Kotlin 语言文档

概述

使用 Kotlin 进行服务器端开发

Kotlin 非常适合开发服务器端应用程序，允许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，可以让你保持熟悉的技术栈，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 到 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。在线项目生成器允许用 Kotlin 快速生成一个新项目。
- Vert.x 是在 JVM 上构建响应式 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。
- 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。

Kotlin 用于服务器端的用户

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

- 使用 Http Servlet 创建 Web 应用程序及使用 Spring Boot 创建 RESTful Web 服务教程将向你展示如何在 Kotlin 中构建和运行非常小的 Web 应用程序。
- 关于更深入的介绍，请查看本站的参考文档及 Kotlin 心印。

使用 Kotlin 进行 Android 开发

Kotlin 非常适合开发 Android 应用程序，将现代语言的所有优势带入 Android 平台而不会引入任何新的限制：

- **兼容性**：Kotlin 与 JDK 6 完全兼容，保障了 Kotlin 应用程序可以在较旧的 Android 设备上运行而无任何问题。Kotlin 工具在 Android Studio 中会完全支持，并且兼容 Android 构建系统。
- **性能**：由于非常相似的字节码结构，Kotlin 应用程序的运行速度与 Java 类似。随着 Kotlin 对内联函数的支持，使用 lambda 表达式的代码通常比用 Java 写的代码运行得更快。
- **互操作性**：Kotlin 可与 Java 进行 100% 的互操作，允许在 Kotlin 应用程序中使用所有现有的 Android 库 。这包括注解处理，所以数据绑定与 Dagger 也是一样。
- **占用**：Kotlin 具有非常紧凑的运行时库，可以通过使用 ProGuard 进一步减少。在实际应用程序中，Kotlin 运行时只增加几百个方法以及 .apk 文件不到 100K 大小。
- **编译时长**：Kotlin 支持高效的增量编译，所以对于清理构建会有额外的开销，增量构建通常与 Java 一样快或者更快。
- **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 到 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。

Kotlin 用于 Android 的案例学习

Kotlin 已被一些大公司成功采用，其中一些公司分享了他们的经验：

- Pinterest 已经成功地将 Kotlin 引入了他们的应用程序中，每个月有 1 亿 5 千万人使用。
- Basecamp 的 Android 应用程序是 100% Kotlin 代码，他们报告了程序员幸福的巨大差异，以及工作质量与速度的巨大改善。
- Keepsafe 的 App Lock 应用程序也已转换为 100% Kotlin，使源代码行数减少 30%、方法数减少 10%。

用于 Android 开发的工具

Kotlin 团队为 Android 开发提供了一套超越标准语言功能的工具：

- Kotlin Android 扩展是一个编译器扩展，可以让你摆脱代码中的 `findViewById()` 调用，并将其替换为合成的编译器生成的属性。
- Anko 是一个提供围绕 Android API 的 Kotlin 友好的包装器的库 ，以及一个可以用 Kotlin 代码替换布局 .xml 文件的 DSL。

下一步

- 下载并安装 Android Studio 3.0，其中包含开箱即用的 Kotlin 支持。
- 按照 Android 与 Kotlin 入门教程创建你的第一个 Kotlin 应用程序。
- 关于更深入的介绍，请查看本站的参考文档及 Kotlin 心印。

另一个很好的资源是 Kotlin for Android Developers，这本书会引导你逐步完成在 Kotlin 中创建真正的 Android 应用程序的过程。

- 检出 Google 的 Kotlin 写的示例项目。

Kotlin JavaScript 概述

Kotlin 提供了 JavaScript 作为目标平台的能力。它通过将 Kotlin 转换为 JavaScript 来实现。目前的实现目标是 ECMAScript 5.1，但也有最终目标为 ECMAScript 2015 的计划。

当你选择 JavaScript 目标时，作为项目一部分的任何 Kotlin 代码以及 Kotlin 附带的标准库都会转换为 JavaScript。然而，这不包括使用的 JDK 和任何 JVM 或 Java 框架或库。任何不是 Kotlin 的文件会在编译期间忽略掉。

Kotlin 编译器努力遵循以下目标：

- 提供最佳大小的输出
- 提供可读的 JavaScript 输出
- 提供与现有模块系统的互操作性
- 在标准库中提供相同的功能，无论是 JavaScript 还是 JVM 目标（尽最大可能程度）。

如何使用

你可能希望在以下情景中将 Kotlin 编译为 JavaScript：

- 创建面向客户端 JavaScript 的 Kotlin 代码
  - **与 DOM 元素交互**。Kotlin 提供了一系列静态类型的接口来与文档对象模型（Document Object Model）交互，允许创建和更新 DOM 元素。
  - **与图形如 WebGL 交互**。你可以使用 Kotlin 在网页上用 WebGL 创建图形元素。
- 创建面向服务器端 JavaScript 的 Kotlin 代码
  - **使用服务器端技术**。你可以使用 Kotlin 与服务器端 JavaScript（如 Node.js）进行交互

Kotlin 可以与现有的第三方库和框架（如 jQuery 或 ReactJS）一起使用。要使用强类型 API 访问第三方框架，可以使用 ts2kt 工具将 TypeScript 定义从 Definitely Typed 类型定义仓库转换为 Kotlin。或者，你可以使用动态类型访问任何框架，而无需强类型。

JetBrains 特地为 React 社区开发并维护了几个工具：React bindings 以及 Create React Kotlin App。后者可以帮你开始使用 Kotlin 构建 React 应用程序而无需构建配置。

Kotlin 兼容 CommonJS、AMD 和 UMD，直截了当与不同的模块系统交互。

Kotlin 转 JavaScript 入门

要了解如何开始使用 JavaScript 平台的 Kotlin，请参考其教程。

Kotlin/Native 用于原生开发

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。它是一个基于 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。它解决了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：

- iOS（arm32、arm64、模拟器 x86_64）
- MacOS（x86_64）
- Android（arm32、arm64）
- Windows（mingw x86_64）
- Linux（x86_64、arm32、MIPS、MIPS 小端次序）
- WebAssembly（wasm32）

互操作

Kotlin/Native 支持与原生世界的双向互操作。一方面，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

另一方面，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

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

不同目标平台的 Kotlin 与 Kotlin/Native 之间支持多平台项目。这是在多个平台之间共享公共 Kotlin 代码的方式，这些平台包括 Android、iOS、服务器端、JVM、客户端、JavaScript、CSS 以及原生平台。

