

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
概述	1.3
Kotlin 多平台	1.3.1
Kotlin 用于服务器端开发	1.3.2
Kotlin 用于 Android 开发	1.3.3
Kotlin 用于 JavaScript 开发	1.3.4
Kotlin 用于原生开发	1.3.5
Kotlin 用于数据科学	1.3.6
Kotlin 用于竞技程序设计	1.3.7
Kotlin 的新特性	1.4
1.6.0 的新特性	1.4.1
早期版本	1.4.2
1.5.30 的新特性	1.4.2.1
1.5.20 的新特性	1.4.2.2
1.5.0 的新特性	1.4.2.3
1.4.30 的新特性	1.4.2.4
1.4.20 的新特性	1.4.2.5
1.4.0 的新特性	1.4.2.6
1.3 的新特性	1.4.2.7
1.2 的新特性	1.4.2.8
1.1 的新特性	1.4.2.9
基础	1.5
基本语法	1.5.1
习惯用法	1.5.2
例学 Kotlin ↗	1.5.3
编码规范	1.5.4
概念	1.6

1.5.30 的新特性

类型	1.6.1
基本类型	1.6.1.1
类型检测与类型转换	1.6.1.2
控制流程	1.6.2
条件与循环	1.6.2.1
返回与跳转	1.6.2.2
异常	1.6.2.3
包与导入	1.6.3
类与对象	1.6.4
类	1.6.4.1
继承	1.6.4.2
属性	1.6.4.3
接口	1.6.4.4
函数式 (SAM) 接口	1.6.4.5
可见性修饰符	1.6.4.6
扩展	1.6.4.7
数据类	1.6.4.8
密封类	1.6.4.9
泛型: in、out、where	1.6.4.10
嵌套类	1.6.4.11
枚举类	1.6.4.12
内联类	1.6.4.13
对象表达式与对象声明	1.6.4.14
委托	1.6.4.15
属性委托	1.6.4.16
类型别名	1.6.4.17
函数	1.6.5
函数	1.6.5.1
lambda 表达式	1.6.5.2
内联函数	1.6.5.3
操作符重载	1.6.5.4
类型安全的构建器	1.6.6

1.5.30 的新特性

空安全	1.6.7
相等性	1.6.8
this 表达式	1.6.9
异步程序设计技术	1.6.10
协程	1.6.11
注解	1.6.12
解构声明	1.6.13
反射	1.6.14
多平台开发	1.7
Kotlin 多平台用于 iOS 与 Android	1.7.1
Kotlin 多平台移动端入门	1.7.1.1
搭建环境	1.7.1.2
创建第一个跨平台移动端应用——教程	1.7.1.3
了解移动端项目结构	1.7.1.4
让 Android 应用程序能用于 iOS——教程	1.7.1.5
发布应用程序	1.7.1.6
Kotlin 多平台用于其他平台	1.7.2
Kotlin 多平台入门	1.7.2.1
了解多平台项目结构	1.7.2.2
手动设置目标	1.7.2.3
创建多平台库	1.7.3
创建多平台库	1.7.3.1
发布多平台库	1.7.3.2
创建并发布多平台库——教程	1.7.3.3
共享代码原则	1.7.4
平台间共享代码	1.7.4.1
接入平台相关 API	1.7.4.2
迁移多平台项目到 Kotlin 1.4.0	1.7.4.3
添加依赖项	1.7.5
添加依赖项	1.7.5.1
添加 Android 依赖项	1.7.5.2
添加 iOS 依赖项	1.7.5.3

1.5.30 的新特性

运行测试	1.7.6
构件编译项	1.7.7
配置编译项	1.7.7.1
构建最终原生二进制文件	1.7.7.2
多平台 Gradle DSL 参考	1.7.8
样例	1.7.9
FAQ	1.7.10
向团队介绍跨平台移动端开发	1.7.11
平台	1.8
JVM	1.8.1
Kotlin/JVM 入门	1.8.1.1
与 Java 比较	1.8.1.2
在 Kotlin 中调用 Java	1.8.1.3
在 Java 中调用 Kotlin	1.8.1.4
Spring	1.8.1.5
使用 Spring Boot 创建用到数据库的 RESTful web 服务——教程	
Spring 框架 Kotlin 文档 ↗	1.8.1.5.2 1.8.1.5.1
使用 Spring Boot 与 Kotlin 构建 web 应用程序——教程 ↗	1.8.1.5.3
使用 Kotlin 协程与 RSocket 创建聊天应用程序——教程 ↗	1.8.1.5.4
在 JVM 平台上用 JUnit 测试代码——教程	1.8.1.6
在项目中混用 Java 与 Kotlin——教程	1.8.1.7
在 Kotlin 中使用 Java 记录类型	1.8.1.8
从 Java 到 Kotlin 迁移指南	1.8.1.9
字符串	1.8.1.9.1
集合	1.8.1.9.2
JavaScript	1.8.2
以 Kotlin/JS for React 入门	1.8.2.1
搭建 Kotlin/JS 项目	1.8.2.2
运行 Kotlin/JS	1.8.2.3
开发服务器与持续编译	1.8.2.4
调试 Kotlin/JS 代码	1.8.2.5
在 Kotlin/JS 平台上运行测试	1.8.2.6

1.5.30 的新特性

Kotlin/JS 无用代码消除	1.8.2.7
Kotlin/JS IR 编译器	1.8.2.8
将 Kotlin/JS 项目迁移到 IR 编译器	1.8.2.9
Kotlin 用于 JS 平台	1.8.2.10
浏览器与 DOM API	1.8.2.10.1
在 Kotlin 中使用 JavaScript 代码	1.8.2.10.2
动态类型	1.8.2.10.3
使用来自 npm 的依赖	1.8.2.10.4
在 JavaScript 中使用 Kotlin 代码	1.8.2.10.5
JavaScript 模块	1.8.2.10.6
Kotlin/JS 反射	1.8.2.10.7
类型安全的 HTML DSL	1.8.2.10.8
用 Dukat 生成外部声明	1.8.2.10.9
Kotlin/JS 动手实践实验室	1.8.2.11
原生	1.8.3
Kotlin/Native 入门——在 IntelliJ IDEA 中	1.8.3.1
Kotlin/Native 入门——使用 Gradle	1.8.3.2
Kotlin/Native 入门——使用命令行编译器	1.8.3.3
与 C 语言互操作	1.8.3.4
与 C 语言互操作性	1.8.3.4.1
映射来自 C 语言的原始数据类型——教程	1.8.3.4.2
映射来自 C 语言的结构与联合类型——教程	1.8.3.4.3
映射来自 C 语言的函数指针——教程	1.8.3.4.4
映射来自 C 语言的字符串——教程	1.8.3.4.5
创建使用 C 语言互操作与 libcurl 的应用——教程	1.8.3.4.6
与 Objective-C 互操作性	1.8.3.5
与 Swift/Objective-C 互操作性	1.8.3.5.1
Kotlin/Native 开发 Apple framework——教程	1.8.3.5.2
CocoaPods 集成	1.8.3.6
CocoaPods 概述	1.8.3.6.1
添加对 Pod 库的依赖	1.8.3.6.2
使用 Kotlin Gradle 项目作为 CocoaPods 依赖项	1.8.3.6.3

1.5.30 的新特性

Kotlin/Native 库	1.8.3.7
平台库	1.8.3.8
Kotlin/Native 开发动态库——教程	1.8.3.9
不可变性与并发	1.8.3.10
处理并发	1.8.3.11
并发概述	1.8.3.11.1
并发可变性	1.8.3.11.2
并发与协程	1.8.3.11.3
调试 Kotlin/Native	1.8.3.12
符号化 iOS 崩溃报告	1.8.3.13
改进 Kotlin/Native 编译时间的技巧	1.8.3.14
Kotlin/Native FAQ	1.8.3.15
Kotlin/Native 中的并发	1.8.3.16
脚本	1.8.4
Kotlin 自定义脚本入门——教程	1.8.4.1
版本发布与路线图	1.9
Kotlin 版本发布	1.9.1
Kotlin 多平台移动端插件版本发布	1.9.2
Kotlin 插件版本发布	1.9.3
Kotlin 路线图	1.9.4
标准库	1.10
集合	1.10.1
集合概述	1.10.1.1
构造集合	1.10.1.2
迭代器	1.10.1.3
区间与数列	1.10.1.4
序列	1.10.1.5
集合操作概述	1.10.1.6
集合转换操作	1.10.1.7
过滤集合	1.10.1.8
加减操作符	1.10.1.9
分组	1.10.1.10

1.5.30 的新特性

取集合的一部分	1.10.1.11
取单个元素	1.10.1.12
排序	1.10.1.13
聚合操作	1.10.1.14
集合写操作	1.10.1.15
List 相关操作	1.10.1.16
Set 相关操作	1.10.1.17
Map 相关操作	1.10.1.18
作用域函数	1.10.2
选择加入要求	1.10.3
官方库	1.11
协程 (kotlinx.coroutines)	1.11.1
协程指南	1.11.1.1
协程基础	1.11.1.2
Kotlin 协程与通道介绍 ↗	1.11.1.3
取消与超时	1.11.1.4
组合挂起函数	1.11.1.5
协程上下文与调度器	1.11.1.6
异步流	1.11.1.7
通道	1.11.1.8
协程异常处理	1.11.1.9
共享的可变状态与并发	1.11.1.10
select 表达式 (实验性的)	1.11.1.11
使用 IntelliJ IDEA 调试协程——教程	1.11.1.12
使用 IntelliJ IDEA 调试 Kotlin Flow——教程	1.11.1.13
序列化 (kotlinx.serialization)	1.11.2
Ktor ↗	1.11.3
API 参考	1.12
标准库 (stdlib) ↗	1.12.1
测试库 (kotlin.test) ↗	1.12.2
协程 (kotlinx.coroutines) ↗	1.12.3
Ktor ↗	1.12.4

1.5.30 的新特性

语言参考	1.13
关键字与操作符	1.13.1
语法 ↗	1.13.2
语言规范 ↗	1.13.3
工具	1.14
构建工具	1.14.1
Gradle	1.14.1.1
Maven	1.14.1.2
Ant	1.14.1.3
IntelliJ IDEA	1.14.2
迁移到 Kotlin 代码风格	1.14.2.1
Android Studio ↗	1.14.3
Eclipse IDE	1.14.4
运行代码片段	1.14.5
编译器	1.14.6
Kotlin 命令行编译器	1.14.6.1
Kotlin 编译器选项	1.14.6.2
编译器插件	1.14.7
全开放编译器插件	1.14.7.1
No-arg 编译器插件	1.14.7.2
带有接收者的 SAM 编译器插件	1.14.7.3
使用 kapt	1.14.7.4
Lombok 编译器插件	1.14.7.5
Kotlin 符号处理 (KSP) API	1.14.8
KSP 概述	1.14.8.1
KSP 快速入门	1.14.8.2
为什么选用 KSP	1.14.8.3
KSP 示例	1.14.8.4
KSP 如何为 Kotlin 代码建模	1.14.8.5
Java 注解处理对应到 KSP 参考	1.14.8.6
增量处理	1.14.8.7
多轮次处理	1.14.8.8

1.5.30 的新特性

KSP 与 Kotlin 多平台	1.14.8.9
在命令行运行 KSP	1.14.8.10
常见问题	1.14.8.11
Kotlin 与 TeamCity 的持续集成	1.14.9
KDoc 与 Dokka	1.14.10
Kotlin 与 OSGi	1.14.11
学习资料	1.15
学习资料概述	1.15.1
例学 Kotlin ↗	1.15.2
Kotlin 心印	1.15.3
Kotlin 基础课程 ↗	1.15.4
Advent of Code 谜题	1.15.5
图书	1.15.6
在 IDE 中学习 (EduTools)	1.15.7
学习 Kotlin	1.15.7.1
讲授 Kotlin	1.15.7.2
其他资源	1.16
FAQ	1.16.1
早期访问计划 (EAP)	1.16.2
参与 Kotlin 早期访问计划	1.16.2.1
安装 Kotlin EAP 插件	1.16.2.2
配置构建采用 EAP	1.16.2.3
贡献力量	1.16.3
Kotlin 演进	1.16.4
演进原则	1.16.4.1
Kotlin 各组件的稳定性	1.16.4.2
Kotlin 各组件的稳定性 (1.4 之前)	1.16.4.3
兼容性	1.16.4.4
Kotlin 1.6 的兼容性指南	1.16.4.4.1
Kotlin 1.5 的兼容性指南	1.16.4.4.2
Kotlin 1.4 的兼容性指南	1.16.4.4.3
Kotlin 1.3 的兼容性指南	1.16.4.4.4

1.5.30 的新特性

兼容模式	1.16.4.4.5
Kotlin 基金会	1.16.5
Kotlin 基金会 ↗	1.16.5.1
语言委员会准则 ↗	1.16.5.2
提交不兼容变更指南 ↗	1.16.5.3
Kotlin 品牌用途准则 ↗	1.16.5.4
Kotlin 基金会 FAQ ↗	1.16.5.5
安全	1.16.6
Kotlin 品牌资料	1.16.7

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，尤其是今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 1.6.10。最近一次更新：2022-03-28。

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://hljtj.me/>）以及公众号与[微博](#)：



公众号



微博

Kotlin 文档

最新稳定版：1.6.10

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

快速了解 Kotlin 语法：关键字、操作符、程序结构

例学 Kotlin

Kotlin 语法简单评注版示例

心印

让你熟悉 Kotlin 的编程练习

命令行编译器

下载并安装 Kotlin 编译器

精选主题

标准库 API 参考

Living essentials for everyday work with Kotlin: IO, files, threading, collections, and much more

编码规范

Recommendations on the code organization, formatting, and naming

基本类型

Kotlin type system: numbers, strings, arrays, and other built-in types

控制流程

Conditions and loops: if, when, for, while

1.5.30 的新特性

空安全

Nullable and non-null types, elvis operator, safe calls and casts

协程

Concurrency: coroutines, flows, channels

新特性

Kotlin 1.6.0 的新特性

Latest features: sealed whens, suspend functions as supertypes, and more

Kotlin 公共路线图

Future plans on Kotlin development

Kotlin 多平台移动端

Android Studio 中 KMM 入门

Useful links to help you start using the SDK

用于 Android Studio 的插件版本发布

Features of the Kotlin Multiplatform Mobile plugin

Kotlin 多平台 Multiverse

Videos about Kotlin Multiplatform Mobile on our YouTube channel

学习 Kotlin

例学 Kotlin

Simple annotated examples for the Kotlin syntax

心印

Programming exercises to get you familiar with Kotlin

JetBrains 学院

Kotlin Basics track on JetBrains Academy

Advent of Code

Code puzzles in idiomatic Kotlin

1.5.30 的新特性

动手实践教程

Complete long-form tutorials to fully grasp a technology

IntelliJ IDEA 中的 EduTools

An IDE plugin for learning and teaching programming languages

图书

The books that we've reviewed and recommend for learning Kotlin

在 YouTube 上观看 Kotlin 视频

Kotlin YouTube channel

Official Kotlin YouTube channel

Kotlin in Spring Framework

Tutorials on using Kotlin with Spring

Webinars with experts

Webinars on using Kotlin for server-side development

Kotlin Multiplatform Multiverse

Tutorials on using Kotlin Multiplatform Mobile

Competitive programming

Kotlin for competitive programming

Kotlin standard library

Tutorials on using the standard library

Talking Kotlin podcast

Talking Kotlin podcast

Kotlin for educators

Stories about teaching Kotlin

Kotlin Online Event 2021

Talks from the Kotlin Online Event 2021

保持联系并贡献力量

为 Kotlin 贡贡献力量

The ways in which you can help the development of Kotlin

1.5.30 的新特性

[参与抢先体验计划 \(EAP\)](#)

Preview versions for trying out features before their official releases

[加入 Kotlin Slack](#)

Official public Kotlin Slack

[在 Twitter 上关注 Kotlin](#)

Official Kotlin Twitter

[在 Reddit 上探讨](#)

Kotlin on Reddit

[参与 Stack Overflow 的讨论](#)

Kotlin on Stack Overflow

Kotlin 入门

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。

选择它来构建强大的应用程序吧！

学习 Kotlin 基础知识

- 如果你已熟悉一门或多门编程语言并想学习 Kotlin，请从这些 [Kotlin 学习资料](#) 开始。
- 如果 Kotlin 是你的第一门编程语言，我们建议从 [《Atomic Kotlin》这本书](#) 开始，或者在 JetBrains 学院报名免费的 [Kotlin 基础课程](#)。

使用 Kotlin 创建强大的应用程序

【后端应用】

以下是开发 Kotlin 服务器端应用程序的第一步。

- 安装 [IntelliJ IDEA 最新版](#)。
- 创建第一个后端应用程序：
 - 从头开始，[使用 IntelliJ IDEA 项目向导创建一个基本的 JVM 应用程序](#)。
 - 如果倾向于更健壮的示例，请选择以下框架之一来创建一个项目：

Spring	Ktor
<p>一套成熟的框架族，拥有全球数百万开发人员都在用的成熟生态系统。</p> <ul style="list-style-type: none">使用 Spring Boot 创建 RESTful web 服务。使用 Spring Boot 与 Kotlin 构建 web 应用程序。Spring Boot 与 Kotlin 及 RSocket 合用。	<p>适用于架构决策时看重自由度的开发者的轻量级框架。</p> <ul style="list-style-type: none">使用 Ktor 创建 HTTP API。使用 Ktor 创建 WebSocket 聊天。使用 Ktor 创建交互式网站。发布服务器端 Kotlin 应用程序：Ktor on Heroku。

1.5.30 的新特性

3. 在应用程序中使用 **Kotlin** 库与第三方库。了解关于[向项目中添加库与工具依赖项](#)的更多信息。

- [Kotlin 标准库](#)提供了许多实用的内容，例如[集合与协程](#)。
- 看看这些[用于 Kotlin 的第三方框架、库与工具](#)。

4. 了解关于 **Kotlin** 用于服务器端开发的更多信息：

- [如何编写第一个单元测试](#)。
- [如何在应用程序中混用 Kotlin 与 Java 代码](#)。

5. 加入 **Kotlin** 服务器端社区：

-  [Slack](#): 获取邀请并加入 `#getting-started`、`#server`、`#spring` 或 `#ktor` 频道。
-  [StackOverflow](#): 订阅“kotlin”、“spring-kotlin”或“ktor”标签。

6. 关注 **Kotlin**:  [Twitter](#)、 [Reddit](#)、 [Youtube](#)，不要错过任何重要的生态系统更新。

如果遇到任何困难和问题，请在我们的[问题跟踪系统](#)提报。

【跨平台移动端应用】

在此可以了解到如何使用 **Kotlin 多平台移动端** 开发及改进跨平台移动端应用程序。

1. [搭建用于跨平台移动端开发的环境](#)。

2. 创建第一个用于 iOS 与 Android 应用程序：

- 从头开始，[使用项目向导创建一个基本的跨平台移动端应用程序](#)。
- 如果有既有的 Android 应用程序并想让它跨平台，那么请完成[让 Android 应用程序也能用于 iOS 教程](#)。
- 如果更倾向于现实生活的示例，那么请克隆并使用既有项目，例如[动手实践教程](#)中的网络与数据存储项目或者任意[样例项目](#)。

3. 使用一整套多平台库 在共享模块中只实现一次所需的业务逻辑。了解关于[添加依赖项](#)的更多信息。

1.5.30 的新特性

库	详情
Ktor	文档 。
Serialization	文档及样例 。
Coroutines	文档及样例 。
DateTime	文档 。
SQLDelight	第三方库。 文档 。

还可以在[社区驱动列表](#)中找到多平台库。



1. 了解关于 Kotlin 多平台移动端的更多信息：

- 了解关于 [Kotlin 多平台](#)的更多信息。
- 浏览 [GitHub 上的样例](#)。
- [创建并发布多平台库](#)。
- 了解 [Netflix](#)、[VMWare](#)、[Yandex](#) 以及[许多其他公司](#)如何使用 Kotlin 多平台。

2. 加入 Kotlin 多平台社区：

-  Slack: [获取邀请](#)并加入 #getting-started 与 #multiplatform 频道。
-  StackOverflow: 订阅“kotlin-multiplatform” 标签。

3. 关注 Kotlin: Twitter、 Reddit、 Youtube, 不要错过任何重要的生态系统更新。

如果遇到任何困难和问题, 请在我们的[问题跟踪系统](#)提报。

【前端 web 应用】

Kotlin 能够将 Kotlin 代码、Kotlin 标准库以及任何兼容的依赖项转换为 JavaScript。

在此可以了解到如何使用 [Kotlin/JS](#) 开发及改进前端 web 应用程序。

1. 安装[IntelliJ IDEA 最新版](#)。

2. 创建第一个前端 web 应用程序：

- 从头开始, 使用 [IntelliJ IDEA 项目向导](#)创建一个基本的浏览器应用程序。
- 如果倾向于更健壮的示例, 那么请完成[使用 React 与 Kotlin/JS 构建 web 应用程序](#)动手实践教程。它有一个可以作为你自己项目良好起点的样例项目, 其中包含有用的片段和模板。
- 查看 [Kotlin/JS 样例](#)列表, 了解关于如何使用 Kotlin/JS 的更多看法。

1.5.30 的新特性

3. 在应用程序中使用库。了解[添加依赖项](#)的更多信息。

库	详情
stdlib	默认所有项目都包含了的 Kotlin 标准库。
kotlinx.browser	用于访问浏览器相关功能的 Kotlin 库，包括典型的顶层对象，如 <code>document</code> 与 <code>window</code> 。
kotlinx.html	使用静态类型的 HTML 构建器生成 DOM 元素的 Kotlin 库。
Ktor	用于联网的 Kotlin 多平台库。
KVision	用于 Kotlin/JS 的一个第三方面向对象 web 框架。
fritz2	一个轻量级、高性能、独立的第三方库，用于在 Kotlin 中构建高度依赖协程与流的反应式 web 应用。
Doodle	一个基于矢量的第三方 UI 框架，使用浏览器的功能来绘制用户界面。
Compose for Web, Compose Multiplatform 的一部分	将 谷歌的 Jetpack Compose UI 工具包 带到浏览器的 JetBrains 框架。
kotlin-wrappers	为最流行的 JavaScript 框架之一提供方便的抽象与深度集成。Kotlin wrappers 还为许多类似技术提供支持，例如 <code>react-redux</code> 、 <code>react-router</code> 或者 <code>styled-components</code> 。

1. 了解关于 Kotlin 用于前端 web 开发的更多信息：

- 新版 Kotlin/JS IR 编译器（目前处于 Beta 状态）。
- 使用来自 npm 的依赖项。
- 在 JavaScript 中使用 Kotlin 代码。

2. 加入 Kotlin 前端 web 社区：

-  Slack：获取邀请并加入 `#getting-started` 与 `#javascript` 频道。
-  StackOverflow：订阅“kotlin-js”标签。

3. 关注 Kotlin： Twitter、 Reddit、 Youtube，不要错过任何重要的生态系统更新。

如果遇到任何困难和问题，请在我们的[问题跟踪系统](#)提报。

【Android 应用】

1.5.30 的新特性

- 如果希望开始使用 Kotlin 用于 Android 开发, 请阅读 [谷歌对 Android 上 Kotlin 入门的建议。](#)
- 如果是 Android 新手并且想学习使用 Kotlin 创建应用程序, 请查看[这门 Udacity 课程。](#)

关注 Kotlin:  Twitter、 Reddit 与  Youtube, 不要错过任何重要的生态系统更新。

【多平台库】

支持多平台程序设计是 Kotlin 的主要优势之一。它减少了为不同平台编写与维护相同代码所花的时间, 同时保留了原生编程的灵活性与优势。

在此可以了解到如何开发并发布多平台库:

1. 安装[IntelliJ IDEA 最新版](#)。
2. 创建多平台库:
 - 从头开始, [创建一个基本项目](#)。
 - 如果倾向于更健壮的示例, 那么请完成[创建并发布多平台库](#)教程。它展示了如何为 JVM、JS 与原生平台创建多平台库, 对其进行测试并发布到本地 Maven 仓库。
 - 使用[这一动手实践](#)构建一个全栈 web 应用程序。
3. 在应用程序中使用库。了解关于[添加对库的依赖](#)的更多内容。

库	详情
Ktor	文档与样例 。
Serialization	文档与样例 。
Coroutines	文档 。
DateTime	文档 。

还可以在[社区驱动列表](#)中找到多平台库。



1. 了解关于 Kotlin 多平台程序设计的更多信息:

- [Kotlin 多平台介绍](#)。
- [Kotlin 多平台所支持平台](#)。
- [Kotlin 多平台程序设计优势](#)。

1.5.30 的新特性

2. 加入 Kotlin 多平台社区：

-  Slack: [获取邀请](#)并加入 #getting-started 与 #multiplatform 频道。
-  StackOverflow: 订阅“kotlin-multiplatform”标签。

3. 关注 Kotlin: Twitter、 Reddit、 Youtube，不要错过任何重要的生态系统更新。

如果遇到任何困难和问题，请在我们的[问题跟踪系统](#)提报。

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](#)。

概述

- Kotlin 多平台
- Kotlin 用于服务器端开发
- Kotlin 用于 Android 开发
- Kotlin 用于 JavaScript 开发
- Kotlin 用于原生开发
- Kotlin 用于数据科学
- Kotlin 用于竞技程序设计

Kotlin 多平台

多平台项目处于 [Alpha](#) 版。语言特性与工具都可能在未来的 Kotlin 版本中发生变化。



支持多平台程序设计是 Kotlin 的主要优势之一。它减少了为[不同平台](#)编写与维护相同代码所花的时间，同时保留了原生编程的灵活性与优势。

Kotlin Multiplatform use cases

Android and iOS applications

Sharing code between mobile platforms is one of the major Kotlin Multiplatform use cases. With Kotlin Multiplatform Mobile, you can build cross-platform mobile applications and share common code between Android and iOS, such as business logic, connectivity, and more.

Check out the [Get started with Kotlin Multiplatform Mobile](#) section and [Kotlin Multiplatform Hands-on: Networking and Data Storage](#), where you will create an application for Android and iOS that includes a module with shared code for both platforms.

Full-stack web applications

Another scenario when code sharing may bring benefits is a connected application where the logic can be reused on both the server and the client side running in the browser. This is covered by Kotlin Multiplatform as well.

See [Build a Full Stack Web App with Kotlin Multiplatform](#) hands-on, where you will create a connected application consisting of a server part, using Kotlin/JVM and a web client, using Kotlin/JS.

Multiplatform libraries

1.5.30 的新特性

Kotlin Multiplatform is also useful for library authors. You can create a multiplatform library with common code and its platform-specific implementations for JVM, JS, and Native platforms. Once published, a multiplatform library can be used in other cross-platform projects as a dependency.

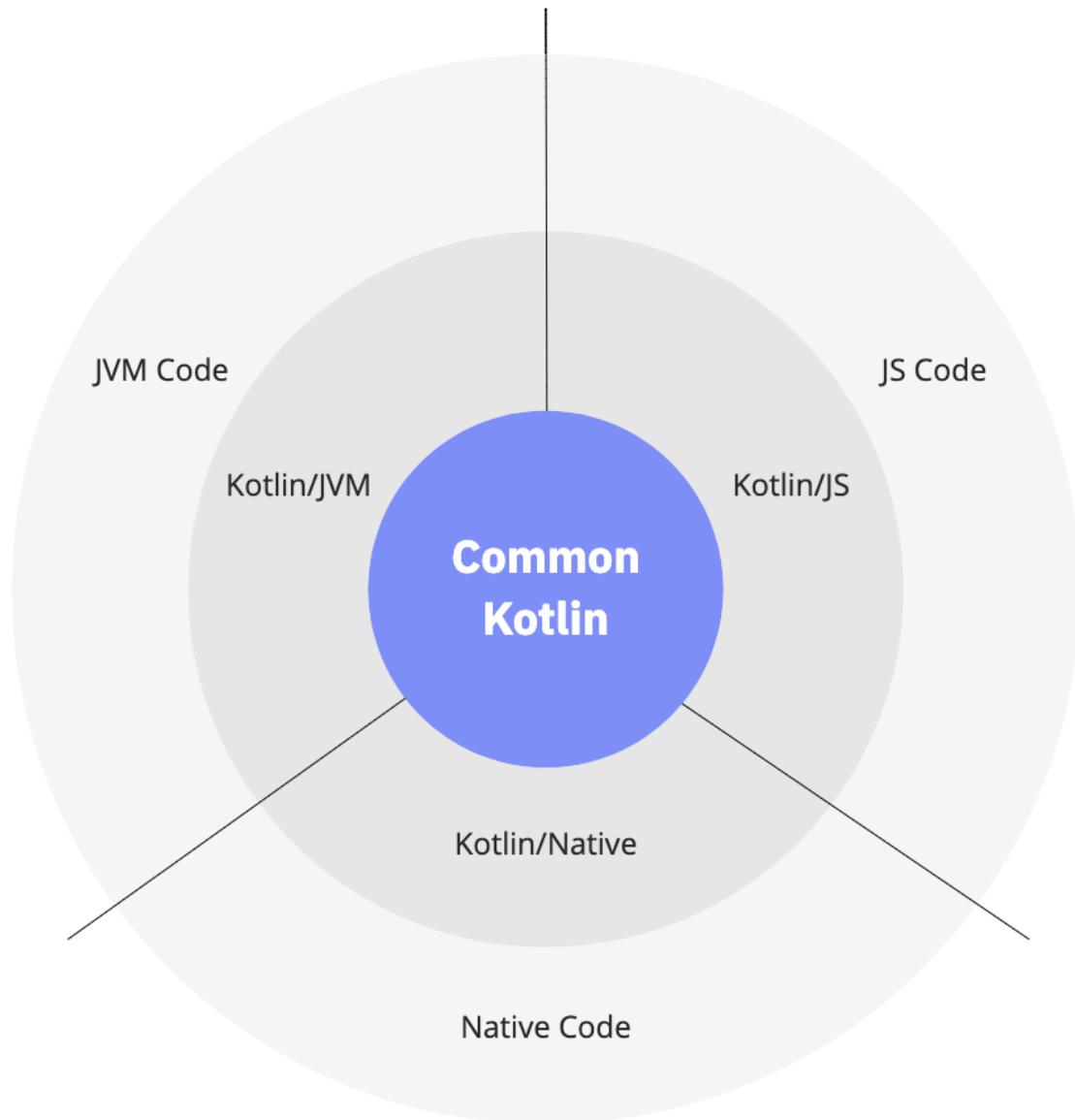
See the [Create and publish a multiplatform library](#) tutorial, where you will create a multiplatform library, test it, and publish it to Maven.

Common code for mobile and web applications

One more popular case for using Kotlin Multiplatform is sharing the same code across Android, iOS, and web apps. It reduces the amount of business logic coded by frontend developers and helps implement products more efficiently, decreasing the coding and testing efforts.

See the [RSS Reader](#) sample project — a cross-platform application for iOS and Android with desktop and web clients implemented as experimental features.

Kotlin 多平台的工作原理



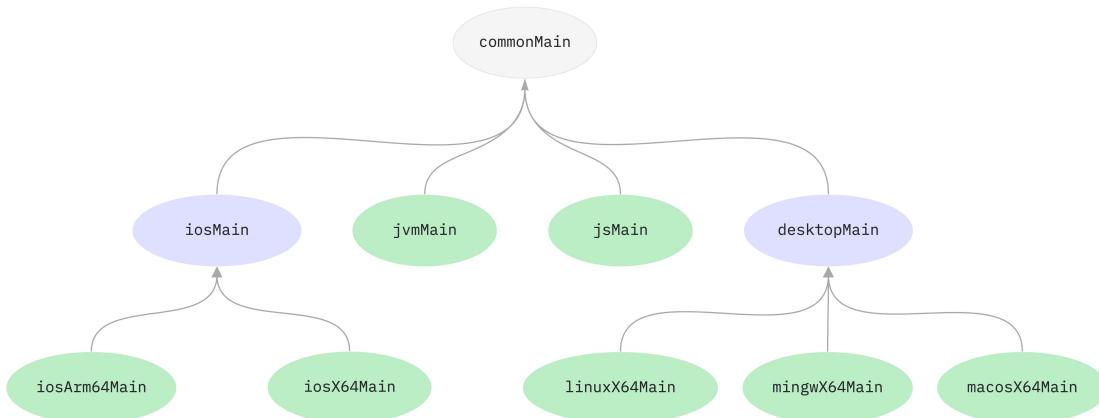
- 公共 **Kotlin** 包括语言、核心库与基本工具。用公共 **Kotlin** 编写的代码适用于所有平台的各个地方。
- 借助 **Kotlin** 多平台库，可以在公共代码以及平台相关代码中复用多平台逻辑。公共代码可以依赖一组涵盖日常任务的库，例如 [HTTP](#)、[序列化 \(serialization\)](#) 与 [管理协程 \(coroutines\)](#)。
- 如需与平台互操作，请使用平台相关的 **Kotlin** 版本。**平台相关的 Kotlin 版本** (**Kotlin/JVM**、**Kotlin/JS**、**Kotlin/Native**) 包含对 **Kotlin** 语言的扩展以及平台相关的库与工具。
- 通过这些平台可以访问平台原生代码 (**JVM**、**JS** 与 **Native**) 并利用所有原生功能。

Code sharing between platforms

1.5.30 的新特性

使用 Kotlin 多平台，花更少的时间为[不同平台](#)编写并维护相同的代码——只需使用 Kotlin 提供的机制进行共享即可：

- 在项目中用到的所有平台之间共享代码。用以共享适用于所有平台的公共业务逻辑。
- 在项目中包含的[某些平台（而不是所有平台）](#)间共享代码。当可以对类似的平台共享大量代码时请这么用。



- 如果需要从共享代码中访问平台相关的 API，请使用 Kotlin 的[预期声明与实际声明](#)机制。

Get started

- Look through [our examples and tutorials](#) if you want to create applications or libraries targeting JVM, JavaScript, and other platforms
- Start with the [Get started with Kotlin Multiplatform Mobile](#) if you want to create iOS and Android applications with shared code

New to Kotlin? Take a look at [Getting started with Kotlin](#).



样例项目

Look through cross-platform application samples to understand how Kotlin Multiplatform works:

- [Kotlin Multiplatform Mobile samples](#)
- [KotlinConf 应用](#)

1.5.30 的新特性

- [KotlinConf Spinner 应用](#)
- [Build a Full Stack Web App with Kotlin Multiplatform hands-on](#)

Kotlin 用于服务器端开发

Kotlin 非常适合开发服务器端应用程序。它可以帮助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）。
- **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。

使用 Kotlin 进行服务器端开发的框架

- [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。
- [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](#)。
- [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。
- [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。

1.5.30 的新特性

- [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。

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。

下一步

- 关于更深入的介绍，请查看本站的 Kotlin 文档及 [Kotlin 心印](#)。
- Watch a webinar "[Micronaut for microservices with Kotlin](#)" and explore a detailed [guide](#) showing how you can use [Kotlin extension functions](#) in the Micronaut framework.

1.5.30 的新特性

- 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。

使用 Kotlin 进行 Android 开发，可以受益于：

- **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。
- **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。
- **Android Jetpack 与其他库中的 Kotlin 支持。**[KTX 扩展](#)为现有的 Android 库添加了 Kotlin 语言特性，如协程、扩展函数、lambdas 与命名参数。
- **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。
- **支持多平台开发。**不仅可以使用 Kotlin 开发 Android，还可以开发 iOS、后端与 Web 应用程序。享受在平台之间共享公共代码的好处。
- **代码安全。**更少的代码与更好的可读性导致更少的错误。Kotlin 编译器检测这些剩余的错误，从而使代码安全。
- **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。根据 Google 的说法，Play 商店前 1000 个应用中有 60% 以上使用 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——详情请见[面向 Kotlin 开发者的谷歌网站](#)。

如果想开始使用 Kotlin 进行 Android 开发，请参阅[在 Android 开发中开始使用 Kotlin](#)。

如果是 Android 的新手，并且想学习使用 Kotlin 创建应用程序，请查看[这门 Udacity 课程](#)。

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.js` 与 `kotlin.multiplatform` Gradle 插件。它们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](#)文档。

Kotlin/JS 的使用场景

有很多使用 Kotlin/JS 的方式。这里列出了可以使用 Kotlin/JS 的场景的一个不完全的清单。

- **使用 Kotlin/JS 编写 Web 前端应用程序**
 - Kotlin/JS 允许以类型安全的方式 **利用功能强大的浏览器与 Web API**。创建、修改文档对象模型 (DOM) 中的元素并与之交互，使用 Kotlin 代码控制 `canvas` 或 WebGL 组件的呈现，并享受现代浏览器所支持的更多特性的访问。
 - 使用 JetBrains 提供的 `kotlin-wrappers` **用 Kotlin/JS 编写完整的，类型安全的 React 应用程序**，它为 React 及其他流行 JavaScript 框架提供方便的抽象与深度集成。`kotlin-wrappers` 还为许多类似技术（例如 `react-redux`、`react-router` 以及 `styled-components`）提供支持。与 JavaScript 生态系统的互操作性意味着可以使用第三方 React 组件与组件库。
 - 使用 **Kotlin/JS 框架**，充分利用 Kotlin 相关概念及其表现力与简洁性（例如 `kvision` 或 `fritz2`）。
- **使用 Kotlin/JS 编写服务器端与无服务器应用程序**
 - Kotlin/JS 提供的 Node.js 目标能够**创建在服务器上运行或在无服务器基础架构上执行的应用程序**。可以享受在 JavaScript 运行时中执行的所有优势，例如更快的启动与更少的内存占用。使用 `kotlinx-nodejs`，可以直接从 Kotlin 代码中对 `Node.js API` 进行类型安全的访问。
- **使用 Kotlin 的多平台项目与其他 Kotlin 目标共享代码**
 - 使用 `multiplatform` 多平台 Gradle 插件时，也可以访问所有 Kotlin/JS 功能。

1.5.30 的新特性

- 如果用 Kotlin 编写的后端，那么可以与用 Kotlin/JS 编写的前端共享公共代码，例如数据模型或逻辑验证，从而能够编写与维护全栈 Web 应用程序。
 - 还可以在 Web 界面与移动应用之间共享业务逻辑（Android 与 iOS），并避免重复实现常见的功能，例如围绕 REST API 端点提供抽象，用户身份验证或者领域模型。
- 创建用于 JavaScript 与 TypeScript 的库
 - 也不必用 Kotlin/JS 编写整个应用程序——而是可以从 Kotlin 代码生成库，这些库可以在 JavaScript 或 TypeScript 编写的任何代码库中作为模块使用，而与所使用的其他框架或技术无关。这种创建混合应用程序的方法可以利用个人与团队在 Web 开发方面已经具备的能力，同时减少重复的工作量、使 Web 目标与应用程序的其他目标平台保持一致变得更加容易。

当然，这并不是如何充分利用 Kotlin/JS 的完整列表，而只是一些精选的使用场景。请尝试不同的组合，并找出最适合项目的方案。

无论具体用例如何，Kotlin/JS 项目都可以使用兼容 **Kotlin 生态系统中的库**，以及第三方的 **JavaScript 与 TypeScript 生态系统中的库**。如果要在 Kotlin 代码中使用后者，可以提供自己的类型安全包装器、使用社区维护的包装器，也可以让 **Dukat** 自动生成 Kotlin 声明。使用 Kotlin/JS 专有的 **动态类型**可以放宽 Kotlin 的类型系统的约束，而跳过创建详细的库包装器，尽管这是以牺牲类型安全为代价。

Kotlin/JS 还与最常见的模块系统兼容：UMD、CommonJS 与 AMD。能够生产与使用 **模块** 意味着能够以结构化的方式与 JavaScript 生态系统进行交互。

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:

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**.

1.5.30 的新特性

Visit <https://kvision.io> 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 <https://www.fritz2.dev> 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 <https://nacular.github.io/doodle/> for documentation, tutorials, and examples.

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

Compose for Web

Compose for Web, a part of Compose Multiplatform, brings [Google's Jetpack Compose UI toolkit](#) to your browser. It allows you to build reactive web user interfaces using the concepts introduced by Jetpack Compose. It provides a DOM API to describe your website, as well as an experimental set of multiplatform layout primitives. Compose for Web also gives you the option to share parts of your UI code and logic across Android, desktop, and the web.

You can find more information about Compose Multiplatform on its [landing page](#).

1.5.30 的新特性

Join the [#compose-web](#) channel on the [Kotlin Slack](#) to discuss Compose for Web, or [#compose](#) for general Compose Multiplatform discussions.

Kotlin/JS 今天与明天

在这个视频中 ([YouTube](#)、[bilibili](#))，Kotlin 开发者布道师 Sebastian Aigner 解释了 Kotlin/JS 的主要优点、分享一些技巧与使用场景，并探讨 Kotlin/JS 的计划与即将发布的特性。

Kotlin/JS 入门

如果不熟悉 Kotlin，那么第一步最好是熟悉该语言的[基本语法](#)。

如需开始将 Kotlin 用于 JavaScript，请参考[搭建 Kotlin/JS 项目](#)。 You can also pick a [hands-on](#) lab to work through or check out the list of [Kotlin/JS sample projects](#) for inspiration. They contain useful snippets and patterns and can serve as nice jump-off points for your own projects.

Kotlin/JS 动手实践实验室

- [Building Web Applications with React and Kotlin/JS](#) guides you through the process of building a simple web application using the React framework, shows how a type-safe Kotlin DSL for HTML makes it easy to build reactive DOM elements, and illustrates how to use third-party React components and obtain information from APIs, all while writing the whole application logic in pure Kotlin/JS.
- [Building a Full Stack Web App with Kotlin Multiplatform](#) teaches the concepts behind building an application that targets Kotlin/JVM and Kotlin/JS by building a client-server application that makes use of shared code, serialization, and other multiplatform paradigms. It also provides a brief introduction to working with Ktor both as a server- and client-side framework.

Kotlin/JS 样例项目

1.5.30 的新特性

- [Full-stack Spring collaborative to-do list](#) shows how to create a to-do list for collaborative work using `kotlin-multiplatform` with JS and JVM targets, Spring for the backend, Kotlin/JS with React for the frontend, and RSocket.
- [Kotlin/JS and React Redux to-do list](#) implements the React Redux to-do list using JS libraries (`react`, `react-dom`, `react-router`, `redux`, and `react-redux`) from npm and Webpack to bundle, minify, and run the project.
- [Full-stack demo application](#) guides you through the process of building an app with a feed containing user-generated posts and comments. All data is stubbed by the fakeJSON and JSON Placeholder services.

新的 Kotlin/JS IR 编译器

新的 [Kotlin/JS IR 编译器](#) (当前稳定性: [Beta](#)) 相对于当前的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。通过从 Kotlin 代码生成 TypeScript 声明文件 (`.d.ts`)，新的编译器使创建混合 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。

如需了解有关新 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，请访问其 [Kotlin/JS IR compiler documentation page](#) and the [migration guide](#).

加入 Kotlin/JS 社区

还可以在官方 [Kotlin Slack](#) 中加入 `#javascript` 频道，同社区与团队交谈。

Kotlin 原生

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。它非常适合开发人员需要生成无需额外运行时或虚拟机的自包含程序的情况。

目标平台

Kotlin/Native 支持以下平台：

- macOS
 - iOS、tvOS、watchOS
- Linux
 - Windows (MinGW)
- Android NDK

[可在这里获取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。编译器可创建：

- 用于多个[平台](#)的可执行文件
- 用于 C/C++ 项目的静态库或[动态](#)库以及 C 语言头文件
- 用于 Swift 与 Objective-C 项目的 [Apple 框架](#)

支持直接在 Kotlin/Native 中使用以下现有库的互操作：

- 静态或动态 [C 语言库](#)
- C 语言、[Swift](#) 以及 [Objective-C](#) 框架

1.5.30 的新特性

将编译后的 Kotlin 代码包含进用 C、C++、Swift、Objective-C 以及其他语言编写的现有项目中会很容易。直接在 Kotlin/Native 中使用现有原生代码、静态或动态 [C 语言库](#)、Swift/Objective-C [框架](#)、图形引擎以及任何其他原生内容也很容易。

Kotlin/Native [库](#)有助于在多个项目之间共享 Kotlin 代码。POSIX、gzip、OpenGL、Metal、Foundation 以及许多其他流行库与 Apple 框架都已预先导入并作为 Kotlin/Native 库包含在编译器包中。

在多个平台之间共享代码

[多平台项目](#)允许在多个平台之间共享公共的 Kotlin 代码，包括：Android、iOS、JVM、JavaScript 与原生。多平台库为公共 Kotlin 代码提供了所需的 API，并且有助于在一处用 Kotlin 开发项目的共享部分，并将其与一些或所有目标平台共享。

可以使用 [Kotlin 移动端多平台](#)通过 Android 与 iOS 之间共享代码创建多平台移动应用程序。

如何入门

教程与文档

刚接触 Kotlin？可以看看 [Kotlin 入门](#)页。

推荐文档：

- [Kotlin 移动端多平台文档](#)
- [多平台文档](#)
- [C 语言互操作](#)
- [Swift/Objective-C 互操作](#)

推荐教程：

- [Kotlin/Native 入门](#)
- [创建第一个跨平台移动端应用程序](#)
- [C 语言 Kotlin/Native 之间的类型映射](#)
- [Kotlin/Native 开发动态库](#)
- [Kotlin/Native 开发 Apple 框架](#)

样例项目

1.5.30 的新特性

- [Kotlin 移动端多平台示例](#)
- [Kotlin/Native 源代码与示例](#)
- [KotlinConf 应用](#)
- [KotlinConf Spinner 应用](#)
- [Kotlin/Native 源代码与示例 \(.tgz\)](#)
- [Kotlin/Native 源代码与示例 \(.zip\)](#)

Kotlin 用于数据科学

从构建数据流水线到生产机器学习模型， Kotlin 可能是处理数据的绝佳选择：

- Kotlin 简洁、易读且易于学习。
- 静态类型与空安全有助于创建可靠的、可维护的、易于故障排除的代码。
- 作为一种 JVM 语言，Kotlin 提供了出色的性能表现，并具有充分利用久经考验的 Java 库的整个生态系统的功能。

交互式编辑器

[Jupyter Notebook](#) 与 [Apache Zeppelin](#) 等笔记本为数据可视化与探索性研究提供了方便的工具。Kotlin 与这些工具集成在一起，可以帮助探索数据、与同事共享发现或建立数据科学和机器学习技能。

Jupyter Kotlin 内核

Jupyter Notebook 是一个开源 Web 应用程序，它允许创建与共享包含代码、可视化与 Markdown 文本的文档（也称为“笔记本”）。[Kotlin-jupyter](#) 是一个开源项目，它为 Jupyter Notebook 带来了 Kotlin 支持。

1.5.30 的新特性

jupyter Kotlin Kernel Examples Last Checkpoint: Last Thursday at 15:16 (autosaved) Logout

File Edit View Insert Cell Kernel Widgets Help Trusted | Kotlin

In [1]: %use lets-plot, krangl

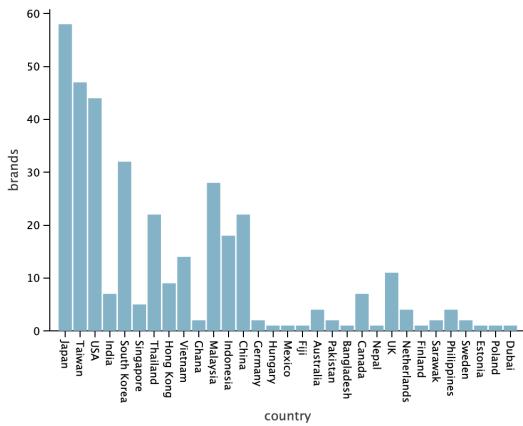
In [2]: val df = DataFrame.readCSV("ramen-ratings.csv")
val processedDF = df.filter{ it["Stars"].isMatching <String>{ !startsWith("Un") } }
.addColumn("StarsAsDouble") { it["Stars"].map <String> { it.toDouble() } }

In [3]: val distinctBrandsPerCountry = processedDF.groupBy("Country").distinct("Brand").groupBy("Country").count()
val (xs, ys) = distinctBrandsPerCountry.rows.map { row -> (row["Country"] as String) to (row["n"] as Int) }.unzip()
val p = lets_plot(mapof("country" to xs, "brands" to ys))

val layer = geom_bar (stat=Stat.identity, fill = "#78B3CA") {
 x = "country"
 y = "brands"
}

p + layer

Out[3]:



Country	Brands
Japan	58
Taiwan	47
USA	43
India	7
South Korea	32
Singapore	5
Thailand	22
Hong Kong	10
Vietnam	14
China	22
Indonesia	18
Malaysia	28
Germany	2
Hungary	1
Fiji	1
Mexico	1
Australia	4
Pakistan	1
Bangladesh	1
Canada	7
Nepal	3
UK	11
Netherlands	4
Finland	1
Sarawak	2
Philippines	3
Sweden	2
Estonia	1
Poland	1
Dubai	1

查看 Kotlin 内核的 [GitHub 仓库](#) 以获取安装说明、文档与示例。

Zeppelin Kotlin 解释器

Apache Zeppelin 是一种流行的基于 Web 的交互式数据分析解决方案。它为 Apache Spark 集群计算系统提供了强大的支持，这对数据工程特别有用。从[版本 0.9.0](#) 开始，Apache Zeppelin 内置了 Kotlin 解释器。

Zeppelin Notebook Job Search anonymous

Kotlin Interpreter Head default

```
fun square(n: Int): Int = n * n  
  
val greeter = { s: String -> println("Hello $s!") }  
val l = listOf("Drive", "to", "develop")  
  
kc.showVars()  
kc.showFunctions()  
  
kc: KotlinContext! = org.apache.zeppelin.kotlin.repl.KotlinRepl$KotlinContext@304ec43a  
x: Int = 1
```

FINISHED

类库

1.5.30 的新特性

Kotlin 社区创建的用于数据相关任务的类库生态系统正在迅速扩展。以下是一些可能会有用的库：

Kotlin 库

- [Multik](#): multidimensional arrays in Kotlin. The library provides Kotlin-idiomatic, type- and dimension-safe API for mathematical operations over multidimensional arrays. Multik offers swappable JVM and native computational engines, and a combination of the two for optimal performance.
- [KotlinDL](#) is a high-level Deep Learning API written in Kotlin and inspired by Keras. It offers simple APIs for training deep learning models from scratch, importing existing Keras models for inference, and leveraging transfer learning for tweaking existing pre-trained models to your tasks.
- [Kotlin for Apache Spark](#) adds a missing layer of compatibility between Kotlin and Apache Spark. It allows Kotlin developers to use familiar language features such as data classes, and lambda expressions as simple expressions in curly braces or method references.
- [kotlin-statistics](#) 是一个为探索性统计与生产统计中提供扩展函数的库。它支持基本的数字列表/序列/数组函数（从 `sum` 到 `skewness`）、切片操作符（诸如 `countBy`、`simpleRegressionBy`）、分箱（binning）操作符、离散 PDF 采样、朴素贝叶斯分类器、聚类、线性回归等等。
- [kmath](#) 是一个受 [NumPy](#) 启发的库。这个库支持代数结构与运算、类数组结构、数学表达式、直方图、流运算、[commons-math](#) 与 [koma](#) 的包装等等。
- [krangl](#) 是一个受 R 语言的 [dplyr](#) 与 Python 的 [pandas](#) 启发的库。这个库提供了采用函数式风格 API 进行数据操作的功能；它还包括过滤、转换、聚合与重塑表格数据的函数。
- [lets-plot](#) 是一个用 Kotlin 编写的统计数据绘图库。Lets-Plot 是多平台的，不仅可以用于 JVM，还可以用于 JS 与 Python。
- [kravis](#) 是另一个用于表格数据可视化的库，其灵感来自于 R 的 [ggplot](#)。
- [londogard-nlp-toolkit](#) is a library that provides utilities when working with natural language processing such as word/subword/sentence embeddings, word-frequencies, stopwords, stemming, and much more.

Java 库

1.5.30 的新特性

因为 Kotlin 提供了与 Java 互操作的头等支持，所以也可以在用于数据科学的 Kotlin 代码中使用 Java 库。以下是这些库的一些示例：

- [DeepLearning4J](#)——一个 Java 深度学习库
- [ND4J](#)——用于 JVM 的高效矩阵数学库
- [Dex](#)——一个基于 Java 的数据可视化工具
- [Smile](#)——一个全面的机器学习、自然语言处理、线性代数、图、插值与可视化系统。除了 Java API，Smile 还提供了函数式的 [Kotlin API](#) 以及 Scala 与 Clojure API。
 - [Smile-NLP-kt](#)——以 Kotlin 扩展函数与接口格式重写了 Smile 的自然语言处理部分的 Scala 隐式内容。
- [Apache Commons Math](#)——一个 Java 通用数学、统计与机器学习库
- [NM Dev](#) - a Java mathematical library that covers all of classical mathematics.
- [OptaPlanner](#)——一个用于优化规划问题的求解器实用程序
- [Charts](#)——一个正在开发中的科学 JavaFX 图表库
- [CoreNLP](#)——一个自然语言处理工具包
- [Apache Mahout](#)——一个回归、聚类与推荐的分布式框架
- [Weka](#)——一组用于数据挖掘任务的机器学习算法
- [Tablesaw](#) - a Java dataframe. It includes a visualization library based on Plot.ly

如果这个列表还不能满足需求，可以在 Thomas Nield 的 [Kotlin Machine Learning Demos](#) GitHub repository with showcases 中找到更多选项。

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[Get started with Kotlin/JVM](#)。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)
```

1.5.30 的新特性

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`）等。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：

```
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
}
```

The `readln()` function is available since [Kotlin 1.6.0](#).



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。

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：

1.5.30 的新特性

```
private fun readInt() = readln().toInt()
private fun readStr() = readln().toString()
// 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：

```
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)
}
```

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：

1.5.30 的新特性

```
private fun readInt() = readln().toInt() // 单个整数
private fun readStrings() = readln().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 = readln()
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：

1.5.30 的新特性

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

学习 Kotlin

Kotlin is easy to learn, especially for those who already know Java. 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

- 1.6.0 的新特性
- 早期版本
 - 1.5.30 的新特性
 - 1.5.20 的新特性
 - 1.5.0 的新特性
 - 1.4.30 的新特性
 - 1.4.20 的新特性
 - 1.4.0 的新特性
 - 1.3 的新特性
 - 1.2 的新特性
 - 1.1 的新特性

Kotlin 1.6.0 的新特性

发布日期: 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.

1.5.30 的新特性

```
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())`.

1.5.30 的新特性

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.

1.5.30 的新特性

```
// Before 1.5.30
val containerA = PostgreSQLContainer<Nothing>(DockerImageName.parse("postgres:13-alpine"))
    .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:

1.5.30 的新特性

```
@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

1.5.30 的新特性

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  
}
```

1.5.30 的新特性

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

1.5.30 的新特性

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

1.5.30 的新特性

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`:

```
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.

1.5.30 的新特性

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 publ:  
<stack trace>
```

- 1.6.0:

```
e: The symbol of unexpected type encountered during IR deserialization: IrClass  
IrTypeAliasSymbol is expected.
```

This could happen if there are two libraries, where one library was compiled
Please check that the project configuration is correct and has consistent ver-

The list of libraries that depend on "org.jetbrains.kotlinx:kotlinx-coroutines-core:
<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](#).

1.5.30 的新特性

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.YarnPl  
rootProject.the<org.jetbrains.kotlin.gradle.targets.js.yarn.YarnRootExtensi  
}
```

【Groovy】

```
rootProject.plugins.withType(org.jetbrains.kotlin.gradle.targets.js.yarn.YarnPl  
rootProject.extensions.getByName<org.jetbrains.kotlin.gradle.targets.js.yar  
}
```

1.5.30 的新特性

- Node.js:

【Kotlin】

```
rootProject.plugins.withType<org.jetbrains.kotlin.gradle.targets.js.nodejs.NodeJsRootExt>
    rootProject.the<org.jetbrains.kotlin.gradle.targets.js.nodejs.NodeJsRootExt>
}
```

【Groovy】

```
rootProject.plugins.withType(org.jetbrains.kotlin.gradle.targets.js.nodejs.NodeJsRootExt)
    rootProject.extensions.getByName<org.jetbrains.kotlin.gradle.targets.js.nodejs.NodeJsRootExt>
}
```

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

1.5.30 的新特性

- 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
<code>readLine()!!</code>	<code>readln()</code>	Reads a line from <code>stdin</code> and returns it, or throws a <code>RuntimeException</code> if EOF has been reached.
<code>readLine()</code>	<code>readlnOrNull()</code>	Reads a line from <code>stdin</code> and returns it, or returns <code>null</code> 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
}
```

1.5.30 的新特性

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
        // 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

1.5.30 的新特性

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/std`

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
}
```

1.5.30 的新特性

- `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](#).



1.5.30 的新特性

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](#).

早期版本

- 1.5.30 的新特性
- 1.5.20 的新特性
- 1.5.0 的新特性
- 1.4.30 的新特性
- 1.4.20 的新特性
- 1.4.0 的新特性
- 1.3 的新特性
- 1.2 的新特性
- 1.1 的新特性

Kotlin 1.5.30 的新特性

发布日期: 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

1.5.30 的新特性

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 all possible types or values of its subject or for some types plus an `else` branch. In other words, it covers all possible 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】

1.5.30 的新特性

```
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】

1.5.30 的新特性

```
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

1.5.30 的新特性

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 dataSource = 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.

1.5.30 的新特性

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-alpine"))
    .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.

1.5.30 的新特性

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

注解类的实例化是 [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](#).



1.5.30 的新特性

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 KEP](#)

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

1.5.30 的新特性

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](#). Learn how to [use Apple silicon targets with the 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

1.5.30 的新特性

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'  
        basePath = "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  
}
```

1.5.30 的新特性

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:

1.5.30 的新特性

```
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】

1.5.30 的新特性

```
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】

1.5.30 的新特性

```
import org.jetbrains.kotlin.gradle.plugin.mpp.apple.XCFrameworkConfig

plugins {
    id 'org.jetbrains.kotlin.multiplatform'
}

kotlin {
    def xcf = new XCFrameworkConfig(project)

    ios {
        binaries.framework {
            basePath = "shared"
            xcf.add(it)
        }
    }
    watchos {
        binaries.framework {
            basePath = "shared"
            xcf.add(it)
        }
    }
    tvos {
        binaries.framework {
            basePath = "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:

1.5.30 的新特性

- 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

1.5.30 的新特性

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】

1.5.30 的新特性

```
kotlin {  
    jvmToolchain {  
        (this as JavaToolchainSpec).languageVersion.set(JavaLanguageVersion.of(<MAJ  
    }  
}
```

【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】

1.5.30 的新特性

```
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:

1.5.30 的新特性

```
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()  
}
```

【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  
>  

```

1.5.30 的新特性

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
<code>Duration.days(45).toString()</code>	<code>45.0d</code>	<code>45d</code>
<code>Duration.days(1.5).toString()</code>	<code>36.0h</code>	<code>1d 12h</code>
<code>Duration.minutes(1230).toString()</code>	<code>20.5h</code>	<code>20h 30m</code>
<code>Duration.minutes(2415).toString()</code>	<code>40.3h</code>	<code>1d 16h 15m</code>
<code>Duration.minutes(920).toString()</code>	<code>920m</code>	<code>15h 20m</code>
<code>Duration.seconds(1.546).toString()</code>	<code>1.55s</code>	<code>1.546s</code>
<code>Duration.milliseconds(25.12).toString()</code>	<code>25.1ms</code>	<code>25.12ms</code>

1.5.30 的新特性

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:

1.5.30 的新特性

```
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](#).



1.5.30 的新特性

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.

1.5.30 的新特性

```
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

发布日期: 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.

1.5.30 的新特性

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](#).



1.5.30 的新特性

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.copyOf()` 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.

1.5.30 的新特性

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'd be very grateful if you would share your feedback with us using this [YouTrack ticket](#).

编译器 bug 修复

1.5.30 的新特性

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.copyOf()` 的性能

We've improved the way `Array.copyOf()` 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

1.5.30 的新特性

在 kapt 中缓存注解处理器的类加载器是 [Experimental](#)。它可能在任何时候被移除或更改。仅在评估目的时使用它。我们希望您在 [YouTrack](#) 上提供反馈。



现在有一个新的实验性功能，使缓存注解处理器的类加载器成为可能。该功能可以提高 kapt 在连续 Gradle 运行中的速度。

要启用此功能，请在您的 `gradle.properties` 文件中使用以下属性：

```
# 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
```

了解更多关于 [kapt](#)。

弃用 `kotlin.parallel.tasks.in.project` 构建属性

随着此发布，Kotlin 并行编译由 [Gradle 并行执行标志](#) `--parallel` 控制。使用此标志，Gradle 并行执行任务，从而提高编译任务的速度并更有效地利用资源。

您不再需要使用 `kotlin.parallel.tasks.in.project` 属性。此属性已弃用并在下一个主要发布版本中将被移除。

标准库

Kotlin 1.5.20 改变了几个与字符相关的函数的平台特定实现，从而实现了跨平台统一。

- [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

1.5.30 的新特性

`Char.toInt()` 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.toInt()` on any Unicode digit character and get its numeric representation.

```
fun main() {
    //sampleStart
    val ten = '\u0661'.toInt() + '\u0039'.toInt() // ARABIC-INDIC DIGIT 0
    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

发布日期: 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.](#)

1.5.30 的新特性

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.

1.5.30 的新特性

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'd 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:

1.5.30 的新特性

- 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

1.5.30 的新特性

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:

1.5.30 的新特性

- For `String` functions:

早期版本	1.5.0 的另一选择
<code>String.toUpperCase()</code>	<code>String.uppercase()</code>
<code>String.toLowerCase()</code>	<code>String.lowercase()</code>
<code>String.capitalize()</code>	<code>String.replaceFirstChar { it.uppercase() }</code>
<code>String.decapitalize()</code>	<code>String.replaceFirstChar { it.lowercase() }</code>

- For `Char` functions:

早期版本	1.5.0 的另一选择
<code>Char.toUpperCase()</code>	<code>Char.uppercaseChar(): Char</code> <code>Char.uppercase(): String</code>
<code>Char.toLowerCase()</code>	<code>Char.lowercaseChar(): Char</code> <code>Char.lowercase(): String</code>
<code>Char.toTitleCase()</code>	<code>Char.titlecaseChar(): Char</code> <code>Char.titlecase(): String</code>

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:

1.5.30 的新特性

```
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 KEP](#).

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.

1.5.30 的新特性

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.

1.5.30 的新特性

```
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}")
//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.isLetterOrDigit() }
    println(letterOrDigitList) // [a, 1]
    println(notLetterOrDigitList) // [+]
//sampleEnd
}
```

Functions for checking the case of a char:

- `Char.isLowerCase()`
- `Char.isUpperCase()`
- `Char.isTitleCase()`

1.5.30 的新特性

```
fun main() {
//sampleStart
    val chars = listOf('Dż', 'Lj', 'Nj', 'Dz', '1', 'A', 'a', '+')
    val (titleCases, notTitleCases) = chars.partition { it.isTitleCase() }
    println(titleCases) // [Dż, 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`.

1.5.30 的新特性

- `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

1.5.30 的新特性

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()  
    }  
}
```

1.5.30 的新特性

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
    // 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 }
    assertEquals(expectedArray, actualArray)
}
```

- `Double` 和 `Float` 数值的 `assertEquals()` 与 `assertNotEquals()` 的新的重载

1.5.30 的新特性

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

1.5.30 的新特性

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

1.5.30 的新特性

Learn more in the [changelog](#) and the [kotlinx-datetime 0.2.0 release blog post](#).

迁移到 Kotlin 1.5.0

IntelliJ IDEA 和 Android Studio 将会建议更新 Kotlin 插件至 1.5.0 版本，一旦版本发布后。

要将现有项目迁移到 Kotlin 1.5.0，只需将 Kotlin 版本更改为 [1.5.0](#) 并重新导入 Gradle 或 Maven 项目。[Learn how to update to Kotlin 1.5.0.](#)

要开始一个新项目，使用 Kotlin 1.5.0，更新 Kotlin 插件并从 **File | New | Project** 运行 Project Wizard。

新的命令行编译器现在可以在[GitHub release page](#)上下载。

Kotlin 1.5.0 是一个 [feature release](#)，因此可能带来不兼容的更改。在[Compatibility Guide for Kotlin 1.5.](#) 中找到此类更改的详细列表。

Kotlin 1.4.30 的新特性

发布日期: 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'd 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.

1.5.30 的新特性

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 记录类型支持 and we'd 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 -> // ...  
}
```

1.5.30 的新特性

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'd 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'd 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:

1.5.30 的新特性

```
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:

1.5.30 的新特性

- 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

1.5.30 的新特性

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是 [Experimental](#)。它可能在任何时候被移除或更改。启用此功能（参见下方的详细信息），并仅在评估目的时使用。我们希望您在 [YouTrack](#) 上提供反馈。



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
<code>String.toUpperCase()</code>	<code>String.uppercase()</code>
<code>String.toLowerCase()</code>	<code>String.lowercase()</code>
<code>String.capitalize()</code>	<code>String.replaceFirstChar { it.uppercase() }</code>
<code>String.decapitalize()</code>	<code>String.replaceFirstChar { it.lowercase() }</code>

- For `Char` functions:

Earlier versions	1.4.30 alternative
<code>Char.toUpperCase()</code>	<code>Char.uppercaseChar(): Char</code> <code>Char.uppercase(): String</code>
<code>Char.toLowerCase()</code>	<code>Char.lowercaseChar(): Char</code> <code>Char.lowercase(): String</code>
<code>Char.toTitleCase()</code>	<code>Char.titlecaseChar(): Char</code> <code>Char.titlecase(): String</code>

For Kotlin/JVM, there are also overloaded `uppercase()`, `lowercase()`, and `titlecase()` functions with an explicit `Locale` parameter.



1.5.30 的新特性

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
```

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

发布日期: 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](#)).

1.5.30 的新特性

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()  
}
```

1.5.30 的新特性

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")  
    }  
}
```

1.5.30 的新特性

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.

Learn how to [create Kotlin/JS applications from templates](#).

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 your 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 [ignoring compilation errors](#) with 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](#).



1.5.30 的新特性

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 `NSError`'s into Kotlin exceptions of type `ForeignException`. They hold the references to the original `NSError`'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")  
}
```

1.5.30 的新特性

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 capabilities.

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.

1.5.30 的新特性

- 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](#)

1.5.30 的新特性

- 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

1.5.30 的新特性

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

发布日期: 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)}")  
}
```

1.5.30 的新特性

[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  
}
```

1.5.30 的新特性

【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 KEP.](#)

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.

1.5.30 的新特性

```
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 = ' ', //trailing comma  
) {  
    // ...  
}
```

```
val colors = listOf(  
    "red",  
    "green",  
    "blue", //trailing comma  
)
```

可调用引用改进

Kotlin 1.4 supports more cases for using callable references:

- 对具有默认参数值的函数的引用
- 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

1.5.30 的新特性

- 可调用引用的挂起转换

对具有默认参数值的函数的引用

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.

1.5.30 的新特性

```
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)
        }
    }
}
```

1.5.30 的新特性

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

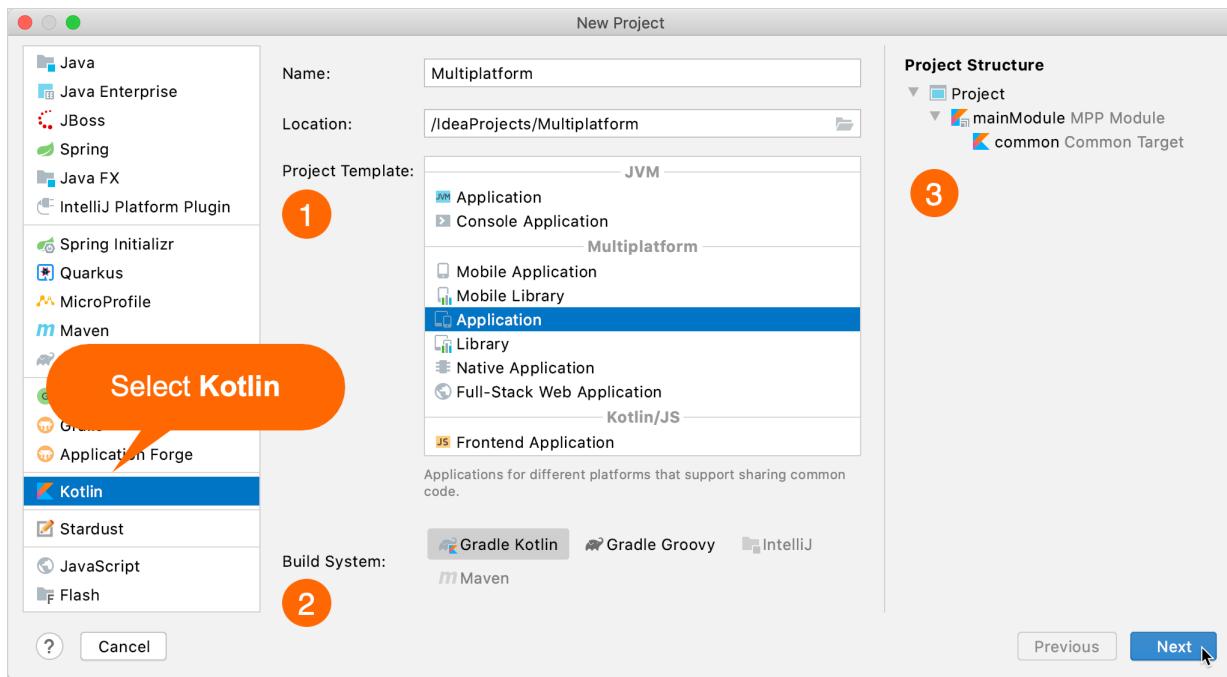
With Kotlin 1.4, you can use the new tools in IntelliJ IDEA to simplify Kotlin development:

- 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.

1.5.30 的新特性



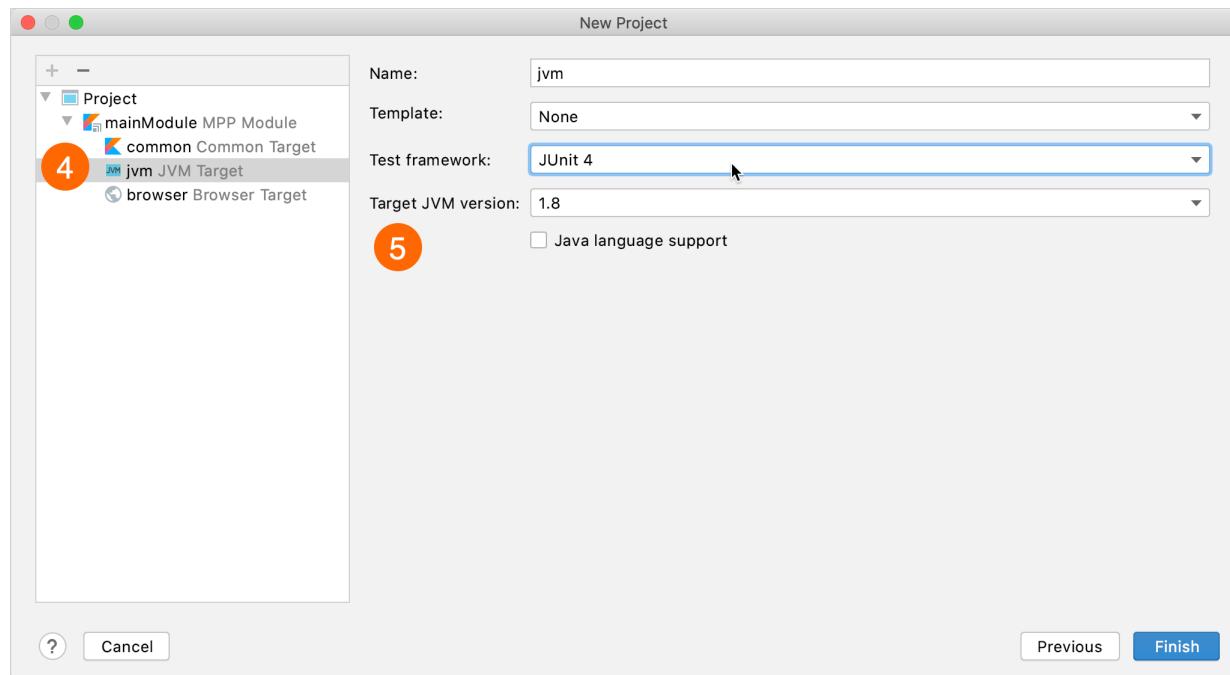
The new Kotlin Project Wizard is both simple and flexible:

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.

1.5.30 的新特性



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.

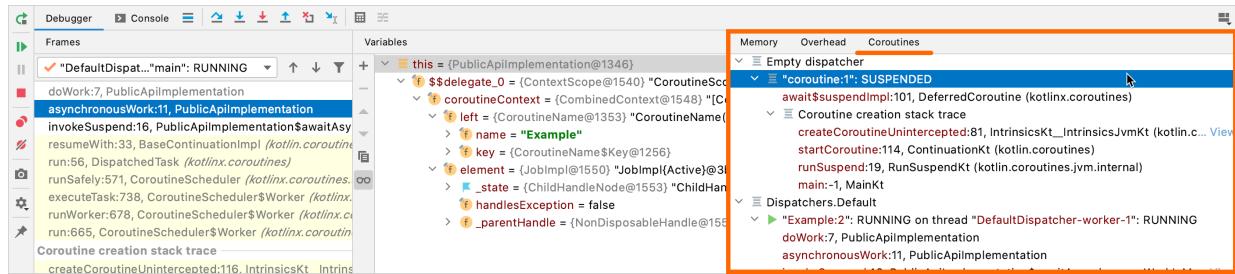
In Kotlin 1.4, debugging coroutines is now much more convenient with the new functionality shipped with the Kotlin plugin.

Debugging works for versions 1.3.8 or later of [kotlinx-coroutines-core](#).



1.5.30 的新特性

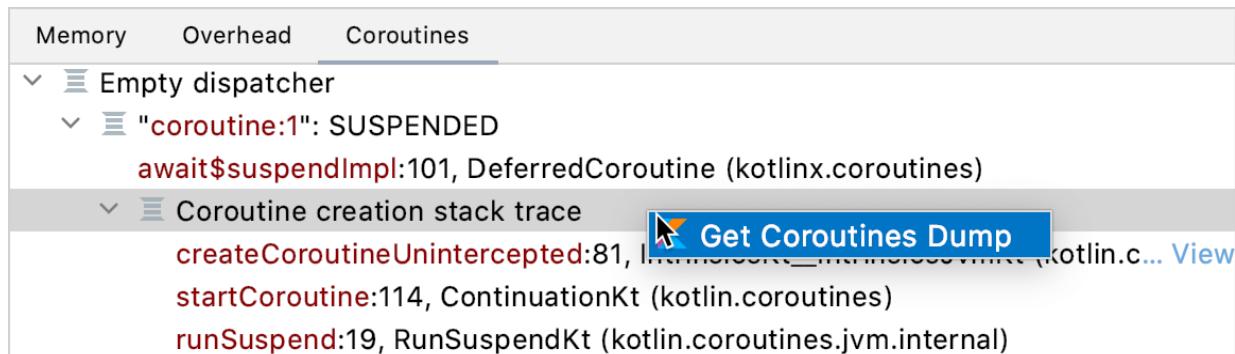
The **Debug Tool Window** now contains a new **Coroutines** tab. In this tab, you can find information about both currently running and suspended coroutines. The coroutines are grouped by the dispatcher they are running on.



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:

1.5.30 的新特性

- 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.isNullOrEmpty() },
    "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

1.5.30 的新特性

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:

1.5.30 的新特性

```
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?`:

1.5.30 的新特性

```
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.

1.5.30 的新特性

```
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

1.5.30 的新特性

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.

1.5.30 的新特性

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-null 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.

1.5.30 的新特性

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](#).

1.5.30 的新特性

- 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. [Read more about how to use the integration here.](#)

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

1.5.30 的新特性

- 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.

1.5.30 的新特性

```
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:

1.5.30 的新特性

```
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.

1.5.30 的新特性

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

1.5.30 的新特性

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](#).

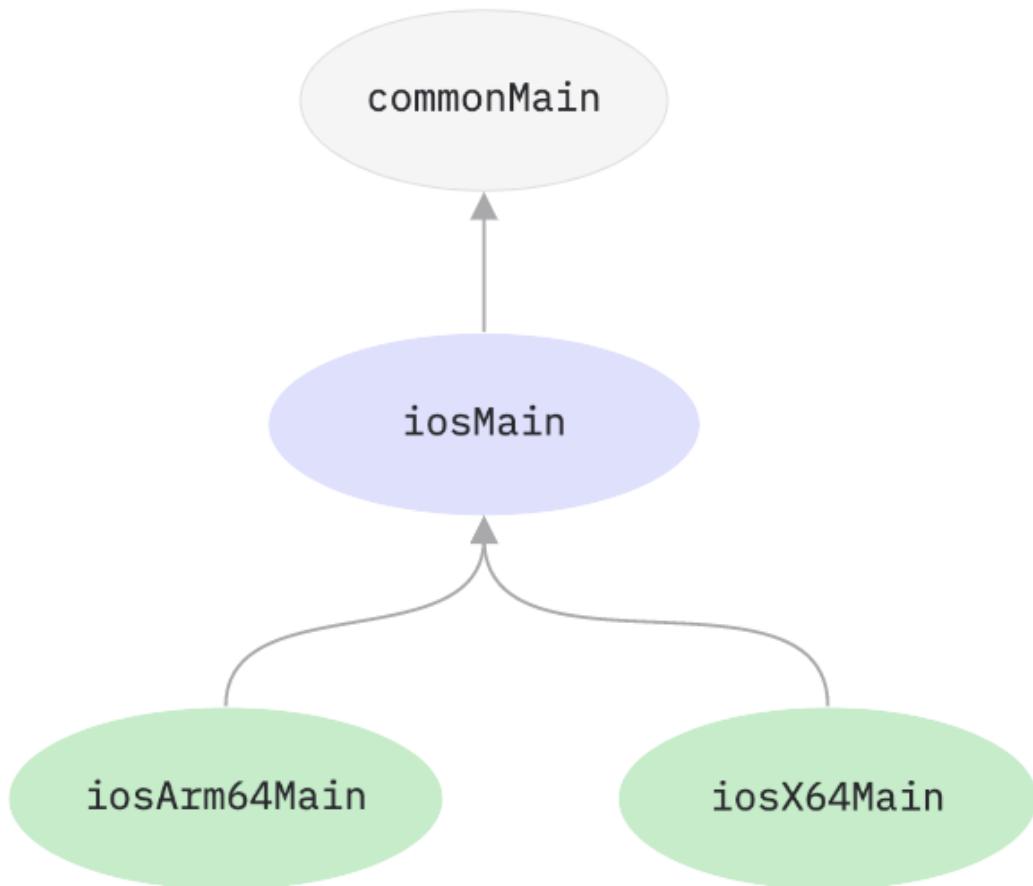
Previously, any code added to a multiplatform project could be placed either in a platform-specific source set, which is limited to one target and can't be reused by any other platform, or in a common source set, like `commonMain` or `commonTest`, which is shared across all the platforms in the project. 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.

1.5.30 的新特性



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.

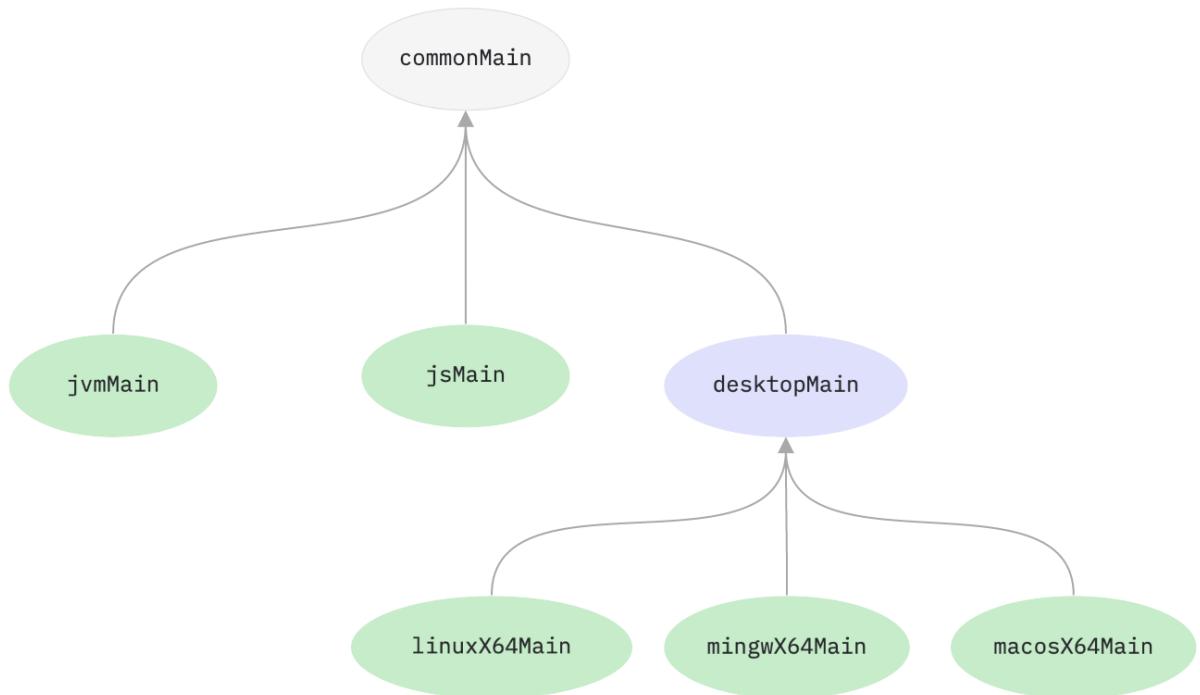
For common combinations of targets, you can create a hierarchical structure with [target shortcuts](#).

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, by connecting the source sets with the `dependsOn` relation.

1.5.30 的新特性



【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】

1.5.30 的新特性

```
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】

1.5.30 的新特性

```
kotlin {  
    sourceSets {  
        val commonMain by getting {  
            dependencies {  
                implementation("org.jetbrains.kotlinx:kotlinx-coroutines-core:1.6.0")  
            }  
        }  
    }  
}
```

【Groovy】

```
kotlin {  
    sourceSets {  
        commonMain {  
            dependencies {  
                implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-core:1.6.0'  
            }  
        }  
    }  
}
```

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

1.5.30 的新特性

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.

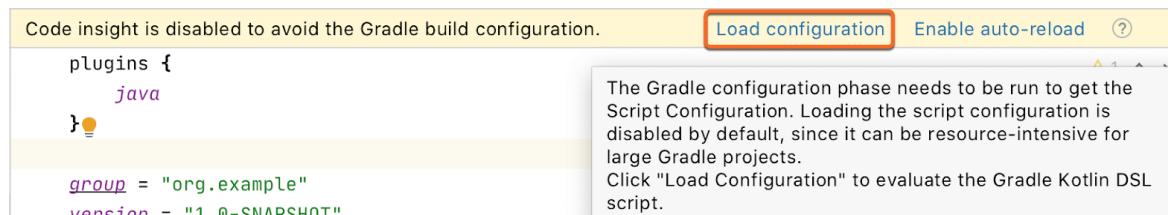
改进了 IDE 对 `*.gradle.kts` 的支持

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.

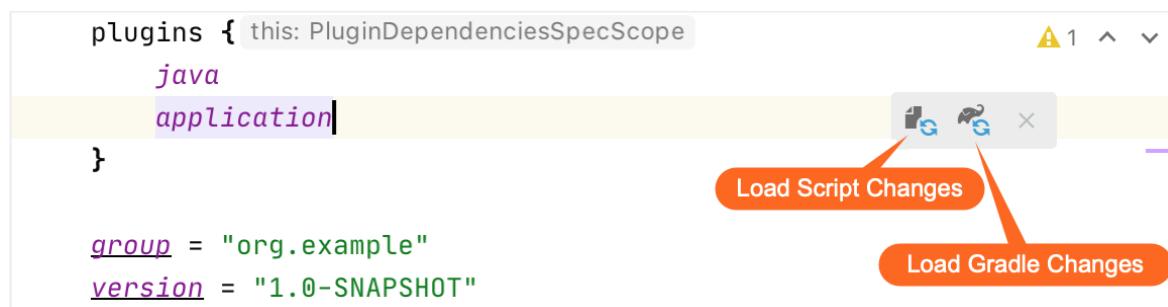
In Gradle versions earlier than 6.0, you need to manually load the script configuration by clicking **Load Configuration** in the editor.

1.5.30 的新特性



In Gradle 6.0 and above, you can explicitly apply changes by clicking **Load Gradle Changes** or by reimporting the Gradle project.

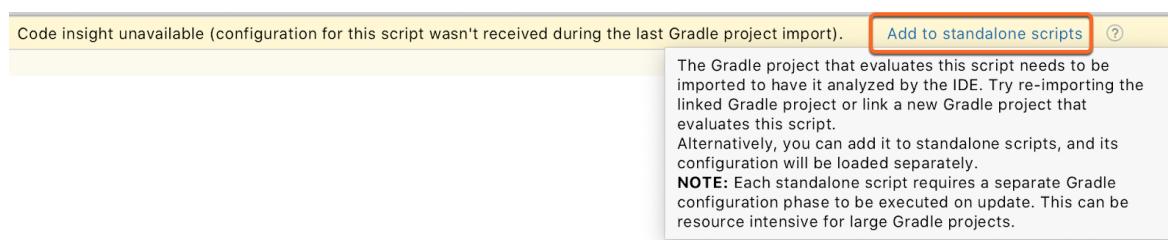
We've added one more action in IntelliJ IDEA 2020.1 with Gradle 6.0 and above – **Load Script Configurations**, which loads changes to the script configurations without updating the whole project. This takes much less time than reimporting the whole project.



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.



1.5.30 的新特性

- *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

1.5.30 的新特性

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.

1.5.30 的新特性

```
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
    //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

1.5.30 的新特性

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 arg

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

1.5.30 的新特性

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()`

1.5.30 的新特性

```
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. Double-ended queue lets you add or remove elements both at the beginning and the 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.

1.5.30 的新特性

```
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)

1.5.30 的新特性

```
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.

1.5.30 的新特性

```
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

1.5.30 的新特性

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

1.5.30 的新特性

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

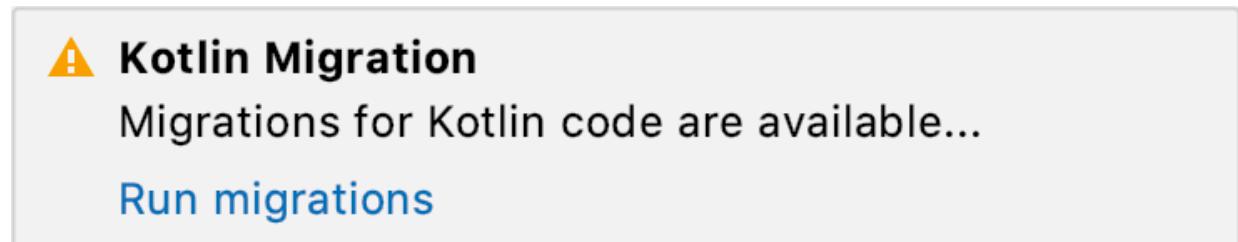
In order to avoid confusion about artifact names, we've renamed `kotlin-scripting-jsr223-embeddable` and `kotlin-scripting-jvm-host-embeddable` to just `kotlin-scripting-jsr223` and `kotlin-scripting-jvm-host`. These artifacts depend on the `kotlin-compiler-embeddable` artifact, which shades the bundled 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

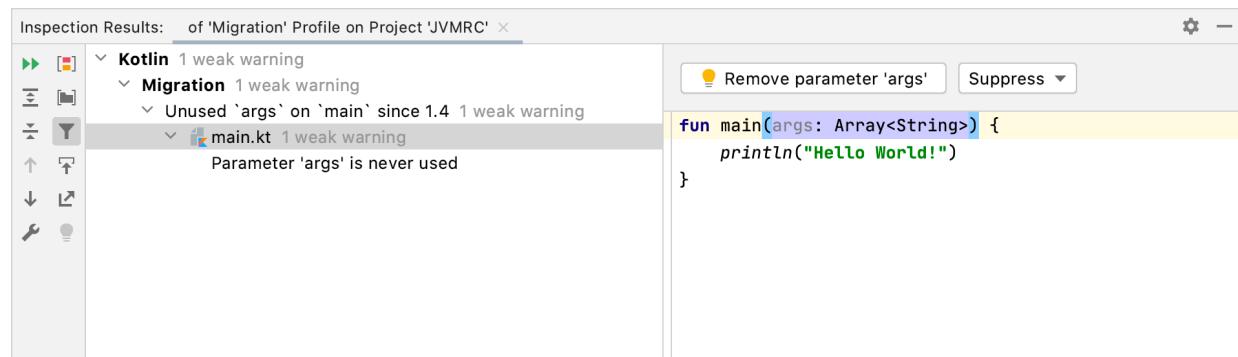
The Kotlin plugin's migration tools help you migrate your projects from earlier versions of Kotlin to 1.4.0.

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

发布日期: 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

1.5.30 的新特性

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：

1.5.30 的新特性

```
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：

1.5.30 的新特性

```
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!")
        }
    }
}
```

1.5.30 的新特性

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

1.5.30 的新特性

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](#).



1.5.30 的新特性

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：

1.5.30 的新特性

```
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
}
```

1.5.30 的新特性

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。

1.5.30 的新特性

```
fun main() {
//sampleStart
    val sourceArr = arrayOf("k", "o", "t", "l", "i", "n")
    val targetArr = sourceArr.copyOf(arrayOfNulls<String>(6), 3, startIndex = 3,
        println(targetArr.contentToString())

    sourceArr.copyOf(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
}
```

1.5.30 的新特性

此外，字符序列与字符串还有一个 `ifBlank` 扩展，它与 `isEmpty` 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](#)。

1.5.30 的新特性

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

发布日期: 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：

1.5.30 的新特性

```
// 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

1.5.30 的新特性

`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数参数具有默认值：

1.5.30 的新特性

```
//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：

1.5.30 的新特性

```
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参数的话。

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 单个具名参数

为了与注解中的数组字面值保持一致，向一个具名参数形式的 vararg 参数传入单个项目的形式（`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

1.5.30 的新特性

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")
}
```

1.5.30 的新特性

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）

1.5.30 的新特性

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

1.5.30 的新特性

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。

1.5.30 的新特性

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

发布日期: 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`:

1.5.30 的新特性

```
// 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

    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：

1.5.30 的新特性

```
//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：

1.5.30 的新特性

```
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

1.5.30 的新特性

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。

1.5.30 的新特性

```
//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：

1.5.30 的新特性

```
class ResourceLoader<T>(id: ResourceID<T>) {  
    operator fun provideDelegate(thisRef: MyUI, property: KProperty<*>): ReadOnlyPr  
        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](#)：

1.5.30 的新特性

```
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。

1.5.30 的新特性

```
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
```

1.5.30 的新特性

```
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：

1.5.30 的新特性

```
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()

1.5.30 的新特性

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。

1.5.30 的新特性

```
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](#)。

1.5.30 的新特性

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)
    }
}
```

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 学院上的免费 [Kotlin 基础课程](#) 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

1.5.30 的新特性

`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
}
```

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)}")
}
```

1.5.30 的新特性

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

定义只读局部变量使用关键字 `val` 定义。只能为其赋值一次。

```
fun main() {
//sampleStart
    val a: Int = 1 // 立即赋值
    val b = 2 // 自动推断出 `Int` 类型
    val c: Int // 如果没有初始值类型不能省略
    c = 3 // 明确赋值
//sampleEnd
    println("a = $a, b = $b, c = $c")
}
```

可重新赋值的变量使用 `var` 关键字。

1.5.30 的新特性

```
fun main() {
    //sampleStart
    var x = 5 // 自动推断出 `Int` 类型
    x += 1
    //sampleEnd
    println("x = $x")
}
```

可以在顶层声明变量。

```
//sampleStart
val PI = 3.14
var x = 0

fun incrementX() {
    x += 1
}
//sampleEnd

fun main() {
    println("x = $x; PI = $PI")
    incrementX()
    println("incrementX()")
    println("x = $x; PI = $PI")
}
```

参见[属性](#)。

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(var height: Double, var length: Double) {
    var 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。

1.5.30 的新特性

```
class Rectangle(var height: Double, var length: Double) {
    var perimeter = (height + length) * 2
}
fun main() {
//sampleStart
    val rectangle = Rectangle(5.0, 2.0)
    println("The perimeter is ${rectangle.perimeter}")
//sampleEnd
}
```

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(var height: Double, var length: Double): Shape() {
    var perimeter = (height + length) * 2
}
```

参见[类以及对象与实例](#)。

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

1.5.30 的新特性

```
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](#)。

1.5.30 的新特性

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

1.5.30 的新特性

```
//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"))
}
```

参见 [when 表达式](#)。

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。

1.5.30 的新特性

```
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。

1.5.30 的新特性

```
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`：

1.5.30 的新特性

```
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-nullables)  
        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

1.5.30 的新特性

```
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：

1.5.30 的新特性

```
//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)} ?: \"E
    }
    printLength("Incomprehensibilities")
    printLength(1000)
    printLength(listOf())
}
```

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)} ?: \"E
    }
    printLength("Incomprehensibilities")
    printLength(1000)
    printLength(listOf())
}
```

甚至

1.5.30 的新特性

```
//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)} ?: \"E
    }
    printLength("Incomprehensibilities")
    printLength("")
    printLength(1000)
}
```

参见[类](#)以及[类型转换](#)。

习惯用法

一些在 Kotlin 中广泛使用的语法规则，如果你有更喜欢的语法规则或者风格，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](#))

函数的默认参数

```
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

1.5.30 的新特性

```
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。

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

1.5.30 的新特性

```
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 until 100) { ..... } // 半开区间：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 {  
    // 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"  
}
```

实例化一个抽象类

1.5.30 的新特性

```
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()  
  
println(files?.size ?: "empty") // 如果 files 为 null, 那么输出“empty”  
  
// To calculate the fallback value in a code block, use `run`  
val fileSize = files?.size ?: run {  
    return someSize  
}  
println(fileSize)
```

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 {  
    ..... // 代码会执行到此处，假如data不为null  
}
```

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

1.5.30 的新特性

```
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：

1.5.30 的新特性

```
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

1.5.30 的新特性

```
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 JsonSyntax
//     .....
//
//     inline fun <reified T: Any> Gson.fromJson(json: JsonElement): T = this.fromJson(jso
```

可空布尔

```
val b: Boolean? = .....
if (b == true) {
    .....
} else {
    // `b` 是 false 或者 null
}
```

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")
```

1.5.30 的新特性

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[在 Java 与 Kotlin 中如何处理字符串的典型任务](#)

例学 Kotlin

 [例学 Kotlin](#)

编码规范

Commonly known and easy-to-follow coding conventions are vital for any programming language. Here we provide guidelines on the code style and code organization for projects that use Kotlin.

Configure style in IDE

Two most popular IDEs for Kotlin - [IntelliJ IDEA](#) and [Android Studio](#) provide powerful support for code styling. You can configure them to automatically format your code in consistence with the given code style.

应用风格指南

1. Go to **Settings/Preferences | Editor | Code Style | Kotlin**.
2. Click **Set from....**
3. Select **Kotlin style guide** .

Verify that your code follows the style guide

1. Go to **Settings/Preferences | Editor | Inspections | Kotlin**.
2. Open **Kotlin | Style issues**.
3. Switch on **File is not formatted according to project settings** inspection.
Additional inspections that verify other issues described in the style guide (such as naming conventions) are enabled by default.

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。

1.5.30 的新特性

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（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。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](#)（也称为 [Pascal 风格](#)），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。

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

1.5.30 的新特性

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`）。
- 类与对象的名称以大写字母开头并使用驼峰风格：

```
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，Android 运行时目前不支持这样的方法名。测试代码中也允许方法名使用下划线。

1.5.30 的新特性

```
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）应该使用大写、下划线分隔的 ([screaming snake case](#)) 名称：

```
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，可以使用大写、下划线分隔的名称 ([screaming snake case](#)) (`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`。

1.5.30 的新特性

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名称的一部分时，如果缩写由两个字母组成，就将其大写（`IOutputStream`）；而如果缩写更长一些，就只大写其首字母（`XmlFormatter`、`HttpInputStream`）。

格式化

Indentation

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号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。

1.5.30 的新特性

```
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 { /*....*/ }
}
```

1.5.30 的新特性

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：

1.5.30 的新特性

```
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：

1.5.30 的新特性

```
@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，并且用一个空自行与 `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。

1.5.30 的新特性

```
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：

1.5.30 的新特性

```
if (!component.isSyncing &&
    !hasAnyKotlinRuntimeInScope(module)
) {
    return createKotlinNotConfiguredPanel(module)
}
```

This helps align the condition and statement bodies.

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

1.5.30 的新特性

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：

1.5.30 的新特性

```
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)
}
```

Trailing commas

A trailing comma is a comma symbol after the last item of a series of elements:

```
class Person(
    val firstName: String,
    val lastName: String,
    val age: Int, // trailing comma
)
```

Using trailing commas has several benefits:

- It makes version-control diffs cleaner – as all the focus is on the changed value.
- It makes it easy to add and reorder elements – there is no need to add or delete the comma if you manipulate elements.
- It simplifies code generation, for example, for object initializers. The last element can also have a comma.

Trailing commas are entirely optional – your code will still work without them. The Kotlin style guide encourages the use of trailing commas at the declaration site and leaves it at your discretion for the call site.

To enable trailing commas in the IntelliJ IDEA formatter, go to **Settings/Preferences | Editor | Code Style | Kotlin**, open the **Other** tab and select the **Use trailing comma** option.

Enumerations

1.5.30 的新特性

```
enum class Direction {  
    NORTH,  
    SOUTH,  
    WEST,  
    EAST, // trailing comma  
}
```

Value arguments

```
fun shift(x: Int, y: Int) { /*...*/ }  
shift(  
    25,  
    20, // trailing comma  
)  
val colors = listOf(  
    "red",  
    "green",  
    "blue", // trailing comma  
)
```

Class properties and parameters

```
class Customer(  
    val name: String,  
    val lastName: String, // trailing comma  
)  
class Customer(  
    val name: String,  
    lastName: String, // trailing comma  
)
```

Function value parameters

1.5.30 的新特性

```
fun powerOf(
    number: Int,
    exponent: Int, // trailing comma
) { /*...*/ }
constructor(
    x: Comparable<Number>,
    y: Iterable<Number>, // trailing comma
) {}
fun print(
    vararg quantity: Int,
    description: String, // trailing comma
) {}
```

Parameters with optional type (including setters)

```
val sum: (Int, Int, Int) -> Int = fun(
    x,
    y,
    z, // trailing comma
): Int {
    return x + y + z
}
println(sum(8, 8, 8))
```

Indexing suffix

```
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, // trailing comma
    ]
```

Parameters in lambdas

1.5.30 的新特性

```
fun main() {
    val x = {
        x: Comparable<Number>,
        y: Iterable<Number>, // trailing comma
        ->
        println("1")
    }
    println(x)
}
```

when entry

```
fun isReferenceApplicable(myReference: KClass<*>) = when (myReference) {
    Comparable::class,
    Iterable::class,
    String::class, // trailing comma
    -> true
    else -> false
}
```

Collection literals (in annotations)

```
annotation class ApplicableFor(val services: Array<String>)
@ApplicableFor([
    "serializer",
    "balancer",
    "database",
    "inMemoryCache", // trailing comma
])
fun run() {}
```

Type arguments

```
fun <T1, T2> foo() {}
fun main() {
    foo<
        Comparable<Number>,
        Iterable<Number>, // trailing comma
        >()
}
```

Type parameters

1.5.30 的新特性

```
class MyMap<
    MyKey,
    MyValue, // trailing comma
> {}
```

Destructuring declarations

```
data class Car(val manufacturer: String, val model: String, val year: Int)
val myCar = Car("Tesla", "Y", 2019)
val (
    manufacturer,
    model,
    year, // trailing comma
) = myCar
val cars = listOf<Car>()
fun printMeanValue() {
    var meanValue: Int = 0
    for ((
        ,
        ,
        year, // trailing comma
    ) 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。 */
```

1.5.30 的新特性

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 return type

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。

1.5.30 的新特性

```
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")
```

默认参数值

优先声明带有默认参数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：

1.5.30 的新特性

```
typealias MouseClickHandler = (Any, MouseEvent) -> Unit
typealias PersonIndex = Map<String, Person>
```

If you use a private or internal type alias for avoiding name collision, prefer the `import ... as ...` mentioned in [Packages and Imports](#).

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。

具名参数

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，请使用具名参数语法，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：

1.5.30 的新特性

```
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`。For example, use this syntax with `if` :

```
if (x == null) ... else ...
```

instead of this one with `when` :

```
when (x) {
    null -> // ....
    else -> // ....
}
```

如果有三个或多个选项时优先使用 `when`。

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。

1.5.30 的新特性

区间上循环

使用 `until` 函数在一个开区间上循环：

```
for (i in 0..n - 1) { /*...*/ } // 不良
for (i in 0 until n) { /*...*/ } // 良好
```

Strings

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

1.5.30 的新特性

在某些情况下，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参数值的单个构造函数，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：

1.5.30 的新特性

```
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）

概念

- 类型
 - 基本类型
 - 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
 - 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
- 类型安全的构建器
- 空安全
- 相等性

1.5.30 的新特性

- `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
- 类型检测与类型转换

基本类型

在 Kotlin 中，所有东西都是对象，在这个意义上讲我们可以在任何变量上调用成员函数与属性。一些类型可以有特殊的内部表示——例如，数字、字符以及布尔可以在运行时表示为原生类型值，但是对于用户来说，它们看起来就像普通的类。在本节中，我们会描述 Kotlin 中使用的基本类型：[数字](#)、[布尔](#)、[字符](#)、[数组与字符串](#)。

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，有四种不同大小的类型，因此值的范围也不同。

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	$-9,223,372,036,854,775,808 (-2^{63})$	$9,223,372,036,854,775,807 (2^{63} - 1)$

所有以未超出 `Int` 最大值的整型值初始化的变量都会推断为 `Int` 类型。如果初始值超过了其最大值，那么推断为 `Long` 类型。如需显式指定 `Long` 型值，请在该值后追加 `L` 后缀。

```
val one = 1 // Int
val threeBillion = 3000000000 // Long
val oneLong = 1L // Long
val oneByte: Byte = 1
```

浮点类型

1.5.30 的新特性

对于实数，Kotlin 提供了浮点类型 `Float` 与 `Double` 类型。根据 [IEEE 754 标准](#)，两种浮点类型的十进制位数（即可以存储多少位十进制数）不同。`Float` 反映了 IEEE 754 单精度，而 `Double` 提供了双精度。

类型	大小 (比特数)	有效数字比特数	指数比特数	十进制位数
<code>Float</code>	32	24	8	6-7
<code>Double</code>	64	53	11	15-16

可以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 // 错误：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。如果这样的值包含多于 6~7 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
```

请注意，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() {
    fun printDouble(d: Double) { print(d) }

    val i = 1
    val d = 1.0
    val f = 1.0f

    printDouble(d)
    //    printDouble(i) // 错误：类型不匹配
    //    printDouble(f) // 错误：类型不匹配
}
```

如需将数值转换为不同的类型，请使用[显式转换](#)。

字面常量

1.5.30 的新特性

数值常量字面值有以下几种:

- 十进制: 123
 - Long 类型用大写 L 标记: 123L
- 十六进制: 0x0F
- 二进制: 0b00001011

不支持八进制。



Kotlin 同样支持浮点数的常规表示方法:

- 默认 double: 123.5、123.5e10
- Float 用 f 或者 F 标记: 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
```

JVM 平台的数字表示

在 JVM 平台数字存储为原生类型 int、double 等。例外情况是当创建可空数字引用如 Int? 或者使用泛型时。在这些场景中，数字会装箱为 Java 类 Integer、Double 等。

请注意，对相同数字的可为空引用可能是不同的对象：

1.5.30 的新特性

```
fun main() {
//sampleStart
    val a: Int = 100
    val boxedA: Int? = a
    val anotherBoxedA: Int? = a

    val b: Int = 10000
    val boxedB: Int? = b
    val anotherBoxedB: Int? = b

    println(boxedA === anotherBoxedA) // true
    println(boxedB === anotherBoxedB) // false
//sampleEnd
}
```

由于 JVM 对 -128 到 127 的整数 (Integer) 应用了内存优化，因此， a 的所有可空引用实际上都是同一对象。但是没有对 b 应用内存优化，所以它们是不同对象。

另一方面，它们仍然相等：

```
fun main() {
//sampleStart
    val b: Int = 10000
    println(b == b) // 输出“true”
    val boxedB: Int? = b
    val anotherBoxedB: Int? = b
    println(boxedB == anotherBoxedB) // 输出“true”
//sampleEnd
}
```

显式转换

由于不同的表示方式，较小类型并不是较大类型的子类型。如果它们是的话，就会出现下述问题：

```
// 假想的代码，实际上并不能编译：
val a: Int? = 1 // 一个装箱的 Int (java.lang.Integer)
val b: Long? = a // 隐式转换产生一个装箱的 Long (java.lang.Long)
print(b == a) // 惊！这将输出“false”鉴于 Long 的 equals() 会检测另一个是否也为 Long
```

所以会悄无声息地失去相等性，更别说同一性了。

因此较小的类型 **不能** 隐式转换为较大的类型。这意味着把 Byte 型值赋给一个 Int 变量必须显式转换。

1.5.30 的新特性

```
fun main() {
    //sampleStart
    val b: Byte = 1 // OK, 字面值会静态检测
    // val i: Int = b // 错误
    val i1: Int = b.toInt()
    //sampleEnd
}
```

所有数字类型都支持转换为其他类型：

- `toByte(): Byte`
- `toShort(): Short`
- `toInt(): Int`
- `toLong(): Long`
- `toFloat(): Float`
- `toDouble(): Double`
- `toChar(): Char`

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，而算术运算会有重载做适当转换，例如：

```
val l = 1L + 3 // Long + Int => Long
```

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
}
```

还可以为自定义类覆盖这些操作符。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。

1.5.30 的新特性

```
fun main() {
    //sampleStart
    val x = 5 / 2
    //println(x == 2.5) // ERROR: Operator '==' cannot be applied to 'Int' and 'Double'
    println(x == 2)
    //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 == 2L)
    //sampleEnd
}
```

如需返回浮点类型，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`。

```
val x = (1 shl 2) and 0x000FF000
```

这是完整的位运算列表：

- `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，当这些操作数**并非**静态类型为浮点数（例如是 `Any`、`Comparable<.....>`、类型参数）时，这些操作使用为 `Float` 与 `Double` 实现的不符合标准的 `equals` 与 `compareTo`，这会出现：

- 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`

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：

- `UByte`：无符号 8 比特整数，范围是 0 到 255
- `UShort`：无符号 16 比特整数，范围是 0 到 65535
- `UIInt`：无符号 32 比特整数，范围是 0 到 $2^{32} - 1$
- `ULong`：无符号 64 比特整数，范围是 0 到 $2^{64} - 1$

无符号类型支持其对应有符号类型的大多数操作。

将类型从无符号类型更改为对应的有符号类型（反之亦然）是二进制不兼容变更。



无符号数组与区间

`Unsigned arrays and operations on them are in Beta.` They can be changed incompatibly at any time. Opt-in is required (see the details below).



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：

1.5.30 的新特性

- `UByteArray` : 无符号字节数组
- `UShortArray` : 无符号短整型数组
- `UIIntArray` : 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'll get a warning that indicates that this feature is not stable yet. To remove the warning, opt in using 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 array are not a stable feature, so an API which 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.

字面值

为使无符号整型更易于使用，Kotlin 提供了用后缀标记整型字面值来表示指定无符号类型 (类似于 `Float` 或 `Long`)：

- `u` 与 `U` 将字面值标记为无符号。The exact type is determined based on the expected type. If no expected type is provided, compiler will use `UInt` or `ULong` depending on the size of 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` 显式将字面值标记为无符号长整型。

```
val a = 1UL // ULong, 即使未提供预期类型并且常量适于 UInt
```

深入探讨

关于技术细节与深入探讨请参见[无符号类型的语言提案](#)。

布尔

The type `Boolean` represents boolean objects that can have two values: `true` and `false`.

`Boolean` has a nullable counterpart `Boolean?` that also has the `null` value.

Built-in operations on booleans include:

- `||` – disjunction (logical *OR*)
- `&&` – conjunction (logical *AND*)
- `!` - negation (logical *NOT*)

`||` and `&&` work lazily.

```
fun main() {
    //sampleStart
    val myTrue: Boolean = true
    val myFalse: Boolean = false
    val boolNull: Boolean? = null

    println(myTrue || myFalse)
    println(myTrue && myFalse)
    println(!myTrue)
    //sampleEnd
}
```

On JVM: nullable references to boolean objects are boxed similarly to [numbers](#).



字符

字符用 `Char` 类型表示。字符字面值用单引号括起来: `'1'`。

特殊字符可以以转义反斜杠 `\` 开始。支持这几个转义序列: `\t`、`\b`、`\n`、`\r`、`\'`、`\"`、`\\\` 与 `\$`。

编码其他字符要用 Unicode 转义序列语法: `'\uFF00'`。

1.5.30 的新特性

```
fun main() {
    //sampleStart
    val aChar: Char = 'a'

    println(aChar)
    println('\n') //prints an extra newline character
    println('\uFF00')
    //sampleEnd
}
```

If a value of character variable is a digit, you can explicitly convert it to an `Int` number using the `digitToInt()` function.

On JVM: Like `numbers`, characters are boxed when a nullable reference is needed. Identity is not preserved by the boxing operation.



字符串

Kotlin 中字符串用 `String` 类型表示。Generally, a string value is a sequence of characters in double quotes (").

```
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]`。 You can iterate over these characters with a `for` loop:

```
fun main() {
    val str = "abcd"
    //sampleStart
    for (c in str) {
        println(c)
    }
    //sampleEnd
}
```

字符串是不可变的。Once you initialize a string, you can't change its value or assign a new value to it. All operations that transform strings return their results in a new `String` object, leaving the original string unchanged.

1.5.30 的新特性

```
fun main() {
//sampleStart
    val str = "abcd"
    println(str.uppercase()) // Create and print a new String object
    println(str) // the original string remains the same
//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")
//sampleEnd
}
```

请注意，在大多数情况下，优先使用[字符串模板](#)或原始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：

- 转义字符串可以包含转义字符
- 原始字符串可以包含换行以及任意文本

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。参见上面的[字符](#)查看支持的转义序列。

原始字符串 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)
"""
```

To remove leading whitespace from raw strings, use the `trimMargin()` function:

1.5.30 的新特性

```
val text = """
|Tell me and I forget.
|Teach me and I remember.
|Involve me and I learn.
|(Benjamin Franklin)
""".trimMargin()
```

默认 `|` 用作边界前缀，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。模板表达式以美元符 (`$`) 开头，要么由一个的名称构成：

```
fun main() {
//sampleStart
    val i = 10
    println("i = $i") // 输出“i = 10”
//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}") // 输出“abc.length is 3”
//sampleEnd
}
```

You can use templates both in raw and escaped strings. To insert the `$` character in a raw string (which doesn't support backslash escaping) before any symbol, which is allowed as a beginning of an [identifier](#), use the following syntax:

```
val price = """
${'$'}_9.99
"""
```

数组

1.5.30 的新特性

数组在 Kotlin 中使用 `Array` 类来表示。它定义了 `get` 与 `set` 函数（按照运算符重载约定这会转变为 `[]`）与 `size` 属性及其他有用的成员函数：

```
class Array<T> private constructor() {
    val size: Int
        operator fun get(index: Int): T
        operator fun set(index: Int, value: T): Unit

    operator fun iterator(): Iterator<T>
    // ...
}
```

可以使用函数 `arrayOf()` 来创建一个数组并传递元素值给它，这样 `arrayOf(1, 2, 3)` 创建了 `array [1, 2, 3]`。或者，函数 `arrayOfNulls()` 可以用于创建一个指定大小的、所有元素都为空的数组。

另一个选项是用接受数组大小以及一个函数参数的 `Array` 构造函数，用作参数的函数能够返回给定索引的元素：

```
fun main() {
    //sampleStart
    // 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 { println(it) }
    //sampleEnd
}
```

如上所述，`[]` 运算符代表调用成员函数 `get()` 与 `set()`。

Kotlin 中数组是不型变的 (*invariant*)。这意味着 Kotlin 不让我们把 `Array<String>` 赋值给 `Array<Any>`，以防止可能的运行时失败（但是你可以使用 `Array<out Any>`，参见 [类型投影](#)）。

原生类型数组

Kotlin 也有无装箱开销的类来表示原生类型数组：`ByteArray`、`ShortArray`、`IntArray` 等等。这些类与 `Array` 并没有继承关系，但是它们有同样的方法属性集。它们也都有相应的工厂方法：

```
val x: IntArray = intArrayOf(1, 2, 3)
x[0] = x[1] + x[2]
```

1.5.30 的新特性

```
// 大小为 5、值为 [0, 0, 0, 0, 0] 的整型数组
val arr = IntArray(5)

// 例如：用常量初始化数组中的值
// 大小为 5、值为 [42, 42, 42, 42, 42] 的整型数组
val arr = IntArray(5) { 42 }

// 例如：使用 lambda 表达式初始化数组中的值
// 大小为 5、值为 [0, 1, 2, 3, 4] 的整型数组（值初始化为其索引值）
var arr = IntArray(5) { it * 1 }
```

类型检测与类型转换

is 与 !is 操作符

使用 `is` 操作符或其否定形式 `!is` 在运行时检测对象是否符合给定类型：

```
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在 Kotlin 中使用显式转换操作符，因为编译器跟踪不可变值的 `is`-检测以及 [显式转换](#)，并在必要时自动插入（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
```

或者在 `&&` 或 `||` 的右侧 and the proper check (regular or negative) is on the left-hand side：

1.5.30 的新特性

```
// `||` 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
}
```

智能转换用于 `when` 表达式 和 `while` 循环 也一样：

```
when (x) {
    is Int -> print(x + 1)
    is String -> print(x.length + 1)
    is IntArray -> print(x.sum())
}
```

请注意，当编译器能保证变量在检测和使用之间不可改变时，智能转换才有效。更具体地，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` 或 `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——如果变量在检测和使用之间没有修改、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

通常，如果转换是不可能的，转换操作符会抛出一个异常。因此，称为**不安全的**。

Kotlin 中的不安全转换使用中缀操作符 `as`。

```
val x: String = y as String
```

请注意，`null` 不能转换为 `String` 因该类型不是[可空的](#)。如果 `y` 为空，上面的代码会抛出一个异常。为了让这样的代码用于可空值，请在类型转换的右侧使用可空类型：

```
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。

类型擦除与泛型检测

Kotlin 在编译时确保涉及 [泛型](#) 操作的类型安全性，而在运行时，泛型类型的实例并未带有关于它们实际类型参数的信息。例如，`List<Foo>` 会被擦除为 `List<*>`。通常，在运行时无法检测一个实例是否属于带有某个类型参数的泛型类型。

因此，编译器会禁止由于类型擦除而无法执行的 `is` 检测，例如 `ints is List<Int>` 或者 `list is T`（类型参数）。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) } // 这些项的类型都是 `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: List<String>) {
    if (list is ArrayList) {
        // `list` 会智能转换为 `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`。

带有[具体化的类型参数](#)的内联函数使其类型实参在每个调用处内联。这就能够对类型参数进行 `arg is T` 检测，但是如果 `arg` 自身是一个泛型实例，其类型参数还是会被擦除。

1.5.30 的新特性

```
//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
// 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
}
```

非受检类型转换

如上所述，类型擦除使运行时不可能对泛型类型实例的类型实参进行检测。另外代码中的泛型可能相互连接不够紧密，以致于编译器无法确保类型安全。

即便如此，有时候我们有高级的程序逻辑来暗示类型安全。例如：

```
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”。

1.5.30 的新特性

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。



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。

```
var max = a
if (a < b) max = b

// With else
var max: Int
if (a > b) {
    max = a
} else {
    max = b
}

// 作为表达式
val max = if (a > b) a else b
```

`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 表达式

`when` defines a conditional expression with multiple branches. It is similar to the `switch` statement in C-like languages. Its simple form looks like this.

1.5.30 的新特性

```
when (x) {
    1 -> print("x == 1")
    2 -> print("x == 2")
    else -> {
        print("x is neither 1 nor 2")
    }
}
```

`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。

`when` 既可以作为表达式使用也可以作为语句使用。如果它被当做表达式，第一个符合条件的分支的值就是整个表达式的值，如果当做语句使用，则忽略个别分支的值。类似于 `if`，每一个分支可以是一个代码块，它的值是块中最后的表达式的值。

如果其他分支都不满足条件将会求值 `else` 分支。如果 `when` 作为一个表达式使用，那么必须有 `else` 分支，除非编译器能够检测出所有的可能情况都已经覆盖了，例如，对于 枚举（enum）类条目与密封（sealed）类子类型】。

```
enum class Bit {
    ZERO, ONE
}

val numericValue = when (getRandomBit()) {
    Bit.ZERO -> 0
    Bit.ONE -> 1
    // 'else' is not required because all cases are covered
}
```

In `when statements`, the `else` branch is mandatory in the following conditions:

- `when` has a subject of an `Boolean` , `enum` , or `sealed` type, or their nullable counterparts.
- branches of `when` don't cover all possible cases for this subject.

1.5.30 的新特性

```
enum class Color {
    RED, GREEN, BLUE
}

when (getColor()) {
    Color.RED -> println("red")
    Color.GREEN -> println("green")
    Color.BLUE -> println("blue")
    // 'else' is not required because all cases are covered
}

when (getColor()) {
    Color.RED -> println("red") // no branches for GREEN and BLUE
    else -> println("not red") // 'else' is required
}
```

To define a common behavior for multiple cases, combine 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")
}
```

还可以检测一个值在 (`in`) 或者不在 (`!in`) 一个区间或者集合中：

```
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")
}
```

另一种选择是检测一个值是 (`is`) 或者不是 (`!is`) 一个特定类型的值。注意：由于智能转换，你可以访问该类型的方法与属性而无需任何额外的检测。

1.5.30 的新特性

```
fun hasPrefix(x: Any) = when(x) {
    is String -> x.startsWith("prefix")
    else -> false
}
```

`when` 也可以用来取代 `if - else if` 链。如果不提供参数，所有的分支条件都是简单的布尔表达式，而当一个分支的条件为真时则执行该分支：

```
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 `when` 的主语（subject，译注：指 `when` 所判断的表达式）捕获到变量中：

```
fun Request.getBody() =
    when (val response = executeRequest()) {
        is Success -> response.body
        is HttpError -> throw HttpException(response.status)
    }
```

在 `when` 主语中引入的变量的作用域仅限于 `when` 主体。

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：

1.5.30 的新特性

- 有一个成员函数或者扩展函数 `iterator()` 返回 `Iterator<T>`：
 - 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) {  
        println(i)  
    }  
    for (i in 6 downTo 0 step 2) {  
        println(i)  
    }  
    //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) {  
        println(array[i])  
    }  
    //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")  
    }  
    //sampleEnd  
}
```

while 循环

1.5.30 的新特性

`while` 和 `do-while` 循环会不断执行它们的主体，直到它们的条件满足。它们之间的区别在于条件检查时间：

- `while` 检查条件，如果满足，执行主体，然后返回到条件检查。
- `do-while` 执行主体，然后检查条件。如果满足，循环重复。所以，`do-while` 执行主体至少执行一次，无论条件是否满足。

```
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，我们可以用标签限定 `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。

返回到标签

1.5.30 的新特性

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` 允许我们从外层函数返回。最重要的一个用途就是从 lambda 表达式中返回。回想一下我们这么写的时候，这个 `return` 表达式从最直接包围它的函数——`foo` 中返回：

```
//sampleStart
fun foo() {
    listOf(1, 2, 3, 4, 5).forEach {
        if (it == 3) return // 非局部直接返回到 foo() 的调用者
        print(it)
    }
    println("this point is unreachable")
}
//sampleEnd

fun main() {
    foo()
}
```

注意，这种非局部的返回只支持传给[内联函数](#)的 lambda 表达式。如需从 lambda 表达式中返回，可给它加标签并用以限定 `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)
    }
    println(" 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。

1.5.30 的新特性

```
//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：

1.5.30 的新特性

```
//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) ”。

异常

异常类

Kotlin 中所有异常类继承自 `Throwable` 类。每个异常都有消息、堆栈回溯信息以及可选的原因。

使用 `throw` 表达式来抛出异常：

```
fun main() {
    //sampleStart
    throw Exception("Hi There!")
    //sampleEnd
}
```

使用 `try catch` 表达式来捕获异常：

```
try {
    // 一些代码
} catch (e: SomeException) {
    // 处理程序
} finally {
    // 可选的 finally 块
}
```

可以有零到多个 `catch` 块，`finally` 块可以省略。但是 `catch` 与 `finally` 块至少需有一个。

Try 是一个表达式

`try` 是一个表达式，意味着它可以有一个返回值：

```
val a: Int? = try { input.toInt() } catch (e: NumberFormatException) { null }
```

`try`-表达式的返回值是 `try` 块中的最后一个表达式或者是（所有）`catch` 块中的最后一个表达式。`finally` 块中的内容不会影响表达式的结果。

受检异常

Kotlin 没有受检异常。这其中有很多原因，但我们会提供一个简单的示例 that illustrates why it is the case。

以下是 JDK 中 `StringBuilder` 类实现的一个示例接口：

```
Appendable append(CharSequence csq) throws IOException;
```

这个签名是说，每次我追加一个字符串到一些东西（一个 `StringBuilder`、某种日志、一个控制台等）上时，我就必须捕获 `IOException`。为什么？因为相应实现可能正在执行 IO 操作（`Writer` 也实现了 `Appendable`）。其结果是这种代码随处可见：

```
try {
    log.append(message)
} catch (IOException e) {
    // 必须要安全
}
```

这并不好，看看《Effective Java》第三版 第 77 条：不要忽略异常就知道了。

Bruce Eckel says this about checked exceptions:

通过一些小程序测试得出的结论是异常规范会同时提高开发者的生产力与代码质量，但是大型软件项目的经验表明一个不同的结论——生产力降低、代码质量很少或没有提高。

And here are some additional thoughts on the matter:

- 《Java 的受检异常是一个错误》 (Java's checked exceptions were a mistake) (Rod Waldhoff)
- 《受检异常的烦恼》 (The Trouble with Checked Exceptions) (Anders Hejlsberg)

If you want to alert callers about possible exceptions when calling Kotlin code from Java, Swift, or Objective-C, you can use the `@Throws` annotation. Read more about using this annotation [for Java](#) and [for Swift and Objective-C](#).

Nothing 类型

1.5.30 的新特性

在 Kotlin 中 `throw` 是表达式，所以你可以使用它（比如）作为 Elvis 表达式的一部分：

```
val s = person.name ?: throw IllegalArgumentException("Name required")
```

`throw` 表达式的类型是 `Nothing` 类型。这个类型没有值，而是用于标记永远不能达到的代码位置。在你自己的代码中，你可以使用 `Nothing` 来标记一个永远不会返回的函数：

```
fun fail(message: String): Nothing {
    throw IllegalArgumentException(message)
}
```

当你调用该函数时，编译器会知道在该调用后就不再继续执行了：

```
val s = person.name ?: fail("Name required")
println(s)      // 在此已知“s”已初始化
```

当处理类型推断时还可能会遇到这个类型。这个类型的可空变体 `Nothing?` 有一个可能的值是 `null`。如果用 `null` 来初始化一个要推断类型的值，而又没有其他信息可用于确定更具体的类型时，编译器会推断出 `Nothing?` 类型：

```
val x = null          // “x”具有类型 `Nothing?`
val l = listOf(null)  // “l”具有类型 `List<Nothing?>`
```

Java 互操作性

与 Java 互操作性相关的信息，请参见 [Java 互操作性章节](#) 中的异常部分。

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
- 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 中的一个类可以有一个主构造函数以及一个或多个次构造函数。主构造函数是类头的一部分：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) { /*....*/ }
```

主构造函数不能包含任何的代码。初始化的代码可以放到以 `init` 关键字作为前缀的初始化块 (*initializer blocks*) 中。

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：

1.5.30 的新特性

```
//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 =
```

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
) { /*...*/ }
```

1.5.30 的新特性

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。委托给主构造函数会作为次构造函数的第一条语句，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：

1.5.30 的新特性

```
//sampleStart
class Constructors {
    init {
        println("Init block")
    }

    constructor(i: Int) {
        println("Constructor $i")
    }
}

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：

```
val invoice = Invoice()

val customer = Customer("Joe Smith")
```

1.5.30 的新特性

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。

1.5.30 的新特性

```
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：

1.5.30 的新特性

```
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：

1.5.30 的新特性

```
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（在此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  
  
    init { println("Initializing a derived class") }  
  
    override val size: Int =  
        (super.size + lastName.length).also { println("Initializing size in the der  
}  
//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

1.5.30 的新特性

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`：

1.5.30 的新特性

```
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.borderColor}")
        }
    }
}
//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>`：

1.5.30 的新特性

```
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：

1.5.30 的新特性

```
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 ge  
        get() = this.width * this.height  
}  
//sampleEnd  
fun main() {  
    val rectangle = Rectangle(3, 4)  
    println("Width=${rectangle.width}, height=${rectangle.height}, area=${rectangle  
}
```

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。

如果你需要改变对一个访问器进行注解或者改变其可见性，但是不需要改变默认的实现，你可以定义访问器而不定义其实现：

1.5.30 的新特性

```
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'
    }
```

`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")
    }
```

1.5.30 的新特性

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：

1.5.30 的新特性

```
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

1.5.30 的新特性

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：

1.5.30 的新特性

```
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：

1.5.30 的新特性

```
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

只有一个抽象方法的接口称为 **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，但只能有一个抽象成员。

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：

1.5.30 的新特性

```
// 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](#)。

函数式接口与类型别名比较

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以及一个抽象成员。 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，默認為 `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：

1.5.30 的新特性

```
// 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` 移位置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：

1.5.30 的新特性

```
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' 是 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的。默认情况下，所有构造函数都是 `public`，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）。

局部声明

局部变量、函数和类不能有可见性修饰符。

模块

1.5.30 的新特性

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。

扩展函数是静态分发的，即他们不是根据接收者类型的虚方法。调用的扩展函数是由函数调用所在的表达式的类型来决定的，而不是由表达式运行时求值结果决定的。例如：

```
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：

1.5.30 的新特性

```
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`，并且可以在函数体内检测 `this == null`。

这样，就可以在没有检测 `null` 的时候调用 Kotlin 中的 `toString()`：检测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：

1.5.30 的新特性

```
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

1.5.30 的新特性

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("kotlin"), 443).connect()
    //Host("kotlin").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。

1.5.30 的新特性

```
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"—一分
    DerivedCaller().call(Derived()) // "Base extension function in DerivedCaller"—二分
}
```

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

创建一些只保存数据的类是件寻常的事。在这些类中，一些标准功能以及一些工具函数往往是由数据机械推导而来的。在 Kotlin 中，这叫做 **数据类** 并以 `data` 标记：

```
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：

- 主构造函数需要至少有一个参数。
- 主构造函数的所有参数需要标记为 `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

1.5.30 的新特性

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  
}
```

在 `toString()`、`equals()`、`hashCode()` 以及 `copy()` 的实现中只会用到 `name` 属性，并且只有一个 `component` 函数 `component1()`。虽然两个 `Person` 对象可以有不同的年龄，但它们会视为相等。

```
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  
    //sampleEnd  
    println("person1 == person2: ${person1 == person2}")  
    println("person1 with age ${person1.age}: ${person1}")  
    println("person2 with age ${person2.age}: ${person2}")  
}
```

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

1.5.30 的新特性

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") // 输出 "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

`Sealed` classes and interfaces represent restricted class hierarchies that provide more control over inheritance. All direct subclasses of a sealed class are known at compile time. No other subclasses may appear after a module with the sealed class is compiled. For example, third-party clients can't extend your sealed class in their code. Thus, each instance of a sealed class has a type from a limited set that is known when this class is compiled.

The same works for sealed interfaces and their implementations: once a module with a sealed interface is compiled, no new implementations can appear.

In some sense, sealed classes are similar to `enum` classes: the set of values for an enum type is also restricted, but each enum constant exists only as a *single instance*, whereas a subclass of a sealed class can have *multiple instances*, each with its own state.

As an example, consider a library's API. It's likely to contain error classes to let the 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 implementing or extending them in the client code. However, the library doesn't know about errors declared outside it, so it can't treat them consistently with its own classes. With a sealed hierarchy of error classes, library authors can be sure that they know all possible error types and no other ones can appear later.

To declare a sealed class or interface, put the `sealed` modifier before its name:

```
sealed interface Error

sealed class IOError(): Error

class FileReadError(val file: File): IOError()
class DatabaseError(val source: DataSource): IOError()

object RuntimeError : Error
```

一个密封类是自身抽象的，它不能直接实例化并可以有抽象（`abstract`）成员。

Constructors of sealed classes can have one of two `visibilities`: `protected` (by default) or `private`:

1.5.30 的新特性

```
sealed class IOError {  
    constructor() { /*...*/ } // protected by default  
    private constructor(description: String): this() { /*...*/ } // private is OK  
    // public constructor(code: Int): this() {} // Error: public and internal are n  
}
```

Location of direct subclasses

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 proper qualified name. They can't be local nor anonymous objects.

`enum` classes can't extend a sealed class (as well as any other class), but they can implement sealed interfaces.



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 same package and module  
  
sealed class IOError(): Error // extended only in same package and module  
open class CustomError(): Error // can be extended wherever it's visible
```

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 `expect` 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.](#)

Sealed classes and when expression

使用密封类的关键好处在于使用 `when 表达式` 的时候。如果能够验证语句覆盖了所有情况，就不需要为该语句再添加一个 `else` 子句了。当然，这只有当你用 `when` 作为表达式（使用结果）而不是作为语句时才有用。

```
fun log(e: Error) = when(e) {
    is FileReadError -> { println("Error while reading file ${e.file}") }
    is DatabaseError -> { println("Error while reading from database ${e.source}") }
    is RuntimeException -> { println("Runtime error") }
    // 不再需要 `else` 子句，因为已经覆盖了所有的情况
}
```

`when` 表达式在 `expect` 密封类的共同代码中仍然需要一个 `else` 分支。这是因为子类的实现可能在共同代码中未知。



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）。

我们来思考下为什么 Java 需要这些神秘的通配符。在 [《Effective Java》第三版](#) 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](#)。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>();
List<Object> objs = strs; // ! ! ! 此处的编译器错误让我们避免了之后的运行时异常。
objs.add(1); // 将一个整数放入一个字符串列表
String s = strs.get(0); // ! ! ! ClassCastException: 无法将整数转换为字符串
```

1.5.30 的新特性

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
void copyAll(Collection<Object> to, Collection<String> from) {
    to.addAll(from);
    // !!! 对于这种简单声明的 addAll 将不能编译：
    // Collection<String> 不是 Collection<Object> 的子类型
}
```

（在 Java 中，你很可能艰难地学到了这个教训，参见《Effective Java》第三版，第 28 条：列表优先于数组）

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`：在 Java 中有 `List<? super String>` 是 `List<Object>` 的一个超类。

1.5.30 的新特性

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 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”，并提出了以下助记符：

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
    // ....
}
```

1.5.30 的新特性

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

1.5.30 的新特性

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 `CharSequence` 数组或一个 `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：

1.5.30 的新特性

```
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<Ha
```

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() }
}
```

1.5.30 的新特性

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`。

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<*>`。

因此，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。

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。编译器会对非受检类型转换发出警告，并且在运行时只对非泛型部分检测（相当于 `foo as List<*>`）。

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。然而，内联函数的具体化的类型参数会由调用处内联函数体中的类型实参所代入，因此可以用于类型检测与转换，与上述泛型类型的实例具有相同限制。

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
```

1.5.30 的新特性

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

1.5.30 的新特性

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.values()) {
        println("$f($a, $b) = ${f.apply(a, b)}")
    }
}
```

使用枚举常量

Kotlin 中的枚举类也有合成方法用于列出定义的枚举常量以及通过名称获取枚举常量。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.values(): Array<EnumClass>
```

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。

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：

1.5.30 的新特性

```
enum class RGB { RED, GREEN, BLUE }

inline fun <reified T : Enum<T>> printAllValues() {
    print(enumValues<T>().joinToString { it.name })
}

printAllValues<RGB>() // 输出 RED, GREEN, BLUE
```

每个枚举常量都具有在枚举类声明中获取其名称与位置的属性：

```
val name: String
val ordinal: Int
```

枚举常量还实现了 `Comparable` 接口， 其中自然顺序是它们在枚举类中定义的顺序。

内联类

有时候，业务逻辑需要围绕某种类型创建包装器。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，性能的损失是很糟糕的，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)
```

The `inline` modifier for inline classes is deprecated.



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

1.5.30 的新特性

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, and have the `init` block:

```
@JvmInline
value class Name(val s: String) {
    init {
        require(s.length > 0) { }
    }

    val length: Int
        get() = s.length

    fun greet() {
        println("Hello, $s")
    }
}

fun main() {
    val name = Name("Kotlin")
    name.greet() // `greet` 方法会作为一个静态方法被调用
    println(name.length) // 属性的 get 方法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
}
```

1.5.30 的新特性

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必须是 `final`。

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'
    // 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。

名字修饰

1.5.30 的新特性

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重命名函数。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。

The mangling scheme has been changed in Kotlin 1.4.30. 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.



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。

1.5.30 的新特性

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
}
```

对象表达式与对象声明

有时候需要创建一个对某个类做了轻微改动的类的对象，而不用为之显式声明新的子类。Kotlin 可以用对象表达式与对象声明处理这种情况。

对象表达式

Object expressions create objects of anonymous classes, that is, classes that aren't explicitly declared with the `class` declaration. Such classes are useful for one-time use. You can define them from scratch, inherit from existing classes, or implement interfaces. Instances of anonymous classes are also called *anonymous objects* because they are defined by an expression, not a name.

