]
     var collaborator: Collaborator
}
```

支持的使用处目标的完整列表为：

- `file`
- `property`（具有此目标的注解对 Java 不可见）
- `field`
- `get`（属性 getter）
- `set`（属性 setter）
- `receiver`（扩展函数或属性的接收者参数）
- `param`（构造函数参数）
- `setparam`（属性 setter 参数）
- `delegate`（为委托属性存储其委托实例的字段）

要标注扩展函数的接收者参数，请使用以下语法：

```
fun @receiver:Fancy String.myExtension() { ... }
```

如果不指定使用处目标，则根据正在使用的注解的 `@Target` 注解来选择目标。如果有多个适用的目标，则使用以下列表中的第一个适用目标：

- `param`
- `property`
- `field`

Java 注解

Java 注解与 Kotlin 100% 兼容：

```
import org.junit.Test
import org.junit.Assert.*
import org.junit.Rule
```

```
import org.junit.rules.*

class Tests {
    // 将 @Rule 注解应用于属性 getter
    @get:Rule val tempFolder = TemporaryFolder()

    @Test fun simple() {
        val f = tempFolder.newFile()
        assertEquals(42, getTheAnswer())
    }
}
```

因为 Java 编写的注解没有定义参数顺序，所以不能使用常规函数调用语法来传递参数。相反，你需要使用具名实参语法：

```
// Java
public @interface Ann {
    int intValue();
    String stringValue();
}
```

```
// Kotlin
@Ann(intValue = 1, stringValue = "abc") class C
```

就像在 Java 中一样，一个特殊的情况是 `value` 参数；它的值无需显式名称指定：

```
// Java
public @interface AnnWithValue {
    String value();
}
```

```
// Kotlin
@AnnWithValue("abc") class C
```

数组作为注解参数

如果 Java 中的 `value` 参数具有数组类型，它会成为 Kotlin 中的一个 `vararg` 参数：

```
// Java
public @interface AnnWithArrayValue {
    String[] value();
}
```

```
// Kotlin
@AnnWithArrayValue("abc", "foo", "bar") class C
```

对于具有数组类型的其他参数，你需要显式使用数组字面值语法或者 `arrayOf(……)`：

```
// Java
public @interface AnnWithArrayMethod {
    String[] names();
}
```

```
@AnnWithArrayMethod(names = ["abc", "foo", "bar"])
class C
```

访问注解实例的属性

注解实例的值会作为属性暴露给 Kotlin 代码：

```
// Java
public @interface Ann {
    int value();
}
```

```
// Kotlin
fun foo(ann: Ann) {
    val i = ann.value
}
```

Ability to not generate JVM 1.8+ annotation targets

If a Kotlin annotation has `TYPE` among its Kotlin targets, the annotation maps to `java.lang.annotation.ElementType.TYPE_USE` in its list of Java annotation targets. This is just like how the `TYPE_PARAMETER` Kotlin target maps to the `java.lang.annotation.ElementType.TYPE_PARAMETER` Java target. This is an issue for Android clients with API levels less than 26, which don't have these targets in the API.

To avoid generating the `TYPE_USE` and `TYPE_PARAMETER` annotation targets, use the new compiler argument `-Xno-new-java-annotation-targets`.

Repeatable annotations

Just like in Java, Kotlin has repeatable annotations, which can be applied to a single code element multiple times. To make your annotation repeatable, mark its declaration with the `@kotlin.annotation.Repeatable` meta-annotation. This will make it repeatable both in Kotlin and Java. Java repeatable annotations are also supported from the Kotlin side.

The main difference with the scheme used in Java is the absence of a *containing annotation*, which the Kotlin compiler generates automatically with a predefined name. For an annotation in the example below, it will generate the containing annotation `@Tag.Container`:

```
@Repeatable
annotation class Tag(val name: String)

// The compiler generates the @Tag.Container containing annotation
```

You can set a custom name for a containing annotation by applying the `@kotlin.jvm.JvmRepeatable` meta-annotation and passing an explicitly declared containing annotation class as an argument:

```
@JvmRepeatable(Tags::class)
annotation class Tag(val name: String)

annotation class Tags(val value: Array<Tag>)
```

To extract Kotlin or Java repeatable annotations via reflection, use the `KAnnotatedElement.findAnnotations()` function.

Learn more about Kotlin repeatable annotations in this KEEP.

解构声明

有时把一个对象解构成很多变量会很方便，例如:

```
val (name, age) = person
```

这种语法称为解构声明。一个解构声明同时创建多个变量。 声明了两个新变量： `name` 和 `age`，并且可以独立使用它们：

```
println(name)
println(age)
```

一个解构声明会被编译成以下代码：

```
val name = person.component1()
val age = person.component2()
```

其中的 `component1()` 和 `component2()` 函数是在 Kotlin 中广泛使用的约定原则的另一个示例。（参见像 `+` 和 `*`、`for`-循环作为示例）。 任何表达式都可以出现在解构声明的右侧，只要可以对它调用所需数量的 component 函数即可。 当然，可以有 `component3()` 和 `component4()` 等等。

> `componentN()` 函数需要用 `operator` 关键字标记，以允许在解构声明中使用它们。

解构声明也可以用在 `for`-循环中：

```
for ((a, b) in collection) { …… }
```

变量 `a` 和 `b` 的值取自对集合中的元素上调用 `component1()` 和 `component2()` 的返回值。

例：从函数中返回两个值

让我们假设我们需要从一个函数返回两个东西。例如，一个结果对象和一个某种状态。 在 Kotlin 中一个简洁的实现方式是声明一个数据类 并返回其实例：

```
data class Result(val result: Int, val status: Status)
fun function(……): Result {
```

```
    // 各种计算

    return Result(result, status)
}

// 现在，使用该函数：
val (result, status) = function(……)
```

因为数据类自动声明 `componentN()` 函数，所以这里可以用解构声明。

> 我们也可以使用标准类 `Pair` 并且让 `function()` 返回 `Pair<Int, Status>`，但是让数据合理命名通常更好。



例：解构声明与映射

可能遍历一个映射（map）最好的方式就是这样：

```
for ((key, value) in map) {
   // 使用该 key、value 做些事情
}
```

为使其能用，我们应该

- 通过提供一个 `iterator()` 函数将映射表示为一个值的序列。
- 通过提供函数 `component1()` 和 `component2()` 来将每个元素呈现为一对。

当然事实上，标准库提供了这样的扩展：

```
operator fun <K, V> Map<K, V>.iterator(): Iterator<Map.Entry<K, V>> = entrySet().itera
tor()
operator fun <K, V> Map.Entry<K, V>.component1() = getKey()
operator fun <K, V> Map.Entry<K, V>.component2() = getValue()
```

因此你可以在 `for` -循环中对映射（以及数据类实例的集合等）自由使用解构声明。

下划线用于未使用的变量

如果在解构声明中你不需要某个变量，那么可以用下划线取代其名称：

```
val (_, status) = getResult()
```

对于以这种方式跳过的组件，不会调用相应的 `componentN()` 操作符函数。

在 lambda 表达式中解构

你可以对 lambda 表达式参数使用解构声明语法。 如果 lambda 表达式具有 `Pair` 类型（或者 `Map.Entry` 或任何其他具有相应 `componentN` 函数的类型）的参数，那么可以通过将它们放在括号中来引入多个新参数来取代单个新参数：

```
map.mapValues { entry -> "${entry.value}!" }
map.mapValues { (key, value) -> "$value!" }
```

注意声明两个参数和声明一个解构对来取代单个参数之间的区别：

```
{ a //-> …… } // 一个参数
{ a, b //-> …… } // 两个参数
{ (a, b) //-> …… } // 一个解构对
{ (a, b), c //-> …… } // 一个解构对以及其他参数
```

如果解构的参数中的一个组件未使用，那么可以将其替换为下划线，以避免编造其名称：

```
map.mapValues { (_, value) -> "$value!" }
```

你可以指定整个解构的参数的类型或者分别指定特定组件的类型：

```
map.mapValues { (_, value): Map.Entry<Int, String> -> "$value!" }

map.mapValues { (_, value: String) -> "$value!" }
```

反射

反射是这样的一组语言和库功能，让你可以在运行时自省你的程序的结构。 Kotlin 中的函数和属性是一等公民，而对其自省（即在运行时获悉一个属性或函数的名称或类型）能力是使用函数式或反应式风格时所必需的。

> Kotlin/JS provides limited support for reflection features. Learn more about reflection in Kotlin/JS.
>
> 

JVM dependency

On the JVM platform, the Kotlin compiler distribution includes the runtime component required for using the reflection features as a separate artifact, `kotlin-reflect.jar`. This is done to reduce the required size of the runtime library for applications that do not use reflection features.

To use reflection in a Gradle or Maven project, add the dependency on `kotlin-reflect`:

- In Gradle:

> 【Kotlin】

```
dependencies {
    implementation(kotlin("reflect"))
}
```

> 【Groovy】

```
dependencies {
    implementation "org.jetbrains.kotlin:kotlin-reflect:2.1.21"
}
```

- In Maven:

```
<dependencies>
```

```
        <dependency>
            <groupId>org.jetbrains.kotlin</groupId>
            <artifactId>kotlin-reflect</artifactId>
        </dependency>
    </dependencies>
```

If you don't use Gradle or Maven, make sure you have `kotlin-reflect.jar` in the classpath of your project. In other supported cases (IntelliJ IDEA projects that use the command-line compiler or Ant), it is added by default. In the command-line compiler and Ant, you can use the `-no-reflect` compiler option to exclude `kotlin-reflect.jar` from the classpath.

类引用

最基本的反射功能是获取 Kotlin 类的运行时引用。要获取对静态已知的 Kotlin 类的引用，可以使用 类字面值 语法：

```
val c = MyClass::class
```

该引用是 KClass 类型的值。

> 对于 JVM 平台：Kotlin 类引用与 Java 类引用不同。要获得 Java 类引用， 请在 `KClass` 实例上使用 `.java` 属性。
>
> 

绑定的类引用

通过使用对象作为接收者，可以用相同的 `::class` 语法获取指定对象的类的引用：

```
val widget: Widget = ……
assert(widget is GoodWidget) { "Bad widget: ${widget::class.qualifiedName}" }
```

你会获得对象的精确类的引用，例如 `GoodWidget` 或 `BadWidget`， 尽管接收者表达式的类型是 `Widget`。

可调用引用

函数、属性以及构造函数的引用可以用于调用或者用作函数类型的实例。

所有可调用引用的公共超类型是 `KCallable<out R>`， 其中 `R` 是返回值类型。对于属性是属性类型，对于构造函数是所构造类型。

函数引用

当有一个具名函数声明如下， you can call it directly (`isOdd(5)`):

```
fun isOdd(x: Int) = x % 2 != 0
```

Alternatively, you can use the function as a function type value, that is, pass it to another function. To do so, use the `::` operator:

```
fun isOdd(x: Int) = x % 2 != 0

fun main() {
//sampleStart
    val numbers = listOf(1, 2, 3)
    println(numbers.filter(::isOdd))
//sampleEnd
}
```

这里 `::isOdd` 是函数类型 `(Int) -> Boolean` 的一个值。

函数引用属于 `KFunction<out R>` 的子类型之一，取决于参数个数。例如 `KFunction3<T1, T2, T3, R>` 。

当上下文中已知函数期望的类型时， `::` 可以用于重载函数。 例如：

```
fun main() {
//sampleStart
    fun isOdd(x: Int) = x % 2 != 0
    fun isOdd(s: String) = s == "brillig" || s == "slithy" || s == "tove"

    val numbers = listOf(1, 2, 3)
    println(numbers.filter(::isOdd)) // 引用到 isOdd(x: Int)
//sampleEnd
}
```

或者，你可以通过将方法引用存储在具有显式指定类型的变量中来提供必要的上下文：

```
val predicate: (String) -> Boolean = ::isOdd   // 引用到 isOdd(x: String)
```

如果需要使用类的成员函数或扩展函数，它需要是限定的，例如 `String::toCharArray` 。

即使以扩展函数的引用初始化一个变量，其推断出的函数类型也会没有接收者，但是它会有一个接受接收者对象的额外参数。如需改为带有接收者的函数类型，请明确指定其类型：

```
val isEmptyStringList: List<String>.() -> Boolean = List<String>::isEmpty
```

示例：函数组合

考虑以下函数：

```
fun <A, B, C> compose(f: (B) -> C, g: (A) -> B): (A) -> C {
    return { x -> f(g(x)) }
}
```

它返回一个传给它的两个函数的组合：`compose(f, g) = f(g(*))`。 你可以将该函数应用于可调用引用：

```
fun <A, B, C> compose(f: (B) -> C, g: (A) -> B): (A) -> C {
    return { x -> f(g(x)) }
}

fun isOdd(x: Int) = x % 2 != 0

fun main() {
//sampleStart
    fun length(s: String) = s.length

    val oddLength = compose(::isOdd, ::length)
    val strings = listOf("a", "ab", "abc")

    println(strings.filter(oddLength))
//sampleEnd
}
```

属性引用

要把属性作为 Kotlin中 的一等对象来访问，可以使用 `::` 操作符：

```
val x = 1

fun main() {
    println(::x.get())
    println(::x.name)
}
```

表达式 `::x` 求值为 `KProperty0<Int>` 类型的属性对象，可以使用 `get()` 读取它的值，或者使用 `name` 属性来获取属性名。更多信息请参见关于 `KProperty` 类的文档。

对于可变属性，例如 `var y = 1`， `::y` 返回 `KMutableProperty0<Int>` 类型的一个值， 该类型有一个 `set()` 方法。

```
var y = 1

fun main() {
    ::y.set(2)
    println(y)
}
```

属性引用可以用在预期具有单个泛型参数的函数的地方：

```
fun main() {
//sampleStart
    val strs = listOf("a", "bc", "def")
    println(strs.map(String::length))
//sampleEnd
}
```

要访问属于类的成员的属性，这样限定它：

```
fun main() {
//sampleStart
    class A(val p: Int)
    val prop = A::p
    println(prop.get(A(1)))
//sampleEnd
}
```

对于扩展属性：

```
val String.lastChar: Char
    get() = this[length - 1]

fun main() {
    println(String::lastChar.get("abc"))
}
```

与 Java 反射的互操作性

在 JVM 平台上，标准库包含反射类的扩展，它提供了与 Java 反射对象之间映射（参见 `kotlin.reflect.jvm` 包）。 例如，要查找一个用作 Kotlin 属性 getter 的 幕后字段或 Java方法，可以这样写：

```
import kotlin.reflect.jvm.*

class A(val p: Int)

fun main() {
```

```
    println(A::p.javaGetter) // 输出 "public final int A.getP()"
    println(A::p.javaField)  // 输出 "private final int A.p"
}
```

要获得对应于 Java 类的 Kotlin 类，请使用 `.kotlin` 扩展属性：

```
fun getKClass(o: Any): KClass<Any> = o.javaClass.kotlin
```

构造函数引用

构造函数可以像方法和属性那样引用。可以将其用于程序期待这样函数类型对象的任何地方：它与该构造函数接受相同参数并且返回相应类型的对象。 通过使用 `::` 操作符并添加类名来引用构造函数。考虑下面的函数， 它期待一个无参并返回 `Foo` 类型的函数参数：

```
class Foo

fun function(factory: () -> Foo) {
    val x: Foo = factory()
}
```

使用 `::Foo` ，类 `Foo` 的零参数构造函数，可以这样调用它：

```
function(::Foo)
```

构造函数的可调用引用的类型也是 `KFunction<out R>` 的子类型之一， 取决于其参数个数。

绑定的函数与属性引用

你可以引用特定对象的实例方法：

```
fun main() {
//sampleStart
    val numberRegex = "\\d+".toRegex()
    println(numberRegex.matches("29"))

    val isNumber = numberRegex::matches
    println(isNumber("29"))
//sampleEnd
}
```

取代直接调用方法 `matches` 的是使用其引用。 这样的引用会绑定到其接收者上。 它可以直接调用（如上例所示）或者用于任何期待一个函数类型表达式的时候：

```
fun main() {
//sampleStart
    val numberRegex = "\\d+".toRegex()
    val strings = listOf("abc", "124", "a70")
    println(strings.filter(numberRegex::matches))
//sampleEnd
}
```

比较绑定的引用与未绑定的引用的类型。 绑定的可调用引用有其接收者“附加”到其上，因此接收者的类型不再是参数：

```
val isNumber: (CharSequence) -> Boolean = numberRegex::matches

val matches: (Regex, CharSequence) -> Boolean = Regex::matches
```

属性引用也可以绑定：

```
fun main() {
//sampleStart
    val prop = "abc"::length
    println(prop.get())
//sampleEnd
}
```

无需显式指定 `this` 作为接收者： `this::foo` 与 `::foo` 是等价的。

绑定的构造函数引用

内部类的构造函数的绑定的可调用引用可通过提供外部类的实例来获得：

```
class Outer {
    inner class Inner
}

val o = Outer()
val boundInnerCtor = o::Inner
```

数据分析

- Kotlin Notebook 入门
  - Kotlin Notebook 简介
  - 1、环境搭建
  - 2、创建第一个 Kotlin Notebook
  - 3、将依赖添加到 Kotlin Notebook
- 分享 Kotlin Notebook
- Kotlin Notebook 的输出格式
- 使用数据源
  - 从文件中取数据
  - 从 web 源与 API 取数据
  - 链接数据库并从中取数据
- Kotlin Notebook 中数据可视化
- 用于数据分析的库

Kotlin Notebook 入门

- Kotlin Notebook 简介
- 1、环境搭建
- 2、创建第一个 Kotlin Notebook
- 3、将依赖添加到 Kotlin Notebook

Kotlin Notebook 入门

Kotlin Notebook is an interactive tool that lets you mix code, visuals, and markdown in one document. You can use notebooks to write and execute code in sections known as *code cells*, see the results instantly, and write down your thoughts. This setup makes it an excellent tool for rapid prototyping, analytics, and data science.

Get to know Kotlin Notebook by completing these steps:

1. Set up an environment for working with Kotlin Notebook
2. Create your first notebook and perform some simple operations
3. Add dependencies to your Kotlin Notebook

Next step

Start by setting up an environment for working with Kotlin Notebook:

Proceed to the next part

环境搭建

This is the first part of the **Getting started with Kotlin Notebook** tutorial:

1 **Set up an environment**

2 Create a Kotlin Notebook

3 Add dependencies to a Kotlin Notebook

Before you create your first Kotlin Notebook, you need to set up an environment.

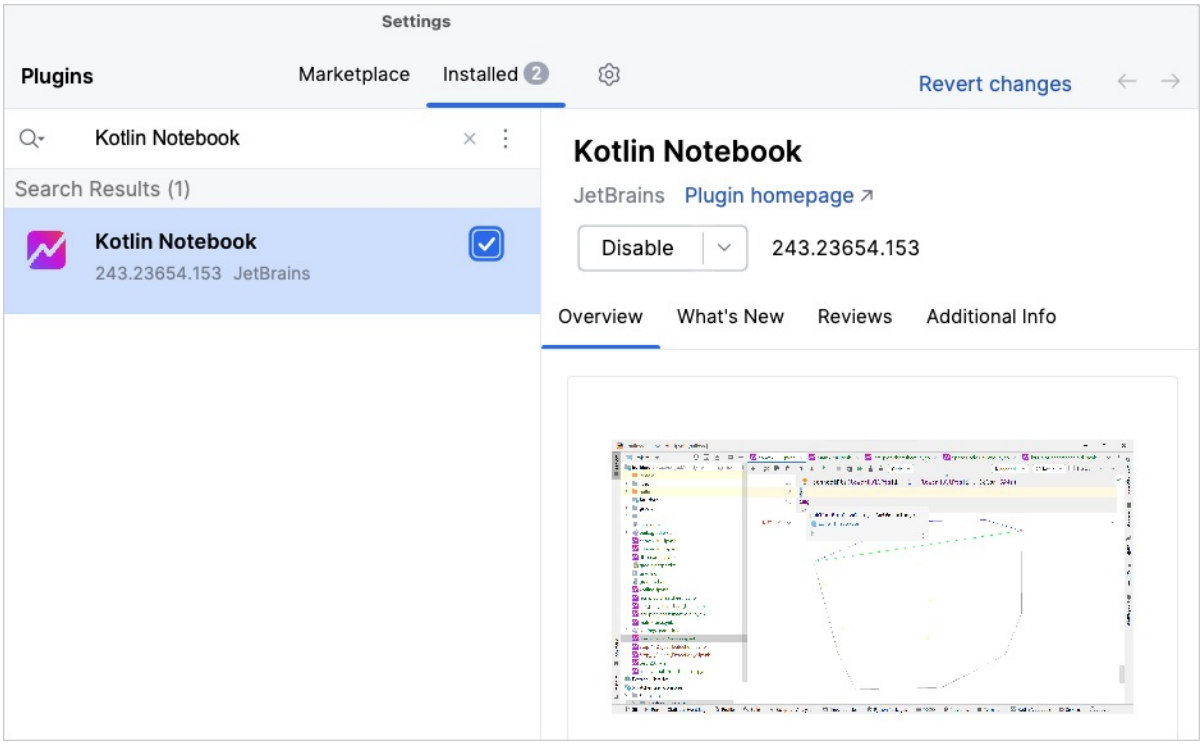
Set up the environment

Kotlin Notebook relies on the Kotlin Notebook plugin, which is bundled and enabled in IntelliJ IDEA by default.

To use Kotlin Notebook, download and install the latest version of IntelliJ IDEA.

If the Kotlin Notebook features are not available, ensure the plugin is enabled:

1. In IntelliJ IDEA, select **IntelliJ IDEA | Settings | Plugins**.
2. In the **Installed** tab, find the **Kotlin Notebook** plugin, and select the checkbox next to the plugin name.



3. Click **OK** to apply the changes and restart your IDE if prompted.

Next step

In the next part of the tutorial, you will learn how to create a Kotlin Notebook.

Proceed to the next chapter

创建第一个 Kotlin Notebook

This is the second part of the **Getting started with Kotlin Notebook** tutorial. Before proceeding, make sure you've completed the previous step.

1 Set up an environment

2 **Create a Kotlin Notebook**

3 Add dependencies to a Kotlin Notebook

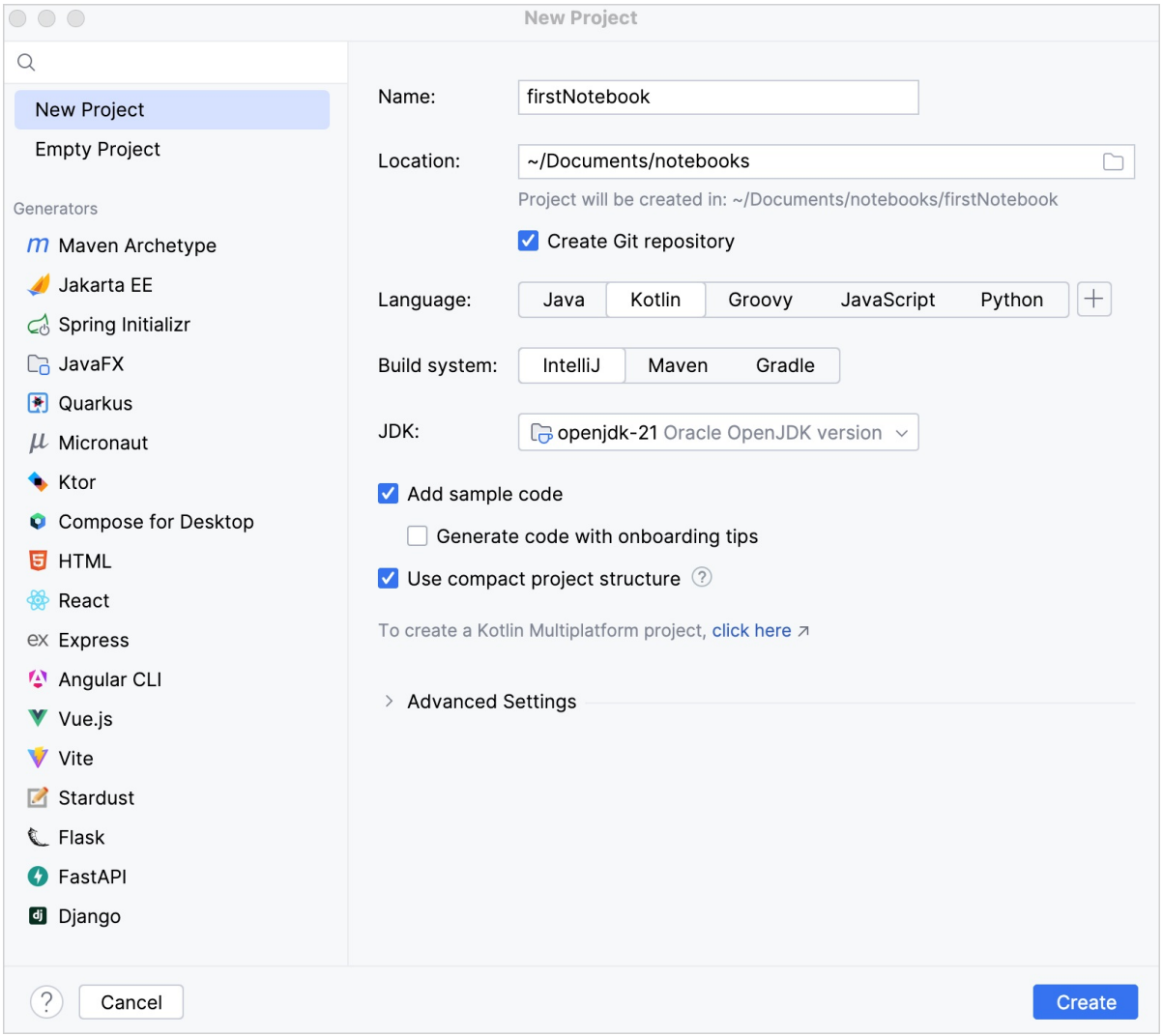
Here, you will learn how to create your first Kotlin Notebook, perform simple operations, and run code cells.

Create an empty project

1. In IntelliJ IDEA, select **File | New | Project**.
2. In the panel on the left, select **New Project**.
3. Name the new project and change its location if necessary.

   > Select the **Create Git repository** checkbox to place the new project under version control. You will be able to do it later at any time.
   >
   > 

4. From the **Language** list, select **Kotlin**.



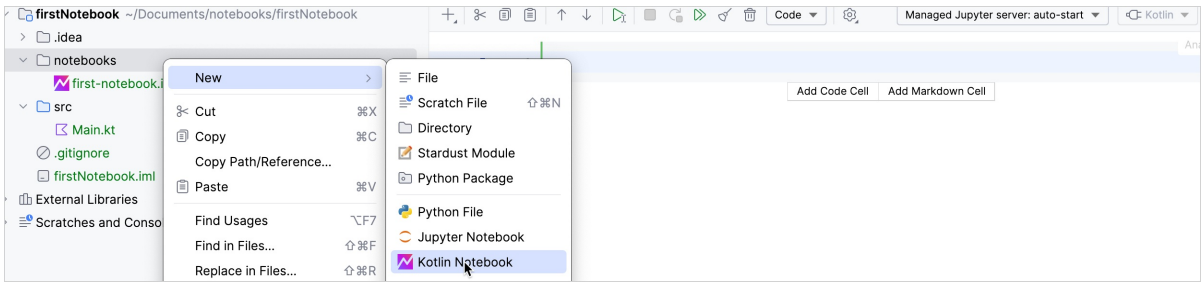
5. Select the **IntelliJ** build system.
6. From the **JDK** list, select the JDK that you want to use in your project.
7. Enable the **Add sample code** option to create a file with a sample `"Hello World!"` application.

> You can also enable the **Generate code with onboarding tips** option to add some additional useful comments to your sample code.
>
> 

8. Click **Create**.

Create a Kotlin Notebook

1. To create a new notebook, select **File | New | Kotlin Notebook**, or right-click on a folder and select **New | Kotlin Notebook**.



2. Set the name of the new notebook, for example, **first-notebook**, and press **Enter**. A new tab with a Kotlin Notebook **first-notebook.ipynb** will open.

3. In the open tab, type the following code in the code cell:

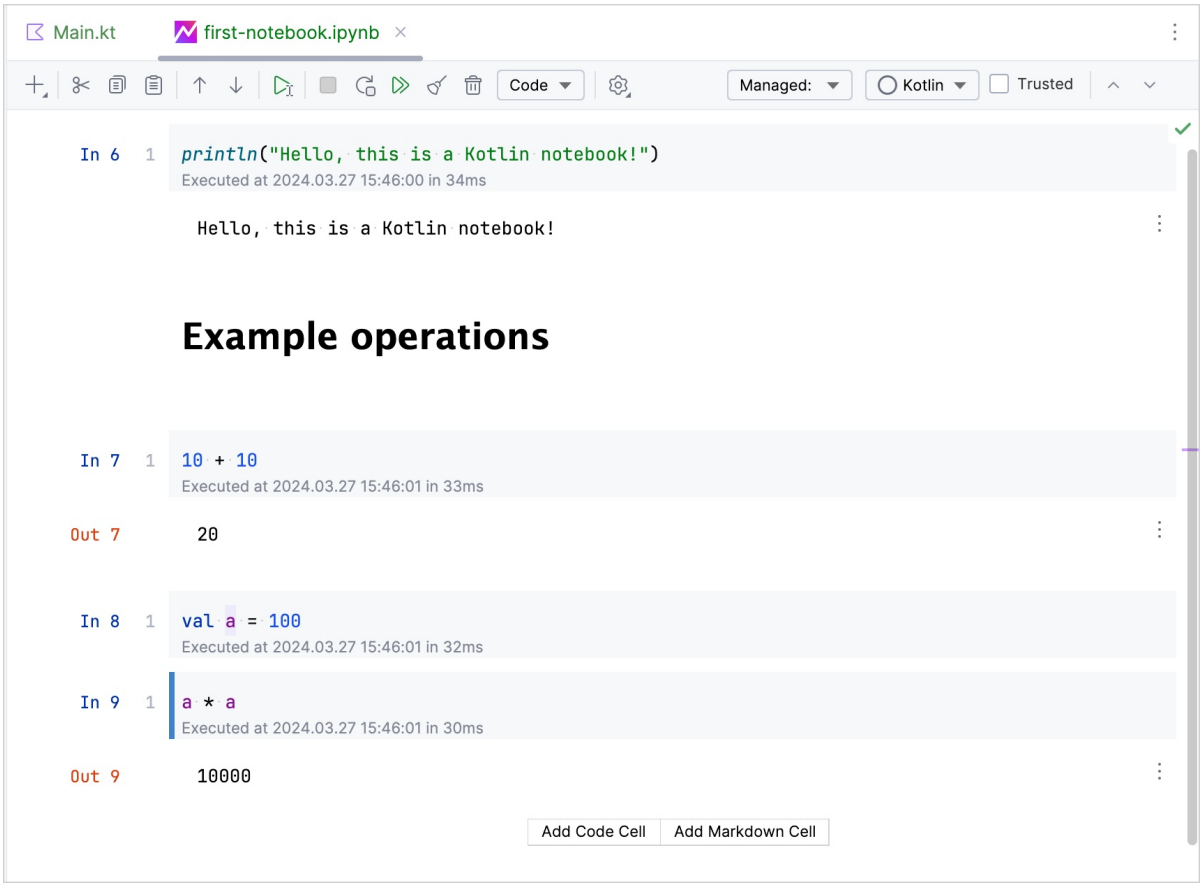
```
println("Hello, this is a Kotlin Notebook!")
```

4. To run a code cell, click the **Run Cell and Select Below**  button or press **Shift** + **Return**.
5. Add a markdown cell by clicking on the **Add Markdown Cell** button.
6. Type `# Example operations` in the cell, and run it the same way you run code cells to render it.
7. In a new code cell, type `10 + 10` and run it.
8. Define a variable in a code cell. For example, `val a = 100`.

> Once you run a code cell with defined variables, those variables become accessible in all other code cells.
>
> 

9. Create a new code cell and add `println(a * a)`.

10. Run all code and markdown cells in the notebook using the **Run All**  button.



Congratulations! You have just created your first Kotlin Notebook.

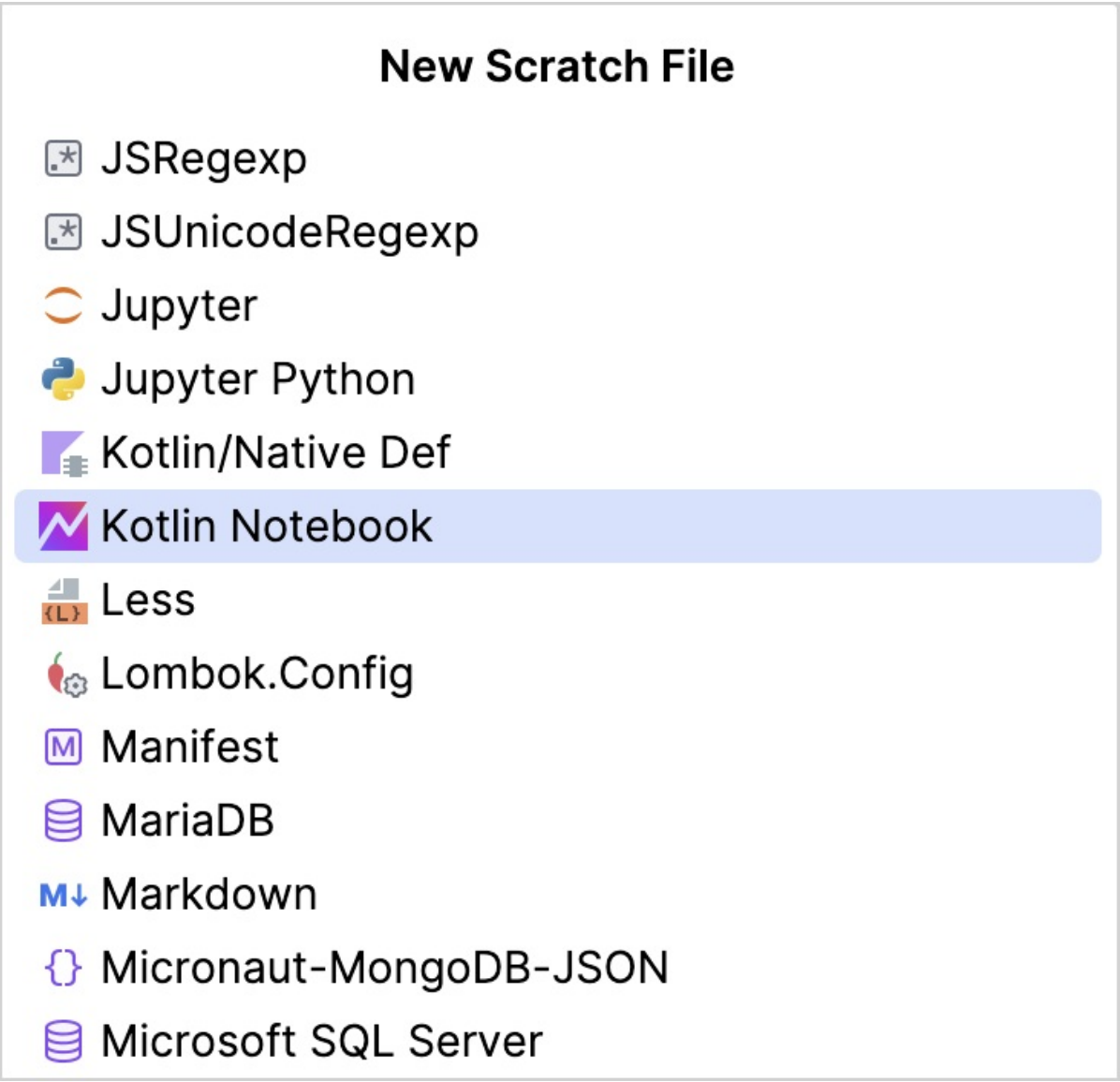
Create a scratch Kotlin Notebook

Starting from IntelliJ IDEA 2024.1.1, you can also create a Kotlin Notebook as a scratch file.

Scratch files allow you to test small pieces of code without creating a new project or modifying an existing one.

To create a scratch Kotlin Notebook:

1. Click **File | New | Scratch File**.
2. Select **Kotlin Notebook** from the **New Scratch File** list.



Next step

In the next part of the tutorial, you will learn how to add dependencies to a Kotlin Notebook.

Proceed to the next chapter

将依赖添加到 Kotlin Notebook

This is the third part of the **Getting started with Kotlin Notebook** tutorial. Before proceeding, make sure you've completed the previous steps.

1. Set up an environment

2. Create a Kotlin Notebook

3. **Add dependencies to a Kotlin Notebook**

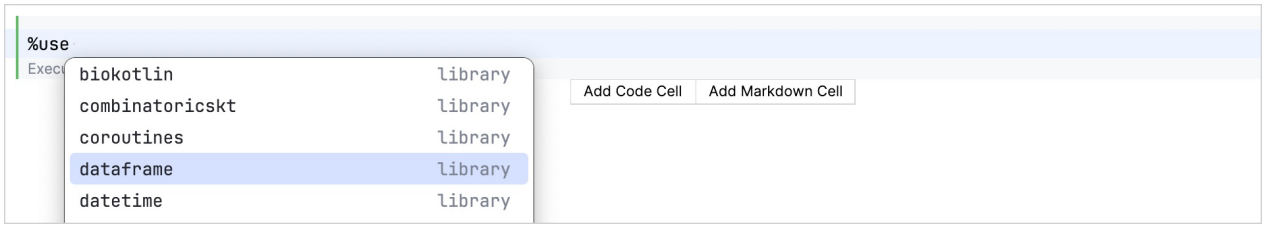
You've already created your first Kotlin Notebook! Now let's learn how to add dependencies to libraries, which is necessary to unlock advanced features.

> The Kotlin standard library can be used out of the box, so you don't have to import it.
>
> 

You can load any library from the Maven repository by specifying its coordinates using Gradle-style syntax in any code cell. However, Kotlin Notebook has a simplified method to load popular libraries in the form of the `%use` statement:

```
// Replace libraryName with the library dependency you want to add
%use libraryName
```

You can also use the autocompletion feature in Kotlin Notebook to quickly access available libraries:



Add Kotlin DataFrame and Kandy libraries to your Kotlin Notebook

Let's add two popular Kotlin library dependencies to your Kotlin Notebook:

- The Kotlin DataFrame library gives you the power to manipulate data in your Kotlin projects. You can use it to retrieve data from APIs, SQL databases, and various file formats, such as CSV or JSON.
- The Kandy library provides a powerful and flexible DSL for creating charts.

To add these libraries:

1. Click **Add Code Cell** to create a new code cell.
2. Enter the following code in the code cell:

```
// Ensures that the latest available library versions are used
%useLatestDescriptors

// Imports the Kotlin DataFrame library
%use dataframe

// Imports the Kotlin Kandy library
%use kandy
```

3. Run the code cell.

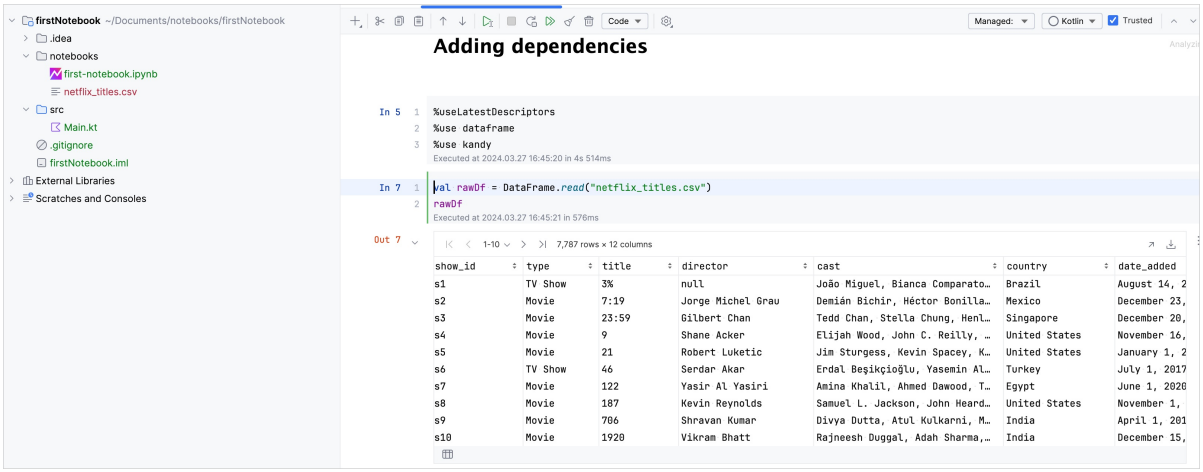
   When a `%use` statement is executed, it downloads the library dependencies and adds the default imports to your notebook.

   > Make sure to run the code cell with the `%use libraryName` line before you run any other code cells that rely on the library.
   >
   > 

4. To import data from a CSV file using the Kotlin DataFrame library, use the `.read()` function in a new code cell:

```
// Creates a DataFrame by importing data from the "netflix_titles.csv" file.
val rawDf = DataFrame.read("netflix_titles.csv")

// Displays the raw DataFrame data
rawDf
```

   > You can download this example CSV from the Kotlin DataFrame examples GitHub repository. Add it to your project directory.



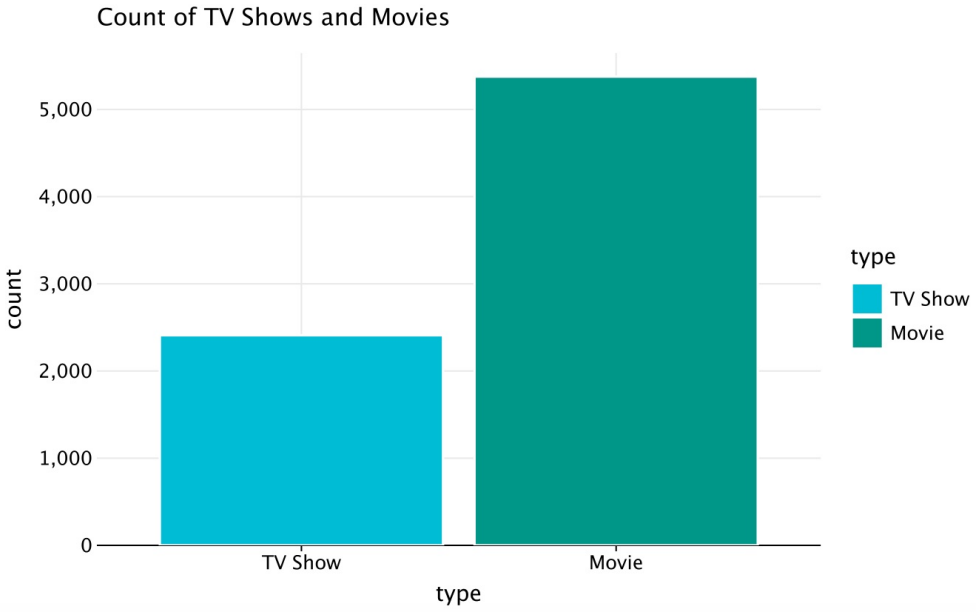
5. In a new code cell, use the `.plot` method to visually represent the distribution of TV shows and Movies in your DataFrame:

```
rawDf
    // Counts the occurrences of each unique value in the column named "type"
    .valueCounts(sort = false) { type }
    // Visualizes data in a bar chart specifying the colors
    .plot {
        bars {
            x(type)
            y("count")
            fillColor(type) {
                scale = categorical(range = listOf(Color.hex("#00BCD4"), Color.he
x("#009688")))
            }
        }

        // Configures the layout of the chart and sets the title
        layout {
            title = "Count of TV Shows and Movies"
            size = 900 to 550
        }
    }
```

The resulting chart:

```
rawDf
    .valueCounts(sort = false) { type }
    .plot { this: DataFramePlotContext<_DataFrameType>
        bars { this: BarsContext
            x(type)
            y("count")
            fillColor(type) { this: LetsPlotNonPositionalMappingParametersContinuous<String, Color>
                scale = categorical(range = listOf(Color.hex("#00BCD4"), Color.hex("#009688")))
            }
        }

        layout { this: Layout
            title = "Count of TV Shows and Movies"
            size = 900 to 550
        }
    }
```

Executed at 2024.03.27 16:45:21 in 229ms



Congratulations on adding and utilizing these libraries in your Kotlin Notebook! This is just a glimpse into what you can achieve with Kotlin Notebook and its supported libraries.

下一步做什么

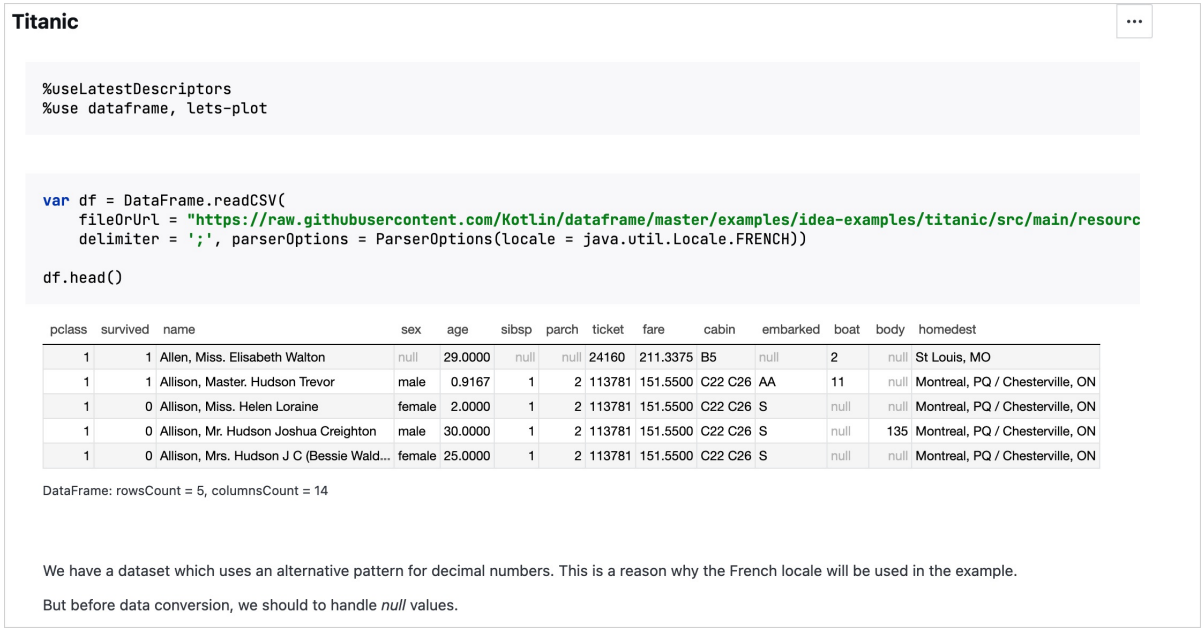
- Learn how to share your Kotlin Notebook
- See more details about adding dependencies to your Kotlin Notebook
- For a more extensive guide using the Kotlin DataFrame library, see Retrieve data from files
- For an extensive overview of tools and resources available for data science and analysis in Kotlin, see Kotlin and Java libraries for data analysis

分享 Kotlin Notebook

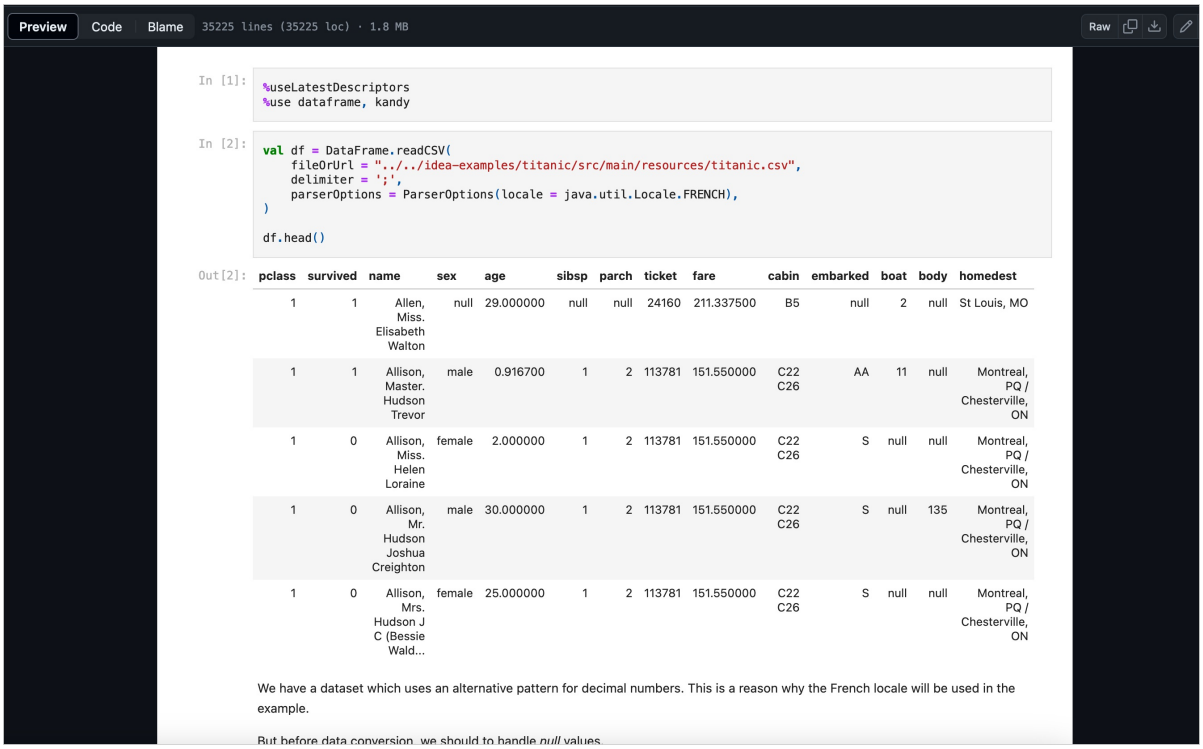
To share a Kotlin Notebook, you can just upload it to any notebook web viewer because Kotlin notebooks follow the universal Jupyter format.

We recommend the following platforms to share Kotlin notebooks:

- **JetBrains Datalore:** This platform not only facilitates the sharing of Kotlin notebooks but also improves their usability. Datalore allows you to execute and edit notebooks, and incorporates advanced features, such as creating interactive reports and scheduling notebook runs. To see it in action, see Kotlin Datalore example using DataFrame.



- **GitHub**: GitHub natively renders Kotlin notebooks, allowing for straightforward sharing and collaboration. For an example, see the Examples of Kotlin DataFrame GitHub repository.



下一步做什么

- Explore data visualization using the Kandy library
- Learn about retrieving data from files, web sources, or databases in Working with data sources
- For an extensive overview of tools and resources available for data science and analysis in Kotlin, see Kotlin and Java libraries for data analysis

Kotlin Notebook 支持的输出格式

Kotlin Notebook supports a variety of output types, including text, HTML, and images. With the help of external libraries, you can expand your output options and visualize your data with charts, spreadsheets, and more.

Each output is a JSON object that maps the Jupiter MIME type to some data. From this map, Kotlin Notebook selects the supported MIME type with the highest priority among other types and renders it like this:

- Text uses the `text/plain` MIME type.
- The BufferedImage class uses the `image/png` MIME type that is mapped to a Base64 string.
- The Image class, as well as the LaTeX format, use the `text/html` MIME type with the `img` tag inside.
- Kotlin DataFrame tables and Kandy plots use their own internal MIME types, which are backed by static HTML or images. This way, you can display them on GitHub.

You can set up the mapping manually, for example, to use Markdown as a cell output:

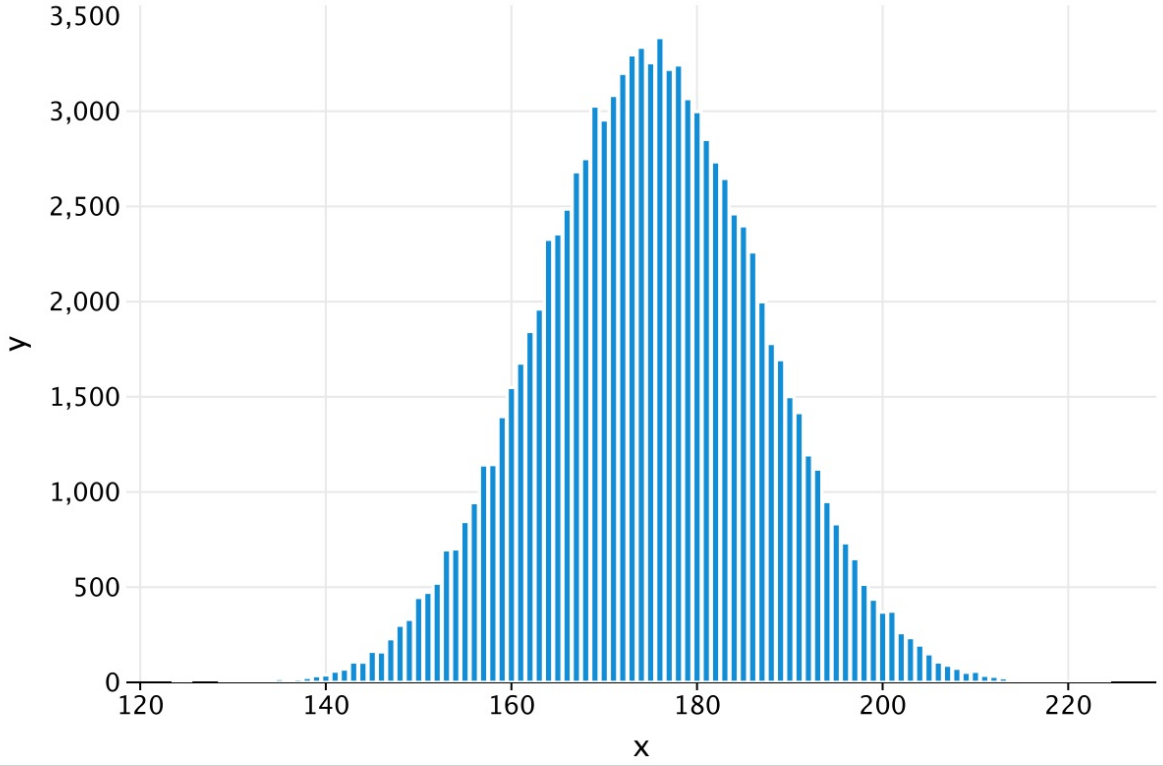
```
MimeTypedResult(
    mapOf(
        "text/plain" to "123",
        "text/markdown" to "# HEADER",
        //other mime:value pairs
    )
)
```

To display any kind of output, use the `DISPLAY()` function. It also enables the combination of several outputs:

```
DISPLAY(HTML("<h2>Gaussian distribution</h2>"))
DISPLAY(LATEX("f(x) = \\frac{1}{\\sigma \\sqrt{2\\pi}} \\cdot e^{-\\frac{(x - \\mu)^2}{2\\sigma^2}}"))

val experimentX = experimentData.map { it.key }
val experimentY = experimentData.map { it.value }

DISPLAY(plot {
    bars {
        x(experimentX)
        y(experimentY)
    }
})
```

Gaussian distribution

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} \cdot e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$



Texts

Plain text

The simplest output type is plain text. It's used in printed statements, variable values, or any text-based output from your code:

```
val a1: Int = 1
val a2: Int = 2
var a3: Int? = a1 + a2

"My answer is $a3"
```

```
Out 13          My answer is 3
```



- If a cell's result cannot be rendered and displayed as any of the output types, it will be printed as plain text using the `toString()` function.
- If your code contains errors, Kotlin Notebook displays an error message and a traceback, providing insights for debugging.

Rich text

Choose cells of the Markdown type to use rich text. This way, you can format the content with Markdown and HTML markup, using lists, tables, font styles, code blocks, and more. HTML can contain CSS styles and JavaScript.

```
## Line magics

| Spell                              | Description
                                                                  | Example
                                                                    |
|-----------------------------------|-----------------------------------------------
-----------------------------------------------------------------|-------------------
-----------------------------------------------------------------|
| <code>%use</code>                  | Injects code for supported libraries: artifact
resolution, default imports, initialization code, type renderers. | <code>%use klaxon(
5.5), lets-plot</code>                                              |

| <code>%trackClasspath</code>       | Logs any changes of current classpath. Useful f
or debugging artifact resolution failures.                        | <code>%trackClassp
ath [on |off]</code>                                                |
| <code>%trackExecution</code>       | Logs pieces of code that are going to be execut
ed. Useful for debugging of libraries support.                    | <code>%trackExecut
ion [all|generated|off]</code>                                      |
| <code>%useLatestDescriptors</code> | Use latest versions of library descriptors avai
lable. By default, bundled descriptors are used.                  | <code>%useLatestDe
scriptors [on|off]</code>                                           |
| <code>%output</code>               | Output capturing settings.
                                                                  | <code>%output --ma
x-cell-size=1000 --no-stdout --max-time=100 --max-buffer=400</code> |
| <code>%logLevel</code>             | Set logging level.
                                                                  | <code>%logLevel [o
ff|error|warn|info|debug]</code>                                     |

<ul><li><a href="https://github.com/Kotlin/kotlin-jupyter/blob/master/docs/magics.md">
Learn more detailes about line magics</a>.</li>
<li><a href="https://github.com/Kotlin/kotlin-jupyter/blob/master/docs/magics.md">See
the full list of supported libraries</a>.</li></ul>
```

Line magics

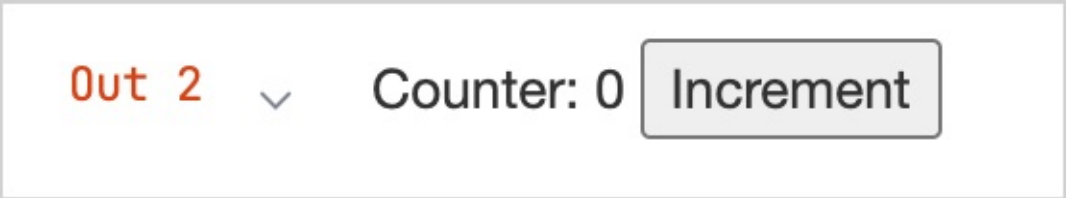
Spell	Description	Example
`%use`	Injects code for supported libraries: artifact resolution, default imports, initialization code, type renderers.	`%use klaxon(5.5), lets-plot`
`%trackClasspath`	Logs any changes of current classpath. Useful for debugging artifact resolution failures.	`%trackClasspath [on |off]`
`%trackExecution`	Logs pieces of code that are going to be executed. Useful for debugging of libraries support.	`%trackExecution [all|generated|off]`
`%useLatestDescriptors`	Use latest versions of library descriptors available. By default, bundled descriptors are used.	`%useLatestDescriptors [on|off]`
`%output`	Output capturing settings.	`%output --max-cell-size=1000 --no-stdout --max-time=100 --max-buffer=400`
`%logLevel`	Set logging level.	`%logLevel [off|error|warn|info|debug]`

- Learn more detailes about line magics.
- See the full list of supported libraries.

HTML

Kotlin Notebook can render HTML directly, executing scripts or even embedding websites:

```
HTML("""
<p>Counter: <span id="ctr">0</span> <button onclick="inc()">Increment</button></p>
<script>
    function inc() {
        let counter = document.getElementById("ctr")
        counter.innerHTML = parseInt(counter.innerHTML) + 1;
}
</script>
""")
```



> Mark your notebook as **Trusted** at the top of the file to be able to execute scripts.

Images

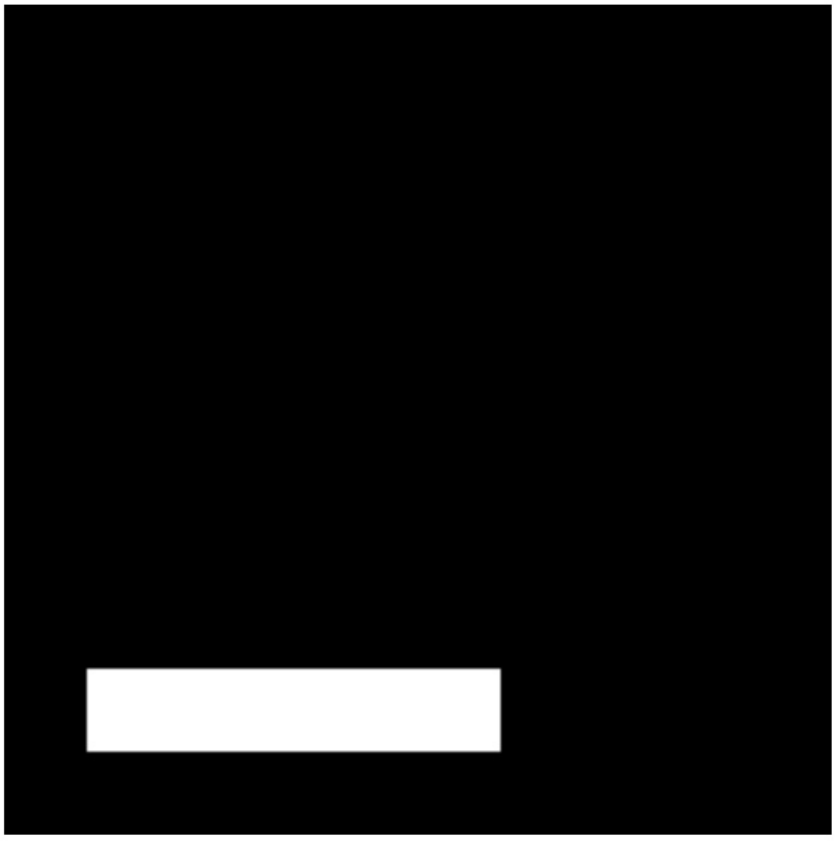
With Kotlin Notebook, you can display images from files, generated graphs, or any other visual media. Static images can be displayed in formats such as `.png`, `jpeg`, and `.svg`.

Buffered images

By default, you can use `BufferedImage` class to display images:

```
import java.awt.Color
import java.awt.image.BufferedImage

val width = 300
val height = width

val image = BufferedImage(width, height, BufferedImage.TYPE_INT_ARGB)

val graphics = image.createGraphics()
graphics.background = Color.BLACK
graphics.clearRect(0, 0, width, height)
graphics.setRenderingHint(
    java.awt.RenderingHints.KEY_ANTIALIASING,
    java.awt.RenderingHints.VALUE_ANTIALIAS_ON
)
graphics.color = Color.WHITE
graphics.fillRect(width / 10, height * 8 / 10, width * 10 / 20, height / 10)
graphics.dispose()
```

Out 6



Loaded images

With the help of the `lib-ext` library, you can extend the standard Jupyter functionality and display images loaded from the network:

```
%use lib-ext(0.11.0-398)
```

```
Image("https://kotlinlang.org/docs/images/kotlin-logo.png", embed = false).withWidth(300)
```

Out 7

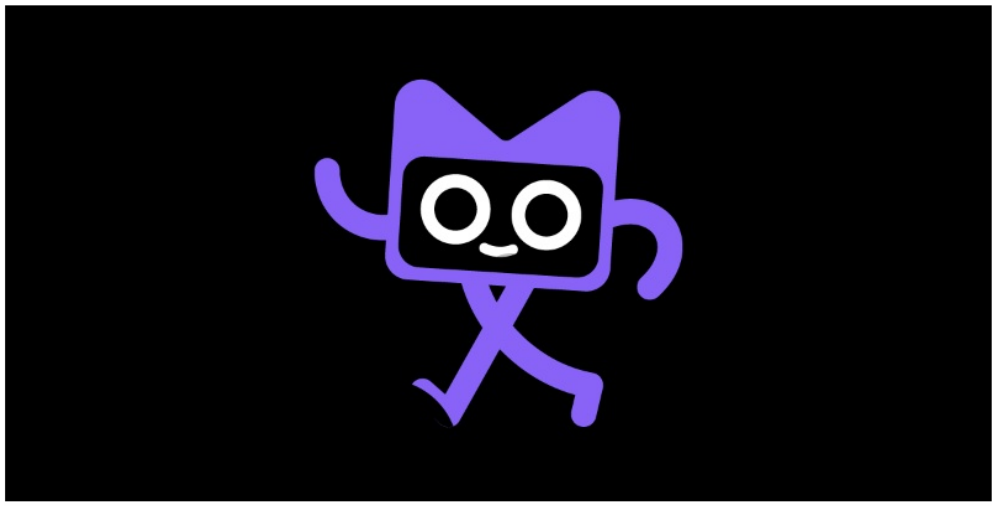


Embedded images

A disadvantage of images loaded from the network is that the image disappears if the link breaks or if you lose the network connection. To work around that, use embedded images, for example:

```
val kotlinMascot = Image("https://blog.jetbrains.com/wp-content/uploads/2023/04/DSGN-16174-Blog-post-banner-and-promo-materials-for-post-about-Kotlin-mascot_3.png", embed = true).withWidth(400)
kotlinMascot
```

Out 8



Math formulas and equations

You can render mathematical formulas and equations using the LaTeX format, a typesetting system widely used in academia:

1. Add the `lib-ext` library that extends the functionality of the Jupyter kernel to your notebook:

```
%use lib-ext(0.11.0-398)
```

2. In the new cell, run your formula:

```
LATEX("c^2 = a^2 + b^2 - 2 a b \\cos\\alpha")
```

Out 5

$$c^2 = a^2 + b^2 - 2ab\cos\alpha$$

Data frames

With Kotlin Notebook, you can visualize structured data with data frames:

1. Add the Kotlin DataFrame library to your notebook:

```
%use dataframe
```

2. Create the data frame and run it in the new cell:

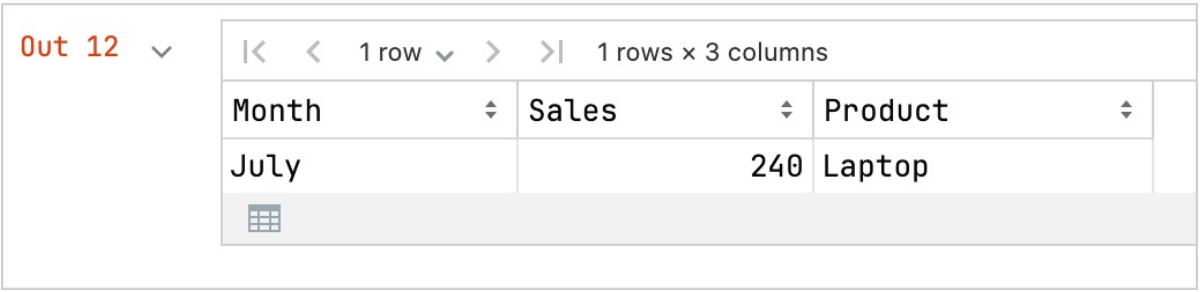
```
val months = listOf(
    "January", "February",
    "March", "April", "May",
    "June", "July", "August",
    "September", "October", "November",
    "December"
)

// Sales data for different products and months:
val salesLaptop = listOf(120, 130, 150, 180, 200, 220, 240, 230, 210, 190, 160, 140
)
val salesSmartphone = listOf(90, 100, 110, 130, 150, 170, 190, 180, 160, 140, 120,
100)
val salesTablet = listOf(60, 70, 80, 90, 100, 110, 120, 110, 100, 90, 80, 70)

// A data frame with columns for Month, Sales, and Product
val dfSales = dataFrameOf(
    "Month" to months + months + months,
    "Sales" to salesLaptop + salesSmartphone + salesTablet,
    "Product" to List(12) { "Laptop" } + List(12) { "Smartphone" } + List(12) { "T
ablet" },
```

```
)
```

The data frame uses the `dataFrameOf()` function and includes the number of products (laptops, smartphones, and tablets) sold in a 12-month period.

3. Explore the data in your frame, for example, by finding the product and month with the highest sales:

```
dfSales.maxBy("Sales")
```



4. You can also export your data frame as a CSV file:

```
// Export your data to CSV format
dfSales.writeCSV("sales-stats.csv")
```

Charts

You can create various charts directly in your Kotlin Notebook to visualize your data:

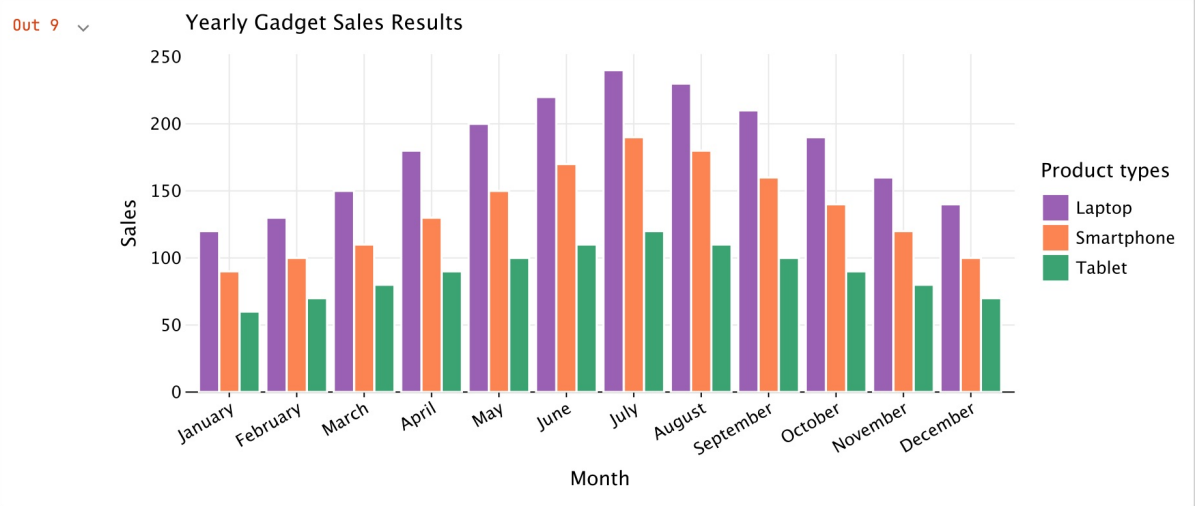
1. Add the Kandy plotting library to your notebook:

```
%use kandy
```

2. Use the same data frame and run the `plot()` function in the new cell:

```
val salesPlot = dfSales.groupBy { Product }.plot {
    bars {
        // Access the data frame's columns used for the X and Y axes
        x(Month)
        y(Sales)
        // Access the data frame's column used for categories and sets colors for
these categories
        fillColor(Product) {
            scale = categorical(
                "Laptop" to Color.PURPLE,
```

```
                "Smartphone" to Color.ORANGE,
                "Tablet" to Color.GREEN
            )
            legend.name = "Product types"
        }
    }
    // Customize the chart's appearance
    layout.size = 1000 to 450
    layout.title = "Yearly Gadget Sales Results"
}

salesPlot
```



3. You can also export your plot in the `.png`, `jpeg`, `.html`, or `.svg` format:

```
// Specify the output format for the plot file:
salesPlot.save("sales-chart.svg")
```

下一步做什么

- Visualize data using the DataFrame and Kandy libraries
- Learn more about rendering and displaying rich output in Kotlin Notebook
- Retrieve data from the CSV and JSON files
- Check out the list of recommended libraries

使用数据源

- 从文件中取数据
- 从 web 源与 API 取数据
- 链接数据库并从中取数据

从文件中取数据

Kotlin Notebook, coupled with the Kotlin DataFrame library, enables you to work with both non-structured and structured data. This combination offers the flexibility to transform non-structured data, such as data found in TXT files, into structured datasets.

For data transformations, you can use such methods as `add`, `split`, `convert`, and `parse`. Additionally, this toolset enables the retrieval and manipulation of data from various structured file formats, including CSV, JSON, XLS, XLSX, and Apache Arrow.

In this guide, you can learn how to retrieve, refine, and handle data through multiple examples.

Before you start

Kotlin Notebook relies on the Kotlin Notebook plugin, which is bundled and enabled in IntelliJ IDEA by default.

If the Kotlin Notebook features are not available, ensure the plugin is enabled. For more information, see Set up an environment.

Create a new Kotlin Notebook:

1. Select **File** | **New** | **Kotlin Notebook**.
2. In the Kotlin Notebook, import the Kotlin DataFrame library by running the following command:

```
%use dataframe
```

Retrieve data from a file

To retrieve data from a file in Kotlin Notebook:

1. Open your Kotlin Notebook file (`.ipynb`).
2. Import the Kotlin DataFrame library by adding `%use dataframe` in a code cell at the start of your notebook.

> Make sure to run the code cell with the `%use dataframe` line before you run any other code cells that rely on the Kotlin DataFrame library.
>
> 

3. Use the `.read()` function from the Kotlin DataFrame library to retrieve data. For example, to read a CSV file, use: `DataFrame.read("example.csv")`.

The `.read()` function automatically detects the input format based on the file extension and content. You can also add other arguments to customize the function, such as specifying the delimiter with `delimiter = ';'`.

> For a comprehensive overview of additional file formats and a variety of read functions, see the Kotlin DataFrame library documentation.

Display data

Once you have the data in your notebook, you can easily store it in a variable and access it by running the following in a code cell:

```
val dfJson = DataFrame.read("jsonFile.json")
dfJson
```

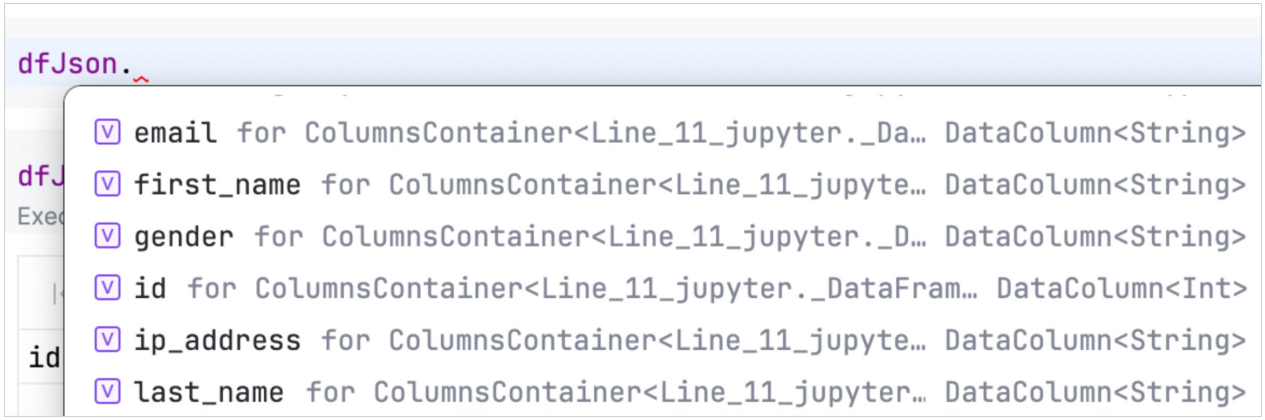
This code displays the data from the file of your choice, such as CSV, JSON, XLS, XLSX, or Apache Arrow.



To gain insights into the structure or schema of your data, apply the `.schema()` function on your DataFrame variable. For example, `dfJson.schema()` lists the type of each column in your JSON dataset.

```
dfJson.schema()
Executed at 2024.03.05 14:21:48 in 42ms

id: Int
first_name: String
last_name: String
email: String
gender: String
ip_address: String
```

You can also use the autocompletion feature in Kotlin Notebook to quickly access and manipulate the properties of your DataFrame. After loading your data, simply type the DataFrame variable followed by a dot to see a list of available columns and their types.



Refine data

Among the various operations available in the Kotlin DataFrame library for refining your dataset, key examples include grouping, filtering, updating, and adding new columns. These functions are essential for data analysis, allowing you to organize, clean, and transform your data effectively.

Let's look at an example where the data includes movie titles and their corresponding release year in the same cell. The goal is to refine this dataset for easier analysis:

1. Load your data into the notebook using the `.read()` function. This example involves reading data from a CSV file named `movies.csv` and creating a DataFrame called `movies`:

   ```
   val movies = DataFrame.read("movies.csv")
   ```

2. Extract the release year from the movie titles using regex and add it as a new column:

   ```
   val moviesWithYear = movies
       .add("year") {
           "\\d{4}".toRegex()
               .findAll(title)
               .lastOrNull()
               ?.value
               ?.toInt()
               ?: -1
       }
   ```

3. Modify the movie titles by removing the release year from each title. This cleans up the titles for consistency:

```
val moviesTitle = moviesWithYear
    .update("title") {
        "\\s*\\(\\d{4}\\)\\s*$".toRegex().replace(title, "")
    }
```

4. Use the `filter` method to focus on specific data. In this case, the dataset is filtered to focus on movies that were released after the year 1996:

```
val moviesNew = moviesWithYear.filter { year >= 1996 }
moviesNew
```

For comparison, here is the dataset before refinement:

movieId	title	genres
1	Toy Story (1995)	Adventure|Animation|Children|Comedy|Fantasy
2	Jumanji (1995)	Adventure|Children|Fantasy
3	Grumpier Old Men (1995)	Comedy|Romance
4	Waiting to Exhale (1995)	Comedy|Drama|Romance
5	Father of the Bride Part II (1995)	Comedy
6	Heat (1995)	Action|Crime|Thriller
7	Sabrina (1995)	Comedy|Romance
8	Tom and Huck (1995)	Adventure|Children
9	Sudden Death (1995)	Action
10	GoldenEye (1995)	Action|Adventure|Thriller

The refined dataset:

movieId	title	genres	year
61	Eye for an Eye	Drama|Thriller	1996
63	Don't Be a Menace to South Central While Drink…	Comedy|Crime	1996
64	Two if by Sea	Comedy|Romance	1996
65	Bio-Dome	Comedy	1996
66	Lawnmower Man 2: Beyond Cyberspace	Action|Sci-Fi|Thriller	1996
70	From Dusk Till Dawn	Action|Comedy|Horror|Thriller	1996
74	Bed of Roses	Drama|Romance	1996
75	Big Bully	Comedy|Drama	1996
79	Juror, The	Drama|Thriller	1996
86	White Squall	Action|Adventure|Drama	1996

This is a practical demonstration of how you can use the Kotlin DataFrame library's methods, like `add`, `update`, and `filter` to effectively refine and analyze data in Kotlin.

> For additional use cases and detailed examples, see Examples of Kotlin Dataframe.

Save DataFrame

After refining data in Kotlin Notebook using the Kotlin DataFrame library, you can easily export your processed data. You can utilize a variety of `.write()` functions for this purpose, which support saving in multiple formats, including CSV, JSON, XLS, XLSX, Apache Arrow, and even HTML tables. This can be particularly useful for sharing your findings, creating reports, or making your data available for further analysis.

Here's how you can filter a DataFrame, remove a column, save the refined data to a JSON file, and open an HTML table in your browser:

1. In Kotlin Notebook, use the `.read()` function to load a file named `movies.csv` into a DataFrame named `moviesDf`:

   ```
   val moviesDf = DataFrame.read("movies.csv")
   ```

2. Filter the DataFrame to only include movies that belong to the "Action" genre using the `.filter` method:

   ```
   val actionMoviesDf = moviesDf.filter { genres.equals("Action") }
   ```

3. Remove the `movieId` column from the DataFrame using `.remove`:

   ```
   val refinedMoviesDf = actionMoviesDf.remove { movieId }
   refinedMoviesDf
   ```

4. The Kotlin DataFrame library offers various write functions to save data in different formats. In this example, the `.writeJson()` function is used to save the modified `movies.csv` as a JSON file:

   ```
   refinedMoviesDf.writeJson("movies.json")
   ```

5. Use the `.toStandaloneHTML()` function to convert the DataFrame into a standalone HTML table and open it in your default web browser:

   ```
   refinedMoviesDf.toStandaloneHTML(DisplayConfiguration(rowsLimit = null)).openInBro
   wser()
   ```

下一步做什么

- Explore data visualization using the Kandy library
- Find additional information about data visualization in Data visualization in Kotlin Notebook

with Kandy

- For an extensive overview of tools and resources available for data science and analysis in Kotlin, see Kotlin and Java libraries for data analysis

从 web 源与 API 取数据

Kotlin Notebook provides a powerful platform for accessing and manipulating data from various web sources and APIs. It simplifies data extraction and analysis tasks by offering an iterative environment where every step can be visualized for clarity. This makes it particularly useful when exploring APIs you are not familiar with.

When used in conjunction with the Kotlin DataFrame library, Kotlin Notebook not only enables you to connect to and fetch JSON data from APIs but also assists in reshaping this data for comprehensive analysis and visualization.

> For Kotlin Notebook examples, see DataFrame examples on GitHub.
>
> 

开始之前

Kotlin Notebook relies on the Kotlin Notebook plugin, which is bundled and enabled in IntelliJ IDEA by default.

If the Kotlin Notebook features are not available, ensure the plugin is enabled. For more information, see Set up an environment.

Create a new Kotlin Notebook:

1. Select **File** | **New** | **Kotlin Notebook**.
2. In the Kotlin Notebook, import the Kotlin DataFrame library by running the following command:

   ```
   %use dataframe
   ```

Fetch data from an API

Fetching data from APIs using the Kotlin Notebook with the Kotlin DataFrame library is achieved through the `.read()` function, which is similar to retrieving data from files, such as CSV or JSON. However, when working with web-based sources, you might require additional formatting to transform the raw API data into a structured format.

Let's look at an example of fetching data from the YouTube Data API:

1. Open your Kotlin Notebook file (`.ipynb`).
2. Import the Kotlin DataFrame library, which is essential for data manipulation tasks. This is done by running the following command in a code cell:

   ```
   %use dataframe
   ```

3. Securely add your API key in a new code cell, which is necessary for authenticating requests to the YouTube Data API. You can obtain your API key from the credentials tab:

   ```
   val apiKey = "YOUR-API_KEY"
   ```

4. Create a load function that takes a path as a string and uses the DataFrame's `.read()` function to fetch data from the YouTube Data API:

   ```
   fun load(path: String): AnyRow = DataRow.read("https://www.googleapis.com/youtube/
   v3/$path&key=$apiKey")
   ```

5. Organize the fetched data into rows and handle the YouTube API's pagination through the `nextPageToken` . This ensures you gather data across multiple pages:

   ```
   fun load(path: String, maxPages: Int): AnyFrame {

       // Initializes a mutable list to store rows of data.
       val rows = mutableListOf<AnyRow>()

       // Sets the initial page path for data loading.
       var pagePath = path
       do {

           // Loads data from the current page path.
           val row = load(pagePath)
           // Adds the loaded data as a row to the list.
           rows.add(row)

           // Retrieves the token for the next page, if available.
           val next = row.getValueOrNull<String>("nextPageToken")
           // Updates the page path for the next iteration, including the new token.
           pagePath = path + "&pageToken=" + next

           // Continues loading pages until there's no next page.
       } while (next != null && rows.size < maxPages)

       // Concatenates and returns all loaded rows as a DataFrame.
       return rows.concat()
   }
   ```

6. Use the previously defined `load()` function to fetch data and create a DataFrame in a new code cell. This example fetches data, or in this case, videos related to Kotlin, with a maximum of 50 results per page, up to a maximum of 5 pages. The result is stored in the `df` variable:

```
val df = load("search?q=kotlin&maxResults=50&part=snippet", 5)
df
```

7. Finally, extract and concatenate items from the DataFrame:

```
val items = df.items.concat()
items
```

Clean and refine data

Cleaning and refining data are crucial steps in preparing your dataset for analysis. The Kotlin DataFrame library offers powerful functionalities for these tasks. Methods like `move`, `concat`, `select`, `parse`, and `join` are instrumental in organizing and transforming your data.

Let's explore an example where the data is already fetched using YouTube's data API. The goal is to clean and restructure the dataset to prepare for in-depth analysis:

1. You can start by reorganizing and cleaning your data. This involves moving certain columns under new headers and removing unnecessary ones for clarity:

```
val videos = items.dropNulls { id.videoId }
    .select { id.videoId named "id" and snippet }
    .distinct()
videos
```

2. Chunk IDs from the cleaned data and load corresponding video statistics. This involves breaking the data into smaller batches and fetching additional details:

```
val statPages = clean.id.chunked(50).map {
    val ids = it.joinToString("%2C")
    load("videos?part=statistics&id=$ids")
}
statPages
```

3. Concatenate the fetched statistics and select relevant columns:

```
val stats = statPages.items.concat().select { id and statistics.all() }.parse()
stats
```

4. Join the existing cleaned data with the newly fetched statistics. This merges two sets of data into a comprehensive DataFrame:

```
val joined = clean.join(stats)
joined
```

This example demonstrates how to clean, reorganize, and enhance your dataset using Kotlin DataFrame's various functions. Each step is designed to refine the data, making it more suitable for in-depth analysis.

在 Kotlin Notebook 中分析数据

After you've successfully fetched and cleaned and refined your data using functions from the Kotlin DataFrame library, the next step is to analyze this prepared dataset to extract meaningful insights.

Methods such as `groupBy` for categorizing data, `sum` and `maxBy` for summary statistics, and `sortBy` for ordering data are particularly useful. These tools allow you to perform complex data analysis tasks efficiently.

Let's look at an example, using `groupBy` to categorize videos by channel, `sum` to calculate total views per category, and `maxBy` to find the latest or most viewed video in each group:

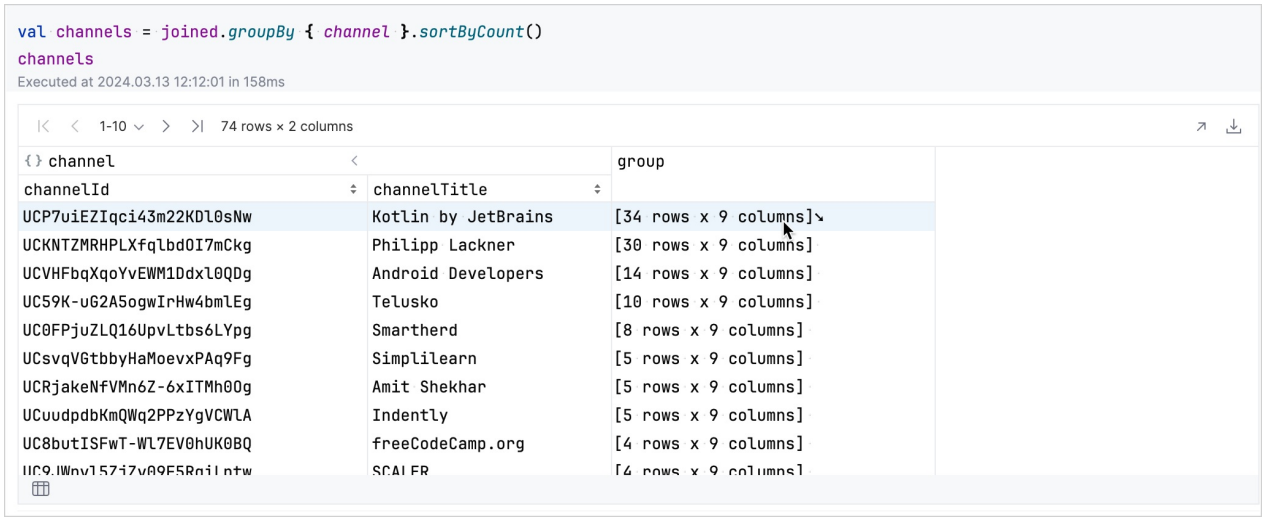
1. Simplify the access to specific columns by setting up references:

```
val view by column<Int>()
```

2. Use the `groupBy` method to group the data by the `channel` column and sort it.

```
val channels = joined.groupBy { channel }.sortByCount()
```

In the resulting table, you can interactively explore the data. Clicking on the `group` field of a row corresponding to a channel expands that row to reveal more details about that channel's videos.



You can click the table icon in the bottom left to return to the grouped dataset.



1. Use `aggregate`, `sum`, `maxBy`, and `flatten` to create a DataFrame summarizing each channel's total views and details of its latest or most viewed video:

```
val aggregated = channels.aggregate {
    viewCount.sum() into view

    val last = maxBy { publishedAt }
    last.title into "last title"
    last.publishedAt into "time"
    last.viewCount into "viewCount"
    // Sorts the DataFrame in descending order by view count and transform it into
 a flat structure.
}.sortByDesc(view).flatten()
aggregated
```

The results of the analysis:

```
val view by column<Int>() // Simplify column access by setting up references, making subsequent operations more intuitive.

val channels = joined.groupBy { channel }.sortByCount().aggregate
 { this: AggregateGroupedDsl<_DataFrameType33>  it: AggregateGroupedDsl<_DataFrameType33>
    viewCount.sum() into view

    val last = maxBy { publishedAt }
    last.title into "last title"
    last.publishedAt into "time"
    last.viewCount into "viewCount" ^aggregate
}.sortByDesc(view).flatten() // Sort the DataFrame in descending order based on the values in the view column and transform it into
 a flat structure.
channels
```

Executed at 2024.03.04 10:53:40 in 127ms

1-10 78 rows × 6 columns

channelId	channelTitle	view	last title	time
UCVEANauRpBFEuaQrmt66cBA	Ketchupy Kotlin	5330608	KRÓL MYCIA RĄCZEK FEAT. KOTLIN - MYCIE RĄCZEK…	2020-11·
UCxPD7bsocoAMq8Dj18kmGyQ	MoureDev by Brais Moure	2985221	KOTLIN: Curso ANDROID desde CERO para PRINCIP…	2020-03·
UC8butISFwT-Wl7EV0hUK0BQ	freeCodeCamp.org	2884525	Kotlin & Android Development Course: Buil…	2023-10·
UCVHFbqXqoYvEWM1Ddxl0QDg	Android Developers	2180328	5 years of Kotlin on Android #Shorts	2022-08·
UCKNTZMRHPLXfqlbdOI7mCkg	Philipp Lackner	1253153	Kotlin Multiplatform in a Nutshell	2024-01·
UC29ju8bIPH5as8OGnQzwJyA	Traversy Media	1089377	Build A Simple Android App With Kotlin	2020-11·
UCsBjURrPoezykLs9EqgamOA	Fireship	1083892	Kotlin in 100 Seconds	2021-11·
UC58_wzhvJta3hDSPvRLDAqg	Anuj Bhaiya	599068	Complete Android Development Roadmap for Begi…	2021-07·
UCwRXb5dUK4cvsHbx-rGzSgw	Derek Banas	540471	Kotlin Tutorial	2017-05·
UCP7uiEZIqci43m22KDl0sNw	Kotlin by JetBrains	509896	Advent of Code 2023 in Kotlin Day 12	2023-12·

For more advanced techniques, see the Kotlin DataFrame documentation.

下一步做什么

- Explore data visualization using the Kandy library
- Find additional information about data visualization in Data visualization in Kotlin Notebook with Kandy
- For an extensive overview of tools and resources available for data science and analysis in Kotlin, see Kotlin and Java libraries for data analysis

链接数据库并从中取数据

Kotlin Notebook offers capabilities for connecting to and retrieving data from various types of SQL databases, such as MariaDB, PostgreSQL, MySQL, and SQLite. Utilizing the Kotlin DataFrame library, Kotlin Notebook can establish connections to databases, execute SQL queries, and import the results for further operations.

For a detailed example, see the Notebook in the KotlinDataFrame SQL Examples GitHub repository.

开始之前

Kotlin Notebook relies on the Kotlin Notebook plugin, which is bundled and enabled in IntelliJ IDEA by default.

If the Kotlin Notebook features are not available, ensure the plugin is enabled. For more information, see Set up an environment.

Create a new Kotlin Notebook:

1. Select **File** | **New** | **Kotlin Notebook**.
2. Ensure you have access to an SQL database, such as MariaDB or MySQL.

连接数据库

You can connect to and interact with an SQL database using specific functions from the Kotlin DataFrame library. You can use `DatabaseConfiguration` to establish a connection to your database and `getSchemaForAllSqlTables()` to retrieve the schema of all tables within it.

Let's look at an example:

1. Open your Kotlin Notebook file (`.ipynb`).
2. Add a dependency for a JDBC (Java Database Connectivity) driver, and specify the JDBC driver version. This example uses MariaDB:

   ```
   USE {
       dependencies("org.mariadb.jdbc:mariadb-java-client:$version")
   }
   ```

3. Import the Kotlin DataFrame library, which is essential for data manipulation tasks, along with the necessary Java libraries for SQL connectivity and utility functions:

```
%use dataframe
import java.sql.DriverManager
import java.util.*
```

4. Use the `DatabaseConfiguration` class to define your database's connection parameters, including the URL, username, and password:

```
val URL = "YOUR_URL"
val USER_NAME = "YOUR_USERNAME"
val PASSWORD = "YOUR_PASSWORD"

val dbConfig = DatabaseConfiguration(URL, USER_NAME, PASSWORD)
```

5. Once connected, use the `getSchemaForAllSqlTables()` function to fetch and display the schema information for each table in the database:

```
val dataschemas = DataFrame.getSchemaForAllSqlTables(dbConfig)

dataschemas.forEach {
    println("---Yet another table schema---")
    println(it)
    println()
}
```

> For more information on connecting to SQL databases, see Read from SQL databases in the Kotlin DataFrame documentation.
>
> 

获取并操作数据

After establishing a connection to an SQL database, you can retrieve and manipulate data in Kotlin Notebook, utilizing the Kotlin DataFrame library. You can use the `readSqlTable()` function to retrieve data. To manipulate data, you can use methods, such as `filter`, `groupBy`, and `convert`.

Let's look at an example of connecting to an IMDB database and retrieving data about movies directed by Quentin Tarantino:

1. Use the `readSqlTable()` function to retrieve data from the "movies" table, setting `limit` to restrict the query to the first 100 records for efficiency:

```
val dfs = DataFrame.readSqlTable(dbConfig, tableName = "movies", limit = 100)
```

2. Use an SQL query to retrieve a specific dataset related to movies directed by Quentin Tarantino. This query selects movie details and combines genres for each movie:

```
val props = Properties()
props.setProperty("user", USER_NAME)
props.setProperty("password", PASSWORD)

val TARANTINO_FILMS_SQL_QUERY = """
    SELECT name, year, rank, GROUP_CONCAT(genre) as "genres"
    FROM movies JOIN movies_directors ON movie_id = movies.id
    JOIN directors ON directors.id=director_id LEFT JOIN movies_genres ON movies.i
d = movies_genres.movie_id
    WHERE directors.first_name = "Quentin" AND directors.last_name = "Tarantino"
    GROUP BY name, year, rank
    ORDER BY year
    """

// Retrieves a list of Quentin Tarantino's movies, including their name, year, ran
k, and a concatenated string of all genres.
// The results are grouped by name, year, rank, and sorted by year.

var dfTarantinoMovies: DataFrame<*>

DriverManager.getConnection(URL, props).use { connection ->
   connection.createStatement().use { st ->
      st.executeQuery(TARANTINO_FILMS_SQL_QUERY).use { rs ->
         val dfTarantinoFilmsSchema = DataFrame.getSchemaForResultSet(rs, connecti
on)
         dfTarantinoFilmsSchema.print()

         dfTarantinoMovies = DataFrame.readResultSet(rs, connection)
         dfTarantinoMovies
      }
   }
}
```

3. After fetching the Tarantino movies dataset, you can further manipulate and filter the data.

```
val df = dfTarantinoMovies
    // Replaces any missing values in the 'year' column with 0.
    .fillNA { year }.with { 0 }

    // Converts the 'year' column to integers.
    .convert { year }.toInt()

    // Filters the data to include only movies released after the year 2000.
    .filter { year > 2000 }
df
```

The resulting output is a DataFrame where missing values in the year column are replaced with 0 using the `fillNA` method. The year column is converted to integer values with the `convert` method, and the data is filtered to include only rows from the year 2000 onwards using the `filter` method.

在 Kotlin Notebook 中分析数据

After establishing a connection to an SQL database, you can use Kotlin Notebook for in-depth data analysis utilizing the Kotlin DataFrame library. This includes functions for grouping, sorting, and aggregating data, helping you to uncover and understand patterns within your data.

Let's dive into an example that involves analyzing actor data from a movie database, focusing on the most frequently occurring first names of actors:

1. Extract data from the "actors" table using the `readSqlTable()` function:

   ```
   val actorDf = DataFrame.readSqlTable(dbConfig, "actors", 10000)
   ```

2. Process the retrieved data to identify the top 20 most common actor first names. This analysis involves several DataFrame methods:

   ```
   val top20ActorNames = actorDf
       // Groups the data by the first_name column to organize it based on actor firs
   t names.
       .groupBy { first_name }

       // Counts the occurrences of each unique first name, providing a frequency dis
   tribution.
       .count()

       // Sorts the results in descending order of count to identify the most common
   names.
       .sortByDesc("count")

       // Selects the top 20 most frequent names for analysis.
       .take(20)
   top20ActorNames
   ```

下一步做什么

- Explore data visualization using the Kandy library
- Find additional information about data visualization in Data visualization in Kotlin Notebook with Kandy

- For an extensive overview of tools and resources available for data science and analysis in Kotlin, see Kotlin and Java libraries for data analysis

用 Kandy 实现 Kotlin Notebook 中数据可视化

Kotlin offers an all-in-one-place solution for powerful and flexible data visualization, providing an intuitive way to present and explore data before diving into complex models.

This tutorial demonstrates how to create different chart types in IntelliJ IDEA using Kotlin Notebook with the Kandy and Kotlin DataFrame libraries.

Before you start

Kotlin Notebook relies on the Kotlin Notebook plugin, which is bundled and enabled in IntelliJ IDEA by default.

If the Kotlin Notebook features are not available, ensure the plugin is enabled. For more information, see Set up an environment.

Create a new Kotlin Notebook:

1. Select **File** | **New** | **Kotlin Notebook**.
2. In your notebook, import the Kandy and Kotlin DataFrame libraries by running the following command:

```
%use kandy
%use dataframe
```

Create the DataFrame

Start by creating the DataFrame containing the records to visualize. This DataFrame stores simulated numbers of the monthly average temperature in three cities: Berlin, Madrid, and Caracas.

Use the `dataFrameOf()` function from the Kotlin DataFrame library to generate the DataFrame. Run the following code snippet in Kotlin Notebook:

```
// The months variable stores a list with the 12 months of the year
val months = listOf(
    "January", "February",
    "March", "April", "May",
    "June", "July", "August",
    "September", "October", "November",
    "December"
)
```

```
// The tempBerlin, tempMadrid, and tempCaracas variables store a list with temperature
 values for each month
val tempBerlin =
    listOf(-0.5, 0.0, 4.8, 9.0, 14.3, 17.5, 19.2, 18.9, 14.5, 9.7, 4.7, 1.0)
val tempMadrid =
    listOf(6.3, 7.9, 11.2, 12.9, 16.7, 21.1, 24.7, 24.2, 20.3, 15.4, 9.9, 6.6)
val tempCaracas =
    listOf(27.5, 28.9, 29.6, 30.9, 31.7, 35.1, 33.8, 32.2, 31.3, 29.4, 28.9, 27.6)

// The df variable stores a DataFrame of three columns, including records of months, t
emperature, and cities
val df = dataFrameOf(
    "Month" to months + months + months,
    "Temperature" to tempBerlin + tempMadrid + tempCaracas,
    "City" to List(12) { "Berlin" } + List(12) { "Madrid" } + List(12) { "Caracas" }
)
```

Explore the structure of the new DataFrame by looking into the first four rows:

```
df.head(4)
```

You can see that the DataFrame has three columns: Month, Temperature, and City. The first four rows of the DataFrame contain records of the temperature in Berlin from January to April:

4 rows 4 rows × 3 columns

Month	Temperature	City
January	-0.5	Berlin
February	0	Berlin
March	4.8	Berlin
April	9	Berlin

> There are different options to access a column's records that can help you increase type safety when working with the Kandy and Kotlin DataFrame libraries together. For more information, see Access APIs.

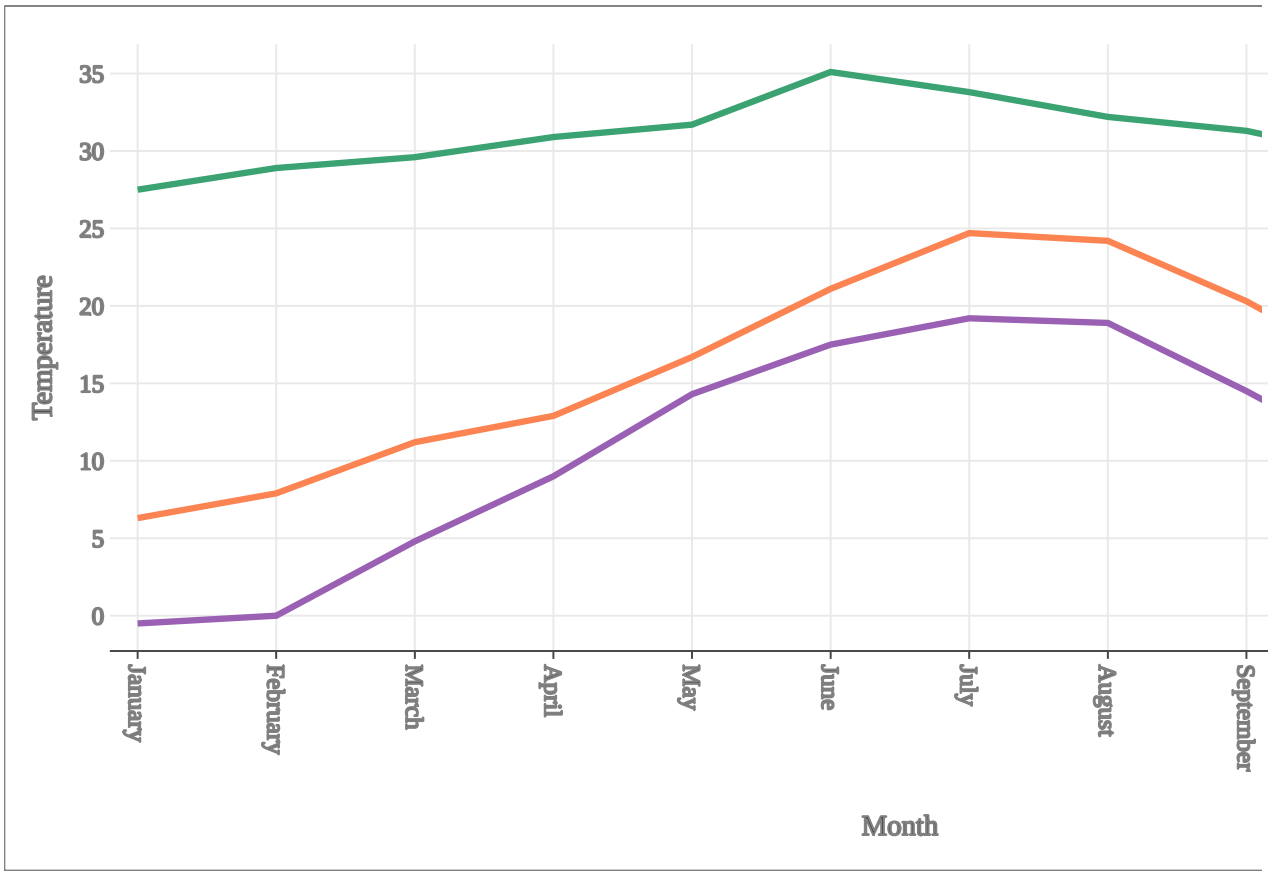
Create a line chart

Let's create a line chart in Kotlin Notebook using the `df` DataFrame from the previous section.

Use the `plot()` function from the Kandy library. Within the `plot()` function, specify the type of chart (in this case, it's `line`) and the values for the X and Y axes. You can customize colors and sizes:

```
df.plot {
    line {
        // Accesses the DataFrame's columns used for the X and Y axes
        x(Month)
        y(Temperature)
        // Accesses the DataFrame's column used for categories and sets colors for the
se categories
        color(City) {
            scale = categorical("Berlin" to Color.PURPLE, "Madrid" to Color.ORANGE, "C
aracas" to Color.GREEN)
        }
        // Customizes the line's size
        width = 1.5
    }
    // Customizes the chart's layout size
    layout.size = 1000 to 450
}
```

Here's the result:



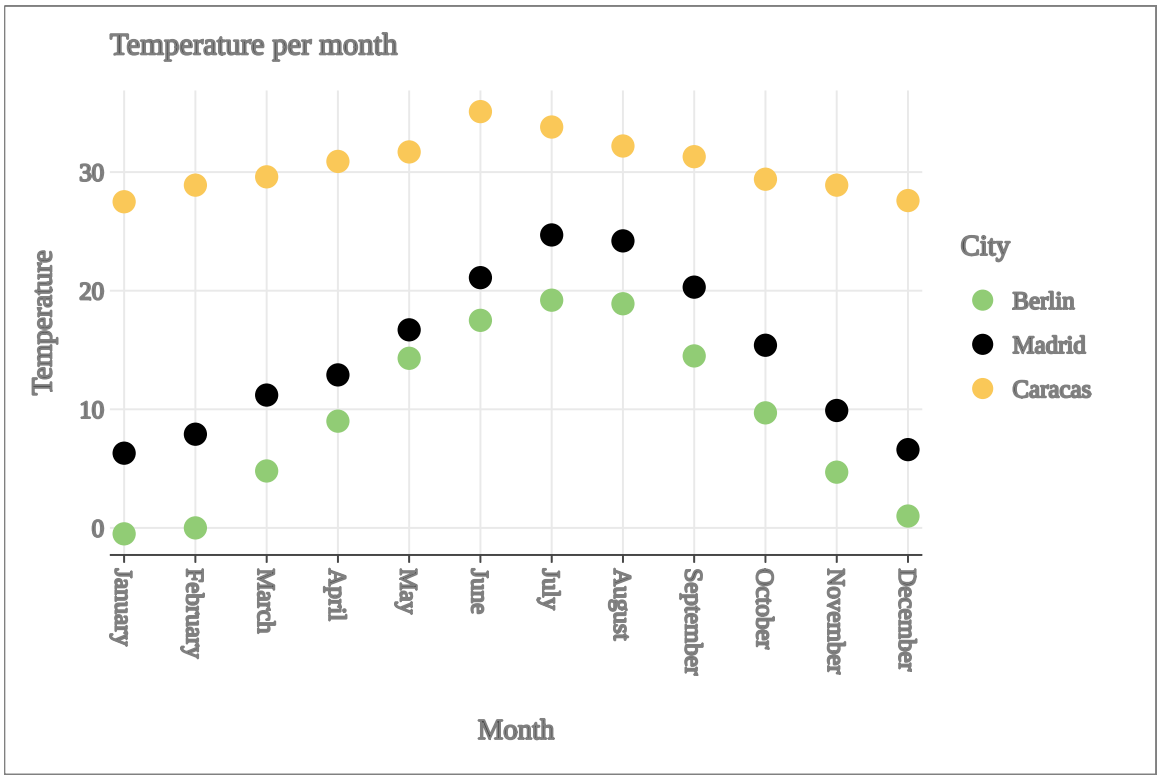
Create a points chart

Now, let's visualize the `df` DataFrame in a points (scatter) chart.

Within the `plot()` function, specify the `points` chart type. Add the X and Y axes' values and the categorical values from the `df` columns. You can also include a heading to your chart:

```
df.plot {
    points {
        // Accesses the DataFrame's columns used for the X and Y axes
        x(Month) { axis.name = "Month" }
        y(Temperature) { axis.name = "Temperature" }
        // Customizes the point's size
        size = 5.5
        // Accesses the DataFrame's column used for categories and sets colors for the
se categories
        color(City) {
            scale = categorical("Berlin" to Color.LIGHT_GREEN, "Madrid" to Color.BLACK
, "Caracas" to Color.YELLOW)
        }
    }
    // Adds a chart heading
    layout.title = "Temperature per month"
}
```

Here's the result:

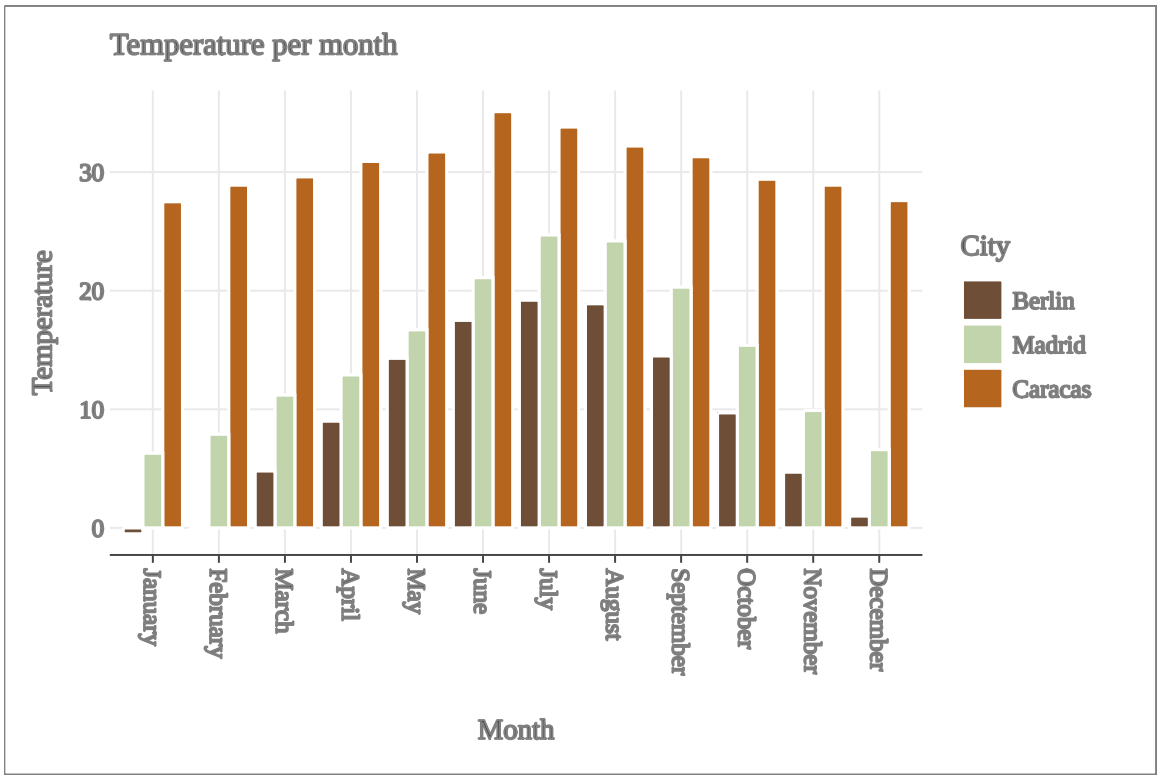


Create a bar chart

Finally, let's create a bar chart grouped by city using the same data as in the previous charts. For colors, you can also use hexadecimal codes:

```
// Groups by cities
df.groupBy { City }.plot {
    // Adds a chart heading
    layout.title = "Temperature per month"
    bars {
        // Accesses the DataFrame's columns used for the X and Y axes
        x(Month)
        y(Temperature)
        // Accesses the DataFrame's column used for categories and sets colors for the
se categories
        fillColor(City) {
            scale = categorical(
                "Berlin" to Color.hex("#6F4E37"),
                "Madrid" to Color.hex("#C2D4AB"),
                "Caracas" to Color.hex("#B5651D")
            )
        }
    }
}
```

Here's the result:



下一步做什么

- Explore more chart examples in the Kandy library documentation
- Explore more advanced plotting options in the Lets-Plot library documentation
- Find additional information about creating, exploring, and managing data frames in the Kotlin DataFrame library documentation
- Learn more about data visualization in Kotlin Notebook in this YouTube video

用于数据分析的 Kotlin 与 Java 库

From data collection to model building, Kotlin offers robust libraries facilitating different tasks in the data pipeline.

In addition to its own libraries, Kotlin is 100% interoperable with Java. This interoperability helps to leverage the entire ecosystem of tried-and-true Java libraries with excellent performance. With this perk, you can easily use either Kotlin or Java libraries when working on Kotlin data projects.

Kotlin libraries

Library	Purpose	Features
Kotlin DataFrame	• Data collection • Data cleaning and processing	• Operations for creating, sorting, and cleaning data frames, feature engineering, and more • Processing of structured data • Support for CSV, JSON, and other input formats • Reading from SQL databases • Connecting with different APIs to access data and increase type safety
Kandy	• Data exploration and visualization	• Powerful, readable, and typesafe DSL for plotting charts of various types • Open-source library written in Kotlin for the JVM • Support for Kotlin Notebook, Datalore, and Jupyter Notebook • Seamless integration with Kotlin DataFrame
KotlinDL	• Model building	• Deep learning API written in Kotlin and inspired by Keras • Training deep learning models from scratch or importing existing Keras and ONNX models for inference • Transferring learning for tailoring existing pre-trained models to your tasks • Support for the Android platform
Multik	• Data cleaning and processing • Model building	• Mathematical operations over multidimensional arrays (linear algebra, statistics, arithmetics, and other calculations) • Creating, copying, indexing, slicing, and other array operations • Kotlin-idiomatic library with benefits such as type and dimension safety and swappable computational engines, running on the JVM or as native code

Kotlin for Apache Spark	• Data collection • Data cleaning and processing • Data exploration and visualization • Model building	• Layer of compatibility between Apache Spark and Kotlin • Apache Spark data transformation operations in Kotlin-idiomatic code • Simple usage of Kotlin features, such as data classes and lambda expressions, in curly braces or method reference
Lets-Plot	• Data exploration and visualization	• Plotting statistical data written in Kotlin • Support for Kotlin Notebook, Datalore, and Jupyter with Kotlin Kernel • Compatible with the JVM, JS, and Python • Embedding charts in Compose Multiplatform applications
KMath	• Data cleaning and processing • Data exploration and visualization • Model building	• Modular library to work with mathematical abstractions in Kotlin Multiplatform (JVM, JS, Native, and Wasm) • APIs for algebraic structures, mathematical expressions, histograms, and streaming operations • Interchangeable wrappers over existing Java and Kotlin libraries, including ND4J, Apache Commons Math, and Multik • Inspired by Python's NumPy but with other additional features like type safety
kravis	• Data exploration and visualization	• Visualization of tabular data • Inspired by R's ggplot • Support for Jupyter with Kotlin Kernel

Java libraries

Since Kotlin provides first-class interoperability with Java, you can use Java libraries for data tasks in your Kotlin code. Here are some examples of such libraries:

Library	Purpose	Features
Tablesaw	• Data collection • Data cleaning and processing • Data exploration and visualization	• Tools for loading, cleaning, transforming, filtering, and summarizing data • Inspired by Plot.ly

CoreNLP	• Data cleaning and processing	• Natural language processing toolkit • Linguistic annotations for text, such as sentiment and quote attributions • Support for eight languages
Smile	• Data cleaning and processing • Data exploration and visualization • Model building	• Ready-made algorithms for machine learning and natural language processing • Linear algebra, graph, interpolation, and visualization tools • Provides functional Kotlin API, Scala API, Clojure API, and more
Smile-NLP-kt	• Data cleaning and processing	• Kotlin rewrite of the Scala implicits for the natural language processing part of Smile • Operations in the format of Kotlin extension functions and interfaces • Sentence breaking, stemming, bag of words, and other tasks
ND4J	• Data cleaning and processing • Model building	• Matrix mathematics library for the JVM • Over 500 mathematical, linear algebra, and deep learning operations
Apache Commons Math	• Data cleaning and processing • Model building	• Mathematics and statistics operations for Java • Correlations, distributions, linear algebra, geometry, and other operations • Machine learning models
NM Dev	• Data cleaning and processing • Model building	• Java math library of numerical algorithms • Object-oriented numerical methods • Linear algebra, optimization, statistics, calculus, and more operations
Apache OpenNLP	• Data cleaning and processing • Model building	• Machine-learning-based toolkit for the processing of natural language text • Tokenization, sentence segmentation, part-of-speech tagging, and other tasks • Built-in tools for data modeling and model validation
Charts	• Data exploration and visualization	• JavaFX library for scientific charts • Complex charts, such as logarithmic, heatmap, and force-directed graph
DeepLearning4J	• Model building	• Deep learning library for Java • Importing and retraining models (Pytorch, Tensorflow, Keras) • Deploying in JVM microservice

		environments, mobile devices, IoT, and Apache Spark
Timefold	• Model building	• Solver utility for optimization planning problems • Compatible with object-oriented and functional programming

平台

- JVM
  - Kotlin/JVM 入门
  - 与 Java 比较
  - 在 Kotlin 中调用 Java
  - 在 Java 中调用 Kotlin
  - Spring
    - 使用 Spring Boot 创建用到数据库的 RESTful web 服务
      - Spring Boot 与 Kotlin 入门
      - 创建使用 Kotlin 的 Spring Boot 项目
      - 向 Spring Boot 项目添加数据类
      - 为 Spring Boot 项目添加数据库支持
      - 使用 Spring Data CrudRepository 进行数据库访问
    - 构建一个使用 Spring AI 基于 Qdrant 中存储的文档来回答问题的 Kotlin 应用——教程
    - Spring 框架 Kotlin 文档↗
    - 使用 Spring Boot 与 Kotlin 构建 web 应用程序——教程↗
    - 使用 Kotlin 协程与 RSocket 创建聊天应用程序——教程↗
  - 在 JVM 平台中用 JUnit 测试代码——教程
  - 在项目中混用 Java 与 Kotlin——教程
  - 在 Kotlin 中使用 Java 记录类型
  - 从 Java 到 Kotlin 迁移指南
    - 字符串
    - 集合
    - 可空性
    - 标准输入
- 原生
  - Kotlin/Native 入门——在 IntelliJ IDEA 中
  - 定义文件
  - 与 C 语言互操作
    - 与 C 语言互操作性
    - 映射来自 C 语言的原始数据类型——教程
    - 映射来自 C 语言的结构与联合类型——教程
    - 映射来自 C 语言的函数指针——教程
    - 映射来自 C 语言的字符串——教程
    - Kotlin/Native 开发动态库——教程
    - 创建使用 C 语言互操作与 libcurl 的应用——教程

    - 与 Objective-C 互操作性
        - 与 Swift/Objective-C 互操作性
        - Kotlin/Native 开发 Apple framework——教程
    - Kotlin/Native 库
    - 平台库
    - 内存管理器
        - Kotlin/Native 内存管理
        - 与 ARC 集成
        - 迁移到新版内存管理器
    - 调试 Kotlin/Native
    - 符号化 iOS 崩溃报告
    - 参考与提示
        - 目标平台支持
        - 改进编译时间
        - 许可证文件
        - Kotlin/Native FAQ
- WebAssembly（Wasm）
    - 在 IntelliJ IDEA 中入门 Kotlin/Wasm
    - Kotlin/Wasm 与 WASI 入门
    - 调试 Kotlin/Wasm 代码
    - 与 JavaScript 的互操作
    - 故障排查
- JavaScript
    - 搭建 Kotlin/JS 项目
    - 运行 Kotlin/JS
    - 开发服务器与持续编译
    - 调试 Kotlin/JS 代码
    - 在 Kotlin/JS 平台中运行测试
    - Kotlin/JS 无用代码消除
    - Kotlin/JS IR 编译器
    - Kotlin 用于 JS 平台
        - 浏览器与 DOM API
        - 在 Kotlin 中使用 JavaScript 代码
        - 动态类型
        - 使用来自 npm 的依赖
        - 在 JavaScript 中使用 Kotlin 代码
        - JavaScript 模块
        - Kotlin/JS 反射
        - 类型安全的 HTML DSL
- 脚本

- Kotlin 自定义脚本入门——教程

JVM

- Kotlin/JVM 入门
- 与 Java 比较
- 在 Kotlin 中调用 Java
- 在 Java 中调用 Kotlin
- Spring
  - 使用 Spring Boot 创建用到数据库的 RESTful web 服务
    - Spring Boot 与 Kotlin 入门
    - 创建使用 Kotlin 的 Spring Boot 项目
    - 向 Spring Boot 项目添加数据类
    - 为 Spring Boot 项目添加数据库支持
    - 使用 Spring Data CrudRepository 进行数据库访问
  - 构建一个使用 Spring AI 基于 Qdrant 中存储的文档来回答问题的 Kotlin 应用——教程
  - Spring 框架 Kotlin 文档↗
  - 使用 Spring Boot 与 Kotlin 构建 web 应用程序——教程↗
  - 使用 Kotlin 协程与 RSocket 创建聊天应用程序——教程↗
- 在 JVM 平台中用 JUnit 测试代码——教程
- 在项目中混用 Java 与 Kotlin——教程
- 在 Kotlin 中使用 Java 记录类型
- 从 Java 到 Kotlin 迁移指南
  - 字符串
  - 集合
  - 可空性
  - 标准输入

Kotlin/JVM 入门

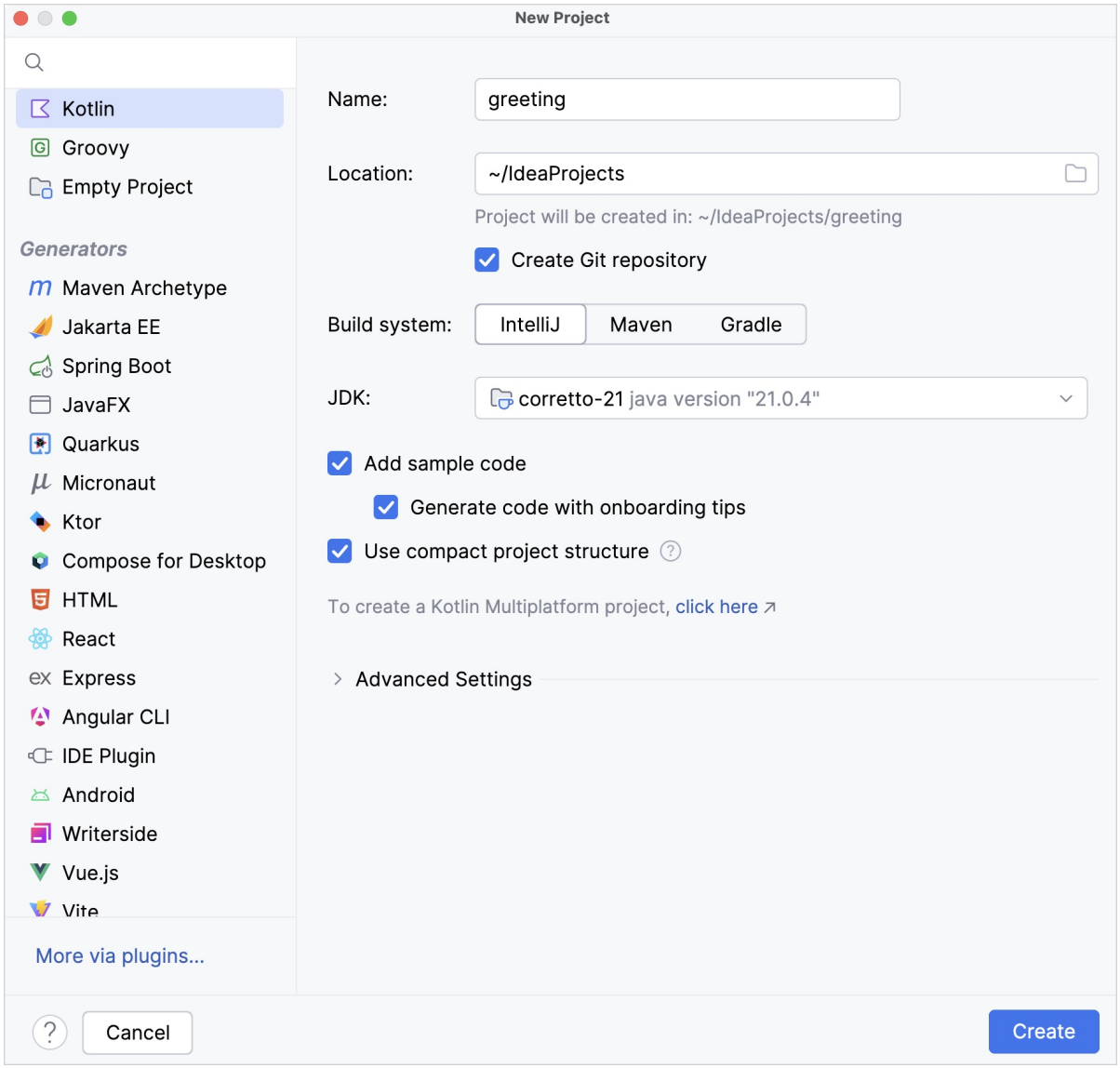
This tutorial demonstrates how to use IntelliJ IDEA for creating a console application.

To get started, first download and install the latest version of IntelliJ IDEA.

Create a project

1. In IntelliJ IDEA, select **File** | **New** | **Project**.
2. In the list on the left, select **Kotlin**.
3. Name the new project and change its location if necessary.

> Select the **Create Git repository** checkbox to place the new project under version control. You will be able to do it later at any time.
>
> 



4. Select the **IntelliJ** build system. It's a native builder that doesn't require downloading additional artifacts.

   If you want to create a more complex project that needs further configuration, select Maven or Gradle. For Gradle, choose a language for the build script: Kotlin or Groovy.

5. From the **JDK** list, select the JDK that you want to use in your project.

   - If the JDK is installed on your computer, but not defined in the IDE, select **Add JDK** and specify the path to the JDK home directory.
   - If you don't have the necessary JDK on your computer, select **Download JDK**.

6. Enable the **Add sample code** option to create a file with a sample `"Hello World!"` application.

   > You can also enable the **Generate code with onboarding tips** option to add some additional useful comments to your sample code.
   >
   > 

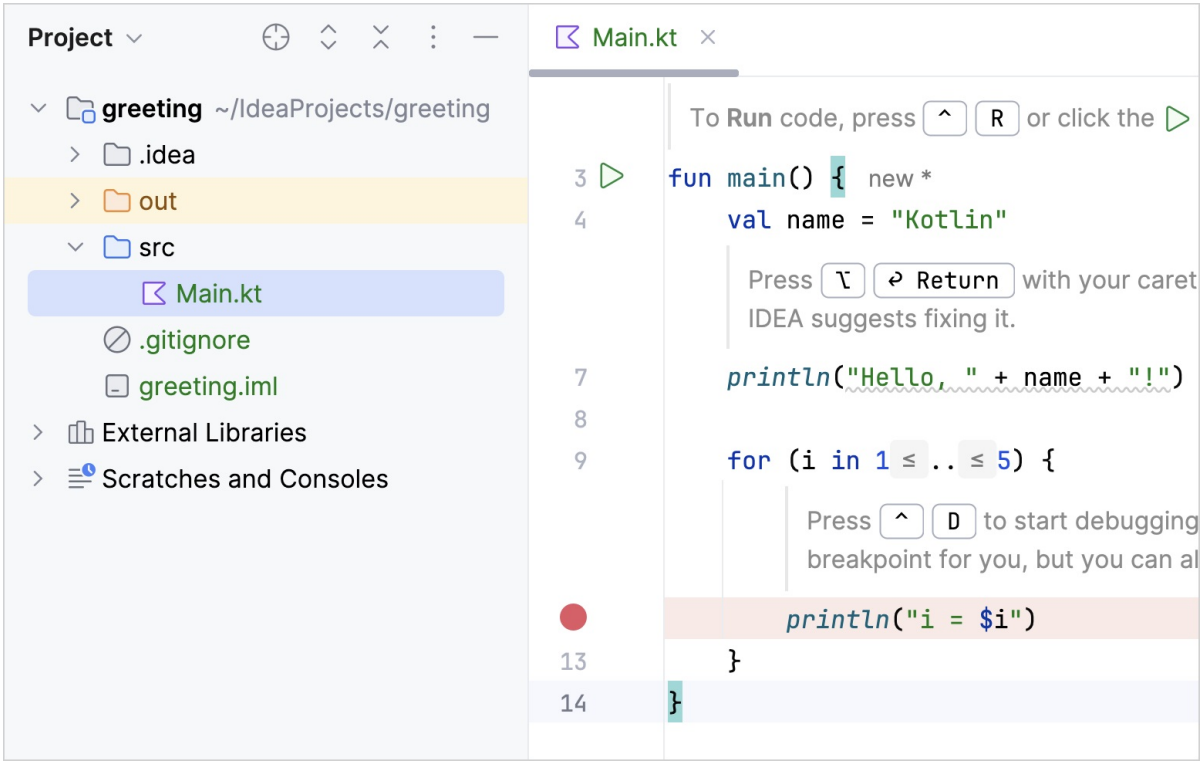
7. Click **Create**.

> If you chose the Gradle build system, you have in your project a build script file: `build.gradle(.kts)`. It includes the `kotlin("jvm")` plugin and dependencies required for your console application. Make sure that you use the latest version of the plugin:
>
> ```
> plugins {
>     kotlin("jvm") version "2.1.21"
>     application
> }
> ```
>
> 

Create an application

1. Open the `Main.kt` file in `src/main/kotlin`.

   The `src` directory contains Kotlin source files and resources. The `Main.kt` file contains sample code that will print `Hello, Kotlin!` as well as several lines with values of the cycle iterator.



2. Modify the code so that it requests your name and says `Hello` to you:

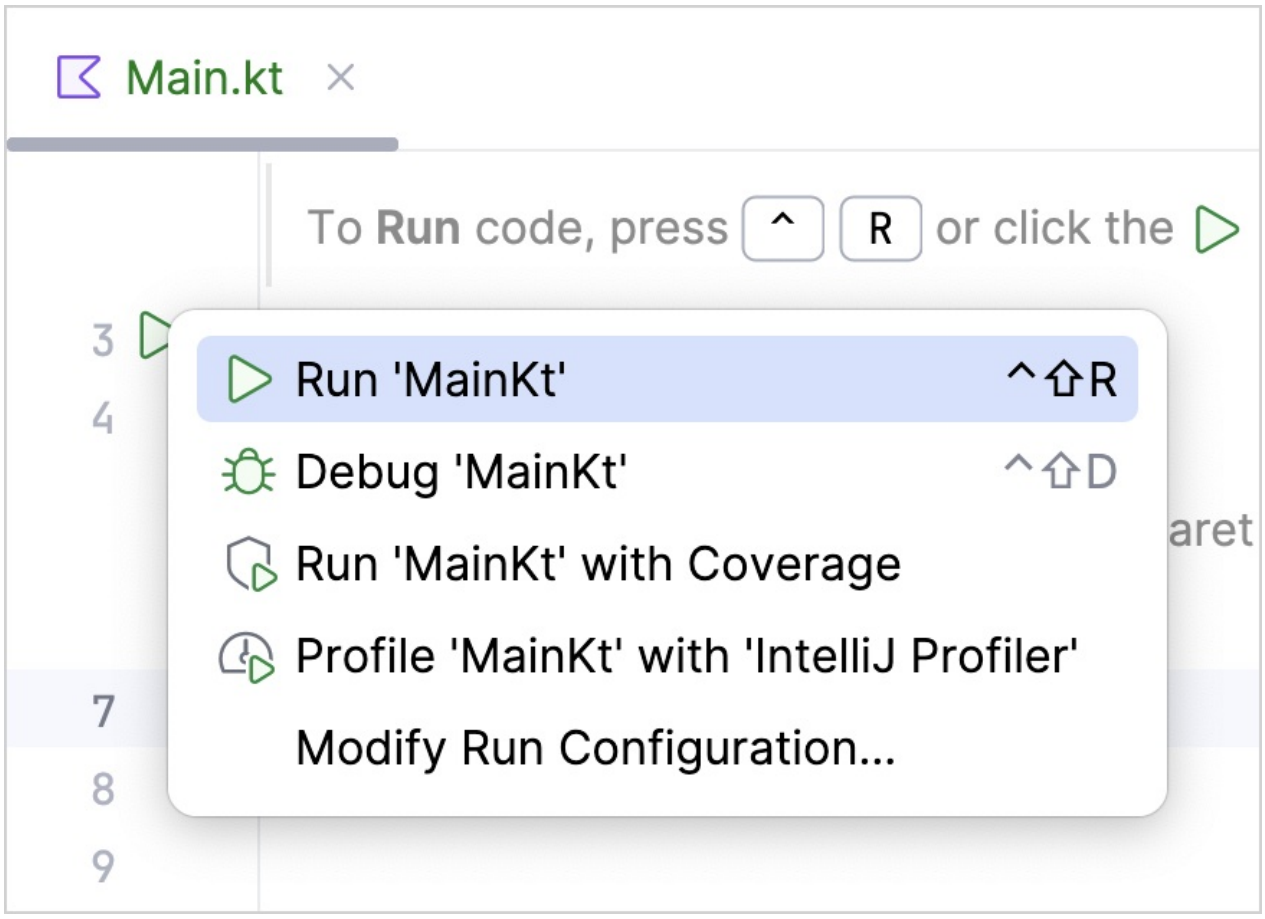
   - Create an input prompt and assign to the `name` variable the value returned by the `readln()` function.

- Let's use a string template instead of concatenation by adding a dollar sign `$` before the variable name directly in the text output like this – `$name` .

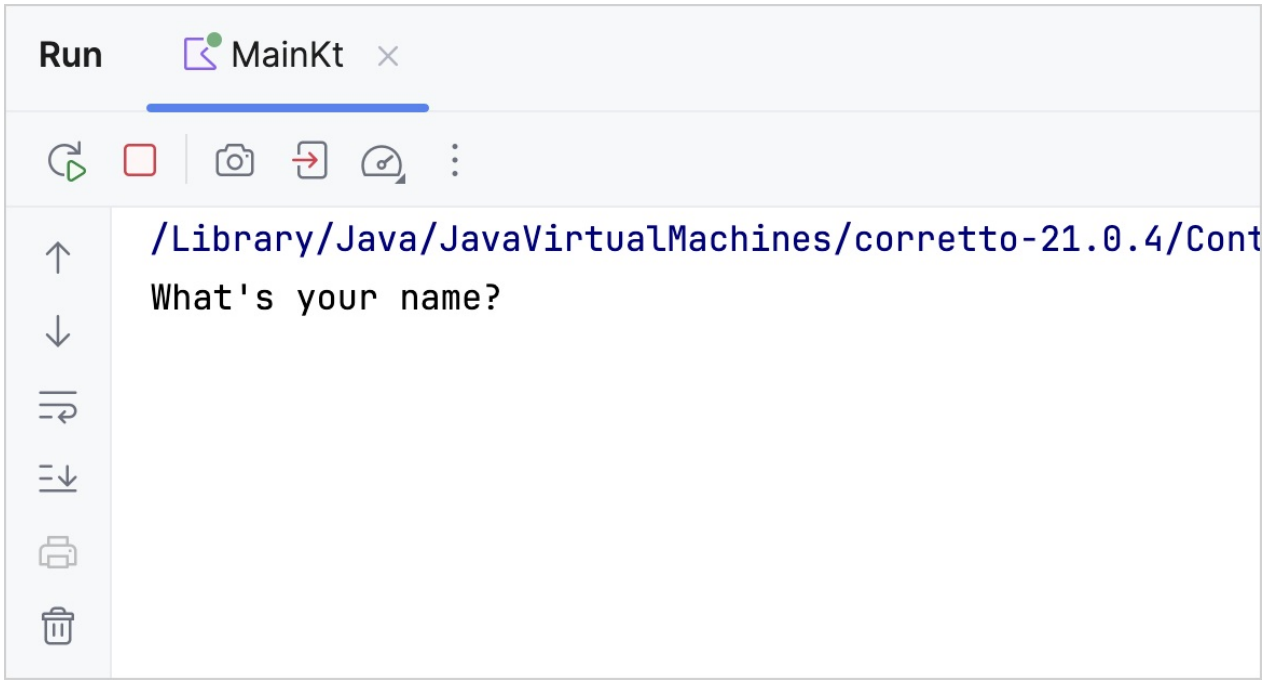
```
fun main() {
    println("What's your name?")
    val name = readln()
    println("Hello, $name!")

    // ...
}
```

Run the application

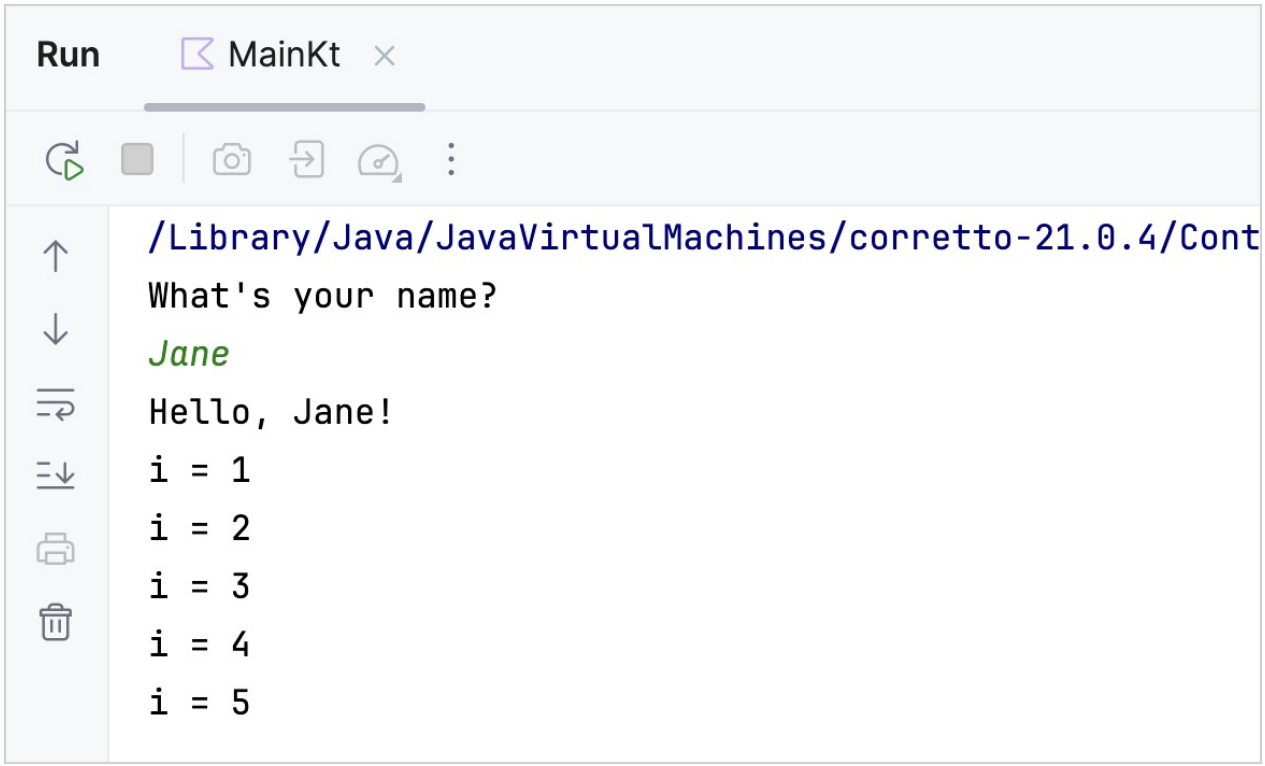
Now the application is ready to run. The easiest way to do this is to click the green **Run** icon in the gutter and select **Run 'MainKt'**.



You can see the result in the **Run** tool window.



Enter your name and accept the greetings from your application!



恭喜！ 你现在运行了第一个 Kotlin 应用程序。

下一步做什么？

Once you've created this application, you can start to dive deeper into Kotlin syntax:

- Add sample code from Kotlin examples
- Install the JetBrains Academy plugin for IDEA and complete exercises from the Kotlin Koans

course

与 Java 比较

Kotlin 解决了一些 Java 中的问题

Kotlin 通过以下措施修复了 Java 中一系列长期困扰我们的问题：

- 空引用由类型系统控制。
- 无原始类型
- Kotlin 中数组是不型变的
- 相对于 Java 的 SAM-转换，Kotlin 有更合适的函数类型
- 没有通配符的使用处型变
- Kotlin 没有受检异常
- Separate interfaces for read-only and mutable collections

Java 有而 Kotlin 没有的东西

- 受检异常
- 不是类的原生类型。字节码会尽可能试用原生类型，但不是显式可用的。
- 静态成员以伴生对象、 顶层函数、 扩展函数或者 @JvmStatic 取代。
- 通配符类型以声明处型变与类型投影取代。
- 三目操作符 `a ? b : c` —— 以 if 表达式取代。
- Records
- Pattern Matching
- package-private visibility modifier

Kotlin 有而 Java 没有的东西

- Lambda 表达式 + 内联函数 = 高性能自定义控制结构
- 扩展函数
- 空安全
- 智能类型转换 (**Java 16**: Pattern Matching for instanceof)
- 字符串模板 (**Java 21**: String Templates (Preview))
- 属性
- 主构造函数
- 一等公民的委托
- 变量与属性类型的类型推断 (**Java 10**: Local-Variable Type Inference)

- 单例
- 声明处型变 & 类型投影
- 区间表达式
- 操作符重载
- 伴生对象
- 数据类
- 协程
- Top-level functions
- Default arguments
- Named parameters
- Infix functions
- Expect and actual declarations
- Explicit API mode and better control of API surface

下一步做什么？

了解如何：

- 在 Java 与 Kotlin 中处理字符串的典型任务。
- Perform typical tasks with collections in Java and Kotlin.
- Handle nullability in Java and Kotlin.

在 Kotlin 中调用 Java

Kotlin 在设计时就考虑了 Java 互操作性。可以从 Kotlin 中自然地调用现存的 Java 代码， 并且在 Java 代码中也可以很顺利地调用 Kotlin 代码。 在本节中，我们会介绍从 Kotlin 中调用 Java 代码的一些细节。

几乎所有 Java 代码都可以使用而没有任何问题：

```
import java.util.*

fun demo(source: List<Int>) {
    val list = ArrayList<Int>()
    // “for”-循环用于 Java 集合:
    for (item in source) {
        list.add(item)
    }
    // 操作符约定同样有效:
    for (i in 0..source.size - 1) {
        list[i] = source[i] // 调用 get 和 set
    }
}
```

Getter 与 Setter

遵循 Java 约定的 getter 与 setter 的方法（名称以 `get` 开头的无参数方法及以 `set` 开头的单参数方法）在 Kotlin 中表示为属性。 Such properties are also called *synthetic properties*. `Boolean` 访问器方法（其中 getter 的名称以 `is` 开头而 setter 的名称以 `set` 开头）会表示为与 getter 方法具有相同名称的属性。

```
import java.util.Calendar

fun calendarDemo() {
    val calendar = Calendar.getInstance()
    if (calendar.firstDayOfWeek == Calendar.SUNDAY) { // 调用 getFirstDayOfWeek()
        calendar.firstDayOfWeek = Calendar.MONDAY // 调用ll setFirstDayOfWeek()
    }
    if (!calendar.isLenient) { // 调用 isLenient()
        calendar.isLenient = true // 调用 setLenient()
    }
}
```

`calendar.firstDayOfWeek` above is an example of a synthetic property.

请注意，如果 Java 类只有一个 setter，它在 Kotlin 中不会作为属性可见，因为 Kotlin 不支持只写（set-only）属性。

Java synthetic property references

> This feature is Experimental. It may be dropped or changed at any time. We recommend that you use it only for evaluation purposes.
>
> 

Starting from Kotlin 1.8.20, you can create references to Java synthetic properties. Consider the following Java code:

```
public class Person {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }
}
```

Kotlin has always allowed you to write `person.age`, where `age` is a synthetic property. Now, you can also create references to `Person::age` and `person::age`. The same applies for `name`, as well.

```
val persons = listOf(Person("Jack", 11), Person("Sofie", 12), Person("Peter", 11))
    persons
         // Call a reference to Java synthetic property:
        .sortedBy(Person::age)
         // Call Java getter via the Kotlin property syntax:
        .forEach { person -> println(person.name) }
```

How to enable Java synthetic property references

To enable this feature, set the `-language-version 2.1` compiler option. In a Gradle project, you can do so by adding the following to your `build.gradle(.kts)`:

> 【Kotlin】

```
tasks
    .withType<org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTask<*>>()
    .configureEach {
        compilerOptions
            .languageVersion
            .set(
                org.jetbrains.kotlin.gradle.dsl.KotlinVersion.KOTLIN_2_1
            )
    }
```

> 【Groovy】

```
tasks
    .withType(org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTask.class)
    .configureEach {
        compilerOptions.languageVersion
            = org.jetbrains.kotlin.gradle.dsl.KotlinVersion.KOTLIN_2_1
}
```

> Prior to Kotlin 1.9.0, to enable this feature you had to set the `-language-version 1.9` compiler option.
>
> 

返回 void 的方法

如果一个 Java 方法返回 `void`，那么从 Kotlin 调用时中返回 `Unit`。 万一有人使用其返回值，它将由 Kotlin 编译器在调用处赋值， 因为该值本身是预先知道的（是 `Unit`）。

将 Kotlin 中是关键字的 Java 标识符进行转义

一些 Kotlin 关键字在 Java 中是有效标识符： `in`、 `object`、 `is` 等。 如果一个 Java 库使用了 Kotlin 关键字作为方法，你仍然可以通过反引号（`）字符转义它来调用该方法：

```
foo.`is`(bar)
```

空安全与平台类型

Java 中的任何引用都可能是 `null`，这使得 Kotlin 对来自 Java 的对象要求严格空安全是不现实的。 Java 声明的类型在 Kotlin 中会以特殊方式对待并称为平台类型。对这种类型的空检测会放宽， 因此它们的安全保证与在 Java 中相同（更多请参见下文）。

考虑以下示例：

```
val list = ArrayList<String>() // 非空（构造函数结果）
list.add("Item")
val size = list.size // 非空（原生 int）
val item = list[0] // 推断为平台类型（普通 Java 对象）
```

当调用平台类型变量的方法时，Kotlin 不会在编译时报告可空性错误， 但在运行时调用可能会失败，因为空指针异常或者 Kotlin 生成的阻止空值传播的断言：

```
item.substring(1) // 允许，如果 item == null 会抛出异常
```

平台类型是不可标示的，意味着不能在语言中明确地写下它们。 当把一个平台值赋值给一个 Kotlin 变量时，可以依赖类型推断（该变量会具有推断出的的平台类型， 如上例中 `item` 所具有的类型），或者可以选择所期望的类型（可空或非空类型均可）：

```
val nullable: String? = item // 允许，没有问题
val notNull: String = item // 允许，运行时可能失败
```

如果选择非空类型，编译器会在赋值时触发一个断言。这防止 Kotlin 的非空变量保存空值。当将平台值传递给期待非空值的 Kotlin 函数时或其他情况，也会触发断言。 总的来说，编译器尽力阻止空值通过程序向远传播（尽管鉴于泛型的原因，有时这不可能完全消除）。

平台类型表示法

如上所述，平台类型不能在程序中显式表述，因此在语言中没有相应语法。 然而，编译器和 IDE 有时需要（例如在错误信息中、参数信息中等）显示他们， 因此有一个助记符来表示它们：

- `T!` 表示“ `T` 或者 `T?` ”，
- `(Mutable)Collection<T>!` 表示“可以可变或不可变、可空或不可空的 `T` 的 Java 集合”，
- `Array<(out) T>!` 表示“可空或者不可空的 `T` （或 `T` 的子类型）的 Java 数组”

可空性注解

具有可空性注解的 Java 类型并不表示为平台类型，而是表示为实际可空或非空的 Kotlin 类型。编译器支持多种可空性注解，包括：

- JetBrains（`org.jetbrains.annotations` 包中的 `@Nullable` 和 `@NotNull`）
- JSpecify（`org.jspecify.annotations`）
- Android（`com.android.annotations` 和 `android.support.annotations`）
- JSR-305（`javax.annotation`，详见下文）
- FindBugs（`edu.umd.cs.findbugs.annotations`）
- Eclipse（`org.eclipse.jdt.annotation`）
- Lombok（`lombok.NonNull`）
- RxJava 3（`io.reactivex.rxjava3.annotations`）

You can specify whether the compiler reports a nullability mismatch based on the information from specific types of nullability annotations. Use the compiler option `-Xnullability-annotations=@<package-name>:<report-level>`. In the argument, specify the fully qualified nullability annotations package and one of these report levels:

- `ignore` to ignore nullability mismatches
- `warn` to report warnings
- `strict` to report errors.

See the full list of supported nullability annotations in the Kotlin compiler source code.

Annotating type arguments and type parameters

You can annotate the type arguments and type parameters of generic types to provide nullability information for them as well.

> All examples in the section use JetBrains nullability annotations from the `org.jetbrains.annotations` package.

Type arguments

考虑这些 Java 声明的注解：

```
@NotNull
Set<@NotNull String> toSet(@NotNull Collection<@NotNull String> elements) { …… }
```

在 Kotlin 中其结果是以下签名：

```
fun toSet(elements: (Mutable)Collection<String>) : (Mutable)Set<String> { …… }
```

When the `@NotNull` annotation is missing from a type argument, you get a platform type instead:

```
fun toSet(elements: (Mutable)Collection<String!>) : (Mutable)Set<String!> { …… }
```

Kotlin also takes into account nullability annotations on type arguments of base classes and interfaces. For example, there are two Java classes with the signatures provided below:

```
public class Base<T> {}
```

```
public class Derived extends Base<@Nullable String> {}
```

In the Kotlin code, passing the instance of `Derived` where the `Base<String>` is assumed produces the warning.

```
fun takeBaseOfNotNullStrings(x: Base<String>) {}

fun main() {
    takeBaseOfNotNullStrings(Derived()) // warning: nullability mismatch
}
```

The upper bound of `Derived` is set to `Base<String?>`, which is different from `Base<String>`.

Learn more about Java generics in Kotlin.

Type parameters

By default, the nullability of plain type parameters in both Kotlin and Java is undefined. In Java, you can specify it using nullability annotations. Let's annotate the type parameter of the `Base` class:

```
public class Base<@NotNull T> {}
```

When inheriting from `Base`, Kotlin expects a non-nullable type argument or type parameter. Thus, the following Kotlin code produces a warning:

```
class Derived<K> : Base<K> {} // warning: K has undefined nullability
```

You can fix it by specifying the upper bound `K : Any`.

Kotlin also supports nullability annotations on the bounds of Java type parameters. Let's add bounds to `Base`:

```
public class BaseWithBound<T extends @NotNull Number> {}
```

Kotlin translates this just as follows:

```
class BaseWithBound<T : Number> {}
```

So passing nullable type as a type argument or type parameter produces a warning.

Annotating type arguments and type parameters works with the Java 8 target or higher. The feature requires that the nullability annotations support the `TYPE_USE` target (`org.jetbrains.annotations` supports this in version 15 and above).

> If a nullability annotation supports other targets that are applicable to a type in addition to the `TYPE_USE` target, `TYPE_USE` takes priority. For example, if `@Nullable` has both `TYPE_USE` and `METHOD` targets, the Java method signature `@Nullable String[] f()` becomes `fun f(): Array<String?>!` in Kotlin.



JSR-305 支持

已支持 JSR-305 中定义的 `@Nonnull` 注解来表示 Java 类型的可空性。

如果 `@Nonnull(when = ...)` 值为 `When.ALWAYS`，那么该注解类型会被视为非空；`When.MAYBE` 与 `When.NEVER` 表示可空类型；而 `When.UNKNOWN` 强制类型为平台类型。

可针对 JSR-305 注解编译库，但不需要为库的消费者将注解构件（如 `jsr305.jar` ）指定为编译依赖。Kotlin 编译器可以从库中读取 JSR-305 注解，并不需要该注解出现在类路径中。

也支持自定义可空限定符（KEEP-79）（见下文）。

类型限定符别称

如果一个注解类型同时标注有 `@TypeQualifierNickname` 与 JSR-305 `@Nonnull`（或者它的其他别称，如 `@CheckForNull` ），那么该注解类型自身将用于 检索精确的可空性，且具有与该可空性注解相同的含义：

```
@TypeQualifierNickname
@Nonnull(when = When.ALWAYS)
@Retention(RetentionPolicy.RUNTIME)
public @interface MyNonnull {
}

@TypeQualifierNickname
```

```
@CheckForNull // 另一个类型限定符别称的别称
@Retention(RetentionPolicy.RUNTIME)
public @interface MyNullable {
}

interface A {
    @MyNullable String foo(@MyNonnull String x);
    // 在 Kotlin（严格模式）中: `fun foo(x: String): String?`

    String bar(List<@MyNonnull String> x);
    // 在 Kotlin（严格模式）中: `fun bar(x: List<String>!): String!`
}
```

类型限定符默认值

`@TypeQualifierDefault` 引入应用时在所标注元素的作用域内定义默认可空性的注解。

这些注解类型应自身同时标注有 `@Nonnull`（或其别称）与 `@TypeQualifierDefault(...)` 注解，后者带有一到多个 `ElementType` 值：

- `ElementType.METHOD` 用于方法的返回值
- `ElementType.PARAMETER` 用于值参数
- `ElementType.FIELD` 用于字段
- `ElementType.TYPE_USE` 适用于任何类型，包括类型参数、类型参数的上界与通配符类型

当类型并未标注可空性注解时使用默认可空性，并且该默认值是由最内层标注有带有与所用类型相匹配的 `ElementType` 的类型限定符默认注解的元素确定。

```
@Nonnull
@TypeQualifierDefault({ElementType.METHOD, ElementType.PARAMETER})
public @interface NonNullApi {
}

@Nonnull(when = When.MAYBE)
@TypeQualifierDefault({ElementType.METHOD, ElementType.PARAMETER, ElementType.TYPE_USE
})
public @interface NullableApi {
}

@NullableApi
interface A {
    String foo(String x); // fun foo(x: String?): String?

    @NotNullApi // 覆盖来自接口的默认值
    String bar(String x, @Nullable String y); // fun bar(x: String, y: String?): String


    // 由于 `@NullableApi` 具有 `TYPE_USE` 元素类型，
    // 因此认为 List<String> 类型参数是可空的：
```

```
    String baz(List<String> x); // fun baz(List<String?>?): String?

    // “x”参数仍然是平台类型，因为有显式
    // UNKNOWN 标记的可空性注解：
    String qux(@Nonnull(when = When.UNKNOWN) String x); // fun baz(x: String!): String?

}
```

> 本例中的类型只在启用了严格模式时出现；否则仍是平台类型。 参见 `@UnderMigration` 注解与编译器配置两节。



也支持包级的默认可空性：

```
// 文件: test/package-info.java
@NonNullApi // 默认将“test”包中所有类型声明为不可空
package test;
```

@UnderMigration 注解

库的维护者可以使用 `@UnderMigration` 注解（在单独的构件 `kotlin-annotations-jvm` 中提供）来定义可为空性类型限定符的迁移状态。

`@UnderMigration(status = ...)` 中的状态值指定了编译器如何处理 Kotlin 中注解类型的不当用法（例如，使用 `@MyNullable` 标注的类型值作为非空值）：

- `MigrationStatus.STRICT` 使注解像任何纯可空性注解一样工作，即对不当用法报错并影响注解声明内的类型在 Kotlin 中的呈现
- `MigrationStatus.WARN`：不当用法报为警告而不是错误； 但注解声明内的类型仍是平台类型
- `MigrationStatus.IGNORE` 则使编译器完全忽略可空性注解

库的维护者还可以将 `@UnderMigration` 状态添加到类型限定符别称与类型限定符默认值：

```
@Nonnull(when = When.ALWAYS)
@TypeQualifierDefault({ElementType.METHOD, ElementType.PARAMETER})
@UnderMigration(status = MigrationStatus.WARN)
public @interface NonNullApi {
}

// 类中的类型是非空的，但是只报警告
// 因为 `@NonNullApi` 标注了 `@UnderMigration(status = MigrationStatus.WARN)`
@NonNullApi
public class Test {}
```

> 可空性注解的迁移状态并不会从其类型限定符别称继承，而是适用于默认类型限定符的用法。
>
> 

如果默认类型限定符使用类型限定符别称，并且它们都标注有 `@UnderMigration`，那么使用默认类型限定符的状态。

编译器配置

可以通过添加带有以下选项的 `-Xjsr305` 编译器标志来配置 JSR-305 检测：

- `-Xjsr305={strict|warn|ignore}` 设置非 `@UnderMigration` 注解的行为。 自定义的可空性限定符，尤其是 `@TypeQualifierDefault` 已经在很多知名库中流传，而用户更新到包含 JSR-305 支持的 Kotlin 版本时可能需要平滑迁移。自 Kotlin 1.1.60 起，这一标志只影响非 `@UnderMigration` 注解。
- `-Xjsr305=under-migration:{strict|warn|ignore}` 覆盖 `@UnderMigration` 注解的行为。 用户可能对库的迁移状态有不同的看法： 他们可能希望在官方迁移状态为 `WARN` 时报错误，反之亦然，他们可能希望推迟错误报告直到他们完成迁移。
- `-Xjsr305=@<fq.name>:{strict|warn|ignore}` 覆盖单个注解的行为，其中 `<fq.name>` 是该注解的完整限定类名。对于不同的注解可以多次出现。这对于管理特定库的迁移状态非常有用。

其中 `strict`、 `warn` 与 `ignore` 值的含义与 `MigrationStatus` 中的相同， 并且只有 `strict` 模式会影响注解声明中的类型在 Kotlin 中的呈现。

> 注意：内置的 JSR-305 注解 `@Nonnull`、 `@Nullable` 与 `@CheckForNull` 总是启用并影响所注解的声明在 Kotlin 中呈现，无论如何配置编译器的 `-Xjsr305` 标志。
>
> 

例如，将 `-Xjsr305=ignore -Xjsr305=under-migration:ignore -Xjsr305=@org.library.MyNullable:warn` 添加到编译器参数中，会使编译器对由 `@org.library.MyNullable` 标注的不当用法生成警告，而忽略所有其他 JSR-305 注解。

默认行为等同于 `-Xjsr305=warn`。 `strict` 值应认为是实验性的（以后可能添加更多检测）。

已映射类型

Kotlin 特殊处理一部分 Java 类型。这样的类型不是“按原样”从 Java 加载，而是 映射 到相应的 Kotlin 类型。 映射只发生在编译期间，运行时表示保持不变。 Java 的原生类型映射到相应的 Kotlin 类型（请记住平台类型）：

Java 类型	Kotlin 类型
`byte`	`kotlin.Byte`
`short`	`kotlin.Short`
`int`	`kotlin.Int`
`long`	`kotlin.Long`
`char`	`kotlin.Char`
`float`	`kotlin.Float`
`double`	`kotlin.Double`
`boolean`	`kotlin.Boolean`

一些非原生的内置类型也会作映射：

Java 类型	Kotlin 类型
`java.lang.Object`	`kotlin.Any!`
`java.lang.Cloneable`	`kotlin.Cloneable!`
`java.lang.Comparable`	`kotlin.Comparable!`
`java.lang.Enum`	`kotlin.Enum!`
`java.lang.annotation.Annotation`	`kotlin.Annotation!`
`java.lang.CharSequence`	`kotlin.CharSequence!`
`java.lang.String`	`kotlin.String!`
`java.lang.Number`	`kotlin.Number!`
`java.lang.Throwable`	`kotlin.Throwable!`

Java 的装箱原始类型映射到可空的 Kotlin 类型：

Java type	Kotlin type
`java.lang.Byte`	`kotlin.Byte?`
`java.lang.Short`	`kotlin.Short?`
`java.lang.Integer`	`kotlin.Int?`
`java.lang.Long`	`kotlin.Long?`
`java.lang.Character`	`kotlin.Char?`
`java.lang.Float`	`kotlin.Float?`
`java.lang.Double`	`kotlin.Double?`
`java.lang.Boolean`	`kotlin.Boolean?`

请注意，用作类型参数的装箱原生类型映射到平台类型：例如，`List<java.lang.Integer>` 在 Kotlin 中会成为 `List<Int!>`。

集合类型在 Kotlin 中可以是只读的或可变的，因此 Java 集合类型作如下映射：（下表中的所有 Kotlin 类型都驻留在 `kotlin.collections` 包中）：

Java 类型	Kotlin 只读类型	Kotlin 可变类型	加载的平
`Iterator<T>`	`Iterator<T>`	`MutableIterator<T>`	`(Mutable)Iter`
`Iterable<T>`	`Iterable<T>`	`MutableIterable<T>`	`(Mutable)Iter`
`Collection<T>`	`Collection<T>`	`MutableCollection<T>`	`(Mutable)Coll`
`Set<T>`	`Set<T>`	`MutableSet<T>`	`(Mutable)Set<`
`List<T>`	`List<T>`	`MutableList<T>`	`(Mutable)List`
`ListIterator<T>`	`ListIterator<T>`	`MutableListIterator<T>`	`(Mutable)List`
`Map<K, V>`	`Map<K, V>`	`MutableMap<K, V>`	`(Mutable)Map<`
`Map.Entry<K, V>`	`Map.Entry<K, V>`	`MutableMap.MutableEntry<K,V>`	`(Mutable)Map.(Mutable)Entry`

Java 的数组按下文所述映射：

Java 类型	Kotlin 类型
`int[]`	`kotlin.IntArray!`
`String[]`	`kotlin.Array<(out) String!>!`

> 这些 Java 类型的静态成员不能在相应 Kotlin 类型的伴生对象中直接访问。要调用它们，请使用 Java 类型的完整限定名，例如 `java.lang.Integer.toHexString(foo)`。

Kotlin 中的 Java 泛型

Kotlin 的泛型与 Java 有点不同（参见泛型）。 当将 Java 类型导入 Kotlin 时，有以下转换：

- Java 的通配符转换成类型投影：
  - `Foo<? extends Bar>` 转换成 `Foo<out Bar!>!`
  - `Foo<? super Bar>` 转换成 `Foo<in Bar!>!`
- Java的原始类型转换成星投影:
  - `List` 转换成 `List<*>!`，也就是 `List<out Any?>!`

和 Java 一样，Kotlin 在运行时不保留泛型：对象不携带传递到他们构造器中的那些类型参数的实际类型。 例如 `ArrayList<Integer>()` 和 `ArrayList<Character>()` 是不能区分的。 这使得执行 `is`-检测不可能照顾到泛型。 Kotlin 只允许 `is`-检测星投影的泛型类型：

```
if (a is List<Int>) // 错误：无法检测它是否真的是一个 Int 列表
// but
if (a is List<*>) // OK：不保证列表的内容
```

Java 数组

与 Java 不同，Kotlin 中的数组是不型变的。这意味着 Kotlin 不允许把一个 `Array<String>` 赋值给一个 `Array<Any>`，从而避免了可能的运行时故障。Kotlin 也禁止把一个子类的数组当做超类的数组传递给 Kotlin 的方法，但是对于 Java 方法，这是允许的（通过 `Array<(out) String>!` 这种形式的平台类型）。

Java 平台上，数组会使用原生数据类型以避免装箱/拆箱操作的开销。由于 Kotlin 隐藏了这些实现细节，因此需要一个变通方法来与 Java 代码进行交互。对于每种原生类型的数组都有一个特化的类（`IntArray`、`DoubleArray`、`CharArray` 等等）来处理这种情况。它们与 `Array` 类无关，并且会编译成 Java 原生类型数组以获得最佳性能。

假设有一个接受 int 数组索引的 Java 方法：

```
public class JavaArrayExample {
    public void removeIndices(int[] indices) {
        // 在此编码……
    }
}
```

在 Kotlin 中可以这样传递一个原生类型的数组：

```
val javaObj = JavaArrayExample()
val array = intArrayOf(0, 1, 2, 3)
javaObj.removeIndices(array)  // 将 int[] 传给方法
```

当编译为 JVM 字节代码时，编译器会优化对数组的访问，这样就不会引入任何开销：

```
val array = arrayOf(1, 2, 3, 4)
array[1] = array[1] * 2 // 不会实际生成对 get() 和 set() 的调用
for (x in array) { // 不会创建迭代器
    print(x)
}
```

即使当使用索引定位时，也不会引入任何开销：

```
for (i in array.indices) {// 不会创建迭代器
    array[i] += 2
}
```

最后，`in`-检测也没有额外开销：

```
if (i in array.indices) { // 同 (i >= 0 && i < array.size)
    print(array[i])
}
```

Java 可变参数

Java 类有时声明一个具有可变数量参数（varargs）的方法来使用索引：

```
public class JavaArrayExample {

    public void removeIndicesVarArg(int... indices) {
        // 在此编码……
    }
}
```

在这种情况下，你需要使用展开运算符 `*` 来传递 `IntArray`：

```
val javaObj = JavaArrayExample()
val array = intArrayOf(0, 1, 2, 3)
javaObj.removeIndicesVarArg(*array)
```

操作符

由于 Java 无法标记用于运算符语法的方法，Kotlin 允许具有正确名称和签名的任何 Java 方法作为运算符重载和其他约定（`invoke()` 等）使用。 不允许使用中缀调用语法调用 Java 方法。

受检异常

在 Kotlin 中，所有异常都是非受检的，这意味着编译器不会强迫你捕获其中的任何一个。 因此，当你调用一个声明受检异常的 Java 方法时，Kotlin 不会强迫你做任何事情：

```
fun render(list: List<*>, to: Appendable) {
    for (item in list) {
        to.append(item.toString()) // Java 会要求我们在这里捕获 IOException
    }
}
```

对象方法

当 Java 类型导入到 Kotlin 中时，类型 `java.lang.Object` 的所有引用都成了 `Any`。 而因为 `Any` 不是平台指定的，它只声明了 `toString()`、 `hashCode()` 和 `equals()` 作为其成员，所以为了能用到 `java.lang.Object` 的其他成员，Kotlin 要用到扩展函数。

wait()/notify()

类型 `Any` 的引用没有提供 `wait()` 与 `notify()` 方法。通常不鼓励使用它们，而建议使用 `java.util.concurrent`。 如果确实需要调用这两个方法的话，那么可以将引用转换为 `java.lang.Object`：

```
(foo as java.lang.Object).wait()
```

getClass()

要取得对象的 Java 类，请在类引用上使用 `java` 扩展属性：

```
val fooClass = foo::class.java
```

上面的代码使用了绑定的类引用。你也可以使用 `javaClass` 扩展属性：

```
val fooClass = foo.javaClass
```

clone()

要覆盖 `clone()`，需要继承 `kotlin.Cloneable`：

```
class Example : Cloneable {
    override fun clone(): Any { …… }
}
```

不要忘记《Effective Java》第三版 的第 13 条: 谨慎地改写*clone*。

finalize()

要覆盖 `finalize()`，所有你需要做的就是简单地声明它，而不需要 `override` 关键字：

```
class C {
    protected fun finalize() {
```

```
            // 终止化逻辑
        }
    }
```

根据 Java 的规则，`finalize()` 不能是 `private` 的。

从 Java 类继承

在 kotlin 中，类的超类中最多只能有一个 Java 类（以及按你所需的多个 Java 接口）。

访问静态成员

Java 类的静态成员会形成该类的“伴生对象”。无法将这样的“伴生对象”作为值来传递，但可以显式访问其成员，例如：

```
if (Character.isLetter(a)) { …… }
```

要访问已映射到 Kotlin 类型的 Java 类型的静态成员，请使用 Java 类型的完整限定名：`java.lang.Integer.bitCount(foo)`。

Java 反射

Java 反射适用于 Kotlin 类，反之亦然。如上所述，你可以使用 `instance::class.java`，`ClassName::class.java` 或者 `instance.javaClass` 通过 `java.lang.Class` 来进入 Java 反射。Do not use `ClassName.javaClass` for this purpose because it refers to `ClassName`'s companion object class, which is the same as `ClassName.Companion::class.java` and not `ClassName::class.java`.

For each primitive type, there are two different Java classes, and Kotlin provides ways to get both. For example, `Int::class.java` will return the class instance representing the primitive type itself, corresponding to `Integer.TYPE` in Java. To get the class of the corresponding wrapper type, use `Int::class.javaObjectType`, which is equivalent of Java's `Integer.class`.

其他支持的情况包括为一个 Kotlin 属性获取一个 Java 的 getter/setter 方法或者幕后字段、为一个 Java 字段获取一个 `KProperty`、为一个 `KFunction` 获取一个 Java 方法或者构造函数，反之亦然。

SAM 转换

Kotlin 支持 Java 以及 Kotlin 接口的 SAM 转换。 对于 Java 来说，这意味着 Kotlin 函数字面值可以被自动的转换成只有一个非默认方法的 Java 接口的实现，只要这个方法的参数类型能够与这个 Kotlin 函数的参数类型相匹配。

你可以这样创建 SAM 接口的实例：

```
val runnable = Runnable { println("This runs in a runnable") }
```

……以及在方法调用中：

```
val executor = ThreadPoolExecutor()
// Java 签名: void execute(Runnable command)
executor.execute { println("This runs in a thread pool") }
```

如果 Java 类有多个接受函数式接口的方法，那么可以通过使用将 lambda 表达式转换为特定的 SAM 类型的适配器函数来选择需要调用的方法。这些适配器函数也会按需由编译器生成：

```
executor.execute(Runnable { println("This runs in a thread pool") })
```

> SAM 转换只适用于接口，而不适用于抽象类，即使这些抽象类也只有一个抽象方法。
>
> 

在 Kotlin 中使用 JNI

要声明一个在本地（C 或 C++）代码中实现的函数，你需要使用 `external` 修饰符来标记它：

```
external fun foo(x: Int): Double
```

其余的过程与 Java 中的工作方式完全相同。

You can also mark property getters and setters as `external`:

```
var myProperty: String
    external get
    external set
```

Behind the scenes, this will create two functions `getMyProperty` and `setMyProperty`, both marked as `external`.

Using Lombok-generated declarations in Kotlin

You can use Java's Lombok-generated declarations in Kotlin code. If you need to generate and use these declarations in the same mixed Java/Kotlin module, you can learn how to do this on the Lombok compiler plugin's page. If you call such declarations from another module, then you don't need to use this plugin to compile that module.

在 Java 中调用 Kotlin

Java 可以轻松调用 Kotlin 代码。 例如，可以在 Java 方法中无缝创建与操作 Kotlin 类的实例。 然而，在将 Kotlin 代码集成到 Java 中时， 需要注意 Java 与 Kotlin 之间的一些差异。 在本页，我们会描述定制 Kotlin 代码与其 Java 客户端的互操作的方法。

属性

Kotlin 属性会编译成以下 Java 元素：

- 一个 getter 方法，名称通过加前缀 `get` 算出
- 一个 setter 方法，名称通过加前缀 `set` 算出（只适用于 `var` 属性）
- 一个私有字段，与属性名称相同（仅适用于具有幕后字段的属性）

例如， `var firstName: String` 编译成以下 Java 声明：

```
private String firstName;

public String getFirstName() {
    return firstName;
}

public void setFirstName(String firstName) {
    this.firstName = firstName;
}
```

如果属性的名称以 `is` 开头，则使用不同的名称映射规则：getter 的名称与属性名称相同，并且 setter 的名称是通过将 `is` 替换为 `set` 获得。 例如，对于属性 `isOpen` ，其 getter 会称做 `isOpen()` ，而其 setter 会称做 `setOpen()` 。 这一规则适用于任何类型的属性，并不仅限于 `Boolean` 。

包级函数

在 `org.example` 包内的 `app.kt` 文件中声明的所有的函数和属性，包括扩展函数， 都编译成一个名为 `org.example.AppKt` 的 Java 类的静态方法。

```
// app.kt
package org.example

class Util
```

```
fun getTime() { /*……*/ }
```

```
// Java
new org.example.Util();
org.example.AppKt.getTime();
```

可以使用 `@JvmName` 注解自定义生成的 Java 类的类名：

```
@file:JvmName("DemoUtils")

package org.example

class Util

fun getTime() { /*……*/ }
```

```
// Java
new org.example.Util();
org.example.DemoUtils.getTime();
```

如果多个文件中生成了相同的 Java 类名（包名相同并且类名相同或者有相同的 `@JvmName` 注解）通常是错误的。 然而，编译器能够生成一个单一的 Java 外观类，它具有指定的名称且包含来自所有文件中具有该名称的所有声明。 要启用生成这样的外观，请在所有相关文件中使用 `@JvmMultifileClass` 注解。

```
// oldutils.kt
@file:JvmName("Utils")
@file:JvmMultifileClass

package org.example

fun getTime() { /*……*/ }
```

```
// newutils.kt
@file:JvmName("Utils")
@file:JvmMultifileClass

package org.example

fun getDate() { /*……*/ }
```

```
// Java
org.example.Utils.getTime();
```

```
org.example.Utils.getDate();
```

实例字段

如果需要在 Java 中将 Kotlin 属性作为字段暴露，那就使用 `@JvmField` 注解对其标注。 该字段将具有与底层属性相同的可见性。如果一个属性具备这几点，那么可以用 `@JvmField` 注解该属性：

- 有幕后字段（backing field）
- 非私有
- 没有 `open`、 `override` 或者 `const` 修饰符
- 不是被委托的属性

```
class User(id: String) {
    @JvmField val ID = id
}
```

```
// Java
class JavaClient {
    public String getID(User user) {
        return user.ID;
    }
}
```

延迟初始化的属性（在Java中）也会暴露为字段。 该字段的可见性与 `lateinit` 属性的 setter 相同。

静态字段

在具名对象或伴生对象中声明的 Kotlin 属性会在该具名对象或包含伴生对象的类中具有静态幕后字段。

通常这些字段是私有的，但可以通过以下方式之一暴露出来：

- `@JvmField` 注解
- `lateinit` 修饰符
- `const` 修饰符

使用 `@JvmField` 标注这样的属性使其成为与属性本身具有相同可见性的静态字段。

```
class Key(val value: Int) {
```

```
    companion object {
        @JvmField
        val COMPARATOR: Comparator<Key> = compareBy<Key> { it.value }
    }
}
```

```
// Java
Key.COMPARATOR.compare(key1, key2);
// Key 类中的 public static final 字段
```

在具名对象或者伴生对象中的一个延迟初始化的属性具有与属性 setter 相同可见性的静态幕后字段。

```
object Singleton {
    lateinit var provider: Provider
}
```

```
// Java
Singleton.provider = new Provider();
// 在 Singleton 类中的 public static 非-final 字段
```

（在类中以及在顶层）以 `const` 声明的属性在 Java 中会成为静态字段：

```
// 文件 example.kt

object Obj {
    const val CONST = 1
}

class C {
    companion object {
        const val VERSION = 9
    }
}

const val MAX = 239
```

在 Java 中：

```
int constant = Obj.CONST;
int max = ExampleKt.MAX;
int version = C.VERSION;
```

静态方法

如上所述，Kotlin 将包级函数表示为静态方法。 Kotlin 还可以为具名对象或伴生对象中定义的函数生成静态方法，如果你将这些函数标注为 `@JvmStatic` 的话。 如果你使用该注解，编译器既会在相应对象的类中生成静态方法， 也会在对象自身中生成实例方法。例如：

```
class C {
    companion object {
        @JvmStatic fun callStatic() {}
        fun callNonStatic() {}
    }
}
```

现在， `callStatic()` 在 Java 中是静态的，而 `callNonStatic()` 不是：

```
C.callStatic(); // 没问题
C.callNonStatic(); // 错误：不是一个静态方法
C.Companion.callStatic(); // 保留实例方法
C.Companion.callNonStatic(); // 唯一的工作方式
```

对于具名对象也同样：

```
object Obj {
    @JvmStatic fun callStatic() {}
    fun callNonStatic() {}
}
```

在 Java 中：

```
Obj.callStatic(); // 没问题
Obj.callNonStatic(); // 错误
Obj.INSTANCE.callNonStatic(); // 没问题，通过单例实例调用
Obj.INSTANCE.callStatic(); // 也没问题
```

自 Kotlin 1.3 起， `@JvmStatic` 也适用于在接口的伴生对象中定义的函数。 这类函数会编译为接口中的静态方法。请注意，接口中的静态方法是 Java 1.8 中引入的， 因此请确保使用相应的编译目标。

```
interface ChatBot {
    companion object {
        @JvmStatic fun greet(username: String) {
            println("Hello, $username")
```

```
        }
    }
}
```

`@JvmStatic` 注解也可以应用于对象或伴生对象的属性， 使其 getter 和 setter 方法在该对象或包含该伴生对象的类中是静态成员。

接口中的默认方法

> 默认方法仅适用于面向 JVM 1.8 及更高版本。
>
> 

自 JDK 1.8 起，Java 中的接口可以包含默认方法。 To make all non-abstract members of Kotlin interfaces default for the Java classes implementing them, compile the Kotlin code with the `-Xjvm-default=all` compiler option.

这是一个带有默认方法的 Kotlin 接口的一个示例：

```
// compile with -Xjvm-default=all

interface Robot {
    fun move() { println("~walking~") }  // will be default in the Java interface
    fun speak(): Unit
}
```

默认实现对于实现该接口的 Java 类都可用。

```
//Java 实现
public class C3PO implements Robot {
    // 来自 Robot 的 move() 实现隐式可用
    @Override
    public void speak() {
        System.out.println("I beg your pardon, sir");
    }
}
```

```
C3PO c3po = new C3PO();
c3po.move(); // 来自 Robot 接口的默认实现
c3po.speak();
```

接口的实现者可以覆盖默认方法。

```
//Java
```

```
public class BB8 implements Robot {
    //自己实现默认方法
    @Override
    public void move() {
        System.out.println("~rolling~");
    }

    @Override
    public void speak() {
        System.out.println("Beep-beep");
    }
}
```

> Prior to Kotlin 1.4, to generate default methods, you could use the `@JvmDefault` annotation on these methods. Compiling with `-Xjvm-default=all` in 1.4+ generally works as if you annotated all non-abstract methods of interfaces with `@JvmDefault` and compiled with `-Xjvm-default=enable`. However, there are cases when their behavior differs. Detailed information about the changes in default methods generation in Kotlin 1.4 is provided in this post on the Kotlin blog.
>
> 

Compatibility modes for default methods

If there are clients that use your Kotlin interfaces compiled without the `-Xjvm-default=all` option, then they may be binary-incompatible with the code compiled with this option. To avoid breaking the compatibility with such clients, use the `-Xjvm-default=all` mode and mark interfaces with the `@JvmDefaultWithCompatibility` annotation. This allows you to add this annotation to all interfaces in the public API once, and you won't need to use any annotations for new non-public code.

> Starting from Kotlin 1.6.20, you can compile modules in the default mode (the `-Xjvm-default=disable` compiler option) against modules compiled with the `-Xjvm-default=all` or `-Xjvm-default=all-compatibility` modes.
>
> 

Learn more about compatibility modes:

disable

Default behavior. Do not generate JVM default methods and prohibit `@JvmDefault` annotation usage.

all

Generate JVM default methods for all interface declarations with bodies in the module. Do not generate `DefaultImpls` stubs for interface declarations with bodies, which are generated by default in the `disable` mode.

If interface inherits a method with body from an interface compiled in the `disable` mode and doesn't override it, then a `DefaultImpls` stub will be generated for it.

Breaks binary compatibility if some client code relies on the presence of `DefaultImpls` classes.

> If interface delegation is used, all interface methods are delegated. The only exception are methods annotated with the deprecated `@JvmDefault` annotation.
>
> 

all-compatibility

In addition to the `all` mode, generate compatibility stubs in the `DefaultImpls` classes. Compatibility stubs could be useful for library and runtime authors to keep backward binary compatibility for existing clients compiled against previous library versions. `all` and `all-compatibility` modes are changing the library ABI surface that clients will use after the recompilation of the library. In that sense, clients might be incompatible with previous library versions. This usually means that you need a proper library versioning, for example, major version increase in SemVer.

The compiler generates all the members of `DefaultImpls` with the `@Deprecated` annotation: you shouldn't use these members in Java code, because the compiler generates them only for compatibility purposes.

In case of inheritance from a Kotlin interface compiled in `all` or `all-compatibility` modes, `DefaultImpls` compatibility stubs will invoke the default method of the interface with standard JVM runtime resolution semantics.

Perform additional compatibility checks for classes inheriting generic interfaces where in some cases additional implicit method with specialized signatures was generated in the `disable` mode: unlike in the `disable` mode, the compiler will report an error if you don't override such method explicitly and don't annotate the class with `@JvmDefaultWithoutCompatibility` (see this YouTrack issue for more details).

可见性

Kotlin 的可见性以下列方式映射到 Java：

- `private` 成员编译成 `private` 成员

- `private` 的顶层声明编译成 `private` 顶层声明。 Package-private accessors are also included, if accessed from within a class.
- `protected` 保持 `protected`（注意 Java 允许访问同一个包中其他类的受保护成员， 而 Kotlin 不能，所以 Java 类会访问更广泛的代码）
- `internal` 声明会成为 Java 中的 `public`。 `internal` 类的成员会通过名字修饰，使其更难以在 Java 中意外使用到，并且根据 Kotlin 规则使其允许重载相同签名的成员而互不可见
- `public` 保持 `public`

KClass

有时你需要调用有 `KClass` 类型参数的 Kotlin 方法。 因为没有从 `Class` 到 `KClass` 的自动转换，所以你必须通过调用 `Class<T>.kotlin` 扩展属性的等价形式来手动进行转换：

```
kotlin.jvm.JvmClassMappingKt.getKotlinClass(MainView.class)
```

用 @JvmName 解决签名冲突

有时我们想让一个 Kotlin 中的具名函数在字节码中有另外一个 JVM 名称。 最突出的例子是由于类型擦除引发的：

```
fun List<String>.filterValid(): List<String>
fun List<Int>.filterValid(): List<Int>
```

这两个函数不能同时定义，因为它们的 JVM 签名是一样的：`filterValid(Ljava/util/List;)Ljava/util/List;`。 如果我们真的希望它们在 Kotlin 中用相同名称，我们需要用 `@JvmName` 去标注其中的一个（或两个），并指定不同的名称作为参数：

```
fun List<String>.filterValid(): List<String>

@JvmName("filterValidInt")
fun List<Int>.filterValid(): List<Int>
```

在 Kotlin 中它们可以用相同的名称 `filterValid` 来访问，而在 Java 中，它们分别是 `filterValid` 和 `filterValidInt`。

同样的技巧也适用于属性 `x` 和函数 `getX()` 共存：

```
val x: Int
```

```
    @JvmName("getX_prop")
    get() = 15

fun getX() = 10
```

如需在没有显式实现 getter 与 setter 的情况下更改属性生成的访问器方法的名称， 可以使用 `@get:JvmName` 与 `@set:JvmName`：

```
@get:JvmName("x")
@set:JvmName("changeX")
var x: Int = 23
```

生成重载

通常，如果你写一个有默认实参值的 Kotlin 函数，在 Java 中只会有一个所有参数都存在的完整参数签名的方法可见，如果希望向 Java 调用者暴露多个重载，可以使用 `@JvmOverloads` 注解。

该注解也适用于构造函数、静态方法等。它不能用于抽象方法，包括在接口中定义的方法。

```
class Circle @JvmOverloads constructor(centerX: Int, centerY: Int, radius: Double = 1.0
) {
    @JvmOverloads fun draw(label: String, lineWidth: Int = 1, color: String = "red") {
/*……*/ }
}
```

对于每一个有默认值的参数，都会生成一个额外的重载，这个重载会把这个参数和它右边的所有参数都移除掉。在上例中，会生成以下代码：

```
// 构造函数:
Circle(int centerX, int centerY, double radius)
Circle(int centerX, int centerY)

// 方法
void draw(String label, int lineWidth, String color) { }
void draw(String label, int lineWidth) { }
void draw(String label) { }
```

请注意，如次构造函数中所述，如果一个类的所有构造函数参数都有默认值，那么会为其生成一个公有的无参构造函数。这就算没有 `@JvmOverloads` 注解也有效。

受检异常

Kotlin 没有受检异常。 所以，通常 Kotlin 函数的 Java 签名不会声明抛出异常。 于是，如果有一个这样的 Kotlin 函数：

```
// example.kt
package demo

fun writeToFile() {
    /*……*/
    throw IOException()
}
```

然后想要在 Java 中调用它并捕捉这个异常：

```
// Java
try {
    demo.Example.writeToFile();
} catch (IOException e) {
    // 错误：writeToFile() 未在 throws 列表中声明 IOException
    // ……
}
```

因为 `writeToFile()` 没有声明 `IOException`，从 Java 编译器得到了一个报错消息。 为了解决这个问题，要在 Kotlin 中使用 `@Throws` 注解。

```
@Throws(IOException::class)
fun writeToFile() {
    /*……*/
    throw IOException()
}
```

空安全性

当从 Java 中调用 Kotlin 函数时，没人阻止我们将 `null` 作为非空参数传递。 这就是为什么 Kotlin 给所有期望非空参数的公有函数生成运行时检测。 这样我们就能在 Java 代码里立即得到 `NullPointerException`。

型变的泛型

当 Kotlin 的类使用了声明处型变，有两种选择可以从 Java 代码中看到它们的用法。例如，假设我们有以下类和两个使用它的函数：

```
class Box<out T>(val value: T)

interface Base
class Derived : Base

fun boxDerived(value: Derived): Box<Derived> = Box(value)
fun unboxBase(box: Box<Base>): Base = box.value
```

一种看似理所当然地将这俩函数转换成 Java 代码的方式可能会是：

```
Box<Derived> boxDerived(Derived value) { …… }
Base unboxBase(Box<Base> box) { …… }
```

问题是，在 Kotlin 中可以这样写 `unboxBase(boxDerived(Derived()))`，但是在 Java 中是行不通的，因为在 Java 中类 `Box` 在其泛型参数 `T` 上是不型变的，于是 `Box<Derived>` 并不是 `Box<Base>` 的子类。要使其在 Java 中工作，必须按以下这样定义 `unboxBase`：

```
Base unboxBase(Box<? extends Base> box) { …… }
```

这个声明使用 Java 的通配符类型（`? extends Base`）来通过使用处型变来模拟声明处型变，因为在 Java 中只能这样。

当它作为参数出现时，为了让 Kotlin 的 API 在 Java 中工作，对于协变定义的 `Box`，编译器生成了 `Box<Super>` 作为 `Box<? extends Super>`（或者对于逆变定义的 `Foo` 生成 `Foo<? super Bar>`）。当它是一个返回值时，不生成通配符，因为否则 Java 客户端将必须处理它们（并且它违反常用 Java 编码风格）。因此，我们的示例中的对应函数实际上翻译如下：

```
// 作为返回类型—没有通配符
Box<Derived> boxDerived(Derived value) { …… }

// 作为参数—有通配符
Base unboxBase(Box<? extends Base> box) { …… }
```

> 当参数类型是 final 时，生成通配符通常没有意义，所以无论在什么地方 `Box<String>` 始终转换为 `Box<String>`。
>
> 

如果在默认不生成通配符的地方需要通配符，可以使用 `@JvmWildcard` 注解：

```
fun boxDerived(value: Derived): Box<@JvmWildcard Derived> = Box(value)
// 将被转换成
// Box<? extends Derived> boxDerived(Derived value) { …… }
```

相反，如果根本不需要默认的通配符转换，可以使用 `@JvmSuppressWildcards`

```
fun unboxBase(box: Box<@JvmSuppressWildcards Base>): Base = box.value
// 会翻译成
// Base unboxBase(Box<Base> box) { …… }
```

> `@JvmSuppressWildcards` 不仅可用于单个类型参数，还可用于整个声明（如函数或类），从而抑制其中的所有通配符。



Nothing 类型翻译

类型 `Nothing` 是特殊的，因为它在 Java 中没有自然的对应。确实，每个 Java 引用类型，包括 `java.lang.Void` 都可以接受 `null` 值，但是 Nothing 不行。因此，这种类型不能在 Java 世界中准确表示。这就是为什么在使用 `Nothing` 参数的地方 Kotlin 生成一个原始类型：

```
fun emptyList(): List<Nothing> = listOf()
// 会翻译成
// List emptyList() { …… }
```

Spring

- 使用 Spring Boot 创建用到数据库的 RESTful web 服务
  - Spring Boot 与 Kotlin 入门
  - 创建使用 Kotlin 的 Spring Boot 项目
  - 向 Spring Boot 项目添加数据类
  - 为 Spring Boot 项目添加数据库支持
  - 使用 Spring Data CrudRepository 进行数据库访问
- 构建一个使用 Spring AI 基于 Qdrant 中存储的文档来回答问题的 Kotlin 应用——教程
- Spring 框架 Kotlin 文档↗
- 使用 Spring Boot 与 Kotlin 构建 web 应用程序——教程↗
- 使用 Kotlin 协程与 RSocket 创建聊天应用程序——教程↗

使用 Spring Boot 创建用到数据库的 RESTful web 服务

- Spring Boot 与 Kotlin 入门
- 创建使用 Kotlin 的 Spring Boot 项目
- 向 Spring Boot 项目添加数据类
- 为 Spring Boot 项目添加数据库支持
- 使用 Spring Data CrudRepository 进行数据库访问

Spring Boot 与 Kotlin 入门

Get started with Spring Boot and Kotlin by completing this tutorial: it walks you through the process of creating a simple application with Spring Boot and adding a database to store the information.

Going through these four steps, you'll learn a lot of essential features of the Kotlin language:

1. Create a Spring Boot project
2. Add a data class to Spring Boot project
3. Add database support for the Spring Boot project
4. Use Spring Data CrudRepository for database access

Next step

Start by creating a Spring Boot project with Kotlin using IntelliJ IDEA.

See also

Look through our Java to Kotlin (J2K) interop and migration guides:

- Calling Java from Kotlin and Calling Kotlin from Java
- Collections in Java and Kotlin
- Strings in Java and Kotlin

Join the community

- **Kotlin slack**: get an invitation and join the #spring and #server channels
- **Stack Overflow**: subscribe to the "kotlin", "spring-kotlin", or "ktor" tags
- **Kotlin YouTube channel**: subscribe and watch videos about Kotlin with Spring

创建使用 Kotlin 的 Spring Boot 项目

This is the first part of the **Get started with Spring Boot and Kotlin** tutorial:

1. **Create a Spring Boot project with Kotlin**
2. Add a data class to the Spring Boot project
3. Add database support for the Spring Boot project
4. Use Spring Data CrudRepository for database access

The first part of the tutorial shows how to create a Spring Boot project with Gradle in IntelliJ IDEA using the Project Wizard.

> This tutorial doesn't require using Gradle as the build system. You can follow the same steps if you use Maven.
>
> 

Before you start

Download and install the latest version of IntelliJ IDEA Ultimate Edition.

> If you use IntelliJ IDEA Community Edition or another IDE, you can generate a Spring Boot project using a web-based project generator.



Create a Spring Boot project

Create a new Spring Boot project with Kotlin by using the Project Wizard in IntelliJ IDEA Ultimate Edition:

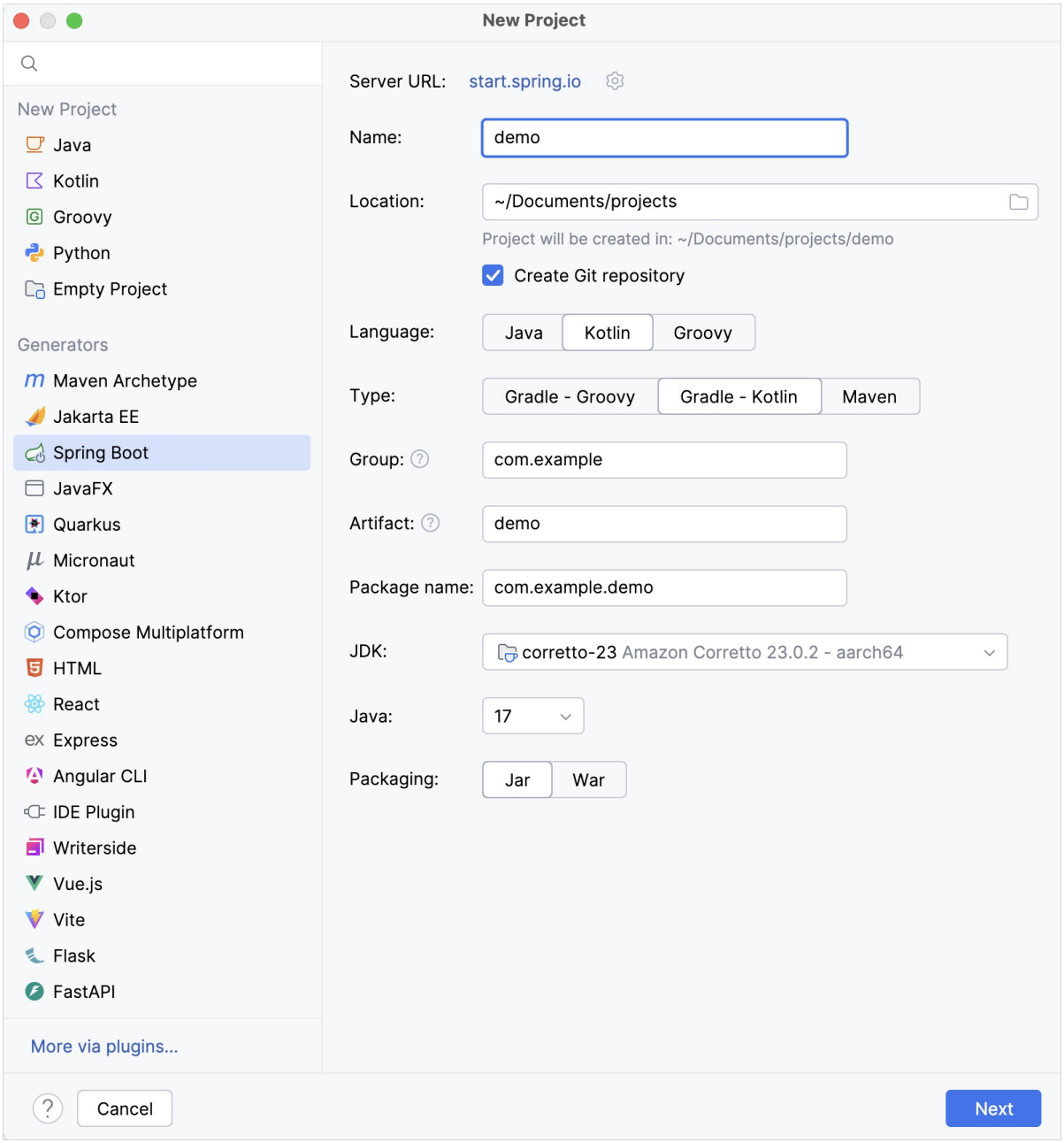
1. In IntelliJ IDEA, select **File** | **New** | **Project**.
2. In the panel on the left, select **New Project** | **Spring Boot**.
3. Specify the following fields and options in the **New Project** window:
    - **Name**: demo
    - **Language**: Kotlin
    - **Type**: Gradle - Kotlin

      > This option specifies the build system and the DSL.
      >
      > 

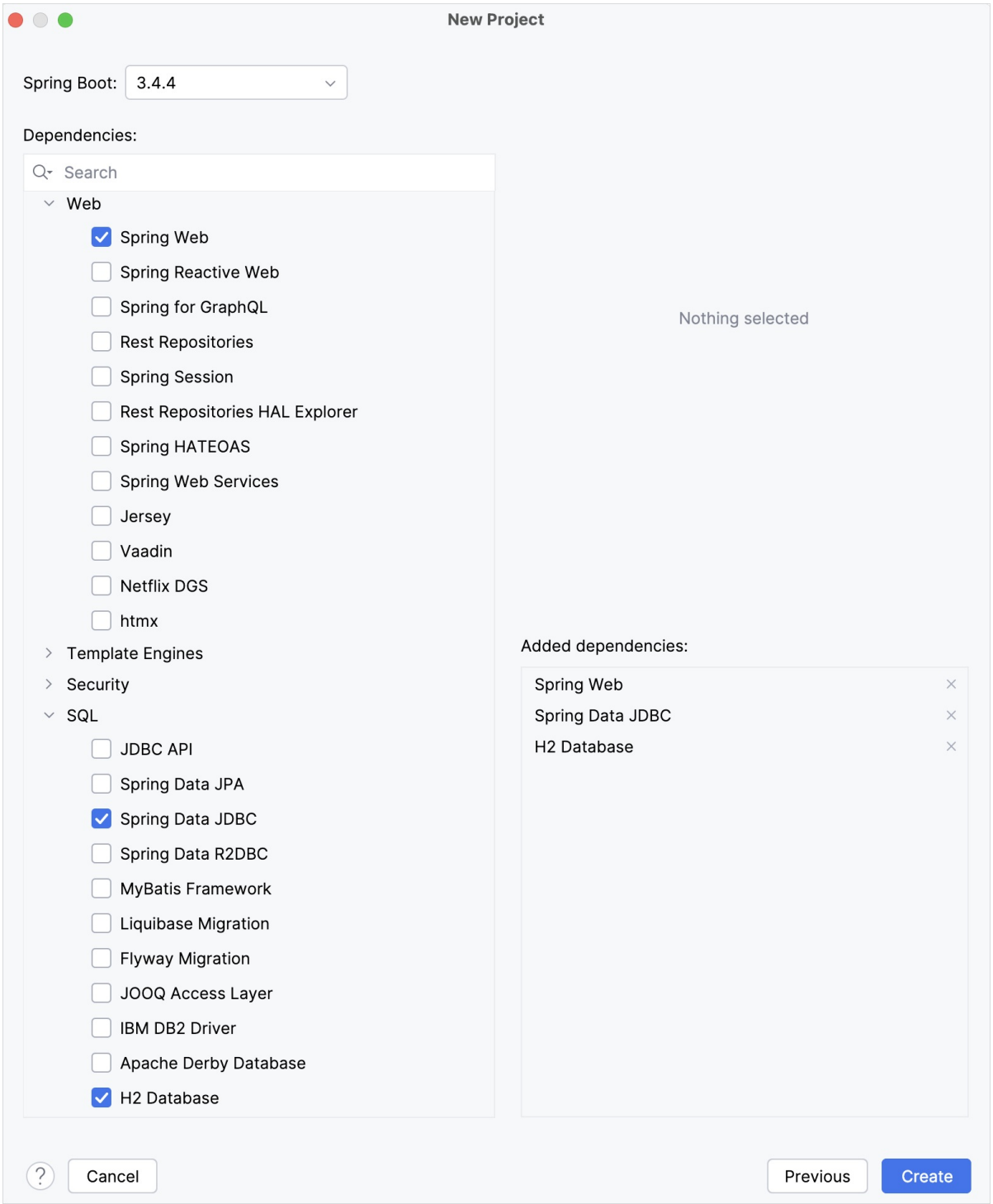
    - **Package name**: com.example.demo
    - **JDK**: Java JDK

      > This tutorial uses **Amazon Corretto version 23**. If you don't have a JDK installed, you can download it from the dropdown list.
      >
      > 

    - **Java**: 17



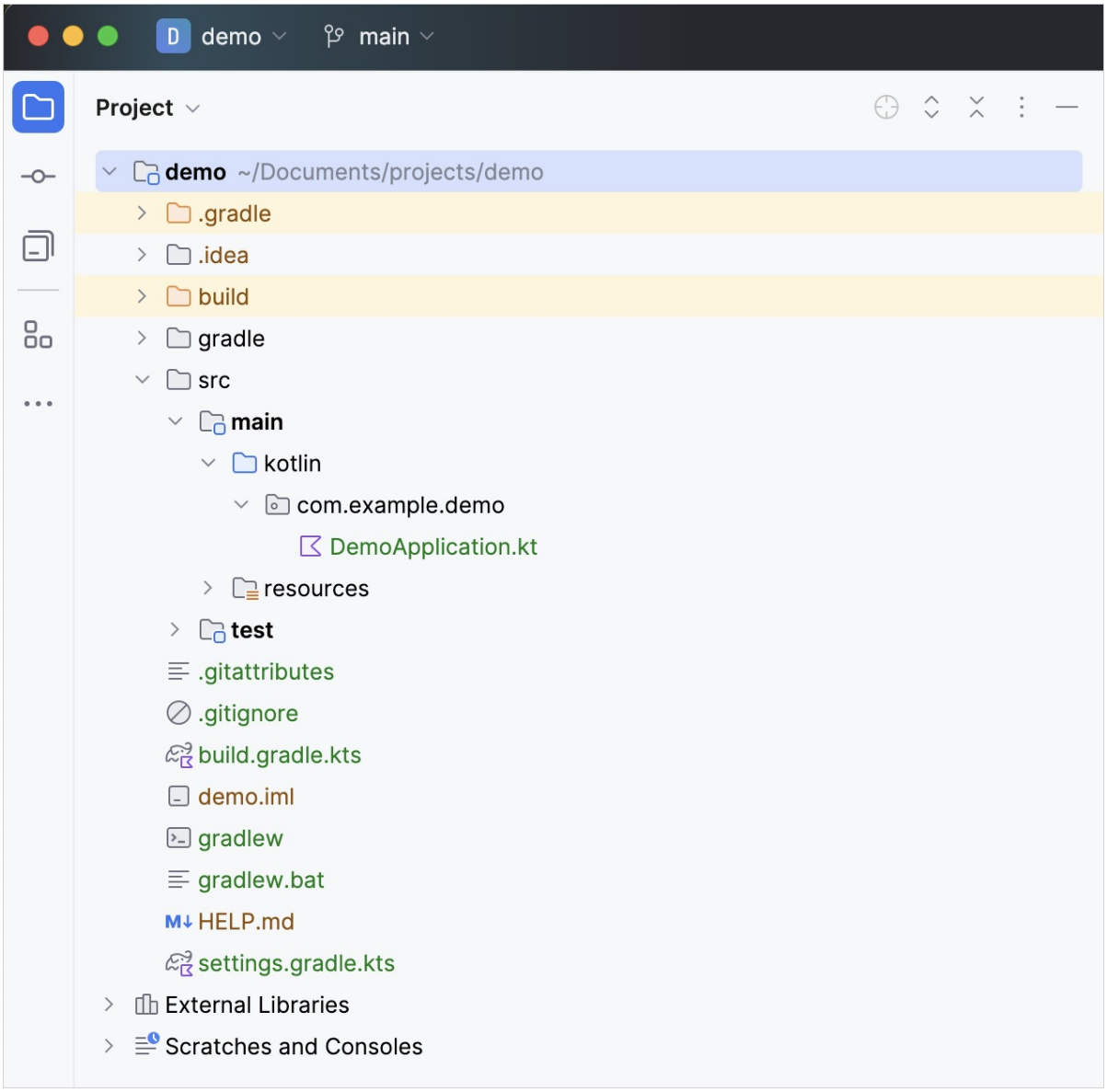
4. Ensure that you have specified all the fields and click **Next**.
5. Select the following dependencies that will be required for the tutorial:
    - **Web | Spring Web**
    - **SQL | Spring Data JDBC**
    - **SQL | H2 Database**



6. Click **Create** to generate and set up the project.

> The IDE will generate and open a new project. It may take some time to download and import the project dependencies.
>
> 

7. After this, you can observe the following structure in the **Project view**:



The generated Gradle project corresponds to the Maven's standard directory layout:

- There are packages and classes under the `main/kotlin` folder that belong to the application.
- The entry point to the application is the `main()` method of the `DemoApplication.kt` file.

Explore the project Gradle build file

Open the `build.gradle.kts` file: it is the Gradle Kotlin build script, which contains a list of the dependencies required for the application.

The Gradle file is standard for Spring Boot, but it also contains necessary Kotlin dependencies, including the kotlin-spring Gradle plugin – `kotlin("plugin.spring")` .

Here is the full script with the explanation of all parts and dependencies:

```
// build.gradle.kts
plugins {
    kotlin("jvm") version "1.9.25" // The version of Kotlin to use
    kotlin("plugin.spring") version "1.9.25" // The Kotlin Spring plugin
    id("org.springframework.boot") version "3.4.5"
    id("io.spring.dependency-management") version "1.1.7"
}

group = "com.example"
version = "0.0.1-SNAPSHOT"

java {
    toolchain {
        languageVersion = JavaLanguageVersion.of(17)
    }
}

repositories {
    mavenCentral()
}

dependencies {
    implementation("org.springframework.boot:spring-boot-starter-data-jdbc")
    implementation("org.springframework.boot:spring-boot-starter-web")
    implementation("com.fasterxml.jackson.module:jackson-module-kotlin") // Jackson ex
tensions for Kotlin for working with JSON
    implementation("org.jetbrains.kotlin:kotlin-reflect") // Kotlin reflection library
, required for working with Spring
    runtimeOnly("com.h2database:h2")
    testImplementation("org.springframework.boot:spring-boot-starter-test")
    testImplementation("org.jetbrains.kotlin:kotlin-test-junit5")
    testRuntimeOnly("org.junit.platform:junit-platform-launcher")
}

kotlin {
    compilerOptions {
        freeCompilerArgs.addAll("-Xjsr305=strict") // `-Xjsr305=strict` enables the st
rict mode for JSR-305 annotations
    }
}

tasks.withType<Test> {
    useJUnitPlatform()
}
```

As you can see, there are a few Kotlin-related artifacts added to the Gradle build file:

1. In the `plugins` block, there are two Kotlin artifacts:
    - `kotlin("jvm")` – the plugin defines the version of Kotlin to be used in the project
    - `kotlin("plugin.spring")` – Kotlin Spring compiler plugin for adding the `open` modifier

to Kotlin classes in order to make them compatible with Spring Framework features

2. In the `dependencies` block, a few Kotlin-related modules listed:
    - `com.fasterxml.jackson.module:jackson-module-kotlin` – the module adds support for serialization and deserialization of Kotlin classes and data classes
    - `org.jetbrains.kotlin:kotlin-reflect` – Kotlin reflection library
3. After the dependencies section, you can see the `kotlin` plugin configuration block. This is where you can add extra arguments to the compiler to enable or disable various language features.

Learn more about the Kotlin compiler options in .

Explore the generated Spring Boot application

Open the `DemoApplication.kt` file:

```
// DemoApplication.kt
package com.example.demo

import org.springframework.boot.autoconfigure.SpringBootApplication
import org.springframework.boot.runApplication

@SpringBootApplication
class DemoApplication

fun main(args: Array<String>) {
    runApplication<DemoApplication>(*args)
}
```

Right after package declaration and import statements you can see the first class declaration, `class DemoApplication`.

In Kotlin, if a class doesn't include any members (properties or functions), you can omit the class body (`{}`) for good.

`@SpringBootApplication annotation` is a convenience annotation in a Spring Boot application. It enables Spring Boot's auto-configuration, component scan, and be able to define an extra configuration on their "application class".

The `main()` function is the entry point to the application.

It is declared as a top-level function outside the `DemoApplication` class. The `main()` function invokes the Spring's `runApplication(*args)` function to start the application with the Spring Framework.

If you check the declaration of the `runApplication()` function, you will see that the parameter of the function is marked with `vararg` modifier: `vararg args: String`. This means that you can pass a variable number of String arguments to the function.

The `args` is a parameter to the `main()` function declared as an array of Strings. Since there is an array of strings, and you want to pass its content to the function, use the spread operator (prefix the array with a star sign `*`).

Create a controller

The application is ready to run, but let's update its logic first.

In the Spring application, a controller is used to handle the web requests. In the same package, next to the `DemoApplication.kt` file, create the `MessageController.kt` file with the `MessageController` class as follows:

```
// MessageController.kt
package com.example.demo

import org.springframework.web.bind.annotation.GetMapping
import org.springframework.web.bind.annotation.RequestParam
import org.springframework.web.bind.annotation.RestController

@RestController
class MessageController {
    @GetMapping("/")
    fun index(@RequestParam("name") name: String) = "Hello, $name!"
}
```

You need to tell Spring that `MessageController` is a REST Controller, so you should mark it with the `@RestController` annotation.

This annotation means this class will be picked up by the component scan because it's in the same package as our `DemoApplication` class.

`@GetMapping` marks the functions of the REST controller that implement the endpoints corresponding to HTTP GET calls:

@GetMapping("/") fun index(@RequestParam("name") name: String) = "Hello, $name!"
The function parameter `name` is marked with `@RequestParam` annotation. This annotation indicates that a method parameter should be bound to a web request parameter.

Hence, if you access the application at the root and supply a request parameter called "name", like `/?name=<your-value>`, the parameter value will be used as an argument for invoking the `index()` function.

Since the `index()` function contains only one statement you can declare it as a single-expression function.

This means the curly braces can be omitted and the body is specified after the equals sign `=`.

The `index()` function does not declare the return type explicitly. Instead, the compiler infers the return type by looking at the result of the statement on the right-hand side from the equals sign `=`.

The type of `Hello, $name!` expression is `String`, hence the return type of the function is also `String`.

`Hello, $name!` expression is called a *String template* in Kotlin.

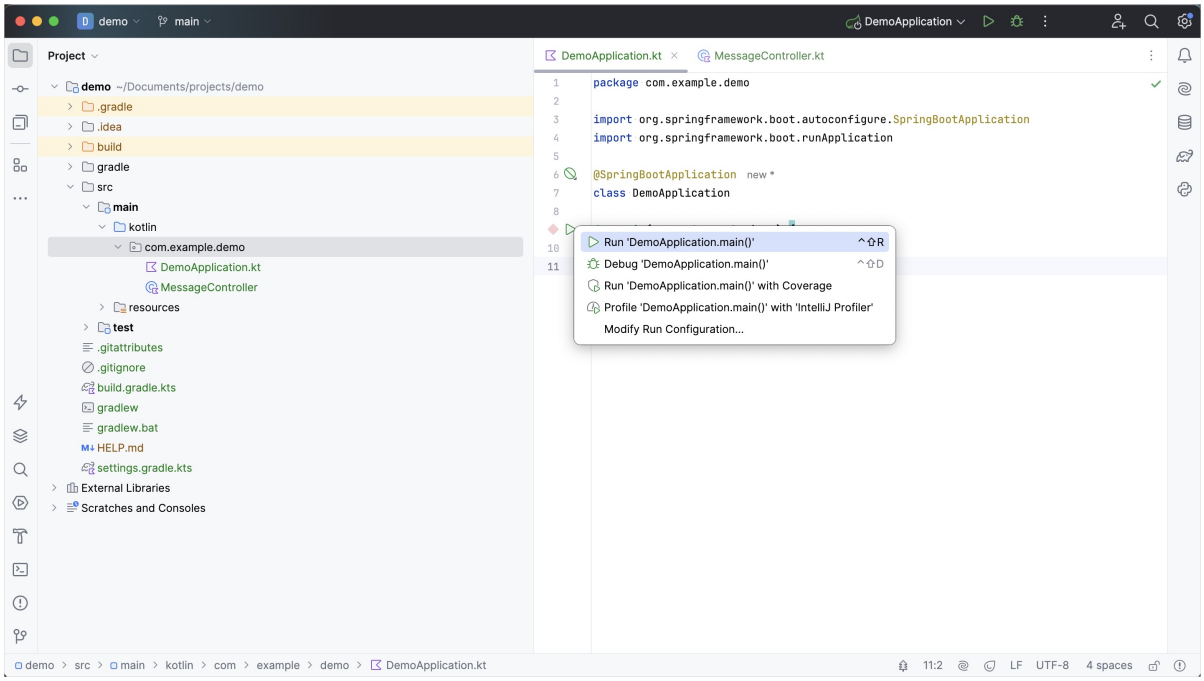
String templates are String literals that contain embedded expressions.

This is a convenient replacement for String concatenation operations.

Run the application

The Spring application is now ready to run:

1. In the `DemoApplication.kt` file, click the green **Run** icon in the gutter beside the `main()` method:



> You can also run the `./gradlew bootRun` command in the terminal.
>
> 
>
> This starts the local server on your computer.

2. Once the application starts, open the following URL:

```
http://localhost:8080?name=John
```

You should see "Hello, John!" printed as a response:



Next step

In the next part of the tutorial, you'll learn about Kotlin data classes and how you can use them in your application.

Proceed to the next chapter

向 Spring Boot 项目添加数据类

This is the second part of the **Getting started with Spring Boot and Kotlin** tutorial. Before proceeding, make sure you've completed previous steps:

1. Create a Spring Boot project with Kotlin

2. **Add a data class to the Spring Boot project**

3. Add database support for Spring Boot project

4. Use Spring Data CrudRepository for database access

In this part of the tutorial, you'll add some more functionality to the application and discover more Kotlin language features, such as data classes. It requires changing the `MessageController` class to respond with a JSON document containing a collection of serialized objects.

Update your application

1. In the same package, next to the `DemoApplication.kt` file, create a `Message.kt` file.
2. In the `Message.kt` file, create a data class with two properties: `id` and `text`:

```
// Message.kt
package com.example.demo

data class Message(val id: String?, val text: String)
```

`Message` class will be used for data transfer: a list of serialized `Message` objects will make up the JSON document that the controller is going to respond to the browser request.

The main purpose of data classes in Kotlin is to hold data. Such classes are marked with the `data` keyword, and some standard functionality and some utility functions are often mechanically derivable from the class structure.

In this example, you declared `Message` as a data class as its main purpose is to store the data.

Properties in Kotlin classes can be declared either as:

- *mutable*, using the `var` keyword
- *read-only*, using the `val` keyword

The `Message` class declares two properties using `val` keyword, the `id` and `text`. The compiler will automatically generate the getters for both of these properties. It will not be possible to reassign the values of these properties after an instance of the `Message` class is created.

Kotlin provides built-in support for nullable types. In Kotlin, the type system distinguishes between references that can hold `null` (*nullable references*) and those that cannot (*non-nullable references*).

For example, a regular variable of type `String` cannot hold `null`. To allow nulls, you can declare a variable as a nullable string by writing `String?`.

The `id` property of the `Message` class is declared as a nullable type this time. Hence, it is possible to create an instance of `Message` class by passing `null` as a value for `id`:

Message(null, "Hello!")

3. In the `MessageController.kt` file, instead of the `index()` function, create the `listMessages()` function returning a list of `Message` objects:

```
// MessageController.kt
package com.example.demo

import org.springframework.web.bind.annotation.GetMapping
import org.springframework.web.bind.annotation.RequestMapping
import org.springframework.web.bind.annotation.RestController

@RestController
@RequestMapping("/")
class MessageController {
    @GetMapping
    fun listMessages() = listOf(
        Message("1", "Hello!"),
```

```
            Message("2", "Bonjour!"),
            Message("3", "Privet!"),
        )
    }
```

The Kotlin Standard Library provides implementations for basic collection types: sets, lists, and maps.

Each collection type can be *read-only* or *mutable*:

- A *read-only* collection comes with operations for accessing collection elements.
- A *mutable* collection comes also with write operations for adding, removing, and updating its elements.

The corresponding factory functions are also provided by the Kotlin Standard Library to create instances of such collections.

In this tutorial, you use the `listOf()` function to create a list of `Message` objects. This is the factory function to create a *read-only* list of objects: you can't add or remove elements from the list.

If it is required to perform write operations on the list, call the `mutableListOf()` function to create a mutable list instance.

A trailing comma is a comma symbol after the **last item** of a series of elements:

Message("3", "Privet!"),
This is a convenient feature of Kotlin syntax and is entirely optional – your code will still work without them.

In the example above, creating a list of `Message` objects includes the trailing comma after the last `listOf()` function argument.

The response from `MessageController` will now be a JSON document containing a collection of `Message` objects.

> Any controller in the Spring application renders JSON response by default if Jackson library is on the classpath. As you specified the `spring-boot-starter-web` dependency in the `build.gradle.kts` file, you received Jackson as a *transitive* dependency. Hence, the application responds with a JSON document if the endpoint returns a data structure that can be serialized to JSON.



Here is a complete code of the `DemoApplication.kt`, `MessageController.kt`, and `Message.kt` files:

```
// DemoApplication.kt
package com.example.demo

import org.springframework.boot.autoconfigure.SpringBootApplication
import org.springframework.boot.runApplication

@SpringBootApplication
class DemoApplication

fun main(args: Array<String>) {
    runApplication<DemoApplication>(*args)
}
```

```
// MessageController.kt
package com.example.demo

import org.springframework.web.bind.annotation.GetMapping
import org.springframework.web.bind.annotation.RequestMapping
import org.springframework.web.bind.annotation.RestController

@RestController
@RequestMapping("/")
class MessageController {
    @GetMapping
    fun listMessages() = listOf(
        Message("1", "Hello!"),
        Message("2", "Bonjour!"),
        Message("3", "Privet!"),
    )
}
```

```
// Message.kt
package com.example.demo

data class Message(val id: String?, val text: String)
```

Run the application

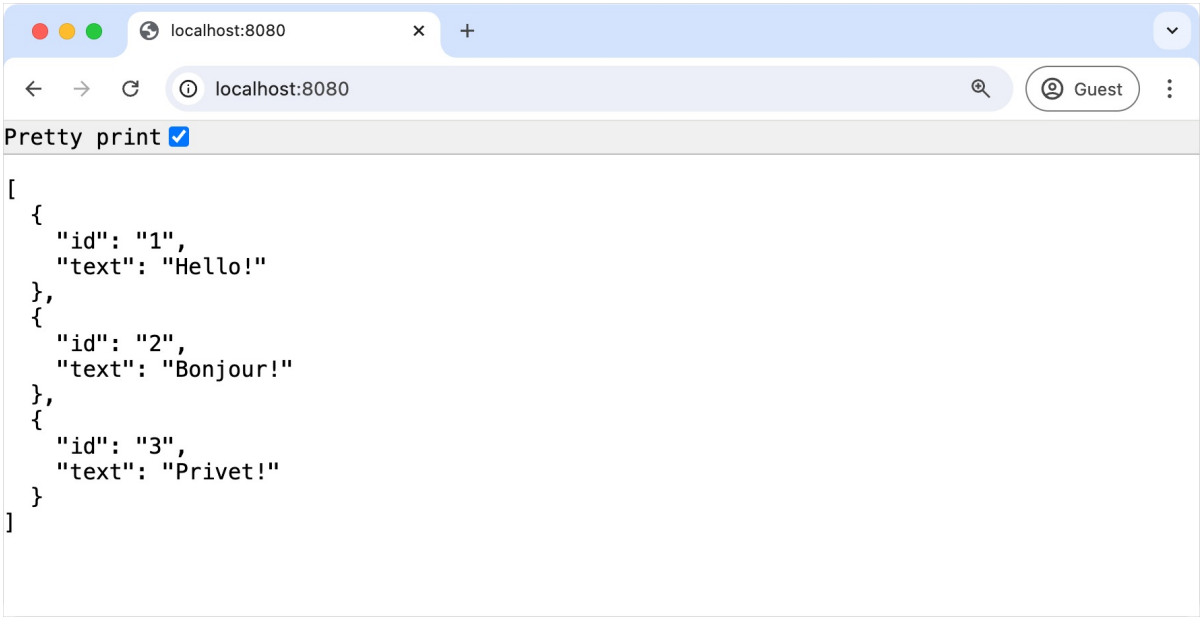
The Spring application is ready to run:

1. Run the application again.

2. Once the application starts, open the following URL:

   ```
   http://localhost:8080
   ```

   You will see a page with a collection of messages in JSON format:



Next step

In the next part of the tutorial, you'll add and configure a database to your project, and make HTTP requests.

Proceed to the next chapter

为 Spring Boot 项目添加数据库支持

This is the third part of the **Getting started with Spring Boot and Kotlin** tutorial. Before proceeding, make sure you've completed previous steps:

1. Create a Spring Boot project with Kotlin

2. Add a data class to the Spring Boot project

3. **Add database support for Spring Boot project**

4. Use Spring Data CrudRepository for database access

In this part of the tutorial, you'll add and configure a database to your project using *Java Database Connectivity* (JDBC). In JVM applications, you use JDBC to interact with databases. For convenience, the Spring Framework provides the `JdbcTemplate` class that simplifies the use of JDBC and helps to avoid common errors.

Add database support

The common practice in Spring Framework based applications is to implement the database access logic within the so-called *service* layer – this is where business logic lives. In Spring, you should mark classes with the `@Service` annotation to imply that the class belongs to the service layer of the application. In this application, you will create the `MessageService` class for this purpose.

In the same package, create the `MessageService.kt` file and the `MessageService` class as follows:

```
// MessageService.kt
```

```
package com.example.demo

import org.springframework.stereotype.Service
import org.springframework.jdbc.core.JdbcTemplate
import java.util.*

@Service
class MessageService(private val db: JdbcTemplate) {
    fun findMessages(): List<Message> = db.query("select * from messages") { response,
 _ ->
        Message(response.getString("id"), response.getString("text"))
    }

    fun save(message: Message): Message {
        db.update(
            "insert into messages values ( ?, ? )",
            message.id, message.text
        )
        return message
    }
}
```

A class in Kotlin has a primary constructor. It can also have one or more secondary constructors. The *primary constructor* is a part of the class header, and it goes after the class name and optional type parameters. In our case, the constructor is `(val db: JdbcTemplate)` .

`val db: JdbcTemplate` is the constructor's argument:

@Service class MessageService(private val db: JdbcTemplate)
The `findMessages()` function calls the `query()` function of the `JdbcTemplate` class. The `query()` function takes two arguments: an SQL query as a String instance, and a callback that will map one object per row:

db.query("...", RowMapper { ... })

The `RowMapper` interface declares only one method, so it is possible to implement it via lambda expression by omitting the name of the interface. The Kotlin compiler knows the interface that the lambda expression needs to be converted to because you use it as a parameter for the function call. This is known as SAM conversion in Kotlin:

db.query("...", { ... })

After the SAM conversion, the query function ends up with two arguments: a String at the first position, and a lambda expression at the last position. According to the Kotlin convention, if the last parameter of a function is a function, then a lambda expression passed as the corresponding argument can be placed outside the parentheses. Such syntax is also known as trailing lambda:

db.query("...") { ... }
For a lambda with multiple parameters, you can use the underscore `_` character to replace the names of the parameters you don't use.

Hence, the final syntax for query function call looks like this:

db.query("select * from messages") { response, _ -> Message(response.getString("id"), response.getString("text")) }

Update the MessageController class

Update `MessageController.kt` to use the new `MessageService` class:

```
// MessageController.kt
package com.example.demo

import org.springframework.http.ResponseEntity
import org.springframework.web.bind.annotation.GetMapping
import org.springframework.web.bind.annotation.PostMapping
import org.springframework.web.bind.annotation.RequestBody
import org.springframework.web.bind.annotation.RequestMapping
import org.springframework.web.bind.annotation.RestController
import java.net.URI

@RestController
@RequestMapping("/")
class MessageController(private val service: MessageService) {
    @GetMapping
    fun listMessages() = service.findMessages()

    @PostMapping
    fun post(@RequestBody message: Message): ResponseEntity<Message> {
        val savedMessage = service.save(message)
        return ResponseEntity.created(URI("/${savedMessage.id}")).body(savedMessage)
    }
}
```

The method responsible for handling HTTP POST requests needs to be annotated with `@PostMapping` annotation. To be able to convert the JSON sent as HTTP Body content into an object, you need to use the `@RequestBody` annotation for the method argument. Thanks to having Jackson library in the classpath of the application, the conversion happens automatically.

`ResponseEntity` represents the whole HTTP response: status code, headers, and body.

Using the `created()` method you configure the response status code (201) and set the location header indicating the context path for the created resource.

Update the MessageService class

The `id` for `Message` class was declared as a nullable String:

```
data class Message(val id: String?, val text: String)
```

It would not be correct to store the `null` as an `id` value in the database though: you need to handle this situation gracefully.

Update your code of the `MessageService.kt` file to generate a new value when the `id` is `null` while storing the messages in the database:

```
// MessageService.kt
package com.example.demo

import org.springframework.stereotype.Service
import org.springframework.jdbc.core.JdbcTemplate
import org.springframework.jdbc.core.query
import java.util.UUID

@Service
class MessageService(private val db: JdbcTemplate) {
    fun findMessages(): List<Message> = db.query("select * from messages") { response,
 _ ->
        Message(response.getString("id"), response.getString("text"))
    }

    fun save(message: Message): Message {
        val id = message.id ?: UUID.randomUUID().toString() // Generate new id if it i
s null
        db.update(
            "insert into messages values ( ?, ? )",
            id, message.text
        )
        return message.copy(id = id) // Return a copy of the message with the new id
    }
}
```

The code `message.id ?: UUID.randomUUID().toString()` uses the Elvis operator (if-not-null-else shorthand) `?:`. If the expression to the left of `?:` is not `null`, the Elvis operator returns it; otherwise, it returns the expression to the right. Note that the expression on the right-hand side is

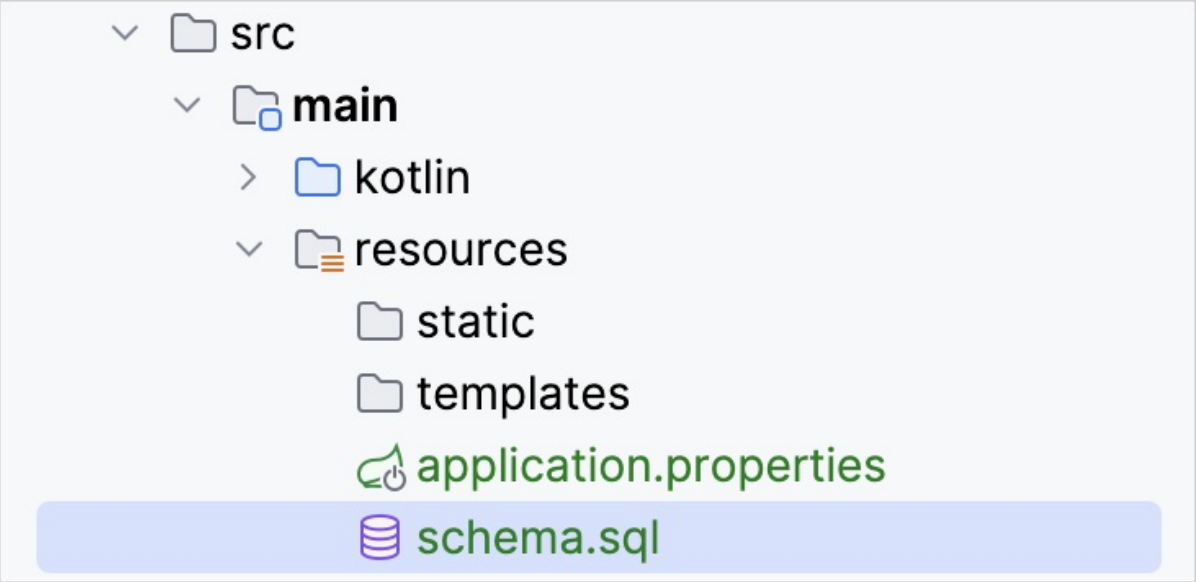
evaluated only if the left-hand side is `null`.

The application code is ready to work with the database. It is now required to configure the data source.

Configure the database

Configure the database in the application:

1. Create `schema.sql` file in the `src/main/resources` directory. It will store the database object definitions:



2. Update the `src/main/resources/schema.sql` file with the following code:

```
-- schema.sql
CREATE TABLE IF NOT EXISTS messages (
id       VARCHAR(60)  PRIMARY KEY,
text     VARCHAR      NOT NULL
);
```

   It creates the `messages` table with two columns: `id` and `text`. The table structure matches the structure of the `Message` class.

3. Open the `application.properties` file located in the `src/main/resources` folder and add the following application properties:

```
spring.application.name=demo
spring.datasource.driver-class-name=org.h2.Driver
spring.datasource.url=jdbc:h2:file:./data/testdb
spring.datasource.username=name
spring.datasource.password=password
```

```
spring.sql.init.schema-locations=classpath:schema.sql
spring.sql.init.mode=always
```

These settings enable the database for the Spring Boot application.

See the full list of common application properties in the Spring documentation.

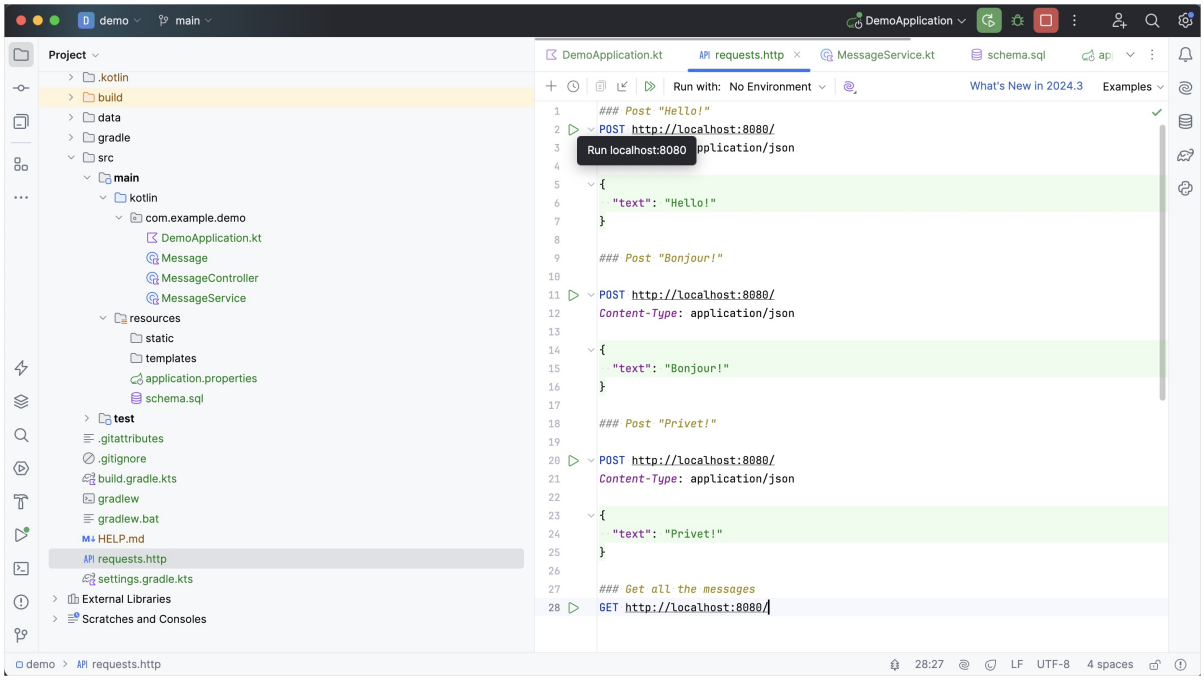
Add messages to database via HTTP request

You should use an HTTP client to work with previously created endpoints. In IntelliJ IDEA, use the embedded HTTP client:

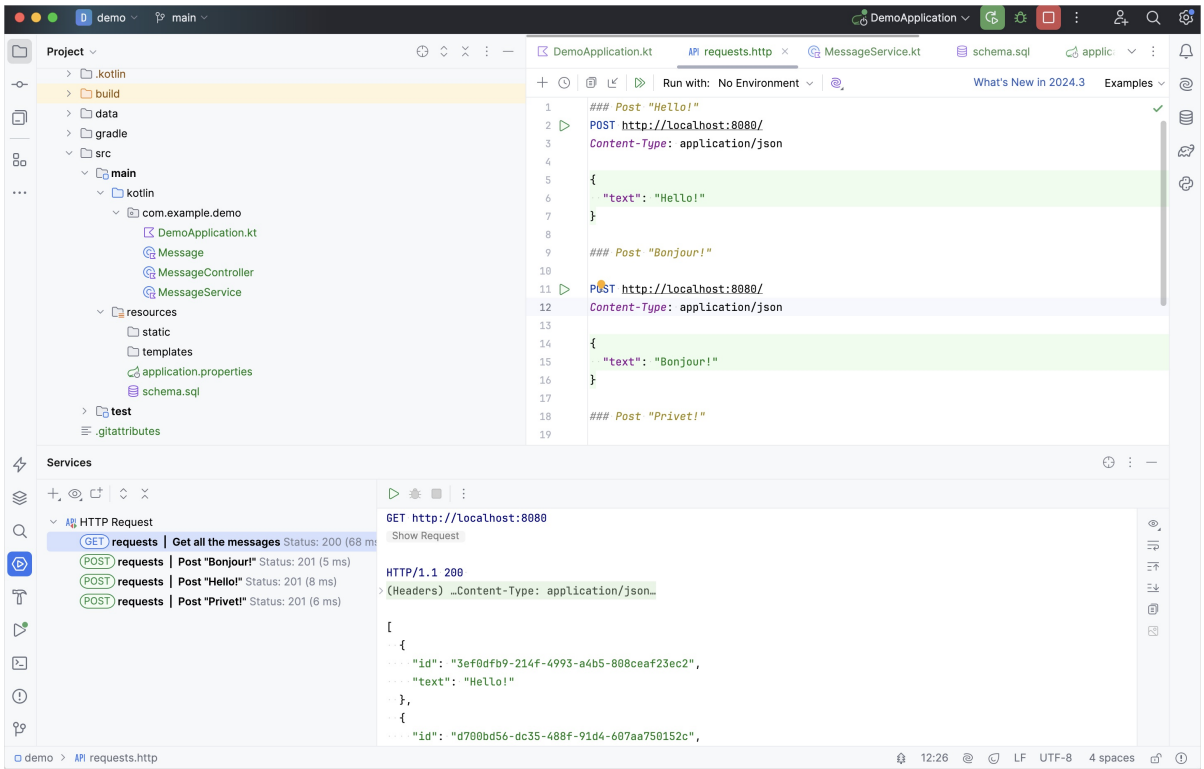
1. Run the application. Once the application is up and running, you can execute POST requests to store messages in the database.
2. Create the `requests.http` file in the project root folder and add the following HTTP requests:

   ```http request

   ## Post "Hello!"

   POST http://localhost:8080/ Content-Type: application/json

   { "text": "Hello!" }

   ## Post "Bonjour!"

   POST http://localhost:8080/ Content-Type: application/json

   { "text": "Bonjour!" }

   ## Post "Privet!"

   POST http://localhost:8080/ Content-Type: application/json

   { "text": "Privet!" }

   ## Get all the messages

   GET http://localhost:8080/ ```
   ```

3. Execute all POST requests. Use the green **Run** icon in the gutter next to the request declaration. These requests write the text messages to the database:



4. Execute the GET request and see the result in the **Run** tool window:



Alternative way to execute requests

You can also use any other HTTP client or the cURL command-line tool. For example, run the following commands in the terminal to get the same result:

```
curl -X POST --location "http://localhost:8080" -H "Content-Type: application/json" -d
"{ \"text\": \"Hello!\" }"

curl -X POST --location "http://localhost:8080" -H "Content-Type: application/json" -d
"{ \"text\": \"Bonjour!\" }"

curl -X POST --location "http://localhost:8080" -H "Content-Type: application/json" -d
"{ \"text\": \"Privet!\" }"

curl -X GET --location "http://localhost:8080"
```

Retrieve messages by id

Extend the functionality of the application to retrieve the individual messages by id.

1. In the `MessageService` class, add the new function `findMessageById(id: String)` to retrieve the individual messages by id:

```
// MessageService.kt
package com.example.demo

import org.springframework.stereotype.Service
import org.springframework.jdbc.core.JdbcTemplate
import org.springframework.jdbc.core.query
import java.util.*

@Service
class MessageService(private val db: JdbcTemplate) {
    fun findMessages(): List<Message> = db.query("select * from messages") { resp
onse, _ ->
        Message(response.getString("id"), response.getString("text"))
    }

    fun findMessageById(id: String): Message? = db.query("select * from messages
where id = ?", id) { response, _ ->
        Message(response.getString("id"), response.getString("text"))
    }.singleOrNull()

    fun save(message: Message): Message {
        val id = message.id ?: UUID.randomUUID().toString() // Generate new id if
 it is null
        db.update(
            "insert into messages values ( ?, ? )",
            id, message.text
        )
        return message.copy(id = id) // Return a copy of the message with the new
 id
    }
```

```
}
```

The `query()` function takes three arguments:

- SQL query string that requires a parameter to run
- `id`, which is a parameter of type String
- `RowMapper` instance, which implemented by a lambda expression

The second parameter for the `query()` function is declared as a *variable argument* (`vararg`). In Kotlin, the position of the variable arguments parameter is not required to be the last in the parameters list.

The `singleOrNull()` function returns a single element, or `null` if the array is empty or has more than one element with the same value.

> The `.query()` function that is used to fetch the message by its id is a Kotlin extension function provided by the Spring Framework. It requires an additional import `import org.springframework.jdbc.core.query` as demonstrated in the code above.
>
> 

2. Add the new `index(...)` function with the `id` parameter to the `MessageController` class:

```
// MessageController.kt
package com.example.demo

import org.springframework.http.ResponseEntity
import org.springframework.web.bind.annotation.GetMapping
import org.springframework.web.bind.annotation.PathVariable
import org.springframework.web.bind.annotation.PostMapping
import org.springframework.web.bind.annotation.RequestBody
import org.springframework.web.bind.annotation.RequestMapping
import org.springframework.web.bind.annotation.RestController
import java.net.URI

@RestController
@RequestMapping("/")
class MessageController(private val service: MessageService) {
    @GetMapping
    fun listMessages() = ResponseEntity.ok(service.findMessages())

    @PostMapping
    fun post(@RequestBody message: Message): ResponseEntity<Message> {
        val savedMessage = service.save(message)
        return ResponseEntity.created(URI("/${savedMessage.id}")).body(savedMessa
ge)
    }

    @GetMapping("/{id}")
    fun getMessage(@PathVariable id: String): ResponseEntity<Message> =
```

```
            service.findMessageById(id).toResponseEntity()

        private fun Message?.toResponseEntity(): ResponseEntity<Message> =
            // If the message is null (not found), set response code to 404
            this?.let { ResponseEntity.ok(it) } ?: ResponseEntity.notFound().build()
    }
```

The message `id` is retrieved from the context path by the Spring Framework as you annotated the new function by `@GetMapping("/{id}")`. By annotating the function argument with `@PathVariable`, you tell the framework to use the retrieved value as a function argument. The new function makes a call to `MessageService` to retrieve the individual message by its id.

Extensions can be defined with a nullable receiver type. If the receiver is `null`, then `this` is also `null`. So when defining an extension with a nullable receiver type, it is recommended performing a `this == null` check inside the function body.

You can also use the null-safe invocation operator (`?.`) to perform the null check as in the `toResponseEntity()` function above:

this?.let { ResponseEntity.ok(it) }
`ResponseEntity` represents the HTTP response, including the status code, headers, and body. It is a generic wrapper that allows you to send customized HTTP responses back to the client with more control over the content.

Here is a complete code of the application:

```
// DemoApplication.kt
package com.example.demo

import org.springframework.boot.autoconfigure.SpringBootApplication
import org.springframework.boot.runApplication

@SpringBootApplication
class DemoApplication

fun main(args: Array<String>) {
    runApplication<DemoApplication>(*args)
}
```

```
// Message.kt
package com.example.demo

data class Message(val id: String?, val text: String)
```

```
// MessageService.kt
package com.example.demo
```

```
import org.springframework.stereotype.Service
import org.springframework.jdbc.core.JdbcTemplate
import org.springframework.jdbc.core.query
import java.util.*

@Service
class MessageService(private val db: JdbcTemplate) {
    fun findMessages(): List<Message> = db.query("select * from messages") { response,
 _ ->
        Message(response.getString("id"), response.getString("text"))
    }

    fun findMessageById(id: String): Message? = db.query("select * from messages where
 id = ?", id) { response, _ ->
        Message(response.getString("id"), response.getString("text"))
    }.singleOrNull()

    fun save(message: Message): Message {
        val id = message.id ?: UUID.randomUUID().toString()
        db.update(
            "insert into messages values ( ?, ? )",
            id, message.text
        )
        return message.copy(id = id)
    }
}
```

```
// MessageController.kt
package com.example.demo

import org.springframework.http.ResponseEntity
import org.springframework.web.bind.annotation.GetMapping
import org.springframework.web.bind.annotation.PathVariable
import org.springframework.web.bind.annotation.PostMapping
import org.springframework.web.bind.annotation.RequestBody
import org.springframework.web.bind.annotation.RequestMapping
import org.springframework.web.bind.annotation.RestController
import java.net.URI

@RestController
@RequestMapping("/")
class MessageController(private val service: MessageService) {
    @GetMapping
    fun listMessages() = ResponseEntity.ok(service.findMessages())

    @PostMapping
    fun post(@RequestBody message: Message): ResponseEntity<Message> {
        val savedMessage = service.save(message)
        return ResponseEntity.created(URI("/${savedMessage.id}")).body(savedMessage)
    }
```

```
    @GetMapping("/{id}")
    fun getMessage(@PathVariable id: String): ResponseEntity<Message> =
        service.findMessageById(id).toResponseEntity()

    private fun Message?.toResponseEntity(): ResponseEntity<Message> =
        this?.let { ResponseEntity.ok(it) } ?: ResponseEntity.notFound().build()
}
```

Run the application

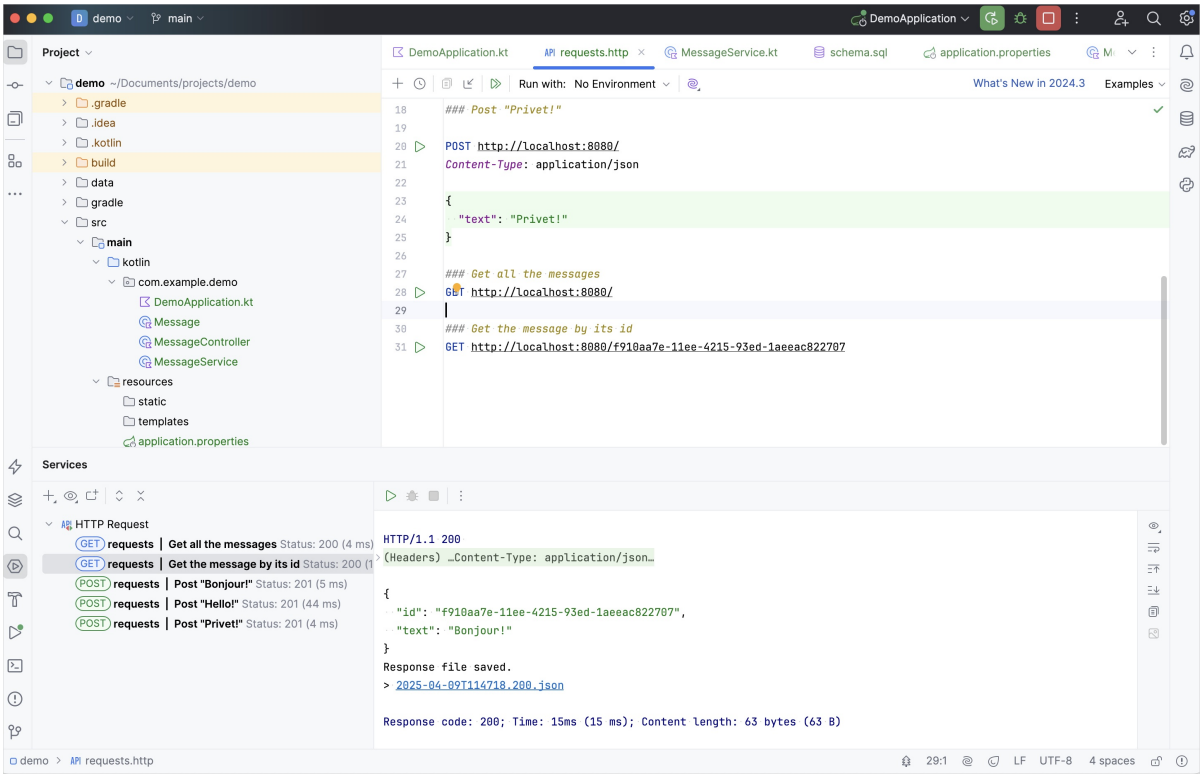
The Spring application is ready to run:

1. Run the application again.
2. Open the `requests.http` file and add the new GET request:

   ```http request

   ## Get the message by its id

   GET http://localhost:8080/id
   ```
3. Execute the GET request to retrieve all the messages from the database.
4. In the **Run** tool window copy one of the ids and add it to the request, like this:

   ```http request

   ## Get the message by its id

   GET http://localhost:8080/f910aa7e-11ee-4215-93ed-1aeeac822707
   ```

   > Put your message id instead of the mentioned above.
   >
   > 
5. Execute the GET request and see the result in the **Run** tool window:



Next step

The final step shows you how to use more popular connection to database using Spring Data.

Proceed to the next chapter

使用 Spring Data CrudRepository 进行数据库访问

This is the final part of the **Getting started with Spring Boot and Kotlin** tutorial. Before proceeding, make sure you've completed previous steps:

① Create a Spring Boot project with Kotlin

② Add a data class to the Spring Boot project

③ Add database support for Spring Boot project

④ **Use Spring Data CrudRepository for database access**

In this part, you will migrate the service layer to use the Spring Data `CrudRepository` instead of `JdbcTemplate` for database access. *CrudRepository* is a Spring Data interface for generic CRUD operations on a repository of a specific type. It provides several methods out of the box for interacting with a database.

Update your application

First, you need to adjust the `Message` class for work with the `CrudRepository` API:

1. Add the `@Table` annotation to the `Message` class to declare mapping to a database table.

Add the `@Id` annotation before the `id` field.

> These annotations also require additional imports.
>
> 

```
// Message.kt
package com.example.demo

import org.springframework.data.annotation.Id
import org.springframework.data.relational.core.mapping.Table

@Table("MESSAGES")
data class Message(@Id val id: String?, val text: String)
```

In addition, to make the use of the `Message` class more idiomatic, you can set the default value for `id` property to null and flip the order of the data class properties:

```
@Table("MESSAGES")
data class Message(val text: String, @Id val id: String? = null)
```

Now if you need to create a new instance of the `Message` class, you can only specify the `text` property as a parameter:

```
val message = Message("Hello") // id is null
```

2. Declare an interface for the `CrudRepository` that will work with the `Message` data class. Create the `MessageRepository.kt` file and add the following code to it:

```
// MessageRepository.kt
package com.example.demo

import org.springframework.data.repository.CrudRepository

interface MessageRepository : CrudRepository<Message, String>
```

3. Update the `MessageService` class. It will now use the `MessageRepository` instead of executing SQL queries:

```
// MessageService.kt
package com.example.demo

import org.springframework.data.repository.findByIdOrNull
import org.springframework.stereotype.Service
```

```
@Service
class MessageService(private val db: MessageRepository) {
    fun findMessages(): List<Message> = db.findAll().toList()

    fun findMessageById(id: String): Message? = db.findByIdOrNull(id)

    fun save(message: Message): Message = db.save(message)
}
```

The `findByIdOrNull()` function is an extension function for `CrudRepository` interface in Spring Data JDBC.

This function works with an assumption that the new object doesn't have an id in the database. Hence, the id **should be null** for insertion.

If the id isn't *null*, `CrudRepository` assumes that the object already exists in the database and this is an *update* operation as opposed to an *insert* operation. After the insert operation, the `id` will be generated by the data store and assigned back to the `Message` instance. This is why the `id` property should be declared using the `var` keyword.

4. Update the messages table definition to generate the ids for the inserted objects. Since `id` is a string, you can use the `RANDOM_UUID()` function to generate the id value by default:

```
-- schema.sql
CREATE TABLE IF NOT EXISTS messages (
    id      VARCHAR(60)  DEFAULT RANDOM_UUID() PRIMARY KEY,
    text    VARCHAR      NOT NULL
);
```

5. Update the name of the database in the `application.properties` file located in the `src/main/resources` folder:

```
spring.application.name=demo
spring.datasource.driver-class-name=org.h2.Driver
spring.datasource.url=jdbc:h2:file:./data/testdb2
spring.datasource.username=name
spring.datasource.password=password
spring.sql.init.schema-locations=classpath:schema.sql
spring.sql.init.mode=always
```

Here is the complete code of the application:

```
// DemoApplication.kt
package com.example.demo
```

```
import org.springframework.boot.autoconfigure.SpringBootApplication
import org.springframework.boot.runApplication

@SpringBootApplication
class DemoApplication

fun main(args: Array<String>) {
    runApplication<DemoApplication>(*args)
}
```

```
// Message.kt
package com.example.demo

import org.springframework.data.annotation.Id
import org.springframework.data.relational.core.mapping.Table

@Table("MESSAGES")
data class Message(val text: String, @Id val id: String? = null)
```

```
// MessageRepository.kt
package com.example.demo

import org.springframework.data.repository.CrudRepository

interface MessageRepository : CrudRepository<Message, String>
```

```
// MessageService.kt
package com.example.demo

import org.springframework.data.repository.findByIdOrNull
import org.springframework.stereotype.Service

@Service
class MessageService(private val db: MessageRepository) {
    fun findMessages(): List<Message> = db.findAll().toList()

    fun findMessageById(id: String): Message? = db.findByIdOrNull(id)

    fun save(message: Message): Message = db.save(message)
}
```

```
// MessageController.kt
package com.example.demo

import org.springframework.http.ResponseEntity
import org.springframework.web.bind.annotation.GetMapping
import org.springframework.web.bind.annotation.PathVariable
```

```
import org.springframework.web.bind.annotation.PostMapping
import org.springframework.web.bind.annotation.RequestBody
import org.springframework.web.bind.annotation.RequestMapping
import org.springframework.web.bind.annotation.RestController
import java.net.URI

@RestController
@RequestMapping("/")
class MessageController(private val service: MessageService) {
    @GetMapping
    fun listMessages() = ResponseEntity.ok(service.findMessages())

    @PostMapping
    fun post(@RequestBody message: Message): ResponseEntity<Message> {
        val savedMessage = service.save(message)
        return ResponseEntity.created(URI("/${savedMessage.id}")).body(savedMessage)
    }

    @GetMapping("/{id}")
    fun getMessage(@PathVariable id: String): ResponseEntity<Message> =
        service.findMessageById(id).toResponseEntity()

    private fun Message?.toResponseEntity(): ResponseEntity<Message> =
        // If the message is null (not found), set response code to 404
        this?.let { ResponseEntity.ok(it) } ?: ResponseEntity.notFound().build()
}
```

Run the application

Congratulations! The application is ready to run again. After replacing `JdbcTemplate` with `CrudRepository`, the functionality remains the same, so the application works just as before.

You can now run the POST and GET HTTP requests from the `requests.http` file and get the same results.

下一步做什么

Get your personal language map to help you navigate Kotlin features and track your progress in studying the language:



- Learn more about Calling Java from Kotlin code and Calling Kotlin from Java code.
- Learn how to convert existing Java code to Kotlin with the Java-to-Kotlin converter.

- Check out our Java to Kotlin migration guides:
  - Strings in Java and Kotlin.
  - Collections in Java and Kotlin.
  - Nullability in Java and Kotlin.

构建一个使用 Spring AI 基于 Qdrant 中存储的文档来回答问题的 Kotlin 应用——教程

In this tutorial, you'll learn how to build a Kotlin app that uses Spring AI to connect to an LLM, store documents in a vector database, and answer questions using context from those documents.

You will use the following tools during this tutorial:

- Spring Boot as the base to configure and run the web application.
- Spring AI to interact with the LLM and perform context-based retrieval.
- IntelliJ IDEA to generate the project and implement the application logic.
- Qdrant as the vector database for similarity search.
- Docker to run Qdrant locally.
- OpenAI as the LLM provider.

开始之前

1. Download and install the latest version of IntelliJ IDEA Ultimate Edition.

   > If you use IntelliJ IDEA Community Edition or another IDE, you can generate a Spring Boot project using a web-based project generator.
   >
   > 

2. Create an OpenAI API key on the OpenAI platform to access the API.
3. Install Docker to run the Qdrant vector database locally.
4. After installing Docker, open your terminal and run the following command to start the container:

   ```
   docker run -p 6333:6333 -p 6334:6334 qdrant/qdrant
   ```

创建项目

> You can use Spring Boot web-based project generator as an alternative to generate your project.
>
> 

Create a new Spring Boot project in IntelliJ IDEA Ultimate Edition:

1. In IntelliJ IDEA, select **File** | **New** | **Project**.
2. In the panel on the left, select **New Project** | **Spring Boot**.
3. Specify the following fields and options in the **New Project** window:

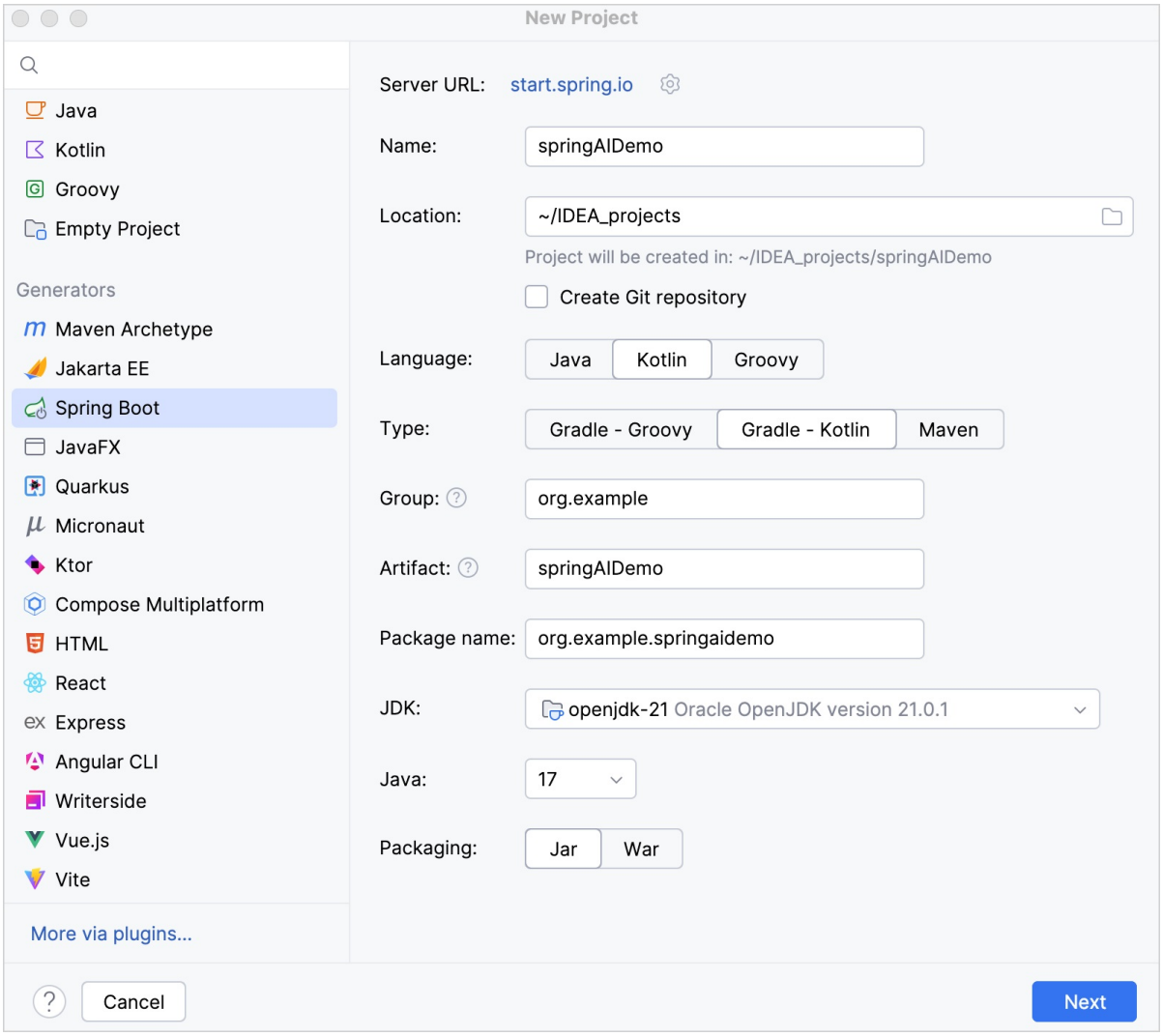
    - **Name**: springAIDemo
    - **Language**: Kotlin
    - **Type**: Gradle - Kotlin

        > This option specifies the build system and the DSL.
        >
        > 

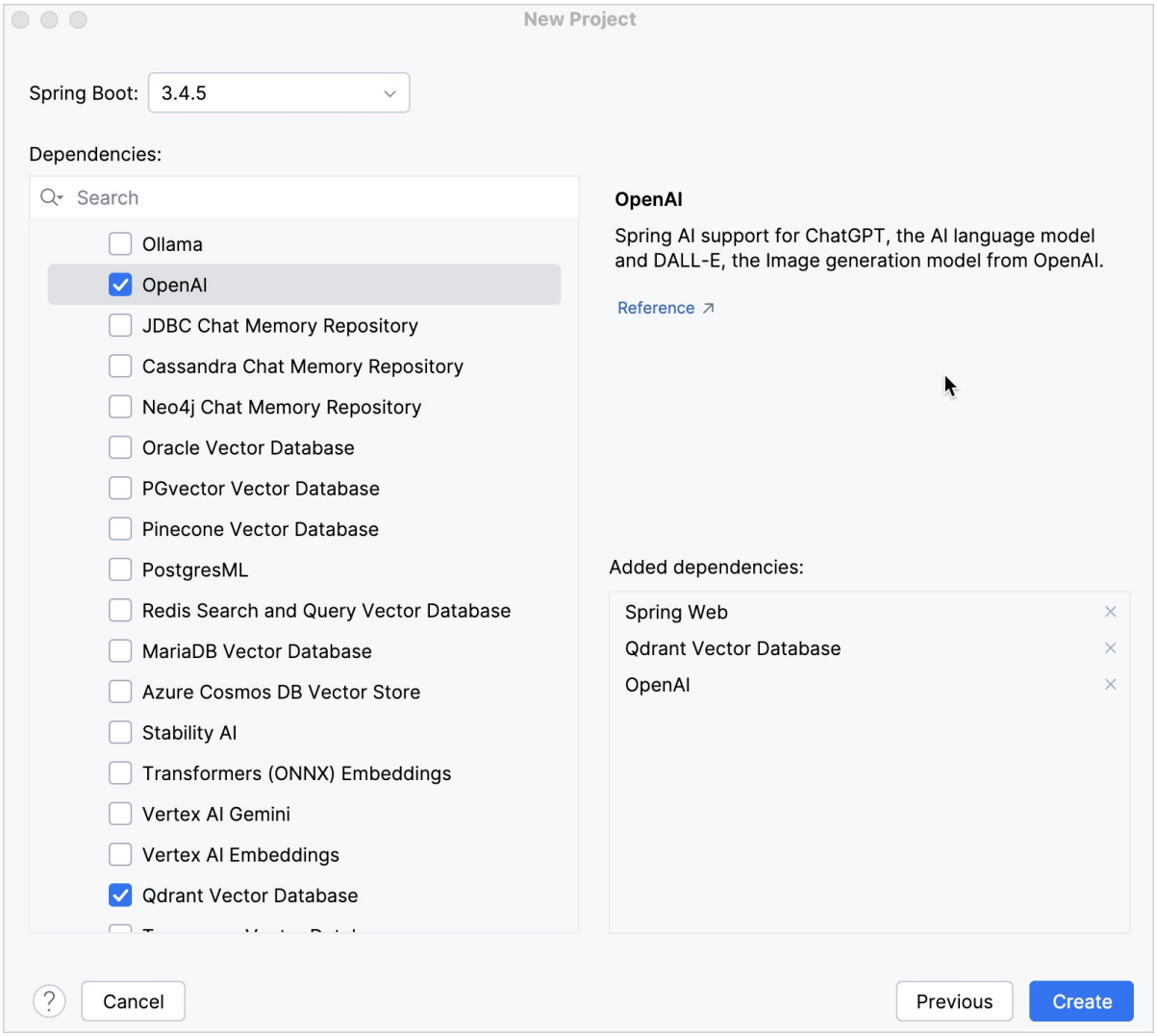
    - **Package name**: com.example.springaidemo
    - **JDK**: Java JDK

        > This tutorial uses **Oracle OpenJDK version 21.0.1**. If you don't have a JDK installed, you can download it from the dropdown list.
        >
        > 

    - **Java**: 17



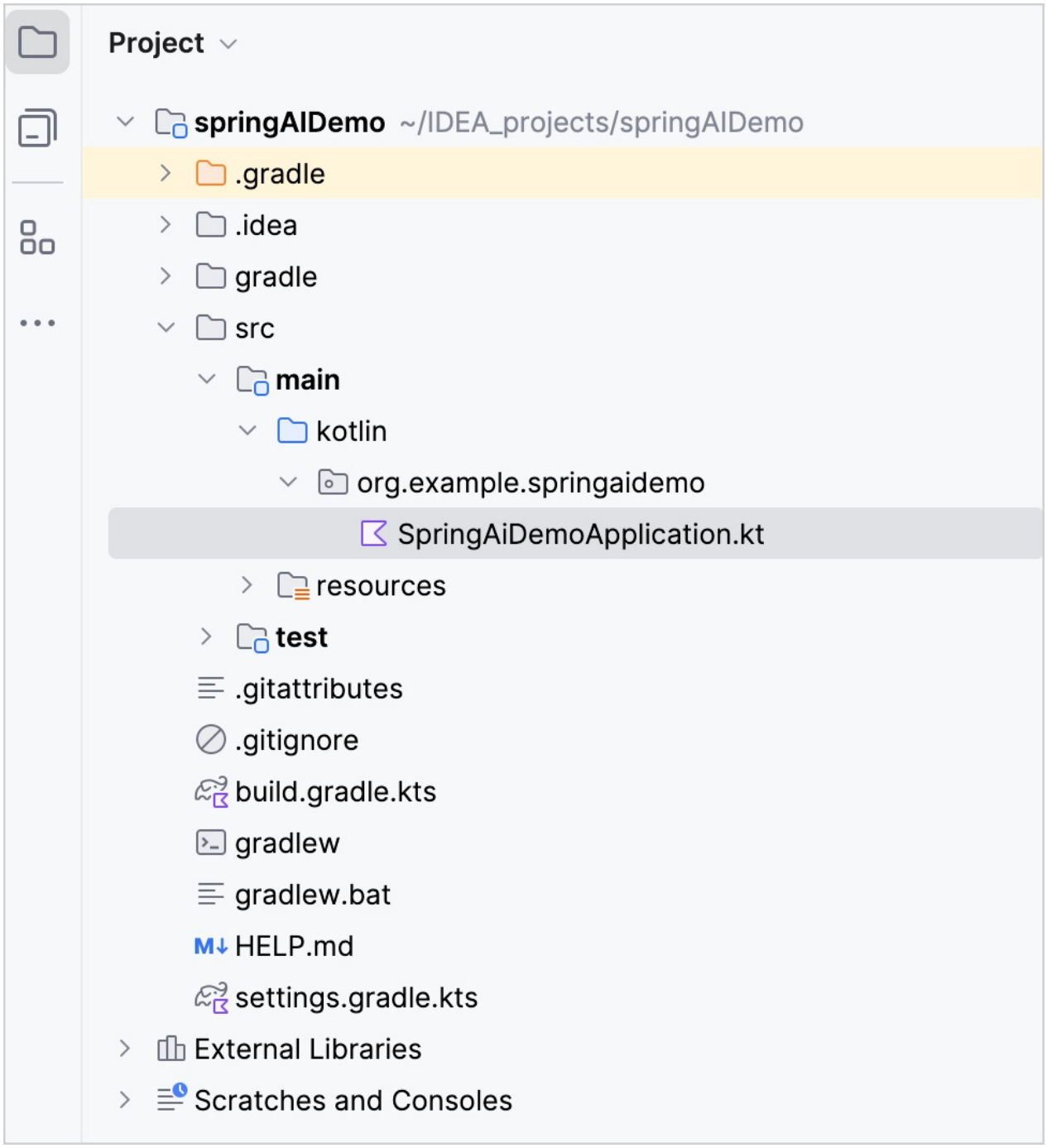
4. Make sure that you have specified all the fields and click **Next**.
5. Select the latest stable Spring Boot version in the **Spring Boot** field.
6. Select the following dependencies required for this tutorial:
    - **Web | Spring Web**
    - **AI | OpenAI**
    - **SQL | Qdrant Vector Database**



7. Click **Create** to generate and set up the project.

> The IDE will generate and open a new project. It may take some time to download and import the project dependencies.

After this, you can see the following structure in the **Project view**:



The generated Gradle project corresponds to the Maven's standard directory layout:

- There are packages and classes under the `main/kotlin` folder that belong to the application.
- The entry point to the application is the `main()` method of the `SpringAiDemoApplication.kt` file.

更新项目配置

1. Update your `build.gradle.kts` Gradle build file with the following:

```
plugins {
```

```
        kotlin("jvm") version "2.1.21"
        kotlin("plugin.spring") version "2.1.21"
        // Rest of the plugins
    }
```

2. Update your `springAiVersion` to `1.0.0-M6`:

```
extra["springAiVersion"] = "1.0.0-M6"
```

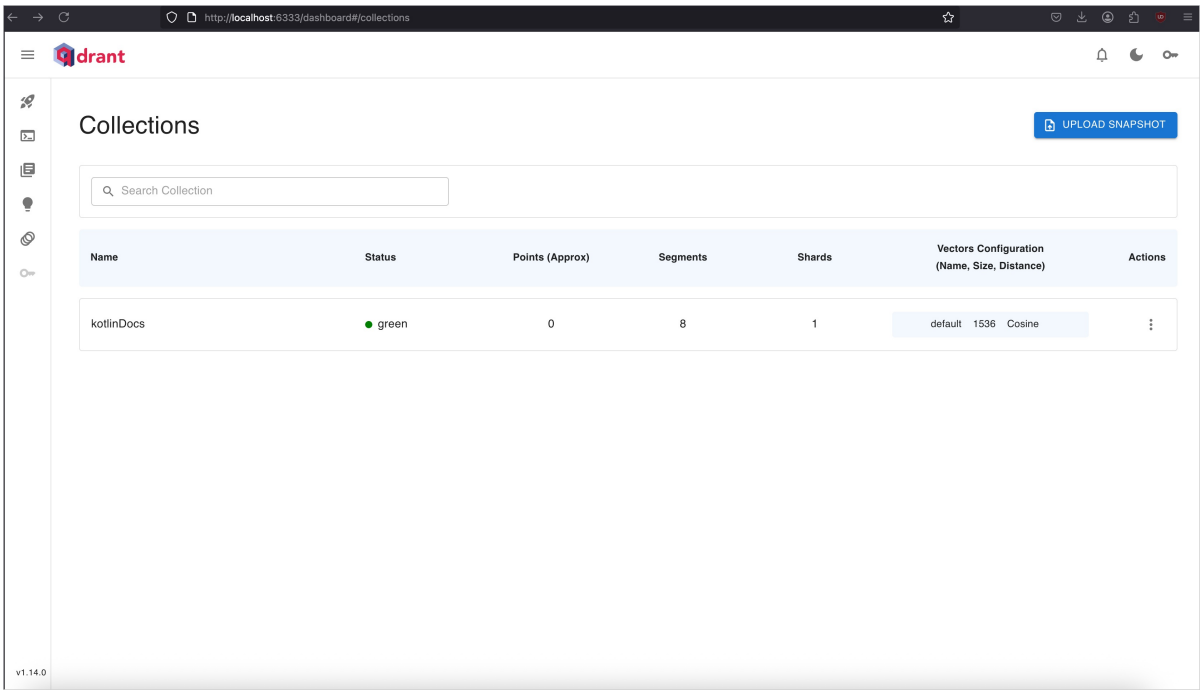
3. Click the **Sync Gradle Changes** button to synchronize the Gradle files.

4. Update your `src/main/resources/application.properties` file with the following:

```
# OpenAI
spring.ai.openai.api-key=YOUR_OPENAI_API_KEY
spring.ai.openai.chat.options.model=gpt-4o-mini
spring.ai.openai.embedding.options.model=text-embedding-ada-002
# Qdrant
spring.ai.vectorstore.qdrant.host=localhost
spring.ai.vectorstore.qdrant.port=6334
spring.ai.vectorstore.qdrant.collection-name=kotlinDocs
spring.ai.vectorstore.qdrant.initialize-schema=true
```

> Set your OpenAI API key to the `spring.ai.openai.api-key` property.
>
> 

5. Run the `SpringAiDemoApplication.kt` file to start the Spring Boot application. Once it's running, open the Qdrant collections page in your browser to see the result:



创建一个 controller 来加载并搜索文档

Create a Spring `@RestController` to search documents and store them in the Qdrant collection:

1. In the `src/main/kotlin/org/example/springaidemo` directory, create a new file named `KotlinSTDController.kt`, and add the following code:

```
package org.example.springaidemo

// Imports the required Spring and utility classes
import org.slf4j.LoggerFactory
import org.springframework.ai.document.Document
import org.springframework.ai.vectorstore.SearchRequest
import org.springframework.ai.vectorstore.VectorStore
import org.springframework.web.bind.annotation.GetMapping
import org.springframework.web.bind.annotation.PostMapping
import org.springframework.web.bind.annotation.RequestMapping
import org.springframework.web.bind.annotation.RequestParam
import org.springframework.web.bind.annotation.RestController
import org.springframework.web.client.RestTemplate
import kotlin.uuid.ExperimentalUuidApi
import kotlin.uuid.Uuid

// Data class representing the chat request payload
data class ChatRequest(val query: String, val topK: Int = 3)

@RestController
@RequestMapping("/kotlin")
class KotlinSTDController(
private val restTemplate: RestTemplate,
private val vectorStore: VectorStore,
) {
private val logger = LoggerFactory.getLogger(this::class.java)

    @OptIn(ExperimentalUuidApi::class)
    @PostMapping("/load-docs")
    fun load() {
        // Loads a list of documents from the Kotlin documentation
        val kotlinStdTopics = listOf(
            "collections-overview", "constructing-collections", "iterators", "ran
ges", "sequences",
            "collection-operations", "collection-transformations", "collection-fi
ltering", "collection-plus-minus",
            "collection-grouping", "collection-parts", "collection-elements", "co
llection-ordering",
            "collection-aggregate", "collection-write", "list-operations", "set-o
perations",
            "map-operations", "read-standard-input", "opt-in-requirements", "scop
e-functions", "time-measurement",
        )
        // Base URL for the documents
```

```
        val url = "https://raw.githubusercontent.com/JetBrains/kotlin-web-site/re
fs/heads/master/docs/topics/"
        // Retrieves each document from the URL and adds it to the vector store
        kotlinStdTopics.forEach { topic ->
            val data = restTemplate.getForObject("$url$topic.md", String::class.j
ava)
            data?.let { it ->
                val doc = Document.builder()
                    // Builds a document with a random UUID
                    .id(Uuid.random().toString())
                    .text(it)
                    .metadata("topic", topic)
                    .build()
                vectorStore.add(listOf(doc))
                logger.info("Document $topic loaded.")
            } ?: logger.warn("Failed to load document for topic: $topic")
        }
    }

    @GetMapping("docs")
    fun query(
        @RequestParam query: String = "operations, filtering, and transformations"
,
        @RequestParam topK: Int = 2
    ): List<Document>? {
        val searchRequest = SearchRequest.builder()
            .query(query)
            .topK(topK)
            .build()
        val results = vectorStore.similaritySearch(searchRequest)
        logger.info("Found ${results?.size ?: 0} documents for query: '$query'")
        return results
    }
}
```

{collapsible="true"}

2. Update the `SpringAiDemoApplication.kt` file to declare a `RestTemplate` bean:

```
package org.example.springaidemo

import org.springframework.boot.autoconfigure.SpringBootApplication
import org.springframework.boot.runApplication
import org.springframework.context.annotation.Bean
import org.springframework.web.client.RestTemplate

@SpringBootApplication
class SpringAiDemoApplication {

    @Bean
    fun restTemplate(): RestTemplate = RestTemplate()
```

```
}

fun main(args: Array<String>) {
    runApplication<SpringAiDemoApplication>(*args)
}
```

{collapsible="true"}

3. Run the application.
4. In the terminal, send a POST request to the `/kotlin/load-docs` endpoint to load the documents:

```
curl -X POST http://localhost:8080/kotlin/load-docs
```

5. Once the documents are loaded, you can search for them with a GET request:

```
curl -X GET http://localhost:8080/kotlin/docs
```



> You can also view the results on the Qdrant collections page.

实现 AI 聊天端点

Once the documents are loaded, the final step is to add an endpoint that answers questions using the documents in Qdrant through Spring AI's Retrieval-Augmented Generation (RAG) support:

1. Open the `KotlinSTDController.kt` file, and import the following classes:

```
import org.springframework.ai.chat.client.ChatClient
import org.springframework.ai.chat.client.advisor.RetrievalAugmentationAdvisor
import org.springframework.ai.chat.client.advisor.SimpleLoggerAdvisor
import org.springframework.ai.chat.prompt.Prompt
import org.springframework.ai.chat.prompt.PromptTemplate
import org.springframework.ai.rag.preretrieval.query.transformation.RewriteQueryTransformer
```

```
import org.springframework.ai.rag.retrieval.search.VectorStoreDocumentRetriever
import org.springframework.web.bind.annotation.RequestBody
```

2. Add `ChatClient.Builder` to the controller's constructor parameters:

```
class KotlinSTDController(
    private val chatClientBuilder: ChatClient.Builder,
    private val restTemplate: RestTemplate,
    private val vectorStore: VectorStore,
)
```

3. Inside the controller class, create a `ChatClient` instance and a query transformer:

```
// Builds the chat client with a simple logging advisor
private val chatClient = chatClientBuilder.defaultAdvisors(SimpleLoggerAdvisor()).
build()
// Builds the query transformer used to rewrite the input query
private val rqtBuilder = RewriteQueryTransformer.builder().chatClientBuilder(chatC
lientBuilder)
```

4. At the bottom of your `KotlinSTDController.kt` file, add a new `chatAsk()` endpoint, with the following logic:

```
    @PostMapping("/chat/ask")
    fun chatAsk(@RequestBody request: ChatRequest): String? {
        // Defines the prompt template with placeholders
        val promptTemplate = PromptTemplate(
            """
            {query}.
            Please provide a concise answer based on the {target} documentation.
        """.trimIndent()
        )

        // Creates the prompt by substituting placeholders with actual values
        val prompt: Prompt =
            promptTemplate.create(mapOf("query" to request.query, "target" to "Kot
lin standard library"))

        // Configures the retrieval advisor to augment the query with relevant doc
uments
        val retrievalAdvisor = RetrievalAugmentationAdvisor.builder()
            .documentRetriever(
                VectorStoreDocumentRetriever.builder()
                    .similarityThreshold(0.7)
                    .topK(request.topK)
                    .vectorStore(vectorStore)
                    .build()
            )
```

```
            .queryTransformers(rqtBuilder.promptTemplate(promptTemplate).build())
            .build()

        // Sends the prompt to the LLM with the retrieval advisor and get the resp
onse
        val response = chatClient.prompt(prompt)
            .advisors(retrievalAdvisor)
            .call()
            .content()
        logger.info("Chat response generated for query: '${request.query}'")
        return response
    }
```

5. Run the application.
6. In the terminal, send a POST request to the new endpoint to see the results:

```
curl -X POST "http://localhost:8080/kotlin/chat/ask" \
     -H "Content-Type: application/json" \
     -d '{"query": "What are the performance implications of using lazy sequences
in Kotlin for large datasets?", "topK": 3}'
```



Congratulations! You now have a Kotlin app that connects to OpenAI and answers questions using context retrieved from documentation stored in Qdrant. Try experimenting with different queries or importing other documents to explore more possibilities.

You can view the completed project in the Spring AI demo GitHub repository, or explore other Spring AI examples in Kotlin AI Examples.

Spring 框架 Kotlin 文档

Spring 框架 Kotlin 文档

使用 Spring Boot 与 Kotlin 构建 web 应用程序——教程

使用 Spring Boot 与 Kotlin 构建 web 应用程序——教程

使用 Kotlin 协程与 RSocket 创建聊天应用程序——教程

使用 Kotlin 协程与 RSocket 创建聊天应用程序——教程

在 JVM 平台中用 JUnit 测试代码——教程

This tutorial shows you how to write a simple unit test in a Kotlin/JVM project and run it with the Gradle build tool.

In this project, you'll use the `kotlin.test` library and run the test using JUnit. If you're working on a multiplatform app, see the Kotlin Multiplatform tutorial.

To get started, first download and install the latest version of IntelliJ IDEA.

Add dependencies

1. Open a Kotlin project in IntelliJ IDEA. If you don't have a project, create one.
2. Open the `build.gradle(.kts)` file and check that the `testImplementation` dependency is present. This dependency allows you to work with `kotlin.test` and `JUnit`:

> 【Kotlin】

```
dependencies {
    // Other dependencies.
    testImplementation(kotlin("test"))
}
```

> 【Groovy】

```
dependencies {
    // Other dependencies.
    testImplementation 'org.jetbrains.kotlin:kotlin-test'
}
```

1. Add the `test` task to the `build.gradle(.kts)` file:

> 【Kotlin】

```
    tasks.test {
        useJUnitPlatform()
    }
```

> 【Groovy】

```
    test {
        useJUnitPlatform()
    }
```

> If you use the `useJUnitPlatform()` function in your build script, the `kotlin-test` library automatically includes JUnit 5 as a dependency. This setup enables access to all JUnit 5 APIs, along with the `kotlin-test` API, in JVM-only projects and JVM tests of Kotlin Multiplatform (KMP) projects.
>
> 

Here's a complete code for the `build.gradle.kts`:

```
plugins {
    kotlin("jvm") version "2.1.21"
}

group = "org.example"
version = "1.0-SNAPSHOT"

repositories {
    mavenCentral()
}

dependencies {
    testImplementation(kotlin("test"))
}

tasks.test {
    useJUnitPlatform()
}
```

Add the code to test it

1. Open the `Main.kt` file in `src/main/kotlin`.

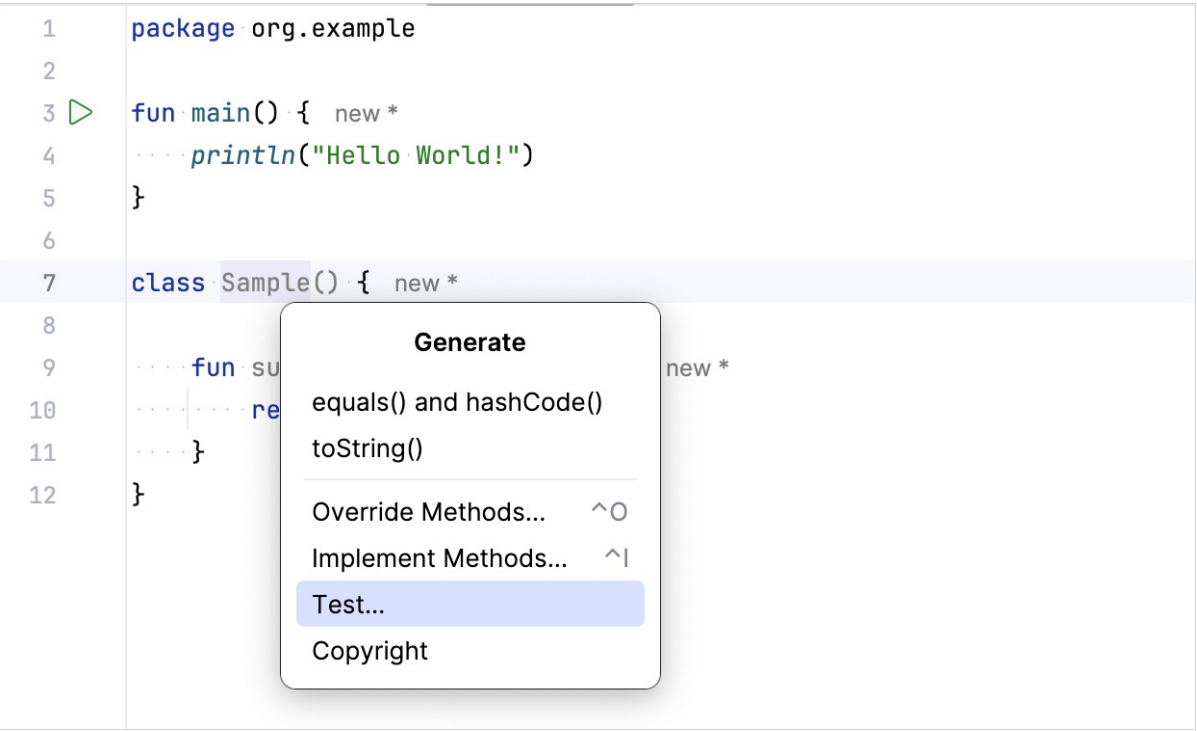
The `src` directory contains Kotlin source files and resources. The `Main.kt` file contains sample code that prints `Hello, World!`.

2. Create the `Sample` class with the `sum()` function that adds two integers together:

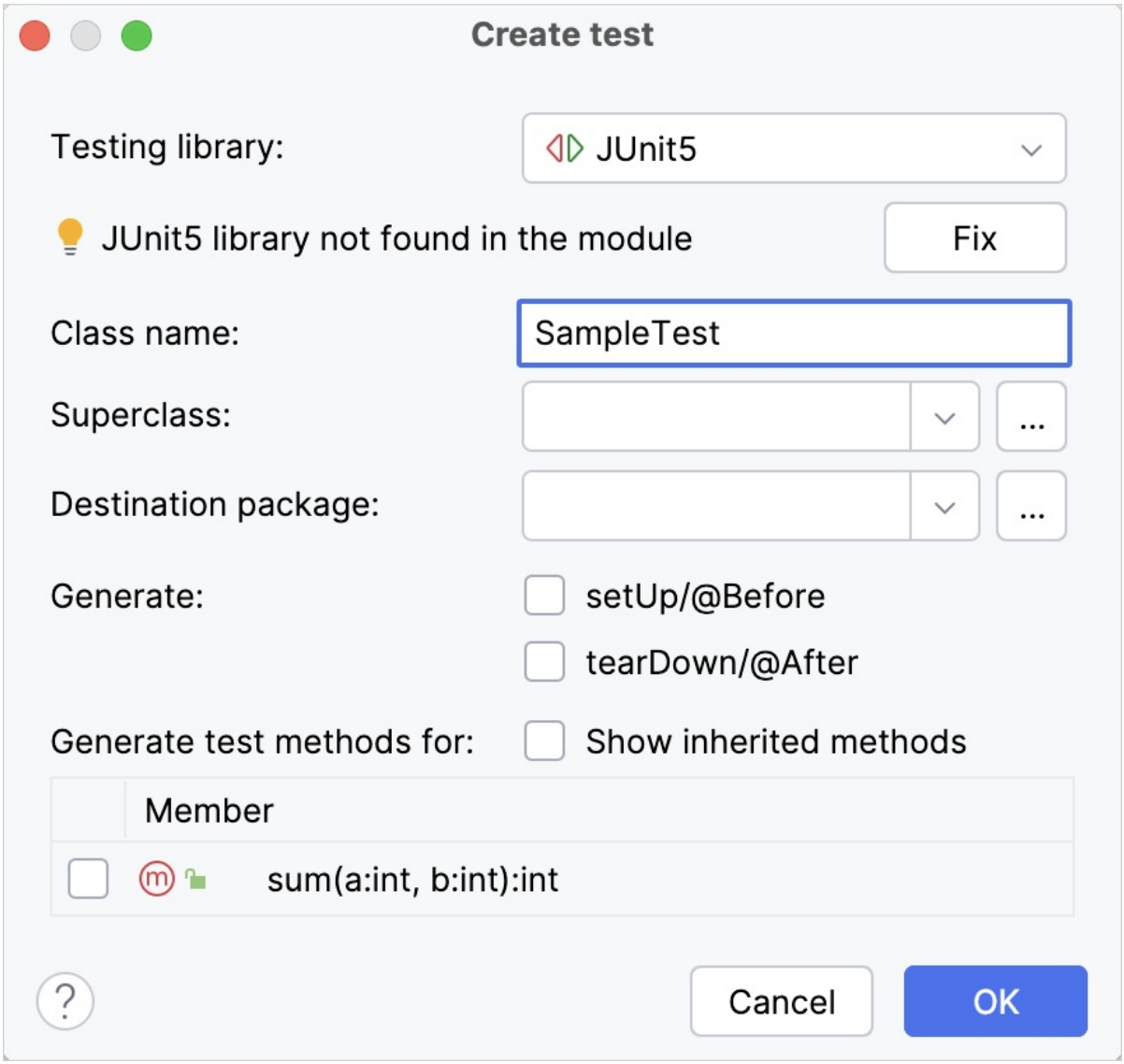
```
class Sample() {
    fun sum(a: Int, b: Int): Int {
        return a + b
    }
}
```

Create a test

1. In IntelliJ IDEA, select **Code** | **Generate** | **Test...** for the `Sample` class:



2. Specify the name of the test class. For example, `SampleTest`:



IntelliJ IDEA creates the `SampleTest.kt` file in the `test` directory. This directory contains Kotlin test source files and resources.

> You can also manually create a `*.kt` file for tests in `src/test/kotlin`.

3. Add the test code for the `sum()` function in `SampleTest.kt`:

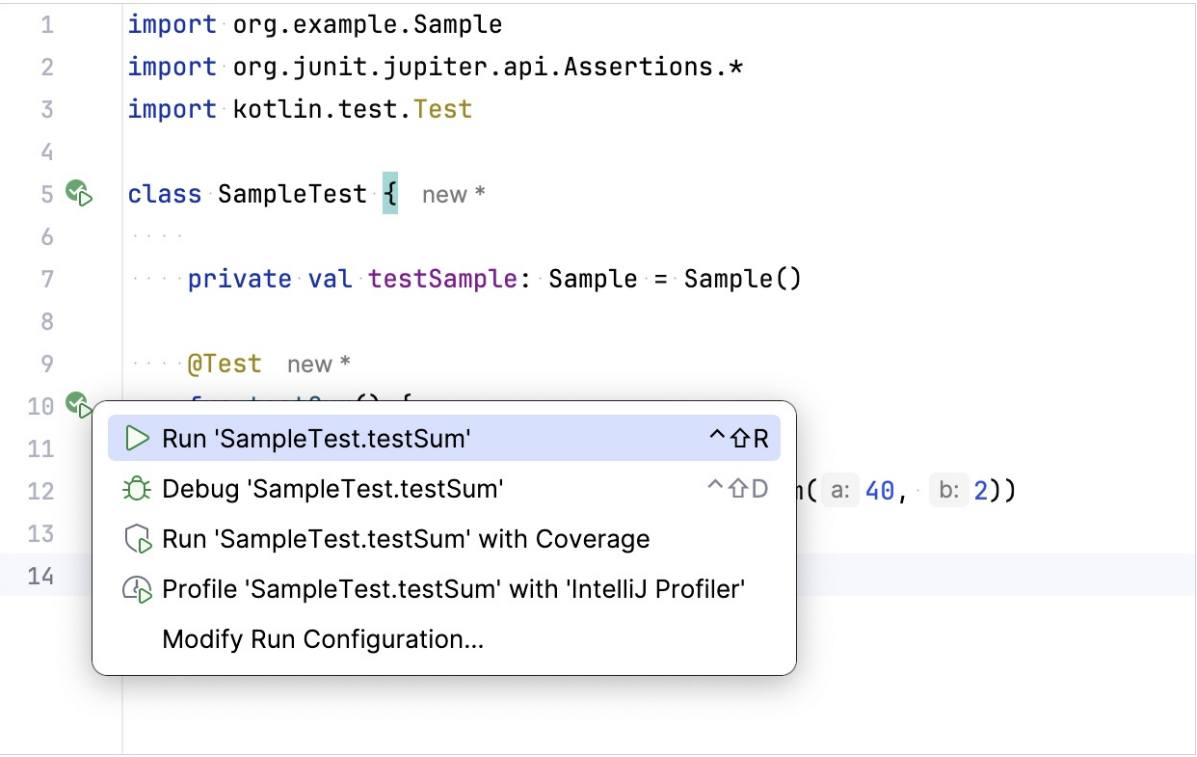
   - Define the test `testSum()` function using the `@Test` annotation.
   - Check that the `sum()` function returns the expected value by using the `assertEquals()` function.

```
import org.example.Sample
import org.junit.jupiter.api.Assertions.*
import kotlin.test.Test

class SampleTest {
    private val testSample: Sample = Sample()
```

```
    @Test
    fun testSum() {
        val expected = 42
        assertEquals(expected, testSample.sum(40, 2))
    }
}
```

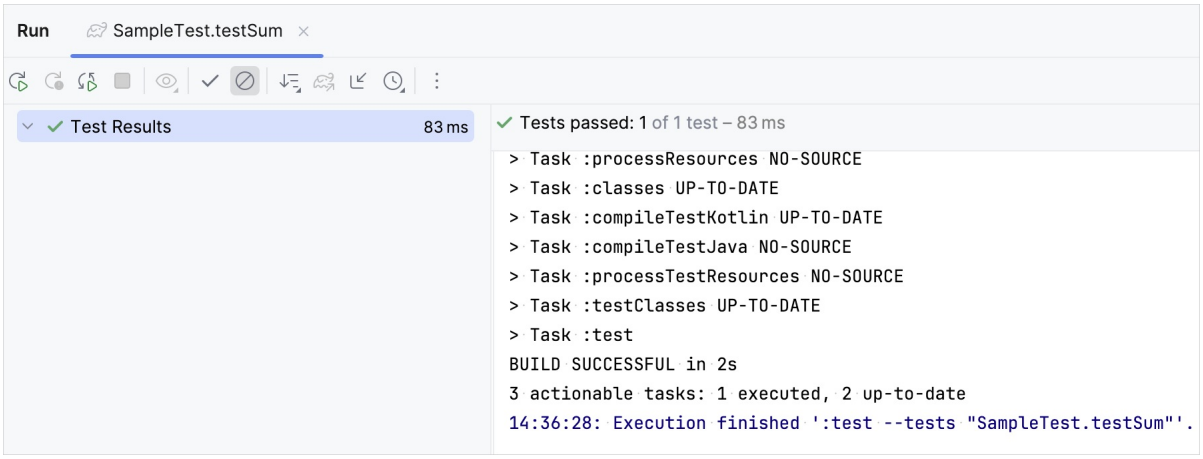
Run a test

1. Run the test using the gutter icon:



> You can also run all project tests via the command-line interface using the `./gradlew check` command.
>
> 

2. Check the result in the **Run** tool window:

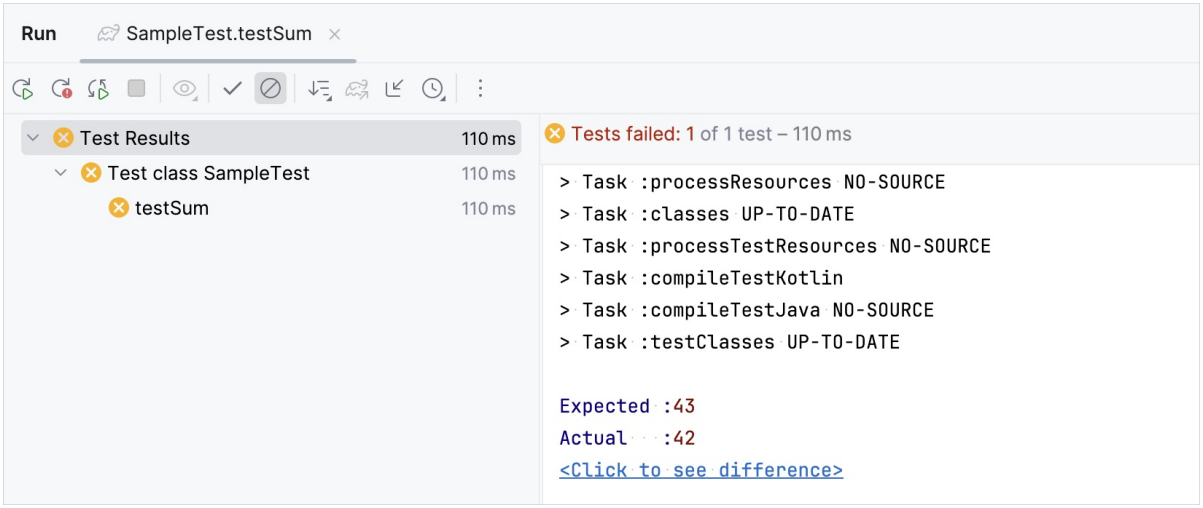


The test function was executed successfully.

3. Make sure that the test works correctly by changing the `expected` variable value to 43:

```
@Test
fun testSum() {
    val expected = 43
    assertEquals(expected, classForTesting.sum(40, 2))
}
```

4. Run the test again and check the result:



The test execution failed.

下一步做什么

Once you've finished your first test, you can:

- Write more tests using other `kotlin.test` functions. For example, use the `assertNotEquals()` function.
- Improve your test output with the Kotlin Power-assert compiler plugin. The plugin enriches the

test output with contextual information.

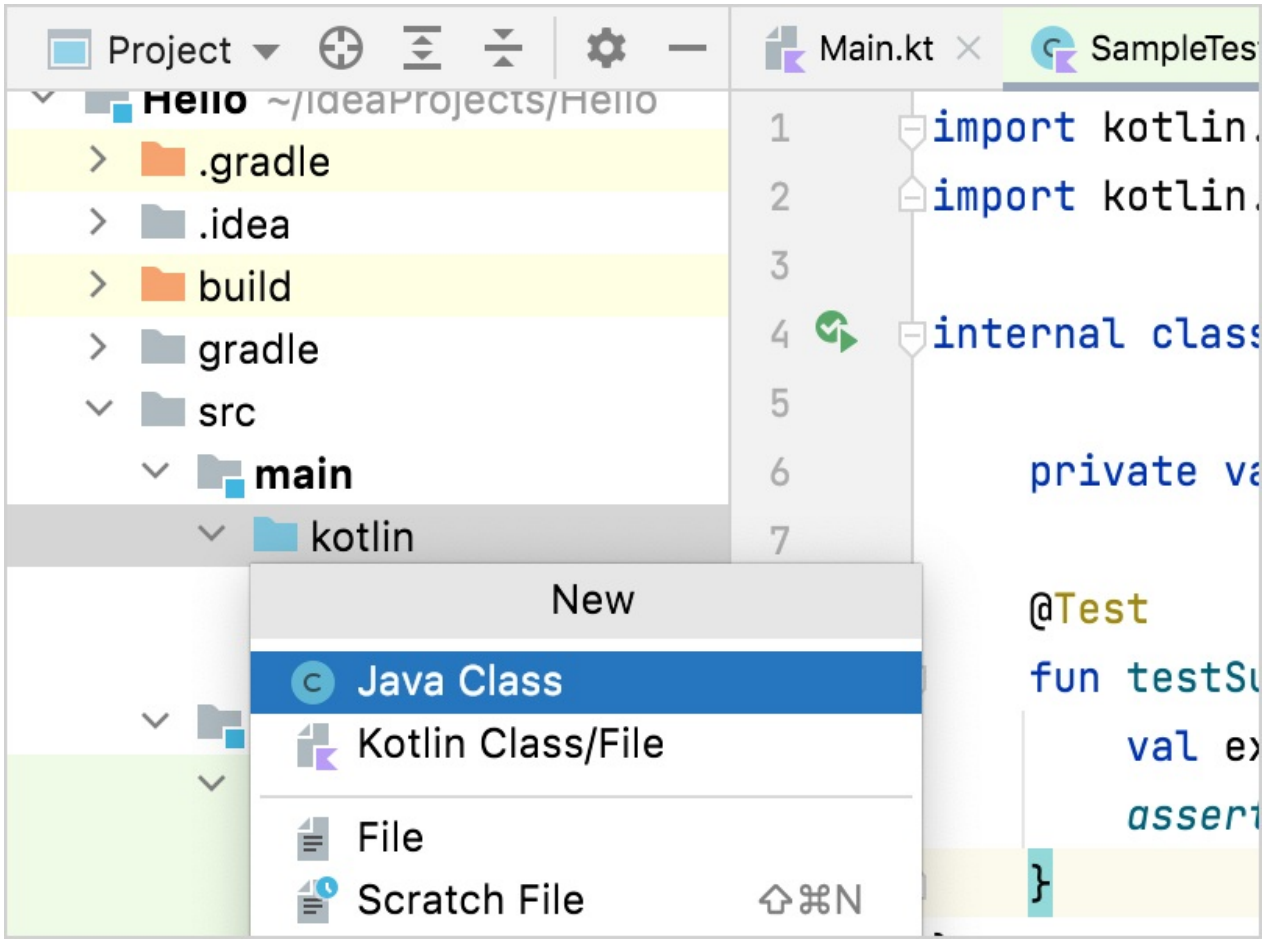
- Create your first server-side application with Kotlin and Spring Boot.

在项目中混用 Java 与 Kotlin——教程

Kotlin provides the first-class interoperability with Java, and modern IDEs make it even better. In this tutorial, you'll learn how to use both Kotlin and Java sources in the same project in IntelliJ IDEA. To learn how to start a new Kotlin project in IntelliJ IDEA, see Getting started with IntelliJ IDEA.

将 Java 源代码添加到现有 Kotlin 项目中

将 Java 类添加到 Kotlin 项目中非常简单。 你需要做的就是在项目内的目录或包中创建一个新的 Java 文件 and go to **File** | **New** | **Java Class** or use the **Alt** + **Insert**/**Cmd** + **N** shortcut.



如果你已经有 Java 类，可以将它们复制到项目目录中。

现在你可以在 Kotlin 中使用这个 Java 类，反之亦然，无需任何进一步的操作。

例如，添加以下 Java 类：

```
public class Customer {

    private String name;
```

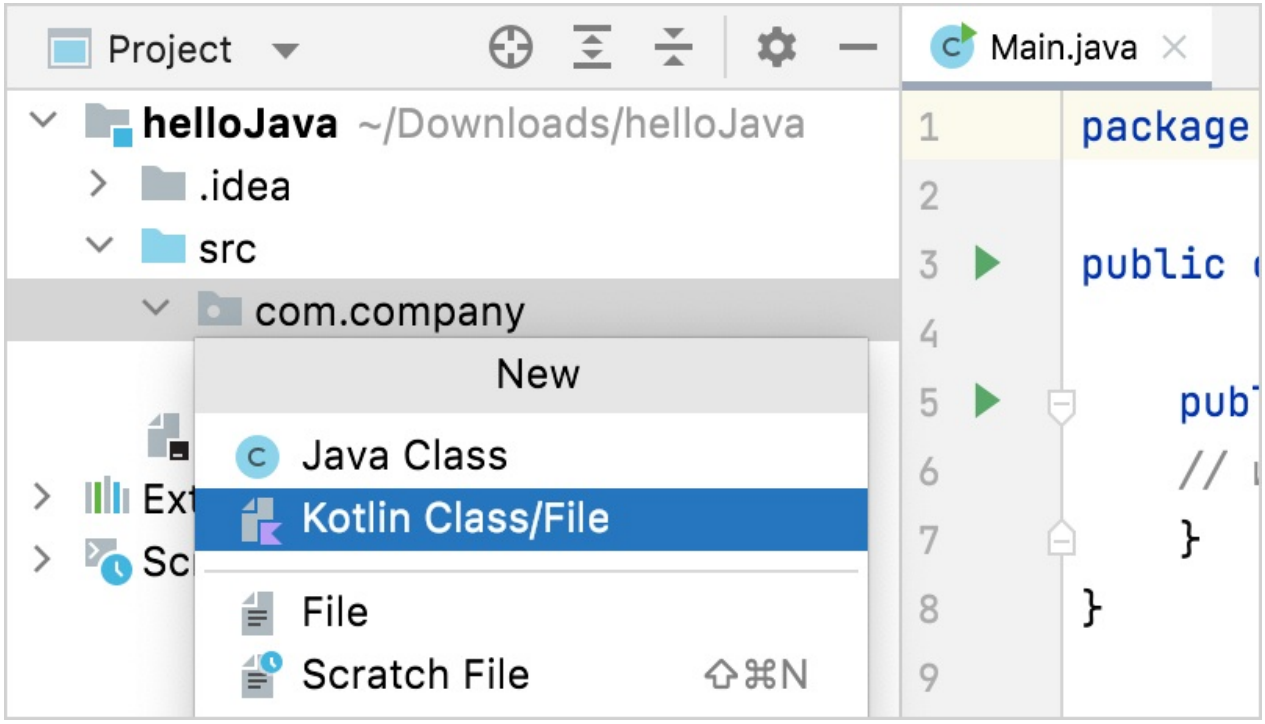
```
    public Customer(String s){
        name = s;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public void placeOrder() {
        System.out.println("A new order is placed by " + name);
    }
}
```

我们可以在 Kotlin 中像使用任何其他类型一样调用它。

```
val customer = Customer("Phase")
println(customer.name)
println(customer.placeOrder())
```

将 Kotlin 源代码添加到现有 Java 项目中

将 Kotlin 文件添加到现有 Java 项目的过程几乎相同。



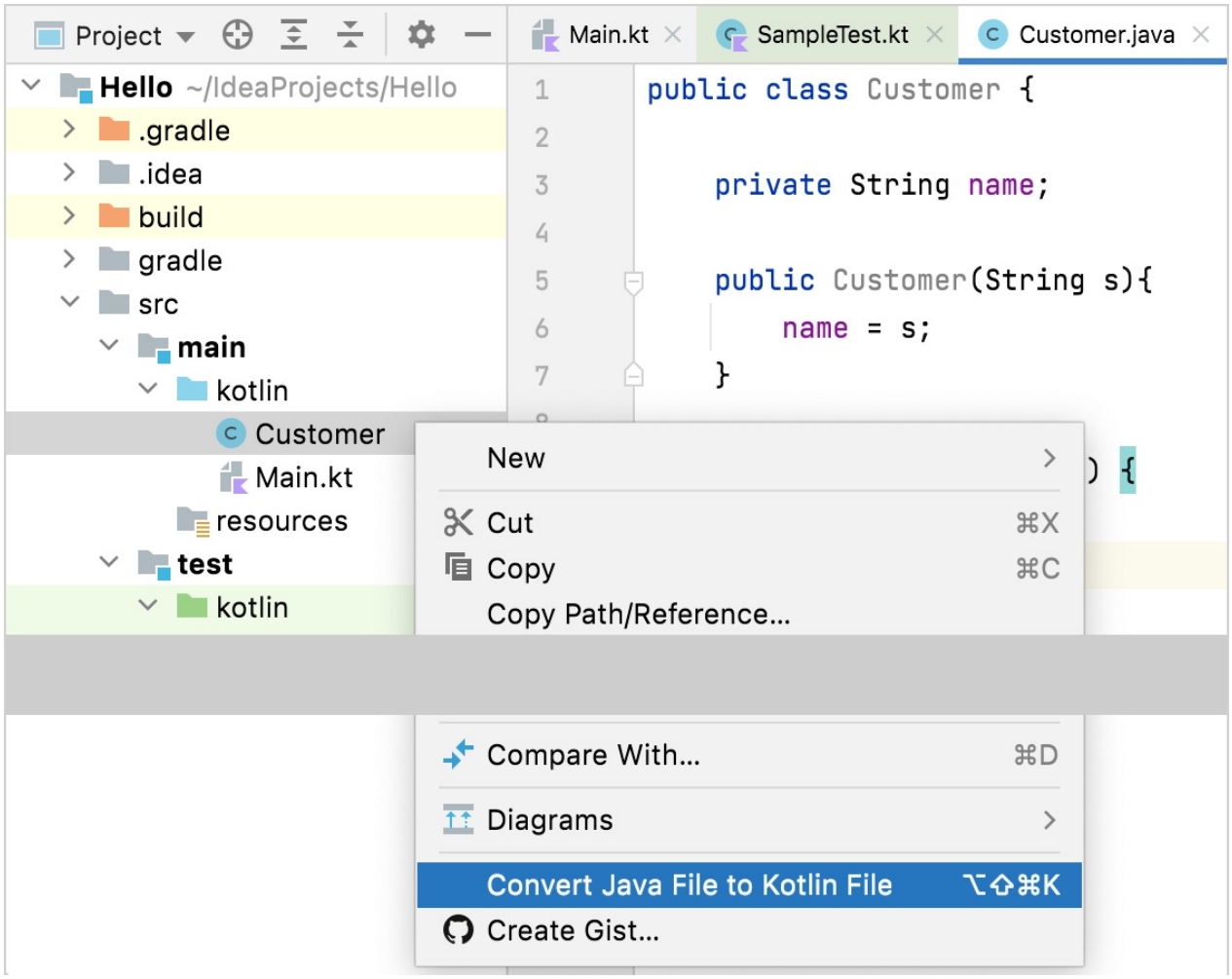
如果这是你第一次将 Kotlin 文件添加到此项目中，IntelliJ IDEA 会自动添加所需的 Kotlin 运行时。



你也可以在 **Tools** | **Kotlin** | **Configure Kotlin in Project** 中手动打开 Kotlin 运行时配置。

使用 J2K 将现有 Java 文件转换为 Kotlin 文件

Kotlin 插件还附带了 Java 到 Kotlin 的转换器 (*J2K*)，自动把 Java 文件转换为 Kotlin 文件。 要在文件上使用 J2K，请在该文件的上下文菜单或 IntelliJ IDEA 的 **Code** 菜单中点击 **Convert Java File to Kotlin File**。



虽然转换器不是完美的，但它在把大多数样板代码从 Java 转换为 Kotlin 方面做得相当不错。但有时需要进行一些手动改进。

在 Kotlin 中使用 Java 记录类型

Records are classes in Java for storing immutable data. Records carry a fixed set of values – the *records components*. They have a concise syntax in Java and save you from having to write boilerplate code:

```
// Java
public record Person (String name, int age) {}
```

The compiler automatically generates a final class inherited from `java.lang.Record` with the following members:

- a private final field for each record component
- a public constructor with parameters for all fields
- a set of methods to implement structural equality: `equals()`, `hashCode()`, `toString()`
- a public method for reading each record component

Records are very similar to Kotlin data classes.

Using Java records from Kotlin code

You can use record classes with components that are declared in Java the same way you would use classes with properties in Kotlin. To access the record component, just use its name like you do for Kotlin properties:

```
val newPerson = Person("Kotlin", 10)
val firstName = newPerson.name
```

Declare records in Kotlin

Kotlin supports record declaration only for data classes, and the data class must meet the requirements.

To declare a record class in Kotlin, use the `@JvmRecord` annotation:

> Applying `@JvmRecord` to an existing class is not a binary compatible change. It alters the naming convention of the class property accessors.
>
> 

```
@JvmRecord
data class Person(val name: String, val age: Int)
```

This JVM-specific annotation enables generating:

- the record components corresponding to the class properties in the class file
- the property accessor methods named according to the Java record naming convention

The data class provides `equals()`, `hashCode()`, and `toString()` method implementations.

Requirements

To declare a data class with the `@JvmRecord` annotation, it must meet the following requirements:

- The class must be in a module that targets JVM 16 bytecode (or 15 if the `-Xjvm-enable-preview` compiler option is enabled).
- The class cannot explicitly inherit any other class (including `Any`) because all JVM records implicitly inherit `java.lang.Record`. However, the class can implement interfaces.
- The class cannot declare any properties with backing fields – except those initialized from the corresponding primary constructor parameters.
- The class cannot declare any mutable properties with backing fields.
- The class cannot be local.
- The primary constructor of the class must be as visible as the class itself.

Enabling JVM records

JVM records require the `16` target version or higher of the generated JVM bytecode.

To specify it explicitly, use the `jvmTarget` compiler option in Gradle or Maven.

Further discussion

See this language proposal for JVM records for further technical details and discussion.

从 Java 到 Kotlin 迁移指南

- 字符串
- 集合
- 可空性
- 标准输入

Java 与 Kotlin 中的字符串

This guide contains examples of how to perform typical tasks with strings in Java and Kotlin. It will help you migrate from Java to Kotlin and write your code in the authentically Kotlin way.

字符串连接

In Java, you can do this in the following way:

```
// Java
String name = "Joe";
System.out.println("Hello, " + name);
System.out.println("Your name is " + name.length() + " characters long");
```

In Kotlin, use the dollar sign (`$`) before the variable name to interpolate the value of this variable into your string:

```
fun main() {
//sampleStart
    // Kotlin
    val name = "Joe"
    println("Hello, $name")
    println("Your name is ${name.length} characters long")
//sampleEnd
}
```

You can interpolate the value of a complicated expression by surrounding it with curly braces, like in `${name.length}` . See string templates for more information.

构建字符串

In Java, you can use the StringBuilder:

```
// Java
StringBuilder countDown = new StringBuilder();
for (int i = 5; i > 0; i--) {
    countDown.append(i);
    countDown.append("\n");
}
System.out.println(countDown);
```

In Kotlin, use buildString() – an inline function that takes logic to construct a string as a lambda argument:

```
fun main() {
//sampleStart
       // Kotlin
       val countDown = buildString {
           for (i in 5 downTo 1) {
               append(i)
               appendLine()
           }
       }
       println(countDown)
//sampleEnd
}
```

Under the hood, the `buildString` uses the same `StringBuilder` class as in Java, and you access it via an implicit `this` inside the lambda.

Learn more about lambda coding conventions.

由集合内元素创建字符串

In Java, you use the Stream API to filter, map, and then collect the items:

```
// Java
List<Integer> numbers = List.of(1, 2, 3, 4, 5, 6);
String invertedOddNumbers = numbers
        .stream()
        .filter(it -> it % 2 != 0)
        .map(it -> -it)
        .map(Object::toString)
        .collect(Collectors.joining("; "));
System.out.println(invertedOddNumbers);
```

In Kotlin, use the joinToString() function, which Kotlin defines for every List:

```
fun main() {
//sampleStart
    // Kotlin
    val numbers = listOf(1, 2, 3, 4, 5, 6)
    val invertedOddNumbers = numbers
        .filter { it % 2 != 0 }
        .joinToString(separator = ";") {"${-it}"}
    println(invertedOddNumbers)
//sampleEnd
}
```

> In Java, if you want spaces between your delimiters and following items, you need to add a space to the delimiter explicitly.
>
> 

Learn more about joinToString() usage.

如果字符串为空白就设置默认值

In Java, you can use the ternary operator:

```
// Java
public void defaultValueIfStringIsBlank() {
    String nameValue = getName();
    String name = nameValue.isBlank() ? "John Doe" : nameValue;
    System.out.println(name);
}

public String getName() {
    Random rand = new Random();
    return rand.nextBoolean() ? "" : "David";
}
```

Kotlin provides the inline function ifBlank() that accepts the default value as an argument:

```
// Kotlin
import kotlin.random.Random

//sampleStart
fun main() {
    val name = getName().ifBlank { "John Doe" }
    println(name)
}

fun getName(): String =
    if (Random.nextBoolean()) "" else "David"
//sampleEnd
```

替换字符串首尾处的（多个）字符

In Java, you can use the replaceAll()) function. The `replaceAll()` function in this case accepts regular expressions `^##` and `##$`, which define strings starting and ending with `##` respectively:

```
// Java
String input = "##place##holder##";
String result = input.replaceAll("^##|##$", "");
System.out.println(result);
```

In Kotlin, use the removeSurrounding() function with the string delimiter `##`:

```
fun main() {
//sampleStart
    // Kotlin
    val input = "##place##holder##"
    val result = input.removeSurrounding("##")
    println(result)
//sampleEnd
}
```

正则替换

In Java, you can use the Pattern and the Matcher classes, for example, to obfuscate some data:

```
// Java
String input = "login: Pokemon5, password: 1q2w3e4r5t";
Pattern pattern = Pattern.compile("\\w*\\d+\\w*");
Matcher matcher = pattern.matcher(input);
String replacementResult = matcher.replaceAll(it -> "xxx");
System.out.println("Initial input: '" + input + "'");
System.out.println("Anonymized input: '" + replacementResult + "'");
```

In Kotlin, you use the Regex class that simplifies working with regular expressions. Additionally, use raw strings to simplify a regex pattern by reducing the count of backslashes:

```
fun main() {
//sampleStart
    // Kotlin
    val regex = Regex("""\w*\d+\w*""") // multiline string
    val input = "login: Pokemon5, password: 1q2w3e4r5t"
    val replacementResult = regex.replace(input, replacement = "xxx")
    println("Initial input: '$input'")
    println("Anonymized input: '$replacementResult'")
//sampleEnd
}
```

字符串拆分

In Java, to split a string with the period character (`.`), you need to use shielding (`\\`). This happens because the split()) function of the `String` class accepts a regular expression as an argument:

```
// Java
System.out.println(Arrays.toString("Sometimes.text.should.be.split".split("\\.")));
```

In Kotlin, use the Kotlin function split(), which accepts varargs of delimiters as input parameters:

```
fun main() {
//sampleStart
    // Kotlin
    println("Sometimes.text.should.be.split".split("."))
//sampleEnd
}
```

If you need to split with a regular expression, use the overloaded `split()` version that accepts the `Regex` as a parameter.

取子串

In Java, you can use the substring()) function, which accepts an inclusive beginning index of a character to start taking the substring from. To take a substring after this character, you need to increment the index:

```
// Java
String input = "What is the answer to the Ultimate Question of Life, the Universe, and
 Everything? 42";
String answer = input.substring(input.indexOf("?") + 1);
System.out.println(answer);
```

In Kotlin, you use the substringAfter() function and don't need to calculate the index of the character you want to take a substring after:

```
fun main() {
//sampleStart
    // Kotlin
    val input = "What is the answer to the Ultimate Question of Life, the Universe, an
d Everything? 42"
    val answer = input.substringAfter("?")
    println(answer)
//sampleEnd
}
```

Additionally, you can take a substring after the last occurrence of a character:

```
fun main() {
//sampleStart
    // Kotlin
    val input = "To be, or not to be, that is the question."
    val question = input.substringAfterLast(",")
    println(question)
//sampleEnd
}
```

使用多行字符串

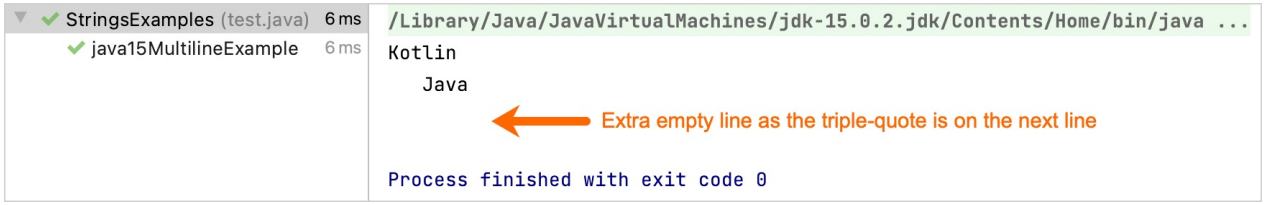
Before Java 15, there were several ways to create a multiline string. For example, using the join()) function of the `String` class:

```
// Java
String lineSeparator = System.getProperty("line.separator");
String result = String.join(lineSeparator,
        "Kotlin",
        "Java");
System.out.println(result);
```

In Java 15, text blocks appeared. There is one thing to keep in mind: if you print a multiline string and the triple-quote is on the next line, there will be an extra empty line:

```
// Java
String result = """
    Kotlin
       Java
    """;
System.out.println(result);
```

The output:

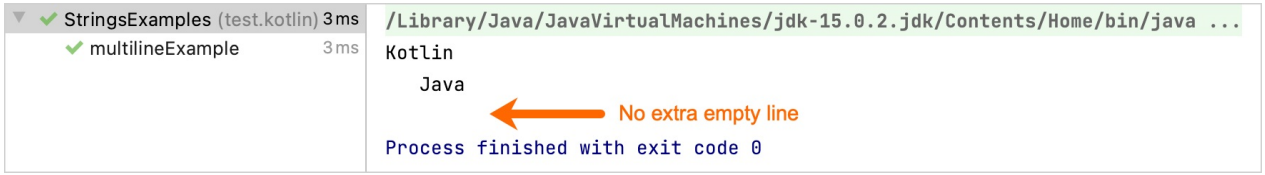


If you put the triple-quote on the same line as the last word, this difference in behavior disappears.

In Kotlin, you can format your line with the quotes on the new line, and there will be no extra empty line in the output. The left-most character of any line identifies the beginning of the line. The difference with Java is that Java automatically trims indents, and in Kotlin you should do it explicitly:

```
fun main() {
//sampleStart
    // Kotlin
    val result = """
        Kotlin
           Java
    """.trimIndent()
    println(result)
//sampleEnd
}
```

The output:



To have an extra empty line, you should add this empty line to your multiline string explicitly.

In Kotlin, you can also use the trimMargin() function to customize the indents:

```
// Kotlin
fun main() {
    val result = """
       #  Kotlin
       #  Java
   """.trimMargin("#")
    println(result)
}
```

Learn more about multiline strings.

下一步做什么？

- Look through other Kotlin idioms.
- Learn how to convert existing Java code to Kotlin with the Java to Kotlin converter.

If you have a favorite idiom, we invite you to share it by sending a pull request.

Java 与 Kotlin 中的集合

Collections are groups of a variable number of items (possibly zero) that are significant to the problem being solved and are commonly operated on. This guide explains and compares collection concepts and operations in Java and Kotlin. It will help you migrate from Java to Kotlin and write your code in the authentically Kotlin way.

The first part of this guide contains a quick glossary of operations on the same collections in Java and Kotlin. It is divided into operations that are the same and operations that exist only in Kotlin. The second part of the guide, starting from Mutability, explains some of the differences by looking at specific cases.

For an introduction to collections, see the Collections overview or watch this video by Sebastian Aigner, Kotlin Developer Advocate.

> All of the examples below use Java and Kotlin standard library APIs only.
>
> 

在 Java 与 Kotlin 中相同的操作

In Kotlin, there are many operations on collections that look exactly the same as their counterparts in Java.

对 list、 set、 queue 与 deque 的操作

描述	共有操作	更多 Kotlin 替代方式
Add an element or elements	`add()`, `addAll()`	Use the `plusAssign ( += )` operator: `collection += element`, `collection += anotherCollection`.
Check whether a collection contains an element or elements	`contains()`, `containsAll()`	Use the `in` keyword to call `contains()` in the operator form: `element in collection`.
Check whether a collection is empty	`isEmpty()`	Use `isNotEmpty()` to check whether a collection is not empty.
Remove under a certain condition	`removeIf()`	
Leave only selected elements	`retainAll()`	

Remove all elements from a collection	`clear()`	
Get a stream from a collection	`stream()`	Kotlin has its own way to process streams: sequences and methods like `map()` and `filter()`.
Get an iterator from a collection	`iterator()`	

对 map 的操作

描述	共有操作	更多 Kotlin 替代方式
Add an element or elements	`put()`, `putAll()`, `putIfAbsent()`	In Kotlin, the assignment `map[key] = value` behaves the same as `put(key, value)`. Also, you may use the `plusAssign ( += )` operator: `map += Pair(key, value)` or `map += anotherMap`.
Replace an element or elements	`put()`, `replace()`, `replaceAll()`	Use the indexing operator `map[key] = value` instead of `put()` and `replace()`.
Get an element	`get()`	Use the indexing operator to get an element: `map[index]`.
Check whether a map contains an element or elements	`containsKey()`, `containsValue()`	Use the `in` keyword to call `contains()` in the operator form: `element in map`.
Check whether a map is empty	`isEmpty()`	Use `isNotEmpty()` to check whether a map is not empty.
Remove an element	`remove(key)`, `remove(key, value)`	Use the `minusAssign ( -= )` operator: `map -= key`.
Remove all elements from a map	`clear()`	
Get a stream from a map	`stream()` on entries, keys, or values	

仅针对 list 的操作

描述	共有操作	更多 Kotlin 替代方式
Get an index of an element	`indexOf()`	
Get the last index of		

an element		
Get an element	`get()`	Use the indexing operator to get an element: `list[index]`.
Take a sublist	`subList()`	
Replace an element or elements	`set()`, `replaceAll()`	Use the indexing operator instead of `set()`: `list[index] = value`.

略有不同的操作

对任意集合类型的操作

描述	Java	Kotlin
Get a collection's size	`size()`	`count()`, `size`
Get flat access to nested collection elements	`collectionOfCollections.forEach(flatCollection::addAll)` or `collectionOfCollections.stream().flatMap().collect()`	`flatten()` or `flatMap()`
Apply the given function to every element	`stream().map().collect()`	`map()`
Apply the provided operation to collection elements sequentially and return the accumulated result	`stream().reduce()`	`reduce()`, `fold()`
Group elements by a classifier and count them	`stream().collect(Collectors.groupingBy(classifier, counting()))`	`eachCount()`
Filter by a condition	`stream().filter().collect()`	`filter()`
Check		

Check whether collection elements satisfy a condition	`stream().noneMatch()`, `stream().anyMatch()`, `stream().allMatch()`	`none()`, `any()`, `all()`
Sort elements	`stream().sorted().collect()`	`sorted()`
Take the first N elements	`stream().limit(N).collect()`	`take(N)`
Take elements with a predicate	`stream().takeWhile().collect()`	`takeWhile()`
Skip the first N elements	`stream().skip(N).collect()`	`drop(N)`
Skip elements with a predicate	`stream().dropWhile().collect()`	`dropWhile()`
Build maps from collection elements and certain values associated with them	`stream().collect(toMap(keyMapper, valueMapper))`	`associate()`

To perform all of the operations listed above on maps, you first need to get an `entrySet` of a map.

对 list 的操作

描述	Java	Kotlin
Sort a list into natural order	`sort(null)`	`sort()`
Sort a list into descending order	`sort(comparator)`	`sortDescending()`
Remove an element from a list	`remove(index)`, `remove(element)`	`removeAt(index)`, `remove(element)` or `collection -= element`
Fill all elements of a list with a certain value	`Collections.fill()`	`fill()`

Get unique elements from a list	`stream().distinct().toList()`	`distinct()`

Java 标准库中不存在的操作

- `zip()`, `unzip()` – transform a collection.
- `aggregate()` – group by a condition.
- `takeLast()`, `takeLastWhile()`, `dropLast()`, `dropLastWhile()` – take or drop elements by a predicate.
- `slice()`, `chunked()`, `windowed()` – retrieve collection parts.
- Plus (`+`) and minus (`-`) operators – add or remove elements.

If you want to take a deep dive into `zip()`, `chunked()`, `windowed()`, and some other operations, watch this video by Sebastian Aigner about advanced collection operations in Kotlin:

YouTube 视频：Advanced Collection Operations

可变性

In Java, there are mutable collections:

```
// Java
// This list is mutable!
public List<Customer> getCustomers() { ... }
```

Partially mutable ones:

```
// Java
List<String> numbers = Arrays.asList("one", "two", "three", "four");
numbers.add("five"); // Fails in runtime with `UnsupportedOperationException`
```

And immutable ones:

```
// Java
```

```
// Java
List<String> numbers = new LinkedList<>();
// This list is immutable!
List<String> immutableCollection = Collections.unmodifiableList(numbers);
immutableCollection.add("five"); // Fails in runtime with `UnsupportedOperationExcepti
on`
```

If you write the last two pieces of code in IntelliJ IDEA, the IDE will warn you that you're trying to modify an immutable object. This code will compile and fail in runtime with `UnsupportedOperationException`. You can't tell whether a collection is mutable by looking at its type.

Unlike in Java, in Kotlin you explicitly declare mutable or read-only collections depending on your needs. If you try to modify a read-only collection, the code won't compile:

```
// Kotlin
val numbers = mutableListOf("one", "two", "three", "four")
numbers.add("five")            // This is OK
val immutableNumbers = listOf("one", "two")
//immutableNumbers.add("five") // Compilation error - Unresolved reference: add
```

Read more about immutability on the Kotlin coding conventions page.

协变性

In Java, you can't pass a collection with a descendant type to a function that takes a collection of the ancestor type. For example, if `Rectangle` extends `Shape`, you can't pass a collection of `Rectangle` elements to a function that takes a collection of `Shape` elements. To make the code compilable, use the `? extends Shape` type so the function can take collections with any inheritors of `Shape`:

```
// Java
class Shape {}

class Rectangle extends Shape {}

public void doSthWithShapes(List<? extends Shape> shapes) {
/* If using just List<Shape>, the code won't compile when calling
this function with the List<Rectangle> as the argument as below */
}

public void main() {
    var rectangles = List.of(new Rectangle(), new Rectangle());
    doSthWithShapes(rectangles);
}
```

In Kotlin, read-only collection types are covariant. This means that if a `Rectangle` class inherits from the `Shape` class, you can use the type `List<Rectangle>` anywhere the `List<Shape>` type is required. In other words, the collection types have the same subtyping relationship as the element types. Maps are covariant on the value type, but not on the key type. Mutable collections aren't covariant – this would lead to runtime failures.

```
// Kotlin
open class Shape(val name: String)

class Rectangle(private val rectangleName: String) : Shape(rectangleName)

fun doSthWithShapes(shapes: List<Shape>) {
    println("The shapes are: ${shapes.joinToString { it.name }}")
}

fun main() {
    val rectangles = listOf(Rectangle("rhombus"), Rectangle("parallelepiped"))
    doSthWithShapes(rectangles)
}
```

Read more about collection types here.

区间与数列

In Kotlin, you can create intervals using ranges. For example, `Version(1, 11)..Version(1, 30)` includes all of the versions from `1.11` to `1.30`. You can check that your version is in the range by using the `in` operator: `Version(0, 9) in versionRange`.

In Java, you need to manually check whether a `Version` fits both bounds:

```
// Java
class Version implements Comparable<Version> {

    int major;
    int minor;

    Version(int major, int minor) {
        this.major = major;
        this.minor = minor;
    }

    @Override
    public int compareTo(Version o) {
        if (this.major != o.major) {
            return this.major - o.major;
        }
        return this.minor - o.minor;
```

```
        }
        return this.minor - o.minor;
    }
}

public void compareVersions() {
    var minVersion = new Version(1, 11);
    var maxVersion = new Version(1, 31);

   System.out.println(
           versionIsInRange(new Version(0, 9), minVersion, maxVersion));
   System.out.println(
           versionIsInRange(new Version(1, 20), minVersion, maxVersion));
}

public Boolean versionIsInRange(Version versionToCheck, Version minVersion,
                                Version maxVersion) {
    return versionToCheck.compareTo(minVersion) >= 0
            && versionToCheck.compareTo(maxVersion) <= 0;
}
```

In Kotlin, you operate with a range as a whole object. You don't need to create two variables and compare a `Version` with them:

```
// Kotlin
class Version(val major: Int, val minor: Int): Comparable<Version> {
    override fun compareTo(other: Version): Int {
        if (this.major != other.major) {
            return this.major - other.major
        }
        return this.minor - other.minor
    }
}

fun main() {
    val versionRange = Version(1, 11)..Version(1, 30)

    println(Version(0, 9) in versionRange)
    println(Version(1, 20) in versionRange)
}
```

As soon as you need to exclude one of the bounds, like to check whether a version is greater than or equal to (`>=`) the minimum version and less than (`<`) the maximum version, these inclusive ranges won't help.

按照多个维度比较

In Java, to compare objects by several criteria, you may use the `comparing()` and `thenComparingX()` functions from the `Comparator` interface. For example, to compare people by their name and age:

```
class Person implements Comparable<Person> {
    String name;
    int age;

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }

    Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    @Override
    public String toString() {
        return this.name + " " + age;
    }
}

public void comparePersons() {
    var persons = List.of(new Person("Jack", 35), new Person("David", 30),
            new Person("Jack", 25));
    System.out.println(persons.stream().sorted(Comparator
            .comparing(Person::getName)
            .thenComparingInt(Person::getAge)).collect(toList()));
}
```

In Kotlin, you just enumerate which fields you want to compare:

```
data class Person(
    val name: String,
    val age: Int
)

fun main() {
    val persons = listOf(Person("Jack", 35), Person("David", 30),
        Person("Jack", 25))
    println(persons.sortedWith(compareBy(Person::name, Person::age)))
}
```

序列

In Java, you can generate a sequence of numbers this way:

```
// Java
int sum = IntStream.iterate(1, e -> e + 3)
    .limit(10).sum();
System.out.println(sum); // Prints 145
```

In Kotlin, use *sequences*. Multi-step processing of sequences is executed lazily when possible – actual computing happens only when the result of the whole processing chain is requested.

```
fun main() {
//sampleStart
    // Kotlin
    val sum = generateSequence(1) {
        it + 3
    }.take(10).sum()
    println(sum) // Prints 145
//sampleEnd
}
```

Sequences may reduce the number of steps that are needed to perform some filtering operations. See the sequence processing example, which shows the difference between `Iterable` and `Sequence`.

从列表中删除元素

In Java, the `remove()`) function accepts an index of an element to remove.

When removing an integer element, use the `Integer.valueOf()` function as the argument for the `remove()` function:

```
// Java
public void remove() {
    var numbers = new ArrayList<>();
    numbers.add(1);
    numbers.add(2);
    numbers.add(3);
    numbers.add(1);
    numbers.remove(1); // This removes by index
    System.out.println(numbers); // [1, 3, 1]
    numbers.remove(Integer.valueOf(1));
    System.out.println(numbers); // [3, 1]
}
```

In Kotlin, there are two types of element removal: by index with `removeAt()` and by value with `remove()`.

```
fun main() {
//sampleStart
    // Kotlin
    val numbers = mutableListOf(1, 2, 3, 1)
    numbers.removeAt(0)
    println(numbers) // [2, 3, 1]
    numbers.remove(1)
    println(numbers) // [2, 3]
//sampleEnd
}
```

遍历 map

In Java, you can traverse a map via `forEach`):

```
// Java
numbers.forEach((k,v) -> System.out.println("Key = " + k + ", Value = " + v));
```

In Kotlin, use a `for` loop or a `forEach`, similar to Java's `forEach`, to traverse a map:

```
// Kotlin
for ((k, v) in numbers) {
    println("Key = $k, Value = $v")
}
// Or
numbers.forEach { (k, v) -> println("Key = $k, Value = $v") }
```

获取可能会空的集合的首末元素

In Java, you can safely get the first and the last items by checking the size of the collection and using indices:

```
// Java
var list = new ArrayList<>();
//...
if (list.size() > 0) {
    System.out.println(list.get(0));
    System.out.println(list.get(list.size() - 1));
}
```

You can also use the `getFirst()`) and `getLast()`) functions for `Deque` and its inheritors:

```
// Java
var deque = new ArrayDeque<>();
//...
if (deque.size() > 0) {
    System.out.println(deque.getFirst());
    System.out.println(deque.getLast());
}
```

In Kotlin, there are the special functions `firstOrNull()` and `lastOrNull()` . Using the `Elvis operator` , you can perform further actions right away depending on the result of a function. For example, `firstOrNull()` :

```
// Kotlin
val emails = listOf<String>() // Might be empty
val theOldestEmail = emails.firstOrNull() ?: ""
val theFreshestEmail = emails.lastOrNull() ?: ""
```

由 list 创建 set

In Java, to create a `Set` from a `List` , you can use the `Set.copyOf`) function:

```
// Java
public void listToSet() {
    var sourceList = List.of(1, 2, 3, 1);
    var copySet = Set.copyOf(sourceList);
    System.out.println(copySet);
}
```

In Kotlin, use the function `toSet()` :

```
fun main() {
//sampleStart
    // Kotlin
    val sourceList = listOf(1, 2, 3, 1)
    val copySet = sourceList.toSet()
    println(copySet)
//sampleEnd
}
```

元素分组

In Java, you can group elements with the Collectors function `groupingBy()`:

```
// Java
public void analyzeLogs() {
    var requests = List.of(
        new Request("https://kotlinlang.org/docs/home.html", 200),
        new Request("https://kotlinlang.org/docs/home.html", 400),
        new Request("https://kotlinlang.org/docs/comparison-to-java.html", 200)
    );
    var urlsAndRequests = requests.stream().collect(
            Collectors.groupingBy(Request::getUrl));
    System.out.println(urlsAndRequests);
}
```

In Kotlin, use the function `groupBy()`:

```
data class Request(
    val url: String,
    val responseCode: Int
)

fun main() {
//sampleStart
    // Kotlin
    val requests = listOf(
        Request("https://kotlinlang.org/docs/home.html", 200),
        Request("https://kotlinlang.org/docs/home.html", 400),
        Request("https://kotlinlang.org/docs/comparison-to-java.html", 200)
    )
    println(requests.groupBy(Request::url))
//sampleEnd
}
```

过滤元素

In Java, to filter elements from a collection, you need to use the Stream API. The Stream API has `intermediate` and `terminal` operations. `filter()` is an intermediate operation, which returns a stream. To receive a collection as the output, you need to use a terminal operation, like `collect()`. For example, to leave only those pairs whose keys end with `1` and whose values are greater than `10`:

```
// Java
public void filterEndsWith() {
    var numbers = Map.of("key1", 1, "key2", 2, "key3", 3, "key11", 11);
    var filteredNumbers = numbers.entrySet().stream()
        .filter(entry -> entry.getKey().endsWith("1") && entry.getValue() > 10)
```

```
        .collect(Collectors.toMap(Map.Entry::getKey, Map.Entry::getValue));
    System.out.println(filteredNumbers);
}
```

In Kotlin, filtering is built into collections, and `filter()` returns the same collection type that was filtered. So, all you need to write is the `filter()` and its predicate:

```
fun main() {
//sampleStart
    // Kotlin
    val numbers = mapOf("key1" to 1, "key2" to 2, "key3" to 3, "key11" to 11)
    val filteredNumbers = numbers.filter { (key, value) -> key.endsWith("1") && value
> 10 }
    println(filteredNumbers)
//sampleEnd
}
```

Learn more about filtering maps here.

按类型过滤元素

In Java, to filter elements by type and perform actions on them, you need to check their types with the `instanceof` operator and then do the type cast:

```
// Java
public void objectIsInstance() {
    var numbers = new ArrayList<>();
    numbers.add(null);
    numbers.add(1);
    numbers.add("two");
    numbers.add(3.0);
    numbers.add("four");
    System.out.println("All String elements in upper case:");
    numbers.stream().filter(it -> it instanceof String)
        .forEach( it -> System.out.println(((String) it).toUpperCase()));
}
```

In Kotlin, you just call `filterIsInstance<NEEDED_TYPE>()` on your collection, and the type cast is done by Smart casts:

```
// Kotlin
fun main() {
//sampleStart
    // Kotlin
    val numbers = listOf(null, 1, "two", 3.0, "four")
    println("All String elements in upper case:")
    numbers.filterIsInstance<String>().forEach {
```

```
        println(it.uppercase())
    }
//sampleEnd
}
```

检验谓词

Some tasks require you to check whether all, none, or any elements satisfy a condition. In Java, you can do all of these checks via the Stream API functions `allMatch()`), `noneMatch()`), and `anyMatch()`):

```
// Java
public void testPredicates() {
    var numbers = List.of("one", "two", "three", "four");
    System.out.println(numbers.stream().noneMatch(it -> it.endsWith("e"))); // false
    System.out.println(numbers.stream().anyMatch(it -> it.endsWith("e"))); // true
    System.out.println(numbers.stream().allMatch(it -> it.endsWith("e"))); // false
}
```

In Kotlin, the extension functions `none()`, `any()`, and `all()` are available for every Iterable object:

```
fun main() {
//sampleStart
// Kotlin
    val numbers = listOf("one", "two", "three", "four")
    println(numbers.none { it.endsWith("e") })
    println(numbers.any { it.endsWith("e") })
    println(numbers.all { it.endsWith("e") })
//sampleEnd
}
```

Learn more about test predicates.

集合转换操作

元素合拢

In Java, you can make pairs from elements with the same positions in two collections by iterating simultaneously over them:

```
// Java
public void zip() {
    var colors = List.of("red", "brown");
```

```
    var animals = List.of("fox", "bear", "wolf");

    for (int i = 0; i < Math.min(colors.size(), animals.size()); i++) {
        String animal = animals.get(i);
        System.out.println("The " + animal.substring(0, 1).toUpperCase()
               + animal.substring(1) + " is " + colors.get(i));
   }
}
```

If you want to do something more complex than just printing pairs of elements into the output, you can use Records. In the example above, the record would be `record AnimalDescription(String animal, String color) {}`.

In Kotlin, use the `zip()` function to do the same thing:

```
fun main() {
//sampleStart
    // Kotlin
    val colors = listOf("red", "brown")
    val animals = listOf("fox", "bear", "wolf")

    println(colors.zip(animals) { color, animal ->
        "The ${animal.replaceFirstChar { it.uppercase() }} is $color" })
//sampleEnd
}
```

`zip()` returns the List of Pair objects.

> If collections have different sizes, the result of `zip()` is the smaller size. The last elements of the larger collection are not included in the result.

元素关联

In Java, you can use the Stream API to associate elements with characteristics:

```
// Java
public void associate() {
    var numbers = List.of("one", "two", "three", "four");
    var wordAndLength = numbers.stream()
        .collect(toMap(number -> number, String::length));
    System.out.println(wordAndLength);
}
```

In Kotlin, use the `associate()` function:

```
fun main() {
//sampleStart
    // Kotlin
    val numbers = listOf("one", "two", "three", "four")
    println(numbers.associateWith { it.length })
//sampleEnd
}
```

下一步做什么？

- Visit Kotlin Koans – complete exercises to learn Kotlin syntax. Each exercise is created as a failing unit test and your job is to make it pass.
- Look through other Kotlin idioms.
- Learn how to convert existing Java code to Kotlin with the Java to Kotlin converter.
- Discover collections in Kotlin.

If you have a favorite idiom, we invite you to share it by sending a pull request.

Java 与 Kotlin 中的可空性

Nullability is the ability of a variable to hold a `null` value. When a variable contains `null`, an attempt to dereference the variable leads to a `NullPointerException`. There are many ways to write code in order to minimize the probability of receiving null pointer exceptions.

This guide covers differences between Java's and Kotlin's approaches to handling possibly nullable variables. It will help you migrate from Java to Kotlin and write your code in authentic Kotlin style.

The first part of this guide covers the most important difference – support for nullable types in Kotlin and how Kotlin processes types from Java code. The second part, starting from Checking the result of a function call, examines several specific cases to explain certain differences.

Learn more about null safety in Kotlin.

Support for nullable types

The most important difference between Kotlin's and Java's type systems is Kotlin's explicit support for nullable types. It is a way to indicate which variables can possibly hold a `null` value. If a variable can be `null`, it's not safe to call a method on the variable because this can cause a `NullPointerException`. Kotlin prohibits such calls at compile time and thereby prevents lots of possible exceptions. At runtime, objects of nullable types and objects of non-nullable types are treated the same: A nullable type isn't a wrapper for a non-nullable type. All checks are performed at compile time. That means there's almost no runtime overhead for working with nullable types in Kotlin.

> We say "almost" because, even though intrinsic checks *are* generated, their overhead is minimal.
>
> 

In Java, if you don't write null checks, methods may throw a `NullPointerException`:

```
// Java
int stringLength(String a) {
    return a.length();
}

void main() {
    stringLength(null); // Throws a `NullPointerException`
}
```

This call will have the following output:

```
java.lang.NullPointerException: Cannot invoke "String.length()" because "a" is null
    at test.java.Nullability.stringLength(Nullability.java:8)
    at test.java.Nullability.main(Nullability.java:12)
    at java.base/java.util.ArrayList.forEach(ArrayList.java:1511)
    at java.base/java.util.ArrayList.forEach(ArrayList.java:1511)
```

In Kotlin, all regular types are non-nullable by default unless you explicitly mark them as nullable. If you don't expect `a` to be `null`, declare the `stringLength()` function as follows:

```
// Kotlin
fun stringLength(a: String) = a.length
```

The parameter `a` has the `String` type, which in Kotlin means it must always contain a `String` instance and it cannot contain `null`. Nullable types in Kotlin are marked with a question mark `?`, for example, `String?`. The situation with a `NullPointerException` at runtime is impossible if `a` is `String` because the compiler enforces the rule that all arguments of `stringLength()` not be `null`.

An attempt to pass a `null` value to the `stringLength(a: String)` function will result in a compile-time error, "Null can not be a value of a non-null type String":



If you want to use this function with any arguments, including `null`, use a question mark after the argument type `String?` and check inside the function body to ensure that the value of the argument is not `null`:

```
// Kotlin
fun stringLength(a: String?): Int = if (a != null) a.length else 0
```

After the check is passed successfully, the compiler treats the variable as if it were of the non-nullable type `String` in the scope where the compiler performs the check.

If you don't perform this check, the code will fail to compile with the following message: "Only safe (?.) or non-nullable asserted (!!.) calls are allowed on a nullable receiver of type String?".

You can write the same shorter – use the safe-call operator ?. (If-not-null shorthand), which allows you to combine a null check and a method call into a single operation:

```
// Kotlin
fun stringLength(a: String?): Int = a?.length ?: 0
```

Platform types

In Java, you can use annotations showing whether a variable can or cannot be `null`. Such annotations aren't part of the standard library, but you can add them separately. For example, you can use the JetBrains annotations `@Nullable` and `@NotNull` (from the `org.jetbrains.annotations` package) or annotations from Eclipse (`org.eclipse.jdt.annotation`). Kotlin can recognize such annotations when you're calling Java code from Kotlin code and will treat types according to their annotations.

If your Java code doesn't have these annotations, then Kotlin will treat Java types as *platform types*. But since Kotlin doesn't have nullability information for such types, its compiler will allow all operations on them. You will need to decide whether to perform null checks, because:

- Just as in Java, you'll get a `NullPointerException` if you try to perform an operation on `null`.
- The compiler won't highlight any redundant null checks, which it normally does when you perform a null-safe operation on a value of a non-nullable type.

Learn more about calling Java from Kotlin in regard to null-safety and platform types.

Support for definitely non-nullable types

In Kotlin, if you want to override a Java method that contains `@NotNull` as an argument, you need Kotlin's definitely non-nullable types.

For example, consider this `load()` method in Java:

```
import org.jetbrains.annotations.*;

public interface Game<T> {
  public T save(T x) {}
  @NotNull
  public T load(@NotNull T x) {}
}
```

To override the `load()` method in Kotlin successfully, you need `T1` to be declared as definitely non-nullable (`T1 & Any`):

```
interface ArcadeGame<T1> : Game<T1> {
  override fun save(x: T1): T1
  // T1 is definitely non-nullable
  override fun load(x: T1 & Any): T1 & Any
}
```

Learn more about generic types that are definitely non-nullable.

Checking the result of a function call

One of the most common situations where you need to check for `null` is when you obtain a result from a function call.

In the following example, there are two classes, `Order` and `Customer`. `Order` has a reference to an instance of `Customer`. The `findOrder()` function returns an instance of the `Order` class, or `null` if it can't find the order. The objective is to process the customer instance of the retrieved order.

Here are the classes in Java:

```
//Java
record Order (Customer customer) {}

record Customer (String name) {}
```

In Java, call the function and do an if-not-null check on the result to proceed with the dereferencing of the required property:

```
// Java
Order order = findOrder();

if (order != null) {
    processCustomer(order.getCustomer());
}
```

Converting the Java code above to Kotlin code directly results in the following:

```
// Kotlin
data class Order(val customer: Customer)

data class Customer(val name: String)

val order = findOrder()
```

```
// Direct conversion
if (order != null){
    processCustomer(order.customer)
}
```

Use the safe-call operator `?.` (If-not-null shorthand) in combination with any of the scope functions from the standard library. The `let` function is usually used for this:

```
// Kotlin
val order = findOrder()

order?.let {
    processCustomer(it.customer)
}
```

Here is a shorter version of the same:

```
// Kotlin
findOrder()?.customer?.let(::processCustomer)
```

Default values instead of null

Checking for `null` is often used in combination with setting the default value in case the null check is successful.

The Java code with a null check:

```
// Java
Order order = findOrder();
if (order == null) {
    order = new Order(new Customer("Antonio"))
}
```

To express the same in Kotlin, use the Elvis operator (If-not-null-else shorthand):

```
// Kotlin
val order = findOrder() ?: Order(Customer("Antonio"))
```

Functions returning a value or null

In Java, you need to be careful when working with list elements. You should always check whether an element exists at an index before you attempt to use the element:

```
// Java
var numbers = new ArrayList<Integer>();
numbers.add(1);
numbers.add(2);

System.out.println(numbers.get(0));
//numbers.get(5) // Exception!
```

The Kotlin standard library often provides functions whose names indicate whether they can possibly return a `null` value. This is especially common in the collections API:

```
fun main() {
//sampleStart
    // Kotlin
    // The same code as in Java:
    val numbers = listOf(1, 2)

    println(numbers[0])  // Can throw IndexOutOfBoundsException if the collection is e
mpty
    //numbers.get(5)     // Exception!

    // More abilities:
    println(numbers.firstOrNull())
    println(numbers.getOrNull(5)) // null
//sampleEnd
}
```

Aggregate operations

When you need to get the biggest element or `null` if there are no elements, in Java you would use the Stream API:

```
// Java
var numbers = new ArrayList<Integer>();
var max = numbers.stream().max(Comparator.naturalOrder()).orElse(null);
System.out.println("Max: " + max);
```

In Kotlin, use aggregate operations:

```
// Kotlin
val numbers = listOf<Int>()
println("Max: ${numbers.maxOrNull()}")
```

Learn more about collections in Java and Kotlin.

Casting types safely

When you need to safely cast a type, in Java you would use the `instanceof` operator and then check how well it worked:

```
// Java
int getStringLength(Object y) {
    return y instanceof String x ? x.length() : -1;
}

void main() {
    System.out.println(getStringLength(1)); // Prints `-1`
}
```

To avoid exceptions in Kotlin, use the safe cast operator `as?`, which returns `null` on failure:

```
// Kotlin
fun main() {
    println(getStringLength(1)) // Prints `-1`
}

fun getStringLength(y: Any): Int {
    val x: String? = y as? String // null
    return x?.length ?: -1 // Returns -1 because `x` is null
}
```

> In the Java example above, the function `getStringLength()` returns a result of the primitive type `int`. To make it return `null`, you can use the *boxed* type `Integer`. However, it's more resource-efficient to make such functions return a negative value and then check the value – you would do the check anyway, but no additional boxing is performed this way.
>
> 

下一步做什么？

- Browse other Kotlin idioms.
- Learn how to convert existing Java code to Kotlin with the Java-to-Kotlin (J2K) converter.
- Check out other migration guides:
  - Strings in Java and Kotlin
  - Collections in Java and Kotlin

If you have a favorite idiom, feel free to share it with us by sending a pull request!

标准输入

> Java Scanner is a slow tool. Use it only when you need the specific functionalities it offers. Otherwise, it's generally preferable to use Kotlin's `readln()` function to read standard input.



To read from the standard input, Java provides the `Scanner` class. Kotlin offers two main ways to read from the standard input: the `Scanner` class, similar to Java, and the `readln()` function.

Read from the standard input with Java Scanner

In Java, the standard input is typically accessed through the `System.in` object. You need to import the `Scanner` class, create an object, and use methods like `.nextLine()` and `.nextInt()` to read different data types:

```
//Java implementation
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        // Reads a single line of input. For example: Hi there!
        System.out.print("Enter a line: ");
        String line = scanner.nextLine();
        System.out.println("You entered: " + line);
        // You entered: Hi there!

        // Reads an integer. For example: 08081990
        System.out.print("Enter an integer: ");
        int number = scanner.nextInt();
        System.out.println("You entered: " + number);
        // You entered: 08081990

        scanner.close();
    }
}
```

Use Java Scanner in Kotlin

Due to Kotlin's interoperability with Java libraries, you can access Java Scanner from Kotlin code out of the box.

To use Java Scanner in Kotlin, you need to import the `Scanner` class and initialize it by passing a `System.in` object that represents the standard input stream and dictates how to read the data. You can use the available reading methods for reading values different from strings, such as `.nextLine()`, `.next()`, and `.nextInt()`:

```
// Imports Java Scanner
import java.util.Scanner

fun main() {
    // Initializes the Scanner
    val scanner = Scanner(System.`in`)

    // Reads a whole string line. For example: "Hello, Kotlin"
    val line = scanner.nextLine()
    print(line)
    // Hello, Kotlin

    // Reads a string. For example: "Hello"
    val string = scanner.next()
    print(string)
    // Hello

    // Reads a number. For example: 123
    val num = scanner.nextInt()
    print(num)
    // 123
}
```

Other useful methods for reading input with Java Scanner are `.hasNext()`, `.useDelimiter()`, and `.close()`:

- The `.hasNext()` method checks if there's more data available in the input. It returns the boolean value `true` if there are remaining elements to iterate and `false` if no more elements are left in the input.
- The `.useDelimiter()` method sets the delimiter for reading input elements. The delimiter is whitespaces by default, but you can specify other characters. For example, `.useDelimiter(",")` reads the input elements separated by commas.
- The `.close()` method closes the input stream associated with the Scanner, preventing further use of the Scanner for reading input.

> Always use the `.close()` method when you're finished using Java Scanner. Closing the Java Scanner releases the resources it consumes and ensures proper program behavior.



Read from the standard input with readln()

In Kotlin, apart from the Java Scanner, you have the `readln()` function. It is the most straightforward way to read input. This function reads a line of text from the standard input and returns it as a string:

```
// Reads a string. For example: Charlotte
val name = readln()

// Reads a string and converts it into an integer. For example: 43
val age = readln().toInt()

println("Hello, $name! You are $age years old.")
// Hello, Charlotte! You are 43 years old.
```

For more information, see Read standard input.

原生

- Kotlin/Native 入门——在 IntelliJ IDEA 中
- 定义文件
- 与 C 语言互操作
  - 与 C 语言互操作性
  - 映射来自 C 语言的原始数据类型——教程
  - 映射来自 C 语言的结构与联合类型——教程
  - 映射来自 C 语言的函数指针——教程
  - 映射来自 C 语言的字符串——教程
  - Kotlin/Native 开发动态库——教程
  - 创建使用 C 语言互操作与 libcurl 的应用——教程
- 与 Objective-C 互操作性
  - 与 Swift/Objective-C 互操作性
  - Kotlin/Native 开发 Apple framework——教程
- Kotlin/Native 库
- 平台库
- 内存管理器
  - Kotlin/Native 内存管理
  - 与 ARC 集成
  - 迁移到新版内存管理器
- 调试 Kotlin/Native
- 符号化 iOS 崩溃报告
- 参考与提示
  - 目标平台支持
  - 改进编译时间
  - 许可证文件
  - Kotlin/Native FAQ

Kotlin/Native 入门

In this tutorial, you'll learn how to create a Kotlin/Native application. Choose the tool that works best for you and create your app using:

- **The IDE**. Here, you can clone the project template from a version control system and use it in IntelliJ IDEA.
- **The Gradle build system**. To better understand how things work under the hood, create build files for your project manually.
- **The command line tool**. You can use the Kotlin/Native compiler, which is shipped as a part of the standard Kotlin distribution, and create the app directly in the command line tool.

  Console compilation may seem easy and straightforward, but it doesn't scale well for larger projects with hundreds of files and libraries. For such projects, we recommend using an IDE or a build system.

With Kotlin/Native, you can compile for different targets, including Linux, macOS, and Windows. While cross-platform compilation is possible, which means using one platform to compile for a different one, in this tutorial, you'll be targeting the same platform you're compiling on.

> If you use a Mac and want to create and run applications for macOS or other Apple targets, you also need to install Xcode Command Line Tools, launch it, and accept the license terms first.
>
> 

In IDE

In this section, you'll learn how to use IntelliJ IDEA to create a Kotlin/Native application. You can use both the Community Edition and the Ultimate Edition.

Create the project

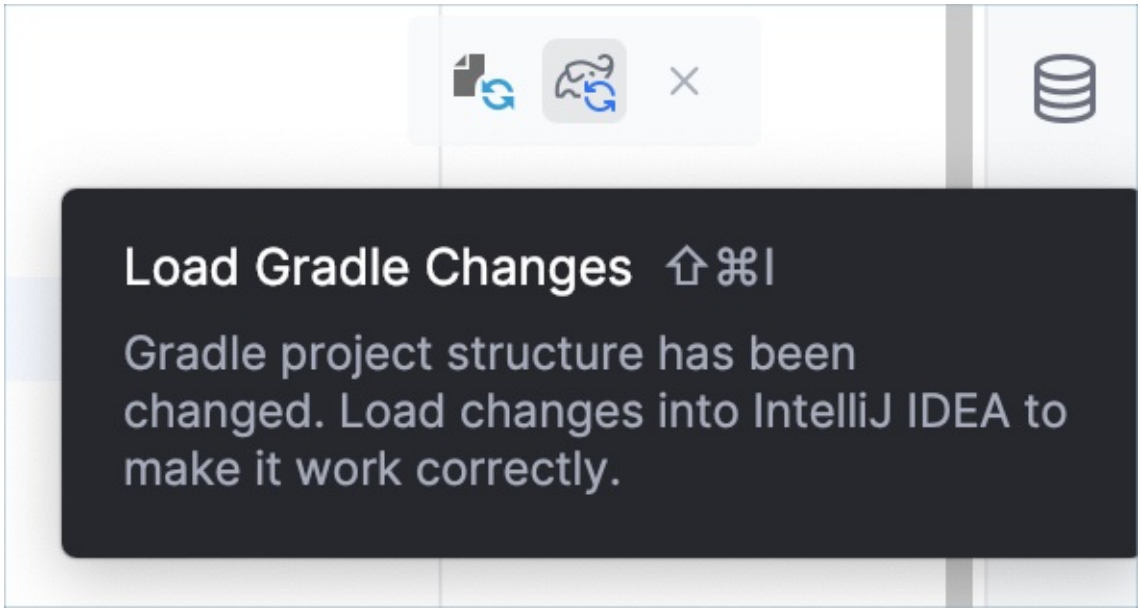
1. Download and install the latest version of IntelliJ IDEA.
2. Clone the project template by selecting **File** | **New** | **Project from Version Control** in IntelliJ IDEA and using this URL:

   ```
   https://github.com/Kotlin/kmp-native-wizard
   ```

3. Open the `gradle/libs.versions.toml` file, which is the version catalog for project dependencies. To create Kotlin/Native applications, you need the Kotlin Multiplatform Gradle plugin, which has the same version as Kotlin. Ensure that you use the latest Kotlin version:

   ```
   [versions]
   kotlin = "2.1.21"
   ```

4. Follow the suggestion to reload Gradle files:



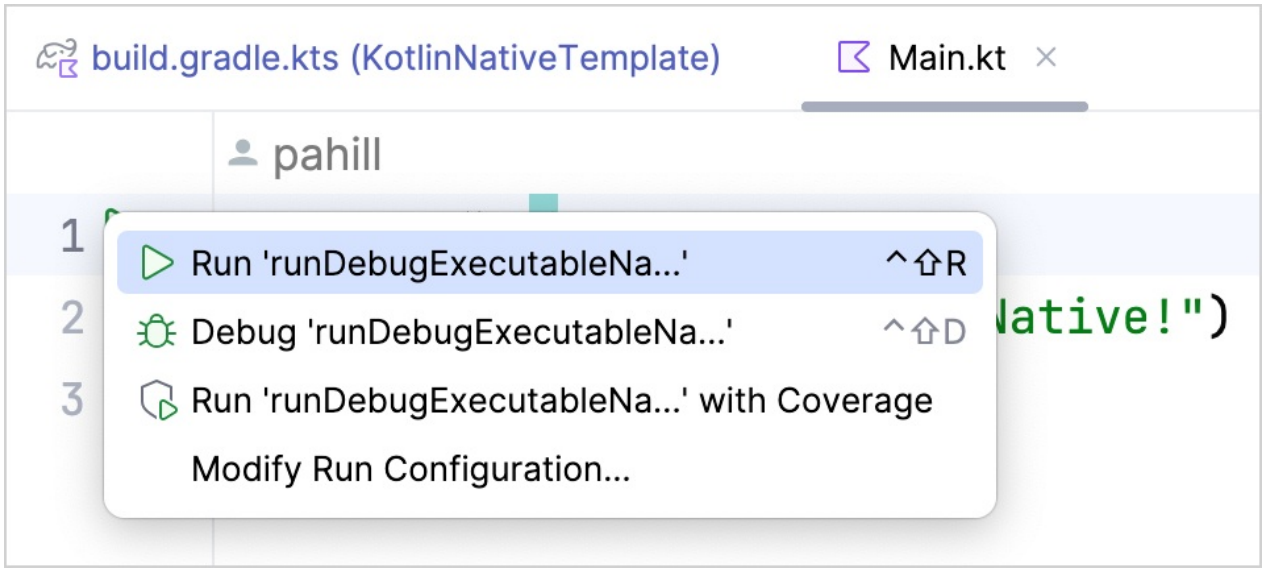
For more information about these settings, see the Multiplatform Gradle DSL reference.

Build and run the application

Open the `Main.kt` file in the `src/nativeMain/kotlin/` directory:

- The `src` directory contains Kotlin source files.
- The `Main.kt` file includes code that prints "Hello, Kotlin/Native!" using the `println()` function.

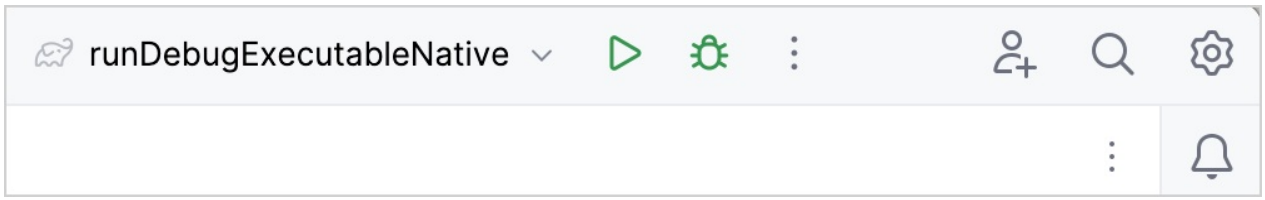
Press the green icon in the gutter to run the code:



IntelliJ IDEA runs the code using the Gradle task and outputs the result in the **Run** tab:

```
> Task :linkDebugExecutableNative

> Task :runDebugExecutableNative
Hello, Kotlin/Native!

BUILD SUCCESSFUL in 7s
```

After the first run, the IDE creates the corresponding run configuration at the top:



> IntelliJ IDEA Ultimate users can install the Native Debugging Support plugin that allows debugging compiled native executables and also automatically creates run configurations for imported Kotlin/Native projects.

You can configure IntelliJ IDEA to build your project automatically:

1. Go to **Settings | Build, Execution, Deployment | Compiler**.
2. On the **Compiler** page, select **Build project automatically**.
3. Apply the changes.

Now, when you make changes in the class files or save the file (Ctrl + S/Cmd + S), IntelliJ IDEA automatically performs an incremental build of the project.

Update the application

Let's add a feature to your application so it can count the number of letters in your name:

1. In the `Main.kt` file, add code to read the input. Use the `readln()` function to read the input value and assign it to the `name` variable:

```
fun main() {
    // Read the input value.
    println("Hello, enter your name:")
    val name = readln()
}
```

2. To run this app using Gradle, specify `System.in` as the input to use in the `build.gradle.kts` file and load the Gradle changes:

```
kotlin {
    //...
    nativeTarget.apply {
        binaries {
            executable {
                entryPoint = "main"
                runTask?.standardInput = System.`in`
            }
        }
    }
    //...
}
```

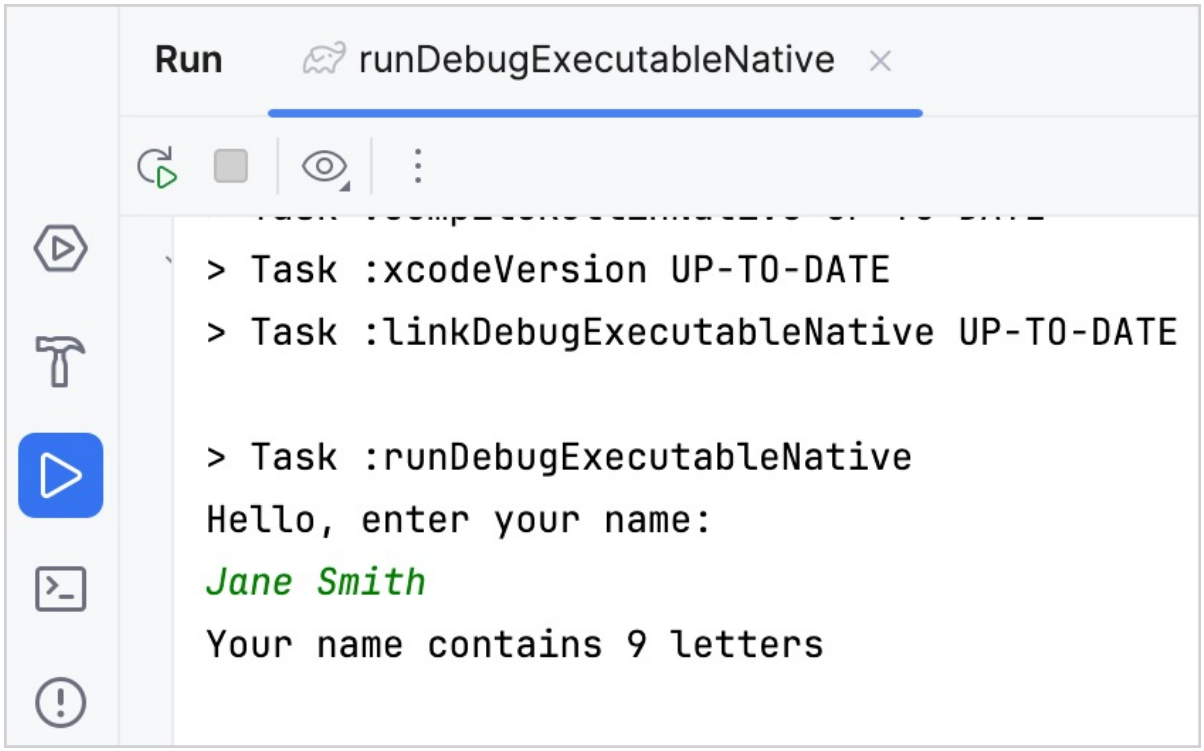
1. Eliminate the whitespaces and count the letters:

   - Use the `replace()` function to remove the empty spaces in the name.
   - Use the scope function `let` to run the function within the object context.
   - Use a string template to insert your name length into the string by adding a dollar sign `$` and enclosing it in curly braces – `${it.length}`. `it` is the default name of a lambda parameter.

```
fun main() {
    // Read the input value.
    println("Hello, enter your name:")
    val name = readln()
    // Count the letters in the name.
    name.replace(" ", "").let {
        println("Your name contains ${it.length} letters")
```

```
    }
}
```

2. Run the application.
3. Enter your name and enjoy the result:



Now let's count only the unique letters in your name:

1. In the `Main.kt` file, declare the new extension function `.countDistinctCharacters()` for `String`:
   - Convert the name to lowercase using the `.lowercase()` function.
   - Convert the input string to a list of characters using the `toList()` function.
   - Select only the distinct characters in your name using the `distinct()` function.
   - Count the distinct characters using the `count()` function.

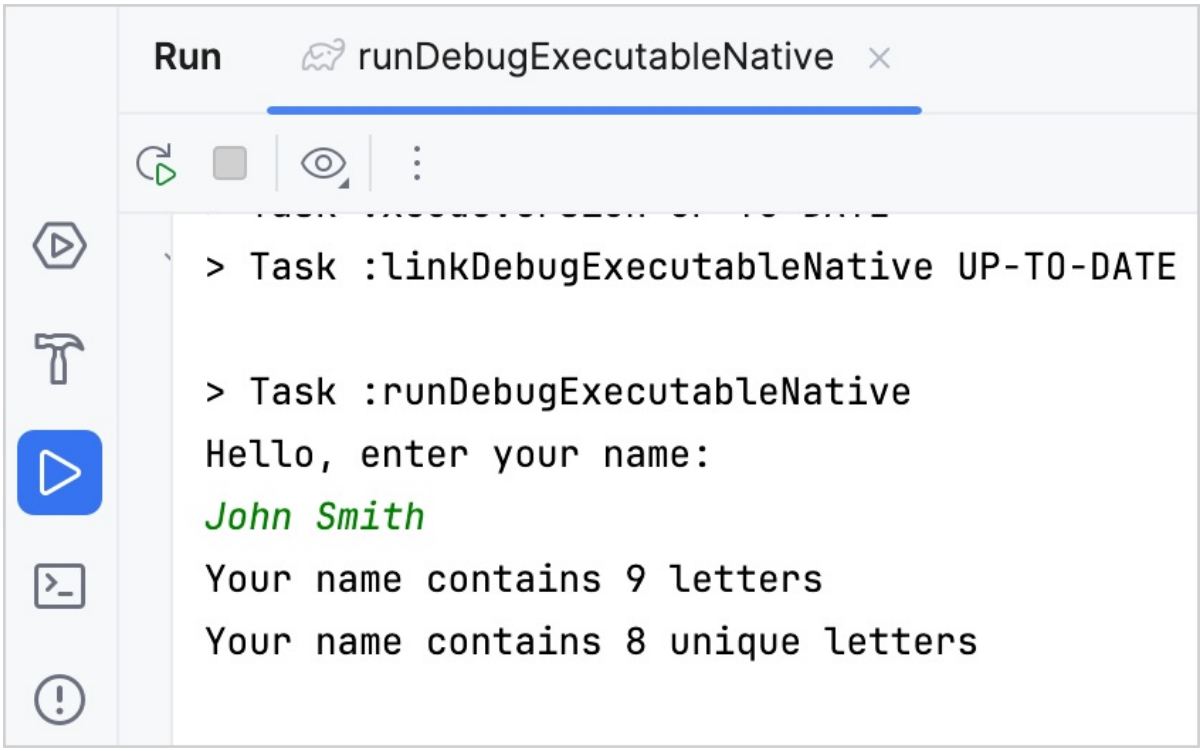
```
fun String.countDistinctCharacters() = lowercase().toList().distinct().count()
```

2. Use the `.countDistinctCharacters()` function to count the unique letters in your name:

```
fun String.countDistinctCharacters() = lowercase().toList().distinct().count()

fun main() {
    // Read the input value.
    println("Hello, enter your name:")
    val name = readln()
    // Count the letters in the name.
    name.replace(" ", "").let {
```

```
        println("Your name contains ${it.length} letters")
        // Print the number of unique letters.
        println("Your name contains ${it.countDistinctCharacters()} unique letters"
)
    }
}
```

3. Run the application.
4. Enter your name and see the result:



Using Gradle

In this section, you'll learn how to manually create a Kotlin/Native application using Gradle. It's the default build system for Kotlin/Native and Kotlin Multiplatform projects, which is also commonly used in Java, Android, and other ecosystems.

Create project files

1. To get started, install a compatible version of Gradle. See the compatibility table to check the Kotlin Gradle plugin (KGP) compatibility with available Gradle versions.
2. Create an empty project directory. Inside it, create a `build.gradle(.kts)` file with the following content:

> 【Kotlin】

```
// build.gradle.kts
plugins {
    kotlin("multiplatform") version "2.1.21"
}

repositories {
    mavenCentral()
}

kotlin {
    macosArm64("native") {  // on macOS
    // linuxArm64("native") // on Linux
    // mingwX64("native")   // on Windows
        binaries {
            executable()
        }
    }
}

tasks.withType<Wrapper> {
    gradleVersion = "8.10"
    distributionType = Wrapper.DistributionType.BIN
}
```

> 【Groovy】

```
// build.gradle
plugins {
    id 'org.jetbrains.kotlin.multiplatform' version '2.1.21'
}

repositories {
    mavenCentral()
}

kotlin {
    macosArm64('native') {  // on macOS
    // linuxArm64('native') // on Linux
    // mingwX64('native')   // on Windows
        binaries {
            executable()
        }
    }
}

wrapper {
    gradleVersion = '8.10'
```

```
        distributionType = 'BIN'
    }
```

You can use different target names, such as `macosArm64`, `iosArm64` `linuxArm64`, and `mingwX64` to define the targets for which you are compiling your code. These target names can optionally take the platform name as a parameter, which in this case is `native`. The platform name is used to generate the source paths and task names in the project.

1. Create an empty `settings.gradle(.kts)` file in the project directory.
2. Create a `src/nativeMain/kotlin` directory and place a `hello.kt` file inside with the following content:

   ```
   fun main() {
       println("Hello, Kotlin/Native!")
   }
   ```

By convention, all sources are located in the `src/<target name>[Main|Test]/kotlin` directories, where `Main` is for the source code and `Test` is for tests. `<target name>` corresponds to the target platform (in this case, `native`), as specified in the build file.

Build and run the project

1. From the root project directory, run the build command:

   ```
   ./gradlew nativeBinaries
   ```

   This command creates the `build/bin/native` directory with two directories inside: `debugExecutable` and `releaseExecutable`. They contain the corresponding binary files.

   By default, the name of the binary file is the same as the project directory.

2. To run the project, execute the following command:

   ```
   build/bin/native/debugExecutable/<project_name>.kexe
   ```

The terminal prints "Hello, Kotlin/Native!".

Open the project in IDE

Now, you can open your project in any IDE that supports Gradle. If you use IntelliJ IDEA:

1. Select **File** | **Open**.
2. Select the project directory and click **Open**. IntelliJ IDEA automatically detects if it's a Kotlin/Native project.

If you encounter a problem with the project, IntelliJ IDEA displays the error message in the **Build** tab.

Using the command-line compiler

In this section, you'll learn how to create a Kotlin/Native application using the Kotlin compiler in the command line tool.

Download and install the compiler

To install the compiler:

1. Go to the Kotlin's GitHub releases page.
2. Look for a file with `kotlin-native` in the name and download one that is suitable for your operating system, for example `kotlin-native-prebuilt-linux-x86_64-2.0.21.tar.gz`.
3. Unpack the archive to a directory of your choice.
4. Open your shell profile and add the path to the compiler's `/bin` directory to the `PATH` environment variable:

   ```
   export PATH="/<path to the compiler>/kotlin-native/bin:$PATH"
   ```

> Although the compiler output has no dependencies or virtual machine requirements, the compiler itself requires Java 1.8 or higher runtime. It's supported by JDK 8 (JAVA SE 8) or later versions.
>
> 

Create the program

Choose a working directory and create a file named `hello.kt`. Update it with the following code:

```
fun main() {
    println("Hello, Kotlin/Native!")
}
```

Compile the code from the console

To compile the application, execute the following command with the downloaded compiler:

```
kotlinc-native hello.kt -o hello
```

The value of the `-o` option specifies the name of the output file, so this call generates the `hello.kexe` binary file on macOS and Linux (and `hello.exe` on Windows).

For the full list of available options, see Kotlin compiler options.

Run the program

To run the program, in your command line tool, navigate to the directory containing the binary file and run the following command:

> 【macOS and Linux】

```
./hello.kexe
```

> 【Windows】

```
./hello.exe
```

The application prints "Hello, Kotlin/Native" to the standard output.

下一步做什么？

- Complete the Create an app using C interop and libcurl tutorial that explains how to create a native HTTP client and interoperate with C libraries.
- Learn how to write Gradle build scripts for real-life Kotlin/Native projects.
- Read more about the Gradle build system in the documentation.

定义文件

Kotlin/Native enables you to consume C and Objective-C libraries, allowing you to use their functionality in Kotlin. A special tool called cinterop takes a C or an Objective-C library and generates the corresponding Kotlin bindings, so that the library's methods can be used in your Kotlin code as usual.

To generate these bindings, each library needs a definition file, usually with the same name as the library. This is a property file that describes exactly how the library should be consumed. See the full list of available properties.

Here's a general workflow when working with a project:

1. Create a `.def` file describing what to include in the bindings.
2. Use the generated bindings in your Kotlin code.
3. Run the Kotlin/Native compiler to produce the final executable.

Create and configure a definition file

Let's create a definition file and generate bindings for a C library:

1. In your IDE, select the `src` folder and create a new directory with **File | New | Directory**.
2. Name the new directory `nativeInterop/cinterop`.

   This is the default convention for `.def` file locations, but it can be overridden in the `build.gradle.kts` file if you use a different location.

3. Select the new subfolder and create a `png.def` file with **File | New | File**.
4. Add the necessary properties:

   ```
   headers = png.h
   headerFilter = png.h
   package = png

   compilerOpts.linux = -I/usr/include -I/usr/include/x86_64-linux-gnu
   linkerOpts.osx = -L/opt/local/lib -L/usr/local/opt/png/lib -lpng
   linkerOpts.linux = -L/usr/lib/x86_64-linux-gnu -lpng
   ```

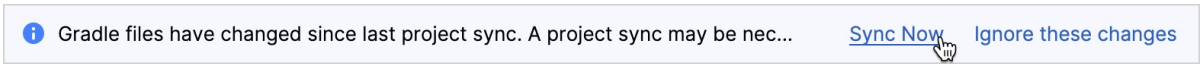
   - `headers` is the list of header files to generate Kotlin stubs for. You can add multiple files to this entry, separating each with a space. In this case, it's only `png.h`. The referenced files need to be available on the specified path (in this case, it's `/usr/include/png`).

- `headerFilter` shows what exactly is included. In C, all the headers are also included when one file references another one with the `#include` directive. Sometimes it's not necessary, and you can add this parameter using glob patterns) to make adjustments.

  You can use `headerFilter` if you don't want to fetch external dependencies (such as system `stdint.h` header) into the interop library. Also, it may be useful for library size optimization and fixing potential conflicts between the system and the provided Kotlin/Native compilation environment.

- If the behavior for a certain platform needs to be modified, you can use a format like `compilerOpts.osx` or `compilerOpts.linux` to provide platform-specific values to the options. In this case, they are macOS (the `.osx` suffix) and Linux (the `.linux` suffix). Parameters without a suffix are also possible (for example, `linkerOpts=`) and applied to all platforms.

5. To generate bindings, synchronize the Gradle files by clicking **Sync Now** in the notification.



After the bindings generation, the IDE can use them as a proxy view of the native library.

> You can also configure bindings generation by using the cinterop tool in the command line.
>
> 

Properties

Here's the full list of properties you can use in your definition file to adjust the content of the generated binaries. For more information, see the corresponding sections below.

Property	Description
`headers`	The list of headers from a library to be included in the bindings.
`modules`	The list of Clang modules from an Objective-C library to be included in the bindings.
`language`	Specifies the language. C is used by default; change to `Objective-C` if necessary.
`compilerOpts`	Compiler options that the cinterop tool passes to the C compiler.
`linkerOpts`	Linker options that the cinterop tool passes to the linker.
`excludedFunctions`	A space-separated list of function names that should be ignored.

`staticLibraries`	Experimental. Includes a static library into `.klib`.
`libraryPaths`	Experimental. A space-separated list of directories where the cinterop tool searches for the library to be included in `.klib`.
`packageName`	Package prefix for the generated Kotlin API.
`headerFilter`	Filters headers by globs and includes only them when importing a library.
`excludeFilter`	Excludes specific headers when importing a library and takes priority over `headerFilter`.
`strictEnums`	A space-separated list of enums that should be generated as Kotlin enums.
`nonStrictEnums`	A space-separated list of enums that should be generated as integral values.
`noStringConversion`	A space-separated list of functions whose `const char*` parameters should not be auto-converted to Kotlin `String`s.
`allowedOverloadsForCFunctions`	By default, it's assumed that C functions have unique names. If several functions have the same name, only one is picked. However, you can change this by specifying these functions in `allowedOverloadsForCFunctions`.
`disableDesignatedInitializerChecks`	Disables the compiler check that doesn't allow calling a non-designated Objective-C initializer as a `super()` constructor.
`foreignExceptionMode`	Wraps exceptions from Objective-C code into Kotlin exceptions with the `ForeignException` type.
`userSetupHint`	Adds a custom message, for example, to help users resolve linker errors.

In addition to the list of properties, you can include custom declarations in your definition file.

Import headers

If a C library does not have a Clang module and instead consists of a set of headers, use the `headers` property to specify headers that should be imported:

```
headers = curl/curl.h
```

Filter headers by globs

You can filter headers by globs using filter properties from the `.def` file. To include declarations from headers, use the `headerFilter` property. If a header matches any of the globs, its declarations are included in the bindings.

The globs are applied to the header paths relative to the appropriate include path elements, for example, `time.h` or `curl/curl.h`. So if the library is usually included with `#include <SomeLibrary/Header.h>`, you can probably filter headers with the following filter:

```
headerFilter = SomeLibrary/**
```

If `headerFilter` is not provided, all the headers are included. However, we encourage you to use `headerFilter` and specify the glob as precisely as possible. In this case, the generated library contains only the necessary declarations. It can help avoid various issues when upgrading Kotlin or tools in your development environment.

Exclude headers

To exclude specific headers, use the `excludeFilter` property. It can be helpful to remove redundant or problematic headers and optimize compilation, as declarations from the specified headers are not included in the bindings:

```
excludeFilter = SomeLibrary/time.h
```

> If the same header is both included with `headerFilter`, and excluded with `excludeFilter`, the specified header will not be included in the bindings.
>
> 

Import modules

If an Objective-C library has a Clang module, use the `modules` property to specify the module to be imported:

```
modules = UIKit
```

Pass compiler and linker options

Use the `compilerOpts` property to pass options to the C compiler, which is used to analyze headers under the hood. To pass options to the linker, which is used to link final executables, use `linkerOpts`. For example:

```
compilerOpts = -DFOO=bar
linkerOpts = -lpng
```

You can also specify target-specific options that apply only to a certain target:

```
compilerOpts = -DBAR=bar
compilerOpts.linux_x64 = -DFOO=foo1
compilerOpts.macos_x64 = -DFOO=foo2
```

With this configuration, headers are analyzed using `-DBAR=bar -DFOO=foo1` on Linux and `-DBAR=bar -DFOO=foo2` on macOS. Note that any definition file option can have both common and platform-specific parts.

Ignore specific functions

Use the `excludedFunctions` property to specify a list of the function names that should be ignored. This can be useful if a function declared in the header isn't guaranteed to be callable, and it's difficult or impossible to determine this automatically. You can also use this property to work around a bug in the interop itself.

Include a static library

> This feature is Experimental. It may be dropped or changed at any time. Use it only for evaluation purposes.
>
> 

Sometimes it's more convenient to ship a static library with your product, rather than assume it is available within the user's environment. To include a static library into `.klib`, use `staticLibrary` and `libraryPaths` properties:

```
headers = foo.h
staticLibraries = libfoo.a
libraryPaths = /opt/local/lib /usr/local/opt/curl/lib
```

When given the above snippet, the cinterop tool searches `libfoo.a` in `/opt/local/lib` and `/usr/local/opt/curl/lib`, and if found, includes the library binary in the `klib`.

When using a `klib` like this in your program, the library is linked automatically.

Configure enums generation

Use the `strictEnums` property to generate enums as Kotlin enums or `nonStrictEnums` to generate them as integral values. If an enum is not included in either of these lists, it is generated based on heuristics.

Set up string conversion

Use the `noStringConversion` property to disable automatic conversion of the `const char*` function parameters as Kotlin `String` s.

Allow calling a non-designated initializer

By default, the Kotlin/Native compiler doesn't allow calling a non-designated Objective-C initializer as a `super()` constructor. This behavior can be inconvenient if the designated Objective-C initializers aren't marked properly in the library. To disable these compiler checks, use the `disableDesignatedInitializerChecks` property.

Handle Objective-C exceptions

By default, the program crashes if Objective-C exceptions reach the Objective-C to Kotlin interop boundary and get to the Kotlin code.

To propagate Objective-C exceptions to Kotlin, enable wrapping with the `foreignExceptionMode = objc-wrap` property. In this case, Objective-C exceptions are translated into Kotlin exceptions that get the `ForeignException` type.

Help resolve linker errors

Linker errors might occur when a Kotlin library depends on C or Objective-C libraries, for example, using the CocoaPods integration. If dependent libraries aren't installed locally on the machine or configured explicitly in the project build script, the "Framework not found" error occurs.

If you're a library author, you can help your users resolve linker errors with custom messages. To do that, add a `userSetupHint=message` property to your `.def` file or pass the `-Xuser-setup-hint` compiler option to `cinterop` .

Add custom declarations

Sometimes it is required to add custom C declarations to the library before generating bindings (for example, for macros). Instead of creating an additional header file with these declarations, you can include them directly to the end of the `.def` file, after a separating line, containing only the separator sequence `---` :

```
headers = errno.h
---

static inline int getErrno() {
    return errno;
}
```

Note that this part of the `.def` file is treated as part of the header file, so functions with the body should be declared as `static`. The declarations are parsed after including the files from the `headers` list.

Generate bindings using command line

In addition to the definition file, you can specify what to include in bindings by passing the corresponding properties as options in the `cinterop` call.

Here's an example of the command that produces a `png.klib` compiled library:

```
cinterop -def png.def -compiler-option -I/usr/local/include -o png
```

Note that the generated bindings are generally platform-specific, so if you are developing for multiple targets, the bindings need to be regenerated.

- For host libraries that are not included in the sysroot search paths, headers may be needed.
- For a typical UNIX library with a configuration script, the `compilerOpts` will likely contain the output of a configuration script with the `--cflags` option (maybe without exact paths).
- The output of a configuration script with `--libs` can be passed to the `linkerOpts` property.

下一步做什么

- Bindings for C-interoperability
- Interoperability with Swift/Objective-C

与 C 语言互操作

- 与 C 语言互操作性
- 映射来自 C 语言的原始数据类型——教程
- 映射来自 C 语言的结构与联合类型——教程
- 映射来自 C 语言的函数指针——教程
- 映射来自 C 语言的字符串——教程
- Kotlin/Native 开发动态库——教程
- 创建使用 C 语言互操作与 libcurl 的应用——教程

与 C 语言互操作性

> The C libraries import is Experimental. All Kotlin declarations generated by the cinterop tool from C libraries should have the `@ExperimentalForeignApi` annotation.
>
> Native platform libraries shipped with Kotlin/Native (like Foundation, UIKit, and POSIX) require opt-in only for some APIs.
>
> 

This document covers general aspects of Kotlin's interoperability with C. Kotlin/Native comes with a cinterop tool, which you can use to quickly generate everything you need to interact with an external C library.

The tool analyzes C headers and produces a straightforward mapping of C types, functions, and strings into Kotlin. The generated stubs then can be imported into an IDE to enable code completion and navigation.

> Kotlin also provides interoperability with Objective-C. Objective-C libraries are imported through the cinterop tool as well. For more details, see Swift/Objective-C interop.
>
> 

Setting up your project

Here's a general workflow when working with a project that needs to consume a C library:

1. Create and configure a definition file. It describes what the cinterop tool should include into Kotlin bindings.
2. Configure your Gradle build file to include cinterop in the build process.
3. Compile and run the project to produce the final executable.

> For a hands-on experience, complete the Create an app using C interop tutorial.
>
> 

In many cases, there's no need to configure custom interoperability with a C library. Instead, you can use APIs available on the platform standardized bindings called platform libraries. For example, POSIX on Linux/macOS platforms, Win32 on the Windows platform, or Apple frameworks on macOS/iOS are available this way.

Bindings

Basic interop types

All the supported C types have corresponding representations in Kotlin:

- Signed, unsigned integral, and floating point types are mapped to their Kotlin counterpart with the same width.
- Pointers and arrays are mapped to `CPointer<T>?`.
- Enums can be mapped to either Kotlin enum or integral values, depending on heuristics and the definition file settings.
- Structs and unions are mapped to types having fields available via the dot notation, i.e. `someStructInstance.field1`.
- `typedef` are represented as `typealias`.

Also, any C type has the Kotlin type representing the lvalue of this type, i.e., the value located in memory rather than a simple immutable self-contained value. Think C++ references as a similar concept. For structs (and `typedef`s to structs), this representation is the main one and has the same name as the struct itself. For Kotlin enums, it's named `${type}.Var`; for `CPointer<T>`, it's `CPointerVar<T>`; and for most other types, it's `${type}Var`.

For types that have both representations, the one with the lvalue has a mutable `.value` property for accessing the value.

Pointer types

The type argument `T` of `CPointer<T>` must be one of the lvalue types described above. For example, the C type `struct S*` is mapped to `CPointer<S>`, `int8_t*` is mapped to `CPointer<int_8tVar>`, and `char**` is mapped to `CPointer<CPointerVar<ByteVar>>`.

C null pointer is represented as Kotlin's `null`, and the pointer type `CPointer<T>` is not nullable, but the `CPointer<T>?` is. The values of this type support all the Kotlin operations related to handling `null`, for example, `?:`, `?.`, `!!`, and so on:

```
val path = getenv("PATH")?.toKString() ?: ""
```

Since the arrays are also mapped to `CPointer<T>`, it supports the `[]` operator for accessing values by index:

```
import kotlinx.cinterop.*

@OptIn(ExperimentalForeignApi::class)
fun shift(ptr: CPointer<ByteVar>, length: Int) {
    for (index in 0 .. length - 2) {
        ptr[index] = ptr[index + 1]
    }
```

```
}
```

The `.pointed` property for `CPointer<T>` returns the lvalue of type `T`, pointed by this pointer. The reverse operation is `.ptr`, it takes the lvalue and returns the pointer to it.

`void*` is mapped to `COpaquePointer` – the special pointer type which is the supertype for any other pointer type. So if the C function takes `void*`, the Kotlin binding accepts any `CPointer`.

Casting a pointer (including `COpaquePointer`) can be done with `.reinterpret<T>`, for example:

```
import kotlinx.cinterop.*

@OptIn(ExperimentalForeignApi::class)
val intPtr = bytePtr.reinterpret<IntVar>()
```

Or:

```
import kotlinx.cinterop.*

@OptIn(ExperimentalForeignApi::class)
val intPtr: CPointer<IntVar> = bytePtr.reinterpret()
```

As is with C, these `.reinterpret` casts are unsafe and can potentially lead to subtle memory problems in the application.

Also, there are unsafe casts between `CPointer<T>?` and `Long` available, provided by the `.toLong()` and `.toCPointer<T>()` extension methods:

```
val longValue = ptr.toLong()
val originalPtr = longValue.toCPointer<T>()
```

> If the type of the result is known from the context, you can omit the type argument thanks to the type inference.
>
> 

Memory allocation

The native memory can be allocated using the `NativePlacement` interface, for example:

```
import kotlinx.cinterop.*

@OptIn(ExperimentalForeignApi::class)
val byteVar = placement.alloc<ByteVar>()
```

Or:

```
import kotlinx.cinterop.*

@OptIn(ExperimentalForeignApi::class)
val bytePtr = placement.allocArray<ByteVar>(5)
```

The most logical placement is in the object `nativeHeap`. It corresponds to allocating native memory with `malloc` and provides an additional `.free()` operation to free allocated memory:

```
import kotlinx.cinterop.*

@OptIn(kotlinx.cinterop.ExperimentalForeignApi::class)
fun main() {
    val size: Long = 0
    val buffer = nativeHeap.allocArray<ByteVar>(size)
    nativeHeap.free(buffer)
}
```

`nativeHeap` requires memory to be freed manually. However, it's often useful to allocate memory with a lifetime bound to the lexical scope. It's helpful if such memory is freed automatically.

To address this, you can use `memScoped { }`. Inside the braces, the temporary placement is available as an implicit receiver, so it's possible to allocate native memory with alloc and allocArray, and the allocated memory will be automatically freed after leaving the scope.

For example, a C function returning values through pointer parameters can be used like:

```
import kotlinx.cinterop.*
import platform.posix.*

@OptIn(ExperimentalForeignApi::class)
val fileSize = memScoped {
    val statBuf = alloc<stat>()
    val error = stat("/", statBuf.ptr)
    statBuf.st_size
}
```

Pass pointers to bindings

Although C pointers are mapped to the `CPointer<T> type`, C function pointer-typed parameters are mapped to `CValuesRef<T>.` When passing a `CPointer<T>` as a value of such a parameter, it's passed to the C function as is. However, a sequence of values can be passed instead of a pointer. In this case, the sequence is passed "by value", i.e., the C function receives the pointer to the temporary copy of that sequence, which is valid only until the function returns.

The `CValuesRef<T>` representation of pointer parameters is designed to support C array literals without explicit native memory allocation. To construct the immutable self-contained sequence of C values, the following methods are provided:

- `${type}Array.toCValues()`, where `type` is the Kotlin primitive type
- `Array<CPointer<T>?>.toCValues()`, `List<CPointer<T>?>.toCValues()`
- `cValuesOf(vararg elements: ${type})`, where `type` is a primitive or pointer

For example:

```
// C:
void foo(int* elements, int count);
...
int elements[] = {1, 2, 3};
foo(elements, 3);
```

```
// Kotlin:

foo(cValuesOf(1, 2, 3), 3)
```

Strings

Unlike other pointers, the parameters of type `const char*` are represented as a Kotlin `String`. So it's possible to pass any Kotlin string to a binding expecting a C string.

There are also some tools available to convert between Kotlin and C strings manually:

- `fun CPointer<ByteVar>.toKString(): String`
- `val String.cstr: CValuesRef<ByteVar>`.

To get the pointer, `.cstr` should be allocated in native memory, for example:

```
val cString = kotlinString.cstr.getPointer(nativeHeap)
```

In all cases, the C string is supposed to be encoded as UTF-8.

To skip automatic conversion and ensure raw pointers are used in the bindings, add the `noStringConversion` property to the `.def` file:

```
noStringConversion = LoadCursorA LoadCursorW
```

This way, any value of type `CPointer<ByteVar>` can be passed as an argument of `const char*` type. If a Kotlin string should be passed, code like this could be used:

```
import kotlinx.cinterop.*

@OptIn(kotlinx.cinterop.ExperimentalForeignApi::class)
memScoped {
    LoadCursorA(null, "cursor.bmp".cstr.ptr)  // for ASCII or UTF-8 version
    LoadCursorW(null, "cursor.bmp".wcstr.ptr) // for UTF-16 version
}
```

Scope-local pointers

It's possible to create a scope-stable pointer of C representation for the `CValues<T>` instance using the `CValues<T>.ptr` extension property, available under `memScoped {}`. It allows using APIs that require C pointers with a lifetime bound to a certain `MemScope`. For example:

```
import kotlinx.cinterop.*

@OptIn(kotlinx.cinterop.ExperimentalForeignApi::class)
memScoped {
    items = arrayOfNulls<CPointer<ITEM>?>(6)
    arrayOf("one", "two").forEachIndexed { index, value -> items[index] = value.cstr.p
tr }
    menu = new_menu("Menu".cstr.ptr, items.toCValues().ptr)
    // ...
}
```

In this example, all values passed to the C API `new_menu()` have a lifetime of the innermost `memScope` it belongs to. Once the control flow leaves the `memScoped` scope, C pointers become invalid.

Pass and receive structs by value

When a C function takes or returns a struct/union `T` by value, the corresponding argument type or return type is represented as `CValue<T>`.

`CValue<T>` is an opaque type, so the structure fields cannot be accessed with the appropriate Kotlin properties. This can be fine if an API uses structures as opaque handles. However, if field access is required, the following conversion methods are available:

- `fun T.readValue(): CValue<T>` converts (the lvalue) `T` to a `CValue<T>`. So to construct the `CValue<T>`, `T` can be allocated, filled, and then converted to `CValue<T>`.
- `CValue<T>.useContents(block: T.() -> R): R` temporarily stores the `CValue<T>` in memory, and then runs the passed lambda with this placed value `T` as receiver. So to read a single field, you can use the following code:

```
val fieldValue = structValue.useContents { field }
```

- `fun cValue(initialize: T.() -> Unit): CValue<T>` applies the provided `initialize` function to allocate `T` in memory and converts the result into a `CValue<T>`.
- `fun CValue<T>.copy(modify: T.() -> Unit): CValue<T>` creates a modified copy of an existing `CValue<T>`. The original value is placed in memory, altered using the `modify()` function, and then converted back into a new `CValue<T>`.
- `fun CValues<T>.placeTo(scope: AutofreeScope): CPointer<T>` places the `CValues<T>` into an `AutofreeScope`, returning a pointer to the allocated memory. The allocated memory is automatically freed when the `AutofreeScope` is disposed.

Callbacks

To convert a Kotlin function to a pointer to a C function, you can use `staticCFunction(::kotlinFunction)`. It's also possible to provide a lambda instead of a function reference. The function or lambda must not capture any values.

Pass user data to callbacks

Often C APIs allow passing some user data to callbacks. Such data is usually provided by the user when configuring the callback. It's passed to some C function (or written to the struct) as `void*`, for example. However, references to Kotlin objects can't be directly passed to C. So they require wrapping before configuring the callback and then unwrapping in the callback itself, to safely swim from Kotlin to Kotlin through the C world. Such wrapping is possible with the `StableRef` class.

To wrap the reference:

```
import kotlinx.cinterop.*

@OptIn(ExperimentalForeignApi::class)
val stableRef = StableRef.create(kotlinReference)
val voidPtr = stableRef.asCPointer()
```

Here, the `voidPtr` is a `COpaquePointer` and can be passed to the C function.

To unwrap the reference:

```
@OptIn(ExperimentalForeignApi::class)
val stableRef = voidPtr.asStableRef<KotlinClass>()
val kotlinReference = stableRef.get()
```

Here, `kotlinReference` is the original wrapped reference.

The created `StableRef` eventually be manually disposed using the `.dispose()` method to prevent memory leaks:

```
stableRef.dispose()
```

After that it becomes invalid, so `voidPtr` can't be unwrapped anymore.

Macros

Every C macro that expands to a constant is represented as a Kotlin property.

Macros without parameters are supported in cases when the compiler can infer the type:

```
int foo(int);
#define FOO foo(42)
```

In this case, `FOO` is available in Kotlin.

To support other macros, you can expose them manually by wrapping them with supported declarations. For example, function-like macro `FOO` can be exposed as a function `foo()` by adding custom declaration to the library:

```
headers = library/base.h

---

static inline int foo(int arg) {
    return FOO(arg);
}
```

Portability

Sometimes the C libraries have function parameters or struct fields of a platform-dependent type, for example, `long` or `size_t`. Kotlin itself doesn't provide either implicit integer casts or C-style integer casts (for example, `(size_t) intValue`), so to make writing portable code in such cases easier, the `convert` method is provided:

```
fun ${type1}.convert<${type2}>(): ${type2}
```

Here, each of `type1` and `type2` must be an integral type, either signed or unsigned.

`.convert<${type}>` has the same semantics as one of the `.toByte`, `.toShort`, `.toInt`, `.toLong`, `.toUByte`, `.toUShort`, `.toUInt` or `.toULong` methods, depending on `type`.

An example of using `convert`:

```
import kotlinx.cinterop.*
import platform.posix.*

@OptIn(ExperimentalForeignApi::class)
fun zeroMemory(buffer: COpaquePointer, size: Int) {
    memset(buffer, 0, size.convert<size_t>())
}
```

Also, the type parameter can be inferred automatically and so may be omitted in some cases.

Object pinning

Kotlin objects could be pinned, i.e. their position in memory is guaranteed to be stable until they are unpinned, and pointers to such objects' inner data could be passed to C functions.

There's a couple of approaches you can take:

- Use the `.usePinned()` extension function that pins an object, executes a block, and unpins it on normal and exception paths:

  ```
  import kotlinx.cinterop.*
  import platform.posix.*

  @OptIn(ExperimentalForeignApi::class)
  fun readData(fd: Int) {
      val buffer = ByteArray(1024)
      buffer.usePinned { pinned ->
          while (true) {
              val length = recv(fd, pinned.addressOf(0), buffer.size.convert(), 0).t
  oInt()
              if (length <= 0) {
                  break
              }
              // Now `buffer` has raw data obtained from the `recv()` call.
          }
      }
  }
  ```

  Here, `pinned` is an object of a special type `Pinned<T>`. It provides useful extensions like `.addressOf()`, which allows getting the address of a pinned array body.

- Use the `.refTo()` extension function that has similar functionality under the hood but, in certain cases, may help you reduce boilerplate code:

```
import kotlinx.cinterop.*
import platform.posix.*

@OptIn(ExperimentalForeignApi::class)
fun readData(fd: Int) {
    val buffer = ByteArray(1024)
    while (true) {
        val length = recv(fd, buffer.refTo(0), buffer.size.convert(), 0).toInt()

        if (length <= 0) {
            break
        }
        // Now `buffer` has raw data obtained from the `recv()` call.
    }
}
```

  Here, `buffer.refTo(0)` has the `CValuesRef` type that pins the array before entering the `recv()` function, passes the address of its zeroth element to the function, and unpins the array after exiting.

Forward declarations

To import forward declarations, use the `cnames` package. For example, to import a `cstructName` forward declaration declared in a C library with a `library.package`, use a special forward declaration package: `import cnames.structs.cstructName`.

Consider two cinterop libraries: one that has a forward declaration of a struct and another with an actual implementation in another package:

```
// First C library
#include <stdio.h>

struct ForwardDeclaredStruct;

void consumeStruct(struct ForwardDeclaredStruct* s) {
    printf("Struct consumed\n");
}
```

```
// Second C library
// Header:
#include <stdlib.h>

struct ForwardDeclaredStruct {
```

```
    int data;
};

// Implementation:
struct ForwardDeclaredStruct* produceStruct() {
    struct ForwardDeclaredStruct* s = malloc(sizeof(struct ForwardDeclaredStruct));
    s->data = 42;
    return s;
}
```

To transfer objects between the two libraries, use an explicit `as` cast in your Kotlin code:

```
// Kotlin code:
fun test() {
    consumeStruct(produceStruct() as CPointer<cnames.structs.ForwardDeclaredStruct>)
}
```

下一步做什么

Learn how types, functions, and strings are mapped between Kotlin and C by completing the following tutorials:

- Mapping primitive data types from C
- Mapping struct and union types from C
- Mapping function pointers from C
- Mapping strings from C

映射来自 C 语言的原生数据类型——教程

This is the first part of the **Mapping Kotlin and C** tutorial series.

1 **Mapping primitive data types from C**

2 Mapping struct and union types from C

3 Mapping function pointers

4 Mapping strings from C

> The C libraries import is Experimental. All Kotlin declarations generated by the cinterop tool from C libraries should have the `@ExperimentalForeignApi` annotation.
>
> Native platform libraries shipped with Kotlin/Native (like Foundation, UIKit, and POSIX) require opt-in only for some APIs.
>
> 

Let's explore which C data types are visible in Kotlin/Native and vice versa and examine advanced C interop-related use cases of Kotlin/Native and multiplatform Gradle builds.

在本教程中会：

- 了解 C 语言中的数据类型
- 创建一个使用这些类型进行导出的 C 库
- 探查为 C 库生成的 Kotlin API

You can use the command line to generate a Kotlin library, either directly or with a script file (such as `.sh` or `.bat` file). However, this approach doesn't scale well for larger projects that have hundreds of files and libraries. Using a build system simplifies the process by downloading and caching the

Kotlin/Native compiler binaries and libraries with transitive dependencies, as well as by running the compiler and tests. Kotlin/Native can use the Gradle build system through the Kotlin Multiplatform plugin.

C 语言中的类型

C 编程语言有以下数据类型：

- 基本类型：`char`、`int`、`float`、`double` 以及带修饰符的 `signed`、`unsigned`、`short`、`long`
- 结构体、联合体、数组
- 指针
- 函数指针

还有一些更多的具体类型：

- 布尔类型（源于 C99）
- `size_t` 与 `ptrdiff_t`（也作 `ssize_t`）
- 固定长度的整型例如：`int32_t` 或 `uint64_t`（源于 C99）

C 语言中还有以下类型限定符：`const`、`volatile`、`restrict`、`atomic`。

Let's see which C data types are visible in Kotlin.

Create a C library

In this tutorial, you won't create a `lib.c` source file, which is only necessary if you want to compile and run your C library. For this setup, you'll only need a `.h` header file that is required for running the cinterop tool.

The cinterop tool generates a Kotlin/Native library (a `.klib` file) for each set of `.h` files. The generated library helps bridge calls from Kotlin/Native to C. It includes Kotlin declarations that correspond to the definitions from the `.h` files.

To create a C library:

1. Create an empty folder for your future project.
2. Inside, create a `lib.h` file with the following content to see how C functions are mapped into Kotlin:

```
#ifndef LIB2_H_INCLUDED
#define LIB2_H_INCLUDED

void ints(char c, short d, int e, long f);
void uints(unsigned char c, unsigned short d, unsigned int e, unsigned long f);
```

```
void doubles(float a, double b);

#endif
```

该文件没有 `extern "C"` 块，在本例中并不需要，但如果使用 C++并重载函数，那么可能是必要的。请参见 Stackoverflow 这贴了解更多细节。

3. Create the `lib.def` definition file with the following content:

```
headers = lib.h
```

4. 将宏或其他 C 定义包含在 cinterop 工具生成的代码中会很有帮助。这样， 方法体同样被编译以及完全包含到二进制文件中。使用这个特性，可以创建一个无需 C 编译器的可运行示例。

5. 为做到这点，在新 `interop.def` 文件中 `---` 分隔符之后，为 `lib.h` 文件中的 C 函数添加实现：

```

---

void ints(char c, short d, int e, long f) { }
void uints(unsigned char c, unsigned short d, unsigned int e, unsigned long f) { }
void doubles(float a, double b) { }
```

`interop.def` 文件提供了在 IDE 中编译、运行或打开应用程序所需的一切。

Create a Kotlin/Native project

> See the Get started with Kotlin/Native tutorial for detailed first steps and instructions on how to create a new Kotlin/Native project and open it in IntelliJ IDEA.
>
> 

To create project files:

1. In your project folder, create a `build.gradle(.kts)` Gradle build file with the following content:

> 【Kotlin】

```
plugins {
```

```
        kotlin("multiplatform") version "2.1.21"
    }

    repositories {
        mavenCentral()
    }

    kotlin {
        macosArm64("native") {    // macOS on Apple Silicon
        // macosX64("native") {   // macOS on x86_64 platforms
        // linuxArm64("native") { // Linux on ARM64 platforms
        // linuxX64("native") {   // Linux on x86_64 platforms
        // mingwX64("native") {   // on Windows
            val main by compilations.getting
            val interop by main.cinterops.creating

            binaries {
                executable()
            }
        }
    }

    tasks.wrapper {
        gradleVersion = "8.10"
        distributionType = Wrapper.DistributionType.BIN
    }
```

> 【Groovy】

```
    plugins {
        id 'org.jetbrains.kotlin.multiplatform' version '2.1.21'
    }

    repositories {
        mavenCentral()
    }

    kotlin {
        macosArm64("native") {    // Apple Silicon macOS
        // macosX64("native") {   // macOS on x86_64 platforms
        // linuxArm64("native") { // Linux on ARM64 platforms
        // linuxX64("native") {   // Linux on x86_64 platforms
        // mingwX64("native") {   // Windows
            compilations.main.cinterops {
                interop
            }

            binaries {
                executable()
            }
```

```
        }
    }

    wrapper {
        gradleVersion = '8.10'
        distributionType = 'BIN'
    }
```

The project file configures the C interop as an additional build step. Check out the Multiplatform Gradle DSL reference to learn about different ways you can configure it.

1. Move your `interop.def`, `lib.h`, and `lib.def` files to the `src/nativeInterop/cinterop` directory.
2. Create a `src/nativeMain/kotlin` directory. This is where you should place all the source files, following Gradle's recommendations on using conventions instead of configurations.

   By default, all the symbols from C are imported to the `interop` package.

3. 在 `src/nativeMain/kotlin` 中，使用以下内容创建一个 `hello.kt` 存根文件：

   ```
   import interop.*
   import kotlinx.cinterop.ExperimentalForeignApi

   @OptIn(ExperimentalForeignApi::class)
   fun main() {
       println("Hello Kotlin/Native!")

       ints(/* fix me*/)
       uints(/* fix me*/)
       doubles(/* fix me*/)
   }
   ```

You'll complete the code later as you learn how C primitive type declarations look from the Kotlin side.

Inspect generated Kotlin APIs for a C library

Let's see how C primitive types are mapped into Kotlin/Native and update the example project accordingly.

Use IntelliJ IDEA's Go to declaration command (Cmd + B/Ctrl + B) to navigate to the following generated API for C functions:

```
fun ints(c: kotlin.Byte, d: kotlin.Short, e: kotlin.Int, f: kotlin.Long)
```

```
fun uints(c: kotlin.UByte, d: kotlin.UShort, e: kotlin.UInt, f: kotlin.ULong)
fun doubles(a: kotlin.Float, b: kotlin.Double)
```

C 语言类型直映射，除了 `char` 类型，它映射到了 `kotlin.Byte` ，因为它通常是 8 位有符号值：

C	Kotlin
char	kotlin.Byte
unsigned char	kotlin.UByte
short	kotlin.Short
unsigned short	kotlin.UShort
int	kotlin.Int
unsigned int	kotlin.UInt
long long	kotlin.Long
unsigned long long	kotlin.ULong
float	kotlin.Float
double	kotlin.Double

Update Kotlin code

Now that you've seen the C definitions, you can update your Kotlin code. `hello.kt` 文件中的代码最终看起来会是这样的：

```
import interop.*
import kotlinx.cinterop.ExperimentalForeignApi

@OptIn(ExperimentalForeignApi::class)
fun main() {
    println("Hello Kotlin/Native!")

    ints(1, 2, 3, 4)
    uints(5u, 6u, 7u, 8u)
    doubles(9.0f, 10.0)
}
```

To verify that everything works as expected, run the `runDebugExecutableNative` Gradle task in your IDE or use the following command to run the code:

```
./gradlew runDebugExecutableNative
```

接下来

In the next part of the series, you'll learn how struct and union types are mapped between Kotlin and C:

Proceed to the next part

See also

Learn more in the Interoperability with C documentation that covers more advanced scenarios.

映射来自 C 语言的结构与联合类型——教程

This is the second part of the **Mapping Kotlin and C** tutorial series. Before proceeding, make sure you've completed the previous step.

1. Mapping primitive data types from C

2. **Mapping struct and union types from C**

3. Mapping function pointers

4. Mapping strings from C

> The C libraries import is Experimental. All Kotlin declarations generated by the cinterop tool from C libraries should have the `@ExperimentalForeignApi` annotation.
>
> Native platform libraries shipped with Kotlin/Native (like Foundation, UIKit, and POSIX) require opt-in only for some APIs.
>
> 

Let's explore which C struct and union declarations are visible from Kotlin and examine advanced C interop-related use cases of Kotlin/Native and multiplatform Gradle builds.

在本教程中可以学到

- 如何映射结构与联合类型
- 在 Kotlin 中如何使用结构与联合类型

映射 C 语言的结构与联合类型

To understand how Kotlin maps struct and union types, let's declare them in C and examine how they are represented in Kotlin.

在之前的教程中，你已经用必要的文件创建了一个 C 库。 对于这一步，请更新 `interop.def` 文件中 `---` 分隔符之后的声明：

```
---

typedef struct {
  int a;
  double b;
} MyStruct;

void struct_by_value(MyStruct s) {}
void struct_by_pointer(MyStruct* s) {}

typedef union {
  int a;
  MyStruct b;
  float c;
} MyUnion;

void union_by_value(MyUnion u) {}
void union_by_pointer(MyUnion* u) {}
```

`interop.def` 文件提供了在 IDE 中编译、运行或打开应用程序所需的一切。

探查为 C 库生成的 Kotlin API

Let's see how C struct and union types are mapped into Kotlin/Native and update your project:

1. In `src/nativeMain/kotlin`, update your `hello.kt` file from the previous tutorial with the following content:

   ```
   import interop.*
   import kotlinx.cinterop.ExperimentalForeignApi

   @OptIn(ExperimentalForeignApi::class)
   fun main() {
       println("Hello Kotlin/Native!")

       struct_by_value(/* fix me*/)
       struct_by_pointer(/* fix me*/)
       union_by_value(/* fix me*/)
       union_by_pointer(/* fix me*/)
   }
   ```

2. To avoid compiler errors, add interoperability to the build process. For that, update your `build.gradle(.kts)` build file with the following content:

> 【Kotlin】

```
    kotlin {
        macosArm64("native") {    // macOS on Apple Silicon
        // macosX64("native") {   // macOS on x86_64 platforms
        // linuxArm64("native") { // Linux on ARM64 platforms
        // linuxX64("native") {   // Linux on x86_64 platforms
        // mingwX64("native") {   // on Windows
            val main by compilations.getting
            val interop by main.cinterops.creating {
                definitionFile.set(project.file("src/nativeInterop/cinterop/interop.de
f"))
            }

            binaries {
                executable()
            }
        }
    }
```

> 【Groovy】

```
    kotlin {
        macosArm64("native") {    // Apple Silicon macOS
        // macosX64("native") {   // macOS on x86_64 platforms
        // linuxArm64("native") { // Linux on ARM64 platforms
        // linuxX64("native") {   // Linux on x86_64 platforms
        // mingwX64("native") {   // Windows
            compilations.main.cinterops {
                interop {
                    definitionFile = project.file('src/nativeInterop/cinterop/interop.
def')
                }
            }

            binaries {
                executable()
            }
        }
    }
```

1. Use IntelliJ IDEA's Go to declaration command (Cmd + B/Ctrl + B) to navigate to the following generated API for C functions, struct, and union:

```
fun struct_by_value(s: kotlinx.cinterop.CValue<interop.MyStruct>)
fun struct_by_pointer(s: kotlinx.cinterop.CValuesRef<interop.MyStruct>?)

fun union_by_value(u: kotlinx.cinterop.CValue<interop.MyUnion>)
fun union_by_pointer(u: kotlinx.cinterop.CValuesRef<interop.MyUnion>?)
```

从技术上讲，在 Kotlin 看来结构与联合类型之间没有区别。 The cinterop tool generates Kotlin types for both struct and union C declarations.

The generated API includes fully qualified package names for `CValue<T>` and `CValuesRef<T>`, reflecting their location in `kotlinx.cinterop`. `CValue<T>` 表示一个值类型的结构体参数，而 `CValuesRef<T>?` 用于传递一个结构体或联合体的指针。

在 Kotlin 中使用结构与联合类型

Using C struct and union types from Kotlin is straightforward thanks to the generated API. The only question is how to create new instances of these types.

我们来看一看生成的函数，它将 `MyStruct` 与 `MyUnion` 作为参数。值类型参数表示为 `kotlinx.cinterop.CValue<T>`，而指针类型参数使用 `kotlinx.cinterop.CValuesRef<T>?`。

Kotlin 提供了一种便利的 API 来创建及处理这些类型。我们来探索如何在实践中使用它。

创建一个 CValue<T>

`CValue<T>` 类型用来传递一个值类型的参数到 C 函数调用。使用 `cValue` 函数来创建 `CValue<T>` 实例。该函数需要一个带接收者的 lambda 函数字面值来就地初始化底层 C 类型。该函数的声明如下所示：

```
fun <reified T : CStructVar> cValue(initialize: T.() -> Unit): CValue<T>
```

以下是如何使用 `cValue` 并传递值类型参数：

```
import interop.*
import kotlinx.cinterop.ExperimentalForeignApi
import kotlinx.cinterop.cValue

@OptIn(ExperimentalForeignApi::class)
fun callValue() {

    val cStruct = cValue<MyStruct> {
```

```
        a = 42
        b = 3.14
    }
    struct_by_value(cStruct)

    val cUnion = cValue<MyUnion> {
        b.a = 5
        b.b = 2.7182
    }

    union_by_value(cUnion)
}
```

使用 CValuesRef<T> 创建结构体与联合体

`CValuesRef<T>` 类型用于在 Kotlin 中将指针类型的参数传递给 C 函数。如需在原生内存中分配 `MyStruct` 与 `MyUnion`，请使用 `kotlinx.cinterop.NativePlacement` 类型的以下扩展函数：

```
fun <reified T : kotlinx.cinterop.CVariable> alloc(): T
```

`NativePlacement` 代表原生内存，类似于 `malloc` 与 `free` 函数。 这里有几个 `NativePlacement` 的实现：

- 全局的实现是 `kotlinx.cinterop.nativeHeap`，但你必须在使用过后调用 `nativeHeap.free()` 来释放内存。
- A safer alternative is `memScoped()`, which creates a short-lived memory scope where all allocations are automatically freed at the end of the block:

  ```
  fun <R> memScoped(block: kotlinx.cinterop.MemScope.() -> R): R
  ```

使用 `memScoped()` 时，调用带指针类型参数的函数的代码看起来会是这样：

```
import interop.*
import kotlinx.cinterop.ExperimentalForeignApi
import kotlinx.cinterop.memScoped
import kotlinx.cinterop.alloc
import kotlinx.cinterop.ptr

@OptIn(ExperimentalForeignApi::class)
fun callRef() {
    memScoped {
        val cStruct = alloc<MyStruct>()
        cStruct.a = 42
        cStruct.b = 3.14
```

```
        struct_by_pointer(cStruct.ptr)

        val cUnion = alloc<MyUnion>()
        cUnion.b.a = 5
        cUnion.b.b = 2.7182

        union_by_pointer(cUnion.ptr)
    }
}
```

这里的 `ptr` 扩展属性可在 `memScoped {}` 代码块中使用，它将 `MyStruct` 和 `MyUnion` 实例转换为原生指针。

Since memory is managed inside the `memScoped {}` block, it's automatically freed at the end of the block. Avoid using pointers outside of this scope to prevent accessing deallocated memory. If you need longer-lived allocations (for example, for caching in a C library), consider using `Arena()` or `nativeHeap`.

在 CValue<T> 与 CValuesRef<T> 之间转换

有时候，需要在一个函数调用中将结构体作为值传递，然后在另一个调用中将同一个结构体作为引用传递。

To do this, you'll need a `NativePlacement`, 但首先，我们来看看如何将 `CValue<T>` 转换为一个指针：

```
import interop.*
import kotlinx.cinterop.ExperimentalForeignApi
import kotlinx.cinterop.cValue
import kotlinx.cinterop.memScoped

@OptIn(ExperimentalForeignApi::class)
fun callMix_ref() {
    val cStruct = cValue<MyStruct> {
        a = 42
        b = 3.14
    }

    memScoped {
        struct_by_pointer(cStruct.ptr)
    }
}
```

这里再次说明，来自 `memScoped {}` 的 `ptr` 扩展属性将 `MyStruct` 实例转换为原生指针。 这些指针只在 `memScoped {}` 块内有效。

如需将指针转换回值类型变量，请调用 `.readValue()` 扩展函数：

```
import interop.*
import kotlinx.cinterop.alloc
import kotlinx.cinterop.ExperimentalForeignApi
import kotlinx.cinterop.memScoped
import kotlinx.cinterop.readValue

@OptIn(ExperimentalForeignApi::class)
fun callMix_value() {
    memScoped {
        val cStruct = alloc<MyStruct>()
        cStruct.a = 42
        cStruct.b = 3.14

        struct_by_value(cStruct.readValue())
    }
}
```

Update Kotlin code

Now that you've learned how to use C declarations in Kotlin code, try to use them in your project.

`hello.kt` 文件中的最终代码看起来会是这样：

```
import interop.*
import kotlinx.cinterop.alloc
import kotlinx.cinterop.cValue
import kotlinx.cinterop.memScoped
import kotlinx.cinterop.ptr
import kotlinx.cinterop.readValue
import kotlinx.cinterop.ExperimentalForeignApi

@OptIn(ExperimentalForeignApi::class)
fun main() {
    println("Hello Kotlin/Native!")

    val cUnion = cValue<MyUnion> {
        b.a = 5
        b.b = 2.7182
    }

    memScoped {
        union_by_value(cUnion)
        union_by_pointer(cUnion.ptr)
    }

    memScoped {
        val cStruct = alloc<MyStruct> {
            a = 42
            b = 3.14
        }
```

```
        struct_by_value(cStruct.readValue())
        struct_by_pointer(cStruct.ptr)
    }
}
```

To verify that everything works as expected, run the `runDebugExecutableNative` Gradle task in your IDE or use the following command to run the code:

```
./gradlew runDebugExecutableNative
```

接下来

In the next part of the series, you'll learn how function pointers are mapped between Kotlin and C:

Proceed to the next part

See also

Learn more in the Interoperability with C documentation that covers more advanced scenarios.

映射来自 C 语言的函数指针——教程

This is the third part of the **Mapping Kotlin and C** tutorial series. Before proceeding, make sure you've completed the previous steps.

1. Mapping primitive data types from C

2. Mapping struct and union types from C

3. **Mapping function pointers**

4. Mapping strings from C

> The C libraries import is Experimental. All Kotlin declarations generated by the cinterop tool from C libraries should have the `@ExperimentalForeignApi` annotation.
>
> Native platform libraries shipped with Kotlin/Native (like Foundation, UIKit, and POSIX) require opt-in only for some APIs.
>
> 

Let's explore which C function pointers are visible from Kotlin and examine advanced C interop-related use cases of Kotlin/Native and multiplatform Gradle builds.

在本教程中会：

- 学习如何将 Kotlin 函数作为 C 函数指针传递
- 在 Kotlin 中使用 C 函数指针

映射 C 中的函数指针类型

理解在 Kotlin 与 C 之间进行映射的方式是，我们来声明两个函数：一个接收函数指针作为参数，另一个返回一个函数指针。

In the first part of the series of the series, you've already created a C library with the necessary files. For this step, update the declarations in the `interop.def` file after the `---` separator:

```
---

int myFun(int i) {
  return i+1;
}

typedef int  (*MyFun)(int);

void accept_fun(MyFun f) {
  f(42);
}

MyFun supply_fun() {
  return myFun;
}
```

`interop.def` 文件提供了在 IDE 中编译、运行或打开应用程序所需的一切。

探查为 C 库生成的 Kotlin API

Let's see how C function pointers are mapped into Kotlin/Native and update your project:

1. In `src/nativeMain/kotlin`, update your `hello.kt` file from the previous tutorial with the following content:

   ```
   import interop.*
   import kotlinx.cinterop.ExperimentalForeignApi

   @OptIn(ExperimentalForeignApi::class)
   fun main() {
       println("Hello Kotlin/Native!")

       accept_fun(/* fix me*/)
       val useMe = supply_fun()
   }
   ```

2. Use IntelliJ IDEA's Go to declaration command (Cmd + B/Ctrl + B) to navigate to the following generated API for C functions:

```
fun myFun(i: kotlin.Int): kotlin.Int
fun accept_fun(f: kotlinx.cinterop.CPointer<kotlinx.cinterop.CFunction<(kotlin.Int
) -> kotlin.Int>>? /* from: interop.MyFun? */)
fun supply_fun(): kotlinx.cinterop.CPointer<kotlinx.cinterop.CFunction<(kotlin.Int
) -> kotlin.Int>>? /* from: interop.MyFun? */
```

As you can see, C function pointers are represented in Kotlin using `CPointer<CFunction<...>>`. The `accept_fun()` function takes an optional function pointer as a parameter, while `supply_fun()` returns a function pointer.

`CFunction<(Int) -> Int>` represents the function signature, and `CPointer<CFunction<...>>?` represents a nullable function pointer. There is an `.invoke()` operator extension function available for all `CPointer<CFunction<...>>` types, allowing you to call function pointers as if they were regular Kotlin functions.

将 Kotlin 函数作为 C 函数指针传递

是时候尝试在 Kotlin 代码中使用 C 函数了。调用 `accept_fun()` 函数并传递 C 函数指针到一个 Kotlin lambda 表达式：

```
import interop.*
import kotlinx.cinterop.staticCFunction
import kotlinx.cinterop.ExperimentalForeignApi

@OptIn(ExperimentalForeignApi::class)
fun myFun() {
    accept_fun(staticCFunction<Int, Int> { it + 1 })
}
```

该调用使用 Kotlin/Native 中的 `staticCFunction {}` 辅助函数将一个 Kotlin lambda 函数包装为 C 函数指针。它只能是非绑定的以及没有发生变量捕捉的 lambda functions。举例来说，它不能捕获函数中的局部变量，只能捕获全局可见的声明。

Ensure that the function doesn't throw any exceptions. 在 `staticCFunction {}` 中抛出异常会导致非确定性的副作用。

在 Kotlin 中使用 C 函数指针

接下来是调用 `supply_fun()` 返回的 C 函数指针：

```
import interop.*
import kotlinx.cinterop.ExperimentalForeignApi
```

```
import kotlinx.cinterop.invoke

@OptIn(ExperimentalForeignApi::class)
fun myFun2() {
    val functionFromC = supply_fun() ?: error("No function is returned")

    functionFromC(42)
}
```

Kotlin 将函数指针返回类型转换为一个可空的 `CPointer<CFunction<>>` 对象。需要首先显式检测 `null` 值，这就是为什么上述代码中使用了 Elvis 操作符。 `cinterop` 工具让你可以像调用常规 Kotlin 函数一样调用 C 函数指针： `functionFromC(42)` 。

Update Kotlin code

Now that you've seen all the definitions, try to use them in your project. `hello.kt` 文件中的代码最终看起来会是这样的：

```
import interop.*
import kotlinx.cinterop.ExperimentalForeignApi
import kotlinx.cinterop.invoke
import kotlinx.cinterop.staticCFunction

@OptIn(ExperimentalForeignApi::class)
fun main() {
    println("Hello Kotlin/Native!")

    val cFunctionPointer = staticCFunction<Int, Int> { it + 1 }
    accept_fun(cFunctionPointer)

    val funFromC = supply_fun() ?: error("No function is returned")
    funFromC(42)
}
```

To verify that everything works as expected, run the `runDebugExecutableNative` Gradle task in your IDE or use the following command to run the code:

```
./gradlew runDebugExecutableNative
```

接下来

In the next part of the series, you'll learn how strings are mapped between Kotlin and C:

Proceed to the next part

See also

Learn more in the Interoperability with C documentation that covers more advanced scenarios.

映射来自 C 语言的字符串——教程

This is the final part of the **Mapping Kotlin and C** tutorial series. Before proceeding, make sure you've completed the previous steps.

① Mapping primitive data types from C

② Mapping struct and union types from C

③ Mapping function pointers

④ **Mapping strings from C**

> The C libraries import is Experimental. All Kotlin declarations generated by the cinterop tool from C libraries should have the `@ExperimentalForeignApi` annotation.
>
> Native platform libraries shipped with Kotlin/Native (like Foundation, UIKit, and POSIX) require opt-in only for some APIs.
>
> 

In the final part of the series, let's see how to deal with C strings in Kotlin/Native.

在本教程中会学到如何：

-
- 传递一个 Kotlin 字符串给 C
- 在 Kotlin 中读取一个 C 字符串
- 接收 C 字符串字节并转换为 Kotlin 字符串

使用 C 语言字符串

C 语言没有专门的字符串类型。方法签名或文档可以帮助你识别给定的 `char *` 在特定上下文中是否表示 C 字符串。

C 语言中的字符串以空值终止，因此会在字节序列末尾添加零字符 `\0` 来标记字符串的结束。通常，使用 UTF-8 编码字符串。 UTF-8 编码使用可变宽度字符，并且向后兼容 ASCII。Kotlin/Native 默认使用 UTF-8 字符编码。

To understand how strings are mapped between Kotlin and C, first create the library headers. In the first part of the series, you've already created a C library with the necessary files. For this step:

1. Update your `lib.h` file with the following function declarations that work with C strings:

   ```
   #ifndef LIB2_H_INCLUDED
   #define LIB2_H_INCLUDED

   void pass_string(char* str);
   char* return_string();
   int copy_string(char* str, int size);

   #endif
   ```

   此示例展示了在 C 语言中传递或接收字符串的常见方式。请谨慎处理 `return_string()` 函数的返回值。确保使用正确的 `free()` 函数来释放返回的 `char*` 。

2. Update the declarations in the `interop.def` file after the `---` separator:

   ```
   ---

   void pass_string(char* str) {
   }

   char* return_string() {
     return "C string";
   }

   int copy_string(char* str, int size) {
       *str++ = 'C';
       *str++ = ' ';
       *str++ = 'K';
       *str++ = '/';
       *str++ = 'N';
       *str++ = 0;
       return 0;
   }
   ```

`interop.def` 文件提供了在 IDE 中编译、运行或打开应用程序所需的一切。

探查为 C 库生成的 Kotlin API

Let's see how C string declarations are mapped into Kotlin/Native:

1. In `src/nativeMain/kotlin`, update your `hello.kt` file from the previous tutorial with the following content:

```
import interop.*
import kotlinx.cinterop.ExperimentalForeignApi

@OptIn(ExperimentalForeignApi::class)
fun main() {
    println("Hello Kotlin/Native!")

    pass_string(/*fix me*/)
    val useMe = return_string()
    val useMe2 = copy_string(/*fix me*/)
}
```

2. Use IntelliJ IDEA's Go to declaration command (Cmd + B/Ctrl + B) to navigate to the following generated API for C functions:

```
fun pass_string(str: kotlinx.cinterop.CValuesRef<kotlinx.cinterop.ByteVarOf<kotlin.
Byte> /* from: kotlinx.cinterop.ByteVar */>?)
fun return_string(): kotlinx.cinterop.CPointer<kotlinx.cinterop.ByteVarOf<kotlin.B
yte> /* from: kotlinx.cinterop.ByteVar */>?
fun copy_string(str: kotlinx.cinterop.CValuesRef<kotlinx.cinterop.ByteVarOf<kotlin.
Byte> /* from: kotlinx.cinterop.ByteVar */>?, size: kotlin.Int): kotlin.Int
```

这些声明看起来很清晰。所有的 `char *` 指针在参数处都被转换为 `str: CValuesRef<ByteVar>?` 而返回值类型则被转换为 `CPointer<ByteVar>?`。Kotlin 将 `char` 类型转换为 `kotlin.Byte` 类型， 因为它通常是 8 位有符号值。

在生成的 Kotlin 声明中， `str` 会定义为 `CValuesRef<ByteVarOf<Byte>>?`。 由于该类型是可空的，可以将 `null` 作为参数值传入。

将 Kotlin 字符串传递给 C

我们来尝试在 Kotlin 中使用这些 API。首先调用 `pass_string()` 函数：

```
import interop.*
import kotlinx.cinterop.ExperimentalForeignApi
import kotlinx.cinterop.cstr
```

```
@OptIn(ExperimentalForeignApi::class)
fun passStringToC() {
    val str = "This is a Kotlin string"
    pass_string(str.cstr)
}
```

将 Kotlin 字符串传递到 C 语言也很直观，得益于 `String.cstr` 扩展属性。 对于涉及 UTF-16 字符的情况，也有 `String.wcstr` 属性。

在 Kotlin 中读取 C 字符串

现在从 `return_string()` 函数获取一个返回的 `char *` 并将其转换为 Kotlin 字符串：

```
import interop.*
import kotlinx.cinterop.ExperimentalForeignApi
import kotlinx.cinterop.toKString

@OptIn(ExperimentalForeignApi::class)
fun passStringToC() {
    val stringFromC = return_string()?.toKString()

    println("Returned from C: $stringFromC")
}
```

Here, the `.toKString()` extension function converts a C string returned from the `return_string()` function into a Kotlin string.

Kotlin provides several extension functions for converting C `char *` strings into Kotlin strings, depending on the encoding:

```
fun CPointer<ByteVarOf<Byte>>.toKString(): String // Standard function for UTF-8 strin
gs
fun CPointer<ByteVarOf<Byte>>.toKStringFromUtf8(): String // Explicitly converts UTF-8
 strings
fun CPointer<ShortVarOf<Short>>.toKStringFromUtf16(): String // Converts UTF-16 encode
d strings
fun CPointer<IntVarOf<Int>>.toKStringFromUtf32(): String // Converts UTF-32 encoded st
rings
```

在 Kotlin 中接收 C 字符串字节

这次，使用 `copy_string()` C 函数将 C 字符串写入给定的缓冲区。它接受两个参数： 指向应写入字符串的内存位置的指针以及允许的缓冲区大小。

该函数还应返回一些内容以指示它是成功还是失败。我们假设 `0` 表示成功， 并且提供的缓冲区足够大：

```
import interop.*
import kotlinx.cinterop.ExperimentalForeignApi
import kotlinx.cinterop.addressOf
import kotlinx.cinterop.usePinned

@OptIn(ExperimentalForeignApi::class)
fun sendString() {
    val buf = ByteArray(255)
    buf.usePinned { pinned ->
        if (copy_string(pinned.addressOf(0), buf.size - 1) != 0) {
            throw Error("Failed to read string from C")
        }
    }

    val copiedStringFromC = buf.decodeToString()
    println("Message from C: $copiedStringFromC")
}
```

这里，首先将一个原生指针传递给 C 函数。 `.usePinned()` 扩展函数临时固定字节数组的原生内存地址。该 C 函数填充了带数据的字节数组。 另一个扩展函数 `ByteArray.decodeToString()` 将字节数组转换为一个 Kotlin 字符串，假设它是 UTF-8 编码的。

Update Kotlin code

Now that you've learned how to use C declarations in Kotlin code, try to use them in your project. `hello.kt` 文件中的代码最终看起来会是这样的：

```
import interop.*
import kotlinx.cinterop.*

@OptIn(ExperimentalForeignApi::class)
fun main() {
    println("Hello Kotlin/Native!")

    val str = "This is a Kotlin string"
    pass_string(str.cstr)

    val useMe = return_string()?.toKString() ?: error("null pointer returned")
    println(useMe)

    val copyFromC = ByteArray(255).usePinned { pinned ->
        val useMe2 = copy_string(pinned.addressOf(0), pinned.get().size - 1)
        if (useMe2 != 0) throw Error("Failed to read a string from C")
```

```
            pinned.get().decodeToString()
        }

        println(copyFromC)
    }
```

To verify that everything works as expected, run the `runDebugExecutableNative` Gradle task in your IDE or use the following command to run the code:

```
./gradlew runDebugExecutableNative
```

接下来

Learn more in the Interoperability with C documentation that covers more advanced scenarios.

Kotlin/Native 开发动态库——教程

You can create dynamic libraries to use Kotlin code from existing programs. This enables code sharing across many platforms or languages, including JVM, Python, Android, and others.

> For iOS and other Apple targets, we recommend generating a framework. See the Kotlin/Native as an Apple framework tutorial.
>
> 

You can use the Kotlin/Native code from existing native applications or libraries. For this, you need to compile the Kotlin code into a dynamic library in the `.so`, `.dylib`, or `.dll` format.

在这篇教程中，将会：

- 将 Kotlin 代码编译为动态库
- 生成 C 的头文件
- 在 C 中调用 Kotlin 动态库
- Compile and run the project

You can use the command line to generate a Kotlin library, either directly or with a script file (such as `.sh` or `.bat` file). However, this approach doesn't scale well for larger projects that have hundreds of files and libraries. Using a build system simplifies the process by downloading and caching the Kotlin/Native compiler binaries and libraries with transitive dependencies, as well as by running the compiler and tests. Kotlin/Native can use the Gradle build system through the Kotlin Multiplatform plugin.

Let's examine the advanced C interop-related usages of Kotlin/Native and Kotlin Multiplatform builds with Gradle.

> If you use a Mac and want to create and run applications for macOS or other Apple targets, you also need to install the Xcode Command Line Tools, launch it, and accept the license terms first.
>
> 

创建 Kotlin 库

Kotlin/Native 编译器可以将 Kotlin 代码编译为一个动态库。动态库常常带有一个 `.h` 头文件，可以通过它来调用编译后的 C 代码。

我们来创建一个 Kotlin 库，并在 C 程序中使用它。

> See the Get started with Kotlin/Native tutorial for detailed first steps and instructions on how to create a new Kotlin/Native project and open it in IntelliJ IDEA.
>
> 

1. Navigate to the `src/nativeMain/kotlin` directory and create the `lib.kt` file with the following library contents:

```
package example

object Object {
    val field = "A"
}

class Clazz {
    fun memberFunction(p: Int): ULong = 42UL
}

fun forIntegers(b: Byte, s: Short, i: UInt, l: Long) { }
fun forFloats(f: Float, d: Double) { }

fun strings(str: String) : String? {
    return "That is '$str' from C"
}

val globalString = "A global String"
```

2. Update your `build.gradle(.kts)` Gradle build file with the following:

> 【Kotlin】

```
plugins {
    kotlin("multiplatform") version "2.1.21"
}

repositories {
    mavenCentral()
}

kotlin {
    macosArm64("native") {    // macOS on Apple Silicon
    // macosX64("native") {   // macOS on x86_64 platforms
    // linuxArm64("native") { // Linux on ARM64 platforms
    // linuxX64("native") {   // Linux on x86_64 platforms
    // mingwX64("native") {   // Windows
        binaries {
            sharedLib {
                baseName = "native"       // macOS and Linux
```

```
                    // baseName = "libnative" // Windows
                }
            }
        }
    }

    tasks.wrapper {
        gradleVersion = "8.10"
        distributionType = Wrapper.DistributionType.ALL
    }
```

【Groovy】

```
    plugins {
        id 'org.jetbrains.kotlin.multiplatform' version '2.1.21'
    }

    repositories {
        mavenCentral()
    }

    kotlin {
        macosArm64("native") {    // Apple Silicon macOS
        // macosX64("native") {   // macOS on x86_64 platforms
        // linuxArm64("native") { // Linux on ARM64 platforms
        // linuxX64("native") {   // Linux on x86_64 platforms
        // mingwX64("native") {   // Windows
            binaries {
                sharedLib {
                    baseName = "native"       // macOS and Linux
                    // baseName = "libnative" // Windows
                }
            }
        }
    }

    wrapper {
        gradleVersion = "8.10"
        distributionType = "ALL"
    }
```

```
* The `binaries {}` block configures the project to generate a dynamic or shared libra
ry.
* `libnative` is used as the library name, the prefix for the generated header file na
me. It also prefixes all
  declarations in the header file.
```

1. 在 IDE 中运行 `linkDebugSharedNative` Gradle 任务，或者在终端中使用以下控制台命令来构建该库：

```
./gradlew linkDebugSharedNative
```

构建将会在 `build/bin/native/debugShared` 目录下生成以下库文件：

- macOS: `libnative_api.h` 与 `libnative.dylib`
- Linux: `libnative_api.h` 与 `libnative.so`
- Windows: `libnative_api.h`、 `libnative.def` 以及 `libnative.dll

> You can also use the `linkNative` Gradle task to generate both `debug` and `release` variants of the library.
>
> 

Kotlin/Native 编译器用相同的规则为所有平台生成 `.h` 文件。让我们来看看 Kotlin 库的 C 语言 API。

生成头文件

Let's examine how Kotlin/Native declarations are mapped to C functions.

In the `build/bin/native/debugShared` directory, open the `libnative_api.h` header file. 第一部分包含了标准的 C/C++ 头文件的首尾：

```
#ifndef KONAN_LIBNATIVE_H
#define KONAN_LIBNATIVE_H
#ifdef __cplusplus
extern "C" {
#endif

/// 所生成的代码的其余部分

#ifdef __cplusplus
}  /* extern "C" */
#endif
#endif  /* KONAN_LIBNATIVE_H */
```

接下来， `libnative_api.h` 包含一个通用类型定义的块：

```
#ifdef __cplusplus
typedef bool            libnative_KBoolean;
#else
typedef _Bool           libnative_KBoolean;
```

```
#endif
typedef unsigned short     libnative_KChar;
typedef signed char        libnative_KByte;
typedef short              libnative_KShort;
typedef int                libnative_KInt;
typedef long long          libnative_KLong;
typedef unsigned char      libnative_KUByte;
typedef unsigned short     libnative_KUShort;
typedef unsigned int       libnative_KUInt;
typedef unsigned long long libnative_KULong;
typedef float              libnative_KFloat;
typedef double             libnative_KDouble;
typedef float __attribute__ ((__vector_size__ (16))) libnative_KVector128;
typedef void*              libnative_KNativePtr;
```

Kotlin 在已创建的 `libnative_api.h` 文件中为所有的声明都添加了 `libnative_` 前缀。以下是完整的类型映射列表：

Kotlin 定义。	C 类型
`libnative_KBoolean`	`bool` 或 `_Bool`
`libnative_KChar`	`unsigned short`
`libnative_KByte`	`signed char`
`libnative_KShort`	`short`
`libnative_KInt`	`int`
`libnative_KLong`	`long long`
`libnative_KUByte`	`unsigned char`
`libnative_KUShort`	`unsigned short`
`libnative_KUInt`	`unsigned int`
`libnative_KULong`	`unsigned long long`
`libnative_KFloat`	`float`
`libnative_KDouble`	`double`
`libnative_KVector128`	`float __attribute__ ((__vector_size__ (16))`
`libnative_KNativePtr`	`void*`

`libnative_api.h` 文件的定义部分展示了如何将 Kotlin 的原生类型映射为 C 的原生类型。 The Kotlin/Native compiler generates these entries automatically for every library. 在这篇从 C 语言中映射原生类型教程中描述了反向映射。

After the automatically generated type definitions, you'll find the separate type definitions used in your library:

```
struct libnative_KType;
typedef struct libnative_KType libnative_KType;
```

```
/// Automatically generated type definitions

typedef struct {
  libnative_KNativePtr pinned;
} libnative_kref_example_Object;
typedef struct {
  libnative_KNativePtr pinned;
} libnative_kref_example_Clazz;
```

In C, the `typedef struct { ... } TYPE_NAME` syntax declares the structure.

> See this StackOverflow thread for more explanations of this pattern.

As you can see from these definitions, Kotlin types are mapped using the same pattern: `Object` is mapped to `libnative_kref_example_Object`，而 `Clazz` 映射到了 `libnative_kref_example_Clazz`。所有结构体都没有包含任何内容， 只有一个带有指针的 `pinned` 字段。该字段类型 `libnative_KNativePtr` 在文件前面部分定义为 `void*`。

由于 C 语言不支持命名空间，所以 Kotlin/Native 编译器生成了长名称，以避免与现有原生项目中的其他符号发生任何可能的冲突。

Service runtime functions

The `libnative_ExportedSymbols` structure defines all the functions provided by Kotlin/Native and your library. It uses nested anonymous structures heavily to mimic packages. The `libnative_` prefix comes from the library name.

`libnative_ExportedSymbols` includes several helper functions in the header file:

```
typedef struct {
  /* Service functions. */
  void (*DisposeStablePointer)(libnative_KNativePtr ptr);
  void (*DisposeString)(const char* string);
```

These functions deal with Kotlin/Native objects. `DisposeStablePointer` is called to release a reference to a Kotlin object, and `DisposeString` is called to release a Kotlin string, which has the `char*` type in C.

The next part of the `libnative_api.h` file consists of structure declarations of runtime functions:

```
libnative_KBoolean (*IsInstance)(libnative_KNativePtr ref, const libnative_KType* type
);
libnative_KBoolean (*IsInstance)(libnative_KNativePtr ref, const libnative_KType* type
);
libnative_kref_kotlin_Byte (*createNullableByte)(libnative_KByte);
```

```
libnative_KByte (*getNonNullValueOfByte)(libnative_kref_kotlin_Byte);
libnative_kref_kotlin_Short (*createNullableShort)(libnative_KShort);
libnative_KShort (*getNonNullValueOfShort)(libnative_kref_kotlin_Short);
libnative_kref_kotlin_Int (*createNullableInt)(libnative_KInt);
libnative_KInt (*getNonNullValueOfInt)(libnative_kref_kotlin_Int);
libnative_kref_kotlin_Long (*createNullableLong)(libnative_KLong);
libnative_KLong (*getNonNullValueOfLong)(libnative_kref_kotlin_Long);
libnative_kref_kotlin_Float (*createNullableFloat)(libnative_KFloat);
libnative_KFloat (*getNonNullValueOfFloat)(libnative_kref_kotlin_Float);
libnative_kref_kotlin_Double (*createNullableDouble)(libnative_KDouble);
libnative_KDouble (*getNonNullValueOfDouble)(libnative_kref_kotlin_Double);
libnative_kref_kotlin_Char (*createNullableChar)(libnative_KChar);
libnative_KChar (*getNonNullValueOfChar)(libnative_kref_kotlin_Char);
libnative_kref_kotlin_Boolean (*createNullableBoolean)(libnative_KBoolean);
libnative_KBoolean (*getNonNullValueOfBoolean)(libnative_kref_kotlin_Boolean);
libnative_kref_kotlin_Unit (*createNullableUnit)(void);
libnative_kref_kotlin_UByte (*createNullableUByte)(libnative_KUByte);
libnative_KUByte (*getNonNullValueOfUByte)(libnative_kref_kotlin_UByte);
libnative_kref_kotlin_UShort (*createNullableUShort)(libnative_KUShort);
libnative_KUShort (*getNonNullValueOfUShort)(libnative_kref_kotlin_UShort);
libnative_kref_kotlin_UInt (*createNullableUInt)(libnative_KUInt);
libnative_KUInt (*getNonNullValueOfUInt)(libnative_kref_kotlin_UInt);
libnative_kref_kotlin_ULong (*createNullableULong)(libnative_KULong);
libnative_KULong (*getNonNullValueOfULong)(libnative_kref_kotlin_ULong);
```

You can use the `IsInstance` function to check if a Kotlin object (referenced with its `.pinned` pointer) is an instance of a type. The actual set of operations generated depends on actual usages.

> Kotlin/Native has its own garbage collector, but it doesn't manage Kotlin objects accessed from C. However, Kotlin/Native provides interoperability with Swift/Objective-C, and the garbage collector is integrated with Swift/Objective-C ARC.



Your library functions

Let's take a look at the separate structure declarations used in your library. The `libnative_kref_example` field mimics the package structure of your Kotlin code with a `libnative_kref.` prefix:

```
typedef struct {
  /* User functions. */
  struct {
    struct {
      struct {
        struct {
          libnative_KType* (*_type)(void);
          libnative_kref_example_Object (*_instance)();
          const char* (*get_field)(libnative_kref_example_Object thiz);
```

```
        } Object;
        struct {
          libnative_KType* (*_type)(void);
          libnative_kref_example_Clazz (*Clazz)();
          libnative_KULong (*memberFunction)(libnative_kref_example_Clazz thiz, libnat
ive_KInt p);
        } Clazz;
        const char* (*get_globalString)();
        void (*forFloats)(libnative_KFloat f, libnative_KDouble d);
        void (*forIntegers)(libnative_KByte b, libnative_KShort s, libnative_KUInt i,
libnative_KLong l);
        const char* (*strings)(const char* str);
      } example;
    } root;
  } kotlin;
} libnative_ExportedSymbols;
```

这段代码使用了匿名的结构体定义。这里，`struct { ... } foo` 在外部结构体中声明了一个匿名结构体类型的字段，该类型没有名称。

由于 C 语言同样也不支持对象，因此使用函数指针来模仿对象语义。一个函数指针被声明为 `RETURN_TYPE (* FIELD_NAME)(PARAMETERS)` 。

`libnative_kref_example_Clazz` 字段表示 Kotlin 中的 `Clazz` 。 `libnative_KULong` 可以使用 `memberFunction` 字段访问。唯一的区别是 `memberFunction` 接受 `thiz` 引用作为第一个参数。由于 C 语言不支持对象，因此明确传递了 `thiz` 指针。

`Clazz` 字段中有一个构造函数（又名 `libnative_kref_example_Clazz_Clazz` ），它充当创建 `Clazz` 实例的构造函数。

Kotlin `object Object` 可作为 `libnative_kref_example_Object` 访问。 `_instance` 函数可以获取到该对象的唯一实例。

属性会转换为函数。 `get_` 与 `set_` 前缀分别用于命名 getter 以及 setter 函数。例如，Kotlin 中的只读属性 `globalString` 在 C 语言中会转换为 `get_globalString` 函数。

全局函数 `forFloats` 、 `forIntegers` 以及 `strings` 会转换为 `libnative_kref_example` 匿名结构体中的函数指针。

入口点

现在知道了 API 是如何创建的，初始化 `libnative_ExportedSymbols` 结构体是起点。 我们接着来看看 `libnative_api.h` 的最后部分：

```
extern libnative_ExportedSymbols* libnative_symbols(void);
```

函数 `libnative_symbols` 让你可以在原生代码中打开通往 Kotlin/Native 库的通道。 This is the entry point for accessing the library. 该库名称用作函数名称的前缀。

> 可能有必要为每个线程托管返回的 `libnative_ExportedSymbols*` 指针。
>
> 

使用 C 中生成的头文件

Using the generated headers from C is straightforward. In the library directory, create the `main.c` file with the following code:

```
#include "libnative_api.h"
#include "stdio.h"

int main(int argc, char** argv) {
  //获取调用 Kotlin/Native 函数的引用
  libnative_ExportedSymbols* lib = libnative_symbols();

  lib->kotlin.root.example.forIntegers(1, 2, 3, 4);
  lib->kotlin.root.example.forFloats(1.0f, 2.0);

  //使用 C 与 Kotlin/Native 的字符串
  const char* str = "Hello from Native!";
  const char* response = lib->kotlin.root.example.strings(str);
  printf("in: %s\nout:%s\n", str, response);
  lib->DisposeString(response);

  //创建 Kotlin 对象实例
  libnative_kref_example_Clazz newInstance = lib->kotlin.root.example.Clazz.Clazz();
  long x = lib->kotlin.root.example.Clazz.memberFunction(newInstance, 42);
  lib->DisposeStablePointer(newInstance.pinned);

  printf("DemoClazz returned %ld\n", x);

  return 0;
}
```

Compile and run the project

On macOS

如需将 C 代码编译并链接到动态库，请打开库目录并运行以下命令：

```
clang main.c libnative.dylib
```

The compiler generates an executable called `a.out`. Run it to execute the Kotlin code from the C library.

On Linux

To compile the C code and link it with the dynamic library, navigate to the library directory and run the following command:

```
gcc main.c libnative.so
```

编译器生成一个名为 `a.out` 的可执行文件。运行它来从 C 库的执行 Kotlin 代码。在 Linux 上， 需要将 `.` 引入到 `LD_LIBRARY_PATH` 来使应用程序知晓从当前文件夹加载 `libnative.so` 库。

On Windows

首先，需要安装一个支持 x64_64 目标平台的 Microsoft Visual C++ 编译器。

最简单的方法是在 Windows 机器上安装 Microsoft Visual Studio。 During installation, select the necessary components to work with C++, for example, **Desktop development with C++**.

在 Windows 上，可以通过生成静态库包装器或手动通过 LoadLibrary 或类似的 Win32API 函数来包含动态库。

我们来使用第一种选项并为 `libnative.dll` 生成静态包装库：

1. 使用工具链中的 `lib.exe` 来生成静态库包装器 `libnative.lib`，它可以在代码中自动使用 DLL：

   ```
   lib /def:libnative.def /out:libnative.lib
   ```

2. 将 `main.c` 编译为可执行文件。将生成的 `libnative.lib` 包含到到构建命令并启动：

   ```
   cl.exe main.c libnative.lib
   ```

这行命令生成了 `main.exe` 文件可供执行。

下一步做什么

- Learn more about interoperability with Swift/Objective-C
- Check out the Kotlin/Native as an Apple framework tutorial

创建使用 C 语言互操作与 libcurl 的应用——教程

This tutorial demonstrates how to use IntelliJ IDEA to create a command-line application. You'll learn how to create a simple HTTP client that can run natively on specified platforms using Kotlin/Native and the libcurl library.

The output will be an executable command-line app that you can run on macOS and Linux and make simple HTTP GET requests.

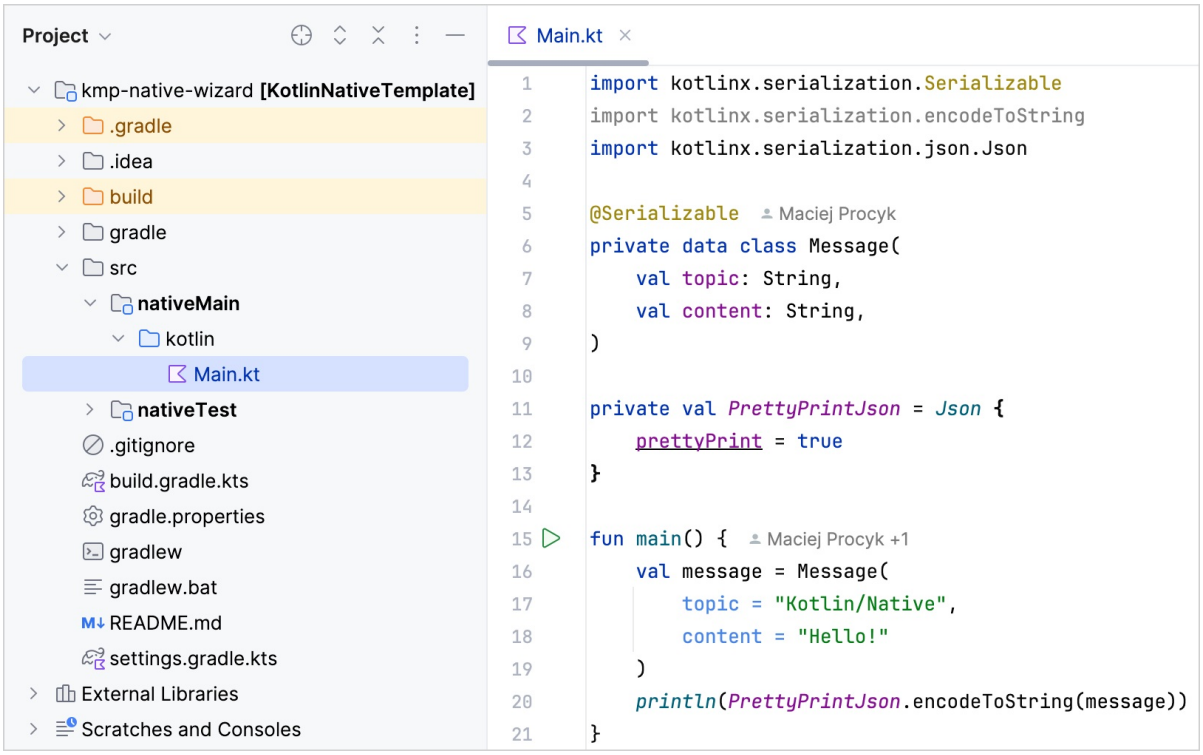
You can use the command line to generate a Kotlin library, either directly or with a script file (such as `.sh` or `.bat` file). However, this approach doesn't scale well for larger projects that have hundreds of files and libraries. Using a build system simplifies the process by downloading and caching the Kotlin/Native compiler binaries and libraries with transitive dependencies, as well as by running the compiler and tests. Kotlin/Native can use the Gradle build system through the Kotlin Multiplatform plugin.

Before you start

1. Download and install the latest version of IntelliJ IDEA.
2. Clone the project template by selecting **File** | **New** | **Project from Version Control** in IntelliJ IDEA and using this URL:

   ```
   https://github.com/Kotlin/kmp-native-wizard
   ```

3. Explore the project structure:



The template includes a project with the files and folders you need to get started. It's important to understand that an application written in Kotlin/Native can target different platforms if the code does not have platform-specific requirements. Your code is placed in the `nativeMain` directory with a corresponding `nativeTest`. For this tutorial, keep the folder structure as is.

4. Open the `build.gradle.kts` file, the build script that contains the project settings. Pay special attention to the following in the build file:

```
kotlin {
    val hostOs = System.getProperty("os.name")
    val isArm64 = System.getProperty("os.arch") == "aarch64"
    val isMingwX64 = hostOs.startsWith("Windows")
    val nativeTarget = when {
        hostOs == "Mac OS X" && isArm64 -> macosArm64("native")
        hostOs == "Mac OS X" && !isArm64 -> macosX64("native")
        hostOs == "Linux" && isArm64 -> linuxArm64("native")
        hostOs == "Linux" && !isArm64 -> linuxX64("native")
        isMingwX64 -> mingwX64("native")
        else -> throw GradleException("Host OS is not supported in Kotlin/Native."
)
    }

    nativeTarget.apply {
        binaries {
            executable {
                entryPoint = "main"
            }
        }
    }
}
```

- Targets are defined using `macosArm64`, `macosX64`, `linuxArm64`, `linuxX64`, and `mingwX64` for macOS, Linux, and Windows. See the complete list of supported platforms.
- The `binaries {}` block defines how the binary is generated and the entry point of the application. These can be left as default values.
- C interoperability is configured as an additional step in the build. By default, all the symbols from C are imported to the `interop` package. You may want to import the whole package in `.kt` files. Learn more about how to configure it.

创建一个定义文件

When writing native applications, you often need access to certain functionalities that are not included in the Kotlin standard library, such as making HTTP requests, reading and writing from disk, and so on.

Kotlin/Native helps consume standard C libraries, opening up an entire ecosystem of functionality that exists for pretty much anything you may need. Kotlin/Native is already shipped with a set of prebuilt platform libraries, which provide some additional common functionality to the standard library.

An ideal scenario for interop is to call C functions as if you are calling Kotlin functions, following the same signature and conventions. This is when the cinterop tool comes in handy. It takes a C library and generates the corresponding Kotlin bindings, so that the library can be used as if it were Kotlin code.

To generate these bindings, each library needs a definition file, usually with the same name as the library. This is a property file that describes exactly how the library should be consumed.

In this app, you'll need the libcurl library to make some HTTP calls. To create its definition file:

1. Select the `src` folder and create a new directory with **File | New | Directory**.
2. Name the new directory **nativeInterop/cinterop**. This is the default convention for header file locations, though it can be overridden in the `build.gradle.kts` file if you use a different location.
3. Select this new subfolder and create a new `libcurl.def` file with **File | New | File**.
4. Update your file with the following code:

```
headers = curl/curl.h
headerFilter = curl/*

compilerOpts.linux = -I/usr/include -I/usr/include/x86_64-linux-gnu
linkerOpts.osx = -L/opt/local/lib -L/usr/local/opt/curl/lib -lcurl
linkerOpts.linux = -L/usr/lib/x86_64-linux-gnu -lcurl
```

- `headers` is the list of header files to generate Kotlin stubs for. You can add multiple files here, separating each with a space. In this case, it's only `curl.h`. The referenced files need to be available on the specified path (in this case, it's `/usr/include/curl`).
- `headerFilter` shows what exactly is included. In C, all the headers are also included when one file references another one with the `#include` directive. Sometimes it's not necessary, and you can add this parameter using glob patterns) to make adjustments.

  You can use `headerFilter` if you don't want to fetch external dependencies (such as system `stdint.h` header) into the interop library. Also, it may be useful for library size optimization and fixing potential conflicts between the system and the provided Kotlin/Native compilation environment.

- If the behavior for a certain platform needs to be modified, you can use a format like `compilerOpts.osx` or `compilerOpts.linux` to provide platform-specific values to the options. In this case, they are macOS (the `.osx` suffix) and Linux (the `.linux` suffix). Parameters without a suffix are also possible (for example, `linkerOpts=`) and applied to all platforms.

For the full list of available options, see Definition file.

> You need to have the `curl` library binaries on your system to make the sample work. On macOS and Linux, they are usually included. On Windows, you can build it from sources (you'll need Microsoft Visual Studio or Windows SDK command-line tools). For more details, see the related blog post. Alternatively, you may want to consider a MinGW/MSYS2 `curl` binary.



将互操作添加到构建过程

To use header files, make sure they are generated as a part of the build process. For this, add the following `compilations {}` block to the `build.gradle.kts` file:

```
nativeTarget.apply {
    compilations.getByName("main") {
        cinterops {
            val libcurl by creating
        }
    }
    binaries {
        executable {
            entryPoint = "main"
        }
    }
```

```
}
```

First, `cinterops` is added, and then an entry for a definition file. By default, the name of the file is used. You can override this with additional parameters:

```
cinterops {
    val libcurl by creating {
        definitionFile.set(project.file("src/nativeInterop/cinterop/libcurl.def"))
        packageName("com.jetbrains.handson.http")
        compilerOpts("-I/path")
        includeDirs.allHeaders("path")
    }
}
```

编写应用程序代码

Now that you have the library and the corresponding Kotlin stubs, you can use them from your application. For this tutorial, convert the simple.c example to Kotlin.

In the `src/nativeMain/kotlin/` folder, update your `Main.kt` file with the following code:

```
import kotlinx.cinterop.*
import libcurl.*

@OptIn(ExperimentalForeignApi::class)
fun main(args: Array<String>) {
    val curl = curl_easy_init()
    if (curl != null) {
        curl_easy_setopt(curl, CURLOPT_URL, "https://example.com")
        curl_easy_setopt(curl, CURLOPT_FOLLOWLOCATION, 1L)
        val res = curl_easy_perform(curl)
        if (res != CURLE_OK) {
            println("curl_easy_perform() failed ${curl_easy_strerror(res)?.toKString()}
")
        }
        curl_easy_cleanup(curl)
    }
}
```

As you can see, explicit variable declarations are eliminated in the Kotlin version, but everything else is pretty much the same as the C version. All the calls you'd expect in the libcurl library are available in the Kotlin equivalent.

> This is a line-by-line literal translation. You could also write this in a more Kotlin idiomatic way.



编译与运行应用程序

1. Compile the application. To do that, run the `runDebugExecutableNative` Gradle task from the task list or use the following command in the terminal:

   ```
   ./gradlew runDebugExecutableNative
   ```

   In this case, the part generated by the cinterop tool is implicitly included in the build.

2. If there are no errors during compilation, click the green **Run** icon in the gutter next to the `main()` function or use the Shift + Cmd + R/Shift + F10 shortcut.

   IntelliJ IDEA opens the **Run** tab and shows the output — the contents of example.com:

   ```
       div {
           width: 600px;
           margin: 5em auto;
           padding: 2em;
           background-color: #fdfdff;
           border-radius: 0.5em;
           box-shadow: 2px 3px 7px 2px rgba(0,0,0,0.02);
       }
       a:link, a:visited {
           color: #38488f;
           text-decoration: none;
       }
       @media (max-width: 700px) {
           div {
               margin: 0 auto;
               width: auto;
           }
       }
       </style>
   </head>

   <body>
   <div>
       <h1>Example Domain</h1>
       <p>This domain is for use in illustrative examples in documents. You may use this
       domain in literature without prior coordination or asking for permission.</p>
       <p><a href="https://www.iana.org/domains/example">More information...</a></p>
   </div>
   </body>
   </html>
   ```

You can see the actual output because the call `curl_easy_perform` prints the result to the standard output. You could hide this using `curl_easy_setopt` .

> You can get the full project code in our GitHub repository.
>
> 

下一步做什么

Learn more about Kotlin's interoperability with C.

与 Objective-C 互操作性

- 与 Swift/Objective-C 互操作性
- Kotlin/Native 开发 Apple framework——教程

与 Swift/Objective-C 互操作性

> The Objective-C libraries import is Experimental. All Kotlin declarations generated by the cinterop tool from Objective-C libraries should have the `@ExperimentalForeignApi` annotation.
>
> Native platform libraries shipped with Kotlin/Native (like Foundation, UIKit, and POSIX) require opt-in only for some APIs.
>
> 

Kotlin/Native provides indirect interoperability with Swift through Objective-C. This document covers how you can use Kotlin declarations in Swift/Objective-C code and Objective-C declarations in Kotlin code.

Some other resources you might find useful:

- The Kotlin-Swift interopedia, a collection of examples on how to use Kotlin declarations in Swift code.
- The Integration with Swift/Objective-C ARC section, covering the details of integration between Kotlin's tracing GC and Objective-C's ARC.

Importing Swift/Objective-C libraries to Kotlin

Objective-C frameworks and libraries can be used in Kotlin code if properly imported to the build (system frameworks are imported by default). For more details, see:

- Create and configure a library definition file
- Configure compilation for native libraries

A Swift library can be used in Kotlin code if its API is exported to Objective-C with `@objc`. Pure Swift modules are not yet supported.

Using Kotlin in Swift/Objective-C

Kotlin modules can be used in Swift/Objective-C code if compiled into a framework:

- See Build final native binaries to see how to declare binaries.
- Check out the Kotlin Multiplatform sample project for an example.

Hide Kotlin declarations from Objective-C and Swift

> The `@HiddenFromObjC` annotation is Experimental and requires opt-in.
>
> 

To make your Kotlin code more Swift/Objective-C-friendly, use the `@HiddenFromObjC` annotation to hide a Kotlin declaration from Objective-C and Swift. It disables the function or property export to Objective-C.

Alternatively, you can mark Kotlin declarations with the `internal` modifier to restrict their visibility in the compilation module. Use `@HiddenFromObjC` if you want to hide the Kotlin declaration from Objective-C and Swift while keeping it visible to other Kotlin modules.

See an example in the Kotlin-Swift interopedia.

Use refining in Swift

> The `@ShouldRefineInSwift` annotation is Experimental and requires opt-in.
>
> 

`@ShouldRefineInSwift` helps to replace a Kotlin declaration with a wrapper written in Swift. The annotation marks a function or property as `swift_private` in the generated Objective-C API. Such declarations get the `__` prefix, which makes them invisible from Swift.

You can still use these declarations in your Swift code to create a Swift-friendly API, but they won't be suggested in the Xcode autocomplete.

- For more information on refining Objective-C declarations in Swift, see the official Apple documentation.
- For an example on how to use the `@ShouldRefineInSwift` annotation, see the Kotlin-Swift interopedia.

Change declaration names

> The `@ObjCName` annotation is Experimental and requires opt-in.
>
> 

To avoid renaming Kotlin declarations, use the `@ObjCName` annotation. It instructs the Kotlin compiler to use the custom Objective-C and Swift name for the annotated class, interface, or another Kotlin entity:

```
@ObjCName(swiftName = "MySwiftArray")
class MyKotlinArray {
    @ObjCName("index")
    fun indexOf(@ObjCName("of") element: String): Int = TODO()
```

```
}

// Usage with the ObjCName annotations
let array = MySwiftArray()
let index = array.index(of: "element")
```

See another example in the Kotlin-Swift interopedia.

Provide documentation with KDoc comments

Documentation is essential for understanding any API. Providing documentation for the shared Kotlin API allows you to communicate with its users on matters of usage, dos and don'ts, and so on.

By default, KDocs comments are not translated into corresponding comments when generating an Objective-C header. For example, the following Kotlin code with KDoc:

```
/**
 * Prints the sum of the arguments.
 * Properly handles the case when the sum doesn't fit in 32-bit integer.
 */
fun printSum(a: Int, b: Int) = println(a.toLong() + b)
```

Will produce an Objective-C declaration without any comments:

```
+ (void)printSumA:(int32_t)a b:(int32_t)b __attribute__((swift_name("printSum(a:b:)"))
);
```

To enable export of KDoc comments, add the following compiler option to your `build.gradle(.kts)`:

> 【Kotlin】

```
kotlin {
    targets.withType<org.jetbrains.kotlin.gradle.plugin.mpp.KotlinNativeTarget> {
        compilations.get("main").compilerOptions.options.freeCompilerArgs.add("-Xexpor
t-kdoc")
    }
}
```

> 【Groovy】

```
kotlin {
    targets.withType(org.jetbrains.kotlin.gradle.plugin.mpp.KotlinNativeTarget) {
        compilations.get("main").compilerOptions.options.freeCompilerArgs.add("-Xexpor
t-kdoc")
    }
}
```

After that, the Objective-C header will contain a corresponding comment:

```
/**
 * Prints the sum of the arguments.
 * Properly handles the case when the sum doesn't fit in 32-bit integer.
 */
+ (void)printSumA:(int32_t)a b:(int32_t)b __attribute__((swift_name("printSum(a:b:)"))
);
```

You'll be able to see comments on classes and methods in autocompletion, for example, in Xcode. If you go to the definition of functions (in the `.h` file), you'll see comments on `@param`, `@return`, and so on.

Known limitations:

> The ability to export KDoc comments to generated Objective-C headers is Experimental. It may be dropped or changed at any time. Opt-in is required (see the details below), and you should use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack.
>
> 

- Dependency documentation is not exported unless it is compiled with `-Xexport-kdoc` itself. The feature is Experimental, so libraries compiled with this option might be incompatible with other compiler versions.
- KDoc comments are mostly exported as is. Many KDoc features, for example `@property`, are not supported.

Mappings

The table below shows how Kotlin concepts are mapped to Swift/Objective-C and vice versa.

"->" and "<-" indicate that mapping only goes one way.

Kotlin	Swift	Objective-C	Notes
`class`	`class`	`@interface`	note

`interface`	`protocol`	`@protocol`	
`constructor` / `create`	Initializer	Initializer	note
Property	Property	Property	note 1, note 2
Method	Method	Method	note 1, note 2
`enum class`	`class`	`@interface`	note
`suspend` ->	`completionHandler:` / `async`	`completionHandler:`	note 1, note 2
`@Throws fun`	`throws`	`error:(NSError**)error`	note
Extension	Extension	Category member	note
`companion` member <-	Class method or property	Class method or property	
`null`	`nil`	`nil`	
`Singleton`	`shared` or `companion` property	`shared` or `companion` property	note
Primitive type	Primitive type / `NSNumber`		note
`Unit` return type	`Void`	`void`	
`String`	`String`	`NSString`	note
`String`	`NSMutableString`	`NSMutableString`	note
`List`	`Array`	`NSArray`	
`MutableList`	`NSMutableArray`	`NSMutableArray`	
`Set`	`Set`	`NSSet`	
`MutableSet`	`NSMutableSet`	`NSMutableSet`	note
`Map`	`Dictionary`	`NSDictionary`	
`MutableMap`	`NSMutableDictionary`	`NSMutableDictionary`	note
Function type	Function type	Block pointer type	note
Inline classes	Unsupported	Unsupported	note

Classes

Name translation

Objective-C classes are imported into Kotlin with their original names. Protocols are imported as interfaces with a `Protocol` name suffix, for example, `@protocol Foo` -> `interface FooProtocol`. These classes and interfaces are placed into a package specified in build configuration (`platform.*`

packages for preconfigured system frameworks).

The names of Kotlin classes and interfaces are prefixed when imported to Objective-C. The prefix is derived from the framework name.

Objective-C does not support packages in a framework. If the Kotlin compiler finds Kotlin classes in the same framework which have the same name but different packages, it renames them. This algorithm is not stable yet and can change between Kotlin releases. To work around this, you can rename the conflicting Kotlin classes in the framework.

Strong linking

Whenever you use an Objective-C class in the Kotlin source, it's marked as a strongly linked symbol. The resulting build artifact mentions related symbols as strong external references.

This means that the app tries to link symbols during the launch dynamically, and if they are unavailable, the app crashes. The crash happens even if symbols were never used. Symbols might be unavailable on a particular device or OS version.

To work around this issue and avoid "Symbol not found" errors, use a Swift or Objective-C wrapper that checks if the class is actually available. See how this workaround was implemented in the Compose Multiplatform framework.

Initializers

A Swift/Objective-C initializer is imported to Kotlin as constructors or as factory methods named `create`. The latter happens with initializers declared in the Objective-C category or as a Swift extension, because Kotlin has no concept of extension constructors.

> Before importing Swift initializers to Kotlin, don't forget to annotate them with `@objc`.
>
> 

Kotlin constructors are imported as initializers to Swift/Objective-C.

Setters

Writeable Objective-C properties overriding read-only properties of the superclass are represented as the `setFoo()` method for the property `foo`. The same goes for a protocol's read-only properties that are implemented as mutable.

Top-level functions and properties

Top-level Kotlin functions and properties are accessible as members of special classes. Each Kotlin file is translated into such a class, for example:

```
// MyLibraryUtils.kt
package my.library

fun foo() {}
```

You can then call the `foo()` function from Swift like this:

```
MyLibraryUtilsKt.foo()
```

See a collection of examples on accessing top-level Kotlin declarations in the Kotlin-Swift interopedia:

- Top-level functions
- Top-level read-only properties
- Top-level mutable properties

Method names translation

Generally, Swift argument labels and Objective-C selector pieces are mapped to Kotlin parameter names. These two concepts have different semantics, so sometimes Swift/Objective-C methods can be imported with a clashing Kotlin signature. In this case, the clashing methods can be called from Kotlin using named arguments, for example:

```
[player moveTo:LEFT byMeters:17]
[player moveTo:UP byInches:42]
```

In Kotlin, it's:

```
player.moveTo(LEFT, byMeters = 17)
player.moveTo(UP, byInches = 42)
```

Here's how the `kotlin.Any` functions are mapped to Swift/Objective-C:

Kotlin	Swift	Objective-C
`equals()`	`isEquals(_:)`	`isEquals:`
`hashCode()`	`hash`	`hash`
`toString()`	`description`	`description`

See an example with data classes in the Kotlin-Swift interopedia.

You can specify a more idiomatic name in Swift or Objective-C, instead of renaming the Kotlin declaration with the `@ObjCName` annotation.

Errors and exceptions

All Kotlin exceptions are unchecked, meaning that errors are caught at runtime. However, Swift has only checked errors that are handled at compile time. So, if Swift or Objective-C code calls a Kotlin method that throws an exception, the Kotlin method should be marked with the `@Throws` annotation, specifying a list of "expected" exception classes.

When compiling to the Swift/Objective-C framework, non-`suspend` functions that have or inherit the `@Throws` annotation are represented as `NSError*`-producing methods in Objective-C and as `throws` methods in Swift. Representations for `suspend` functions always have an `NSError*` / `Error` parameter in the completion handler.

When a Kotlin function called from Swift/Objective-C code throws an exception which is an instance of one of the classes specified with `@Throws` or their subclasses, the exception is propagated as an `NSError`. Other Kotlin exceptions reaching Swift/Objective-C are considered unhandled and cause program termination.

`suspend` functions without `@Throws` propagate only `CancellationException` (as `NSError`). Non-`suspend` functions without `@Throws` don't propagate Kotlin exceptions at all.

Note that the opposite reversed translation is not implemented yet: Swift/Objective-C error-throwing methods aren't imported to Kotlin as exception-throwing.

See an example in the Kotlin-Swift interopedia.

Enums

Kotlin enums are imported into Objective-C as `@interface` and into Swift as `class`. These data structures have properties corresponding to each enum value. Consider this Kotlin code:

```
// Kotlin
enum class Colors {
    RED, GREEN, BLUE
}
```

You can access the properties of this enum class from Swift as follows:

```
// Swift
Colors.red
Colors.green
Colors.blue
```

To use variables of a Kotlin enum in a Swift `switch` statement, provide a default statement to prevent a compilation error:

```
switch color {
    case .red: print("It's red")
    case .green: print("It's green")
    case .blue: print("It's blue")
    default: fatalError("No such color")
}
```

See another example in the Kotlin-Swift interopedia.

Suspending functions

> Support for calling `suspend` functions from Swift code as `async` is Experimental. It may be dropped or changed at any time. Use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack.
>
> 

Kotlin's suspending functions (`suspend`) are presented in the generated Objective-C headers as functions with callbacks, or completion handlers in Swift/Objective-C terminology.

Starting from Swift 5.5, Kotlin's `suspend` functions are also available for calling from Swift as `async` functions without using the completion handlers. Currently, this functionality is highly experimental and has certain limitations. See this YouTrack issue for details.

- Learn more about the `async` / `await` mechanism in the Swift documentation.
- See an example and recommendations on third-party libraries that implement the same functionality in the Kotlin-Swift interopedia.

Extensions and category members

Members of Objective-C categories and Swift extensions are generally imported to Kotlin as extensions. That's why these declarations can't be overridden in Kotlin, and the extension initializers aren't available as Kotlin constructors.

> Currently, there are two exceptions. Starting with Kotlin 1.8.20, category members that are declared in the same headers as the NSView class (from the AppKit framework) or UIView classes (from the UIKit framework) are imported as members of these classes. This means that you can override methods that subclass from NSView or UIView.
>
> 

Kotlin extensions to "regular" Kotlin classes are imported to Swift and Objective-C as extensions and category members, respectively. Kotlin extensions to other types are treated as top-level declarations with an additional receiver parameter. These types include:

- Kotlin `String` type
- Kotlin collection types and subtypes
- Kotlin `interface` types
- Kotlin primitive types
- Kotlin `inline` classes
- Kotlin `Any` type
- Kotlin function types and subtypes
- Objective-C classes and protocols

See a collection of examples in the Kotlin-Swift interopedia.

Kotlin singletons

Kotlin singleton (made with an `object` declaration, including `companion object`) is imported to Swift/Objective-C as a class with a single instance.

The instance is available through the `shared` and `companion` properties.

For the following Kotlin code:

```
object MyObject {
    val x = "Some value"
}

class MyClass {
    companion object {
        val x = "Some value"
    }
}
```

Access these objects as follows:

```
MyObject.shared
MyObject.shared.x
MyClass.companion
MyClass.Companion.shared
```

> Access objects through `[MySingleton mySingleton]` in Objective-C and `MySingleton()` in Swift has been deprecated.
>
> 

See more examples in the Kotlin-Swift interopedia:

- How to access Kotlin objects using `shared`
- How to access members of Kotlin companion objects from Swift.

Primitive types

Kotlin primitive type boxes are mapped to special Swift/Objective-C classes. For example, the `kotlin.Int` box is represented as the `KotlinInt` class instance in Swift (or the `${prefix}Int` instance in Objective-C, where `prefix` is the framework's name prefix). These classes are derived from `NSNumber`, so the instances are proper `NSNumber`s supporting all corresponding operations.

The `NSNumber` type is not automatically translated to Kotlin primitive types when used as a Swift/Objective-C parameter type or return value. The reason is that the `NSNumber` type doesn't provide enough information about a wrapped primitive value type, for example, `NSNumber` is statically not known to be `Byte`, `Boolean`, or `Double`. So Kotlin primitive values should be cast to and from `NSNumber` manually.

Strings

When a Kotlin `String` is passed to Swift, it's first exported as an Objective-C object, and then the Swift compiler copies it one more time for a Swift conversion. This results in additional runtime overhead.

To avoid that, access Kotlin strings in Swift directly as an Objective-C `NSString` instead. See the conversion example.

NSMutableString

`NSMutableString` Objective-C class is not available from Kotlin. All instances of `NSMutableString` are copied when passed to Kotlin.

Collections

Kotlin -> Objective-C -> Swift

When a Kotlin collection is passed to Swift, it's first converted to an Objective-C equivalent, and then the Swift compiler copies the entire collection and converts it into a Swift-native collection as described in the mappings table.

This last conversion leads to performance costs. To prevent this, when using Kotlin collections in Swift, explicitly cast them to their Objective-C counterparts: `NSDictionary`, `NSArray`, or `NSSet`.

See the conversion example

For example, the following Kotlin declaration:

```
val map: Map<String, String>
```

In Swift, might look like this:

```
map[key]?.count ?? 0
```

Here, the `map` is implicitly converted to Swift's `Dictionary`, and its string values are mapped to Swift's `String`. This results in a performance cost.

To avoid the conversion, explicitly cast `map` to Objective-C's `NSDictionary` and access values as `NSString` instead:

```
let nsMap: NSDictionary = map as NSDictionary
(nsMap[key] as? NSString)?.length ?? 0
```

This ensures that the Swift compiler doesn't perform an additional conversion step.

Swift -> Objective-C -> Kotlin

Swift/Objective-C collections are mapped to Kotlin as described in the mappings table, except for `NSMutableSet` and `NSMutableDictionary`.

`NSMutableSet` isn't converted to a Kotlin's `MutableSet`. To pass an object to Kotlin `MutableSet`, explicitly create this kind of Kotlin collection. To do this, use, for example, the `mutableSetOf()` function in Kotlin or the `KotlinMutableSet` class in Swift and `${prefix}MutableSet` in Objective-C (`prefix` is the framework names prefix). The same is true for `MutableMap`.

See an example in the Kotlin-Swift interopedia.

Function types

Kotlin function-typed objects (for example, lambdas) are converted to functions in Swift and blocks in Objective-C. See an example of a Kotlin function with a lambda in the Kotlin-Swift interopedia.

However, there is a difference in how types of parameters and return values are mapped when translating a function and a function type. In the latter case, primitive types are mapped to their boxed representation. Kotlin `Unit` return value is represented as a corresponding `Unit` singleton in Swift/Objective-C. The value of this singleton can be retrieved the same way as for any other Kotlin `object`. See singletons in the table above.

Consider the following Kotlin function:

```
fun foo(block: (Int) -> Unit) { ... }
```

It's represented in Swift as follows:

```
func foo(block: (KotlinInt) -> KotlinUnit)
```

And you can call it like this:

```
foo {
    bar($0 as! Int32)
    return KotlinUnit()
}
```

Generics

Objective-C supports "lightweight generics" defined in classes, with a relatively limited feature set. Swift can import generics defined on classes to help provide additional type information to the compiler.

Generic feature support for Objective-C and Swift differ from Kotlin, so the translation will inevitably lose some information, but the features supported retain meaningful information.

For specific examples on how to use Kotlin generics in Swift, see the Kotlin-Swift interopedia.

Limitations

Objective-C generics do not support all features of either Kotlin or Swift, so there will be some information lost in the translation.

Generics can only be defined on classes, not on interfaces (protocols in Objective-C and Swift) or functions.

Nullability

Kotlin and Swift both define nullability as part of the type specification, while Objective-C defines nullability on methods and properties of a type. So, the following Kotlin code:

```
class Sample<T>() {
    fun myVal(): T
}
```

Looks in Swift like this:

```
class Sample<T>() {
    fun myVal(): T?
}
```

To support a potentially nullable type, the Objective-C header needs to define `myVal` with a nullable return value.

To mitigate this, when defining your generic classes, provide a non-nullable type constraint if the generic type should *never* be null:

```
class Sample<T : Any>() {
    fun myVal(): T
}
```

That will force the Objective-C header to mark `myVal` as non-nullable.

Variance

Objective-C allows generics to be declared covariant or contravariant. Swift has no support for variance. Generic classes coming from Objective-C can be force-cast as needed.