多平台库为公共 Kotlin 代码提供了必要的 API，并有助于在 Kotlin 代码中一次性开发项目的共享部分，从而将其与所有目标平台共享。

如何开始

教程与文档

Kotlin 新手?可以看看入门页。

建议的文档页：

- C 语言互操作
- Swift/Objective-C 互操作

推荐的教程：

- 基本的 Kotlin/Native 应用程序
- 多平台项目：iOS 与 Android
- 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

示例项目

- Kotlin/Native 源代码与示例
- KotlinConf app
- KotlinConf Spinner app
- Kotlin/Native 源代码与示例（.tgz）
- Kotlin/Native 源代码与示例（.zip）

在 GitHub 上还有更多示例。

用于异步编程等场景的协程

异步或非阻塞程序设计是新的现实。无论我们创建服务端应用、桌面应用还是移动端应用，都很重要的一点是，我们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）。

这个问题有很多方法，在 Kotlin 中我们采用非常灵活的方法，在语言级提供协程支持，而将大部分功能委托给库，这与 Kotlin 的理念非常一致。

额外收益是，协程不仅打开了异步编程的大门，还提供了大量其他的可能性，例如并发、参与者（actor）等。

如何开始

Tutorials and Documentation

Kotlin 新手?可以看看入门页。

精选文档页：

- 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

推荐的教程：

- 你的第一个 Kotlin 协程程序
- 异步程序设计

示例项目

- kotlinx.coroutines 示例与源代码
- KotlinConf app

在 GitHub 上还有更多示例

多平台程序设计

> 多平台项目是 Kotlin 1.2 与 1.3 中的实验性特性。本文档中描述的所有的语言与工具特性在未来的版本中都可能会发生变化。

在所有平台上都能用是 Kotlin 的一个明确目标，但我们将其视为一个更重要的目标——在多个平台之间共享代码的前提。有了对 JVM、Android、JavaScript、iOS、Linux、Windows、 Mac 甚至像 STM32 这样的嵌入式系统的支持，Kotlin 可以处理现代应用程序的任何组件与所有组件。这为代码与专业知识的复用带来了宝贵的收益，节省了工作量去完成更具挑战任务，而不是将所有东西都实现两次或多次。

它是如何工作的

总得来说，多平台并不是为所有平台编译全部代码。这个模型有其明显的局限性，我们知道现代应用程序需要访问其所运行平台的独有特性。Kotlin 并不会限制你只使用其中所有 API 的公共子集。每个组件都可以根据需要与其他组件共享尽可能多的代码，而通过语言所提供的 `expect/actual` 机制可以随时访问平台 API。

以下是在极简版日志框架中公共逻辑与平台逻辑之间代码共享与交互的示例。其公共代码如下所示：

```
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)
```

├ 为所有平台编译

├ 预期的平台相关 API

├ 可在公共代码中使用预期的 API

它期待目标平台为 `writeLogMessage` 提供平台相关实现，然后公共代码就可以使用此声明而无需考虑它是如何实现的。

在 JVM 上，可以提供一个将日志写到标准输出的实现：

```
internal actual fun writeLogMessage(message: String, logLevel: LogLevel) {
    println("[$logLevel]: $message")
}
```

在 JavaScript 世界中可用的是一组完全不同的 API，因此可以实现为将日志记录到控制台：

```
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)
    }
}
```

在 1.3 中我们重新设计了整个多平台模型。我们用于描述多平台 Gradle 项目的新版 DSL 更加灵活，我们会继续努力使项目配置更加简单。

多平台库

公共代码可以依赖于一组涵盖日常任务的库，例如 HTTP、serialization 以及协程管理。此外，丰富的标准库在所有平台上都可用。

你可以随时编写自己的库，提供一个公共的 API，而在每个平台上以不同的方式实现。

使用场景

Android——iOS

移动平台之间共享代码是 Kotlin 多平台的主要使用场景之一，现在可以通过在 Android 与 iOS 之间共享部分代码（如业务逻辑、连接等）来构建移动应用。

参见：多平台项目：iOS 与 Android

客户端——服务端

代码共享可以带来收益的另一个场景是互联应用，其中的逻辑可以在服务器与运行在浏览器中的客户端中复用。Kotlin 多平台也覆盖了这个场景。

Ktor 框架适用于在互联系统中构建异步的服务器与客户端。

如何开始

Tutorials and Documentation

Kotlin 新手?可以看看[入门]Getting Started页。

建议的文档页：

— 搭建一个多平台项目

— 平台相关声明

推荐的教程：

— 多平台 Kotlin 库

— 多平台项目：iOS 与 Android

示例项目

— KotlinConf app

— KotlinConf Spinner app

在 GitHub 上还有更多示例

Kotlin 1.1 的新特性

目录

JavaScript

从 Kotlin 1.1 开始，JavaScript 目标平台不再当是实验性的。所有语言功能都支持，并且有许多新的工具用于与前端开发环境集成。更详细改动列表，请参见下文 。

协程（实验性的）

Kotlin 1.1 的关键新特性是 *协程*，它带来了 `future` / `await`、`yield` 以及类似的编程模式的支持。Kotlin 的设计中的关键特性是协程执行的实现是语言库的一部分，而不是语言的一部分，所以你不必绑定任何特定的编程范式或并发库。

协程实际上是一个轻量级的线程，可以挂起并稍后恢复。协程通过 *挂起函数* 支持：对这样的函数的调用可能会挂起协程，并启动一个新的协程，我们通常使用匿名挂起函数（即挂起 lambda 表达式）。

我们来看看在外部库 kotlinx.coroutines 中实现的 `async` / `await`：

```
// 在后台线程池中运行该代码
fun asyncOverlay() = async(CommonPool) {
    // 启动两个异步操作
    val original = asyncLoadImage("original")
    val overlay = asyncLoadImage("overlay")
    // 然后应用叠加到两个结果
    applyOverlay(original.await(), overlay.await())
}

// 在 UI 上下文中启动新的协程
launch(UI) {
    // 等待异步叠加完成
    val image = asyncOverlay().await()
    // 然后在 UI 中显示
    showImage(image)
}
```

这里，`async { …… }` 启动一个协程，当我们使用 `await()` 时，挂起协程的执行，而执行正在等待的操作，并且在等待的操作完成时恢复（可能在不同的线程上）。