Creating anonymous objects from scratch

Object expressions start with the `object` keyword.

If you just need an object that doesn't have any nontrivial supertypes, write its members in curly braces after `object :`

```
fun main() {
    //sampleStart
    val helloWorld = object {
        val hello = "Hello"
        val world = "World"
        // object expressions extend Any, so `override` is required on `toString()`
        override fun toString() = "$hello $world"
    }
    //sampleEnd
    print(helloWorld)
}
```

Inheriting anonymous objects from supertypes

如需创建一个继承自某个（或某些）类型的匿名类的对象，specify this type after `object` and a colon (`:`). Then implement or override the members of this class as if you were [inheriting](#) from it:

1.5.30 的新特性

```
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。多个超类型可以由跟在冒号后面的逗号分隔的列表指定：

```
open class A(x: Int) {
    public open val y: Int = x
}

interface B { /*...*/ }

val ab: A = object : A(1), B {
    override val y = 15
}
```

Using anonymous objects as return and value types

When an anonymous object is used as a type of a local or `private` but not `inline` declaration (function or property), all its members are accessible via this function or property:

```
class C {
    private fun getObject() = object {
        val x: String = "x"
    }

    fun printX() {
        println(getObject().x)
    }
}
```

If this function or property is public or private inline, 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

1.5.30 的新特性

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:

```
interface A {  
    fun funFromA() {}  
}  
interface B  
  
class C {  
    // The return type is Any. x is not accessible  
    fun getObject() = object {  
        val x: String = "x"  
    }  
  
    // The return type is A; x is not accessible  
    fun getObjectA() = object: A {  
        override fun funFromA() {}  
        val x: String = "x"  
    }  
  
    // The return type is B; funFromA() and x are not accessible  
    fun getObjectB(): B = object: A, B { // explicit return type is required  
        override fun funFromA() {}  
        val x: String = "x"  
    }  
}
```

Accessing variables from anonymous objects

对象表达式中的代码可以访问来自包含它的作用域的变量：

1.5.30 的新特性

```
fun countClicks(window: JComponent) {
    var clickCount = 0
    var enterCount = 0

    window.addMouseListener(object : MouseAdapter() {
        override fun mouseClicked(e: MouseEvent) {
            clickCount++
        }

        override fun mouseEntered(e: MouseEvent) {
            enterCount++
        }
    })
    // ....
}
```

对象声明

单例模式在一些场景中很有用，而 Kotlin 使单例声明变得很容易：

```
object DataProviderManager {
    fun registerDataProvider(provider: DataProvider) {
        // ....
    }

    val allDataProviders: Collection<DataProvider>
        get() = // ....
}
```

这称为对象声明。并且它总是在 `object` 关键字后跟一个名称。就像变量声明一样，对象声明不是一个表达式，不能用在赋值语句的右边。

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，直接使用其名称即可：

```
DataProviderManager.registerDataProvider.....)
```

这些对象可以有超类型：

1.5.30 的新特性

```
object DefaultListener : MouseAdapter() {
    override fun mouseClicked(e: MouseEvent) { ..... }

    override fun mouseEntered(e: MouseEvent) { ..... }
}
```

对象声明不能在局部作用域（即不能直接嵌套在函数内部），但是它们可以嵌套到其他对象声明或非内部类中。



伴生对象

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()
    }
}
```

该伴生对象的成员可通过只使用类名作为限定符来调用：

```
val instance = MyClass.create()
```

可以省略伴生对象的名称，在这种情况下将使用名称 `Companion`：

```
class MyClass {
    companion object {
    }

    val x = MyClass.Companion
```

Class members can access the private members of the corresponding companion object.

其自身所用的类的名称（不是另一个名称的限定符）可用作对该类的伴生对象（无论是否具名）的引用：

1.5.30 的新特性

```
class MyClass1 {  
    companion object Named { }  
}  
  
val x = MyClass1  
  
class MyClass2 {  
    companion object { }  
}  
  
val y = MyClass2
```

请注意，即使伴生对象的成员看起来像其他语言的静态成员，在运行时他们仍然是真实对象的实例成员，而且，例如还可以实现接口：

```
interface Factory<T> {  
    fun create(): T  
}  
  
class MyClass {  
    companion object : Factory<MyClass> {  
        override fun create(): MyClass = MyClass()  
    }  
}  
  
val f: Factory<MyClass> = MyClass
```

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。

对象表达式和对象声明之间的语义差异

对象表达式和对象声明之间有一个重要的语义差别：

- 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 b = BaseImpl(10)
    Derived(b).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`：

1.5.30 的新特性

```
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 b = BaseImpl(10)
    Derived(b).printMessage()
    Derived(b).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(val 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)。

1.5.30 的新特性

```
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。

1.5.30 的新特性

```
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

1.5.30 的新特性

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）相同或者与其超类型。
- `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：

1.5.30 的新特性

- `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）相同或者与其超类型。
- `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(): ReadWriteProperty<Any?, Int> =
    object : ReadWriteProperty<Any?, Int> {
        var curValue = 0
        override fun getValue(thisRef: Any?, property: KProperty<*>): Int = curValue
        override fun setValue(thisRef: Any?, property: KProperty<*>, value: Int) {
            curValue = value
        }
    }

val readOnly: Int by resourceDelegate() // ReadWriteProperty as val
var readWrite: Int by resourceDelegate()
```

Translation rules for delegated properties

本质上说，Kotlin 编译器会为每个委托属性生成辅助属性并委托给它。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。

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:

1.5.30 的新特性

```
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)
}
```

1.5.30 的新特性

`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)
}
```

1.5.30 的新特性

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：

1.5.30 的新特性

```
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

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

Functions are called using the standard approach:

```
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 { /*...*/ }
```

You can use a [trailing comma](#) when you declare function parameters:

```
fun powerOf(  
    number: Int,  
    exponent: Int, // trailing comma  
) { /*...*/ }
```

默认参数

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：

1.5.30 的新特性

```
fun read(  
    b: ByteArray,  
    off: Int = 0,  
    len: Int = b.size,  
) { /*...*/ }
```

A default value is defined using `=` after the type.

覆盖方法总是使用与基类型方法相同的默认参数值。当覆盖一个有默认参数值的方法时，必须从签名中省略默认参数值：

```
open class A {  
    open fun foo(i: Int = 10) { /*...*/ }  
}  
  
class B : A() {  
    override fun foo(i: Int) { /*...*/ } // 不能有默认值。  
}
```

如果一个默认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[具名参数](#)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
```

如果在默认参数之后的最后一个参数是[lambda 表达式](#)，那么它既可以作为具名参数在括号内传入，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
```

具名参数

1.5.30 的新特性

When calling a function, you can name one or more of its arguments. This can be helpful when a function has many arguments and it's difficult to associate a value with an argument, especially if it's a boolean or `null` value.

When you use named arguments in a function call, you can freely change the order they are listed in, and if you want to use their default values, you can just leave these arguments out altogether.

Consider the following function, `reformat()`, which has 4 arguments with default values.

```
fun reformat(  
    str: String,  
    normalizeCase: Boolean = true,  
    upperCaseFirstLetter: Boolean = true,  
    divideByCamelHumps: Boolean = false,  
    wordSeparator: Char = ' ',  
) { /*....*/ }
```

When calling this function, you don't have to name all its arguments:

```
reformat(  
    "String!",  
    false,  
    upperCaseFirstLetter = false,  
    divideByCamelHumps = true,  
    '_'  
)
```

You can skip all the ones with default values:

```
reformat("This is a long String!")
```

You are also able to skip specific arguments with default values, rather than omitting them all. However, after the first skipped argument, you must name all subsequent arguments:

```
reformat("This is a short String!", upperCaseFirstLetter = false, wordSeparator = '
```

You can pass a [variable number of arguments](#) (`vararg`) with names using the `spread` operator:

1.5.30 的新特性

```
fun foo(vararg strings: String) { /*....*/ }

foo(strings = *arrayOf("a", "b", "c"))
```

对于 JVM 平台：在调用 Java 函数时不能使用具名参数语法，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

当函数返回单个表达式时，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`，在这种情况下显式声明返回类型是[可选的](#)。

1.5.30 的新特性

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 is an Array
        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，可以使用具名参数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，那么使用伸展 (*spread*) 操作符（在数组前面加 `*`）：

```
val a = arrayOf(1, 2, 3)
val list = asList(-1, 0, *a, 4)
```

If you want to pass a `primitive type array` into `vararg`，you need to convert it to a regular (typed) array using the `toTypedArray()` function:

```
val a = intArrayOf(1, 2, 3) // IntArray is a primitive type array
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：

1.5.30 的新特性

- 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

1.5.30 的新特性

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 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：

1.5.30 的新特性

```
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。 For some algorithms that would normally use loops you can use a recursive function instead without a risk of stack overflow. 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)
    }
}
```

1.5.30 的新特性

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。

See also:

- [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](#) $(R, T) \rightarrow R$ ，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。

1.5.30 的新特性

```
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。

1.5.30 的新特性

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
```

1.5.30 的新特性

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：

1.5.30 的新特性

```
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

1.5.30 的新特性

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](#):

1.5.30 的新特性

```
class HTML {  
    fun body(): Unit { ..... }  
}  
  
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) { ..... }
```

1.5.30 的新特性

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）。



非局部返回

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：

1.5.30 的新特性

```
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` 和 `continue` 在内联的 lambda 表达式中还不可用，但我们也计划支持它们。



具体化的类型参数

1.5.30 的新特性

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>()
```

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"))
}
```

1.5.30 的新特性

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
<code>+a</code>	<code>a.unaryPlus()</code>
<code>-a</code>	<code>a.unaryMinus()</code>
<code>!a</code>	<code>a.not()</code>

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`。

1.5.30 的新特性

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`。

1.5.30 的新特性

- 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
<code>a + b</code>	<code>a.plus(b)</code>
<code>a - b</code>	<code>a.minus(b)</code>
<code>a * b</code>	<code>a.times(b)</code>
<code>a / b</code>	<code>a.div(b)</code>
<code>a % b</code>	<code>a.rem(b)</code>
<code>a..b</code>	<code>a.rangeTo(b)</code>

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
<code>a in b</code>	<code>b.contains(a)</code>
<code>a !in b</code>	<code>!b.contains(a)</code>

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

1.5.30 的新特性

表达式	翻译为
<code>a[i]</code>	<code>a.get(i)</code>
<code>a[i, j]</code>	<code>a.get(i, j)</code>
<code>a[i_1, ..., i_n]</code>	<code>a.get(i_1, ..., i_n)</code>
<code>a[i] = b</code>	<code>a.set(i, b)</code>
<code>a[i, j] = b</code>	<code>a.set(i, j, b)</code>
<code>a[i_1, ..., i_n] = b</code>	<code>a.set(i_1, ..., i_n, b)</code>

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
<code>a()</code>	<code>a.invoke()</code>
<code>a(i)</code>	<code>a.invoke(i)</code>
<code>a(i, j)</code>	<code>a.invoke(i, j)</code>
<code>a(i_1, ..., i_n)</code>	<code>a.invoke(i_1, ..., i_n)</code>

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
<code>a += b</code>	<code>a.plusAssign(b)</code>
<code>a -= b</code>	<code>a.minusAssign(b)</code>
<code>a *= b</code>	<code>a.timesAssign(b)</code>
<code>a /= b</code>	<code>a.divAssign(b)</code>
<code>a %= b</code>	<code>a.remAssign(b)</code>

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。

1.5.30 的新特性

- 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
<code>a == b</code>	<code>a?.equals(b) ?: (b === null)</code>
<code>a != b</code>	<code>!(a?.equals(b) ?: (b === null))</code>

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
<code>a > b</code>	<code>a.compareTo(b) > 0</code>
<code>a < b</code>	<code>a.compareTo(b) < 0</code>
<code>a >= b</code>	<code>a.compareTo(b) >= 0</code>
<code>a <= b</code>	<code>a.compareTo(b) <= 0</code>

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。

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）。以下是构建器的样例应用场景：

- 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"}


```

1.5.30 的新特性

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：

1.5.30 的新特性

```
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：

1.5.30 的新特性

```
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`：

1.5.30 的新特性

```
@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  
    }  
    // ....  
}
```

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。

1.5.30 的新特性

```
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) {
```

1.5.30 的新特性

```
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
}
```

空安全

可空类型与非空类型

Kotlin 的类型系统旨在消除来自代码空引用的危险，也称为《十亿美元的错误》。

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 中 NPE 的可能的原因只可能是：

- 显式调用 `throw NullPointerException()`。
- 使用了下文描述的 `!!` 操作符。
- 数据在初始化时不一致，例如当：
 - 传递一个在构造函数中出现的未初始化的 `this` 并用于其他地方（“泄漏 `this`”）。
 - 超类的构造函数调用一个开放成员，该成员在派生中类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；
 - 用于 Java 互操作的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>` 中加入 `null`，就需要一个 `MutableList<String?>` 才能处理。
 - 由外部 Java 代码引发的其他问题。

在 Kotlin 中，类型系统区分一个引用可以容纳 `null`（可空引用）还是不能容纳（非空引用）。例如，`String` 类型的常规变量不能容纳 `null`：

```
fun main() {
    //sampleStart
    var a: String = "abc" // 默认情况下，常规初始化意味着非空
    a = null // 编译错误
    //sampleEnd
}
```

如果要允许为空，可以声明一个变量为可空字符串（写作 `String?`）：

1.5.30 的新特性

```
fun main() {
//sampleStart
    var b: String? = "abc" // 可以设置为空
    b = null // ok
    print(b)
//sampleEnd
}
```

现在，如果你调用 `a` 的方法或者访问它的属性，它保证不会导致 `NPE`，这样你就放心地使用：

```
val l = a.length
```

但是如果你想访问 `b` 的同一个属性，那么这是不安全的，并且编译器会报告一个错误：

```
val l = b.length // 错误：变量“b”可能为空
```

但是，还是需要访问该属性，对吧？有几种方式可以做到。

在条件中检测 `null`

首先，你可以显式检测 `b` 是否为 `null`，并分别处理两种可能：

```
val l = if (b != null) b.length else -1
```

编译器会跟踪所执行检测的信息，并允许你在 `if` 内部调用 `length`。同时，也支持更复杂（更智能）的条件：

```
fun main() {
//sampleStart
    val b: String? = "Kotlin"
    if (b != null && b.length > 0) {
        print("String of length ${b.length}")
    } else {
        print("Empty string")
    }
//sampleEnd
}
```

1.5.30 的新特性

请注意，这只适用于 `b` 是不可变的情况（即在检测与使用之间没有修改过的局部变量，或是有幕后字段且不可覆盖的 `val` 成员），因为否则可能会发生在检测之后 `b` 又变为 `null` 的情况。

安全的调用

Your second option for accessing a property on a nullable variable is using the safe call operator `?.` :

```
fun main() {
    //sampleStart
    val a = "Kotlin"
    val b: String? = null
    println(b?.length)
    println(a?.length) // 无需安全调用
    //sampleEnd
}
```

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?`。

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
```

如果任意一个属性（环节）为 `null`，这个链式调用就会返回 `null`。

如果要只对非空值执行某个操作，安全调用操作符可以与 `let` 一起使用：

```
fun main() {
    //sampleStart
    val listWithNulls: List<String?> = listOf("Kotlin", null)
    for (item in listWithNulls) {
        item?.let { println(it) } // 输出 Kotlin 并忽略 null
    }
    //sampleEnd
}
```

安全调用也可以出现在赋值的左侧。这样，如果调用链中的任何一个接收者为 `null` 都会跳过赋值，而右侧的表达式根本不会求值：

1.5.30 的新特性

```
// 如果 `person` 或者 `person.department` 其中之一为空，都不会调用该函数：  
person?.department?.head = managersPool.getManager()
```

Elvis 操作符

当有一个可空的引用 `b` 时，可以说“如果 `b` 不是 `null`，就使用它；否则使用某个非空的值”：

```
val l: Int = if (b != null) b.length else -1
```

Instead of writing the complete `if` expression, you can also express this with the Elvis operator `?:` :

```
val l = b?.length ?: -1
```

如果 `?:` 左侧表达式不是 `null`，Elvis 操作符就返回其左侧表达式，否则返回右侧表达式。请注意，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，所以它们也可以用在 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? {  
    val parent = node.getParent() ?: return null  
    val name = node.getName() ?: throw IllegalArgumentException("name expected")  
    // ...  
}
```

!! 操作符

第三种选择是为 NPE 爱好者准备的：非空断言运算符（`!!`）将任何值转换为非空类型，若该值为 `null` 则抛出异常。可以写 `b!!`，这会返回一个非空的 `b` 值（例如：在我们示例中的 `String`）或者如果 `b` 为 `null`，就会抛出一个 `NPE` 异常：

```
val l = b!!.length
```

因此，如果你想要一个 NPE，你可以得到它，但是你必须显式要求它，否则它不会不期而至。

安全的类型转换

如果对象不是目标类型，那么常规类型转换可能会导致 `ClassCastException`。另一个选择是使用安全的类型转换，如果尝试转换不成功则返回 `null`：

```
val aInt: Int? = a as? Int
```

可空类型的集合

如果你有一个可空类型元素的集合，并且想要过滤非空元素，你可以使用 `filterNotNull` 来实现：

```
val nullableList: List<Int?> = listOf(1, 2, null, 4)
val intList: List<Int> = nullableList.filterNotNull()
```

相等性

Kotlin 中有两类相等性：

- **结构相等** (`==` ——用 `equals()` 检测)；
- **引用相等** (`===` ——两个引用指向同一对象)。

结构相等

结构相等由 `==` 以及其否定形式 `!=` 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。

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`。

如需提供自定义的相等检测实现，请覆盖 `equals(other: Any?): Boolean` 函数。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

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`。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。

浮点数相等性

1.5.30 的新特性

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](#)。

否则会使用不符合该标准的结构相等性检测，这会导致 `NaN` 等于其自身，而 `-0.0` 不等于 `0.0`。

参见：[浮点数比较](#)。

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
            }

            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

1.5.30 的新特性

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() {
    //sampleStart
    fun printLine() { println("Top-level 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) // Top-level function
    //sampleEnd()
}
```

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。

1.5.30 的新特性

- 线程不容易使用。线程的 Debug，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。该模式通常被称为标题圣诞树（大括号代表树的分支）。
- 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（这也可能会根据语言/平台而有不同的术语），是当我们发起调用的时候，我们承诺在某些时候它将返回一个名为 Promise 的可被操作的对象。

1.5.30 的新特性

```
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：

1.5.30 的新特性

“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。但是返回类型依然昰我们想要的类型。
- 编写这段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）。

1.5.30 的新特性

- 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

Asynchronous or non-blocking programming is an important part of the development landscape. 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 solves this problem in a flexible way by providing [coroutine](#) support at the language level and delegating most of the functionality to libraries.

In addition to opening the doors to asynchronous programming, coroutines also provide a wealth of other possibilities, such as concurrency and actors.

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](#)
- [Debug coroutines using IntelliJ IDEA](#)
- [Debug Kotlin Flow using IntelliJ IDEA – tutorial](#)

Sample projects

- [kotlinx.coroutines](#) 示例与源代码

1.5.30 的新特性

- [KotlinConf app](#)

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) { ..... }
```

1.5.30 的新特性

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 === o
```

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。

1.5.30 的新特性

```
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"))
}
```

Instantiation of annotation classes for Kotlin/Native is not yet available.



Learn more about instantiation of annotation classes in [this KEP](#).

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

1.5.30 的新特性

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：

1.5.30 的新特性

- `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，你需要使用具名参数语法：

```
// 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();
}
```

1.5.30 的新特性

```
// 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 AnnWithValue {  
    String[] value();  
}
```

```
// Kotlin  
@AnnWithValue("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  
}
```

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

1.5.30 的新特性

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().it
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("org.jetbrains.kotlin:kotlin-reflect:1.6.10")  
}
```

【Groovy】

```
dependencies {  
    implementation "org.jetbrains.kotlin:kotlin-reflect:1.6.10"  
}
```

- In Maven:

1.5.30 的新特性

```
<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

1.5.30 的新特性

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)
```

1.5.30 的新特性

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：

1.5.30 的新特性

```
val x = 1

fun main() {
    println(::x.get())
    println(::x.name)
}
```

表达式 `::x` 求值为 `KProperty<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 `KMutableProperty<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：

1.5.30 的新特性

```
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：

1.5.30 的新特性

```
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：

1.5.30 的新特性

```
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
```

多平台开发

- [Kotlin 多平台用于 iOS 与 Android](#)
 - [Kotlin 多平台移动端入门](#)
 - [搭建环境](#)
 - [创建第一个跨平台移动端应用——教程](#)
 - [了解移动端项目结构](#)
 - [让 Android 应用程序能用于 iOS——教程](#)
 - [发布应用程序](#)
- [Kotlin 多平台用于其他平台](#)
 - [Kotlin 多平台入门](#)
 - [了解多平台项目结构](#)
 - [手动设置目标](#)
- [创建多平台库](#)
 - [创建多平台库](#)
 - [发布多平台库](#)
 - [创建并发布多平台库——教程](#)
- [共享代码原则](#)
 - [平台间共享代码](#)
 - [接入平台相关 API](#)
 - [迁移多平台项目到 Kotlin 1.4.0](#)
- [添加依赖项](#)
 - [添加依赖项](#)
 - [添加 Android 依赖项](#)
 - [添加 iOS 依赖项](#)
- [运行测试](#)
- [构件编译项](#)
 - [配置编译项](#)
 - [构建最终原生二进制文件](#)
- [多平台 Gradle DSL 参考](#)
- [样例](#)
- [FAQ](#)
- [向团队介绍跨平台移动端开发](#)

Kotlin 多平台用于 iOS 与 Android

- [Kotlin 多平台移动端入门](#)
- [搭建环境](#)
- [创建第一个跨平台移动端应用——教程](#)
- [了解移动端项目结构](#)
- [让 Android 应用程序能用于 iOS——教程](#)
- [发布应用程序](#)

Kotlin 多平台移动端入门

Kotlin Multiplatform Mobile is in [Alpha](#). Language features and tooling may change in future Kotlin versions.

Beta is expected in spring 2022. Check out [Kotlin Multiplatform Mobile Beta Roadmap Video Highlights](#) to learn about upcoming features. You can also see how [different companies](#) already use Kotlin for cross-platform app development.



Kotlin Multiplatform Mobile (KMM) is an SDK designed to simplify the development of cross-platform mobile applications. You can share common code between iOS and Android apps and write platform-specific code only where it's necessary. For example, to implement a native UI or when working with platform-specific APIs.

Watch the introductory [video](#), in which Kotlin Product Marketing Manager Ekaterina Petrova explains what Kotlin Multiplatform Mobile is and how to use it in your project. With Ekaterina, you'll set up an environment and prepare to create your first cross-platform mobile application:

YouTube 视频: [Meet Kotlin Multiplatform Mobile!](#)

You can also check out other videos about [Kotlin Multiplatform Multiverse](#) on YouTube.

Supported platforms

- Android applications and libraries
- [Android NDK](#) (ARM64 and ARM32)
- Apple iOS devices (ARM64 and ARM32) and simulators
- Apple watchOS devices (ARM64 and ARM32) and simulators

[Kotlin Multiplatform](#) technology also supports [other platforms](#), including JavaScript, Linux, Windows, and WebAssembly.

Start from scratch

1.5.30 的新特性

- Set up the environment for cross-platform mobile development
- Create your first app that works both on Android and iOS with IDE
- Check out the list of sample projects
- Introduce cross-platform mobile development to your team

Make your Android application work on iOS

If you already have an Android mobile application and want to make it cross-platform, here are some resources to help you get started:

- Set up the environment for cross-platform development
- Make a sample Android application work well on iOS

Get help

- **Kotlin Slack:** Get an [invite](#) and join the `#multiplatform` channel
- **StackOverflow:** Subscribe to the “[kotlin-multiplatform](#)” tag
- **Kotlin issue tracker:** [Report a new issue](#)

搭建环境

Before you begin [creating your first application](#) to work on both iOS and Android, start by setting up an environment for Kotlin Multiplatform Mobile development:

1. If you are going to work with shared code or Android-specific code, you can work on any computer with an operating system supported by [Android Studio](#).
If you also want to write iOS-specific code and run an iOS application on a simulated or real device, use a Mac with a macOS. These steps cannot be performed on other operating systems, such as Microsoft Windows. This is due to an Apple requirement.
2. Install [Android Studio](#) 4.2 or 2020.3.1 Canary 8 or higher.
You will use Android Studio for creating your multiplatform applications and running them on simulated or hardware devices.
3. If you need to write iOS-specific code and run an iOS application, install [Xcode](#) – version 11.3 or higher.
Most of the time, Xcode will work in the background. You will use it to add Swift or Objective-C code to your iOS application.
4. Make sure that you have a [compatible Kotlin plugin](#) installed.
In Android Studio, select **Tools | Kotlin | Configure Kotlin Plugin Updates** and check the current Kotlin plugin version. If needed, update to the latest version in the **Stable** update channel.
5. Install the *Kotlin Multiplatform Mobile* plugin.
In Android Studio, select **Preferences | Plugins**, search for the plugin *Kotlin Multiplatform Mobile* in **Marketplace** and install it.



Kotlin Multiplatform Mobile

↓ 34.9K ☆ 4.86 JetBrains

Languages

[Plugin homepage ↗](#)

The Kotlin Multiplatform Mobile (KMM) plugin helps you develop applications that work on both Android and iOS. With the KMM Plugin for Android Studio, you can:

- Write business logic just once and share the code on both platforms.
- Run and debug the iOS part of your application on iOS targets straight from Android Studio.
- Quickly create a new multiplatform project, or add a multiplatform module into an existing one.

[Release notes ↗](#)

[Issue tracker ↗](#)

[More about KMM ↗](#)

Check out [Kotlin Multiplatform Mobile plugin release notes](#).

6. Install the [JDK](#) if you haven't already done so.

To check if it's installed, run the command `java -version` in the Terminal.

Now it's time to [create your first cross-platform mobile application](#).

创建第一个跨平台移动端应用——教程

Here you will learn how to create and run your first Kotlin Multiplatform Mobile application.

1. [Set up an environment](#) for cross-platform mobile development by installing the necessary tools on a suitable operating system.

You will need a Mac with macOS to complete certain steps in this tutorial, which include writing iOS-specific code and running an iOS application. These steps cannot be performed on other operating systems, such as Microsoft Windows. This is due to an Apple requirement.

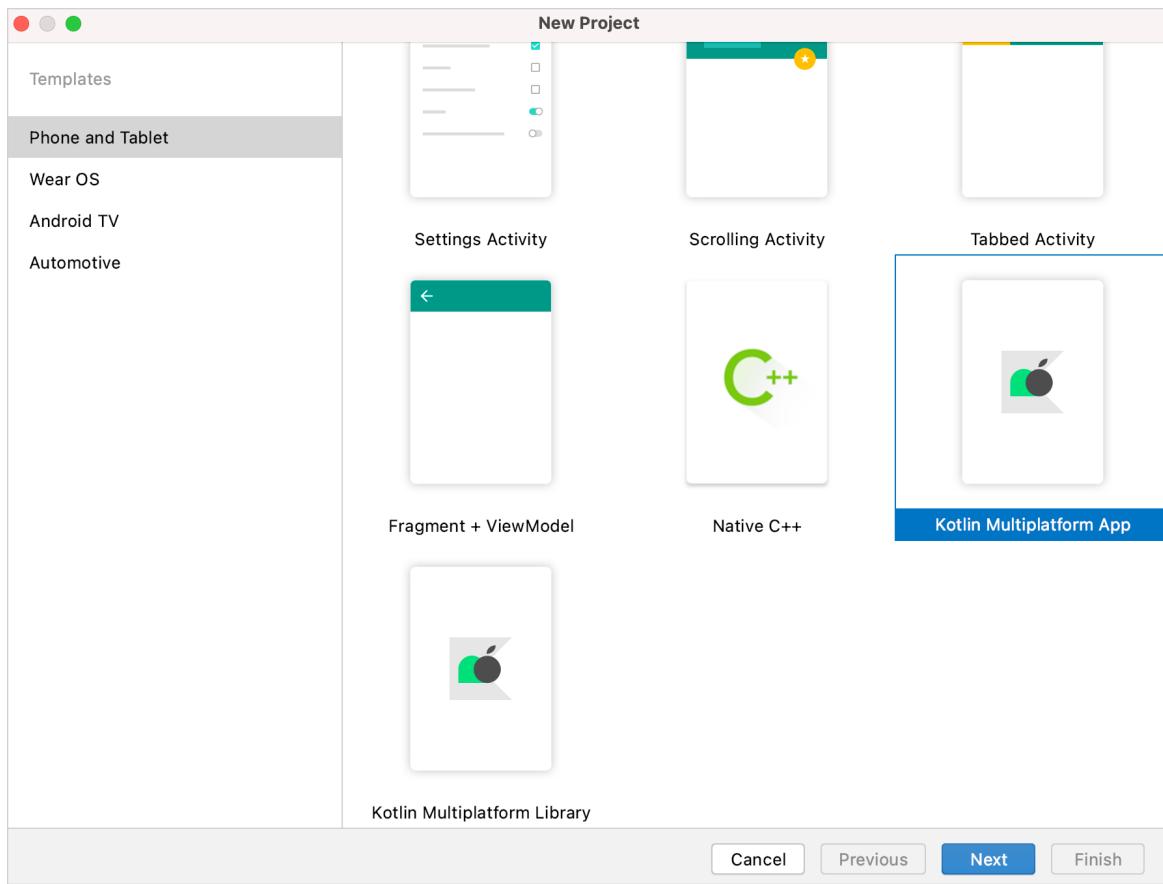


You can also watch a video of this tutorial created by Ekaterina Petrova, Kotlin Developer Advocate.

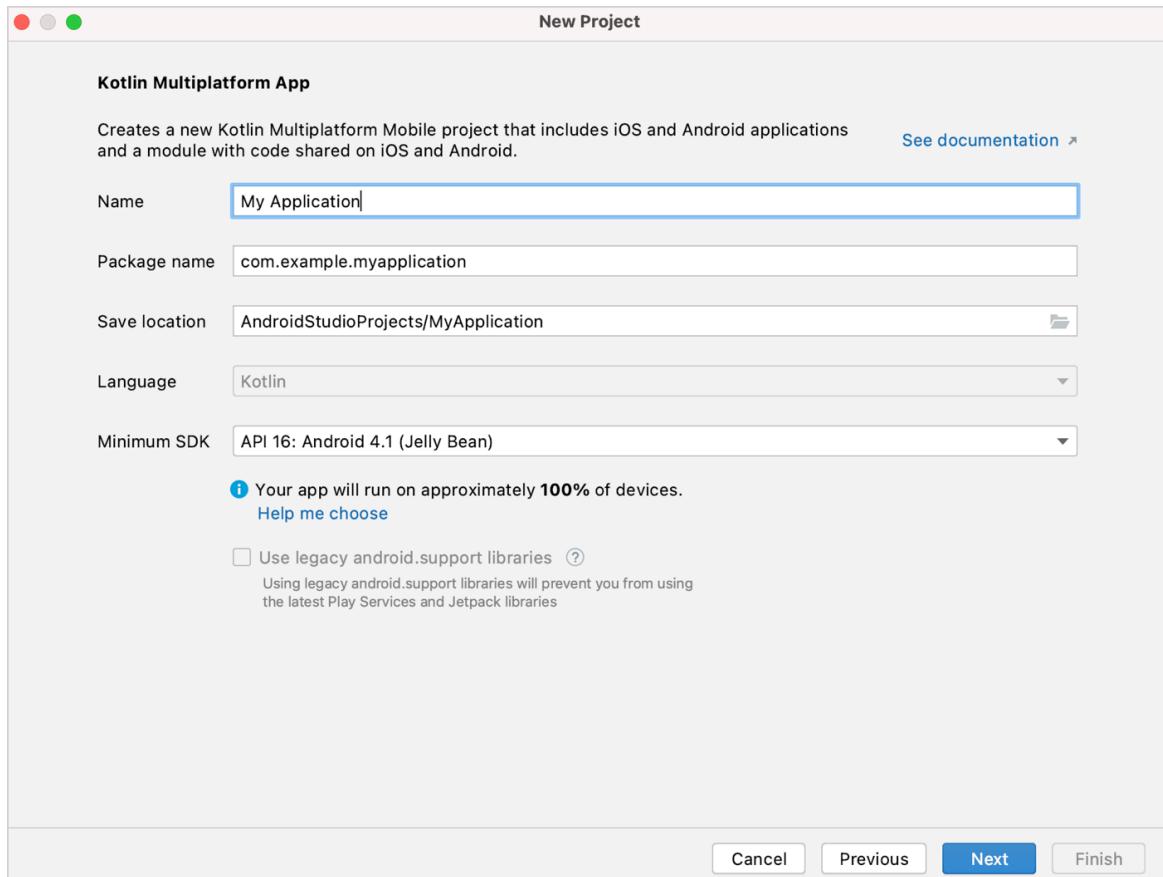
YouTube 视频: [Kotlin Multiplatform Multiverse, Episode 2: Your First Kotlin Multiplatform Mobile App Tutorial](#)

1. In Android Studio, select **File | New | New Project**.
2. Select **Kotlin Multiplatform App** in the list of project templates, and click **Next**.

1.5.30 的新特性



3. Specify a name for your first application, and click **Next**.

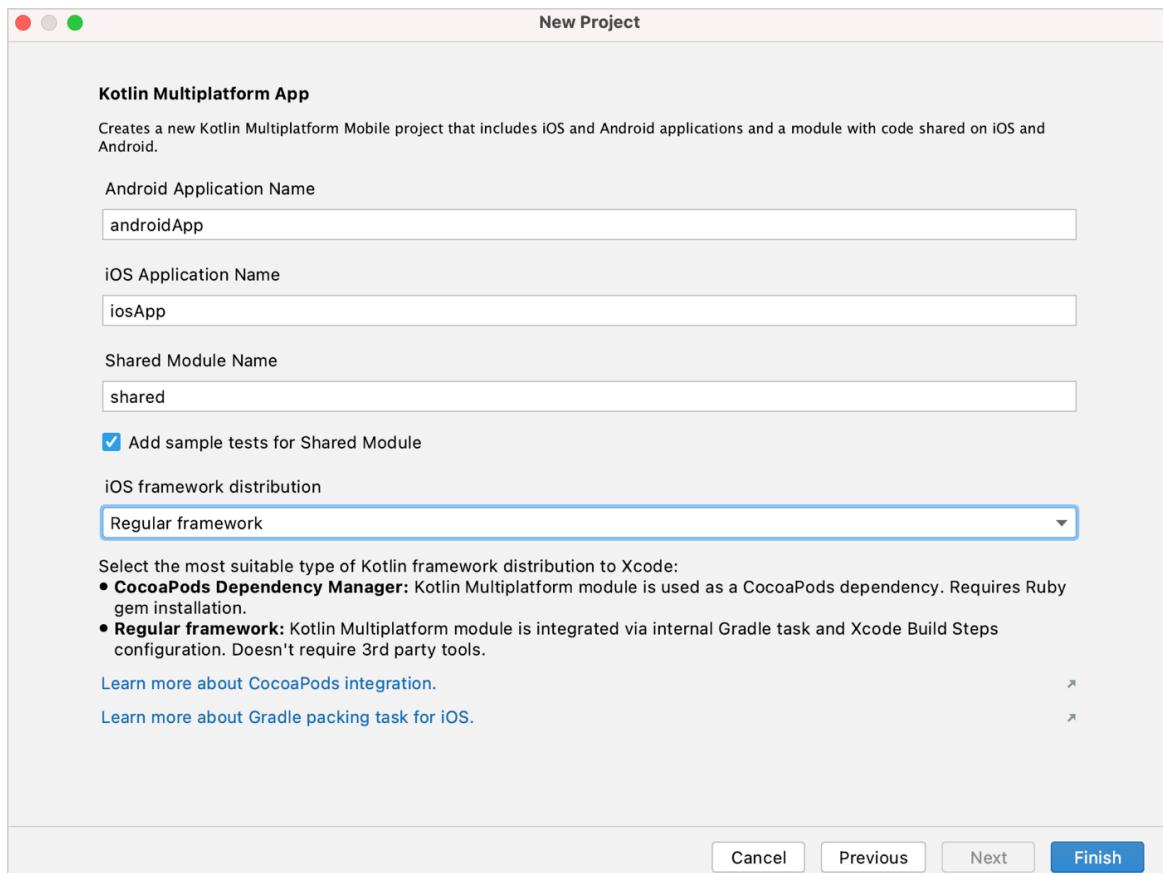


4. In the window that opens, do the following:

1.5.30 的新特性

- Keep the default names for the application and shared folders.
- Select the checkbox to generate sample tests for your project.
- Select **Regular framework** in the list of iOS framework distribution options.

Click **Finish** to create a new project.

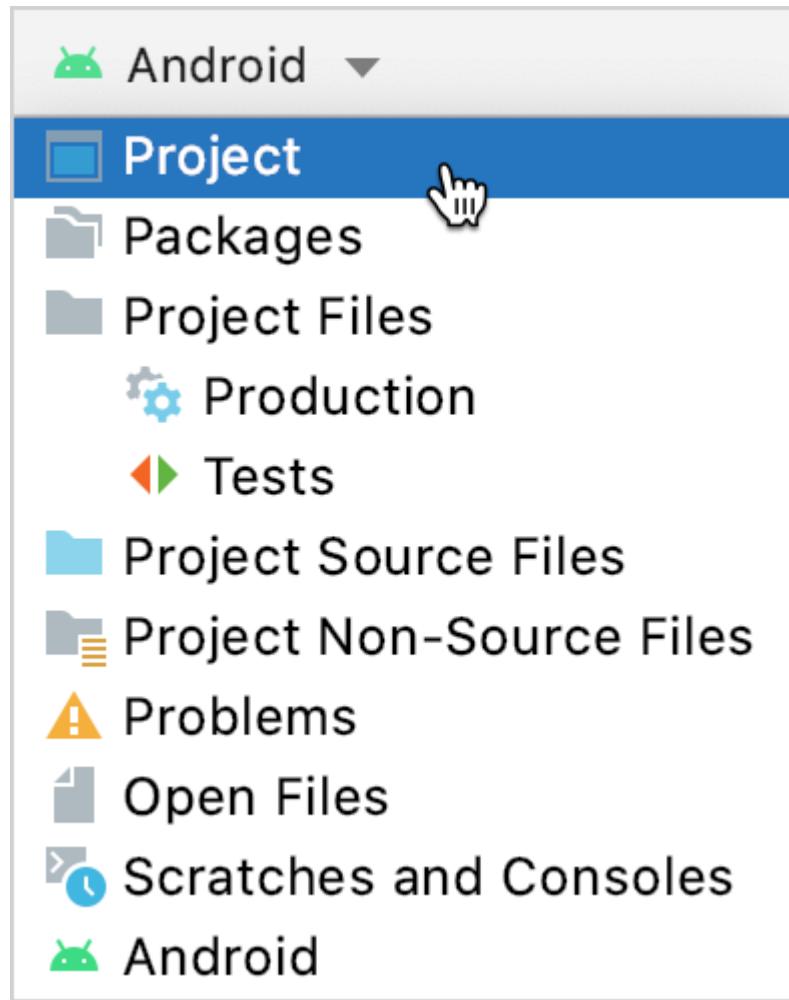


If you want to use Kotlin Multiplatform module as a CocoaPods dependency, select the **CocoaPods dependency manager** option. To learn more about CocoaPods dependencies, see [CocoaPods integration](#).



Now wait while your project is set up. It may take some time to download and set up the required components when you do this for the first time.

To view the complete structure of your mobile multiplatform project, switch the view from **Android to Project**. You can [understand the project structure](#) and how you can use this.

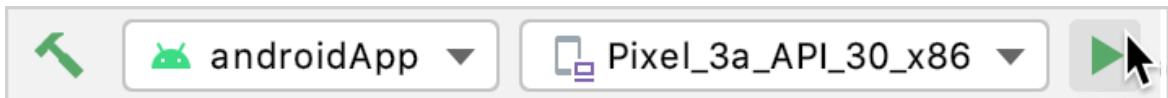


Run your application

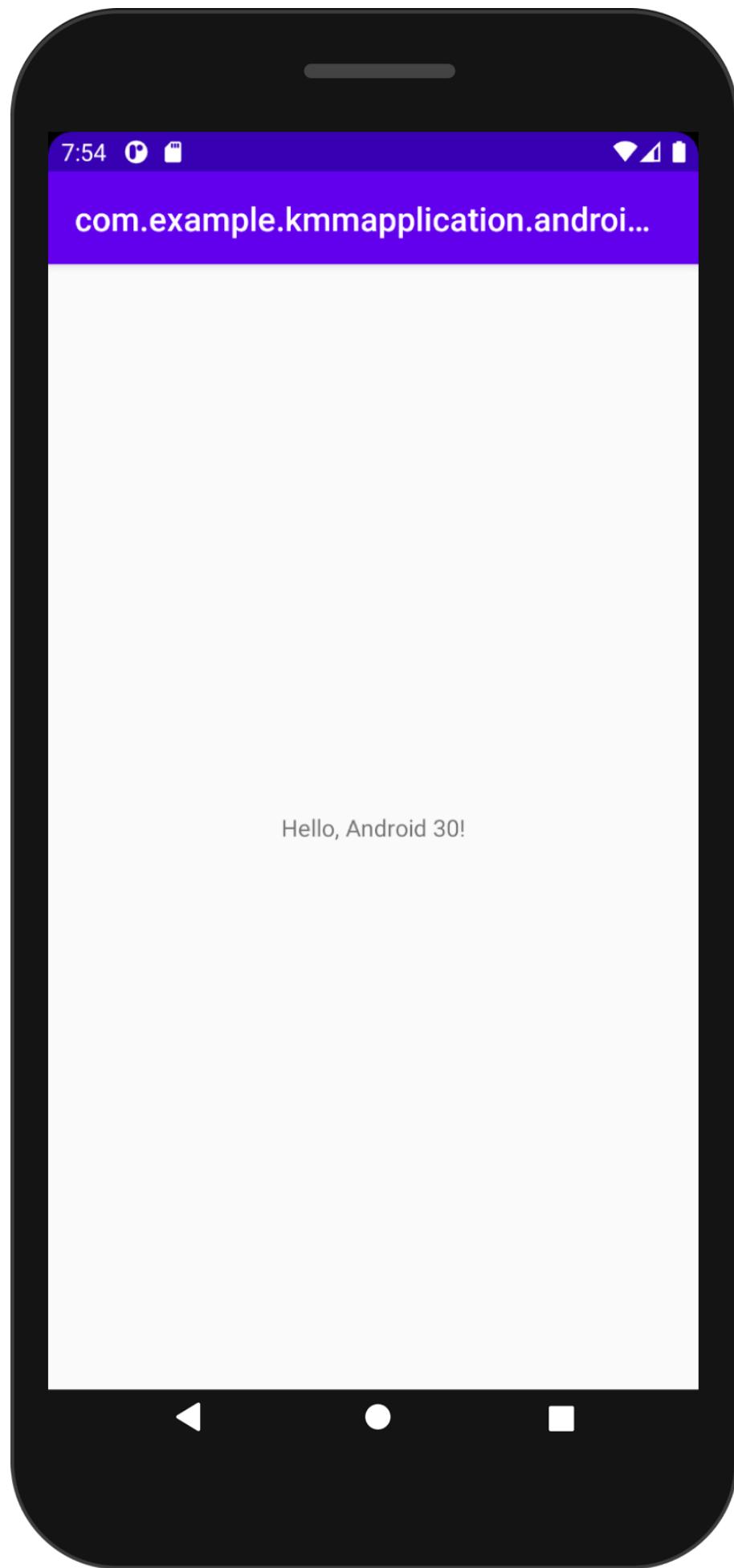
You can run your multiplatform application on [Android](#) or [iOS](#).

Run your application on Android

- In the list of run configurations, select **androidApp** and then click **Run**.



1.5.30 的新特性



Run on a different Android simulated device

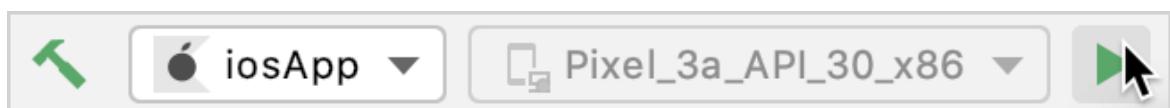
Learn how to [configure the Android Emulator](#) and run your application on a different simulated device.

Run on a real Android device

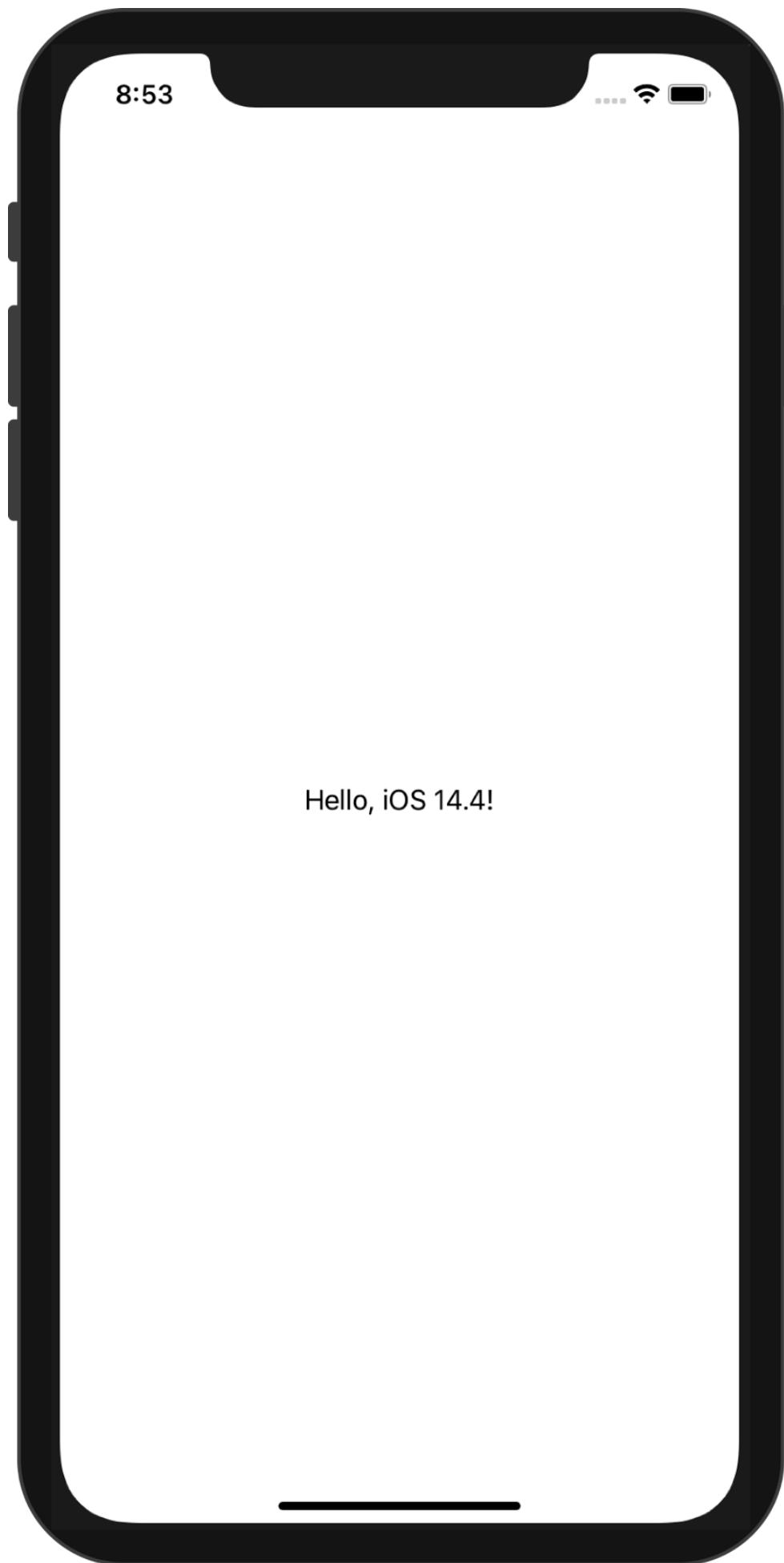
Learn how to [configure and connect a hardware device](#) and run your application on it.

Run your application on iOS

- In the list of run configurations, select **iosApp** and then click **Run**.



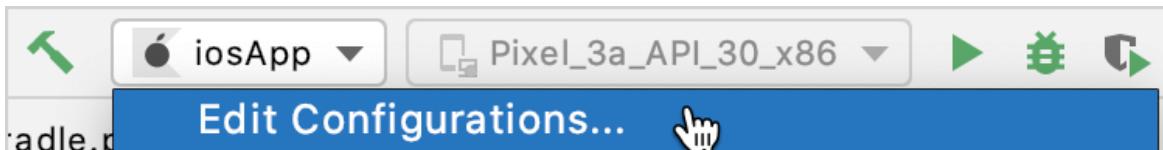
1.5.30 的新特性



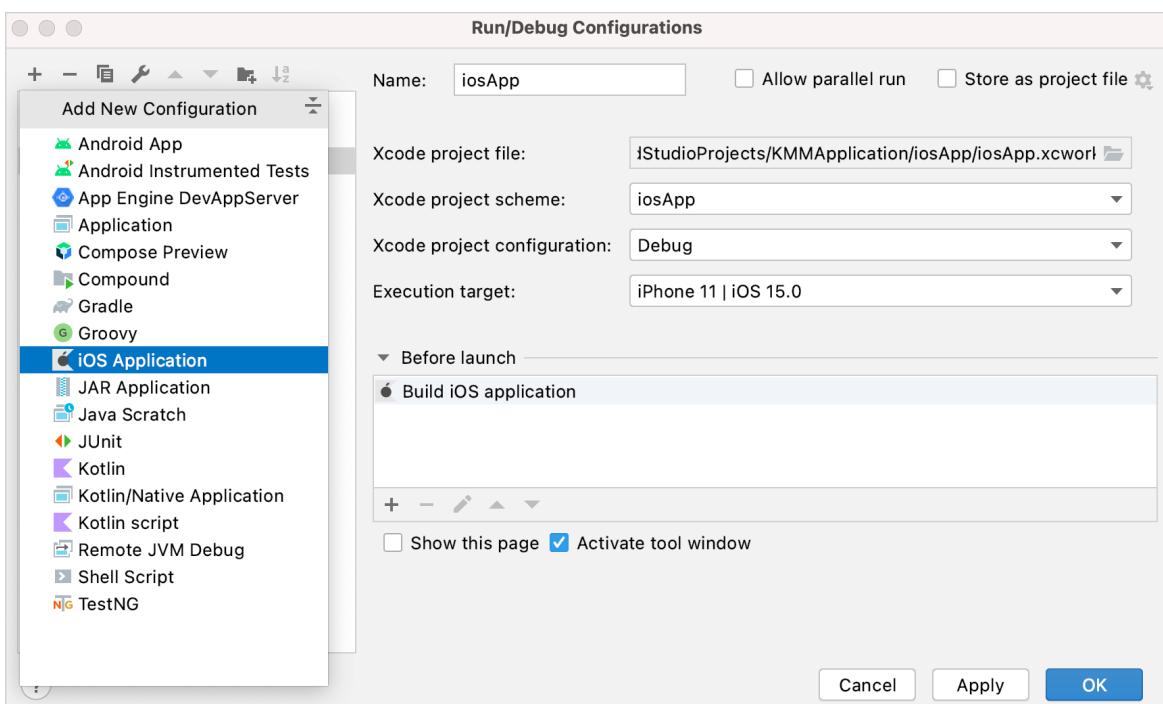
Run on a different iPhone simulated device

If you want to run your application on another simulated device, you can add a new run configuration.

1. In the list of run configurations, click **Edit Configurations**.



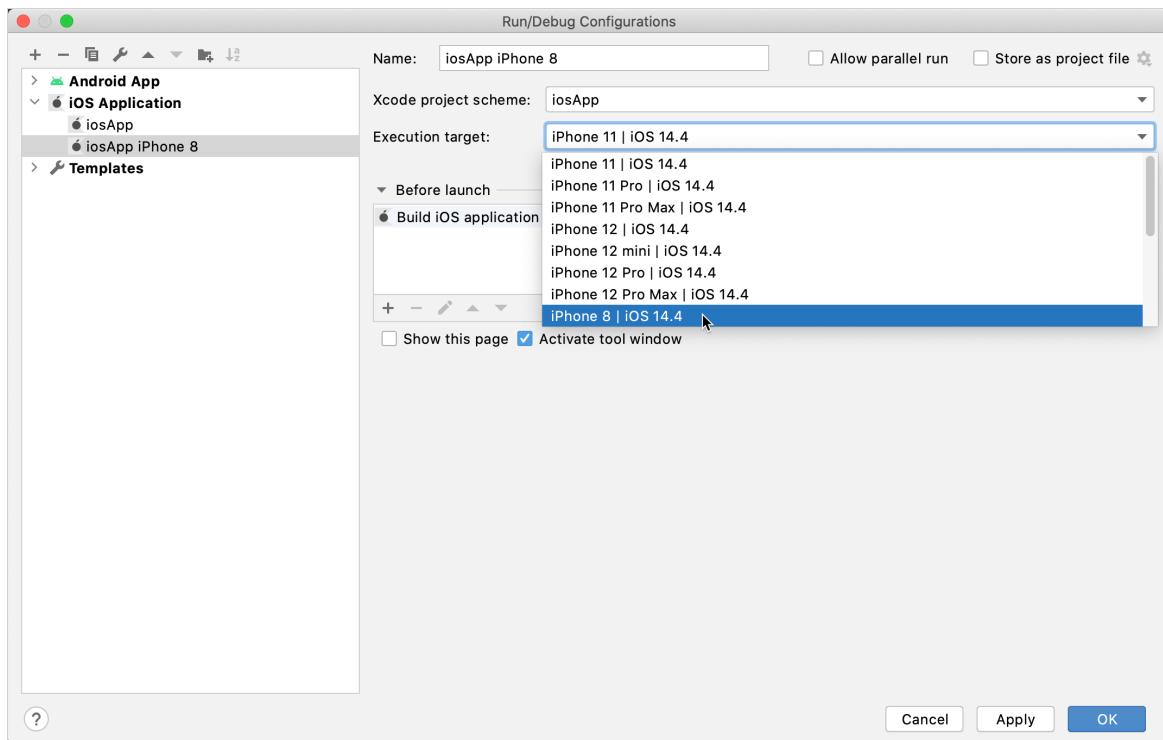
2. Click the + button above the list of configurations and select **iOS Application**.



3. Name your configuration.

4. Select a simulated device in the **Execution target** list, and then click **OK**.

1.5.30 的新特性



5. Click **Run** to run your application on the new simulated device.

Run on a real iPhone device

1. [Connect a real iPhone device to Xcode](#).
2. [Create a run configuration](#) by selecting iPhone in the **Execution target** list.
3. Click **Run** to run your application on the iPhone device.

If your build fails, follow the workaround described in [this issue](#).



Run tests

You can run tests to check that the shared code works correctly on both platforms. Of course, you can also write and run tests to check the platform-specific code.

Run tests on iOS

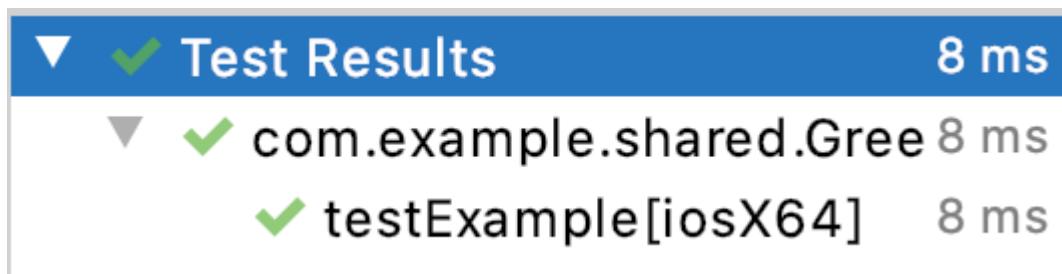
1. Open the file `iosTest.kt` in
`shared/src/iosTest/kotlin/com.example.kmmapplication.shared`.
Directories with `Test` in their name contain tests.
This file includes a sample test for iOS.

1.5.30 的新特性



2. Click the **Run** icon in the gutter next to the test.

Tests run on a simulator without UI. Congratulations! The test has passed – see test results in the console.



Run tests on Android

For Android, follow a procedure that is very similar to the one for running tests on iOS.

1. Open the file `androidTest.kt` in

`shared/src/androidTest/kotlin/com.example.kmmapplication.shared`.

2. Click the **Run** gutter icon next to the test.

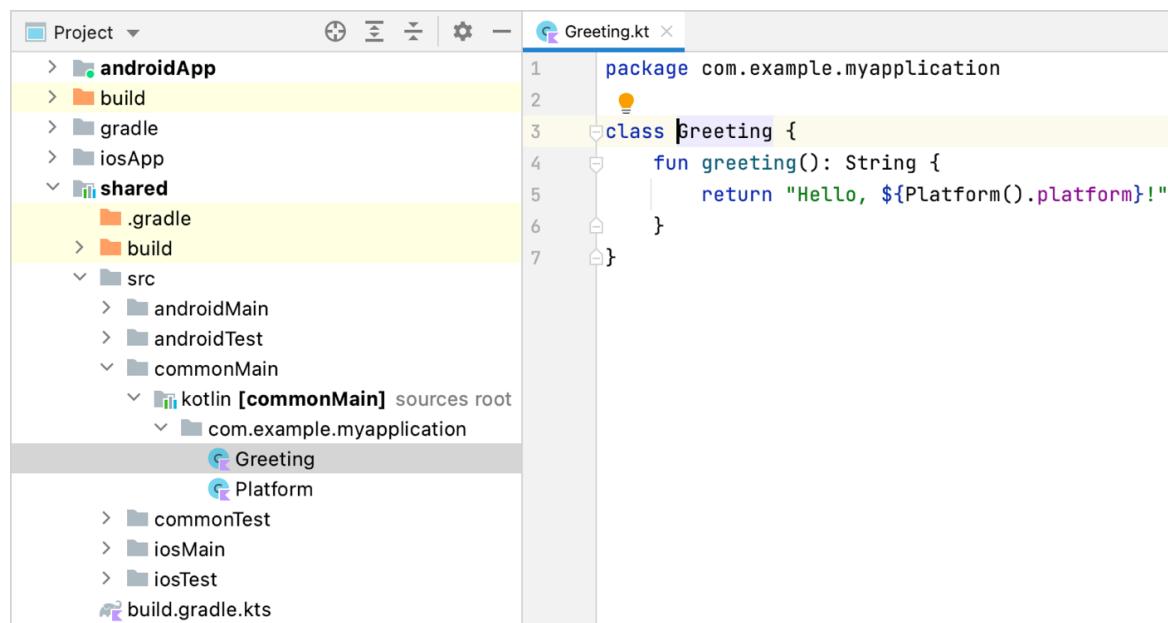
Update your application

1. Open the file `Greeting.kt` in

`shared/src/commonMain/kotlin/com.example.kmmapplication.shared`.

This directory stores the shared code for both platforms – Android and iOS. If you make changes to the shared code, you will see changes in both applications.

1.5.30 的新特性



2. Update the shared code – use the Kotlin standard library function that works on all platforms and reverts text: `reversed()`.

```
class Greeting {
    fun greeting(): String {
        return "Guess what it is! > ${Platform().platform.reversed()}!"
    }
}
```

3. Run the updated application on Android.

1.5.30 的新特性



1.5.30 的新特性

4. Run the updated application on iOS.

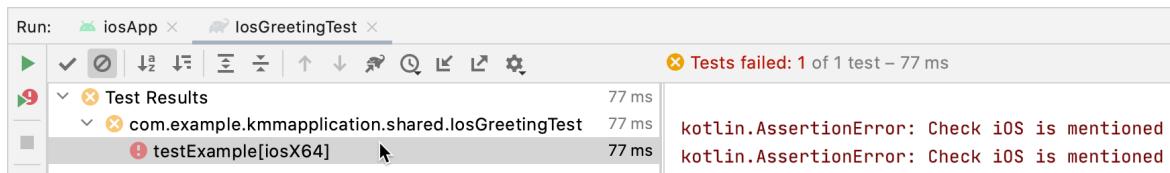
1.5.30 的新特性



1.5.30 的新特性

5. Run tests on Android and iOS.

As you see, the tests fail. Update the tests to pass. You know how to do this, right? ;)



Next steps

Once you've played with your first cross-platform mobile application, you can:

- [Understand the project structure](#)
- [Complete a tutorial on making your Android application work on iOS](#)

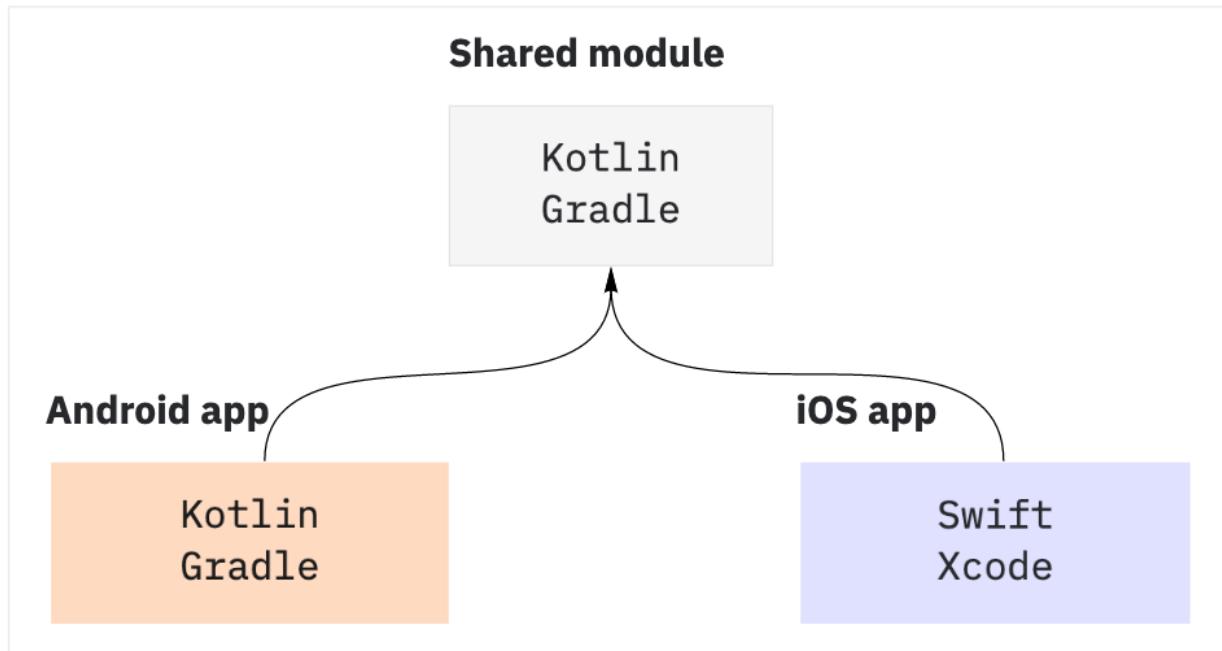
了解移动端项目结构

The purpose of the Kotlin Multiplatform Mobile technology is unifying the development of applications with common logic for Android and iOS platforms. To make this possible, it uses a mobile-specific structure of [Kotlin Multiplatform](#) projects. This page describes the structure of a basic cross-platform mobile project. Note that this structure isn't the only possible way to organize your project; however, we recommend it as a starting point.

A basic Kotlin Mobile Multiplatform project consists of three components:

- *Shared module* – a Kotlin module that contains common logic for both Android and iOS applications. Builds into an Android library and an iOS framework. Uses Gradle as a build system.
- *Android application* – a Kotlin module that builds into the Android application. Uses Gradle as a build system.
- *iOS application* – an Xcode project that builds into the iOS application.

Root project



This is the structure of a Multiplatform Mobile project that you create with a Project Wizard in IntelliJ IDEA or Android Studio. Real-life projects can have more complex structure; we consider these three components essential.

Let's take a closer look at the basic project and its components.

Root project

The root project is a Gradle project that holds the shared module and the Android application as its subprojects. They are linked together via the [Gradle multi-project mechanism](#).

【Kotlin】

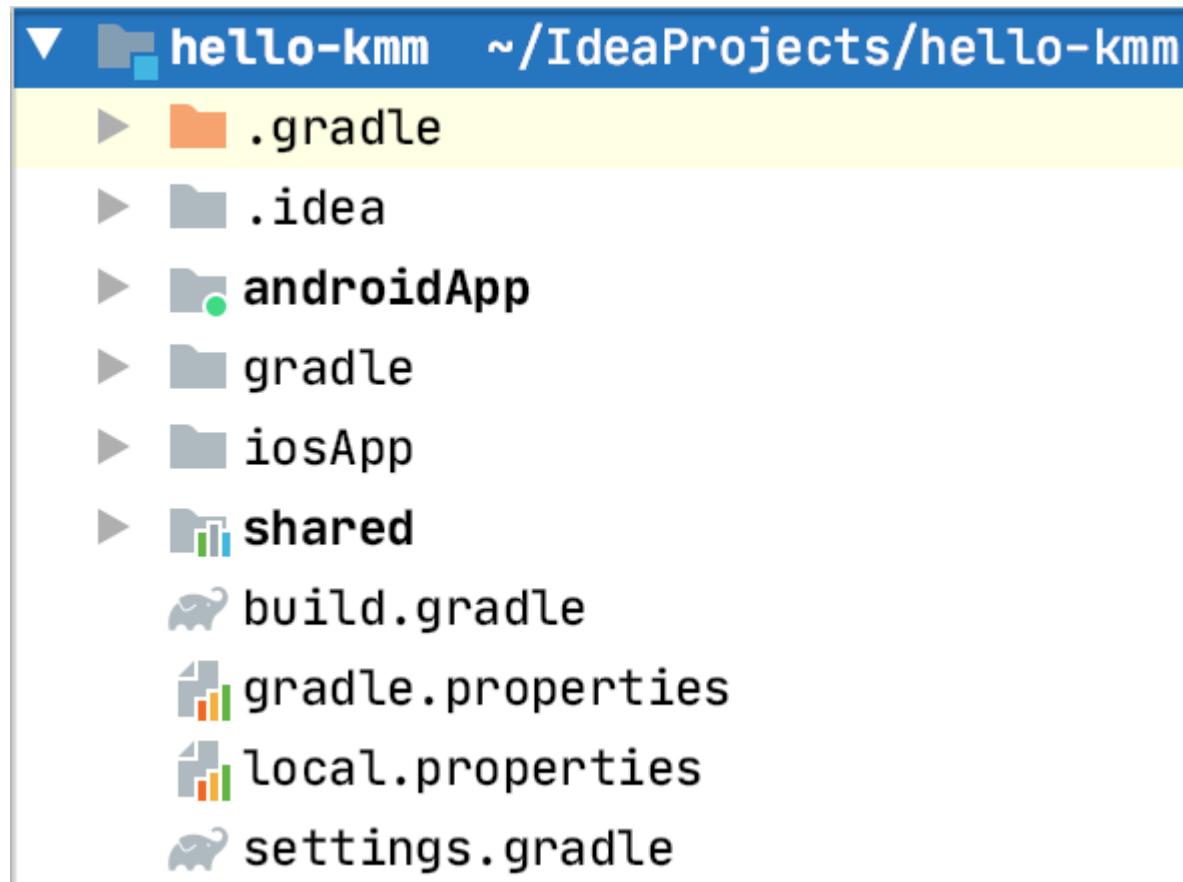
```
// settings.gradle.kts
include(":shared")
include(":androidApp")
```

【Groovy】

```
// settings.gradle
include ':shared'
include ':androidApp'
```

The iOS application is produced from an Xcode project. It's stored in a separate directory within the root project. Xcode uses its own build system; thus, the iOS application project isn't connected with other parts of the Multiplatform Mobile project via Gradle. Instead, it uses the shared module as an external artifact – framework. For details on integration between the shared module and the iOS application, see [iOS application](#).

This is a basic structure of a cross-platform mobile project:



The root project does not hold source code. You can use it to store global configuration in its `build.gradle(.kts)` or `gradle.properties`, for example, add repositories or define global configuration variables.

For more complex projects, you can add more modules into the root project by creating them in the IDE and linking via `include` declarations in the Gradle settings.

Shared module

Shared module contains the core application logic used in both target platforms: classes, functions, and so on. This is a [Kotlin Multiplatform](#) module that compiles into an Android library and an iOS framework. It uses Gradle with the Kotlin Multiplatform plugin applied and has targets for Android and iOS.

【Kotlin】

1.5.30 的新特性

```
plugins {
    kotlin("multiplatform") version "1.6.10"
    // ...
}

kotlin {
    android()
    ios()
}
```

【Groovy】

```
plugins {
    id 'org.jetbrains.kotlin.multiplatform' version '1.6.10'
    // ...
}

kotlin {
    android()
    ios()
}
```

Source sets

The shared module contains the code that is common for Android and iOS applications. However, to implement the same logic on Android and iOS, you sometimes need to write two platform-specific versions of it. To handle such cases, Kotlin offers the [expect/actual](#) mechanism. The source code of the shared module is organized in three source sets accordingly:

- `commonMain` stores the code that works on both platforms, including the `expect` declarations
- `androidMain` stores Android-specific parts, including `actual` implementations
- `iosMain` stores iOS-specific parts, including `actual` implementations

Each source set has its own dependencies. Kotlin standard library is added automatically to all source sets, you don't need to declare it in the build script.

【Kotlin】

1.5.30 的新特性

```
kotlin {  
    sourceSets {  
        val commonMain by getting  
        val androidMain by getting {  
            dependencies {  
                implementation("androidx.core:core-ktx:1.2.0")  
            }  
        }  
        val iosMain by getting  
        // ...  
    }  
}
```

【Groovy】

```
kotlin {  
    sourceSets {  
        commonMain {  
        }  
        androidMain {  
            dependencies {  
                implementation 'androidx.core:core-ktx:1.2.0'  
            }  
        }  
        iosMain {  
        }  
  
        // ...  
    }  
}
```

When you write your code, add the dependencies you need to the corresponding source sets. Read [Multiplatform documentation on adding dependencies](#) for more information.

Along with `*Main` source sets, there are three matching test source sets:

- `commonTest`
- `androidTest`
- `iosTest`

Use them to store unit tests for common and platform-specific source sets accordingly. By default, they have dependencies on Kotlin test library, providing you with means for Kotlin unit testing: annotations, assertion functions and other. You can add dependencies on other test libraries you need.

1.5.30 的新特性

【Kotlin】

```
kotlin {  
    sourceSets {  
        // ...  
        val commonTest by getting {  
            dependencies {  
                implementation(kotlin("test-common"))  
                implementation(kotlin("test-annotations-common"))  
            }  
        }  
        val androidTest by getting  
        val iosTest by getting  
    }  
  
}
```

【Groovy】

```
kotlin {  
    sourceSets {  
        // ...  
  
        commonTest {  
            dependencies {  
                implementation kotlin('test-common')  
                implementation kotlin('test-annotations-common')  
            }  
        }  
        androidTest {  
  
        }  
        iosTest {  
  
        }  
    }  
}
```

The main and test source sets described above are default. The Kotlin Multiplatform plugin generates them automatically upon target creation. In your project, you can add more source sets for specific purposes. For more information, see [Multiplatform DSL reference](#).

Android library

1.5.30 的新特性

The configuration of the Android library produced from the shared module is typical for Android projects. To learn about Android libraries creation, see [Create an Android library](#) in the Android developer documentation.

To produce the Android library, two more Gradle plugins are used in addition to Kotlin Multiplatform:

- Android library
- Kotlin Android extensions

【Kotlin】

```
plugins {  
    // ...  
    id("com.android.library")  
    id("kotlin-android-extensions")  
}
```

【Groovy】

```
plugins {  
    // ...  
    id 'com.android.library'  
    id 'kotlin-android-extensions'  
}
```

The configuration of Android library is stored in the `android {}` top-level block of the shared module's build script:

【Kotlin】

```
android {  
    compileSdk = 29  
    defaultConfig {  
        minSdk = 24  
        targetSdk = 29  
        versionCode = 1  
        versionName = "1.0"  
    }  
    buildTypes {  
        getByName("release") {  
            isMinifyEnabled = false  
        }  
    }  
}
```

1.5.30 的新特性

【Groovy】

```
android {  
    compileSdk 29  
    defaultConfig {  
        minSdk 24  
        targetSdk 29  
        versionCode 1  
        versionName '1.0'  
    }  
    buildTypes {  
        release {  
            minifyEnabled false  
        }  
    }  
}
```

It's typical for any Android project. You can edit it to suit your needs. To learn more, see the [Android developer documentation](#).

iOS framework

For using in iOS applications, the shared module compiles into a framework – a kind of hierarchical directory with shared resources used on the Apple platforms. This framework connects to the Xcode project that builds into an iOS application.

The framework is produced via the [Kotlin/Native](#) compiler. The framework configuration is stored in the `ios {}` block of the build script within `kotlin {}`. It defines the output type `framework` and the string identifier `baseName` that is used to form the name of the output artifact. Its default value matches the Gradle module name. For a real project, it's likely that you'll need a more complex configuration of the framework production. For details, see [Multiplatform documentation](#).

【Kotlin】

1.5.30 的新特性

```
kotlin {  
    // ...  
    ios {  
        binaries {  
            framework {  
                baseName = "shared"  
            }  
        }  
    }  
}
```

【Groovy】

```
kotlin {  
    // ...  
    ios {  
        binaries {  
            framework {  
                baseName = 'shared'  
            }  
        }  
    }  
}
```

Additionally, there is a Gradle task `embedAndSignAppleFrameworkForXcode` that exposes the framework to the Xcode project from which the iOS application is built. It uses the configuration of the iOS application project to define the build mode (`debug` or `release`) and provide the appropriate framework version to the specified location.

The task is built-in in the multiplatform plugin. It executes upon each build of the Xcode project to provide the latest version of the framework for the iOS application. For details, see [iOS application](#).

Android application

The Android application part of a Multiplatform Mobile project is a typical Android application written in Kotlin. In a basic cross-platform mobile project, it uses three Gradle plugins:

- Kotlin Android
- Android Application
- Kotlin Android Extensions

1.5.30 的新特性

【Kotlin】

```
plugins {  
    id("com.android.application")  
    kotlin("android")  
    id("kotlin-android-extensions")  
}
```

【Groovy】

```
plugins {  
    id 'com.android.application'  
    id 'org.jetbrains.kotlin.android'  
    id 'kotlin-android-extensions'  
}
```

To access the shared module code, the Android application uses it as a project dependency.

【Kotlin】

```
dependencies {  
    implementation(project(":shared"))  
    //..  
}
```

【Groovy】

```
dependencies {  
    implementation project(':shared')  
    //..  
}
```

Besides this dependency, the Android application uses the Kotlin standard library (which is added automatically) and some common Android dependencies:

【Kotlin】

1.5.30 的新特性

```
dependencies {  
    //...  
    implementation("androidx.core:core-ktx:1.2.0")  
    implementation("androidx.appcompat:appcompat:1.1.0")  
    implementation("androidx.constraintlayout:constraintlayout:1.1.3")  
}
```

【Groovy】

```
dependencies {  
    //...  
    implementation 'androidx.core:core-ktx:1.2.0'  
    implementation 'androidx.appcompat:appcompat:1.1.0'  
    implementation 'androidx.constraintlayout:constraintlayout:1.1.3'  
}
```

Add your project's Android-specific dependencies to this block. The build configuration of the Android application is located in the `android {}` top-level block of the build script:

【Kotlin】

```
android {  
    compileSdk = 29  
    defaultConfig {  
        applicationId = "org.example.androidApp"  
        minSdk = 24  
        targetSdk = 29  
        versionCode = 1  
        versionName = "1.0"  
    }  
    buildTypes {  
        getByName("release") {  
            isMinifyEnabled = false  
        }  
    }  
}
```

【Groovy】

1.5.30 的新特性

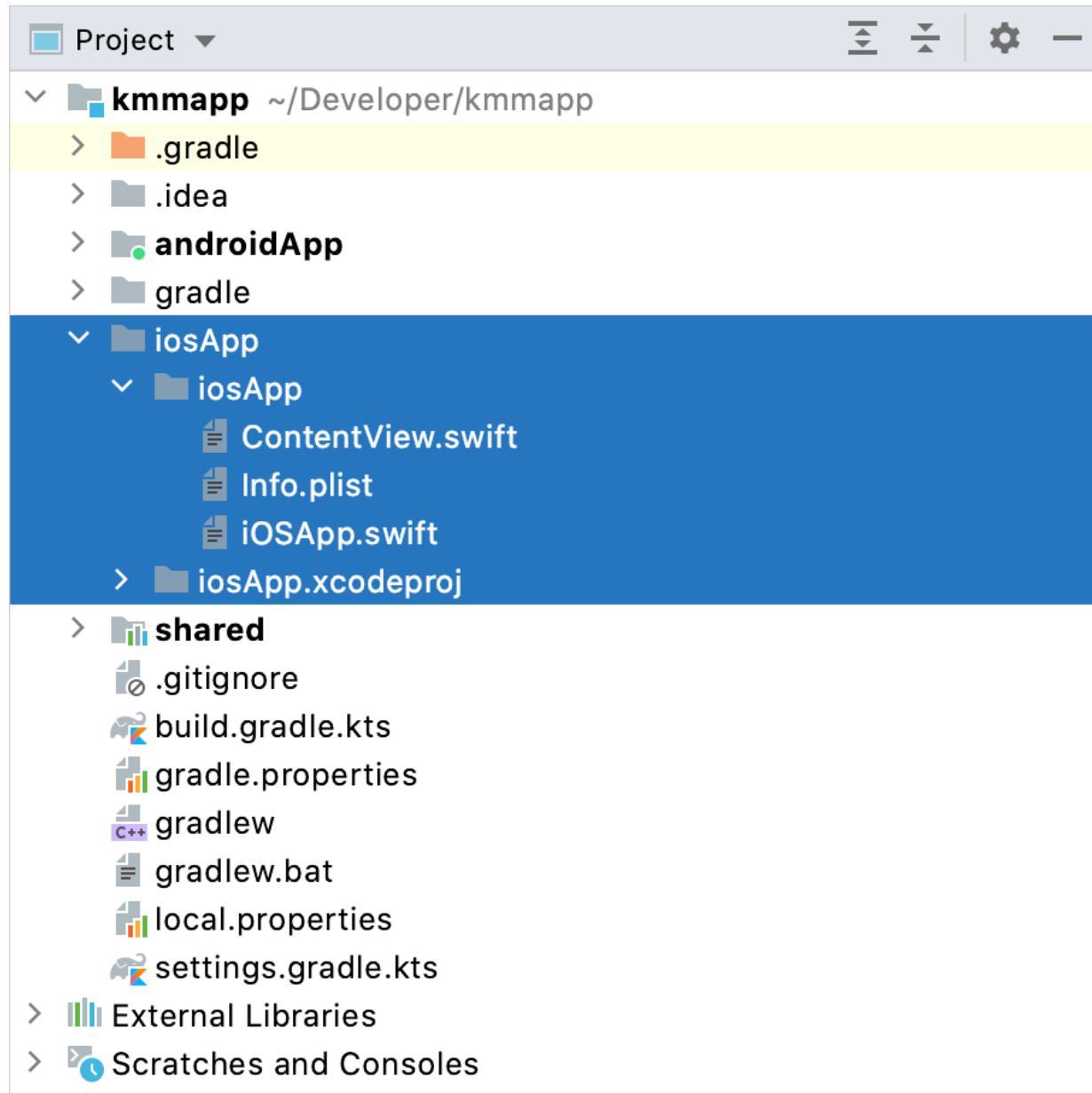
```
android {  
    compileSdk 29  
    defaultConfig {  
        applicationId 'org.example.androidApp'  
        minSdk 24  
        targetSdk 29  
        versionCode 1  
        versionName '1.0'  
    }  
    buildTypes {  
        'release' {  
            minifyEnabled false  
        }  
    }  
}
```

It's typical for any Android project. You can edit it to suit your needs. To learn more, see the [Android developer documentation](#).

iOS application

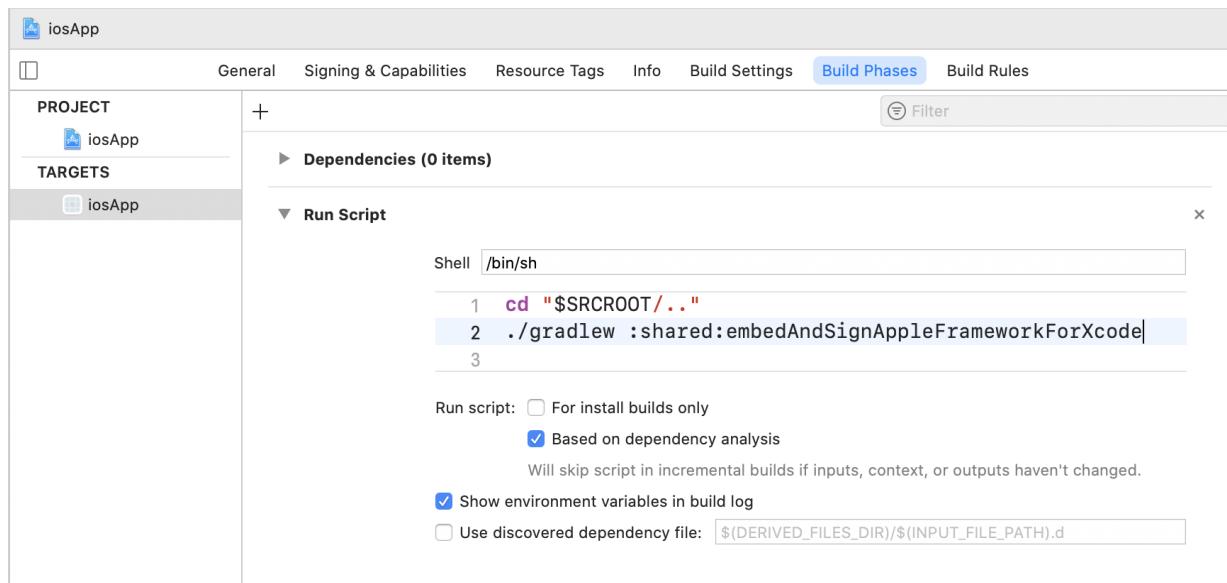
The iOS application is produced from an Xcode project generated automatically by the Project Wizard. It resides in a separate directory within the root project.

1.5.30 的新特性



For each build of the iOS application, the project obtains the latest version of the framework. To do this, it uses a **Run Script** build phase that executes the `embedAndSignAppleFrameworkForXcode` Gradle task from the shared module. This task generates the `.framework` with the needed configuration, depending on the Xcode environment settings, and puts the artifact into the `DerivedData` Xcode directory.

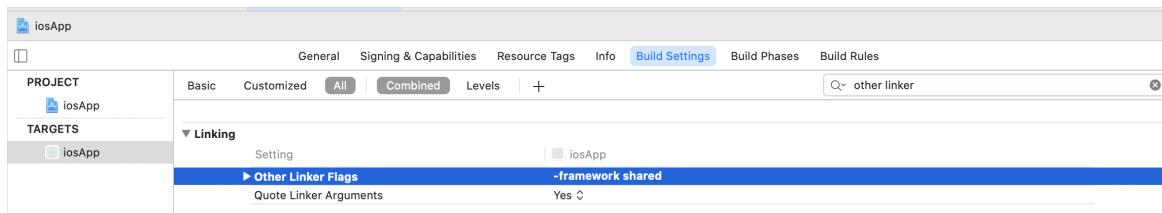
1.5.30 的新特性



To embed framework into the application and make the declarations from the shared module available in the source code of the iOS application, the following build settings should be configured properly:

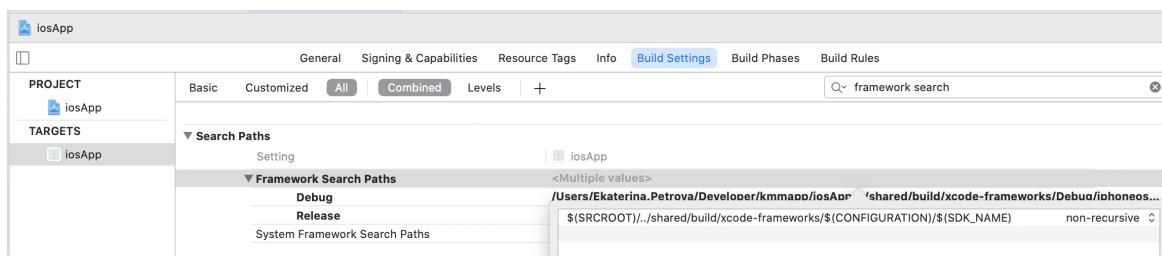
1. Other Linker flags under the Linking section:

```
$(inherited) -framework shared
```



2. Framework Search Paths under the Search Paths section:

```
$(SRCROOT)/../shared/build/xcode-frameworks/$(CONFIGURATION)/$(SDK_NAME)
```



In other aspects, the Xcode part of a cross-platform mobile project is a typical iOS application project. To learn more about creating iOS application, see the [Xcode documentation](#).

让 Android 应用程序能用于 iOS——教程

Here you can learn how to make your existing Android application cross-platform so that it works both on Android and iOS. You'll be able to write code and test it for both Android and iOS only once, in one place.

This tutorial uses a [sample Android application](#) with a single screen for entering a username and password. The credentials are validated and saved to an in-memory database.

If you aren't familiar with Kotlin Multiplatform Mobile, you can learn how to [create and configure a cross-platform mobile application from scratch](#) first.

Prepare an environment for development

1. Install Android Studio 4.2 or Android Studio 2020.3.1 Canary 8 or higher and [other tools for cross-platform mobile development](#) on macOS.

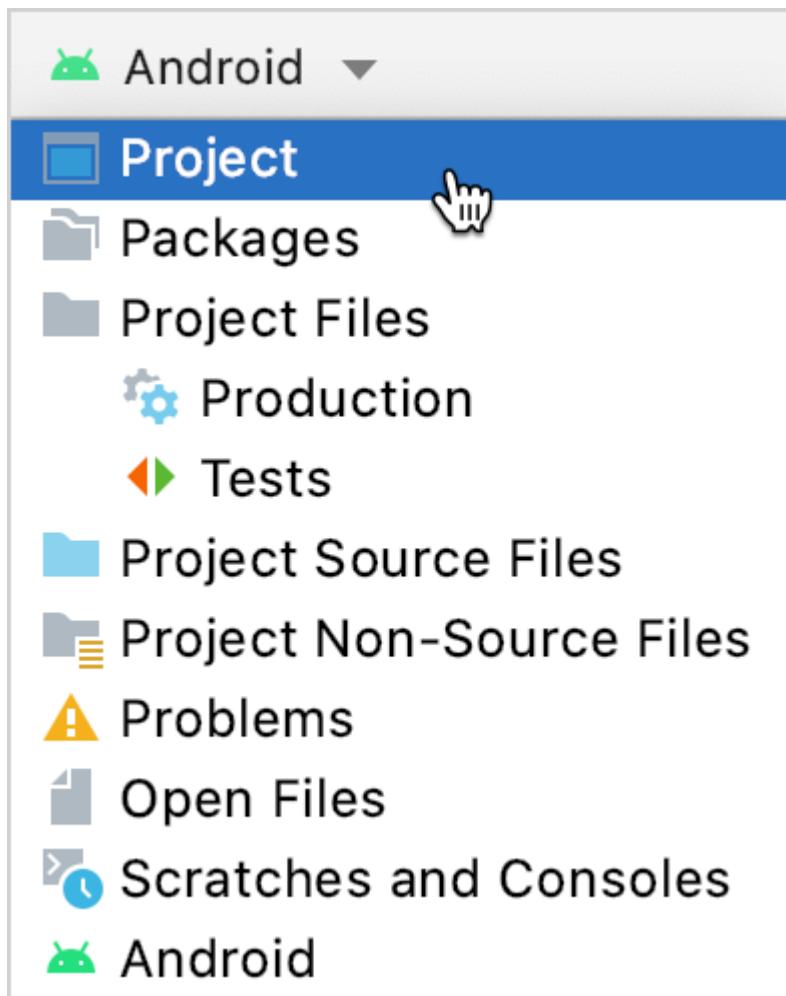
You will need a Mac with macOS to complete certain steps in this tutorial, which include writing iOS-specific code and running an iOS application. These steps cannot be performed on other operating systems, such as Microsoft Windows. This is due to an Apple requirement.



2. In Android Studio, create a new project from version control:

```
https://github.com/Kotlin/kmm-integration-sample.
```

3. Switch to the **Project** view.



Make your code cross-platform

To make your application work on iOS, you'll first make your code cross-platform, and then you'll reuse your cross-platform code in a new iOS application.

To make your code cross-platform:

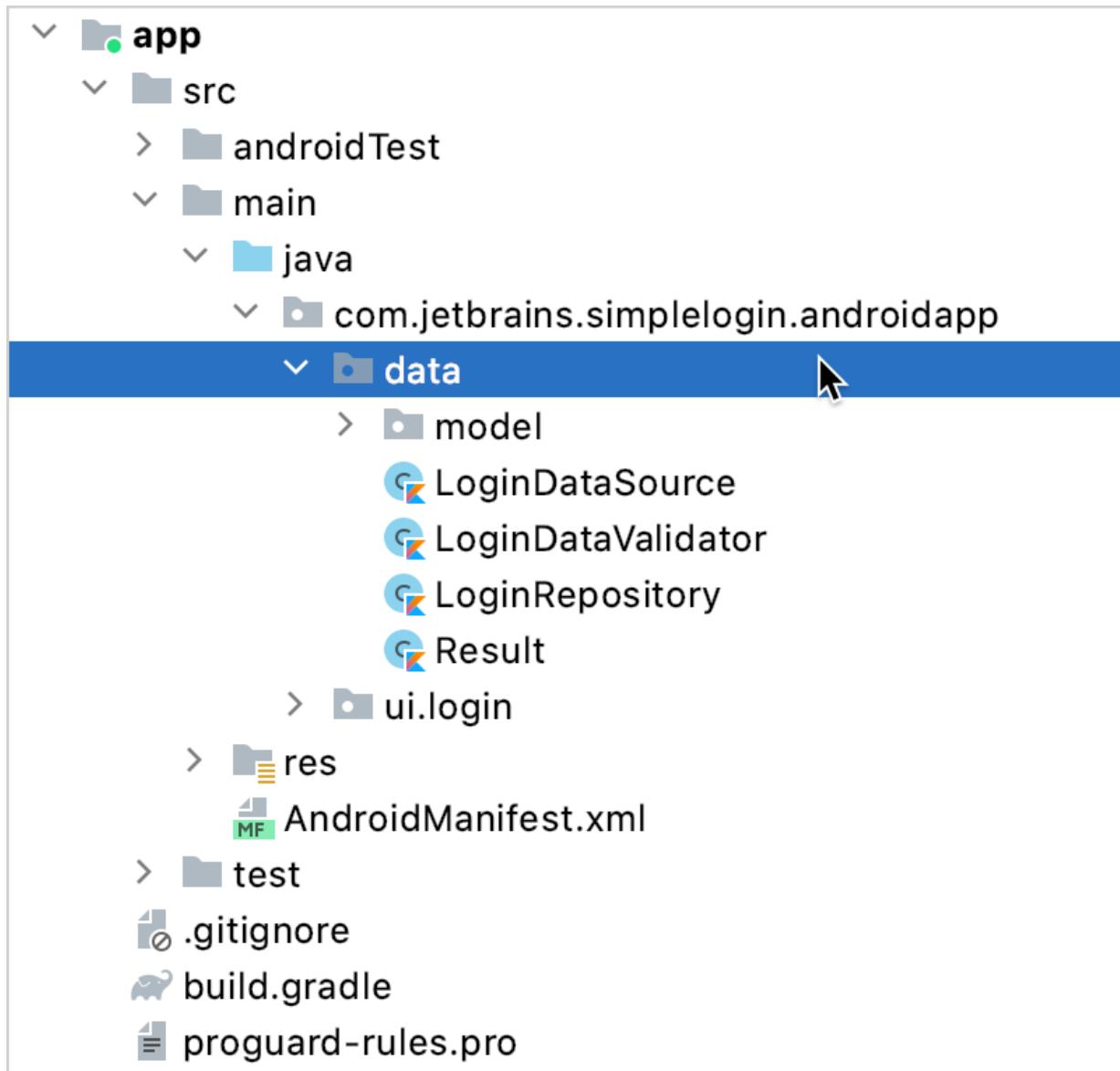
1. [Decide what code to make cross-platform.](#)
2. [Create a shared module for cross-platform code.](#)
3. [Add a dependency on the shared module to your Android application.](#)
4. [Make the business logic cross-platform.](#)
5. [Run your cross-platform application on Android.](#)

Decide what code to make cross-platform

1.5.30 的新特性

Decide which code of your Android application is better to share for iOS and which to keep native. A simple rule is: share what you want to reuse as much as possible. The business logic is often the same for both Android and iOS, so it's a great candidate for reuse.

In your sample Android application, the business logic is stored in the package `com.jetbrains.simplelogin.androidapp.data`. Your future iOS application will use the same logic, so you should make it cross-platform, as well.



Create a shared module for cross-platform code

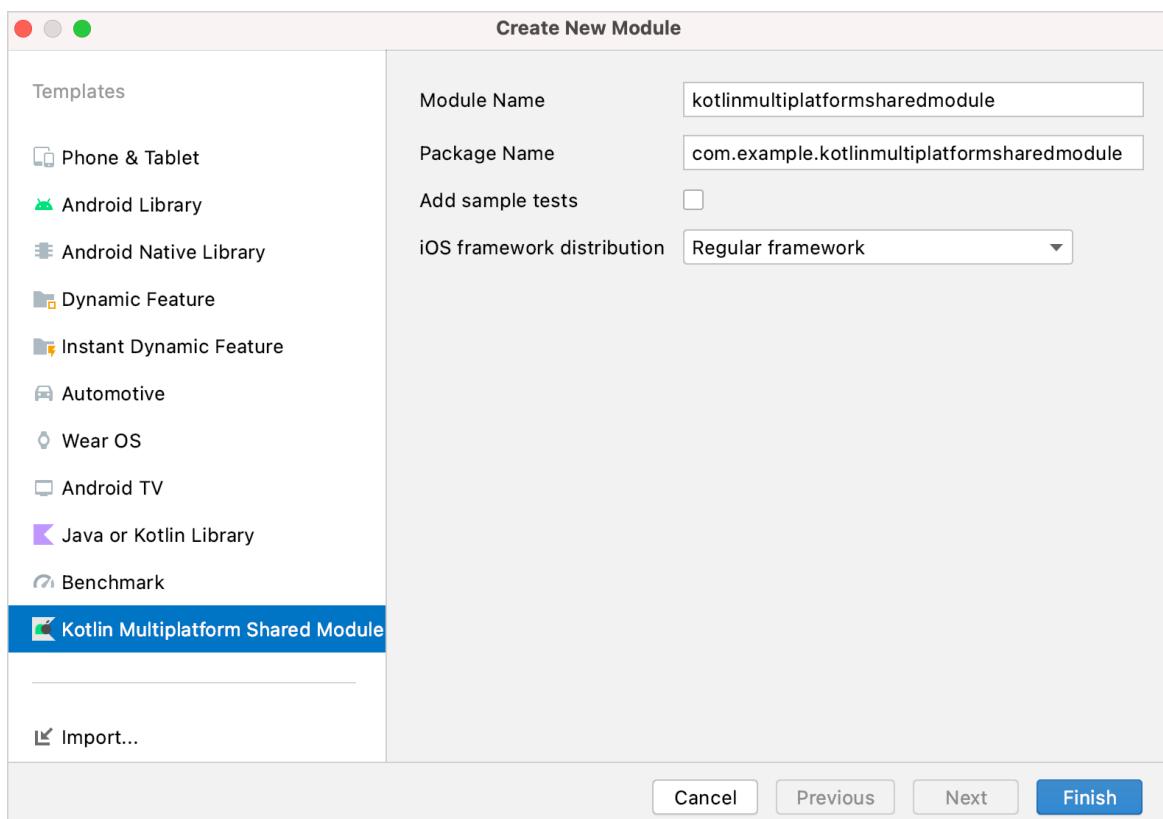
The cross-platform code that is used for both iOS and Android *is stored* in the shared module. Kotlin Multiplatform provides a special wizard for creating such modules.

1.5.30 的新特性

In your Android project, create a Kotlin Multiplatform shared module for your cross-platform code. Later you'll connect it to your existing Android application and your future iOS application.

1. In Android Studio, click **File | New | New Module**.
2. In the list of templates, select **Kotlin Multiplatform Shared Module**, enter the module name `shared`, and select the **Regular framework** in the list of iOS framework distribution options.

This is required for connecting the shared module to the iOS application.



3. Click **Finish**.

The wizard will create the Kotlin Multiplatform shared module, update the configuration files, and create files with classes that demonstrate the benefits of Kotlin Multiplatform. You can learn more about the [project structure](#).

Add a dependency on the shared module to your Android application

To use cross-platform code in your Android application, connect the shared module to it, move the business logic code there, and make this code cross-platform.

1.5.30 的新特性

1. Ensure that `compileSdkVersion` and `minSdkVersion` in `build.gradle.kts` of the `shared` module are the same as those in the `build.gradle` of your Android application in the `app` module.
If they are different, update them in the `build.gradle.kts` of the shared module. Otherwise, you'll encounter a compile error.
2. Add a dependency on the shared module to the `build.gradle` of your Android application.

```
dependencies {  
    implementation project(':shared')  
}
```

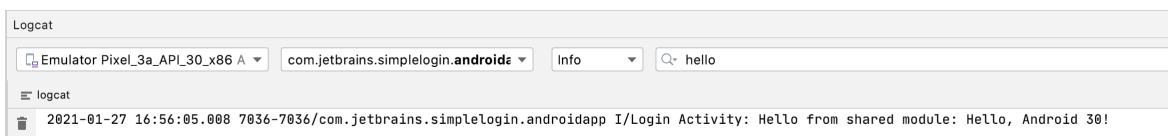
3. Synchronize the Gradle files by clicking **Sync Now** in the warning.

Gradle files have changed since last project sync. A project syn... [Sync Now](#) [Ignore these changes](#)

4. To make sure that the shared module is successfully connected to your application, dump the `greeting()` function result to the log by updating the `onCreate()` method of the `LoginActivity` class.

```
override fun onCreate(savedInstanceState: Bundle?) {  
    super.onCreate(savedInstanceState)  
  
    Log.i("Login Activity", "Hello from shared module: " + Greeting().greeting())  
}
```

5. Search for `Hello` in the log, and you'll find the greeting from the shared module.



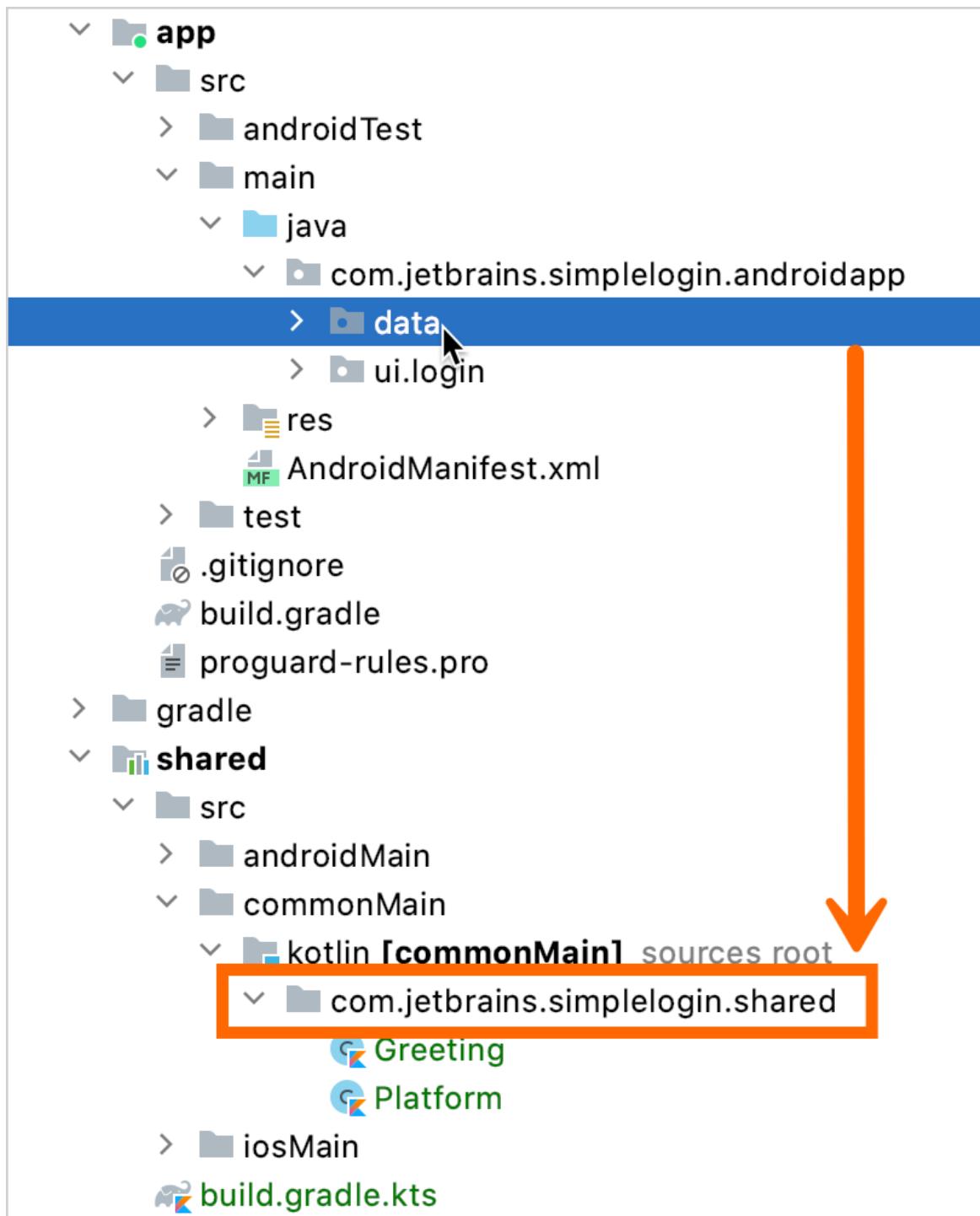
Make the business logic cross-platform

You can now extract the business logic code to the Kotlin Multiplatform shared module and make it platform-independent. This is necessary for reusing the code for both Android and iOS.

1. Move the business logic code `com.jetbrains.simplelogin.androidapp.data` from the `app` directory to the `com.jetbrains.simplelogin.shared` package in the `shared/src/commonMain` directory. You can drag and drop the package or refactor it

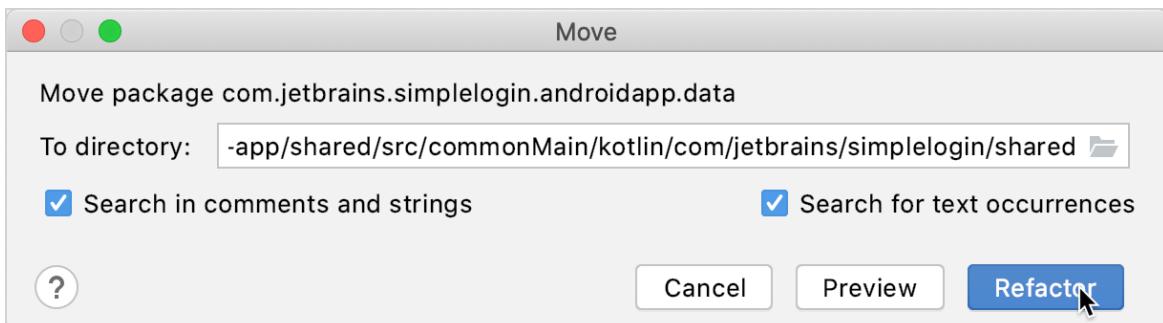
1.5.30 的新特性

by moving everything from one directory to another.

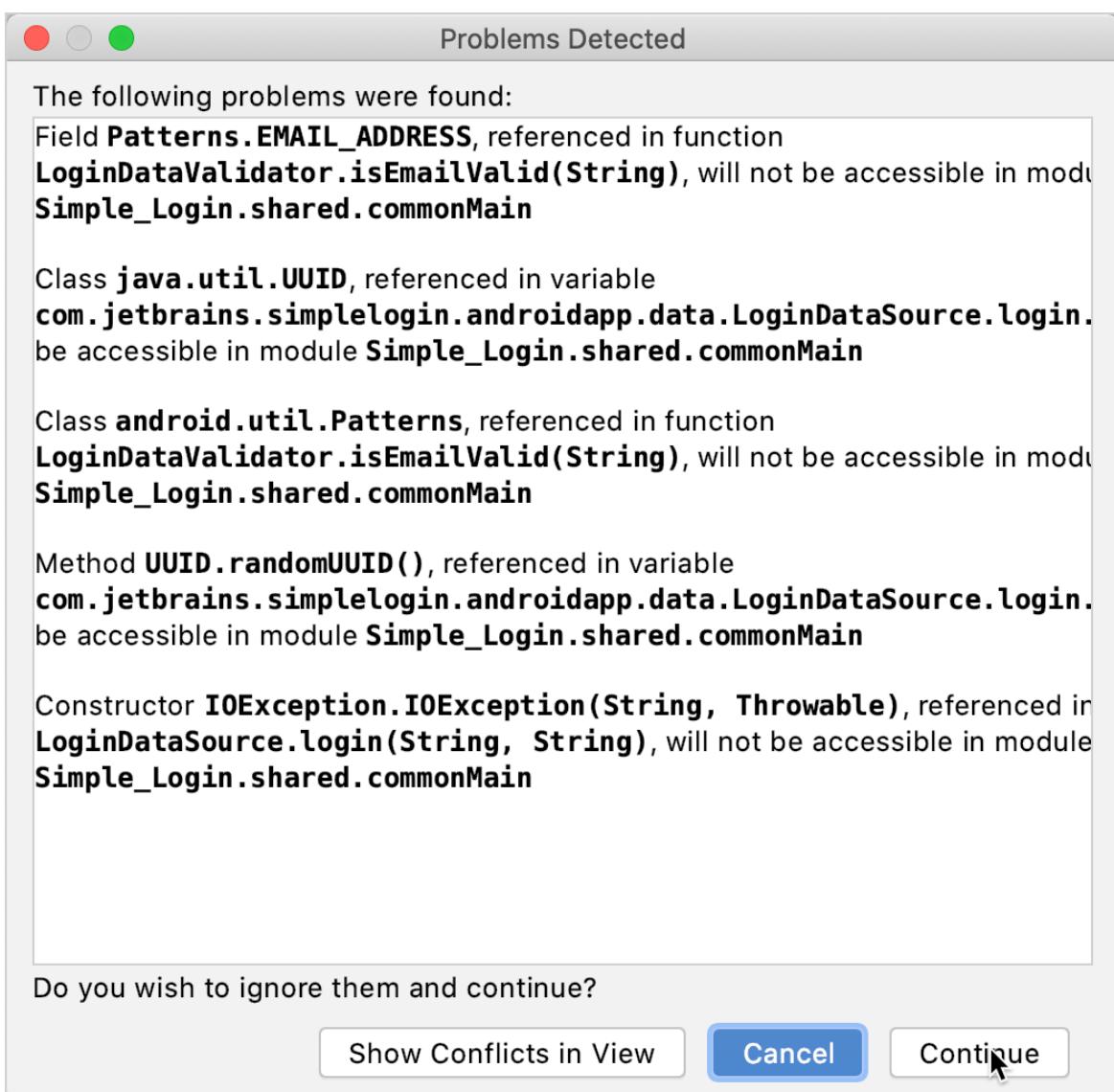


2. When Android Studio asks what you'd like to do, select to move the package, and then approve the refactoring.

1.5.30 的新特性



3. Ignore all warnings about platform-dependent code and click **Continue**.



4. Remove Android-specific code by replacing it with cross-platform Kotlin code or connecting to Android-specific APIs using `expect` and `actual` declarations. See the following sections for details.

Replace Android-specific code with cross-platform code

1.5.30 的新特性

To make your code work well on both Android and iOS, replace all JVM dependencies with Kotlin dependencies wherever possible.

1. In the `login()` function of the `LoginDataSource` class, replace `I0Exception`, which is not available in Kotlin, with `RuntimeException`.

```
// Before
return Result.Error(I0Exception("Error logging in", e))
```

```
// After
return Result.Error(RuntimeException("Error logging in", e))
```

2. For email validation, replace the `Patterns` class from the `android.utils` package with a Kotlin regular expression matching the pattern in the `LoginDataValidator` class:

```
// Before
private fun isValidEmail(email: String) = Patterns.EMAIL_ADDRESS.matcher(email).matches()
```

```
// After
private fun isValidEmail(email: String) = emailRegex.matches(email)

companion object {
    private val emailRegex =
        ("[a-zA-Z0-9\\+\\.\\_\\%\\-\\+]{1,256}" +
         "\\@" +
         "[a-zA-Z0-9] [a-zA-Z0-9\\-]{0,64}" +
         "(" +
         "\\." +
         "[a-zA-Z0-9] [a-zA-Z0-9\\-]{0,25}" +
         ")+").toRegex()
}
```

Connect to platform-specific APIs from the cross-platform code

A universally unique identifier (UUID) for `fakeUser` in `LoginDataSource` is generated using the `java.util.UUID` class, which is not available for iOS.

```
val fakeUser = LoggedInUser(java.util.UUID.randomUUID().toString(), "Jane Doe")
```

1.5.30 的新特性

Since the Kotlin standard library doesn't provide functionality for generating UUIDs, you still need to use platform-specific functionality for this case.

Provide the `expect` declaration for the `randomUUID()` function in the shared code and its `actual` implementations for each platform – Android and iOS – in the corresponding source sets. You can learn more about [connecting to platform-specific APIs](#).

1. Remove the `java.util.UUID` class from the common code:

```
val fakeUser = LoggedInUser(randomUUID(), "Jane Doe")
```

2. Create a `Utils.kt` file in the `shared/src/commonMain` directory and provide the `expect` declaration:

```
package com.jetbrains.simplelogin.shared

expect fun randomUUID(): String
```

3. Create a `Utils.kt` file in the `shared/src/androidMain` directory and provide the `actual` implementation for `randomUUID()` in Android:

```
package com.jetbrains.simplelogin.shared

import java.util.*
actual fun randomUUID() = UUID.randomUUID().toString()
```

4. Create a `Utils.kt` file in the `shared/src/iosMain` directory and provide the `actual` implementation for `randomUUID()` in iOS:

```
package com.jetbrains.simplelogin.shared

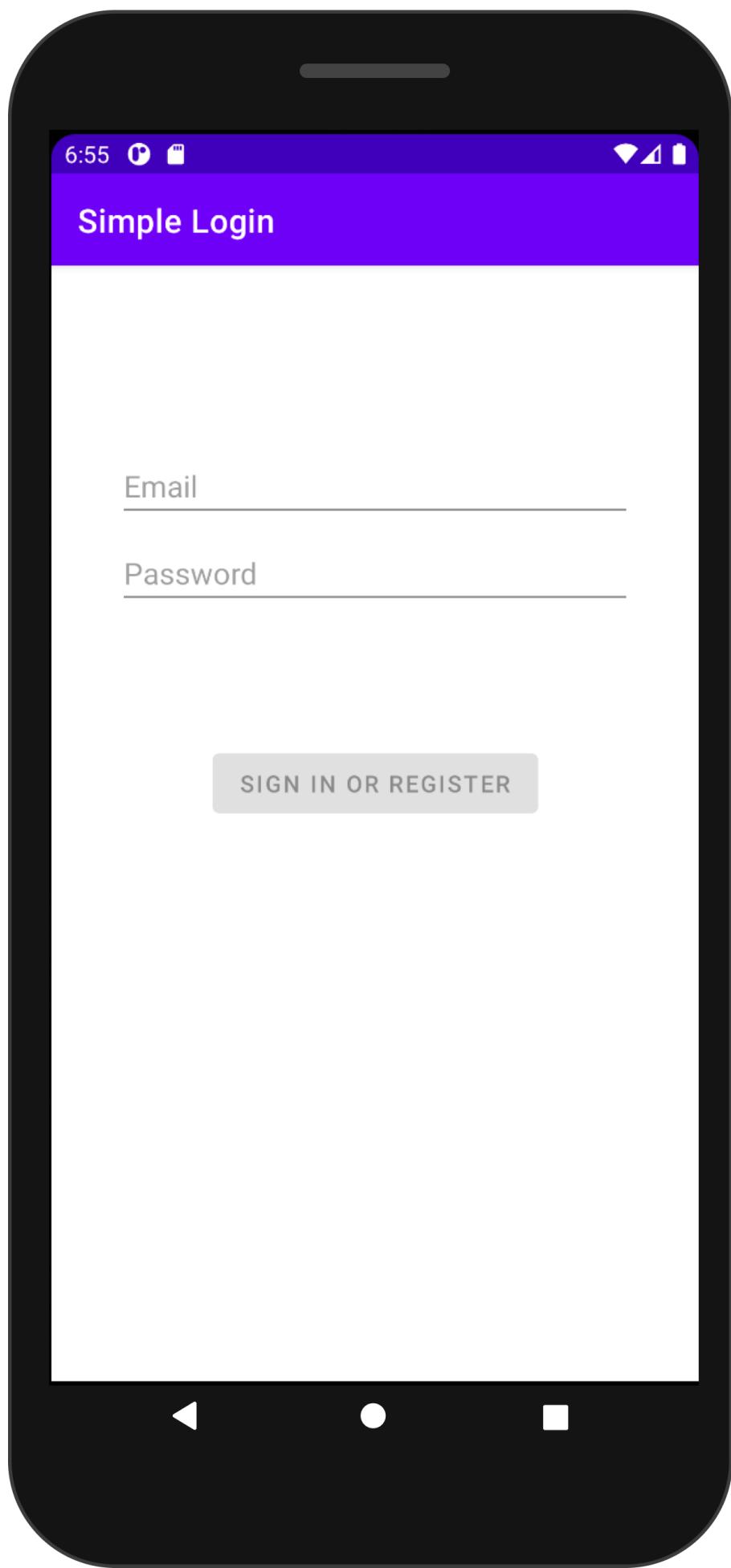
import platform.Foundation.NSUUID
actual fun randomUUID(): String = NSUUID().UUIDString()
```

For Android and iOS, Kotlin will use different platform-specific implementations.

Run your cross-platform application on Android

Run your cross-platform application for Android to make sure it works.

1.5.30 的新特性



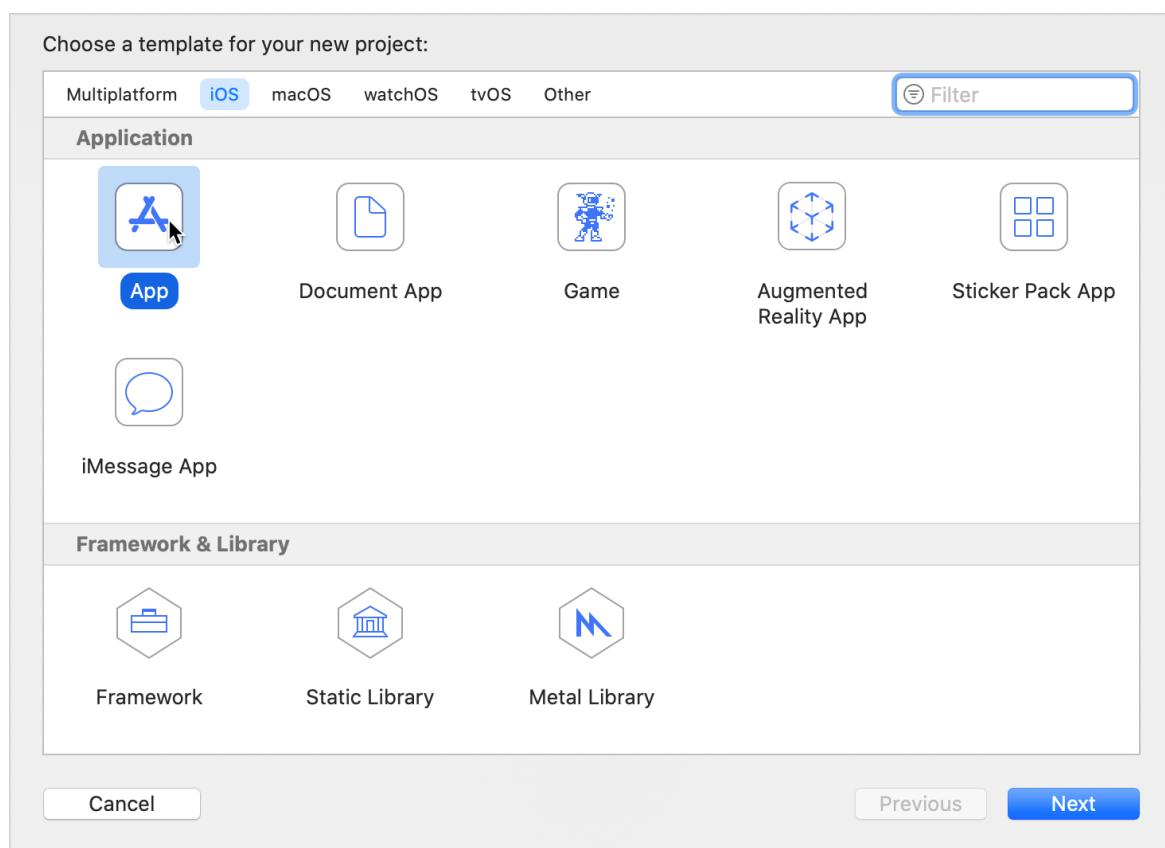
Make your cross-platform application work on iOS

Once you've made your Android application cross-platform, you can create an iOS application and reuse the shared business logic in it.

1. Create an iOS project in Xcode.
2. Connect the framework to your iOS project.
3. Use the shared module from Swift.

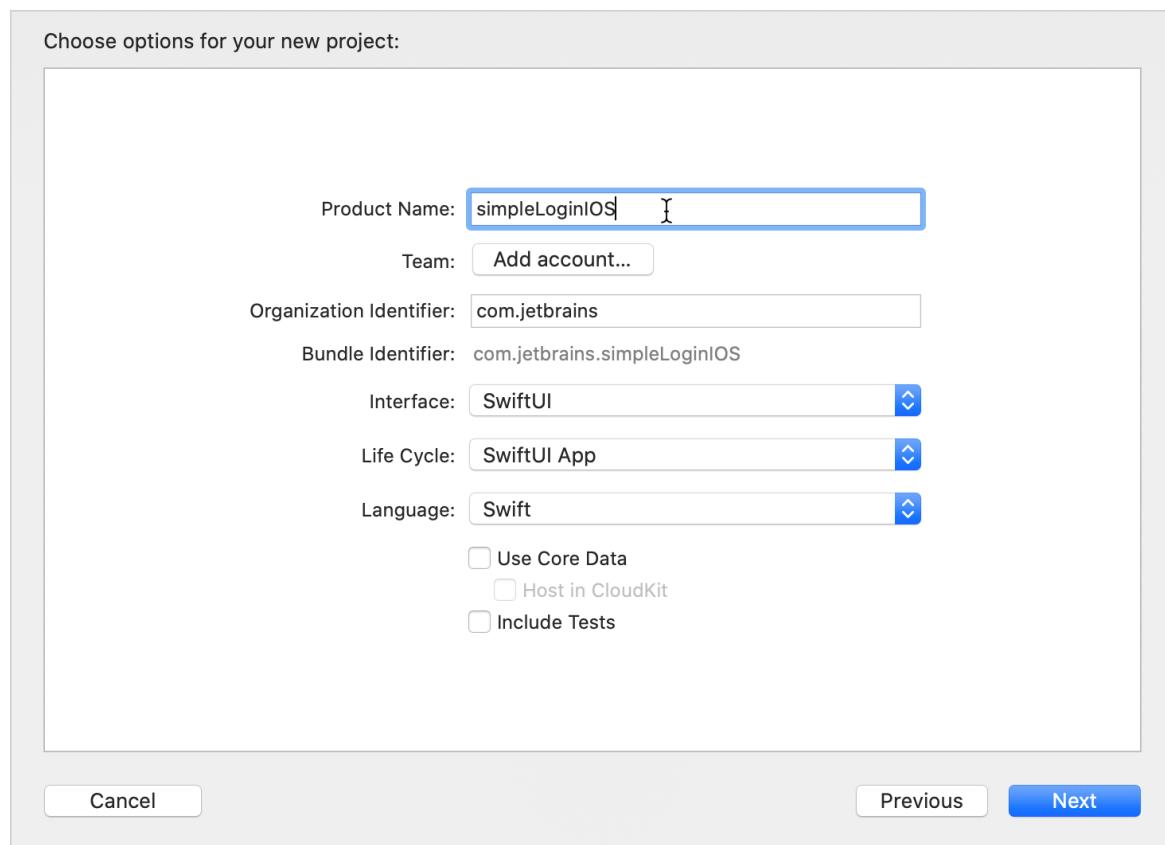
Create an iOS project in Xcode

1. In Xcode, click **File | New | Project**.
2. Select a template for an iOS app and click **Next**.



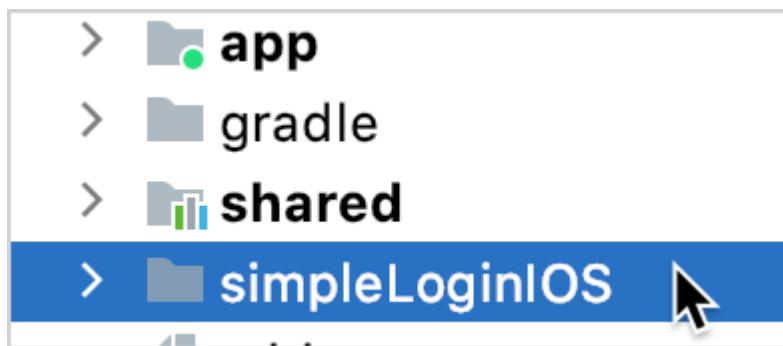
3. As the product name, specify **simpleLoginIOS** and click **Next**.

1.5.30 的新特性

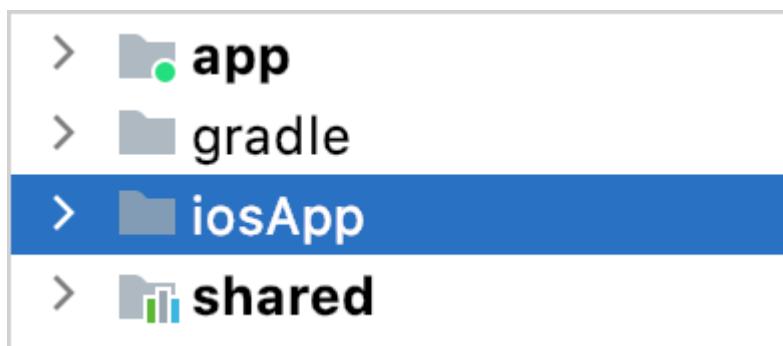


4. As the location for your project, select the directory that stores your cross-platform application, for example, `multiplatform-integrate-into-existing-app`.

In Android Studio, you'll get the following structure:



You can rename the `simpleLoginIOS` directory to `iosApp` for consistency with other top-level directories of your cross-platform project.



1.5.30 的新特性

Connect the framework to your iOS project

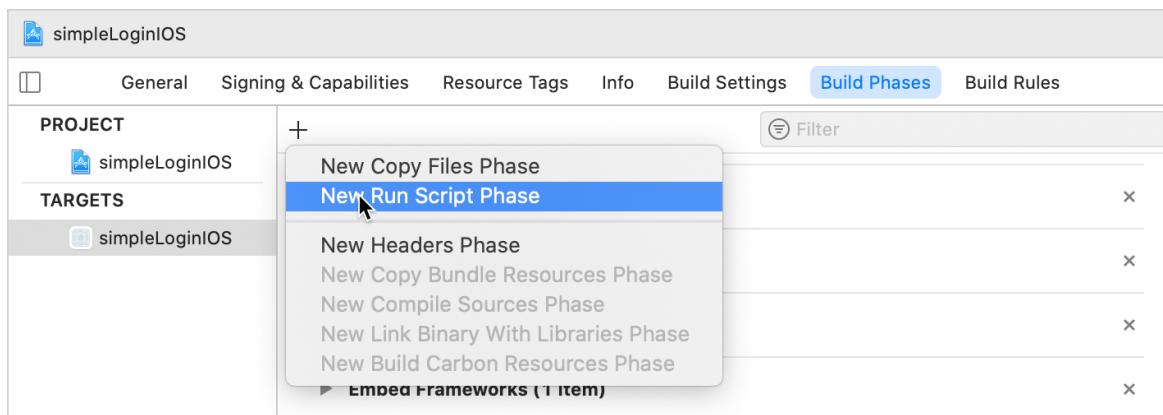
Once you have the framework, you can connect it to your iOS project manually.

An alternative is to [configure integration via Cocoapods](#), but that integration is beyond the scope of this tutorial.



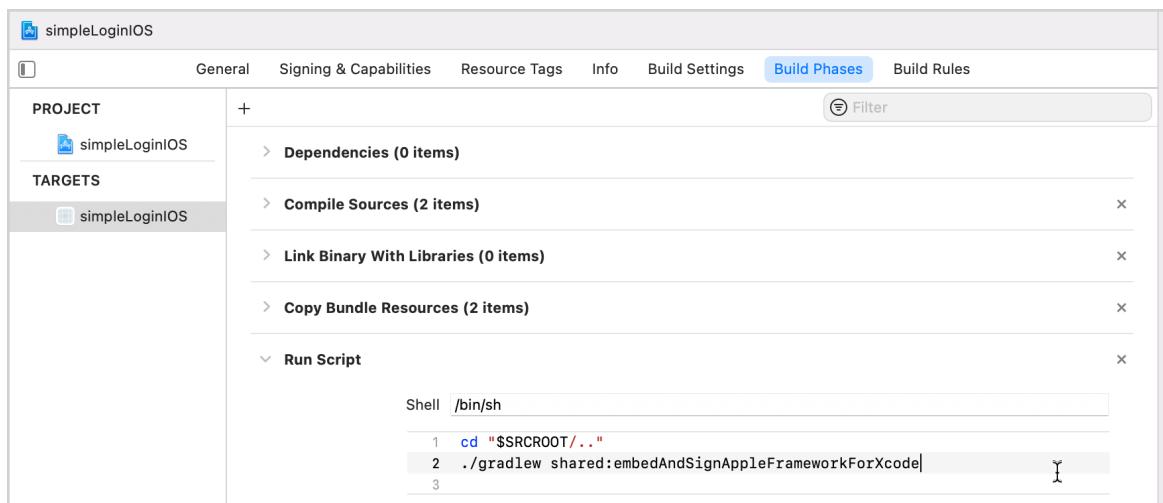
Connect your framework to the iOS project manually:

1. In Xcode, open the iOS project settings by double-clicking the project name.
2. On the **Build Phases** tab of the project settings, click the **+** and add **New Run Script Phase**.



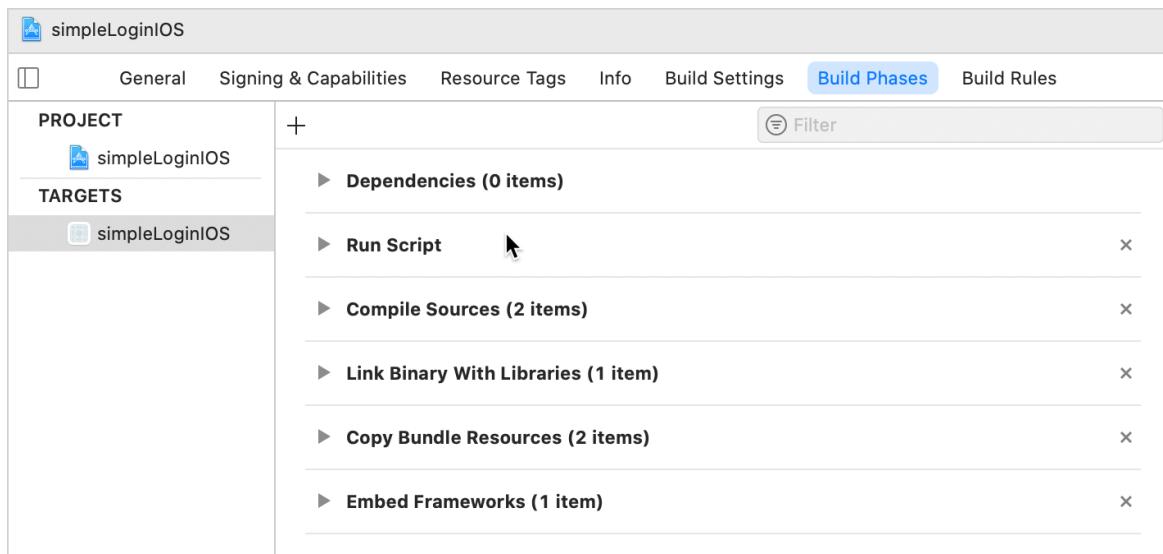
3. Add the following script:

```
cd "$SRCROOT/.."  
.gradlew :shared:embedAndSignAppleFrameworkForXcode
```



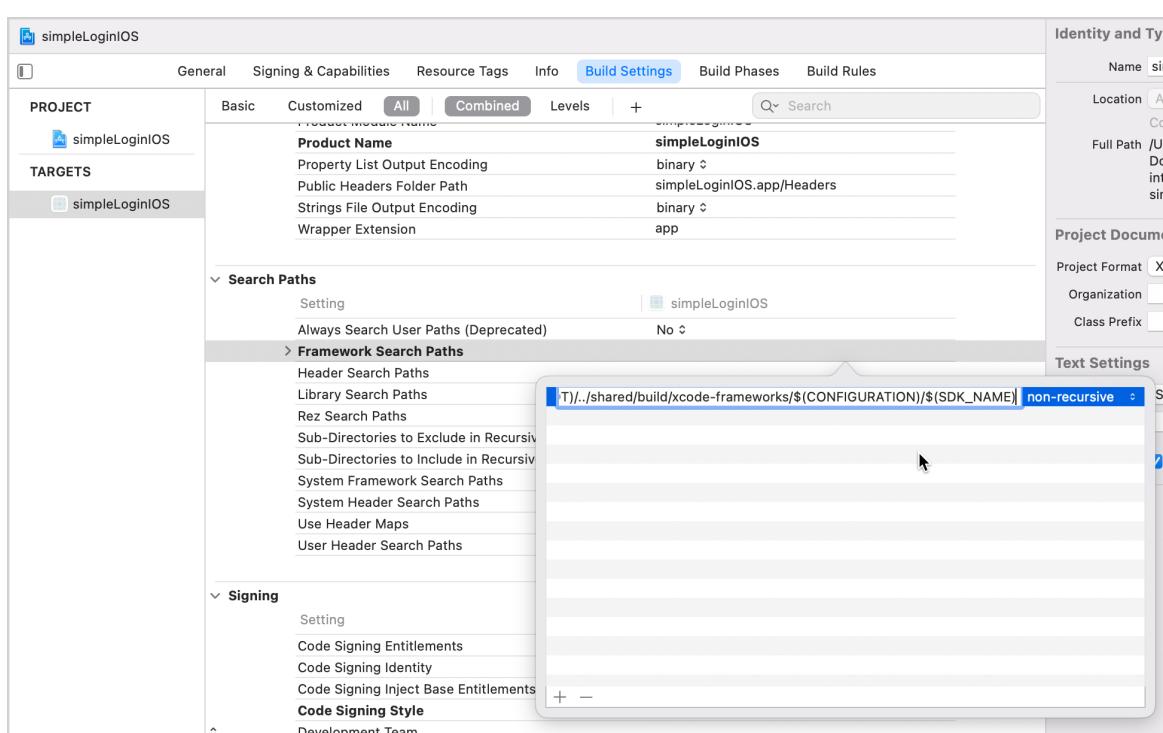
4. Move the **Run Script** phase before the **Compile Sources** phase.

1.5.30 的新特性



5. On the **Build Settings** tab, specify the **Framework Search Path** under **Search Paths**:

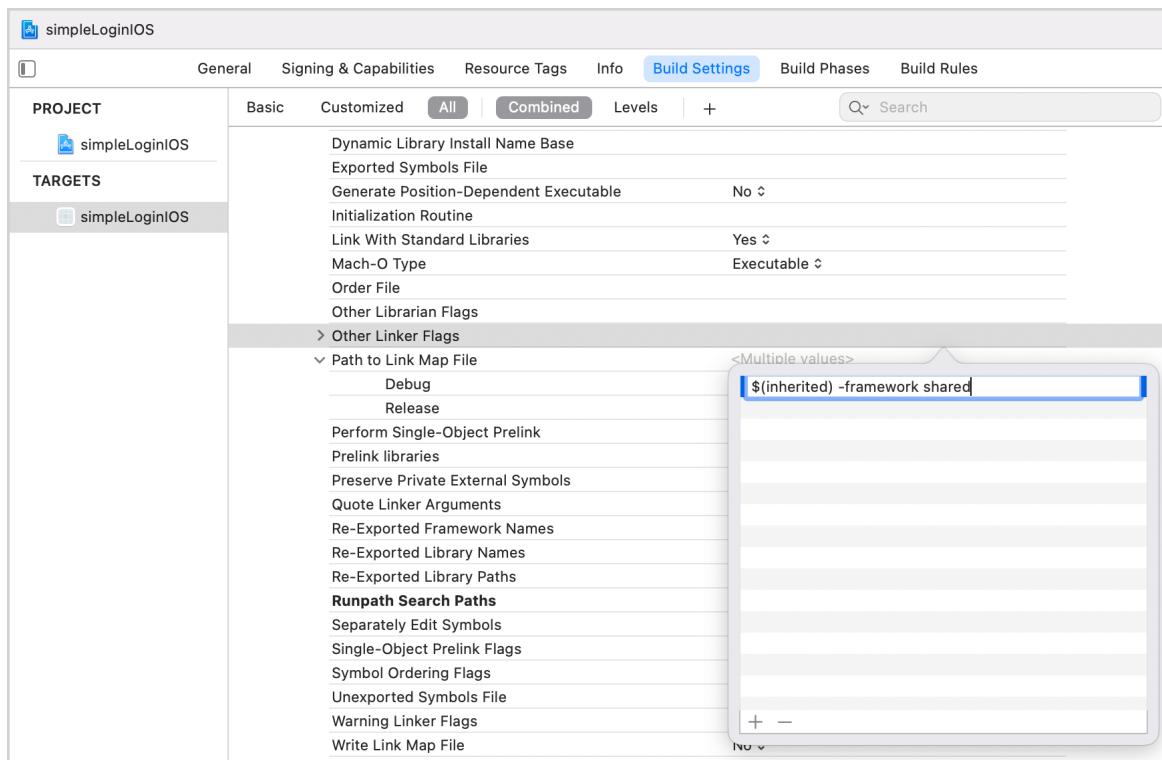
```
$(SRCROOT)/../shared/build/xcode-frameworks/$(CONFIGURATION)/$(SDK_NAME)
```



6. On the **Build Settings** tab, specify the **Other Linker flags** under **Linking**:

```
$(inherited) -framework shared
```

1.5.30 的新特性



7. Build the project in Xcode. If everything is set up correctly, the project will successfully build.

Use the shared module from Swift

1. In Xcode, open the `ContentView.swift` file and import the `shared` module.

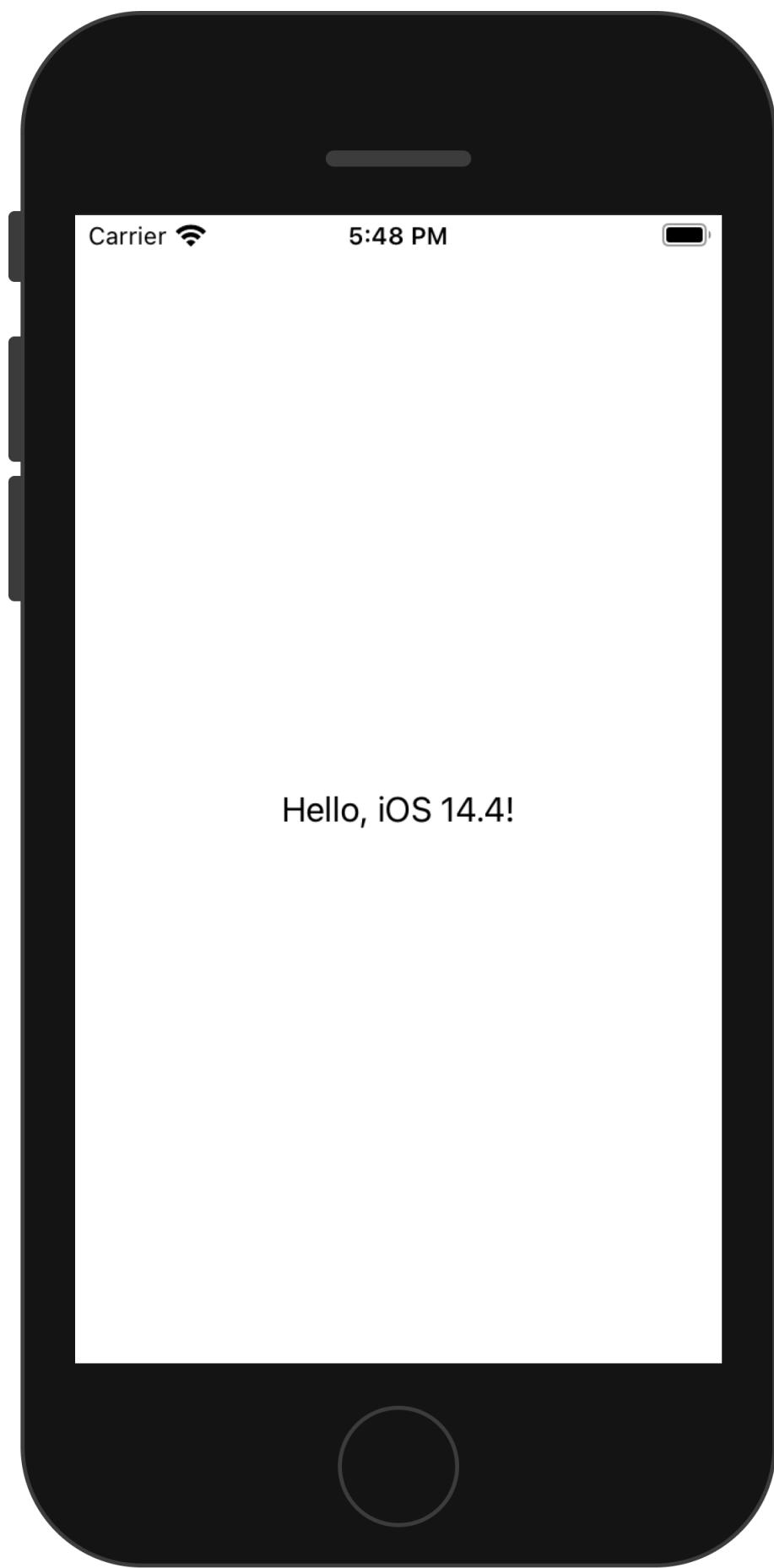
```
import shared
```

2. To check that it is properly connected, use the `greeting()` function from the Kotlin Multiplatform module:

```
import SwiftUI
import shared

struct ContentView: View {
    var body: some View {
        Text(Greeting().greeting())
            .padding()
    }
}
```

1.5.30 的新特性



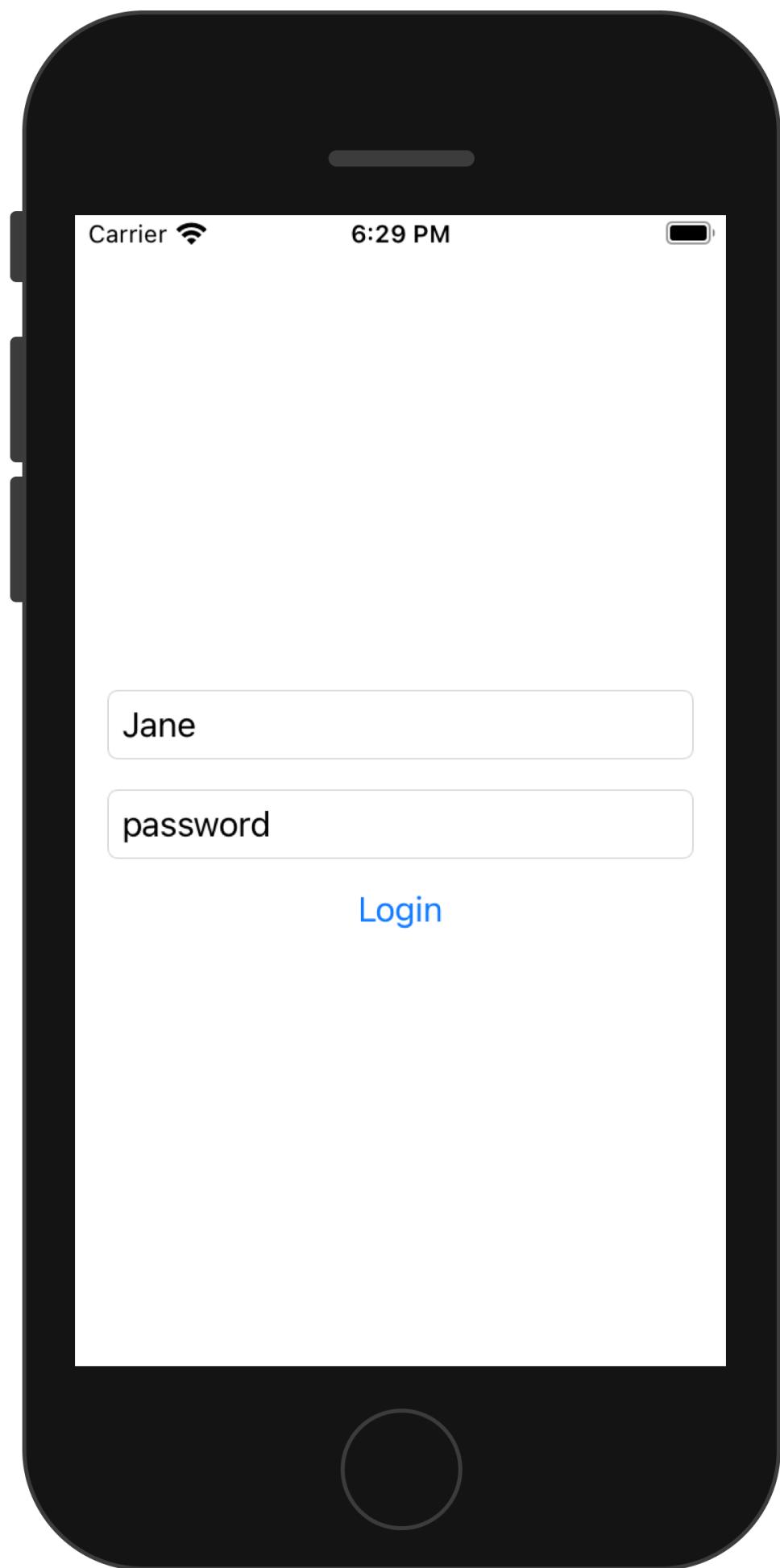
1.5.30 的新特性

3. In `ContentView.swift`, write code for using data from the Kotlin Multiplatform module and rendering the application UI.
4. In `simpleLoginIOSApp.swift`, import the `shared` module and specify the arguments for the `ContentView()` function:

```
import SwiftUI
import shared

@main
struct SimpleLoginIOSApp: App {
    var body: some Scene {
        WindowGroup {
            ContentView(viewModel: .init(loginRepository: LoginRepository(data:
        }
    }
}
```

1.5.30 的新特性



Enjoy the results – update the logic only once

Now your application is cross-platform. You can update the business logic in one place and see results on both Android and iOS.

1. In Android Studio, change the validation logic for a user's password in the

`checkPassword()` function of the `LoginDataValidator` class:

```
package com.jetbrains.simplelogin.shared.data

class LoginDataValidator {
    ...
    fun checkPassword(password: String): Result {
        return when {
            password.length < 5 -> Result.Error("Password must be >5 characters")
            password.toLowerCase() == "password" -> Result.Error("Password should not be 'password'")
            else -> Result.Success
        }
    }
    ...
}
```

2. Update `gradle.properties` to connect your iOS application to Android Studio for running it on a simulated or real device right there:

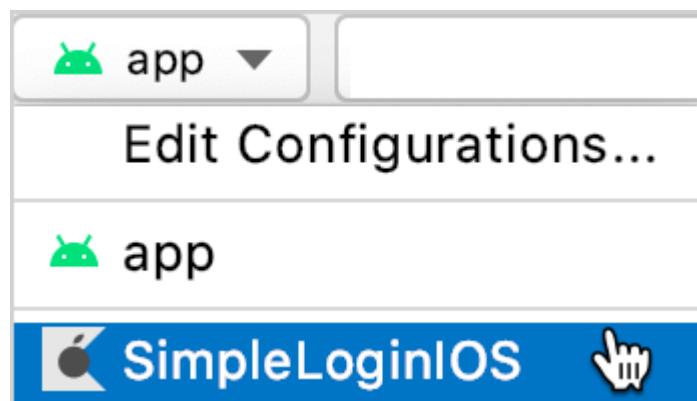
```
xcodeproj=iosApp/SimpleLoginIOS.xcodeproj
```

3. Synchronize the Gradle files by clicking **Sync Now** in the warning.

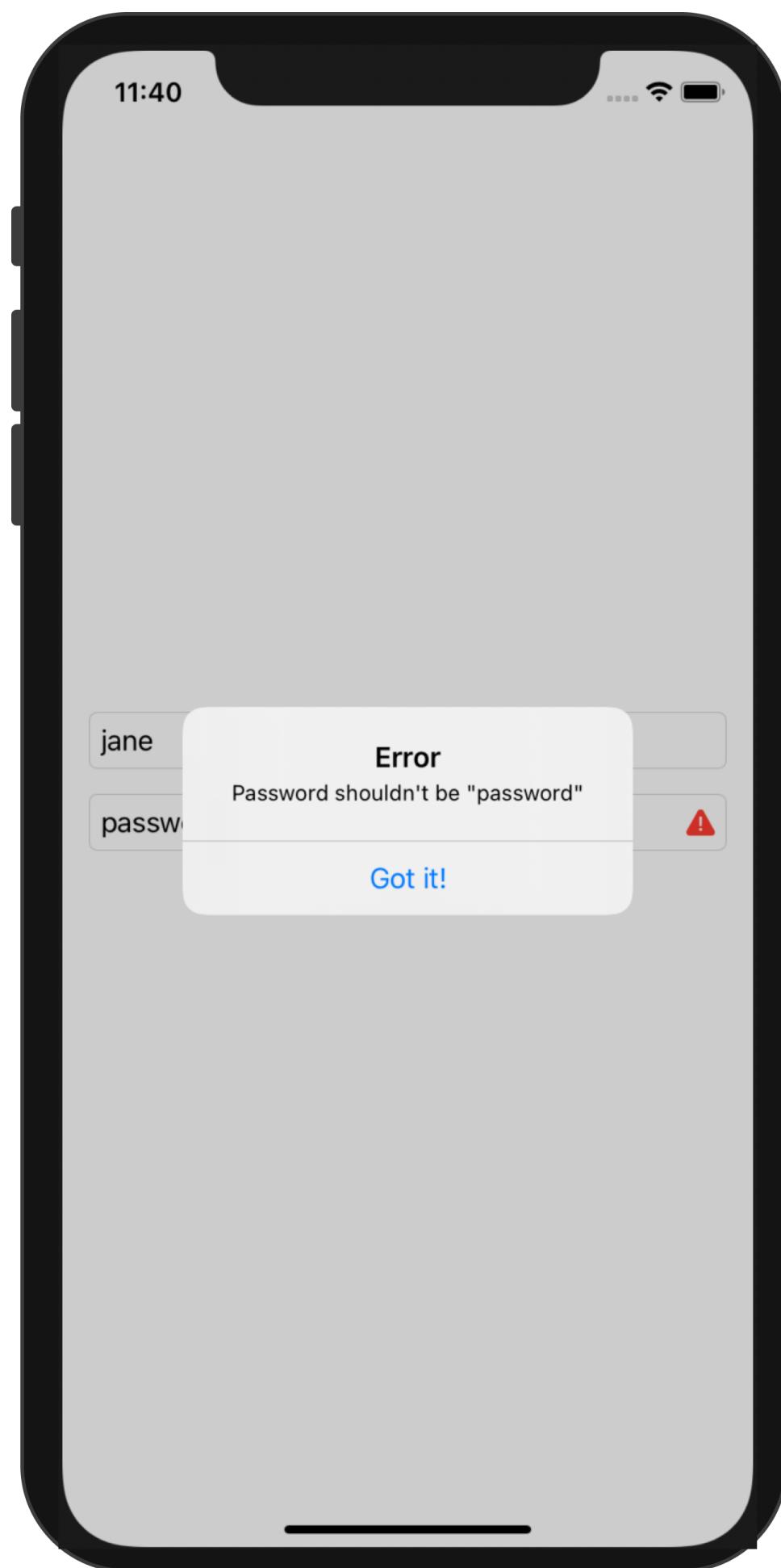
Gradle files have changed since last project sync. A project sync... [Sync Now](#) [Ignore these changes](#)

You will see the new run configuration **simpleLoginIOS** for running your iOS application right from Android Studio.

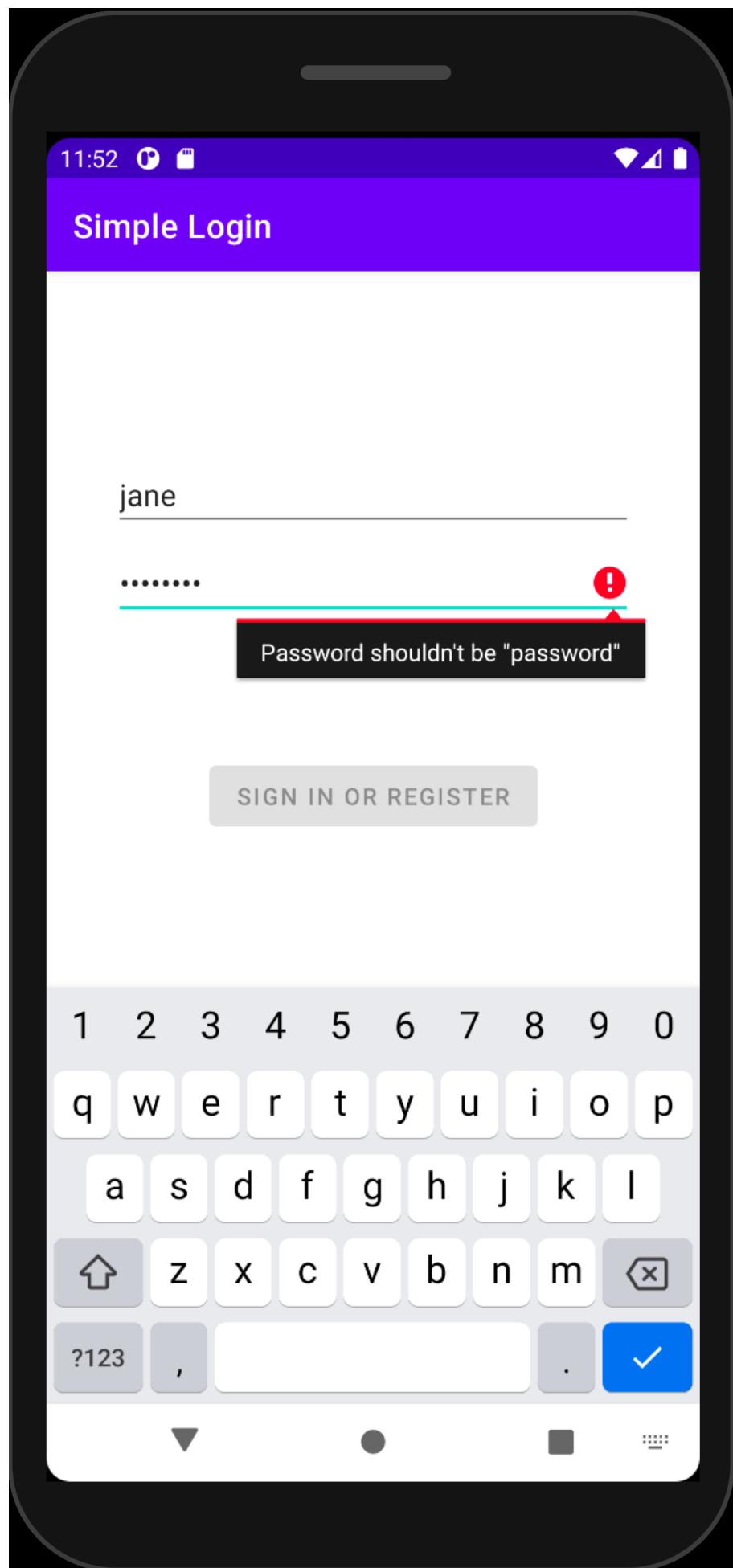
1.5.30 的新特性



1.5.30 的新特性



1.5.30 的新特性



1.5.30 的新特性

You can review the [final code for this tutorial](#).

What else to share?

You've shared the business logic of your application, but you can also decide to share other layers of your application. For example, the `ViewModel` class code is almost the same for [Android](#) and [iOS applications](#), and you can share it if your mobile applications should have the same presentation layer.

下一步做什么？

Once you've made your Android application cross-platform, you can move on and:

- [Add dependencies on multiplatform libraries](#)
- [Add Android dependencies](#)
- [Add iOS dependencies](#)
- [Learn about concurrency](#)

You can also check out community resources:

- [Video: 3 ways to get your Kotlin JVM code ready for Kotlin Multiplatform Mobile](#)

发布应用程序

Once your mobile apps are ready for release, it's time to deliver them to the users by publishing them in app stores. Multiple stores are available for each platform.

However, in this article we'll focus on the official ones: [Google Play Store](#) and [Apple App Store](#). You'll learn how to prepare Kotlin Multiplatform Mobile applications for publishing, and we'll highlight the parts of this process that deserve special attention.

Android app

Since [Kotlin is the main language for Android development](#), Kotlin Multiplatform Mobile has no obvious effect on compiling the project and building the Android app. Both the Android library produced from the shared module and the Android app itself are typical Android Gradle modules; they are no different from other Android libraries and apps. Thus, publishing the Android app from a Kotlin Multiplatform project is no different from the usual process described in the [Android developer documentation](#).

iOS app

The iOS app from a Kotlin Multiplatform project is built from a typical Xcode project, so the main stages involved in publishing it are the same as described in the [iOS developer documentation](#).

What is specific to Kotlin Multiplatform projects is compiling the shared Kotlin module into a framework and linking it to the Xcode project. Generally, all integration between the shared module and the Xcode project is done automatically by the [Kotlin Multiplatform Mobile plugin for Android Studio](#). However, if you don't use the plugin, bear in mind the following when building and bundling the iOS project in Xcode:

- The shared Kotlin library compiles down to the native framework.
- You need to connect the framework compiled for the specific platform to the iOS app project.
- In the Xcode project settings, specify the path to the framework to search for the build system.
- After building the project, you should launch and test the app to make sure that there are no issues when working with the framework in runtime.

1.5.30 的新特性

There are two ways you can connect the shared Kotlin module to the iOS project:

- Use the [Kotlin/Native Cocoapods plugin](#), which allows you to use a multiplatform project with native targets as a CocoaPods dependency in your iOS project.
- Manually configure your Multiplatform project to create an iOS framework and the XCode project to obtain its latest version. The Kotlin Multiplatform Mobile plugin for Android Studio usually does this configuration. [Understand the project structure](#) to implement it yourself.

Symbolicating crash reports

To help developers make their apps better, iOS provides a means for analyzing app crashes. For detailed crash analysis, it uses special debug symbol (`.dSYM`) files that match memory addresses in crash reports with locations in the source code, such as functions or line numbers.

By default, the release versions of iOS frameworks produced from the shared Kotlin module have an accompanying `.dSYM` file. This helps you analyze crashes that happen in the shared module's code.

When an iOS app is rebuilt from bitcode, its `dSYM` file becomes invalid. For such cases, you can compile the shared module to a static framework that stores the debug information inside itself. For instructions on setting up crash report symbolication in binaries produced from Kotlin modules, see the [Kotlin/Native documentation](#).

Kotlin 多平台用于其他平台

- [Kotlin 多平台入门](#)
- [了解多平台项目结构](#)
- [手动设置目标](#)

Kotlin 多平台入门

Multiplatform projects are in [Alpha](#). Language features and tooling may change in future Kotlin versions.



Support for multiplatform programming is one of Kotlin's key benefits. It reduces time spent writing and maintaining the same code for [different platforms](#) while retaining the flexibility and benefits of native programming.

Learn more about [Kotlin Multiplatform benefits](#).

Start from scratch

- [Create and publish a multiplatform library](#) teaches how to create a multiplatform library available for JVM, JS, and Native and which can be used from any other common code (for example, shared with Android and iOS). It also shows how to write tests which will be executed on all platforms and use an efficient implementation provided by a specific platform.
- [Build a Full Stack Web App with Kotlin Multiplatform](#) teaches the concepts behind building an application that targets Kotlin/JVM and Kotlin/JS by building a client-server application that makes use of shared code, serialization, and other multiplatform paradigms. It also provides a brief introduction to working with Ktor both as a server- and client-side framework.
- [Create your first Kotlin Multiplatform Mobile application](#) shows how to create a mobile application that works on Android and iOS with the help of the [Kotlin Multiplatform Mobile plugin for Android Studio](#). Create, run, and test your first multiplatform mobile application.

Dive deep into Kotlin Multiplatform

Once you have gained some experience with Kotlin Multiplatform and want to know how to solve particular cross-platform development tasks:

- [Share code on platforms](#) in your Kotlin Multiplatform project.

1.5.30 的新特性

- [Connect to platform-specific APIs](#) using the Kotlin mechanism of expected and actual declarations.
- [Set up targets manually](#) for your Kotlin Multiplatform project.
- [Add dependencies](#) on the standard, test, or another kotlinx library.
- [Configure compilations](#) for production and test purposes in your project.
- [Run tests](#) for JVM, JavaScript, Android, Linux, Windows, macOS, iOS, watchOS, and tvOS simulators.
- [Publish a multiplatform library](#) to the Maven repository.
- [Build native binaries](#) as executables or shared libraries, like universal frameworks or XCFrameworks.

Get help

- **Kotlin Slack:** Get an [invite](#) and join the [#multiplatform](#) channel
- **StackOverflow:** Subscribe to the “[kotlin-multiplatform](#)” tag
- **Kotlin issue tracker:** [Report a new issue](#)

了解多平台项目结构

探索多平台项目的主要部分：

- 多平台插件
- 目标
- 源代码集
- 编译项

多平台插件

创建多平台项目时，项目向导会自动在 `build.gradle(.kts)` 文件中应用 `kotlin-multiplatform` 插件。

也可以手动应用它。

`kotlin-multiplatform` 插件适用于 Gradle 6.1 或更高版本。



【Kotlin】

```
plugins {  
    kotlin("multiplatform") version "1.6.10"  
}
```

【Groovy】

```
plugins {  
    id 'org.jetbrains.kotlin.multiplatform' version '1.6.10'  
}
```

`kotlin-multiplatform` 插件可配置项目以创建可在多个平台上工作的应用程序或库，并为在这些平台上构建做好准备。

在 `build.gradle(.kts)` 文件中，它在顶层创建 `kotlin` 扩展，其中包括目标、源代码集与依赖项的配置。

目标

一个多平台项目针对以不同目标表示的多个平台。目标是构建的一部分，负责为特定平台（例如 macOS、iOS 或 Android）构建、测试与打包应用程序。请参阅[支持的平台列表](#)。

创建多平台项目时，会将目标添加到 `build.gradle` (`build.gradle.kts`) 文件中的 `kotlin` 块中。

```
kotlin {  
    jvm()  
    js {  
        browser {}  
    }  
}
```

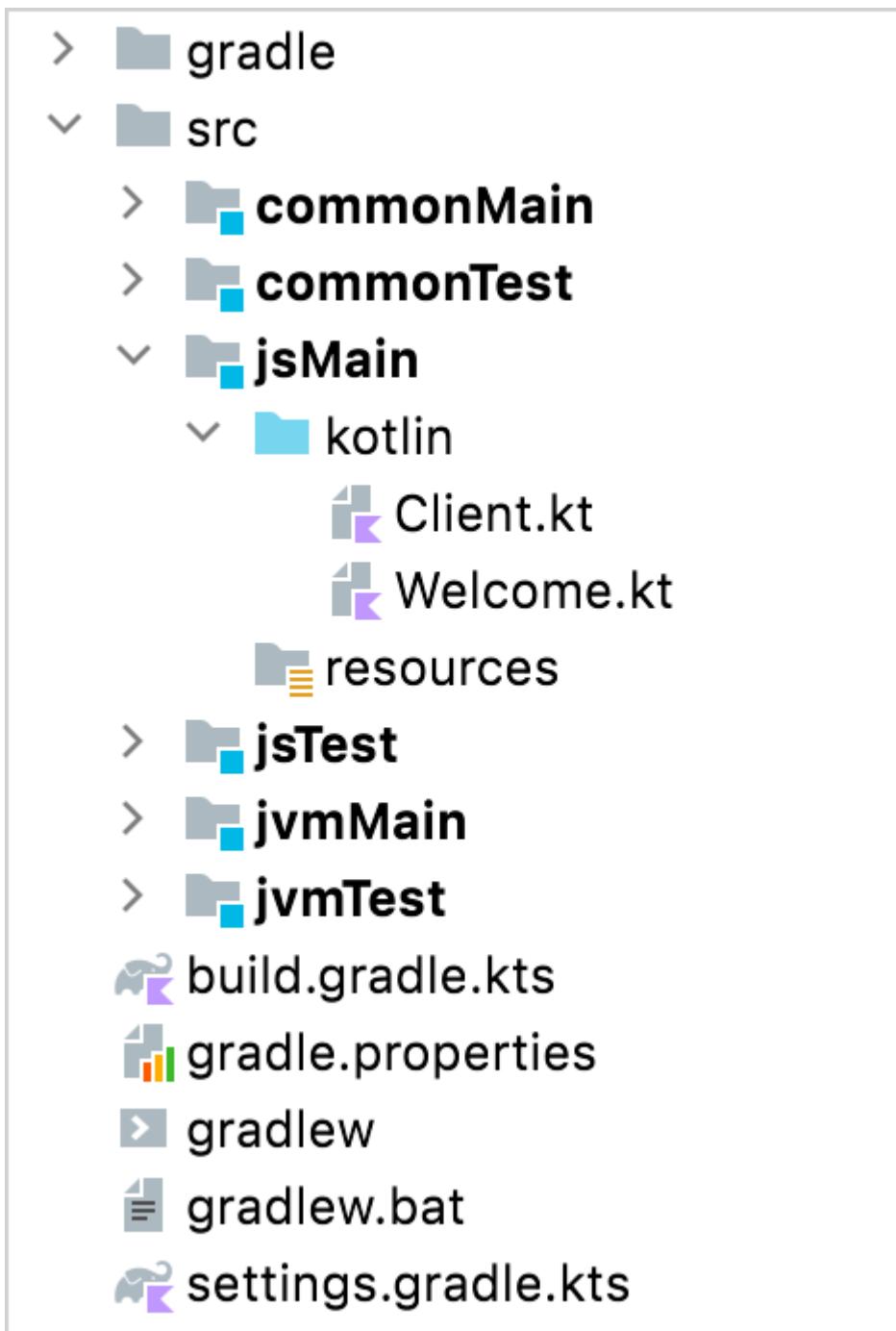
了解如何[手动设置目标](#)。

源代码集

该项目包括带有 Kotlin 源代码集的 `src` 目录，这些源代码集是 Kotlin 代码文件的集合，以及源代码集的资源、依赖与语言设置。可以在 Kotlin 编译项中使用一个或多个源代码集目标平台。

每个源代码集目录都包含 Kotlin 代码文件 (`kotlin` 目录) 与 `resources`。项目向导会为公共代码以及所有已添加目标的 `main` 与 `test` 编译项创建默认源代码集。

1.5.30 的新特性



源代码集名称区分大小写。



源代码集被添加到顶层 `kotlin` 块的 `sourceSets` 块中。

【Kotlin】

1.5.30 的新特性

```
kotlin {  
    sourceSets {  
        val commonMain by getting { /* ... */ }  
        val commonTest by getting { /* ... */ }  
        val jvmMain by getting { /* ... */ }  
        val jvmTest by getting { /* ... */ }  
        val jsMain by getting { /* ... */ }  
        val jsTest by getting { /* ... */ }  
    }  
}
```

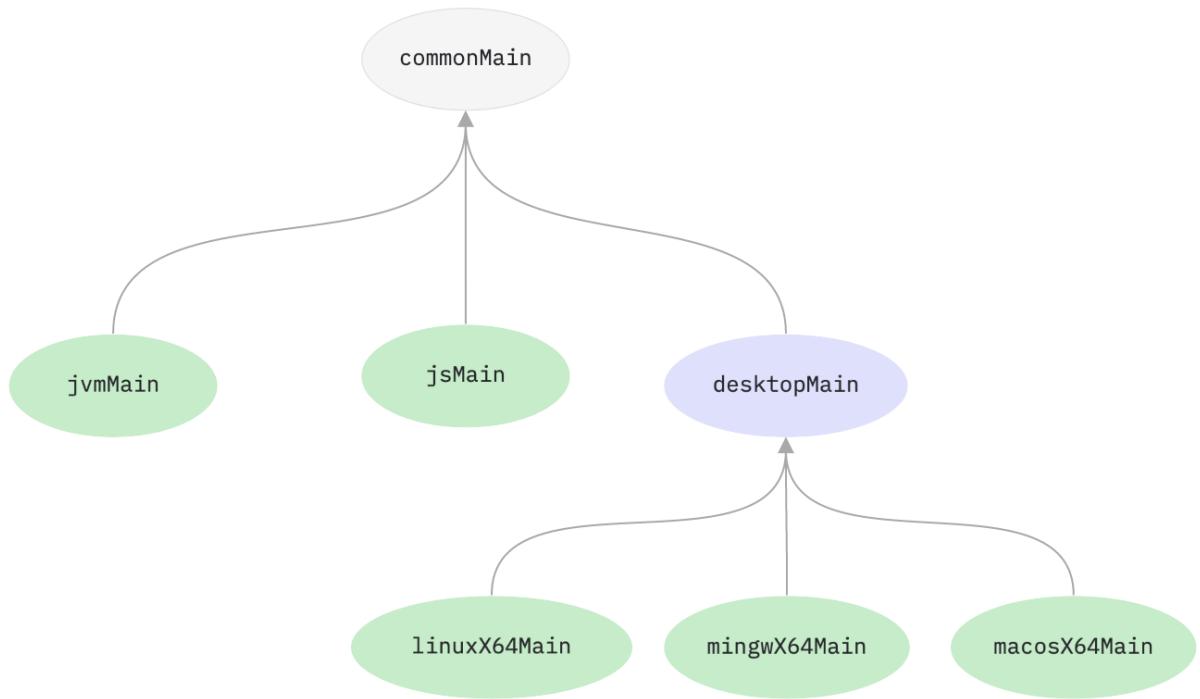
【Groovy】

```
kotlin {  
    sourceSets {  
        commonMain { /* ... */}  
        commonTest { /* ... */}  
        jvmMain { /* ... */}  
        jvmTest { /* ... */ }  
        jsMain { /* ... */}  
        jsTest { /* ... */ }  
    }  
}
```

源代码集形成一个层次结构，用于共享公共代码。在多个目标之间共享的源代码集中，可以使用所有这些目标可用的特定于平台的语言特性与依赖。

例如，所有 Kotlin 原生特性都可以在 `desktopMain` 源代码集中可用，该源代码集的目标是 Linux(`linuxX64`)、Windows(`mingwX64`) 与 macOS(`macosX64`) 平台。

1.5.30 的新特性



了解如何[构建源代码集的层次结构](#)。

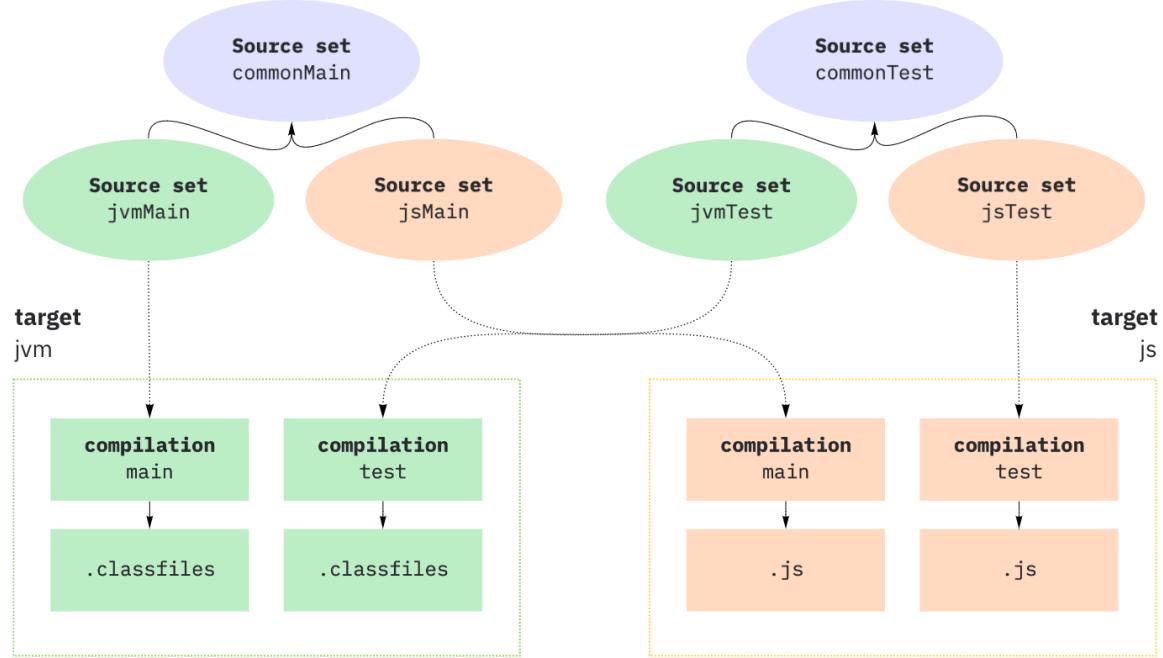
编译项

每个目标可以具有一个或多个编译项，例如，用于生产与测试目的。

对于每个目标，默认编译项包括：

- 针对 JVM、JS 与原生目标的 `main` 与 `test` 编译项。
- 针对 Android 目标的每个 [Android 构建变体](#) 的编译。

1.5.30 的新特性



每个编译都有默认的源代码集，其中包含特定于该编译的源代码与依赖。

了解如何[配置编译项](#)。

为 Kotlin 多平台手动设置目标

You can add targets when [creating a project with the Project Wizard](#). If you need to add a target later, you can do this manually using target presets for [supported platforms](#).

Learn more about [additional settings for targets](#).

```
kotlin {  
    jvm() // Create a JVM target with the default name 'jvm'  
  
    linuxX64() {  
        /* Specify additional settings for the 'linux' target here */  
    }  
}
```

Each target can have one or more [compilations](#). In addition to default compilations for test and production purposes, you can [create custom compilations](#).

区分一个平台的多个目标

You can have several targets for one platform in a multiplatform library. For example, these targets can provide the same API but use different libraries during runtime, such as testing frameworks and logging solutions. Dependencies on such a multiplatform library may fail to resolve because it isn't clear which target to choose.

To solve this, mark the targets on both the library author and consumer sides with a custom attribute, which Gradle uses during dependency resolution.

For example, consider a testing library that supports both JUnit and TestNG in the two targets. The library author needs to add an attribute to both targets as follows:

【Kotlin】

1.5.30 的新特性

```
val testFrameworkAttribute = Attribute.of("com.example.testFramework", String::class.java)

kotlin {
    jvm("junit") {
        attributes.attribute(testFrameworkAttribute, "junit")
    }
    jvm("testng") {
        attributes.attribute(testFrameworkAttribute, "testng")
    }
}
```

【Groovy】

```
def testFrameworkAttribute = Attribute.of('com.example.testFramework', String)

kotlin {
    jvm('junit') {
        attributes.attribute(testFrameworkAttribute, 'junit')
    }
    jvm('testng') {
        attributes.attribute(testFrameworkAttribute, 'testng')
    }
}
```

The consumer has to add the attribute to a single target where the ambiguity arises.

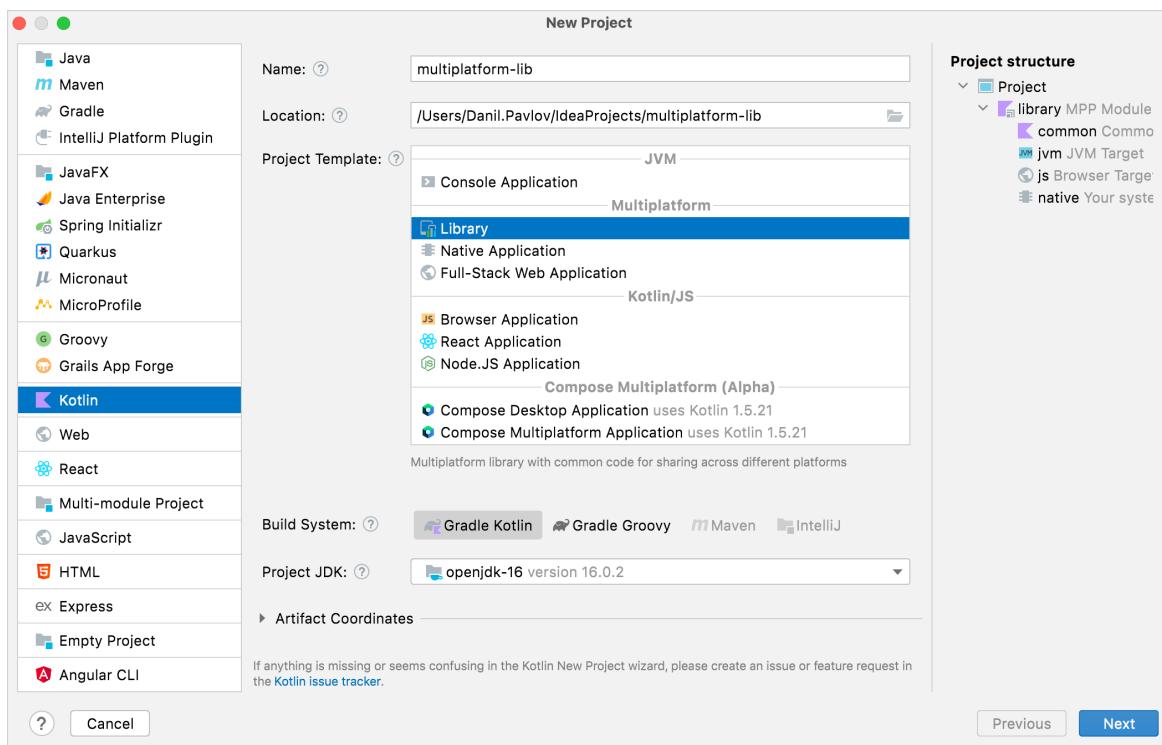
创建多平台库

- [创建多平台库](#)
- [发布多平台库](#)
- [创建并发布多平台库——教程](#)

创建多平台库

本节提供了创建多平台库的步骤。还可以完成本[教程](#)，在教程中将创建一个多平台库，对其进行测试，然后将其发布到 Maven。

1. 在 IntelliJ IDEA 中，选择 **文件(File) | 新建(New) | 项目(Project)**。
2. 在左侧面板中，选择 **Kotlin**。
3. 输入项目名称，然后在 **Multiplatform** 分组中选择 **Library** 作为项目模板。



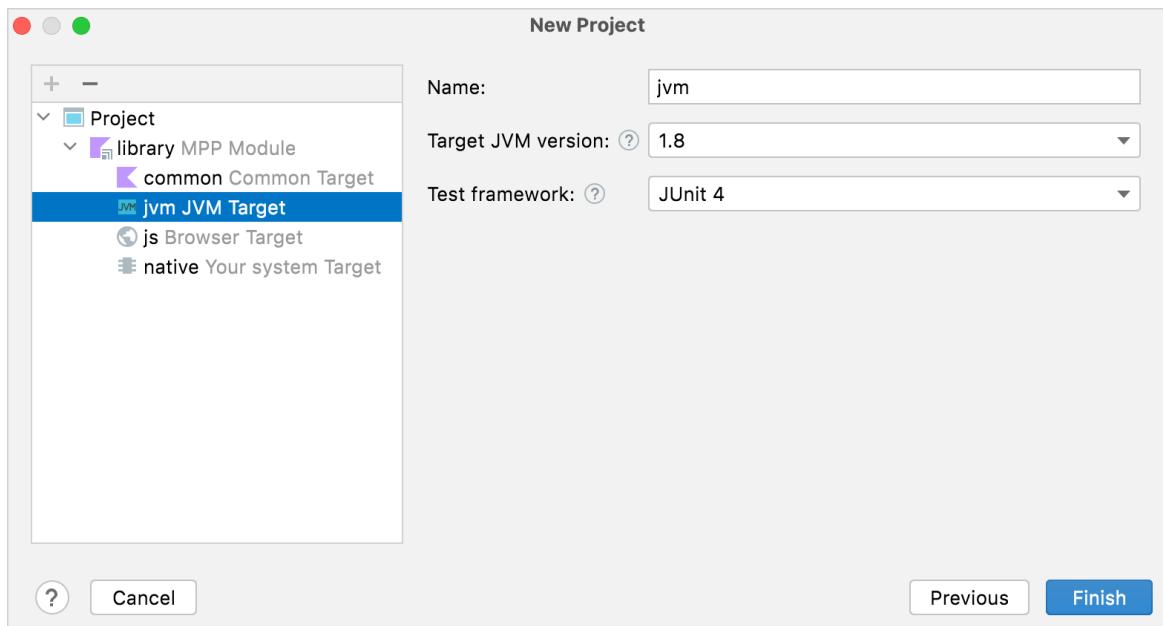
4. 选择 Gradle DSL——Kotlin 或 Groovy。

5. 点击 **下一步(Next)**。

可以通过在下一个屏幕上单击 **Finish** 来完成创建项目，或在必要时进行配置：

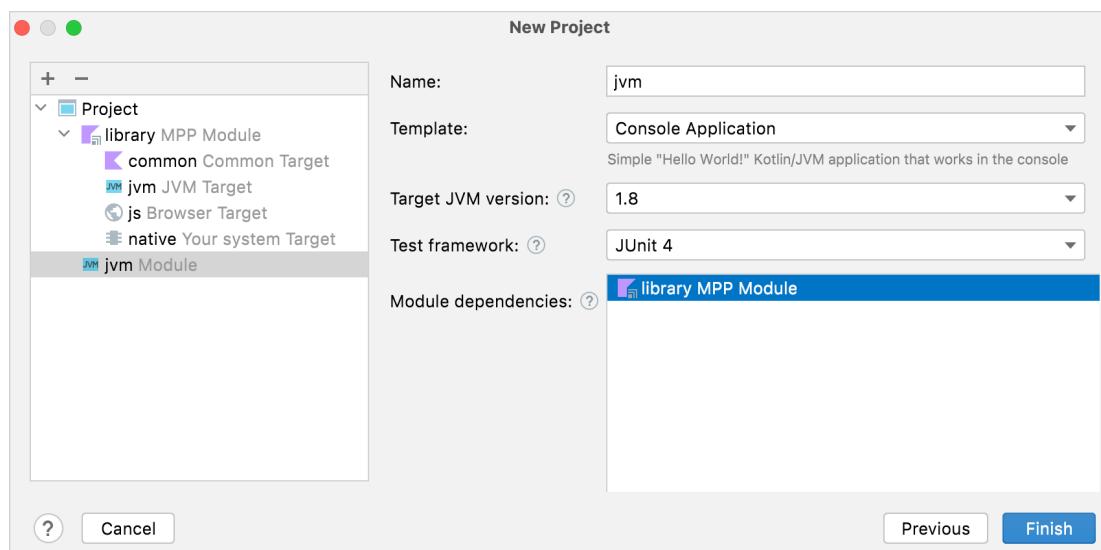
1. 通过单击 **+** 图标添加目标平台与模块。
2. 配置目标设置，例如目标 JVM 版本与测试框架。

1.5.30 的新特性



3. 如有必要，可以指定模块之间的依赖关系：

- 多平台与 Android 模块
- 多平台与 iOS 模块
- JVM 模块



4. 点击 **Finish**。

The new project opens.

下一步做什么？

- Understand the multiplatform project structure.
- Create and publish a multiplatform library – tutorial.
- Create your first cross-platform mobile application – tutorial.

1.5.30 的新特性

- Create a full-stack web app with Kotlin Multiplatform – hands-on tutorial.

发布多平台库

You can publish a multiplatform library to a Maven repository with the `maven-publish` Gradle plugin. Specify the group, version, and the `repositories` where the library should be published. The plugin creates publications automatically.

```
plugins {
    //...
    id("maven-publish")
}

group = "com.example"
version = "1.0"

publishing {
    repositories {
        maven {
            //...
        }
    }
}
```

Complete the [tutorial on creating and publishing a multiplatform library](#) to get hands-on experience.

Structure of publications

When used with `maven-publish`, the Kotlin plugin automatically creates publications for each target that can be built on the current host, except for the Android target, which needs an [additional step to configure publishing](#).

Publications of a multiplatform library include an additional `root` publication `kotlinMultiplatform` that 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 [adding dependencies](#).

This `kotlinMultiplatform` publication includes metadata artifacts and references the other publications as its variants.

1.5.30 的新特性

Some repositories, such as Maven Central, require that the root module contains a JAR artifact without a classifier, for example `kotlinMultiplatform-1.0.jar`.

The Kotlin Multiplatform plugin automatically produces the required artifact with the embedded metadata artifacts.

This means you don't have to customize your build by adding an empty artifact to the root module of your library to meet the repository's requirements.



The `kotlinMultiplatform` publication may also need the sources and documentation artifacts if that is required by the repository. In that case, add those artifacts by using `artifact(...)` in the publication's scope.

Avoid duplicate publications

To avoid duplicate publications of modules that can be built on several platforms (like JVM and JS), configure the publishing tasks for these modules to run conditionally.

You can detect the platform in the script, introduce a flag such as `isMainHost` and set it to `true` for the main target platform. Alternatively, you can pass the flag from an external source, for example, from CI configuration.

This simplified example ensures that publications are only uploaded when `isMainHost=true` is passed. This means that a publication that can be published from multiple platforms will be published only once – from the main host.

【Kotlin】

1.5.30 的新特性

```
kotlin {
    jvm()
    js()
    mingwX64()
    linuxX64()
    val publicationsFromMainHost =
        listOf(jvm(), js()).map { it.name } + "kotlinMultiplatform"
    publishing {
        publications {
            matching { it.name in publicationsFromMainHost }.all {
                val targetPublication = this@all
                tasks.withType<AbstractPublishToMaven>()
                    .matching { it.publication == targetPublication }
                    .configureEach { onlyIf { findProperty("isMainHost") == "true" } }
            }
        }
    }
}
```

【Groovy】

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

    def publicationsFromMainHost =
        [jvm(), js()].collect { it.name } + "kotlinMultiplatform"
    publishing {
        publications {
            matching { it.name in publicationsFromMainHost }.all { targetPublication ->
                tasks.withType(AbstractPublishToMaven)
                    .matching { it.publication == targetPublication }
                    .configureEach { onlyIf { findProperty("isMainHost") == "true" } }
            }
        }
    }
}
```

By default, each publication includes a sources JAR that contains the sources used by the main compilation of the target.

发布 Android 库

1.5.30 的新特性

To publish an Android library, you need to provide additional configuration.

By default, no artifacts of an Android library are published. To publish artifacts produced by a set of [Android variants](#), specify the variant names in the Android target block:

```
kotlin {  
    android {  
        publishLibraryVariants("release", "debug")  
    }  
}
```

The example works for Android libraries without [product flavors](#). For a library with product flavors, the variant names also contain the flavors, like `fooBarDebug` or `fooBazRelease`.

The default publishing setup is as follows:

- If the published variants have the same build type (for example, all of them are `release` or `debug`), they will be compatible with any consumer build type.
- If the published variants have different build types, then only the release variants will be compatible with consumer build types that are not among the published variants. All other variants (such as `debug`) will only match the same build type on the consumer side, unless the consumer project specifies the [matching fallbacks](#).

If you want to make every published Android variant compatible with only the same build type used by the library consumer, set this Gradle property:

```
kotlin.android.buildTypeAttribute.keep=true .
```

You can also publish variants grouped by the product flavor, so that the outputs of the different build types are placed in a single module, with the build type becoming a classifier for the artifacts (the release build type is still published with no classifier). This mode is disabled by default and can be enabled as follows:

```
kotlin {  
    android {  
        publishLibraryVariantsGroupedByFlavor = true  
    }  
}
```

1.5.30 的新特性

It is not recommended that you publish variants grouped by the product flavor in case they have different dependencies, as those will be merged into one dependencies list.



创建并发布多平台库——教程

In this tutorial, you will learn how to create a multiplatform library for JVM, JS, and Native platforms, write common tests for all platforms, and publish the library to a local Maven repository.

This library converts raw data – strings and byte arrays – to the [Base64](#) format. It can be used on Kotlin/JVM, Kotlin/JS, and any available Kotlin/Native platform.

You will use different ways to implement the conversion to the Base64 format on different platforms:

- For JVM – the [java.util.Base64 class](#).
- For JS – the [btoa\(\) function](#).
- For Kotlin/Native – your own implementation.

You will also test your code using common tests, and then publish the library to your local Maven repository.

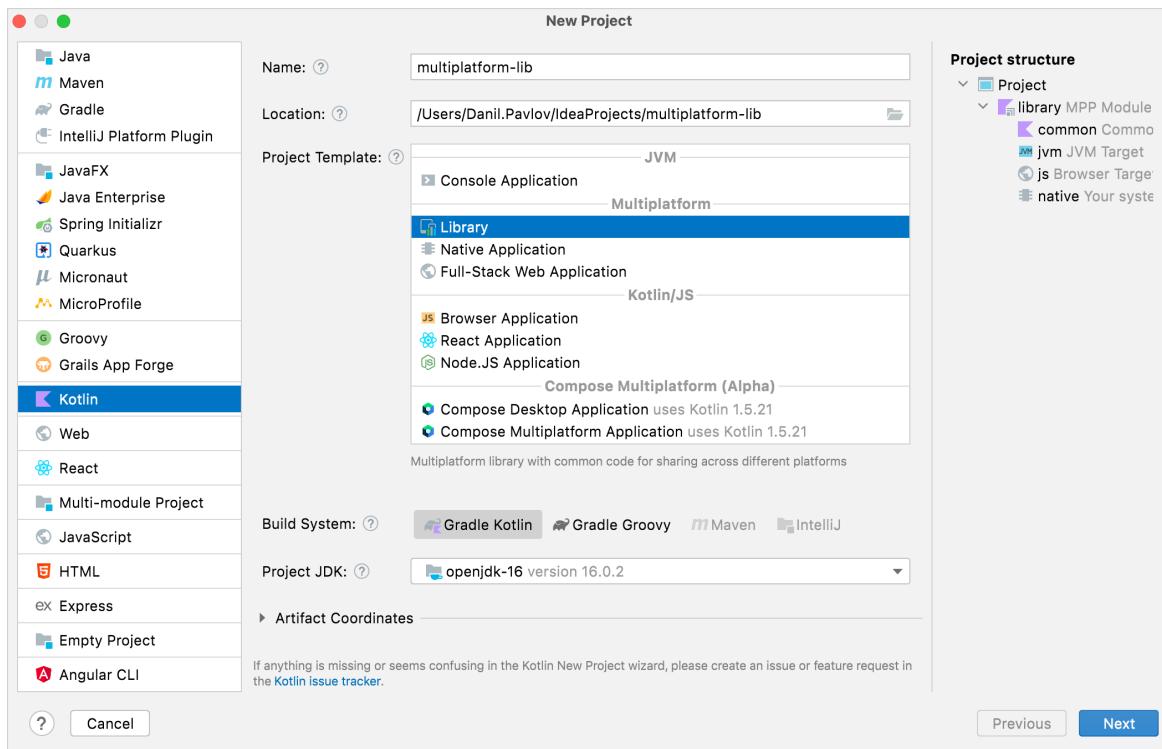
搭建环境

You can complete this tutorial on any operating system. Download and install the [latest version of IntelliJ IDEA](#) with the [latest Kotlin plugin](#).

创建一个项目

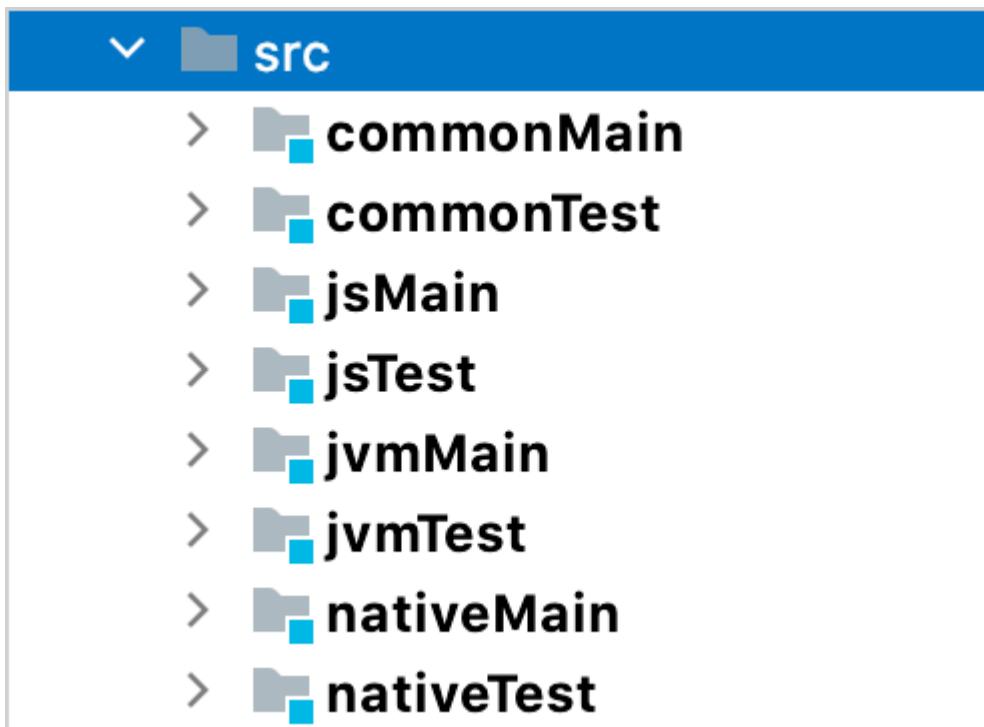
1. In IntelliJ IDEA, select **File | New | Project**.
2. In the left-hand panel, select **Kotlin**.
3. Enter a project name, then in the **Multiplatform** section select **Library** as the project template.

1.5.30 的新特性



4. Select the Gradle DSL – Kotlin or Groovy.
5. Specify the [JDK](#), which is required for developing Kotlin projects.
6. Click **Next**, and then click **Finish**.

The wizard will create a sample multiplatform library with the following structure:



Write cross-platform code

1.5.30 的新特性

Define the classes and interfaces you are going to implement in the common code.

1. In the `commonMain/kotlin` directory, create the `org.jetbrains.base64` package.
2. Create the `Base64.kt` file in the new package.
3. Define the `Base64Encoder` interface that converts bytes to the `Base64` format:

```
package org.jetbrains.base64

interface Base64Encoder {
    fun encode(src: ByteArray): ByteArray
}
```

4. Define the `Base64Factory` object to provide an instance of the `Base64Encoder` interface to the common code:

```
expect object Base64Factory {
    fun createEncoder(): Base64Encoder
}
```

The factory object is marked with the `expect` keyword in the cross-platform code. For each platform, you should provide an `actual` implementation of the `Base64Factory` object with the platform-specific encoder. Learn more about [platform-specific implementations](#).

提供平台相关实现

Now you will create the `actual` implementations of the `Base64Factory` object for each platform:

- [JVM](#)
- [JS](#)
- [Native](#)

JVM

1. In the `jvmMain/kotlin` directory, create the `org.jetbrains.base64` package.
2. Create the `Base64.kt` file in the new package.
3. Provide a simple implementation of the `Base64Factory` object that delegates to the `java.util.Base64` class:

1.5.30 的新特性

IDEA inspections help create `actual` implementations for an `expect` declaration.



```
package org.jetbrains.base64
import java.util.*

actual object Base64Factory {
    actual fun createEncoder(): Base64Encoder = JvmBase64Encoder
}

object JvmBase64Encoder : Base64Encoder {
    override fun encode(src: ByteArray): ByteArray = Base64.getEncoder().encod
}
```

Pretty simple, right? You've provided a platform-specific implementation by using a straightforward delegation to a third-party implementation.

JS

The JS implementation will be very similar to the JVM one.

1. In the `jsMain/kotlin` directory, create the `org.jetbrains.base64` package.
2. Create the `Base64.kt` file in the new package.
3. Provide a simple implementation of the `Base64Factory` object that delegates to the `btoa()` function.

```
package org.jetbrains.base64

import kotlinx.browser.window

actual object Base64Factory {
    actual fun createEncoder(): Base64Encoder = JsBase64Encoder
}

object JsBase64Encoder : Base64Encoder {
    override fun encode(src: ByteArray): ByteArray {
        val string = src.decodeToString()
        val encodedString = window.btoa(string)
        return encodedString.encodeToByteArray()
    }
}
```

1.5.30 的新特性

Native

Unfortunately, there is no third-party implementation available for all Kotlin/Native targets, so you need to write it yourself.

1. In the `nativeMain/kotlin` directory, create the `org.jetbrains.base64` package.
2. Create the `Base64.kt` file in the new package.
3. Provide your own implementation for the `Base64Factory` object:

```
package org.jetbrains.base64

private val BASE64_ALPHABET: String = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz"
private val BASE64_MASK: Byte = 0x3f
private val BASE64_PAD: Char = '='
private val BASE64_INVERSE_ALPHABET = IntArray(256) {
    BASE64_ALPHABET.indexOf(it.toChar())
}

private fun Int.toBase64(): Char = BASE64_ALPHABET[this]

actual object Base64Factory {
    actual fun createEncoder(): Base64Encoder = NativeBase64Encoder
}

object NativeBase64Encoder : Base64Encoder {
    override fun encode(src: ByteArray): ByteArray {
        fun ByteArray.getOrZero(index: Int): Int = if (index >= size) 0 else this[index]
        // 4n / 3 是预期的 Base64 有效载荷
        val result = ArrayList<Byte>(4 * src.size / 3)
        var index = 0
        while (index < src.size) {
            val symbolsLeft = src.size - index
            val padSize = if (symbolsLeft >= 3) 0 else (3 - symbolsLeft) * 8
            val chunk = (src.getOrZero(index) shl 16) or (src.getOrZero(index + 1) shl 8) or (src.getOrZero(index + 2))
            index += 3

            for (i in 3 downTo padSize) {
                val char = (chunk shr (6 * i)) and BASE64_MASK.toInt()
                result.add(char.toBase64().code.toByte())
            }
            // 使用 '=' 填充
            repeat(padSize) { result.add(BASE64_PAD.code.toByte()) }
        }

        return result.toByteArray()
    }
}
```

Test your library

Now when you have `actual` implementations of the `Base64Factory` object for all platforms, it's time to test your multiplatform library.

To save time on testing, you can write common tests that will be executed on all platforms instead of testing each platform separately.

Prerequisites

Before writing tests, add the `encodeToString` method with the default implementation to the `Base64Encoder` interface, which is defined in

`commonMain/kotlin/org/jetbrains/base64/Base64.kt`. This implementation converts byte arrays to strings, which are much easier to test.

```
interface Base64Encoder {
    fun encode(src: ByteArray): ByteArray

    fun encodeToString(src: ByteArray): String {
        val encoded = encode(src)
        return buildString(encoded.size) {
            encoded.forEach { append(it.toChar()) }
        }
    }
}
```

You can also provide a more efficient implementation of this method for a specific platform, for example, for JVM in `jvmMain/kotlin/org/jetbrains/base64/Base64.kt`:

```
object JvmBase64Encoder : Base64Encoder {
    override fun encode(src: ByteArray): ByteArray = Base64.getEncoder().encode(src)
    override fun encodeToString(src: ByteArray): String = Base64.getEncoder().encod
```

One of the benefits of a multiplatform library is having a default implementation with optional platform-specific overrides.

Write common tests

Now you have a string-based API that you can cover with basic tests.

1. In the `commonTest/kotlin` directory, create the `org.jetbrains.base64` package.

1.5.30 的新特性

2. Create the `Base64Test.kt` file in the new package.

3. Add tests to this file:

```
package org.jetbrains.base64

import kotlin.test.Test

class Base64Test {
    @Test
    fun testEncodeToString() {
        checkEncodeToString("Kotlin is awesome", "S290bGluIGlzIGF3ZXNvbWU=")
    }

    @Test
    fun testPaddedStrings() {
        checkEncodeToString("", "")
        checkEncodeToString("1", "MQ==")
        checkEncodeToString("22", "MjI=")
        checkEncodeToString("333", "MzMz")
        checkEncodeToString("4444", "NDQ0NA==")
    }

    private fun checkEncodeToString(input: String, expectedOutput: String) {
        assertEquals(expectedOutput, Base64Factory.createEncoder().encodeToString(input))
    }

    private fun String.asciiToByteArray() = ByteArray(length) {
        get(it).toByte()
    }
}
```

4. In the Terminal, execute the `check` Gradle task:

```
./gradlew check
```

You can also run the `check` Gradle task by double-clicking it in the list of Gradle tasks.



The tests will run on all platforms (JVM, JS, and Native).

Add platform-specific tests

1.5.30 的新特性

You can also add tests that will be run only for a specific platform. For example, you can add UTF-16 tests on JVM. Just follow the same steps as for common tests, but create the `Base64Test` file in `jvmTest/kotlin/org/jetbrains/base64`:

```
package org.jetbrains.base64

import org.junit.Test
import kotlin.test.assertEquals

class Base64JvmTest {
    @Test
    fun testNonAsciiString() {
        val utf8String = "Gödel"
        val actual = Base64Factory.createEncoder().encodeToString(utf8String.toByte
            assertEquals("R802ZGVs", actual)
    }
}
```

This test will automatically run on the JVM platform in addition to the common tests.

Publish your library to the local Maven repository

Your multiplatform library is ready for publishing so that you can use it in other projects.

To publish your library, use the [maven-publish Gradle plugin](#).

1. In the `build.gradle(.kts)` file, apply the `maven-publish` plugin and specify the group and version of your library:

【Kotlin】

```
plugins {
    kotlin("multiplatform") version "1.6.10"
    id("maven-publish")
}

group = "org.jetbrains.base64"
version = "1.0.0"
```

【Groovy】

1.5.30 的新特性

```
plugins {
    id 'org.jetbrains.kotlin.multiplatform' version '1.6.10'
    id 'maven-publish'
}

group = 'org.jetbrains.base64'
version = '1.0.0'
```

1. In the Terminal, run the `publishToMavenLocal` Gradle task to publish your library to your local Maven repository:

```
./gradlew publishToMavenLocal
```

You can also run the `publishToMavenLocal` Gradle task by double-clicking it in the list of Gradle tasks.



Your library will be published to the local Maven repository.

Add a dependency on the published library

Now you can add your library to other multiplatform projects as a dependency.

Add the `mavenLocal()` repository and add a dependency on your library to the `build.gradle(.kts)` file.

【Kotlin】

```
repositories {
    mavenCentral()
    mavenLocal()
}

kotlin {
    sourceSets {
        val commonMain by getting {
            dependencies {
                implementation("org.jetbrains.base64:Base64:1.0.0")
            }
        }
    }
}
```

【Groovy】

```
repositories {  
    mavenCentral()  
    mavenLocal()  
}  
  
kotlin {  
    sourceSets {  
        commonMain {  
            dependencies {  
                implementation 'org.jetbrains.base64:Base64:1.0.0'  
            }  
        }  
    }  
}
```

Summary

In this tutorial, you:

- Created a multiplatform library with platform-specific implementations.
- Wrote common tests that are executed on all platforms.
- Published your library to the local Maven repository.

下一步做什么？

- Learn more about [publishing multiplatform libraries](#).
- Learn more about [Kotlin Multiplatform](#).
- [Create your first cross-platform mobile application – tutorial](#).
- [Create a full-stack web app with Kotlin Multiplatform – hands-on tutorial](#).

共享代码原则

- 平台间共享代码
- 接入平台相关 API
- 迁移多平台项目到 Kotlin 1.4.0

平台间共享代码

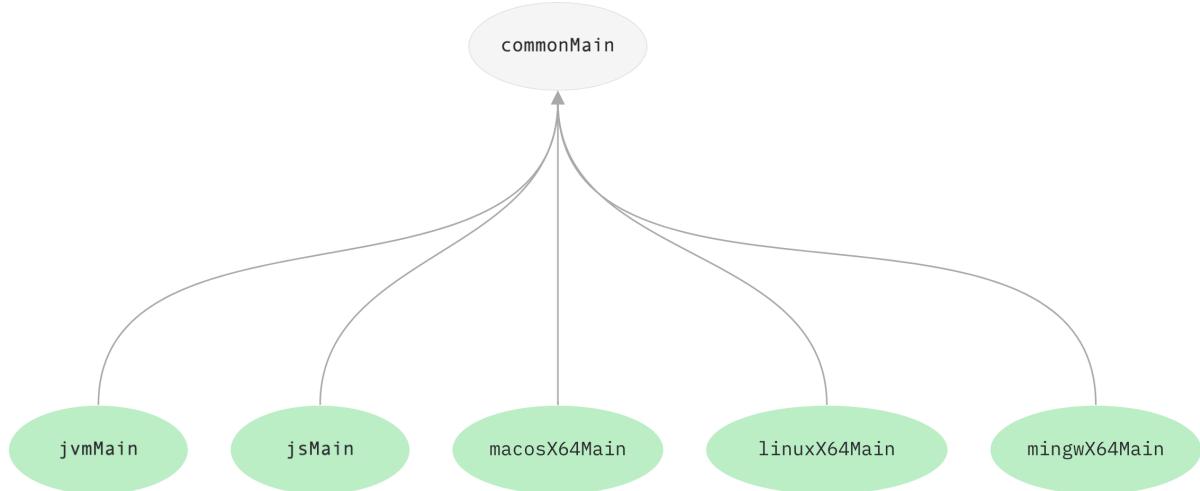
With Kotlin Multiplatform, you can share the code using the mechanisms Kotlin provides:

- Share code among all platforms used in your project. Use it for sharing the common business logic that applies to all platforms.
- Share code among some platforms included in your project but not all. You can reuse much of the code in similar platforms using a hierarchical structure. You can use [target shortcuts](#) for common combinations of targets or [create the hierarchical structure manually](#).

If you need to access platform-specific APIs from the shared code, use the Kotlin mechanism of [expected and actual declarations](#).

对所有平台共享代码

If you have business logic that is common for all platforms, you don't need to write the same code for each platform – just share it in the common source set.



All platform-specific source sets depend on the common source set by default. You don't need to specify any `dependsOn` relations manually for default source sets, such as `jvmMain`, `macosX64Main`, and others.

If you need to access platform-specific APIs from the shared code, use the Kotlin mechanism of [expected and actual declarations](#).

对相似平台共享代码

You often need to create several native targets 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 is for iOS ARM64 devices, the other is 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 the same. So iOS-specific code could be shared between them.

Evidently, 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.

In this case, you can share code across native targets in your project using the hierarchical structure.

To enable the hierarchy structure support, add the following option to your `gradle.properties`.

```
kotlin.mpp.enableGranularSourceSetsMetadata=true
```

There are two ways you can create the hierarchical structure:

- [Use target shortcuts](#) to easily create the hierarchy structure for common combinations of native targets.
- [Configure the hierarchical structure manually](#).

Learn more about [sharing code in libraries](#) and [using Native libraries in the hierarchical structure](#).

1.5.30 的新特性

Due to a [known issue](#), you won't be able to use IDE features, such as code completion and highlighting, for the shared native source set in a multiplatform project with hierarchical structure support if your project depends on:

- Multiplatform libraries that don't support the hierarchical structure.
- Third-party native libraries, with the exception of [platform libraries](#) supported out of the box.

This issue applies only to the shared native source set. The IDE will correctly support the rest of the code.

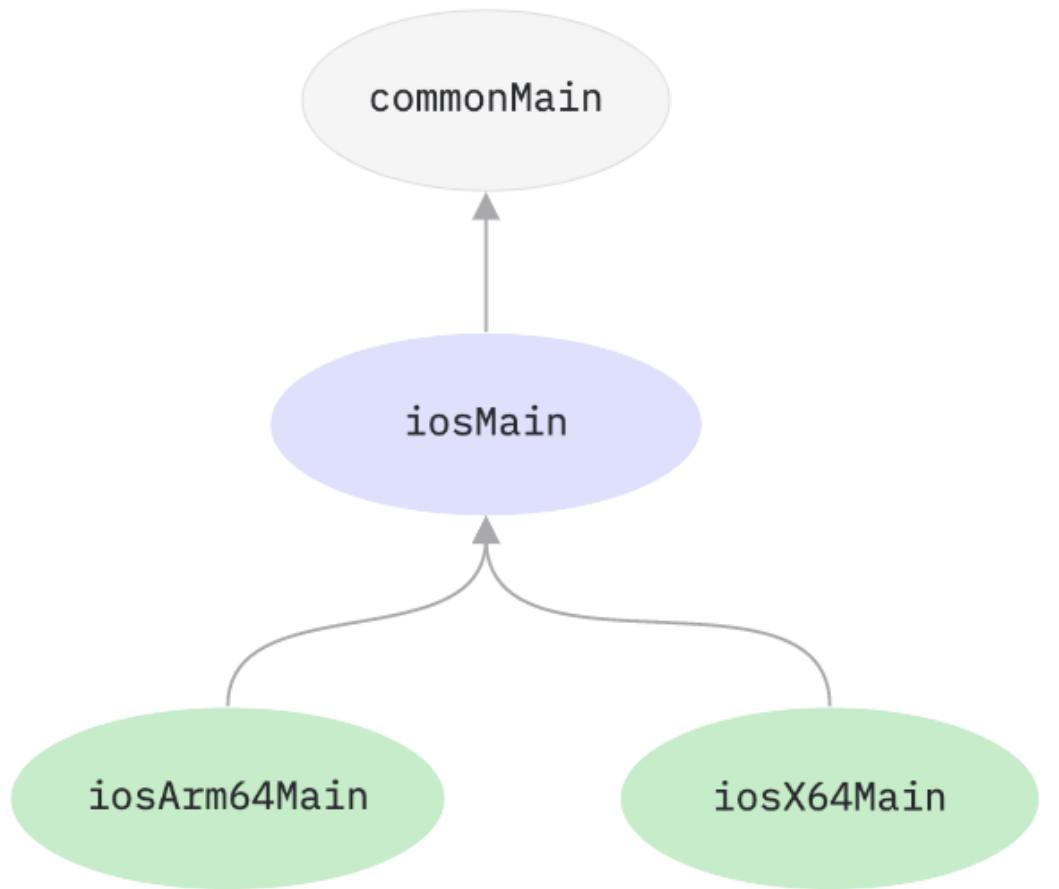
Learn how to [work around this issue](#) for similar source sets, such as `iosArm64` and `iosX64`.



使用目标快捷方式

In a typical multiplatform project with two iOS-related targets – `iosArm64` and `iosX64`, the hierarchical structure includes an intermediate source set (`iosMain`), which is used by the platform-specific source sets.

1.5.30 的新特性



The `kotlin-multiplatform` plugin provides target shortcuts for creating structures for common combinations of targets.

Target shortcut	Targets
<code>ios</code>	<code>iosArm64</code> , <code>iosX64</code>
<code>watchos</code>	<code>watchosArm32</code> , <code>watchosArm64</code> , <code>watchosX64</code>
<code>tvos</code>	<code>tvosArm64</code> , <code>tvosX64</code>

All shortcuts create similar hierarchical structures in the code. For example, the `ios` shortcut creates the following hierarchical structure:

【Kotlin】

1.5.30 的新特性

```
kotlin {  
    sourceSets{  
        val commonMain by sourceSets.getting  
        val iosX64Main by sourceSets.getting  
        val iosArm64Main by sourceSets.getting  
        val iosMain by sourceSets.creating {  
            dependsOn(commonMain)  
            iosX64Main.dependsOn(this)  
            iosArm64Main.dependsOn(this)  
        }  
    }  
}
```

【Groovy】

```
kotlin {  
    sourceSets{  
        iosMain {  
            dependsOn(commonMain)  
            iosX64Main.dependsOn(it)  
            iosArm64Main.dependsOn(it)  
        }  
    }  
}
```

Target shortcuts and ARM64 (Apple Silicon) simulators

The target shortcuts `ios`, `watchos`, and `tvos` don't include the simulator targets for ARM64 (Apple Silicon) platforms: `iosSimulatorArm64`, `watchosSimulatorArm64`, and `tvosSimulatorArm64`. If you use the target shortcuts and want to build the project for an Apple Silicon simulator, adjust the build script the following way:

1. Add the `*SimulatorArm64` simulator target you need.
2. Connect the simulator target with the shortcut using the source set dependencies (`dependsOn`).

【Kotlin】

1.5.30 的新特性

```
kotlin {
    ios()
    // Add the ARM64 simulator target
    iosSimulatorArm64()

    val iosMain by sourceSets.getting
    val iosTest by sourceSets.getting
    val iosSimulatorArm64Main by sourceSets.getting
    val iosSimulatorArm64Test by sourceSets.getting

    // Set up dependencies between the source sets
    iosSimulatorArm64Main.dependsOn(iosMain)
    iosSimulatorArm64Test.dependsOn(iosTest)
}
```

【Groovy】

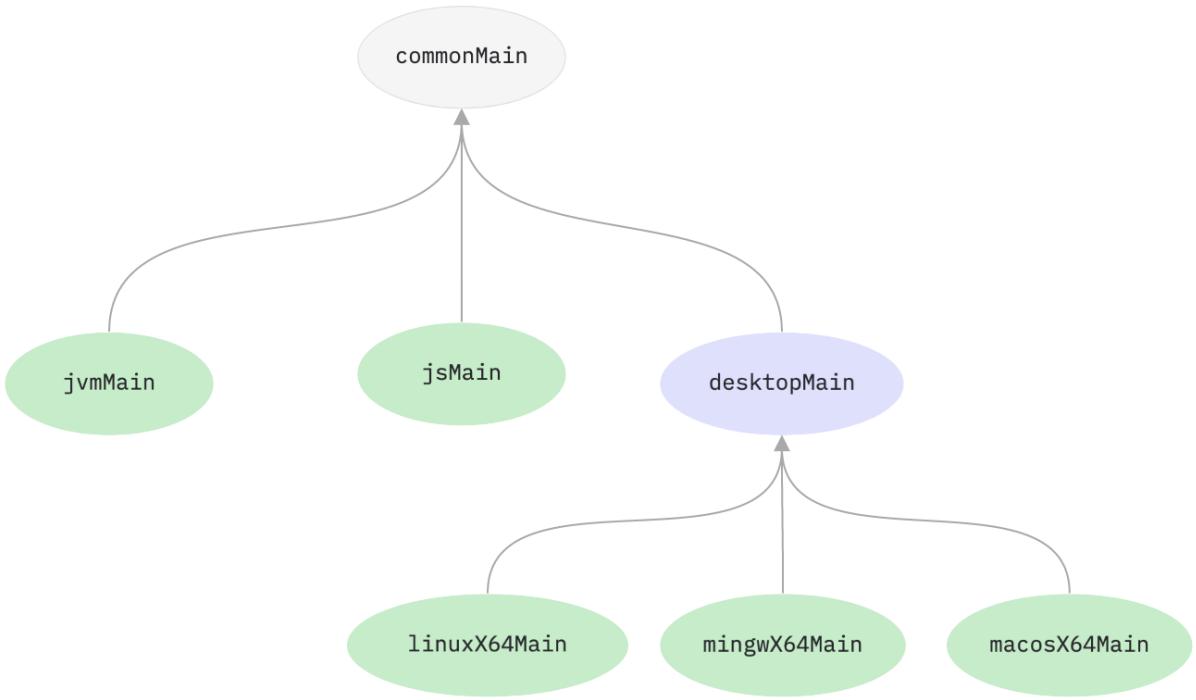
```
kotlin {
    ios()
    // Add the ARM64 simulator target
    iosSimulatorArm64()

    // Set up dependencies between the source sets
    sourceSets {
        // ...
        iosSimulatorArm64Main {
            dependsOn(iosMain)
        }
        iosSimulatorArm64Test {
            dependsOn(iosTest)
        }
    }
}
```

手动配置层次结构

To create the hierarchical structure manually, introduce an intermediate source set that holds the shared code for several targets and create a structure of the source sets including the intermediate one.

1.5.30 的新特性



For example, if you want to share code among native Linux, Windows, and macOS targets – `linuxX64M` , `mingwX64` , and `macosX64` :

1. Add the intermediate source set `desktopMain` that holds the shared logic for these targets.
2. Specify the hierarchy of source sets using 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】

1.5.30 的新特性

```
kotlin {  
    sourceSets {  
        desktopMain {  
            dependsOn(commonMain)  
        }  
        linuxX64Main {  
            dependsOn(desktopMain)  
        }  
        mingwX64Main {  
            dependsOn(desktopMain)  
        }  
        macosX64Main {  
            dependsOn(desktopMain)  
        }  
    }  
}
```

You can have a shared source set for the following combinations of targets:

- JVM + JS + Native
- JVM + Native
- JS + Native
- JVM + JS
- Native

Kotlin doesn't currently support sharing a source set for these combinations:

- Several JVM targets
- JVM + Android targets
- Several JS targets

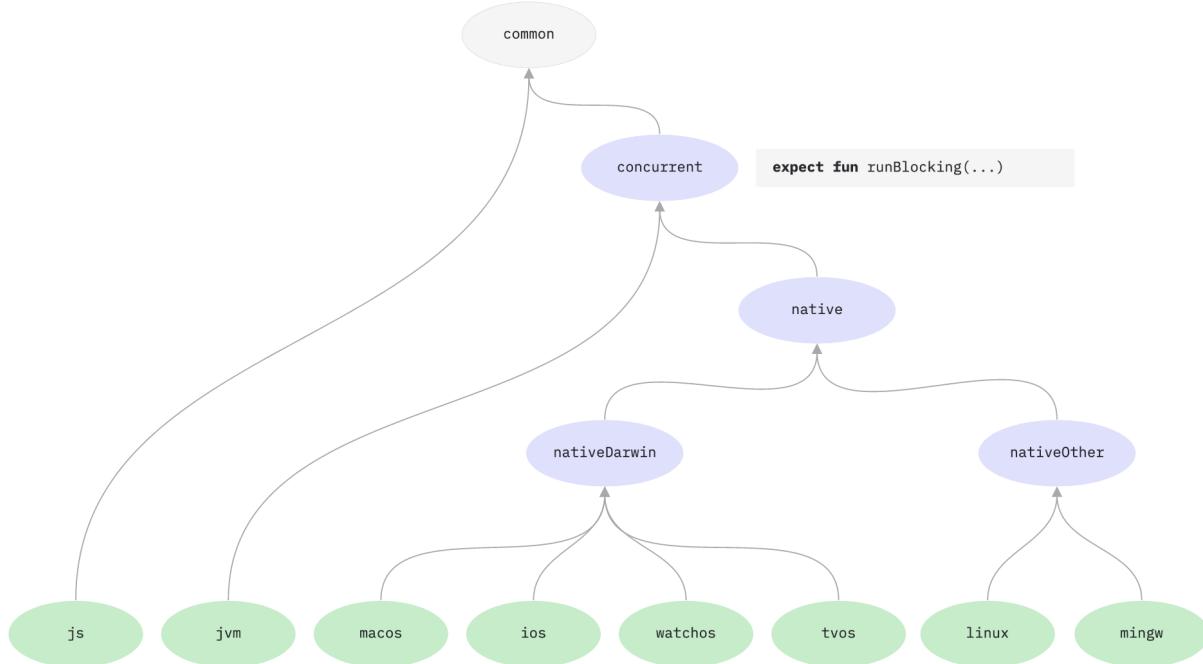
If you need to access platform-specific APIs from a shared native source set, IntelliJ IDEA will help you detect common declarations that you can use in the shared native code. For other cases, use the Kotlin mechanism of [expected and actual declarations](#).

在库中共享代码

Thanks to the hierarchical project structure, libraries can also provide common APIs for a subset of targets. When a [library is published](#), the API of its intermediate source sets is embedded into the library artifacts along with information about the project structure. When you use this library, the intermediate source sets of your project access only those APIs of the library which are available to the targets of each source set.

1.5.30 的新特性

For example, check out the following source set hierarchy from the `kotlinx.coroutines` repository:



The `concurrent` source set declares the function `runBlocking` and is compiled for the JVM and the native targets. Once the `kotlinx.coroutines` library is updated and published with the hierarchical project structure, you can depend on it and call `runBlocking` from a source set that is shared between the JVM and native targets since it matches the “targets signature” of the library’s `concurrent` source set.

在层次结构中使用原生库

You can use platform-dependent libraries like Foundation, UIKit, and POSIX in source sets shared among several native targets. This helps 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.

To enable usage of platform-dependent libraries in shared source sets, add the following to your `gradle.properties`:

```
kotlin.mpp.enableGranularSourceSetsMetadata=true  
kotlin.native.enableDependencyPropagation=false
```

1.5.30 的新特性

In addition to [platform libraries](#) shipped with Kotlin/Native, this approach can also handle custom [cinterop libraries](#) making them available in shared source sets. To enable this support, specify the additional `kotlin.mpp.enableCInteropCommonization` key:

```
kotlin.mpp.enableCInteropCommonization=true
```

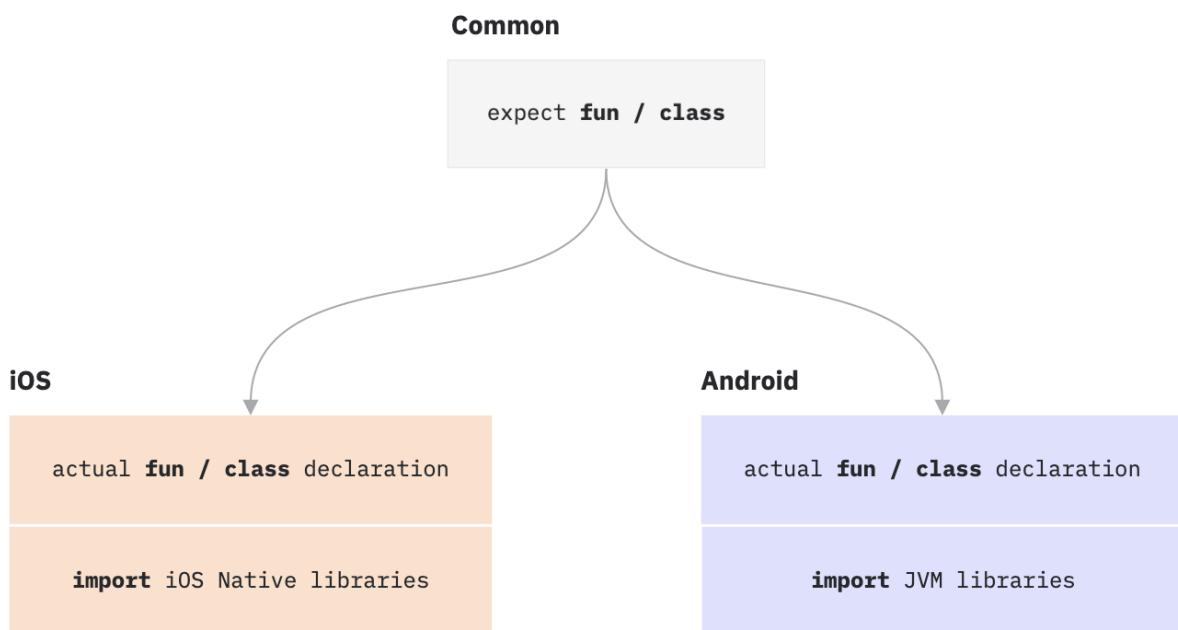
接入平台相关 API

The `expect / actual` feature is in [Beta](#). It 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.



If you're developing a multiplatform application that needs to access platform-specific APIs that implement the required functionality (for example, [generating a UUID](#)), use the Kotlin mechanism of *expected and actual declarations*.

With this mechanism, a common source set defines an *expected declaration*, and platform source sets must provide the *actual declaration* that corresponds to the expected declaration. This works for most Kotlin declarations, such as functions, classes, interfaces, enumerations, properties, and annotations.



The compiler ensures that every declaration marked with the `expect` keyword in the common module has the corresponding declarations marked with the `actual` keyword in all platform modules. The IDE provides tools that help you create the missing actual declarations.

1.5.30 的新特性

Use expected and actual declarations only for Kotlin declarations that have platform-specific dependencies. Implementing as much functionality as possible in the shared module is better, even if doing so takes more time.

Don't overuse expected and actual declarations – in some cases, an [interface](#) may be a better choice because it is more flexible and easier to test.



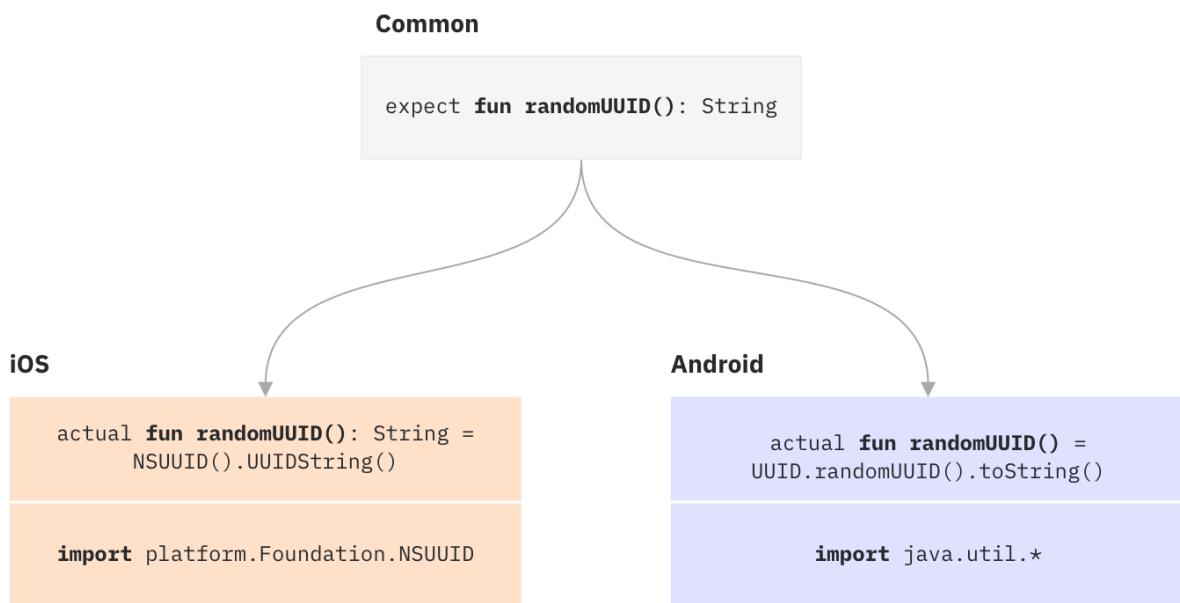
Learn how to [add dependencies on platform-specific libraries](#).

Examples

For simplicity, the following examples use intuitive target names, like iOS and Android. However, in your Gradle build files, you need to use a specific target name from [the list of supported targets](#).

Generate a UUID

Let's assume that you are developing iOS and Android applications using Kotlin Multiplatform Mobile and you want to generate a universally unique identifier (UUID):



For this purpose, declare the expected function `randomUUID()` with the `expect` keyword in the common module. Don't include any implementation code.

1.5.30 的新特性

```
// Common  
expect fun randomUUID(): String
```

In each platform-specific module (iOS and Android), provide the actual implementation for the function `randomUUID()` expected in the common module. Use the `actual` keyword to mark the actual implementation.

The following examples show the implementation of this for Android and iOS. Platform-specific code uses the `actual` keyword and the expected name for the function.

```
// Android  
import java.util.*  
  
actual fun randomUUID() = UUID.randomUUID().toString()
```

```
// iOS  
import platform.Foundation.NSUUID  
  
actual fun randomUUID(): String = NSUUID().UUIDString()
```

Implement a logging framework

Another example of code sharing and interaction between the common and platform logic, JS and JVM in this case, in a minimalistic logging framework:

```
// Common  
enum class LogLevel {  
    DEBUG, WARN, ERROR  
}  
  
internal expect fun writeLogMessage(message: String, logLevel: LogLevel)  
  
fun logDebug(message: String) = writeLogMessage(message, LogLevel.DEBUG)  
fun logWarn(message: String) = writeLogMessage(message, LogLevel.WARN)  
fun logError(message: String) = writeLogMessage(message, LogLevel.ERROR)
```

```
// JVM  
internal actual fun writeLogMessage(message: String, logLevel: LogLevel) {  
    println("[${logLevel}]: $message")  
}
```

1.5.30 的新特性

For JavaScript, a completely different set of APIs is available, and the `actual` declaration will look like this.

```
// JS
internal actual fun writeLogMessage(message: String, logLevel: LogLevel) {
    when (logLevel) {
        LogLevel.DEBUG -> console.log(message)
        LogLevel.WARN -> console.warn(message)
        LogLevel.ERROR -> console.error(message)
    }
}
```

Send and receive messages from a WebSocket

Consider developing a chat platform for iOS and Android using Kotlin Multiplatform Mobile. Let's see how you can implement sending and receiving messages from a WebSocket.

For this purpose, define a common logic that you don't need to duplicate in all platform modules – just add it once to the common module. However, the actual implementation of the WebSocket class differs from platform to platform. That's why you should use `expect / actual` declarations for this class.

In the common module, declare the expected class `PlatformSocket()` with the `expect` keyword. Don't include any implementation code.

```
//Common
internal expect class PlatformSocket(
    url: String
) {
    fun openSocket(listener: PlatformSocketListener)
    fun closeSocket(code: Int, reason: String)
    fun sendMessage(msg: String)
}

interface PlatformSocketListener {
    fun onOpen()
    fun onFailure(t: Throwable)
    fun onMessage(msg: String)
    fun onClosing(code: Int, reason: String)
    fun onClosed(code: Int, reason: String)
}
```

1.5.30 的新特性

In each platform-specific module (iOS and Android), provide the actual implementation for the class `PlatformSocket()` expected in the common module. Use the `actual` keyword to mark the actual implementation.

The following examples show the implementation of this for Android and iOS.

```
//Android
import okhttp3.OkHttpClient
import okhttp3.Request
import okhttp3.Response
import okhttp3.WebSocket

internal actual class PlatformSocket actual constructor(url: String) {
    private val socketEndpoint = url
    private var webSocket: WebSocket? = null
    actual fun openSocket(listener: PlatformSocketListener) {
        val socketRequest = Request.Builder().url(socketEndpoint).build()
        val webClient = OkHttpClient().newBuilder().build()
        webSocket = webClient.newWebSocket(
            socketRequest,
            object : okhttp3.WebSocketListener() {
                override fun onOpen(webSocket: WebSocket, response: Response) =
                    super.onOpen(webSocket, response)
                override fun onFailure(webSocket: WebSocket, t: Throwable, response: Response?) =
                    super.onFailure(webSocket, t, response)
                override fun onMessage(webSocket: WebSocket, text: String) = listener.onMessage(text)
                override fun onClosing(webSocket: WebSocket, code: Int, reason: String) =
                    super.onClosing(webSocket, code, reason)
                override fun onClosed(webSocket: WebSocket, code: Int, reason: String) =
                    super.onClosed(webSocket, code, reason)
            }
        )
    }
    actual fun closeSocket(code: Int, reason: String) {
        webSocket?.close(code, reason)
        webSocket = null
    }
    actual fun sendMessage(msg: String) {
        webSocket?.send(msg)
    }
}
```

Android implementation uses the third-party library [OkHttp](#). Add the corresponding dependency to `build.gradle(.kts)` in the shared module:

【Kotlin】

1.5.30 的新特性

```
sourceSets {  
    val androidMain by getting {  
        dependencies {  
            implementation("com.squareup.okhttp3:okhttp:$okhttp_version")  
        }  
    }  
}
```

【Groovy】

```
commonMain {  
    dependencies {  
        implementation "com.squareup.okhttp3:okhttp:$okhttp_version"  
    }  
}
```

iOS implementation uses `NSURLSession` from the standard Apple SDK and doesn't require additional dependencies.

1.5.30 的新特性

```
//iOS
import platform.Foundation.*
import platform.darwin.NSObject

internal actual class PlatformSocket actual constructor(url: String) {
    private val socketEndpoint = NSURL.URLWithString(url)!!
    private var webSocket: NSURLSessionWebSocketTask? = null
    actual fun openSocket(listener: PlatformSocketListener) {
        val urlSession = NSURLSession.sessionWithConfiguration(
            configuration = NSURLSessionConfiguration.defaultSessionConfiguration()
        )
        delegate = object : NSObject(), NSURLSessionWebSocketDelegateProtocol {
            override fun URLSession(
                session: NSURLSession,
                webSocketTask: NSURLSessionWebSocketTask,
                didOpenWithProtocol: String?
            ) {
                listener.onOpen()
            }
            override fun URLSession(
                session: NSURLSession,
                webSocketTask: NSURLSessionWebSocketTask,
                didCloseWithCode: NSURLSessionWebSocketCloseCode,
                reason: NSData?
            ) {
                listener.onClosed(didCloseWithCode.toInt(), reason.toString())
            }
        },
        delegateQueue = NSOperationQueue.currentQueue()
    }
    webSocket = urlSession.webSocketTaskWithURL(socketEndpoint)
    listenMessages(listener)
    webSocket?.resume()
}

private fun listenMessages(listener: PlatformSocketListener) {
    webSocket?.receiveMessageWithCompletionHandler { message, nsError ->
        when {
            nsError != null -> {
                listener.onFailure(Throwable(nsError.description))
            }
            message != null -> {
                message.string?.let { listener.onMessage(it) }
            }
        }
        listenMessages(listener)
    }
}
actual fun closeSocket(code: Int, reason: String) {
    webSocket?.cancelWithCloseCode(code.toLong(), null)
    webSocket = null
}
```

1.5.30 的新特性

```
actual fun sendMessage(msg: String) {
    val message = NSURLSessionWebSocketMessage(msg)
    webSocket?.sendMessage(message) { err ->
        err?.let { println("send $msg error: $it") }
    }
}
```

And here is the common logic in the common module that uses the platform-specific class `PlatformSocket()`.

1.5.30 的新特性

```
//Common
class AppSocket(url: String) {
    private val ws = PlatformSocket(url)
    var socketError: Throwable? = null
        private set
    var currentState: State = State.CLOSED
        private set(value) {
            field = value
            stateListener?.invoke(value)
        }
    var stateListener: ((State) -> Unit)? = null
        set(value) {
            field = value
            value?.invoke(currentState)
        }
    var messageListener: ((msg: String) -> Unit)? = null
    fun connect() {
        if (currentState != State.CLOSED) {
            throw IllegalStateException("The socket is available.")
        }
        socketError = null
        currentState = State.CONNECTING
        ws.openSocket(socketListener)
    }
    fun disconnect() {
        if (currentState != State.CLOSED) {
            currentState = State.CLOSING
            ws.closeSocket(1000, "The user has closed the connection.")
        }
    }
    fun send(msg: String) {
        if (currentState != State.CONNECTED) throw IllegalStateException("The connection is not established")
        ws.sendMessage(msg)
    }
    private val socketListener = object : PlatformSocketListener {
        override fun onOpen() {
            currentState = State.CONNECTED
        }
        override fun onFailure(t: Throwable) {
            socketError = t
            currentState = State.CLOSED
        }
        override fun onMessage(msg: String) {
            messageListener?.invoke(msg)
        }
        override fun onClosing(code: Int, reason: String) {
            currentState = State.CLOSING
        }
        override fun onClosed(code: Int, reason: String) {
            currentState = State.CLOSED
        }
    }
}
```

1.5.30 的新特性

```
    }
}

enum class State {
    CONNECTING,
    CONNECTED,
    CLOSING,
    CLOSED
}
}
```

Rules for expected and actual declarations

The main rules regarding expected and actual declarations are:

- An expected declaration is marked with the `expect` keyword; the actual declaration is marked with the `actual` keyword.
- `expect` and `actual` declarations have the same name and are located in the same package (have the same fully qualified name).
- `expect` declarations never contain any implementation code and are abstract by default.
- In interfaces, functions in `expect` declarations cannot have bodies, but their `actual` counterparts can be non-abstract and have a body. It allows the inheritors not to implement a particular function.

To indicate that common inheritors don't need to implement a function, mark it as `open`. All its `actual` implementations will be required to have a body:

```
// Common
expect interface Mascot {
    open fun display(): String
}

class MascotImpl : Mascot {
    // it's ok not to implement `display()`: all `actual`s are guaranteed to have bodies
}

// Platform-specific
actual interface Mascot {
    actual fun display(): String {
        TODO()
    }
}
```

1.5.30 的新特性

During each platform compilation, the compiler ensures that every declaration marked with the `expect` keyword in the common or intermediate source set has the corresponding declarations marked with the `actual` keyword in all platform source sets. The IDE provides tools that help you create the missing actual declarations.

If you have a platform-specific library that you want to use in shared code while providing your own implementation for another platform, you can provide a `typealias` to an existing class as the actual declaration:

```
expect class AtomicRef<V>(value: V) {  
    fun get(): V  
    fun set(value: V)  
    fun getAndSet(value: V): V  
    fun compareAndSet(expect: V, update: V): Boolean  
}
```

```
actual typealias AtomicRef<V> = java.util.concurrent.atomic.AtomicReference<V>
```

迁移多平台项目到 Kotlin 1.4.0

Kotlin 1.4.0 comes with lots of features and improvements in the tooling for multiplatform programming. Some of them just work out of the box on existing projects, and some require additional configuration steps. This guide will help you migrate your multiplatform projects to 1.4.0 or higher and get the benefits of all its new features.

对于多平台项目作者

更新 Gradle

Starting with 1.4.0, Kotlin multiplatform projects require Gradle 6.0 or later. Make sure that your projects use the proper version of Gradle and upgrade it if needed. See the [Gradle documentation](#) for non-Kotlin-specific migration instructions.

简化构建配置

Gradle module metadata provides rich publishing and dependency resolution features that are used in Kotlin Multiplatform Projects. In Gradle 6.0 and above, module metadata is used in dependency resolution and included in publications by default. Thus, once you update to such a version, you can remove

```
enableFeaturePreview("GRADLE_METADATA")
```

 from the project's `settings.gradle` file.

If you use libraries published with metadata, you only have to specify dependencies on them only once in the shared source set, as opposed to specifying dependencies on different variants of the same library in the shared and platform-specific source sets prior to 1.4.0.

Starting from 1.4.0, you also no longer need to declare a dependency on `stdlib` in each source set manually – it [will now be added by default](#). The version of the automatically added standard library will be the same as the version of the Kotlin Gradle plugin, since they have the same versioning.

With these features, you can make your Gradle build file much more concise and easy to read:

1.5.30 的新特性

```
//...
android()
ios()
js()

sourceSets {
    commonMain {
        dependencies {
            implementation("org.jetbrains.kotlinx:kotlinx-coroutines-core:$coroutineVersion")
        }
    }
}
//...
```

Don't use kotlinx library artifact names with suffixes `-common` or `-native`, as they are no longer supported. Instead, use the library root artifact name, which in the example above is `kotlinx-coroutines-core`.

尝试分层项目结构

With [the new hierarchical project structure support](#), you can share code among several targets in a multiplatform project. 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.

By enabling the hierarchical structure along with its ability to use platform-dependent libraries in shared source sets, you can eliminate the need to use certain workarounds to get IDE support for sharing source sets among several native targets, for example `iosArm64` and `iosX64`:

```
kotlin {
    // workaround 1: select iOS target platform depending on the Xcode environment
    val iosTarget: (String, KotlinNativeTarget.() -> Unit) -> KotlinNativeTarget =
        if (System.getenv("SDK_NAME")?.startsWith("iphoneos") == true)
            ::iosArm64
        else
            ::iosX64

    iosTarget("ios")
}
```

1.5.30 的新特性

```
# workaround 2: make symbolic links to use one source set for two targets
ln -s iosMain iosArm64Main && ln -s iosMain iosX64Main
```

Instead of doing this, you can create a hierarchical structure with [target shortcuts](#) available for typical multi-target scenarios, or you can manually declare and connect the source sets. For example, you can create two iOS targets and a shared source set with the `ios()` shortcut:

```
kotlin {
    ios() // iOS device and simulator targets; iosMain and iosTest source sets
}
```

To enable the hierarchical project structure along with the use of platform-dependent libraries in shared source sets, just add the following to your `gradle.properties`:

```
kotlin.mpp.enableGranularSourceSetsMetadata=true
kotlin.native.enableDependencyPropagation=false
```

In future versions, the hierarchical project structure will become default for Kotlin multiplatform project, so we strongly encourage you to start using it now.

对于库作者

Migrate from Gradle Bintray plugin to Maven Publish plugin

If you're using `gradle-bintray-plugin` for library publication, migrate your projects to `maven-publish` plugin instead. See how we've done this for [kotlinx.serialization](#). Learn more about [publishing multiplatform libraries](#).

If for some reason you need to publish to Bintray and use the Gradle Bintray plugin, remember that this plugin doesn't support publishing Gradle module metadata. Use [this workaround](#) to fix this.

遵循默认库的布局

1.5.30 的新特性

The layout of kotlinx libraries has changed and now corresponds to the default layout, which we recommend using: The *root* or *umbrella* library module now has a name without a suffix (for example, `kotlinx-coroutines-core` instead of `kotlinx-coroutines-core-native`). Publishing libraries with [maven-publish Gradle plugin](#) follows this layout by default. Learn more about [publishing multiplatform libraries](#).

迁移到分层项目结构

A hierarchical project structure allows reusing code in similar targets, as well as publishing and consuming libraries with granular APIs targeting similar platforms. We recommend that you switch to the hierarchical project structure in your libraries when migrating to Kotlin 1.4.0 or higher:

- By default, libraries published with the hierarchical project structure are compatible only with projects that have hierarchical project structure. To enable compatibility with non-hierarchical projects, add the following to the `gradle.properties` file in your library project:

```
kotlin.mpp.enableCompatibilityMetadataVariant=true
```

- Libraries published without the hierarchical project structure can't be used in a shared native source set. For example, users with `ios()` shortcuts in their `build.gradle.(kts)` files won't be able to use your library in their iOS-shared code.

The compatibility between multiplatform projects and libraries is as follows:

Library with hierarchical project structure	Project with hierarchical project structure	Compatibility
Yes	Yes	<input checked="" type="checkbox"/>
Yes	No	Need to enable with <code>enableCompatibilityMetadataVariant</code>
No	Yes	Library can't be used in a shared native source set
No	Yes	<input checked="" type="checkbox"/>

In future versions, the hierarchical project structure with the usage of platform-dependent libraries in shared source sets will be the default in multiplatform projects. So the sooner you support it, the sooner users will be able to migrate. We'll also be

1.5.30 的新特性

我们很感激你报告任何你在我们的问题追踪器中发现的 bug。

To enable hierarchical project structure support, add the following to your `gradle.properties` file:

```
kotlin.mpp.enableGranularSourceSetsMetadata=true  
kotlin.mpp.enableCompatibilityMetadataVariant=true // to enable compatibility with
```

对于构建作者

检查任务名称

The introduction of the hierarchical project structure in multiplatform projects resulted in a couple of changes to the names of some Gradle tasks:

- The `metadataJar` task has been renamed to `allMetadataJar`.
- There are new `compile<SourceSet>KotlinMetadata` tasks for all published intermediate source sets.

These changes are relevant only for projects with the hierarchical project structure.

对于 Kotlin/JS 目标

与 npm 依赖关系管理相关的变更

When declaring dependencies on npm packages, you are now required to explicitly specify a version or version range based on [npm's semver syntax](#). Specifying multiple version ranges is also supported.

While we don't recommend it, you can use a wildcard `*` in place of a version number if you do not want to specify a version or version range explicitly.

与 Kotlin/JS IR 编译器相关的变更

Kotlin 1.4.0 introduces the Alpha IR compiler for Kotlin/JS. Learn more about the [Kotlin/JS IR compiler's backend and how to configure it](#).

1.5.30 的新特性

To choose between the different Kotlin/JS compiler options, set the key

`kotlin.js.compiler` in your `gradle.properties` to `legacy`, `ir`, or `both`. Alternatively, pass `LEGACY`, `IR`, or `BOTH` to the `js` function in your `build.gradle(.kts)`.

```
kotlin {  
    js(IR) { // or: LEGACY, BOTH  
        // ...  
    }  
    binaries.executable()  
}
```

both 模式的变更

Choosing `both` as the compiler option (so that it will compile with both the legacy and the IR backend) means that some Gradle tasks are renamed to explicitly mark them as only affecting the legacy compilation. `compileKotlinJs` is renamed to `compileKotlinJsLegacy`, and `compileTestKotlinJs` is renamed to `compileTestKotlinJsLegacy`.

显式切换可执行文件的创建

When using the IR compiler, the `binaries.executable()` instruction must be present in the `js` target configuration block of your `build.gradle(.kts)`. If this option is omitted, only Kotlin-internal library files are generated. These files can be used from other projects, but not run on their own.

For backwards compatibility, when using the legacy compiler for Kotlin/JS, including or omitting `binaries.executable()` will have no effect – executable files will be generated in either case. To make the legacy backend stop producing executable files without the presence of `binaries.executable()` (for example, to improve build times where runnable artifacts aren't required), set `kotlin.js.generate.executable.default=false` in your `gradle.properties`.

与 Dukat 相关的变更

The Dukat integration for Gradle has received minor naming and functionality changes with Kotlin 1.4.0.

- The `kotlin.js.experimental.generateKotlinExternals` flag has been renamed to `kotlin.js.generate.externals`. It controls the default behavior of Dukat for all

1.5.30 的新特性

specified npm dependencies.

- The `npm` dependency function now takes a third parameter after the package name and version: `generateExternals`. This allows you to individually control whether Dukat should generate declarations for a specific dependency, and it overrides the `generateKotlinExternals` setting.

Learn how to [manually trigger the generation of Kotlin externals](#).

在 Kotlin 1.3.x 项目中使用 Kotlin 1.4.x 构建的构件

The choice between the `IR` and `LEGACY` compilers was not yet available in Kotlin 1.3.xx. Because of this, you may encounter a Gradle error `Cannot choose between the following variants...` if one of your dependencies (or any transitive dependency) was built using Kotlin 1.4+ but your project uses Kotlin 1.3.xx. A workaround is provided [here](#).

添加依赖项

- [添加依赖项](#)
- [添加 Android 依赖项](#)
- [添加 iOS 依赖项](#)

Adding dependencies on multiplatform libraries

Every program requires a set of libraries to operate successfully. A Kotlin Multiplatform project can depend on multiplatform libraries that work for all target platforms, platform-specific libraries, and other multiplatform projects.

To add a dependency on a library, set a dependency of the required [type](#) (for example, `implementation`) in the `dependencies` block in your [Gradle](#) build script.

【Kotlin】

```
kotlin {  
    sourceSets {  
        val commonMain by getting {  
            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 the top level](#).

Dependency on a Kotlin library

Standard library

1.5.30 的新特性

A dependency on a standard library (`stdlib`) in each source set is added automatically. The version of the standard library is the same as the version of the `kotlin-multiplatform` 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 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](#).

Test libraries

The `kotlin.test` API is available for multiplatform tests. When you [create a multiplatform project](#), the Project Wizard automatically adds test dependencies to common and platform-specific source sets.

If you didn't use the Project Wizard to create your project, you can [add the dependencies manually](#).

kotlinx libraries

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 base artifact name, such as `kotlinx-coroutines-core`.

【Kotlin】

```
kotlin {  
    sourceSets {  
        val commonMain by getting {  
            dependencies {  
                implementation("org.jetbrains.kotlinx:kotlinx-coroutines-core:1.6.0")  
            }  
        }  
    }  
}
```

【Groovy】

1.5.30 的新特性

```
kotlin {  
    sourceSets {  
        commonMain {  
            dependencies {  
                implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-core:1.6.0'  
            }  
        }  
    }  
}
```

If you use a kotlinx library and need a [platform-specific dependency](#), you can use platform-specific variants of libraries with suffixes such as `-jvm` or `-js`, for example, `kotlinx-coroutines-core-jvm`.

【Kotlin】

```
kotlin {  
    sourceSets {  
        val jvmMain by getting {  
            dependencies {  
                implementation("org.jetbrains.kotlinx:kotlinx-coroutines-core-jvm:1")  
            }  
        }  
    }  
}
```

【Groovy】

```
kotlin {  
    sourceSets {  
        jvmMain {  
            dependencies {  
                implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-core-jvm:1'  
            }  
        }  
    }  
}
```

Dependency on Kotlin Multiplatform libraries

1.5.30 的新特性

You can add dependencies on libraries that have adopted Kotlin Multiplatform technology, such as [SQLDelight](#). The authors of these libraries usually provide guides for adding their dependencies to your project.

Check out this [community-maintained list of Kotlin Multiplatform libraries](#).

Library shared for all source sets

If you want to use a library from all source sets, you can add it only to the common source set. The Kotlin Multiplatform Mobile plugin will automatically add the corresponding parts to any other source sets.

【Kotlin】

```
kotlin {  
    sourceSets["commonMain"].dependencies {  
        implementation("io.ktor:ktor-client-core:1.6.7")  
    }  
    sourceSets["androidMain"].dependencies {  
        //dependency to platform part of kotlinx.coroutines will be added automatically  
    }  
}
```

【Groovy】

```
kotlin {  
    sourceSets {  
        commonMain {  
            dependencies {  
                implementation 'io.ktor:ktor-client-core:1.6.7'  
            }  
        }  
        androidMain {  
            dependencies {  
                //dependency to platform part of kotlinx.coroutines will be added automatically  
            }  
        }  
    }  
}
```

Library used in specific source sets

1.5.30 的新特性

If you want to use a multiplatform library just for specific source sets, you can add it exclusively to them. The specified library declarations will then be available only in those source sets.

Don't use a platform-specific name in such cases, like SQLDelight `native-driver` in the example below. Find the exact name in the library's documentation.



【Kotlin】

```
kotlin {  
    sourceSets["commonMain"].dependencies {  
        // kotlinx.coroutines will be available in all source sets  
        implementation("org.jetbrains.kotlinx:kotlinx-coroutines-core:1.6.0")  
    }  
    sourceSets["androidMain"].dependencies {  
    }  
    sourceSets["iosX64Main"].dependencies {  
        // SQLDelight will be available only in the iOS source set, but not in Android  
        implementation("com.squareup.sqlodelight:native-driver:1.4.1")  
    }  
}
```

【Groovy】

```
kotlin {  
    sourceSets {  
        commonMain {  
            dependencies {  
                // kotlinx.coroutines will be available in all source sets  
                implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-core:1.6.0'  
            }  
        }  
        androidMain {  
            dependencies {}  
        }  
        iosMain {  
            dependencies {  
                // SQLDelight will be available only in the iOS source set, but not in Android  
                implementation 'com.squareup.sqlodelight:native-driver:1.4.1'  
            }  
        }  
    }  
}
```

1.5.30 的新特性

When using a multiplatform library that does not have [hierarchical structure support](#) in a multiplatform project that does, you won't be able to use IDE features, such as code completion and highlighting, for the shared iOS source set.

This is a [known issue](#), and we are working on resolving it. In the meantime, you can use [this workaround](#).



Dependency on another multiplatform project

You can connect one multiplatform project to another as a dependency. To do this, simply add a project dependency to the source set that needs it. If you want to use a dependency in all source sets, add it to the common one. In this case, other source sets will get their versions automatically.

【Kotlin】

```
kotlin {  
    sourceSets["commonMain"].dependencies {  
        implementation(project(":some-other-multiplatform-module"))  
    }  
    sourceSets["androidMain"].dependencies {  
        //platform part of :some-other-multiplatform-module will be added automatic  
    }  
}
```

【Groovy】

1.5.30 的新特性

```
kotlin {  
    sourceSets {  
        commonMain {  
            dependencies {  
                implementation project(':some-other-multiplatform-module')  
            }  
        }  
        androidMain {  
            dependencies {  
                //platform part of :some-other-multiplatform-module will be added a  
            }  
        }  
    }  
}
```

What's next?

Check out other resources on adding dependencies in multiplatform projects and learn more about:

- [Adding Android dependencies](#)
- [Adding iOS dependencies](#)

添加 Android 依赖项

The workflow for adding Android-specific dependencies to a Kotlin Multiplatform module is the same as it is for pure Android projects: declare the dependency in your Gradle file and import the project. After that, you can use this dependency in your Kotlin code.

We recommend declaring Android dependencies in Multiplatform Mobile projects by adding them to a specific Android source set:

【Kotlin】

```
sourceSets["androidMain"].dependencies {  
    implementation("com.example.android:app-magic:12.3")  
}
```

【Groovy】

```
sourceSets {  
    androidMain {  
        dependencies {  
            implementation 'com.example.android:app-magic:12.3'  
        }  
    }  
}
```

Moving what was a top-level dependency in an Android project to a specific source set in a Multiplatform Mobile project might be difficult if the top-level dependency had a non-trivial configuration name. For example, to move a `debugImplementation` dependency from the top level of an Android project, you'll need to add an `implementation` dependency to the source set named `androidDebug`. To minimize the effort you have to put in to deal with migration problems like this, you can add a `dependencies` block inside the `android` block:

【Kotlin】

1.5.30 的新特性

```
android {  
    //...  
    dependencies {  
        implementation("com.example.android:app-magic:12.3")  
    }  
}
```

【Groovy】

```
android {  
    //...  
    dependencies {  
        implementation 'com.example.android:app-magic:12.3'  
    }  
}
```

Dependencies declared here will be treated exactly the same as dependencies from the top-level block, but declaring them this way will also separate Android dependencies visually in your build script and make it less confusing.

Putting dependencies into a standalone `dependencies` block at the end of the script, in a way that is idiomatic to Android projects, is also supported. However, we strongly recommend **against** doing this because configuring a build script with Android dependencies in the top-level block and other target dependencies in each source set is likely to cause confusion.

What's next?

Check out other resources on adding dependencies in multiplatform projects and learn more about:

- [Adding dependencies in the official Android documentation](#)
- [Adding dependencies on multiplatform libraries or other multiplatform projects](#)
- [Adding iOS dependencies](#)

添加 iOS 依赖项

Apple SDK dependencies (such as Foundation or Core Bluetooth) are available as a set of prebuilt libraries in Kotlin Multiplatform Mobile projects. They do not require any additional configuration.

You can also reuse other libraries and frameworks from the iOS ecosystem in your iOS source sets. Kotlin supports interoperability with Objective-C dependencies and Swift dependencies if their APIs are exported to Objective-C with the `@objc` attribute. Pure Swift dependencies are not yet supported.

Integration with the CocoaPods dependency manager is also supported with the same limitation – you cannot use pure Swift pods.

We recommend [using CocoaPods](#) to handle iOS dependencies in Kotlin Multiplatform Mobile projects. [Manage dependencies manually](#) only if you want to tune the interop process specifically or if you have some other strong reason to do so.

When using third-party iOS libraries in multiplatform projects with [hierarchical structure support](#), for example with the `ios()` [target shortcut](#), you won't be able to use IDE features, such as code completion and highlighting, for the shared iOS source set.

This is a [known issue](#), and we are working on resolving it. In the meantime, you can use [this workaround](#).

This issue doesn't apply to [platform libraries](#) supported out of the box.



With CocoaPods

1. Perform [initial CocoaPods integration setup](#).
2. Add a dependency on a Pod library from the CocoaPods repository that you want to use by including the `pod()` function call in `build.gradle.kts` (`build.gradle`) of your project.

【Kotlin】

1.5.30 的新特性

```
kotlin {  
    cocoapods {  
        //..  
        pod("AFNetworking") {  
            version = "~> 4.0.1"  
        }  
    }  
}
```

【Groovy】

```
kotlin {  
    cocoapods {  
        //..  
        pod('AFNetworking') {  
            version = '~> 4.0.1'  
        }  
    }  
}
```

You can add the following dependencies on a Pod library:

- * [From the CocoaPods repository](native-cocoapods-libraries.md#from-the-cocoapods-repository)
- * [On a locally stored library](native-cocoapods-libraries.md#on-a-locally-stored-library)
- * [From a custom Git repository](native-cocoapods-libraries.md#from-a-custom-git-repository)
- * [From an archive](native-cocoapods-libraries.md#from-a-zip-tar-or-jar-archive)
- * [From a custom Podspec repository](native-cocoapods-libraries.md#from-a-custom-podspec-repository)
- * [With custom cinterop options](native-cocoapods-libraries.md#with-custom-cinterop-options)
- * [On a static Pod library](native-cocoapods-libraries.md#on-a-static-pod-library)

1. Re-import the project.

To use the dependency in your Kotlin code, import the package `cocoapods.<library-name>`. For the example above, it's:

```
import cocoapods.AFNetworking.*
```

Without CocoaPods

1.5.30 的新特性

If you don't want to use CocoaPods, you can use the cinterop tool to create Kotlin bindings for Objective-C or Swift declarations. This will allow you to call them from the Kotlin code.

The steps differ a bit for [libraries](#) and [frameworks](#), but the idea remains the same.

1. Download your dependency.
2. Build it to get its binaries.
3. Create a special `.def` file that describes this dependency to cinterop.
4. Adjust your build script to generate bindings during the build.

Add a library without CocoaPods

1. Download the library source code and place it somewhere where you can reference it from your project.
2. Build a library (library authors usually provide a guide on how to do this) and get a path to the binaries.
3. In your project, create a `.def` file, for example `DateTools.def`.
4. Add a first string to this file: `language = Objective-C`. If you want to use a pure C dependency, omit the language property.
5. Provide values for two mandatory properties:
 - o `headers` describes which headers will be processed by cinterop.
 - o `package` sets the name of the package these declarations should be put into.

For example:

```
headers = DateTools.h
package = DateTools
```

6. Add information about interoperability with this library to the build script:

- o Pass the path to the `.def` file. This path can be omitted if your `.def` file has the same name as cinterop and is placed in the `src/nativeInterop/cinterop/` directory.
- o Tell cinterop where to look for header files using the `includeDirs` option.
- o Configure linking to library binaries.

【Kotlin】

1.5.30 的新特性

```
kotlin {
    iosX64() {
        compilations.getByName("main") {
            val DateTools by cinterop.creating {
                // Path to .def file
                defFile("src/nativeInterop/cinterop/DateTools.def")

                // Directories for header search (an analogue of the -I<path> command-line option)
                includeDirs("include>this/directory", "path/to/another/directory")
            }
            val anotherInterop by cinterop.creating { /* ... */ }
        }

        binaries.all {
            // Linker options required to link to the library.
            linkerOpts("-L/path/to/library/binaries", "-lbinaryname")
        }
    }
}
```

【Groovy】

```
kotlin {
    iosX64 {
        compilations.main {
            cinterop {
                DateTools {
                    // Path to .def file
                    defFile("src/nativeInterop/cinterop/DateTools.def")

                    // Directories for header search (an analogue of the -I<path> command-line option)
                    includeDirs("include>this/directory", "path/to/another/directory")
                }
                anotherInterop { /* ... */ }
            }
        }

        binaries.all {
            // Linker options required to link to the library.
            linkerOpts "-L/path/to/library/binaries", "-lbinaryname"
        }
    }
}
```

1.5.30 的新特性

1. Build the project.

Now you can use this dependency in your Kotlin code. To do that, import the package you've set up in the `package` property in the `.def` file. For the example above, this will be:

```
import DateTools.*
```

Add a framework without CocoaPods

1. Download the framework source code and place it somewhere that you can reference it from your project.
2. Build the framework (framework authors usually provide a guide on how to do this) and get a path to the binaries.
3. In your project, create a `.def` file, for example `MyFramework.def`.
4. Add the first string to this file: `language = Objective-C`. If you want to use a pure C dependency, omit the language property.
5. Provide values for these two mandatory properties:
 - `modules` – the name of the framework that should be processed by the cinterop.
 - `package` – the name of the package these declarations should be put into.For example:

```
modules = MyFramework
package = MyFramework
```

6. Add information about interoperability with the framework to the build script:

- Pass the path to the `.def` file. This path can be omitted if your `.def` file has the same name as the cinterop and is placed in the `src/nativeInterop/cinterop/` directory.
- Pass the framework name to the compiler and linker using the `-framework` option. Pass the path to the framework sources and binaries to the compiler and linker using the `-F` option.

【Kotlin】

1.5.30 的新特性

```
kotlin {
    iosX64() {
        compilations.getByName("main") {
            val DateTools by cinterop.creating {
                // Path to .def file
                defFile("src/nativeInterop/cinterop/DateTools.def")

                compilerOpts("-framework", "MyFramework", "-F/path/to/framework")
            }
            val anotherInterop by cinterop.creating { /* ... */ }
        }

        binaries.all {
            // Tell the linker where the framework is located.
            linkerOpts("-framework", "MyFramework", "-F/path/to/framework/")
        }
    }
}
```

【Groovy】

```
kotlin {
    iosX64 {
        compilations.main {
            cinterop {
                DateTools {

                    // Path to .def file
                    defFile("src/nativeInterop/cinterop/MyFramework.def")

                    compilerOpts("-framework", "MyFramework", "-F/path/to/framework")
                }
                anotherInterop { /* ... */ }
            }
        }

        binaries.all {
            // Tell the linker where the framework is located.
            linkerOpts("-framework", "MyFramework", "-F/path/to/framework/")
        }
    }
}
```

1. Build the project.

1.5.30 的新特性

Now you can use this dependency in your Kotlin code. To do this, import the package you've set up in the package property in the `.def` file. For the example above, this will be:

```
import MyFramework.*
```

Learn more about [Objective-C and Swift interop](#) and [configuring cinterop from Gradle](#).

Workaround to enable IDE support for the shared iOS source set

Due to a [known issue](#), you won't be able to use IDE features, such as code completion and highlighting, for the shared iOS source set in a multiplatform project with [hierarchical structure support](#) if your project depends on:

- Multiplatform libraries that don't support the hierarchical structure.
- Third-party iOS libraries, with the exception of [platform libraries](#) supported out of the box.

This issue applies only to the shared iOS source set. The IDE will correctly support the rest of the code.

All projects created with the Kotlin Multiplatform Mobile Project Wizard support the hierarchical structure, which means this issue affects them.



To enable IDE support in these cases, you can work around the issue by adding the following code to `build.gradle.(kts)` in the `shared` directory of your project:

【Kotlin】

```
val iosTarget: (String, KotlinNativeTarget.() -> Unit) -> KotlinNativeTarget =  
    if (System.getenv("SDK_NAME")?.startsWith("iphoneos") == true)  
        ::iosArm64  
    else  
        ::iosX64  
  
iosTarget("ios")
```

【Groovy】

1.5.30 的新特性

```
def iosTarget
if (System.getenv("SDK_NAME")?.startsWith("iphoneos")) {
    iosTarget = kotlin.&iosArm64
} else {
    iosTarget = kotlin.&iosX64
}
```

In this code sample, the configuration of iOS targets depends on the environment variable `SDK_NAME`, which is managed by Xcode. For each build, you'll have only one iOS target, named `ios`, that uses the `iosMain` source set. There will be no hierarchy of the `iosMain`, `iosArm64`, and `iosX64` source sets.

This is a temporary workaround. If you are a library author, we recommend that you [migrate to the hierarchical structure](#) as soon as possible.

With this workaround, Kotlin Multiplatform tooling analyzes your code against only the one native target that is active during the current build. This might lead to various errors during the complete build with all targets, and errors are more likely if your project contains other native targets in addition to the iOS ones.



What's next?

Check out other resources on adding dependencies in multiplatform projects and learn more about:

- [Adding dependencies on multiplatform libraries or other multiplatform projects](#)
- [Adding Android dependencies](#)

运行 Kotlin 多平台测试

By default, Kotlin supports running tests for JVM, JS, Android, Linux, Windows, macOS as well as iOS, watchOS, and tvOS simulators. To run tests for other Kotlin/Native targets, you need to configure them manually in an appropriate environment, emulator, or test framework.

Required dependencies

The `kotlin.test` API is available for multiplatform tests. When you [create a multiplatform project](#), the Project Wizard automatically adds test dependencies to common and platform-specific source sets.

If you didn't use the Project Wizard to create your project, you can [add the dependencies manually](#).

Run tests for one or more targets

To run tests for all targets, run the `check` task.

To run tests for a particular target suitable for testing, run a test task
`<targetName>Test` .

Test shared code

For testing shared code, you can use [actual declarations](#) in your tests.

For example, to test the shared code in `commonMain` :

1.5.30 的新特性

```
expect object Platform {
    val name: String
}

fun hello(): String = "Hello from ${Platform.name}"

class Proxy {
    fun proxyHello() = hello()
}
```

You can use the following test in `commonTest` :

```
import kotlin.test.Test
import kotlin.test.assertTrue

class SampleTests {
    @Test
    fun testProxy() {
        assertTrue(Proxy().proxyHello().isNotEmpty())
    }
}
```

And the following test in `iosTest` :

```
import kotlin.test.Test
import kotlin.test.assertTrue

class SampleTestsIOS {
    @Test
    fun testHello() {
        assertTrue("iOS" in hello())
    }
}
```

You can also learn how to create and run multiplatform tests in the [Create and publish a multiplatform library – tutorial](#).

构件编译项

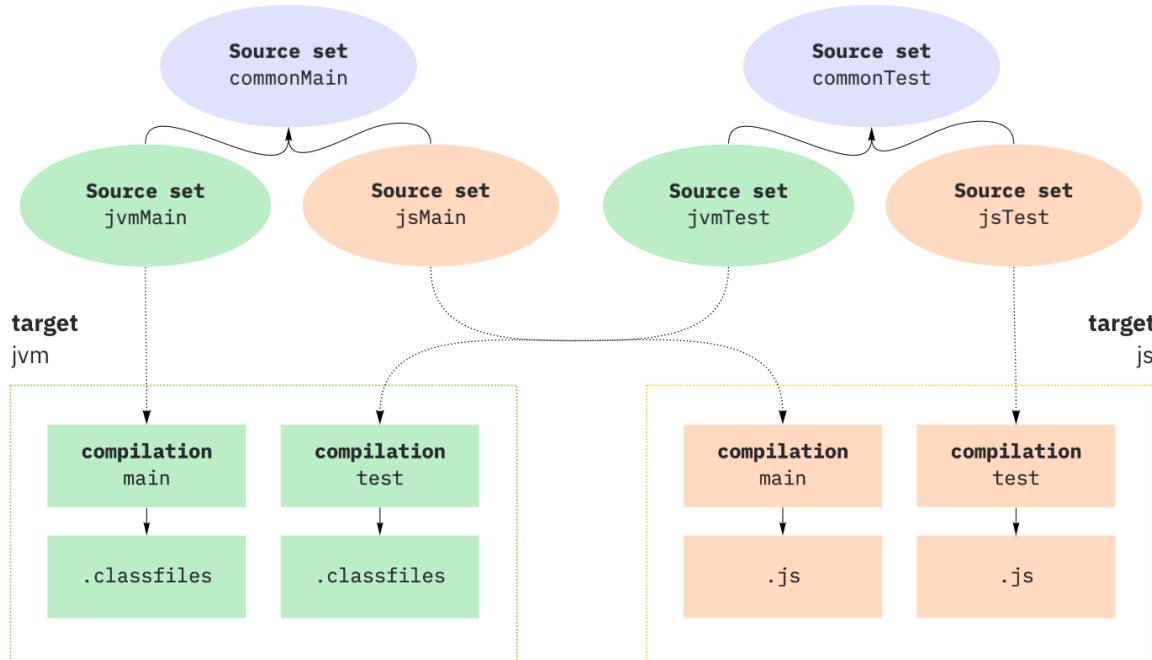
- 配置编译项
- 构建最终原生二进制文件

配置编译项

Kotlin multiplatform projects use compilations for producing artifacts. Each target can have one or more compilations, for example, for production and test purposes.

For each target, default compilations include:

- `main` and `test` compilations for JVM, JS, and Native targets.
- A [compilation per Android build variant](#), for Android targets.



If you need to compile something other than production code and unit tests, for example, integration or performance tests, you can [create a custom compilation](#).

You can configure how artifacts are produced in:

- [所有编译项](#) in your project at once.
- [一个目标的编译项](#) since one target can have multiple compilations.
- [具体编译项](#).

See the [list of compilation parameters](#) and [compiler options](#) available for all or specific targets.

配置所有编译项

1.5.30 的新特性

```
kotlin {  
    targets.all {  
        compilations.all {  
            kotlinOptions {  
                allWarningsAsErrors = true  
            }  
        }  
    }  
}
```

为一个目标配置编译项

【Kotlin】

```
kotlin {  
    targets.jvm.compilations.all {  
        kotlinOptions {  
            sourceMap = true  
            metaInfo = true  
        }  
    }  
}
```

【Groovy】

```
kotlin {  
    jvm().compilations.all {  
        kotlinOptions {  
            sourceMap = true  
            metaInfo = true  
        }  
    }  
}
```

配置一个编译项

【Kotlin】

1.5.30 的新特性

```
kotlin {  
    jvm {  
        val main by compilations.getting {  
            kotlinOptions {  
                jvmTarget = "1.8"  
            }  
        }  
    }  
}
```

【Groovy】

```
kotlin {  
    jvm().compilations.main {  
        kotlinOptions {  
            jvmTarget = "1.8"  
        }  
    }  
}
```

创建自定义编译项

If you need to compile something other than production code and unit tests, for example, integration or performance tests, create a custom compilation.

For example, to create a custom compilation for integration tests of the `jvm()` target, add a new item to the `compilations` collection.

For custom compilations, you need to set up all dependencies manually. The default source set of a custom compilation does not depend on the `commonMain` and the `commonTest` source sets.



【Kotlin】

1.5.30 的新特性

```
kotlin {  
    jvm() {  
        compilations {  
            val main by getting  
  
            val integrationTest by compilations.creating {  
                defaultSourceSet {  
                    dependencies {  
                        // Compile against the main compilation's compile classpath  
                        implementation(main.compileDependencyFiles + main.output.cl  
                        implementation(kotlin("test-junit"))  
                        /* ... */  
                    }  
                }  
            }  
  
            // Create a test task to run the tests produced by this compilation  
            tasks.register<Test>("integrationTest") {  
                // Run the tests with the classpath containing the compile depe  
                // runtime dependencies, and the outputs of this compilation:  
                classpath = compileDependencyFiles + runtimeDependencyFiles + o  
  
                // Run only the tests from this compilation's outputs:  
                testClassesDirs = output.classesDirs  
            }  
        }  
    }  
}
```

【Groovy】

1.5.30 的新特性

```
kotlin {  
    jvm() {  
        compilations.create('integrationTest') {  
            defaultSourceSet {  
                dependencies {  
                    def main = compilations.main  
                    // Compile against the main compilation's compile classpath and  
                    implementation(main.compileDependencyFiles + main.output.classesDir)  
                    implementation kotlin('test-junit')  
                    /* ... */  
                }  
            }  
        }  
  
        // Create a test task to run the tests produced by this compilation:  
        tasks.register('jvmIntegrationTest', Test) {  
            // Run the tests with the classpath containing the compile dependent  
            // runtime dependencies, and the outputs of this compilation:  
            classpath = compileDependencyFiles + runtimeDependencyFiles + output  
  
            // Run only the tests from this compilation's outputs:  
            testClassesDirs = output.classesDirs  
        }  
    }  
}
```

You also need to create a custom compilation in other cases, for example, if you want to combine compilations for different JVM versions in your final artifact, or you have already set up source sets in Gradle and want to migrate to a multiplatform project.

Use Java sources in JVM compilations

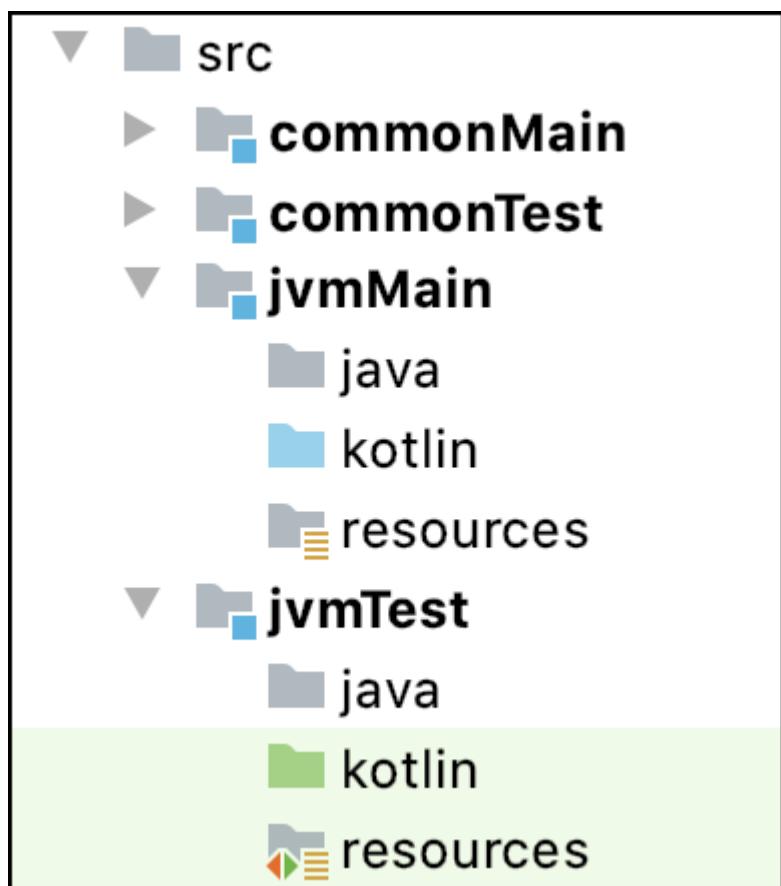
When [creating a project with the Project Wizard](#), Java sources are included in the compilations of the JVM target.

In the build script, the following section applies the Gradle `java` plugin and configures the target to cooperate with it:

```
kotlin {  
    jvm {  
        withJava()  
    }  
}
```

1.5.30 的新特性

The Java source files are placed in the child directories of the Kotlin source roots. For example, the paths are:



The common source sets cannot include Java sources.

Due to current limitations, the Kotlin plugin replaces some tasks configured by the Java plugin:

- The target's JAR task instead of `jar` (for example, `jvmJar`).
- The target's test task instead of `test` (for example, `jvmTest`).
- The resources are processed by the equivalent tasks of the compilations instead of `*ProcessResources` tasks.

The publication of this target is handled by the Kotlin plugin and doesn't require steps that are specific for the Java plugin.

配置与原生语言的互操作

Kotlin provides [interoperability with native languages](#) and DSL to configure this for a specific compilation.

1.5.30 的新特性

Native language	Supported platforms	Comments
C	All platforms, except for WebAssembly	
Objective-C	Apple platforms (macOS, iOS, watchOS, tvOS)	
Swift via Objective-C	Apple platforms (macOS, iOS, watchOS, tvOS)	Kotlin can use only Swift declarations marked with the <code>@objc</code> attribute.

A compilation can interact with several native libraries. Configure interoperability in the `cinterop` block of the compilation with [available parameters](#).

【Kotlin】

```
kotlin {  
    linuxX64 { // Replace with a target you need.  
        compilations.getByName("main") {  
            val myInterop by cinterop.creating {  
                // Def-file describing the native API.  
                // The default path is src/nativeInterop/cinterop/<interop-name>.de  
                defFile(project.file("def-file.def"))  
  
                // Package to place the Kotlin API generated.  
                packageName("org.sample")  
  
                // Options to be passed to compiler by cinterop tool.  
                compilerOpts("-Ipath/to/headers")  
  
                // Directories to look for headers.  
  
                includeDirs.apply {  
                    // Directories for header search (an equivalent of the -I<path>  
                    allHeaders("path1", "path2")  
  
                    // Additional directories to search headers listed in the 'head  
                    // -headerFilterAdditionalSearchPrefix command line option equi  
                    headerFilterOnly("path1", "path2")  
                }  
                // A shortcut for includeDirs.allHeaders.  
                includeDirs("include/directory", "another/directory")  
            }  
  
            val anotherInterop by cinterop.creating { /* ... */ }  
        }  
    }  
}
```

1.5.30 的新特性

【Groovy】

```
kotlin {  
    linuxX64 { // Replace with a target you need.  
        compilations.main {  
            cinterops {  
                myInterop {  
                    // Def-file describing the native API.  
                    // The default path is src/nativeInterop/cinterop/<interop-name>  
                    defFile project.file("def-file.def")  
  
                    // Package to place the Kotlin API generated.  
                    packageName 'org.sample'  
  
                    // Options to be passed to compiler by cinterop tool.  
                    compilerOpts '-Ipath/to/headers'  
  
                    // Directories for header search (an equivalent of the -I<path>  
                    includeDirs.allHeaders("path1", "path2")  
  
                    // Additional directories to search headers listed in the 'head  
                    // -headerFilterAdditionalSearchPrefix command line option equi  
                    includeDirs.headerFilterOnly("path1", "path2")  
  
                    // A shortcut for includeDirs.allHeaders.  
                    includeDirs("include/directory", "another/directory")  
                }  
  
                anotherInterop { /* ... */ }  
            }  
        }  
    }  
}
```

Android 编译项

The compilations created for an Android target by default are tied to [Android build variants](#): for each build variant, a Kotlin compilation is created under the same name.

Then, for each [Android source set](#) compiled for each of the variants, a Kotlin source set is created under that source set name prepended by the target name, like the Kotlin source set `androidDebug` for an Android source set `debug` and the Kotlin target named `android`. These Kotlin source sets are added to the variants' compilations accordingly.

1.5.30 的新特性

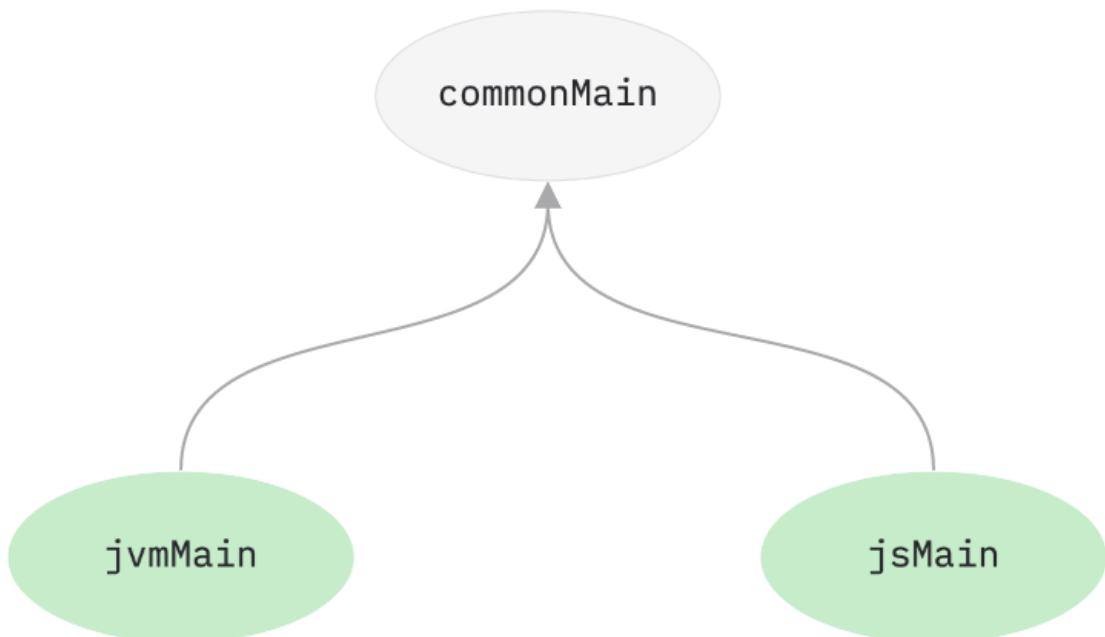
The default source set `commonMain` is added to each production (application or library) variant's compilation. The `commonTest` source set is similarly added to the compilations of unit test and instrumented test variants.

Annotation processing with `kapt` is also supported, but due to current limitations it requires that the Android target is created before the `kapt` dependencies are configured, which needs to be done in a top-level `dependencies` block rather than within Kotlin source set dependencies.