```
data class SomeData(val num: Int = 42) : BaseData()
class GenVarOut<out T : Any>(val arg: T)
```

```
let variOut = GenVarOut<SomeData>(arg: sd)
let variOutAny : GenVarOut<BaseData> = variOut as! GenVarOut<BaseData>
```

Constraints

In Kotlin, you can provide upper bounds for a generic type. Objective-C also supports this, but that support is unavailable in more complex cases and is currently not supported in the Kotlin - Objective-C interop. The exception here being a non-nullable upper bound will make Objective-C methods/properties non-nullable.

To disable

To have the framework header written without generics, add the following compiler option in your build file:

```
binaries.framework {
    freeCompilerArgs += "-Xno-objc-generics"
}
```

Forward declarations

To import forward declarations, use the `objcnames.classes` and `objcnames.protocols` packages. For example, to import a `objcprotocolName` forward declaration declared in an Objective-C library with a `library.package`, use a special forward declaration package: `import objcnames.protocols.objcprotocolName`.

Consider two objcinterop libraries: one that uses `objcnames.protocols.ForwardDeclaredProtocolProtocol` and another with an actual implementation in another package:

```
// First objcinterop library
#import <Foundation/Foundation.h>

@protocol ForwardDeclaredProtocol;

NSString* consumeProtocol(id<ForwardDeclaredProtocol> s) {
    return [NSString stringWithUTF8String:"Protocol"];
}
```

```
// Second objcinterop library
// Header:
#import <Foundation/Foundation.h>
@protocol ForwardDeclaredProtocol
@end
// Implementation:
@interface ForwardDeclaredProtocolImpl : NSObject <ForwardDeclaredProtocol>
@end

id<ForwardDeclaredProtocol> produceProtocol() {
    return [ForwardDeclaredProtocolImpl new];
}
```

To transfer objects between the two libraries, use an explicit `as` cast in your Kotlin code:

```
// Kotlin code:
fun test() {
    consumeProtocol(produceProtocol() as objcnames.protocols.ForwardDeclaredProtocolPr
otocol)
}
```

> You can only cast to `objcnames.protocols.ForwardDeclaredProtocolProtocol` from the corresponding real class. Otherwise, you'll get an error.
>
> 

Casting between mapped types

When writing Kotlin code, an object may need to be converted from a Kotlin type to the equivalent Swift/Objective-C type (or vice versa). In this case, a plain old Kotlin cast can be used, for example:

```
val nsArray = listOf(1, 2, 3) as NSArray
val string = nsString as String
val nsNumber = 42 as NSNumber
```

Subclassing

Subclassing Kotlin classes and interfaces from Swift/Objective-C

Kotlin classes and interfaces can be subclassed by Swift/Objective-C classes and protocols.

Subclassing Swift/Objective-C classes and protocols from Kotlin

Swift/Objective-C classes and protocols can be subclassed with a Kotlin `final` class. Non-`final` Kotlin classes inheriting Swift/Objective-C types aren't supported yet, so it is not possible to declare a complex class hierarchy inheriting Swift/Objective-C types.

Normal methods can be overridden using the `override` Kotlin keyword. In this case, the overriding method must have the same parameter names as the overridden one.

Sometimes it is required to override initializers, for example when subclassing `UIViewController`. Initializers imported as Kotlin constructors can be overridden by Kotlin constructors marked with the `@OverrideInit` annotation:

```
class ViewController : UIViewController {
    @OverrideInit constructor(coder: NSCoder) : super(coder)

    ...
}
```

The overriding constructor must have the same parameter names and types as the overridden one.

To override different methods with clashing Kotlin signatures, you can add the `@ObjCSignatureOverride` annotation to the class. The annotation instructs the Kotlin compiler to ignore conflicting overloads, in case several functions with the same argument types, but different argument names, are inherited from the Objective-C class.

By default, the Kotlin/Native compiler doesn't allow calling a non-designated Objective-C initializer as a `super()` constructor. This behavior can be inconvenient if the designated initializers aren't marked properly in the Objective-C library. To disable these compiler checks, add the `disableDesignatedInitializerChecks = true` to the library's `.def` file.

C features

See Interoperability with C for an example case where the library uses some plain C features, such as unsafe pointers, structs, and so on.

Unsupported

Some features of the Kotlin programming language are not yet mapped into the respective features of Objective-C or Swift. Currently, the following features are not properly exposed in generated framework headers:

- Inline classes (arguments are mapped as either underlying primitive type or `id`)
- Custom classes implementing standard Kotlin collection interfaces (`List`, `Map`, `Set`) and other special classes
- Kotlin subclasses of Objective-C classes

Kotlin/Native 开发 Apple framework——教程

> The Objective-C libraries import is Experimental. All Kotlin declarations generated by the cinterop tool from Objective-C libraries should have the `@ExperimentalForeignApi` annotation.
>
> Native platform libraries shipped with Kotlin/Native (like Foundation, UIKit, and POSIX) require opt-in only for some APIs.



Kotlin/Native 提供与 Swift/Objective-C 的双向互操作性。既可以在 Kotlin 代码中使用 Objective-C framework 与库，也可以在 Swift/Objective-C 代码中使用 Kotlin 模块。

Kotlin/Native comes with a set of pre-imported system frameworks; it's also possible to import an existing framework and use it from Kotlin. 在本教程中，可学到如何创建自己的 framework 以及如何在 macOS 与 iOS 上的 Swift/Objective-C 应用程序中使用 Kotlin/Native 代码。

在本教程中会：

- 创建一个 Kotlin 库并将其编译为 framework
- 检查生成的 Swift/Objective-C API 代码
- 在 Objective-C 中使用 framework
- 在 Swift 中使用 framework

你可以使用命令行来生成 Kotlin framework，可以直接生成，也可以通过脚本文件（例如 `.sh` 或 `.bat` 文件）生成。 不过，对于有数百个文件与库的大型项目来说，这种方法的可伸缩性不佳。 使用构建系统可以简化这个过程，它会下载并缓存 Kotlin/Native 编译器二进制文件与库（及其传递），并运行编译器和测试。 Kotlin/Native 可以通过 Kotlin 多平台插件 使用 Gradle 构建系统。

> 如果使用 Mac 并且想要创建并运行 iOS 或其他 Apple 平台的应用程序，还需要先安装 Xcode Command Line Tools，然后启动它，并接受许可条款。



创建一个 Kotlin 库

> 参见 Kotlin/Native 入门教程以了解详细的入门步骤 以及如何创建新的 Kotlin/Native 项目并在 IntelliJ IDEA 中打开的说明。



Kotlin/Native 编译器可以由 Kotlin 代码生产一个 macOS 与 iOS 适用的 framework。所创建的 framework 包含了在 Swift/Objective-C 中使用它所需的所有声明与二进制文件。

我们首先创建一个 Kotlin 库：

1. 在 `src/nativeMain/kotlin` 目录中，创建 `lib.kt` 文件，并在其中编写库的内容：

```
package example

object Object {
    val field = "A"
}

interface Interface {
    fun iMember() {}
}

class Clazz : Interface {
    fun member(p: Int): ULong? = 42UL
}

fun forIntegers(b: Byte, s: UShort, i: Int, l: ULong?) { }
fun forFloats(f: Float, d: Double?) { }

fun strings(str: String?) : String {
    return "That is '$str' from C"
}

fun acceptFun(f: (String) -> String?) = f("Kotlin/Native rocks!")
fun supplyFun() : (String) -> String? = { "$it is cool!" }
```

2. Update your `build.gradle(.kts)` Gradle build file with the following:

> 【Kotlin】

```
plugins {
    kotlin("multiplatform") version "2.1.21"
}

repositories {
    mavenCentral()
}

kotlin {
    iosArm64("native") {
        binaries {
            framework {
                baseName = "Demo"
```

```
                }
            }
        }
    }

    tasks.wrapper {
        gradleVersion = "8.10"
        distributionType = Wrapper.DistributionType.ALL
    }
```

> 【Groovy】

```
    plugins {
        id 'org.jetbrains.kotlin.multiplatform' version '2.1.21'
    }

    repositories {
        mavenCentral()
    }

    kotlin {
        iosArm64("native") {
            binaries {
                framework {
                    baseName = "Demo"
                }
            }
        }
    }

    wrapper {
        gradleVersion = "8.10"
        distributionType = "ALL"
    }
```

`binaries {}` 代码块配置项目生成动态或共享库。

```
Kotlin/Native supports the `iosArm64`, `iosX64`, and `iosSimulatorArm64` targets for i
OS, as well as `macosX64` and
`macosArm64` targets for macOS. So, you can replace the `iosArm64()` with the respecti
ve Gradle function for your
target platform:
```

目标 平台/设备	Gradle 函数

macOS x86_64	`macosX64()`
macOS ARM64	`macosArm64()`
iOS ARM64	`iosArm64()`
iOS Simulator (x86_64)	`iosX64()`
iOS Simulator (ARM64)	`iosSimulatorArm64()`

For information on other supported Apple targets, see Kotlin/Native target support.

1. 在 IDE 中运行 `linkDebugFrameworkNative` Gradle 任务，或者在终端中使用如下控制台命令来构建 framework：

   ```
   ./gradlew linkDebugFrameworkNative
   ```

构建生成的 framework 位于 `build/bin/native/debugFramework` 目录。

> You can also use the `linkNative` Gradle task to generate both `debug` and `release` variants of the framework.
>
> 

生成的 Framework 头文件

每个 framework 变体都包含一个头文件。这些头文件不依赖目标平台。头文件包含你的 Kotlin 代码的定义以及一些 Kotlin 级的声明。Let's see what's inside.

Kotlin/Native 运行时声明

In the `build/bin/native/debugFramework/Demo.framework/Headers` directory, open the `Demo.h` header file. 来看看 Kotlin 的运行时声明：

```
NS_ASSUME_NONNULL_BEGIN
#pragma clang diagnostic push
#pragma clang diagnostic ignored "-Wunknown-warning-option"
#pragma clang diagnostic ignored "-Wincompatible-property-type"
#pragma clang diagnostic ignored "-Wnullability"

#pragma push_macro("_Nullable_result")
#if !__has_feature(nullability_nullable_result)
#undef _Nullable_result
#define _Nullable_result _Nullable
#endif

__attribute__((swift_name("KotlinBase")))
```

```
- (instancetype)init __attribute__((unavailable));
+ (instancetype)new __attribute__((unavailable));
+ (void)initialize __attribute__((objc_requires_super));
@end

@interface DemoBase (DemoBaseCopying) <NSCopying>
@end

__attribute__((swift_name("KotlinMutableSet")))
@interface DemoMutableSet<ObjectType> : NSMutableSet<ObjectType>
@end

__attribute__((swift_name("KotlinMutableDictionary")))
@interface DemoMutableDictionary<KeyType, ObjectType> : NSMutableDictionary<KeyType, O
bjectType>
@end

@interface NSError (NSErrorDemoKotlinException)
@property (readonly) id _Nullable kotlinException;
@end
```

Kotlin 类在 Swift/Objective-C 中拥有一个 `KotlinBase` 基类，该类在这里继承自 `NSObject` 类。 同样也有集合与异常的包装器。大多数的集合类型都映射到了 Swift/Objective-C 中相似的集合类型：

Kotlin	Swift	Objective-C
List	Array	NSArray
MutableList	NSMutableArray	NSMutableArray
Set	Set	NSSet
MutableSet	NSMutableSet	NSMutableSet
Map	Dictionary	NSDictionary
MutableMap	NSMutableDictionary	NSMutableDictionary

Kotlin 数值与 NSNumber

The next part of the `Demo.h` file contains type mappings between Kotlin/Native number types and `NSNumber`. The base class is called `DemoNumber` in Objective-C and `KotlinNumber` in Swift. It extends `NSNumber`.

For each Kotlin number type, there is a corresponding predefined child class:

Kotlin	Swift	Objective-C	Simple type
-	KotlinNumber	<Package>Number	-
Byte	KotlinByte	<Package>Byte	char

UByte	KotlinUByte	<Package>UByte	unsigned char
Short	KotlinShort	<Package>Short	short
UShort	KotlinUShort	<Package>UShort	unsigned short
Int	KotlinInt	<Package>Int	int
UInt	KotlinUInt	<Package>UInt	unsigned int
Long	KotlinLong	<Package>Long	long long
ULong	KotlinULong	<Package>ULong	unsigned long long
Float	KotlinFloat	<Package>Float	float
Double	KotlinDouble	<Package>Double	double
Boolean	KotlinBoolean	<Package>Boolean	BOOL/Bool

每个数字类型都有一个类方法，用于从对应的简单类型创建新实例。此外，还有一个实例方法用于提取一个简单的值。原理上，所有这样的声明看起来像这样：

```
__attribute__((swift_name("Kotlin__TYPE__")))
@interface Demo__TYPE__ : DemoNumber
- (instancetype)initWith__TYPE__:(__CTYPE__)value;
+ (instancetype)numberWith__TYPE__:(__CTYPE__)value;
@end;
```

这里，`__TYPE__` 是简单类型的名称之一，而 `__CTYPE__` 是对应的 Objective-C 类型， 例如 `initWithChar(char)` 。

这些类型用于将装箱的 Kotlin 数字类型映射到 Swift/Objective-C。 在 Swift 中，可以调用构造函数来创建一个实例，例如 `KotlinLong(value: 42)` 。

Kotlin 中的类与对象

我们来看看如何将 `class` 与 `object` 映射到 Swift/Objective-C。生成的 `Demo.h` 文件包含 `Class` 、 `Interface` 与 `Object` 的确切定义：

```
__attribute__((swift_name("Interface")))
@protocol DemoInterface
@required
- (void)iMember __attribute__((swift_name("iMember()")));
@end

__attribute__((objc_subclassing_restricted))
__attribute__((swift_name("Clazz")))
@interface DemoClazz : DemoBase <DemoInterface>
- (instancetype)init __attribute__((swift_name("init()"))) __attribute__((objc_designa
ted_initializer));
+ (instancetype)new __attribute__((availability(swift, unavailable, message="use objec
t initializers instead")));
- (DemoULong * _Nullable)memberP:(int32_t)p __attribute__((swift_name("member(p:)")));
```

```
@end

__attribute__((objc_subclassing_restricted))
__attribute__((swift_name("Object")))
@interface DemoObject : DemoBase
+ (instancetype)alloc __attribute__((unavailable));
+ (instancetype)allocWithZone:(struct _NSZone *)zone __attribute__((unavailable));
+ (instancetype)object __attribute__((swift_name("init()")));
@property (class, readonly, getter=shared) DemoObject *shared __attribute__((swift_nam
e("shared")));
@property (readonly) NSString *field __attribute__((swift_name("field")));
@end
```

这段代码中的 Objective-C attribute 有助于在 Swift 和 Objective-C 语言中使用该 framework。`DemoInterface`、`DemoClazz` 和 `DemoObject` 分别被创建为 `Interface`、`Clazz` 和 `Object`。

`Interface` 会转换为 `@protocol`，而 `class` 和 `object` 都以 `@interface` 表示。`Demo` 前缀来自于 framework 的名称。可空的返回值类型 `ULong?` 会转换到 Objective-C 中的 `DemoULong`。

Kotlin 中的全局声明

所有 Kotlin 中的全局函数都会转化为 Objective-C 中的 `DemoLibKt` 以及 Swift 中的 `LibKt`，其中 `Demo` 是由 `kotlinc-native` 的 `-output` 参数设置的 framework 名称：

```
__attribute__((objc_subclassing_restricted))
__attribute__((swift_name("LibKt")))
@interface DemoLibKt : DemoBase
+ (NSString * _Nullable)acceptFunF:(NSString * _Nullable (^)(NSString *))f __attribute
__((swift_name("acceptFun(f:)")));
+ (void)forFloatsF:(float)f d:(DemoDouble * _Nullable)d __attribute__((swift_name("for
Floats(f:d:)")));
+ (void)forIntegersB:(int8_t)b s:(uint16_t)s i:(int32_t)i l:(DemoULong * _Nullable)l _
_attribute__((swift_name("forIntegers(b:s:i:l:)")));
+ (NSString *)stringsStr:(NSString * _Nullable)str __attribute__((swift_name("strings(
str:)")));
+ (NSString * _Nullable (^)(NSString *))supplyFun __attribute__((swift_name("supplyFun
()")));
@end
```

Kotlin `String` 与 Objective-C `NSString*` 是透明映射的。类似地，Kotlin 的 `Unit` 类型会映射到 `void`。原生类型直接映射。不可空的原生类型透明地映射。可空的原生类型被映射到 `Kotlin<TYPE>*` 类型，如表中所示。包括高阶函数 `acceptFunF` 与 `supplyFun` 都接收 Objective-C 块。

可以在 Swift/Objective-C 互操作性 中找到关于类型映射的更多信息。

垃圾回收与引用计数

Swift 与 Objective-C 使用自动引用计数（ARC）。Kotlin/Native 拥有自己的垃圾回收，它也与 Swift/Objective-C ARC 相集成。

Unused Kotlin objects are automatically removed. You don't need to take additional steps to control the lifetime of Kotlin/Native instances from Swift or Objective-C.

在 Objective-C 中使用代码

我们来在 Objective-C 中调用 framework。在 framework 目录中，使用以下代码创建 `main.m` 文件：

```
#import <Foundation/Foundation.h>
#import <Demo/Demo.h>

int main(int argc, const char * argv[]) {
    @autoreleasepool {
        [DemoObject.shared field];

        DemoClazz* clazz = [[ DemoClazz alloc] init];
        [clazz memberP:42];

        [DemoLibKt forIntegersB:1 s:1 i:3 l:[DemoULong numberWithUnsignedLongLong:4]];
        [DemoLibKt forIntegersB:1 s:1 i:3 l:nil];

        [DemoLibKt forFloatsF:2.71 d:[DemoDouble numberWithDouble:2.71]];
        [DemoLibKt forFloatsF:2.71 d:nil];

        NSString* ret = [DemoLibKt acceptFunF:^NSString * _Nullable(NSString * it) {
            return [it stringByAppendingString:@" Kotlin is fun"];
        }];

        NSLog(@"%@", ret);
        return 0;
    }
}
```

这里，您可以直接在 Objective-C 代码中调用 Kotlin 类。Kotlin 对象使用 `<对象名称>.shared` 类属性，这让你可以获取对象的唯一实例并在其上调用对象方法。

广泛使用的模式是用于创建 `Clazz` 类的实例。在 Objective-C 上调用 `[[ DemoClazz alloc] init]`。也可以使用没有参数的构造函数 `[DemoClazz new]`。

Kotlin 源中的全局声明的作用域位于 Objective-C 中的 `DemoLibKt` 类之下。所有 Kotlin 函数都被转化为该类的类方法。

`strings` 函数转化为 Objective-C 中的 `DemoLibKt.stringsStr` 函数，因此可以给它直接传递 `NSString`。返回值也同样可见为 `NSString`。

在 Swift 中使用代码

生成的 framework 拥有辅助 attribute，使其在 Swift 中使用更加容易。Let's convert the previous Objective-C example into Swift.

In the framework directory, create the `main.swift` file with the following code:

```
import Foundation
import Demo

let kotlinObject = Object.shared

let field = Object.shared.field

let clazz = Clazz()
clazz.member(p: 42)

LibKt.forIntegers(b: 1, s: 2, i: 3, l: 4)
LibKt.forFloats(f: 2.71, d: nil)

let ret = LibKt.acceptFun { "\($0) Kotlin is fun" }
if (ret != nil) {
    print(ret!)
}
```

There are some small differences between the original Kotlin code and its Swift version. In Kotlin, any object declaration has only one instance. The `Object.shared` syntax is used to access this single instance.

Kotlin function and property names are translated as is. Kotlin's `String` is turned into Swift's `String`. Swift hides `NSNumber*` boxing too. You can also pass a Swift closure to Kotlin and call a Kotlin lambda function from Swift.

You can find more information about type mapping in Interoperability with Swift/Objective-C.

Connect the framework to your iOS project

Now you can connect the generated framework to your iOS project as a dependency. There are multiple ways to set it up and automate the process, choose the method that suits you best:

Choose iOS integration method →

下一步做什么

- Learn more about interoperability with Objective-C
- See how interoperability with C is implemented in Kotlin
- Check out the Kotlin/Native as a dynamic library tutorial

Kotlin/Native 库

Kotlin 编译器细节

用 Kotlin/Native 编译器生成一个库，请使用 `-produce library` 或者 `-p library` 标志。例如：

```
$ kotlinc-native foo.kt -p library -o bar
```

这个命令会生成一个带有 `foo.kt` 编译后的内容的库 `bar.klib`。

链接到一个库请使用 `-library <库名>` or `-l <库名>` 标志。例如：

```
$ kotlinc-native qux.kt -l bar
```

这个命令会由 `qux.kt` 与 `bar.klib` 生成 `program.kexe`

cinterop 工具细节

cinterop 工具为原生库生成 `.klib` 包装作为其主要输出。 例如，使用 Kotlin/Native 发行版中提供的简单 `libgit2.def` 原生库定义文件

```
$ cinterop -def samples/gitchurn/src/nativeInterop/cinterop/libgit2.def -compiler-opti
on -I/usr/local/include -o libgit2
```

会得到 `libgit2.klib`。

更多详情请参见 C Interop.

klib 实用程序

klib 库管理实用程序可以探查与安装库。

可以使用以下命令：

- `content` – list library contents:

  ```
  $ klib contents <库名>
  ```

- `info` ——探查库的簿记细节

```
$ klib info <库名>
```

- `install` ——将库安装到默认位置，使用

```
$ klib install <库名>
```

- `remove` ——将库从默认存储库中删除，使用

```
$ klib remove <库名>
```

上述所有命令都接受一个额外的 `-repository <目录>` 参数，用于指定与默认不同的存储库。

```
$ klib <命令> <库名> -repository <目录>
```

几个示例

首先创建一个库。 将微型库的源代码写到 `kotlinizer.kt` 中：

```
package kotlinizer
val String.kotlinized
    get() = "Kotlin $this"
```

```
$ kotlinc-native kotlinizer.kt -p library -o kotlinizer
```

该库已在当前目录中创建：

```
$ ls kotlinizer.klib
kotlinizer.klib
```

现在来看看库的内容：

```
$ klib contents kotlinizer
```

可以将 `kotlinizer` 安装到默认存储库：

```
$ klib install kotlinizer
```

从当前目录中删除它的任何痕迹：

```
$ rm kotlinizer.klib
```

创建一个非常短的程序并写到 `use.kt` 中：

```
import kotlinizer.*

fun main(args: Array<String>) {
    println("Hello, ${"world".kotlinized}!")
}
```

现在编译该程序链接到刚刚创建的库：

```
$ kotlinc-native use.kt -l kotlinizer -o kohello
```

并运行该程序：

```
$ ./kohello.kexe
Hello, Kotlin world!
```

乐在其中！

高级主题

库搜索顺序

当给出 `-library foo` 标志时，编译器按照以下顺序搜索 `foo` 库：

- 当前编译目录或者一个绝对路径。
- 以 `-repo` 标志指定的所有存储库。
- 安装在默认存储库。

> The default repository is `~/.konan`. You can change it by setting the `kotlin.data.dir` Gradle property.
>
> Alternatively, you can use the `-Xkonan-data-dir` compiler option to configure your custom path to the directory via the `cinterop` and `konanc` tools.
>
> 

- 安装在 `$installation/klib` 目录中的库。

库格式

Kotlin/Native 是包含预定义目录结构的 zip 文件， 具有以下布局：

当 `foo.klib` 解压为 `foo/` 时会有：

```
- foo/
  - $component_name/
    - ir/
      - 序列化的 Kotlin IR。
    - targets/
      - $platform/
        - kotlin/
          - Kotlin 编译为 LLVM 位码（bitcode）。
        - native/
          - 附加原生对象的位码文件。
      - $another_platform/
        - 可以有几个平台相关的 kotlin 与原生对。
    - linkdata/
      - 一组带有序列化的链接元数据的 ProtoBuf 文件。
    - resources/
      - 图像等普通资源。(尚未使用)。
  - manifest—描述库的 java 属性格式文件。
```

可以在安装的 `klib/stdlib` 目录中找到示例布局。

Using relative paths in klibs

> Using relative paths in klibs is available since Kotlin 1.6.20.
>
> 

A serialized IR representation of source files is a part of a `klib` library. It includes paths of files for generating proper debug information. By default, stored paths are absolute. With the `-Xklib-relative-path-base` compiler option, you can change the format and use only relative path in the artifact. To make it work, pass one or multiple base paths of source files as an argument:

> **【Kotlin】**

```
import org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTask
// ...

tasks.named<KotlinCompilationTask<*>>("compileKotlin").configure {
```

```
    // $base is a base path of source files
    compilerOptions.freeCompilerArgs.add("-Xklib-relative-path-base=$base")
}
```

> 【Groovy】

```
import org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTask
// ...

tasks.named('compileKotlin', KotlinCompilationTask) {
    compilerOptions {
        // $base is a base path of source files
        freeCompilerArgs.add("-Xklib-relative-path-base=$base")
    }
}
```

平台库

为了提供对操作系统原生服务的访问，Kotlin/Native 发行版包含了一组为每个目标平台预构建的库。 这些被称为平台库。

The packages from platform libraries are available by default. You don't need to specify additional link options to use them. The Kotlin/Native compiler automatically detects which platform libraries are accessed and links the necessary ones.

However, platform libraries in the compiler distribution are merely wrappers and bindings to the native libraries. That means you need to install native libraries themselves (`.so` , `.a` , `.dylib` , `.dll` , and so on) on your local machine.

POSIX 绑定

Kotlin provides the POSIX platform library for all UNIX- and Windows-based targets, including Android and iOS. These platform libraries contain bindings to the platform's implementation, which follows the POSIX standard.

To use the library, import it into your project:

```
import platform.posix.*
```

> The `platform.posix` contents differ across platforms because of variations in POSIX implementations.
>
> 

You can explore the contents of the `posix.def` file for each supported platform here:

- iOS
- macOS
- tvOS
- watchOS
- Linux
- Windows (MinGW)
- Android

The POSIX platform library is not available for the WebAssembly target.

Popular native libraries

Kotlin/Native provides bindings for various popular native libraries that are commonly used on different platforms, such as OpenGL, zlib, and Foundation.

On Apple platforms, the `objc` library is included to enable interoperability with Objective-C APIs.

You can explore the native libraries available for Kotlin/Native targets in your compiler distribution, depending on your setup:

- If you installed a standalone Kotlin/Native compiler:
    1. Go to the unpacked archive with the compiler distribution, for example, `kotlin-native-prebuilt-macos-aarch64-2.1.0`.
    2. Navigate to the `klib/platform` directory.
    3. Choose the folder with the corresponding target.
- If you use the Kotlin plugin in your IDE (bundled with IntelliJ IDEA and Android Studio):
    1. In your command line tool, run the following to navigate to the `.konan` folder:

> 【macOS and Linux】

```
~/.konan/
```

> 【Windows】

```
%\USERPROFILE%\.konan
```

---

1. Open the Kotlin/Native compiler distribution, for example, `kotlin-native-prebuilt-macos-aarch64-2.1.0`.
2. Navigate to the `klib/platform` directory.
3. Choose the folder with the corresponding target.

> If you'd like to explore the definition file for each supported platform library: in the compiler distribution folder, navigate to the `konan/platformDef` directory and choose the necessary target.
>
> 

下一步做什么

Learn more about interoperability with Swift/Objective-C

内存管理器

- Kotlin/Native 内存管理
- 与 ARC 集成
- 迁移到新版内存管理器

Kotlin/Native 内存管理

Kotlin/Native uses a modern memory manager that is similar to the JVM, Go, and other mainstream technologies, including the following features:

- Objects are stored in a shared heap and can be accessed from any thread.
- Tracing garbage collection is performed periodically to collect objects that are not reachable from the "roots", like local and global variables.

Garbage collector

Kotlin/Native's garbage collector (GC) algorithm is constantly evolving. Currently, it functions as a stop-the-world mark and concurrent sweep collector that does not separate the heap into generations.

The GC is executed on a separate thread and started based on the memory pressure heuristics or by a timer. Alternatively, it can be called manually.

The GC processes the mark queue on several threads in parallel, including application threads, the GC thread, and optional marker threads. Application threads and at least one GC thread participate in the marking process. By default, application threads must be paused when the GC is marking objects in the heap.

> You can disable the parallelization of the mark phase with the `kotlin.native.binary.gcMarkSingleThreaded=true` compiler option. However, this may increase the garbage collector's pause time on large heaps.
>
> 

When the marking phase is completed, the GC processes weak references and nullifies reference points to an unmarked object. By default, weak references are processed concurrently to decrease the GC pause time.

See how to monitor and optimize garbage collection.

Enable garbage collection manually

To force-start the garbage collector, call `kotlin.native.internal.GC.collect()`. This method triggers a new collection and waits for its completion.

Monitor GC performance

To monitor the GC performance, you can look through its logs and diagnose issues. To enable logging, set the following compiler option in your Gradle build script:

```
-Xruntime-logs=gc=info
```

Currently, the logs are only printed to `stderr`.

On Apple platforms, you can take advantage of the Xcode Instruments toolkit to debug iOS app performance. The garbage collector reports pauses with signposts available in Instruments. Signposts enable custom logging within your app, allowing you to check if a GC pause corresponds to an application freeze.

To track GC-related pauses in your app:

1. To enable the feature, set the following compiler option in your `gradle.properties` file:

   ```
   kotlin.native.binary.enableSafepointSignposts=true
   ```

2. Open Xcode, go to **Product** | **Profile** or press Cmd + I. This action compiles your app and launches Instruments.
3. In the template selection, select **os_signpost**.
4. Configure it by specifying `org.kotlinlang.native.runtime` as **subsystem** and `safepoint` as **category**.
5. Click the red record button to run your app and start recording signpost events:



Here, each blue blob on the lowest graph represents a separate signpost event, which is a GC pause.

Optimize GC performance

To improve GC performance, you can enable concurrent marking to decrease the GC pause time. This allows the marking phase of garbage collection to run simultaneously with application threads.

The feature is currently Experimental. To enable it, set the following compiler option in your `gradle.properties` file:

```
kotlin.native.binary.gc=cms
```

Disable garbage collection

It's recommended to keep the GC enabled. However, you can disable it in certain cases, such as for testing purposes or if you encounter issues and have a short-lived program. To do so, set the following binary option in your `gradle.properties` file:

```
kotlin.native.binary.gc=noop
```

> With this option enabled, the GC doesn't collect Kotlin objects, so memory consumption will keep rising as long as the program runs. Be careful not to exhaust the system memory.
>
> 

Memory consumption

Kotlin/Native uses its own memory allocator. It divides system memory into pages, allowing independent sweeping in consecutive order. Each allocation becomes a memory block within a page, and the page keeps track of block sizes. Different page types are optimized for various allocation sizes. The consecutive arrangement of memory blocks ensures efficient iteration through all allocated blocks.

When a thread allocates memory, it searches for a suitable page based on the allocation size. Threads maintain a set of pages for different size categories. Typically, the current page for a given size can accommodate the allocation. If not, the thread requests a different page from the shared allocation space. This page may already be available, require sweeping, or have to be created first.

The Kotlin/Native memory allocator comes with protection against sudden spikes in memory allocations. It prevents situations where the mutator starts to allocate a lot of garbage quickly and the GC thread cannot keep up with it, making the memory usage grow endlessly. In this case, the GC forces a stop-the-world phase until the iteration is completed.

You can monitor memory consumption yourself, check for memory leaks, and adjust memory consumption.

Check for memory leaks

To access the memory manager metrics, call `kotlin.native.internal.GC.lastGCInfo()`. This method returns statistics for the last run of the garbage collector. The statistics can be useful for:

- Debugging memory leaks when using global variables
- Checking for leaks when running tests

```
import kotlin.native.internal.*
import kotlin.test.*

class Resource

val global = mutableListOf<Resource>()

@OptIn(ExperimentalStdlibApi::class)
fun getUsage(): Long {
```

```
    GC.collect()
    return GC.lastGCInfo!!.memoryUsageAfter["heap"]!!.totalObjectsSizeBytes
}

fun run() {
    global.add(Resource())
    // The test will fail if you remove the next line
    global.clear()
}

@Test
fun test() {
    val before = getUsage()
    // A separate function is used to ensure that all temporary objects are cleared
    run()
    val after = getUsage()
    assertEquals(before, after)
}
```

Adjust memory consumption

If there are no memory leaks in the program, but you still see unexpectedly high memory consumption, try updating Kotlin to the latest version. We're constantly improving the memory manager, so even a simple compiler update might improve memory consumption.

If you continue to experience high memory consumption after updating, switch to the system memory allocator by using the following compiler option in your Gradle build script:

```
-Xallocator=std
```

If this doesn't improve your memory consumption, report an issue in YouTrack.

Unit tests in the background

In unit tests, nothing processes the main thread queue, so don't use `Dispatchers.Main` unless it was mocked. Mocking it can be done by calling `Dispatchers.setMain` from `kotlinx-coroutines-test`.

If you don't rely on `kotlinx.coroutines` or if `Dispatchers.setMain` doesn't work for you for some reason, try the following workaround for implementing the test launcher:

```
package testlauncher

import platform.CoreFoundation.*
import kotlin.native.concurrent.*
import kotlin.native.internal.test.*
import kotlin.system.*
```

```
fun mainBackground(args: Array<String>) {
    val worker = Worker.start(name = "main-background")
    worker.execute(TransferMode.SAFE, { args.freeze() }) {
        val result = testLauncherEntryPoint(it)
        exitProcess(result)
    }
    CFRunLoopRun()
    error("CFRunLoopRun should never return")
}
```

Then, compile the test binary with the `-e testlauncher.mainBackground` compiler option.

下一步做什么

- Migrate from the legacy memory manager
- Check the specifics of integration with Swift/Objective-C ARC

与 Swift/Objective-C ARC 集成

Kotlin and Objective-C use different memory management strategies. Kotlin has a tracing garbage collector, while Objective-C relies on automatic reference counting (ARC).

The integration between these strategies is usually seamless and generally requires no additional work. However, there are some specifics you should keep in mind:

Threads

Deinitializers

Deinitialization on the Swift/Objective-C objects and the objects they refer to is called on the main thread if these objects are passed to Kotlin on the main thread, for example:

```
// Kotlin
class KotlinExample {
    fun action(arg: Any) {
        println(arg)
    }
}
```

```
// Swift
class SwiftExample {
    init() {
        print("init on \(Thread.current)")
    }

    deinit {
        print("deinit on \(Thread.current)")
    }
}

func test() {
    KotlinExample().action(arg: SwiftExample())
}
```

The resulting output:

```
init on <_NSMainThread: 0x600003bc0000>{number = 1, name = main}
shared.SwiftExample
deinit on <_NSMainThread: 0x600003bc0000>{number = 1, name = main}
```

Deinitialization on the Swift/Objective-C objects is called on a special GC thread instead of the main one if:

- Swift/Objective-C objects are passed to Kotlin on a thread other than main.
- The main dispatch queue isn't processed.

If you want to call deinitialization on a special GC thread explicitly, set `kotlin.native.binary.objcDisposeOnMain=false` in your `gradle.properties`. This option enables deinitialization on a special GC thread, even if Swift/Objective-C objects were passed to Kotlin on the main thread.

A special GC thread complies with the Objective-C runtime, meaning that it has a run loop and drain autorelease pools.

Completion handlers

When calling Kotlin suspending functions from Swift, completion handlers might be called on threads other than the main one, for example:

```
// Kotlin
// coroutineScope, launch, and delay are from kotlinx.coroutines
suspend fun asyncFunctionExample() = coroutineScope {
    launch {
        delay(1000L)
        println("World!")
    }
    println("Hello")
}
```

```
// Swift
func test() {
    print("Running test on \(Thread.current)")
    PlatformKt.asyncFunctionExample(completionHandler: { _ in
        print("Running completion handler on \(Thread.current)")
    })
}
```

The resulting output:

```
Running test on <_NSMainThread: 0x600001b100c0>{number = 1, name = main}
Hello
World!
Running completion handler on <NSThread: 0x600001b45bc0>{number = 7, name = (null)}
```

Garbage collection and lifecycle

Object reclamation

An object is reclaimed only during the garbage collection. This applies to Swift/Objective-C objects that cross interop boundaries into Kotlin/Native, for example:

```
// Kotlin
class KotlinExample {
    fun action(arg: Any) {
        println(arg)
    }
}
```

```
// Swift
class SwiftExample {
    deinit {
        print("SwiftExample deinit")
    }
}

func test() {
    swiftTest()
    kotlinTest()
}

func swiftTest() {
    print(SwiftExample())
    print("swiftTestFinished")
}

func kotlinTest() {
    KotlinExample().action(arg: SwiftExample())
    print("kotlinTest finished")
}
```

The resulting output:

```
shared.SwiftExample
SwiftExample deinit
swiftTestFinished
shared.SwiftExample
kotlinTest finished
SwiftExample deinit
```

Objective-C objects lifecycle

The Objective-C objects might live longer than they should, which sometimes might cause performance issues. For example, when a long-running loop creates several temporary objects that cross the Swift/Objective-C interop boundary on each iteration.

In the GC logs, there's a number of stable refs in the root set. If this number keeps growing, it may indicate that the Swift/Objective-C objects are not freed up when they should. In this case, try the `autoreleasepool` block around loop bodies that do interop calls:

```
// Kotlin
fun growingMemoryUsage() {
    repeat(Int.MAX_VALUE) {
        NSLog("$it\n")
    }
}

fun steadyMemoryUsage() {
    repeat(Int.MAX_VALUE) {
        autoreleasepool {
            NSLog("$it\n")
        }
    }
}
```

Garbage collection of Swift and Kotlin objects' chains

Consider the following example:

```
// Kotlin
interface Storage {
    fun store(arg: Any)
}

class KotlinStorage(var field: Any? = null) : Storage {
    override fun store(arg: Any) {
        field = arg
    }
}

class KotlinExample {
    fun action(firstSwiftStorage: Storage, secondSwiftStorage: Storage) {
        // Here, we create the following chain:
        // firstKotlinStorage -> firstSwiftStorage -> secondKotlinStorage -> secondSwi
ftStorage.
        val firstKotlinStorage = KotlinStorage()
        firstKotlinStorage.store(firstSwiftStorage)
        val secondKotlinStorage = KotlinStorage()
        firstSwiftStorage.store(secondKotlinStorage)
        secondKotlinStorage.store(secondSwiftStorage)
    }
```

```
}
```

```
// Swift
class SwiftStorage : Storage {

    let name: String

    var field: Any? = nil

    init(_ name: String) {
        self.name = name
    }

    func store(arg: Any) {
        field = arg
    }

    deinit {
        print("deinit SwiftStorage \(name)")
    }
}

func test() {
    KotlinExample().action(
        firstSwiftStorage: SwiftStorage("first"),
        secondSwiftStorage: SwiftStorage("second")
    )
}
```

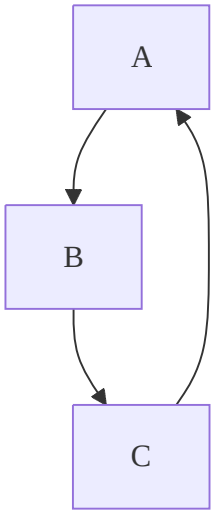
It takes some time between "deinit SwiftStorage first" and "deinit SwiftStorage second" messages to appear in the log. The reason is that `firstKotlinStorage` and `secondKotlinStorage` are collected in different GC cycles. Here's the sequence of events:

1. `KotlinExample.action` finishes. `firstKotlinStorage` is considered "dead" because nothing references it, while `secondKotlinStorage` is not because it is referenced by `firstSwiftStorage`.
2. First GC cycle starts, and `firstKotlinStorage` is collected.
3. There are no references to `firstSwiftStorage`, so it is "dead" as well, and `deinit` is called.
4. Second GC cycle starts. `secondKotlinStorage` is collected because `firstSwiftStorage` is no longer referencing it.
5. `secondSwiftStorage` is finally reclaimed.

It requires two GC cycles to collect these four objects because deinitialization of Swift and Objective-C objects happens after the GC cycle. The limitation stems from `deinit`, which can call arbitrary code, including the Kotlin code that cannot be run during the GC pause.

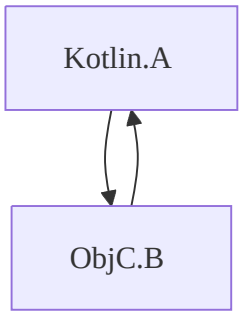
Retain cycles

In a *retain cycle*, a number of objects refer each other using strong references cyclically:


A
B
C

Kotlin's tracing GC and Objective-C's ARC handle retain cycles differently. When objects become unreachable, Kotlin's GC can properly reclaim such cycles, while Objective-C's ARC cannot. Therefore, retain cycles of Kotlin objects can be reclaimed, while retain cycles of Swift/Objective-C objects cannot.

Consider the case when a retain cycle contains both Objective-C and Kotlin objects:


Kotlin.A
ObjC.B

This involves combining Kotlin's and Objective-C's memory management models that cannot handle (reclaim) retain cycles together. That means if at least one Objective-C object is present, the retain cycle of a whole graph of objects cannot be reclaimed, and it's impossible to break the cycle from the Kotlin side.

Unfortunately, no special instruments are currently available to automatically detect retain cycles in Kotlin/Native code. To avoid retain cycles, use weak or unowned references.

Support for background state and App Extensions

The current memory manager does not track application state by default and does not integrate with App Extensions out of the box.

It means that the memory manager doesn't adjust GC behavior accordingly, which might be harmful in some cases. To change this behavior, add the following Experimental binary option to your `gradle.properties`:

```
kotlin.native.binary.appStateTracking=enabled
```

It turns off a timer-based invocation of the garbage collector when the application is in the background, so GC is called only when memory consumption becomes too high.

下一步做什么

Learn more about Swift/Objective-C interoperability.

迁移到新版内存管理器

> Support for the legacy memory manager has been completely removed in Kotlin 1.9.20. Migrate your projects to the current memory model, enabled by default since Kotlin 1.7.20.
>
> 

This guide compares the new Kotlin/Native memory manager with the legacy one and describes how to migrate your projects.

The most noticeable change in the new memory manager is lifting restrictions on object sharing. You don't need to freeze objects to share them between threads, specifically:

- Top-level properties can be accessed and modified by any thread without using `@SharedImmutable`.
- Objects passing through interop can be accessed and modified by any thread without freezing them.
- `Worker.executeAfter` no longer requires operations to be frozen.
- `Worker.execute` no longer requires producers to return an isolated object subgraph.
- Reference cycles containing `AtomicReference` and `FreezableAtomicReference` do not cause memory leaks.

Apart from easy object sharing, the new memory manager also brings other major changes:

- Global properties are initialized lazily when the file they are defined in is accessed first. Previously global properties were initialized at the program startup. As a workaround, you can mark properties that must be initialized at the program start with the `@EagerInitialization` annotation. Before using, check its documentation.
- `by lazy {}` properties support thread-safety modes and do not handle unbounded recursion.
- Exceptions that escape `operation` in `Worker.executeAfter` are processed like in other runtime parts, by trying to execute a user-defined unhandled exception hook or terminating the program if the hook was not found or failed with an exception itself.
- Freezing is deprecated and always disabled.

Follow these guidelines to migrate your projects from the legacy memory manager:

Update Kotlin

The new Kotlin/Native memory manager has been enabled by default since Kotlin 1.7.20. Check the Kotlin version and update to the latest one if necessary.

Update dependencies

Update to version 1.6.0 or later. Do not use versions with the `native-mt` suffix.

There are also some specifics with the new memory manager you should keep in mind:

- Every common primitive (channels, flows, coroutines) works through the Worker boundaries, since freezing is not required.
- `Dispatchers.Default` is backed by a pool of Workers on Linux and Windows and by a global queue on Apple targets.
- Use `newSingleThreadContext` to create a coroutine dispatcher that is backed by a Worker.
- Use `newFixedThreadPoolContext` to create a coroutine dispatcher backed by a pool of `N` Workers.
- `Dispatchers.Main` is backed by the main queue on Darwin and by a standalone Worker on other platforms.

Update to version 2.0 or later.
The majority of libraries should work without any changes, however, there might be exceptions.

Make sure that you update dependencies to the latest versions, and there is no difference between library versions for the legacy and the new memory manager.

Update your code

To support the new memory manager, remove usages of the affected API:

Old API	What to do
`@SharedImmutable`	You can remove all usages, though there are no warnings for using this API in the new memory manager.
The `FreezableAtomicReference` class	Use `AtomicReference` instead.
The `FreezingException` class	Remove all usages.
The `InvalidMutabilityException` class	Remove all usages.
The `IncorrectDereferenceException` class	Remove all usages.
The `freeze()` function	Remove all usages.
The `isFrozen` property	You can remove all usages. Since freezing is deprecated, the property always returns `false`.

The `ensureNeverFrozen()` function	Remove all usages.
The `atomicLazy()` function	Use `lazy()` instead.
The `MutableData` class	Use any regular collection instead.
The `WorkerBoundReference<out T : Any>` class	Use `T` directly.
The `DetachedObjectGraph<T>` class	Use `T` directly. To pass the value through the C interop, use the StableRef class.

下一步做什么

- Learn more about the new memory manager
- Check the specifics of integration with Swift/Objective-C ARC
- Learn how to reference objects safely from different coroutines

调试 Kotlin/Native

Currently, the Kotlin/Native compiler produces debug info compatible with the DWARF 2 specification, so modern debugger tools can perform the following operations:

- breakpoints
- stepping
- inspection of type information
- variable inspection

> Supporting the DWARF 2 specification means that the debugger tool recognizes Kotlin as C89, because before the DWARF 5 specification, there is no identifier for the Kotlin language type in specification.
>
> 

Produce binaries with debug info with Kotlin/Native compiler

To produce binaries with the Kotlin/Native compiler, use the `-g` option on the command line.

```
0:b-debugger-fixes:minamoto@unit-703(0)# cat - > hello.kt
fun main(args: Array<String>) {
  println("Hello world")
  println("I need your clothes, your boots and your motocycle")
}
0:b-debugger-fixes:minamoto@unit-703(0)# dist/bin/konanc -g hello.kt -o terminator
KtFile: hello.kt
0:b-debugger-fixes:minamoto@unit-703(0)# lldb terminator.kexe
(lldb) target create "terminator.kexe"
Current executable set to 'terminator.kexe' (x86_64).
(lldb) b kfun:main(kotlin.Array<kotlin.String>)
Breakpoint 1: where = terminator.kexe`kfun:main(kotlin.Array<kotlin.String>) + 4 at he
llo.kt:2, address = 0x00000001000012e4
(lldb) r
Process 28473 launched: '/Users/minamoto/ws/.git-trees/debugger-fixes/terminator.kexe'
 (x86_64)
Process 28473 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = breakpoint 1.1
    frame #0: 0x00000001000012e4 terminator.kexe`kfun:main(kotlin.Array<kotlin.String>
) at hello.kt:2
   1    fun main(args: Array<String>) {
-> 2      println("Hello world")
   3      println("I need your clothes, your boots and your motocycle")
```

```
   4    }
(lldb) n
Hello world
Process 28473 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = step over
    frame #0: 0x00000001000012f0 terminator.kexe`kfun:main(kotlin.Array<kotlin.String>
) at hello.kt:3
   1    fun main(args: Array<String>) {
   2      println("Hello world")
-> 3      println("I need your clothes, your boots and your motocycle")
   4    }
(lldb)
```

Breakpoints

Modern debuggers provide several ways to set a breakpoint, see below for a tool-by-tool breakdown:

lldb

- by name

```
  (lldb) b -n kfun:main(kotlin.Array<kotlin.String>)
  Breakpoint 4: where = terminator.kexe`kfun:main(kotlin.Array<kotlin.String>) + 4
 at hello.kt:2, address = 0x00000001000012e4
```

`-n` *is optional, this flag is applied by default*

- by location (filename, line number)

```
  (lldb) b -f hello.kt -l 1
  Breakpoint 1: where = terminator.kexe`kfun:main(kotlin.Array<kotlin.String>) + 4
 at hello.kt:2, address = 0x00000001000012e4
```

- by address

```
  (lldb) b -a 0x00000001000012e4
  Breakpoint 2: address = 0x00000001000012e4
```

- by regex, you might find it useful for debugging generated artifacts, like lambda etc. (where used `#` symbol in name).

```
  3: regex = 'main\(', locations = 1
    3.1: where = terminator.kexe`kfun:main(kotlin.Array<kotlin.String>) + 4 at hel
lo.kt:2, address = terminator.kexe[0x00000001000012e4], unresolved, hit count = 0
```

gdb

- by regex

```
(gdb) rbreak main(
Breakpoint 1 at 0x1000109b4
struct ktype:kotlin.Unit &kfun:main(kotlin.Array<kotlin.String>);
```

- by name **unusable**, because `:` is a separator for the breakpoint by location

```
(gdb) b kfun:main(kotlin.Array<kotlin.String>)
No source file named kfun.
Make breakpoint pending on future shared library load? (y or [n]) y
Breakpoint 1 (kfun:main(kotlin.Array<kotlin.String>)) pending
```

- by location

```
(gdb) b hello.kt:1
Breakpoint 2 at 0x100001704: file /Users/minamoto/ws/.git-trees/hello.kt, line 1
.
```

- by address

```
(gdb) b *0x100001704
Note: breakpoint 2 also set at pc 0x100001704.
Breakpoint 3 at 0x100001704: file /Users/minamoto/ws/.git-trees/hello.kt, line 2
.
```

Stepping

Stepping functions works mostly the same way as for C/C++ programs.

Variable inspection

Variable inspections for `var` variables works out of the box for primitive types. For non-primitive types there are custom pretty printers for lldb in `konan_lldb.py`:

```
λ cat main.kt | nl
     1  fun main(args: Array<String>) {
     2      var x = 1
```

```
     3      var y = 2
     4      var p = Point(x, y)
     5      println("p = $p")
     6  }

     7  data class Point(val x: Int, val y: Int)

λ lldb ./program.kexe -o 'b main.kt:5' -o
(lldb) target create "./program.kexe"
Current executable set to './program.kexe' (x86_64).
(lldb) b main.kt:5
Breakpoint 1: where = program.kexe`kfun:main(kotlin.Array<kotlin.String>) + 289 at mai
n.kt:5, address = 0x000000000040af11
(lldb) r
Process 4985 stopped
* thread #1, name = 'program.kexe', stop reason = breakpoint 1.1
    frame #0: program.kexe`kfun:main(kotlin.Array<kotlin.String>) at main.kt:5
   2        var x = 1
   3        var y = 2
   4        var p = Point(x, y)
-> 5        println("p = $p")
   6    }
   7
   8    data class Point(val x: Int, val y: Int)

Process 4985 launched: './program.kexe' (x86_64)
(lldb) fr var
(int) x = 1
(int) y = 2
(ObjHeader *) p = 0x00000000007643d8
(lldb) command script import dist/tools/konan_lldb.py
(lldb) fr var
(int) x = 1
(int) y = 2
(ObjHeader *) p = [x: ..., y: ...]
(lldb) p p
(ObjHeader *) $2 = [x: ..., y: ...]
(lldb) script lldb.frame.FindVariable("p").GetChildMemberWithName("x").Dereference().G
etValue()
'1'
(lldb)
```

Getting representation of the object variable (var) could also be done using the built-in runtime function `Konan_DebugPrint` (this approach also works for gdb, using a module of command syntax):

```
0:b-debugger-fixes:minamoto@unit-703(0)# cat ../debugger-plugin/1.kt | nl -p
     1  fun foo(a:String, b:Int) = a + b
     2  fun one() = 1
     3  fun main(arg:Array<String>) {
     4    var a_variable = foo("(a_variable) one is ", 1)
     5    var b_variable = foo("(b_variable) two is ", 2)
```

```
     6    var c_variable = foo("(c_variable) two is ", 3)
     7    var d_variable = foo("(d_variable) two is ", 4)
     8    println(a_variable)
     9    println(b_variable)
    10    println(c_variable)
    11    println(d_variable)
    12  }
0:b-debugger-fixes:minamoto@unit-703(0)# lldb ./program.kexe -o 'b -f 1.kt -l 9' -o r
(lldb) target create "./program.kexe"
Current executable set to './program.kexe' (x86_64).
(lldb) b -f 1.kt -l 9
Breakpoint 1: where = program.kexe`kfun:main(kotlin.Array<kotlin.String>) + 463 at 1.k
t:9, address = 0x0000000100000dbf
(lldb) r
(a_variable) one is 1
Process 80496 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = breakpoint 1.1
    frame #0: 0x0000000100000dbf program.kexe`kfun:main(kotlin.Array<kotlin.String>) a
t 1.kt:9
   6      var c_variable = foo("(c_variable) two is ", 3)
   7      var d_variable = foo("(d_variable) two is ", 4)
   8      println(a_variable)
-> 9      println(b_variable)
   10     println(c_variable)
   11     println(d_variable)
   12   }

Process 80496 launched: './program.kexe' (x86_64)
(lldb) expression -- (int32_t)Konan_DebugPrint(a_variable)
(a_variable) one is 1(int32_t) $0 = 0
(lldb)
```

Known issues

- performance of Python bindings.

符号化 iOS 崩溃报告

Debugging an iOS application crash sometimes involves analyzing crash reports. More info about crash reports can be found in the Apple documentation.

Crash reports generally require symbolication to become properly readable: symbolication turns machine code addresses into human-readable source locations. The document below describes some specific details of symbolicating crash reports from iOS applications using Kotlin.

Producing .dSYM for release Kotlin binaries

To symbolicate addresses in Kotlin code (e.g. for stack trace elements corresponding to Kotlin code) `.dSYM` bundle for Kotlin code is required.

By default, Kotlin/Native compiler produces `.dSYM` for release (i.e. optimized) binaries on Darwin platforms. This can be disabled with `-Xadd-light-debug=disable` compiler flag. At the same time, this option is disabled by default for other platforms. To enable it, use the `-Xadd-light-debug=enable` compiler option.

> 【Kotlin】

```
kotlin {
    targets.withType<org.jetbrains.kotlin.gradle.plugin.mpp.KotlinNativeTarget> {
        binaries.all {
            freeCompilerArgs += "-Xadd-light-debug={enable|disable}"
        }
    }
}
```

> 【Groovy】

```
kotlin {
    targets.withType(org.jetbrains.kotlin.gradle.plugin.mpp.KotlinNativeTarget) {
        binaries.all {
            freeCompilerArgs += "-Xadd-light-debug={enable|disable}"
        }
    }
}
```

In projects created from IntelliJ IDEA or AppCode templates these `.dSYM` bundles are then discovered by Xcode automatically.

Make frameworks static when using rebuild from bitcode

Rebuilding Kotlin-produced framework from bitcode invalidates the original `.dSYM`. If it is performed locally, make sure the updated `.dSYM` is used when symbolicating crash reports.

If rebuilding is performed on App Store side, then `.dSYM` of rebuilt *dynamic* framework seems discarded and not downloadable from App Store Connect. In this case, it may be required to make the framework static.

> 【Kotlin】

```
kotlin {
    targets.withType<org.jetbrains.kotlin.gradle.plugin.mpp.KotlinNativeTarget> {
        binaries.withType<org.jetbrains.kotlin.gradle.plugin.mpp.Framework> {
            isStatic = true
        }
    }
}
```

> 【Groovy】

```
kotlin {
    targets.withType(org.jetbrains.kotlin.gradle.plugin.mpp.KotlinNativeTarget) {
        binaries.withType(org.jetbrains.kotlin.gradle.plugin.mpp.Framework) {
            isStatic = true
        }
    }
}
```

参考与提示

- 目标平台支持
- 改进编译时间
- 许可证文件
- Kotlin/Native FAQ

Kotlin/Native 目标平台支持

The Kotlin/Native compiler supports a great number of different targets, though it is hard to provide the same level of support for all of them. This document describes which targets Kotlin/Native supports and breaks them into several tiers depending on how well the compiler supports them.

> We can adjust the number of tiers, the list of supported targets, and their features as we go.
>
> 

Mind the following terms used in tier tables:

- **Gradle target name** is a target name that is used in the Kotlin Multiplatform Gradle plugin to enable the target.
- **Target triple** is a target name according to the `<architecture>-<vendor>-<system>-<abi>` structure that is commonly used by compilers.
- **Running tests** indicates out of the box support for running tests in Gradle and IDE.

  This is only available on a native host for the specific target. For example, you can run `macosX64` and `iosX64` tests only on macOS x86-64 host.

Tier 1

- The target is regularly tested on CI to be able to compile and run.
- We provide a source and binary compatibility between compiler releases.

Gradle target name	Target triple	Running tests	Description
Apple macOS hosts only:			
`macosX64`	`x86_64-apple-macos`		Apple macOS on x86_64 platforms
`macosArm64`	`aarch64-apple-macos`		Apple macOS on Apple Silicon platforms
`iosSimulatorArm64`	`aarch64-apple-ios-simulator`		Apple iOS simulator on Apple Silicon platforms
`iosX64`	`x86_64-apple-ios-simulator`		Apple iOS simulator on x86-64 platforms
`iosArm64`	`aarch64-apple-ios`		Apple iOS and iPadOS on ARM64 platforms

Tier 2

- The target is regularly tested on CI to be able to compile, but may not be automatically tested to be able to run.
- We're doing our best to provide a source and binary compatibility between compiler releases.

Gradle target name	Target triple	Running tests	Description
`linuxX64`	`x86_64-unknown-linux-gnu`		Linux on x86_64 platforms
`linuxArm64`	`aarch64-unknown-linux-gnu`		Linux on ARM64 platforms
Apple macOS hosts only:			
`watchosSimulatorArm64`	`aarch64-apple-watchos-simulator`		Apple watchOS simulator on Apple Silicon platforms
`watchosX64`	`x86_64-apple-watchos-simulator`		Apple watchOS 64-bit simulator on x86_64 platforms
`watchosArm32`	`armv7k-apple-watchos`		Apple watchOS on ARM32 platforms
`watchosArm64`	`arm64_32-apple-watchos`		Apple watchOS on ARM64 platforms with ILP32
`tvosSimulatorArm64`	`aarch64-apple-tvos-simulator`		Apple tvOS simulator on Apple Silicon platforms
`tvosX64`	`x86_64-apple-tvos-simulator`		Apple tvOS simulator on x86_64 platforms
`tvosArm64`	`aarch64-apple-tvos`		Apple tvOS on ARM64 platforms

Tier 3

- The target is not guaranteed to be tested on CI.
- We can't promise a source and binary compatibility between different compiler releases, though such changes for these targets are quite rare.

Gradle target name	Target triple	Running tests	Description
`androidNativeArm32`	`arm-unknown-linux-androideabi`		Android NDK on ARM32 platforms
	`aarch64-unknown-`		Android NDK on ARM64

			platforms
`androidNativeX86`	`i686-unknown-linux-android`		Android NDK on x86 platforms
`androidNativeX64`	`x86_64-unknown-linux-android`		Android NDK on x86_64 platforms
`mingwX64`	`x86_64-pc-windows-gnu`		64-bit Windows 7 and later using MinGW compatibility layer
Apple macOS hosts only:			
`watchosDeviceArm64`	`aarch64-apple-watchos`		Apple watchOS on ARM64 platforms

> The `linuxArm32Hfp` target is deprecated and will be removed in future releases.
>
> 

For library authors

We don't recommend library authors to test more targets or provide stricter guarantees than the Kotlin/Native compiler does. You can use the following approach when considering support for native targets:

- Support all the targets from tier 1, 2, and 3.
- Regularly test targets from tier 1 and 2 that support running tests out of the box.

The Kotlin team uses this approach in the official Kotlin libraries, for example, kotlinx.coroutines and kotlinx.serialization.

改进编译时间的技巧

The Kotlin/Native compiler is constantly receiving updates that improve its performance. With the latest Kotlin/Native compiler and a properly configured build environment, you can significantly improve the compilation times of your projects with Kotlin/Native targets.

Read on for our tips on how to speed up the Kotlin/Native compilation process.

General recommendations

Use the latest version of Kotlin

This way, you always get the latest performance improvements. The most recent Kotlin version is 2.1.21.

Avoid creating huge classes

Try to avoid huge classes that take a long time to compile and load during execution.

Preserve downloaded and cached components between builds

When compiling projects, Kotlin/Native downloads the required components and caches some results of its work to the `$USER_HOME/.konan` directory. The compiler uses this directory for subsequent compilations, making them take less time to complete.

When building in containers (such as Docker) or with continuous integration systems, the compiler may have to create the `~/.konan` directory from scratch for each build. To avoid this step, configure your environment to preserve `~/.konan` between builds. For example, redefine its location using the `kotlin.data.dir` Gradle property.

Alternatively, you can use the `-Xkonan-data-dir` compiler option to configure your custom path to the directory via the `cinterop` and `konanc` tools.

Gradle configuration

The first compilation with Gradle usually takes more time than subsequent ones due to the need to download the dependencies, build caches, and perform additional steps. You should build your project at least twice to get an accurate reading of the actual compilation times.

Below are some recommendations for configuring Gradle for better compilation performance.

Increase Gradle heap size

To increase the Gradle heap size, add `org.gradle.jvmargs=-Xmx3g` to your `gradle.properties` file.

If you use parallel builds, you might need to choose the right number of workers with the `org.gradle.workers.max` property or the `--max-workers` command-line option. The default value is the number of CPU processors.

Build only necessary binaries

Don't run Gradle tasks that build the whole project, such as `build` or `assemble`, unless you really need to. These tasks build the same code more than once, increasing the compilation times. In typical cases, such as running tests from IntelliJ IDEA or starting the app from Xcode, the Kotlin tooling avoids executing unnecessary tasks.

If you have a non-typical case or build configuration, you might need to choose the task yourself:

- `linkDebug*`. To run your code during development, you usually need only one binary, so running the corresponding `linkDebug*` task should be enough.
- `embedAndSignAppleFrameworkForXcode`. Since iOS simulators and devices have different processor architectures, it's a common approach to distribute a Kotlin/Native binary as a universal (fat) framework.

  However, during local development, it's faster to build the `.framework` file only for the platform you're using. To build a platform-specific framework, use the embedAndSignAppleFrameworkForXcode task.

Build only for necessary targets

Similarly to the recommendation above, don't build a binary for all native platforms at once. For example, compiling an XCFramework (using an `*XCFramework` task) builds the same code for all targets, which takes proportionally more time than building for a single target.

If you do need XCFrameworks for your setup, you can reduce the number of targets. For example, you don't need `iosX64` if you don't run this project on iOS simulators on Intel-based Macs.

> Binaries for different targets are built with `linkDebug*$Target` and `linkRelease*$Target` Gradle tasks. You can look for the executed tasks in the build log or in the Gradle build scan by running a Gradle build with the `--scan` option.



Don't build unnecessary release binaries

Kotlin/Native supports two build modes, debug and release. Release is highly optimized, and this takes a lot of time: compilation of release binaries takes an order of magnitude more time than debug binaries.

Apart from an actual release, all these optimizations might be unnecessary in a typical development cycle. If you use a task with `Release` in its name during the development process, consider replacing it with `Debug`. Similarly, instead of running `assembleXCFramework`, you can run `assembleSharedDebugXCFramework`, for example.

> Release binaries are built with `linkRelease*` Gradle tasks. You can check for them in the build log or in the Gradle build scan by running a Gradle build with the `--scan` option.
>
> 

Don't disable Gradle daemon

Don't disable the Gradle daemon without having a good reason. By default, Kotlin/Native runs from the Gradle daemon. When it's enabled, the same JVM process is used, and there is no need to warm it up for each compilation.

Don't use transitive export

Using `transitiveExport = true` disables dead code elimination in many cases, so the compiler has to process a lot of unused code. It increases the compilation time. Instead, use the `export` method explicitly for exporting the required projects and dependencies.

Don't export modules too much

Try to avoid unnecessary module export. Each exported module negatively affects compilation time and binary size.

Use Gradle build caching

Enable the Gradle build cache feature:

- **Local build cache**. For local caching, add `org.gradle.caching=true` to your `gradle.properties` file or run the build with the `--build-cache` option in the command line.
- **Remote build cache**. Learn how to configure the remote build cache for continuous integration environments.

Use Gradle configuration cache

To use the Gradle configuration cache, add `org.gradle.configuration-cache=true` to your `gradle.properties` file.

> Configuration cache also enables running `link*` tasks in parallel which could heavily load the machine, specifically with a lot of CPU cores. This issue will be fixed in KT-70915.
>
> 

Enable previously disabled features

There are Kotlin/Native properties that disable the Gradle daemon and compiler caches:

- `kotlin.native.disableCompilerDaemon=true`
- `kotlin.native.cacheKind=none`
- `kotlin.native.cacheKind.$target=none`, where `$target` is a Kotlin/Native compilation target, for example `iosSimulatorArm64`.

If you had issues with these features before and added these lines to your `gradle.properties` file or Gradle arguments, remove them and check whether the build completes successfully. It is possible that these properties were added previously to work around issues that have already been fixed.

Try incremental compilation of klib artifacts

With incremental compilation, if only a part of the `klib` artifact produced by the project module changes, just a part of `klib` is further recompiled into a binary.

This feature is Experimental. To enable it, add the `kotlin.incremental.native=true` option to your `gradle.properties` file. If you face any problems, create an issue in YouTrack.

Windows configuration

Windows Security may slow down the Kotlin/Native compiler. You can avoid this by adding the `.konan` directory, which is located in `%\USERPROFILE%` by default, to Windows Security exclusions. Learn how to add exclusions to Windows Security.

Kotlin/Native 二进制文件的许可证文件

Like many other open-source projects, Kotlin relies on third-party code, meaning that the Kotlin project includes some code not developed by JetBrains or the Kotlin programming language contributors. Sometimes it is derived work, such as code rewritten from C++ to Kotlin.

> You can find licenses for the third-party work used in Kotlin in our GitHub repository:
>
> - Kotlin compiler
> - Kotlin/Native
>
> 

In particular, the Kotlin/Native compiler produces binaries that can include third-party code, data, or derived work. This means that the Kotlin/Native-compiled binaries are subject to the terms and conditions of the third-party licenses.

In practice, if you distribute a Kotlin/Native-compiled final binary, you should always include necessary license files in your binary distribution. The files should be accessible to users of your distribution in a readable form.

Always include the following license files for the corresponding projects:

Project	Files to be included
Kotlin Apache Harmony GWT Guava	• Apache license 2.0 • Apache Harmony copyright notice
libbacktrace	3-clause BSD license with copyright notice
mimalloc	MIT license Include in case you use the mimalloc memory allocator instead of the default one (the `-Xallocator=mimalloc` compiler option is set).
Unicode character database	Unicode license
Multi-producer/multi-consumer bounded queue	Copyright notice

The `mingwX64` target requires additional license files:

Project	Files to be included
MinGW-w64 headers and runtime libraries	• MinGW-w64 runtime license • Winpthreads license

> None of these libraries require the distributed Kotlin/Native binaries to be open-sourced.
>
> 

Kotlin/Native FAQ

How do I run my program?

Define a top-level function `fun main(args: Array<String>)` or just `fun main()` if you are not interested in passed arguments, please ensure it's not in a package. Also, compiler switch `-entry` could be used to make any function taking `Array<String>` or no arguments and return `Unit` as an entry point.

What is Kotlin/Native memory management model?

Kotlin/Native uses an automated memory management scheme that is similar to what Java or Swift provide.

Learn about the Kotlin/Native memory manager

How do I create a shared library?

Use the `-produce dynamic` compiler option or `binaries.sharedLib()` in your Gradle build file:

```
kotlin {
    iosArm64("mylib") {
        binaries.sharedLib()
    }
}
```

It produces a platform-specific shared object (`.so` on Linux, `.dylib` on macOS, and `.dll` on Windows targets) and a C language header, allowing the use of all public APIs available in your Kotlin/Native program from C/C++ code.

Complete the Kotlin/Native as a dynamic library tutorial

How do I create a static library or an object file?

Use the `-produce static` compiler option or `binaries.staticLib()` in your Gradle build file:

```
kotlin {
```

```
    iosArm64("mylib") {
        binaries.staticLib()
    }
}
```

It produces a platform-specific static object (`.a` library format) and a C language header, allowing you to use all the public APIs available in your Kotlin/Native program from C/C++ code.

How do I run Kotlin/Native behind a corporate proxy?

As Kotlin/Native needs to download a platform specific toolchain, you need to specify `-Dhttp.proxyHost=xxx -Dhttp.proxyPort=xxx` as the compiler's or `gradlew` arguments, or set it via the `JAVA_OPTS` environment variable.

How do I specify a custom Objective-C prefix/name for my Kotlin framework?

Use the `-module-name` compiler option or matching Gradle DSL statement.

> 【Kotlin】

```
kotlin {
    iosArm64("myapp") {
        binaries.framework {
            freeCompilerArgs += listOf("-module-name", "TheName")
        }
    }
}
```

> 【Groovy】

```
kotlin {
    iosArm64("myapp") {
        binaries.framework {
            freeCompilerArgs += ["-module-name", "TheName"]
        }
    }
}
```

How do I rename the iOS framework?

The default name is for an iOS framework is `<project name>.framework`. To set a custom name, use the `baseName` option. This will also set the module name.

```
kotlin {
    iosArm64("myapp") {
       binaries {
          framework {
              baseName = "TheName"
          }
       }
    }
}
```

How do I enable bitcode for my Kotlin framework?

Bitcode embedding was deprecated in Xcode 14 and removed in Xcode 15 for all Apple targets. The Kotlin/Native compiler does not support bitcode embedding since Kotlin 2.0.20.

If you're using earlier versions of Xcode but want to upgrade to Kotlin 2.0.20 or later versions, disable bitcode embedding in your Xcode projects.

How do I reference objects safely from different coroutines?

To safely access or update an object across multiple coroutines in Kotlin/Native, consider using concurrency-safe constructs, such as `@Volatile` and `AtomicReference`.

Use `@Volatile` to annotate a `var` property. This makes all reads and writes to the property's backing field atomic. In addition, writes become immediately visible to other threads. When another thread accesses this property, it observes not only the updated value but also the changes that happened before the update.

Alternatively, use AtomicReference, which supports atomic reads and updates. On Kotlin/Native, it wraps a volatile variable and performs atomic operations. Kotlin also provides a set of types for atomic operations tailored to specific data types. You can use `AtomicInt`, `AtomicLong`, `AtomicBoolean`, `AtomicArray`, as well as `AtomicIntArray` and `AtomicLongArray`.

For more information about access to shared mutable state, see the Coroutines documentation.

How can I compile my project with unreleased versions of Kotlin/Native?

First, please consider trying preview versions.

In case you need an even more recent development version, you can build Kotlin/Native from source code: clone Kotlin repository and follow these steps.

WebAssembly（Wasm）

- 在 IntelliJ IDEA 中入门 Kotlin/Wasm
- Kotlin/Wasm 与 WASI 入门
- 调试 Kotlin/Wasm 代码
- 与 JavaScript 的互操作
- 故障排查

Get started with Kotlin/Wasm and Compose Multiplatform

> Kotlin/Wasm is in Alpha. It may be changed at any time.
>
> Join the Kotlin/Wasm community.
>
> 

This tutorial demonstrates how to run a Compose Multiplatform app with Kotlin/Wasm in IntelliJ IDEA, and generate artifacts to publish as a site on GitHub pages.

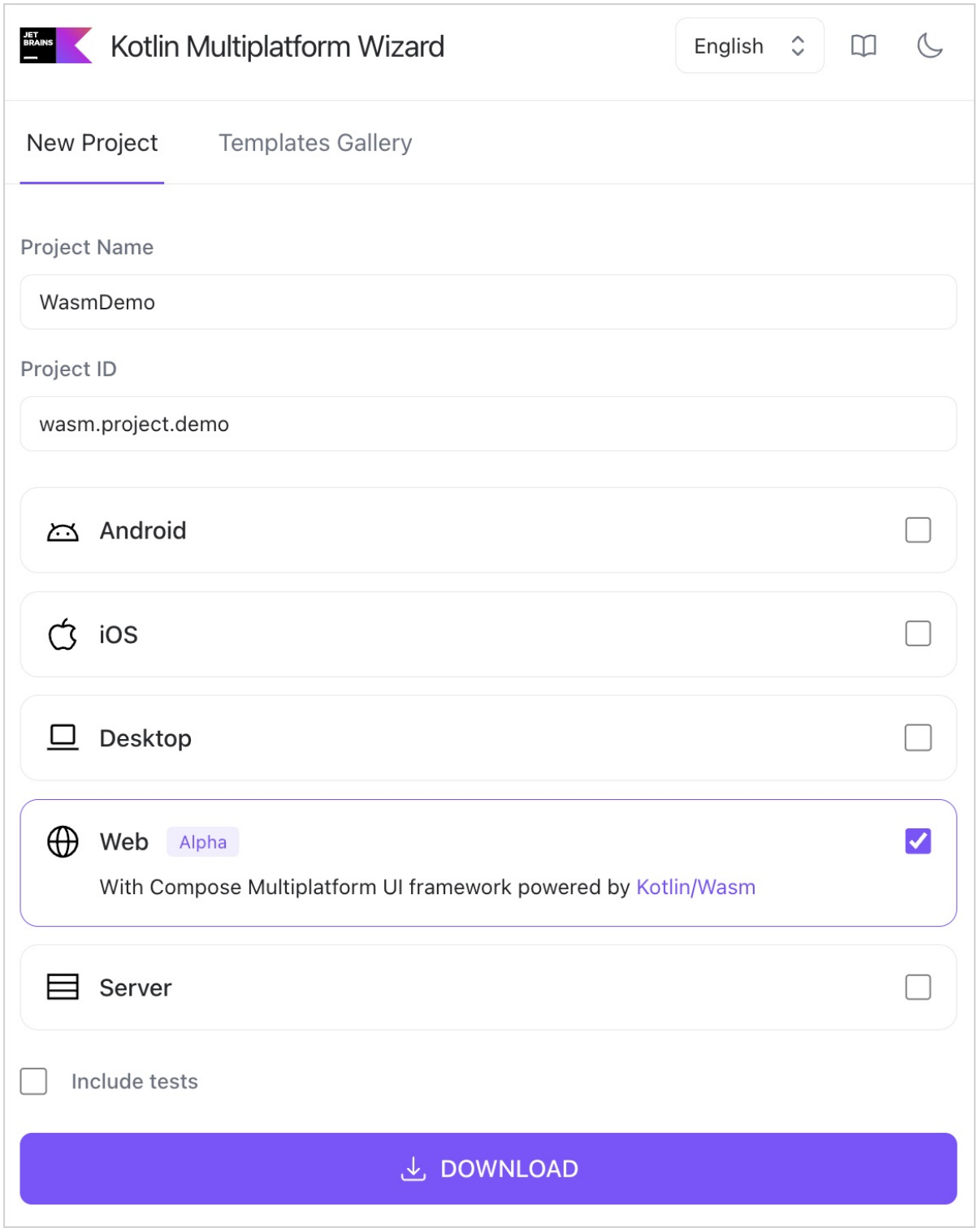
Before you start

Create a project using the Kotlin Multiplatform wizard:

1. Open the Kotlin Multiplatform wizard.
2. On the **New Project** tab, change the project name and ID to your preference. In this tutorial, we set the name to "WasmDemo" and the ID to "wasm.project.demo".

   > These are the name and ID of the project directory. You can also leave them as they are.
   >
   > 

3. Select the **Web** option. Make sure that no other options are selected.
4. Click the **Download** button and unpack the resulting archive.



Open the project in IntelliJ IDEA

1. Download and install the latest version of IntelliJ IDEA.
2. On the Welcome screen of IntelliJ IDEA, click **Open** or select **File | Open** in the menu bar.
3. Navigate to the unpacked "WasmDemo" folder and click **Open**.

Run the application

1. In IntelliJ IDEA, open the **Gradle** tool window by selecting **View** | **Tool Windows** | **Gradle**.

   You can find the Gradle tasks in the Gradle tool window once the project loads.

   > You need at least Java 11 as your Gradle JVM for the tasks to load successfully.
   >
   > 

2. In **wasmdemo** | **Tasks** | **kotlin browser**, select and run the **wasmJsBrowserDevelopmentRun** task.

   Alternatively, you can run the following command in the terminal from the `WasmDemo` root directory:

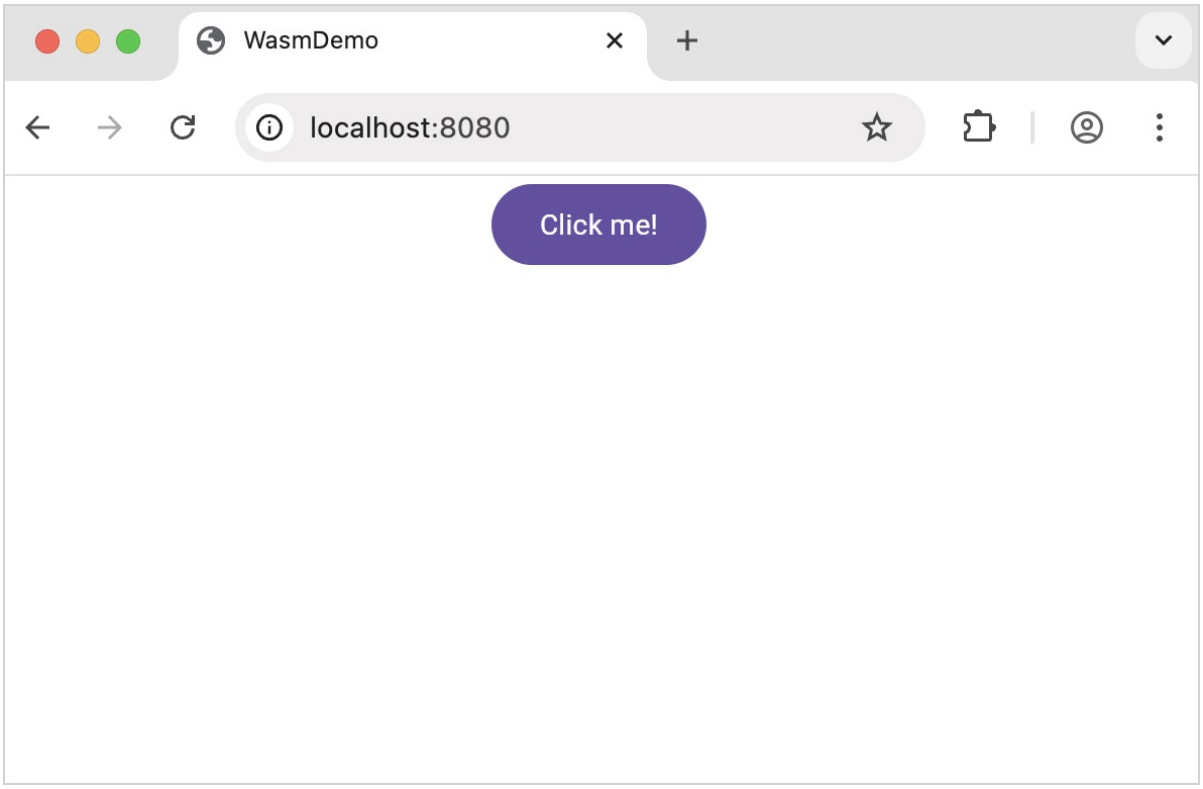
   ```
   ./gradlew wasmJsBrowserDevelopmentRun -t
   ```

3. Once the application starts, open the following URL in your browser:

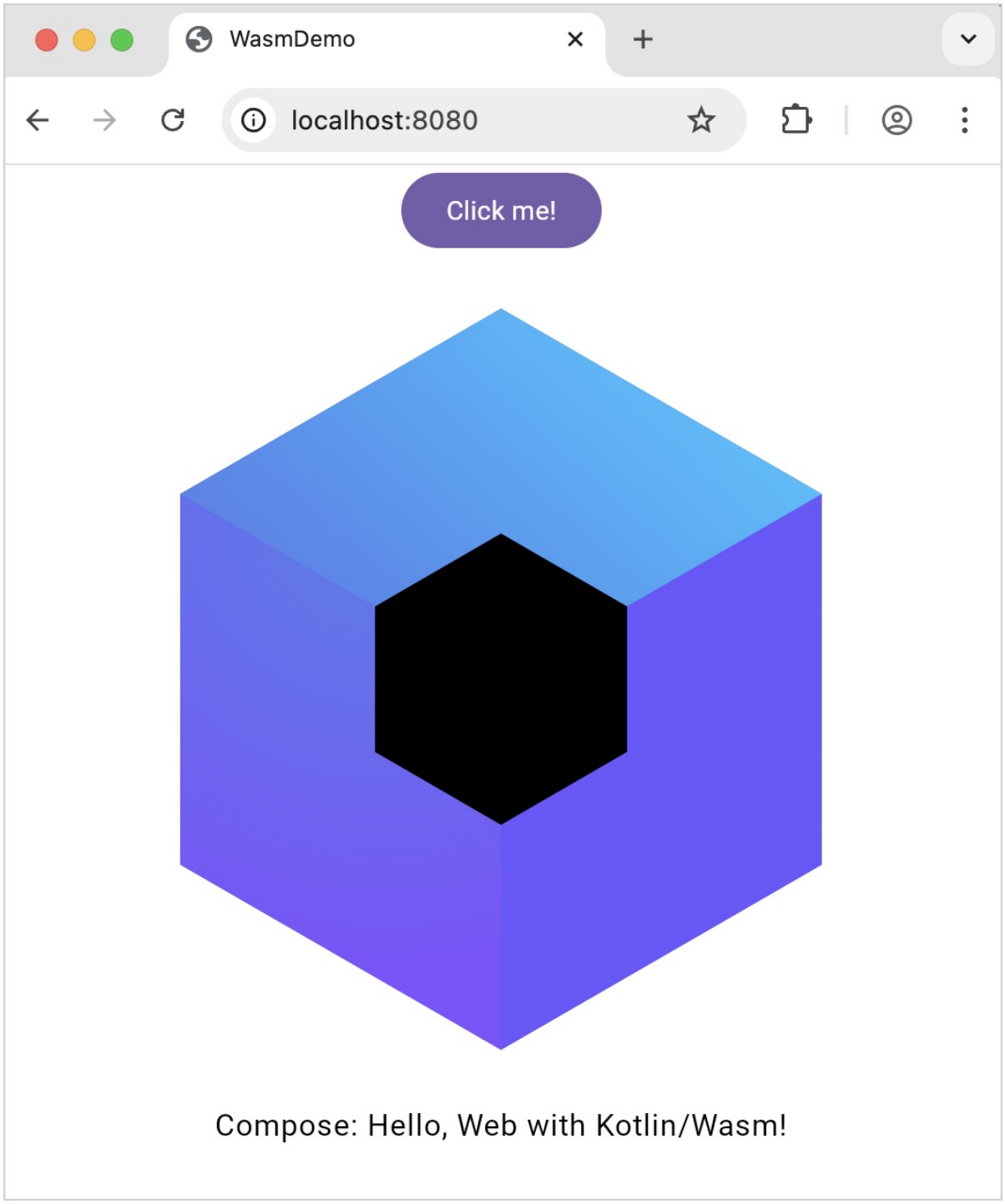
   ```
   http://localhost:8080/
   ```

   > The port number can vary because the 8080 port may be unavailable. You can find the actual port number printed in the Gradle build console.
   >
   > 

   You see a "Click me!" button. Click it:

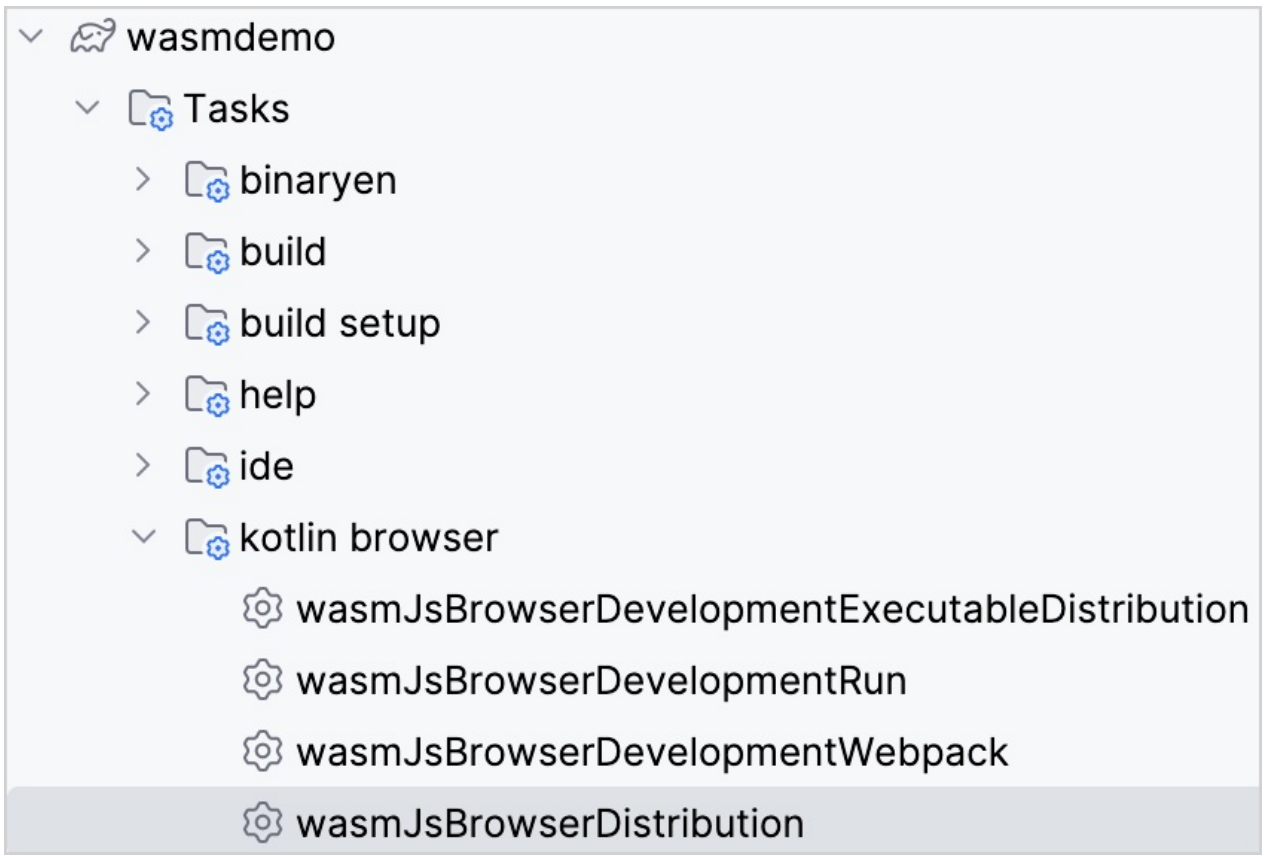


Now you see the Compose Multiplatform logo:



Generate artifacts

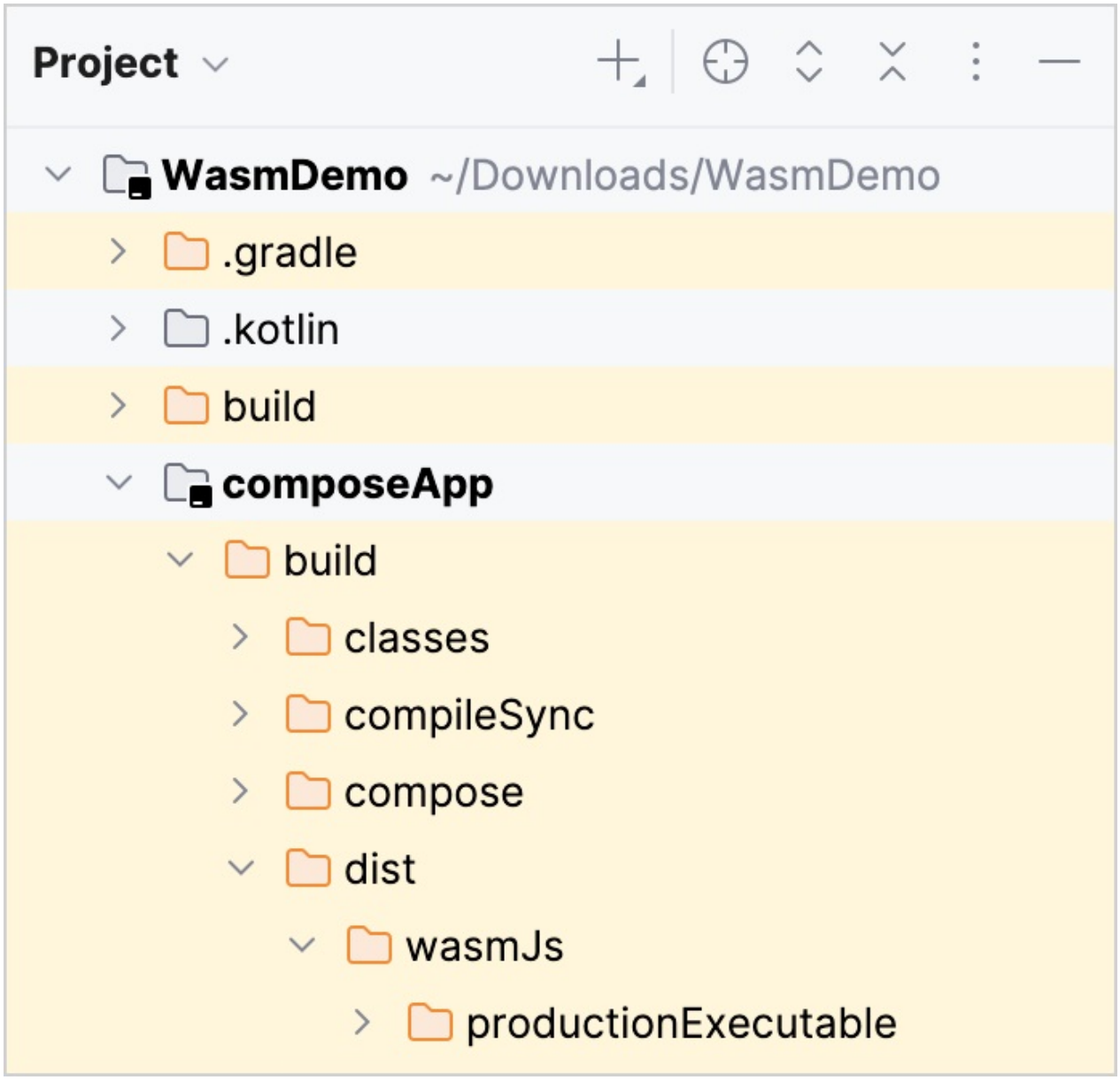
In **wasmdemo** | **Tasks** | **kotlin browser**, select and run the **wasmJsBrowserDistribution** task.



Alternatively, you can run the following command in the terminal from the `WasmDemo` root directory:

```
./gradlew wasmJsBrowserDistribution
```

Once the application task completes, you can find the generated artifacts in the `composeApp/build/dist/wasmJs/productionExecutable` directory:

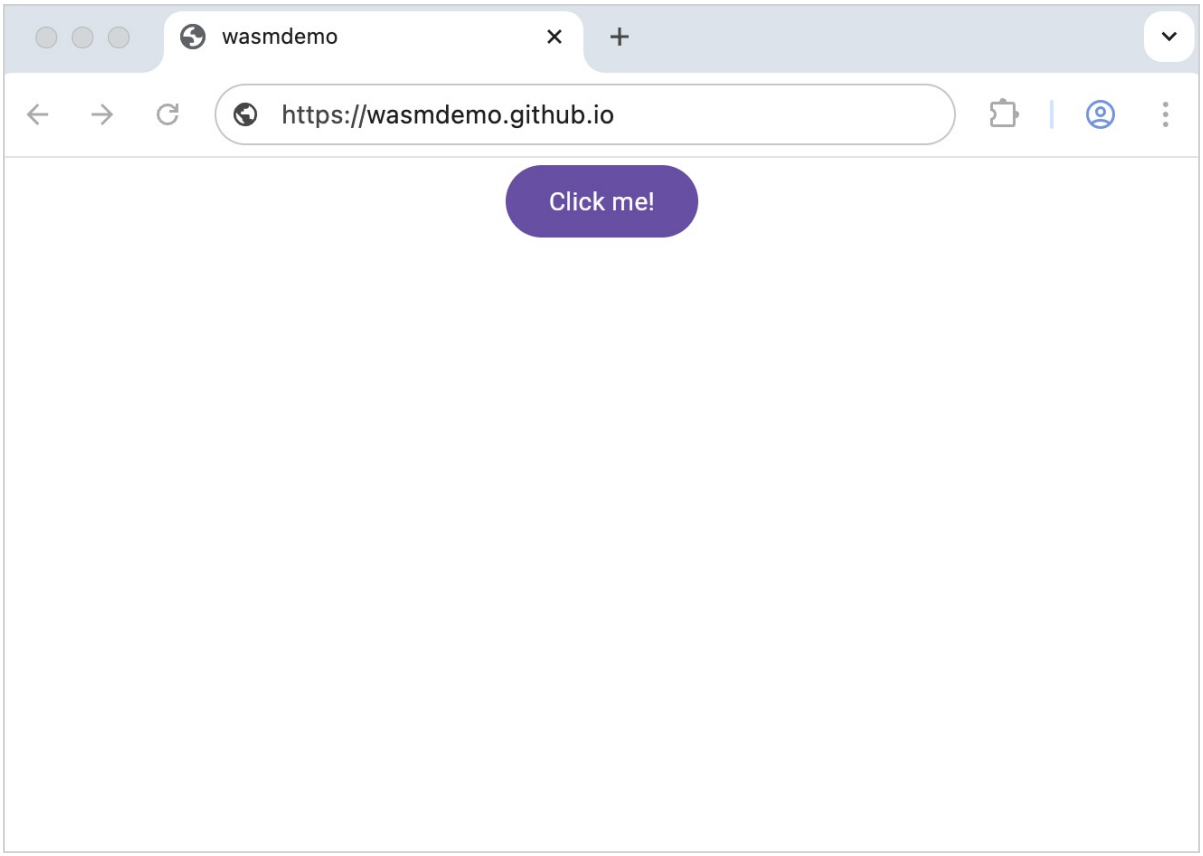


Publish on GitHub pages

1. Copy all the contents in your `productionExecutable` directory into the repository where you want to create a site.
2. Follow GitHub's instructions for creating your site.

   > It can take up to 10 minutes for changes to your site to publish after you push the changes to GitHub.

3. In a browser, navigate to your GitHub pages domain.



Congratulations! You have published your artifacts on GitHub pages.

下一步做什么？

Join the Kotlin/Wasm community in Kotlin Slack:

Join the Kotlin Slack channel →

Try more Kotlin/Wasm examples:

- Compose image viewer
- Jetsnack application
- Node.js example
- WASI example
- Compose example

Kotlin/Wasm 与 WASI 入门

> Kotlin/Wasm is in Alpha. It may be changed at any time.
>
> Join the Kotlin/Wasm community.
>
> 

This tutorial demonstrates how to run a simple Kotlin/Wasm application using the WebAssembly System Interface (WASI) in various WebAssembly virtual machines.

You can find examples of an application running on Node.js, Deno, and WasmEdge virtual machines. The output is a simple application that uses the standard WASI API.

Currently, Kotlin/Wasm supports WASI 0.1, also known as Preview 1. Support for WASI 0.2 is planned for future releases.

> The Kotlin/Wasm toolchain provides Node.js tasks (`wasmWasiNode*`) out of the box. Other task variants in the project, such as those utilizing Deno or WasmEdge, are included as custom tasks.
>
> 

Before you start

1. Download and install the latest version of IntelliJ IDEA.
2. Clone the Kotlin/Wasm WASI template repository by selecting **File | New | Project from Version Control** in IntelliJ IDEA.

   You can also clone it from the command line:

   ```
   git clone git@github.com:Kotlin/kotlin-wasm-wasi-template.git
   ```

Run the application

1. Open the **Gradle** tool window by selecting **View** | **Tool Windows** | **Gradle**.

   In the **Gradle** tool window, you can find the Gradle tasks under **kotlin-wasm-wasi-example** once the project loads.

   > You need at least Java 11 as your Gradle JVM for the tasks to load successfully.

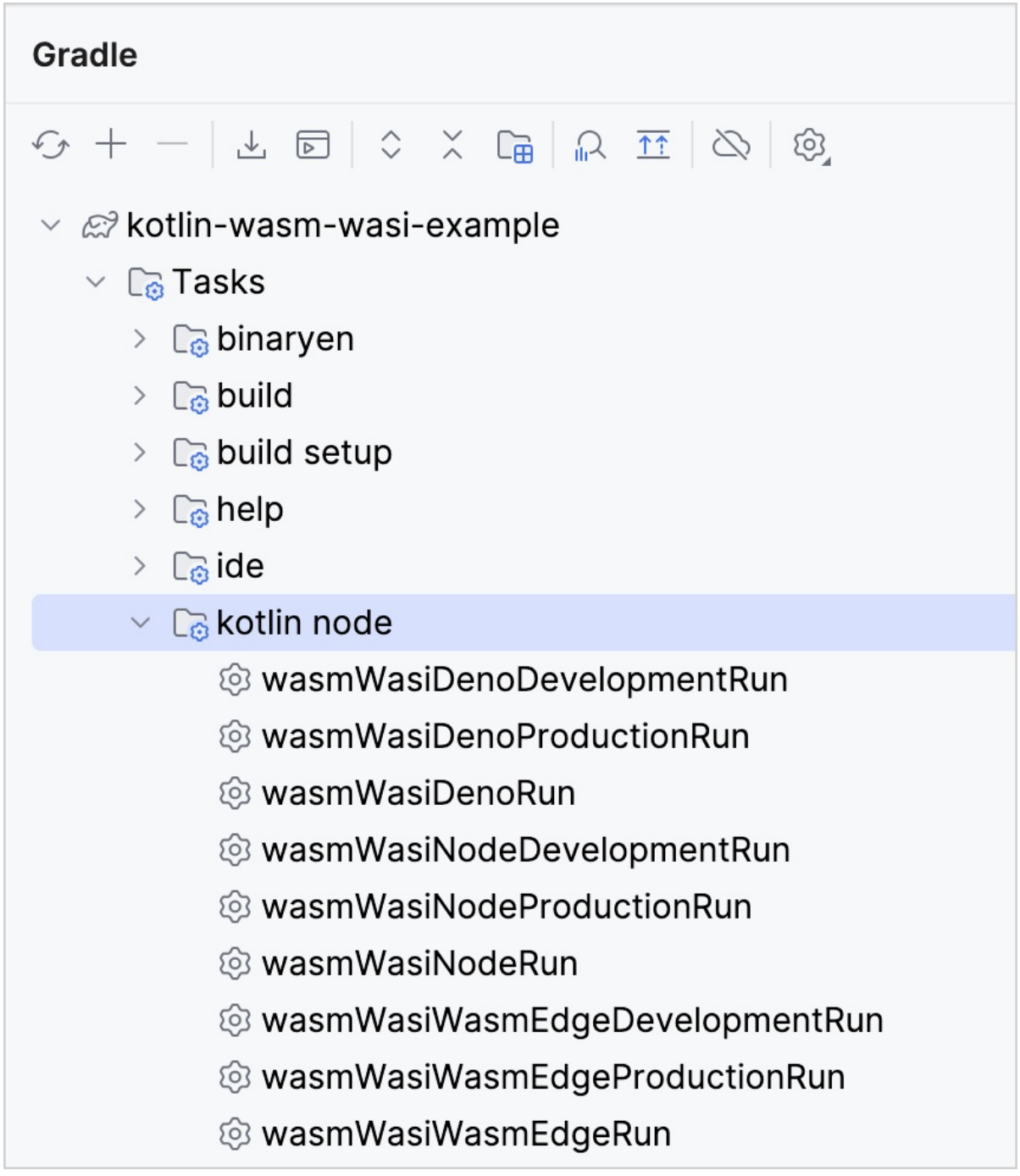


2. From **kotlin-wasm-wasi-example** | **Tasks** | **kotlin node**, select and run one of the following Gradle tasks:

   - **wasmWasiNodeRun** to run the application in Node.js.
   - **wasmWasiDenoRun** to run the application in Deno.
   - **wasmWasiWasmEdgeRun** to run the application in WasmEdge.

   > When using Deno on a Windows platform, ensure `deno.exe` is installed. For more information, see Deno's installation documentation.



Alternatively, run one of the following commands in the terminal from the `kotlin-wasm-wasi-template` root directory:

- To run the application in Node.js:

  ```
  ./gradlew wasmWasiNodeRun
  ```

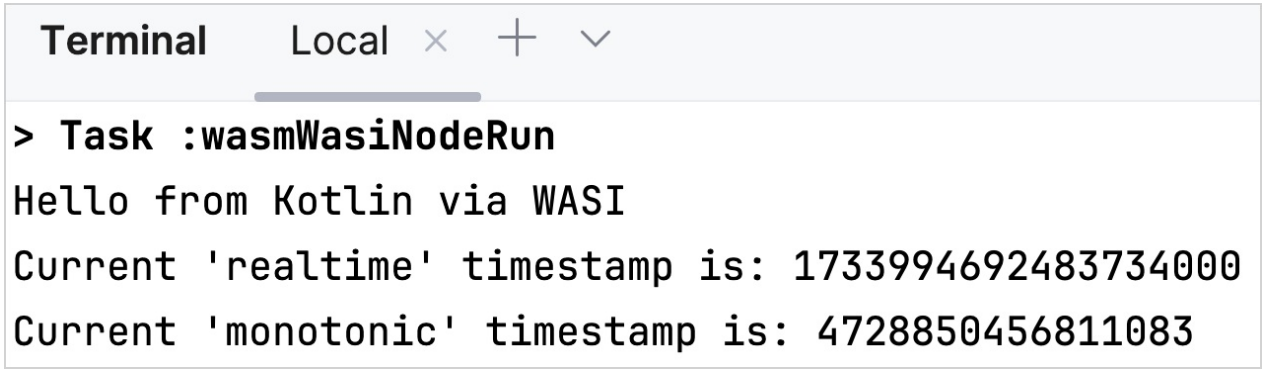
- To run the application in Deno:

  ```
  ./gradlew wasmWasiDenoRun
  ```

- To run the application in WasmEdge:

  ```
  ./gradlew wasmWasiWasmEdgeRun
  ```

The terminal displays a message when your application is built successfully:

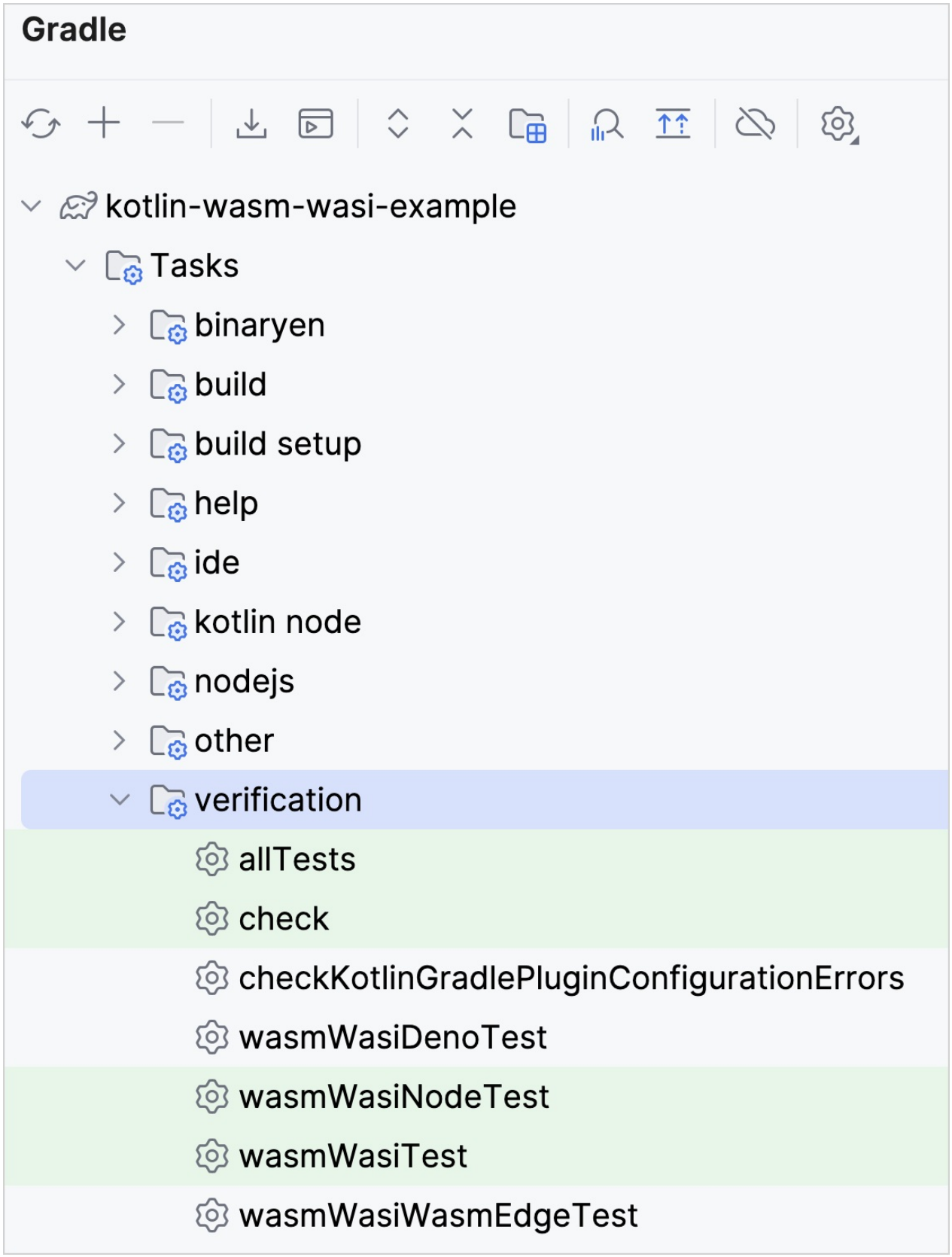


Test the application

You can also test that the Kotlin/Wasm application works correctly across various virtual machines.

In the Gradle tool window, run one of the following Gradle tasks from **kotlin-wasm-wasi-example** | **Tasks** | **verification**:

- **wasmWasiNodeTest** to test the application in Node.js.
- **wasmWasiDenoTest** to test the application in Deno.
- **wasmWasiWasmEdgeTest** to test the application in WasmEdge.



Alternatively, run one of the following commands in the terminal from the `kotlin-wasm-wasi-template` root directory:

- To test the application in Node.js:

```
./gradlew wasmWasiNodeTest
```

- To test the application in Deno:

```
./gradlew wasmWasiDenoTest
```

- To test the application in WasmEdge:

```
./gradlew wasmWasiWasmEdgeTest
```

The terminal displays the test results:

```
Terminal   Local
> Task :wasmWasiDenoTest
##teamcity[testSuiteStarted name='' flowId='wasmTcAdapter639728143']
##teamcity[testSuiteStarted name='WasiTest' flowId='wasmTcAdapter639728143']
##teamcity[testStarted name='mainTest' flowId='wasmTcAdapter639728143']
##teamcity[testFinished name='mainTest' flowId='wasmTcAdapter639728143']
##teamcity[testSuiteFinished name='WasiTest' flowId='wasmTcAdapter639728143']
##teamcity[testSuiteFinished name='' flowId='wasmTcAdapter639728143']

BUILD SUCCESSFUL in 482ms
8 actionable tasks: 2 executed, 6 up-to-date
```

下一步做什么？

Join the Kotlin/Wasm community in Kotlin Slack:

Join the Kotlin Slack channel →

Try more Kotlin/Wasm examples:

- Compose image viewer
- Jetsnack application
- Node.js example
- Compose example

调试 Kotlin/Wasm 代码

> Kotlin/Wasm is Alpha. It may be changed at any time.
>
> 

This tutorial demonstrates how to use your browser to debug your Compose Multiplatform application built with Kotlin/Wasm.

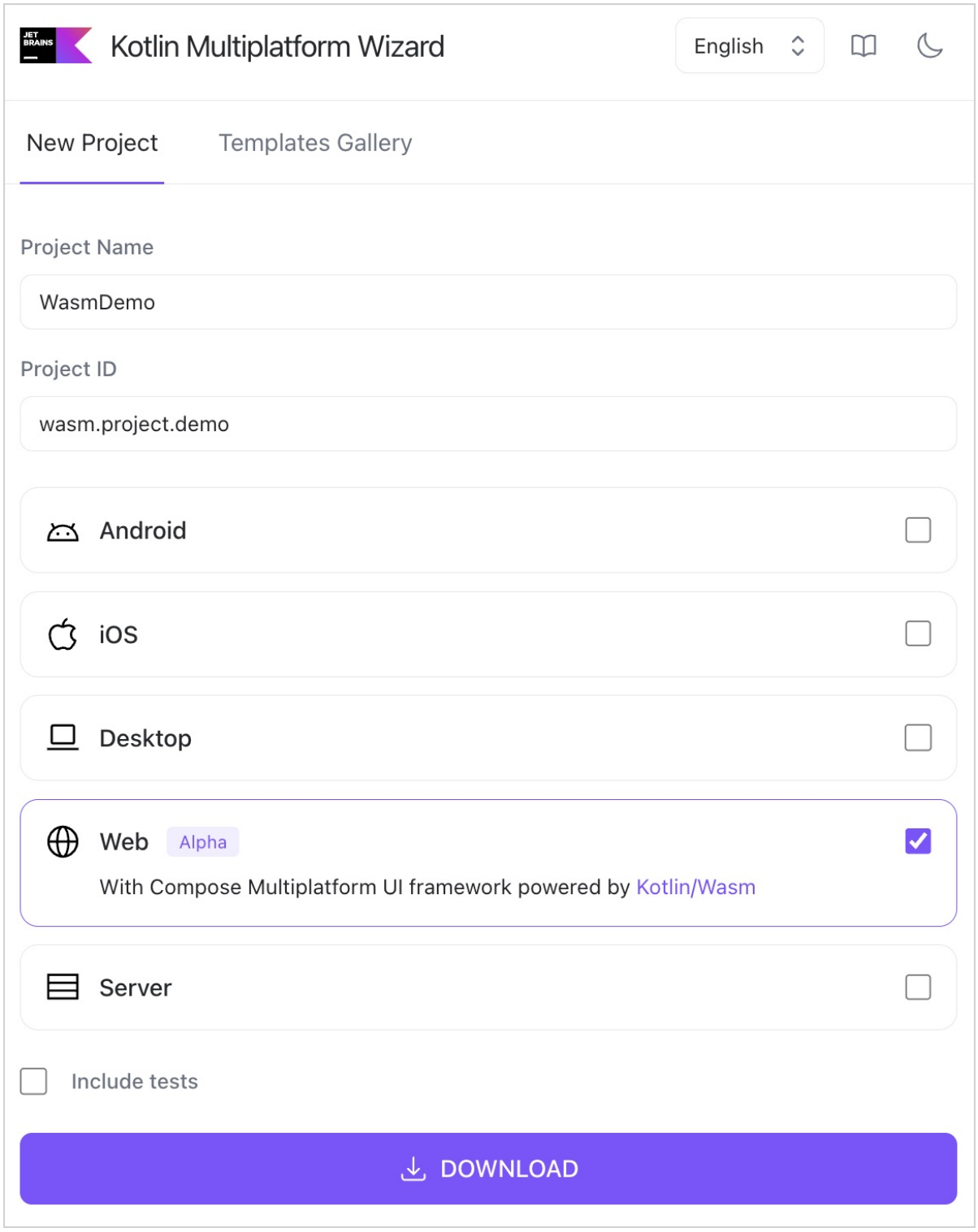
Before you start

Create a project using the Kotlin Multiplatform wizard:

1. Open the Kotlin Multiplatform wizard.
2. On the **New Project** tab, change the project name and ID to your preference. In this tutorial, we set the name to "WasmDemo" and the ID to "wasm.project.demo".

   > These are the name and ID of the project directory. You can also leave them as they are.
   >
   > 

3. Select the **Web** option. Make sure that no other options are selected.
4. Click the **Download** button and unpack the resulting archive.



Open the project in IntelliJ IDEA

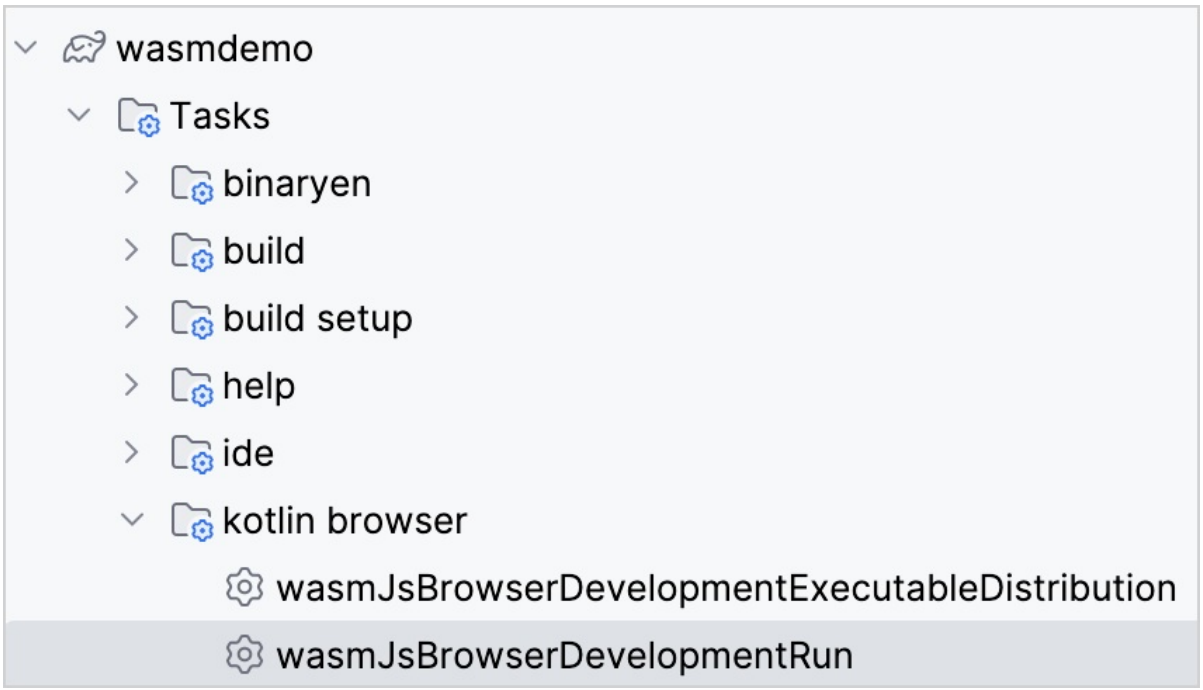
1. Download and install the latest version of IntelliJ IDEA.
2. On the Welcome screen of IntelliJ IDEA, click **Open** or select **File | Open** in the menu bar.
3. Navigate to the unpacked "WasmDemo" folder and click **Open**.

Run the application

1. In IntelliJ IDEA, open the **Gradle** tool window by selecting **View** | **Tool Windows** | **Gradle**.

   > You need at least Java 11 as your Gradle JVM for the tasks to load successfully.
   >
   > 

2. In **composeApp** | **Tasks** | **kotlin browser**, select and run the **wasmJsBrowserDevelopmentRun** task.

   Alternatively, you can run the following command in the terminal from the `WasmDemo` root directory:

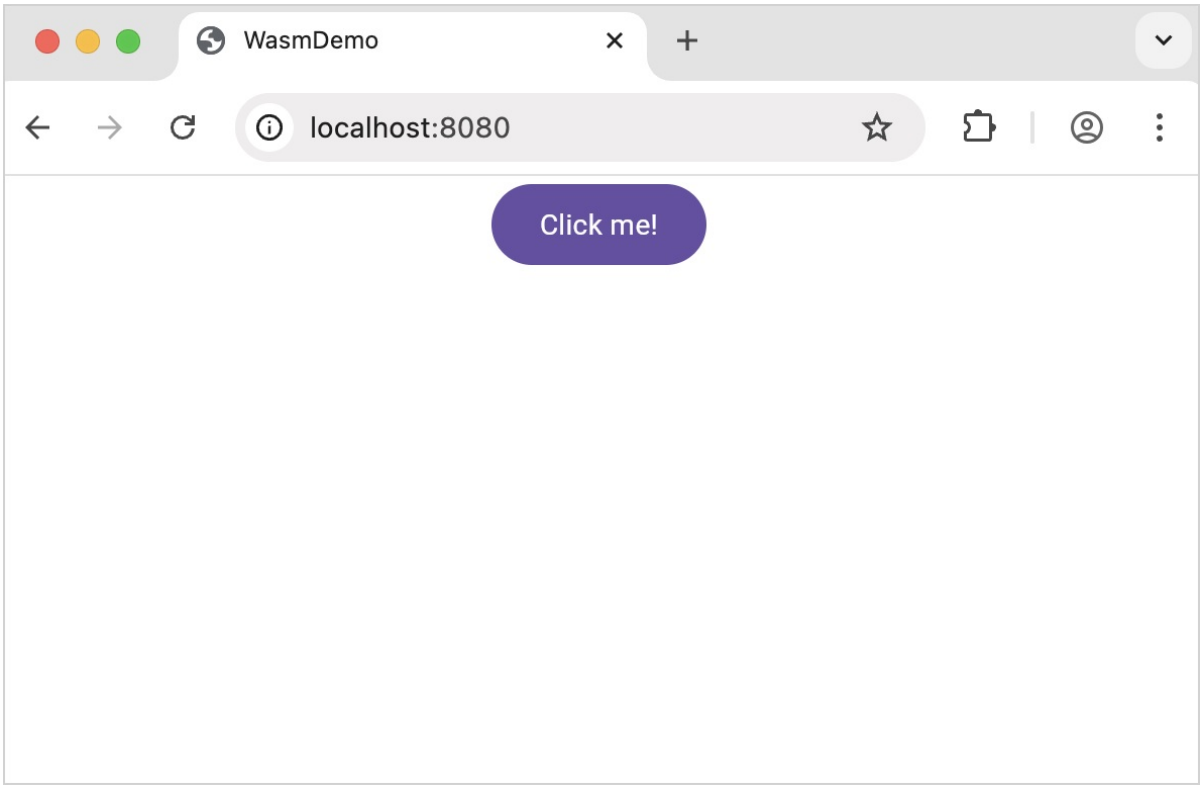
   ```
   ./gradlew wasmJsBrowserDevelopmentRun
   ```

3. Once the application starts, open the following URL in your browser:

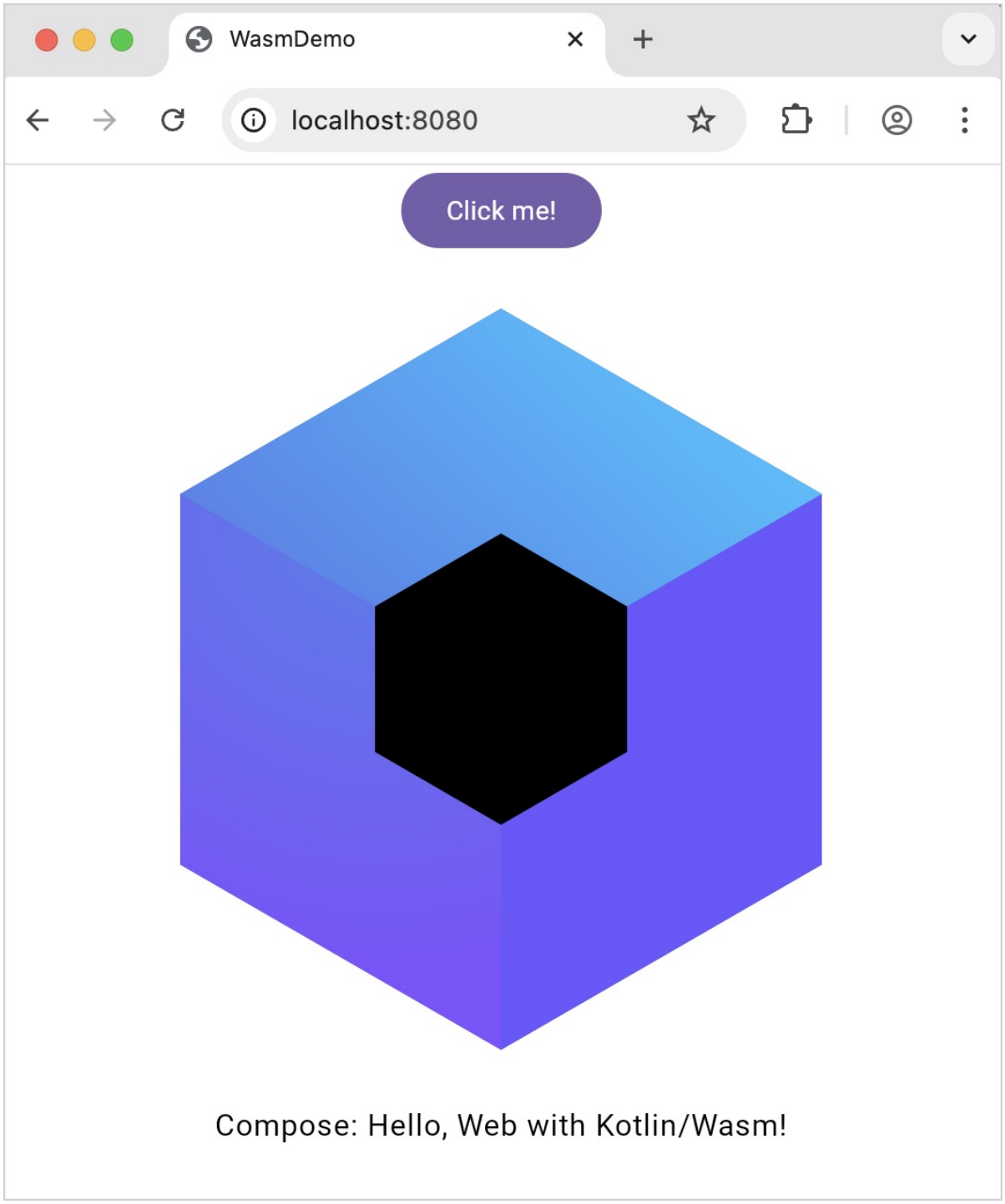
   ```
   http://localhost:8080/
   ```

   > The port number can vary because the 8080 port may be unavailable. You can find the actual port number printed in the Gradle build console.
   >
   > 

   You see a "Click me!" button. Click it:



Now you see the Compose Multiplatform logo:



Debug in your browser

> Currently, debugging is only available in your browser. In the future, you will be able to debug your code in IntelliJ IDEA.

You can debug this Compose Multiplatform application in your browser out of the box, without additional configurations.

However, for other projects, you may need to configure additional settings in your Gradle build file. For more information about how to configure your browser for debugging, expand the next section.

Configure your browser for debugging

Enable access to project's sources

By default, browsers can't access some of the project's sources necessary for debugging. To provide access, you can configure the Webpack DevServer to serve these sources. In the `ComposeApp` directory, add the following code snippets to your `build.gradle.kts` file.

Add this import as a top-level declaration:

```
import org.jetbrains.kotlin.gradle.targets.js.webpack.KotlinWebpackConfig
```

Add this code snippet inside the `commonWebpackConfig{}` block, located in the `wasmJs{}` target DSL and `browser{}` platform DSL within `kotlin{}`:

```
devServer = (devServer ?: KotlinWebpackConfig.DevServer()).apply {
    static = (static ?: mutableListOf()).apply {
        // Serve sources to debug inside browser
        add(project.rootDir.path)
        add(project.projectDir.path)
    }
}
```

The resulting code block looks like this:

```
kotlin {
    @OptIn(ExperimentalWasmDsl::class)
    wasmJs {
        moduleName = "composeApp"
        browser {
            commonWebpackConfig {
                outputFileName = "composeApp.js"
                devServer = (devServer ?: KotlinWebpackConfig.DevServer()).apply {
                    static = (static ?: mutableListOf()).apply {
                        // Serve sources to debug inside browser
                        add(project.rootDir.path)
                        add(project.projectDir.path)
                    }
                }
            }
        }
    }
}
```

> Currently, you can't debug library sources. We will support this in the future.
>
> 

Use custom formatters

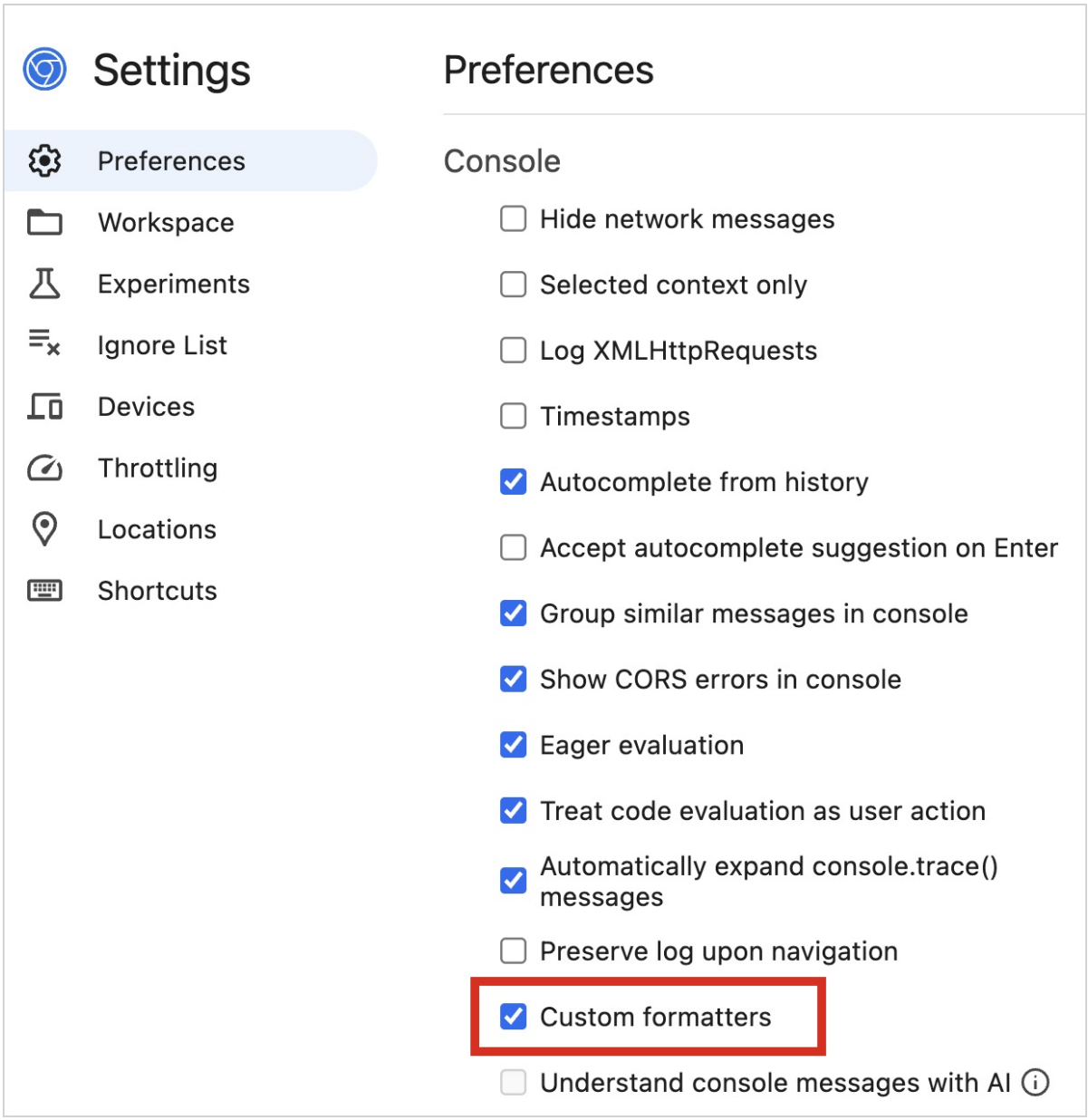
Custom formatters help display and locate variable values in a more user-friendly and comprehensible manner when debugging Kotlin/Wasm code.

Custom formatters are enabled by default in development builds, so you don't need additional Gradle configurations.

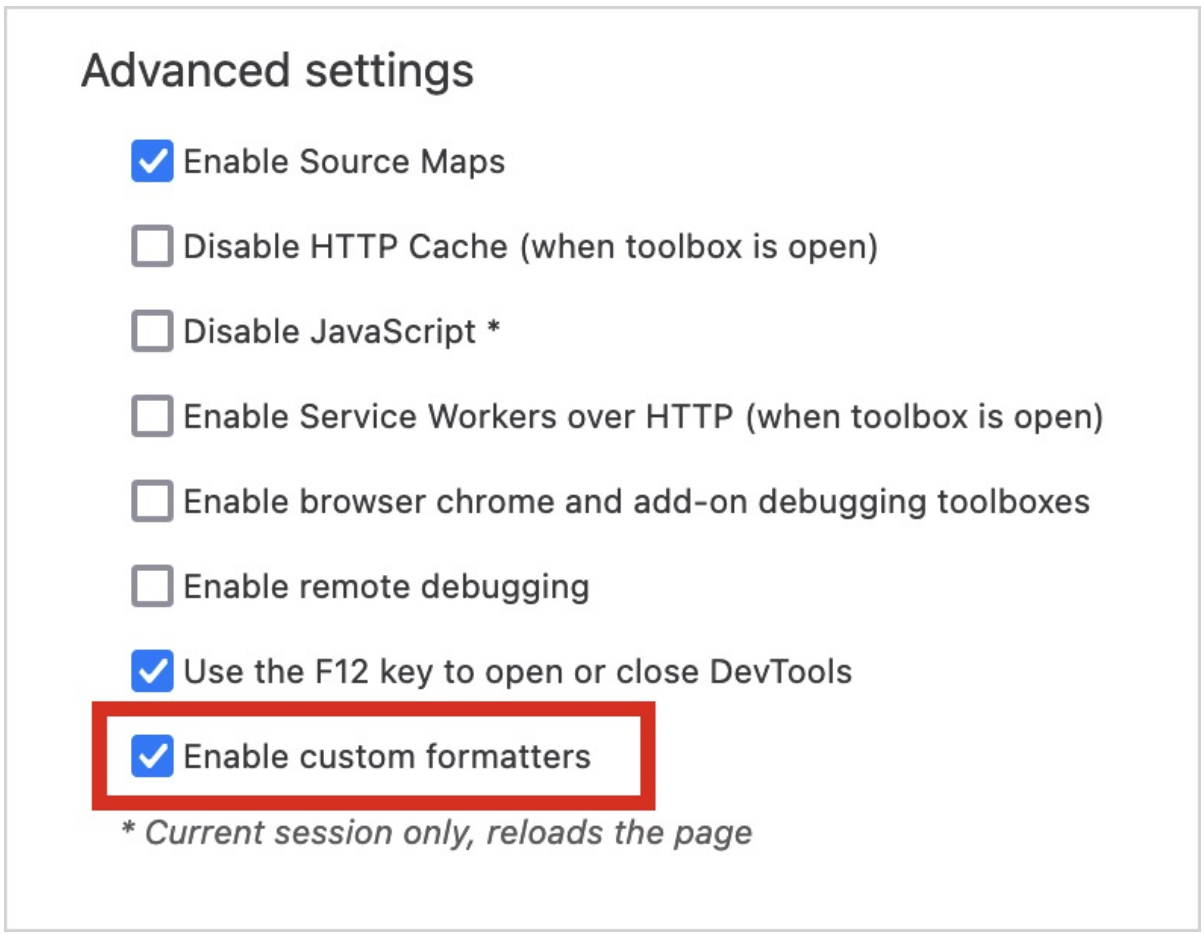
This feature is supported in Firefox and Chromium-based browsers as it uses the custom formatters API.

To use this feature, ensure that custom formatters are enabled in your browser's developer tools:

- In Chrome DevTools, find the custom formatters checkbox in **Settings | Preferences | Console**:



- In Firefox DevTools, find the custom formatters checkbox in **Settings | Advanced settings**:



Custom formatters work for Kotlin/Wasm development builds. If you have specific requirements for production builds, you need to adjust your Gradle configuration accordingly. Add the following compiler option to the `wasmJs {}` block:

```
// build.gradle.kts
kotlin {
    wasmJs {
        // ...

        compilerOptions {
            freeCompilerArgs.add("-Xwasm-debugger-custom-formatters")
        }
    }
}
```

After enabling custom formatters, you can continue with the debugging tutorial.

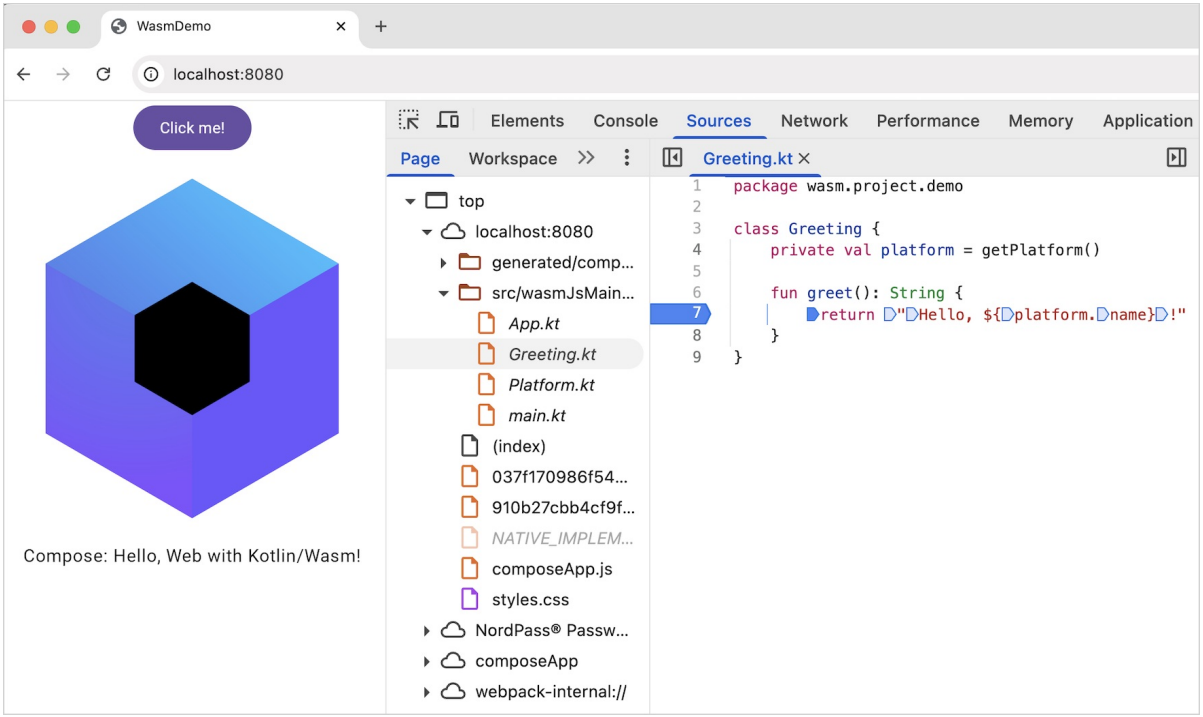
Debug your Kotlin/Wasm application

> This tutorial uses the Chrome browser, but you should be able to follow these steps with other browsers. For more information, see Browser versions.

1. In the browser window of the application, right-click and select the **Inspect** action to access developer tools. Alternatively, you can use the **F12** shortcut or select **View** | **Developer** | **Developer Tools**.

2. Switch to the **Sources** tab and select the Kotlin file to debug. In this tutorial, we'll work with the `Greeting.kt` file.

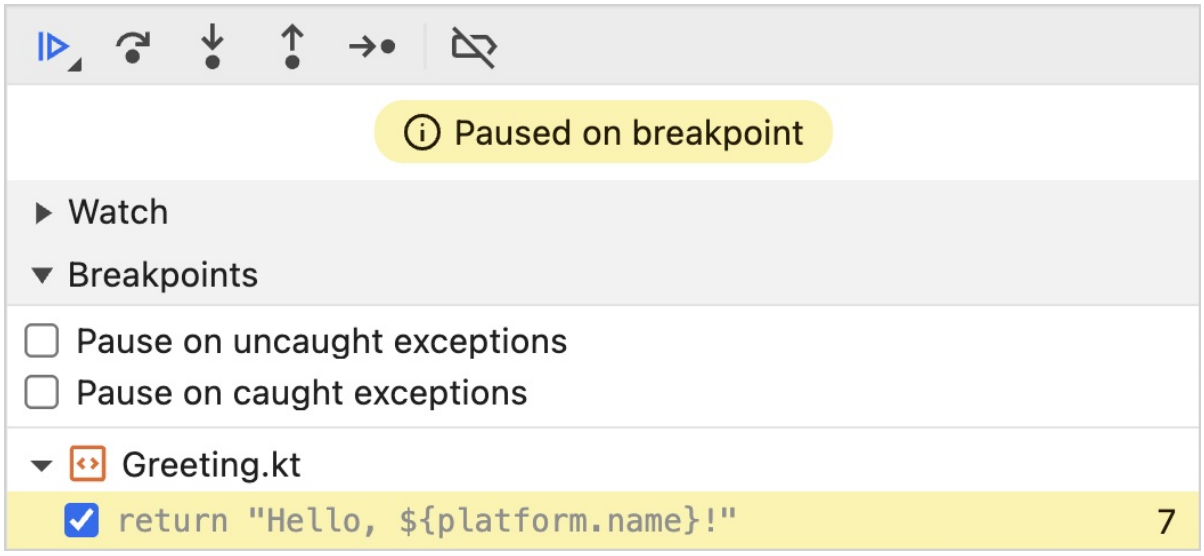
3. Click on the line numbers to set breakpoints on the code that you want to inspect. Only the lines with darker numbers can have breakpoints.



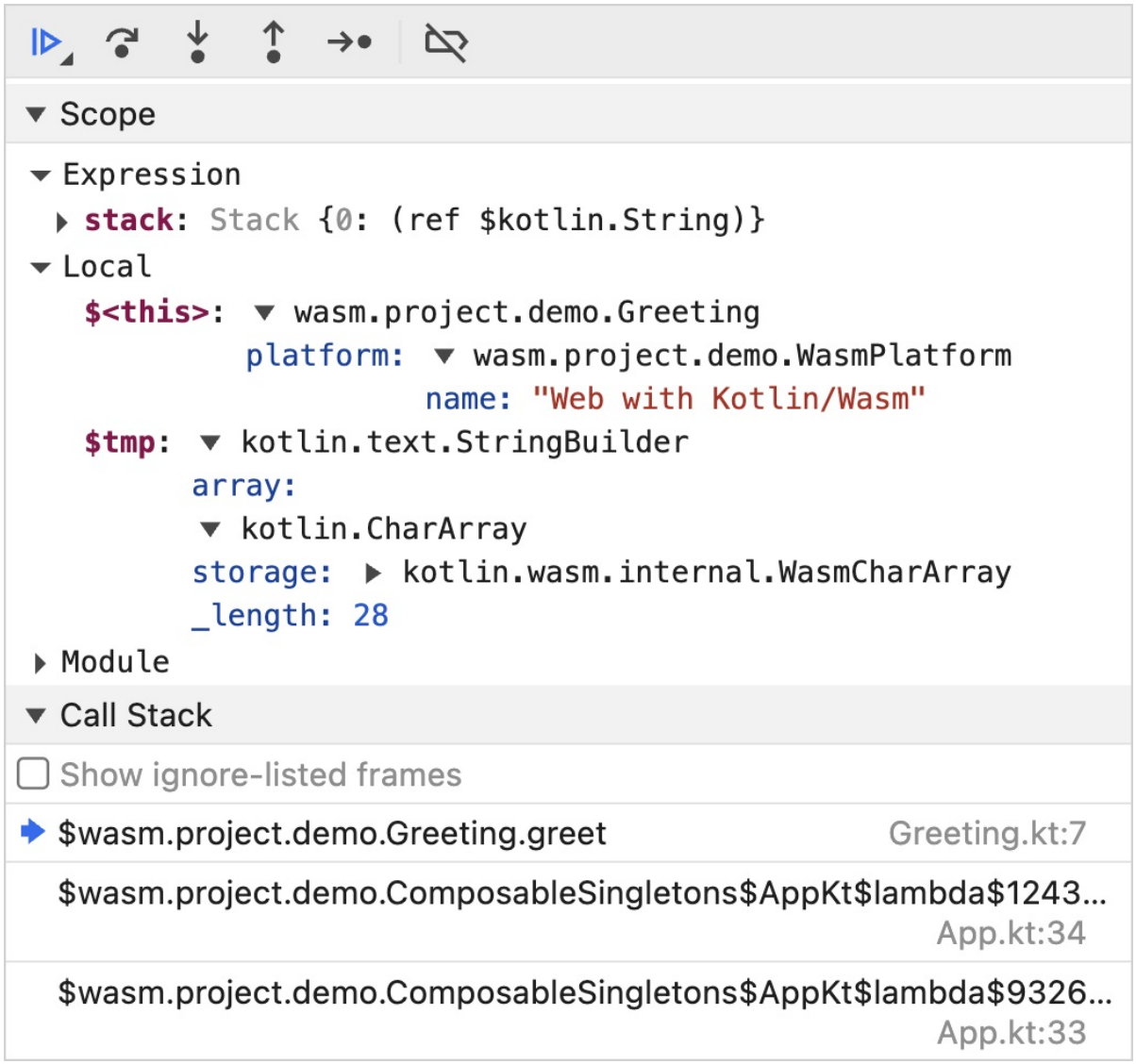
4. Click on the **Click me!** button to interact with the application. This action triggers the execution of the code, and the debugger pauses when the execution reaches a breakpoint.

5. In the debugging pane, use the debugging control buttons to inspect variables and code execution at the breakpoints:

    - Step into to investigate a function more deeply.
    - Step over to execute the current line and pause on the next line.
    - Step out to execute the code until it exits the current function.



6. Check the **Call stack** and **Scope** panes to trace the sequence of function calls and pinpoint the location of any errors.



For an improved visualization of the variable values, see *Use custom formatters* within the Configure your browser for debugging section.

7. Make changes to your code and run the application again to verify that everything works as expected.
8. Click on the line numbers with breakpoints to remove the breakpoints.

Leave feedback

We would appreciate any feedback you may have on your debugging experience!

- Slack: get a Slack invite and provide your feedback directly to the developers in our #webassembly channel.
- Provide your feedback in YouTrack.

下一步做什么？

- See Kotlin/Wasm debugging in action in this YouTube video.
- Try the Kotlin/Wasm examples from our `kotlin-wasm-examples` repository:
  - Compose image viewer
  - Jetsnack application
  - Node.js example
  - WASI example
  - Compose example

与 JavaScript 的互操作

Kotlin/Wasm allows you to use both JavaScript code in Kotlin and Kotlin code in JavaScript.

As with Kotlin/JS, the Kotlin/Wasm compiler also has interoperability with JavaScript. If you are familiar with Kotlin/JS interoperability, you can notice that Kotlin/Wasm interoperability is similar. However, there are key differences to consider.

> Kotlin/Wasm is Alpha. It may be changed at any time. Use it in scenarios before production. We would appreciate your feedback in YouTrack.
>
> 

Use JavaScript code in Kotlin

Learn how to use JavaScript code in Kotlin by using `external` declarations, functions with JavaScript code snippets, and the `@JsModule` annotation.

External declarations

External JavaScript code is not visible in Kotlin by default. To use JavaScript code in Kotlin, you can describe its API with `external` declarations.

JavaScript functions

Consider this JavaScript function:

```
function greet (name) {
    console.log("Hello, " + name + "!");
}
```

You can declare it in Kotlin as an `external` function:

```
external fun greet(name: String)
```

External functions don't have bodies, and you can call it as a regular Kotlin function:

```
fun main() {
    greet("Alice")
}
```

JavaScript properties

Consider this global JavaScript variable:

```
let globalCounter = 0;
```

You can declare it in Kotlin using external `var` or `val` properties:

```
external var globalCounter: Int
```

These properties are initialized externally. The properties can't have `= value` initializers in Kotlin code.

JavaScript classes

Consider this JavaScript class:

```
class Rectangle {
    constructor (height, width) {
        this.height = height;
        this.width = width;
    }

    area () {
        return this.height * this.width;
    }
}
```

You can use it in Kotlin as an external class:

```
external class Rectangle(height: Double, width: Double) : JsAny {
    val height: Double
    val width: Double
    fun area(): Double
}
```

All declarations inside the `external` class are implicitly considered external.

External interfaces

You can describe the shape of a JavaScript object in Kotlin. Consider this JavaScript function and what it returns:

```
function createUser (name, age) {
    return { name: name, age: age };
}
```

See how its shape can be described in Kotlin with an `external interface User` type:

```
external interface User : JsAny {
    val name: String
    val age: Int
}

external fun createUser(name: String, age: Int): User
```

External interfaces don't have runtime type information and are a compile-time-only concept. Therefore, external interfaces have some restrictions compared to regular interfaces:

- You can't use them on the right-hand side of `is` checks.
- You can't use them in class literal expressions (such as `User::class`).
- You can't pass them as reified type arguments.
- Casting with `as` to external interfaces always succeed.

External objects

Consider these JavaScript variables holding an object:

```
let Counter = {
    value: 0,
    step: 1,
    increment () {
        this.value += this.step;
    }
};
```

You can use them in Kotlin as an external object:

```
external object Counter : JsAny {
    fun increment()
    val value: Int
    var step: Int
}
```

External type hierarchy

Similar to regular classes and interfaces, you can declare external declarations to extend other external classes and implement external interfaces. However, you can't mix external and non-external declarations in the same type hierarchy.

Kotlin functions with JavaScript code

You can add a JavaScript snippet to Kotlin/Wasm code by defining a function with `= js("code")` body:

```
fun getCurrentURL(): String =
    js("window.location.href")
```

If you want to run a block of JavaScript statements, surround your code inside the string with curly brackets `{}` :

```
fun setLocalSettings(value: String): Unit = js(
    """{
        localStorage.setItem('settings', value);
}"""
)
```

If you want to return an object, surround the curly brackets `{}` with parentheses `()` :

```
fun createJsUser(name: String, age: Int): JsAny =
    js("({ name: name, age: age })")
```

Kotlin/Wasm treats calls to the `js()` function in a special way, and the implementation has some restrictions:

- A `js()` function call must be provided with a string literal argument.
- A `js()` function call must be the only expression in the function body.
- The `js()` function is only allowed to be called from package-level functions.
- The function return type must be provided explicitly.
- Types are restricted, similar to `external fun` .

The Kotlin compiler puts the code string into a function in the generated JavaScript file and imports it into WebAssembly format. The Kotlin compiler doesn't verify these JavaScript snippets. If there are JavaScript syntax errors, they are reported when you run your JavaScript code.

> The `@JsFun` annotation has similar functionality and will likely be deprecated.
>
> 

JavaScript modules

By default, external declarations correspond to the JavaScript global scope. If you annotate a Kotlin file with the `@JsModule` annotation, then all external declarations within it are imported from the specified module.

Consider this JavaScript code sample:

```
// users.mjs
export let maxUsers = 10;

export class User {
    constructor (username) {
        this.username = username;
    }
}
```

Use this JavaScript code in Kotlin with the `@JsModule` annotation:

```
// Kotlin
@file:JsModule("./users.mjs")

external val maxUsers: Int

external class User : JsAny {
    constructor(username: String)

    val username: String
}
```

Array interoperability

You can copy JavaScript's `JsArray<T>` into Kotlin's native `Array` or `List` types; likewise, you can copy these Kotlin types to `JsArray<T>`.

To convert `JsArray<T>` to `Array<T>` or the other way around, use one of the available adapter functions.

Here's an example of conversion between generic types:

```
val list: List<JsString> =
    listOf("Kotlin", "Wasm").map { it.toJsString() }

// Uses .toJsArray() to convert List or Array to JsArray
val jsArray: JsArray<JsString> = list.toJsArray()

// Uses .toArray() and .toList() to convert it back to Kotlin types
```

```
val kotlinArray: Array<JsString> = jsArray.toArray()
val kotlinList: List<JsString> = jsArray.toList()
```

Similar adapter functions are available for converting typed arrays to their Kotlin equivalents (for example, `IntArray` and `Int32Array`). For detailed information and implementation, see the `kotlinx-browser` repository.

Here's an example of conversion between typed arrays:

```
import org.khronos.webgl.*

    // ...

    val intArray: IntArray = intArrayOf(1, 2, 3)

    // Uses .toInt32Array() to convert Kotlin IntArray to JavaScript Int32Array
    val jsInt32Array: Int32Array = intArray.toInt32Array()

    // Uses toIntArray() to convert JavaScript Int32Array back to Kotlin IntArray
    val kotlnIntArray: IntArray = jsInt32Array.toIntArray()
```

Use Kotlin code in JavaScript

Learn how to use your Kotlin code in JavaScript by using the `@JsExport` annotation.

Functions with the @JsExport annotation

To make a Kotlin/Wasm function available to JavaScript code, use the `@JsExport` annotation:

```
// Kotlin/Wasm

@JsExport
fun addOne(x: Int): Int = x + 1
```

Kotlin/Wasm functions marked with the `@JsExport` annotation are visible as properties on a `default` export of the generated `.mjs` module. You can then use this function in JavaScript:

```
// JavaScript

import exports from "./module.mjs"

exports.addOne(10)
```

The Kotlin/Wasm compiler is capable of generating TypeScript definitions from any `@JsExport` declarations in your Kotlin code. These definitions can be used by IDEs and JavaScript tools to provide code autocompletion, help with type-checks, and make it easier to consume Kotlin code from JavaScript and TypeScript.

The Kotlin/Wasm compiler collects any top-level functions marked with the `@JsExport` annotation and automatically generates TypeScript definitions in a `.d.ts` file.

To generate TypeScript definitions, in your `build.gradle.kts` file in the `wasmJs{}` block, add the `generateTypeScriptDefinitions()` function:

```
kotlin {
    wasmJs {
        binaries.executable()
        browser {
        }
        generateTypeScriptDefinitions()
    }
}
```

> Generating TypeScript declaration files in Kotlin/Wasm is Experimental. It may be dropped or changed at any time.
>
> 

Type correspondence

Kotlin/Wasm allows only certain types in signatures of JavaScript interop declarations. These limitations apply uniformly to declarations with `external`, `= js("code")` or `@JsExport`.

See how Kotlin types correspond to Javascript types:

Kotlin	JavaScript
`Byte`, `Short`, `Int`, `Char`, `UByte`, `UShort`, `UInt`,	`Number`
`Float`, `Double`,	`Number`
`Long`, `ULong`,	`BigInt`
`Boolean`,	`Boolean`
`String`,	`String`
`Unit` in return position	`undefined`
Function type, for example `(String) -> Int`	Function
`JsAny` and subtypes	Any JavaScript value

`JsReference`	Opaque reference to Kotlin object
Other types	Not supported

You can use nullable versions of these types as well.

JsAny type

JavaScript values are represented in Kotlin using the `JsAny` type and its subtypes.

The Kotlin/Wasm standard library provides representation for some of these types:

- Package `kotlin.js`:
  - `JsAny`
  - `JsBoolean`, `JsNumber`, `JsString`
  - `JsArray`
  - `Promise`

You can also create custom `JsAny` subtypes by declaring an `external` interface or class.

JsReference type

Kotlin values can be passed to JavaScript as opaque references using the `JsReference` type.

For example, if you want to expose this Kotlin class `User` to JavaScript:

```
class User(var name: String)
```

You can use the `toJsReference()` function to create `JsReference<User>` and return it to JavaScript:

```
@JsExport
fun createUser(name: String): JsReference<User> {
    return User(name).toJsReference()
}
```

These references aren't directly available in JavaScript and behave like empty frozen JavaScript objects. To operate on these objects, you need to export more functions to JavaScript using the `get()` method where you unwrap the reference value:

```
@JsExport
fun setUserName(user: JsReference<User>, name: String) {
    user.get().name = name
}
```

You can create a class and change its name from JavaScript:

```
import UserLib from "./userlib.mjs"

let user = UserLib.createUser("Bob");
UserLib.setUserName(user, "Alice");
```

Type parameters

JavaScript interop declarations can have type parameters if they have an upper bound of `JsAny` or its subtypes. For example:

```
external fun <T : JsAny> processData(data: JsArray<T>): T
```

Exception handling

You can use Kotlin `try-catch` expression to catch JavaScript exceptions. However, accessing specific details about the thrown value in Kotlin/Wasm isn't possible by default.

You can configure the `JsException` type to include the original error message and stack trace from JavaScript. To do so, add the following compiler option to your `build.gradle.kts` file:

```
kotlin {
    wasmJs {
        compilerOptions {
            freeCompilerArgs.add("-Xwasm-attach-js-exception")
        }
    }
}
```

This behavior depends on the `WebAssembly.JSTag` API, which is only available in certain browsers:

- **Chrome:** Supported from version 115
- **Firefox:** Supported from version 129
- **Safari:** Not yet supported

Here's an example demonstrating this behavior:

```
external object JSON {
    fun <T: JsAny> parse(json: String): T
}

fun main() {
    try {
```

```
        JSON.parse("an invalid JSON")
    } catch (e: JsException) {
        println("Thrown value is: ${e.thrownValue}")
        // SyntaxError: Unexpected token 'a', "an invalid JSON" is not valid JSON

        println("Message: ${e.message}")
        // Message: Unexpected token 'a', "an invalid JSON" is not valid JSON

        println("Stacktrace:")
        // Stacktrace:

        // Prints the full JavaScript stack trace
        e.printStackTrace()
    }
}
```

With the `-Xwasm-attach-js-exception` compiler option enabled, the `JsException` type provides specific details from the JavaScript error. Without enabling this compiler option, `JsException` includes only a generic message stating that an exception was thrown while running JavaScript code.

If you try to use a JavaScript `try-catch` expression to catch Kotlin/Wasm exceptions, it looks like a generic `WebAssembly.Exception` without directly accessible messages and data.

Kotlin/Wasm and Kotlin/JS interoperability differences

Although Kotlin/Wasm interoperability shares similarities with Kotlin/JS interoperability, there are key differences to consider:

	Kotlin/Wasm	Kotlin/JS
External enums	Doesn't support external enum classes.	Supports external enum classes.
Type extensions	Doesn't support non-external types to extend external types.	Supports non-external types.
`JsName` annotation	Only has an effect when annotating external declarations.	Can be used to change names of regular non-external declarations.
`js()` function	`js("code")` function calls are allowed as a single expression body of package-level functions.	The `js("code")` function can be called in any context and returns a `dynamic` value.
Module	Supports ES modules only. There is no analog of the `@JsNonModule` annotation. Provides its exports as properties on the `default` object.	Supports ES modules and legacy module systems. Provides named ESM

	exports as properties on the `default` object. Allows exporting package-level functions only.	exports. Allows exporting classes and objects.
Types	Applies stricter type restrictions uniformly to all interop declarations `external` , `= js("code")` , and `@JsExport` . Allows a select number of built-in Kotlin types and `JsAny` subtypes.	Allows all types in `external` declarations. Restricts types that can be used in `@JsExport` .
Long	Type corresponds to JavaScript `BigInt` .	Visible as a custom class in JavaScript.
Arrays	Not supported in interop directly yet. You can use the new `JsArray` type instead.	Implemented as JavaScript arrays.
Other types	Requires `JsReference<>` to pass Kotlin objects to JavaScript.	Allows the use of non-external Kotlin class types in external declarations.
Exception handling	You can catch any JavaScript exception with the `JsException` and `Throwable` types.	Can catch JavaScript `Error` using the `Throwable` type. It can catch any JavaScript exception using the `dynamic` type.
Dynamic types	Does not support the `dynamic` type. Use `JsAny` instead (see sample code below).	Supports the `dynamic` type.

> Kotlin/JS dynamic type for interoperability with untyped or loosely typed objects is not supported in Kotlin/Wasm. Instead of `dynamic` type, you can use `JsAny` type:
>
> ```
> // Kotlin/JS
> fun processUser(user: dynamic, age: Int) {
>     // ...
>     user.profile.updateAge(age)
>     // ...
> }
>
> // Kotlin/Wasm
> private fun updateUserAge(user: JsAny, age: Int): Unit =
>     js("{ user.profile.updateAge(age); }")
>
> fun processUser(user: JsAny, age: Int) {
>     // ...
>     updateUserAge(user, age)
>     // ...
> }
> ```
>
> 

Web-related browser APIs

The `kotlinx-browser` library is a standalone library that provides JavaScript browser APIs, including:

- Package `org.khronos.webgl`:
  - Typed arrays, like `Int8Array`.
  - WebGL types.
- Packages `org.w3c.dom.*`:
  - DOM API types.
- Package `kotlinx.browser`:
  - DOM API global objects, like `window` and `document`.

To use the declarations from the `kotlinx-browser` library, add it as a dependency in your project's build configuration file:

```
val wasmJsMain by getting {
    dependencies {
        implementation("org.jetbrains.kotlinx:kotlinx-browser:0.3")
    }
}
```

故障排查

> Kotlin/Wasm is Alpha. It may be changed at any time. Use it in scenarios before production. We would appreciate your feedback in YouTrack.
>
> 

Kotlin/Wasm relies on new WebAssembly proposals like garbage collection and exception handling to introduce improvements and new features within WebAssembly.

However, to ensure these features work properly, you need an environment that supports the new proposals. In some cases, you may need to set up the environment to make it compatible with the proposals.

Browser versions

To run applications built with Kotlin/Wasm in a browser, you need a browser version supporting the new WebAssembly garbage collection (WasmGC) feature. Check if the browser version supports the new WasmGC by default or if you need to make changes to the environment.

Chrome

- **For version 119 or later:**

  Works by default.

- **For older versions:**

  > To run applications in an older browser, you need a Kotlin version older than 1.9.20.
  >
  > 

  1. In your browser, go to `chrome://flags/#enable-webassembly-garbage-collection`.
  2. Enable **WebAssembly Garbage Collection**.
  3. Relaunch your browser.

Chromium-based

Including Chromium-based browsers such as Edge, Brave, Opera, or Samsung Internet.

- **For version 119 or later:**

Works by default.

- **For older versions:**

  > To run applications in an older browser, you need a Kotlin version older than 1.9.20.
  >
  > 

  Run the application with the `--js-flags=--experimental-wasm-gc` command line argument.

Firefox

- **For version 120 or later:**

  Works by default.

- **For version 119:**

  1. In your browser, go to `about:config`.
  2. Enable `javascript.options.wasm_gc` option.
  3. Refresh the page.

Safari/WebKit

- **For version 18.2 or later:**

  Works by default.

- **For older versions:**

  Not supported.

> Safari 18.2 is available for iOS 18.2, iPadOS 18.2, visionOS 2.2, macOS 15.2, macOS Sonoma, and macOS Ventura. On iOS and iPadOS, Safari 18.2 is bundled with the operating system. To get it, update your device to version 18.2 or later.
>
> For more information, see the Safari release notes.
>
> 

Wasm proposals support

Kotlin/Wasm improvements are based on WebAssembly proposals. Here you can find details about the support for WebAssembly's garbage collection and (legacy) exception handling proposals.

Garbage collection proposal

Since Kotlin 1.9.20, the Kotlin toolchain uses the latest version of the Wasm garbage collection (WasmGC) proposal.

For this reason, we strongly recommend that you update your Wasm projects to the latest version of Kotlin. We also recommend you use the latest versions of browsers with the Wasm environment.

Exception handling proposal

The Kotlin toolchain uses the legacy exception handling proposal by default which allows running produced Wasm binaries in wider range of environments.

Since Kotlin 2.0.0, we have introduced support for the new version of Wasm exception handling proposal within Kotlin/Wasm.

This update ensures the new exception handling proposal aligns with Kotlin requirements, enabling the use of Kotlin/Wasm on virtual machines that only support the latest version of the proposal.

The new exception handling proposal is activated using the `-Xwasm-use-new-exception-proposal` compiler option. It is turned off by default.

> Learn more about setting up projects, using dependencies, and other tasks with our Kotlin/Wasm examples.
>
> 

Use default import

Importing Kotlin/Wasm code into Javascript has shifted to named exports, moving away from default exports.

If you still want to use a default import, generate a new JavaScript wrapper module. Create a `.mjs` file with the following snippet:

```
// Specifies the path to the main .mjs file
import * as moduleExports from "./wasm-test.mjs";

export { moduleExports as default };
```

You can place your new `.mjs` file in the resources folder, and it will automatically be placed next to the main `.mjs` file during the build process.

You can also place your `.mjs` file in a custom location. In this case, you need to either manually move it next to the main `.mjs` file or adjust the path in the import statement to match its location.

Slow Kotlin/Wasm compilation

When working on Kotlin/Wasm projects, you may experience slow compilation times. This happens because the Kotlin/Wasm toolchain recompiles the entire codebase every time you make a change.

To mitigate this issue, Kotlin/Wasm targets support incremental compilation, which enables the compiler to recompile only those files relevant to changes from the last compilation.

Using incremental compilation reduces the compilation time. It doubles the development speed for now, with plans to improve it further in future releases.

In the current setup, incremental compilation for the Wasm targets is disabled by default. To enable it, add the following line to your project's `local.properties` or `gradle.properties` file:

```
kotlin.incremental.wasm=true
```

> Try out the Kotlin/Wasm incremental compilation and share your feedback. Your insights help make the feature stable and enabled by default sooner.
>
> 

JavaScript

- 搭建 Kotlin/JS 项目
- 运行 Kotlin/JS
- 开发服务器与持续编译
- 调试 Kotlin/JS 代码
- 在 Kotlin/JS 平台中运行测试
- Kotlin/JS 无用代码消除
- Kotlin/JS IR 编译器
- Kotlin 用于 JS 平台
  - 浏览器与 DOM API
  - 在 Kotlin 中使用 JavaScript 代码
  - 动态类型
  - 使用来自 npm 的依赖
  - 在 JavaScript 中使用 Kotlin 代码
  - JavaScript 模块
  - Kotlin/JS 反射
  - 类型安全的 HTML DSL

搭建 Kotlin/JS 项目

Kotlin/JS 项目使用 Gradle 作为构建系统。为了开发者轻松管理其 Kotlin/JS 项目，我们提供了 `kotlin.multiplatform` Gradle 插件，该插件提供项目配置工具以及用以自动执行 JavaScript 开发中常用的例程的帮助程序。

该插件使用 npm 或 Yarn 包管理器在后台下载 npm 依赖，并使用 webpack 由 Kotlin 项目构建 JavaScript 包。 可以直接从 Gradle 构建文件中对依赖项管理与配置进行很大程度的调整， 并且可以选择覆盖自动生成的配置以实现完全控制。

You can apply the `org.jetbrains.kotlin.multiplatform` plugin to a Gradle project manually in the `build.gradle(.kts)` file:

> 【Kotlin】

```
plugins {
    kotlin("multiplatform") version "2.1.21"
}
```

> 【Groovy】

```
plugins {
    id 'org.jetbrains.kotlin.multiplatform' version '2.1.21'
}
```

The Kotlin Multiplatform Gradle plugin lets you manage aspects of your project in the `kotlin {}` block of the build script:

```
kotlin {
    // ...
}
```

在 `kotlin {}` 块内，可以管理以下几方面：

- 选择执行环境: 浏览器或 Node.js
- Support for ES2015 features: classes, modules, and generators

- 管理依赖: Maven 和 npm
- 配置 run 任务
- 配置 test 任务
- 为浏览器项目配置 webpack 绑定与 CSS 支持
- 分发目标目录与模块名
- 项目的 `package.json` 文件

执行环境

Kotlin/JS 项目可以针对两个不同的执行环境：

- Browser，用于浏览器中客户端脚本
- Node.js，用于在浏览器外部运行 JavaScript 代码，例如，用于服务器端脚本。

要定义 Kotlin/JS 项目的目标执行环境，请在 `js {}` 块添加 `browser {}` 或 `nodejs {}`：

```
kotlin {
    js {
        browser {
        }
        binaries.executable()
    }
}
```

指令 `binaries.executable()` 明确指示 Kotlin 编译器发出可执行的 `.js` 文件。 省略 `binaries.executable()` 将导致编译器仅生成 Kotlin 内部的库文件，该文件可以从其他项目中使用，而不能单独运行。

> This is typically faster than creating executable files, and can be a possible optimization when dealing with non-leaf modules of your project.
>
> 

Kotlin 多平台插件会自动配置其任务与所选环境配合工作。 这项操作包括下载与安装运行和测试应用程序所需的环境与依赖项。 这让开发者无需额外配置就可以构建、运行和测试简单项目。 For projects targeting Node.js, there is also an option to use an existing Node.js installation. Learn how to use pre-installed Node.js.

Support for ES2015 features

Kotlin provides an Experimental support for the following ES2015 features:

- Modules that simplify your codebase and improve maintainability.

- Classes that allow incorporating OOP principles, resulting in cleaner and more intuitive code.
- Generators for compiling suspend functions that improve the final bundle size and help with debugging.

You can enable all the supported ES2015 features at once by adding the `es2015` compilation target to your `build.gradle(.kts)` file:

```
tasks.withType<KotlinJsCompile>().configureEach {
    kotlinOptions {
        target = "es2015"
    }
}
```

Learn more about ES2015 (ECMAScript 2015, ES6) in the official documentation.

依赖项

就像其他任何的 Gradle 项目一样，Kotlin/JS 项目支持位于构建脚本的 `dependencies {}` 块中的传统 Gradle 依赖声明。

> 【Kotlin】

```
dependencies {
    implementation("org.example.myproject", "1.1.0")
}
```

> 【Groovy】

```
dependencies {
    implementation 'org.example.myproject:1.1.0'
}
```

Kotlin Multiplatform Gradle 插件还支持构建脚本的 `kotlin {}` 块中特定 `sourceSets` 的依赖声明。

> 【Kotlin】

```
kotlin {
    sourceSets {
        val jsMain by getting {
            dependencies {
                implementation("org.example.myproject:1.1.0")
            }
        }
    }
}
```

【Groovy】

```
kotlin {
    sourceSets {
        jsMain {
            dependencies {
                implementation 'org.example.myproject:1.1.0'
            }
        }
    }
}
```

在面向 JavaScript 时，并非所有适用于 Kotlin 编程语言的库都可用： 仅可以使用包含 Kotlin/JS 构件的库。



如果添加的库对来自 npm 的包有依赖，Gradle 也会自动解析这些传递依赖。

Kotlin 标准库

The dependencies on the standard library are added automatically. The version of the standard library is the same as the version of the Kotlin Multiplatform plugin.

For multiplatform tests, the `kotlin.test` API is available. When you create a multiplatform project, you can add test dependencies to all the source sets by using a single dependency in `commonTest`:

【Kotlin】

```
kotlin {
    sourceSets {
        commonTest.dependencies {
```

```
            implementation(kotlin("test")) // Brings all the platform dependencies aut
omatically
        }
    }
}
```

【Groovy】

```
kotlin {
    sourceSets {
        commonTest {
            dependencies {
                implementation kotlin("test") // Brings all the platform dependencies
automatically
            }
        }
    }
}
```

npm 依赖

在 JavaScript 中，管理依赖项最常用的方式是 npm。 它提供了最大的 JavaScript 模块公开存储库。

Kotlin Multiplatform Gradle 插件使你可以在 Gradle 构建脚本中声明 npm 依赖项，正如声明其他依赖项的方式。

要声明 npm 依赖项，将其名称与版本传给依赖项声明内的 `npm()` 函数。 还可以根据 npm 的 semver 语法指定一个或多个版本范围。

【Kotlin】

```
dependencies {
    implementation(npm("react", "> 14.0.0 <=16.9.0"))
}
```

【Groovy】

```
dependencies {
```

```
    implementation npm('react', '> 14.0.0 <=16.9.0')
}
```

By default, the plugin uses a separate instance of the Yarn package manager to download and install npm dependencies. It works out of the box without additional configuration, but you can tune it to specific needs.

You can also work with npm dependencies directly using the npm package manager instead. To use npm as your package manager, in your `gradle.properties` file, set the following property:

```
kotlin.js.yarn=false
```

除了常规的依赖之外，还有三种依赖类型可以从 Gradle DSL 中使用。 要了解更多关于哪种类型的依赖最适合使用的信息，请查看 npm 链接的官方文档：

- devDependencies，经过 `devNpm(...)`，
- optionalDependencies 经过 `optionalNpm(...)`，与
- peerDependencies 经过 `peerNpm(...)`.

安装 npm 依赖项后，你可以按照在 Kotlin 中调用 JS 中所述，在代码中使用其 API。

run 任务

Kotlin 多平台 Gradle 插件提供了一个 `jsBrowserDevelopmentRun` 任务，使你无需额外配置即可运行纯 Kotlin/JS 项目。

对于运行 Kotlin/JS 项目在浏览器中，此任务是 `browserDevelopmentRun` 任务的别名（在 Kotlin 多平台项目中也可用）。它使用 webpack-dev-server 来服务 JavaScript 构件。 如果要自定义 `webpack-dev-server` 的配置，例如更改服务器端口， 请使用 webpack 配置文件。

对于运行针对 Node.js 的 Kotlin/JS项目， use the `jsNodeDevelopmentRun` task that is an alias for the `nodeRun` task.

要运行项目，请执行标准生命周期的 `jsBrowserDevelopmentRun` 任务，或对应的别名：

```
./gradlew jsBrowserDevelopmentRun
```

要在对源文件进行更改后自动触发应用程序的重新构建，请使用 Gradle 持续构建（continuous build）特性：

```
./gradlew jsBrowserDevelopmentRun --continuous
```

或者

```
./gradlew jsBrowserDevelopmentRun -t
```

一旦项目构建成功， `webpack-dev-server` 将自动刷新浏览器页面。

test 任务

Kotlin Multiplatform Gradle 插件会自动为项目设置测试基础结构。对于浏览器项目，它将下载并安装具有其他必需依赖的 Karma 测试运行程序； 对于 Node.js 项目，使用 Mocha 测试框架。

该插件还提供了有用的测试功能，例如：

- 源代码映射文件生成
- 测试报告生成
- 在控制台中测试运行结果

该插件默认使用 Headless Chrome 来运行浏览器测试。你还可以通过在构建脚本中的 `useKarma {}` 块中添加相应的条目，从而在其他浏览器中运行测试：

```
kotlin {
    js {
        browser {
            testTask {
                useKarma {
                    useIe()
                    useSafari()
                    useFirefox()
                    useChrome()
                    useChromeCanary()
                    useChromeHeadless()
                    usePhantomJS()
                    useOpera()
                }
            }
        }
        binaries.executable()
        // ……
    }
}
```

Alternatively, you can add test targets for browsers in the `gradle.properties` file:

```
kotlin.js.browser.karma.browsers=firefox,safari
```

This approach allows you to define a list of browsers for all modules, and then add specific browsers in the build scripts of particular modules.

请注意，Kotlin Multiplatform Gradle 插件不会自动安装这些浏览器，只会使用其执行环境中可用的浏览器。例如，如果要在持续集成服务器上执行 Kotlin/JS 测试， 请确保已安装要测试的浏览器。

如果要跳过测试，请将 `enabled = false` 这一行添加到 `testTask {}` 中：

```
kotlin {
    js {
        browser {
            testTask {
                enabled = false
            }
        }
        binaries.executable()
        // ……
    }
}
```

要运行测试，请执行标准生命周期 `check` 任务：

```
./gradlew check
```

To specify environment variables used by your Node.js test runners (for example, to pass external information to your tests, or to fine-tune package resolution), use the `environment()` function with a key-value pair inside the `testTask {}` block in your build script:

```
kotlin {
    js {
        nodejs {
            testTask {
                environment("key", "value")
            }
        }
    }
}
```

Karma 配置

Kotlin Multiplatform Gradle 插件会在构建时自动生成 Karma 配置文件，其中包括来自 `build.gradle(.kts)` 中的 `kotlin.js.browser.testTask.useKarma {}` 块的设置。可以在 `build/js/packages/projectName-test/karma.conf.js` 中找到该文件。 要调整 Karma 使用的配

置，请将其他配置文件放在项目根目录中下名为 `karma.config.d` 的目录中。 此目录中的所有 `.js` 配置文件都将被拾取， 并在构建时自动合并到生成的 `karma.conf.js` 中。

所有 Karma 配置功能在 Karma 文档中都有详细描述。

Webpack 绑定

对于浏览器目标，Kotlin 多平台 Gradle 插件使用众所周知的 Webpack 模块捆绑器。

webpack version

The Kotlin Multiplatform plugin uses webpack 5.

If you have projects created with plugin versions earlier than 1.5.0, you can temporarily switch back to webpack 4 used in these versions by adding the following line to the project's `gradle.properties`：

```
kotlin.js.webpack.major.version=4
```

webpack task

最常见的 webpack 调整可以直接通过 Gradle 构建文件中的 `kotlin.js.browser.webpackTask {}` 配置块进行：

- `outputFileName` ——Webpacked 输出文件的名称。 在执行 webpack 任务后，它将在 `<projectDir>/build/dist/<targetName>` 中生成。默认值为项目名称。
- `output.libraryTarget` ——Webpacked 输出的模块系统。 了解有关 Kotlin/JS 项目可用的模块系统的更多信息。默认值为 `umd`。

```
webpackTask {
    outputFileName = "mycustomfilename.js"
    output.libraryTarget = "commonjs2"
}
```

还可以在 `commonWebpackConfig {}` 块中配置常用的 webpack 设置， 以用于绑定、运行与测试任务。

webpack configuration file

Kotlin Multiplatform Gradle 插件会在构建时自动生成一个标准的 webpack 配置文件。 该文件在 `build/js/packages/projectName/webpack.config.js`。

如果要进一步调整 webpack 配置，请将其他配置文件放在项目根目录中名为 `webpack.config.d` 的目录中。 在构建项目时，所有 `.js` 配置文件都会自动被合并到 `build/js/packages/projectName/webpack.config.js` 文件中。 例如，无需添加新的 webpack loader， 请将以下内容添加到 `webpack.config.d` 目录中的 `.js` 文件中：

> In this case, the configuration object is the `config` global object. You need to modify it in your script.
>
> 

```
config.module.rules.push({
    test: /\.extension$/,
    loader: 'loader-name'
});
```

所有 webpack 配置功能在其 文档 中都有详细说明。

Building executables

为了通过 webpack 构建可执行的 JavaScript 构件，Kotlin 多平台 Gradle 插件包含 `browserDevelopmentWebpack` 与 `browserProductionWebpack` Gradle 任务。

- `browserDevelopmentWebpack` 创建较大的开发构件，但是创建时间很少。 这样，在活动开发过程中使用 `browserDevelopmentWebpack` 任务。
- `browserProductionWebpack` 将无用代码消除应用于生成的构件，并缩小生成的 JavaScript 文件，这需要更多时间，但生成的可执行文件的体积较小。因此， 在准备生产用项目时，请使用 `browserProductionWebpack` 任务。

  执行任一任务分别获得用于开发或生产的构件。除非另有规定， 否则生成的文件将在 `build/dist` 中可用。

```
./gradlew browserProductionWebpack
```

请注意，只有将目标配置为生成可执行文件（通过 `binaries.executable()` ）时，这些任务才可用。

CSS

Kotlin Multiplatform Gradle 插件还支持 webpack 的 CSS 与 style 加载器。尽管可以通过直接修改用于构建项目的 Webpack 配置文件来更改所有选项，但是最常用的设置可以直接从 `build.gradle(.kts)` 文件获得。

要在项目中打开 CSS 支持，请在 `commonWebpackConfig {}` 块的 Gradle 构建文件中设置 `cssSupport.enabled` 选项。使用向导创建新项目时，默认情况下也会启用此配置。

> 【Kotlin】

```
browser {
    commonWebpackConfig {
        cssSupport {
            enabled.set(true)
        }
    }
}
```

> 【Groovy】

```
browser {
    commonWebpackConfig {
        cssSupport {
            it.enabled = true
        }
    }
}
```

另外，可以为选定的任务添加 CSS 支持，例如 `webpackTask {}`、`runTask {}` 与 `testTask {}`。

> 【Kotlin】

```
browser {
    webpackTask {
        cssSupport {
            enabled.set(true)
        }
    }
    runTask {
        cssSupport {
            enabled.set(true)
        }
    }
    testTask {
```

```
        useKarma {
            // ...
            webpackConfig.cssSupport {
                enabled.set(true)
            }
        }
    }
}
```

【Groovy】

```
browser {
    webpackTask {
        cssSupport {
            it.enabled = true
        }
    }
    runTask {
        cssSupport {
            it.enabled = true
        }
    }
    testTask {
        useKarma {
            // ……
            webpackConfig.cssSupport {
                it.enabled = true
            }
        }
    }
}
```

在项目中激活 CSS 支持有助于防止在尝试使用未配置项目中的样式表时发生的常见错误，例如 `Module parse failed: Unexpected character '@' (14:0)`。

可以使用 `cssSupport.mode` 指定应如何处理遇到的 CSS。可以使用以下值：

- `"inline"`（默认）：将样式添加到全局 `<style>` 标签中。
- `"extract"`：样式被提取到单独的文件中。然后可以将它们包含在 HTML 页面中。
- `"import"`：样式作为字符串处理。如果需要从代码访问 CSS（例如：`val styles = require("main.css")`），那么此功能很有用。

要对同一项目使用不同的模式，请使用 `cssSupport.rules`。在这里，可以指定 `KotlinWebpackCssRules` 的列表， 每个列表定义一个模式，比如 include 与 exclude 模式。

Node.js

For Kotlin/JS projects targeting Node.js, the plugin automatically downloads and installs the Node.js environment on the host. You can also use an existing Node.js instance if you have it.

Configuring Node.js settings

You can configure Node.js settings for each subproject, or set them for the project as a whole.

For example, to set the Node.js version for a specific subproject, add the following lines to its Gradle block in your `build.gradle(.kts)` file:

> 【Kotlin】

```
project.plugins.withType<org.jetbrains.kotlin.gradle.targets.js.nodejs.NodeJsPlugin> {
    project.the<org.jetbrains.kotlin.gradle.targets.js.nodejs.NodeJsEnvSpec>().version
 = "your Node.js version"
}
```

> 【Groovy】

```
project.plugins.withType(org.jetbrains.kotlin.gradle.targets.js.nodejs.NodeJsPlugin) {
    project.extensions.getByType(org.jetbrains.kotlin.gradle.targets.js.nodejs.NodeJsE
nvSpec).version = "your Node.js version"
}
```

To set a version for the entire project, including all subprojects, apply the same code to the `allProjects {}` block:

> 【Kotlin】

```
allprojects {
    project.plugins.withType<org.jetbrains.kotlin.gradle.targets.js.nodejs.NodeJsPlugi
n> {
        project.the<org.jetbrains.kotlin.gradle.targets.js.nodejs.NodeJsEnvSpec>().ver
sion = "your Node.js version"
    }
}
```

【Groovy】

```
allprojects {
    project.plugins.withType(org.jetbrains.kotlin.gradle.targets.js.nodejs.NodeJsPlugi
n) {
        project.extensions.getByType(org.jetbrains.kotlin.gradle.targets.js.nodejs.Nod
eJsEnvSpec).version = "your Node.js version"
}
```

> Using the `NodeJsRootPlugin` class to configure Node.js setting for the entire project is deprecated and will eventually stop being supported.
>
> 

Use pre-installed Node.js

If Node.js is already installed on the host where you build Kotlin/JS projects, you can configure the Kotlin Multiplatform Gradle plugin to use it instead of installing its own Node.js instance.

To use a pre-installed Node.js instance, add the following lines to your `build.gradle(.kts)` file:

【Kotlin】

```
project.plugins.withType(org.jetbrains.kotlin.gradle.targets.js.nodejs.NodeJsPlugin) {
    // Set to `true` for default behavior
    project.extensions.getByType(org.jetbrains.kotlin.gradle.targets.js.nodejs.NodeJsE
nvSpec).download = false
}
```

【Groovy】

```
project.plugins.withType<org.jetbrains.kotlin.gradle.targets.js.nodejs.NodeJsPlugin> {
    // Set to `true` for default behavior
    project.the<org.jetbrains.kotlin.gradle.targets.js.nodejs.NodeJsEnvSpec>().downloa
d = false
}
```

Yarn

By default, to download and install your declared dependencies at build time, the plugin manages its own instance of the Yarn package manager. It works out of the box without additional configuration, but you can tune it or use Yarn already installed on your host.

Additional Yarn features: .yarnrc

要配置其他 Yarn 特性，请将 `.yarnrc` 文件放在项目的根目录中。 在构建时，它会被自动拾取。

例如，要将自定义 registry 用于 npm 软件包， 请将以下行添加到项目根目录中名为 `.yarnrc` 的文件中：

```
registry "http://my.registry/api/npm/"
```

要了解有关 `.yarnrc` 的更多信息，请访问 Yarn 官方文档。

Use pre-installed Yarn

If Yarn is already installed on the host where you build Kotlin/JS projects, you can configure the Kotlin Multiplatform Gradle plugin to use it instead of installing its own Yarn instance.

To use the pre-installed Yarn instance, add the following lines to `build.gradle(.kts)`:

> 【Kotlin】

```
rootProject.plugins.withType<org.jetbrains.kotlin.gradle.targets.js.yarn.YarnPlugin> {
    rootProject.the<org.jetbrains.kotlin.gradle.targets.js.yarn.YarnRootExtension>().d
ownload = false
    // "true" for default behavior
}
```

> 【Groovy】

```
rootProject.plugins.withType(org.jetbrains.kotlin.gradle.targets.js.yarn.YarnPlugin) {
    rootProject.extensions.getByType(org.jetbrains.kotlin.gradle.targets.js.yarn.YarnR
ootExtension).download = false
}
```

Version locking via kotlin-js-store

> Version locking via `kotlin-js-store` is available since Kotlin 1.6.10.
>
> 

The `kotlin-js-store` directory in the project root is automatically generated by the Kotlin Multiplatform Gradle plugin to hold the `yarn.lock` file, which is necessary for version locking. The lockfile is entirely managed by the Yarn plugin and gets updated during the execution of the `kotlinNpmInstall` Gradle task.

To follow a recommended practice, commit `kotlin-js-store` and its contents to your version control system. It ensures that your application is being built with the exact same dependency tree on all machines.

If needed, you can change both directory and lockfile names in `build.gradle(.kts)`:

> **【Kotlin】**

```
rootProject.plugins.withType<org.jetbrains.kotlin.gradle.targets.js.yarn.YarnPlugin> {
    rootProject.the<org.jetbrains.kotlin.gradle.targets.js.yarn.YarnRootExtension>().l
ockFileDirectory =
        project.rootDir.resolve("my-kotlin-js-store")
    rootProject.the<org.jetbrains.kotlin.gradle.targets.js.yarn.YarnRootExtension>().l
ockFileName = "my-yarn.lock"
}
```

> **【Groovy】**

```
rootProject.plugins.withType(org.jetbrains.kotlin.gradle.targets.js.yarn.YarnPlugin) {
    rootProject.extensions.getByType(org.jetbrains.kotlin.gradle.targets.js.yarn.YarnR
ootExtension).lockFileDirectory =
        file("my-kotlin-js-store")
    rootProject.extensions.getByType(org.jetbrains.kotlin.gradle.targets.js.yarn.YarnR
ootExtension).lockFileName = 'my-yarn.lock'
}
```

> Changing the name of the lockfile may cause dependency inspection tools to no longer pick up the file.



To learn more about `yarn.lock`, visit the official Yarn documentation.

Reporting that yarn.lock has been updated

Kotlin/JS provides Gradle settings that could notify you if the `yarn.lock` file has been updated. You can use these settings when you want to be notified if `yarn.lock` has been changed silently during the CI build process:

- `YarnLockMismatchReport`, which specifies how changes to the `yarn.lock` file are reported. You can use one of the following values:
  - `FAIL` fails the corresponding Gradle task. This is the default.
  - `WARNING` writes the information about changes in the warning log.
  - `NONE` disables reporting.
- `reportNewYarnLock`, which reports about the recently created `yarn.lock` file explicitly. By default, this option is disabled: it's a common practice to generate a new `yarn.lock` file at the first start. You can use this option to ensure that the file has been committed to your repository.
- `yarnLockAutoReplace`, which replaces `yarn.lock` automatically every time the Gradle task is run.

To use these options, update `build.gradle(.kts)` as follows:

【Kotlin】

```
import org.jetbrains.kotlin.gradle.targets.js.yarn.YarnLockMismatchReport
import org.jetbrains.kotlin.gradle.targets.js.yarn.YarnRootExtension

rootProject.plugins.withType(org.jetbrains.kotlin.gradle.targets.js.yarn.YarnPlugin::c
lass.java) {
    rootProject.the<YarnRootExtension>().yarnLockMismatchReport =
        YarnLockMismatchReport.WARNING // NONE | FAIL
    rootProject.the<YarnRootExtension>().reportNewYarnLock = false // true
    rootProject.the<YarnRootExtension>().yarnLockAutoReplace = false // true
}
```

【Groovy】

```
import org.jetbrains.kotlin.gradle.targets.js.yarn.YarnLockMismatchReport
import org.jetbrains.kotlin.gradle.targets.js.yarn.YarnRootExtension

rootProject.plugins.withType(org.jetbrains.kotlin.gradle.targets.js.yarn.YarnPlugin) {
```

```
    rootProject.extensions.getByType(org.jetbrains.kotlin.gradle.targets.js.yarn.YarnR
ootExtension).yarnLockMismatchReport =
        YarnLockMismatchReport.WARNING // NONE | FAIL
    rootProject.extensions.getByType(org.jetbrains.kotlin.gradle.targets.js.yarn.YarnR
ootExtension).reportNewYarnLock = false // true
    rootProject.extensions.getByType(org.jetbrains.kotlin.gradle.targets.js.yarn.YarnR
ootExtension).yarnLockAutoReplace = false // true
}
```

Installing npm dependencies with --ignore-scripts by default

> Installing npm dependencies with `--ignore-scripts` by default is available since Kotlin 1.6.10.
>
> 

To reduce the likelihood of executing malicious code from compromised npm packages, the Kotlin Multiplatform Gradle plugin prevents the execution of lifecycle scripts during the installation of npm dependencies by default.

You can explicitly enable lifecycle scripts execution by adding the following lines to `build.gradle(.kts)`:

> **【Kotlin】**

```
rootProject.plugins.withType<org.jetbrains.kotlin.gradle.targets.js.yarn.YarnPlugin> {

    rootProject.the<org.jetbrains.kotlin.gradle.targets.js.yarn.YarnRootExtension>().i
gnoreScripts = false
}
```

> **【Groovy】**

```
rootProject.plugins.withType(org.jetbrains.kotlin.gradle.targets.js.yarn.YarnPlugin) {
    rootProject.extensions.getByType(org.jetbrains.kotlin.gradle.targets.js.yarn.YarnR
ootExtension).ignoreScripts = false
}
```

分发目标目录

默认情况下，Kotlin/JS 项目构建的结果位于项目根目录下的 `/build/dist/<targetName>/<binaryName>` 目录中。

> Prior to Kotlin 1.9.0, the default distribution target directory was `/build/distributions`.
>
> {style="note" }

要为项目分发文件设置另一个位置，in your build script inside the `browser {}` block, add a `distribution {}` block and assign a value to the `outputDirectory` property by using the `set()` method. 运行项目构建任务后，Gradle 会将输出的内容同项目资源一起保存在此位置。

> **【Kotlin】**

```
kotlin {
    js {
        browser {
            distribution {
                outputDirectory.set(projectDir.resolve("output"))
            }
        }
        binaries.executable()
        // ……
    }
}
```

> **【Groovy】**

```
kotlin {
    js {
        browser {
            distribution {
                outputDirectory = file("$projectDir/output")
            }
        }
        binaries.executable()
        // ……
    }
}
```

模块名

要调整 JavaScript 模块 的名称（在 `build/js/packages/myModuleName` 中生成），包括相应的 `.js` 与 `.d.ts` 文件，使用 `moduleName` 选项：

```
js {
    moduleName = "myModuleName"
}
```

请注意，这不会影响 `build/dist` 中的 Webpack 输出。

package.json 定制

`package.json` 文件保存 JavaScript 包的元数据。流行的软件仓库（例如 npm）要求所有已发布的软件包都具有此类文件。 软件仓库使用该文件来跟踪与管理软件包发布。

Kotlin Multiplatform Gradle 插件会在构建期间自动为 Kotlin/JS 项目生成 `package.json`。 默认情况下，该文件包含基本数据：名称、版本、许可证、依赖项以及一些其他软件包属性。

除了基本的软件包属性外， `package.json` 还可定义 JavaScript 项目的行为方式， 例如，识别可运行的脚本。

可以通过 Gradle DSL 将自定义条目添加到项目的 `package.json` 中。 如需将自定义字段添加到 `package.json` 中，请使用编译 `package.json` 块中的 `customField()` 函数：

```
kotlin {
    js {
        compilations["main"].packageJson {
            customField("hello", mapOf("one" to 1, "two" to 2))
        }
    }
}
```

在构建项目时，此代码会将以下代码块添加到 `package.json` 文件：

```
"hello": {
    "one": 1,
    "two": 2
}
```

在 npm 文档中了解有关为 npm 仓库编写 `package.json` 文件的更多信息。

运行 Kotlin/JS

Since Kotlin/JS projects are managed with the Kotlin Multiplatform Gradle plugin, you can run your project using the appropriate tasks. If you're starting with a blank project, ensure that you have some sample code to execute. Create the file `src/jsMain/kotlin/App.kt` and fill it with a small "Hello, World"-type code snippet:

```
fun main() {
    console.log("Hello, Kotlin/JS!")
}
```

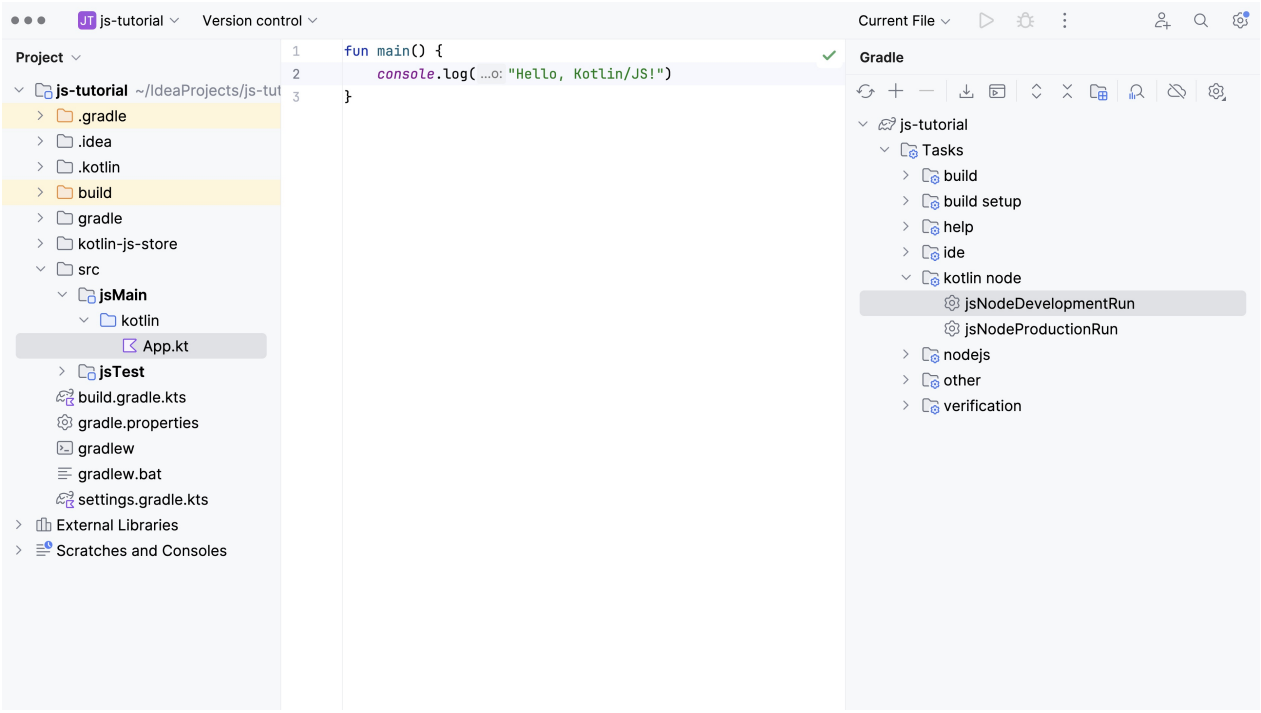
Depending on the target platform, some platform-specific extra setup might be required to run your code for the first time.

Run the Node.js target

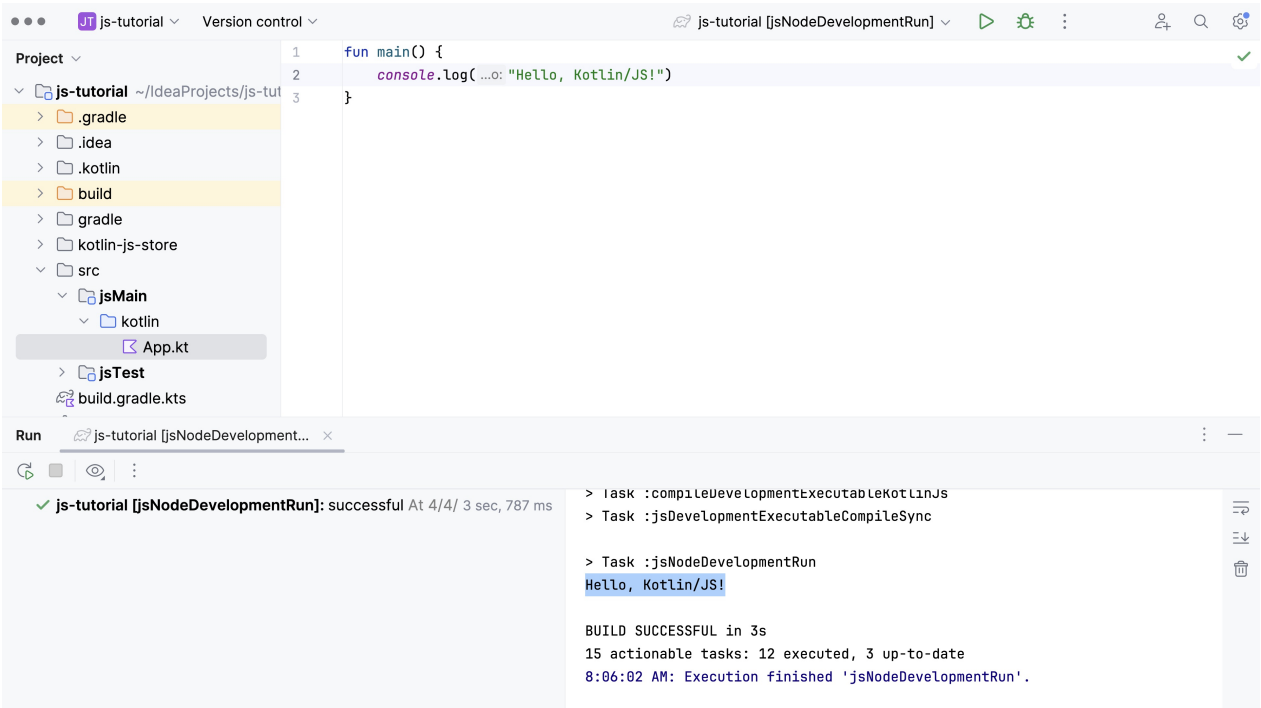
When targeting Node.js with Kotlin/JS, you can simply execute the `jsNodeDevelopmentRun` Gradle task. This can be done for example via the command line, using the Gradle wrapper:

```
./gradlew jsNodeDevelopmentRun
```

If you're using IntelliJ IDEA, you can find the `jsNodeDevelopmentRun` action in the Gradle tool window:



On first start, the `kotlin.multiplatform` Gradle plugin will download all required dependencies to get you up and running. After the build is completed, the program is executed, and you can see the logging output in the terminal:



Run the browser target

When targeting the browser, your project is required to have an HTML page. This page will be served by the development server while you are working on your application, and should embed your compiled Kotlin/JS file. Create and fill an HTML file `/src/jsMain/resources/index.html` :

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>JS Client</title>
</head>
<body>
<script src="js-tutorial.js"></script>
</body>
</html>
```

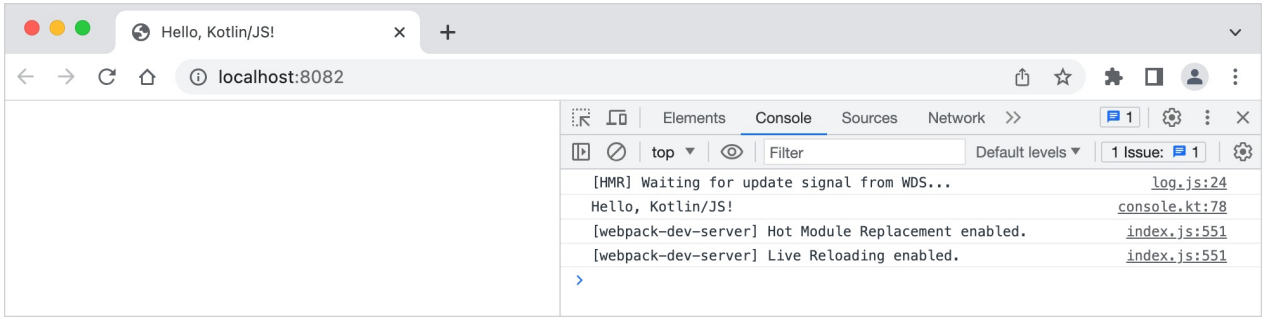
By default, the name of your project's generated artifact (which is created through webpack) that needs to be referenced is your project name (in this case, `js-tutorial`). If you've named your project `followAlong`, make sure to embed `followAlong.js` instead of `js-tutorial.js`

After making these adjustments, start the integrated development server. You can do this from the command line via the Gradle wrapper:

```
./gradlew jsBrowserDevelopmentRun
```

When working from IntelliJ IDEA, you can find the `jsBrowserDevelopmentRun` action in the Gradle tool window.

After the project has been built, the embedded `webpack-dev-server` will start running, and will open a (seemingly empty) browser window pointing to the HTML file you specified previously. To validate that your program is running correctly, open the developer tools of your browser (for example by right-clicking and choosing the *Inspect* action). Inside the developer tools, navigate to the console, where you can see the results of the executed JavaScript code:



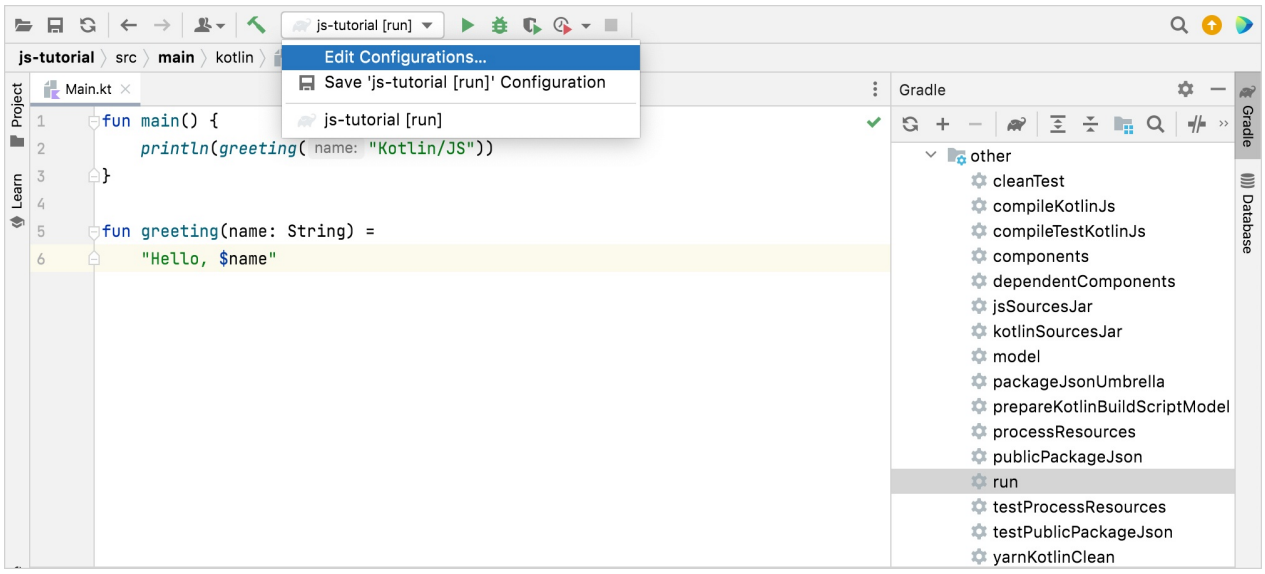
With this setup, you can recompile your project after each code change to see your changes. Kotlin/JS also supports a more convenient way of automatically rebuilding the application while you are developing it. To find out how to set up this *continuous mode*, check out the corresponding tutorial.

开发服务器与持续编译

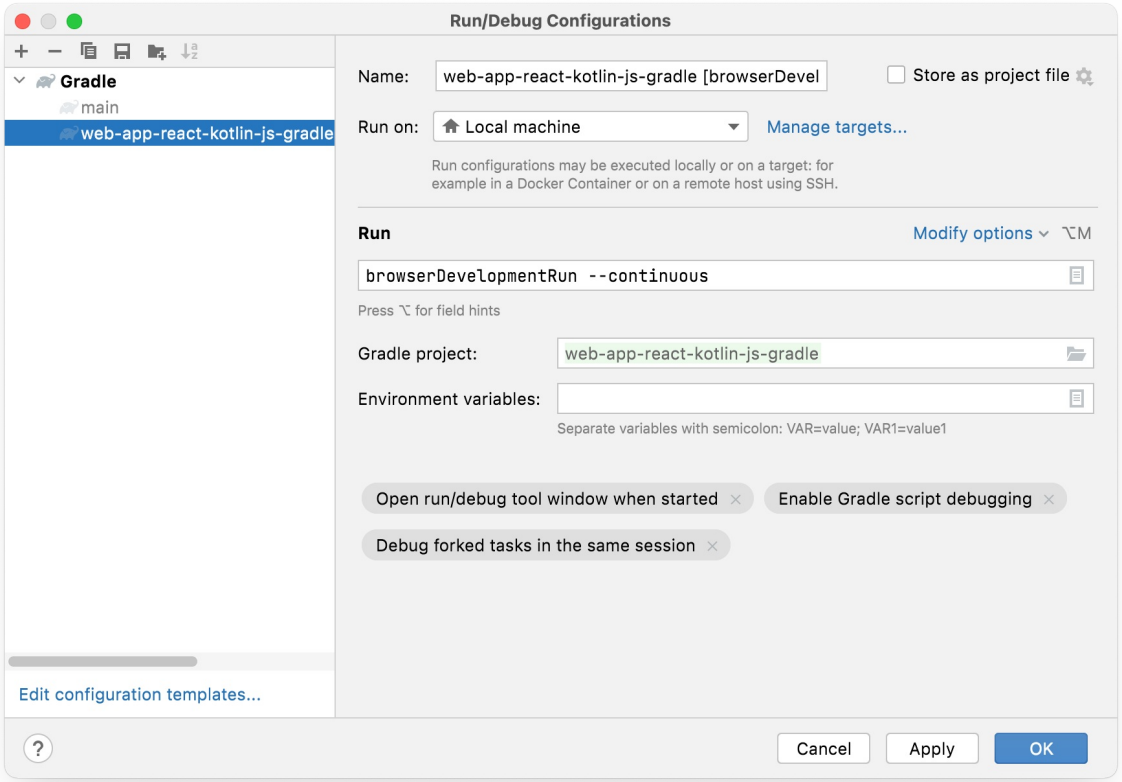
Instead of manually compiling and executing a Kotlin/JS project every time you want to see the changes you made, you can use the *continuous compilation* mode. Instead of using the regular `run` command, invoke the Gradle wrapper in *continuous* mode:

```
./gradlew run --continuous
```

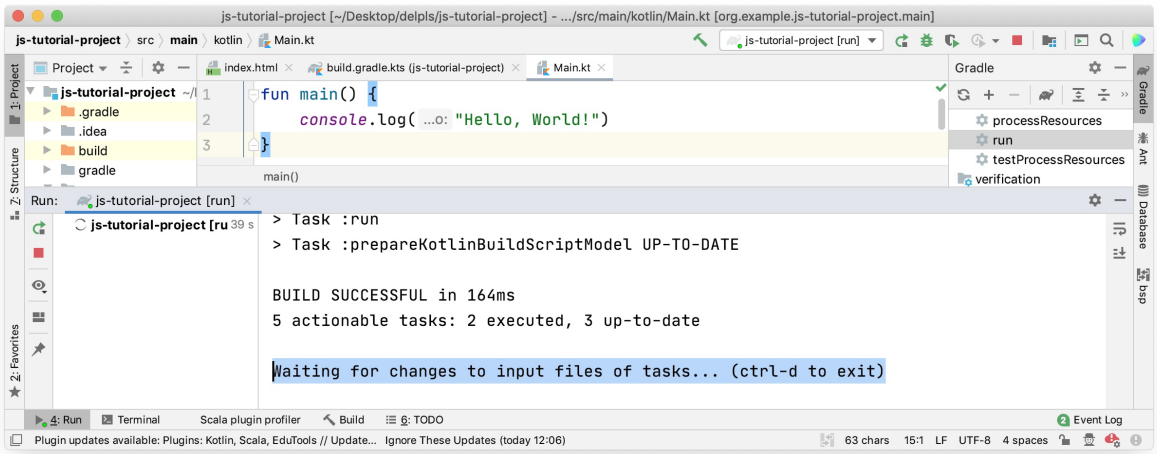
If you are working in IntelliJ IDEA, you can pass the same flag via the *run configuration*. After running the Gradle `run` task for the first time from the IDE, IntelliJ IDEA automatically generates a run configuration for it, which you can edit:



Enabling continuous mode via the **Run/Debug Configurations** dialog is as easy as adding the `--continuous` flag to the arguments for the run configuration:



When executing this run configuration, you can note that the Gradle process continues watching for changes to the program:



Once a change has been detected, the program will be recompiled automatically. If you still have the page open in the browser, the development server will trigger an automatic reload of the page, and the changes will become visible. This is thanks to the integrated `webpack-dev-server` that is managed by the Kotlin Multiplatform Gradle plugin.

调试 Kotlin/JS 代码

JavaScript source maps provide mappings between the minified code produced by bundlers or minifiers and the actual source code a developer works with. This way, the source maps enable support for debugging the code during its execution.

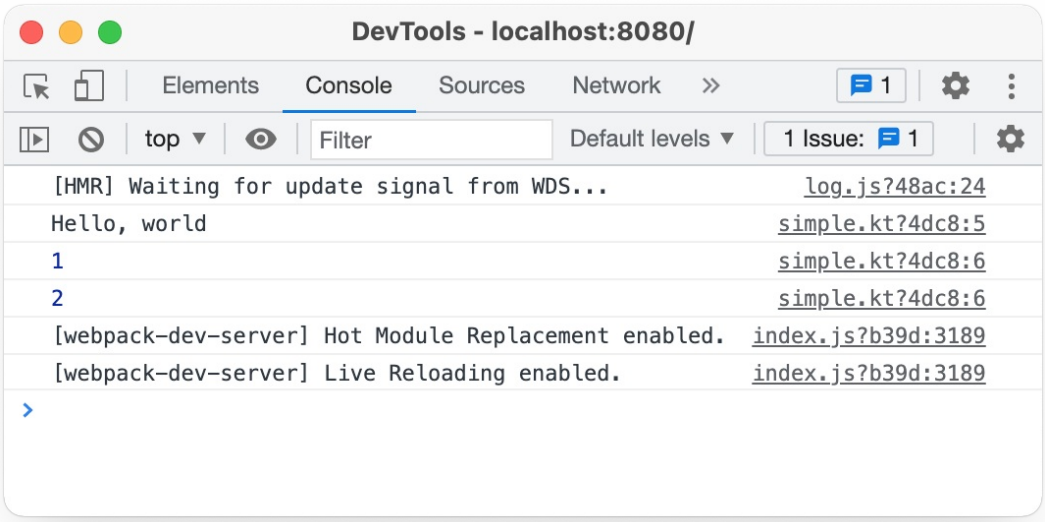
The Kotlin Multiplatform Gradle plugin automatically generates source maps for the project builds, making them available without any additional configuration.

Debug in browser

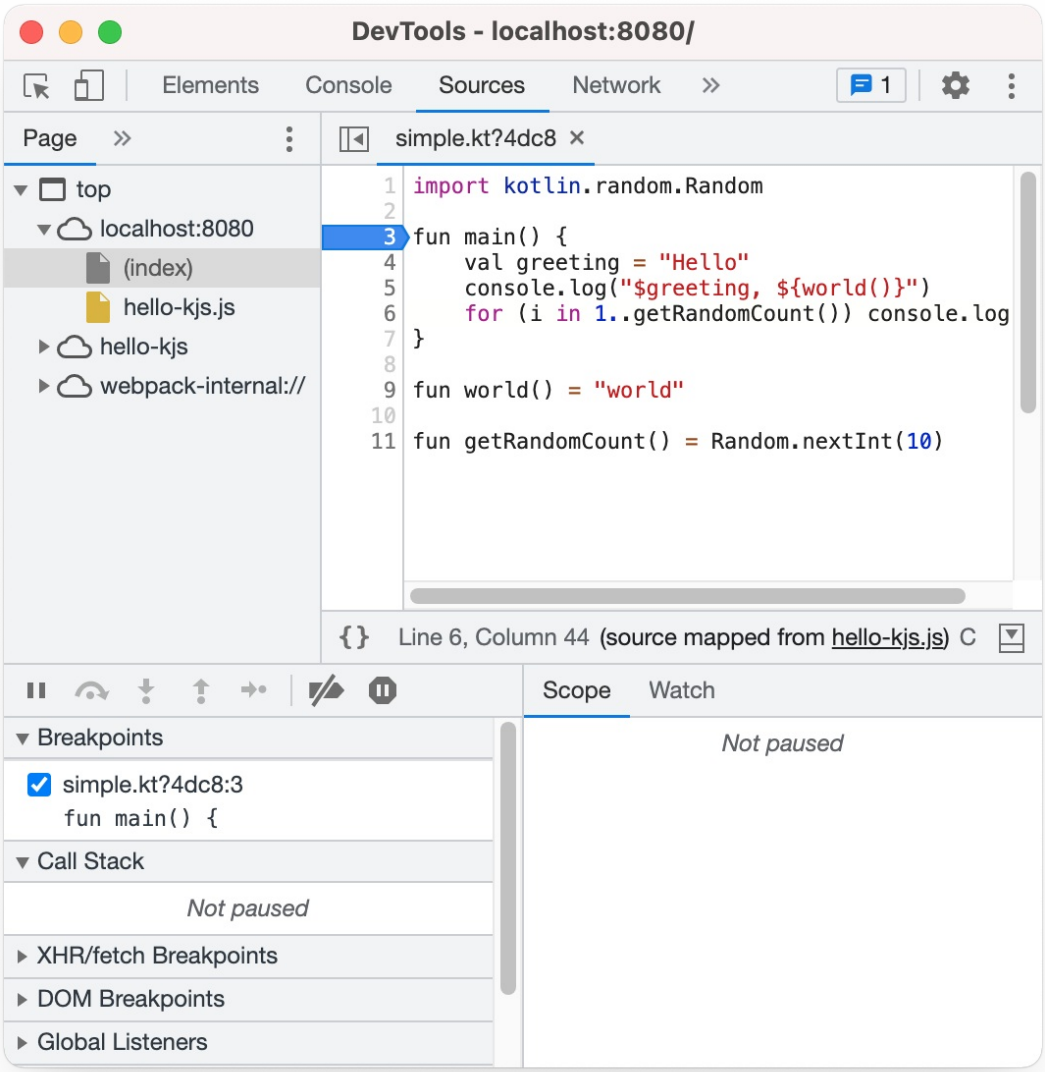
Most modern browsers provide tools that allow inspecting the page content and debugging the code that executes on it. Refer to your browser's documentation for more details.

To debug Kotlin/JS in the browser:

1. Run the project by calling one of the available *run* Gradle tasks, for example, `browserDevelopmentRun` or `jsBrowserDevelopmentRun` in a multiplatform project. Learn more about running Kotlin/JS.
2. Navigate to the page in the browser and launch its developer tools (for example, by right-clicking and selecting the **Inspect** action). Learn how to find the developer tools in popular browsers.
3. If your program is logging information to the console, navigate to the **Console** tab to see this output. Depending on your browser, these logs can reference the Kotlin source files and lines they come from:



1. Click the file reference on the right to navigate to the corresponding line of code. Alternatively, you can manually switch to the **Sources** tab and find the file you need in the file tree. Navigating to the Kotlin file shows you the regular Kotlin code (as opposed to minified JavaScript):



You can now start debugging the program. Set a breakpoint by clicking on one of the line numbers. The developer tools even support setting breakpoints within a statement. As with regular JavaScript code, any set breakpoints will persist across page reloads. This also makes it possible to debug Kotlin's `main()` method which is executed when the script is loaded for the first time.

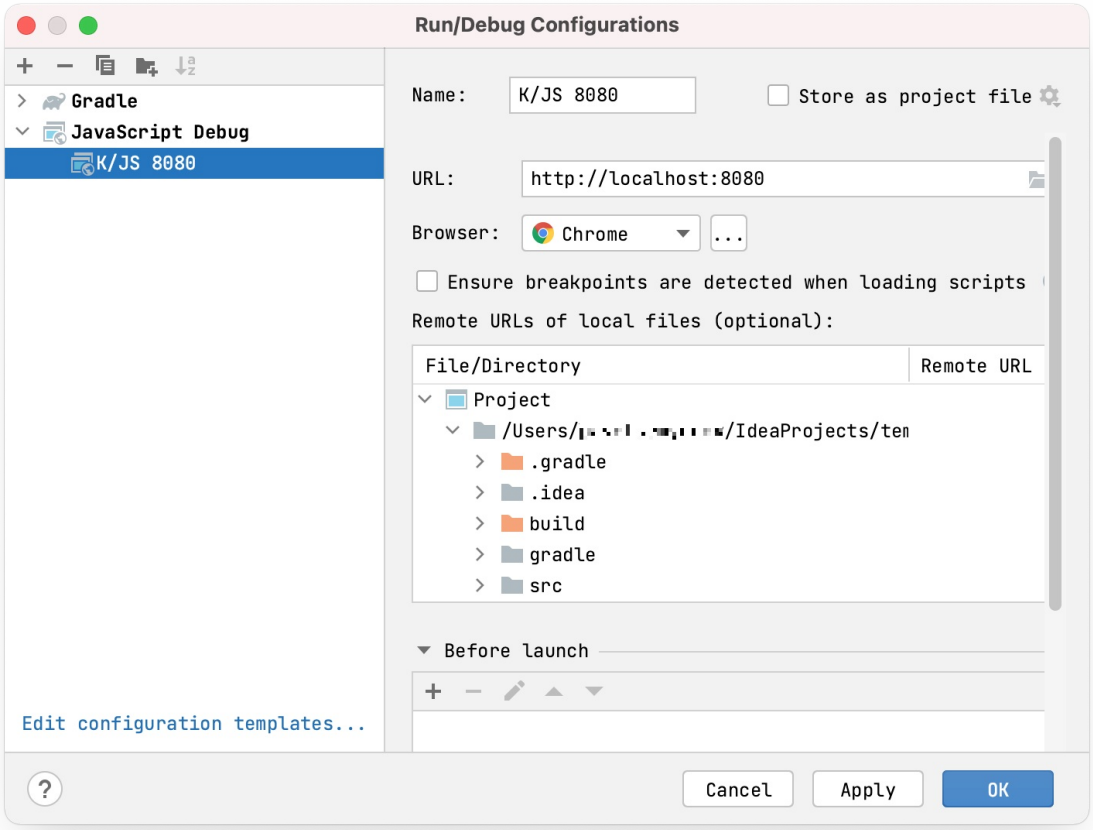
Debug in the IDE

IntelliJ IDEA Ultimate provides a powerful set of tools for debugging code during development.

For debugging Kotlin/JS in IntelliJ IDEA, you'll need a **JavaScript Debug** configuration. To add such a debug configuration:

1. Go to **Run | Edit Configurations**.

2. Click + and select **JavaScript Debug**.
3. Specify the configuration **Name** and provide the **URL** on which the project runs (`http://localhost:8080` by default).



1. Save the configuration.

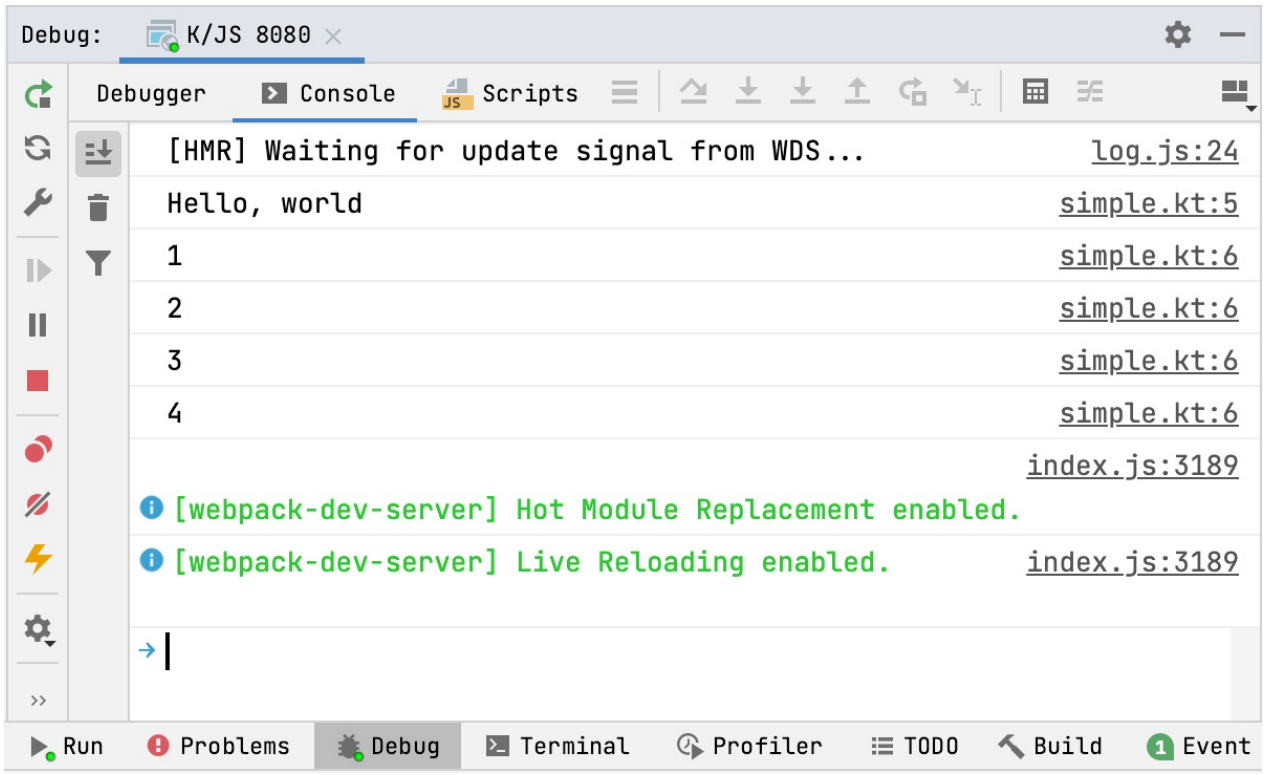
Learn more about setting up JavaScript debug configurations.

Now you're ready to debug your project!

1. Run the project by calling one of the available *run* Gradle tasks, for example, `browserDevelopmentRun` or `jsBrowserDevelopmentRun` in a multiplatform project. Learn more about running Kotlin/JS.
2. Start the debugging session by running the JavaScript debug configuration you've created previously:



1. You can see the console output of your program in the **Debug** window in IntelliJ IDEA. The output items reference the Kotlin source files and lines they come from:



1. Click the file reference on the right to navigate to the corresponding line of code.

You can now start debugging the program using the whole set of tools that the IDE offers: breakpoints, stepping, expression evaluation, and more. Learn more about debugging in IntelliJ IDEA.

> Because of the limitations of the current JavaScript debugger in IntelliJ IDEA, you may need to rerun the JavaScript debug to make the execution stop on breakpoints.
>
> 

Debug in Node.js

If your project targets Node.js, you can debug it in this runtime.

To debug a Kotlin/JS application targeting Node.js:

1. Build the project by running the `build` Gradle task.
2. Find the resulting `.js` file for Node.js in the `build/js/packages/your-module/kotlin/` directory inside your project's directory.
3. Debug it in Node.js as described in the Node.js Debugging Guide.

下一步做什么？

Now that you know how to start debug sessions with your Kotlin/JS project, learn to make efficient use of the debugging tools:

- Learn how to debug JavaScript in Google Chrome
- Get familiar with IntelliJ IDEA JavaScript debugger
- Learn how to debug in Node.js.

If you run into any problems

If you face any issues with debugging Kotlin/JS, please report them to our issue tracker, YouTrack

在 Kotlin/JS 平台中运行测试

The Kotlin Multiplatform Gradle plugin lets you run tests through a variety of test runners that can be specified via the Gradle configuration.

When you create a multiplatform project, you can add test dependencies to all the source sets, including the JavaScript target, by using a single dependency in `commonTest`:

> 【Kotlin】

```
// build.gradle.kts

kotlin {
    sourceSets {
         commonTest.dependencies {
            implementation(kotlin("test")) // This makes test annotations and function
ality available in JS
        }
    }
}
```

> 【Groovy】

```
// build.gradle

kotlin {
    sourceSets {
        commonTest {
            dependencies {
                implementation kotlin("test") // This makes test annotations and funct
ionality available in JS
            }
        }
    }
}
```

You can tune how tests are executed in Kotlin/JS by adjusting the settings available in the `testTask` block in the Gradle build script. For example, using the Karma test runner together with a headless instance of Chrome and an instance of Firefox looks like this:

```
kotlin {
    js {
        browser {
            testTask {
                useKarma {
                    useChromeHeadless()
                    useFirefox()
                }
            }
        }
    }
}
```

For a detailed description of the available functionality, check out the Kotlin/JS reference on configuring the test task.

Please note that by default, no browsers are bundled with the plugin. This means that you'll have to ensure they're available on the target system.

To check that tests are executed properly, add a file `src/jsTest/kotlin/AppTest.kt` and fill it with this content:

```
import kotlin.test.Test
import kotlin.test.assertEquals

class AppTest {
    @Test
    fun thingsShouldWork() {
        assertEquals(listOf(1,2,3).reversed(), listOf(3,2,1))
    }

    @Test
    fun thingsShouldBreak() {
        assertEquals(listOf(1,2,3).reversed(), listOf(1,2,3))
    }
}
```

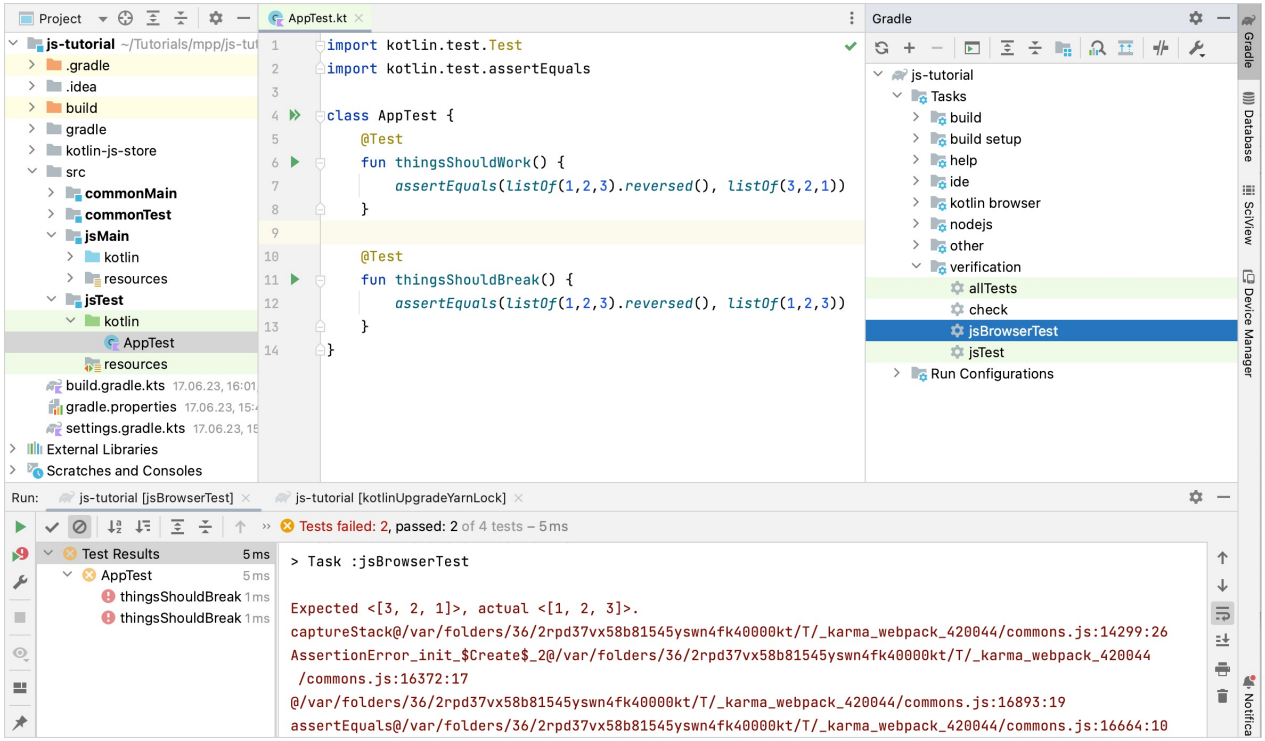
To run the tests in the browser, execute the `jsBrowserTest` task via IntelliJ IDEA, or use the gutter icons to execute all or individual tests:



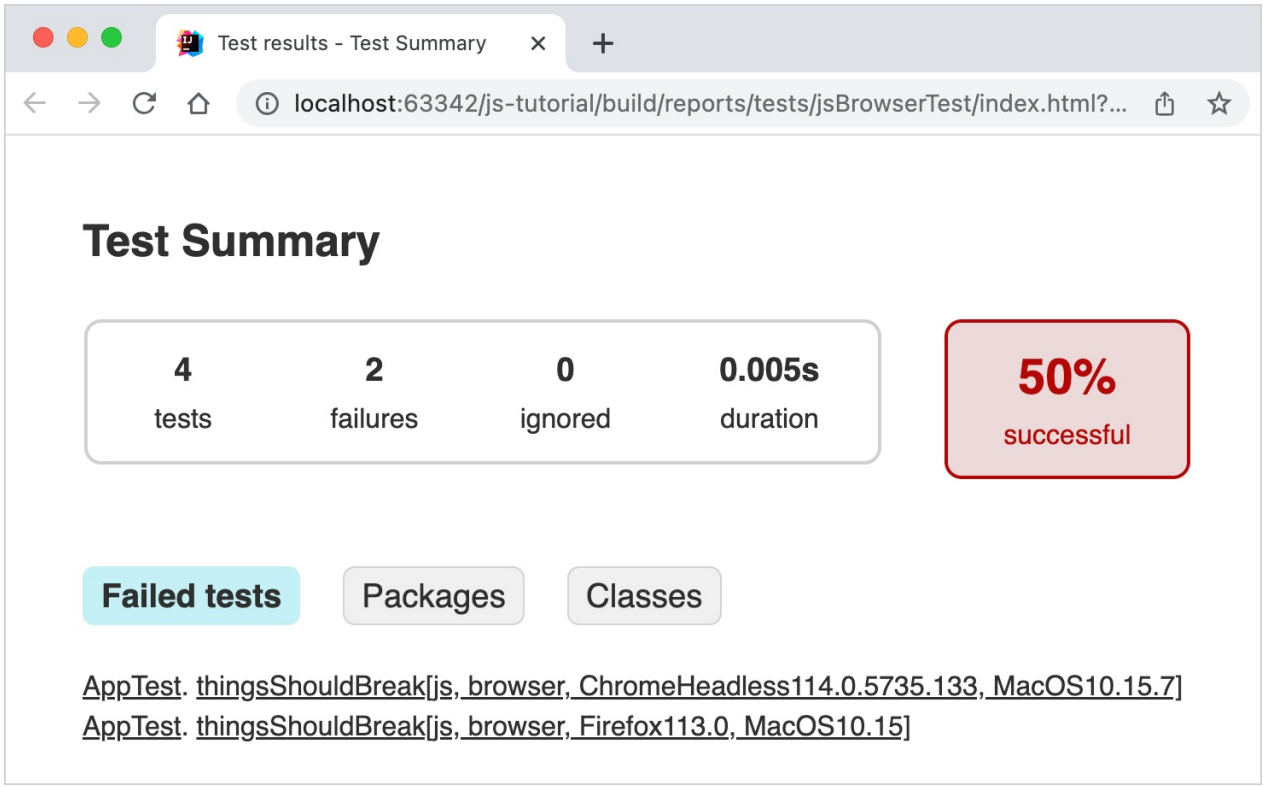
Alternatively, if you want to run the tests via the command line, use the Gradle wrapper:

```
./gradlew jsBrowserTest
```

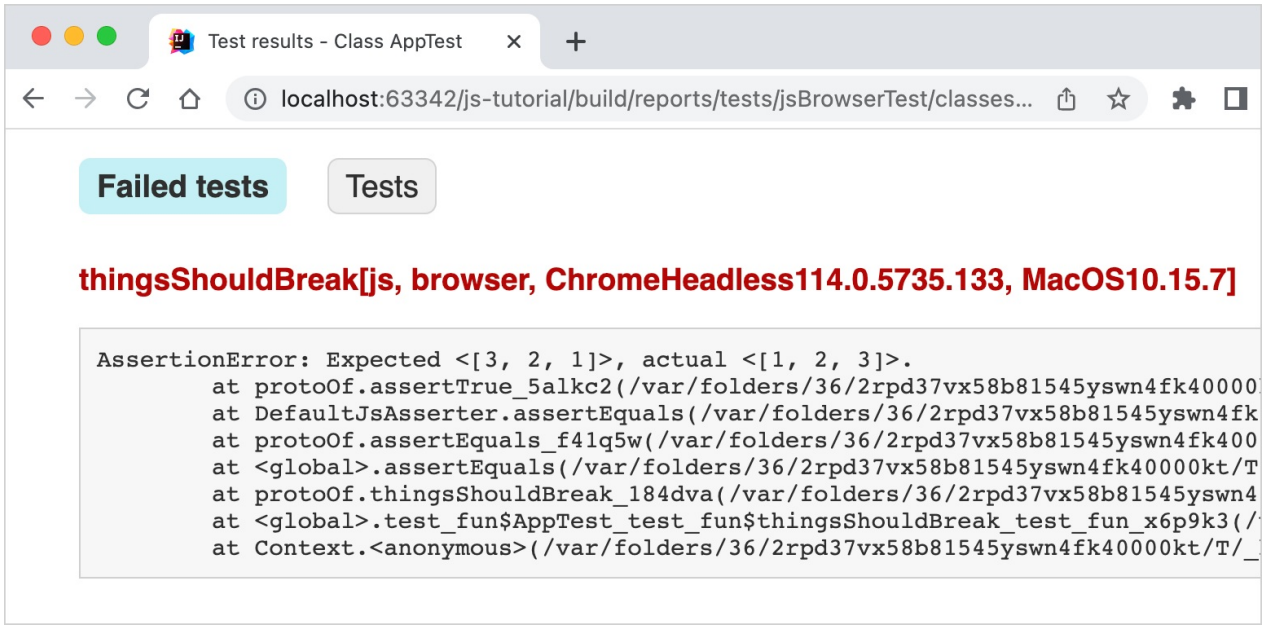
After running the tests from IntelliJ IDEA, the **Run** tool window will show the test results. You can click failed tests to see their stack trace, and navigate to the corresponding test implementation via a double click.



After each test run, regardless of how you executed the test, you can find a properly formatted test report from Gradle in `build/reports/tests/jsBrowserTest/index.html`. Open this file in a browser to see another overview of the test results:



If you are using the set of example tests shown in the snippet above, one test passes, and one test breaks, which gives the resulting total of 50% successful tests. To get more information about individual test cases, you can navigate via the provided hyperlinks:



Kotlin/JS 无用代码消除

> The dead code elimination (DCE) tool is deprecated. The DCE tool was designed for the legacy JS backend, which is now obsolete. The current JS IR backend supports DCE out of the box, and the `@JsExport` annotation allows specifying which Kotlin functions and classes to retain during DCE.
>
> 

Kotlin Multiplatform Gradle 插件包含一个无用代码消除（*DCE*）工具。 无用代码消除通常也称为 摇树。 通过删除未使用的属性、函数和类，它减小了大小或生成的 JavaScript 代码。

在以下情况下会出现未使用的声明：

- 函数是内联的，永远不会直接调用（除了少数情况总是发生）。
- 模块使用共享库。没有 DCE 的情况下，未使用的组件仍会进入结果包。 例如，Kotlin 标准库中包含用于操作列表、数组、字符序列、DOM 适配器的函数。 所有这些功能将需要约 1.3MB 的 JavaScript 文件。 一个简单的 “Hello, world”应用程序仅需要控制台例程，整个程序只有几 KB。

Kotlin Multiplatform Gradle 插件在构建生产包时会自动处理 DCE，例如：使用 `browserProductionWebpack` 任务。开发包任务（例如 `browserDevelopmentWebpack` ）不包含 DCE。

DCE and JavaScript IR compiler

The application of DCE with the IR compiler is as follows:

- DCE is disabled when compiling for development, which corresponds to the following Gradle tasks:
  - `browserDevelopmentRun`
  - `browserDevelopmentWebpack`
  - `nodeDevelopmentRun`
  - `compileDevelopmentExecutableKotlinJs`
  - `compileDevelopmentLibraryKotlinJs`
  - Other Gradle tasks including "development" in their name
- DCE is enabled when compiling for production, which corresponds to the following Gradle tasks:
  - `browserProductionRun`
  - `browserProductionWebpack`

- `compileProductionExecutableKotlinJs`
- `compileProductionLibraryKotlinJs`
- Other Gradle tasks including "production" in their name

With the `@JsExport` annotation, you can specify the declarations you want DCE to treat as roots.

从 DCE 排除的声明

有时，即使未在模块中使用函数或类，也可能需要在结果 JavaScript 代码中保留一个函数或一个类， 例如：如果要在客户端 JavaScript 代码中使用它，则可能会保留该函数或类。

为了避免某些声明被删除，请将 `dceTask` 代码块添加到 Gradle 构建脚本中，并将这些声明列为 `keep` 函数的参数。 参数必须是声明的完整限定名，并且模块名称为前缀：`moduleName.dot.separated.package.name.declarationName`

> 函数与模块名称在生成的 JavaScript 代码中会被修饰， 除非指定了其他名称。为了避免消除这些函数，请在 `keep` 参数中使用修饰的名称—— 就是出现在所生成 JavaScript 代码中的名称。
>
> 

```
kotlin {
    js {
        browser {
            dceTask {
                keep("myKotlinJSModule.org.example.getName", "myKotlinJSModule.org.exa
mple.User" )
            }
            binaries.executable()
        }
    }
}
```

If you want to keep a whole package or module from elimination, you can use its fully qualified name as it appears in the generated JavaScript code.