标准库通过 `yield` 和 `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：

```
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
intln(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。

其他语言功能

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：

```
typealias OscarWinners = Map<String, String>

fun countLaLaLand(oscarWinners: OscarWinners) =
        oscarWinners.count { it.value.contains("La La Land") }

// 请注意，类型名称（初始名和类型别名）是可互换的：
fun checkLaLaLandIsTheBestMovie(oscarWinners: Map<String, String>) =
        oscarWinners["Best picture"] == "La La Land"
```

更详细信息请参阅其 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：

```
val numberRegex = "\\d+".toRegex()
val numbers = listOf("abc", "123", "456").filter(numberRegex::matches)
```

更详细信息请参阅其 KEEP。

密封类和数据类

Kotlin 1.1 删除了一些对 Kotlin 1.0 中已存在的密封类和数据类的限制。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：

```
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)))
```

更详细信息请参阅其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。这里有一个例子：

```
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!" })
```

更详细信息请参阅其文档及其 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：

```
map.forEach { _, value -> println("$value!") }
```

这也适用于解构声明：

```
val (_, status) = getResult()
```

更详细信息请参阅其 KEEP。

数字字面值中的下划线

正如在 Java 8 中一样，Kotlin 现在允许在数字字面值中使用下划线来分隔数字分组：

```
val oneMillion = 1_000_000
val hexBytes = 0xFF_EC_DE_5E
val bytes = 0b11010010_01101001_10010100_10010010
```

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：

```
data class Person(val name: String, val age: Int) {
val isAdult get() = age >= 20 // 属性类型推断为 “Boolean”
```

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。

```
public val <T> List<T>.lastIndex: Int
    inline get() = this.size - 1
```

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。

更详细信息请参阅其文档及其 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：

```
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...")
}
```

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查属性名称，我们可以这样写：

```
class ResourceLoader<T>(id: ResourceID<T>) {
    operator fun provideDelegate(thisRef: MyUI, property: KProperty<*>):
ReadOnlyProperty<MyUI, T> {
        checkProperty(thisRef, property.name)
        …… // 属性创建
    }

    private fun checkProperty(thisRef: MyUI, name: String) { …… }
}

fun <T> bindResource(id: ResourceID<T>): ResourceLoader<T> { …… }

class MyUI {
    val image by bindResource(ResourceID.image_id)
    val text by bindResource(ResourceID.text_id)
}
```

`provideDelegate` 方法在创建 `MyUI` 实例期间将会为每个属性调用，并且可以立即执行必要的验证。

更详细信息请参阅其文档。

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：

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

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

对于 DSL 中隐式接收者的作用域控制

`@DslMarker` 注解允许限制来自 DSL 上下文中的外部作用域的接收者的使用。考虑那个典型的 HTML 构建器示例：

```
table {
    tr {
        td { + "Text" }
    }
}
```

在 Kotlin 1.0 中，传递给 `td` 的 lambda 表达式中的代码可以访问三个隐式接收者：传递给 `table`、`tr` 和 `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。

更详细信息请参阅其文档及其 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、但对于集合和序列很有用的扩展函数，它允许对操作链中的集合/序列的每个元素执行一些操作，可能带有副作用。对于迭代其行为像 `forEach` 但是也进一步返回可迭代实例。对于序列它返回一个包装序列，它在元素迭代时延迟应用给定的动作。

```
inputDir.walk()
        .filter { it.isFile && it.name.endsWith(".txt") }
        .onEach { println("Moving $it to $outputDir") }
        .forEach { moveFile(it, File(outputDir, it.toRelativeString(inputDir))) }
```

also()、takeIf() 和 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：

```
fun Block.copy() = Block().also {
    it.content = this.content
}
```

`takeIf` 就像单个值的 `filter`。它检查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-操作符和及早返回，它允许编写如下结构：

```
val outDirFile = File(outputDir.path).takeIf { it.exists() } ?: return false
// 对现有的 outDirFile 做些事情
```

```
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
```

`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：

```
val result = string.takeUnless(String::isEmpty)
```

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：

```
val frequencies = words.groupingBy { it.first() }.eachCount()
```

Map.toMap() 和 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、键的序列和键的数组。

```
val map = mapOf("key" to 42)
val emptyMap = map - "key"
```

minOf() 和 maxOf()

这些函数可用于查找两个或三个给定值中的最小和最大值，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。

```
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 })
```

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` 和 `MutableList` 实例的函数，并通过调用 lambda 表达式来初始化每个元素：

```
val squares = List(10) { index -> index * index }
val mutable = MutableList(10) { 0 }
```

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。

```
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
```

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` 和 `AbstractMap`，而对于可变集合，有 `AbstractMutableCollection`、`AbstractMutableList`、`AbstractMutableSet` 和 `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` 和 `contentDeepEquals`），哈希码计算（`contentHashCode` 和 `contentDeepHashCode`），以及转换成一个字符串（`contentToString` 和 `contentDeepToString`）。它们都支持 JVM（它们作为 `java.util.Arrays` 中的相应函数的别名）和 JS（在 Kotlin 标准库中提供实现）。

```
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
```

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，接口和 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 和 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` 和 `kotlin-stdlib-jre8` maven 构件，而不是标准的 `kotlin-stdlib`。这些构件是在 `kotlin-stdlib` 之上的微小扩展，它们将它作为传递依赖项带到项目中。

字节码中的参数名

Kotlin 现在支持在字节码中存储参数名。这可以使用命令行选项 `-java-parameters` 启用。

常量内联

编译器现在将 `const val` 属性的值内联到使用它们的位置。

可变闭包变量

用于在 lambda 表达式中捕获可变闭包变量的装箱类不再具有 volatile 字段。此更改提高了性能，但在一些罕见的使用情况下可能导致新的竞争条件。如果受此影响，你需要提供自己的同步机制来访问变量。

javax.scripting 支持

Kotlin 现在与javax.script API（JSR-223）集成。其 API 允许在运行时求值代码段：

```
val engine = ScriptEngineManager().getEngineByExtension("kts")!!
engine.eval("val x = 3")
println(engine.eval("x + 2"))  // 输出 5
```

关于使用 API 的示例项目参见这里 。

kotlin.reflect.full

为 Java 9 支持准备，在 `kotlin-reflect.jar` 库中的扩展函数和属性已移动到 `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查的代码，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 平台允许对类和属性使用 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)
    }
}
```