```
kotlin {  
    android { /* ... */ }  
}  
  
dependencies {  
    kapt("com.my.annotation:processor:1.0.0")  
}
```

源代码集层次结构的编译项

Kotlin can build a [source set hierarchy](#) with the `dependsOn` relation.



If the source set `jvmMain` depends on a source set `commonMain` then:

- Whenever `jvmMain` is compiled for a certain target, `commonMain` takes part in that compilation as well and is also compiled into the same target binary form, such as

1.5.30 的新特性

JVM class files.

- Sources of `jvmMain` 'see' the declarations of `commonMain`, including internal declarations, and also see the [dependencies](#) of `commonMain`, even those specified as `implementation dependencies`.
- `jvmMain` can contain platform-specific implementations for the [expected declarations](#) of `commonMain`.
- The resources of `commonMain` are always processed and copied along with the resources of `jvmMain`.
- The [language settings](#) of `jvmMain` and `commonMain` should be consistent.

Language settings are checked for consistency in the following ways:

- `jvmMain` should set a `languageVersion` that is greater than or equal to that of `commonMain`.
- `jvmMain` should enable all unstable language features that `commonMain` enables (there's no such requirement for bugfix features).
- `jvmMain` should use all experimental annotations that `commonMain` uses.
- `apiVersion`, bugfix language features, and `progressiveMode` can be set arbitrarily.

构建最终原生二进制文件

By default, a Kotlin/Native target is compiled down to a `*.klib` library artifact, which can be consumed by Kotlin/Native itself as a dependency but cannot be executed or used as a native library.

To declare final native binaries such as executables or shared libraries, use the `binaries` property of a native target. This property represents a collection of native binaries built for this target in addition to the default `*.klib` artifact and provides a set of methods for declaring and configuring them.

The `kotlin-multiplatform` plugin doesn't create any production binaries by default. The only binary available by default is a debug test executable that lets you run unit tests from the `test` compilation.



声明二进制文件

Use the following factory methods to declare elements of the `binaries` collection.

Factory method	Binary kind	Available for
<code>executable</code>	Product executable	All native targets
<code>test</code>	Test executable	All native targets
<code>sharedLib</code>	Shared native library	All native targets, except for WebAssembly
<code>staticLib</code>	Static native library	All native targets, except for WebAssembly
<code>framework</code>	Objective-C framework	macOS, iOS, watchOS, and tvOS targets only

The simplest version doesn't require any additional parameters and creates one binary for each build type. Currently, two build types are available:

- `DEBUG` – produces a non-optimized binary with debug information
- `RELEASE` – produces an optimized binary without debug information

The following snippet creates two executable binaries, debug and release:

1.5.30 的新特性

```
kotlin {  
    linuxX64 { // Define your target instead.  
        binaries {  
            executable {  
                // Binary configuration.  
            }  
        }  
    }  
}
```

You can drop the lambda if there is no need for [additional configuration](#):

```
binaries {  
    executable()  
}
```

You can specify for which build types to create binaries. In the following example, only the `debug` executable is created:

【Kotlin】

```
binaries {  
    executable(listOf(DEBUG)) {  
        // Binary configuration.  
    }  
}
```

【Groovy】

```
binaries {  
    executable([DEBUG]) {  
        // Binary configuration.  
    }  
}
```

You can also declare binaries with custom names:

【Kotlin】

1.5.30 的新特性

```
binaries {
    executable("foo", listOf(DEBUG)) {
        // Binary configuration.
    }

    // It's possible to drop the list of build types
    // (in this case, all the available build types will be used).
    executable("bar") {
        // Binary configuration.
    }
}
```

【Groovy】

```
binaries {
    executable('foo', [DEBUG]) {
        // Binary configuration.
    }

    // It's possible to drop the list of build types
    // (in this case, all the available build types will be used).
    executable('bar') {
        // Binary configuration.
    }
}
```

The first argument sets a name prefix, which is the default name for the binary file. For example, for Windows the code produces the files `foo.exe` and `bar.exe`. You can also use the name prefix to [access the binary in the build script](#).

访问二进制文件

You can access binaries to [configure them](#) or get their properties (for example, the path to an output file).

You can get a binary by its unique name. This name is based on the name prefix (if it is specified), build type, and binary kind following the pattern: `<optional-name-prefix><build-type><binary-kind>`, for example, `releaseFramework` or `testDebugExecutable`.

1.5.30 的新特性

Static and shared libraries have the suffixes static and shared respectively, for example, `fooDebugStatic` or `barReleaseShared`.



【Kotlin】

```
// Fails if there is no such binary.  
binaries["fooDebugExecutable"]  
binaries.getByName("fooDebugExecutable")  
  
// Returns null if there is no such binary.  
binaries.findByName("fooDebugExecutable")
```

【Groovy】

```
// Fails if there is no such binary.  
binaries['fooDebugExecutable']  
binaries.fooDebugExecutable  
binaries.getByName('fooDebugExecutable')  
  
// Returns null if there is no such binary.  
binaries.findByName('fooDebugExecutable')
```

Alternatively, you can access a binary by its name prefix and build type using typed getters.

【Kotlin】

```
// Fails if there is no such binary.  
binaries.getExecutable("foo", DEBUG)  
binaries.getExecutable(DEBUG) // Skip the first argument if the name prefix  
binaries.getExecutable("bar", "DEBUG") // You also can use a string for build type.  
  
// Similar getters are available for other binary kinds:  
// getFramework, getStaticLib and getSharedLib.  
  
// Returns null if there is no such binary.  
binaries.findExecutable("foo", DEBUG)  
  
// Similar getters are available for other binary kinds:  
// findFramework, findStaticLib and findSharedLib.
```

【Groovy】

1.5.30 的新特性

```
// Fails if there is no such binary.  
binaries.getExecutable('foo', DEBUG)  
binaries.getExecutable(DEBUG) // Skip the first argument if the name prefix  
binaries.getExecutable('bar', 'DEBUG') // You also can use a string for build type.  
  
// Similar getters are available for other binary kinds:  
// getFramework, getStaticLib and getSharedLib.  
  
// Returns null if there is no such binary.  
binaries.findExecutable('foo', DEBUG)  
  
// Similar getters are available for other binary kinds:  
// findFramework, findStaticLib and findSharedLib.
```

将依赖项导出到二进制文件

When building an Objective-C framework or a native library (shared or static), you may need to pack not just the classes of the current project, but also the classes of its dependencies. Specify which dependencies to export to a binary using the `export` method.

【Kotlin】

```
kotlin {  
    sourceSets {  
        macosMain.dependencies {  
            // Will be exported.  
            api(project(":dependency"))  
            api("org.example:exported-library:1.0")  
            // Will not be exported.  
            api("org.example:not-exported-library:1.0")  
        }  
    }  
    macosX64("macos").binaries {  
        framework {  
            export(project(":dependency"))  
            export("org.example:exported-library:1.0")  
        }  
        sharedLib {  
            // It's possible to export different sets of dependencies to different  
            // frameworks.  
            export(project(':dependency'))  
        }  
    }  
}
```

1.5.30 的新特性

【Groovy】

```
kotlin {  
    sourceSets {  
        macosMain.dependencies {  
            // Will be exported.  
            api project':dependency'  
            api 'org.example:exported-library:1.0'  
            // Will not be exported.  
            api 'org.example:not-exported-library:1.0'  
        }  
    }  
    macosX64("macos").binaries {  
        framework {  
            export project':dependency'  
            export 'org.example:exported-library:1.0'  
        }  
        sharedLib {  
            // It's possible to export different sets of dependencies to different  
            export project':dependency'  
        }  
    }  
}
```

For example, you implement several modules in Kotlin and want to access them from Swift. Usage of several Kotlin/Native frameworks in a Swift application is limited, but you can create an umbrella framework and export all these modules to it.

You can export only `api dependencies` of the corresponding source set.



When you export a dependency, it includes all of its API to the framework API. The compiler adds the code from this dependency to the framework, even if you use a small fraction of it. This disables dead code elimination for the exported dependency (and for its dependencies, to some extent).

By default, export works non-transitively. This means that if you export the library `foo` depending on the library `bar`, only methods of `foo` are added to the output framework.

You can change this behavior using the `transitiveExport` option. If set to `true`, the declarations of the library `bar` are exported as well.

1.5.30 的新特性

It is not recommended to use `transitiveExport` : it adds all transitive dependencies of the exported dependencies to the framework. This could increase both compilation time and binary size.

In most cases, you don't need to add all these dependencies to the framework API. Use `export` explicitly for the dependencies you need to directly access from your Swift or Objective-C code.



【Kotlin】

```
binaries {  
    framework {  
        export(project(":dependency"))  
        // Export transitively.  
        transitiveExport = true  
    }  
}
```

【Groovy】

```
binaries {  
    framework {  
        export project(':dependency')  
        // Export transitively.  
        transitiveExport = true  
    }  
}
```

构建 universal frameworks

By default, an Objective-C framework produced by Kotlin/Native supports only one platform. However, you can merge such frameworks into a single universal (fat) binary using the `lipo tool`. This operation especially makes sense for 32-bit and 64-bit iOS frameworks. In this case, you can use the resulting universal framework on both 32-bit and 64-bit devices.

The fat framework must have the same base name as the initial frameworks.



【Kotlin】

1.5.30 的新特性

```
import org.jetbrains.kotlin.gradle.tasks.FatFrameworkTask

kotlin {
    // Create and configure the targets.
    val ios32 = iosArm32("ios32")
    val ios64 = iosArm64("ios64")
    configure(listOf(ios32, ios64)) {
        binaries.framework {
            basePath = "my_framework"
        }
    }
    // Create a task to build a fat framework.
    tasks.register<FatFrameworkTask>("debugFatFramework") {
        // The fat framework must have the same base name as the initial frameworks
        basePath = "my_framework"
        // The default destination directory is "<build directory>/fat-framework".
        destinationDir = buildDir.resolve("fat-framework/debug")
        // Specify the frameworks to be merged.
        from(
            ios32.binaries.getFramework("DEBUG"),
            ios64.binaries.getFramework("DEBUG")
        )
    }
}
```

【Groovy】

1.5.30 的新特性

```
import org.jetbrains.kotlin.gradle.tasks.FatFrameworkTask

kotlin {
    // Create and configure the targets.
    targets {
        iosArm32("ios32")
        iosArm64("ios64")
        configure([ios32, ios64]) {
            binaries.framework {
                basePath = "my_framework"
            }
        }
    }
    // Create a task building a fat framework.
    tasks.register("debugFatFramework", FatFrameworkTask) {
        // The fat framework must have the same base name as the initial frameworks
        basePath = "my_framework"
        // The default destination directory is "<build directory>/fat-framework".
        destinationDir = file("$buildDir/fat-framework/debug")
        // Specify the frameworks to be merged.
        from(
            targets.ios32.binaries.getFramework("DEBUG"),
            targets.ios64.binaries.getFramework("DEBUG")
        )
    }
}
```

Build XCFrameworks

All Kotlin Multiplatform projects can use XCFrameworks as an output to gather logic for all the target platforms and architectures in a single bundle. Unlike [universal \(fat\) frameworks](#), you don't need to remove all unnecessary architectures before publishing the application to the App Store.

【Kotlin】

1.5.30 的新特性

```
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】

1.5.30 的新特性

```
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, Kotlin Gradle plugin will register three Gradle tasks:

- assembleXCFramework
- assembleDebugXCFramework (additionally debug artifact that contains dSYMs)
- assembleReleaseXCFramework

多平台 Gradle DSL 参考

Multiplatform projects are in [Alpha](#). Language features and tooling may change in future Kotlin versions.



Kotlin 多平台 Gradle 插件是用于创建 [Kotlin 多平台项目](#)的工具。这里我们提供了它的参考；在为 Kotlin 多平台项目编写 Gradle 构建脚本时，用它作提醒。[Learn the concepts of Kotlin Multiplatform projects, how to create and configure them.](#)

id 与版本

Kotlin 多平台 Gradle 插件的全限定名是 `org.jetbrains.kotlin.multiplatform`。如果你使用 Kotlin Gradle DSL，那么你可以通过 `kotlin("multiplatform")` 来应用插件。插件版本与 Kotlin 发行版本相匹配。最新的版本是：1.6.10。

【Kotlin】

```
plugins {  
    kotlin("multiplatform") version "1.6.10"  
}
```

【Groovy】

```
plugins {  
    id 'org.jetbrains.kotlin.multiplatform' version '1.6.10'  
}
```

顶层块

`kotlin` 是在 Gradle 构建脚本中用于配置多平台项目的顶层块。`kotlin` 块内，你可以使用以下块：

1.5.30 的新特性

块	介绍
\<目标名称>	声明项目的特定目标，所有可用的目标名称已陈列在 目标 部分中。
targets	项目的所有目标。
presets	所有预定义的目标。使用这个同时 配置多个预定义目标 。
sourceSets	配置预定义和声明自定义项目的源代码集。

目标

目标 是构建的一部分，负责构建编译、测试、以及针对某个已支持平台打包软件。

Kotlin provides target presets for each platform. See how to [use a target preset](#).

Each target can have one or more [compilations](#). In addition to default compilations for test and production purposes, you can [create custom compilations](#).

多平台项目的目标在 `kotlin` 块中的相应代码块中描述，例如：`jvm`、`android` 以及 `iosArm64`。以下是可用目标的完整列表：

1.5.30 的新特性

Target platform	Target preset	Comments
Kotlin/JVM	jvm	
Kotlin/JS	js	<p>Select the execution environment:</p> <ul style="list-style-type: none"> • <code>browser {}</code> for applications running in the browser. • <code>nodejs{}</code> for applications running on Node.js. <p>Learn more in Setting up a Kotlin/JS project.</p>
Android applications and libraries	android	<p>Manually apply an Android Gradle plugin – <code>com.android.application</code> or <code>com.android.library</code> .</p> <p>You can only create one Android target per Gradle subproject.</p>
Android NDK	<ul style="list-style-type: none"> • <code>androidNativeArm32</code> — Android NDK on ARM (ARM32) platforms • <code>androidNativeArm64</code> — Android NDK on ARM64 platforms • <code>androidNativeX86</code> — Android NDK on x86 platforms • <code>androidNativeX64</code> — Android NDK on x86_64 platforms 	<p>The 64-bit target requires a Linux or macOS host.</p> <p>You can build the 32-bit target on any supported host.</p>

1.5.30 的新特性

iOS	<ul style="list-style-type: none"><code>iosArm32</code> — Apple iOS on ARM (ARM32) platforms (Apple iPhone 5 and earlier)<code>iosArm64</code> — Apple iOS on ARM64 platforms (Apple iPhone 5s and newer)<code>iosX64</code> — Apple iOS simulator on <code>x86_64</code> platforms<code>iosSimulatorArm64</code> — Apple iOS simulator on Apple Silicon platforms	Requires a macOS host.
watchOS	<ul style="list-style-type: none"><code>watchosArm32</code> — Apple watchOS on ARM (ARM32) platforms (Apple Watch Series 3 and earlier)<code>watchosArm64</code> — Apple watchOS on <code>ARM64_32</code> platforms (Apple Watch Series 4 and newer)<code>watchosX86</code> — Apple watchOS 32-bit simulator (<code>watchOS 6.3</code> and earlier) on <code>x86_64</code> platforms<code>watchosX64</code> — Apple watchOS 64-bit simulator (<code>watchOS 7.0</code> and newer) on <code>x86_64</code> platforms<code>watchosSimulatorArm64</code> — Apple watchOS simulator on Apple Silicon platforms	Requires a macOS host.
tvOS	<ul style="list-style-type: none"><code>tvosArm64</code> — Apple tvOS on ARM64 platforms (Apple TV 4th generation and newer)<code>tvosX64</code> — Apple tvOS simulator on <code>x86_64</code> platforms<code>tvosSimulatorArm64</code> — Apple tvOS simulator on Apple Silicon platforms	Requires a macOS host.

1.5.30 的新特性

macOS	<ul style="list-style-type: none"><code>macosX64</code> — Apple macOS on x86_64 platforms<code>macosArm64</code> — Apple macOS on Apple Silicon platforms	Requires a macOS host.
Linux	<ul style="list-style-type: none"><code>linuxArm64</code> — Linux on ARM64 platforms, for example, Raspberry Pi<code>linuxArm32Hfp</code> — Linux on hard-float ARM (ARM32) platforms<code>linuxMips32</code> — Linux on MIPS platforms<code>linuxMipsel32</code> — Linux on little-endian MIPS (mipsel) platforms<code>linuxX64</code> — Linux on x86_64 platforms	<p>Linux MIPS targets (<code>linuxMips32</code> and <code>linuxMipsel32</code>) require a Linux host.</p> <p>You can build other Linux targets on any supported host.</p>
Windows	<ul style="list-style-type: none"><code>mingwX64</code> — 64-bit Microsoft Windows<code>mingwX86</code> — 32-bit Microsoft Windows	
WebAssembly	<code>wasm32</code>	

A target that is not supported by the current host is ignored during building and therefore not published.



```
kotlin {  
    jvm()  
    iosX64()  
    macosX64()  
    js().browser()  
}
```

目标的配置项可以包含这两个部分：

- 可用于所有目标的[公共目标配置](#)。
- 目标特定的配置项。

Each target can have one or more [compilations](#).

公共目标配置

In any target block, you can use the following declarations:

Name	Description
attributes	Attributes used for disambiguating targets for a single platform.
preset	The preset that the target has been created from, if any.
platformType	Designates the Kotlin platform of this target. Available values: <code>jvm</code> , <code>androidJvm</code> , <code>js</code> , <code>native</code> , <code>common</code> .
artifactsTaskName	The name of the task that builds the resulting artifacts of this target.
components	The components used to setup Gradle publications.

JVM 目标

In addition to [common target configuration](#), `jvm` targets have a specific function:

Name	Description
<code>withJava()</code>	Includes Java sources into the JVM target's compilations.

Use this function for projects that contain both Java and Kotlin source files. Note that the default source directories for Java sources don't follow the Java plugin's defaults. Instead, they are derived from the Kotlin source sets. For example, if the JVM target has the default name `jvm`, the paths are `src/jvmMain/java` (for production Java sources) and `src/jvmTest/java` for test Java sources. Learn more about [Java sources in JVM compilations](#).

```
kotlin {
    jvm {
        withJava()
    }
}
```

JavaScript 目标

The `js` block describes the configuration of JavaScript targets. It can contain one of two blocks depending on the target execution environment:

1.5.30 的新特性

Name	Description
browser	Configuration of the browser target.
nodejs	Configuration of the Node.js target.

Learn more about [configuring Kotlin/JS projects](#).

Browser

`browser` can contain the following configuration blocks:

Name	Description
testRuns	Configuration of test execution.
runTask	Configuration of project running.
webpackTask	Configuration of project bundling with Webpack .
dceTask	Configuration of Dead Code Elimination .
distribution	Path to output files.

```
kotlin {  
    js().browser {  
        webpackTask { /* ... */ }  
        testRuns { /* ... */ }  
        dceTask {  
            keep("myKotlinJsApplication.org.example.keepFromDce")  
        }  
        distribution {  
            directory = File("$projectDir/customdir/")  
        }  
    }  
}
```

Node.js

`nodejs` can contain configurations of test and run tasks:

Name	Description
testRuns	Configuration of test execution.
runTask	Configuration of project running.

1.5.30 的新特性

```
kotlin {  
    js().nodejs {  
        runTask { /* ... */ }  
        testRuns { /* ... */ }  
    }  
}
```

Native 目标

For native targets, the following specific blocks are available:

Name	Description
binaries	Configuration of binaries to produce.
cinterops	Configuration of interop with C libraries .

Binaries

There are the following kinds of binaries:

Name	Description
executable	Product executable.
test	Test executable.
sharedLib	Shared library.
staticLib	Static library.
framework	Objective-C framework.

```
kotlin {  
    linuxX64 { // Use your target instead.  
        binaries {  
            executable {  
                // Binary configuration.  
            }  
        }  
    }  
}
```

For binaries configuration, the following parameters are available:

1.5.30 的新特性

Name	Description
compilation	The compilation from which the binary is built. By default, <code>test</code> binaries are based on the <code>test</code> compilation while other binaries - on the <code>main</code> compilation.
linkerOpts	Options passed to a system linker during binary building.
baseName	Custom base name for the output file. The final file name will be formed by adding system-dependent prefix and postfix to this base name.
entryPoint	The entry point function for executable binaries. By default, it's <code>main()</code> in the root package.
outputFile	Access to the output file.
linkTask	Access to the link task.
runTask	Access to the run task for executable binaries. For targets other than <code>linuxX64</code> , <code>macosX64</code> , or <code>mingwX64</code> the value is <code>null</code> .
isStatic	For Objective-C frameworks. Includes a static library instead of a dynamic one.

【Kotlin】

1.5.30 的新特性

```
binaries {
    executable("my_executable", listOf(RELEASE)) {
        // Build a binary on the basis of the test compilation.
        compilation = compilations["test"]

        // Custom command line options for the linker.
        linkerOpts = mutableListOf("-L/lib/search/path", "-L/another/search/path",

        // Base name for the output file.
        baseName = "foo"

        // Custom entry point function.
        entryPoint = "org.example.main"

        // Accessing the output file.
        println("Executable path: ${outputFile.absolutePath}")

        // Accessing the link task.
        linkTask.dependsOn(additionalPreprocessingTask)

        // Accessing the run task.
        // Note that the runTask is null for non-host platforms.
        runTask?.dependsOn(prepareForRun)
    }

    framework("my_framework" listOf(RELEASE)) {
        // Include a static library instead of a dynamic one into the framework.
        isStatic = true
    }
}
```

【Groovy】

1.5.30 的新特性

```
binaries {
    executable('my_executable', [RELEASE]) {
        // Build a binary on the basis of the test compilation.
        compilation = compilations.test

        // Custom command line options for the linker.
        linkerOpts = ['-L/lib/search/path', '-L/another/search/path', '-lmylib']

        // Base name for the output file.
        baseName = 'foo'

        // Custom entry point function.
        entryPoint = 'org.example.main'

        // Accessing the output file.
        println("Executable path: ${outputFile.absolutePath}")

        // Accessing the link task.
        linkTask.dependsOn(additionalPreprocessingTask)

        // Accessing the run task.
        // Note that the runTask is null for non-host platforms.
        runTask?.dependsOn(prepareForRun)
    }

    framework('my_framework' [RELEASE]) {
        // Include a static library instead of a dynamic one into the framework.
        isStatic = true
    }
}
```

Learn more about [building native binaries](#).

CInterops

`cinterop` 是一个描述与本地库进行互操作的集合。要提供一个互操作，添加一个到 `cinterop` 的条目并定义其参数：

Name	Description
<code>defFile</code>	<code>def</code> 文件描述本地 API。
<code>packageName</code>	生成的 Kotlin API 的包前缀。
<code>compilerOpts</code>	编译器工具传递给编译器的选项。
<code>includeDirs</code>	查找头文件的目录。

Learn more how to [configure interop with native languages](#).

1.5.30 的新特性

【Kotlin】

```
kotlin {
    linuxX64 { // Replace with a target you need.
        compilations.getByName("main") {
            val myInterop by cinterop.creating {
                // Def-file describing the native API.
                // The default path is src/nativeInterop/cinterop/<interop-name>.def
                defFile(project.file("def-file.def"))

                // Package to place the Kotlin API generated.
                packageName("org.sample")

                // Options to be passed to compiler by cinterop tool.
                compilerOpts("-Ipath/to/headers")

                // Directories for header search (an analogue of the -I<path> compiler option).
                includeDirs.allHeaders("path1", "path2")

                // A shortcut for includeDirs.allHeaders.
                includeDirs("include/directory", "another/directory")
            }
        }

        val anotherInterop by cinterop.creating { /* ... */ }
    }
}
```

【Groovy】

1.5.30 的新特性

```
kotlin {  
    linuxX64 { // Replace with a target you need.  
        compilations.main {  
            cinterop {  
                myInterop {  
                    // Def-file describing the native API.  
                    // The default path is src/nativeInterop/cinterop/<interop-name>  
                    defFile project.file("def-file.def")  
  
                    // Package to place the Kotlin API generated.  
                    packageName 'org.sample'  
  
                    // Options to be passed to compiler by cinterop tool.  
                    compilerOpts '-Ipath/to/headers'  
  
                    // Directories for header search (an analogue of the -I<path> command-line option).  
                    includeDirs.allHeaders("path1", "path2")  
  
                    // A shortcut for includeDirs.allHeaders.  
                    includeDirs("include/directory", "another/directory")  
                }  
  
                anotherInterop { /* ... */ }  
            }  
        }  
    }  
}
```

Android 目标

The Kotlin Multiplatform plugin contains two specific functions for android targets. Two functions help you configure [build variants](#):

Name	Description
<code>publishLibraryVariants()</code>	Specifies build variants to publish. Learn more about publishing Android libraries .
<code>publishAllLibraryVariants()</code>	Publishes all build variants.

```
kotlin {  
    android {  
        publishLibraryVariants("release", "debug")  
    }  
}
```

Learn more about [compilation for Android](#).

1.5.30 的新特性

The `android` configuration inside `kotlin` doesn't replace the build configuration of any Android project. Learn more about writing build scripts for Android projects in [Android developer documentation](#).



源代码集

The `sourceSets` block describes source sets of the project. A source set contains Kotlin source files that participate in compilations together, along with their resources, dependencies, and language settings.

A multiplatform project contains [predefined](#) source sets for its targets; developers can also create [custom](#) source sets for their needs.

预定义源代码集

Predefined source sets are set up automatically upon creation of a multiplatform project. Available predefined source sets are the following:

Name	Description
<code>commonMain</code>	Code and resources shared between all platforms. Available in all multiplatform projects. Used in all main compilations of a project.
<code>commonTest</code>	Test code and resources shared between all platforms. Available in all multiplatform projects. Used in all test compilations of a project.
<code>\</code>	Target-specific sources for a compilation. <code>\</code> is the name of a predefined target and <code>\</code> is the name of a compilation for this target. Examples: <code>jsTest</code> , <code>jvmMain</code> .

With Kotlin Gradle DSL, the sections of predefined source sets should be marked by `getting` .

【Kotlin】

```
kotlin {  
    sourceSets {  
        val commonMain by getting { /* ... */ }  
    }  
}
```

1.5.30 的新特性

【Groovy】

```
kotlin {  
    sourceSets {  
        commonMain { /* ... */ }  
    }  
}
```

Learn more about [source sets](#).

自定义源代码集

Custom source sets are created by the project developers manually. To create a custom source set, add a section with its name inside the `sourceSets` section. If using Kotlin Gradle DSL, mark custom source sets [by creating](#).

【Kotlin】

```
kotlin {  
    sourceSets {  
        val myMain by creating { /* ... */ } // create a new source set by the name  
    }  
}
```

【Groovy】

```
kotlin {  
    sourceSets {  
        myMain { /* ... */ } // create or configure a source set by the name 'myMain'  
    }  
}
```

Note that a newly created source set isn't connected to other ones. To use it in the project's compilations, [connect it with other source sets](#).

源代码集参数

Configurations of source sets are stored inside the corresponding blocks of `sourceSets`. A source set has the following parameters:

1.5.30 的新特性

Name	Description
kotlin.srcDir	Location of Kotlin source files inside the source set directory.
resources.srcDir	Location of resources inside the source set directory.
dependsOn	Connection with another source set .
dependencies	依赖项 of the source set.
languageSettings	语言设置 applied to the source set.

【Kotlin】

```
kotlin {  
    sourceSets {  
        val commonMain by getting {  
            kotlin.srcDir("src")  
            resources.srcDir("res")  
  
            dependencies {  
                /* ... */  
            }  
        }  
    }  
}
```

【Groovy】

```
kotlin {  
    sourceSets {  
        commonMain {  
            kotlin.srcDir('src')  
            resources.srcDir('res')  
  
            dependencies {  
                /* ... */  
            }  
        }  
    }  
}
```

编译项

1.5.30 的新特性

A target can have one or more compilations, for example, for production or testing. There are [predefined compilations](#) that are added automatically upon target creation. You can additionally create [custom compilations](#).

To refer to all or some particular compilations of a target, use the `compilations` object collection. From `compilations`, you can refer to a compilation by its name.

Learn more about [configuring compilations](#).

预定义编译项

Predefined compilations are created automatically for each target of a project except for Android targets. Available predefined compilations are the following:

Name	Description
main	Compilation for production sources.
test	Compilation for tests.

【Kotlin】

```
kotlin {  
    jvm {  
        val main by compilations.getting {  
            output // get the main compilation output  
        }  
  
        compilations["test"].runtimeDependencyFiles // get the test runtime classpath  
    }  
}
```

【Groovy】

```
kotlin {  
    jvm {  
        compilations.main.output // get the main compilation output  
        compilations.test.runtimeDependencyFiles // get the test runtime classpath  
    }  
}
```

自定义编译项

1.5.30 的新特性

In addition to predefined compilations, you can create your own custom compilations. To create a custom compilation, add a new item into the `compilations` collection. If using Kotlin Gradle DSL, mark custom compilations by creating .

Learn more about creating a [custom compilation](#).

【Kotlin】

```
kotlin {  
    jvm() {  
        compilations {  
            val integrationTest by compilations.creating {  
                defaultSourceSet {  
                    dependencies {  
                        /* ... */  
                    }  
                }  
  
                // Create a test task to run the tests produced by this compilation  
                tasks.register<Test>("integrationTest") {  
                    /* ... */  
                }  
            }  
        }  
    }  
}
```

【Groovy】

```
kotlin {  
    jvm() {  
        compilations.create('integrationTest') {  
            defaultSourceSet {  
                dependencies {  
                    /* ... */  
                }  
            }  
  
            // Create a test task to run the tests produced by this compilation:  
            tasks.register('jvmIntegrationTest', Test) {  
                /* ... */  
            }  
        }  
    }  
}
```

编译项参数

A compilation has the following parameters:

Name	Description
<code>defaultSourceSet</code>	The compilation's default source set.
<code>kotlinSourceSets</code>	Source sets participating in the compilation.
<code>allKotlinSourceSets</code>	Source sets participating in the compilation and their connections via <code>dependsOn()</code> .
<code>kotlinOptions</code>	Compiler options applied to the compilation. For the list of available options, see Compiler options .
<code>compileKotlinTask</code>	Gradle task for compiling Kotlin sources.
<code>compileKotlinTaskName</code>	Name of <code>compileKotlinTask</code> .
<code>compileAllTaskName</code>	Name of the Gradle task for compiling all sources of a compilation.
<code>output</code>	The compilation output.
<code>compileDependencyFiles</code>	Compile-time dependency files (classpath) of the compilation.
<code>runtimeDependencyFiles</code>	Runtime dependency files (classpath) of the compilation.

【Kotlin】

1.5.30 的新特性

```
kotlin {  
    jvm {  
        val main by compilations.getting {  
            kotlinOptions {  
                // Setup the Kotlin compiler options for the 'main' compilation:  
                jvmTarget = "1.8"  
            }  
  
            compileKotlinTask // get the Kotlin task 'compileKotlinJvm'  
            output // get the main compilation output  
        }  
  
        compilations["test"].runtimeDependencyFiles // get the test runtime classpath  
    }  
  
    // Configure all compilations of all targets:  
    targets.all {  
        compilations.all {  
            kotlinOptions {  
                allWarningsAsErrors = true  
            }  
        }  
    }  
}
```

【Groovy】

```
kotlin {  
    jvm {  
        compilations.main.kotlinOptions {  
            // Setup the Kotlin compiler options for the 'main' compilation:  
            jvmTarget = "1.8"  
        }  
  
        compilations.main.compileKotlinTask // get the Kotlin task 'compileKotlinJv  
        compilations.main.output // get the main compilation output  
        compilations.test.runtimeDependencyFiles // get the test runtime classpath  
    }  
  
    // Configure all compilations of all targets:  
    targets.all {  
        compilations.all {  
            kotlinOptions {  
                allWarningsAsErrors = true  
            }  
        }  
    }  
}
```

依赖项

The `dependencies` block of the source set declaration contains the dependencies of this source set.

Learn more about [configuring dependencies](#).

There are four types of dependencies:

Name	Description
<code>api</code>	Dependencies used in the API of the current module.
<code>implementation</code>	Dependencies used in the module but not exposed outside it.
<code>compileOnly</code>	Dependencies used only for compilation of the current module.
<code>runtimeOnly</code>	Dependencies available at runtime but not visible during compilation of any module.

【Kotlin】

```
kotlin {
    sourceSets {
        val commonMain by getting {
            dependencies {
                api("com.example:foo-metadata:1.0")
            }
        }
        val jvm6Main by getting {
            dependencies {
                implementation("com.example:foo-jvm6:1.0")
            }
        }
    }
}
```

【Groovy】

1.5.30 的新特性

```
kotlin {  
    sourceSets {  
        commonMain {  
            dependencies {  
                api 'com.example:foo-metadata:1.0'  
            }  
        }  
        jvm6Main {  
            dependencies {  
                implementation 'com.example:foo-jvm6:1.0'  
            }  
        }  
    }  
}
```

Additionally, source sets can depend on each other and form a hierarchy. In this case, the `dependsOn()` relation is used.

Source set dependencies can also be declared in the top-level `dependencies` block of the build script. In this case, their declarations follow the pattern `<sourceSetName><DependencyKind>`, for example, `commonMainApi`.

【Kotlin】

```
dependencies {  
    "commonMainApi"("com.example:foo-common:1.0")  
    "jvm6MainApi"("com.example:foo-jvm6:1.0")  
}
```

【Groovy】

```
dependencies {  
    commonMainApi 'com.example:foo-common:1.0'  
    jvm6MainApi 'com.example:foo-jvm6:1.0'  
}
```

语言设置

The `languageSettings` block of a source set defines certain aspects of project analysis and build. The following language settings are available:

1.5.30 的新特性

Name	Description
languageVersion	Provides source compatibility with the specified version of Kotlin.
apiVersion	Allows using declarations only from the specified version of Kotlin bundled libraries.
enableLanguageFeature	Enables the specified language feature. The available values correspond to the language features that are currently experimental or have been introduced as such at some point.
optIn	Allows using the specified opt-in annotation .
progressiveMode	Enables the progressive mode .

【Kotlin】

```
kotlin {  
    sourceSets.all {  
        languageSettings.apply {  
            languageVersion = "1.4" // possible values: "1.0", "1.1", "1.2", "1.3",  
            apiVersion = "1.4" // possible values: "1.0", "1.1", "1.2", "1.3", "1.4"  
            enableLanguageFeature("InlineClasses") // language feature name  
            optIn("kotlin.ExperimentalUnsignedTypes") // annotation FQ-name  
            progressiveMode = true // false by default  
        }  
    }  
}
```

【Groovy】

```
kotlin {  
    sourceSets.all {  
        languageSettings {  
            languageVersion = '1.4' // possible values: '1.0', '1.1', '1.2', '1.3',  
            apiVersion = '1.4' // possible values: '1.0', '1.1', '1.2', '1.3', '1.4'  
            enableLanguageFeature('InlineClasses') // language feature name  
            optIn('kotlin.ExperimentalUnsignedTypes') // annotation FQ-name  
            progressiveMode = true // false by default  
        }  
    }  
}
```

样例

This is a curated list of Kotlin Multiplatform Mobile samples.

Do you have a great sample that you would like to add to the list?

Feel free to [create a pull request](#) and tell us about it! You can use this [PR example](#) for reference.

1.5.30 的新特性

Sample name	What's shared?	Popular libraries used	UI Framework	iOS integration
Kotlin Multiplatform Mobile Sample	Algorithms	-	XML, SwiftUI	Xcode build phases
KMM RSS Reader	Models, Networking, Data Storage, UI State	SQLDelight, Ktor, DateTime, multiplatform-settings, Napier, kotlinx.serialization	Jetpack Compose, SwiftUI	Xcode build phases
KaMPKit	Models, Networking, Data Storage, ViewModels	Koin, SQLDelight, Ktor, DateTime, multiplatform-settings, Kermit	Jetpack Compose, SwiftUI	CocoaPods
moko-template	Models, Networking, Data Storage, ViewModels	Moko Libraries, Ktor, multiplatform-settings	-	CocoaPods
PeopleInSpace	Models, Networking, Data Storage	Koin, SQLDelight, Ktor	Jetpack Compose, SwiftUI	CocoaPods Swift Packages
GitFox SDK	Models, Networking, Interactors	Ktor	XML, UIKit	Xcode build phases

1.5.30 的新特性

D-KMP-sample	Networking, Data Storage, ViewModels, Navigation	SQLDelight, Ktor, DateTime, multiplatform-settings	Jetpack Compose, SwiftUI	Xcode build phases
Food2Fork Recipe App	Models, Networking, Data Storage, Interactors	SQLDelight, Ktor, DateTime	Jetpack Compose, SwiftUI	CocoaPods
kmm-ktor-sample	Networking	Ktor, kotlinx.serialization, Napier	XML, SwiftUI	Xcode build phases
Currency Converter Calculator	Models, Networking, Data Storage, Algorithms, ViewModels	Ktor, SQLDelight, koin, moko-resources, kotlinx.datetime, multiplatform-settings	XML, SwiftUI	CocoaPods
todoapp	Models, Networking, Presentation, Navigation and UI	SQLDelight, Decompose, MVIKotlin, Reaktive	Jetpack Compose, SwiftUI	Xcode build phases
kmm-arch-demo	Models, Networking, ViewModels, UI State	Ktor, kotlinx.serialization	XML, SwiftUI	CocoaPods

1.5.30 的新特性

Codeforces WatchR	Models, Networking, Data Storage, UI State	SQLDelight, Ktor, kotlinx.serialization	XML, UIKit	CocoaPods
CatViewerDemo	Models, Networking, Data Storage, ViewModels	Ktor, multiplatform-settings, kotlinx.serialization	Jetpack Compose, SwiftUI	Xcode build phases
Praxis KMM	Models, Networking, Data Storage	kotlinx.serialization, Ktor, Koin, SQLDelight	Jetpack Compose, SwiftUI, ReactJS	CocoaPods, Swift Packages
Bookshelf	Models, Networking, Data Storage	Realm-Kotlin, Ktor, kotlinx.serialization	Jetpack Compose, SwiftUI	CocoaPods
kotlin-with-cocoapods-sample	-	-	-	CocoaPods
multitarget-xcode-with-kotlin-cocoapods-sample	-	-	-	CocoaPods
mpp-sample-lib	Algorithms	-	-	-

FAQ

What is Kotlin Multiplatform Mobile?

Kotlin Multiplatform Mobile (KMM) is an SDK for cross-platform mobile development. You can develop multiplatform mobile applications and share parts of your applications between Android and iOS, such as core layers, business logic, presentation logic, and more.

Kotlin Mobile uses the [multiplatform abilities of Kotlin](#) and the features designed for mobile development, such as CocoaPods integration and the [Android Studio Plugin](#).

You may want to watch this introductory [video](#), in which Kotlin Product Marketing Manager Ekaterina Petrova explains in detail what Kotlin Multiplatform Mobile is and how you can use it in your projects. With Ekaterina, you'll set up an environment and prepare for creating your first cross-platform mobile application with Kotlin Multiplatform Mobile.

What is the Kotlin Multiplatform Mobile plugin?

The [Kotlin Multiplatform Mobile plugin](#) for Android Studio helps you develop applications that work on both Android and iOS.

With the Kotlin Multiplatform Mobile plugin, you can:

- Run, test, and debug the iOS part of your application on iOS targets straight from Android Studio.
- Quickly create a new multiplatform project.
- Add a multiplatform module into an existing project.

The Kotlin Multiplatform Mobile plugin works only on macOS. This is because iOS simulators, per the Apple requirement, can run only on macOS but not on any other operating systems, such as Microsoft Windows or Linux.

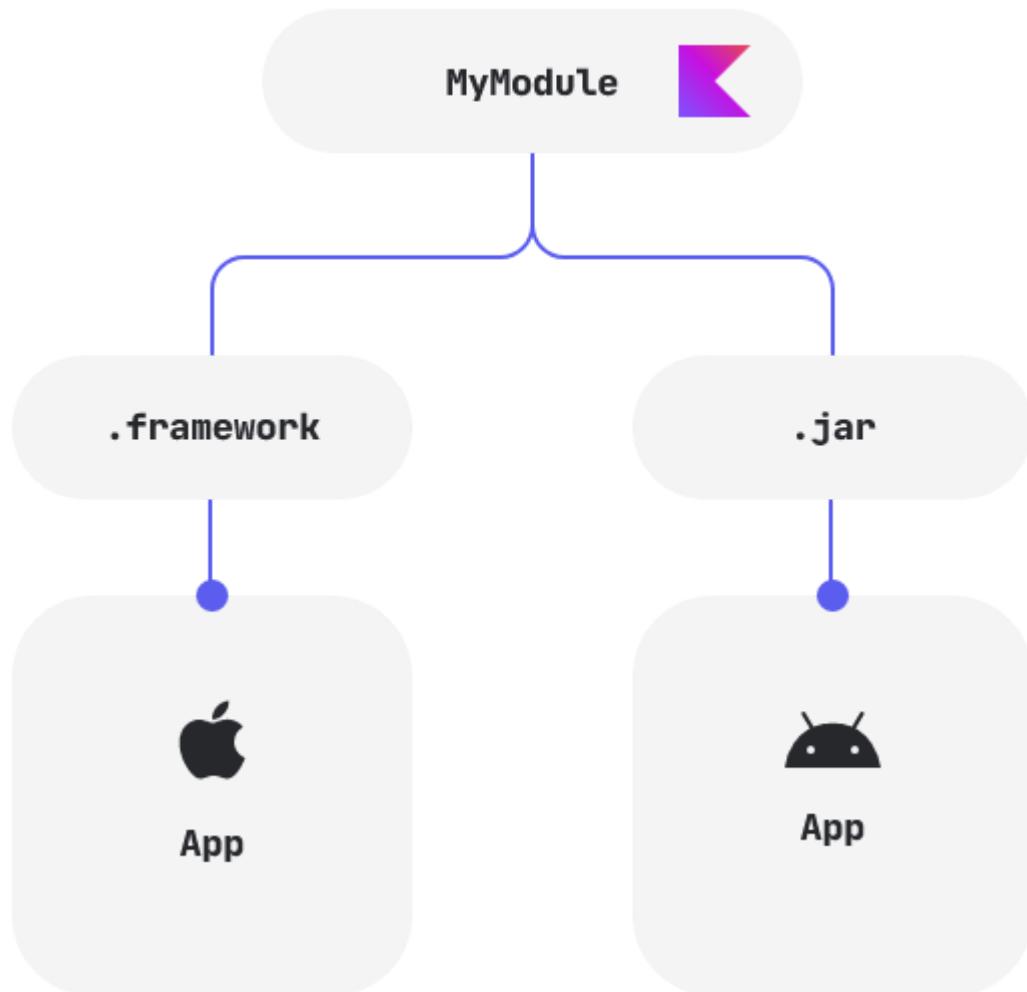
The good news is that you can work with cross-platform projects on Android even without the Kotlin Multiplatform Mobile plugin. If you are going to work with shared code or Android-specific code, you can work on any operating system supported by Android Studio.

What is Kotlin/Native and how does it relate to Kotlin Multiplatform Mobile?

[Kotlin/Native](#) is a technology for compiling Kotlin code to native binaries, which can run without a virtual machine. It consists of an [LLVM](#)-based backend for the Kotlin compiler and a native implementation of the Kotlin standard library.

Kotlin/Native is primarily designed to allow compilation for platforms where virtual machines are not desirable or possible, such as embedded devices and iOS. It is particularly suitable for situations when the developer needs to produce a self-contained program that does not require an additional runtime or virtual machine. And that is exactly the case with iOS development.

Shared code, written in Kotlin, is compiled to JVM bytecode for Android with Kotlin/JVM and to native binaries for iOS with Kotlin/Native. It makes the integration with Kotlin Multiplatform Mobile seamless on both platforms.



What are the plans for the technology evolution?

1.5.30 的新特性

Kotlin Multiplatform Mobile is one of the focus areas of the [Kotlin roadmap](#). To see which parts we're working on right now, check out the [roadmap details](#). Most of the recent changes affect the **Kotlin Multiplatform** and **Kotlin/Native** sections.

The following video presents our plans on the upcoming stage of the Kotlin Multiplatform Mobile development – its promotion to [Beta](#):

YouTube 视频: [Kotlin Multiplatform Mobile Beta Roadmap](#)

Can I run an iOS application on Microsoft Windows or Linux?

If you want to write iOS-specific code and run an iOS application on a simulated or real device, use a Mac with a macOS ([use the Kotlin Multiplatform Mobile plugin for it](#)). This is because iOS simulators can run only on macOS, per the Apple requirement, but cannot run on other operating systems, such as Microsoft Windows or Linux.

If you are going to work with shared code or Android-specific code, you can work on any operating system supported by Android Studio.

Where can I get complete examples to play with?

- [Curated samples](#)
- Several [hands-on tutorials](#)

In which IDE should I work on my cross-platform app?

You can work in [Android Studio](#). Android Studio allows the use of the [Kotlin Multiplatform Mobile plugin](#), which is a part of the Kotlin ecosystem. Enable the Kotlin Multiplatform Mobile plugin in Android Studio if you want to write iOS-specific code and launch an iOS application on a simulated or real device. The plugin can be used only on macOS.

Most of our adopters use Android Studio. However, if there is any reason for you not to use it, there is another option: you can use [IntelliJ IDEA](#). IntelliJ IDEA provides the ability to create a multiplatform mobile application from the Project Wizard, but you won't be able to launch an iOS application from the IDE.

How can I write concurrent code in Kotlin Multiplatform Mobile projects?

You can learn how to work with [concurrency](#) on the documentation portal.

Working with concurrent code in cross-platform mobile projects might not seem straightforward, as different memory management approaches are used in Kotlin/JVM and Kotlin/Native. The current approach for Kotlin/Native has some [limitations](#). The new Kotlin/Native memory management model is on the [roadmap](#) and the team is working on a solution for it.

How can I speed up my Kotlin Multiplatform module compilation for iOS?

See these [tips for improving Kotlin/Native compilation times](#).

向团队介绍跨平台移动端开发

These recommendations will help you introduce your team to Kotlin Multiplatform Mobile:

- Start with empathy
- Explain how Kotlin Multiplatform Mobile works
- Show the value using case studies
- Offer a proof by creating a sample project yourself
- Prepare for questions from your team
- Support your team during the adaptation

Start with empathy

Software development is a team game, with each critical decision needing the approval of all team members. Integrating any cross-platform technology will significantly affect the development process for your mobile application. So before you start integrating Kotlin Multiplatform Mobile in your project, you'll need to introduce your team to the technology and guide them gently to see it's worth adopting.

Understanding the people who work on your project is the first step to successful integration. Your boss is responsible for delivering features with the best quality in the shortest time possible. To them, any new technology is a risk. Your colleagues have a different perspective, as well. They have experience building apps with the “native” technology stack. They know how to write the UI and business logic, work with dependencies, test, and debug code in the IDE, and they are already familiar with the language. Switching to a different ecosystem is very uncomfortable, as it always means leaving your comfort zone.

Given all that, be ready to face lots of biases and answer a lot of questions when advocating for the move to Kotlin Multiplatform Mobile. As you do, never lose sight of what your team needs. Some of the advice below might be useful for preparing your pitch.

Explain how it works

1.5.30 的新特性

At this stage, you need to get rid of any preexisting bad feelings about cross-platform mobile applications and show that using Kotlin Multiplatform in your project is not only possible but also won't bring regular cross-platform problems. You should explain why there won't be any problems, such as:

- *Limitations of using all iOS and Android features* – Whenever a task cannot be solved in the shared code or whenever you want to use specific native features, you can use the expect/actual pattern to seamlessly write platform-specific code.
- *Performance issues* – Shared code written in Kotlin is compiled to different output formats for different targets: to Java bytecode for Android and to native binaries for iOS. Thus, there is no additional runtime overhead when it comes to executing this code on platforms, and the performance is comparable to native apps.
- *Legacy code problems* – No matter how large your project is, your existing code will not prevent you from integrating Kotlin Multiplatform. You can start writing cross-platform code at any moment and connect it to your iOS and Android Apps as a regular dependency, or you can use the code you've already written and simply modify it to be compatible with iOS.

Being able to explain *how* technology works is important, as nobody likes when a discussion seems to rely on magic. People might think the worst if anything is unclear to them, so be careful not to make the mistake of thinking something is too obvious to warrant explanation. Instead, try to explain all the basic concepts before moving on to the next stage. This document on [multiplatform programming](#) could help you systemize your knowledge to prepare for this experience.

Show the value

Understanding how the technology works is necessary, but not enough. Your team needs to see the gains of using it, and the way you present these gains should be related to your product. Kotlin Multiplatform Mobile allows you to use a single codebase for the business logic of iOS and Android apps. So if you develop a very thin client and the majority of the code is UI logic, then the main power of Kotlin Multiplatform Mobile will be unused in your project. However, if your application has complex business logic, for example if you have features like networking, data storage, payments, complex computations, or data synchronization, then this logic could easily be written and shared between iOS and Android so you can experience the real power of the technology.

1.5.30 的新特性

At this stage, you need to explain the main gains of using Kotlin Multiplatform in your product. One of the ways is to share stories of other companies who already benefit from the technology. The successful experience of these teams, especially ones with similar product objectives, could become a key factor in the final decision.

Citing case studies of different companies who already use Kotlin Multiplatform in production could significantly help you make a compelling argument:

- [Chalk.com](#) – The UI for each of the Chalk.com apps is native to the platform, but otherwise almost everything for their apps can be shared with Kotlin Multiplatform Mobile.
- [Cash App](#) – A lot of the app's business logic, including the ability to search through all transactions, is implemented with Kotlin Multiplatform Mobile.
- [Yandex.Disk](#) – They started out by experimenting with the integration of a small feature, and as the experiment was considered successful, they implemented their whole data synchronization logic in Kotlin Multiplatform Mobile.

Explore [the case studies page](#) for inspirational references.

Offer proof

The theory is good, but putting it into practice is ultimately most important. As one option to make your case more convincing, you can take the risky choice of devoting some of your personal free time to creating something with Kotlin Multiplatform and then bringing in the results for your team to discuss. Your prototype could be some sort of test project, which you would write from scratch and which would demonstrate features that are needed in your application. [Networking & data storage – hands-on tutorial](#) can guide you well on this process.

The more relevant examples could be produced by experimenting with your current project. You could take one existing feature implemented in Kotlin and make it cross-platform, or you could even create a new Multiplatform Module in your existing project, take one non-priority feature from the bottom of the backlog, and implement it in the shared module. [Make your Android application work on iOS – tutorial](#) provides a step-by-step guide based on a sample project.

The new [Kotlin Multiplatform Mobile plugin for Android Studio](#) will allow you to accomplish either of these tasks in the shortest amount of time by using the **Kotlin Multiplatform App** or **Kotlin Multiplatform Library** wizards.

Prepare for questions

No matter how detailed your pitch is, your team will have a lot of questions. Listen carefully, and try to answer them all patiently. You might expect the majority of the questions to come from the iOS part of the team, as they are the developers who aren't used to seeing Kotlin in their everyday developer routine. This list of some of the most common questions could help you here:

Q: I heard applications based on cross-platform technologies can be rejected from the AppStore. Is taking this risk worth it?

A: The Apple Store has strict guidelines for application publishing. One of the limitations is that apps may not download, install, or execute code which introduces or changes features or functionality of the app ([App Store Review Guideline 2.5.2](#)). This is relevant for some cross-platform technologies, but not for Kotlin Multiplatform Mobile. Shared Kotlin code compiles to native binaries with Kotlin/Native, bundles a regular iOS framework into your app, and doesn't provide the ability for dynamic code execution.

Q: Multiplatform projects are built with Gradle, and Gradle has an extremely steep learning curve. Do I need to spend a lot of time now trying to configure my project?

A: There's actually no need. There are various ways to organize the work process around building Kotlin mobile applications. First, only Android developers could be responsible for the builds, in which case the iOS team would only write code or even only consume the resulting artifact. You also can organize some workshops or practice pair programming while facing tasks that require working with Gradle, and this would increase your team's Gradle skills. You can explore different ways of and choose the one that's most appropriate for your team.

Also, in basic scenarios, you simply need to configure your project at the start, and then you just add dependencies to it. The new AS plugin makes configuring your project much easier, so it can now be done in a few clicks.

When only the Android part of the team works with shared code, the iOS developers don't even need to learn Kotlin. But when you are ready for your team to move to the next stage, where everyone contributes to the shared code, making the transition won't take much time. The similarities between the syntax and functionality of Swift and Kotlin greatly reduce the work required to learn how to read and write shared Kotlin code. [Try it yourself!](#)

1.5.30 的新特性

Q: I heard that Kotlin Multiplatform Mobile is experimental technology. Does that mean that we shouldn't use it for production?

A: Experimental status means we and the whole Kotlin community are just trying out an idea, but if it doesn't work, it may be dropped anytime. However, after the release of Kotlin 1.4, **Kotlin Multiplatform Mobile is in Alpha** status. This means the Kotlin team is fully committed to working to improve and evolve this technology and will not suddenly drop it. However, before going Beta, there could be some migration issues yet. But even experimental status doesn't prevent a feature from being used successfully in production, as long as you understand all the risks. Check [the Kotlin evolution page](#) for information about the stability statuses of Kotlin Multiplatform components.

Q: There are not enough multiplatform libraries to implement the business logic, it's much easier to find native alternatives.

A: Of course, we can't compare the number of multiplatform libraries with React Native, for example. But it took five years for React Native to expand their ecosystem to its current size. Kotlin Multiplatform Mobile is still young, but the ecosystem has tremendous potential as there are already a lot of modern libraries written in Kotlin that can be easily ported to multiplatform.

It's also a great time to be an iOS developer in the Kotlin Multiplatform open-source community because the iOS experience is in demand and there are plenty of opportunities to gain recognition from iOS-specific contributions.

And the more your team digs into the technology, the more interesting and complex their questions will be. Don't worry if you don't have the answers – Kotlin Multiplatform has a large and [supportive community in the Kotlin Slack](#), where a lot of developers who already use it can help you. We would be very thankful if you could [share with us](#) the most popular questions asked by your team. This information will help us understand what topics need to be covered in the documentation.

Be supportive

After you decide to use Kotlin Multiplatform, there will be an adaptation period as your team experiments with the technology. And your mission will not be over yet! By providing continuous support for your teammates, you will reduce the time it takes for your team to dive into the technology and achieve their first results.

Here are some tips on how you can support your team at this stage:

1.5.30 的新特性

- Collect the questions you were asked during the previous stage on the “*Kotlin Multiplatform: Frequently asked questions*” wiki page and share it with your team.
- Create a `#kotlin-multiplatform-support` Slack channel and become the most active user there.
- Organize an informal team building event with popcorn and pizza where you watch educational or inspirational videos about Kotlin Multiplatform. “[Shipping a Mobile Multiplatform Project on iOS & Android](#)” by Ben Asher & Alec Strong could be a good choice.

The reality is that you probably will not change people's hearts and minds in a day or even a week. But patience and attentiveness to the needs of your colleagues will undoubtedly bring results.

The Kotlin Multiplatform Mobile team looks forward to hearing [your story](#).

We'd like to thank the [Touchlab team](#) for helping us write this article.

Community resources

- [Kotlin Multiplatform vs Flutter. What Should You Choose to Build a Software Project?](#)

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——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](#)
- [JavaScript](#)
 - [以 Kotlin/JS for React 入门](#)
 - [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](#)
 - [将 Kotlin/JS 项目迁移到 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](#)

1.5.30 的新特性

- 类型安全的 HTML DSL
- 用 Dukat 生成外部声明
- Kotlin/JS 动手实践实验室
- 原生
 - Kotlin/Native 入门——在 IntelliJ IDEA 中
 - Kotlin/Native 入门——使用 Gradle
 - Kotlin/Native 入门——使用命令行编译器
 - 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
 - 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
 - CocoaPods 集成
 - CocoaPods 概述
 - 添加对 Pod 库的依赖
 - 使用 Kotlin Gradle 项目作为 CocoaPods 依赖项
 - Kotlin/Native 库
 - 平台库
 - Kotlin/Native 开发动态库——教程
 - 不可变性与并发
 - 处理并发
 - 并发概述
 - 并发可变性
 - 并发与协程
 - 调试 Kotlin/Native
 - 符号化 iOS 崩溃报告
 - 改进 Kotlin/Native 编译时间的技巧
 - Kotlin/Native FAQ
 - Kotlin/Native 中的并发
- 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——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
 - \[字符串\]\(#\)
 - \[集合\]\(#\)](#)

Kotlin/JVM 入门

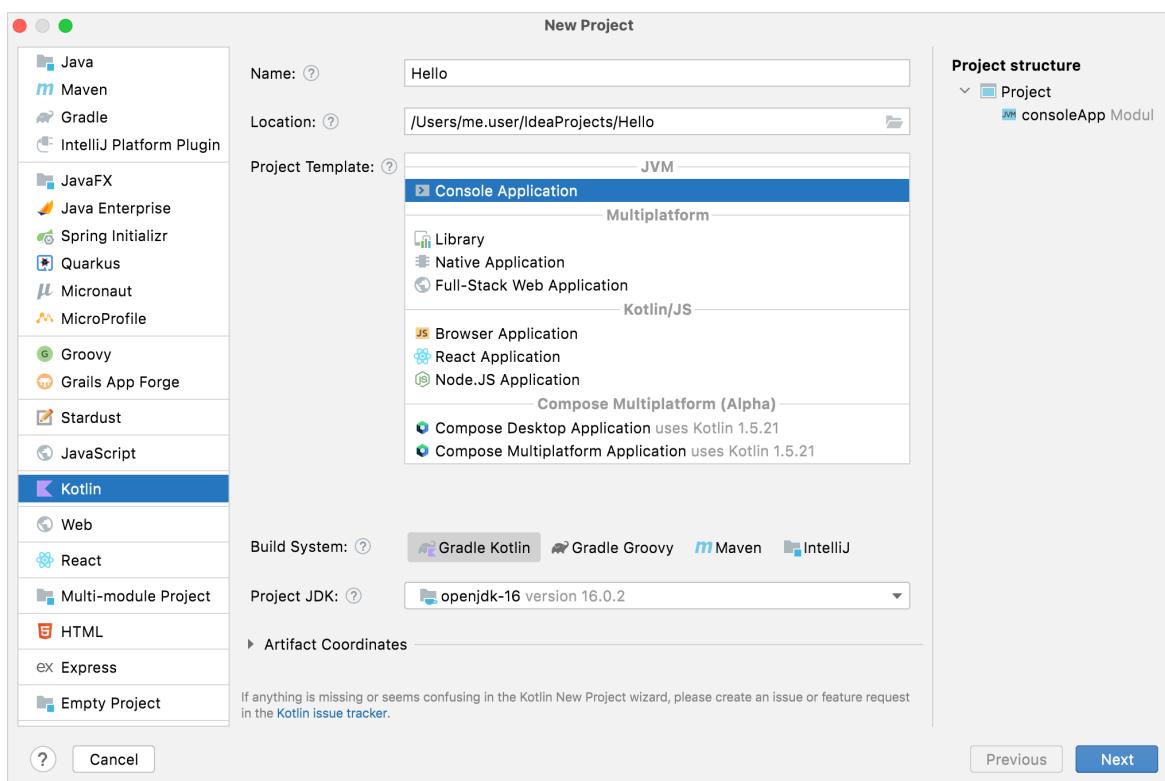
This tutorial demonstrates how to use IntelliJ IDEA for creating a console application.

To get started, first download and install the latest version of [IntelliJ IDEA](#).

Create an application

Once you've installed IntelliJ IDEA, it's time to create your first Kotlin application.

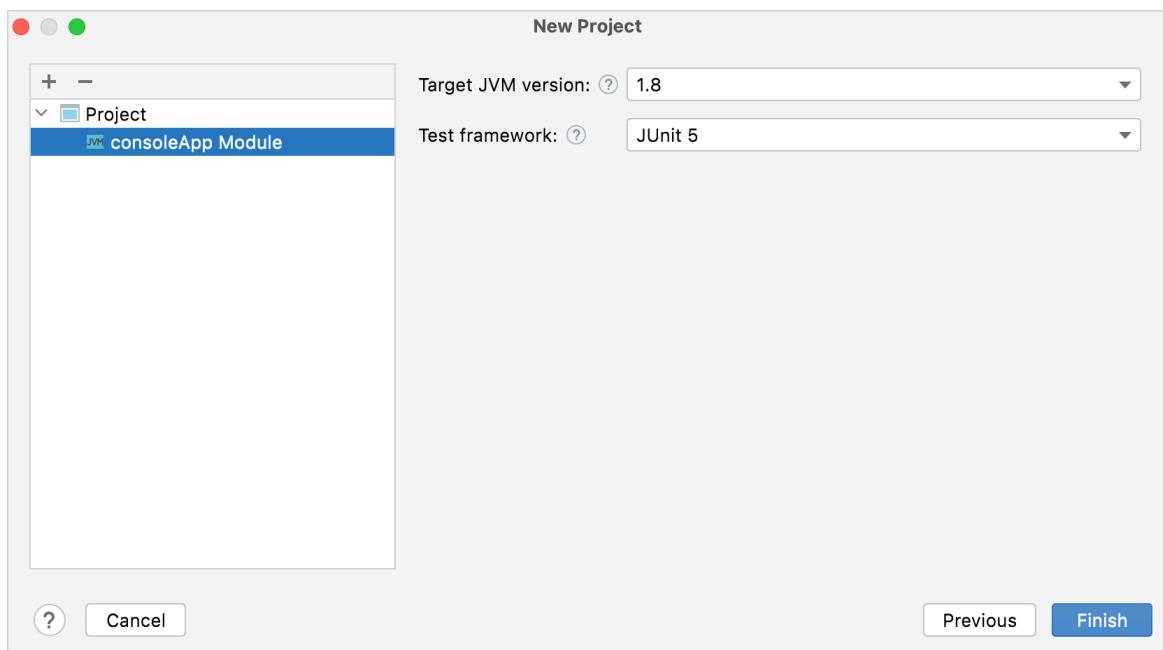
1. In IntelliJ IDEA, select **File | New | Project**.
2. In the panel on the left, select **Kotlin**.
3. Enter a project name, select **Console Application** as the project template, and click **Next**.



By default, your project will use the Gradle build system with Kotlin DSL.

4. Go through and accept the default configuration, then click **Finish**. Your project will open.

1.5.30 的新特性

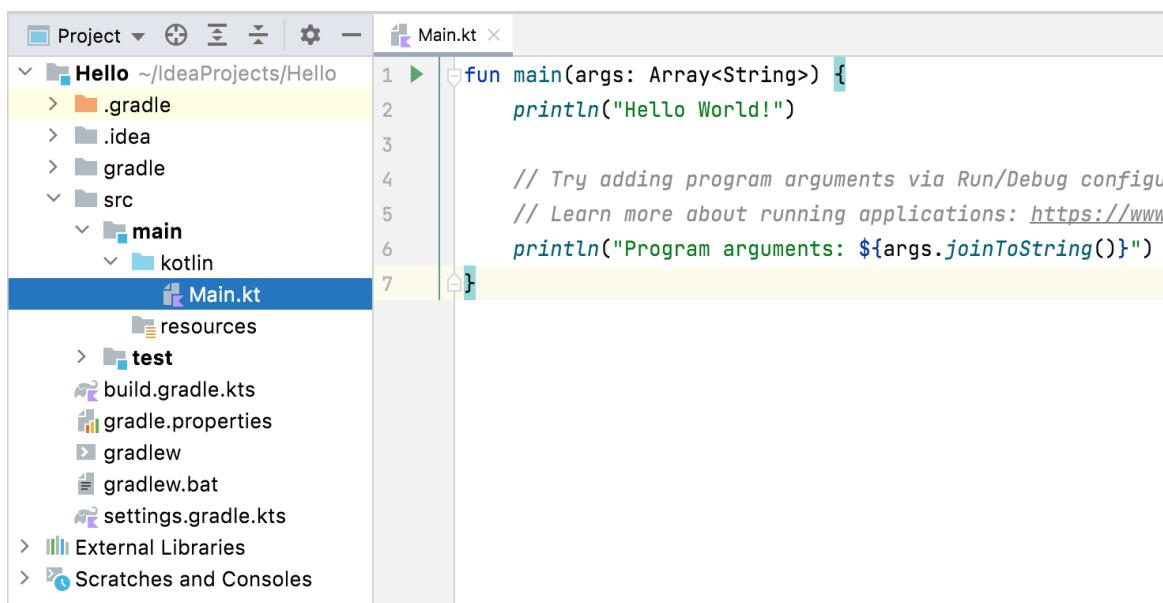


5. Open the `build.gradle.kts` file, the build script created by default based on your configuration. It includes the `kotlin("jvm")` plugin and dependencies required for your console application. Ensure that you use the latest version of the plugin:

```
plugins {
    kotlin("jvm") version "1.6.10"
    application
}
```

6. 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!`.



1.5.30 的新特性

7. Modify the code so that it requests your name and says `Hello` to you specifically, and not to the whole world:

- Introduce a local variable `name` with the keyword `val`. It will get its value from an input where you will enter your name – `readln()`.

The `readln()` function is available since [Kotlin 1.6.0](#).

Ensure that you have installed the latest version of the [Kotlin plugin](#).



- Use a string template by adding a dollar sign `$` before this variable name directly in the text output like this – `$name`.

```
fun main() {  
    println("What's your name?")  
    val name = readln()  
    println("Hello, $name!")  
}
```

A screenshot of a code editor showing the file `Main.kt`. The code is as follows:

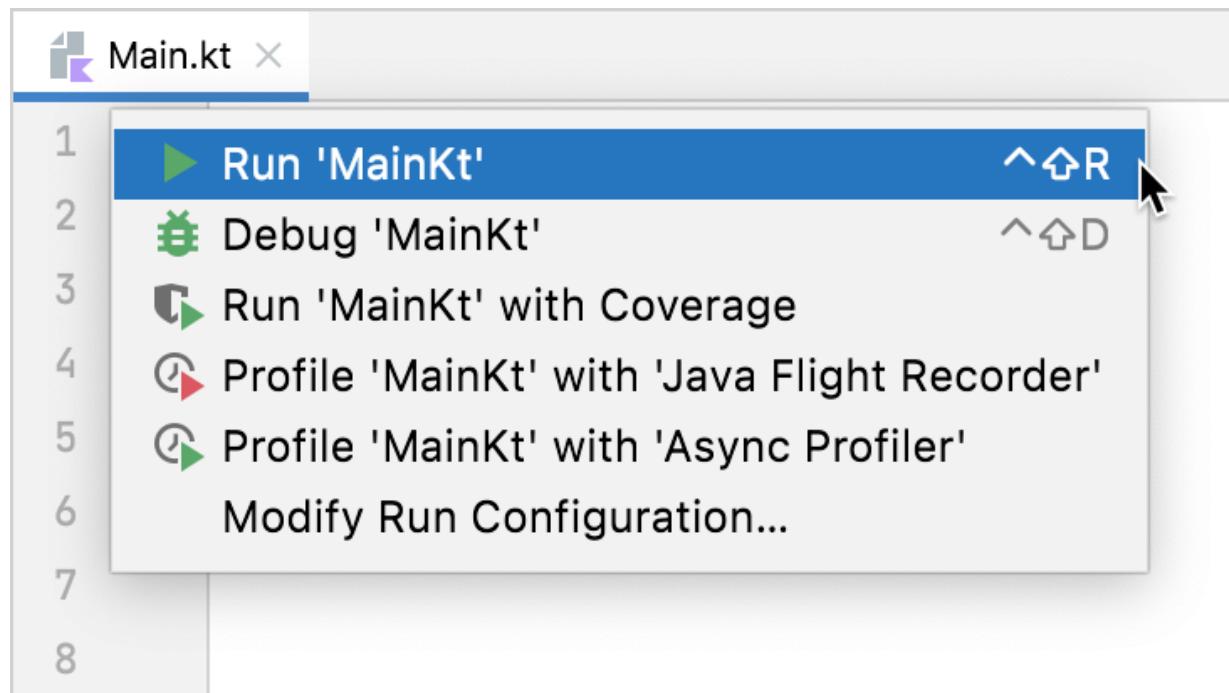
```
1 ► fun main() {  
2     println("What's your name?")  
3     val name = readln()  
4     println("Hello, $name!")  
5 }  
6
```

The gutter on the left has several icons: a green play button at line 1, a blue folder icon at line 2, a blue open folder icon at line 3, a blue open folder icon at line 4, and a grey folder icon at line 5.

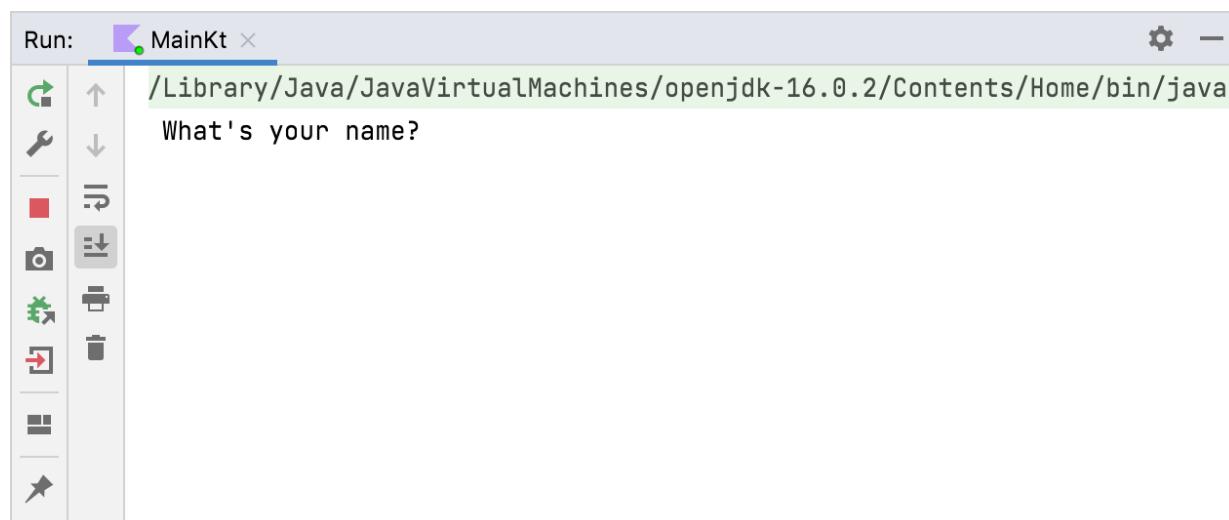
Run the application

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'**.

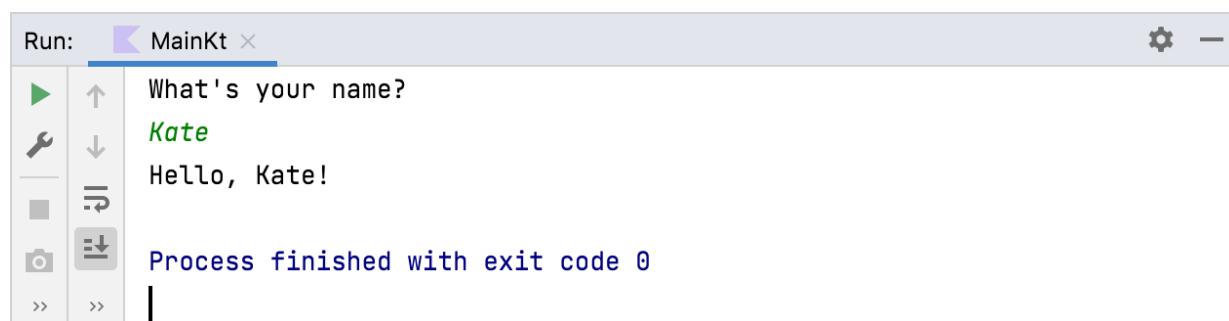
1.5.30 的新特性



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？

1.5.30 的新特性

Once you've created this application, you can start to dive deeper into Kotlin syntax:

- Add sample code from [Kotlin examples](#)
- Install the [EduTools 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

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。

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
- 字符串模板
- 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
- 单例
- 声明处型变 & 类型投影
- 区间表达式

1.5.30 的新特性

- 操作符重载
- 伴生对象
- 数据类
- 分离用于只读与可变集合的接口
- 协程

下一步做什么？

了解下[在 Java 与 Kotlin 中如何处理字符串的典型任务。](#)

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。`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 setFirstDayOfWeek()
    }
    if (!calendar.isLenient()) { // 调用 isLenient()
        calendar.isLenient = true // 调用 setLenient()
    }
}
```

1.5.30 的新特性

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。

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 可能会抛出异常
```

1.5.30 的新特性

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.nullness`)
- `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

1.5.30 的新特性

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> {}
```

1.5.30 的新特性

```
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> {}
```

1.5.30 的新特性

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). Pass the `-Xtype-enhancement-improvements-strict-mode` compiler option to report errors in Kotlin code that uses nullability which deviates from the nullability annotations from Java.

Note: If a nullability annotation supports other targets that are applicable to a type in addition to the `TYPE_USE` target, then `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：

1.5.30 的新特性

```
@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。

1.5.30 的新特性

```
@Nonnull
@TypeQualifierDefault({ElementType.METHOD, ElementType.PARAMETER})
public @interface NonNullApi {
}

@Nonnull(when = When.MAYBE)
@TypeQualifierDefault({ElementType.METHOD, ElementType.PARAMETER, ElementType.TYPE})
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?): St

    // 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!): Str
}
```

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）：

1.5.30 的新特性

- `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。

1.5.30 的新特性

- `-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
<code>byte</code>	<code>kotlin.Byte</code>
<code>short</code>	<code>kotlin.Short</code>
<code>int</code>	<code>kotlin.Int</code>
<code>long</code>	<code>kotlin.Long</code>
<code>char</code>	<code>kotlin.Char</code>
<code>float</code>	<code>kotlin.Float</code>
<code>double</code>	<code>kotlin.Double</code>
<code>boolean</code>	<code>kotlin.Boolean</code>

一些非原生的内置类型也会作映射：

1.5.30 的新特性

Java 类型	Kotlin 类型
<code>java.lang.Object</code>	<code>kotlin.Any!</code>
<code>java.lang.Cloneable</code>	<code>kotlin.Cloneable!</code>
<code>java.lang.Comparable</code>	<code>kotlin.Comparable!</code>
<code>java.lang.Enum</code>	<code>kotlin.Enum!</code>
<code>java.lang.annotation.Annotation</code>	<code>kotlin.Annotation!</code>
<code>java.lang.CharSequence</code>	<code>kotlin(CharSequence!</code>
<code>java.lang.String</code>	<code>kotlin.String!</code>
<code>java.lang.Number</code>	<code>kotlin.Number!</code>
<code>java.lang.Throwable</code>	<code>kotlin.Throwable!</code>

Java 的装箱原始类型映射到可空的 Kotlin 类型：

Java type	Kotlin type
<code>java.lang.Byte</code>	<code>kotlin.Byte?</code>
<code>java.lang.Short</code>	<code>kotlin.Short?</code>
<code>java.lang.Integer</code>	<code>kotlin.Int?</code>
<code>java.lang.Long</code>	<code>kotlin.Long?</code>
<code>java.lang.Character</code>	<code>kotlin.Char?</code>
<code>java.lang.Float</code>	<code>kotlin.Float?</code>
<code>java.lang.Double</code>	<code>kotlin.Double?</code>
<code>java.lang.Boolean</code>	<code>kotlin.Boolean?</code>

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）：

1.5.30 的新特性

Java 类型	Kotlin 只读类型	Kotlin 可变类型	加
Iterator<T>	Iterator<T>	MutableIterator<T>	(Mutable
Iterable<T>	Iterable<T>	MutableIterable<T>	(Mutabl
Collection<T>	Collection<T>	MutableCollection<T>	(Mutabl
Set<T>	Set<T>	MutableSet<T>	(Mutabl
List<T>	List<T>	MutableList<T>	(Mutabl
ListIterator<T>	ListIterator<T>	MutableListIterator<T>	(Mutabl
Map<K, V>	Map<K, V>	MutableMap<K, V>	(Mutabl
Map.Entry<K, V>	Map.Entry<K, V>	MutableMap.MutableEntry<K,V>	(Mutabl (Mutabl

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：

1.5.30 的新特性

```
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：

1.5.30 的新特性

```
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

1.5.30 的新特性

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`：

1.5.30 的新特性

```
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

1.5.30 的新特性

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") })
```

1.5.30 的新特性

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。

1.5.30 的新特性

```
// 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() { /*....*/ }
```

1.5.30 的新特性

```
// 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

1.5.30 的新特性

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：

1.5.30 的新特性

```
// 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 const = 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，编译器既会在相应用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
```

1.5.30 的新特性

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：

1.5.30 的新特性

```
// 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");
    }
}
```

1.5.30 的新特性

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 mode for default methods

If there are clients that use your Kotlin interfaces compiled without the new `-Xjvm-default=all` option, then they can be incompatible with the same code compiled with this option.

To avoid breaking the compatibility with such clients, compile your Kotlin code in the *compatibility mode* by specifying the `-Xjvm-default=all-compatibility` compiler option. In this case, all the code that uses the previous version will work fine with the new one. However, the compatibility mode adds some overhead to the resulting bytecode size.

There is no need to consider compatibility for new interfaces, as no clients have used them before. You can minimize the compatibility overhead by excluding these interfaces from the compatibility mode. To do this, annotate them with the `@JvmDefaultWithoutCompatibility` annotation. Such interfaces compile the same way as with `-Xjvm-default=all`.

Additionally, in the `all-compatibility` mode you can use `@JvmDefaultWithoutCompatibility` to annotate all interfaces which are not exposed in the public API and therefore aren't used by the existing clients.

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` 的顶层声明编译成包级局部声明
- `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)

1.5.30 的新特性

- `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：

1.5.30 的新特性

```
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")
@Setter:JvmName("changeX")
var x: Int = 23
```

生成重载

通常，如果你写一个有默认参数值的 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 =
    @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

1.5.30 的新特性

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两函数转换成 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) { ..... }
```

1.5.30 的新特性

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——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——教程

This tutorial walks you through the process of creating a simple application with Spring Boot and adding a database to store the information.

In this tutorial, you will:

- Create an application with an HTTP endpoint
- Learn how to return a data objects list in the JSON format
- Create a database for storing objects
- Use endpoints for writing and retrieving database objects

You can download and explore the [completed project](#) or watch a video of this tutorial:

YouTube 视频: [Spring Time in Kotlin. Getting Started](#)

Before you start

Download and install the latest version of [IntelliJ IDEA](#).

Bootstrap the project

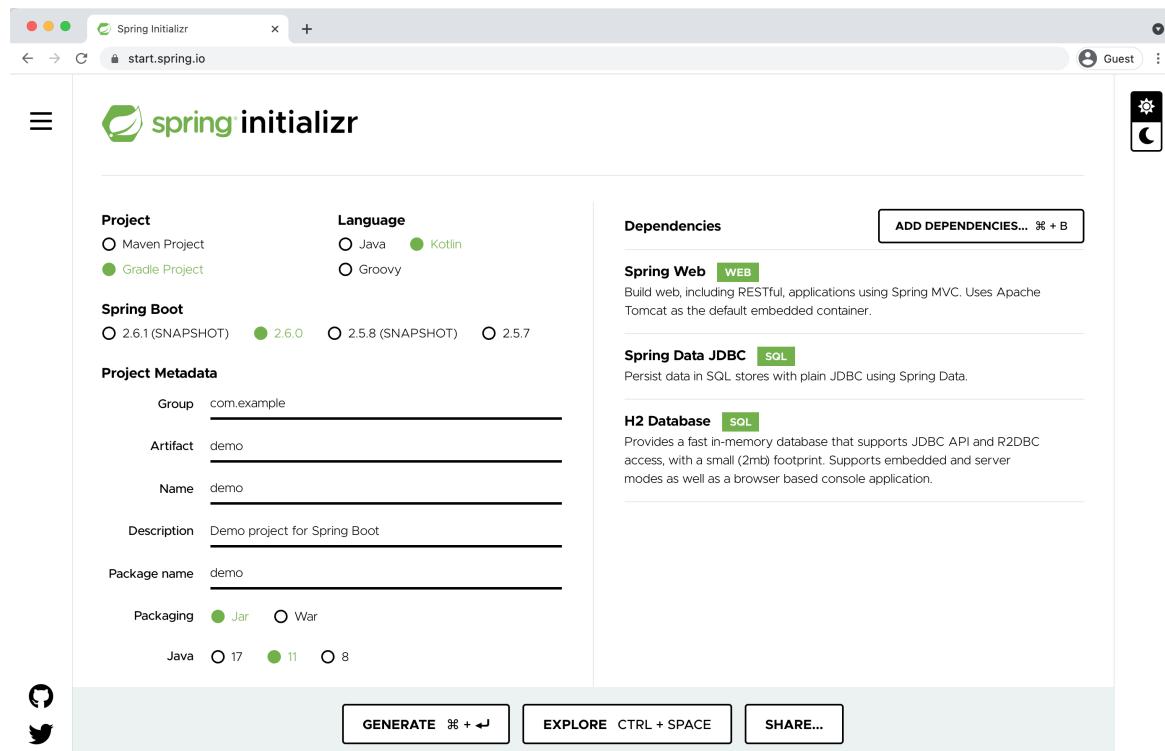
Use Spring Initializr to create a new project:

You can also create a new project using [IntelliJ IDEA](#) with the Spring Boot plugin



1. Open [Spring Initializr](#). This link opens the page with the project settings for this tutorial already filled in. This project uses **Gradle**, **Kotlin**, **Spring Web**, **Spring Data JDBC**, and **H2 Database**:

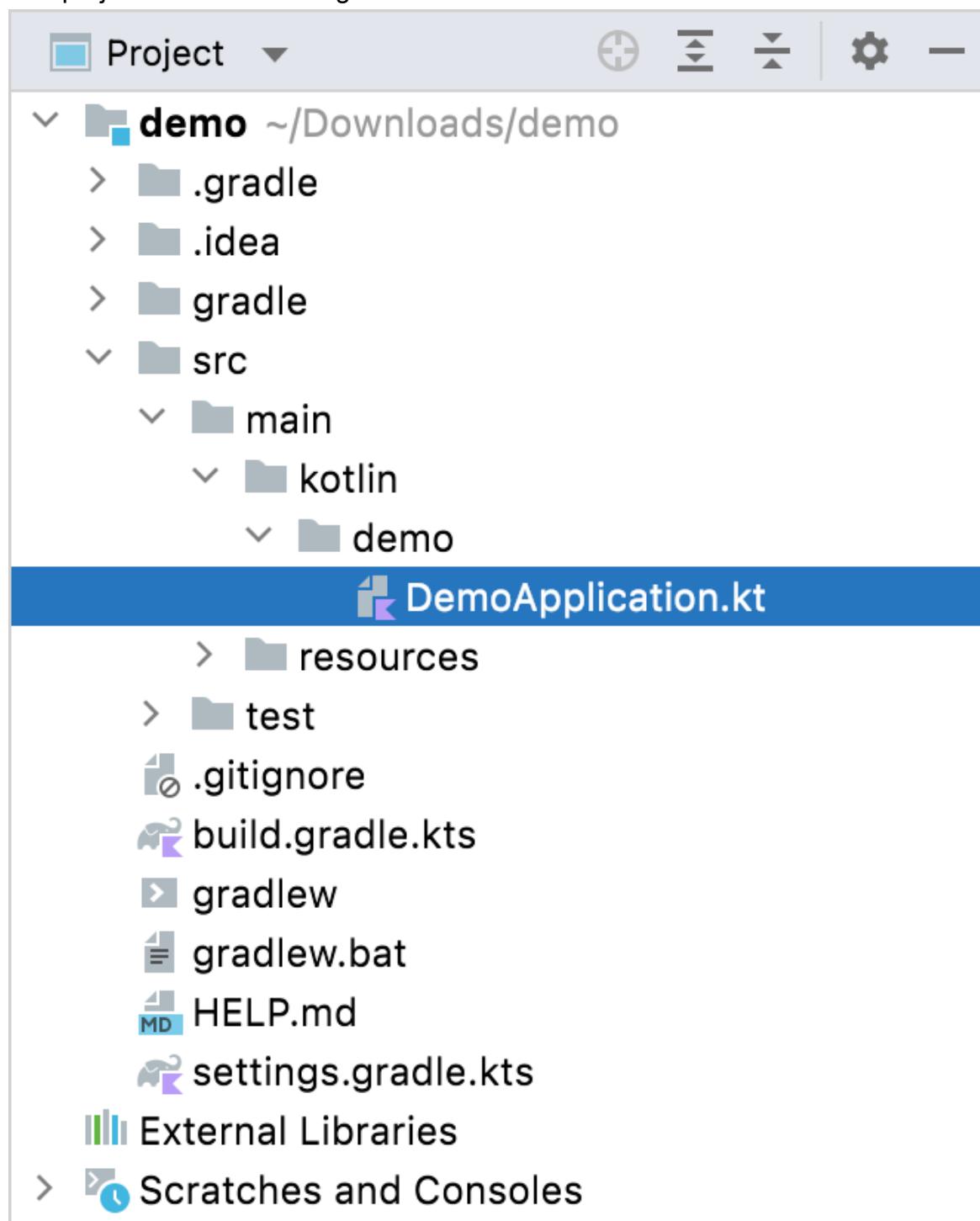
1.5.30 的新特性



2. Click **GENERATE** at the bottom of the screen. Spring Initializr will generate the project with the specified settings. The download starts automatically.
3. Unpack the **.zip** file and open it in IntelliJ IDEA.

1.5.30 的新特性

The project has the following structure:



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 build file

Open the `build.gradle.kts` file.

1.5.30 的新特性

This 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.

Explore the Spring Boot application

Open the `DemoApplication.kt` file:

```
package demo

import org.springframework.boot.autoconfigure.SpringBootApplication
import org.springframework.boot.runApplication

@SpringBootApplication
class DemoApplication

fun main(args: Array<String>) {
    runApplication<DemoApplication>(*args)
}
```

Note that the Kotlin application file differs from a Java application file:

- While Spring Boot looks for a public static `main()` method, the Kotlin application uses a [top-level function](#) defined outside `DemoApplication` class.
- The `DemoApplication` class is not declared as `open`, since the [kotlin-spring](#) plugin does that automatically.

Create a data class and a controller

To create an endpoint, add a [data class](#) and a controller to your project:

1. In the `DemoApplication.kt` file, create a `Message` data class with two properties:

`id` and `text`:

```
data class Message(val id: String?, val text: String)
```

2. In the same file, create a `MessageResource` class which will serve the requests and return a JSON document containing a collection of `Message` objects:

1.5.30 的新特性

```
@RestController
class MessageResource {
    @GetMapping
    fun index(): List<Message> = listOf(
        Message("1", "Hello!"),
        Message("2", "Bonjour!"),
        Message("3", "Privet!"),
    )
}
```

Full code of the `DemoApplication.kt` :

```
package demo

import org.springframework.boot.autoconfigure.SpringBootApplication
import org.springframework.boot.runApplication
import org.springframework.data.annotation.Id
import org.springframework.web.bind.annotation.GetMapping
import org.springframework.web.bind.annotation.RestController

@SpringBootApplication
class DemoApplication

fun main(args: Array<String>) {
    runApplication<DemoApplication>(*args)
}

@RestController
class MessageResource {
    @GetMapping
    fun index(): List<Message> = listOf(
        Message("1", "Hello!"),
        Message("2", "Bonjour!"),
        Message("3", "Privet!"),
    )
}

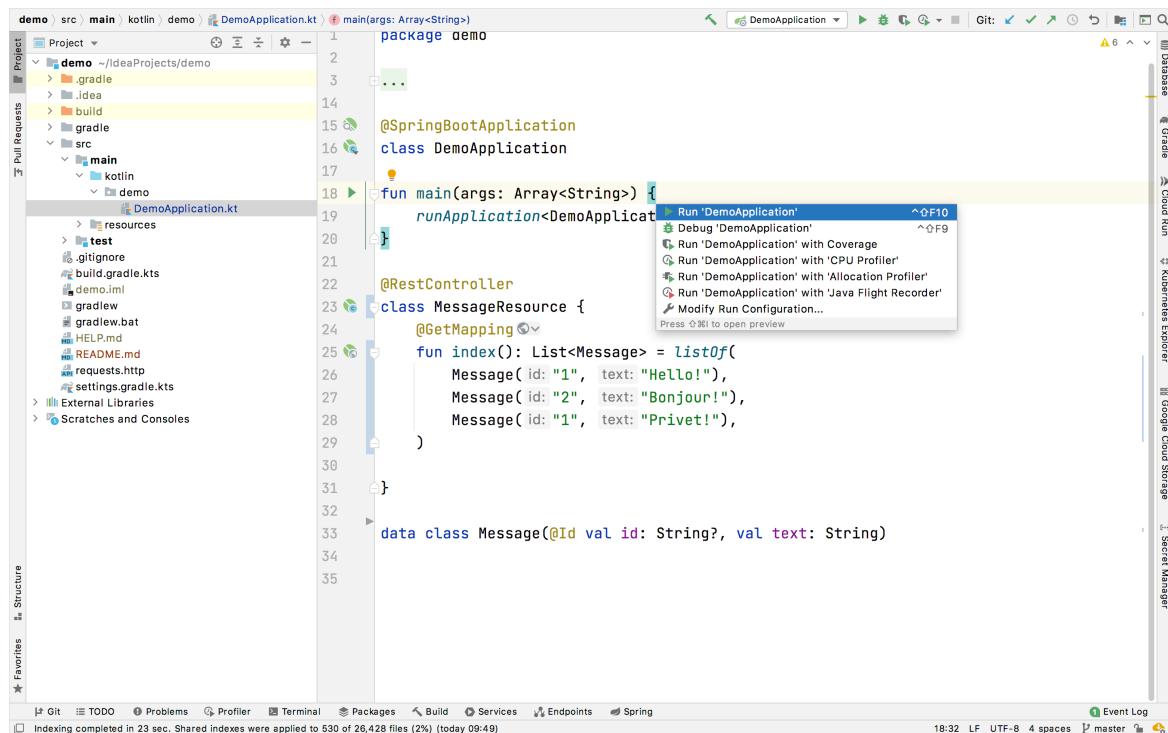
data class Message(val id: String?, val text: String)
```

Run the application

The application is now ready to run:

1. Click the green **Run** icon in the gutter beside the `main()` method or use the **Alt+Enter** shortcut to invoke the launch menu in IntelliJ IDEA:

1.5.30 的新特性

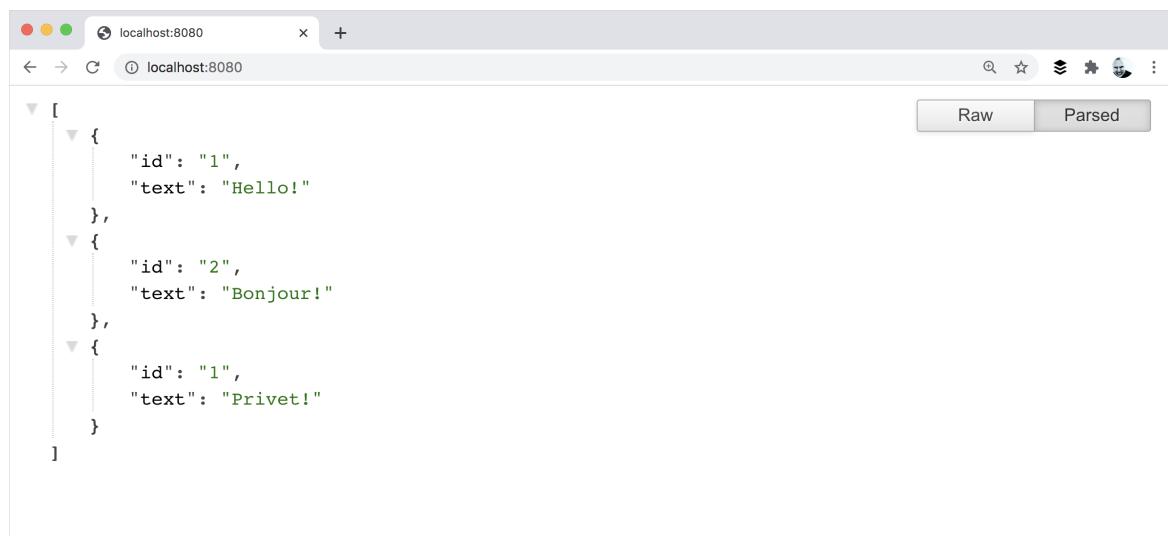


You can also run the `./gradlew bootRun` command in the terminal.



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:



Add database support

To use a database in your application, first create two endpoints: one for saving messages and one for retrieving them:

1.5.30 的新特性

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:

```
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)
```

2. Use the [Spring Data Repository API](#) to access the database:

```
import org.springframework.data.jdbc.repository.query.Query
import org.springframework.data.repository.CrudRepository

interface MessageRepository : CrudRepository<Message, String>{

    @Query("select * from messages")
    fun findMessages(): List<Message>
}
```

When you call the `findMessages()` method on an instance of `MessageRepository`, it will execute the corresponding database query:

```
select * from messages
```

This query retrieves a list of all `Message` objects in the database table.

3. Create the `MessageService` class:

```
import org.springframework.stereotype.Service

@Service
class MessageService(val db: MessageRepository) {

    fun findMessages(): List<Message> = db.findMessages()

    fun post(message: Message){
        db.save(message)
    }
}
```

This class contains two methods:

1.5.30 的新特性

- `post()` for writing a new `Message` object to the database
- `findMessages()` for getting all the messages from the database

4. Update the `MessageResource` class:

```
```kotlin import org.springframework.web.bind.annotation.RequestBody import  
org.springframework.web.bind.annotation.PostMapping
```

```
@RestController class MessageResource(val service: MessageService) {
 @GetMapping fun index(): List = service.findMessages()
```

```
 @PostMapping
 fun post(@RequestBody message: Message) {
 service.post(message)
 }
```

```
}
```

Now it uses `MessageService` to work with the database.

## Configure the database

Configure the database in the application:

1. Create a new folder called `sql` in the `src/main/resources` with the `schema.sql` the database scheme:

```
![Create a new folder](/images/spring-boot-sql-schema.png)
```

2. Update the `src/main/resources/sql/schema.sql` file with the following code:

```
```sql  
CREATE TABLE IF NOT EXISTS messages (  
    id          VARCHAR(60)  DEFAULT RANDOM_UUID() PRIMARY KEY,  
    text        VARCHAR      NOT NULL  
);
```

It creates the `messages` table with two fields: `id` and `text`. The table structure matches the structure of the `Message` class.

1. Open the `application.properties` file located in the `src/main/resources` folder and add the following application properties:

1.5.30 的新特性

```
spring.datasource.driver-class-name=org.h2.Driver  
spring.datasource.url=jdbc:h2:file:./data/testdb  
spring.datasource.username=sa  
spring.datasource.password=password  
spring.datasource.schemaclasspath:sql/schema.sql  
spring.datasource.initialization-mode=always
```

These settings enable the database for the Spring Boot application. See the full list of common application properties in the [Spring documentation](#).

Execute HTTP requests

You should use an HTTP client to work with previously created endpoints. In IntelliJ IDEA, you can 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 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

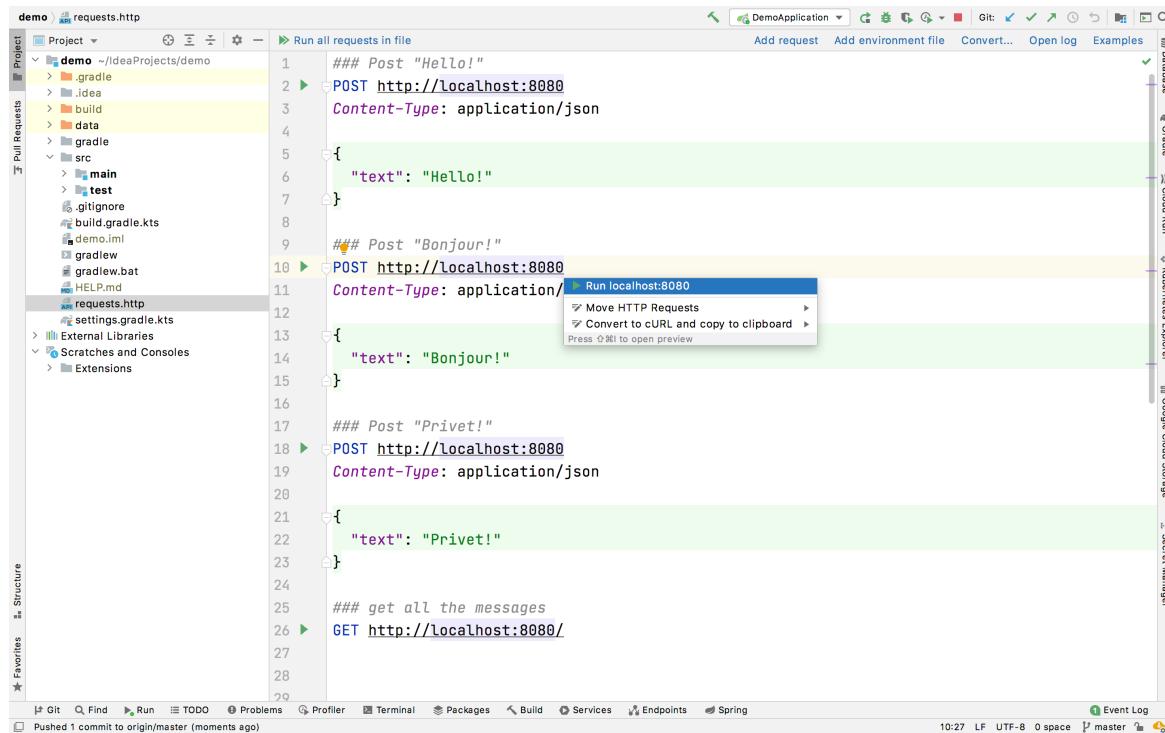
```
{ "text": "Privet!" }
```

### Get all the messages

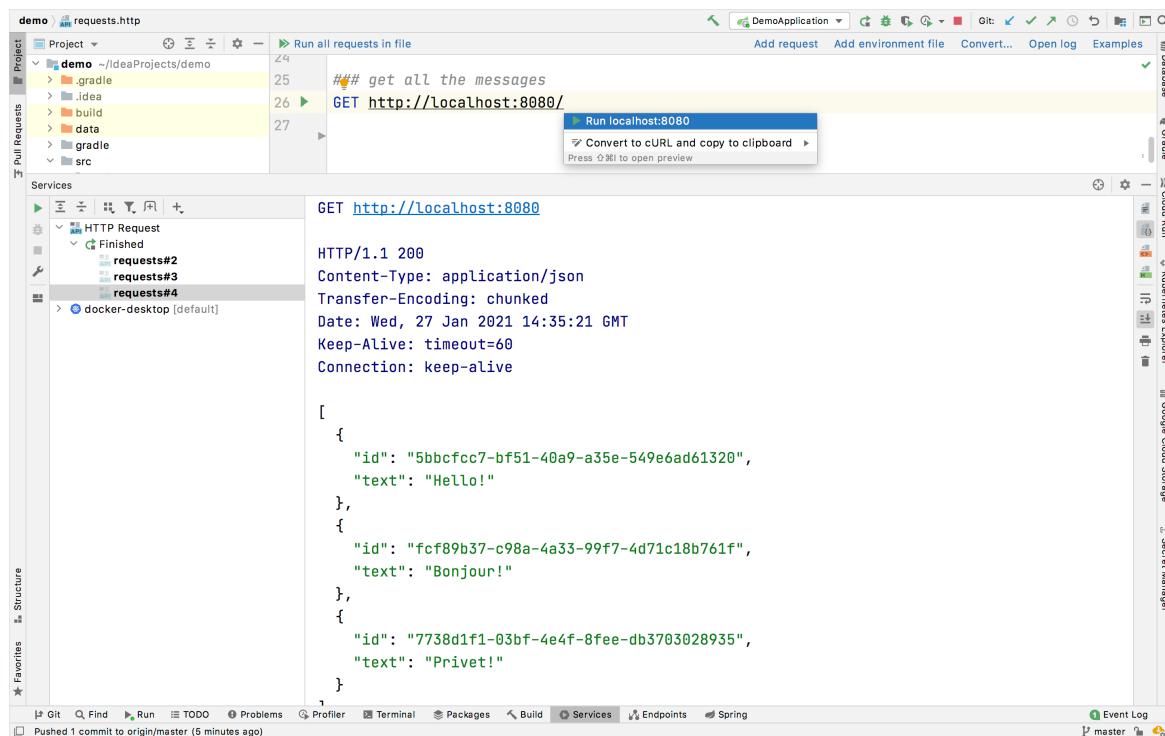
### 1.5.30 的新特性

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

### 1.5.30 的新特性

You can also use any other HTTP client or cURL command-line tool. For example, you can run the following commands in the terminal to get the same result:

```
curl -X POST --location "http://localhost:8080" -H "Content-Type: application/json"
curl -X POST --location "http://localhost:8080" -H "Content-Type: application/json"
curl -X POST --location "http://localhost:8080" -H "Content-Type: application/json"
curl -X GET --location "http://localhost:8080"
```

## 下一步做什么？

For more tutorials, check out the Spring website:

- [Building web applications with Spring Boot and Kotlin](#)
- [Spring Boot with Kotlin Coroutines and RSocket](#)

# 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 平台上用 JUnit 测试代码——教程

This tutorial will show you how to write a simple unit test and run it with the Gradle build tool.

The example in the tutorial has the `kotlin.test` library under the hood and runs the test using JUnit.

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 already have a project, [create one](#).

Specify **JUnit 5** as your test framework when creating your project.



2. Open the `build.gradle(.kts)` file and add the following dependency to the Gradle configuration. This dependency will allow 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】

## 1.5.30 的新特性

```
tasks.test {
 useJUnitPlatform()
}
```

### 【Groovy】

```
test {
 useJUnitPlatform()
}
```

If you created the project using the **New Project** wizard, the task will be added automatically.



## 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 will print `Hello, World!`.

2. Create the `Sample` class with the `sum()` function that adds two integers together:

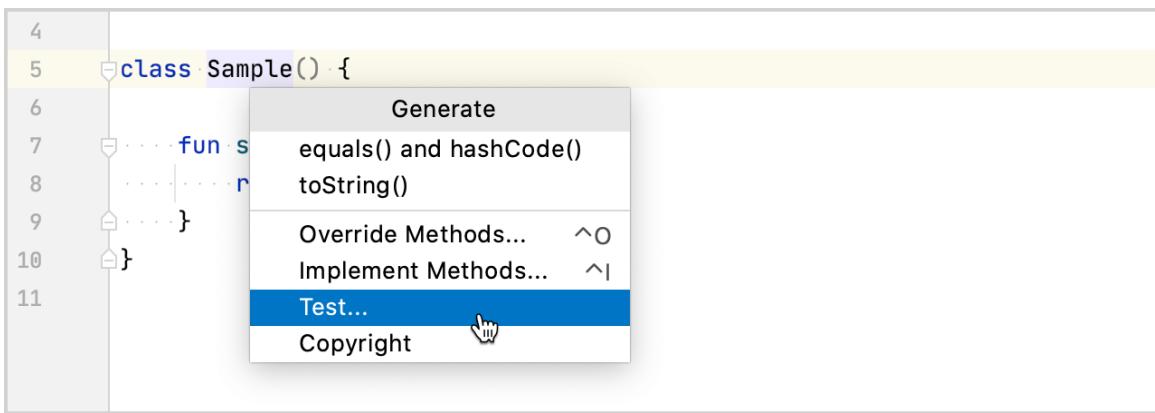
```
class Sample() {

 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.

### 1.5.30 的新特性



- 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`.



- 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 kotlin.test.Test
import kotlin.test.assertEquals

internal class SampleTest {

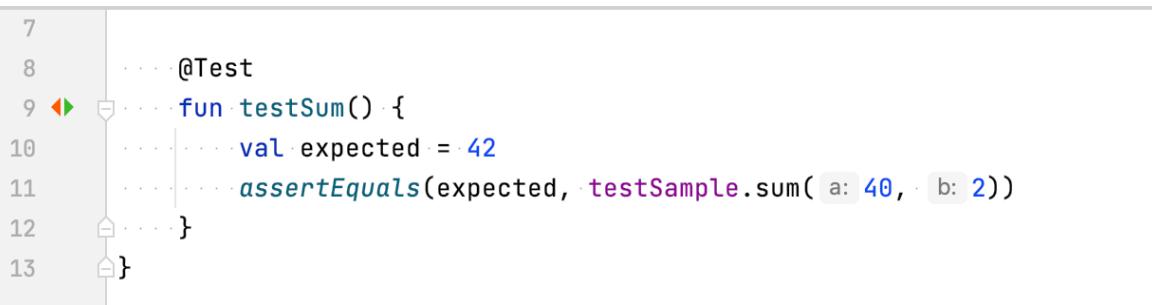
 private val testSample: Sample = Sample()

 @Test
 fun testSum() {
 val expected = 42
 assertEquals(expected, testSample.sum(40, 2))
 }
}
```

## Run a test

- Run the test using the gutter icon.

### 1.5.30 的新特性

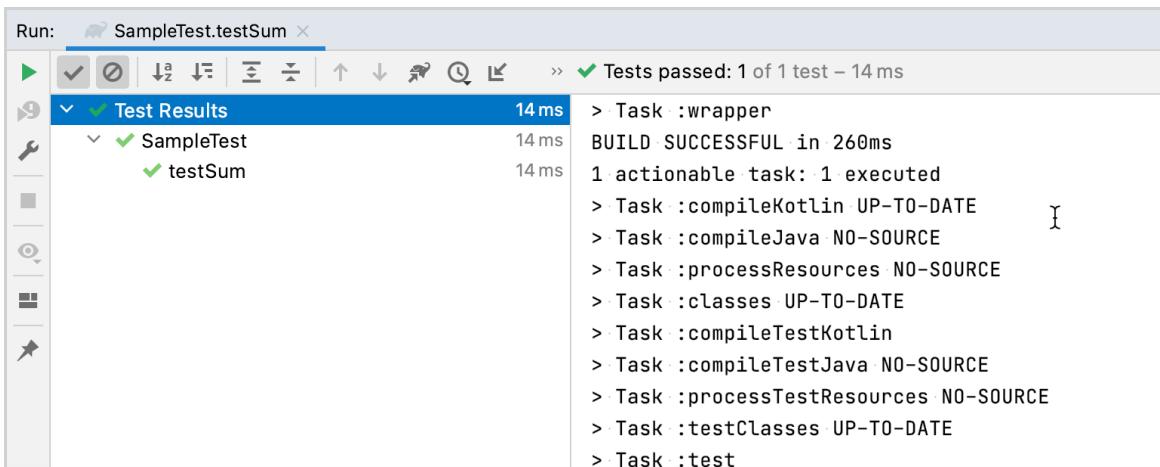


```
7
8 ... @Test
9 fun testSum() {
10 ... val expected = 42
11 ... assertEquals(expected, testSample.sum(a: 40, b: 2))
12 }
13 }
```

You can also run all project tests via the command-line interface using the `./gradlew check` command.



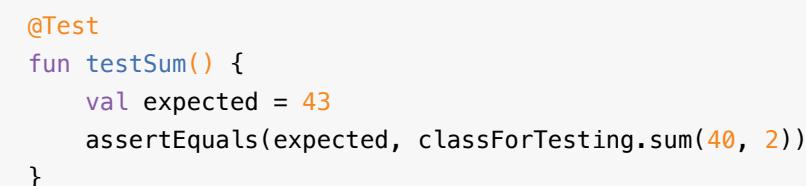
### 2. Check the result in the Run tool window:



The Run tool window displays the test results for `SampleTest.testSum`. It shows 1 test passed in 14 ms. The test was part of the `SampleTest` class, specifically the `testSum` method. The build was successful in 260ms, and there were 1 actionable task executed. The log output includes tasks like `:wrapper`, `:compileKotlin`, `:compileJava`, `:processResources`, `:classes`, `:compileTestKotlin`, `:compileTestJava`, `:processTestResources`, `:testClasses`, and `:test`.

The test function was executed successfully.

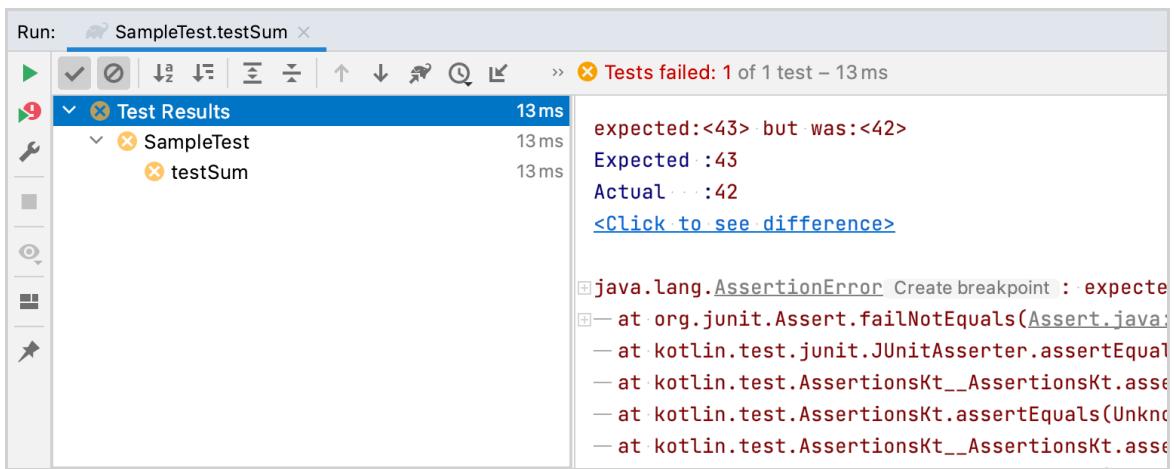
### 3. Make sure that the test works correctly by changing the `expected` variable value to 43:



```
@Test
fun testSum() {
 val expected = 43
 assertEquals(expected, classForTesting.sum(40, 2))
}
```

### 4. Run the test again and check the result:

### 1.5.30 的新特性



The test execution failed.

## What's next

Once you've finished your first test, you can:

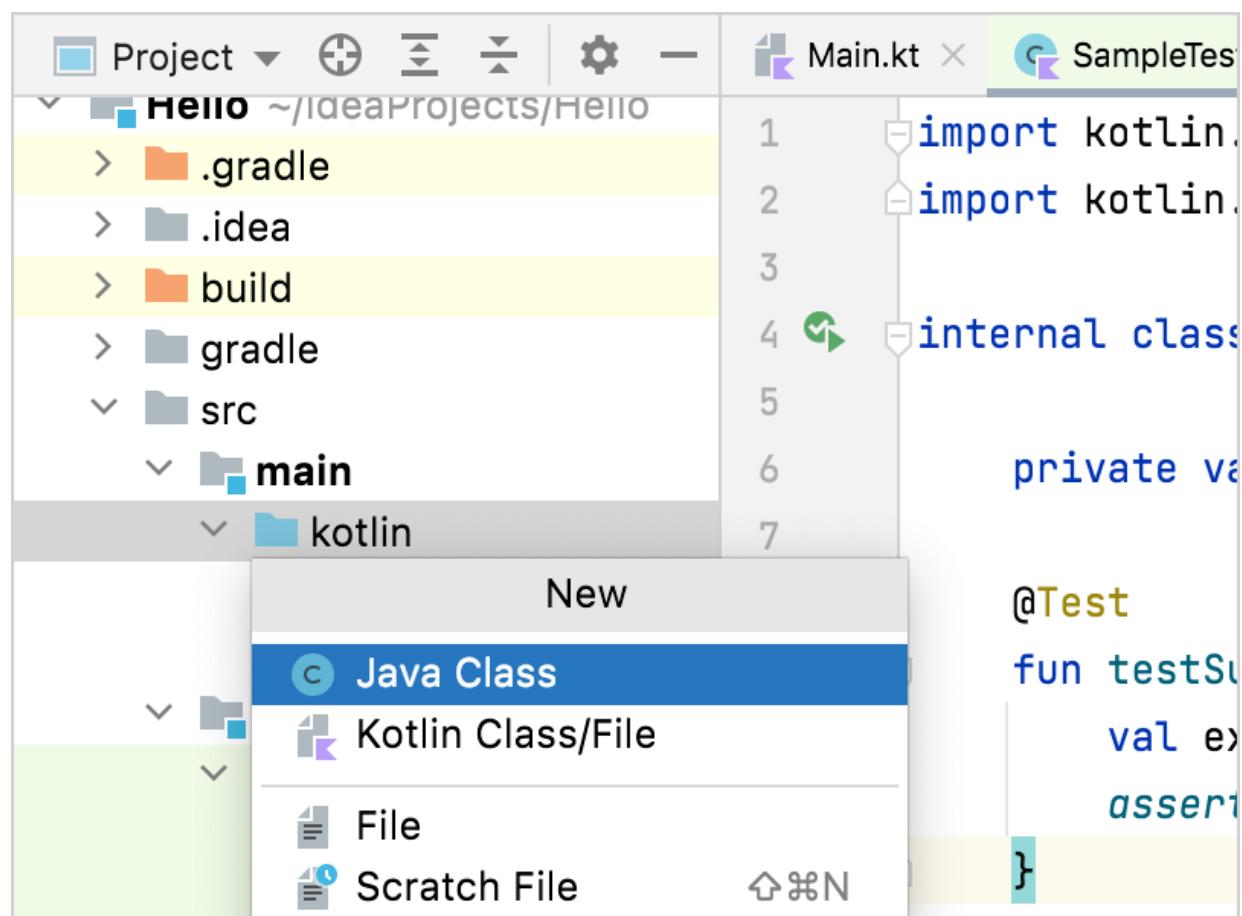
- Try to write another test using other `kotlin.test` functions. For example, you could use the `assertNotEquals()` function.
- [Create your first application](#) with Kotlin and Spring Boot.
- Watch [these video tutorials](#) on YouTube, which demonstrate how to use Spring Boot with Kotlin and JUnit 5.

# 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：

### 1.5.30 的新特性

```
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);
 }
}
```

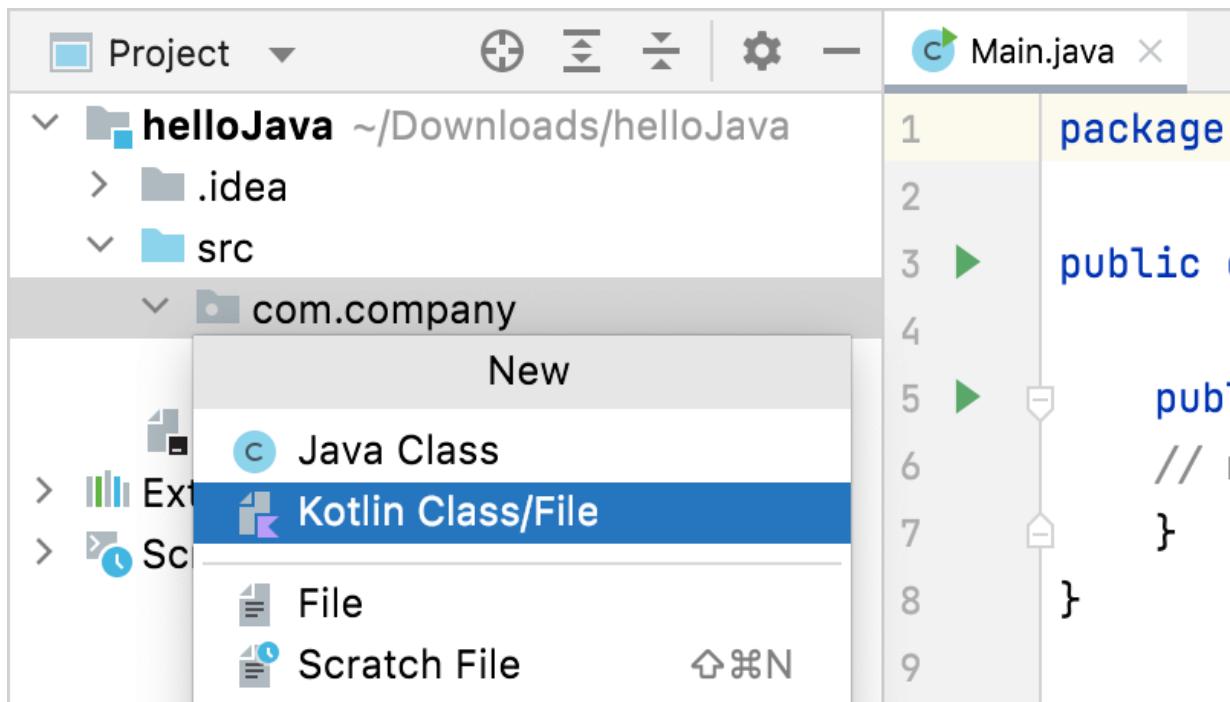
我们可以在 Kotlin 中像使用任何其他类型一样调用它。

```
val customer = Customer("Phase")
println(customer.name)
println(customer.placeOrder())
```

## 将 Kotlin 源代码添加到现有 Java 项目中

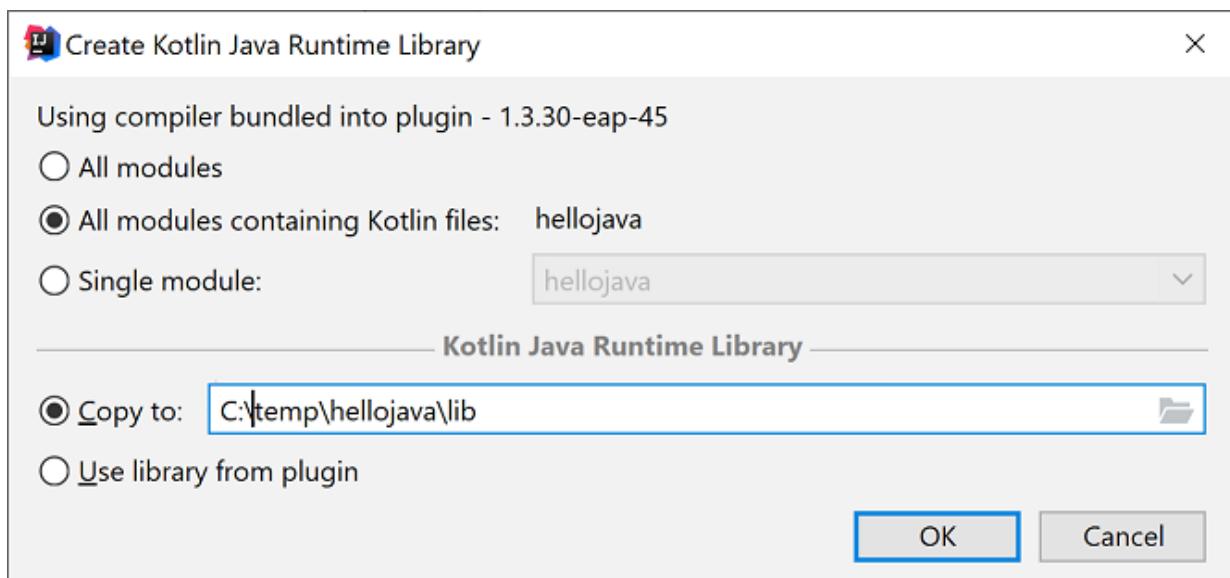
将 Kotlin 文件添加到现有 Java 项目的过程几乎相同。

### 1.5.30 的新特性



如果这是你第一次将 Kotlin 文件添加到此项目中，IntelliJ IDEA 会提示你添加所需的 Kotlin 运行时。对于 Java 项目，将 Kotlin 运行时配置为 **Kotlin Java Module**。

下一步是决定要配置哪些模块（如果项目有多个模块）以及是否想要将运行时库添加到项目中或使用当前 Kotlin 插件提供的那些库。

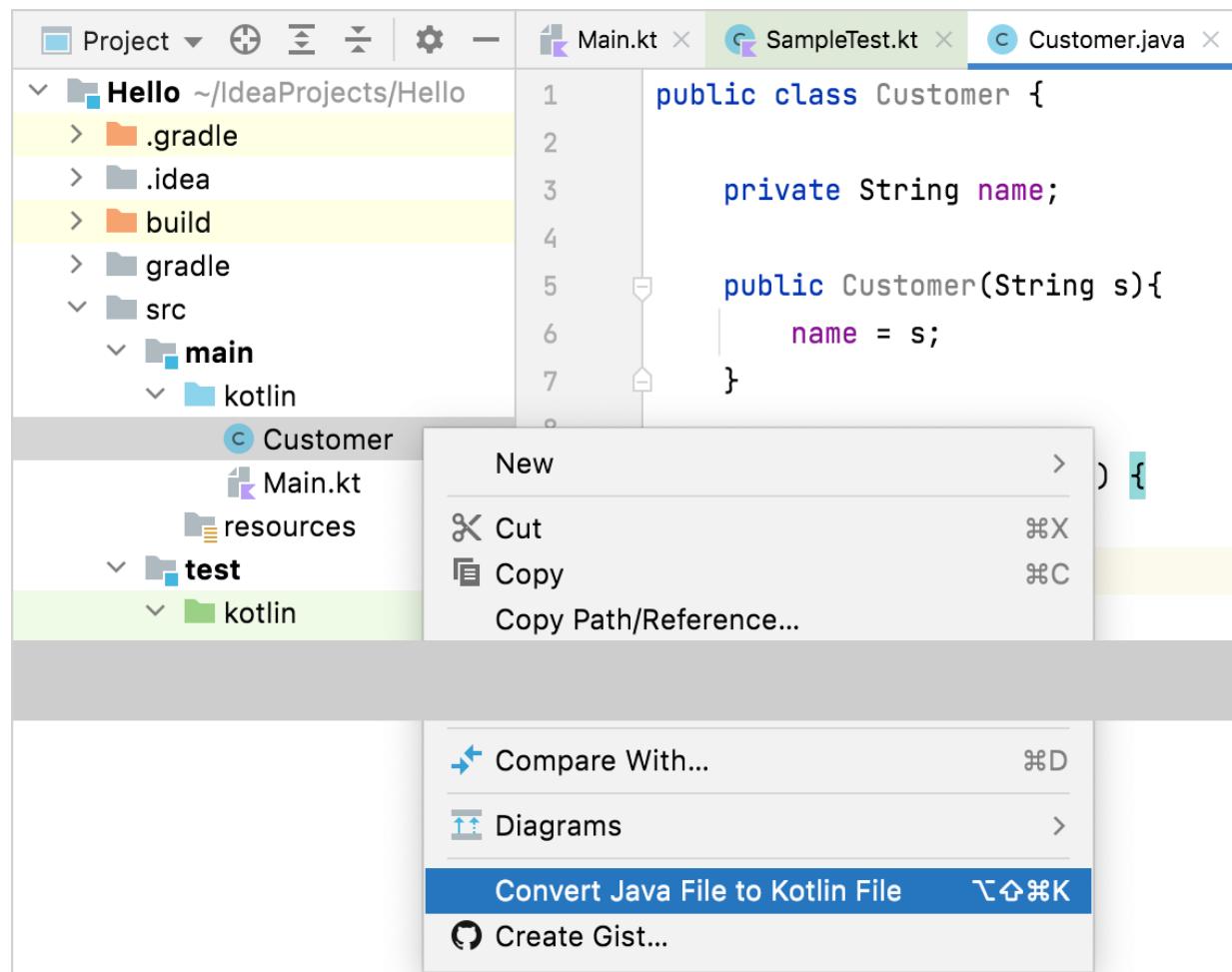


你也可以在 **Tools | Kotlin | Configure Kotlin in Project** 中手动打开 Kotlin 运行时配置。

## 使用 J2K 将现有 Java 文件转换为 Kotlin 文件

### 1.5.30 的新特性

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:

### 1.5.30 的新特性

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

### 1.5.30 的新特性

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

# 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](#):

### 1.5.30 的新特性

```
// 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);
```

### 1.5.30 的新特性

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{ "${-it}" }
 println(invertedOddNumbers)
 //sampleEnd
}
```

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:

### 1.5.30 的新特性

```
// 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 [replaceFirst\(\)](#) and the [replaceAll\(\)](#) functions. The `replaceAll()` function in this case accepts the regular expression `##$`, which defines a string ending with `##`:

```
// Java
String input = "##place##holder##";
String result = input.replaceFirst("##", "").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:

### 1.5.30 的新特性

```
// 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*""") // raw 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
}
```

### 1.5.30 的新特性

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,
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,
 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

### 1.5.30 的新特性

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
""".stripIndent();
System.out.println(result);
```

The output:

```
StringsExamples (test.java) 6 ms /Library/Java/JavaVirtualMachines/jdk-15.0.2.jdk/Contents/Home/bin/java ...
 ✓ java15MultilineExample 6 ms
 Kotlin
 Java
 ← Extra empty line as the triple-quote is on the next line
 Process finished with exit code 0
```

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.