> 避免删除整个软件包或模块会阻止 DCE 删除许多未使用的声明。 因此，最好逐个选择应从 DCE 中排除的单个声明。
>
> 

禁用 DCE

要完全关闭 DCE，可以使用 `dceTask` 中的 `devMode` 选项：

```
kotlin {
    js {
        browser {
            dceTask {
                dceOptions.devMode = true
            }
        }
        binaries.executable()
    }
}
```

Kotlin/JS IR 编译器

Kotlin/JS IR 编译器后端是围绕 Kotlin/JS 进行创新的主要焦点，并为该技术的发展铺平了道路。

Kotlin/JS IR 编译器后端没有直接从 Kotlin 源代码生成 JavaScript 代码，而是利用了一种新途径。首先将 Kotlin 源代码转换为 Kotlin 中间表示（IR），然后将其编译为 JavaScript。对于 Kotlin/JS，这可以进行积极的优化，并可以改善以前的编译器中存在的痛点，例如生成的代码大小（通过消除无效代码）以及 JavaScript 与 TypeScript 生态系统的互操作性。

从 Kotlin 1.4.0 开始，可以通过 Kotlin Multiplatform Gradle 插件使用 IR 编译器后端。要在项目中启用它，请将编译器类型传递给 Gradle 构建脚本中的 `js` 函数：

```
kotlin {
    js(IR) { // 或：LEGACY、BOTH
        // ...
        binaries.executable() // not applicable to BOTH, see details below
    }
}
```

- `IR` 使用 Kotlin/JS 的新 IR 编译器后端。
- `LEGACY` 使用旧版编译器后端。
- `BOTH` 使用新的 IR 编译器以及默认的编译器后端编译项目。这个模式用于创作与两个后端兼容的库。

> The old compiler backend has been deprecated since Kotlin 1.8.0. Starting with Kotlin 1.9.0, using compiler types `LEGACY` or `BOTH` leads to an error.
>
> 

还可以使用键值 `kotlin.js.compiler=ir` 在 `gradle.properties` 文件中设置编译器类型。但是，`build.gradle(.kts)` 中的任何设置都会覆盖此行为。

Lazy initialization of top-level properties

For better application startup performance, the Kotlin/JS IR compiler initializes top-level properties lazily. This way, the application loads without initializing all the top-level properties used in its code. It initializes only the ones needed at startup; other properties receive their values later when the code that uses them actually runs.

```
val a = run {
```

```
    val result = // intensive computations
    println(result)
    result
} // value is computed upon the first usage
```

If for some reason you need to initialize a property eagerly (upon the application start), mark it with the `@EagerInitialization` {nullable="true"} annotation.

Incremental compilation for development binaries

The JS IR compiler provides the *incremental compilation mode for development binaries* that speeds up the development process. In this mode, the compiler caches the results of `compileDevelopmentExecutableKotlinJs` Gradle task on the module level. It uses the cached compilation results for unchanged source files during subsequent compilations, making them complete faster, especially with small changes.

Incremental compilation is enabled by default. To disable incremental compilation for development binaries, add the following line to the project's `gradle.properties` or `local.properties`:

```
kotlin.incremental.js.ir=false // true by default
```

> The clean build in the incremental compilation mode is usually slower because of the need to create and populate the caches.
>
> 

Output mode

You can choose how the JS IR compiler outputs `.js` files in your project:

- **One per module**. By default, the JS compiler outputs separate `.js` files for each module of a project as a compilation result.
- **One per project**. You can compile the whole project into a single `.js` file by adding the following line to `gradle.properties`:

  ```
  kotlin.js.ir.output.granularity=whole-program // 'per-module' is the default
  ```

- **One per file**. You can set up a more granular output that generates one (or two, if the file contains exported declarations) JavaScript file per each Kotlin file. To enable the per-file compilation mode:

1. Add the `useEsModules()` function to your build file to support ECMAScript modules:

   ```
   // build.gradle.kts
   kotlin {
       js(IR) {
           useEsModules() // Enables ES2015 modules
           browser()
       }
   }
   ```

   Alternatively, you can use the `es2015` compilation target to support ES2015 features in your project.

2. Apply the `-Xir-per-file` compiler option or update your `gradle.properties` file with:

   ```
   # gradle.properties
   kotlin.js.ir.output.granularity=per-file // 'per-module' is the default
   ```

Minification of member names in production

The Kotlin/JS IR compiler uses its internal information about the relationships of your Kotlin classes and functions to apply more efficient minification, shortening the names of functions, properties, and classes. This reduces the size of resulting bundled applications.

This type of minification is automatically applied when you build your Kotlin/JS application in production mode, and enabled by default. To disable member name minification, use the `-Xir-minimized-member-names` compiler option:

```
kotlin {
    js(IR) {
        compilations.all {
            compileTaskProvider.configure {
                compilerOptions.freeCompilerArgs.add("-Xir-minimized-member-names=fals
e")
            }
        }
    }
}
```

预览：TypeScript 声明文件（d.ts）的生成

> The generation of TypeScript declaration files (`d.ts`) is Experimental. It may be dropped or changed at any time. Opt-in is required (see the details below), and you should use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack.
>
> 

Kotlin/JS IR 编译器能够从 Kotlin 代码生成 TypeScript 定义。在混合应用程序上工作时，JavaScript 工具与 IDE 可以使用这些定义来提供自动补全功能、支持静态分析器， 并使在 JavaScript 与 TypeScript 项目中更容易包含 Kotlin 代码。

If your project produces executable files (`binaries.executable()`), the Kotlin/JS IR compiler collects any top-level declarations marked with `@JsExport` and automatically generates TypeScript definitions in a `.d.ts` file.

If you want to generate TypeScript definitions, you have to explicitly configure this in your Gradle build file. Add `generateTypeScriptDefinitions()` to your `build.gradle.kts` file in the `js` section. For example:

```
kotlin {
    js {
        binaries.executable()
        browser {
        }
        generateTypeScriptDefinitions()
    }
}
```

可以在 `build/js/packages/<package_name>/kotlin` 中找到其定义以及相应的未打包 Web 的 JavaScript 代码。

IR 编译器的当前限制

新的 IR 编译器后端的主要变化是与默认后端 没有二进制兼容性。 A library created with the new IR compiler uses a `klib` format and can't be used from the default backend. In the meantime, a library created with the old compiler is a `jar` with `js` files, which can't be used from the IR backend.

如果要为项目使用 IR 编译器后端，则需要 将所有 **Kotlin** 依赖项更新为支持该新后端的版本。由 JetBrains 针对 Kotlin/JS 发布的针对 Kotlin 1.4+ 的库已经包含了与新的 IR 编译器后端一起使用所需的所有构件。

可能库开发者 希望提供与当前编译器后端以及新的 IR 编译器后端的兼容性，请另外查看“为 IR 编译器编写库”相关部分 部分。

与默认后端相比，IR 编译器后端也存在一些差异。在尝试新的后端时， 最好注意这些可能的缺陷。

- 一些 依赖默认后端特定特性的库，例如 `kotlin-wrappers`，可能会显示一些问题。可以在 YouTrack 上跟踪调查与进度。
- 默认情况下，IR 后端根本 不会使 **Kotlin** 声明可用于 **JavaScript**。要使 Kotlin 声明对 JavaScript 可见，必须使用 `@JsExport` 对其进行注解。

Migrating existing projects to the IR compiler

Due to significant differences between the two Kotlin/JS compilers, making your Kotlin/JS code work with the IR compiler may require some adjustments. Learn how to migrate existing Kotlin/JS projects to the IR compiler in the Kotlin/JS IR compiler migration guide.

为 IR 编译器创作具有向后兼容性的库

对于库维护者，希望提供与默认后端以及新的 IR 编译器后端的兼容性，那么可以使用编译器选择的设置，该设置可为两个后端创建构件， 从而保持与以下版本的兼容性。为现有的用户以及下一代 Kotlin 编译器提供支持。 可以使用 `gradle.properties` 文件中的 `kotlin.js.compiler=both` 设置打开这种所谓的 `both` 模式， 也可以将其设置为 `build.gradle(.kts)` 文件内 `js` 块内项目特有的选项之一：

```
kotlin {
    js(BOTH) {
        // ...
    }
}
```

When in `both` mode, the IR compiler backend and default compiler backend are both used when building a library from your sources (hence the name). This means that both `klib` files with Kotlin IR as well as `jar` files for the default compiler will be generated. When published under the same Maven coordinate, Gradle will automatically choose the right artifact depending on the use case – `js` for the old compiler, `klib` for the new one. This enables you to compile and publish your library for projects that are using either of the two compiler backends.

Kotlin 用于 JS 平台

- 浏览器与 DOM API
- 在 Kotlin 中使用 JavaScript 代码
- 动态类型
- 使用来自 npm 的依赖
- 在 JavaScript 中使用 Kotlin 代码
- JavaScript 模块
- Kotlin/JS 反射
- 类型安全的 HTML DSL

浏览器与 DOM API

The Kotlin/JS standard library lets you access browser-specific functionality using the `kotlinx.browser` package, which includes typical top-level objects such as `document` and `window`. The standard library provides typesafe wrappers for the functionality exposed by these objects wherever possible. As a fallback, the `dynamic` type is used to provide interaction with functions that do not map well into the Kotlin type system.

Interaction with the DOM

For interaction with the Document Object Model (DOM), you can use the variable `document`. For example, you can set the background color of our website through this object:

```
document.bgColor = "FFAA12"
```

The `document` object also provides you a way to retrieve a specific element by ID, name, class name, tag name and so on. All returned elements are of type `Element?`. To access their properties, you need to cast them to their appropriate type. For example, assume that you have an HTML page with an email `<input>` field:

```
<body>
    <input type="text" name="email" id="email"/>

    <script type="text/javascript" src="tutorial.js"></script>
</body>
```

Note that your script is included at the bottom of the `body` tag. This ensures that the DOM is fully available before the script is loaded.

With this setup, you can access elements of the DOM. To access the properties of the `input` field, invoke `getElementById` and cast it to `HTMLInputElement`. You can then safely access its properties, such as `value`:

```
val email = document.getElementById("email") as HTMLInputElement
email.value = "hadi@jetbrains.com"
```

Much like you reference this `input` element, you can access other elements on the page, casting them to the appropriate types.

To see how to create and structure elements in the DOM in a concise way, check out the Typesafe HTML DSL.

在 Kotlin 中使用 JavaScript 代码

Kotlin 最初被设计为能够与 Java 平台轻松互操作。它将 Java 类视为 Kotlin 类，并且 Java 也将 Kotlin 类视为 Java 类。

但是，JavaScript 是一种动态类型语言，这意味着它不会在编译期检测类型。可以通过 动态类型在 Kotlin 中自由地与 JavaScript 交流。如果想要使用 Kotlin 类型系统的全部威力， 可以为 JavaScript 库创建 Kotlin 编译器与周边工具可理解的外部声明。

内联 JavaScript

你可以使用 `js()` 函数将 JavaScript 代码嵌入到 Kotlin 代码中。 例如：

```
fun jsTypeOf(o: Any): String {
    return js("typeof o")
}
```

因为 `js` 的参数是在编译期解析并且按原样翻译成 JavaScript 代码的，因此它必须是字符串常量。因此，以下代码是不正确的：

```
fun jsTypeOf(o: Any): String {
    return js(getTypeof() + " o") // 此处报错
}

fun getTypeof() = "typeof"
```

> As the JavaScript code is parsed by the Kotlin compiler, not all ECMAScript features might be supported. In this case, you can encounter compilation errors.
>
> 

请注意，调用 `js()` 会返回动态类型的结果，该结果在编译时不提供任何类型安全性。

external 修饰符

要告诉 Kotlin 某个声明是用纯 JavaScript 编写的，你应该用 `external` 修饰符来标记它。 当编译器看到这样的声明时，它假定相应类、函数或属性的实现是由外部提供的（由开发人员或者通过 npm 依赖项），因此不会尝试从声明中生成任何 JavaScript 代码。 这也是为什么 `external` 声明不能具有主体的原因。例如：

```
external fun alert(message: Any?): Unit

external class Node {
    val firstChild: Node

    fun append(child: Node): Node

    fun removeChild(child: Node): Node

    // 等等
}

external val window: Window
```

请注意，嵌套的声明会继承 `external` 修饰符，这也是 `Node` 类中成员函数和属性之前没有 `external` 修饰符的原因。

`external` 修饰符只允许在包级声明中使用。 你不能声明一个非 `external` 类的 `external` 成员。

声明类的（静态）成员

在 JavaScript 中，你可以在原型或者类本身上定义成员：

```
function MyClass() { …… }
MyClass.sharedMember = function() { /* 实现 */ };
MyClass.prototype.ownMember = function() { /* 实现 */ };
```

Kotlin 中没有这样的语法。然而，在 Kotlin 中我们有伴生（`companion`）对象。 Kotlin 以特殊的方式处理 `external` 类的伴生对象：替代期待一个对象的是， 它假定伴生对象的成员就是该类自身的成员。可以这样描述来自上例中的 `MyClass` ：

```
external class MyClass {
    companion object {
        fun sharedMember()
    }

    fun ownMember()
}
```

声明可选参数

如果正在为具有可选参数的 JavaScript 函数编写外部声明，请使用 `definedExternally` 。 这将默认值的生成委托给 JavaScript 函数本身：

```
external fun myFunWithOptionalArgs(
    x: Int,
    y: String = definedExternally,
    z: String = definedExternally
)
```

使用此外部声明，可以调用带有一个必需参数和两个可选参数的 `myFunWithOptionalArgs`，其中默认值由 `myFunWithOptionalArgs` 的 JavaScript 实现计算得出。

扩展 JavaScript 类

你可以轻松扩展 JavaScript 类，因为它们是 Kotlin 类。只需定义一个 `external open` 类并用非 `external` 类扩展它。例如：

```
open external class Foo {
    open fun run()
    fun stop()
}

class Bar : Foo() {
    override fun run() {
        window.alert("Running!")
    }

    fun restart() {
        window.alert("Restarting")
    }
}
```

有一些限制：

- 当一个外部基类的函数被签名重载时，不能在派生类中覆盖它。
- 不能覆盖一个使用默认实参的函数。
- 不能用外部类扩展非外部类。

external 接口

JavaScript 没有接口的概念。当函数期望其参数支持 `foo` 和 `bar` 两个方法时，只需传入实际具有这些方法的对象。

在静态类型的 Kotlin 中，你可以使用接口来表达这一概念：

```
external interface HasFooAndBar {
    fun foo()

    fun bar()
```

```
}

external fun myFunction(p: HasFooAndBar)
```

外部接口的典型使用场景是描述设置对象。例如：

```
external interface JQueryAjaxSettings {
    var async: Boolean

    var cache: Boolean

    var complete: (JQueryXHR, String) -> Unit

    // 等等
}

fun JQueryAjaxSettings(): JQueryAjaxSettings = js("{}")

external class JQuery {
    companion object {
        fun get(settings: JQueryAjaxSettings): JQueryXHR
    }
}

fun sendQuery() {
    JQuery.get(JQueryAjaxSettings().apply {
        complete = { (xhr, data) ->
            window.alert("Request complete")
        }
    })
}
```

外部接口有一些限制：

- 它们不能在 `is` 检测的右侧使用。
- 它们不能作为具体化类型参数传递。
- 它们不能用在类的字面值表达式（例如 `I::class` ）中。
- `as` 转换为外部接口总是成功。 强制转换为外部接口会产生“未检查强制转换到外部接口（Unchecked cast to external interface ）”编译时警告。可以使用 `@Suppress("UNCHECKED_CAST_TO_EXTERNAL_INTERFACE")` 注解取消警告。

  IntelliJ IDEA 还可以自动生成 `@Suppress` 注解。通过灯泡图标或 `Alt + Enter` 打开意图菜单，然后单击“未检查强制转换到外部接口”检查旁边的小箭头。在这里，可以选择抑制作用域，IDE 将相应地将注解添加到文件中。

强制转换

除了“unsafe”强制转换运算符 `as` （在无法进行强制转换时抛出 `ClassCastException` ）之外， Kotlin/JS 还提供 `unsafeCast<T>()` 。使用 `unsafeCast` 时， 在运行时 完全不进行类型检查。例如，考虑以下两种方法：

```
fun usingUnsafeCast(s: Any) = s.unsafeCast<String>()
fun usingAsOperator(s: Any) = s as String
```

它们将进行相应的编译：

```
function usingUnsafeCast(s) {
    return s;
}

function usingAsOperator(s) {
    var tmp$;
    return typeof (tmp$ = s) === 'string' ? tmp$ : throwCCE();
}
```

Equality

Kotlin/JS has particular semantics for equality checks compared to other platforms.

In Kotlin/JS, the Kotlin referential equality operator (`===`) always translates to the JavaScript strict equality operator (`===`).

The JavaScript `===` operator checks not only that two values are equal but also that the types of these two values are equal:

```
fun main() {
    val name = "kotlin"
    val value1 = name.substring(0, 1)
    val value2 = name.substring(0, 1)

    println(if (value1 === value2) "yes" else "no")
    // Prints 'yes' on Kotlin/JS
    // Prints 'no' on other platforms
}
```

Also, in Kotlin/JS, the `Byte` , `Short` , `Int` , `Float` , and `Double` numeric types are all represented with the `Number` JavaScript type in runtime. Therefore, the values of these five types are indistinguishable:

```
fun main() {
    println(1.0 as Any === 1 as Any)
```

```
    // Prints 'true' on Kotlin/JS
    // Prints 'false' on other platforms
}
```

> For more information about equality in Kotlin, see the Equality documentation.
>
> 

动态类型

> 在面向 JVM 平台的代码中不支持动态类型。
>
> 

Being a statically typed language, Kotlin still has to interoperate with untyped or loosely typed environments, such as the JavaScript ecosystem. To facilitate these use cases, the `dynamic` type is available in the language:

```
val dyn: dynamic = ……
```

`dynamic` 类型基本上关闭了 Kotlin 的类型检测系统：

- A value of the `dynamic` type can be assigned to any variable or passed anywhere as a parameter.
- Any value can be assigned to a variable of the `dynamic` type or passed to a function that takes `dynamic` as a parameter.
- `null` -checks are disabled for the `dynamic` type values.

The most peculiar feature of `dynamic` is that we are allowed to call **any** property or function with any parameters on a `dynamic` variable:

```
dyn.whatever(1, "foo", dyn) // “whatever”在任何地方都没有定义
dyn.whatever(*arrayOf(1, 2, 3))
```

On the JavaScript platform 这段代码会按照原样编译：在生成的 JavaScript 代码中，Kotlin中的 `dyn.whatever(1)` 变为 `dyn.whatever(1)` 。

当在 `dynamic` 类型的值上调用 Kotlin 写的函数时，请记住由 Kotlin 到 JavaScript 编译器执行的名字修饰。你可能需要使用 @JsName 注解 为需要调用的函数分配明确定义的名称。

动态调用总是返回 `dynamic` 作为结果，因此可以自由链式调用：

```
dyn.foo().bar.baz()
```

当把一个 lambda 表达式传给一个动态调用时，它的所有参数默认都是 `dynamic` 类型的：

```
dyn.foo {
    x -> x.bar() // x 是 dynamic
}
```

使用 `dynamic` 类型值的表达式会按照原样转换为 JavaScript，并且不使用 Kotlin 运算符约定。 支持以下运算符：

- 二元：`+`、`-`、`*`、`/`、`%`、`>`、`<`、`>=`、`<=`、`==`、`!=`、`===`、`!==`、`&&`、`||`
- 一元
  - 前置：`-`、`+`、`!`
  - 前置及后置：`++`、`--`
- 赋值：`+=`、`-=`、`*=`、`/=`、`%=`
- 索引访问：
  - 读：`d[a]`，多于一个参数会出错
  - 写：`d[a1] = a2`，`[]` 中有多于一个参数会出错

`in`、`!in` 以及 `..` 操作对于 `dynamic` 类型的值是禁用的。

更多技术说明请参见规范文档。

使用来自 npm 的依赖

In Kotlin/JS projects, all dependencies can be managed through the Gradle plugin. This includes Kotlin/Multiplatform libraries such as `kotlinx.coroutines`, `kotlinx.serialization`, or `ktor-client`.

For depending on JavaScript packages from npm, the Gradle DSL exposes an `npm` function that lets you specify packages you want to import from npm. Let's consider the import of an NPM package called `is-sorted`.

The corresponding part in the Gradle build file looks as follows:

```
dependencies {
    // ...
    implementation(npm("is-sorted", "1.0.5"))
}
```

Because JavaScript modules are usually dynamically typed and Kotlin is a statically typed language, you need to provide a kind of adapter. In Kotlin, such adapters are called *external declarations*. For the `is-sorted` package which offers only one function, this declaration is small to write. Inside the source folder, create a new file called `is-sorted.kt`, and fill it with these contents:

```
@JsModule("is-sorted")
@JsNonModule
external fun <T> sorted(a: Array<T>): Boolean
```

Please note that if you're using CommonJS as a target, the `@JsModule` and `@JsNonModule` annotations need to be adjusted accordingly.

This JavaScript function can now be used just like a regular Kotlin function. Because we provided type information in the header file (as opposed to simply defining parameter and return type to be `dynamic`), proper compiler support and type-checking is also available.

```
console.log("Hello, Kotlin/JS!")
console.log(sorted(arrayOf(1,2,3)))
console.log(sorted(arrayOf(3,1,2)))
```

Running these three lines either in the browser or Node.js, the output shows that the call to `sorted` was properly mapped to the function exported by the `is-sorted` package:

```
Hello, Kotlin/JS!
```

```
true
false
```

Because the JavaScript ecosystem has multiple ways of exposing functions in a package (for example through named or default exports), other npm packages might need a slightly altered structure for their external declarations.

To learn more about how to write declarations, please refer to Calling JavaScript from Kotlin.

在 JavaScript 中使用 Kotlin 代码

根据所选的 JavaScript 模块系统，Kotlin/JS 编译器会生成不同的输出。 当然通常 Kotlin 编译器生成正常的 JavaScript 类，可以在 JavaScript 代码中自由地使用的函数和属性。不过，应该记住一些微妙的事情。

在 plain 模式中用独立的 JavaScript 隔离声明

如果将模块种类明确设置为 `plain`，为了防止损坏全局对象， Kotlin 创建一个包含当前模块中所有 Kotlin 声明的对象。这意味着对于一个模块 `myModule`， 所有的声明都可以通过 `myModule` 对象在 JavaScript 中使用。例如：

```
fun foo() = "Hello"
```

可以在 JavaScript 中这样调用：

```
alert(myModule.foo());
```

将 Kotlin 模块编译为 JavaScript 模块〔例如 UMD，（这是 `browser` 与 `nodejs` 目标的默认设置）、CommonJS 或 AMD〕时，此方法不适用。在这种情况下，声明将以选择的 JavaScript 模块系统指定的格式暴露。例如，当使用 UMD 或 CommonJS 时，调用处可能如下所示：

```
alert(require('myModule').foo());
```

查看有关 JavaScript 模块的文章，以获取有关 JavaScript 模块系统专题的更多信息。

包结构

Kotlin 将其包结构暴露给 JavaScript，因此除非你在根包中定义声明， 否则必须在 JavaScript 中使用完整限定名。例如：

```
package my.qualified.packagename

fun foo() = "Hello"
```

例如，当使用 UMD 或 CommonJS 时，调用处可能如下所示：

```
alert(require('myModule').my.qualified.packagename.foo())
```

或者，在使用 `plain` 格式作为模块系统设置的情况下：

```
alert(myModule.my.qualified.packagename.foo());
```

@JsName 注解

在某些情况下（例如为了支持重载），Kotlin 编译器会修饰（mangle）JavaScript 代码中生成的函数和属性的名称。要控制生成的名称，可以使用 `@JsName` 注解：

```
// 模块“kjs”
class Person(val name: String) {
    fun hello() {
        println("Hello $name!")
    }

    @JsName("helloWithGreeting")
    fun hello(greeting: String) {
        println("$greeting $name!")
    }
}
```

现在，你可以通过以下方式在 JavaScript 中使用这个类：

```
// 如有必要，根据所选模块系统导入“kjs”
var person = new kjs.Person("Dmitry");   // 引用到模块“kjs”
person.hello();                          // 输出“Hello Dmitry!”
person.helloWithGreeting("Servus");      // 输出“Servus Dmitry!”
```

如果我们没有指定 `@JsName` 注解，相应函数的名称会包含从函数签名计算而来的后缀，例如 `hello_61zpoe$`。

请注意，在某些情况下，Kotlin 编译器不应用修饰：

- `external` 声明不会被修饰
- 从 `external` 类继承的非 `external` 类中的任何重写函数都不会被修饰。

`@JsName` 的参数需要是一个常量字符串字面值，该字面值是一个有效的标识符。 任何尝试将非标识符字符串传递给 `@JsName` 时，编译器都会报错。 以下示例会产生编译期错误：

```
@JsName("new C()")   // 此处出错
external fun newC()
```

@JsExport 注解

> This feature is Experimental. 其设计可能会在将来的版本中更改。
>
> 

通过将 `@JsExport` 注解应用于顶级声明（如类或函数），可以从 JavaScript 使用 Kotlin 声明。注解会导出所有嵌套声明，并使用 Kotlin 中给出的名称。 也可以使用 `@file:JsExport` 将其应用于文件级。

要解决导出中的歧义（例如，具有相同名称的函数的重载），可以将 `@JsExport` 批注与 `@JsName` 一起使用，以指定生成与导出函数的名称。

在当前的 IR 编译器后端中， the `@JsExport` annotation is the only way to make your functions visible from Kotlin.

对于多平台项目， `@JsExport` 也可以在公共代码中使用。它仅在针对 JavaScript 目标进行编译时才有效，并且还允许导出非平台特有的 Kotlin 声明。

@JsStatic

> This feature is Experimental. It may be dropped or changed at any time. Use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack.
>
> 

The `@JsStatic` annotation instructs the compiler to generate additional static methods for the target declaration. This helps you use static members from your Kotlin code directly in JavaScript.

You can apply the `@JsStatic` annotation to functions defined in named objects, as well as in companion objects declared inside classes and interfaces. If you use this annotation, the compiler will generate both a static method of the object and an instance method in the object itself. For example:

```
// Kotlin
class C {
    companion object {
        @JsStatic
        fun callStatic() {}
        fun callNonStatic() {}
    }
}
```

Now, the `callStatic()` function is static in JavaScript while the `callNonStatic()` function is not:

```
// JavaScript
C.callStatic();              // Works, accessing the static function
C.callNonStatic();           // Error, not a static function in the generated JavaScri
```

```
pt
C.Companion.callStatic();    // Instance method remains
C.Companion.callNonStatic(); // The only way it works
```

It's also possible to apply the `@JsStatic` annotation to a property of an object or a companion object, making its getter and setter methods static members in that object or the class containing the companion object.

JavaScript 中的 Kotlin 类型

See how Kotlin types are mapped to JavaScript ones:

Kotlin	JavaScript	Comments
`Byte`, `Short`, `Int`, `Float`, `Double`	`Number`	
`Char`	`Number`	The number represents the character's code.
`Long`	Not supported	There is no 64-bit integer number type in JavaScript, so it is emulated by a Kotlin class.
`Boolean`	`Boolean`	
`String`	`String`	
`Array`	`Array`	
`ByteArray`	`Int8Array`	
`ShortArray`	`Int16Array`	
`IntArray`	`Int32Array`	
`CharArray`	`UInt16Array`	Carries the property `$type$ == "CharArray"`.
`FloatArray`	`Float32Array`	
`DoubleArray`	`Float64Array`	
`LongArray`	`Array<kotlin.Long>`	Carries the property `$type$ == "LongArray"`. Also see Kotlin's Long type comment.
`BooleanArray`	`Int8Array`	Carries the property `$type$ == "BooleanArray"`.
`List`, `MutableList`	`KtList`, `KtMutableList`	Exposes an `Array` via `KtList.asJsReadonlyArrayView` or `KtMutableList.asJsArrayView`.

`Map`, `MutableMap`	`KtMap`, `KtMutableMap`	Exposes an ES2015 `Map` via `KtMap.asJsReadonlyMapView` or `KtMutableMap.asJsMapView`.	
`Set`, `MutableSet`	`KtSet`, `KtMutableSet`	Exposes an ES2015 `Set` via `KtSet.asJsReadonlySetView` or `KtMutableSet.asJsSetView`.	
`Unit`	Undefined	Exportable when used as return type, but not when used as parameter type.	
`Any`	`Object`		
`Throwable`	`Error`		
Nullable `Type?`	`Type	null	undefined`	
All other Kotlin types (except for those marked with `JsExport` annotation)	Not supported	Includes Kotlin's unsigned integer types.	

Additionally, it is important to know that:

- Kotlin preserves overflow semantics for `kotlin.Int`, `kotlin.Byte`, `kotlin.Short`, `kotlin.Char` and `kotlin.Long`.
- Kotlin cannot distinguish between numeric types at runtime (except for `kotlin.Long`), so the following code works:

```
fun f() {
    val x: Int = 23
    val y: Any = x
    println(y as Float)
}
```

- Kotlin 在 JavaScript 中保留了惰性对象初始化。
- Kotlin 不会在 JavaScript 中实现顶层属性的惰性初始化。

JavaScript 模块

可以将 Kotlin 项目编译为适用于各种流行模块系统的 JavaScript 模块。我们目前支持以下 JavaScript 模块配置：

- 统一模块定义（UMD，Unified Module Definitions），它与 *AMD* 和 *CommonJS* 兼容。UMD 模块也可以在不导入或没有模块系统的情况下执行。这是 `browser` 与 `nodejs` 目标的默认选项。
- 异步模块定义（AMD，Asynchronous Module Definition），它尤其为 RequireJS 库所使用。
- CommonJS，广泛用于 Node.js/npm（ `require` 函数和 `module.exports` 对象）。
- 无模块（Plain）。不为任何模块系统编译。可以在全局作用域中以其名称访问模块。

浏览器目标

如果打算在 web 浏览器环境中运行代码并希望使用 UMD 之外的模块系统，那么可以在 `webpackTask` 配置块中指定所需的模块类型。例如，要切换到 CommonJS，请使用：

```
kotlin {
    js {
        browser {
            webpackTask {
                output.libraryTarget = "commonjs2"
            }
        }
        binaries.executable()
    }
}
```

Webpack 提供了 `commonjs` 与 `commonjs2` 这两种不同的 CommonJS“风味”，它们影响声明的可用方式。在大多数情况下，可能希望使用 `commonjs2`，该模块将 `module.exports` 语法添加到所生成的库中。或者，也可以选择 `commonjs` 选项，严格遵守 CommonJS 规范。 如需了解有关 `commonjs` 与 `commonjs2` 之间的区别的更多信息，请参见 Webpack 版本库查看。

JavaScript 库与 Node.js 文件

如果要创建一个用于 JavaScript 或 Node.js 环境的库，并且希望使用不同的模块系统，那么其说明会略有不同。

选择目标模块系统

如需选择目标模块系统，请在 Gradle 构建脚本中设置 `moduleKind` 编译器选项。

> 【Kotlin】

```
tasks.withType<org.jetbrains.kotlin.gradle.targets.js.ir.KotlinJsIrLink> {
    compilerOptions.moduleKind.set(org.jetbrains.kotlin.gradle.dsl.JsModuleKind.MODULE
_COMMONJS)
}
```

> 【Groovy】

```
compileKotlinJs.compilerOptions.moduleKind = org.jetbrains.kotlin.gradle.dsl.JsModuleK
ind.MODULE_COMMONJS
```

可用值为：`umd`（默认）、`commonjs`、`amd`、`plain`。

> 这与调整 `webpackTask.output.libraryTarget` 不同。库目标更改了 *webpack* 生成的 输出（在代码已编译之后）。`compilerOptions.moduleKind` 更改 由 *Kotlin* 编译器 生成的输出。



在 Kotlin Gradle DSL 中，还有一个用于设置 CommonJS 模块种类的快捷方式：

```
kotlin {
    js {
        useCommonJs()
        // ...
    }
}
```

@JsModule 注解

要告诉 Kotlin 一个 `external` 类、 包、 函数或者属性是一个 JavaScript 模块，你可以使用 `@JsModule` 注解。考虑你有以下 CommonJS 模块叫“hello”：

```
module.exports.sayHello = function (name) { alert("Hello, " + name); }
```

你应该在 Kotlin 中这样声明：

```
@JsModule("hello")
external fun sayHello(name: String)
```

将 @JsModule 应用到包

一些 JavaScript 库导出包（命名空间）而不是函数和类。 从 JavaScript 角度讲，它是一个具有一些成员的对象，这些成员是类、函数和属性。 将这些包作为 Kotlin 对象导入通常看起来不自然。 编译器可以使用以下助记符将导入的 JavaScript 包映射到 Kotlin 包：

```
@file:JsModule("extModule")

package ext.jspackage.name

external fun foo()

external class C
```

其中相应的 JavaScript 模块的声明如下：

```
module.exports = {
  foo: { /* 此处一些代码 */ },
  C: { /* 此处一些代码 */ }
}
```

标有 `@file:JsModule` 注解的文件无法声明非外部成员。 下面的示例会产生编译期错误：

```
@file:JsModule("extModule")

package ext.jspackage.name

external fun foo()

fun bar() = "!" + foo() + "!" // 此处报错
```

导入更深的包层次结构

在前文示例中，JavaScript 模块导出单个包。 但是，一些 JavaScript 库会从模块中导出多个包。 Kotlin 也支持这种场景，尽管你必须为每个导入的包声明一个新的 `.kt` 文件。

例如，让示例更复杂一些：

```
module.exports = {
  mylib: {
    pkg1: {
      foo: function () { /* 此处一些代码 */ },
      bar: function () { /* 此处一些代码 */ }
    },
    pkg2: {
      baz: function () { /* 此处一些代码 */ }
    }
  }
}
```

要在 Kotlin 中导入该模块，你必须编写两个 Kotlin 源文件：

```
@file:JsModule("extModule")
@file:JsQualifier("mylib.pkg1")

package extlib.pkg1

external fun foo()

external fun bar()
```

以及

```
@file:JsModule("extModule")
@file:JsQualifier("mylib.pkg2")

package extlib.pkg2

external fun baz()
```

@JsNonModule 注解

当一个声明标有 `@JsModule` 、当你并不把它编译到一个 JavaScript 模块时，你不能在 Kotlin 代码中使用它。 通常，开发人员将他们的库既作为 JavaScript 模块也作为可下载的 `.js` 文件分发， 可以将这些文件复制到项目的静态资源，并通过 `<script>` 标签包含。 如需告诉 Kotlin，可以在非模块环境中使用一个 `@JsModule` 声明，请添加 `@JsNonModule` 注解。例如，考虑以下 JavaScript 代码：

```
function topLevelSayHello (name) { alert("Hello, " + name); }

if (module && module.exports) {
  module.exports = topLevelSayHello;
}
```

在 Kotlin 中可以这样描述：

```
@JsModule("hello")
@JsNonModule
@JsName("topLevelSayHello")
external fun sayHello(name: String)
```

Kotlin 标准库使用的模块系统

Kotlin 以 Kotlin/JS 标准库作为单个文件分发，该文件本身被编译为 UMD 模块， 因此可以使用上述任何模块系统。在大多数 Kotlin/JS 使用场景中，建议对 `kotlin-stdlib-js` 使用 Gradle 依赖项, 它也在 NPM 上作为 `kotlin` 包提供。

Kotlin/JS 反射

Kotlin/JS provides a limited support for the Kotlin reflection API. The only supported parts of the API are:

- Class references (`::class`)
- `KType` and `typeof()`
- `KClass` and `createInstance()`

Class references

The `::class` syntax returns a reference to the class of an instance, or the class corresponding to the given type. In Kotlin/JS, the value of a `::class` expression is a stripped-down KClass implementation that supports only:

- simpleName and isInstance() members.
- cast() and safeCast() extension functions.

除此之外，你可以使用 KClass.js 访问与 JsClass 类对应的实例。 该 `JsClass` 实例本身就是对构造函数的引用。 这可以用于与期望构造函数的引用的 JS 函数进行互操作。

KType and typeOf()

The `typeof()` function constructs an instance of `KType` for a given type. The `KType` API is fully supported in Kotlin/JS except for Java-specific parts.

KClass and createInstance()

The `createInstance()` function from the KClass interface creates a new instance of the specified class, which is useful for getting the runtime reference to a Kotlin class.

Example

Here is an example of the reflection usage in Kotlin/JS.

```
open class Shape
class Rectangle : Shape()
```

```
inline fun <reified T> accessReifiedTypeArg() =
    println(typeOf<T>().toString())

fun main() {
    val s = Shape()
    val r = Rectangle()

    println(r::class.simpleName) // Prints "Rectangle"
    println(Shape::class.simpleName) // Prints "Shape"
    println(Shape::class.js.name) // Prints "Shape"

    println(Shape::class.isInstance(r)) // Prints "true"
    println(Rectangle::class.isInstance(s)) // Prints "false"
    val rShape = Shape::class.cast(r) // Casts a Rectangle "r" to Shape

    accessReifiedTypeArg<Rectangle>() // Accesses the type via typeOf(). Prints "Recta
ngle"
}
```

类型安全的 HTML DSL

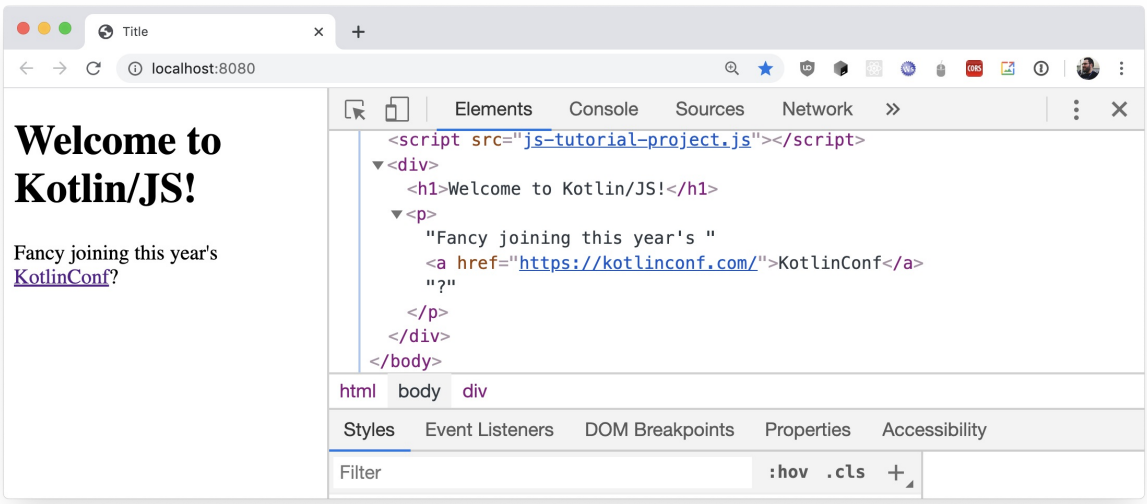
The kotlinx.html library provides the ability to generate DOM elements using statically typed HTML builders (and besides JavaScript, it is even available on the JVM target!) To use the library, include the corresponding repository and dependency to our `build.gradle.kts` file:

```
repositories {
    // ...
    mavenCentral()
}

dependencies {
    implementation(kotlin("stdlib-js"))
    implementation("org.jetbrains.kotlinx:kotlinx-html-js:0.8.0")
    // ...
}
```

Once the dependency is included, you can access the different interfaces provided to generate the DOM. To render a headline, some text, and a link, the following snippet would be sufficient, for example:

```
import kotlinx.browser.*
import kotlinx.html.*
import kotlinx.html.dom.*

fun main() {
    document.body!!.append.div {
        h1 {
            +"Welcome to Kotlin/JS!"
        }
        p {
            +"Fancy joining this year's "
            a("https://kotlinconf.com/") {
                +"KotlinConf"
            }
            +"?"
        }
    }
}
```

When running this example in the browser, the DOM will be assembled in a straightforward way. This is easily confirmed by checking the Elements of the website using the developer tools of our browser:



To learn more about the `kotlinx.html` library, check out the GitHub Wiki, where you can find more information about how to create elements without adding them to the DOM, binding to events like `onClick`, and examples on how to apply CSS classes to your HTML elements, to name just a few.

脚本

- Kotlin 自定义脚本入门——教程

Kotlin 自定义脚本入门——教程

> Kotlin custom scripting is Experimental. It may be dropped or changed at any time. Use it only for evaluation purposes. We appreciate your feedback on it in YouTrack.
>
> 

Kotlin scripting is the technology that enables executing Kotlin code as scripts without prior compilation or packaging into executables.

For an overview of Kotlin scripting with examples, check out the talk Implementing the Gradle Kotlin DSL by Rodrigo Oliveira from KotlinConf'19.

In this tutorial, you'll create a Kotlin scripting project that executes arbitrary Kotlin code with Maven dependencies. You'll be able to execute scripts like this:

```
@file:Repository("https://maven.pkg.jetbrains.space/public/p/kotlinx-html/maven")
@file:DependsOn("org.jetbrains.kotlinx:kotlinx-html-jvm:0.7.3")

import kotlinx.html.*
import kotlinx.html.stream.*
import kotlinx.html.attributes.*

val addressee = "World"

print(
    createHTML().html {
        body {
            h1 { +"Hello, $addressee!" }
        }
    }
)
```

The specified Maven dependency (`kotlinx-html-jvm` for this example) will be resolved from the specified Maven repository or local cache during execution and used for the rest of the script.

Project structure

A minimal Kotlin custom scripting project contains two parts:

- *Script definition* – a set of parameters and configurations that define how this script type should be recognized, handled, compiled, and executed.
- *Scripting host* – an application or component that handles script compilation and execution –

actually running scripts of this type.

With all of this in mind, it's best to split the project into two modules.

Before you start

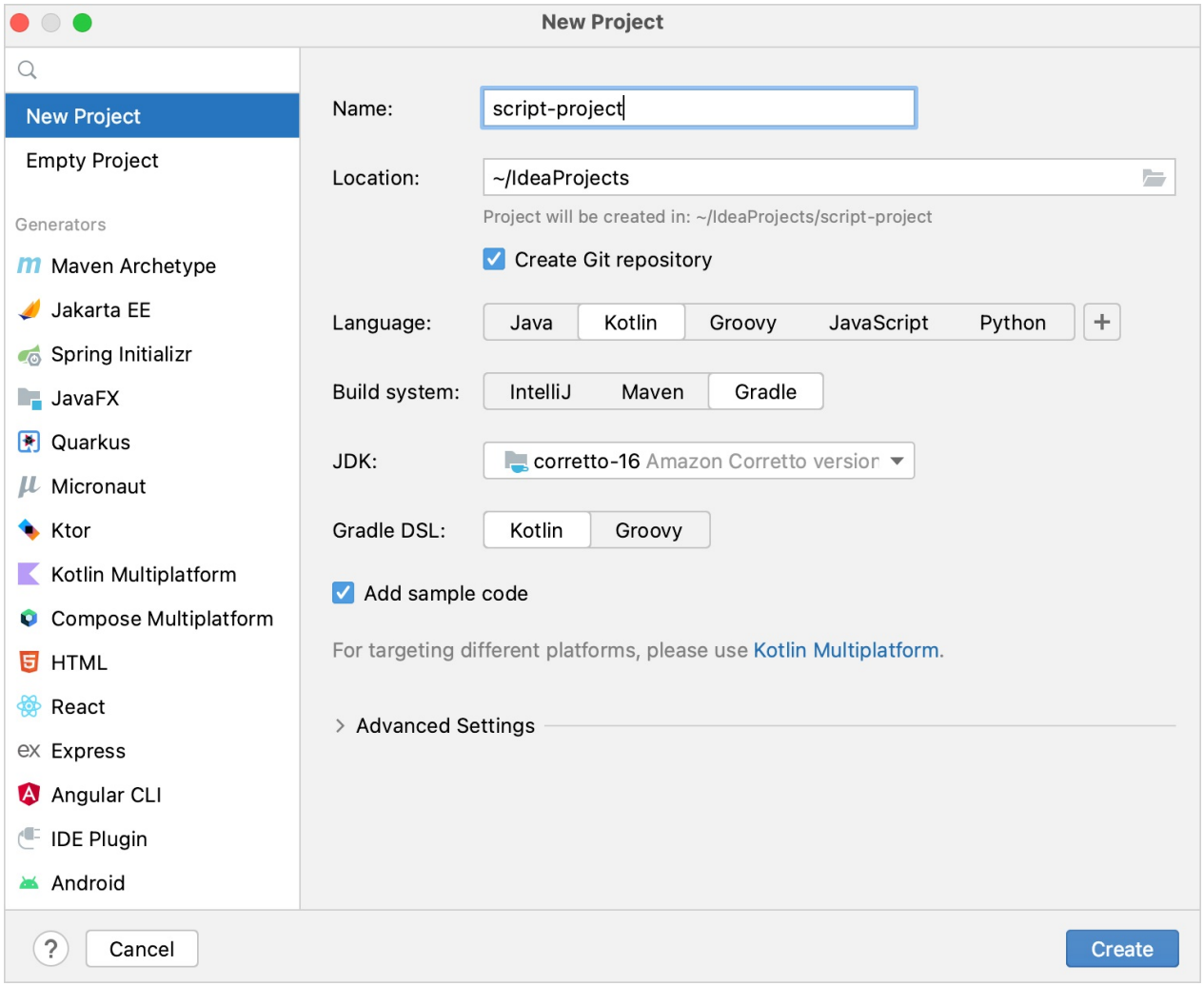
Download and install the latest version of IntelliJ IDEA.

Create a project

1. In IntelliJ IDEA, select **File** | **New** | **Project**.
2. In the panel on the left, select **New Project**.
3. Name the new project and change its location if necessary.

   > Select the **Create Git repository** checkbox to place the new project under version control. You will be able to do it later at any time.
   >
   > 

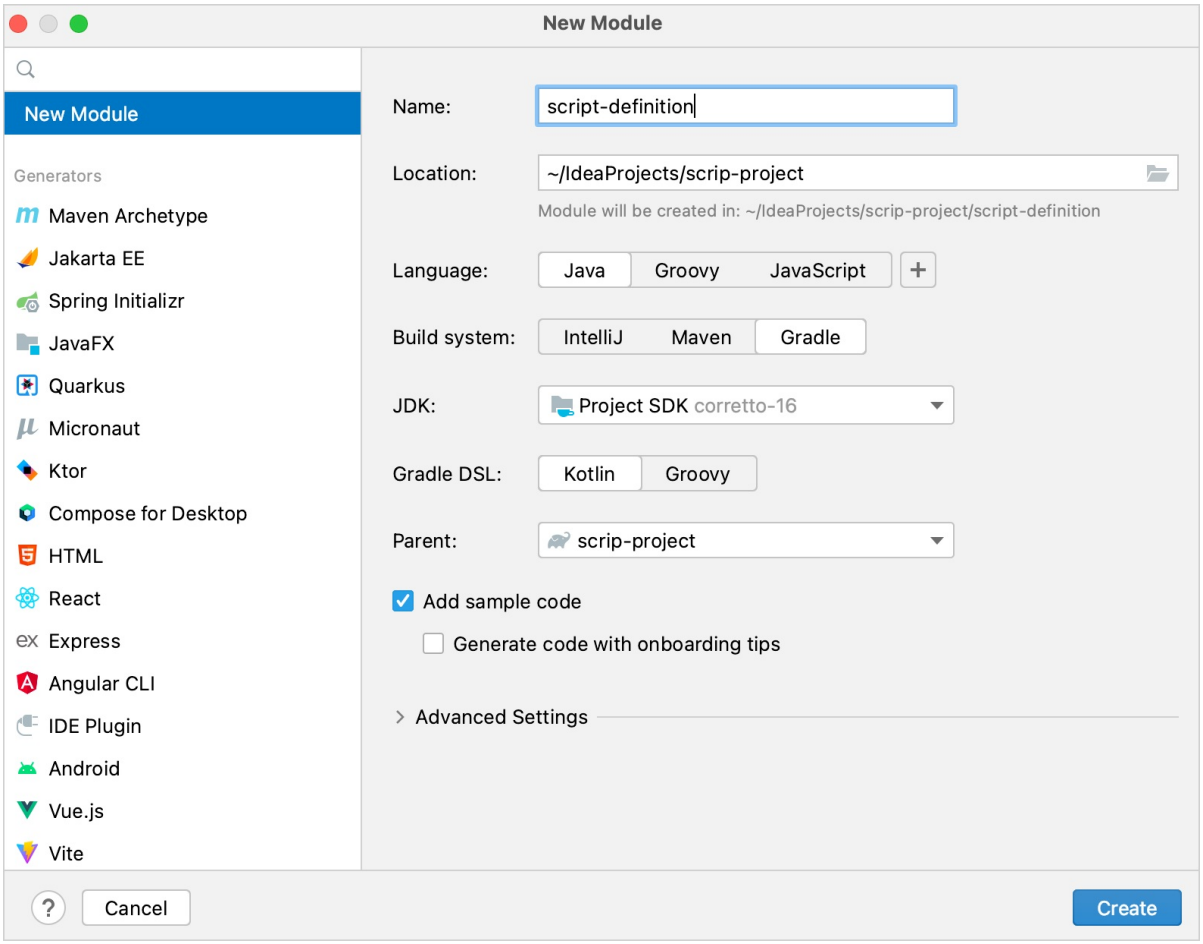
4. From the **Language** list, select **Kotlin**.
5. Select the **Gradle** build system.
6. From the **JDK** list, select the JDK that you want to use in your project.
   - If the JDK is installed on your computer, but not defined in the IDE, select **Add JDK** and specify the path to the JDK home directory.
   - If you don't have the necessary JDK on your computer, select **Download JDK**.
7. Select the Kotlin or Gradle language for the **Gradle DSL**.
8. Click **Create**.



Add scripting modules

Now you have an empty Kotlin/JVM Gradle project. Add the required modules, script definition and scripting host:

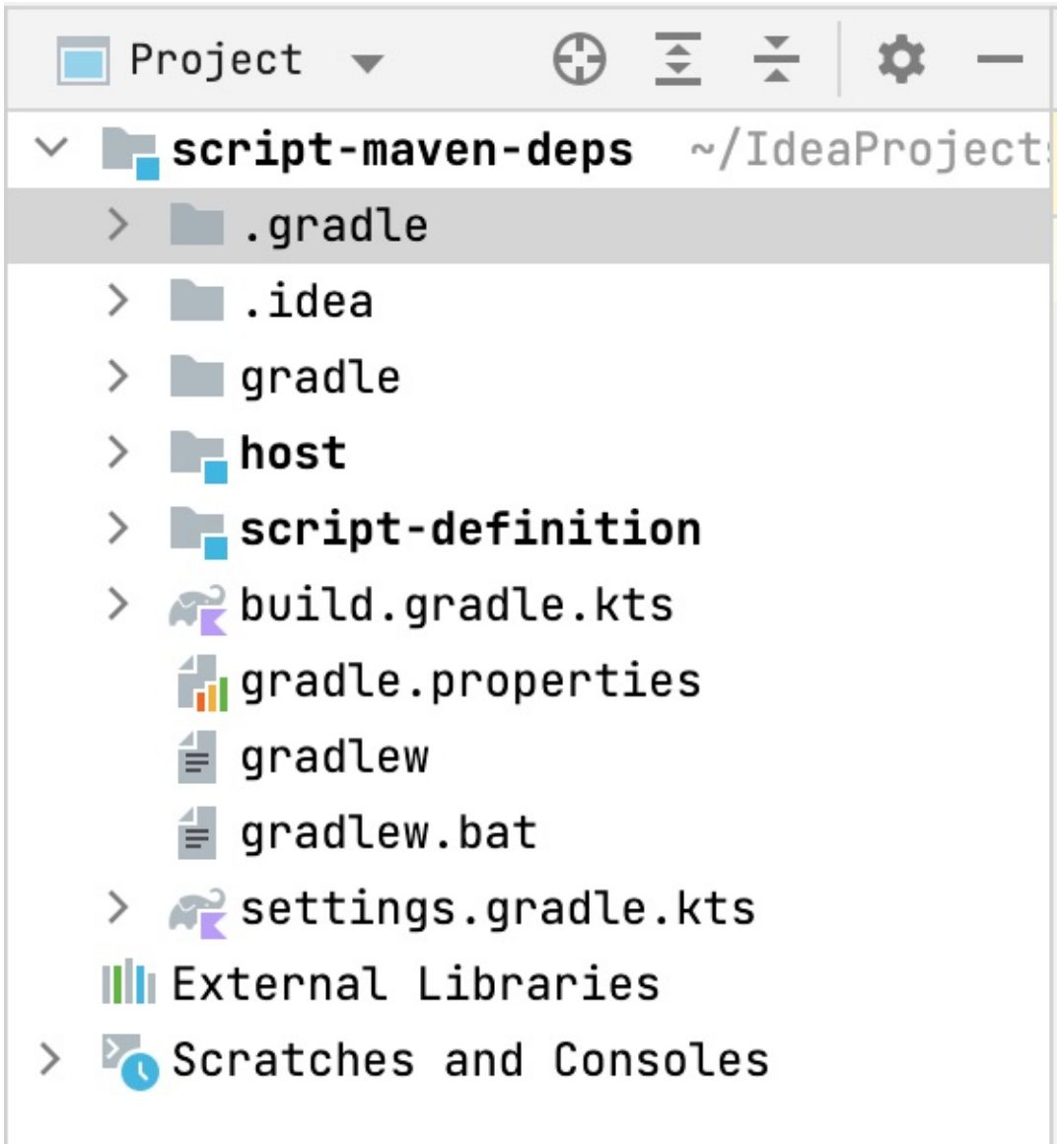
1. In IntelliJ IDEA, select **File | New | Module**.
2. In the panel on the left, select **New Module**. This module will be the script definition.
3. Name the new module and change its location if necessary.
4. From the **Language** list, select **Java**.
5. Select the **Gradle** build system and Kotlin for the **Gradle DSL** if you want to write the build script in Kotlin.
6. As a module's parent, select the root module.
7. Click **Create**.



8. In the module's `build.gradle(.kts)` file, remove the `version` of the Kotlin Gradle plugin. It is already in the root project's build script.

9. Repeat previous steps one more time to create a module for the scripting host.

The project should have the following structure:



You can find an example of such a project and more Kotlin scripting examples in the kotlin-script-examples GitHub repository.

Create a script definition

First, define the script type: what developers can write in scripts of this type and how it will be handled. In this tutorial, this includes support for the `@Repository` and `@DependsOn` annotations in the scripts.

1. In the script definition module, add the dependencies on the Kotlin scripting components in the `dependencies` block of `build.gradle(.kts)`. These dependencies provide the APIs you will need for the script definition:

> 【Kotlin】

```
dependencies {
    implementation("org.jetbrains.kotlin:kotlin-scripting-common")
    implementation("org.jetbrains.kotlin:kotlin-scripting-jvm")
    implementation("org.jetbrains.kotlin:kotlin-scripting-dependencies")
    implementation("org.jetbrains.kotlin:kotlin-scripting-dependencies-maven")
    // coroutines dependency is required for this particular definition
    implementation("org.jetbrains.kotlinx:kotlinx-coroutines-core:1.10.2")
}
```

> 【Groovy】

```
dependencies {
    implementation 'org.jetbrains.kotlin:kotlin-scripting-common'
    implementation 'org.jetbrains.kotlin:kotlin-scripting-jvm'
    implementation 'org.jetbrains.kotlin:kotlin-scripting-dependencies'
    implementation 'org.jetbrains.kotlin:kotlin-scripting-dependencies-maven'
    // coroutines dependency is required for this particular definition
    implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-core:1.10.2'

}
```

---

1. Create the `src/main/kotlin/` directory in the module and add a Kotlin source file, for example, `scriptDef.kt`.
2. In `scriptDef.kt`, create a class. It will be a superclass for scripts of this type, so declare it `abstract` or `open`.

   ```
   // abstract (or open) superclass for scripts of this type
   abstract class ScriptWithMavenDeps
   ```

   This class will also serve as a reference to the script definition later.
3. To make the class a script definition, mark it with the `@KotlinScript` annotation. Pass two parameters to the annotation:
   - `fileExtension` – a string ending with `.kts` that defines a file extension for scripts of this type.
   - `compilationConfiguration` – a Kotlin class that extends `ScriptCompilationConfiguration` and defines the compilation specifics for this script definition. You'll create it in the next step.

   ```
   // @KotlinScript annotation marks a script definition class
   @KotlinScript(
   ```

```
    // File extension for the script type
    fileExtension = "scriptwithdeps.kts",
    // Compilation configuration for the script type
    compilationConfiguration = ScriptWithMavenDepsConfiguration::class
)
abstract class ScriptWithMavenDeps

object ScriptWithMavenDepsConfiguration: ScriptCompilationConfiguration()
```

> In this tutorial, we provide only the working code without explaining Kotlin scripting API. You can find the same code with a detailed explanation on GitHub.



4. Define the script compilation configuration as shown below.

```
object ScriptWithMavenDepsConfiguration : ScriptCompilationConfiguration(
    {
        // Implicit imports for all scripts of this type
        defaultImports(DependsOn::class, Repository::class)
        jvm {
            // Extract the whole classpath from context classloader and use it as
dependencies
            dependenciesFromCurrentContext(wholeClasspath = true)
        }
        // Callbacks
        refineConfiguration {
            // Process specified annotations with the provided handler
            onAnnotations(DependsOn::class, Repository::class, handler = ::config
ureMavenDepsOnAnnotations)
        }
    }
)
```

The `configureMavenDepsOnAnnotations` function is as follows:

```
// Handler that reconfigures the compilation on the fly
fun configureMavenDepsOnAnnotations(context: ScriptConfigurationRefinementContext)
: ResultWithDiagnostics<ScriptCompilationConfiguration> {
    val annotations = context.collectedData?.get(ScriptCollectedData.collectedAnn
otations)?.takeIf { it.isNotEmpty() }
        ?: return context.compilationConfiguration.asSuccess()
    return runBlocking {
        resolver.resolveFromScriptSourceAnnotations(annotations)
    }.onSuccess {
        context.compilationConfiguration.with {
            dependencies.append(JvmDependency(it))
        }.asSuccess()
    }
}
```

```
 private val resolver = CompoundDependenciesResolver(FileSystemDependenciesResolve
r(), MavenDependenciesResolver())
```

You can find the full code here.

Create a scripting host

The next step is creating the scripting host – the component that handles the script execution.

1. In the scripting host module, add the dependencies in the `dependencies` block of `build.gradle(.kts)`:
   - Kotlin scripting components that provide the APIs you need for the scripting host
   - The script definition module you created previously

> 【Kotlin】

```
dependencies {
    implementation("org.jetbrains.kotlin:kotlin-scripting-common")
    implementation("org.jetbrains.kotlin:kotlin-scripting-jvm")
    implementation("org.jetbrains.kotlin:kotlin-scripting-jvm-host")
    implementation(project(":script-definition")) // the script definition module
}
```

> 【Groovy】

```
dependencies {
    implementation 'org.jetbrains.kotlin:kotlin-scripting-common'
    implementation 'org.jetbrains.kotlin:kotlin-scripting-jvm'
    implementation 'org.jetbrains.kotlin:kotlin-scripting-jvm-host'
    implementation project(':script-definition') // the script definition module
}
```

---

1. Create the `src/main/kotlin/` directory in the module and add a Kotlin source file, for example, `host.kt`.

2. Define the `main` function for the application. In its body, check that it has one argument – the path to the script file – and execute the script. You'll define the script execution in a separate function `evalFile` in the next step. Declare it empty for now.

   `main` can look like this:

```
fun main(vararg args: String) {
    if (args.size != 1) {
        println("usage: <app> <script file>")
    } else {
        val scriptFile = File(args[0])
        println("Executing script $scriptFile")
        evalFile(scriptFile)
    }
}
```

3. Define the script evaluation function. This is where you'll use the script definition. Obtain it by calling `createJvmCompilationConfigurationFromTemplate` with the script definition class as a type parameter. Then call `BasicJvmScriptingHost().eval`, passing it the script code and its compilation configuration. `eval` returns an instance of `ResultWithDiagnostics`, so set it as your function's return type.

```
fun evalFile(scriptFile: File): ResultWithDiagnostics<EvaluationResult> {
    val compilationConfiguration = createJvmCompilationConfigurationFromTemplate<
ScriptWithMavenDeps>()
    return BasicJvmScriptingHost().eval(scriptFile.toScriptSource(), compilationC
onfiguration, null)
}
```

4. Adjust the `main` function to print information about the script execution:

```
fun main(vararg args: String) {
    if (args.size != 1) {
        println("usage: <app> <script file>")
    } else {
        val scriptFile = File(args[0])
        println("Executing script $scriptFile")
        val res = evalFile(scriptFile)
        res.reports.forEach {
            if (it.severity > ScriptDiagnostic.Severity.DEBUG) {
                println(" : ${it.message}" + if (it.exception == null) "" else ":
${it.exception}")
            }
        }
    }
}
```

You can find the full code here

Run scripts

To check how your scripting host works, prepare a script to execute and a run configuration.

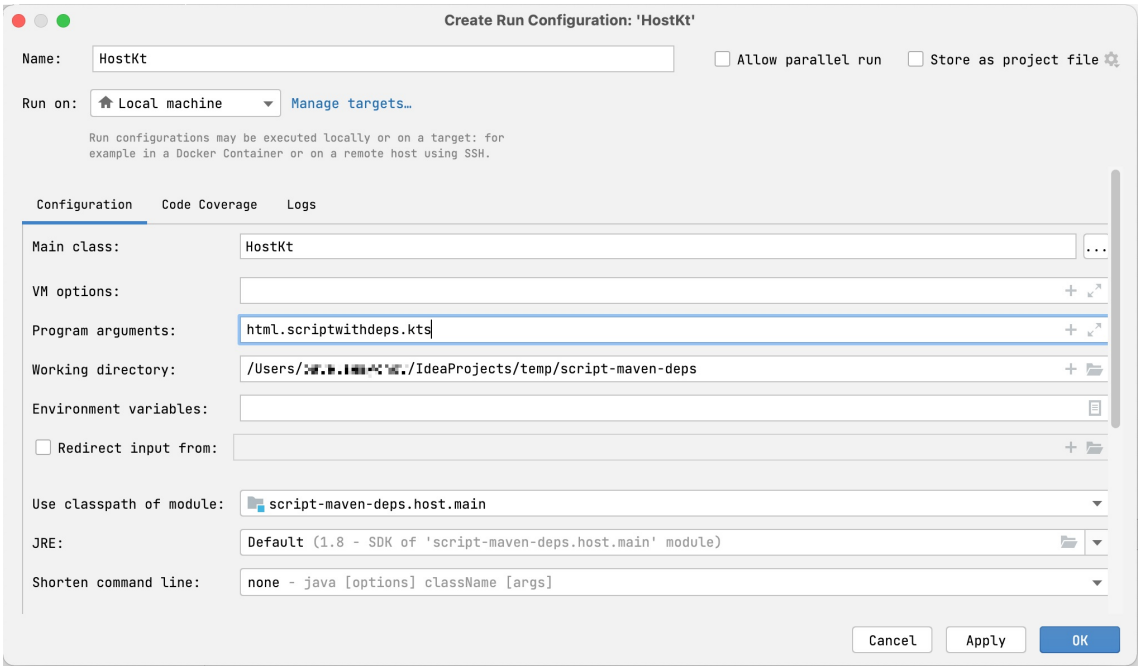
1. Create the file `html.scriptwithdeps.kts` with the following content in the project root directory:

```
@file:Repository("https://maven.pkg.jetbrains.space/public/p/kotlinx-html/maven")
@file:DependsOn("org.jetbrains.kotlinx:kotlinx-html-jvm:0.7.3")

import kotlinx.html.*; import kotlinx.html.stream.*; import kotlinx.html.attributes.*

val addressee = "World"

print(
    createHTML().html {
        body {
            h1 { +"Hello, $addressee!" }
        }
    }
)
```

   It uses functions from the `kotlinx-html-jvm` library which is referenced in the `@DependsOn` annotation argument.

2. Create a run configuration that starts the scripting host and executes this file:

   i. Open `host.kt` and navigate to the `main` function. It has a **Run** gutter icon on the left.
   ii. Right-click the gutter icon and select **Modify Run Configuration**.
   iii. In the **Create Run Configuration** dialog, add the script file name to **Program arguments** and click **OK**.



3. Run the created configuration.

You'll see how the script is executed, resolving the dependency on `kotlinx-html-jvm` in the specified repository and printing the results of calling its functions:

```
<html>
  <body>
    <h1>Hello, World!</h1>
  </body>
</html>
```

Resolving dependencies may take some time on the first run. Subsequent runs will complete much faster because they use downloaded dependencies from the local Maven repository.

下一步做什么？

Once you've created a simple Kotlin scripting project, find more information on this topic:

- Read the Kotlin scripting KEEP
- Browse more Kotlin scripting examples
- Watch the talk Implementing the Gradle Kotlin DSL by Rodrigo Oliveira

Kotlin 多平台开发

Kotlin 多平台开发

标准库

- 集合
  - 集合概述
  - 构造集合
  - 迭代器
  - 区间与数列
  - 序列
  - 集合操作概述
  - 集合转换操作
  - 过滤集合
  - 加减操作符
  - 分组
  - 取集合的一部分
  - 取单个元素
  - 排序
  - 聚合操作
  - 集合写操作
  - List 相关操作
  - Set 相关操作
  - Map 相关操作
- 读取标准输入
- 选择加入要求
- 作用域函数
- 时间度量

集合

- 集合概述
- 构造集合
- 迭代器
- 区间与数列
- 序列
- 集合操作概述
- 集合转换操作
- 过滤集合
- 加减操作符
- 分组
- 取集合的一部分
- 取单个元素
- 排序
- 聚合操作
- 集合写操作
- List 相关操作
- Set 相关操作
- Map 相关操作

集合概述

Kotlin 标准库提供了一整套用于管理集合的工具，集合是可变数量（可能为零）的一组条目，各种集合对于解决问题都具有重要意义，并且经常用到。

集合是大多数编程语言的常见概念，因此如果熟悉像 Java 或者 Python 语言的集合，那么可以跳过这一介绍转到详细部分。

集合通常包含相同类型（及其子类型）的一些对象。集合中的对象称为元素或条目。例如，一个系的所有学生组成一个集合，可以用于计算他们的平均年龄。

以下是 Kotlin 相关的集合类型：

- *List* 是一个有序集合，可通过索引（反映元素位置的整数）访问元素。 元素可以在 list 中出现多次。列表的一个示例是电话号码：有一组数字、这些数字的顺序很重要并且数字可以重复。
- *Set* 是唯一元素的集合。它反映了集合（set）的数学抽象：一组无重复的对象。一般来说 set 中元素的顺序并不重要。例如，the numbers on lottery tickets form a set: they are unique, and their order is not important.
- *Map*（或者字典）是一组键值对。键是唯一的，每个键都刚好映射到一个值。 值可以重复。map 对于存储对象之间的逻辑连接非常有用，例如，员工的 ID 与员工的位置。

Kotlin 让你可以独立于所存储对象的确切类型来操作集合。换句话说，将 `String` 添加到 `String` list 中的方式与添加 `Int` 或者用户自定义类的到相应 list 中的方式相同。 因此，Kotlin 标准库为创建、填充、管理任何类型的集合提供了泛型的（通用的，双关）接口、类与函数。

这些集合接口与相关函数位于 `kotlin.collections` 包中。我们来大致了解下其内容。

> Arrays are not a type of collection. For more information, see Arrays.
>
> 

集合类型

Kotlin 标准库提供了基本集合类型的实现： set、list 以及 map。 一对接口代表每种集合类型：

- 一个 只读 接口，提供访问集合元素的操作。
- 一个 可变 接口，通过写操作扩展相应的只读接口：添加、删除及更新其元素。

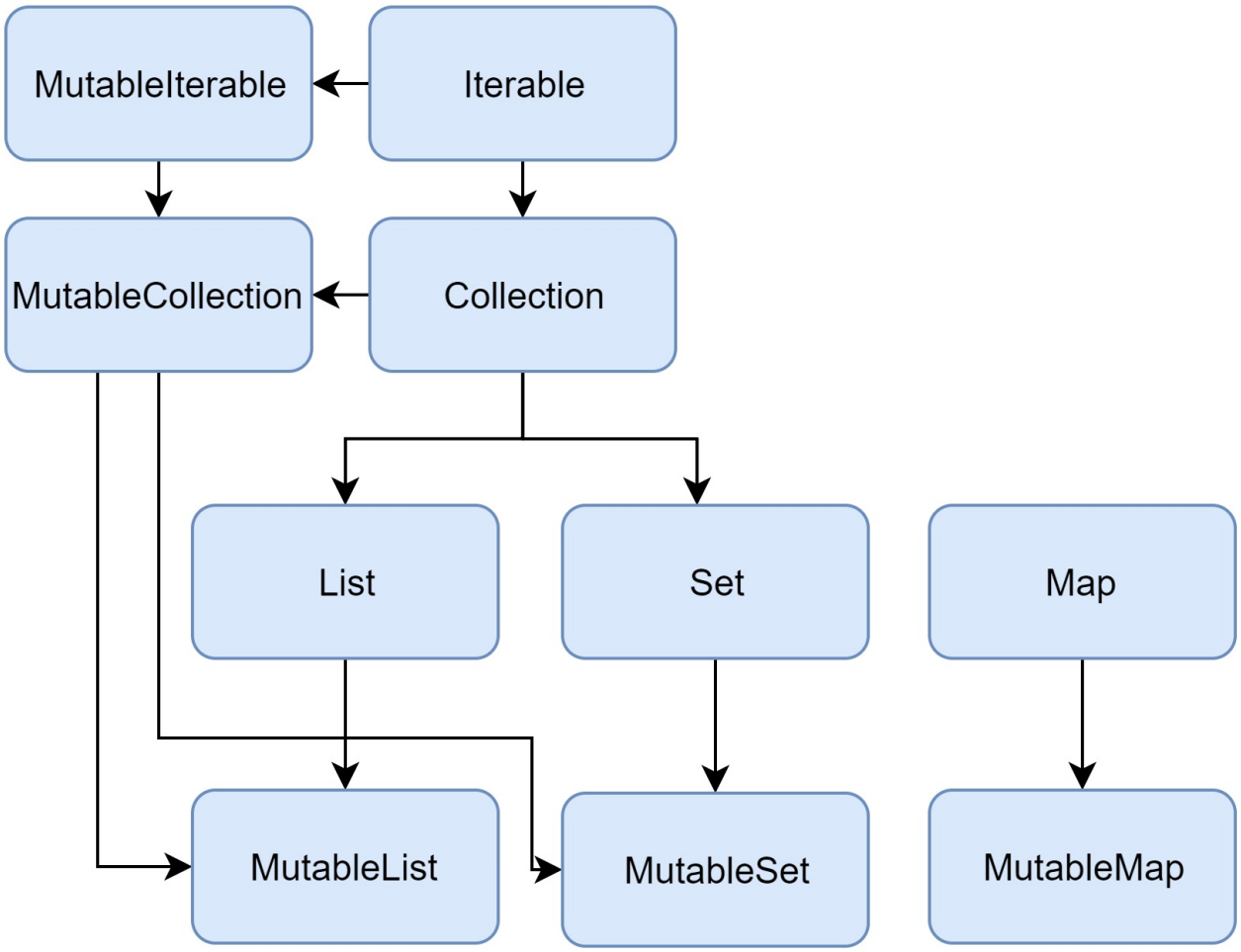
Note that a mutable collection doesn't have to be assigned to a `var`. Write operations with a mutable collection are still possible even if it is assigned to a `val`. The benefit of assigning mutable collections to `val` is that you protect the reference to the mutable collection from modification. Over time, as your code grows and becomes more complex, it becomes even more important to prevent unintentional modification to references. Use `val` as much as possible for safer and more robust code. If you try to reassign a `val` collection, you get a compilation error:

```
fun main() {
//sampleStart
    val numbers = mutableListOf("one", "two", "three", "four")
    numbers.add("five")   // 这是可以的
    println(numbers)
    //numbers = mutableListOf("six", "seven")      // 编译错误
//sampleEnd

}
```

只读集合类型是型变的。 这意味着，如果类 `Rectangle` 继承自 `Shape`，则可以在需要 `List<Shape>` 的任何地方使用 `List<Rectangle>`。 换句话说，集合类型与元素类型具有相同的子类型关系。 map 在值（value）类型上是型变的，但在键（key）类型上不是。

反之，可变集合不是型变的；否则将导致运行时故障。 如果 `MutableList<Rectangle>` 是 `MutableList<Shape>` 的子类型，你可以在其中插入其他 `Shape` 的继承者（例如，`Circle`），从而违反了它的 `Rectangle` 类型参数。

下面是 Kotlin 集合接口的图表：



让我们来看看接口及其实现。 To learn about `Collection`, read the section below. To learn about `List`, `Set`, and `Map`, you can either read the corresponding sections or watch a video by Sebastian Aigner, Kotlin Developer Advocate:

YouTube 视频：Kotlin Collections Overview

Collection

`Collection<T>` 是集合层次结构的根。这个接口表示一个只读集合的共同行为：检索大小、检测是否为成员等等。 `Collection` 继承自 `Iterable <T>` 接口，它定义了迭代元素的操作。可以使用 `Collection` 作为适用于不同集合类型的函数的参数。对于更具体的情况，请使用 `Collection` 的继承者： `List` 与 `Set` 。

```
fun printAll(strings: Collection<String>) {
    for(s in strings) print("$s ")
    println()
}

fun main() {
    val stringList = listOf("one", "two", "one")
    printAll(stringList)

    val stringSet = setOf("one", "two", "three")
    printAll(stringSet)
}
```

`MutableCollection<T>` 是一个具有写操作的 `Collection` 接口，例如 `add` 以及 `remove`。

```
fun List<String>.getShortWordsTo(shortWords: MutableList<String>, maxLength: Int) {
    this.filterTo(shortWords) { it.length <= maxLength }
    // throwing away the articles
    val articles = setOf("a", "A", "an", "An", "the", "The")
    shortWords -= articles
}

fun main() {
    val words = "A long time ago in a galaxy far far away".split(" ")
    val shortWords = mutableListOf<String>()
    words.getShortWordsTo(shortWords, 3)
    println(shortWords)
}
```

List

`List<T>` 以指定的顺序存储元素，并提供使用索引访问元素的方法。索引从 0（第一个元素的索引）开始直到 `lastIndex`（即 `(list.size - 1)`）。

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")
    println("Number of elements: ${numbers.size}")
    println("Third element: ${numbers.get(2)}")
    println("Fourth element: ${numbers[3]}")
    println("Index of element \"two\" ${numbers.indexOf("two")}")
//sampleEnd
}
```

List 元素（包括空值）可以重复：List 可以包含任意数量的相同对象或单个对象的出现。 如果两个 List 在相同的位置具有相同大小和相同结构的元素， 那么认为它们是相等的。

```
data class Person(var name: String, var age: Int)

fun main() {
//sampleStart
    val bob = Person("Bob", 31)
    val people = listOf(Person("Adam", 20), bob, bob)
    val people2 = listOf(Person("Adam", 20), Person("Bob", 31), bob)
    println(people == people2)
    bob.age = 32
    println(people == people2)
//sampleEnd
}
```

`MutableList<T>` 是可以进行写操作的 `List`，例如用于在特定位置添加或删除元素。

```
fun main() {
//sampleStart
    val numbers = mutableListOf(1, 2, 3, 4)
    numbers.add(5)
    numbers.removeAt(1)
    numbers[0] = 0
    numbers.shuffle()
    println(numbers)
//sampleEnd
}
```

如你所见，在某些方面，List 与数组（Array）非常相似。 但是，有一个重要的区别：数组的大小是在初始化时定义的，永远不会改变; 反之，List 没有预定义的大小；作为写操作的结果，可以更改 List 的大小：添加、 更新或删除元素。

在 Kotlin 中， `MutableList` 的默认实现是 `ArrayList`， 可以将其视为可调整大小的数组。

Set

`Set<T>` 存储唯一的元素； 它们的顺序通常是未定义的。 `null` 元素也是唯一的：一个 `Set` 只能包含一个 `null`。 当两个 `set` 具有相同的大小并且对于一个 `set` 中的每个元素都能在另一个 `set` 中存在相同元素，则两个 `set` 相等。

```
fun main() {
//sampleStart
    val numbers = setOf(1, 2, 3, 4)
    println("Number of elements: ${numbers.size}")
    if (numbers.contains(1)) println("1 is in the set")

    val numbersBackwards = setOf(4, 3, 2, 1)
    println("The sets are equal: ${numbers == numbersBackwards}")
//sampleEnd
}
```

`MutableSet` 是一个带有来自 `MutableCollection` 的写操作接口的 `Set`。

`MutableSet` 的默认实现 - `LinkedHashSet` —— 保留元素插入的顺序。 因此，依赖于顺序的函数，例如 `first()` 或 `last()`，会在这些 `set` 上返回可预测的结果。

```
fun main() {
//sampleStart
    val numbers = setOf(1, 2, 3, 4)  // LinkedHashSet is the default implementation
    val numbersBackwards = setOf(4, 3, 2, 1)

    println(numbers.first() == numbersBackwards.first())
    println(numbers.first() == numbersBackwards.last())
//sampleEnd
}
```

另一种实现方式 – `HashSet` —— 不声明元素的顺序，所以在它上面调用这些函数会返回不可预测的结果。但是， `HashSet` 只需要较少的内存来存储相同数量的元素。

Map

`Map<K, V>` 不是 `Collection` 接口的继承者；但是它也是 Kotlin 的一种集合类型。 `Map` 存储键-值 对（或 条目）；键是唯一的，但是不同的键可以与相同的值配对。 `Map` 接口提供特定的函数进行通过键访问值、搜索键和值等操作。

```
fun main() {
//sampleStart
    val numbersMap = mapOf("key1" to 1, "key2" to 2, "key3" to 3, "key4" to 1)

    println("All keys: ${numbersMap.keys}")
    println("All values: ${numbersMap.values}")
    if ("key2" in numbersMap) println("Value by key \"key2\": ${numbersMap["key2"]}")

    if (1 in numbersMap.values) println("The value 1 is in the map")
    if (numbersMap.containsValue(1)) println("The value 1 is in the map") // 同上
//sampleEnd
}
```

无论键值对的顺序如何，包含相同键值对的两个 `Map` 是相等的。

```
fun main() {
//sampleStart
    val numbersMap = mapOf("key1" to 1, "key2" to 2, "key3" to 3, "key4" to 1)
    val anotherMap = mapOf("key2" to 2, "key1" to 1, "key4" to 1, "key3" to 3)

    println("The maps are equal: ${numbersMap == anotherMap}")
```

```
//sampleEnd
}
```

`MutableMap` 是一个具有写操作的 `Map` 接口，可以使用该接口添加一个新的键值对或更新给定键的值。

```
fun main() {
//sampleStart
    val numbersMap = mutableMapOf("one" to 1, "two" to 2)
    numbersMap.put("three", 3)
    numbersMap["one"] = 11

    println(numbersMap)
//sampleEnd
}
```

`MutableMap` 的默认实现 – `LinkedHashMap` —— 迭代 Map 时保留元素插入的顺序。 反之，另一种实现 – `HashMap` —— 不声明元素的顺序。

ArrayDeque

`ArrayDeque<T>` is an implementation of a double-ended queue, which allows you to add or remove elements both at the beginning or end of the queue. As such, `ArrayDeque` also fills the role of both a Stack and Queue data structure in Kotlin. Behind the scenes, `ArrayDeque` is realized using a resizable array that automatically adjusts in size when required:

```
fun main() {
    val deque = ArrayDeque(listOf(1, 2, 3))

    deque.addFirst(0)
    deque.addLast(4)
    println(deque) // [0, 1, 2, 3, 4]

    println(deque.first()) // 0
    println(deque.last()) // 4

    deque.removeFirst()
    deque.removeLast()
    println(deque) // [1, 2, 3]
}
```

构造集合

由元素构造

创建集合的最常用方法是使用标准库函数 `listOf<T>()`、`setOf<T>()`、`mutableListOf<T>()`、`mutableSetOf<T>()`。如果以逗号分隔的集合元素列表作为参数，编译器会自动检测元素类型。创建空集合时，须明确指定类型。

```
val numbersSet = setOf("one", "two", "three", "four")
val emptySet = mutableSetOf<String>()
```

同样的，Map 也有这样的函数 `mapOf()` 与 `mutableMapOf()`。映射的键和值作为 `Pair` 对象传递（通常使用中缀函数 `to` 创建）。

```
val numbersMap = mapOf("key1" to 1, "key2" to 2, "key3" to 3, "key4" to 1)
```

注意，`to` 符号创建了一个短时存活的 `Pair` 对象，因此建议仅在性能不重要时才使用它。为避免过多的内存使用，请使用其他方法。例如，可以创建可写 Map 并使用写入操作填充它。`apply()` 函数可以帮助保持初始化流畅。

```
val numbersMap = mutableMapOf<String, String>().apply { this["one"] = "1"; this["two"]
 = "2" }
```

Create with collection builder functions

Another way of creating a collection is to call a builder function – `buildList()`, `buildSet()`, or `buildMap()`. They create a new, mutable collection of the corresponding type, populate it using write operations, and return a read-only collection with the same elements:

```
val map = buildMap { // this is MutableMap<String, Int>, types of key and value are in
ferred from the `put()` calls below
    put("a", 1)
    put("b", 0)
    put("c", 4)
}

println(map) // {a=1, b=0, c=4}
```

空集合

还有用于创建没有任何元素的集合的函数：`emptyList()`、`emptySet()` 与 `emptyMap()`。 创建空集合时，应指定集合将包含的元素类型。

```
val empty = emptyList<String>()
```

list 的初始化函数

对于 List，有一个接受 List 的大小与初始化函数的类似构造函数的函数，该初始化函数根据索引定义元素的值。

```
fun main() {
//sampleStart
    val doubled = List(3, { it * 2 })  // 如果你想操作这个集合，应使用 MutableList
    println(doubled)
//sampleEnd
}
```

具体类型构造函数

要创建具体类型的集合，例如 `ArrayList` 或 `LinkedList`，可以使用这些类型的构造函数。类似的构造函数对于 `Set` 与 `Map` 的各实现中均有提供。

```
val linkedList = LinkedList<String>(listOf("one", "two", "three"))
val presizedSet = HashSet<Int>(32)
```

复制

要创建与现有集合具有相同元素的集合，可以使用复制函数。 标准库中的集合复制函数创建了具有相同元素引用的 浅 复制集合。 因此，对集合元素所做的更改会反映在其所有副本中。

在特定时刻通过集合复制函数，例如 `toList()`、`toMutableList()`、`toSet()` 等等。create a snapshot of a collection at a specific moment. 结果是创建了一个具有相同元素的新集合 如果在源代码集合中添加或删除元素，则不会影响副本。副本也可以独立于源代码集合进行更改。

```
class Person(var name: String)
fun main() {
//sampleStart
```

```
    val alice = Person("Alice")
    val sourceList = mutableListOf(alice, Person("Bob"))
    val copyList = sourceList.toList()
    sourceList.add(Person("Charles"))
    alice.name = "Alicia"
    println("First item's name is: ${sourceList[0].name} in source and ${copyList[0].n
ame} in copy")
    println("List size is: ${sourceList.size} in source and ${copyList.size} in copy")
//sampleEnd
}
```

这些函数还可用于将集合转换为其他类型，例如根据 List 构建 Set，反之亦然。

```
fun main() {
//sampleStart
    val sourceList = mutableListOf(1, 2, 3)
    val copySet = sourceList.toMutableSet()
    copySet.add(3)
    copySet.add(4)
    println(copySet)
//sampleEnd
}
```

或者，可以创建对同一集合实例的新引用。使用现有集合初始化集合变量时，将创建新引用。 因此，当通过引用更改集合实例时，更改将反映在其所有引用中。

```
fun main() {
//sampleStart
    val sourceList = mutableListOf(1, 2, 3)
    val referenceList = sourceList
    referenceList.add(4)
    println("Source size: ${sourceList.size}")
//sampleEnd
}
```

集合的初始化可用于限制其可变性。例如，如果构建了一个 `MutableList` 的 `List` 引用，当你试图通过此引用修改集合的时候，编译器会抛出错误。

```
fun main() {
//sampleStart
    val sourceList = mutableListOf(1, 2, 3)
    val referenceList: List<Int> = sourceList
    //referenceList.add(4)            // 编译错误
    sourceList.add(4)
    println(referenceList) // 显示 sourceList 当前状态
//sampleEnd
}
```

调用其他集合的函数

可以通过其他集合各种操作的结果来创建集合。例如，过滤列表会创建与过滤器匹配的新元素列表：

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")
    val longerThan3 = numbers.filter { it.length > 3 }
    println(longerThan3)
//sampleEnd
}
```

映射由转换结果生成列表：

```
fun main() {
//sampleStart
    val numbers = setOf(1, 2, 3)
    println(numbers.map { it * 3 })
    println(numbers.mapIndexed { idx, value -> value * idx })
//sampleEnd
}
```

关联生成 Map:

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")
    println(numbers.associateWith { it.length })
//sampleEnd
}
```

关于 Kotlin 中集合操作的更多信息，参见集合操作概述.

迭代器

对于遍历集合元素， Kotlin 标准库支持 迭代器 的常用机制—— 对象可按顺序提供对元素的访问权限，而不会暴露集合的底层结构。 当需要逐个处理集合的所有元素（例如打印值或对其进行类似更新）时， 迭代器非常有用。

`Iterable<T>` 接口的继承者（包括 `Set` 与 `List` ）可以通过调用 `iterator()` 函数获得迭代器。

一旦获得迭代器它就指向集合的第一个元素；调用 `next()` 函数将返回此元素，并将迭代器指向下一个元素（如果下一个元素存在）。

一旦迭代器通过了最后一个元素，它就不能再用于检索元素；也无法重新指向到以前的任何位置。要再次遍历集合，请创建一个新的迭代器。

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")
    val numbersIterator = numbers.iterator()
    while (numbersIterator.hasNext()) {
        println(numbersIterator.next())
        // one
        // two
        // three
        // four
    }
//sampleEnd
}
```

遍历 `Iterable` 集合的另一种方法是众所周知的 `for` 循环。在集合中使用 `for` 循环时， 会隐式获取迭代器。因此，以下代码与上述示例等效：

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")
    for (item in numbers) {
        println(item)
        // one
        // two
        // three
        // four
    }
//sampleEnd
}
```

最后，有一个好用的 `forEach()` 函数，可自动迭代集合并为每个元素执行给定的代码。因此，等效的示例如下所示：

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")
    numbers.forEach {
        println(it)
        // one
        // two
        // three
        // four
    }
//sampleEnd
}
```

List 迭代器

对于列表，有一个特殊的迭代器实现： `ListIterator` 。 它支持列表双向迭代：正向与反向。

反向迭代由 `hasPrevious()` 与 `previous()` 函数实现。 此外， `ListIterator` 通过 `nextIndex()` 与 `previousIndex()` 函数提供有关元素索引的信息。

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")
    val listIterator = numbers.listIterator()
    while (listIterator.hasNext()) listIterator.next()
    println("Iterating backwards:")
    // Iterating backwards:
    while (listIterator.hasPrevious()) {
        print("Index: ${listIterator.previousIndex()}")
        println(", value: ${listIterator.previous()}")
        // Index: 3, value: four
        // Index: 2, value: three
        // Index: 1, value: two
        // Index: 0, value: one
    }
//sampleEnd
}
```

具有双向迭代的能力意味着 `ListIterator` 在到达最后一个元素后仍可以使用。

可变迭代器

为了迭代可变集合，于是有了 `MutableIterator` 来扩展 `Iterator` 使其具有元素删除函数 `remove()` 。 因此，可以在迭代时从集合中删除元素。

```
fun main() {
//sampleStart
    val numbers = mutableListOf("one", "two", "three", "four")
    val mutableIterator = numbers.iterator()

    mutableIterator.next()
    mutableIterator.remove()
    println("After removal: $numbers")
    // After removal: [two, three, four]
//sampleEnd
}
```

除了删除元素， `MutableListIterator` 还可以在迭代列表时插入和替换元素（使用 `add()` 与 `set()` 函数）。

```
fun main() {
//sampleStart
    val numbers = mutableListOf("one", "four", "four")
    val mutableListIterator = numbers.listIterator()

    mutableListIterator.next()
    mutableListIterator.add("two")
    println(numbers)
    // [one, two, four, four]
    mutableListIterator.next()
    mutableListIterator.set("three")
    println(numbers)
    // [one, two, three, four]
//sampleEnd
}
```

区间与数列

Ranges and progressions define sequences of values in Kotlin, supporting range operators, iteration, custom step values, and arithmetic progressions.

Ranges

Kotlin 可通过调用 `kotlin.ranges` 包中的 `rangeTo()` 及 `.rangeUntil()` 函数轻松地创建两个值的区间。

A range represents an ordered set of values with a defined start and end. By default, it increments by 1 at each step. For example, `1..4` represents the numbers 1, 2, 3, and 4.

To create:

- a closed-ended range, call the `.rangeTo()` function with the `..` operator. This includes both the start and end values.
- an open-ended range, call the `.rangeUntil()` function with the `..<` operator. This includes the start value but excludes the end value.

For example:

```
fun main() {
//sampleStart
    // Closed-ended range: includes both 1 and 4
    println(4 in 1..4)
    // true

    // Open-ended range: includes 1, excludes 4
    println(4 in 1..<4)
    // false
//sampleEnd
}
```

Ranges are particularly useful for iterating over `for` loops:

```
fun main() {
//sampleStart
    for (i in 1..4) print(i)
    // 1234
//sampleEnd
}
```

如需反向迭代数字，请使用 `downTo` 函数取代 `..` 。

```
fun main() {
//sampleStart
    for (i in 4 downTo 1) print(i)
    // 4321
//sampleEnd
}
```

也可以使用 `step()` 函数以自定义步长迭代数字，而不是默认增量 1：

```
fun main() {
//sampleStart
    for (i in 0..8 step 2) print(i)
    println()
    // 02468
    for (i in 0..<8 step 2) print(i)
    println()
    // 0246
    for (i in 8 downTo 0 step 2) print(i)
    // 86420
//sampleEnd
}
```

数列

整数类型（例如 `Int` 、 `Long` 与 `Char` ）的区间可视为等差数列。 在 Kotlin 中，这些数列由特殊类型定义： `IntProgression` 、 `LongProgression` 与 `CharProgression` 。

数列具有三个基本属性： `first` 元素、 `last` 元素和一个非零的 `step` 。 首个元素为 `first` ，后续元素是前一个元素加上一个 `step` 。 以确定的步长在数列上进行迭代等效于 Java/JavaScript 中基于索引的 `for` 循环。

```
for (int i = first; i <= last; i += step) {
  // ……
}
```

通过迭代数列隐式创建区间时，此数列的 `first` 与 `last` 元素是区间的端点， `step` 为 1 。

```
fun main() {
//sampleStart
    for (i in 1..10) print(i)
    // 12345678910
//sampleEnd
}
```

要指定数列步长，请在区间上使用 `step` 函数。

```
fun main() {
//sampleStart
    for (i in 1..8 step 2) print(i)
    // 1357
//sampleEnd
}
```

数列的 `last` 元素是这样计算的：

- 对于正步长：不大于结束值且满足 `(last - first) % step == 0` 的最大值。
- 对于负步长：不小于结束值且满足 `(last - first) % step == 0` 的最小值。

因此，`last` 元素并非总与指定的结束值相同。

```
fun main() {
//sampleStart
    for (i in 1..9 step 3) print(i) // 最后一个元素是 7
    // 147
//sampleEnd
}
```

数列实现 `Iterable<N>`，其中 `N` 分别是 `Int`、`Long` 或 `Char`，因此可以在各种集合函数（如 `map`、`filter` 与其他）中使用它们。

```
fun main() {
//sampleStart
    println((1..10).filter { it % 2 == 0 })
    // [2, 4, 6, 8, 10]
//sampleEnd
}
```

序列

除了集合之外，Kotlin 标准库还包含另一种类型——序列（ `Sequence<T>` ）。 Unlike collections, sequences don't contain elements, they produce them while iterating. 序列提供与 `Iterable` 相同的函数，但实现另一种方法来进行多步骤集合处理。

当 `Iterable` 的处理包含多个步骤时，它们会优先执行：每个处理步骤完成并返回其结果——中间集合。 在此集合上执行以下步骤。反过来， 序列的多步处理在可能的情况下会延迟执行：仅当请求整个处理链的结果时才进行实际计算。

操作执行的顺序也不同： `Sequence` 对每个元素逐个执行所有处理步骤。 反过来， `Iterable` 完成整个集合的每个步骤，然后进行下一步。

因此，这些序列可避免生成中间步骤的结果，从而提高了整个集合处理链的性能。 但是，序列的惰性性质增加了一些开销，这些开销在处理较小的集合或进行更简单的计算时可能很重要。 因此，应该同时考虑使用 `Sequence` 与 `Iterable` ， 并确定在哪种情况更适合。

构造

由元素

要创建一个序列，请调用 `sequenceOf()` 函数，列出元素作为其参数。

```
val numbersSequence = sequenceOf("four", "three", "two", "one")
```

由 Iterable

如果已经有一个 `Iterable` 对象（例如 `List` 或 `Set` ），则可以通过调用 `asSequence()` 从而创建一个序列。

```
val numbers = listOf("one", "two", "three", "four")
val numbersSequence = numbers.asSequence()
```

由函数

创建序列的另一种方法是通过使用计算其元素的函数来构建序列。 要基于函数构建序列，请以该函数作为参数调用 `generateSequence()` 。 （可选）可以将第一个元素指定为显式值或函数调用的结果。 当提供的函数返回 `null` 时，序列生成停止。因此，以下示例中的序列是无限的。

```
fun main() {
//sampleStart
    val oddNumbers = generateSequence(1) { it + 2 } // `it` 是上一个元素
    println(oddNumbers.take(5).toList())
    //println(oddNumbers.count())     // 错误：此序列是无限的。
//sampleEnd
}
```

要使用 `generateSequence()` 创建有限序列，请提供一个函数，该函数在需要的最后一个元素之后返回 `null`。

```
fun main() {
//sampleStart
    val oddNumbersLessThan10 = generateSequence(1) { if (it < 8) it + 2 else null }
    println(oddNumbersLessThan10.count())
//sampleEnd
}
```

由组块

最后，有一个函数可以逐个或按任意大小的组块生成序列元素——`sequence()` 函数。 此函数采用一个 lambda 表达式，其中包含 `yield()` 与 `yieldAll()` 函数的调用。 它们将一个元素返回给序列使用者，并暂停 `sequence()` 的执行，直到使用者请求下一个元素。 `yield()` 使用单个元素作为参数；`yieldAll()` 中可以采用 `Iterable` 对象、 `Iterator` 或其他 `Sequence`。 `yieldAll()` 的 `Sequence` 参数可以是无限的。 当然，这样的调用必须是最后一个： 之后的所有调用都永远不会执行。

```
fun main() {
//sampleStart
    val oddNumbers = sequence {
        yield(1)
        yieldAll(listOf(3, 5))
        yieldAll(generateSequence(7) { it + 2 })
    }
    println(oddNumbers.take(5).toList())
//sampleEnd
}
```

序列操作

关于序列操作，根据其状态要求可以分为以下几类：

- 无状态 操作不需要状态，并且可以独立处理每个元素，例如 `map()` 或 `filter()`。 无状态操作还可能需要少量常数个状态来处理元素，例如 `take()` 与 `drop()`。
- 有状态 操作需要大量状态，通常与序列中元素的数量成比例。

如果序列操作返回延迟生成的另一个序列，则称为 中间序列。 否则，该操作为 末端 操作。 末端操作的示例为 `toList()` 或 `sum()`。只能通过末端操作才能检索序列元素。

序列可以多次迭代；但是，某些序列实现可能会约束自己仅迭代一次。其文档中特别提到了这一点。

序列处理示例

我们通过一个示例来看 `Iterable` 与 `Sequence` 之间的区别。

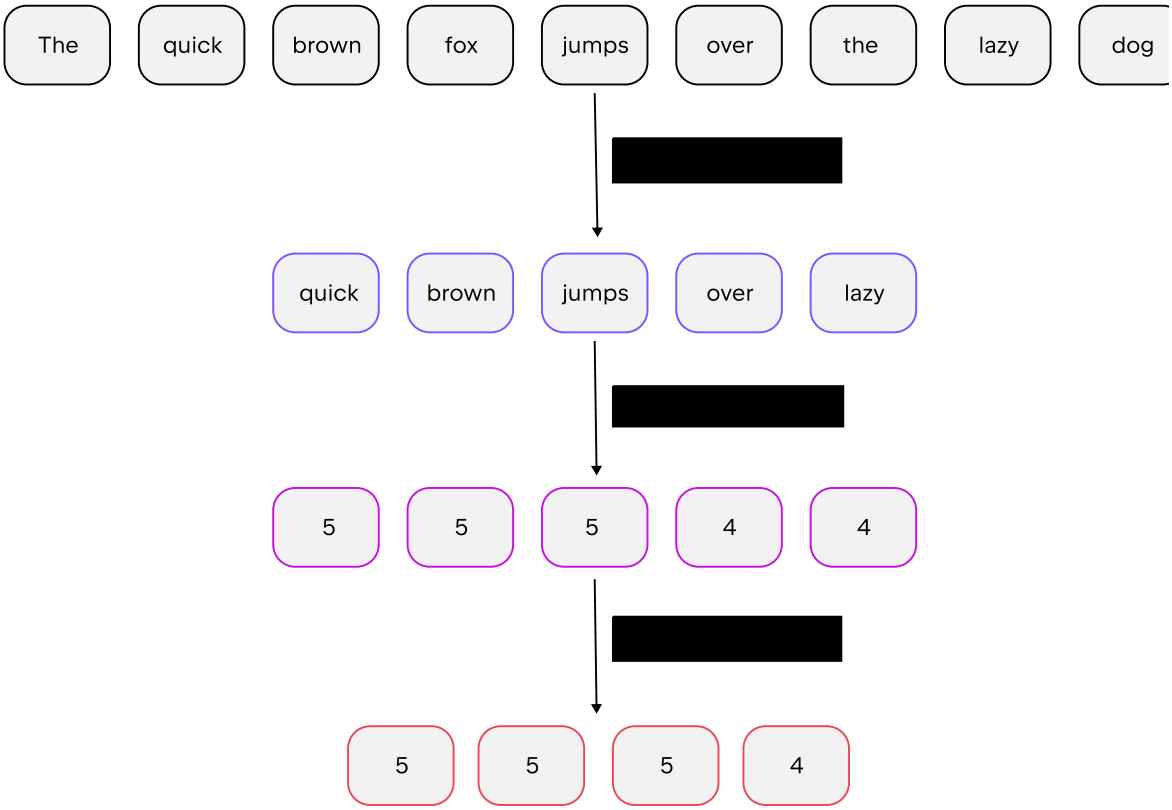
Iterable

假定有一个单词列表。下面的代码过滤长于三个字符的单词，并输出前四个单词的长度。

```
fun main() {
//sampleStart
    val words = "The quick brown fox jumps over the lazy dog".split(" ")
    val lengthsList = words.filter { println("filter: $it"); it.length > 3 }
        .map { println("length: ${it.length}"); it.length }
        .take(4)

    println("Lengths of first 4 words longer than 3 chars:")
    println(lengthsList)
//sampleEnd
}
```

运行此代码时，会看到 `filter()` 与 `map()` 函数的执行顺序与代码中出现的顺序相同。 首先，将看到 `filter`：对于所有元素，然后是 `length`：对于在过滤之后剩余的元素， 然后是最后两行的输出。

列表处理如下图：



Sequence

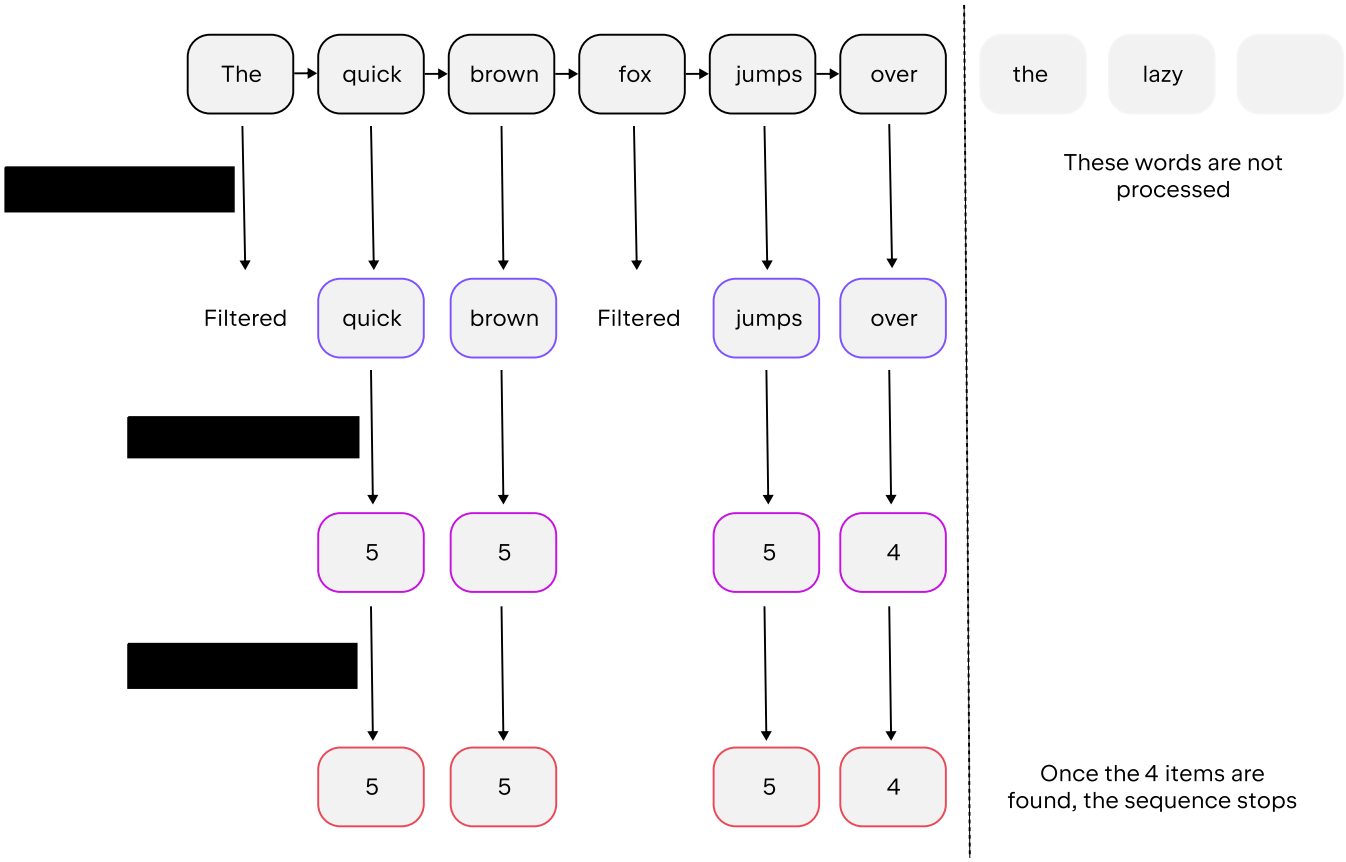
现在用序列写相同的逻辑：

```
fun main() {
//sampleStart
    val words = "The quick brown fox jumps over the lazy dog".split(" ")
    // 将列表转换为序列
    val wordsSequence = words.asSequence()

    val lengthsSequence = wordsSequence.filter { println("filter: $it"); it.length > 3
 }
        .map { println("length: ${it.length}"); it.length }
        .take(4)

    println("Lengths of first 4 words longer than 3 chars")
    // 末端操作：以列表形式获取结果。
    println(lengthsSequence.toList())
//sampleEnd
}
```

此代码的输出表明，仅在构建结果列表时才调用 `filter()` 与 `map()` 函数。 因此，首先看到文本 `"Lengths of.."` 的行，然后开始进行序列处理。 请注意，对于过滤后剩余的元素，映射在过滤下一个元素之前执行。 当结果大小达到 4 时，处理将停止，因为它是 `take(4)` 可以返

回的最大大小。

序列处理如下图：



In this example, the lazy processing of elements and stopping after finding four items reduces the number of operations compared to using a list approach.

集合操作概述

Kotlin 标准库提供了用于对集合执行操作的多种函数。这包括简单的操作，例如获取或添加元素，以及更复杂的操作，包括搜索、排序、过滤、 转换等。

扩展与成员函数

集合操作在标准库中以两种方式声明：集合接口的成员函数和扩展函数。

成员函数定义了对于集合类型是必不可少的操作。例如，`Collection` 包含函数 `isEmpty()` 来检查其是否为空；`List` 包含用于对元素进行索引访问的 `get()`， 等等。

创建自己的集合接口实现时，必须实现其成员函数。 为了使新实现的创建更加容易，请使用标准库中集合接口的框架实现：`AbstractCollection`、 `AbstractList`、 `AbstractSet`、 `AbstractMap` 及其相应可变抽象类。

其他集合操作被声明为扩展函数。这些是过滤、转换、排序以及其他集合处理函数。

公共操作

公共操作可用于只读集合与可变集合。 公共操作分为以下几类：

- 集合转换
- 集合过滤
- `plus` 与 `minus` 操作符
- 分组
- 取集合的一部分
- 取单个元素
- 集合排序
- 集合聚合操作

这些页面中描述的操作将返回其结果，而不会影响原始集合。例如，一个过滤操作产生一个新集合，其中包含与过滤谓词匹配的所有元素。 此类操作的结果应存储在变量中，或以其他方式使用，例如，传到其他函数中。

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")
    numbers.filter { it.length > 3 }  // `numbers` 没有任何改变，结果丢失
```

```
    println("numbers are still $numbers")
    val longerThan3 = numbers.filter { it.length > 3 } // 结果存储在 `longerThan3` 中
    println("numbers longer than 3 chars are $longerThan3")
//sampleEnd
}
```

对于某些集合操作，有一个选项可以指定 目标 对象。 目标是一个可变集合，该函数将其结果项附加到该可变对象中，而不是在新对象中返回它们。 对于执行带有目标的操作，有单独的函数，其名称中带有 `To` 后缀，例如，用 `filterTo()` 代替 `filter()` 以及用 `associateTo()` 代替 `associate()` 。 这些函数将目标集合作为附加参数。

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")
    val filterResults = mutableListOf<String>()  // 目标对象
    numbers.filterTo(filterResults) { it.length > 3 }
    numbers.filterIndexedTo(filterResults) { index, _ -> index == 0 }
    println(filterResults) // 包含两个操作的结果
//sampleEnd
}
```

为了方便起见，这些函数将目标集合返回了，因此可以在函数调用的相应参数中直接创建它：

```
fun main() {
    val numbers = listOf("one", "two", "three", "four")
//sampleStart
    // 将数字直接过滤到新的哈希集中，
    // 从而消除结果中的重复项
    val result = numbers.mapTo(HashSet()) { it.length }
    println("distinct item lengths are $result")
//sampleEnd
}
```

具有目标的函数可用于过滤、关联、分组、展平以及其他操作。 关于目标操作的完整列表，请参见 Kotlin collections reference。

写操作

对于可变集合，还存在可更改集合状态的 写操作 。这些操作包括添加、删除和更新元素。写操作在集合写操作以及 List 写操作与 Map 写操作的相应部分中列出。

对于某些操作，有成对的函数可以执行相同的操作：一个函数就地应用该操作， 另一个函数将结果作为单独的集合返回。 例如， `sort()` 就地对可变集合进行排序，因此其状态发生了变化； `sorted()` 创建一个新集合，该集合包含按排序顺序相同的元素。

```
fun main() {
//sampleStart
    val numbers = mutableListOf("one", "two", "three", "four")
    val sortedNumbers = numbers.sorted()
    println(numbers == sortedNumbers)  // false
    numbers.sort()
    println(numbers == sortedNumbers)  // true
//sampleEnd
}
```

集合转换操作

Kotlin 标准库为集合 转换 提供了一组扩展函数。 这些函数根据提供的转换规则从现有集合中构建新集合。 在此页面中，我们将概述可用的集合转换函数。

映射

映射 转换从另一个集合的元素上的函数结果创建一个集合。 基本的映射函数是 `map()`。 它将给定的 lambda 函数应用于每个后续元素，并返回 lambda 结果列表。 结果的顺序与元素的原始顺序相同。 如需应用还要用到元素索引作为参数的转换，请使用 `mapIndexed()`。

```
fun main() {
//sampleStart
    val numbers = setOf(1, 2, 3)
    println(numbers.map { it * 3 })
    println(numbers.mapIndexed { idx, value -> value * idx })
//sampleEnd
}
```

如果转换在某些元素上产生 `null` 值，那么可以通过调用 `mapNotNull()` 函数取代 `map()` 或 `mapIndexedNotNull()` 取代 `mapIndexed()` 来从结果集中过滤掉 `null` 值。

```
fun main() {
//sampleStart
    val numbers = setOf(1, 2, 3)
    println(numbers.mapNotNull { if ( it == 2) null else it * 3 })
    println(numbers.mapIndexedNotNull { idx, value -> if (idx == 0) null else value *
idx })
//sampleEnd
}
```

映射转换时，有两个选择：转换键，使值保持不变，反之亦然。 要将指定转换应用于键，请使用 `mapKeys()`； 反之， `mapValues()` 转换值。 这两个函数都使用将映射条目作为参数的转换，因此可以操作其键与值。

```
fun main() {
//sampleStart
    val numbersMap = mapOf("key1" to 1, "key2" to 2, "key3" to 3, "key11" to 11)
    println(numbersMap.mapKeys { it.key.uppercase() })
```

```
    println(numbersMap.mapValues { it.value + it.key.length })
//sampleEnd
}
```

合拢

合拢 转换是根据两个集合中具有相同位置的元素构建配对。 在 Kotlin 标准库中，这是通过 `zip()` 扩展函数完成的。

在一个集合（或数组）上以另一个集合（或数组）作为参数调用时，`zip()` 返回 `Pair` 对象的列表（`List`）。 接收者集合的元素是这些配对中的第一个元素。

如果集合的大小不同，则 `zip()` 的结果为较小集合的大小；结果中不包含较大集合的后续元素。

`zip()` 也可以中缀形式调用 `a zip b` 。

```
fun main() {
//sampleStart
    val colors = listOf("red", "brown", "grey")
    val animals = listOf("fox", "bear", "wolf")
    println(colors zip animals)

    val twoAnimals = listOf("fox", "bear")
    println(colors.zip(twoAnimals))
//sampleEnd
}
```

也可以使用带有两个参数的转换函数来调用 `zip()`：接收者元素和参数元素。 在这种情况下，结果 `List` 包含在具有相同位置的接收者对和参数元素对上调用的转换函数的返回值。

```
fun main() {
//sampleStart
    val colors = listOf("red", "brown", "grey")
    val animals = listOf("fox", "bear", "wolf")

    println(colors.zip(animals) { color, animal -> "The ${animal.replaceFirstChar { it
.uppercase() }} is $color"})
//sampleEnd
}
```

当拥有 `Pair` 的 `List` 时，可以进行反向转换 *unzipping*——从这些键值对中构建两个列表：

- 第一个列表包含原始列表中每个 `Pair` 的键。

- 第二个列表包含原始列表中每个 `Pair` 的值。

要分割键值对列表，请调用 `unzip()` 。

```
fun main() {
//sampleStart
    val numberPairs = listOf("one" to 1, "two" to 2, "three" to 3, "four" to 4)
    println(numberPairs.unzip())
//sampleEnd
}
```

关联

关联 转换允许从集合元素和与其关联的某些值构建 Map。 在不同的关联类型中，元素可以是关联 Map 中的键或值。

基本的关联函数 `associateWith()` 创建一个 `Map` ，其中原始集合的元素是键，并通过给定的转换函数从中产生值。 如果两个元素相等，则仅最后一个保留在 Map 中。

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")
    println(numbers.associateWith { it.length })
//sampleEnd
}
```

为了使用集合元素作为值来构建 Map，有一个函数 `associateBy()` 。 它需要一个函数，该函数根据元素的值返回键。如果两个元素的键相等，则仅最后一个保留在 Map 中。

还可以使用值转换函数来调用 `associateBy()` 。

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")

    println(numbers.associateBy { it.first().uppercaseChar() })
    println(numbers.associateBy(keySelector = { it.first().uppercaseChar() }, valueTra
nsform = { it.length }))
//sampleEnd
}
```

另一种构建 Map 的方法是使用函数 `associate()`，其中 Map 键和值都是通过集合元素生成的。 它需要一个 lambda 函数，该函数返回 `Pair`：键和相应 Map 条目的值。

请注意，`associate()` 会生成临时的 `Pair` 对象，这可能会影响性能。 因此，当性能不是很关键或比其他选项更可取时，应使用 `associate()`。

后者的一个示例：从一个元素一起生成键和相应的值。

```
fun main() {
data class FullName (val firstName: String, val lastName: String)

fun parseFullName(fullName: String): FullName {
    val nameParts = fullName.split(" ")
    if (nameParts.size == 2) {
        return FullName(nameParts[0], nameParts[1])
    } else throw Exception("Wrong name format")
}

//sampleStart
    val names = listOf("Alice Adams", "Brian Brown", "Clara Campbell")
    println(names.associate { name -> parseFullName(name).let { it.lastName to it.firs
tName } })
//sampleEnd
}
```

此时，首先在一个元素上调用一个转换函数，然后根据该函数结果的属性建立 Pair。

展平

如需操作嵌套的集合，那么可能会发现提供对嵌套集合元素进行打平访问的标准库函数很有用。

第一个函数为 `flatten()`。 可以在一个集合的集合（例如，一个 `Set` 组成的 `List`）上调用它。 该函数返回嵌套集合中的所有元素的一个 `List`。

```
fun main() {
//sampleStart
    val numberSets = listOf(setOf(1, 2, 3), setOf(4, 5, 6), setOf(1, 2))
    println(numberSets.flatten())
//sampleEnd
}
```

另一个函数——`flatMap()` 提供了一种灵活的方式来处理嵌套的集合。 它需要一个函数将一个集合元素映射到另一个集合。 因此，`flatMap()` 返回单个列表其中包含所有元素的值。 所以，`flatMap()` 表现为 `map()`（以集合作为映射结果）与 `flatten()` 的连续调用。

```
data class StringContainer(val values: List<String>)

fun main() {
//sampleStart
    val containers = listOf(
        StringContainer(listOf("one", "two", "three")),
        StringContainer(listOf("four", "five", "six")),
        StringContainer(listOf("seven", "eight"))
    )
    println(containers.flatMap { it.values })
//sampleEnd
}
```

字符串表示

如果需要以可读格式检索集合内容，请使用将集合转换为字符串的函数：`joinToString()` 与 `joinTo()`。

`joinToString()` 根据提供的参数从集合元素构建单个 `String`。 `joinTo()` 执行相同的操作，但将结果附加到给定的 `Appendable` 对象。

当使用默认实参调用时，函数返回的结果类似于在集合上调用 `toString()`： 各元素的字符串表示形式以空格分隔而成的 `String`。

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")

    println(numbers)
    println(numbers.joinToString())

    val listString = StringBuffer("The list of numbers: ")
    numbers.joinTo(listString)
    println(listString)
//sampleEnd
}
```

要构建自定义字符串表示形式，可以在函数参数 `separator`、`prefix` 与 `postfix` 中指定其参数。 结果字符串将以 `prefix` 开头，以 `postfix` 结尾。除最后一个元素外，`separator` 将位于每个元素之后。

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")
    println(numbers.joinToString(separator = " | ", prefix = "start: ", postfix = ": e
nd"))
//sampleEnd
}
```

对于较大的集合，可能需要指定 `limit` ——将包含在结果中元素的数量。 如果集合大小超出 `limit`，所有其他元素将被 `truncated` 参数的单个值替换。

```
fun main() {
//sampleStart
    val numbers = (1..100).toList()
    println(numbers.joinToString(limit = 10, truncated = "<...>"))
//sampleEnd
}
```

最后，要自定义元素本身的表示形式，请提供 `transform` 函数。

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")
    println(numbers.joinToString { "Element: ${it.uppercase()}"})
//sampleEnd
}
```

过滤集合

过滤是最常用的集合处理任务之一。在Kotlin中，过滤条件由 谓词 定义——接受一个集合元素并且返回布尔值的 lambda 表达式， 其返回值含义： `true` 说明给定元素与谓词匹配， `false` 则表示不匹配。

标准库包含了一组让你能够通过单个调用就可以过滤集合的扩展函数。 这些函数不会改变原始集合，因此它们既可用于可变集合也可用于只读集合。 为了操作过滤结果，应该在过滤后将其赋值给变量或链接其他函数。

按谓词过滤

基本的过滤函数是 `filter()` 。当使用一个谓词来调用时， `filter()` 返回与其匹配的集合元素。对于 `List` 和 `Set` ，过滤结果都是一个 `List` ，对 `Map` 来说结果还是一个 `Map` 。

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")
    val longerThan3 = numbers.filter { it.length > 3 }
    println(longerThan3)

    val numbersMap = mapOf("key1" to 1, "key2" to 2, "key3" to 3, "key11" to 11)
    val filteredMap = numbersMap.filter { (key, value) -> key.endsWith("1") && value >
10}
    println(filteredMap)
//sampleEnd
}
```

`filter()` 中的谓词只能检查元素的值。如果想在过滤中使用元素在集合中的位置，应该使用 `filterIndexed()` 。它接受一个带有两个参数的谓词：元素的索引和元素的值。

如果想使用否定条件来过滤集合，请使用 `filterNot()` 。它返回一个让谓词产生 `false` 的元素列表。

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")

    val filteredIdx = numbers.filterIndexed { index, s -> (index != 0) && (s.length < 5
)  }
    val filteredNot = numbers.filterNot { it.length <= 3 }
```

```
    println(filteredIdx)
    println(filteredNot)
//sampleEnd
}
```

还有一些函数能够通过过滤给定类型的元素来缩小元素的类型：

- `filterIsInstance()` 返回给定类型的集合元素。在一个 `List<Any>` 上被调用时，`filterIsInstance<T>()` 返回一个 `List<T>`，从而让你能够在集合元素上调用 `T` 类型的函数。

```
fun main() {
//sampleStart
    val numbers = listOf(null, 1, "two", 3.0, "four")
    println("All String elements in upper case:")
    numbers.filterIsInstance<String>().forEach {
        println(it.uppercase())
    }
//sampleEnd
}
```

- `filterNotNull()` 返回所有的非空元素。在一个 `List<T?>` 上被调用时，`filterNotNull()` 返回一个 `List<T: Any>`，从而让你能够将所有元素视为非空对象。

```
fun main() {
//sampleStart
    val numbers = listOf(null, "one", "two", null)
    numbers.filterNotNull().forEach {
    println(it.length)   // 对可空的 String 来说长度不可用
    }
//sampleEnd
}
```

划分

另一个过滤函数 – `partition()` – 通过一个谓词过滤集合并且将不匹配的元素存放在一个单独的列表中。因此，你得到一个 `List` 的 `Pair` 作为返回值：第一个列表包含与谓词匹配的元素并且第二个列表包含原始集合中的所有其他元素。

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")
    val (match, rest) = numbers.partition { it.length > 3 }
```

```
    println(match)
    println(rest)
//sampleEnd
}
```

检验谓词

最后，有些函数只是针对集合元素简单地检测一个谓词：

- 如果至少有一个元素匹配给定谓词，那么 `any()` 返回 `true`。
- 如果没有元素与给定谓词匹配，那么 `none()` 返回 `true`。
- 如果所有元素都匹配给定谓词，那么 `all()` 返回 `true`。 请注意，在一个空集合上使用任何有效的谓词去调用 `all()` 都会返回 `true` 。这种行为在逻辑上被称为 *vacuous truth*。

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")

    println(numbers.any { it.endsWith("e") })
    println(numbers.none { it.endsWith("a") })
    println(numbers.all { it.endsWith("e") })

    println(emptyList<Int>().all { it > 5 })   // vacuous truth
//sampleEnd
}
```

`any()` 和 `none()` 也可以不带谓词使用：在这种情况下它们只是用来检查集合是否为空。 如果集合中有元素， `any()` 返回 `true` ，否则返回 `false` ； `none()` 则相反。

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")
    val empty = emptyList<String>()

    println(numbers.any())
    println(empty.any())

    println(numbers.none())
    println(empty.none())
//sampleEnd
}
```

加减操作符

在 Kotlin 中，为集合定义了 `plus` (`+`) 和 `minus` (`-`) 操作符。 它们把一个集合作为第一个操作数；第二个操作数可以是一个元素或者是另一个集合。 返回值是一个新的只读集合：

- `plus` 的结果包含原始集合 和 第二个操作数中的元素。
- `minus` 的结果包含原始集合中的元素，但第二个操作数中的元素 除外。 如果第二个操作数是一个元素，那么 `minus` 移除其在原始集合中的 第一次 出现；如果是一个集合，那么移除其元素在原始集合中的 所有 出现。

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")

    val plusList = numbers + "five"
    val minusList = numbers - listOf("three", "four")
    println(plusList)
    println(minusList)
//sampleEnd
}
```

关于 map 的 `plus` 和 `minus` 操作符的详细信息，请参见 Map 相关操作。 也为集合定义了广义赋值操作符 `plusAssign` (`+=`) 与 `minusAssign` (`-=`)。 然而，对于只读集合，它们实际上使用 `plus` 或者 `minus` 操作符并尝试将结果赋值给同一变量。 因此，它们仅在由 `var` 声明的只读集合中可用。 对于可变集合，如果它是一个 `val` ，那么它们会修改集合。更多详细信息请参见集合写操作。

分组

Kotlin 标准库提供用于对集合元素进行分组的扩展函数。 基本函数 `groupBy()` 使用一个 lambda 函数并返回一个 `Map` 。 在此 Map 中，每个键都是 lambda 结果，而对应的值是返回此结果的元素 `List` 。 例如，可以使用此函数将 `String` 列表按首字母分组。

还可以使用第二个 lambda 参数（值转换函数）调用 `groupBy()` 。 在带有两个 lambda 的 `groupBy()` 结果 Map 中，由 `keySelector` 函数生成的键映射到值转换函数的结果，而不是原始元素。

This example illustrates using the `groupBy()` function to group the strings by their first letter, iterating through the groups on the resulting `Map` with the `for` operator, and then transforming the values to uppercase using the `valueTransform` function:

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four", "five")

    // Groups the strings by their first letter using groupBy()
    val groupedByFirstLetter = numbers.groupBy { it.first().uppercase() }
    println(groupedByFirstLetter)
    // {O=[one], T=[two, three], F=[four, five]}

    // Iterates through each group and prints the key and its associated values
    for ((key, value) in groupedByFirstLetter) {
        println("Key: $key, Values: $value")
    }
    // Key: O, Values: [one]
    // Key: T, Values: [two, three]
    // Key: F, Values: [four, five]

    // Groups the strings by their first letter and transforms the values to uppercase
    val groupedAndTransformed = numbers.groupBy(keySelector = { it.first() }, valueTra
nsform = { it.uppercase() })
    println(groupedAndTransformed)
    // {o=[ONE], t=[TWO, THREE], f=[FOUR, FIVE]}
//sampleEnd
}
```

如果要对元素进行分组，然后一次将操作应用于所有分组，请使用 `groupingBy()` 函数。 它返回一个 `Grouping` 类型的实例。 通过 `Grouping` 实例，可以以一种惰性的方式将操作应用于所有组：这些分组实际上是刚好在执行操作前构建的。

即， `Grouping` 支持以下操作：

- `eachCount()` 计算每个组中的元素。
- `fold()` 与 `reduce()` 对每个组分别执行 fold 与 reduce 操作，作为一个单独的集合并返回结果。
- `aggregate()` 随后将给定操作应用于每个组中的所有元素并返回结果。 这是对 `Grouping` 执行任何操作的通用方法。当折叠或缩小不够时，可使用它来实现自定义操作。

You can use the `for` operator on the resulting `Map` to iterate through the groups created by the `groupingBy()` function. This allows you to access each key and the count of elements associated with that key.

The following example demonstrates how to group strings by their first letter using the `groupingBy()` function, count the elements in each group, and then iterate through each group to print the key and count of elements:

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four", "five")

    // Groups the strings by their first letter using groupingBy() and counts the elem
ents in each group
    val grouped = numbers.groupingBy { it.first() }.eachCount()

    // Iterates through each group and prints the key and its associated values
    for ((key, count) in grouped) {
        println("Key: $key, Count: $count")
        // Key: o, Count: 1
        // Key: t, Count: 2
        // Key: f, Count: 2
    }
//sampleEnd
}
```

取集合的一部分

Kotlin 标准库包含用于取集合的一部分的扩展函数。 这些函数提供了多种方法来选择结果集合的元素：显式列出其位置、 指定结果大小等。

slice

`slice()` 返回具有给定索引的集合元素列表。 索引既可以是作为区间传入的也可以是作为整数值的集合传入的。

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four", "five", "six")
    println(numbers.slice(1..3))
    println(numbers.slice(0..4 step 2))
    println(numbers.slice(setOf(3, 5, 0)))
//sampleEnd
}
```

take 与 drop

要从头开始获取指定数量的元素，请使用 `take()` 函数。 要从尾开始获取指定数量的元素，请使用 `takeLast()`。 当调用的数字大于集合的大小时，两个函数都将返回整个集合。

要从头或从尾去除给定数量的元素，请调用 `drop()` 或 `dropLast()` 函数。

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four", "five", "six")
    println(numbers.take(3))
    println(numbers.takeLast(3))
    println(numbers.drop(1))
    println(numbers.dropLast(5))
//sampleEnd
}
```

还可以使用谓词来定义要获取或去除的元素的数量。 有四个与上述功能相似的函数：

- `takeWhile()` 是带有谓词的 `take()`：它将不停获取元素直到排除与谓词匹配的首个元

素。如果首个集合元素与谓词匹配，则结果为空。

- `takeLastWhile()` 与 `takeLast()` 类似：它从集合末尾获取与谓词匹配的元素区间。区间的首个元素是与谓词不匹配的最后一个元素右边的元素。如果最后一个集合元素与谓词匹配，则结果为空。
- `dropWhile()` 与具有相同谓词的 `takeWhile()` 相反：它将首个与谓词不匹配的元素返回到末尾。
- `dropLastWhile()` 与具有相同谓词的 `takeLastWhile()` 相反：它返回从开头到最后一个与谓词不匹配的元素。

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four", "five", "six")
    println(numbers.takeWhile { !it.startsWith('f') })
    println(numbers.takeLastWhile { it != "three" })
    println(numbers.dropWhile { it.length == 3 })
    println(numbers.dropLastWhile { it.contains('i') })
//sampleEnd
}
```

Chunked

要将集合分解为给定大小的“块”，请使用 `chunked()` 函数。 `chunked()` 采用一个参数（块的大小），并返回一个 `List` 其中包含给定大小的 `List` 。 第一个块从第一个元素开始并包含 `size` 元素，第二个块包含下一个 `size` 元素，依此类推。 最后一个块的大小可能较小。

```
fun main() {
//sampleStart
    val numbers = (0..13).toList()
    println(numbers.chunked(3))
//sampleEnd
}
```

还可以立即对返回的块应用转换。 为此，请在调用 `chunked()` 时将转换作为 lambda 函数提供。 lambda 参数是集合的一块。当通过转换调用 `chunked()` 时， 这些块是临时的 `List` ，应立即在该 lambda 中使用。

```
fun main() {
//sampleStart
    val numbers = (0..13).toList()
    println(numbers.chunked(3) { it.sum() })  // `it` 为原始集合的一个块
//sampleEnd
```

```
}
```

Windowed

可以检索给定大小的集合元素中所有可能区间。 获取它们的函数称为 `windowed()`： 它返回一个元素区间列表，比如通过给定大小的滑动窗口查看集合，则会看到该区间。 与 `chunked()` 不同， `windowed()` 返回从每个集合元素开始的元素区间（窗口）。 所有窗口都作为单个 `List` 的元素返回。

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four", "five")
    println(numbers.windowed(3))
//sampleEnd
}
```

`windowed()` 通过可选参数提供更大的灵活性：

- `step` 定义两个相邻窗口的第一个元素之间的距离。默认情况下，该值为 1，因此结果包含从所有元素开始的窗口。如果将 step 增加到 2，将只收到以奇数元素开头的窗口：第一个、第三个等。
- `partialWindows` 包含从集合末尾的元素开始的较小的窗口。例如，如果请求三个元素的窗口，就不能为最后两个元素构建它们。在本例中，启用 `partialWindows` 将包括两个大小为2与1的列表。

最后，可以立即对返回的区间应用转换。 为此，在调用 `windowed()` 时将转换作为 lambda 函数提供。

```
fun main() {
//sampleStart
    val numbers = (1..10).toList()
    println(numbers.windowed(3, step = 2, partialWindows = true))
    println(numbers.windowed(3) { it.sum() })
//sampleEnd
}
```

要构建两个元素的窗口，有一个单独的函数—— `zipWithNext()`。 它创建接收器集合的相邻元素对。 请注意， `zipWithNext()` 不会将集合分成几对；它为 每个 元素创建除最后一个元素外的对， 因此它在 `[1, 2, 3, 4]` 上的结果为 `[[1, 2], [2, 3], [3, 4]]`，而不是 `[[1, 2 ], [3, 4]]`。 `zipWithNext()` 也可以通过转换函数来调用；它应该以接收者集合的两个元素作为参数。

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four", "five")
    println(numbers.zipWithNext())
    println(numbers.zipWithNext() { s1, s2 -> s1.length > s2.length})
//sampleEnd
}
```

取单个元素

Kotlin 集合提供了一套从集合中检索单个元素的函数。 此页面描述的函数适用于 list 和 set。

正如 list 的定义所言，list 是有序集合。 因此，list 中的每个元素都有其位置可供你引用。 除了此页面上描述的函数外，list 还提供了更广泛的一套方法去按索引检索和搜索元素。 更多详细信息，请参见 List 相关操作。

反过来，从定义来看，set 并不是有序集合。 但是，Kotlin 中的 `Set` 按某些顺序存储元素。 这些可以是插入顺序（在 `LinkedHashSet` 中）、自然排序顺序（在 `SortedSet` 中）或者其他顺序。 一组元素的顺序也可以是未知的。 在这种情况下，元素仍会以某种顺序排序，因此，依赖元素位置的函数仍会返回其结果。 但是，除非调用者知道所使用的 `Set` 的具体实现，否则这些结果对于调用者是不可预测的。

按位置取

为了检索特定位置的元素，有一个函数 `elementAt()`。 用一个整数作为参数来调用它，你会得到给定位置的集合元素。 第一个元素的位置是 `0`，最后一个元素的位置是 `(size - 1)`。

`elementAt()` 对于不提供索引访问或非静态已知提供索引访问的集合很有用。 在使用 `List` 的情况下，使用索引访问操作符（`get()` 或 `[]`）更为习惯。

```
fun main() {
//sampleStart
    val numbers = linkedSetOf("one", "two", "three", "four", "five")
    println(numbers.elementAt(3))

    val numbersSortedSet = sortedSetOf("one", "two", "three", "four")
    println(numbersSortedSet.elementAt(0)) // 元素以升序存储
//sampleEnd
}
```

还有一些有用的别名来检索集合的第一个与最后一个元素：`first()` 与 `last()`。

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four", "five")
    println(numbers.first())
    println(numbers.last())
//sampleEnd
}
```

为了避免在检索位置不存在的元素时出现异常，请使用 `elementAt()` 的安全变体：

- 当指定位置超出集合范围时， `elementAtOrNull()` 返回 null。
- `elementAtOrElse()` 还接受一个 lambda 表达式，该表达式能将一个 `Int` 参数映射为一个集合元素类型的实例。 当使用一个越界位置来调用时， `elementAtOrElse()` 返回对给定值调用该 lambda 表达式的结果。

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four", "five")
    println(numbers.elementAtOrNull(5))
    println(numbers.elementAtOrElse(5) { index -> "The value for index $index is undef
ined"})
//sampleEnd
}
```

按条件取

函数 `first()` 和 `last()` 还可以让你在集合中搜索与给定谓词匹配的元素。 当你使用测试集合元素的谓词调用 `first()` 时，你会得到对其调用谓词产生 `true` 的第一个元素。 反过来，带有一个谓词的 `last()` 返回与其匹配的最后一个元素。

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four", "five", "six")
    println(numbers.first { it.length > 3 })
    println(numbers.last { it.startsWith("f") })
//sampleEnd
}
```

如果没有元素与谓词匹配，两个函数都会抛异常。 为了避免它们，请改用 `firstOrNull()` 与 `lastOrNull()`： 如果找不到匹配的元素，它们将返回 `null`。

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four", "five", "six")
    println(numbers.firstOrNull { it.length > 6 })
//sampleEnd
}
```

如果别名的名称更适合，那么可以使用别名：

- 使用 `find()` 代替 `firstOrNull()`
- 使用 `findLast()` 代替 `lastOrNull()`

```
fun main() {
//sampleStart
    val numbers = listOf(1, 2, 3, 4)
    println(numbers.find { it % 2 == 0 })
    println(numbers.findLast { it % 2 == 0 })
//sampleEnd
}
```

Retrieve with selector

If you need to map the collection before retrieving the element, there is a function `firstNotNullOf()`. It combines 2 actions:

- Maps the collection with the selector function
- Returns the first non-null value in the result

`firstNotNullOf()` throws the `NoSuchElementException` if the resulting collection doesn't have a non-nullable element. Use the counterpart `firstNotNullOfOrNull()` to return null in this case.

```
fun main() {
//sampleStart
    val list = listOf<Any>(0, "true", false)
    // Converts each element to string and returns the first one that has required len
gth
    val longEnough = list.firstNotNullOf { item -> item.toString().takeIf { it.length
>= 4 } }
    println(longEnough)
//sampleEnd
}
```

随机取元素

如果需要检索集合的一个随机元素，那么请调用 `random()` 函数。 你可以不带参数或者使用一个 `Random` 对象作为随机源来调用它。

```
fun main() {
//sampleStart
```

```
    val numbers = listOf(1, 2, 3, 4)
    println(numbers.random())
//sampleEnd
}
```

On empty collections, `random()` throws an exception. To receive `null` instead, use `randomOrNull()`

检测元素存在与否

如需检查集合中某个元素的存在，可以使用 `contains()` 函数。 如果存在一个集合元素等于（ `equals()` ）函数参数，那么它返回 `true` 。 你可以使用 `in` 关键字以操作符的形式调用 `contains()` 。

如需一次检查多个实例的存在，可以使用这些实例的集合作为参数调用 `containsAll()` 。

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four", "five", "six")
    println(numbers.contains("four"))
    println("zero" in numbers)

    println(numbers.containsAll(listOf("four", "two")))
    println(numbers.containsAll(listOf("one", "zero")))
//sampleEnd
}
```

此外，你可以通过调用 `isEmpty()` 和 `isNotEmpty()` 来检查集合中是否包含任何元素。

```
fun main() {
//sampleStart
  val numbers = listOf("one", "two", "three", "four", "five", "six")
  println(numbers.isEmpty())
  println(numbers.isNotEmpty())

  val empty = emptyList<String>()
  println(empty.isEmpty())
  println(empty.isNotEmpty())
//sampleEnd
}
```

排序

元素的顺序是某些集合类型的一个重要方面。 例如，如果拥有相同元素的两个列表的元素顺序不同，那么这两个列表也不相等。

在 Kotlin 中，可以通过多种方式定义对象的顺序。

首先，有 自然 顺序。它是为 `Comparable` 接口的实现定义的。 当没有指定其他顺序时，使用自然顺序为它们排序。

大多数内置类型是可比较的：

- 数值类型使用传统的数值顺序：`1` 大于 `0`；`-3.4f` 大于 `-5f`，以此类推。
- `Char` 和 `String` 使用字典顺序：`b` 大于 `a`；`world` 大于 `hello`。

如需为用户定义的类型定义一个自然顺序，可以让这个类型实现 `Comparable`。 这需要实现 `compareTo()` 函数。 `compareTo()` 必须将另一个具有相同类型的对象作为参数并返回一个整数值来显示哪个对象更大：

- 正值表明接收者对象更大。
- 负值表明它小于参数。
- 0 说明对象相等。

Below is a class for ordering versions that consist of the major and the minor part.

```
class Version(val major: Int, val minor: Int): Comparable<Version> {
    override fun compareTo(other: Version): Int = when {
        this.major != other.major -> this.major compareTo other.major // compareTo() i
n the infix form
        this.minor != other.minor -> this.minor compareTo other.minor
        else -> 0
    }
}

fun main() {
    println(Version(1, 2) > Version(1, 3))
    println(Version(2, 0) > Version(1, 5))
}
```

自定义 顺序让你可以按自己喜欢的方式对任何类型的实例进行排序。 特别是，你可以为不可比较类型定义顺序，或者为可比较类型定义非自然顺序。 如需为类型定义自定义顺序，可以为其创建一个 `Comparator`。 `Comparator` 包含 `compare()` 函数：它接受一个类的两个实例并返回它们之间比较的整数结果。 如上所述，对结果的解释与 `compareTo()` 的结果相同。

```
fun main() {
//sampleStart
    val lengthComparator = Comparator { str1: String, str2: String -> str1.length - st
r2.length }
    println(listOf("aaa", "bb", "c").sortedWith(lengthComparator))
//sampleEnd
}
```

有了 `lengthComparator`，你可以按照字符串的长度而不是默认的字典顺序来排列字符串。

定义一个 `Comparator` 的一种比较简短的方式是标准库中的 `compareBy()` 函数。`compareBy()` 接受一个 lambda 表达式，该表达式从一个实例产生一个 `Comparable` 值， 并将自定义顺序定义为生成值的自然顺序。

使用 `compareBy()`，上面示例中的长度比较器如下所示：

```
fun main() {
//sampleStart
    println(listOf("aaa", "bb", "c").sortedWith(compareBy { it.length }))
//sampleEnd
}
```

You can also define an order based on multiple criteria. For example, to sort strings by their length and alphabetically when the lengths are equal, you can write:

```
fun main() {
//sampleStart
    val sortedStrings = listOf("aaa", "bb", "c", "b", "a", "aa", "ccc")
        .sortedWith { a, b ->
           when (val compareLengths = a.length.compareTo(b.length)) {
             0 -> a.compareTo(b)
             else -> compareLengths
           }
         }

    println(sortedStrings)
    // [a, b, c, aa, bb, aaa, ccc]
//sampleEnd
}
```

Since sorting by multiple criteria is a common scenario, the Kotlin standard library provides the `thenBy()` function that you can use to add a secondary sorting rule.

For example, you can combine `compareBy()` with `thenBy()` to sort strings by their length first and alphabetically second, just like in the previous example:

```
fun main() {
//sampleStart
    val sortedStrings = listOf("aaa", "bb", "c", "b", "a", "aa", "ccc")
        .sortedWith(compareBy<String> { it.length }.thenBy { it })

    println(sortedStrings)
    // [a, b, c, aa, bb, aaa, ccc]
//sampleEnd
}
```

Kotlin 集合包提供了用于按照自然顺序、自定义顺序甚至随机顺序对集合排序的函数。 在此页面上，我们将介绍适用于只读集合的排序函数。 这些函数将它们的结果作为一个新集合返回，集合里包含了按照请求顺序排序的来自原始集合的元素。 如果想学习就地对可变集合排序的函数，请参见 List 相关操作。

自然顺序

基本的函数 `sorted()` 和 `sortedDescending()` 返回集合的元素，这些元素按照其自然顺序升序和降序排序。 这些函数适用于 `Comparable` 元素的集合。

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")

    println("Sorted ascending: ${numbers.sorted()}")
    println("Sorted descending: ${numbers.sortedDescending()}")
//sampleEnd
}
```

自定义顺序

为了按照自定义顺序排序或者对不可比较对象排序，可以使用函数 `sortedBy()` 和 `sortedByDescending()`。 它们接受一个将集合元素映射为 `Comparable` 值的选择器函数，并以该值的自然顺序对集合排序。

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")

    val sortedNumbers = numbers.sortedBy { it.length }
    println("Sorted by length ascending: $sortedNumbers")
    val sortedByLast = numbers.sortedByDescending { it.last() }
    println("Sorted by the last letter descending: $sortedByLast")
```

```
//sampleEnd
}
```

如需为集合排序定义自定义顺序，可以提供自己的 `Comparator`。 为此，调用传入 `Comparator` 的 `sortedWith()` 函数。 使用此函数，按照字符串长度排序如下所示：

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")
    println("Sorted by length ascending: ${numbers.sortedWith(compareBy { it.length })
}")
//sampleEnd
}
```

倒序

你可以使用 `reversed()` 函数以相反的顺序检索集合。

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")
    println(numbers.reversed())
//sampleEnd
}
```

`reversed()` 返回带有元素副本的新集合。 因此，如果你之后改变了原始集合，这并不会影响先前获得的 `reversed()` 的结果。

另一个反向函数—— `asReversed()` ——返回相同集合实例的一个反向视图，因此，如果原始列表不会发生变化，那么它会比 `reversed()` 更轻量，更合适。

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")
    val reversedNumbers = numbers.asReversed()
    println(reversedNumbers)
//sampleEnd
}
```

如果原始列表是可变的，那么其所有更改都会反映在其反向视图中，反之亦然。

```
fun main() {
//sampleStart
```

```
    val numbers = mutableListOf("one", "two", "three", "four")
    val reversedNumbers = numbers.asReversed()
    println(reversedNumbers)
    numbers.add("five")
    println(reversedNumbers)
//sampleEnd
}
```

但是，如果列表的可变性未知或者源根本不是一个列表，那么 `reversed()` 更合适，因为其结果是一个未来不会更改的副本。

随机顺序

最后，`shuffled()` 函数返回一个包含了以随机顺序排序的集合元素的新的 `List`。 你可以不带参数或者使用 `Random` 对象来调用它。

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")
    println(numbers.shuffled())
//sampleEnd
}
```

聚合操作

Kotlin 集合包含用于常用的 聚合操作（基于集合内容返回单个值的操作）的函数。 其中大多数是众所周知的，并且其工作方式与在其他语言中相同。

- `minOrNull()` 与 `maxOrNull()` 分别返回最小和最大的元素。 On empty collections, they return `null`.
- `average()` 返回数字集合中元素的平均值。
- `sum()` 返回数字集合中元素的总和。
- `count()` 返回集合中元素的数量。

```
fun main() {
    val numbers = listOf(6, 42, 10, 4)

    println("Count: ${numbers.count()}")
    println("Max: ${numbers.maxOrNull()}")
    println("Min: ${numbers.minOrNull()}")
    println("Average: ${numbers.average()}")
    println("Sum: ${numbers.sum()}")
}
```

还有一些通过某些选择器函数或自定义 `Comparator` 来检索最小和最大元素的函数。

- `maxByOrNull()` 与 `minByOrNull()` 接受一个选择器函数并返回使选择器返回最大或最小值的元素。
- `maxWithOrNull()` 与 `minWithOrNull()` 接受一个 `Comparator` 对象并且根据此 `Comparator` 对象返回最大或最小元素。
- `maxOfOrNull()` and `minOfOrNull()` take a selector function and return the largest or the smallest return value of the selector itself.
- `maxOfWithOrNull()` and `minOfWithOrNull()` take a `Comparator` object and return the largest or smallest selector return value according to that `Comparator`.

These functions return `null` on empty collections. There are also alternatives – `maxOf`, `minOf`, `maxOfWith`, and `minOfWith` – which do the same as their counterparts but throw a `NoSuchElementException` on empty collections.

```
fun main() {
//sampleStart
    val numbers = listOf(5, 42, 10, 4)
    val min3Remainder = numbers.minByOrNull { it % 3 }
    println(min3Remainder)
```

```
    val strings = listOf("one", "two", "three", "four")
    val longestString = strings.maxWithOrNull(compareBy { it.length })
    println(longestString)
//sampleEnd
}
```

Besides regular `sum()`, there is an advanced summation function `sumOf()` that takes a selector function and returns the sum of its application to all collection elements. Selector can return different numeric types: `Int`, `Long`, `Double`, `UInt`, and `ULong` (also `BigInteger` and `BigDecimal` on the JVM).

```
fun main() {
//sampleStart
    val numbers = listOf(5, 42, 10, 4)
    println(numbers.sumOf { it * 2 })
    println(numbers.sumOf { it.toDouble() / 2 })
//sampleEnd
}
```

fold 与 reduce

对于更特定的情况，有函数 `reduce()` 和 `fold()`，它们依次将所提供的操作应用于集合元素并返回累积的结果。 操作有两个参数：先前的累积值和集合元素。

这两个函数的区别在于：`fold()` 接受一个初始值并将其用作第一步的累积值，而 `reduce()` 的第一步则将第一个和第二个元素作为第一步的操作参数。

```
fun main() {
//sampleStart
    val numbers = listOf(5, 2, 10, 4)

    val simpleSum = numbers.reduce { sum, element -> sum + element }
    println(simpleSum)
    val sumDoubled = numbers.fold(0) { sum, element -> sum + element * 2 }
    println(sumDoubled)

    //错误：第一个元素在结果中没有加倍
    //val sumDoubledReduce = numbers.reduce { sum, element -> sum + element * 2 }
    //println(sumDoubledReduce)
//sampleEnd
}
```

上面的实例展示了区别：`fold()` 用于计算加倍的元素之和。 如果将相同的函数传给 `reduce()`，那么它会返回另一个结果，因为在第一步中它将列表的第一个和第二个元素作为参数，所以第一个元素不会被加倍。

如需将函数以相反的顺序应用于元素，可以使用函数 `reduceRight()` 和 `foldRight()` 它们的工作方式类似于 `fold()` 和 `reduce()`，但从最后一个元素开始，然后再继续到前一个元素。记住，在使用 foldRight 或 reduceRight 时，操作参数会更改其顺序：第一个参数变为元素，然后第二个参数变为累积值。

```
fun main() {
//sampleStart
    val numbers = listOf(5, 2, 10, 4)
    val sumDoubledRight = numbers.foldRight(0) { element, sum -> sum + element * 2 }
    println(sumDoubledRight)
//sampleEnd
}
```

你还可以使用将元素索引作为参数的操作。 为此，使用函数 `reduceIndexed()` 与 `foldIndexed()` 传递元素索引作为操作的第一个参数。

最后，还有将这些操作从右到左应用于集合元素的函数—— `reduceRightIndexed()` 与 `foldRightIndexed()` 。

```
fun main() {
//sampleStart
    val numbers = listOf(5, 2, 10, 4)
    val sumEven = numbers.foldIndexed(0) { idx, sum, element -> if (idx % 2 == 0) sum
+ element else sum }
    println(sumEven)

    val sumEvenRight = numbers.foldRightIndexed(0) { idx, element, sum -> if (idx % 2
== 0) sum + element else sum }
    println(sumEvenRight)
//sampleEnd
}
```

All reduce operations throw an exception on empty collections. To receive `null` instead, use their `*OrNull()` counterparts:

- `reduceOrNull()`
- `reduceRightOrNull()`
- `reduceIndexedOrNull()`
- `reduceRightIndexedOrNull()`

For cases where you want to save intermediate accumulator values, there are functions `runningFold()` (or its synonym `scan()`) and `runningReduce()`.

```
fun main() {
//sampleStart
    val numbers = listOf(0, 1, 2, 3, 4, 5)
    val runningReduceSum = numbers.runningReduce { sum, item -> sum + item }
    val runningFoldSum = numbers.runningFold(10) { sum, item -> sum + item }
//sampleEnd
    val transform = { index: Int, element: Int -> "N = ${index + 1}: $element" }
    println(runningReduceSum.mapIndexed(transform).joinToString("\n", "Sum of first N
elements with runningReduce:\n"))
    println(runningFoldSum.mapIndexed(transform).joinToString("\n", "Sum of first N el
ements with runningFold:\n"))
}
```

If you need an index in the operation parameter, use `runningFoldIndexed()` or `runningReduceIndexed()`.

集合写操作

可变集合支持更改集合内容的操作，例如添加或删除元素。 在此页面上，我们将描述实现 `MutableCollection` 的所有写操作。 关于 `List` 与 `Map` 可用的更多特定操作，请分别参见 List 相关操作与 Map 相关操作。

添加元素

要将单个元素添加到列表或集合，请使用 `add()` 函数。指定的对象将添加到集合的末尾。

```
fun main() {
//sampleStart
    val numbers = mutableListOf(1, 2, 3, 4)
    numbers.add(5)
    println(numbers)
//sampleEnd
}
```

`addAll()` 将参数对象的每个元素添加到列表或集合中。参数可以是 `Iterable` 、 `Sequence` 或 `Array` 。 接收者的类型和参数可能不同，例如，你可以将所有内容从 `Set` 添加到 `List` 。

当在列表上调用时， `addAll()` 会按照在参数中出现的顺序添加各个新元素。 你也可以调用 `addAll()` 时指定一个元素位置作为第一参数。 参数集合的第一个元素会被插入到这个位置。 其他元素将跟随在它后面，将接收者元素移到末尾。

```
fun main() {
//sampleStart
    val numbers = mutableListOf(1, 2, 5, 6)
    numbers.addAll(arrayOf(7, 8))
    println(numbers)
    numbers.addAll(2, setOf(3, 4))
    println(numbers)
//sampleEnd
}
```

你还可以使用 `plus` 运算符 - `plusAssign` (`+=`) 添加元素。 当应用于可变集合时， `+=` 将第二个操作数(一个元素或另一个集合)追加到集合的末尾。

```
fun main() {
//sampleStart
    val numbers = mutableListOf("one", "two")
    numbers += "three"
    println(numbers)
    numbers += listOf("four", "five")
    println(numbers)
//sampleEnd
}
```

删除元素

若要从可变集合中移除元素，请使用 `remove()` 函数。 `remove()` 接受元素值，并删除该值的一个匹配项。

```
fun main() {
//sampleStart
    val numbers = mutableListOf(1, 2, 3, 4, 3)
    numbers.remove(3)                    // 删除了第一个 `3`
    println(numbers)
    numbers.remove(5)                    // 什么都没删除
    println(numbers)
//sampleEnd
}
```

要一次删除多个元素，有以下函数：

- `removeAll()` 移除参数集合中存在的所有元素。 或者，你可以用谓词作为参数来调用它；在这种情况下，函数移除谓词产生 `true` 的所有元素。
- `retainAll()` 与 `removeAll()` 相反：它移除除参数集合中的元素之外的所有元素。 当与谓词一起使用时，它只留下与之匹配的元素。
- `clear()` 从列表中移除所有元素并将其置空。

```
fun main() {
//sampleStart
    val numbers = mutableListOf(1, 2, 3, 4)
    println(numbers)
    numbers.retainAll { it >= 3 }
    println(numbers)
    numbers.clear()
    println(numbers)

    val numbersSet = mutableSetOf("one", "two", "three", "four")
    numbersSet.removeAll(setOf("one", "two"))
    println(numbersSet)
```

```
//sampleEnd
}
```

从集合中移除元素的另一种方法是使用 `minusAssign`（`-=`）——原地修改版的 `minus` 操作符。 `minus` 操作符。 第二个参数可以是元素类型的单个实例或另一个集合。 右边是单个元素时，`-=` 会移除它的第一个匹配项。 反过来，如果它是一个集合，那么它的所有元素的每次出现都会删除。 例如，如果列表包含重复的元素，它们将被同时删除。 第二个操作数可以包含集合中不存在的元素。这些元素不会影响操作的执行。

```
fun main() {
//sampleStart
    val numbers = mutableListOf("one", "two", "three", "three", "four")
    numbers -= "three"
    println(numbers)
    numbers -= listOf("four", "five")
    //numbers -= listOf("four")    // 与上述相同
    println(numbers)
//sampleEnd
}
```

更新元素

list 和 map 还提供更新元素的操作。 它们在 List 相关操作与 Map 相关操作中有所描述。 对于 set 来说，更新没有意义，因为它实际上是移除一个元素并添加另一个元素。

List 相关操作

`List` 是 Kotlin 标准库中最受欢迎的集合类型。对列表元素的索引访问为 List 提供了一组强大的操作。

按索引取元素

List 支持按索引取元素的所有常用操作： `elementAt()` 、 `first()` 、 `last()` 与取单个元素中列出的其他操作。 List 的特点是能通过索引访问特定元素，因此读取元素的最简单方法是按索引检索它。 这是通过 `get()` 函数或简写语法 `[index]` 来传递索引参数完成的。

如果 List 长度小于指定的索引，则抛出异常。 另外，还有两个函数能避免此类异常：

- `getOrElse()` 提供用于计算默认值的函数，如果集合中不存在索引，则返回默认值。
- `getOrNull()` 返回 `null` 作为默认值。

```
fun main() {
//sampleStart
    val numbers = listOf(1, 2, 3, 4)
    println(numbers.get(0))
    println(numbers[0])
    //numbers.get(5)                        // exception!
    println(numbers.getOrNull(5))             // null
    println(numbers.getOrElse(5, {it}))       // 5
//sampleEnd
}
```

取列表的一部分

除了取集合的一部分中常用的操作， List 还提供 `subList()` 函数，该函数将指定元素范围的视图作为列表返回。 因此，如果原始集合的元素发生变化，则它在先前创建的子列表中也会发生变化，反之亦然。

```
fun main() {
//sampleStart
    val numbers = (0..13).toList()
    println(numbers.subList(3, 6))
//sampleEnd
}
```

查找元素位置

线性查找

在任何列表中，都可以使用 `indexOf()` 或 `lastIndexOf()` 函数找到元素的位置。 它们返回与列表中给定参数相等的元素的第一个或最后一个位置。 如果没有这样的元素，则两个函数均返回 `-1` 。

```
fun main() {
//sampleStart
    val numbers = listOf(1, 2, 3, 4, 2, 5)
    println(numbers.indexOf(2))
    println(numbers.lastIndexOf(2))
//sampleEnd
}
```

还有一对函数接受谓词并搜索与之匹配的元素：

- `indexOfFirst()` 返回与谓词匹配的第一个元素的索引，如果没有此类元素，则返回 `-1` 。
- `indexOfLast()` 返回与谓词匹配的最后一个元素的索引，如果没有此类元素，则返回 `-1` 。

```
fun main() {
//sampleStart
    val numbers = mutableListOf(1, 2, 3, 4)
    println(numbers.indexOfFirst { it > 2})
    println(numbers.indexOfLast { it % 2 == 1})
//sampleEnd
}
```

在有序列表中二分查找

还有另一种搜索列表中元素的方法——二分查找算法。 它的工作速度明显快于其他内置搜索功能，但要求该列表按照一定的顺序（自然排序或函数参数中提供的另一种排序）按升序排序过。 否则，结果是不确定的。

要搜索已排序列表中的元素，请调用 `binarySearch()` 函数，并将该值作为参数传递。 如果存在这样的元素，则函数返回其索引；否则，将返回 `(-insertionPoint - 1)` ，其中 `insertionPoint` 为应插入此元素的索引，以便列表保持排序。 如果有多个具有给定值的元素，搜索则可以返回其任何索引。

还可以指定要搜索的索引区间：在这种情况下，该函数仅在两个提供的索引之间搜索。

```
fun main() {
//sampleStart
    val numbers = mutableListOf("one", "two", "three", "four")
    numbers.sort()
    println(numbers)
    println(numbers.binarySearch("two"))  // 3
    println(numbers.binarySearch("z")) // -5
    println(numbers.binarySearch("two", 0, 2))  // -3
//sampleEnd
}
```

Comparator 二分搜索

如果列表元素不是 `Comparable`，则应提供一个用于二分搜索的 `Comparator`。 该列表必须根据此 `Comparator` 以升序排序。来看一个例子：

```
data class Product(val name: String, val price: Double)

fun main() {
//sampleStart
    val productList = listOf(
        Product("WebStorm", 49.0),
        Product("AppCode", 99.0),
        Product("DotTrace", 129.0),
        Product("ReSharper", 149.0))

    println(productList.binarySearch(Product("AppCode", 99.0), compareBy<Product> { it
.price }.thenBy { it.name }))
//sampleEnd
}
```

这是一个不可 `排序` 的 `Product` 实例列表，以及一个定义排序的 `Comparator`：如果 `p1` 的价格小于 `p2` 的价格，则产品 `p1` 在产品 `p2` 之前。 因此，按照此顺序对列表进行升序排序后，使用 `binarySearch()` 查找指定的 `Product` 的索引。

当列表使用与自然排序不同的顺序时（例如，对 `String` 元素不区分大小写的顺序），自定义 Comparator 也很方便。

```
fun main() {
//sampleStart
    val colors = listOf("Blue", "green", "ORANGE", "Red", "yellow")
    println(colors.binarySearch("RED", String.CASE_INSENSITIVE_ORDER)) // 3
```

```
//sampleEnd
}
```

比较函数二分搜索

使用 比较 函数的二分搜索无需提供明确的搜索值即可查找元素。 取而代之的是，它使用一个比较函数将元素映射到 `Int` 值，并搜索函数返回 0 的元素。 该列表必须根据提供的函数以升序排序；换句话说， 比较的返回值必须从一个列表元素增长到下一个列表元素。

```
import kotlin.math.sign
//sampleStart
data class Product(val name: String, val price: Double)

fun priceComparison(product: Product, price: Double) = sign(product.price - price).toI
nt()

fun main() {
    val productList = listOf(
        Product("WebStorm", 49.0),
        Product("AppCode", 99.0),
        Product("DotTrace", 129.0),
        Product("ReSharper", 149.0))

    println(productList.binarySearch { priceComparison(it, 99.0) })
}
//sampleEnd
```

Comparator 与比较函数二分搜索都可以针对列表区间执行。

List 写操作

除了集合写操作中描述的集合修改操作之外， 可变列表还支持特定的写操作。 这些操作使用索引来访问元素以扩展列表修改功能。

添加

要将元素添加到列表中的特定位置，请使用 `add()` 或 `addAll()` 并提供元素插入的位置作为附加参数。 位置之后的所有元素都将向右移动。

```
fun main() {
//sampleStart
    val numbers = mutableListOf("one", "five", "six")
    numbers.add(1, "two")
```

```
    numbers.addAll(2, listOf("three", "four"))
    println(numbers)
//sampleEnd
}
```

更新

列表还提供了在指定位置替换元素的函数——`set()` 及其操作符形式 `[]`。`set()` 不会更改其他元素的索引。

```
fun main() {
//sampleStart
    val numbers = mutableListOf("one", "five", "three")
    numbers[1] =  "two"
    println(numbers)
//sampleEnd
}
```

`fill()` 简单地将所有集合元素的值替换为指定值。

```
fun main() {
//sampleStart
    val numbers = mutableListOf(1, 2, 3, 4)
    numbers.fill(3)
    println(numbers)
//sampleEnd
}
```

删除

要从列表中删除指定位置的元素，请使用 `removeAt()` 函数，并将位置作为参数。 在元素被删除之后出现的所有元素索引将减 1。

```
fun main() {
//sampleStart
    val numbers = mutableListOf(1, 2, 3, 4, 3)
    numbers.removeAt(1)
    println(numbers)
//sampleEnd
}
```

排序

在集合排序中，描述了按特定顺序检索集合元素的操作。 对于可变列表，标准库中提供了类似的扩展函数，这些扩展函数可以执行相同的排序操作。 将此类操作应用于列表实例时，它将更改指定实例中元素的顺序。

就地排序函数的名称与应用于只读列表的函数的名称相似，但没有 `ed/d` 后缀：

- `sort*` 在所有排序函数的名称中代替 `sorted*`：`sort()`、`sortDescending()`、`sortBy()` 等等。
- `shuffle()` 代替 `shuffled()`。
- `reverse()` 代替 `reversed()`。

`asReversed()` 在可变列表上调用会返回另一个可变列表，该列表是原始列表的反向视图。在该视图中的更改将反映在原始列表中。 以下示例展示了可变列表的排序函数：

```
fun main() {
//sampleStart
    val numbers = mutableListOf("one", "two", "three", "four")

    numbers.sort()
    println("Sort into ascending: $numbers")
    numbers.sortDescending()
    println("Sort into descending: $numbers")

    numbers.sortBy { it.length }
    println("Sort into ascending by length: $numbers")
    numbers.sortByDescending { it.last() }
    println("Sort into descending by the last letter: $numbers")

    numbers.sortWith(compareBy<String> { it.length }.thenBy { it })
    println("Sort by Comparator: $numbers")

    numbers.shuffle()
    println("Shuffle: $numbers")

    numbers.reverse()
    println("Reverse: $numbers")
//sampleEnd
}
```

Set 相关操作

Kotlin 集合包中包含 set 常用操作的扩展函数：找出集合间的交集、并集或差集。

要将两个集合合并为一个（并集），可使用 `union()` 函数。也能以中缀形式使用 `a union b`。注意，对于有序集合，操作数的顺序很重要。在结果集合中，左侧操作数在前：

```
fun main() {
//sampleStart
    val numbers = setOf("one", "two", "three")

    // output according to the order
    println(numbers union setOf("four", "five"))
    // [one, two, three, four, five]
    println(setOf("four", "five") union numbers)
    // [four, five, one, two, three]
//sampleEnd
}
```

To find an intersection between two collections (elements present in both of them), use the `intersect()` function. To find collection elements not present in another collection, use the `subtract()` function. Both these functions can be called in the infix form as well, for example, `a intersect b`:

```
fun main() {
//sampleStart
    val numbers = setOf("one", "two", "three")

    // same output
    println(numbers intersect setOf("two", "one"))
    // [one, two]
    println(numbers subtract setOf("three", "four"))
    // [one, two]
    println(numbers subtract setOf("four", "three"))
    // [one, two]
//sampleEnd
}
```

To find the elements present in either one of the two collections but not in their intersection, you can also use the `union()` function. For this operation (known as symmetric difference), calculate the differences between the two collections and merge the results:

```
fun main() {
//sampleStart
```

```
    val numbers = setOf("one", "two", "three")
    val numbers2 = setOf("three", "four")

    // merge differences
    println((numbers - numbers2) union (numbers2 - numbers))
    // [one, two, four]
//sampleEnd
}
```

You can also apply `union()`, `intersect()`, and `subtract()` functions to lists. However, their result is *always* a `Set`. In this result, all the duplicate elements are merged into one and the index access is not available:

```
fun main() {
//sampleStart
    val list1 = listOf(1, 1, 2, 3, 5, 8, -1)
    val list2 = listOf(1, 1, 2, 2, 3, 5)

    // result of intersecting two lists is a Set
    println(list1 intersect list2)
    // [1, 2, 3, 5]

    // equal elements are merged into one
    println(list1 union list2)
    // [1, 2, 3, 5, 8, -1]
//sampleEnd
}
```

Map 相关操作

在 map 中，键和值的类型都是用户定义的。 对基于键的访问启用了各种 map 特有的处理函数，从键获取值到对键和值进行单独过滤。 在此页面上，我们提供了来自标准库的 map 处理功能的描述。

取键与值

要从 Map 中检索值，必须提供其键作为 `get()` 函数的参数。 还支持简写 `[key]` 语法。 如果找不到给定的键，则返回 `null` 。 还有一个函数 `getValue()` ，它的行为略有不同：如果在 Map 中找不到键，则抛出异常。 此外，还有两个选项可以解决键缺失的问题：

- `getOrElse()` 与 list 的工作方式相同：对于不存在的键，其值由给定的 lambda 表达式返回。
- `getOrDefault()` 如果找不到键，则返回指定的默认值。

```
fun main() {
//sampleStart
    val numbersMap = mapOf("one" to 1, "two" to 2, "three" to 3)
    println(numbersMap.get("one"))
    println(numbersMap["one"])
    println(numbersMap.getOrDefault("four", 10))
    println(numbersMap["five"])               // null
    //numbersMap.getValue("six")      // exception!
//sampleEnd
}
```

要对 map 的所有键或所有值执行操作，可以从属性 `keys` 和 `values` 中相应地检索它们。 `keys` 是 Map 中所有键的集合， `values` 是 Map 中所有值的集合。

```
fun main() {
//sampleStart
    val numbersMap = mapOf("one" to 1, "two" to 2, "three" to 3)
    println(numbersMap.keys)
    println(numbersMap.values)
//sampleEnd
}
```

过滤

可以使用 `filter()` 函数来过滤 map 或其他集合。 对 map 使用 `filter()` 函数时， `Pair` 将作为参数的谓词传递给它。 它将使用谓词同时过滤其中的键和值。

```
fun main() {
//sampleStart
    val numbersMap = mapOf("key1" to 1, "key2" to 2, "key3" to 3, "key11" to 11)
    val filteredMap = numbersMap.filter { (key, value) -> key.endsWith("1") && value > 10}
    println(filteredMap)
//sampleEnd
}
```

还有两种用于过滤 map 的特定函数：按键或按值。 这两种方式，都有对应的函数： `filterKeys()` 和 `filterValues()` 。 两者都将返回一个新 Map ，其中包含与给定谓词相匹配的条目。 `filterKeys()` 的谓词仅检查元素键， `filterValues()` 的谓词仅检查值。

```
fun main() {
//sampleStart
    val numbersMap = mapOf("key1" to 1, "key2" to 2, "key3" to 3, "key11" to 11)
    val filteredKeysMap = numbersMap.filterKeys { it.endsWith("1") }
    val filteredValuesMap = numbersMap.filterValues { it < 10 }

    println(filteredKeysMap)
    println(filteredValuesMap)
//sampleEnd
}
```

加减操作符

由于需要访问元素的键， `plus` （ `+` ）与 `minus` （ `-` ）运算符对 map 的作用与其他集合不同。 `plus` 返回包含两个操作数元素的 `Map` ：左侧的 Map 与右侧的 Pair 或另一个 Map 。 当右侧操作数中有左侧 `Map` 中已存在的键时， 该条目将使用右侧的值。

```
fun main() {
//sampleStart
    val numbersMap = mapOf("one" to 1, "two" to 2, "three" to 3)
    println(numbersMap + Pair("four", 4))
    println(numbersMap + Pair("one", 10))
    println(numbersMap + mapOf("five" to 5, "one" to 11))
//sampleEnd
}
```

`minus` 将根据左侧 `Map` 条目创建一个新 `Map` ，右侧操作数带有键的条目将被剔除。 因此，右侧操作数可以是单个键或键的集合： list 、 set 等。

```
fun main() {
//sampleStart
    val numbersMap = mapOf("one" to 1, "two" to 2, "three" to 3)
    println(numbersMap - "one")
    println(numbersMap - listOf("two", "four"))
//sampleEnd
}
```

关于在可变 Map 中使用 `plusAssign` （ `+=` ）与 `minusAssign` （ `-=` ）操作符的详细信息，请参见 Map 写操作 。

Map 写操作

Mutable Map （可变 Map ）提供特定的 Map 写操作。 这些操作使你可以使用键来访问或更改 Map 值。

Map 写操作的一些规则：

- 值可以更新。 反过来，键也永远不会改变：添加条目后，键是不变的。
- 每个键都有一个与之关联的值。也可以添加和删除整个条目。

下面是对可变 Map 中可用写操作的标准库函数的描述。

添加与更新条目

要将新的键值对添加到可变 Map ，请使用 `put()` 。 将新条目放入 `LinkedHashMap` （Map的默认实现）后，会添加该条目，以便在 Map 迭代时排在最后。 在 Map 类中，新元素的位置由其键顺序定义。

```
fun main() {
//sampleStart
    val numbersMap = mutableMapOf("one" to 1, "two" to 2)
    numbersMap.put("three", 3)
    println(numbersMap)
//sampleEnd
}
```

要一次添加多个条目，请使用 `putAll()` 。 其参数可以是 `Map` 或一组 `Pair` ： `Iterable` 、 `Sequence` 或 `Array` 。

```
fun main() {
//sampleStart
    val numbersMap = mutableMapOf("one" to 1, "two" to 2, "three" to 3)
    numbersMap.putAll(setOf("four" to 4, "five" to 5))
    println(numbersMap)
//sampleEnd
}
```

如果给定键已存在于 Map 中，则 `put()` 与 `putAll()` 都将覆盖值。 因此，可以使用它们来更新 Map 条目的值。

```
fun main() {
//sampleStart
    val numbersMap = mutableMapOf("one" to 1, "two" to 2)
    val previousValue = numbersMap.put("one", 11)
    println("value associated with 'one', before: $previousValue, after: ${numbersMap["one"]}")
    println(numbersMap)
//sampleEnd
}
```

还可以使用快速操作符将新条目添加到 Map 。 有两种方式：

- `plusAssign` （ `+=` ）操作符。
- `[]` 操作符为 `set()` 的别名。

```
fun main() {
//sampleStart
    val numbersMap = mutableMapOf("one" to 1, "two" to 2)
    numbersMap["three"] = 3     // 调用 numbersMap.put("three", 3)
    numbersMap += mapOf("four" to 4, "five" to 5)
    println(numbersMap)
//sampleEnd
}
```

使用 Map 中存在的键进行操作时，将覆盖相应条目的值。

删除条目

要从可变 Map 中删除条目，请使用 `remove()` 函数。 调用 `remove()` 时，可以传递键或整个键值对。 如果同时指定键和值，则仅当键值都匹配时，才会删除此的元素。

```
fun main() {
//sampleStart
    val numbersMap = mutableMapOf("one" to 1, "two" to 2, "three" to 3)
    numbersMap.remove("one")
    println(numbersMap)
    numbersMap.remove("three", 4)            //不会删除任何条目
    println(numbersMap)
//sampleEnd
}
```

还可以通过键或值从可变 Map 中删除条目。 在 Map 的 `.keys` 或 `.values` 中调用 `remove()` 并提供键或值来删除条目。 在 `.values` 中调用时， `remove()` 仅删除给定值匹配到的的第一个条目。

```
fun main() {
//sampleStart
    val numbersMap = mutableMapOf("one" to 1, "two" to 2, "three" to 3, "threeAgain" t
o 3)
    numbersMap.keys.remove("one")
    println(numbersMap)
    numbersMap.values.remove(3)
    println(numbersMap)
//sampleEnd
}
```

`minusAssign` （ `-=` ）操作符也可用于可变 Map 。

```
fun main() {
//sampleStart
    val numbersMap = mutableMapOf("one" to 1, "two" to 2, "three" to 3)
    numbersMap -= "two"
    println(numbersMap)
    numbersMap -= "five"             //不会删除任何条目
    println(numbersMap)
//sampleEnd
}
```

读取标准输入

Use the `readln()` function to read data from the standard input. It reads the whole line as a string:

```
// Reads and stores the user input in a variable. For example: Hi there!
val myInput = readln()

println(myInput)
// Hi there!

// Reads and prints the user input without storing it in a variable. For example: Hi,
Kotlin!
println(readln())
// Hi, Kotlin!
```

To work with data types other than strings, you can convert the input using conversion functions like `.toInt()`, `.toLong()`, `.toDouble()`, `.toFloat()`, or `.toBoolean()`. It is possible to read multiple inputs of different data types and store each input in a variable:

```
// Converts the input from a string to an integer value. For example: 12
val myNumber = readln().toInt()
println(myNumber)
// 12

// Converts the input from a string to a double value. For example: 345
val myDouble = readln().toDouble()
println(myDouble)
// 345.0

// Converts the input from a string to a boolean value. For example: true
val myBoolean = readln().toBoolean()
println(myBoolean)
// true
```

These conversion functions assume the user enters a valid representation of the target data type. For example, converting "hello" to an integer using `.toInt()` would result in an exception as the function expects a number in the string input.

To read several input elements separated by a delimiter, use the `.split()` function specifying the delimiter. The following code sample reads from the standard input, splits the input into a list of elements based on the delimiter, and converts each element of the list into a specific type:

```
// Reads the input, assuming the elements are separated by spaces, and converts them i
nto integers. For example: 1 2 3
```

```
val numbers = readln().split(' ').map { it.toInt() }
println(numbers)
//[1, 2, 3]

// Reads the input, assuming the elements are separated by commas, and converts them i
nto doubles. For example: 4,5,6
val doubles = readln().split(',').map { it.toDouble() }
println(doubles)
//[4.0, 5.0, 6.0]
```

> For an alternative way to read user input in Kotlin/JVM, see Standard input with Java Scanner.
>
> 

Handle standard input safely

You can use the `.toIntOrNull()` function to safely convert user input from a string to an integer. This function returns an integer if the conversion is successful. However, if the input is not a valid representation of an integer, it returns `null`:

```
// Returns null if the input is invalid. For example: Hello!
val wrongInt = readln().toIntOrNull()
println(wrongInt)
// null

// Converts a valid input from a string to an integer. For example: 13
val correctInt = readln().toIntOrNull()
println(correctInt)
// 13
```

The `readlnOrNull()` function is also helpful in safely handling the user input. The `readlnOrNull()` function reads from the standard input and returns null if the end of the input is reached, whereas `readln()` throws an exception in such a case.

选择加入要求

Kotlin 标准库提供了一种机制，用于要求并明确同意使用某些 API 元素。 通过这种机制，库作者可以将需要选择加入的特定条件告知用户， 例如，当某个 API 处于实验状态，并且将来可能会更改时。

为了保护用户，编译器会对这些条件发出警告，并要求用户在使用 API 之前选择加入。

Opt in to API

如果库作者将其库的 API 声明标记为要求选择加入， 必须在代码中使用它之前明确表示同意。 有多种方式可以选择加入。我们建议选择最适合实际情况的方式。

Opt in locally

To opt in to a specific API element when you use it in your code, use the `@OptIn` annotation with a reference to the experimental API marker. For example, suppose you want to use the `DateProvider` class, which requires an opt-in:

```
// 库代码
@RequiresOptIn(message = "This API is experimental. It could change in the future with
out notice.")
@Retention(AnnotationRetention.BINARY)
@Target(AnnotationTarget.CLASS, AnnotationTarget.FUNCTION)
annotation class MyDateTime

@MyDateTime
// 要求选择加入的类
class DateProvider
```

In your code, before declaring a function that uses the `DateProvider` class, add the `@OptIn` annotation with a reference to the `MyDateTime` annotation class:

```
// 客户端代码
@OptIn(MyDateTime::class)

// Uses DateProvider
fun getDate(): Date {
    val dateProvider: DateProvider
    // ...
}
```

It's important to note that with this approach, if the `getDate()` function is called elsewhere in your code or used by another developer, no opt-in is required:

```
// Client code
@OptIn(MyDateTime::class)

// Uses DateProvider
fun getDate(): Date {
    val dateProvider: DateProvider
    // ...
}

fun displayDate() {
    // OK: No opt-in is required
    println(getDate())
}
```

The opt-in requirement is not propagated, which means others might unknowingly use experimental APIs. To avoid this, it is safer to propagate the opt-in requirements.

Propagate opt-in requirements

在代码中使用供第三方使用的 API（例如库）时，你也可以把其选择加入的要求传播到自己的 API。为此，请在你的声明上标注与该库使用的相同要求选择加入的注解。

For example, before declaring a function that uses the `DateProvider` class, add the `@MyDateTime` annotation:

```
// Client code
@MyDateTime
fun getDate(): Date {
    // OK: the function requires opt-in as well
    val dateProvider: DateProvider
    // ...
}

fun displayDate() {
    println(getDate())
    // Error: getDate() requires opt-in
}
```

如本例所示，带注释的函数看起来是 `@MyDateTime` API 的一部分。 这种选择加入会将选择加入的要求传播到 `getDate()` 函数的用户。

If the signature of an API element includes a type that requires opt-in, the signature itself must also require opt-in. Otherwise, if an API element doesn't require opt-in, but its signature includes a type that does, using it triggers an error.

```
// Client code
@MyDateTime
fun getDate(dateProvider: DateProvider = DateProvider()): Date

@MyDateTime
fun displayDate() {
    // OK: the function requires opt-in as well
    println(getDate())
}
```

Similarly, if you apply `@OptIn` to a declaration whose signature includes a type that requires opt-in, the opt-in requirement still propagates:

```
//客户端代码
@OptIn(MyDateTime::class)
// Propagates opt-in due to DateProvider in the signature
fun getDate(dateProvider: DateProvider = DateProvider()): Date

fun displayDate() {
    println(getDate())
    // Error: getDate() requires opt-in
}
```

When propagating opt-in requirements, it's important to understand that if an API element becomes stable and no longer has an opt-in requirement, any other API elements that still have the opt-in requirement remain experimental. For example, suppose a library author removes the opt-in requirement for the `getDate()` function because it's now stable:

```
// Library code
// No opt-in requirement
fun getDate(): Date {
    val dateProvider: DateProvider
    // ...
}
```

If you use the `displayDate()` function without removing the opt-in annotation, it remains experimental even though the opt-in is no longer needed:

```
// Client code

// Still experimental!
@MyDateTime
```

```
fun displayDate() {
    // Uses a stable library function
    println(getDate())
}
```

Opt in to multiple APIs

To opt in to multiple APIs, mark the declaration with all their opt-in requirement annotations. For example:

```
@ExperimentalCoroutinesApi
@FlowPreview
```

Or alternatively with `@OptIn`:

```
@OptIn(ExperimentalCoroutinesApi::class, FlowPreview::class)
```

Opt in a file

要在一个文件的所有函数和类中使用要求选择加入的 API，请在文件的顶部， 文件包说明和导入声明前添加文件级注释 `@file:OptIn` 。

```
//客户端代码
@file:OptIn(MyDateTime::class)
```

Opt in a module

> The `-opt-in` compiler option is available since Kotlin 1.6.0. For earlier Kotlin versions, use `-Xopt-in`.
>
> 

If you don't want to annotate every usage of APIs that require opt-in, you can opt in to them for your whole module. To opt in to using an API in a module, compile it with the argument `-opt-in`, specifying the fully qualified name of the opt-in requirement annotation of the API you use: `-opt-in=org.mylibrary.OptInAnnotation`. Compiling with this argument has the same effect as if every declaration in the module has the annotation `@OptIn(OptInAnnotation::class)`.

如果使用 Gradle 构建模块，可以添加如下参数：

> 【Kotlin】

```
import org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTask
// ...

tasks.named<KotlinCompilationTask<*>>("compileKotlin").configure {
    compilerOptions.optIn.add("org.mylibrary.OptInAnnotation")
}
```

> 【Groovy】

```
import org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTask
// ...

tasks.named('compileKotlin', KotlinCompilationTask) {
    compilerOptions {
        optIn.add('org.mylibrary.OptInAnnotation')
    }
}
```

如果你的 Gradle 模块是多平台模块，请使用 `optIn` 方法：

> 【Kotlin】

```
sourceSets {
    all {
        languageSettings.optIn("org.mylibrary.OptInAnnotation")
    }
}
```

> 【Groovy】

```
sourceSets {
    all {
        languageSettings {
            optIn('org.mylibrary.OptInAnnotation')
        }
    }
}
```

对于 Maven，请使用以下内容：

```
<build>
    <plugins>
        <plugin>
            <groupId>org.jetbrains.kotlin</groupId>
            <artifactId>kotlin-maven-plugin</artifactId>
            <version>${kotlin.version}</version>
            <executions>...</executions>
            <configuration>
                <args>
                    <arg>-opt-in=org.mylibrary.OptInAnnotation</arg>

                </args>
            </configuration>
        </plugin>
    </plugins>
</build>
```

要在模块级别选择加入多个 API，请为每个要求选择加入的 API 添加以上描述的参数之一。

Opt in to inherit from a class or interface

Sometimes, a library author provides an API but wants to require users to explicitly opt in before they can extend it. For example, the library API may be stable for use but not for inheritance, as it might be extended in the future with new abstract functions. Library authors can enforce this by marking open or abstract classes and non-functional interfaces with the `@SubclassOptInRequired` annotation.

To opt in to use such an API element and extend it in your code, use the `@SubclassOptInRequired` annotation with a reference to the annotation class. For example, suppose you want to use the `CoreLibraryApi` interface, which requires an opt-in:

```
// Library code
@RequiresOptIn(
 level = RequiresOptIn.Level.WARNING,
 message = "Interfaces in this library are experimental"
)
annotation class UnstableApi()

@SubclassOptInRequired(UnstableApi::class)
// An interface requiring opt-in to extend
interface CoreLibraryApi
```

In your code, before creating a new interface that inherits from the `CoreLibraryApi` interface, add the `@SubclassOptInRequired` annotation with a reference to the `UnstableApi` annotation class:

```
// Client code
@SubclassOptInRequired(UnstableApi::class)
interface SomeImplementation : CoreLibraryApi
```

Note that when you use the `@SubclassOptInRequired` annotation on a class, the opt-in requirement is not propagated to any inner or nested classes:

```
// Library code
@RequiresOptIn
annotation class ExperimentalFeature

@SubclassOptInRequired(ExperimentalFeature::class)
open class FileSystem {
    open class File
}

// Client code

// Opt-in is required
class NetworkFileSystem : FileSystem()

// Nested class
// No opt-in required
class TextFile : FileSystem.File()
```

Alternatively, you can opt in by using the `@OptIn` annotation. You can also use an experimental marker annotation to propagate the requirement further to any uses of the class in your code:

```
// Client code
// With @OptIn annotation
@OptInRequired(UnstableApi::class)
interface SomeImplementation : CoreLibraryApi

// With annotation referencing annotation class
// Propagates the opt-in requirement further
@UnstableApi
interface SomeImplementation : CoreLibraryApi
```

Require opt-in to use API

You can require users of your library to opt in before they are able to use your API. Additionally, you can inform users about any special conditions for using your API until you decide to remove the opt-in requirement.

创建选择加入要求的注解

如需让用户选择加入才能使用你的模块 API，请创建一个注解类，作为要求选择加入的注解。这个类必须使用 `@RequiresOptIn` 标注：

```
@RequiresOptIn
@Retention(AnnotationRetention.BINARY)
@Target(AnnotationTarget.CLASS, AnnotationTarget.FUNCTION)
annotation class MyDateTime
```

要求选择加入的注解必须满足几个要求。它们必须具备：

- `BINARY` 或 `RUNTIME` retention。
- `EXPRESSION`、 `FILE`、 `TYPE` 或 `TYPE_PARAMETER` 作为 target。
- 没有参数。

选择加入的要求可以具有以下两个严格级别之一：

- `RequiresOptIn.Level.ERROR`。选择加入是强制性的。 否则，使用标记 API 的代码将无法编译。 这是默认级别。
- `RequiresOptIn.Level.WARNING`。选择加入不是强制性的，而是建议使用的。 没有它，编译器会发出警告。

要设置所需的级别，请指定 `@RequiresOptIn` 注解的 `level` 参数。

另外，你可以向 API 用户提供一个 `message`。编译器会将此消息显示给尝试使用该 API 但未选择加入的用户：

```
@RequiresOptIn(level = RequiresOptIn.Level.WARNING, message = "This API is experimenta
l. It can be incompatibly changed in the future.")
@Retention(AnnotationRetention.BINARY)
@Target(AnnotationTarget.CLASS, AnnotationTarget.FUNCTION)
annotation class ExperimentalDateTime
```

如果你发布了多个需要选择加入的独立功能，请为每个功能声明一个注解。 这让你的客户可以更安全地使用你的 API，因为他们只能使用其明确接受的功能。 这也意味着你可以独立地从功能中删除选择加入的要求，which makes your API easier to maintain.

标记 API 元素

要在使用 API 时要求选择加入，请给它的声明添加要求选择加入的注解。

```
@MyDateTime
class DateProvider
```

```
@MyDateTime
fun getTime(): Time {}
```

Note that for some language elements, an opt-in requirement annotation is not applicable:

- You can't annotate a backing field or a getter of a property, just the property itself.
- You can't annotate a local variable or a value parameter.

Require opt-in to extend API

There may be times when you want more granular control over which specific parts of your API can be used and extended. For example, when you have some API that is stable to use but:

- **Unstable to implement** due to ongoing evolution, such as when you have a family of interfaces where you expect to add new abstract functions without default implementations.
- **Delicate or fragile to implement**, such as individual functions that need to behave in a coordinated manner.
- **Has a contract that may be weakened in the future** in a backward-incompatible way for external implementations, such as changing an input parameter `T` to a nullable version `T?` where the code didn't previously consider `null` values.

In such cases, you can require users to opt in to your API before they can extend it further. Users can extend your API by inheriting from the API or implementing abstract functions. By using the `@SubclassOptInRequired` annotation, you can enforce this requirement to opt-in for open or abstract classes and non-functional interfaces.

To add the opt-in requirement to an API element, use the `@SubclassOptInRequired` annotation with a reference to the annotation class:

```
@RequiresOptIn(
 level = RequiresOptIn.Level.WARNING,
 message = "Interfaces in this library are experimental"
)
annotation class UnstableApi()

@SubclassOptInRequired(UnstableApi::class)
// An interface requiring opt-in to extend
interface CoreLibraryApi
```

Note that when you use the `@SubclassOptInRequired` annotation to require opt-in, the requirement is not propagated to any inner or nested classes.

For a real-world example of how to use the `@SubclassOptInRequired` annotation in your API, check out the `SharedFlow` interface in the `kotlinx.coroutines` library.

稳定前 API 的选择加入要求

如果要求选择加入尚未稳定的特性，请仔细处理 API 由实验状态到稳定状态的转换， 以避免破坏客户端代码。

当稳定前 API 稳定之后并以稳定状态发布后，请从你的声明中删除要求选择加入的注解。 客户就可以不受限制地使用它们。但是，你应该将注解类留在模块中，以便与现有的客户代码保持兼容。

为了鼓励 API 用户通过从代码中删除任何注解并重新编译来更新其模块，请将注解标记为 `@Deprecated` 并在弃用 message 中提供说明。

```
@Deprecated("This opt-in requirement is not used anymore. Remove its usages from your
code.")
@RequiresOptIn
annotation class ExperimentalDateTime
```

作用域函数

Kotlin 标准库包含几个函数，它们的唯一目的是在对象的上下文中执行代码块。 当对一个对象调用这样的函数并提供一个 lambda 表达式时，它会形成一个临时作用域。在此作用域中，可以访问该对象而无需其名称。这些函数称为作用域函数。 共有以下五种：`let`、 `run`、 `with`、 `apply`、 以及 `also`。

这些函数基本上都执行同样的操作：在一个对象上执行一个代码块。不同的是这个对象在块中如何使用，以及整个表达式的结果是什么。

下面是作用域函数用法的典型示例:

```
data class Person(var name: String, var age: Int, var city: String) {
    fun moveTo(newCity: String) { city = newCity }
    fun incrementAge() { age++ }
}

fun main() {
//sampleStart
    Person("Alice", 20, "Amsterdam").let {
        println(it)
        it.moveTo("London")
        it.incrementAge()
        println(it)
    }
//sampleEnd
}
```

如果不使用 `let` 来写这段代码，就必须引入一个新变量，并在每次使用它时重复其名称。

```
data class Person(var name: String, var age: Int, var city: String) {
    fun moveTo(newCity: String) { city = newCity }
    fun incrementAge() { age++ }
}

fun main() {
//sampleStart
    val alice = Person("Alice", 20, "Amsterdam")
    println(alice)
    alice.moveTo("London")
    alice.incrementAge()
    println(alice)
//sampleEnd
}
```

作用域函数没有引入任何新的技术，但是它们可以使你的代码更加简洁易读。

由于作用域函数的许多相似性，因此为你的案例选择正确的函数可能有点棘手。 选择主要取决于你的意图与在项目中使用的一致性。下面我们将详细描述各种作用域函数的差异及其约定用法。

函数选择

为了帮助你选择合适的作用域函数，我们提供了总结它们之间的主要差异的这张表。

函数	对象引用	返回值	是否是扩展函数
`let`	`it`	Lambda 表达式结果	是
`run`	`this`	Lambda 表达式结果	是
`run`	-	Lambda 表达式结果	不是：调用无需上下文对象
`with`	`this`	Lambda 表达式结果	不是：把上下文对象当做参数
`apply`	`this`	上下文对象	是
`also`	`it`	上下文对象	是

Detailed information about these functions is provided in the dedicated sections below.

以下是根据预期目的选择作用域函数的简短指南：

- 对一个非空（non-null）对象执行 lambda 表达式：`let`
- 将表达式作为变量引入为局部作用域中：`let`
- 对象配置：`apply`
- 对象配置并且计算结果：`run`
- 在需要表达式的地方运行语句：非扩展的 `run`
- 附加效果：`also`
- 一个对象的一组函数调用：`with`

不同作用域函数的使用场景存在重叠，可以根据项目或团队中使用的特定约定来选择使用哪些函数。

虽然作用域函数可以让代码更加简洁，但是要避免过度使用它们：这会使代码难以阅读并可能导致错误。 我们还建议避免嵌套作用域函数，同时链式调用它们时要小心：因为很容易混淆当前上下文对象与 `this` 或 `it` 的值。

区别

由于作用域函数本质上都非常相似，因此了解它们之间的区别很重要。 每个作用域函数之间有两个主要区别：

- 引用上下文对象的方式。
- 返回值。

上下文对象：this 还是 it

在传入作用域函数的 lambda 表达式内部，上下文对象可以不使用其实际名称而是使用一个更简短的引用来访问。 每个作用域函数都使用以下两种方式之一来引用上下文对象：作为 lambda 表达式的接收者（`this`）或者作为 lambda 表达式的参数（`it`）。两者都提供了同样的功能，因此我们会针对不同的场景描述两者的优缺点，并提供使用建议。

```
fun main() {
    val str = "Hello"
    // this
    str.run {
        println("The string's length: $length")
        //println("The string's length: ${this.length}") // 和上句效果相同
    }

    // it
    str.let {
        println("The string's length is ${it.length}")
    }
}
```

this

`run`、`with` 以及 `apply` 通过关键字 `this` 将上下文对象引用为 lambda 表达式的接收者。因此，在它们的 lambda 表达式中可以像在普通的类函数中一样访问上下文对象。

在大多数场景，当你访问接收者对象时你可以省略 `this`， 来让你的代码更简短。 相对地，如果省略了 `this`，就很难区分接收者对象的成员及外部对象或函数。因此，对于主要对对象的成员进行操作（调用其函数或赋值其属性）的 lambda 表达式， 建议将上下文对象作为接收者（`this`）。

```
data class Person(var name: String, var age: Int = 0, var city: String = "")

fun main() {
//sampleStart
    val adam = Person("Adam").apply {
        age = 20                       // 同 this.age = 20
        city = "London"
    }
    println(adam)
```

```
//sampleEnd
}
```

it

反过来，`let` 及 `also` 将上下文对象引用为 lambda 表达式参数。如果没有指定参数名，对象可以用隐式默认名称 `it` 访问。`it` 比 `this` 简短，带有 `it` 的表达式通常更易读。

不过，当调用对象函数或属性时，不能像 `this` 这样隐式地访问对象。 因此，当上下文对象在作用域中主要用作函数调用中的参数时，通过 `it` 访问上下文对象会更好。 在代码块中使用多个变量时，`it` 也更好一些。

```
import kotlin.random.Random

fun writeToLog(message: String) {
    println("INFO: $message")
}

fun main() {
//sampleStart
    fun getRandomInt(): Int {
        return Random.nextInt(100).also {
            writeToLog("getRandomInt() generated value $it")
        }
    }

    val i = getRandomInt()
    println(i)
//sampleEnd
}
```

The example below demonstrates referencing the context object as a lambda argument with argument name: `value`.

```
import kotlin.random.Random

fun writeToLog(message: String) {
    println("INFO: $message")
}

fun main() {
//sampleStart
    fun getRandomInt(): Int {
        return Random.nextInt(100).also { value ->
            writeToLog("getRandomInt() generated value $value")
        }
    }
```

```
    val i = getRandomInt()
    println(i)
//sampleEnd
}
```

返回值

根据返回结果，作用域函数可以分为以下两类：

- `apply` 及 `also` 返回上下文对象。
- `let`、`run` 及 `with` 返回 lambda 表达式结果.

You should consider carefully what return value you want based on what you want to do next in your code. This helps you to choose the best scope function to use.

上下文对象

`apply` 及 `also` 的返回值是上下文对象本身。因此，它们可以作为辅助步骤包含在调用链中：可以继续在同一个对象上一个接一个地进行链式函数调用。

```
fun main() {
//sampleStart
    val numberList = mutableListOf<Double>()
    numberList.also { println("Populating the list") }
        .apply {
            add(2.71)
            add(3.14)
            add(1.0)
        }
        .also { println("Sorting the list") }
        .sort()
//sampleEnd
    println(numberList)
}
```

它们还可以用在返回上下文对象的函数的 return 语句中。

```
import kotlin.random.Random

fun writeToLog(message: String) {
    println("INFO: $message")
}

fun main() {
//sampleStart
    fun getRandomInt(): Int {
        return Random.nextInt(100).also {
```

```
            writeToLog("getRandomInt() generated value $it")
        }
    }

    val i = getRandomInt()
//sampleEnd
}
```

Lambda 表达式结果

`let`、`run` 及 `with` 返回 lambda 表达式的结果。所以，在需要使用其结果给一个变量赋值，或者在需要对其结果进行链式操作等情况下，可以使用它们。

```
fun main() {
//sampleStart
    val numbers = mutableListOf("one", "two", "three")
    val countEndsWithE = numbers.run {
        add("four")
        add("five")
        count { it.endsWith("e") }
    }
    println("There are $countEndsWithE elements that end with e.")
//sampleEnd
}
```

此外，还可以忽略返回值，仅使用作用域函数为局部变量创建一个临时作用域。

```
fun main() {
//sampleStart
    val numbers = mutableListOf("one", "two", "three")
    with(numbers) {
        val firstItem = first()
        val lastItem = last()
        println("First item: $firstItem, last item: $lastItem")
    }
//sampleEnd
}
```

几个函数

为了帮助你为应用场景选择合适的作用域函数，我们会详细地描述它们并且提供一些使用建议。 从技术角度讲，作用域函数在很多场景里是可以互换的，所以这些示例展示了使用它们的约定。

let

- 上下文对象作为 lambda 表达式的参数（ `it` ）来访问。
- 返回值是 lambda 表达式的结果。

`let` 可用于在调用链的结果上调用一个或多个函数。例如，以下代码打印对集合的两个操作的结果：

```
fun main() {
//sampleStart
    val numbers = mutableListOf("one", "two", "three", "four", "five")
    val resultList = numbers.map { it.length }.filter { it > 3 }
    println(resultList)
//sampleEnd
}
```

With `let`, you can rewrite the above example so that you're not assigning the result of the list operations to a variable:

```
fun main() {
//sampleStart
    val numbers = mutableListOf("one", "two", "three", "four", "five")
    numbers.map { it.length }.filter { it > 3 }.let {
        println(it)
        // 如果需要可以调用更多函数
    }
//sampleEnd
}
```

如果传给 `let` 的代码块仅包含以 `it` 作为参数的单个函数，那么可以使用方法引用（ `::` ）代替 lambda 表达式参数：

```
fun main() {
//sampleStart
    val numbers = mutableListOf("one", "two", "three", "four", "five")
    numbers.map { it.length }.filter { it > 3 }.let(::println)
//sampleEnd
}
```

`let` 经常用于执行包含非空值代码块。如需对非空对象执行操作， 可对其使用安全调用操作符 `?.` 并调用 `let` 在 lambda 表达式中执行操作。

```
fun processNonNullString(str: String) {}

fun main() {
//sampleStart
    val str: String? = "Hello"
    //processNonNullString(str)       // 编译错误：str 可能为空
```

```
    val length = str?.let {
        println("let() called on $it")
        processNonNullString(it)      // 编译通过: 'it' 在 '?.let { }' 中必不为空
        it.length
    }
//sampleEnd
}
```

也可以使用 `let` 引入作用域受限的局部变量使代码更易读。 如需为上下文对象定义一个新变量，可提供其名称作为 lambda 表达式参数来代替默认的 `it` 。

```
fun main() {
//sampleStart
    val numbers = listOf("one", "two", "three", "four")
    val modifiedFirstItem = numbers.first().let { firstItem ->
        println("The first item of the list is '$firstItem'")
        if (firstItem.length >= 5) firstItem else "!" + firstItem + "!"
    }.uppercase()
    println("First item after modifications: '$modifiedFirstItem'")
//sampleEnd
}
```

with

- 上下文对象作为接收者（ `this` ）使用。
- 返回值是 lambda 表达式结果。

As `with` is not an extension function: the context object is passed as an argument, but inside the lambda, it's available as a receiver (`this`).

我们建议当不需要使用 lambda 表达式结果时，使用 `with` 来调用上下文对象上的函数。 在代码中， `with` 可以理解为“对于这个对象，执行以下操作。”

```
fun main() {
//sampleStart
    val numbers = mutableListOf("one", "two", "three")
    with(numbers) {
        println("'with' is called with argument $this")
        println("It contains $size elements")
    }
//sampleEnd
}
```

还可以用 `with` 来引入一个辅助对象，其属性或函数用于计算一个值。

```
fun main() {
//sampleStart
```

```
    val numbers = mutableListOf("one", "two", "three")
    val firstAndLast = with(numbers) {
        "The first element is ${first()}," +
        " the last element is ${last()}"
    }
    println(firstAndLast)
//sampleEnd
}
```

run

- 上下文对象 作为接收者（ `this` ）来访问。
- 返回值 是 lambda 表达式结果。

`run` does the same as `with` but it is implemented as an extension function. So like `let`, you can call it on the context object using dot notation.

当 lambda 表达式同时初始化对象并计算返回值时， `run` 很有用。

```
class MultiportService(var url: String, var port: Int) {
    fun prepareRequest(): String = "Default request"
    fun query(request: String): String = "Result for query '$request'"
}

fun main() {
//sampleStart
    val service = MultiportService("https://example.kotlinlang.org", 80)

    val result = service.run {
        port = 8080
        query(prepareRequest() + " to port $port")
    }

    // 同样的代码如果用 let() 函数来写:
    val letResult = service.let {
        it.port = 8080
        it.query(it.prepareRequest() + " to port ${it.port}")
    }
//sampleEnd
    println(result)
    println(letResult)
}
```

You can also invoke `run` as a non-extension function. The non-extension variant of `run` has no context object, but it still returns the lambda result. 非扩展 `run` 可以在需要表达式的地方执行一个由多个语句组成的块。 In code, non-extension `run` can be read as "*run the code block and compute the result.*"

```
fun main() {
//sampleStart
    val hexNumberRegex = run {
        val digits = "0-9"
        val hexDigits = "A-Fa-f"
        val sign = "+-"

        Regex("[$sign]?[$digits$hexDigits]+")
    }

    for (match in hexNumberRegex.findAll("+123 -FFFF !%*& 88 XYZ")) {
        println(match.value)
    }
//sampleEnd
}
```

apply

- 上下文对象 作为接收者（ `this` ）来访问。
- 返回值 是上下文对象本身。

As `apply` returns the context object itself, we recommend that 对于不返回值且主要在接收者（ `this` ）对象的成员上运行的代码块使用它。 `apply` 最常见的使用场景是用于对象配置。这样的调用可以理解为“将<!--- ->以下赋值操作应用于对象”。

```
data class Person(var name: String, var age: Int = 0, var city: String = "")

fun main() {
//sampleStart
    val adam = Person("Adam").apply {
        age = 32
        city = "London"
    }
    println(adam)
//sampleEnd
}
```

Another use case for `apply` is to include `apply` in multiple call chains for more complex processing.

also

- 上下文对象作为 lambda 表达式的参数（ `it` ）来访问。
- 返回值是上下文对象本身。

`also` 对于执行一些将上下文对象作为参数的操作很有用。 对于需要引用对象而不是其属性与函数的操作，或者不想屏蔽来自外部作用域的 `this` 引用时，请使用 `also`。

当你在代码中看到 `also` 时，可以将其理解为“并且用该对象执行以下操作”。

```
fun main() {
//sampleStart
    val numbers = mutableListOf("one", "two", "three")
    numbers
        .also { println("The list elements before adding new one: $it") }
        .add("four")
//sampleEnd
}
```

takeIf 与 takeUnless

除了作用域函数外，标准库还包含函数 `takeIf` 及 `takeUnless`。 这俩函数让你可以将对象状态检查嵌入到调用链中。

当在对象上调用谓词时，若满足给定的谓词，则 `takeIf` 返回此对象。 否则返回 `null`。因此，`takeIf` 是单个对象的过滤函数。

`takeUnless` has the opposite logic of `takeIf`. When called on an object along with a predicate, `takeUnless` returns `null` if it satisfies the given predicate. Otherwise, it returns the object.

When using `takeIf` or `takeUnless`, the object is available as a lambda argument (`it`).

```
import kotlin.random.*

fun main() {
//sampleStart
    val number = Random.nextInt(100)

    val evenOrNull = number.takeIf { it % 2 == 0 }
    val oddOrNull = number.takeUnless { it % 2 == 0 }
    println("even: $evenOrNull, odd: $oddOrNull")
//sampleEnd
}
```

> 当在 `takeIf` 及 `takeUnless` 之后链式调用其他函数，不要忘记执行空检查或使用安全调用 (`?.`)，因为他们的返回值是可为空的。
>
> 

```
fun main() {
//sampleStart
```

```
    val str = "Hello"
    val caps = str.takeIf { it.isNotEmpty() }?.uppercase()
   //val caps = str.takeIf { it.isNotEmpty() }.uppercase() // 编译错误
    println(caps)
//sampleEnd
}
```

`takeIf` 及 `takeUnless` 与作用域函数结合使用时特别有用。 例如，可以将 `takeIf` 及 `takeUnless` 与 `let` 链式调用，以便在与给定谓词匹配的对象上运行代码块。为此， 请在对象上调用 `takeIf` ，然后通过安全调用（ `?.` ）调用 `let` 。对于与谓词不匹配的对象，`takeIf` 返回 `null` ，并且不调用 `let` 。

```
fun main() {
    //sampleStart
    fun displaySubstringPosition(input: String, sub: String) {
        input.indexOf(sub).takeIf { it >= 0 }?.let {
            println("The substring $sub is found in $input.")
            println("Its start position is $it.")
        }
    }

    displaySubstringPosition("010000011", "11")
    displaySubstringPosition("010000011", "12")
//sampleEnd
}
```

For comparison, below is an example of how the same function can be written without using `takeIf` or scope functions:

```
fun main() {
    //sampleStart
    fun displaySubstringPosition(input: String, sub: String) {
        val index = input.indexOf(sub)
        if (index >= 0) {
            println("The substring $sub is found in $input.")
            println("Its start position is $index.")
        }
    }

    displaySubstringPosition("010000011", "11")
    displaySubstringPosition("010000011", "12")
//sampleEnd
}
```

时间度量

The Kotlin standard library gives you the tools to calculate and measure time in different units. Accurate time measurement is important for activities like:

- Managing threads or processes
- Collecting statistics
- Detecting timeouts
- Debugging

By default, time is measured using a monotonic time source, but other time sources can be configured. For more information, see Create time source.

Calculate duration

To represent an amount of time, the standard library has the `Duration` class. A `Duration` can be expressed in the following units from the `DurationUnit` enum class:

- `NANOSECONDS`
- `MICROSECONDS`
- `MILLISECONDS`
- `SECONDS`
- `MINUTES`
- `HOURS`
- `DAYS`

A `Duration` can be positive, negative, zero, positive infinity, or negative infinity.

Create duration

To create a `Duration`, use the extension properties available for `Int`, `Long`, and `Double` types: `nanoseconds`, `microseconds`, `milliseconds`, `seconds`, `minutes`, `hours`, and `days`.

> Days refer to periods of 24 hours. They are not calendar days.
>
> 

For example:

```
import kotlin.time.*
import kotlin.time.Duration.Companion.nanoseconds
import kotlin.time.Duration.Companion.milliseconds
```

```
import kotlin.time.Duration.Companion.seconds
import kotlin.time.Duration.Companion.minutes
import kotlin.time.Duration.Companion.days

fun main() {
//sampleStart
    val fiveHundredMilliseconds: Duration = 500.milliseconds
    val zeroSeconds: Duration = 0.seconds
    val tenMinutes: Duration = 10.minutes
    val negativeNanosecond: Duration = (-1).nanoseconds
    val infiniteDays: Duration = Double.POSITIVE_INFINITY.days
    val negativeInfiniteDays: Duration = Double.NEGATIVE_INFINITY.days

    println(fiveHundredMilliseconds) // 500ms
    println(zeroSeconds)             // 0s
    println(tenMinutes)              // 10m
    println(negativeNanosecond)      // -1ns
    println(infiniteDays)            // Infinity
    println(negativeInfiniteDays)    // -Infinity
//sampleEnd
}
```

You can also perform basic arithmetic with `Duration` objects:

```
import kotlin.time.*
import kotlin.time.Duration.Companion.seconds

fun main() {
//sampleStart
    val fiveSeconds: Duration = 5.seconds
    val thirtySeconds: Duration = 30.seconds

    println(fiveSeconds + thirtySeconds)
    // 35s
    println(thirtySeconds - fiveSeconds)
    // 25s
    println(fiveSeconds * 2)
    // 10s
    println(thirtySeconds / 2)
    // 15s
    println(thirtySeconds / fiveSeconds)
    // 6.0
    println(-thirtySeconds)
    // -30s
    println((-thirtySeconds).absoluteValue)
    // 30s
//sampleEnd
}
```

Get string representation

It can be useful to have a string representation of a `Duration` so that you can print, serialize, transfer, or store it.

To get a string representation, use the `.toString()` function. By default, the time is reported using each unit that is present. For example: `1h 0m 45.677s` or `-(6d 5h 5m 28.284s)`

To configure the output, use the `.toString()` function with your desired `DurationUnit` and number of decimal places as function parameters:

```
import kotlin.time.Duration
import kotlin.time.Duration.Companion.milliseconds
import kotlin.time.DurationUnit

fun main() {
//sampleStart
    // Print in seconds with 2 decimal places
    println(5887.milliseconds.toString(DurationUnit.SECONDS, 2))
    // 5.89s
//sampleEnd
}
```

To get an ISO-8601-compatible string, use the `toIsoString()` function:

```
import kotlin.time.Duration.Companion.seconds

fun main() {
//sampleStart
    println(86420.seconds.toIsoString()) // PT24H0M20S
//sampleEnd
}
```

Convert duration

To convert your `Duration` into a different `DurationUnit`, use the following properties:

- `inWholeNanoseconds`
- `inWholeMicroseconds`
- `inWholeSeconds`
- `inWholeMinutes`
- `inWholeHours`
- `inWholeDays`

For example:

```
import kotlin.time.Duration
import kotlin.time.Duration.Companion.minutes
```

```
fun main() {
//sampleStart
    val thirtyMinutes: Duration = 30.minutes
    println(thirtyMinutes.inWholeSeconds)
    // 1800
//sampleEnd
}
```

Alternatively, you can use your desired `DurationUnit` as a function parameter in the following extension functions:

- `.toInt()`
- `.toDouble()`
- `.toLong()`

For example:

```
import kotlin.time.Duration.Companion.seconds
import kotlin.time.DurationUnit

fun main() {
//sampleStart
    println(270.seconds.toDouble(DurationUnit.MINUTES))
    // 4.5
//sampleEnd
}
```

Compare duration

To check if `Duration` objects are equal, use the equality operator (`==`):

```
import kotlin.time.Duration
import kotlin.time.Duration.Companion.hours
import kotlin.time.Duration.Companion.minutes

fun main() {
//sampleStart
    val thirtyMinutes: Duration = 30.minutes
    val halfHour: Duration = 0.5.hours
    println(thirtyMinutes == halfHour)
    // true
//sampleEnd
}
```

To compare `Duration` objects, use the comparison operators (`<` , `>`):

```
import kotlin.time.Duration.Companion.microseconds
import kotlin.time.Duration.Companion.nanoseconds

fun main() {
//sampleStart
    println(3000.microseconds < 25000.nanoseconds)
    // false
//sampleEnd
}
```

Break duration into components

To break down a `Duration` into its time components and perform a further action, use the overload of the `toComponents()` function. Add your desired action as a function or lambda expression as a function parameter.

For example:

```
import kotlin.time.Duration
import kotlin.time.Duration.Companion.minutes

fun main() {
//sampleStart
    val thirtyMinutes: Duration = 30.minutes
    println(thirtyMinutes.toComponents { hours, minutes, _, _ -> "${hours}h:${minutes}
m" })
    // 0h:30m
//sampleEnd
}
```

In this example, the lambda expression has `hours` and `minutes` as function parameters with underscores (`_`) for the unused `seconds` and `nanoseconds` parameters. The expression returns a concatenated string using string templates to get the desired output format of `hours` and `minutes` .

Measure time

To track the passage of time, the standard library provides tools so that you can easily:

- Measure the time taken to execute some code with your desired time unit.
- Mark a moment in time.
- Compare and subtract two moments in time.
- Check how much time has passed since a specific moment in time.
- Check whether the current time has passed a specific moment in time.

Measure code execution time

To measure the time taken to execute a block of code, use the `measureTime` inline function:

```
import kotlin.time.measureTime

fun main() {
//sampleStart
    val timeTaken = measureTime {
        Thread.sleep(100)
    }
    println(timeTaken) // e.g. 103 ms
//sampleEnd
}
```

To measure the time taken to execute a block of code **and** return the value of the block of code, use inline function `measureTimedValue`.

For example:

```
import kotlin.time.measureTimedValue

fun main() {
//sampleStart
    val (value, timeTaken) = measureTimedValue {
        Thread.sleep(100)
        42
    }
    println(value)     // 42
    println(timeTaken) // e.g. 103 ms
//sampleEnd
}
```

By default, both functions use a monotonic time source.

Mark moments in time

To mark a specific moment in time, use the `TimeSource` interface and the `markNow()` function to create a `TimeMark`:

```
import kotlin.time.*

fun main() {
   val timeSource = TimeSource.Monotonic
   val mark = timeSource.markNow()
}
```

Measure differences in time

To measure differences between `TimeMark` objects from the same time source, use the subtraction operator (`-`).

To compare `TimeMark` objects from the same time source, use the comparison operators (`<` , `>`).

For example:

```
import kotlin.time.*

fun main() {
//sampleStart
   val timeSource = TimeSource.Monotonic
   val mark1 = timeSource.markNow()
   Thread.sleep(500) // Sleep 0.5 seconds.
   val mark2 = timeSource.markNow()

   repeat(4) { n ->
       val mark3 = timeSource.markNow()
       val elapsed1 = mark3 - mark1
       val elapsed2 = mark3 - mark2

       println("Measurement 1.${n + 1}: elapsed1=$elapsed1, elapsed2=$elapsed2, diff=$
{elapsed1 - elapsed2}")
   }

   println(mark2 > mark1) // This is true, as mark2 was captured later than mark1.
   // true
//sampleEnd
}
```

To check if a deadline has passed or a timeout has been reached, use the `hasPassedNow()` and `hasNotPassedNow()` extension functions:

```
import kotlin.time.*
import kotlin.time.Duration.Companion.seconds

fun main() {
//sampleStart
   val timeSource = TimeSource.Monotonic
   val mark1 = timeSource.markNow()
   val fiveSeconds: Duration = 5.seconds
   val mark2 = mark1 + fiveSeconds

   // It hasn't been 5 seconds yet
   println(mark2.hasPassedNow())
   // false

   // Wait six seconds
```

```
    Thread.sleep(6000)
    println(mark2.hasPassedNow())
    // true

//sampleEnd
}
```

Time sources

By default, time is measured using a monotonic time source. Monotonic time sources only move forward and are not affected by variations like timezones. An alternative to monotonic time is elapsed real time which is also known as wall-clock time. Elapsed real time is measured relative to another point in time.

Default time sources per platform

This table explains the default source of monotonic time for each platform:

Platform	Source
Kotlin/JVM	`System.nanoTime()`
Kotlin/JS (Node.js)	`process.hrtime()`
Kotlin/JS (browser)	`window.performance.now()` or `Date.now()`
Kotlin/Native	`std::chrono::high_resolution_clock` or `std::chrono::steady_clock`

Create time source

There are some cases where you might want to use a different time source. For example in Android, `System.nanoTime()` only counts time while the device is active. It loses track of time when the device enters deep sleep. To keep track of time while the device is in deep sleep, you can create a time source that uses `SystemClock.elapsedRealtimeNanos()`):

```
object RealtimeMonotonicTimeSource : AbstractLongTimeSource(DurationUnit.NANOSECONDS)
{
    override fun read(): Long = SystemClock.elapsedRealtimeNanos()
}
```

Then you can use your time source to make time measurements:

```
fun main() {
    val elapsed: Duration = RealtimeMonotonicTimeSource.measureTime {
```

```
            Thread.sleep(100)
        }
        println(elapsed) // e.g. 103 ms
    }
```

For more information about the `kotlin.time` package, see our standard library API reference.

官方库

- 协程（kotlinx.coroutines）
  - 协程指南
  - 协程基础
  - Kotlin 协程与通道介绍↗
  - 取消与超时
  - 组合挂起函数
  - 协程上下文与调度器
  - 异步流
  - 通道
  - 协程异常处理
  - 共享的可变状态与并发
  - select 表达式（实验性的）
  - 使用 IntelliJ IDEA 调试协程——教程
  - 使用 IntelliJ IDEA 调试 Kotlin Flow——教程
- 序列化（kotlinx.serialization）
- Kotlin Metadata JVM（kotlin-metadata-jvm）
- Lincheck（kotlinx.lincheck）
  - Lincheck 指南
  - 用 Lincheck 编写第一个测试
  - 测试策略
  - 操作参数
  - 数据结构约束
  - 进度保证
  - 时序规范
- Ktor↗

协程（kotlinx.coroutines）

- 协程指南
- 协程基础
- Kotlin 协程与通道介绍↗
- 取消与超时
- 组合挂起函数
- 协程上下文与调度器
- 异步流
- 通道
- 协程异常处理
- 共享的可变状态与并发
- select 表达式（实验性的）
- 使用 IntelliJ IDEA 调试协程——教程
- 使用 IntelliJ IDEA 调试 Kotlin Flow——教程

协程指南

Kotlin 仅在标准库中提供最基本底层 API 以便其他库能够利用协程。与许多其他具有类似功能的语言不同， `async` 与 `await` 在 Kotlin 中并不是关键字，甚至都不是标准库的一部分。此外，Kotlin 的 挂起函数 概念为异步操作提供了比 future 与 promise 更安全、更不易出错的抽象。

`kotlinx.coroutines` 是由 JetBrains 开发的功能丰富的协程库。它包含本指南中涵盖的很多启用高级协程的原语，包括 `launch` 、 `async` 等等。

本文是关于 `kotlinx.coroutines` 核心特性的指南，包含一系列示例，并分为不同的主题。

为了使用协程以及按照本指南中的示例演练，需要添加对 `kotlinx-coroutines-core` 模块的依赖，如项目中的 README 文件所述。

目录

- 协程基础
- Hands-on: Intro to coroutines and channels
- 取消与超时
- 组合挂起函数
- 协程上下文与调度器
- 异步流
- 通道
- 协程异常处理
- 共享的可变状态与并发
- Select 表达式（实验性的）
- Tutorial: Debug coroutines using IntelliJ IDEA
- Tutorial: Debug Kotlin Flow using IntelliJ IDEA

其他参考资料

- 使用协程进行 UI 编程指南
- 协程设计文档（KEEP）
- 完整的 kotlinx.coroutines API 参考文档
- Best practices for coroutines in Android
- Additional Android resources for Kotlin coroutines and flow

协程基础

这一部分包括基础的协程概念。

第一个协程程序

A *coroutine* is an instance of a suspendable computation. It is conceptually similar to a thread, in the sense that it takes a block of code to run that works concurrently with the rest of the code. However, a coroutine is not bound to any particular thread. It may suspend its execution in one thread and resume in another one.

Coroutines can be thought of as light-weight threads, but there is a number of important differences that make their real-life usage very different from threads.

Run the following code to get to your first working coroutine:

```
import kotlinx.coroutines.*

//sampleStart
fun main() = runBlocking { // this: CoroutineScope
    launch { // launch a new coroutine and continue
        delay(1000L) // non-blocking delay for 1 second (default time unit is ms)
        println("World!") // print after delay
    }
    println("Hello") // main coroutine continues while a previous one is delayed
}
//sampleEnd
```

> 可以在这里获取完整代码。
>
> 

代码运行的结果：

```
Hello
World!
```

Let's dissect what this code does.

launch is a *coroutine builder*. It launches a new coroutine concurrently with the rest of the code, which continues to work independently. That's why `Hello` has been printed first.

delay is a special *suspending function*. It *suspends* the coroutine for a specific time. Suspending a coroutine does not *block* the underlying thread, but allows other coroutines to run and use the underlying thread for their code.

runBlocking is also a coroutine builder that bridges the non-coroutine world of a regular `fun main()` and the code with coroutines inside of `runBlocking { ... }` curly braces. This is highlighted in an IDE by `this: CoroutineScope` hint right after the `runBlocking` opening curly brace.

If you remove or forget `runBlocking` in this code, you'll get an error on the launch call, since `launch` is declared only on the CoroutineScope:

```
Unresolved reference: launch
```

The name of `runBlocking` means that the thread that runs it (in this case — the main thread) gets *blocked* for the duration of the call, until all the coroutines inside `runBlocking { ... }` complete their execution. You will often see `runBlocking` used like that at the very top-level of the application and quite rarely inside the real code, as threads are expensive resources and blocking them is inefficient and is often not desired.

Structured concurrency

Coroutines follow a principle of **structured concurrency** which means that new coroutines can only be launched in a specific CoroutineScope which delimits the lifetime of the coroutine. The above example shows that runBlocking establishes the corresponding scope and that is why the previous example waits until `World!` is printed after a second's delay and only then exits.

In a real application, you will be launching a lot of coroutines. Structured concurrency ensures that they are not lost and do not leak. An outer scope cannot complete until all its children coroutines complete. Structured concurrency also ensures that any errors in the code are properly reported and are never lost.

Extract function refactoring

Let's extract the block of code inside `launch { ... }` into a separate function. When you perform "Extract function" refactoring on this code, you get a new function with the `suspend` modifier. This is your first *suspending function*. Suspending functions can be used inside coroutines just like regular functions, but their additional feature is that they can, in turn, use other suspending functions (like `delay` in this example) to *suspend* execution of a coroutine.

```
import kotlinx.coroutines.*

//sampleStart
```

```
fun main() = runBlocking { // this: CoroutineScope
    launch { doWorld() }
    println("Hello")
}

// this is your first suspending function
suspend fun doWorld() {
    delay(1000L)
    println("World!")
}
//sampleEnd
```

> 可以在这里获取完整代码。
>
> 

Scope builder

In addition to the coroutine scope provided by different builders, it is possible to declare your own scope using the coroutineScope builder. It creates a coroutine scope and does not complete until all launched children complete.

runBlocking and coroutineScope builders may look similar because they both wait for their body and all its children to complete. The main difference is that the runBlocking method *blocks* the current thread for waiting, while coroutineScope just suspends, releasing the underlying thread for other usages. Because of that difference, runBlocking is a regular function and coroutineScope is a suspending function.

You can use `coroutineScope` from any suspending function. For example, you can move the concurrent printing of `Hello` and `World` into a `suspend fun doWorld()` function:

```
import kotlinx.coroutines.*

//sampleStart
fun main() = runBlocking {
    doWorld()
}

suspend fun doWorld() = coroutineScope {  // this: CoroutineScope
    launch {
        delay(1000L)
        println("World!")
    }
    println("Hello")
}
//sampleEnd
```

> 可以在这里获取完整代码。
>
> 

This code also prints:

```
Hello
World!
```

Scope builder and concurrency

A coroutineScope builder can be used inside any suspending function to perform multiple concurrent operations. Let's launch two concurrent coroutines inside a `doWorld` suspending function:

```
import kotlinx.coroutines.*

//sampleStart
// Sequentially executes doWorld followed by "Done"
fun main() = runBlocking {
    doWorld()
    println("Done")
}

// Concurrently executes both sections
suspend fun doWorld() = coroutineScope { // this: CoroutineScope
    launch {
        delay(2000L)
        println("World 2")
    }
    launch {
        delay(1000L)
        println("World 1")
    }
    println("Hello")
}
//sampleEnd
```

> 可以在这里获取完整代码。
>
> 

Both pieces of code inside `launch { ... }` blocks execute *concurrently*, with `World 1` printed first, after a second from start, and `World 2` printed next, after two seconds from start. A coroutineScope in `doWorld` completes only after both are complete, so `doWorld` returns and allows `Done` string to be printed only after that:

```
Hello
World 1
World 2
Done
```

An explicit job

A launch coroutine builder returns a Job object that is a handle to the launched coroutine and can be used to explicitly wait for its completion. For example, you can wait for completion of the child coroutine and then print "Done" string:

```
import kotlinx.coroutines.*

fun main() = runBlocking {
//sampleStart
    val job = launch { // launch a new coroutine and keep a reference to its Job
        delay(1000L)
        println("World!")
    }
    println("Hello")
    job.join() // wait until child coroutine completes
    println("Done")
//sampleEnd
}
```

> 可以在这里获取完整代码。
>
> 

This code produces:

```
Hello
World!
Done
```

协程很轻量

Coroutines are less resource-intensive than JVM threads. Code that exhausts the JVM's available memory when using threads can be expressed using coroutines without hitting resource limits. For example, the following code launches 50,000 distinct coroutines that each waits 5 seconds and then prints a period ('.') while consuming very little memory:

```
import kotlinx.coroutines.*
```

```
fun main() = runBlocking {
    repeat(50_000) { // 启动大量的协程
        launch {
            delay(5000L)
            print(".")
        }
    }
}
```

> 可以在这里获取完整代码。
>
> 

If you write the same program using threads (remove `runBlocking`, replace `launch` with `thread`, and replace `delay` with `Thread.sleep`), it will consume a lot of memory. Depending on your operating system, JDK version, and its settings, it will either throw an out-of-memory error or start threads slowly so that there are never too many concurrently running threads.

Kotlin 协程与通道介绍

Kotlin 协程与通道介绍

取消与超时

这一部分包含了协程的取消与超时。

取消协程的执行

在一个长时间运行的应用程序中，你也许需要对你的后台协程进行细粒度的控制。 比如说，一个用户也许关闭了一个启动了协程的界面，那么现在协程的执行结果已经不再被需要了，这时，它应该是可以被取消的。 该 launch 函数返回了一个可以被用来取消运行中的协程的 Job：

```
import kotlinx.coroutines.*

fun main() = runBlocking {
//sampleStart
    val job = launch {
        repeat(1000) { i ->
            println("job: I'm sleeping $i ...")
            delay(500L)
        }
    }
    delay(1300L) // 延迟一段时间
    println("main: I'm tired of waiting!")
    job.cancel() // 取消该作业
    job.join() // 等待作业执行结束
    println("main: Now I can quit.")
//sampleEnd
}
```

> 可以在这里获取完整代码。
>
> 

程序执行后的输出如下：

```
job: I'm sleeping 0 ...
job: I'm sleeping 1 ...
job: I'm sleeping 2 ...
main: I'm tired of waiting!
main: Now I can quit.
```

一旦 main 函数调用了 `job.cancel`，我们在其它的协程中就看不到任何输出，因为它被取消了。 这里也有一个可以使 Job 挂起的函数 cancelAndJoin 它合并了对 cancel 以及 join 的调用。

取消是协作的

协程的取消是 协作 的。一段协程代码必须协作才能被取消。 所有 `kotlinx.coroutines` 中的挂起函数都是 可被取消的 。它们检查协程的取消， 并在取消时抛出 CancellationException。然而，如果协程正在执行计算任务，并且没有检查取消的话，那么它是不能被取消的，就如如下示例代码所示：

```
import kotlinx.coroutines.*

fun main() = runBlocking {
//sampleStart
    val startTime = System.currentTimeMillis()
    val job = launch(Dispatchers.Default) {
        var nextPrintTime = startTime
        var i = 0
        while (i < 5) { // 一个执行计算的循环，只是为了占用 CPU
            // 每秒打印消息两次
            if (System.currentTimeMillis() >= nextPrintTime) {
                println("job: I'm sleeping ${i++} ...")
                nextPrintTime += 500L
            }
        }
    }
    delay(1300L) // 等待一段时间
    println("main: I'm tired of waiting!")
    job.cancelAndJoin() // 取消一个作业并且等待它结束
    println("main: Now I can quit.")
//sampleEnd
}
```

> 可以在这里获取完整代码。
>
> 

运行示例代码，并且我们可以看到它连续打印出了“I'm sleeping”，甚至在调用取消后， 作业仍然执行了五次循环迭代并运行到了它结束为止。

The same problem can be observed by catching a CancellationException and not rethrowing it:

```
import kotlinx.coroutines.*

fun main() = runBlocking {
//sampleStart
    val job = launch(Dispatchers.Default) {
        repeat(5) { i ->
            try {
                // print a message twice a second
                println("job: I'm sleeping $i ...")
                delay(500)
```

```
            } catch (e: Exception) {
                // log the exception
                println(e)
            }
        }
    }
    delay(1300L) // delay a bit
    println("main: I'm tired of waiting!")
    job.cancelAndJoin() // cancels the job and waits for its completion
    println("main: Now I can quit.")
//sampleEnd
}
```

> You can get the full code here.
>
> 

While catching `Exception` is an anti-pattern, this issue may surface in more subtle ways, like when using the `runCatching` function, which does not rethrow CancellationException.

使计算代码可取消

我们有两种方法来使执行计算的代码可以被取消。第一种方法是定期调用挂起函数来检查取消。对于这种目的 yield 是一个好的选择。 另一种方法是显式的检查取消状态。让我们试试第二种方法。

将前一个示例中的 `while (i < 5)` 替换为 `while (isActive)` 并重新运行它。

```
import kotlinx.coroutines.*

fun main() = runBlocking {
//sampleStart
    val startTime = System.currentTimeMillis()
    val job = launch(Dispatchers.Default) {
        var nextPrintTime = startTime
        var i = 0
        while (isActive) { // 可以被取消的计算循环
            // 每秒打印消息两次
            if (System.currentTimeMillis() >= nextPrintTime) {
                println("job: I'm sleeping ${i++} ...")
                nextPrintTime += 500L
            }
        }
    }
    delay(1300L) // 等待一段时间
    println("main: I'm tired of waiting!")
    job.cancelAndJoin() // 取消该作业并等待它结束
    println("main: Now I can quit.")
```

```
//sampleEnd
}
```

> 可以在这里获取完整代码。
>
> 

你可以看到，现在循环被取消了。isActive 是一个可以被使用在 CoroutineScope 中的扩展属性。

在 `finally` 中释放资源

我们通常使用如下的方法处理在被取消时抛出 CancellationException 的可被取消的挂起函数。比如说，`try {……} finally {……}` 表达式以及 Kotlin 的 `use` 函数一般在协程被取消的时候执行它们的终结动作：

```
import kotlinx.coroutines.*

fun main() = runBlocking {
//sampleStart
    val job = launch {
        try {
            repeat(1000) { i ->
                println("job: I'm sleeping $i ...")
                delay(500L)
            }
        } finally {
            println("job: I'm running finally")
        }
    }
    delay(1300L) // 延迟一段时间
    println("main: I'm tired of waiting!")
    job.cancelAndJoin() // 取消该作业并且等待它结束
    println("main: Now I can quit.")
//sampleEnd
}
```

> 可以在这里获取完整代码。
>
> 

join 和 cancelAndJoin 等待了所有的终结动作执行完毕， 所以运行示例得到了下面的输出：

```
job: I'm sleeping 0 ...
job: I'm sleeping 1 ...
job: I'm sleeping 2 ...
main: I'm tired of waiting!
```

```
job: I'm running finally
main: Now I can quit.
```

运行不能取消的代码块

在前一个例子中任何尝试在 `finally` 块中调用挂起函数的行为都会抛出 CancellationException，因为这里持续运行的代码是可以被取消的。通常，这并不是一个问题，所有良好的关闭操作（关闭一个文件、取消一个作业、或是关闭任何一种通信通道）通常都是非阻塞的，并且不会调用任何挂起函数。然而，在真实的案例中，当你需要挂起一个被取消的协程，你可以将相应的代码包装在 `withContext(NonCancellable) {……}` 中，并使用 withContext 函数以及 NonCancellable 上下文，见如下示例所示：

```
import kotlinx.coroutines.*

fun main() = runBlocking {
//sampleStart
    val job = launch {
        try {
            repeat(1000) { i ->
                println("job: I'm sleeping $i ...")
                delay(500L)
            }
        } finally {
            withContext(NonCancellable) {
                println("job: I'm running finally")
                delay(1000L)
                println("job: And I've just delayed for 1 sec because I'm non-cancella
ble")
            }
        }
    }
    delay(1300L) // 延迟一段时间
    println("main: I'm tired of waiting!")
    job.cancelAndJoin() // 取消该作业并等待它结束
    println("main: Now I can quit.")
//sampleEnd
}
```

> 可以在这里获取完整代码。
>
> 

超时

在实践中绝大多数取消一个协程的理由是它有可能超时。 当你手动追踪一个相关 Job 的引用并启动了一个单独的协程在延迟后取消追踪，这里已经准备好使用 withTimeout 函数来做这件事。 来看看示例代码：

```
import kotlinx.coroutines.*

fun main() = runBlocking {
//sampleStart
    withTimeout(1300L) {
        repeat(1000) { i ->
            println("I'm sleeping $i ...")
            delay(500L)
        }
    }
//sampleEnd
}
```

> 可以在这里获取完整代码。
>
> 

运行后得到如下输出：

```
I'm sleeping 0 ...
I'm sleeping 1 ...
I'm sleeping 2 ...
Exception in thread "main" kotlinx.coroutines.TimeoutCancellationException: Timed out
waiting for 1300 ms
```

withTimeout 抛出了 `TimeoutCancellationException`，它是 CancellationException 的子类。 我们之前没有在控制台上看到堆栈跟踪信息的打印。这是因为在被取消的协程中 `CancellationException` 被认为是协程执行结束的正常原因。 然而，在这个示例中我们在 `main` 函数中正确地使用了 `withTimeout`。

由于取消只是一个例外，所有的资源都使用常用的方法来关闭。 如果你需要做一些各类使用超时的特别的额外操作，可以使用类似 withTimeout 的 withTimeoutOrNull 函数，并把这些会超时的代码包装在 `try {...} catch (e: TimeoutCancellationException) {...}` 代码块中，而 withTimeoutOrNull 通过返回 `null` 来进行超时操作，从而替代抛出一个异常：

```
import kotlinx.coroutines.*

fun main() = runBlocking {
//sampleStart
    val result = withTimeoutOrNull(1300L) {
        repeat(1000) { i ->
            println("I'm sleeping $i ...")
            delay(500L)
```

```
        }
        "Done" // 在它运行得到结果之前取消它
    }
    println("Result is $result")
//sampleEnd
}
```

> 可以在这里获取完整代码。
>
> 

运行这段代码时不再抛出异常：

```
I'm sleeping 0 ...
I'm sleeping 1 ...
I'm sleeping 2 ...
Result is null
```

Asynchronous timeout and resources

The timeout event in withTimeout is asynchronous with respect to the code running in its block and may happen at any time, even right before the return from inside of the timeout block. Keep this in mind if you open or acquire some resource inside the block that needs closing or release outside of the block.

For example, here we imitate a closeable resource with the `Resource` class that simply keeps track of how many times it was created by incrementing the `acquired` counter and decrementing the counter in its `close` function. Now let us create a lot of coroutines, each of which creates a `Resource` at the end of the `withTimeout` block and releases the resource outside the block. We add a small delay so that it is more likely that the timeout occurs right when the `withTimeout` block is already finished, which will cause a resource leak.

```
import kotlinx.coroutines.*

//sampleStart
var acquired = 0

class Resource {
    init { acquired++ } // Acquire the resource
    fun close() { acquired-- } // Release the resource
}

fun main() {
    runBlocking {
        repeat(10_000) { // Launch 10K coroutines
            launch {
```

```
                val resource = withTimeout(60) { // Timeout of 60 ms
                    delay(50) // Delay for 50 ms
                    Resource() // Acquire a resource and return it from withTimeout bl
ock
                }
                resource.close() // Release the resource
            }
        }
    }
    // Outside of runBlocking all coroutines have completed
    println(acquired) // Print the number of resources still acquired
}
//sampleEnd
```

> You can get the full code here.
>
> 

If you run the above code, you'll see that it does not always print zero, though it may depend on the timings of your machine. You may need to tweak the timeout in this example to actually see non-zero values.

> Note that incrementing and decrementing `acquired` counter here from 10K coroutines is completely thread-safe, since it always happens from the same thread, the one used by `runBlocking`. More on that will be explained in the chapter on coroutine context.
>
> 

To work around this problem you can store a reference to the resource in a variable instead of returning it from the `withTimeout` block.

```
import kotlinx.coroutines.*

var acquired = 0

class Resource {
    init { acquired++ } // Acquire the resource
    fun close() { acquired-- } // Release the resource
}

fun main() {
//sampleStart
    runBlocking {
        repeat(10_000) { // Launch 10K coroutines
            launch {
                var resource: Resource? = null // Not acquired yet
                try {
                    withTimeout(60) { // Timeout of 60 ms
                        delay(50) // Delay for 50 ms
                        resource = Resource() // Store a resource to the variable if a
```

```
cquired
                    }
                    // We can do something else with the resource here
                } finally {
                    resource?.close() // Release the resource if it was acquired
                }
            }
        }
    }
    // Outside of runBlocking all coroutines have completed
    println(acquired) // Print the number of resources still acquired
//sampleEnd
}
```

> You can get the full code here.
>
> 

This example always prints zero. Resources do not leak.

组合挂起函数

本节介绍了将挂起函数组合的各种方法。

默认顺序调用

假设我们在不同的地方定义了两个进行某种调用远程服务或者进行计算的挂起函数。我们只假设它们都是有用的，但是实际上它们在这个示例中只是为了该目的而延迟了一秒钟：

```
suspend fun doSomethingUsefulOne(): Int {
    delay(1000L) // 假设我们在这里做了一些有用的事
    return 13
}

suspend fun doSomethingUsefulTwo(): Int {
    delay(1000L) // 假设我们在这里也做了一些有用的事
    return 29
}
```

如果需要按 顺序 调用它们，我们接下来会做什么——首先调用 `doSomethingUsefulOne` 接下来调用 `doSomethingUsefulTwo`，并且计算它们结果的和吗？ 实际上，如果我们要根据第一个函数的结果来决定是否我们需要调用第二个函数或者决定如何调用它时，我们就会这样做。

我们使用普通的顺序来进行调用，因为这些代码是运行在协程中的，只要像常规的代码一样 顺序 都是默认的。下面的示例展示了测量执行两个挂起函数所需要的总时间：

```
import kotlinx.coroutines.*
import kotlin.system.*

fun main() = runBlocking<Unit> {
//sampleStart
    val time = measureTimeMillis {
        val one = doSomethingUsefulOne()
        val two = doSomethingUsefulTwo()
        println("The answer is ${one + two}")
    }
    println("Completed in $time ms")
//sampleEnd
}

suspend fun doSomethingUsefulOne(): Int {
    delay(1000L) // 假设我们在这里做了些有用的事
    return 13
}
```

```
suspend fun doSomethingUsefulTwo(): Int {
    delay(1000L) // 假设我们在这里也做了一些有用的事
    return 29
}
```

> 可以在这里获取完整代码。
>
> 

它的打印输出如下：

```
The answer is 42
Completed in 2017 ms
```

使用 async 并发

如果 `doSomethingUsefulOne` 与 `doSomethingUsefulTwo` 之间没有依赖，并且我们想更快的得到结果，让它们进行 并发 吗？这就是 async 可以帮助我们的地方。

在概念上，async 就类似于 launch。它启动了一个单独的协程，这是一个轻量级的线程并与其它所有的协程一起并发的工作。不同之处在于 `launch` 返回一个 Job 并且不附带任何结果值，而 `async` 返回一个 Deferred —— 一个轻量级的非阻塞 future， 这代表了一个将会在稍后提供结果的 promise。你可以使用 `.await()` 在一个延期的值上得到它的最终结果， 但是 `Deferred` 也是一个 `Job`，所以如果需要的话，你可以取消它。

```
import kotlinx.coroutines.*
import kotlin.system.*

fun main() = runBlocking<Unit> {
//sampleStart
    val time = measureTimeMillis {
        val one = async { doSomethingUsefulOne() }
        val two = async { doSomethingUsefulTwo() }
        println("The answer is ${one.await() + two.await()}")
    }
    println("Completed in $time ms")
//sampleEnd
}

suspend fun doSomethingUsefulOne(): Int {
    delay(1000L) // 假设我们在这里做了些有用的事
    return 13
}

suspend fun doSomethingUsefulTwo(): Int {
    delay(1000L) // 假设我们在这里也做了些有用的事
```

```
    return 29
}
```

> 可以在这里获取完整代码。
>
> 

它的打印输出如下：

```
The answer is 42
Completed in 1017 ms
```

这里快了两倍，因为两个协程并发执行。 请注意，使用协程进行并发总是显式的。

惰性启动的 async

可选的，async 可以通过将 `start` 参数设置为 CoroutineStart.LAZY 而变为惰性的。 在这个模式下，只有结果通过 await 获取的时候协程才会启动，或者在 `Job` 的 start 函数调用的时候。运行下面的示例：

```
import kotlinx.coroutines.*
import kotlin.system.*

fun main() = runBlocking<Unit> {
//sampleStart
    val time = measureTimeMillis {
        val one = async(start = CoroutineStart.LAZY) { doSomethingUsefulOne() }
        val two = async(start = CoroutineStart.LAZY) { doSomethingUsefulTwo() }
        // 执行一些计算
        one.start() // 启动第一个
        two.start() // 启动第二个
        println("The answer is ${one.await() + two.await()}")
    }
    println("Completed in $time ms")
//sampleEnd
}

suspend fun doSomethingUsefulOne(): Int {
    delay(1000L) // 假设我们在这里做了些有用的事
    return 13
}

suspend fun doSomethingUsefulTwo(): Int {
    delay(1000L) // 假设我们在这里也做了些有用的事
    return 29
}
```

> 可以在这里获取完整代码。
>
> 

它的打印输出如下：

```
The answer is 42
Completed in 1017 ms
```

因此，在先前的例子中这里定义的两个协程没有执行，但是控制权在于程序员准确的在开始执行时调用 start。我们首先 调用 `one`，然后调用 `two`，接下来等待这个协程执行完毕。

注意，如果我们只是在 `println` 中调用 await，而没有在单独的协程中调用 start，这将会导致顺序行为，直到 await 启动该协程 执行并等待至它结束，这并不是惰性的预期用例。 在计算一个值涉及挂起函数时，这个 `async(start = CoroutineStart.LAZY)` 的用例用于替代标准库中的 `lazy` 函数。

async 风格的函数

> This programming style with async functions is provided here only for illustration, because it is a popular style in other programming languages. Using this style with Kotlin coroutines is **strongly discouraged** for the reasons explained below.
>
> 

We can define async-style functions that invoke `doSomethingUsefulOne` and `doSomethingUsefulTwo` *asynchronously* using the async coroutine builder using a GlobalScope reference to opt-out of the structured concurrency. We name such functions with the "...Async" suffix to highlight the fact that they only start asynchronous computation and one needs to use the resulting deferred value to get the result.