Kotlin 1.2 的新特性

目录

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。

— 常规模块针对指定的平台，既可以是平台模块的依赖，也可以依赖平台模块。

当你为指定平台编译多平台项目时，既会生成公共代码也会生成平台相关代码。

多平台项目支持的一个主要特点是可以通过**预期声明与实际声明**来表达公共代码对平台相关部分的依赖关系。一个预期声明指定一个 API（类、接口、注解、顶层声明等）。一个实际声明要么是该 API 的平台相关实现，要么是一个引用到在一个外部库中该 API 的一个既有实现的别名。这是一个示例：

在公共代码中：

```
// 预期平台相关 API:
expect fun hello(world: String): String

fun greet() {
    // 该预期 API 的用法:
    val greeting = hello("multi-platform 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，请参见其documentation。

其他语言特性

注解中的数组字面值

自 Kotlin 1.2 起，注解的数组参数可以通过新的数组字面值语法传入，而无需使用 `arrayOf` 函数：

```
@CacheConfig(cacheNames = ["books", "default"])
public class BookRepositoryImpl {
    // ……
}
```

该数组字面值语法仅限于注解参数。

lateinit 顶层属性与局部变量

`lateinit` 修饰符现在可以用于顶层属性与局部变量了。例如，后者可用于当一个 lambda 表达式作为构造函数参数传给一个对象时，引用另一个必须稍后定义的对象：

```
class Node<T>(val value: T, val next: () -> Node<T>)

fun main(args: Array<String>) {
    // 三个节点的环：
    lateinit var third: Node<Int>

    val second = Node(2, next = { third })
    val first = Node(1, next = { second })

    third = Node(3, next = { first })

    val nodes = generateSequence(first) { it.next() }
    println("Values in the cycle: ${nodes.take(7).joinToString { it.value.toString() }},
...")
}
```

检查 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：

```
    println("isInitialized before assignment: " + this::lateinitVar.isInitialized)
    lateinitVar = "value"
    println("isInitialized after assignment: " + this::lateinitVar.isInitialized)
```

内联函数带有默认函数式参数

内联函数现在允许其内联函式数参数具有默认值：

```
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)" }
```

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：

```
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<*>
```

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：

```
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
    }
}
```

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。

阻断性变更：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 单个命名参数

为了与注解中的数组字面值保持一致，向一个命名参数形式的 vararg 参数传入单个项目的用法（`foo(items = i)`）已被弃用。请使用伸展操作符连同相应的数组工厂函数：

```
foo(items = *intArrayOf(1))
```

在这种情况下有一项防止性能下降的优化可以消除冗余的数组创建。单参数形式在 Kotlin 1.2 中会产生警告，而在 Kotlin 1.3中会放弃。

弃用：扩展 Throwable 的泛型类的内部类

继承自 `Throwable` 的泛型类的内部类可能会在 throw-catch 场景中违反类型安全性，因此已弃用，在 Kotlin 1.2 中会是警告，而在 Kotlin 1.3中会是错误。

弃用：修改只读属性的幕后字段

通过在自定义 getter 中赋值 `field = ……` 来修改只读属性的幕后字段的用法已被弃用，在 Kotlin 1.2 中会是警告，而在 Kotlin 1.3中会是错误。

标准库

Kotlin 标准库构件与拆分包

Kotlin 标准库现在完全兼容 Java 9 的模块系统，它禁止拆分包（多个 jar 文件声明的类在同一包中）。为了支持这点，我们引入了新的 `kotlin-stdlib-jdk7` 与 `kotlin-stdlib-jdk8`，它们取代了旧版的 `kotlin-stdlib-jre7` 与 `kotlin-stdlib-jre8`。

在 Kotlin 看来新的构件中的声明在相同的包名内，而在 Java看来有不同的包名。因此，切换到新的构件无需修改任何源代码。

确保与新的模块系统兼容的另一处变更是在 `kotlin-reflect` 库中删除了 `kotlin.reflect` 包中弃用的声明。如果你正在使用它们，你需要切换到使用 `kotlin.reflect.full` 包中的声明，自 Kotlin 1.1 起就支持这个包了。

windowed、chunked、zipWithNext

用于 `Iterable<T>`、`Sequence<T>` 与 `CharSequence` 的新的扩展覆盖了这些应用场景：缓存或批处理（`chunked`）、滑动窗口与计算滑动均值（`windowed`）以及处理成对的后续条目（`zipWithNext`）：

```
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 }
```

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`：