```
fun main() {
//sampleStart
 // Kotlin
 val result = """
 Kotlin
 Java
 """.trimIndent()
 println(result)
//sampleEnd
}
```

## 1.5.30 的新特性

The output:



```
StringsExamples (test.kotlin) 3ms
 multilineExample 3ms
 Kotlin
 Java
 ← No extra empty line
 Process finished with exit code 0
```

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 `dequeue` 的操作

### 1.5.30 的新特性

描述	共有操作	更多 Kotlin 替代方式
Add an element or elements	<code>add()</code> , <code>addAll()</code>	Use the <code>plusAssign ( += ) operator</code> : <code>collection += element</code> , <code>collection += anotherCollection</code> .
Check whether a collection contains an element or elements	<code>contains()</code> , <code>containsAll()</code>	Use the <code>in keyword</code> to call <code>contains()</code> in the operator form: <code>element in collection</code> .
Check whether a collection is empty	<code>isEmpty()</code>	Use <code>isNotEmpty()</code> to check whether a collection is not empty.
Remove under a certain condition	<code>removeIf()</code>	
Leave only selected elements	<code>retainAll()</code>	
Remove all elements from a collection	<code>clear()</code>	
Get a stream from a collection	<code>stream()</code>	Kotlin has its own way to process streams: <code>sequences</code> and methods like <code>map()</code> and <code>filter()</code> .
Get an iterator from a collection	<code>iterator()</code>	

## 对 map 的操作

### 1.5.30 的新特性

描述	共有操作	更多 Kotlin 替代方式
Add an element or elements	<code>put()</code> , <code>putAll()</code> , <code>putIfAbsent()</code>	In Kotlin, the assignment <code>map[key] = value</code> behaves the same as <code>put(key, value)</code> . Also, you may use the <code>plusAssign ( += ) operator</code> : <code>map += Pair(key, value)</code> or <code>map += anotherMap</code> .
Replace an element or elements	<code>put()</code> , <code>replace()</code> , <code>replaceAll()</code>	Use the indexing operator <code>map[key] = value</code> instead of <code>put()</code> and <code>replace()</code> .
Get an element	<code>get()</code>	Use the indexing operator to get an element: <code>map[index]</code> .
Check whether a map contains an element or elements	<code>containsKey()</code> , <code>containsValue()</code>	Use the <code>in keyword</code> to call <code>contains()</code> in the operator form: <code>element in map</code> .
Check whether a map is empty	<code>isEmpty()</code>	Use <code>isNotEmpty()</code> to check whether a map is not empty.
Remove an element	<code>remove(key)</code> , <code>remove(key, value)</code>	Use the <code>minusAssign ( -= ) operator</code> : <code>map -= key</code> .
Remove all elements from a map	<code>clear()</code>	
Get a stream from a map	<code>stream()</code> 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	<code>indexOf()</code>	
Get the last index of an element	<code>lastIndexOf()</code>	
Get an element	<code>get()</code>	Use the indexing operator to get an element: <code>list[index]</code> .
Take a sublist	<code>subList()</code>	
Replace an element or elements	<code>set()</code> , <code>replaceAll()</code>	Use the indexing operator instead of <code>set()</code> : <code>list[index] = value</code> .

1.5.30 的新特性

## 略有不同的操作

### 对任意集合类型的操作

### 1.5.30 的新特性

描述	Java	Kotlin
Get a collection's size	<code>size()</code>	<code>count</code> <code>size</code>
Get flat access to nested collection elements	<code>collectionOfCollections.forEach(flatCollection::addAll)</code> or <code>collectionOfCollections.stream().flatMap().collect()</code>	<code>flatOr</code> <code>flatMap</code>
Apply the given function to every element	<code>stream().map().collect()</code>	<code>map()</code>
Apply the provided operation to collection elements sequentially and return the accumulated result	<code>stream().reduce()</code>	<code>reduce</code> <code>fold</code>
Group elements by a classifier and count them	<code>stream().collect(Collectors.groupingBy(classifier, counting()))</code>	<code>eachC</code>
Filter by a condition	<code>stream().filter().collect()</code>	<code>filter</code>
Check whether collection elements satisfy a condition	<code>stream().noneMatch() , stream().anyMatch() , stream().allMatch()</code>	<code>none()</code> <code>any()</code> <code>all()</code>
Sort elements	<code>stream().sorted().collect()</code>	<code>sorter</code>
Take the first N elements	<code>stream().limit(N).collect()</code>	<code>take()</code>
Take elements with a predicate	<code>stream().takeWhile().collect()</code>	<code>takeW</code>

### 1.5.30 的新特性

描述	Java	Kotlin
Skip the first N elements	<code>stream().skip(N).collect()</code>	<code>drop()</code>
Skip elements with a predicate	<code>stream().dropWhile().collect()</code>	<code>dropWhile()</code>
Build maps from collection elements and certain values associated with them	<code>stream().collect(toMap(keyMapper, valueMapper))</code>	<code>associate()</code>

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	<code>sort(null)</code>	<code>sort()</code>
Sort a list into descending order	<code>sort(comparator)</code>	<code>sortDescending()</code>
Remove an element from a list	<code>remove(index)</code> , <code>remove(element)</code>	<code>removeAt(index)</code> , <code>remove(element)</code> or <code>collection -= element</code>
Fill all elements of a list with a certain value	<code>Collections.fill()</code>	<code>fill()</code>
Get unique elements from a list	<code>stream().distinct().toList()</code>	<code>distinct()</code>

## 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.

### 1.5.30 的新特性

- `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
List<String> numbers = new LinkedList<>();
// This list is immutable!
List<String> immutableCollection = Collections.unmodifiableList(numbers);
immutableCollection.add("five"); // Fails in runtime with `UnsupportedOperationException`
```

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:

### 1.5.30 的新特性

```
// 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.

### 1.5.30 的新特性

```
// 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:

### 1.5.30 的新特性

```
// 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;
 }
}

public void compareVersions() {
 var minVersion = new Version(1, 11);
 var maxVersion = new Version(1, 31);

 System.out.println(
 versionInRange(new Version(0, 9), minVersion, maxVersion));
 System.out.println(
 versionInRange(new Version(1, 20), minVersion, maxVersion));
}

public Boolean versionInRange(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:

### 1.5.30 的新特性

```
// 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:

### 1.5.30 的新特性

```
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

### 1.5.30 的新特性

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:

### 1.5.30 的新特性

```
// 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:

### 1.5.30 的新特性

```
// 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()` :

### 1.5.30 的新特性

```
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:

### 1.5.30 的新特性

```
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 % 2 == 0 }
 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<Object>();
 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:

### 1.5.30 的新特性

```
// 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:

### 1.5.30 的新特性

```
// 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.

## JavaScript

- 以 Kotlin/JS for React 入门
- 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
- 将 Kotlin/JS 项目迁移到 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
  - 用 Dukat 生成外部声明
- Kotlin/JS 动手实践实验室

# 以 Kotlin/JS for React 入门

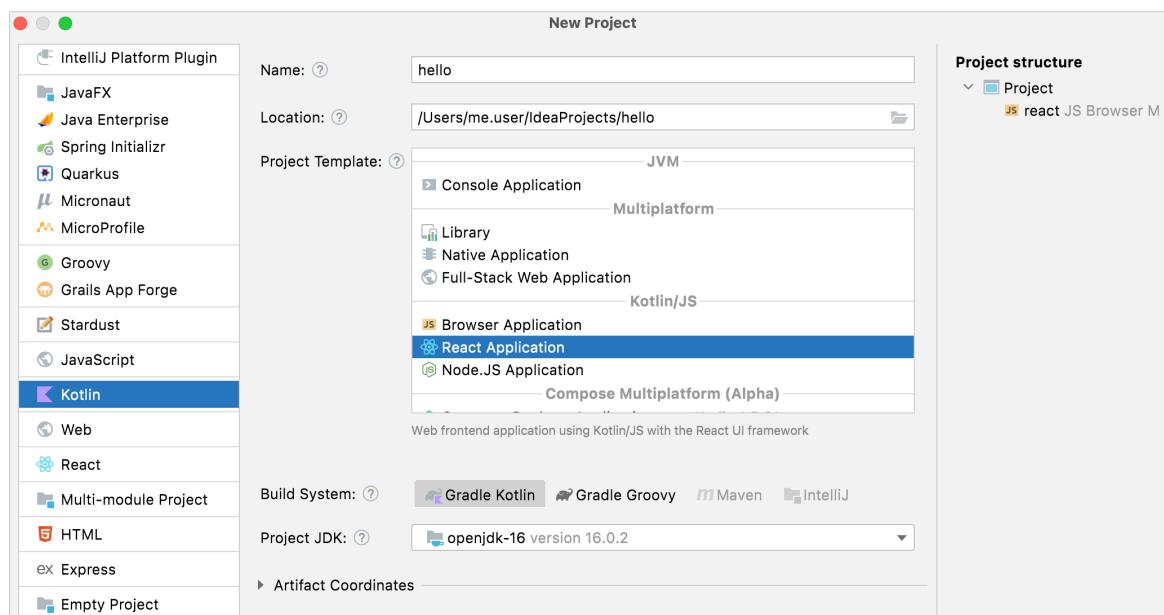
This tutorial demonstrates how to use IntelliJ IDEA for creating a frontend application with Kotlin/JS with React.

To get started, install the latest version of [IntelliJ IDEA](#).

## Create an application

Once you've installed IntelliJ IDEA, it's time to create your first frontend application based on Kotlin/JS with React.

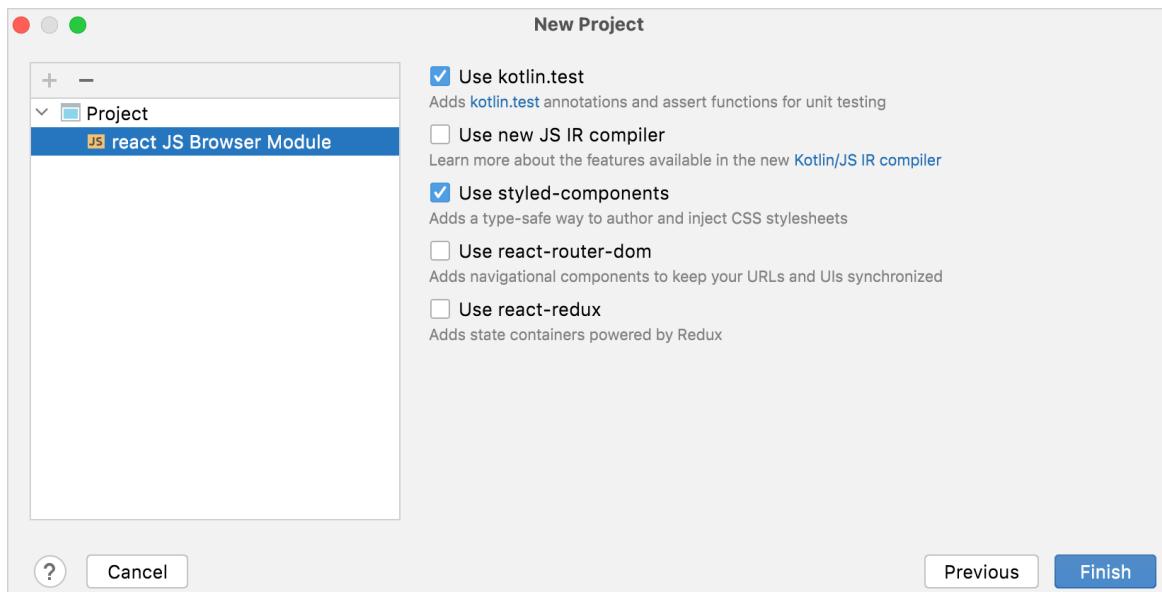
1. In IntelliJ IDEA, select **File | New | Project**.
2. In the panel on the left, select **Kotlin**.
3. Enter a project name, select **React Application** as the project template, and click **Next**.



By default, your project will use Gradle with Kotlin DSL as the build system.

4. Select the **Use styled-components** checkbox and click **Finish**. Your project will open.

### 1.5.30 的新特性

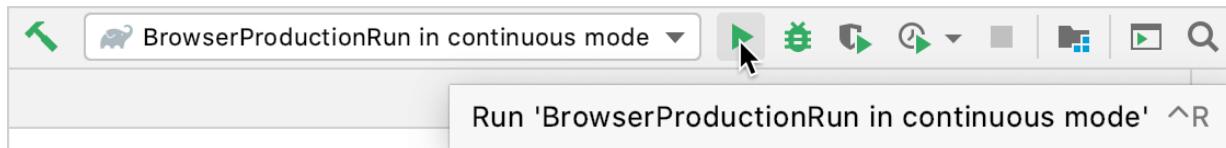


5. Open the `build.gradle.kts` file, the build script created by default based on your configuration. It includes the `kotlin("js")` plugin and dependencies required for your frontend application. Ensure that you use the latest version of the plugin:

```
plugins {
 kotlin("js") version "1.6.10"
}
```

## Run the application

Start the application by clicking **Run** next to the run configuration at the top of the screen.



Your default web browser opens the URL <http://localhost:8080/> with your frontend application.



Enter your name in the text box and accept the greetings from your application!

## Update the application

### 1.5.30 的新特性

## Show your name backwards

1. Open the file `welcome.kt` in `src/main/kotlin`.

The `src` directory contains Kotlin source files and resources. The file `welcome.kt` includes sample code that renders the web page you've just seen.

The screenshot shows the IntelliJ IDEA interface. On the left, the project tree displays a project named 'hello' with subfolders like '.gradle', '.idea', 'build', 'gradle', 'src', 'main', and 'kotlin'. Under 'src/main/kotlin', there are files 'Client.kt' and 'Welcome.kt', with 'Welcome.kt' being the active file. The code in 'Welcome.kt' is as follows:

```
class Welcome(props: WelcomeProps) : RComponent<WelcomeProps, WelcomeState>(props) {
 init {
 state = WelcomeState(props.name)
 }

 override fun RBuilder.render() {
 styledDiv { this: StyledDOMBuilder<DIV>
 css { this: CSSBuilder
 +WelcomeStyles.textContainer
 }
 +"Hello, ${state.name}"
 }
 styledInput { this: StyledDOMBuilder<INPUT>
 css { this: CSSBuilder
 +WelcomeStyles.textInput
 }
 attrs { this: INPUT
 type = InputType.text
 value = state.name
 onChangeFunction = { event ->
 setState(
 WelcomeState(name = (event.target as HTMLInputElement).value)
)
 }
 }
 }
 }
}
```

2. Change the code of `styledDiv` to show your name backwards.

- Use the standard library function `reversed()` to reverse your name.
- Use a [string template](#) for your reversed name by adding a dollar sign `$` and enclosing it in curly braces –  `${state.name.reversed()}` .

```
styledDiv {
 css {
 +WelcomeStyles.textContainer
 }
 +"Hello ${state.name}!"
 +" Your name backwards is ${state.name.reversed()}!"
}
```

3. Save your changes to the file.

4. Go to the browser and enjoy the result.

You will see the changes only if your previous application is still running. If you've stopped your application, [run it again](#).

## 1.5.30 的新特性

Hello Kate! Your name backwards is etaK!

Kate

## Add an image

1. Open the file `welcome.kt` in `src/main/kotlin`.
2. Add a `div` container with a child image element `img` after the `styledInput` block.

Make sure that you import the `react.dom.*` and `styled.*` packages.



```
div {
 img(src = "https://placekitten.com/408/287") {}
}
```

3. Save your changes to the file.
4. Go to the browser and enjoy the result.  
You will only see the changes if your previous application is still running. If you've stopped your application, [run it again](#).

Hello Kotlin/JS! Your name backwards is SJ/niltoK!

Kotlin/JS



## Add a button that changes text

1. Open the file `welcome.kt` in `src/main/kotlin`.
2. Add a `button` element with an `onClickFunction` event handler.

Make sure that you import the package `kotlinx.html.js.*`.



```
button {
 attrs.onClickFunction = {
 setState(
 WelcomeState(name = "Some name")
)
 }
 +"Change name"
}
```

### 1.5.30 的新特性

3. Save your changes to the file.
4. Go to the browser and enjoy the result.

You will only see the changes if your previous application is still running. If you've stopped your application, [run it again](#).

Hello Kotlin/JS! Your name backwards is SJ/niltoK!

Kotlin/JS



## 下一步做什么？

Once you have created your first application, you can go to Kotlin hands-on labs and complete long-form Kotlin/JS tutorials or check out the list of Kotlin/JS sample projects for inspiration. Both types of resources contain useful snippets and patterns and can serve as a nice jump-off point for your own projects.

## Hands-on labs

### 1.5.30 的新特性

- [Building Web Applications with React and Kotlin/JS](#) guides you through the process of building a simple web application using the React framework, shows how a type-safe Kotlin DSL for HTML makes it easy to build reactive DOM elements, and illustrates how to use third-party React components and obtain information from APIs, all while writing the whole application logic in pure Kotlin/JS.
- [Building a Full Stack Web App with Kotlin Multiplatform](#) teaches the concepts behind building an application that targets Kotlin/JVM and Kotlin/JS by building a client-server application that makes use of shared code, serialization, and other multiplatform paradigms. It also provides a brief introduction to working with Ktor both as a server- and client-side framework.

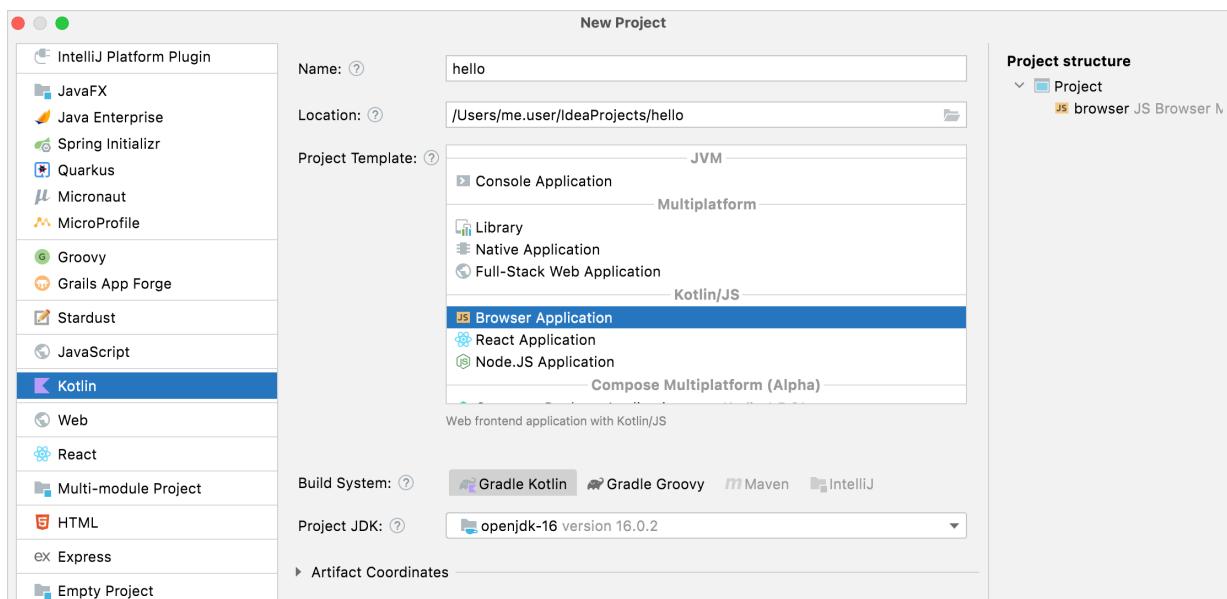
## Sample projects

- [Full-stack Spring collaborative to-do list](#) shows how to create a to-do list for collaborative work using `kotlin-multiplatform` with JS and JVM targets, Spring for the backend, Kotlin/JS with React for the frontend, and RSocket.
- [Kotlin/JS and React Redux to-do list](#) implements the React Redux to-do list using JS libraries (`react`, `react-dom`, `react-router`, `redux`, and `react-redux`) from npm and Webpack to bundle, minify, and run the project.
- [Full-stack demo application](#) guides you through the process of building an app with a feed containing user-generated posts and comments. All data is stubbed by the fakeJSON and JSON Placeholder services.

# 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.js` 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。例如，该插件会在后台下载 `Yarn` 软件包管理器，用于管理 `npm` 依赖，并且可以使用 `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。

要在 IntelliJ IDEA 中创建 Kotlin/JS 项目，请转至 **文件(File) | 新建(New) | 项目(Project)**。然后选择 **Kotlin** 并选择最适合的 Kotlin/JS 目标。不要忘记选择构建脚本的语言：Groovy 或 Kotlin。



另外，还可以在 Gradle 构建文件 (`build.gradle` 或 `build.gradle.kts`) 中手动将 `org.jetbrains.kotlin.js` 插件应用于 Gradle 项目。

## 【Kotlin】

```
plugins {
 kotlin("js") version "1.6.10"
}
```

## 【Groovy】

### 1.5.30 的新特性

```
plugins {
 id 'org.jetbrains.kotlin.js' version '1.6.10'
}
```

Kotlin/JS Gradle 插件可让你在构建脚本的 `kotlin` 部分中管理项目的各个方面。

```
kotlin {
 //...
}
```

在 `kotlin` 部分中，你可以管理以下方面：

- [选择执行环境](#): 浏览器或 Node.js
- [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` 部分添加 `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。使用当前的 Kotlin/JS 编译器时, 这是默认行为, 但是如果在使用 [Kotlin/JS IR 编译器](#)或在 `gradle.properties` 中设置了 `kotlin.js.generate.executable.default=false`。在这些

### 1.5.30 的新特性

情况下，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。（这通常比创建可执行文件快，并且在处理项目的非叶模块时可以是一种优化。）

Kotlin/JS 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 are also an option to use an existing Node.js installation. Learn how to [use pre-installed Node.js](#).

## 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` 部分的传统 Gradle [依赖声明](#)。

### 【Kotlin】

```
dependencies {
 implementation("org.example.myproject", "1.1.0")
}
```

### 【Groovy】

```
dependencies {
 implementation 'org.example.myproject:1.1.0'
}
```

Kotlin/JS Gradle 插件还支持构建脚本的 `kotlin` 部分中特定 `sourceSets` 的依赖声明。

### 【Kotlin】

```
kotlin {
 sourceSets["main"].dependencies {
 implementation("org.example.myproject", "1.1.0")
 }
}
```

### 【Groovy】

### 1.5.30 的新特性

```
kotlin {
 sourceSets {
 main {
 dependencies {
 implementation 'org.example.myproject:1.1.0'
 }
 }
 }
}
```

请注意，在针对 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

所有 Kotlin/JS 项目都必须依赖 Kotlin/JS [标准库](#)，并且是隐含的——无需添加任何构件。

如果你的项目包含用 Kotlin 编写的测试，那么还应该添加 [kotlin.test](#) 依赖项：

### 【Kotlin】

```
dependencies {
 testImplementation(kotlin("test-js"))
}
```

### 【Groovy】

```
dependencies {
 testImplementation 'org.jetbrains.kotlin:kotlin-test-js'
}
```

## 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/JS Gradle 插件使你可以在 Gradle 构建脚本中声明 npm 依赖项，类似于声明其他依赖项的方式。

### 1.5.30 的新特性

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')
}
```

The plugin uses 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. Learn how to [configure Yarn in Kotlin/JS Gradle plugin](#).

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/JS 插件提供了一个 `run` 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，`run` 任务是 `nodeRun` 任务的别名（在 Kotlin 多平台项目中也可用）。

要运行项目，请执行标准生命周期的 `run` 任务，或对应的别名：

### 1.5.30 的新特性

```
./gradlew run
```

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 run --continuous
```

或者

```
./gradlew run -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/JS 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` 部分中添加相应的条目，从而在其他浏览器中运行测试：

## 1.5.30 的新特性

```
kotlin {
 js {
 browser {
 testTask {
 useKarma {
 useIE()
 useSafari()
 useFirefox()
 useChrome()
 useChromeCanary()
 useChromeHeadless()
 usePhantomJS()
 useOpera()
 }
 }
 }
 binaries.executable()
 //
 }
}
```

请注意，Kotlin/JS 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
```

## Karma 配置

### 1.5.30 的新特性

Kotlin/JS 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/JS 插件使用众所周知的 [Webpack](#) 模块捆绑器。

### webpack version

The Kotlin/JS 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/distibution/` 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/JS 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，要添加新的 [webpack loader](#)，请将以下内容添加到 `webpack.config.d` 中的 `.js` 文件中：

```
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/JS 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/distributions` 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

### 1.5.30 的新特性

Kotlin/JS 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。

```
browser {
 commonWebpackConfig {
 cssSupport.enabled = true
 }
 binaries.executable()
}
```

另外，可以为选定的任务添加 CSS 支持，例如 `webpackTask`、`runTask` 与 `testTask`。

```
webpackTask {
 cssSupport.enabled = true
}
runTask {
 cssSupport.enabled = true
}
testTask {
 useKarma {
 //
 webpackConfig.cssSupport.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)`。

您可以使用 `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.

### 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/JS Gradle plugin to use it instead of installing its own Node.js instance.

To use the pre-installed Node.js instance, add the following lines to your

```
build.gradle(.kts) :
```

#### 【Kotlin】

```
rootProject.plugins.withType<org.jetbrains.kotlin.gradle.targets.js.nodejs.NodeJsRootExtension>
 rootProject.the<org.jetbrains.kotlin.gradle.targets.js.nodejs.NodeJsRootExtension>
 // or true for default behavior
}
```

#### 【Groovy】

```
rootProject.plugins.withType(org.jetbrains.kotlin.gradle.targets.js.nodejs.NodeJsRootExtension)
 rootProject.extensions.getByName<org.jetbrains.kotlin.gradle.targets.js.nodejs.NodeJsRootExtension>
}
```

## Yarn

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。

### 1.5.30 的新特性

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/JS Gradle plugin to use it instead of installing its own Yarn instance.

To use the pre-installed Yarn instance, add the following lines to your

```
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>(
 // or true for default behavior
)
```

### 【Groovy】

```
rootProject.plugins.withType(org.jetbrains.kotlin.gradle.targets.js.yarn.YarnPlugin)
 rootProject.extensions.getByName<org.jetbrains.kotlin.gradle.targets.js.yarn.YarnRootExtension>()
```

## 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/distribution` 目录中。

要为项目分发文件设置另一个位置，请在构建脚本中的 `browser` 里添加 `distribution`，然后为它的 `directory` 属性赋值。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】

### 1.5.30 的新特性

```
kotlin {
 js {
 browser {
 distribution {
 directory = File("$projectDir/output/")
 }
 }
 binaries.executable()
 //
 }
}
```

### 【Groovy】

```
kotlin {
 js {
 browser {
 distribution {
 directory = 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/distributions` 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。

### 1.5.30 的新特性

Kotlin/JS 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、许可证与依赖项，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。要将自定义字段添加到您的 `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。

## 疑难解答

When building a Kotlin/JS project using Kotlin 1.3.xx, you may encounter a Gradle error if one of your dependencies (or any transitive dependency) was built using Kotlin 1.4 or higher: Could not determine the dependencies of task '`:client:jsTestPackageJson`'. / Cannot choose between the following variants . This is a known problem, a workaround is provided [here](#).

# 运行 Kotlin/JS

Since Kotlin/JS projects are managed with the Kotlin/JS 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/main/kotlin/App.kt` and fill it with a small "Hello, World"-type code snippet:

```
fun main() {
 console.log("Hello, Kotlin/JS!")
}
```

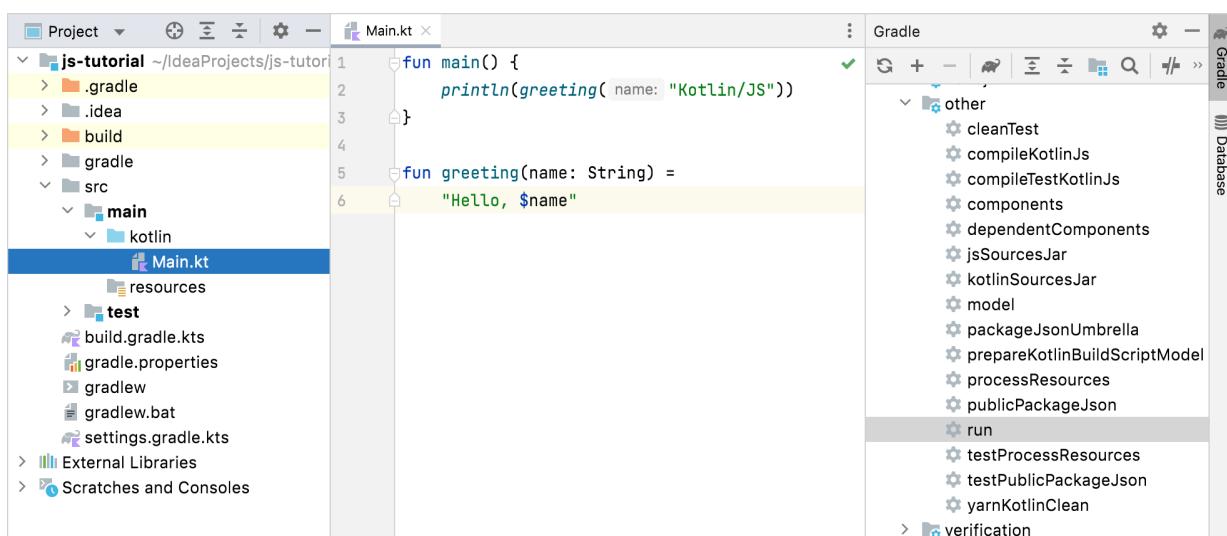
Depending on the target platform, some platform-specific extra setup might be required to run your code for the first time.

## Run the Node.js target

When targeting Node.js with Kotlin/JS, you can simply execute the `run` Gradle task. This can be done for example via the command line, using the Gradle wrapper:

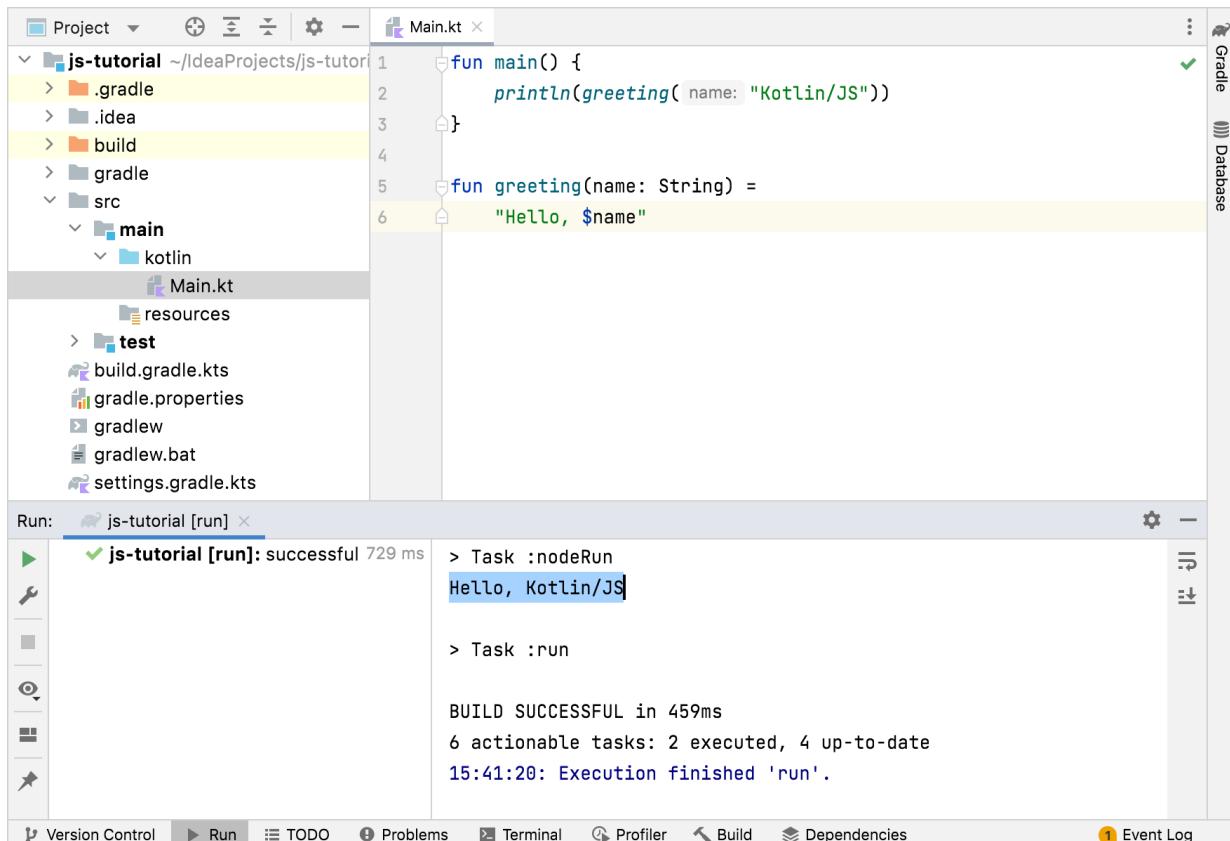
```
./gradlew run
```

If you're using IntelliJ IDEA, you can find the `run` action in the Gradle tool window:



### 1.5.30 的新特性

On first start, the `kotlin.js` 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/main/resources/index.html :
```

```
<!doctype html>
<html lang="en">
<head>
 <meta charset="UTF-8">
 <title>Hello, Kotlin/JS!</title>
</head>
<body>

</body>
<script src="jsTutorial.js"></script>
</html>
```

### 1.5.30 的新特性

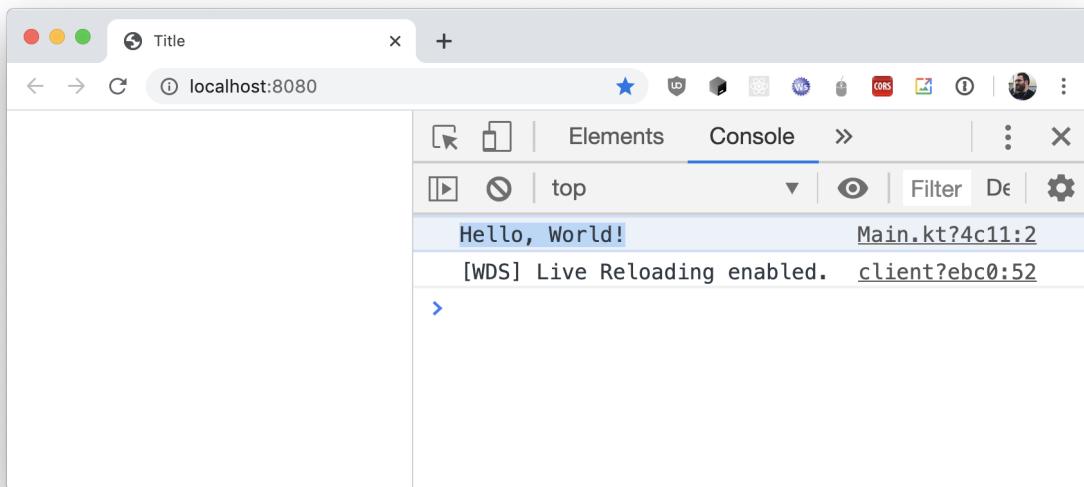
By default, the name of your project's generated artifact (which is created through webpack) that needs to be referenced is your project name (in this case, `jsTutorial`). If you've named your project `followAlong`, make sure to embed `followAlong.js` instead of `jsTutorial.js`.

After making these adjustments, start the integrated development server. You can do this from the command line via the Gradle wrapper:

```
./gradlew run
```

When working from IntelliJ IDEA, you can find the `run` 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:



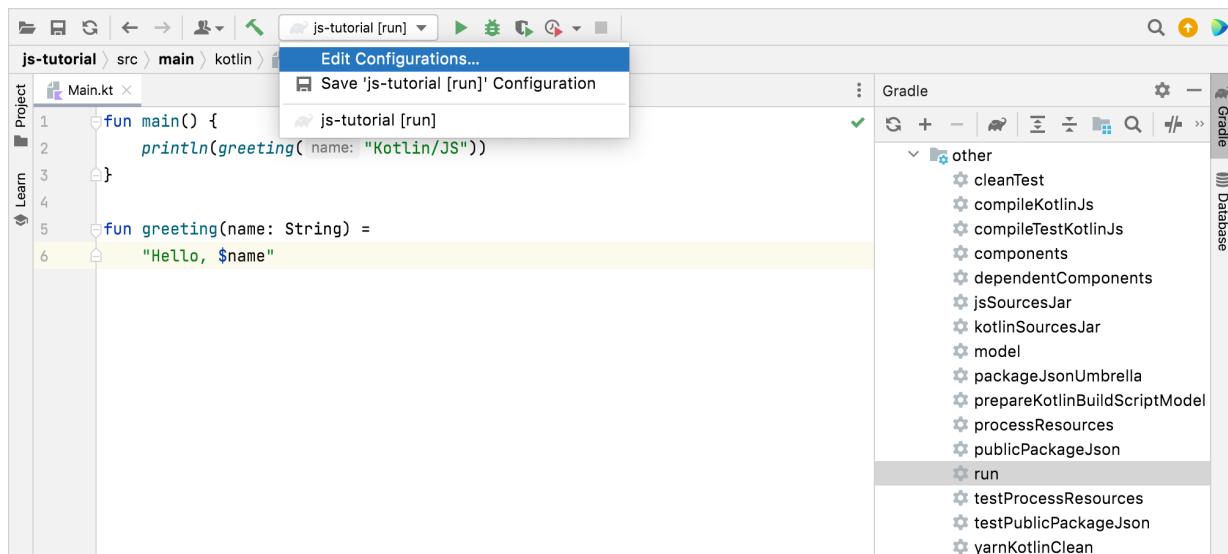
With this setup, you can recompile your project after each code change to see your changes. Kotlin/JS also supports a more convenient way of automatically rebuilding the application while you are developing it. To find out how to set up this *continuous mode*, check out the [corresponding tutorial](#).

## 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:

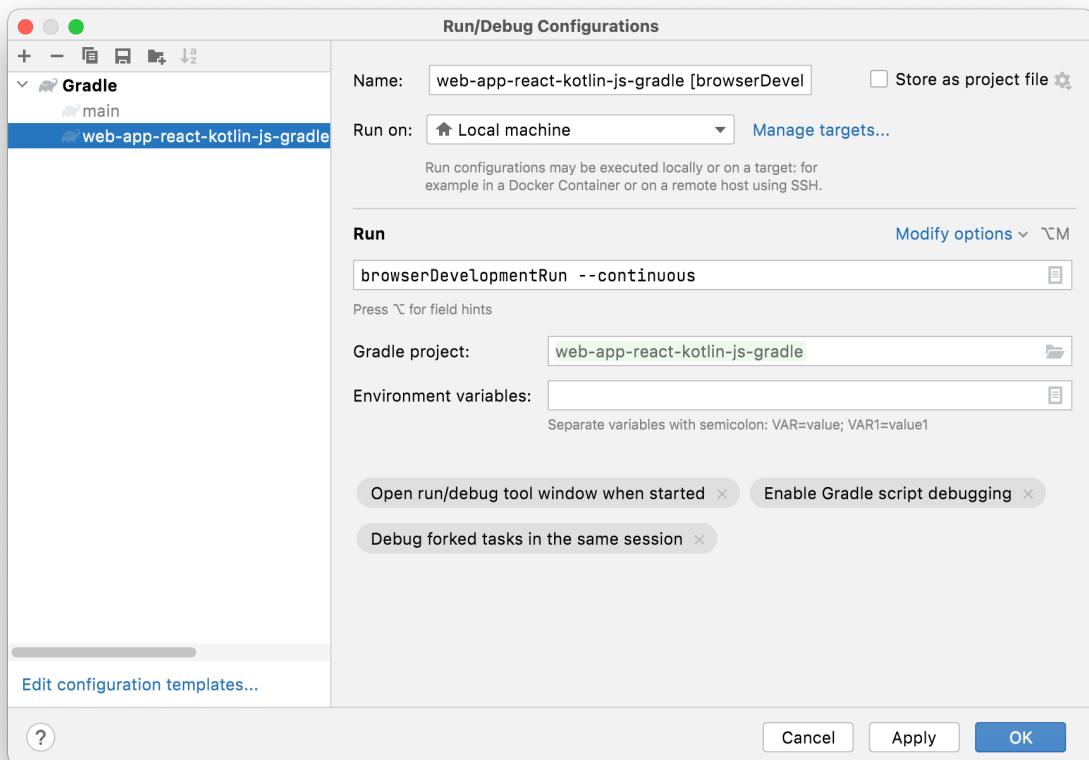
```
./gradlew run --continuous
```

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:

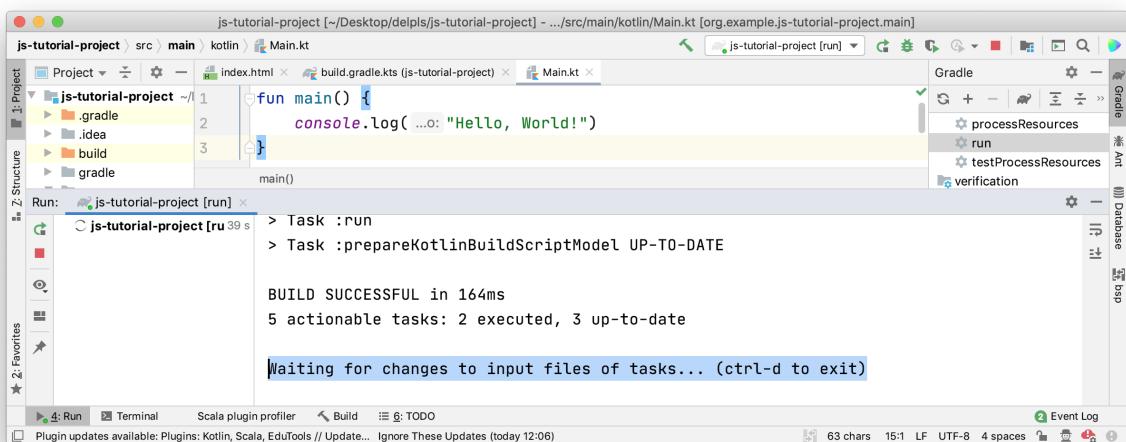


Enabling continuous mode via the **Run/Debug Configurations** dialog is as easy as adding the `--continuous` flag to the arguments for the run configuration:

## 1.5.30 的新特性



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/JS 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/JS Gradle plugin automatically generates source maps for the project builds, making them available without any additional configuration.

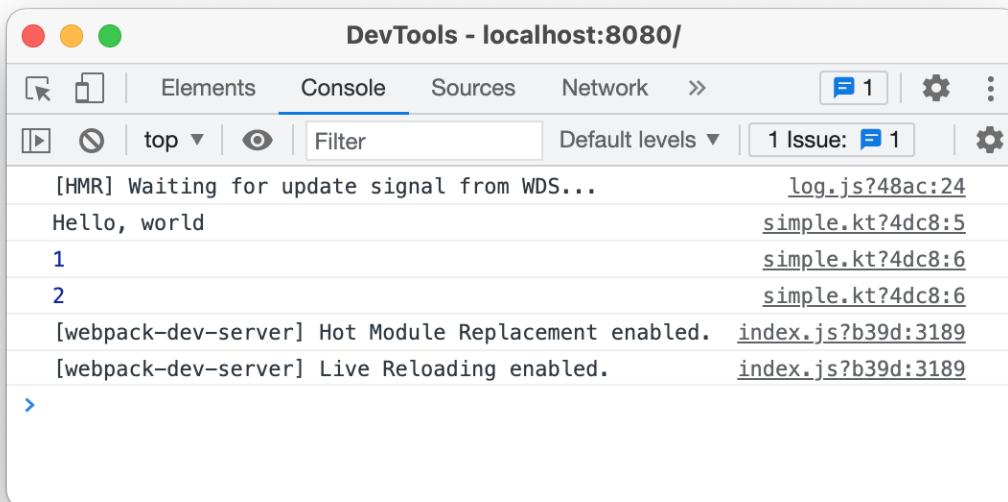
## Debug in browser

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:

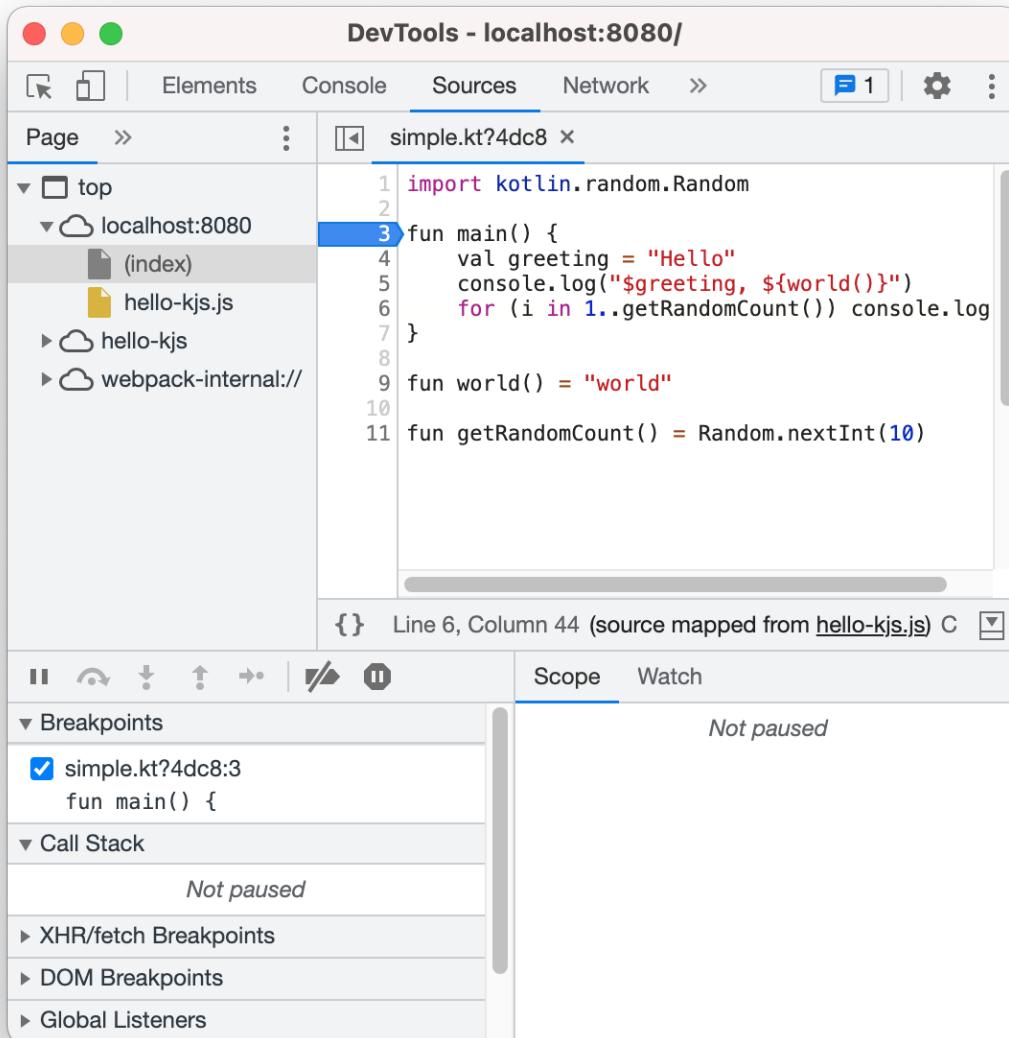
1. Run the project by calling one of the available *run* Gradle tasks, for example, `browserDevelopmentRun` or `jsBrowserDevelopmentRun` in a multiplatform project. Learn more about [running Kotlin/JS](#).
2. Navigate to the page in the browser and launch its developer tools (for example, by right-clicking 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:

## 1.5.30 的新特性



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):

### 1.5.30 的新特性



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.

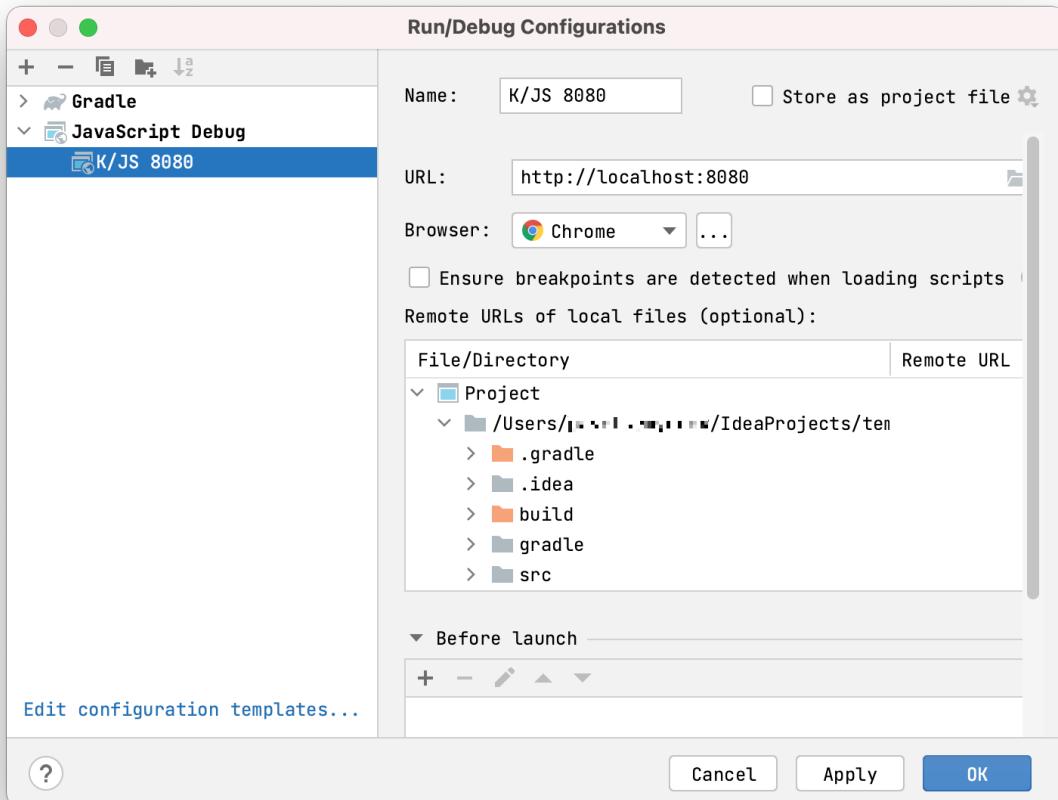
## Debug in the IDE

[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:

### 1.5.30 的新特性

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!

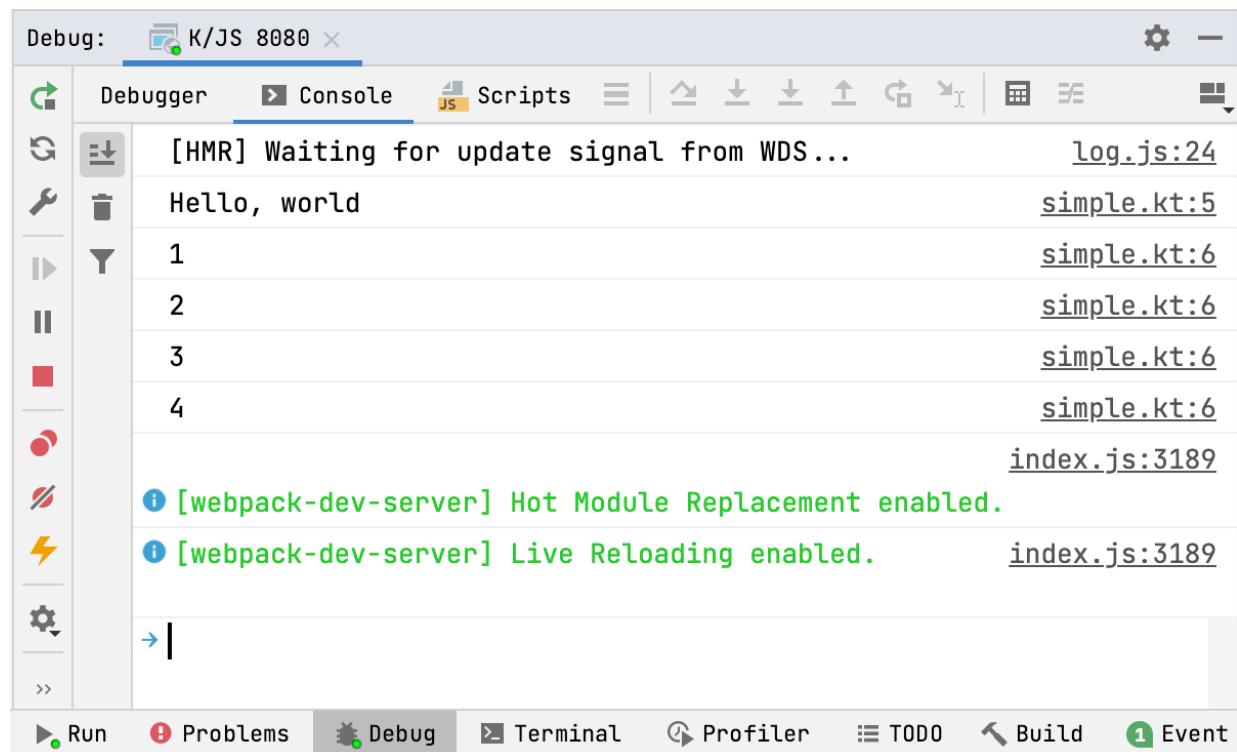
1. Run the project by calling one of the available `run` Gradle tasks, for example, `browserDevelopmentRun` or `jsBrowserDevelopmentRun` in a multiplatform project.  
Learn more about [running Kotlin/JS](#).
2. Start the debugging session by running the JavaScript debug configuration you've created previously:

### 1.5.30 的新特性



```
hello-kjs – simple.kt [hello-kjs.main]
getRandomCount() K/JS 8080
simple.kt index.html favicon.ico
1 import kotlin.random.Random
2
3 fun main() {
4 val greeting = "Hello"
5 console.log("$greeting, ${world()}")
6 for (i in 1..getRandomCount()) console.log(i)
7 }
8
9 fun world() = "world"
10
11 fun getRandomCount() = Random.nextInt(until: 10)
```

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:



```
Debug: K/JS 8080
Debugger Console Scripts
[HMR] Waiting for update signal from WDS... log.js:24
Hello, world simple.kt:5
1 simple.kt:6
2 simple.kt:6
3 simple.kt:6
4 simple.kt:6
index.js:3189
[webpack-dev-server] Hot Module Replacement enabled.
[webpack-dev-server] Live Reloading enabled. index.js:3189
→ |
```

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](#).

### 1.5.30 的新特性

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/JS Gradle plugin lets you run tests through a variety of test runners that can be specified via the Gradle configuration. In order to make test annotations and functionality available for the JavaScript target, add the correct platform artifact for

`kotlin.test` in `build.gradle.kts` :

```
dependencies {
 // ...
 testImplementation(kotlin("test-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:

```
target {
 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/test/kotlin/AppTest.kt` and fill it with this content:

## 1.5.30 的新特性

```
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 `browserTest` task via IntelliJ IDEA, or use the gutter icons to execute all or individual tests:

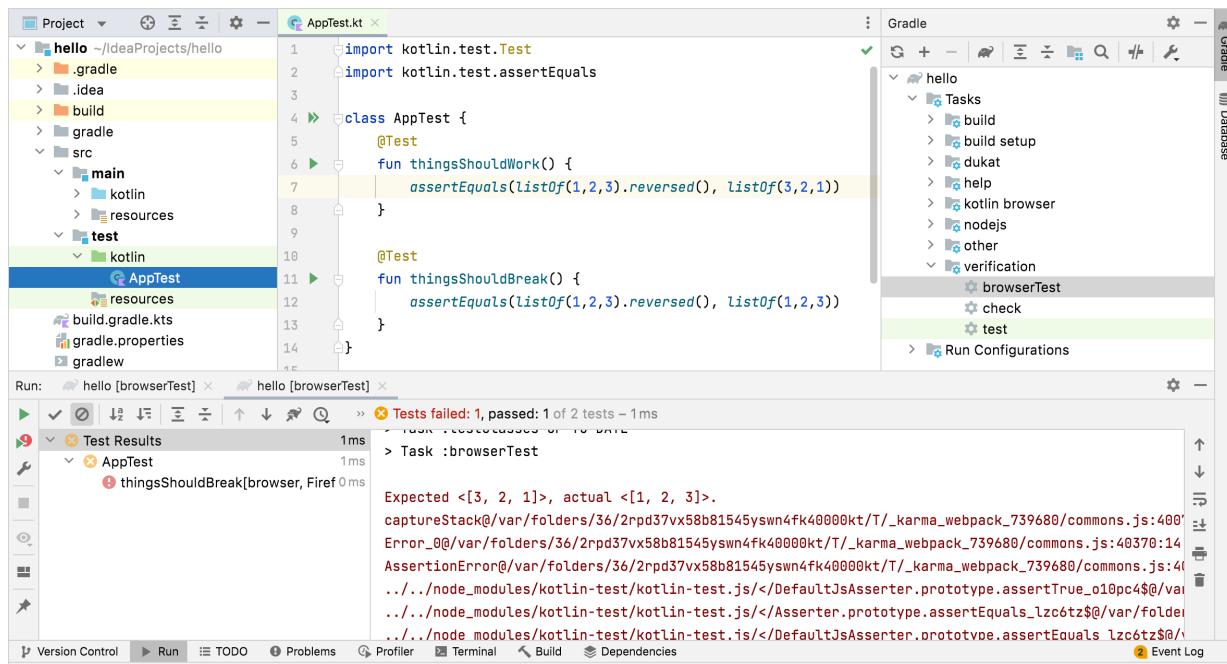


Alternatively, if you want to run the tests via the command line, use the Gradle wrapper:

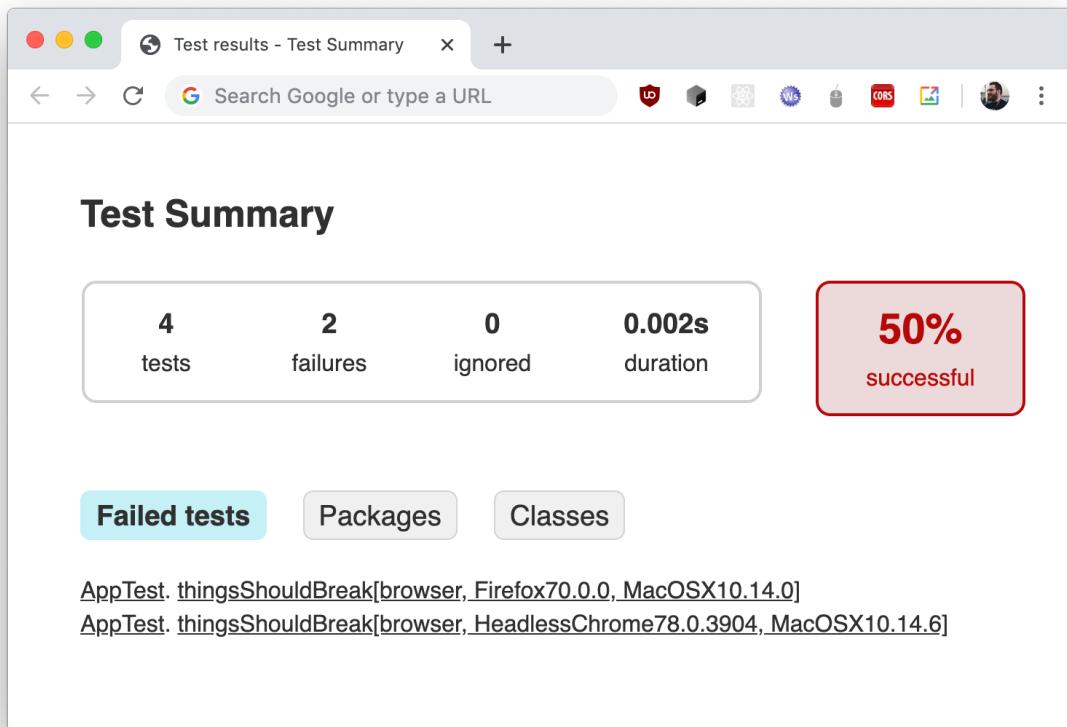
```
./gradlew browserTest
```

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.

## 1.5.30 的新特性



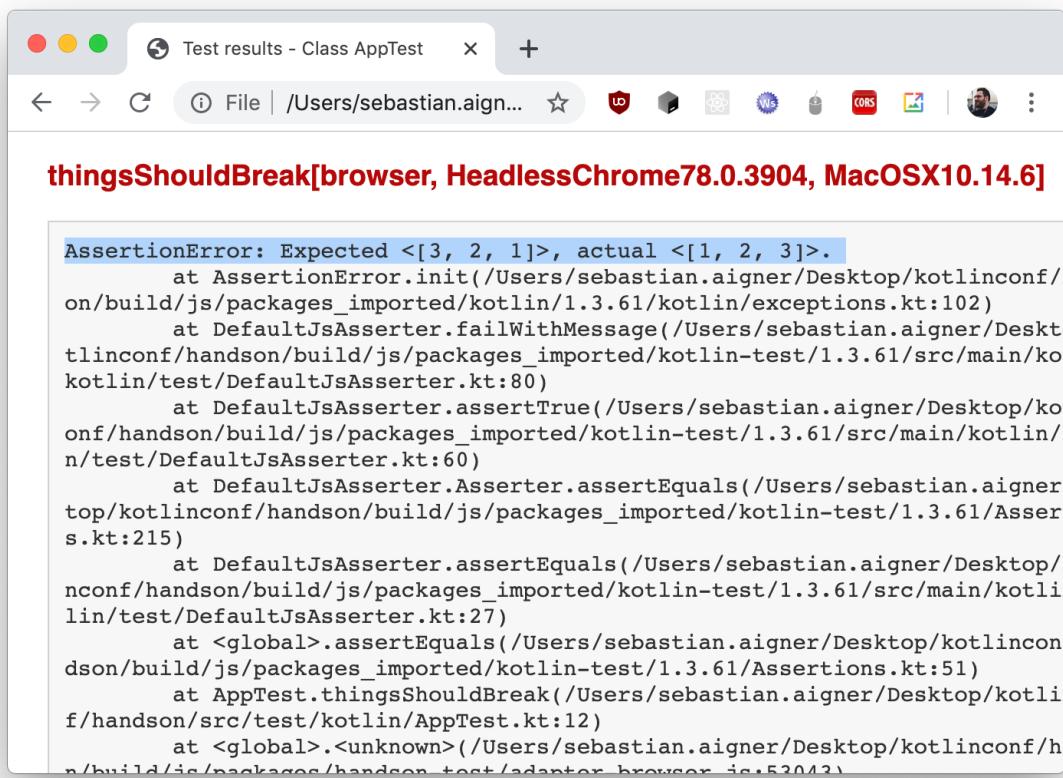
After each test run, regardless of how you executed the test, you can find a properly formatted test report from Gradle in `build/reports/tests/browserTest/index.html`. Open this file in a browser to see another overview of the test results:



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

## 1.5.30 的新特性

hyperlinks:



The screenshot shows a browser window with the title "Test results - Class AppTest". The URL in the address bar is "/Users/sebastian.aigner...". The main content area displays a red error message: "thingsShouldBreak[browser, HeadlessChrome78.0.3904, MacOSX10.14.6]". Below the message is a detailed stack trace of the Assertion Error:

```
AssertionError: Expected <[3, 2, 1]>, actual <[1, 2, 3]>.
 at AssertionError.init(/Users/sebastian.aigner/Desktop/kotlinconf/handson/build/js/packages_imported/kotlin/1.3.61/kotlin/exceptions.kt:102)
 at DefaultJsAssert.failWithMessage(/Users/sebastian.aigner/Desktop/kotlinconf/handson/build/js/packages_imported/kotlin-test/1.3.61/src/main/kotlin/test/DefaultJsAssert.kt:80)
 at DefaultJsAssert.assertTrue(/Users/sebastian.aigner/Desktop/kotlinconf/handson/build/js/packages_imported/kotlin-test/1.3.61/src/main/kotlin/test/DefaultJsAssert.kt:60)
 at DefaultJsAssert.assertEquals(/Users/sebastian.aigner/Desktop/kotlinconf/handson/build/js/packages_imported/kotlin-test/1.3.61/Asserts.kt:215)
 at DefaultJsAssert.assertEquals(/Users/sebastian.aigner/Desktop/kotlinconf/handson/build/js/packages_imported/kotlin-test/1.3.61/src/main/kotlin/test/DefaultJsAssert.kt:27)
 at <global>.assertEquals(/Users/sebastian.aigner/Desktop/kotlinconf/handson/build/js/packages_imported/kotlin-test/1.3.61/Assertions.kt:51)
 at AppTest.thingsShouldBreak(/Users/sebastian.aigner/Desktop/kotlinconf/handson/src/test/kotlin/AppTest.kt:12)
 at <global>.<unknown>(/Users/sebastian.aigner/Desktop/kotlinconf/handson/build/js/packages_imported/handson-test/adaptor_browser_js-53043)
```

## Kotlin/JS 无用代码消除

Kotlin/JS 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/JS 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。

## 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。



### 1.5.30 的新特性

```
kotlin {
 js {
 browser {
 dceTask {
 keep("myKotlinJSMODULE.org.example.getName", "myKotlinJSMODULE.org.
 }
 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

The Kotlin/JS IR compiler is in [Beta](#). It is almost stable, but migration steps may be required in the future. We'll do our best to minimize any changes you have to make.



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/JS 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` 使用默认编译器后端。
- `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](#)。

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。

## 忽略编译错误

### 1.5.30 的新特性

*Ignore compilation errors mode* 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 编译器提供了默认后端中不可用的新编译模式——忽略编译错误。在这种模式下，即使其代码包含错误，也可以试运行应用程序。例如，当执行复杂的重构或在系统的某个部分上进行工作时，而该部分与另一部分中的编译错误完全无关。

使用这种新的编译器模式，编译器会忽略所有有毛病的代码。因此，可以试运行应用程序并尝试没毛病的部分代码。如果试运行到在编译过程中出了毛病的代码，那么会得到运行时异常。

在两个容忍策略之间选择，以忽略代码中的编译错误：

- `SEMANTIC`：编译器将接受语法上正确但语义上没有意义的代码。例如，为字符串变量赋值一个数字（类型不匹配）。
- `SYNTAX`：编译器将接受任何代码，即使其中包含语法错误。无论编写什么内容，编译器仍尝试生成可运行的可执行文件。

作为实验特性，忽略编译错误需要选择加入。要启用此模式，请添加 `-Xerror-tolerance-policy={SEMANTIC|SYNTAX}` 编译器选项：

```
kotlin {
 js(IR) {
 compilations.all {
 compileKotlinTask.kotlinOptions.freeCompilerArgs += listOf("-Xerror-tole
 }
 }
}
```

## 顶层属性的延迟初始化

顶层属性的延迟初始化是 [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](#).



For better application startup performance, the Kotlin/JS IR compiler offers an option to initialize 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;

### 1.5.30 的新特性

other properties receive their values later when the code that uses them actually runs.

As an experimental feature, lazy initialization of top-level properties requires an opt-in. To use the lazy initialization of top-level properties, add the `-Xir-property-lazy-initialization` option when compiling the code with the JS IR compiler:

#### 【Kotlin】

```
tasks.withType<Kotlin2JsCompile> {
 kotlinOptions {
 freeCompilerArgs += "-Xir-property-lazy-initialization"
 }
}
```

#### 【Groovy】

```
tasks.withType(Kotlin2JsCompile) {
 kotlinOptions {
 freeCompilerArgs += "-Xir-property-lazy-initialization"
 }
}
```

## 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。

在产生可执行文件（`binaries.executable()`）的项目中，标有 `@JsExport` 的顶级声明将生成一个 `.d.ts` 文件，其中包含导出的 Kotlin 声明的 TypeScript 定义。可以在 `build/js/packages/<package_name>/kotlin` 中找到这些声明以及相应的未打包 Web 的 JavaScript 代码。

## IR 编译器的当前限制

### 1.5.30 的新特性

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**。Kotlin/JS 的两个后端之间缺乏这种兼容性，这意味着使用新的 IR 编译器后端创建的库无法在默认后端使用，反之亦然。

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` 块内特定于项目的选项之一：

### 1.5.30 的新特性

```
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.

# 将 Kotlin/JS 项目迁移到 IR 编译器

The Kotlin/JS IR compiler is in [Beta](#). It is 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 are replacing the current Kotlin/JS compiler with [the IR-based compiler](#) in order to unify Kotlin's behavior on all platforms and to make it possible to implement new JS-specific optimizations, among other reasons. You can learn more about the internal differences between the two compilers in the blog post [Migrating our Kotlin/JS app to the new IR compiler](#) by Sebastian Aigner.

Due to the significant differences between the compilers, switching your Kotlin/JS project from the old backend to the new one may require adjusting your code. On this page, we've compiled a list of known migration issues along with suggested solutions.

Install the [Kotlin/JS Inspection pack](#) plugin to get valuable tips on how to fix some of the issues that occur during migration.



Note that this guide may change over time as we fix issues and find new ones. Please help us keep it complete – report any issues you encounter when switching to the IR compiler by submitting them to our issue tracker [YouTrack](#) or filling out [this form](#).

## Convert JS- and React-related classes and interfaces to external interfaces

**Issue:** Using Kotlin interfaces and classes (including data classes) that derive from pure JS classes, such as React's `State` and `Props`, can cause a `ClassCastException`. Such exceptions appear because the compiler attempts to work with instances of these classes as if they were Kotlin objects, when they actually come from JS.

**Solution:** convert all classes and interfaces that derive from pure JS classes to [external interfaces](#):

### 1.5.30 的新特性

```
// Replace this
interface AppState : State { }
interface AppProps : Props { }
data class CustomComponentState(var name: String) : State
```

```
// With this
external interface AppState : State { }
external interface AppProps : Props { }
external interface CustomComponentState : State {
 var name: String
}
```

In IntelliJ IDEA, you can use these [structural search and replace](#) templates to automatically mark interfaces as `external`:

- [Template for `State`](#)
- [Template for `Props`](#)

## Convert properties of external interfaces to `var`

**Issue:** properties of external interfaces in Kotlin/JS code can't be read-only (`val`) properties because their values can be assigned only after the object is created with `js()` or `jsObject()` (a helper function from [kotlin-wrappers](#)):

```
val myState = js("{}") as CustomComponentState
myState.name = "name"
```

**Solution:** convert all properties of external interfaces to `var`:

```
// Replace this
external interface CustomComponentState : State {
 val name: String
}
```

```
// With this
external interface CustomComponentState : State {
 var name: String
}
```

## Make boolean properties nullable in external interfaces

**Issue:** JavaScript treats the `null` or undefined value of a boolean variable as `false`. So, boolean properties can be used in expressions without being defined. This is okay in JavaScript, but not in Kotlin.

```
external interface ComponentProps: Props {
 var isInitialized: Boolean
 var visible: Boolean
}
```

```
val props = js("{}") as ComponentProps
props.isInitialized = true
// visible is not initialized - OK in JS - means it's false
```

If you try to use such a property in a function overridden in Kotlin (for example, a React `button`), you'll get a `ClassCastException`:

```
button {
 attrs {
 autoFocus = props.visible // ClassCastException here
 }
}
```

**Solution:** make all `Boolean` properties of external interfaces nullable (`Boolean?`):

```
// Replace this
external interface ComponentProps: Props {
 var visible: Boolean
}
```

```
// With this
external interface ComponentProps: Props {
 var visible: Boolean?
}
```

## Convert functions with receivers in external interfaces to regular functions

**Issue:** external declarations can't contain functions with receivers, such as extension functions or properties with corresponding functional types.

**Solution:** convert such functions and properties to regular functions by adding the receiver object as an argument:

```
// Replace this
external interface ButtonProps : Props {
 var inside: StyledDOMBuilder<BUTTON>.() -> Unit
}
```

```
external interface ButtonProps : Props {
 var inside: (StyledDOMBuilder<BUTTON>) -> Unit
}
```

## Create plain JS objects for interoperability

**Issue:** properties of a Kotlin object that implements an external interface are not *enumerable*. This means that they are not visible for operations that iterate over the object's properties, for example:

- `for (var name in obj)`
- `console.log(obj)`
- `JSON.stringify(obj)`

Although they are still accessible by the name: `obj.myProperty`

### 1.5.30 的新特性

```
external interface AppProps { var name: String }
data class AppPropsImpl(override var name: String) : AppProps
fun main() {
 val jsApp = js("{name: 'App1'}") as AppProps // plain JS object
 println("Kotlin sees: ${jsApp.name}") // "App1"
 println("JSON.stringify sees:" + JSON.stringify(jsApp)) // {"name": "App1"} - OK

 val ktApp = AppPropsImpl("App2") // Kotlin object
 println("Kotlin sees: ${ktApp.name}") // "App2"
 // JSON sees only the backing field, not the property
 println("JSON.stringify sees:" + JSON.stringify(ktApp)) // {"_name_3": "App2"}
}
```

**Solution 1:** create plain JavaScript objects with `js()` or `jsObject()` (a helper function from `kotlin-wrappers`):

```
external interface AppProps { var name: String }
data class AppPropsImpl(override var name: String) : AppProps
```

```
// Replace this
val ktApp = AppPropsImpl("App1") // Kotlin object
```

```
// With this
val jsApp = js("{name: 'App1'}") as AppProps // or jsObject {}
```

**Solution 2:** create objects with `kotlin.js.json()`:

```
// or with this
val jsonApp = kotlin.js.json(Pair("name", "App1")) as AppProps
```

## Replace `toString()` calls on function references with `.name`

**Issue:** in the IR backend, calling `toString()` on function references doesn't produce unique values.

**Solution:** use the `name` property instead of `toString()`.

## Explicitly specify binaries.executable() in the build script

**Issue:** the compiler doesn't produce executable `.js` files.

This may happen because the default compiler produces JavaScript executables by default while the IR compiler needs an explicit instruction to do this. Learn more in the [Kotlin/JS project setup instruction](#).

**Solution:** add the line `binaries.executable()` to the project's `build.gradle(.kts)`.

```
kotlin {
 js(IR) {
 browser {
 }
 binaries.executable()
 }
}
```

## 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
- 用 Dukat 生成外部声明

# 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"
```

### 1.5.30 的新特性

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。

还提供了一种实验工具，可为 npm 依赖自动创建 Kotlin 外部声明，该声明提供称为 [Dukat](#) 的类型定义（TypeScript / `d.ts`）。

## 内联 JavaScript

你可以使用 `js()` 函数将一些 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"
```

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：

### 1.5.30 的新特性

```
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

### 1.5.30 的新特性

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参数的函数。
- 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。

### 1.5.30 的新特性

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。

### 1.5.30 的新特性

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();
}
```

## 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：

### 1.5.30 的新特性

```
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)))
```

### 1.5.30 的新特性

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"
```

### 1.5.30 的新特性

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。

### 1.5.30 的新特性

`@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

`@JsExport` 注解当前标记为实验性的。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。

`@JsExport` 注解在当前的默认编译器后端与新的 IR 编译器后端中可用。如果以 IR 编译器后端为目标，则 you **must** use the `@JsExport` annotation to make your functions visible from Kotlin in the first place.

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，并且还允许导出不是特定于平台的 Kotlin 声明。

## 在 JavaScript 中的 Kotlin 类型

- 除了 `kotlin.Long` 的 Kotlin 数字类型映射到 JavaScript `Number`。
- `kotlin.Char` 映射到 JavaScript `Number` 来表示字符代码。
- Kotlin 在运行时无法区分数字类型（`kotlin.Long` 除外），因此以下代码能够工作：

```
fun f() {
 val x: Int = 23
 val y: Any = x
 println(y as Float)
}
```

### 1.5.30 的新特性

- Kotlin 保留了 `kotlin.Int`、`kotlin.Byte`、`kotlin.Short`、`kotlin.Char` 和 `kotlin.Long` 的溢出语义。
- `kotlin.Long` 没有映射到任何 JavaScript 对象，因为 JavaScript 中没有 64 位整数，它是由一个 Kotlin 类模拟的。
- `kotlin.String` 映射到 JavaScript `String`。
- `kotlin.Any` 映射到 JavaScript `Object` (`new Object()`、`{}` 等)。
- `kotlin.Array` 映射到 JavaScript `Array`。
- Kotlin 集合 (`List`、`Set`、`Map` 等) 没有映射到任何特定的 JavaScript 类型。
- `kotlin.Throwable` 映射到 JavaScript `Error`。
- 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。

## 原生数组

原生数组转换到 JavaScript 时采用 `TypedArray`：

- `kotlin.ByteArray`、`-.ShortArray`、`-.IntArray`、`-.FloatArray` 以及 `-.DoubleArray` 会相应地映射为 JavaScript 中的 `Int8Array`、`Int16Array`、`Int32Array`、`Float32Array` 以及 `Float64Array`。
- `kotlin.BooleanArray` 会映射为 JavaScript 中具有 `$type$ == "BooleanArray"` 属性的 `Int8Array`。
- `kotlin.CharArray` 会映射为 JavaScript 中具有 `$type$ == "CharArray"` 属性的 `UIInt16Array`。
- `kotlin.LongArray` 会映射为 JavaScript 中具有 `$type$ == "LongArray"` 属性的 `kotlin.Long` 的数组。

# 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

如果以浏览器为目标并且要使用与 UMD 不同的模块系统，则可以在 `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。虽然在大多数情况下，可能希望使用 `commonjs2`，该模块将 `module.exports` 语法添加到所生成的库中，但也可以选择“纯”`commonjs` 选项，该选项完全实现 CommonJS 规范。如需了解有关 `commonjs` 与 `commonjs2` 之间的区别的更多信息，请[在此处](#)查看。

## JavaScript 库与 Node.js 文件

### 1.5.30 的新特性

如果正在创建一个将从 JavaScript 或 Node.js 文件中使用的库，并且希望使用不同的模块系统，那么说明会略有不同。

## 选择目标模块系统

要选择模块种类，请在 Gradle 构建脚本中设置 `moduleKind` 编译器选项。

### 【Kotlin】

```
tasks.named<KotlinJsCompile>("compileKotlinJs").configure {
 kotlinOptions.moduleKind = "commonjs"
}
```

### 【Groovy】

```
compileKotlinJs.kotlinOptions.moduleKind = "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）。`kotlinOptions.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); }
```

### 1.5.30 的新特性

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。

### 1.5.30 的新特性

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：

### 1.5.30 的新特性

```
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。虽然在大多数 Kotlin/JS 用例中，建议对 `kotlin-stdlib-js` 使用 Gradle 依赖项，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()` function.

## 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.

## Example

Here is an example of the reflection usage in Kotlin/JS.

### 1.5.30 的新特性

```
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 "Re
}
```

# 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 {
 // ...
 jcenter()
}

dependencies {
 implementation(kotlin("stdlib-js"))
 implementation("org.jetbrains.kotlinx:kotlinx-html-js:0.7.1")
 // ...
}
```

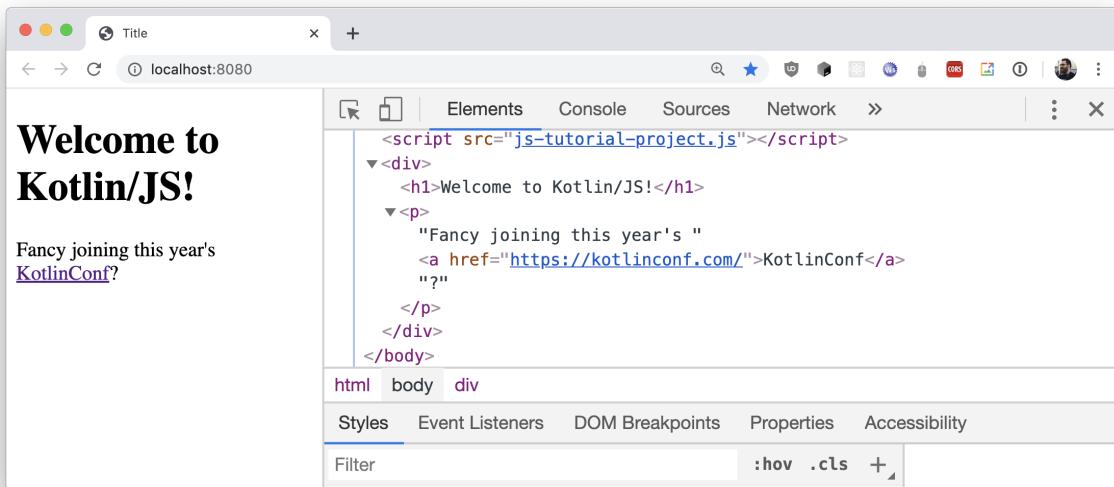
Once the dependency is included, you can access the different interfaces provided to generate the DOM. To render a headline, some text, and a link, the following snippet would be sufficient, for example:

```
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"
 }
 +"?"
 }
 }
}
```

### 1.5.30 的新特性

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.

# 用 Dukat 生成外部声明

Dukat 仍处于[实验状态](#)。如果遇到任何问题，请在 Dukat 的[问题跟踪器](#)中报告这些问题。



[Dukat](#) 是当前正在开发的工具，它允许将 TypeScript 声明文件（`.d.ts`）自动转换为 Kotlin 外部声明。这旨在使在 Kotlin 中以类型安全的方式使用来自 JavaScript 生态系统的库更加舒适，从而减少了为 JS 库手动编写外部声明与包装的需求。

Kotlin/JS Gradle 插件提供了与 Dukat 的集成。启用后，将自动为提供 TypeScript 定义的 npm 依赖项生成类型安全的 Kotlin 外部声明。可以通过两种不同的方式选择是否以及何时生成 Dukat 声明：在构建时以及通过 Gradle 任务手动进行。

## 在构建时生成外部声明

`npm` 依赖函数在包名称与版本之后采用第三个参数：`generateExternals`。这使得可以控制 Dukat 是否应为特定依赖项生成声明：

### 【Kotlin】

```
dependencies {
 implementation(npm("decamelize", "4.0.0", generateExternals = true))
}
```

### 【Groovy】

```
dependencies {
 implementation(npm('decamelize', '4.0.0', true))
}
```

如果希望使用的依赖项存储库不提供 TypeScript 定义，那么还可以使用通过[DefinitelyTyped](#) 存储库提供的类型。在这种情况下，请确保同时为 `your-package` 与 `@types/your-package` 添加 `npm` 依赖项（`generateExternals = true`）。

可以在 `gradle.properties` 文件中使用选项 `kotlin.js.generate.externals` 来同时为所有 npm 依赖项设置生成器的行为。像往常一样，单个显式设置优先于该常规选项。

## 通过 Gradle 任务手动生成外部声明

如果想完全控制 Dukat 生成的声明、想要应用手动调整、或者遇到自动生成的外部组件的麻烦，还可以通过触发 Gradle 任务 `generateExternals` (`jsGenerateExternals` with the `multiplatform` plugin) 手动为所有 npm 依赖项创建声明。这将在项目根目录中名为 `externals` 的目录中生成声明。在这里，可以查看生成的代码，并将想要使用的任何部分复制到源代码目录中。

建议仅在源文件夹中手动提供外部声明，或在构建时为任何单个依赖项启用外部声明的生成。两种办法都可以解决问题。

# Kotlin/JS 动手实践实验室

Hands-on labs are long-form tutorials that help you get to know a technology by guiding you through a self-contained project related to a specific topic.

They include sample projects, which can serve as jumping-off points for your own projects, and contain useful snippets and patterns.

For Kotlin/JS, the following hands-on labs are currently available:

- [Building Web Applications with React and Kotlin/JS](#) guides you through the process of building a simple web application using the React framework, shows how a typesafe Kotlin DSL for HTML makes it convenient to build reactive DOM elements, and illustrates how to use third-party React components, and how to obtain information from APIs, while writing the whole application logic in pure Kotlin/JS.
- [Building a Full Stack Web App with Kotlin Multiplatform](#) teaches the concepts behind building an application that targets Kotlin/JVM and Kotlin/JS by building a client-server application that makes use of common code, serialization, and other multiplatform paradigms. It also provides a brief introduction into working with Ktor both as a server- and client-side framework.

We are continuously working on expanding the set of hands-on labs to make it as easy as possible for you to learn more about Kotlin/JS and adjacent technologies.

# 原生

- [Kotlin/Native 入门——在 IntelliJ IDEA 中](#)
- [Kotlin/Native 入门——使用 Gradle](#)
- [Kotlin/Native 入门——使用命令行编译器](#)
- [与 C 语言互操作
  - \[与 C 语言互操作性\]\(#\)
  - \[映射来自 C 语言的原始数据类型——教程\]\(#\)
  - \[映射来自 C 语言的结构与联合类型——教程\]\(#\)
  - \[映射来自 C 语言的函数指针——教程\]\(#\)
  - \[映射来自 C 语言的字符串——教程\]\(#\)
  - \[创建使用 C 语言互操作与 libcurl 的应用——教程\]\(#\)](#)
- [与 Objective-C 互操作性
  - \[与 Swift/Objective-C 互操作性\]\(#\)
  - \[Kotlin/Native 开发 Apple framework——教程\]\(#\)](#)
- [CocoaPods 集成
  - \[CocoaPods 概述\]\(#\)
  - \[添加对 Pod 库的依赖\]\(#\)
  - \[使用 Kotlin Gradle 项目作为 CocoaPods 依赖项\]\(#\)](#)
- [Kotlin/Native 库](#)
- [平台库](#)
- [Kotlin/Native 开发动态库——教程](#)
- [不可变性与并发](#)
- [处理并发
  - \[并发概述\]\(#\)
  - \[并发可变性\]\(#\)
  - \[并发与协程\]\(#\)](#)
- [调试 Kotlin/Native](#)
- [符号化 iOS 崩溃报告](#)
- [改进 Kotlin/Native 编译时间的技巧](#)
- [Kotlin/Native FAQ](#)
- [Kotlin/Native 中的并发](#)

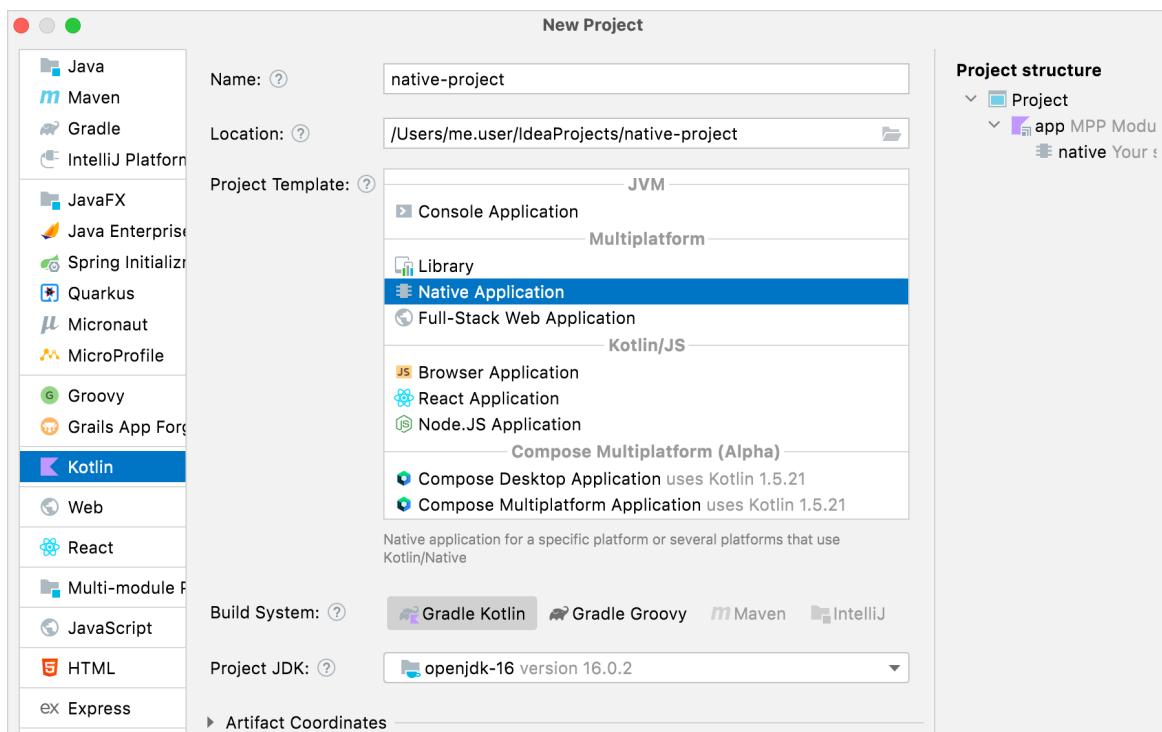
# Kotlin/Native 入门——在 IntelliJ IDEA 中

This tutorial demonstrates how to use IntelliJ IDEA for creating a Kotlin/Native application.

To get started, install the latest version of [IntelliJ IDEA](#). The tutorial is applicable to both IntelliJ IDEA Community Edition and the Ultimate Edition.

## Create a new Kotlin/Native project in IntelliJ IDEA

1. In IntelliJ IDEA, select **File | New | Project**.
2. In the panel on the left, select **Kotlin**.
3. Enter a project name, select **Native Application** as the project template, and click **Next**.



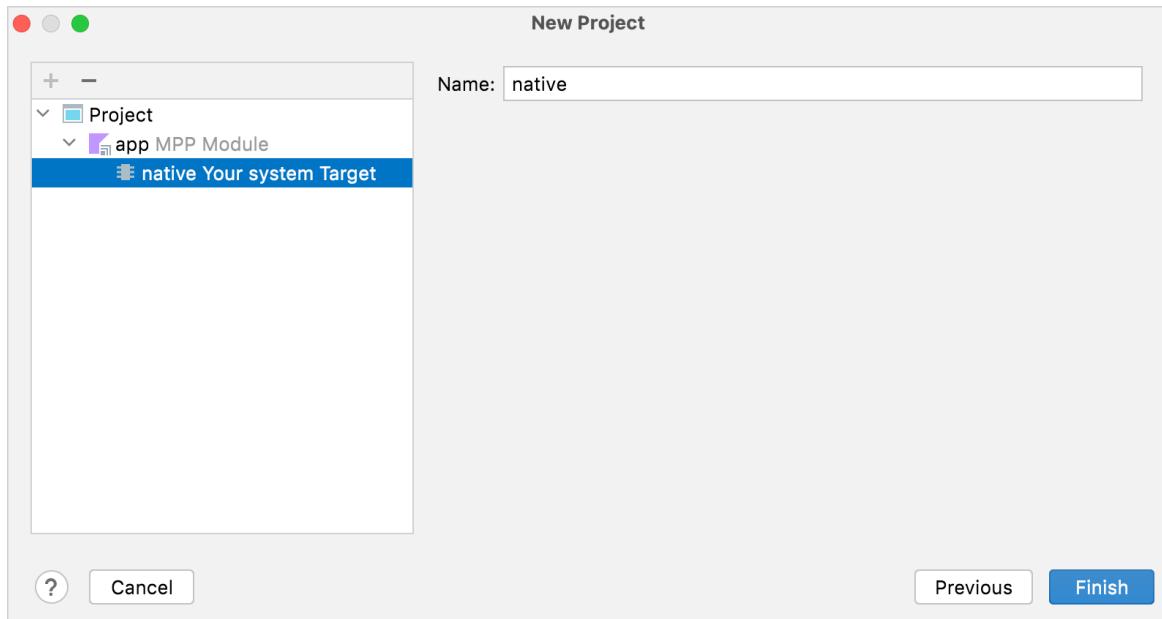
By default, your project will use Gradle with Kotlin DSL as the build system.

**Kotlin/Native doesn't support Maven and IntelliJ IDEA native builder.**



### 1.5.30 的新特性

4. Accept the default configuration on the next screen and click **Finish**. Your project will open.



By default, the wizard creates the necessary `main.kt` file with code that prints "Hello, Kotlin/Native!" to the standard output.

5. Open the `build.gradle.kts` file, the build script that contains the project settings. To create Kotlin/Native applications, you need the Kotlin Multiplatform Gradle plugin installed. Ensure that you use the latest version of the plugin:

```
plugins {
 kotlin("multiplatform") version "1.6.10"
}
```

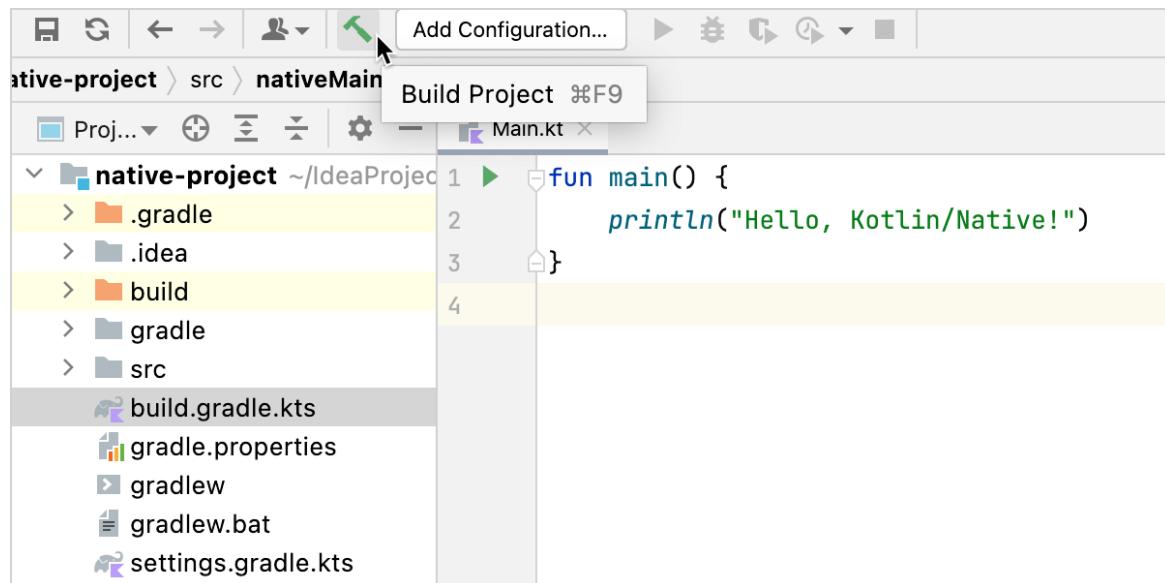
- Read more about these settings in the [Multiplatform Gradle DSL reference](#).
- Read more about the Gradle build system in the [documentation](#).



## Build and run the application

1. Click **Build Project** next to the run configuration at the top of the screen:

### 1.5.30 的新特性



2. On the **Terminal** tab, run the following command:

```
build/bin/native/debugExecutable/<your_app_name>.kexe
```

IntelliJ IDEA prints "Hello, Kotlin/Native!".

You can [configure IntelliJ IDEA](#) to build your project automatically:

1. Go to **Settings/Preferences | 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 the incremental build of the project.

## Update the application

### Count the letters in your name

1. Open the file `main.kt` in `src/nativeMain/kotlin`.

The `src` directory contains the Kotlin source files and resources. The file `main.kt` includes sample code that prints "Hello, Kotlin/Native!" using the `println()` function.

2. Add code to read the input. Use the `readln()` function to read the input value and assign it to the `name` variable:

### 1.5.30 的新特性

The `readln()` function is available since [Kotlin 1.6.0](#).

Ensure that you have installed the latest version of the [Kotlin plugin](#).



```
fun main() {
 // Read the input value.
 println("Hello, enter your name:")
 val name = readln()
}
```

#### 3. 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")
 }
}
```

#### 4. Save the changes and run the build command:

```
build/bin/native/debugExecutable/<your_app_name>.kexe
```

#### 5. Enter your name and enjoy the result:

```
Terminal: Local + ▾
~/IdeaProjects/native-project ➜ build/bin/native/debugExecutable/native-project.kexe
Hello, enter your name:
John Smith
Your name contains 9 letters
```

## Count the unique letters in your name

1. Open the file `main.kt` in `src/nativeMain/kotlin`.

### 1.5.30 的新特性

#### 2. 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()
```

#### 3. 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")
 }
}
```

#### 4. Save the changes and run the build command:

```
build/bin/native/debugExecutable/<your_app_name>.kexe
```

#### 5. Enter your name and enjoy the result:



```
Terminal: Local + ▾
~/IdeaProjects/native-project ➔ build/bin/native/debugExecutable/native-project.kexe
Hello, enter your name:
John Smith
Your name contains 9 letters
Your name contains 8 unique letters
```

# 下一步做什么？

### 1.5.30 的新特性

Once you have created your first application, you can go to Kotlin hands-on labs and complete long-form tutorials on Kotlin/Native.

For Kotlin/Native, the following hands-on labs are currently available:

- [Learn about the concurrency model in Kotlin/Native](#) shows you how to build a command-line application and work with states in a multi-threaded environment.
- [Create an app using C Interop and libcurl](#) explains how to create a native HTTP client and interoperate with C libraries.

# Kotlin/Native 入门——使用 Gradle

Gradle 是一个在 Java、Android 与其他生态系统项目中非常常用的构建系统。它是 Kotlin/Native 与 Multiplatform 的默认构建系统。

尽管包括 IntelliJ IDEA 在内的大多数 IDE 都可以生成相应的 Gradle 文件，但还是建议看一下如何手动创建此 Gradle 文件，以更好地了解事物的本质。如果想使用 IDE，请参阅[使用 IntelliJ IDEA 的 Hello Kotlin/Native](#)。

Gradle 支持两种语言的构建脚本：

- Groovy 脚本为 `build.gradle` 文件
- Kotlin 脚本为 `build.gradle.kts` 文件

Groovy 语言是 Gradle 最早支持的脚本语言，它利用了该语言的动态类型与运行时特性。也可以在 Gradle 脚本中使用 Kotlin。作为一种静态类型的语言，当涉及到编译与错误检测时，可以在 IDE 中更好地发挥作用。

两种都可以使用，示例将展示两种语言的语法。

## Create project files

First, create a project directory. Inside it, create `build.gradle` or `build.gradle.kts` Gradle build file with the following contents:

【Kotlin】

## 1.5.30 的新特性

```
plugins {
 kotlin("multiplatform") version "1.6.10"
}

repositories {
 mavenCentral()
}

kotlin {
 macosX64("native") { // on macOS
 // linuxX64("native") // on Linux
 // mingwX64("native") // on Windows
 binaries {
 executable()
 }
 }
}

tasks.withType<Wrapper> {
 gradleVersion = "6.7.1"
 distributionType = Wrapper.DistributionType.BIN
}
```

## 【Groovy】

```
plugins {
 id 'org.jetbrains.kotlin.multiplatform' version '1.6.10'
}

repositories {
 mavenCentral()
}

kotlin {
 macosX64('native') { // on macOS
 // linuxX64('native') // on Linux
 // mingwX64('native') // on Windows
 binaries {
 executable()
 }
 }
}

wrapper {
 gradleVersion = '6.7.1'
 distributionType = 'BIN'
}
```

## 1.5.30 的新特性

Next, create an empty `settings.gradle` or `settings.gradle.kts` file in the project folder.

取决于目标平台，不同的[函数](#)，例

如：`macosX64`、`mingwX64`、`linuxX64`、`iosX64`，用于创建 Kotlin 目标。函数名称是其编译代码的平台。这些函数可以选择将目标名称作为参数，在本例中为“native”。指定的 目标名称 用于在项目中生成源路径和任务名称。

按照惯例，所有源代码都位于 `src/<目标名称>[Main|Test]/kotlin` 文件夹中，其中 `main` 用于源代码，而 `test` 用于测试。`<目标名称>` 对应于构建文件中指定的目标平台（在本例中为 `native`）。

创建一个文件夹 `src/nativeMain/kotlin`，并在其中放置文件 `hello.kt`，其内容如下：

```
fun main() {
 println("Hello Kotlin/Native!")
}
```

## 构建项目

在项目根目录中，通过运行执行构建

```
gradle nativeBinaries
```

这将创建一个文件夹 `build/bin/native`，其中包含两个子文件夹 `debugExecutable` 与 `releaseExecutable` 以及相应的二进制文件。默认情况下，二进制文件的名称与项目文件夹的名称相同。

## 在 IDE 中打开项目

任何支持 Gradle 的 IDE 都应允许在 IDE 中打开项目。对于 [IntelliJ IDEA](#)，只需打开项目文件夹，会自动将其检测为 Kotlin/Native 项目。

## 下一步做什么？

Learn how to [write Gradle build scripts for real-life Kotlin/Native projects](#).

# Kotlin/Native 入门——使用命令行编译器

## 获取编译器

Kotlin/Native 编译器适用于 macOS、Linux 及 Windows。它是一个命令行工具，作为标准 Kotlin 发行版的一部分提供，可以从 [GitHub 发行版](#) 下载。它支持包括 iOS (arm32、arm64、simulator x86\_64)、Windows (mingw32 及 x86\_64) 在内的多个不同目标平台，Linux (x86\_64、arm64、MIPS)、macOS (x86\_64)、Raspberry PI、STM32、WASM。[关于目标的完整列表，请参见这里](#)。进行跨平台交叉编译，这意味着可以使用一个平台针对另一个平台进行编译，在这个 <https://github.com/JetBrains/kotlin/releases/tag/v1.6.10> 场景中，我们将针对跟本机相同的单平台。

尽管编译器的输出没有任何依赖项或虚拟机要求，但编译器本身需要 [Java 1.8 或更高版本的运行时](#)。

Install the compiler by unpacking its archive to a directory of your choice and adding the path to its `/bin` directory to the `PATH` environment variable.

## Write "Hello Kotlin/Native" program

该应用程序将在标准输出上打印 "Hello Kotlin/Native"。在选择的工作目录中，创建一个名为 `hello.kt` 的文件，并输入以下内容：

```
fun main() {
 println("Hello Kotlin/Native!")
}
```

## 在控制台编译代码

要编译这个应用程序，请[下载](#)编译器来执行以下命令：

```
kotlinc-native hello.kt -o hello
```

### 1.5.30 的新特性

`-o` 选项的值指定了输出文件的名称，所以这个调用应该生成一个 `hello.kexe` (Linux 及 macOS) 或 `hello.exe` (Windows) 二进制文件。关于可用编译器选项的完整列表，请参见[编译器选项参考](#)。

虽然从控制台编译看起来简单明了，但它对于包含数百个文件和库的大型项目来说，这种方法不太适用。对于现实项目，建议使用[构建系统](#)和[集成开发环境](#)。

## 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
- 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

Kotlin/Native follows the general tradition of Kotlin to provide excellent existing platform software interoperability. In the case of a native platform, the most important interoperability target is a C library. So Kotlin/Native comes with a `cinterop` tool, which can be used to quickly generate everything needed to interact with an external library.

The following workflow is expected when interacting with the native library:

1. Create a `.def` file describing what to include into bindings.
2. Use the `cinterop` tool to produce Kotlin bindings.
3. Run the Kotlin/Native compiler on an application to produce the final executable.

The interoperability tool analyses C headers and produces a "natural" mapping of the types, functions, and constants into the Kotlin world. The generated stubs can be imported into an IDE for the purpose of code completion and navigation.

Interoperability with Swift/Objective-C is provided too and covered in [Objective-C interop](#).

## Platform libraries

Note that in many cases there's no need to use custom interoperability library creation mechanisms described below, as for APIs available on the platform standardized bindings called [platform libraries](#) could be used. For example, POSIX on Linux/macOS platforms, Win32 on Windows platform, or Apple frameworks on macOS/iOS are available this way.

## Simple example

Install libgit2 and prepare stubs for the git library:

```
cd samples/gitchurn
../../dist/bin/cinterop -def src/nativeInterop/cinterop/libgit2.def \
-compiler-option -I/usr/local/include -o libgit2
```

### 1.5.30 的新特性

Compile the client:

```
../../../../dist/bin/kotlinc src/gitChurnMain/kotlin \
 -library libgit2 -o GitChurn
```

Run the client:

```
./GitChurn.kexe ../../
```

## Create bindings for a new library

To create bindings for a new library, start from creating a `.def` file. Structurally it's a simple property file, which looks like this:

```
headers = png.h
headerFilter = png.h
package = png
```

Then run the `cinterop` tool with something like this (note that for host libraries that are not included in the sysroot search paths, headers may be needed):

```
cinterop -def png.def -compiler-option -I/usr/local/include -o png
```

This command will produce a `png.klib` compiled library and `png-build/kotlin` directory containing Kotlin source code for the library.

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.

Note that the generated bindings are generally platform-specific, so if you are developing for multiple targets, the bindings need to be regenerated.

After the generation of bindings, they can be used by the IDE as a proxy view of the native library.

For a typical Unix library with a config script, the `compilerOpts` will likely contain the output of a config script with the `--cflags` flag (maybe without exact paths).

### 1.5.30 的新特性

The output of a config script with `--libs` will be passed as a `-linkedArgs` `kotlinc` flag value (quoted) when compiling.

## Select library headers

When library headers are imported to a C program with the `#include` directive, all of the headers included by these headers are also included in the program. So all header dependencies are included in generated stubs as well.

This behavior is correct but it can be very inconvenient for some libraries. So it is possible to specify in the `.def` file which of the included headers are to be imported. The separate declarations from other headers can also be imported in case of direct dependencies.

## Filter headers by globs

It is possible to filter headers by globs. The `headerFilter` property value from the `.def` file is treated as a space-separated list of globs. If the included header matches any of the globs, then the declarations from this header are included into the bindings.

The globs are applied to the header paths relative to the appropriate include path elements, e.g. `time.h` or `curl/curl.h`. So if the library is usually included with `#include <SomeLibrary/Header.h>`, then it would probably be correct to filter headers with

```
headerFilter = SomeLibrary/**
```

If a `headerFilter` is not specified, then all headers are included.

## Filter headers by module maps

Some libraries have proper `module.modulemap` or `module.map` files in their headers. For example, macOS and iOS system libraries and frameworks do. The [module map file](#) describes the correspondence between header files and modules. When the module maps are available, the headers from the modules that are not included directly can be filtered out using the experimental `excludeDependentModules` option of the `.def` file:

### 1.5.30 的新特性

```
headers = OpenGL/gl.h OpenGL/glu.h GLUT/glut.h
compilerOpts = -framework OpenGL -framework GLUT
excludeDependentModules = true
```

When both `excludeDependentModules` and `headerFilter` are used, they are applied as an intersection.

## C compiler and linker options

Options passed to the C compiler (used to analyze headers, such as preprocessor definitions) and the linker (used to link final executables) can be passed in the definition file as `compilerOpts` and `linkerOpts` respectively. For example:

```
compilerOpts = -DF00=bar
linkerOpts = -lpng
```

Target-specific options only applicable to the certain target can be specified as well:

```
compilerOpts = -DBAR=bar
compilerOpts.linux_x64 = -DF00=foo1
compilerOpts.mac_x64 = -DF00=foo2
```

With such a configuration, C headers will be analyzed with `-DBAR=bar -DF00=foo1` on Linux and with `-DBAR=bar -DF00=foo2` on macOS . Note that any definition file option can have both common and the platform-specific part.

## Add custom declarations

Sometimes it is required to add custom C declarations to the library before generating bindings (e.g., 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;
}
```

### 1.5.30 的新特性

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.

## Include a static library in your klib

Sometimes it is 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` clauses. For example:

```
headers = foo.h
staticLibraries = libfoo.a
libraryPaths = /opt/local/lib /usr/local/opt/curl/lib
```

When given the above snippet the `cinterop` tool will search `libfoo.a` in `/opt/local/lib` and `/usr/local/opt/curl/lib`, and if it is found include the library binary into `klib`.

When using such `klib` in your program, the library is linked automatically.

## 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 hints](#).
- 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

### 1.5.30 的新特性

representation is the main one and has the same name as the struct itself, for Kotlin enums it is named  `${type}Var` , for `CPointer<T>` it is `CPointerVar<T>` , and for most other types it is  `${type}Var` .

For types that have both representations, the one with a "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, e.g., 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` , e.g. `?:` , `?.` , `!!` etc.:

```
val path = getenv("PATH")?.toKString() ?: ""
```

Since the arrays are also mapped to `CPointer<T>` , it supports the `[]` operator for accessing values by index:

```
fun shift(ptr: CPointer<BytePtr>, 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*` , then the Kotlin binding accepts any `CPointer` .

Casting a pointer (including `COpaquePointer` ) can be done with `.reinterpret<T>` , e.g.:

```
val intPtr = bytePtr.reinterpret<IntVar>()
```

or

### 1.5.30 的新特性

```
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>()
```

Note that if the type of the result is known from the context, the type argument can be omitted as usual due to the type inference.

## Memory allocation

The native memory can be allocated using the `NativePlacement` interface, e.g.

```
val byteVar = placement.alloc<ByteVar>()
```

or

```
val bytePtr = placement.allocArray<ByteVar>(5)
```

The most "natural" placement is in the object `nativeHeap`. It corresponds to allocating native memory with `malloc` and provides an additional `.free()` operation to free allocated memory:

```
val buffer = nativeHeap.allocArray<ByteVar>(size)
<use buffer>
nativeHeap.free(buffer)
```

However, the lifetime of allocated memory is often bound to the lexical scope. It is possible to define such scope with `memScoped { ... }`. Inside the braces, the temporary placement is available as an implicit receiver, so it is possible to allocate native memory with `alloc` and `allocArray`, and the allocated memory will be automatically freed after leaving the scope.

### 1.5.30 的新特性

For example, the C function returning values through pointer parameters can be used like

```
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, the C function pointer-typed parameters are mapped to `CValuesRef<T>`. When passing `CPointer<T>` as the value of such a parameter, it is passed to the C function as is. However, the 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

### 1.5.30 的新特性

Unlike other pointers, the parameters of type `const char*` are represented as a Kotlin `String`. So it is 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, e.g.

```
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, a `noStringConversion` statement in the `.def` file could be used, i.e.

```
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:

```
memScoped {
 LoadCursorA(null, "cursor.bmp".cstr.ptr) // for ASCII version
 LoadCursorW(null, "cursor.bmp".wcstr.ptr) // for Unicode version
}
```

## Scope-local pointers

It is possible to create a scope-stable pointer of C representation of `CValues<T>` instance using the `CValues<T>.ptr` extension property, available under `memScoped { ... }`. It allows using the APIs which require C pointers with a lifetime bound to a certain `MemScope`. For example:

```
memScoped {
 items = arrayOfNulls<CPointer<ITEM>?>(6)
 arrayOf("one", "two").forEachIndexed { index, value -> items[index] = value.cst
 menu = new_menu("Menu".cstr.ptr, items.toCValues().ptr)
 ...
}
```

### 1.5.30 的新特性

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 the 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. It should be possible, if an API uses structures as handles, but if field access is required, there are the following conversion methods 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 places the `CValue<T>` to memory, and then runs the passed lambda with this placed value `T` as receiver. So to read a single field, the following code can be used:

```
val fieldValue = structValue.useContents { field }
```

## Callbacks

To convert a Kotlin function to a pointer to a C function,

`staticCFunction(::kotlinFunction)` can be used. It is also able to provide the 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 is passed to some C function (or written to the struct) as e.g. `void*`. 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 `StableRef` class.

To wrap the reference:

### 1.5.30 的新特性

```
val stableRef = StableRef.create(kotlinReference)
val voidPtr = stableRef.asCPointer()
```

where the `voidPtr` is a `COpaquePointer` and can be passed to the C function.

To unwrap the reference:

```
val stableRef = voidPtr.asStableRef<KotlinClass>()
val kotlinReference = stableRef.get()
```

where `kotlinReference` is the original wrapped reference.

The created `StableRef` should 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.

See the `samples/libcurl` for more details.

## Macros

Every C macro that expands to a constant is represented as a Kotlin property. Other macros are not supported. However, they can be exposed manually by wrapping them with supported declarations. E.g. function-like macro `F00` can be exposed as function `foo` by [adding the custom declaration](#) to the library:

```
headers = library/base.h

static inline int foo(int arg) {
 return F00(arg);
}
```

## Definition file hints

The `.def` file supports several options for adjusting the generated bindings.

### 1.5.30 的新特性

- `excludedFunctions` property value specifies a space-separated list of the names of functions that should be ignored. This may be required because a function declared in the C header is not generally guaranteed to be really callable, and it is often hard or impossible to figure this out automatically. This option can also be used to workaround a bug in the interop itself.
- `strictEnums` and `nonStrictEnums` properties values are space-separated lists of the enums that should be generated as a Kotlin enum or as integral values correspondingly. If the enum is not included into any of these lists, then it is generated according to the heuristics.
- `noStringConversion` property value is space-separated lists of the functions whose `const char*` parameters shall not be autoconverted as Kotlin string

## Portability

Sometimes the C libraries have function parameters or struct fields of a platform-dependent type, e.g. `long` or `size_t`. Kotlin itself doesn't provide neither implicit integer casts nor C-style integer casts (e.g. `(size_t) intValue`), so to make writing portable code in such cases easier, the `convert` method is provided:

```
fun ${type1}.convert<${type2}>(): ${type2}
```

where 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`.

The example of using `convert`:

```
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

### 1.5.30 的新特性

Kotlin objects could be pinned, i.e. their position in memory is guaranteed to be stable until unpinned, and pointers to such objects inner data could be passed to the C functions. For example

```
fun readData(fd: Int): String {
 val buffer = ByteArray(1024)
 buffer.usePinned { pinned ->
 while (true) {
 val length = recv(fd, pinned.addressOf(0), buffer.size.convert(), 0).to
 if (length <= 0) {
 break
 }
 // Now `buffer` has raw data obtained from the `recv()` call.
 }
 }
}
```

Here we use service function `usePinned`, which pins an object, executes block and unpins it on normal and exception paths.