> GlobalScope is a delicate API that can backfire in non-trivial ways, one of which will be explained below, so you must explicitly opt-in into using `GlobalScope` with `@OptIn(DelicateCoroutinesApi::class)`.
>
> 

```
// somethingUsefulOneAsync 函数的返回值类型是 Deferred<Int>
@OptIn(DelicateCoroutinesApi::class)
fun somethingUsefulOneAsync() = GlobalScope.async {
    doSomethingUsefulOne()
}

// somethingUsefulTwoAsync 函数的返回值类型是 Deferred<Int>
@OptIn(DelicateCoroutinesApi::class)
```

```
fun somethingUsefulTwoAsync() = GlobalScope.async {
    doSomethingUsefulTwo()
}
```

注意，这些 `xxxAsync` 函数不是 挂起 函数。它们可以在任何地方使用。 然而，它们总是在调用它们的代码中意味着异步（这里的意思是 并发 ）执行。

下面的例子展示了它们在协程的外面是如何使用的：

```
import kotlinx.coroutines.*
import kotlin.system.*

//sampleStart
// 注意，在这个示例中我们在 `main` 函数的右边没有加上 `runBlocking`
fun main() {
    val time = measureTimeMillis {
        // 我们可以在协程外面启动异步执行
        val one = somethingUsefulOneAsync()
        val two = somethingUsefulTwoAsync()
        // 但是等待结果必须调用其它的挂起或者阻塞
        // 当我们等待结果的时候，这里我们使用 `runBlocking { …… }` 来阻塞主线程
        runBlocking {
            println("The answer is ${one.await() + two.await()}")
        }
    }
    println("Completed in $time ms")
}
//sampleEnd

@OptIn(DelicateCoroutinesApi::class)
fun somethingUsefulOneAsync() = GlobalScope.async {
    doSomethingUsefulOne()
}

@OptIn(DelicateCoroutinesApi::class)
fun somethingUsefulTwoAsync() = GlobalScope.async {
    doSomethingUsefulTwo()
}

suspend fun doSomethingUsefulOne(): Int {
    delay(1000L) // 假设我们在这里做了些有用的事
    return 13
}

suspend fun doSomethingUsefulTwo(): Int {
    delay(1000L) // 假设我们在这里也做了些有用的事
    return 29
}
```

> 可以在这里获取完整代码。



考虑一下如果 `val one = somethingUsefulOneAsync()` 这一行和 `one.await()` 表达式这里在代码中有逻辑错误， 并且程序抛出了异常以及程序在操作的过程中中止，将会发生什么。 通常情况下，一个全局的异常处理者会捕获这个异常，将异常打印成日记并报告给开发者，但是反之该程序将会继续执行其它操作。但是这里我们的 `somethingUsefulOneAsync` 仍然在后台执行， 尽管如此，启动它的那次操作也会被终止。这个程序将不会进行结构化并发，如下一小节所示。

使用 async 的结构化并发

让我们使用使用 async 的并发这一小节的例子并且提取出一个函数并发的调用 `doSomethingUsefulOne` 与 `doSomethingUsefulTwo` 并且返回它们两个的结果之和。 由于 async 被定义为了 CoroutineScope 上的扩展，我们需要将它写在作用域内，并且这是 coroutineScope 函数所提供的：

```
suspend fun concurrentSum(): Int = coroutineScope {
    val one = async { doSomethingUsefulOne() }
    val two = async { doSomethingUsefulTwo() }
    one.await() + two.await()
}
```

这种情况下，如果在 `concurrentSum` 函数内部发生了错误，并且它抛出了一个异常， 所有在作用域中启动的协程都会被取消。

```
import kotlinx.coroutines.*
import kotlin.system.*

fun main() = runBlocking<Unit> {
//sampleStart
    val time = measureTimeMillis {
        println("The answer is ${concurrentSum()}")
    }
    println("Completed in $time ms")
//sampleEnd
}

suspend fun concurrentSum(): Int = coroutineScope {
    val one = async { doSomethingUsefulOne() }
    val two = async { doSomethingUsefulTwo() }
    one.await() + two.await()
}

suspend fun doSomethingUsefulOne(): Int {
    delay(1000L) // 假设我们在这里做了些有用的事
    return 13
```

```
}

suspend fun doSomethingUsefulTwo(): Int {
    delay(1000L) // 假设我们在这里也做了些有用的事
    return 29
}
```

> 可以在这里获取完整代码。
>
> 

从上面的 `main` 函数的输出可以看出，我们仍然可以同时执行这两个操作：

```
The answer is 42
Completed in 1017 ms
```

取消始终通过协程的层次结构来进行传递：

```
import kotlinx.coroutines.*

fun main() = runBlocking<Unit> {
    try {
        failedConcurrentSum()
    } catch(e: ArithmeticException) {
        println("Computation failed with ArithmeticException")
    }
}

suspend fun failedConcurrentSum(): Int = coroutineScope {
    val one = async<Int> {
        try {
            delay(Long.MAX_VALUE) // 模拟一个长时间的运算
            42
        } finally {
            println("First child was cancelled")
        }
    }
    val two = async<Int> {
        println("Second child throws an exception")
        throw ArithmeticException()
    }
    one.await() + two.await()
}
```

> 可以在这里获取完整代码。
>
> 

请注意，如果其中一个子协程（即 `two` ）失败，第一个 `async` 以及等待中的父协程都会被取消：

```
Second child throws an exception
First child was cancelled
Computation failed with ArithmeticException
```

协程上下文与调度器

协程总是运行在一些以 CoroutineContext 类型为代表的上下文中，它们被定义在了 Kotlin 的标准库里。

协程上下文是各种不同元素的集合。其中主元素是协程中的 Job， 我们在前面的文档中见过它以及它的调度器，而本文将对它进行介绍。

调度器与线程

协程上下文包含一个 协程调度器 （ 参见 CoroutineDispatcher ）它确定了相关的协程在哪个线程或哪些线程上执行。协程调度器可以将协程限制在一个特定的线程执行，或将它分派到一个线程池，亦或是让它不受限地运行。

所有的协程构建器诸如 launch 和 async 接收一个可选的 CoroutineContext 参数，它可以被用来显式的为一个新协程或其它上下文元素指定一个调度器。

尝试下面的示例：

```
import kotlinx.coroutines.*

fun main() = runBlocking<Unit> {
//sampleStart
    launch { // 运行在父协程的上下文中，即 runBlocking 主协程
        println("main runBlocking      : I'm working in thread ${Thread.currentThread(
).name}")
    }
    launch(Dispatchers.Unconfined) { // 不受限的——将工作在主线程中
        println("Unconfined            : I'm working in thread ${Thread.currentThread(
).name}")
    }
    launch(Dispatchers.Default) { // 将会获取默认调度器
        println("Default               : I'm working in thread ${Thread.currentThread(
).name}")
    }
    launch(newSingleThreadContext("MyOwnThread")) { // 将使它获得一个新的线程
        println("newSingleThreadContext: I'm working in thread ${Thread.currentThread(
).name}")
    }
//sampleEnd
}
```

> 可以在这里获取完整代码。



它执行后得到了如下输出（也许顺序会有所不同）：

```
Unconfined            : I'm working in thread main
Default               : I'm working in thread DefaultDispatcher-worker-1
newSingleThreadContext: I'm working in thread MyOwnThread
main runBlocking      : I'm working in thread main
```

当调用 `launch { …… }` 时不传参数，它从启动了它的 CoroutineScope 中承袭了上下文（以及调度器）。在这个案例中，它从 `main` 线程中的 `runBlocking` 主协程承袭了上下文。

Dispatchers.Unconfined 是一个特殊的调度器且似乎也运行在 `main` 线程中，但实际上， 它是一种不同的机制，这会在后文中讲到。

The default dispatcher is used when no other dispatcher is explicitly specified in the scope. It is represented by Dispatchers.Default and uses a shared background pool of threads.

newSingleThreadContext 为协程的运行启动了一个线程。 一个专用的线程是一种非常昂贵的资源。 在真实的应用程序中两者都必须被释放，当不再需要的时候，使用 close 函数，或存储在一个顶层变量中使它在整个应用程序中被重用。

非受限调度器 vs 受限调度器

Dispatchers.Unconfined 协程调度器在调用它的线程启动了一个协程，但它仅仅只是运行到第一个挂起点。挂起后，它恢复线程中的协程，而这完全由被调用的挂起函数来决定。非受限的调度器非常适用于执行不消耗 CPU 时间的任务，以及不更新局限于特定线程的任何共享数据（如UI）的协程。

另一方面，该调度器默认继承了外部的 CoroutineScope。 runBlocking 协程的默认调度器，特别是， 当它被限制在了调用者线程时，继承自它将会有效地限制协程在该线程运行并且具有可预测的 FIFO 调度。

```
import kotlinx.coroutines.*

fun main() = runBlocking<Unit> {
//sampleStart
    launch(Dispatchers.Unconfined) { // 非受限的——将和主线程一起工作
        println("Unconfined      : I'm working in thread ${Thread.currentThread().name}
")
        delay(500)
        println("Unconfined      : After delay in thread ${Thread.currentThread().name}
")
    }
    launch { // 父协程的上下文，主 runBlocking 协程
        println("main runBlocking: I'm working in thread ${Thread.currentThread().name}
```

```
")
        delay(1000)
        println("main runBlocking: After delay in thread ${Thread.currentThread().name}
")
    }
//sampleEnd
}
```

> 可以在这里获取完整代码。

执行后的输出：

```
Unconfined      : I'm working in thread main
main runBlocking: I'm working in thread main
Unconfined      : After delay in thread kotlinx.coroutines.DefaultExecutor
main runBlocking: After delay in thread main
```

所以，该协程的上下文继承自 `runBlocking {...}` 协程并在 `main` 线程中运行，当 delay 函数调用的时候，非受限的那个协程在默认的执行者线程中恢复执行。

> 非受限的调度器是一种高级机制，可以在某些极端情况下提供帮助而不需要调度协程以便稍后执行或产生不希望的副作用， 因为某些操作必须立即在协程中执行。 非受限调度器不应该在通常的代码中使用。

调试协程与线程

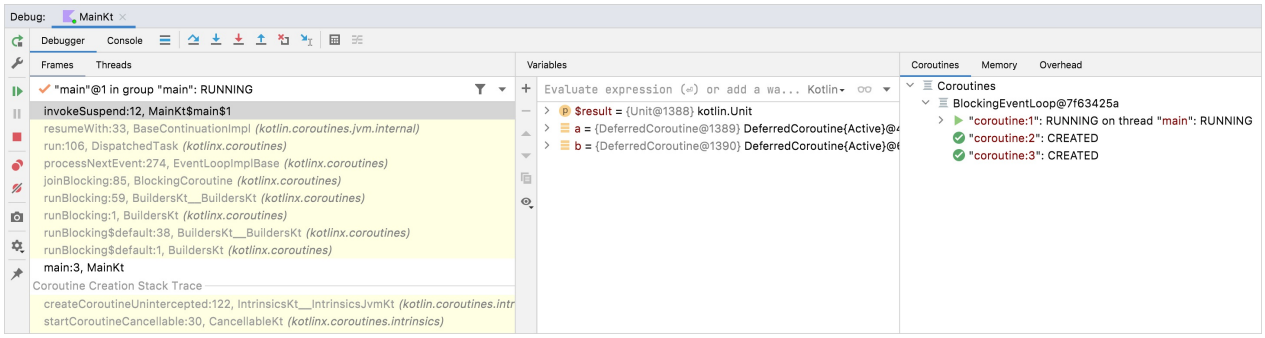
协程可以在一个线程上挂起并在其它线程上恢复。 如果没有特殊工具，甚至对于一个单线程的调度器也是难以弄清楚协程在何时何地正在做什么事情。

用 IDEA 调试

Kotlin 插件的协程调试器简化了 IntelliJ IDEA 中的协程调试.

> 调试适用于 1.3.8 或更高版本的 `kotlinx-coroutines-core` 。

调试工具窗口包含 **Coroutines** 标签。在这个标签中，你可以同时找到运行中与已挂起的协程的相关信息。 这些协程以它们所运行的调度器进行分组。



使用协程调试器，你可以：

- 检查每个协程的状态。
- 查看正在运行的与挂起的的协程的局部变量以及捕获变量的值。
- 查看完整的协程创建栈以及协程内部的调用栈。栈包含所有带有变量的栈帧，甚至包含那些在标准调试期间会丢失的栈帧。
- 获取包含每个协程的状态以及栈信息的完整报告。要获取它，请右键单击 **Coroutines** 选项卡，然后点击 **Get Coroutines Dump**。

要开始协程调试，你只需要设置断点并在调试模式下运行应用程序即可。

在这篇教程中学习更多的协程调试知识。

用日志调试

另一种调试线程应用程序而不使用协程调试器的方法是让线程在每一个日志文件的日志声明中打印线程的名字。这种特性在日志框架中是普遍受支持的。但是在使用协程时，单独的线程名称不会给出很多协程上下文信息，所以 `kotlinx.coroutines` 包含了调试工具来让它更简单。

使用 `-Dkotlinx.coroutines.debug` JVM 参数运行下面的代码：

```
import kotlinx.coroutines.*

fun log(msg: String) = println("[${Thread.currentThread().name}] $msg")

fun main() = runBlocking<Unit> {
//sampleStart
    val a = async {
        log("I'm computing a piece of the answer")
        6
    }
    val b = async {
        log("I'm computing another piece of the answer")
        7
    }
    log("The answer is ${a.await() * b.await()}")
//sampleEnd
}
```

> 可以在这里获取完整代码。
>
> 

这里有三个协程，包括 `runBlocking` 内的主协程 (#1)， 以及计算延期的值的另外两个协程 `a` (#2) 和 `b` (#3)。 它们都在 `runBlocking` 上下文中执行并且被限制在了主线程内。 这段代码的输出如下：

```
[main @coroutine#2] I'm computing a piece of the answer
[main @coroutine#3] I'm computing another piece of the answer
[main @coroutine#1] The answer is 42
```

这个 `log` 函数在方括号种打印了线程的名字，并且你可以看到它是 `main` 线程，并且附带了当前正在其上执行的协程的标识符。这个标识符在调试模式开启时，将连续分配给所有创建的协程。

> 当 JVM 以 `-ea` 参数配置运行时，调试模式也会开启。 你可以在 DEBUG_PROPERTY_NAME 属性的文档中阅读有关调试工具的更多信息。
>
> 

在不同线程间跳转

使用 `-Dkotlinx.coroutines.debug` JVM 参数运行下面的代码（参见调试）：

```
import kotlinx.coroutines.*

fun log(msg: String) = println("[${Thread.currentThread().name}] $msg")

fun main() {
    newSingleThreadContext("Ctx1").use { ctx1 ->
        newSingleThreadContext("Ctx2").use { ctx2 ->
            runBlocking(ctx1) {
                log("Started in ctx1")
                withContext(ctx2) {
                    log("Working in ctx2")
                }
                log("Back to ctx1")
            }
        }
    }
}
```

> 可以在这里获取完整代码。



它演示了一些新技术。其中一个使用 runBlocking 来显式指定了一个上下文，并且另一个使用 withContext 函数来改变协程的上下文，而仍然驻留在相同的协程中，正如可以在下面的输出中所见到的：

```
[Ctx1 @coroutine#1] Started in ctx1
[Ctx2 @coroutine#1] Working in ctx2
[Ctx1 @coroutine#1] Back to ctx1
```

注意，在这个例子中，当我们不再需要某个在 newSingleThreadContext 中创建的线程的时候，它使用了 Kotlin 标准库中的 `use` 函数来释放该线程。

上下文中的作业

协程的 Job 是上下文的一部分，并且可以使用 `coroutineContext [Job]` 表达式在上下文中检索它：

```
import kotlinx.coroutines.*

fun main() = runBlocking<Unit> {
//sampleStart
    println("My job is ${coroutineContext[Job]}")
//sampleEnd
}
```

> 可以在这里获取完整代码。



在调试模式下，它将输出如下这些信息：

```
My job is "coroutine#1":BlockingCoroutine{Active}@6d311334
```

请注意，CoroutineScope 中的 isActive 只是 `coroutineContext[Job]?.isActive == true` 的一种方便的快捷方式。

子协程

当一个协程被其它协程在 CoroutineScope 中启动的时候， 它将通过 CoroutineScope.coroutineContext 来承袭上下文，并且这个新协程的 Job 将会成为父协程作业的子 作业。当一个父协程被取消的时候，所有它的子协程也会被递归的取消。

However, this parent-child relation can be explicitly overriden in one of two ways:

1. When a different scope is explicitly specified when launching a coroutine (for example, `GlobalScope.launch`), then it does not inherit a `Job` from the parent scope.
2. When a different `Job` object is passed as the context for the new coroutine (as shown in the example below), then it overrides the `Job` of the parent scope.

In both cases, the launched coroutine is not tied to the scope it was launched from and operates independently.

```
import kotlinx.coroutines.*

fun main() = runBlocking<Unit> {
//sampleStart
    // 启动一个协程来处理某种传入请求（request）
    val request = launch {
        // 生成了两个子作业
        launch(Job()) {
            println("job1: I run in my own Job and execute independently!")
            delay(1000)
            println("job1: I am not affected by cancellation of the request")
        }
        // 另一个则承袭了父协程的上下文
        launch {
            delay(100)
            println("job2: I am a child of the request coroutine")
            delay(1000)
            println("job2: I will not execute this line if my parent request is cancel
led")
        }
    }
    delay(500)
    request.cancel() // 取消请求（request）的执行
    println("main: Who has survived request cancellation?")
    delay(1000) // 主线程延迟一秒钟来看看发生了什么
//sampleEnd
}
```

> 可以在这里获取完整代码。
>
> 

这段代码的输出如下：

```
job1: I run in my own Job and execute independently!
job2: I am a child of the request coroutine
main: Who has survived request cancellation?
job1: I am not affected by cancellation of the request
```

父协程的职责

一个父协程总是等待所有的子协程执行结束。父协程并不显式的跟踪所有子协程的启动，并且不必使用 Job.join 在最后的时候等待它们：

```
import kotlinx.coroutines.*

fun main() = runBlocking<Unit> {
//sampleStart
    // 启动一个协程来处理某种传入请求（request）
    val request = launch {
        repeat(3) { i -> // 启动少量的子作业
            launch  {
                delay((i + 1) * 200L) // 延迟 200 毫秒、400 毫秒、600 毫秒的时间
                println("Coroutine $i is done")
            }
        }
        println("request: I'm done and I don't explicitly join my children that are st
ill active")
    }
    request.join() // 等待请求的完成，包括其所有子协程
    println("Now processing of the request is complete")
//sampleEnd
}
```

> 可以在这里获取完整代码。 >

结果如下所示：

```
request: I'm done and I don't explicitly join my children that are still active
Coroutine 0 is done
Coroutine 1 is done
Coroutine 2 is done
Now processing of the request is complete
```

命名协程以用于调试

当协程经常打印日志并且你只需要关联来自同一个协程的日志记录时， 则自动分配的 id 是非常好的。然而，当一个协程与特定请求的处理相关联时或做一些特定的后台任务，最好将其明确命名以用于调试目的。 CoroutineName 上下文元素与线程名具有相同的目的。当调试模式开启时，它被包含在正在执行此协程的线程名中。

下面的例子演示了这一概念：

```
import kotlinx.coroutines.*
```

```
fun log(msg: String) = println("[${Thread.currentThread().name}] $msg")

fun main() = runBlocking(CoroutineName("main")) {
//sampleStart
    log("Started main coroutine")
    // 运行两个后台值计算
    val v1 = async(CoroutineName("v1coroutine")) {
        delay(500)
        log("Computing v1")
        6
    }
    val v2 = async(CoroutineName("v2coroutine")) {
        delay(1000)
        log("Computing v2")
        7
    }
    log("The answer for v1 * v2 = ${v1.await() * v2.await()}")
//sampleEnd
}
```

> 可以在这里获取完整代码。
>
> 

程序执行使用了 `-Dkotlinx.coroutines.debug` JVM 参数，输出如下所示：

```
[main @main#1] Started main coroutine
[main @v1coroutine#2] Computing v1
[main @v2coroutine#3] Computing v2
[main @main#1] The answer for v1 * v2 = 42
```

组合上下文中的元素

有时我们需要在协程上下文中定义多个元素。我们可以使用 `+` 操作符来实现。 比如说，我们可以显式指定一个调度器来启动协程并且同时显式指定一个命名：

```
import kotlinx.coroutines.*

fun main() = runBlocking<Unit> {
//sampleStart
    launch(Dispatchers.Default + CoroutineName("test")) {
        println("I'm working in thread ${Thread.currentThread().name}")
    }
//sampleEnd
}
```

> 可以在这里获取完整代码。
>
> 

这段代码使用了 `-Dkotlinx.coroutines.debug` JVM 参数，输出如下所示：

```
I'm working in thread DefaultDispatcher-worker-1 @test#2
```

协程作用域

让我们将关于上下文，子协程以及作业的知识综合在一起。假设我们的应用程序拥有一个具有生命周期的对象，但这个对象并不是一个协程。举例来说，我们编写了一个 Android 应用程序并在 Android 的 activity 上下文中启动了一组协程来使用异步操作拉取并更新数据以及执行动画等等。所有这些协程必须在这个 activity 销毁的时候取消以避免内存泄漏。当然，我们也可以手动操作上下文与作业，以结合 activity 的生命周期与它的协程，但是 `kotlinx.coroutines` 提供了一个封装：CoroutineScope 的抽象。 你应该已经熟悉了协程作用域，因为所有的协程构建器都声明为在它之上的扩展。

我们通过创建一个 CoroutineScope 实例来管理协程的生命周期，并使它与 activity 的生命周期相关联。 `CoroutineScope` 可以通过 CoroutineScope() 创建或者通过MainScope() 工厂函数。前者创建了一个通用作用域，而后者为使用 Dispatchers.Main 作为默认调度器的 UI 应用程序 创建作用域：

```
class Activity {
    private val mainScope = MainScope()

    fun destroy() {
        mainScope.cancel()
    }
    // 继续运行……
```

现在，我们可以使用定义的 `mainScope` 在这个 `Activity` 的作用域内启动协程。 对于该示例，我们启动了十个协程，它们会延迟不同的时间：

```
    // 在 Activity 类中
    fun doSomething() {
        // 在示例中启动了 10 个协程，且每个都工作了不同的时长
        repeat(10) { i ->
            mainScope.launch {
                delay((i + 1) * 200L) // 延迟 200 毫秒、400 毫秒、600 毫秒等等不同的时间
                println("Coroutine $i is done")
            }
        }
    }
```

```
} // Activity 类结束
```

在 main 函数中我们创建 activity，调用测试函数 `doSomething`，并且在 500 毫秒后销毁这个 activity。 这取消了从 `doSomething` 启动的所有协程。我们可以观察到这些是由于在销毁之后， 即使我们再等一会儿，activity 也不再打印消息。

```
import kotlinx.coroutines.*

class Activity {
    private val mainScope = CoroutineScope(Dispatchers.Default) // use Default for tes
t purposes

    fun destroy() {
        mainScope.cancel()
    }

    fun doSomething() {
        // 在示例中启动了 10 个协程，且每个都工作了不同的时长
        repeat(10) { i ->
            mainScope.launch {
                delay((i + 1) * 200L) // 延迟 200 毫秒、400 毫秒、600 毫秒等等不同的时间
                println("Coroutine $i is done")
            }
        }
    }
} // Activity 类结束

fun main() = runBlocking<Unit> {
//sampleStart
    val activity = Activity()
    activity.doSomething() // 运行测试函数
    println("Launched coroutines")
    delay(500L) // 延迟半秒钟
    println("Destroying activity!")
    activity.destroy() // 取消所有的协程
    delay(1000) // 为了在视觉上确认它们没有工作
//sampleEnd
}
```

> 可以在这里获取完整代码。

这个示例的输出如下所示：

```
Launched coroutines
Coroutine 0 is done
Coroutine 1 is done
Destroying activity!
```

你可以看到，只有前两个协程打印了消息，而另一个协程在 `Activity.destroy()` 中单次调用了 `job.cancel()` 。

> 注意，Android 在所有具有生命周期的实体中都对协程作用域提供了一等的支持。 请参见相关文档。



线程局部数据

有时，能够将一些线程局部数据传递到协程与协程之间是很方便的。 然而，由于它们不受任何特定线程的约束，如果手动完成，可能会导致出现样板代码。

`ThreadLocal` ， asContextElement 扩展函数在这里会充当救兵。它创建了额外的上下文元素，且保留给定 `ThreadLocal` 的值，并在每次协程切换其上下文时恢复它。

它很容易在下面的代码中演示：

```
import kotlinx.coroutines.*

val threadLocal = ThreadLocal<String?>() // 声明线程局部变量

fun main() = runBlocking<Unit> {
//sampleStart
    threadLocal.set("main")
    println("Pre-main, current thread: ${Thread.currentThread()}, thread local value: '
${threadLocal.get()}'")
    val job = launch(Dispatchers.Default + threadLocal.asContextElement(value = "launc
h")) {
        println("Launch start, current thread: ${Thread.currentThread()}, thread local
 value: '${threadLocal.get()}'")
        yield()
        println("After yield, current thread: ${Thread.currentThread()}, thread local
value: '${threadLocal.get()}'")
    }
    job.join()
    println("Post-main, current thread: ${Thread.currentThread()}, thread local value:
 '${threadLocal.get()}'")
//sampleEnd
}
```

> 可以在这里获取完整代码。



在这个例子中我们使用 Dispatchers.Default 在后台线程池中启动了一个新的协程，所以它工作在线程池中的不同线程中，但它仍然具有线程局部变量的值， 我们指定使用 `threadLocal.asContextElement(value = "launch")` ， 无论协程执行在哪个线程中都是没有问题的。 因此，其输出如（调试）所示：

```
Pre-main, current thread: Thread[main @coroutine#1,5,main], thread local value: 'main'
Launch start, current thread: Thread[DefaultDispatcher-worker-1 @coroutine#2,5,main],
thread local value: 'launch'
After yield, current thread: Thread[DefaultDispatcher-worker-2 @coroutine#2,5,main], t
hread local value: 'launch'
Post-main, current thread: Thread[main @coroutine#1,5,main], thread local value: 'main
'
```

这很容易忘记去设置相应的上下文元素。如果运行协程的线程不同， 在协程中访问的线程局部变量则可能会产生意外的值。 为了避免这种情况，建议使用 ensurePresent 方法并且在不正确的使用时快速失败。

`ThreadLocal` 具有一流的支持，可以与任何 `kotlinx.coroutines` 提供的原语一起使用。 但它有一个关键限制，即：当一个线程局部变量变化时，则这个新值不会传播给协程调用者（因为上下文元素无法追踪所有 `ThreadLocal` 对象访问），并且下次挂起时更新的值将丢失。 使用 withContext 在协程中更新线程局部变量，详见 asContextElement。

另外，一个值可以存储在一个可变的域中，例如 `class Counter(var i: Int)` ，是的，反过来， 可以存储在线程局部的变量中。然而，在这个案例中你完全有责任来进行同步可能的对这个可变的域进行的并发的修改。

对于高级的使用，例如，那些在内部使用线程局部传递数据的用于与日志记录 MDC 集成，以及事务上下文或任何其它库，请参见需要实现的 ThreadContextElement 接口的文档。

异步流

挂起函数可以异步的返回单个值，但是该如何异步返回多个计算好的值呢？这正是 Kotlin 流（Flow）的用武之地。

表示多个值

在 Kotlin 中可以使用集合来表示多个值。 比如说，我们有一个 `simple` 函数，它返回一个包含三个数字的 List， 然后使用 forEach 打印它们：

```
fun simple(): List<Int> = listOf(1, 2, 3)

fun main() {
    simple().forEach { value -> println(value) }
}
```

> 可以在这里获取完整代码。
>
> 

这段代码输出如下：

```
1
2
3
```

序列

如果使用一些消耗 CPU 资源的阻塞代码计算数字（每次计算需要 100 毫秒）那么我们可以使用 Sequence 来表示数字：

```
fun simple(): Sequence<Int> = sequence { // 序列构建器
    for (i in 1..3) {
        Thread.sleep(100) // 假装我们正在计算
        yield(i) // 产生下一个值
    }
}

fun main() {
    simple().forEach { value -> println(value) }
}
```

> 可以在这里获取完整代码。
>
> 

这段代码输出相同的数字，但在打印每个数字之前等待 100 毫秒。

挂起函数

然而，计算过程阻塞运行该代码的主线程。 当这些值由异步代码计算时，我们可以使用 `suspend` 修饰符标记函数 `simple`， 这样它就可以在不阻塞的情况下执行其工作并将结果作为列表返回：

```
import kotlinx.coroutines.*

//sampleStart
suspend fun simple(): List<Int> {
    delay(1000) // 假装我们在这里做了一些异步的事情
    return listOf(1, 2, 3)
}

fun main() = runBlocking<Unit> {
    simple().forEach { value -> println(value) }
}
//sampleEnd
```

> 可以在这里获取完整代码。
>
> 

这段代码将会在等待一秒之后打印数字。

流

使用 List 结果类型，意味着我们只能一次返回所有值。 为了表示异步计算的值流（stream），我们可以使用 Flow 类型（正如同步计算值会使用 Sequence 类型）：

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

//sampleStart
fun simple(): Flow<Int> = flow { // 流构建器
    for (i in 1..3) {
        delay(100) // 假装我们在这里做了一些有用的事情
        emit(i) // 发送下一个值
    }
}

fun main() = runBlocking<Unit> {
```

```
    // 启动并发的协程以验证主线程并未阻塞
    launch {
        for (k in 1..3) {
            println("I'm not blocked $k")
            delay(100)
        }
    }
    // 收集这个流
    simple().collect { value -> println(value) }
}
//sampleEnd
```

> 可以在这里获取完整代码。
>
> 

这段代码在不阻塞主线程的情况下每等待 100 毫秒打印一个数字。在主线程中运行一个单独的协程每 100 毫秒打印一次 “I'm not blocked” 已经经过了验证。

```
I'm not blocked 1
1
I'm not blocked 2
2
I'm not blocked 3
3
```

注意使用 Flow 的代码与先前示例的下述区别：

- Flow 类型的构建器函数名为 flow。
- `flow { ... }` 构建块中的代码可以挂起。
- 函数 `simple` 不再标有 `suspend` 修饰符。
- 流使用 emit 函数 发射 值。
- 流使用 collect 函数 收集 值。

> 我们可以在 `simple` 的 `flow { ... }` 函数体内使用 `Thread.sleep` 代替 delay 以观察主线程在本案例中被阻塞了。
>
> 

流是冷的

Flow 是一种类似于序列的冷流 — 这段 flow 构建器中的代码直到流被收集的时候才运行。这在以下的示例中非常明显：

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*
```

```
//sampleStart
fun simple(): Flow<Int> = flow {
    println("Flow started")
    for (i in 1..3) {
        delay(100)
        emit(i)
    }
}

fun main() = runBlocking<Unit> {
    println("Calling simple function...")
    val flow = simple()
    println("Calling collect...")
    flow.collect { value -> println(value) }
    println("Calling collect again...")
    flow.collect { value -> println(value) }
}
//sampleEnd
```

> 可以在这里获取完整代码。
>
> 

打印如下：

```
Calling simple function...
Calling collect...
Flow started
1
2
3
Calling collect again...
Flow started
1
2
3
```

这是返回一个流的 `simple` 函数没有标记 `suspend` 修饰符的主要原因。 The `simple()` call itself returns quickly and does not wait for anything. The flow starts afresh every time it is collected and that is why we see "Flow started" every time we call `collect` again.

流取消基础

流采用与协程同样的协作取消。像往常一样，流的收集可以在当流在一个可取消的挂起函数（例如 delay）中挂起的时候取消。 以下示例展示了当 withTimeoutOrNull 块中代码在运行的时候流是如何在超时的情况下取消并停止执行其代码的：

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

//sampleStart
fun simple(): Flow<Int> = flow {
    for (i in 1..3) {
        delay(100)
        println("Emitting $i")
        emit(i)
    }
}

fun main() = runBlocking<Unit> {
    withTimeoutOrNull(250) { // 在 250 毫秒后超时
        simple().collect { value -> println(value) }
    }
    println("Done")
}
//sampleEnd
```

> 可以在这里获取完整代码。
>
> 

注意，在 `simple` 函数中流仅发射两个数字，产生以下输出：

```
Emitting 1
1
Emitting 2
2
Done
```

See Flow cancellation checks section for more details.

流构建器

先前示例中的 `flow { ... }` 构建器是最基础的一个。还有其他构建器让流可以这样声明：

- flowOf 构建器定义了一个发射固定值集的流。
- 使用 `.asFlow()` 扩展函数，可以将各种集合与序列转换为流。

For example, the snippet that prints the numbers 1 to 3 from a flow can be rewritten as follows:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

fun main() = runBlocking<Unit> {
```

```
//sampleStart
    // 将一个整数区间转化为流
    (1..3).asFlow().collect { value -> println(value) }
//sampleEnd
}
```

> 可以在这里获取完整代码。
>
> 

过渡流操作符

可以使用操作符转换流，就像转换集合与序列一样。 过渡操作符应用于上游流，并返回下游流。 这些操作符也是冷操作符，就像流一样。这类操作符本身不是挂起函数。它运行的速度很快，返回新的转换流的定义。

基础的操作符拥有相似的名字，比如 map 与 filter。 这些操作符与序列版的主要区别在于这些操作符中的代码可以调用挂起函数。

举例来说，一个请求中的流可以使用 map 操作符映射出结果，即使执行一个长时间的请求操作也可以使用挂起函数来实现：

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

//sampleStart
suspend fun performRequest(request: Int): String {
    delay(1000) // 模仿长时间运行的异步工作
    return "response $request"
}

fun main() = runBlocking<Unit> {
    (1..3).asFlow() // 一个请求流
        .map { request -> performRequest(request) }
        .collect { response -> println(response) }
}
//sampleEnd
```

> 可以在这里获取完整代码。
>
> 

它产生以下三行，每行都比前一行晚一秒出现：

```
response 1
response 2
response 3
```

转换操作符

在流转换操作符中，最通用的一种称为 transform。它可以用来模仿简单的转换，例如 map 与 filter，以及实施更复杂的转换。 使用 `transform` 操作符，我们可以 发射 任意值任意次。

比如说，使用 `transform` 我们可以在执行长时间运行的异步请求之前发射一个字符串并跟踪这个响应：

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

suspend fun performRequest(request: Int): String {
    delay(1000) // 模仿长时间运行的异步任务
    return "response $request"
}

fun main() = runBlocking<Unit> {
//sampleStart
    (1..3).asFlow() // 一个请求流
        .transform { request ->
            emit("Making request $request")
            emit(performRequest(request))
        }
        .collect { response -> println(response) }
//sampleEnd
}
```

> 可以在这里获取完整代码。

这段代码的输出如下：

```
Making request 1
response 1
Making request 2
response 2
Making request 3
response 3
```

限长操作符

限长过渡操作符（例如 take）在流触及相应限制的时候会将它的执行取消。协程中的取消操作总是通过抛出异常来执行，这样所有的资源管理函数（如 `try {...} finally {...}` 块）会在取消的情况下正常运行：

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

//sampleStart
fun numbers(): Flow<Int> = flow {
    try {
        emit(1)
        emit(2)
        println("This line will not execute")
        emit(3)
    } finally {
        println("Finally in numbers")
    }
}

fun main() = runBlocking<Unit> {
    numbers()
        .take(2) // 只获取前两个
        .collect { value -> println(value) }
}
//sampleEnd
```

> 可以在这里获取完整代码。
>
> 

这段代码的输出清楚地表明，`numbers()` 函数中对 `flow {...}` 函数体的执行在发射出第二个数字后停止：

```
1
2
Finally in numbers
```

末端流操作符

末端操作符是在流上用于启动流收集的挂起函数。 collect 是最基础的末端操作符，但是还有另外一些更方便使用的末端操作符：

- 转化为各种集合，例如 toList 与 toSet。
- 获取第一个（first）值与确保流发射单个（single）值的操作符。
- 使用 reduce 与 fold 将流规约到单个值。

举例来说：

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*
```

```
fun main() = runBlocking<Unit> {
//sampleStart
    val sum = (1..5).asFlow()
        .map { it * it } // 数字 1 至 5 的平方
        .reduce { a, b -> a + b } // 求和（末端操作符）
    println(sum)
//sampleEnd
}
```

> 可以在这里获取完整代码。
>
> 

打印单个数字：

```
55
```

流是连续的

流的每次单独收集都是按顺序执行的，除非进行特殊操作的操作符使用多个流。该收集过程直接在协程中运行，该协程调用末端操作符。 默认情况下不启动新协程。 从上游到下游每个过渡操作符都会处理每个发射出的值然后再交给末端操作符。

请参见以下示例，该示例过滤偶数并将其映射到字符串：

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

fun main() = runBlocking<Unit> {
//sampleStart
    (1..5).asFlow()
        .filter {
            println("Filter $it")
            it % 2 == 0
        }
        .map {
            println("Map $it")
            "string $it"
        }.collect {
            println("Collect $it")
        }
//sampleEnd
}
```

> 可以在这里获取完整代码。
>
> 

执行：

```
Filter 1
Filter 2
Map 2
Collect string 2
Filter 3
Filter 4
Map 4
Collect string 4
Filter 5
```

流上下文

流的收集总是在调用协程的上下文中发生。例如，如果有一个流 `simple`，然后以下代码在它的编写者指定的上下文中运行，而无论流 `simple` 的实现细节如何：

```
withContext(context) {
    simple().collect { value ->
        println(value) // 运行在指定上下文中
    }
}
```

流的该属性称为 上下文保存 。

所以默认的， `flow { ... }` 构建器中的代码运行在相应流的收集器提供的上下文中。举例来说，考虑打印线程的一个 `simple` 函数的实现， 它被调用并发射三个数字：

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

fun log(msg: String) = println("[${Thread.currentThread().name}] $msg")

//sampleStart
fun simple(): Flow<Int> = flow {
    log("Started simple flow")
    for (i in 1..3) {
        emit(i)
    }
}

fun main() = runBlocking<Unit> {
    simple().collect { value -> log("Collected $value") }
}
//sampleEnd
```

> 可以在这里获取完整代码。
>
> 

运行这段代码：

```
[main @coroutine#1] Started simple flow
[main @coroutine#1] Collected 1
[main @coroutine#1] Collected 2
[main @coroutine#1] Collected 3
```

由于 `simple().collect` 是在主线程调用的，那么 `simple` 的流主体也是在主线程调用的。 这是快速运行或异步代码的理想默认形式，它不关心执行的上下文并且不会阻塞调用者。

A common pitfall when using withContext

然而，长时间运行的消耗 CPU 的代码也许需要在 Dispatchers.Default 上下文中执行，并且更新 UI 的代码也许需要在 Dispatchers.Main 中执行。通常，withContext 用于在 Kotlin 协程中改变代码的上下文，但是 `flow {...}` 构建器中的代码必须遵循上下文保存属性，并且不允许从其他上下文中发射（emit）。

尝试运行下面的代码：

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

//sampleStart
fun simple(): Flow<Int> = flow {
    // 在流构建器中更改消耗 CPU 代码的上下文的错误方式
    kotlinx.coroutines.withContext(Dispatchers.Default) {
        for (i in 1..3) {
            Thread.sleep(100) // 假装我们以消耗 CPU 的方式进行计算
            emit(i) // 发射下一个值
        }
    }
}

fun main() = runBlocking<Unit> {
    simple().collect { value -> println(value) }
}
//sampleEnd
```

> 可以在这里获取完整代码。
>
> 

这段代码产生如下的异常：

```
Exception in thread "main" java.lang.IllegalStateException: Flow invariant is violated
:
        Flow was collected in [CoroutineId(1), "coroutine#1":BlockingCoroutine{Active}
@5511c7f8, BlockingEventLoop@2eac3323],
        but emission happened in [CoroutineId(1), "coroutine#1":DispatchedCoroutine{Ac
tive}@2dae0000, Dispatchers.Default].
        Please refer to 'flow' documentation or use 'flowOn' instead
    at ...
```

flowOn 操作符

例外的是 flowOn 函数，该函数用于更改流发射的上下文。 以下示例展示了更改流上下文的正确方法，该示例还通过打印相应线程的名字以展示它们的工作方式：

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

fun log(msg: String) = println("[${Thread.currentThread().name}] $msg")

//sampleStart
fun simple(): Flow<Int> = flow {
    for (i in 1..3) {
        Thread.sleep(100) // 假装我们以消耗 CPU 的方式进行计算
        log("Emitting $i")
        emit(i) // 发射下一个值
    }
}.flowOn(Dispatchers.Default) // 在流构建器中改变消耗 CPU 代码上下文的正确方式

fun main() = runBlocking<Unit> {
    simple().collect { value ->
        log("Collected $value")
    }
}
//sampleEnd
```

> 可以在这里获取完整代码。
>
> 

注意，当收集发生在主线程中，`flow { ... }` 是如何在后台线程中工作的：

这里要观察的另一件事是 flowOn 操作符已改变流的默认顺序性。 现在收集发生在一个协程中（“coroutine#1”）而发射发生在运行于另一个线程中与收集协程并发运行的另一个协程（“coroutine#2”）中。当上游流必须改变其上下文中的 CoroutineDispatcher 的时候，flowOn 操作符创建了另一个协程。

缓冲

从收集流所花费的时间来看，将流的不同部分运行在不同的协程中将会很有帮助，特别是当涉及到长时间运行的异步操作时。例如，考虑一种情况， 一个 `simple` 流的发射很慢，它每花费 100 毫秒才产生一个元素；而收集器也非常慢， 需要花费 300 毫秒来处理元素。让我们看看从该流收集三个数字要花费多长时间：

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*
import kotlin.system.*

//sampleStart
fun simple(): Flow<Int> = flow {
    for (i in 1..3) {
        delay(100) // 假装我们异步等待了 100 毫秒
        emit(i) // 发射下一个值
    }
}

fun main() = runBlocking<Unit> {
    val time = measureTimeMillis {
        simple().collect { value ->
            delay(300) // 假装我们花费 300 毫秒来处理它
            println(value)
        }
    }
    println("Collected in $time ms")
}
//sampleEnd
```

> 可以在这里获取完整代码。
>
> 

它会产生这样的结果，整个收集过程大约需要 1200 毫秒（3 个数字，每个花费 400 毫秒）：

```
1
2
3
Collected in 1220 ms
```

我们可以在流上使用 buffer 操作符来并发运行这个 `simple` 流中发射元素的代码以及收集的代码， 而不是顺序运行它们：

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*
import kotlin.system.*
```

```
fun simple(): Flow<Int> = flow {
    for (i in 1..3) {
        delay(100) // 假装我们异步等待了 100 毫秒
        emit(i) // 发射下一个值
    }
}

fun main() = runBlocking<Unit> {
//sampleStart
    val time = measureTimeMillis {
        simple()
            .buffer() // 缓冲发射项，无需等待
            .collect { value ->
                delay(300) // 假装我们花费 300 毫秒来处理它
                println(value)
            }
    }
    println("Collected in $time ms")
//sampleEnd
}
```

> 可以在这里获取完整代码。
>
> 

它产生了相同的数字，只是更快了，由于我们高效地创建了处理流水线， 仅仅需要等待第一个数字产生的 100 毫秒以及处理每个数字各需花费的 300 毫秒。这种方式大约花费了 1000 毫秒来运行：

```
1
2
3
Collected in 1071 ms
```

> 注意，当必须更改 CoroutineDispatcher 时， flowOn 操作符使用了相同的缓冲机制， 但是我们在这里显式地请求缓冲而不改变执行上下文。
>
> 

合并

当流代表部分操作结果或操作状态更新时，可能没有必要处理每个值，而是只处理最新的那个。在本示例中，当收集器处理它们太慢的时候， conflate 操作符可以用于跳过中间值。构建前面的示例：

```
import kotlinx.coroutines.*
```

```
import kotlinx.coroutines.flow.*
import kotlin.system.*

fun simple(): Flow<Int> = flow {
    for (i in 1..3) {
        delay(100) // 假装我们异步等待了 100 毫秒
        emit(i) // 发射下一个值
    }
}

fun main() = runBlocking<Unit> {
//sampleStart
    val time = measureTimeMillis {
        simple()
            .conflate() // 合并发射项，不对每个值进行处理
            .collect { value ->
                delay(300) // 假装我们花费 300 毫秒来处理它
                println(value)
            }
    }
    println("Collected in $time ms")
//sampleEnd
}
```

> 可以在这里获取完整代码。

我们看到，虽然第一个数字仍在处理中，但第二个和第三个数字已经产生，因此第二个是 *conflated* ，只有最新的（第三个）被交付给收集器：

```
1
3
Collected in 758 ms
```

处理最新值

当发射器和收集器都很慢的时候，合并是加快处理速度的一种方式。它通过删除发射值来实现。 另一种方式是取消缓慢的收集器，并在每次发射新值的时候重新启动它。有一组与 `xxx` 操作符执行相同基本逻辑的 `xxxLatest` 操作符，但是在新值产生的时候取消执行其块中的代码。让我们在先前的示例中尝试更换 conflate 为 collectLatest：

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*
import kotlin.system.*

fun simple(): Flow<Int> = flow {
    for (i in 1..3) {
```

```
        delay(100) // 假装我们异步等待了 100 毫秒
        emit(i) // 发射下一个值
    }
}

fun main() = runBlocking<Unit> {
//sampleStart
    val time = measureTimeMillis {
        simple()
            .collectLatest { value -> // 取消并重新发射最后一个值
                println("Collecting $value")
                delay(300) // 假装我们花费 300 毫秒来处理它
                println("Done $value")
            }
    }
    println("Collected in $time ms")
//sampleEnd
}
```

> 可以在这里获取完整代码。
>
> 

由于 collectLatest 的函数体需要花费 300 毫秒，但是新值每 100 秒发射一次，我们看到该代码块对每个值运行，但是只收集最后一个值：

```
Collecting 1
Collecting 2
Collecting 3
Done 3
Collected in 741 ms
```

组合多个流

组合多个流有很多种方式。

Zip

就像 Kotlin 标准库中的 Sequence.zip 扩展函数一样， 流拥有一个 zip 操作符用于组合两个流中的相关值：

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

fun main() = runBlocking<Unit> {
//sampleStart
```

```
    val nums = (1..3).asFlow() // 数字 1..3
    val strs = flowOf("one", "two", "three") // 字符串
    nums.zip(strs) { a, b -> "$a -> $b" } // 组合单个字符串
        .collect { println(it) } // 收集并打印
//sampleEnd
}
```

> 可以在这里获取完整代码。
>
> 

示例打印如下：

```
1 -> one
2 -> two
3 -> three
```

Combine

当流表示一个变量或操作的最新值时（请参阅相关小节 conflation），可能需要执行计算，这依赖于相应流的最新值，并且每当上游流产生值的时候都需要重新计算。这种相应的操作符家族称为 combine。

例如，先前示例中的数字如果每 300 毫秒更新一次，但字符串每 400 毫秒更新一次， 然后使用 zip 操作符合并它们，但仍会产生相同的结果， 尽管每 400 毫秒打印一次结果：

> 我们在本示例中使用 onEach 过渡操作符来延时每次元素发射并使该流更具说明性以及更简洁。
>
> 

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

fun main() = runBlocking<Unit> {
//sampleStart

    val nums = (1..3).asFlow().onEach { delay(300) } // 发射数字 1..3，间隔 300 毫秒
    val strs = flowOf("one", "two", "three").onEach { delay(400) } // 每 400 毫秒发射一
次字符串
    val startTime = System.currentTimeMillis() // 记录开始的时间
    nums.zip(strs) { a, b -> "$a -> $b" } // 使用“zip”组合单个字符串
        .collect { value -> // 收集并打印
            println("$value at ${System.currentTimeMillis() - startTime} ms from start"
)
        }
//sampleEnd
}
```

> 可以在这里获取完整代码。
>
> 

然而，当在这里使用 combine 操作符来替换 zip：

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

fun main() = runBlocking<Unit> {
//sampleStart

    val nums = (1..3).asFlow().onEach { delay(300) } // 发射数字 1..3，间隔 300 毫秒
    val strs = flowOf("one", "two", "three").onEach { delay(400) } // 每 400 毫秒发射一
次字符串
    val startTime = System.currentTimeMillis() // 记录开始的时间
    nums.combine(strs) { a, b -> "$a -> $b" } // 使用“combine”组合单个字符串
        .collect { value -> // 收集并打印
            println("$value at ${System.currentTimeMillis() - startTime} ms from start"
)
        }
//sampleEnd
}
```

> 可以在这里获取完整代码。
>
> 

我们得到了完全不同的输出，其中，`nums` 或 `strs` 流中的每次发射都会打印一行：

```
1 -> one at 452 ms from start
2 -> one at 651 ms from start
2 -> two at 854 ms from start
3 -> two at 952 ms from start
3 -> three at 1256 ms from start
```

展平流

流表示异步接收的值序列，因此很容易遇到这样的情况： 每个值都会触发对另一个值序列的请求。比如说，我们可以拥有下面这样一个返回间隔 500 毫秒的两个字符串流的函数：

```
fun requestFlow(i: Int): Flow<String> = flow {
    emit("$i: First")
    delay(500) // 等待 500 毫秒
    emit("$i: Second")
```

```
}
```

现在，如果我们有一个包含三个整数的流，并对每个整数调用 `requestFlow`，如下所示：

```
(1..3).asFlow().map { requestFlow(it) }
```

然后我们会得到一个包含流的流（`Flow<Flow<String>>`），需要将其进行展平为单个流以进行下一步处理。集合与序列都拥有 flatten 与 flatMap 操作符来做这件事。然而，由于流具有异步的性质，因此需要不同的展平模式， 为此，存在一系列的流展平操作符。

flatMapConcat

将流的流串联由 flatMapConcat 与 flattenConcat 操作符提供。它们是相应序列操作符最相近的类似物。它们在等待内部流完成之后开始收集下一个值，如下面的示例所示：

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

fun requestFlow(i: Int): Flow<String> = flow {
    emit("$i: First")
    delay(500) // 等待 500 毫秒
    emit("$i: Second")
}

fun main() = runBlocking<Unit> {
//sampleStart
    val startTime = System.currentTimeMillis() // 记录开始时间
    (1..3).asFlow().onEach { delay(100) } // 每 100 毫秒发射一个数字
        .flatMapConcat { requestFlow(it) }

        .collect { value -> // 收集并打印
            println("$value at ${System.currentTimeMillis() - startTime} ms from start"
)
        }
//sampleEnd
}
```

> 可以在这里获取完整代码。
>
> 

在输出中可以清楚地看到 flatMapConcat 的顺序性质：

```
1: First at 121 ms from start
1: Second at 622 ms from start
2: First at 727 ms from start
```

```
2: Second at 1227 ms from start
3: First at 1328 ms from start
3: Second at 1829 ms from start
```

flatMapMerge

另一种展平操作是并发收集所有传入的流，并将它们的值合并到一个单独的流，以便尽快的发射值。 它由 flatMapMerge 与 flattenMerge 操作符实现。他们都接收可选的用于限制并发收集的流的个数的 `concurrency` 参数（默认情况下，它等于 DEFAULT_CONCURRENCY）。

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

fun requestFlow(i: Int): Flow<String> = flow {
    emit("$i: First")
    delay(500) // 等待 500 毫秒
    emit("$i: Second")
}

fun main() = runBlocking<Unit> {
//sampleStart
    val startTime = System.currentTimeMillis() // 记录开始时间
    (1..3).asFlow().onEach { delay(100) } // 每 100 毫秒发射一个数字
        .flatMapMerge { requestFlow(it) }

        .collect { value -> // 收集并打印
            println("$value at ${System.currentTimeMillis() - startTime} ms from start"
)
        }
//sampleEnd
}
```

> 可以在这里获取完整代码。
>
> 

flatMapMerge 的并发性质很明显：

```
1: First at 136 ms from start
2: First at 231 ms from start
3: First at 333 ms from start
1: Second at 639 ms from start
2: Second at 732 ms from start
3: Second at 833 ms from start
```

> 注意，flatMapMerge 会顺序调用代码块（本示例中的 `{ requestFlow(it) }`），但是并发收集结果流，相当于执行顺序是首先执行 `map { requestFlow(it) }` 然后在其返回结果上调用 flattenMerge。



flatMapLatest

与 collectLatest 操作符类似（在"处理最新值" 小节中已经描述过），也有相对应的“最新”展平模式，在发出新流后立即取消先前流的收集。 这由 flatMapLatest 操作符来实现。

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

fun requestFlow(i: Int): Flow<String> = flow {
    emit("$i: First")
    delay(500) // 等待 500 毫秒
    emit("$i: Second")
}

fun main() = runBlocking<Unit> {
//sampleStart
    val startTime = System.currentTimeMillis() // 记录开始时间
    (1..3).asFlow().onEach { delay(100) } // 每 100 毫秒发射一个数字
        .flatMapLatest { requestFlow(it) }

        .collect { value -> // 收集并打印
            println("$value at ${System.currentTimeMillis() - startTime} ms from start"
)
        }
//sampleEnd
}
```

> 可以在这里获取完整代码。



该示例的输出很好的展示了 flatMapLatest 的工作方式：

```
1: First at 142 ms from start
2: First at 322 ms from start
3: First at 425 ms from start
3: Second at 931 ms from start
```

> 注意，flatMapLatest 在收到一个新值时取消了块中的所有代码 (本示例中的 `{ requestFlow(it) }`)。 这在该特定示例中不会有什么区别，由于调用 `requestFlow` 自身的速度是很快的，不会发生挂起， 所以不会被取消。然而，a differnce in output would be visible if we were to use suspending functions like `delay` in `requestFlow`.
>
> 

流异常

当运算符中的发射器或代码抛出异常时，流收集可以带有异常的完成。 有几种处理异常的方法。

收集器 try 与 catch

收集者可以使用 Kotlin 的 `try/catch` 块来处理异常：

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

//sampleStart
fun simple(): Flow<Int> = flow {
    for (i in 1..3) {
        println("Emitting $i")
        emit(i) // 发射下一个值
    }
}

fun main() = runBlocking<Unit> {
    try {
        simple().collect { value ->
            println(value)
            check(value <= 1) { "Collected $value" }
        }
    } catch (e: Throwable) {
        println("Caught $e")
    }
}
//sampleEnd
```

> 可以在这里获取完整代码。
>
> 

这段代码成功的在末端操作符 collect 中捕获了异常，并且， 如我们所见，在这之后不再发出任何值：

```
Emitting 1
1
Emitting 2
2
Caught java.lang.IllegalStateException: Collected 2
```

一切都已捕获

前面的示例实际上捕获了在发射器或任何过渡或末端操作符中发生的任何异常。 例如，让我们修改代码以便将发出的值映射为字符串， 但是相应的代码会产生一个异常：

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

//sampleStart
fun simple(): Flow<String> =
    flow {
        for (i in 1..3) {
            println("Emitting $i")
            emit(i) // 发射下一个值
        }
    }
    .map { value ->
        check(value <= 1) { "Crashed on $value" }
        "string $value"
    }

fun main() = runBlocking<Unit> {
    try {
        simple().collect { value -> println(value) }
    } catch (e: Throwable) {
        println("Caught $e")
    }
}
//sampleEnd
```

> 可以在这里获取完整代码。
>
> 

仍然会捕获该异常并停止收集：

```
Emitting 1
string 1
Emitting 2
Caught java.lang.IllegalStateException: Crashed on 2
```

异常透明性

但是，发射器的代码如何封装其异常处理行为？

流必须对异常透明，即在 `flow { ... }` 构建器内部的 `try/catch` 块中发射值是违反异常透明性的。这样可以保证收集器抛出的一个异常能被像先前示例中那样的 `try/catch` 块捕获。

发射器可以使用 catch 操作符来保留此异常的透明性并允许封装它的异常处理。catch 操作符的代码块可以分析异常并根据捕获到的异常以不同的方式对其做出反应：

- 可以使用 `throw` 重新抛出异常。
- 可以使用 catch 代码块中的 emit 将异常转换为值发射出去。
- 可以将异常忽略，或用日志打印，或使用一些其他代码处理它。

例如，让我们在捕获异常的时候发射文本：

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

fun simple(): Flow<String> =
    flow {
        for (i in 1..3) {
            println("Emitting $i")
            emit(i) // 发射下一个值
        }
    }
    .map { value ->
        check(value <= 1) { "Crashed on $value" }
        "string $value"
    }

fun main() = runBlocking<Unit> {
//sampleStart
    simple()
        .catch { e -> emit("Caught $e") } // 发射一个异常
        .collect { value -> println(value) }
//sampleEnd
}
```

> 可以在这里获取完整代码。
>
> 

即使我们不再在代码的外层使用 `try/catch`，示例的输出也是相同的。

透明捕获

catch 过渡操作符遵循异常透明性，仅捕获上游异常（ `catch` 操作符上游的异常，但是它下面的不是）。 如果 `collect { ... }` 块（位于 `catch` 之下）抛出一个异常，那么异常会逃逸：

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

//sampleStart
fun simple(): Flow<Int> = flow {
    for (i in 1..3) {
        println("Emitting $i")
        emit(i)
    }
}

fun main() = runBlocking<Unit> {
    simple()
        .catch { e -> println("Caught $e") } // 不会捕获下游异常
        .collect { value ->
            check(value <= 1) { "Collected $value" }
            println(value)
        }
}
//sampleEnd
```

> 可以在这里获取完整代码。

尽管有 `catch` 操作符，但不会打印“Caught ...”消息：

```
Emitting 1
1
Emitting 2
Exception in thread "main" java.lang.IllegalStateException: Collected 2
    at ...
```

声明式捕获

我们可以将 catch 操作符的声明性与处理所有异常的期望相结合，将 collect 操作符的代码块移动到 onEach 中，并将其放到 `catch` 操作符之前。收集该流必须由调用无参的 `collect()` 来触发：

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

fun simple(): Flow<Int> = flow {
```

```
    for (i in 1..3) {
        println("Emitting $i")
        emit(i)
    }
}

fun main() = runBlocking<Unit> {
//sampleStart
    simple()
        .onEach { value ->
            check(value <= 1) { "Collected $value" }
            println(value)
        }
        .catch { e -> println("Caught $e") }
        .collect()
//sampleEnd
}
```

> 可以在这里获取完整代码。
>
> 

现在我们可以看到已经打印了“Caught ...”消息，并且我们可以在没有显式使用 `try/catch` 块的情况下捕获所有异常：

```
Emitting 1
1
Emitting 2
Caught java.lang.IllegalStateException: Collected 2
```

流完成

当流收集完成时（普通情况或异常情况），它可能需要执行一个动作。 你可能已经注意到，它可以通过两种方式完成：命令式或声明式。

命令式 finally 块

除了 `try` / `catch` 之外，收集器还能使用 `finally` 块在 `collect` 完成时执行一个动作。

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

//sampleStart
fun simple(): Flow<Int> = (1..3).asFlow()

fun main() = runBlocking<Unit> {
    try {
```

```
        simple().collect { value -> println(value) }
    } finally {
        println("Done")
    }
}
//sampleEnd
```

> 可以在这里获取完整代码。
>
> 

这段代码打印出 `simple` 流产生的三个数字，后面跟一个“Done”字符串：

```
1
2
3
Done
```

声明式处理

对于声明式，流拥有 onCompletion 过渡操作符，它在流完全收集时调用。

可以使用 onCompletion 操作符重写前面的示例，并产生相同的输出：

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

fun simple(): Flow<Int> = (1..3).asFlow()

fun main() = runBlocking<Unit> {
//sampleStart
    simple()
        .onCompletion { println("Done") }
        .collect { value -> println(value) }
//sampleEnd
}
```

> 可以在这里获取完整代码。
>
> 

onCompletion 的主要优点是其 lambda 表达式的可空参数 `Throwable` 可以用于确定流收集是正常完成还是有异常发生。在下面的示例中 `simple` 流在发射数字 1 之后抛出了一个异常：

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

//sampleStart
```

```
fun simple(): Flow<Int> = flow {
    emit(1)
    throw RuntimeException()
}

fun main() = runBlocking<Unit> {
    simple()
        .onCompletion { cause -> if (cause != null) println("Flow completed exceptiona
lly") }
        .catch { cause -> println("Caught exception") }
        .collect { value -> println(value) }
}
//sampleEnd
```

> 可以在这里获取完整代码。
>
> 

如你所期望的，它打印了：

```
1
Flow completed exceptionally
Caught exception
```

onCompletion 操作符与 catch 不同，它不处理异常。我们可以看到前面的示例代码，异常仍然流向下游。它将被提供给后面的 `onCompletion` 操作符，并可以由 `catch` 操作符处理。

成功完成

与 catch 操作符的另一个不同点是 onCompletion 能观察到所有异常并且仅在上游流成功完成（没有取消或失败）的情况下接收一个 `null` 异常。

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

//sampleStart
fun simple(): Flow<Int> = (1..3).asFlow()

fun main() = runBlocking<Unit> {
    simple()
        .onCompletion { cause -> println("Flow completed with $cause") }
        .collect { value ->
            check(value <= 1) { "Collected $value" }
            println(value)
        }
}
//sampleEnd
```

> 可以在这里获取完整代码。
>
> 

我们可以看到完成时 cause 不为空，因为流由于下游异常而中止：

```
1
Flow completed with java.lang.IllegalStateException: Collected 2
Exception in thread "main" java.lang.IllegalStateException: Collected 2
```

命令式还是声明式

现在我们知道如何收集流，并以命令式与声明式的方式处理其完成及异常情况。 这里有一个很自然的问题是，哪种方式应该是首选的？为什么？ 作为一个库，我们不主张采用任何特定的方式，并且相信这两种选择都是有效的， 应该根据自己的喜好与代码风格进行选择。

启动流

使用流表示来自一些源的异步事件是很简单的。 在这个案例中，我们需要一个类似 `addEventListener` 的函数，该函数注册一段响应的代码处理即将到来的事件，并继续进行进一步的处理。onEach 操作符可以担任该角色。 然而， `onEach` 是一个过渡操作符。我们也需要一个末端操作符来收集流。 否则仅调用 `onEach` 是无效的。

如果我们在 `onEach` 之后使用 collect 末端操作符，那么后面的代码会一直等待直至流被收集：

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

//sampleStart
// 模仿事件流
fun events(): Flow<Int> = (1..3).asFlow().onEach { delay(100) }

fun main() = runBlocking<Unit> {
    events()
        .onEach { event -> println("Event: $event") }
        .collect() // <--- 等待流收集
    println("Done")
}
//sampleEnd
```

> 可以在这里获取完整代码。
>
> 

你可以看到它的输出：

```
Event: 1
Event: 2
Event: 3
Done
```

launchIn 末端操作符可以在这里派上用场。使用 `launchIn` 替换 `collect` 我们可以在单独的协程中启动流的收集，这样就可以立即继续进一步执行代码：

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

// 模仿事件流
fun events(): Flow<Int> = (1..3).asFlow().onEach { delay(100) }

//sampleStart
fun main() = runBlocking<Unit> {
    events()
        .onEach { event -> println("Event: $event") }
        .launchIn(this) // <--- 在单独的协程中执行流
    println("Done")
}
//sampleEnd
```

> 可以在这里获取完整代码。
>
> 

它打印了：

```
Done
Event: 1
Event: 2
Event: 3
```

`launchIn` 必要的参数 CoroutineScope 指定了用哪一个协程来启动流的收集。在先前的示例中这个作用域来自 runBlocking 协程构建器，在这个流运行的时候，runBlocking 作用域等待它的子协程执行完毕并防止 main 函数返回并终止此示例。

在实际的应用中，作用域来自于一个寿命有限的实体。在该实体的寿命终止后，相应的作用域就会被取消，即取消相应流的收集。这种成对的 `onEach { ... }.launchIn(scope)` 工作方式就像 `addEventListener` 一样。而且，这不需要相应的 `removeEventListener` 函数， 因为取消与结构化并发可以达成这个目的。

注意，launchIn 也会返回一个 Job，可以在不取消整个作用域的情况下仅取消相应的流收集或对其进行 join。

流取消检测

为方便起见，流构建器对每个发射值执行附加的 ensureActive 检测以进行取消。 这意味着从 `flow { ... }` 发出的繁忙循环是可以取消的：

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

//sampleStart
fun foo(): Flow<Int> = flow {
    for (i in 1..5) {
        println("Emitting $i")
        emit(i)
    }
}

fun main() = runBlocking<Unit> {
    foo().collect { value ->
        if (value == 3) cancel()
        println(value)
    }
}
//sampleEnd
```

> 可以在这里获取完整的代码。

仅得到不超过 3 的数字，在尝试发出 4 之后抛出 CancellationException；

```
Emitting 1
1
Emitting 2
2
Emitting 3
3
Emitting 4
Exception in thread "main" kotlinx.coroutines.JobCancellationException: BlockingCorout
ine was cancelled; job="coroutine#1":BlockingCoroutine{Cancelled}@6d7b4f4c
```

但是，出于性能原因，大多数其他流操作不会自行执行其他取消检测。 例如，如果使用 IntRange.asFlow 扩展来编写相同的繁忙循环， 并且没有在任何地方暂停，那么就没有取消的检测；

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

//sampleStart
fun main() = runBlocking<Unit> {
    (1..5).asFlow().collect { value ->
        if (value == 3) cancel()
        println(value)
    }
}
//sampleEnd
```

> 可以在这里获取完整的代码。
>
> 

收集从 1 到 5 的所有数字，并且仅在从 `runBlocking` 返回之前检测到取消：

```
1
2
3
4
5
Exception in thread "main" kotlinx.coroutines.JobCancellationException: BlockingCorout
ine was cancelled; job="coroutine#1":BlockingCoroutine{Cancelled}@3327bd23
```

让繁忙的流可取消

在协程处于繁忙循环的情况下，必须明确检测是否取消。 可以添加 `.onEach { currentCoroutineContext().ensureActive() }`，但是这里提供了一个现成的 cancellable 操作符来执行此操作：

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

//sampleStart
fun main() = runBlocking<Unit> {
    (1..5).asFlow().cancellable().collect { value ->
        if (value == 3) cancel()
        println(value)
    }
}
//sampleEnd
```

> 可以在这里获取完整的代码。
>
> 

使用 `cancellable` 操作符，仅收集从 1 到 3 的数字：

```
1
2
3
Exception in thread "main" kotlinx.coroutines.JobCancellationException: BlockingCorout
ine was cancelled; job="coroutine#1":BlockingCoroutine{Cancelled}@5ec0a365
```

流（Flow）与响应式流（Reactive Streams）

对于熟悉响应式流（Reactive Streams）或诸如 RxJava 与 Project Reactor 这样的响应式框架的人来说， Flow 的设计也许看起来会非常熟悉。

确实，其设计灵感来源于响应式流以及其各种实现。但是 Flow 的主要目标是拥有尽可能简单的设计， 对 Kotlin 以及挂起友好且遵从结构化并发。没有响应式的先驱及他们大量的工作，就不可能实现这一目标。你可以阅读 Reactive Streams and Kotlin Flows 这篇文章来了解完成 Flow 的故事。

虽然有所不同，但从概念上讲，Flow 依然是响应式流，并且可以将它转换为响应式（规范及符合 TCK）的发布者（Publisher），反之亦然。 这些开箱即用的转换器可以在 `kotlinx.coroutines` 提供的相关响应式模块（ `kotlinx-coroutines-reactive` 用于 Reactive Streams， `kotlinx-coroutines-reactor` 用于 Project Reactor，以及 `kotlinx-coroutines-rx2` / `kotlinx-coroutines-rx3` 用于 RxJava2/RxJava3）中找到。 集成模块包含 `Flow` 与其他实现之间的转换，与 Reactor 的 `Context` 集成以及与一系列响应式实体配合使用的挂起友好的使用方式。

通道

延期的值提供了一种便捷的方法使单个值在多个协程之间进行相互传输。 通道提供了一种在流中传输值的方法。

通道基础

一个 Channel 是一个和 `BlockingQueue` 非常相似的概念。其中一个不同是它代替了阻塞的 `put` 操作并提供了挂起的 send，还替代了阻塞的 `take` 操作并提供了挂起的 receive。

```
import kotlinx.coroutines.*
import kotlinx.coroutines.channels.*

fun main() = runBlocking {
//sampleStart
    val channel = Channel<Int>()
    launch {
        // 这里可能是消耗大量 CPU 运算的异步逻辑，
        // 我们将仅仅做 5 次整数的平方并发送
        for (x in 1..5) channel.send(x * x)
    }
    // 这里我们打印了 5 次被接收的整数：
    repeat(5) { println(channel.receive()) }
    println("Done!")
//sampleEnd
}
```

> 可以在这里获取完整代码。

这段代码的输出如下：

```
1
4
9
16
25
Done!
```

关闭与迭代通道

和队列不同，一个通道可以通过被关闭来表明没有更多的元素将会进入通道。 在接收者中可以定期的使用 `for` 循环来从通道中接收元素。

从概念上来说，一个 close 操作就像向通道发送了一个特殊的关闭指令。 这个迭代停止就说明关闭指令已经被接收了。所以这里保证所有先前发送出去的元素都在通道关闭前被接收到。

```
import kotlinx.coroutines.*
import kotlinx.coroutines.channels.*

fun main() = runBlocking {
//sampleStart
    val channel = Channel<Int>()
    launch {
        for (x in 1..5) channel.send(x * x)
        channel.close() // 我们结束发送
    }
    // 这里我们使用 `for` 循环来打印所有被接收到的元素（直到通道被关闭）
    for (y in channel) println(y)
    println("Done!")
//sampleEnd
}
```

> 可以在这里获取完整代码。
>
> 

构建通道生产者

协程生成一系列元素的模式很常见。 这是 生产者——消费者 模式的一部分，并且经常能在并发的代码中看到它。 你可以将生产者抽象成一个函数，并且使通道作为它的参数，但这与必须从函数中返回结果的常识相违悖。

这里有一个名为 produce 的便捷的协程构建器，可以很容易的在生产者端正确工作， 并且我们使用扩展函数 consumeEach 在消费者端替代 `for` 循环：

```
import kotlinx.coroutines.*
import kotlinx.coroutines.channels.*

//sampleStart
fun CoroutineScope.produceSquares(): ReceiveChannel<Int> = produce {
    for (x in 1..5) send(x * x)
}

fun main() = runBlocking {
    val squares = produceSquares()
    squares.consumeEach { println(it) }
    println("Done!")
```

```
//sampleEnd
}
```

> 可以在这里获取完整代码。
>
> 

管道

管道是一种一个协程在流中开始生产可能无穷多个元素的模式：

```
fun CoroutineScope.produceNumbers() = produce<Int> {
    var x = 1
    while (true) send(x++) // 在流中开始从 1 生产无穷多个整数
}
```

并且另一个或多个协程开始消费这些流，做一些操作，并生产了一些额外的结果。 在下面的例子中，对这些数字仅仅做了平方操作：

```
fun CoroutineScope.square(numbers: ReceiveChannel<Int>): ReceiveChannel<Int> = produce
 {
    for (x in numbers) send(x * x)
}
```

主要的代码启动并连接了整个管道：

```
import kotlinx.coroutines.*
import kotlinx.coroutines.channels.*

fun main() = runBlocking {
//sampleStart
    val numbers = produceNumbers() // 从 1 开始生成整数
    val squares = square(numbers) // 整数求平方
    repeat(5) {
        println(squares.receive()) // 输出前五个
    }
    println("Done!") // 至此已完成
    coroutineContext.cancelChildren() // 取消子协程
//sampleEnd
}

fun CoroutineScope.produceNumbers() = produce<Int> {
    var x = 1
    while (true) send(x++) // 从 1 开始的无限的整数流
}
```

```
fun CoroutineScope.square(numbers: ReceiveChannel<Int>): ReceiveChannel<Int> = produce
 {
    for (x in numbers) send(x * x)
}
```

> 可以在这里获取完整代码。
>
> 

> 所有创建了协程的函数被定义在了 CoroutineScope 的扩展上， 所以我们可以依靠结构化并发来确保没有常驻在我们的应用程序中的全局协程。
>
> 

使用管道的素数

让我们来展示一个极端的例子——在协程中使用一个管道来生成素数。我们开启了一个数字的无限序列。

```
fun CoroutineScope.numbersFrom(start: Int) = produce<Int> {
    var x = start
    while (true) send(x++) // 开启了一个无限的整数流
}
```

在下面的管道阶段中过滤了来源于流中的数字，删除了所有可以被给定素数整除的数字。

```
fun CoroutineScope.filter(numbers: ReceiveChannel<Int>, prime: Int) = produce<Int> {
    for (x in numbers) if (x % prime != 0) send(x)
}
```

现在我们开启了一个从 2 开始的数字流管道，从当前的通道中取一个素数， 并为每一个我们发现的素数启动一个流水线阶段：

```
numbersFrom(2) -> filter(2) -> filter(3) -> filter(5) -> filter(7) ……
```

下面的例子打印了前十个素数， 在主线程的上下文中运行整个管道。直到所有的协程在该主协程 runBlocking 的作用域中被启动完成。 我们不必使用一个显式的列表来保存所有被我们已经启动的协程。 我们使用 cancelChildren 扩展函数在我们打印了前十个素数以后来取消所有的子协程。

```
import kotlinx.coroutines.*
import kotlinx.coroutines.channels.*
```

```
fun main() = runBlocking {
//sampleStart
    var cur = numbersFrom(2)
    repeat(10) {
        val prime = cur.receive()
        println(prime)
        cur = filter(cur, prime)
    }
    coroutineContext.cancelChildren() // 取消所有的子协程来让主协程结束
//sampleEnd
}

fun CoroutineScope.numbersFrom(start: Int) = produce<Int> {
    var x = start
    while (true) send(x++) // 从 start 开始过滤整数流
}

fun CoroutineScope.filter(numbers: ReceiveChannel<Int>, prime: Int) = produce<Int> {
    for (x in numbers) if (x % prime != 0) send(x)
}
```

> 可以在这里获取完整代码。
>
> 

这段代码的输出如下：

```
2
3
5
7
11
13
17
19
23
29
```

注意，你可以在标准库中使用 `iterator` 协程构建器来构建一个相似的管道。 使用 `iterator` 替换 `produce`、 `yield` 替换 `send`、 `next` 替换 `receive`、 `Iterator` 替换 `ReceiveChannel` 来摆脱协程作用域，你将不再需要 `runBlocking`。 然而，如上所示，如果你在 Dispatchers.Default 上下文中运行它，使用通道的管道的好处在于它可以充分利用多核心 CPU。

不过，这是一种非常不切实际的寻找素数的方法。在实践中，管道调用了另外的一些挂起中的调用（就像异步调用远程服务）并且这些管道不能内置使用 `sequence` / `iterator`，因为它们不被允许随意的挂起，不像 `produce` 是完全异步的。

扇出

多个协程也许会接收相同的管道，在它们之间进行分布式工作。 让我们启动一个定期产生整数的生产者协程（每秒十个数字）：

```
fun CoroutineScope.produceNumbers() = produce<Int> {
    var x = 1 // 从 1 开始
    while (true) {
        send(x++) // 产生下一个数字
        delay(100) // 等待 0.1 秒
    }
}
```

接下来我们可以得到几个处理器协程。在这个示例中，它们只是打印它们的 id 和接收到的数字：

```
fun CoroutineScope.launchProcessor(id: Int, channel: ReceiveChannel<Int>) = launch {
    for (msg in channel) {
        println("Processor #$id received $msg")
    }
}
```

现在让我们启动五个处理器协程并让它们工作将近一秒。看看发生了什么：

```
import kotlinx.coroutines.*
import kotlinx.coroutines.channels.*

fun main() = runBlocking<Unit> {
//sampleStart
    val producer = produceNumbers()
    repeat(5) { launchProcessor(it, producer) }
    delay(950)
    producer.cancel() // 取消协程生产者从而将它们全部杀死
//sampleEnd
}

fun CoroutineScope.produceNumbers() = produce<Int> {
    var x = 1 // start from 1
    while (true) {
        send(x++) // 产生下一个数字
        delay(100) // 等待 0.1 秒
    }
}

fun CoroutineScope.launchProcessor(id: Int, channel: ReceiveChannel<Int>) = launch {
    for (msg in channel) {
        println("Processor #$id received $msg")
    }
```

```
}
```

> 可以在这里获取完整代码。
>
> 

该输出将类似于如下所示，尽管接收每个特定整数的处理器 id 可能会不同：

```
Processor #2 received 1
Processor #4 received 2
Processor #0 received 3
Processor #1 received 4
Processor #3 received 5
Processor #2 received 6
Processor #4 received 7
Processor #0 received 8
Processor #1 received 9
Processor #3 received 10
```

注意，取消生产者协程将关闭它的通道，从而最终终止处理器协程正在执行的此通道上的迭代。

还有，注意我们如何使用 `for` 循环显式迭代通道以在 `launchProcessor` 代码中执行扇出。与 `consumeEach` 不同，这个 `for` 循环是安全完美地使用多个协程的。如果其中一个处理器协程执行失败，其它的处理器协程仍然会继续处理通道，而通过 `consumeEach` 编写的处理器始终在正常或非正常完成时消耗（取消）底层通道。

扇入

多个协程可以发送到同一个通道。 比如说，让我们创建一个字符串的通道，和一个在这个通道中以指定的延迟反复发送一个指定字符串的挂起函数：

```
suspend fun sendString(channel: SendChannel<String>, s: String, time: Long) {
    while (true) {
        delay(time)
        channel.send(s)
    }
}
```

现在，我们启动了几个发送字符串的协程，让我们看看会发生什么（在示例中，我们在主线程的上下文中作为主协程的子协程来启动它们）：

```
import kotlinx.coroutines.*
import kotlinx.coroutines.channels.*
```

```
fun main() = runBlocking {
//sampleStart
    val channel = Channel<String>()
    launch { sendString(channel, "foo", 200L) }
    launch { sendString(channel, "BAR!", 500L) }
    repeat(6) { // 接收前六个
        println(channel.receive())
    }
    coroutineContext.cancelChildren() // 取消所有子协程来让主协程结束
//sampleEnd
}

suspend fun sendString(channel: SendChannel<String>, s: String, time: Long) {
    while (true) {
        delay(time)
        channel.send(s)
    }
}
```

> 可以在这里获取完整代码。
>
> 

输出如下：

```
foo
foo
BAR!
foo
foo
BAR!
```

带缓冲的通道

到目前为止展示的通道都是没有缓冲区的。无缓冲的通道在发送者和接收者相遇时传输元素（也称“对接”）。如果发送先被调用，则它将被挂起直到接收被调用， 如果接收先被调用，它将被挂起直到发送被调用。

Channel() 工厂函数与 produce 建造器通过一个可选的参数 `capacity` 来指定 缓冲区大小 。缓冲允许发送者在被挂起前发送多个元素， 就像 `BlockingQueue` 有指定的容量一样，当缓冲区被占满的时候将会引起阻塞。

看看如下代码的表现：

```
import kotlinx.coroutines.*
import kotlinx.coroutines.channels.*
```

```
fun main() = runBlocking<Unit> {
//sampleStart
    val channel = Channel<Int>(4) // 启动带缓冲的通道
    val sender = launch { // 启动发送者协程
        repeat(10) {
            println("Sending $it") // 在每一个元素发送前打印它们
            channel.send(it) // 将在缓冲区被占满时挂起
        }
    }
    // 没有接收到东西……只是等待……
    delay(1000)
    sender.cancel() // 取消发送者协程
//sampleEnd
}
```

> 可以在这里获取完整代码。
>
> 

使用缓冲通道并给 capacity 参数传入 四 它将打印“sending” 五 次：

```
Sending 0
Sending 1
Sending 2
Sending 3
Sending 4
```

前四个元素被加入到了缓冲区并且发送者在试图发送第五个元素的时候被挂起。

通道是公平的

发送和接收操作是 公平的 并且尊重调用它们的多个协程。它们遵守先进先出原则，可以看到第一个协程调用 `receive` 并得到了元素。在下面的例子中两个协程“乒”和“乓”都从共享的“桌子”通道接收到这个“球”元素。

```
import kotlinx.coroutines.*
import kotlinx.coroutines.channels.*

//sampleStart
data class Ball(var hits: Int)

fun main() = runBlocking {
    val table = Channel<Ball>() // 一个共享的 table（桌子）
    launch { player("ping", table) }
    launch { player("pong", table) }
    table.send(Ball(0)) // 乒乓球
    delay(1000) // 延迟 1 秒钟
```

```
    coroutineContext.cancelChildren() // 游戏结束，取消它们
}

suspend fun player(name: String, table: Channel<Ball>) {
    for (ball in table) { // 在循环中接收球
        ball.hits++
        println("$name $ball")
        delay(300) // 等待一段时间
        table.send(ball) // 将球发送回去
    }
}
//sampleEnd
```

> 可以在这里得到完整代码
>
> 

“乒”协程首先被启动，所以它首先接收到了球。甚至虽然“乒” 协程在将球发送会桌子以后立即开始接收，但是球还是被“乓” 协程接收了，因为它一直在等待着接收球：

```
ping Ball(hits=1)
pong Ball(hits=2)
ping Ball(hits=3)
pong Ball(hits=4)
```

注意，有时候通道执行时由于执行者的性质而看起来不那么公平。点击这个提案来查看更多细节。

计时器通道

计时器通道是一种特别的会合通道，每次经过特定的延迟都会从该通道进行消费并产生 `Unit`。 虽然它看起来似乎没用，它被用来构建分段来创建复杂的基于时间的 produce 管道和进行窗口化操作以及其它时间相关的处理。 可以在 select 中使用计时器通道来进行“打勾”操作。

使用工厂方法 ticker 来创建这些通道。 为了表明不需要其它元素，请使用 ReceiveChannel.cancel 方法。

现在让我们看看它是如何在实践中工作的：

```
import kotlinx.coroutines.*
import kotlinx.coroutines.channels.*

//sampleStart
fun main() = runBlocking<Unit> {
    val tickerChannel = ticker(delayMillis = 200, initialDelayMillis = 0) //创建计时器通道
```

```
    var nextElement = withTimeoutOrNull(1) { tickerChannel.receive() }
    println("Initial element is available immediately: $nextElement") // no initial de
lay

    nextElement = withTimeoutOrNull(100) { tickerChannel.receive() } // all subsequent
 elements have 200ms delay
    println("Next element is not ready in 100 ms: $nextElement")

    nextElement = withTimeoutOrNull(120) { tickerChannel.receive() }
    println("Next element is ready in 200 ms: $nextElement")

    // 模拟大量消费延迟
    println("Consumer pauses for 300ms")
    delay(300)
    // 下一个元素立即可用
    nextElement = withTimeoutOrNull(1) { tickerChannel.receive() }
    println("Next element is available immediately after large consumer delay: $nextEl
ement")
    // 请注意，`receive` 调用之间的暂停被考虑在内，下一个元素的到达速度更快
    nextElement = withTimeoutOrNull(120) { tickerChannel.receive() }
    println("Next element is ready in 100ms after consumer pause in 300ms: $nextElement
")

    tickerChannel.cancel() // 表明不再需要更多的元素
}
//sampleEnd
```

> 可以在这里获取完整代码。

它的打印如下：

```
Initial element is available immediately: kotlin.Unit
Next element is not ready in 100 ms: null
Next element is ready in 200 ms: kotlin.Unit
Consumer pauses for 300ms
Next element is available immediately after large consumer delay: kotlin.Unit
Next element is ready in 100ms after consumer pause in 300ms: kotlin.Unit
```

请注意，ticker 知道可能的消费者暂停，并且默认情况下会调整下一个生成的元素如果发生暂停则延迟，试图保持固定的生成元素率。

给可选的 `mode` 参数传入 TickerMode.FIXED_DELAY 可以保持固定元素之间的延迟。

协程异常处理

本节内容涵盖了异常处理与在异常上取消。 我们已经知道被取消的协程会在挂起点抛出 CancellationException 并且它会被协程的机制所忽略。在这里我们会看看在取消过程中抛出异常或同一个协程的多个子协程抛出异常时会发生什么。

异常的传播

协程构建器有两种形式：自动传播异常（launch）或向用户暴露异常（async 与 produce）。 当这些构建器用于创建一个根协程时，即该协程不是另一个协程的子协程， 前者这类构建器将异常视为未捕获异常，类似 Java 的 `Thread.uncaughtExceptionHandler`， 而后者则依赖用户来最终消费异常，例如通过 await 或 receive（produce 与 receive 的相关内容包含于通道章节）。

可以通过一个使用 GlobalScope 创建根协程的简单示例来进行演示：

> GlobalScope is a delicate API that can backfire in non-trivial ways. Creating a root coroutine for the whole application is one of the rare legitimate uses for `GlobalScope`, so you must explicitly opt-in into using `GlobalScope` with `@OptIn(DelicateCoroutinesApi::class)`.

```
import kotlinx.coroutines.*

//sampleStart
@OptIn(DelicateCoroutinesApi::class)
fun main() = runBlocking {
    val job = GlobalScope.launch { // launch 根协程
        println("Throwing exception from launch")
        throw IndexOutOfBoundsException() // 我们将在控制台打印 Thread.defaultUncaughtExc
eptionHandler
    }
    job.join()
    println("Joined failed job")
    val deferred = GlobalScope.async { // async 根协程
        println("Throwing exception from async")
        throw ArithmeticException() // 没有打印任何东西，依赖用户去调用等待
    }
    try {
        deferred.await()
        println("Unreached")
    } catch (e: ArithmeticException) {
        println("Caught ArithmeticException")
    }
}
//sampleEnd
```

> 可以在这里获取完整代码。

这段代码的输出如下（调试）：

```
Throwing exception from launch
Exception in thread "DefaultDispatcher-worker-1 @coroutine#2" java.lang.IndexOutOfBoun
dsException
Joined failed job
Throwing exception from async
Caught ArithmeticException
```

CoroutineExceptionHandler

将未捕获异常打印到控制台的默认行为是可自定义的。 根协程中的 CoroutineExceptionHandler 上下文元素可以被用于这个根协程通用的 `catch` 块，及其所有可能自定义了异常处理的子协程。 它类似于 `Thread.uncaughtExceptionHandler` 。 你无法从 `CoroutineExceptionHandler` 的异常中恢复。当调用处理者的时候，协程已经完成并带有相应的异常。通常，该处理者用于记录异常，显示某种错误消息，终止和（或）重新启动应用程序。

`CoroutineExceptionHandler` 仅在未捕获的异常上调用 — 没有以其他任何方式处理的异常。特别是，所有子协程（在另一个 Job 上下文中创建的协程）委托<!-- 它们的父协程处理它们的异常，然后它们也委托给其父协程，以此类推直到根协程， 因此永远不会使用在其上下文中设置的 `CoroutineExceptionHandler` 。 除此之外，async 构建器始终会捕获所有异常并将其表示在结果 Deferred 对象中， 因此它的 `CoroutineExceptionHandler` 也无效。

> 在监督作用域内运行的协程不会将异常传播到其父协程，并且会从此规则中排除。本文档的另一个小节——监督提供了更多细节。

```
import kotlinx.coroutines.*

@OptIn(DelicateCoroutinesApi::class)
fun main() = runBlocking {
//sampleStart
    val handler = CoroutineExceptionHandler { _, exception ->
        println("CoroutineExceptionHandler got $exception")
    }
    val job = GlobalScope.launch(handler) { // 根协程，运行在 GlobalScope 中
        throw AssertionError()
    }
    val deferred = GlobalScope.async(handler) { // 同样是根协程，但使用 async 代替了 launch
```

```
        throw ArithmeticException() // 没有打印任何东西，依赖用户去调用 deferred.await()
    }
    joinAll(job, deferred)
//sampleEnd
}
```

> 可以在这里获取完整代码。
>
> 

这段代码的输出如下：

```
CoroutineExceptionHandler got java.lang.AssertionError
```

取消与异常

取消与异常紧密相关。协程内部使用 `CancellationException` 来进行取消，这个异常会被所有的处理者忽略，所以那些可以被 `catch` 代码块捕获的异常仅仅应该被用来作为额外调试信息的资源。 当一个协程使用 Job.cancel 取消的时候，它会被终止，但是它不会取消它的父协程。

```
import kotlinx.coroutines.*

fun main() = runBlocking {
//sampleStart
    val job = launch {
        val child = launch {
            try {
                delay(Long.MAX_VALUE)
            } finally {
                println("Child is cancelled")
            }
        }
        yield()
        println("Cancelling child")
        child.cancel()
        child.join()
        yield()
        println("Parent is not cancelled")
    }
    job.join()
//sampleEnd
}
```

> 可以在这里获取完整代码。



这段代码的输出如下：

```
Cancelling child
Child is cancelled
Parent is not cancelled
```

如果一个协程遇到了 `CancellationException` 以外的异常，它将使用该异常取消它的父协程。这个行为无法被覆盖，并且用于为结构化的并发（structured concurrency）提供稳定的协程层级结构。 CoroutineExceptionHandler 的实现并不是用于子协程。

> 在这些示例中，CoroutineExceptionHandler 总是被设置在由 GlobalScope 启动的协程中。将异常处理者设置在 runBlocking 主作用域内启动的协程中是没有意义的，尽管子协程已经设置了异常处理者， 但是主协程也总是会被取消的。



当父协程的所有子协程都结束后，原始的异常才会被父协程处理， 见下面这个例子。

```
import kotlinx.coroutines.*

@OptIn(DelicateCoroutinesApi::class)
fun main() = runBlocking {
//sampleStart
    val handler = CoroutineExceptionHandler { _, exception ->
        println("CoroutineExceptionHandler got $exception")
    }
    val job = GlobalScope.launch(handler) {
        launch { // 第一个子协程
            try {
                delay(Long.MAX_VALUE)
            } finally {
                withContext(NonCancellable) {
                    println("Children are cancelled, but exception is not handled unti
l all children terminate")
                    delay(100)
                    println("The first child finished its non cancellable block")
                }
            }
        }
        launch { // 第二个子协程
            delay(10)
            println("Second child throws an exception")
            throw ArithmeticException()
        }
    }
    job.join()
//sampleEnd
}
```

> 可以在这里获取完整代码。
>
> 

这段代码的输出如下：

```
Second child throws an exception
Children are cancelled, but exception is not handled until all children terminate
The first child finished its non cancellable block
CoroutineExceptionHandler got java.lang.ArithmeticException
```

异常聚合

当协程的多个子协程因异常而失败时， 一般规则是“取第一个异常”，因此将处理第一个异常。 在第一个异常之后发生的所有其他异常都作为被抑制的异常绑定至第一个异常。

```
import kotlinx.coroutines.*
import java.io.*

@OptIn(DelicateCoroutinesApi::class)
fun main() = runBlocking {
    val handler = CoroutineExceptionHandler { _, exception ->
        println("CoroutineExceptionHandler got $exception with suppressed ${exception.
suppressed.contentToString()}")
    }
    val job = GlobalScope.launch(handler) {
        launch {
            try {
                delay(Long.MAX_VALUE) // 当另一个同级的协程因 IOException  失败时，它将被取消

            } finally {
                throw ArithmeticException() // 第二个异常
            }
        }
        launch {
            delay(100)
            throw IOException() // 首个异常
        }
        delay(Long.MAX_VALUE)
    }
    job.join()
}
```

> 可以在这里获取完整代码。



这段代码的输出如下：

```
CoroutineExceptionHandler got java.io.IOException with suppressed [java.lang.Arithmeti
cException]
```

> 注意，这个机制当前只能在 Java 1.7 以上的版本中使用。 在 JS 和原生环境下暂时会受到限制，但将来会取消。



取消异常是透明的，默认情况下是未包装的：

```
import kotlinx.coroutines.*
import java.io.*

@OptIn(DelicateCoroutinesApi::class)
fun main() = runBlocking {
//sampleStart
    val handler = CoroutineExceptionHandler { _, exception ->
        println("CoroutineExceptionHandler got $exception")
    }
    val job = GlobalScope.launch(handler) {
        val innerJob = launch { // 该栈内的协程都将被取消
            launch {
                launch {
                    throw IOException() // 原始异常
                }
            }
        }
        try {
            innerJob.join()
        } catch (e: CancellationException) {
            println("Rethrowing CancellationException with original cause")
            throw e // 取消异常被重新抛出，但原始 IOException 得到了处理
        }
    }
    job.join()
//sampleEnd
}
```

> 可以在这里获取完整代码。



这段代码的输出如下：

```
Rethrowing CancellationException with original cause
```

```
CoroutineExceptionHandler got java.io.IOException
```

监督

正如我们之前研究的那样，取消是在协程的整个层次结构中传播的双向关系。让我们看一下需要单向取消的情况。

此类需求的一个良好示例是在其作用域内定义作业的 UI 组件。如果任何一个 UI 的子作业执行失败了，它并不总是有必要取消（有效地杀死）整个 UI 组件， 但是如果 UI 组件被销毁了（并且它的作业也被取消了），由于其结果不再需要了，因此有必要取消所有子作业。

另一个例子是服务进程孵化了一些子作业并且需要 监督 它们的执行，追踪它们的故障并在这些子作业执行失败的时候重启。

监督作业

SupervisorJob 可以用于这些目的。 它类似于常规的 Job，唯一的不同是：SupervisorJob 的取消只会向下传播。这是很容易用以下示例演示：

```
import kotlinx.coroutines.*

fun main() = runBlocking {
//sampleStart
    val supervisor = SupervisorJob()
    with(CoroutineScope(coroutineContext + supervisor)) {
        // 启动第一个子作业——这个示例将会忽略它的异常（不要在实践中这么做！）
        val firstChild = launch(CoroutineExceptionHandler { _, _ ->  }) {
            println("The first child is failing")
            throw AssertionError("The first child is cancelled")
        }
        // 启动第二个子作业
        val secondChild = launch {
            firstChild.join()
            // 取消了第一个子作业且没有传播给第二个子作业
            println("The first child is cancelled: ${firstChild.isCancelled}, but the
second one is still active")
            try {
                delay(Long.MAX_VALUE)
            } finally {
                // 但是取消了监督的传播
                println("The second child is cancelled because the supervisor was canc
elled")
            }
        }
        // 等待直到第一个子作业失败且执行完成
        firstChild.join()
        println("Cancelling the supervisor")
```

```
        supervisor.cancel()
        secondChild.join()
    }
//sampleEnd
}
```

> 可以在这里获取完整代码。
>
> 

这段代码的输出如下：

```
The first child is failing
The first child is cancelled: true, but the second one is still active
Cancelling the supervisor
The second child is cancelled because the supervisor was cancelled
```

监督作用域

对于作用域的并发，可以用 supervisorScope 来替代 coroutineScope 来实现相同的目的。它只会单向的传播并且当作业自身执行失败的时候将所有子作业全部取消。作业自身也会在所有的子作业执行结束前等待， 就像 coroutineScope 所做的那样。

```
import kotlin.coroutines.*
import kotlinx.coroutines.*

fun main() = runBlocking {
//sampleStart
    try {
        supervisorScope {
            val child = launch {
                try {
                    println("The child is sleeping")
                    delay(Long.MAX_VALUE)
                } finally {
                    println("The child is cancelled")
                }
            }
            // 使用 yield 来给我们的子作业一个机会来执行打印
            yield()
            println("Throwing an exception from the scope")
            throw AssertionError()
        }
    } catch(e: AssertionError) {
        println("Caught an assertion error")
    }
//sampleEnd
}
```

> 可以在这里获取完整代码。
>
> 

这段代码的输出如下：

```
The child is sleeping
Throwing an exception from the scope
The child is cancelled
Caught an assertion error
```

监督协程中的异常

常规的作业和监督作业之间的另一个重要区别是异常处理。 监督协程中的每一个子作业应该通过异常处理机制处理自身的异常。 这种差异来自于子作业的执行失败不会传播给它的父作业的事实。 这意味着在 supervisorScope 内部直接启动的协程确实使用了设置在它们作用域内的 CoroutineExceptionHandler，与父协程的方式相同（参见 CoroutineExceptionHandler 小节以获知更多细节）。

```
import kotlin.coroutines.*
import kotlinx.coroutines.*

fun main() = runBlocking {
//sampleStart
    val handler = CoroutineExceptionHandler { _, exception ->
        println("CoroutineExceptionHandler got $exception")
    }
    supervisorScope {
        val child = launch(handler) {
            println("The child throws an exception")
            throw AssertionError()
        }
        println("The scope is completing")
    }
    println("The scope is completed")
//sampleEnd
}
```

> 可以在这里获取完整代码。
>
> 

这段代码的输出如下：

```
The scope is completing
The child throws an exception
CoroutineExceptionHandler got java.lang.AssertionError
```

```
The scope is completed
```

共享的可变状态与并发

协程可用多线程调度器（比如默认的 Dispatchers.Default ）并行执行。这样就可以提出所有常见的并行问题。主要的问题是同步访问共享的可变状态。 协程领域对这个问题的一些解决方案类似于多线程领域中的解决方案， 但其它解决方案则是独一无二的。

问题

我们启动一百个协程，它们都做一千次相同的操作。我们同时会测量它们的完成时间以便进一步的比较：

```
suspend fun massiveRun(action: suspend () -> Unit) {
    val n = 100  // 启动的协程数量
    val k = 1000 // 每个协程重复执行同一动作的次数
    val time = measureTimeMillis {
        coroutineScope { // 协程的作用域
            repeat(n) {
                launch {
                    repeat(k) { action() }
                }
            }
        }
    }
    println("Completed ${n * k} actions in $time ms")
}
```

我们从一个非常简单的动作开始：使用多线程的 Dispatchers.Default 来递增一个共享的可变变量。

```
import kotlinx.coroutines.*
import kotlin.system.*

suspend fun massiveRun(action: suspend () -> Unit) {
    val n = 100  // 启动的协程数量
    val k = 1000 // 每个协程重复执行同一动作的次数
    val time = measureTimeMillis {
        coroutineScope { // 协程的作用域
            repeat(n) {
                launch {
                    repeat(k) { action() }
                }
            }
        }
    }
```

```
    println("Completed ${n * k} actions in $time ms")
}

//sampleStart
var counter = 0

fun main() = runBlocking {
    withContext(Dispatchers.Default) {
        massiveRun {
            counter++
        }
    }
    println("Counter = $counter")
}
//sampleEnd
```

> 可以在这里获取完整代码。
>
> 

这段代码最后打印出什么结果？它不太可能打印出“Counter = 100000”，因为一百个协程在多个线程中同时递增计数器但没有做并发处理。

volatile 无济于事

有一种常见的误解：volatile 可以解决并发问题。让我们尝试一下：

```
import kotlinx.coroutines.*
import kotlin.system.*

suspend fun massiveRun(action: suspend () -> Unit) {
    val n = 100  // 启动的协程数量
    val k = 1000 // 每个协程重复执行同一动作的次数
    val time = measureTimeMillis {
        coroutineScope { // 协程的作用域
            repeat(n) {
                launch {
                    repeat(k) { action() }
                }
            }
        }
    }
    println("Completed ${n * k} actions in $time ms")
}

//sampleStart
@Volatile // 在 Kotlin 中 `volatile` 是一个注解
var counter = 0
```

```
fun main() = runBlocking {
    withContext(Dispatchers.Default) {
        massiveRun {
            counter++
        }
    }
    println("Counter = $counter")
}
//sampleEnd
```

> 可以在这里获取完整代码。
>
> 

这段代码运行速度更慢了，但我们仍然不总是最后得到“Counter = 100000”这个结果，因为 volatile 变量保证可线性化（这是“原子”的技术术语）读取和写入变量，但在大量动作（在我们的示例中即“递增”操作）发生时并不提供原子性。

线程安全的数据结构

一种对线程、协程都有效的常规解决方法，就是使用线程安全（也称为同步的、 可线性化、原子）的数据结构，它为需要在共享状态上执行的相应操作提供所有必需的同步处理。在简单的计数器场景中，我们可以使用具有 `incrementAndGet` 原子操作的 `AtomicInteger` 类：

```
import kotlinx.coroutines.*
import java.util.concurrent.atomic.*
import kotlin.system.*

suspend fun massiveRun(action: suspend () -> Unit) {
    val n = 100  // 启动的协程数量
    val k = 1000 // 每个协程重复执行同一动作的次数
    val time = measureTimeMillis {
        coroutineScope { // 协程的作用域
            repeat(n) {
                launch {
                    repeat(k) { action() }
                }
            }
        }
    }
    println("Completed ${n * k} actions in $time ms")
}

//sampleStart
val counter = AtomicInteger()

fun main() = runBlocking {
    withContext(Dispatchers.Default) {
```

```
        massiveRun {
            counter.incrementAndGet()
        }
    }
    println("Counter = $counter")
}
//sampleEnd
```

> 可以在这里获取完整代码。
>
> 

这是针对此类特定问题的最快解决方案。它适用于普通计数器、集合、队列和其他标准数据结构以及它们的基本操作。然而，它并不容易被扩展来应对复杂状态、或一些没有现成的线程安全实现的复杂操作。

以细粒度限制线程

限制线程 是解决共享可变状态问题的一种方案：对特定共享状态的所有访问权都限制在单个线程中。它通常应用于 UI 程序中：所有 UI 状态都局限于单个事件分发线程或应用主线程中。这在协程中很容易实现，通过使用一个单线程上下文。

```
import kotlinx.coroutines.*
import kotlin.system.*

suspend fun massiveRun(action: suspend () -> Unit) {
    val n = 100  // 启动的协程数量
    val k = 1000 // 每个协程重复执行同一动作的次数
    val time = measureTimeMillis {
        coroutineScope { // 协程的作用域
            repeat(n) {
                launch {
                    repeat(k) { action() }
                }
            }
        }
    }
    println("Completed ${n * k} actions in $time ms")
}

//sampleStart
val counterContext = newSingleThreadContext("CounterContext")
var counter = 0

fun main() = runBlocking {
    withContext(Dispatchers.Default) {
        massiveRun {
            // 将每次自增限制在单线程上下文中
```

```
            withContext(counterContext) {
                counter++
            }
        }
    }
    println("Counter = $counter")
}
//sampleEnd
```

> 可以在这里获取完整代码。
>
> 

这段代码运行非常缓慢，因为它进行了 细粒度 的线程限制。每个增量操作都得使用 [withContext(counterContext)] 块从多线程 Dispatchers.Default 上下文切换到单线程上下文。

以粗粒度限制线程

在实践中，线程限制是在大段代码中执行的，例如：状态更新类业务逻辑中大部分都是限于单线程中。下面的示例演示了这种情况， 在单线程上下文中运行每个协程。

```
import kotlinx.coroutines.*
import kotlin.system.*

suspend fun massiveRun(action: suspend () -> Unit) {
    val n = 100  // 启动的协程数量
    val k = 1000 // 每个协程重复执行同一动作的次数
    val time = measureTimeMillis {
        coroutineScope { // 协程的作用域
            repeat(n) {
                launch {
                    repeat(k) { action() }
                }
            }
        }
    }
    println("Completed ${n * k} actions in $time ms")
}

//sampleStart
val counterContext = newSingleThreadContext("CounterContext")
var counter = 0

fun main() = runBlocking {
    // 将一切都限制在单线程上下文中
    withContext(counterContext) {
        massiveRun {
            counter++
        }
```

```
    }
    println("Counter = $counter")
}
//sampleEnd
```

> 可以在这里获取完整代码。
>
> 

这段代码运行更快而且打印出了正确的结果。

互斥

该问题的互斥解决方案：使用永远不会同时执行的 关键代码块 来保护共享状态的所有修改。在阻塞的世界中，你通常会为此目的使用 `synchronized` 或者 `ReentrantLock`。 在协程中的替代品叫做 Mutex。它具有 lock 和 unlock 方法， 可以隔离关键的部分。关键的区别在于 `Mutex.lock()` 是一个挂起函数，它不会阻塞线程。

还有 withLock 扩展函数，可以方便的替代常用的 `mutex.lock(); try { …… } finally { mutex.unlock() }` 模式：

```
import kotlinx.coroutines.*
import kotlinx.coroutines.sync.*
import kotlin.system.*

suspend fun massiveRun(action: suspend () -> Unit) {
    val n = 100  // 启动的协程数量
    val k = 1000 // 每个协程重复执行同一动作的次数
    val time = measureTimeMillis {
        coroutineScope { // 协程的作用域
            repeat(n) {
                launch {
                    repeat(k) { action() }
                }
            }
        }
    }
    println("Completed ${n * k} actions in $time ms")
}

//sampleStart
val mutex = Mutex()
var counter = 0

fun main() = runBlocking {
    withContext(Dispatchers.Default) {
        massiveRun {
            // 用锁保护每次自增
```

```
            mutex.withLock {
                counter++
            }
        }
    }
    println("Counter = $counter")
}
//sampleEnd
```

> 可以在这里获取完整代码。
>
> 

此示例中锁是细粒度的，因此会付出一些代价。但是对于某些必须定期修改共享状态的场景，它是一个不错的选择，但是没有自然线程可以限制此状态。

select 表达式（实验性的）

select 表达式可以同时等待多个挂起函数，并 选择 第一个可用的。

> Select 表达式在 `kotlinx.coroutines` 中是一个实验性的特性。这些 API 在 `kotlinx.coroutines` 库即将到来的更新中可能会发生改变。



在通道中 select

我们现在有两个字符串生产者：`fizz` 与 `buzz` 。其中 `fizz` 每 500 毫秒生成一个“Fizz”字符串：

```
fun CoroutineScope.fizz() = produce<String> {
    while (true) { // 每 500 毫秒发送一个 "Fizz"
        delay(500)
        send("Fizz")
    }
}
```

接着 `buzz` 每 1000 毫秒生成一个 “Buzz!” 字符串：

```
fun CoroutineScope.buzz() = produce<String> {
    while (true) { // 每 1000 毫秒发送一个"Buzz!"
        delay(1000)
        send("Buzz!")
    }
}
```

使用 receive 挂起函数，我们可以从两个通道接收 其中一个 的数据。 但是 select 表达式允许我们使用其 onReceive 子句 同时 从两者接收：

```
suspend fun selectFizzBuzz(fizz: ReceiveChannel<String>, buzz: ReceiveChannel<String>)
 {
    select<Unit> { // <Unit> 意味着该 select 表达式不返回任何结果
        fizz.onReceive { value ->  // 这是第一个 select 子句
            println("fizz -> '$value'")
        }
        buzz.onReceive { value ->  // 这是第二个 select 子句
            println("buzz -> '$value'")
        }
    }
```

```
}
```

让我们运行代码 7 次：

```
import kotlinx.coroutines.*
import kotlinx.coroutines.channels.*
import kotlinx.coroutines.selects.*

fun CoroutineScope.fizz() = produce<String> {
    while (true) { // 每 500 毫秒发送一个 "Fizz"
        delay(500)
        send("Fizz")
    }
}

fun CoroutineScope.buzz() = produce<String> {
    while (true) { // 每 1000 毫秒发送一个 "Buzz!"
        delay(1000)
        send("Buzz!")
    }
}

suspend fun selectFizzBuzz(fizz: ReceiveChannel<String>, buzz: ReceiveChannel<String>)
 {
    select<Unit> { // <Unit> 意味着该 select 表达式不返回任何结果
        fizz.onReceive { value ->  // 这是第一个 select 子句
            println("fizz -> '$value'")
        }
        buzz.onReceive { value ->  // 这是第二个 select 子句
            println("buzz -> '$value'")
        }
    }
}

fun main() = runBlocking<Unit> {
//sampleStart
    val fizz = fizz()
    val buzz = buzz()
    repeat(7) {
        selectFizzBuzz(fizz, buzz)
    }
    coroutineContext.cancelChildren() // 取消 fizz 和 buzz 协程
//sampleEnd
}
```

> 可以在这里获取完整代码。
>
> 

这段代码的执行结果如下：

```
fizz -> 'Fizz'
buzz -> 'Buzz!'
fizz -> 'Fizz'
fizz -> 'Fizz'
buzz -> 'Buzz!'
fizz -> 'Fizz'
fizz -> 'Fizz'
```

通道关闭时 select

select 中的 onReceive 子句在已经关闭的通道执行会发生失败，并导致相应的 `select` 抛出异常。我们可以使用 onReceiveCatching 子句在关闭通道时执行特定操作。以下示例还显示了 `select` 是一个返回其查询方法结果的表达式：

```
suspend fun selectAorB(a: ReceiveChannel<String>, b: ReceiveChannel<String>): String =
    select<String> {
        a.onReceiveCatching { it ->
            val value = it.getOrNull()
            if (value != null) {
                "a -> '$value'"
            } else {
                "Channel 'a' is closed"
            }
        }
        b.onReceiveCatching { it ->
            val value = it.getOrNull()
            if (value != null) {
                "b -> '$value'"
            } else {
                "Channel 'b' is closed"
            }
        }
    }
```

现在有一个生成四次“Hello”字符串的 `a` 通道， 和一个生成四次“World”字符串的 `b` 通道，我们在这两个通道上使用它：

```
import kotlinx.coroutines.*
import kotlinx.coroutines.channels.*
import kotlinx.coroutines.selects.*

suspend fun selectAorB(a: ReceiveChannel<String>, b: ReceiveChannel<String>): String =
    select<String> {
        a.onReceiveCatching { it ->
            val value = it.getOrNull()
            if (value != null) {
                "a -> '$value'"
```

```
            } else {
                "Channel 'a' is closed"
            }
        }
        b.onReceiveCatching { it ->
            val value = it.getOrNull()
            if (value != null) {
                "b -> '$value'"
            } else {
                "Channel 'b' is closed"
            }
        }
    }

fun main() = runBlocking<Unit> {
//sampleStart
    val a = produce<String> {
        repeat(4) { send("Hello $it") }
    }
    val b = produce<String> {
        repeat(4) { send("World $it") }
    }
    repeat(8) { // 打印最早的八个结果
        println(selectAorB(a, b))
    }
    coroutineContext.cancelChildren()
//sampleEnd
}
```

> 可以在这里获取完整代码。

这段代码的结果非常有趣，所以我们会在更多细节中分析它：

```
a -> 'Hello 0'
a -> 'Hello 1'
b -> 'World 0'
a -> 'Hello 2'
a -> 'Hello 3'
b -> 'World 1'
Channel 'a' is closed
Channel 'a' is closed
```

有几个结果可以通过观察得出。

首先，`select` 偏向于 第一个子句，当可以同时选到多个子句时， 第一个子句将被选中。在这里，两个通道都在不断地生成字符串，因此 `a` 通道作为 select 中的第一个子句获胜。然而因为我们使用的是无缓冲通道，所以 `a` 在其调用 send 时会不时地被挂起，进而 `b` 也有机会发送。

第二个观察结果是，当通道已经关闭时， 会立即选择 onReceiveCatching。

Select 以发送

Select 表达式具有 onSend 子句，可以很好的与选择的偏向特性结合使用。

我们来编写一个整数生成器的示例，当主通道上的消费者无法跟上它时，它会将值发送到 `side` 通道上：

```
fun CoroutineScope.produceNumbers(side: SendChannel<Int>) = produce<Int> {
    for (num in 1..10) { // 生产从 1 到 10 的 10 个数值
        delay(100) // 延迟 100 毫秒
        select<Unit> {
            onSend(num) {} // 发送到主通道
            side.onSend(num) {} // 或者发送到 side 通道
        }
    }
}
```

消费者将会非常缓慢，每个数值处理需要 250 毫秒：

```
import kotlinx.coroutines.*
import kotlinx.coroutines.channels.*
import kotlinx.coroutines.selects.*

fun CoroutineScope.produceNumbers(side: SendChannel<Int>) = produce<Int> {
    for (num in 1..10) { // 生产从 1 到 10 的 10 个数值
        delay(100) // 延迟 100 毫秒
        select<Unit> {
            onSend(num) {} // 发送到主通道
            side.onSend(num) {} // 或者发送到 side 通道
        }
    }
}

fun main() = runBlocking<Unit> {
//sampleStart
    val side = Channel<Int>() // 分配 side 通道
    launch { // 对于 side 通道来说，这是一个很快的消费者
        side.consumeEach { println("Side channel has $it") }
    }
    produceNumbers(side).consumeEach {
        println("Consuming $it")
        delay(250) // 不要着急，让我们正确消化消耗被发送来的数字
    }
    println("Done consuming")
    coroutineContext.cancelChildren()
//sampleEnd
```

```
}
```

> 可以在这里获取完整代码。
>
> 

让我们看看会发生什么：

```
Consuming 1
Side channel has 2
Side channel has 3
Consuming 4
Side channel has 5
Side channel has 6
Consuming 7
Side channel has 8
Side channel has 9
Consuming 10
Done consuming
```

Select 延迟值

延迟值可以使用 onAwait 子句查询。 让我们启动一个异步函数，它在随机的延迟后会延迟返回字符串：

```
fun CoroutineScope.asyncString(time: Int) = async {
    delay(time.toLong())
    "Waited for $time ms"
}
```

让我们随机启动十余个异步函数，每个都延迟随机的时间。

```
fun CoroutineScope.asyncStringsList(): List<Deferred<String>> {
    val random = Random(3)
    return List(12) { asyncString(random.nextInt(1000)) }
}
```

现在 main 函数在等待第一个函数完成，并统计仍处于激活状态的延迟值的数量。注意，我们在这里使用 `select` 表达式事实上是作为一种 Kotlin DSL， 所以我们可以用任意代码为它提供子句。在这种情况下，我们遍历一个延迟值的队列，并为每个延迟值提供 `onAwait` 子句的调用。

```
import kotlinx.coroutines.*
import kotlinx.coroutines.selects.*
```

```
import java.util.*

fun CoroutineScope.asyncString(time: Int) = async {
    delay(time.toLong())
    "Waited for $time ms"
}

fun CoroutineScope.asyncStringsList(): List<Deferred<String>> {
    val random = Random(3)
    return List(12) { asyncString(random.nextInt(1000)) }
}

fun main() = runBlocking<Unit> {
//sampleStart
    val list = asyncStringsList()
    val result = select<String> {
        list.withIndex().forEach { (index, deferred) ->
            deferred.onAwait { answer ->
                "Deferred $index produced answer '$answer'"
            }
        }
    }
    println(result)
    val countActive = list.count { it.isActive }
    println("$countActive coroutines are still active")
//sampleEnd
}
```

> 可以在这里获取完整代码。
>
> 

该输出如下：

```
Deferred 4 produced answer 'Waited for 128 ms'
11 coroutines are still active
```

在延迟值通道上切换

我们现在来编写一个通道生产者函数，它消费一个产生延迟字符串的通道，并等待每个接收的延迟值，但它只在下一个延迟值到达或者通道关闭之前处于运行状态。此示例将 onReceiveCatching 和 onAwait 子句放在同一个 `select` 中：

```
fun CoroutineScope.switchMapDeferreds(input: ReceiveChannel<Deferred<String>>) = produ
ce<String> {
    var current = input.receive() // 从第一个接收到的延迟值开始
    while (isActive) { // 循环直到被取消或关闭
        val next = select<Deferred<String>?> { // 从这个 select 中返回下一个延迟值或 null
```

```
            input.onReceiveCatching { update ->
                update.getOrNull()
            }
            current.onAwait { value ->
                send(value) // 发送当前延迟生成的值
                input.receiveCatching().getOrNull() // 然后使用从输入通道得到的下一个延迟值
            }
        }
        if (next == null) {
            println("Channel was closed")
            break // 跳出循环
        } else {
            current = next
        }
    }
}
```

为了测试它，我们将用一个简单的异步函数，它在特定的延迟后返回特定的字符串：

```
fun CoroutineScope.asyncString(str: String, time: Long) = async {
    delay(time)
    str
}
```

main 函数只是启动一个协程来打印 `switchMapDeferreds` 的结果并向它发送一些测试数据：

```
import kotlinx.coroutines.*
import kotlinx.coroutines.channels.*
import kotlinx.coroutines.selects.*

fun CoroutineScope.switchMapDeferreds(input: ReceiveChannel<Deferred<String>>) = produ
ce<String> {
    var current = input.receive() // 从第一个接收到的延迟值开始
    while (isActive) { // 循环直到被取消或关闭
        val next = select<Deferred<String>?> { // 从这个 select 中返回下一个延迟值或 null
            input.onReceiveCatching { update ->
                update.getOrNull()
            }
            current.onAwait { value ->
                send(value) // 发送当前延迟生成的值
                input.receiveCatching().getOrNull() // 然后使用从输入通道得到的下一个延迟值
            }
        }
        if (next == null) {
            println("Channel was closed")
            break // 跳出循环
        } else {
            current = next
        }
    }
```

```
}

fun CoroutineScope.asyncString(str: String, time: Long) = async {
    delay(time)
    str
}

fun main() = runBlocking<Unit> {
//sampleStart
    val chan = Channel<Deferred<String>>() // 测试使用的通道
    launch { // 启动打印协程
        for (s in switchMapDeferreds(chan))
            println(s) // 打印每个获得的字符串
    }
    chan.send(asyncString("BEGIN", 100))
    delay(200) // 充足的时间来生产 "BEGIN"
    chan.send(asyncString("Slow", 500))
    delay(100) // 不充足的时间来生产 "Slow"
    chan.send(asyncString("Replace", 100))
    delay(500) // 在最后一个前给它一点时间
    chan.send(asyncString("END", 500))
    delay(1000) // 给执行一段时间
    chan.close() // 关闭通道……
    delay(500) // 然后等待一段时间来让它结束
//sampleEnd
}
```

> 可以在这里获取完整代码。
>
> 

这段代码的执行结果：

```
BEGIN
Replace
END
Channel was closed
```

使用 IntelliJ IDEA 调试协程——教程

This tutorial demonstrates how to create Kotlin coroutines and debug them using IntelliJ IDEA.

The tutorial assumes you have prior knowledge of the coroutines concept.

Create coroutines

1. Open a Kotlin project in IntelliJ IDEA. If you don't have a project, create one.
2. To use the `kotlinx.coroutines` library in a Gradle project, add the following dependency to `build.gradle(.kts)`:

> 【Kotlin】

```
dependencies {
    implementation("org.jetbrains.kotlinx:kotlinx-coroutines-core:1.10.2")
}
```

> 【Groovy】

```
dependencies {
    implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-core:1.10.2'
}
```

For other build systems, see instructions in the `kotlinx.coroutines` README.

1. Open the `Main.kt` file in `src/main/kotlin`.

   The `src` directory contains Kotlin source files and resources. The `Main.kt` file contains sample code that will print `Hello World!`.

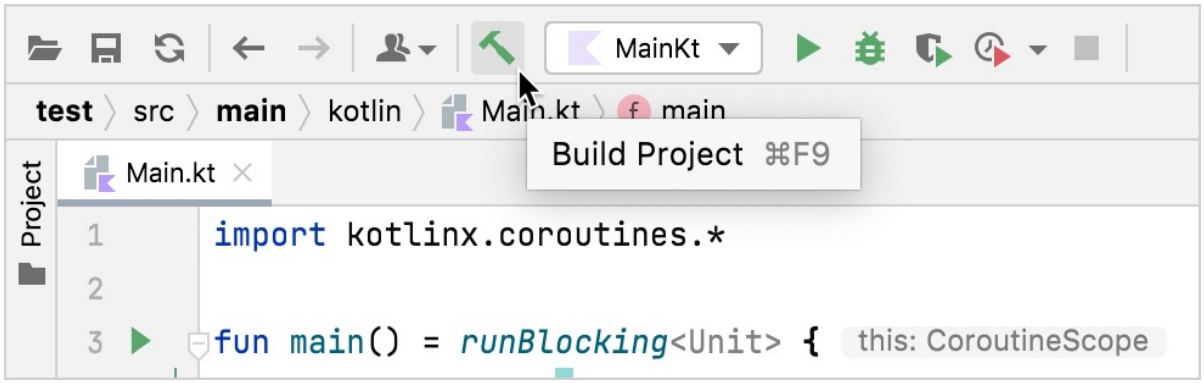
2. Change code in the `main()` function:

   - Use the `runBlocking()` block to wrap a coroutine.
   - Use the `async()` function to create coroutines that compute deferred values `a` and `b`.
   - Use the `await()` function to await the computation result.

- Use the `println()` function to print computing status and the result of multiplication to the output.

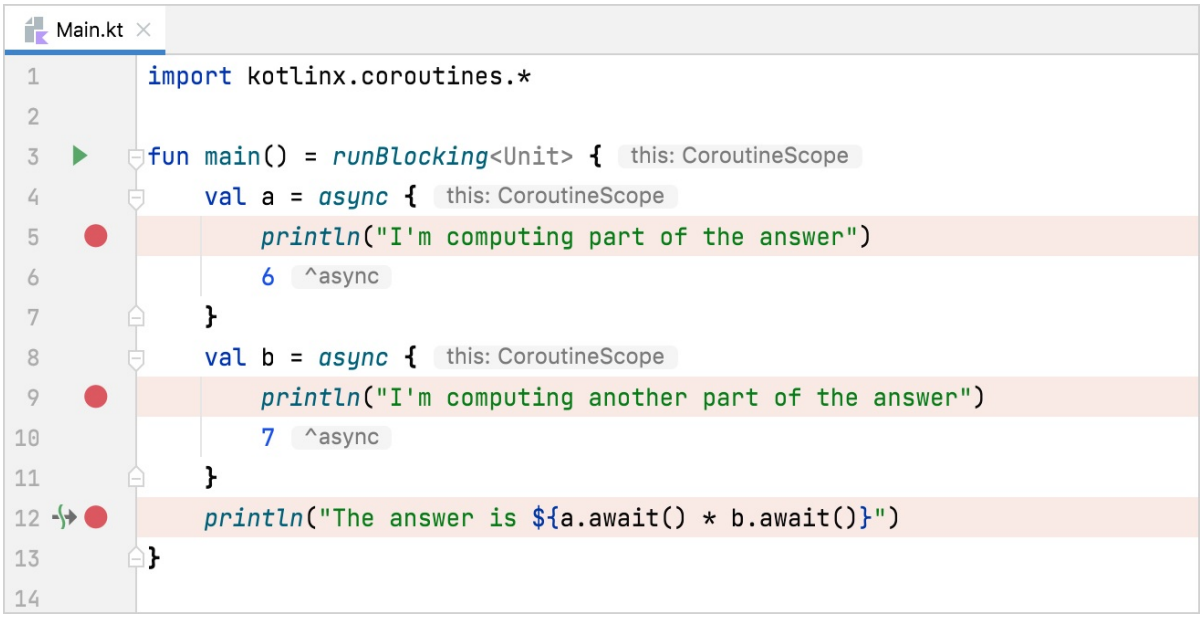
```
import kotlinx.coroutines.*

fun main() = runBlocking<Unit> {
    val a = async {
        println("I'm computing part of the answer")
        6
    }
    val b = async {
        println("I'm computing another part of the answer")
        7
    }
    println("The answer is ${a.await() * b.await()}")
}
```

3. Build the code by clicking **Build Project**.

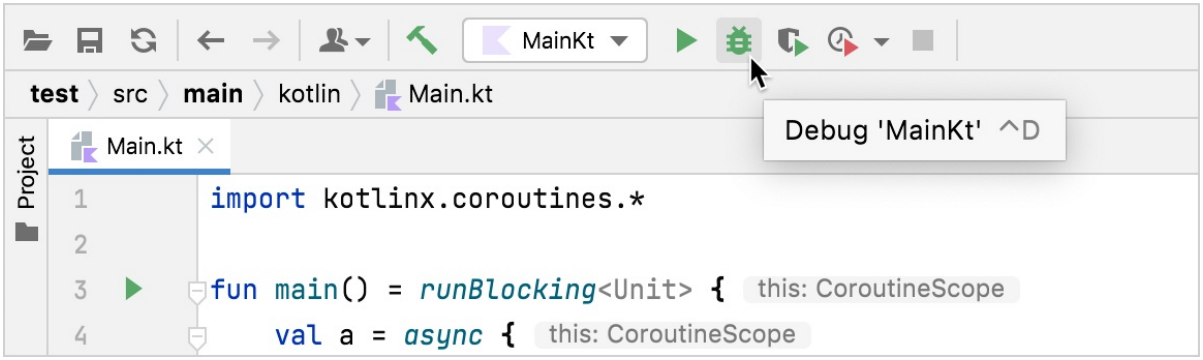


Debug coroutines

1. Set breakpoints at the lines with the `println()` function call:

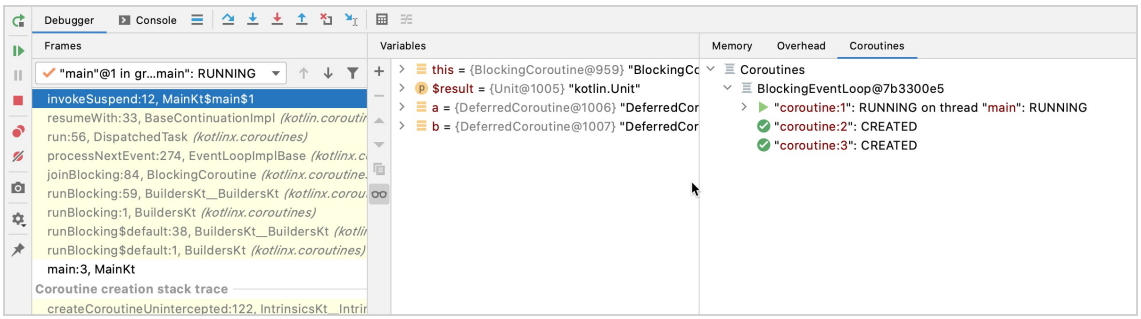


2. Run the code in debug mode by clicking **Debug** next to the run configuration at the top of the screen.

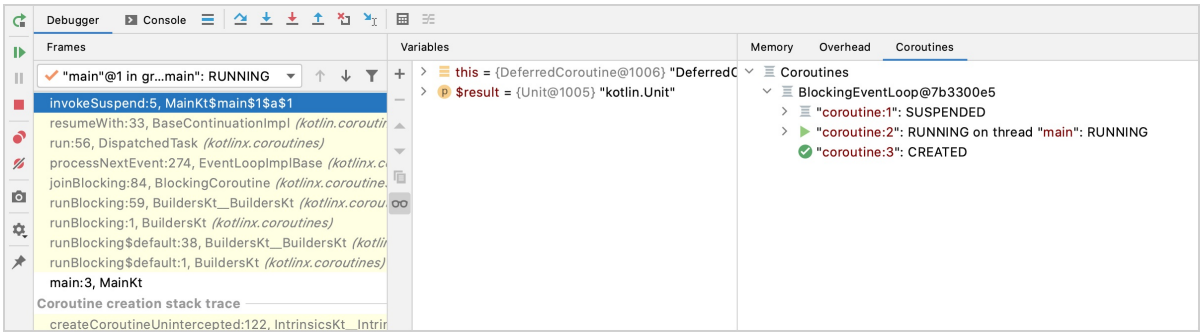


The **Debug** tool window appears:

- The **Frames** tab contains the call stack.
- The **Variables** tab contains variables in the current context.
- The **Coroutines** tab contains information on running or suspended coroutines. It shows that there are three coroutines. The first one has the **RUNNING** status, and the other two have the **CREATED** status.



3. Resume the debugger session by clicking **Resume Program** in the **Debug** tool window:



Now the **Coroutines** tab shows the following:

- The first coroutine has the **SUSPENDED** status – it is waiting for the values so it can multiply them.
- The second coroutine is calculating the `a` value – it has the **RUNNING** status.
- The third coroutine has the **CREATED** status and isn't calculating the value of `b`.

4. Resume the debugger session by clicking **Resume Program** in the **Debug** tool window:



Now the **Coroutines** tab shows the following:

- The first coroutine has the **SUSPENDED** status – it is waiting for the values so it can multiply them.
- The second coroutine has computed its value and disappeared.
- The third coroutine is calculating the value of `b` – it has the **RUNNING** status.

Using IntelliJ IDEA debugger, you can dig deeper into each coroutine to debug your code.

Optimized-out variables

If you use `suspend` functions, in the debugger, you might see the "was optimized out" text next to a variable's name:



This text means that the variable's lifetime was decreased, and the variable doesn't exist anymore. It is difficult to debug code with optimized variables because you don't see their values. You can disable this behavior with the `-Xdebug` compiler option.

> **Never use this flag in production**: `-Xdebug` can cause memory leaks.
>
> ⚠

使用 IntelliJ IDEA 调试 Kotlin Flow——教程

This tutorial demonstrates how to create Kotlin Flow and debug it using IntelliJ IDEA.

The tutorial assumes you have prior knowledge of the coroutines and Kotlin Flow concepts.

Create a Kotlin flow

Create a Kotlin flow with a slow emitter and a slow collector:

1. Open a Kotlin project in IntelliJ IDEA. If you don't have a project, create one.
2. To use the `kotlinx.coroutines` library in a Gradle project, add the following dependency to `build.gradle(.kts)`:

> 【Kotlin】

```
dependencies {
    implementation("org.jetbrains.kotlinx:kotlinx-coroutines-core:1.10.2")
}
```

> 【Groovy】

```
dependencies {
    implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-core:1.10.2'
}
```

For other build systems, see instructions in the `kotlinx.coroutines` README.

1. Open the `Main.kt` file in `src/main/kotlin`.

   The `src` directory contains Kotlin source files and resources. The `Main.kt` file contains sample code that will print `Hello World!`.

2. Create the `simple()` function that returns a flow of three numbers:

   - Use the `delay()` function to imitate CPU-consuming blocking code. It suspends the coroutine for 100 ms without blocking the thread.

- Produce the values in the `for` loop using the `emit()` function.

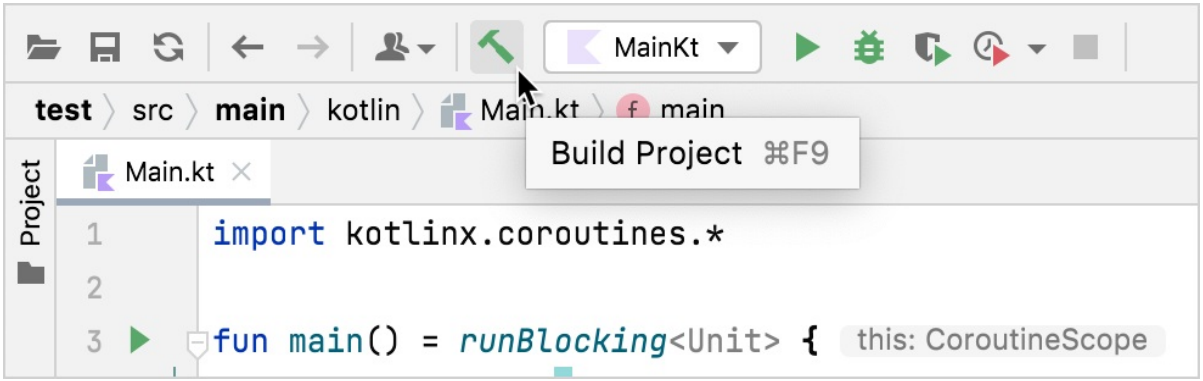
```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*
import kotlin.system.*

fun simple(): Flow<Int> = flow {
    for (i in 1..3) {
        delay(100)
        emit(i)
    }
}
```

3. Change the code in the `main()` function:

   - Use the `runBlocking()` block to wrap a coroutine.
   - Collect the emitted values using the `collect()` function.
   - Use the `delay()` function to imitate CPU-consuming code. It suspends the coroutine for 300 ms without blocking the thread.
   - Print the collected value from the flow using the `println()` function.

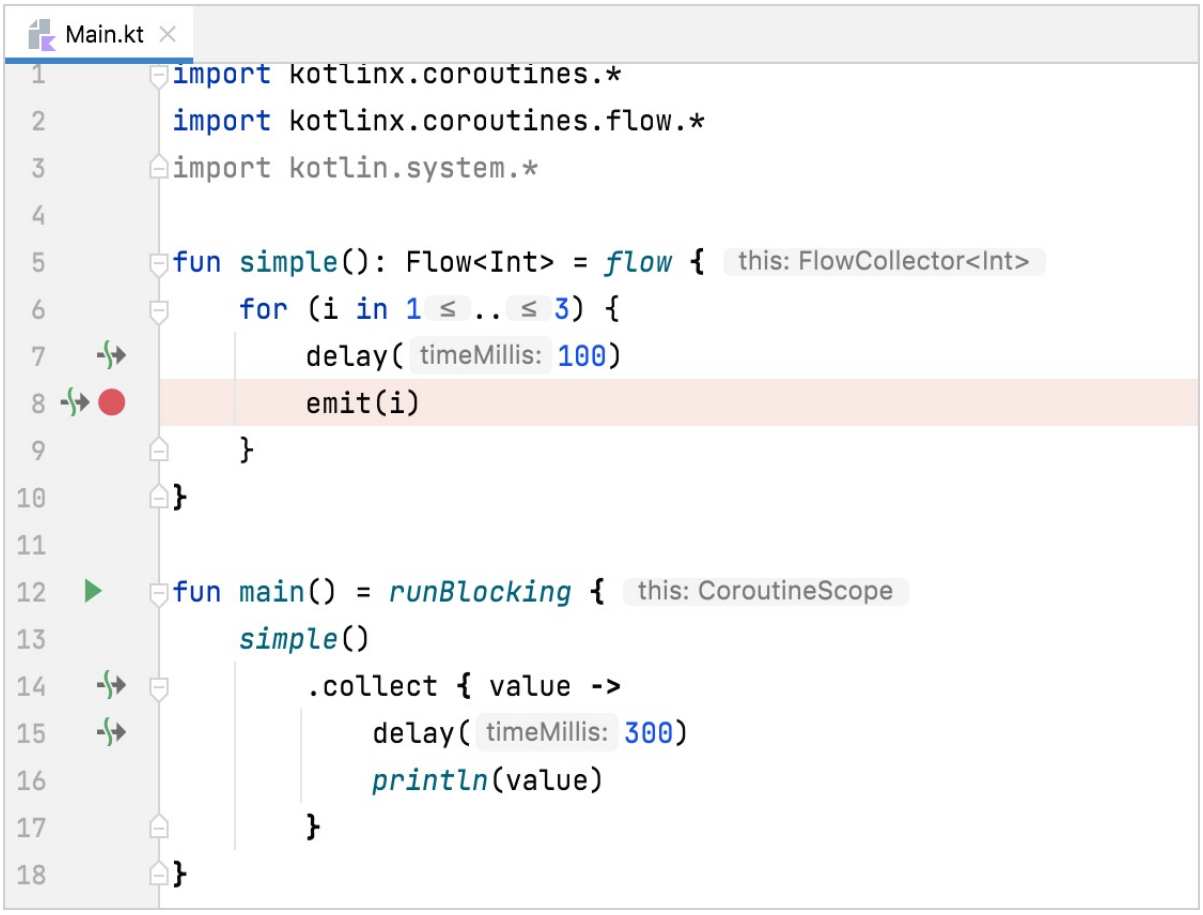
```
fun main() = runBlocking {
    simple()
        .collect { value ->
            delay(300)
            println(value)
        }
}
```

4. Build the code by clicking **Build Project**.

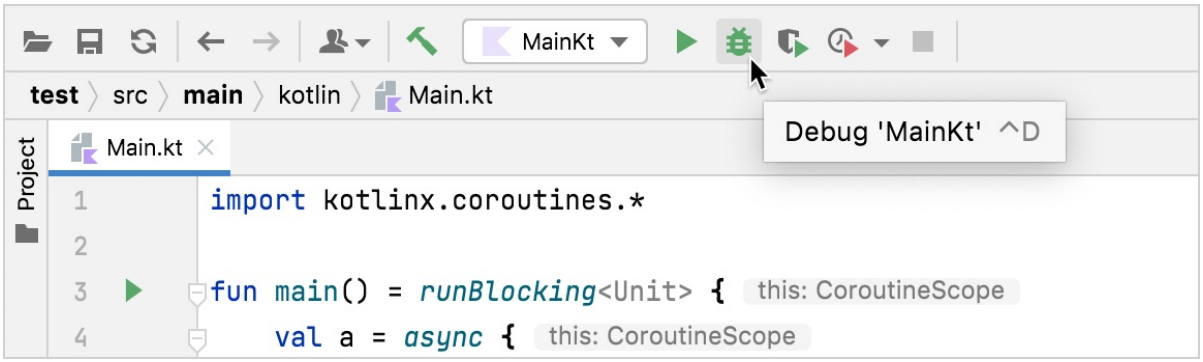


Debug the coroutine

1. Set a breakpoint at the line where the `emit()` function is called:

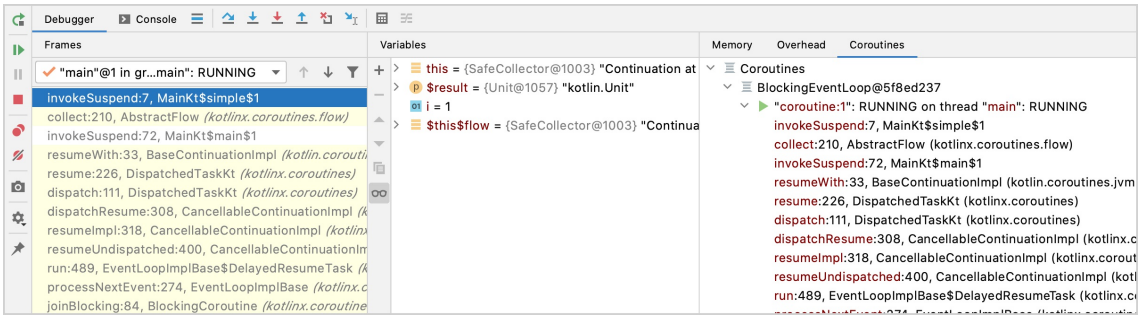


2. Run the code in debug mode by clicking **Debug** next to the run configuration at the top of the screen.

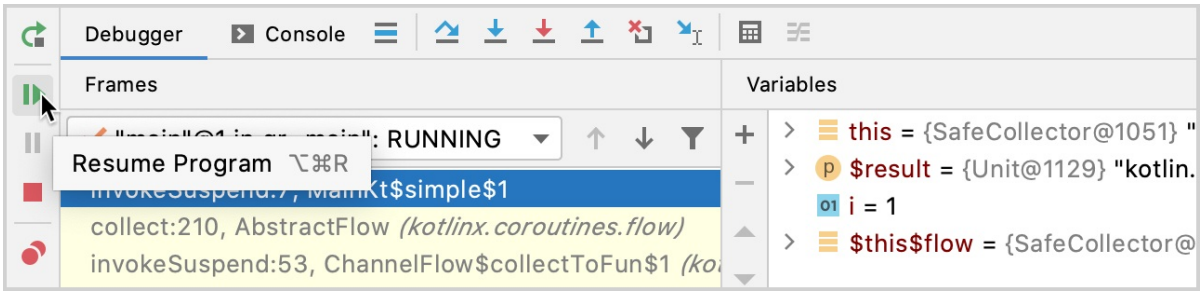


The **Debug** tool window appears:

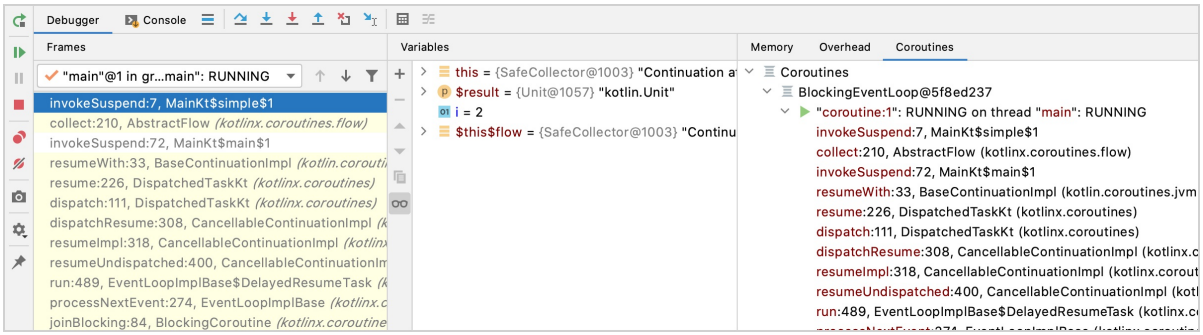
- The **Frames** tab contains the call stack.
- The **Variables** tab contains variables in the current context. It tells us that the flow is emitting the first value.
- The **Coroutines** tab contains information on running or suspended coroutines.



3. Resume the debugger session by clicking **Resume Program** in the **Debug** tool window. The program stops at the same breakpoint.



Now the flow emits the second value.



Optimized-out variables

If you use `suspend` functions, in the debugger, you might see the "was optimized out" text next to a variable's name:



This text means that the variable's lifetime was decreased, and the variable doesn't exist anymore. It is difficult to debug code with optimized variables because you don't see their values. You can disable this behavior with the `-Xdebug` compiler option.

> **Never use this flag in production**: `-Xdebug` can cause memory leaks.
>
> ⚠

Add a concurrently running coroutine

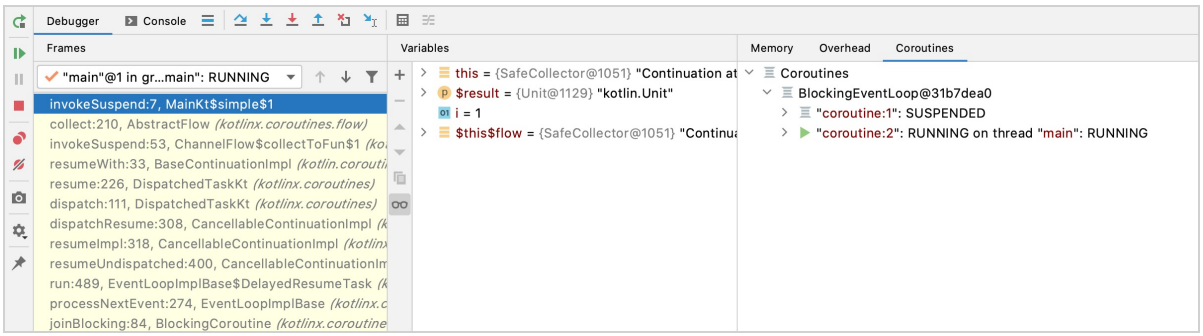
1. Open the `Main.kt` file in `src/main/kotlin`.
2. Enhance the code to run the emitter and collector concurrently:
   - Add a call to the `buffer()` function to run the emitter and collector concurrently.
     `buffer()` stores emitted values and runs the flow collector in a separate coroutine.

```
fun main() = runBlocking<Unit> {
    simple()
        .buffer()
        .collect { value ->
            delay(300)
            println(value)
        }
}
```

3. Build the code by clicking **Build Project**.

Debug a Kotlin flow with two coroutines

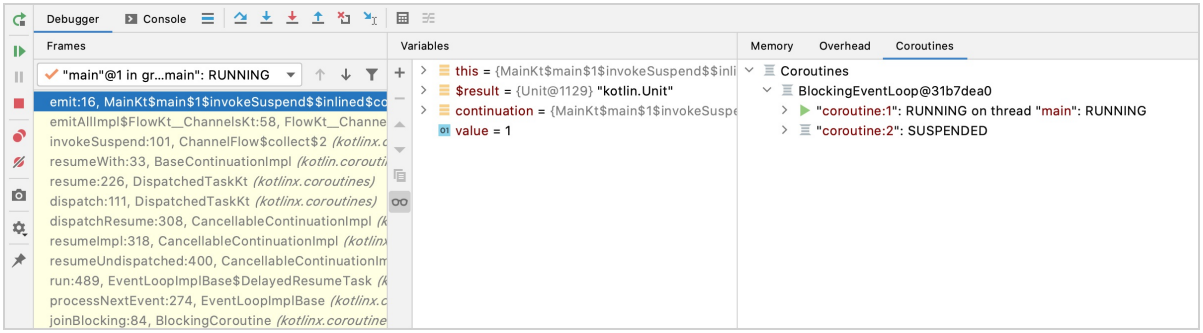
1. Set a new breakpoint at `println(value)`.
2. Run the code in debug mode by clicking **Debug** next to the run configuration at the top of the screen.



The **Debug** tool window appears.

In the **Coroutines** tab, you can see that there are two coroutines running concurrently. The flow collector and emitter run in separate coroutines because of the `buffer()` function. The `buffer()` function buffers emitted values from the flow. The emitter coroutine has the **RUNNING** status, and the collector coroutine has the **SUSPENDED** status.

3. Resume the debugger session by clicking **Resume Program** in the **Debug** tool window.



Now the collector coroutine has the **RUNNING** status, while the emitter coroutine has the **SUSPENDED** status.

You can dig deeper into each coroutine to debug your code.

序列化

Serialization is the process of converting data used by an application to a format that can be transferred over a network or stored in a database or a file. In turn, *deserialization* is the opposite process of reading data from an external source and converting it into a runtime object. Together, they are essential to most applications that exchange data with third parties.

Some data serialization formats, such as JSON and protocol buffers are particularly common. Being language-neutral and platform-neutral, they enable data exchange between systems written in any modern language.

In Kotlin, data serialization tools are available in a separate component, kotlinx.serialization. It consists of several parts: the `org.jetbrains.kotlin.plugin.serialization` Gradle plugin, runtime libraries, and compiler plugins.

Compiler plugins, `kotlinx-serialization-compiler-plugin` and `kotlinx-serialization-compiler-plugin-embeddable`, are published directly to Maven Central. The second plugin is designed for working with the `kotlin-compiler-embeddable` artifact, which is the default option for scripting artifacts. Gradle adds compiler plugins to your projects as compiler arguments.

Libraries

`kotlinx.serialization` provides sets of libraries for all supported platforms – JVM, JavaScript, Native – and for various serialization formats – JSON, CBOR, protocol buffers, and others. You can find the complete list of supported serialization formats below.

All Kotlin serialization libraries belong to the `org.jetbrains.kotlinx:` group. Their names start with `kotlinx-serialization-` and have suffixes that reflect the serialization format. Examples:

- `org.jetbrains.kotlinx:kotlinx-serialization-json` provides JSON serialization for Kotlin projects.
- `org.jetbrains.kotlinx:kotlinx-serialization-cbor` provides CBOR serialization.

Platform-specific artifacts are handled automatically; you don't need to add them manually. Use the same dependencies in JVM, JS, Native, and multiplatform projects.

Note that the `kotlinx.serialization` libraries use their own versioning structure, which doesn't match Kotlin's versioning. Check out the releases on GitHub to find the latest versions.

Formats

`kotlinx.serialization` includes libraries for various serialization formats:

- JSON: `kotlinx-serialization-json`
- Protocol buffers: `kotlinx-serialization-protobuf`
- CBOR: `kotlinx-serialization-cbor`
- Properties: `kotlinx-serialization-properties`
- HOCON: `kotlinx-serialization-hocon` (only on JVM)

Note that all libraries except JSON serialization (`kotlinx-serialization-json`) are Experimental, which means their API can be changed without notice.

There are also community-maintained libraries that support more serialization formats, such as YAML or Apache Avro. For detailed information about available serialization formats, see the `kotlinx.serialization` documentation.

Example: JSON serialization

Let's take a look at how to serialize Kotlin objects into JSON.

Add plugins and dependencies

Before starting, you must configure your build script so that you can use Kotlin serialization tools in your project:

1. Apply the Kotlin serialization Gradle plugin `org.jetbrains.kotlin.plugin.serialization` (or `kotlin("plugin.serialization")` in the Kotlin Gradle DSL).

> 【Kotlin】

```
plugins {
    kotlin("jvm") version "2.1.21"
    kotlin("plugin.serialization") version "2.1.21"
}
```

> 【Groovy】

```
plugins {
    id 'org.jetbrains.kotlin.jvm' version '2.1.21'
    id 'org.jetbrains.kotlin.plugin.serialization' version '2.1.21'
}
```

1. Add the JSON serialization library dependency: `org.jetbrains.kotlinx:kotlinx-serialization-json:1.8.1`

> 【Kotlin】

```
dependencies {
    implementation("org.jetbrains.kotlinx:kotlinx-serialization-json:1.8.1")
}
```

> 【Groovy】

```
dependencies {
    implementation 'org.jetbrains.kotlinx:kotlinx-serialization-json:1.8.1'
}
```

Now you're ready to use the serialization API in your code. The API is located in the `kotlinx.serialization` package and its format-specific subpackages, such as `kotlinx.serialization.json`.

Serialize and deserialize JSON

1. Make a class serializable by annotating it with `@Serializable`.

```
import kotlinx.serialization.Serializable

@Serializable
data class Data(val a: Int, val b: String)
```

1. Serialize an instance of this class by calling `Json.encodeToString()`.

```
import kotlinx.serialization.Serializable
import kotlinx.serialization.json.Json
import kotlinx.serialization.encodeToString

@Serializable
data class Data(val a: Int, val b: String)

fun main() {
```

```
    val json = Json.encodeToString(Data(42, "str"))
}
```

As a result, you get a string containing the state of this object in the JSON format: `{"a": 42, "b": "str"}`

> You can also serialize object collections, such as lists, in a single call:
>
> ```
> val dataList = listOf(Data(42, "str"), Data(12, "test"))
> val jsonList = Json.encodeToString(dataList)
> ```
>
> 

1. Use the `decodeFromString()` function to deserialize an object from JSON:

```
import kotlinx.serialization.Serializable
import kotlinx.serialization.json.Json
import kotlinx.serialization.decodeFromString

@Serializable
data class Data(val a: Int, val b: String)

fun main() {
    val obj = Json.decodeFromString<Data>("""{"a":42, "b": "str"}""")
}
```

That's it! You have successfully serialized objects into JSON strings and deserialized them back into objects.

下一步做什么

For more information about serialization in Kotlin, see the Kotlin Serialization Guide.

You can explore different aspects of Kotlin serialization in the following resources:

- Learn more about Kotlin serialization and its core concepts
- Explore the built-in serializable classes of Kotlin
- Look at serializers in more detail and learn how to create custom serializers
- Discover how polymorphic serialization is handled in Kotlin
- Look into the various JSON features handling Kotlin serialization
- Learn more about the experimental serialization formats supported by Kotlin

Kotlin Metadata JVM 库

The `kotlin-metadata-jvm` library provides tools to read, modify, and generate metadata from Kotlin classes compiled for the JVM. This metadata, stored in the `@Metadata` annotation within `.class` files, is used by libraries and tools such as `kotlin-reflect` to inspect Kotlin-specific constructs such as properties, functions, and classes at runtime.

> The `kotlin-reflect` library relies on metadata to retrieve Kotlin-specific class details at runtime. Any inconsistencies between the metadata and the actual `.class` file may lead to incorrect behavior when using reflection.
>
> 

You can also use the Kotlin Metadata JVM library to inspect various declaration attributes such as visibility or modality, or to generate and embed metadata into `.class` files.

Add the library to your project

To include the Kotlin Metadata JVM library in your project, add the corresponding dependency configuration based on your build tool.

> The Kotlin Metadata JVM library follows the same versioning as the Kotlin compiler and standard library. Ensure that the version you use matches your project's Kotlin version.
>
> 

Gradle

Add the following dependency to your `build.gradle(.kts)` file:

> 【Kotlin】

```
// build.gradle.kts
repositories {
    mavenCentral()
}

dependencies {
    implementation("org.jetbrains.kotlin:kotlin-metadata-jvm:2.1.21")
}
```

【Groovy】

```
// build.gradle
repositories {
    mavenCentral()
}

dependencies {
    implementation 'org.jetbrains.kotlin:kotlin-metadata-jvm:2.1.21'
}
```

Maven

Add the following dependency to your `pom.xml` file.

```
<project>
    <dependencies>
        <dependency>
            <groupId>org.jetbrains.kotlin</groupId>
            <artifactId>kotlin-metadata-jvm</artifactId>
            <version>2.1.21</version>
        </dependency>
    </dependencies>
    ...
</project>
```

Read and parse metadata

The `kotlin-metadata-jvm` library extracts structured information from compiled Kotlin `.class` files, such as class names, visibility, and signatures. You can use it in projects that need to analyze compiled Kotlin declarations. For example, the Binary Compatibility Validator (BCV) relies on `kotlin-metadata-jvm` to print public API declarations.

You can start exploring Kotlin class metadata by retrieving the `@Metadata` annotation from a compiled class using reflection:

```
fun main() {
    // Specifies the fully qualified name of the class
    val clazz = Class.forName("org.example.SampleClass")

    // Retrieves the @Metadata annotation
    val metadata = clazz.getAnnotation(Metadata::class.java)
```

```
    // Checks if the metadata is present
    if (metadata != null) {
        println("This is a Kotlin class with metadata.")
    } else {
        println("This is not a Kotlin class.")
    }
}
```

After retrieving the `@Metadata` annotation, use either the `readLenient()` or the `readStrict()` function from the `KotlinClassMetadata` API to parse it. These functions extract detailed information about classes or files, while addressing different compatibility requirements:

- `readLenient()` : Use this function to read metadata, including metadata generated by newer Kotlin compiler versions. This function doesn't support modifying or writing metadata.
- `readStrict()` : Use this function when you need to modify and write metadata. The `readStrict()` function only works with metadata generated by Kotlin compiler versions fully supported by your project.

  > The `readStrict()` function supports metadata formats up to one version beyond `JvmMetadataVersion.LATEST_STABLE_SUPPORTED` , which corresponds to the latest Kotlin version used in the project. For example, if your project depends on `kotlin-metadata-jvm:2.1.0` , `readStrict()` can process metadata up to Kotlin `2.2.x` ; otherwise, it throws an error to prevent mishandling unknown formats.
  >
  > For more information, see the Kotlin Metadata GitHub repository.
  >
  > 

When parsing metadata, the `KotlinClassMetadata` instance provides structured information about class or file-level declarations. For classes, use the `kmClass` property to analyze detailed class-level metadata, such as the class name, functions, properties, and attributes like visibility. For file-level declarations, the metadata is represented by the `kmPackage` property, which includes top-level functions and properties from file facades generated by the Kotlin compiler.

The following code example demonstrates how to use `readLenient()` to parse metadata, analyze class-level details with `kmClass` , and retrieve file-level declarations with `kmPackage` :

```
// Imports the necessary libraries
import kotlin.metadata.jvm.*
import kotlin.metadata.*

fun main() {
    // Specifies the fully qualified class name
    val className = "org.example.SampleClass"

    try {
```

```
        // Retrieves the class object for the specified name
        val clazz = Class.forName(className)

        // Retrieves the @Metadata annotation
        val metadataAnnotation = clazz.getAnnotation(Metadata::class.java)
        if (metadataAnnotation != null) {
            println("Kotlin Metadata found for class: $className")

            // Parses metadata using the readLenient() function
            val metadata = KotlinClassMetadata.readLenient(metadataAnnotation)
            when (metadata) {
                is KotlinClassMetadata.Class -> {
                    val kmClass = metadata.kmClass
                    println("Class name: ${kmClass.name}")

                    // Iterates over functions and checks visibility
                    kmClass.functions.forEach { function ->
                        val visibility = function.visibility
                        println("Function: ${function.name}, Visibility: $visibility")
                    }
                }
                is KotlinClassMetadata.FileFacade -> {
                    val kmPackage = metadata.kmPackage

                    // Iterates over functions and checks visibility
                    kmPackage.functions.forEach { function ->
                        val visibility = function.visibility
                        println("Function: ${function.name}, Visibility: $visibility")
                    }
                }
                else -> {
                    println("Unsupported metadata type: $metadata")
                }
            }
        } else {
            println("No Kotlin Metadata found for class: $className")
        }
    } catch (e: ClassNotFoundException) {
        println("Class not found: $className")
    } catch (e: Exception) {
        println("Error processing metadata: ${e.message}")
        e.printStackTrace()
    }
}
```

Extract metadata from bytecode

While you can retrieve metadata using reflection, another approach is to extract it from bytecode using a bytecode manipulation framework such as ASM.

You can do this by following these steps:

1. Read the bytecode of a `.class` file using the ASM library's `ClassReader` class. This class processes the compiled file and populates a `ClassNode` object, which represents the class structure.
2. Extract the `@Metadata` from the `ClassNode` object. The example below uses a custom extension function `findAnnotation()` for this.
3. Parse the extracted metadata using the `KotlinClassMetadata.readLenient()` function.
4. Inspect the parsed metadata with the `kmClass` and `kmPackage` properties.

Here's an example:

```
// Imports the necessary libraries
import kotlin.metadata.jvm.*
import kotlin.metadata.*
import org.objectweb.asm.*
import org.objectweb.asm.tree.*
import java.io.File

// Checks if an annotation refers to a specific name
fun AnnotationNode.refersToName(name: String) =
    desc.startsWith('L') && desc.endsWith(';') && desc.regionMatches(1, name, 0, name.
length)

// Retrieves annotation values by key
private fun List<Any>.annotationValue(key: String): Any? {
    for (index in (0 until size / 2)) {
        if (this[index * 2] == key) {
            return this[index * 2 + 1]
        }
    }
    return null
}

// Defines a custom extension function to locate an annotation by its name in a ClassN
ode
fun ClassNode.findAnnotation(annotationName: String, includeInvisible: Boolean = false)
: AnnotationNode? {
    val visible = visibleAnnotations?.firstOrNull { it.refersToName(annotationName) }
    if (!includeInvisible) return visible
    return visible ?: invisibleAnnotations?.firstOrNull { it.refersToName(annotationNa
me) }
}

// Operator to simplify retrieving annotation values
operator fun AnnotationNode.get(key: String): Any? = values.annotationValue(key)

// Extracts Kotlin metadata from a class node
fun ClassNode.readMetadataLenient(): KotlinClassMetadata? {
    val metadataAnnotation = findAnnotation("kotlin/Metadata", false) ?: return null
    @Suppress("UNCHECKED_CAST")
    val metadata = Metadata(
```

```
            kind = metadataAnnotation["k"] as Int?,
            metadataVersion = (metadataAnnotation["mv"] as List<Int>?)?.toIntArray(),
            data1 = (metadataAnnotation["d1"] as List<String>?)?.toTypedArray(),
            data2 = (metadataAnnotation["d2"] as List<String>?)?.toTypedArray(),
            extraString = metadataAnnotation["xs"] as String?,
            packageName = metadataAnnotation["pn"] as String?,
            extraInt = metadataAnnotation["xi"] as Int?
        )
        return KotlinClassMetadata.readLenient(metadata)
    }

    // Converts a file to a ClassNode for bytecode inspection
    fun File.toClassNode(): ClassNode {
        val node = ClassNode()
        this.inputStream().use { ClassReader(it).accept(node, ClassReader.SKIP_CODE) }
        return node
    }

    fun main() {
        val classFilePath = "build/classes/kotlin/main/org/example/SampleClass.class"
        val classFile = File(classFilePath)

        // Reads the bytecode and processes it into a ClassNode object
        val classNode = classFile.toClassNode()

        // Locates the @Metadata annotation and reads it leniently
        val metadata = classNode.readMetadataLenient()
        if (metadata != null && metadata is KotlinClassMetadata.Class) {
            // Inspects the parsed metadata
            val kmClass = metadata.kmClass

            // Prints class details
            println("Class name: ${kmClass.name}")
            println("Functions:")
            kmClass.functions.forEach { function ->
                println("- ${function.name}, Visibility: ${function.visibility}")
            }
        }
    }
```

Modify metadata

When using tools like ProGuard to shrink and optimize bytecode, some declarations may be removed from `.class` files. ProGuard automatically updates metadata to keep it consistent with the modified bytecode.

However, if you're developing a custom tool that modifies Kotlin bytecode in a similar way, you need to ensure that metadata is adjusted accordingly. With the `kotlin-metadata-jvm` library, you can update declarations, adjust attributes, and remove specific elements.

For example, if you use a JVM tool that deletes private methods from Java class files, you must also delete private functions from Kotlin metadata to maintain consistency:

1. Parse the metadata by using the `readStrict()` function to load the `@Metadata` annotation into a structured `KotlinClassMetadata` object.
2. Apply modifications by adjusting the metadata, such as filtering functions or altering attributes, directly within `kmClass` or other metadata structures.
3. Use the `write()` function to encode the modified metadata into a new `@Metadata` annotation.

Here's an example where private functions are removed from a class's metadata:

```
// Imports the necessary libraries
import kotlin.metadata.jvm.*
import kotlin.metadata.*

fun main() {
    // Specifies the fully qualified class name
    val className = "org.example.SampleClass"

    try {
        // Retrieves the class object for the specified name
        val clazz = Class.forName(className)

        // Retrieves the @Metadata annotation
        val metadataAnnotation = clazz.getAnnotation(Metadata::class.java)
        if (metadataAnnotation != null) {
            println("Kotlin Metadata found for class: $className")

            // Parses metadata using the readStrict() function
            val metadata = KotlinClassMetadata.readStrict(metadataAnnotation)
            if (metadata is KotlinClassMetadata.Class) {
                val kmClass = metadata.kmClass

                // Removes private functions from the class metadata
                kmClass.functions.removeIf { it.visibility == Visibility.PRIVATE }
                println("Removed private functions. Remaining functions: ${kmClass.fun
ctions.map { it.name }}")

                // Serializes the modified metadata back
                val newMetadata = metadata.write()
                // After modifying the metadata, you need to write it into the class f
ile
                // To do so, you can use a bytecode manipulation framework such as ASM

                println("Modified metadata: ${newMetadata}")
            } else {
```

```
                println("The metadata is not a class.")
            }
        } else {
            println("No Kotlin Metadata found for class: $className")
        }
    } catch (e: ClassNotFoundException) {
        println("Class not found: $className")
    } catch (e: Exception) {
        println("Error processing metadata: ${e.message}")
        e.printStackTrace()
    }
}
```

> Instead of separately calling `readStrict()` and `write()`, you can use the `transform()` function. This function parses metadata, applies transformations through a lambda, and writes the modified metadata automatically.
>
> 

Create metadata from scratch

To create metadata for a Kotlin class file from scratch using the Kotlin Metadata JVM library:

1. Create an instance of `KmClass`, `KmPackage`, or `KmLambda`, depending on the type of metadata you want to generate.
2. Add attributes to the instance, such as the class name, visibility, constructors, and function signatures.

   > You can use the `apply()` scope function to reduce boilerplate code while setting properties.
   >
   > 

3. Use the instance to create a `KotlinClassMetadata` object, which can generate a `@Metadata` annotation.
4. Specify the metadata version, such as `JvmMetadataVersion.LATEST_STABLE_SUPPORTED`, and set flags (`0` for no flags, or copy flags from existing files if necessary).
5. Use the `ClassWriter` class from ASM to embed metadata fields, such as `kind`, `data1` and `data2` into a `.class` file.

The following example demonstrates how to create metadata for a simple Kotlin class:

```
// Imports the necessary libraries
import kotlin.metadata.*
import kotlin.metadata.jvm.*
```

```
import org.objectweb.asm.*

fun main() {
    // Creates a KmClass instance
    val klass = KmClass().apply {
        name = "Hello"
        visibility = Visibility.PUBLIC
        constructors += KmConstructor().apply {
            visibility = Visibility.PUBLIC
            signature = JvmMethodSignature("<init>", "()V")
        }
        functions += KmFunction("hello").apply {
            visibility = Visibility.PUBLIC
            returnType = KmType().apply {
                classifier = KmClassifier.Class("kotlin/String")
            }
            signature = JvmMethodSignature("hello", "()Ljava/lang/String;")
        }
    }

    // Serializes a KotlinClassMetadata.Class instance, including the version and flag
s, into a @kotlin.Metadata annotation
    val annotationData = KotlinClassMetadata.Class(
        klass, JvmMetadataVersion.LATEST_STABLE_SUPPORTED, 0
    ).write()

    // Generates a .class file with ASM
    val classBytes = ClassWriter(0).apply {
        visit(Opcodes.V1_6, Opcodes.ACC_PUBLIC, "Hello", null, "java/lang/Object", null
)
        // Writes @kotlin.Metadata instance to the .class file
        visitAnnotation("Lkotlin/Metadata;", true).apply {
            visit("mv", annotationData.metadataVersion)
            visit("k", annotationData.kind)
            visitArray("d1").apply {
                annotationData.data1.forEach { visit(null, it) }
                visitEnd()
            }
            visitArray("d2").apply {
                annotationData.data2.forEach { visit(null, it) }
                visitEnd()
            }
            visitEnd()
        }
        visitEnd()
    }.toByteArray()

    // Writes the generated class file to disk
    java.io.File("Hello.class").writeBytes(classBytes)

    println("Metadata and .class file created successfully.")
}
```

> For a more detailed example, see the Kotlin Metadata JVM GitHub repository.
>
> 

下一步做什么

- See the API reference for the Kotlin Metadata JVM library.
- Check out the Kotlin Metadata JVM GitHub repository.
- Learn about module metadata and working with `.kotlin_module` files.

Lincheck（kotlinx.lincheck）

- Lincheck 指南
- 用 Lincheck 编写第一个测试
- 测试策略
- 操作参数
- 数据结构约束
- 进度保证
- 时序规范

Lincheck 指南

Lincheck is a practical and user-friendly framework for testing concurrent algorithms on the JVM. It provides a simple and declarative way to write concurrent tests.

With the Lincheck framework, instead of describing how to perform tests, you can specify *what to test* by declaring all the operations to examine and the required correctness property. As a result, a typical concurrent Lincheck test contains only about 15 lines.

When given a list of operations, Lincheck automatically:

- Generates a set of random concurrent scenarios.
- Examines them using either stress-testing or bounded model checking.
- Verifies that the results of each invocation satisfy the required correctness property (linearizability is the default one).

Add Lincheck to your project

To enable the Lincheck support, include the corresponding repository and dependency to the Gradle configuration. In your `build.gradle(.kts)` file, add the following:

> 【Kotlin】

```
repositories {
    mavenCentral()
}

dependencies {
    testImplementation("org.jetbrains.kotlinx:lincheck:2.39")
}
```

> 【Groovy】

```
repositories {
    mavenCentral()
}

dependencies {
    testImplementation "org.jetbrains.kotlinx:lincheck:2.39"
```

```
}
```

Explore Lincheck

This guide will help you get in touch with the framework and try the most useful features with examples. Learn the Lincheck features step-by-step:

1. Write your first test with Lincheck
2. Choose your testing strategy
3. Configure operation arguments
4. Consider popular algorithm constraints
5. Check the algorithm for non-blocking progress guarantees
6. Define sequential specification of the algorithm

Additional references

- "How we test concurrent algorithms in Kotlin Coroutines" by Nikita Koval: Video. KotlinConf 2023
- "Lincheck: Testing concurrency on the JVM" workshop by Maria Sokolova: Part 1, Part 2. Hydra 2021

用 Lincheck 编写第一个测试

This tutorial demonstrates how to write your first Lincheck test, set up the Lincheck framework, and use its basic API. You will create a new IntelliJ IDEA project with an incorrect concurrent counter implementation and write a test for it, finding and analyzing the bug afterward.

Create a project

Open an existing Kotlin project in IntelliJ IDEA or create a new one. When creating a project, use the Gradle build system.

Add required dependencies

1. Open the `build.gradle(.kts)` file and make sure that `mavenCentral()` is added to the repository list.
2. Add the following dependencies to the Gradle configuration:

> 【Kotlin】

```
repositories {
    mavenCentral()
}

dependencies {
    // Lincheck dependency
    testImplementation("org.jetbrains.kotlinx:lincheck:2.39")
    // This dependency allows you to work with kotlin.test and JUnit:
    testImplementation("junit:junit:4.13")
}
```

> 【Groovy】

```
repositories {
    mavenCentral()
}

dependencies {
```

```
        // Lincheck dependency
        testImplementation "org.jetbrains.kotlinx:lincheck:2.39"
        // This dependency allows you to work with kotlin.test and JUnit:
        testImplementation "junit:junit:4.13"
    }
```

Write a concurrent counter and run the test

1. In the `src/test/kotlin` directory, create a `BasicCounterTest.kt` file and add the following code with a buggy concurrent counter and a Lincheck test for it:

```
import org.jetbrains.kotlinx.lincheck.annotations.*
import org.jetbrains.kotlinx.lincheck.*
import org.jetbrains.kotlinx.lincheck.strategy.stress.*
import org.junit.*

class Counter {
    @Volatile
    private var value = 0

    fun inc(): Int = ++value
    fun get() = value
}

class BasicCounterTest {
    private val c = Counter() // Initial state

    // Operations on the Counter
    @Operation
    fun inc() = c.inc()

    @Operation
    fun get() = c.get()

    @Test // JUnit
    fun stressTest() = StressOptions().check(this::class) // The magic button
}
```

   This Lincheck test automatically:

   - Generates several random concurrent scenarios with the specified `inc()` and `get()` operations.
   - Performs a lot of invocations for each of the generated scenarios.
   - Verifies that each invocation result is correct.
2. Run the test above, and you will see the following error:

```
= Invalid execution results =
| ------------------- |
| Thread 1 | Thread 2 |
| ------------------- |
| inc(): 1 | inc(): 1 |
| ------------------- |
```

Here, Lincheck found an execution that violates the counter atomicity – two concurrent increments ended with the same result `1`. It means that one increment has been lost, and the behavior of the counter is incorrect.

Trace the invalid execution

Besides showing invalid execution results, Lincheck can also provide an interleaving that leads to the error. This feature is accessible with the model checking testing strategy, which examines numerous executions with a bounded number of context switches.

1. To switch the testing strategy, replace the `options` type from `StressOptions()` to `ModelCheckingOptions()`. The updated `BasicCounterTest` class will look like this:

```
import org.jetbrains.kotlinx.lincheck.annotations.*
import org.jetbrains.kotlinx.lincheck.check
import org.jetbrains.kotlinx.lincheck.strategy.managed.modelchecking.*
import org.junit.*

class Counter {
    @Volatile
    private var value = 0

    fun inc(): Int = ++value
    fun get() = value
}

class BasicCounterTest {
    private val c = Counter()

    @Operation
    fun inc() = c.inc()

    @Operation
    fun get() = c.get()

    @Test
    fun modelCheckingTest() = ModelCheckingOptions().check(this::class)
}
```

2. Run the test again. You will get the execution trace that leads to incorrect results:

```
= Invalid execution results =
| ------------------- |
| Thread 1 | Thread 2 |
| ------------------- |
| inc(): 1 | inc(): 1 |
| ------------------- |

The following interleaving leads to the error:
| ------------------------------------------------------------------------ |
| Thread 1 |                          Thread  2                             |
| ------------------------------------------------------------------------ |
|          | inc()                                                         |
|          |   inc(): 1 at BasicCounterTest.inc(BasicCounterTest.kt:18)    |
|          |     value.READ: 0 at Counter.inc(BasicCounterTest.kt:10)      |
|          |     switch                                                    |
| inc(): 1 |                                                               |
|          |     value.WRITE(1) at Counter.inc(BasicCounterTest.kt:10)     |
|          |     value.READ: 1 at Counter.inc(BasicCounterTest.kt:10)      |
|          |   result: 1                                                   |
| ------------------------------------------------------------------------ |
```

   According to the trace, the following events have occurred:

   - **T2**: The second thread starts the `inc()` operation, reading the current counter value (`value.READ: 0`) and pausing.
   - **T1**: The first thread executes `inc()`, which returns `1`, and finishes.
   - **T2**: The second thread resumes and increments the previously obtained counter value, incorrectly updating the counter to `1`.

> Get the full code.
>
> 

Test the Java standard library

Let's now find a bug in the standard Java's `ConcurrentLinkedDeque` class. The Lincheck test below finds a race between removing and adding an element to the head of the deque:

```
import org.jetbrains.kotlinx.lincheck.*
import org.jetbrains.kotlinx.lincheck.annotations.*
import org.jetbrains.kotlinx.lincheck.strategy.managed.modelchecking.*
import org.junit.*
import java.util.concurrent.*

class ConcurrentDequeTest {
```

```
    private val deque = ConcurrentLinkedDeque<Int>()

    @Operation
    fun addFirst(e: Int) = deque.addFirst(e)

    @Operation
    fun addLast(e: Int) = deque.addLast(e)

    @Operation
    fun pollFirst() = deque.pollFirst()

    @Operation
    fun pollLast() = deque.pollLast()

    @Operation
    fun peekFirst() = deque.peekFirst()

    @Operation
    fun peekLast() = deque.peekLast()

    @Test
    fun modelCheckingTest() = ModelCheckingOptions().check(this::class)
}
```

Run `modelCheckingTest()`. The test will fail with the following output:

```
= Invalid execution results =
| ---------------------------------------- |
|      Thread 1     |       Thread 2       |
| ---------------------------------------- |
| addLast(22): void |                      |
| ---------------------------------------- |
| pollFirst(): 22   | addFirst(8): void    |
|                   | peekLast(): 22 [-,1] |
| ---------------------------------------- |

---
All operations above the horizontal line | ----- | happen before those below the line
---
Values in "[..]" brackets indicate the number of completed operations
in each of the parallel threads seen at the beginning of the current operation
---

The following interleaving leads to the error:
| ---------------------------------------------------------------------------------
--------------------------------------------- |
|                                                Thread 1
                       |       Thread 2       |
| ---------------------------------------------------------------------------------
--------------------------------------------- |
| pollFirst()
```

```
                        |                        |
|   pollFirst(): 22 at ConcurrentDequeTest.pollFirst(ConcurrentDequeTest.kt:17)
                        |                        |
|     first(): Node@1 at ConcurrentLinkedDeque.pollFirst(ConcurrentLinkedDeque.java:91
5)                      |                        |
|     item.READ: null at ConcurrentLinkedDeque.pollFirst(ConcurrentLinkedDeque.java:91
7)                      |                        |
|     next.READ: Node@2 at ConcurrentLinkedDeque.pollFirst(ConcurrentLinkedDeque.java:
925)                    |                        |
|     item.READ: 22 at ConcurrentLinkedDeque.pollFirst(ConcurrentLinkedDeque.java:917)
                        |                        |
|     prev.READ: null at ConcurrentLinkedDeque.pollFirst(ConcurrentLinkedDeque.java:91
9)                      |                        |
|     switch
                        |                        |
|
                        | addFirst(8): void      |
|
                        | peekLast(): 22         |
|     compareAndSet(Node@2,22,null): true at ConcurrentLinkedDeque.pollFirst(Concurren
tLinkedDeque.java:920) |                        |
|     unlink(Node@2) at ConcurrentLinkedDeque.pollFirst(ConcurrentLinkedDeque.java:921
)                       |                        |
|   result: 22
                        |                        |
| ------------------------------------------------------------------------------------
------------------------------------------- |
```

> Get the full code.
>
> 

Next step

Choose your testing strategy and configure test execution.

See also

- How to generate operation arguments
- Popular algorithm constraints
- Checking for non-blocking progress guarantees
- Define sequential specification of the algorithm

压力测试和模型检测

Lincheck offers two testing strategies: stress testing and model checking. Learn what happens under the hood of both approaches using the `Counter` we coded in the `BasicCounterTest.kt` file in the previous step:

```
class Counter {
    @Volatile
    private var value = 0

    fun inc(): Int = ++value
    fun get() = value
}
```

Stress testing

Write a stress test

Create a concurrent stress test for the `Counter`, following these steps:

1. Create the `CounterTest` class.
2. In this class, add the field `c` of the `Counter` type, creating an instance in the constructor.
3. List the counter operations and mark them with the `@Operation` annotation, delegating their implementations to `c`.
4. Specify the stress testing strategy using `StressOptions()`.
5. Invoke the `StressOptions.check()` function to run the test.

The resulting code will look like this:

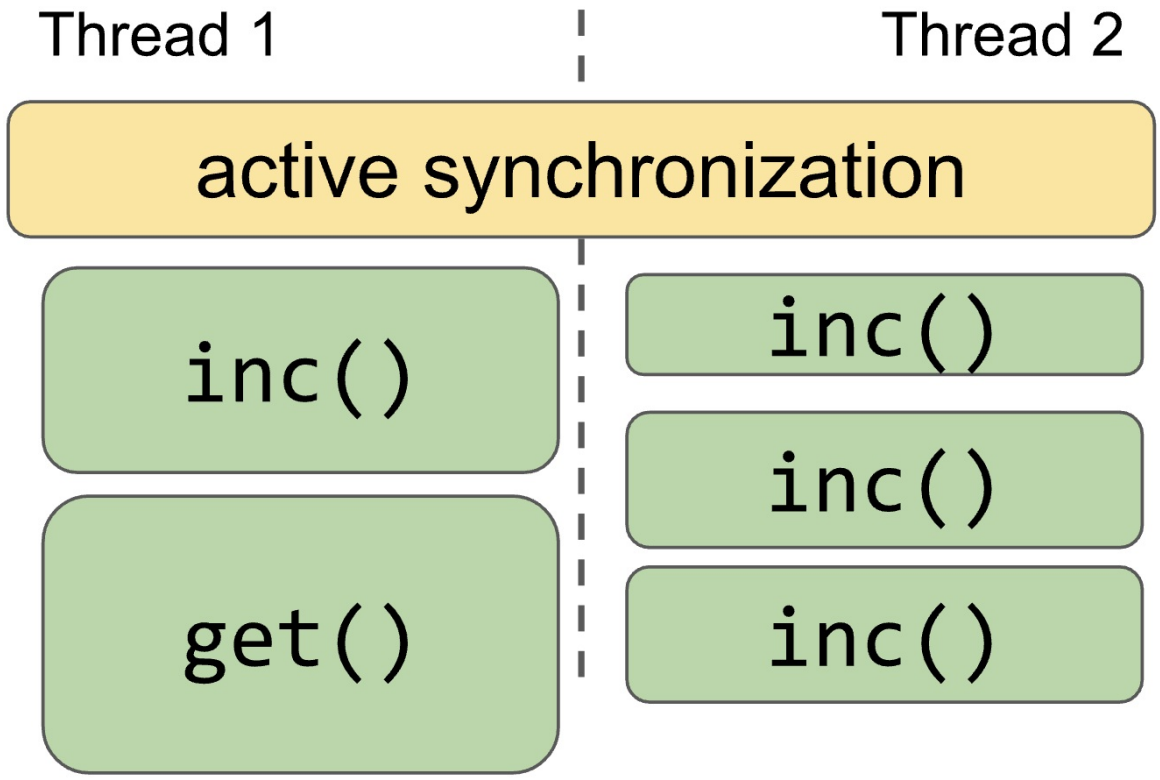
```
import org.jetbrains.kotlinx.lincheck.annotations.*
import org.jetbrains.kotlinx.lincheck.check
import org.jetbrains.kotlinx.lincheck.strategy.stress.*
import org.junit.*

class CounterTest {
    private val c = Counter() // Initial state

    // Operations on the Counter
    @Operation
    fun inc() = c.inc()

    @Operation
    fun get() = c.get()
```

```
    @Test // Run the test
    fun stressTest() = StressOptions().check(this::class)
}
```

How stress testing works

At first, Lincheck generates a set of concurrent scenarios using the operations marked with `@Operation`. Then it launches native threads, synchronizing them at the beginning to guarantee that operations start simultaneously. Finally, Lincheck executes each scenario on these native threads multiple times, expecting to hit an interleaving that produces incorrect results.

The figure below shows a high-level scheme of how Lincheck may execute generated scenarios:



Model checking

The main concern regarding stress testing is that you may spend hours trying to understand how to reproduce the found bug. To help you with that, Lincheck supports bounded model checking, which automatically provides an interleaving for reproducing bugs.

A model checking test is constructed the same way as the stress test. Just replace the `StressOptions()` that specify the testing strategy with `ModelCheckingOptions()`.

Write a model checking test

To change the stress testing strategy to model checking, replace `StressOptions()` with `ModelCheckingOptions()` in your test:

```
import org.jetbrains.kotlinx.lincheck.annotations.*
import org.jetbrains.kotlinx.lincheck.check
import org.jetbrains.kotlinx.lincheck.strategy.managed.modelchecking.*
import org.junit.*

class CounterTest {
    private val c = Counter() // Initial state

    // Operations on the Counter
    @Operation
    fun inc() = c.inc()

    @Operation
    fun get() = c.get()

    @Test // Run the test
    fun modelCheckingTest() = ModelCheckingOptions().check(this::class)
}
```

How model checking works

Most bugs in complicated concurrent algorithms can be reproduced with classic interleavings, switching the execution from one thread to another. Besides, model checkers for weak memory models are very complicated, so Lincheck uses a bounded model checking under the *sequential consistency memory model*.

In short, Lincheck analyzes all interleavings, starting with one context switch, then two, continuing the process until the specified number of interleaving is examined. This strategy allows finding an incorrect schedule with the lowest possible number of context switches, making further bug investigation easier.

To control the execution, Lincheck inserts special switch points into the testing code. These points identify where a context switch can be performed. Essentially, these are shared memory accesses, such as field and array element reads or updates in the JVM, as well as `wait/notify` and `park/unpark` calls. To insert a switch point, Lincheck transforms the testing code on the fly using the ASM framework, adding internal function invocations to the existing code.

As the model checking strategy controls the execution, Lincheck can provide the trace that leads to the invalid interleaving, which is extremely helpful in practice. You can see the example of trace for the incorrect execution of the `Counter` in the Write your first test with Lincheck tutorial.

Which testing strategy is better?

The *model checking strategy* is preferable for finding bugs under the sequentially consistent memory model since it ensures better coverage and provides a failing execution trace if an error is found.

Although *stress testing* doesn't guarantee any coverage, checking algorithms for bugs introduced by low-level effects, such as a missed `volatile` modifier, is still helpful. Stress testing is also a great help in discovering rare bugs that require many context switches to reproduce, and it's impossible to analyze them all due to the current restrictions in the model checking strategy.

Configure the testing strategy

To configure the testing strategy, set options in the `<TestingMode>Options` class.

1. Set the options for scenario generation and execution for the `CounterTest`:

```
import org.jetbrains.kotlinx.lincheck.annotations.*
import org.jetbrains.kotlinx.lincheck.check
import org.jetbrains.kotlinx.lincheck.strategy.stress.*
import org.junit.*

class CounterTest {
    private val c = Counter()

    @Operation
    fun inc() = c.inc()

    @Operation
    fun get() = c.get()

    @Test
    fun stressTest() = StressOptions() // Stress testing options:
        .actorsBefore(2) // Number of operations before the parallel part
        .threads(2) // Number of threads in the parallel part
        .actorsPerThread(2) // Number of operations in each thread of the paralle
l part
        .actorsAfter(1) // Number of operations after the parallel part
        .iterations(100) // Generate 100 random concurrent scenarios
        .invocationsPerIteration(1000) // Run each generated scenario 1000 times
        .check(this::class) // Run the test
}
```

2. Run `stressTest()` again, Lincheck will generate scenarios similar to the one below:

```
| ------------------ |
| Thread 1 | Thread 2 |
```

```
  | ------------------- |
  | inc()    |          |
  | inc()    |          |
  | ------------------- |
  | get()    | inc()    |
  | inc()    | get()    |
  | ------------------- |
  | inc()    |          |
  | ------------------- |
```

Here, there are two operations before the parallel part, two threads for each of the two operations, followed after that by a single operation in the end.

You can configure your model checking tests in the same way.

Scenario minimization

You may already have noticed that detected errors are usually represented with a scenario smaller than the specified in the test configuration. Lincheck tries to minimize the error, actively removing an operation while it's possible to keep the test from failing.

Here's the minimized scenario for the counter test above:

```
= Invalid execution results =
| ------------------- |
| Thread 1 | Thread 2 |
| ------------------- |
| inc()    | inc()    |
| ------------------- |
```

As it's easier to analyze smaller scenarios, scenario minimization is enabled by default. To disable this feature, add `minimizeFailedScenario(false)` to the `[Stress, ModelChecking]Options` configuration.

Logging data structure states

Another useful feature for debugging is *state logging*. When analyzing an interleaving that leads to an error, you usually draw the data structure changes on a sheet of paper, changing the state after each event. To automize this procedure, you can provide a special method that returns a `String` representation of the data structure, so Lincheck prints the state representation after each event in the interleaving that modifies the data structure.

For this, define a method that doesn't take arguments and is marked with the `@StateRepresentation` annotation. The method should be thread-safe, non-blocking, and never modify the data structure.

1. In the `Counter` example, the `String` representation is simply the value of the counter. Thus, to print the counter states in the trace, add the `stateRepresentation()` function to the `CounterTest`:

```
import org.jetbrains.kotlinx.lincheck.annotations.*
import org.jetbrains.kotlinx.lincheck.check
import org.jetbrains.kotlinx.lincheck.strategy.managed.modelchecking.*
import org.junit.Test

class CounterTest {
    private val c = Counter()

    @Operation
    fun inc() = c.inc()

    @Operation
    fun get() = c.get()

    @StateRepresentation
    fun stateRepresentation() = c.get().toString()

    @Test
    fun modelCheckingTest() = ModelCheckingOptions().check(this::class)
}
```

2. Run the `modelCheckingTest()` now and check the states of the `Counter` printed at the switch points that modify the counter state (they start with `STATE:`):

```
 = Invalid execution results =
| ------------------- |
| Thread 1 | Thread 2 |
| ------------------- |
| STATE: 0            |
| ------------------- |
| inc(): 1 | inc(): 1 |
| ------------------- |
| STATE: 1            |
| ------------------- |

 The following interleaving leads to the error:
| ------------------------------------------------------------------------ |
| Thread 1 |                          Thread 2                             |
| ------------------------------------------------------------------------ |
|          | inc()                                                         |
|          |   inc(): 1 at CounterTest.inc(CounterTest.kt:10)              |
|          |     value.READ: 0 at Counter.inc(BasicCounterTest.kt:10)      |
|          |     switch                                                    |
```

```
| inc(): 1 |                                                                 |
| STATE: 1 |                                                                 |
|          |     value.WRITE(1) at Counter.inc(BasicCounterTest.kt:10) |
|          |     STATE: 1                                                   |
|          |     value.READ: 1 at Counter.inc(BasicCounterTest.kt:10)  |
|          |   result: 1                                                    |
| ---------------------------------------------------------------------- |
```

In case of stress testing, Lincheck prints the state representation right before and after the parallel part of the scenario, as well as at the end.

> - Get the full code of these examples
> - See more test examples



Next step

Learn how to configure arguments passed to the operations and when it can be useful.

操作参数

In this tutorial, you'll learn how to configure operation arguments.

Consider this straightforward `MultiMap` implementation below. It's based on the `ConcurrentHashMap`, internally storing a list of values:

```
import java.util.concurrent.*

class MultiMap<K, V> {
    private val map = ConcurrentHashMap<K, List<V>>()

    // Maintains a list of values
    // associated with the specified key.
    fun add(key: K, value: V) {
        val list = map[key]
        if (list == null) {
            map[key] = listOf(value)
        } else {
            map[key] = list + value
        }
    }

    fun get(key: K): List<V> = map[key] ?: emptyList()
}
```

Is this `MultiMap` implementation linearizable? If not, an incorrect interleaving is more likely to be detected when accessing a small range of keys, thus, increasing the possibility of processing the same key concurrently.

For this, configure the generator for a `key: Int` parameter:

1. Declare the `@Param` annotation.
2. Specify the integer generator class: `@Param(gen = IntGen::class)`. Lincheck supports random parameter generators for almost all primitives and strings out of the box.
3. Define the range of values generated with the string configuration `@Param(conf = "1:2")`.
4. Specify the parameter configuration name (`@Param(name = "key")`) to share it for several operations.

   Below is the stress test for `MultiMap` that generates keys for `add(key, value)` and `get(key)` operations in the range of `[1..2]`:

   ```
   import java.util.concurrent.*
   import org.jetbrains.kotlinx.lincheck.annotations.*
   import org.jetbrains.kotlinx.lincheck.check
   ```

```
import org.jetbrains.kotlinx.lincheck.paramgen.*
import org.jetbrains.kotlinx.lincheck.strategy.stress.*
import org.jetbrains.kotlinx.lincheck.strategy.managed.modelchecking.*
import org.junit.*

class MultiMap<K, V> {
    private val map = ConcurrentHashMap<K, List<V>>()

    // Maintains a list of values
    // associated with the specified key.
    fun add(key: K, value: V) {
        val list = map[key]
        if (list == null) {
            map[key] = listOf(value)
        } else {
            map[key] = list + value
        }
    }

    fun get(key: K): List<V> = map[key] ?: emptyList()
}

@Param(name = "key", gen = IntGen::class, conf = "1:2")
class MultiMapTest {
    private val map = MultiMap<Int, Int>()

    @Operation
    fun add(@Param(name = "key") key: Int, value: Int) = map.add(key, value)

    @Operation
    fun get(@Param(name = "key") key: Int) = map.get(key)

    @Test
    fun stressTest() = StressOptions().check(this::class)

    @Test
    fun modelCheckingTest() = ModelCheckingOptions().check(this::class)
}
```

5. Run the `stressTest()` and see the following output:

```
= Invalid execution results =
| ---------------------------------- |
|    Thread 1     |     Thread 2     |
| ---------------------------------- |
| add(2, 0): void | add(2, -1): void |
| ---------------------------------- |
| get(2): [0]     |                  |
| ---------------------------------- |
```

6. Finally, run `modelCheckingTest()`. It fails with the following output:

```
= Invalid execution results =
| --------------------------------- |
|    Thread 1     |     Thread 2    |
| --------------------------------- |
| add(2, 0): void | add(2, -1): void |
| --------------------------------- |
| get(2): [-1]    |                  |
| --------------------------------- |

---
All operations above the horizontal line | ----- | happen before those below the l
ine
---

The following interleaving leads to the error:
| ------------------------------------------------------------------------ |
|    Thread 1     |                       Thread 2                       |
| ------------------------------------------------------------------------ |
|                 | add(2, -1)                                           |
|                 |   add(2,-1) at MultiMapTest.add(MultiMap.kt:31)      |
|                 |     get(2): null at MultiMap.add(MultiMap.kt:15)     |
|                 |     switch                                           |
| add(2, 0): void |                                                      |
|                 |     put(2,[-1]): [0] at MultiMap.add(MultiMap.kt:17) |
|                 |   result: void                                       |
| ------------------------------------------------------------------------ |
```

Due to the small range of keys, Lincheck quickly reveals the race: when two values are being added concurrently by the same key, one of the values may be overwritten and lost.

> Get the full code.
>
> 

Next step

Learn how to test data structures that set access constraints on the execution, such as single-producer single-consumer queues.

数据结构约束

Some data structures may require a part of operations not to be executed concurrently, such as single-producer single-consumer queues. Lincheck provides out-of-the-box support for such contracts, generating concurrent scenarios according to the restrictions.

Consider the single-consumer queue from the JCTools library. Let's write a test to check correctness of its `poll()`, `peek()`, and `offer(x)` operations.

In your `build.gradle(.kts)` file, add the JCTools dependency:

> 【Kotlin】

```
dependencies {
    // jctools dependency
    testImplementation("org.jctools:jctools-core:3.3.0")
}
```

> 【Groovy】

```
dependencies {
    // jctools dependency
    testImplementation "org.jctools:jctools-core:3.3.0"
}
```

To meet the single-consumer restriction, ensure that all `poll()` and `peek()` consuming operations are called from a single thread. For that, we can set the `nonParallelGroup` parameter of the corresponding `@Operation` annotations to the same value, e.g. `"consumers"`.

Here is the resulting test:

```
import org.jctools.queues.atomic.*
import org.jetbrains.kotlinx.lincheck.annotations.*
import org.jetbrains.kotlinx.lincheck.check
import org.jetbrains.kotlinx.lincheck.strategy.managed.modelchecking.*
import org.jetbrains.kotlinx.lincheck.strategy.stress.*
import org.junit.*
```

```
class MPSCQueueTest {
    private val queue = MpscLinkedAtomicQueue<Int>()

    @Operation
    fun offer(x: Int) = queue.offer(x)

    @Operation(nonParallelGroup = "consumers")
    fun poll(): Int? = queue.poll()

    @Operation(nonParallelGroup = "consumers")
    fun peek(): Int? = queue.peek()

    @Test
    fun stressTest() = StressOptions().check(this::class)

    @Test
    fun modelCheckingTest() = ModelCheckingOptions().check(this::class)
}
```

Here is an example of the scenario generated for this test:

```
= Iteration 15 / 100 =
| --------------------- |
| Thread 1   | Thread 2  |
| --------------------- |
| poll()     |           |
| poll()     |           |
| peek()     |           |
| peek()     |           |
| peek()     |           |
| --------------------- |
| offer(-1) | offer(0)  |
| offer(0)  | offer(-1) |
| peek()    | offer(-1) |
| offer(1)  | offer(1)  |
| peek()    | offer(1)  |
| --------------------- |
| peek()    |           |
| offer(-2) |           |
| offer(-2) |           |
| offer(2)  |           |
| offer(-2) |           |
| --------------------- |
```

Note that all consuming `poll()` and `peek()` invocations are performed from a single thread, thus satisfying the "single-consumer" restriction.

> Get the full code.
>
> 

Next step

Learn how to check your algorithm for progress guarantees with the model checking strategy.

进度保证

Many concurrent algorithms provide non-blocking progress guarantees, such as lock-freedom and wait-freedom. As they are usually non-trivial, it's easy to add a bug that blocks the algorithm. Lincheck can help you find liveness bugs using the model checking strategy.

To check the progress guarantee of the algorithm, enable the `checkObstructionFreedom` option in `ModelCheckingOptions()`:

```
ModelCheckingOptions().checkObstructionFreedom()
```

Create a `ConcurrentMapTest.kt` file. Then add the following test to detect that `ConcurrentHashMap::put(key: K, value: V)` from the Java standard library is a blocking operation:

```
import org.jetbrains.kotlinx.lincheck.*
import org.jetbrains.kotlinx.lincheck.annotations.*
import org.jetbrains.kotlinx.lincheck.strategy.managed.modelchecking.*
import org.junit.*
import java.util.concurrent.*

class ConcurrentHashMapTest {
    private val map = ConcurrentHashMap<Int, Int>()

    @Operation
    fun put(key: Int, value: Int) = map.put(key, value)

    @Test
    fun modelCheckingTest() = ModelCheckingOptions()
        .actorsBefore(1) // To init the HashMap
        .actorsPerThread(1)
        .actorsAfter(0)
        .minimizeFailedScenario(false)
        .checkObstructionFreedom()
        .check(this::class)
}
```

Run the `modelCheckingTest()`. You should get the following result:

```
= Obstruction-freedom is required but a lock has been found =
| --------------------- |
|  Thread 1  | Thread 2  |
| --------------------- |
| put(1, -1) |           |
| --------------------- |
| put(2, -2) | put(3, 2) |
```

```
| ---------------------- |

---
All operations above the horizontal line | ----- | happen before those below the line
---

The following interleaving leads to the error:
| ------------------------------------------------------------------------------
------------------------------------------------------------------------------
--------- |
|                                        Thread 1
     |                                        Thread 2
          |
| ------------------------------------------------------------------------------
------------------------------------------------------------------------------
--------- |
|
     | put(3, 2)
          |
|
     |   put(3,2) at ConcurrentHashMapTest.put(ConcurrentMapTest.kt:11)
          |
|
     |     putVal(3,2,false) at ConcurrentHashMap.put(ConcurrentHashMap.java:1006)
          |
|
     |       table.READ: Node[]@1 at ConcurrentHashMap.putVal(ConcurrentHashMap.java:1
014)      |
|
     |       tabAt(Node[]@1,0): Node@1 at ConcurrentHashMap.putVal(ConcurrentHashMap.j
ava:1018) |
|
     |       MONITORENTER at ConcurrentHashMap.putVal(ConcurrentHashMap.java:1031)
          |
|
     |       tabAt(Node[]@1,0): Node@1 at ConcurrentHashMap.putVal(ConcurrentHashMap.j
ava:1032) |
|
     |       next.READ: null at ConcurrentHashMap.putVal(ConcurrentHashMap.java:1046)
          |
|
     |       switch
          |
| put(2, -2)
     |
          |
|   put(2,-2) at ConcurrentHashMapTest.put(ConcurrentMapTest.kt:11)
     |
          |
|     putVal(2,-2,false) at ConcurrentHashMap.put(ConcurrentHashMap.java:1006)
     |
          |
|       table.READ: Node[]@1 at ConcurrentHashMap.putVal(ConcurrentHashMap.java:1014)
```

```
      |
           |
|       tabAt(Node[]@1,0): Node@1 at ConcurrentHashMap.putVal(ConcurrentHashMap.java:1
018) |
           |
|       MONITORENTER at ConcurrentHashMap.putVal(ConcurrentHashMap.java:1031)
      |
           |
| ------------------------------------------------------------------------------------
------------------------------------------------------------------------------------
--------- |
```

Now let's add a test for the non-blocking `ConcurrentSkipListMap<K, V>`, expecting the test to pass successfully:

```
class ConcurrentSkipListMapTest {
    private val map = ConcurrentSkipListMap<Int, Int>()

    @Operation
    fun put(key: Int, value: Int) = map.put(key, value)

    @Test
    fun modelCheckingTest() = ModelCheckingOptions()
        .checkObstructionFreedom()
        .check(this::class)
}
```

> The common non-blocking progress guarantees are (from strongest to weakest):
>
> - **wait-freedom**, when each operation is completed in a bounded number of steps no matter what other threads do.
> - **lock-freedom**, which guarantees system-wide progress so that at least one operation is completed in a bounded number of steps while a particular operation may be stuck.
> - **obstruction-freedom**, when any operation is completed in a bounded number of steps if all the other threads pause.
>
> 

At the moment, Lincheck supports only the obstruction-freedom progress guarantees. However, most real-life liveness bugs add unexpected blocking code, so the obstruction-freedom check will also help with lock-free and wait-free algorithms.

> - Get the full code of the example.
> - See another example where the Michael-Scott queue implementation is tested for progress guarantees.
>
> 

Next step

Learn how to specify the sequential specification of the testing algorithm explicitly, improving the Lincheck tests robustness.

时序规范

To be sure that the algorithm provides correct sequential behavior, you can define its *sequential specification* by writing a straightforward sequential implementation of the testing data structure.

> This feature also allows you to write a single test instead of two separate sequential and concurrent tests.
>
> 

To provide a sequential specification of the algorithm for verification:

1. Implement a sequential version of all the testing methods.
2. Pass the class with sequential implementation to the `sequentialSpecification()` option:

   ```
   StressOptions().sequentialSpecification(SequentialQueue::class)
   ```

For example, here is the test to check correctness of `j.u.c.ConcurrentLinkedQueue` from the Java standard library.

```
import org.jetbrains.kotlinx.lincheck.*
import org.jetbrains.kotlinx.lincheck.annotations.*
import org.jetbrains.kotlinx.lincheck.strategy.stress.*
import org.junit.*
import java.util.*
import java.util.concurrent.*

class ConcurrentLinkedQueueTest {
    private val s = ConcurrentLinkedQueue<Int>()

    @Operation
    fun add(value: Int) = s.add(value)

    @Operation
    fun poll(): Int? = s.poll()

    @Test
    fun stressTest() = StressOptions()
        .sequentialSpecification(SequentialQueue::class.java)
        .check(this::class)
}

class SequentialQueue {
    private val s = LinkedList<Int>()

    fun add(x: Int) = s.add(x)
```

```
    fun poll(): Int? = s.poll()
}
```

> Get the full code of the examples.
>
> 

Ktor

Ktor

API 参考

- 标准库（stdlib）↗
- 测试库（kotlin.test）↗
- 协程（kotlinx.coroutines）↗
- 序列化（kotlinx.serialization）↗
- Kotlin I/O 库（kotlinx-io）↗
- 日期与时间（kotlinx-datetime）↗
- JVM 元数据（kotlinx-metadata-jvm）↗
- Ktor↗
- Kotlin Gradle 插件（kotlin-gradle-plugin）↗

标准库（stdlib）

标准库（stdlib）

测试库（kotlin.test）

测试库（kotlin.test）

协程（kotlinx.coroutines）

协程（kotlinx.coroutines）

序列化（kotlinx.serialization）

序列化（kotlinx.serialization）

Kotlin I/O 库（kotlinx-io）

Kotlin I/O 库（kotlinx-io）

日期与时间（kotlinx-datetime）

日期与时间（kotlinx-datetime）

JVM 元数据（kotlinx-metadata-jvm）

JVM 元数据（kotlinx-metadata-jvm）

Ktor

Ktor

Kotlin Gradle 插件（kotlin-gradle-plugin）

Kotlin Gradle 插件（kotlin-gradle-plugin）

语言参考

- 关键字与操作符
- 语法↗
- 语言规范↗

关键字与操作符

硬关键字

以下符号会始终解释为关键字，不能用作标识符：

- `as`
  - 用于类型转换。
  - 为导入指定一个别名
- `as?` 用于安全类型转换。
- `break` 终止循环的执行。
- `class` 声明一个类。
- `continue` 继续最近层循环的下一步。
- `do` 开始一个 do/while 循环（后置条件的循环）。
- `else` 定义一个 if 表达式条件为 false 时执行的分支。
- `false` 指定布尔类型的“假”值。
- `for` 开始一个 for 循环。
- `fun` 声明一个函数。
- `if` 开始一个 if 表达式。
- `in`
  - 指定在 for 循环中迭代的对象。
  - 用作中缀操作符以检测一个值属于一个区间、 一个集合或者其他定义“contains”方法的实体。
  - 在 when 表达式中用于上述目的。
  - 将一个类型参数标记为逆变。
- `!in`
  - 用作中缀操作符以检测一个值不属于一个区间、 一个集合或者其他定义“contains”方法的实体。
  - 在 when 表达式中用于上述目的。
- `interface` 声明一个接口。
- `is`
  - 检测一个值具有指定类型。
  - 在 when 表达式中用于上述目的。
- `!is`
  - 检测一个值不具有指定类型。
  - 在 when 表达式中用于上述目的。
- `null` 是表示不指向任何对象的对象引用的常量。
- `object` 同时声明一个类及其实例。

- `package` 指定当前文件的包。
- `return` 从最近层的函数或匿名函数返回。
- `super`
  - 引用一个方法或属性的超类实现。
  - 在次构造函数中调用超类构造函数。
- `this`
  - 引用当前接收者。
  - 在次构造函数中调用同一个类的另一个构造函数。
- `throw` 抛出一个异常。
- `true` 指定布尔类型的“真”值。
- `try` 开始一个异常处理块。
- `typealias` 声明一个类型别名。
- `typeof` 保留以供未来使用。
- `val` 声明一个只读属性或局部变量。
- `var` 声明一个可变属性或局部变量。
- `when` 开始一个 when 表达式（执行其中一个给定分支）。
- `while` 开始一个 while 循环（前置条件的循环）。

软关键字

以下符号在适用的上下文中充当关键字，而在其他上下文中可用作标识符：

- `by`
  - 将接口的实现委托给另一个对象。
  - 将属性访问器的实现委托给另一个对象。
- `catch` 开始一个处理指定异常类型的块。
- `constructor` 声明一个主构造函数或次构造函数。
- `delegate` 用作注解使用处目标。
- `dynamic` 引用一个 Kotlin/JS 代码中的动态类型。
- `field` 用作注解使用处目标。
- `file` 用作注解使用处目标。
- `finally` 开始一个当 try 块退出时总会执行的块。
- `get`
  - 声明属性的 getter。
  - 用作注解使用处目标。
- `import` 将另一个包中的声明导入当前文件。
- `init` 开始一个初始化块。
- `param` 用作注解使用处目标。
- `property` 用作注解使用处目标。
- `receiver` 用作注解使用处目标。

- `set`
  - 声明属性的 setter。
  - 用作注解使用处目标。
- `setparam` 用作注解使用处目标。
- `value` with the `class` keyword declares an inline class。
- `where` 指定泛型类型参数的约束。

修饰符关键字

以下符号作为声明中修饰符列表中的关键字，并可用作其他上下文中的标识符：

- `abstract` 将一个类或成员标记为抽象。
- `actual` 表示多平台项目中的一个平台相关实现。
- `annotation` 声明一个注解类。
- `companion` 声明一个伴生对象。
- `const` 将属性标记为编译期常量。
- `crossinline` 禁止传递给内联函数的 lambda 中的非局部返回。
- `data` 指示编译器为类生成典型成员。
- `enum` 声明一个枚举。
- `expect` 将一个声明标记为平台特有，并期待在平台模块中实现。
- `external` 将一个声明标记为在 Kotlin 外实现（通过 JNI 访问或者在 JavaScript 中实现）。
- `final` 禁止成员覆盖。
- `infix` 允许用中缀表示法调用函数。
- `inline` 告诉编译器在调用处内联传给它的函数和 lambda 表达式。
- `inner` 允许在嵌套类中引用外部类实例。
- `internal` 将一个声明标记为在当前模块中可见。
- `lateinit` 允许在构造函数之外初始化非空属性。
- `noinline` 关闭传给内联函数的 lambda 表达式的内联。
- `open` 允许一个类子类化或覆盖成员。
- `operator` 将一个函数标记为重载一个操作符或者实现一个约定。
- `out` 将类型参数标记为协变。
- `override` 将一个成员标记为超类成员的覆盖。
- `private` 将一个声明标记为在当前类或文件中可见。
- `protected` 将一个声明标记为在当前类及其子类中可见。
- `public` 将一个声明标记为在任何地方可见。
- `reified` 将内联函数的类型参数标记为在运行时可访问。
- `sealed` 声明一个密封类（限制子类化的类）。
- `suspend` 将一个函数或 lambda 表达式标记为挂起式（可用做协程）。
- `tailrec` 将一个函数标记为尾递归（允许编译器将递归替换为迭代）。

- `vararg` 允许一个参数传入可变数量的参数。

特殊标识符

以下标识符由编译器在指定上下文中定义，并且可以用作其他上下文中的常规标识符：

- `field` 用在属性访问器内部来引用该属性的幕后字段。
- `it` 用在 lambda 表达式内部来隐式引用其参数。

操作符和特殊符号

Kotlin 支持以下操作符和特殊符号：

- `+`、`-`、`*`、`/`、`%` —— 数学操作符
  - `*` 也用于将数组传递给 vararg 参数。
- `=`
  - 赋值操作符。
  - 也用于指定参数的默认值。
- `+=`、`-=`、`*=`、`/=`、`%=` —— 广义赋值操作符。
- `++`、`--` —— 递增与递减操作符。
- `&&`、`||`、`!` —— 逻辑“与”、“或”、“非”操作符（对于位运算，请使用相应的中缀函数）。
- `==`、`!=` —— 相等操作符（对于非原生类型会翻译为调用 `equals()`）。
- `===`、`!==` —— 引用相等操作符。
- `<`、`>`、`<=`、`>=` —— 比较操作符（对于非原生类型会翻译为调用 `compareTo()`）。
- `[`、`]` —— 索引访问操作符（会翻译为调用 `get` 与 `set`）。
- `!!` 断言一个表达式非空。
- `?.` 执行安全调用（如果接收者非空，就调用一个方法或访问一个属性）。
- `?:` 如果左侧的值为空，就取右侧的值（elvis 操作符）。
- `::` 创建一个成员引用或者一个类引用。
- `..`、`..<` 创建区间。
- `:` 分隔声明中的名称与类型。
- `?` 将类型标记为可空。
- `->`
  - 分隔 lambda 表达式的参数与主体。
  - 分隔在函数类型中的参数类型与返回类型声明。
  - 分隔 when 表达式分支的条件与代码体。
- `@`
  - 引入一个注解。

    - 引入或引用一个循环标签。
    - 引入或引用一个 lambda 表达式标签。
    - 引用一个来自外部作用域的 “this”表达式。
    - 引用一个外部超类。
- `;` 分隔位于同一行的多个语句。
- `$` 在字符串模版中引用变量或者表达式。
- `_`
    - 在 lambda 表达式中代替未使用的参数。
    - 在解构声明中代替未使用的参数。

For operator precedence, see this reference in Kotlin grammar.

语法

语法

语言规范

语言规范

工具

- 构建工具
  - Gradle
    - Gradle 概述
    - Gradle 入门——教程
    - 配置 Gradle 项目
    - Gradle 最佳实践
    - Kotlin Gradle 插件中的编译器选项
    - Kotlin Gradle 插件中的编译项与缓存
    - 支持 Gradle 插件变体
  - Maven
  - Ant
- Dokka
  - 简介
  - Dokka 入门
  - 运行 Dokka
    - Gradle
    - Maven
    - CLI
  - 输出格式
    - HTML
    - Markdown
    - Javadoc
  - Dokka 插件
  - 模块文档
- 用于 Kotlin 开发的 IDE
- 使用 IntelliJ IDEA 迁移到 Kotlin 代码风格
- Kotlin Notebook
- 用于 Kotlin 的 Lets-Plot
- 运行代码片段
- Kotlin 与 TeamCity 的持续集成
- 编写代码文档：KDoc
- Kotlin 与 OSGi

构建工具

- Gradle
  - Gradle 概述
  - Gradle 入门——教程
  - 配置 Gradle 项目
  - Gradle 最佳实践
  - Kotlin Gradle 插件中的编译器选项
  - Kotlin Gradle 插件中的编译项与缓存
  - 支持 Gradle 插件变体
- Maven
- Ant

Gradle

- Gradle 概述
- Gradle 入门——教程
- 配置 Gradle 项目
- Gradle 最佳实践
- Kotlin Gradle 插件中的编译器选项
- Kotlin Gradle 插件中的编译项与缓存
- 支持 Gradle 插件变体

Gradle

Gradle 是一个用于管理并自动化构建流程的构建工具。它会帮你下载项目所需的依赖、 将你的项目代码打包并准备编译。你可以在 Gradle 官网学习 Gradle 相关的基础知识和特性。

你可以参照这些针对不同平台的 Gradle 配置说明 来为你的项目进行配置， 或者参考这个简易的逐步教程来搭建起一个简单的，在后台显示出“Hello World”的 Kotlin 项目。

> 你可以在此处获取到关于 Kotlin、Gradle 和 Android Gradle 插件版本之间的兼容性信息。
>
> 

在这个章节中，你同样可以学到：

- 编译器选项的配置和传递。
- 增量编译、缓存支持、构建日志以及 Kotlin 守护进程。
- 对 Gradle 插件变体的支持。

下一步做什么？

了解更多关于：

- **Gradle Kotlin DSL。** Gradle Kotlin DSL 是一种 用于高效、快捷编写构建脚本的领域特定语言。
- 注解处理。 Kotlin 通过 Kotlin Symbol processing API 支持注解处理。
- 生成文档。 如需生成 Kotlin 项目的文档，请使用 Dokka； 相关配置说明请参见 Dokka README。 Dokka 支持混合语言项目，并且可以生成多种格式的输出， 包括标准 Javadoc。
- **OSGi。** 关于 OSGi 支持请参见 Kotlin OSGi 页。

以 Gradle 与 Kotlin/JVM 入门

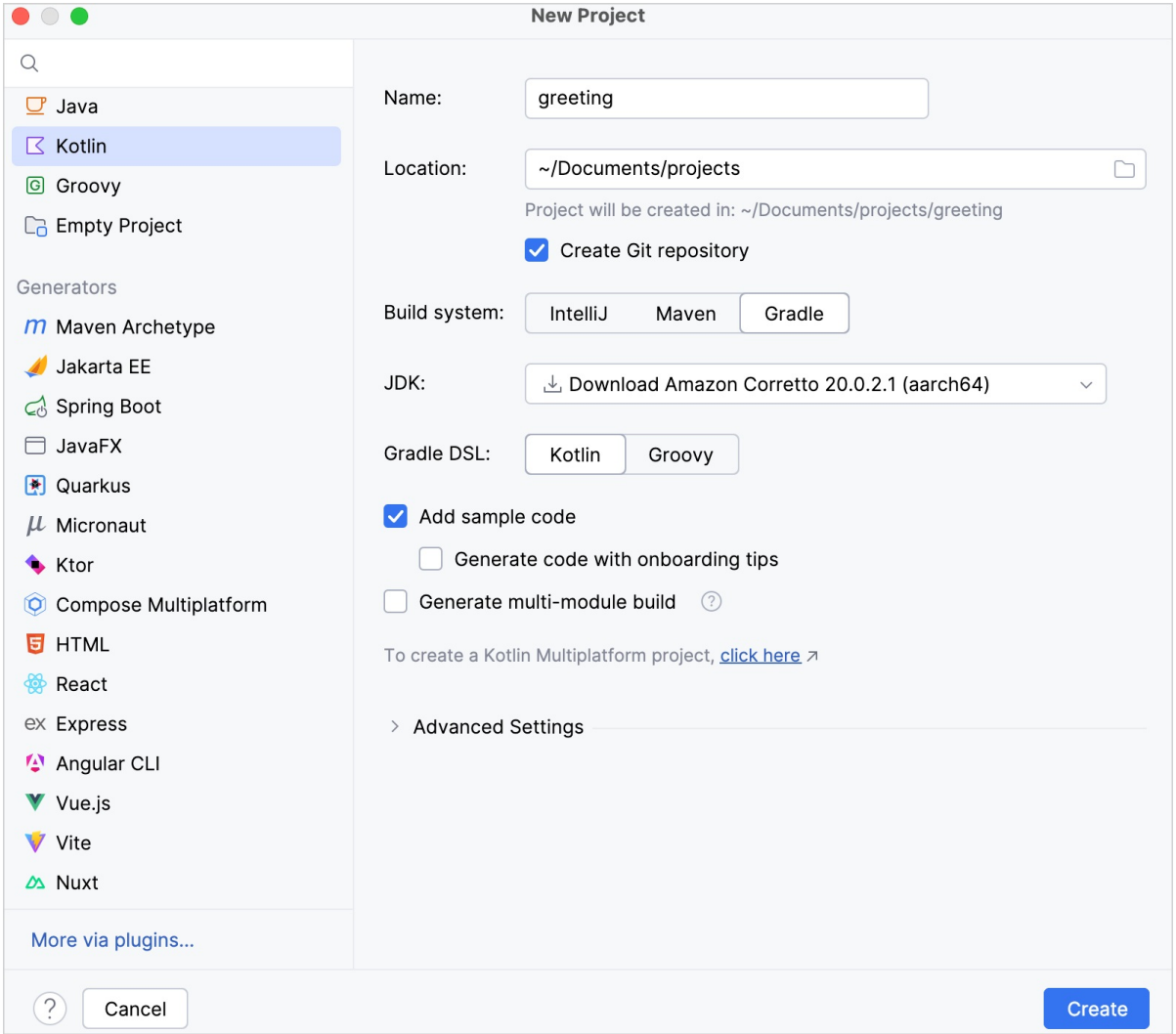
本教程演示了如何通过 InteliJ IDEA 和 Gradle 来创建一个 JVM 控制台应用程序。

首先，请先下载最新版本的 IntelliJ IDEA。

创建项目

1. 在 IntelliJ IDEA 中选择文件 | 新建 | 项目。
2. 在面板的左侧，选择**Kotlin**。
3. 根据需要修改项目的名称和位置。

> 勾选创建 **Git** 仓库来让你的项目弃启用版本控制。 你也可以在项目创建之后再来创建。



4. 在构建系统这一行选择 **Gradle**。
5. 在 **JDK** 这一行选择你想要应用在项目中的 JDK 版本。
    - 如果你的计算机中已经安装了 JDK，但是 IDE 并没有识别。选择添加 **JDK** 并选择 JDK 的 Home 路径。
    - 如果你的计算机中尚未安装 JDK，选择 下载 **JDK**。
6. Select the **Kotlin** DSL for Gradle.
7. 勾选添加示例代码来创建一个初始的 `"Hello World!"` 应用。

    > You can also enable the **Generate code with onboarding tips** option to add some additional useful comments to your sample code.
    >
    > 

8. 点击创建。

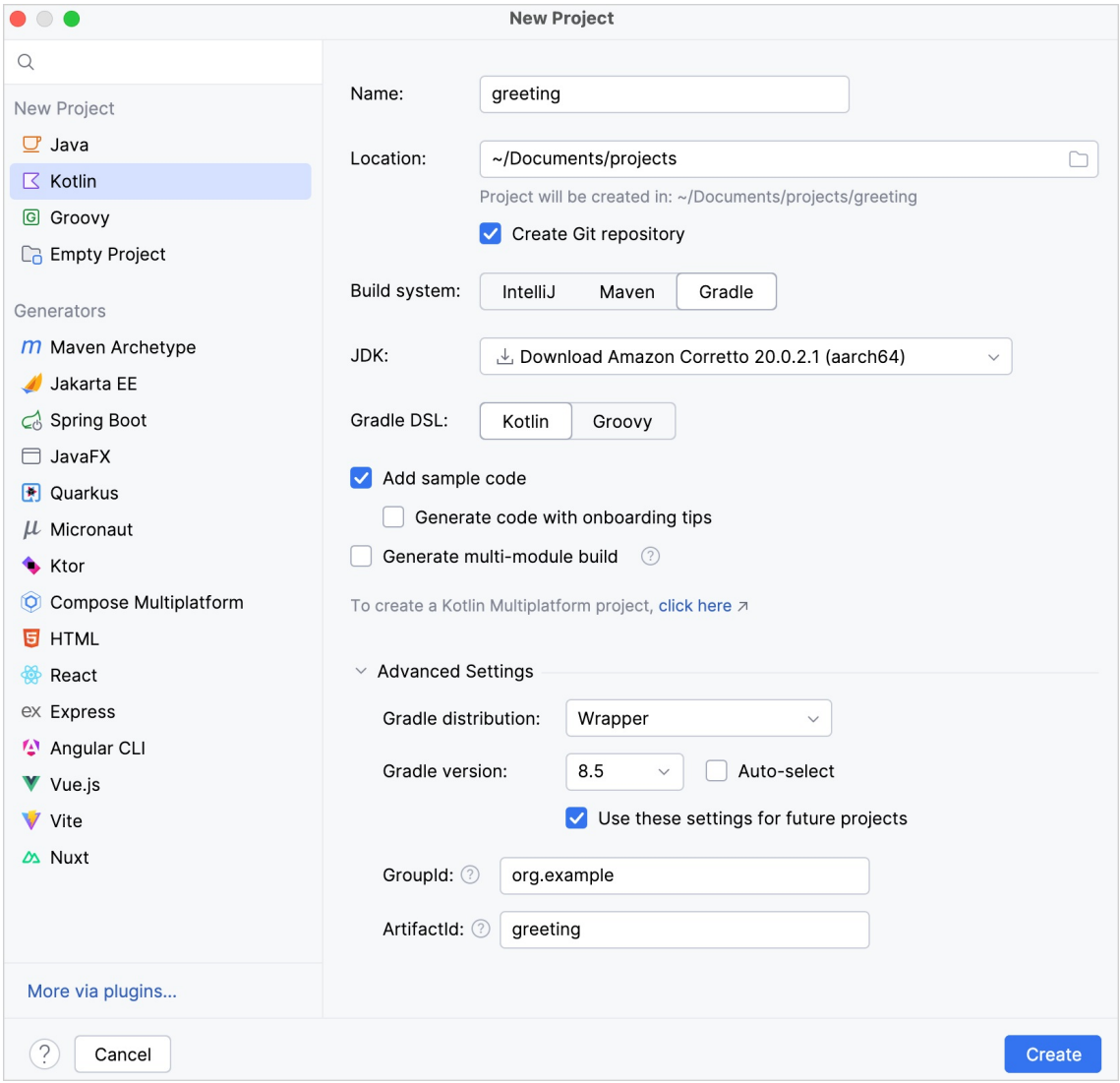
你已经成功创建了一个 Gradle 项目！

Specify a Gradle version for your project

You can explicitly specify a Gradle version for your project under the **Advanced Settings** section, either by using the Gradle Wrapper or a local installation of Gradle:

- **Gradle Wrapper:**
    1. From the **Gradle distribution** list, select **Wrapper**.
    2. Disable the **Auto-select** checkbox.
    3. From the **Gradle version** list, select your Gradle version.
- **Local installation:**

    1. From the **Gradle distribution** list, select **Local installation**.
    2. For **Gradle location**, specify the path of your local Gradle version.



探索构建脚本

打开 `build.gradle.kts` 文件。这个是 Gradle Kotlin的构建脚本，它包含了和 Kotlin 相关的配置以及应用所需的其他部分。

```
plugins {
    kotlin("jvm") version "2.1.21" // Kotlin version to use
}

group = "org.example" // 公司域名，例如：`org.jetbrains`
version = "1.0-SNAPSHOT" // 构建工件的版本号

repositories { // 依赖的源，见下方 1▫ 注释
    mavenCentral() // Maven 中心仓库，见下方 2▫ 注释
}

dependencies { // 项目中需要用到的库，见下方 3▫ 注释
    // 在仓库在找到你需要的依赖，并将他们的依赖添加代码复制到这里
    testImplementation(kotlin("test")) // Kotlin 的测试库
```

```
}

tasks.test { // 见下方 4□ 注释
    useJUnitPlatform() // 用于测试的 JUnitPlatform 依赖，见下方 6□ 注释
}
```

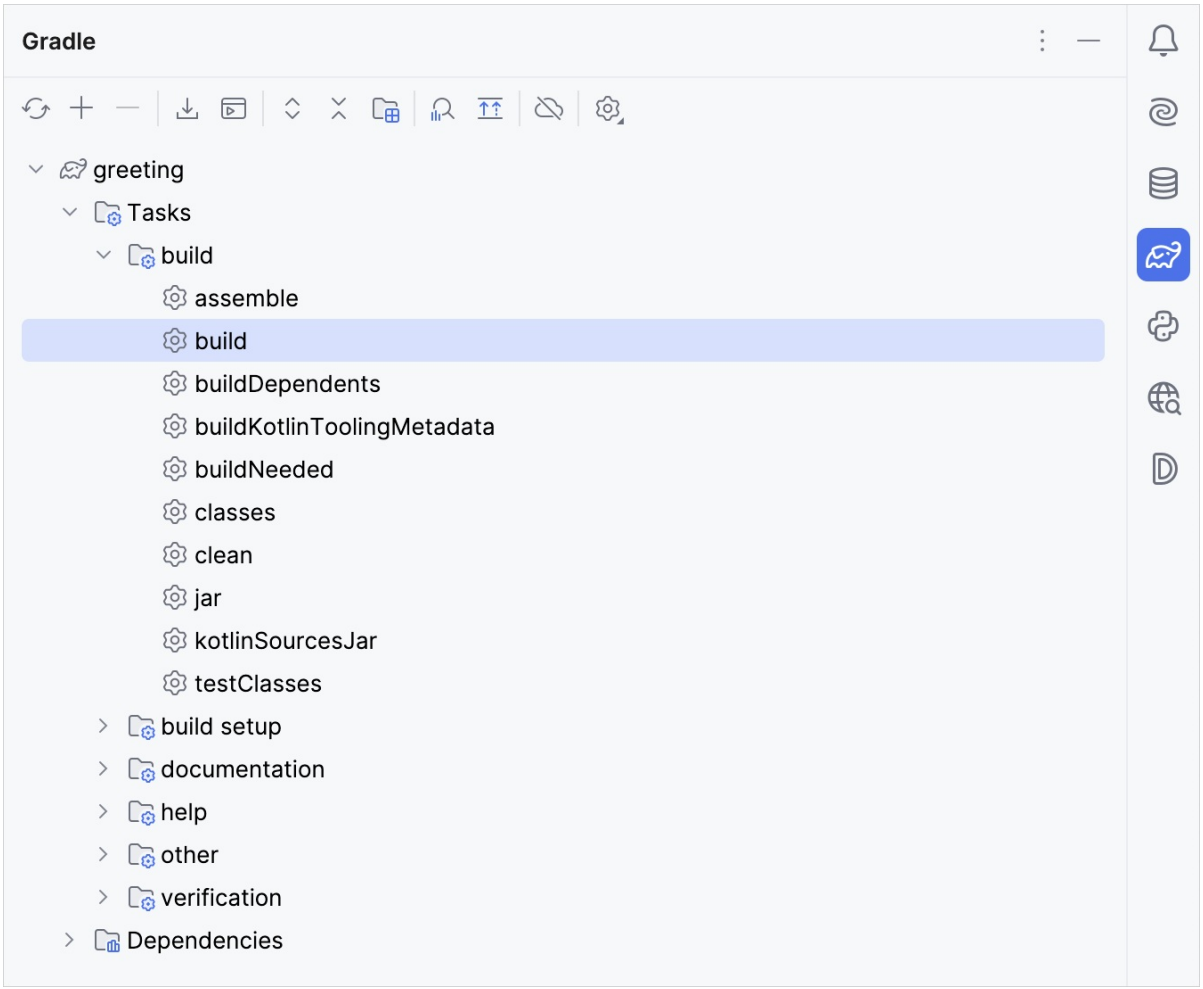
- 1 了解更多关于依赖源的信息。
- 2 Maven 中心仓库也可以使用 Google 的 Maven 仓库或者公司的私有仓库替代。
- 3 了解更多关于依赖声明的信息。
- 4 了解更多关于任务的信息。
- 5 用于测试的 JUnitPlatform.

如你所见，在 Gradle 的构建文件中也有一些关于 Kotlin 的配置项：

1. 在 `plugins {}` 块中，有一行 `kotlin("jvm")` 。这个插件定义了项目中所用的 Kotlin 版本。
2. 在 `dependencies {}` 块中，有一行 `testImplementation(kotlin("test"))` 。 了解更多关于配置用于测试的库的信息。

运行应用

1. Open the Gradle window by selecting **View** | **Tool Windows** | **Gradle**:

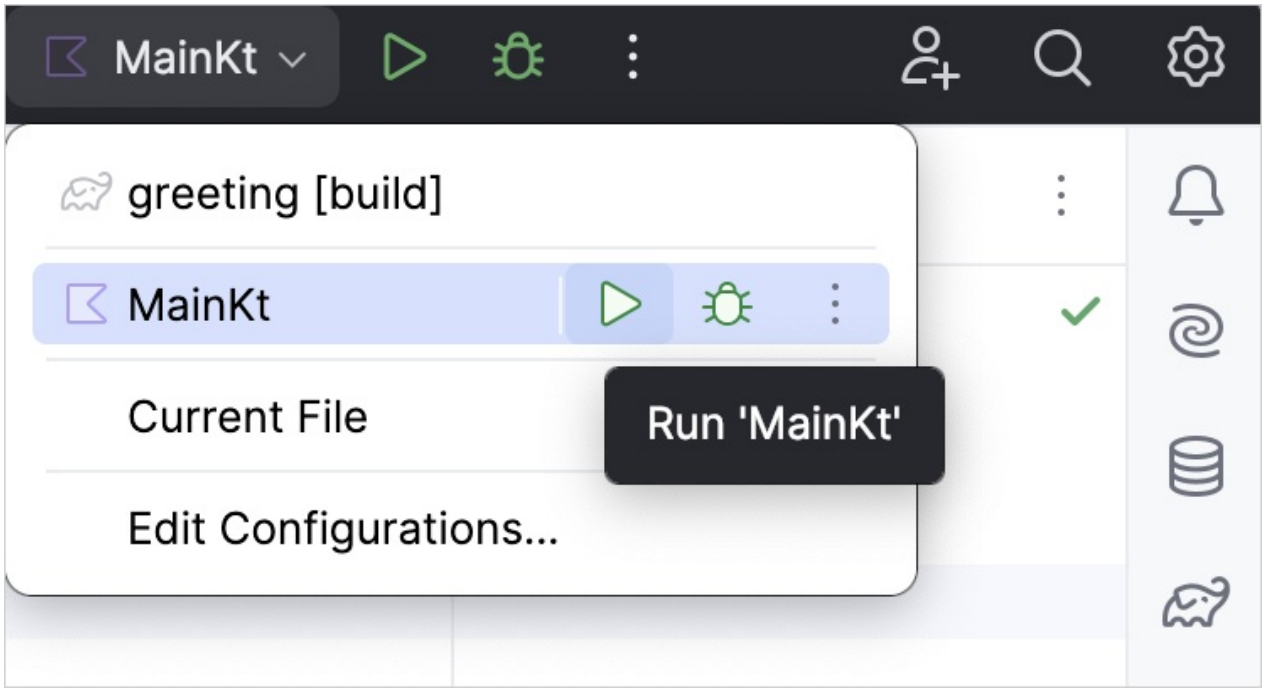


2. Execute the **build** Gradle task in `Tasks\build\`. In the **Build** window, `BUILD SUCCESSFUL` appears. It means that Gradle built the application successfully.

3. In `src/main/kotlin`, open the `Main.kt` file:

    - `src` directory contains Kotlin source files and resources.
    - `Main.kt` file contains sample code that will print `Hello World!`.

4. Run the application by clicking the green **Run** icon in the gutter and select **Run 'MainKt'**.



你可以在运行的窗口中看到结果：



恭喜！你刚刚运行了你的第一个 Kotlin 应用。

下一步做什么？

了解更多关于：

- Gradle 构建文件的属性。
- 针对不同平台配置库的依赖关系。
- 编译器选项的配置和传递。
- 增量编译、缓存支持、构建日志以及 Kotlin 守护进程。

配置 Gradle 项目

To build a Kotlin project with Gradle, you need to add the Kotlin Gradle plugin to your build script file `build.gradle(.kts)` and configure the project's dependencies there.

> To learn more about the contents of a build script, visit the Explore the build script section.
>
> 

Apply the plugin

To apply the Kotlin Gradle plugin, use the `plugins{}` block from the Gradle plugins DSL:

> 【Kotlin】

```
plugins {
    // Replace `<……>` with the plugin name appropriate for your target environment
    kotlin("<……>") version "2.1.21"
    // For example, if your target environment is JVM:
    // kotlin("jvm") version "2.1.21"
}
```

> 【Groovy】

```
plugins {
    // Replace `<……>` with the plugin name appropriate for your target environment
    id 'org.jetbrains.kotlin.<……>' version '2.1.21'
    // For example, if your target environment is JVM:
    // id 'org.jetbrains.kotlin.jvm' version '2.1.21'
}
```

> The Kotlin Gradle plugin (KGP) and Kotlin share the same version numbering.
>
> 

When configuring your project, check the Kotlin Gradle plugin (KGP) compatibility with available Gradle versions. In the following table, there are the minimum and maximum **fully supported** versions of Gradle and Android Gradle plugin (AGP):

KGP version	Gradle min and max versions	AGP min and max versions
2.1.20	7.6.3–8.12.1	7.3.1–8.7.2
2.1.0–2.1.10	7.6.3–8.10*	7.3.1–8.7.2
2.0.20–2.0.21	6.8.3–8.8*	7.1.3–8.5
2.0.0	6.8.3–8.5	7.1.3–8.3.1
1.9.20–1.9.25	6.8.3–8.1.1	4.2.2–8.1.0
1.9.0–1.9.10	6.8.3–7.6.0	4.2.2–7.4.0
1.8.20–1.8.22	6.8.3–7.6.0	4.1.3–7.4.0
1.8.0–1.8.11	6.8.3–7.3.3	4.1.3–7.2.1
1.7.20–1.7.22	6.7.1–7.1.1	3.6.4–7.0.4
1.7.0–1.7.10	6.7.1–7.0.2	3.4.3–7.0.2
1.6.20–1.6.21	6.1.1–7.0.2	3.4.3–7.0.2

> *Kotlin 2.0.20–2.0.21 and Kotlin 2.1.0–2.1.10 are fully compatible with Gradle up to 8.6. Gradle versions 8.7–8.10 are also supported, with only one exception: If you use the Kotlin Multiplatform Gradle plugin, you may see deprecation warnings in your multiplatform projects calling the `withJava()` function in the JVM target. For more information, see Java source sets created by default.
>
> 

You can also use Gradle and AGP versions up to the latest releases, but if you do, keep in mind that you might encounter deprecation warnings or some new features might not work.

For example, the Kotlin Gradle plugin and the `kotlin-multiplatform` plugin 2.1.21 require the minimum Gradle version of 7.6.3 for your project to compile.

Similarly, the maximum fully supported version is 8.12.1. It doesn't have deprecated Gradle methods and properties, and supports all the current Gradle features.

Kotlin Gradle plugin data in a project

By default, the Kotlin Gradle plugin stores persistent project-specific data at the root of the project, in the `.kotlin` directory.

> Do not commit the `.kotlin` directory to version control. For example, if you are using Git, add `.kotlin` to your project's `.gitignore` file.
>
> 

There are properties you can add to the `gradle.properties` file of your project to configure this behavior:

Gradle property	Description
`kotlin.project.persistent.dir`	Configures the location where your project-level data is stored. Default: `<project-root-directory>/.kotlin`
`kotlin.project.persistent.dir.gradle.disableWrite`	Controls whether writing Kotlin data to the `.gradle` directory is disabled (for backward compatibility with older IDEA versions). Default: false

面向 JVM

如需面向 JVM 平台，请应用 Kotlin JVM 插件。

> 【Kotlin】

```
plugins {
    kotlin("jvm") version "2.1.21"
}
```

> 【Groovy】

```
plugins {
    id "org.jetbrains.kotlin.jvm" version "2.1.21"
}
```

在这个代码块中的 `version` 应该是字面值，并且不能在其他构建脚本中应用。

Kotlin and Java sources

Kotlin 源代码可以与 Java 源代码放在相同目录或者不同目录中。

默认约定是使用不同的目录：

```
project
    - src
        - main (root)
            - kotlin
            - java
```

> Do not store Java `.java` files in the `src/*/kotlin` directory, as the `.java` files will not be compiled.
>
> Instead, you can use `src/main/java`.
>
> 

如果不使用默认约定，那么应该更新相应的 `sourceSets` 属性：

> 【Kotlin】

```
sourceSets.main {
    java.srcDirs("src/main/myJava", "src/main/myKotlin")
}
```

> 【Groovy】

```
sourceSets {
    main.kotlin.srcDirs += 'src/main/myKotlin'
    main.java.srcDirs += 'src/main/myJava'
}
```

Check for JVM target compatibility of related compile tasks

In the build module, you may have related compile tasks, for example:

- `compileKotlin` and `compileJava`
- `compileTestKotlin` and `compileTestJava`

> `main` and `test` source set compile tasks are not related.



For related tasks like these, the Kotlin Gradle plugin checks for JVM target compatibility. Different values of the `jvmTarget` attribute in the `kotlin` extension or task and `targetCompatibility` in the `java` extension or task cause JVM target incompatibility. For example: the `compileKotlin` task has `jvmTarget=1.8`, and the `compileJava` task has (or inherits) `targetCompatibility=15`.

Configure the behavior of this check for the whole project by setting the `kotlin.jvm.target.validation.mode` property in the `gradle.properties` file to:

- `error` – the plugin fails the build; the default value for projects on Gradle 8.0+.
- `warning` – the plugin prints a warning message; the default value for projects on Gradle less than 8.0.
- `ignore` – the plugin skips the check and doesn't produce any messages.

You can also configure it at task level in your `build.gradle(.kts)` file:

【Kotlin】

```
tasks.withType<org.jetbrains.kotlin.gradle.tasks.KotlinJvmCompile>().configureEach {
    jvmTargetValidationMode.set(org.jetbrains.kotlin.gradle.dsl.jvm.JvmTargetValidatio
nMode.WARNING)
}
```

【Groovy】

```
tasks.withType(org.jetbrains.kotlin.gradle.tasks.KotlinJvmCompile.class).configureEach
 {
    jvmTargetValidationMode = org.jetbrains.kotlin.gradle.dsl.jvm.JvmTargetValidationM
ode.WARNING
}
```

To avoid JVM target incompatibility, configure a toolchain or align JVM versions manually.

What can go wrong if targets are incompatible

There are two ways of manually setting JVM targets for Kotlin and Java source sets:

- The implicit way via setting up a Java toolchain.
- The explicit way via setting the `jvmTarget` attribute in the `kotlin` extension or task and

`targetCompatibility` in the `java` extension or task.

JVM target incompatibility occurs if you:

- Explicitly set different values of `jvmTarget` and `targetCompatibility`.
- Have a default configuration, and your JDK is not equal to `1.8`.

Let's consider a default configuration of JVM targets when you have only the Kotlin JVM plugin in your build script and no additional settings for JVM targets:

> 【Kotlin】

```
plugins {
    kotlin("jvm") version "2.1.21"
}
```

> 【Groovy】

```
plugins {
    id "org.jetbrains.kotlin.jvm" version "2.1.21"
}
```

When there is no explicit information about the `jvmTarget` value in the build script, its default value is `null`, and the compiler translates it to the default value `1.8`. The `targetCompatibility` equals the current Gradle's JDK version, which is equal to your JDK version (unless you use a Java toolchain approach). Assuming that your JDK version is `17`, your published library artifact will declare itself compatible with JDK 17+: `org.gradle.jvm.version=17`, which is wrong. In this case, you have to use Java 17 in your main project to add this library, even though the bytecode's version is `1.8`. Configure a toolchain to solve this issue.

Gradle Java toolchains support

> A warning for Android users. To use Gradle toolchain support, use the Android Gradle plugin (AGP) version 8.1.0-alpha09 or higher.
>
> Gradle Java toolchain support is available only from AGP 7.4.0. Nevertheless, because of this issue, AGP did not set `targetCompatibility` to be equal to the toolchain's JDK until the version 8.1.0-alpha09. If you use versions less than 8.1.0-alpha09, you need to configure

> `targetCompatibility` manually via `compileOptions`. Replace the placeholder `<MAJOR_JDK_VERSION>` with the JDK version you would like to use:
>
> ```
> android {
>     compileOptions {
>         sourceCompatibility = <MAJOR_JDK_VERSION>
>         targetCompatibility = <MAJOR_JDK_VERSION>
>     }
> }
> ```
>
> 

Gradle 6.7 introduced Java toolchains support. Using this feature, you can:

- Use a JDK and a JRE that are different from the ones in Gradle to run compilations, tests, and executables.
- Compile and test code with a not-yet-released language version.

With toolchains support, Gradle can autodetect local JDKs and install missing JDKs that Gradle requires for the build. Now Gradle itself can run on any JDK and still reuse the remote build cache feature for tasks that depend on a major JDK version.

The Kotlin Gradle plugin supports Java toolchains for Kotlin/JVM compilation tasks. JS and Native tasks don't use toolchains. The Kotlin compiler always runs on the JDK the Gradle daemon is running on. A Java toolchain:

- Sets the `-jdk-home` option available for JVM targets.
- Sets the `compilerOptions.jvmTarget` to the toolchain's JDK version if the user doesn't set the `jvmTarget` option explicitly. If the user doesn't configure the toolchain, the `jvmTarget` field uses the default value. Learn more about JVM target compatibility.
- Sets the toolchain to be used by any Java compile, test and javadoc tasks.
- Affects which JDK `kapt` workers are running on.

Use the following code to set a toolchain. Replace the placeholder `<MAJOR_JDK_VERSION>` with the JDK version you would like to use:

> 【Kotlin】

```
kotlin {
    jvmToolchain {
        languageVersion.set(JavaLanguageVersion.of(<MAJOR_JDK_VERSION>))
    }
    // Or shorter:
    jvmToolchain(<MAJOR_JDK_VERSION>)
    // For example:
```

```
    jvmToolchain(17)
}
```

【Groovy】

```
kotlin {
    jvmToolchain {
        languageVersion = JavaLanguageVersion.of(<MAJOR_JDK_VERSION>)
    }
    // Or shorter:
    jvmToolchain(<MAJOR_JDK_VERSION>)
    // For example:
    jvmToolchain(17)
}
```

Note that setting a toolchain via the `kotlin` extension updates the toolchain for Java compile tasks as well.

You can set a toolchain via the `java` extension, and Kotlin compilation tasks will use it:

【Kotlin】

```
java {
    toolchain {
        languageVersion.set(JavaLanguageVersion.of(<MAJOR_JDK_VERSION>))
    }
}
```

【Groovy】

```
java {
    toolchain {
        languageVersion = JavaLanguageVersion.of(<MAJOR_JDK_VERSION>)
    }
}
```

If you use Gradle 8.0.2 or higher, you also need to add a toolchain resolver plugin. This type of plugin manages which repositories to download a toolchain from. As an example, add to your `settings.gradle(.kts)` the following plugin:

> 【Kotlin】

```
plugins {
    id("org.gradle.toolchains.foojay-resolver-convention") version("0.9.0")
}
```

> 【Groovy】

```
plugins {
    id 'org.gradle.toolchains.foojay-resolver-convention' version '0.9.0'
}
```

Check that the version of `foojay-resolver-convention` corresponds to your Gradle version on the Gradle site.

> To understand which toolchain Gradle uses, run your Gradle build with the log level `--info` and find a string in the output starting with `[KOTLIN] Kotlin compilation 'jdkHome' argument:`. The part after the colon will be the JDK version from the toolchain.
>
> 

To set any JDK (even local) for a specific task, use the Task DSL.

Learn more about Gradle JVM toolchain support in the Kotlin plugin.

Set JDK version with the Task DSL

The Task DSL allows setting any JDK version for any task implementing the `UsesKotlinJavaToolchain` interface. At the moment, these tasks are `KotlinCompile` and `KaptTask`. If you want Gradle to search for the major JDK version, replace the `<MAJOR_JDK_VERSION>` placeholder in your build script:

> 【Kotlin】

```
val service = project.extensions.getByType<JavaToolchainService>()
val customLauncher = service.launcherFor {
    languageVersion.set(JavaLanguageVersion.of(<MAJOR_JDK_VERSION>))
}
project.tasks.withType<UsesKotlinJavaToolchain>().configureEach {
    kotlinJavaToolchain.toolchain.use(customLauncher)
}
```

> 【Groovy】

```
JavaToolchainService service = project.getExtensions().getByType(JavaToolchainService.
class)
Provider<JavaLauncher> customLauncher = service.launcherFor {
    it.languageVersion = JavaLanguageVersion.of(<MAJOR_JDK_VERSION>)
}
tasks.withType(UsesKotlinJavaToolchain::class).configureEach { task ->
    task.kotlinJavaToolchain.toolchain.use(customLauncher)
}
```

Or you can specify the path to your local JDK and replace the placeholder `<LOCAL_JDK_VERSION>` with this JDK version:

```
tasks.withType<UsesKotlinJavaToolchain>().configureEach {
    kotlinJavaToolchain.jdk.use(
        "/path/to/local/jdk", // Put a path to your JDK
        JavaVersion.<LOCAL_JDK_VERSION> // For example, JavaVersion.17
    )
}
```

Associate compiler tasks

You can *associate* compilations by setting up such a relationship between them that one compilation uses the compiled outputs of the other. Associating compilations establishes `internal` visibility between them.

The Kotlin compiler associates some compilations by default, such as the `test` and `main` compilations of each target. If you need to express that one of your custom compilations is connected to another, create your own associated compilation.

To make the IDE support associated compilations for inferring visibility between source sets, add the following code to your `build.gradle(.kts)`:

【Kotlin】

```
val integrationTestCompilation = kotlin.target.compilations.create("integrationTest")
{
    associateWith(kotlin.target.compilations.getByName("main"))
}
```

【Groovy】

```
integrationTestCompilation {
    kotlin.target.compilations.create("integrationTest") {
        associateWith(kotlin.target.compilations.getByName("main"))
    }
}
```

Here, the `integrationTest` compilation is associated with the `main` compilation that gives access to `internal` objects from functional tests.

Configure with Java Modules (JPMS) enabled

To make the Kotlin Gradle plugin work with Java Modules, add the following lines to your build script and replace `YOUR_MODULE_NAME` with a reference to your JPMS module, for example, `org.company.module`:

【Kotlin】

```
// Add the following three lines if you use a Gradle version less than 7.0
java {
    modularity.inferModulePath.set(true)
}

tasks.named("compileJava", JavaCompile::class.java) {
    options.compilerArgumentProviders.add(CommandLineArgumentProvider {
        // Provide compiled Kotlin classes to javac – needed for Java/Kotlin mixed sou
rces to work
        listOf("--patch-module", "YOUR_MODULE_NAME=${sourceSets["main"].output.asPath}"
)
    })
}
```

【Groovy】

```
// Add the following three lines if you use a Gradle version less than 7.0
java {
    modularity.inferModulePath = true
}

tasks.named("compileJava", JavaCompile.class) {
    options.compilerArgumentProviders.add(new CommandLineArgumentProvider() {
        @Override
        Iterable<String> asArguments() {
            // Provide compiled Kotlin classes to javac – needed for Java/Kotlin mixed
 sources to work
            return ["--patch-module", "YOUR_MODULE_NAME=${sourceSets["main"].output.as
Path}"]
        }
    })
}
```

> Put `module-info.java` into the `src/main/java` directory as usual.
>
> For a module, a package name in Kotlin files should be equal to the package name from `module-info.java` to avoid a "package is empty or does not exist" build failure.
>
> 

Learn more about:

- Building modules for the Java Module System
- Building applications using the Java Module System
- What "module" means in Kotlin

Other details

Learn more about Kotlin/JVM.

Disable use of artifact in compilation task

In some rare scenarios, you can experience a build failure caused by a circular dependency error. For example, when you have multiple compilations where one compilation can see all internal declarations of another, and the generated artifact relies on the output of both compilation tasks:

```
FAILURE: Build failed with an exception.