```
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")
```

kotlin-stdlib 中的数学运算

为满足由来已久的需求，Kotlin 1.2 添加了 JVM 与 JS 公用的用于数学运算的 `kotlin.math` API，包含以下内容：

- 常量：`PI` 与 `E`；
- 三角函数：`cos`、`sin`、`tan` 及其反函数：`acos`、`asin`、`atan`、`atan2`；
- 双曲函数：`cosh`、`sinh`、`tanh` 及其反函数：`acosh`、`asinh`、`atanh`
- 指数函数：`pow`（扩展函数）、`sqrt`、`hypot`、`exp`、`expm1`；
- 对数函数：`log`、`log2`、`log10`、`ln`、`ln1p`；
- 取整：

  - `ceil`、`floor`、`truncate`、`round`（奇进偶舍）函数；
  - `roundToInt`、`roundToLong`（四舍五入）扩展函数；
- 符号与绝对值：
  - `abs` 与 `sign` 函数；
  - `absoluteValue` 与 `sign` 扩展属性；
  - `withSign` 扩展函数；
- 两个数的最值函数：`max` 与 `min`；
- 二进制表示：
  - `ulp` 扩展属性；
  - `nextUp`、`nextDown`、`nextTowards` 扩展函数；
  - `toBits`、`toRawBits`、`Double.fromBits`（这些在 `kotlin` 包中）。

这些函数同样也有 `Float` 参数版本（但不包括常量）。

用于 BigInteger 与 BigDecimal 的操作符与转换

Kotlin 1.2 引入了一些使用 `BigInteger` 与 `BigDecimal` 运算以及由其他数字类型创建它们的函数。具体如下：

- `toBigInteger` 用于 `Int` 与 `Long`；
- `toBigDecimal` 用于 `Int`、`Long`、`Float`、`Double` 以及 `BigInteger`；
- 算术与位运算操作符函数：
  - 二元操作符 `+`、`-`、`*`、`/`、`%` 以及中缀函数 `and`、`or`、`xor`、`shl`、`shr`；
  - 一元操作符 `-`、`++`、`--` 以及函数 `inv`。

浮点数到比特的转换

添加了用于将 `Double` 及 `Float` 与其比特表示形式相互转换的函数：

- `toBits` 与 `toRawBits` 对于 `Double` 返回 `Long` 而对于 `Float` 返回 `Int`；
- `Double.fromBits` 与 `Float.fromBits` 用于有相应比特表示形式创建浮点数。

正则表达式现在可序列化

`kotlin.text.Regex` 类现在已经是 `Serializable` 的了并且可用在可序列化的继承结构中。

如果可用，Closeable.use 会调用 Throwable.addSuppressed

当在其他异常之后关闭资源期间抛出一个异常，`Closeable.use` 函数会调用 `Throwable.addSuppressed` 。

要启用这个行为，需要依赖项中有 `kotlin-stdlib-jdk7` 。

JVM 后端

构造函数调用规范化

自 1.0 版起，Kotlin 就已支持带有复杂控制流的表达式，诸如 try-catch 表达式以及内联函数。根据 Java 虚拟机规范这样的代码是有效的。不幸的是，当这样的表达式出现在构造函数调用的参数中时，一些字节码处理工具不能很好地处理这种代码 。

为了缓解这种字节码处理工具用户的这一问题，我们添加了一个命令行选项（`-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，针对 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'`（“针对 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针对 JVM 1.8 编译。

阻断性变更：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。

阻断性变更：修正平台 null 透过内联扩展接收者逃逸

在平台类型的空值上调用内联扩展函数并没有检测接收者是否为 null，因而允许 null 逃逸到其他代码中。Kotlin 1.2 在调用处强制执行这项检测，如果接收者为空就抛出异常。

要切换到旧版行为，请将回退标志 `-Xno-receiver-assertions` 传给编译器。

JavaScript 后端

默认启用 TypedArrays 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 1.3 的新特性

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 之间共享的源集）。
- 现在支持发布多平台库了。

更多相关信息，请参考多平台程序设计文档。

契约

Kotlin 编译器会做大量的静态分析工作，以提供警告并减少模版代码。其中最显著的特性之一就是智能转换——能够根据类型检测自动转换类型。

```
fun foo(s: String?) {
    if (s != null) s.length // 编译器自动将“s”转换为“String”
}
```

然而，一旦将这些检测提取到单独的函数中，所有智能转换都立即消失了：

```
fun String?.isNotNull(): Boolean = this != null

fun foo(s: String?) {
    if (s.isNotNull()) s.length // 没有智能转换 :(
}
```

为了改善在此类场景中的行为，Kotlin 1.3 引入了称为 *契约* 的实验性机制。

*契约*让一个函数能够以编译器理解的方式显式描述其行为。目前支持两大类场景：

— 通过声明函数调用的结果与所传参数值之间的关系来改进智能转换分析：

```
fun require(condition: Boolean) {
    // 这是一种语法格式，告诉编译器：
    // “如果这个函数成功返回，那么传入的‘condition’为 true”
    contract { returns() implies condition }
    if (!condition) throw IllegalArgumentException(...)
}

fun foo(s: String?) {
    require(s is String)
    // s 在这里智能转换为“String”，因为否则
    // “require”会抛出异常
}
```

— 在存在高阶函数的情况下改进变量初始化的分析：

```
fun synchronize(lock: Any?, block: () -> Unit) {
    // 告诉编译器：
    // “这个函数会在此时此处调用‘block’，并且刚好只调用一次”
    contract { callsInPlace(block, EXACTLY_ONCE) }
}

fun foo() {
    val x: Int
    synchronize(lock) {
        x = 42 // 编译器知道传给“synchronize”的 lambda 表达式刚好
               // 只调了一次，因此不会报重复赋值错
    }
    println(x) // 编译器知道一定会调用该 lambda 表达式而执行
               // 初始化操作，因此可以认为“x”在这里已初始化
}
```

标准库中的契约

`stdlib`（kotlin 标准库）已经利用契约带来了如上所述的对代码分析的改进。这部分契约是**稳定版的**，这意味着你现在就可以从改进的代码分析中受益，而无需任何额外的 opt-ins：

```
fun bar(x: String?) {
    if (!x.isNullOrEmpty()) {
        println("length of '$x' is ${x.length}") // 哇，已经智能转换为非空！
    }
}
```

自定义契约

可以为自己的函数声明契约，不过这个特性是**实验性的**，因为目前的语法尚处于早期原型状态，并且很可能还会更改。另外还要注意，目前 Kotlin 编译器并不会验证契约，因此程序员有责任编写正确合理的契约。

通过调用标准库（stdlib）函数 `contract` 来引入自定义契约，该函数提供了 DSL 作用域：

```
fun String?.isNullOrEmpty(): Boolean {
    contract {
        returns(false) implies (this@isNullOrEmpty != null)
    }
    return this == null || isEmpty()
}
```

请参见 KEEP 中关于语法与兼容性注意事项的详细信息。

将 when 主语捕获到变量中

在 Kotlin 1.3 中，可以将 `when` 表达式主语捕获到变量中：

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

虽然已经可以在 `when` 表达式前面提取这个变量，但是在 `when` 中的 `val` 使其作用域刚好限制在 `when` 主体中，从而防止命名空间污染。关于 `when` 表达式的完整文档请参见这里。

接口中伴生对象的 @JvmStatic 与 @JvmField

对于 Kotlin 1.3，可以使用注解 `@JvmStatic` 与 `@JvmField` 标记接口的 `companion` 对象成员。在类文件中会将这些成员提升到相应接口中并标记为 `static`。

例如，以下 Kotlin 代码：

```
interface Foo {
    companion object {
        @JvmField
        val answer: Int = 42

        @JvmStatic
        fun sayHello() {
            println("Hello, world!")
        }
    }
}
```

相当于这段 Java 代码：

```
interface Foo {
    public static int answer = 42;
    public static void sayHello() {
        // ……
    }
}
```

注解类中的内嵌声明

在 Kotlin 1.3 中，注解可以有内嵌的类、接口、对象与伴生对象：