# 映射来自 C 语言的原生数据类型——教程

在本教程中会学习到如何将 C 的数据类型在 Kotlin/Native 中可见，反之亦然。将会：

- 看到什么是 [C 语言中的数据类型](#)
- 创建一个 [小型 C 库](#) 来使这些类型向外暴露
- 探查为 C 库生成的 [Kotlin API](#)
- 找到如何将 [Kotlin 中的原生类型](#) 映射到 C 的方法

## C 语言中的类型

在 C 语言中都有什么类型？我们以维基百科上的 [C 语言数据类型](#) 这篇文章作为基础。

在 C 语言中有如下这些类型：

- 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`。

在 Kotlin 中查看 C 数据类型的最佳方法就是尝试一下。

## C 库示例

创建一个 `lib.h` 文件来看看如何将 C 函数映射到 Kotlin：

### 1.5.30 的新特性

```
#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
```

该文件缺少了本例不需要的 `extern "C"` 块，但是如果在使用 C++ 的重载函数的时候这也许是必要的。[C++ 兼容性问题](#)包含了更多关于此内容的细节。

对于每组 `.h` 文件，使用来自 Kotlin/Native 的 `cinterop 工具`来生成 Kotlin/Native 库，或者 `.klib`。生成的库将会桥接 Kotlin/Native 到 C 语言的调用。这包括在 `.h` 文件中各定义的 Kotlin 声明。只需要一个 `.h` 文件来运行 `cinterop` 工具。并且不需要创建一个 `lib.c` 文件，除非想编译并运行该示例。更多关于这些内容的细节被涵盖在[与 C 语言互操作](#)页面。这篇教程的内容足够使用下面的内容来创建 `lib.def` 文件：

```
headers = lib.h
```

可以在 `---` 分隔符之后将所有声明直接包含在 `.def` 文件中。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。方法体同样被编译以及完全包含到二进制文件中。使用这个功能并且在不使用 C 编译器的情况下得到一个可运行的示例。为了实现这个，需要在 `lib.h` 文件中添加 C 函数的实现，并将这些函数放入 `.def` 文件中。会得到如下的 `interop.def` 结果：

```

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` 文件足以编译和运行应用程序或在 IDE 中打开它。现在是时候创建项目文件，且在 [IntelliJ IDEA](#) 中打开并运行它。

## 探查为 C 库生成的 Kotlin API

## 1.5.30 的新特性

While it is possible to use the command line, either directly or by combining it with a script file (such as `.sh` or `.bat` file), this approach doesn't scale well for big projects that have hundreds of files and libraries. It is then better to use the Kotlin/Native compiler with a build system, as it helps to download and cache the Kotlin/Native compiler binaries and libraries with transitive dependencies and run the compiler and tests. Kotlin/Native can use the [Gradle](#) build system through the [kotlin-multiplatform](#) plugin.

We covered the basics of setting up an IDE compatible project with Gradle in the [A Basic Kotlin/Native Application](#) tutorial. Please check it out if you are looking for detailed first steps and instructions on how to start a new Kotlin/Native project and open it in IntelliJ IDEA. In this tutorial, we'll look at the advanced C interop related usages of Kotlin/Native and [multiplatform](#) builds with Gradle.

First, create a project folder. All the paths in this tutorial will be relative to this folder. Sometimes the missing directories will have to be created before any new files can be added.

Use the following `build.gradle(.kts)` Gradle build file:

【Kotlin】

## 1.5.30 的新特性

```
plugins {
 kotlin("multiplatform") version "1.6.10"
}

repositories {
 mavenCentral()
}

kotlin {
 linuxX64("native") { // on Linux
 // macosX64("native") { // on x86_64 macOS
 // macosArm64("native") { // on Apple Silicon macOS
 // mingwX64("native") { // on Windows
 val main by compilations.getting
 val interop by main.cinterops.creating

 binaries {
 executable()
 }
 }
 }

 tasks.wrapper {
 gradleVersion = "6.7.1"
 distributionType = Wrapper.DistributionType.BIN
 }
}
```

【Groovy】

## 1.5.30 的新特性

```
plugins {
 id 'org.jetbrains.kotlin.multiplatform' version '1.6.10'
}

repositories {
 mavenCentral()
}

kotlin {
 linuxX64('native') { // on Linux
 // macosX64("native") { // on x86_64 macOS
 // macosArm64("native") { // on Apple Silicon macOS
 // mingwX64('native') { // on Windows
 compilations.main.cinterops {
 interop
 }

 binaries {
 executable()
 }
 }
 }

 wrapper {
 gradleVersion = '6.7.1'
 distributionType = 'BIN'
 }
}
```

The project file configures the C interop as an additional step of the build. Let's move the `interop.def` file to the `src/nativeInterop/cinterop` directory. Gradle recommends using conventions instead of configurations, for example, the source files are expected to be in the `src/nativeMain/kotlin` folder. By default, all the symbols from C are imported to the `interop` package, you may want to import the whole package in our `.kt` files. Check out the [kotlin-multiplatform](#) plugin documentation to learn about all the different ways you could configure it.

使用以下内容来创建一个 `src/nativeMain/kotlin/hello.kt` 存根文件来查看 C 的原生类型声明是如何在 Kotlin 中可见的：

### 1.5.30 的新特性

```
import interop.*\n\nfun main() {\n println("Hello Kotlin/Native!")\n\n ints(/* fix me */)\n uints(/* fix me */)\n doubles(/* fix me */)\n}
```

现在已经准备好在 IntelliJ IDEA 中打开这个项目并且看看如何修正这个示例项目。当做完这些之后，检查 C 的原生类型是如何映射到 Kotlin/Native 的。

## Kotlin 中的原生类型

在 IntelliJ IDEA 的 **Go to | Declaration** 或编译器错误的帮助下会看到为 C 函数生成的 API：

```
fun ints(c: Byte, d: Short, e: Int, f: Long)\nfun uints(c: UByte, d: UShort, e: UInt, f: ULong)\nfun doubles(a: Float, b: Double)
```

C 类型按照我们期望的方式进行了映射，注意，`char` 类型映射到了 `kotlin.Byte`，因为它通常是 8 位有符号值。

C	Kotlin
char	<code>kotlin.Byte</code>
unsigned char	<code>kotlin.UByte</code>
short	<code>kotlin.Short</code>
unsigned short	<code>kotlin.UShort</code>
int	<code>kotlin.Int</code>
unsigned int	<code>kotlin.UInt</code>
long long	<code>kotlin.Long</code>
unsigned long long	<code>kotlin.ULong</code>
float	<code>kotlin.Float</code>
double	<code>kotlin.Double</code>

## 修正代码

看过了所有的定义并且是时候来修改代码了。在 IDE 中运行

`runDebugExecutableNative` Gradle 任务或使用下面的命令来运行代码：

```
./gradlew runDebugExecutableNative
```

`hello.kt` 文件中的代码最终看起来会是这样的：

```
import interop.*

fun main() {
 println("Hello Kotlin/Native!")

 ints(1, 2, 3, 4)
 uints(5, 6, 7, 8)
 doubles(9.0f, 10.0)
}
```

## 接下来

在接下来的几篇教程中继续探索更复杂的 C 语言类型及其在 Kotlin/Native 中的表示：

- 映射来自 C 语言的结构与联合类型
- 映射来自 C 语言的函数指针
- 映射来自 C 语言的字符串

这篇与 C 语言互操作文档涵盖了更多的高级互操作场景。

# 映射来自 C 语言的结构与联合类型——教程

这是本系列的第二篇教程。本系列的第一篇教程是[映射来自 C 语言的原生数据类型](#)。系列其余教程包括[映射来自 C 语言的函数指针](#)与[映射来自 C 语言的字符串](#)。

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

理解在 Kotlin 与 C 之间进行映射的最好方式是尝试编写一个小型示例。我们将在 C 语言中声明一个结构体与一个联合体，并以此来观察如何将它们映射到 Kotlin 中。

Kotlin/Native 附带 `cinterop` 工具，该工具可以生成 C 语言与 Kotlin 之间的绑定。它使用一个 `.def` 文件指定一个 C 库来导入。更多的细节将在[与 C 库互操作](#)教程中讨论。

在[之前的教程](#)中创建过一个 `lib.h` 文件。这次，在 `---` 分割行之后，直接将那些声明导入到 `interop.def` 文件：

```

```

```
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) {}
```

该 `interop.def` 文件足够用来编译并运行应用程序，或在 IDE 中打开它。现在创建项目文件，并在 [IntelliJ IDEA](#) 中打开该项目，然后运行它。

## 探查为 C 库生成的 Kotlin API

While it is possible to use the command line, either directly or by combining it with a script file (such as `.sh` or `.bat` file), this approach doesn't scale well for big projects that have hundreds of files and libraries. It is then better to use the Kotlin/Native compiler with a build system, as it helps to download and cache the Kotlin/Native compiler binaries and libraries with transitive dependencies and run the compiler and tests. Kotlin/Native can use the [Gradle](#) build system through the [kotlin-multiplatform](#) plugin.

We covered the basics of setting up an IDE compatible project with Gradle in the [A Basic Kotlin/Native Application](#) tutorial. Please check it out if you are looking for detailed first steps and instructions on how to start a new Kotlin/Native project and open it in IntelliJ IDEA. In this tutorial, we'll look at the advanced C interop related usages of Kotlin/Native and [multiplatform](#) builds with Gradle.

First, create a project folder. All the paths in this tutorial will be relative to this folder. Sometimes the missing directories will have to be created before any new files can be added.

## 1.5.30 的新特性

Use the following `build.gradle(.kts)` Gradle build file:

### 【Kotlin】

```
plugins {
 kotlin("multiplatform") version "1.6.10"
}

repositories {
 mavenCentral()
}

kotlin {
 linuxX64("native") { // on Linux
 // macosX64("native") { // on x86_64 macOS
 // macosArm64("native") { // on Apple Silicon macOS
 // mingwX64("native") { // on Windows
 val main by compilations.getting
 val interop by main.cinterops.creating

 binaries {
 executable()
 }
 }
 }

 tasks.wrapper {
 gradleVersion = "6.7.1"
 distributionType = Wrapper.DistributionType.BIN
 }
}
```

### 【Groovy】

## 1.5.30 的新特性

```
plugins {
 id 'org.jetbrains.kotlin.multiplatform' version '1.6.10'
}

repositories {
 mavenCentral()
}

kotlin {
 linuxX64('native') { // on Linux
 // macosX64("native") { // on x86_64 macOS
 // macosArm64("native") { // on Apple Silicon macOS
 // mingwX64('native') { // on Windows
 compilations.main.cinterops {
 interop
 }

 binaries {
 executable()
 }
 }
 }

 wrapper {
 gradleVersion = '6.7.1'
 distributionType = 'BIN'
 }
}
```

The project file configures the C interop as an additional step of the build. Let's move the `interop.def` file to the `src/nativeInterop/cinterop` directory. Gradle recommends using conventions instead of configurations, for example, the source files are expected to be in the `src/nativeMain/kotlin` folder. By default, all the symbols from C are imported to the `interop` package, you may want to import the whole package in our `.kt` files. Check out the [kotlin-multiplatform](#) plugin documentation to learn about all the different ways you could configure it.

Create a `src/nativeMain/kotlin/hello.kt` stub file with the following content to see how C declarations are visible from Kotlin:

### 1.5.30 的新特性

```
import interop.*

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*/)
}
```

现在已经准备好在 IntelliJ IDEA 中打开这个项目并且看看如何修正这个示例项目。当做了这些之后，会看到 C 的原生类型已经被映射到了 Kotlin/Native。

## Kotlin 中的原生类型

通过 IntelliJ IDEA 的**Go to | Declaration** 或编译器错误的帮助，会看到如下的为 C 函数、`struct` 以及 `union` 生成的 API：

```
fun struct_by_value(s: CValue<MyStruct>)
fun struct_by_pointer(s: CValuesRef<MyStruct>?)

fun union_by_value(u: CValue<MyUnion>)
fun union_by_pointer(u: CValuesRef<MyUnion>?)

class MyStruct constructor(rawPtr: NativePtr /* = NativePtr */) : CStructVar {
 var a: Int
 var b: Double
 companion object : CStructVar.Type
}

class MyUnion constructor(rawPtr: NativePtr /* = NativePtr */) : CStructVar {
 var a: Int
 val b: MyStruct
 var c: Float
 companion object : CStructVar.Type
}
```

可以看到 `cinterop` 为我们的 `struct` 与 `union` 类型生成了包装类型。为在 C 中声明的 `MyStruct` 与 `MyUnion` 类型，分别为其生成了 Kotlin 类 `MyStruct` 与 `MyUnion`。该包装器继承自 `CStructVar` 基类并将所有的字段声明为了 Kotlin 属性。它使用 `CValue<T>` 来表示一个值类型的结构体参数并使用 `CValuesRef<T>?` 来表示传递一个结构体或共用体的指针。

### 1.5.30 的新特性

从技术上讲，在 Kotlin 看来 `struct` 与 `union` 类型之间没有区别。请注意，Kotlin 中 `MyUnion` 类的 `a`、`b` 以及 `c` 属性使用了相同的位置来进行读写值的操作，就像 C 语言中的 `union` 一样。

更多细节与高级用例将在 [C 互操作文档](#) 中介绍

## 在 Kotlin 中使用结构与联合类型

在 Kotlin 中使用为 C 的 `struct` 与 `union` 类型生成的包装器非常简单。由于生成了属性，使得在 Kotlin 代码中使用它们是非常自然的。迄今为止唯一的问题是，如何为这些类创建新的实例。正如在 `MyStruct` 与 `MyUnion` 的声明中所见，它们的构造函数需要一个 `NativePtr`。当然，不愿意手动处理指针。作为替代，可以使用 Kotlin API 来为我们实例化这些对象。

我们来看一看生成的函数，它将 `MyStruct` 与 `MyUnion` 作为参数。看到了值类型参数表示为 `kotlinx.cinterop.CValue<T>`。而指针类型参数表示为 `kotlinx.cinterop.CValuesRef<T>`。Kotlin 给我们提供了 API 使得处理这两者都非常简单，我们来尝试一下并看看结果。

### 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>` 对象实例。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>
```

现在是时候来看看如何使用 `cValue` 并传递值类型参数：

### 1.5.30 的新特性

```
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。首先，需要 `MyStruct` 与 `MyUnion` 类的实例。直接在原生内存中创建它们。使用

```
fun <reified T : kotlinx.cinterop.CVariable> alloc(): T
```

`kotlinx.cinterop.NativePlacement` 上的扩展函数来做这个。

`NativePlacement` 代表原生内存，类似于 `malloc` 与 `free` 函数。这里有几个 `NativePlacement` 的实现。其中全局的那个是调用 `kotlinx.cinterop.nativeHeap` 并且不要忘记在使用过后调用 `nativeHeap.free(...)` 函数来释放内存。

另一个配置是使用

```
fun <R> memScoped(block: kotlinx.cinterop.MemScope.() -> R): R
```

函数。它创建一个短生命周期的内存分配作用域，并且所有的分配都将在 `block` 结束之后自动清理。

调用带指针类型参数的函数的代码看起来会是这样：

### 1.5.30 的新特性

```
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)
 }
}
```

请注意，这段代码使用的扩展属性 `ptr` 来自 `memScoped` lambda 表达式的接收者类型，将 `MyStruct` 与 `MyUnion` 实例转换为原生指针。

`MyStruct` 与 `MyUnion` 类具有指向原生内存的指针。当 `memScoped` 函数结束的时候，即 `block` 结尾的时候，内存将释放。请确保指针没有在 `memScoped` 调用的外部使用。可以为指针使用 `Arena()` 或 `nativeHeap` 这样应该有更长的可用时间，或者将它们缓存在 C 库中。

## 在 `CValue<T>` 与 `CValuesRef<T>` 之间转换

当然，这里有一些用例——当需要将一个结构体作为值传递给一个调用，另一种是将同一个结构体作为引用传递给另一个调用。这在 Kotlin/Native 中同样也是可行的。这里将需要一个 `NativePlacement`。

我们看看现在首先将 `CValue<T>` 转换为一个指针：

```
fun callMix_ref() {
 val cStruct = cValue<MyStruct> {
 a = 42
 b = 3.14
 }

 memScoped {
 struct_by_pointer(cStruct.ptr)
 }
}
```

### 1.5.30 的新特性

这段代码使用的扩展属性 `ptr` 来自 `memScoped` `lambda` 表达式的接收者类型，将 `MyStruct` 与 `MyUnion` 实例转换为原生指针。这些指针只在 `memScoped` 块内是有效的。

对于反向转换，即将指针转换为值类型变量，我们可以调用 `readValue()` 扩展函数：

```
fun callMix_value() {
 memScoped {
 val cStruct = alloc<MyStruct>()
 cStruct.a = 42
 cStruct.b = 3.14

 struct_by_value(cStruct.readValue())
 }
}
```

## 运行代码

现在，学习了如何在我们的代码中使用 C 声明，已经准备好在一个真实的示例中尝试它的输出。我们来修改代码并看看如何[在 IDE 中](#)调用 `runDebugExecutableNative` Gradle 任务来运行它。或者使用以下的控制台命令：

```
./gradlew runDebugExecutableNative
```

`hello.kt` 文件中的最终代码看起来会是这样：

### 1.5.30 的新特性

```
import interop.*
import kotlinx.cinterop.alloc
import kotlinx.cinterop.cValue
import kotlinx.cinterop.memScoped
import kotlinx.cinterop.ptr
import kotlinx.cinterop.readValue

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)
 }
}
```

## 接下来

在以下几篇相关的教程中继续浏览 C 语言的类型以及它们在 Kotlin/Native 中的表示：

- 映射来自 C 语言的原生数据类型
- 映射来自 C 语言的函数指针
- 映射来自 C 语言的字符串

这篇与 [C 语言互操作文档](#)涵盖了更多的高级互操作场景

# 映射来自 C 语言的函数指针——教程

这是本系列的第三篇教程。本系列的第一篇教程是[映射来自 C 语言的原生数据类型](#)。系列其余教程包括[映射来自 C 语言的结构与联合类型](#)与[映射来自 C 语言的字符串](#)。

在本篇教程中我们将学习如何：

- 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 之间进行映射的最好方式是尝试编写一个小型示例。声明一个函数，它接收一个函数指针作为参数，而另一个函数返回一个函数指针。

Kotlin/Native 附带 `cinterop` 工具；该工具可以生成 C 语言与 Kotlin 之间的绑定。它使用一个 `.def` 文件指定一个 C 库来导入。更多的细节将在[与 C 库互操作](#)教程中讨论。

最快速的尝试 C API 映射的方法是将所有的 C 声明写到 `interop.def` 文件，而不用创建任何 `.h` 或 `.c` 文件。在 `.def` 文件中，所有的 C 声明都在特殊的 `---` 分割行之后。

```

int myFun(int i) {
 return i+1;
}

typedef int (*MyFun)(int);

void accept_fun(MyFun f) {
 f(42);
}

MyFun supply_fun() {
 return myFun;
}
```

该 `interop.def` 文件足够用来编译并运行应用程序，或在 IDE 中打开它。现在是时候创建项目文件，并在 [IntelliJ IDEA](#) 中打开这个项目，然后运行它。

## 探查为 C 库生成的 Kotlin API

While it is possible to use the command line, either directly or by combining it with a script file (such as `.sh` or `.bat` file), this approach doesn't scale well for big projects that have hundreds of files and libraries. It is then better to use the Kotlin/Native compiler with a build system, as it helps to download and cache the Kotlin/Native compiler binaries and libraries with transitive dependencies and run the compiler and tests. Kotlin/Native can use the [Gradle](#) build system through the [kotlin-multiplatform](#) plugin.

We covered the basics of setting up an IDE compatible project with Gradle in the [A Basic Kotlin/Native Application](#) tutorial. Please check it out if you are looking for detailed first steps and instructions on how to start a new Kotlin/Native project and open it in IntelliJ IDEA. In this tutorial, we'll look at the advanced C interop related usages of Kotlin/Native and [multiplatform](#) builds with Gradle.

First, create a project folder. All the paths in this tutorial will be relative to this folder. Sometimes the missing directories will have to be created before any new files can be added.

Use the following `build.gradle(.kts)` Gradle build file:

【Kotlin】

## 1.5.30 的新特性

```
plugins {
 kotlin("multiplatform") version "1.6.10"
}

repositories {
 mavenCentral()
}

kotlin {
 linuxX64("native") { // on Linux
 // macosX64("native") { // on x86_64 macOS
 // macosArm64("native") { // on Apple Silicon macOS
 // mingwX64("native") { // on Windows
 val main by compilations.getting
 val interop by main.cinterops.creating

 binaries {
 executable()
 }
 }
 }

 tasks.wrapper {
 gradleVersion = "6.7.1"
 distributionType = Wrapper.DistributionType.BIN
 }
}
```

【Groovy】

## 1.5.30 的新特性

```
plugins {
 id 'org.jetbrains.kotlin.multiplatform' version '1.6.10'
}

repositories {
 mavenCentral()
}

kotlin {
 linuxX64('native') { // on Linux
 // macosX64("native") { // on x86_64 macOS
 // macosArm64("native") { // on Apple Silicon macOS
 // mingwX64('native') { // on Windows
 compilations.main.cinterops {
 interop
 }

 binaries {
 executable()
 }
 }
 }

 wrapper {
 gradleVersion = '6.7.1'
 distributionType = 'BIN'
 }
}
```

The project file configures the C interop as an additional step of the build. Let's move the `interop.def` file to the `src/nativeInterop/cinterop` directory. Gradle recommends using conventions instead of configurations, for example, the source files are expected to be in the `src/nativeMain/kotlin` folder. By default, all the symbols from C are imported to the `interop` package, you may want to import the whole package in our `.kt` files. Check out the [kotlin-multiplatform](#) plugin documentation to learn about all the different ways you could configure it.

我们使用下面的内容创建一个 `src/nativeMain/kotlin/hello.kt` 存根文件，以用来观察 C 中的原生类型是如何在 Kotlin 中可见的：

### 1.5.30 的新特性

```
import interop.*\n\nfun main() {\n println("Hello Kotlin/Native!")\n\n accept_fun(https://kotlinlang.org/*fix me */)\n val useMe = supply_fun()\n}
```

现在已经准备好[在 IntelliJ IDEA 中打开这个项目](#)并且看看如何修正这个示例项目。当做了这些之后，观察到 C 函数是如何映射到 Kotlin/Native 声明的。

## Kotlin 中的 C 函数指针

通过 IntelliJ IDEA 的 **Go To | Declaration or Usages** 或编译器错误的帮助，可以看到为 C 函数生成的如下声明：

```
fun accept_fun(f: MyFun? /* = CPointer<CFunction<(Int) -> Int>>? */)\nfun supply_fun(): MyFun? /* = CPointer<CFunction<(Int) -> Int>>? */\n\nfun myFun(i: kotlin.Int): kotlin.Int\n\ntypealias MyFun = kotlndx.cinterop.CPointer<kotlndx.cinterop.CFunction<(kotlin.Int)\n\n\ntypealias MyFunVar = kotlndx.cinterop.CPointerVarOf<lib.MyFun>
```

可以看到 C 中的函数的 `typedef` 已经转换成了 Kotlin `typealias`。它使用 `CPointer<..>` 类型表示指针参数，使用 `CFunction<(Int)->Int>` 表示函数签名。这里有一个 `invoke` 操作符扩展函数，它可以用于所有的 `CPointer<CFunction<..>` 类型，因此它可以在任何一个可以调用其它 Kotlin 函数的地方调用。

## 将 Kotlin 函数作为 C 函数指针传递

是时候尝试在我们的 Kotlin 程序中使用 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：

### 1.5.30 的新特性

```
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，它不能使用函数中的局部变量。我们只能使用全局可见的声明。抛出来自 `staticCFunction{..}` 的异常将导致非确定性的副作用。确保代码不会从中抛出任何意想不到的异常是非常重要的。

## 在 Kotlin 中使用 C 函数指针

接下来是调用 `supply_fun()` 获得一个 C 函数指针，并调用它：

```
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到一个可空的 `CPointer<CFunction<..>` 对象。这里首先需要显式检查 `null` 值。为此，上述代码中使用 **elvis 操作符**。`cinterop` 工具帮助我们将一个 C 函数指针转换为一个 Kotlin 中可以简单调用的对象。这就是我们在最后一行所做的事。

## 修复代码

看过了所有的声明，是时候修改并运行代码了。在 IDE 中运行 `runDebugExecutableNative` Gradle 任务或使用以下的命令来运行该代码：

```
./gradlew runDebugExecutableNative
```

`hello.kt` 文件中的代码最终看起来会是这样的：

### 1.5.30 的新特性

```
import interop.*
import kotlinx.cinterop.*

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)
}
```

## 接下来

继续在以下几篇教程中继续探索更多的 C 语言类型及其在 Kotlin/Native 中的表示：

- 映射来自 C 语言的原生数据类型
- 映射来自 C 语言的结构与联合类型
- 映射来自 C 语言的字符串

这篇[与 C 语言互操作](#)涵盖了更多的高级互操作场景。

# 映射来自 C 语言的字符串——教程

这是本系列的最后一篇教程。本系列的第一篇教程是[映射来自 C 语言的原生数据类型](#)。系列其余教程包括[映射来自 C 语言的结构与联合类型](#)与[映射来自 C 语言的函数指针](#)。

在本教程中会看到如何在 Kotlin/Native 中处理 C 语言的字符串。将会学习到如何：

- 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

在 C 语言中，没有专门用于字符串的类型。开发者需要从方法签名或文档中才能得知给定的 `char *` 在上下文中是否表示 C 字符串。C 语言中的字符串以空值终止，即在字节序列尾部添加零字符 `\0` 来标记字符串终止。通常，使用 [UTF-8 编码字符串](#)。UTF-8 编码使用可变长度的字符，并且它向后兼容 [ASCII](#)。Kotlin/Native 默认使用 UTF-8 字符编码。

理解在 C 语言与 Kotlin 之间进行映射的最好方式是尝试编写一个小型示例。为此创建一个小型库的头文件。首先，为以下处理 C 字符串的函数声明创建一个 `lib.h` 文件。

```
#ifndef LIB2_H_INCLUDED
#define LIB2_H_INCLUDED

void pass_string(char* str);
char* return_string();
int copy_string(char* str, int size);

#endif
```

在示例中，见到了大多数情况下受欢迎的方式来传递或接收 C 语言中的字符串。小心地返回 `return_string`。通常，最好确保我们使用正确的函数来处理被正确调用 `free(..)` 函数返回的 `char*`。

Kotlin/Native 附带 `cinterop` 工具；该工具可以生成 C 语言与 Kotlin 之间的绑定。它使用一个 `.def` 文件指定一个 C 库来导入。更多的细节将在[与 C 库互操作](#)教程中讨论。最快速的尝试 C API 映射的方法是将所有的 C 声明写到 `interop.def` 文件，而不用创

### 1.5.30 的新特性

建任何 `.h` 或 `.c` 文件。在 `.def` 文件中，所有的 C 声明都在特殊的 `---` 分割行之后。

```
headers = lib.h

void pass_string(char* str) {

char* return_string() {
 return "C stirng";
}

int copy_string(char* str, int size) {
 *str++ = 'C';
 *str++ = ' ';
 *str++ = 'K';
 *str++ = '/';
 *str++ = 'N';
 *str++ = 0;
 return 0;
}
```

该 `interop.def` 文件足够用来编译并运行应用程序，或在 IDE 中打开它。现在是时候创建项目文件，并在 [IntelliJ IDEA](#) 中打开这个项目，然后运行它。

## 探查为 C 库生成的 Kotlin API

While it is possible to use the command line, either directly or by combining it with a script file (such as `.sh` or `.bat` file), this approach doesn't scale well for big projects that have hundreds of files and libraries. It is then better to use the Kotlin/Native compiler with a build system, as it helps to download and cache the Kotlin/Native compiler binaries and libraries with transitive dependencies and run the compiler and tests. Kotlin/Native can use the [Gradle](#) build system through the [kotlin-multiplatform](#) plugin.

We covered the basics of setting up an IDE compatible project with Gradle in the [A Basic Kotlin/Native Application](#) tutorial. Please check it out if you are looking for detailed first steps and instructions on how to start a new Kotlin/Native project and open it in IntelliJ IDEA. In this tutorial, we'll look at the advanced C interop related usages of Kotlin/Native and [multiplatform](#) builds with Gradle.

## 1.5.30 的新特性

First, create a project folder. All the paths in this tutorial will be relative to this folder. Sometimes the missing directories will have to be created before any new files can be added.

Use the following `build.gradle(.kts)` Gradle build file:

### 【Kotlin】

```
plugins {
 kotlin("multiplatform") version "1.6.10"
}

repositories {
 mavenCentral()
}

kotlin {
 linuxX64("native") { // on Linux
 // macosX64("native") { // on x86_64 macOS
 // macosArm64("native") { // on Apple Silicon macOS
 // mingwX64("native") { // on Windows
 val main by compilations.getting
 val interop by main.cinterops.creating

 binaries {
 executable()
 }
 }
 }

 tasks.wrapper {
 gradleVersion = "6.7.1"
 distributionType = Wrapper.DistributionType.BIN
 }
}
```

### 【Groovy】

## 1.5.30 的新特性

```
plugins {
 id 'org.jetbrains.kotlin.multiplatform' version '1.6.10'
}

repositories {
 mavenCentral()
}

kotlin {
 linuxX64('native') { // on Linux
 // macosX64("native") { // on x86_64 macOS
 // macosArm64("native") { // on Apple Silicon macOS
 // mingwX64('native') { // on Windows
 compilations.main.cinterops {
 interop
 }

 binaries {
 executable()
 }
 }
 }

 wrapper {
 gradleVersion = '6.7.1'
 distributionType = 'BIN'
 }
}
```

The project file configures the C interop as an additional step of the build. Let's move the `interop.def` file to the `src/nativeInterop/cinterop` directory. Gradle recommends using conventions instead of configurations, for example, the source files are expected to be in the `src/nativeMain/kotlin` folder. By default, all the symbols from C are imported to the `interop` package, you may want to import the whole package in our `.kt` files. Check out the [kotlin-multiplatform](#) plugin documentation to learn about all the different ways you could configure it.

我们使用下面的内容创建一个 `src/nativeMain/kotlin/hello.kt` 存根文件，以用来观察 C 中的原生类型是如何在 Kotlin 中可见的：

### 1.5.30 的新特性

```
import interop.*

fun main() {
 println("Hello Kotlin/Native!")

 pass_string(/*fix me*/)
 val useMe = return_string()
 val useMe2 = copy_string(/*fix me*/)
}
```

现在已经准备好在 IntelliJ IDEA 中打开这个项目并且看看如何修正这个示例项目。当做了这些之后，可以看下 C 函数是如何映射到 Kotlin/Native 声明的。

## Kotlin 中的原生类型

通过 IntelliJ IDEA 的 **Go to | Declaration** 或编译器错误的帮助，可以看到如下为 C 函数生成的声明：

```
fun pass_string(str: CValuesRef<ByteVar> /* = ByteVarOf<Byte> */?)
fun return_string(): CPointer<ByteVar> /* = ByteVarOf<Byte> */?
fun copy_string(str: CValuesRef<ByteVar> /* = ByteVarOf<Byte> */?, size: Int): 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，可以看到 `str` 被 `CValuesRef<ByteVar>?` 表示。该类型是可空的，可以简单地将 Kotlin `null` 作为参数值。

## 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`：

```
fun passStringToC() {
 val str = "this is a Kotlin String"
 pass_string(str.cstr)
}
```

将 Kotlin 字符串传递到 C 非常简单，幸亏事实上在 Kotlin 中有 `String.cstr` 扩展属性来应对这种情况。当需要 UTF-16 字符编码时，也有 `String.wcstr` 来应对这种情况。

## 在 Kotlin 中读取 C 字符串

这次会从 `return_string` 函数获取一个返回的 `char *` 并将其转换为一个 Kotlin 字符串。为此在 Kotlin 中做了如下这些事：

```
fun passStringToC() {
 val stringFromC = return_string()?.toKString()

 println("Returned from C: $stringFromC")
}
```

在上面的示例中的代码使用了 `toKString()` 扩展函数。请不要与 `toString()` 函数混淆。`toKString()` 在 Kotlin 中拥有两个版本的重载扩展函数：

```
fun CPointer<ByteVar>.toKString(): String
fun CPointer<ShortVar>.toKString(): String
```

第一个重载扩展函数将得到一个 `char *` 作为 UTF-8 字符串并转换到 String。第二个重载函数做了相同的事，但是它针对 UTF-16 字符串。

## 在 Kotlin 中接收 C 字符串字节

这次我们将要求 C 函数将 C 字符串写入给定的缓冲区。这个函数被称为 `copy_string`。它需要一个指针来定位写入字符的位置并分配缓冲区的大小。该函数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：

```
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.stringFromUtf8()
 println("Message from C: $copiedStringFromC")
}
```

### 1.5.30 的新特性

首先，需要有一个原生的指针来传递这个 C 函数。使用 `usePinned` 扩展函数来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。使用另一个扩展函数 `ByteArray.stringFromUtf8()` 将字节数组转换为一个 Kotlin `String`，假设它是 UTF-8 编码的。

## 修复代码

看过了所有的定义并且是时候来修改代码了。[在 IDE 中运行](#)

`runDebugExecutableNative` Gradle 任务或使用下面的命令来运行代码：

```
./gradlew runDebugExecutableNative
```

`hello.kt` 文件中的代码最终看起来会是这样的：

```
import interop.*
import kotlinx.cinterop.*

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 string from C")
 pinned.get().stringFromUtf8()
 }

 println(copyFromC)
}
```

## 接下来

继续在以下几篇教程中继续探索更多的 C 语言类型及其在 Kotlin/Native 中的表示：

- 映射来自 C 语言的原生数据类型
- 映射来自 C 语言的结构与联合类型

### 1.5.30 的新特性

- 映射来自 C 语言的函数指针

这篇[与 C 语言互操作文档](#)涵盖了更多的高级互操作场景

# 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.

While it is possible to use the command line, either directly or by combining it with a script file (such as a `.sh` or a `.bat` file), this approach doesn't scale well for big projects with hundreds of files and libraries. In this case, it is better to use the Kotlin/Native compiler with a build system, as it helps download and cache the Kotlin/Native compiler binaries and libraries with transitive dependencies and run the compiler and tests. Kotlin/Native can use the [Gradle](#) build system through the [kotlin-multiplatform](#) Plugin.

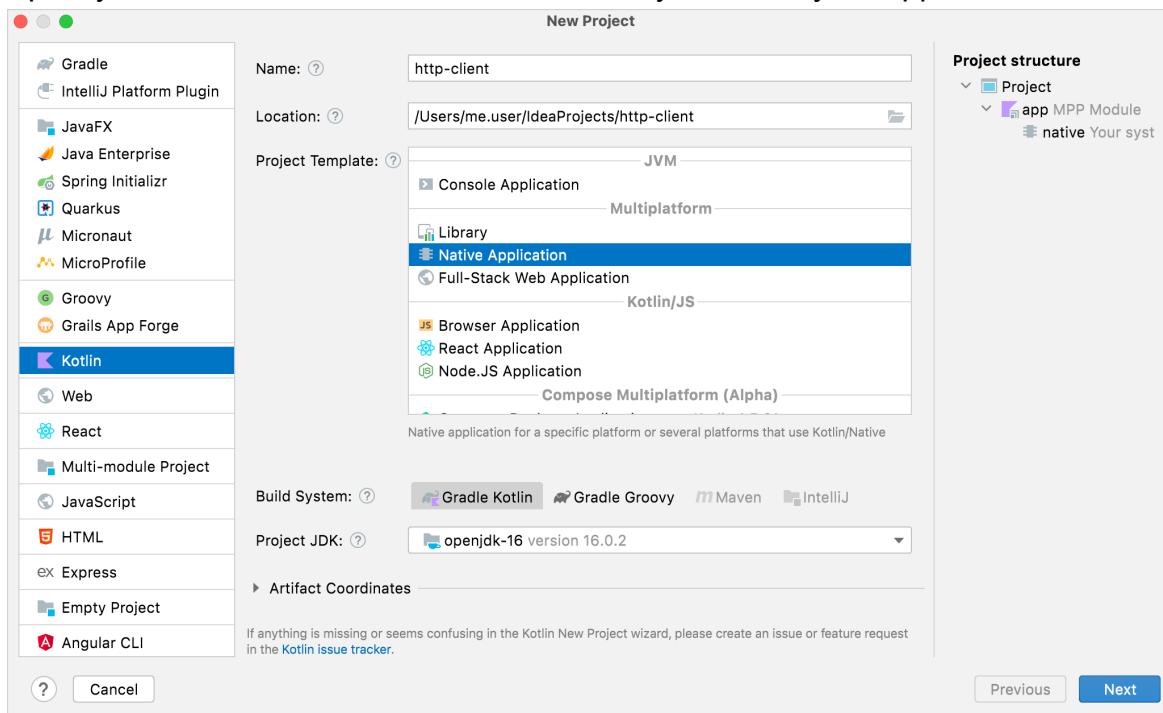
To get started, install the latest version of [IntelliJ IDEA](#). The tutorial is suitable for both IntelliJ IDEA Community Edition and IntelliJ IDEA Ultimate.

## 创建一个 Kotlin/Native 项目

1. In IntelliJ IDEA, select **File | New | Project**.
2. In the panel on the left, select **Kotlin | Native Application**.

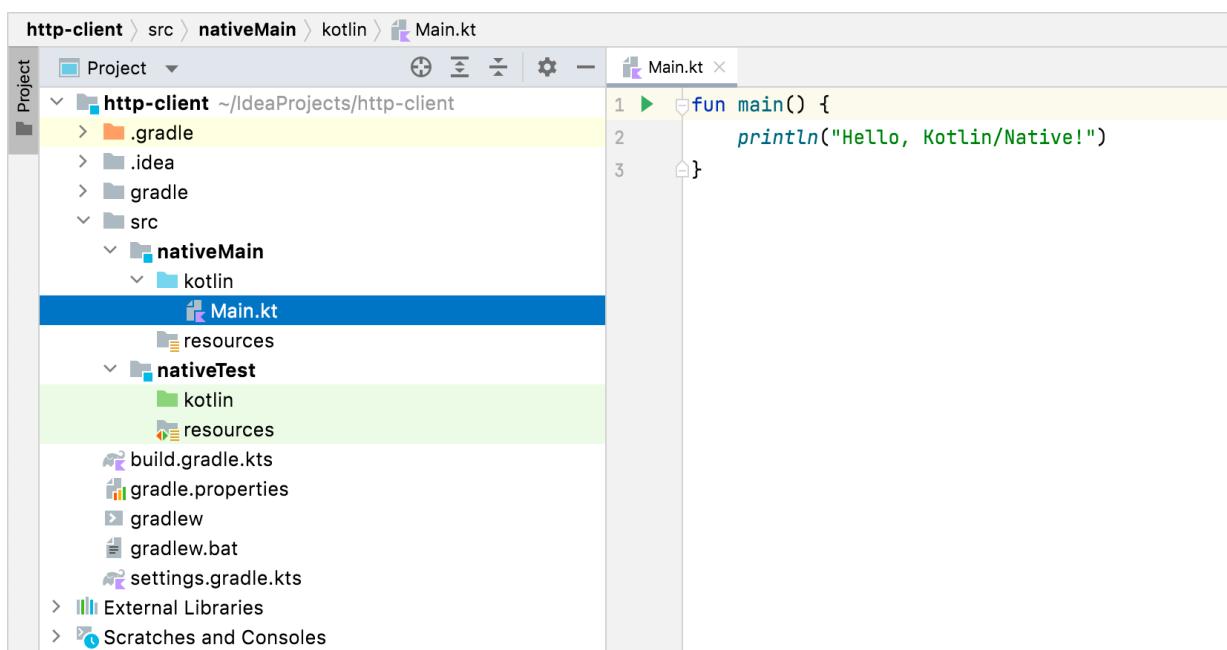
## 1.5.30 的新特性

### 3. Specify the name and select the folder where you'll save your application.



### 4. Click **Next** and then **Finish**.

IntelliJ IDEA will create a new project with the files and folders you need to get you 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 a folder named `NativeMain` with its corresponding `NativeTest`. For this tutorial, keep the folder structure as is.



Along with your new project, a `build.gradle(.kts)` file is generated. Pay special attention to the following in the build file:

## 1.5.30 的新特性

### 【Kotlin】

```
kotlin {
 val hostOs = System.getProperty("os.name")
 val isMingwX64 = hostOs.startsWith("Windows")
 val nativeTarget = when {
 hostOs == "Mac OS X" -> macosX64("native")
 hostOs == "Linux" -> linuxX64("native")
 isMingwX64 -> mingwX64("native")
 else -> throw GradleException("Host OS is not supported in Kotlin/Native.")
 }

 nativeTarget.apply {
 binaries {
 executable {
 entryPoint = "main"
 }
 }
 }
}
```

### 【Groovy】

```
kotlin {
 def hostOs = System.getProperty("os.name")
 def isMingwX64 = hostOs.startsWith("Windows")
 def nativeTarget
 if (hostOs == "Mac OS X") nativeTarget = macosX64('native')
 else if (hostOs == "Linux") nativeTarget = linuxX64("native")
 else if (isMingwX64) nativeTarget = mingwX64("native")
 else throw new FileNotFoundException("Host OS is not supported in Kotlin/Native")

 nativeTarget.with {
 binaries {
 executable {
 entryPoint = 'main'
 }
 }
 }
}
```

- Targets are defined using `macosX64`, `linuxX64`, and `mingwX64` for macOS, Linux, and Windows. For a complete list of supported platforms, see the [Kotlin Native overview](#).

### 1.5.30 的新特性

- The entry itself defines a series of properties to indicate how the binary is generated and the entry point of the applications. 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, create a library definition `.def` file that contains some information about the necessary headers. In this app, you'll need the `libcurl` library to make some HTTP calls. To create a definition file:

1. Select the `src` folder and create a new directory with **File | New | Directory**.
2. Name new directory `nativeInterop/cinterop`. This is the default convention for header file locations, though it can be overridden in the `build.gradle` 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:

### 1.5.30 的新特性

```
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. You can add multiple files to this entry, separating each with a `\` on a new line. In this case, it's only `curl.h`. The referenced files need to be relative to the folder where the definition file is, or be available on the system 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 fine-tune things.

`headerFilter` is an optional argument and is mostly used when the library is installed as a system library. You don't want to fetch external dependencies (such as system `stdint.h` header) into the interop library. It may be important to optimize the library size and fix potential conflicts between the system and the provided Kotlin/Native compilation environment.

- The next lines are about providing linker and compiler options, which can vary depending on different target platforms. 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.

The convention is that each library gets its definition file, usually with the same name as the library. For more information on all the options available to `cinterop`, see [the Interop section](#).

You need to have the `curl` library binaries on your system to make the sample work. On macOS and Linux, it is usually included. On Windows, you can build it from [sources](#) (you'll need Visual Studio or Windows SDK Commandline 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

## 1.5.30 的新特性

To use header files, make sure they are generated as a part of the build process. For this, add the following entry to the `build.gradle(.kts)` file:

### 【Kotlin】

```
nativeTarget.apply {
 compilations.getByName("main") {
 cinterops {
 val libcurl by creating
 }
 }
 binaries {
 executable {
 entryPoint = "main"
 }
 }
}
```

### 【Groovy】

```
nativeTarget.with {
 compilations.main {
 cinterops {
 libcurl
 }
 }
 binaries {
 executable {
 entryPoint = 'main'
 }
 }
}
```

The new lines are marked with `// NL`. First, `cinterops` is added, and then an entry for each `def` file. By default, the name of the file is used. You can override this with additional parameters:

### 【Kotlin】

```
val libcurl by creating {
 defFile(project.file("src/nativeInterop/cinterop/libcurl.def"))
 packageName("com.jetbrains.handson.http")
 compilerOpts("-I/path")
 includeDirs.allHeaders("path")
}
```

## 1.5.30 的新特性

### 【Groovy】

```
libcurl {
 defFile project.file("src/nativeInterop/cinterop/libcurl.def")
 packageName 'com.jetbrains.handson.http'
 compilerOpts '-I/path'
 includeDirs.allHeaders("path")
}
```

See the [Interoperability with C](#) section for more details on the available options.

## 编写应用程序代码

Now you have the library and the corresponding Kotlin stubs and 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.*

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 following command in the terminal:

```
./gradlew runDebugExecutableNative
```

In this case, the `cinterop` generated part is implicitly included in the build.

2. If there are no errors during compilation, click the green **Run** icon in the gutter beside the `main()` method or use the **Alt+Enter** shortcut to invoke the launch menu in IntelliJ IDEA.

IntelliJ IDEA opens the **Run** tab and shows the output — the contents of

<https://example.com> :

```
<!doctype html>
<html>
<head>
 <title>Example Domain</title>

 <meta charset="utf-8" />
 <meta http-equiv="Content-type" content="text/html; charset=utf-8" />
 <meta name="viewport" content="width=device-width, initial-scale=1" />
 <style type="text/css">
body {
 background-color: #f0f0f2;
 margin: 0;
 padding: 0;
 font-family: -apple-system, system-ui, BlinkMacSystemFont, "Segoe UI", "Open Sans", "Helvetica Neue", Helvetica, Arial, sans-serif;
}

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>
```

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 code [here](#).



## 下一步做什么？

For a complete example of using the `libcurl`, see the [libcurl sample of the Kotlin/Native project](#) that shows how to abstract the code into Kotlin classes as well as display headers. It also demonstrates how to make the steps easier by combining them into a shell script or a Gradle build.

## 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

This document covers some details of Kotlin/Native interoperability with Swift/Objective-C.

## Usage

Kotlin/Native provides bidirectional interoperability with Objective-C. Objective-C frameworks and libraries can be used in Kotlin code if properly imported to the build (system frameworks are imported by default). See [compilation configurations](#) for more details. 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.

Kotlin modules can be used in Swift/Objective-C code if compiled into a framework ([see here for how to declare binaries](#)). See [calculator sample](#) for an example.

## 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.

### 1.5.30 的新特性

Kotlin	Swift	Objective-C	Notes
class	class	@interface	<a href="#">note</a>
interface	protocol	@protocol	
constructor / create	Initializer	Initializer	<a href="#">note</a>
Property	Property	Property	<a href="#">note 1</a> , <a href="#">note 2</a>
Method	Method	Method	<a href="#">note 1</a> , <a href="#">note 2</a>
suspend ->	completionHandler: / async	completionHandler:	<a href="#">note 1</a> , <a href="#">note 2</a>
@Throws fun	throws	error: (NSError**)error	<a href="#">note</a>
Extension	Extension	Category member	<a href="#">note</a>
companion member <-	Class method or property	Class method or property	
null	nil	nil	
Singleton	shared or companion property	shared or companion property	<a href="#">note</a>
Primitive type	Primitive type / NSNumber		<a href="#">note</a>
Unit return type	Void	void	
String	String	NSString	
String	NSMutableString	NSMutableString	<a href="#">note</a>
List	Array	NSArray	
MutableList	NSMutableArray	NSMutableArray	
Set	Set	NSSet	
MutableSet	NSMutableSet	NSMutableSet	<a href="#">note</a>
Map	Dictionary	NSDictionary	
MutableMap	NSMutableDictionary	NSMutableDictionary	<a href="#">note</a>
Function type	Function type	Block pointer type	<a href="#">note</a>
Inline classes	Unsupported	Unsupported	<a href="#">note</a>

## Name translation

### 1.5.30 的新特性

Objective-C classes are imported into Kotlin with their original names. Protocols are imported as interfaces with `Protocol` name suffix, i.e. `@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. Thus, the Kotlin compiler renames Kotlin classes which have the same name but different package in the same framework. This algorithm is not stable yet and can change between Kotlin releases. As a workaround, you can rename the conflicting Kotlin classes in the framework.

## Initializers

Swift/Objective-C initializers are imported to Kotlin as constructors and 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.

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 `setFoo()` method for the property `foo`. 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. E.g.

```
// MyLibraryUtils.kt
package my.library

fun foo() {}
```

can be called from Swift like

### 1.5.30 的新特性

```
MyLibraryUtilsKt.foo()
```

## Method names translation

Generally Swift argument labels and Objective-C selector pieces are mapped to Kotlin parameter names. Anyway 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, e.g.:

```
[player moveTo:LEFT byMeters:17]
[player moveTo:UP byInches:42]
```

in Kotlin it would be:

```
player.moveTo(LEFT, byMeters = 17)
player.moveTo(UP, byInches = 42)
```

## Errors and exceptions

Kotlin has no concept of checked exceptions, all Kotlin exceptions are unchecked. Swift has only checked errors. So if Swift or Objective-C code calls a Kotlin method which throws an exception to be handled, then the Kotlin method should be marked with a `@Throws` annotation specifying a list of "expected" exception classes.

When compiling to Objective-C/Swift framework, non-`suspend` functions having or inheriting `@Throws` annotation are represented as `NSError*`-producing methods in Objective-C and as `throws` methods in Swift. Representations for `suspend` functions always have `NSError* / Error` parameter in completion handler.

When Kotlin function called from Swift/Objective-C code throws an exception which is an instance of one of the `@Throws`-specified classes or their subclasses, it is propagated as `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.

### 1.5.30 的新特性

Note that the opposite reversed translation is not implemented yet: Swift/Objective-C error-throwing methods aren't imported to Kotlin as exception-throwing.

## 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 Swift](#).

## Extensions and category members

Members of Objective-C categories and Swift extensions are 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.

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

## 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.



## NSNumber

Kotlin primitive type boxes are mapped to special Swift/Objective-C classes. For example, `kotlin.Int` box is represented as `KotlinInt` class instance in Swift (or `NSString` instance in Objective-C, where `prefix` is the framework names prefix). These classes are derived from `NSNumber`, so the instances are proper `NSNumber`s supporting all corresponding operations.

`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 `NSNumber` type doesn't provide enough information about a wrapped primitive value type, i.e.

### 1.5.30 的新特性

`NSNumber` is statically not known to be a e.g. `Byte`, `Boolean`, or `Double`. So Kotlin primitive values should be cast to/from `NSNumber` manually (see [below](#)).

## NSMutableString

`NSMutableString` Objective-C class is not available from Kotlin. All instances of `NSMutableString` are copied when passed to Kotlin.

## Collections

Kotlin collections are converted to Swift/Objective-C collections as described in the table above. Swift/Objective-C collections are mapped to Kotlin in the same way, except for `NSMutableSet` and `NSMutableDictionary`. `NSMutableSet` isn't converted to a Kotlin `MutableSet`. To pass an object for Kotlin `MutableSet`, you can create this kind of Kotlin collection explicitly by either creating it in Kotlin with e.g. `mutableSetOf()`, or using the `KotlinMutableSet` class in Swift (or  `${prefix}MutableSet` in Objective-C, where `prefix` is the framework names prefix). The same holds for `MutableMap`.

## Function types

Kotlin function-typed objects (e.g. lambdas) are converted to Swift functions / Objective-C blocks. 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 in the same way as it is for any other Kotlin object (see singletons in the table above). To sum the things up:

```
fun foo(block: (Int) -> Unit) { ... }
```

would be represented in Swift as

```
func foo(block: (KotlinInt) -> KotlinUnit)
```

and can be called like

### 1.5.30 的新特性

```
foo {
 bar($0 as! Int32)
 return KotlinUnit()
}
```

## Generics

Objective-C supports "lightweight generics" defined on 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.

## 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. As such, the following:

```
class Sample<T>() {
 fun myVal(): T
}
```

will (logically) look like this:

```
class Sample<T>() {
 fun myVal(): T?
}
```

In order to support a potentially nullable type, the Objective-C header needs to define `myVal` with a nullable return value.

### 1.5.30 的新特性

To mitigate this, when defining your generic classes, if the generic type should *never* be null, provide a non-null type constraint:

```
class Sample<T : Any>() {
 fun myVal(): T
}
```

That will force the Objective-C header to mark `myVal` as non-null.

## 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-null upper bound will make Objective-C methods/properties non-null.

## To disable

To have the framework header written without generics, add the flag to the compiler config:

```
binaries.framework {
 freeCompilerArgs += "-Xno-objc-generics"
}
```

## Casting between mapped types

### 1.5.30 的新特性

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, e.g.

```
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, e.g. 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.

### 1.5.30 的新特性

To override different methods with clashing Kotlin signatures, you can add a `@Suppress("CONFLICTING_OVERLOADS")` annotation to the class.

By default the Kotlin/Native compiler doesn't allow calling a non-designated Objective-C initializer as a `super(...)` constructor. This behaviour can be inconvenient if the designated initializers aren't marked properly in the Objective-C library. Adding a `disableDesignatedInitializerChecks = true` to the `.def` file for this library would disable these compiler checks.

## 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 Kotlin programming language are not yet mapped into respective features of Objective-C or Swift. Currently, 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

Kotlin/Native 提供与 Objective-C/Swift 的双向互操作性。 Objective-C framework 与库可以在 Kotlin 代码中使用。 Kotlin 模块同样可以在 Swift/Objective-C 代码中使用。除此之外，Kotlin/Native 也拥有 [C 互操作性](#)。这篇 [Kotlin/Native 开发动态库](#) 教程包含了更多信息。

在本教程中将看到如何在 Objective-C 与 Swift 编写的 macOS 与 iOS 应用程序中使用 Kotlin/Native 代码。

在本教程中将：

- 创建一个 [Kotlin 库](#) 并将它编译为 framework
- 检查生成的 [Objective-C 与 Swift API](#) 代码
- 在 [Objective-C 与 Swift](#) 中使用 framework
- 为 [macOS 与 iOS 配置 Xcode](#) 以使用 framework

## 创建一个 Kotlin 库

The Kotlin/Native 编译器可以使 Kotlin 代码为 macOS 与 iOS 生产一个 framework 的输出。生成的 framework 包含在 Objective-C 与 Swift 中所有使用所需的声明与二进制文件。理解这项技术的最佳方式是自己进行一下尝试。我们首先来创建一个小型的 Kotlin 库，并在 Objective-C 程序中使用它。

创建 `hello.kt` 文件，并在其中编写库的内容：

## 1.5.30 的新特性

```
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!" }
```

While it is possible to use the command line, either directly or by combining it with a script file (such as `.sh` or `.bat` file), this approach doesn't scale well for big projects that have hundreds of files and libraries. It is therefore better to use the Kotlin/Native compiler with a build system, as it helps to download and cache the Kotlin/Native compiler binaries and libraries with transitive dependencies and run the compiler and tests. Kotlin/Native can use the [Gradle](#) build system through the [kotlin-multiplatform](#) plugin.

We covered the basics of setting up an IDE compatible project with Gradle in the [A Basic Kotlin/Native Application](#) tutorial. Please check it out if you are looking for detailed first steps and instructions on how to start a new Kotlin/Native project and open it in IntelliJ IDEA. In this tutorial, we'll look at the advanced C interop related usages of Kotlin/Native and [multiplatform](#) builds with Gradle.

First, create a project folder. All the paths in this tutorial will be relative to this folder. Sometimes the missing directories will have to be created before any new files can be added.

Use the following `build.gradle(.kts)` Gradle build file:

【Kotlin】

## 1.5.30 的新特性

```
plugins {
 kotlin("multiplatform") version "1.6.10"
}

repositories {
 mavenCentral()
}

kotlin {
 macosX64("native") {
 binaries {
 framework {
 basePath = "Demo"
 }
 }
 }
}

tasks.wrapper {
 gradleVersion = "6.7.1"
 distributionType = Wrapper.DistributionType.ALL
}
```

## 【Groovy】

```
plugins {
 id 'org.jetbrains.kotlin.multiplatform' version '1.6.10'
}

repositories {
 mavenCentral()
}

kotlin {
 macosX64("native") {
 binaries {
 framework {
 basePath = "Demo"
 }
 }
 }
}

wrapper {
 gradleVersion = "6.7.1"
 distributionType = "ALL"
}
```

### 1.5.30 的新特性

将源文件移动到项目下的 `src/nativeMain/kotlin` 文件夹。当使用 [kotlin-多平台](#) 插件的时候这是定位文件的默认路径。使用插件。使用以下代码块来配置项目生成动态或共享库：

```
binaries {
 framework {
 baseName = "Demo"
 }
}
```

Along with macOS `x64`, Kotlin/Native supports macOS `arm64` and iOS `arm32`, `arm64` and `x64` targets. You may replace the `macosX64` with respective functions as shown in the table:

目标 平台/设备	Gradle 函数
macOS x86_64	<code>macosX64()</code>
macOS ARM 64	<code>macosArm64()</code>
iOS ARM 32	<code>iosArm32()</code>
iOS ARM 64	<code>iosArm64()</code>
iOS Simulator (x86_64)	<code>iosX64()</code>

运行 `linkNative` Gradle 任务以在 IDE 中构建该库，或者使用如下的控制台命令：

```
./gradlew linkNative
```

依据配置的不同，构建生成的 framework 位于 `build/bin/native/debugFramework` 或 `build/bin/native/releaseFramework` 文件夹。我们来看看里面是什么。

## 生成的 Framework 头文件

每个创建的 framework 头文件都包含在 `<Framework>/Headers/Demo.h` 中。这个头文件不依赖目标平台（至少需要 Kotlin/Native v.0.9.2）。它包含我们的 Kotlin 代码的定义与一些 Kotlin 级的声明。

Kotlin/Native 导出符号的方式如有变更，恕不另行通知。



## Kotlin/Native 运行时声明

### 1.5.30 的新特性

来看看 Kotlin 的运行时声明：

```
NS_ASSUME_NONNULL_BEGIN

@interface KotlinBase : NSObject
- (instancetype)init __attribute__((unavailable));
+ (instancetype)new __attribute__((unavailable));
+ (void)initialize __attribute__((objc_reQUIRES_SUPER));
@end;

@interface KotlinBase (KotlinBaseCopying) <NSCopying>
@end;

__attribute__((objc_runtime_name("KotlinMutableSet")))
__attribute__((swift_name("KotlinMutableSet")))
@interface DemoMutableSet<ObjectType> : NSMutableSet<ObjectType>
@end;

__attribute__((objc_runtime_name("KotlinMutableDictionary")))
__attribute__((swift_name("KotlinMutableDictionary")))
@interface DemoMutableDictionary<KeyType, ObjectType> : NSMutableDictionary<KeyType
@end;

@interface NSError (NSErrorKotlinException)
@property (readonly) id _Nullable kotlinException;
@end;
```

Kotlin 类在 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从另一边映射到了相似的集合类型：

	<b>Kotlin</b>	<b>Swift</b>	<b>Objective-C</b>
	List	Array	NSArray
	MutableList	NSMutableArray	NSMutableArray
	Set	Set	NSSet
	Map	Dictionary	NSDictionary
	MutableMap	NSMutableDictionary	NSMutableDictionary

## Kotlin 数值与 NSNumber

下一步，`<Framework>/Headers/Demo.h` 包含了 Kotlin/Native 数字类型与 `NSNumber` 之间的映射。在 Objective-C 中拥有一个基类名为 `DemoNumber`，而在 Swift 中是 `KotlinNumber`。它继承自 `NSNumber`。这里有每个作为子类的 Kotlin 数字类型：

### 1.5.30 的新特性

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，用于从相关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，声明看起来像这样：

```
__attribute__((objc_runtime_name("Kotlin__TYPE__")))
__attribute__((swift_name("Kotlin__TYPE__")))
@interface Demo__TYPE__ : DemoNumber
- (instancetype)initWith__TYPE__:(__CTYPE__)value;
+ (instancetype)numberWith__TYPE__:(__CTYPE__)value;
@end;
```

其中 `_TYPE_` 是简单类型的名称之一，而 `_CTYPE_` 是相关的 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 Objective-C 与 Swift。在 Swift 中，可以简单的调用构造函数来创建一个示例，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 Objective-C 与 Swift。生成的 `<Framework>/Headers/Demo.h` 文件包含 `Class`、`Interface` 与 `Object` 的确切定义：

### 1.5.30 的新特性

```
NS_ASSUME_NONNULL_BEGIN

__attribute__((objc_subclassing_restricted))
__attribute__((swift_name("Object")))
@interface DemoObject : KotlinBase
+ (instancetype)alloc __attribute__((unavailable));
+ (instancetype)allocWithZone:(struct _NSZone *)zone __attribute__((unavailable));
+ (instancetype)object __attribute__((swift_name("init())));
@property (readonly) NSString *field;
@end;

__attribute__((swift_name("Interface")))
@protocol DemoInterface
@required
- (void)iMember __attribute__((swift_name("iMember())));
@end;

__attribute__((objc_subclassing_restricted))
__attribute__((swift_name("Clazz")))
@interface DemoClazz : KotlinBase <DemoInterface>
- (instancetype)init __attribute__((swift_name("init())))
__attribute__((objc_designated_initializer));
+ (instancetype)new __attribute__((availability(swift, unavailable, message="use ob
- (DemoLong * _Nullable)memberP:(int32_t)p __attribute__((swift_name("member(p:)))
@end;
```

这段代码有各种 Objective-C attribute，旨在提供在 Objective-C 与 Swift 语言中使用该 framework 的帮助。 DemoClazz 、 DemoInterface 、 DemoObject 被分别创建为 Clazz 、 Interface 与 Object 。 Interface 被转换为 @protocol ，同样 class 与 object 都以 @interface 表示。 Demo 前缀来自于 kotlinc-native 编译器的 -output 参数与 framework 的名称。可以看到这里的可空的返回值类型 ULONG? 被转换到 Objective-C 中的 DemoLong\* 。

## Kotlin 中的全局声明

在 Demo 作为 framework 名称的地方且为 kotlinc-native 设置了 -output 参数时，所有 Kotlin 中的全局声明都被转化为 Objective-C 中的 DemoLibKt 以及 Swift 中的 LibKt 。

### 1.5.30 的新特性

```
NS_ASSUME_NONNULL_BEGIN

__attribute__((objc_subclassing_restricted))
__attribute__((swift_name("LibKt")))
@interface DemoLibKt : KotlinBase
+ (void)forIntegersB:(int8_t)b s:(int16_t)s i:(int32_t)i l:(DemoLong * _Nullable)l
+ (void)forFloatsF:(float)f d:(DemoDouble * _Nullable)d __attribute__((swift_name("
+ (NSString *)stringsStr:(NSString * _Nullable)str __attribute__((swift_name("strin
+ (NSString * _Nullable)acceptFunF:(NSString * _Nullable (^)(NSString *))f __attrib
+ (NSString * _Nullable (^)(NSString *))supplyFun __attribute__((swift_name("supply
@end;
```

可以看到 Kotlin `String` 与 Objective-C `NSString *` 是透明映射的。类似地，Kotlin 的 `Unit` 类型被映射到 `void`。我们看到原生类型直接映射。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，如上表所示。包括高阶函数 `acceptFunF` 与 `supplyFun`，都接收一个 Objective-C 块。

更多的关于所有其它类型的映射细节可以在这篇 [Objective-C 互操作](#) 文档中找到。

## 垃圾回收与引用计数

Objective-C 与 Swift 使用引用计数。Kotlin/Native 也同样拥有自己的垃圾回收。Kotlin/Native 的垃圾回收与 Objective-C/Swift 引用计数相集成。不需要在 Swift 或 Objective-C 中使用任何其它特别的方式去控制 Kotlin/Native 实例的生命周期。

## 在 Objective-C 中使用代码

让我们在 Objective-C 中调用代码。为此，使用下面的内容创建 `main.m` 文件：

### 1.5.30 的新特性

```
#import <Foundation/Foundation.h>
#import <Demo/Demo.h>

int main(int argc, const char * argv[]) {
 @autoreleasepool {
 [[DemoObject object] 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;
 }
}
```

这里直接在 Objective-C 代码中调用 Kotlin 类。一个 Kotlin `object` 拥有类方法函数 `object`，这使我们在想获取对象的唯一实例时调用 `object` 方法就可以了。广泛使用的用于创建 `Clazz` 类实例的模式。在 Objective-C 上调用 `[[ DemoClazz alloc] init]`。我们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。所有的方法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，可以给它直接传递 `NSString`。而返回值也同样可以看作 `NSString`。

## 在 Swift 中使用代码

这个使用 Kotlin/Native 编译的 framework 拥有辅助 attribute 来使它在 Swift 中的使用更为容易。将之前的 Objective-C 示例覆盖为 Swift。其结果是，将在 `main.swift` 中包含下面的代码：

### 1.5.30 的新特性

```
import Foundation
import Demo

let kotlinObject = Object()
assert(kotlinObject === Object(), "Kotlin object has only one instance")

let field = Object().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!)
}
```

这段 Kotlin 代码被转换为看起来非常相似的 Swift 代码。但是它们有一些小差异。在 Kotlin 中任何 `object` 只拥有一个实例。Kotlin `object Object` 现在在 Swift 中拥有一个构造函数，我们使用 `Object()` 语法来访问它唯一的实例。在 Swift 中该实例总是相同的，所以 `Object() == Object()` 为 true。方法与属性名称按原样转换。即 Kotlin `String` 被转换为 Swift `String`。Swift 同样隐藏了 `NSNumber*` 的装箱。我们可以传一个 Swift 闭包给 Kotlin，与在 Swift 中调用 Kotlin lambda 函数是相同的。

更多关于类型映射的信息可以在这篇 [Objective-C 互操作](#) 文档中找到。

## Xcode 与 Framework 依赖

需要配置一个 Xcode 项目来使用我们的 framework。这个配置依赖于目标平台。

### Xcode 用于 macOS 目标平台

First, in the **General** tab of the **target** configuration, under the **Linked Frameworks and Libraries** section, you need to include our framework. This will make Xcode look at our framework and resolve imports both from Objective-C and Swift.

第二步是配置 framework 生产的二进制文件的搜索路径。它也被称作 `rpath` 或[运行时搜索路径](#)。二进制文件使用路径来查找所需的 framework。我们不推荐，在不需要的时候在操作系统中去安装其它 framework。应该了解未来应用程序的布局，举例来说，

### 1.5.30 的新特性

可能在应用程序包下有 `Frameworks` 文件夹，其中包含所使用的所有 framework。  
`@rpath` 参数可以被配置到 Xcode。需要打开 **project** 配置项并找到 **Runpath Search Paths** 选项。在这里指定编译 framework 的相对路径。

## Xcode 用于 iOS 目标平台

First, you need to include the compiled framework in the Xcode project. To do this, add the framework to the **Frameworks, Libraries, and Embedded Content** section of the **General** tab of the **target** configuration page.

The second step is to then include the framework path into the **Framework Search Paths** section of the **Build Settings** tab of the **target** configuration page. It is possible to use the `$(PROJECT_DIR)` macro to simplify the setup.

iOS 模拟器需要一个为 `ios_x64` 目标平台编译的 framework，它位于我们案例中的 `ios_sim` 文件夹。

这个 [Stackoverflow 主题](#) 包含了一些更多的建议。同样，[CocoaPods](#) 包管理器也可能有助于自动化该过程。

## 接下来

Kotlin/Native 与 Objective-C 以及 Swift 语言之间拥有双向互操作性。Kotlin 对象集成了 Objective-C/Swift 的引用计数。Kotlin 对象可以被自动释放。这篇 [Objective-C 互操作](#) 文档包含了更多关于互操作实现细节的信息。当然，也可以导入一个外部的 framework 并在 Kotlin 中使用它。Kotlin/Native 附带一套良好的预导入系统 framework。

Kotlin/Native 同样支持 C 互操作。查看这篇 [Kotlin/Native 开发动态库](#) 教程。

## CocoaPods 集成

- CocoaPods 概述
- 添加对 Pod 库的依赖
- 使用 Kotlin Gradle 项目作为 CocoaPods 依赖项

## CocoaPods 概述与设置

Kotlin/Native provides integration with the [CocoaPods dependency manager](#). You can add dependencies on Pod libraries as well as use a multiplatform project with native targets as a CocoaPods dependency.

You can manage Pod dependencies directly in IntelliJ IDEA and enjoy all the additional features such as code highlighting and completion. You can build the whole Kotlin project with Gradle and not ever have to switch to Xcode.

Use Xcode only when you need to write Swift/Objective-C code or run your application on a simulator or device. To work correctly with Xcode, you should [update your Podfile](#).

Depending on your project and purposes, you can add dependencies between [a Kotlin project and a Pod library](#) as well as [a Kotlin Gradle project and an Xcode project](#).

## Set up the environment to work with CocoaPods

1. Install the [CocoaPods dependency manager](#):

```
$ sudo gem install cocoapods
```

2. Install the [cocoapods-generate](#) plugin:

```
$ sudo gem install cocoapods-generate
```

If you encounter any problems during the installation, follow the [official CocoaPods installation guide](#)



## Add and configure Kotlin CocoaPods Gradle plugin

## 1.5.30 的新特性

1. In `build.gradle(.kts)` of your project, apply the CocoaPods plugin as well as the Kotlin Multiplatform plugin:

```
plugins {
 kotlin("multiplatform") version "1.6.10"
 kotlin("native.cocoapods") version "1.6.10"
}
```

2. Configure `summary`, `homepage`, and `frameworkName` of the `Podspec` file in the `cocoapods` block.

`version` is a version of the Gradle project:

```
plugins {
 kotlin("multiplatform") version "1.6.10"
 kotlin("native.cocoapods") version "1.6.10"
}

// CocoaPods requires the podspec to have a version.
version = "1.0"

kotlin {
 cocoapods {

 framework {
 // Mandatory properties
 // Configure fields required by CocoaPods.
 summary = "Some description for a Kotlin/Native module"
 homepage = "Link to a Kotlin/Native module homepage"
 // Framework name configuration. Use this property instead of dep
 baseName = "MyFramework"

 // Optional properties
 // (Optional) Dynamic framework support
 isStatic = false
 // (Optional) Dependency export
 export(project(":anotherKMMModule"))
 transitiveExport = false // This is default.
 // (Optional) Bitcode embedding
 embedBitcode(BITCODE)
 }

 // Maps custom Xcode configuration to NativeBuildType
 xcodeConfigurationToNativeBuildType["CUSTOM_DEBUG"] = NativeBuildType.
 xcodeConfigurationToNativeBuildType["CUSTOM_RELEASE"] = NativeBuildTyp
 }
}
```

### 1.5.30 的新特性

See the full syntax of Kotlin DSL in the [Kotlin Gradle plugin repository](#).



3. Re-import the project.
4. Generate the [Gradle wrapper](#) to avoid compatibility issues during an Xcode build.

When applied, the CocoaPods plugin does the following:

- Adds both `debug` and `release` frameworks as output binaries for all macOS, iOS, tvOS, and watchOS targets.
- Creates a `podspec` task which generates a [Podspec](#) file for the project.

The `Podspec` file includes a path to an output framework and script phases that automate building this framework during the build process of an Xcode project.

## Update Podfile for Xcode

If you want to import your Kotlin project in an Xcode project, you need to make some changes to your Podfile:

- If your project has any Git, HTTP, or custom Podspec repository dependencies, you should also specify the path to the Podspec in the Podfile.

For example, if you add a dependency on `podspecWithFilesExample`, declare the path to the Podspec in the Podfile:

```
target 'ios-app' do
 # ... other dependencies ...
 pod 'podspecWithFilesExample', :path => 'cocoapods/externalSources/url/po
end
```

The `:path` should contain the filepath to the Pod.

- When you add a library from the custom Podspec repository, you should also specify the [location](#) of specs at the beginning of your Podfile:

```
source 'https://github.com/Kotlin/kotlin-cocoapods-spec.git'

target 'kotlin-cocoapods-xcproj' do
 # ... other dependencies ...
 pod 'example'
end
```

## 1.5.30 的新特性

Re-import the project after making changes in the Podfile.



If you don't make these changes to the Podfile, the `podInstall` task will fail, and the CocoaPods plugin will show an error message in the log.

Check out the `withXcproject` branch of the [sample project](#), which contains an example of Xcode integration with an existing Xcode project named `kotlin-cocoapods-xcproj`.

## 添加对 Pod 库的依赖

To add dependencies between a Kotlin project and a Pod library, you should [complete the initial configuration](#). This allows you to add dependencies on different types of Pod libraries.

When you add a new dependency and re-import the project in IntelliJ IDEA, the new dependency will be added automatically. No additional steps are required.

To use your Kotlin project with Xcode, you should [make changes in your project Podfile](#).

A Kotlin project requires the `pod()` function call in `build.gradle.kts` (`build.gradle`) for adding a Pod dependency. Each dependency requires its separate function call. You can specify the parameters for the dependency in the configuration block of the function.

If you don't specify the minimum deployment target version and a dependency Pod requires a higher deployment target, you will get an error.



You can find a sample project [here](#).

## From the CocoaPods repository

1. Specify the name of a Pod library in the `pod()` function.

In the configuration block, you can specify the version of the library using the `version` parameter. To use the latest version of the library, you can just omit this parameter altogether.

You can add dependencies on subspecs.



2. Specify the minimum deployment target version for the Pod library.

### 1.5.30 的新特性

```
kotlin {
 ios()

 cocoapods {
 ios.deploymentTarget = "13.5"

 summary = "CocoaPods test library"
 homepage = "https://github.com/JetBrains/kotlin"

 pod("AFNetworking") {
 version = "~> 4.0.1"
 }
 }
}
```

### 3. Re-import the project.

To use these dependencies from the Kotlin code, import the packages `cocoapods.<library-name>`.

```
import cocoapods.AFNetworking.*
```

## On a locally stored library

### 1. Specify the name of a Pod library in the `pod()` function.

In the configuration block, specify the path to the local Pod library: use the `path()` function in the `source` parameter value.

You can add local dependencies on subspecs as well. The `cocoapods` block can include dependencies to Pods stored locally and Pods from the CocoaPods repository at the same time.



### 2. Specify the minimum deployment target version for the Pod library.

### 1.5.30 的新特性

```
kotlin {
 ios()

 cocoapods {
 summary = "CocoaPods test library"
 homepage = "https://github.com/JetBrains/kotlin"

 ios.deploymentTarget = "13.5"

 pod("pod_dependency") {
 version = "1.0"
 source = path(project.file("../pod_dependency"))
 }
 pod("subspec_dependency/Core") {
 version = "1.0"
 source = path(project.file("../subspec_dependency"))
 }
 pod("AFNetworking") {
 version = "~> 4.0.1"
 }
 }
}
```

You can also specify the version of the library using `version` parameter in the configuration block. To use the latest version of the library, omit the `version` parameter.



### 3. Re-import the project.

To use these dependencies from the Kotlin code, import the packages `cocoapods.`

```
<library-name> .
```

```
import cocoapods.pod_dependency.*
import cocoapods.subspec_dependency.*
import cocoapods.AFNetworking.*
```

## From a custom Git repository

### 1. Specify the name of a Pod library in the `pod()` function.

In the configuration block, specify the path to the git repository: use the `git()` function in the `source` parameter value.

## 1.5.30 的新特性

Additionally, you can specify the following parameters in the block after `git()`:

- `commit` – to use a specific commit from the repository
- `tag` – to use a specific tag from the repository
- `branch` – to use a specific branch from the repository

The `git()` function prioritizes passed parameters in the following order: `commit`, `tag`, `branch`. If you don't specify a parameter, the Kotlin plugin uses `HEAD` from the `master` branch.

You can combine `branch`, `commit`, and `tag` parameters to get the specific version of a Pod.



## 2. Specify the minimum deployment target version for the Pod library.

```
kotlin {
 ios()

 cocoapods {
 summary = "CocoaPods test library"
 homepage = "https://github.com/JetBrains/kotlin"

 ios.deploymentTarget = "13.5"

 pod("AFNetworking") {
 source = git("https://github.com/AFNetworking/AFNetworking") {
 tag = "4.0.0"
 }
 }

 pod("JSONModel") {
 source = git("https://github.com/jsonmodel/jsonmodel.git") {
 branch = "key-mapper-class"
 }
 }

 pod("CocoaLumberjack") {
 source = git("https://github.com/CocoaLumberjack/CocoaLumberjack.git") {
 commit = "3e7f595e3a459c39b917aacf9856cd2a48c4dbf3"
 }
 }
 }
}
```

## 3. Re-import the project.

### 1.5.30 的新特性

To work correctly with Xcode, you should specify the path to the Podspec in your Podfile. For example:

```
target 'ios-app' do
 # ... other pod dependencies ...
 pod 'JSONModel', :path => '../cocoapods/kmm-with-cocoapods-sample/kotlin-li
end
```



To use these dependencies from the Kotlin code, import the packages `cocoapods.`

`<library-name> .`

```
import cocoapods.AFNetworking.*
import cocoapods.JSONModel.*
import cocoapods.CocoaLumberjack.*
```

## From a zip, tar, or jar archive

1. Specify the name of a Pod library in the `pod()` function.

In the configuration block, specify the path to the archive: use the `url()` function with an arbitrary HTTP address in the `source` parameter value.

Additionally, you can specify the boolean `flatten` parameter as a second argument for the `url()` function. This parameter indicates that all the Pod files are located in the root directory of the archive.

2. Specify the minimum deployment target version for the Pod library.

### 1.5.30 的新特性

```
kotlin {
 ios()

 cocoapods {
 summary = "CocoaPods test library"
 homepage = "https://github.com/JetBrains/kotlin"

 ios.deploymentTarget = "13.5"

 pod("pod_dependency") {
 source = url("https://github.com/Kotlin/kmm-with-cocoapods-sample")
 }
 }
}
```

### 3. Re-import the project.

To work correctly with Xcode, you should specify the path to the Podspec in your Podfile. For example:

```
target 'ios-app' do
 # ... other pod dependencies ...
 pod 'podspecWithFilesExample', :path => '../cocoapods/kmm-with-cocoapods-sample'
end
```



To use these dependencies from the Kotlin code, import the packages `cocoapods.`

`<library-name> .`

```
import cocoapods.pod_dependency.*
```

## From a custom Podspec repository

1. Specify the HTTP address to the custom Podspec repository using the `url()` inside the `specRepos` block.
2. Specify the name of a Pod library in the `pod()` function.
3. Specify the minimum deployment target version for the Pod library.

## 1.5.30 的新特性

```
kotlin {
 ios()

 cocoapods {
 summary = "CocoaPods test library"
 homepage = "https://github.com/JetBrains/kotlin"

 ios.deploymentTarget = "13.5"

 specRepos {
 url("https://github.com/Kotlin/kotlin-cocoapods-spec.git")
 }
 pod("example")
 }
}
```

### 4. Re-import the project.

To work correctly with Xcode, you should specify the location of specs at the beginning of your Podfile. For example,

```
source 'https://github.com/Kotlin/kotlin-cocoapods-spec.git'
```

You should also specify the path to the Podspec in your Podfile. For example:

```
target 'ios-app' do
 # ... other pod dependencies ...
 pod 'podspecWithFilesExample', :path => '../cocoapods/kmm-with-cocoapods-sai'
end
```



To use these dependencies from the Kotlin code, import the packages `cocoapods.`

```
<library-name> .
```

```
import cocoapods.example.*
```

## With custom cinterop options

### 1. Specify the name of a Pod library in the `pod()` function.

In the configuration block, specify the cinterop options:

### 1.5.30 的新特性

- `extra0pts` – to specify the list of options for a Pod library. For example, specific flags: `extra0pts = listOf("-compiler-option")`
- `packageName` – to specify the package name. If you specify this, you can import the library using the package name: `import <packageName>`.

2. Specify the minimum deployment target version for the Pod library.

```
kotlin {
 ios()

 cocoapods {
 summary = "CocoaPods test library"
 homepage = "https://github.com/JetBrains/kotlin"

 ios.deploymentTarget = "13.5"

 useLibraries()

 pod("YandexMapKit") {
 packageName = "YandexMK"
 }
 }
}
```

3. Re-import the project.

To use these dependencies from the Kotlin code, import the packages `cocoapods.`

`<library-name> .`

```
import cocoapods.YandexMapKit.*
```

If you use the `packageName` parameter, you can import the library using the package name `import <packageName>`:

```
import YandexMK.YMKPoint
import YandexMK.YMKDistance
```

## On a static Pod library

1. Specify the name of the library using the `pod()` function.
2. Call the `useLibraries()` function – it enables a special flag for static libraries.

## 1.5.30 的新特性

### 3. Specify the minimum deployment target version for the Pod library.

```
kotlin {
 ios()

 cocoapods {
 summary = "CocoaPods test library"
 homepage = "https://github.com/JetBrains/kotlin"

 ios.deploymentTarget = "13.5"

 pod("YandexMapKit") {
 version = "~> 3.2"
 }
 useLibraries()
 }
}
```

### 4. Re-import the project.

To use these dependencies from the Kotlin code, import the packages `cocoapods.`

`<library-name> .`

```
import cocoapods.YandexMapKit.*
```

# 使用 Kotlin Gradle 项目作为 CocoaPods 依赖项

You can use a Kotlin Multiplatform project with native targets as a CocoaPods dependency. You can include such a dependency in the Podfile of the Xcode project by its name and path to the project directory containing the generated Podspec.

This dependency will be automatically built (and rebuilt) along with this project. Such an approach simplifies importing to Xcode by removing a need to write the corresponding Gradle tasks and Xcode build steps manually.

You can add dependencies between a Kotlin Gradle project and an Xcode project with one or several targets. It's also possible to add dependencies between a Gradle project and multiple Xcode projects. However, in this case, you need to add a dependency by calling `pod install` manually for each Xcode project. In other cases, it's done automatically.

- To correctly import the dependencies into the Kotlin/Native module, the `Podfile` must contain either `use_modular_headers!` or `use_frameworks!` directive.
- If you don't specify the minimum deployment target version and a dependency Pod requires a higher deployment target, you will get an error.



## Xcode project with one target

1. Create an Xcode project with a `Podfile` if you haven't done so yet.
2. Add the path to your Xcode project `Podfile` with `podfile = project.file(..)` to `build.gradle.kts` (`build.gradle`) of your Kotlin project. This step helps synchronize your Xcode project with Gradle project dependencies by calling `pod install` for your `Podfile`.
3. Specify the minimum target version for the Pod library.

### 1.5.30 的新特性

```
kotlin {
 ios()

 cocoapods {
 summary = "CocoaPods test library"
 homepage = "https://github.com/JetBrains/kotlin"
 ios.deploymentTarget = "13.5"
 pod("AFNetworking") {
 version = "~> 4.0.0"
 }
 podfile = project.file("../ios-app/Podfile")
 }
}
```

4. Add the name and path of the Gradle project you want to include in the Xcode project to `Podfile`.

```
use_frameworks!

platform :ios, '13.5'

target 'ios-app' do
 pod 'kotlin_library', :path => '../kotlin-library'
end
```

5. Re-import the project.

## Xcode project with several targets

1. Create an Xcode project with a `Podfile` if you haven't done so yet.
2. Add the path to your Xcode project `Podfile` with `podfile = project.file(..)` to `build.gradle.kts` (`build.gradle`) of your Kotlin project. This step helps synchronize your Xcode project with Gradle project dependencies by calling `pod install` for your `Podfile`.
3. Add dependencies to the Pod libraries you want to use in your project with `pod()`.
4. For each target, specify the minimum target version for the Pod library.

### 1.5.30 的新特性

```
kotlin {
 ios()
 tvos()

 cocoapods {
 summary = "CocoaPods test library"
 homepage = "https://github.com/JetBrains/kotlin"
 ios.deploymentTarget = "13.5"
 tvos.deploymentTarget = "13.4"

 pod("AFNetworking") {
 version = "~> 4.0.0"
 }
 podfile = project.file("../severalTargetsXcodeProject/Podfile") // specify
 }
}
```

5. Add the name and path of the Gradle project you want to include in the Xcode project to the `Podfile`.

```
target 'iosApp' do
 use_frameworks!
 platform :ios, '13.5'
 # Pods for iosApp
 pod 'kotlin_library', :path => '../kotlin-library'
end

target 'TVosApp' do
 use_frameworks!
 platform :tvos, '13.4'

 # Pods for TVosApp
 pod 'kotlin_library', :path => '../kotlin-library'
end
```

6. Re-import the project.

You can find a sample project [here](#).

## 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-o
```

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:

### 1.5.30 的新特性

```
$ 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：

### 1.5.30 的新特性

```
$ 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。

### 1.5.30 的新特性

- 安装在默认存储库（目前默认为 `~/.konan`，不过可以通过设置 `KONAN_DATA_DIR` 来更改）中的库。
- 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/stdc` 目录中找到示例布局。

# 平台库

为了提供对用户原生操作系统服务的访问，Kotlin/Native 发行版包含了一组为每个目标平台预构建的库。我们称之为**平台库**。

## POSIX 绑定

对于所有基于 Unix 或 Windows 的目标平台（包括 Android 与 iOS 目标平台），我们提供了 `posix` 平台库。它包含对 [POSIX 标准](#) 的平台实现的绑定。

使用该库只需导入之：

```
import platform.posix.*
```

唯一不可用的目标平台是 [WebAssembly](#)。

请注意，`platform.posix` 的内容在不同平台上并不相同，就像不同的 POSIX 实现一样略有不同。

## 热门原生库

还有很多平台库可用于所在主机以及交叉编译目标。Kotlin/Native 发行版可以在适用的平台上访问 OpenGL、zlib 以及其他热门原生库。

在苹果平台上提供了 `objc` 库，用来与 [Objective-C](#) 进行互操作。

详细信息请核查发行版的 `dist/klib/platform/$target` 的内容。

## 默认可用

来自平台库的包都默认可用。使用时无需指定特殊的链接标志。Kotlin/Native 编译器会自动检测访问了哪些平台库，并自动链接所需的库。

另一方面，发行版中的平台库仅仅是对原生库的包装与绑定。这意味着计算机上需要已经安装了原生库自身（`.so`、`.a`、`.dylib`、`.dll` 等）。

## 示例

### 1.5.30 的新特性

Kotlin/Native 版本库中提供了大量的示例演示平台库的使用。详见[样例](#)。

# Kotlin/Native 开发动态库——教程

学校如何使用位于已经存在的原生应用程序或库中的代码。为此，需要将 Kotlin 代码编译为动态库，例如 `.so`、`.dylib` 以及 `.dll`。

Kotlin/Native 也可以与 Apple 技术紧密集成。这篇 [Kotlin/Native 开发 Apple Framework 教程](#) 解释了如何将代码编译为 Swift 或 Objective-C framework。

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
- 将示例代码编译并运行于 Linux 与 Mac 以及 Windows

## 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头文件，即 `.h` 文件，会通过它来调用编译后的 C 代码。

理解这些技术的最佳方法是尝试它们。首先创建第一个小型的 Kotlin 库，并在 C 程序中使用它。

先在 Kotlin 中创建一个库文件，然后将其保存为 `hello.kt`：

## 1.5.30 的新特性

```
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"
```

While it is possible to use the command line, either directly or by combining it with a script file (such as `.sh` or `.bat` file), this approach doesn't scale well for big projects that have hundreds of files and libraries. It is then better to use the Kotlin/Native compiler with a build system, as it helps to download and cache the Kotlin/Native compiler binaries and libraries with transitive dependencies and run the compiler and tests. Kotlin/Native can use the [Gradle](#) build system through the [kotlin-multiplatform](#) plugin.

We covered the basics of setting up an IDE compatible project with Gradle in the [A Basic Kotlin/Native Application](#) tutorial. Please check it out if you are looking for detailed first steps and instructions on how to start a new Kotlin/Native project and open it in IntelliJ IDEA. In this tutorial, we'll look at the advanced C interop related usages of Kotlin/Native and [multiplatform](#) builds with Gradle.

First, create a project folder. All the paths in this tutorial will be relative to this folder. Sometimes the missing directories will have to be created before any new files can be added.

Use the following `build.gradle(.kts)` Gradle build file:

【Kotlin】

### 1.5.30 的新特性

```
plugins {
 kotlin("multiplatform") version "1.6.10"
}

repositories {
 mavenCentral()
}

kotlin {
 linuxX64("native") { // on Linux
 // macosX64("native") { // on x86_64 macOS
 // macosArm64("native") { // on Apple Silicon macOS
 // mingwX64("native") { // on Windows
 binaries {
 sharedLib {
 baseName = "native" // on Linux and macOS
 // baseName = "libnative" // on Windows
 }
 }
 }
 }
}

tasks.wrapper {
 gradleVersion = "6.7.1"
 distributionType = Wrapper.DistributionType.ALL
}
```

【Groovy】

## 1.5.30 的新特性

```
plugins {
 id 'org.jetbrains.kotlin.multiplatform' version '1.6.10'
}

repositories {
 mavenCentral()
}

kotlin {
 linuxX64("native") { // on Linux
 // macosX64("native") { // on x86_64 macOS
 // macosArm64("native") { // on Apple Silicon macOS
 // mingwX64("native") { // on Windows
 binaries {
 sharedLib {
 baseName = "native" // on Linux and macOS
 // baseName = "libnative" // on Windows
 }
 }
 }
 }
}

wrapper {
 gradleVersion = "6.7.1"
 distributionType = "ALL"
}
```

Move the sources file into the `src/nativeMain/kotlin` folder under the project. This is the default path, for where sources are located, when the `kotlin-multiplatform` plugin is used. Use the following block to instruct and configure the project to generate a dynamic or shared library:

```
binaries {
 sharedLib {
 baseName = "native" // on Linux and macOS
 // baseName = "libnative" // on Windows
 }
}
```

`libnative` 用作库名，即生成的头文件名前缀。它同样也是该头文件中所有声明的前缀。

现在可以在 [IntelliJ IDEA 中打开这个项目](#) 并且可以看到如何修正这个示例项目。在我们这样做的时候， 我们将会研究 C 函数如何映射为 Kotlin/Native 声明。

### 1.5.30 的新特性

运行这个 `linkNative` Gradle 任务来在 IDE 中构建该库。或者运行下面这行控制台命令：

```
./gradlew linkNative
```

构建将会在 `build/bin/native/debugShared` 文件夹下生成以下文件，并取决于目标操作系统：

- macOS: `libnative_api.h` 与 `libnative.dylib`
- Linux: `libnative_api.h` 与 `libnative.so`
- Windows: `libnative_api.h`、`libnative_symbols.def` 以及 `libnative.dll`

Kotlin/Native 编译器用相似的规则在所有的平台上生成 `.h` 文件。

来看看我们的 Kotlin 库的 C 语言 API。

## 生成头文件

在 `libnative_api.h` 中，会发现如下代码。我们来探讨其中部分代码，以便更容易理解。

Kotlin/Native 的外部符号如有变更，将不会另外说明。



第一部分包含了标准的 C/C++ 头文件的首尾：

```
#ifndef KONAN_DEMO_H
#define KONAN_DEMO_H
#ifdef __cplusplus
extern "C" {
#endif

/// 生成的代码的其余部分在这里

#ifdef __cplusplus
} /* extern "C" */
#endif
#endif /* KONAN_DEMO_H */
```

在 `libnative_api.h` 中的上述内容之后，会有一个声明通用类型定义的块：

### 1.5.30 的新特性

```
#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 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。  
让我们以更容易阅读的方式来查看类型的映射：

Kotlin 定义。	C 类型
<code>libnative_KBoolean</code>	<code>bool</code> 或 <code>_Bool</code>
<code>libnative_KChar</code>	<code>unsigned short</code>
<code>libnative_KByte</code>	<code>signed char</code>
<code>libnative_KShort</code>	<code>short</code>
<code>libnative_KInt</code>	<code>int</code>
<code>libnative_KLong</code>	<code>long long</code>
<code>libnative_KUByte</code>	<code>unsigned char</code>
<code>libnative_KUShort</code>	<code>unsigned short</code>
<code>libnative_KUInt</code>	<code>unsigned int</code>
<code>libnative_KULong</code>	<code>unsigned long long</code>
<code>libnative_KFloat</code>	<code>float</code>
<code>libnative_KDouble</code>	<code>double</code>
<code>libnative_KNativePtr</code>	<code>void*</code>

这个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。在这篇[从 C 语言中映射原生类型](#)教程中描述了反向映射。

`libnative_api.h` 文件的下一个部分包含了在该库中使用的类型的定义：

### 1.5.30 的新特性

```
struct libnative_KType;
typedef struct libnative_KType libnative_KType;

typedef struct {
 libnative_KNativePtr pinned;
} libnative_kref_example_Object;

typedef struct {
 libnative_KNativePtr pinned;
} libnative_kref_example_Class;
```

`typedef struct { .. } TYPE_NAME` 语法在 C 语言中用于声明一个结构体。

Stackoverflow 上的[这个问题](#)提供了对该模式的更多解释。

As you can see from these definitions, the Kotlin object `Object` is mapped into `libnative_kref_example_Object`, 而 `Class` 映射到了 `libnative_kref_example_Class`。这两个结构体都没有包含任何东西，但是 `pinned` 字段是一个指针，该字段类型 `libnative_KNativePtr` 定义在 `void*` 之上。

C 语言中没有支持命名空间，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。

定义的很大部分位于 `libnative_api.h` 文件。它包含了我们的 Kotlin/Native 库世界的定义：

### 1.5.30 的新特性

```
typedef struct {
 /* Service functions. */
 void (*DisposeStablePointer)(libnative_KNativePtr ptr);
 void (*DisposeString)(const char* string);
 libnative_KBoolean (*IsInstance)(libnative_KNativePtr ref, const libnative_KType*
 /* User functions. */
 struct {
 struct {
 struct {
 void (*forIntegers)(libnative_KByte b, libnative_KShort s, libnative_KUInt
 void (*forFloats)(libnative_KFloat f, libnative_KDouble d);
 const char* (*strings)(const char* str);
 const char* (*get_globalString)();
 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_Class (*Clazz)();
 libnative_KULong (*memberFunction)(libnative_kref_example_Class thiz, lib
 } Class;
 } example;
 } root;
 } kotlin;
} libnative_ExportedSymbols;
```

这段代码使用了匿名的结构体定义。代码 `struct { .. } foo` 在结构体外部声明了一个匿名结构体类型，这个类型没有名字。

C 语言同样也不支持对象。人们使用函数指针来模仿对象语义。一个函数指针被声明在 `RETURN_TYPE (* FIELD_NAME)(PARAMETERS)` 后面。它的可读性很差，但我们应该能够从上面的结构体中看到函数指针字段。

## 运行时函数

阅读下面的代码。其中有这个 `libnative_ExportedSymbols` 结构体，它定义了所有 Kotlin/Native 中的以及我们的库提供给我们的函数。它使用嵌套匿名结构，以模仿包。`libnative_` 前缀来源于库的名字。

`libnative_ExportedSymbols` 结构体包含了几个辅助函数：

### 1.5.30 的新特性

```
void (*DisposeStablePointer)(libnative_KNativePtr ptr);
void (*DisposeString)(const char* string);
libnative_KBoolean (*IsInstance)(libnative_KNativePtr ref, const libnative_KType* t
```

这些函数用于处理 Kotlin/Native 的对象。调用 `DisposeStablePointer` 来释放一个 Kotlin 对象，而 `DisposeString` 用于释放一个 Kotlin 字符串，该字符串具有 C 中的 `char*` 类型。`IsInstance` 函数可以用于检查一个 Kotlin 类型或者一个 `libnative_KNativePtr` 是否是某个类型的实例。实际的生成操作取决于实际的使用情况。

Kotlin/Native 拥有垃圾回收机制，但是它不能帮助我们处理来源于 C 的 Kotlin 对象。Kotlin/Native 可以与 Objective-C 以及 Swift 进行互操作，并且结合了它们的引用计数。这篇 [Objective-C 互操作](#) 包含了更多关于此内容的细节。当然，也可以参考这篇 [Kotlin/Native 开发 Apple Framework](#) 文档。

## 库函数

我们来看看 `kotlin.root.example` 字段，它使用 `kotlin.root.` 前缀模仿 Kotlin 代码的包结构。

这里有一个 `kotlin.root.example.Clazz` 字段用来表示 Kotlin 中的 `Clazz`。这个 `Clazz#memberFunction` 是可以使用 `memberFunction` 字段访问的。唯一的区别是 `memberFunction` 接受 `this` 引用作为第一个参数。C 语言不支持对象，所以这是为什么明确使用 `this` 指针访问的原因。

`Clazz` 字段中有一个构造函数（又名 `kotlin.root.example.Clazz.Clazz`），这是创建 `Clazz` 实例的构造函数。

Kotlin `object Object` 是可以被 `kotlin.root.example.Object` 访问的。这里的 `_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。举例来说，Kotlin 中的只读属性 `globalString` 在 C 中会转换为 `get_globalString` 函数。

全局函数 `forInts`、`forFloats` 以及 `strings` 在 `kotlin.root.example` 匿名结构体中被转换为函数指针。

## 入口点

可以看到 API 是如何创建的。首先，需要初始化 `libnative_ExportedSymbols` 结构体。我们来看看 `libnative_api.h` 的最新部分：

### 1.5.30 的新特性

```
extern libnative_ExportedSymbols* libnative_symbols(void);
```

函数 `libnative_symbols` 允许在原生代码中打开 Kotlin/Native 库。This is the entry point you'll use. 该库名称被用作函数名称的前缀。

Kotlin/Native 对象引用不支持多线程访问。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

使用 C 中的头文件非常简单明了。通过下面的代码创建了一个 `main.c` 文件：

```
#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_CClazz newInstance = lib->kotlin.root.example.CClazz();
 long x = lib->kotlin.root.example.CClazz.memberFunction(newInstance, 42);
 lib->DisposeStablePointer(newInstance.pinned);

 printf("DemoClazz returned %ld\n", x);

 return 0;
}
```

## 将示例编译并运行于 Linux 以及 macOS

### 1.5.30 的新特性

在 macOS 10.13 的 Xcode 上，使用如下命令将 C 代码编译并链接到动态库：

```
clang main.c libnative.dylib
```

在 Linux 上使用相似的命令：

```
gcc main.c libnative.so
```

编译器生成一个名为 `a.out` 的可执行文件。运行它来查看 Kotlin 代码是如何调用 C 库来运行的。在 Linux 上，将需要将 `.` 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。

## 将示例编译并运行于 Windows

首先，需要安装一个支持 64 位目标操作系统的 Microsoft Visual C++ 编译器。最简单的方法是在我们的 Windows 机器上安装相同版本的 Microsoft Visual Studio。

在本例中，会使用 `x64 Native Tools Command Prompt <VERSION>` 控制台。会看到在开始菜单中打开控制台的快捷方式。它附带一个 Microsoft Visual Studio 包。

在 Windows 上，动态库可以通过生成的静态库包装器以及手动编写代码的形式导入，使用 `LoadLibrary` 处理或类似的 Win32API 功能。使用第一种选项并为 `libnative.dll` 生成静态包装器，如下所述。

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_symbols.def /out:libnative.lib
```

现在已经准备好将 `main.c` 编译为可执行文件。将生成的 `libnative.lib` 导入到构建命令并启动：

```
cl.exe main.c libnative.lib
```

这行命令生成了 `main.exe` 文件可供执行。

## 接下来

### 1.5.30 的新特性

动态库是从现有程序使用 Kotlin 代码的主要方式。可以使用它们来共享代码到许多平台以及其他语言，包括 JVM、[Python](#)、iOS、Android 以及其他平台。

Kotlin/Native 同样也可以与 Objective-C 以及 Swift 紧密集成。这部分内容被包含在 [Kotlin/Native 开发 Apple Framework 教程中](#)。

## Kotlin/Native 中的不可变性与并发

Kotlin/Native 实现了严格的可变性检测，确保了重要的不变式：对象要么不可变，要么在同一时刻只在单个线程中访问（`mutable XOR global`）。

在 Kotlin/Native 中不可变性是运行时属性，可以将 `kotlin.native.concurrent.freeze` 函数应用到任意对象子图。它使从给定的对象递归可达的所有对象都不可变。这样的转换是单向操作。例如即这些对象之后不能解冻。一些天然不可变的对象（如 `kotlin.String`、`kotlin.Int` 与其他原生类型，以及 `AtomicInt` 与 `AtomicReference`）默认就是冻结的。如果对已冻结对象应用了修改操作，那么会抛出 `InvalidMutabilityException` 异常。

为了实现 `mutable XOR global` 不变式，所有全局可见状态（目前有 `object` 单例与枚举）都会自动冻结。如果对象无需冻结，可以使用 `kotlin.native.ThreadLocal` 注解，这会使该对象状态成为线程局部的，因此可修改（但是变更后的状态对其他线程不可见）。

非原生类型的顶层/全局变量默认只能在主线程（即首先初始化 *Kotlin/Native* 运行时的线程）中访问。在其他线程中访问会引发 `IncorrectDereferenceException` 异常。如需其他线程可访问这种变量，可以使用 `@ThreadLocal` 注解将该值标记为线程局部，或者使用 `@SharedImmutable` 注解，它会冻结该值并使其他线程可访问。See [Global variables and singletons](#).

`AtomicReference` 类可用于将变更后的冻结状态发布给其他线程，从而构建共享缓存等模式。See [Atomic primitives and references](#).

## Concurrency in Kotlin/Native

Kotlin/Native runtime doesn't encourage a classical thread-oriented concurrency model with mutually exclusive code blocks and conditional variables, as this model is known to be error-prone and unreliable. Instead, we suggest a collection of alternative approaches, allowing you to use hardware concurrency and implement blocking IO. Those approaches are as follows, and they will be elaborated on in further sections:

### Workers

### 1.5.30 的新特性

Instead of threads, Kotlin/Native runtime offers the concept of workers: concurrently executed control flow streams with an associated request queue. Workers are very similar to the actors in the Actor Model. A worker can exchange Kotlin objects with another worker so that at any moment, each mutable object is owned by a single worker, but ownership can be transferred. See section [Object transfer and freezing](#).

Once a worker is started with the `Worker.start` function call, it can be addressed with its own unique integer worker id. Other workers, or non-worker concurrency primitives, such as OS threads, can send a message to the worker with the `execute` call.

```
val future = execute(TransferMode.SAFE, { SomeDataForWorker() }) {
 // data returned by the second function argument comes to the
 // worker routine as 'input' parameter.
 input ->
 // Here we create an instance to be returned when someone consumes result future
 WorkerResult(input.stringParam + " result")
}

future.consume {
 // Here we see result returned from routine above. Note that future object or
 // id could be transferred to another worker, so we don't have to consume future
 // in same execution context it was obtained.
 result -> println("result is $result")
}
```

The call to `execute` uses a function passed as its second parameter to produce an object subgraph (for example, a set of mutually referring objects) which is then passed as a whole to that worker. It is then no longer available to the thread that initiated the request. This property is checked if the first parameter is `TransferMode.SAFE` by graph traversal and is just assumed to be true if it is `TransferMode.UNSAFE`. The last parameter to `execute` is a special Kotlin lambda, which is not allowed to capture any state and is actually invoked in the target worker's context. Once processed, the result is transferred to whatever consumes it in the future, and it is attached to the object graph of that worker/thread.

If an object is transferred in `UNSAFE` mode and is still accessible from multiple concurrent executors, the program will likely crash unexpectedly, so consider that last resort in optimizing, not a general-purpose mechanism.

For a complete example, please refer to the [workers example](#) in the Kotlin/Native repository.

## Object transfer and freezing

### 1.5.30 的新特性

An important invariant that Kotlin/Native runtime maintains is that the object is either owned by a single thread/worker, or it is immutable (*shared XOR mutable*). This ensures that the same data has a single mutator, and so there is no need for locking to exist. To achieve such an invariant, we use the concept of not externally referred object subgraphs. This is a subgraph without external references from outside of the subgraph, which could be checked algorithmically with  $O(N)$  complexity (in ARC systems), where  $N$  is the number of elements in such a subgraph. Such subgraphs are usually produced as a result of a lambda expression, for example, some builder, and may not contain objects referred to externally.

Freezing is a runtime operation making a given object subgraph immutable by modifying the object header so that future mutation attempts throw an

`InvalidMutabilityException`. It is deep, so if an object has a pointer to other objects, the transitive closure of such objects will be frozen. Freezing is a one-way transformation; frozen objects cannot be unfrozen. Frozen objects have a nice property that, due to their immutability, they can be freely shared between multiple workers/threads without breaking the "mutable XOR shared" invariant.

If an object is frozen, it can be checked with an extension property `isFrozen`, and if it is, object sharing is allowed. Currently, Kotlin/Native runtime only freezes the enum objects after creation, although additional auto-freezing of certain provably immutable objects could be implemented in the future.

## Object subgraph detachment

An object subgraph without external references can be disconnected using

`DetachedObjectGraph<T>` to a `COpaquePointer` value, which could be stored in `void*` data, so the disconnected object subgraphs can be stored in a C data structure, and later attached back with `DetachedObjectGraph<T>.attach()` in an arbitrary thread or a worker. Together with [raw memory sharing](#), it allows side-channel object transfer between concurrent threads if the worker mechanisms are insufficient for a particular task. Note that object detachment may require an explicit leaving function holding object references and then performing cyclic garbage collection. For example, code like:

### 1.5.30 的新特性

```
val graph = DetachedObjectGraph {
 val map = mutableMapOf<String, String>()
 for (entry in map.entries) {
 // ...
 }
 map
}
```

will not work as expected and will throw runtime exception, as there are uncollected cycles in the detached graph, while:

```
val graph = DetachedObjectGraph {
 {
 val map = mutableMapOf<String, String>()
 for (entry in map.entries) {
 // ...
 }
 map
 }().also {
 kotlin.native.internal.GC.collect()
 }
}
```

will work properly, as holding references will be released, and then cyclic garbage affecting the reference counter is collected.

## Raw shared memory

Considering the strong ties between Kotlin/Native and C via interoperability, in conjunction with the other mechanisms mentioned above, it is possible to build popular data structures, like concurrent hashmap or shared cache, with Kotlin/Native. It is possible to rely upon shared C data and store references to detached object subgraphs in it. Consider the following .def file:

```
package = global

typedef struct {
 int version;
 void* kotlinObject;
} SharedData;

SharedData sharedData;
```

### 1.5.30 的新特性

After running the cinterop tool, it can share Kotlin data in a versionized global structure, and interact with it from Kotlin transparently via autogenerated Kotlin like this:

```
class SharedData(rawPtr: NativePtr) : CStructVar(rawPtr) {
 var version: Int
 var kotlinObject: COpaquePointer?
}
```

So in combination with the top-level variable declared above, it can allow looking at the same memory from different threads and building traditional concurrent structures with platform-specific synchronization primitives.

## Global variables and singletons

Frequently, global variables are a source of unintended concurrency issues, so *Kotlin/Native* implements the following mechanisms to prevent the unintended sharing of state via global objects:

- global variables, unless specially marked, can be only accessed from the main thread (that is, the thread *Kotlin/Native* runtime was first initialized), if other thread access such a global, `IncorrectDereferenceException` is thrown
- for global variables marked with the `@kotlin.native.ThreadLocal` annotation, each thread keeps a thread-local copy, so changes are not visible between threads
- for global variables marked with the `@kotlin.native.SharedImmutable` annotation value is shared, but frozen before publishing, so each thread sees the same value
- singleton objects unless marked with `@kotlin.native.ThreadLocal` are frozen and shared, lazy values allowed, unless cyclic frozen structures were attempted to be created
- enums are always frozen

These mechanisms combined allow natural race-free programming with code reuse across platforms in Multiplatform projects.

## Atomic primitives and references

*Kotlin/Native* standard library provides primitives for safe working with concurrently mutable data, namely `AtomicInt`, `AtomicLong`, `AtomicNativePtr`, `AtomicReference` and `FreezableAtomicReference` in the package `kotlin.native.concurrent.Atomic`. primitives allow concurrency-safe update operations, such as increment, decrement,

### 1.5.30 的新特性

and compare-and-swap, along with value setters and getters. Atomic primitives are always considered frozen by the runtime, and while their fields can be updated with the regular `field.value += 1`, it is not concurrency safe. The value must be changed using dedicated operations so it is possible to perform concurrent-safe global counters and similar data structures.

Some algorithms require shared mutable references across multiple workers. For example, the global mutable configuration could be implemented as an immutable instance of properties list atomically replaced with the new version on configuration update as the whole in a single transaction. This way, no inconsistent configuration could be seen, and at the same time, the configuration could be updated as needed. To achieve such functionality, Kotlin/Native runtime provides two related classes:

`kotlin.native.concurrent.AtomicReference` and

`kotlin.native.concurrent.FreezableAtomicReference`. Atomic reference holds a reference to a frozen or immutable object, and its value could be updated by set or compare-and-swap operation. Thus, a dedicated set of objects could be used to create mutable shared object graphs (of immutable objects). Cycles in the shared memory could be created using atomic references. Kotlin/Native runtime doesn't support garbage collecting cyclic data when the reference cycle goes through

`AtomicReference` or frozen `FreezableAtomicReference`. So to avoid memory leaks, atomic references that are potentially parts of shared cyclic data should be zeroed out once no longer needed.

If atomic reference value is attempted to be set to a non-frozen value, a runtime exception is thrown.

Freezable atomic reference is similar to the regular atomic reference until frozen behaves like a regular box for a reference. After freezing, it behaves like an atomic reference and can only hold a reference to a frozen object.

## 处理并发

- 并发概述
- 并发可变性
- 并发与协程

## 并发概述

When you extend your development experience from Android to Kotlin Multiplatform Mobile, you will encounter a different state and concurrency model for iOS. This is a Kotlin/Native model that compiles Kotlin code to native binaries that can run without a virtual machine, for example on iOS.

Having mutable memory available to multiple threads at the same time, if unrestricted, is known to be risky and prone to error. Languages like Java, C++, and Swift/Objective-C let multiple threads access the same state in an unrestricted way. Concurrency issues are unlike other programming issues in that they are often very difficult to reproduce. You may not see them locally while developing, and they may happen sporadically. And sometimes you can only see them in production under load.

In short, just because your tests pass, you can't necessarily be sure that your code is OK.

Not all languages are designed this way. JavaScript simply does not allow you to access the same state concurrently. At the other end of the spectrum is Rust, with its language-level management of concurrency and states, which makes it very popular.

## Rules for state sharing

Kotlin/Native introduces rules for sharing states between threads. These rules exist to prevent unsafe shared access to mutable states. If you come from a JVM background and write concurrent code, you may need to change the way you architect your data, but doing so will allow you to achieve the same results without risky side effects.

It is also important to point out that there are [ways to work around these rules](#). The intent is to make working around these rules something that you rarely have to do, if ever.

There are just two simple rules regarding state and concurrency.

### Rule 1: Mutable state == 1 thread

### 1.5.30 的新特性

If your state is mutable, only one thread can see it at a time. Any regular class state that you would normally use in Kotlin is considered by the Kotlin/Native runtime as *mutable*. If you aren't using concurrency, Kotlin/Native behaves the same as any other Kotlin code, with the exception of [global state](#).

```
data class SomeData(var count:Int)

fun simpleState(){
 val sd = SomeData(42)
 sd.count++
 println("My count is ${sd.count}") // It will be 43
}
```

If there's only one thread, you won't have concurrency issues. Technically this is referred to as *thread confinement*, which means that you cannot change the UI from a background thread. Kotlin/Native's state rules formalize that concept for all threads.

## Rule 2: Immutable state == many threads

If a state can't be changed, multiple threads can safely access it. In Kotlin/Native, *immutable* doesn't mean everything is a `val`. It means *frozen state*.

## Immutable and frozen state

The example below is immutable by definition – it has 2 `val` elements, and both are of final immutable types.

```
data class SomeData(val s:String, val i:Int)
```

This next example may be immutable or mutable. It is not clear what `SomeInterface` will do internally at compile time. In Kotlin, it is not possible to determine deep immutability statically at compile time.

```
data class SomeData(val s:String, val i:SomeInterface)
```

Kotlin/Native needs to verify that some part of a state really is immutable at runtime. The runtime could simply go through the whole state and verify that each part is deeply immutable, but that would be inflexible. And if you needed to do that every time the

### 1.5.30 的新特性

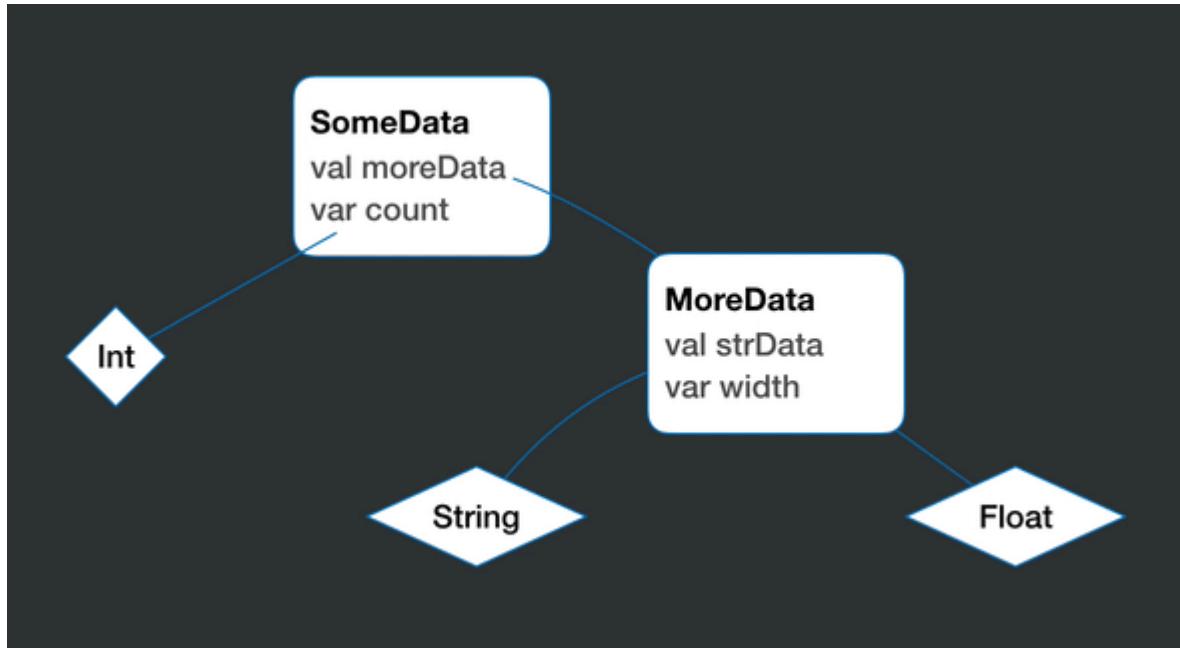
runtime wanted to check mutability, there would be significant consequences for performance.

Kotlin/Native defines a new runtime state called *frozen*. Any instance of an object may be frozen. If an object is frozen:

1. You cannot change any part of its state. Attempting to do so will result in a runtime exception: `InvalidMutabilityException`. A frozen object instance is 100%, runtime-verified, immutable.
2. Everything it references is also frozen. All other objects it has a reference to are guaranteed to be frozen. This means that, when the runtime needs to determine whether an object can be shared with another thread, it only needs to check whether that object is frozen. If it is, the whole graph is also frozen and is safe to be shared.

The Native runtime adds an extension function `freeze()` to all classes. Calling `freeze()` will freeze an object, and everything referenced by the object, recursively.

```
data class MoreData(val strData: String, var width: Float)
data class SomeData(val moreData: MoreData, var count: Int)
//...
val sd = SomeData(MoreData("abc", 10.0), 0)
sd.freeze()
```



- `freeze()` is a one-way operation. You can't *unfreeze* something.
- `freeze()` is not available in shared Kotlin code, but several libraries provide expect and actual declarations for using it in shared code. However, if you're using

### 1.5.30 的新特性

a concurrency library, like `kotlinx.coroutines`, it will likely freeze data that crosses thread boundaries automatically.

`freeze` is not unique to Kotlin. You can also find it in [Ruby](#) and [JavaScript](#).

## Global state

Kotlin allows you to define a state as globally available. If left simply mutable, the global state would violate [Rule 1](#).

To conform to Kotlin/Native's state rules, the global state has some special conditions. These conditions freeze the state or make it visible only to a single thread.

## Global object

Global `object` instances are frozen by default. This means that all threads can access them, but they are immutable. The following won't work.

```
object SomeState{
 var count = 0
 fun add(){
 count++ //This will throw an exception
 }
}
```

Trying to change `count` will throw an exception because `SomeState` is frozen (which means all of its data is frozen).

You can make a global object *thread local*, which will allow it to be mutable and give each thread a copy of its state. Annotate it with `@ThreadLocal`.

```
@ThreadLocal
object SomeState{
 var count = 0
 fun add(){
 count++ //👍
 }
}
```

If different threads read `count`, they'll get different values, because each thread has its own copy.

These global object rules also apply to companion objects.

### 1.5.30 的新特性

```
class SomeState{
 companion object{
 var count = 0
 fun add(){
 count++ //This will throw an exception
 }
 }
}
```

## Global properties

Global properties are a special case. *They are only available to the main thread*, but they are mutable. Accessing them from other threads will throw an exception.

```
val hello = "Hello" //Only main thread can see this
```

You can annotate them with :

- `@SharedImmutable` , which will make them globally available but frozen.
- `@ThreadLocal` , which will give each thread its own mutable copy.

This rule applies to global properties with backing fields. Computed properties and global functions do not have the main thread restriction.

## Current and future models

Kotlin/Native's concurrency rules will require some adjustment in architecture design, but with the help of libraries and new best practices, day to day development is basically unaffected. In fact, adhering to Kotlin/Native's rules regarding multiplatform code will result in safer concurrency across the cross-platform mobile application. You can try out the Kotlin/Native concurrency model in [this hands-on tutorial](#).

In the Kotlin Multiplatform application, you have Android and iOS targets with different state rules. Some teams, generally ones working on larger applications, share code for very specific functionality, and often manage concurrency in the host platform. This will require explicit freezing of states returned from Kotlin, but otherwise, it is straightforward.

## 1.5.30 的新特性

A more extensive model, where concurrency is managed in Kotlin and the host communicates on its main thread to shared code, is simpler from a state management perspective. Concurrency libraries, like [kotlinx.coroutines](#), will help automate freezing. You'll also be able to leverage the power of [coroutines](#) in your code and increase efficiency by sharing more code.

However, the current Kotlin/Native concurrency model has a number of deficiencies. For example, mobile developers are used to freely sharing their objects between threads, and they have already developed a number of approaches and architectural patterns to avoid data races while doing so. It is possible to write efficient applications that do not block the main thread using Kotlin/Native, but the ability to do so comes with a steep learning curve.

That's why we are working on creating a new memory manager and concurrency model for Kotlin/Native that will help us remove these drawbacks. Learn more about [where we are going with this](#).

*This material was prepared by [Touchlab](#) for publication by JetBrains.*

## 并发可变性

When it comes to working with iOS, [Kotlin/Native's state and concurrency model](#) has [two simple rules](#).

1. A mutable, non-frozen state is visible to only one thread at a time.
2. An immutable, frozen state can be shared between threads.

The result of following these rules is that you can't change [global states](#), and you can't change the same shared state from multiple threads. In many cases, simply changing your approach to how you design your code will work fine, and you don't need concurrent mutability. States were mutable from multiple threads in JVM code, but they didn't *need* to be.

However, in many other cases, you may need arbitrary thread access to a state, or you may have *service* objects that should be available to the entire application. Or maybe you simply don't want to go through the potentially costly exercise of redesigning existing code. Whatever the reason, *it will not always be feasible to constrain a mutable state to a single thread*.

There are various techniques that help you work around these restrictions, each with their own pros and cons:

- [Atomics](#)
- [Thread-isolated states](#)
- [Low-level capabilities](#)

## Atomics

Kotlin/Native provides a set of Atomic classes that can be frozen while still supporting changes to the value they contain. These classes implement a special-case handling of states in the Kotlin/Native runtime. This means that you can change values inside a frozen state.

The Kotlin/Native runtime includes a few different variations of Atomics. You can use them directly or from a library.

### 1.5.30 的新特性

Kotlin provides an experimental low-level `kotlinx.atomicfu` library that is currently used only for internal purposes and is not supported for general usage. You can also use [Stately](#), a utility library for multiplatform compatibility with Kotlin/Native-specific concurrency, developed by [Touchlab](#).

## AtomicInt / AtomicLong

The first two are simple numerics: `AtomicInt` and `AtomicLong`. They allow you to have a shared `Int` or `Long` that can be read and changed from multiple threads.

```
object AtomicDataCounter {
 val count = AtomicInt(3)

 fun addOne() {
 count.increment()
 }
}
```

The example above is a global `object`, which is frozen by default in Kotlin/Native. In this case, however, you can change the value of `count`. It's important to note that you can change the value of `count` *from any thread*.

## AtomicReference

`AtomicReference` holds an object instance, and you can change that object instance. The object you put in `AtomicReference` must be frozen, but you can change the value that `AtomicReference` holds. For example, the following won't work in Kotlin/Native:

```
data class SomeData(val i: Int)

object GlobalData {
 var sd = SomeData(0)

 fun storeNewValue(i: Int) {
 sd = SomeData(i) //Doesn't work
 }
}
```

According to the [rules of global state](#), global `object` values are frozen in Kotlin/Native, so trying to modify `sd` will fail. You could implement it instead with `AtomicReference`:

### 1.5.30 的新特性

```
data class SomeData(val i: Int)

object GlobalData {
 val sd = AtomicReference(SomeData(0).freeze())

 fun storeNewValue(i: Int) {
 sd.value = SomeData(i).freeze()
 }
}
```

The `AtomicReference` itself is frozen, which lets it live inside something that is frozen.

The data in the `AtomicReference` instance is explicitly frozen in the code above.

However, in the multiplatform libraries, the data will be frozen automatically. If you use the Kotlin/Native runtime's `AtomicReference`, you *should* remember to call `freeze()` explicitly.

`AtomicReference` can be very useful when you need to share a state. There are some drawbacks to consider, however.

Accessing and changing values in an `AtomicReference` is very costly performance-wise *relative to* a standard mutable state. If performance is a concern, you may want to consider using another approach involving a [thread-isolated state](#).

There is also a potential issue with memory leaks, which will be resolved in the future. In situations where the object kept in the `AtomicReference` has cyclical references, it may leak memory if you don't explicitly clear it out:

- If you have state that may have cyclic references and needs to be reclaimed, you should use a nullable type in the `AtomicReference` and set it to null explicitly when you're done with it.
- If you're keeping `AtomicReference` in a global object that never leaves scope, this won't matter (because the memory never needs to be reclaimed during the life of the process).

### 1.5.30 的新特性

```
class Container(a:A) {
 val atom = AtomicReference<A?>(a.freeze())

 /**
 * Call when you're done with Container
 */
 fun clear(){
 atom.value = null
 }
}
```

Finally, there's also a consistency concern. Setting/getting values in `AtomicReference` is itself atomic, but if your logic requires a longer chain of thread exclusion, you'll need to implement that yourself. For example, if you have a list of values in an `AtomicReference` and you want to scan them first before adding a new one, you'll need to have some form of concurrency management that `AtomicReference` alone does not provide.

The following won't protect against duplicate values in the list if called from multiple threads:

```
object MyListCache {
 val atomicList = AtomicReference(listOf<String>().freeze())
 fun addEntry(s:String){
 val l = atomicList.value
 val newList = mutableListOf<String>()
 newList.addAll(l)
 if(!newList.contains(s)){
 newList.add(s)
 }
 atomicList.value = newList.freeze()
 }
}
```

You will need to implement some form of locking or check-and-set logic to ensure proper concurrency.

## Thread-isolated state

[Rule 1 of Kotlin/Native state](#) is that a mutable state is visible to only one thread. Atomics allow mutability from any thread. Isolating a mutable state to a single thread, and allowing other threads to communicate with that state, is an alternative method for

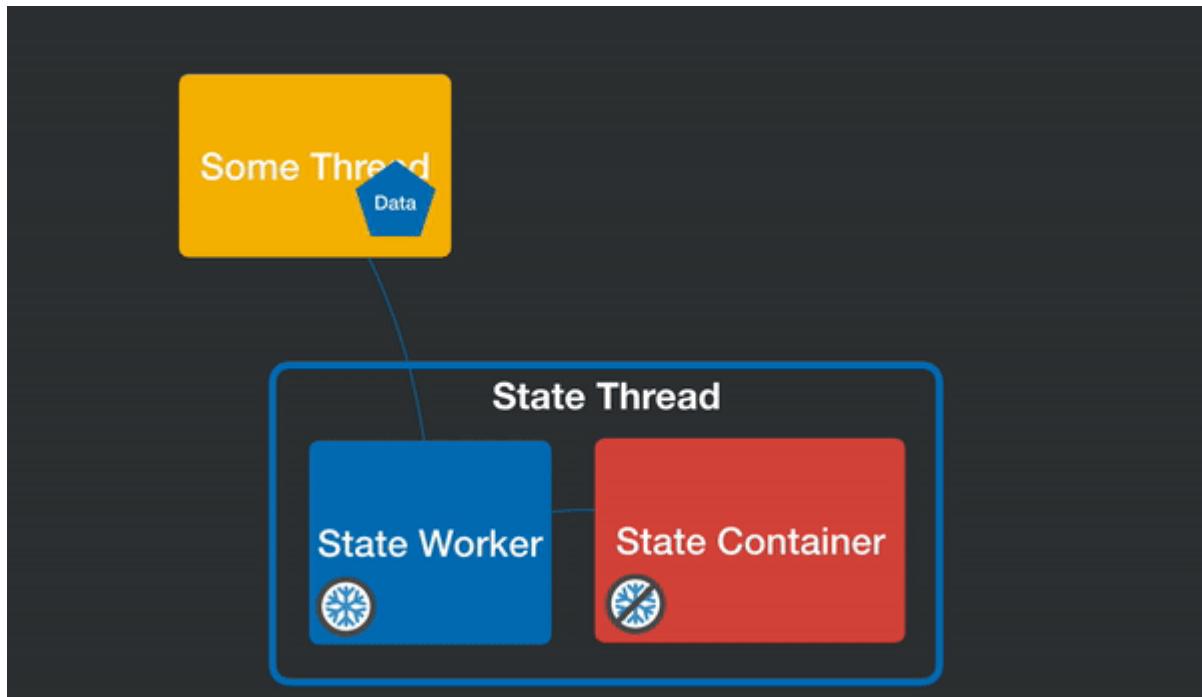
### 1.5.30 的新特性

achieving concurrent mutability.

To do this, create a work queue that has exclusive access to a thread, and create a mutable state that lives in just that thread. Other threads communicate with the mutable thread by scheduling *work* on the work queue.

Data that goes in or comes out, if any, needs to be frozen, but the mutable state hidden in the worker thread remains mutable.

Conceptually it looks like the following: one thread pushes a frozen state into the state worker, which stores it in the mutable state container. Another thread later schedules work that takes that state out.



Implementing thread-isolated states is somewhat complex, but there are libraries that provide this functionality.

## AtomicReference vs. thread-isolated state

For simple values, `AtomicReference` will likely be an easier option. For cases with significant states, and potentially significant state changes, using a thread-isolated state may be a better choice. The main performance penalty is actually crossing over threads. But in performance tests with collections, for example, a thread-isolated state significantly outperforms a mutable state implemented with `AtomicReference`.

### 1.5.30 的新特性

The thread-isolated state also avoids the consistency issues that `AtomicReference` has. Because all operations happen in the state thread, and because you're scheduling work, you can perform operations with multiple steps and guarantee consistency without managing thread exclusion. Thread isolation is a design feature of the Kotlin/Native state rules, and isolating mutable states works with those rules.

The thread-isolated state is also more flexible insofar as you can make mutable states concurrent. You can use any type of mutable state, rather than needing to create complex concurrent implementations.

## Low-level capabilities

Kotlin/Native has some more advanced ways of sharing concurrent states. To achieve high performance, you may need to avoid the concurrency rules altogether.

This is a more advanced topic. You should have a deep understanding of how concurrency in Kotlin/Native works under the hood, and you'll need to be very careful when using this approach. Learn more about [concurrency](#).



Kotlin/Native runs on top of C++ and provides interop with C and Objective-C. If you are running on iOS, you can also pass lambda arguments into your shared code from Swift. All of this native code runs outside of the Kotlin/Native state restrictions.

That means that you can implement a concurrent mutable state in a native language and have Kotlin/Native talk to it.

You can use [Objective-C interop](#) to access low-level code. You can also use Swift to implement Kotlin interfaces or pass in lambdas that Kotlin code can call from any thread.

One of the benefits of a platform-native approach is performance. On the negative side, you'll need to manage concurrency on your own. Objective-C does not know about `frozen`, but if you store states from Kotlin in Objective-C structures, and share them between threads, the Kotlin states definitely need to be frozen. Kotlin/Native's runtime will generally warn you about issues, but it's possible to cause concurrency problems in native code that are very, very difficult to track down. It is also very easy to create memory leaks.

### 1.5.30 的新特性

Since in the Kotlin Multiplatform application you are also targeting the JVM, you'll need alternate ways to implement anything you use platform native code for. This will obviously take more work and may lead to platform inconsistencies.

*This material was prepared by [Touchlab](#) for publication by JetBrains.*

## 并发与协程

When working with mobile platforms, you may need to write multithreaded code that runs in parallel. For this, you can use the [standard `kotlinx.coroutines` library](#) or its [multithreaded version](#) and [alternative solutions](#).

Review the pros and cons of each solution and choose the one that works best for your situation.

Learn more about [concurrency, the current approach, and future improvements](#).

## Coroutines

Coroutines are light-weight threads that allow you to write asynchronous non-blocking code. Kotlin provides the [`kotlinx.coroutines`](#) library with a number of high-level coroutine-enabled primitives.

The current version of `kotlinx.coroutines`, which can be used for iOS, supports usage only in a single thread. You cannot send work to other threads by changing a [dispatcher](#).

For Kotlin 1.6.10, the recommended coroutines version is `1.6.0`.

You can suspend execution and do work on other threads while using a different mechanism for scheduling and managing that work. However, this version of `kotlinx.coroutines` cannot change threads on its own.

There is also [another version of `kotlinx.coroutines`](#) that provides support for multiple threads.

Get acquainted with the main concepts for using coroutines:

- [Asynchronous vs. parallel processing](#)
- [Dispatcher for changing threads](#)
- [Frozen captured data](#)
- [Frozen returned data](#)

## Asynchronous vs. parallel processing

Asynchronous and parallel processing are different.

### 1.5.30 的新特性

Within a coroutine, the processing sequence may be suspended and resumed later. This allows for asynchronous, non-blocking code, without using callbacks or promises. That is asynchronous processing, but everything related to that coroutine can happen in a single thread.

The following code makes a network call using [Ktor](#). In the main thread, the call is initiated and suspended, while another underlying process performs the actual networking. When completed, the code resumes in the main thread.

```
val client = HttpClient()
//Running in the main thread, start a `get` call
client.get<String>("https://example.com/some/rest/call")
//The get call will suspend and let other work happen in the main thread, and resume
```

That is different from parallel code that needs to be run in another thread. Depending on your purpose and the libraries you use, you may never need to use multiple threads.

## Dispatcher for changing threads

Coroutines are executed by a dispatcher that defines which thread the coroutine will be executed on. There are a number of ways in which you can specify the dispatcher, or change the one for the coroutine. For example:

```
suspend fun differentThread() = withContext(Dispatchers.Default){
 println("Different thread")
}
```

`withContext` takes both a dispatcher as an argument and a code block that will be executed by the thread defined by the dispatcher. Learn more about [coroutine context and dispatchers](#).

To perform work on a different thread, specify a different dispatcher and a code block to execute. In general, switching dispatchers and threads works similar to the JVM, but there are differences related to freezing captured and returned data.

## Frozen captured data

To run code on a different thread, you pass a `functionBlock`, which gets frozen and then runs in another thread.

### 1.5.30 的新特性

```
fun <R> runOnDifferentThread(functionBlock: () -> R)
```

You will call that function as follows:

```
runOnDifferentThread {
 //Code run in another thread
}
```

As described in the [concurrency overview](#), a state shared between threads in Kotlin/Native must be frozen. A function argument is a state itself, which will be frozen along with anything it captures.

Coroutine functions that cross threads use the same pattern. To allow function blocks to be executed on another thread, they are frozen.

In the following example, the data class instance `dc` will be captured by the function block and will be frozen when crossing threads. The `println` statement will print `true`.

```
val dc = DataClass("Hello")
withContext(Dispatchers.Default) {
 println("${dc.isFrozen}")
}
```

When running parallel code, be careful with the captured state. Sometimes it's obvious when the state will be captured, but not always. For example:

```
class SomeModel(val id:IdRec){
 suspend fun saveData() = withContext(Dispatchers.Default){
 saveToDb(id)
 }
}
```

The code inside `saveData` runs on another thread. That will freeze `id`, but because `id` is a property of the parent class, it will also freeze the parent class.

## Frozen returned data

Data returned from a different thread is also frozen. Even though it's recommended that you return immutable data, you can return a mutable state in a way that doesn't allow a returned value to be changed.

### 1.5.30 的新特性

```
val dc = withContext(Dispatchers.Default) {
 DataClass("Hello Again")
}

println("${dc.isFrozen}")
```

It may be a problem if a mutable state is isolated in a single thread and coroutine threading operations are used for communication. If you attempt to return data that retains a reference to the mutable state, it will also freeze the data by association.

Learn more about the [thread-isolated state](#).

## Multithreaded coroutines

A [special branch](#) of the `kotlinx.coroutines` library provides support for using multiple threads. It is a separate branch for the reasons listed in the [future concurrency model blog post](#).

However, you can still use the multithreaded version of `kotlinx.coroutines` in production, taking its specifics into account.

The current version for Kotlin 1.6.10 is `1.6.0-native-mt`.

To use the multithreaded version, add a dependency for the `commonMain` source set in `build.gradle.kts`:

```
commonMain {
 dependencies {
 implementation "org.jetbrains.kotlinx:kotlinx-coroutines-core:1.6.0-native-
 }
}
```

When using other libraries that also depend on `kotlinx.coroutines`, such as Ktor, make sure to specify the multithreaded version of `kotlinx-coroutines`. You can do this with `strictly`:

```
implementation ("org.jetbrains.kotlinx:kotlinx-coroutines-core:1.6.0-native-mt"){
 version {
 strictly("1.6.0-native-mt")
 }
}
```

### 1.5.30 的新特性

Because the main version of `kotlinx.coroutines` is a single-threaded one, libraries will almost certainly rely on this version. If you see `InvalidMutabilityException` related to a coroutine operation, it's very likely that you are using the wrong version.

Using multithreaded coroutines may result in *memory leaks*. This can be a problem for complex coroutine scenarios under load. We are working on a solution for this.



See a [complete example of using multithreaded coroutines in a Kotlin Multiplatform application](#).

## Alternatives to `kotlinx-coroutines`

There are a few alternative ways to run parallel code.

### CoroutineWorker

[CoroutinesWorker](#) is a library published by AutoDesk that implements some features of coroutines across threads using the single-threaded version of `kotlinx.coroutines`.

For simple suspend functions this is a pretty good option, but it does not support Flow and other structures.

### Reaktive

[Reaktive](#) is an Rx-like library that implements Reactive extensions for Kotlin Multiplatform. It has some coroutine extensions but is primarily designed around RX and threads.

### Custom processor

For simpler background tasks, you can create your own processor with wrappers around platform specifics. See a [simple example](#).

### Platform concurrency

### 1.5.30 的新特性

In production, you can also rely on the platform to handle concurrency. This could be helpful if the shared Kotlin code will be used for business logic or data operations rather than architecture.

To share a state in iOS across threads, that state needs to be [frozen](#). The concurrency libraries mentioned here will freeze your data automatically. You will rarely need to do so explicitly, if ever.

If you return data to the iOS platform that should be shared across threads, ensure that data is frozen before leaving the iOS boundary.

Kotlin has the concept of frozen only for Kotlin/Native platforms including iOS. To make `freeze` available in common code, you can create expect and actual implementations for `freeze`, or use `stately-common`, which provides this functionality. In Kotlin/Native, `freeze` will freeze your state, while on the JVM it'll do nothing.

To use `stately-common`, add a dependency for the `commonMain` source set in `build.gradle.kts`:

```
commonMain {
 dependencies {
 implementation "co.touchlab:stately-common:1.0.x"
 }
}
```

*This material was prepared by [Touchlab](#) for publication by JetBrains.*

## 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.

## 1.5.30 的新特性

```
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
(lldb) r
Process 28473 launched: '/Users/minamoto/ws/.git-trees/debugger-fixes/terminator.kexe'
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>)
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>)
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>)
```

### 1.5.30 的新特性

`-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>)
```

- 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
```

## 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
```

- by address

### 1.5.30 的新特性

```
(gdb) b *0x100001704
Note: breakpoint 2 also set at pc 0x100001704.
Breakpoint 3 at 0x100001704: file /Users/minamoto/ws/.git-trees/hello.kt, line 1
```

## 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` :

### 1.5.30 的新特性

```
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
(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(
`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):

### 1.5.30 的新特性

```
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' --(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
(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>
 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}"
 }
 }
}
```

### 1.5.30 的新特性

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
 }
 }
}
```

## Decode inlined stack frames

Xcode doesn't seem to properly decode stack trace elements of inlined function calls (these aren't only Kotlin `inline` functions but also functions that are inlined when optimizing machine code). So some stack trace elements may be missing. If this is the

### 1.5.30 的新特性

case, consider using `lldb` to process crash report that is already symbolicated by Xcode, for example:

```
$ lldb -b -o "script import lldb.macosx" -o "crashlog file.crash"
```

This command should output crash report that is additionally processed and includes inlined stack trace elements.

More details can be found in [LLDB documentation](#).

# 改进 Kotlin/Native 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 most recent version of Kotlin.** This way you will always have the latest performance improvements.
- **Avoid creating huge classes.** They 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 `KONAN_DATA_DIR` environment variable.

## 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.

Here are some recommendations for configuring Gradle for better compilation performance:

### 1.5.30 的新特性

- **Increase the Gradle heap size.** Add `org.gradle.jvmargs=-Xmx3g` to `gradle.properties`. If you use [parallel builds](#), you might need to make the heap even larger or choose the right number of threads with `org.gradle.parallel.threads`.
- **Build only the binaries you need.** 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. Keep in mind that compiling a release binary (`linkRelease*`) takes more time than compiling a debug one.
- `packForXcode` : 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. During local development, it will be faster to build the `.framework` for only the platform you're using.

To build a platform-specific framework, call the `packForXcode` task generated by the [Kotlin Multiplatform Mobile project wizard](#).

Remember that in this case, you will need to clean the build using `./gradlew clean` after switching between the device and the simulator. See [this issue](#) for details.



- **Don't disable the Gradle daemon** without having a good reason to. [Kotlin/Native runs from the Gradle daemon](#) by default. 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 `transitiveExport = true`**. Using transitive export disables dead code elimination in many cases: the compiler has to process a lot of unused code. It increases the compilation time. Use `export` explicitly for exporting the required projects and dependencies.
- **Use the Gradle build caches:**

## 1.5.30 的新特性

- **Local build cache:** Add `org.gradle.caching=true` to your `gradle.properties` or run with `--build-cache` on the command line.
- **Remote build cache** in continuous integration environments. Learn how to [configure the remote build cache](#).
- **Enable previously disabled features of Kotlin/Native.** There are properties that disable the Gradle daemon and compiler caches –  
`kotlin.native.disableCompilerDaemon=true` and `kotlin.native.cacheKind=none`. If you had issues with these features before and added these lines to your `gradle.properties` 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.

## Windows OS configuration

- **Configure Windows Security.** 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 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 provides an automated memory management scheme, similar to what Java or Swift provides. The current implementation includes an automated reference counter with a cycle collector to collect cyclical garbage.

### How do I create a shared library?

Use the `-produce dynamic` compiler switch, or `binaries.sharedLib()` in Gradle.

```
kotlin {
 iosArm64("mylib") {
 binaries.sharedLib()
 }
}
```

It will produce 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. See [this example](#) of using such a shared object to provide a bridge between Python and Kotlin/Native.

## How do I create a static library or an object file?

Use the `-produce static` compiler switch, or `binaries.staticLib()` in Gradle.

```
kotlin {
 iosArm64("mylib") {
 binaries.staticLib()
 }
}
```

It will produce 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】

### 1.5.30 的新特性

```
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 {
 basePath = "TheName"
 }
 }
 }
}
```

## How do I enable bitcode for my Kotlin framework?

By default gradle plugin adds it on iOS target.

- For debug build it embeds placeholder LLVM IR data as a marker.
- For release build it embeds bitcode as data.

Or commandline arguments: `-Xembed-bitcode` (for release) and `-Xembed-bitcode-marker` (debug)

Setting this in a Gradle DSL:

### 1.5.30 的新特性

```
kotlin {
 iosArm64("myapp") {
 binaries {
 framework {
 // Use "marker" to embed the bitcode marker (for debug builds).
 // Use "disable" to disable embedding.
 embedBitcode("bitcode") // for release binaries.
 }
 }
 }
}
```

These options have nearly the same effect as clang's `-fembed-bitcode / -fembed-bitcode-marker` and swiftc's `-embed-bitcode / -embed-bitcode-marker`.

## Why do I see `InvalidMutabilityException` ?

It likely happens, because you are trying to mutate a frozen object. An object can transfer to the frozen state either explicitly, as objects reachable from objects on which the `kotlin.native.concurrent.freeze` is called, or implicitly (i.e. reachable from `enum` or global singleton object - see the next question).

## How do I make a singleton object mutable?

Currently, singleton objects are immutable (i.e. frozen after creation), and it's generally considered good practise to have the global state immutable. If for some reason you need a mutable state inside such an object, use the `@konan.ThreadLocal` annotation on the object. Also the `kotlin.native.concurrent.AtomicReference` class could be used to store different pointers to frozen objects in a frozen object and automatically update them.

## 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](#).

# Kotlin/Native 中的并发

Kotlin/Native 运行时并不鼓励带有互斥代码块与条件变量的经典线程式并发模型，因为已知该模型易出错且不可靠。相反，我们建议使用一系列替代方法，让你可以使用硬件并发并实现阻塞 IO。这些方法如下，并且分别会在后续各部分详细阐述：

- 带有消息传递的 worker
- 对象子图所有权转移
- 对象子图冻结
- 对象子图分离
- 使用 C 语言全局变量的原始共享内存
- Atomic primitives and references
- 用于阻塞操作的协程（本文档未涉及）

## Worker

Kotlin/Native 运行时提供了 worker 的概念来取代线程：并发执行的控制流以及与其关联的请求队列。Worker 非常像参与者模型中的参与者。一个 worker 可以与另一个 worker 交换 Kotlin 对象，从而在任何时刻每个可变对象都隶属于单个 worker，不过所有权可以转移。请参见[对象转移与冻结](#)部分。

一旦以 `Worker.start` 函数调用启动了一个 worker，就可以使用其自身唯一的整数 `worker id` 来寻址。其他 worker 或者非 worker 的并发原语（如 OS 线程）可以使用 `execute` 调用向 worker 发消息。

```
val future = execute(TransferMode.SAFE, { SomeDataForWorker() }) {
 // 第二个函数参数所返回的数据
 // 作为“input”参数进入 worker 例程
 input ->
 // 这里我们创建了一个当有人消费结果 future 时返回的实例
 WorkerResult(input.stringParam + " result")
}

future.consume {
 // 这里我们查看从上文例程中返回的结果。请注意 future 对象或
 // id 都可以转移给另一个 worker，所以并不是必须要在
 // 获得 future 的同一上下文中消费之。
 result -> println("result is $result")
}
```

### 1.5.30 的新特性

调用 `execute` 会使用作为第二个参数传入的函数来生成一个对象子图（即一组相互引用的对象）然后将其作为一个整体传给该 `worker`，之后发出该请求的线程不可以再使用该对象子图。如果第一个参数是 `TransferMode.SAFE`，那么会通过图遍历来检测这一属性；而如果第一个参数是 `TransferMode.UNSAFE` 那么直接假定为 `true`。`execute` 的最后一个参数是一个特殊 Kotlin lambda 表达式，不可以捕获任何状态，并且实际上是在目标 `worker` 的上下文中调用。一旦处理完毕，就将结果转移给将会消费它地方，并将其附加到该 `worker`/线程的对象图中。

如果一个对象以 `UNSAFE` 模式转移，并且依然在多个并发执行子中访问，那么该程序可能会意外崩溃，因此考虑将 `UNSAFE` 作为最后的优化手段而不是通用机制来使用。

更完整的示例请参考 [Kotlin/Native 版本库中的 worker 示例](#)。

## 对象转移与冻结

Kotlin/Native 运行时维护的一个重要的不变式是，对象要么归单个线程/`worker` 所有，要么不可变（共享 XOR 可变）。这确保了同一数据只有一个修改方，因此不需要锁定。为了实现这个不变式，我们使用了非外部引用的对象子图的概念。这是一个没有来自子图以外的外部引用的子图，（在 ARC 系统中）可由  $O(N)$  复杂度进行算法检测，其中  $N$  是这种子图中元素的数量。这种子图通常是作为 lambda 表达式的结果而产生的（例如某些构建器），并且可能不含外部引用的对象。

冻结是一种运行时操作，通过修改对象头使给定的对象子图不可变，这样之后的修改企图都会抛出 `InvalidMutabilityException`。它是深度冻结，因此如果一个对象有指向其他对象的指针——这些对象的传递闭包也都会被冻结。冻结是单向转换，冻结的对象不能解冻。冻结的对象有一个很好的属性，由于其不可变性，它们可以在多个 `worker`/线程之间自由共享，而不会破坏“可变 XOR 共享”不变式。

一个对象是否已冻结，可以使用扩展属性 `isFrozen` 来检测，如果冻结了就可以共享。目前，Kotlin/Native 运行时只能在枚举对象创建后进行冻结，尽管将来可能实现自动冻结某些可证明不可变的对象。

## 对象子图分离

没有外部引用的对象子图可以使用 `DetachedObjectGraph<T>` 断开到 `COpaquePointer` 值的连接，该值可以存储在 `void*` 数据中，因此断开连接的对象子图可以存储在 C 语言数据结构中，并且之后还能在任意线程或 `worker` 中通过

`DetachedObjectGraph<T>.attach()` 加回。如果 `worker` 机制不足以完成特定任务，那么

### 1.5.30 的新特性

可以将对象子图分离与[原始共享内存](#)相结合，能够在并发线程之间进行旁路对象传输。

Note, that object detachment may require explicit leaving function holding object references and then performing cyclic garbage collection. For example, code like:

```
val graph = DetachedObjectGraph {
 val map = mutableMapOf<String, String>()
 for (entry in map.entries) {
 // ...
 }
 map
}
```

will not work as expected and will throw runtime exception, as there are uncollected cycles in the detached graph, while:

```
val graph = DetachedObjectGraph {
 {
 val map = mutableMapOf<String, String>()
 for (entry in map.entries) {
 // ...
 }
 map
 }().also {
 kotlin.native.internal.GC.collect()
 }
}
```

will work properly, as holding references will be released, and then cyclic garbage affecting reference counter is collected.