What went wrong:
Circular dependency between the following tasks:
:lib:compileKotlinJvm
--- :lib:jvmJar
     \--- :lib:compileKotlinJvm (*)
(*) - details omitted (listed previously)
```

To fix this circular dependency error, we've added a Gradle property: `archivesTaskOutputAsFriendModule`. This property controls the use of artifact inputs in the compilation task and determines if a task dependency is created as a result.

By default, this property is set to `true` to track the task dependency. If you encounter a circular dependency error, you can disable the use of the artifact in the compilation task to remove the task dependency and avoid the circular dependency error.

To disable the use of the artifact in the compilation task, add the following to your `gradle.properties` file:

```
kotlin.build.archivesTaskOutputAsFriendModule=false
```

Lazy Kotlin/JVM task creation

Starting from Kotlin 1.8.20, the Kotlin Gradle plugin registers all tasks and doesn't configure them on a dry run.

编译任务的 destinationDirectory 支持非默认位置

If you override the Kotlin/JVM `KotlinJvmCompile` / `KotlinCompile` task's `destinationDirectory` location, update your build script. You need to explicitly add `sourceSets.main.kotlin.classesDirectories` to `sourceSets.main.outputs` in your JAR file:

```
tasks.jar(type: Jar) {
    from sourceSets.main.outputs
    from sourceSets.main.kotlin.classesDirectories
}
```

Targeting multiple platforms

Projects targeting multiple platforms, called multiplatform projects, require the `kotlin-multiplatform` plugin.

> The `kotlin-multiplatform` plugin works with Gradle 7.6.3 or later.
>
> 

【Kotlin】

```
plugins {
    kotlin("multiplatform") version "2.1.21"
}
```

【Groovy】

```
plugins {
    id 'org.jetbrains.kotlin.multiplatform' version '2.1.21'
}
```

Learn more about Kotlin Multiplatform for different platforms and Kotlin Multiplatform for iOS and Android.

面向 Android

It's recommended to use Android Studio for creating Android applications. Learn how to use the Android Gradle plugin.

Targeting JavaScript

When targeting JavaScript, use the `kotlin-multiplatform` plugin as well. Learn more about setting up a Kotlin/JS project

【Kotlin】

```
plugins {
    kotlin("multiplatform") version "2.1.21"
}
```

【Groovy】

```
plugins {
    id 'org.jetbrains.kotlin.multiplatform' version '2.1.21'
}
```

Kotlin and Java sources for JavaScript

这个插件只适用于 Kotlin 文件，因此建议将 Kotlin 和 Java 文件分开（当同一项目包含 Java 文件时）。 If you don't store them separately , specify the source folder in the `sourceSets{}` block:

【Kotlin】

```
kotlin {
    sourceSets["main"].apply {
        kotlin.srcDir("src/main/myKotlin")
    }
}
```

【Groovy】

```
kotlin {
    sourceSets {
        main.kotlin.srcDirs += 'src/main/myKotlin'
    }
}
```

Triggering configuration actions with the KotlinBasePlugin interface

To trigger some configuration action whenever any Kotlin Gradle plugin (JVM, JS, Multiplatform, Native, and others) is applied, use the `KotlinBasePlugin` interface that all Kotlin plugins inherit from:

> 【Kotlin】

```
import org.jetbrains.kotlin.gradle.plugin.KotlinBasePlugin

// ...

project.plugins.withType<KotlinBasePlugin>() {
    // Configure your action here
}
```

> 【Groovy】

```
import org.jetbrains.kotlin.gradle.plugin.KotlinBasePlugin

// ...

project.plugins.withType(KotlinBasePlugin.class) {
    // Configure your action here
}
```

Configure dependencies

To add a dependency on a library, set the dependency of the required type (for example, `implementation`) in the `dependencies{}` block of the source sets DSL.

> 【Kotlin】

```
kotlin {
    sourceSets {
        commonMain.dependencies {
            implementation("com.example:my-library:1.0")
        }
    }
}
```

> 【Groovy】

```
kotlin {
```

```
    sourceSets {
        commonMain {
            dependencies {
                implementation 'com.example:my-library:1.0'
            }
        }
    }
}
```

Alternatively, you can set dependencies at top level.

Dependency types

Choose the dependency type based on your requirements.

Type	Description	When to use
`api`	Used both during compilation and at runtime and is exported to library consumers.	If any type from a dependency is used in the public API of the current module, use an `api` dependency.
`implementation`	Used during compilation and at runtime for the current module, but is not exposed for compilation of other modules depending on the one with the `implementation` dependency.	Use for dependencies needed for the internal logic of a module. If a module is an endpoint application which is not published, use `implementation` dependencies instead of `api` dependencies.
`compileOnly`	Used for compilation of the current module and is not available at runtime nor during compilation of other modules.	Use for APIs which have a third-party implementation available at runtime.
`runtimeOnly`	Available at runtime but is not visible during compilation of any module.	

对标准库的依赖

A dependency on the standard library (`stdlib`) is added automatically to each source set. The version of the standard library used is the same as the version of the Kotlin Gradle plugin.

For platform-specific source sets, the corresponding platform-specific variant of the library is used, while a common standard library is added to the rest. The Kotlin Gradle plugin selects the appropriate JVM standard library depending on the `compilerOptions.jvmTarget` compiler option of your Gradle build script.

If you declare a standard library dependency explicitly (for example, if you need a different version), the Kotlin Gradle plugin won't override it or add a second standard library.

If you don't need a standard library at all, you can add the following Gradle property to your `gradle.properties` file:

```
kotlin.stdlib.default.dependency=false
```

Versions alignment of transitive dependencies

From Kotlin standard library version 1.9.20, Gradle uses metadata included in the standard library to automatically align transitive `kotlin-stdlib-jdk7` and `kotlin-stdlib-jdk8` dependencies.

If you add a dependency for any Kotlin standard library version between 1.8.0 – 1.9.10, for example: `implementation("org.jetbrains.kotlin:kotlin-stdlib:1.8.0")`, then the Kotlin Gradle Plugin uses this Kotlin version for transitive `kotlin-stdlib-jdk7` and `kotlin-stdlib-jdk8` dependencies. This avoids class duplication from different standard library versions. Learn more about merging `kotlin-stdlib-jdk7` and `kotlin-stdlib-jdk8` into `kotlin-stdlib`. You can disable this behavior with the `kotlin.stdlib.jdk.variants.version.alignment` Gradle property in your `gradle.properties` file:

```
kotlin.stdlib.jdk.variants.version.alignment=false
```

Other ways to align versions

- If you have issues with version alignment, you can align all versions via the Kotlin BOM. Declare a platform dependency on `kotlin-bom` in your build script:

> 【Kotlin】

```
implementation(platform("org.jetbrains.kotlin:kotlin-bom:2.1.21"))
```

> 【Groovy】

```
implementation platform('org.jetbrains.kotlin:kotlin-bom:2.1.21')
```

- If you don't add a dependency for a standard library version, but you have two different dependencies that transitively bring different old versions of the Kotlin standard library, then you can explicitly require `2.1.21` versions of these transitive libraries:

> 【Kotlin】

```
dependencies {
    constraints {
        add("implementation", "org.jetbrains.kotlin:kotlin-stdlib-jdk7") {
            version {
                require("2.1.21")
            }
        }
        add("implementation", "org.jetbrains.kotlin:kotlin-stdlib-jdk8") {
            version {
                require("2.1.21")
            }
        }
    }
}
```

> 【Groovy】

```
dependencies {
    constraints {
        add("implementation", "org.jetbrains.kotlin:kotlin-stdlib-jdk7") {
            version {
                require("2.1.21")
            }
        }
        add("implementation", "org.jetbrains.kotlin:kotlin-stdlib-jdk8") {
            version {
                require("2.1.21")
            }
        }
    }
}
```

- If you add a dependency for Kotlin standard library version `2.1.21`: `implementation("org.jetbrains.kotlin:kotlin-stdlib:2.1.21")`, and an old version (earlier than `1.8.0`) of the Kotlin Gradle plugin, update the Kotlin Gradle plugin to match the standard library version:

> 【Kotlin】

```
plugins {
    // replace `<...>` with the plugin name
    kotlin("<...>") version "2.1.21"
}
```

> 【Groovy】

```
plugins {
    // replace `<...>` with the plugin name
    id "org.jetbrains.kotlin.<...>" version "2.1.21"
}
```

- If you use versions prior to `1.8.0` of `kotlin-stdlib-jdk7` / `kotlin-stdlib-jdk8`, for example, `implementation("org.jetbrains.kotlin:kotlin-stdlib-jdk7:SOME_OLD_KOTLIN_VERSION")`, and a dependency that transitively brings `kotlin-stdlib:1.8+`, replace your `kotlin-stdlib-jdk<7/8>:SOME_OLD_KOTLIN_VERSION` with `kotlin-stdlib-jdk*:2.1.21` or exclude the transitive `kotlin-stdlib:1.8+` from the library that brings it:

> 【Kotlin】

```
dependencies {
    implementation("com.example:lib:1.0") {
        exclude(group = "org.jetbrains.kotlin", module = "kotlin-stdlib")
    }
}
```

> 【Groovy】

```
    dependencies {
        implementation("com.example:lib:1.0") {
            exclude group: "org.jetbrains.kotlin", module: "kotlin-stdlib"
        }
    }
```

Set dependencies on test libraries

The `kotlin.test` API is available for testing Kotlin projects on all supported platforms. Add the `kotlin-test` dependency to the `commonTest` source set, so that the Gradle plugin can infer the corresponding test dependencies for each test source set.

Kotlin/Native targets do not require additional test dependencies, and the `kotlin.test` API implementations are built-in.

> 【Kotlin】

```
kotlin {
    sourceSets {
         commonTest.dependencies {
             implementation(kotlin("test")) // This brings all the platform dependenci
es automatically
         }
    }
}
```

> 【Groovy】

```
kotlin {
    sourceSets {
        commonTest {
            dependencies {
                implementation kotlin("test") // This brings all the platform dependen
cies automatically
            }
        }
    }
}
```

> You can use shorthand for a dependency on a Kotlin module, for example, kotlin("test") for "org.jetbrains.kotlin:kotlin-test".
>
> 

You can use the `kotlin-test` dependency in any shared or platform-specific source set as well.

JVM variants of kotlin-test

For Kotlin/JVM, Gradle uses JUnit 4 by default. Therefore, the `kotlin("test")` dependency resolves to the variant for JUnit 4, namely `kotlin-test-junit`.

You can choose JUnit 5 or TestNG by calling `useJUnitPlatform()` or `useTestNG()` in the test task of your build script. The following example is for a Kotlin Multiplatform project:

> 【Kotlin】

```
kotlin {
    jvm {
        testRuns["test"].executionTask.configure {
            useJUnitPlatform()
        }
    }
    sourceSets {
        commonTest.dependencies {
            implementation(kotlin("test"))
        }
    }
}
```

> 【Groovy】

```
kotlin {
    jvm {
        testRuns["test"].executionTask.configure {
            useJUnitPlatform()
        }
    }
    sourceSets {
        commonTest {
            dependencies {
                implementation kotlin("test")
            }
        }
    }
```

```
}
```

The following example is for a JVM project:

【Kotlin】

```
dependencies {
    testImplementation(kotlin("test"))
}

tasks {
    test {
        useTestNG()
    }
}
```

【Groovy】

```
dependencies {
    testImplementation 'org.jetbrains.kotlin:kotlin-test'
}

test {
    useTestNG()
}
```

Learn how to test code using JUnit on the JVM.

Automatic JVM variant resolution can sometimes cause problems for your configuration. In that case, you can specify the necessary framework explicitly and disable the automatic resolution by adding this line to the project `gradle.properties` file:

```
kotlin.test.infer.jvm.variant=false
```

If you have used a variant of `kotlin("test")` in your build script explicitly and your project build stopped working with a compatibility conflict, see this issue in the Compatibility guide.

Set a dependency on a kotlinx library

If you use a multiplatform library and need to depend on the shared code, set the dependency only once in the shared source set. Use the library's base artifact name, such as `kotlinx-coroutines-core` or `ktor-client-core`:

> 【Kotlin】

```
kotlin {
    sourceSets {
        commonMain.dependencies {
            implementation("org.jetbrains.kotlinx:kotlinx-coroutines-core:1.10.2")
        }
    }
}
```

> 【Groovy】

```
kotlin {
    sourceSets {
        commonMain {
            dependencies {
                implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-core:1.10.2'
            }
        }
    }
}
```

If you need a kotlinx library for a platform-specific dependency, you can still use the library's base artifact name in the corresponding platform source set:

> 【Kotlin】

```
kotlin {
    sourceSets {
        jvmMain.dependencies {
            implementation("org.jetbrains.kotlinx:kotlinx-coroutines-core:1.10.2")
        }
    }
}
```

> 【Groovy】

```
kotlin {
    sourceSets {
        jvmMain {
            dependencies {
                implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-core:1.10.2'
            }
        }
    }
}
```

Set dependencies at top level

Alternatively, you can specify the dependencies at top level, using the following pattern for the configuration names: `<sourceSetName><DependencyType>`. This can be helpful for some Gradle built-in dependencies, like `gradleApi()`, `localGroovy()`, or `gradleTestKit()`, which are not available in the source sets' dependency DSL.

> 【Kotlin】

```
dependencies {
    "commonMainImplementation"("com.example:my-library:1.0")
}
```

> 【Groovy】

```
dependencies {
    commonMainImplementation 'com.example:my-library:1.0'
}
```

Declare repositories

You can declare a publicly-available repository to use its open source dependencies. In the `repositories{}` block, set the name of the repository:

> 【Kotlin】

```
repositories {
    mavenCentral()
}
```

> 【Groovy】

```
repositories {
    mavenCentral()
}
```

Popular repositories are Maven Central and Google's Maven repository.

> If you also work with Maven projects, we recommend avoiding adding `mavenLocal()` as a repository because you may experience problems when switching between Gradle and Maven projects. If you must add the `mavenLocal()` repository, add it as the last repository in your `repositories{}` block. For more information, see The case for mavenLocal().
>
> 

If you need to declare the same repositories in more than one subproject, declare the repositories centrally in the `dependencyResolutionManagement{}` block in your `settings.gradle(.kts)` file:

> 【Kotlin】

```
dependencyResolutionManagement {
    repositories {
        mavenCentral()
    }
}
```

> 【Groovy】

```
dependencyResolutionManagement {
    repositories {
        mavenCentral()
    }
}
```

Any declared repositories in subprojects override repositories declared centrally. For more information on how to control this behavior and what options are available, see Gradle's documentation.

下一步做什么？

Learn more about:

- Compiler options and how to pass them.
- Incremental compilation, caches support, build reports, and the Kotlin daemon.
- Gradle basics and specifics.
- Support for Gradle plugin variants.

Gradle 最佳实践

Gradle is a build system used by many Kotlin projects to automate and manage the build process.

Getting the best out of Gradle is essential to help you spend less time managing and waiting for builds, and more time coding. Here we provide a set of best practices split into two key areas: **organizing** and **optimizing** your projects.

Organize

This section focuses on structuring your Gradle projects to improve clarity, maintainability, and scalability.

Use Kotlin DSL

Use Kotlin DSL instead of the traditional Groovy DSL. You avoid learning another language and gain the benefits of strict typing. Strict typing lets IDEs provide better support for refactoring and auto-completion, making development more efficient.

Find more information in Gradle's Kotlin DSL primer.

Read Gradle's blog about Kotlin DSL becoming the default for Gradle builds.

Use a version catalog

Use a version catalog in a `libs.versions.toml` file to centralize dependency management. This enables you to define and reuse versions, libraries, and plugins consistently across projects.

```
[versions]
kotlinxCoroutines = "1.10.2"

[libraries]
kotlinxCoroutines = { module = "org.jetbrains.kotlinx:kotlinx-coroutines-core", versio
n.ref = "kotlinxCoroutines" }
```

With the following dependency added to your `build.gradle.kts` file:

```
dependencies {
    implementation(libs.kotlinxCoroutines)
}
```

Learn more in Gradle's documentation about Dependency management basics.

Use convention plugins

Use convention plugins to encapsulate and reuse common build logic across multiple build files. Moving a shared configuration into a plugin helps simplify and modularize your build scripts.

Although the initial setup may be time-consuming, it's easy to maintain and add new build logic once you complete it.

Learn more in Gradle's documentation about Convention plugins.

Optimize

This section provides strategies to enhance the performance and efficiency of your Gradle builds.

Use local build cache

Use a local build cache to save time by reusing outputs produced by other builds. The build cache can retrieve outputs from any earlier build that you have already created.

Learn more in Gradle's documentation about their Build cache.

Use configuration cache

> The configuration cache doesn't support all core Gradle plugins yet. For the latest information, see Gradle's table of supported plugins.
>
> 

Use the configuration cache to significantly improve build performance by caching the result of the configuration phase and reusing it for subsequent builds. If Gradle detects no changes in the build configuration or related dependencies, it skips the configuration phase.

Learn more in Gradle's documentation about their Configuration cache.

Improve build times for multiple targets

When your multiplatform project includes multiple targets, tasks like `build` and `assemble` can compile the same code multiple times for each target, leading to longer compilation times.

If you're actively developing and testing a specific platform, run the corresponding `linkDebug*` task instead.

For more information, see Tips for improving compilation time.

Migrate from kapt to KSP

If you're using a library that relies on the kapt compiler plugin, check whether you can switch to using the Kotlin Symbol Processing (KSP) API instead. The KSP API improves build performance by reducing annotation processing time. KSP is faster and more efficient than kapt, as it processes source code directly without generating intermediary Java stubs.

For guidance on the migration steps, see Google's migration guide.

To learn more about how KSP compares to kapt, check out why KSP.

Use modularization

> Modularization only benefits projects of moderate to large size. It doesn't provide advantages for projects based on a microservices architecture.
>
> 

Use a modularized project structure to increase build speed and enable easier parallel development. Structure your project into one root project and one or more subprojects. If changes only affect one of the subprojects, Gradle rebuilds only that specific subproject.

```
.
└── root-project/
    ├── settings.gradle.kts
    ├── app subproject/
    │   └── build.gradle.kts
    └── lib subproject/
        └── build.gradle.kts
```

Learn more in Gradle's documentation about Structuring projects with Gradle.

Set up CI/CD

Set up a CI/CD process to significantly reduce build time by using incremental builds and caching dependencies. Add persistent storage or use a remote build cache to see these benefits. This process doesn't have to be time-consuming, as some providers, like GitHub, offer this service almost out of the box.

Explore Gradle's community cookbook on Using Gradle with Continuous Integration systems.

Use remote build cache

Like the local build cache, the remote build cache helps you save time by reusing outputs from other builds. It can retrieve task outputs from any earlier build that anyone has already run, not just the last one.

The remote build cache uses a cache server to share task outputs across builds. For example, in a development environment with a CI/CD server, all builds on the server populate the remote cache. When you check out the main branch to start a new feature, you can immediately access incremental builds.

Keep in mind that a slow internet connection might make transferring cached results slower than running tasks locally.

Learn more in Gradle's documentation about their Build cache.

Kotlin Gradle 插件中的编译器选项

每个 Kotlin 的发行版本都包含对于以下目标代码的编译器： JVM、JavaScript，以及支持平台的本地二进制文件。

这些编译器会在这些步骤中被调用：

- IDE，当你在你的 Kotlin 项目上点击编译或运行按钮时。
- Gradle，当你在控制台或者 IDE 上调用 `gradle build` 命令时。
- Maven，当你在控制台或者 IDE 上调用 `mvn compile` 或者 `mvn test-compile` 命令时。

你也可以用命令行手动调用 Kotlin 编译器， 详见使用命令行编译器教程。

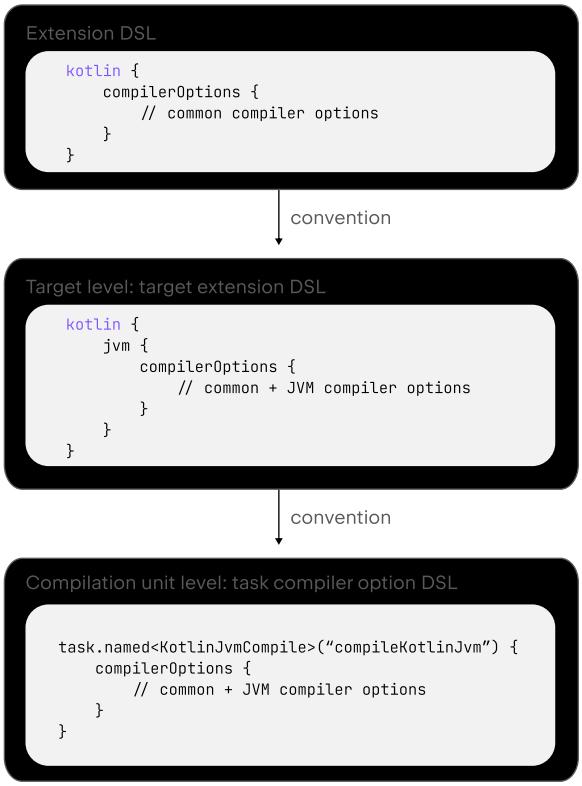
配置选项

Kotlin 编译器有一系列用于控制编译过程的参数。

The Gradle DSL allows comprehensive configuration of compiler options. It is available for Kotlin Multiplatform and JVM/Android projects.

With the Gradle DSL, you can configure compiler options within the build script at three levels:

- **Extension level**, in the `kotlin {}` block for all targets and shared source sets.
- **Target level**, in the block for a specific target.
- **Compilation unit level**, usually in a specific compilation task.



The settings at a higher level are used as a convention (default) for a lower level:

- Compiler options set at the extension level are the default for target-level options, including shared source sets like `commonMain`, `nativeMain`, and `commonTest`.
- Compiler options set at the target level are the default for options at the compilation unit (task) level, like `compileKotlinJvm` and `compileTestKotlinJvm` tasks.

In turn, configurations made at a lower level override related settings at a higher level:

- Task-level compiler options override related configurations at the target or the extension level.
- Target-level compiler options override related configurations at the extension level.

To find out which level of compiler arguments is applied to the compilation, use the `DEBUG` level of Gradle logging. For JVM and JS/WASM tasks, search for the `"Kotlin compiler args:"` string within the logs; for Native tasks, search for the `"Arguments ="` string.

> If you're a third-party plugin author, it's best to apply your configuration on the project level to avoid overriding issues. You can use the new Kotlin plugin DSL extension types for this. It's recommended that you document this configuration on your side explicitly.

Extension level

You can configure common compiler options for all the targets and shared source sets in the `compilerOptions {}` block at the top level:

```
kotlin {
    compilerOptions {
        optIn.add("kotlin.RequiresOptIn")
    }
}
```

Target level

You can configure compiler options for the JVM/Android target in the `compilerOptions {}` block inside the `target {}` block:

```
kotlin {
    target {
        compilerOptions {
            optIn.add("kotlin.RequiresOptIn")
        }
    }
}
```

In Kotlin Multiplatform projects, you can configure compiler options inside the specific target. For example, `jvm { compilerOptions {}}`. For more information, see Multiplatform Gradle DSL reference.

Compilation unit level

You can configure compiler options for a specific compilation unit or task in a `compilerOptions {}` block inside the task configuration:

```
tasks.named<KotlinJvmCompile>("compileKotlin"){
    compilerOptions {
        optIn.add("kotlin.RequiresOptIn")
    }
}
```

You can also access and configure compiler options at a compilation unit level via `KotlinCompilation`:

```
kotlin {
    target {
        val main by compilations.getting {
            compileTaskProvider.configure {
```

```
                compilerOptions {

                }
            }
        }
    }
}
```

If you want to configure a plugin of a target different from JVM/Android and Kotlin Multiplatform, use the `compilerOptions {}` property of the corresponding Kotlin compilation task. The following examples show how to set this configuration up in both Kotlin and Groovy DSLs:

> 【Kotlin】

```
tasks.named("compileKotlin", org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTask::
class.java) {
    compilerOptions {
        apiVersion.set(org.jetbrains.kotlin.gradle.dsl.KotlinVersion.KOTLIN_2_0)
    }
}
```

> 【Groovy】

```
tasks.named('compileKotlin', org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTask.c
lass) {
    compilerOptions {
        apiVersion.set(org.jetbrains.kotlin.gradle.dsl.KotlinVersion.KOTLIN_2_0)
    }
}
```

面向 JVM

As explained before, you can define compiler options for your JVM/Android projects at the extension, target, and compilation unit levels (tasks).

Default JVM 环境下的编译任务，对于生产代码叫做 `compileKotlin`，而对于测试代码则叫做 `compileTestKotlin`。对于自定义源代码集（source set），这些任务命名遵循 `compile<Name>Kotlin` 模式。

You can see the list of Android compilation tasks by running the `gradlew tasks --all` command in the terminal and searching for `compile*Kotlin` task names in the `Other tasks` group.

有几点是需要注意的：

- `android.kotlinOptions` 和 `kotlin.compilerOptions` 这两个配置会相互覆写。只有最新（即最下面）的一个会生效。
- `kotlin.compilerOptions` 会影响项目中所有 Kotlin 编译任务的配置。
- 对于那些应用到 `kotlin.compilerOptions` 的任务，你可以通过 `tasks.named<KotlinJvmCompile>("compileKotlin") { }`（或者 `tasks.withType<KotlinJvmCompile>().configureEach { }`）来覆写配置。

面向 JavaScript

JavaScript 的编译任务，对于生产代码叫做 `compileKotlinJs`，对于测试代码叫做 `compileTestKotlinJs`。 对于自定义源代码集（source set），这些任务命名遵循 `compile<Name>KotlinJs` 模式。

要配置单个任务，请使用其名称：

> 【Kotlin】

```
import org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTask
// ……

val compileKotlin: KotlinCompilationTask<*> by tasks

compileKotlin.compilerOptions.suppressWarnings.set(true)
```

> 【Groovy】

```
import org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTask
// ……

tasks.named('compileKotlin', KotlinCompilationTask) {
    compilerOptions {
        suppressWarnings = true
    }
}
```

请注意，对于 Gradle Kotlin DSL，首先从项目的 `tasks` 中获取任务。

相应地，为 JS 与公共目标使用类型 `Kotlin2JsCompile` 与 `KotlinCompileCommon`。

You can see the list of JavaScript compilation tasks by running the `gradlew tasks --all` command in the terminal and searching for `compile*KotlinJS` task names in the `Other tasks` group.

All Kotlin compilation tasks

你也可以在项目中对所有的 Kotlin 编译任务进行配置：

> 【Kotlin】

```
import org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTask
// ……

tasks.named<KotlinCompilationTask<*>>("compileKotlin").configure {
    compilerOptions { /*……*/ }
}
```

> 【Groovy】

```
import org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTask
// ...

tasks.named('compileKotlin', KotlinCompilationTask) {
    compilerOptions { /*……*/ }
}
```

所有编译器选项

这里列出了 Gradle 编译器的完整选项列表：

常规属性

名称	描述	可能的值	默认值
`optIn`	用于配置选择加入的编译器参数的列表	`listOf( /* opt-ins */ )`	`emptyList()`

`progressiveMode`	启用渐进式编译器模式	`true`，`false`	`false`
`extraWarnings`	Enables additional declaration, expression, and type compiler checks that emit warnings if true	`true`，`false`	`false`

JVM 特有的属性

名称	描述	可能的值	默认值
`javaParameters`	为方法参数生成 Java 1.8 反射的元数据		false
`jvmTarget`	生成的 JVM 字节码的目标版本	"1.8"、 "9"、 "10"、 ……、 "22"、 "23"。 另见编译器选项的类型	"1.8"
`noJdk`	不要自动在类路径中包含 Java 运行时		false
`jvmTargetValidationMode`	• 验证 Kotlin 和 Java 之间 [JVM 目标的兼容性] (gradle-configure-project.md#check-for-jvm-target-compatibility-of-related-compile-tasks) • 该编译器选项属于 `KotlinCompile` 任务	`WARNING`，`ERROR`，`IGNORE`	`ERROR`

JVM 与 JavaScript 的公共属性

名称	描述	可能的值	默认值
`allWarningsAsErrors`	任何警告都报告为错误		false
`suppressWarnings`	不生成警告		false
`verbose`	启用详细日志输出。仅在已启用 Gradle debug 日志时才有效		false

`freeCompilerArgs`	附加编译器参数的列表。你也可以在这里使用实验性的 `-X` 参数。见例。		[]		`apiVersion`
`apiVersion`	Restrict the use of declarations to those from the specified version of bundled libraries	"1.8", "1.9", "2.0", "2.1", "2.2" (EXPERIMENTAL)			
`languageVersion`	Provide source compatibility with the specified version of Kotlin	"1.8", "1.9", "2.0", "2.1", "2.2" (EXPERIMENTAL)			

> 我们计划在今后的发行版本中将 `freeCompilerArgs` 选项弃用。如果你因此无法在 Kotlin Gradle DSL 中配置某些选项 请提出一个 Issue。
>
> 

通过 freeCompilerArgs 选项配置额外的参数的示例

通过使用 `freeCompilerArgs` 属性来应用包括实验性参数在内的额外编译器参数。 可以在这个属性中加入一个单独的编译器参数或者一个编译器参数的列表。

> 【Kotlin】

```
import org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTask
// ...

kotlin {
    compilerOptions {
        // Specifies the version of the Kotlin API and the JVM target
        apiVersion.set(KotlinVersion.KOTLIN_2_1)
        jvmTarget.set(JvmTarget.JVM_1_8)

        // 单个实验性参数
        freeCompilerArgs.add("-Xexport-kdoc")
```

```
        // 单个附加参数
        freeCompilerArgs.add("-Xno-param-assertions")

        // 参数列表
        freeCompilerArgs.addAll(
            listOf(
                "-Xno-receiver-assertions",
                "-Xno-call-assertions"
            )
        )
    }
}
```

> 【Groovy】

```
import org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTask
// ...

tasks.named('compileKotlin', KotlinCompilationTask) {
    compilerOptions {
        // Specifies the version of the Kotlin API and the JVM target
        apiVersion = KotlinVersion.KOTLIN_2_1
        jvmTarget = JvmTarget.JVM_1_8

        // 单个实验性参数
        freeCompilerArgs.add("-Xexport-kdoc")

        // 单个附加参数，可以是键值对
        freeCompilerArgs.add("-Xno-param-assertions")

        // 参数列表
        freeCompilerArgs.addAll(["-Xno-receiver-assertions", "-Xno-call-assertions"])
    }
}
```

> The `freeCompilerArgs` attribute is available at the extension, target, and compilation unit (task) levels.
>
> 

设置 languageVersion 的示例

要设置语言的版本，请使用如下格式的语法：

> 【Kotlin】

```
kotlin {
    compilerOptions {
        languageVersion.set(org.jetbrains.kotlin.gradle.dsl.KotlinVersion.KOTLIN_2_1)
    }
}
```

> 【Groovy】

```
tasks
    .withType(org.jetbrains.kotlin.gradle.tasks.KotlinCompilationTask.class)
    .configureEach {
        compilerOptions.languageVersion =
            org.jetbrains.kotlin.gradle.dsl.KotlinVersion.KOTLIN_2_1
    }
```

另见编译器选项的类型。

JavaScript 特有的属性

名称	描述	可能的值
`friendModulesDisabled`	禁用内部声明导出	
`main`	指定是否在执行时调用 `main` 函数	`JsMainFunctionExecutionMode.CALL`，`JsMainFunctionExecutionMode.NO_CALL`

`moduleKind`	编译器生成的 JS 模块类型	`JsModuleKind.MODULE_AMD`，`JsModuleKind.MODULE_PLA`
		`JsModuleKind.MODULE_ES`，`JsModuleKind.MODULE_COMM`
		`JsModuleKind.MODULE_UMD`

`sourceMap`	生成源代码映射（source map）	

`sourceMapEmbedSources`	将源代码嵌入到源代码映射中	`JsSourceMapEmbedMode.SOURCE_MAP_SOURCE_CONTENT_IN`
		`JsSourceMapEmbedMode.SOURCE_MAP_SOURCE_CONTENT_NE`
		`JsSourceMapEmbedMode.SOURCE_MAP_SOURCE_CONTENT_AL`

	将 Kotlin	

`sourceMapNamesPolicy`	代码中声明的变量和函数添加到源代码映射中。详见编译器引用。	`JsSourceMapNamesPolicy.SOURCE_MAP_NAMES_POLICY_FU`
		`JsSourceMapNamesPolicy.SOURCE_MAP_NAMES_POLICY_SI`
		`JsSourceMapNamesPolicy.SOURCE_MAP_NAMES_POLICY_NO`

`sourceMapPrefix`	将指定前缀添加到源代码映射中的路径	
`target`	生成指定 ECMA 版本的 JS 文件	`"es5"` , `"es2015"`
`useEsClasses`	Let generated JavaScript code use ES2015 classes. Enabled by default in case of ES2015 target usage	

编译器选项的类型

一些 `compilerOptions` 使用独有的类型来替代字符串类型

选项	类型	
`jvmTarget`	`JvmTarget`	`compilerOptions.jvmTarget.set`
`apiVersion` and `languageVersion`	`KotlinVersion`	`compilerOptions.languageVersi`
`main`	`JsMainFunctionExecutionMode`	`compilerOptions.main.set(JsMa`
`moduleKind`	`JsModuleKind`	`compilerOptions.moduleKind.se`
`sourceMapEmbedSources`	`JsSourceMapEmbedMode`	`compilerOptions.sourceMapEmbe`
`sourceMapNamesPolicy`	`JsSourceMapNamesPolicy`	`compilerOptions.sourceMapName`

下一步做什么？

了解更多关于：

- Kotlin Multiplatform DSL reference.
- 增量编译、缓存支持、构建日志以及 Kotlin 守护进程。
- Gradle 的基础知识和特性。
- 对 Gradle 插件变体的支持。

Kotlin Gradle 插件中的编译项与缓存

On this page, you can learn about the following topics:

- 增量编译
- Gradle 构建缓存支持
- Gradle configuration cache support
- The Kotlin daemon and how to use it with Gradle
- Rolling back to the previous compiler
- Defining Kotlin compiler execution strategy
- Kotlin compiler fallback strategy
- Trying the latest language version
- Build reports

增量编译

Kotlin Gradle 插件支持增量编译，which is enabled by default for Kotlin/JVM and Kotlin/JS projects. 增量编译会跟踪多次构建之间类路径中文件的变更，以便只编译这些变更所影响的文件。 This approach works with Gradle's build cache and supports compilation avoidance.

For Kotlin/JVM, incremental compilation relies on classpath snapshots, which capture the API structure of modules to determine when recompilation is necessary. To optimize the overall pipeline, the Kotlin compiler uses two types of classpath snapshots:

- **Fine-grained snapshots:** include detailed information about class members, such as properties or functions. When member-level changes are detected, the Kotlin compiler recompiles only the classes that depend on the modified members. To maintain performance, the Kotlin Gradle plugin creates coarse-grained snapshots for `.jar` files in the Gradle cache.
- **Coarse-grained snapshots:** only contain the class ABI hash. When a part of ABI changes, the Kotlin compiler recompiles all classes that depend on the changed class. This is useful for classes that change infrequently, such as external libraries.

> Kotlin/JS projects use a different incremental compilation approach based on history files.
>
> 

有几种方法可以禁增量编译：

- 对于 Kotlin/JVM 设置 `kotlin.incremental=false` 。
- 对于 Kotlin/JS projects 设置 `kotlin.incremental.js=false` 。

- Use `-Pkotlin.incremental=false` or `-Pkotlin.incremental.js=false` as a command line parameter.

  该参数必须添加到后续每个子构建。

当禁用增量编译时，增量缓存会在该构建后失效。首次构建决不会是增量的。

> Sometimes problems with incremental compilation become visible several rounds after the failure occurs. Use build reports to track the history of changes and compilations. This can help you to provide reproducible bug reports.
>
> 

To learn more about how our current incremental compilation approach works and compares to the previous one, see our blog post.

Gradle 构建缓存支持

The Kotlin plugin uses the Gradle build cache, which stores the build outputs for reuse in future builds.

如需禁用所有 Kotlin 任务的缓存，请将系统属性 `kotlin.caching.enabled` 设置为 `false`（运行构建带上参数 `-Dkotlin.caching.enabled=false`）。

Gradle configuration cache support

The Kotlin plugin uses the Gradle configuration cache, which speeds up the build process by reusing the results of the configuration phase for subsequent builds.

See the Gradle documentation to learn how to enable the configuration cache. After you enable this feature, the Kotlin Gradle plugin automatically starts using it.

The Kotlin daemon and how to use it with Gradle

The Kotlin daemon:

- Runs with the Gradle daemon to compile the project.
- Runs separately from the Gradle daemon when you compile the project with an IntelliJ IDEA built-in build system.

The Kotlin daemon starts at the Gradle execution stage when one of the Kotlin compile tasks starts to compile sources. The Kotlin daemon stops either with the Gradle daemon or after two idle hours with no Kotlin compilation.

The Kotlin daemon uses the same JDK that the Gradle daemon does.

Setting Kotlin daemon's JVM arguments

Each of the following ways to set arguments overrides the ones that came before it:

- Gradle daemon arguments inheritance
- `kotlin.daemon.jvm.options` system property
- `kotlin.daemon.jvmargs` property
- `kotlin` extension
- Specific task definition

Gradle daemon arguments inheritance

By default, the Kotlin daemon inherits a specific set of arguments from the Gradle daemon but overwrites them with any JVM arguments specified directly for the Kotlin daemon. For example, if you add the following JVM arguments in the `gradle.properties` file:

```
org.gradle.jvmargs=-Xmx1500m -Xms500m -XX:MaxMetaspaceSize=1g
```

These arguments are then added to the Kotlin daemon's JVM arguments:

```
-Xmx1500m -XX:ReservedCodeCacheSize=320m -XX:MaxMetaspaceSize=1g -XX:UseParallelGC -ea
 -XX:+UseCodeCacheFlushing -XX:+HeapDumpOnOutOfMemoryError -Djava.awt.headless=true -D
java.rmi.server.hostname=127.0.0.1 --add-exports=java.base/sun.nio.ch=ALL-UNNAMED
```

> To learn more about the Kotlin daemon's default behavior with JVM arguments, see Kotlin daemon's behavior with JVM arguments.
>
> 

kotlin.daemon.jvm.options system property

If the Gradle daemon's JVM arguments have the `kotlin.daemon.jvm.options` system property – use it in the `gradle.properties` file:

```
org.gradle.jvmargs=-Dkotlin.daemon.jvm.options=-Xmx1500m,Xms500m
```

When passing arguments, follow these rules:

- Use the minus sign `-` **only** before the arguments `Xmx`, `XX:MaxMetaspaceSize`, and `XX:ReservedCodeCacheSize`.
- Separate arguments with commas (`,`) *without* spaces. Arguments that come after a space will be used for the Gradle daemon, not for the Kotlin daemon.

> Gradle ignores these properties if all the following conditions are satisfied:
>
> - Gradle is using JDK 1.9 or higher.
> - The version of Gradle is between 7.0 and 7.1.1 inclusively.
> - Gradle is compiling Kotlin DSL scripts.
> - The Kotlin daemon isn't running.
>
> To overcome this, upgrade Gradle to the version 7.2 (or higher) or use the `kotlin.daemon.jvmargs` property – see the following section.



kotlin.daemon.jvmargs property

You can add the `kotlin.daemon.jvmargs` property in the `gradle.properties` file:

```
kotlin.daemon.jvmargs=-Xmx1500m -Xms500m
```

Note that if you don't specify the `ReservedCodeCacheSize` argument here or in Gradle's JVM arguments, the Kotlin Gradle plugin applies a default value of `320m`:

```
-Xmx1500m -XX:ReservedCodeCacheSize=320m -Xms500m
```

kotlin extension

You can specify arguments in the `kotlin` extension:

> **【Kotlin】**

```
kotlin {
    kotlinDaemonJvmArgs = listOf("-Xmx486m", "-Xms256m", "-XX:+UseParallelGC")
}
```

【Groovy】

```
kotlin {
    kotlinDaemonJvmArgs = ["-Xmx486m", "-Xms256m", "-XX:+UseParallelGC"]
}
```

Specific task definition

You can specify arguments for a specific task:

【Kotlin】

```
tasks.withType<CompileUsingKotlinDaemon>().configureEach {
    kotlinDaemonJvmArguments.set(listOf("-Xmx486m", "-Xms256m", "-XX:+UseParallelGC"))
}
```

【Groovy】

```
tasks.withType(CompileUsingKotlinDaemon).configureEach { task ->
    task.kotlinDaemonJvmArguments = ["-Xmx1g", "-Xms512m"]
}
```

> In this case a new Kotlin daemon instance can start on task execution. Learn more about Kotlin daemon's behavior with JVM arguments.
>
> 

Kotlin daemon's behavior with JVM arguments

When configuring the Kotlin daemon's JVM arguments, note that:

- It is expected to have multiple instances of the Kotlin daemon running at the same time when different subprojects or tasks have different sets of JVM arguments.
- A new Kotlin daemon instance starts only when Gradle runs a related compilation task and existing Kotlin daemons do not have the same set of JVM arguments. Imagine that your project has a lot of subprojects. Most of them require some heap memory for a Kotlin daemon, but one module requires a lot (though it is rarely compiled). In this case, you should provide a different

set of JVM arguments for such a module, so a Kotlin daemon with a larger heap size would start only for developers who touch this specific module.

> If you are already running a Kotlin daemon that has enough heap size to handle the compilation request, even if other requested JVM arguments are different, this daemon will be reused instead of starting a new one.
>
> 

If the following arguments aren't specified, the Kotlin daemon inherits them from the Gradle daemon:

- `-Xmx`
- `-XX:MaxMetaspaceSize`
- `-XX:ReservedCodeCacheSize`. If not specified or inherited, the default value is `320m`.

The Kotlin daemon has the following default JVM arguments:

- `-XX:UseParallelGC`. This argument is only applied if no other garbage collector is specified.
- `-ea`
- `-XX:+UseCodeCacheFlushing`
- `-Djava.awt.headless=true`
- `-D{java.servername.property}={localhostip}`
- `--add-exports=java.base/sun.nio.ch=ALL-UNNAMED`. This argument is only applied for JDK versions 16 or higher.

> The list of default JVM arguments for the Kotlin daemon may vary between versions. You can use a tool like VisualVM to check the actual settings of a running JVM process, like the Kotlin daemon.
>
> 

Rolling back to the previous compiler

From Kotlin 2.0.0, the K2 compiler is used by default.

To use the previous compiler from Kotlin 2.0.0 onwards, either:

- In your `build.gradle.kts` file, set your language version to `1.9`.

  OR

- Use the following compiler option: `-language-version 1.9`.

To learn more about the benefits of the K2 compiler, see the K2 compiler migration guide.

Defining Kotlin compiler execution strategy

Kotlin compiler execution strategy defines where the Kotlin compiler is executed and if incremental compilation is supported in each case.

There are three compiler execution strategies:

Strategy	Where Kotlin compiler is executed	Incremental compilation	Other characteristics and notes
Daemon	Inside its own daemon process	Yes	*The default and fastest strategy*. Can be shared between different Gradle daemons and multiple parallel compilations.
In process	Inside the Gradle daemon process	No	May share the heap with the Gradle daemon. The "In process" execution strategy is *slower* than the "Daemon" execution strategy. Each worker creates a separate Kotlin compiler classloader for each compilation.
Out of process	In a separate process for each compilation	No	The slowest execution strategy. Similar to the "In process", but additionally creates a separate Java process within a Gradle worker for each compilation.

To define a Kotlin compiler execution strategy, you can use one of the following properties:

- The `kotlin.compiler.execution.strategy` Gradle property.
- The `compilerExecutionStrategy` compile task property.

The task property `compilerExecutionStrategy` takes priority over the Gradle property `kotlin.compiler.execution.strategy`.

The available values for the `kotlin.compiler.execution.strategy` property are:

1. `daemon` (default)
2. `in-process`
3. `out-of-process`

Use the Gradle property `kotlin.compiler.execution.strategy` in `gradle.properties`:

```
kotlin.compiler.execution.strategy=out-of-process
```

The available values for the `compilerExecutionStrategy` task property are:

1. `org.jetbrains.kotlin.gradle.tasks.KotlinCompilerExecutionStrategy.DAEMON` (default)
2. `org.jetbrains.kotlin.gradle.tasks.KotlinCompilerExecutionStrategy.IN_PROCESS`
3. `org.jetbrains.kotlin.gradle.tasks.KotlinCompilerExecutionStrategy.OUT_OF_PROCESS`

Use the task property `compilerExecutionStrategy` in your build scripts:

> 【Kotlin】

```
import org.jetbrains.kotlin.gradle.tasks.CompileUsingKotlinDaemon
import org.jetbrains.kotlin.gradle.tasks.KotlinCompilerExecutionStrategy

// ...

tasks.withType<CompileUsingKotlinDaemon>().configureEach {
    compilerExecutionStrategy.set(KotlinCompilerExecutionStrategy.IN_PROCESS)
}
```

> 【Groovy】

```
import org.jetbrains.kotlin.gradle.tasks.CompileUsingKotlinDaemon
import org.jetbrains.kotlin.gradle.tasks.KotlinCompilerExecutionStrategy

// ...

tasks.withType(CompileUsingKotlinDaemon)
    .configureEach {
        compilerExecutionStrategy = KotlinCompilerExecutionStrategy.IN_PROCESS
    }
```

Kotlin compiler fallback strategy

The Kotlin compiler's fallback strategy is to run a compilation outside a Kotlin daemon if the daemon somehow fails. If the Gradle daemon is on, the compiler uses the "In process" strategy. If the Gradle daemon is off, the compiler uses the "Out of process" strategy.

When this fallback happens, you have the following warning lines in your Gradle's build output:

```
Failed to compile with Kotlin daemon: java.lang.RuntimeException: Could not connect to
 Kotlin compile daemon
[exception stacktrace]
Using fallback strategy: Compile without Kotlin daemon
Try ./gradlew --stop if this issue persists.
```

However, a silent fallback to another strategy can consume a lot of system resources or lead to non-deterministic builds. Read more about this in this YouTrack issue. To avoid this, there is a Gradle property `kotlin.daemon.useFallbackStrategy`, whose default value is `true`. When the value is `false`, builds fail on problems with the daemon's startup or communication. Declare this property in `gradle.properties`:

```
kotlin.daemon.useFallbackStrategy=false
```

There is also a `useDaemonFallbackStrategy` property in Kotlin compile tasks, which takes priority over the Gradle property if you use both.

> 【Kotlin】

```
tasks {
    compileKotlin {
        useDaemonFallbackStrategy.set(false)
    }
}
```

> 【Groovy】

```
tasks.named("compileKotlin").configure {
    useDaemonFallbackStrategy = false
}
```

If there is insufficient memory to run the compilation, you can see a message about it in the logs.

Trying the latest language version

Starting with Kotlin 2.0.0, to try the latest language version, set the `kotlin.experimental.tryNext` property in your `gradle.properties` file. When you use this property, the Kotlin Gradle plugin increments the language version to one above the default value for your Kotlin version. For example, in Kotlin 2.0.0, the default language version is 2.0, so the property configures language version 2.1.

Alternatively, you can run the following command:

```
./gradlew assemble -Pkotlin.experimental.tryNext=true
```

In build reports, you can find the language version used to compile each task.

Build reports

Build reports contain the durations of different compilation phases and any reasons why compilation couldn't be incremental. Use build reports to investigate performance issues when the compilation time is too long or when it differs for the same project.

Kotlin build reports help you to investigate problems with build performance more efficiently than with Gradle build scans that have a single Gradle task as the unit of granularity.

There are two common cases that analyzing build reports for long-running compilations can help you resolve:

- The build wasn't incremental. Analyze the reasons and fix underlying problems.
- The build was incremental but took too much time. Try reorganizing source files — split big files, save separate classes in different files, refactor large classes, declare top-level functions in different files, and so on.

Build reports also show the Kotlin version used in the project. In addition, starting with Kotlin 1.9.0, you can see which compiler was used to compile the code in your Gradle build scans.

Learn how to read build reports and about how JetBrains uses build reports.

Enabling build reports

To enable build reports, declare where to save the build report output in `gradle.properties`:

```
kotlin.build.report.output=file
```

The following values and their combinations are available for the output:

Option	Description
`file`	Saves build reports in a human-readable format to a local file. By default, it's `${project_folder}/build/reports/kotlin-build/${project_name}-timestamp.txt`
`single_file`	Saves build reports in a format of an object to a specified local file.
`build_scan`	Saves build reports in the `custom values` section of the build scan. Note that the Gradle Enterprise plugin limits the number of custom values and their length. In big projects, some values could be lost.
`http`	Posts build reports using HTTP(S). The POST method sends metrics in JSON format. You can see the current version of the sent data in the Kotlin

	repository. You can find samples of HTTP endpoints in this blog post
`json`	Saves build reports in JSON format to a local file. Set the location for your build reports in `kotlin.build.report.json.directory` (see below). By default, it's name is `${project_name}-build-<date-time>-<index>.json` .

Here's a list of available options for `kotlin.build.report` :

```
# Required outputs. Any combination is allowed
kotlin.build.report.output=file,single_file,http,build_scan,json

# Mandatory if single_file output is used. Where to put reports
# Use instead of the deprecated `kotlin.internal.single.build.metrics.file` property
kotlin.build.report.single_file=some_filename

# Mandatory if json output is used. Where to put reports
kotlin.build.report.json.directory=my/directory/path

# Optional. Output directory for file-based reports. Default: build/reports/kotlin-bui
ld/
kotlin.build.report.file.output_dir=kotlin-reports

# Optional. Label for marking your build report (for example, debug parameters)
kotlin.build.report.label=some_label
```

Options, applicable only to HTTP:

```
# Mandatory. Where to post HTTP(S)-based reports
kotlin.build.report.http.url=http://127.0.0.1:8080

# Optional. User and password if the HTTP endpoint requires authentication
kotlin.build.report.http.user=someUser
kotlin.build.report.http.password=somePassword

# Optional. Add a Git branch name of a build to a build report
kotlin.build.report.http.include_git_branch.name=true|false

# Optional. Add compiler arguments to a build report
# If a project contains many modules, its compiler arguments in the report can be very
 heavy and not that helpful
kotlin.build.report.include_compiler_arguments=true|false
```

Limit of custom values

To collect build scans' statistics, Kotlin build reports use Gradle's custom values. Both you and different Gradle plugins can write data to custom values. The number of custom values has a limit. See the current maximum custom value count in the Build scan plugin docs.

If you have a big project, a number of such custom values may be quite big. If this number exceeds the limit, you can see the following message in the logs:

```
Maximum number of custom values (1,000) exceeded
```

To reduce the number of custom values the Kotlin plugin produces, you can use the following property in `gradle.properties`:

```
kotlin.build.report.build_scan.custom_values_limit=500
```

Switching off collecting project and system properties

HTTP build statistic logs can contain some project and system properties. These properties can change builds' behavior, so it's useful to log them in build statistics. These properties can store sensitive data, for example, passwords or a project's full path.

You can disable collection of these statistics by adding the `kotlin.build.report.http.verbose_environment` property to your `gradle.properties`.

> JetBrains doesn't collect these statistics. You choose a place where to store your reports.
>
> 

下一步做什么？

Learn more about:

- Gradle basics and specifics.
- Support for Gradle plugin variants.

支持 Gradle 插件变体

Gradle 7.0 introduced a new feature for Gradle plugin authors — plugins with variants. This feature makes it easier to add support for latest Gradle features while maintaining compatibility with older Gradle versions. Learn more about variant selection in Gradle.

With Gradle plugin variants, the Kotlin team can ship different Kotlin Gradle plugin (KGP) variants for different Gradle versions. The goal is to support the base Kotlin compilation in the `main` variant, which corresponds to the oldest supported versions of Gradle. Each variant will have implementations for Gradle features from a corresponding release. The latest variant will support the latest Gradle feature set. With this approach, it is possible to extend support for older Gradle versions with limited functionality.

Currently, there are the following variants of the Kotlin Gradle plugin:

Variant's name	Corresponding Gradle versions
`main`	7.6.0–7.6.3
`gradle80`	8.0–8.0.2
`gradle81`	8.1.1
`gradle82`	8.2.1–8.4
`gradle85`	8.5 and higher

In future Kotlin releases, more variants will be added.

To check which variant your build uses, enable the `--info` log level and find a string in the output starting with `Using Kotlin Gradle plugin` , for example, `Using Kotlin Gradle plugin main variant` .

问题排查

> Here are workarounds for some known issues with variant selection in Gradle:
>
> - ResolutionStrategy in pluginManagement is not working for plugins with multivariants
> - Plugin variants are ignored when a plugin is added as the `buildSrc` common dependency

Gradle 无法在自定义配置中选择 KGP 变体

This is an expected situation that Gradle can't select a KGP variant in a custom configuration. If you use a custom Gradle configuration:

> 【Kotlin】

```
configurations.register("customConfiguration") {
    // ...
}
```

> 【Groovy】

```
configurations.register("customConfiguration") {
    // ...
}
```

And want to add a dependency on the Kotlin Gradle plugin, for example:

> 【Kotlin】

```
dependencies {
    customConfiguration("org.jetbrains.kotlin:kotlin-gradle-plugin:2.1.21")
}
```

> 【Groovy】

```
dependencies {
    customConfiguration 'org.jetbrains.kotlin:kotlin-gradle-plugin:2.1.21'
}
```

You need to add the following attributes to your `customConfiguration`:

> 【Kotlin】

```
configurations {
    customConfiguration {
        attributes {
            attribute(
                Usage.USAGE_ATTRIBUTE,
                project.objects.named(Usage.class, Usage.JAVA_RUNTIME)
            )
            attribute(
                Category.CATEGORY_ATTRIBUTE,
                project.objects.named(Category.class, Category.LIBRARY)
            )
            // If you want to depend on a specific KGP variant:
            attribute(
                GradlePluginApiVersion.GRADLE_PLUGIN_API_VERSION_ATTRIBUTE,
                project.objects.named("7.0")
            )
        }
    }
}
```

> 【Groovy】

```
configurations {
    customConfiguration {
        attributes {
            attribute(
                Usage.USAGE_ATTRIBUTE,
                project.objects.named(Usage, Usage.JAVA_RUNTIME)
            )
            attribute(
                Category.CATEGORY_ATTRIBUTE,
                project.objects.named(Category, Category.LIBRARY)
            )
            // If you want to depend on a specific KGP variant:
            attribute(
                GradlePluginApiVersion.GRADLE_PLUGIN_API_VERSION_ATTRIBUTE,
                project.objects.named('7.0')
            )
        }
    }
}
```

Otherwise, you will receive an error similar to this:

```
 > Could not resolve all files for configuration ':customConfiguration'.
      > Could not resolve org.jetbrains.kotlin:kotlin-gradle-plugin:1.7.0.
```

```
         Required by:
             project :
          > Cannot choose between the following variants of org.jetbrains.kotlin:kotlin
-gradle-plugin:1.7.0:
              - gradle70RuntimeElements
              - runtimeElements
            All of them match the consumer attributes:
              - Variant 'gradle70RuntimeElements' capability org.jetbrains.kotlin:kotli
n-gradle-plugin:1.7.0:
                  - Unmatched attributes:
```

下一步做什么？

Learn more about Gradle basics and specifics.

Maven

Maven is a build system that you can use to build and manage any Java-based project.

Configure and enable the plugin

`kotlin-maven-plugin` 用于编译 Kotlin 源代码与模块，目前只支持 Maven V3。

In your `pom.xml` file, define the version of Kotlin you want to use in the `kotlin.version` property:

```
<properties>
    <kotlin.version>{{ site.data.releases.latest.version }}</kotlin.version>
</properties>
```

To enable `kotlin-maven-plugin`, update your `pom.xml` file:

```
<plugins>
    <plugin>
        <artifactId>kotlin-maven-plugin</artifactId>
        <groupId>org.jetbrains.kotlin</groupId>
        <version>2.1.21</version>
    </plugin>
</plugins>
```

Use JDK 17

To use JDK 17, in your `.mvn/jvm.config` file, add:

```
--add-opens=java.base/java.lang=ALL-UNNAMED
--add-opens=java.base/java.io=ALL-UNNAMED
```

Declare repositories

By default, the `mavenCentral` repository is available for all Maven projects. To access artifacts in other repositories, specify the ID and URL of each repository in the `<repositories>` element:

```
<repositories>
    <repository>
        <id>spring-repo</id>
```

```
        <url>https://repo.spring.io/release</url>
    </repository>
</repositories>
```

> If you declare `mavenLocal()` as a repository in a Gradle project, you may experience problems when switching between Gradle and Maven projects. For more information, see Declare repositories.
>
> 

Set dependencies

Kotlin 有一个广泛的标准库可用于应用程序。 To use the standard library in your project, 在 `pom.xml` 文件中配置以下依赖关系：

```
<dependencies>
    <dependency>
        <groupId>org.jetbrains.kotlin</groupId>
        <artifactId>kotlin-stdlib</artifactId>
        <version>${kotlin.version}</version>
    </dependency>
</dependencies>
```

> If you're targeting JDK 7 or 8 with Kotlin versions older than:
>
> - 1.8, use `kotlin-stdlib-jdk7` or `kotlin-stdlib-jdk8`, respectively.
> - 1.2, use `kotlin-stdlib-jre7` or `kotlin-stdlib-jre8`, respectively.
>
> 

如果你的项目使用 Kotlin 反射 或者测试设施，那么你还需要添加相应的依赖项。 其构件 ID 对于反射库是 `kotlin-reflect`，对于测试库是 `kotlin-test` 与 `kotlin-test-junit`。

编译只有 Kotlin 的源代码

要编译源代码，请在 `<build>` 标签中指定源代码目录：

```
<build>
    <sourceDirectory>${project.basedir}/src/main/kotlin</sourceDirectory>
    <testSourceDirectory>${project.basedir}/src/test/kotlin</testSourceDirectory>
</build>
```

需要引用 Kotlin Maven 插件来编译源代码：

```
<build>
    <plugins>
        <plugin>
            <groupId>org.jetbrains.kotlin</groupId>
            <artifactId>kotlin-maven-plugin</artifactId>
            <version>${kotlin.version}</version>

            <executions>
                <execution>
                    <id>compile</id>
                    <goals>
                        <goal>compile</goal>
                    </goals>
                </execution>

                <execution>
                    <id>test-compile</id>
                    <goals>
                        <goal>test-compile</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

Starting from Kotlin 1.8.20, you can replace the whole `<executions>` element above with `<extensions>true</extensions>`. Enabling extensions automatically adds the `compile`, `test-compile`, `kapt`, and `test-kapt` executions to your build, bound to their appropriate lifecycle phases. If you need to configure an execution, you need to specify its ID. You can find an example of this in the next section.

> If several build plugins overwrite the default lifecycle and you have also enabled the `extensions` option, the last plugin in the `<build>` section has priority in terms of lifecycle settings. All earlier changes to lifecycle settings are ignored.

同时编译 Kotlin 与 Java 源代码

要编译混合代码应用程序，必须在 Java 编译器之前调用 Kotlin 编译器。 按照 Maven 的方式，这意味着应该使用以下方法在 `maven-compiler-plugin` 之前运行 `kotlin-maven-plugin`。 确保 `pom.xml` 文件中的 `kotlin` 插件位于 `maven-compiler-plugin` 之前：

```
<build>
    <plugins>
```

```
        <plugin>
            <groupId>org.jetbrains.kotlin</groupId>
            <artifactId>kotlin-maven-plugin</artifactId>
            <version>${kotlin.version}</version>
            <extensions>true</extensions> <!-- You can set this option
            to automatically take information about lifecycles -->
            <executions>
                <execution>
                    <id>compile</id>
                    <goals>
                        <goal>compile</goal> <!-- You can skip the <goals> element
                        if you enable extensions for the plugin -->
                    </goals>
                    <configuration>
                        <sourceDirs>
                            <sourceDir>${project.basedir}/src/main/kotlin</sourceDir>
                            <sourceDir>${project.basedir}/src/main/java</sourceDir>
                        </sourceDirs>
                    </configuration>
                </execution>
                <execution>
                    <id>test-compile</id>
                    <goals>
                        <goal>test-compile</goal> <!-- You can skip the <goals> elemen
t
                        if you enable extensions for the plugin -->
                    </goals>
                    <configuration>
                        <sourceDirs>
                            <sourceDir>${project.basedir}/src/test/kotlin</sourceDir>
                            <sourceDir>${project.basedir}/src/test/java</sourceDir>
                        </sourceDirs>
                    </configuration>
                </execution>
            </executions>
        </plugin>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>3.5.1</version>
            <executions>
                <!-- 替换会被 Maven 特别处理的 default-compile -->
                <execution>
                    <id>default-compile</id>
                    <phase>none</phase>
                </execution>
                <!-- 替换会被 Maven 特别处理的 default-testCompile -->
                <execution>
                    <id>default-testCompile</id>
                    <phase>none</phase>
                </execution>
                <execution>
                    <id>java-compile</id>
```

```
                    <phase>compile</phase>
                    <goals>
                        <goal>compile</goal>
                    </goals>
                </execution>
                <execution>
                    <id>java-test-compile</id>
                    <phase>test-compile</phase>
                    <goals>
                        <goal>testCompile</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

Enable incremental compilation

为了使构建更快，可以通过添加 `kotlin.compiler.incremental` 属性来启用增量编译：

```
<properties>
    <kotlin.compiler.incremental>true</kotlin.compiler.incremental>
</properties>
```

或者，使用 `-Dkotlin.compiler.incremental=true` 选项运行构建。

Configure annotation processing

See `kapt` – Using in Maven.

Create JAR file

要创建一个仅包含模块代码的小型 JAR 文件，请在 Maven `pom.xml` 文件中的 `build->plugins` 下面包含以下内容， 其中 `main.class` 定义为一个属性，并指向主 Kotlin 或 Java 类：

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-jar-plugin</artifactId>
    <version>2.6</version>
    <configuration>
        <archive>
```

```
            <manifest>
                <addClasspath>true</addClasspath>
                <mainClass>${main.class}</mainClass>
            </manifest>
        </archive>
    </configuration>
</plugin>
```

Create a self-contained JAR file

要创建一个独立的（self-contained）JAR 文件，包含模块中的代码及其依赖项，请在 Maven `pom.xml` 文件中的 `build->plugins` 下面包含以下内容其中 `main.class` 定义为一个属性，并指向主 Kotlin 或 Java 类：

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-assembly-plugin</artifactId>
    <version>2.6</version>
    <executions>
        <execution>
            <id>make-assembly</id>
            <phase>package</phase>
            <goals> <goal>single</goal> </goals>
            <configuration>
                <archive>
                    <manifest>
                        <mainClass>${main.class}</mainClass>
                    </manifest>
                </archive>
                <descriptorRefs>
                    <descriptorRef>jar-with-dependencies</descriptorRef>
                </descriptorRefs>
            </configuration>
        </execution>
    </executions>
</plugin>
```

这个独立的 JAR 文件可以直接传给 JRE 来运行应用程序：

```
java -jar target/mymodule-0.0.1-SNAPSHOT-jar-with-dependencies.jar
```

指定编译器选项

可以将额外的编译器选项与参数指定为 Maven 插件节点的 `<configuration>` 元素下的标签：

```
<plugin>
    <groupId>org.jetbrains.kotlin</groupId>
    <artifactId>kotlin-maven-plugin</artifactId>
    <version>${kotlin.version}</version>
    <extensions>true</extensions> <!-- If you want to enable automatic addition of exe
cutions to your build -->
    <executions>……</executions>
    <configuration>
        <nowarn>true</nowarn>  <!-- 禁用警告 -->
        <args>
            <arg>-Xjsr305=strict</arg> <!-- 对 JSR-305 注解启用严格模式 -->
            ...
        </args>
    </configuration>
</plugin>
```

许多选项还可以通过属性来配置：

```
<project ……>
    <properties>
        <kotlin.compiler.languageVersion>2.1</kotlin.compiler.languageVersion>
    </properties>
</project>
```

支持以下属性：

JVM 特有的属性

名称	属性名	描述	可能的值	默认
`nowarn`		不生成警告	true、false	fal
`languageVersion`	`kotlin.compiler.languageVersion`	提供与指定语言版本源代码兼容性	"1.8"、"1.9"、"2.0"、"2.1"、"2.2"（实验性的）	
`apiVersion`	`kotlin.compiler.apiVersion`	只允许使用来自捆绑库的指定版本中的声明	"1.8"、"1.9"、"2.0"、"2.1"、"2.2"（实验性的）	
		包含要编译		该目

`sourceDirs`		源文件的目录		代根录
`compilerPlugins`		启用编译器插件		[]
`pluginOptions`		编译器插件的选项		[]
`args`		额外的编译器参数		[]
`jvmTarget`	`kotlin.compiler.jvmTarget`	生成的 JVM 字节码的目标版本	"1.8"、"9"、"10"、……、"20"、"23"	"1
`jdkHome`	`kotlin.compiler.jdkHome`	Include a custom JDK from the specified location into the classpath instead of the default JAVA_HOME		

Use BOM

To use a Kotlin Bill of Materials (BOM), write a dependency on `kotlin-bom`:

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.jetbrains.kotlin</groupId>
            <artifactId>kotlin-bom</artifactId>
            <version>2.1.21</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

生成文档

标准的 Javadoc 生成插件（`maven-javadoc-plugin`）不支持 Kotlin 代码。要生成 Kotlin 项目的文档，请使用 Dokka。 Dokka 支持混合语言项目，并且可以生成多种格式的输出，包括标准 Javadoc。 For more information about how to configure Dokka in your Maven project, see Maven.

Enable OSGi support

Learn how to enable OSGi support in your Maven project.

Ant

获取 Ant 任务

Kotlin 为 Ant 提供了三个任务：

- `kotlinc`：面向 JVM 的 Kotlin 编译器
- `kotlin2js`：面向 JavaScript 的 Kotlin 编译器
- `withKotlin`：使用标准 *javac* Ant 任务时编译 Kotlin 文件的任务

这仨任务在 *kotlin-ant.jar* 库中定义，该库位于 Kotlin 编译器包中的 `lib` 文件夹中。 需要 Ant 1.8.2+ 版本。

面向 JVM 只用 Kotlin 源代码

当项目由 Kotlin 专用源代码组成时，编译项目的最简单方法是使用 `kotlinc` 任务：

```
<project name="Ant Task Test" default="build">
    <typedef resource="org/jetbrains/kotlin/ant/antlib.xml" classpath="${kotlin.lib}/k
otlin-ant.jar"/>

    <target name="build">
        <kotlinc src="hello.kt" output="hello.jar"/>
    </target>
</project>
```

其中 `${kotlin.lib}` 指向解压缩 Kotlin 独立编译器所在文件夹。

面向 JVM 只用 Kotlin 源代码且多根

如果项目由多个源代码根组成，那么使用 `src` 作为元素来定义路径：

```
<project name="Ant Task Test" default="build">
    <typedef resource="org/jetbrains/kotlin/ant/antlib.xml" classpath="${kotlin.lib}/k
otlin-ant.jar"/>

    <target name="build">
        <kotlinc output="hello.jar">
            <src path="root1"/>
            <src path="root2"/>
        </kotlinc>
```

```
    </target>
</project>
```

面向 JVM 使用 Kotlin 和 Java 源代码

如果项目由 Kotlin 和 Java 源代码组成，虽然可以使用 `kotlinc` 来避免任务参数的重复，但是建议使用 *withKotlin* 任务：

```
<project name="Ant Task Test" default="build">
    <typedef resource="org/jetbrains/kotlin/ant/antlib.xml" classpath="${kotlin.lib}/k
otlin-ant.jar"/>

    <target name="build">
        <delete dir="classes" failonerror="false"/>
        <mkdir dir="classes"/>
        <javac destdir="classes" includeAntRuntime="false" srcdir="src">
            <withKotlin/>
        </javac>
        <jar destfile="hello.jar">
            <fileset dir="classes"/>
        </jar>
    </target>
</project>
```

还可以将正在编译的模块的名称指定为 `moduleName` 属性：

```
<withKotlin moduleName="myModule"/>
```

面向 JavaScript 用单个源文件夹

```
<project name="Ant Task Test" default="build">
    <typedef resource="org/jetbrains/kotlin/ant/antlib.xml" classpath="${kotlin.lib}/k
otlin-ant.jar"/>

    <target name="build">
        <kotlin2js src="root1" output="out.js"/>
    </target>
</project>
```

面向 JavaScript 用 Prefix、 PostFix 以及 sourcemap 选项

```
<project name="Ant Task Test" default="build">
    <taskdef resource="org/jetbrains/kotlin/ant/antlib.xml" classpath="${kotlin.lib}/k
otlin-ant.jar"/>

    <target name="build">
        <kotlin2js src="root1" output="out.js" outputPrefix="prefix" outputPostfix="po
stfix" sourcemap="true"/>
    </target>
</project>
```

面向 JavaScript 用单个源文件夹以及 metaInfo 选项

如果要将翻译结果作为 Kotlin/JavaScript 库分发，那么 `metaInfo` 选项会很有用。 如果 `metaInfo` 设置为 `true`，则在编译期间将创建具有二进制元数据的额外的 JS 文件。该文件应该与翻译结果一起分发：

```
<project name="Ant Task Test" default="build">
    <typedef resource="org/jetbrains/kotlin/ant/antlib.xml" classpath="${kotlin.lib}/k
otlin-ant.jar"/>

    <target name="build">
        <!-- 会创建 out.meta.js，其中包含二进制元数据 -->
        <kotlin2js src="root1" output="out.js" metaInfo="true"/>
    </target>
</project>
```

参考

元素和属性的完整列表如下：

kotlinc 和 kotlin2js 的公共属性

名称	描述	必需	默认值
`src`	要编译的 Kotlin 源文件或目录	是	
`nowarn`	禁止所有编译警告	否	false
`noStdlib`	不要将 Kotlin 标准库包含进 classpath	否	false
`failOnError`	在编译期间检测到错误时，会导致构建失败	否	true

kotlinc 属性

名称	描述	必需	默认值
`output`	目标目录或 .jar 文件名	是	
`classpath`	编译类路径	否	
`classpathref`	编译类路径引用	否	
`includeRuntime`	Kotlin 运行时库是否包含在 jar 中，如果 `output` 是 .jar 文件的话	否	true
`moduleName`	编译的模块的名称	否	目标（如果指定的话）或项目的名称

kotlin2js 属性

名称	描述	必需
`output`	目标文件	是
`libraries`	Kotlin 库的路径	否
`outputPrefix`	生成的 JavaScript 文件所用前缀	否
`outputSuffix`	生成的 JavaScript 文件所用后缀	否
`sourcemap`	是否要生成 sourcemap 文件	否
`metaInfo`	是否要生成具有二进制描述符的元数据文件	否
`main`	编译器是否生成调用 main 函数的代码	否

传递原始编译器参数

如需传递原始编译器参数，可以使用带 `value` 或 `line` 属性的 `<compilerarg>` 元素。 可以放在 `<kotlinc>`、 `<kotlin2js>` 与 `<withKotlin>` 任务元素内，如下所示：

```
<kotlinc src="${test.data}/hello.kt" output="${temp}/hello.jar">
    <compilerarg value="-Xno-inline"/>
    <compilerarg line="-Xno-call-assertions -Xno-param-assertions"/>
    <compilerarg value="-Xno-optimize"/>
</kotlinc>
```

当运行 `kotlinc -help` 时，会显示可以使用的参数的完整列表。

Dokka

- 简介
- Dokka 入门
- 运行 Dokka
  - Gradle
  - Maven
  - CLI
- 输出格式
  - HTML
  - Markdown
  - Javadoc
- Dokka 插件
- 模块文档

简介

Dokka 是 Kotlin 的 API 文档引擎。

就像 Kotlin 本身一样，Dokka 支持混合语言项目。 它理解 Kotlin 的 KDoc 注释与 Java 的 Javadoc 注释。

Dokka 可以生成多种格式的文档，包括它自己的现代 HTML 格式、 多种风格的 Markdown 以及 Java 的 Javadoc HTML。

以下是一些将 Dokka 用于其 API 参考文档的库：

- kotlinx.coroutines
- Bitmovin
- Hexagon
- Ktor
- OkHttp

可以使用 Gradle、 Maven 或者在命令行运行 Dokka。 它也是高度可插拔的。

参见 Dokka 入门 来迈出使用 Dokka 的第一步。

社区

Dokka 在 Kotlin 社区 Slack 中有个专属的 `#dokka` 频道， 可以在其中讨论 Dokka 及其插件以及如何开发插件，也可以与维护人取得联系。

Dokka 入门

Below you can find simple instructions to help you get started with Dokka.

> 【Gradle Kotlin DSL】

Apply the Gradle plugin for Dokka in the root build script of your project:

```
plugins {
    id("org.jetbrains.dokka") version "2.0.0"
}
```

When documenting multi-project builds, you need to apply the Gradle plugin within subprojects as well:

```
subprojects {
    apply(plugin = "org.jetbrains.dokka")
}
```

To generate documentation, run the following Gradle tasks:

- `dokkaHtml` for single-project builds
- `dokkaHtmlMultiModule` for multi-project builds

By default, the output directory is set to `/build/dokka/html` and `/build/dokka/htmlMultiModule`.

To learn more about using Dokka with Gradle, see Gradle.

> 【Gradle Groovy DSL】

Apply the Gradle plugin for Dokka in the root build script of your project:

```
plugins {
    id 'org.jetbrains.dokka' version '2.0.0'
}
```

When documenting multi-project builds, you need to apply the Gradle plugin within subprojects as well:

```
subprojects {
    apply plugin: 'org.jetbrains.dokka'
}
```

To generate documentation, run the following Gradle tasks:

- `dokkaHtml` for single-project builds
- `dokkaHtmlMultiModule` for multi-project builds

By default, the output directory is set to `/build/dokka/html` and `/build/dokka/htmlMultiModule`.

To learn more about using Dokka with Gradle, see Gradle.

> 【Maven】

Add the Maven plugin for Dokka to the `plugins` section of your POM file:

```
<build>
    <plugins>
        <plugin>
            <groupId>org.jetbrains.dokka</groupId>
            <artifactId>dokka-maven-plugin</artifactId>
            <version>2.0.0</version>
            <executions>
                <execution>
                    <phase>pre-site</phase>
                    <goals>
                        <goal>dokka</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

To generate documentation, run the `dokka:dokka` goal.

By default, the output directory is set to `target/dokka`.

To learn more about using Dokka with Maven, see Maven.

运行 Dokka

- Gradle
- Maven
- CLI

Gradle

To generate documentation for a Gradle-based project, you can use the Gradle plugin for Dokka.

It comes with basic autoconfiguration for your project, has convenient Gradle tasks for generating documentation, and provides a great deal of configuration options to customize the output.

You can play around with Dokka and see how it can be configured for various projects by visiting our Gradle example projects.

Apply Dokka

The recommended way of applying the Gradle plugin for Dokka is with the plugins DSL:

> 【Kotlin】

```
plugins {
    id("org.jetbrains.dokka") version "2.0.0"
}
```

> 【Groovy】

```
plugins {
    id 'org.jetbrains.dokka' version '2.0.0'
}
```

When documenting multi-project builds, you need to apply the Gradle plugin for Dokka within subprojects as well. You can use `allprojects {}` or `subprojects {}` Gradle configurations to achieve that:

> 【Gradle Kotlin DSL】

```
subprojects {
    apply(plugin = "org.jetbrains.dokka")
```

```
}
```

【Gradle Groovy DSL】

```
subprojects {
    apply plugin: 'org.jetbrains.dokka'
}
```

See Configuration examples if you are not sure where to apply Dokka.

> Under the hood, Dokka uses the Kotlin Gradle plugin to perform autoconfiguration of source sets for which documentation is to be generated. Make sure to apply the Kotlin Gradle Plugin or configure source sets manually.
>
> 
>
> If you are using Dokka in a precompiled script plugin, you need to add the Kotlin Gradle plugin as a dependency for it to work properly.
>
> 

If you cannot use the plugins DSL for some reason, you can use the legacy method of applying plugins.

Generate documentation

The Gradle plugin for Dokka comes with HTML, Markdown and Javadoc output formats built in. It adds a number of tasks for generating documentation, both for single and multi-project builds.

Single-project builds

Use the following tasks to build documentation for simple, single-project applications and libraries:

Task	Description
`dokkaHtml`	Generates documentation in HTML format.

Experimental formats

Task	Description

`dokkaGfm`	Generates documentation in GitHub Flavored Markdown format.
`dokkaJavadoc`	Generates documentation in Javadoc format.
`dokkaJekyll`	Generates documentation in Jekyll compatible Markdown format.

By default, generated documentation is located in the `build/dokka/{format}` directory of your project. The output location, among other things, can be configured.

Multi-project builds

For documenting multi-project builds, make sure that you apply the Gradle plugin for Dokka within subprojects that you want to generate documentation for, as well as in their parent project.

MultiModule tasks

`MultiModule` tasks generate documentation for each subproject individually via `Partial` tasks, collect and process all outputs, and produce complete documentation with a common table of contents and resolved cross-project references.

Dokka creates the following tasks for **parent** projects automatically:

Task	Description
`dokkaHtmlMultiModule`	Generates multi-module documentation in HTML output format.

Experimental formats (multi-module)

Task	Description
`dokkaGfmMultiModule`	Generates multi-module documentation in GitHub Flavored Markdown output format.
`dokkaJekyllMultiModule`	Generates multi-module documentation in Jekyll compatible Markdown output format.

> The Javadoc output format does not have a `MultiModule` task, but a `Collector` task can be used instead.
>
> 

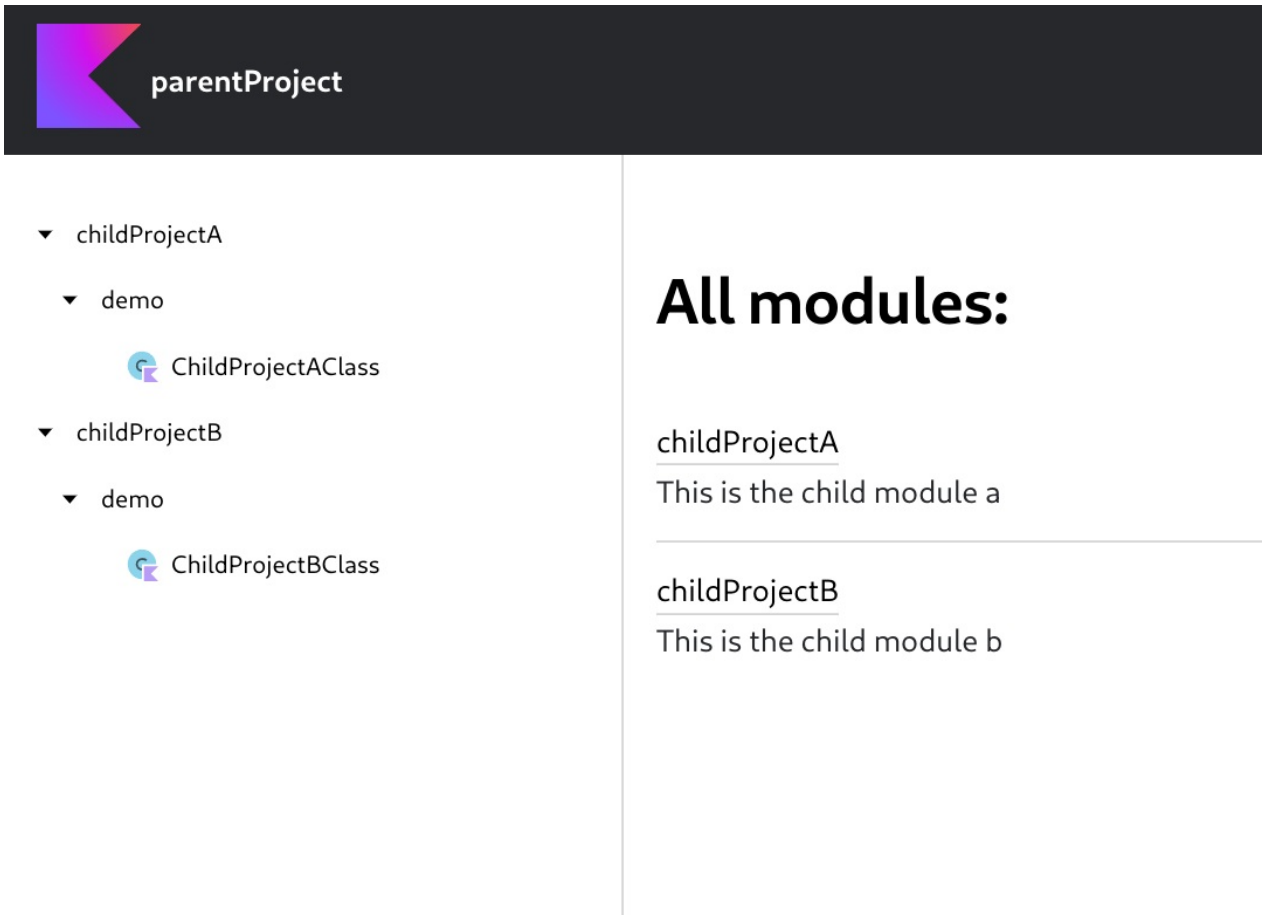
By default, you can find ready-to-use documentation under `{parentProject}/build/dokka/{format}MultiModule` directory.

MultiModule results

Given a project with the following structure:

```
parentProject
    └── childProjectA
        ├── demo
            ├── ChildProjectAClass
    └── childProjectB
        ├── demo
            ├── ChildProjectBClass
```

These pages are generated after running `dokkaHtmlMultiModule`:



See our multi-module project example for more details.

Collector tasks

Similar to `MultiModule` tasks, `Collector` tasks are created for each parent project: `dokkaHtmlCollector`, `dokkaGfmCollector`, `dokkaJavadocCollector` and `dokkaJekyllCollector`.

A `Collector` task executes the corresponding single-project task for each subproject (for example, `dokkaHtml`), and merges all outputs into a single virtual project.

The resulting documentation looks as if you have a single-project build that contains all declarations from the subprojects.

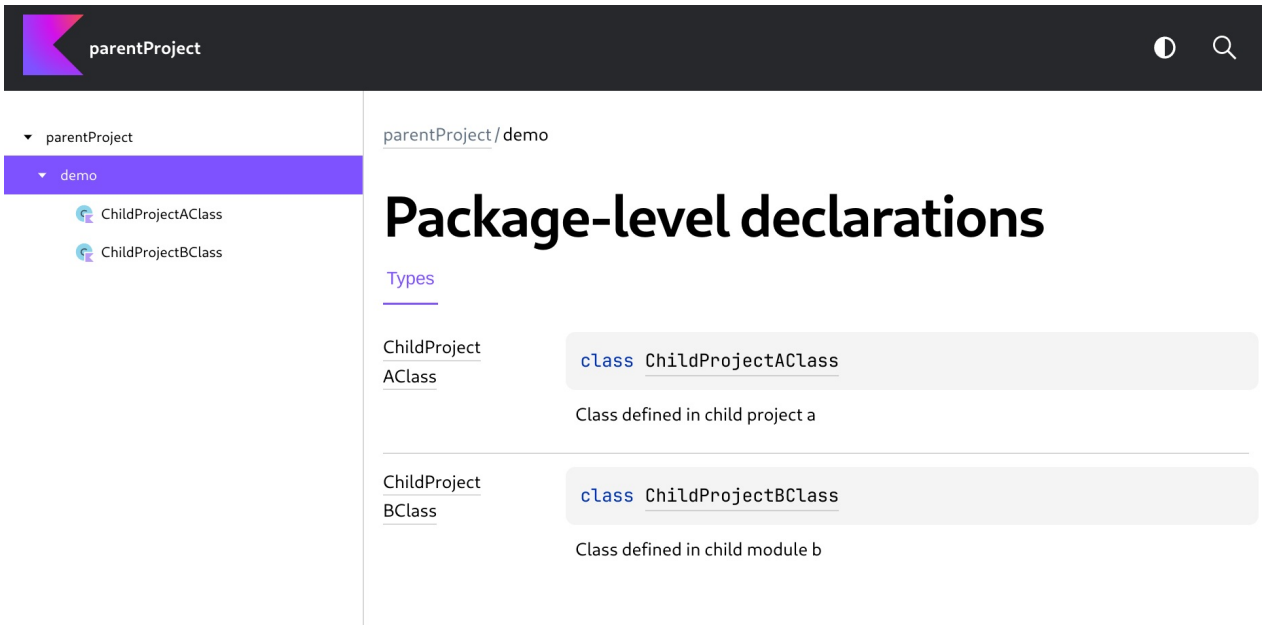
> Use the `dokkaJavadocCollector` task if you need to create Javadoc documentation for your multi-project build.
>
> 

Collector results

Given a project with the following structure:

```
parentProject
    └── childProjectA
        ├── demo
            ├── ChildProjectAClass
    └── childProjectB
        ├── demo
            ├── ChildProjectBClass
```

These pages are generated after running `dokkaHtmlCollector`:



See our multi-module project example for more details.

Partial tasks

Each subproject has `Partial` tasks created for it: `dokkaHtmlPartial`, `dokkaGfmPartial`, and `dokkaJekyllPartial`.

These tasks are not intended to be run independently, they are called by the parent's MultiModule task.

However, you can configure `Partial` tasks to customize Dokka for your subprojects.

> Output generated by `Partial` tasks contains unresolved HTML templates and references, so it cannot be used on its own without post-processing done by the parent's `MultiModule` task.
>
> 
>
> If you want to generate documentation for a single subproject only, use single-project tasks. For example, `:subprojectName:dokkaHtml`.
>
> 

Build javadoc.jar

If you want to publish your library to a repository, you may need to provide a `javadoc.jar` file that contains API reference documentation of your library.

For example, if you want to publish to Maven Central, you must supply a `javadoc.jar` alongside your project. However, not all repositories have that rule.

The Gradle plugin for Dokka does not provide any way to do this out of the box, but it can be achieved with custom Gradle tasks. One for generating documentation in HTML format and another one for Javadoc format:

> 【Kotlin】

```
tasks.register<Jar>("dokkaHtmlJar") {
    dependsOn(tasks.dokkaHtml)
    from(tasks.dokkaHtml.flatMap { it.outputDirectory })
    archiveClassifier.set("html-docs")
}

tasks.register<Jar>("dokkaJavadocJar") {
    dependsOn(tasks.dokkaJavadoc)
    from(tasks.dokkaJavadoc.flatMap { it.outputDirectory })
    archiveClassifier.set("javadoc")
}
```

> 【Groovy】

```
tasks.register('dokkaHtmlJar', Jar.class) {
    dependsOn(dokkaHtml)
    from(dokkaHtml)
    archiveClassifier.set("html-docs")
}
```

```
tasks.register('dokkaJavadocJar', Jar.class) {
    dependsOn(dokkaJavadoc)
    from(dokkaJavadoc)
    archiveClassifier.set("javadoc")
}
```

> If you publish your library to Maven Central, you can use services like javadoc.io to host your library's API documentation for free and without any setup. It takes documentation pages straight from the `javadoc.jar`. It works well with the HTML format as demonstrated in this example.
>
> 

Configuration examples

Depending on the type of project that you have, the way you apply and configure Dokka differs slightly. However, configuration options themselves are the same, regardless of the type of your project.

For simple and flat projects with a single `build.gradle.kts` or `build.gradle` file found in the root of your project, see Single-project configuration.

For a more complex build with subprojects and multiple nested `build.gradle.kts` or `build.gradle` files, see Multi-project configuration.

Single-project configuration

Single-project builds usually have only one `build.gradle.kts` or `build.gradle` file in the root of the project, and typically have the following structure:

> 【Kotlin】

Single platform:

```
.
├── build.gradle.kts
└── src
    └── main
        └── kotlin
            └── HelloWorld.kt
```

Multiplatform:

```
.
├── build.gradle.kts
└── src
    └── commonMain
        └── kotlin
            └── Common.kt
    └── jvmMain
        └── kotlin
            └── JvmUtils.kt
    └── nativeMain
        └── kotlin
            └── NativeUtils.kt
```

> 【Groovy】

Single platform:

```
.
├── build.gradle
└── src
    └── main
        └── kotlin
            └── HelloWorld.kt
```

Multiplatform:

```
.
├── build.gradle
└── src
    └── commonMain
        └── kotlin
            └── Common.kt
    └── jvmMain
        └── kotlin
            └── JvmUtils.kt
    └── nativeMain
        └── kotlin
            └── NativeUtils.kt
```

In such projects, you need to apply Dokka and its configuration in the root `build.gradle.kts` or `build.gradle` file.

You can configure tasks and output formats individually:

> 【Kotlin】

Inside `./build.gradle.kts`:

```
plugins {
    id("org.jetbrains.dokka") version "2.0.0"
}

tasks.dokkaHtml {
    outputDirectory.set(layout.buildDirectory.dir("documentation/html"))
}

tasks.dokkaGfm {
    outputDirectory.set(layout.buildDirectory.dir("documentation/markdown"))
}
```

> 【Groovy】

Inside `./build.gradle`:

```
plugins {
    id 'org.jetbrains.dokka' version '2.0.0'
}

dokkaHtml {
    outputDirectory.set(file("build/documentation/html"))
}

dokkaGfm {
    outputDirectory.set(file("build/documentation/markdown"))
}
```

Or you can configure all tasks and output formats at the same time:

> 【Kotlin】

Inside `./build.gradle.kts`:

```
import org.jetbrains.dokka.gradle.DokkaTask
```

```
import org.jetbrains.dokka.gradle.DokkaTaskPartial
import org.jetbrains.dokka.DokkaConfiguration.Visibility

plugins {
    id("org.jetbrains.dokka") version "2.0.0"
}

// Configure all single-project Dokka tasks at the same time,
// such as dokkaHtml, dokkaJavadoc and dokkaGfm.
tasks.withType<DokkaTask>().configureEach {
    dokkaSourceSets.configureEach {
        documentedVisibilities.set(
            setOf(
                Visibility.PUBLIC,
                Visibility.PROTECTED,
            )
        )

        perPackageOption {
            matchingRegex.set(".*internal.*")
            suppress.set(true)
        }
    }
}
```

> 【Groovy】

Inside `./build.gradle`:

```
import org.jetbrains.dokka.gradle.DokkaTask
import org.jetbrains.dokka.gradle.DokkaTaskPartial
import org.jetbrains.dokka.DokkaConfiguration.Visibility

plugins {
    id 'org.jetbrains.dokka' version '2.0.0'
}

// Configure all single-project Dokka tasks at the same time,
// such as dokkaHtml, dokkaJavadoc and dokkaGfm.
tasks.withType(DokkaTask.class) {
    dokkaSourceSets.configureEach {
        documentedVisibilities.set([
                Visibility.PUBLIC,
                Visibility.PROTECTED
        ])

        perPackageOption {
            matchingRegex.set(".*internal.*")
            suppress.set(true)
        }
```

```
    }
}
```

Multi-project configuration

Gradle's multi-project builds are more complex in structure and configuration. They usually have multiple nested `build.gradle.kts` or `build.gradle` files, and typically have the following structure:

> 【Kotlin】

```
.
├── build.gradle.kts
├── settings.gradle.kts
├── subproject-A
    └── build.gradle.kts
    └── src
        └── main
            └── kotlin
                └── HelloFromA.kt
├── subproject-B
    └── build.gradle.kts
    └── src
        └── main
            └── kotlin
                └── HelloFromB.kt
```

> 【Groovy】

```
.
├── build.gradle
├── settings.gradle
├── subproject-A
    └── build.gradle
    └── src
        └── main
            └── kotlin
                └── HelloFromA.kt
├── subproject-B
    └── build.gradle
    └── src
        └── main
            └── kotlin
```

```
                └── HelloFromB.kt
```

In this case, there are multiple ways of applying and configuring Dokka.

Subproject configuration

To configure subprojects in a multi-project build, you need to configure `Partial` tasks.

You can configure all subprojects at the same time in the root `build.gradle.kts` or `build.gradle` file, using Gradle's `allprojects {}` or `subprojects {}` configuration blocks:

> 【Kotlin】

In the root `./build.gradle.kts`：

```kotlin
import org.jetbrains.dokka.gradle.DokkaTaskPartial

plugins {
    id("org.jetbrains.dokka") version "2.0.0"
}

subprojects {
    apply(plugin = "org.jetbrains.dokka")

    // configure only the HTML task
    tasks.dokkaHtmlPartial {
        outputDirectory.set(layout.buildDirectory.dir("docs/partial"))
    }

    // configure all format tasks at once
    tasks.withType<DokkaTaskPartial>().configureEach {
        dokkaSourceSets.configureEach {
            includes.from("README.md")
        }
    }
}
```

> 【Groovy】

In the root `./build.gradle`：

```groovy
import org.jetbrains.dokka.gradle.DokkaTaskPartial
```

```
plugins {
    id 'org.jetbrains.dokka' version '2.0.0'
}

subprojects {
    apply plugin: 'org.jetbrains.dokka'

    // configure only the HTML task
    dokkaHtmlPartial {
        outputDirectory.set(file("build/docs/partial"))
    }

    // configure all format tasks at once
    tasks.withType(DokkaTaskPartial.class) {
        dokkaSourceSets.configureEach {
            includes.from("README.md")
        }
    }
}
```

Alternatively, you can apply and configure Dokka within subprojects individually.

For example, to have specific settings for the `subproject-A` subproject only, you need to apply the following code inside `./subproject-A/build.gradle.kts`:

> **【Kotlin】**

Inside `./subproject-A/build.gradle.kts`:

```
apply(plugin = "org.jetbrains.dokka")

// configuration for subproject-A only.
tasks.dokkaHtmlPartial {
    outputDirectory.set(layout.buildDirectory.dir("docs/partial"))
}
```

> **【Groovy】**

Inside `./subproject-A/build.gradle`:

```
apply plugin: 'org.jetbrains.dokka'

// configuration for subproject-A only.
dokkaHtmlPartial {
```

```
    outputDirectory.set(file("build/docs/partial"))
}
```

Parent project configuration

If you want to configure something which is universal across all documentation and does not belong to the subprojects - in other words, it's a property of the parent project - you need to configure the `MultiModule` tasks.

For example, if you want to change the name of your project which is used in the header of the HTML documentation, you need to apply the following inside the root `build.gradle.kts` or `build.gradle` file:

> 【Kotlin】

In the root `./build.gradle.kts` file:

```
plugins {
    id("org.jetbrains.dokka") version "2.0.0"
}

tasks.dokkaHtmlMultiModule {
    moduleName.set("WHOLE PROJECT NAME USED IN THE HEADER")
}
```

> 【Groovy】

In the root `./build.gradle` file:

```
plugins {
    id 'org.jetbrains.dokka' version '2.0.0'
}

dokkaHtmlMultiModule {
    moduleName.set("WHOLE PROJECT NAME USED IN THE HEADER")
}
```

Configuration options

Dokka has many configuration options to tailor your and your reader's experience.

Below are some examples and detailed descriptions for each configuration section. You can also find an example with all configuration options applied at the bottom of the page.

See Configuration examples for more details on where to apply configuration blocks and how.

General configuration

Here is an example of general configuration of any Dokka task, regardless of source set or package:

> 【Kotlin】

```
import org.jetbrains.dokka.gradle.DokkaTask

// Note: To configure multi-project builds, you need
//       to configure Partial tasks of the subprojects.
//       See "Configuration example" section of documentation.
tasks.withType<DokkaTask>().configureEach {
    moduleName.set(project.name)
    moduleVersion.set(project.version.toString())
    outputDirectory.set(layout.buildDirectory.dir("dokka/$name"))
    failOnWarning.set(false)
    suppressObviousFunctions.set(true)
    suppressInheritedMembers.set(false)
    offlineMode.set(false)

    // ..
    // source set configuration section
    // ..
}
```

> 【Groovy】

```
import org.jetbrains.dokka.gradle.DokkaTask

// Note: To configure multi-project builds, you need
//       to configure Partial tasks of the subprojects.
//       See "Configuration example" section of documentation.
tasks.withType(DokkaTask.class) {
    moduleName.set(project.name)
    moduleVersion.set(project.version.toString())
    outputDirectory.set(file("build/dokka/$name"))
    failOnWarning.set(false)
    suppressObviousFunctions.set(true)
```

```
    suppressInheritedMembers.set(false)
    offlineMode.set(false)

    // ..
    // source set configuration section
    // ..
}
```

The display name used to refer to the module. It is used for the table of contents, navigation, logging, etc.

If set for a single-project build or a `MultiModule` task, it is used as the project name.

Default: Gradle project name

The module version. If set for a single-project build or a `MultiModule` task, it is used as the project version.

Default: Gradle project version

The directory to where documentation is generated, regardless of format. It can be set on a per-task basis.

The default is `{project}/{buildDir}/{format}`, where `{format}` is the task name with the "dokka" prefix removed. For the `dokkaHtmlMultiModule` task, it is `project/buildDir/htmlMultiModule`.

Whether to fail documentation generation if Dokka has emitted a warning or an error. The process waits until all errors and warnings have been emitted first.

This setting works well with `reportUndocumented`.

Default: `false`

Whether to suppress obvious functions.

A function is considered to be obvious if it is:

- Inherited from `kotlin.Any`, `Kotlin.Enum`, `java.lang.Object` or `java.lang.Enum`, such as `equals`, `hashCode`, `toString`.
- Synthetic (generated by the compiler) and does not have any documentation, such as `dataClass.componentN` or `dataClass.copy`.

Default: `true`

Whether to suppress inherited members that aren't explicitly overridden in a given class.

Note: This can suppress functions such as `equals` / `hashCode` / `toString`, but cannot suppress synthetic functions such as `dataClass.componentN` and `dataClass.copy`. Use `suppressObviousFunctions` for that.

Default: `false`

Whether to resolve remote files/links over your network.

This includes package-lists used for generating external documentation links. For example, to make classes from the standard library clickable.

Setting this to `true` can significantly speed up build times in certain cases, but can also worsen documentation quality and user experience. For example, by not resolving class/member links from your dependencies, including the standard library.

Note: You can cache fetched files locally and provide them to Dokka as local paths. See `externalDocumentationLinks` section.

Default: `false`

Source set configuration

Dokka allows configuring some options for Kotlin source sets:

> 【Kotlin】

```
import org.jetbrains.dokka.DokkaConfiguration.Visibility
import org.jetbrains.dokka.gradle.DokkaTask
import org.jetbrains.dokka.Platform
import java.net.URL

// Note: To configure multi-project builds, you need
//       to configure Partial tasks of the subprojects.
//       See "Configuration example" section of documentation.
tasks.withType<DokkaTask>().configureEach {
    // ..
    // general configuration section
    // ..

    dokkaSourceSets {
        // configuration exclusive to the 'linux' source set
        named("linux") {
            dependsOn("native")
            sourceRoots.from(file("linux/src"))
        }
        configureEach {
            suppress.set(false)
```

```
            displayName.set(name)
            documentedVisibilities.set(setOf(Visibility.PUBLIC))
            reportUndocumented.set(false)
            skipEmptyPackages.set(true)
            skipDeprecated.set(false)
            suppressGeneratedFiles.set(true)
            jdkVersion.set(8)
            languageVersion.set("1.7")
            apiVersion.set("1.7")
            noStdlibLink.set(false)
            noJdkLink.set(false)
            noAndroidSdkLink.set(false)
            includes.from(project.files(), "packages.md", "extra.md")
            platform.set(Platform.DEFAULT)
            sourceRoots.from(file("src"))
            classpath.from(project.files(), file("libs/dependency.jar"))
            samples.from(project.files(), "samples/Basic.kt", "samples/Advanced.kt")

            sourceLink {
                // Source link section
            }
            externalDocumentationLink {
                // External documentation link section
            }
            perPackageOption {
                // Package options section
            }
        }
    }
}
```

> 【Groovy】

```
import org.jetbrains.dokka.DokkaConfiguration.Visibility
import org.jetbrains.dokka.gradle.DokkaTask
import org.jetbrains.dokka.Platform
import java.net.URL

// Note: To configure multi-project builds, you need
//       to configure Partial tasks of the subprojects.
//       See "Configuration example" section of documentation.
tasks.withType(DokkaTask.class) {
    // ..
    // general configuration section
    // ..

    dokkaSourceSets {
        // configuration exclusive to the 'linux' source set
        named("linux") {
            dependsOn("native")
```

```
            sourceRoots.from(file("linux/src"))
        }
        configureEach {
            suppress.set(false)
            displayName.set(name)
            documentedVisibilities.set([Visibility.PUBLIC])
            reportUndocumented.set(false)
            skipEmptyPackages.set(true)
            skipDeprecated.set(false)
            suppressGeneratedFiles.set(true)
            jdkVersion.set(8)
            languageVersion.set("1.7")
            apiVersion.set("1.7")
            noStdlibLink.set(false)
            noJdkLink.set(false)
            noAndroidSdkLink.set(false)
            includes.from(project.files(), "packages.md", "extra.md")
            platform.set(Platform.DEFAULT)
            sourceRoots.from(file("src"))
            classpath.from(project.files(), file("libs/dependency.jar"))
            samples.from(project.files(), "samples/Basic.kt", "samples/Advanced.kt")

            sourceLink {
                // Source link section
            }
            externalDocumentationLink {
                // External documentation link section
            }
            perPackageOption {
                // Package options section
            }
        }
    }
}
```

Whether this source set should be skipped when generating documentation.

Default: `false`

The display name used to refer to this source set.

The name is used both externally (for example, as source set name visible to documentation readers) and internally (for example, for logging messages of `reportUndocumented`).

By default, the value is deduced from information provided by the Kotlin Gradle plugin.

The set of visibility modifiers that should be documented.

This can be used if you want to document `protected` / `internal` / `private` declarations, as well as if you want to exclude `public` declarations and only document internal API.

This can be configured on per-package basis.

Default: `DokkaConfiguration.Visibility.PUBLIC`

Whether to emit warnings about visible undocumented declarations, that is declarations without KDocs after they have been filtered by `documentedVisibilities` and other filters.

This setting works well with `failOnWarning`.

This can be configured on per-package basis.

Default: `false`

Whether to skip packages that contain no visible declarations after various filters have been applied.

For example, if `skipDeprecated` is set to `true` and your package contains only deprecated declarations, it is considered to be empty.

Default: `true`

Whether to document declarations annotated with `@Deprecated`.

This can be configured on per-package basis.

Default: `false`

Whether to document/analyze generated files.

Generated files are expected to be present under the `{project}/{buildDir}/generated` directory.

If set to `true`, it effectively adds all files from that directory to the `suppressedFiles` option, so you can configure it manually.

Default: `true`

The JDK version to use when generating external documentation links for Java types.

For example, if you use `java.util.UUID` in some public declaration signature, and this option is set to `8`, Dokka generates an external documentation link to JDK 8 Javadocs for it.

Default: JDK 8

The Kotlin language version used for setting up analysis and @sample environment.

By default, the latest language version available to Dokka's embedded compiler is used.

The Kotlin API version used for setting up analysis and @sample environment.

By default, it is deduced from `languageVersion`.

Whether to generate external documentation links that lead to the API reference documentation of Kotlin's standard library.

Note: Links **are** generated when `noStdLibLink` is set to `false`.

Default: `false`

Whether to generate external documentation links to JDK's Javadocs.

The version of JDK Javadocs is determined by the `jdkVersion` option.

Note: Links **are** generated when `noJdkLink` is set to `false`.

Default: `false`

Whether to generate external documentation links to the Android SDK API reference.

This is only relevant in Android projects, ignored otherwise.

Note: Links **are** generated when `noAndroidSdkLink` is set to `false`.

Default: `false`

A list of Markdown files that contain module and package documentation.

The contents of the specified files are parsed and embedded into documentation as module and package descriptions.

See Dokka gradle example for an example of what it looks like and how to use it.

The platform to be used for setting up code analysis and @sample environment.

The default value is deduced from information provided by the Kotlin Gradle plugin.

The source code roots to be analyzed and documented. Acceptable inputs are directories and individual `.kt` / `.java` files.

By default, source roots are deduced from information provided by the Kotlin Gradle plugin.

The classpath for analysis and interactive samples.

This is useful if some types that come from dependencies are not resolved/picked up automatically.

This option accepts both `.jar` and `.klib` files.

By default, classpath is deduced from information provided by the Kotlin Gradle plugin.

A list of directories or files that contain sample functions which are referenced via the @sample KDoc tag.

Source link configuration

The `sourceLinks` configuration block allows you to add a `source` link to each signature that leads to the `remoteUrl` with a specific line number. (The line number is configurable by setting `remoteLineSuffix`).

This helps readers to find the source code for each declaration.

For an example, see the documentation for the `count()` function in `kotlinx.coroutines`.

> 【Kotlin】

```
import org.jetbrains.dokka.gradle.DokkaTask
import java.net.URL

// Note: To configure multi-project builds, you need
//       to configure Partial tasks of the subprojects.
//       See "Configuration example" section of documentation.
tasks.withType<DokkaTask>().configureEach {
    // ..
    // general configuration section
    // ..

    dokkaSourceSets.configureEach {
        // ..
        // source set configuration section
        // ..

        sourceLink {
            localDirectory.set(projectDir.resolve("src"))
            remoteUrl.set(URL("https://github.com/kotlin/dokka/tree/master/src"))
            remoteLineSuffix.set("#L")
        }
    }
}
```

> 【Groovy】

```
import org.jetbrains.dokka.gradle.DokkaTask
import java.net.URL

// Note: To configure multi-project builds, you need
//       to configure Partial tasks of the subprojects.
//       See "Configuration example" section of documentation.
tasks.withType(DokkaTask.class) {
    // ..
```

```
    // general configuration section
    // ..

    dokkaSourceSets.configureEach {
        // ..
        // source set configuration section
        // ..

        sourceLink {
            localDirectory.set(file("src"))
            remoteUrl.set(new URL("https://github.com/kotlin/dokka/tree/master/src"))
            remoteLineSuffix.set("#L")
        }
    }
}
```

The path to the local source directory. The path must be relative to the root of the current project.

The URL of the source code hosting service that can be accessed by documentation readers, like GitHub, GitLab, Bitbucket, etc. This URL is used to generate source code links of declarations.

The suffix used to append the source code line number to the URL. This helps readers navigate not only to the file, but to the specific line number of the declaration.

The number itself is appended to the specified suffix. For example, if this option is set to `#L` and the line number is 10, the resulting URL suffix is `#L10`.

Suffixes used by popular services:

- GitHub: `#L`
- GitLab: `#L`
- Bitbucket: `#lines-`

Default: `#L`

Package options

The `perPackageOption` configuration block allows setting some options for specific packages matched by `matchingRegex`.

> 【Kotlin】

```
import org.jetbrains.dokka.DokkaConfiguration.Visibility
import org.jetbrains.dokka.gradle.DokkaTask
```

```
// Note: To configure multi-project builds, you need
//       to configure Partial tasks of the subprojects.
//       See "Configuration example" section of documentation.
tasks.withType<DokkaTask>().configureEach {
    // ..
    // general configuration section
    // ..

    dokkaSourceSets.configureEach {
        // ..
        // source set configuration section
        // ..

        perPackageOption {
            matchingRegex.set(".*api.*")
            suppress.set(false)
            skipDeprecated.set(false)
            reportUndocumented.set(false)
            documentedVisibilities.set(setOf(Visibility.PUBLIC))
        }
    }
}
```

> 【Groovy】

```
import org.jetbrains.dokka.DokkaConfiguration.Visibility
import org.jetbrains.dokka.gradle.DokkaTask

// Note: To configure multi-project builds, you need
//       to configure Partial tasks of the subprojects.
//       See "Configuration example" section of documentation.
tasks.withType(DokkaTask.class) {
    // ..
    // general configuration section
    // ..

    dokkaSourceSets.configureEach {
        // ..
        // Source set configuration section
        // ..

        perPackageOption {
            matchingRegex.set(".*api.*")
            suppress.set(false)
            skipDeprecated.set(false)
            reportUndocumented.set(false)
            documentedVisibilities.set([Visibility.PUBLIC])
        }
    }
```

```
}
```

The regular expression that is used to match the package.

Default: `.*`

Whether this package should be skipped when generating documentation.

Default: `false`

Whether to document declarations annotated with `@Deprecated`.

This can be configured on source set level.

Default: `false`

Whether to emit warnings about visible undocumented declarations, that is declarations without KDocs after they have been filtered by `documentedVisibilities` and other filters.

This setting works well with `failOnWarning`.

This can be configured on source set level.

Default: `false`

The set of visibility modifiers that should be documented.

This can be used if you want to document `protected` / `internal` / `private` declarations within this package, as well as if you want to exclude `public` declarations and only document internal API.

This can be configured on source set level.

Default: `DokkaConfiguration.Visibility.PUBLIC`

External documentation links configuration

The `externalDocumentationLink` block allows the creation of links that lead to the externally hosted documentation of your dependencies.

For example, if you are using types from `kotlinx.serialization`, by default they are unclickable in your documentation, as if they are unresolved. However, since the API reference documentation for `kotlinx.serialization` is built by Dokka and is published on kotlinlang.org, you can configure external documentation links for it. Thus allowing Dokka to generate links for types from the library, making them resolve successfully and clickable.

By default, external documentation links for Kotlin standard library, JDK, Android SDK and AndroidX are configured.

【Kotlin】

```
import org.jetbrains.dokka.gradle.DokkaTask
import java.net.URL

// Note: To configure multi-project builds, you need
//       to configure Partial tasks of the subprojects.
//       See "Configuration example" section of documentation.
tasks.withType<DokkaTask>().configureEach {
    // ..
    // general configuration section
    // ..

    dokkaSourceSets.configureEach {
        // ..
        // source set configuration section
        // ..

        externalDocumentationLink {
            url.set(URL("https://kotlinlang.org/api/kotlinx.serialization/"))
            packageListUrl.set(
                rootProject.projectDir.resolve("serialization.package.list").toURL()
            )
        }
    }
}
```

【Groovy】

```
import org.jetbrains.dokka.gradle.DokkaTask
import java.net.URL

// Note: To configure multi-project builds, you need
//       to configure Partial tasks of the subprojects.
//       See "Configuration example" section of documentation.
tasks.withType(DokkaTask.class) {
    // ..
    // general configuration section
    // ..

    dokkaSourceSets.configureEach {
        // ..
        // source set configuration section
        // ..

        externalDocumentationLink {
            url.set(new URL("https://kotlinlang.org/api/kotlinx.serialization/"))
```

```
            packageListUrl.set(
                file("serialization.package.list").toURL()
            )
        }
    }
}
```

The root URL of documentation to link to. It **must** contain a trailing slash.

Dokka does its best to automatically find `package-list` for the given URL, and link declarations together.

If automatic resolution fails or if you want to use locally cached files instead, consider setting the `packageListUrl` option.

The exact location of a `package-list`. This is an alternative to relying on Dokka automatically resolving it.

Package lists contain information about the documentation and the project itself, such as module and package names.

This can also be a locally cached file to avoid network calls.

Complete configuration

Below you can see all possible configuration options applied at the same time.

> 【Kotlin】

```
import org.jetbrains.dokka.DokkaConfiguration.Visibility
import org.jetbrains.dokka.gradle.DokkaTask
import org.jetbrains.dokka.Platform
import java.net.URL

// Note: To configure multi-project builds, you need
//       to configure Partial tasks of the subprojects.
//       See "Configuration example" section of documentation.
tasks.withType<DokkaTask>().configureEach {
    moduleName.set(project.name)
    moduleVersion.set(project.version.toString())
    outputDirectory.set(layout.buildDirectory.dir("dokka/$name"))
    failOnWarning.set(false)
    suppressObviousFunctions.set(true)
    suppressInheritedMembers.set(false)
    offlineMode.set(false)
```

```
    dokkaSourceSets {
        named("linux") {
            dependsOn("native")
            sourceRoots.from(file("linux/src"))
        }
        configureEach {
            suppress.set(false)
            displayName.set(name)
            documentedVisibilities.set(setOf(Visibility.PUBLIC))
            reportUndocumented.set(false)
            skipEmptyPackages.set(true)
            skipDeprecated.set(false)
            suppressGeneratedFiles.set(true)
            jdkVersion.set(8)
            languageVersion.set("1.7")
            apiVersion.set("1.7")
            noStdlibLink.set(false)
            noJdkLink.set(false)
            noAndroidSdkLink.set(false)
            includes.from(project.files(), "packages.md", "extra.md")
            platform.set(Platform.DEFAULT)
            sourceRoots.from(file("src"))
            classpath.from(project.files(), file("libs/dependency.jar"))
            samples.from(project.files(), "samples/Basic.kt", "samples/Advanced.kt")

            sourceLink {
                localDirectory.set(projectDir.resolve("src"))
                remoteUrl.set(URL("https://github.com/kotlin/dokka/tree/master/src"))
                remoteLineSuffix.set("#L")
            }

            externalDocumentationLink {
                url.set(URL("https://kotlinlang.org/api/latest/jvm/stdlib/"))
                packageListUrl.set(
                    rootProject.projectDir.resolve("stdlib.package.list").toURL()
                )
            }

            perPackageOption {
                matchingRegex.set(".*api.*")
                suppress.set(false)
                skipDeprecated.set(false)
                reportUndocumented.set(false)
                documentedVisibilities.set(
                    setOf(
                        Visibility.PUBLIC,
                        Visibility.PRIVATE,
                        Visibility.PROTECTED,
                        Visibility.INTERNAL,
                        Visibility.PACKAGE
                    )
                )
```

```
            }
        }
    }
}
```

> 【Groovy】

```
import org.jetbrains.dokka.DokkaConfiguration.Visibility
import org.jetbrains.dokka.gradle.DokkaTask
import org.jetbrains.dokka.Platform
import java.net.URL

// Note: To configure multi-project builds, you need
//       to configure Partial tasks of the subprojects.
//       See "Configuration example" section of documentation.
tasks.withType(DokkaTask.class) {
    moduleName.set(project.name)
    moduleVersion.set(project.version.toString())
    outputDirectory.set(file("build/dokka/$name"))
    failOnWarning.set(false)
    suppressObviousFunctions.set(true)
    suppressInheritedMembers.set(false)
    offlineMode.set(false)

    dokkaSourceSets {
        named("linux") {
            dependsOn("native")
            sourceRoots.from(file("linux/src"))
        }
        configureEach {
            suppress.set(false)
            displayName.set(name)
            documentedVisibilities.set([Visibility.PUBLIC])
            reportUndocumented.set(false)
            skipEmptyPackages.set(true)
            skipDeprecated.set(false)
            suppressGeneratedFiles.set(true)
            jdkVersion.set(8)
            languageVersion.set("1.7")
            apiVersion.set("1.7")
            noStdlibLink.set(false)
            noJdkLink.set(false)
            noAndroidSdkLink.set(false)
            includes.from(project.files(), "packages.md", "extra.md")
            platform.set(Platform.DEFAULT)
            sourceRoots.from(file("src"))
            classpath.from(project.files(), file("libs/dependency.jar"))
            samples.from(project.files(), "samples/Basic.kt", "samples/Advanced.kt")

            sourceLink {
```

```
                localDirectory.set(file("src"))
                remoteUrl.set(new URL("https://github.com/kotlin/dokka/tree/master/src"
))
                remoteLineSuffix.set("#L")
            }

            externalDocumentationLink {
                url.set(new URL("https://kotlinlang.org/api/latest/jvm/stdlib/"))
                packageListUrl.set(
                        file("stdlib.package.list").toURL()
                )
            }

            perPackageOption {
                matchingRegex.set(".*api.*")
                suppress.set(false)
                skipDeprecated.set(false)
                reportUndocumented.set(false)
                documentedVisibilities.set([Visibility.PUBLIC])
            }
        }
    }
}
```

Maven

To generate documentation for a Maven-based project, you can use the Maven plugin for Dokka.

> Compared to the Gradle plugin for Dokka, the Maven plugin has only basic features and does not provide support for multi-module builds.
>
> 

You can play around with Dokka and see how it can be configured for a Maven project by visiting our Maven example project.

Apply Dokka

To apply Dokka, you need to add `dokka-maven-plugin` to the `plugins` section of your POM file:

```
<build>
    <plugins>
        <plugin>
            <groupId>org.jetbrains.dokka</groupId>
            <artifactId>dokka-maven-plugin</artifactId>
            <version>2.0.0</version>
            <executions>
                <execution>
                    <phase>pre-site</phase>
                    <goals>
                        <goal>dokka</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

Generate documentation

The following goals are provided by the Maven plugin:

Goal	Description
`dokka:dokka`	Generates documentation with Dokka plugins applied. HTML format by default.

Experimental

Goal	Description
`dokka:javadoc`	Generates documentation in Javadoc format.
`dokka:javadocJar`	Generates a `javadoc.jar` file that contains documentation in Javadoc format.

Other output formats

By default, the Maven plugin for Dokka builds documentation in HTML output format.

All other output formats are implemented as Dokka plugins. In order to generate documentation in the desired format, you have to add it as a Dokka plugin to the configuration.

For example, to use the experimental GFM format, you have to add `gfm-plugin` artifact:

```
<plugin>
    <groupId>org.jetbrains.dokka</groupId>
    <artifactId>dokka-maven-plugin</artifactId>
    ...
    <configuration>
        <dokkaPlugins>
            <plugin>
                <groupId>org.jetbrains.dokka</groupId>
                <artifactId>gfm-plugin</artifactId>
                <version>2.0.0</version>
            </plugin>
        </dokkaPlugins>
    </configuration>
</plugin>
```

With this configuration, running the `dokka:dokka` goal produces documentation in GFM format.

To learn more about Dokka plugins, see Dokka plugins.

Build javadoc.jar

If you want to publish your library to a repository, you may need to provide a `javadoc.jar` file that contains API reference documentation of your library.

For example, if you want to publish to Maven Central, you must supply a `javadoc.jar` alongside your project. However, not all repositories have that rule.

Unlike the Gradle plugin for Dokka, the Maven plugin comes with a ready-to-use `dokka:javadocJar` goal. By default, it generates documentation in Javadoc output format in the `target` folder.

If you are not satisfied with the built-in goal or want to customize the output (for example, you want to generate documentation in HTML format instead of Javadoc), similar behavior can be achieved by adding the Maven JAR plugin with the following configuration:

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-jar-plugin</artifactId>
    <version>3.3.0</version>
    <executions>
        <execution>
            <goals>
                <goal>test-jar</goal>
            </goals>
        </execution>
        <execution>
            <id>dokka-jar</id>
            <phase>package</phase>
            <goals>
                <goal>jar</goal>
            </goals>
            <configuration>
                <classifier>dokka</classifier>
                <classesDirectory>${project.build.directory}/dokka</classesDirectory>
                <skipIfEmpty>true</skipIfEmpty>
            </configuration>
        </execution>
    </executions>
</plugin>
```

The documentation and the `.jar` archive for it are then generated by running `dokka:dokka` and `jar:jar@dokka-jar` goals:

```
mvn dokka:dokka jar:jar@dokka-jar
```

> If you publish your library to Maven Central, you can use services like javadoc.io to host your library's API documentation for free and without any setup. It takes documentation pages straight from the `javadoc.jar`. It works well with the HTML format as demonstrated in this example.

Configuration example

Maven's plugin configuration block can be used to configure Dokka.

Here is an example of a basic configuration that only changes the output location of your documentation:

```
<plugin>
    <groupId>org.jetbrains.dokka</groupId>
    <artifactId>dokka-maven-plugin</artifactId>
    ...
    <configuration>
        <outputDir>${project.basedir}/target/documentation/dokka</outputDir>
    </configuration>
</plugin>
```

Configuration options

Dokka has many configuration options to tailor your and your reader's experience.

Below are some examples and detailed descriptions for each configuration section. You can also find an example with all configuration options applied at the bottom of the page.

General configuration

```
<plugin>
    <groupId>org.jetbrains.dokka</groupId>
    <artifactId>dokka-maven-plugin</artifactId>
    <!--  ...  -->
    <configuration>
        <skip>false</skip>
        <moduleName>${project.artifactId}</moduleName>
        <outputDir>${project.basedir}/target/documentation</outputDir>
        <failOnWarning>false</failOnWarning>
        <suppressObviousFunctions>true</suppressObviousFunctions>
        <suppressInheritedMembers>false</suppressInheritedMembers>
        <offlineMode>false</offlineMode>
        <sourceDirectories>
            <dir>${project.basedir}/src</dir>
        </sourceDirectories>
        <documentedVisibilities>
            <visibility>PUBLIC</visibility>
            <visibility>PROTECTED</visibility>
        </documentedVisibilities>
        <reportUndocumented>false</reportUndocumented>
        <skipDeprecated>false</skipDeprecated>
        <skipEmptyPackages>true</skipEmptyPackages>
        <suppressedFiles>
            <file>/path/to/dir</file>
```

```
            <file>/path/to/file</file>
        </suppressedFiles>
        <jdkVersion>8</jdkVersion>
        <languageVersion>1.7</languageVersion>
        <apiVersion>1.7</apiVersion>
        <noStdlibLink>false</noStdlibLink>
        <noJdkLink>false</noJdkLink>
        <includes>
            <include>packages.md</include>
            <include>extra.md</include>
        </includes>
        <classpath>${project.compileClasspathElements}</classpath>
        <samples>
            <dir>${project.basedir}/samples</dir>
        </samples>
        <sourceLinks>
            <!-- Separate section -->
        </sourceLinks>
        <externalDocumentationLinks>
            <!-- Separate section -->
        </externalDocumentationLinks>
        <perPackageOptions>
            <!-- Separate section -->
        </perPackageOptions>
    </configuration>
</plugin>
```

Whether to skip documentation generation.

Default: `false`

The display name used to refer to the project/module. It's used for the table of contents, navigation, logging, etc.

Default: `{project.artifactId}`

The directory to where documentation is generated, regardless of format.

Default: `{project.basedir}/target/dokka`

Whether to fail documentation generation if Dokka has emitted a warning or an error. The process waits until all errors and warnings have been emitted first.

This setting works well with `reportUndocumented`.

Default: `false`

Whether to suppress obvious functions.

A function is considered to be obvious if it is:

- Inherited from `kotlin.Any`, `Kotlin.Enum`, `java.lang.Object` or `java.lang.Enum`, such as `equals`, `hashCode`, `toString`.
- Synthetic (generated by the compiler) and does not have any documentation, such as `dataClass.componentN` or `dataClass.copy`.

Default: `true`

Whether to suppress inherited members that aren't explicitly overridden in a given class.

Note: This can suppress functions such as `equals` / `hashCode` / `toString`, but cannot suppress synthetic functions such as `dataClass.componentN` and `dataClass.copy`. Use `suppressObviousFunctions` for that.

Default: `false`

Whether to resolve remote files/links over your network.

This includes package-lists used for generating external documentation links. For example, to make classes from the standard library clickable.

Setting this to `true` can significantly speed up build times in certain cases, but can also worsen documentation quality and user experience. For example, by not resolving class/member links from your dependencies, including the standard library.

Note: You can cache fetched files locally and provide them to Dokka as local paths. See `externalDocumentationLinks` section.

Default: `false`

The source code roots to be analyzed and documented. Acceptable inputs are directories and individual `.kt` / `.java` files.

Default: `{project.compileSourceRoots}`

The set of visibility modifiers that should be documented.

This can be used if you want to document `protected` / `internal` / `private` declarations, as well as if you want to exclude `public` declarations and only document internal API.

Can be configured on per-package basis.

Default: `PUBLIC`

Whether to emit warnings about visible undocumented declarations, that is declarations without KDocs after they have been filtered by `documentedVisibilities` and other filters.

This setting works well with `failOnWarning`.

This can be overridden at package level.

Default: `false`

Whether to document declarations annotated with `@Deprecated`.

This can be overridden at package level.

Default: `false`

Whether to skip packages that contain no visible declarations after various filters have been applied.

For example, if `skipDeprecated` is set to `true` and your package contains only deprecated declarations, it is considered to be empty.

Default: `true`

The directories or individual files that should be suppressed, meaning that declarations from them are not documented.

The JDK version to use when generating external documentation links for Java types.

For example, if you use `java.util.UUID` in some public declaration signature, and this option is set to `8`, Dokka generates an external documentation link to JDK 8 Javadocs for it.

Default: JDK 8

The Kotlin language version used for setting up analysis and @sample environment.

By default, the latest language version available to Dokka's embedded compiler is used.

The Kotlin API version used for setting up analysis and @sample environment.

By default, it is deduced from `languageVersion`.

Whether to generate external documentation links that lead to the API reference documentation of Kotlin's standard library.

Note: Links **are** generated when `noStdLibLink` is set to `false`.

Default: `false`

Whether to generate external documentation links to JDK's Javadocs.

The version of JDK Javadocs is determined by the `jdkVersion` option.

Note: Links **are** generated when `noJdkLink` is set to `false`.

Default: `false`

A list of Markdown files that contain module and package documentation

The contents of specified files are parsed and embedded into documentation as module and package descriptions.

The classpath for analysis and interactive samples.

This is useful if some types that come from dependencies are not resolved/picked up automatically. This option accepts both `.jar` and `.klib` files.

Default: `{project.compileClasspathElements}`

A list of directories or files that contain sample functions which are referenced via @sample KDoc tag.

Source link configuration

The `sourceLinks` configuration block allows you to add a `source` link to each signature that leads to the `url` with a specific line number. (The line number is configurable by setting `lineSuffix`).

This helps readers to find the source code for each declaration.

For an example, see the documentation for the `count()` function in `kotlinx.coroutines`.

```
<plugin>
    <groupId>org.jetbrains.dokka</groupId>
    <artifactId>dokka-maven-plugin</artifactId>
    <!--  ...  -->
    <configuration>
        <sourceLinks>
            <link>
                <path>src</path>
                <url>https://github.com/kotlin/dokka/tree/master/src</url>
                <lineSuffix>#L</lineSuffix>
            </link>
        </sourceLinks>
    </configuration>
</plugin>
```

The path to the local source directory. The path must be relative to the root of the current module.

Note: Only Unix based paths are allowed, Windows-style paths will throw an error.

The URL of the source code hosting service that can be accessed by documentation readers, like GitHub, GitLab, Bitbucket, etc. This URL is used to generate source code links of declarations.

The suffix used to append source code line number to the URL. This helps readers navigate not only to the file, but to the specific line number of the declaration.

The number itself is appended to the specified suffix. For example, if this option is set to `#L` and the line number is 10, the resulting URL suffix is `#L10`.

Suffixes used by popular services:

- GitHub: `#L`
- GitLab: `#L`
- Bitbucket: `#lines-`

External documentation links configuration

The `externalDocumentationLinks` block allows the creation of links that lead to the externally hosted documentation of your dependencies.

For example, if you are using types from `kotlinx.serialization`, by default they are unclickable in your documentation, as if they are unresolved. However, since the API reference documentation for `kotlinx.serialization` is built by Dokka and is published on kotlinlang.org, you can configure external documentation links for it. Thus allowing Dokka to generate links for types from the library, making them resolve successfully and clickable.

By default, external documentation links for Kotlin standard library and JDK are configured.

```
<plugin>
    <groupId>org.jetbrains.dokka</groupId>
    <artifactId>dokka-maven-plugin</artifactId>
    <!--  ...  -->
    <configuration>
        <externalDocumentationLinks>
            <link>
                <url>https://kotlinlang.org/api/kotlinx.serialization/</url>
                <packageListUrl>file:/${project.basedir}/serialization.package.list</p
ackageListUrl>
            </link>
        </externalDocumentationLinks>
    </configuration>
</plugin>
```

The root URL of documentation to link to. It **must** contain a trailing slash.

Dokka does its best to automatically find the `package-list` for the given URL, and link declarations together.

If automatic resolution fails or if you want to use locally cached files instead, consider setting the `packageListUrl` option.

The exact location of a `package-list`. This is an alternative to relying on Dokka automatically resolving it.

Package lists contain information about the documentation and the project itself, such as module and package names.

This can also be a locally cached file to avoid network calls.

Package options

The `perPackageOptions` configuration block allows setting some options for specific packages matched by `matchingRegex`.

```
<plugin>
    <groupId>org.jetbrains.dokka</groupId>
    <artifactId>dokka-maven-plugin</artifactId>
    <!--  ...  -->
    <configuration>
        <perPackageOptions>
            <packageOptions>
                <matchingRegex>.*api.*</matchingRegex>
                <suppress>false</suppress>
                <reportUndocumented>false</reportUndocumented>
                <skipDeprecated>false</skipDeprecated>
                <documentedVisibilities>
                    <visibility>PUBLIC</visibility>
                    <visibility>PRIVATE</visibility>
                    <visibility>PROTECTED</visibility>
                    <visibility>INTERNAL</visibility>
                    <visibility>PACKAGE</visibility>
                </documentedVisibilities>
            </packageOptions>
        </perPackageOptions>
    </configuration>
</plugin>
```

The regular expression that is used to match the package.

Default: `.*`

Whether this package should be skipped when generating documentation.

Default: `false`

The set of visibility modifiers that should be documented.

This can be used if you want to document `protected` / `internal` / `private` declarations within this package, as well as if you want to exclude `public` declarations and only document internal API.

Default: `PUBLIC`

Whether to document declarations annotated with `@Deprecated`.

This can be set on project/module level.

Default: `false`

Whether to emit warnings about visible undocumented declarations, that is declarations without KDocs after they have been filtered by `documentedVisibilities` and other filters.

This setting works well with `failOnWarning`.

Default: `false`

Complete configuration

Below you can see all the possible configuration options applied at the same time.

```
<plugin>
    <groupId>org.jetbrains.dokka</groupId>
    <artifactId>dokka-maven-plugin</artifactId>
    <!--  ...  -->
    <configuration>
        <skip>false</skip>
        <moduleName>${project.artifactId}</moduleName>
        <outputDir>${project.basedir}/target/documentation</outputDir>
        <failOnWarning>false</failOnWarning>
        <suppressObviousFunctions>true</suppressObviousFunctions>
        <suppressInheritedMembers>false</suppressInheritedMembers>
        <offlineMode>false</offlineMode>
        <sourceDirectories>
            <dir>${project.basedir}/src</dir>
        </sourceDirectories>
        <documentedVisibilities>
            <visibility>PUBLIC</visibility>
            <visibility>PRIVATE</visibility>
            <visibility>PROTECTED</visibility>
            <visibility>INTERNAL</visibility>
            <visibility>PACKAGE</visibility>
        </documentedVisibilities>
        <reportUndocumented>false</reportUndocumented>
        <skipDeprecated>false</skipDeprecated>
        <skipEmptyPackages>true</skipEmptyPackages>
        <suppressedFiles>
            <file>/path/to/dir</file>
            <file>/path/to/file</file>
        </suppressedFiles>
        <jdkVersion>8</jdkVersion>
        <languageVersion>1.7</languageVersion>
        <apiVersion>1.7</apiVersion>
        <noStdlibLink>false</noStdlibLink>
        <noJdkLink>false</noJdkLink>
        <includes>
            <include>packages.md</include>
```

```
            <include>extra.md</include>
        </includes>
        <classpath>${project.compileClasspathElements}</classpath>
        <samples>
            <dir>${project.basedir}/samples</dir>
        </samples>
        <sourceLinks>
            <link>
                <path>src</path>
                <url>https://github.com/kotlin/dokka/tree/master/src</url>
                <lineSuffix>#L</lineSuffix>
            </link>
        </sourceLinks>
        <externalDocumentationLinks>
            <link>
                <url>https://kotlinlang.org/api/latest/jvm/stdlib/</url>
                <packageListUrl>file:/${project.basedir}/stdlib.package.list</packageL
istUrl>
            </link>
        </externalDocumentationLinks>
        <perPackageOptions>
            <packageOptions>
                <matchingRegex>.*api.*</matchingRegex>
                <suppress>false</suppress>
                <reportUndocumented>false</reportUndocumented>
                <skipDeprecated>false</skipDeprecated>
                <documentedVisibilities>
                    <visibility>PUBLIC</visibility>
                    <visibility>PRIVATE</visibility>
                    <visibility>PROTECTED</visibility>
                    <visibility>INTERNAL</visibility>
                    <visibility>PACKAGE</visibility>
                </documentedVisibilities>
            </packageOptions>
        </perPackageOptions>
    </configuration>
</plugin>
```

CLI

If for some reason you cannot use Gradle or Maven build tools, Dokka has a command line (CLI) runner for generating documentation.

In comparison, it has the same, if not more, capabilities as the Gradle plugin for Dokka. Although it is considerably more difficult to set up as there is no autoconfiguration, especially in multiplatform and multi-module environments.

Get started

The CLI runner is published to Maven Central as a separate runnable artifact.

You can find it on Maven Central or download it directly.

With the `dokka-cli-2.0.0.jar` file saved on your computer, run it with the `-help` option to see all available configuration options and their description:

```
java -jar dokka-cli-2.0.0.jar -help
```

It also works for some nested options, such as `-sourceSet`:

```
java -jar dokka-cli-2.0.0.jar -sourceSet -help
```

Generate documentation

Prerequisites

Since there is no build tool to manage dependencies, you have to provide dependency `.jar` files yourself.

Listed below are the dependencies that you need for any output format:

Group	Artifact	Version	Link
`org.jetbrains.dokka`	`dokka-base`	2.0.0	download
`org.jetbrains.dokka`	`analysis-kotlin-descriptors`	2.0.0	download

Below are the additional dependencies that you need for HTML output format:

Group	Artifact	Version	Link
`org.jetbrains.kotlinx`	`kotlinx-html-jvm`	0.8.0	download
`org.freemarker`	`freemarker`	2.3.31	download

Run with command line options

You can pass command line options to configure the CLI runner.

At the very least you need to provide the following options:

- `-pluginsClasspath` - a list of absolute/relative paths to downloaded dependencies, separated by semi-colons `;`
- `-sourceSet` - an absolute path to code sources to generate documentation for
- `-outputDir` - an absolute/relative path of the documentation output directory

```
java -jar dokka-cli-2.0.0.jar \
     -pluginsClasspath "./dokka-base-2.0.0.jar;./analysis-kotlin-descriptors-2.0.0.jar
;./kotlinx-html-jvm-0.8.0.jar;./freemarker-2.3.31.jar" \
     -sourceSet "-src /home/myCoolProject/src/main/kotlin" \
     -outputDir "./dokka/html"
```

Executing the given example generates documentation in HTML output format.

See Command line options for more configuration details.

Run with JSON configuration

It's possible to configure the CLI runner with JSON. In this case, you need to provide an absolute/relative path to the JSON configuration file as the first and only argument. All other configuration options are parsed from it.

```
java -jar dokka-cli-2.0.0.jar dokka-configuration.json
```

At the very least, you need the following JSON configuration file:

```
{
  "outputDir": "./dokka/html",
  "sourceSets": [
    {
      "sourceSetID": {
        "scopeId": "moduleName",
        "sourceSetName": "main"
      },
      "sourceRoots": [
```

```
        "/home/myCoolProject/src/main/kotlin"
      ]
    }
  ],
  "pluginsClasspath": [
    "./dokka-base-2.0.0.jar",
    "./kotlinx-html-jvm-0.8.0.jar",
    "./analysis-kotlin-descriptors-2.0.0.jar",
    "./freemarker-2.3.31.jar"
  ]
}
```

See JSON configuration options for more details.

Other output formats

By default, the `dokka-base` artifact contains the HTML output format only.

All other output formats are implemented as Dokka plugins. In order to use them, you have to put them on the plugins classpath.

For example, if you want to generate documentation in the experimental GFM output format, you need to download and pass gfm-plugin's JAR (download) into the `pluginsClasspath` configuration option.

Via command line options:

```
java -jar dokka-cli-2.0.0.jar \
     -pluginsClasspath "./dokka-base-2.0.0.jar;...;./gfm-plugin-2.0.0.jar" \
     ...
```

Via JSON configuration:

```
{
  ...
  "pluginsClasspath": [
    "./dokka-base-2.0.0.jar",
    "...",
    "./gfm-plugin-2.0.0.jar"
  ],
  ...
}
```

With the GFM plugin passed to `pluginsClasspath`, the CLI runner generates documentation in the GFM output format.

For more information, see Markdown and Javadoc pages.

Command line options

To see the list of all possible command line options and their detailed description, run:

```
java -jar dokka-cli-2.0.0.jar -help
```

Short summary:

Option	Description
`moduleName`	Name of the project/module.
`moduleVersion`	Documented version.
`outputDir`	Output directory path, `./dokka` by default.
`sourceSet`	Configuration for a Dokka source set. Contains nested configura options.
`pluginsConfiguration`	Configuration for Dokka plugins.
`pluginsClasspath`	List of jars with Dokka plugins and their dependencies. Accepts multiple paths separated by semicolons.
`offlineMode`	Whether to resolve remote files/links over network.
`failOnWarning`	Whether to fail documentation generation if Dokka has emitted a warning or an error.
`delayTemplateSubstitution`	Whether to delay substitution of some elements. Used in increm builds of multi-module projects.
`noSuppressObviousFunctions`	Whether to suppress obvious functions such as those inherited fr `kotlin.Any` and `java.lang.Object`.
`includes`	Markdown files that contain module and package documentation Accepts multiple values separated by semicolons.
`suppressInheritedMembers`	Whether to suppress inherited members that aren't explicitly overridden in a given class.
`globalPackageOptions`	Global list of package configuration options in format `"matchingRegex,-deprecated,-privateApi,+warnUndocumented,+suppress;+visibility:PUBLIC` Accepts multiple values separated by semicolons.
`globalLinks`	Global external documentation links in format `{url}^{packageListUrl}`. Accepts multiple values separated by
`globalSrcLink`	Global mapping between a source directory and a Web service fo browsing the code. Accepts multiple paths separated by semicol
`helpSourceSet`	Prints help for the nested `-sourceSet` configuration.
`loggingLevel`	Logging level, possible values: `DEBUG, PROGRESS, INFO, WARN, ERROR`.

`help, h`	Usage info.

Source set options

To see the list of command line options for the nested `-sourceSet` configuration, run:

```
java -jar dokka-cli-2.0.0.jar -sourceSet -help
```

Short summary:

Option	Description
`sourceSetName`	Name of the source set.
`displayName`	Display name of the source set, used both internally and externally.
`classpath`	Classpath for analysis and interactive samples. Accepts multiple paths separated by semicolons.
`src`	Source code roots to be analyzed and documented. Accepts multiple paths separated by semicolons.
`dependentSourceSets`	Names of the dependent source sets in format `moduleName/sourceSetName`. Accepts multiple values separated by semicolons.
`samples`	List of directories or files that contain sample functions. Accepts multiple paths separated by semicolons.
`includes`	Markdown files that contain module and package documentation. Accepts multiple paths separated by semicolons.
`documentedVisibilities`	Visibilities to be documented. Accepts multiple values separated by semicolons. Possible values: `PUBLIC`, `PRIVATE`, `PROTECTED`, `INTERNAL`, `PACKAGE`.
`reportUndocumented`	Whether to report undocumented declarations.
`noSkipEmptyPackages`	Whether to create pages for empty packages.
`skipDeprecated`	Whether to skip deprecated declarations.
`jdkVersion`	Version of JDK to use for linking to JDK Javadocs.
`languageVersion`	Language version used for setting up analysis and samples.
`apiVersion`	Kotlin API version used for setting up analysis and samples.
`noStdlibLink`	Whether to generate links to the Kotlin standard library.
`noJdkLink`	Whether to generate links to JDK Javadocs.
`suppressedFiles`	Paths to files to be suppressed. Accepts multiple paths

`suppressedFiles`	separated by semicolons.
`analysisPlatform`	Platform used for setting up analysis.
`perPackageOptions`	List of package source set configurations in format `matchingRegexp,-deprecated,-privateApi,+warnUndocumented,+suppress;...`. Accepts multiple values separated by semicolons.
`externalDocumentationLinks`	External documentation links in format `{url}^{packageListUrl}`. Accepts multiple values separated by `^^`.
`srcLink`	Mapping between a source directory and a Web service for browsing the code. Accepts multiple paths separated by semicolons.

JSON configuration

Below are some examples and detailed descriptions for each configuration section. You can also find an example with all configuration options applied at the bottom of the page.

General configuration

```
{
  "moduleName": "Dokka Example",
  "moduleVersion": null,
  "outputDir": "./build/dokka/html",
  "failOnWarning": false,
  "suppressObviousFunctions": true,
  "suppressInheritedMembers": false,
  "offlineMode": false,
  "includes": [
    "module.md"
  ],
  "sourceLinks":  [
    { "_comment": "Options are described in a separate section" }
  ],
  "perPackageOptions": [
    { "_comment": "Options are described in a separate section" }
  ],
  "externalDocumentationLinks":  [
    { "_comment": "Options are described in a separate section" }
  ],
  "sourceSets": [
    { "_comment": "Options are described in a separate section" }
  ],
  "pluginsClasspath": [
    "./dokka-base-2.0.0.jar",
```

```
      "./analysis-kotlin-descriptors-2.0.0.jar",
      "./freemarker-2.3.31.jar"
    ]
  }
```

The display name used to refer to the module. It is used for the table of contents, navigation, logging, etc.

Default: `root`

The module version.

Default: empty

The directory to where documentation is generated, regardless of output format.

Default: `./dokka`

Whether to fail documentation generation if Dokka has emitted a warning or an error. The process waits until all errors and warnings have been emitted first.

This setting works well with `reportUndocumented`

Default: `false`

Whether to suppress obvious functions.

A function is considered to be obvious if it is:

- Inherited from `kotlin.Any`, `Kotlin.Enum`, `java.lang.Object` or `java.lang.Enum`, such as `equals`, `hashCode`, `toString`.
- Synthetic (generated by the compiler) and does not have any documentation, such as `dataClass.componentN` or `dataClass.copy`.

Default: `true`

Whether to suppress inherited members that aren't explicitly overridden in a given class.

Note: This can suppress functions such as `equals` / `hashCode` / `toString`, but cannot suppress synthetic functions such as `dataClass.componentN` and `dataClass.copy`. Use `suppressObviousFunctions` for that.

Default: `false`

Whether to resolve remote files/links over your network.

This includes package-lists used for generating external documentation links. For example, to make classes from the standard library clickable.

Setting this to `true` can significantly speed up build times in certain cases, but can also worsen documentation quality and user experience. For example, by not resolving class/member links from your dependencies, including the standard library.

Note: You can cache fetched files locally and provide them to Dokka as local paths. See `externalDocumentationLinks` section.

Default: `false`

A list of Markdown files that contain module and package documentation.

The contents of specified files are parsed and embedded into documentation as module and package descriptions.

This can be configured on per-package basis.

Individual and additional configuration of Kotlin source sets.

For a list of possible options, see source set configuration.

The global configuration of source links that is applied for all source sets.

For a list of possible options, see source link configuration.

The global configuration of matched packages, regardless of the source set they are in.

For a list of possible options, see per-package configuration.

The global configuration of external documentation links, regardless of the source set they are used in.

For a list of possible options, see external documentation links configuration.

A list of JAR files with Dokka plugins and their dependencies.

Source set configuration

How to configure Kotlin source sets:

```
{
  "sourceSets": [
    {
      "displayName": "jvm",
      "sourceSetID": {
        "scopeId": "moduleName",
        "sourceSetName": "main"
      },
      "dependentSourceSets": [
        {
          "scopeId": "dependentSourceSetScopeId",
```

```
            "sourceSetName": "dependentSourceSetName"
          }
        ],
        "documentedVisibilities": ["PUBLIC", "PRIVATE", "PROTECTED", "INTERNAL", "PACKAG
E"],
        "reportUndocumented": false,
        "skipEmptyPackages": true,
        "skipDeprecated": false,
        "jdkVersion": 8,
        "languageVersion": "1.7",
        "apiVersion": "1.7",
        "noStdlibLink": false,
        "noJdkLink": false,
        "includes": [
          "module.md"
        ],
        "analysisPlatform": "jvm",
        "sourceRoots": [
          "/home/ignat/IdeaProjects/dokka-debug-mvn/src/main/kotlin"
        ],
        "classpath": [
          "libs/kotlin-stdlib-2.1.21.jar",
          "libs/kotlin-stdlib-common-2.1.21.jar"
        ],
        "samples": [
          "samples/basic.kt"
        ],
        "suppressedFiles": [
          "src/main/kotlin/org/jetbrains/dokka/Suppressed.kt"
        ],
        "sourceLinks":  [
          { "_comment": "Options are described in a separate section" }
        ],
        "perPackageOptions": [
          { "_comment": "Options are described in a separate section" }
        ],
        "externalDocumentationLinks":  [
          { "_comment": "Options are described in a separate section" }
        ]
      }
    ]
}
```


The display name used to refer to this source set.

The name is used both externally (for example, the source set name is visible to documentation readers) and internally (for example, for logging messages of `reportUndocumented`).

The platform name can be used if you don't have a better alternative.

The technical ID of the source set

The set of visibility modifiers that should be documented.

This can be used if you want to document `protected` / `internal` / `private` declarations, as well as if you want to exclude `public` declarations and only document internal API.

This can be configured on per-package basis.

Possible values:

- `PUBLIC`
- `PRIVATE`
- `PROTECTED`
- `INTERNAL`
- `PACKAGE`

Default: `PUBLIC`

Whether to emit warnings about visible undocumented declarations, that is declarations without KDocs after they have been filtered by `documentedVisibilities` and other filters.

This setting works well with `failOnWarning`.

This can be configured on per-package basis.

Default: `false`

Whether to skip packages that contain no visible declarations after various filters have been applied.

For example, if `skipDeprecated` is set to `true` and your package contains only deprecated declarations, it is considered to be empty.

Default for CLI runner is `false`.

Whether to document declarations annotated with `@Deprecated`.

This can be configured on per-package basis.

Default: `false`

The JDK version to use when generating external documentation links for Java types.

For example, if you use `java.util.UUID` in some public declaration signature, and this option is set to `8`, Dokka generates an external documentation link to JDK 8 Javadocs for it.

The Kotlin language version used for setting up analysis and @sample environment.

The Kotlin API version used for setting up analysis and @sample environment.

Whether to generate external documentation links that lead to the API reference documentation of Kotlin's standard library.

Note: Links **are** generated when `noStdLibLink` is set to `false`.

Default: `false`

Whether to generate external documentation links to JDK's Javadocs.

The version of JDK Javadocs is determined by the `jdkVersion` option.

Note: Links **are** generated when `noJdkLink` is set to `false`.

Default: `false`

A list of Markdown files that contain module and package documentation.

The contents of the specified files are parsed and embedded into documentation as module and package descriptions.

Platform to be used for setting up code analysis and @sample environment.

Possible values:

- `jvm`
- `common`
- `js`
- `native`

The source code roots to be analyzed and documented. Acceptable inputs are directories and individual `.kt` / `.java` files.

The classpath for analysis and interactive samples.

This is useful if some types that come from dependencies are not resolved/picked up automatically.

This option accepts both `.jar` and `.klib` files.

A list of directories or files that contain sample functions which are referenced via the @sample KDoc tag.

The files to be suppressed when generating documentation.

A set of parameters for source links that is applied only for this source set.

For a list of possible options, see source link configuration.

A set of parameters specific to matched packages within this source set.

For a list of possible options, see per-package configuration.

A set of parameters for external documentation links that is applied only for this source set.

For a list of possible options, see external documentation links configuration.

Source link configuration

The `sourceLinks` configuration block allows you to add a `source` link to each signature that leads to the `remoteUrl` with a specific line number. (The line number is configurable by setting `remoteLineSuffix`).

This helps readers to find the source code for each declaration.

For an example, see the documentation for the `count()` function in `kotlinx.coroutines`.

You can configure source links for all source sets together at the same time, or individually:

```
{
  "sourceLinks": [
    {
      "localDirectory": "src/main/kotlin",
      "remoteUrl": "https://github.com/Kotlin/dokka/tree/master/src/main/kotlin",
      "remoteLineSuffix": "#L"
    }
  ]
}
```

The path to the local source directory.

The URL of the source code hosting service that can be accessed by documentation readers, like GitHub, GitLab, Bitbucket, etc. This URL is used to generate source code links of declarations.

The suffix used to append the source code line number to the URL. This helps readers navigate not only to the file, but to the specific line number of the declaration.

The number itself is appended to the specified suffix. For example, if this option is set to `#L` and the line number is 10, the resulting URL suffix is `#L10`.

Suffixes used by popular services:

- GitHub: `#L`
- GitLab: `#L`
- Bitbucket: `#lines-`

Default: empty (no suffix)

Per-package configuration

The `perPackageOptions` configuration block allows setting some options for specific packages matched by `matchingRegex`.

You can add package configurations for all source sets together at the same time, or individually:

```
{
  "perPackageOptions": [
    {
      "matchingRegex": ".*internal.*",
      "suppress": false,
      "skipDeprecated": false,
      "reportUndocumented": false,
      "documentedVisibilities": ["PUBLIC", "PRIVATE", "PROTECTED", "INTERNAL", "PACKAG
E"]
    }
  ]
}
```

The regular expression that is used to match the package.

Whether this package should be skipped when generating documentation.

Default: `false`

Whether to document declarations annotated with `@Deprecated`.

This can be set on project/module level.

Default: `false`

Whether to emit warnings about visible undocumented declarations, that is declarations without KDocs after they have been filtered by `documentedVisibilities` and other filters.

This setting works well with `failOnWarning`.

This can be configured on source set level.

Default: `false`

The set of visibility modifiers that should be documented.

This can be used if you want to document `protected` / `internal` / `private` declarations within this package, as well as if you want to exclude `public` declarations and only document internal API.

Can be configured on source set level.

Default: `PUBLIC`

External documentation links configuration

The `externalDocumentationLinks` block allows the creation of links that lead to the externally hosted documentation of your dependencies.

For example, if you are using types from `kotlinx.serialization`, by default they are unclickable in your documentation, as if they are unresolved. However, since the API reference documentation for `kotlinx.serialization` is built by Dokka and is published on kotlinlang.org, you can configure external documentation links for it. Thus allowing Dokka to generate links for types from the library, making them resolve successfully and clickable.

You can configure external documentation links for all source sets together at the same time, or individually:

```
{
  "externalDocumentationLinks": [
    {
      "url": "https://kotlinlang.org/api/kotlinx.serialization/",
      "packageListUrl": "https://kotlinlang.org/api/kotlinx.serialization/package-list"

    }
  ]
}
```

The root URL of documentation to link to. It **must** contain a trailing slash.

Dokka does its best to automatically find `package-list` for the given URL, and link declarations together.

If automatic resolution fails or if you want to use locally cached files instead, consider setting the `packageListUrl` option.

The exact location of a `package-list`. This is an alternative to relying on Dokka automatically resolving it.

Package lists contain information about the documentation and the project itself, such as module and package names.

This can also be a locally cached file to avoid network calls.

Complete configuration

Below you can see all possible configuration options applied at the same time.

```
{
  "moduleName": "Dokka Example",
  "moduleVersion": null,
  "outputDir": "./build/dokka/html",
  "failOnWarning": false,
  "suppressObviousFunctions": true,
  "suppressInheritedMembers": false,
```

```
  "offlineMode": false,
  "sourceLinks": [
    {
      "localDirectory": "src/main/kotlin",
      "remoteUrl": "https://github.com/Kotlin/dokka/tree/master/src/main/kotlin",
      "remoteLineSuffix": "#L"
    }
  ],
  "externalDocumentationLinks": [
    {
      "url": "https://docs.oracle.com/javase/8/docs/api/",
      "packageListUrl": "https://docs.oracle.com/javase/8/docs/api/package-list"
    },
    {
      "url": "https://kotlinlang.org/api/latest/jvm/stdlib/",
      "packageListUrl": "https://kotlinlang.org/api/latest/jvm/stdlib/package-list"
    }
  ],
  "perPackageOptions": [
    {
      "matchingRegex": ".*internal.*",
      "suppress": false,
      "reportUndocumented": false,
      "skipDeprecated": false,
      "documentedVisibilities": ["PUBLIC", "PRIVATE", "PROTECTED", "INTERNAL", "PACKAG
E"]
    }
  ],
  "sourceSets": [
    {
      "displayName": "jvm",
      "sourceSetID": {
        "scopeId": "moduleName",
        "sourceSetName": "main"
      },
      "dependentSourceSets": [
        {
          "scopeId": "dependentSourceSetScopeId",
          "sourceSetName": "dependentSourceSetName"
        }
      ],
      "documentedVisibilities": ["PUBLIC", "PRIVATE", "PROTECTED", "INTERNAL", "PACKAG
E"],
      "reportUndocumented": false,
      "skipEmptyPackages": true,
      "skipDeprecated": false,
      "jdkVersion": 8,
      "languageVersion": "1.7",
      "apiVersion": "1.7",
      "noStdlibLink": false,
      "noJdkLink": false,
      "includes": [
        "module.md"
```

```
      ],
      "analysisPlatform": "jvm",
      "sourceRoots": [
        "/home/ignat/IdeaProjects/dokka-debug-mvn/src/main/kotlin"
      ],
      "classpath": [
        "libs/kotlin-stdlib-2.1.21.jar",
        "libs/kotlin-stdlib-common-2.1.21.jar"
      ],
      "samples": [
        "samples/basic.kt"
      ],
      "suppressedFiles": [
        "src/main/kotlin/org/jetbrains/dokka/Suppressed.kt"
      ],
      "sourceLinks": [
        {
          "localDirectory": "src/main/kotlin",
          "remoteUrl": "https://github.com/Kotlin/dokka/tree/master/src/main/kotlin",
          "remoteLineSuffix": "#L"
        }
      ],
      "externalDocumentationLinks": [
        {
          "url": "https://docs.oracle.com/javase/8/docs/api/",
          "packageListUrl": "https://docs.oracle.com/javase/8/docs/api/package-list"
        },
        {
          "url": "https://kotlinlang.org/api/latest/jvm/stdlib/",
          "packageListUrl": "https://kotlinlang.org/api/latest/jvm/stdlib/package-list"

        }
      ],
      "perPackageOptions": [
        {
          "matchingRegex": ".*internal.*",
          "suppress": false,
          "reportUndocumented": false,
          "skipDeprecated": false,
          "documentedVisibilities": ["PUBLIC", "PRIVATE", "PROTECTED", "INTERNAL", "PA
CKAGE"]
        }
      ]
    }
  ],
  "pluginsClasspath": [
    "./dokka-base-2.0.0.jar",
    "./kotlinx-html-jvm-0.8.0.jar",
    "./analysis-kotlin-descriptors-2.0.0.jar",
    "./freemarker-2.3.31.jar"
  ],
  "pluginsConfiguration": [
    {
```

```
      "fqPluginName": "org.jetbrains.dokka.base.DokkaBase",
      "serializationFormat": "JSON",
      "values": "{\"separateInheritedMembers\":false,\"footerMessage\":\"© 2021 pretty
 good Copyright\"}"
    }
  ],
  "includes": [
    "module.md"
  ]
}
```

输出格式

- HTML
- Markdown
- Javadoc

HTML

HTML is Dokka's default and recommended output format. It is currently in Beta and approaching the Stable release.

You can see an example of the output by browsing documentation for kotlinx.coroutines.

Generate HTML documentation

HTML as an output format is supported by all runners. To generate HTML documentation, follow these steps depending on your build tool or runner:

- For Gradle, run `dokkaHtml` or `dokkaHtmlMultiModule` tasks.
- For Maven, run the `dokka:dokka` goal.
- For CLI runner, run with HTML dependencies set.

> HTML pages generated by this format need to be hosted on a web server in order to render everything correctly.
>
> You can use any free static site hosting service, such as GitHub Pages.
>
> Locally, you can use the built-in IntelliJ web server.
>
> 

Configuration

HTML format is Dokka's base format, so it is configurable through `DokkaBase` and `DokkaBaseConfiguration` classes:

> **【Kotlin】**

Via type-safe Kotlin DSL:

```
import org.jetbrains.dokka.base.DokkaBase
import org.jetbrains.dokka.gradle.DokkaTask
import org.jetbrains.dokka.base.DokkaBaseConfiguration

buildscript {
    dependencies {
```

```
        classpath("org.jetbrains.dokka:dokka-base:2.0.0")
    }
}

tasks.withType<DokkaTask>().configureEach {
    pluginConfiguration<DokkaBase, DokkaBaseConfiguration> {
        customAssets = listOf(file("my-image.png"))
        customStyleSheets = listOf(file("my-styles.css"))
        footerMessage = "(c) 2022 MyOrg"
        separateInheritedMembers = false
        templatesDir = file("dokka/templates")
        mergeImplicitExpectActualDeclarations = false
    }
}
```

Via JSON:

```
import org.jetbrains.dokka.gradle.DokkaTask

tasks.withType<DokkaTask>().configureEach {
    val dokkaBaseConfiguration = """
    {
      "customAssets": ["${file("assets/my-image.png")}"],
      "customStyleSheets": ["${file("assets/my-styles.css")}"],
      "footerMessage": "(c) 2022 MyOrg",
      "separateInheritedMembers": false,
      "templatesDir": "${file("dokka/templates")}",
      "mergeImplicitExpectActualDeclarations": false
    }
    """
    pluginsMapConfiguration.set(
        mapOf(
            // fully qualified plugin name to json configuration
            "org.jetbrains.dokka.base.DokkaBase" to dokkaBaseConfiguration
        )
    )
}
```

> 【Groovy】

```
import org.jetbrains.dokka.gradle.DokkaTask

tasks.withType(DokkaTask.class) {
    String dokkaBaseConfiguration = """
    {
      "customAssets": ["${file("assets/my-image.png")}"],
      "customStyleSheets": ["${file("assets/my-styles.css")}"],
      "footerMessage": "(c) 2022 MyOrg"
```

```
      "separateInheritedMembers": false,
      "templatesDir": "${file("dokka/templates")}",
      "mergeImplicitExpectActualDeclarations": false
    }
    """
    pluginsMapConfiguration.set(
            // fully qualified plugin name to json configuration
            ["org.jetbrains.dokka.base.DokkaBase": dokkaBaseConfiguration]
    )
}
```

> 【Maven】

```
<plugin>
    <groupId>org.jetbrains.dokka</groupId>
    <artifactId>dokka-maven-plugin</artifactId>
    ...
    <configuration>
        <pluginsConfiguration>
            <!-- Fully qualified plugin name -->
            <org.jetbrains.dokka.base.DokkaBase>
                <!-- Options by name -->
                <customAssets>
                    <asset>${project.basedir}/my-image.png</asset>
                </customAssets>
                <customStyleSheets>
                    <stylesheet>${project.basedir}/my-styles.css</stylesheet>
                </customStyleSheets>
                <footerMessage>(c) MyOrg 2022 Maven</footerMessage>
                <separateInheritedMembers>false</separateInheritedMembers>
                <templatesDir>${project.basedir}/dokka/templates</templatesDir>
                <mergeImplicitExpectActualDeclarations>false</mergeImplicitExpectActua
lDeclarations>
            </org.jetbrains.dokka.base.DokkaBase>
        </pluginsConfiguration>
    </configuration>
</plugin>
```

> 【CLI】

Via command line options:

```
java -jar dokka-cli-2.0.0.jar \
     ...
     -pluginsConfiguration "org.jetbrains.dokka.base.DokkaBase={\"customAssets\": [\"m
y-image.png\"], \"customStyleSheets\": [\"my-styles.css\"], \"footerMessage\": \"(c) 2
```

```
022 MyOrg\", \"separateInheritedMembers\": false, \"templatesDir\": \"dokka/templates\
", \"mergeImplicitExpectActualDeclarations\": false}
"
```

Via JSON configuration:

```
{
  "moduleName": "Dokka Example",
  "pluginsConfiguration": [
    {
      "fqPluginName": "org.jetbrains.dokka.base.DokkaBase",
      "serializationFormat": "JSON",
      "values": "{\"customAssets\": [\"my-image.png\"], \"customStyleSheets\": [\"my-s
tyles.css\"], \"footerMessage\": \"(c) 2022 MyOrg\", \"separateInheritedMembers\": fal
se, \"templatesDir\": \"dokka/templates\", \"mergeImplicitExpectActualDeclarations\":
false}"
    }
  ]
}
```

Configuration options

The table below contains all of the possible configuration options and their purpose.

Option	Description
`customAssets`	List of paths for image assets to be bundled with documentation. The image assets can have any file extension. For more information, see Customizing assets.
`customStyleSheets`	List of paths for `.css` stylesheets to be bundled with documentation and used for rendering. For more information, see Customizing styles.
`templatesDir`	Path to the directory containing custom HTML templates. For more information, see Templates.
`footerMessage`	The text displayed in the footer.
`separateInheritedMembers`	This is a boolean option. If set to `true`, Dokka renders properties/functions and inherited properties/inherited functions separately. This is disabled by default.
`mergeImplicitExpectActualDeclarations`	This is a boolean option. If set to `true`, Dokka merges declarations that are not declared as expect/actual, but have the same fully qualified name. This can be useful for legacy codebases. This is disabled by default.

For more information about configuring Dokka plugins, see Configuring Dokka plugins.

Customization

To help you add your own look and feel to your documentation, the HTML format supports a number of customization options.

Customize styles

You can use your own stylesheets by using the `customStyleSheets` configuration option. These are applied to every page.

It's also possible to override Dokka's default stylesheets by providing files with the same name:

Stylesheet name	Description
`style.css`	Main stylesheet, contains most of the styles used across all pages
`logo-styles.css`	Header logo styling
`prism.css`	Styles for PrismJS syntax highlighter

The source code for all of Dokka's stylesheets is available on GitHub.

Customize assets

You can provide your own images to be bundled with documentation by using the `customAssets` configuration option.

These files are copied to the `<output>/images` directory.

It's possible to override Dokka's images and icons by providing files with the same name. The most useful and relevant one being `logo-icon.svg`, which is the image that's used in the header. The rest is mostly icons.

You can find all images used by Dokka on GitHub.

Change the logo

To customize the logo, you can begin by providing your own asset for `logo-icon.svg`.

If you don't like how it looks, or you want to use a `.png` file instead of the default `.svg` file, you can override the `logo-styles.css` stylesheet to customize it.

For an example of how to do this, see our custom format example project.

Modify the footer

You can modify text in the footer by using the `footerMessage` configuration option.

Templates

Dokka provides the ability to modify FreeMarker templates used for generating documentation pages.

You can change the header completely, add your own banners/menus/search, load analytics, change body styling and so on.

Dokka uses the following templates:

Template	Description
`base.ftl`	Defines the general design of all pages to be rendered.
`includes/header.ftl`	The page header that by default contains the logo, version, source set selector, light/dark theme switch, and search.
`includes/footer.ftl`	The page footer that contains the `footerMessage` configuration option and copyright.
`includes/page_metadata.ftl`	Metadata used within `<head>` container.
`includes/source_set_selector.ftl`	The source set selector in the header.

The base template is `base.ftl` and it includes all of the remaining listed templates. You can find the source code for all of Dokka's templates on GitHub.

You can override any template by using the `templatesDir` configuration option. Dokka searches for the exact template names within the given directory. If it fails to find user-defined templates, it uses the default templates.

Variables

The following variables are available inside all templates:

Variable	Description
`${pageName}`	The page name
`${footerMessage}`	The text which is set by the `footerMessage` configuration option
`${sourceSets}`	A nullable list of source sets for multi-platform pages. Each item has `name`, `platform`, and `filter` properties.
`${projectName}`	The project name. It's available only within the `template_cmd` directive.
`${pathToRoot}`	The path to root from the current page. It's useful for locating assets and

	is available only within the `template_cmd` directive.

Variables `projectName` and `pathToRoot` are available only within the `template_cmd` directive as they require more context and thus they need to be resolved at later stages by the MultiModule task:

```
<@template_cmd name="projectName">
   <span>${projectName}</span>
</@template_cmd>
```

Directives

You can also use the following Dokka-defined directives:

Variable	Description
`<@content/>`	The main page content.
`<@resources/>`	Resources such as scripts and stylesheets.
`<@version/>`	The module version taken from configuration. If the versioning plugin is applied, it is replaced with a version navigator.

Markdown

> The Markdown output formats are still in Alpha, so you may find bugs and experience migration issues when using them. **You use them at your own risk.**
>
> 

Dokka is able to generate documentation in GitHub Flavored and Jekyll compatible Markdown.

These formats give you more freedom in terms of hosting documentation as the output can be embedded right into your documentation website. For example, see OkHttp's API reference pages.

Markdown output formats are implemented as Dokka plugins, maintained by the Dokka team, and they are open source.

GFM

The GFM output format generates documentation in GitHub Flavored Markdown.

> 【Gradle】

The Gradle plugin for Dokka comes with the GFM output format included. You can use the following tasks with it:

Task	Description
`dokkaGfm`	Generates GFM documentation for a single project.
`dokkaGfmMultiModule`	A `MultiModule` task created only for parent projects in multi-project builds. It generates documentation for subprojects and collects all outputs in a single place with a common table of contents.
`dokkaGfmCollector`	A `Collector` task created only for parent projects in multi-project builds. It calls `dokkaGfm` for every subproject and merges all outputs into a single virtual project.

> 【Maven】

Since GFM format is implemented as a Dokka plugin, you need to apply it as a plugin dependency:

```
<plugin>
    <groupId>org.jetbrains.dokka</groupId>
```

```
    <artifactId>dokka-maven-plugin</artifactId>
    ...
    <configuration>
        <dokkaPlugins>
            <plugin>
                <groupId>org.jetbrains.dokka</groupId>
                <artifactId>gfm-plugin</artifactId>
                <version>2.0.0</version>
            </plugin>
        </dokkaPlugins>
    </configuration>
</plugin>
```

After configuring this, running the `dokka:dokka` goal produces documentation in GFM format.

For more information, see the Maven plugin documentation for Other output formats.

> 【CLI】

Since GFM format is implemented as a Dokka plugin, you need to download the JAR file and pass it to `pluginsClasspath`.

Via command line options:

```
java -jar dokka-cli-2.0.0.jar \
     -pluginsClasspath "./dokka-base-2.0.0.jar;...;./gfm-plugin-2.0.0.jar" \
     ...
```

Via JSON configuration:

```
{
  ...
  "pluginsClasspath": [
    "./dokka-base-2.0.0.jar",
    "...",
    "./gfm-plugin-2.0.0.jar"
  ],
  ...
}
```

For more information, see the CLI runner's documentation for Other output formats.

You can find the source code on GitHub.

Jekyll

The Jekyll output format generates documentation in Jekyll compatible Markdown.

> 【Gradle】

The Gradle plugin for Dokka comes with the Jekyll output format included. You can use the following tasks with it:

Task	Description
`dokkaJekyll`	Generates Jekyll documentation for a single project.
`dokkaJekyllMultiModule`	A `MultiModule` task created only for parent projects in multi-project builds. It generates documentation for subprojects and collects all outputs in a single place with a common table of contents.
`dokkaJekyllCollector`	A `Collector` task created only for parent projects in multi-project builds. It calls `dokkaJekyll` for every subproject and merges all outputs into a single virtual project.

> 【Maven】

Since Jekyll format is implemented as a Dokka plugin, you need to apply it as a plugin dependency:

```
<plugin>
    <groupId>org.jetbrains.dokka</groupId>
    <artifactId>dokka-maven-plugin</artifactId>
    ...
    <configuration>
        <dokkaPlugins>
            <plugin>
                <groupId>org.jetbrains.dokka</groupId>
                <artifactId>jekyll-plugin</artifactId>
                <version>2.0.0</version>
            </plugin>
        </dokkaPlugins>
    </configuration>
</plugin>
```

After configuring this, running the `dokka:dokka` goal produces documentation in GFM format.

For more information, see the Maven plugin's documentation for Other output formats.

> 【CLI】

Since Jekyll format is implemented as a Dokka plugin, you need to download the JAR file. This format is also based on GFM format, so you need to provide it as a dependency as well. Both JARs need to be passed to `pluginsClasspath`:

Via command line options:

```
java -jar dokka-cli-2.0.0.jar \
     -pluginsClasspath "./dokka-base-2.0.0.jar;...;./gfm-plugin-2.0.0.jar;./jekyll-plu
gin-2.0.0.jar" \
     ...
```

Via JSON configuration:

```
{
  ...
  "pluginsClasspath": [
    "./dokka-base-2.0.0.jar",
    "...",
    "./gfm-plugin-2.0.0.jar",
    "./jekyll-plugin-2.0.0.jar"
  ],
  ...
}
```

For more information, see the CLI runner's documentation for Other output formats.

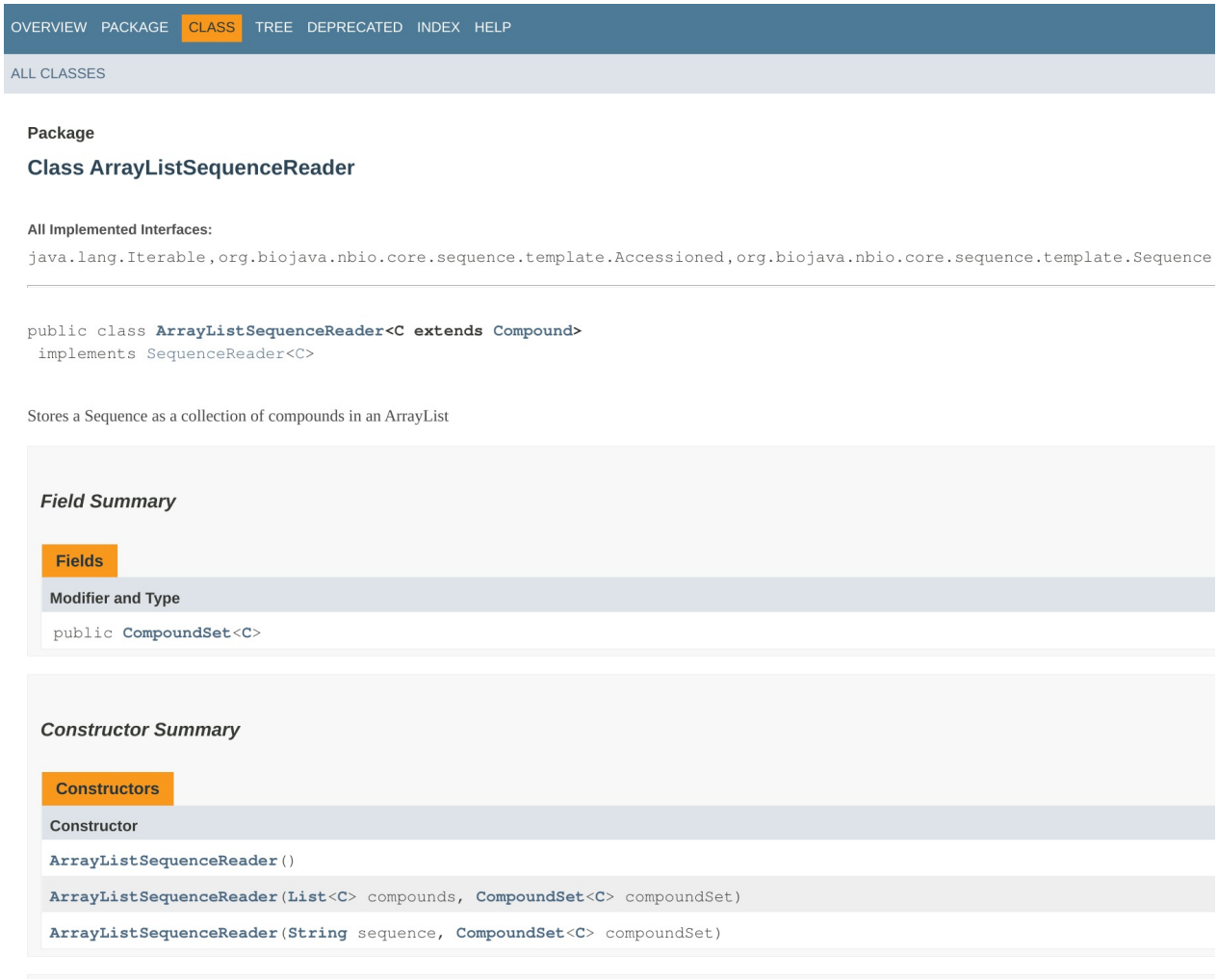
You can find the source code on GitHub.

Javadoc

> The Javadoc output format is still in Alpha, so you may find bugs and experience migration issues when using it. Successful integration with tools that accept Java's Javadoc HTML as input is not guaranteed. **You use it at your own risk.**
>
> 

Dokka's Javadoc output format is a lookalike of Java's Javadoc HTML format.

It tries to visually mimic HTML pages generated by the Javadoc tool, but it's not a direct implementation or an exact copy.



All Kotlin code and signatures are rendered as seen from Java's perspective. This is achieved with our Kotlin as Java Dokka plugin, which comes bundled and applied by default for this format.

The Javadoc output format is implemented as a Dokka plugin, and it is maintained by the Dokka team. It is open source and you can find the source code on GitHub.

Generate Javadoc documentation

> The Javadoc format does not support multiplatform projects.
>
> 

> 【Gradle】

The Gradle plugin for Dokka comes with the Javadoc output format included. You can use the following tasks:

Task	Description
`dokkaJavadoc`	Generates Javadoc documentation for a single project.
`dokkaJavadocCollector`	A `Collector` task created only for parent projects in multi-project builds. It calls `dokkaJavadoc` for every subproject and merges all outputs into a single virtual project.

The `javadoc.jar` file can be generated separately. For more information, see Building `javadoc.jar`.

> 【Maven】

The Maven plugin for Dokka comes with the Javadoc output format built in. You can generate documentation by using the following goals:

Goal	Description
`dokka:javadoc`	Generates documentation in Javadoc format
`dokka:javadocJar`	Generates a `javadoc.jar` file that contains documentation in Javadoc format

---

> 【CLI】

Since the Javadoc output format is a Dokka plugin, you need to download the plugin's JAR file.

The Javadoc output format has two dependencies that you need to provide as additional JAR files:

- kotlin-as-java plugin
- korte-jvm

Via command line options:

```
java -jar dokka-cli-2.0.0.jar \
     -pluginsClasspath "./dokka-base-2.0.0.jar;...;./javadoc-plugin-2.0.0.jar" \
     ...
```

Via JSON configuration:

```
{
  ...
  "pluginsClasspath": [
    "./dokka-base-2.0.0.jar",
    "...",
    "./kotlin-as-java-plugin-2.0.0.jar",
    "./korte-jvm-3.3.0.jar",
    "./javadoc-plugin-2.0.0.jar"
  ],
  ...
}
```

For more information, see Other output formats in the CLI runner's documentation.

Dokka 插件

Dokka was built from the ground up to be easily extensible and highly customizable, which allows the community to implement plugins for missing or very specific features that are not provided out of the box.

Dokka plugins range anywhere from supporting other programming language sources to exotic output formats. You can add support for your own KDoc tags or annotations, teach Dokka how to render different DSLs that are found in KDoc descriptions, visually redesign Dokka's pages to be seamlessly integrated into your company's website, integrate it with other tools and so much more.

If you want to learn how to create Dokka plugins, see Developer guides.

Apply Dokka plugins

Dokka plugins are published as separate artifacts, so to apply a Dokka plugin you only need to add it as a dependency. From there, the plugin extends Dokka by itself - no further action is needed.

> Plugins that use the same extension points or work in a similar way can interfere with each other. This may lead to visual bugs, general undefined behaviour or even failed builds. However, it should not lead to concurrency issues since Dokka does not expose any mutable data structures or objects.
>
> If you notice problems like this, it's a good idea to check which plugins are applied and what they do.
>
> 

Let's have a look at how you can apply the mathjax plugin to your project:

> **【Kotlin】**

The Gradle plugin for Dokka creates convenient dependency configurations that allow you to apply plugins universally or for a specific output format only.

```
dependencies {
    // Is applied universally
    dokkaPlugin("org.jetbrains.dokka:mathjax-plugin:2.0.0")

    // Is applied for the single-module dokkaHtml task only
    dokkaHtmlPlugin("org.jetbrains.dokka:kotlin-as-java-plugin:2.0.0")
```

```
    // Is applied for HTML format in multi-project builds
    dokkaHtmlPartialPlugin("org.jetbrains.dokka:kotlin-as-java-plugin:2.0.0")
}
```

> When documenting multi-project builds, you need to apply Dokka plugins within subprojects as well as in their parent project.
>
> 

> 【Groovy】

The Gradle plugin for Dokka creates convenient dependency configurations that allow you to apply Dokka plugins universally or for a specific output format only.

```
dependencies {
    // Is applied universally
    dokkaPlugin 'org.jetbrains.dokka:mathjax-plugin:2.0.0'

    // Is applied for the single-module dokkaHtml task only
    dokkaHtmlPlugin 'org.jetbrains.dokka:kotlin-as-java-plugin:2.0.0'

    // Is applied for HTML format in multi-project builds
    dokkaHtmlPartialPlugin 'org.jetbrains.dokka:kotlin-as-java-plugin:2.0.0'
}
```

> When documenting multi-project builds, you need to apply Dokka plugins within subprojects as well as in their parent project.
>
> 

> 【Maven】

```
<plugin>
    <groupId>org.jetbrains.dokka</groupId>
    <artifactId>dokka-maven-plugin</artifactId>
    ...
    <configuration>
        <dokkaPlugins>
            <plugin>
                <groupId>org.jetbrains.dokka</groupId>
                <artifactId>mathjax-plugin</artifactId>
                <version>2.0.0</version>
            </plugin>
        </dokkaPlugins>
```

```
    </configuration>
</plugin>
```

> 【CLI】

If you are using the CLI runner with command line options, Dokka plugins should be passed as `.jar` files to `-pluginsClasspath`:

```
java -jar dokka-cli-2.0.0.jar \
     -pluginsClasspath "./dokka-base-2.0.0.jar;...;./mathjax-plugin-2.0.0.jar" \
     ...
```

If you are using JSON configuration, Dokka plugins should be specified under `pluginsClasspath`.

```
{
  ...
  "pluginsClasspath": [
    "./dokka-base-2.0.0.jar",
    "...",
    "./mathjax-plugin-2.0.0.jar"
  ],
  ...
}
```

Configure Dokka plugins

Dokka plugins can also have configuration options of their own. To see which options are available, consult the documentation of the plugins you are using.

Let's have a look at how you can configure the `DokkaBase` plugin, which is responsible for generating HTML documentation, by adding a custom image to the assets (`customAssets` option), by adding custom style sheets (`customStyleSheets` option), and by modifying the footer message (`footerMessage` option):

> 【Kotlin】

Gradle's Kotlin DSL allows for type-safe plugin configuration. This is achievable by adding the plugin's artifact to the classpath dependencies in the `buildscript` block, and then importing plugin and configuration classes:

```
import org.jetbrains.dokka.base.DokkaBase
import org.jetbrains.dokka.gradle.DokkaTask
import org.jetbrains.dokka.base.DokkaBaseConfiguration

buildscript {
    dependencies {
        classpath("org.jetbrains.dokka:dokka-base:2.0.0")
    }
}

tasks.withType<DokkaTask>().configureEach {
    pluginConfiguration<DokkaBase, DokkaBaseConfiguration> {
        customAssets = listOf(file("my-image.png"))
        customStyleSheets = listOf(file("my-styles.css"))
        footerMessage = "(c) 2022 MyOrg"
    }
}
```

Alternatively, plugins can be configured via JSON. With this method, no additional dependencies are needed.

```
import org.jetbrains.dokka.gradle.DokkaTask

tasks.withType<DokkaTask>().configureEach {
    val dokkaBaseConfiguration = """
    {
      "customAssets": ["${file("assets/my-image.png")}"],
      "customStyleSheets": ["${file("assets/my-styles.css")}"],
      "footerMessage": "(c) 2022 MyOrg"
    }
    """
    pluginsMapConfiguration.set(
        mapOf(
            // fully qualified plugin name to json configuration
            "org.jetbrains.dokka.base.DokkaBase" to dokkaBaseConfiguration
        )
    )
}
```

> 【Groovy】

```
import org.jetbrains.dokka.gradle.DokkaTask

tasks.withType(DokkaTask.class) {
    String dokkaBaseConfiguration = """
    {
      "customAssets": ["${file("assets/my-image.png")}"],
```

```
      "customStyleSheets": ["${file("assets/my-styles.css")}"],
      "footerMessage": "(c) 2022 MyOrg"
    }
    """
    pluginsMapConfiguration.set(
            // fully qualified plugin name to json configuration
            ["org.jetbrains.dokka.base.DokkaBase": dokkaBaseConfiguration]
    )
}
```

> 【Maven】

```
<plugin>
    <groupId>org.jetbrains.dokka</groupId>
    <artifactId>dokka-maven-plugin</artifactId>
    ...
    <configuration>
        <pluginsConfiguration>
            <!-- Fully qualified plugin name -->
            <org.jetbrains.dokka.base.DokkaBase>
                <!-- Options by name -->
                <customAssets>
                    <asset>${project.basedir}/my-image.png</asset>
                </customAssets>
                <customStyleSheets>
                    <stylesheet>${project.basedir}/my-styles.css</stylesheet>
                </customStyleSheets>
                <footerMessage>(c) MyOrg 2022 Maven</footerMessage>
            </org.jetbrains.dokka.base.DokkaBase>
        </pluginsConfiguration>
    </configuration>
</plugin>
```

> 【CLI】

If you are using the CLI runner with command line options, use the `-pluginsConfiguration` option that accepts JSON configuration in the form of `fullyQualifiedPluginName=json`.

If you need to configure multiple plugins, you can pass multiple values separated by `^^`.

```
java -jar dokka-cli-2.0.0.jar \
     ...
     -pluginsConfiguration "org.jetbrains.dokka.base.DokkaBase={\"customAssets\": [\"m
y-image.png\"], \"customStyleSheets\": [\"my-styles.css\"], \"footerMessage\": \"(c) 2
022 MyOrg CLI\"}"
```

If you are using JSON configuration, there exists a similar `pluginsConfiguration` array that accepts JSON configuration in `values`.

```
{
  "moduleName": "Dokka Example",
  "pluginsConfiguration": [
    {
      "fqPluginName": "org.jetbrains.dokka.base.DokkaBase",
      "serializationFormat": "JSON",
      "values": "{\"customAssets\": [\"my-image.png\"], \"customStyleSheets\": [\"my-s
tyles.css\"], \"footerMessage\": \"(c) 2022 MyOrg\"}"
    }
  ]
}
```

Notable plugins

Here are some notable Dokka plugins that you might find useful:

Name	Description
Android documentation plugin	Improves the documentation experience on Android
Versioning plugin	Adds version selector and helps to organize documentation for different versions of your application/library
MermaidJS HTML plugin	Renders MermaidJS diagrams and visualizations found in KDocs
Mathjax HTML plugin	Pretty prints mathematics found in KDocs
Kotlin as Java plugin	Renders Kotlin signatures as seen from Java's perspective

If you are a Dokka plugin author and would like to add your plugin to this list, get in touch with maintainers via Slack or GitHub.

模块文档

Documentation for a module as a whole, as well as packages in that module, can be provided as separate Markdown files.

File format

Inside the Markdown file, the documentation for the module as a whole and for individual packages is introduced by the corresponding first-level headings. The text of the heading **must** be **Module** `<module name>` for a module, and **Package** `<package qualified name>` for a package.

The file doesn't have to contain both module and package documentation. You can have files that contain only package or module documentation. You can even have a Markdown file per module or package.

Using Markdown syntax, you can add:

- Headings up to level 6
- Emphasis with bold or italic formatting
- Links
- Inline code
- Code blocks
- Blockquotes

Here's an example file containing both module and package documentation:

```
# Module kotlin-demo

This content appears under your module name.

# Package org.jetbrains.kotlin.demo

This content appears under your package name in the packages list.
It also appears under the first-level heading on your package's page.

## Level 2 heading for package org.jetbrains.kotlin.demo

Content after this heading is also part of documentation for `org.jetbrains.kotlin.dem
o`

# Package org.jetbrains.kotlin.demo2

This content appears under your package name in the packages list.
It also appears under the first-level heading on your package's page.
```

```
## Level 2 heading for package org.jetbrains.kotlin.demo2

Content after this heading is also part of documentation for `org.jetbrains.kotlin.dem
o2`
```

To explore an example project with Gradle, see Dokka gradle example.

Pass files to Dokka

To pass these files to Dokka, you need to use the relevant **includes** option for Gradle, Maven, or CLI:

> 【Gradle】

Use the includes option in Source set configuration.

> 【Maven】

Use the includes option in General configuration.

> 【CLI】

If you are using command line configuration, use the includes option in Source set options.

If you are using JSON configuration, use the includes option in General configuration.

用于 Kotlin 开发的 IDE

JetBrains provides the official Kotlin support for the following IDEs and code editors: IntelliJ IDEA and Android Studio.

Other IDEs and code editors only have Kotlin community-supported plugins.

IntelliJ IDEA

IntelliJ IDEA is an IDE designed for JVM languages, such as Kotlin and Java, to maximize developer productivity. It does the routine and repetitive tasks for you by providing clever code completion, static code analysis, and refactorings. It lets you focus on the bright side of software development, making it not only productive but also an enjoyable experience.

The Kotlin plugin is bundled with each IntelliJ IDEA release. Each IDEA release introduces new features and upgrades that improve the experience for Kotlin developers in the IDE. See What's new in IntelliJ IDEA for the latest updates and improvements for Kotlin.

Read more about IntelliJ IDEA in the official documentation.

Android Studio

Android Studio is the official IDE for Android app development, based on IntelliJ IDEA. On top of IntelliJ's powerful code editor and developer tools, Android Studio offers even more features that enhance your productivity when building Android apps.

Kotlin plugin is bundled with each Android Studio release.

Read more about Android Studio in the official documentation.

Eclipse

Eclipse allows developers to write their applications in different programming languages, including Kotlin. It also has the Kotlin plugin: originally developed by JetBrains, now the Kotlin plugin is supported by the Kotlin community contributors.

You can install the Kotlin plugin manually from the Marketplace.

The Kotlin team manages the development and contribution process to the Kotlin plugin for Eclipse. If you want to contribute to the plugin, send a pull request to its repository on GitHub.

与 Kotlin 语言版本的兼容性

For IntelliJ IDEA and Android Studio, the Kotlin plugin is bundled with each release. When the new Kotlin version is released, these tools will suggest updating Kotlin to the latest version automatically. See the latest supported language version in Kotlin releases.

其他 IDE 支持

JetBrains doesn't provide Kotlin plugins for other IDEs. However, some of the other IDEs and source editors, such as Eclipse, Visual Studio Code, and Atom, have their own Kotlin plugins supported by the Kotlin community.

You can use any text editor to write the Kotlin code, but without IDE-related features: code formatting, debugging tools, and so on. To use Kotlin in text editors, you can download the latest Kotlin command-line compiler (`kotlin-compiler-2.1.21.zip`) from Kotlin GitHub Releases and install it manually. Also, you could use package managers, such as Homebrew, SDKMAN!, and Snap package.

下一步做什么？

- Start your first project using IntelliJ IDEA IDE
- Create your first cross-platform mobile app using Android Studio

迁移到 Kotlin 代码风格

Kotlin 编码规范与 IntelliJ IDEA 格式化程序

Kotlin 编码规范影响了编写地道 Kotlin 代码的几个方面，其中包括一组旨在提高 Kotlin 代码可读性的格式建议。

遗憾的是，IntelliJ IDEA 中内置的代码格式化工具在这篇文档发布很久之前就已经在使用了，并且现在具有默认设置， 该默认设置产生的格式不同于现在建议格式。

接下来，通过改变 IntelliJ IDEA 中的默认设置并使格式与 Kotlin 编码规范一致来消除这种隔阂似乎是符合逻辑的。但这意味着所有现有 Kotlin 项目将在安装 Kotlin 插件后启用新的代码格式。这并不是插件更新的预期结果，对不对？

这就是为什么我们有以下迁移计划的原因：

- 从 Kotlin 1.3 开始，默认情况下启用官方代码格式，并且仅对新项目启用（旧风格可以手动启用）
- 现有项目的作者可以选择迁移到现有的 Kotlin 编码规范
- 现有项目的作者可以选择进行显式声明在项目中使用旧代码风格（这样， 将来不会因改变默认值而影响项目）
- 切换到默认格式，使其与 Kotlin 1.4 中的 Kotlin 编码规范一致

“Kotlin 编码规范” 与 “IntelliJ IDEA 默认代码风格” 之间的差异

最显着的变化是延续缩进策略。使用双缩进来显示多行表达式尚未在前一行结束是一个好主意。这是一个非常简单且通用的规则，但是以这种方式格式化时， 一些 Kotlin 构造看起来有些尴尬。在 Kotlin 编码规范中，建议在之前强制使用长延续缩进的场景中使用单个缩进。

```
class User(
        val name: String,
        val address: String) {
    val key: String = getUserId(name, address)
}

fun findUser(
        users: List<User>,
        key: String? = null,
        name: String? = null): User? {
    val filteredUsers = users
            .asSequence()
            .filter { user -> key == null || user.key == key }
            .filter { user -> name == null || user.name.contains(name) }

    return filteredUsers.single()
}
```

```
class User(
    val name: String,
    val address: String
) {
    val key: String = getUserId(name, address)
}

fun findUser(
    users: List<User>,
    key: String? = null,
    name: String? = null
): User? {
    val filteredUsers = users
        .asSequence()
        .filter { user -> key == null || user.key == key }
        .filter { user -> name == null || user.name.contains(name) }

    return filteredUsers.single()
}
```

实际上，很多代码都会受到影响，因此可以将其视为重大的代码风格更新。

迁移到新的代码风格讨论

如果没有使用旧风格的代码，那么从新项目开始就采用新的代码风格应该是很自然的过程。因此，从 1.3 版开始，Kotlin IntelliJ 插件使用默认情况下启用的编码规范文档中的格式创建新项目。

在现有项目中更改格式是一项更加艰巨的任务，应该与团队讨论所有注意事项然后一起开始。

更改现有项目中的代码风格的主要缺点是，blame/annotate 版本控制系统特性将更频繁地指向无关的提交。尽管每种版本控制系统都有某种方式可以解决此问题（IntelliJ IDEA 中可以使用“Annotate Previous Revision”），但重要的是确定新风格是否值得所有努力。将修改格式的提交与有意义的更改分开的做法可以为以后的调查提供很大帮助。

对于大型团队来说，迁移也可能会比较困难，因为在多个子系统中提交大量文件可能会在个人的分支中产生合并冲突。尽管每个冲突解决方案通常都很琐碎，但明智的做法是知道当前是否正在使用大型特性分支。

通常，对于小型项目，建议一次转换所有文件。

对于大中型项目，决定可能会很艰难。如果还没有准备好立即更新许多文件，那么可以决定逐模块迁移，或者继续只对已修改文件逐步迁移。

迁移到新的代码风格

可以在 **Settings/Preferences** | **Editor** | **Code Style** | **Kotlin** 对话框中切换 Kotlin 代码风格。将 Scheme 切换到 **Project** 并从下方选择 **Set from...** | **Kotlin style guide**。

为了向所有项目开发人员共享这些更改，必须将 `.idea/codeStyle` 文件夹提交给版本控制系统。

如果使用外部构建系统来配置项目，并且已决定不共享 `.idea/codeStyle` 文件夹，那么可以通过附加属性强制使用 Kotlin 编码规范：

在 Gradle 中

在项目根目录的 `gradle.properties` 文件中添加 `kotlin.code.style=official` 属性，并将其提交到版本控制系统。

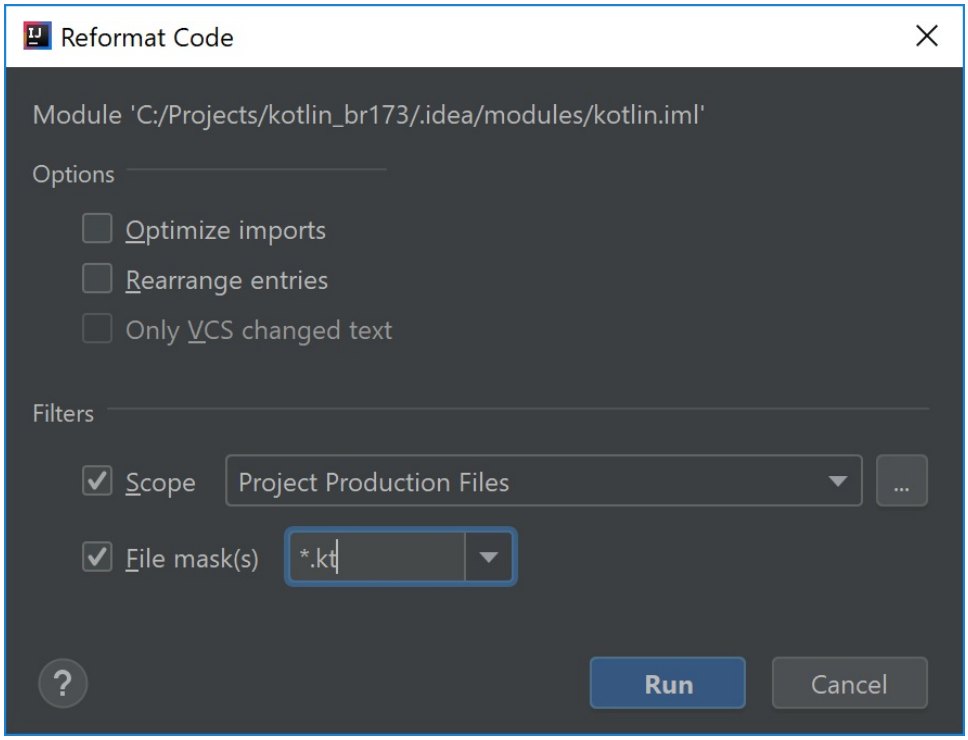
在 Maven 中

在项目根目录的 `pom.xml` 文件中添加 `kotlin.code.style official` 属性。

```
<properties>
  <kotlin.code.style>official</kotlin.code.style>
</properties>
```

> 设置 **kotlin.code.style** 选项可能会在项目导入期间修改代码风格方案，并且可能会更改代码风格设置。
>
> 

更新代码风格设置后，在所需范围的项目视图中选择 **Reformat Code**。



对于逐步迁移，可以启用 **File is not formatted according to project settings**（文件未根据项目设置格式化）探查项。 这会突出显示应修改格式的地方。启用 **Apply only to modified files”**（仅应用于修改后的文件）选项后， 检查会仅在修改后的文件中显示格式问题。无论如何，此类文件应该尽快修改并提交。

在项目中存储旧的代码风格

It's always possible to explicitly set the IntelliJ IDEA code style as the correct code style for the project:

1. In **Settings/Preferences** | **Editor** | **Code Style** | **Kotlin**, switch to the **Project** scheme.
2. Open the **Load/Save** tab and in the **Use defaults from** select **Kotlin obsolete IntelliJ IDEA codestyle**.

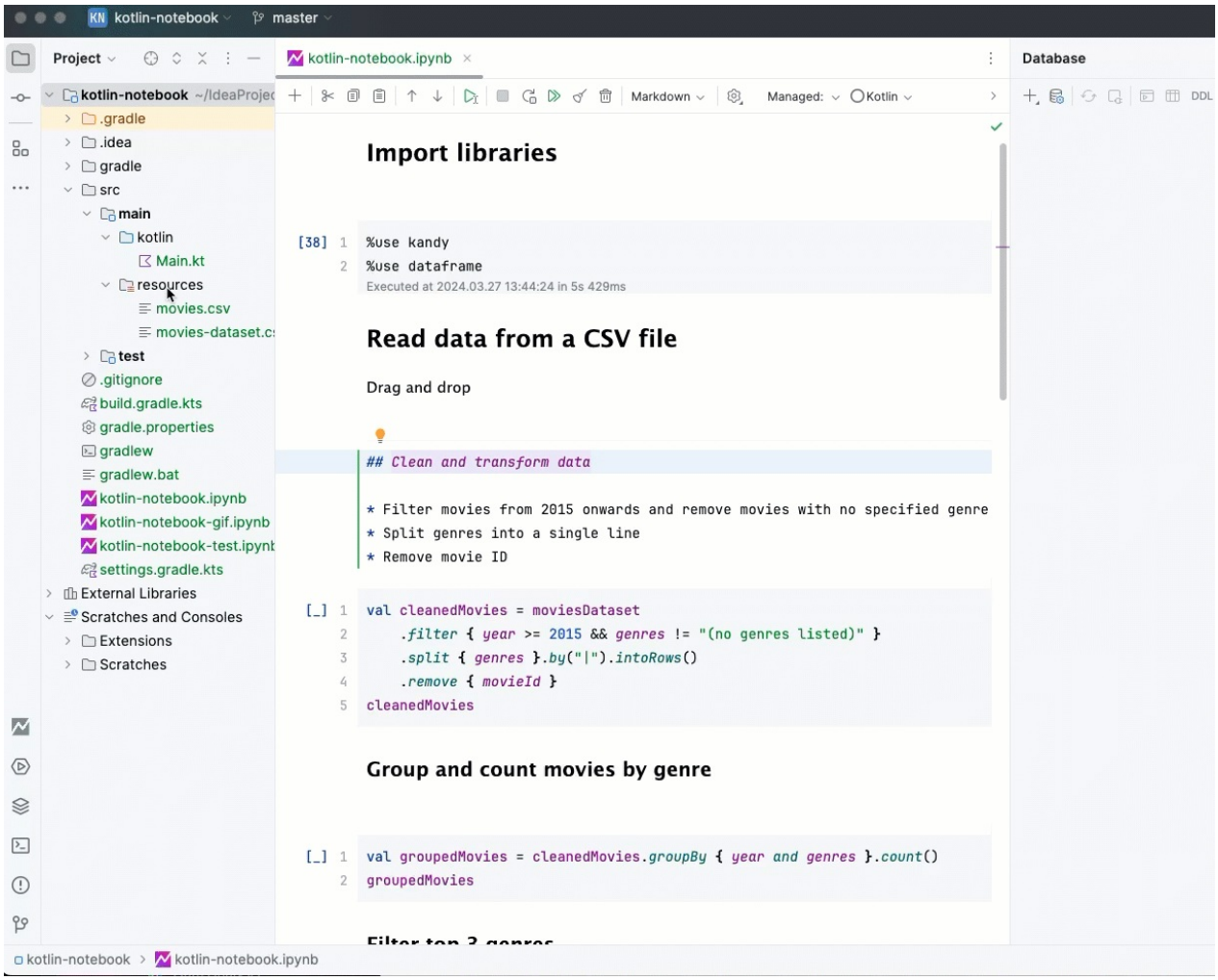
为了在项目开发人员的 `.idea/codeStyle` 文件夹中共享更改，必须将其提交给版本控制系统。 另外， **kotlin.code.style**=**obsolete** 可以用于配置了 Gradle 或 Maven 的项目。

Kotlin Notebook

Kotlin Notebook provides an interactive environment to create and edit notebooks, leveraging the full potential of Kotlin's capabilities.

Kotlin Notebook relies on the Kotlin Notebook plugin, which is bundled and enabled in IntelliJ IDEA by default.

Get ready for a seamless coding experience where you can develop and experiment with Kotlin code, receive immediate outputs, and integrate code, visuals, and text within the IntelliJ IDEA ecosystem.



The Kotlin Notebook plugin comes with various features to boost your development process, such as:

- Accessing APIs within cells
- Importing and exporting files with a few clicks
- Using REPL commands for a quick project exploration
- Getting a rich set of output formats
- Managing dependencies intuitively with annotations or Gradle-like syntax
- Importing various libraries with a single line of code or even adding new libraries to your project

- Getting insights for debugging with error messages and traceback

Kotlin Notebook is based on our Kotlin Kernel for Jupyter Notebooks, making it easy to integrate with other Kotlin notebook solutions. Without compatibility issues, you can effortlessly share your work among Kotlin Notebook, Datalore, and Kotlin-Jupyter Notebook.

With these capabilities, you can embark on a wide range of tasks, from simple code experiments to comprehensive data projects.

Dive deeper into the sections below to discover what you can achieve with Kotlin Notebook!

Data analytics and visualization

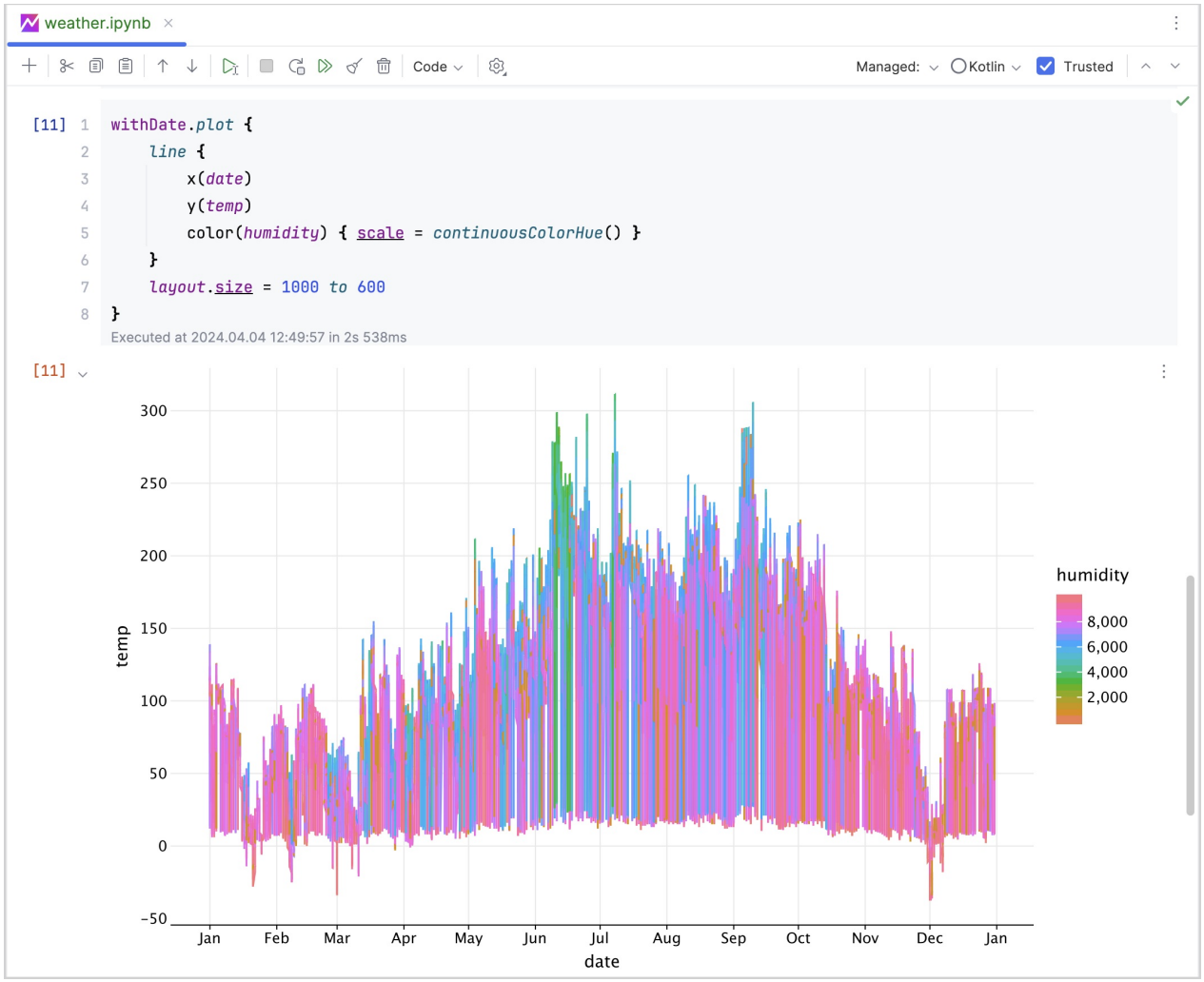
Whether you're conducting preliminary data exploration or completing an end-to-end data analysis project, Kotlin Notebook has the right tools for you.

Within Kotlin Notebook, you can intuitively integrate libraries that let you retrieve, transform, plot, and model your data while getting immediate outputs of your operations.

For analytics-related tasks, the Kotlin DataFrame library provides robust solutions. This library facilitates loading, creating, filtering, and cleaning structured data.

Kotlin DataFrame also supports seamless connection with SQL databases and reads data right in the IDE from different file formats, including CSV, JSON, and TXT.

Kandy, an open-source Kotlin library, allows you to create charts of various types. Kandy's idiomatic, readable, and type-safe features let you visualize data effectively and gain valuable insights.



Prototyping

Kotlin Notebook provides an interactive environment for running code in small chunks and seeing the results in real-time. This hands-on approach enables rapid experimentation and iteration during the prototyping phase.

With the help of Kotlin Notebook, you can test the concepts of solutions early in the ideation stage. Additionally, Kotlin Notebook supports both collaborative and reproducible work, enabling the generation and assessment of new ideas.

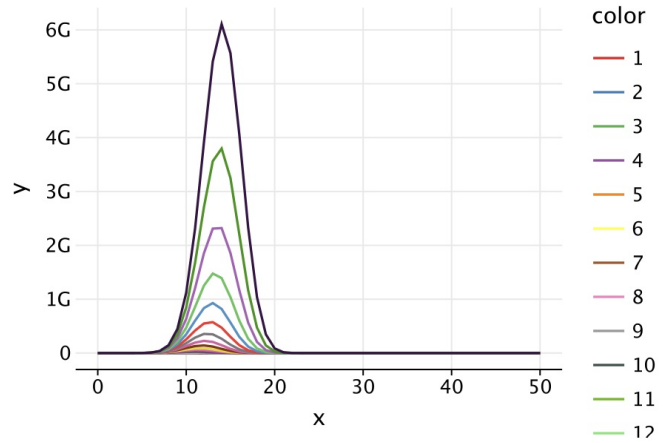
prototyping.ipynb

Code

```
[6] 1 val valsRange = 0 ≤ .. ≤ 50
    2 val funsRange = 0 ≤ .. ≤ 50
    3 plotFuns(valsRange.toList(), funsRange.map { i -> {j -> Ag(i, j)} })
```

Executed at 2024.05.03 15:11:36 in 650ms

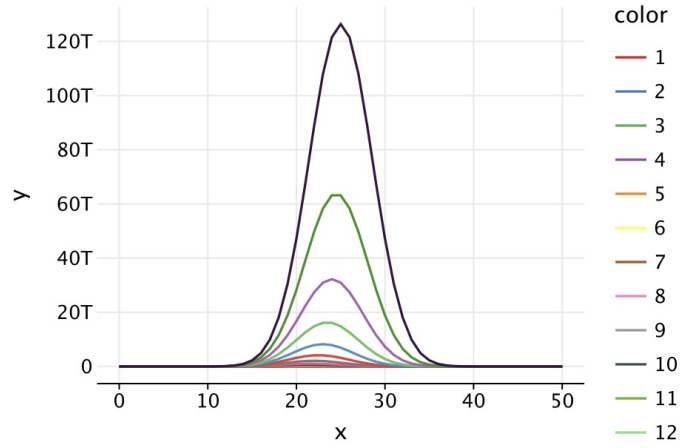
[6]




```
[7] 1 plotFuns(valsRange.toList(), funsRange.map { i -> {j -> if (i >= j) Cn(i, j) else 0} })
```

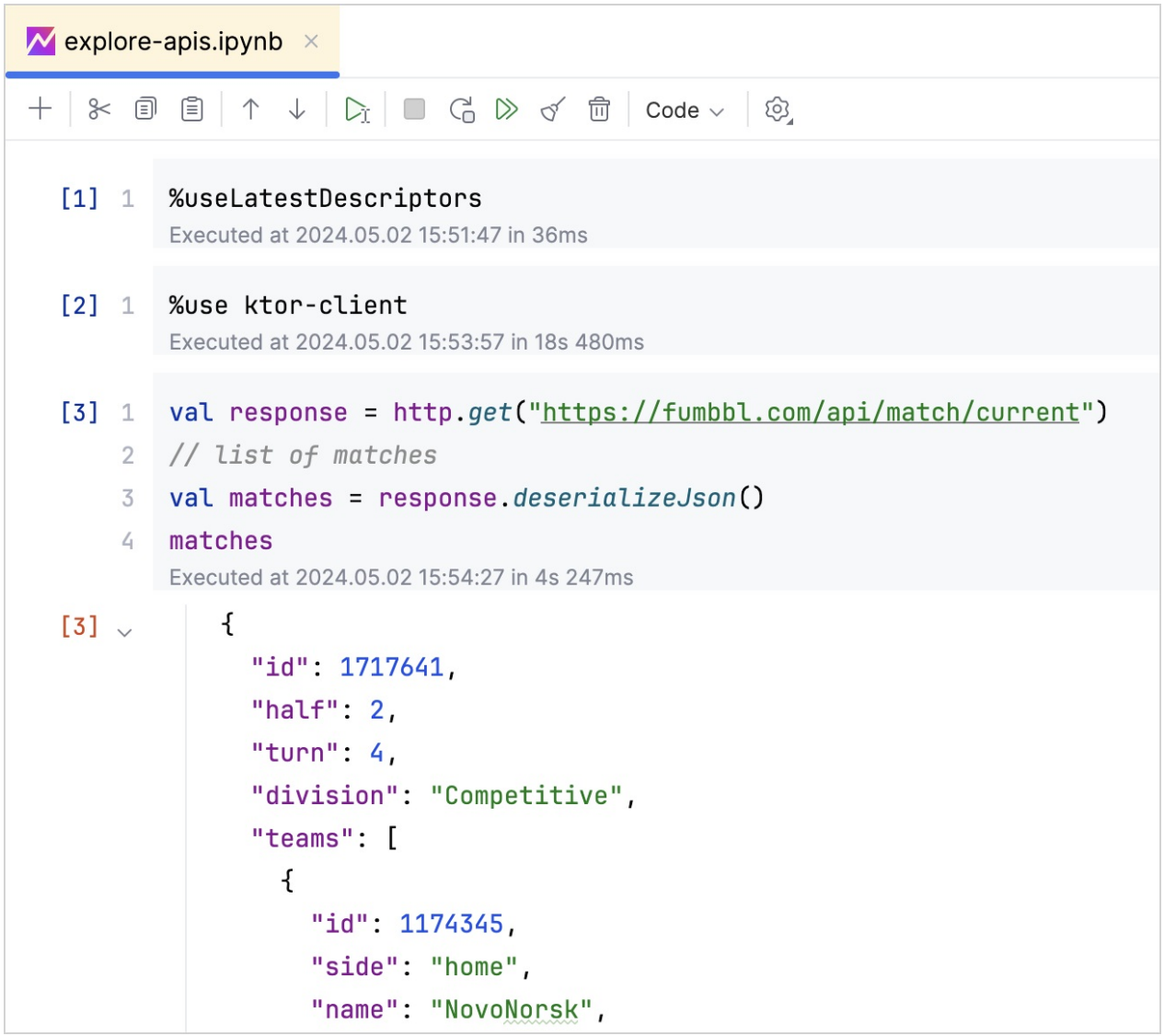
Executed at 2024.05.03 15:11:36 in 119ms

[7]



Backend development

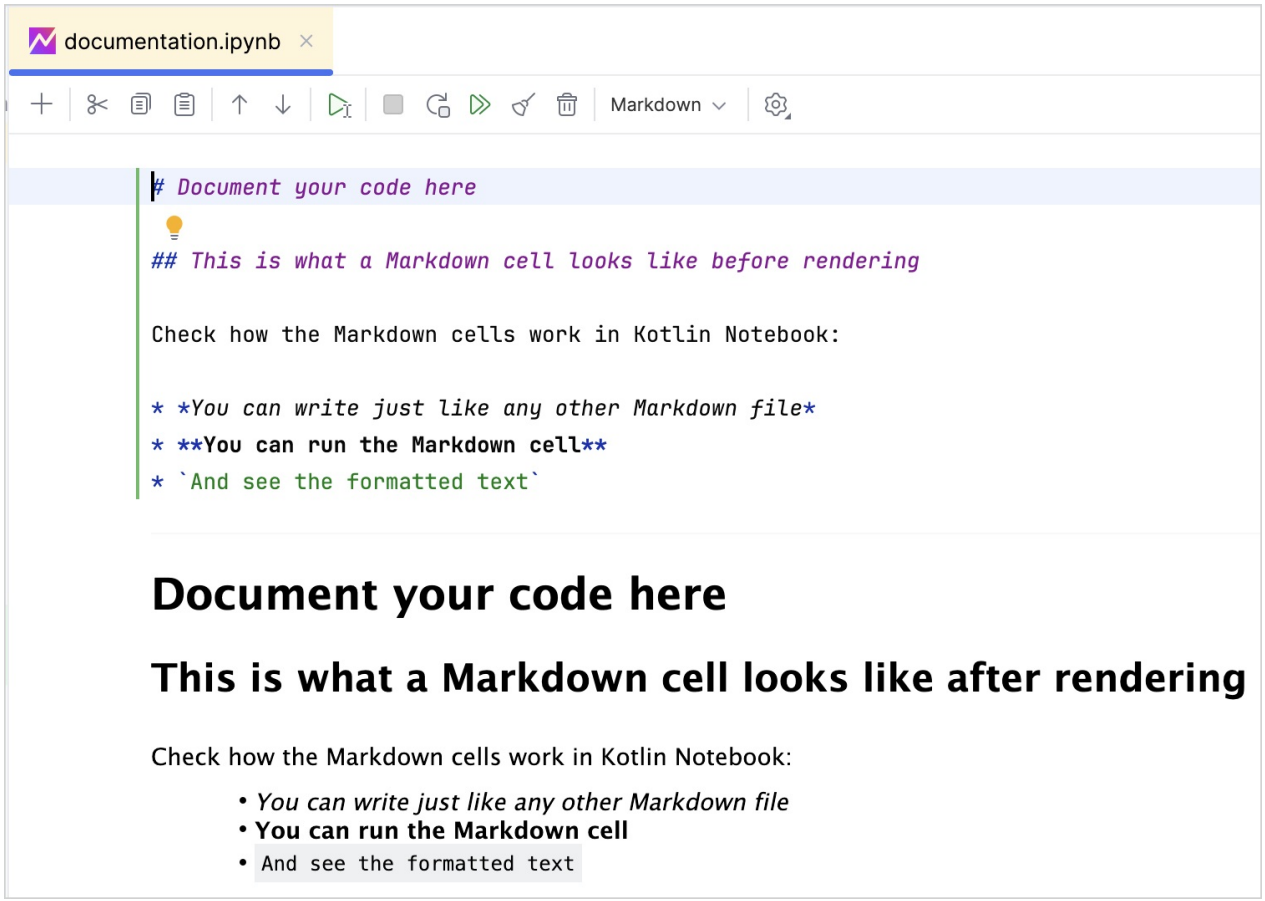
Kotlin Notebook offers the ability to call APIs within cells and work with protocols like OpenAPI. Its capability to interact with external services and APIs makes it useful for certain backend development scenarios, such as retrieving information and reading JSON files directly within your notebook environment.



Code documentation

In Kotlin Notebook, you can include inline comments and text annotations within code cells to provide additional context, explanations, and instructions relevant to the code snippets.

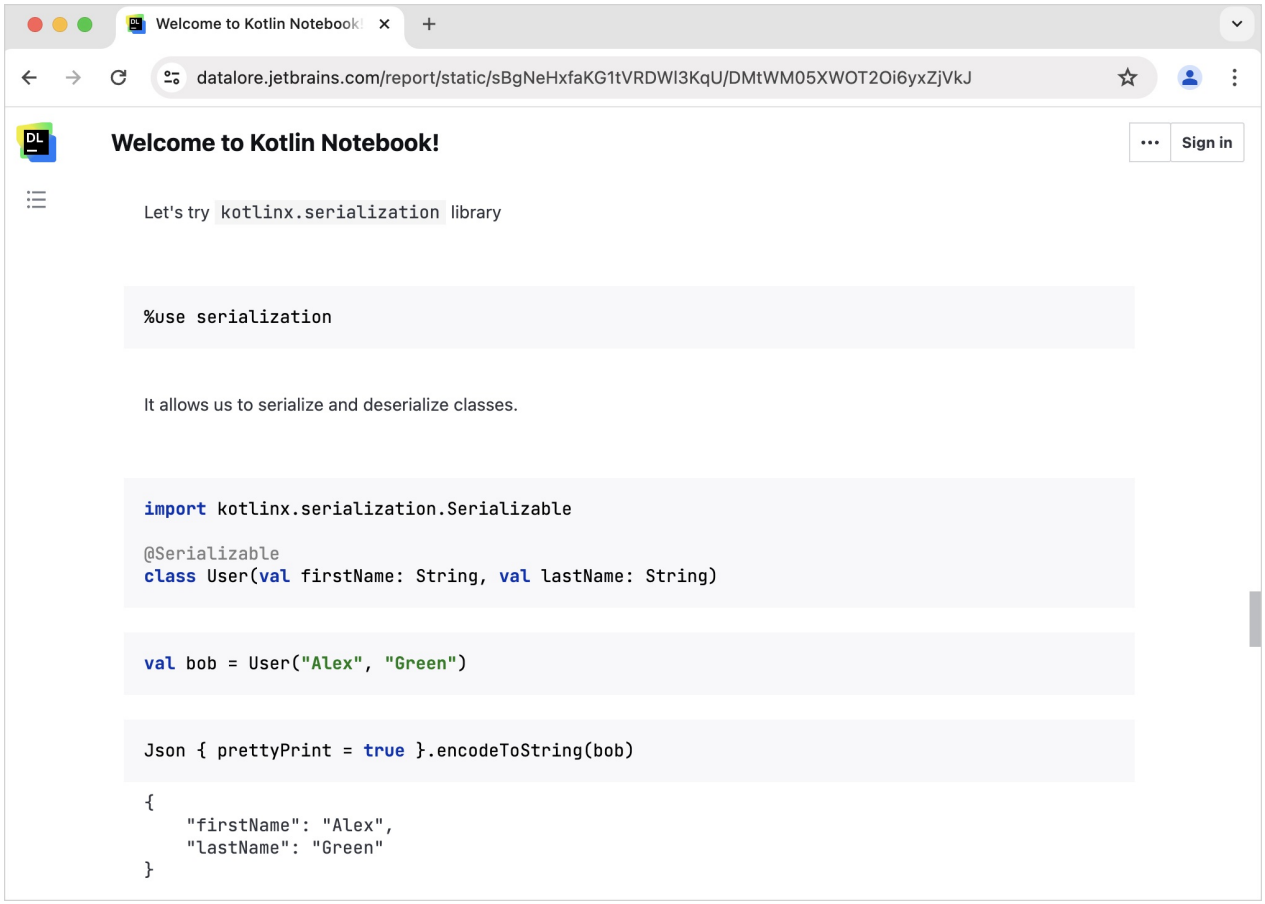
You can also write text in Markdown cells, which support rich formatting options such as headers, lists, links, images, and more. To render a Markdown cell and see the formatted text, simply run it as you would a code cell.



Sharing code and outputs

Given Kotlin Notebook's adherence to the universal Jupyter format, it's possible to share your code and outputs across different notebooks. You can open, edit, and run your Kotlin Notebook with any Jupyter client, such as Jupyter Notebook or Jupyter Lab.

You can also distribute your work by sharing the `.ipynb` notebook file with any notebook web viewer. One option is GitHub, which natively renders this format. Another option is JetBrain's Datalore platform, which facilitates sharing, running, and editing notebooks with advanced features like scheduled notebook runs.



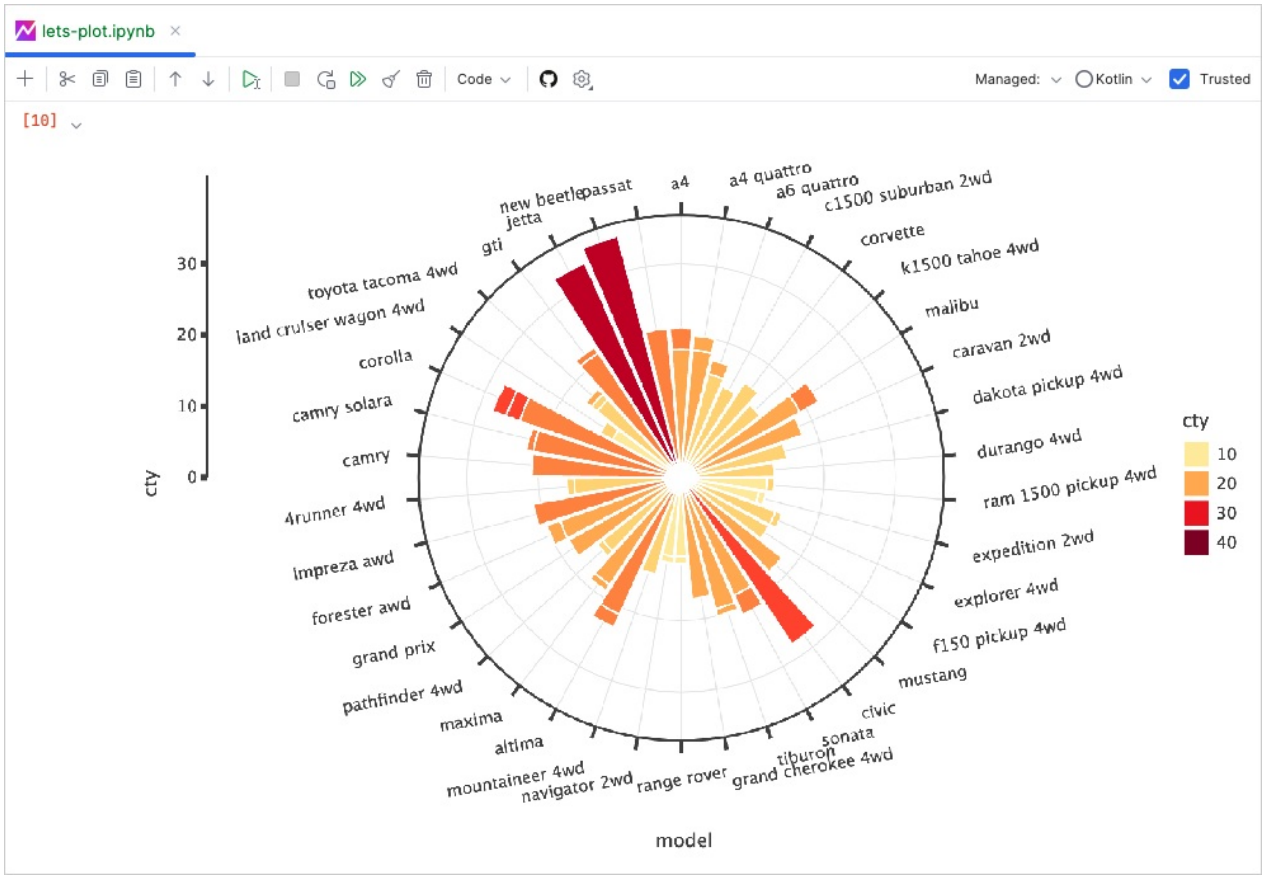
下一步做什么

- Learn about the Kotlin Notebook's usage and key features.
- Try out Kotlin Notebook.
- Deep dive into Kotlin for data analysis.

Data visualization with Lets-Plot for Kotlin

Lets-Plot for Kotlin (LPK) is a multiplatform plotting library that ports the R's ggplot2 library to Kotlin. LPK brings the feature-rich ggplot2 API to the Kotlin ecosystem, making it suitable for scientists and statisticians who require sophisticated data visualization capabilities.

LPK targets various platforms, including Kotlin notebooks, Kotlin/JS, JVM's Swing, JavaFX, and Compose Multiplatform. Additionally, LPK has seamless integration with IntelliJ, DataGrip, DataSpell, and PyCharm.



This tutorial demonstrates how to create different plot types with the LPK and Kotlin DataFrame libraries using Kotlin Notebook in IntelliJ IDEA.

Before you start

Kotlin Notebook relies on the Kotlin Notebook plugin, which is bundled and enabled in IntelliJ IDEA by default.

If the Kotlin Notebook features are not available, ensure the plugin is enabled. For more information, see Set up an environment.

Create a new Kotlin Notebook to work with Lets-Plot:

1. Select **File** | **New** | **Kotlin Notebook**.
2. In your notebook, import the LPK and Kotlin DataFrame libraries by running the following command:

```
%use lets-plot
%use dataframe
```

Prepare the data

Let's create a DataFrame that stores simulated numbers of the monthly average temperature in three cities: Berlin, Madrid, and Caracas.

Use the `dataFrameOf()` function from the Kotlin DataFrame library to generate the DataFrame. Paste and run the following code snippet in your Kotlin Notebook:

```
// The months variable stores a list with 12 months of the year
val months = listOf(
    "January", "February",
    "March", "April", "May",
    "June", "July", "August",
    "September", "October", "November",
    "December"
)
// The tempBerlin, tempMadrid, and tempCaracas variables store a list with temperature
 values for each month
val tempBerlin =
    listOf(-0.5, 0.0, 4.8, 9.0, 14.3, 17.5, 19.2, 18.9, 14.5, 9.7, 4.7, 1.0)
val tempMadrid =
    listOf(6.3, 7.9, 11.2, 12.9, 16.7, 21.1, 24.7, 24.2, 20.3, 15.4, 9.9, 6.6)
val tempCaracas =
    listOf(27.5, 28.9, 29.6, 30.9, 31.7, 35.1, 33.8, 32.2, 31.3, 29.4, 28.9, 27.6)

// The df variable stores a DataFrame of three columns, including monthly records, tem
perature, and cities
val df = dataFrameOf(
    "Month" to months + months + months,
    "Temperature" to tempBerlin + tempMadrid + tempCaracas,
    "City" to List(12) { "Berlin" } + List(12) { "Madrid" } + List(12) { "Caracas" }
)
df.head(4)
```

You can see that the DataFrame has three columns: Month, Temperature, and City. The first four rows of the DataFrame contain records of the temperature in Berlin from January to April:

4 rows 4 rows × 3 columns

Month	Temperature	City
January	-0.5	Berlin
February	0	Berlin
March	4.8	Berlin
April	9	Berlin

To create a plot using the LPK library, you need to convert your data (`df`) into a `Map` type that stores the data in key-value pairs. You can easily convert a DataFrame into a `Map` using the `.toMap()` function:

```
val data = df.toMap()
```

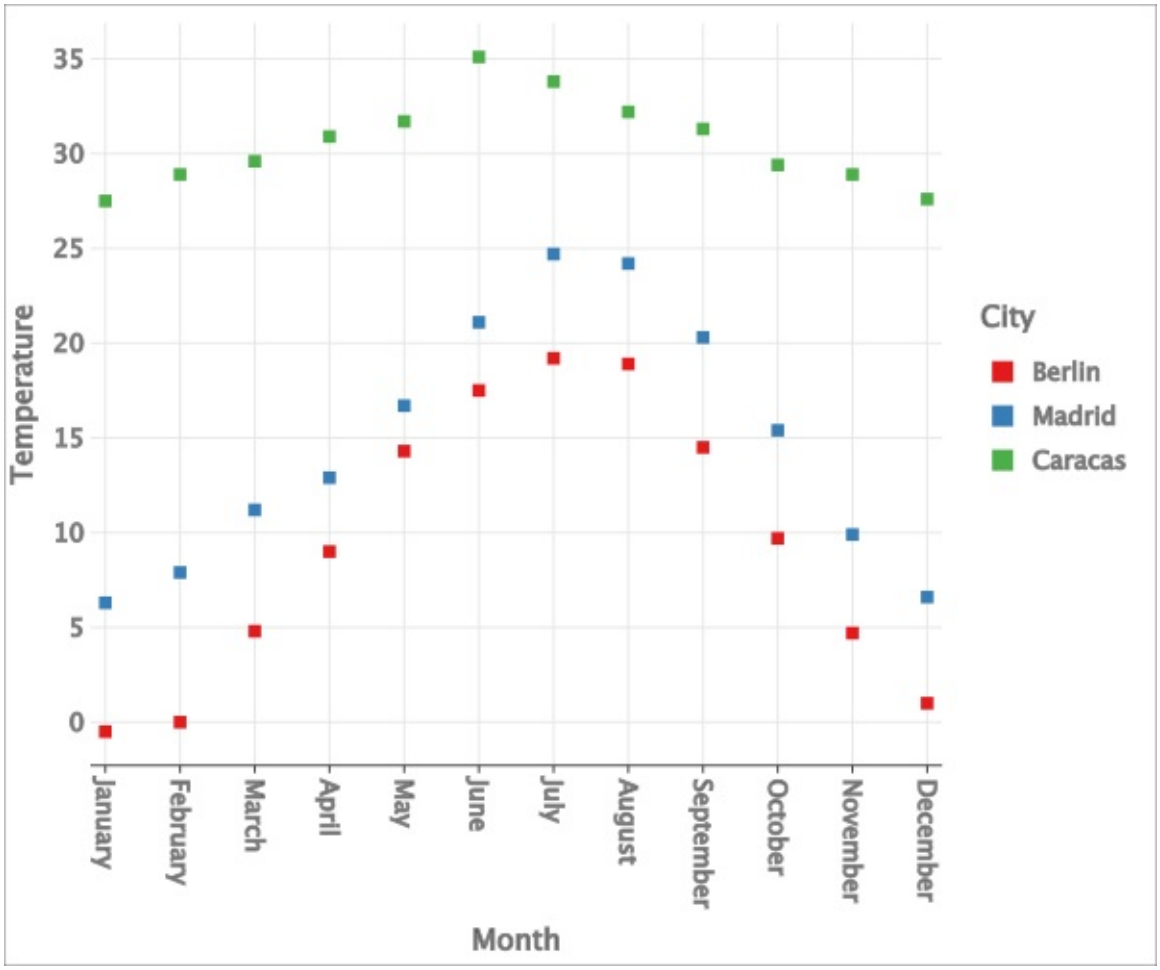
Create a scatter plot

Let's create a scatter plot in Kotlin Notebook with the LPK library.