```
annotation class Foo {
    enum class Direction { UP, DOWN, LEFT, RIGHT }

    annotation class Bar

    companion object {
        fun foo(): Int = 42
        val bar: Int = 42
    }
}
```

无参的 `main`

按照惯例，Kotlin 程序的入口点是一个签名类似于 `main(args: Array<String>)` 的函数，其中 `args` 表示传给该程序的命令行参数。然而，并非每个应用程序都支持命令行参数，因此这个参数往往到最后都没有用到。

Kotlin 1.3 引入了一种更简单的无参 `main` 形式。现在 Kotlin 版的 `Hello,World` 缩短了19个字符！

```
fun main() {
    println("Hello, world!")
}
```

更多元的函数

在 Kotlin 中用带有不同数量参数的泛型类来表示函数类型：`Function0<R>`、`Function1<P0, R>`、`Function2<P0, P1, R>`……这种方式有一个问题是这个列表是有限的，目前只到 `Function22`。

Kotlin 1.3 放宽了这一限制，并添加了对具有更多元数（参数个数）的函数的支持：

```
fun trueEnterpriseComesToKotlin(block: (Any, Any, …… /* 42 个 */, Any) -> Any) {
    block(Any(), Any(), ……, Any())
}
```

渐进模式

Kotlin 非常注重代码的稳定性与向后兼容性：Kotlin 兼容性策略提到“破坏性变更”（例如，会使以前编译正常的代码现在不能通过编译的变更）只能在主版本（1.2、1.3 等）中引入。

我们相信很多用户可以使用更快的周期，其中关键的编译器修复会即时生效，从而使代码更安全、更正确。因此 Kotlin 1.3 引入了*渐进*编译器模式，可以通过将参数 `-progressive` 传给编译器来启用。

在渐进模式下，语言语义中的一些修复可以即时生效。所有这些修复都有以下两个重要特征：

— 保留了源代码与旧版编译器的向后兼容性，这意味着可以通过非渐进式编译器编译所有可由渐进式编译器编译的代码。

— 只是在某种意义上使代码*更安全*——例如，可以禁用某些不安全的智能转换，可以将所生成代码的行为更改为更可预测、更稳定，等等。

启用渐进模式可能需要重写一些代码，但不会太多——所有在渐进模式启用的修复都已经过精心挑选、通过审核并提供迁移辅助工具。我们希望对于任何积极维护、即将快速更新到最新语言版本的代码库来说，渐进模式都是一个不错的选择。

内联类

> 内联类在 Kotlin 1.3 起才可用，并且目前是**实验性**的。详见其参考文档。

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：

```
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)
}
```

详见内联类的参考文档。

无符号整型

> 无符号整数仅在 Kotlin 1.3 起才可用，并且目前是**实验性**的。详见其参考文档。

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：

```
// 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
```

详见其参考文档。

@JvmDefault

> `@JvmDefault` 仅在 Kotlin 1.3 起才可用，并且目前是**实验性**的。详见其参考文档页。

Kotlin 兼容很多 Java 版本，其中包括不支持默认方法的 Java 6 与 Java 7。为了方便起见，Kotlin 编译器可以变通突破这个限制，不过这个变通方法与 Java 8 引入的 `default` 方法并不兼容。

这可能会是 Java 互操作性的一个问题，因此 Kotlin 1.3 引入了 `@JvmDefault` 注解。以此注解标注的方法会生成为 JVM 平台的 `default` 方法：

```
interface Foo {
    // 会生成为“default”方法
    @JvmDefault
    fun foo(): Int = 42
}
```

> 警告！以 `@JvmDefault` 注解标注的 API 会对二进制兼容性产生严重影响。在生产中使用 `@JvmDefault` 之前，请务必仔细阅读其参考文档页。

标准库

多平台 `Random`

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。

```
val number = Random.nextInt(42)  // 数字在区间 [0, 上限) 内
println(number)
```

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。

```
val sourceArr = arrayOf("k", "o", "t", "l", "i", "n")
val targetArr = sourceArr.copyInto(arrayOfNulls<String>(6), 3, startIndex = 3, endIndex =
6)
println(targetArr.contentToString())

sourceArr.copyInto(targetArr, startIndex = 0, endIndex = 3)
println(targetArr.contentToString())
```

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) }`。

```
val keys = 'a'..'f'
val map = keys.associateWith { it.toString().repeat(5).capitalize() }
map.forEach { println(it) }
```

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 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"))
```

此外，字符序列与字符串还有一个 `ifBlank` 扩展，它与 `ifEmpty` 类似，只是会检测字符串是否全部都是空白符而不只是空串。

```
val s = "    \n"
println(s.ifBlank { "<blank>" })
println(s.ifBlank { null })
```

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` / `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 IDE 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。

> 请注意，尽管 kotlinx.serialization 现在随 Kotlin 编译器发行版一起发行，但它仍是一个**实验性的**特性。

脚本更新

> 请注意，脚本支持是一项**实验性的**特性，这意味着不会对 API 提供兼容性保证。

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。

开始

基本语法

定义包

包的声明应处于源文件顶部：

```
package my.demo

import java.util.*

// ……
```

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。

定义函数

带有两个 `Int` 参数、返回 `Int` 的函数：

```
fun sum(a: Int, b: Int): Int {
    return a + b
}
```

将表达式作为函数体、返回值类型自动推断的函数：

```
fun sum(a: Int, b: Int) = a + b
```

函数返回无意义的值：

```
fun printSum(a: Int, b: Int): Unit {
    println("sum of $a and $b is ${a + b}")
}
```

`Unit` 返回类型可以省略：

```
fun printSum(a: Int, b: Int) {
    println("sum of $a and $b is ${a + b}")
}
```

参见函数。

定义变量

定义只读局部变量使用关键字 `val` 定义。只能为其赋值一次。

```
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
```

可重新赋值的变量使用 `var` 关键字：

```
var x = 5 // 自动推断出 `Int` 类型
x += 1
```

顶层变量：

```
val PI = 3.14
var x = 0

fun incrementX() {
    x += 1
}
```

参见属性与字段。

注释

正如 Java 与 JavaScript，Kotlin 支持行注释及块注释。

```
// 这是一个行注释

/* 这是一个多行的
   块注释。 */
```

与 Java 不同的是，Kotlin 的块注释可以嵌套。

参见编写 Kotlin 代码文档 查看关于文档注释语法的信息。

使用字符串模板

```
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"
```

参见字符串模板。

使用条件表达式

```
fun maxOf(a: Int, b: Int): Int {
    if (a > b) {
        return a
    } else {
        return b
    }
}
```