## 原始共享内存

考虑到 Kotlin/Native 与 C 语言之间通过互操作性的紧密联系，结合上文中提到的其他机制，可以构建流行的数据结构，如并发的 hashmap 或者与 Kotlin/Native 共享缓存。可以依赖共享的 C 语言数据，并在其中存储分离的对象子图的引用。考虑以下 .def 文件：

### 1.5.30 的新特性

```
package = global

typedef struct {
 int version;
 void* kotlinObject;
} SharedData;

SharedData sharedData;
```

在运行 cinterop 工具之后，可以在版本化的全局结构中共享 Kotlin 数据，并通过自动生成的 Kotlin 代码在 Kotlin 中与其透明交互，如下所示：

```
class SharedData(rawPtr: NativePtr) : CStructVar(rawPtr) {
 var version: Int
 var kotlinObject: COpaquePointer?
}
```

因此，结合上文声明的顶层变量，可以让不同的线程看到相同的内存，并使用平台相关的同步原语来构建传统的并发结构。

## 全局变量与单例

全局变量常常是非预期并发问题的根源，因此 *Kotlin/Native* 实现了以下机制来防止意外通过全局对象共享状态：

- 全局变量（除非特别标记过）都只能在主线程（即首次初始化 *Kotlin/Native* 运行时的线程）中访问，如果其他线程访问这样的全局变量就会抛出 `IncorrectDereferenceException`
- 对于标有 `@kotlin.native.ThreadLocal` 注解的全局变量，每个线程都保留线程局部副本，因此变更在线程之间并不可见
- 对于标有 `@kotlin.native.SharedImmutable` 注解的变量，其值是共享的，但是在发布之前会被冻结，因此每个线程都会看到相同的值
- 单例对象（除非标有 `@kotlin.native.ThreadLocal`）都是冻结且共享的，允许惰性值（除非企图创建循环冻结结构）
- 枚举总是冻结的

结合起来，这些机制允许在多平台项目中跨平台复用代码的自然竞态冻结编程。

## Atomic primitives and references

### 1.5.30 的新特性

Kotlin/Native standard library provides primitives for safe working with concurrently mutable data, namely `AtomicInt`, `AtomicLong`, `AtomicNativePtr`, `AtomicReference` and `FreezableAtomicReference` in the package `kotlin.native.concurrent.Atomic`. Atomic primitives allows concurrency-safe update operations, such as increment, decrement and compare-and-swap, along with value setters and getters. Atomic primitives are considered always frozen by the runtime, and while their fields can be updated with the regular `field.value += 1`, it is not concurrency safe. Value must be changed using dedicated operations, so it is possible to perform concurrent-safe global counters and similar data structures.

Some algorithms require shared mutable references across the multiple workers, for example global mutable configuration could be implemented as an immutable instance of properties list atomically replaced with the new version on configuration update as the whole in a single transaction. This way no inconsistent configuration could be seen, and at the same time configuration could be updated as needed. To achieve such functionality Kotlin/Native runtime provides two related classes:

`kotlin.native.concurrent.AtomicReference` and

`kotlin.native.concurrent.FreezableAtomicReference`. Atomic reference holds reference to a frozen or immutable object, and its value could be updated by set or compare-and-swap operation. Thus, dedicated set of objects could be used to create mutable shared object graphs (of immutable objects). Cycles in the shared memory could be created using atomic references. Kotlin/Native runtime doesn't support garbage collecting cyclic data when reference cycle goes through `AtomicReference` or frozen `FreezableAtomicReference`. So to avoid memory leaks atomic references that are potentially parts of shared cyclic data should be zeroed out once no longer needed.

If atomic reference value is attempted to be set to non-frozen value runtime exception is thrown.

Freezable atomic reference is similar to the regular atomic reference, but until frozen behaves like regular box for a reference. After freezing it behaves like an atomic reference, and can only hold a reference to a frozen object.

## 脚本

- [Kotlin 自定义脚本入门——教程](#)

# Kotlin 自定义脚本入门——教程

Kotlin scripting in [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:

### 1.5.30 的新特性

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.

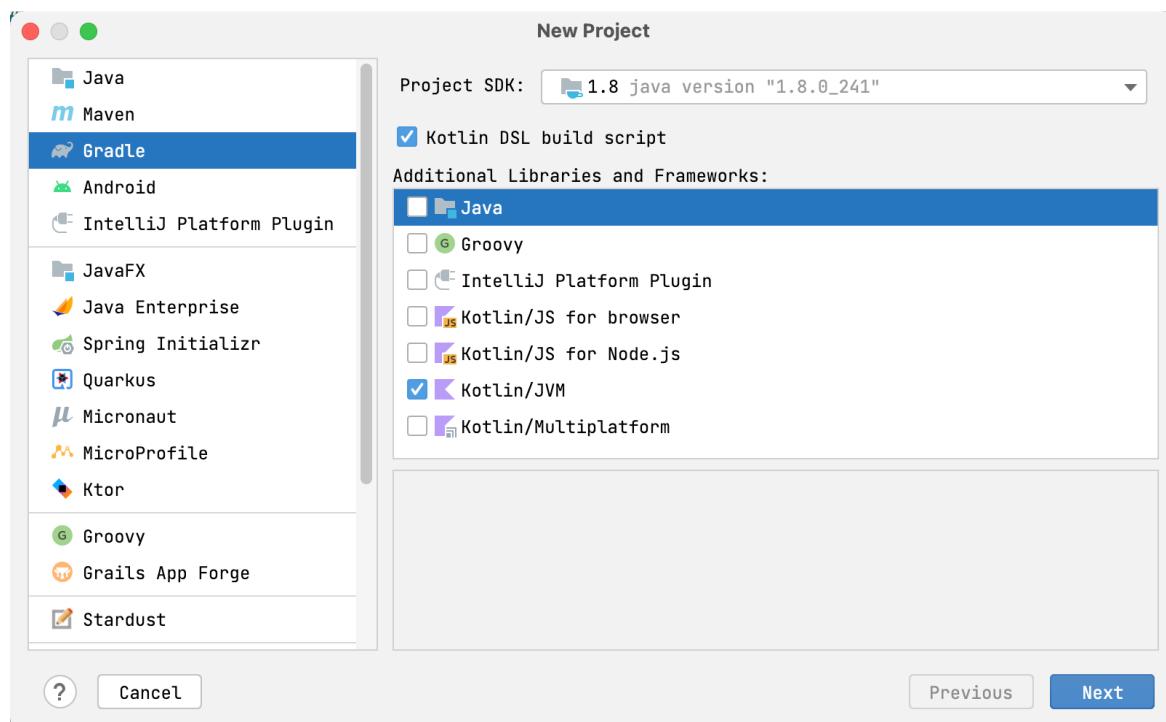
## Before you start

Download and install the latest version of [IntelliJ IDEA](#).

## Set up the project structure

Create a Kotlin/JVM Gradle project with two modules:

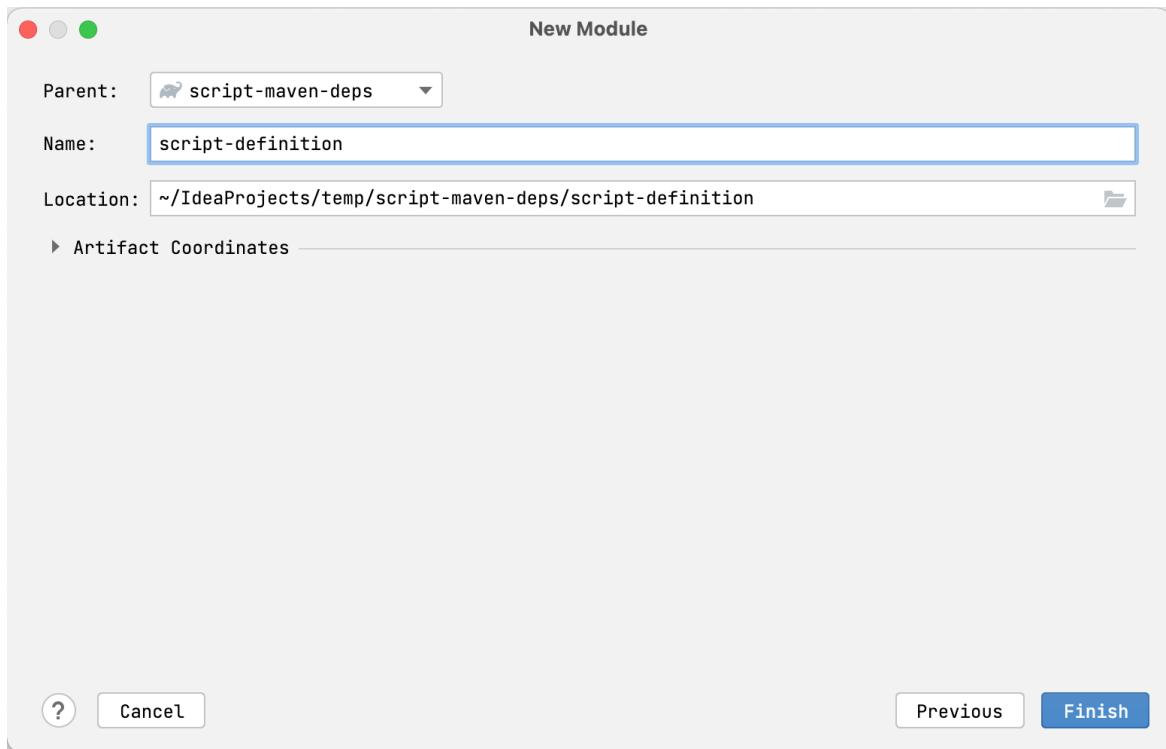
1. Go to **File | New | Project**.
2. Create a new **Gradle** project with **Kotlin/JVM**. Select the **Kotlin DSL build script** checkbox to write the build script in Kotlin.



Now you have an empty Kotlin/JVM Gradle project where you will add the required modules: script definition and scripting host.

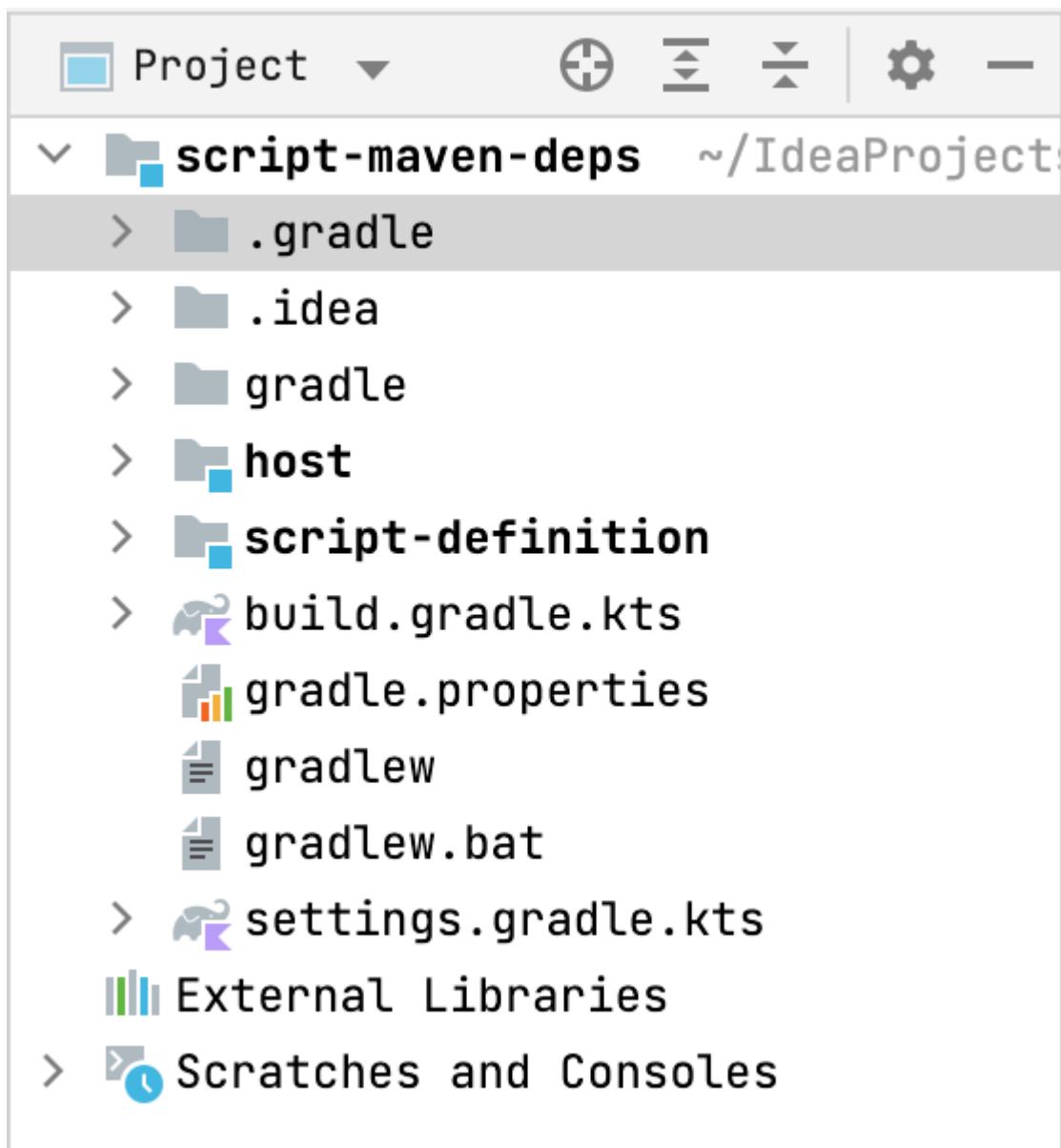
### 1.5.30 的新特性

3. Go to **File | New | Module** and add a new **Gradle** module with **Kotlin/JVM**. Select the **Kotlin DSL build script** checkbox if you want to write the build script in Kotlin. This module will be the script definition.
4. Give the module a name and select the root module as its parent.



5. 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.
6. Repeat steps 3, 4, and 5 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】

## 1.5.30 的新特性

```
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:1.6.0")
}
```

### 【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-jvm:1.6.0'

}
```

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.

### 1.5.30 的新特性

```
// @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
 dependenciesFromCurrentContext(wholeClasspath = true)
 }
 // Callbacks
 refineConfiguration {
 // Process specified annotations with the provided handler
 onAnnotations(DependsOn::class, Repository::class, handler = ::configureMavenDepsOnAnnotations)
 }
}
```

The `configureMavenDepsOnAnnotations` function is as follows:

### 1.5.30 的新特性

```
// Handler that reconfigures the compilation on the fly
fun configureMavenDepsOnAnnotations(context: ScriptConfigurationRefinementContext)
 : CompilationConfiguration = runBlocking {
 val annotations = context.collectedData?.get(ScriptCollectedData.collectedAnnotations)
 ?: return context.compilationConfiguration.asSuccess()
 resolver.resolveFromScriptSourceAnnotations(annotations)
}.onSuccess {
 context.compilationConfiguration.with {
 dependencies.append(JvmDependency(it))
 }.asSuccess()
}
}

private val resolver = CompoundDependenciesResolver(FileSystemDependenciesResolver)
```

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】

## 1.5.30 的新特性

```
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()
 return BasicJvmScriptingHost().eval(scriptFile.toScriptSource(), compilationConfiguration)
}
```

4. Adjust the `main` function to print information about the script execution:

## 1.5.30 的新特性

```
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
 }
 }
 }
}
```

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"

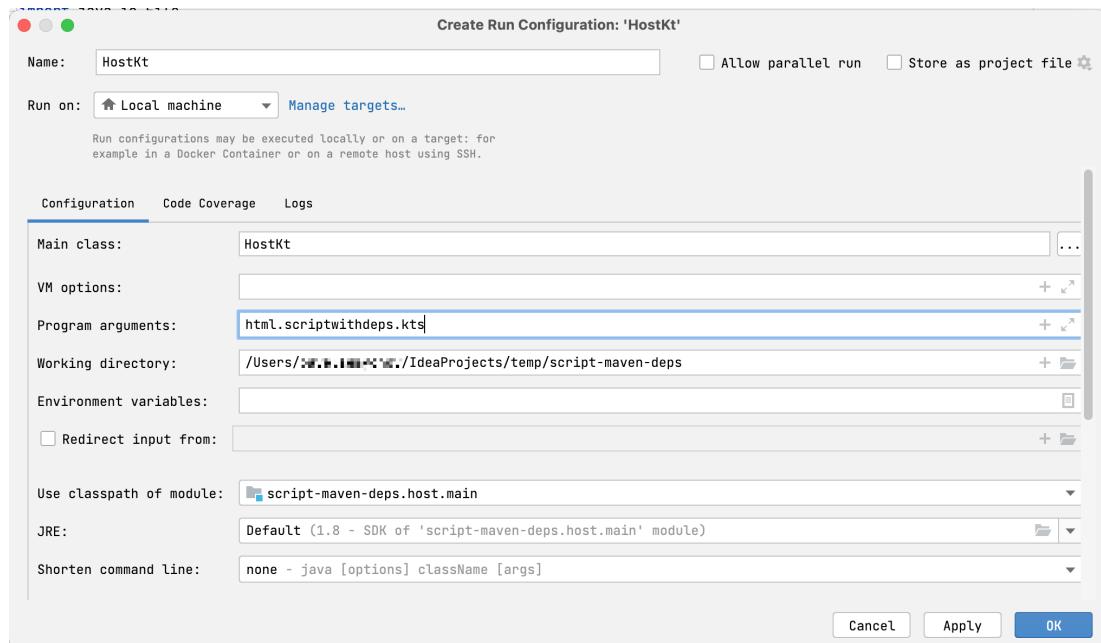
print(
 createHTML().html {
 body {
 h1 { +"Hello, $addressee!" }
 }
 }
)
```

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:

## 1.5.30 的新特性

- 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.

## What's next?

Once you've created a simple Kotlin scripting project, find more information on this topic:

- Read the [Kotlin scripting KEP](#)

### 1.5.30 的新特性

- [Browse more Kotlin scripting examples](#)
- [Watch the talk Implementing the Gradle Kotlin DSL by Rodrigo Oliveira](#)

## 版本发布与路线图

- [Kotlin 版本发布](#)
- [Kotlin 多平台移动端插件版本发布](#)
- [Kotlin 插件版本发布](#)
- [Kotlin 路线图](#)

# Kotlin 版本发布

We ship different types of releases:

- *Feature releases* (1.x) that bring major changes in the language.
- *Incremental releases* (1.x.y) that are shipped between feature releases and include updates in the tooling, performance improvements, and bug fixes.
- *Bug fix releases* (1.x.yz) that include bug fixes for incremental releases.

For example, for the feature release 1.3 we had several incremental releases including 1.3.10, 1.3.20, and 1.3.70. For 1.3.70, we had 2 bug fix releases – 1.3.71 and 1.3.72.

For each incremental and feature release, we also ship several preview (*EAP*) versions for you to try new features before they are released. See [Early Access Preview](#) for details.

Learn more about [types of Kotlin releases and their compatibility](#).

## 更新到新版本

IntelliJ IDEA and Android Studio suggest updating to a new release once it is out. When you accept the suggestion, it automatically updates the Kotlin plugin to the new version. You can check the Kotlin version in **Tools | Kotlin | Configure Kotlin Plugin Updates**.

If you have projects created with earlier Kotlin versions, change the Kotlin version in your projects and update kotlinx libraries if necessary – check the [recommended versions](#).

If you are migrating to the new feature release, Kotlin plugin's migration tools will help you with the migration.

## IDE support

The IDE support for the latest version of the language is available for the following versions of IntelliJ IDEA and Android Studio:

- IntelliJ IDEA:

### 1.5.30 的新特性

- Latest stable ([IntelliJ IDEA 2021.3 version](#))
- Previous stable ([IntelliJ IDEA 2021.2 version](#))
- [Early access](#) versions
- Android Studio:
  - [Latest released](#) version
  - [Early access](#) versions

## 版本发布详情

The following table lists details of latest Kotlin releases.

You can also use [preview versions of Kotlin](#).

## 1.5.30 的新特性

版本信息	版本重点	推荐的 kotlinx 库版本
<b>1.6.10</b> Released: <b>December 14, 2021</b>  <a href="#">Release on GitHub</a>	A bug fix release for Kotlin 1.6.0.  <a href="#">Learn more about Kotlin 1.6.0.</a>	<ul style="list-style-type: none"><li>• <a href="#">kotlinx.serialization</a> version: 1.3.1</li><li>• <a href="#">kotlinx.coroutines</a> version: 1.6.0</li><li>• <a href="#">kotlinx.atomicfu</a> version: 0.17.0</li><li>• <a href="#">ktor</a> version: 2.0.0-beta-1</li><li>• <a href="#">kotlinx.html</a> version: 0.7.2</li><li>• <a href="#">kotlinx-nodejs</a> version: 0.0.7</li></ul> <p>The versions of libraries from <code>kotlin-wrappers</code> (such as <code>kotlin-react</code> ) can be found in the <a href="#">corresponding repository</a>.</p>
<b>1.6.0</b> Released: <b>November 16, 2021</b>  <a href="#">Release on GitHub</a>	A feature release with new language features, performance improvements, and evolutionary changes such as stabilizing experimental APIs.  Learn more in: <ul style="list-style-type: none"><li>• <a href="#">Release blog post</a></li><li>• <a href="#">What's new in Kotlin 1.6.0</a></li><li>• <a href="#">Compatibility Guide</a></li></ul>	<ul style="list-style-type: none"><li>• <a href="#">kotlinx.serialization</a> version: 1.3.0</li><li>• <a href="#">kotlinx.coroutines</a> version: 1.6.0</li><li>• <a href="#">kotlinx.atomicfu</a> version: 0.16.3</li><li>• <a href="#">ktor</a> version: 1.6.4</li><li>• <a href="#">kotlinx.html</a> version: 0.7.2</li><li>• <a href="#">kotlinx-nodejs</a> version: 0.0.7</li></ul> <p>The versions of libraries from <code>kotlin-wrappers</code> (such as <code>kotlin-react</code> ) can be found in the <a href="#">corresponding repository</a>.</p>

## 1.5.30 的新特性

<p><b>1.5.32</b> Released: <b>November 29, 2021</b>  <a href="#">Release on GitHub</a></p>	<p>A bug fix release for Kotlin 1.5.31.  <a href="#">Learn more about Kotlin 1.5.30.</a></p>	<ul style="list-style-type: none"><li>• <a href="#">kotlinx.serialization</a> version: 1.3.0-RC</li><li>• <a href="#">kotlinx.coroutines</a> version: 1.5.2</li><li>• <a href="#">kotlinx.atomicfu</a> version: 0.16.3</li><li>• <a href="#">ktor</a> version: 1.6.3</li><li>• <a href="#">kotlinx.html</a> version: 0.7.2</li><li>• <a href="#">kotlinx-nodejs</a> version: 0.0.7</li></ul> <p>The versions of libraries from <code>kotlin-wrappers</code> (such as <code>kotlin-react</code> ) can be found in the <a href="#">corresponding repository</a>.</p>
<p><b>1.5.31</b> Released: <b>September 20, 2021</b>  <a href="#">Release on GitHub</a></p>	<p>A bug fix release for Kotlin 1.5.30.  <a href="#">Learn more about Kotlin 1.5.30.</a></p>	<ul style="list-style-type: none"><li>• <a href="#">kotlinx.serialization</a> version: 1.3.0-RC</li><li>• <a href="#">kotlinx.coroutines</a> version: 1.5.2</li><li>• <a href="#">kotlinx.atomicfu</a> version: 0.16.3</li><li>• <a href="#">ktor</a> version: 1.6.3</li><li>• <a href="#">kotlinx.html</a> version: 0.7.2</li><li>• <a href="#">kotlinx-nodejs</a> version: 0.0.7</li></ul> <p>The versions of libraries from <code>kotlin-wrappers</code> (such as <code>kotlin-react</code> ) can be found in the <a href="#">corresponding repository</a>.</p>

## 1.5.30 的新特性

<p><b>1.5.30</b> Released: <b>August 23, 2021</b>  <a href="#">Release on GitHub</a></p>	<p>An incremental release with various improvements such as:</p> <ul style="list-style-type: none"><li>• Instantiation of annotation classes on JVM</li><li>• Improved opt-in requirement mechanism and type inference</li><li>• Kotlin/JS IR backend in Beta</li><li>• Support for Apple Silicon targets</li><li>• Improved CocoaPods support</li><li>• Gradle: Java toolchain support and improved daemon configuration</li></ul> <p>Learn more in:</p> <ul style="list-style-type: none"><li>• <a href="#">Release blog post</a></li><li>• <a href="#">What's new in Kotlin 1.5.30</a></li></ul>	<ul style="list-style-type: none"><li>• <b>kotlinx.serialization</b> version: 1.3.0-RC</li><li>• <b>kotlinx.coroutines</b> version: 1.5.1</li><li>• <b>kotlinx.atomicfu</b> version: 0.16.2</li><li>• <b>ktor</b> version: 1.6.2</li><li>• <b>kotlinx.html</b> version: 0.7.2</li><li>• <b>kotlinx-nodejs</b> version: 0.0.7</li></ul> <p>The versions of libraries from <code>kotlin-wrappers</code> (such as <code>kotlin-react</code>) can be found in the <a href="#">corresponding repository</a>.</p>
<p><b>1.5.21</b> Released: <b>July 13, 2021</b>  <a href="#">Release on GitHub</a></p>	<p>A bug fix release for Kotlin 1.5.20.</p> <p>Learn more about <a href="#">Kotlin 1.5.20</a>.</p>	<ul style="list-style-type: none"><li>• <b>kotlinx.serialization</b> version: 1.2.1</li><li>• <b>kotlinx.coroutines</b> version: 1.5.0</li><li>• <b>kotlinx.atomicfu</b> version: 0.16.1</li><li>• <b>Ktor</b> version: 1.6.0</li><li>• <b>kotlinx.html</b> version: 0.7.2</li><li>• <b>kotlinx-nodejs</b> version: 0.0.7</li></ul> <p>The versions of libraries from <code>kotlin-wrappers</code> (such as <code>kotlin-react</code>) can be found in the <a href="#">corresponding repository</a>.</p>

## 1.5.30 的新特性

<b>1.5.20</b> Released: <b>June 24, 2021</b>  <a href="#">Release on GitHub</a>	<p>An incremental release with various improvements such as:</p> <ul style="list-style-type: none"><li>• String concatenation via <code>invokedynamic</code> on JVM by default</li><li>• Improved support for Lombok and support for JSpecify</li><li>• Kotlin/Native: KDoc export to Objective-C headers and faster <code>Array.copyOf()</code> inside one array</li><li>• Gradle: caching of annotation processors' classloaders and support for the <code>--parallel</code> Gradle property</li><li>• Aligned behavior of stdlib functions across platforms</li></ul> <p>Learn more in:</p> <ul style="list-style-type: none"><li>• <a href="#">Release blog post</a></li><li>• <a href="#">What's new in Kotlin 1.5.20</a></li></ul>	<ul style="list-style-type: none"><li>• <code>kotlinx.serialization</code> version: 1.2.1</li><li>• <code>kotlinx.coroutines</code> version: 1.5.0</li><li>• <code>kotlinx.atomicfu</code> version: 0.16.1</li><li>• <code>ktor</code> version: 1.6.0</li><li>• <code>kotlinx.html</code> version: 0.7.2</li><li>• <code>kotlinx-nodejs</code> version: 0.0.7</li></ul> <p>The versions of libraries from <code>kotlin-wrappers</code> (such as <code>kotlin-react</code>) can be found in the <a href="#">corresponding repository</a>.</p>
<b>1.5.10</b> Released: <b>May 24, 2021</b>  <a href="#">Release on GitHub</a>	<p>A bug fix release for Kotlin 1.5.0.</p> <p>Learn more about <a href="#">Kotlin 1.5.0</a>.</p>	<ul style="list-style-type: none"><li>• <code>kotlinx.serialization</code> version: 1.2.1</li><li>• <code>kotlinx.coroutines</code> version: 1.5.0</li><li>• <code>kotlinx.atomicfu</code> version: 0.16.1</li><li>• <code>ktor</code> version: 1.5.4</li><li>• <code>kotlinx.html</code> version: 0.7.2</li><li>• <code>kotlinx-nodejs</code> version: 0.0.7</li></ul> <p>The versions of libraries from <code>kotlin-wrappers</code> (such as <code>kotlin-react</code>) can be found in the <a href="#">corresponding repository</a>.</p>

## 1.5.30 的新特性

<p><b>1.5.0</b> Released: <b>May 5, 2021</b></p> <p><a href="#">Release on GitHub</a></p>	<p>A feature release with new language features, performance improvements, and evolutionary changes such as stabilizing experimental APIs.</p> <p>Learn more in:</p> <ul style="list-style-type: none"><li>• <a href="#">Release blog post</a></li><li>• <a href="#">What's new in Kotlin 1.5.0</a></li><li>• <a href="#">Compatibility Guide</a></li></ul>	<ul style="list-style-type: none"><li>• <a href="#">kotlinx.serialization</a> version: 1.2.1</li><li>• <a href="#">kotlinx.coroutines</a> version: 1.5.0-RC</li><li>• <a href="#">kotlinx.atomicfu</a> version: 0.16.1</li><li>• <a href="#">ktor</a> version: 1.5.3</li><li>• <a href="#">kotlinx.html</a> version: 0.7.2</li><li>• <a href="#">kotlinx-nodejs</a> version: 0.0.7</li></ul> <p>The versions of libraries from <code>kotlin-wrappers</code> (such as <code>kotlin-react</code>) can be found in the <a href="#">corresponding repository</a>.</p>
<p><b>1.4.32</b> Released: <b>March 22, 2021</b></p> <p><a href="#">Release on GitHub</a></p>	<p>A bug fix release for Kotlin 1.4.30.</p> <p>Learn more about <a href="#">Kotlin 1.4.30</a>.</p>	<ul style="list-style-type: none"><li>• <a href="#">kotlinx.serialization</a> version: 1.1.0</li><li>• <a href="#">kotlinx.coroutines</a> version: 1.4.3</li><li>• <a href="#">kotlinx.atomicfu</a> version: 0.15.2</li><li>• <a href="#">ktor</a> version: 1.5.2</li><li>• <a href="#">kotlinx.html</a> version: 0.7.2</li><li>• <a href="#">kotlinx-nodejs</a> version: 0.0.7</li></ul> <p>The versions of libraries from <code>kotlin-wrappers</code> (such as <code>kotlin-react</code>) can be found in the <a href="#">corresponding repository</a>.</p>

## 1.5.30 的新特性

<p><b>1.4.31</b> Released: <b>February 25, 2021</b>  <a href="#">Release on GitHub</a></p>	<p>A bug fix release for Kotlin 1.4.30  <a href="#">Learn more about Kotlin 1.4.30.</a></p>	<ul style="list-style-type: none"><li>• <a href="#">kotlinx.serialization</a> version: 1.1.0</li><li>• <a href="#">kotlinx.coroutines</a> version: 1.4.2</li><li>• <a href="#">kotlinx.atomicfu</a> version: 0.15.1</li><li>• <a href="#">ktor</a> version: 1.5.1</li><li>• <a href="#">kotlinx.html</a> version: 0.7.2</li><li>• <a href="#">kotlinx-nodejs</a> version: 0.0.7</li></ul> <p>The versions of libraries from <code>kotlin-wrappers</code> (such as <code>kotlin-react</code> ) can be found in the <a href="#">corresponding repository</a>.</p>
<p><b>1.4.30</b> Released: <b>February 3, 2021</b>  <a href="#">Release on GitHub</a></p>	<p>An incremental release with various improvements such as:</p> <ul style="list-style-type: none"><li>• New JVM backend, now in Beta</li><li>• Preview of new language features</li><li>• Improved Kotlin/Native performance</li><li>• Standard library API improvements</li></ul> <p>Learn more in:</p> <ul style="list-style-type: none"><li>• <a href="#">Release blog post</a></li><li>• <a href="#">What's new in Kotlin 1.4.30</a></li></ul>	<ul style="list-style-type: none"><li>• <a href="#">kotlinx.serialization</a> version: 1.1.0-RC</li><li>• <a href="#">kotlinx.coroutines</a> version: 1.4.2</li><li>• <a href="#">kotlinx.atomicfu</a> version: 0.15.1</li><li>• <a href="#">ktor</a> version: 1.5.1</li><li>• <a href="#">kotlinx.html</a> version: 0.7.2</li><li>• <a href="#">kotlinx-nodejs</a> version: 0.0.7</li></ul> <p>The versions of libraries from <code>kotlin-wrappers</code> (such as <code>kotlin-react</code> ) can be found in the <a href="#">corresponding repository</a>.</p>

## 1.5.30 的新特性

<p><b>1.4.21</b> Released: <b>December 7, 2020</b>  <a href="#">Release on GitHub</a></p>	<p>A bug fix release for Kotlin 1.4.20  <a href="#">Learn more about Kotlin 1.4.20.</a></p>	<ul style="list-style-type: none"><li>• <a href="#">kotlinx.serialization</a> version: 1.0.1</li><li>• <a href="#">kotlinx.coroutines</a> version: 1.4.1</li><li>• <a href="#">kotlinx.atomicfu</a> version: 0.14.4</li><li>• <a href="#">ktor</a> version: 1.4.1</li><li>• <a href="#">kotlinx.html</a> version: 0.7.2</li><li>• <a href="#">kotlinx-nodejs</a> version: 0.0.6</li></ul> <p>The versions of libraries from <code>kotlin-wrappers</code> (such as <code>kotlin-react</code> ) can be found in the <a href="#">corresponding repository</a>.</p>
<p><b>1.4.20</b> Released: <b>November 23, 2020</b>  <a href="#">Release on GitHub</a></p>	<p>An incremental release with various improvements such as:</p> <ul style="list-style-type: none"><li>• Supporting new JVM features, like string concatenation via <code>invokedynamic</code></li><li>• Improved performance and exception handling for KMM projects</li><li>• Extensions for JDK Path: <code>Path("dir") / "file.txt"</code></li></ul> <p>Learn more in:</p> <ul style="list-style-type: none"><li>• <a href="#">Release blog post</a></li><li>• <a href="#">What's new in Kotlin 1.4.20</a></li></ul>	<ul style="list-style-type: none"><li>• <a href="#">kotlinx.serialization</a> version: 1.0.1</li><li>• <a href="#">kotlinx.coroutines</a> version: 1.4.1</li><li>• <a href="#">kotlinx.atomicfu</a> version: 0.14.4</li><li>• <a href="#">ktor</a> version: 1.4.1</li><li>• <a href="#">kotlinx.html</a> version: 0.7.2</li><li>• <a href="#">kotlinx-nodejs</a> version: 0.0.6</li></ul> <p>The versions of libraries from <code>kotlin-wrappers</code> (such as <code>kotlin-react</code> ) can be found in the <a href="#">corresponding repository</a>.</p>

## 1.5.30 的新特性

<p><b>1.4.10</b> 发布于: <b>2020-09-07</b>  <a href="#">GitHub 上的版本发布</a></p>	<p>A bug fix release for Kotlin 1.4.0.  <a href="#">Learn more about Kotlin 1.4.0.</a></p>	<ul style="list-style-type: none"><li>• <a href="#">kotlinx.serialization</a> version: 1.0.0-RC</li><li>• <a href="#">kotlinx.coroutines</a> version: 1.3.9</li><li>• <a href="#">kotlinx.atomicfu</a> version: 0.14.4</li><li>• <a href="#">ktor</a> version: 1.4.0</li><li>• <a href="#">kotlinx.html</a> version: 0.7.2</li><li>• <a href="#">kotlinx-nodejs</a> version: 0.0.6</li></ul> <p>The versions of libraries from <code>kotlin-wrappers</code> (such as <code>kotlin-react</code> ) can be found in the <a href="#">corresponding repository</a>.</p>
<p><b>1.4.0</b> 发布于: <b>2020-08-17</b>  <a href="#">GitHub 上的版本发布</a></p>	<p>A feature release with many features and improvements that mostly focus on quality and performance.  Learn more in:<ul style="list-style-type: none"><li>• <a href="#">Release blog post</a></li><li>• <a href="#">What's new in Kotlin 1.4.0</a></li><li>• <a href="#">Compatibility Guide</a></li><li>• <a href="#">Migrating to Kotlin 1.4.0</a></li></ul></p>	<ul style="list-style-type: none"><li>• <a href="#">kotlinx.serialization</a> version: 1.0.0-RC</li><li>• <a href="#">kotlinx.coroutines</a> version: 1.3.9</li><li>• <a href="#">kotlinx.atomicfu</a> version: 0.14.4</li><li>• <a href="#">ktor</a> version: 1.4.0</li><li>• <a href="#">kotlinx.html</a> version: 0.7.2</li><li>• <a href="#">kotlinx-nodejs</a> version: 0.0.6</li></ul> <p>The versions of libraries from <code>kotlin-wrappers</code> (such as <code>kotlin-react</code> ) can be found in the <a href="#">corresponding repository</a>.</p>

## 1.5.30 的新特性

<p><b>1.3.72</b> 发布于: <b>2020-04-15</b> <a href="#">GitHub 上的版本发布</a></p>	<p>A bug fix release for Kotlin 1.3.70. <a href="#">Learn more about Kotlin 1.3.70.</a></p>	<ul style="list-style-type: none"><li>• <a href="#">kotlinx.serialization</a> version: <a href="#">0.20.0</a></li><li>• <a href="#">kotlinx.coroutines</a> version: <a href="#">1.3.8</a></li><li>• <a href="#">kotlinx.atomicfu</a> version: <a href="#">0.14.2</a></li><li>• <a href="#">ktor</a> version: <a href="#">1.3.2</a></li><li>• <a href="#">kotlinx.html</a> version: <a href="#">0.7.1</a></li><li>• <a href="#">kotlinx-nodejs</a> version: <a href="#">0.0.3</a></li></ul> <p>The versions of libraries from <code>kotlin-wrappers</code> (such as <code>kotlin-react</code>) can be found in the <a href="#">corresponding repository</a>.</p>
---------------------------------------------------------------------------------------	-------------------------------------------------------------------------------------------------	-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

On the JVM, you usually can use library versions other than the recommended ones.



# Kotlin 多平台移动端插件版本发布

Since Kotlin Multiplatform Mobile is now in [Alpha](#), we are working on stabilizing the corresponding [plugin for Android Studio](#) and will be regularly releasing new versions that include new features, improvements, and bug fixes.

Ensure that you have the latest version of the Kotlin Multiplatform Mobile plugin!

## Update to the new release

Android Studio will suggest updating to a new Kotlin Multiplatform Mobile plugin release as soon as it is available. If you accept the suggestion, it will automatically update the plugin to the latest version. You'll need to restart Android Studio to complete the plugin installation.

You can check the plugin version and update it manually in **Settings/Preferences | Plugins**.

You need a compatible version of Kotlin for the plugin to work correctly. You can find compatible versions in the [release details](#). You can check your Kotlin version and update it in **Settings/Preferences | Plugins** or in **Tools | Kotlin | Configure Kotlin Plugin Updates**.

If you do not have a compatible version of Kotlin installed, the Kotlin Multiplatform Mobile plugin will be disabled. You will need to update your Kotlin version, and then enable the plugin in **Settings/Preferences | Plugins**.



## 版本发布详情

The following table lists the details of the latest Kotlin Multiplatform Mobile plugin releases:

## 1.5.30 的新特性

Release info	Release highlights	Compatible Kotlin version
<b>0.3.1</b>  Released: 15 February, 2022	<ul style="list-style-type: none"><li>Enabled M1 iOS simulator in Kotlin Multiplatform Mobile wizards.</li><li>Improved performance for indexing XcProjects: <a href="#">KT-49777</a>, <a href="#">KT-50779</a>.</li><li>Build scripts clean up: use <code>kotlin("test")</code> instead of <code>kotlin("test-common")</code> and <code>kotlin("test-annotations-common")</code>.</li><li>Increase compatibility range with <a href="#">Kotlin plugin version</a>.</li><li>Fixed the problem with JVM debug on Windows host.</li><li>Fixed the problem with the invalid version after disabling the plugin.</li></ul>	<ul style="list-style-type: none"><li>Kotlin 1.5.0—1.6.*</li></ul>
<b>0.3.0</b>  Released: 16 November, 2021	<ul style="list-style-type: none"><li>New <a href="#">Kotlin Multiplatform Library wizard</a>.</li><li>Support for the new type of Kotlin Multiplatform library distribution: <a href="#">XCFramework</a>.</li><li>Enabled <a href="#">hierarchical project structure</a> for new cross-platform mobile projects.</li><li>Support for <a href="#">explicit iOS targets declaration</a>.</li><li>Enabled <a href="#">Kotlin Multiplatform Mobile plugin wizards</a> on non-Mac machines.</li><li>Support for subfolders in the <a href="#">Kotlin Multiplatform module wizard</a>.</li><li>Support for <a href="#">Xcode Assets.xcassets file</a>.</li><li>Fixed the plugin classloader exception.</li><li>Updated the CocoaPods Gradle Plugin template.</li><li>Kotlin/Native debugger type evaluation improvements.</li><li>Fixed iOS device launching with Xcode 13.</li></ul>	<ul style="list-style-type: none"><li>Kotlin 1.6.0</li></ul>

## 1.5.30 的新特性

<b>0.2.7</b>  Released: August 2, 2021	<ul style="list-style-type: none"><li>Added Xcode configuration option for AppleRunConfiguration.</li><li>Added support Apple M1 simulators.</li><li>Added information about Xcode integration options in Project Wizard.</li><li>Added error notification after a project with CocoaPods was generated, but the CocoaPods gem has not been installed.</li><li>Added support Apple M1 simulator target in generated shared module with Kotlin 1.5.30.</li><li>Cleared generated Xcode project with Kotlin 1.5.20.</li><li>Fixed launching Xcode Release configuration on a real iOS device.</li><li>Fixed simulator launching with Xcode 12.5.</li></ul>	<ul style="list-style-type: none"><li>Kotlin 1.5.10</li></ul>
<b>0.2.6</b>  Released: June 10, 2021	<ul style="list-style-type: none"><li>Compatibility with Android Studio Bumblebee Canary 1.</li><li>Support for <a href="#">Kotlin 1.5.20</a>: using the new framework-packing task for Kotlin/Native in the Project Wizard.</li></ul>	<ul style="list-style-type: none"><li>Kotlin 1.5.10</li></ul>
<b>0.2.5</b>  Released: May 25, 2021	<ul style="list-style-type: none"><li>Fixed compatibility with Android Studio Arctic Fox 2020.3.1 Beta 1 and higher.</li></ul>	<ul style="list-style-type: none"><li>Kotlin 1.5.10</li></ul>
<b>0.2.4</b>  Released: May 5, 2021	<p>Use this version of the plugin with Android Studio 4.2 or Android Studio 2020.3.1 Canary 8 or higher.</p> <ul style="list-style-type: none"><li>Compatibility with <a href="#">Kotlin 1.5.0</a>.</li><li>Ability to use the CocoaPods dependency manager in the Kotlin Multiplatform module for iOS integration.</li></ul>	<ul style="list-style-type: none"><li>Kotlin 1.5.0</li></ul>
<b>0.2.3</b>  Released: April 5, 2021	<ul style="list-style-type: none"><li>The Project Wizard: improvements in naming modules.</li><li>Ability to use the CocoaPods dependency manager in the Project Wizard for iOS integration.</li><li>Better readability of gradle.properties in new projects.</li><li>Sample tests are no longer generated if "Add sample tests for Shared Module" is unchecked.</li><li>Fixes and other improvements.</li></ul>	<ul style="list-style-type: none"><li>Kotlin 1.4.30</li></ul>

## 1.5.30 的新特性

<b>0.2.2</b>  Released: March 3, 2021	<ul style="list-style-type: none"><li>Ability to open Xcode-related files in Xcode.</li><li>Ability to set up a location for the Xcode project file in the iOS run configuration.</li><li>Support for Android Studio 2020.3.1 Canary 8.</li><li>Fixes and other improvements.</li></ul>	<ul style="list-style-type: none"><li>Kotlin 1.4.30</li></ul>
<b>0.2.1</b>  Released: February 15, 2021	<p>Use this version of the plugin with Android Studio 4.2.</p> <ul style="list-style-type: none"><li>Infrastructure improvements.</li><li>Fixes and other improvements.</li></ul>	<ul style="list-style-type: none"><li>Kotlin 1.4.30</li></ul>
<b>0.2.0</b>  Released: November 23, 2020	<ul style="list-style-type: none"><li>Support for iPad devices.</li><li>Support for custom scheme names that are configured in Xcode.</li><li>Ability to add custom build steps for the iOS run configuration.</li><li>Ability to debug a custom Kotlin/Native binary.</li><li>Simplified the code generated by Kotlin Multiplatform Mobile Wizards.</li><li>Removed support for the <a href="#">Kotlin Android Extensions plugin</a>, which is deprecated in Kotlin 1.4.20.</li><li>Fixed saving physical device configuration after disconnecting from the host.</li><li>Other fixes and improvements.</li></ul>	<ul style="list-style-type: none"><li>Kotlin 1.4.20</li></ul>
<b>0.1.3</b>  Released: October 2, 2020	<ul style="list-style-type: none"><li>Added compatibility with iOS 14 and Xcode 12.</li><li>Fixed naming in platform tests created by the Kotlin Multiplatform Mobile Wizard.</li></ul>	<ul style="list-style-type: none"><li>Kotlin 1.4.10</li><li>Kotlin 1.4.20</li></ul>
<b>0.1.2</b>  Released: September 29, 2020	<ul style="list-style-type: none"><li>Fixed compatibility with <a href="#">Kotlin 1.4.20-M1</a>.</li><li>Enabled error reporting to JetBrains by default.</li></ul>	<ul style="list-style-type: none"><li>Kotlin 1.4.10</li><li>Kotlin 1.4.20</li></ul>
<b>0.1.1</b>  Released: September 10, 2020	<ul style="list-style-type: none"><li>Fixed compatibility with Android Studio Canary 8 and higher.</li></ul>	<ul style="list-style-type: none"><li>Kotlin 1.4.10</li><li>Kotlin 1.4.20</li></ul>
<b>0.1.0</b>  Released: August 31, 2020	<ul style="list-style-type: none"><li>The first version of the Kotlin Multiplatform Mobile plugin. Learn more in the <a href="#">blog post</a>.</li></ul>	<ul style="list-style-type: none"><li>Kotlin 1.4.0</li><li>Kotlin 1.4.10</li></ul>

# Kotlin 插件版本发布

The [IntelliJ Kotlin plugin](#) and [IntelliJ IDEA](#) are on the same release cycle. To speed up the testing and delivery of new features, the plugin and the platform have been moved to the same codebase and ship simultaneously. Kotlin releases happen independently according to the [release cadence](#).

Kotlin and the Kotlin plugin have distinct sets of features:

- Kotlin releases contain language, compiler, and standard library features.
- Kotlin plugin releases introduce only IDE related features. For example, code formatting and debugging tools.

This also affects the versioning of the Kotlin plugin. Releases now have the same version as the simultaneous IntelliJ IDEA release. This creates some limitations that are important to emphasize:

- The EAP version of Kotlin works only with the **stable version** of the IDE. That means that you can't install the Kotlin EAP version to the EAP IDEA release.
- The Kotlin plugin is based on the **previous stable version** of the Kotlin compiler. You can still update the Kotlin version in your project, but some IDE-related features might not be available. We are working on stabilizing the process so that the next versions of the plugin will be based on the latest stable version of the compiler.

You can learn more about new release cadence in this [blog post](#).

## Update to a new release

IntelliJ IDEA and Android Studio suggest updating to a new release once it is out. When you accept the suggestion, it automatically updates the Kotlin plugin to the new version. You can check the Kotlin plugin version in **Tools | Kotlin | Configure Kotlin Plugin Updates**.

If you are migrating to the new feature release, Kotlin plugin's migration tools will help you with the migration.

## Release details

## 1.5.30 的新特性

The following table lists the details of the latest Kotlin plugin releases:

Release info	Release highlights
<b>2021.3</b>  Released: November 30, 2021	<ul style="list-style-type: none"><li>• Better debugging experience</li><li>• Performance improvements</li><li>• Editor inline hints</li><li>• New refactorings and improved inspections and intentions</li></ul> <p>Learn more in:</p> <ul style="list-style-type: none"><li>• <a href="#">What's New in IntelliJ IDEA 2021.3</a></li></ul>
<b>2021.2</b>  Released: July 27, 2021	<ul style="list-style-type: none"><li>• 性能提升</li><li>• Better debugging experience</li><li>• Remote development support</li></ul> <p>Learn more in:</p> <ul style="list-style-type: none"><li>• <a href="#">What's New in IntelliJ IDEA 2021.2</a></li></ul>
<b>2021.1</b>  Released: April 7, 2021	<ul style="list-style-type: none"><li>• 性能提升</li><li>• Evaluation of custom getters during debugging</li><li>• Improved Change Signature refactoring</li><li>• Code completion for type arguments</li><li>• UML diagrams for Kotlin classes</li></ul> <p>Learn more in:</p> <ul style="list-style-type: none"><li>• <a href="#">What's New in IntelliJ IDEA 2021.1</a></li></ul>
<b>2020.3</b>  Released: December 1, 2020	<ul style="list-style-type: none"><li>• 内联重构的新类型</li><li>• 结构化搜索替换</li><li>• EditorConfig 支持</li><li>• Jetpack Compose for Desktop 的项目模板</li></ul> <p>Learn more in:</p> <ul style="list-style-type: none"><li>• <a href="#">IntelliJ IDEA 2020.3 release blog post</a></li></ul>

# Kotlin 路线图

最后修改时间	2021 年 11 月
下次更新时间	2021 年 4 月

欢迎来看 Kotlin 路线图！一窥 Kotlin 团队的工作重点。

## 关键优先事项

这个路线图的目标是给出一个大的图景。这里列出了主要优先事项——我们投入精力最多的领域：

- **快速流转**: 让“变更-测试-调试”周期非常快。
- **新版编译器**: 重写 Kotlin 编译器，针对速度、并行性与统一性进行优化。晚些还会研究插件化。
- **快速流畅的 IDE**: 改善 Kotlin 插件的稳定性与性能。
- **Kotlin 用于 JVM 服务器端开发**: 在整个 Kotlin 生态系统中扩展对服务器端使用场景的支持。
- **Kotlin 多平台移动端**: 改善在移动平台上共享代码的用户体验与特性集。

## 以子系统划分的 Kotlin 路线图

To view the biggest projects we're working on, visit the [YouTrack board](#) or the [Roadmap details](#) table.

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](#)).

### YouTrack board

Visit the [roadmap board](#) in our issue tracker YouTrack 

## 1.5.30 的新特性

In Progress, Open, Submitted		56	Fixed
<b>Kotlin/Native</b>			
KT-42293 Native: provide binary compatibility between incremental releases Native, Middle-end. IR	KT-42294 Improve compilation time Native, Middle-end. IR	KT-42297 Improve exporting Kotlin code to Objective-C Native, ObjC Ex...	KT-44321 Support Apple Silicon without Rosetta 2 Native, Platfor...
KT-49520 Promote new memory manager to Alpha Native	KT-49521 Direct interoperability with Swift Native	KT-42296 Prototype a new garbage collector Native	KT-42296 Prototype a new garbage collector Native
+			
<b>Kotlin Multiplatform</b>			
KT-4329 Improve UX of using Native libraries in Kotlin IDE, Native, Tools. Commoniz...	KT-42466 HMPP: JVM & Android intermediate source set IDE, Android, IDE, Multiplatf..., Tools.	KT-49523 Improve environment setup experience for KMM projects IDE, Multiplatf...	KT-44325 Improve frontend and IDE import stability for Multiplatform projects Frontend, IDE, Multiplatf...
KT-49524 Improve DSL for managing Kotlin/Native binary output Tools. Gradle...	KT-49525 Improve stability and robustness of the multiplatform toolchain IDE, Multiplatf...		KT-44326 Introduce a complex KMM application sample Docs & Examples, KMM Plugin
+			
<b>Kotlin/JVM</b>			
KT-46770 Stabilize JVM-specific experimental features Backend, JVM	KT-46767 Maintain the new JVM IR backend Backend, JVM	KT-46768 Improve new JVM IR backend compilation time Backend, JVM	KT-42287 Make the new JVM IR backend Stable Backend, JVM
KT-49513 Release kotlinx-metadata-jvm as Stable Libraries	KT-49514 Fix issues related to inline classes on the JVM Backend, JVM, I...	KT-17699 Allow private top-level classes or type aliases with same name in different files on JVM Language Design	
+			

## Roadmap details

### 1.5.30 的新特性

子系统	当前聚焦	推迟后续
语言	<ul style="list-style-type: none"><li>• <span style="background-color: #4f81bd; border: 1px solid #4f81bd; padding: 2px 5px;">NEW</span> Research and prototype namespace-based solution for statics and static extensions</li><li>• <span style="background-color: #4f81bd; border: 1px solid #4f81bd; padding: 2px 5px;">NEW</span> Consider supporting inline sealed classes</li><li>• <span style="background-color: #4f81bd; border: 1px solid #4f81bd; padding: 2px 5px;">NEW</span> Allow denotable definitely not-null types</li><li>• <span style="background-color: #4f81bd; border: 1px solid #4f81bd; padding: 2px 5px;">NEW</span> Design and implement solution for <code>toString</code> on objects</li><li>• <span style="background-color: #4f81bd; border: 1px solid #4f81bd; padding: 2px 5px;">NEW</span> Provide modern and performant replacement for <code>Enum.values()</code></li><li>• <span style="background-color: #4f81bd; border: 1px solid #4f81bd; padding: 2px 5px;">NEW</span> Allow implementation by delegation to inlined value of inline class</li><li>• Release <code>OptIn</code> annotations</li><li>• Release builder inference</li><li>• Support sealed (exhaustive) whens</li><li>• Prototype multiple receivers</li></ul>	
编译器核心	<ul style="list-style-type: none"><li>• <span style="background-color: #4f81bd; border: 1px solid #4f81bd; padding: 2px 5px;">NEW</span> Stabilize the K2 Compiler Plugin API</li><li>• <span style="background-color: #4f81bd; border: 1px solid #4f81bd; padding: 2px 5px;">NEW</span> Investigate JS backend for the K2 compiler</li><li>• Release the K2/JVM compiler in Alpha</li><li>• Maintain the current compiler</li><li>• Finalize support for <code>jSpecify</code></li><li>• Improve Kotlin scripting</li></ul>	

## 1.5.30 的新特性

<b>Kotlin/JVM</b>	<ul style="list-style-type: none"><li>•  Release <code>kotlinx-metadata-jvm</code> as Stable</li><li>•  Fix issues related to inline classes on the JVM</li><li>•  Allow private top-level classes or type aliases with same name in different files on JVM</li><li>•  Provide ability to enumerate all direct subclasses of a sealed class at compile-time without reflection</li><li>•  Support method references to functional interface constructors</li><li>•  Support Java synthetic property references</li><li>• Maintain the new JVM IR backend</li><li>• Improve the new JVM IR backend compilation time</li><li>• Stabilize JVM-specific experimental features</li></ul>	
<b>Kotlin/JS</b>	<ul style="list-style-type: none"><li>• Make the new JS IR backend Stable</li><li>• JS IR BE: Add an ability to generate separate JS files for each module</li><li>• Maintain the old JS backend by fixing critical bugs</li></ul>	<ul style="list-style-type: none"><li>•  JS: support ES6 as compilation target</li><li>•  Improve Dukat support</li></ul>
<b>Kotlin/Wasm</b>	<ul style="list-style-type: none"><li>• Implement an experimental version of Kotlin/Wasm compiler backend</li></ul>	Note: Wasm support in <a href="#">Kotlin/Native</a> (through LLVM) will be deprecated and removed

## 1.5.30 的新特性

Kotlin/Native	<ul style="list-style-type: none"><li>•  Promote new memory manager to Alpha</li><li>• Improve compilation time</li><li>• Improve exporting Kotlin code to Objective-C</li><li>• Native: provide binary compatibility between incremental releases</li></ul>	<ul style="list-style-type: none"><li>•  Support building Kotlin/Native for Mac Catalyst (x86-64 and arm64)</li><li>•  Support direct interoperability with Swift</li><li>•  Support running Kotlin/Native-produced binaries on Alpine Linux</li></ul>
Kotlin 多平台	<ul style="list-style-type: none"><li>•  Improve environment setup experience for KMM projects</li><li>•  Improve DSL for managing Kotlin/Native binary output</li><li>•  Improve stability and robustness of the multiplatform toolchain</li><li>• Improve UX of using Native libraries in Kotlin</li></ul>	<ul style="list-style-type: none"><li>•  Improve Kotlin/Native debugging experience</li><li>•  Implement code sharing between JVM and Android</li></ul>
IDE	<ul style="list-style-type: none"><li>•  Make update of compiler/platform versions faster</li><li>•  Improve Multiplatform support</li><li>•  Stabilize Eclipse plugin</li><li>• Prototype the IDE plugin with the new compiler frontend</li><li>• Improve IDE performance</li><li>• Improve debugging experience</li><li>• Improve the New Project wizard</li></ul>	<ul style="list-style-type: none"><li>•  Implement advanced tooling that users have in Java but is missing in Kotlin</li><li>•  Improve the quality of less frequently used features</li></ul>
构建工具	<ul style="list-style-type: none"><li>•  Provide better experience with Kotlin Daemon</li><li>•  Make kapt work out of the box with latest JDKs</li><li>• Improve the performance of Gradle incremental compilation</li><li>• Improve the user experience with the Kotlin Gradle plugin</li></ul>	<ul style="list-style-type: none"><li>•  Improve the quality of Gradle import</li></ul>

## 1.5.30 的新特性

库	<ul style="list-style-type: none"><li>• <b>NEW</b> Release Dokka as Stable</li><li>• <b>NEW</b> Launch <code>kotlinx-kover</code> and productize it further</li><li>• <b>NEW</b> Release <code>kotlinx-serialization</code> 1.4</li><li>• <b>NEW</b> Release <code>kotlinx-coroutines</code> 1.7</li><li>• Stabilize and document <code>atomicfu</code></li><li>• Improve <code>kotlinx-datetime</code> library</li><li>• Support <code>java.nio.Path</code> extension in the standard library</li></ul>	
Website	<ul style="list-style-type: none"><li>• <b>NEW</b> Improve Kotlin Playground</li><li>• <b>NEW</b> Provide infrastructure for documentation localization by community</li><li>• Make the Kotlin website mobile friendly</li><li>• Make the UI and navigation consistent</li><li>• Update community graphic assets to the new Kotlin visual style</li></ul>	

- 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.
- It lists some things that are postponed and will NOT get the team's attention in the nearest future.
- We will adjust our priorities as we go and update the roadmap approximately every six months.



## What's changed since May 2021

### Completed items

We've **completed** the following items from the previous roadmap:

### 1.5.30 的新特性

- Language: Support programmatic creation of annotation class instances
- Language: Stabilize typeOf
- Language: Allow repeating annotations with runtime retention when compiling under Java 8
- Language: Support annotations on class type parameters
- Language: Improve type inference in corner cases for popular Java APIs
- Language: Support for JVM sealed classes
- Compiler core: Inferring types based on self upper bounds
- Compiler core: Work on services for the new compiler to interact with IDE
- Kotlin/Native: Implement safe initialization for top-level properties
- Kotlin/Native: Prototype a new garbage collector
- Multiplatform: Support the Apple silicon target in the Kotlin Multiplatform tooling
- Multiplatform: Improve dependency management for iOS
- IDE: Move the Kotlin plugin to the IntelliJ platform development infrastructure
- Build tools: Decrease time for opening Gradle projects
- Libraries: Improve `kotlinx-serialization` (release 1.3.0)
- Libraries: Improve `kotlinx-coroutines` (release 1.6.0)
- Libraries: Stabilize Duration API in the standard library
- Libraries: Get rid of `!!` after `readLine()` in the standard library
- Libraries: Improve usability of multi-threaded coroutines library for Kotlin/Native
- Website: Revamp Kotlin documentation

## Postponed items

We've decided to **postpone** the following items from the previous roadmap:

- Kotlin/Native: Support building Kotlin/Native for Mac Catalyst (x86-64 and arm64)
- Kotlin/Native: Support direct interoperability with Swift
- Kotlin/Native: Support running Kotlin/Native-produced binaries on Alpine Linux
- Kotlin/JS: Improve Dukat support
- Kotlin/JS: JS: support ES6 as compilation target
- Multiplatform: Improve Kotlin/Native debugging experience
- Multiplatform: Implement code sharing between JVM and Android
- IDE: Implement advanced tooling that users have in Java but is missing in Kotlin
- IDE: Improve the quality of less frequently used features
- Build tools: Improve the quality of Gradle import

## 1.5.30 的新特性

Other postponed items remain in this state from earlier roadmap versions.

## New items

We've **added** the following items to the roadmap:

-  Language: Research and prototype namespace-based solution for statics and static extensions
-  Language: Consider supporting inline sealed classes
-  Language: Allow denotable definitely not-null types
-  Language: Design and implement solution for `toString` on objects
-  Language: Provide modern and performant replacement for `Enum.values()`
-  Language: Allow implementation by delegation to inlined value of inline class
-  Compiler core: Stabilize the K2 Compiler Plugin API
-  Compiler core: Investigate JS backend for the K2 compiler
-  Kotlin/JVM: Release `kotlinx-metadata-jvm` as Stable
-  Kotlin/JVM: Fix issues related to inline classes on the JVM
-  Kotlin/JVM: Allow private top-level classes or type aliases with same name in different files on JVM
-  Kotlin/JVM: Provide ability to enumerate all direct subclasses of a sealed class at compile-time without reflection
-  Kotlin/JVM: Support method references to functional interface constructors
-  Kotlin/JVM: Support Java synthetic property references
-  Kotlin/Native: Promote new memory manager to Alpha
-  Multiplatform: Improve environment setup experience for KMM projects
-  Multiplatform: Improve DSL for managing Kotlin/Native binary output
-  Multiplatform: Improve stability and robustness of the multiplatform toolchain
-  IDE: Make update of compiler/platform versions faster
-  IDE: Improve Multiplatform support
-  IDE: Stabilize Eclipse plugin
-  Build tools: Provide better experience with Kotlin Daemon
-  Build tools: Make kapt work out of the box with latest JDKs
-  Libraries: Release Dokka as Stable
-  Libraries: Launch `kotlinx-kover` and productize it further
-  Libraries: Release `kotlinx-serialization` 1.4
-  Libraries: Release `kotlinx-coroutines` 1.7
-  Website: Improve Kotlin Playground
-  Website: Provide infrastructure for documentation localization by community

## Removed items

We've **removed** the following items from the roadmap:

- ✗ Kotlin/Native: Support interoperability with C++
- ✗ Multiplatform: [Improve Gradle and Compiler error messages](#)
- ✗ Build tools: Improve Kotlin Maven support
- ✗ Libraries: Implement any new multiplatform libraries
- ✗ Libraries: `kotlinx-cli`
- ✗ Libraries: `binary-compatibility-validator`
- ✗ Libraries: `kotlinx-io`

## Items in progress

All other previously identified roadmap items are in progress. You can check their [YouTrack tickets](#) for updates.

# 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
- 作用域函数
- 选择加入要求

# 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的一些（数目也可以为零）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。

# 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。

### 1.5.30 的新特性

请注意，更改可变集合不需要它是以 `var` 定义的变量：写操作修改同一个可变集合对象，因此引用不会改变。但是，如果尝试对 `val` 集合重新赋值，你将收到编译错误。

```
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

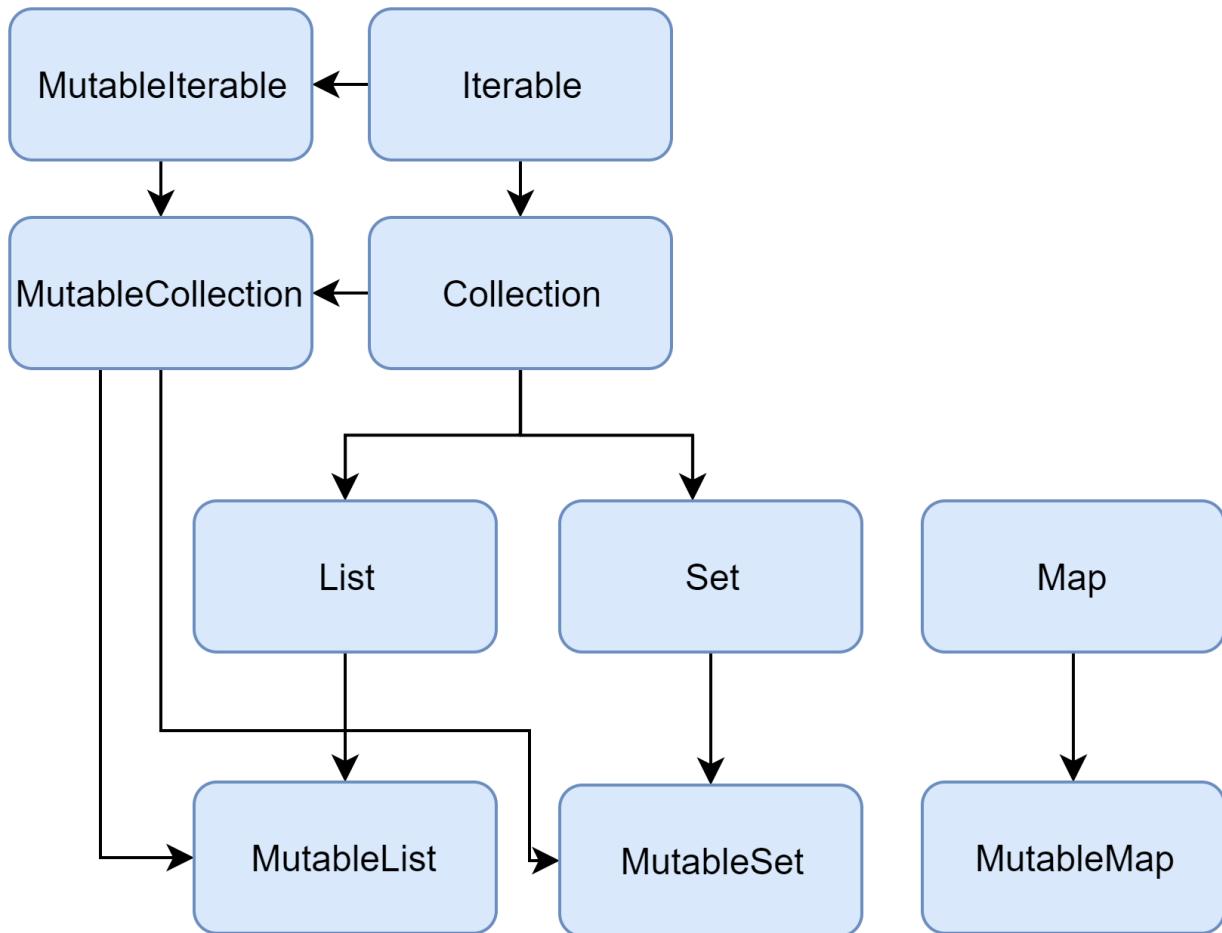
}
```

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：

### 1.5.30 的新特性



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`。

### 1.5.30 的新特性

```
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
}
```

### 1.5.30 的新特性

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，`List` 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。

### 1.5.30 的新特性

```
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`。

`Set` 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。

### 1.5.30 的新特性

```
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
}
```

`Map` 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。

## 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["tw
```

## Create with collection builder functions

Another way of creating a collection is to call a builder function – `buildList()`，`buildSet()`，或 `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
 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构造函数，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。

### 1.5.30 的新特性

```
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].name} 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。

### 1.5.30 的新特性

```
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](#)生成转换结果列表：

```
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
}
```

### 1.5.30 的新特性

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())
 }
 //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)
 }
 //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：

### 1.5.30 的新特性

```
fun main() {
 //sampleStart
 val numbers = listOf("one", "two", "three", "four")
 numbers.forEach {
 println(it)
 }
 //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:")
 while (listIterator.hasPrevious()) {
 print("Index: ${listIterator.previousIndex()}")
 println(", value: ${listIterator.previous()}")
 }
 //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。

### 1.5.30 的新特性

```
fun main() {
//sampleStart
 val numbers = mutableListOf("one", "two", "three", "four")
 val mutableIterator = numbers.iterator()

 mutableIterator.next()
 mutableIterator.remove()
 println("After removal: $numbers")
//sampleEnd
}
```

除了删除元素，`MutableListIterator` 还可以在迭代列表时插入和替换元素。

```
fun main() {
//sampleStart
 val numbers = mutableListOf("one", "four", "four")
 val mutableListIterator = numbers.listIterator()

 mutableListIterator.next()
 mutableListIterator.add("two")
 mutableListIterator.next()
 mutableListIterator.set("three")
 println(numbers)
//sampleEnd
}
```

# 区间与数列

Kotlin 可通过调用 `kotlin.ranges` 包中的 `rangeTo()` 函数及其操作符形式的 `..` 轻松地创建两个值的区间。通常，`rangeTo()` 会辅以 `in` 或 `!in` 函数。

```
fun main() {
 val i = 1
 //sampleStart
 if (i in 1..4) { // 等同于 1 <= i && i <= 4
 print(i)
 }
 //sampleEnd
}
```

整数类型区间（`IntRange`、`LongRange`、`CharRange`）还有一个拓展特性：可以对其进行迭代。这些区间也是相应整数类型的等差数列。

这种区间通常用于 `for` 循环中的迭代。

```
fun main() {
 //sampleStart
 for (i in 1..4) print(i)
 //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)
 //sampleEnd
}
```

也可以通过任意步长（不一定为 1）迭代数字。这是通过 `step` 函数完成的。

### 1.5.30 的新特性

```
fun main() {
//sampleStart
 for (i in 1..8 step 2) print(i)
 println()
 for (i in 8 downTo 1 step 2) print(i)
//sampleEnd
}
```

要迭代不包含其结束元素的数字区间，请使用 `until` 函数：

```
fun main() {
//sampleStart
 for (i in 1 until 10) { // i in 1 到 10, 不含 10
 print(i)
 }
//sampleEnd
}
```

## 区间

区间从数学意义上定义了一个封闭的间隔：它由两个端点值定义，这两个端点值都包含在该区间内。区间是为可比较类型定义的：具有顺序，可以定义任意实例是否在两个给定实例之间的区间内。

区间的主要操作是 `contains`，通常以 `in` 与 `!in` 操作符的形式使用。

要为类创建一个区间，请在区间起始值上调用 `rangeTo()` 函数，并提供结束值作为参数。`rangeTo()` 通常以操作符 `..` 形式调用。

### 1.5.30 的新特性

```
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() {
//sampleStart
 val versionRange = Version(1, 11)..Version(1, 30)
 println(Version(0, 9) in versionRange)
 println(Version(1, 20) in versionRange)
//sampleEnd
}
```

## 数列

如上个示例所示，整数类型的区间（例如 `Int`、`Long` 与 `Char`）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)
//sampleEnd
}
```

要指定数列步长，请在区间上使用 `step` 函数。

### 1.5.30 的新特性

```
fun main() {
//sampleStart
 for (i in 1..8 step 2) print(i)
//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
//sampleEnd
}
```

要创建反向迭代的数列，请在定义其区间时使用 `downTo` 而不是 `..`。

```
fun main() {
//sampleStart
 for (i in 4 downTo 1) print(i)
//sampleEnd
}
```

If you already have a progression, you can iterate it in reverse order with the `reversed` function:

```
fun main() {
//sampleStart
 for (i in (1..4).reversed()) print(i)
//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。

## 1.5.30 的新特性

```
fun main() {
 //sampleStart
 println((1..10).filter { it % 2 == 0 })
 //sampleEnd
}
```

# 序列

除了集合之外，Kotlin 标准库还包含另一种容器类型——序列（`Sequence<T>`）。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

### 1.5.30 的新特性

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。

### 1.5.30 的新特性

```
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。

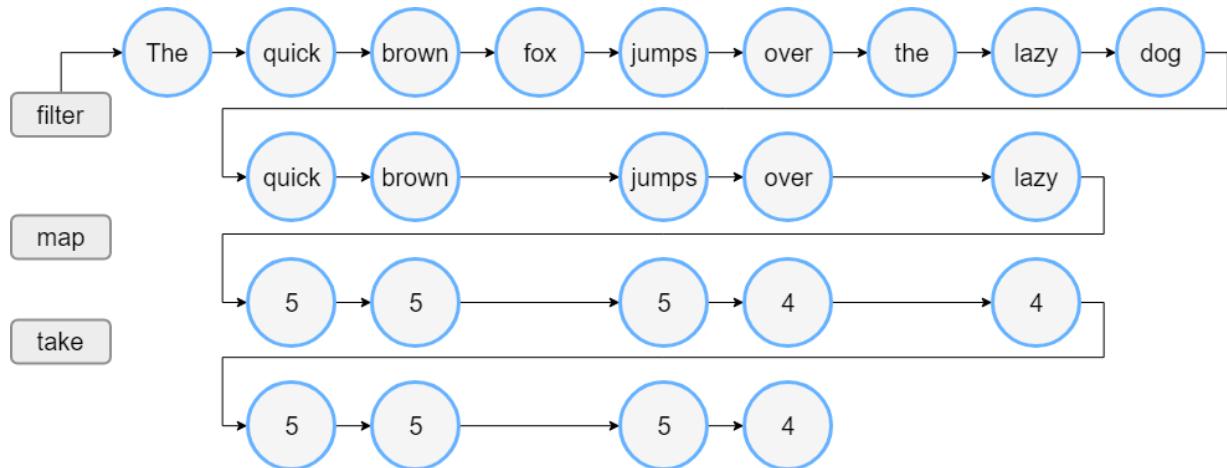
### 1.5.30 的新特性

```
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：

### 1.5.30 的新特性

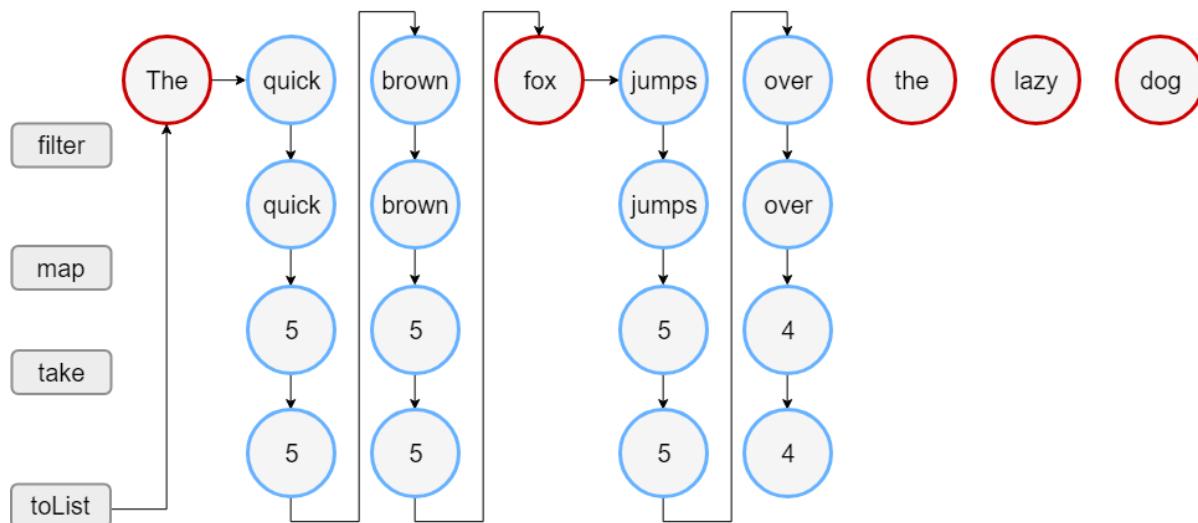
```
fun main() {
//sampleStart
 val words = "The quick brown fox jumps over the lazy dog".split(" ")
 // 将列表转换为序列
 val wordsSequence = words.asSequence()

 val lengthsSequence = wordsSequence.filter { println("filter: $it"); it.length
 .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：



在此示例中，序列处理需要 18 个步骤，而不是 23 个步骤来执行列表操作。

## 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。

## 1.5.30 的新特性

```
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，因此您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 })
 //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。

### 1.5.30 的新特性

```
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。

### 1.5.30 的新特性

```
fun main() {
//sampleStart
 val colors = listOf("red", "brown", "grey")
 val animals = listOf("fox", "bear", "wolf")

 println(colors.zip(animals) { color, animal -> "The ${animal.replaceFirstChar {
//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
}
```

### 1.5.30 的新特性

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() }, value
 //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.f
 //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

### 1.5.30 的新特性

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。

当使用默认参数调用时，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`。

### 1.5.30 的新特性

```
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 = ""))
//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。

### 1.5.30 的新特性

```
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 % 2 == 0 }
 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。

## 1.5.30 的新特性

```
fun main() {
//sampleStart
 val numbers = listOf("one", "two", "three", "four")

 val filteredIdx = numbers.filterIndexed { index, s -> (index != 0) && (s.length > 3)
 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

### 1.5.30 的新特性

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。

### 1.5.30 的新特性

```
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。

```
fun main() {
 //sampleStart
 val numbers = listOf("one", "two", "three", "four", "five")

 println(numbers.groupBy { it.first().uppercase() })
 println(numbers.groupBy(keySelector = { it.first() }, valueTransform = { it.uppercase() }))
 //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。

```
fun main() {
 //sampleStart
 val numbers = listOf("one", "two", "three", "four", "five", "six")
 println(numbers.groupingBy { it.first() }.eachCount())
 //sampleEnd
}
```

## 1.5.30 的新特性

# 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作入的。

```
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
}
```

### 1.5.30 的新特性

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第一个元素。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。

### 1.5.30 的新特性

```
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。

## 1.5.30 的新特性

```
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()`。

### 1.5.30 的新特性

```
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 un
//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`。

### 1.5.30 的新特性

```
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-null element. Use the counterpart `firstNotNullOfOrNull()` to return null in this case.

```
fun main() {
//sampleStart
 val list = listOf(0, "true", false)
 // Converts each element to string and returns the first one that has required
 val longEnough = list.firstNotNullOf { item -> item.toString().takeIf { it.leng
 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。

### 1.5.30 的新特性

```
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` 接口的继承者定义的。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，可以让这个类型继承 `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(
 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

### 1.5.30 的新特性

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 -
 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
}
```

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。

### 1.5.30 的新特性

```
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。

### 1.5.30 的新特性

```
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。

## 1.5.30 的新特性

```
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.

### 1.5.30 的新特性

```
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。

### 1.5.30 的新特性

```
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()`。

## 1.5.30 的新特性

```
fun main() {
//sampleStart
 val numbers = listOf(5, 2, 10, 4)
 val sumEven = numbers.foldIndexed(0) { idx, sum, element -> if (idx % 2 == 0) s
 println(sumEven)

 val sumEvenRight = numbers.foldRightIndexed(0) { idx, element, sum -> if (idx % 2 == 0) s
 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"))
 println(runningFoldSum.mapIndexed(transform).joinToString("\n", "Sum of first 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
}
```

### 1.5.30 的新特性

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) // 去除了第一个 `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。

### 1.5.30 的新特性

```
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：

- `getOrDefault()` 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.getOrDefault(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。

### 1.5.30 的新特性

```
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
}
```

### 1.5.30 的新特性

## 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> {
 //sampleEnd
 })
}
```

### 1.5.30 的新特性

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).

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

### 1.5.30 的新特性

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：

## 1.5.30 的新特性

```
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。

要查找两个集合中都存在的元素（交集），请使用 `intersect()`。要查找另一个集合中不存在的集合元素（差集），请使用 `subtract()`。这两个函数也能以中缀形式调用，例如，`a intersect b`。

```
fun main() {
 //sampleStart
 val numbers = setOf("one", "two", "three")

 println(numbers union setOf("four", "five"))
 println(setOf("four", "five") union numbers)

 println(numbers intersect setOf("two", "one"))
 println(numbers subtract setOf("three", "four"))
 println(numbers subtract setOf("four", "three")) // 相同的输出
 //sampleEnd
}
```

You can also apply `union`, `intersect`, and `subtract` to `List`. However, their result is *always* a `Set`, even on lists. 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)
 println(list1 intersect list2) // result on two lists is a Set
 println(list1 union list2) // equal elements are merged into one
 //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特定于 `map` 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：

- `getOrDefault()` 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 % 2 == 0 }
 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。

### 1.5.30 的新特性

```
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。

### 1.5.30 的新特性

```
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: ${numbersM
 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。

### 1.5.30 的新特性

```
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第一个条目。

```
fun main() {
//sampleStart
 val numbersMap = mutableMapOf("one" to 1, "two" to 2, "three" to 3, "threeAgain"
 numbersMap.keys.remove("one")
 println(numbersMap)
 numbersMap.values.remove(3)
 println(numbersMap)
//sampleEnd
}
```

`minusAssign` (`-=`) 操作符也可用于可变 Map。

### 1.5.30 的新特性

```
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
}
```

## 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`。

这些函数基本上做了同样的事情：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。

下面是作用域函数的典型用法：

```
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。

### 1.5.30 的新特性

```
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。

由于作用域函数的相似性质，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和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及其约定用法之间的区别。

## 函数选择

为了帮助你选择合适的作用域函数，我们提供了它们之间的主要区别表。

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

The detailed information about the differences 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

### 1.5.30 的新特性

- 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`

不同函数的使用场景存在重叠，你可以根据项目或团队中使用的特定约定选择函数。

尽管作用域函数是使代码更简洁的一种方法，但请避免过度使用它们：这会降低代码的可读性并可能导致错误。避免嵌套作用域函数，同时链式调用它们时要小心：此时很容易对当前上下文对象及 `this` 或 `it` 的值感到困惑。

## 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`

在作用域函数的 `lambda` 表达式里，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访问上下文对象：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，因此我们将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

### 1.5.30 的新特性

`run`、`with` 以及 `apply` 通过关键字 `this` 引用上下文对象。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 = "")\n\nfun main() {\n //sampleStart\n val adam = Person("Adam").apply {\n age = 20\n city = "London"\n }\n println(adam)\n //sampleEnd\n}
```

### it

反过来，`let` 及 `also` 将上下文对象作为 `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，使用 `it` 作为上下文对象会更好。若在代码块中使用多个变量，则 `it` 也更好。

### 1.5.30 的新特性

```
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
}
```

此外，当将上下文对象作为参数传递时，可以为上下文对象指定在作用域内的自定义名称。

```
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。

### 1.5.30 的新特性

- `let`、`run` 及 `with` 返回 lambda 表达式结果。

这两个选项使你可以根据在代码中的后续操作来选择适当的函数。

## 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：你可以继续在同一个对象上进行链式函数调用。

```
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 writeToFile(message: String) {
 println("INFO: $message")
}

fun main() {
 //sampleStart
 fun getRandomInt(): Int {
 return Random.nextInt(100).also {
 writeToFile("getRandomInt() generated value $it")
 }
 }

 val i = getRandomInt()
 //sampleEnd
}
```

## Lambda 表达式结果

### 1.5.30 的新特性

`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

为了帮助你为你的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定义通用使用风格的约定用法。

### 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：

### 1.5.30 的新特性

```
fun main() {
//sampleStart
 val numbers = mutableListOf("one", "two", "three", "four", "five")
 val resultList = numbers.map { it.length }.filter { it > 3 }
 println(resultList)
//sampleEnd
}
```

使用 `let`，可以写成这样：

```
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
}
```

若代码块仅包含以 `it` 作为参数的单个函数，则可以使用方法引用( `::` )代替 lambda 表达式：

```
fun main() {
//sampleStart
 val numbers = mutableListOf("one", "two", "three", "four", "five")
 numbers.map { it.length }.filter { it > 3 }.let(::println)
//sampleEnd
}
```

`let` 经常用于仅使用非空值执行代码块。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。

### 1.5.30 的新特性

```
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
}
```

使用 `let` 的另一种情况是引入作用域受限的局部变量以提高代码的可读性。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

一个非扩展函数：上下文对象作为参数传递，但是在 `lambda` 表达式内部，它可以作为接收者（`this`）使用。返回值是 `lambda` 表达式结果。

我们建议使用 `with` 来调用上下文对象上的函数，而不使用 `lambda` 表达式结果。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
}
```

### 1.5.30 的新特性

`with` 的另一个使用场景是引入一个辅助对象，其属性或函数将用于计算一个值。

```
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` 和 `with` 做同样的事情，但是调用方式和 `let` 一样——作为上下文对象的扩展函数。

当 lambda 表达式同时包含对象初始化和返回值的计算时，`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)
}
```

### 1.5.30 的新特性

除了在接收者对象上调用 `run` 之外，还可以将其用作非扩展函数。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。

```
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。

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 `apply`。`apply` 的常见情况是对象配置。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
}
```

将接收者作为返回值，你可以轻松地将 `apply` 包含到调用链中以进行更复杂的处理。

## 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。

### 1.5.30 的新特性

`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
}
```

## takelf 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。

当以提供的谓词在对象上进行调用时，若该对象与谓词匹配，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。反之，`takeUnless` 如果不匹配谓词，则返回对象，如果匹配则返回 `null`。该对象作为 lambda 表达式参数 (`it`) 来访问。

```
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安全调用 (`?.`)，因为他们的返回值是可为空的。

### 1.5.30 的新特性

```
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` 与作用域函数一起特别有用。一个很好的例子是用 `let` 链接它们，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
}
```

没有标准库函数时，相同的函数看起来是这样的：

```
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
}
```

# 选择加入要求

要求选择加入的注解 `@RequiresOptIn` 与 `@OptIn` 是 Beta 版。It is almost stable, but migration steps may be required in the future. We'll do our best to minimize any changes you have to make. See the details in the [Beta status of the opt-in requirements](#) section.



Kotlin 标准库提供了一种机制，用于要求并明确同意使用 API 的某些元素。通过这种机制，库开发人员可以将使用其 API 需要选择加入的特定条件告知用户，例如，如果某个 API 处于实验状态，并且将来可能会更改。

为了避免潜在的问题，编译器会向此类 API 的用户发出警告，告知他们这些条件，并要求他们在使用 API 之前选择加入。

## 选择使用 API

如果库作者将一个库的 API 声明标记为 [要求选择加入](#)，你应该明确同意在代码中使用它。有多种方式可以选择加入使用此类 API，所有方法均不受技术限制。你可以自由选择最适合自己的方式。

### 传播选择加入

在使用供第三方（库）使用的 API 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 API 主体声明中标注[要求选择加入的注解](#)。这可以让你使用要求选择加入的 API 元素。

```
// 库代码
@RequiresOptIn(message = "This API is experimental. It may be changed in the future")
@Retention(AnnotationRetention.BINARY)
@Target(AnnotationTarget.CLASS, AnnotationTarget.FUNCTION)
annotation class MyDateTime // 要求选择加入的注解

@MyDateTime
class DateProvider // 要求选择加入的类
```

### 1.5.30 的新特性

```
// 客户端代码
fun getYear(): Int {
 val dateProvider: DateProvider // 错误: DateProvider 要求选择加入
 // ...
}

@MyDateTime
fun getDate(): Date {
 val dateProvider: DateProvider // OK: 该函数也要求选择加入
 // ...
}

fun displayDate() {
 println(getDate()) // 错误: getDate() 要求选择加入
}
```

如本例所示，带注释的函数看起来是 `@MyDateTime` API 的一部分。因此，这种选择加入会将选择加入的要求传播到客户端代码；其客户将看到相同的警告消息，并且也必须同意。

Implicit usages of APIs that require opt-in also require opt-in. If an API element doesn't have an opt-in requirement annotation but its signature includes a type declared as requiring opt-in, its usage will still raise a warning. See the example below.

```
// Client code
fun getDate(dateProvider: DateProvider): Date { // Error: DateProvider requires opt-in
 // ...
}

fun displayDate() {
 println(getDate()) // Warning: the signature of getDate() contains DateProvider
}
```

要使用多个需要选择加入的API，请在声明中标记所有需要选择加入的注解。

## 非传播的选择加入

在不公开其自身API的模块（例如应用程序）中，你可以选择使用 API 而无需将选择加入的要求传播到代码中。这种情况下，请使用 `@OptIn` 标记你的声明，并以要求选择加入的注解作为参数：

### 1.5.30 的新特性

```
// 库代码
@RequiresOptIn(message = "This API is experimental. It may be changed in the future")
@Retention(AnnotationRetention.BINARY)
@Target(AnnotationTarget.CLASS, AnnotationTarget.FUNCTION)
annotation class MyDateTime // 要求选择加入的注解

@MyDateTime
class DateProvider // 要求选择加入的类
```

```
//客户端代码
@OptIn(MyDateTime::class)
fun getDate(): Date { // 使用 DateProvider; 不传播选择加入的要求
 val dateProvider: DateProvider
 // ...
}

fun displayDate() {
 println(getDate()) // OK: 不要求选择加入
}
```

当有人调用函数 `getDate()` 时，不会通知他们函数主体中使用的选择加入 API 的要求。

Note that if `@OptIn` applies to the declaration whose signature contains a type declared as requiring opt-in, the opt-in will still propagate:

```
// Client code
@OptIn(MyDateTime::class)
fun getDate(dateProvider: DateProvider): Date { // Has DateProvider as a part of a
 // ...
}

fun displayDate() {
 println(getDate()) // Warning: getDate() requires opt-in
}
```

要在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)
```

## 模块范围的选择加入

### 1.5.30 的新特性

The `-opt-in` compiler option is available since Kotlin 1.6.0. For earlier Kotlin versions, use `-Xopt-in`.



如果你不想在使用要求选择加入 API 的每个地方都添加注解，则可以为整个模块选择加入这些 API。要选择在模块中使用 API，请使用参数 `-opt-in` 进行编译，使用 `-opt-in = org.mylibrary.OptInAnnotation` 指定该 API 使用的要求选择加入注解的标准名称。使用此参数进行编译的效果与模块中每个声明都有注解 `@OptIn(OptInAnnotation::class)` 的效果相同。

如果使用 Gradle 构建模块，可以添加如下参数：

#### 【Kotlin】

```
tasks.withType<org.jetbrains.kotlin.gradle.tasks.KotlinCompile>().configureEach {
 kotlinOptions.freeCompilerArgs += "-opt-in=org.mylibrary.OptInAnnotation"
}
```

#### 【Groovy】

```
tasks.withType(org.jetbrains.kotlin.gradle.tasks.KotlinCompile).configureEach {
 kotlinOptions {
 freeCompilerArgs += "-opt-in=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】

### 1.5.30 的新特性

```
sourceSets {
 all {
 languageSettings {
 optIn('org.mylibrary.OptInAnnotation')
 }
 }
}
```

对于 Maven，它将是：

```
<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。

## API 要求选择加入

### 创建选择加入要求的注解

如果想获得使用者使用你的模块 API 的明确同意，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](#) 注解：

```
@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以下几个要求：

### 1.5.30 的新特性

- `BINARY` 或 `RUNTIME` `retention`
- `targets` 中没有 `EXPRESSION`、`FILE`、`TYPE` 及 `TYPE_PARAMETER`
- 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。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，你可以提供一个 `message` 来通知用户有关使用该 API 的特定条件。编译器会将其显示给使用该 API 但未选择加入的用户。

```
@RequiresOptIn(level = RequiresOptIn.Level.WARNING, message = "This API is experimental")
@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。这使你的用户可以更安全地使用 API：他们只能使用其明确接受的功能。这也使你可以独立地从功能中删除选择加入的要求。

## 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:

- Overriding methods can only have opt-in annotations that are present on their basic declarations.
- You cannot annotate a backing field or a getter of a property, just the property itself.

### 1.5.30 的新特性

- You cannot annotate a local variable or a value parameter.

## 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，请从声明中删除其要求选择加入的注解。客户端将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，以便与现有的客户端代码保持兼容。

为了让 API 用户相应地更新其模块（从代码中删除注解并重新编译），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 yo
@RequiresOptIn
annotation class ExperimentalDateTime
```

## 选择加入要求的 Beta 状态

选择加入要求的机制目前是 [Beta 版](#)。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.

为了让使用注解 `@OptIn` 和 `@RequiresOptIn` 的用户了解其稳定前状态，编译器会在编译代码时发出警告：

```
This annotation should be used with the compiler argument '-opt-
in=kotlin.RequiresOptIn'
```

如需移除警告，请添加编译器参数 `-opt-in=kotlin.RequiresOptIn`。

Learn more about recent changes to opt-in requirements in [this KEP](#).

# 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`)
- 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 是一门仅在标准库中提供最基本底层 API 以便各种其他库能够利用协程的语言。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](#)

### 1.5.30 的新特性

- 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 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.

### 1.5.30 的新特性

`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 in 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 be only 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

### 1.5.30 的新特性

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:

### 1.5.30 的新特性

```
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:

### 1.5.30 的新特性

```
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:

### 1.5.30 的新特性

```
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

运行以下代码：

```
import kotlinx.coroutines.*

//sampleStart
fun main() = runBlocking {
 repeat(100_000) { // 启动大量的协程
 launch {
 delay(5000L)
 print(".")
 }
 }
//sampleEnd
```

可以在[这里](#)获取完整代码。



### 1.5.30 的新特性

它启动了 10 万个协程，并且在 5 秒钟后，每个协程都输出一个点。

现在，尝试使用线程来实现 (remove runBlocking , replace launch with thread , and replace delay with Thread.sleep )。会发生什么？（很可能你的代码会产生某种内存不足的错误）

## 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.
```

### 1.5.30 的新特性

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。

## 使计算代码可取消

### 1.5.30 的新特性

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：

## 1.5.30 的新特性

```
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：

### 1.5.30 的新特性

```
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-cance
 }
 }
 }
 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：

### 1.5.30 的新特性

```
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 o
```

`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：

### 1.5.30 的新特性

```
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 this counter from its `close` function. Let us run a lot of coroutines with the small timeout try acquire this resource from inside of the `withTimeout` block after a bit of delay and release it from outside.

### 1.5.30 的新特性

```
import kotlinx.coroutines.*

//sampleStart
var acquired = 0

class Resource {
 init { acquired++ } // Acquire the resource
 fun close() { acquired-- } // Release the resource
}

fun main() {
 runBlocking {
 repeat(100_000) { // Launch 100K 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
 }
 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 timeouts in this example to actually see non-zero values.

Note, that incrementing and decrementing `acquired` counter here from 100K coroutines is completely safe, since it always happens from the same main thread. More on that will be explained in the chapter on coroutine context.



To workaround this problem you can store a reference to the resource in the variable as opposed to returning it from the `withTimeout` block.

## 1.5.30 的新特性

```
import kotlinx.coroutines.*

var acquired = 0

class Resource {
 init { acquired++ } // Acquire the resource
 fun close() { acquired-- } // Release the resource
}

fun main() {
//sampleStart
 runBlocking {
 repeat(100_000) { // Launch 100K 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 i
 }
 // 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：

### 1.5.30 的新特性

```
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。

### 1.5.30 的新特性

```
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：

### 1.5.30 的新特性

```
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

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：

## 1.5.30 的新特性

```
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。



这种带有异步函数的编程风格仅供参考，因为这在其它编程语言中是一种受欢迎的风格。在 Kotlin 的协程中使用这种风格是**强烈不推荐的**，原因如下所述。



### 1.5.30 的新特性

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器会捕获这个异常，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 `COROUTINE_SCOPE` 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。

## 1.5.30 的新特性

```
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：

## 1.5.30 的新特性

```
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。



### 1.5.30 的新特性

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 `COROUTINE_SCOPE` 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 `COROUTINE_SCOPE`。`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。

### 1.5.30 的新特性

```
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

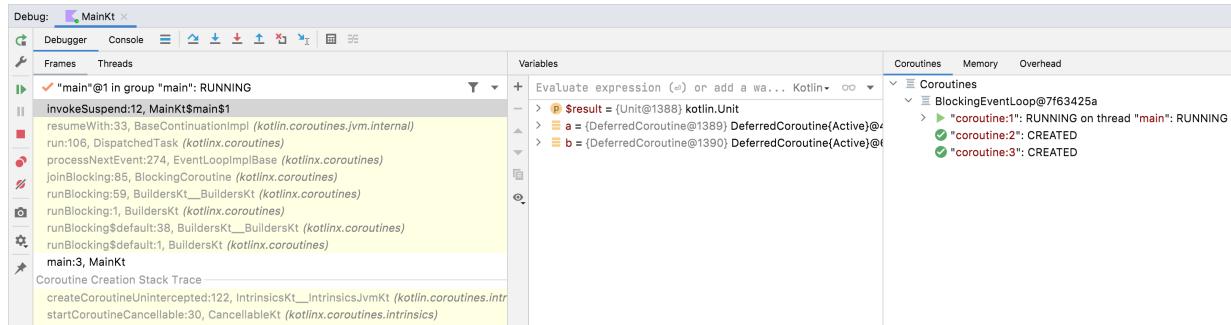
### 1.5.30 的新特性

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：

### 1.5.30 的新特性

```
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` 函数在方括号中打印了线程的名字，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](#)）：

### 1.5.30 的新特性

```
import kotlinx.coroutines.*

fun log(msg: String) = println("[${Thread.currentThread().name}] $msg")

fun main() {
 //sampleStart
 newSingleThreadContext("Ctx1").use { ctx1 ->
 newSingleThreadContext("Ctx2").use { ctx2 ->
 runBlocking(ctx1) {
 log("Started in ctx1")
 withContext(ctx2) {
 log("Working in ctx2")
 }
 log("Back to ctx1")
 }
 }
 }
 //sampleEnd
}
```

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：

### 1.5.30 的新特性

```
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 show 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.

### 1.5.30 的新特性

```
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 can")
 }
 }
 delay(500)
 request.cancel() // 取消请求 (request) 的执行
 delay(1000) // 延迟一秒钟来看看发生了什么
 println("main: Who has survived request cancellation?")
 //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
job1: I am not affected by cancellation of the request
main: Who has survived request cancellation?
```

## 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：

### 1.5.30 的新特性

```
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 still 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：

### 1.5.30 的新特性

```
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")
 252
 }
 val v2 = async(CoroutineName("v2coroutine")) {
 delay(1000)
 log("Computing v2")
 6
 }
 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：

### 1.5.30 的新特性

```
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.....
```

### 1.5.30 的新特性

现在，我们可以使用定义的 `scope` 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。

### 1.5.30 的新特性

```
import kotlinx.coroutines.*

class Activity {
 private val mainScope = CoroutineScope(Dispatchers.Default) // use Default for
 // thread pool

 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()`。

### 1.5.30 的新特性

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
 val job = launch(Dispatchers.Default + threadLocal.asContextElement(value = "la
 println("Launch start, current thread: ${Thread.currentThread()}, thread lo
 yield()
 println("After yield, current thread: ${Thread.currentThread()}, thread loc
 }
 job.join()
 println("Post-main, current thread: ${Thread.currentThread()}, thread local val

//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：

### 1.5.30 的新特性

```
Pre-main, current thread: Thread[main @coroutine#1,5,main], thread local value: 'ma
Launch start, current thread: Thread[DefaultDispatcher-worker-1 @coroutine#2,5,main]
After yield, current thread: Thread[DefaultDispatcher-worker-2 @coroutine#2,5,main]
Post-main, current thread: Thread[main @coroutine#1,5,main], thread local value: 'm
```

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：

### 1.5.30 的新特性

```
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

### 1.5.30 的新特性

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：

- 名为 [flow](#) 的 [Flow](#) 类型构建器函数。
- `flow { ... }` 构建块中的代码可以挂起。
- 函数 `simple` 不再标有 `suspend` 修饰符。

### 1.5.30 的新特性

- 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：

### 1.5.30 的新特性

```
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。通过它自己，`simple()` 调用会尽快返回且不会进行任何等待。该流在每次收集的时候启动，这就是为什么当我们再次调用 `collect` 时我们会看到“Flow started”。

## 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
```

### 1.5.30 的新特性

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。还有其他构建器使流的声明更简单：

- `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。

因此，从流中打印从 1 到 3 的数字的示例可以写成：

```
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

### 1.5.30 的新特性

可以使用操作符转换流，就像使用集合与序列一样。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`。流与序列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，每一行每秒出现一次：

```
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：

### 1.5.30 的新特性

```
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：

### 1.5.30 的新特性

```
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：

## 1.5.30 的新特性

```
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：

### 1.5.30 的新特性

```
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：

### 1.5.30 的新特性

```
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。

## 1.5.30 的新特性

### withContext 发出错误

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{Action}
but emission happened in [CoroutineId(1), "coroutine#1":DispatchedCoroutine{Action}
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：

### 1.5.30 的新特性

```
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：

## 1.5.30 的新特性

```
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：

### 1.5.30 的新特性

```
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

### 1.5.30 的新特性

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

### 1.5.30 的新特性

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。

## 1.5.30 的新特性

### 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。



## 1.5.30 的新特性

```
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
 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
 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:

### 1.5.30 的新特性

```
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，所以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，并为每个整数调用 `requestFlow`，如下所示：

```
(1..3).asFlow().map { requestFlow(it) }
```

然后我们得到了一个包含流的流（`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

连接模式由 `flatMapConcat` 与 `flattenConcat` 操作符实现。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：

### 1.5.30 的新特性

```
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

另一种展平模式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`）。

### 1.5.30 的新特性

```
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](#)”小节中已经讨论过），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。

## 1.5.30 的新特性

```
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` 在一个新值到来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，如果我们要在块中调用诸如 `delay` 之类的挂起函数，这将会被表现出来。



## 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。

## 1.5.30 的新特性

### 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：

## 1.5.30 的新特性

```
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。

### 1.5.30 的新特性

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：

## 1.5.30 的新特性

```
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：

### 1.5.30 的新特性

```
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。

### 1.5.30 的新特性

```
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
}
```

### 1.5.30 的新特性

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 exceptionally")
 .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。

### 1.5.30 的新特性

```
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。

### 1.5.30 的新特性

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
```

### 1.5.30 的新特性

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：

## 1.5.30 的新特性

```
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: BlockingCor
```

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；

### 1.5.30 的新特性

```
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: BlockingCor
```

## 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
```

### 1.5.30 的新特性

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: BlockingCor
```

## 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背。

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：

### 1.5.30 的新特性

```
import kotlinx.coroutines.*
import kotlinx.coroutines.channels.*

fun CoroutineScope.produceSquares(): ReceiveChannel<Int> = produce {
 for (x in 1..5) send(x * x)
}

fun main() = runBlocking {
 //sampleStart
 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> = prod
 for (x in numbers) send(x * x)
}
```

主要的代码启动并连接了整个管道：

### 1.5.30 的新特性

```
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> = prod
 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
}
```

### 1.5.30 的新特性

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。



### 1.5.30 的新特性

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：

### 1.5.30 的新特性

```
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：

### 1.5.30 的新特性

```
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）：

### 1.5.30 的新特性

```
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。

### 1.5.30 的新特性

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“兵”和“兵”都从共享的“桌子”通道接收到这个“球”元素。

## 1.5.30 的新特性

```
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



“兵”协程首先被启动，所以它首先接收到了球。甚至虽然“兵”协程在将球发送会桌子以后立即开始接收，但是球还是被“兵”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

### 1.5.30 的新特性

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.*

fun main() = runBlocking<Unit> {
 val tickerChannel = ticker(delayMillis = 100, initialDelayMillis = 0) // 创建计时器通道
 var nextElement = withTimeoutOrNull(1) { tickerChannel.receive() }
 println("Initial element is available immediately: $nextElement") // no initial delay

 nextElement = withTimeoutOrNull(50) { tickerChannel.receive() } // all subsequent elements are delayed
 println("Next element is not ready in 50 ms: $nextElement")

 nextElement = withTimeoutOrNull(60) { tickerChannel.receive() }
 println("Next element is ready in 100 ms: $nextElement")

 // 模拟大量消费延迟
 println("Consumer pauses for 150ms")
 delay(150)
 // 下一个元素立即可用
 nextElement = withTimeoutOrNull(1) { tickerChannel.receive() }
 println("Next element is available immediately after large consumer delay: $nextElement")
 // 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(60) { tickerChannel.receive() }
 println("Next element is ready in 50ms after consumer pause in 150ms: $nextElement")

 tickerChannel.cancel() // 表明不再需要更多的元素
}
```

可以在[这里](#)获取完整代码。



它的打印如下：

### 1.5.30 的新特性

```
Initial element is available immediately: kotlin.Unit
Next element is not ready in 50 ms: null
Next element is ready in 100 ms: kotlin.Unit
Consumer pauses for 150ms
Next element is available immediately after large consumer delay: kotlin.Unit
Next element is ready in 50ms after consumer pause in 150ms: 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` 与 `actor`）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)`.



### 1.5.30 的新特性

```
import kotlinx.coroutines.*

@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.defaultUncaughtExceptionHandler().handleException()
 }
 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")
 }
}
```

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-2 @coroutine#2" java.lang.IndexOutOfBoundsException: Index: 0, Size: 0
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。当调用处理器的时候，协程已经完成并带有相应的异常。通常，该处理器用于记录异常，显示某种错误消息，终止和（或）重新启动应用程序。

### 1.5.30 的新特性

在 JVM 中可以重定义一个全局的异常处理器来将所有的协程通过 `ServiceLoader` 注册到 `CoroutineExceptionHandler`。全局异常处理器就如同 `Thread.defaultUncaughtExceptionHandler` )一样，在没有更多的指定的异常处理器被注册的时候被使用。在 Android 中，`uncaughtExceptionPreHandler` 被设置在全局协程异常处理器中。

`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器忽略，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。

### 1.5.30 的新特性

在这些示例中，`CoroutineExceptionHandler` 总是被设置在由 `GlobalScope` 启动的协程中。将异常处理器设置在 `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器，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 u
 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：

### 1.5.30 的新特性

```
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.ArithmetricException
```

## 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.suppressedExceptions}")
 }
 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，它将被抑制
 } finally {
 throw ArithmetricException() // 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。



注意：上面的代码将只在 JDK7 以上支持 `suppressed` 异常的环境中才能正确工作。



### 1.5.30 的新特性

这段代码的输出如下：

```
CoroutineExceptionHandler got java.io.IOException with suppressed [java.lang.Arithm
```

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 inner = launch { // 该栈内的协程都将被取消
 launch {
 launch {
 throw IOException() // 原始异常
 }
 }
 }
 try {
 inner.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：

## 1.5.30 的新特性

```
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 t
 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 c
 }
 }
 // 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

### 1.5.30 的新特性

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

### 1.5.30 的新特性

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。

## 1.5.30 的新特性

```
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：

### 1.5.30 的新特性

```
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，但我们最后仍然没有得到“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

### 1.5.30 的新特性

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

### 1.5.30 的新特性

**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

### 1.5.30 的新特性

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。



### 1.5.30 的新特性

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。

## Actors

一个 `actor` 是由协程、被限制并封装到该协程中的状态以及一个与其它协程通信的 `通道` 组合而成的一个实体。一个简单的 `actor` 可以简单的写成一个函数，但是一个拥有复杂状态的 `actor` 更适合由类来表示。

有一个 `actor` 协程构建器，它可以方便地将 `actor` 的邮箱通道组合到其作用域中（用来接收消息）、组合发送 `channel` 与结果集对象，这样对 `actor` 的单个引用就可以作为其句柄持有。

使用 `actor` 的第一步是定义一个 `actor` 要处理的消息类。Kotlin 的 [密封类](#)很适合这种场景。我们使用 `IncCounter` 消息（用来递增计数器）和 `GetCounter` 消息（用来获取值）来定义 `CounterMsg` 密封类。后者需要发送回复。[CompletableDeferred](#) 通信原语表示未来可知（可传达）的单个值，这里被用于此目的。

```
// 计数器 Actor 的各种类型
sealed class CounterMsg
object IncCounter : CounterMsg() // 递增计数器的单向消息
class GetCounter(val response: CompletableDeferred<Int>) : CounterMsg() // 携带回复的
```

接下来我们定义一个函数，使用 `actor` 协程构建器来启动一个 `actor`:

```
// 这个函数启动一个新的计数器 actor
fun CoroutineScope.counterActor() = actor<CounterMsg> {
 var counter = 0 // actor 状态
 for (msg in channel) { // 即将到来消息的迭代器
 when (msg) {
 is IncCounter -> counter++
 is GetCounter -> msg.response.complete(counter)
 }
 }
}
```

`main` 函数代码很简单：

### 1.5.30 的新特性

```
import kotlinx.coroutines.*
import kotlinx.coroutines.channels.*
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个动作的次数
 val time = measureTimeMillis {
 coroutineScope { // 协程的作用域
 repeat(n) {
 launch {
 repeat(k) { action() }
 }
 }
 }
 println("Completed ${n * k} actions in $time ms")
 }

 // 计数器 Actor 的各种类型
 sealed class CounterMsg
 object IncCounter : CounterMsg() // 递增计数器的单向消息
 class GetCounter(val response: CompletableDeferred<Int>) : CounterMsg() // 携带回复的消息

 // 这个函数启动一个新的计数器 actor
 fun CoroutineScope.counterActor() = actor<CounterMsg> {
 var counter = 0 // actor 状态
 for (msg in channel) { // 即将到来消息的迭代器
 when (msg) {
 is IncCounter -> counter++
 is GetCounter -> msg.response.complete(counter)
 }
 }
 }
}

//sampleStart
fun main() = runBlocking<Unit> {
 val counter = counterActor() // 创建该 actor
 withContext(Dispatchers.Default) {
 massiveRun {
 counter.send(IncCounter)
 }
 }
 // 发送一条消息以用来从一个 actor 中获取计数值
 val response = CompletableDeferred<Int>()
 counter.send(GetCounter(response))
 println("Counter = ${response.await()}")
 counter.close() // 关闭该actor
}
//sampleEnd
```

### 1.5.30 的新特性

可以在[这里](#)获取完整代码。



actor 本身执行时所处上下文（就正确性而言）无关紧要。一个 actor 是一个协程，而一个协程是按顺序执行的，因此将状态限制到特定协程可以解决共享可变状态的问题。实际上，actor 可以修改自己的私有状态，但只能通过消息互相影响（避免任何锁定）。

actor 在高负载下比锁更有效，因为在这种情况下它总是有工作要做，而且根本不需要切换到不同的上下文。

注意，`actor` 协程构建器是一个双重的 `produce` 协程构建器。一个 actor 与它接收消息的通道相关联，而一个 producer 与它发送元素的通道相关联。



## 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` 和 `buzz`。其中 `fizz` 每 300 毫秒生成一个“Fizz”字符串：

```
fun CoroutineScope.fizz() = produce<String> {
 while (true) { // 每 300 毫秒发送一个 "Fizz"
 delay(300)
 send("Fizz")
 }
}
```

接着 `buzz` 每 500 毫秒生成一个“Buzz!”字符串：

```
fun CoroutineScope.buzz() = produce<String> {
 while (true) { // 每 500 毫秒发送一个"Buzz!"
 delay(500)
 send("Buzz!")
 }
}
```

使用 `receive` 挂起函数，我们可以从两个通道接收 其中的一个 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：

### 1.5.30 的新特性

```
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：

### 1.5.30 的新特性

```
import kotlinx.coroutines.*
import kotlinx.coroutines.channels.*
import kotlinx.coroutines.selects.*

fun CoroutineScope.fizz() = produce<String> {
 while (true) { // 每 300 毫秒发送一个 "Fizz"
 delay(300)
 send("Fizz")
 }
}

fun CoroutineScope.buzz() = produce<String> {
 while (true) { // 每 500 毫秒发送一个 "Buzz!"
 delay(500)
 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：

### 1.5.30 的新特性

```
fizz -> 'Fizz'
buzz -> 'Buzz!'
fizz -> 'Fizz'
fizz -> 'Fizz'
buzz -> 'Buzz!'
fizz -> 'Fizz'
buzz -> 'Buzz!'
```

## 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：

## 1.5.30 的新特性

```
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，所以我们在更多细节中分析它：

### 1.5.30 的新特性

```
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` 倾向于第一个子句，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：

## 1.5.30 的新特性

```
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。

## 1.5.30 的新特性

```
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：

## 1.5.30 的新特性

```
fun CoroutineScope.switchMapDeferreds(input: ReceiveChannel<Deferred<String>>) = pr
 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
 }
 }
 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：

## 1.5.30 的新特性

```
import kotlinx.coroutines.*
import kotlinx.coroutines.channels.*
import kotlinx.coroutines.selects.*

fun CoroutineScope.switchMapDeferreds(input: ReceiveChannel<Deferred<String>>) = pr
 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
}
```

### 1.5.30 的新特性

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.

Debugging works for `kotlinx-coroutines-core` version 1.3.8 or later.



## Create coroutines

1. Open a Kotlin project in IntelliJ IDEA. If you don't have a project, [create one](#).
2. 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!`.

3. Change code in the `main()` function:

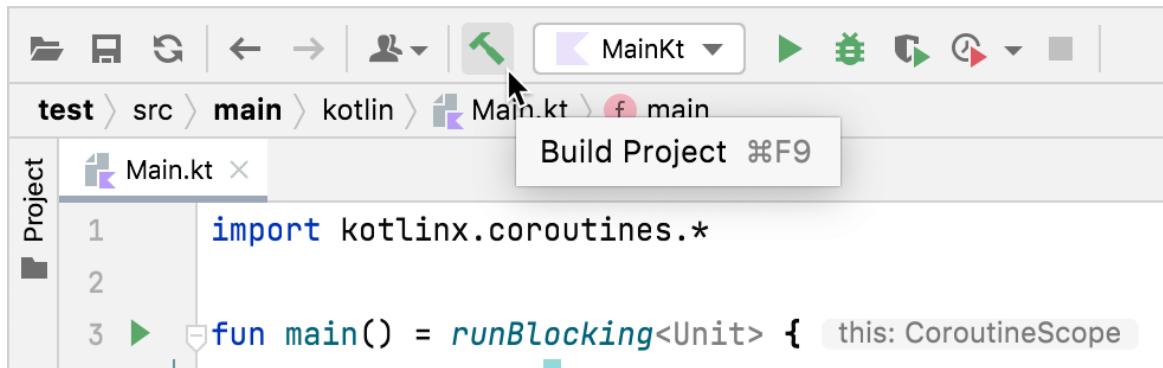
- Use the `runBlocking()` block to wrap a coroutine.
- Use the `async()` function to create coroutines that compute deferred values `a` and `b`.
- Use the `await()` function to await the computation result.
- Use the `println()` function to print computing status and the result of multiplication to the 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()}")
}
```

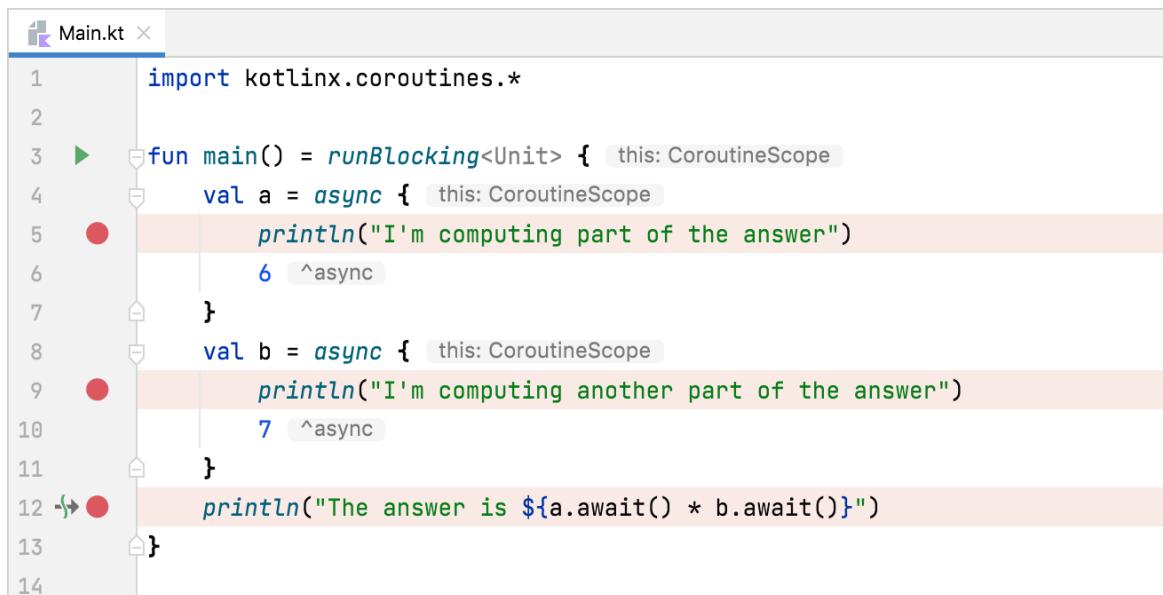
## 1.5.30 的新特性

### 4. Build the code by clicking **Build Project**.

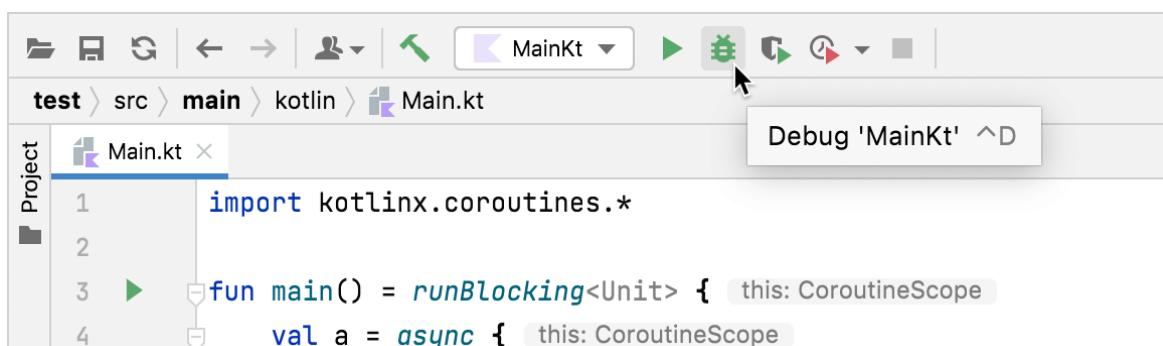


## Debug coroutines

### 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.

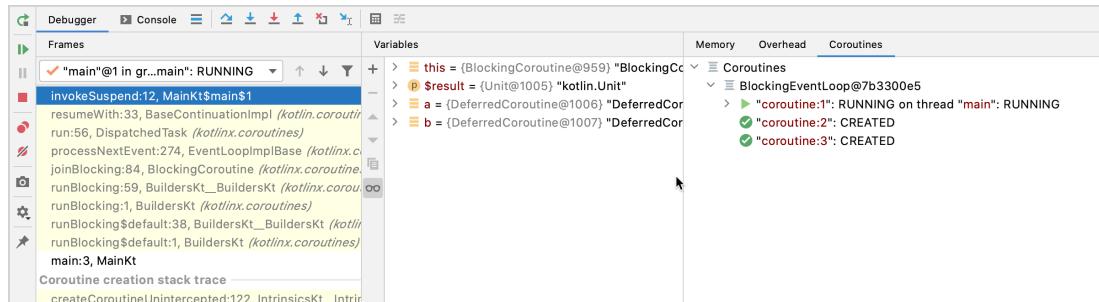


The **Debug** tool window appears:

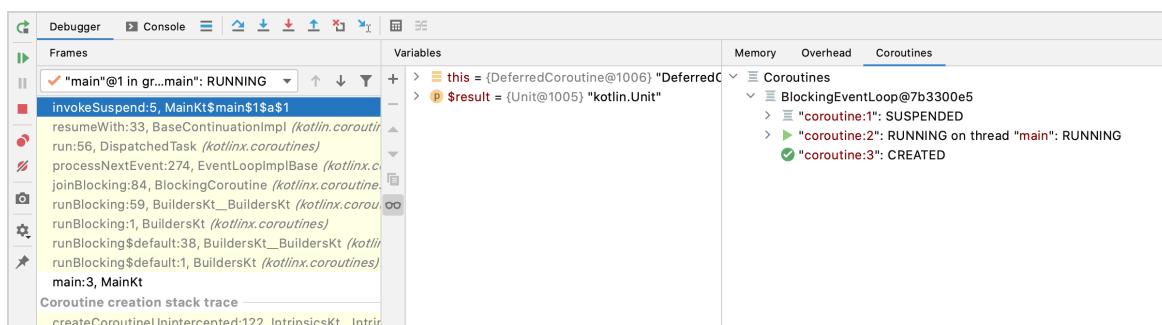
- The **Frames** tab contains the call stack.

### 1.5.30 的新特性

- 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.



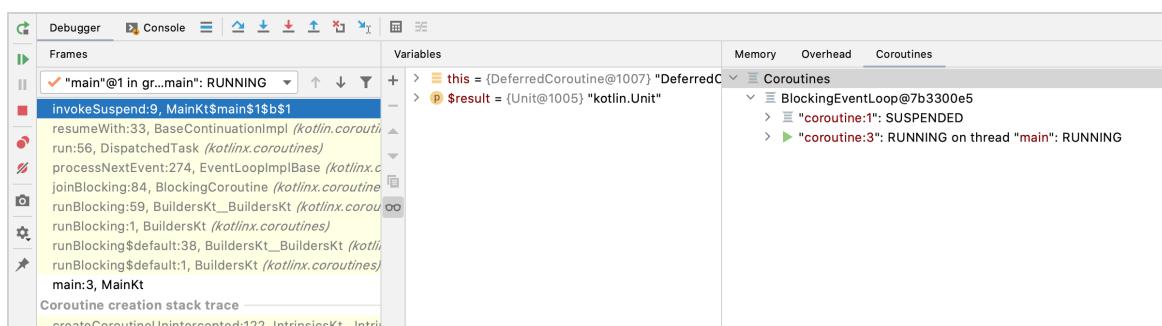
3. Resume the debugger session by clicking **Resume program** in the **Debug** tool window:



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:

### 1.5.30 的新特性

- 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.

# 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.

Debugging works for `kotlinx-coroutines-core` version 1.3.8 or later.



## 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. 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!`.

3. 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)
 }
}
```

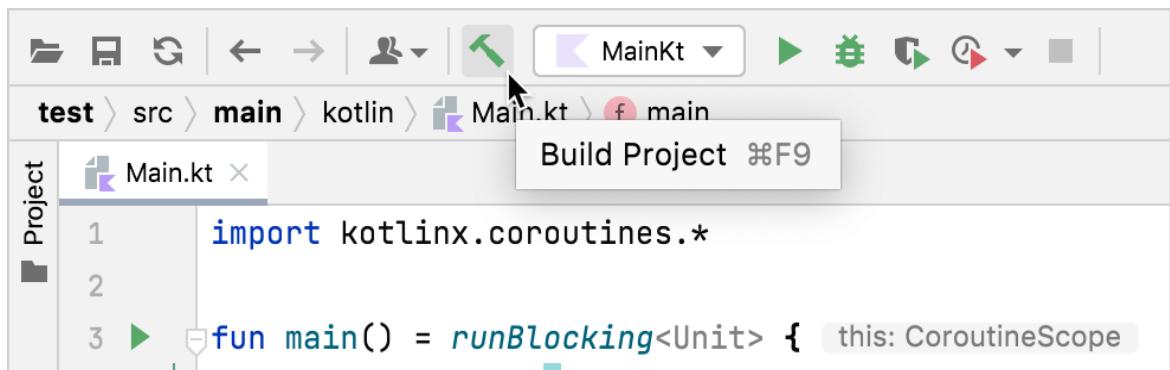
4. Change the code in the `main()` function:

### 1.5.30 的新特性

- 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)
 }
}
```

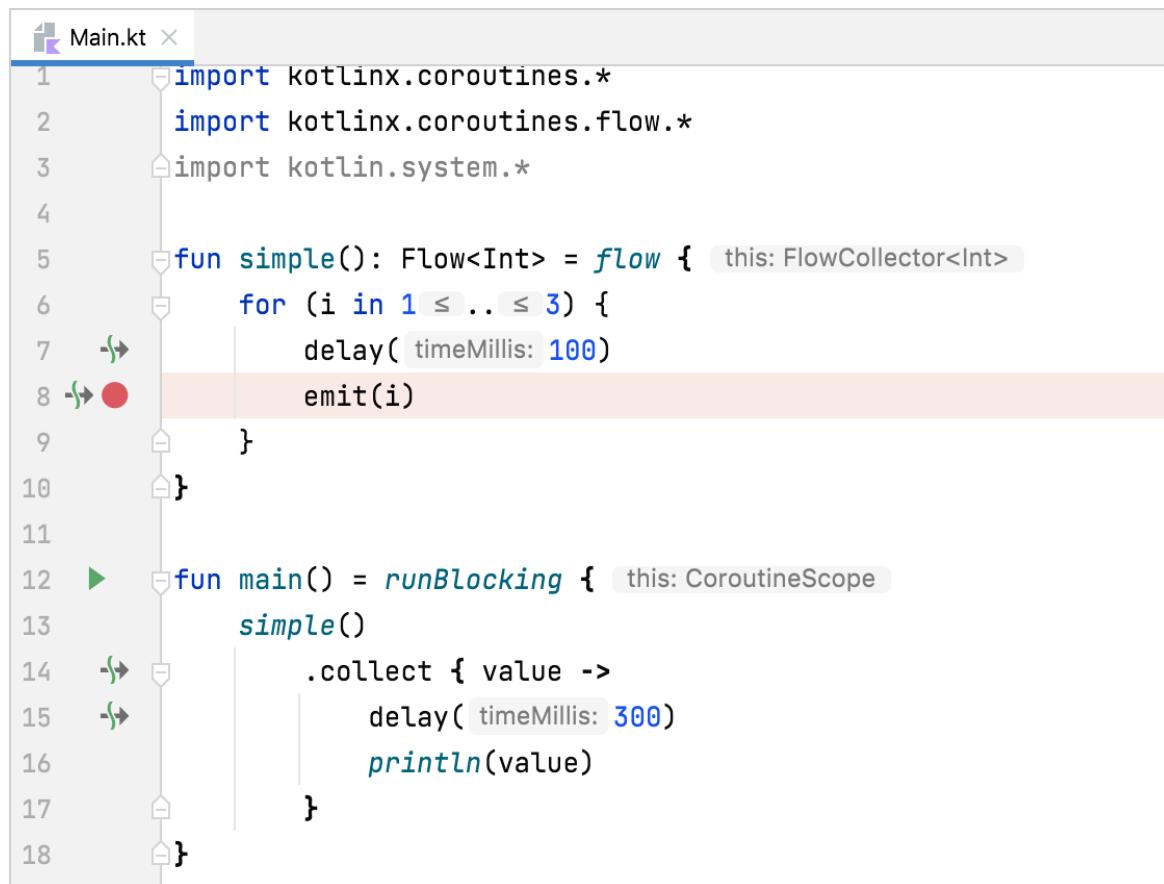
### 5. Build the code by clicking **Build Project**.



## Debug the coroutine

### 1. Set a breakpoint at the at the line where the `emit()` function is called:

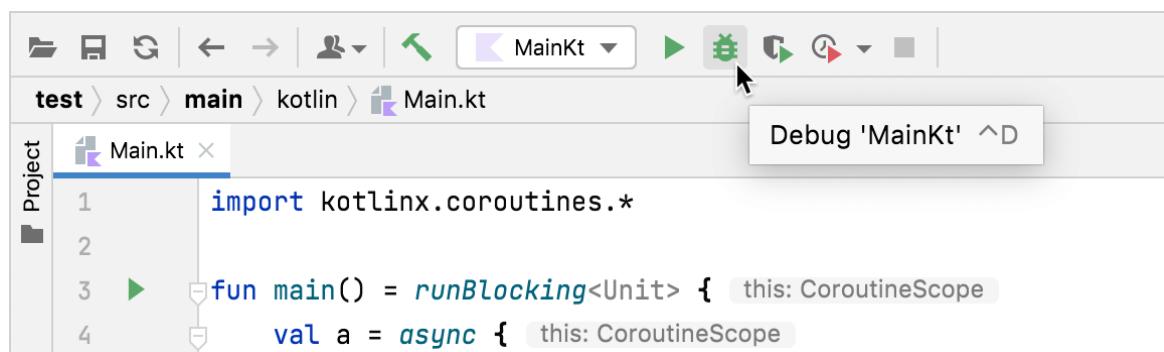
### 1.5.30 的新特性



Main.kt

```
1 import kotlinx.coroutines.*
2 import kotlinx.coroutines.flow.*
3 import kotlin.system.*
4
5 fun simple(): Flow<Int> = flow { this: FlowCollector<Int>
6 for (i in 1 .. 3) {
7 delay(timeMillis: 100)
8 emit(i)
9 }
10 }
11
12 fun main() = runBlocking { this: CoroutineScope
13 simple()
14 .collect { value ->
15 delay(timeMillis: 300)
16 println(value)
17 }
18 }
```

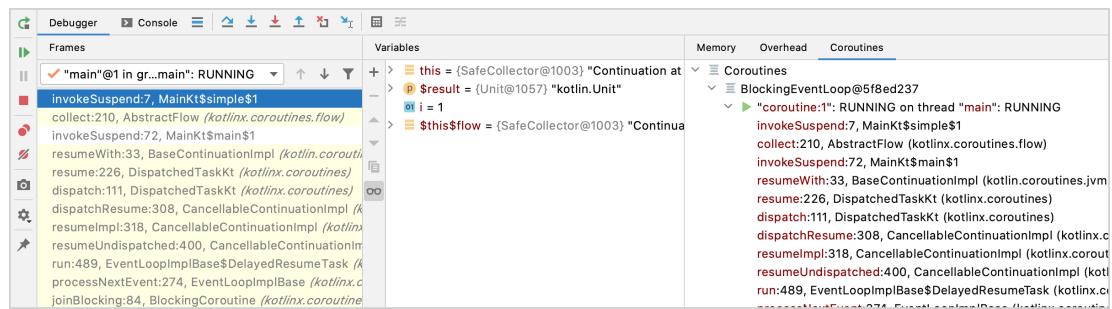
2. Run the code in debug mode by clicking **Debug** next to the run configuration at the top of the screen.



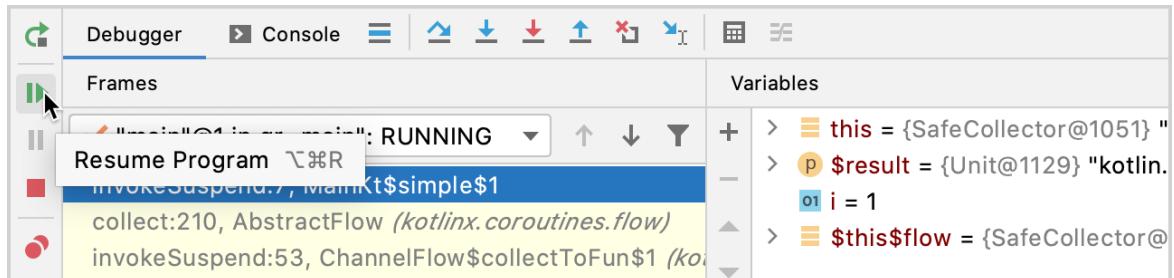
The **Debug** tool window appears:

- 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.

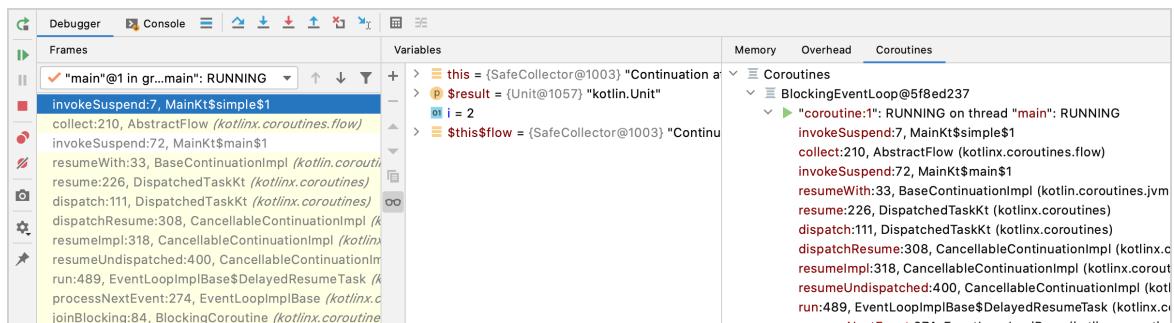
## 1.5.30 的新特性



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.



## 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.

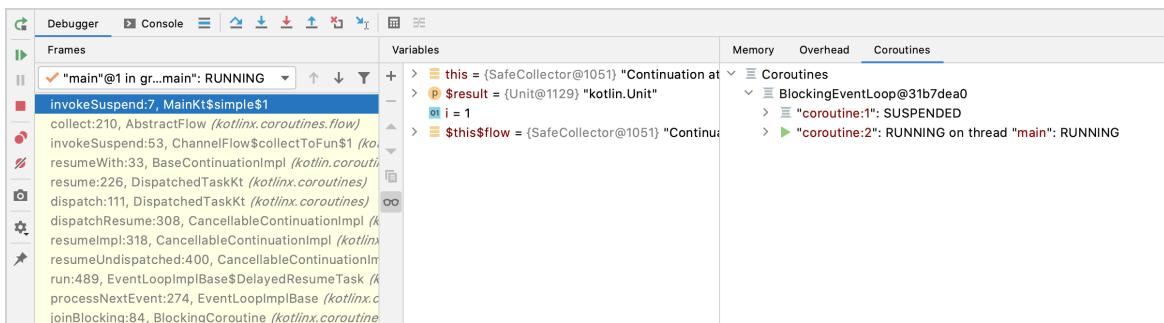
### 1.5.30 的新特性

```
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.

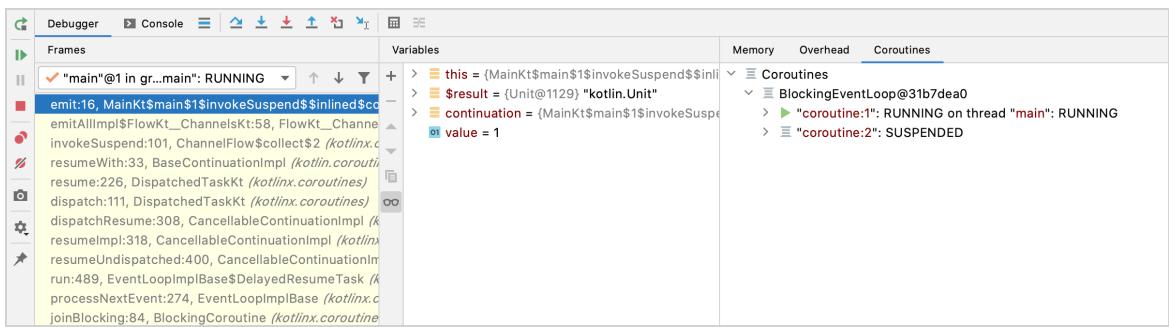


The **Debug** tool window appears.