Once you have your data in the `Map` format, use the `geomPoint()` function from the LPK library to generate the scatter plot. You can specify the values for the X and Y axes, as well as define categories and their color. Additionally, you can customize the plot's size and point shapes to suit your needs:

```
// Specifies X and Y axes, categories and their color, plot size, and plot type
val scatterPlot =
    letsPlot(data) { x = "Month"; y = "Temperature"; color = "City" } + ggsize(600, 500
) + geomPoint(shape = 15)
scatterPlot
```

Here's the result:

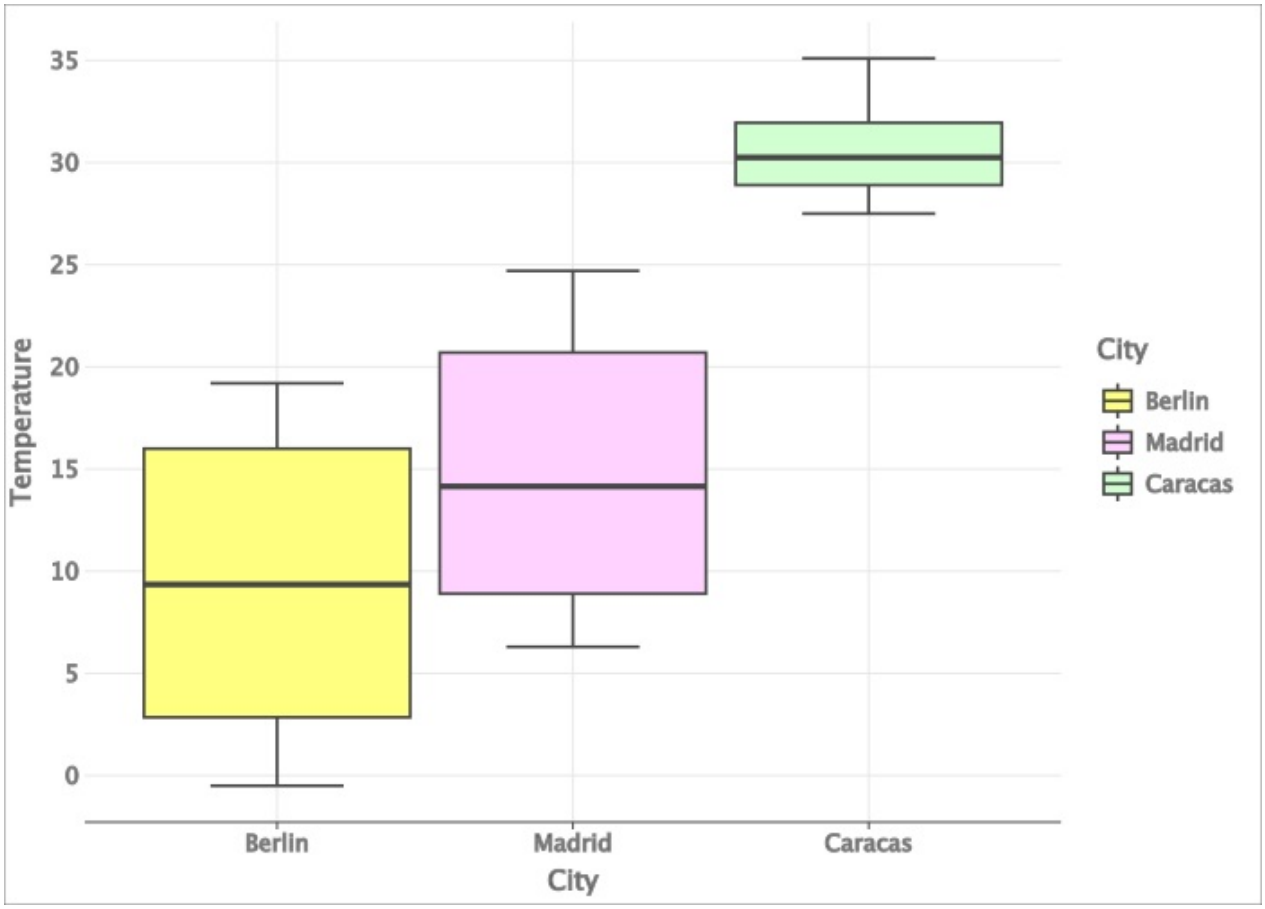


Create a box plot

Let's visualize the data in a box plot. Use the `geomBoxplot()` function from the LPK library to generate the plot and customize colors with the `scaleFillManual()` function:

```
// Specifies X and Y axes, categories, plot size, and plot type
val boxPlot = ggplot(data) { x = "City"; y = "Temperature" } + ggsize(700, 500) + geom
Boxplot { fill = "City" } +
    // Customizes colors
    scaleFillManual(values = listOf("light_yellow", "light_magenta", "light_green"))
boxPlot
```

Here's the result:



Create a 2D density plot

Now, let's create a 2D density plot to visualize the distribution and concentration of some random data.

Prepare the data for the 2D density plot

1. Import the dependencies to process the data and generate the plot:

```
%use lets-plot

@file:DependsOn("org.apache.commons:commons-math3:3.6.1")
import org.apache.commons.math3.distribution.MultivariateNormalDistribution
```

> For more information about importing dependencies to Kotlin Notebook, see the Kotlin Notebook documentation.

2. Paste and run the following code snippet in your Kotlin Notebook to create sets of 2D data points:

```
// Defines covariance matrices for three distributions
```

```
val cov0: Array<DoubleArray> = arrayOf(
    doubleArrayOf(1.0, -.8),
    doubleArrayOf(-.8, 1.0)
)

val cov1: Array<DoubleArray> = arrayOf(
    doubleArrayOf(1.0, .8),
    doubleArrayOf(.8, 1.0)
)

val cov2: Array<DoubleArray> = arrayOf(
    doubleArrayOf(10.0, .1),
    doubleArrayOf(.1, .1)
)

// Defines the number of samples
val n = 400

// Defines means for three distributions
val means0: DoubleArray = doubleArrayOf(-2.0, 0.0)
val means1: DoubleArray = doubleArrayOf(2.0, 0.0)
val means2: DoubleArray = doubleArrayOf(0.0, 1.0)

// Generates random samples from three multivariate normal distributions
val xy0 = MultivariateNormalDistribution(means0, cov0).sample(n)
val xy1 = MultivariateNormalDistribution(means1, cov1).sample(n)
val xy2 = MultivariateNormalDistribution(means2, cov2).sample(n)
```

From the code above, the `xy0`, `xy1`, and `xy2` variables store arrays with 2D (`x, y`) data points.

3. Convert your data into a `Map` type:

```
val data = mapOf(
    "x" to (xy0.map { it[0] } + xy1.map { it[0] } + xy2.map { it[0] }).toList(),
    "y" to (xy0.map { it[1] } + xy1.map { it[1] } + xy2.map { it[1] }).toList()
)
```

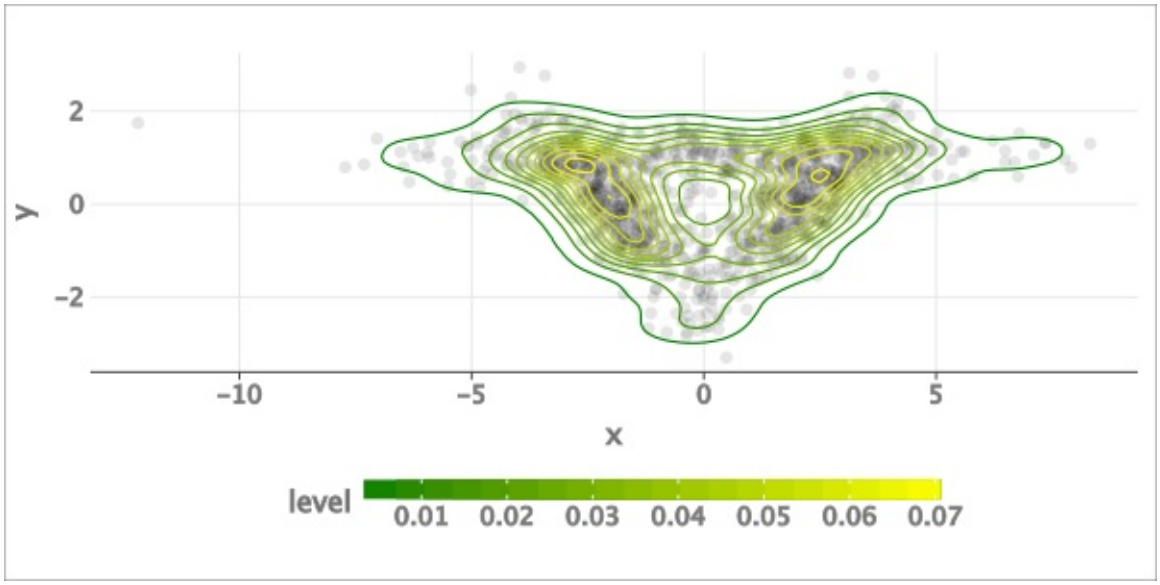
Generate the 2D density plot

Using the `Map` from the previous step, create a 2D density plot (`geomDensity2D`) with a scatter plot (`geomPoint`) in the background to better visualize the data points and outliers. You can use the `scaleColorGradient()` function to customize the scale of colors:

```
val densityPlot = letsPlot(data) { x = "x"; y = "y" } + ggsize(600, 300) + geomPoint(
    color = "black",
    alpha = .1
) + geomDensity2D { color = "..level.." } +
```

```
        scaleColorGradient(low = "dark_green", high = "yellow", guide = guideColorbar(
barHeight = 10, barWidth = 300)) +
        theme().legendPositionBottom()
densityPlot
```

Here's the result:



下一步做什么

- Explore more plot examples in the Lets-Plot for Kotlin's documentation.
- Check the Lets-Plot for Kotlin's API reference.
- Learn about transforming and visualizing data with Kotlin in the Kotlin DataFrame and Kandy library documentation.
- Find additional information about the Kotlin Notebook's usage and key features.

运行代码片段

Kotlin code is typically organized into projects with which you work in an IDE, a text editor, or another tool. However, if you want to quickly see how a function works or find an expression's value, there's no need to create a new project and build it. Check out these three handy ways to run Kotlin code instantly in different environments:

- Scratch files and worksheets in the IDE.
- Kotlin Playground in the browser.
- ki shell in the command line.

IDE: scratches and worksheets

IntelliJ IDEA and Android Studio support Kotlin scratch files and worksheets.

- *Scratch files* (or just *scratches*) let you create code drafts in the same IDE window as your project and run them on the fly. Scratches are not tied to projects; you can access and run all your scratches from any IntelliJ IDEA window on your OS.

  To create a Kotlin scratch, click **File** | **New** | **Scratch File** and select the **Kotlin** type.

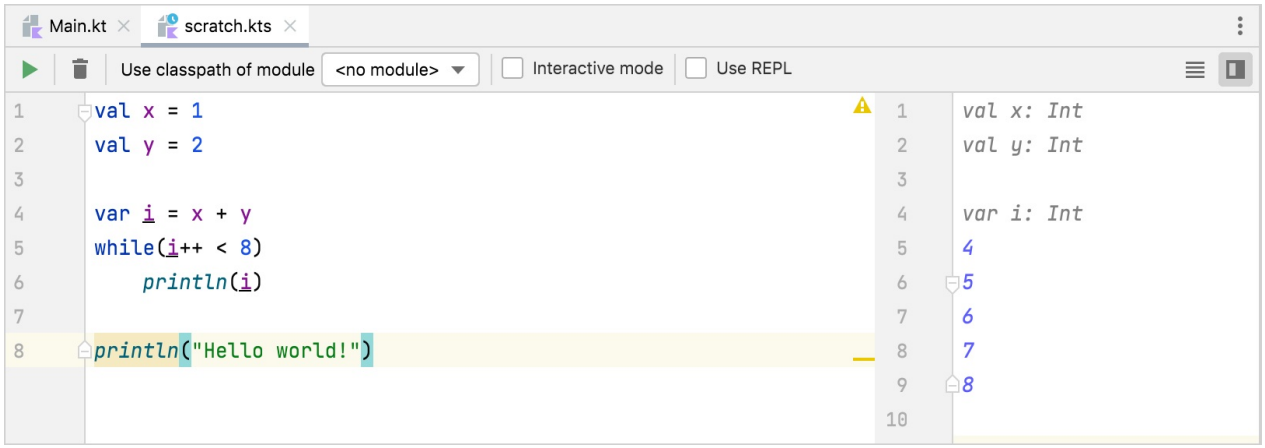
- *Worksheets* are project files: they are stored in project directories and tied to the project modules. Worksheets are useful for writing pieces of code that don't actually make a software unit but should still be stored together in a project, such as educational or demo materials.

  To create a Kotlin worksheet in a project directory, right-click the directory in the project tree and select **New** | **Kotlin Class/File** | **Kotlin Worksheet**.

  > Kotlin worksheets aren't supported in K2 mode. We're working on providing an alternative with similar functionality.
  >
  > 

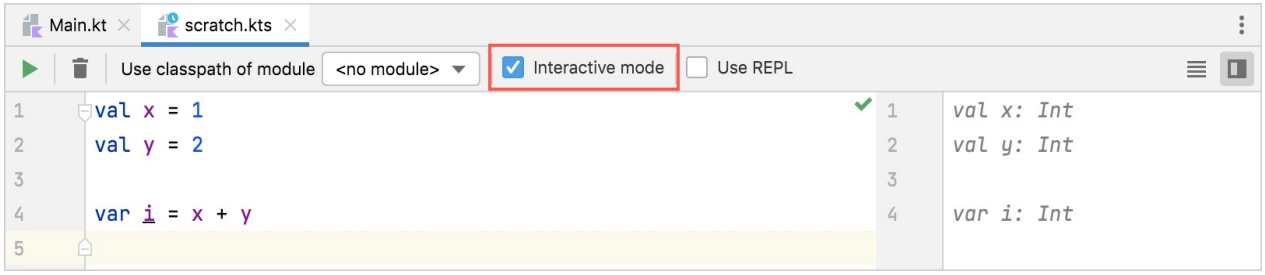
Syntax highlighting, auto-completion, and other IntelliJ IDEA code editing features are supported in scratches and worksheets. There's no need to declare the `main()` function – all the code you write is executed as if it were in the body of `main()`.

Once you have finished writing your code in a scratch or a worksheet, click **Run**. The execution results will appear in the lines opposite your code.



Interactive mode

The IDE can run code from scratches and worksheets automatically. To get execution results as soon as you stop typing, switch on **Interactive mode**.



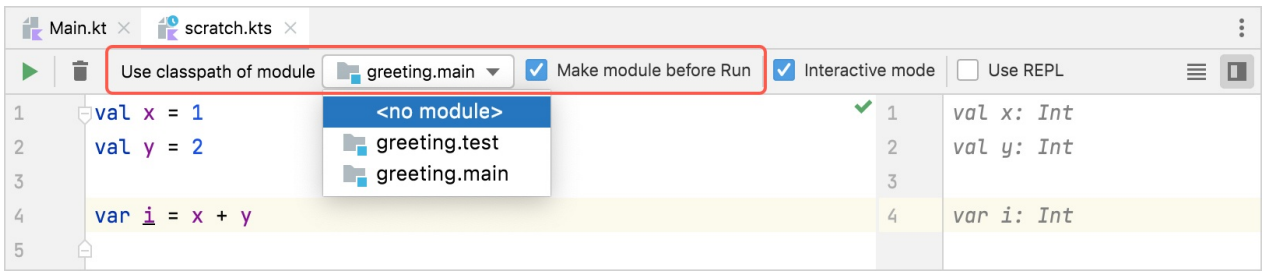
Use modules

You can use classes or functions from a Kotlin project in your scratches and worksheets.

Worksheets automatically have access to classes and functions from the module where they reside.

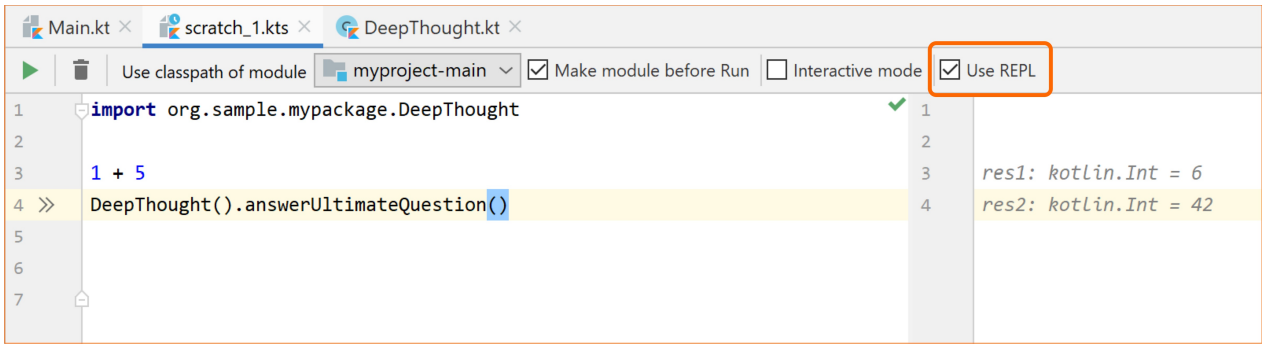
To use classes or functions from a project in a scratch, import them into the scratch file with the `import` statement, as usual. Then write your code and run it with the appropriate module selected in the **Use classpath of module** list.

Both scratches and worksheets use the compiled versions of connected modules. So, if you modify a module's source files, the changes will propagate to scratches and worksheets when you rebuild the module. To rebuild the module automatically before each run of a scratch or a worksheet, select **Make module before Run**.



Run as REPL

To evaluate each particular expression in a scratch or a worksheet, run it with **Use REPL** selected. The code lines will run sequentially, providing the results of each call. You can later use the results in the same file by reffering to their auto-generated `res*` names (they are shown in the corresponding lines).



Browser: Kotlin Playground

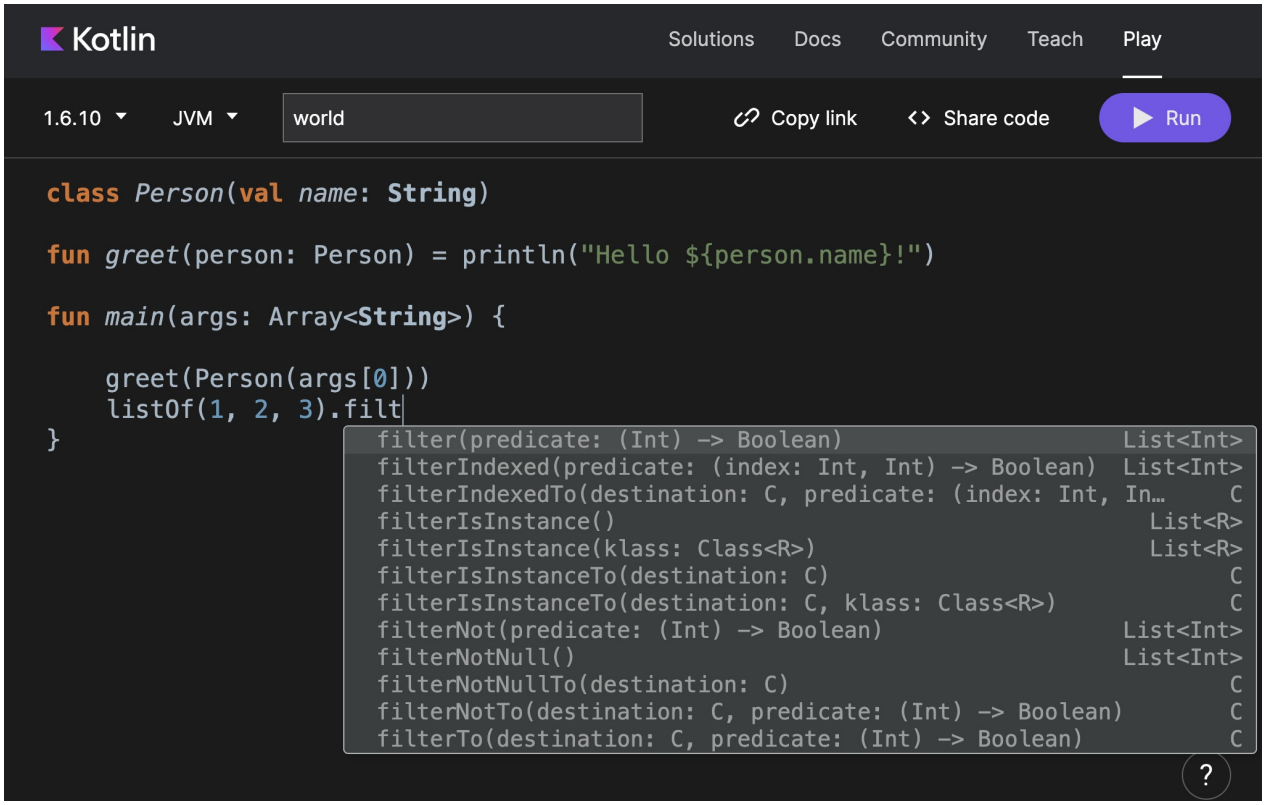
Kotlin Playground is an online application for writing, running, and sharing Kotlin code in your browser.

Write and edit code

In the Playground's editor area, you can write code just as you would in a source file:

- Add your own classes, functions, and top-level declarations in an arbitrary order.
- Write the executable part in the body of the `main()` function.

As in typical Kotlin projects, the `main()` function in the Playground can have the `args` parameter or no parameters at all. To pass program arguments upon execution, write them in the **Program arguments** field.

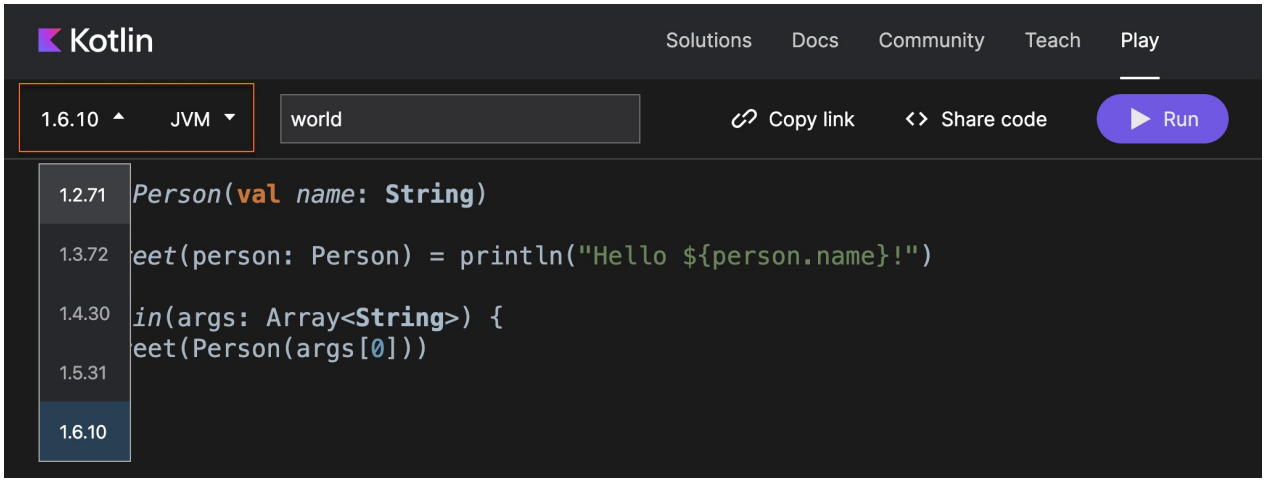


The Playground highlights the code and shows code completion options as you type. It automatically imports declarations from the standard library and `kotlinx.coroutines`.

Choose execution environment

The Playground provides ways to customize the execution environment:

- Multiple Kotlin versions, including available previews of future versions.
- Multiple backends to run the code in: JVM, JS (legacy or IR compiler, or Canvas), or JUnit.



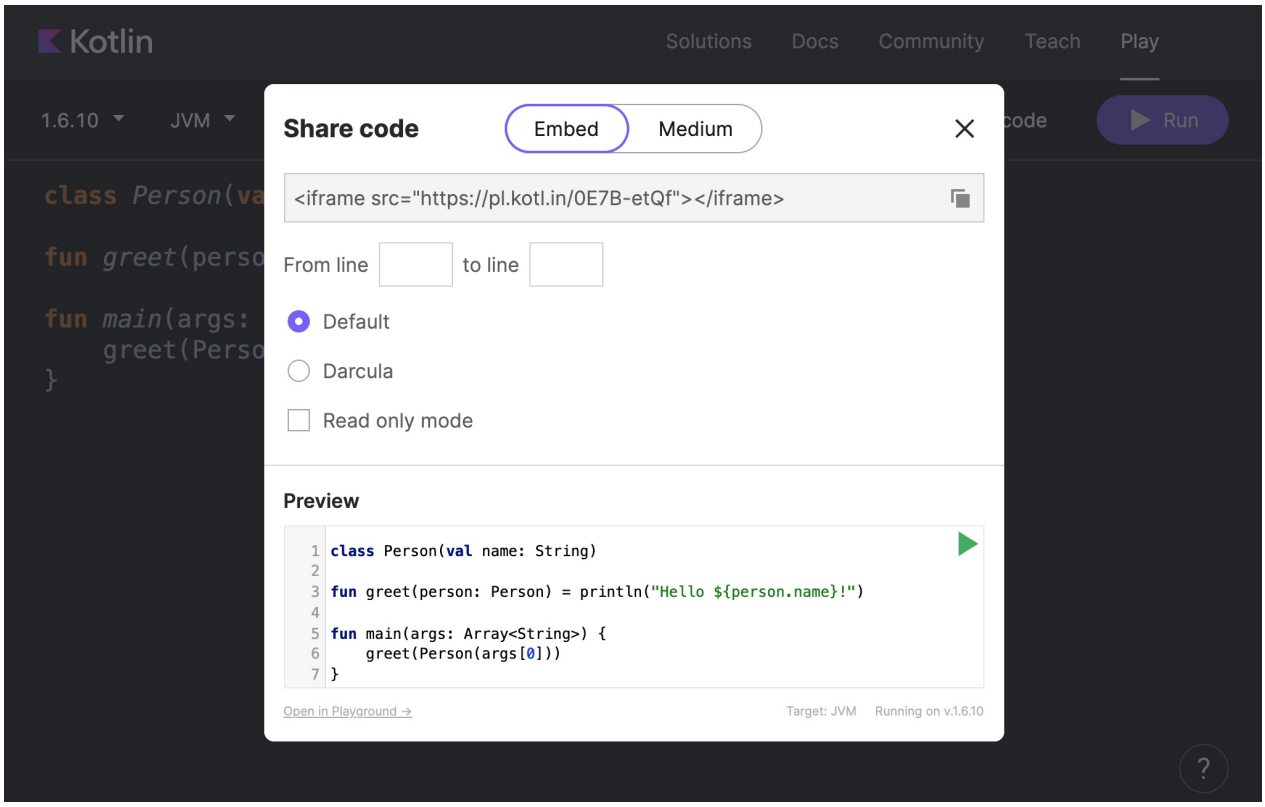
For JS backends, you can also see the generated JS code.



Share code online

Use the Playground to share your code with others – click **Copy link** and send it to anyone you want to show the code to.

You can also embed code snippets from the Playground into other websites and even make them runnable. Click **Share code** to embed your sample into any web page or into a Medium article.



Command line: ki shell

The ki shell (*Kotlin Interactive Shell*) is a command-line utility for running Kotlin code in the terminal. It's available for Linux, macOS, and Windows.

The ki shell provides basic code evaluation capabilities, along with advanced features such as:

- code completion
- type checks
- external dependencies
- paste mode for code snippets
- scripting support

See the ki shell GitHub repository for more details.

Install and run ki shell

To install the ki shell, download the latest version of it from GitHub and unzip it in the directory of your choice.

On macOS, you can also install the ki shell with Homebrew by running the following command:

```
brew install ki
```

To start the ki shell, run `bin/ki.sh` on Linux and macOS (or just `ki` if the ki shell was installed with Homebrew) or `bin\ki.bat` on Windows.

Once the shell is running, you can immediately start writing Kotlin code in your terminal. Type `:help` (or `:h`) to see the commands that are available in the ki shell.

Code completion and highlighting

The ki shell shows code completion options when you press **Tab**. It also provides syntax highlighting as you type. You can disable this feature by entering `:syntax off` .

```
ki-shell 0.4.5/1.6.10
type :h for help
[0] val x = 1 + 2 + 3                                                          ]
[1] var greeting = "Hello"                                                     ]
[2] listOf(1, 2, 3).fil                                                        ]
file                 filterIndexedTo(      filterIsInstanceTo(   filterNotNullTo(
filter {             filterIsInstance(     filterNot {           filterNotTo(
filterIndexed {      filterIsInstance()    filterNotNull()       filterTo(
```

When you press **Enter**, the ki shell evaluates the entered line and prints the result. Expression values are printed as variables with auto-generated names like `res*` . You can later use such variables in the code you run. If the construct entered is incomplete (for example, an `if` with a condition but without the body), the shell prints three dots and expects the remaining part.

```
ki-shell 0.4.5/1.6.10
type :h for help
[0] val greeting = "Hello"                                                     ]
[1] greeting + " world!"                                                       ]
res1: String = Hello world!
[2] println(res1)                                                              ]
Hello world!
[3] if (res1.length > 10)                                                      ]
...
```

Check an expression's type

For complex expressions or APIs that you don't know well, the ki shell provides the `:type` (or `:t`) command, which shows the type of an expression:

```
ki-shell 0.4.5/1.6.10
type :h for help
[0] :t sequenceOf("one", "two", "three").associateWith { it.length }
Map<String, Int>
[1]
```

Load code

If the code you need is stored somewhere else, there are two ways to load it and use it in the ki shell:

- Load a source file with the `:load` (or `:l`) command.
- Copy and paste the code snippet in paste mode with the `:paste` (or `:p`) command.

```
ki-shell 0.4.5/1.6.10
type :h for help
[0] :l Desktop/sourceFile.kt
[1] :ls
val variableFromFile: String
fun functionFromFile(): Unit
[2] println(variableFromFile)
Hello from file!
[3] functionFromFile()
Calling function from file
[4]
```

The `ls` command shows available symbols (variables and functions).

Add external dependencies

Along with the standard library, the ki shell also supports external dependencies. This lets you try out third-party libraries in it without creating a whole project.

To add a third-party library in the ki shell, use the `:dependsOn` command. By default, the ki shell works with Maven Central, but you can use other repositories if you connect them using the `:repository` command:

```
ki-shell 0.4.5/1.6.10
type :h for help
[0] :repository https://maven.pkg.jetbrains.space/public/p/kotlinx-html/maven
[1] :dependsOn org.jetbrains.kotlinx:kotlinx-html-jvm:0.7.3
[2] import kotlinx.html.*
[3] import kotlinx.html.stream.*
[4] val html = createHTML().html {
...body {
...h1 { "Hello" }
...}
...}
[9] html
res5: String = <html>
  <body>
    <h1></h1>
  </body>
</html>

[10]
```

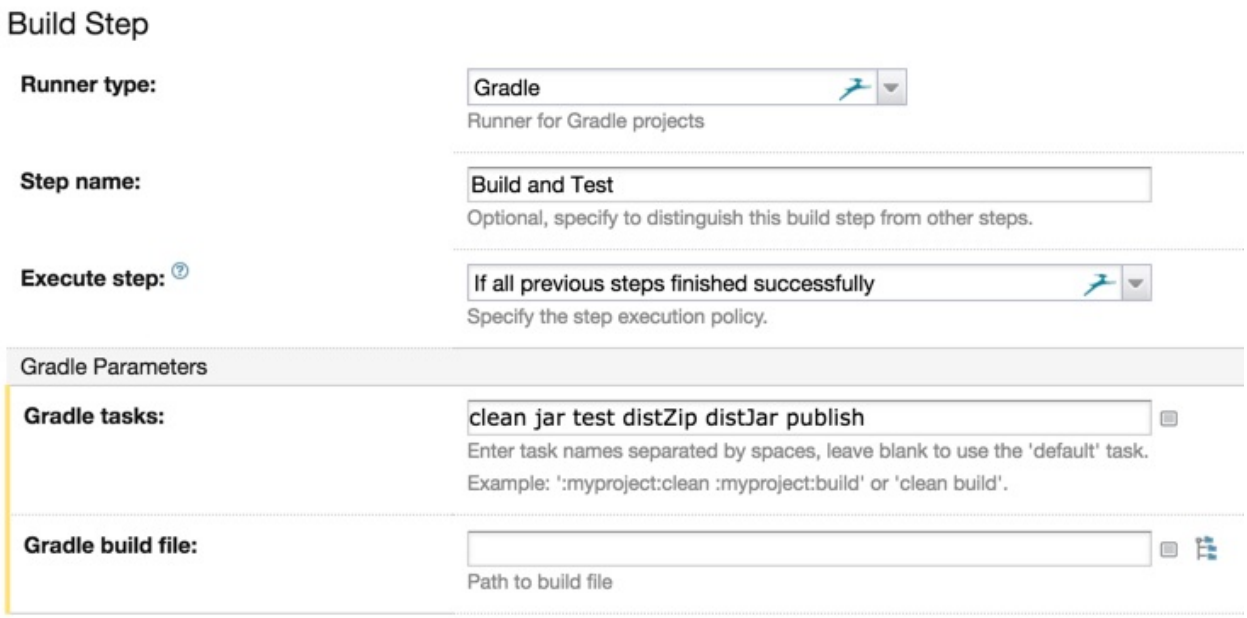
Kotlin 与 TeamCity 的持续集成

在本页，可以了解到如何设置 TeamCity 来构建 Kotlin 项目。 关于 TeamCity 的更多基础信息，请参阅其文档页， 其中包含有关安装，基本配置等信息。

Kotlin 可以使用不同的构建工具，因此，如果使用的是 Ant、Maven 或 Gradle 等标准工具，那么 Kotlin 项目的设置过程与任何其他语言或库都没有什么不同。 在使用 IntelliJ IDEA 的内部构建系统时，有一些小要求与不同之处， TeamCity 也支持该系统。

Gradle、Maven 与 Ant

如果使用 Ant、Maven 或 Gradle，它们的设置过程并不复杂。所需要做的就是定义构建步骤。例如，如果使用 Gradle，那么只需定义所需的参数，比如步骤名与需要为 Runner 类型执行的 Gradle 任务。



由于 Kotlin 所需的所有依赖项均在 Gradle 文件中定义，因此无需进行其他特殊配置即可使 Kotlin 正常运行。

如果使用 Ant 或 Maven，则应用相同的配置。唯一的区别是 Runner 类型分别是 Ant 或 Maven。

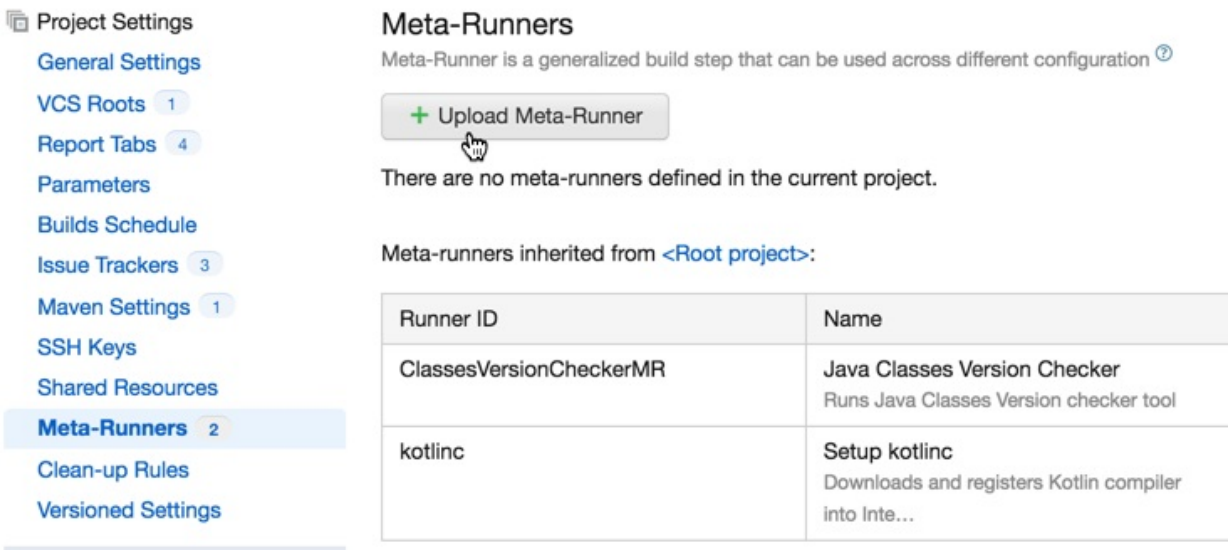
IntelliJ IDEA 的构建系统

如果在 TeamCity 中使用 IntelliJ IDEA 的构建系统，那么需要确保 IntelliJ IDEA 使用的 Kotlin 版本与 TeamCity 运行的版本相同。 可能需要下载 Kotlin 插件的特定版本， 并将其安装在 TeamCity 上。

幸运的是，已经有一个 Meta-Runner 可以处理大多数手工操作。 如果不熟悉 TeamCity Meta-Runner 的概念，请参阅文档。 这是无需编写插件即可引入自定义 Runner 的非常简单而强大的方法。

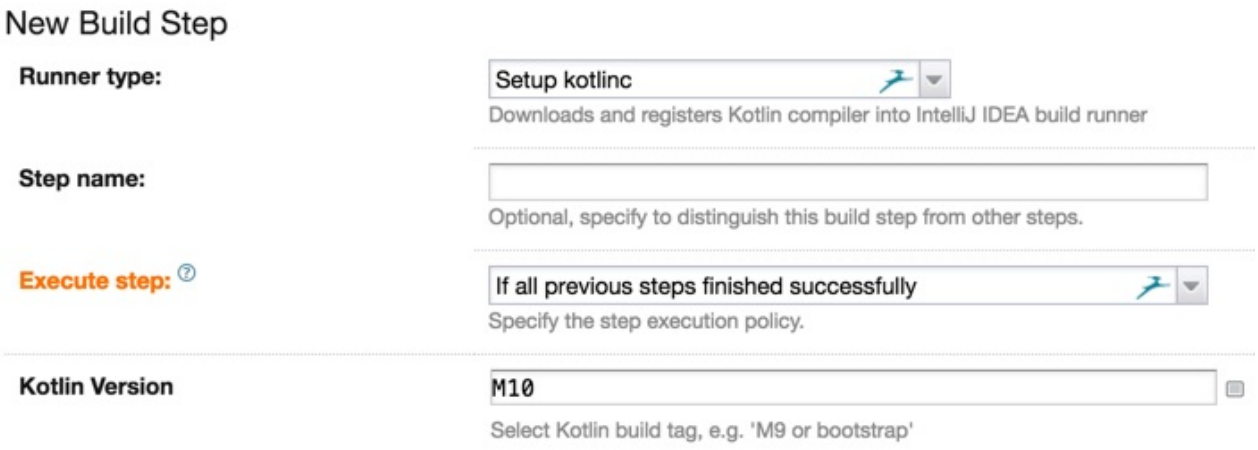
下载并安装 Meta-Runner

The meta-runner for Kotlin is available on GitHub. Download that meta-runner and import it from the TeamCity user interface



设置 Kotlin 编译器获取步骤

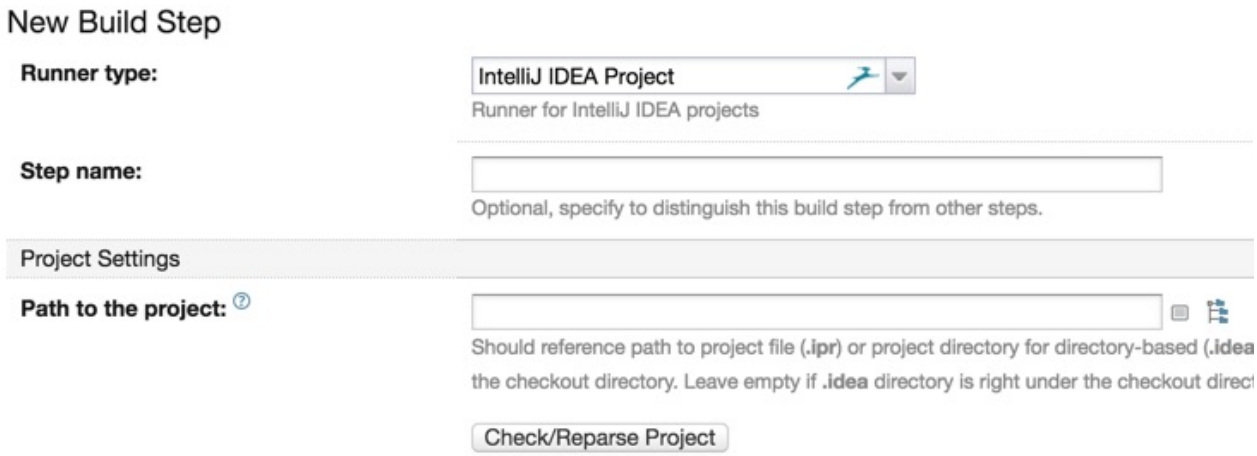
基本上，此步骤仅限于定义步骤名称和所需的 Kotlin 版本。并可以使用标签。



运行程序将根据 IntelliJ IDEA 项目中的路径设置，将属性 `system.path.macro.KOTLIN.BUNDLED` 的值设置为正确的值。 但是，这个值需要在 TeamCity 中定义（可以设置为任何值）。 因此，需要将其定义为系统变量。

设置 Kotlin 编译步骤

最后一步是定义项目的实际编译，该项目使用标准的 IntelliJ IDEA Runner 类型。



这样，项目现在应该构建并生成相应的工件。

其他 CI 服务器

如果使用与 TeamCity 不同的持续集成工具，只要它支持任何构建工具或者调用命令行工具，那么应该都可以将 Kotlin 编译和自动化作为 CI 流程的一部分。

编写 Kotlin 代码文档：KDoc

用来编写 Kotlin 代码文档的语言（相当于 Java 的 Javadoc）称为 **KDoc**。本质上 KDoc 是将 Javadoc 的块标签（block tags）语法（扩展为支持 Kotlin 的特定构造）和 Markdown 的内联标记（inline markup）结合在一起。

> Kotlin 的文档引擎：Dokka，理解 KDoc 并可以用于生成各种格式的文档。 更多信息请参阅 Dokka 文档。
>
> 

KDoc 语法

像 Javadoc 一样，KDoc 注释也以 `/**` 开头、以 `*/` 结尾。注释的每一行可以以星号开头，该星号不会当作注释内容的一部分。

按惯例来说，文档文本的第一段（到第一行空白行结束）是该元素的总体描述，接下来的注释是详细描述。

每个块标签都以一个新行开始且以 `@` 字符开头。

以下是使用 KDoc 编写类文档的一个示例：

```
/**
 * 一组*成员*。
 *
 * 这个类没有有用的逻辑；它只是一个文档示例。
 *
 * @param T 这个组中的成员的类型。
 * @property name 这个组的名称。
 * @constructor 创建一个空组。
 */
class Group<T>(val name: String) {
    /**
     * 将 [member] 添加到这个组。
     * @return 这个组的新大小。
     */
    fun add(member: T): Int { …… }
}
```

块标签

KDoc 目前支持以下块标签（block tags）：

@param 名称

用于函数的值参数或者类、属性或函数的类型参数。 为了更好地将参数名称与描述分开，如果你愿意，可以将参数的名称括在方括号中。因此，以下两种语法是等效的：

```
@param name 描述。
@param[name] 描述。
```

@return

用于函数的返回值。

@constructor

用于类的主构造函数。

@receiver

用于扩展函数的接收者。

@property 名称

用于类中具有指定名称的属性。这个标签可用于在主构造函数中声明的属性，当然直接在属性定义的前面放置 doc 注释会很别扭。

@throws 类、 @exception 类

用于方法可能抛出的异常。因为 Kotlin 没有受检异常，所以也没有期望所有可能的异常都写文档，但是当它会为类的用户提供有用的信息时， 仍然可以使用这个标签。

@sample 标识符

将具有指定限定的名称的函数的主体嵌入到当前元素的文档中， 以显示如何使用该元素的示例。

@see 标识符

将到指定类或方法的链接添加到文档的另请参见块。

@author

指定要编写文档的元素的作者。

@since

指定要编写文档的元素引入时的软件版本。

@suppress

从生成的文档中排除元素。可用于不是模块的官方 API 的一部分但还是必须在对外可见的元素。

> KDoc 不支持 `@deprecated` 这个标签。作为替代，请使用 `@Deprecated` 注解。
>
> 

内联标记

对于内联标记，KDoc 使用常规 Markdown 语法，扩展了支持用于链接到代码中其他元素的简写语法。

链接到元素

要链接到另一个元素（类、方法、属性或参数），只需将其名称放在方括号中：

```
为此目的，请使用方法 [foo]。
```

如果要为链接指定自定义标签（label），add it in another set of square brackets before the element link:

```
为此目的，请使用[这个方法][foo]。
```

你还可以在元素链接中使用限定的名称。请注意，与 Javadoc 不同，限定的名称总是使用点字符来分隔组件，即使在方法名称之前：

```
使用 [kotlin.reflect.KClass.properties] 来枚举类的属性。
```

元素链接中的名称与正写文档的元素内使用该名称使用相同的规则解析。 特别是，这意味着如果你已将名称导入当前文件，那么当你在 KDoc 注释中使用它时， 不需要再对其进行完整限定。

请注意 KDoc 没有用于解析链接中的重载成员的任何语法。 因为 Kotlin 的文档生成工具将一个函数的所有重载的文档放在同一页面上，标识一个特定的重载函数并不是链接生效所必需的。

External links

To add an external link, use the typical Markdown syntax:

```
For more information about KDoc syntax, see [KDoc](<example-URL>).
```

下一步做什么？

了解如何使用 Kotlin 的文档生成工具：Dokka。

Kotlin 与 OSGi

如果想要在项目中启用 Kotlin OSGi 支持，需要引入 `kotlin-osgi-bundle` 而不是常规的 Kotlin 库。 建议删除 `kotlin-runtime`、 `kotlin-stdlib` 与 `kotlin-reflect` 依赖， 因为 `kotlin-osgi-bundle` 已经包含了所有这些。当引入外部 Kotlin 库时也应该注意。 大多数常规 Kotlin 依赖不是 OSGi-就绪的，所以不应该使用且应从项目中删除。

Maven

将 Kotlin OSGi 包引入到 Maven 项目中：

```
<dependencies>
    <dependency>
        <groupId>org.jetbrains.kotlin</groupId>
        <artifactId>kotlin-osgi-bundle</artifactId>
        <version>${kotlin.version}</version>
    </dependency>
</dependencies>
```

从外部库中排除标准库（注意“星排除”只在 Maven 3 中有效）：

```
<dependency>
    <groupId>some.group.id</groupId>
    <artifactId>some.library</artifactId>
    <version>some.library.version</version>

    <exclusions>
        <exclusion>
            <groupId>org.jetbrains.kotlin</groupId>
            <artifactId>*</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

Gradle

将 `kotlin-osgi-bundle` 引入到 Gradle 项目中：

> 【Kotlin】

```
dependencies {
    implementation(kotlin("osgi-bundle"))
}
```

> 【Groovy】

```
dependencies {
    implementation "org.jetbrains.kotlin:kotlin-osgi-bundle:2.1.21"
}
```

要排除作为传递依赖的默认 Kotlin 库，你可以使用以下方法：

> 【Kotlin】

```
dependencies {
    implementation("some.group.id:some.library:someversion") {
        exclude(group = "org.jetbrains.kotlin")
    }
}
```

> 【Groovy】

```
dependencies {
    implementation('some.group.id:some.library:someversion') {
        exclude group: 'org.jetbrains.kotlin'
    }
}
```

FAQ

为什么不只是添加必需的清单选项到所有 **Kotlin** 库

尽管它是提供 OSGi 支持的最好的方式，遗憾的是现在做不到，是因为不能轻易消除的所谓的“包拆分”问题并且这么大的变化不可能现在规划。有 `Require-Bundle` 功能，但它也不是最好的选择，不推荐使用。 所以决定为 OSGi 做一个单独的构件。

编译器与插件

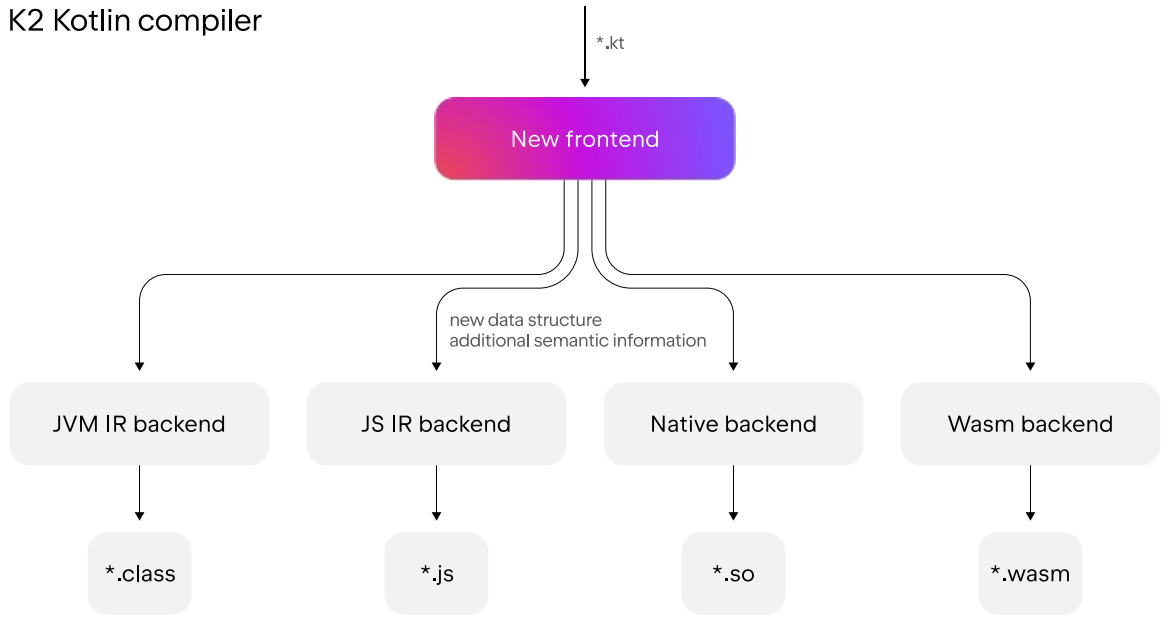
- 编译器
  - K2 编译器迁移指南
  - Kotlin 命令行编译器
  - Kotlin 编译器选项
- 编译器插件
  - 全开放编译器插件
  - No-arg 编译器插件
  - 带有接收者的 SAM 编译器插件
  - kapt 编译器插件
  - Lombok 编译器插件
  - Power-assert
- Compose 编译器
  - Compose 编译器迁移指南
  - Compose 编译器选项 DSL
- Kotlin 符号处理（KSP）API
  - KSP 概述
  - KSP 快速入门
  - 为什么选用 KSP
  - KSP 示例
  - KSP 如何为 Kotlin 代码建模
  - Java 注解处理对应到 KSP 参考
  - 增量处理
  - 多轮次处理
  - KSP 与 Kotlin 多平台
  - 在命令行运行 KSP
  - 常见问题

编译器

- K2 编译器迁移指南
- Kotlin 命令行编译器
- Kotlin 编译器选项

K2 编译器迁移指南

As the Kotlin language and ecosystem have continued to evolve, so has the Kotlin compiler. The first step was the introduction of the new JVM and JS IR (Intermediate Representation) backends that share logic, simplifying code generation for targets on different platforms. Now, the next stage of its evolution brings a new frontend known as K2.



With the arrival of the K2 compiler, the Kotlin frontend has been completely rewritten and features a new, more efficient architecture. The fundamental change the new compiler brings is the use of one unified data structure that contains more semantic information. This frontend is responsible for performing semantic analysis, call resolution, and type inference.

The new architecture and enriched data structure enables the K2 compiler to provide the following benefits:

- **Improved call resolution and type inference**. The compiler behaves more consistently and understands your code better.
- **Easier introduction of syntactic sugar for new language features**. In the future, you'll be able to use more concise, readable code when new features are introduced.
- **Faster compilation times**. Compilation times can be significantly faster.
- **Enhanced IDE performance**. Starting with 2025.1, IntelliJ IDEA uses K2 mode to analyze your Kotlin code, increasing stability and providing performance improvements. For more information, see Support in IDEs.

This guide:

- Explains the benefits of the new K2 compiler.
- Highlights changes you might encounter during migration and how to adapt your code accordingly.
- Describes how you can roll back to the previous version.

> The new K2 compiler is enabled by default starting with 2.0.0. For more information on the new features provided in Kotlin 2.0.0, as well as the new K2 compiler, see What's new in Kotlin 2.0.0.
>
> 

Performance improvements

To evaluate the performance of the K2 compiler, we ran performance tests on two open-source projects: Anki-Android and Exposed. Here are the key performance improvements that we found:

- The K2 compiler brings up to 94% compilation speed gains. For example, in the Anki-Android project, clean build times were reduced from 57.7 seconds in Kotlin 1.9.23 to 29.7 seconds in Kotlin 2.0.0.
- The initialization phase is up to 488% faster with the K2 compiler. For example, in the Anki-Android project, the initialization phase for incremental builds was cut from 0.126 seconds in Kotlin 1.9.23 to just 0.022 seconds in Kotlin 2.0.0.
- The Kotlin K2 compiler is up to 376% quicker in the analysis phase compared to the previous compiler. For example, in the Anki-Android project, analysis times for incremental builds were slashed from 0.581 seconds in Kotlin 1.9.23 to only 0.122 seconds in Kotlin 2.0.0.

For more details on these improvements and to learn more about how we analyzed the performance of the K2 compiler, see our blog post.

Language feature improvements

The Kotlin K2 compiler improves language features related to smart-casting and Kotlin Multiplatform.

智能转换

The Kotlin compiler can automatically cast an object to a type in specific cases, saving you the trouble of having to explicitly specify it yourself. This is called smart-casting. The Kotlin K2 compiler now performs smart casts in even more scenarios than before.

In Kotlin 2.0.0, we've made improvements related to smart casts in the following areas:

- Local variables and further scopes
- Type checks with the logical `or` operator
- Inline functions
- Properties with function types
- Exception handling
- Increment and decrement operators

Local variables and further scopes

Previously, if a variable was evaluated as not `null` within an `if` condition, the variable would be smart-cast. Information about this variable would then be shared further within the scope of the `if` block.

However, if you declared the variable **outside** the `if` condition, no information about the variable would be available within the `if` condition, so it couldn't be smart-cast. This behavior was also seen with `when` expressions and `while` loops.

From Kotlin 2.0.0, if you declare a variable before using it in your `if`, `when`, or `while` condition, then any information collected by the compiler about the variable will be accessible in the corresponding block for smart-casting.

This can be useful when you want to do things like extract boolean conditions into variables. Then, you can give the variable a meaningful name, which will improve your code readability and make it possible to reuse the variable later in your code. For example:

```
class Cat {
    fun purr() {
        println("Purr purr")
    }
}

fun petAnimal(animal: Any) {
    val isCat = animal is Cat
    if (isCat) {
        // In Kotlin 2.0.0, the compiler can access
        // information about isCat, so it knows that
        // animal was smart-cast to the type Cat.
        // Therefore, the purr() function can be called.
        // In Kotlin 1.9.20, the compiler doesn't know
        // about the smart cast, so calling the purr()
        // function triggers an error.
        animal.purr()
    }
}

fun main(){
    val kitty = Cat()
```

```
    petAnimal(kitty)
    // Purr purr
}
```

Type checks with the logical or operator

In Kotlin 2.0.0, if you combine type checks for objects with an `or` operator (`||`), a smart cast is made to their closest common supertype. Before this change, a smart cast was always made to the `Any` type.

In this case, you still had to manually check the object type afterward before you could access any of its properties or call its functions. For example:

```
interface Status {
    fun signal() {}
}

interface Ok : Status
interface Postponed : Status
interface Declined : Status

fun signalCheck(signalStatus: Any) {
    if (signalStatus is Postponed || signalStatus is Declined) {
        // signalStatus is smart-cast to a common supertype Status
        signalStatus.signal()
        // Prior to Kotlin 2.0.0, signalStatus is smart cast
        // to type Any, so calling the signal() function triggered an
        // Unresolved reference error. The signal() function can only
        // be called successfully after another type check:

        // check(signalStatus is Status)
        // signalStatus.signal()
    }
}
```

> The common supertype is an **approximation** of a union type. Union types are not currently supported in Kotlin.
>
> 

Inline functions

In Kotlin 2.0.0, the K2 compiler treats inline functions differently, allowing it to determine in combination with other compiler analyses whether it's safe to smart-cast.

Specifically, inline functions are now treated as having an implicit `callsInPlace` contract. This means that any lambda functions passed to an inline function are called in place. Since lambda functions are called in place, the compiler knows that a lambda function can't leak references to any variables contained within its function body.

The compiler uses this knowledge along with other compiler analyses to decide whether it's safe to smart-cast any of the captured variables. For example:

```
interface Processor {
    fun process()
}

inline fun inlineAction(f: () -> Unit) = f()

fun nextProcessor(): Processor? = null

fun runProcessor(): Processor? {
    var processor: Processor? = null
    inlineAction {
        // In Kotlin 2.0.0, the compiler knows that processor
        // is a local variable and inlineAction() is an inline function, so
        // references to processor can't be leaked. Therefore, it's safe
        // to smart-cast processor.

        // If processor isn't null, processor is smart-cast
        if (processor != null) {
            // The compiler knows that processor isn't null, so no safe call
            // is needed
            processor.process()

            // In Kotlin 1.9.20, you have to perform a safe call:
            // processor?.process()
        }

        processor = nextProcessor()
    }

    return processor
}
```

Properties with function types

In previous versions of Kotlin, there was a bug that meant that class properties with a function type weren't smart-cast. We fixed this behavior in Kotlin 2.0.0 and the K2 compiler. For example:

```
class Holder(val provider: (() -> Unit)?) {
    fun process() {
        // In Kotlin 2.0.0, if provider isn't null,
```

```
        // it is smart-cast
        if (provider != null) {
            // The compiler knows that provider isn't null
            provider()

            // In 1.9.20, the compiler doesn't know that provider isn't
            // null, so it triggers an error:
            // Reference has a nullable type '(() -> Unit)?', use explicit '?.invoke()
' to make a function-like call instead
        }
    }
}
```

This change also applies if you overload your `invoke` operator. For example:

```
interface Provider {
    operator fun invoke()
}

interface Processor : () -> String

class Holder(val provider: Provider?, val processor: Processor?) {
    fun process() {
        if (provider != null) {
            provider()
            // In 1.9.20, the compiler triggers an error:
            // Reference has a nullable type 'Provider?', use explicit '?.invoke()' to
 make a function-like call instead
        }
    }
}
```

Exception handling

In Kotlin 2.0.0, we've made improvements to exception handling so that smart cast information can be passed on to `catch` and `finally` blocks. This change makes your code safer as the compiler keeps track of whether your object has a nullable type. For example:

```
//sampleStart
fun testString() {
    var stringInput: String? = null
    // stringInput is smart-cast to String type
    stringInput = ""
    try {
        // The compiler knows that stringInput isn't null
        println(stringInput.length)
        // 0

        // The compiler rejects previous smart cast information for
```

```
        // stringInput. Now stringInput has the String? type.
        stringInput = null

        // Trigger an exception
        if (2 > 1) throw Exception()
        stringInput = ""
    } catch (exception: Exception) {
        // In Kotlin 2.0.0, the compiler knows stringInput
        // can be null, so stringInput stays nullable.
        println(stringInput?.length)
        // null

        // In Kotlin 1.9.20, the compiler says that a safe call isn't
        // needed, but this is incorrect.
    }
}
//sampleEnd
fun main() {
    testString()
}
```

Increment and decrement operators

Prior to Kotlin 2.0.0, the compiler didn't understand that the type of an object can change after using an increment or decrement operator. As the compiler couldn't accurately track the object type, your code could lead to unresolved reference errors. In Kotlin 2.0.0, this has been fixed:

```
interface Rho {
    operator fun inc(): Sigma = TODO()
}

interface Sigma : Rho {
    fun sigma() = Unit
}

interface Tau {
    fun tau() = Unit
}

fun main(input: Rho) {
    var unknownObject: Rho = input

    // Check if unknownObject inherits from the Tau interface
    // Note, it's possible that unknownObject inherits from both
    // Rho and Tau interfaces.
    if (unknownObject is Tau) {

        // Use the overloaded inc() operator from interface Rho.
        // In Kotlin 2.0.0, the type of unknownObject is smart-cast to
        // Sigma.
```

```
        ++unknownObject

        // In Kotlin 2.0.0, the compiler knows unknownObject has type
        // Sigma, so the sigma() function can be called successfully.
        unknownObject.sigma()

        // In Kotlin 1.9.20, the compiler doesn't perform a smart cast
        // when inc() is called so the compiler still thinks that
        // unknownObject has type Tau. Calling the sigma() function
        // throws a compile-time error.

        // In Kotlin 2.0.0, the compiler knows unknownObject has type
        // Sigma, so calling the tau() function throws a compile-time
        // error.
        unknownObject.tau()
        // Unresolved reference 'tau'

        // In Kotlin 1.9.20, since the compiler mistakenly thinks that
        // unknownObject has type Tau, the tau() function can be called,
        // but it throws a ClassCastException.
    }
}
```

Kotlin Multiplatform

There are improvements in the K2 compiler related to Kotlin Multiplatform in the following areas:

- Separation of common and platform sources during compilation
- Different visibility levels of expected and actual declarations

Separation of common and platform sources during compilation

Previously, the design of the Kotlin compiler prevented it from keeping common and platform source sets separate at compile time. As a consequence, common code could access platform code, which resulted in different behavior between platforms. In addition, some compiler settings and dependencies from common code used to leak into platform code.

In Kotlin 2.0.0, our implementation of the new Kotlin K2 compiler included a redesign of the compilation scheme to ensure strict separation between common and platform source sets. This change is most noticeable when you use expected and actual functions. Previously, it was possible for a function call in your common code to resolve to a function in platform code. For example:

Common code	Platform code
```kotlin fun foo(x: Any) = println("common foo") fun exampleFunction() { foo(42) } ```	```kotlin // JVM fun foo(x: Int) = println("platform foo") // JavaScript // There is no foo() function overload on the JavaScript platform ```

In this example, the common code has different behavior depending on which platform it is run on:

- On the JVM platform, calling the `foo()` function in the common code results in the `foo()` function from the platform code being called as `platform foo`.
- On the JavaScript platform, calling the `foo()` function in the common code results in the `foo()` function from the common code being called as `common foo`, as there is no such function available in the platform code.

In Kotlin 2.0.0, common code doesn't have access to platform code, so both platforms successfully resolve the `foo()` function to the `foo()` function in the common code: `common foo`.

In addition to the improved consistency of behavior across platforms, we also worked hard to fix cases where there was conflicting behavior between IntelliJ IDEA or Android Studio and the compiler. For instance, when you used expected and actual classes, the following would happen:

Common code	Platform code
```kotlin expect class Identity { fun confirmIdentity(): String } fun common() { // Before 2.0.0, it triggers an IDE-only error Identity().confirmIdentity() // RESOLUTION_TO_CLASSIFIER : Expected class Identity has no default constructor. } ```	```kotlin actual class Identity { actual fun confirmIdentity() = "expect class fun: jvm" } ```

In this example, the expected class `Identity` has no default constructor, so it can't be called successfully in common code. Previously, an error was only reported by the IDE, but the code still compiled successfully on the JVM. However, now the compiler correctly reports an error:

```
Expected class 'expect class Identity : Any' does not have default constructor
```

When resolution behavior doesn't change

We're still in the process of migrating to the new compilation scheme, so the resolution behavior is still the same when you call functions that aren't within the same source set. You'll notice this difference mainly when you use overloads from a multiplatform library in your common code.

Suppose you have a library, which has two `whichFun()` functions with different signatures:

```
// Example library

// MODULE: common
fun whichFun(x: Any) = println("common function")

// MODULE: JVM
fun whichFun(x: Int) = println("platform function")
```

If you call the `whichFun()` function in your common code, the function that has the most relevant argument type in the library will be resolved:

```
// A project that uses the example library for the JVM target

// MODULE: common
fun main(){
    whichFun(2)
    // platform function
}
```

In comparison, if you declare the overloads for `whichFun()` within the same source set, the function from the common code will be resolved because your code doesn't have access to the platform-specific version:

```
// Example library isn't used

// MODULE: common
fun whichFun(x: Any) = println("common function")

fun main(){
    whichFun(2)
    // common function
}

// MODULE: JVM
fun whichFun(x: Int) = println("platform function")
```

Similar to multiplatform libraries, since the `commonTest` module is in a separate source set, it also still has access to platform-specific code. Therefore, the resolution of calls to functions in the `commonTest` module exhibits the same behavior as in the old compilation scheme.

In the future, these remaining cases will be more consistent with the new compilation scheme.

Different visibility levels of expected and actual declarations

Before Kotlin 2.0.0, if you used expected and actual declarations in your Kotlin Multiplatform project, they had to have the same visibility level. Kotlin 2.0.0 now also supports different visibility levels but **only** if the actual declaration is *more* permissive than the expected declaration. For example:

```
expect internal class Attribute // Visibility is internal
actual class Attribute          // Visibility is public by default,
                                // which is more permissive
```

Similarly, if you are using a type alias in your actual declaration, the visibility of the **underlying type** should be the same or more permissive than the expected declaration. For example:

```
expect internal class Attribute                 // Visibility is internal
internal actual typealias Attribute = Expanded

class Expanded                                  // Visibility is public by default,
                                                // which is more permissive
```

How to enable the Kotlin K2 compiler

Starting with Kotlin 2.0.0, the Kotlin K2 compiler is enabled by default.

To upgrade the Kotlin version, change it to 2.0.0 or a later release in your Gradle and Maven build scripts.

To have the best experience with IntelliJ IDEA or Android Studio, consider enabling K2 mode in your IDE.

Use Kotlin build reports with Gradle

Kotlin build reports provide information about the time spent in different compilation phases for Kotlin compiler tasks, as well as which compiler and Kotlin version were used, and whether the compilation was incremental. These build reports are useful for assessing your build performance. They offer more insight into the Kotlin compilation pipeline than Gradle build scans do because they give you an overview of the performance of all Gradle tasks.

How to enable build reports

To enable build reports, declare where you'd like to save the build report output in your `gradle.properties` file:

```
kotlin.build.report.output=file
```

The following values and their combinations are available for the output:

Option	Description
`file`	Saves build reports in a human-readable format to a local file. By default, it's `${project_folder}/build/reports/kotlin-build/${project_name}-timestamp.txt`
`single_file`	Saves build reports in a format of an object to a specified local file.

`build_scan`	Saves build reports in the `custom values` section of the build scan. Note that the Gradle Enterprise plugin limits the number of custom values and their length. In big projects, some values could be lost.
`http`	Posts build reports using HTTP(S). The POST method sends metrics in JSON format. You can see the current version of the sent data in the Kotlin repository. You can find samples of HTTP endpoints in this blog post
`json`	Saves build reports in JSON format to a local file. Set the location for your build reports in `kotlin.build.report.json.directory`. By default, it's name is `${project_name}-build-<date-time>-<index>.json`.

For more information on what is possible with build reports, see Build reports.

Support in IDEs

K2 mode in IntelliJ IDEA and Android Studio uses the K2 compiler to improve code analysis, code completion, and highlighting.

Starting with IntelliJ IDEA 2025.1, K2 mode is enabled by default.

In Android Studio, you can enable K2 mode starting with 2024.1 by following these steps:

1. Go to **Settings** | **Languages & Frameworks** | **Kotlin**.
2. Select the **Enable K2 mode** option.

Previous IDE behavior

If you want to go back to the previous IDE behavior, you can disable K2 mode:

1. Go to **Settings** | **Languages & Frameworks** | **Kotlin**.
2. Deselect the **Enable K2 mode** option.

> We plan to introduce Stable language features after Kotlin 2.1.0. Until then, you can continue to use the previous IDE features for code analysis, and you won't encounter any code highlighting issues due to unrecognized language features.
>
> 

Try the Kotlin K2 compiler in the Kotlin Playground

The Kotlin Playground supports Kotlin 2.0.0 and later releases. Check it out!

How to roll back to the previous compiler

To use the previous compiler in Kotlin 2.0.0 and later releases, either:

- In your `build.gradle.kts` file, set your language version to `1.9`.

  OR

- Use the following compiler option: `-language-version 1.9`.

Changes

With the introduction of the new frontend, the Kotlin compiler has undergone several changes. Let's start by highlighting the most significant modifications affecting your code, explaining what has changed and detailing best practices going forward. If you'd like to learn more, we've organized these changes into subject areas to facilitate your further reading.

This section highlights the following modifications:

- Immediate initialization of open properties with backing fields
- Deprecated synthetic setters on a projected receiver
- Forbidden use of inaccessible generic types
- Consistent resolution order of Kotlin properties and Java fields with the same name
- Improved null safety for Java primitive arrays
- Stricter rules for abstract members in expected classes

Immediate initialization of open properties with backing fields

What's changed?

In Kotlin 2.0, all `open` properties with backing fields must be immediately initialized; otherwise, you'll get a compilation error. Previously, only `open var` properties needed to be initialized right away, but now this extends to `open val` properties with backing fields too:

```
open class Base {
    open val a: Int
    open var b: Int

    init {
        // Error starting with Kotlin 2.0 that earlier compiled successfully
        this.a = 1 //Error: open val must have initializer
        // Always an error
        this.b = 1 // Error: open var must have initializer
    }
}

class Derived : Base() {
    override val a: Int = 2
```

```
    override var b = 2
}
```

This change makes the compiler's behavior more predictable. Consider an example where an `open val` property is overridden by a `var` property with a custom setter.

If a custom setter is used, deferred initialization can lead to confusion because it's unclear whether you want to initialize the backing field or to invoke the setter. In the past, if you wanted to invoke the setter, the old compiler couldn't guarantee that the setter would then initialize the backing field.

What's the best practice now?

We encourage you to always initialize open properties with backing fields, as we believe this practice is both more efficient and less error-prone.

However, if you don't want to immediately initialize a property, you can:

- Make the property `final`.
- Use a private backing property that allows for deferred initialization.

For more information, see the corresponding issue in YouTrack.

Deprecated synthetics setter on a projected receiver

What's changed?

If you use the synthetic setter of a Java class to assign a type that conflicts with the class's projected type, an error is triggered.

Suppose you have a Java class named `Container` that contains the `getFoo()` and `setFoo()` methods:

```
public class Container<E> {
    public E getFoo() {
        return null;
    }
    public void setFoo(E foo) {}
}
```

If you have the following Kotlin code, where instances of the `Container` class have projected types, using the `setFoo()` method will always generate an error. However, only from Kotlin 2.0.0 will the synthetic `foo` property trigger an error:

```
fun exampleFunction(starProjected: Container<*>, inProjected: Container<in Number>, sa
mpleString: String) {
    starProjected.setFoo(sampleString)
```

```
    // Error since Kotlin 1.0

    // Synthetic setter `foo` is resolved to the `setFoo()` method
    starProjected.foo = sampleString
    // Error since Kotlin 2.0.0

    inProjected.setFoo(sampleString)
    // Error since Kotlin 1.0

    // Synthetic setter `foo` is resolved to the `setFoo()` method
    inProjected.foo = sampleString
    // Error since Kotlin 2.0.0
}
```

What's the best practice now?

If you see that this change introduces errors in your code, you might wish to reconsider how you structure your type declarations. It could be that you don't need to use type projections, or perhaps you need to remove any assignments from your code.

For more information, see the corresponding issue in YouTrack.

Forbidden use of inaccessible generic types

What's changed?

Due to the new architecture of our K2 compiler, we've changed how we handle inaccessible generic types. Generally, you should never rely on inaccessible generic types in your code because this indicates a misconfiguration in your project's build configuration, preventing the compiler from accessing the necessary information to compile. In Kotlin 2.0.0, you can't declare or call a function literal with an inaccessible generic type, nor use a generic type with inaccessible generic type arguments. This restriction helps you avoid compiler errors later in your code.

For example, let's say that you declared a generic class in one module:

```
// Module one
class Node<V>(val value: V)
```

If you have another module (module two) with a dependency configured on module one, your code can access the `Node<V>` class and use it as a type in function types:

```
// Module two
fun execute(func: (Node<Int>) -> Unit) {}
// Function compiles successfully
```

However, if your project is misconfigured such that you have a third module (module three) that depends only on module two, the Kotlin compiler won't be able to access the `Node<V>` class in **module one** when compiling the third module. Now, any lambdas or anonymous functions in module three that use the `Node<V>` type trigger errors in Kotlin 2.0.0, thus preventing avoidable compiler errors, crashes, and run-time exceptions later in your code:

```
// Module three
fun test() {
    // Triggers an error in Kotlin 2.0.0, as the type of the implicit
    // lambda parameter (it) resolves to Node, which is inaccessible
    execute {}

    // Triggers an error in Kotlin 2.0.0, as the type of the unused
    // lambda parameter (_) resolves to Node, which is inaccessible
    execute { _ -> }

    // Triggers an error in Kotlin 2.0.0, as the type of the unused
    // anonymous function parameter (_) resolves to Node, which is inaccessible
    execute(fun (_) {})
}
```

In addition to function literals triggering errors when they contain value parameters of inaccessible generic types, errors also occur when a type has an inaccessible generic type argument.

For example, you have the same generic class declaration in module one. In module two, you declare another generic class: `Container<C>`. In addition, you declare functions in module two that use `Container<C>` with generic class `Node<V>` as a type argument:

Module one	Module two
```kotlin // Module one class Node(val value: V) ```	```kotlin // Module two class Container(vararg val content: C) // Functions with generic class type that // also have a generic class type argument fun produce(): Container> = Container(Node(42)) fun consume(arg: Container>) {} ```

If you try to call these functions in module three, an error is triggered in Kotlin 2.0.0 because the generic class `Node<V>` is inaccessible from module three:

```
// Module three
fun test() {
    // Triggers an error in Kotlin 2.0.0, as generic class Node<V> is
    // inaccessible
    consume(produce())
}
```

In future releases we will continue to deprecate the use of inaccessible types in general. We have already started in Kotlin 2.0.0 by adding warnings for some scenarios with inaccessible types, including non-generic ones.

For example, let's use the same module setup as the previous examples, but turn the generic class `Node<V>` into a non-generic class `IntNode`, with all functions declared in module two:

Module one	Module two
```kotlin // Module one class IntNode(val value: Int) ```	```kotlin // Module two // A function that contains a lambda // parameter with `IntNode` type fun execute(func: (IntNode) -> Unit) {} class Container(vararg val content: C) // Functions with generic class type // that has `IntNode` as a type argument fun produce(): Container = Container(IntNode(42)) fun consume(arg: Container) {} ```

If you call these functions in module three, some warnings are triggered:

```
// Module three
fun test() {
    // Triggers warnings in Kotlin 2.0.0, as class IntNode is
    // inaccessible.

    execute {}
    // Class 'IntNode' of the parameter 'it' is inaccessible.

    execute { _ -> }
    execute(fun (_) {})
    // Class 'IntNode' of the parameter '_' is inaccessible.

    // Will trigger a warning in future Kotlin releases, as IntNode is
    // inaccessible.
    consume(produce())
}
```

What's the best practice now?

If you encounter new warnings regarding inaccessible generic types, it's highly likely that there's an issue with your build system configuration. We recommend checking your build scripts and configuration.

As a last resort, you can configure a direct dependency for module three on module one. Alternatively, you can modify your code to make the types accessible within the same module.

For more information, see the corresponding issue in YouTrack.

Consistent resolution order of Kotlin properties and Java fields with the same name

What's changed?

Before Kotlin 2.0.0, if you worked with Java and Kotlin classes that inherited from each other and contained Kotlin properties and Java fields with the same name, the resolution behavior of the duplicated name was inconsistent. There was also conflicting behavior between IntelliJ IDEA and the compiler. When developing the new resolution behavior for Kotlin 2.0.0, we aimed to cause the least impact to users.

For example, suppose there is a Java class `Base`:

```
public class Base {
    public String a = "a";

    public String b = "b";
}
```

Let's say there is also a Kotlin class `Derived` that inherits from the aforementioned `Base` class:

```
class Derived : Base() {
    val a = "aa"

    // Declares custom get() function
    val b get() = "bb"
}

fun main() {
    // Resolves Derived.a
    println(a)
    // aa

    // Resolves Base.b
    println(b)
    // b
}
```

Prior to Kotlin 2.0.0, `a` resolves to the Kotlin property within the `Derived` Kotlin class, whereas `b` resolves to the Java field in the `Base` Java class.

In Kotlin 2.0.0, the resolution behavior in the example is consistent, ensuring that the Kotlin property supersedes the Java field of the same name. Now, `b` resolves to: `Derived.b`.

> Prior to Kotlin 2.0.0, if you used IntelliJ IDEA to go to the declaration or usage of `a`, it would incorrectly navigate to the Java field when it should have navigated to the Kotlin property.
>
> From Kotlin 2.0.0, IntelliJ IDEA correctly navigates to the same location as the compiler.
>
> 

The general rule is that the subclass takes precedence. The previous example demonstrates this, as the Kotlin property `a` from the `Derived` class is resolved because `Derived` is a subclass of the `Base` Java class.

In the event that the inheritance is reversed and a Java class inherits from a Kotlin class, the Java field in the subclass takes precedence over the Kotlin property with the same name.

Consider this example:

Kotlin	Java
```kotlin open class Base { val a = "aa" } ```	```java public class Derived extends Base { public String a = "a"; } ```

Now in the following code:

```
fun main() {
    // Resolves Derived.a
    println(a)
    // a
}
```

What's the best practice now?

If this change affects your code, consider whether you really need to use duplicate names. If you want to have Java or Kotlin classes that each contain a field or property with the same name and that each inherit from one another, keep in mind that the field or property in the subclass will take precedence.

For more information, see the corresponding issue in YouTrack.

Improved null safety for Java primitive arrays

What's changed?

Starting with Kotlin 2.0.0, the compiler correctly infers the nullability of Java primitive arrays imported to Kotlin. Now, it retains native nullability from the `TYPE_USE` annotations used with Java primitive arrays and emits errors when their values are not used according to annotations.

Usually, when Java types with `@Nullable` and `@NotNull` annotations are called from Kotlin, they receive the appropriate native nullability:

```
interface DataService {
    @NotNull ResultContainer<@Nullable String> fetchData();
}
```

```
val dataService: DataService = ...
```

```
dataService.fetchData() // -> ResultContainer<String?>
```

Previously, however, when Java primitive arrays were imported to Kotlin, all `TYPE_USE` annotations were lost, resulting in platform nullability and possibly unsafe code:

```
interface DataProvider {
    int @Nullable [] fetchData();
}
```

```
val dataService: DataProvider = ...
dataService.fetchData() // -> IntArray .. IntArray?
// No error, even though `dataService.fetchData()` might be `null` according to annota
tions
// This might result in a NullPointerException
dataService.fetchData()[0]
```

Note that this issue never affected nullability annotations on the declaration itself, only the `TYPE_USE` ones.

What's the best practice now?

In Kotlin 2.0.0, null safety for Java primitive arrays is now standard in Kotlin, so check your code for new warnings and errors if you use them:

- Any code that uses a `@Nullable` Java primitive array without an explicit nullability check or attempts to pass `null` to a Java method expecting a non-nullable primitive array will now fail to compile.
- Using a `@NotNull` primitive array with a nullability check now emits "Unnecessary safe call" or "Comparison with null always false" warnings.

For more information, see the corresponding issue in YouTrack.

Stricter rules for abstract members in expected classes

> Expected and actual classes are in Beta. They are almost stable, but you may need to perform migration steps in the future. We'll do our best to minimize any further changes for you to make.
>
> 

What's changed?

Due to the separation of common and platform sources during compilation with the K2 compiler, we've implemented stricter rules for abstract members in expected classes.

With the previous compiler, it was possible for an expected non-abstract class to inherit an abstract function without overriding the function. Since the compiler could access both common and platform code at the same time, the compiler could see whether the abstract function had a corresponding override and definition in the actual class.

Now that common and platform sources are compiled separately, the inherited function must be explicitly overridden in the expected class so that the compiler knows the function is not abstract. Otherwise, the compiler reports an `ABSTRACT_MEMBER_NOT_IMPLEMENTED` error.

For example, let's say you have a common source set where you declare an abstract class called `FileSystem` that has an abstract function `listFiles()`. You define the `listFiles()` function in the platform source set as part of an actual declaration.

In your common code, if you have an expected non-abstract class called `PlatformFileSystem` that inherits from the `FileSystem` class, the `PlatformFileSystem` class inherits the abstract function `listFiles()`. However, you can't have an abstract function in a non-abstract class in Kotlin. To make the `listFiles()` function non-abstract, you must declare it as an override without the `abstract` keyword:

Common code	Platform code
```kotlin abstract class FileSystem { abstract fun listFiles() } expect open class PlatformFileSystem() : FileSystem { // In Kotlin 2.0.0, an explicit override is needed expect override fun listFiles() // Before Kotlin 2.0.0, an override wasn't needed } ```	```kotlin actual open class PlatformFileSystem : FileSystem { actual override fun listFiles() {} } ```

What's the best practice now?

If you inherit abstract functions in an expected non-abstract class, add a non-abstract override.

For more information, see the corresponding issue in YouTrack.

Per subject area

These subject areas list changes that are unlikely to affect your code but provide links to the relevant YouTrack issues for further reading. Changes listed with an asterisk (*) next to the Issue ID are explained at the beginning of the section.

Type inference

Issue ID	Title
KT-64189	Incorrect type in compiled function signature of property reference if the type is Normal explicitly

KT-47986	Forbid implicit inferring a type variable into an upper bound in the builder inference context
KT-59275	K2: Require explicit type arguments for generic annotation calls in array literals
KT-53752	Missed subtyping check for an intersection type
KT-59138	Change Java type parameter based types default representation in Kotlin
KT-57178	Change inferred type of prefix increment to return type of getter instead of return type of inc() operator
KT-57609	K2: Stop relying on the presence of @UnsafeVariance using for contravariant parameters
KT-57620	K2: Forbid resolution to subsumed members for raw types
KT-64641	K2: Properly inferred type of callable reference to a callable with extension-function parameter
KT-57011	Make real type of a destructuring variable consistent with explicit type when specified
KT-38895	K2: Fix inconsistent behavior with integer literals overflow
KT-54862	Anonymous type can be exposed from anonymous function from type argument
KT-22379	Condition of while-loop with break can produce unsound smartcast
KT-62507	K2: Prohibit smart cast in common code for expect/actual top-level property
KT-65750	Increment and plus operators that change return type must affect smart casts
KT-65349	[LC] K2: specifying variable types explicitly breaks bound smart casts in some cases that worked in K1

Generics

Issue ID	Title
KT-54309*	Deprecate use of a synthetic setter on a projected receiver
KT-57600	Forbid overriding of Java method with raw-typed parameter with generic typed parameter
KT-	

54663	Forbid passing possibly nullable type parameter to `in` projected DNN parameter
KT-54066	Deprecate upper bound violation in typealias constructors
KT-49404	Fix type unsoundness for contravariant captured type based on Java class
KT-61718	Forbid unsound code with self upper bounds and captured types
KT-61749	Forbid unsound bound violation in generic inner class of generic outer class
KT-62923	K2: Introduce PROJECTION_IN_IMMEDIATE_ARGUMENT_TO_SUPERTYPE for projections of outer super types of inner class
KT-63243	Report MANY_IMPL_MEMBER_NOT_IMPLEMENTED when inheriting from collection of primitives with an extra specialized implementation from another supertype
KT-60305	K2: Prohibit constructor call and inheritance on type alias that has variance modifiers in expanded type
KT-64965	Fix type hole caused by improper handling of captured types with self-upper bounds
KT-64966	Forbid generic delegating constructor calls with wrong type for generic parameter
KT-65712	Report missing upper bound violation when upper bound is captured type

Resolution

Issue ID	Title
KT-55017*	Choose Kotlin property from derived class during overload resolution with Java field from base class
KT-58260	Make invoke convention works consistently with expected desugaring
KT-62866	K2: Change qualifier resolution behavior when companion object is preferred against static scope
KT-57750	Report ambiguity error when resolving types and having the same-named classes star imported
KT-63558	K2: migrate resolution around COMPATIBILITY_WARNING
KT-51194	False negative CONFLICTING_INHERITED_MEMBERS when dependency class contained in two different versions of the same dependency

37592	invoke
KT-51666	Qualified this: introduce/prioritize this qualified with type case
KT-54166	Confirm unspecified behavior in case of FQ name conflicts in classpath
KT-64431	K2: forbid using typealiases as qualifier in imports
KT-56520	K1/K2: incorrect work of resolve tower for type references with ambiguity at lower level

Visibility

Issue ID	Title
KT-64474*	Declare usages of inaccessible types as unspecified behavior
KT-55179	False negative PRIVATE_CLASS_MEMBER_FROM_INLINE on calling private class companion object member from internal inline function
KT-58042	Make synthetic property invisible if equivalent getter is invisible even when overridden declaration is visible
KT-64255	Forbid accessing internal setter from a derived class in another module
KT-33917	Prohibit to expose anonymous types from private inline functions
KT-54997	Forbid implicit non-public-API accesses from public-API inline function
KT-56310	Smart casts should not affect visibility of protected members
KT-65494	Forbid access to overlooked private operator functions from public inline function
KT-65004	K1: Setter of var, which overrides protected val, is generates as public
KT-64972	Forbid overriding by private members in link-time for Kotlin/Native

Annotations

Issue ID	Title
KT-	

KT-58723	Forbid annotating statements with an annotation if it has no EXPRESSION target
KT-49930	Ignore parentheses expression during `REPEATED_ANNOTATION` checking
KT-57422	K2: Prohibit use-site 'get' targeted annotations on property getters
KT-46483	Prohibit annotation on type parameter in where clause
KT-64299	Companion scope is ignored for resolution of annotations on companion object
KT-64654	K2: Introduced ambiguity between user and compiler-required annotations
KT-64527	Annotations on enum values shouldn't be copied to enum value classes
KT-63389	K2: `WRONG_ANNOTATION_TARGET` is reported on incompatible annotations of a type wrapped into `()?`
KT-63388	K2: `WRONG_ANNOTATION_TARGET` is reported on catch parameter type's annotations

Null safety

Issue ID	Title
KT-54521*	Deprecate unsafe usages of array types annotated as Nullable in Java
KT-41034	K2: Change evaluation semantics for combination of safe calls and convention operators
KT-50850	Order of supertypes defines nullability parameters of inherited functions
KT-53982	Keep nullability when approximating local types in public signatures
KT-62998	Forbid assignment of a nullable to a not-null Java field as a selector of unsafe assignment
KT-63209	Report missing errors for error-level nullable arguments of warning-level Java types

Java interoperability

Issue ID	Title
KT-53061	Forbid Java and Kotlin classes with the same FQ name in sources

KT-49882	Classes inherited from Java collections have inconsistent behavior depending on order of supertypes
KT-66324	K2: unspecified behavior in case of Java class inheritance from a Kotlin private class
KT-66220	Passing java vararg method to inline function leads to array of arrays in runtime instead of just an array
KT-66204	Allow to override internal members in K-J-K hierarchy

Properties

Issue ID	Title
KT-57555*	[LC] Forbid deferred initialization of open properties with backing field
KT-58589	Deprecate missed MUST_BE_INITIALIZED when no primary constructor is presented or when class is local
KT-64295	Forbid recursive resolve in case of potential invoke calls on properties
KT-57290	Deprecate smart cast on base class property from invisible derived class if base class is from another module
KT-62661	K2: Missed OPT_IN_USAGE_ERROR for data class properties

Control flow

Issue ID	Title
KT-56408	Inconsistent rules of CFA in class initialization block between K1 and K2
KT-57871	K1/K2 inconsistency on if-conditional without else-branch in parenthesis
KT-42995	False negative "VAL_REASSIGNMENT" in try/catch block with initialization in scope function
KT-65724	Propagate data flow information from try block to catch and finally blocks

Enum classes

Issue	Title

KT-57608	Prohibit access to the companion object of enum class during initialization of enum entry
KT-34372	Report missed error for virtual inline method in enum classes
KT-52802	Report ambiguity resolving between property/field and enum entry
KT-47310	Change qualifier resolution behavior when companion property is preferred against enum entry

Functional (SAM) interfaces

Issue ID	Title
KT-52628	Deprecate SAM constructor usages which require OptIn without annotation
KT-57014	Prohibit returning values with incorrect nullability from lambda for SAM constructor of JDK function interfaces
KT-64342	SAM conversion of parameter types of callable references leads to CCE

Companion object

Issue ID	Title
KT-54316	Out-of-call reference to companion object's member has invalid signature
KT-47313	Change (V)::foo reference resolution when V has a companion

Miscellaneous

Issue ID	Title
KT-59739*	K2/MPP reports [ABSTRACT_MEMBER_NOT_IMPLEMENTED] for inheritor in common code when the implementation is located in the actual counterpart
KT-49015	Qualified this: change behavior in case of potential label conflicts
KT-56545	Fix incorrect functions mangling in JVM backend in case of accidental clashing overload in a Java subclass
KT-62019	[LC issue] Prohibit suspend-marked anonymous function declarations in statement positions
KT-	

KT-55111	OptIn: forbid constructor calls with default arguments under marker
KT-61182	Unit conversion is accidentally allowed to be used for expressions on variables + invoke resolution
KT-55199	Forbid promoting callable references with adaptations to KFunction
KT-65776	[LC] K2 breaks `false && ...` and `false || ...`
KT-65682	[LC] Deprecate `header`/`impl` keywords
KT-45375	Generate all Kotlin lambdas via invokedynamic + LambdaMetafactory by default

Compatibility with Kotlin releases

The following Kotlin releases have support for the new K2 compiler:

Kotlin release	Stability level
2.0.0–2.1.21	Stable
1.9.20–1.9.25	Beta
1.9.0–1.9.10	JVM is Beta
1.7.0–1.8.22	Alpha

Compatibility with Kotlin libraries

If you're working with Kotlin/JVM, the K2 compiler works with libraries compiled with any version of Kotlin.

If you're working with Kotlin Multiplatform, the K2 compiler is guaranteed to work with libraries compiled with Kotlin version 1.9.20 and onwards.

Compiler plugins support

Currently, the Kotlin K2 compiler supports the following Kotlin compiler plugins:

- `all-open`
- AtomicFU
- `jvm-abi-gen`

- `js-plain-objects`
- kapt
- Lombok
- `no-arg`
- Parcelize
- SAM with receiver
- Serialization

In addition, the Kotlin K2 compiler supports:

- The Jetpack Compose 1.5.0 compiler plugin and later versions.
- Kotlin Symbol Processing (KSP) since KSP2.

> If you use any additional compiler plugins, check their documentation to see if they are compatible with K2.
>
> 

Upgrade your custom compiler plugins

> Custom compiler plugins use the plugin API, which is Experimental. As a result, the API may change at any time, so we can't guarantee backward compatibility.
>
> 

The upgrade process has two paths depending on the type of custom plugin you have.

Backend-only compiler plugins

If your plugin implements only `IrGenerationExtension` extension points, the process is the same as for any other new compiler release. Check if there are any changes to the API that you use and make changes if necessary.

Backend and frontend compiler plugins

If your plugin uses frontend-related extension points, you need to rewrite the plugin using the new K2 compiler API. For an introduction to the new API, see FIR Plugin API.

> If you have questions about upgrading your custom compiler plugin, join our #compiler Slack channel, and we will do our best to help you.
>
> 

Share your feedback on the new K2 compiler

We would appreciate any feedback you may have!

- Report any problems you face migrating to the new K2 compiler in our issue tracker.
- Enable the Send usage statistics option to allow JetBrains to collect anonymous data about K2 usage.

Kotlin 命令行编译器

每个 Kotlin 版本都附带了一个独立版的编译器。 可以手动或者通过包管理器下载其最新版。

> 安装命令行编译器并不是使用 Kotlin 的必要步骤。 常见的做法是使用具有官方 Kotlin 支持的 IDE 或代码编辑器来编写 Kotlin 应用程序， 例如 IntelliJ IDEA 或者 Android Studio。 它们开箱即用地提供完整的 Kotlin 支持。
>
> Learn how to get started with Kotlin in an IDE.
>
> 

安装编译器

手动安装

To install the Kotlin compiler manually:

1. 在 GitHub Releases 页下载最新版本（ `kotlin-compiler-2.1.21.zip` ）。
2. 将独立编译器解压缩到一个目录中，并可选择将其 `bin` 目录添加到系统路径。 `bin` 目录包含了在 Windows、macOS 与 Linux 上编译及运行 Kotlin 所需的脚本。

> If you want to use the Kotlin command-line compiler on Windows, we recommend installing it manually.
>
> 

SDKMAN!

An easier way to install Kotlin on UNIX-based systems, such as macOS, Linux, Cygwin, FreeBSD, and Solaris, is SDKMAN!. It also works in Bash and ZSH shells. Learn how to install SDKMAN!.

To install the Kotlin compiler via SDKMAN!, run the following command in the terminal:

```
sdk install kotlin
```

Homebrew

Alternatively, on macOS you can install the compiler via Homebrew:

```
brew update
```

```
brew install kotlin
```

Snap 包

If you use Snap on Ubuntu 16.04 or later, you can install the compiler from the command line:

```
sudo snap install --classic kotlin
```

创建并运行应用程序

1. Create a simple console JVM application in Kotlin that displays `"Hello, World!"`. In a code editor, create a new file called `hello.kt` with the following code:

   ```
   fun main() {
       println("Hello, World!")
   }
   ```

2. Compile the application using the Kotlin compiler:

   ```
   kotlinc hello.kt -include-runtime -d hello.jar
   ```

   - The `-d` option indicates the output path for generated class files, which may be either a directory or a **.jar** file.
   - The `-include-runtime` option makes the resulting **.jar** file self-contained and runnable by including the Kotlin runtime library in it.

   To see all available options, run:

   ```
   kotlinc -help
   ```

3. Run the application:

   ```
   java -jar hello.jar
   ```

编译库

If you're developing a library to be used by other Kotlin applications, you can build the **.jar** file without including the Kotlin runtime:

```
kotlinc hello.kt -d hello.jar
```

Since binaries compiled this way depend on the Kotlin runtime, you should ensure that it is present in the classpath whenever your compiled library is used.

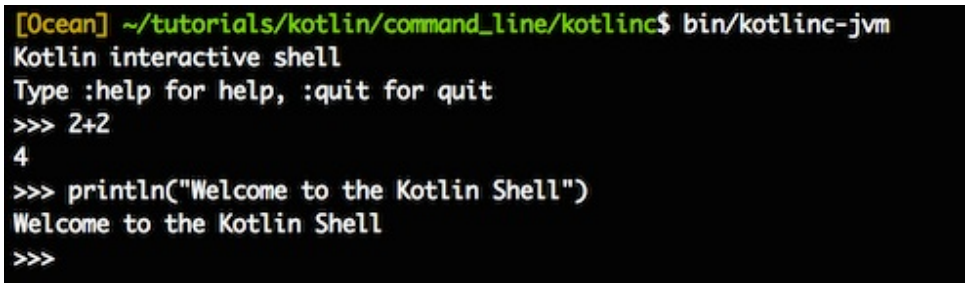
You can also use the `kotlin` script to run binaries produced by the Kotlin compiler:

```
kotlin -classpath hello.jar HelloKt
```

`HelloKt` is the main class name that the Kotlin compiler generates for the file named `hello.kt`.

运行 REPL

You can run the compiler without parameters to have an interactive shell. In this shell, you can type any valid Kotlin code and see the results.



运行脚本

You can use Kotlin as a scripting language. A Kotlin script is a Kotlin source file (`.kts`) with top-level executable code.

```
import java.io.File

// Get the passed in path, i.e. "-d some/path" or use the current path.
val path = if (args.contains("-d")) args[1 + args.indexOf("-d")]
           else "."

val folders = File(path).listFiles { file -> file.isDirectory() }
folders?.forEach { folder -> println(folder) }
```

To run a script, pass the `-script` option to the compiler with the corresponding script file:

```
kotlinc -script list_folders.kts -- -d <path_to_folder_to_inspect>
```

Kotlin provides experimental support for script customization, such as adding external properties, providing static or dynamic dependencies, and so on. Customizations are defined by so-called *script definitions* – annotated kotlin classes with the appropriate support code. The script filename extension is used to select the appropriate definition. Learn more about Kotlin custom scripting.

Properly prepared script definitions are detected and applied automatically when the appropriate jars are included in the compilation classpath. Alternatively, you can specify definitions manually by passing the `-script-templates` option to the compiler:

```
kotlinc -script-templates org.example.CustomScriptDefinition -script custom.script1.kt
s
```

For additional details, see the KEEP-75.

Kotlin 编译器选项

Each release of Kotlin includes compilers for the supported targets: JVM, JavaScript, and native binaries for supported platforms.

These compilers are used by:

- The IDE, when you click the **Compile** or **Run** button for your Kotlin project.
- Gradle, when you call `gradle build` in a console or in the IDE.
- Maven, when you call `mvn compile` or `mvn test-compile` in a console or in the IDE.

You can also run Kotlin compilers manually from the command line as described in the Working with command-line compiler tutorial.

编译器选项

Kotlin compilers have a number of options for tailoring the compiling process. Compiler options for different targets are listed on this page together with a description of each one.

There are several ways to set the compiler options and their values (*compiler arguments*):

- In IntelliJ IDEA, write in the compiler arguments in the **Additional command line parameters** text box in **Settings/Preferences** | **Build, Execution, Deployment** | **Compiler** | **Kotlin Compiler**.
- If you're using Gradle, specify the compiler arguments in the `compilerOptions` property of the Kotlin compilation task. For details, see Gradle compiler options.
- If you're using Maven, specify the compiler arguments in the `<configuration>` element of the Maven plugin node. For details, see Maven.
- If you run a command-line compiler, add the compiler arguments directly to the utility call or write them into an argfile.

  For example:

  ```
  $ kotlinc hello.kt -include-runtime -d hello.jar
  ```

  > On Windows, when you pass compiler arguments that contain delimiter characters (whitespace, `=`, `;`, `,`), surround these arguments with double quotes (`"`).
  >
  > ```
  > $ kotlinc.bat hello.kt -include-runtime -d "My Folder\hello.jar"
  > ```



Common options

The following options are common for all Kotlin compilers.

-version

Display the compiler version.

-nowarn

Suppress the compiler from displaying warnings during compilation.

-Werror

Turn any warnings into a compilation error.

-Wextra

Enable additional declaration, expression, and type compiler checks that emit warnings if true.

-verbose

Enable verbose logging output which includes details of the compilation process.

-script

Evaluate a Kotlin script file. When called with this option, the compiler executes the first Kotlin script (`*.kts`) file among the given arguments.

-help (-h)

Display usage information and exit. Only standard options are shown. To show advanced options, use `-X` .

-X

Display information about the advanced options and exit. These options are currently unstable: their names and behavior may be changed without notice.

-kotlin-home *path*

Specify a custom path to the Kotlin compiler used for the discovery of runtime libraries.

-P plugin:pluginId:optionName=value

Pass an option to a Kotlin compiler plugin. Core plugins and their options are listed in the Core compiler plugins section of the documentation.

-language-version *version*

Provide source compatibility with the specified version of Kotlin.

-api-version *version*

Allow using declarations only from the specified version of Kotlin bundled libraries.

-progressive

Enable the progressive mode for the compiler.

In the progressive mode, deprecations and bug fixes for unstable code take effect immediately, instead of going through a graceful migration cycle. Code written in the progressive mode is backwards compatible; however, code written in a non-progressive mode may cause compilation errors in the progressive mode.

@argfile

Read the compiler options from the given file. Such a file can contain compiler options with values and paths to the source files. Options and paths should be separated by whitespaces. For example:

```
-include-runtime -d hello.jar hello.kt
```

To pass values that contain whitespaces, surround them with single (') or double (") quotes. If a value contains quotation marks in it, escape them with a backslash (\).

```
-include-runtime -d 'My folder'
```

You can also pass multiple argument files, for example, to separate compiler options from source files.

```
$ kotlinc @compiler.options @classes
```

If the files reside in locations different from the current directory, use relative paths.

```
$ kotlinc @options/compiler.options hello.kt
```

-opt-in *annotation*

Enable usages of API that requires opt-in with a requirement annotation with the given fully qualified name.

-Xsuppress-warning

Suppresses specific warnings globally across the whole project, for example:

```
kotlinc -Xsuppress-warning=NOTHING_TO_INLINE -Xsuppress-warning=NO_TAIL_CALLS_FOUND ma
in.kt
```

Kotlin/JVM compiler options

The Kotlin compiler for JVM compiles Kotlin source files into Java class files. The command-line tools for Kotlin to JVM compilation are `kotlinc` and `kotlinc-jvm`. You can also use them for executing Kotlin script files.

In addition to the common options, Kotlin/JVM compiler has the options listed below.

-classpath *path* (-cp *path*)

Search for class files in the specified paths. Separate elements of the classpath with system path separators (**;** on Windows, **:** on macOS/Linux). The classpath can contain file and directory paths, ZIP, or JAR files.

-d *path*

Place the generated class files into the specified location. The location can be a directory, a ZIP, or a JAR file.

-include-runtime

Include the Kotlin runtime into the resulting JAR file. Makes the resulting archive runnable on any Java-enabled environment.

-jdk-home *path*

Use a custom JDK home directory to include into the classpath if it differs from the default `JAVA_HOME`.

-Xjdk-release=version

Specify the target version of the generated JVM bytecode. Limit the API of the JDK in the classpath to the specified Java version. Automatically sets `-jvm-target version`. Possible values are `1.8`, `9`, `10`, ..., `21`.

> This option is not guaranteed to be effective for each JDK distribution.
>
> 

-jvm-target *version*

Specify the target version of the generated JVM bytecode. Possible values are `1.8`, `9`, `10`, ..., `21`. The default value is `1.8`.

-java-parameters

Generate metadata for Java 1.8 reflection on method parameters.

-module-name *name* (JVM)

Set a custom name for the generated `.kotlin_module` file.

-no-jdk

Don't automatically include the Java runtime into the classpath.

-no-reflect

Don't automatically include the Kotlin reflection (`kotlin-reflect.jar`) into the classpath.

-no-stdlib (JVM)

Don't automatically include the Kotlin/JVM stdlib (`kotlin-stdlib.jar`) and Kotlin reflection (`kotlin-reflect.jar`) into the classpath.

-script-templates *classnames[,]*

Script definition template classes. Use fully qualified class names and separate them with commas (**,**).

Kotlin/JS compiler options

The Kotlin compiler for JS compiles Kotlin source files into JavaScript code. The command-line tool for Kotlin to JS compilation is `kotlinc-js` .

In addition to the common options, Kotlin/JS compiler has the options listed below.

-target {es5|es2015}

Generate JS files for the specified ECMA version.

-libraries *path*

Paths to Kotlin libraries with `.meta.js` and `.kjsm` files, separated by the system path separator.

-main *{call|noCall}*

Define whether the `main` function should be called upon execution.

-meta-info

Generate `.meta.js` and `.kjsm` files with metadata. Use this option when creating a JS library.

-module-kind {umd|commonjs|amd|plain}

The kind of JS module generated by the compiler:

- `umd` - a Universal Module Definition module
- `commonjs` - a CommonJS module
- `amd` - an Asynchronous Module Definition module
- `plain` - a plain JS module

To learn more about the different kinds of JS module and the distinctions between them, see this article.

-no-stdlib (JS)

Don't automatically include the default Kotlin/JS stdlib into the compilation dependencies.

-output *filepath*

Set the destination file for the compilation result. The value must be a path to a `.js` file including its name.

-output-postfix *filepath*

Add the content of the specified file to the end of the output file.

-output-prefix *filepath*

Add the content of the specified file to the beginning of the output file.

-source-map

Generate the source map.

-source-map-base-dirs *path*

Use the specified paths as base directories. Base directories are used for calculating relative paths in the source map.

-source-map-embed-sources *{always|never|inlining}*

Embed source files into the source map.

-source-map-names-policy *{simple-names|fully-qualified-names|no}*

Add variable and function names that you declared in Kotlin code into the source map.

Setting	Description	Example output
`simple-names`	Variable names and simple function names are added. (Default)	`main`
`fully-qualified-names`	Variable names and fully qualified function names are added.	`com.example.kjs.playground.main`
`no`	No variable or function names are added.	N/A

-source-map-prefix

Add the specified prefix to paths in the source map.

Kotlin/Native compiler options

Kotlin/Native compiler compiles Kotlin source files into native binaries for the supported platforms. The command-line tool for Kotlin/Native compilation is `kotlinc-native`.

In addition to the common options, Kotlin/Native compiler has the options listed below.

-enable-assertions (-ea)

Enable runtime assertions in the generated code.

-g

Enable emitting debug information. This option lowers the optimization level and should not be combined with the `-opt` option.

-generate-test-runner (-tr)

Produce an application for running unit tests from the project.

-generate-no-exit-test-runner (-trn)

Produce an application for running unit tests without an explicit process exit.

-include-binary *path* (-ib *path*)

Pack external binary within the generated klib file.

-library *path* (-l *path*)

Link with the library. To learn about using libraries in Kotlin/native projects, see Kotlin/Native libraries.

-library-version *version* (-lv *version*)

Set the library version.

-list-targets

List the available hardware targets.

-manifest *path*

Provide a manifest addend file.

-module-name *name* (Native)

Specify a name for the compilation module. This option can also be used to specify a name prefix for the declarations exported to Objective-C: How do I specify a custom Objective-C prefix/name for my Kotlin framework?

-native-library *path* (-nl *path*)

Include the native bitcode library.

-no-default-libs

Disable linking user code with the prebuilt platform libraries distributed with the compiler.

-nomain

Assume the `main` entry point to be provided by external libraries.

-nopack

Don't pack the library into a klib file.

-linker-option

Pass an argument to the linker during binary building. This can be used for linking against some native library.

-linker-options *args*

Pass multiple arguments to the linker during binary building. Separate arguments with whitespaces.

-nostdlib

Don't link with stdlib.

-opt

Enable compilation optimizations and produce a binary with better runtime performance. It's not recommended to combine it with the `-g` option, which lowers the optimization level.

-output *name* (-o *name*)

Set the name for the output file.

-entry *name* (-e *name*)

Specify the qualified entry point name.

-produce *output* (-p *output*)

Specify output file kind:

- `program`
- `static`
- `dynamic`
- `framework`
- `library`
- `bitcode`

-repo *path* (-r *path*)

Library search path. For more information, see Library search sequence.

-target *target*

Set hardware target. To see the list of available targets, use the `-list-targets` option.

编译器插件

- 全开放编译器插件
- No-arg 编译器插件
- 带有接收者的 SAM 编译器插件
- kapt 编译器插件
- Lombok 编译器插件
- Power-assert

全开放编译器插件

Kotlin has classes and their members `final` by default, which makes it inconvenient to use frameworks and libraries such as Spring AOP that require classes to be `open`. The `all-open` compiler plugin adapts Kotlin to the requirements of those frameworks and makes classes annotated with a specific annotation and their members open without the explicit `open` keyword.

For instance, when you use Spring, you don't need all the classes to be open, but only classes annotated with specific annotations like `@Configuration` or `@Service`. The `all-open` plugin allows you to specify such annotations.

Kotlin provides `all-open` plugin support both for Gradle and Maven with the complete IDE integration.

> For Spring, you can use the `kotlin-spring` compiler plugin.
>
> 

Gradle

Add the plugin in your `build.gradle(.kts)` file:

> 【Kotlin】

```
plugins {
    kotlin("plugin.allopen") version "2.1.21"
}
```

> 【Groovy】

```
plugins {
    id "org.jetbrains.kotlin.plugin.allopen" version "2.1.21"
}
```

Then specify the list of annotations that will make classes open:

> 【Kotlin】

```
allOpen {
    annotation("com.my.Annotation")
    // annotations("com.another.Annotation", "com.third.Annotation")
}
```

> 【Groovy】

```
allOpen {
    annotation("com.my.Annotation")
    // annotations("com.another.Annotation", "com.third.Annotation")
}
```

If the class (or any of its superclasses) is annotated with `com.my.Annotation`, the class itself and all its members will become open.

It also works with meta-annotations:

```
@com.my.Annotation
annotation class MyFrameworkAnnotation

@MyFrameworkAnnotation
class MyClass // will be all-open
```

`MyFrameworkAnnotation` is annotated with the all-open meta-annotation `com.my.Annotation`, so it becomes an all-open annotation as well.

Maven

Add the plugin in your `pom.xml` file:

```
<plugin>
    <artifactId>kotlin-maven-plugin</artifactId>
    <groupId>org.jetbrains.kotlin</groupId>
    <version>${kotlin.version}</version>

    <configuration>
        <compilerPlugins>
            <!-- Or "spring" for the Spring support -->
            <plugin>all-open</plugin>
```

```
        </compilerPlugins>

        <pluginOptions>
            <!-- Each annotation is placed on its own line -->
            <option>all-open:annotation=com.my.Annotation</option>
            <option>all-open:annotation=com.their.AnotherAnnotation</option>
        </pluginOptions>
    </configuration>

    <dependencies>
        <dependency>
            <groupId>org.jetbrains.kotlin</groupId>
            <artifactId>kotlin-maven-allopen</artifactId>
            <version>${kotlin.version}</version>
        </dependency>
    </dependencies>
</plugin>
```

Please refer to the Gradle section for the detailed information about how all-open annotations work.

Spring support

If you use Spring, you can enable the `kotlin-spring` compiler plugin instead of specifying Spring annotations manually. The `kotlin-spring` is a wrapper on top of `all-open`, and it behaves exactly the same way.

Add the `spring` plugin in your `build.gradle(.kts)` file:

> 【Kotlin】

```
plugins {
    id("org.jetbrains.kotlin.plugin.spring") version "2.1.21"
}
```

> 【Groovy】

```
plugins {
    id "org.jetbrains.kotlin.plugin.spring" version "2.1.21"
}
```

In Maven, the `spring` plugin is provided by the `kotlin-maven-allopen` plugin dependency, so to enable it in your `pom.xml` file:

```
<compilerPlugins>
    <plugin>spring</plugin>
</compilerPlugins>

<dependencies>
    <dependency>
        <groupId>org.jetbrains.kotlin</groupId>
        <artifactId>kotlin-maven-allopen</artifactId>
        <version>${kotlin.version}</version>
    </dependency>
</dependencies>
```

The plugin specifies the following annotations:

- `@Component`
- `@Async`
- `@Transactional`
- `@Cacheable`
- `@SpringBootTest`

Thanks to meta-annotations support, classes annotated with `@Configuration`, `@Controller`, `@RestController`, `@Service` or `@Repository` are automatically opened since these annotations are meta-annotated with `@Component`.

Of course, you can use both `kotlin-allopen` and `kotlin-spring` in the same project.

> If you generate the project template by the start.spring.io service, the `kotlin-spring` plugin will be enabled by default.
>
> 

Command-line compiler

All-open compiler plugin JAR is available in the binary distribution of the Kotlin compiler. You can attach the plugin by providing the path to its JAR file using the `-Xplugin` kotlinc option:

```
-Xplugin=$KOTLIN_HOME/lib/allopen-compiler-plugin.jar
```

You can specify all-open annotations directly, using the `annotation` plugin option, or enable the *preset*:

```
# The plugin option format is: "-P plugin:<plugin id>:<key>=<value>".
# Options can be repeated.

-P plugin:org.jetbrains.kotlin.allopen:annotation=com.my.Annotation
-P plugin:org.jetbrains.kotlin.allopen:preset=spring
```

Presets that available for the `all-open` plugin are: `spring`, `micronaut`, and `quarkus`.

No-arg 编译器插件

The *no-arg* compiler plugin generates an additional zero-argument constructor for classes with a specific annotation.

The generated constructor is synthetic, so it can't be directly called from Java or Kotlin, but it can be called using reflection.

This allows the Java Persistence API (JPA) to instantiate a class although it doesn't have the zero-parameter constructor from Kotlin or Java point of view (see the description of `kotlin-jpa` plugin below).

In your Kotlin file

Add new annotations to mark the code that needs a zero-argument constructor:

```
package com.my

annotation class Annotation
```

Gradle

Add the plugin using Gradle's plugins DSL:

> 【Kotlin】

```
plugins {
    kotlin("plugin.noarg") version "2.1.21"
}
```

> 【Groovy】

```
plugins {
    id "org.jetbrains.kotlin.plugin.noarg" version "2.1.21"
}
```

Then specify the list of no-arg annotations that must lead to generating a no-arg constructor for the annotated classes:

```
noArg {
    annotation("com.my.Annotation")
}
```

Enable `invokeInitializers` option if you want the plugin to run the initialization logic from the synthetic constructor. By default, it is disabled.

```
noArg {
    invokeInitializers = true
}
```

Maven

```
<plugin>
    <artifactId>kotlin-maven-plugin</artifactId>
    <groupId>org.jetbrains.kotlin</groupId>
    <version>${kotlin.version}</version>

    <configuration>
        <compilerPlugins>
            <!-- Or "jpa" for JPA support -->
            <plugin>no-arg</plugin>
        </compilerPlugins>

        <pluginOptions>
            <option>no-arg:annotation=com.my.Annotation</option>
            <!-- Call instance initializers in the synthetic constructor -->
            <!-- <option>no-arg:invokeInitializers=true</option> -->
        </pluginOptions>
    </configuration>

    <dependencies>
        <dependency>
            <groupId>org.jetbrains.kotlin</groupId>
            <artifactId>kotlin-maven-noarg</artifactId>
            <version>${kotlin.version}</version>
        </dependency>
    </dependencies>
</plugin>
```

JPA 支持

As with the `kotlin-spring` plugin wrapped on top of `all-open`, `kotlin-jpa` is wrapped on top of `no-arg`. The plugin specifies `@Entity`, `@Embeddable`, and `@MappedSuperclass` *no-arg* annotations automatically.

Add the plugin using the Gradle plugins DSL:

> 【Kotlin】

```
plugins {
    kotlin("plugin.jpa") version "2.1.21"
}
```

> 【Groovy】

```
plugins {
    id "org.jetbrains.kotlin.plugin.jpa" version "2.1.21"
}
```

In Maven, enable the `jpa` plugin:

```
<compilerPlugins>
    <plugin>jpa</plugin>
</compilerPlugins>
```

命令行编译器

Add the plugin JAR file to the compiler plugin classpath and specify annotations or presets:

```
-Xplugin=$KOTLIN_HOME/lib/noarg-compiler-plugin.jar
-P plugin:org.jetbrains.kotlin.noarg:annotation=com.my.Annotation
-P plugin:org.jetbrains.kotlin.noarg:preset=jpa
```

带有接收者的 SAM 编译器插件

The *sam-with-receiver* compiler plugin makes the first parameter of the annotated Java "single abstract method" (SAM) interface method a receiver in Kotlin. This conversion only works when the SAM interface is passed as a Kotlin lambda, both for SAM adapters and SAM constructors (see the SAM conversions documentation for more details).

Here is an example:

```
public @interface SamWithReceiver {}

@SamWithReceiver
public interface TaskRunner {
    void run(Task task);
}
```

```
fun test(context: TaskContext) {
    val runner = TaskRunner {
        // Here 'this' is an instance of 'Task'

        println("$name is started")
        context.executeTask(this)
        println("$name is finished")
    }
}
```

Gradle

The usage is the same to all-open and no-arg, except the fact that sam-with-receiver does not have any built-in presets, and you need to specify your own list of special-treated annotations.

> 【Kotlin】

```
plugins {
    kotlin("plugin.sam.with.receiver") version "2.1.21"
}
```

> 【Groovy】

```
plugins {
    id "org.jetbrains.kotlin.plugin.sam.with.receiver" version "2.1.21"
}
```

Then specify the list of SAM-with-receiver annotations:

```
samWithReceiver {
    annotation("com.my.SamWithReceiver")
}
```

Maven

```
<plugin>
    <artifactId>kotlin-maven-plugin</artifactId>
    <groupId>org.jetbrains.kotlin</groupId>
    <version>${kotlin.version}</version>

    <configuration>
        <compilerPlugins>
            <plugin>sam-with-receiver</plugin>
        </compilerPlugins>

        <pluginOptions>
            <option>
                sam-with-receiver:annotation=com.my.SamWithReceiver
            </option>
        </pluginOptions>
    </configuration>

    <dependencies>
        <dependency>
            <groupId>org.jetbrains.kotlin</groupId>
            <artifactId>kotlin-maven-sam-with-receiver</artifactId>
            <version>${kotlin.version}</version>
        </dependency>
    </dependencies>
</plugin>
```

Command-line compiler

Add the plugin JAR file to the compiler plugin classpath and specify the list of sam-with-receiver annotations:

```
-Xplugin=$KOTLIN_HOME/lib/sam-with-receiver-compiler-plugin.jar
-P plugin:org.jetbrains.kotlin.samWithReceiver:annotation=com.my.SamWithReceiver
```

kapt 编译器插件

> kapt is in maintenance mode. We are keeping it up-to-date with recent Kotlin and Java releases but have no plans to implement new features. Please use the Kotlin Symbol Processing API (KSP) for annotation processing. See the list of libraries supported by KSP.
>
> 

Annotation processors (see JSR 269) are supported in Kotlin with the *kapt* compiler plugin.

In a nutshell, you can use libraries such as Dagger or Data Binding in your Kotlin projects.

Please read below about how to apply the *kapt* plugin to your Gradle/Maven build.

Use in Gradle

Follow these steps:

1. Apply the `kotlin-kapt` Gradle plugin:

> 【Kotlin】

```
plugins {
    kotlin("kapt") version "2.1.21"
}
```

> 【Groovy】

```
plugins {
    id "org.jetbrains.kotlin.kapt" version "2.1.21"
}
```

---

1. Add the respective dependencies using the `kapt` configuration in your `dependencies` block:

> 【Kotlin】

```
dependencies {
    kapt("groupId:artifactId:version")
}
```

【Groovy】

```
dependencies {
    kapt 'groupId:artifactId:version'
}
```

1. If you previously used the Android support for annotation processors, replace usages of the `annotationProcessor` configuration with `kapt`. If your project contains Java classes, `kapt` will also take care of them.

   If you use annotation processors for your `androidTest` or `test` sources, the respective `kapt` configurations are named `kaptAndroidTest` and `kaptTest`. Note that `kaptAndroidTest` and `kaptTest` extends `kapt`, so you can just provide the `kapt` dependency and it will be available both for production sources and tests.

Try Kotlin K2 compiler

> Support for K2 in the kapt compiler plugin is Experimental. Opt-in is required (see details below), and you should use it only for evaluation purposes.
>
> 

From Kotlin 1.9.20, you can try using the kapt compiler plugin with the K2 compiler, which brings performance improvements and many other benefits. To use the K2 compiler in your Gradle project, add the following option to your `gradle.properties` file:

```
kapt.use.k2=true
```

If you use the Maven build system, update your `pom.xml` file:

```
<configuration>
   ...
   <args>
      <arg>-Xuse-k2-kapt</arg>
   </args>
</configuration>
```

> To enable the kapt plugin in your Maven project, see .
>
> 

If you encounter any issues when using kapt with the K2 compiler, please report them to our issue tracker.

Annotation processor arguments

Use `arguments {}` block to pass arguments to annotation processors:

```
kapt {
    arguments {
        arg("key", "value")
    }
}
```

Gradle 构建缓存支持

The kapt annotation processing tasks are cached in Gradle by default. However, annotation processors run arbitrary code that may not necessarily transform the task inputs into the outputs, might access and modify the files that are not tracked by Gradle etc. If the annotation processors used in the build cannot be properly cached, it is possible to disable caching for kapt entirely by adding the following lines to the build script, in order to avoid false-positive cache hits for the kapt tasks:

```
kapt {
    useBuildCache = false
}
```

Improve the speed of builds that use kapt

Run kapt tasks in parallel

To improve the speed of builds that use kapt, you can enable the Gradle Worker API for kapt tasks. Using the Worker API lets Gradle run independent annotation processing tasks from a single project in parallel, which in some cases significantly decreases the execution time.

When you use the custom JDK home feature in the Kotlin Gradle plugin, kapt task workers use only process isolation mode. Note that the `kapt.workers.isolation` property is ignored.

If you want to provide additional JVM arguments for a kapt worker process, use the input `kaptProcessJvmArgs` of the `KaptWithoutKotlincTask`:

> 【Kotlin】

```
tasks.withType<org.jetbrains.kotlin.gradle.internal.KaptWithoutKotlincTask>()
    .configureEach {
        kaptProcessJvmArgs.add("-Xmx512m")
    }
```

> 【Groovy】

```
tasks.withType(org.jetbrains.kotlin.gradle.internal.KaptWithoutKotlincTask.class)
    .configureEach {
        kaptProcessJvmArgs.add('-Xmx512m')
    }
```

Caching for annotation processors' classloaders

> 在 kapt 中缓存注解处理器的类加载器 is Experimental. It may be dropped or changed at any time. Use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack.
>
> 

Caching for annotation processors' classloaders helps kapt perform faster if you run many Gradle tasks consecutively.

To enable this feature, use the following properties in your `gradle.properties` file:

```
# positive value will enable caching
# use the same value as the number of modules that use kapt
kapt.classloaders.cache.size=5

# disable for caching to work
kapt.include.compile.classpath=false
```

If you run into any problems with caching for annotation processors, disable caching for them:

```
# specify annotation processors' full names to disable caching for them
kapt.classloaders.cache.disableForProcessors=[annotation processors full names]
```

Measure performance of annotation processors

Get a performance statistics on the annotation processors execution using the `-Kapt-show-processor-timings` plugin option. An example output:

```
Kapt Annotation Processing performance report:
com.example.processor.TestingProcessor: total: 133 ms, init: 36 ms, 2 round(s): 97 ms,
 0 ms
com.example.processor.AnotherProcessor: total: 100 ms, init: 6 ms, 1 round(s): 93 ms
```

You can dump this report into a file with the plugin option `-Kapt-dump-processor-timings` (`org.jetbrains.kotlin.kapt3:dumpProcessorTimings`). The following command will run kapt and dump the statistics to the `ap-perf-report.file` file:

```
kotlinc -cp $MY_CLASSPATH \
-Xplugin=kotlin-annotation-processing-SNAPSHOT.jar -P \
plugin:org.jetbrains.kotlin.kapt3:aptMode=stubsAndApt,\
plugin:org.jetbrains.kotlin.kapt3:apclasspath=processor/build/libs/processor.jar,\
plugin:org.jetbrains.kotlin.kapt3:dumpProcessorTimings=ap-perf-report.file \
-Xplugin=$JAVA_HOME/lib/tools.jar \
-d cli-tests/out \
-no-jdk -no-reflect -no-stdlib -verbose \
sample/src/main/
```

Measure the number of files generated with annotation processors

The `kotlin-kapt` Gradle plugin can report statistics on the number of generated files for each annotation processor.

This is useful to track if there are unused annotation processors as a part of the build. You can use the generated report to find modules that trigger unnecessary annotation processors and update the modules to prevent that.

Enable the statistics in two steps:

- Set the `showProcessorStats` flag to `true` in your `build.gradle(.kts)` :

  ```
  kapt {
      showProcessorStats = true
  }
  ```

- Set the `kapt.verbose` Gradle property to `true` in your `gradle.properties` :

  ```
  kapt.verbose=true
  ```

> You can also enable verbose output via the command line option `verbose`.
>
> 

The statistics will appear in the logs with the `info` level. You'll see the `Annotation processor stats:` line followed by statistics on the execution time of each annotation processor. After these lines, there will be the `Generated files report:` line followed by statistics on the number of generated files for each annotation processor. For example:

```
[INFO] Annotation processor stats:
[INFO] org.mapstruct.ap.MappingProcessor: total: 290 ms, init: 1 ms, 3 round(s): 289 m
s, 0 ms, 0 ms
[INFO] Generated files report:
[INFO] org.mapstruct.ap.MappingProcessor: total sources: 2, sources per round: 2, 0, 0
```

Compile avoidance for kapt

To improve the times of incremental builds with kapt, it can use the Gradle compile avoidance. With compile avoidance enabled, Gradle can skip annotation processing when rebuilding a project. Particularly, annotation processing is skipped when:

- The project's source files are unchanged.
- The changes in dependencies are ABI compatible. For example, the only changes are in method bodies.

However, compile avoidance can't be used for annotation processors discovered in the compile classpath since *any changes* in them require running the annotation processing tasks.

To run kapt with compile avoidance:

- Add the annotation processor dependencies to the `kapt*` configurations manually as described above.
- Turn off the discovery of annotation processors in the compile classpath by adding this line to your `gradle.properties` file:

```
kapt.include.compile.classpath=false
```

Incremental annotation processing

kapt supports incremental annotation processing that is enabled by default. Currently, annotation processing can be incremental only if all annotation processors being used are incremental.

To disable incremental annotation processing, add this line to your `gradle.properties` file:

```
kapt.incremental.apt=false
```

Note that incremental annotation processing requires incremental compilation to be enabled as well.

Inherit annotation processors from superconfigurations

You can define a common set of annotation processors in a separate Gradle configuration as a superconfiguration and extend it further in kapt-specific configurations for your subprojects.

As an example, for a subproject using Dagger, in your `build.gradle(.kts)` file, use the following configuration:

```
val commonAnnotationProcessors by configurations.creating
configurations.named("kapt") { extendsFrom(commonAnnotationProcessors) }

dependencies {
    implementation("com.google.dagger:dagger:2.48.1")
    commonAnnotationProcessors("com.google.dagger:dagger-compiler:2.48.1")
}
```

In this example, the `commonAnnotationProcessors` Gradle configuration is your common superconfiguration for annotation processing that you want to be used for all your projects. You use the `extendsFrom()` method to add `commonAnnotationProcessors` as a superconfiguration. kapt sees that the `commonAnnotationProcessors` Gradle configuration has a dependency on the Dagger annotation processor. Therefore, kapt includes the Dagger annotation processor in its configuration for annotation processing.

Java compiler options

kapt uses Java compiler to run annotation processors.

Here is how you can pass arbitrary options to javac:

```
kapt {
    javacOptions {
        // Increase the max count of errors from annotation processors.
```

```
        // Default is 100.
        option("-Xmaxerrs", 500)
    }
}
```

Non-existent type correction

Some annotation processors (such as `AutoFactory`) rely on precise types in declaration signatures. By default, kapt replaces every unknown type (including types for the generated classes) to `NonExistentClass`, but you can change this behavior. Add the option to the `build.gradle(.kts)` file to enable error type inferring in stubs:

```
kapt {
    correctErrorTypes = true
}
```

Use in Maven

Add an execution of the `kapt` goal from kotlin-maven-plugin before `compile`:

```
<execution>
    <id>kapt</id>
    <goals>
        <goal>kapt</goal> <!-- You can skip the <goals> element
        if you enable extensions for the plugin -->
    </goals>
    <configuration>
        <sourceDirs>
            <sourceDir>src/main/kotlin</sourceDir>
            <sourceDir>src/main/java</sourceDir>
        </sourceDirs>
        <annotationProcessorPaths>
            <!-- Specify your annotation processors here -->
            <annotationProcessorPath>
                <groupId>com.google.dagger</groupId>
                <artifactId>dagger-compiler</artifactId>
                <version>2.9</version>
            </annotationProcessorPath>
        </annotationProcessorPaths>
    </configuration>
</execution>
```

To configure the level of annotation processing, set one of the following as the `aptMode` in the `<configuration>` block:

- `stubs` – only generate stubs needed for annotation processing.
- `apt` – only run annotation processing.
- `stubsAndApt` – (default) generate stubs and run annotation processing.

For example:

```
<configuration>
   ...
   <aptMode>stubs</aptMode>
</configuration>
```

To enable the kapt plugin with the K2 compiler, add the `-Xuse-k2-kapt` compiler option:

```
<configuration>
   ...
   <args>
      <arg>-Xuse-k2-kapt</arg>
   </args>
</configuration>
```

Use in IntelliJ build system

kapt is not supported for IntelliJ IDEA's own build system. Launch the build from the "Maven Projects" toolbar whenever you want to re-run the annotation processing.

Use in CLI

kapt compiler plugin is available in the binary distribution of the Kotlin compiler.

You can attach the plugin by providing the path to its JAR file using the `Xplugin` kotlinc option:

```
-Xplugin=$KOTLIN_HOME/lib/kotlin-annotation-processing.jar
```

Here is a list of the available options:

- `sources` (*required*): An output path for the generated files.
- `classes` (*required*): An output path for the generated class files and resources.
- `stubs` (*required*): An output path for the stub files. In other words, some temporary directory.
- `incrementalData`: An output path for the binary stubs.
- `apclasspath` (*repeatable*): A path to the annotation processor JAR. Pass as many `apclasspath` options as the number of JARs that you have.

- `apoptions` : A base64-encoded list of the annotation processor options. See AP/javac options encoding for more information.
- `javacArguments` : A base64-encoded list of the options passed to javac. See AP/javac options encoding for more information.
- `processors` : A comma-specified list of annotation processor qualified class names. If specified, kapt does not try to find annotation processors in `apclasspath` .
- `verbose` : Enable verbose output.
- `aptMode` (*required*)
  - `stubs` – only generate stubs needed for annotation processing.
  - `apt` – only run annotation processing.
  - `stubsAndApt` – generate stubs and run annotation processing.
- `correctErrorTypes` : For more information, see Non-existent type correction. Disabled by default.
- `dumpFileReadHistory` : An output path to dump for each file a list of classes used during annotation processing.

The plugin option format is: `-P plugin:<plugin id>:<key>=<value>` . Options can be repeated.

An example:

```
-P plugin:org.jetbrains.kotlin.kapt3:sources=build/kapt/sources
-P plugin:org.jetbrains.kotlin.kapt3:classes=build/kapt/classes
-P plugin:org.jetbrains.kotlin.kapt3:stubs=build/kapt/stubs

-P plugin:org.jetbrains.kotlin.kapt3:apclasspath=lib/ap.jar
-P plugin:org.jetbrains.kotlin.kapt3:apclasspath=lib/anotherAp.jar

-P plugin:org.jetbrains.kotlin.kapt3:correctErrorTypes=true
```

Generate Kotlin sources

kapt can generate Kotlin sources. Just write the generated Kotlin source files to the directory specified by `processingEnv.options["kapt.kotlin.generated"]` , and these files will be compiled together with the main sources.

Note that kapt does not support multiple rounds for the generated Kotlin files.

AP/Javac options encoding

`apoptions` and `javacArguments` CLI options accept an encoded map of options.

Here is how you can encode options by yourself:

```
fun encodeList(options: Map<String, String>): String {
    val os = ByteArrayOutputStream()
    val oos = ObjectOutputStream(os)

    oos.writeInt(options.size)
    for ((key, value) in options.entries) {
        oos.writeUTF(key)
        oos.writeUTF(value)
    }

    oos.flush()
    return Base64.getEncoder().encodeToString(os.toByteArray())
}
```

Keep Java compiler's annotation processors

By default, kapt runs all annotation processors and disables annotation processing by javac. However, you may need some of javac's annotation processors working (for example, Lombok).

In the Gradle build file, use the option `keepJavacAnnotationProcessors` :

```
kapt {
    keepJavacAnnotationProcessors = true
}
```

If you use Maven, you need to specify concrete plugin settings. See this example of settings for the Lombok compiler plugin.

Lombok 编译器插件

> The Lombok compiler plugin is Experimental. It may be dropped or changed at any time. Use it only for evaluation purposes. We would appreciate your feedback on it in YouTrack.
>
> 

The Kotlin Lombok compiler plugin allows the generation and use of Java's Lombok declarations by Kotlin code in the same mixed Java/Kotlin module. If you call such declarations from another module, then you don't need to use this plugin for the compilation of that module.

The Lombok compiler plugin cannot replace Lombok, but it helps Lombok work in mixed Java/Kotlin modules. Thus, you still need to configure Lombok as usual when using this plugin. Learn more about how to configure the Lombok compiler plugin.

Supported annotations

The plugin supports the following annotations:

- `@Getter`, `@Setter`
- `@Builder`, `@SuperBuilder`
- `@NoArgsConstructor`, `@RequiredArgsConstructor`, and `@AllArgsConstructor`
- `@Data`
- `@With`
- `@Value`

We're continuing to work on this plugin. To find out the detailed current state, visit the Lombok compiler plugin's README.

Currently, we don't have plans to support the `@Tolerate` annotation. However, we can consider this if you vote for the @Tolerate issue in YouTrack.

> Kotlin compiler ignores Lombok annotations if you use them in Kotlin code.
>
> 

Gradle

Apply the `kotlin-plugin-lombok` Gradle plugin in the `build.gradle(.kts)` file:

> 【Kotlin】

```
plugins {
    kotlin("plugin.lombok") version "2.1.21"
    id("io.freefair.lombok") version "8.13"
}
```

> 【Groovy】

```
plugins {
    id 'org.jetbrains.kotlin.plugin.lombok' version '2.1.21'
    id 'io.freefair.lombok' version '8.13'
}
```

See this test project with examples of the Lombok compiler plugin in use.

Using the Lombok configuration file

If you use a Lombok configuration file `lombok.config`, you need to set the file's path so that the plugin can find it. The path must be relative to the module's directory. For example, add the following code to your `build.gradle(.kts)` file:

> 【Kotlin】

```
kotlinLombok {
    lombokConfigurationFile(file("lombok.config"))
}
```

> 【Groovy】

```
kotlinLombok {
    lombokConfigurationFile file("lombok.config")
}
```

See this test project with examples of the Lombok compiler plugin and `lombok.config` in use.

Maven

To use the Lombok compiler plugin, add the plugin `lombok` to the `compilerPlugins` section and the dependency `kotlin-maven-lombok` to the `dependencies` section. If you use a Lombok configuration file `lombok.config`, provide a path to it to the plugin in the `pluginOptions`. Add the following lines to the `pom.xml` file:

```
<plugin>
    <groupId>org.jetbrains.kotlin</groupId>
    <artifactId>kotlin-maven-plugin</artifactId>
    <version>${kotlin.version}</version>
    <configuration>
        <compilerPlugins>
            <plugin>lombok</plugin>
        </compilerPlugins>
        <pluginOptions>
            <option>lombok:config=${project.basedir}/lombok.config</option>
        </pluginOptions>
    </configuration>
    <dependencies>
        <dependency>
            <groupId>org.jetbrains.kotlin</groupId>
            <artifactId>kotlin-maven-lombok</artifactId>
            <version>${kotlin.version}</version>
        </dependency>
        <dependency>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
            <version>1.18.20</version>
            <scope>provided</scope>
        </dependency>
    </dependencies>
</plugin>
```

See this test project example of the Lombok compiler plugin and `lombok.config` in use.

Using with kapt

By default, the kapt compiler plugin runs all annotation processors and disables annotation processing by javac. To run Lombok along with kapt, set up kapt to keep javac's annotation processors working.

If you use Gradle, add the option to the `build.gradle(.kts)` file:

```
kapt {
    keepJavacAnnotationProcessors = true
}
```

In Maven, use the following settings to launch Lombok with Java's compiler:

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>3.5.1</version>
    <configuration>
        <source>1.8</source>
        <target>1.8</target>
        <annotationProcessorPaths>
            <annotationProcessorPath>
                <groupId>org.projectlombok</groupId>
                <artifactId>lombok</artifactId>
                <version>${lombok.version}</version>
            </annotationProcessorPath>
        </annotationProcessorPaths>
    </configuration>
</plugin>
```

The Lombok compiler plugin works correctly with kapt if annotation processors don't depend on the code generated by Lombok.

Look through the test project examples of kapt and the Lombok compiler plugin in use:

- Using Gradle.
- Using Maven

Command-line compiler

Lombok compiler plugin JAR is available in the binary distribution of the Kotlin compiler. You can attach the plugin by providing the path to its JAR file using the `Xplugin` kotlinc option:

```
-Xplugin=$KOTLIN_HOME/lib/lombok-compiler-plugin.jar
```

If you want to use the `lombok.config` file, replace `<PATH_TO_CONFIG_FILE>` with a path to your `lombok.config`:

```
# The plugin option format is: "-P plugin:<plugin id>:<key>=<value>".
# Options can be repeated.

-P plugin:org.jetbrains.kotlin.lombok:config=<PATH_TO_CONFIG_FILE>
```

Power-assert 编译器插件

> The Power-assert compiler plugin is Experimental. It may be changed at any time. Use it only for evaluation purposes. We would appreciate your feedback in YouTrack.
>
> 

The Kotlin Power-assert compiler plugin improves the debugging experience by providing detailed failure messages with contextual information. It simplifies the process of writing tests by automatically generating intermediate values in failure messages. It helps you understand why a test failed without needing complex assertion libraries.

This is an example message provided by the plugin:

```
Incorrect length
assert(hello.length == world.substring(1, 4).length) { "Incorrect length" }
|     |      |  |     |               |
|     |      |  |     |               3
|     |      |  |     orl
|     |      |  world!
|     |      false
|     5
       Hello
```

The Power-assert plugin key features:

- **Enhanced error messages**: The plugin captures and displays the values of variables and sub-expressions within the assertion to clearly identify the cause of failure.
- **Simplified testing**: Automatically generates informative failure messages, reducing the need for complex assertion libraries.
- **Support for multiple functions**: By default, it transforms `assert()` function calls but can also transform other functions, such as `require()`, `check()`, and `assertTrue()`.

Apply the plugin

To enable the Power-assert plugin, configure your `build.gradle(.kts)` file as follows:

> 【Kotlin】

```
// build.gradle.kts
```

```
plugins {
    kotlin("multiplatform") version "2.0.0"
    kotlin("plugin.power-assert") version "2.0.0"
}
```

> 【Groovy】

```
// build.gradle
plugins {
    id 'org.jetbrains.kotlin.multiplatform' version '2.0.0'
    id 'org.jetbrains.kotlin.plugin.power-assert' version '2.0.0'
}
```

Configure the plugin

The Power-assert plugin provides several options to customize its behavior:

- **`functions`** : A list of fully-qualified function paths. The Power-assert plugin will transform the calls to these functions. If not specified, only `kotlin.assert()` calls will be transformed by default.
- **`includedSourceSets`** : A list of Gradle source sets that the Power-assert plugin will transform. If not specified, all *test source sets* will be transformed by default.

To customize the behavior, add the `powerAssert {}` block to your build script file:

> 【Kotlin】

```
// build.gradle.kts
powerAssert {
    functions = listOf("kotlin.assert", "kotlin.test.assertTrue", "kotlin.test.assertEquals", "kotlin.test.assertNull")
    includedSourceSets = listOf("commonMain", "jvmMain", "jsMain", "nativeMain")
}
```

> 【Groovy】

```
// build.gradle
powerAssert {
```

```
    functions = ["kotlin.assert", "kotlin.test.assertTrue", "kotlin.test.assertEquals"
, "kotlin.test.assertNull"]
    includedSourceSets = ["commonMain", "jvmMain", "jsMain", "nativeMain"]
}
```

Since the plugin is Experimental, you will see warnings every time you build your app. To exclude these warnings, add this `@OptIn` annotation before declaring the `powerAssert {}` block:

```
import org.jetbrains.kotlin.gradle.ExperimentalKotlinGradlePluginApi

@OptIn(ExperimentalKotlinGradlePluginApi::class)
powerAssert {
    ...
}
```

Use the plugin

This section provides examples of using the Power-assert compiler plugin.

See the complete code of the build script file `build.gradle.kts` for all these examples:

```
import org.jetbrains.kotlin.gradle.ExperimentalKotlinGradlePluginApi

plugins {
    kotlin("jvm") version "2.1.21"
    kotlin("plugin.power-assert") version "2.1.21"
}

group = "org.example"
version = "1.0-SNAPSHOT"

repositories {
    mavenCentral()
}

dependencies {
    testImplementation(kotlin("test"))
}

tasks.test {
    useJUnitPlatform()
}

@OptIn(ExperimentalKotlinGradlePluginApi::class)
powerAssert {
    functions = listOf("kotlin.assert", "kotlin.test.assertEquals", "kotlin.test.asser
```

```
tTrue", "kotlin.test.assertNull", "kotlin.require", "org.example.AssertScope.assert")
}
```

Assert function

Consider the following test with the `assert()` function:

```
import kotlin.test.Test

class SampleTest {

    @Test
    fun testFunction() {
        val hello = "Hello"
        val world = "world!"
        assert(hello.length == world.substring(1, 4).length) { "Incorrect length" }
    }
}
```

If you run the `testFunction()` test with the Power-assert plugin enabled, you get the explicit failure message:

```
Incorrect length
assert(hello.length == world.substring(1, 4).length) { "Incorrect length" }
|     |      |  |     |               |
|     |      |  |     |               3
|     |      |  |     orl
|     |      |  world!
|     |      false
|     5
       Hello
```

To get a more complete error message, always inline the variable into the test function parameters. Consider the following test function:

```
class ComplexExampleTest {

    data class Person(val name: String, val age: Int)

    @Test
    fun testComplexAssertion() {
        val person = Person("Alice", 10)
        val isValidName = person.name.startsWith("A") && person.name.length > 3
        val isValidAge = person.age in 21..28
        assert(isValidName && isValidAge)
    }
}
```

The output of the executed code doesn't provide enough information to find the cause of the problem:

```
Assertion failed
assert(isValidName && isValidAge)
|              |
|              false
       true
```

Inline the variable into the `assert()` function:

```
class ComplexExampleTest {

    data class Person(val name: String, val age: Int)

    @Test
    fun testComplexAssertion() {
        val person = Person("Alice", 10)
        assert(person.name.startsWith("A") && person.name.length > 3 && person.age > 20
 && person.age < 29)
    }
}
```

After execution, you get more explicit information about what went wrong:

```
Assertion failed
assert(person.name.startsWith("A") && person.name.length > 3 && person.age > 20 && per
son.age < 29)
|      |    |                 |      |    |      |      |      |   |
|      |    |                 |      |    |      |      |      |   false
|      |    |                 |      |    |      |      |      10
|      |    |                 |      |    |      |      Person(name=Alice, age=10)
|      |    |                 |      |    |      true
|      |    |                 |      |    5
|      |    |                 |      Alice
|      |    |                 Person(name=Alice, age=10)
|      |    true
|      Alice
       Person(name=Alice, age=10)
```

Beyond assert function

The Power-assert plugin can transform various functions beyond `assert` which is transformed by default. Functions like `require()`, `check()`, `assertTrue()`, `assertEqual()` and others can also be transformed, if they have a form that allows taking a `String` or `() -> String` value as the last parameter.

Before using a new function in a test, specify the function in the `powerAssert {}` block of your build script file. For example, the `require()` function:

```
// build.gradle.kts
import org.jetbrains.kotlin.gradle.ExperimentalKotlinGradlePluginApi

@OptIn(ExperimentalKotlinGradlePluginApi::class)
powerAssert {
    functions = listOf("kotlin.assert", "kotlin.require")
}
```

After adding the function, you can use it in your tests:

```
class RequireExampleTest {

    @Test
    fun testRequireFunction() {
        val value = ""
        require(value.isNotEmpty()) { "Value should not be empty" }
    }
}
```

The output for this example uses the Power-assert plugin to provide detailed information about the failed test:

```
Value should not be empty
require(value.isNotEmpty()) { "Value should not be empty" }
|     |
|     false
```

The message shows the intermediate values that lead to the failure, making it easier to debug.

Soft assertions

The Power-assert plugin supports soft assertions, which do not immediately fail the test but instead collect assertion failures and report them at the end of the test run. This can be useful when you want to see all assertion failures in a single run without stopping at the first failure.

To enable soft assertions, implement the way you will collect error messages:

```
fun <R> assertSoftly(block: AssertScope.() -> R): R {
    val scope = AssertScopeImpl()
    val result = scope.block()
    if (scope.errors.isNotEmpty()) {
        throw AssertionError(scope.errors.joinToString("\n"))
    }
```

```
    return result
}

interface AssertScope {
    fun assert(assertion: Boolean, message: (() -> String)? = null)
}

class AssertScopeImpl : AssertScope {
    val errors = mutableListOf<String>()
    override fun assert(assertion: Boolean, message: (() -> String)?) {
        if (!assertion) {
            errors.add(message?.invoke() ?: "Assertion failed")
        }
    }
}
```

Add these functions to the `powerAssert {}` block to make them available for the Power-assert plugin:

```
@OptIn(ExperimentalKotlinGradlePluginApi::class)
powerAssert {
    functions = listOf("kotlin.assert", "kotlin.test.assert", "org.example.AssertScope
.assert")
}
```

> You should specify the full name of the package where you declare the `AssertScope.assert()` function.
>
> 

After that, you could use it in your test code:

```
// Import the assertSoftly() function
import org.example.assertSoftly

class SoftAssertExampleTest1 {

    data class Employee(val name: String, val age: Int, val salary: Int)

    @Test
    fun `test employees data`() {
        val employees = listOf(
            Employee("Alice", 30, 60000),
            Employee("Bob", 45, 80000),
            Employee("Charlie", 55, 40000),
            Employee("Dave", 150, 70000)
        )

        assertSoftly {
            for (employee in employees) {
```

```
                assert(employee.age < 100) { "${employee.name} has an invalid age: ${e
mployee.age}" }
                assert(employee.salary > 50000) { "${employee.name} has an invalid sal
ary: ${employee.salary}" }
            }
        }
    }
}
```

In the output, all the `assert()` function error messages will be printed one after another:

```
Charlie has an invalid salary: 40000
assert(employee.salary > 50000) { "${employee.name} has an invalid salary: ${employee.
salary}" }
|        |      |
|        |      false
|        40000
       Employee(name=Charlie, age=55, salary=40000)
Dave has an invalid age: 150
assert(employee.age < 100) { "${employee.name} has an invalid age: ${employee.age}" }
|        |   |
|        |   false
|        150
       Employee(name=Dave, age=150, salary=70000)
```

下一步做什么

- Look through a simple project with the plugin enabled and a more complex project with multiple source sets.

Compose 编译器

- Compose 编译器迁移指南
- Compose 编译器选项 DSL

Compose 编译器迁移指南

The Compose compiler is supplemented by a Gradle plugin, which simplifies setup and offers easier access to compiler options. When applied with the Android Gradle plugin (AGP), this Compose compiler plugin will override the coordinates of the Compose compiler supplied automatically by AGP.

The Compose compiler has been merged into the Kotlin repository since Kotlin 2.0.0. This helps smooth the migration of your projects to Kotlin 2.0.0 and later, as the Compose compiler ships simultaneously with Kotlin and will always be compatible with Kotlin of the same version.

To use the new Compose compiler plugin in your project, apply it for each module that uses Compose. Read on for details on how to migrate a Jetpack Compose project. For a Compose Multiplatform project, refer to the multiplatform migration guide.

Migrating a Jetpack Compose project

When migrating to Kotlin 2.0.0 or newer from 1.9, you should adjust your project configuration depending on the way you deal with the Compose compiler. We recommend using the Kotlin Gradle plugin and the Compose compiler Gradle plugin to automate configuration management.

Managing the Compose compiler with Gradle plugins

For Android modules:

1. Add the Compose compiler Gradle plugin to the Gradle version catalog:

```
[versions]
# ...
kotlin = "2.1.21"

[plugins]
# ...
org-jetbrains-kotlin-android = { id = "org.jetbrains.kotlin.android", version.ref
= "kotlin" }
compose-compiler = { id = "org.jetbrains.kotlin.plugin.compose", version.ref = "ko
tlin" }
```

2. Add the Gradle plugin to the root `build.gradle.kts` file:

```
plugins {
```

```
  // ...
  alias(libs.plugins.compose.compiler) apply false
}
```

3. Apply the plugin to every module that uses Jetpack Compose:

```
plugins {
  // ...
  alias(libs.plugins.compose.compiler)
}
```

4. If you are using compiler options for the Jetpack Compose compiler, set them in the `composeCompiler {}` block. See the list of compiler options for reference.

5. If you reference Compose compiler artifacts directly, you can remove these references and let the Gradle plugins take care of things.

Using Compose compiler without Gradle plugins

If you are not using Gradle plugins to manage the Compose compiler, update any direct references to old Maven artifacts in your project:

- Change `androidx.compose.compiler:compiler` to `org.jetbrains.kotlin:kotlin-compose-compiler-plugin-embeddable`
- Change `androidx.compose.compiler:compiler-hosted` to `org.jetbrains.kotlin:kotlin-compose-compiler-plugin`

下一步做什么

- See Google's announcement about the Compose compiler moving to the Kotlin repository.
- If you are using Jetpack Compose to build an Android app, check out our guide on how to make it multiplatform.

Compose 编译器选项 DSL

The Compose compiler Gradle plugin offers a DSL for various compiler options. You can use it to configure the compiler in the `composeCompiler {}` block of the `build.gradle.kts` file for the module you're applying the plugin to.

There are two kinds of options you can specify:

- General compiler settings, which can be disabled or enabled as needed in any given project.
- Feature flags that enable or disable new and experimental features, which should eventually become part of the baseline.

You can find the list of available general settings and the list of supported feature flags in the Compose compiler Gradle plugin API reference.

Here's an example configuration:

```
composeCompiler {
    includeSourceInformation = true

    featureFlags = setOf(
        ComposeFeatureFlag.StrongSkipping.disabled(),
        ComposeFeatureFlag.OptimizeNonSkippingGroups
    )
}
```

> The Gradle plugin provides defaults for several Compose compiler options that were only specified manually before Kotlin 2.0. If you have any of them set up with `freeCompilerArgs`, for example, Gradle reports a duplicate options error.
>
> 

Purpose and use of feature flags

Feature flags are organized into a separate set of options to minimize changes to top-level properties as new flags are continuously rolled out and deprecated.

To enable a feature flag that is disabled by default, specify it in the set, for example:

```
featureFlags = setOf(ComposeFeatureFlag.OptimizeNonSkippingGroups)
```

To disable a feature flag that is enabled by default, call the `disabled()` function on it, for example:

```
featureFlags = setOf(ComposeFeatureFlag.StrongSkipping.disabled())
```

If you are configuring the Compose compiler directly, use the following syntax to pass feature flags to it:

```
-P plugin:androidx.compose.compiler.plugins.kotlin:featureFlag=<flag name>
```

See the list of supported feature flags in the Compose compiler Gradle plugin API reference.

Kotlin 符号处理（KSP）API

- KSP 概述
- KSP 快速入门
- 为什么选用 KSP
- KSP 示例
- KSP 如何为 Kotlin 代码建模
- Java 注解处理对应到 KSP 参考
- 增量处理
- 多轮次处理
- KSP 与 Kotlin 多平台
- 在命令行运行 KSP
- 常见问题

Kotlin 符号处理 API

Kotlin Symbol Processing (*KSP*) is an API that you can use to develop lightweight compiler plugins. KSP provides a simplified compiler plugin API that leverages the power of Kotlin while keeping the learning curve at a minimum. Compared to kapt, annotation processors that use KSP can run up to two times faster.

- To learn more about how KSP compares to kapt, check out why KSP.
- To get started writing a KSP processor, take a look at the KSP quickstart.

Overview

The KSP API processes Kotlin programs idiomatically. KSP understands Kotlin-specific features, such as extension functions, declaration-site variance, and local functions. It also models types explicitly and provides basic type checking, such as equivalence and assign-compatibility.

The API models Kotlin program structures at the symbol level according to Kotlin grammar. When KSP-based plugins process source programs, constructs like classes, class members, functions, and associated parameters are accessible for the processors, while things like `if` blocks and `for` loops are not.

Conceptually, KSP is similar to KType in Kotlin reflection. The API allows processors to navigate from class declarations to corresponding types with specific type arguments and vice-versa. You can also substitute type arguments, specify variances, apply star projections, and mark nullabilities of types.

Another way to think of KSP is as a preprocessor framework of Kotlin programs. By considering KSP-based plugins as *symbol processors*, or simply *processors*, the data flow in a compilation can be described in the following steps:

1. Processors read and analyze source programs and resources.
2. Processors generate code or other forms of output.
3. The Kotlin compiler compiles the source programs together with the generated code.

Unlike a full-fledged compiler plugin, processors cannot modify the code. A compiler plugin that changes language semantics can sometimes be very confusing. KSP avoids that by treating the source programs as read-only.

You can also get an overview of KSP in this video:

YouTube 视频：Kotlin Symbol Processing (KSP)

How KSP looks at source files

Most processors navigate through the various program structures of the input source code. Before diving into usage of the API, let's see at how a file might look from KSP's point of view:

```
KSFile
  packageName: KSName
  fileName: String
  annotations: List<KSAnnotation>  (File annotations)
  declarations: List<KSDeclaration>
    KSClassDeclaration // class, interface, object
      simpleName: KSName
      qualifiedName: KSName
      containingFile: String
      typeParameters: KSTypeParameter
      parentDeclaration: KSDeclaration
      classKind: ClassKind
      primaryConstructor: KSFunctionDeclaration
      superTypes: List<KSTypeReference>
      // contains inner classes, member functions, properties, etc.
      declarations: List<KSDeclaration>
    KSFunctionDeclaration // top level function
      simpleName: KSName
      qualifiedName: KSName
      containingFile: String
      typeParameters: KSTypeParameter
      parentDeclaration: KSDeclaration
      functionKind: FunctionKind
      extensionReceiver: KSTypeReference?
      returnType: KSTypeReference
      parameters: List<KSValueParameter>
      // contains local classes, local functions, local variables, etc.
      declarations: List<KSDeclaration>
    KSPropertyDeclaration // global variable
      simpleName: KSName
      qualifiedName: KSName
      containingFile: String
      typeParameters: KSTypeParameter
```

```
        parentDeclaration: KSDeclaration
        extensionReceiver: KSTypeReference?
        type: KSTypeReference
        getter: KSPropertyGetter
          returnType: KSTypeReference
        setter: KSPropertySetter
          parameter: KSValueParameter
```

This view lists common things that are declared in the file: classes, functions, properties, and so on.

SymbolProcessorProvider: the entry point

KSP expects an implementation of the `SymbolProcessorProvider` interface to instantiate `SymbolProcessor`:

```
interface SymbolProcessorProvider {
    fun create(environment: SymbolProcessorEnvironment): SymbolProcessor
}
```

While `SymbolProcessor` is defined as:

```
interface SymbolProcessor {
    fun process(resolver: Resolver): List<KSAnnotated> // Let's focus on this
    fun finish() {}
    fun onError() {}
}
```

A `Resolver` provides `SymbolProcessor` with access to compiler details such as symbols. A processor that finds all top-level functions and non-local functions in top-level classes might look something like the following:

```
class HelloFunctionFinderProcessor : SymbolProcessor() {
    // ...
    val functions = mutableListOf<KSClassDeclaration>()
    val visitor = FindFunctionsVisitor()

    override fun process(resolver: Resolver) {
        resolver.getAllFiles().forEach { it.accept(visitor, Unit) }
    }

    inner class FindFunctionsVisitor : KSVisitorVoid() {
        override fun visitClassDeclaration(classDeclaration: KSClassDeclaration, data:
Unit) {
            classDeclaration.getDeclaredFunctions().forEach { it.accept(this, Unit) }
        }
```

```
        override fun visitFunctionDeclaration(function: KSFunctionDeclaration, data: U
nit) {
            functions.add(function)
        }

        override fun visitFile(file: KSFile, data: Unit) {
            file.declarations.forEach { it.accept(this, Unit) }
        }
    }
    // ...

    class Provider : SymbolProcessorProvider {
        override fun create(environment: SymbolProcessorEnvironment): SymbolProcessor
= TODO()
    }
}
```

Resources

- Quickstart
- Why use KSP?
- Examples
- How KSP models Kotlin code
- Reference for Java annotation processor authors
- Incremental processing notes
- Multiple round processing notes
- KSP on multiplatform projects
- Running KSP from command line
- FAQ

Supported libraries

The table includes a list of popular libraries on Android and their various stages of support for KSP:

Library	Status
Room	Officially supported
Moshi	Officially supported
RxHttp	Officially supported
Kotshi	Officially supported
Lyricist	Officially supported

Lich SavedState	Officially supported
gRPC Dekorator	Officially supported
EasyAdapter	Officially supported
Koin Annotations	Officially supported
Glide	Officially supported
Micronaut	Officially supported
Epoxy	Officially supported
Paris	Officially supported
Auto Dagger	Officially supported
SealedX	Officially supported
Ktorfit	Officially supported
Mockative	Officially supported
DeeplinkDispatch	Supported via airbnb/DeepLinkDispatch#323
Dagger	Alpha
Motif	Alpha
Hilt	In progress
Auto Factory	Not yet supported

KSP 快速入门

For a quick start, you can create your own processor or get a sample one.

Add a processor

To add a processor, you need to include the KSP Gradle Plugin and add a dependency on the processor:

1. Add the KSP Gradle Plugin `com.google.devtools.ksp` to your `build.gradle(.kts)` file:

> 【Kotlin】

```
plugins {
    id("com.google.devtools.ksp") version "2.1.21-2.0.1"
}
```

> 【Groovy】

```
plugins {
    id 'com.google.devtools.ksp' version '2.1.21-2.0.1'
}
```

---

1. Add a dependency on the processor. This example uses Dagger. Replace it with the processor you want to add.

> 【Kotlin】

```
dependencies {
    implementation("com.google.dagger:dagger-compiler:2.51.1")
    ksp("com.google.dagger:dagger-compiler:2.51.1")
}
```

> 【Groovy】

```
    dependencies {
        implementation 'com.google.dagger:dagger-compiler:2.51.1'
        ksp 'com.google.dagger:dagger-compiler:2.51.1'
    }
```

---

1. Run `./gradlew build`. You can find the generated code in the `build/generated/ksp` directory.

Here is a full example:

> 【Kotlin】

```
plugins {
    id("com.google.devtools.ksp") version "2.1.21-2.0.1"
    kotlin("jvm")
}

repositories {
    mavenCentral()
}

dependencies {
    implementation(kotlin("stdlib-jdk8"))
    implementation("com.google.dagger:dagger-compiler:2.51.1")
    ksp("com.google.dagger:dagger-compiler:2.51.1")
}
```

> 【Groovy】

```
plugins {
    id 'com.google.devtools.ksp' version '2.1.21-2.0.1'
    id 'org.jetbrains.kotlin.jvm' version '2.1.21'
}

repositories {
    mavenCentral()
}

dependencies {
    implementation 'org.jetbrains.kotlin:kotlin-stdlib:2.1.21'
    implementation 'com.google.dagger:dagger-compiler:2.51.1'
```

```
    ksp 'com.google.dagger:dagger-compiler:2.51.1'
}
```

Create a processor of your own

1. Create an empty gradle project.
2. Specify version `2.1.21` of the Kotlin plugin in the root project for use in other project modules:

> 【Kotlin】

```
plugins {
    kotlin("jvm") version "2.1.21" apply false
}

buildscript {
    dependencies {
        classpath(kotlin("gradle-plugin", version = "2.1.21"))
    }
}
```

> 【Groovy】

```
plugins {
    id 'org.jetbrains.kotlin.jvm' version '2.1.21' apply false
}

buildscript {
    dependencies {
        classpath 'org.jetbrains.kotlin:kotlin-gradle-plugin:2.1.21'
    }
}
```

1. Add a module for hosting the processor.
2. In the module's build script, apply Kotlin plugin and add the KSP API to the `dependencies` block.

> 【Kotlin】

```
plugins {
    kotlin("jvm")
}

repositories {
    mavenCentral()
}

dependencies {
    implementation("com.google.devtools.ksp:symbol-processing-api:2.1.21-2.0.1")
}
```

> 【Groovy】

```
plugins {
    id 'org.jetbrains.kotlin.jvm' version '2.1.21'
}

repositories {
    mavenCentral()
}

dependencies {
    implementation 'com.google.devtools.ksp:symbol-processing-api:2.1.21-2.0.1'
}
```

---

1. You'll need to implement `com.google.devtools.ksp.processing.SymbolProcessor` and `com.google.devtools.ksp.processing.SymbolProcessorProvider`. Your implementation of `SymbolProcessorProvider` will be loaded as a service to instantiate the `SymbolProcessor` you implement. Note the following:
   - Implement `SymbolProcessorProvider.create()` to create a `SymbolProcessor`. Pass dependencies that your processor needs (such as `CodeGenerator`, processor options) through the parameters of `SymbolProcessorProvider.create()`.
   - Your main logic should be in the `SymbolProcessor.process()` method.
   - Use `resolver.getSymbolsWithAnnotation()` to get the symbols you want to process, given the fully-qualified name of an annotation.
   - A common use case for KSP is to implement a customized visitor (interface `com.google.devtools.ksp.symbol.KSVisitor`) for operating on symbols. A simple template visitor is `com.google.devtools.ksp.symbol.KSDefaultVisitor`.
   - For sample implementations of the `SymbolProcessorProvider` and `SymbolProcessor`

interfaces, see the following files in the sample project.

- `src/main/kotlin/BuilderProcessor.kt`
- `src/main/kotlin/TestProcessor.kt`

- After writing your own processor, register your processor provider to the package by including its fully-qualified name in `src/main/resources/META-INF/services/com.google.devtools.ksp.processing.SymbolProcessorProvider`.

Use your own processor in a project

1. Create another module that contains a workload where you want to try out your processor.

> 【Kotlin】

```
pluginManagement {
    repositories {
        gradlePluginPortal()
    }
}
```

> 【Groovy】

```
pluginManagement {
    repositories {
        gradlePluginPortal()
    }
}
```

---

1. In the module's build script, apply the `com.google.devtools.ksp` plugin with the specified version and add your processor to the list of dependencies.

> 【Kotlin】

```
plugins {
    id("com.google.devtools.ksp") version "2.1.21-2.0.1"
}

dependencies {
```

```
        implementation(kotlin("stdlib-jdk8"))
        implementation(project(":test-processor"))
        ksp(project(":test-processor"))
    }
```

> 【Groovy】

```
    plugins {
        id 'com.google.devtools.ksp' version '2.1.21-2.0.1'
    }

    dependencies {
        implementation 'org.jetbrains.kotlin:kotlin-stdlib:2.1.21'
        implementation project(':test-processor')
        ksp project(':test-processor')
    }
```

---

1. Run `./gradlew build`. You can find the generated code under `build/generated/ksp`.

Here's a sample build script to apply the KSP plugin to a workload:

> 【Kotlin】

```
plugins {
    id("com.google.devtools.ksp") version "2.1.21-2.0.1"
    kotlin("jvm")
}

repositories {
    mavenCentral()
}

dependencies {
    implementation(kotlin("stdlib-jdk8"))
    implementation(project(":test-processor"))
    ksp(project(":test-processor"))
}
```

> 【Groovy】

```
plugins {
```

```
    id 'com.google.devtools.ksp' version '2.1.21-2.0.1'
    id 'org.jetbrains.kotlin.jvm' version '2.1.21'
}

repositories {
    mavenCentral()
}

dependencies {
    implementation 'org.jetbrains.kotlin:kotlin-stdlib:2.1.21'
    implementation project(':test-processor')
    ksp project(':test-processor')
}
```

Pass options to processors

Processor options in `SymbolProcessorEnvironment.options` are specified in gradle build scripts:

```
ksp {
    arg("option1", "value1")
    arg("option2", "value2")
    ...
}
```

Make IDE aware of generated code

> Generated source files are registered automatically since KSP 1.8.0-1.0.9. If you're using KSP 1.0.9 or newer and don't need to make the IDE aware of generated resources, feel free to skip this section.
>
> 

By default, IntelliJ IDEA or other IDEs don't know about the generated code. So it will mark references to generated symbols unresolvable. To make an IDE be able to reason about the generated symbols, mark the following paths as generated source roots:

```
build/generated/ksp/main/kotlin/
build/generated/ksp/main/java/
```

If your IDE supports resource directories, also mark the following one:

```
build/generated/ksp/main/resources/
```

It may also be necessary to configure these directories in your KSP consumer module's build script:

> 【Kotlin】

```
kotlin {
    sourceSets.main {
        kotlin.srcDir("build/generated/ksp/main/kotlin")
    }
    sourceSets.test {
        kotlin.srcDir("build/generated/ksp/test/kotlin")
    }
}
```

> 【Groovy】

```
kotlin {
    sourceSets {
        main.kotlin.srcDirs += 'build/generated/ksp/main/kotlin'
        test.kotlin.srcDirs += 'build/generated/ksp/test/kotlin'
    }
}
```

If you are using IntelliJ IDEA and KSP in a Gradle plugin then the above snippet will give the following warning:

```
Execution optimizations have been disabled for task ':publishPluginJar' to ensure corr
ectness due to the following reasons:
Gradle detected a problem with the following location: '../build/generated/ksp/main/ko
tlin'.
Reason: Task ':publishPluginJar' uses this output of task ':kspKotlin' without declari
ng an explicit or implicit dependency.
```

In this case, use the following script instead:

> 【Kotlin】

```
plugins {
    // ...
```

```
    idea
}

idea {
    module {
        // Not using += due to https://github.com/gradle/gradle/issues/8749
        sourceDirs = sourceDirs + file("build/generated/ksp/main/kotlin") // or tasks[
"kspKotlin"].destination
        testSourceDirs = testSourceDirs + file("build/generated/ksp/test/kotlin")
        generatedSourceDirs = generatedSourceDirs + file("build/generated/ksp/main/kot
lin") + file("build/generated/ksp/test/kotlin")
    }
}
```

> 【Groovy】

```
plugins {
    // ...
    id 'idea'
}

idea {
    module {
        // Not using += due to https://github.com/gradle/gradle/issues/8749
        sourceDirs = sourceDirs + file('build/generated/ksp/main/kotlin') // or tasks[
"kspKotlin"].destination
        testSourceDirs = testSourceDirs + file('build/generated/ksp/test/kotlin')
        generatedSourceDirs = generatedSourceDirs + file('build/generated/ksp/main/kot
lin') + file('build/generated/ksp/test/kotlin')
    }
}
```

为什么选用 KSP

Compiler plugins are powerful metaprogramming tools that can greatly enhance how you write code. Compiler plugins call compilers directly as libraries to analyze and edit input programs. These plugins can also generate output for various uses. For example, they can generate boilerplate code, and they can even generate full implementations for specially-marked program elements, such as `Parcelable`. Plugins have a variety of other uses and can even be used to implement and fine-tune features that are not provided directly in a language.

While compiler plugins are powerful, this power comes at a price. To write even the simplest plugin, you need to have some compiler background knowledge, as well as a certain level of familiarity with the implementation details of your specific compiler. Another practical issue is that plugins are often closely tied to specific compiler versions, meaning you might need to update your plugin each time you want to support a newer version of the compiler.

KSP makes creating lightweight compiler plugins easier

KSP is designed to hide compiler changes, minimizing maintenance efforts for processors that use it. KSP is designed not to be tied to the JVM so that it can be adapted to other platforms more easily in the future. KSP is also designed to minimize build times. For some processors, such as Glide, KSP reduces full compilation times by up to 25% when compared to kapt.

KSP is itself implemented as a compiler plugin. There are prebuilt packages on Google's Maven repository that you can download and use without having to build the project yourself.

Comparison to kotlinc compiler plugins

`kotlinc` compiler plugins have access to almost everything from the compiler and therefore have maximum power and flexibility. On the other hand, because these plugins can potentially depend on anything in the compiler, they are sensitive to compiler changes and need to be maintained frequently. These plugins also require a deep understanding of `kotlinc`'s implementation, so the learning curve can be steep.

KSP aims to hide most compiler changes through a well-defined API, though major changes in compiler or even the Kotlin language might still require to be exposed to API users.

KSP tries to fulfill common use cases by providing an API that trades power for simplicity. Its capability is a strict subset of a general `kotlinc` plugin. For example, while `kotlinc` can examine expressions and statements and can even modify code, KSP cannot.

While writing a `kotlinc` plugin can be a lot of fun, it can also take a lot of time. If you aren't in a position to learn `kotlinc` 's implementation and do not need to modify source code or read expressions, KSP might be a good fit.

Comparison to reflection

KSP's API looks similar to `kotlin.reflect` . The major difference between them is that type references in KSP need to be resolved explicitly. This is one of the reasons why the interfaces are not shared.

Comparison to kapt

kapt is a remarkable solution which makes a large amount of Java annotation processors work for Kotlin programs out-of-box. The major advantages of KSP over kapt are improved build performance, not tied to JVM, a more idiomatic Kotlin API, and the ability to understand Kotlin-only symbols.

To run Java annotation processors unmodified, kapt compiles Kotlin code into Java stubs that retain information that Java annotation processors care about. To create these stubs, kapt needs to resolve all symbols in the Kotlin program. The stub generation costs roughly 1/3 of a full `kotlinc` analysis and the same order of `kotlinc` code-generation. For many annotation processors, this is much longer than the time spent in the processors themselves. For example, Glide looks at a very limited number of classes with a predefined annotation, and its code generation is fairly quick. Almost all of the build overhead resides in the stub generation phase. Switching to KSP would immediately reduce the time spent in the compiler by 25%.

For performance evaluation, we implemented a simplified version of Glide in KSP to make it generate code for the Tachiyomi project. While the total Kotlin compilation time of the project is 21.55 seconds on our test device, it took 8.67 seconds for kapt to generate the code, and it took 1.15 seconds for our KSP implementation to generate the code.

Unlike kapt, processors in KSP do not see input programs from Java's point of view. The API is more natural to Kotlin, especially for Kotlin-specific features such as top-level functions. Because KSP doesn't delegate to `javac` like kapt, it doesn't assume JVM-specific behaviors and can be used with other platforms potentially.

Limitations

While KSP tries to be a simple solution for most common use cases, it has made several trade-offs compared to other plugin solutions. The following are not goals of KSP:

- Examining expression-level information of source code.
- Modifying source code.
- 100% compatibility with the Java Annotation Processing API.

KSP 示例

Get all member functions

```
fun KSClassDeclaration.getDeclaredFunctions(): Sequence<KSFunctionDeclaration> =
    declarations.filterIsInstance<KSFunctionDeclaration>()
```

Check whether a class or function is local

```
fun KSDeclaration.isLocal(): Boolean =
    parentDeclaration != null && parentDeclaration !is KSClassDeclaration
```

Find the actual class or interface declaration that the type alias points to

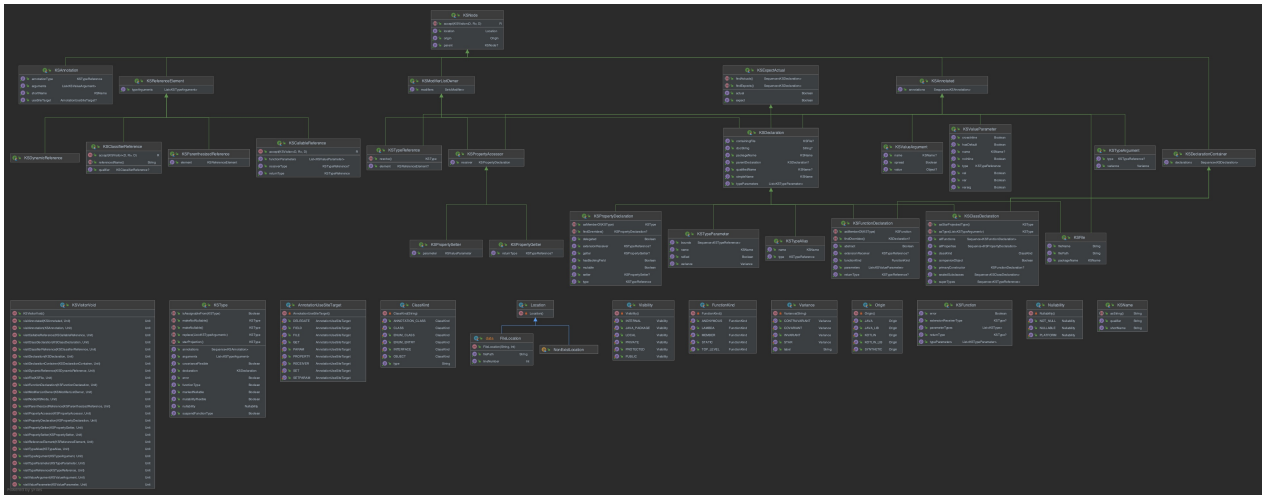
```
fun KSTypeAlias.findActualType(): KSClassDeclaration {
    val resolvedType = this.type.resolve().declaration
    return if (resolvedType is KSTypeAlias) {
        resolvedType.findActualType()
    } else {
        resolvedType as KSClassDeclaration
    }
}
```

Collect suppressed names in a file annotation

```
// @file:kotlin.Suppress("Example1", "Example2")
fun KSFile.suppressedNames(): Sequence<String> = annotations
    .filter {
        it.shortName.asString() == "Suppress" &&
        it.annotationType.resolve().declaration.qualifiedName?.asString() == "kotlin.S
uppress"
    }.flatMap {
        it.arguments.flatMap {
            (it.value as Array<String>).toList()
        }
    }
```

KSP 如何为 Kotlin 代码建模

You can find the API definition in the KSP GitHub repository. The diagram shows an overview of how Kotlin is modeled in KSP:



{thumbnail="true" width="800" thumbnail-same-file="true"}

> See the full-sized diagram.
>
> 

Type and resolution

The resolution takes most of the cost of the underlying API implementation. So type references are designed to be resolved by processors explicitly (with a few exceptions). When a *type* (such as `KSFunctionDeclaration.returnType` or `KSAnnotation.annotationType`) is referenced, it is always a `KSTypeReference`, which is a `KSReferenceElement` with annotations and modifiers.

```
interface KSFunctionDeclaration : ... {
  val returnType: KSTypeReference?
  // ...
}

interface KSTypeReference : KSAnnotated, KSModifierListOwner {
  val type: KSReferenceElement
}
```

A `KSTypeReference` can be resolved to a `KSType`, which refers to a type in Kotlin's type system.

A `KSTypeReference` has a `KSReferenceElement`, which models Kotlin's program structure: namely, how the reference is written. It corresponds to the `type` element in Kotlin's grammar.

A `KSReferenceElement` can be a `KSClassifierReference` or `KSCallableReference`, which contains a lot of useful information without the need for resolution. For example, `KSClassifierReference` has `referencedName`, while `KSCallableReference` has `receiverType`, `functionArguments`, and `returnType`.

If the original declaration referenced by a `KSTypeReference` is needed, it can usually be found by resolving to `KSType` and accessing through `KSType.declaration`. Moving from where a type is mentioned to where its class is defined looks like this:

```
val ksType: KSType = ksTypeReference.resolve()
val ksDeclaration: KSDeclaration = ksType.declaration
```

Type resolution is costly and therefore has explicit form. Some of the information obtained from resolution is already available in `KSReferenceElement`. For example, `KSClassifierReference.referencedName` can filter out a lot of elements that are not interesting. You should resolve type only if you need specific information from `KSDeclaration` or `KSType`.

`KSTypeReference` pointing to a function type has most of its information in its element. Although it can be resolved to the family of `Function0`, `Function1`, and so on, these resolutions don't bring any more information than `KSCallableReference`. One use case for resolving function type references is dealing with the identity of the function's prototype.

Java 注解处理对应到 KSP 参考

Program elements

Java	Closest facility in KSP	Notes
`AnnotationMirror`	`KSAnnotation`	
`AnnotationValue`	`KSValueArguments`	
`Element`	`KSDeclaration` / `KSDeclarationContainer`	
`ExecutableElement`	`KSFunctionDeclaration`	
`PackageElement`	`KSFile`	KSP doesn't model packages as program elements
`Parameterizable`	`KSDeclaration`	
`QualifiedNameable`	`KSDeclaration`	
`TypeElement`	`KSClassDeclaration`	
`TypeParameterElement`	`KSTypeParameter`	
`VariableElement`	`KSValueParameter` / `KSPropertyDeclaration`	

Types

KSP requires explicit type resolution, so some functionalities in Java can only be carried out by `KSType` and the corresponding elements before resolution.

Java	Closest facility in KSP	Notes
`ArrayType`	`KSBuiltIns.arrayType`	
`DeclaredType`	`KSType` / `KSClassifierReference`	
`ErrorType`	`KSType.isError`	
`ExecutableType`	`KSType` / `KSCallableReference`	
`IntersectionType`	`KSType` / `KSTypeParameter`	
`NoType`	`KSType.isError`	N/A in KSP
`NullType`		N/A in KSP
`PrimitiveType`	`KSBuiltIns`	Not exactly same as primitive type in Java

`ReferenceType`	`KSTypeReference`	
`TypeMirror`	`KSType`	
`TypeVariable`	`KSTypeParameter`	
`UnionType`	N/A	Kotlin has only one type per catch block. `UnionType` is also not observable by even Java annotation processors
`WildcardType`	`KSType` / `KSTypeArgument`	

Misc

Java	Closest facility in KSP	Notes
`Name`	`KSName`	
`ElementKind`	`ClassKind` / `FunctionKind`	
`Modifier`	`Modifier`	
`NestingKind`	`ClassKind` / `FunctionKind`	
`AnnotationValueVisitor`		
`ElementVisitor`	`KSVisitor`	
`AnnotatedConstruct`	`KSAnnotated`	
`TypeVisitor`		
`TypeKind`	`KSBuiltIns`	Some can be found in builtins, otherwise check `KSClassDeclaration` for `DeclaredType`
`ElementFilter`	`Collection.filterIsInstance`	
`ElementKindVisitor`	`KSVisitor`	
`ElementScanner`	`KSTopDownVisitor`	
`SimpleAnnotationValueVisitor`		Not needed in KSP
`SimpleElementVisitor`	`KSVisitor`	
`SimpleTypeVisitor`		
`TypeKindVisitor`		
`Types`	`Resolver` / `utils`	Some of the `utils` are also integrated into symbol interfaces
`Elements`	`Resolver` / `utils`	

Details

See how functionalities of Java annotation processing API can be carried out by KSP.

AnnotationMirror

Java	KSP equivalent
`getAnnotationType`	`ksAnnotation.annotationType`
`getElementValues`	`ksAnnotation.arguments`

AnnotationValue

Java	KSP equivalent
`getValue`	`ksValueArgument.value`

Element

Java	KSP equivalent
`asType`	`ksClassDeclaration.asType(...)` is available for `KSClassDeclaration` only. Type arguments need to be supplied.
`getAnnotation`	To be implemented
`getAnnotationMirrors`	`ksDeclaration.annotations`
`getEnclosedElements`	`ksDeclarationContainer.declarations`
`getEnclosingElements`	`ksDeclaration.parentDeclaration`
`getKind`	Type check and cast following `ClassKind` or `FunctionKind`
`getModifiers`	`ksDeclaration.modifiers`
`getSimpleName`	`ksDeclaration.simpleName`

ExecutableElement

Java	KSP equivalent
`getDefaultValue`	To be implemented
`getParameters`	`ksFunctionDeclaration.parameters`
`getReceiverType`	`ksFunctionDeclaration.parentDeclaration`
`getReturnType`	`ksFunctionDeclaration.returnType`
`getSimpleName`	`ksFunctionDeclaration.simpleName`
`getThrownTypes`	Not needed in Kotlin
`getTypeParameters`	`ksFunctionDeclaration.typeParameters`
`isDefault`	Check whether parent declaration is an interface or not

`isVarArgs`	`ksFunctionDeclaration.parameters.any { it.isVarArg }`

Parameterizable

Java	KSP equivalent
`getTypeParameters`	`ksFunctionDeclaration.typeParameters`

QualifiedNameable

Java	KSP equivalent
`getQualifiedName`	`ksDeclaration.qualifiedName`

TypeElement

Java	KSP equivalent
`getEnclosedElements`	`ksClassDeclaration.declarations`
`getEnclosingElement`	`ksClassDeclaration.parentDeclaration`
`getInterfaces`	```kotlin // Should be able to do without resolution ksClassDeclaration.superTypes .map { it.resolve() } .filter { (it?.declaration as? KSClassDeclaration)?.classKind == ClassKind.INTERFACE } ```
`getNestingKind`	Check `KSClassDeclaration.parentDeclaration` and `inner` modifier
`getQualifiedName`	`ksClassDeclaration.qualifiedName`
`getSimpleName`	`ksClassDeclaration.simpleName`
`getSuperclass`	```kotlin // Should be able to do without resolution ksClassDeclaration.superTypes .map { it.resolve() } .filter { (it?.declaration as? KSClassDeclaration)?.classKind == ClassKind.CLASS } ```
`getTypeParameters`	`ksClassDeclaration.typeParameters`

TypeParameterElement

Java	KSP equivalent
`getBounds`	`ksTypeParameter.bounds`
`getEnclosingElement`	`ksTypeParameter.parentDeclaration`
`getGenericElement`	`ksTypeParameter.parentDeclaration`

VariableElement

Java	KSP equivalent
`getConstantValue`	To be implemented
`getEnclosingElement`	`ksValueParameter.parentDeclaration`
`getSimpleName`	`ksValueParameter.simpleName`

ArrayType

Java	KSP equivalent
`getComponentType`	`ksType.arguments.first()`

DeclaredType

Java	KSP equivalent
`asElement`	`ksType.declaration`
`getEnclosingType`	`ksType.declaration.parentDeclaration`
`getTypeArguments`	`ksType.arguments`

ExecutableType

> A `KSType` for a function is just a signature represented by the `FunctionN<R, T1, T2, ..., TN>` family.
>
> 

Java	KSP equivalent
`getParameterTypes`	`ksType.declaration.typeParameters`, `ksFunctionDeclaration.parameters.map { it.type }`
`getReceiverType`	`ksFunctionDeclaration.parentDeclaration.asType(...)`
`getReturnType`	`ksType.declaration.typeParameters.last()`
`getThrownTypes`	Not needed in Kotlin
`getTypeVariables`	`ksFunctionDeclaration.typeParameters`

IntersectionType

Java	KSP equivalent
`getBounds`	`ksTypeParameter.bounds`

TypeMirror

Java	KSP equivalent
`getKind`	Compare with types in `KSBuiltIns` for primitive types, `Unit` type, otherwise declared types

TypeVariable

Java	KSP equivalent
`asElement`	`ksType.declaration`
`getLowerBound`	To be decided. Only needed if capture is provided and explicit bound checking is needed.
`getUpperBound`	`ksTypeParameter.bounds`

WildcardType

Java	KSP equivalent
`getExtendsBound`	```kotlin if (ksTypeArgument.variance == Variance.COVARIANT) ksTypeArgument.type else null ```
`getSuperBound`	```kotlin if (ksTypeArgument.variance == Variance.CONTRAVARIANT) ksTypeArgument.type else null ```

Elements

Java	KSP equivalent
`getAllAnnotationMirrors`	`KSDeclarations.annotations`
`getAllMembers`	`getAllFunctions`, `getAllProperties` is to be implemented
`getBinaryName`	To be decided, see Java Specification
`getConstantExpression`	There is constant value, not expression
`getDocComment`	To be implemented
`getElementValuesWithDefaults`	To be implemented
`getName`	`resolver.getKSNameFromString`
`getPackageElement`	Package not supported, while package information can be retri Operation on package is not possible for KSP
`getPackageOf`	Package not supported
`getTypeElement`	`Resolver.getClassDeclarationByName`
`hides`	To be implemented
`isDeprecated`	```kotlin KsDeclaration.annotations.any { it.annotationType.resolve()!!.declaration.qualifiedName!!.asSt

	== Deprecated::class.qualifiedName } ```
`overrides`	`KSFunctionDeclaration.overrides` / `KSPropertyDeclaration.overrides` (member function of respective class)
`printElements`	KSP has basic `toString()` implementation on most classes

Types

Java	KSP equivalent
`asElement`	`ksType.declaration`
`asMemberOf`	`resolver.asMemberOf`
`boxedClass`	Not needed
`capture`	To be decided
`contains`	`KSType.isAssignableFrom`
`directSuperTypes`	`(ksType.declaration as KSClassDeclaration).superTypes`
`erasure`	`ksType.starProjection()`
`getArrayType`	`ksBuiltIns.arrayType.replace(...)`
`getDeclaredType`	`ksClassDeclaration.asType`
`getNoType`	`ksBuiltIns.nothingType` / `null`
`getNullType`	Depending on the context, `KSType.markNullable` could be useful
`getPrimitiveType`	Not needed, check for `KSBuiltins`
`getWildcardType`	Use `Variance` in places expecting `KSTypeArgument`
`isAssignable`	`ksType.isAssignableFrom`
`isSameType`	`ksType.equals`
`isSubsignature`	`functionTypeA == functionTypeB` / `functionTypeA == functionTypeB.starProjection()`
`isSubtype`	`ksType.isAssignableFrom`
`unboxedType`	Not needed

增量处理

Incremental processing is a processing technique that avoids re-processing of sources as much as possible. The primary goal of incremental processing is to reduce the turn-around time of a typical change-compile-test cycle. For general information, see Wikipedia's article on incremental computing.

To determine which sources are *dirty* (those that need to be reprocessed), KSP needs processors' help to identify which input sources correspond to which generated outputs. To help with this often cumbersome and error-prone process, KSP is designed to require only a minimal set of *root sources* that processors use as starting points to navigate the code structure. In other words, a processor needs to associate an output with the sources of the corresponding `KSNode` if the `KSNode` is obtained from any of the following:

- `Resolver.getAllFiles`
- `Resolver.getSymbolsWithAnnotation`
- `Resolver.getClassDeclarationByName`
- `Resolver.getDeclarationsFromPackage`

Incremental processing is currently enabled by default. To disable it, set the Gradle property `ksp.incremental=false`. To enable logs that dump the dirty set according to dependencies and outputs, use `ksp.incremental.log=true`. You can find these log files in the `build` output directory with a `.log` file extension.

On the JVM, classpath changes, as well as Kotlin and Java source changes, are tracked by default. To track only Kotlin and Java source changes, disable classpath tracking by setting the `ksp.incremental.intermodule=false` Gradle property.

Aggregating vs Isolating

Similar to the concepts in Gradle annotation processing, KSP supports both *aggregating* and *isolating* modes. Note that unlike Gradle annotation processing, KSP categorizes each output as either aggregating or isolating, rather than the entire processor.

An aggregating output can potentially be affected by any input changes, except removing files that don't affect other files. This means that any input change results in a rebuild of all aggregating outputs, which in turn means reprocessing of all corresponding registered, new, and modified source files.

As an example, an output that collects all symbols with a particular annotation is considered an aggregating output.

An isolating output depends only on its specified sources. Changes to other sources do not affect an isolating output. Note that unlike Gradle annotation processing, you can define multiple source files for a given output.

As an example, a generated class that is dedicated to an interface it implements is considered isolating.

To summarize, if an output might depend on new or any changed sources, it is considered aggregating. Otherwise, the output is isolating.

Here's a summary for readers familiar with Java annotation processing:

- In an isolating Java annotation processor, all the outputs are isolating in KSP.
- In an aggregating Java annotation processor, some outputs can be isolating and some can be aggregating in KSP.

How it is implemented

The dependencies are calculated by the association of input and output files, instead of annotations. This is a many-to-many relation.

The dirtiness propagation rules due to input-output associations are:

1. If an input file is changed, it will always be reprocessed.
2. If an input file is changed, and it is associated with an output, then all other input files associated with the same output will also be reprocessed. This is transitive, namely, invalidation happens repeatedly until there is no new dirty file.
3. All input files that are associated with one or more aggregating outputs will be reprocessed. In other words, if an input file isn't associated with any aggregating outputs, it won't be reprocessed (unless it meets 1. or 2. in the above).

Reasons are:

1. If an input is changed, new information can be introduced and therefore processors need to run again with the input.
2. An output is made out of a set of inputs. Processors may need all the inputs to regenerate the output.
3. `aggregating=true` means that an output may potentially depend on new information, which can come from either new files, or changed, existing files. `aggregating=false` means that processor is sure that the information only comes from certain input files and never from other or new files.

Example 1

A processor generates `outputForA` after reading class `A` in `A.kt` and class `B` in `B.kt`, where `A` extends `B`. The processor got `A` by `Resolver.getSymbolsWithAnnotation` and then got `B` by `KSClassDeclaration.superTypes` from `A`. Because the inclusion of `B` is due to `A`, `B.kt` doesn't need to be specified in `dependencies` for `outputForA`. You can still specify `B.kt` in this case, but it is unnecessary.

```
// A.kt
@Interesting
class A : B()

// B.kt
open class B

// Example1Processor.kt
class Example1Processor : SymbolProcessor {
    override fun process(resolver: Resolver) {
        val declA = resolver.getSymbolsWithAnnotation("Interesting").first() as KSClas
sDeclaration
        val declB = declA.superTypes.first().resolve().declaration
        // B.kt isn't required, because it can be deduced as a dependency by KSP
        val dependencies = Dependencies(aggregating = true, declA.containingFile!!)
        // outputForA.kt
        val outputName = "outputFor${declA.simpleName.asString()}"
        // outputForA depends on A.kt and B.kt
        val output = codeGenerator.createNewFile(dependencies, "com.example", outputNa
me, "kt")
        output.write("// $declA : $declB\n".toByteArray())
        output.close()
    }
    // ...
}
```

Example 2

Consider that a processor generates `outputA` after reading `sourceA` and `outputB` after reading `sourceB`.

When `sourceA` is changed:

- If `outputB` is aggregating, both `sourceA` and `sourceB` are reprocessed.
- If `outputB` is isolating, only `sourceA` is reprocessed.

When `sourceC` is added:

- If `outputB` is aggregating, both `sourceC` and `sourceB` are reprocessed.
- If `outputB` is isolating, only `sourceC` is reprocessed.

When `sourceA` is removed, nothing needs to be reprocessed.

When `sourceB` is removed, nothing needs to be reprocessed.

How file dirtiness is determined

A dirty file is either directly *changed* by users or indirectly *affected* by other dirty files. KSP propagates dirtiness in two steps:

- Propagation by *resolution tracing*: Resolving a type reference (implicitly or explicitly) is the only way to navigate from one file to another. When a type reference is resolved by a processor, a changed or affected file that contains a change that may potentially affect the resolution result will affect the file containing that reference.
- Propagation by *input-output correspondence*: If a source file is changed or affected, all other source files having some output in common with that file are affected.

Note that both of them are transitive and the second forms equivalence classes.

Reporting bugs

To report a bug, please set Gradle properties `ksp.incremental=true` and `ksp.incremental.log=true`, and perform a clean build. This build produces two log files:

- `build/kspCaches/<source set>/logs/kspDirtySet.log`
- `build/kspCaches/<source set>/logs/kspSourceToOutputs.log`

You can then run successive incremental builds, which will generate two additional log files:

- `build/kspCaches/<source set>/logs/kspDirtySetByDeps.log`
- `build/kspCaches/<source set>/logs/kspDirtySetByOutputs.log`

These logs contain file names of sources and outputs, plus the timestamps of the builds.

多轮次处理

KSP supports *multiple round processing*, or processing files over multiple rounds. It means that subsequent rounds use an output from previous rounds as additional input.

Changes to your processor

To use multiple round processing, the `SymbolProcessor.process()` function needs to return a list of deferred symbols (`List<KSAnnotated>`) for invalid symbols. Use `KSAnnotated.validate()` to filter invalid symbols to be deferred to the next round.

The following sample code shows how to defer invalid symbols by using a validation check:

```
override fun process(resolver: Resolver): List<KSAnnotated> {
    val symbols = resolver.getSymbolsWithAnnotation("com.example.annotation.Builder")
    val result = symbols.filter { !it.validate() }
    symbols
        .filter { it is KSClassDeclaration && it.validate() }
        .map { it.accept(BuilderVisitor(), Unit) }
    return result
}
```

Multiple round behavior

Deferring symbols to the next round

Processors can defer the processing of certain symbols to the next round. When a symbol is deferred, processor is waiting for other processors to provide additional information. It can continue deferring the symbol as many rounds as needed. Once the other processors provide the required information, the processor can then process the deferred symbol. Processor should only defer invalid symbols which are lacking necessary information. Therefore, processors should **not** defer symbols from classpath, KSP will also filter out any deferred symbols that are not from source code.

As an example, a processor that creates a builder for an annotated class might require all parameter types of its constructors to be valid (resolved to a concrete type). In the first round, one of the parameter type is not resolvable. Then in the second round, it becomes resolvable because of the generated files from the first round.

Validating symbols

A convenient way to decide if a symbol should be deferred is through validation. A processor should know which information is necessary to properly process the symbol. Note that validation usually requires resolution which can be expensive, so we recommend checking only what is required. Continuing with the previous example, an ideal validation for the builder processor checks only whether all resolved parameter types of the constructors of annotated symbols contain `isError == false`.

KSP provides a default validation utility. For more information, see the Advanced section.

Termination condition

Multiple round processing terminates when a full round of processing generates no new files. If unprocessed deferred symbols still exist when the termination condition is met, KSP logs an error message for each processor with unprocessed deferred symbols.

Files accessible at each round

Both newly generated files and existing files are accessible through a `Resolver`. KSP provides two APIs for accessing files: `Resolver.getAllFiles()` and `Resolver.getNewFiles()`. `getAllFiles()` returns a combined list of both existing files and newly generated files, while `getNewFiles()` returns only newly generated files.

Changes to getSymbolsAnnotatedWith()

To avoid unnecessary reprocessing of symbols, `getSymbolsAnnotatedWith()` returns only those symbols found in newly generated files, together with the symbols from deferred symbols from the last round.

Processor instantiating

A processor instance is created only once, which means you can store information in the processor object to be used for later rounds.

Information consistent cross rounds

All KSP symbols will not be reusable across multiple rounds, as the resolution result can potentially change based on what was generated in a previous round. However, since KSP does not allow modifying existing code, some information such as the string value for a symbol name should still be reusable. To summarize, processors can store information from previous rounds but need to bear in mind that this information might be invalid in future rounds.

Error and exception handling

When an error (defined by processor calling `KSPLogger.error()`) or exception occurs, processing stops after the current round completes. All processors will call the `onError()` method and will **not** call the `finish()` method.

Note that even though an error has occurred, other processors continue processing normally for that round. This means that error handling occurs after processing has completed for the round.

Upon exceptions, KSP will try to distinguish the exceptions from KSP and exceptions from processors. Exceptions will result in a termination of processing immediately and be logged as an error in KSPLogger. Exceptions from KSP should be reported to KSP developers for further investigation. At the end of the round where exceptions or errors happened, all processors will invoke onError() function to do their own error handling.

KSP provides a default no-op implementation for `onError()` as part of the `SymbolProcessor` interface. You can override this method to provide your own error handling logic.

Advanced

Default behavior for validation

The default validation logic provided by KSP validates all directly reachable symbols inside the enclosing scope of the symbol that is being validated. Default validation checks whether references in the enclosed scope are resolvable to a concrete type but does not recursively dive into the referenced types to perform validation.

Write your own validation logic

Default validation behavior might not be suitable for all cases. You can reference `KSValidateVisitor` and write your own validation logic by providing a custom `predicate` lambda, which is then used by `KSValidateVisitor` to filter out symbols that need to be checked.

KSP 与 Kotlin 多平台

For a quick start, see a sample Kotlin Multiplatform project defining a KSP processor.

Starting from KSP 1.0.1, applying KSP on a multiplatform project is similar to that on a single platform, JVM project. The main difference is that, instead of writing the `ksp(...)` configuration in dependencies, `add(ksp<Target>)` or `add(ksp<SourceSet>)` is used to specify which compilation targets need symbol processing, before compilation.

```
plugins {
    kotlin("multiplatform")
    id("com.google.devtools.ksp")
}

kotlin {
    jvm()
    linuxX64 {
        binaries {
            executable()
        }
    }
}

dependencies {
    add("kspCommonMainMetadata", project(":test-processor"))
    add("kspJvm", project(":test-processor"))
    add("kspJvmTest", project(":test-processor")) // Not doing anything because there'
s no test source set for JVM
    // There is no processing for the Linux x64 main source set, because kspLinuxX64 i
sn't specified
    // add("kspLinuxX64Test", project(":test-processor"))
}
```

Compilation and processing

In a multiplatform project, Kotlin compilation may happen multiple times (`main` , `test` , or other build flavors) for each platform. So is symbol processing. A symbol processing task is created whenever there is a Kotlin compilation task and a corresponding `ksp<Target>` or `ksp<SourceSet>` configuration is specified.

For example, in the above `build.gradle.kts` , there are 4 compilation tasks: common/metadata, JVM main, Linux x64 main, Linux x64 test, and 3 symbol processing tasks: common/metadata, JVM main, Linux x64 test.

Avoid the ksp(...) configuration on KSP 1.0.1+

Before KSP 1.0.1, there is only one, unified `ksp(...)` configuration available. Therefore, processors either applies to all compilation targets, or nothing at all. Note that the `ksp(...)` configuration not only applies to the main source set, but also the test source set if it exists, even on traditional, non-multiplatform projects. This brought unnecessary overheads to build time.

Starting from KSP 1.0.1, per-target configurations are provided as shown in the above example. In the future:

1. For multiplatform projects, the `ksp(...)` configuration will be deprecated and removed.
2. For single platform projects, the `ksp(...)` configuration will only apply to the main, default compilation. Other targets like `test` will need to specify `kspTest(...)` in order to apply processors.

Starting from KSP 1.0.1, there is an early access flag `-DallowAllTargetConfiguration=false` to switch to the more efficient behavior. If the current behavior is causing performance issues, please give it a try. The default value of the flag will be flipped from `true` to `false` on KSP 2.0.

在命令行运行 KSP

KSP is a Kotlin compiler plugin and needs to run with Kotlin compiler. Download and extract them.

```
#!/bin/bash

# Kotlin compiler
wget https://github.com/JetBrains/kotlin/releases/download/v2.1.21/kotlin-compiler-2.1
.21.zip
unzip kotlin-compiler-2.1.21.zip

# KSP
wget https://github.com/google/ksp/releases/download/2.1.21-2.0.1/artifacts.zip
unzip artifacts.zip
```

To run KSP with `kotlinc`, pass the `-Xplugin` option to `kotlinc`.

```
-Xplugin=/path/to/symbol-processing-cmdline-2.1.21-2.0.1.jar
```

This is different from the `symbol-processing-2.1.21-2.0.1.jar`, which is designed to be used with `kotlin-compiler-embeddable` when running with Gradle. The command line `kotlinc` needs `symbol-processing-cmdline-2.1.21-2.0.1.jar`.

You'll also need the API jar.

```
-Xplugin=/path/to/symbol-processing-api-2.1.21-2.0.1.jar
```

See the complete example:

```
#!/bin/bash

KSP_PLUGIN_ID=com.google.devtools.ksp.symbol-processing
KSP_PLUGIN_OPT=plugin:$KSP_PLUGIN_ID

KSP_PLUGIN_JAR=./com/google/devtools/ksp/symbol-processing-cmdline/2.1.21-2.0.1/symbol
-processing-cmdline-2.1.21-2.0.1.jar
KSP_API_JAR=./com/google/devtools/ksp/symbol-processing-api/2.1.21-2.0.1/symbol-proces
sing-api-2.1.21-2.0.1.jar
KOTLINC=./kotlinc/bin/kotlinc

AP=/path/to/your-processor.jar

mkdir out
$KOTLINC \
```

```
        -Xplugin=$KSP_PLUGIN_JAR \
        -Xplugin=$KSP_API_JAR \
        -Xallow-no-source-files \
        -P $KSP_PLUGIN_OPT:apclasspath=$AP \
        -P $KSP_PLUGIN_OPT:projectBaseDir=. \
        -P $KSP_PLUGIN_OPT:classOutputDir=./out \
        -P $KSP_PLUGIN_OPT:javaOutputDir=./out \
        -P $KSP_PLUGIN_OPT:kotlinOutputDir=./out \
        -P $KSP_PLUGIN_OPT:resourceOutputDir=./out \
        -P $KSP_PLUGIN_OPT:kspOutputDir=./out \
        -P $KSP_PLUGIN_OPT:cachesDir=./out \
        -P $KSP_PLUGIN_OPT:incremental=false \
        -P $KSP_PLUGIN_OPT:apoption=key1=value1 \
        -P $KSP_PLUGIN_OPT:apoption=key2=value2 \
        $*
```

KSP FAQ

Why KSP?

KSP has several advantages over kapt:

- It is faster.
- The API is more fluent for Kotlin users.
- It supports multiple round processing on generated Kotlin sources.
- It is being designed with multiplatform compatibility in mind.

Why is KSP faster than kapt?

kapt has to parse and resolve every type reference in order to generate Java stubs, whereas KSP resolves references on-demand. Delegating to javac also takes time.

Additionally, KSP's incremental processing model has a finer granularity than just isolating and aggregating. It finds more opportunities to avoid reprocessing everything. Also, because KSP traces symbol resolutions dynamically, a change in a file is less likely to pollute other files and therefore the set of files to be reprocessed is smaller. This is not possible for kapt because it delegates processing to javac.

Is KSP Kotlin-specific?

KSP can process Java sources as well. The API is unified, meaning that when you parse a Java class and a Kotlin class you get a unified data structure in KSP.

How to upgrade KSP?

KSP has API and implementation. The API rarely changes and is backward compatible: there can be new interfaces, but old interfaces never change. The implementation is tied to a specific compiler version. With the new release, the supported compiler version can change.

Processors only depend on API and therefore are not tied to compiler versions. However, users of processors need to bump KSP version when bumping the compiler version in their project. Otherwise, the following error will occur:

```
ksp-a.b.c is too old for kotlin-x.y.z. Please upgrade ksp or downgrade kotlin-gradle-p
lugin
```

> Users of processors don't need to bump processor's version because processors only depend on API.
>
> 

For example, some processor is released and tested with KSP 1.0.1, which depends strictly on Kotlin 1.6.0. To make it work with Kotlin 1.6.20, the only thing you need to do is bump KSP to a version (for example, KSP 1.1.0) that is built for Kotlin 1.6.20.

Can I use a newer KSP implementation with an older Kotlin compiler?

If the language version is the same, Kotlin compiler is supposed to be backward compatible. Bumping Kotlin compiler should be trivial most of the time. If you need a newer KSP implementation, please upgrade the Kotlin compiler accordingly.

How often do you update KSP?

KSP tries to follow Semantic Versioning as close as possible. With KSP version `major.minor.patch`,

- `major` is reserved for incompatible API changes. There is no pre-determined schedule for this.
- `minor` is reserved for new features. This is going to be updated approximately quarterly.
- `patch` is reserved for bug fixes and new Kotlin releases. It's updated roughly monthly.

Usually a corresponding KSP release is available within a couple of days after a new Kotlin version is released, including the pre-releases (Beta or RC).

Besides Kotlin, are there other version requirements for libraries?

Here is a list of requirements for libraries/infrastructures:

- Android Gradle Plugin 7.1.3+
- Gradle 6.8.3+

What is KSP's future roadmap?

The following items have been planned:

- Support new Kotlin compiler
- Improve support to multiplatform. For example, running KSP on a subset of targets/sharing computations between targets.
- Improve performance. There are a bunch of optimizations to be done!
- Keep fixing bugs.

Please feel free to reach out to us in the #ksp channel in Kotlin Slack (get an invite) if you would like to discuss any ideas. Filing GitHub issues/feature requests or pull requests are also welcome!

学习资料

- 学习资料概述
- 例学 Kotlin↗
- Kotlin 心印
- Kotlin 核心课程↗
- Kotlin 动手实践
- Kotlin 小窍门
- 图书
- Advent of Code 谜题
- 在 IDE 中学习（JetBrains 学院）
  - 学习 Kotlin
  - 讲授 Kotlin
- 库作者指南
  - 简介
  - 最小化心理复杂性
    - 概述
    - 简单
    - 可读性
    - 一致性
    - 可预测性
    - 可调试性
    - 可测试性
  - 反向兼容性
  - 内容翔实的文档
  - 构建多平台库

学习资料概述

You can use the following materials and resources for learning Kotlin:

- Basic syntax – get a quick overview of the Kotlin syntax.
- Idioms – learn how to write idiomatic Kotlin code for popular cases.
    - Java to Kotlin migration guide: Strings – learn how to perform typical tasks with strings in Java and Kotlin.
    - Java to Kotlin migration guide: Collections — learn how to perform typical tasks with collections in Java and Kotlin.
    - Java to Kotlin migration guide: Nullability — learn how to handle nullability in Java and Kotlin.
- Kotlin Koans – complete exercises to learn the Kotlin syntax. Each exercise is created as a failing unit test and your job is to make it pass. Recommended for developers with Java experience.
- Kotlin by example – review a set of small and simple annotated examples for the Kotlin syntax.
- Kotlin Core track by JetBrains Academy – learn all the Kotlin essentials while creating working applications step by step.
- Kotlin books – find books we've reviewed and recommend for learning Kotlin.
- Kotlin tips – watch short videos where the Kotlin team shows you how to use Kotlin in a more efficient and idiomatic way, so you can have more fun when writing code.
- Advent of Code puzzles – learn idiomatic Kotlin and test your language skills by completing short and fun tasks.
- Kotlin hands-on tutorials – complete long-form tutorials to fully grasp a technology. These tutorials guide you through a self-contained project related to a specific topic.
- Kotlin for Java Developers – learn the similarities and differences between Java and Kotlin in this course on Coursera.
- Kotlin documentation in PDF format – read our documentation offline.

例学 Kotlin

例学 Kotlin

Kotlin 心印

Kotlin 心印是一系列主要为 Java 开发人员设计、帮助你熟悉 Kotlin 语法的练习。 每个练习都是不能通过的单元测试，你的任务就是使其通过。 可以通过下列方式中的一种来完成 Kotlin 心印任务：

- 可以使用 Kotlin 心印的在线版。
- 可以通过安装 JetBrains 学院插件并选择 Kotlin Koans 课程，直接在 IntelliJ IDEA 或 Android Studio 中完成任务。

无论选择何种方式来解 Kotlin 心印，都可以看到每项任务的答案：

- 在线版，点击“**Show answer**”（显示答案）。
- 对于 JetBrains 学院插件，请尝试先自行作答，如果答案不正确选择 **Peek solution**（速览答案）。

我们建议你在完成任务后核对答案，以便将你的解答与标准答案进行比较。 请不要作弊哦！

Kotlin 核心课程

Kotlin 核心课程

Kotlin 动手实践

A series of hands-on tutorials where you can create applications with Kotlin using a variety of different technologies and targeting multiple platforms. The exercises are divided into a series of steps, walking you through each section.

Building Reactive Spring Boot applications with Kotlin coroutines and RSocket

Build a simple chat application using Spring Boot and Kotlin, and learn about the benefits of using Kotlin for server-side development from a syntax perspective.

Start

Building web applications with React and Kotlin/JS

Create a React Application using Kotlin/JS, and see how you can leverage Kotlin's type system, library ecosystem, and interoperability features.

Start

Building web applications with Spring Boot and Kotlin

Build a simple blog application by combining the power of Spring Boot and Kotlin.

Start

Creating HTTP APIs with Ktor

Create a backend API for your application that responds to HTTP requests.

Start

Creating a WebSocket chat with Ktor

Create a simple Chat application using Ktor including both a JVM server and a JVM client.

Start

Creating an interactive website with Ktor

Learn how to serve files, use templating engines such as Freemarker and the kotlinx.html DSL, and work with form input from Ktor.

Start

Introduction to Kotlin coroutines and channels

Learn about coroutines in Kotlin and how you can communicate between them using channels.

Start

Introduction to Kotlin/Native

Create a simple HTTP client that can run natively on multiple platforms using Kotlin/Native and libcurl.

Start

Kotlin Multiplatform: networking and data storage

Learn how to create a mobile application for Android and iOS using Kotlin Multiplatform with Ktor and SQLDelight.

Start

Targeting iOS and Android with Kotlin Multiplatform

Learn how to create a mobile application that can target both iOS and Android using Kotlin Multiplatform.

Start

Kotlin 小窍门

Kotlin Tips is a series of short videos where members of the Kotlin team show how to use Kotlin in a more efficient and idiomatic way to have more fun when writing code.

Subscribe to our YouTube channel to not miss new Kotlin Tips videos.

null + null in Kotlin

What happens when you add `null + null` in Kotlin, and what does it return? Sebastian Aigner addresses this mystery in our latest quick tip. Along the way, he also shows why there's no reason to be scared of nullables:

YouTube 视频：Kotlin Tips: null + null in Kotlin

Deduplicating collection items

Got a Kotlin collection that contains duplicates? Need a collection with only unique items? Let Sebastian Aigner show you how to remove duplicates from your lists, or turn them into sets in this Kotlin tip:

YouTube 视频：Kotlin Tips: Deduplicating Collection Items

The suspend and inline mystery

How come functions like `repeat()`, `map()` and `filter()` accept suspending functions in their lambdas, even though their signatures aren't coroutines-aware? In this episode of Kotlin Tips Sebastian Aigner solves the riddle: it has something to do with the inline modifier:

YouTube 视频：Kotlin Tips: The Suspend and Inline Mystery

Unshadowing declarations with their fully qualified name

Shadowing means having two declarations in a scope have the same name. So, how do you pick? In this episode of Kotlin Tips Sebastian Aigner shows you a simple Kotlin trick to call exactly the function that you need, using the power of fully qualified names:

YouTube 视频：Kotlin Tips: Unshadowing Declarations

Return and throw with the Elvis operator

Elvis has entered the building once more! Sebastian Aigner explains why the operator is named after the famous singer, and how you can use `?:` in Kotlin to return or throw. The magic behind the scenes? The Nothing type.

YouTube 视频：Kotlin Tips: Return and Throw with the Elvis Operator

Destructuring declarations

With destructuring declarations in Kotlin, you can create multiple variables from a single object, all at once. In this video Sebastian Aigner shows you a selection of things that can be destructured – pairs, lists, maps, and more. And what about your own objects? Kotlin's component functions provide an answer for those as well:

YouTube 视频：Kotlin Tips: Destructuring Declarations

Operator functions with nullable values

In Kotlin, you can override operators like addition and subtraction for your classes and supply your own logic. But what if you want to allow null values, both on their left and right sides? In this video, Sebastian Aigner answers this question:

YouTube 视频：Kotlin Tips: Operator Functions With Nullable Values

Timing code

Watch Sebastian Aigner give a quick overview of the `measureTimedValue()` function, and learn how you can time your code:

YouTube 视频：Kotlin Tips: Timing Code

Improving loops

In this video, Sebastian Aigner will demonstrate how to improve loops to make your code more readable, understandable, and concise:

YouTube 视频：Kotlin Tips: Improving Loops

Strings

In this episode, Kate Petrova shows three tips to help you work with Strings in Kotlin:

YouTube 视频：Kotlin Tips: Strings

Doing more with the Elvis operator

In this video, Sebastian Aigner will show how to add more logic to the Elvis operator, such as logging to the right part of the operator:

YouTube 视频：Kotlin Tips: The Elvis Operator

Kotlin collections

In this episode, Kate Petrova shows three tips to help you work with Kotlin Collections:

YouTube 视频：Kotlin Tips: Kotlin Collections

下一步做什么？

- See the complete list of Kotlin Tips in our YouTube playlist
- Learn how to write idiomatic Kotlin code for popular cases

Kotlin 图书

More and more authors write books for learning Kotlin in different languages. We are very thankful to all of them and appreciate all their efforts in helping us increase a number of professional Kotlin developers.

Here are just a few books we've reviewed and recommend you for learning Kotlin. You can find more books on our community website.

	Atomic Kotlin is for both beginning and experienced programmers! From Bruce Eckel, author of the multi-award-winning Thinking in C++ and Thinking in Java, and Svetlana Isakova, Kotlin Developer Advocate at JetBrains, comes a book that breaks the language concepts into small, easy-to-digest "atoms", along with a free course consisting of exercises supported by hints and solutions directly inside IntelliJ IDEA!
	Head First Kotlin is a complete introduction to coding in Kotlin. This hands-on book helps you learn the Kotlin language with a unique method that goes beyond syntax and how-to manuals and teaches you how to think like a great Kotlin developer. You'll learn everything from language fundamentals to collections, generics, lambdas, and higher-order functions. Along the way, you'll get to play with both object-oriented and functional programming. If you want to really understand Kotlin, this is the book for you.
	Kotlin in Action teaches you to use the Kotlin language for production-quality applications. Written for experienced Java developers, this example-rich book goes further than most language books, covering interesting topics like building DSLs with natural language syntax. The book is written by Dmitry Jemerov and Svetlana Isakova, developers on the Kotlin team. Chapter 6, covering the Kotlin type system, and chapter 11, covering DSLs, are available as a free preview on the publisher web site.
	Kotlin Programming: The Big Nerd Ranch Guide In this book you will learn to work effectively with the Kotlin language through carefully considered examples designed to teach you Kotlin's elegant style and features. Starting from first principles, you will work your way to advanced usage of Kotlin, empowering you to create programs that are more reliable with less code.



Programming Kotlin is written by Venkat Subramaniam.

Programmers don't just use Kotlin, they love it. Even Google has adopted it as a first-class language for Android development.

With Kotlin, you can intermix imperative, functional, and object-oriented styles of programming and benefit from the approach that's most suitable for the problem at hand.

Learn to use the many features of this highly concise, fluent, elegant, and expressive statically typed language with easy-to-understand examples.

Learn to write maintainable, high-performing JVM and Android applications, create DSLs, program asynchronously, and much more.



The Joy of Kotlin teaches you the right way to code in Kotlin.

In this insight-rich book, you'll master the Kotlin language while exploring coding techniques that will make you a better developer no matter what language you use. Kotlin natively supports a functional style of programming, so seasoned author Pierre-Yves Saumont begins by reviewing the FP principles of immutability, referential transparency, and the separation between functions and effects.

Then, you'll move deeper into using Kotlin in the real world, as you learn to handle errors and data properly, encapsulate shared state mutations, and work with laziness.

This book will change the way you code — and give you back some of the joy you had when you first started.

用地道 Kotlin 代码求解 Advent of Code 谜题

Advent of Code is an annual December event, where holiday-themed puzzles are published every day from December 1 to December 25. With the permission of Eric Wastl, creator of Advent of Code, we'll show how to solve these puzzles using the idiomatic Kotlin style:

- Advent of Code 2024
- Advent of Code 2023
-
-
-

Get ready for Advent of Code

We'll take you through the basic tips on how to get up and running with solving Advent of Code challenges with Kotlin:

- Use this GitHub template to create projects
- Check out the welcome video by Kotlin Developer Advocate, Sebastian Aigner:

YouTube 视频：Get Ready for Advent of Code 2021

Advent of Code 2022

Day 1: Calorie counting

Learn about the Kotlin Advent of Code template and convenience functions for working with strings and collections in Kotlin, such as `maxOf()` and `sumOf()`. See how extension functions can help you structure your solution in a nice manner.

- Read the puzzle description on Advent of Code
- Check out the solution in the video:

Advent of Code 2022 Day 1 | Kotlin

Day 2: Rock paper scissors

Understand operations on the `Char` type in Kotlin, see how the `Pair` type and the `to` constructor work well with pattern matching. Understand how to order your own objects using the `compareTo()` function.

- Read the puzzle description on Advent of Code
- Check out the solution in the video:

Advent of Code 2022 Day 2 | Kotlin

Day 3: Rucksack reorganization

Learn how the kotlinx.benchmark library helps you understand the performance characteristics of your code. See how set operations like `intersect` can help you select overlapping data, and see performance comparisons between different implementations of the same solution.

- Read the puzzle description on Advent of Code
- Check out the solution in the video:

Advent of Code 2022 Day 3 | Kotlin

Day 4: Camp cleanup

See how `infix` and `operator` functions can make your code more expressive, and how extension functions for the `String` and `IntRange` types make it easy to parse input.

- Read the puzzle description on Advent of Code
- Check out the solution in the video:

Advent of Code 2022 Day 4 | Kotlin

Day 5: Supply stacks

Learn about constructing more complex objects with factory functions, how to use regular expressions, and the double-ended `ArrayDeque` type.

- Read the puzzle description on Advent of Code
- Check out the solution in the video:

Advent of Code 2022 Day 5 | Kotlin

Day 6: Tuning trouble

See more in-depth performance investigations with the kotlinx.benchmark library, comparing the characteristics of 16 different variations of the same solution.

- Read the puzzle description on Advent of Code
- Check out the solution in the video:

Advent of Code 2022 Day 6 | Kotlin

Day 7: No space left on device

Learn how to model tree structures, and see a demo of generating Kotlin code programmatically.

- Read the puzzle description on Advent of Code
- Check out the solution in the video:

Advent of Code 2022 Day 7 | Kotlin

Day 8: Treetop tree house

See the `sequence` builder in action, and how far a first draft of a program and an idiomatic Kotlin solution can differ (with special guest Roman Elizarov!).

- Read the puzzle description on Advent of Code
- Check out the solution in the video:

Advent of Code 2022 Day 8 | Kotlin

Day 9: Rope bridge

See the `run` function, labeled returns, and convenient standard library functions like `coerceIn`, or `zipWithNext`. See how you can construct lists of given sizes using the `List` and `MutableList` constructors, and get a peek at a Kotlin-based visualization of the problem statement.

- Read the puzzle description on Advent of Code
- Check out the solution in the video:

Advent of Code 2022 Day 9 | Kotlin

Day 10: Cathode-ray tube

Learn how ranges and the `in` operator make checking ranges natural, how function parameters can be turned into receivers, and a brief exploration of the `tailrec` modifier.

- Read the puzzle description on Advent of Code
- Check out the solution in the video:

Advent of Code 2022 Day 10 | Kotlin

Day 11: Monkey in the middle

See how you can move from mutable, imperative code to a more functional approach that makes use of immutable and read-only data structures. Learn about context receivers and how our guest built his own visualization library just for Advent of Code.

- Read the puzzle description on Advent of Code
- Check out the solution in the video:

Advent of Code 2022 Day 11 | Kotlin

Day 12: Hill Climbing algorithm

Use queues, `ArrayDeque`, function references, and the `tailrec` modifier to solve path finding problems with Kotlin.

- Read the puzzle description on Advent of Code
- Check out the solution in the video:

Advent of Code 2022 Day 12 | Kotlin

Advent of Code 2021

> Read our blog post about Advent of Code 2021

Day 1: Sonar sweep

Apply windowed and count functions to work with pairs and triplets of integers.

- Read the puzzle description on Advent of Code
- Check out the solution from Anton Arhipov on the Kotlin Blog or watch the video:

Advent of Code 2021 in Kotlin, Day 1: Sonar Sweep

Day 2: Dive!

Learn about destructuring declarations and the `when` expression.

- Read the puzzle description on Advent of Code
- Check out the solution from Pasha Finkelshteyn on GitHub or watch the video:

 Advent of Code 2021 in Kotlin, Day 2: Dive!

Day 3: Binary diagnostic

Explore different ways to work with binary numbers.

- Read the puzzle description on Advent of Code
- Check out the solution from Sebastian Aigner on Kotlin Blog or watch the video:

 Advent of Code 2021 in Kotlin, Day 3: Binary Diagnostic

Day 4: Giant squid

Learn how to parse the input and introduce some domain classes for more convenient processing.

- Read the puzzle description on Advent of Code
- Check out the solution from Anton Arhipov on the GitHub or watch the video:

 Advent of Code 2021 in Kotlin, Day 4: Giant Squid

Advent of Code 2020

> You can find all the solutions for the Advent of Code 2020 puzzles in our GitHub repository.
>
> 

Day 1: Report repair

Explore input handling, iterating over a list, different ways of building a map, and using the `let` function to simplify your code.

- Read the puzzle description on Advent of Code
- Check out the solution from Svetlana Isakova on the Kotlin Blog or watch the video:

 Learn Kotlin With the Kotlin Team: Advent of Code 2020 #1

Day 2: Password philosophy

Explore string utility functions, regular expressions, operations on collections, and how the `let` function can be helpful to transform your expressions.

- Read the puzzle description on Advent of Code
- Check out the solution from Svetlana Isakova on the Kotlin Blog or watch the video:

Learn Kotlin with The Kotlin Team: Advent of Code 2020 #2

Day 3: Toboggan trajectory

Compare imperative and more functional code styles, work with pairs and the `reduce()` function, edit code in the column selection mode, and fix integer overflows.

- Read the puzzle description on Advent of Code
- Check out the solution from Mikhail Dvorkin on GitHub or watch the video:

Learn Kotlin with the Kotlin Team: Advent of Code 2020 #3

Day 4: Passport processing

Apply the `when` expression and explore different ways of how to validate the input: utility functions, working with ranges, checking set membership, and matching a particular regular expression.

- Read the puzzle description on Advent of Code
- Check out the solution from Sebastian Aigner on the Kotlin Blog or watch the video:

Learn Kotlin with the Kotlin Team: Advent of Code 2020 #4

Day 5: Binary boarding

Use the Kotlin standard library functions (`replace()` , `toInt()` , `find()`) to work with the binary representation of numbers, explore powerful local functions, and learn how to use the `max()` function in Kotlin 1.5.

- Read the puzzle description on Advent of Code
- Check out the solution from Svetlana Isakova on the Kotlin Blog or watch the video:

Learn Kotlin with the Kotlin Team: Advent of Code 2020 #5

Day 6: Custom customs

Learn how to group and count characters in strings and collections using the standard library functions: `map()` , `reduce()` , `sumOf()` , `intersect()` , and `union()` .

- Read the puzzle description on Advent of Code
- Check out the solution from Anton Arhipov on the Kotlin Blog or watch the video:

Learn Kotlin with the Kotlin Team: Advent of Code 2020 #6

Day 7: Handy haversacks

Learn how to use regular expressions, use Java's `compute()` method for HashMaps from Kotlin for dynamic calculations of the value in the map, use the `forEachLine()` function to read files, and compare two types of search algorithms: depth-first and breadth-first.

- Read the puzzle description on Advent of Code
- Check out the solution from Pasha Finkelshteyn on the Kotlin Blog or watch the video:

Learn Kotlin with the Kotlin Team: Advent of Code 2020 #7

Day 8: Handheld halting

Apply sealed classes and lambdas to represent instructions, apply Kotlin sets to discover loops in the program execution, use sequences and the `sequence { }` builder function to construct a lazy collection, and try the experimental `measureTimedValue()` function to check performance metrics.

- Read the puzzle description on Advent of Code
- Check out the solution from Sebastian Aigner on the Kotlin Blog or watch the video:

Learn Kotlin with the Kotlin Team: Advent of Code 2020 #8

Day 9: Encoding error

Explore different ways to manipulate lists in Kotlin using the `any()`, `firstOrNull()`, `firstNotNullOfOrNull()`, `windowed()`, `takeIf()`, and `scan()` functions, which exemplify an idiomatic Kotlin style.

- Read the puzzle description on Advent of Code
- Check out the solution from Svetlana Isakova on the Kotlin Blog or watch the video:

Learn Kotlin with the Kotlin Team: Advent of Code 2020 #9

下一步做什么？

- Complete more tasks with Kotlin Koans
- Create working applications with the free Kotlin Core track by JetBrains Academy

在 IDE 中学习（JetBrains 学院）

- 学习 Kotlin
- 讲授 Kotlin

通过 JetBrains 学院插件学习 Kotlin

Android Studio 与 IntelliJ IDEA 都提供了 JetBrains 学院插件， 可以通过代码练习任务来学习 Kotlin。

看一看《初学者入门指南（英文）》， 可以帮助开始使用 IntelliJ IDEA 中的 Kotlin 心印课程。 解决交互式编码难题，并在 IDE 内部获得即时反馈。

如果要使用 JetBrains 学院插件进行教学，请阅读通过 JetBrains 学院插件讲授 Kotlin。

通过 JetBrains 学院插件讲授 Kotlin

使用 Android Studio 与 IntelliJ IDEA 中都可用的 JetBrains 学院插件， 可以通过代码练习任务来讲授 Kotlin。

看一看《教育者入门指南（英文）》， 了解如何创建一个简单的 Kotlin 课程，其中包括一组编程任务与综合测试。

如果想使用 JetBrains 学院插件学习 Kotlin，请阅读 通过 JetBrains 学院插件学习 Kotlin。

库作者指南

- 简介
- 最小化心理复杂性
  - 概述
  - 简单
  - 可读性
  - 一致性
  - 可预测性
  - 可调试性
  - 可测试性
- 反向兼容性
- 内容翔实的文档
- 构建多平台库

库作者指南简介

This guide contains a summary of best practices and ideas to consider when designing libraries.

To be effective, a library must achieve certain fundamental objectives. Specifically, it should:

- Define its problem domain and implement a set of related functional requirements that solve the problems it defined. For example, an HTTP client may aim to support all HTTP request types and understand the various headers, content types, and status codes.
- Meet the non-functional criteria appropriate to the problem domain. These typically involve performance, reliability, security, and usability. The relative importance of these criteria varies widely. For example, a library designed for batch processing may not require the same level of performance as one intended for day trading.

  > The process of identifying and defining functional and non-functional requirements is a complex topic that has been extensively studied in software engineering. This guide does not cover these topics in depth, as they are beyond its scope.
  >
  

The main focus of this guide is to explore the characteristics that a library must have to stay relevant and popular with its users. These characteristics include:

- **Minimize mental complexity:** All developers must consider the readability and maintainability of their code. It's crucial to reduce the mental effort required for others to read, understand, and use your APIs. Achieving this involves creating libraries that are clear, consistent, predictable, and easy to debug.
- **Backward compatibility:** When releasing a new version of an API, ensure the existing API remains operational. Clearly communicate and document any breaking changes well in advance. Provide a straightforward, clear, and gradual pathway for users to adopt the new API or design changes.
- **Informative documentation:** The documentation that accompanies the library needs to do more than repeat function and type declarations. It should be comprehensive and specifically tailored to the library's audience. It should accurately reflect the needs and scenarios of various user roles, ensuring it provides essential information without being overly simplistic or complex. Always include clear examples, balancing explanatory text with practical code samples.

Additionally, **building your Kotlin library with multiplatform support** can broaden its applicability across projects targeting various environments. Designing APIs to work reliably in both shared and platform-specific code can improve the library’s versatility and usability across all supported targets.

The following sections dig deeper into these characteristics, with practical advice on how you can provide the best possible experience to users of your libraries.

What's next

- Explore strategies for minimizing mental complexity in Minimizing mental complexity.
- Learn about maintaining backward compatibility in Backward compatibility.
- For an extensive overview on effective documentation practices, see Informative documentation.
- Discover best practices for building multiplatform libraries in Building Kotlin library for multiplatform.

最小化心理复杂性

- 概述
- 简单
- 可读性
- 一致性
- 可预测性
- 可调试性
- 可测试性

最小化心理复杂性概述

Users need to quickly and accurately build a mental model of your library's functions and abstractions before using it. The best way to achieve this is by minimizing the amount of complexity they encounter.

Strategies for minimizing mental complexity include:

- **Simplicity:** Strive for an API that provides the most functionality with the fewest components, reusing existing Kotlin types and structures to avoid redundancy. Where possible, create a small set of core abstractions and build additional functionality on top of them.
- **Readability:** Write the API in a declarative style to make the code's intent clear. Choose names for abstractions directly from the problem domain, unless it's absolutely necessary to invent new ones. Use basic data types for their intended purposes. Clearly distinguish between core and optional functionality.
- **Consistency:** Maintain a single, clear approach for every design aspect of your API. Use uniform naming conventions, error handling strategies, and patterns, whether they are object-oriented or functional.
- **Predictability:** Design your library to adhere to the ‘principle of least surprise'. Ensure the default settings match the most common use cases, allowing users to accomplish their tasks with the simplest and shortest code. Allow extensions to your library only in clearly specified ways to maintain consistency and predictability.
- **Debuggability:** Ensure your library aids users in troubleshooting by facilitating the extraction of information and navigation through nested function calls. When exceptions are thrown, both the type and content of the exception should match the underlying issue, providing all necessary details to effectively diagnose and resolve problems. It should be possible to capture and output the state of domain objects, and to view any intermediate representations.
- **Testability:** Ensure that your library, as well as the code that uses it, can be easily tested.

The following sections give more detailed information on implementing these strategies in Kotlin.

Next step

To begin exploring these strategies in depth, you can start with learning about simplicity in the next section.

Proceed to the next part

简单

The fewer concepts your users need to understand and the more explicitly these are communicated, the simpler their mental model is likely to be. This can be achieved by limiting the number of operations and abstractions in the API.

Ensure that the visibility of declarations in your library is set appropriately to keep internal implementation details out of the public API. Only APIs that are explicitly designed and documented for public use should be accessible to users.

In the next part of the guide, we'll discuss some guidelines for promoting simplicity.

Use explicit API mode

We recommend using the explicit API mode feature of the Kotlin compiler, which forces you to explicitly state your intentions when you're designing the API for your library.

With explicit API mode, you must:

- Add visibility modifiers to your declarations to make them public, instead of relying on the default public visibility. This ensures that you've considered what you're exposing as part of the public API.
- Define the types for all your public functions and properties to prevent unintended changes to your API from inferred types.

Reuse existing concepts

One way to limit the size of your API is to reuse existing types. For example, instead of creating a new type for durations, you can use `kotlin.time.Duration`. This approach not only streamlines development but also improves interoperability with other libraries.

Be careful when relying on types from third-party libraries or platform-specific types, as they can tie your library to these elements. In such cases, the costs may outweigh the benefits.

Reusing common types such as `String`, `Long`, `Pair`, and `Triple` can be effective, but this should not stop you from developing abstract data types if they better encapsulate domain-specific logic.

Define and build on top of core API

Another route to simplicity is to define a small conceptual model based around a limited set of core operations. Once the behavior of these operations is clearly documented, you can expand the API by developing new operations that build directly on or combine these core functions.

For example:

- In the Kotlin Flows API, common operations like `filter` and `map` are built on top of the `transform` operation.
- In the Kotlin Time API the `measureTime` function utilizes `TimeSource.Monotonic`.

While it's often beneficial to base additional operations on these core components, it's not always necessary. You may find opportunities to introduce optimized or platform-specific variations that expand the functionality or adapt more broadly to different inputs.

As long as users are able to solve non-trivial problems with the core operations and can refactor their solutions with additional operations without altering any behavior, the simplicity of the conceptual model is preserved.

Next step

In the next part of the guide, you'll learn about readability.

Proceed to the next part

可读性

Creating a readable API involves more than just writing clean code. It requires thoughtful design that simplifies integration and usage. This section explores how you can enhance API readability by structuring your library with composability in mind, utilizing domain-specific languages (DSLs) for concise and expressive setup, and using extension functions and properties for clear and maintainable code.

Prefer Explicit Composability

Libraries often provide advanced operators that allow for customization. For example, an operation might permit users to supply their own data structures, networking channels, timers, or lifecycle observers. However, introducing these customization options through additional function parameters can significantly increase the complexity of the API.

Instead of adding more parameters for customization, it's more effective to design an API where different behaviors can be composed together. For example, in the coroutine Flows API both buffering and conflation are implemented as separate functions. These can be chained together with more basic operations like `filter` and `map`, instead of each basic operation accepting parameters to control buffering and conflation.

Another example involves the Modifiers API in Jetpack Compose. This allows Composable components to accept a single `Modifier` parameter that handles common customization options, such as padding, sizing, and background color. This approach avoids the need for each Composable to accept separate parameters for these customizations, streamlining the API and reducing complexity.

```
Box(
    modifier = Modifier
        .padding(10.dp)
        .onClick { println("Box clicked!") }
        .fillMaxWidth()
        .fillMaxHeight()
        .verticalScroll(rememberScrollState())
        .horizontalScroll(rememberScrollState())
) {
    // Box content goes here
}
```

Use DSLs

A Kotlin library can significantly improve readability by providing a builder DSL. Using a DSL allows you to concisely repeat domain-specific data declarations. For example, consider the following sample from a Ktor-based server application:

```
fun Application.module() {
    install(ContentNegotiation) {
        json(Json {
            prettyPrint = true
            isLenient = true
        })
    }
    routing {
        post("/article") {
            call.respond<String>(HttpStatusCode.Created, ...)
        }
        get("/article/list") {
            call.respond<List<CreateArticle>>(...)
        }
        get("/article/{id}") {
            call.respond<Article>(...)
        }
    }
}
```

This sets up an application, installing the `ContentNegotiation` plugin configured to use Json serialization, and sets up routing so that the application responds to requests on various `/article` endpoints.

For a detailed description of creating DSLs, see Type-safe builders. The following points are worth noting in the context of creating libraries:

- The functions used in the DSL are builder functions, which take a lambda with receiver as the final parameter. This design allows these functions to be called without parentheses, making the syntax clearer. The lambda being passed can be used to configure the entity being created. In the example above the lambda passed to the `routing` function is used to configure the details of the routing.
- Factory functions that create instances of classes should have the same name as the return type and start with a capital letter. You can see this in the sample above with the creation of the `Json` instance. These functions may still take lambda parameters for configuration. For more information, see Coding conventions.
- As it's not possible to ensure that required properties have been set within the lambda supplied to a builder function at compile time, we recommend passing required values as function parameters.

Using DSLs to build objects not only improves readability but also improves backward compatibility, and simplifies the documentation process. For example, take the following function:

```
fun Json(prettyPrint: Boolean, isLenient: Boolean): Json
```

This function could replace the `Json{}` DSL builder. However, the DSL approach has noticeable benefits:

- Backward compatibility is easier to maintain with the DSL builder than with this function, as adding new configuration options simply means adding new properties (or in other examples, new functions), which is a backward-compatible change, unlike changing the parameter list of an existing function.
- It also makes creating and maintaining documentation easier. You can document each property separately at its point of declaration, instead of having to document many parameters of a function, all in one place.

Use extension functions and properties

We recommend using extension functions and properties to improve readability.

Classes and interfaces should define the core concept of a type. Additional functionality and information should be written as extension functions and properties. This makes it clear to the reader that the additional functionality can be implemented on top of the core concept, and additional information can be calculated from the data in the type.

For example, the `CharSequence` type (which `String` also implements) only contains the most basic information and operators to access its contents:

```
interface CharSequence {
    val length: Int
    operator fun get(index: Int): Char
    fun subSequence(startIndex: Int, endIndex: Int): CharSequence
}
```

Functionality commonly associated with strings is mostly defined as extension functions, which can all be implemented on top of the core concepts and basic APIs of the type:

```
inline fun CharSequence.isEmpty(): Boolean = length == 0
inline fun CharSequence.isNotEmpty(): Boolean = length > 0

inline fun CharSequence.trimStart(predicate: (Char) -> Boolean): CharSequence {
    for (index in this.indices)
        if (!predicate(this[index]))
            return subSequence(index, length)
    return ""
}
```

Consider declaring computed properties and normal methods as extensions. Only regular properties, overrides, and overloaded operators should be declared as members by default.

Avoid using the boolean type as an argument

Consider the following function:

```
fun doWork(optimizeForSpeed: Boolean) { ... }
```

If you were to provide this function in your API it could be invoked as:

```
doWork(true)
doWork(optimizeForSpeed=true)
```

In the first call it is impossible to infer what the boolean argument is for, unless you are reading the code in an IDE with Parameter Name Hints enabled. Using named arguments does clarify the intention, but there is no way to force your users to adopt this style. Consequently, to improve readability, your code should not use boolean types as arguments.

Alternatively, the API could create a separate function specifically for the task controlled by the boolean argument. This function should have a descriptive name that indicates what it does.

For example, the following extensions are available on the `Iterable` interface:

```
fun <T, R> Iterable<T>.map(transform: (T) -> R): List<R>
fun <T, R : Any> Iterable<T>.mapNotNull(
    transform: (T) -> R?
): List<R>
```

Instead of the single method:

```
fun <T, R> Iterable<T>.map(
    includeNullResults: Boolean = true,
    transform: (T) -> R
): List<R>
```

Another good approach could be to use an `enum` class to define different operation modes. This approach is useful if there are several modes of operation, or if you expect these modes to change over time.

Use numeric types appropriately

Kotlin defines a set of numeric types that you may use as part of your API. Here's how to use them appropriately:

- Use the `Int`, `Long` and `Double` types as arithmetic types. They represent values with which calculations are performed.
- Avoid using arithmetic types for non-arithmetic entities. For example, if you represent an ID as a `Long`, your users might be tempted to compare IDs, on the assumption they are assigned in order. This could lead to unreliable or meaningless results, or create dependencies on implementations that could change without warning. A better strategy is to define a specialized class for the ID abstraction. You could use Inline value classes to build such abstractions without affecting performance. See the `Duration` class for an example.
- The `Byte`, `Float` and `Short` types are memory layout types. They are used to restrict the amount of memory available for storing a value, such as in caches or when transmitting data over a network. These types should only be used when the underlying data reliably fits within that type, and calculations are not required.
- The unsigned integer types `UByte`, `UShort`, `UInt` and `ULong` should be used to utilize the full range of positive values available in a given format. They are suitable for scenarios requiring values beyond the range of signed types or for interoperability with native libraries. However, avoid using them in situations where the domain only requires non-negative integers.

Next step

In the next part of the guide, you'll learn about consistency.

Proceed to the next part

一致性

Consistency is crucial in API design to ensure ease of use. By maintaining consistent parameter order, naming conventions, and error handling mechanisms, your library will be more intuitive and reliable for users. Following these best practices helps avoid confusion and misuse, leading to a better developer experience and more robust applications.