使用 if 作为表达式:

```
fun maxOf(a: Int, b: Int) = if (a > b) a else b
```

参见if 表达式。

使用可空值及 null 检测

当某个变量的值可以为 null 的时候,必须在声明处的类型后添加 ? 来标识该引用可为空。

如果 str 的内容不是数字返回 null:

```
fun parseInt(str: String): Int? {
    // ……
}
```

使用返回可空值的函数:

```
fun 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,因为他们可能为 null
    if (x != null && y != null) {
        // 在空检测后,x 与 y 会自动转换为非空值(non-nullable)
        println(x * y)
    }
    else {
        println("either '$arg1' or '$arg2' is not a number")
    }
}
```

或者

```
// ……
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)
```

参见空安全。

使用类型检测及自动类型转换

`is` 运算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：

```
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
}
```

或者

```
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
}
```

甚至

```
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
}
```

参见类以及类型转换。

使用 for 循环

```
val items = listOf("apple", "banana", "kiwifruit")
for (item in items) {
    println(item)
}
```

或者

```
val items = listOf("apple", "banana", "kiwifruit")
for (index in items.indices) {
    println("item at $index is ${items[index]}")
}
```

参见 for 循环。

使用 while 循环

```
val items = listOf("apple", "banana", "kiwifruit")
var index = 0
while (index < items.size) {
    println("item at $index is ${items[index]}")
    index++
}
```

参见 while 循环。

使用 when 表达式

```
fun describe(obj: Any): String =
    when (obj) {
        1          -> "One"
        "Hello"    -> "Greeting"
        is Long    -> "Long"
        !is String -> "Not a string"
        else       -> "Unknown"
    }
```

参见 when 表达式。

使用区间 (range)

使用 in 运算符来检测某个数字是否在指定区间内：

```
val x = 10
val y = 9
if (x in 1..y+1) {
    println("fits in range")
}
```

检测某个数字是否在指定区间外:

```
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")
}
```

区间迭代:

```
for (x in 1..5) {
    print(x)
}
```

或数列迭代：

```
for (x in 1..10 step 2) {
    print(x)
}
println()
for (x in 9 downTo 0 step 3) {
    print(x)
}
```

参见区间。

使用集合

对集合进行迭代:

```
for (item in items) {
    println(item)
}
```

使用 in 运算符来判断集合内是否包含某实例：

```
when {
    "orange" in items -> println("juicy")
    "apple" in items -> println("apple is fine too")
}
```

使用 lambda 表达式来过滤（filter）与映射（map）集合：

```
val fruits = listOf("banana", "avocado", "apple", "kiwifruit")
fruits
  .filter { it.startsWith("a") }
  .sortedBy { it }
  .map { it.toUpperCase() }
  .forEach { println(it) }
```

参见高阶函数及Lambda表达式。

创建基本类及其实例：

```
val rectangle = Rectangle(5.0, 2.0) // 不需要“new”关键字
val triangle = Triangle(3.0, 4.0, 5.0)
```

参见类以及对象与实例。

习惯用法

一些在 Kotlin 中广泛使用的语法习惯，如果你有更喜欢的语法习惯或者风格，建一个 pull request 贡献给我们吧！

创建 DTOs（POJOs/POCOs）

```
data class Customer(val name: String, val email: String)
```

会为 `Customer` 类提供以下功能：

- 所有属性的 getters（对于 `var` 定义的还有 setters）
- `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 }
```

字符串内插

```
println("Name $name")
```

类型判断

```
when (x) {
    is Foo //-> ……
    is Bar //-> ……
    else   //-> ……
}
```

遍历 map/pair型list

```
for ((k, v) in map) {
    println("$k -> $v")
}
```

`k`、`v` 可以改成任意名字。

使用区间

```
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) { …… }
if (x in 1..10) { …… }
```

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

```
println(map["key"])
map["key"] = value
```

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"
}
```

If not null 缩写

```
val files = File("Test").listFiles()

println(files?.size)
```

If not null and else 缩写

```
val files = File("Test").listFiles()

println(files?.size ?: "empty")
```

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() ?: ""
```

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) } ?: defaultValueIfValueIsNull
```

返回 when 表达式

```
fun transform(color: String): Int {
    return when (color) {
        "Red" -> 0
        "Green" -> 1
        "Blue" -> 2
        else -> throw IllegalArgumentException("Invalid color param value")
    }
}
```

“try/catch”表达式

```
fun test() {
    val result = try {
        count()
    } catch (e: ArithmeticException) {
        throw IllegalStateException(e)
    }

    // 使用 result
}
```

“if”表达式

```
fun foo(param: Int) {
    val result = if (param == 1) {
        "one"
    } else if (param == 2) {
        "two"
    } else {
        "three"
    }
}
```

返回类型为 `Unit` 的方法的 Builder 风格用法

```
fun arrayOfMinusOnes(size: Int): IntArray {
    return IntArray(size).apply { fill(-1) }
}
```

单表达式函数

```
fun theAnswer() = 42
```

等价于

```
fun theAnswer(): Int {
    return 42
}
```

单表达式函数与其它惯用法一起使用能简化代码，例如和 `when` 表达式一起使用：

```
fun transform(color: String): Int = when (color) {
    "Red" -> 0
    "Green" -> 1
    "Blue" -> 2
    else -> throw IllegalArgumentException("Invalid color param value")
}
```

对一个对象实例调用多个方法（`with`）

```
class Turtle {
    fun penDown()
    fun penUp()
    fun turn(degrees: Double)
    fun forward(pixels: Double)
}

val myTurtle = Turtle()
with(myTurtle) { // 画一个 100 像素的正方形
    penDown()
    for(i in 1..4) {
        forward(100.0)
        turn(90.0)
    }
    penUp()
}
```

Java 7 的 try with resources

```
val stream = Files.newInputStream(Paths.get("/some/file.txt"))
stream.buffered().reader().use { reader ->
    println(reader.readText())
}
```

对于需要泛型信息的泛型函数的适宜形式

```
//  public final class Gson {
//      ……
//      public <T> T fromJson(JsonElement json, Class<T> classOfT) throws
JsonSyntaxException {
//      ……

inline fun <reified T: Any> Gson.fromJson(json: JsonElement): T = this.fromJson(json,
T::class.java)
```