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.

### 1.5.30 的新特性



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 an essential part of 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 two main parts: the Gradle plugin – `org.jetbrains.kotlin.plugin.serialization` and the runtime libraries.

## 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-core`) 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.

Before starting, you'll need to 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 "1.6.10"
 kotlin("plugin.serialization") version "1.6.10"
}
```

### 【Groovy】

```
plugins {
 id 'org.jetbrains.kotlin.jvm' version '1.6.10'
 id 'org.jetbrains.kotlin.plugin.serialization' version '1.6.10'
}
```

## 1.5.30 的新特性

1. Add the JSON serialization library dependency: `org.jetbrains.kotlinx:kotlinx-serialization-json:1.3.2`

### 【Kotlin】

```
dependencies {
 implementation("org.jetbrains.kotlinx:kotlinx-serialization-json:1.3.2")
}
```

### 【Groovy】

```
dependencies {
 implementation 'org.jetbrains.kotlinx:kotlinx-serialization-json:1.3.2'
}
```

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`.

First, make a class serializable by annotating it with `@Serializable`.

```
import kotlinx.serialization.Serializable

@Serializable
data class Data(val a: Int, val b: String)
```

You can now 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.

## 1.5.30 的新特性

```
val dataList = listOf(Data(42, "str"), Data(12, "test"))
val jsonList = Json.encodeToString(dataList)
```

To deserialize an object from JSON, use the `decodeFromString()` function:

```
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"}""")
}
```

For more information about serialization in Kotlin, see the [Kotlin Serialization Guide](#).

1.5.30 的新特性

## Ktor

 [Ktor](#)

## API 参考

- 标准库 (stdlib) ↗
- 测试库 (kotlin.test) ↗
- 协程 (kotlinx.coroutines) ↗
- Ktor ↗

## 标准库（stdlib）

 [标准库（stdlib）](#)

## 测试库（kotlin.test）

 [测试库（kotlin.test）](#)

## 协程 (kotlinx.coroutines)

 [协程 \(kotlinx.coroutines\)](#)

1.5.30 的新特性

## Ktor

 [Ktor](#)

## 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。

### 1.5.30 的新特性

- `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。

### 1.5.30 的新特性

- `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相关，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。

### 1.5.30 的新特性

- `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。
- `..` 创建一个区间。

### 1.5.30 的新特性

- `:` 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。

# 语法

 [语法](#)

# 语言规范

 [语言规范](#)

# 工具

- 构建工具
  - [Gradle](#)
  - [Maven](#)
  - [Ant](#)
- IntelliJ IDEA
  - [迁移到 Kotlin 代码风格](#)
- Android Studio ↗
- Eclipse IDE
- 运行代码片段
- 编译器
  - [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](#)
  - [使用 kapt](#)
  - [Lombok 编译器插件](#)
- 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 与 TeamCity 的持续集成
- KDoc 与 Dokka
- Kotlin 与 OSGi

## 构建工具

- Gradle
- Maven
- Ant

## Gradle

In order to build a Kotlin project with Gradle, you should [apply the Kotlin Gradle plugin to your project](#) and [configure the dependencies](#).

## 插件与版本

使用 [Gradle 插件 DSL](#) 应用 Kotlin Gradle 插件。

The Kotlin Gradle plugin and the `kotlin-multiplatform` plugin 1.6.10 require Gradle 6.1 or later.

### 【Kotlin】

```
plugins {
 kotlin("<...>") version "1.6.10"
}
```

### 【Groovy】

```
plugins {
 id 'org.jetbrains.kotlin.<...>' version '1.6.10'
}
```

需要将其中的占位符 `<...>` 替换为在后续部分讨论的插件名之一。

## Targeting multiple platforms

Projects targeting [multiple platforms](#), called [multiplatform projects](#), require the `kotlin-multiplatform` plugin. [Learn more about the plugin](#).

The `kotlin-multiplatform` plugin works with Gradle 6.1 or later.



### 【Kotlin】

### 1.5.30 的新特性

```
plugins {
 kotlin("multiplatform") version "1.6.10"
}
```

#### 【Groovy】

```
plugins {
 id 'org.jetbrains.kotlin.multiplatform' version '1.6.10'
}
```

## 面向 JVM

如需面向 JVM 平台，请应用 Kotlin JVM 插件。

#### 【Kotlin】

```
plugins {
 kotlin("jvm") version "1.6.10"
}
```

#### 【Groovy】

```
plugins {
 id "org.jetbrains.kotlin.jvm" version "1.6.10"
}
```

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。

或者使用旧版 `apply plugin` 方式：

```
apply plugin: 'kotlin'
```

不建议在 Kotlin Gradle DSL 中以 `apply` 的方式应用 Kotlin 插件——[参见为什么](#)。

## Kotlin and Java sources

Kotlin 源代码可以与 Java 源代码放在相同文件夹或者不同文件夹中。默认约定是使用不同的文件夹：

## 1.5.30 的新特性

```
project
 - src
 - main (root)
 - kotlin
 - 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 such related tasks, the Kotlin Gradle plugin checks for JVM target compatibility. Different values of `jvmTarget` in the `kotlin` extension and `targetCompatibility` in the `java` extension cause incompatibility. For example: the `compileKotlin` task has `jvmTarget=1.8`, and the `compileJava` task has (or `inherits`) `targetCompatibility=15`.

Control the behavior of this check by setting the `kotlin.jvm.target.validation.mode` property in the `build.gradle` file equal to:

- `warning` – the default value; the Kotlin Gradle plugin will print a warning message.

### 1.5.30 的新特性

- `error` – the plugin will fail the build.
- `ignore` – the plugin will skip the check and won't produce any messages.

## Set custom JDK home

By default, Kotlin compile tasks use the current Gradle JDK. If you need to change the JDK by some reason, you can set the JDK home in the following ways:

- For Gradle 6.7 and later – with [Java toolchains](#) or the [Task DSL](#) to set a local JDK.
- For earlier Gradle versions without Java toolchains (up to 6.6) – with the [UsesKotlinJavaToolchain interface and the Task DSL](#).

The `jdkHome` compiler option is deprecated since Kotlin 1.5.30.



When you use a custom JDK, note that [kapt task workers](#) use [process isolation mode](#) only, and ignore the `kapt.workers.isolation` property.

## Gradle Java toolchains support

Gradle 6.7 introduced [Java toolchains support](#). Using this feature, you can:

- Use a JDK and a JRE that are different from the Gradle ones 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 uses the JDK the Gradle daemon is running on. A Java toolchain:

- Sets the `jdkHome` option available for JVM targets.
- Sets the `kotlinOptions.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 will use the default value. Learn more about [JVM target compatibility](#).
- Affects which JDK [kapt workers](#) are running on.

## 1.5.30 的新特性

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(<MAJ
 }
}
```

### 【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.

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 the specific task, use the Task DSL.

## Setting JDK version with the Task DSL

If you use a Gradle version earlier than 6.7, there is no [Java toolchains support](#). You can use the Task DSL that 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】

## 1.5.30 的新特性

```
val service = project.extensions.getByName<JavaToolchainService>()
val customLauncher = service.launcherFor {
 it.languageVersion.set(JavaLanguageVersion.of(<MAJOR_JDK_VERSION>)) // "8"
}
project.tasks.withType<UsesKotlinJavaToolchain>().configureEach {
 kotlinJavaToolchain.toolchain.use(customLauncher)
}
```

### 【Groovy】

```
JavaToolchainService service = project.getExtensions().getByName<JavaToolchainService>()
Provider<JavaLauncher> customLauncher = service.launcherFor {
 it.languageVersion.set(JavaLanguageVersion.of(<MAJOR_JDK_VERSION>)) // "8"
}
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
)
}
```

# Targeting JavaScript

When targeting only JavaScript, use the `kotlin-js` plugin. [Learn more](#)

### 【Kotlin】

```
plugins {
 kotlin("js") version "1.6.10"
}
```

### 【Groovy】

### 1.5.30 的新特性

```
plugins {
 id 'org.jetbrains.kotlin.js' version '1.6.10'
}
```

## 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'
 }
}
```

## 面向 Android

It's recommended to use Android Studio for creating Android applications. [Learn how to use Android Gradle plugin.](#)

## Configuring 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】

## 1.5.30 的新特性

```
kotlin {
 sourceSets {
 val commonMain by getting {
 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 the top level](#).

## Dependency types

Choose the dependency type based on your requirements.

### 1.5.30 的新特性

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 will select the appropriate JVM standard library depending on the `kotlinOptions.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 do not need a standard library at all, you can add the opt-out option to the `gradle.properties`:

```
kotlin.stdlib.default.dependency=false
```

## Set dependencies on test libraries

The `kotlin.test` API is available for testing Kotlin projects on all supported platforms.

Add the dependency `kotlin-test` to the `commonTest` source set, and the Gradle plugin will infer the corresponding test dependencies for each test source set:

- `kotlin-test-common` and `kotlin-test-annotations-common` for common source sets
- `kotlin-test-junit` for JVM source sets
- `kotlin-test-js` for Kotlin/JS source sets

Kotlin/Native targets do not require additional test dependencies, and the `kotlin.test` API implementations are built-in.

### 【Kotlin】

```
kotlin {
 sourceSets {
 val commonTest by getting {
 dependencies {
 implementation(kotlin("test")) // This brings all the platform depen
 }
 }
 }
}
```

### 【Groovy】

```
kotlin {
 sourceSets {
 commonTest {
 dependencies {
 implementation kotlin("test") // This brings all the platform depen
 }
 }
 }
}
```

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.

## 1.5.30 的新特性

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 {
 val commonTest by getting {
 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】

## 1.5.30 的新特性

```
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.](#)

If you need to use a different JVM test framework, disable automatic testing framework selection by adding the line `kotlin.test.infer.jvm.variant=false` to the project's `gradle.properties` file. After doing this, add the framework as a Gradle dependency.

If you had used a variant of `kotlin("test")` in your build script explicitly and 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 kotlinx library and need a platform-specific dependency, you can use platform-specific variants of libraries with suffixes such as `-jvm` or `-js`, for example, `kotlinx-coroutines-core-jvm`. You can also use the library's base artifact name instead – `kotlinx-coroutines-core`.

### 【Kotlin】

## 1.5.30 的新特性

```
kotlin {
 sourceSets {
 val jvmMain by getting {
 dependencies {
 implementation("org.jetbrains.kotlinx:kotlinx-coroutines-core-jvm:1.6.0")
 }
 }
 }
}
```

### 【Groovy】

```
kotlin {
 sourceSets {
 jvmMain {
 dependencies {
 implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-core-jvm:1.6.0'
 }
 }
 }
}
```

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 {
 val commonMain by getting {
 dependencies {
 implementation("org.jetbrains.kotlinx:kotlinx-coroutines-core:1.6.0")
 }
 }
 }
}
```

### 【Groovy】

### 1.5.30 的新特性

```
kotlin {
 sourceSets {
 commonMain {
 dependencies {
 implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-core:1.6.0'
 }
 }
 }
}
```

## Set dependencies at the top level

Alternatively, you can specify the dependencies at the 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'
}
```

## 注解处理

Kotlin 通过 Kotlin 注解处理工具 `kapt` 支持注解处理。

## 增量编译

Kotlin Gradle 插件支持支持增量编译。增量编译会跟踪多次构建之间源文件的变更，因此只会编译这些变更所影响的文件。

Kotlin/JVM 与 Kotlin/JS 项目均支持增量编译 and is enabled by default since Kotlin 1.1.1。

### 1.5.30 的新特性

有几种方法可以禁增量编译：

- `kotlin.incremental=false` 用于 Kotlin/JVM。
- `kotlin.incremental.js=false` 用于 Kotlin/JS projects。
- 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，并且任何具有禁用增量编译的构建都会使增量缓存失效。

首次构建决不会是增量的。

## 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，请将系统属性标志 `kotlin.caching.enabled` 设置为 `false`（运行构建带上参数 `-Dkotlin.caching.enabled=false`）。

如果使用 `kapt`，请注意默认情况下不会缓存注解处理任务。不过，可以[手动为它们启用缓存](#)。

## Gradle configuration cache support

The configuration cache is available in Gradle 6.5 and later as an experimental feature. You can check the [Gradle releases page](#) to see whether it has been promoted to stable.



The Kotlin plugin uses the [Gradle configuration cache](#), which speeds up the build process by reusing the results of the configuration phase.

See the [Gradle documentation](#) to learn how to enable the configuration cache. After you enable this feature, the Kotlin Gradle plugin will automatically start using it.

## 编译器选项

使用 Kotlin 编译任务的 `kotlinOptions` 属性来指定附加的编译选项。

### 1.5.30 的新特性

当面向 JVM 时，对于生产代码这些任务称为 `compileKotlin` 而对于测试代码称为 `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。

Android 项目中的任务名称包含构建变体 名称，并遵循 `compile<BuildVariant>Kotlin` 的模式，例如 `compileDebugKotlin` 或 `compileReleaseUnitTestKotlin`。

当面向 JavaScript 时，任务 `compileKotlinJs` 用于生产代码，而 `compileTestKotlinJs` 用于测试代码，以及对于自定义源代码集称为 `compile<Name>KotlinJs`。

要配置单个任务，请使用其名称。示例：

#### 【Kotlin】

```
import org.jetbrains.kotlin.gradle.tasks.KotlinCompile
// ...

val compileKotlin: KotlinCompile by tasks

compileKotlin.kotlinOptions.suppressWarnings = true
```

#### 【Groovy】

```
compileKotlin {
 kotlinOptions.suppressWarnings = true
}

//或者

compileKotlin {
 kotlinOptions {
 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`。

也可以在项目中配置所有 Kotlin 编译任务：

#### 【Kotlin】

### 1.5.30 的新特性

```
tasks.withType<org.jetbrains.kotlin.gradle.tasks.KotlinCompile>().configureEach {
 kotlinOptions { /*.....*/ }
}
```

#### 【Groovy】

```
tasks.withType(org.jetbrains.kotlin.gradle.tasks.KotlinCompile).configureEach {
 kotlinOptions { /*.....*/ }
}
```

Gradle 任务的完整选项列表如下：

## JVM、JS 与 JS DCE 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 <a href="#">已启用 Gradle debug 日志</a> 时才有效		false
freeCompilerArgs	附加编译器参数的列表		[]

## JVM 与 JS 的公共属性

名称	描述	可能的值	默认值
apiVersion	限制只使用来自内置库的指定版本中的声明	"1.3" (已弃用)、"1.4" (已弃用)、"1.5"、"1.6"、"1.7" (实验性)	
languageVersion	提供与指定 Kotlin 版本源代码级兼容	"1.4" (已弃用)、"1.5"、"1.6"、"1.7" (实验性)	

## JVM 特有的属性

### 1.5.30 的新特性

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
jdkHome	将来自指定位置的自定义 JDK 而不是默认的 JAVA_HOME 包含到类路径中。 Direct setting is deprecated since 1.5.30, use other ways to set this option.		
jvmTarget	生成的 JVM 字节码的目标版本	"1.6" (已弃用)、 "1.8"、 "9"、 "10"、 "11"、 "12"、 "13"、 "14"、 "15"、 "16"、 "17"	"1.8"
noJdk	不要自动在类路径中包含 Java 运行时		false
useOldBackend	Use the old JVM backend		false

## JS 特有的属性

### 1.5.30 的新特性

名称	描述	可能的值	默认值
<code>friendModulesDisabled</code>	禁用内部声明导出		false
<code>main</code>	定义是否在执行时调用 <code>main</code> 函数	"call"、 "noCall"	"call"
<code>metaInfo</code>	使用元数据生成 <code>.meta.js</code> 与 <code>.kjsm</code> 文件。用于创建库		true
<code>moduleKind</code>	编译器生成的 JS 模块类型	"umd"、 "commonjs"、 "amd"、 "plain"	"umd"
<code>noStdlib</code>	不要自动将默认的 Kotlin/JS <code>stdlib</code> 包含到编译依赖项中		true
<code>outputFile</code>	编译结果的目标 <code>*.js</code> 文件		"\js/packages\kotlin\js"
<code>sourceMap</code>	生成源代码映射 (source map)		true
<code>sourceMapEmbedSources</code>	将源代码嵌入到源代码映射中	"never"、 "always"、 "inlining"	

### 1.5.30 的新特性

名称	描述	可能的值	默认值
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	"v5"	"v5"
typedArrays	将原生数组转换为 JS 带类型数组		true

## 生成文档

要生成 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](#)。

## 使用 Gradle Kotlin DSL

使用 [Gradle Kotlin DSL](#) 时, 请使用 `plugins { .... }` 块应用 Kotlin 插件。如果使用 `apply { plugin(....) }` 来应用的话, 可能会遇到未解析的到由 Gradle Kotlin DSL 所生成扩展的引用问题。为了解决这个问题, 可以注释掉出错的用法, 运行 Gradle 任务 `kotlinDslAccessorsSnapshot`, 然后解除该用法注释并重新运行构建或者重新将项目导入到 IDE 中。

## Kotlin daemon and using it with Gradle

The Kotlin daemon:

### 1.5.30 的新特性

- Runs along with the Gradle daemon to compile the project.
- Runs separately 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 Kotlin compile tasks starts compiling the sources. The Kotlin daemon stops along 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 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. 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 in the `gradle.properties` file:

```
org.gradle.jvmargs=-Dkotlin.daemon.jvm.options=-Xmx1500m,Xms=500m
```

When passing the arguments, follow these rules:

- Use the minus sign `-` before the arguments `Xmx`, `XX:MaxMetaspaceSize`, and `XX:ReservedCodeCacheSize` and don't use it before all other arguments.
- 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.
- There is no running Kotlin daemon.

To overcome this, upgrade Gradle to the version 7.2 (or higher) or use the `kotlin.daemon.jvmargs` property – see the following item.



## 1.5.30 的新特性

- 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.withType<CompileUsingKotlinDaemon>().configureEach {
 kotlinDaemonJvmArguments.set(listOf("-Xmx486m", "-Xms256m", "-XX:+UseParallelGC"))
}
```

### 【Groovy】

```
tasks.withType(CompileUsingKotlinDaemon::class).configureEach { task ->
 task.kotlinDaemonJvmArguments.set(["-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:

### 1.5.30 的新特性

- 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 `Xmx` is not specified, the Kotlin daemon will inherit it from the Gradle daemon.

# Maven

## 插件与版本

`kotlin-maven-plugin` 用于编译 Kotlin 源代码与模块，目前只支持 Maven V3。

通过 `kotlin.version` 属性定义要使用的 Kotlin 版本：

```
<properties>
 <kotlin.version>{{ site.data.releases.latest.version }}</kotlin.version>
</properties>
```

## 依赖

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 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>
```

如果是面向 JDK 7 或 JDK 8，那么可以使用扩展版本的 Kotlin 标准库。其中包含为新版 JDK 所增 API 而加的额外的扩展函数。使用 `kotlin-stdlib-jdk7` 或 `kotlin-stdlib-jdk8` 取代 `kotlin-stdlib`，这取决于你的 JDK 版本（对于 Kotlin 1.1.x 用 `kotlin-stdlib-jre7` 与 `kotlin-stdlib-jre8`，因为相应的 `jdk` 构件在 1.2.0 才引入）。

For Kotlin versions older than 1.2, use `kotlin-stdlib-jre7` and `kotlin-stdlib-jre8`.



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>
```

## 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:

## 1.5.30 的新特性

```
<build>
 <plugins>
 <plugin>
 <groupId>org.jetbrains.kotlin</groupId>
 <artifactId>kotlin-maven-plugin</artifactId>
 <version>${kotlin.version}</version>
 <executions>
 <execution>
 <id>compile</id>
 <goals>
 <goal>compile</goal>
 </goals>
 <configuration>
 <sourceDirs>
 <sourceDir>${project.basedir}/src/main/kotlin</sourceDir>
 <sourceDir>${project.basedir}/src/main/java</sourceDir>
 </sourceDirs>
 </configuration>
 </execution>
 <execution>
 <id>test-compile</id>
 <goals> <goal>test-compile</goal> </goals>
 <configuration>
 <sourceDirs>
 <sourceDir>${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>
```

### 1.5.30 的新特性

```
<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>
```

## 增量编译

为了使构建更快，可以为 Maven 启用增量编译（从 Kotlin 1.1.2 起支持）。为了做到这一点，需要定义 `kotlin.compiler.incremental` 属性：

```
<properties>
 <kotlin.compiler.incremental>true</kotlin.compiler.incremental>
</properties>
```

或者，使用 `-Dkotlin.compiler.incremental=true` 选项运行构建。

## 注解处理

请参见 [Kotlin 注解处理工具（`kapt`）](#) 的描述。

## Jar 文件

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：

### 1.5.30 的新特性

```
<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>
```

## 独立的 Jar 文件

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：

## 1.5.30 的新特性

```
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>
 <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>1.0</kotlin.compiler.languageVersion>
 </properties>
</project>
```

支持以下属性：

## JVM 与 JS 的公共属性

### 1.5.30 的新特性

名称	属性名	描述	可能的值
nowarn		不生成警告	true、false
languageVersion	kotlin.compiler.languageVersion	提供与指定语言版本源代码兼容性	"1.4"（已弃用）,"1.5"、"1.6"、"1.7"（实验性）
apiVersion	kotlin.compiler.apiVersion	只允许使用来自捆绑库的指定版本中的声明	"1.3"（已弃用）,"1.4"（已弃用）、"1.5"、"1.6"、"1.7"（实验性）
sourceDirs		包含要编译源文件的目录	
compilerPlugins		启用编译器插件	
pluginOptions		编译器插件的选项	

### 1.5.30 的新特性

名称	属性名	描述	可能的值
args		额外的编译器参数	

### JVM 特有的属性

名称	属性名	描述	可能的值	默认值
jvmTarget	kotlin.compiler.jvmTarget	生成的 JVM 字节码的目标版本	"1.6" (已弃用)、 "1.8"、 "9"、 "10"、 "11"、 "12"、 "13"、 "14"、 "15"、 "16"、 "17"	"1.8"
jdkHome	kotlin.compiler.jdkHome	Include a custom JDK from the specified location into the classpath instead of the default JAVA_HOME		

### JS 特有的属性

### 1.5.30 的新特性

名称	属性名	描述	可能的值	默认值
<code>outputFile</code>		Destination *.js file for the compilation result		
<code>metaInfo</code>		使用元数据生成 .meta.js 与 .kjsm 文件。用于创建库	true、false	true
<code>sourceMap</code>		生成源代码映射 (source map)	true、false	false
<code>sourceMapEmbedSources</code>		将源代码嵌入到源代码映射中	"never"、"always"、"inlining"	"inlining"
<code>sourceMapPrefix</code>		Add the specified prefix to paths in the source map		
<code>moduleKind</code>		The kind of JS module generated by the compiler	"umd", "commonjs", "amd", "plain"	"umd"

## 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](#)；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

对于 OSGi 支持，请参见 [Kotlin OSGi 页](#)。

# 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}\

 <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：

### 1.5.30 的新特性

```
<project name="Ant Task Test" default="build">
 <totype resource="org/jetbrains/kotlin/ant/antlib.xml" classpath="${kotlin.lib}>

 <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">
 <totype resource="org/jetbrains/kotlin/ant/antlib.xml" classpath="${kotlin.lib}>

 <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

### 1.5.30 的新特性

```
<project name="Ant Task Test" default="build">
 <typedef resource="org/jetbrains/kotlin/ant/antlib.xml" classpath="${kotlin.lib}>
 <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}>
 <target name="build">
 <kotlin2js src="root1" output="out.js" outputPrefix="prefix" outputPostfix="postfix" 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}>
 <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：

### 1.5.30 的新特性

## 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
failOngetError	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。

## IntelliJ IDEA

- 迁移到 Kotlin 代码风格

## 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。

### 1.5.30 的新特性

The diagram illustrates the evolution of a Java code snippet from an older style to a newer, more modern style. On the left, the original code uses traditional Java syntax with explicit null checks and separate filter operations. On the right, the updated code uses the new Kotlin-style null safety features and the `filter` extension function, which allows for more concise and readable code.

```
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。

### 1.5.30 的新特性

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

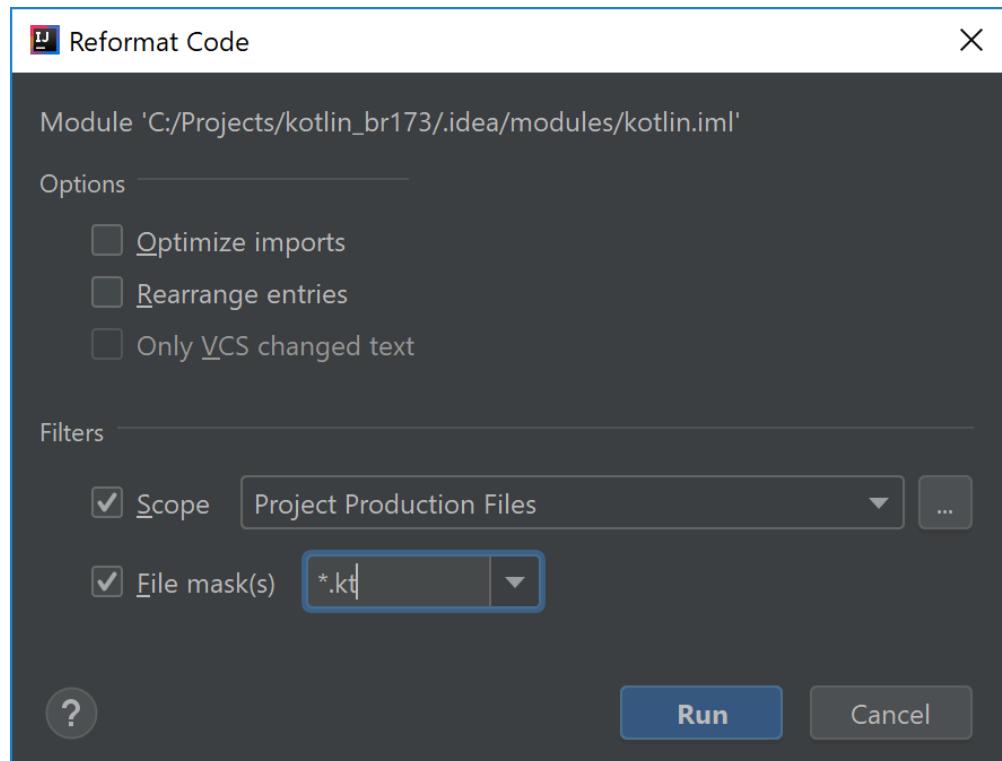
在项目根目录的 `pom.xml` 文件中添加 `kotlin.code.style official` 属性。

```
<properties>
 <kotlin.code.style>official</kotlin.code.style>
</properties>
```

设置 `kotlin.code.style` 选项可能会影响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**。



### 1.5.30 的新特性

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。

1.5.30 的新特性

# Android Studio

 [Android Studio](#)

# Eclipse IDE

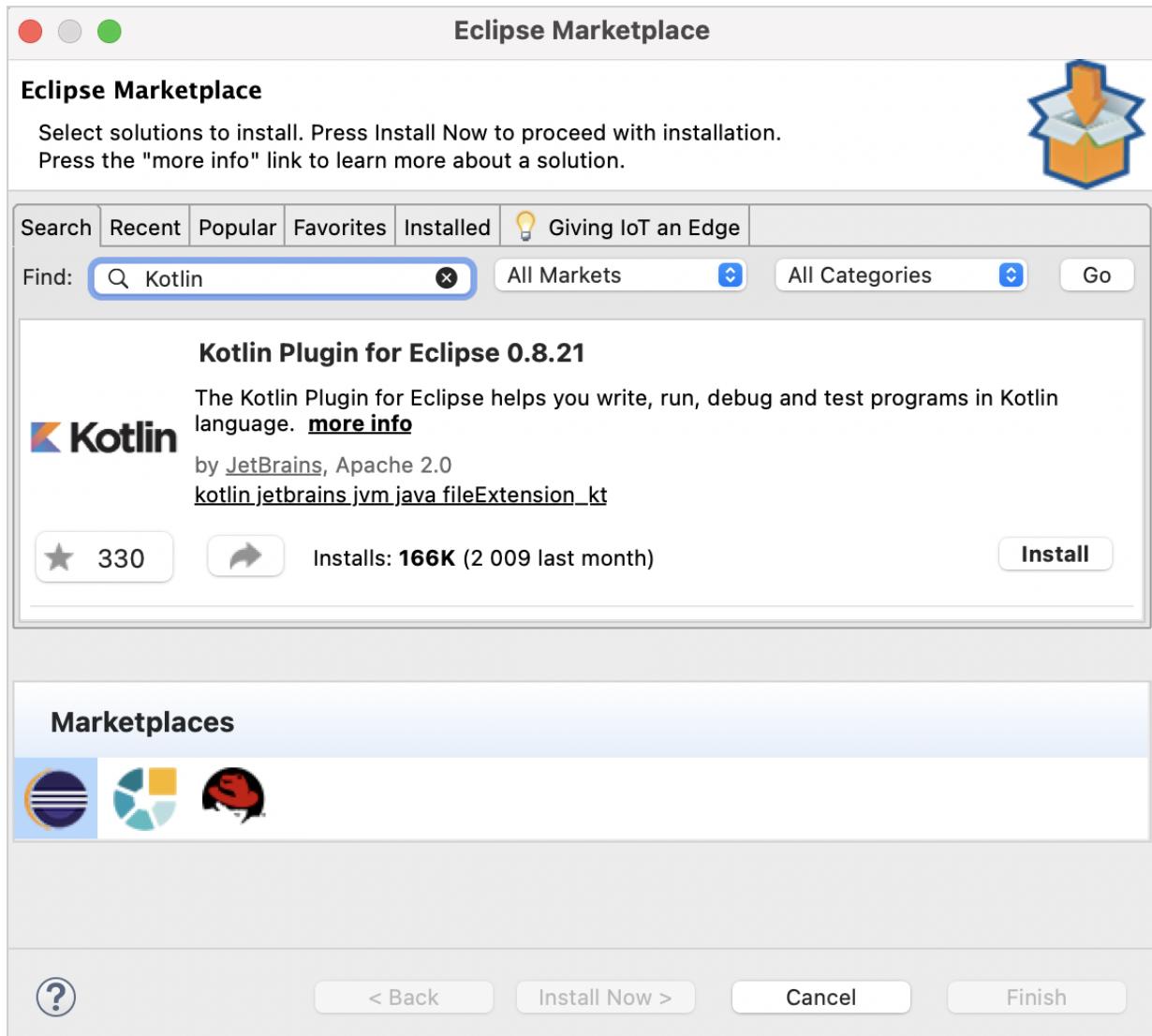
[Eclipse IDE](#) is a widely known IDE that offers various packages for development in different languages and for different platforms. You can use it for writing Kotlin code. On this page, you will learn how to get started with Kotlin in Eclipse IDE.

## 搭建环境

首先，需要在系统上安装 Eclipse IDE。可以从[下载页面](#)下载最新版本。建议使用 **Eclipse IDE for Java Developers** 软件包。

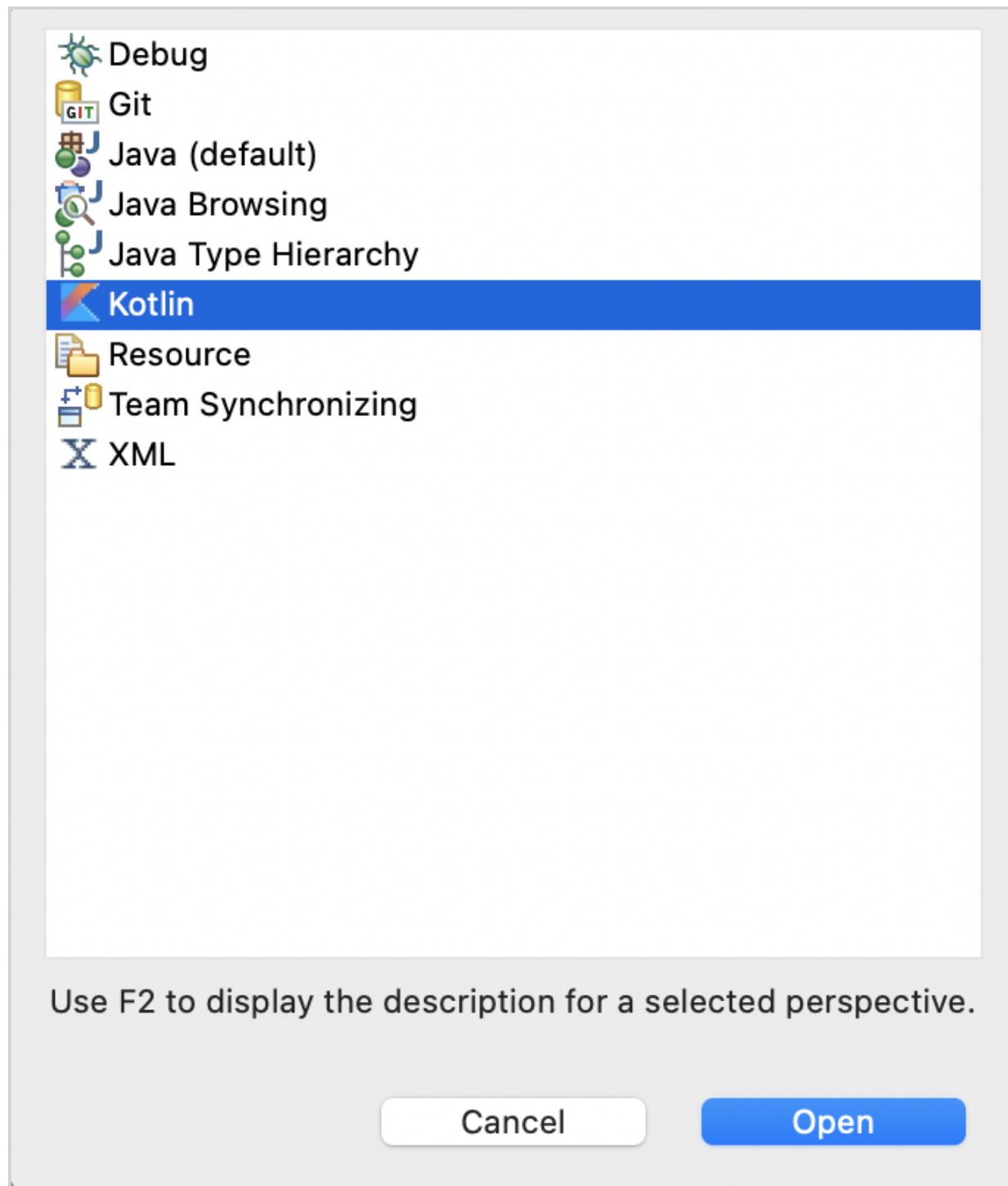
要将 Kotlin 支持添加到 Eclipse IDE 中，请安装 **Kotlin Plugin for Eclipse**。我们建议从 [Eclipse Marketplace](#) 安装 Kotlin 插件。Open the **Help | Eclipse Marketplace...** menu and search for **Kotlin Plugin for Eclipse**:

## 1.5.30 的新特性



安装插件并重新启动 Eclipse 后，请确保插件安装正确：打开菜单 **Window | Perspective | Open Perspective | Other...** 中的 **Kotlin perspective**

### 1.5.30 的新特性

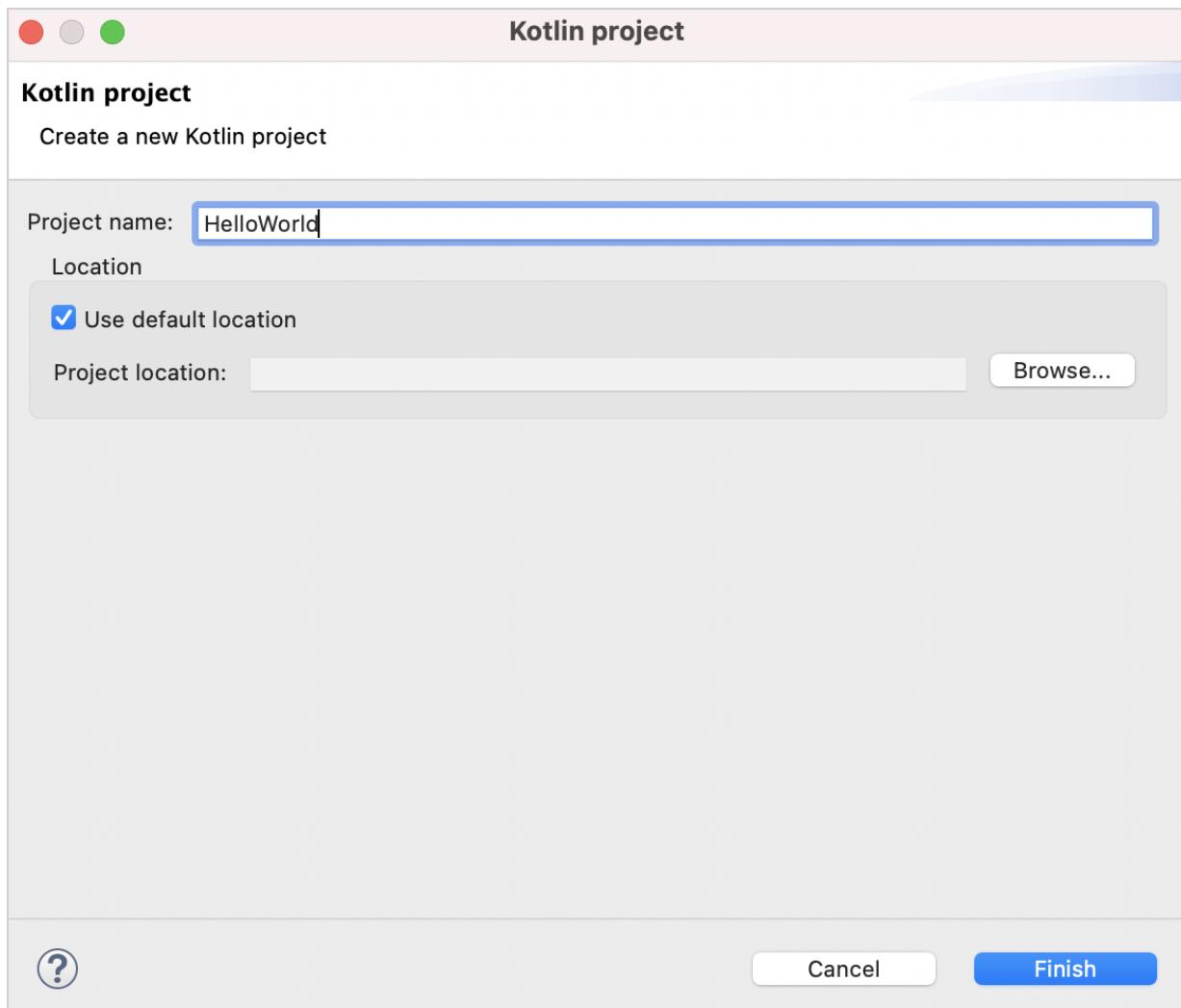


## 创建一个新的项目

现在你已准备好创建一个新的 Kotlin 项目。

1. 选择 **File | New | Kotlin Project**。

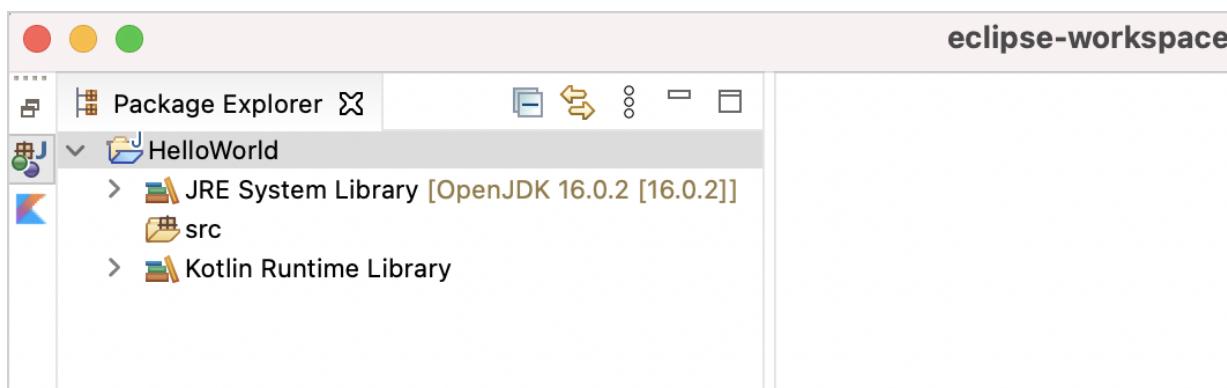
### 1.5.30 的新特性



一个空的 Kotlin/JVM 项目创建完成。

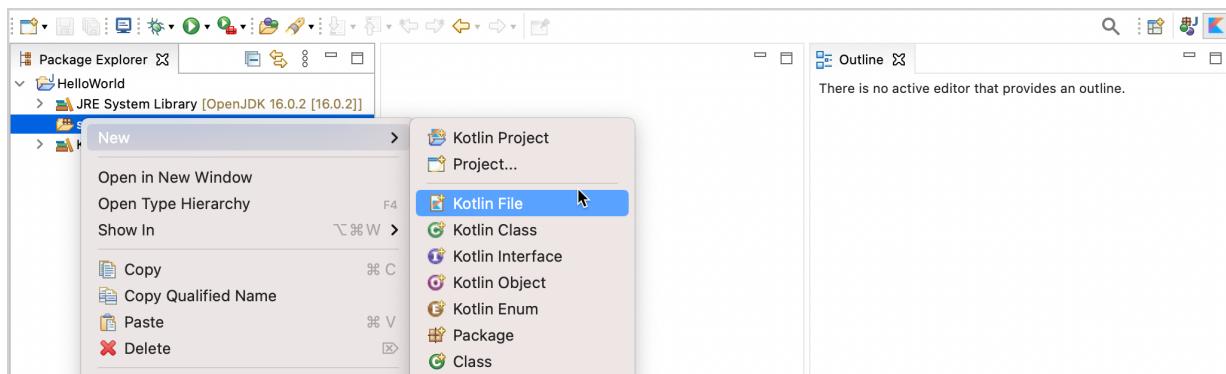
对于 Eclipse IDE，这个项目也是一个 Java 项目但是配置了 Kotlin 特性，意思是它可以构建 Kotlin 并且可以引用 Kotlin 的运行时库。这个解决方案的好处是你可以添加 Kotlin 和 Java 代码到同一个项目。

项目结构如下所示：

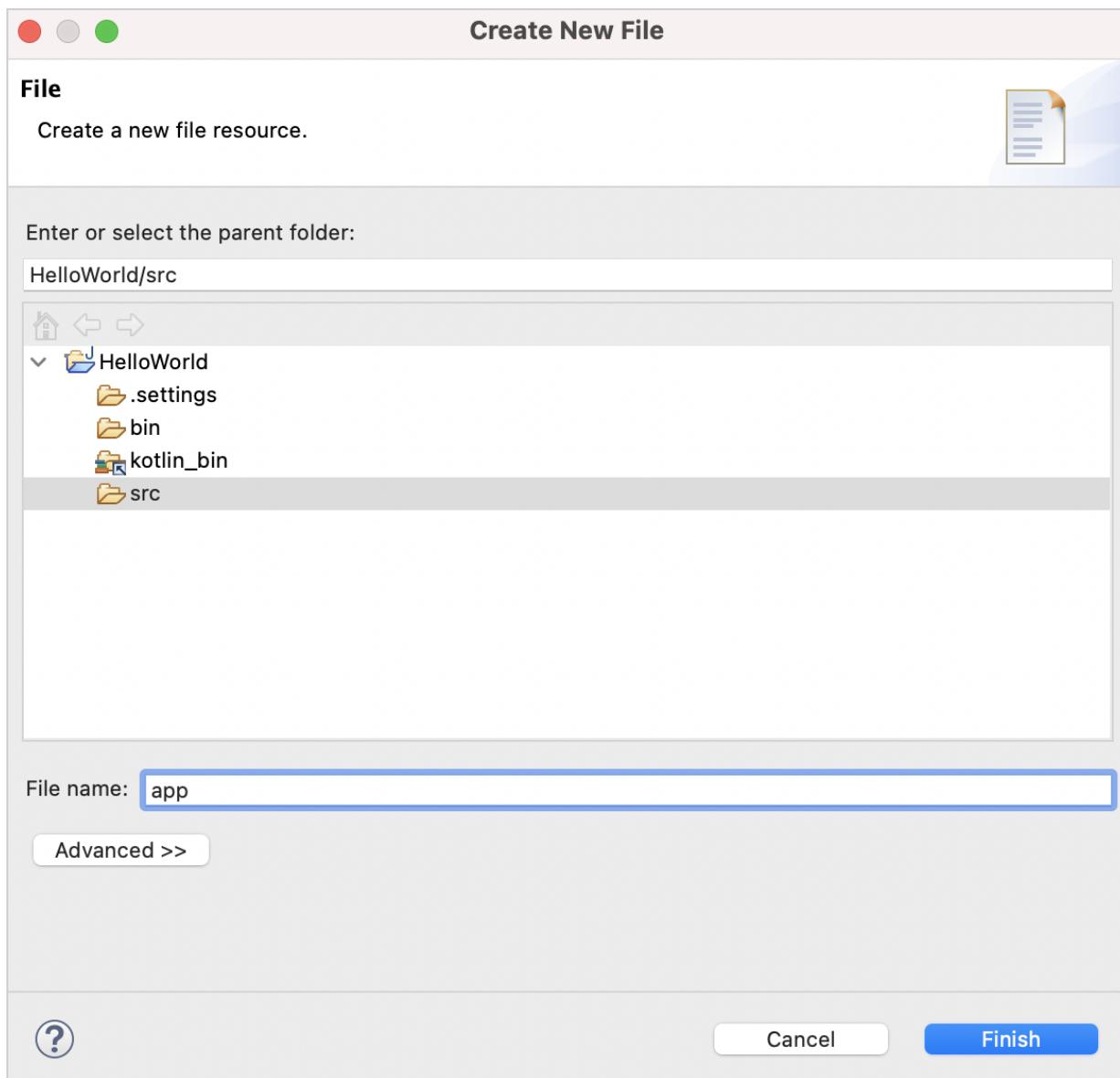


Now, create a new Kotlin file in the source directory.

### 1.5.30 的新特性

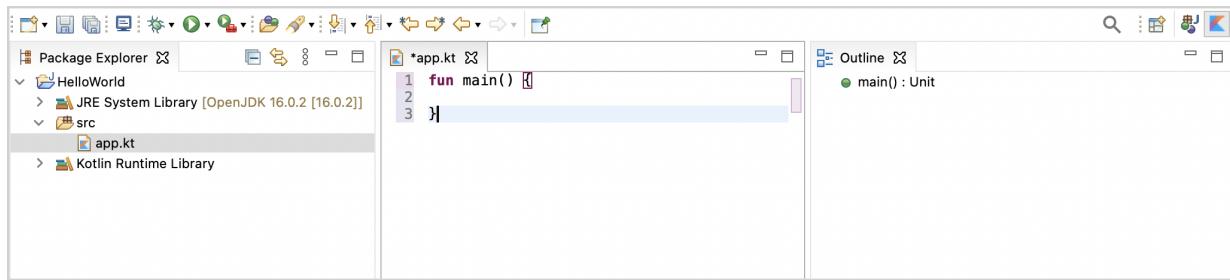


You can enter the name without the `.kt` extension. Eclipse will add it automatically.

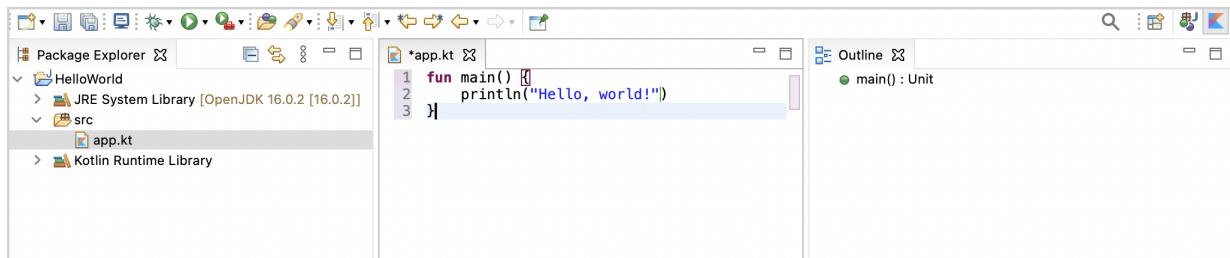


Once you have a source file, add the `main` function - the entry point to a Kotlin application. You can simply type `main` and invoke code completion by hitting `Ctrl + Space`.

## 1.5.30 的新特性

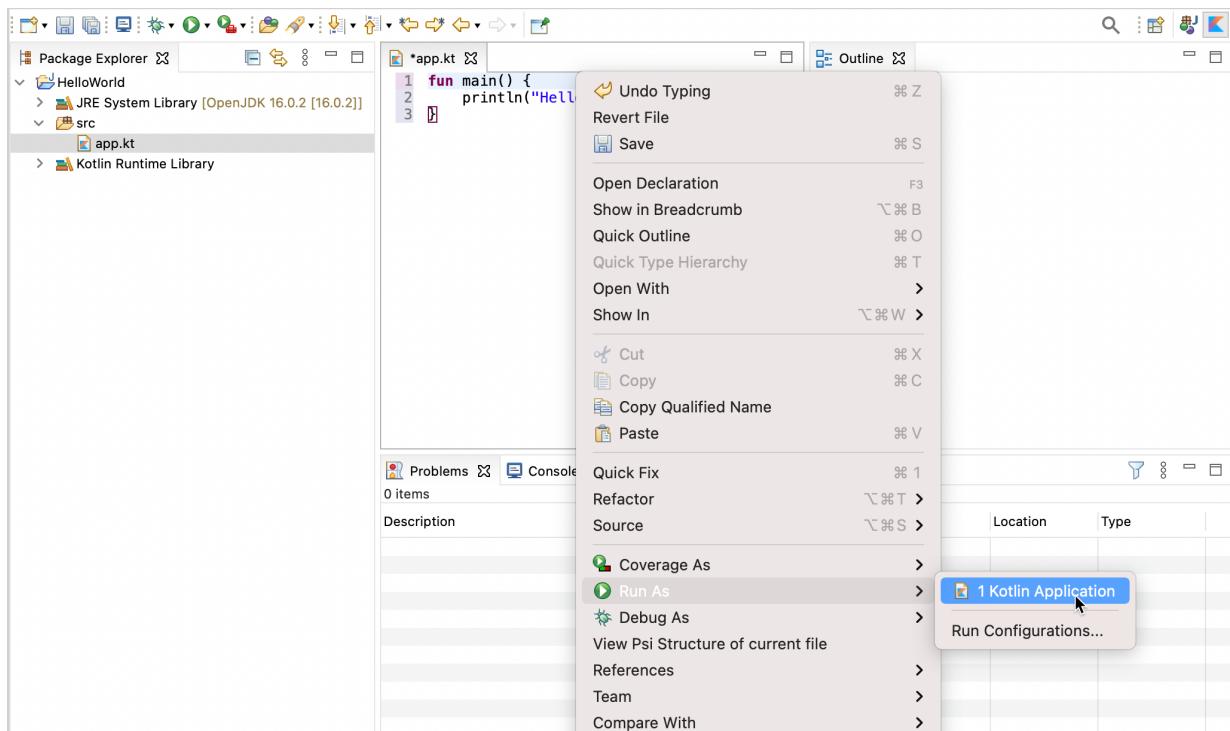


Finally, add a simple line of Kotlin code to print a message:



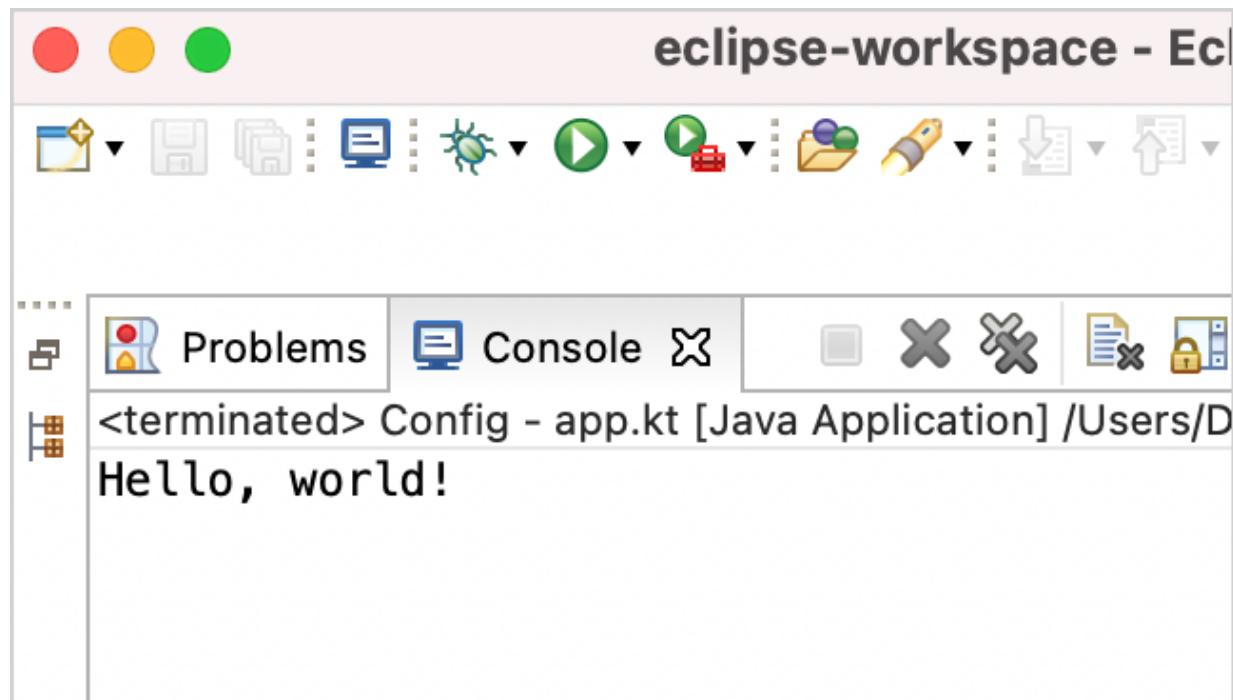
## 运行该应用程序

要运行该应用程序，请右键单击主文件中的某个位置，然后选择 **Run As | Kotlin Application**。



如果一切顺利，你可以在 **Console** 窗口看到返回结果。

### 1.5.30 的新特性



Congratulations! You now have your Kotlin application running in Eclipse IDE.

# 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.
- [Kt 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.

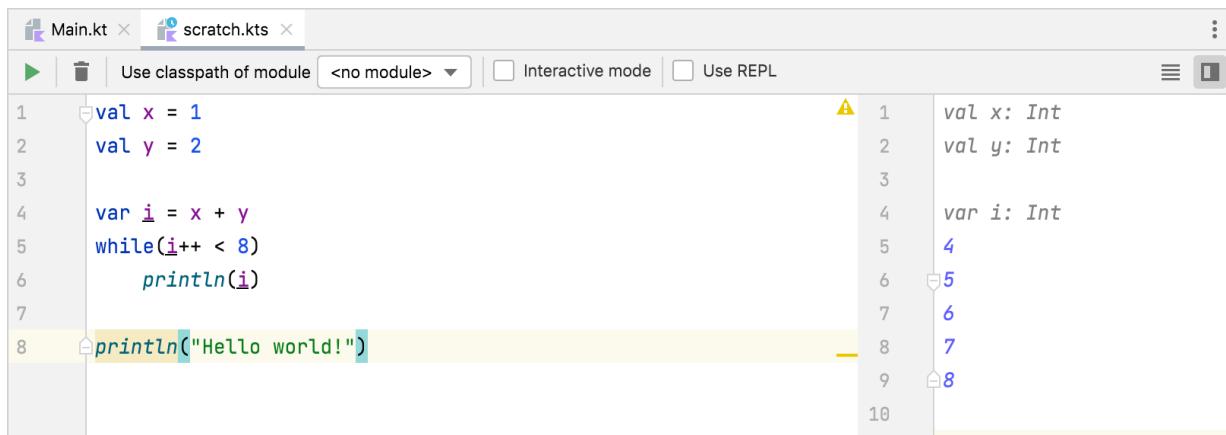
- *Worksheets* are project files: they are stored in project directories and tied to the project modules. Worksheets are useful for writing pieces of code that don't actually make a software unit but should still be stored together in a project, such as educational or demo materials.

To create a Kotlin worksheet in a project directory, right-click the directory in the project tree and select **New | Kotlin Worksheet**.

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.

### 1.5.30 的新特性

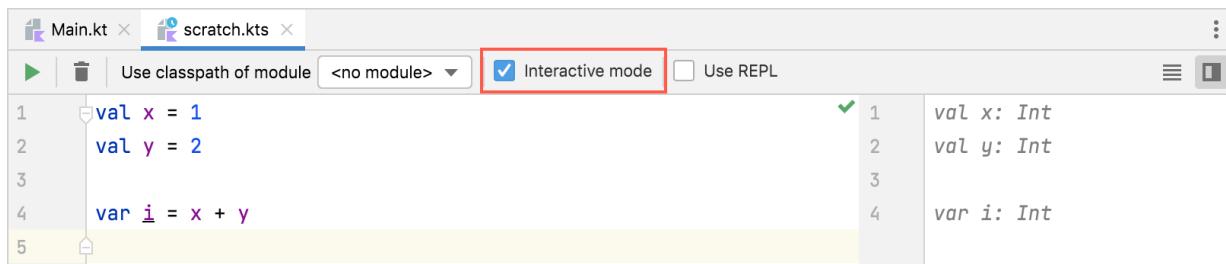


```
Main.kt x scratch.kts x
Use classpath of module <no module> ▾ | Interactive mode | Use REPL
1 val x = 1
2 val y = 2
3
4 var i = x + y
5 while(i++ < 8)
6 println(i)
7
8 println("Hello world!")
```

1	val x: Int
2	val y: Int
3	
4	var i: Int
5	4
6	5
7	6
8	7
9	8
10	

## 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**.



```
Main.kt x scratch.kts x
Use classpath of module <no module> ▾ | Interactive mode | Use REPL
1 val x = 1
2 val y = 2
3
4 var i = x + y
5
```

1	val x: Int
2	val y: Int
3	
4	var i: Int
5	

## Use modules

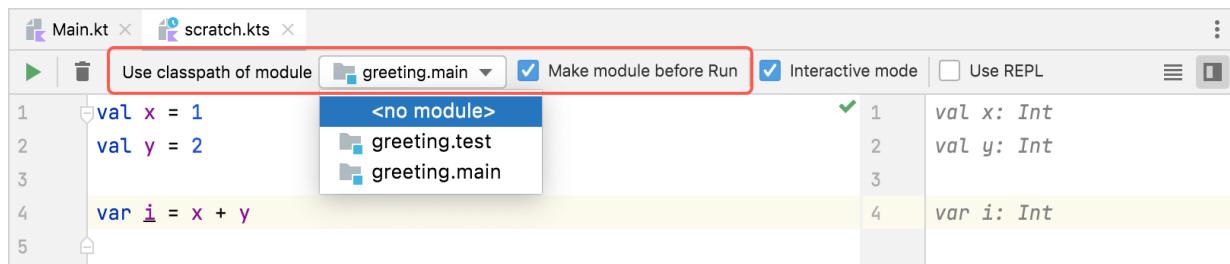
You can use classes or functions from a Kotlin project in your scratches and worksheets.

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.

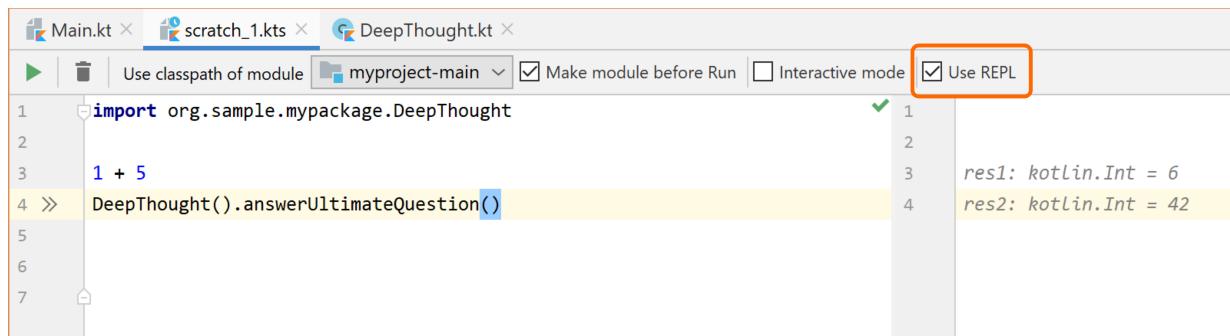
Both scratches and worksheets use the compiled versions of connected modules. So, if you modify a module's source files, the changes will propagate to scratches and worksheets when you rebuild the module. To rebuild the module automatically before each run of a scratch or a worksheet, select **Make module before Run**.

### 1.5.30 的新特性



## 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 referring 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.

## 1.5.30 的新特性

The screenshot shows the Kotlin Playground interface. At the top, there are dropdown menus for '1.6.10' and 'JVM'. A search bar contains the text 'world'. To the right of the search bar are buttons for 'Copy link', 'Share code', and a purple 'Run' button. Below the search bar, the code editor displays the following Kotlin code:

```
class Person(val name: String)

fun greet(person: Person) = println("Hello ${person.name}!")

fun main(args: Array<String>) {
 greet(Person(args[0]))
 listOf(1, 2, 3).filt|
}
```

A code completion dropdown menu is open at the end of the word 'filt'. It lists several options starting with 'filter':

- filter(predicate: (Int) -> Boolean)
- filterIndexed(predicate: (index: Int, Int) -> Boolean)
- filterIndexedTo(destination: C, predicate: (index: Int, In...)
- filterIsInstance()
- filterIsInstance(klass: Class<R>)
- filterIsInstanceTo(destination: C)
- filterIsInstanceTo(destination: C, klass: Class<R>)
- filterNot(predicate: (Int) -> Boolean)
- filterNotNull()
- filterNotNullTo(destination: C)
- filterNotTo(destination: C, predicate: (Int) -> Boolean)
- filterTo(destination: C, predicate: (Int) -> Boolean)

At the bottom right of the code editor is a small circular icon with a question mark.

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.

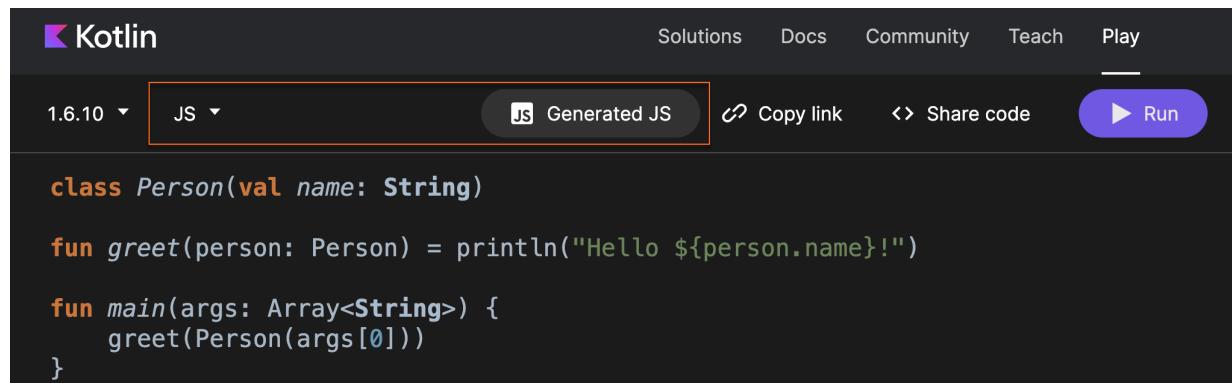
The screenshot shows the Kotlin Playground interface. At the top, there are dropdown menus for '1.6.10' and 'JVM'. A search bar contains the text 'world'. To the right of the search bar are buttons for 'Copy link', 'Share code', and a purple 'Run' button. Below the search bar, the code editor displays the same code as the previous screenshot. On the far left, there is a vertical list of Kotlin version numbers:

- 1.2.71
- 1.3.72
- 1.4.30
- 1.5.31
- 1.6.10

The '1.6.10' option is highlighted with a red border.

For JS backends, you can also see the generated JS code.

## 1.5.30 的新特性



The screenshot shows the Kotlin Playground interface. At the top, there's a navigation bar with links for Solutions, Docs, Community, Teach, and Play. Below the navigation bar, there are dropdown menus for version (1.6.10) and target (JS). A button labeled "Generated JS" is highlighted with an orange border. To the right of this are "Copy link", "Share code", and "Run" buttons. The main area contains the following Kotlin code:

```
class Person(val name: String)

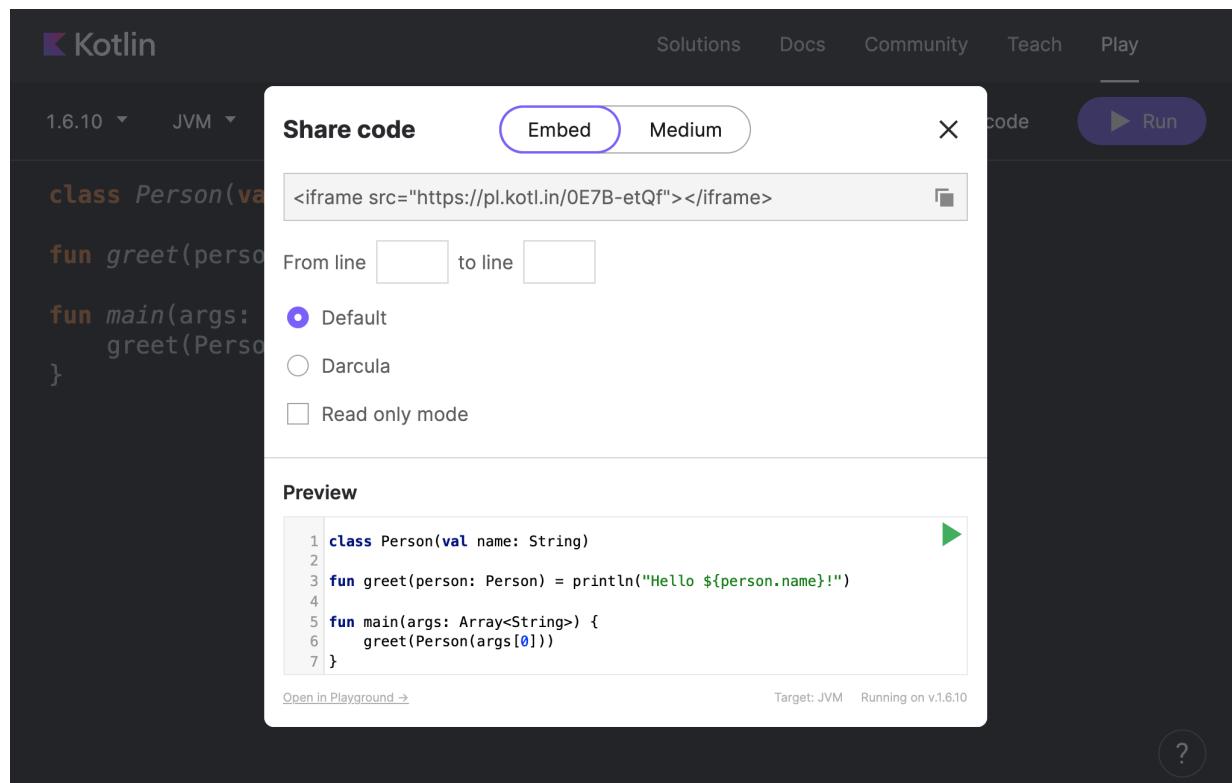
fun greet(person: Person) = println("Hello ${person.name}!")

fun main(args: Array<String>) {
 greet(Person(args[0]))
}
```

## 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.

## 1.5.30 的新特性

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

### 1.5.30 的新特性

(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.

## 1.5.30 的新特性

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]]]
```

## 编译器

- [Kotlin 命令行编译器](#)
- [Kotlin 编译器选项](#)

# Kotlin 命令行编译器

Every Kotlin release ships with a standalone version of the compiler. You can download the latest version (`kotlin-compiler-1.6.10.zip`) from [GitHub Releases](#).

Installing the command-line compiler is not an essential step to use Kotlin. A general way to write Kotlin applications is using an IDE - [IntelliJ IDEA](#) or [Android Studio](#). They provide full Kotlin support out of the box without needing additional components. Learn how to [get started with Kotlin in an IDE](#).



## Install the compiler

### Manual install

Unzip the standalone compiler into a directory and optionally add the `bin` directory to the system path. The `bin` directory contains the scripts needed to compile and run Kotlin on Windows, OS X, and Linux.

### SDKMAN!

An easier way to install Kotlin on UNIX-based systems, such as OS X, 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 OS X you can install the compiler via [Homebrew](#).

```
$ brew update
$ brew install kotlin
```

## Snap package

If you use [Snap](#) on Ubuntu 16.04 or later, you can install the compiler from the command line:

```
$ sudo snap install --classic kotlin
```

## Create and run an application

1. Create a simple application in Kotlin that displays "Hello, World!". In your favorite editor, create a new file called `hello.kt` with the following lines:

```
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
```

## Compile a library

If you're developing a library to be used by other Kotlin applications, you can build the `.jar` file without including the Kotlin runtime in it.

### 1.5.30 的新特性

```
$ kotlinc hello.kt -d hello.jar
```

Since binaries compiled this way depend on the Kotlin runtime, you should make sure the latter 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`.

## Run the 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.

```
[Ocean] ~/tutorials/kotlin/command_line/kotlinc$ bin/kotlinc-jvm
Kotlin interactive shell
Type :help for help, :quit for quit
>>> 2+2
4
>>> println("Welcome to the Kotlin Shell")
Welcome to the Kotlin Shell
>>>
```

## Run scripts

Kotlin can also be used as a scripting language. A 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) }
```

## 1.5.30 的新特性

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.scrip
```

For additional details, please consult 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.

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 `kotlinOptions` property of the Kotlin compilation task. For details, see [Gradle](#).
- 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
```

### 1.5.30 的新特性

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.

### **-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 *pluginId:optionName=value***

Pass an option to a Kotlin compiler plugin. Available plugins and their options are listed in the **Tools > 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
```

### 1.5.30 的新特性

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.

# 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.

1.5.30 的新特性

## **-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`.

## **-jvm-target *version***

Specify the target version of the generated JVM bytecode. Possible values are `1.6` (DEPRECATED), `1.8`, `9`, `10`, `11`, `12`, `13`, `14`, `15`, `16`, and `17`. 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*[,]**

## 1.5.30 的新特性

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.

## **-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.

1.5.30 的新特性

## **-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-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)**

### 1.5.30 的新特性

Enable runtime assertions in the generated code.

## **-g**

Enable emitting debug information.

## **-generate-test-runner (-tr)**

Produce an application for running unit tests from the project.

## **-generate-worker-test-runner (-trw)**

Produce an application for running unit tests in a [worker thread](#).

## **-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 [default 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**

### 1.5.30 的新特性

Enable compilation optimizations.

## **-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
- 使用 kapt
- Lombok 编译器插件

# 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`. *All-open* allows to specify such annotations.

We provide *all-open* plugin support both for Gradle and Maven with the complete IDE integration.

For Spring, you can use the `kotlin-spring` compiler plugin ([see below](#)).



## Gradle

Add the plugin artifact to the build script dependencies and apply the plugin:

```
buildscript {
 dependencies {
 classpath "org.jetbrains.kotlin:kotlin-allopen:$kotlin_version"
 }
}

apply plugin: "kotlin-allopen"
```

As an alternative, you can enable it using the `plugins` block:

```
plugins {
 id "org.jetbrains.kotlin.plugin.allopen" version "1.6.10"
}
```

Then specify the list of annotations that will make classes open:

### 1.5.30 的新特性

```
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

Here's how to use all-open with Maven:

### 1.5.30 的新特性

```
<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.

As with all-open, add the plugin to the build script dependencies:

```
buildscript {
 dependencies {
 classpath "org.jetbrains.kotlin:kotlin-allopen:$kotlin_version"
 }
}

apply plugin: "kotlin-spring" // instead of "kotlin-allopen"
```

### 1.5.30 的新特性

Or using the Gradle plugins DSL:

```
plugins {
 id "org.jetbrains.kotlin.plugin.spring" version "1.6.10"
}
```

In Maven, the `spring` plugin is provided by the `kotlin-maven-allopen` plugin dependency, so to enable it:

```
<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.

Note that if you use the project template generated by the [start.spring.io](https://start.spring.io) service, the `kotlin-spring` plugin will be enabled by default.

## Command-line compiler

## 1.5.30 的新特性

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 presets available now for all-open are `spring` , `micronaut` , and `quarkus` .

```
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
```

## 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](#)).

## Gradle

Add the plugin and specify the list of annotations that must lead to generating a no-arg constructor for the annotated classes.

```
buildscript {
 dependencies {
 classpath "org.jetbrains.kotlin:kotlin-noarg:$kotlin_version"
 }
}

apply plugin: "kotlin-noarg"
```

Or using the Gradle plugins DSL:

```
plugins {
 id "org.jetbrains.kotlin.plugin.noarg" version "1.6.10"
}
```

Then specify the list of no-arg annotations:

```
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.

### 1.5.30 的新特性

```
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.

That's how you add the plugin in Gradle:

## 1.5.30 的新特性

```
buildscript {
 dependencies {
 classpath "org.jetbrains.kotlin:kotlin-noarg:$kotlin_version"
 }
}

apply plugin: "kotlin-jpa"
```

Or using the Gradle plugins DSL:

```
plugins {
 id "org.jetbrains.kotlin.plugin.jpa" version "1.6.10"
}
```

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.

### 1.5.30 的新特性

```
buildscript {
 dependencies {
 classpath "org.jetbrains.kotlin:kotlin-sam-with-receiver:$kotlin_version"
 }
}

apply plugin: "kotlin-sam-with-receiver"
```

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

### 1.5.30 的新特性

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
```

## 使用 kapt

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.

## Using in Gradle

Apply the `kotlin-kapt` Gradle plugin:

### 【Kotlin】

```
plugins {
 kotlin("kapt") version "1.6.10"
}
```

### 【Groovy】

```
plugins {
 id "org.jetbrains.kotlin.kapt" version "1.6.10"
}
```

Alternatively, you can use the `apply plugin` syntax:

```
apply plugin: 'kotlin-kapt'
```

Then add the respective dependencies using the `kapt` configuration in your `dependencies` block:

### 1.5.30 的新特性

#### 【Kotlin】

```
dependencies {
 kapt("groupId:artifactId:version")
}
```

#### 【Groovy】

```
dependencies {
 kapt 'groupId:artifactId:version'
}
```

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.

## 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
 }
```

## Improving the speed of builds that use kapt

### Running 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. However, running kapt with Gradle worker API enabled can result in increased memory consumption due to parallel execution.

To use the Gradle worker API for parallel execution of kapt tasks, add this line to your `gradle.properties` file:

```
kapt.use.worker.api=true
```

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.

### Caching for annotation processors' classloaders

在 kapt 中缓存注解处理器的类加载器是 [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:

### 1.5.30 的新特性

```
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]
```

## 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

### 1.5.30 的新特性

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.

## 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` file to enable error type inferring in stubs:

```
kapt {
 correctErrorTypes = true
}
```

## Using in Maven

### 1.5.30 的新特性

Add an execution of the `kapt` goal from kotlin-maven-plugin before `compile`:

```
<execution>
 <id>kapt</id>
 <goals>
 <goal>kapt</goal>
 </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>
```

Please note that kapt is still 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.

## Using 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.

### 1.5.30 的新特性

- `apclasspath` (*repeatable*): A path to the annotation processor JAR. Pass as many `apclasspath` options as many JARs 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`: See [below](#). Disabled by default.

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
```

## Generating 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

## 1.5.30 的新特性

`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())
}
```

## Keeping 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 `keepJavaAnnotationProcessors` :

```
kapt {
 keepJavaAnnotationProcessors = 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 make the plugin seeing Lombok's config](#).

## Supported annotations

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.

Kotlin compiler ignores Lombok annotations if you use them in Kotlin code.



## Gradle

### 1.5.30 的新特性

Apply the `kotlin-plugin-lombok` Gradle plugin in the `build.gradle(.kts)` file:

#### 【Kotlin】

```
plugins {
 kotlin("plugin.lombok") version "1.6.10"
 id("io.freefair.lombok") version "5.3.0"
}
```

#### 【Groovy】

```
plugins {
 id 'org.jetbrains.kotlin.plugin.lombok' version '1.6.10'
 id 'io.freefair.lombok' version '5.3.0'
}
```

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`, provide a path to it to the plugin. The path should be relative to the module's directory. 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

### 1.5.30 的新特性

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:

## 1.5.30 的新特性

```
 kapt {
 keepJavaAnnotationProcessors = 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](#)

## 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 2 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.

### 1.5.30 的新特性

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:

### 1.5.30 的新特性

```
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

### 1.5.30 的新特性

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:

### 1.5.30 的新特性

```
class HelloFunctionFinderProcessor : SymbolProcessor() {
 // ...
 val functions = mutableListOf<String>()
 val visitor = FindFunctionsVisitor()

 override fun process(resolver: Resolver) {
 resolver.getAllFiles().map { it.accept(visitor, Unit) }
 }

 inner class FindFunctionsVisitor : KSVisitorVoid() {
 override fun visitClassDeclaration(classDeclaration: KSClassDeclaration, data: Unit) {
 classDeclaration.getDeclaredFunctions().map { it.accept(this, Unit) }
 }

 override fun visitFunctionDeclaration(function: KSFunctionDeclaration, data: Unit) {
 functions.add(function)
 }

 override fun visitFile(file: KSFile, data: Unit) {
 file.declarations.map { it.accept(this, Unit) }
 }
 }
 // ...
}

class Provider : SymbolProcessorProvider {
 override fun create(environment: SymbolProcessorEnvironment): SymbolProcessor
}
}
```

## 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 below includes a list of popular libraries on Android and their various stages of support for KSP.

Library	Status	Tracking issue for KSP
Room	<a href="#">Officially supported</a>	
Moshi	<a href="#">Officially supported</a>	
RxHttp	<a href="#">Officially supported</a>	
Kotshi	<a href="#">Officially supported</a>	
Lyricist	<a href="#">Officially supported</a>	
Lich SavedState	<a href="#">Officially supported</a>	
gRPC Dekorator	<a href="#">Officially supported</a>	
EasyAdapter	<a href="#">Officially supported</a>	
Auto Factory	Not yet supported	<a href="#">Link</a>
Dagger	Not yet supported	<a href="#">Link</a>
Hilt	Not yet supported	<a href="#">Link</a>
Glide	Not yet supported	<a href="#">Link</a>
DeeplinkDispatch	<a href="#">Supported via airbnb/DeepLinkDispatch#323</a>	
Micronaut	In Progress	<a href="#">Link</a>

# KSP 快速入门

For a quick start, you can create your own processor or get a [sample one](#).

## Create a processor of your own

1. Create an empty gradle project.
2. Specify version `1.6.10` of the Kotlin plugin in the root project for use in other project modules:

### 【Kotlin】

```
plugins {
 kotlin("jvm") version "1.6.10" apply false
}

buildscript {
 dependencies {
 classpath(kotlin("gradle-plugin", version = "1.6.10"))
 }
}
```

### 【Groovy】

```
plugins {
 id 'org.jetbrains.kotlin.jvm' version '1.6.10' apply false
}

buildscript {
 dependencies {
 classpath 'org.jetbrains.kotlin:kotlin-gradle-plugin:1.6.10'
 }
}
```

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】

## 1.5.30 的新特性

```
plugins {
 kotlin("jvm")
}

repositories {
 mavenCentral()
}

dependencies {
 implementation("com.google.devtools.ksp:symbol-processing-api:1.6.10-1.0.2")
}
```

### 【Groovy】

```
plugins {
 id 'org.jetbrains.kotlin.jvm' version '1.6.10'
}

repositories {
 mavenCentral()
}

dependencies {
 implementation 'com.google.devtools.ksp:symbol-processing-api:1.6.10-1.0.2'
}
```

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:
  - o Implement `SymbolProcessorProvider.create()` to create a `SymbolProcessor`. Pass dependencies that your processor needs (such as `CodeGenerator`, processor options) through the parameters of `SymbolProcessorProvider.create()`.
  - o Your main logic should be in the `SymbolProcessor.process()` method.
  - o Use `resolver.getSymbolsWithAnnotation()` to get the symbols you want to process, given the fully-qualified name of an annotation.
  - o 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`.
  - o For sample implementations of the `SymbolProcessorProvider` and `SymbolProcessor` interfaces, see the following files in the sample project.

### 1.5.30 的新特性

- `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 `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 "1.6.10-1.0.2"
}

dependencies {
 implementation(kotlin("stdlib-jdk8"))
 implementation(project(":test-processor"))
 ksp(project(":test-processor"))
}
```

## 1.5.30 的新特性

### 【Groovy】

```
plugins {
 id 'com.google.devtools.ksp' version '1.6.10-1.0.2'
}

dependencies {
 implementation 'org.jetbrains.kotlin:kotlin-stdlib:$kotlin_version'
 implementation project(':test-processor')
 ksp project(':test-processor')
}
```

1. Run `./gradlew build`. You can find the generated code under `build/generated/source/ksp .`

Here's a sample build script to apply the KSP plugin to a workload:

### 【Kotlin】

```
plugins {
 id("com.google.devtools.ksp") version "1.6.10-1.0.2"
 kotlin("jvm")
}

repositories {
 mavenCentral()
}

dependencies {
 implementation(kotlin("stdlib-jdk8"))
 implementation(project(":test-processor"))
 ksp(project(":test-processor"))
}
```

### 【Groovy】

### 1.5.30 的新特性

```
plugins {
 id 'com.google.devtools.ksp' version '1.6.10-1.0.2'
 id 'org.jetbrains.kotlin.jvm' version '1.6.10'
}

repositories {
 mavenCentral()
}

dependencies {
 implementation 'org.jetbrains.kotlin:kotlin-stdlib:1.6.10'
 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

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/
```

## 1.5.30 的新特性

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 c
Gradle detected a problem with the following location: '../build/generated/ksp/main'
Reason: Task ':publishPluginJar' uses this output of task ':kspKotlin' without decl
```

In this case, use the following script instead:

### 【Kotlin】

## 1.5.30 的新特性

```
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
 testSourceDirs = testSourceDirs + file("build/generated/ksp/test/kotlin")
 generatedSourceDirs = generatedSourceDirs + file("build/generated/ksp/main/ko
 }
}
```

## 【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
 testSourceDirs = testSourceDirs + file('build/generated/ksp/test/kotlin')
 generatedSourceDirs = generatedSourceDirs + file('build/generated/ksp/main/ko
 }
}
```

## 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 `kotlinx compiler plugins`

`kotlinx 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

### 1.5.30 的新特性

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%.

### 1.5.30 的新特性

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.

We are also exploring several additional features. These features are currently unavailable:

- IDE integration: Currently IDEs know nothing about the generated code.

## KSP 示例

### Get all member functions

```
fun KSClassDeclaration.getDeclaredFunctions(): List<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
 }
}
```

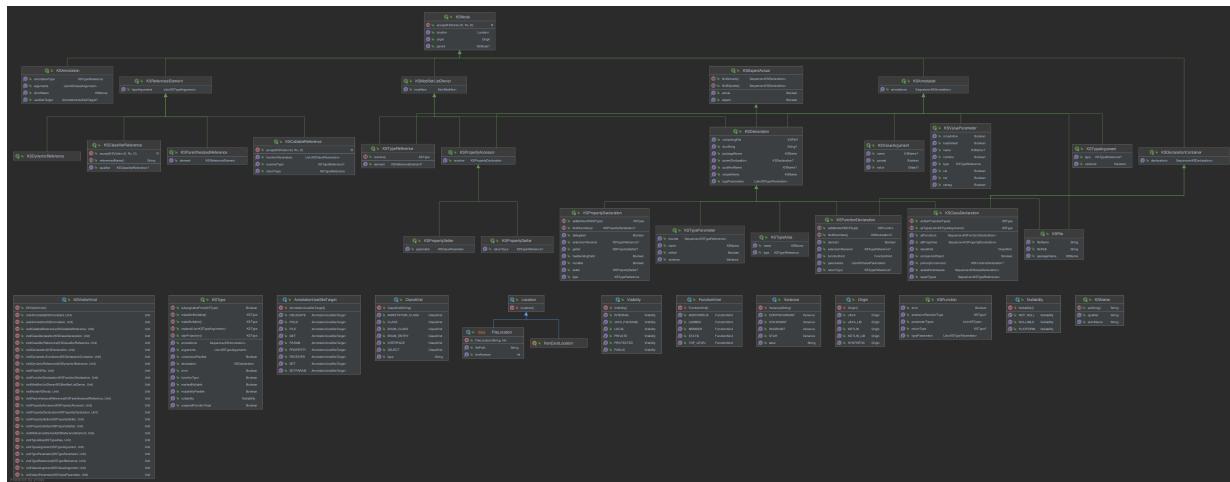
### Collect suppressed names in a file annotation

## 1.5.30 的新特性

```
// @file:kotlin.Suppress("Example1", "Example2")
fun KSFile.suppressedNames(): List<String> {
 val ignoredNames = mutableListOf<String>()
 annotations.filter {
 it.shortName.asString() == "Suppress" && it.annotationType.resolve()?.declaration != null
 }.forEach {
 val argValues: List<String> = it.arguments.flatMap { it.value }
 ignoredNames.addAll(argValues)
 }
 return ignoredNames
}
```

# 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:



# 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 `KTypeReference` can be resolved to a `KType`, 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.

### 1.5.30 的新特性

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.

### 1.5.30 的新特性

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

### 1.5.30 的新特性

Java	Closest facility in KSP	Notes
Name	KSName	
ElementKind	ClassKind / FunctionKind	
Modifier	Modifier	
NestingKind	ClassKind / FunctionKind	
AnnotationValueVisitor		
ElementVisitor	KSPVisitor	
AnnotatedConstruct	KSAnnotated	
TypeVisitor		
TypeKind	KSBuiltIns	Some can be found in builtins, otherwise check KSClassDeclaration for DeclaredType
ElementFilter	Collection.filterIsInstance	
ElementKindVisitor	KSPVisitor	
ElementScanner	KSTopDownVisitor	
SimpleAnnotationValueVisitor		Not needed in KSP
SimpleElementVisitor	KSPVisitor	
SimpleTypeVisitor		
TypeKindVisitor		
Types	Resolver / utils	Some of the utilities 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

### 1.5.30 的新特性

## 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

### 1.5.30 的新特性

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 == ClassKindINTERFACE } ```
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 == ClassKindCLASS } ```
getTypeParameters	ksClassDeclaration.typeParameters

## TypeParameterElement

Java	KSP equivalent
getBounds	ksTypeParameter.bounds
getEnclosingElement	ksTypeParameter.parentDeclaration
getGenericElement	ksTypeParameter.parentDeclaration

## VariableElement

### 1.5.30 的新特性

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

### 1.5.30 的新特性

## TypeMirror

Java	KSP equivalent
getKind	Compare with types in <code>KSBuiltIns</code> for primitive types, <code>Unit</code> type, otherwise declared types

## TypeVariable

Java	KSP equivalent
asElement	<code>ksType.declaration</code>
getLowerBound	To be decided. Only needed if capture is provided and explicit bound checking is needed.
getUpperBound	<code>ksTypeParameter.bounds</code>

## WildcardType

Java	KSP equivalent
getExtendsBound	<code>```kotlin if (ksTypeArgument.variance == Variance.COVARIANT) ksTypeArgument.type else null ```</code>
getSuperBound	<code>```kotlin if (ksTypeArgument.variance == Variance.CONTRAVARIANT) ksTypeArgument.type else null ```</code>

## Elements

### 1.5.30 的新特性

Java	KSP equivalent
getAllAnnotationMirrors	KSDeclarations.annotations
getAllMembers	getAllFunctions , getAllProperties is to be implemented
getBinaryName	To be decided, see <a href="#">Java Specification</a>
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 retrieved. Operation on package is not possible for now.
getPackageOf	Package not supported
getTypeElement	Resolver.getClassDeclarationByName
hides	To be implemented
isDeprecated	<pre>```kotlin KsDeclaration.annotations.any {     it.annotationType.resolve()!!..declaration.qualifiedName == Deprecated::class.qualifiedName }```</pre>
overrides	KSFunctionDeclaration.overrides / KSPropertyDeclaration.overrides (member function class)
printElements	KSP has basic <code>toString()</code> implementation on most elements

## Types

## 1.5.30 的新特性

Java	KSP equivalent
asElement	<code>ksType.declaration</code>
asMemberOf	<code>resolver.asMemberOf</code>
boxedClass	Not needed
capture	To be decided
contains	<code>KSType.isAssignableFrom</code>
directSuperTypes	<code>(ksType.declaration as KSClassDeclaration).superTypes</code>
erasure	<code>ksType.starProjection()</code>
getArrayType	<code>ksBuiltIns.arrayType.replace(...)</code>
getDeclaredType	<code>ksClassDeclaration.asType</code>
getNoType	<code>ksBuiltIns.nothingType / null</code>
getNullType	Depending on the context, <code>KSType.markNullable</code> could be useful
getPrimitiveType	Not needed, check for <code>KSBuiltins</code>
getWildcardType	Use <code>Variance</code> in places expecting <code>KSTypeArgument</code>
isAssignable	<code>ksType.isAssignableFrom</code>
isSameType	<code>ksType.equals</code>
isSubsignature	<code>functionTypeA == functionTypeB / functionTypeA == functionTypeB.starProjection()</code>
isSubtype	<code>ksType.isAssignableFrom</code>
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`

Currently, only changes in Kotlin and Java sources are tracked. Changes to the classpath, namely to other modules or libraries, trigger a full re-processing of all sources by default. To track changes in classpath, set the Gradle property

`ksp.incremental.intermodule=true` for an experimental implementation on JVM.



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 folder with a `.log` file extension.

## 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.

### 1.5.30 的新特性

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.

### 1.5.30 的新特性

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 KSClassDeclaration
 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", outputName)
 output.write("// $declA : $declB\n".toByteArray())
 output.close()
 }
 // ...
}
```

## Example 2

### 1.5.30 的新特性

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`

### 1.5.30 的新特性

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.

### 1.5.30 的新特性

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

### 1.5.30 的新特性

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 `KSPLocator.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 KSPLocator. 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

### 1.5.30 的新特性

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 {
 withJava()
 }
 linuxX64() {
 binaries {
 executable()
 }
 }
 sourceSets {
 val commonMain by getting
 val linuxX64Main by getting
 val linuxX64Test by getting
 }
}

dependencies {
 add("kspMetadata", project(":test-processor"))
 add("kspJvm", project(":test-processor"))
 add("kspJvmTest", project(":test-processor")) // Not doing anything because the
 // There is no processing for the Linux x64 main source set, because kspLinuxX6
 add("kspLinuxX64Test", project(":test-processor"))
}
```

# Compilation and processing

### 1.5.30 的新特性

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 compilations: 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/v1.6.10/kotlin-compiler-
unzip kotlin-compiler-1.6.10.zip

KSP
wget https://github.com/google/ksp/releases/download/1.6.10-1.0.2/artifacts.zip
unzip artifacts.zip
```

To run KSP with `kotlinc`, pass the `-Xplugin` option to `kotlinc`.

```
-Xplugin=/path/to/symbol-processing cmdline-1.6.10-1.0.2.jar
```

This is different from the `symbol-processing-1.6.10-1.0.2.jar`, which is designed to be used with `kotlin-compiler-embeddable` when running with Gradle. The command line `kotlinc` needs `symbol-processing cmdline-1.6.10-1.0.2.jar`.

You'll also need the API jar.

```
-Xplugin=/path/to/symbol-processing api-1.6.10-1.0.2.jar
```

See the complete example:

## 1.5.30 的新特性

```
#!/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/1.6.10-1.0.2/sym
KSP_API_JAR=./com/google/devtools/ksp/symbol-processing-api/1.6.10-1.0.2/symbol-pro
KOTLINC=./kotlin/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 \
*
```

# 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:

### 1.5.30 的新特性

```
ksp-a.b.c is too old for kotlin-x.y.z. Please upgrade ksp or downgrade kotlin-gradl
```

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 (e.g., M1/M2/RC).

## Besides Kotlin, are there other version requirements to libraries?

Here is a list of requirements for libraries/infrastructures:

- Android Gradle Plugin 4.1.0+
- Gradle 6.5+

## 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!

# 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。

### Build Step

The screenshot shows the 'Build Step' configuration screen. It includes fields for 'Runner type' (set to 'Gradle'), 'Step name' (set to 'Build and Test'), 'Execute step' (set to 'If all previous steps finished successfully'), and 'Gradle Parameters' sections for 'Gradle tasks' (containing 'clean jar test distZip distJar publish') and 'Gradle build file' (containing 'Path to build file').

<b>Runner type:</b>	Gradle
Runner for Gradle projects	
<b>Step name:</b>	Build and Test
Optional, specify to distinguish this build step from other steps.	
<b>Execute step:</b> ⓘ	If all previous steps finished successfully
Specify the step execution policy.	
<b>Gradle Parameters</b>	
<b>Gradle tasks:</b>	clean jar test distZip distJar publish
Enter task names separated by spaces, leave blank to use the 'default' task. Example: ':myproject:clean :myproject:build' or 'clean build'.	
<b>Gradle build file:</b>	Path to build file

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

### 1.5.30 的新特性

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

**Project Settings**

**General Settings**

**VCS Roots 1**

**Report Tabs 4**

Parameters

Builds Schedule

Issue Trackers 3

Maven Settings 1

SSH Keys

Shared Resources

**Meta-Runners 2**

Clean-up Rules

Versioned Settings

**Meta-Runners**

Meta-Runner is a generalized build step that can be used across different configuration [?](#)

**Upload Meta-Runner**

There are no meta-runners defined in the current project.

Meta-runners inherited from <Root project>:

Runner ID	Name
ClassesVersionCheckerMR	Java Classes Version Checker Runs Java Classes Version checker tool
kotlinc	Setup kotlinc Downloads and registers Kotlin compiler into Inte...

## 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。

### New Build Step

**Runner type:**

Downloads and registers Kotlin compiler into IntelliJ IDEA build runner

**Step name:**

Optional, specify to distinguish this build step from other steps.

**Execute step:** [②](#)

Specify the step execution policy.

**Kotlin Version**

Select Kotlin build tag, e.g. 'M9 or bootstrap'

### 1.5.30 的新特性

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。

### New Build Step

Runner type: **IntelliJ IDEA Project** 

Runner for IntelliJ IDEA projects

Step name:  Optional, specify to distinguish this build step from other steps.

Project Settings

Path to the project:  

Should reference path to project file (.ipr) or project directory for directory-based (.idea) the checkout directory. Leave empty if .idea directory is right under the checkout direct

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

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。

## 生成文档

Kotlin 的文档生成工具称为 [Dokka](#)。其使用说明请参见 [Dokka README](#)。

Dokka 有 Gradle、Maven 与 Ant 的插件，因此你可以将文档生成集成到你的构建过程中。

## 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：

## 1.5.30 的新特性

```
/**
 * 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

### 1.5.30 的新特性

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），请使用 Markdown 引用样式语法：

为此目的，请使用`[这个方法] [foo]`。

你还可以在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。

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 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。

## 模块和包文档

作为一个整体的模块、以及该模块中的包的文档，由单独的 Markdown 文件提供，并且使用 `-include` 命令行参数或 Ant、Maven 和 Gradle 中的相应插件将该文件的路径传递给 Dokka。

在该文件内部，作为一个整体的模块和分开的软件包的文档由相应的一级标题引入。标题的文本对于模块必须是 **Module** <模块名>，对于包必须是 **Package** <限定的包名>。

以下是该文件的一个示例内容：

## 1.5.30 的新特性

```
Module kotlin-demo
```

该模块显示 Dokka 语法的用法。

```
Package org.jetbrains.kotlin.demo
```

包含各种有用的东西。

```
二级标题
```

这个标题下的文本也是 `org.jetbrains.kotlin.demo` 文档的一部分。

```
Package org.jetbrains.kotlin.demo2
```

另一个包中有用的东西。

# 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：

## 1.5.30 的新特性

```
compile "org.jetbrains.kotlin:kotlin-osgi-bundle:$kotlinVersion"
```

要排除作为传递依赖的默认 Kotlin 库，你可以使用以下方法：

```
dependencies {
 compile (
 [group: 'some.group.id', name: 'some.library', version: '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。

# 学习资料

- 学习资料概述
- 例学 Kotlin ↗
- Kotlin 心印
- Kotlin 基础课程 ↗
- Advent of Code 谜题
- 图书
- 在 IDE 中学习 (EduTools)
  - 学习 Kotlin
  - 讲授 Kotlin

## 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.
- [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.
- [Kotlin by example](#) – review a set of small and simple annotated examples for the Kotlin syntax.
- [Kotlin Basics track](#) – learn all the Kotlin essentials while creating working applications step by step on JetBrains Academy.
- [Advent of Code puzzles](#) – learn idiomatic Kotlin and practice your language skills by completing short and fun tasks.
- [Kotlin books](#) – find books we've reviewed and recommend for learning Kotlin.
- [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](#) – course for developers with experience in Java. It shows the similarities between the two languages and focuses on what's going to be different.
- [Kotlin documentation in the PDF format](#) – read the whole documentation offline.

## 例学 Kotlin

 [例学 Kotlin](#)

## Kotlin 心印

Kotlin 心印是一系列让你熟悉 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。可以通过下列两种方式之一使用 Kotlin 心印：

1. 可以使用 Kotlin 心印的[在线版](#)。
2. 可以通过[安装 EduTools 插件](#)并[选择 Kotlin Koans 课程](#)来完成 IntelliJ IDEA 或 Android Studio 中的任务。

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”，而在 EduTools 插件内选择“Peek solution”。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！

1.5.30 的新特性

# Kotlin 基础课程

 [Kotlin 基础课程](#)

# 用地道 Kotlin 代码求解 Advent of Code 谜题

[Advent of Code 2021](#) is coming! At JetBrains, we're proud to be supporting Advent of Code this year as one of its top sponsors. To get you in the mood for AoC 2021, we've prepared a [primer video](#) and a [blogpost](#). Get ready to jingle with Advent of Code in Kotlin! Have fun, learn new things, and win prizes!



[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.

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:

YouTube 视频: [Kotlin Tutorial: Advent of Code Puzzles, Day 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.

### 1.5.30 的新特性

- Read the puzzle description on [Advent of Code](#)
- Check out the solution from Svetlana Isakova on the [Kotlin Blog](#) or watch the video:

YouTube 视频: [Kotlin Tutorial: Advent of Code Puzzles, Day 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:

YouTube 视频: [Kotlin Tutorial: Adopting a Functional Style for Advent of Code Puzzles](#)

## 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:

YouTube 视频: [Kotlin Tutorial: Validating and Sanitizing Input. Advent of Code Puzzles](#)

## 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](#)

### 1.5.30 的新特性

- Check out the solution from Svetlana Isakova on the [Kotlin Blog](#) or watch the video:

YouTube 视频: [Kotlin Tutorial: Binary Representation of Numbers. Advent of Code Puzzles](#)

## 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:

YouTube 视频: [Idiomatic Kotlin: Operations with Sets](#)

## 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:

YouTube 视频: [Idiomatic Kotlin: Solving Advent of Code Puzzles and Traversing Trees](#)

## 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.

### 1.5.30 的新特性

- Read the puzzle description on [Advent of Code](#)
- Check out the solution from Sebastian Aigner on the [Kotlin Blog](#) or watch the video:

YouTube 视频: [Sealed Classes, Sequences, Immutability: Idiomatic Kotlin Solving Advent of Code Puzzles](#)

## Day 9: Encoding error

Explore different ways to manipulate lists in Kotlin using the `any()`, `firstOrNull()`, `firstNotNullOrOrNull()`, `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:

YouTube 视频: [Manipulating lists using windowed, scan, firstNotNullOfOrNull: Solving Advent of Code Puzzles](#)

## 下一步做什么？

- Complete more tasks with [Kotlin Koans](#)
- Create working applications with the free [Kotlin Basics track](#)

## 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](#).

## 1.5.30 的新特性

	<p><b>Atomic Kotlin</b> is for both beginning and experienced programmers!</p> <p>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!</p>
	<p><b>Head First Kotlin</b> 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.</p> <p>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.</p> <p>If you want to really understand Kotlin, this is the book for you.</p>
	<p><b>Kotlin in Action</b> 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.</p> <p>The book is written by Dmitry Jemerov and Svetlana Isakova, developers on the Kotlin team.</p> <p>Chapter 6, covering the Kotlin type system, and chapter 11, covering DSLs, are available as a free preview on the <a href="#">publisher web site</a>.</p>
	<p><b>Kotlin Programming: The Big Nerd Ranch Guide</b></p> <p>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.</p> <p>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.</p>

## 1.5.30 的新特性

	<p><a href="#">Programming Kotlin</a> is written by Venkat Subramaniam.</p> <p>Programmers don't just use Kotlin, they love it. Even Google has adopted it as a first-class language for Android development.</p> <p>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.</p> <p>Learn to use the many features of this highly concise, fluent, elegant, and expressive statically typed language with easy-to-understand examples.</p> <p>Learn to write maintainable, high-performing JVM and Android applications, create DSLs, program asynchronously, and much more.</p>
	<p><a href="#">The Joy of Kotlin</a> teaches you the right way to code in Kotlin.</p> <p>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.</p> <p>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.</p> <p>This book will change the way you code — and give you back some of the joy you had when you first started.</p>

## 在 IDE 中学习 (EduTools)

- 学习 Kotlin
- 讲授 Kotlin

## 通过 EduTools 插件学习 Kotlin

Android Studio 与 IntelliJ IDEA 都提供了 [EduTools 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 EduTools 插件进行教学，请阅读[通过 EduTools 插件讲授 Kotlin](#)。

## 通过 EduTools 插件讲授 Kotlin

使用 [Android Studio](#) 与 [IntelliJ IDEA](#) 中都可用的 [EduTools 插件](#)，可以通过代码练习任务来讲授 Kotlin。查看《[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。

如果要使用 EduTools 插件学习 Kotlin，请阅读 [通过 EduTools 插件学习 Kotlin](#)。

# 其他资源

- [FAQ](#)
- [早期访问计划 \(EAP\)](#)
  - 参与 Kotlin 早期访问计划
  - 安装 Kotlin EAP 插件
  - 配置构建采用 EAP
- [贡献力量](#)
- [Kotlin 演进](#)
  - 演进原则
  - Kotlin 各组件的稳定性
  - Kotlin 各组件的稳定性 (1.4 之前)
  - 兼容性
    - 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 ↗](#)
- [安全](#)
- [Kotlin 品牌资料](#)
  - [Kotlin 吉祥物](#)
  - [Kotlin 徽标](#)
  - [Kotlin 之夜品牌](#)
  - [KUG 品牌](#)
  - [宣传资料 ↗](#)

# FAQ

## Kotlin 是什么？

Kotlin 是一门面向 JVM、Android、JavaScript 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 1.6.10，发布于 2021-12-14。

## 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 support 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 还是 Android。随着原生 Kotlin (Kotlin/Native) 目前的进展，对其他平台（如嵌入式系统、macOS 和 iOS）的支持即将就绪。人们将 Kotlin 用于移动端和服务器端应用程序、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。除了用于后端 Web，你还可以使用 Kotlin/JS 用于客户端 Web。Kotlin 可以使用 [DefinitelyTyped](#) 中的定义来获取常见 JavaScript 库的静态类型版，并且它与现有的模块系统（如 AMD 和 CommonJS）兼容。更多信息请查看 [客户端开发中的资源](#)。

## 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 项目的一部分。它将 Kotlin 编译成无需虚拟机 (VM) 即可运行的原生代码。仍处于 beta 测试阶段，不过已经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？

### 1.5.30 的新特性

所有主要的 Java IDE 都支持 Kotlin，包括 [IntelliJ IDEA](#)、[Android Studio](#) 与 [Eclipse](#)。另外，有一个[命令行编译器](#)可用，为编译和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 17。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。

## 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。

## 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 主要由 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 [Roman Elizarov](#)。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、[在线尝试](#)以及访问相关资源。

## 有没有关于 Kotlin 的书？

There are a number of books available for Kotlin. Some of them we have reviewed and can recommend to start with. They are listed on the [Books](#) page. For more books, see the community-maintained list at [kotlin.link](#).

## Kotlin 有没有在线课程？

You can learn all the Kotlin essentials while creating working applications with the [Kotlin Basics track](#) on JetBrains Academy.

A few other courses you can take:

- [Pluralsight Course: Getting Started with Kotlin by Kevin Jones](#)
- [O'Reilly Course: Introduction to Kotlin Programming by Hadi Hariri](#)
- [Udemy Course: 10 Kotlin Tutorials for Beginneres by Peter Sommerhoff](#)

You can also check out the other tutorials and content on our [YouTube channel](#).

## 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。官方的年度 [KotlinConf](#) 由 JetBrains 主办。分别于 [2017 年](#) 在旧金山、[2018 年](#) 在阿姆斯特丹、[2019 年](#) 在哥本哈根举行。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](#)。

1.5.30 的新特性

## Kotlin 上社交媒体吗？

上。最活跃的 Kotlin 帐号是 [Twitter 上的](#)。

## 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。

## 早期访问计划 (EAP)

- 参与 Kotlin 早期访问计划
- 安装 Kotlin EAP 插件
- 配置构建采用 EAP

# 参与 Kotlin 早期访问计划

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 Milestone (*M*) builds before every feature (*1.x*) and incremental (*1.x.y*) 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.

## [Install the Kotlin EAP Plugin for IDEA or Android Studio](#)

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.



If you have already installed the EAP version and want to work on projects that were created previously, check [our instructions on how to configure your build to support this version](#).

## Build details

## 1.5.30 的新特性

Build info	Build highlights	Recommended kotlinx library versions
<b>1.6.20-RC2</b> Released: March 23, 2022 <a href="#">Release on GitHub</a>	<p>Stabilization of Kotlin 1.6.20-M1 features.</p> <p>For more details, please refer to the <a href="#">changelog</a>.</p>	Same as for Kotlin 1.6.20-M1, see below.
<b>1.6.20-RC</b> Released: March 1, 2022 <a href="#">Release on GitHub</a>	<p>Stabilization of Kotlin 1.6.20-M1 features.</p> <p>For more details, please refer to the <a href="#">changelog</a>.</p>	Same as for Kotlin 1.6.20-M1, see the row below.

## 1.5.30 的新特性

<p><b>1.6.20-M1</b> Released: <b>February 8, 2022</b></p> <p><a href="#">Release on GitHub</a></p>	<ul style="list-style-type: none"><li>• Language: prototype of context receivers</li><li>• Kotlin/JVM: experimental parallel compilation of a single module, deprecation of old <code>-xjvm-default</code> modes, new <code>@JvmDefaultWithCompatibility</code> annotation</li><li>• Kotlin/Native: concurrent sweep phase in the new memory manager, support for Xcode 13 SDKs, instantiation of annotation classes, support for resolving source locations with <code>libbacktrace</code>, improved "could not build module" cinterop report, performance improvements</li><li>• Multiplatform: hierarchical structure support enabled by default for all new multiplatform projects</li><li>• CocoaPods Gradle plugin: customization of podspec properties, accepting the podspec version when configuring CocoaPods for Kotlin/Native, new tasks for generating a suitable podspec for XCFramework builds</li><li>• Kotlin/JS: incremental compilation in the IR backend, lazy initialization of top-level properties by default, inline <code>Char</code> class, ability to use inline classes in external types, export improvements</li><li>• Gradle: new flexible way of defining a Kotlin compiler execution strategy, removed <code>kotlin.parallel.tasks.in.project</code> build option, deprecated <code>kapt.use.worker.api</code> and <code>kotlin.coroutines</code> build options</li></ul> <p>For more details, please refer to the <a href="#">changelog</a> or <a href="#">this blog post</a>.</p>	<ul style="list-style-type: none"><li>• <b>kotlinx.serialization</b> version: 1.3.2</li><li>• <b>kotlinx.coroutines</b> version: 1.6.0</li><li>• <b>kotlinx.atomicfu</b> version: 0.17.0</li><li>• <b>kotlinx-datetime</b> version: 0.3.2</li><li>• <b>ktor</b> version: 2.0.0-beta-1</li><li>• <b>kotlinx.html</b> version: 0.7.2</li><li>• <b>kotlinx-nodejs</b> version: 0.0.7</li></ul> <p>The versions of libraries from <code>kotlin-wrappers</code> (such as <code>kotlin-react</code>) can be found in the <a href="#">corresponding repository</a>.</p>
------------------------------------------------------------------------------------------------------------	----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

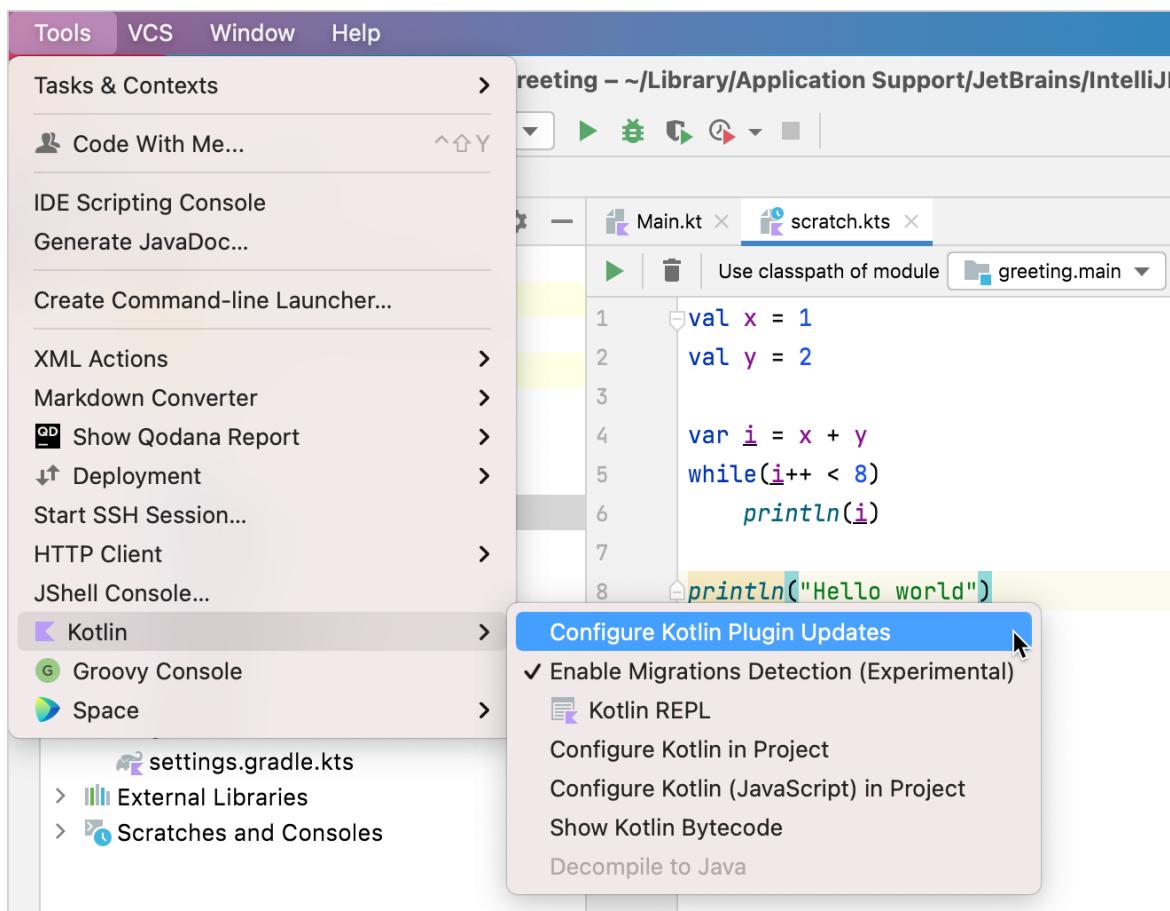
# 为 IntelliJ IDEA 或 Android Studio 安装 EAP 插件

Latest Kotlin EAP release: **1.6.20-RC2**

[Explore Kotlin EAP release details](#)

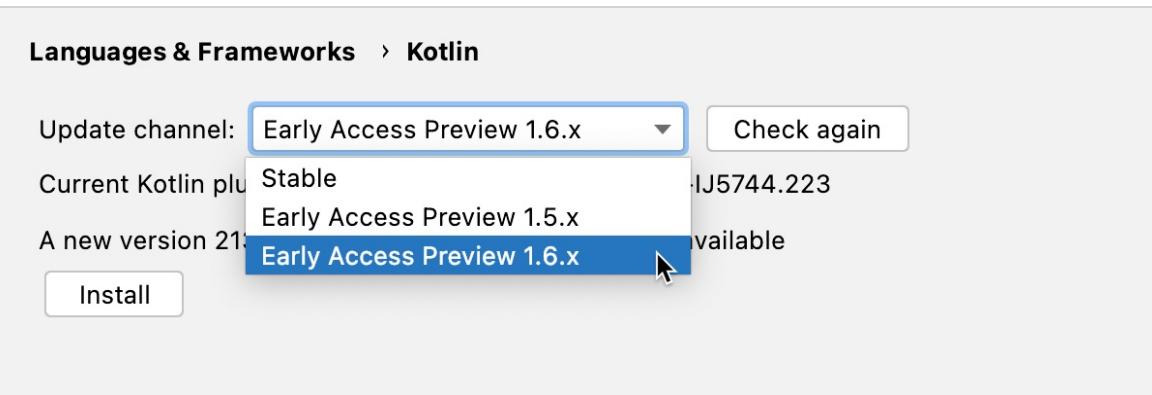
You can follow these instructions to install the preview version of the Kotlin Plugin for IntelliJ IDEA or Android Studio.

## 1. Select Tools | Kotlin | Configure Kotlin Plugin Updates.

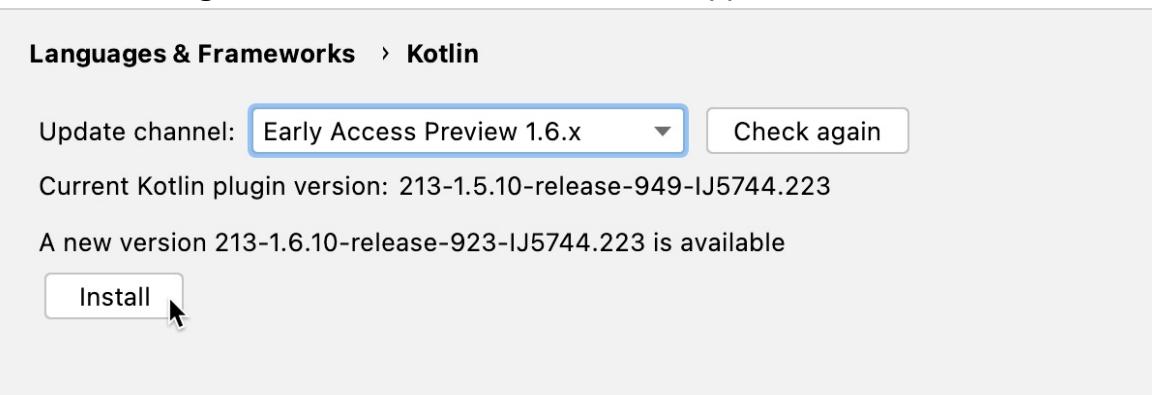


1. In the **Update channel** list, select the **Early Access Preview X** channel, where X is the latest version of Kotlin.

## 1.5.30 的新特性



1. Click **Check again**. The latest EAP build version appears.



1. Click **Install**.

If you want to work on existing projects that were created before installing the EAP version, you need to [configure your build for EAP](#).

## 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: in **Tools | Kotlin | Configure Kotlin Plugin Updates**, select the **Stable** update channel and click **Install**.

## 配置构建采用 EAP

If you create new projects using the EAP version of Kotlin, you don't need to perform any additional steps. The [Kotlin Plugin](#) will do everything for you!

You only need to configure your build manually for existing projects — projects that were created before installing the EAP version.

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 `1.6.20-RC2`. [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】

## 1.5.30 的新特性

```
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 – `1.6.0-RC3` – that is compatible with `1.6.20-RC2`.

### 【Kotlin】

## 1.5.30 的新特性

```
dependencies {
 implementation("org.jetbrains.kotlinx:kotlinx-coroutines-core:1.6.0-RC3")
}
```

### 【Groovy】

```
dependencies {
 implementation "org.jetbrains.kotlinx:kotlinx-coroutines-core:1.6.0-RC3"
}
```

## Configure in Maven

In the sample Maven project definition, replace `KOTLIN-EAP-VERSION` with the actual version, such as `1.6.20-RC2`. [Available EAP versions are listed here.](#)

## 1.5.30 的新特性

```
<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>
```

## 为 Kotlin 贡献力量

Kotlin 是一个开源项目，遵循 [Apache 2.0 License](#)。源代码、工具、文档，甚至这个网站都是在 [GitHub](#) 上维护的。虽然 Kotlin 主要由 JetBrains 开发，但有数百名外部贡献者参与 Kotlin 项目，我们一直在寻找更多的人来帮助我们。

## Participate in Early Access Preview

你可以通过 [参加 Kotlin 早期访问预览（EAP）](#) 来帮助我们改进 Kotlin，并提供宝贵的反馈。

每次发布，Kotlin 都会包含一些预览构建，让你可以在生产前尝试最新的功能。你可以在我们的问题追踪器 [YouTrack](#) 上报告任何发现的 bug，我们会在最终发布前尝试修复它们。这样，你就可以比标准的 Kotlin 发布周期更早地获得 bug 修复。

## Contribute to the compiler and standard library

如果你想为 Kotlin 编译器和标准库做出贡献，可以去 [JetBrains/Kotlin GitHub](#)，查看最新版本的 Kotlin，然后按照 [如何贡献](#) 的说明操作。

你可以通过完成 [开放任务](#) 来帮助我们。请保持与我们的沟通，因为我们可能会对你的更改提出问题和评论。否则，我们将无法纳入你的贡献。

## Contribute to the Kotlin IDE plugin

Kotlin IDE 插件是 IntelliJ IDEA 存储库的一部分。

要为 Kotlin IDE 插件做出贡献，请克隆 [IntelliJ IDEA 存储库](#) 并按照 [如何贡献](#) 的说明操作。

## 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](mailto: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.

## Create tutorials or videos

If you've created tutorials or videos for Kotlin, please share them with us via [feedback@kotlinlang.org](mailto:feedback@kotlinlang.org).

## Translate documentation to other languages

### 1.5.30 的新特性

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](https://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

### 1.5.30 的新特性

- 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

### 1.5.30 的新特性

- 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

## Kotlin 演进

- 演进原则
- Kotlin 各组件的稳定性
- Kotlin 各组件的稳定性 (1.4 之前)
- 兼容性
  - [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 演进

## 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是舒适的。

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。** 我们承认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，使其与用户需求保持一致、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的旧特性，并且完全成为历史特性。

**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”的确切含义可能不会有争议，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，仅通过反射可以观察到的行为更改，未记录/未定义的行为更改，重命名二进制文件等。有时这些更改非常重要，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，有时需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）来声明每个不兼容的更改，并提供自动迁移辅助工具。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

### 1.5.30 的新特性

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，它在社区的帮助下并根据 [Kotlin 基金会](#)来推动 kotlin 的发展。

[首席语言设计师](#)（现为 Roman Elizarov）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, William R. Cook 与 Roman Elizarov 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感到满意做出最终决定。为此，它依赖[此处](#)提供的一组准则。

## 特性发布与增量发布

类似 1.2、1.3 等版本的稳定版本通常被认为是对语言进行重大更改的特性版本。通常，在特性发布之间会发布增量发布，编号为 1.2.20、1.2.30 等。

增量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。稳定前特性可以随时被添加、删除或更改。

特性发布通常会添加新特性，并且可能会删除或更改以前不推荐使用的特性。某项特性从稳定前到稳定版的过渡也包含在特性版本的发布中。

## 早期预览版本

在发布稳定版本之前，我们通常会发布许多称为 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。特性版本的早期预览版本通常会生成二进制文件，这些二进制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。最终发布的二进制文件通常没有此限制。

## 稳定前特性

### 1.5.30 的新特性

根据上述反馈环原则，我们在语言的开放和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。

稳定前特性通常会在经过几次迭代后逐渐达到稳定状态。

## 不同组件的状态

要查看 Kotlin 的不同组件（Kotlin/JVM、JS、Native、各种库等）的稳定性状态，请查阅[链接](#)。

## Libraries

A language is nothing without its ecosystem, so we pay extra attention to enabling 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 ABI stability guarantees under the constraints of separate compilation. This is why every change in the language is examined from the point of view of binary compatibility.

On the other hand, a lot depends on the library authors being careful about which changes are safe to make. Thus it's very important 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.

### 1.5.30 的新特性

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 keys

Command line keys accepted by the compiler are also a kind of public API, and they are subject to the same considerations. Supported flags (those that don't have the "-X" or "-XX" prefix) can be added only in feature releases and should be properly deprecated before removing them. The "-X" and "-XX" flags 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 any more. 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 unmigrated code.

## Compatibility flags

We provide the `-language-version X.Y` and `-api-version X.Y` flags 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 development for three previous language and API versions in addition to the latest stable one.

Using an older `kotlin-stdlib` or `kotlin-reflect` with a newer compiler without specifying compatibility flags is not recommended, and the compiler will report a [warning](#) when this happens.

### 1.5.30 的新特性

Actively maintained code bases can benefit from getting bug fixes ASAP, without waiting for a full deprecation cycle to complete. Currently, such project can enable the `-progressive` flag and get such fixes enabled even in incremental releases.

All flags are available on the command line as well as [Gradle](#) and [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 very important 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, i.e. a newer compiler can read older binaries (e.g. 1.3 understands 1.0 through 1.2),
- Older compilers reject binaries that rely on new features (e.g. a 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 feature release, but not later ones (in the cases when new features are not used, e.g. 1.3 can understand most binaries from 1.4, but not 1.5).

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.

Please note that not all target platforms have reached this level of stability (but Kotlin/JVM has).

# 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 are evolved in the backward-compatible way following the [principles](#) of *Comfortable Updates* and *Keeping the Language Modern*. Among such stable components are, for example, the Kotlin compiler for the JVM, the Standard Library, and Coroutines.

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 very early stage, some are more mature. We mark them as **Experimental**, **Alpha** or **Beta** depending on how quickly each component is evolving and how much risk the users are taking when adopting it.

## Stability levels explained

Here's a quick guide to these stability levels and their meaning:

**Experimental** 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 decided to productize this idea, but it hasn't reached the 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.

**Stable** means "use it even in most conservative scenarios":

### 1.5.30 的新特性

- 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.

## 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 in explicitly to avoid accidental usages of features that have not been released as stable.

## Current stability of Kotlin components

### 1.5.30 的新特性

组件	状态	状态起始版本	备注
Kotlin/JVM	Stable	1.0	
kotlin-stdlib (JVM)	已稳定	1.0	
协程	已稳定	1.3	
kotlin-reflect (JVM)	Beta	1.0	
Kotlin/JS (旧版后端)	已稳定	1.3	
Kotlin/JVM (基于 IR)	已稳定	1.5	
Kotlin/JS (基于 IR)	Beta	1.5	
Kotlin/Native 运行时	Beta	1.3	
KLib 二进制	Alpha	1.4	
多平台项目	Alpha	1.3	
Kotlin/Native 与 C 语言及 Objective C 互操作	Beta	1.3	
CocoaPods 集成	Beta	1.3	
用于 Android Studio 的 Kotlin Multiplatform Mobile 插件	Alpha	0.3.0	与语言版本独立
expect/actual 语言特性	Beta	1.2	
KDoc 语法	已稳定	1.0	
Dokka	Beta	1.6	
脚本语法与语义	Alpha	1.2	
脚本嵌入与扩展 API	Beta	1.5	
脚本 IDE 支持	实验性的	1.2	
CLI 脚本	Alpha	1.2	

### 1.5.30 的新特性

组件	状态	状态起始版本	备注
编译器插件 API	实验性的	1.0	
序列化编译器插件	已稳定	1.4	
序列化核心库	已稳定	1.0.0	与语言版本独立
内联类	已稳定	1.5	
无符号算术	已稳定	1.5	
stdlib 中的契约	已稳定	1.3	
用户自定义契约	实验性的	1.3	
默认情况下，所有其他实验性特性	实验性的	N/A	

*The pre-1.4 version of this page is available [here](#).*

## Kotlin 各组件的稳定性 (1.4 之前)

There can be different modes of stability depending of how quickly a component is evolving:

- **Moving fast (MF)**: no compatibility should be expected between even [incremental releases](#), any functionality can be added, removed or changed without warning.
- **Additions in Incremental Releases (AIR)**: things can be added in an incremental release, removals and changes of behavior should be avoided and announced in a previous incremental release if necessary.
- **Stable Incremental Releases (SIR)**: incremental releases are fully compatible, only optimizations and bug fixes happen. Any changes can be made in a [feature release](#).
- **Fully Stable (FS)**: incremental releases are fully compatible, only optimizations and bug fixes happen. Feature releases are backwards compatible.

Source and binary compatibility may have different modes for the same component, e.g. the source language can reach full stability before the binary format stabilizes, or vice versa.

The provisions of the [Kotlin evolution policy](#) fully apply only to components that have reached Full Stability (FS). From that point on incompatible changes have to be approved by the Language Committee.

### 1.5.30 的新特性

组件	状态	其实版本	对于源代码	对于二进制
Kotlin/JVM	1.0	FS	FS	
kotlin-stdlib (JVM)	1.0	FS	FS	
KDoc 语法	1.0	FS	N/A	
协程	1.3	FS	FS	
kotlin-reflect (JVM)	1.0	SIR	SIR	
Kotlin/JS	1.1	AIR	MF	
Kotlin/Native	1.3	AIR	MF	
Kotlin 脚本 (*.kts)	1.2	AIR	MF	
dokka	0.1	MF	N/A	
Kotlin 脚本 API	1.2	MF	MF	
编译器插件 API	1.0	MF	MF	
序列化	1.3	MF	MF	
多平台项目	1.2	MF	MF	
内联类	1.3	MF	MF	
无符号算术	1.3	MF	MF	
默认情况下，所有其他实验性特性	N/A	MF	MF	

## 兼容性

- [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 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**

## 1.5.30 的新特性

**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

### 1.5.30 的新特性

**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

### 1.5.30 的新特性

**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

### 1.5.30 的新特性

**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

## Prohibit super calls from public-abi inline functions

### 1.5.30 的新特性

**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

### 1.5.30 的新特性

**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

### 1.5.30 的新特性

**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`

## 1.5.30 的新特性

**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`

### 1.5.30 的新特性

**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

### 1.5.30 的新特性

**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

### 1.5.30 的新特性

**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

## 1.5.30 的新特性

**Issue:** [KT-48587](#)

**Component:** kotlin-stdlib (JS)

**Incompatible change type:** source, binary

**Short summary:** the functions `HashMap.createEntrySet` and  
`AbstractMutableCollection.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

### 1.5.30 的新特性

**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

# 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

### 1.5.30 的新特性

**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

### 1.5.30 的新特性

**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

### 1.5.30 的新特性

**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

### 1.5.30 的新特性

**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

## 1.5.30 的新特性

**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

## 1.5.30 的新特性

**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
- $\geq 1.5$ : 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

### 1.5.30 的新特性

**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

### 1.5.30 的新特性

**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

### 1.5.30 的新特性

**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

### 1.5.30 的新特性

**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
- $\geq 1.6$ : 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

## 1.5.30 的新特性

**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-
}
```

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`

### 1.5.30 的新特性

**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

### 1.5.30 的新特性

**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

### 1.5.30 的新特性

**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

### 1.5.30 的新特性

**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

### 1.5.30 的新特性

**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

### 1.5.30 的新特性

**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

### 1.5.30 的新特性

**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

### 1.5.30 的新特性

**Issue:** [KT-37280](#)

**Component:** Core language

**Incompatible change type:** source

**Short summary:** since Kotlin 1.4, unchecked 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

### 1.5.30 的新特性

**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

### 1.5.30 的新特性

**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

### 1.5.30 的新特性

**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

### 1.5.30 的新特性

**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

### 1.5.30 的新特性

**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

### 1.5.30 的新特性

**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 val`s

## 1.5.30 的新特性

**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

### 1.5.30 的新特性

**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

### 1.5.30 的新特性

**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`

### 1.5.30 的新特性

**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

## 1.5.30 的新特性

**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

### 1.5.30 的新特性

**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 ( `đ` instead of `Đ` )

**Deprecation cycle:**

- < 1.4: digraphs are capitalized in the upper case ( `Đ` )
- >= 1.4: digraphs are capitalized in the title case ( `đ` )

## 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

### 1.5.30 的新特性

**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

### 1.5.30 的新特性

**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

### 1.5.30 的新特性

**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

### 1.5.30 的新特性

**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

### 1.5.30 的新特性

**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

### 1.5.30 的新特性

**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

### 1.5.30 的新特性

**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 properties are reported as deprecated (the compiler issues warnings)
- >=1.3: deprecation warnings elevated to errors

## mod operator convention

### 1.5.30 的新特性

**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

### 1.5.30 的新特性

**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

### 1.5.30 的新特性

**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 will 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

### 1.5.30 的新特性

**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

## 1.5.30 的新特性

**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-stdlib` 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, and 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 three previous language and API versions in addition to the latest stable one.

*Below, we use OV for "Older Version", and NV for "Newer Version".*

## Binary compatibility warnings

If you use the NV Kotlin compiler and have the OV standard library or the OV reflection library in the classpath, it can be a sign that the project is misconfigured. To prevent unexpected problems during compilation or at runtime, we suggest either updating the dependencies to NV, or specifying the API version / language version arguments explicitly. Otherwise, the compiler detects that something can go wrong and reports a warning.

For example, if OV = 1.0 and NV = 1.1, you can observe one of the following warnings:

- Runtime JAR files in the classpath have the version 1.0, which is older than 1.1. Consider using the runtime of version 1.1, or pass '`-api-version 1.0`' explicitly to make the compiler aware of the available APIs to the runtime of version 1.0.

This means that you're using the Kotlin compiler 1.1 against the standard or reflection library of version 1.0. This can be handled in different ways:

### 1.5.30 的新特性

- If you intend to use the APIs from the 1.1 Standard Library, or language features that depend on those APIs, you should upgrade the dependency to the version 1.1.
- If you want to keep your code compatible with the 1.0 standard library, you can pass `-api-version 1.0`.
- If you've just upgraded to Kotlin 1.1 but can not use new language features yet (e.g. because some of your teammates may not have upgraded), you can pass `-language-version 1.0`, which will restrict all APIs and language features to 1.0.

- Runtime JAR files in the classpath should have the same version. These files will be:  
`kotlin-reflect.jar` (version 1.0)  
`kotlin-stdlib.jar` (version 1.1)  
Consider providing an explicit dependency on `kotlin-reflect 1.1` to prevent subtle errors at runtime.  
Some runtime JAR files in the classpath have an incompatible version. Consider removing them from the classpath.

This means that you have a dependency on libraries of different versions, for example the 1.1 standard library and the 1.0 reflection library. To prevent subtle errors at runtime, we recommend you to use the same version of all Kotlin libraries. In this case, consider adding an explicit dependency on the 1.1 reflection library.

- Some JAR files in the classpath have the Kotlin Runtime library bundled into them. This may cause difficult to debug problems if there's a different version of the runtime library in the classpath. Consider removing these libraries from the classpath.

This means that there's a library in the classpath which does not depend on the Kotlin standard library as a Gradle/Maven dependency, but is distributed in the same artifact with it (i.e. has it *bundled*). Such a library may cause issues because standard build tools do not consider it an instance of the Kotlin standard library, thus it's not subject to the dependency version resolution mechanisms, and you can end up with several versions of the same library in the classpath. Consider contacting the authors of such a library and suggesting to use the Gradle/Maven dependency instead.

## 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](#)

1.5.30 的新特性

## Kotlin 基金会 FAQ

 [Kotlin 基金会 FAQ](#)

# 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](#).

# Kotlin 品牌资料

 [Kotlin 品牌资料](#)