Preserve parameter order, naming and usage

When designing a library, maintain consistency in the ordering of arguments, the naming scheme, and the use of overloading. For example, if one of your existing methods has `offset` and `length` parameters, you should not switch to alternatives like `startIndex` and `endIndex` for a new method unless there is a compelling reason.

Overloaded functions provided by the library should behave identically. Users expect the behavior to remain consistent when they change the type of a value they pass into your library. For example, these calls all create identical instances, as the input is semantically the same:

```
BigDecimal(200)
BigDecimal(200L)
BigDecimal("200")
```

Avoid mixing parameter names like `startIndex` and `stopIndex` with synonyms like `beginIndex` and `endIndex`. Similarly, choose one term for values in collections, such as `element`, `item`, `entry`, or `entity`, and stick with it.

Name related methods consistently and predictably. As an example, the Kotlin standard library contains pairs like `first` and `firstOrNull`, `single` or `singleOrNull`. These pairs clearly indicate that some might return `null` while others might throw an exception. Parameters should be declared from the general to the specific, so essential inputs appear first and optional inputs last. For example, in `CharSequence.findAnyOf` the `strings` collection goes first, followed by the `startIndex`, and finally the `ignoreCase` flag.

Consider a library managing employee records, and provides the following API to search for employees:

```
fun findStaffBySeniority(
    startIndex: Int,
    minYearsServiceExclusive: Int
): List<Employee>
```

```
fun findStaffByAge(
    minAgeInclusive: Int,
    startIndex: Int
): List<Employee>
```

This API would be extremely hard to use correctly. There are multiple parameters of the same type presented in an inconsistent order, and used in an inconsistent way. Users of your library will likely make incorrect assumptions about new functions based on their experience with existing ones.

Use Object-Oriented design for data and state

Kotlin supports both the Object-Oriented and Functional programming styles. Use classes to represent data and state in your API. When the data and state is hierarchical, consider using inheritance.

If all the state required can be passed as parameters, prefer using top-level functions. When calls to these functions will be chained, consider writing them as extension functions to improve readability.

Choose the appropriate error handling mechanism

Kotlin provides several mechanisms for error handling. Your API can throw an exception, return a `null` value, use a custom result type, or use the built-in `Result` type. Ensure that your library uses these options consistently and appropriately.

When data cannot be fetched or calculated, use a nullable return type and return `null` to indicate missing data. In other cases, throw an exception or return a `Result` type.

Consider providing overloads of functions, where one throws an exception, while the other wraps it in a result type instead. In these cases, use the `Catching` suffix to indicate that exceptions are caught in the function. For example, the standard library has the `run` and `runCatching` functions using this convention, and the coroutines library has `receive` and `receiveCatching` methods for channels.

Avoid using exceptions for normal control flow. Design your API to allow for condition checks before attempting operations, thus preventing unnecessary error handling. Command / Query Separation is a useful pattern that can be applied here.

Maintain conventions and quality

The final aspect of consistency relates, not to the design of the library itself, but to maintaining a high level of quality.

You should use automated tools (linters) for static analysis to ensure your code follows both general Kotlin conventions and project-specific conventions.

A Kotlin library should also provide a suite of unit and integration tests covering all documented behaviors of all the API entry points. Tests should include a wide range of inputs, especially known boundary and edge cases. Any untested behavior should be assumed to be (at best) unreliable.

Use this suite of tests during development to verify that changes do not break existing behavior. Run these tests on every release as part of a standardized build and release pipeline. Tools like Kover can be integrated into your build process to measure coverage and generate reports.

Next step

In the next part of the guide, you'll learn about predictability.

Proceed to the next part

可预测性

To design a robust and user-friendly Kotlin library, it's essential to anticipate common use cases, allow for extensibility, and enforce proper usage. Following best practices for default settings, error handling, and state management ensures a seamless experience for users while maintaining the integrity and quality of the library.

Do the right thing by default

Your library should anticipate the "happy path" for each use case, and provide default settings accordingly. Users should not need to supply default values for the library to function correctly.

For example, when using the Ktor `HttpClient` the most common use case is sending a GET request to the server. This can be accomplished using the code below, where only essential information needs to be specified:

```
val client = HttpClient(CIO)
val response: HttpResponse = client.get("https://ktor.io/")
```

It's not necessary to provide values for mandatory HTTP headers or custom event handlers for possible status codes in the response.

If there is no obvious "happy path" for a use case or if a parameter should have a default value but there is no non-contentious option, it likely indicates a flaw in the requirement analysis.

Allow opportunities for extension

When the correct choice cannot be anticipated, allow users to specify their preferred approach. Your library should also let the user supply their own approach or use a third-party extension.

For example, with the Ktor `HttpClient`, users are encouraged to install support for content negotiation when configuring the client, and to specify their preferred serialization formats:

```
val client = HttpClient(CIO) {
    install(ContentNegotiation) {
        json(Json {
            prettyPrint = true
            isLenient = true
        })
    }
```

```
}
```

Users can choose which plugins to install or create their own using the separate API for defining client plugins.

Additionally, users can define extension functions and properties for types in the library. As a library author, you can make this easier by designing with extensions in mind, and ensuring your library's types have clear core concepts.

Prevent unwanted and invalid extensions

Users should not be able to extend your library in ways that violate its original design or are impossible within the rules of the problem domain.

For example, when marshaling data to and from JSON, only six types are supported in the output format: `object`, `array`, `number`, `string`, `boolean`, and `null`.

If you create an open class or interface called `JsonElement`, users could create invalid derived types, like `JsonDate`. Instead, you can make the `JsonElement` interface sealed and provide an implementation for each type:

```
sealed interface JsonElement

class JsonNumber(val value: Number) : JsonElement
class JsonObject(val values: Map<String, JsonElement>) : JsonElement
class JsonArray(val values: List<JsonElement>) : JsonElement
class JsonBoolean(val value: Boolean) : JsonElement
class JsonString(val value: String) : JsonElement
object JsonNull : JsonElement
```

Sealed types also enable the compiler to ensure your `when` expressions are exhaustive, without requiring an `else` statement, improving readability and consistency.

Avoid exposing mutable state

When managing multiple values, your API should, whenever possible, accept and/or return read-only collections. Mutable collections are not thread-safe and introduce complexity and unpredictability into your library.

For example, if a user modifies a mutable collection returned from an API entry point, it will be unclear whether they are modifying the implementation's structure or a copy. Similarly, if users can modify the values within a collection after passing it to a library, it will be unclear whether this affects

the implementation.

Since arrays are mutable collections, avoid using them in your API. If arrays must be used, make defensive copies before sharing data with users. This ensures your data structures remain unmodified.

This policy of making defensive copies is automatically performed by the compiler for `vararg` arguments. When using the spread operator to pass an existing array where a `vararg` argument is expected, a copy of your array is automatically created.

This behavior is demonstrated in the following example:

```
fun main() {
    fun demo(vararg input: String): Array<out String> = input

    val originalArray = arrayOf("one", "two", "three", "four")
    val newArray = demo(*originalArray)

    originalArray[1] = "ten"

    //prints "one, ten, three, four"
    println(originalArray.joinToString())

    //prints "one, two, three, four"
    println(newArray.joinToString())
}
```

Validate inputs and state

Ensure that your library is used correctly by validating inputs and existing state before implementation proceeds. Use the `require` function to verify inputs and the `check` function to validate existing state.

The `require` function throws an `IllegalArgumentException` if its condition is `false`, causing the function to fail immediately with an appropriate error message:

```
fun saveUser(username: String, password: String) {
    require(username.isNotBlank()) { "Username should not be blank" }
    require(username.all { it.isLetterOrDigit() }) {
        "Username can only contain letters and digits, was: $username"
    }
    require(password.isNotBlank()) { "Password should not be blank" }
    require(password.length >= 7) {
        "Password must contain at least 7 characters"
    }

    /* Implementation can proceed */
}
```

Error messages should include relevant inputs to help users determine the cause of the failure, as shown above by the error message for usernames that contain invalid characters, which includes the incorrect username. An exception to this practice is when including a value in the error message could reveal information that might be used maliciously as part of a security exploit, which is why the error message for the password's length does not include the password input.

Similarly, the `check` function throws an `IllegalStateException` if its condition is `false`. Use this function to verify the state of an instance, as shown in the example below:

```
class ShoppingCart {
    private val contents = mutableListOf<Item>()

    fun addItem(item: Item) {
       contents.add(item)
    }

    fun purchase(): Amount {
       check(contents.isNotEmpty()) {
           "Cannot purchase an empty cart"
       }
       // Calculate and return amount
    }
}
```

Next step

In the next part of the guide, you'll learn about debuggability.

Proceed to the next part

可调试性

Users of your library will build on its functionality, and the features they build will contain errors that need to be identified and resolved. This error resolution process might be conducted within a debugger during development or using logging and observability tools in production. Your library can follow these best practices to make debugging it easier.

Provide a toString method for stateful types

For every type that contains state, provide a meaningful `toString` implementation. This implementation should return an intelligible representation of the instance's current content, even for internal types.

Since `toString` representations of types are often written to logs, consider security when implementing this method and avoid returning sensitive user data.

Ensure the format used to describe the state is as consistent as possible across the different types in your library. This format should be explicitly described and thoroughly documented when it is part of a contract implemented by your API. The output from your `toString` methods may support parsing, for example in automated test suites.

For example, consider the following types from a library supporting service subscriptions:

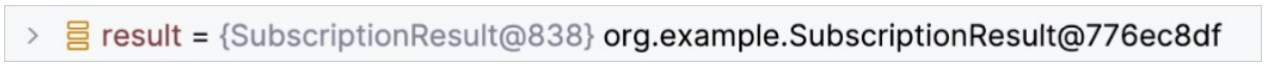
```
enum class SubscriptionResultReason {
    Success, InsufficientFunds, IncompatibleAccount
}

class SubscriptionResult(
    val result: Boolean,
    val reason: SubscriptionResultReason,
    val description: String
)
```

Without a `toString` method, printing a `SubscriptionResult` instance is not very useful:

```
fun main() {
    val result = SubscriptionResult(
       false,
       IncompatibleAccount,
       "Users account does not support this type of subscription"
    )
```

```
    //prints 'org.example.SubscriptionResult@13221655'
    println(result)
}
```

Nor is the information readily displayed in the debugger:



Adding a simple `toString` implementation improves the output significantly in both cases:

```
//prints 'Subscription failed (reason=IncompatibleAccount, description="Users
// account does not support this type of subscription")'
override fun toString(): String {
    val resultText = if(result) "succeeded" else "failed"
    return "Subscription $resultText (reason=$reason, description=\"$description\")"
}
```



While it might be tempting to use data classes to gain a `toString` method automatically, it's not recommended for backward compatibility reasons. Data classes are discussed in more detail in the Avoid using data classes in your API section.

Note that the state described in the `toString` method does not need to be information from the problem domain. It can relate to the status of ongoing requests (as in the example above), the health of connections to external services, or intermediate state within an ongoing operation.

For example, consider the following builder type:

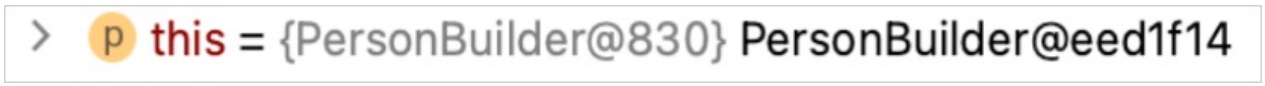
```
class Person(
    val name: String?,
    val age: Int?,
    val children: List<Person>
) {
    override fun toString(): String =
        "Person(name=$name, age=$age, children=$children)"
}

class PersonBuilder {
    var name: String? = null
    var age: Int? = null
    val children = arrayListOf<Person>()


    fun child(personBuilder: PersonBuilder.() -> Unit = {}) {
       children.add(person(personBuilder))
    }
    fun build(): Person = Person(name, age, children)
}
```

```
fun person(personBuilder: PersonBuilder.() -> Unit = {}): Person =
    PersonBuilder().apply(personBuilder).build()
```

This is how you would use this type:



If you halt the code at the breakpoint displayed on the image above, the information displayed will not be helpful:



Adding a simple `toString` implementation results in a much more helpful output:

```
override fun toString(): String =
    "PersonBuilder(name=$name, age=$age, children=$children)"
```

With this addition, the debugger shows:



This way, you can immediately see which fields are set and which are not.

Adopt and document a policy for handling exceptions

As discussed in the Choose appropriate error handling mechanism section, there are occasions when it's appropriate for your library to throw an exception to signal an error. You may create your own exception types for this purpose.

Libraries that abstract and simplify low-level APIs will also need to handle exceptions thrown by their dependencies. A library might choose to suppress the exception, pass it on as it is, convert it to a different type of an exception, or signal the error to users in a different way.

Any of these options could be valid, depending on the context. For example:

- If a user adopts library A purely for the convenience of simplifying library B, it may be appropriate for library A to rethrow any exceptions generated by library B without modification.
- If library A adopts library B purely as an internal implementation detail, then library-specific exceptions thrown by library B should never be exposed to users of library A.

You must adopt and document a consistent approach to exception handling so users can make productive use of your library. This is especially important for debugging. Users of your library should be able to recognise, in the debugger and in logs, when an exception has originated from your library.

The type of the exception should indicate the type of the error, and the data in the exception should help the user locate the root cause of the issue. A common pattern is to wrap a low-level exception in a library-specific one, with the original exception accessible as the `cause`.

Next step

In the next part of the guide, you'll learn about testability.

Proceed to the next part

可测试性

In addition to testing your library, ensure that code using your library is also testable.

Avoid global state and stateful top-level functions

Your library should not rely on state in global variables or provide stateful top-level functions as part of its public API. Such variables and functions make testing code that uses the library difficult, as tests need to find ways to control these global values.

For example, a library might define a globally accessible function that provides access to the current time:

```
val instant: Instant = Clock.now()
println(instant)
```

Any code using this API will be difficult to test, as the call to the `now()` function will always return the real current time, while in tests it's often desirable to return fake values instead.

To enable testability, the `kotlinx-datetime` library has an API that lets users get a `Clock` instance, and then use that to get the current time:

```
val clock: Clock = Clock.System
val instant: Instant = clock.now()
println(instant)
```

This allows users of a library to inject a `Clock` instance into their own classes, and replace the real implementation with a fake one during tests.

What's next

If you haven't already, consider checking out these pages:

- Learn about maintaining backward compatibility in the Backward compatibility page.
- For an extensive overview on effective documentation practices, see Informative Documentation.

面向库作者的反向兼容性指南

The most common motivation for creating a library is to expose functionality to a wider community. This community might be a single team, a company, a particular industry, or a technology platform. In every case backward compatibility will be an important consideration. The wider the community the more important backward compatibility becomes, since you will be less aware of who your users are and what constraints they work within.

Backward compatibility is not a single term, but can be defined at the binary, source and behavioral levels. More information about these types is provided in this section.

Note that:

- It is possible to break binary compatibility without breaking source compatibility, as well as the other way around.
- It is desirable but very difficult to guarantee source compatibility. As a library author, you must consider every possible way a function or type could be invoked or instantiated by a user of the library. Source compatibility is typically an aspiration, not a promise.

The rest of this section describes actions you can take, and tools you can use to help ensure the different kinds of compatibility.

Compatibility types

Binary compatibility means that a new version of a library can replace a previously compiled version of the library. Any software that was compiled against the previous version of the library should continue to work correctly.

> Learn more about binary compatibility in the Binary compatibility validator's README or from the Evolving Java-based APIs document.
>
> 

Source compatibility means that a new version of a library can replace a previous one without modifying any of the source code that uses the library. However, the outputs from compiling this client code may no longer be compatible with the outputs from compiling the library, so the client code must be rebuilt against the new version of the library to guarantee compatibility.

Behavioral compatibility means that a new version of the library does not modify the existing functionality, except to fix bugs. The same features are involved and they have the same semantics.

Use the Binary compatibility validator

JetBrains provides a Binary compatibility validator tool, which can be used to ensure binary compatibility across different versions of your API.

This tool is implemented as a Gradle plugin, and it adds two tasks to your build:

- The `apiDump` task creates a human-readable `.api` file that describes your API.
- The `apiCheck` task compares the saved description of the API to classes compiled in the current build.

The `apiCheck` task is invoked at build time by the standard Gradle `check` task. When compatibility is broken, the build fails. At that point, you should run the `apiDump` task manually and compare the differences between the old and new versions. If you are satisfied with the changes, you can update the existing `.api` file, which resides within your VCS.

The validator has experimental support for validating KLibs produced by multiplatform libraries.

Specify return types explicitly

As discussed in the Kotlin coding guidelines, you should always explicitly specify function return types and property types within the API. See also the section about Explicit API mode.

Consider the following example, where the library author creates a `JsonDeserializer` and, for convenience, uses an extension function to associate it with the `Int` type:

```
class JsonDeserializer<T>(private val fromJson: (String) -> T) {
    fun deserialize(input: String): T {
        ...
    }
}

fun Int.defaultDeserializer() = JsonDeserializer { ... }
```

Let's say the author replaces this implementation with a `JsonOrXmlDeserializer`:

```
class JsonOrXmlDeserializer<T>(
    private val fromJson: (String) -> T,
    private val fromXML: (String) -> T
) {
    fun deserialize(input: String): T {
        ...
    }
}
```

```
fun Int.defaultDeserializer() = JsonOrXmlDeserializer({ ... }, { ... })
```

Existing functionality will continue to work, with the added ability to deserialize XML. However, this breaks binary compatibility.

Avoid adding arguments to existing API functions

Adding non-default arguments to a public API breaks both binary and source compatibility, as users are required to provide more information on an invocation than before. However, even adding default arguments can break compatibility.

For example, imagine you have the following function in `lib.kt`:

```
fun fib() = … // Returns zero
```

And the following function in `client.kt`:

```
fun main() {
    println(fib()) // Prints zero
}
```

Compiling these two files on the JVM would produce the outputs `LibKt.class` and `ClientKt.class`.

Let's say you reimplement and compile the `fib` function to represent the Fibonacci sequence, such that `fib(3)` returns 2, `fib(4)` returns 3, and so on. You add a parameter but give it a default value of zero to preserve the existing behavior:

```
fun fib(input: Int = 0) = … // Returns Fibonacci member
```

You now need to recompile the file `lib.kt`. You might expect that the `client.kt` file does not need to be recompiled, and the associated class file can be invoked as follows:

```
$ kotlin ClientKt.class
```

But if you try this, a `NoSuchMethodError` occurs:

```
Exception in thread "main" java.lang.NoSuchMethodError: 'int LibKt.fib()'
       at LibKt.main(fib.kt:2)
       at LibKt.main(fib.kt)
       …
```

This is because the signature of the method changed in the bytecode generated by the Kotlin/JVM compiler, breaking binary compatibility.

Source compatibility, however, is preserved. If you recompile both files, the program will run as before.

Use overloads to preserve compatibility

When writing Kotlin code for the JVM, you can use the `@JvmOverloads` annotation on functions with default arguments. This generates overloads of the function, one for each parameter with a default argument that may be omitted from the end of the parameter list. With these individual generated functions, adding a new parameter to the end of the parameter list preserves binary compatibility, as it doesn't change any existing functions in the output, just adds a new one.

For example, the above function might be annotated like this:

```
@JvmOverloads
fun fib(input: Int = 0) = …
```

This would generate two methods in the output bytecode, one with no parameter and one with an `Int` parameter:

```
public final static fib()I
public final static fib(I)I
```

For all Kotlin targets, you may choose to manually create several overloads of your function instead of a single function that accepts default arguments to preserve binary compatibility. In the example above, this means creating a separate `fib` function for the case where you wish to take an `Int` parameter:

```
fun fib() = …
fun fib(input: Int) = …
```

Avoid widening or narrowing return types

When evolving an API, it is common to want to widen or narrow the return type of a function. For example, in an upcoming version of your API, you might want to switch a return type from `List` to `Collection` or from `Collection` to `List`.

You might want to narrow the type to `List` to meet user requests for indexing support. Conversely, you might want to widen the type to `Collection` because you realize the data you are working with has no natural order.

It is easy to see why widening a return type breaks compatibility. For example, converting from `List` to `Collection` breaks all the code that uses indexing.

You might think that narrowing a return type, for example from `Collection` to `List` would preserve compatibility. Unfortunately, while source compatibility is preserved, binary compatibility is broken.

Let's say you have a demo function in the file `Library.kt`:

```
public fun demo(): Number = 3
```

And a client for the function in `Client.kt`:

```
fun main() {
    println(demo()) // Prints 3
}
```

Let's imagine a scenario where you change the return type of demo and only recompile `Library.kt`:

```
fun demo(): Int = 3
```

When you rerun the client, the following error will occur (on the JVM):

```
Exception in thread "main" java.lang.NoSuchMethodError: 'java.lang.Number Library.demo
()'
        at ClientKt.main(call.kt:2)
        at ClientKt.main(call.kt)
        …
```

This happens because of the following instruction in the bytecode generated from the `main` method:

```
0: invokestatic  #12 // Method Library.demo:()Ljava/lang/Number;
```

The JVM is trying to invoke a static method called demo which returns a `Number`. However, as this method no longer exists, you have broken binary compatibility.

Avoid using data classes in your API

In regular development, the strength of data classes is the extra functions that are generated for you. In API design, this strength becomes a weakness.

For example, let's say you use the following data class in your API:

```
data class User(
    val name: String,
    val email: String
)
```

Later, you might want to add a property called `active`:

```
data class User(
    val name: String,
    val email: String,
    val active: Boolean = true
)
```

This would break binary compatibility in two ways. Firstly, the generated constructor will have a different signature. Additionally, the signature of the generated `copy` method changes.

The original signature (on Kotlin/JVM) would be:

```
public final User copy(java.lang.String, java.lang.String)
```

After adding the `active` property, the signature becomes:

```
public final User copy(java.lang.String, java.lang.String, boolean)
```

As with the constructor, this breaks binary compatibility.

It's possible to work around these issues by manually writing a secondary constructor and overriding the `copy` method. However, the effort involved negates the convenience of using a data class.

Another issue with data classes is that changing the order of constructor arguments affects the generated `componentX` methods, which are used for destructuring. Even if it does not break binary compatibility, changing the order will definitely break behavioral compatibility.

Considerations for using the PublishedApi annotation

Kotlin allows inline functions to be a part of your library's API. Calls to these functions will be inlined into the client code written by your users. This can introduce compatibility issues, so these functions are not allowed to call non-public-API declarations.

If you need to call an internal API of your library from an inlined public function, you can do so by annotating it with `@PublishedApi`. This makes the internal declaration effectively public, as references to it will end up in compiled client code. Therefore, it must be treated the same as public declarations when making changes to it, as these changes might affect binary compatibility.

Evolve APIs pragmatically

There are cases where you need to make breaking changes to your library's API over time by removing or changing an existing declaration. In this section, we'll discuss how to handle such cases pragmatically.

When users upgrade to a newer version of your library, they should not end up with unresolved references to your library's APIs in their project's source code. Instead of immediately removing something from your library's public API, you should follow a deprecation cycle. This way, you give your users time to migrate to an alternative.

Use the `@Deprecated` annotation on the old declaration to indicate that it's being replaced. The parameters of this annotation provide important details about the deprecation:

- The `message` should explain what's being changed and why.
- The `replaceWith` parameter should be used where possible to provide automatic migration to a new API.
- The deprecation's level should be used to deprecate the API gradually. For more information, see the Deprecated page of the Kotlin documentation.

Generally, a deprecation should first produce a warning, then an error, and then hide the declaration. This process should occur across several minor releases, giving users time to make any required changes in their projects. Breaking changes, such as removing an API, should happen only in major releases. A library may adopt different versioning and deprecation strategies, but this must be communicated to its users to set the correct expectations.

You can learn more in the Kotlin Evolution principles document or in the Evolving your Kotlin API painlessly for clients talk by Leonid Startsev from KotlinConf 2023.

Use the RequiresOptIn mechanism

The Kotlin standard library provides the opt-in mechanism to require explicit consent from users before they use a part of your API. This is based on creating marker annotations, which are themselves annotated with `@RequiresOptIn`. You should use this mechanism to manage expectations concerning source and behavioral compatibility, especially when introducing new APIs to your library.

If you choose to use this mechanism, we recommend following these best practices:

- Use the opt-in mechanism to provide different guarantees to different parts of the API. For example, you could mark features as *Preview*, *Experimental*, and *Delicate*. Each category should be clearly explained in your documentation and in KDoc comments, with appropriate warning messages.
- If your library uses an experimental API, propagate the annotation to your own users. This ensures your users are aware that you have dependencies which are still evolving.
- Avoid using the opt-in mechanism to deprecate already existing declarations in your library. Use `@Deprecated` instead, as described in the Evolve APIs pragmatically section.

What's next

If you haven't already, consider checking out these pages:

- Explore strategies for minimizing mental complexity in the Minimizing mental complexity page.
- For an extensive overview on effective documentation practices, see Informative documentation.

库作者创建内容翔实的文档的最佳实践

The documentation you provide for your library is crucial. It can determine whether users investigate your library, adopt it within their projects, and persevere when they encounter difficulties. Developers today have unprecedented choice between languages, libraries, frameworks and platforms. Therefore, it is essential to engage and inform your users; otherwise they may pursue other options.

In the earliest versions of your library, feedback from users will be scarce. Fortunately, creating and refining documentation can serve as a feedback loop to greatly increase the quality of your project. As such, creating documentation should never be seen as a burden, and should not be pushed down the list of priorities when creating a library.

Effective documentation not only informs users but also drives the development and refinement of your library. Here are several key ways documentation can guide your development process:

- You should be able to explain, in a couple of paragraphs, what your library does, who will benefit from using it, and what the advantages are over alternative approaches. If you cannot do this, reconsider the scope and objectives of your project.
- You should be able to create a "Getting Started" guide that can get a potential user up and running as quickly as possible. What counts as *quickly* will depend on the problem domain, but you can compare against similar libraries on other platforms. The guide should hook the user into a feedback loop that keeps getting easier and faster while always producing reliable results. Creating this guide will help you identify sudden increases in complexity (cliff edges) that could hinder the user's progress.
- The act of documenting a function forces you to consider all the edge cases, such as valid ranges of inputs, exceptions that might be thrown, and how performance degrades as the work increases. This can often lead to improvements in the function signatures and the underlying implementation.
- If the code required to initialize your library always eclipses the code required to accomplish a task, rethink your configuration options.
- If you cannot create clear examples of performing basic tasks with standard options, consider optimizing your API for day-to-day use.
- If you cannot demonstrate how to test your library without using real data sources and online services, consider providing test doubles for components that access the network (and the outside world in general).

The sooner you provide documentation for your library, the sooner it can be tested by real-world users. Feedback from these tests can then be used to improve the design.

Provide comprehensive documentation

Your library must provide sufficient documentation to let users adopt it with minimum effort. This documentation should include:

- A Getting Started guide
- An in-depth description of the API
- Longer examples for common use cases (also known as recipes)
- Links to resources such as blogs, articles, webinars, and conference talks

The Getting Started guide should cover how your library integrates with the supported build systems. It should include a brief description of the most commonly used entities, with small examples of how they are used. At each point where the library interacts with the outside world, specify the necessary steps to configure the environment and how to verify that these steps have been completed successfully. State explicitly if no steps are necessary.

If possible, provide separate versions of the documentation for each version of the library you support. This prevents users from viewing information that is either outdated or too recent. When this is not possible, clearly mark sections of the documentation that relate to parts of the API that have been redesigned.

Create personas for your users

Creating and evaluating documentation without a clear understanding of the intended audience is challenging. Defining multiple personas for the types of users who will read your documentation can be helpful.

Consider the user's constraints, such as the pre-existing software stack within which they need to operate. Reviewers of the documentation can adopt these personas to make their conclusions more meaningful.

When you lack specific information about your users, it's best to be pessimistic. For example, do not assume expertise in the latest or most advanced features of Kotlin. Keep your code examples as simple as possible.

Personas are especially useful when real-world users cannot be consulted due to time, budgetary constraints, or confidentiality agreements. Over time, as you gain a better understanding of your users, refine the personas to match their needs more accurately.

Document by example whenever possible

Documentation by example is one of the most cost-effective ways to explain basic concepts to users. Whenever possible, provide simple and clear code examples that help to explain or demonstrate the current topic or concept being discussed.

The KDoc documentation format lets you use inline markup using Markdown in your documentation comments. Use inline code snippets in comments to showcase the usage of an API. For an example, see the source code and rendered documentation of the test dispatchers of the coroutines library.

Providing examples like this can avoid the need to write lengthy descriptions of expected inputs, possible outputs, and failure modes. However, the context for each example needs to be clear, as do the circumstances in which it is relevant. Simply providing a folder of uncommented sample programs does not qualify as documentation.

Thoroughly document your API

Every supported API entry point should be documented using KDoc.

Kotlin's documentation engine, Dokka, includes only public declarations in its outputs by default. As discussed in the Simplicity section, you should minimize your public API and remove public entry points that you don't want users to access. If there are APIs you cannot hide from users by controlling their visibility, omit them from the documentation using the suppress directive.

Begin the description of entry points with a clear, high-level description of what the function does. Avoid simply restating the signature in natural language.

For example, don't say “takes a `String` and returns a `Connection` ” but instead perhaps “Attempts to connect to the database specified by the input string, returning a `Connection` if successful, and throwing a `ConnectionTimeoutException` otherwise”.

Specify each input's expected values and behavior with different inputs. Explain the range of valid values and what happens when invalid values are provided. For example, if a string input is supposed to be a URL, describe what happens when the string is empty, invalid, uses an unsupported protocol, or refers to a location that does not exist.

Document every exception an API entry point may throw. Discuss failure conditions in the general description and reserve the exception section for detailed information. This enhances readability and helps the reader focus. Instead, include this information organically into the general description. Providing usage examples whenever possible also helps users understand how to use the API correctly.

> We recommend learning about technical writing to improve the clarity and effectiveness of your documentation. Consider exploring resources, such as this course from Google (part one and part two).



Document lambda parameters

When an API entry point takes a lambda, the user is providing some functionality that your library will execute on their behalf. This requires extra documentation in at least two areas.

First, document what happens if the lambda throws an exception. Consider addressing the following questions:

- Will this cause an immediate failure, will the lambda be invoked repeatedly, or is there a fallback behavior?
- If the calling function needs to exit, will it rethrow the exception thrown from the lambda or a different one?
- If the exception is different, will it include the original?

Additionally, unless the function is declared as inline,document any special behavior relating to concurrency. Ensure the following are covered:

- Will the lambda be invoked in the same thread as the caller?
- If the lambda is not invoked in the same thread as the caller, which thread (or thread pool) will it be invoked in?
- Could multiple copies of the lambda be run in parallel?
- Which other jobs of work may be using that thread?
- Can the user specify a thread for the library to use?
- Where multiple lambdas are being invoked, what guarantees are provided regarding sequencing?

Use explicit links in documentation

It is very rare for an API entry point to be completely independent of other functionality in your library. Typically, calls must be made in a specific sequence, there are multiple options for performing a particular task, and entry points that perform related tasks are used in similar ways. For example, functions like `format` and `parse` mirror each other.

Use the `@see` tag or internal links to make these relationships explicit in your documentation. This helps the reader by enabling them to chunk) the information together, building a better-integrated mental map of the library.

Be self-contained where possible

When describing what makes an input valid, it is tempting to simply refer to the relevant standard, such as those produced by the W3C, IEEE, or Unicode Consortium. While providing these links can be helpful, the reader should not be forced to refer to an external specification to discover basic information, like the set of whitespace characters.

Wherever possible, the documentation should be self-contained. It should provide enough information for the user to understand the typical use of each API entry point. You can refer the user to external documents for edge cases that will not happen in common use.

Use simple English

It's important to use simple and clear English when creating documentation. This ensures that your content is accessible to a global audience, including those whose first language is not English. Avoid using complex words, jargon, Latin phrases, or idiomatic expressions that might confuse readers. Instead, use straightforward language and concise sentences.

Simple English also makes the documentation easier to translate if needed. Clear, unambiguous text reduces the risk of misinterpretation and improves the overall readability.

What's next

If you haven't already, consider checking out these pages:

- Explore strategies for minimizing mental complexity in the Minimizing mental complexity page.
- Learn about maintaining backward compatibility in the Backward compatibility page.

构建适用于多平台的 Kotlin 库

When creating a Kotlin library, consider building and publishing it with support for Kotlin Multiplatform. This broadens the target audience of your library, making it compatible with projects targeting multiple platforms.

The following sections provide guidelines to help you build a Kotlin Multiplatform library effectively.

Maximize your reach

To make your library available to the largest number of projects as a dependency, aim to support as many target platforms of Kotlin Multiplatform as possible.

If your library doesn't support the platforms used by a multiplatform project, whether it's a library or an application, it becomes difficult for that project to depend on your library. In that case, projects can use your library for some platforms and need to implement separate solutions for others, or they will choose an alternative library altogether that supports all their platforms.

To streamline artifact production, you can try Experimental cross-compilation to publish Kotlin Multiplatform libraries from any host. This allows you to generate `.klib` artifacts for Apple targets without an Apple machine. We plan to stabilize this feature and further improve library publication in the future. Please leave your feedback about this feature in our issue tracker YouTrack.

> For Kotlin/Native targets, consider using a tiered approach to support all possible targets.
>
> 

Design APIs for use from common code

When creating a library, design the APIs to be usable from common Kotlin code, instead of writing platform-specific implementations.

Provide reasonable default configurations when possible and include platform-specific configuration options. Good defaults allow users to use the library's APIs from common Kotlin code, without the need to write platform-specific implementations to configure the library.

Place APIs in the broadest relevant source set using the following priority:

- **The `commonMain` source set:** APIs in the `commonMain` source set are available to all platforms the library supports. Aim to place most of your library's API here.
- **Intermediate source sets:** If some platforms don't support certain APIs, use intermediate source

sets to target specific platforms. For example, you can create a `concurrent` source set for targets that support multi-threading or a `nonJvm` source set for all non-JVM targets.

- **Platform-specific source sets:** For platform-specific APIs, use source sets like `androidMain`.

> To learn more about the source sets of Kotlin Multiplatform projects, see Hierarchical project structure.
>
> 

Ensure consistent behavior across platforms

To ensure your library behaves consistently on all supported platforms, APIs in a multiplatform library should accept the same range of valid inputs, perform the same actions, and return the same results across all platforms. Similarly, the library should treat invalid inputs uniformly and report errors or throw exceptions consistently on all platforms.

Inconsistent behavior makes the library difficult to use and forces users to add conditional logic in common code to manage platform-specific differences.

You can use `expect` and `actual` declarations to declare functions in common code with platform-specific implementations that have full access to the native APIs of each platform. These implementations must also have the same behavior to ensure that they can be used reliably from common code.

When APIs behave consistently across platforms, they only need to be documented once in the `commonMain` source set.

> If platform differences are unavoidable, such as when one platform supports a broader set of inputs, minimize them where possible. For example, you might not want to limit one platform's functionality to match the others. In such cases, clearly document the specific differences.
>
> {style=”note”}

Test on all platforms

Multiplatform libraries can have multiplatform tests written in common code that runs on all platforms. Executing this common test suite regularly on your supported platforms can ensure that the library behaves correctly and consistently.

Regularly testing Kotlin/Native targets across all published platforms can be challenging. However, to ensure broader compatibility, consider publishing the library for all the targets it can support, using a tiered method when testing compatibility.

Use the `kotlin-test` library to write tests in common code and execute them with platform-specific test runners.

Consider non-Kotlin users

Kotlin Multiplatform offers interoperability with native APIs and languages across its supported target platforms. When creating a Kotlin Multiplatform library, consider whether users might need to use your library's types and declarations from languages other than Kotlin.

For example, if some types from your library will be exposed to Swift code through interoperability, design those types to be easily accessible from Swift. The Kotlin-Swift interopedia provides useful insights into how Kotlin APIs appear when called from Swift.

Promote your library

Your library can be featured on the JetBrains' search platform. It's designed to make it easy to look for Kotlin Multiplatform libraries based on their target platforms.

Libraries that meet the criteria are added automatically. For more information on how to add your library, see FAQ.

早期访问计划（EAP）

- 参与 Kotlin 早期访问计划
- 配置构建采用 EAP

参与 Kotlin 早期访问计划

Latest Kotlin EAP release: **2.2.0-RC3**

You can participate in the Kotlin Early Access Preview (EAP) to try out the latest Kotlin features before they are released.

We ship a few Beta (*Beta*) and Release Candidate (*RC*) builds before every language (*2.x.0*) and tooling (*2.x.20*) release.

We'll be very thankful if you find and report bugs to our issue tracker YouTrack. It is very likely that we'll be able to fix them before the final release, which means you won't need to wait until the next Kotlin release for your issues to be addressed.

By participating in the Early Access Preview and reporting bugs, you contribute to Kotlin and help us make it better for everyone in the growing Kotlin community. We appreciate your help a lot!

If you have any questions and want to participate in discussions, you are welcome to join the #eap channel in Kotlin Slack. In this channel, you can also get notifications about new EAP builds.

Configure your project for the Kotlin EAP version

> By participating in the EAP, you expressly acknowledge that the EAP version may not be reliable, may not work as intended, and may contain errors.
>
> Please note that we don't provide any guarantees of compatibility between EAP and final versions of the same release.
>
> 

How the EAP can help you be more productive with Kotlin

- **Prepare for the Stable release**. If you work on a complex multimodule project, participating in the EAP may streamline your experience when you adopt the Stable release version. The sooner you update to the Stable version, the sooner you can take advantage of its performance improvements and new language features.

  The migration of huge and complex projects might take a while, not only because of their size, but also because some specific use cases may not have been covered by the Kotlin team yet. By participating in the EAP and continuously testing new versions of Kotlin, you can provide us

with early feedback about your specific use cases. This will help us address as many issues as possible and ensure you can safely update to the Stable version when it's released. Check out how Slack benefits from testing Android, Kotlin, and Gradle pre-release versions.

- **Keep your library up-to-date**. If you're a library author, updating to the new Kotlin version is extremely important. Using older versions could block your users from updating Kotlin in their projects. Working with EAP versions allows you to support the latest Kotlin versions in your library almost immediately with the Stable release, which makes your users happier and your library more popular.
- **Share the experience**. If you're a Kotlin enthusiast and enjoy contributing to the Kotlin ecosystem by creating educational content, trying new features in the Kotlin EAP allows you to be among the first to share the experience of using the new cool features with the community.

Build details

Build info	Build highlights
2.2.0-RC3 Released: **June 16, 2025** Release on GitHub	A language release with previews of new features and tooling updates. For more details, please refer to the changelog or What's new in Kotlin 2.2.0-RC3.

配置构建采用 EAP

Latest Kotlin EAP release: **2.2.0-RC3**

Explore Kotlin EAP release details

To configure your build to use the EAP version of Kotlin, you need to:

- Specify the EAP version of Kotlin. Available EAP versions are listed here.
- Change the versions of dependencies to EAP ones. The EAP version of Kotlin may not work with the libraries of the previously released version.

The following procedures describe how to configure your build in Gradle and Maven:

- Configure in Gradle
- Configure in Maven

Configure in Gradle

This section describes how you can:

- Adjust the Kotlin version
- Adjust versions in dependencies

Adjust the Kotlin version

In the `plugins` block within `build.gradle(.kts)`, change the `KOTLIN-EAP-VERSION` to the actual EAP version, such as `2.2.0-RC3`. Available EAP versions are listed here.

Alternatively, you can specify the EAP version in the `pluginManagement` block in `settings.gradle(.kts)` – see Gradle documentation for details.

Here is an example for the Multiplatform project.

> 【Kotlin】

```
plugins {
    java
    kotlin("multiplatform") version "KOTLIN-EAP-VERSION"
}

repositories {
```

```
    mavenCentral()
}
```

【Groovy】

```
plugins {
    id 'java'
    id 'org.jetbrains.kotlin.multiplatform' version 'KOTLIN-EAP-VERSION'
}

repositories {
    mavenCentral()
}
```

Adjust versions in dependencies

If you use kotlinx libraries in your project, your versions of the libraries may not be compatible with the EAP version of Kotlin.

To resolve this issue, you need to specify the version of a compatible library in dependencies. For a list of compatible libraries, see EAP build details.

> In most cases we create libraries only for the first EAP version of a specific release and these libraries work with the subsequent EAP versions for this release.
>
> If there are incompatible changes in next EAP versions, we release a new version of the library.

Here is an example.

For the **kotlinx.coroutines** library, add the version number – `1.10.2` – that is compatible with `2.2.0-RC3`.

【Kotlin】

```
dependencies {
    implementation("org.jetbrains.kotlinx:kotlinx-coroutines-core:1.10.2")
}
```

> 【Groovy】

```
dependencies {
    implementation "org.jetbrains.kotlinx:kotlinx-coroutines-core:1.10.2"
}
```

Configure in Maven

In the sample Maven project definition, replace `KOTLIN-EAP-VERSION` with the actual version, such as `2.2.0-RC3`. Available EAP versions are listed here.

```
<project ...>
    <properties>
        <kotlin.version>KOTLIN-EAP-VERSION</kotlin.version>
    </properties>

    <repositories>
        <repository>
           <id>mavenCentral</id>
           <url>https://repo1.maven.org/maven2/</url>
        </repository>
    </repositories>

    <pluginRepositories>
       <pluginRepository>
          <id>mavenCentral</id>
          <url>https://repo1.maven.org/maven2/</url>
       </pluginRepository>
    </pluginRepositories>

    <dependencies>
        <dependency>
            <groupId>org.jetbrains.kotlin</groupId>
            <artifactId>kotlin-stdlib</artifactId>
            <version>${kotlin.version}</version>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.jetbrains.kotlin</groupId>
                <artifactId>kotlin-maven-plugin</artifactId>
                <version>${kotlin.version}</version>
                ...
            </plugin>
        </plugins>
```

```
    </build>
</project>
```

If you run into any problems

- Report an issue to our issue tracker, YouTrack.
- Find help in the #eap channel in Kotlin Slack (get an invite).
- Roll back to the latest stable version: change it in your build script file.

其他资源

- FAQ
- Kotlin 兼容性指南
  - Kotlin 2.1 的兼容性指南
  - Kotlin 2.0 的兼容性指南
  - Kotlin 1.9 的兼容性指南
  - Kotlin 1.8 的兼容性指南
  - Kotlin 1.7.20 的兼容性指南
  - Kotlin 1.7 的兼容性指南
  - Kotlin 1.6 的兼容性指南
  - Kotlin 1.5 的兼容性指南
  - Kotlin 1.4 的兼容性指南
  - Kotlin 1.3 的兼容性指南
  - 兼容模式
- Kotlin 基金会
  - Kotlin 基金会↗
  - 语言委员会准则↗
  - 提交不兼容变更指南↗
  - Kotlin 品牌用途准则↗
  - Kotlin 基金会 FAQ↗
- Google Summer of Code
  - 概述
  - Google Summer of Code with Kotlin 2025↗
  - Google Summer of Code with Kotlin 2024↗
  - Google Summer of Code with Kotlin 2023↗
- 安全
- 社区
  - 贡献力量
  - KUG 准则
  - Kotlin 之夜准则
  - Kotlin Slack 的行为准则与指南
- Kotlin 品牌素材
- 宣传资料↗

FAQ

Kotlin 是什么？

Kotlin 是一门面向 JVM、Android、JavaScript、Wasm 以及原生平台的开源静态类型编程语言。 它是由 JetBrains 开发的。该项目开始于 2010 年并且很早就已开源。 第一个官方 1.0 版发布于 2016 年 2 月。

Kotlin 的当前版本是多少？

目前发布的版本是 2.1.21，发布于 2025-05-13。

可以在 GitHub 上找到更多信息。

Kotlin 是免费的吗？

是。Kotlin 是免费的，已经免费并会保持免费。它是遵循 Apache 2.0 许可证开发的，其源代码可以在 GitHub 上获得。

Kotlin 是面向对象还是函数式语言？

Kotlin 既具有面向对象又具有函数式结构。你既可以按 OO 风格也可以按 FP 风格使用，还可以混合使用两种风格。 通过对诸如高阶函数、函数类型和 lambda 表达式等功能的一等支持，Kotlin 是一个很好的选择， 如果你正在进行或探索函数式编程的话。

Kotlin 能给我超出 Java 语言的哪些优点？

Kotlin 更简洁。粗略估计显示，代码行数减少约 40%。 它也更安全——例如对不可空类型的支持使应用程序不易发生 NPE。 其他功能包括智能类型转换、高阶函数、扩展函数和带接收者的 lambda 表达式，提供了编写富于表现力的代码的能力以及易于创建 DSL 的能力。

Kotlin 与 Java 语言兼容吗？

兼容。Kotlin 与 Java 语言可以 100% 互操作，并且主要强调确保你现有的代码库可以与Kotlin 正确交互。你可以轻松地在 Java 中调用 Kotlin 代码以及在 Kotlin 中调用 Java 代码。 这使得采用 Kotlin 更容易、风险更低。内置于 IDE 的自动化 Java 到 Kotlin 转换器可简化现有代码的迁移。

我可以用 Kotlin 做什么？

Kotlin 可用于任何类型的开发，无论是服务器端、客户端 Web、 Android 还是多平台库。 随着原生 Kotlin（Kotlin/Native）平台目前的进展，还支持其他平台，如嵌入式系统、macOS 与 iOS。 人们将 Kotlin 用于移动端与服务器端应用程序、使用 JavaScript 或 JavaFX的客户端、以及数据科学， 仅举这几例。

我可以用 Kotlin 进行 Android 开发吗？

可以。Kotlin 已作为 Android 平台的一等语言而支持。已经有数百种应用程序在使用 Kotlin 用于 Android 开发，比如 Basecamp、Pinterest 等等。更多信息请查看 Android 开发资源。

我可以用 Kotlin 进行服务器端开发吗？

可以。Kotlin 与 JVM 100% 兼容，因此你可以使用任何现有的框架，如 Spring Boot、 vert.x 或 JSF。另外还有一些 Kotlin 写的特定框架，例如 Ktor。 更多信息请查看服务器端开发资源。

我可以用 Kotlin 进行 web 开发吗？

可以。For backend web development, Kotlin works well with frameworks such as Ktor and Spring, enabling you to build server-side applications efficiently. 此外，你还可以使用 Kotlin/Wasm 用于客户端 Web 开发。 Learn how to get started with Kotlin/Wasm.

我可以用 Kotlin 进行桌面开发吗？

可以。你可以使用任何 Java UI 框架如 JavaFx、Swing 或其他框架。 另外还有 Kotlin 特定框架，如 TornadoFX。

我可以用 Kotlin 进行原生开发吗？

可以。Kotlin/Native 是 Kotlin 的一部分。它将 Kotlin 编译成无需虚拟机（VM）即可运行的原生代码。 可以在主流的桌面与移动端平台甚至某些物联网（IoT）设备上试用。 更多详细信息请查阅 Kotlin/Native 文档。

哪些 IDE 支持 Kotlin？

Kotlin 在 IntelliJ IDEA、 Android Studio 中都有完整的开箱即用支持（内置 JetBrains 开发的官方 Kotlin 插件）。

其他 IDE 与代码编辑器只有 Kotlin 社区支持的插件。

还可以尝试使用 Kotlin Playground 在浏览器中编写、运行及共享 Kotlin 代码。

此外，还提供了命令行编译器，为编译与运行应用程序提供了直接的支持。

哪些构建工具支持 Kotlin？

在 JVM 端，主要构建工具包括 Gradle、Maven、 Ant 和 Kobalt。还有一些可用于构建客户端 JavaScript 的构建工具。

Kotlin 会编译成什么？

当面向JVM 平台时，Kotlin 生成 Java 兼容的字节码。

当面向 JavaScript 时，Kotlin 会转译到 ES5.1，并生成与包括 AMD 和 CommonJS 在内的模块系统相兼容的代码。

当面向原生平台时，Kotlin 会（通过 LLVM）生成平台相关的代码。

Kotlin 面向哪些版本的 JVM？

Kotlin 会让你选择用于执行的 JVM 版本。默认情况下，Kotlin/JVM 编译器会生成兼容 Java 8 的字节码。 如果要利用 Java 新版本中提供的优化功能，可以将目标 Java 版本显式指定为 9 到 23。 请注意，这种情况下生成的字节码可能无法在较低版本中运行。 自 Kotlin 1.5起，编译器不支持生成兼容低于 Java 8 版本的字节码。

Kotlin 难吗？

Kotlin 是受 Java、C#、JavaScript、Scala 以及 Groovy 等现有语言的启发。我们已经努力确保 Kotlin 易于学习， 所以人们可以在几天之内轻松转向、阅读和编写 Kotlin。 学习惯用的 Kotlin 和使用更多它的高级功能可能需要一点时间，但总体来说这不是一个复杂的语言。 如需了解更多信息，请查阅我们的学习资料。

哪些公司使用 Kotlin？

有太多使用 Kotlin 的公司可列，而有些更明显的公司已经公开宣布使用 Kotlin，分别通过博文、Github 版本库或者演讲宣布，包括 Square、 Pinterest、 Basecamp 以及 Corda。

谁开发 Kotlin？

Kotlin 由 JetBrains 的一个工程师团队开发（目前团队规模为 100+）。 其首席语言设计师是 Michail Zarečenskij。除了核心团队，GitHub 上还有 250 多个外部贡献者。

在哪里可以了解关于 Kotlin 更多？

最好的起始地方好是本网站（原文是英文官网）。从那里你可以下载编译器、 To start with Kotlin, you can install one of the official IDEs or try it online.

有没有关于 Kotlin 的图书？

有许多关于 Kotlin 的图书。其中一些我们已经审阅过并且可以推荐作为入门图书。 这些都列在了图书页。 如需了解更多图书，请参见 kotlin.link 上社区维护的列表。

Kotlin 有没有在线课程？

可以在学习 JetBrains Academy 的 Kotlin 核心课程创建可工作应用程序的同时学习所有 Kotlin 基础知识。

可以选择的一些其他课程：

- Pluralsight Course: Getting Started with Kotlin，作者：Kevin Jones
- O'Reilly Course: Introduction to Kotlin Programming，作者：Hadi Hariri
- Udemy Course: 10 Kotlin Tutorials for Beginners，作者：Peter Sommerhoff

还可以在我们（官方）的 YouTube 频道上查看其他教程与内容。

有没有 Kotlin 社区？

有！Kotlin 有一个非常有活力的社区。Kotlin 开发人员常出现在 Kotlin 论坛、 StackOverflow 上并且更积极地活跃在 Kotlin Slack （截至 2020 年 4 月有近 30000 名成员）上。

有没有 Kotlin 活动？

有！现在有很多用户组和集会组专注于 Kotlin。你可以在网站上找到一个列表。 此外，还有世界各地的社区组织的 Kotlin 之夜活动。

有没有 Kotlin 大会？

有！ KotlinConf 是由 JetBrains 主办的年度大会，汇集了来自世界各地的开发者、爱好者与专家，分享他们的 Kotlin 知识与经验。

除了技术讲座与研讨会之外，KotlinConf 还提供联络机会、社区互动与社交活动，与会者可以与其他 Kotliner 联系并交流想法。 它是在 Kotlin 生态系统内促进协作与社区建设的一个平台。

Kotlin 也会在全球不同地方举行大会。你可以在 官网上找到即将到来的会谈列表。

Kotlin 上社交媒体吗？

上。 可以订阅 Kotlin YouTube 频道以及在 Twitter 上关注 Kotlin。

其他在线 Kotlin 资源呢？

网站上有一堆在线资源，包括社区成员的 Kotlin 文摘、 通讯、播客等等。

在哪里可以获得高清 Kotlin 徽标？

徽标可以在这里下载。 使用该徽标时，请遵循压缩包中的 `guidelines.pdf` 以及 Kotlin 品牌使用指南 中的简单规则。

如需了解更多信息，请查阅 Kotlin 品牌素材页。

Kotlin 兼容性指南

- Kotlin 2.1 的兼容性指南
- Kotlin 2.0 的兼容性指南
- Kotlin 1.9 的兼容性指南
- Kotlin 1.8 的兼容性指南
- Kotlin 1.7.20 的兼容性指南
- Kotlin 1.7 的兼容性指南
- Kotlin 1.6 的兼容性指南
- Kotlin 1.5 的兼容性指南
- Kotlin 1.4 的兼容性指南
- Kotlin 1.3 的兼容性指南
- 兼容模式

Kotlin 2.1 的兼容性指南

Keeping the Language Modern and *Comfortable Updates* are among the fundamental principles in Kotlin Language Design. The former says that constructs which obstruct language evolution should be removed, and the latter says that this removal should be well-communicated beforehand to make code migration as smooth as possible.

While most of the language changes were already announced through other channels, like update changelogs or compiler warnings, this document summarizes them all, providing a complete reference for migration from Kotlin 2.0 to Kotlin 2.1.

Basic terms

In this document we introduce several kinds of compatibility:

- *source*: source-incompatible change stops code that used to compile fine (without errors or warnings) from compiling anymore
- *binary*: two binary artifacts are said to be binary-compatible if interchanging them doesn't lead to loading or linkage errors
- *behavioral*: a change is said to be behavioral-incompatible if the same program demonstrates different behavior before and after applying the change

Remember that those definitions are given only for pure Kotlin. Compatibility of Kotlin code from the other languages perspective (for example, from Java) is out of the scope of this document.

Language

Remove language versions 1.4 and 1.5

> **Issue**: KT-60521
>
> **Component**: Core language
>
> **Incompatible change type**: source
>
> **Short summary**: Kotlin 2.1 introduces language version 2.1 and removes support for language versions 1.4 and 1.5. Language versions 1.6 and 1.7 are deprecated.
>
> **Deprecation cycle**:
>
> - 1.6.0: report a warning for language version 1.4

- 1.9.0: report a warning for language version 1.5
- 2.1.0: report a warning for language versions 1.6 and 1.7; raise the warning to an error for language versions 1.4 and 1.5

Change the typeOf() function behavior on Kotlin/Native

Issue: KT-70754

Component: Core language

Incompatible change type: behavioral

Short summary: The behavior of the `typeOf()` function on Kotlin/Native is aligned with Kotlin/JVM to ensure consistency across platforms.

Deprecation cycle:

- 2.1.0: align the `typeOf()` function behavior on Kotlin/Native

Prohibit exposing types through type parameters' bounds

Issue: KT-69653

Component: Core language

Incompatible change type: source

Short summary: Exposing types with lower visibility through type parameter bounds is now prohibited, addressing inconsistencies in type visibility rules. This change ensures that bounds on type parameters follow the same visibility rules as classes, preventing issues like IR validation errors in the JVM.

Deprecation cycle:

- 2.1.0: report a warning for exposing types via type parameter bounds with lower visibility
- 2.2.0: raise the warning to an error

Prohibit inheriting an abstract var property and a val property with the same name

Issue: KT-58659

Component: Core language

Incompatible change type: source

> **Short summary**: If a class inherits an abstract `var` property from an interface and a `val` property with the same name from a superclass, it now triggers a compilation error. This resolves runtime crashes caused by missing setters in such cases.
>
> **Deprecation cycle**:
>
> - 2.1.0: report a warning (or an error in progressive mode) when a class inherits an abstract `var` property from an interface and a `val` property with the same name from a superclass
> - 2.2.0: raise the warning to an error

Report error when accessing uninitialized enum entries

> **Issue**: KT-68451
>
> **Component**: Core language
>
> **Incompatible change type**: source
>
> **Short summary**: The compiler now reports an error when uninitialized enum entries are accessed during enum class or entry initialization. This aligns the behavior with member property initialization rules, preventing runtime exceptions and ensuring consistent logic.
>
> **Deprecation cycle**:
>
> - 2.1.0: report an error when accessing uninitialized enum entries

Changes in K2 smart cast propagation

> **Issue**: KTLC-34
>
> **Component**: Core language
>
> **Incompatible change type**: behavioral
>
> **Short summary**: The K2 compiler changes its behavior for smart cast propagation by introducing bidirectional propagation of type information for inferred variables, like `val x = y`. Explicitly typed variables, such as `val x: T = y`, no longer propagate type information, ensuring stricter adherence to declared types.
>
> **Deprecation cycle**:
>
> - 2.1.0: enable the new behavior

Correct the handling of member-extension property overrides in Java subclasses

> **Issue**: KTLC-35
>
> **Component**: Core language
>
> **Incompatible change type**: behavioral
>
> **Short summary**: The getter for member-extension properties overridden by Java subclasses is now hidden in the subclass's scope, aligning its behavior with that of regular Kotlin properties.
>
> **Deprecation cycle**:
>
> - 2.1.0: enable the new behavior

Correct visibility alignment for getters and setters of var properties overriding a protected val

> **Issue**: KTLC-36
>
> **Component**: Core language
>
> **Incompatible change type**: binary
>
> **Short summary**: The visibility of getters and setters for `var` properties overriding a `protected val` property is now consistent, with both inheriting the visibility of the overridden `val` property.
>
> **Deprecation cycle**:
>
> - 2.1.0: enforce consistent visibility for both getters and setters in K2; K1 remains unaffected

Raise severity of JSpecify nullability mismatch diagnostics to errors

> **Issue**: KTLC-11
>
> **Component**: Core language
>
> **Incompatible change type**: source
>
> **Short summary**: Nullability mismatches from `org.jspecify.annotations`, such as `@NonNull`, `@Nullable`, and `@NullMarked` are now treated as errors instead of warnings, enforcing stricter type safety for Java interoperability. To adjust the severity of these diagnostics, use the `-Xnullability-annotations` compiler option.
>
> **Deprecation cycle**:
>
> - 1.6.0: report warnings for potential nullability mismatches

- 1.8.20: expand warnings to specific JSpecify annotations, including: `@Nullable`, `@NullnessUnspecified`, `@NullMarked`, and legacy annotations in `org.jspecify.nullness` (JSpecify 0.2 and earlier)
- 2.0.0: add support for the `@NonNull` annotation
- 2.1.0: change default mode to `strict` for JSpecify annotations, converting warnings into errors; use `-Xnullability-annotations=@org.jspecify.annotations:warning` or `-Xnullability-annotations=@org.jspecify.annotations:ignore` to override the default behavior

Change overload resolution to prioritize extension functions over invoke calls in ambiguous cases

Issue: KTLC-37

Component: Core language

Incompatible change type: behavioral

Short summary: Overload resolution now consistently prioritizes extension functions over invoke calls in ambiguous cases. This resolves inconsistencies in the resolution logic for local functions and properties. The change applies only after recompilation, without affecting precompiled binaries.

Deprecation cycle:

- 2.1.0: change overload resolution to consistently prioritize extension functions over `invoke` calls for extension functions with matching signatures; this change applies only after recompilation and does not affect precompiled binaries

Prohibit returning nullable values from lambdas in SAM constructors of JDK function interfaces

Issue: KTLC-42

Component: Core language

Incompatible change type: source

Short summary: Returning nullable values from lambdas in SAM constructors of JDK function interfaces now triggers a compilation error if the specified type argument is non-nullable. This resolves issues where nullability mismatches could lead to runtime exceptions, ensuring stricter type safety.

Deprecation cycle:

- 2.0.0: report a deprecation warning for nullable return values in SAM constructors of JDK function interfaces
- 2.1.0: enable the new behavior by default

Correct handling of private members conflicting with public members in Kotlin/Native

Issue: KTLC-43

Component: Core language

Incompatible change type: behavioral

Short summary: In Kotlin/Native, private members no longer override or conflict with public members in a superclass, aligning behavior with Kotlin/JVM. This resolves inconsistencies in override resolution and eliminates unexpected behavior caused by separate compilation.

Deprecation cycle:

- 2.1.0: private functions and properties in Kotlin/Native no longer override or affect public members in a superclass, aligning with JVM behavior

Forbid access to private operator functions in public inline functions

Issue: KTLC-71

Component: Core language

Incompatible change type: source

Short summary: Private operator functions such as `getValue()`, `setValue()`, `provideDelegate()`, `hasNext()`, and `next()` can no longer be accessed in public inline functions.

Deprecation cycle:

- 2.0.0: report a deprecation warning for accessing private operator functions in public inline functions
- 2.1.0: raise the warning to an error

Prohibit passing invalid arguments to invariant parameters annotated with @UnsafeVariance

Issue: KTLC-72

> **Component**: Core language
>
> **Incompatible change type**: source
>
> **Short summary**: The compiler now ignores `@UnsafeVariance` annotations during type checks, enforcing stricter type safety for invariant type parameters. This prevents invalid calls that rely on `@UnsafeVariance` to bypass expected type checks.
>
> **Deprecation cycle**:
>
> - 2.1.0: activate the new behavior

Report nullability errors for error-level nullable arguments of warning-level Java types

> **Issue**: KTLC-100
>
> **Component**: Core language
>
> **Incompatible change type**: source
>
> **Short summary**: The compiler now detects nullability mismatches in Java methods where a warning-level nullable type contains type arguments with stricter, error-level nullability. This ensures that previously ignored errors in type arguments are reported correctly.
>
> **Deprecation cycle**:
>
> - 2.0.0: report a deprecation warning for nullability mismatches in Java methods with stricter type arguments
> - 2.1.0: raise the warning to an error

Report implicit usages of inaccessible types

> **Issue**: KTLC-3
>
> **Component**: Core language
>
> **Incompatible change type**: source
>
> **Short summary**: The compiler now reports usages of inaccessible types in function literals and type arguments, preventing compilation and runtime failures caused by incomplete type information.
>
> **Deprecation cycle**:
>
> - 2.0.0: report warnings for function literals with parameters or receivers of inaccessible non-generic types and for types with inaccessible type argument; report errors for function

> literals with parameters or receivers of inaccessible generic types and for types with inaccessible generic type arguments in specific scenarios
> - 2.1.0: raise warnings to errors for function literals with parameters and receivers of inaccessible non-generic types
> - 2.2.0: raise warnings to errors for types with inaccessible type arguments

Standard library

Deprecate locale-sensitive case conversion functions for Char and String

> **Issue**: KT-43023
>
> **Component**: kotlin-stdlib
>
> **Incompatible change type**: source
>
> **Short summary**: Among other Kotlin standard library APIs, locale-sensitive case conversion functions for `Char` and `String`, such as `Char.toUpperCase()` and `String.toLowerCase()`, are deprecated. Replace them with locale-agnostic alternatives like `String.lowercase()` or explicitly specify the locale for locale-sensitive behavior, such as `String.lowercase(Locale.getDefault())`.
>
> For a comprehensive list of deprecated Kotlin standard library APIs in Kotlin 2.1.0, see KT-71628.
>
> **Deprecation cycle**:
>
> - 1.4.30: introduce locale-agnostic alternatives as experimental API
> - 1.5.0: deprecate locale-sensitive case conversion functions with a warning
> - 2.1.0: raise the warning to an error

Remove kotlin-stdlib-common JAR artifact

> **Issue**: KT-62159
>
> **Component**: kotlin-stdlib
>
> **Incompatible change type**: binary
>
> **Short summary**: The `kotlin-stdlib-common.jar` artifact, previously used for legacy multiplatform declarations metadata, is deprecated and replaced by `.klib` files as the standard format for common multiplatform declarations metadata. This change does not affect the main `kotlin-stdlib.jar` or `kotlin-stdlib-all.jar` artifacts.

> **Deprecation cycle**:
>
> - 2.1.0: deprecate and remove `kotlin-stdlib-common.jar` artifact

Deprecate appendln() in favor of appendLine()

> **Issue**: KTLC-27
>
> **Component**: kotlin-stdlib
>
> **Incompatible change type**: source
>
> **Short summary**: `StringBuilder.appendln()` is deprecated in favor of `StringBuilder.appendLine()`.
>
> **Deprecation cycle**:
>
> - 1.4.0: the `appendln()` function is deprecated; report a warning on use
> - 2.1.0: raise the warning to an error

Deprecate freezing-related APIs in Kotlin/Native

> **Issue**: KT-69545
>
> **Component**: kotlin-stdlib
>
> **Incompatible change type**: source
>
> **Short summary**: Freezing-related APIs in Kotlin/Native, previously marked with the `@FreezingIsDeprecated` annotation, are now deprecated. This aligns with the introduction of the new memory manager that removes the need for freezing objects for thread sharing. For migration details, see the Kotlin/Native migration guide.
>
> **Deprecation cycle**:
>
> - 1.7.20: deprecate freezing-related APIs with a warning
> - 2.1.0: raise the warning to an error

Change Map.Entry behavior to fail-fast on structural modification

> **Issue**: KTLC-23
>
> **Component**: kotlin-stdlib
>
> **Incompatible change type**: behavioral

> **Short summary**: Accessing a `Map.Entry` key-value pair after its associated map has been structurally modified now throws a `ConcurrentModificationException`.
>
> **Deprecation cycle**:
>
> - 2.1.0: throw an exception when a map structural modification is detected

Tools

Deprecate KotlinCompilationOutput#resourcesDirProvider

> **Issue**: KT-69255
>
> **Component**: Gradle
>
> **Incompatible change type**: source
>
> **Short summary**: The `KotlinCompilationOutput#resourcesDirProvider` field is deprecated. Use `KotlinSourceSet.resources` in your Gradle build script instead to add additional resource directories.
>
> **Deprecation cycle**:
>
> - 2.1.0: `KotlinCompilationOutput#resourcesDirProvider` is deprecated

Deprecate registerKotlinJvmCompileTask(taskName, moduleName) function

> **Issue**: KT-69927
>
> **Component**: Gradle
>
> **Incompatible change type**: source
>
> **Short summary**: The `registerKotlinJvmCompileTask(taskName, moduleName)` function is deprecated in favor of the new `registerKotlinJvmCompileTask(taskName, compilerOptions, explicitApiMode)` function, which now accepts `KotlinJvmCompilerOptions`. This allows you to pass a `compilerOptions` instance, typically from an extension or target, with values used as conventions for the task's options.
>
> **Deprecation cycle**:
>
> - 2.1.0: the `registerKotlinJvmCompileTask(taskName, moduleName)` function is deprecated

Deprecate registerKaptGenerateStubsTask(taskName) function

> **Issue**: KT-70383
>
> **Component**: Gradle
>
> **Incompatible change type**: source
>
> **Short summary**: The `registerKaptGenerateStubsTask(taskName)` function is deprecated. Use the new `registerKaptGenerateStubsTask(compileTask, kaptExtension, explicitApiMode)` function instead. This new version allows you to link values as conventions from the relevant `KotlinJvmCompile` task, ensuring both tasks are using the same set of options.
>
> **Deprecation cycle**:
>
> - 2.1.0: the `registerKaptGenerateStubsTask(taskName)` function is deprecated

Deprecate KotlinTopLevelExtension and KotlinTopLevelExtensionConfig interfaces

> **Issue**: KT-71602
>
> **Component**: Gradle
>
> **Incompatible change type**: behavioral
>
> **Short summary**: `KotlinTopLevelExtension` and `KotlinTopLevelExtensionConfig` interfaces are deprecated in favor of a new `KotlinTopLevelExtension` interface. This interface merges `KotlinTopLevelExtensionConfig`, `KotlinTopLevelExtension`, and `KotlinProjectExtension` to streamline API hierarchy, and provide official access to the JVM toolchain and compiler properties.
>
> **Deprecation cycle**:
>
> - 2.1.0: the `KotlinTopLevelExtension` and `KotlinTopLevelExtensionConfig` interfaces are deprecated

Remove kotlin-compiler-embeddable from build runtime dependencies

> **Issue**: KT-61706
>
> **Component**: Gradle
>
> **Incompatible change type**: source
>
> **Short summary**: The `kotlin-compiler-embeddable` dependency is removed from the runtime in Kotlin Gradle Plugin (KGP). Required modules are now included directly in KGP artifacts, with the Kotlin language version limited to 2.0 to support compatibility with Gradle Kotlin

> runtime in versions below 8.2.
>
> **Deprecation cycle**:
>
> - 2.1.0: report a warning on using `kotlin-compiler-embeddable`
> - 2.2.0: raise the warning to an error

Hide compiler symbols from the Kotlin Gradle Plugin API

> **Issue**: KT-70251
>
> **Component**: Gradle
>
> **Incompatible change type**: source
>
> **Short summary**: Compiler module symbols bundled within the Kotlin Gradle Plugin (KGP), such as `KotlinCompilerVersion`, are hidden from the public API to prevent unintended access in build scripts.
>
> **Deprecation cycle**:
>
> - 2.1.0: report a warning on accessing these symbols
> - 2.2.0: raise the warning to an error

Add support for multiple stability configuration files

> **Issue**: KT-68345
>
> **Component**: Gradle
>
> **Incompatible change type**: source
>
> **Short summary**: The `stabilityConfigurationFile` property in the Compose extension is deprecated in favor of a new `stabilityConfigurationFiles` property, which allows specifying multiple configuration files.
>
> **Deprecation cycle**:
>
> - 2.1.0: `stabilityConfigurationFile` property is deprecated

Remove deprecated platform plugin IDs

> **Issue**: KT-65565
>
> **Component**: Gradle
>
> **Incompatible change type**: source

> **Short summary**: Support for these platform plugin IDs have been removed:
>
> - `kotlin-platform-common`
> - `org.jetbrains.kotlin.platform.common`
>
> **Deprecation cycle**:
>
> - 1.3: the platform plugin IDs are deprecated
> - 2.1.0: the platform plugin IDs are no longer supported

Compatibility guide for Kotlin 2.0

Keeping the Language Modern and *Comfortable Updates* are among the fundamental principles in Kotlin Language Design. The former says that constructs which obstruct language evolution should be removed, and the latter says that this removal should be well-communicated beforehand to make code migration as smooth as possible.

While most of the language changes were already announced through other channels, like updated changelogs or compiler warnings, this document provides a complete reference for migration from Kotlin 1.9 to Kotlin 2.0.

> The Kotlin K2 compiler is introduced as part of Kotlin 2.0. For information on the benefits of the new compiler, changes you might encounter during migration, and how to roll back to the previous compiler, see K2 compiler migration guide.
>
> 

Basic terms

In this document we introduce several kinds of compatibility:

- *source*: source-incompatible change stops code that used to compile fine (without errors or warnings) from compiling anymore
- *binary*: two binary artifacts are said to be binary-compatible if interchanging them doesn't lead to loading or linkage errors
- *behavioral*: a change is said to be behavioral-incompatible if the same program demonstrates different behavior before and after applying the change

Remember that those definitions are given only for pure Kotlin. Compatibility of Kotlin code from the other languages perspective (for example, from Java) is out of the scope of this document.

Language

Deprecate use of a synthetic setter on a projected receiver

> **Issue**: KT-54309
>
> **Component**: Core language
>
> **Incompatible change type**: source

> **Short summary**: If you use the synthetic setter of a Java class to assign a type that conflicts with the class's projected type, an error is triggered.
>
> **Deprecation cycle**:
>
> - 1.8.20: report a warning when a synthetic property setter has a projected parameter type in contravariant position making call-site argument types incompatible
> - 2.0.0: raise the warning to an error

Correct mangling when calling functions with inline class parameters that are overloaded in a Java subclass

> **Issue**: KT-56545
>
> **Component**: Core language
>
> **Incompatible change type**: behavioral
>
> **Deprecation cycle**:
>
> - 2.0.0: use the correct mangling behavior in function invocations; to revert to the previous behaviour, use the `-XXLanguage:-MangleCallsToJavaMethodsWithValueClasses` compiler option.

Correct type approximation algorithm for contravariant captured types

> **Issue**: KT-49404
>
> **Component**: Core language
>
> **Incompatible change type**: source
>
> **Deprecation cycle**:
>
> - 1.8.20: report a warning on problematic calls
> - 2.0.0: raise the warning to an error

Prohibit accessing property value before property initialization

> **Issue**: KT-56408
>
> **Component**: Core language
>
> **Incompatible change type**: source
>
> **Deprecation cycle**:

> - 2.0.0: report an error when a property is accessed before initialization in the affected contexts

Report error when there's ambiguity in imported classes with the same name

> **Issue**: KT-57750
>
> **Component**: Core language
>
> **Incompatible change type**: source
>
> **Deprecation cycle**:
>
> - 2.0.0: report an error when resolving a class name that is present in several packages imported with a star import

Generate Kotlin lambdas via invokedynamic and LambdaMetafactory by default

> **Issue**: KT-45375
>
> **Component**: Core language
>
> **Incompatible change type**: behavioral
>
> **Deprecation cycle**:
>
> - 2.0.0: implement new behavior; lambdas are generated using `invokedynamic` and `LambdaMetafactory` by default

Forbid if condition with one branch when an expression is required

> **Issue**: KT-57871
>
> **Component**: Core language
>
> **Incompatible change type**: source
>
> **Deprecation cycle**:
>
> - 2.0.0: report an error in case the `if` condition has only one branch

Prohibit violation of self upper bounds by passing a star-projection of a generic type

> **Issue**: KT-61718
>
> **Component**: Core language
>
> **Incompatible change type**: source
>
> **Deprecation cycle**:
>
> - 2.0.0: report an error when self upper bounds are violated by passing a star-projection of a generic type

Approximate anonymous types in private inline functions return type

> **Issue**: KT-54862
>
> **Component**: Core language
>
> **Incompatible change type**: source
>
> **Deprecation cycle**:
>
> - 1.9.0: report a warning on private inline functions if the inferred return type contains an anonymous type
> - 2.0.0: approximate return type of such private inline functions to a supertype

Change overload resolution behavior to prioritize local extension function calls over invoke conventions of local functional type properties

> **Issue**: KT-37592
>
> **Component**: Core language
>
> **Incompatible change type**: behavioral
>
> **Deprecation cycle**:
>
> - 2.0.0: new overload resolution behavior; function calls are consistently prioritized over invoke conventions

Report error when an inherited member conflict occurs due to a change in a supertype from binary dependency

> **Issue**: KT-51194

> **Component**: Core language
>
> **Incompatible change type**: source
>
> **Deprecation cycle**:
>
> - 1.7.0: report a warning CONFLICTING_INHERITED_MEMBERS_WARNING on declarations where inherited member conflict has occurred in the supertype from binary dependency
> - 2.0.0: raise the warning to an error: CONFLICTING_INHERITED_MEMBERS

Ignore @UnsafeVariance annotations on parameters in invariant types

> **Issue**: KT-57609
>
> **Component**: Core language
>
> **Incompatible change type**: source
>
> **Deprecation cycle**:
>
> - 2.0.0: implement new behavior; the `@UnsafeVariance` annotation is ignored when reporting errors about type mismatch in contravariant parameters

Change type for out-of-call references to a companion object's member

> **Issue**: KT-54316
>
> **Component**: Core language
>
> **Incompatible change type**: source
>
> **Deprecation cycle**:
>
> - 1.8.20: report a warning on a companion object function reference type inferred as an unbound reference
> - 2.0.0: change the behavior so that companion object function references are inferred as bound references in all usage contexts

Prohibit exposure of anonymous types from private inline functions

> **Issue**: KT-33917

> **Component**: Core language
>
> **Incompatible change type**: source
>
> **Deprecation cycle**:
>
> - 1.3.0: report a warning on calls to own members of anonymous objects, returned from private inline functions
> - 2.0.0: approximate return type of such private inline functions to a supertype and don't resolve calls to anonymous object members

Report error for an unsound smart cast after a while-loop break

> **Issue**: KT-22379
>
> **Component**: Core language
>
> **Incompatible change type**: source
>
> **Deprecation cycle**:
>
> - 2.0.0: implement new behavior; the old behavior can be restored by switching to language version 1.9

Report error when a variable of an intersection type is assigned a value that is not a subtype of that intersection type

> **Issue**: KT-53752
>
> **Component**: Core language
>
> **Incompatible change type**: source
>
> **Deprecation cycle**:
>
> - 2.0.0: report an error when a variable having an intersection type is assigned a value that is not a subtype of that intersection type

Require opt-in when an interface constructed with a SAM constructor contains a method that requires an opt-in

> **Issue**: KT-52628
>
> **Component**: Core language
>
> **Incompatible change type**: source

> **Deprecation cycle**:
>
> - 1.7.20: report a warning for `OptIn` usages via SAM constructor
> - 2.0.0: raise the warning to an error for `OptIn` usages via SAM constructor (or keep reporting the warning if `OptIn` marker severity is a warning)

Prohibit upper bound violation in typealias constructors

> **Issue**: KT-54066
>
> **Component**: Core language
>
> **Incompatible change type**: source
>
> **Deprecation cycle**:
>
> - 1.8.0: introduce a warning for cases when the upper bound is violated in typealias constructors
> - 2.0.0: raise the warning to an error in the K2 compiler

Make the real type of a destructuring variable consistent with the explicit type when specified

> **Issue**: KT-57011
>
> **Component**: Core language
>
> **Incompatible change type**: source
>
> **Deprecation cycle**:
>
> - 2.0.0: implement new behavior; the real type of destructuring variable is now consistent with the explicit type when specified

Require opt-in when calling a constructor that has parameter types with default values that require an opt-in

> **Issue**: KT-55111
>
> **Component**: Core language
>
> **Incompatible change type**: source
>
> **Deprecation cycle**:
>
> - 1.8.20: report a warning on constructor calls that has parameter types which require opt-in
> - 2.0.0: raise the warning to an error (or keep reporting a warning if the `OptIn` marker

> severity is a warning)

Report ambiguity between a property and an enum entry with the same name at the same scope level

> **Issue**: KT-52802
>
> **Component**: Core language
>
> **Incompatible change type**: source
>
> **Deprecation cycle**:
>
> - 1.7.20: report a warning when the compiler resolves to a property instead of an enum entry at the same scope level
> - 2.0.0: report ambiguity when the compiler encounters both a property and an enum entry with the same name at the same scope level in the K2 compiler (leaving the warning as is in the old compiler)

Change qualifier resolution behavior to prefer companion property over enum entry

> **Issue**: KT-47310
>
> **Component**: Core language
>
> **Incompatible change type**: source
>
> **Deprecation cycle**:
>
> - 2.0.0: implement new resolution behavior; companion property is preferred over enum entry

Resolve invoke call receiver type and the invoke function type as if written in desugared form

> **Issue**: KT-58260
>
> **Component**: Core language
>
> **Incompatible change type**: source
>
> **Deprecation cycle**:
>
> - 2.0.0: resolve invoke call receiver type and the invoke function type independently as if they were written in a desugared form

Prohibit exposing private class members through non-private inline functions

> **Issue**: KT-55179
>
> **Component**: Core language
>
> **Incompatible change type**: source
>
> **Deprecation cycle**:
>
> - 1.9.0: report the `PRIVATE_CLASS_MEMBER_FROM_INLINE_WARNING` warning when calling private class companion object member from internal inline functions
> - 2.0.0: raise this warning to the `PRIVATE_CLASS_MEMBER_FROM_INLINE` error

Correct nullability of definitely non-null types in projected generic types

> **Issue**: KT-54663
>
> **Component**: Core language
>
> **Incompatible change type**: source
>
> **Deprecation cycle**:
>
> - 2.0.0: implement new behavior; projected types take into account all in-place not-null types

Change inferred type of prefix increment to match getter's return type instead of inc() operator's return type

> **Issue**: KT-57178
>
> **Component**: Core language
>
> **Incompatible change type**: source
>
> **Deprecation cycle**:
>
> - 2.0.0: implement new behavior; inferred type of prefix increment is changed to match getter's return type instead of the `inc()` operator's return type

Enforce bound checks when inheriting inner classes from generic inner classes declared in superclasses

> **Issue**: KT-61749

> **Component**: Core language
>
> **Incompatible change type**: source
>
> **Deprecation cycle**:
>
> - 2.0.0: report an error when upper bound of the type parameter of a generic inner superclass is violated

Forbid assigning callable references with SAM types when the expected type is a function type with a function type parameter

> **Issue**: KT-64342
>
> **Component**: Core language
>
> **Incompatible change type**: source
>
> **Deprecation cycle**:
>
> - 2.0.0: report a compilation error on callable references with SAM types when the expected type is a function type with a function type parameter

Consider companion object scope for annotation resolution on companion objects

> **Issue**: KT-64299
>
> **Component**: Core language
>
> **Incompatible change type**: behavioral
>
> **Deprecation cycle**:
>
> - 2.0.0: implement new behavior; the companion object scope is now not ignored during annotation resolution on companion objects

Change evaluation semantics for combination of safe calls and convention operators

> **Issue**: KT-41034
>
> **Component**: Core language
>
> **Incompatible change type**: behavioral
>
> **Deprecation cycle**:

- 1.4.0: report warnings on each incorrect call
- 2.0.0: implement new resolution behavior

Require properties with backing field and a custom setter to be immediately initialized

Issue: KT-58589

Component: Core language

Incompatible change type: behavioral

Deprecation cycle:

- 1.9.20: introduce the `MUST_BE_INITIALIZED` warning for cases where there is no primary constructor
- 2.0.0: raise the warning to an error

Prohibit Unit conversion on arbitrary expressions in invoke operator convention call

Issue: KT-61182

Component: Core language

Incompatible change type: behavioral

Deprecation cycle:

- 2.0.0: report an error when Unit conversion is applied to an arbitrary expression on variable and invoke resolution; use the `-XXLanguage:+UnitConversionsOnArbitraryExpressions` compiler option to keep the previous behavior with affected expressions.

Forbid nullable assignment to non-null Java field when the field is accessed with a safe call

Issue: KT-62998

Component: Core language

Incompatible change type: source

Deprecation cycle:

- 2.0.0: report to an error in case a nullable is assigned to a non-null Java field

Require star-projected type when overriding a Java method containing a raw-type parameter

> **Issue**: KT-57600
>
> **Component**: Core language
>
> **Incompatible change type**: source
>
> **Deprecation cycle**:
>
> - 2.0.0: implement new behavior; overriding is forbidden for raw type parameters

Change (V)::foo reference resolution when V has a companion

> **Issue**: KT-47313
>
> **Component**: Core language
>
> **Incompatible change type**: behavioral
>
> **Deprecation cycle**:
>
> - 1.6.0: report a warning on callable references currently bound to companion object instances
> - 2.0.0: implement new behavior; adding parentheses around a type no longer makes it a reference to the type's companion object instance

Forbid implicit non-public API access in effectively public inline functions

> **Issue**: KT-54997
>
> **Component**: Core language
>
> **Incompatible change type**: source
>
> **Deprecation cycle**:
>
> - 1.8.20: report a compilation warning when implicit non-public API is accessed in public inline functions
> - 2.0.0: raise the warning to an error

Prohibit use-site get annotations on property getters

> **Issue**: KT-57422

> **Component**: Core language
>
> **Incompatible change type**: source
>
> **Deprecation cycle**:
>
> - 1.9.0: report a warning (an error in the progressive mode) on use-site `get` annotations on getters
> - 2.0.0: raise the warning to the `INAPPLICABLE_TARGET_ON_PROPERTY` error; use `-XXLanguage:-ProhibitUseSiteGetTargetAnnotations` to revert to a warning

Prevent implicit inference of type parameters into upper bounds in builder inference lambda functions

> **Issue**: KT-47986
>
> **Component**: Core language
>
> **Incompatible change type**: source
>
> **Deprecation cycle**:
>
> - 1.7.20: report a warning (or an error in the progressive mode) when the type parameter for a type argument cannot be inferred into declared upper bounds
> - 2.0.0: raise the warning to an error

Keep nullability when approximating local types in public signatures

> **Issue**: KT-53982
>
> **Component**: Core language
>
> **Incompatible change type**: source
>
> **Deprecation cycle**:
>
> - 1.8.0: flexible types are approximated by flexible supertypes; report a warning when a declaration is inferred to have a non-nullable type that should be nullable, prompting to specify the type explicitly to avoid NPEs
> - 2.0.0: nullable types are approximated by nullable supertypes

Remove special handling for false && ... and false || ... for the purposes of smart-casting

> **Issue**: KT-65776

> **Component**: Core language
>
> **Incompatible change type**: source
>
> **Deprecation cycle**:
>
> - 2.0.0: implement new behavior; no special handling for `false && ...` and `false || ...`

Forbid inline open functions in enums

> **Issue**: KT-34372
>
> **Component**: Core language
>
> **Incompatible change type**: source
>
> **Deprecation cycle**:
>
> - 1.8.0: report a warning on inline open functions in enums
> - 2.0.0: raise the warning to an error

Tools

Visibility changes in Gradle

> **Issue**: KT-64653
>
> **Component**: Gradle
>
> **Incompatible change type**: source
>
> **Short summary**: Previously, certain Kotlin DSL functions and properties intended for a specific DSL context would inadvertently leak into other DSL contexts. We've added the `@KotlinGradlePluginDsl` annotation, which prevents the exposure of the Kotlin Gradle plugin DSL functions and properties to levels where they are not intended to be available. The following levels are separated from each other:
>
> - Kotlin extension
> - Kotlin target
> - Kotlin compilation
> - Kotlin compilation task
>
> **Deprecation cycle**:
>
> - 2.0.0: for most popular cases, the compiler reports warnings with suggestions on how to fix them if your build script is configured incorrectly; otherwise, the compiler reports an error

Deprecate kotlinOptions DSL

> **Issue**: KT-63419
>
> **Component**: Gradle
>
> **Incompatible change type**: source
>
> **Short summary**: The ability to configure compiler options through the `kotlinOptions` DSL and the related `KotlinCompile<KotlinOptions>` task interface have been deprecated.
>
> **Deprecation cycle**:
>
> - 2.0.0: report a warning

Deprecate compilerOptions in KotlinCompilation DSL

> **Issue**: KT-65568
>
> **Component**: Gradle
>
> **Incompatible change type**: source
>
> **Short summary**: The ability to configure the `compilerOptions` property in the `KotlinCompilation` DSL has been deprecated.
>
> **Deprecation cycle**:
>
> - 2.0.0: report a warning

Deprecate old ways of CInteropProcess handling

> **Issue**: KT-62795
>
> **Component**: Gradle
>
> **Incompatible change type**: source
>
> **Short summary**: the `CInteropProcess` task and the `CInteropSettings` class now use the `definitionFile` property instead of `defFile` and `defFileProperty`.
>
> This eliminates the need to add extra `dependsOn` relations between the `CInteropProcess` task and the task that generates `defFile` when the `defFile` is dynamically generated.
>
> In Kotlin/Native projects, Gradle now lazily verifies the presence of the `definitionFile` property after the connected task has run later in the build process.
>
> **Deprecation cycle**:

- 2.0.0: `defFile` and `defFileProperty` parameters are deprecated

Remove kotlin.useK2 Gradle property

Issue: KT-64379

Component: Gradle

Incompatible change type: behavioral

Short summary: The `kotlin.useK2` Gradle property has been removed. In Kotlin 1.9.*, it could be used to enable the K2 compiler. In Kotlin 2.0.0 and later, the K2 compiler is enabled by default, so the property has no effect and cannot be used to switch back to the previous compiler.

Deprecation cycle:

- 1.8.20: the `kotlin.useK2` Gradle property is deprecated
- 2.0.0: the `kotlin.useK2` Gradle property is removed

Remove deprecated platform plugin IDs

Issue: KT-65187

Component: Gradle

Incompatible change type: source

Short summary: support for these platform plugin IDs have been removed:

- `kotlin-platform-android`
- `kotlin-platform-jvm`
- `kotlin-platform-js`
- `org.jetbrains.kotlin.platform.android`
- `org.jetbrains.kotlin.platform.jvm`
- `org.jetbrains.kotlin.platform.js`

Deprecation cycle:

- 1.3: the platform plugin IDs are deprecated
- 2.0.0: the platform plugin IDs are no longer supported

Remove outputFile JavaScript compiler option

Issue: KT-61116

Component: Gradle

> **Incompatible change type**: source
>
> **Short summary**: The `outputFile` JavaScript compiler option has been removed. Instead, you can use the `destinationDirectory` property of the `Kotlin2JsCompile` task to specify the directory where the compiled JavaScript output files are written.
>
> **Deprecation cycle**:
>
> - 1.9.25: the `outputFile` compiler option is deprecated
> - 2.0.0: the `outputFile` compiler option is removed

Kotlin 1.9 的兼容性指南

Keeping the Language Modern and *Comfortable Updates* are among the fundamental principles in Kotlin Language Design. The former says that constructs which obstruct language evolution should be removed, and the latter says that this removal should be well-communicated beforehand to make code migration as smooth as possible.

While most of the language changes were already announced through other channels, like update changelogs or compiler warnings, this document summarizes them all, providing a complete reference for migration from Kotlin 1.8 to Kotlin 1.9.

Basic terms

In this document we introduce several kinds of compatibility:

- *source*: source-incompatible change stops code that used to compile fine (without errors or warnings) from compiling anymore
- *binary*: two binary artifacts are said to be binary-compatible if interchanging them doesn't lead to loading or linkage errors
- *behavioral*: a change is said to be behavioral-incompatible if the same program demonstrates different behavior before and after applying the change

Remember that those definitions are given only for pure Kotlin. Compatibility of Kotlin code from the other languages perspective (for example, from Java) is out of the scope of this document.

Language

Remove language version 1.3

> **Issue**: KT-61111
>
> **Component**: Core language
>
> **Incompatible change type**: source
>
> **Short summary**: Kotlin 1.9 introduces language version 1.9 and removes support for language version 1.3.
>
> **Deprecation cycle**:
>
> - 1.6.0: report a warning

> - 1.9.0: raise the warning to an error

Prohibit super constructor call when the super interface type is a function literal

> **Issue**: KT-46344
>
> **Component**: Core language
>
> **Incompatible change type**: source
>
> **Short summary**: If an interface inherits from a function literal type, Kotlin 1.9 prohibits super constructor calls because no such constructor exists.
>
> **Deprecation cycle**:
>
> - 1.7.0: report a warning (or an error in progressive mode)
> - 1.9.0: raise the warning to an error

Prohibit cycles in annotation parameter types

> **Issue**: KT-47932
>
> **Component**: Core language
>
> **Incompatible change type**: source
>
> **Short summary**: Kotlin 1.9 prohibits the type of an annotation being used as one of its parameter types, either directly or indirectly. This prevents cycles from being created. However, you are allowed to have parameter types that are an `Array` or a `vararg` of the annotation type.
>
> **Deprecation cycle**:
>
> - 1.7.0: report a warning (or an error in progressive mode) on cycles in types of annotation parameters
> - 1.9.0: raise the warning to an error, `-XXLanguage:-ProhibitCyclesInAnnotations` can be used to temporarily revert to pre-1.9 behavior

Prohibit use of @ExtensionFunctionType annotation on function types with no parameters

> **Issue**: KT-43527
>
> **Component**: Core language

> **Incompatible change type**: source
>
> **Short summary**: Kotlin 1.9 prohibits using the `@ExtensionFunctionType` annotation on function types with no parameters, or on types that aren't function types.
>
> **Deprecation cycle**:
>
> - 1.7.0: report a warning for annotations on types that aren't function types, report an error for annotations on types that **are** function types
> - 1.9.0: raise the warning for function types to an error

Prohibit Java field type mismatch on assignment

> **Issue**: KT-48994
>
> **Component**: Kotlin/JVM
>
> **Incompatible change type**: source
>
> **Short summary**: Kotlin 1.9 reports a compiler error if it detects that the type of a value assigned to a Java field doesn't match the Java field's projected type.
>
> **Deprecation cycle**:
>
> - 1.6.0: report a warning (or an error in the progressive mode) when a projected Java field type doesn't match the assigned value type
> - 1.9.0: raise the warning to an error, `-XXLanguage:-RefineTypeCheckingOnAssignmentsToJavaFields` can be used to temporarily revert to pre-1.9 behavior

No source code excerpts in platform-type nullability assertion exceptions

> **Issue**: KT-57570
>
> **Component**: Kotlin/JVM
>
> **Incompatible change type**: behavioral
>
> **Short summary**: In Kotlin 1.9, exception messages for expression null checks do not include source code excerpts. Instead, the name of the method or field is displayed. If the expression is not a method or field, there is no additional information provided in the message.
>
> **Deprecation cycle**:
>
> - < 1.9.0: exception messages generated by expression null checks contain source code excerpts

> - 1.9.0: exception messages generated by expression null checks contain method or field names only, `-XXLanguage:-NoSourceCodeInNotNullAssertionExceptions` can be used to temporarily revert to pre-1.9 behavior

Prohibit the delegation of super calls to an abstract superclass member

> **Issues**: KT-45508, KT-49017, KT-38078
>
> **Component**: Core language
>
> **Incompatible change type**: source
>
> **Short summary**: Kotlin will report a compile error when an explicit or implicit super call is delegated to an *abstract* member of the superclass, even if there's a default implementation in a super interface.
>
> **Deprecation cycle**:
>
> - 1.5.20: introduce a warning when non-abstract classes that do not override all abstract members are used
> - 1.7.0: report a warning if a super call, in fact, accesses an abstract member from a superclass
> - 1.7.0: report an error in all affected cases if the `-Xjvm-default=all` or `-Xjvm-default=all-compatibility` compatibility modes are enabled; report an error in the progressive mode
> - 1.8.0: report an error in cases of declaring a concrete class with a non-overridden abstract method from the superclass, and super calls of `Any` methods are overridden as abstract in the superclass
> - 1.9.0: report an error in all affected cases, including explicit super calls to an abstract method from the super class

Deprecate confusing grammar in when-with-subject

> **Issue**: KT-48385
>
> **Component**: Core language
>
> **Incompatible change type**: source
>
> **Short summary**: Kotlin 1.6 deprecated several confusing grammar constructs in `when` condition expressions.
>
> **Deprecation cycle**:

- 1.6.20: introduce a deprecation warning on the affected expressions
- 1.8.0: raise this warning to an error, `-XXLanguage:-ProhibitConfusingSyntaxInWhenBranches` can be used to temporarily revert to the pre-1.8 behavior
- >= 2.1: repurpose some deprecated constructs for new language features

Prevent implicit coercions between different numeric types

Issue: KT-48645

Component: Kotlin/JVM

Incompatible change type: behavioral

Short summary: Kotlin will avoid converting numeric values automatically to a primitive numeric type where only a downcast to that type is needed semantically.

Deprecation cycle:

- < 1.5.30: the old behavior in all affected cases
- 1.5.30: fix the downcast behavior in generated property delegate accessors, `-Xuse-old-backend` can be used to temporarily revert to the pre-1.5.30 fix behavior
- >= 2.0: fix the downcast behavior in other affected cases

Prohibit upper bound violation in a generic type alias usage (a type parameter used in a generic type argument of a type argument of the aliased type)

Issue: KT-54066

Component: Core language

Incompatible change type: source

Short summary: Kotlin will prohibit using a type alias with type arguments that violate the upper bound restrictions of the corresponding type parameters of the aliased type in case when the typealias type parameter is used as a generic type argument of a type argument of the aliased type, for example, `typealias Alias<T> = Base<List<T>>`.

Deprecation cycle:

- 1.8.0: report a warning when a generic typealias usage has type arguments violating upper bound constraints of the corresponding type parameters of the aliased type
- 2.0.0: raise the warning to an error

Keep nullability when approximating local types in public signatures

> **Issue**: KT-53982
>
> **Component**: Core language
>
> **Incompatible change type**: source, binary
>
> **Short summary**: when a local or anonymous type is returned from an expression-body function without an explicitly specified return type, Kotlin compiler infers (or approximates) the return type using the known supertype of that type. During this, the compiler can infer a non-nullable type where the null value could in fact be returned.
>
> **Deprecation cycle**:
>
> - 1.8.0: approximate flexible types by flexible supertypes
> - 1.8.0: report a warning when a declaration is inferred to have a non-nullable type that should be nullable, prompting users to specify the type explicitly
> - 2.0.0: approximate nullable types by nullable supertypes, `-XXLanguage:-KeepNullabilityWhenApproximatingLocalType` can be used to temporarily revert to the pre-2.0 behavior

Do not propagate deprecation through overrides

> **Issue**: KT-47902
>
> **Component**: Core language
>
> **Incompatible change type**: source
>
> **Short summary**: Kotlin 1.9 will no longer propagate deprecation from a deprecated member in the superclass to its overriding member in the subclass, thus providing an explicit mechanism for deprecating a member of the superclass while leaving it non-deprecated in the subclass.
>
> **Deprecation cycle**:
>
> - 1.6.20: reporting a warning with the message of the future behavior change and a prompt to either suppress this warning or explicitly write a `@Deprecated` annotation on an override of a deprecated member
> - 1.9.0: stop propagating deprecation status to the overridden members. This change also takes effect immediately in the progressive mode

Prohibit using collection literals in annotation classes anywhere except their parameters declaration

> **Issue**: KT-39041
>
> **Component**: Core language
>
> **Incompatible change type**: source
>
> **Short summary**: Kotlin allows using collection literals in a restricted way - for passing arrays to parameters of annotation classes or specifying default values for these parameters. However besides that, Kotlin allowed using collections literals anywhere else inside an annotation class, for example, in its nested object. Kotlin 1.9 will prohibit using collection literals in annotation classes anywhere except their parameters' default values.
>
> **Deprecation cycle**:
>
> - 1.7.0: report a warning (or an error in the progressive mode) on array literals in nested objects in annotation classes
> - 1.9.0: raise the warning to an error

Prohibit forward referencing of parameters in default value expressions

> **Issue**: KT-25694
>
> **Component**: Core language
>
> **Incompatible change type**: source
>
> **Short summary**: Kotlin 1.9 will prohibit forward referencing of parameters in default value expressions of other parameters. This ensures that by the time the parameter is accessed in a default value expression, it would already have a value either passed to the function or initialized by its own default value expression.
>
> **Deprecation cycle**:
>
> - 1.7.0: report a warning (or an error in the progressive mode) when a parameter with default value is references in default value of another parameter that comes before it
> - 1.9.0: raise the warning to an error, `-XXLanguage:-ProhibitIllegalValueParameterUsageInDefaultArguments` can be used to temporarily revert to the pre-1.9 behavior

Prohibit extension calls on inline functional parameters

> **Issue**: KT-52502
>
> **Component**: Core language

> **Incompatible change type**: source
>
> **Short summary**: while Kotlin allowed passing an inline functional parameter to another inline function as a receiver, it always resulted in compiler exceptions when compiling such code. Kotlin 1.9 will prohibit this, thus reporting an error instead of crashing the compiler.
>
> **Deprecation cycle**:
>
> - 1.7.20: report a warning (or an error in the progressive mode) for inline extension calls on inline functional parameters
> - 1.9.0: raise the warning to an error

Prohibit calls to infix functions named suspend with an anonymous function argument

> **Issue**: KT-49264
>
> **Component**: Core language
>
> **Incompatible change type**: source
>
> **Short summary**: Kotlin 1.9 will no longer allow calling infix functions named `suspend` that have a single argument of a functional type passed as an anonymous function literal.
>
> **Deprecation cycle**:
>
> - 1.7.20: report a warning on suspend infix calls with an anonymous function literal
> - 1.9.0: raise the warning to an error, `-XXLanguage:-ModifierNonBuiltinSuspendFunError` can be used to temporarily revert to the pre-1.9 behavior
> - TODO: Change how the `suspend fun` token sequence is interpreted by the parser

Prohibit using captured type parameters in inner classes against their variance

> **Issue**: KT-50947
>
> **Component**: Core language
>
> **Incompatible change type**: source
>
> **Short summary**: Kotlin 1.9 will prohibit using type parameters of an outer class having `in` or `out` variance in an inner class of that class in positions violating that type parameters' declared variance.
>
> **Deprecation cycle**:
>
> - 1.7.0: report a warning (or an error in the progressive mode) when an outer class' type

> parameter usage position violates the variance rules of that parameter
> - 1.9.0: raise the warning to an error, `-XXLanguage:-ReportTypeVarianceConflictOnQualifierArguments` can be used to temporarily revert to the pre-1.9 behavior

Prohibit recursive call of a function without explicit return type in compound assignment operators

> **Issue**: KT-48546
>
> **Component**: Core language
>
> **Incompatible change type**: source
>
> **Short summary**: Kotlin 1.9 will prohibit calling a function without explicitly specified return type in an argument of a compound assignment operator inside that function's body, as it currently does in other expressions inside the body of that function.
>
> **Deprecation cycle**:
>
> - 1.7.0: report a warning (or an error in the progressive mode) when a function without explicitly specified return type is called recursively in that function's body in a compound assignment operator argument
> - 1.9.0: raise the warning to an error

Prohibit unsound calls with expected @NotNull T and given Kotlin generic parameter with nullable bound

> **Issue**: KT-36770
>
> **Component**: Kotlin/JVM
>
> **Incompatible change type**: source
>
> **Short summary**: Kotlin 1.9 will prohibit method calls where a value of a potentially nullable generic type is passed for a `@NotNull`-annotated parameter of a Java method.
>
> **Deprecation cycle**:
>
> - 1.5.20: report a warning when an unconstrained generic type parameter is passed where a non-nullable type is expected
> - 1.9.0: report a type mismatch error instead of the warning above,

> `-XXLanguage:-ProhibitUsingNullableTypeParameterAgainstNotNullAnnotated` can be used to temporarily revert to the pre-1.8 behavior

Prohibit access to members of a companion of an enum class from entry initializers of this enum

> **Issue**: KT-49110
>
> **Component**: Core language
>
> **Incompatible change type**: source
>
> **Short summary**: Kotlin 1.9 will prohibit all kinds of access to the companion object of an enum from an enum entry initializer.
>
> **Deprecation cycle**:
>
> - 1.6.20: report a warning (or an error in the progressive mode) on such companion member access
> - 1.9.0: raise the warning to an error, `-XXLanguage:-ProhibitAccessToEnumCompanionMembersInEnumConstructorCall` can be used to temporarily revert to the pre-1.8 behavior

Deprecate and remove Enum.declaringClass synthetic property

> **Issue**: KT-49653
>
> **Component**: Kotlin/JVM
>
> **Incompatible change type**: source
>
> **Short summary**: Kotlin allowed using the synthetic property `declaringClass` on `Enum` values produced from the method `getDeclaringClass()` of the underlying Java class `java.lang.Enum` even though this method is not available for Kotlin `Enum` type. Kotlin 1.9 will prohibit using this property, proposing to migrate to the extension property `declaringJavaClass` instead.
>
> **Deprecation cycle**:
>
> - 1.7.0: report a warning (or an error in the progressive mode) on `declaringClass` property usages, propose the migration to `declaringJavaClass` extension
> - 1.9.0: raise the warning to an error, `-XXLanguage:-ProhibitEnumDeclaringClass` can be used to temporarily revert to the pre-1.9 behavior
> - 2.0.0: remove `declaringClass` synthetic property

Deprecate enable and compatibility modes of the compiler option -Xjvm-default

> **Issues**: KT-46329, KT-54746
>
> **Component**: Kotlin/JVM
>
> **Incompatible change type**: source
>
> **Short summary**: Kotlin 1.9 prohibits using the `enable` and `compatibility` modes of the `-Xjvm-default` compiler option.
>
> **Deprecation cycle**:
>
> - 1.6.20: introduce a warning on the `enable` and `compatibility` modes of the `-Xjvm-default` compiler option
> - 1.9.0: raise this warning to an error

Prohibit implicit inferring a type variable into an upper bound in the builder inference context

> **Issue**: KT-47986
>
> **Component**: Core language
>
> **Incompatible change type**: source
>
> **Short summary**: Kotlin 2.0 will prohibit inferring a type variable into the corresponding type parameter's upper bound in the absence of any use-site type information in the scope of builder inference lambda functions, the same way as it does currently in other contexts.
>
> **Deprecation cycle**:
>
> - 1.7.20: report a warning (or an error in the progressive mode) when a type parameter is inferred into declared upper bounds in the absence of use-site type information
> - 2.0.0: raise the warning to an error

Standard library

Warn about potential overload resolution change when Range/Progression starts implementing Collection

> **Issue**: KT-49276
>
> **Component**: Core language / kotlin-stdlib

> **Incompatible change type**: source
>
> **Short summary**: it is planned to implement the `Collection` interface in the standard progressions and concrete ranges inherited from them in Kotlin 1.9. This could make a different overload selected in the overload resolution if there are two overloads of some method, one accepting an element and another accepting a collection. Kotlin will make this situation visible by reporting a warning or an error when such overloaded method is called with a range or progression argument.
>
> **Deprecation cycle**:
>
> - 1.6.20: report a warning when an overloaded method is called with the standard progression or its range inheritor as an argument if implementing the `Collection` interface by this progression/range leads to another overload being selected in this call in future
> - 1.8.0: raise this warning to an error
> - 2.1.0: stop reporting the error, implement `Collection` interface in progressions thus changing the overload resolution result in the affected cases

Migrate declarations from kotlin.dom and kotlin.browser packages to kotlinx.*

> **Issue**: KT-39330
>
> **Component**: kotlin-stdlib (JS)
>
> **Incompatible change type**: source
>
> **Short summary**: declarations from the `kotlin.dom` and `kotlin.browser` packages are moved to the corresponding `kotlinx.*` packages to prepare for extracting them from stdlib.
>
> **Deprecation cycle**:
>
> - 1.4.0: introduce the replacement API in `kotlinx.dom` and `kotlinx.browser` packages
> - 1.4.0: deprecate the API in `kotlin.dom` and `kotlin.browser` packages and propose the new API above as a replacement
> - 1.6.0: raise the deprecation level to an error
> - 1.8.20: remove the deprecated functions from stdlib for JS-IR target
> - >= 2.0: move the API in kotlinx.* packages to a separate library

Deprecate some JS-only API

> **Issue**: KT-48587
>
> **Component**: kotlin-stdlib (JS)

> **Incompatible change type**: source
>
> **Short summary**: a number of JS-only functions in stdlib are deprecated for removal. They include: `String.concat(String)`, `String.match(regex: String)`, `String.matches(regex: String)`, and the `sort` functions on arrays taking a comparison function, for example, `Array<out T>.sort(comparison: (a: T, b: T) -> Int)`.
>
> **Deprecation cycle**:
>
> - 1.6.0: deprecate the affected functions with a warning
> - 1.9.0: raise the deprecation level to an error
> - >=2.0: remove the deprecated functions from the public API

Tools

Remove enableEndorsedLibs flag from Gradle setup

> **Issue**: KT-54098
>
> **Component**: Gradle
>
> **Incompatible change type**: source
>
> **Short summary**: the `enableEndorsedLibs` flag is no longer supported in Gradle setup.
>
> **Deprecation cycle**:
>
> - < 1.9.0: `enableEndorsedLibs` flag is supported in Gradle setup
> - 1.9.0: `enableEndorsedLibs` flag is **not** supported in Gradle setup

Remove Gradle conventions

> **Issue**: KT-52976
>
> **Component**: Gradle
>
> **Incompatible change type**: source
>
> **Short summary**: Gradle conventions were deprecated in Gradle 7.1 and have been removed in Gradle 8.
>
> **Deprecation cycle**:
>
> - 1.7.20: Gradle conventions deprecated
> - 1.9.0: Gradle conventions removed

Remove classpath property of KotlinCompile task

> **Issue**: KT-53748
>
> **Component**: Gradle
>
> **Incompatible change type**: source
>
> **Short summary**: the `classpath` property of the `KotlinCompile` task is removed.
>
> **Deprecation cycle**:
>
> - 1.7.0: the `classpath` property is deprecated
> - 1.8.0: raise the deprecation level to an error
> - 1.9.0: remove the deprecated functions from the public API

Deprecate kotlin.internal.single.build.metrics.file property

> **Issue**: KT-53357
>
> **Component**: Gradle
>
> **Incompatible change type**: source
>
> **Short summary**: deprecate the `kotlin.internal.single.build.metrics.file` property used to define a single file for build reports. Use the property `kotlin.build.report.single_file` instead with `kotlin.build.report.output=single_file`.
>
> **Deprecation cycle:**
>
> - 1.8.0: raise the deprecation level to a warning
> - >= 1.9: delete the property

Kotlin 1.8 的兼容性指南

Keeping the Language Modern and *Comfortable Updates* are among the fundamental principles in Kotlin Language Design. The former says that constructs which obstruct language evolution should be removed, and the latter says that this removal should be well-communicated beforehand to make code migration as smooth as possible.

While most of the language changes were already announced through other channels, like update changelogs or compiler warnings, this document summarizes them all, providing a complete reference for migration from Kotlin 1.7 to Kotlin 1.8.

Basic terms

In this document we introduce several kinds of compatibility:

- *source*: source-incompatible change stops code that used to compile fine (without errors or warnings) from compiling anymore
- *binary*: two binary artifacts are said to be binary-compatible if interchanging them doesn't lead to loading or linkage errors
- *behavioral*: a change is said to be behavioral-incompatible if the same program demonstrates different behavior before and after applying the change

Remember that those definitions are given only for pure Kotlin. Compatibility of Kotlin code from the other languages perspective (for example, from Java) is out of the scope of this document.

Language

Prohibit the delegation of super calls to an abstract superclass member

> **Issues**: KT-45508, KT-49017, KT-38078
>
> **Component**: Core language
>
> **Incompatible change type**: source
>
> **Short summary**: Kotlin will report a compile error when an explicit or implicit super call is delegated to an *abstract* member of the superclass, even if there's a default implementation in a super interface

> **Deprecation cycle**:
>
> - 1.5.20: introduce a warning when non-abstract classes that do not override all abstract members are used
> - 1.7.0: report a warning if a super call, in fact, accesses an abstract member from a superclass
> - 1.7.0: report an error in all affected cases if the `-Xjvm-default=all` or `-Xjvm-default=all-compatibility` compatibility modes are enabled; report an error in the progressive mode
> - 1.8.0: report an error in cases of declaring a concrete class with a non-overridden abstract method from the superclass, and super calls of `Any` methods are overridden as abstract in the superclass
> - 1.9.0: report an error in all affected cases, including explicit super calls to an abstract method from the super class

Deprecate confusing grammar in when-with-subject

> **Issue**: KT-48385
>
> **Component**: Core language
>
> **Incompatible change type**: source
>
> **Short summary**: Kotlin 1.6 deprecated several confusing grammar constructs in `when` condition expressions
>
> **Deprecation cycle**:
>
> - 1.6.20: introduce a deprecation warning on the affected expressions
> - 1.8.0: raise this warning to an error, `-XXLanguage:-ProhibitConfusingSyntaxInWhenBranches` can be used to temporarily revert to the pre-1.8 behavior
> - >= 1.9: repurpose some deprecated constructs for new language features

Prevent implicit coercions between different numeric types

> **Issue**: KT-48645
>
> **Component**: Kotlin/JVM
>
> **Incompatible change type**: behavioral
>
> **Short summary**: Kotlin will avoid converting numeric values automatically to a primitive numeric type where only a downcast to that type is needed semantically

> **Deprecation cycle**:
>
> - < 1.5.30: the old behavior in all affected cases
> - 1.5.30: fix the downcast behavior in generated property delegate accessors, `-Xuse-old-backend` can be used to temporarily revert to the pre-1.5.30 fix behavior
> - >= 1.9: fix the downcast behavior in other affected cases

Make private constructors of sealed classes really private

> **Issue**: KT-44866
>
> **Component**: Core language
>
> **Incompatible change type**: source
>
> **Short summary**: after relaxing restrictions on where the inheritors of sealed classes could be declared in the project structure, the default visibility of sealed class constructors became protected. However, until 1.8, Kotlin still allowed calling explicitly declared private constructors of sealed classes outside those classes' scopes
>
> **Deprecation cycle**:
>
> - 1.6.20: report a warning (or an error in the progressive mode) when a private constructor of a sealed class is called outside that class
> - 1.8.0: use default visibility rules for private constructors (a call to a private constructor can be resolved only if this call is inside the corresponding class), the old behavior can be brought back temporarily by specifying the `-XXLanguage:-UseConsistentRulesForPrivateConstructorsOfSealedClasses` compiler argument

Prohibit using operator == on incompatible numeric types in builder inference context

> **Issue**: KT-45508
>
> **Component**: Core language
>
> **Incompatible change type**: source
>
> **Short summary**: Kotlin 1.8 will prohibit using the operator `==` on incompatible numeric types, for example, `Int` and `Long`, in scopes of builder inference lambda functions, the same way as it currently does in other contexts
>
> **Deprecation cycle**:
>
> - 1.6.20: report a warning (or an error in the progressive mode) when the operator `==` is used on incompatible numeric types

- 1.8.0: raise the warning to an error, `-XXLanguage:-ProperEqualityChecksInBuilderInferenceCalls` can be used to temporarily revert to the pre-1.8 behavior

Prohibit if without else and non-exhaustive when in right hand side of elvis operator

Issue: KT-44705

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.8 will prohibit using a non-exhaustive `when` or the `if` expression without an `else` branch on the right hand side of the Elvis operator (`?:`). Previously, it was allowed if the Elvis operator's result was not used as an expression

Deprecation cycle:

- 1.6.20: report a warning (or an error in the progressive mode) on such non-exhaustive if and when expressions
- 1.8.0: raise this warning to an error,

`-XXLanguage:-ProhibitNonExhaustiveIfInRhsOfElvis` can be used to temporarily revert to the pre-1.8 behavior

Prohibit upper bound violation in a generic type alias usage (one type parameter used in several type arguments of the aliased type)

Issues: KT-29168

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.8 will prohibit using a type alias with type arguments that violate the upper bound restrictions of the corresponding type parameters of the aliased type in case when one typealias type parameter is used in several type arguments of the aliased type, for example, `typealias Alias<T> = Base<T, T>`

Deprecation cycle:

- 1.7.0: report a warning (or an error in the progressive mode) on usages of a type alias with

type arguments violating upper bound constraints of the corresponding type parameters of the aliased type

- 1.8.0: raise this warning to an error, `-XXLanguage:-ReportMissingUpperBoundsViolatedErrorOnAbbreviationAtSupertypes` can be used to temporarily revert to the pre-1.8 behavior

Prohibit upper bound violation in a generic type alias usage (a type parameter used in a generic type argument of a type argument of the aliased type)

Issue: KT-54066

Component: Core language

Incompatible change type: source

Short summary: Kotlin will prohibit using a type alias with type arguments that violate the upper bound restrictions of the corresponding type parameters of the aliased type in case when the typealias type parameter is used as a generic type argument of a type argument of the aliased type, for example, `typealias Alias<T> = Base<List<T>>`

Deprecation cycle:

- 1.8.0: report a warning when a generic typealias usage has type arguments violating upper bound constraints of the corresponding type parameters of the aliased type
- >=1.10: raise the warning to an error

Prohibit using a type parameter declared for an extension property inside delegate

Issue: KT-24643

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.8 will prohibit delegating extension properties on a generic type to generic types that use the type parameter of the receiver in an unsafe way

Deprecation cycle:

- 1.6.0: report a warning (or an error in the progressive mode) when delegating an extension property to a type that uses type parameters inferred from the delegated property's type arguments in a particular way
- 1.8.0: raise the warning to an error, `-XXLanguage:-`

> `ForbidUsingExtensionPropertyTypeParameterInDelegate` can be used to temporarily revert to the pre-1.8 behavior

Forbid @Synchronized annotation on suspend functions

> **Issue**: KT-48516
>
> **Component**: Core language
>
> **Incompatible change type**: source
>
> **Short summary**: Kotlin 1.8 will prohibit placing the `@Synchronized` annotation on suspend functions because a suspending call should not be allowed to happen inside a synchronized block
>
> **Deprecation cycle**:
>
> - 1.6.0: report a warning on suspend functions annotated with the `@Synchronized` annotation, the warning is reported as an error in the progressive mode
> - 1.8.0: raise the warning to an error, `-XXLanguage:-SynchronizedSuspendError` can be used to temporarily revert to the pre-1.8 behavior

Prohibit using spread operator for passing arguments to non-vararg parameters

> **Issue**: KT-48162
>
> **Component**: Core language
>
> **Incompatible change type**: source
>
> **Short summary**: Kotlin allowed passing arrays with the spread operator (`*`) to non-vararg array parameters in certain conditions. Since Kotlin 1.8, this will be prohibited
>
> **Deprecation cycle**:
>
> - 1.6.0: report a warning (or an error in the progressive mode) on using the spread operator where a non-vararg array parameter is expected
> - 1.8.0: raise the warning to an error, `-XXLanguage:-ReportNonVarargSpreadOnGenericCalls` can be used to temporarily revert to the pre-1.8 behavior

Prohibit null-safety violation in lambdas passed to functions overloaded by lambda return type

> **Issue**: KT-49658

> **Component**: Core language
>
> **Incompatible change type**: source
>
> **Short summary**: Kotlin 1.8 will prohibit returning `null` from lambdas passed to functions overloaded by those lambdas' return types when overloads don't allow a nullable return type. Previously, it was allowed when `null` was returned from one of the branches of the `when` operator
>
> **Deprecation cycle**:
>
> - 1.6.20: report a type mismatch warning (or an error in the progressive mode)
> - 1.8.0: raise the warning to an error, `-XXLanguage:-DontLoseDiagnosticsDuringOverloadResolutionByReturnType` can be used to temporarily revert to the pre-1.8 behavior

Keep nullability when approximating local types in public signatures

> **Issue**: KT-53982
>
> **Component**: Core language
>
> **Incompatible change type**: source, binary
>
> **Short summary**: when a local or anonymous type is returned from an expression-body function without an explicitly specified return type, Kotlin compiler infers (or approximates) the return type using the known supertype of that type. During this, the compiler can infer a non-nullable type where the null value could in fact be returned
>
> **Deprecation cycle**:
>
> - 1.8.0: approximate flexible types by flexible supertypes
> - 1.8.0: report a warning when a declaration is inferred to have a non-nullable type that should be nullable, prompting users to specify the type explicitly
> - 1.9.0: approximate nullable types by nullable supertypes, `-XXLanguage:-KeepNullabilityWhenApproximatingLocalType` can be used to temporarily revert to the pre-1.9 behavior

Do not propagate deprecation through overrides

> **Issue**: KT-47902
>
> **Component**: Core language
>
> **Incompatible change type**: source

> **Short summary**: Kotlin 1.9 will no longer propagate deprecation from a deprecated member in the superclass to its overriding member in the subclass, thus providing an explicit mechanism for deprecating a member of the superclass while leaving it non-deprecated in the subclass
>
> **Deprecation cycle**:
>
> - 1.6.20: reporting a warning with the message of the future behavior change and a prompt to either suppress this warning or explicitly write a `@Deprecated` annotation on an override of a deprecated member
> - 1.9.0: stop propagating deprecation status to the overridden members. This change also takes effect immediately in the progressive mode

Prohibit implicit inferring a type variable into an upper bound in the builder inference context

> **Issue**: KT-47986
>
> **Component**: Core language
>
> **Incompatible change type**: source
>
> **Short summary**: Kotlin 1.9 will prohibit inferring a type variable into the corresponding type parameter's upper bound in the absence of any use-site type information in the scope of builder inference lambda functions, the same way as it does currently in other contexts
>
> **Deprecation cycle**:
>
> - 1.7.20: report a warning (or an error in the progressive mode) when a type parameter is inferred into declared upper bounds in the absence of use-site type information
> - 1.9.0: raise the warning to an error, `-XXLanguage:-ForbidInferringPostponedTypeVariableIntoDeclaredUpperBound` can be used to temporarily revert to the pre-1.9 behavior

Prohibit using collection literals in annotation classes anywhere except their parameters declaration

> **Issue**: KT-39041
>
> **Component**: Core language
>
> **Incompatible change type**: source
>
> **Short summary**: Kotlin allows using collection literals in a restricted way - for passing arrays to parameters of annotation classes or specifying default values for these parameters. However besides that, Kotlin allowed using collections literals anywhere else inside an annotation class,

> for example, in its nested object. Kotlin 1.9 will prohibit using collection literals in annotation classes anywhere except their parameters' default values.
>
> **Deprecation cycle**:
>
> - 1.7.0: report a warning (or an error in the progressive mode) on array literals in nested objects in annotation classes
> - 1.9.0: raise the warning to an error

Prohibit forward referencing of parameters with default values in default value expressions

> **Issue**: KT-25694
>
> **Component**: Core language
>
> **Incompatible change type**: source
>
> **Short summary**: Kotlin 1.9 will prohibit forward referencing of parameters with default values in default value expressions of other parameters. This ensures that by the time the parameter is accessed in a default value expression, it would already have a value either passed to the function or initialized by its own default value expression
>
> **Deprecation cycle**:
>
> - 1.7.0: report a warning (or an error in the progressive mode) when a parameter with default value is references in default value of another parameter that comes before it
> - 1.9.0: raise the warning to an error, `-XXLanguage:-ProhibitIllegalValueParameterUsageInDefaultArguments` can be used to temporarily revert to the pre-1.9 behavior

Prohibit extension calls on inline functional parameters

> **Issue**: KT-52502
>
> **Component**: Core language
>
> **Incompatible change type**: source
>
> **Short summary**: while Kotlin allowed passing an inline functional parameter to another inline function as a receiver, it always resulted in compiler exceptions when compiling such code. Kotlin 1.9 will prohibit this, thus reporting an error instead of crashing the compiler
>
> **Deprecation cycle**:
>
> - 1.7.20: report a warning (or an error in the progressive mode) for inline extension calls on

> inline functional parameters
> - 1.9.0: raise the warning to an error

Prohibit calls to infix functions named suspend with an anonymous function argument

> **Issue**: KT-49264
>
> **Component**: Core language
>
> **Incompatible change type**: source
>
> **Short summary**: Kotlin 1.9 will no longer allow calling infix functions named `suspend` that have a single argument of a functional type passed as an anonymous function literal
>
> **Deprecation cycle**:
>
> - 1.7.20: report a warning on suspend infix calls with an anonymous function literal
> - 1.9.0: raise the warning to an error, `-XXLanguage:-ModifierNonBuiltinSuspendFunError` can be used to temporarily revert to the pre-1.9 behavior
> - >=1.10: Change how the `suspend fun` token sequence is interpreted by the parser

Prohibit using captured type parameters in inner classes against their variance

> **Issue**: KT-50947
>
> **Component**: Core language
>
> **Incompatible change type**: source
>
> **Short summary**: Kotlin 1.9 will prohibit using type parameters of an outer class having `in` or `out` variance in an inner class of that class in positions violating that type parameters' declared variance
>
> **Deprecation cycle**:
>
> - 1.7.0: report a warning (or an error in the progressive mode) when an outer class' type parameter usage position violates the variance rules of that parameter
> - 1.9.0: raise the warning to an error, `-XXLanguage:-ReportTypeVarianceConflictOnQualifierArguments` can be used to temporarily revert to the pre-1.9 behavior

Prohibit recursive call of a function without explicit return type in compound assignment operators

> **Issue**: KT-48546
>
> **Component**: Core language
>
> **Incompatible change type**: source
>
> **Short summary**: Kotlin 1.9 will prohibit calling a function without explicitly specified return type in an argument of a compound assignment operator inside that function's body, as it currently does in other expressions inside the body of that function
>
> **Deprecation cycle**:
>
> - 1.7.0: report a warning (or an error in the progressive mode) when a function without explicitly specified return type is called recursively in that function's body in a compound assignment operator argument
> - 1.9.0: raise the warning to an error

Prohibit unsound calls with expected @NotNull T and given Kotlin generic parameter with nullable bound

> **Issue**: KT-36770
>
> **Component**: Kotlin/JVM
>
> **Incompatible change type**: source
>
> **Short summary**: Kotlin 1.9 will prohibit method calls where a value of a potentially nullable generic type is passed for a `@NotNull`-annotated parameter of a Java method
>
> **Deprecation cycle**:
>
> - 1.5.20: report a warning when an unconstrained generic type parameter is passed where a non-nullable type is expected
> - 1.9.0: report a type mismatch error instead of the warning above,
>
>
>
> `-XXLanguage:-ProhibitUsingNullableTypeParameterAgainstNotNullAnnotated` can be used to temporarily revert to the pre-1.8 behavior

Prohibit access to members of a companion of an enum class from entry initializers of this enum

> **Issue**: KT-49110
>
> **Component**: Core language

> **Incompatible change type**: source
>
> **Short summary**: Kotlin 1.9 will prohibit all kinds of access to the companion object of an enum from an enum entry initializer
>
> **Deprecation cycle**:
>
> - 1.6.20: report a warning (or an error in the progressive mode) on such companion member access
> - 1.9.0: raise the warning to an error, `-XXLanguage:-ProhibitAccessToEnumCompanionMembersInEnumConstructorCall` can be used to temporarily revert to the pre-1.8 behavior

Deprecate and remove Enum.declaringClass synthetic property

> **Issue**: KT-49653
>
> **Component**: Kotlin/JVM
>
> **Incompatible change type**: source
>
> **Short summary**: Kotlin allowed using the synthetic property `declaringClass` on `Enum` values produced from the method `getDeclaringClass()` of the underlying Java class `java.lang.Enum` even though this method is not available for Kotlin `Enum` type. Kotlin 1.9 will prohibit using this property, proposing to migrate to the extension property `declaringJavaClass` instead
>
> **Deprecation cycle**:
>
> - 1.7.0: report a warning (or an error in the progressive mode) on `declaringClass` property usages, propose the migration to `declaringJavaClass` extension
> - 1.9.0: raise the warning to an error, `-XXLanguage:-ProhibitEnumDeclaringClass` can be used to temporarily revert to the pre-1.9 behavior
> - >=1.10: remove `declaringClass` synthetic property

Deprecate the enable and the compatibility modes of the compiler option -Xjvm-default

> **Issue**: KT-46329
>
> **Component**: Kotlin/JVM
>
> **Incompatible change type**: source
>
> **Short summary**: Kotlin 1.6.20 warns about the usage of the `enable` and `compatibility` modes of the `-Xjvm-default` compiler option

> **Deprecation cycle**:
>
> - 1.6.20: introduce a warning on the `enable` and `compatibility` modes of the `-Xjvm-default` compiler option
> - >= 1.9: raise this warning to an error

Standard library

Warn about potential overload resolution change when Range/Progression starts implementing Collection

> **Issue**: KT-49276
>
> **Component**: Core language / kotlin-stdlib
>
> **Incompatible change type**: source
>
> **Short summary**: it is planned to implement the `Collection` interface in the standard progressions and concrete ranges inherited from them in Kotlin 1.9. This could make a different overload selected in the overload resolution if there are two overloads of some method, one accepting an element and another accepting a collection. Kotlin will make this situation visible by reporting a warning or an error when such overloaded method is called with a range or progression argument
>
> **Deprecation cycle**:
>
> - 1.6.20: report a warning when an overloaded method is called with the standard progression or its range inheritor as an argument if implementing the `Collection` interface by this progression/range leads to another overload being selected in this call in future
> - 1.8.0: raise this warning to an error
> - 1.9.0: stop reporting the error, implement `Collection` interface in progressions thus changing the overload resolution result in the affected cases

Migrate declarations from kotlin.dom and kotlin.browser packages to kotlinx.*

> **Issue**: KT-39330
>
> **Component**: kotlin-stdlib (JS)
>
> **Incompatible change type**: source
>
> **Short summary**: declarations from the `kotlin.dom` and `kotlin.browser` packages are moved to the corresponding `kotlinx.*` packages to prepare for extracting them from stdlib

> **Deprecation cycle**:
>
> - 1.4.0: introduce the replacement API in `kotlinx.dom` and `kotlinx.browser` packages
> - 1.4.0: deprecate the API in `kotlin.dom` and `kotlin.browser` packages and propose the new API above as a replacement
> - 1.6.0: raise the deprecation level to an error
> - 1.8.20: remove the deprecated functions from stdlib for JS-IR target
> - >= 1.9: move the API in kotlinx.* packages to a separate library

Deprecate some JS-only API

> **Issue**: KT-48587
>
> **Component**: kotlin-stdlib (JS)
>
> **Incompatible change type**: source
>
> **Short summary**: a number of JS-only functions in stdlib are deprecated for removal. They include: `String.concat(String)`, `String.match(regex: String)`, `String.matches(regex: String)`, and the `sort` functions on arrays taking a comparison function, for example, `Array<out T>.sort(comparison: (a: T, b: T) -> Int)`
>
> **Deprecation cycle**:
>
> - 1.6.0: deprecate the affected functions with a warning
> - 1.9.0: raise the deprecation level to an error
> - >=1.10.0: remove the deprecated functions from the public API

Tools

Raise deprecation level of classpath property of KotlinCompile task

> **Issue**: KT-51679
>
> **Component**: Gradle
>
> **Incompatible change type**: source
>
> **Short summary**: the `classpath` property of the `KotlinCompile` task is deprecated
>
> **Deprecation cycle**:
>
> - 1.7.0: the `classpath` property is deprecated
> - 1.8.0: raise the deprecation level to an error

> - >=1.9.0: remove the deprecated functions from the public API

Remove kapt.use.worker.api Gradle property

> **Issue**: KT-48827
>
> **Component**: Gradle
>
> **Incompatible change type**: behavioral
>
> **Short summary**: remove the `kapt.use.worker.api` property that allowed to run kapt via Gradle Workers API (default: true)
>
> **Deprecation cycle**:
>
> - 1.6.20: raise the deprecation level to a warning
> - 1.8.0: remove this property

Remove kotlin.compiler.execution.strategy system property

> **Issue**: KT-51831
>
> **Component**: Gradle
>
> **Incompatible change type**: behavioral
>
> **Short summary**: remove the `kotlin.compiler.execution.strategy` system property used to choose a compiler execution strategy. Use the Gradle property `kotlin.compiler.execution.strategy` or the compile task property `compilerExecutionStrategy` instead
>
> **Deprecation cycle:**
>
> - 1.7.0: raise the deprecation level to a warning
> - 1.8.0: remove the property

Changes in compiler options

> **Issues**: KT-27301, KT-48532
>
> **Component**: Gradle
>
> **Incompatible change type**: source, binary
>
> **Short summary**: this change might affect Gradle plugins authors. In `kotlin-gradle-plugin`, there are additional generic parameters to some internal types (you should add generic types or `*`). `KotlinNativeLink` task does not inherit the `AbstractKotlinNativeCompile` task

> anymore. `KotlinJsCompilerOptions.outputFile` and the related `KotlinJsOptions.outputFile` options are deprecated. Use the `Kotlin2JsCompile.outputFileProperty` task input instead. The `kotlinOptions` task input and the `kotlinOptions{...}` task DSL are in a support mode and will be deprecated in upcoming releases. `compilerOptions` and `kotlinOptions` can not be changed on a task execution phase (see one exception in What's new in Kotlin 1.8). `freeCompilerArgs` returns an immutable `List<String>` – `kotlinOptions.freeCompilerArgs.remove("something")` will fail. The `useOldBackend` property that allowed to use the old JVM backend is removed
>
> **Deprecation cycle:**
>
> - 1.8.0: `KotlinNativeLink` task does not inherit the `AbstractKotlinNativeCompile`. `KotlinJsCompilerOptions.outputFile` and the related `KotlinJsOptions.outputFile` options are deprecated. The `useOldBackend` property that allowed to use the old JVM backend is removed.

Deprecate kotlin.internal.single.build.metrics.file property

> **Issue**: KT-53357
>
> **Component**: Gradle
>
> **Incompatible change type**: source
>
> **Short summary**: deprecate the `kotlin.internal.single.build.metrics.file` property used to define a single file for build reports. Use the property `kotlin.build.report.single_file` instead with `kotlin.build.report.output=single_file`
>
> **Deprecation cycle:**
>
> - 1.8.0: raise the deprecation level to a warning >= 1.9: delete the property

Kotlin 1.7.20 的兼容性指南

Keeping the Language Modern and *Comfortable Updates* are among the fundamental principles in Kotlin Language Design. The former says that constructs which obstruct language evolution should be removed, and the latter says that this removal should be well-communicated beforehand to make code migration as smooth as possible.

Usually incompatible changes happen only in feature releases, but this time we have to introduce two such changes in an incremental release to limit spread of the problems introduced by changes in Kotlin 1.7.

This document summarizes them, providing a reference for migration from Kotlin 1.7.0 and 1.7.10 to Kotlin 1.7.20.

Basic terms

In this document we introduce several kinds of compatibility:

- *source*: source-incompatible change stops code that used to compile fine (without errors or warnings) from compiling anymore
- *binary*: two binary artifacts are said to be binary-compatible if interchanging them doesn't lead to loading or linkage errors
- *behavioral*: a change is said to be behavioral-incompatible if the same program demonstrates different behavior before and after applying the change

Remember that those definitions are given only for pure Kotlin. Compatibility of Kotlin code from the other languages perspective (for example, from Java) is out of the scope of this document.

Language

Rollback attempt to fix proper constraints processing

> **Issue**: KT-53813
>
> **Component**: Core language
>
> **Incompatible change type**: source

> **Short summary**: Rollback an attempt of fixing issues in type inference constraints processing appeared in 1.7.0 after implementing the change described in KT-52668. The attempt was made in 1.7.10, but it in turn introduced new problems.
>
> **Deprecation cycle**:
>
> - 1.7.20: Rollback to 1.7.0 behavior

Forbid some builder inference cases to avoid problematic interaction with multiple lambdas and resolution

> **Issue**: KT-53797
>
> **Component**: Core language
>
> **Incompatible change type**: source
>
> **Short summary**: Kotlin 1.7 introduced a feature called unrestricted builder inference, so that even the lambdas passed to parameters not annotated with `@BuilderInference` could benefit from the builder inference. However, that could cause several problems if more than one such lambda occurred in a function invocation.
>
> Kotlin 1.7.20 will report an error if more than one lambda function having the corresponding parameter not annotated with `@BuilderInference` requires using builder inference to complete inferring the types in the lambda.
>
> **Deprecation cycle**:
>
> - 1.7.20: report an error on such lambda functions,
>
>
>
> `-XXLanguage:+NoBuilderInferenceWithoutAnnotationRestriction` can be used to temporarily revert to the pre-1.7.20 behavior

Kotlin 1.7 的兼容性指南

Keeping the Language Modern and *Comfortable Updates* are among the fundamental principles in Kotlin Language Design. The former says that constructs which obstruct language evolution should be removed, and the latter says that this removal should be well-communicated beforehand to make code migration as smooth as possible.

While most of the language changes were already announced through other channels, like update changelogs or compiler warnings, this document summarizes them all, providing a complete reference for migration from Kotlin 1.6 to Kotlin 1.7.

Basic terms

In this document we introduce several kinds of compatibility:

- *source*: source-incompatible change stops code that used to compile fine (without errors or warnings) from compiling anymore
- *binary*: two binary artifacts are said to be binary-compatible if interchanging them doesn't lead to loading or linkage errors
- *behavioral*: a change is said to be behavioral-incompatible if the same program demonstrates different behavior before and after applying the change

Remember that those definitions are given only for pure Kotlin. Compatibility of Kotlin code from the other languages perspective (for example, from Java) is out of the scope of this document.

Language

Make safe call result always nullable

> **Issue**: KT-46860
>
> **Component**: Core language
>
> **Incompatible change type**: source
>
> **Short summary**: Kotlin 1.7 will consider the type of safe call result always nullable, even when the receiver of the safe call is non-nullable
>
> **Deprecation cycle**:
>
> - <1.3: report a warning on an unnecessary safe call on non-nullable receivers

- 1.6.20: warn additionally that the result of an unnecessary safe call will change its type in the next version
- 1.7.0: change the type of safe call result to nullable,

`-XXLanguage:-SafeCallsAreAlwaysNullable` can be used to temporarily revert to the pre-1.7 behavior

Prohibit the delegation of super calls to an abstract superclass member

Issues: KT-45508, KT-49017, KT-38078

Component: Core language

Incompatible change type: source

Short summary: Kotlin will report a compile error when an explicit or implicit super call is delegated to an *abstract* member of the superclass, even if there's a default implementation in a super interface

Deprecation cycle:

- 1.5.20: introduce a warning when non-abstract classes that do not override all abstract members are used
- 1.7.0: report an error if a super call, in fact, accesses an abstract member from a superclass
- 1.7.0: report an error if the `-Xjvm-default=all` or `-Xjvm-default=all-compatibility` compatibility modes are enabled; report an error in the progressive mode
- >=1.8.0: report an error in all cases

Prohibit exposing non-public types through public properties declared in a non-public primary constructor

Issue: KT-28078

Component: Core language

Incompatible change type: source

Short summary: Kotlin will prevent declaring public properties having non-public types in a private primary constructor. Accessing such properties from another package could lead to an `IllegalAccessError`

Deprecation cycle:

- 1.3.20: report a warning on a public property that has a non-public type and is declared in a non-public constructor
- 1.6.20: raise this warning to an error in the progressive mode
- 1.7.0: raise this warning to an error

Prohibit access to uninitialized enum entries qualified with the enum name

Issue: KT-41124

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.7 will prohibit access to uninitialized enum entries from the enum static initializer block when these entries are qualified with the enum name

Deprecation cycle:

- 1.7.0: report an error when uninitialized enum entries are accessed from the enum static initializer block

Prohibit computing constant values of complex boolean expressions in when condition branches and conditions of loops

Issue: KT-39883

Component: Core language

Incompatible change type: source

Short summary: Kotlin will no longer make exhaustiveness and control flow assumptions based on constant boolean expressions other than literal `true` and `false`

Deprecation cycle:

- 1.5.30: report a warning when exhaustiveness of `when` or control flow reachability is determined based on a complex constant boolean expression in `when` branch or loop condition
- 1.7.0: raise this warning to an error

Make when statements with enum, sealed, and Boolean subjects exhaustive by default

Issue: KT-47709

> **Component**: Core language
>
> **Incompatible change type**: source
>
> **Short summary**: Kotlin 1.7 will report an error about the `when` statement with an enum, sealed, or Boolean subject being non-exhaustive
>
> **Deprecation cycle**:
>
> - 1.6.0: introduce a warning when the `when` statement with an enum, sealed, or Boolean subject is non-exhaustive (error in the progressive mode)
> - 1.7.0: raise this warning to an error

Deprecate confusing grammar in when-with-subject

> **Issue**: KT-48385
>
> **Component**: Core language
>
> **Incompatible change type**: source
>
> **Short summary**: Kotlin 1.6 deprecated several confusing grammar constructs in `when` condition expressions
>
> **Deprecation cycle**:
>
> - 1.6.20: introduce a deprecation warning on the affected expressions
> - 1.8.0: raise this warning to an error
> - >= 1.8: repurpose some deprecated constructs for new language features

Type nullability enhancement improvements

> **Issue**: KT-48623
>
> **Component**: Kotlin/JVM
>
> **Incompatible change type**: source
>
> **Short summary**: Kotlin 1.7 will change how it loads and interprets type nullability annotations in Java code
>
> **Deprecation cycle**:
>
> - 1.4.30: introduce warnings for cases where more precise type nullability could lead to an error
> - 1.7.0: infer more precise nullability of Java types, `-XXLanguage:-TypeEnhancementImprovementsInStrictMode` can be used to temporarily revert to the pre-

> 1.7 behavior

Prevent implicit coercions between different numeric types

> **Issue**: KT-48645
>
> **Component**: Kotlin/JVM
>
> **Incompatible change type**: behavioral
>
> **Short summary**: Kotlin will avoid converting numeric values automatically to a primitive numeric type where only a downcast to that type was needed semantically
>
> **Deprecation cycle**:
>
> - < 1.5.30: the old behavior in all affected cases
> - 1.5.30: fix the downcast behavior in generated property delegate accessors, `-Xuse-old-backend` can be used to temporarily revert to the pre-1.5.30 fix behavior
> - >= 1.7.20: fix the downcast behavior in other affected cases

Deprecate the enable and the compatibility modes of the compiler option -Xjvm-default

> **Issue**: KT-46329
>
> **Component**: Kotlin/JVM
>
> **Incompatible change type**: source
>
> **Short summary**: Kotlin 1.6.20 warns about the usage of `enable` and `compatibility` modes of the `-Xjvm-default` compiler option
>
> **Deprecation cycle**:
>
> - 1.6.20: introduce a warning on the `enable` and `compatibility` modes of the `-Xjvm-default` compiler option
> - >= 1.8.0: raise this warning to an error

Prohibit calls to functions named suspend with a trailing lambda

> **Issue**: KT-22562
>
> **Component**: Core language
>
> **Incompatible change type**: source

> **Short summary**: Kotlin 1.6 no longer allows calling user functions named `suspend` that have the single argument of a functional type passed as a trailing lambda
>
> **Deprecation cycle**:
>
> - 1.3.0: introduce a warning on such function calls
> - 1.6.0: raise this warning to an error
> - 1.7.0: introduce changes to the language grammar so that `suspend` before `{` is parsed as a keyword

Prohibit smart cast on a base class property if the base class is from another module

> **Issue**: KT-52629
>
> **Component**: Core language
>
> **Incompatible change type**: source
>
> **Short summary**: Kotlin 1.7 will no longer allow smart casts on properties of a superclass if that class is located in another module
>
> **Deprecation cycle**:
>
> - 1.6.0: report a warning on a smart cast on a property declared in the superclass located in another module
> - 1.7.0: raise this warning to an error,
>
> `-XXLanguage:-ProhibitSmartcastsOnPropertyFromAlienBaseClass` can be used to temporarily revert to the pre-1.7 behavior

Do not neglect meaningful constraints during type inference

> **Issue**: KT-52668
>
> **Component**: Core language
>
> **Incompatible change type**: source
>
> **Short summary**: Kotlin 1.4−1.6 neglected some type constraints during type inference due to an incorrect optimization. It could allow writing unsound code, causing `ClassCastException` at runtime. Kotlin 1.7 takes these constraints into account, thus prohibiting the unsound code
>
> **Deprecation cycle**:

> - 1.5.20: report a warning on expressions where a type mismatch would happen if all the type inference constraints were taken into account
> - 1.7.0: take all the constraints into account, thus raising this warning to an error,
>
> `-XXLanguage:-ProperTypeInferenceConstraintsProcessing` can be used to temporarily revert to the pre-1.7 behavior

Standard library

Gradually change the return type of collection min and max functions to non-nullable

> **Issue**: KT-38854
>
> **Component**: kotlin-stdlib
>
> **Incompatible change type**: source
>
> **Short summary**: the return type of collection `min` and `max` functions will be changed to non-nullable in Kotlin 1.7
>
> **Deprecation cycle**:
>
> - 1.4.0: introduce `...OrNull` functions as synonyms and deprecate the affected API (see details in the issue)
> - 1.5.0: raise the deprecation level of the affected API to an error
> - 1.6.0: hide the deprecated functions from the public API
> - 1.7.0: reintroduce the affected API but with non-nullable return type

Deprecate floating-point array functions: contains, indexOf, lastIndexOf

> **Issue**: KT-28753
>
> **Component**: kotlin-stdlib
>
> **Incompatible change type**: source
>
> **Short summary**: Kotlin deprecates floating-point array functions `contains`, `indexOf`, `lastIndexOf` that compare values using the IEEE-754 order instead of the total order
>
> **Deprecation cycle**:

- 1.4.0: deprecate the affected functions with a warning
- 1.6.0: raise the deprecation level to an error
- 1.7.0: hide the deprecated functions from the public API

Migrate declarations from kotlin.dom and kotlin.browser packages to kotlinx.*

Issue: KT-39330

Component: kotlin-stdlib (JS)

Incompatible change type: source

Short summary: declarations from the `kotlin.dom` and `kotlin.browser` packages are moved to the corresponding `kotlinx.*` packages to prepare for extracting them from stdlib

Deprecation cycle:

- 1.4.0: introduce the replacement API in `kotlinx.dom` and `kotlinx.browser` packages
- 1.4.0: deprecate the API in `kotlin.dom` and `kotlin.browser` packages and propose the new API above as a replacement
- 1.6.0: raise the deprecation level to an error
- >= 1.8: remove the deprecated functions from stdlib
- >= 1.8: move the API in kotlinx.* packages to a separate library

Deprecate some JS-only API

Issue: KT-48587

Component: kotlin-stdlib (JS)

Incompatible change type: source

Short summary: a number of JS-only functions in stdlib are deprecated for removal. They include: `String.concat(String)`, `String.match(regex: String)`, `String.matches(regex: String)`, and the `sort` functions on arrays taking a comparison function, for example, `Array<out T>.sort(comparison: (a: T, b: T) -> Int)`

Deprecation cycle:

- 1.6.0: deprecate the affected functions with a warning
- 1.8.0: raise the deprecation level to an error
- 1.9.0: remove the deprecated functions from the public API

Tools

Remove KotlinGradleSubplugin class

> **Issue**: KT-48831
>
> **Component**: Gradle
>
> **Incompatible change type**: source
>
> **Short summary**: remove the `KotlinGradleSubplugin` class. Use the `KotlinCompilerPluginSupportPlugin` class instead
>
> **Deprecation cycle**:
>
> - 1.6.0: raise the deprecation level to an error
> - 1.7.0: remove the deprecated class

Remove useIR compiler option

> **Issue**: KT-48847
>
> **Component**: Gradle
>
> **Incompatible change type**: source
>
> **Short summary**: remove the deprecated and hidden `useIR` compiler option
>
> **Deprecation cycle**:
>
> - 1.5.0: raise the deprecation level to a warning
> - 1.6.0: hide the option
> - 1.7.0: remove the deprecated option

Deprecate kapt.use.worker.api Gradle property

> **Issue**: KT-48826
>
> **Component**: Gradle
>
> **Incompatible change type**: source
>
> **Short summary**: deprecate the `kapt.use.worker.api` property that allowed to run kapt via Gradle Workers API (default: true)
>
> **Deprecation cycle**:
>
> - 1.6.20: raise the deprecation level to a warning
> - >= 1.8.0: remove this property

Remove kotlin.experimental.coroutines Gradle DSL option and kotlin.coroutines Gradle property

> **Issue**: KT-50494
>
> **Component**: Gradle
>
> **Incompatible change type**: source
>
> **Short summary**: remove the `kotlin.experimental.coroutines` Gradle DSL option and the `kotlin.coroutines` property
>
> **Deprecation cycle**:
>
> - 1.6.20: raise the deprecation level to a warning
> - 1.7.0: remove the DSL option, its enclosing `experimental` block, and the property

Deprecate useExperimentalAnnotation compiler option

> **Issue**: KT-47763
>
> **Component**: Gradle
>
> **Incompatible change type**: source
>
> **Short summary**: remove the hidden `useExperimentalAnnotation()` Gradle function used to opt in to using an API in a module. `optIn()` function can be used instead
>
> **Deprecation cycle:**
>
> - 1.6.0: hide the deprecation option
> - 1.7.0: remove the deprecated option

Deprecate kotlin.compiler.execution.strategy system property

> **Issue**: KT-51830
>
> **Component**: Gradle
>
> **Incompatible change type**: source
>
> **Short summary**: deprecate the `kotlin.compiler.execution.strategy` system property used to choose a compiler execution strategy. Use the Gradle property `kotlin.compiler.execution.strategy` or the compile task property `compilerExecutionStrategy` instead
>
> **Deprecation cycle:**

> - 1.7.0: raise the deprecation level to a warning
> - > 1.7.0: remove the property

Remove kotlinOptions.jdkHome compiler option

> **Issue**: KT-46541
>
> **Component**: Gradle
>
> **Incompatible change type**: source
>
> **Short summary**: remove the `kotlinOptions.jdkHome` compiler option used to include a custom JDK from the specified location into the classpath instead of the default `JAVA_HOME`. Use Java toolchains instead
>
> **Deprecation cycle:**
>
> - 1.5.30: raise the deprecation level to a warning
> - > 1.7.0: remove the option

Remove noStdlib compiler option

> **Issue**: KT-49011
>
> **Component**: Gradle
>
> **Incompatible change type**: source
>
> **Short summary**: remove the `noStdlib` compiler option. The Gradle plugin uses the `kotlin.stdlib.default.dependency=true` property to control whether the Kotlin standard library is present
>
> **Deprecation cycle:**
>
> - 1.5.0: raise the deprecation level to a warning
> - 1.7.0: remove the option

Remove kotlin2js and kotlin-dce-plugin plugins

> **Issue**: KT-48276
>
> **Component**: Gradle
>
> **Incompatible change type**: source

> **Short summary**: remove the `kotlin2js` and `kotlin-dce-plugin` plugins. Instead of `kotlin2js`, use the new `org.jetbrains.kotlin.js` plugin. Dead code elimination (DCE) works when the Kotlin/JS Gradle plugin is properly configured

>

> **Deprecation cycle:**
>
> - 1.4.0: raise the deprecation level to a warning
> - 1.7.0: remove the plugins

Changes in compile tasks

> **Issue**: KT-32805
>
> **Component**: Gradle
>
> **Incompatible change type**: source
>
> **Short summary**: Kotlin compile tasks no longer inherit the Gradle `AbstractCompile` task and that's why the `sourceCompatibility` and `targetCompatibility` inputs are no longer available in Kotlin users' scripts. The `SourceTask.stableSources` input is no longer available. The `sourceFilesExtensions` input was removed. The deprecated `Gradle destinationDir: File` output was replaced with the `destinationDirectory: DirectoryProperty` output. The `classpath` property of the `KotlinCompile` task is deprecated
>
> **Deprecation cycle:**
>
> - 1.7.0: inputs are not available, the output is replaced, the `classpath` property is deprecated

Kotlin 1.6 的兼容性指南

Keeping the Language Modern and *Comfortable Updates* are among the fundamental principles in Kotlin Language Design. The former says that constructs which obstruct language evolution should be removed, and the latter says that this removal should be well-communicated beforehand to make code migration as smooth as possible.

While most of the language changes were already announced through other channels, like update changelogs or compiler warnings, this document summarizes them all, providing a complete reference for migration from Kotlin 1.5 to Kotlin 1.6.

Basic terms

In this document we introduce several kinds of compatibility:

- *source*: source-incompatible change stops code that used to compile fine (without errors or warnings) from compiling anymore
- *binary*: two binary artifacts are said to be binary-compatible if interchanging them doesn't lead to loading or linkage errors
- *behavioral*: a change is said to be behavioral-incompatible if the same program demonstrates different behavior before and after applying the change

Remember that those definitions are given only for pure Kotlin. Compatibility of Kotlin code from the other languages perspective (for example, from Java) is out of the scope of this document.

Language

Make when statements with enum, sealed, and Boolean subjects exhaustive by default

> **Issue**: KT-47709
>
> **Component**: Core language
>
> **Incompatible change type**: source
>
> **Short summary**: Kotlin 1.6 will warn about the `when` statement with an enum, sealed, or Boolean subject being non-exhaustive
>
> **Deprecation cycle**:

> - 1.6.0: introduce a warning when the `when` statement with an enum, sealed, or Boolean subject is non-exhaustive (error in the progressive mode)
> - 1.7.0: raise this warning to an error

Deprecate confusing grammar in when-with-subject

> **Issue**: KT-48385
>
> **Component**: Core language
>
> **Incompatible change type**: source
>
> **Short summary**: Kotlin 1.6 will deprecate several confusing grammar constructs in `when` condition expressions
>
> **Deprecation cycle**:
>
> - 1.6.20: introduce a deprecation warning on the affected expressions
> - 1.8.0: raise this warning to an error
> - >= 1.8: repurpose some deprecated constructs for new language features

Prohibit access to class members in the super constructor call of its companion and nested objects

> **Issue**: KT-25289
>
> **Component**: Core language
>
> **Incompatible change type**: source
>
> **Short summary**: Kotlin 1.6 will report an error for arguments of super constructor call of companion and regular objects if the receiver of such arguments refers to the containing declaration
>
> **Deprecation cycle**:
>
> - 1.5.20: introduce a warning on the problematic arguments
> - 1.6.0: raise this warning to an error, `-XXLanguage:-ProhibitSelfCallsInNestedObjects` can be used to temporarily revert to the pre-1.6 behavior

Type nullability enhancement improvements

> **Issue**: KT-48623
>
> **Component**: Kotlin/JVM

> **Incompatible change type**: source
>
> **Short summary**: Kotlin 1.7 will change how it loads and interprets type nullability annotations in Java code
>
> **Deprecation cycle**:
>
> - 1.4.30: introduce warnings for cases where more precise type nullability could lead to an error
> - 1.7.0: infer more precise nullability of Java types, `-XXLanguage:-TypeEnhancementImprovementsInStrictMode` can be used to temporarily revert to the pre-1.7 behavior

Prevent implicit coercions between different numeric types

> **Issue**: KT-48645
>
> **Component**: Kotlin/JVM
>
> **Incompatible change type**: behavioral
>
> **Short summary**: Kotlin will avoid converting numeric values automatically to a primitive numeric type where only a downcast to that type was needed semantically
>
> **Deprecation cycle**:
>
> - < 1.5.30: the old behavior in all affected cases
> - 1.5.30: fix the downcast behavior in generated property delegate accessors, `-Xuse-old-backend` can be used to temporarily revert to the pre-1.5.30 fix behavior
> - >= 1.6.20: fix the downcast behavior in other affected cases

Prohibit declarations of repeatable annotation classes whose container annotation violates JLS

> **Issue**: KT-47928
>
> **Component**: Kotlin/JVM
>
> **Incompatible change type**: source
>
> **Short summary**: Kotlin 1.6 will check that the container annotation of a repeatable annotation satisfies the same requirements as in JLS 9.6.3: array-typed value method, retention, and target
>
> **Deprecation cycle**:
>
> - 1.5.30: introduce a warning on repeatable container annotation declarations violating JLS requirements (error in the progressive mode)

- 1.6.0: raise this warning to an error, `-XXLanguage:-RepeatableAnnotationContainerConstraints` can be used to temporarily disable the error reporting

Prohibit declaring a nested class named Container in a repeatable annotation class

Issue: KT-47971

Component: Kotlin/JVM

Incompatible change type: source

Short summary: Kotlin 1.6 will check that a repeatable annotation declared in Kotlin doesn't have a nested class with the predefined name `Container`

Deprecation cycle:

- 1.5.30: introduce a warning on nested classes with the name `Container` in a Kotlin-repeatable annotation class (error in the progressive mode)
- 1.6.0: raise this warning to an error, `-XXLanguage:-RepeatableAnnotationContainerConstraints` can be used to temporarily disable the error reporting

Prohibit @JvmField on a property in the primary constructor that overrides an interface property

Issue: KT-32753

Component: Kotlin/JVM

Incompatible change type: source

Short summary: Kotlin 1.6 will outlaw annotating a property declared in the primary constructor that overrides an interface property with the `@JvmField` annotation

Deprecation cycle:

- 1.5.20: introduce a warning on the `@JvmField` annotation on such properties in the primary constructor
- 1.6.0: raise this warning to an error, `-XXLanguage:-ProhibitJvmFieldOnOverrideFromInterfaceInPrimaryConstructor` can be used to temporarily disable the error reporting

Deprecate the enable and the compatibility modes of the compiler option -Xjvm-default

> **Issue**: KT-46329
>
> **Component**: Kotlin/JVM
>
> **Incompatible change type**: source
>
> **Short summary**: Kotlin 1.6.20 will warn about the usage of `enable` and `compatibility` modes of the `-Xjvm-default` compiler option
>
> **Deprecation cycle**:
>
> - 1.6.20: introduce a warning on the `enable` and `compatibility` modes of the `-Xjvm-default` compiler option
> - >= 1.8.0: raise this warning to an error

Prohibit super calls from public-abi inline functions

> **Issue**: KT-45379
>
> **Component**: Core language
>
> **Incompatible change type**: source
>
> **Short summary**: Kotlin 1.6 will outlaw calling functions with a `super` qualifier from public or protected inline functions and properties
>
> **Deprecation cycle**:
>
> - 1.5.0: introduce a warning on super calls from public or protected inline functions or property accessors
> - 1.6.0: raise this warning to an error, `-XXLanguage:-ProhibitSuperCallsFromPublicInline` can be used to temporarily disable the error reporting

Prohibit protected constructor calls from public inline functions

> **Issue**: KT-48860
>
> **Component**: Core language
>
> **Incompatible change type**: source
>
> **Short summary**: Kotlin 1.6 will outlaw calling protected constructors from public or protected inline functions and properties

> **Deprecation cycle**:
>
> - 1.4.30: introduce a warning on protected constructor calls from public or protected inline functions or property accessors
> - 1.6.0: raise this warning to an error, `-XXLanguage:-ProhibitProtectedConstructorCallFromPublicInline` can be used to temporarily disable the error reporting

Prohibit exposing private nested types from private-in-file types

> **Issue**: KT-20094
>
> **Component**: Core language
>
> **Incompatible change type**: source
>
> **Short summary**: Kotlin 1.6 will outlaw exposing private nested types and inner classes from private-in-file types
>
> **Deprecation cycle**:
>
> - 1.5.0: introduce a warning on private types exposed from private-in-file types
> - 1.6.0: raise this warning to an error, `-XXLanguage:-PrivateInFileEffectiveVisibility` can be used to temporarily disable the error reporting

Annotation target is not analyzed in several cases for annotations on a type

> **Issue**: KT-28449
>
> **Component**: Core language
>
> **Incompatible change type**: source
>
> **Short summary**: Kotlin 1.6 will no longer allow annotations on types that should not be applicable to types
>
> **Deprecation cycle**:
>
> - 1.5.20: introduce an error in the progressive mode
> - 1.6.0: introduce an error, `-XXLanguage:-ProperCheckAnnotationsTargetInTypeUsePositions` can be used to temporarily disable the error reporting

Prohibit calls to functions named suspend with a trailing lambda

> **Issue**: KT-22562
>
> **Component**: Core language
>
> **Incompatible change type**: source
>
> **Short summary**: Kotlin 1.6 will no longer allow calling functions named `suspend` that have the single argument of a functional type passed as a trailing lambda
>
> **Deprecation cycle**:
>
> - 1.3.0: introduce a warning on such function calls
> - 1.6.0: raise this warning to an error
> - >= 1.7.0: introduce changes to the language grammar, so that `suspend` before `{` is parsed as a keyword

标准库

Remove brittle contains optimization in minus/removeAll/retainAll

> **Issue**: KT-45438
>
> **Component**: kotlin-stdlib
>
> **Incompatible change type**: behavioral
>
> **Short summary**: Kotlin 1.6 will no longer perform conversion to set for the argument of functions and operators that remove several elements from collection/iterable/array/sequence.
>
> **Deprecation cycle**:
>
> - < 1.6: the old behavior: the argument is converted to set in some cases
> - 1.6.0: if the function argument is a collection, it's no longer converted to `Set`. If it's not a collection, it can be converted to `List` instead.
>
>   The old behavior can be temporarily turned back on JVM by setting the system property `kotlin.collections.convert_arg_to_set_in_removeAll=true`
> - >= 1.7: the system property above will no longer have an effect

Change value generation algorithm in Random.nextLong

> **Issue**: KT-47304
>
> **Component**: kotlin-stdlib
>
> **Incompatible change type**: behavioral
>
> **Short summary**: Kotlin 1.6 changes the value generation algorithm in the `Random.nextLong` function to avoid producing values out of the specified range.
>
> **Deprecation cycle**:
>
> - 1.6.0: the behavior is fixed immediately

Gradually change the return type of collection min and max functions to non-nullable

> **Issue**: KT-38854
>
> **Component**: kotlin-stdlib
>
> **Incompatible change type**: source
>
> **Short summary**: the return type of collection `min` and `max` functions will be changed to non-nullable in Kotlin 1.7
>
> **Deprecation cycle**:
>
> - 1.4.0: introduce `...OrNull` functions as synonyms and deprecate the affected API (see details in the issue)
> - 1.5.0: raise the deprecation level of the affected API to an error
> - 1.6.0: hide the deprecated functions from the public API
> - >= 1.7: reintroduce the affected API but with non-nullable return type

Deprecate floating-point array functions: contains, indexOf, lastIndexOf

> **Issue**: KT-28753
>
> **Component**: kotlin-stdlib
>
> **Incompatible change type**: source
>
> **Short summary**: Kotlin deprecates floating-point array functions `contains`, `indexOf`, `lastIndexOf` that compare values using the IEEE-754 order instead of the total order
>
> **Deprecation cycle**:

- 1.4.0: deprecate the affected functions with a warning
- 1.6.0: raise the deprecation level to an error
- >= 1.7: hide the deprecated functions from the public API

Migrate declarations from kotlin.dom and kotlin.browser packages to kotlinx.*

Issue: KT-39330

Component: kotlin-stdlib (JS)

Incompatible change type: source

Short summary: declarations from the `kotlin.dom` and `kotlin.browser` packages are moved to the corresponding `kotlinx.*` packages to prepare for extracting them from stdlib

Deprecation cycle:

- 1.4.0: introduce the replacement API in `kotlinx.dom` and `kotlinx.browser` packages
- 1.4.0: deprecate the API in `kotlin.dom` and `kotlin.browser` packages and propose the new API above as a replacement
- 1.6.0: raise the deprecation level to an error
- >= 1.7: remove the deprecated functions from stdlib
- >= 1.7: move the API in kotlinx.* packages to a separate library

Make Regex.replace function not inline in Kotlin/JS

Issue: KT-27738

Component: kotlin-stdlib (JS)

Incompatible change type: source

Short summary: the `Regex.replace` function with the functional `transform` parameter will no longer be inline in Kotlin/JS

Deprecation cycle:

- 1.6.0: remove the `inline` modifier from the affected function

Different behavior of the Regex.replace function in JVM and JS when replacement string contains group reference

Issue: KT-28378

Component: kotlin-stdlib (JS)

> **Incompatible change type**: behavioral
>
> **Short summary**: the function `Regex.replace` in Kotlin/JS with the replacement pattern string will follow the same syntax of that pattern as in Kotlin/JVM
>
> **Deprecation cycle**:
>
> - 1.6.0: change the replacement pattern handling in `Regex.replace` of the Kotlin/JS stdlib

Use the Unicode case folding in JS Regex

> **Issue**: KT-45928
>
> **Component**: kotlin-stdlib (JS)
>
> **Incompatible change type**: behavioral
>
> **Short summary**: the `Regex` class in Kotlin/JS will use `unicode` flag when calling the underlying JS Regular expressions engine to search and compare characters according to the Unicode rules. This brings certain version requirements of the JS environment and causes more strict validation of unnecessary escaping in the regex pattern string.
>
> **Deprecation cycle**:
>
> - 1.5.0: enable the Unicode case folding in most functions of the JS `Regex` class
> - 1.6.0: enable the Unicode case folding in the `Regex.replaceFirst` function

Deprecate some JS-only API

> **Issue**: KT-48587
>
> **Component**: kotlin-stdlib (JS)
>
> **Incompatible change type**: source
>
> **Short summary**: a number of JS-only functions in stdlib are deprecated for removal. They include: `String.concat(String)`, `String.match(regex: String)`, `String.matches(regex: String)`, and the `sort` functions on arrays taking a comparison function, for example, `Array<out T>.sort(comparison: (a: T, b: T) -> Int)`
>
> **Deprecation cycle**:
>
> - 1.6.0: deprecate the affected functions with a warning
> - 1.7.0: raise the deprecation level to an error
> - 1.8.0: remove the deprecated functions from the public API

Hide implementation- and interop-specific functions from the public API of classes in Kotlin/JS

> **Issue**: KT-48587
>
> **Component**: kotlin-stdlib (JS)
>
> **Incompatible change type**: source, binary
>
> **Short summary**: the functions `HashMap.createEntrySet` and `AbstactMutableCollection.toJSON` change their visibility to internal
>
> **Deprecation cycle**:
>
> - 1.6.0: make the functions internal, thus removing them from the public API

Tools

Deprecate KotlinGradleSubplugin class

> **Issue**: KT-48830
>
> **Component**: Gradle
>
> **Incompatible change type**: source
>
> **Short summary**: the class `KotlinGradleSubplugin` will be deprecated in favor of `KotlinCompilerPluginSupportPlugin`
>
> **Deprecation cycle**:
>
> - 1.6.0: raise the deprecation level to an error
> - >= 1.7.0: remove the deprecated class

Remove kotlin.useFallbackCompilerSearch build option

> **Issue**: KT-46719
>
> **Component**: Gradle
>
> **Incompatible change type**: source
>
> **Short summary**: remove the deprecated 'kotlin.useFallbackCompilerSearch' build option
>
> **Deprecation cycle**:
>
> - 1.5.0: raise the deprecation level to a warning

> - 1.6.0: remove the deprecated option

Remove several compiler options

> **Issue**: KT-48847
>
> **Component**: Gradle
>
> **Incompatible change type**: source
>
> **Short summary**: remove the deprecated `noReflect` and `includeRuntime` compiler options
>
> **Deprecation cycle**:
>
> - 1.5.0: raise the deprecation level to an error
> - 1.6.0: remove the deprecated options

Deprecate useIR compiler option

> **Issue**: KT-48847
>
> **Component**: Gradle
>
> **Incompatible change type**: source
>
> **Short summary**: hide the deprecated `useIR` compiler option
>
> **Deprecation cycle**:
>
> - 1.5.0: raise the deprecation level to a warning
> - 1.6.0: hide the option
> - >= 1.7.0: remove the deprecated option

Deprecate kapt.use.worker.api Gradle property

> **Issue**: KT-48826
>
> **Component**: Gradle
>
> **Incompatible change type**: source
>
> **Short summary**: deprecate the `kapt.use.worker.api` property that allowed to run kapt via Gradle Workers API (default: true)
>
> **Deprecation cycle**:
>
> - 1.6.20: raise the deprecation level to a warning
> - >= 1.8.0: remove this property

Remove kotlin.parallel.tasks.in.project Gradle property

> **Issue**: KT-46406
>
> **Component**: Gradle
>
> **Incompatible change type**: source
>
> **Short summary**: remove the `kotlin.parallel.tasks.in.project` property
>
> **Deprecation cycle**:
>
> - 1.5.20: raise the deprecation level to a warning
> - 1.6.20: remove this property

Deprecate kotlin.experimental.coroutines Gradle DSL option and kotlin.coroutines Gradle property

> **Issue**: KT-50369
>
> **Component**: Gradle
>
> **Incompatible change type**: source
>
> **Short summary**: deprecate the `kotlin.experimental.coroutines` Gradle DSL option and the `kotlin.coroutines` property
>
> **Deprecation cycle**:
>
> - 1.6.20: raise the deprecation level to a warning
> - >= 1.7.0: remove the DSL option and the property

Kotlin 1.5 的兼容性指南

Keeping the Language Modern and *Comfortable Updates* are among the fundamental principles in Kotlin Language Design. The former says that constructs which obstruct language evolution should be removed, and the latter says that this removal should be well-communicated beforehand to make code migration as smooth as possible.

While most of the language changes were already announced through other channels, like update changelogs or compiler warnings, this document summarizes them all, providing a complete reference for migration from Kotlin 1.4 to Kotlin 1.5.

Basic terms

In this document we introduce several kinds of compatibility:

- *source*: source-incompatible change stops code that used to compile fine (without errors or warnings) from compiling anymore
- *binary*: two binary artifacts are said to be binary-compatible if interchanging them doesn't lead to loading or linkage errors
- *behavioral*: a change is said to be behavioral-incompatible if the same program demonstrates different behavior before and after applying the change

Remember that those definitions are given only for pure Kotlin. Compatibility of Kotlin code from the other languages perspective (for example, from Java) is out of the scope of this document.

Language and stdlib

Forbid spread operator in signature-polymorphic calls

> **Issue**: KT-35226
>
> **Component**: Core language
>
> **Incompatible change type**: source
>
> **Short summary**: Kotlin 1.5 will outlaw the use of spread operator (*) on signature-polymorphic calls
>
> **Deprecation cycle**:
>
> - < 1.5: introduce warning for the problematic operator at call-site

> - >= 1.5: raise this warning to an error, `-XXLanguage:-ProhibitSpreadOnSignaturePolymorphicCall` can be used to temporarily revert to pre-1.5 behavior

Forbid non-abstract classes containing abstract members invisible from that classes (internal/package-private)

> **Issue**: KT-27825
>
> **Component**: Core language
>
> **Incompatible change type**: source
>
> **Short summary**: Kotlin 1.5 will outlaw non-abstract classes containing abstract members invisible from that classes (internal/package-private)
>
> **Deprecation cycle**:
>
> - < 1.5: introduce warning for the problematic classes
> - >= 1.5: raise this warning to an error, `-XXLanguage:-ProhibitInvisibleAbstractMethodsInSuperclasses` can be used to temporarily revert to pre-1.5 behavior

Forbid using array based on non-reified type parameters as reified type arguments on JVM

> **Issue**: KT-31227
>
> **Component**: Core language
>
> **Incompatible change type**: source
>
> **Short summary**: Kotlin 1.5 will outlaw using array based on non-reified type parameters as reified type arguments on JVM
>
> **Deprecation cycle**:
>
> - < 1.5: introduce warning for the problematic calls
> - >= 1.5: raise this warning to an error, `-XXLanguage:-ProhibitNonReifiedArraysAsReifiedTypeArguments` can be used to temporarily revert to pre-1.5 behavior

Forbid secondary enum class constructors which do not delegate to the primary constructor

> **Issue**: KT-35870
>
> **Component**: Core language
>
> **Incompatible change type**: source
>
> **Short summary**: Kotlin 1.5 will outlaw secondary enum class constructors which do not delegate to the primary constructor
>
> **Deprecation cycle**:
>
> - < 1.5: introduce warning for the problematic constructors
> - >= 1.5: raise this warning to an error, `-XXLanguage:-RequiredPrimaryConstructorDelegationCallInEnums` can be used to temporarily revert to pre-1.5 behavior

Forbid exposing anonymous types from private inline functions

> **Issue**: KT-33917
>
> **Component**: Core language
>
> **Incompatible change type**: source
>
> **Short summary**: Kotlin 1.5 will outlaw exposing anonymous types from private inline functions
>
> **Deprecation cycle**:
>
> - < 1.5: introduce warning for the problematic constructors
> - >= 1.5: raise this warning to an error, `-XXLanguage:-ApproximateAnonymousReturnTypesInPrivateInlineFunctions` can be used to temporarily revert to pre-1.5 behavior

Forbid passing non-spread arrays after arguments with SAM-conversion

> **Issue**: KT-35224
>
> **Component**: Core language
>
> **Incompatible change type**: source
>
> **Short summary**: Kotlin 1.5 will outlaw passing non-spread arrays after arguments with SAM-conversion
>
> **Deprecation cycle**:

> - 1.3.70: introduce warning for the problematic calls
> - >= 1.5: raise this warning to an error, `-XXLanguage:-ProhibitVarargAsArrayAfterSamArgument` can be used to temporarily revert to pre-1.5 behavior

Support special semantics for underscore-named catch block parameters

> **Issue**: KT-31567
>
> **Component**: Core language
>
> **Incompatible change type**: source
>
> **Short summary**: Kotlin 1.5 will outlaw references to the underscore symbol (`_`) that is used to omit parameter name of an exception in the catch block
>
> **Deprecation cycle**:
>
> - 1.4.20: introduce warning for the problematic references
> - >= 1.5: raise this warning to an error, `-XXLanguage:-ForbidReferencingToUnderscoreNamedParameterOfCatchBlock` can be used to temporarily revert to pre-1.5 behavior

Change implementation strategy of SAM conversion from anonymous class-based to invokedynamic

> **Issue**: KT-44912
>
> **Component**: Kotlin/JVM
>
> **Incompatible change type**: behavioral
>
> **Short summary**: Since Kotlin 1.5, implementation strategy of SAM (single abstract method) conversion will be changed from generating an anonymous class to using the `invokedynamic` JVM instruction
>
> **Deprecation cycle**:
>
> - 1.5: change implementation strategy of SAM conversion, `-Xsam-conversions=class` can be used to revert implementation scheme to the one that used before

Performance issues with the JVM IR-based backend

> **Issue**: KT-48233

> **Component**: Kotlin/JVM
>
> **Incompatible change type**: behavioral
>
> **Short summary**: Kotlin 1.5 uses the IR-based backend for the Kotlin/JVM compiler by default. The old backend is still used by default for earlier language versions.
>
> You might encounter some performance degradation issues using the new compiler in Kotlin 1.5. We are working on fixing such cases.
>
> **Deprecation cycle**:
>
> - < 1.5: by default, the old JVM backend is used
> - >= 1.5: by default, the IR-based backend is used. If you need to use the old backend in Kotlin 1.5, add the following lines to the project's configuration file to temporarily revert to pre-1.5 behavior:
>
> In Gradle:

>

```
tasks.withType<org.jetbrains.kotlin.gradle.dsl.KotlinJvmCompile> {
  kotlinOptions.useOldBackend = true
}
```

```
tasks.withType(org.jetbrains.kotlin.gradle.dsl.KotlinJvmCompile) {
  kotlinOptions.useOldBackend = true
}
```

> In Maven:

```
<configuration>
    <args>
        <arg>-Xuse-old-backend</arg>
    </args>
</configuration>
```

> Support for this flag will be removed in one of the future releases.

New field sorting in the JVM IR-based backend

> **Issue**: KT-46378

> **Component**: Kotlin/JVM
>
> **Incompatible change type**: behavioral
>
> **Short summary**: Since version 1.5, Kotlin uses the IR-based backend that sorts JVM bytecode differently: it generates fields declared in the constructor before fields declared in the body, while it's vice versa for the old backend. The new sorting may change the behavior of programs that use serialization frameworks that depend on the field order, such as Java serialization.
>
> **Deprecation cycle**:
>
> - < 1.5: by default, the old JVM backend is used. It has fields declared in the body before fields declared in the constructor.
> - >= 1.5: by default, the new IR-based backend is used. Fields declared in the constructor are generated before fields declared in the body. As a workaround, you can temporarily switch to the old backend in Kotlin 1.5. To do that, add the following lines to the project's configuration file:
>
> In Gradle:

>

```
tasks.withType<org.jetbrains.kotlin.gradle.dsl.KotlinJvmCompile> {
  kotlinOptions.useOldBackend = true
}
```

```
tasks.withType(org.jetbrains.kotlin.gradle.dsl.KotlinJvmCompile) {
  kotlinOptions.useOldBackend = true
}
```

> In Maven:

```
<configuration>
    <args>
        <arg>-Xuse-old-backend</arg>
    </args>
</configuration>
```

> Support for this flag will be removed in one of the future releases.

Generate nullability assertion for delegated properties with a generic call in the delegate expression

> **Issue**: KT-44304
>
> **Component**: Kotlin/JVM
>
> **Incompatible change type**: behavioral
>
> **Short summary**: Since Kotlin 1.5, the Kotlin compiler will emit nullability assertions for delegated properties with a generic call in the delegate expression
>
> **Deprecation cycle**:
>
> - 1.5: emit nullability assertion for delegated properties (see details in the issue), `-Xuse-old-backend` or `-language-version 1.4` can be used to temporarily revert to pre-1.5 behavior

Turn warnings into errors for calls with type parameters annotated by @OnlyInputTypes

> **Issue**: KT-45861
>
> **Component**: Core language
>
> **Incompatible change type**: source
>
> **Short summary**: Kotlin 1.5 will outlaw calls like `contains`, `indexOf`, and `assertEquals` with senseless arguments to improve type safety
>
> **Deprecation cycle**:
>
> - 1.4.0: introduce warning for the problematic constructors
> - >= 1.5: raise this warning to an error, `-XXLanguage:-StrictOnlyInputTypesChecks` can be used to temporarily revert to pre-1.5 behavior

Use the correct order of arguments execution in calls with named vararg

> **Issue**: KT-17691
>
> **Component**: Kotlin/JVM
>
> **Incompatible change type**: behavioral
>
> **Short summary**: Kotlin 1.5 will change the order of arguments execution in calls with named vararg
>
> **Deprecation cycle**:
>
> - < 1.5: introduce warning for the problematic constructors

> - >= 1.5: raise this warning to an error, `-XXLanguage:-UseCorrectExecutionOrderForVarargArguments` can be used to temporarily revert to pre-1.5 behavior

Use default value of the parameter in operator functional calls

> **Issue**: KT-42064
>
> **Component**: Kotlin/JVM
>
> **Incompatible change type**: behavioral
>
> **Short summary**: Kotlin 1.5 will use default value of the parameter in operator calls
>
> **Deprecation cycle**:
>
> - < 1.5: old behavior (see details in the issue)
> - >= 1.5: behavior changed, `-XXLanguage:-JvmIrEnabledByDefault` can be used to temporarily revert to pre-1.5 behavior

Produce empty reversed progressions in for loops if regular progression is also empty

> **Issue**: KT-42533
>
> **Component**: Kotlin/JVM
>
> **Incompatible change type**: behavioral
>
> **Short summary**: Kotlin 1.5 will produce empty reversed progressions in for loops if regular progression is also empty
>
> **Deprecation cycle**:
>
> - < 1.5: old behavior (see details in the issue)
> - >= 1.5: behavior changed, `-XXLanguage:-JvmIrEnabledByDefault` can be used to temporarily revert to pre-1.5 behavior

Straighten Char-to-code and Char-to-digit conversions out

> **Issue**: KT-23451
>
> **Component**: kotlin-stdlib
>
> **Incompatible change type**: source
>
> **Short summary**: Since Kotlin 1.5, conversions of Char to number types will be deprecated

> **Deprecation cycle**:
>
> - 1.5: deprecate `Char.toInt()/toShort()/toLong()/toByte()/toDouble()/toFloat()` and the reverse functions like `Long.toChar()`, and propose replacement

Inconsistent case-insensitive comparison of characters in kotlin.text functions

> **Issue**: KT-45496
>
> **Component**: kotlin-stdlib
>
> **Incompatible change type**: behavioral
>
> **Short summary**: Since Kotlin 1.5, `Char.equals` will be improved in case-insensitive case by first comparing whether the uppercase variants of characters are equal, then whether the lowercase variants of those uppercase variants (as opposed to the characters themselves) are equal
>
> **Deprecation cycle**:
>
> - < 1.5: old behavior (see details in the issue)
> - 1.5: change behavior for `Char.equals` function

Remove default locale-sensitive case conversion API

> **Issue**: KT-43023
>
> **Component**: kotlin-stdlib
>
> **Incompatible change type**: source
>
> **Short summary**: Since Kotlin 1.5, default locale-sensitive case conversion functions like `String.toUpperCase()` will be deprecated
>
> **Deprecation cycle**:
>
> - 1.5: deprecate case conversions functions with the default locale (see details in the issue), and propose replacement

Gradually change the return type of collection min and max functions to non-nullable

> **Issue**: KT-38854
>
> **Component**: kotlin-stdlib (JVM)

> **Incompatible change type**: source
>
> **Short summary**: return type of collection `min` and `max` functions will be changed to non-nullable in 1.6
>
> **Deprecation cycle**:
>
> - 1.4: introduce `...OrNull` functions as synonyms and deprecate the affected API (see details in the issue)
> - 1.5.0: raise the deprecation level of the affected API to error
> - >=1.6: reintroduce the affected API but with non-nullable return type

Raise the deprecation level of conversions of floating-point types to Short and Byte

> **Issue**: KT-30360
>
> **Component**: kotlin-stdlib (JVM)
>
> **Incompatible change type**: source
>
> **Short summary**: conversions of floating-point types to `Short` and `Byte` deprecated in Kotlin 1.4 with `WARNING` level will cause errors since Kotlin 1.5.0.
>
> **Deprecation cycle**:
>
> - 1.4: deprecate `Double.toShort()/toByte()` and `Float.toShort()/toByte()` and propose replacement
> - 1.5.0: raise the deprecation level to error

Tools

Do not mix several JVM variants of kotlin-test in a single project

> **Issue**: KT-40225
>
> **Component**: Gradle
>
> **Incompatible change type**: behavioral
>
> **Short summary**: several mutually exclusive `kotlin-test` variants for different testing frameworks could have been in a project if one of them is brought by a transitive dependency. From 1.5.0, Gradle won't allow having mutually exclusive `kotlin-test` variants for different testing frameworks.

> **Deprecation cycle**:
>
> - < 1.5: having several mutually exclusive `kotlin-test` variants for different testing frameworks is allowed
> - >= 1.5: behavior changed,
>
>
>
> Gradle throws an exception like "Cannot select module with conflict on capability...". Possible solutions:
>
> - use the same `kotlin-test` variant and the corresponding testing framework as the transitive dependency brings.
> - find another variant of the dependency that doesn't bring the `kotlin-test` variant transitively, so you can use the testing framework you would like to use.
> - find another variant of the dependency that brings another `kotlin-test` variant transitively, which uses the same testing framework you would like to use.
> - exclude the testing framework that is brought transitively. The following example is for excluding JUnit 4:
>
>
> ```
> configurations {
>     testImplementation.get().exclude("org.jetbrains.kotlin", "kotlin-test
> -junit")
> }
> ```
>
> After excluding the testing framework, test your application. If it stopped working, rollback excluding changes, use the same testing framework as the library does, and exclude your testing framework.

Kotlin 1.4 的兼容性指南

Keeping the Language Modern and *Comfortable Updates* are among the fundamental principles in Kotlin Language Design. The former says that constructs which obstruct language evolution should be removed, and the latter says that this removal should be well-communicated beforehand to make code migration as smooth as possible.

While most of the language changes were already announced through other channels, like update changelogs or compiler warnings, this document summarizes them all, providing a complete reference for migration from Kotlin 1.3 to Kotlin 1.4.

Basic terms

In this document we introduce several kinds of compatibility:

- *source*: source-incompatible change stops code that used to compile fine (without errors or warnings) from compiling anymore
- *binary*: two binary artifacts are said to be binary-compatible if interchanging them doesn't lead to loading or linkage errors
- *behavioral*: a change is said to be behavioral-incompatible if the same program demonstrates different behavior before and after applying the change

Remember that those definitions are given only for pure Kotlin. Compatibility of Kotlin code from the other languages perspective (for example, from Java) is out of the scope of this document.

Language and stdlib

Unexpected behavior with in infix operator and ConcurrentHashMap

> **Issue**: KT-18053
>
> **Component**: Core language
>
> **Incompatible change type**: source
>
> **Short summary**: Kotlin 1.4 will outlaw auto operator `contains` coming from the implementors of `java.util.Map` written in Java
>
> **Deprecation cycle**:

- < 1.4: introduce warning for problematic operators at call-site
- >= 1.4: raise this warning to an error, `-XXLanguage:-ProhibitConcurrentHashMapContains` can be used to temporarily revert to pre-1.4 behavior

Prohibit access to protected members inside public inline members

Issue: KT-21178

Component: Core language

Incompatible change type: source

Short summary: Kotlin 1.4 will prohibit access to protected members from public inline members.

Deprecation cycle:

- < 1.4: introduce warning at call-site for problematic cases
- 1.4: raise this warning to an error, `-XXLanguage:-ProhibitProtectedCallFromInline` can be used to temporarily revert to pre-1.4 behavior

Contracts on calls with implicit receivers

Issue: KT-28672

Component: Core Language

Incompatible change type: behavioral

Short summary: smart casts from contracts will be available on calls with implicit receivers in 1.4

Deprecation cycle:

- < 1.4: old behavior (see details in the issue)
- >= 1.4: behavior changed, `-XXLanguage:-ContractsOnCallsWithImplicitReceiver` can be used to temporarily revert to pre-1.4 behavior

Inconsistent behavior of floating-point number comparisons

Issues: KT-22723

Component: Core language

Incompatible change type: behavioral

> **Short summary**: since Kotlin 1.4, Kotlin compiler will use IEEE 754 standard to compare floating-point numbers
>
> **Deprecation cycle**:
>
> - < 1.4: old behavior (see details in the issue)
> - >= 1.4: behavior changed, `-XXLanguage:-ProperIeee754Comparisons` can be used to temporarily revert to pre-1.4 behavior

No smart cast on the last expression in a generic lambda

> **Issue**: KT-15020
>
> **Component**: Core Language
>
> **Incompatible change type**: behavioral
>
> **Short summary**: smart casts for last expressions in lambdas will be correctly applied since 1.4
>
> **Deprecation cycle**:
>
> - < 1.4: old behavior (see details in the issue)
> - >= 1.4: behavior changed, `-XXLanguage:-NewInference` can be used to temporarily revert to pre-1.4 behavior. Note that this flag will also disable several new language features.

Do not depend on the order of lambda arguments to coerce result to Unit

> **Issue**: KT-36045
>
> **Component**: Core language
>
> **Incompatible change type**: source
>
> **Short summary**: since Kotlin 1.4, lambda arguments will be resolved independently without implicit coercion to `Unit`
>
> **Deprecation cycle**:
>
> - < 1.4: old behavior (see details in the issue)
> - >= 1.4: behavior changed, `-XXLanguage:-NewInference` can be used to temporarily revert to pre-1.4 behavior. Note that this flag will also disable several new language features.

Wrong common supertype between raw and integer literal type leads to unsound code

> **Issue**: KT-35681
>
> **Components**: Core language
>
> **Incompatible change type**: source
>
> **Short summary**: since Kotlin 1.4, common supertype between raw `Comparable` type and integer literal type will be more specific
>
> **Deprecation cycle**:
>
> - < 1.4: old behavior (see details in the issue)
> - >= 1.4: behavior changed, `-XXLanguage:-NewInference` can be used to temporarily revert to pre-1.4 behavior. Note that this flag will also disable several new language features.

Type safety problem because several equal type variables are instantiated with a different types

> **Issue**: KT-35679
>
> **Component**: Core language
>
> **Incompatible change type**: source
>
> **Short summary**: since Kotlin 1.4, Kotlin compiler will prohibit instantiating equal type variables with different types
>
> **Deprecation cycle**:
>
> - < 1.4: old behavior (see details in the issue)
> - >= 1.4: behavior changed, `-XXLanguage:-NewInference` can be used to temporarily revert to pre-1.4 behavior. Note that this flag will also disable several new language features.

Type safety problem because of incorrect subtyping for intersection types

> **Issues**: KT-22474
>
> **Component**: Core language
>
> **Incompatible change type**: source
>
> **Short summary**: in Kotlin 1.4, subtyping for intersection types will be refined to work more correctly
>
> **Deprecation cycle**:

- < 1.4: old behavior (see details in the issue)
- >= 1.4: behavior changed, `-XXLanguage:-NewInference` can be used to temporarily revert to pre-1.4 behavior. Note that this flag will also disable several new language features.

No type mismatch with an empty when expression inside lambda

Issue: KT-17995

Component: Core language

Incompatible change type: source

Short summary: since Kotlin 1.4, there will be a type mismatch for empty `when` expression if it's used as the last expression in a lambda

Deprecation cycle:

- < 1.4: old behavior (see details in the issue)
- >= 1.4: behavior changed, `-XXLanguage:-NewInference` can be used to temporarily revert to pre-1.4 behavior. Note that this flag will also disable several new language features.

Return type Any inferred for lambda with early return with integer literal in one of possible return values

Issue: KT-20226

Component: Core language

Incompatible change type: source

Short summary: since Kotlin 1.4, integer type returning from a lambda will be more specific for cases when there is early return

Deprecation cycle:

- < 1.4: old behavior (see details in the issue)
- >= 1.4: behavior changed, `-XXLanguage:-NewInference` can be used to temporarily revert to pre-1.4 behavior. Note that this flag will also disable several new language features.

Proper capturing of star projections with recursive types

Issue: KT-33012

Component: Core language

Incompatible change type: source

> **Short summary**: since Kotlin 1.4, more candidates will become applicable because capturing for recursive types will work more correctly
>
> **Deprecation cycle**:
>
> - < 1.4: old behavior (see details in the issue)
> - >= 1.4: behavior changed, `-XXLanguage:-NewInference` can be used to temporarily revert to pre-1.4 behavior. Note that this flag will also disable several new language features.

Common supertype calculation with non-proper type and flexible one leads to incorrect results

> **Issue**: KT-37054
>
> **Component**: Core language
>
> **Incompatible change type**: behavioral
>
> **Short summary**: since Kotlin 1.4, common supertype between flexible types will be more specific protecting from runtime errors
>
> **Deprecation cycle**:
>
> - < 1.4: old behavior (see details in the issue)
> - >= 1.4: behavior changed, `-XXLanguage:-NewInference` can be used to temporarily revert to pre-1.4 behavior. Note that this flag will also disable several new language features.

Type safety problem because of lack of captured conversion against nullable type argument

> **Issue**: KT-35487
>
> **Component**: Core language
>
> **Incompatible change type**: source
>
> **Short summary**: since Kotlin 1.4, subtyping between captured and nullable types will be more correct protecting from runtime errors
>
> **Deprecation cycle**:
>
> - < 1.4: old behavior (see details in the issue)
> - >= 1.4: behavior changed, `-XXLanguage:-NewInference` can be used to temporarily revert to pre-1.4 behavior. Note that this flag will also disable several new language features.

Preserve intersection type for covariant types after unchecked cast

> **Issue**: KT-37280
>
> **Component**: Core language
>
> **Incompatible change type**: source
>
> **Short summary**: since Kotlin 1.4, uchecked casts of covariant types produce the intersection type for smart casts, not the type of the unchecked cast.
>
> **Deprecation cycle**:
>
> - < 1.4: old behavior (see details in the issue)
> - >= 1.4: behavior changed, `-XXLanguage:-NewInference` can be used to temporarily revert to pre-1.4 behavior. Note that this flag will also disable several new language features.

Type variable leaks from builder inference because of using this expression

> **Issue**: KT-32126
>
> **Component**: Core language
>
> **Incompatible change type**: source
>
> **Short summary**: since Kotlin 1.4, using `this` inside builder functions like `sequence {}` is prohibited if there are no other proper constraints
>
> **Deprecation cycle**:
>
> - < 1.4: old behavior (see details in the issue)
> - >= 1.4: behavior changed, `-XXLanguage:-NewInference` can be used to temporarily revert to pre-1.4 behavior. Note that this flag will also disable several new language features.

Wrong overload resolution for contravariant types with nullable type arguments

> **Issue**: KT-31670
>
> **Component**: Core language
>
> **Incompatible change type**: source
>
> **Short summary**: since Kotlin 1.4, if two overloads of a function that takes contravariant type arguments differ only by the nullability of the type (such as `In<T>` and `In<T?>`), the nullable type is considered more specific.
>
> **Deprecation cycle**:

- < 1.4: old behavior (see details in the issue)
- >= 1.4: behavior changed, `-XXLanguage:-NewInference` can be used to temporarily revert to pre-1.4 behavior. Note that this flag will also disable several new language features.

Builder inference with non-nested recursive constraints

Issue: KT-34975

Component: Core language

Incompatible change type: source

Short summary: since Kotlin 1.4, builder functions such as `sequence {}` with type that depends on a recursive constraint inside the passed lambda cause a compiler error.

Deprecation cycle:

- < 1.4: old behavior (see details in the issue)
- >= 1.4: behavior changed, `-XXLanguage:-NewInference` can be used to temporarily revert to pre-1.4 behavior. Note that this flag will also disable several new language features.

Eager type variable fixation leads to a contradictory constraint system

Issue: KT-25175

Component: Core language

Incompatible change type: source

Short summary: since Kotlin 1.4, the type inference in certain cases works less eagerly allowing to find the constraint system that is not contradictory.

Deprecation cycle:

- < 1.4: old behavior (see details in the issue)
- >= 1.4: behavior changed, `-XXLanguage:-NewInference` can be used to temporarily revert to pre-1.4 behavior. Note that this flag will also disable several new language features.

Prohibit tailrec modifier on open functions

Issue: KT-18541

Component: Core language

Incompatible change type: source

> **Short summary**: since Kotlin 1.4, functions can't have `open` and `tailrec` modifiers at the same time.
>
> **Deprecation cycle**:
>
> - < 1.4: report a warning on functions that have `open` and `tailrec` modifiers together (error in the progressive mode).
> - >= 1.4: raise this warning to an error.

The INSTANCE field of a companion object more visible than the companion object class itself

> **Issue**: KT-11567
>
> **Component**: Kotlin/JVM
>
> **Incompatible change type**: source
>
> **Short summary**: since Kotlin 1.4, if a companion object is private, then its field `INSTANCE` will be also private
>
> **Deprecation cycle**:
>
> - < 1.4: the compiler generates object `INSTANCE` with a deprecated flag
> - >= 1.4: companion object `INSTANCE` field has proper visibility

Outer finally block inserted before return is not excluded from the catch interval of the inner try block without finally

> **Issue**: KT-31923
>
> **Component**: Kotlin/JVM
>
> **Incompatible change type**: behavioral
>
> **Short summary**: since Kotlin 1.4, the catch interval will be computed properly for nested `try/catch` blocks
>
> **Deprecation cycle**:
>
> - < 1.4: old behavior (see details in the issue)
> - >= 1.4: behavior changed, `-XXLanguage:-ProperFinally` can be used to temporarily revert to pre-1.4 behavior

Use the boxed version of an inline class in return type position for covariant and generic-specialized overrides

> **Issues**: KT-30419
>
> **Component**: Kotlin/JVM
>
> **Incompatible change type**: behavioral
>
> **Short summary**: since Kotlin 1.4, functions using covariant and generic-specialized overrides will return boxed values of inline classes
>
> **Deprecation cycle**:
>
> - < 1.4: old behavior (see details in the issue)
> - >= 1.4: behavior changed

Do not declare checked exceptions in JVM bytecode when using delegation to Kotlin interfaces

> **Issue**: KT-35834
>
> **Component**: Kotlin/JVM
>
> **Incompatible change type**: source
>
> **Short summary**: Kotlin 1.4 will not generate checked exceptions during interface delegation to Kotlin interfaces
>
> **Deprecation cycle**:
>
> - < 1.4: old behavior (see details in the issue)
> - >= 1.4: behavior changed, `-XXLanguage:-DoNotGenerateThrowsForDelegatedKotlinMembers` can be used to temporarily revert to pre-1.4 behavior

Changed behavior of signature-polymorphic calls to methods with a single vararg parameter to avoid wrapping the argument into another array

> **Issue**: KT-35469
>
> **Component**: Kotlin/JVM
>
> **Incompatible change type**: source
>
> **Short summary**: Kotlin 1.4 will not wrap the argument into another array on a signature-polymorphic call
>
> **Deprecation cycle**:

- < 1.4: old behavior (see details in the issue)
- >= 1.4: behavior changed

Incorrect generic signature in annotations when KClass is used as a generic parameter

Issue: KT-35207

Component: Kotlin/JVM

Incompatible change type: source

Short summary: Kotlin 1.4 will fix incorrect type mapping in annotations when KClass is used as a generic parameter

Deprecation cycle:

- < 1.4: old behavior (see details in the issue)
- >= 1.4: behavior changed

Forbid spread operator in signature-polymorphic calls

Issue: KT-35226

Component: Kotlin/JVM

Incompatible change type: source

Short summary: Kotlin 1.4 will prohibit the use of spread operator (*) on signature-polymorphic calls

Deprecation cycle:

- < 1.4: report a warning on the use of a spread operator in signature-polymorphic calls
- >= 1.5: raise this warning to an error, `-XXLanguage:-ProhibitSpreadOnSignaturePolymorphicCall` can be used to temporarily revert to pre-1.4 behavior

Change initialization order of default values for tail-recursive optimized functions

Issue: KT-31540

Component: Kotlin/JVM

Incompatible change type: behavioral

> **Short summary**: Since Kotlin 1.4, the initialization order for tail-recursive functions will be the same as for regular functions
>
> **Deprecation cycle**:
>
> - < 1.4: report a warning at declaration-site for problematic functions
> - >= 1.4: behavior changed, `-XXLanguage:-ProperComputationOrderOfTailrecDefaultParameters` can be used to temporarily revert to pre-1.4 behavior

Do not generate ConstantValue attribute for non-const vals

> **Issue**: KT-16615
>
> **Component**: Kotlin/JVM
>
> **Incompatible change type**: behavioral
>
> **Short summary**: Since Kotlin 1.4, the compiler will not generate the `ConstantValue` attribute for non-`const` `val`s
>
> **Deprecation cycle**:
>
> - < 1.4: report a warning through an IntelliJ IDEA inspection
> - >= 1.4: behavior changed, `-XXLanguage:-NoConstantValueAttributeForNonConstVals` can be used to temporarily revert to pre-1.4 behavior

Generated overloads for @JvmOverloads on open methods should be final

> **Issue**: KT-33240
>
> **Components**: Kotlin/JVM
>
> **Incompatible change type**: source
>
> **Short summary**: overloads for functions with `@JvmOverloads` will be generated as `final`
>
> **Deprecation cycle**:
>
> - < 1.4: old behavior (see details in the issue)
> - >= 1.4: behavior changed, `-XXLanguage:-GenerateJvmOverloadsAsFinal` can be used to temporarily revert to pre-1.4 behavior

Lambdas returning kotlin.Result now return boxed value instead of unboxed

> **Issue**: KT-39198
>
> **Component**: Kotlin/JVM
>
> **Incompatible change type**: behavioral
>
> **Short summary**: since Kotlin 1.4, lambdas returning values of `kotlin.Result` type will return boxed value instead of unboxed
>
> **Deprecation cycle**:
>
> - < 1.4: old behavior (see details in the issue)
> - >= 1.4: behavior changed

Unify exceptions from null checks

> **Issue**: KT-22275
>
> **Component**: Kotlin/JVM
>
> **Incompatible change type**: behavior
>
> **Short summary**: Starting from Kotlin 1.4, all runtime null checks will throw a `java.lang.NullPointerException`
>
> **Deprecation cycle**:
>
> - < 1.4: runtime null checks throw different exceptions, such as `KotlinNullPointerException`, `IllegalStateException`, `IllegalArgumentException`, and `TypeCastException`
> - >= 1.4: all runtime null checks throw a `java.lang.NullPointerException`. `-Xno-unified-null-checks` can be used to temporarily revert to pre-1.4 behavior

Comparing floating-point values in array/list operations contains, indexOf, lastIndexOf: IEEE 754 or total order

> **Issue**: KT-28753
>
> **Component**: kotlin-stdlib (JVM)
>
> **Incompatible change type**: behavioral
>
> **Short summary**: the `List` implementation returned from `Double/FloatArray.asList()` will implement `contains`, `indexOf`, and `lastIndexOf`, so that they use total order equality
>
> **Deprecation cycle**:
>
> - < 1.4: old behavior (see details in the issue)

> - >= 1.4: behavior changed

Gradually change the return type of collection min and max functions to non-nullable

> **Issue**: KT-38854
>
> **Component**: kotlin-stdlib (JVM)
>
> **Incompatible change type**: source
>
> **Short summary**: return type of collection `min` and `max` functions will be changed to non-nullable in 1.6
>
> **Deprecation cycle**:
>
> - 1.4: introduce `...OrNull` functions as synonyms and deprecate the affected API (see details in the issue)
> - 1.5.x: raise the deprecation level of the affected API to error
> - >=1.6: reintroduce the affected API but with non-nullable return type

Deprecate appendln in favor of appendLine

> **Issue**: KT-38754
>
> **Component**: kotlin-stdlib (JVM)
>
> **Incompatible change type**: source
>
> **Short summary**: `StringBuilder.appendln()` will be deprecated in favor of `StringBuilder.appendLine()`
>
> **Deprecation cycle**:
>
> - 1.4: introduce `appendLine` function as a replacement for `appendln` and deprecate `appendln`
> - >=1.5: raise the deprecation level to error

Deprecate conversions of floating-point types to Short and Byte

> **Issue**: KT-30360
>
> **Component**: kotlin-stdlib (JVM)
>
> **Incompatible change type**: source

> **Short summary**: since Kotlin 1.4, conversions of floating-point types to `Short` and `Byte` will be deprecated
>
> **Deprecation cycle**:
>
> - 1.4: deprecate `Double.toShort()/toByte()` and `Float.toShort()/toByte()` and propose replacement
> - >=1.5: raise the deprecation level to error

Fail fast in Regex.findAll on an invalid startIndex

> **Issue**: KT-28356
>
> **Component**: kotlin-stdlib
>
> **Incompatible change type**: behavioral
>
> **Short summary**: since Kotlin 1.4, `findAll` will be improved to check that `startIndex` is in the range of the valid position indices of the input char sequence at the moment of entering `findAll`, and throw `IndexOutOfBoundsException` if it's not
>
> **Deprecation cycle**:
>
> - < 1.4: old behavior (see details in the issue)
> - >= 1.4: behavior changed

Remove deprecated kotlin.coroutines.experimental

> **Issue**: KT-36083
>
> **Component**: kotlin-stdlib
>
> **Incompatible change type**: source
>
> **Short summary**: since Kotlin 1.4, the deprecated `kotlin.coroutines.experimental` API is removed from stdlib
>
> **Deprecation cycle**:
>
> - < 1.4: `kotlin.coroutines.experimental` is deprecated with the `ERROR` level
> - >= 1.4: `kotlin.coroutines.experimental` is removed from stdlib. On the JVM, a separate compatibility artifact is provided (see details in the issue).

Remove deprecated mod operator

> **Issue**: KT-26654

> **Component**: kotlin-stdlib
>
> **Incompatible change type**: source
>
> **Short summary**: since Kotlin 1.4, `mod` operator on numeric types is removed from stdlib
>
> **Deprecation cycle**:
>
> - < 1.4: `mod` is deprecated with the `ERROR` level
> - >= 1.4: `mod` is removed from stdlib

Hide Throwable.addSuppressed member and prefer extension instead

> **Issue**: KT-38777
>
> **Component**: kotlin-stdlib
>
> **Incompatible change type**: behavioral
>
> **Short summary**: `Throwable.addSuppressed()` extension function is now preferred over the `Throwable.addSuppressed()` member function
>
> **Deprecation cycle**:
>
> - < 1.4: old behavior (see details in the issue)
> - >= 1.4: behavior changed

capitalize should convert digraphs to title case

> **Issue**: KT-38817
>
> **Component**: kotlin-stdlib
>
> **Incompatible change type**: behavioral
>
> **Short summary**: `String.capitalize()` function now capitalizes digraphs from the Serbo-Croatian Gaj's Latin alphabet in the title case (`Dž` instead of `DŽ`)
>
> **Deprecation cycle**:
>
> - < 1.4: digraphs are capitalized in the upper case (`DŽ`)
> - >= 1.4: digraphs are capitalized in the title case (`Dž`)

Tools

Compiler arguments with delimiter characters must be passed in double quotes on Windows

> **Issue**: KT-41309
>
> **Component**: CLI
>
> **Incompatible change type**: behavioral
>
> **Short summary**: on Windows, `kotlinc.bat` arguments that contain delimiter characters (whitespace, `=`, `;`, `,`) now require double quotes (`"`)
>
> **Deprecation cycle**:
>
> - < 1.4: all compiler arguments are passed without quotes
> - >= 1.4: compiler arguments that contain delimiter characters (whitespace, `=`, `;`, `,`) require double quotes (`"`)

KAPT: Names of synthetic $annotations() methods for properties have changed

> **Issue**: KT-36926
>
> **Component**: KAPT
>
> **Incompatible change type**: behavioral
>
> **Short summary**: names of synthetic `$annotations()` methods generated by KAPT for properties have changed in 1.4
>
> **Deprecation cycle**:
>
> - < 1.4: names of synthetic `$annotations()` methods for properties follow the template `<propertyName>@annotations()`
> - >= 1.4: names of synthetic `$annotations()` methods for properties include the `get` prefix: `get<PropertyName>@annotations()`

Kotlin 1.3 的兼容性指南

Keeping the Language Modern and *Comfortable Updates* are among the fundamental principles in Kotlin Language Design. The former says that constructs which obstruct language evolution should be removed, and the latter says that this removal should be well-communicated beforehand to make code migration as smooth as possible.

While most of the language changes were already announced through other channels, like update changelogs or compiler warnings, this document summarizes them all, providing a complete reference for migration from Kotlin 1.2 to Kotlin 1.3.

Basic terms

In this document we introduce several kinds of compatibility:

- *Source*: source-incompatible change stops code that used to compile fine (without errors or warnings) from compiling anymore
- *Binary*: two binary artifacts are said to be binary-compatible if interchanging them doesn't lead to loading or linkage errors
- *Behavioral*: a change is said to be behavioral-incompatible if one and the same program demonstrates different behavior before and after applying the change

Remember that those definitions are given only for pure Kotlin. Compatibility of Kotlin code from the other languages perspective (e.g. from Java) is out of the scope of this document.

Incompatible changes

Evaluation order of constructor arguments regarding <clinit> call

> **Issue**: KT-19532
>
> **Component**: Kotlin/JVM
>
> **Incompatible change type**: behavioral
>
> **Short summary**: evaluation order with respect to class initialization is changed in 1.3
>
> **Deprecation cycle**:
>
> - <1.3: old behavior (see details in the Issue)
> - >= 1.3: behavior changed, `-Xnormalize-constructor-calls=disable` can be used to

> temporarily revert to pre-1.3 behavior. Support for this flag is going to be removed in the next major release.

Missing getter-targeted annotations on annotation constructor parameters

> **Issue**: KT-25287
>
> **Component**: Kotlin/JVM
>
> **Incompatible change type**: behavioral
>
> **Short summary**: getter-target annotations on annotations constructor parameters will be properly written to classfiles in 1.3
>
> **Deprecation cycle**:
>
> - <1.3: getter-target annotations on annotation constructor parameters are not applied
> - >=1.3: getter-target annotations on annotation constructor parameters are properly applied and written to the generated code

Missing errors in class constructor's @get: annotations

> **Issue**: KT-19628
>
> **Component**: Core language
>
> **Incompatible change type**: Source
>
> **Short summary**: errors in getter-target annotations will be reported properly in 1.3
>
> **Deprecation cycle**:
>
> - <1.2: compilation errors in getter-target annotations were not reported, causing incorrect code to be compiled fine.
> - 1.2.x: errors reported only by tooling, the compiler still compiles such code without any warnings
> - >=1.3: errors reported by the compiler too, causing erroneous code to be rejected

Nullability assertions on access to Java types annotated with @NotNull

> **Issue**: KT-20830
>
> **Component**: Kotlin/JVM

> **Incompatible change type**: Behavioral
>
> **Short summary**: nullability assertions for Java-types annotated with not-null annotations will be generated more aggressively, causing code which passes `null` here to fail faster.
>
> **Deprecation cycle**:
>
> - <1.3: the compiler could miss such assertions when type inference was involved, allowing potential `null` propagation during compilation against binaries (see Issue for details).
> - >=1.3: the compiler generates missed assertions. This can cause code which was (erroneously) passing `null`s here fail faster.
>
> `-XXLanguage:-StrictJavaNullabilityAssertions` can be used to temporarily return to the pre-1.3 behavior. Support for this flag will be removed in the next major release.

Unsound smartcasts on enum members

> **Issue**: KT-20772
>
> **Component**: Core language
>
> **Incompatible change type**: Source
>
> **Short summary**: a smartcast on a member of one enum entry will be correctly applied to only this enum entry
>
> **Deprecation cycle**:
>
> - <1.3: a smartcast on a member of one enum entry could lead to an unsound smartcast on the same member of other enum entries.
> - >=1.3: smartcast will be properly applied only to the member of one enum entry.
>
> `-XXLanguage:-SoundSmartcastForEnumEntries` will temporarily return old behavior. Support for this flag will be removed in the next major release.

val backing field reassignment in getter

> **Issue**: KT-16681
>
> **Components**: Core language

> **Incompatible change type**: Source
>
> **Short summary**: reassignment of the backing field of `val`-property in its getter is now prohibited
>
> **Deprecation cycle**:
>
> - <1.2: Kotlin compiler allowed to modify backing field of `val` in its getter. Not only it violates Kotlin semantic, but also generates ill-behaved JVM bytecode which reassigns `final` field.
> - 1.2.X: deprecation warning is reported on code which reassigns backing field of `val`
> - >=1.3: deprecation warnings are elevated to errors

Array capturing before the for-loop where it is iterated

> **Issue**: KT-21354
>
> **Component**: Kotlin/JVM
>
> **Incompatible change type**: Source
>
> **Short summary**: if an expression in for-loop range is a local variable updated in a loop body, this change affects loop execution. This is inconsistent with iterating over other containers, such as ranges, character sequences, and collections.
>
> **Deprecation cycle**:
>
> - <1.2: described code patterns are compiled fine, but updates to local variable affect loop execution
> - 1.2.X: deprecation warning reported if a range expression in a for-loop is an array-typed local variable which is assigned in a loop body
> - 1.3: change behavior in such cases to be consistent with other containers

Nested classifiers in enum entries

> **Issue**: KT-16310
>
> **Component**: Core language
>
> **Incompatible change type**: Source
>
> **Short summary**: since Kotlin 1.3, nested classifiers (classes, object, interfaces, annotation classes, enum classes) in enum entries are prohibited
>
> **Deprecation cycle**:
>
> - <1.2: nested classifiers in enum entries are compiled fine, but may fail with exception at

> runtime
> - 1.2.X: deprecation warnings reported on the nested classifiers
> - >=1.3: deprecation warnings elevated to errors

Data class overriding copy

> **Issue**: KT-19618
>
> **Components**: Core language
>
> **Incompatible change type**: Source
>
> **Short summary**: since Kotlin 1.3, data classes are prohibited to override `copy()`
>
> **Deprecation cycle**:
>
> - <1.2: data classes overriding `copy()` are compiled fine but may fail at runtime/expose strange behavior
> - 1.2.X: deprecation warnings reported on data classes overriding `copy()`
> - >=1.3: deprecation warnings elevated to errors

Inner classes inheriting Throwable that capture generic parameters from the outer class

> **Issue**: KT-17981
>
> **Component**: Core language
>
> **Incompatible change type**: Source
>
> **Short summary**: since Kotlin 1.3, inner classes are not allowed to inherit `Throwable`
>
> **Deprecation cycle**:
>
> - <1.2: inner classes inheriting `Throwable` are compiled fine. If such inner classes happen to capture generic parameters, it could lead to strange code patterns which fail at runtime.
> - 1.2.X: deprecation warnings reported on inner classes inheriting `Throwable`
> - >=1.3: deprecation warnings elevated to errors

Visibility rules regarding complex class hierarchies with companion objects

> **Issues**: KT-21515, KT-25333
>
> **Component**: Core language

> **Incompatible change type**: Source
>
> **Short summary**: since Kotlin 1.3, rules of visibility by short names are stricter for complex class hierarchies involving companion objects and nested classifiers.
>
> **Deprecation cycle**:
>
> - <1.2: old visibility rules (see Issue for details)
> - 1.2.X: deprecation warnings reported on short names which are not going to be accessible anymore. Tooling suggests automated migration by adding full name.
> - >=1.3: deprecation warnings elevated to errors. Offending code should add full qualifiers or explicit imports

Non-constant vararg annotation parameters

> **Issue**: KT-23153
>
> **Component**: Core language
>
> **Incompatible change type**: Source
>
> **Short summary**: since Kotlin 1.3, setting non-constant values as vararg annotation parameters is prohibited
>
> **Deprecation cycle**:
>
> - <1.2: the compiler allows to pass non-constant value for vararg annotation parameter, but actually drops that value during bytecode generation, leading to non-obvious behavior
> - 1.2.X: deprecation warnings reported on such code patterns
> - >=1.3: deprecation warnings elevated to errors

Local annotation classes

> **Issue**: KT-23277
>
> **Component**: Core language
>
> **Incompatible change type**: Source
>
> **Short summary**: since Kotlin 1.3 local annotation classes are not supported
>
> **Deprecation cycle**:
>
> - <1.2: the compiler compiled local annotation classes fine
> - 1.2.X: deprecation warnings reported on local annotation classes
> - >=1.3: deprecation warnings elevated to errors

Smartcasts on local delegated properties

> **Issue**: KT-22517
>
> **Component**: Core language
>
> **Incompatible change type**: Source
>
> **Short summary**: since Kotlin 1.3 smartcasts on local delegated properties are not allowed
>
> **Deprecation cycle**:
>
> - <1.2: the compiler allowed to smartcast local delegated property, which could lead to unsound smartcast in case of ill-behaved delegates
> - 1.2.X: smartcasts on local delegated properries are reported as deprecated (the compiler issues warnings)
> - >=1.3: deprecation warnings elevated to errors

mod operator convention

> **Issues**: KT-24197
>
> **Component**: Core language
>
> **Incompatible change type**: Source
>
> **Short summary**: since Kotlin 1.3 declaration of `mod` operator is prohibited, as well as calls which resolve to such declarations
>
> **Deprecation cycle**:
>
> - 1.1.X, 1.2.X: report warnings on declarations of `operator mod`, as well as on calls which resolve to it
> - 1.3.X: elevate warnings to error, but still allow to resolve to `operator mod` declarations
> - 1.4.X: do not resolve calls to `operator mod` anymore

Passing single element to vararg in named form

> **Issues**: KT-20588, KT-20589. See also KT-20171
>
> **Component**: Core language
>
> **Incompatible change type**: Source
>
> **Short summary**: in Kotlin 1.3, assigning single element to vararg is deprecated and should be replaced with consecutive spread and array construction.

> **Deprecation cycle**:
>
> - <1.2: assigning one value element to vararg in named form compiles fine and is treated as assigning *single* element to array, causing non-obvious behavior when assigning array to vararg
> - 1.2.X: deprecation warnings are reported on such assignments, users are suggested to switch to consecutive spread and array construction.
> - 1.3.X: warnings are elevated to errors
> - >= 1.4: change semantic of assigning single element to vararg, making assignment of array equivalent to the assignment of a spread of an array

Retention of annotations with target EXPRESSION

> **Issue**: KT-13762
>
> **Component**: Core language
>
> **Incompatible change type**: Source
>
> **Short summary**: since Kotlin 1.3, only `SOURCE` retention is allowed for annotations with target `EXPRESSION`
>
> **Deprecation cycle**:
>
> - <1.2: annotations with target `EXPRESSION` and retention other than `SOURCE` are allowed, but silently ignored at use-sites
> - 1.2.X: deprecation warnings are reported on declarations of such annotations
> - >=1.3: warnings are elevated to errors

Annotations with target PARAMETER shouldn't be applicable to parameter's type

> **Issue**: KT-9580
>
> **Component**: Core language
>
> **Incompatible change type**: Source
>
> **Short summary**: since Kotlin 1.3, error about wrong annotation target will be properly reported when annotation with target `PARAMETER` is applied to parameter's type
>
> **Deprecation cycle**:
>
> - <1.2: aforementioned code patterns are compiled fine; annotations are silently ignored and not present in the bytecode
> - 1.2.X: deprecation warnings are reported on such usages

> - >=1.3: warnings are elevated to errors

Array.copyOfRange throws an exception when indices are out of bounds instead of enlarging the returned array

> **Issue**: KT-19489
>
> **Component**: kotlin-stdlib (JVM)
>
> **Incompatible change type**: Behavioral
>
> **Short summary**: since Kotlin 1.3, ensure that the `toIndex` argument of `Array.copyOfRange`, which represents the exclusive end of the range being copied, is not greater than the array size and throw `IllegalArgumentException` if it is.
>
> **Deprecation cycle**:
>
> - <1.3: in case `toIndex` in the invocation of `Array.copyOfRange` is greater than the array size, the missing elements in range fill be filled with `null`s, violating soundness of the Kotlin type system.
> - >=1.3: check that `toIndex` is in the array bounds, and throw exception if it isn't

Progressions of ints and longs with a step of Int.MIN_VALUE and Long.MIN_VALUE are outlawed and won't be allowed to be instantiated

> **Issue**: KT-17176
>
> **Component**: kotlin-stdlib (JVM)
>
> **Incompatible change type**: Behavioral
>
> **Short summary**: since Kotlin 1.3, prohibit step value for integer progressions being the minimum negative value of its integer type (`Long` or `Int`), so that calling `IntProgression.fromClosedRange(0, 1, step = Int.MIN_VALUE)` will throw `IllegalArgumentException`
>
> **Deprecation cycle**:
>
> - <1.3: it was possible to create an `IntProgression` with `Int.MIN_VALUE` step, which yields two values `[0, -2147483648]`, which is non-obvious behavior
> - >=1.3: throw `IllegalArgumentException` if the step is the minimum negative value of its integer type

Check for index overflow in operations on very long sequences

> **Issue**: KT-16097
>
> **Component**: kotlin-stdlib (JVM)
>
> **Incompatible change type**: Behavioral
>
> **Short summary**: since Kotlin 1.3, make sure `index`, `count` and similar methods do not overflow for long sequences. See the Issue for the full list of affected methods.
>
> **Deprecation cycle**:
>
> - <1.3: calling such methods on very long sequences could produce negative results due to integer overflow
> - >=1.3: detect overflow in such methods and throw exception immediately

Unify split by an empty match regex result across the platforms

> **Issue**: KT-21049
>
> **Component**: kotlin-stdlib (JVM)
>
> **Incompatible change type**: Behavioral
>
> **Short summary**: since Kotlin 1.3, unify behavior of `split` method by empty match regex across all platforms
>
> **Deprecation cycle**:
>
> - <1.3: behavior of described calls is different when comparing JS, JRE 6, JRE 7 versus JRE 8+
> - >=1.3: unify behavior across the platforms

Discontinued deprecated artifacts in the compiler distribution

> **Issue**: KT-23799
>
> **Component**: other
>
> **Incompatible change type**: Binary
>
> **Short summary**: Kotlin 1.3 discontinues the following deprecated binary artifacts:
>
> - `kotlin-runtime`: use `kotlin-stdlib` instead
> - `kotlin-stdlib-jre7/8`: use `kotlin-stdlib-jdk7/8` instead
> - `kotlin-jslib` in the compiler distribution: use `kotlin-stdlib-js` instead
>
> **Deprecation cycle**:

- 1.2.X: the artifacts were marked as deprecated, the compiler reported warning on usage of those artifacts
- >=1.3: the artifacts are discontinued

Annotations in stdlib

Issue: KT-21784

Component: kotlin-stdlib (JVM)

Incompatible change type: Binary

Short summary: Kotlin 1.3 removes annotations from the package `org.jetbrains.annotations` from stdlib and moves them to the separate artifacts shipped with the compiler: `annotations-13.0.jar` and `mutability-annotations-compat.jar`

Deprecation cycle:

- <1.3: annotations were shipped with the stdlib artifact
- >=1.3: annotations ship in separate artifacts

兼容模式

When a big team is migrating onto a new version, it may appear in an "inconsistent state" at some point, when some developers have already updated but others haven't. To prevent the former from writing and committing code that others may not be able to compile, we provide the following command line switches (also available in the IDE and Gradle/Maven):

- `-language-version X.Y` - compatibility mode for Kotlin language version X.Y, reports errors for all language features that came out later.
- `-api-version X.Y` - compatibility mode for Kotlin API version X.Y, reports errors for all code using newer APIs from the Kotlin Standard Library (including the code generated by the compiler).

Currently, we support the development for at least three previous language and API versions in addition to the latest stable one.

Kotlin 基金会

- Kotlin 基金会↗
- 语言委员会准则↗
- 提交不兼容变更指南↗
- Kotlin 品牌用途准则↗
- Kotlin 基金会 FAQ↗

Kotlin 基金会

Kotlin 基金会

语言委员会准则

语言委员会准则

提交不兼容变更指南

提交不兼容变更指南

Kotlin 品牌用途准则

Kotlin 品牌用途准则

Kotlin 基金会 FAQ

Kotlin 基金会 FAQ

Google Summer of Code

- 概述
- Google Summer of Code with Kotlin 2025↗
- Google Summer of Code with Kotlin 2024↗
- Google Summer of Code with Kotlin 2023↗

Google Summer of Code with Kotlin

The Kotlin Foundation provides mentorship for Google Summer of Code (GSoC), a global online program that introduces new contributors to open-source development.

Learn more about GSoC on the official Google Summer of Code website.

GSoC 2025: project ideas

The Kotlin Foundation has published a list of project ideas for Google Summer of Code 2025. If you're interested in contributing, check out the available projects and guidelines: .

Past GSoC projects with Kotlin

You can explore past Google Summer of Code projects and ideas with Kotlin:

-
-

Google Summer of Code with Kotlin 2025

Google Summer of Code with Kotlin 2025

Google Summer of Code with Kotlin 2024

Google Summer of Code with Kotlin 2024

Google Summer of Code with Kotlin 2023

Google Summer of Code with Kotlin 2023

安全

We do our best to make sure our products are free of security vulnerabilities. To reduce the risk of introducing a vulnerability, you can follow these best practices:

- Always use the latest Kotlin release. For security purposes, we sign our releases published on Maven Central with these PGP keys:
  - Key ID: **kt-a@jetbrains.com**
  - Fingerprint: **2FBA 29D0 8D2E 25EE 84C1 32C3 0729 A0AF F899 9A87**
  - Key size: **RSA 3072**
- Use the latest versions of your application's dependencies. If you need to use a specific version of a dependency, periodically check if any new security vulnerabilities have been discovered. You can follow the guidelines from GitHub or browse known vulnerabilities in the CVE base.

We are very eager and grateful to hear about any security issues you find. To report vulnerabilities that you discover in Kotlin, please post a message directly to our issue tracker or send us an email.

For more information on how our responsible disclosure process works, please check the JetBrains Coordinated Disclosure Policy.

社区

- 贡献力量
- KUG 准则
- Kotlin 之夜准则
- Kotlin Slack 的行为准则与指南

为 Kotlin 贡献力量

Kotlin is an open-source project under the Apache 2.0 License. The source code, tooling, documentation, and even this website are maintained on GitHub. Kotlin is developed by JetBrains, but we are always on the lookout for more people to help us.

Participate in Early Access Preview

You can help us improve Kotlin by participating in Kotlin Early Access Preview (EAP) and providing us with your valuable feedback.

For every release, Kotlin ships a few preview builds where you can try out the latest features before they go to production. You can report any bugs you find to our issue tracker YouTrack and we will try to fix them before a final release. This way, you can get bug fixes earlier than the standard Kotlin release cycle.

Contribute to the compiler and standard library

If you want to contribute to the Kotlin compiler and standard library, go to JetBrains/Kotlin GitHub, check out the latest Kotlin version, and follow the instructions on how to contribute.

You can help us by completing open tasks. Please keep an open line of communication with us because we may have questions and comments on your changes. Otherwise, we won't be able to incorporate your contributions.

Contribute to the Kotlin IDE plugin

Kotlin IDE plugin is a part of the IntelliJ IDEA repository.

To contribute to the Kotlin IDE plugin, clone the IntelliJ IDEA repository and follow the instructions on how to contribute.

Contribute to other Kotlin libraries and tools

Besides the standard library that provides core capabilities, Kotlin has a number of additional (kotlinx) libraries that extend its functionality. Each kotlinx library is developed in a separate repository, has its own versioning and release cycle.

If you want to contribute to a kotlinx library (such as kotlinx.coroutines or kotlinx.serialization) and tools, go to Kotlin GitHub, choose the repository you are interested in and clone it.

Follow the contribution process described for each library and tool, such as kotlinx.serialization, ktor and others.

If you have a library that could be useful to other Kotlin developers, let us know via feedback@kotlinlang.org.

Contribute to the documentation

If you've found an issue in the Kotlin documentation, feel free to check out the documentation source code on GitHub and send us a pull request. Follow these guidelines on style and formatting.

Please keep an open line of communication with us because we may have questions and comments on your changes. Otherwise, we won't be able to incorporate your contributions.

Translate documentation to other languages

You are welcome to translate the Kotlin documentation into your own language and publish the translation on your website. However, we won't be able to host your translation in the main repository and publish it on kotlinlang.org.

This site is the official documentation for the language, and we ensure that all the information here is correct and up to date. Unfortunately, we won't be able to review documentation in other languages.

译注：补充或改善中文站（即本站）翻译，可以 fork Kotlin 语言中文站项目并提 Pull Request。 目前《翻译指南》还在制订中，不过 #35 有一些草稿， 初次翻译请务必先阅读 #35 及其中链接对应的内容。

Hold events and presentations

If you've given or just plan to give presentations or hold events on Kotlin, please fill out the form. We'll feature them on the event list.

中文站翻译贡献者

Kotlin 编程语言中文站翻译贡献者如下（排名不分先后）：

- 灰蓝天际

- 晓_晨DEV
- S_arige
- dingsx
- 空白
- LyndonChin
- Jacky Xu
- wahchi
- _Y
- EasonZhou
- pecpwee
- Airoyee
- 化缘
- drakeet
- cx9527
- zsn012
- xwSurfer
- chiahaolu
- DemoJameson
- SnakeEys
- wssccc
- 592357390
- imknown
- XuToTo
- ovear
- boris1993
- iamhefang
- Maxwell1987
- henryhuang
- 高金龙
- shiyongkun
- yangxiaobinhaoshuai
- 乔禹昂
- capcdk
- luozejiaqun
- shuhuaxie
- okou19900722
- jonzoro
- ypxgxf
- cnfn
- Mikyou

- HoshinoTented
- wangkezun
- TangHuaiZhe
- singlepig
- phxnirvana
- Farubaba
- renzhi(任智)
- 刘文俊
- johnpoint
- whataa
- caixingke
- clarkhan
- yinhutjfox
- Him188
- 常少威
- coderWangJie
- javaisgood
- jimmysuncpt
- star0224
- yourzeromax
- Yue-plus
- CZKGO
- wsq22
- gentrio
- easyfirst
- Guolianxing
- BobEve
- 7forz
- Andy1245-dot
- Aaro0n
- plain-dev
- willawang8908
- singleNeuron
- ShreckYe
- deanssss
- cznno
- hepan
- small-ora
- ijunjie
- oncealong

- gtn1024
- jixiaoyong
- shisheng1998
- Smileslime47
- sammary114
- cos800

KUG 准则

A Kotlin User Group, or KUG, is a community that is dedicated to Kotlin and that offers you a place to share your Kotlin programming experience with like-minded people.

To become a KUG, your community should have some specific features shared by every KUG. It should:

- Provide Kotlin-related content, with regular meetups as the main form of activity.
- Host regular events (at least once every 3 months) with open registration and without any restriction for attendance.
- Be driven and organized by the community, and it should not use events to earn money or gain any other business benefits from members and attendees.
- Follow and ensure a code of conduct to provide a welcoming environment for attendees of any background and experience (check out our recommended Code of Conduct).

There are no limits regarding the format for KUG meetups. They can take place in whatever fashion works best for the community, whether that includes presentations, hands-on labs, lectures, hackathons, or informal beer-driven get-togethers.

> For Kotlin User Group brand assets, see Kotlin brand assets documentation.
>
> 

How to run a KUG?

- To promote group cohesion and prevent miscommunication, we recommend keeping to a limit of one KUG per city. Check out the list of KUGs to see if there is already a KUG in your area.
- Use the official KUG logo and branding. Check out the branding guidelines.
- Keep your user group active. Run meetups regularly, at least once every 3 months.
- Announce your KUG meetups at least 2 weeks in advance. The announcement should contain a list of talks and the names of the speakers, as well as the location, timing, and any other crucial info about the event.
- KUG events should be free or, if you need to cover organizing expenses, limit prices to a maximum of 10 USD.
- Your group should have a code of conduct available for all members.

If your community has all the necessary features and follows these guidelines, you are ready to Apply to be a new KUG.

Have a question? Contact us

Support for KUGs from JetBrains

Active KUGs that host at least 1 meetup every 3 months can apply for the community support program, which includes:

- Official KUG branding.
- A special entry on the Kotlin website.
- Free licenses for JetBrains products to raffle off at meetups.
- Priority support for Kotlin events and campaigns.
- Help with recruiting Kotlin speakers for your events.

Support from JetBrains for other tech communities

If you organize any other tech communities, you can apply for support as well. By doing so, you may receive:

- Free licenses for JetBrains products to raffle off at meetups.
- Information about Kotlin official events and campaigns.
- Kotlin stickers.
- Help with recruiting Kotlin speakers for your events.

Kotlin 之夜准则

Kotlin Night is a meetup that includes 3-4 talks on Kotlin or related technologies.

> For Kotlin Night brand assets, see Kotlin brand assets documentation.
>
> 

Event guidelines

- Please use the branding materials we've provided. Having all events and materials in the same style will help keep the Kotlin Night experience consistent.
- Kotlin Night should be a free event. A minimal fee can be charged to cover expenses, but it should remain a non-profit event.
- The event should be announced publicly and open for all people to attend without any kind of discrimination.
- If you publish the contents of the talks online after the event, they must be free and accessible to everyone, without any sign-up or registration procedures.
- Recordings are optional but recommended, and they should also be made available. If you decide to record the talks, we suggest having a plan to ensure the quality is good.
- The talks should primarily be about Kotlin and should not focus on marketing or sales.
- The event can serve food and drinks optionally.

Event requirements

JetBrains is excited to support your Kotlin Night event. Because we want all events to provide the same high-quality experience, we need organizers to ensure that some basic requirements are met for the event to receive JetBrains support. As an organizer, you are responsible for the following aspects of the event:

1. The location and everything required to host the event, including booking a comfortable venue. Please make sure that:
    - All the participants are aware of the exact date, place, and starting time of the event, along with the event schedule and program.
    - There is enough space as well as food and beverages, if you provide them, for everyone.
    - You have a plan with your speakers. This includes a schedule, topics, abstracts for the talks, and any necessary equipment for the presentations.
2. Content and speakers

    - Feel free to invite presenters from your local community, from neighboring countries, or even from all over the globe. You don't have to have any JetBrains representatives or speakers at your event. However, we are always happy to hear about more Kotlin Nights, so feel free to notify us.
3. Announcements and promotion
    - Announce your event at least three weeks before the date of a meetup.
    - Include the schedule, topics, abstracts, and speaker bios in the announcement.
    - Spread the word on social media.
4. Providing event material to JetBrains after the event
    - We would be glad to announce your event at kotlinlang.org, and we would appreciate it if you provided slides and video materials for a follow-up posting.

JetBrains support

JetBrains provides support with:

- Access to Kotlin Night Branding, which includes the name and logos
- Merchandise, such as stickers and t-shirts for speakers and small souvenirs for attendees
- A listing for the event on the Kotlin Talks page
- Help to reach out to speakers to take part in the event, if necessary
- Help to find a location if possible (via contacts, etc.), as well as help to identify possible partnerships with local businesses

Kotlin Slack 的行为准则与指南

Kotlin Slack aims to be an inclusive space committed to providing a friendly, safe and welcoming environment for all, regardless of gender, sexual orientation, ability, ethnicity, socioeconomic status, and religion (or lack thereof).

For this to be the case, it is vital that we all follow a basic set of guidelines and most importantly, adhere to the code of conduct. As such, please make sure you read this Code of Conduct and Basic Usage Guidelines in its entirety. This isn't your regular License Agreement that you should scroll through and agree blindly. It's here for a reason and also contains practical information.

- How to behave
- How not to behave
- How to report issues
- Moderators
- Basic usage guidelines
- Moderators
- Copyright

How to behave

- Participate in an authentic and active way. In doing so, you contribute to the health and longevity of this community.
- Exercise consideration, respect and empathy in your speech and actions. Remember, we have all been through different stages of learning when adopting technologies.
- Refrain from demeaning, discriminatory, or harassing behavior and speech.
- Disagreements on things are fine, argumentative behavior or trolling are not.

How not to behave

- Do *not* perform threats of violence or use violent language directed against another person.
- Do *not* make jokes of sexist, racist, homophobic, transphobic, ableist or otherwise discriminatory nature, or use language of this nature.
- Do *not* post or display sexually explicit or violent material.
- Do *not* post or threaten to post other people's personally identifying information ("doxing").
- Do *not* make personal insults, particularly those related to gender, sexual orientation, race, religion, or disability.
- Do *not* engage in sexual attention. This includes, sexualized comments or jokes and sexual

advances.
- Do *not* advocate for, or encourage, any of the above behavior.

How to report issues

If someone is acting inappropriately or violating this Code of Conduct in any shape or form, and they are not receptive to your feedback, or you prefer not to confront them, please get in touch with one of the Administrators. The main Administrators are Ilya Ryzhenkov (@orangy) and Hadi Hariri (@hhariri).

Basic usage guidelines

There are over 50 000 users on many different timezones using the Kotlin Slack, and the number is growing rapidly. The influx of messages per day and the numerous channels can lead to a lot of noise and little value in the long run. If we all follow a set of guidelines, it can help make things more useful and bearable for everyone.

- Please remember this is a Slack team for Kotlin. If you have generic questions that are about a technology (which may or may not be used with Kotlin), maybe it's better to use another forum, such as StackOverflow.
- This Slack is not an official support channel. It is a place where the community hangs out along with some members of the JetBrains and Kotlin team (suffix [JetBrains]).
- This Slack is not the best place for reporting bugs. For this, please use YouTrack.
- Find the right channel to ask your question. There are channels for pretty much every topic.
- New channels should be focused around a well-known technology or area of interest. Create them sparingly. And before requesting for a new channel, make sure it doesn't exist in any other shape or form. Ask an administrator to create a new channel on the #meta channel. If you do end up creating a channel, make sure you set the topic.
- Please refrain from cross-posting the same message on multiple channels. It is considered spamming.
- When you ask a question, please be patient. Don't repeat it. Or at least not immediately.
- Please do not ping or mention someone directly to get your questions answered, especially project owners, whether the project is Kotlin or other, unless they specifically indicate that you can.
- Don't split messages into multiple ones. Ask it all in a single message.
- Use code blocks. Don't paste code as plain text. Slack supports Kotlin markup. If the code you are pasting is longer than a few lines, use "Code or text snippet" available from `+` menu next to message input. Only first few lines will be displayed to all users and people interested in the code can expand your code to look into details.

- While it's polite to say "Hi" or "I have a question" before asking something, it doesn't scale and leads to noise. You can say hello in the same message you ask the question if you like.
- Using threads is not required, but do take into account that it does allow people to more easily follow conversations, especially those jumping in late.
- Use reactions to show gratitude as opposed to a message. It reduces the noise and gets the message across.
- Don't use reactions to tell people they're in the wrong place or asking the wrong question. If they're on the wrong channel, point them to the right one. If their question is badly worded, help them correct it. Have empathy.
- Take into account cultural differences. As they say, what the British say isn't what the British mean. Remember this both as the receiver and producer of messages.
- Don't create integrations with Slack. Given this is running on the free tier, the number is limited, and thus it won't be approved.
- @channel, @here and other forms of notifications are disabled, even if you own a channel (unfortunately, Slack doesn't allow fine-grained permissions).
- Use the #meta channel if you're unsure about how to do something.

Moderators

A moderator is a member of the Kotlin Slack community who volunteers to keep the kotlinlang Slack a safe and welcoming place for other members. Moderators are approved by kotlinlang Slack administrators. Their responsibilities and the guidelines they follow are described below.

Responsibilities

- Monitor Slack channels to ensure compliance with the Code of Conduct.
- Help community members follow the rules.
- If a channel has additional rules, ensure these are visible and up to date.
- Monitor the #reports and #meta channels.
- Participate in the private #moderators channel.

Guidelines

- Moderators communicate in a friendly manner and do their best to help other community members.
- Any changes suggested by a moderator should be discussed by all moderators and administrators before coming into effect.

Process

- Each moderator oversees at least one channel, or more if possible.
- When moderators notice inappropriate behavior, they explain the rules and guidelines to the members and help figure out how to correct the unwanted behavior. If the unwanted behavior persists, moderators report the case to administrators for further actions. In the case of obvious and/or particularly blatant violations, offending members can be reported immediately.
- To set up or update a channel's rules, the moderator initiates a change proposal and discusses it with the appropriate community members.
- Communication regarding moderation takes place in the private #moderators channel.
- If community members notice that a moderator is ignoring their responsibilities (or engaging in any unwanted behavior), they can alert administrators.

Moderator status acquisition and revocation process

- Any member is welcome to apply to be a moderator once they have been in the Slack community for longer than 6 months.
- Moderator status is provided based upon the decision of the administrators.
- Moderator status can be revoked if (this list is not exhaustive and can be updated on a case-by-case basis):
  - A moderator ignores their responsibilities, which leads to issues in a channel.
  - A moderator violates these guidelines.

The list of moderators

At the moment we have 17 confirmed moderators, and they will be responsible for observing a total of 64 channels. For all other channels, if any issues arise, members can ping any of the moderators or admins directly, or they can post a message in #meta.

- Alexander Nozik (@altavir) – #mathematics, #science, #datascience, #education
- Anderson Lameck (@andylamax) – #coroutines, #kotlin-native, #webassembly, #serialization, #random, #javascript, #react
- Andrey Mischenko (@gildor) – #coroutines, #android, #gradle, #multiplatform, #getting-started, #kotlin-asia, #singapore
- Eric Ampire (@Eric Ampire) – #events
- Holger Steinhauer (@Holger Steinhauer) – #fosdem, #berlin, #vkug, #german-lang, #server
- Louis CAD (@louiscad) – #coroutines, #android
- Magda Miu (@Magda Miu) – #android, #100daysofkotlin
- Maryam Alhuthayfi (@Maryam Alhuthayfi) – #compose, #books, #ksp, #eap, #dagger, and #kontributors.
- Nicola Corti (@gammax) – #feed, #detekt, #appintro, #chucker, #fosdem, #london, #berlin, #hamburg, #sweden, #ktlint,

kug-leads, #kotlinconf, #koin, #koin-dev, #spek, #vkug

- Paulien van Alst (@Paulien van Alst) – #detekt, #koin, #mockk
- Qian Jin (@qian) – #kotlin-native, #android-studio, #french
- Raul Raja (@raulraja) - #arrow, #arrow-contributors, #arrow-meta
- Sam Sam (@sam) – #kotest, #kotest-contributors
- Simon Vergauwen (@simon.vergauwen) – #arrow, #arrow-contributors, #arrow-meta
- Youssef Shoaib (@Youssef Shoaib) – #getting-started, #feed, #language-evolution, #language-proposals
- Zach Klippenstein (@Zach Klippenstein) – #coroutines, #compose, #compose-desktop, #compose-web, #squarelibraries,

library-development

Copyright

This Code of Conduct is distributed under a Creative Commons Attribution-ShareAlike license.
Portions of a text derived from the Citizen Code of Conduct.

Kotlin 素材

Kotlin 品牌素材

宣传资料

宣传资料