使用可空布尔

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

编码规范

本页包含当前 Kotlin 语言的编码风格。

- 源代码组织
- 命名规则
- 格式化
- 文档注释
- 避免重复结构
- 语言特性的惯用法
- 库的编码规范

应用风格指南

如需根据本风格指南配置 IntelliJ 格式化程序，请安装 Kotlin 插件 1.2.20 或更高版本，转到“Settings | Editor | Code Style | Kotlin”，点击右上角的“Set from…”链接，并从菜单中选择“Predefined style / Kotlin style guide”。

如需验证代码已按风格指南格式化，请转到探查设置（Inspections）并启用 “Kotlin | Style issues | File is not formatted according to project settings”探查项。验证风格指南中描述的其他问题（如命名约定）的附加探查项默认已启用。

源代码组织

目录结构

在混合语言项目中，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）。

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.foo.bar”中的文件应该放在源代码根目录下的“foo/bar”子目录中）。

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（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定义相同的地方。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。不要只是为了保存 “Foo 的所有扩展函数”而创建文件。

类布局

通常，一个类的内容按以下顺序排列：

- 属性声明与初始化块
- 次构造函数
- 方法声明
- 伴生对象

不要按字母顺序或者可见性对方法声明排序，也不要将常规方法与扩展方法分开。而是要把相关的东西放在一起，这样从上到下阅读类的人就能够跟进所发生事情的逻辑。选择一个顺序（高级别优先，或者相反）并坚持下去。

将嵌套类放在紧挨使用这些类的代码之后。如果打算在外部使用嵌套类，而且类中并没有引用这些类，那么把它们放到末尾，在伴生对象之后。

接口实现布局

在实现一个接口时，实现成员的顺序应该与该接口的成员顺序相同（如果需要，还要插入用于实现的额外的私有方法）

重载布局

在类中总是将重载放在一起。

命名规则

Kotlin 遵循 Java 命名约定。尤其是：

包的名称总是小写且不使用下划线（`org.example.myproject`）。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（`org.example.myProject`）。

类与对象的名称以大写字母开头并使用驼峰：

```
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、使用驼峰而不使用下划线：

```
fun processDeclarations() { …… }
var declarationCount = ……
```

例外：用于创建类实例的工厂函数可以与要创建的类具有相同的名称：

```
abstract class 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。

```
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常规驼峰名称：

```
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（`enum class Color { RED, GREEN }`）也可使用以大写字母开头的常规驼峰名称，具体取决于用途。

幕后属性的名称

如果一个类有两个概念上相同的属性，一个是公共 API 的一部分，另一个是实现细节，那么使用下划线作为私有属性名称的前缀：

```
class C {
    private val _elementList = mutableListOf<Element>()

    val elementList: List<Element>
         get() = _elementList
}
```

选择好名称

类的名称通常是用来解释类*是*什么的名词或者名词短语：`List`、`PersonReader`。

方法的名称通常是动词或动词短语，说明该方法*做*什么：`close`、`readPersons`。修改对象或者返回一个新对象的名称也应遵循建议。例如 `sort` 是对一个集合就地排序，而 `sorted` 是返回一个排序后的集合副本。

名称应该表明实体的目的是什么，所以最好避免在名称中使用无意义的单词（`Manager`、`Wrapper` 等）。

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（`IOStream`）；而如果缩写更长一些，就只大写其首字母（`XmlFormatter`、`HttpInputStream`）。

格式化

在大多数情况下，Kotlin 遵循 Java 编码规范。

使用 4 个空格缩进。不要使用 tab。

对于花括号，将左花括号放在结构起始处的行尾，而将右花括号放在与左括结构横向对齐的单独一行。

```
if (elements != null) {
    for (element in elements) {
        // ……
    }
}
```

（注意：在 Kotlin 中，分号是可选的，因此换行很重要。语言设计采用 Java 风格的花括号格式，如果尝试使用不同的格式化风格，那么可能会遇到意外的行为。）

横向空白

在二元操作符左右留空格（`a + b`）。例外：不要在“range to”操作符（`0..i`）左右留空格。

不要在一元运算符左右留空格（`a++`）

在控制流关键字（`if`、`when`、`for` 以及 `while`）与相应的左括号之间留空格。

不要在主构造函数声明、方法声明或者方法调用的左括号之前留空格。

```
class A(val x: Int)

fun foo(x: Int) { ... }

fun bar() {
    foo(1)
}
```

绝不在 `(`、`[` 之后或者 `]`、`)` 之前留空格。

绝不在 `.` 或者 `?.` 左右留空格：`foo.bar().filter { it > 2 }.joinToString()`，`foo?.bar()`

在 `//` 之后留一个空格：`// 这是一条注释`

不要在用于指定类型参数的尖括号前后留空格：`class Map<K, V> { …… }`

不要在 `::` 前后留空格：`Foo::class`、`String::length`

不要在用于标记可空类型的 `?` 前留空格：`String?`

作为一般规则，避免任何类型的水平对齐。将标识符重命名为不同长度的名称不应该影响声明或者任何用法的格式。

冒号

在以下场景中的 `:` 之前留一个空格：

- 当它用于分隔类型与超类型时；
- 当委托给一个超类的构造函数或者同一类的另一个构造函数时；
- 在 `object` 关键字之后。

而当分隔声明与其类型时，不要在 `:` 之前留空格。

在 `:` 之后总要留一个空格。

```
abstract class Foo<out T : Any> : IFoo {
    abstract fun foo(a: Int): T
}

class FooImpl : Foo() {
    constructor(x: String) : this(x) { …… }

    val x = object : IFoo { …… }
}
```

类头格式化

具有少数主构造函数参数的类可以写成一行：

```
class Person(id: Int, name: String)
```

具有较长类头的类应该格式化，以使每个主构造函数参数都在带有缩进的独立的行中。另外，右括号应该位于一个新行上。如果使用了继承，那么超类的构造函数调用或者所实现接口的列表应该与右括号位于同一行：

```
class Person(
    id: Int,
    name: String,
    surname: String
) : Human(id, name) { …… }
```

对于多个接口，应该将超类构造函数调用放在首位，然后将每个接口应放在不同的行中：

```
class Person(
    id: Int,
    name: String,
    surname: String
) : Human(id, name),
    KotlinMaker { …… }
```

对于具有很长超类型列表的类，在冒号后面换行，并横向对齐所有超类型名：

```
class MyFavouriteVeryLongClassHolder :
    MyLongHolder<MyFavouriteVeryLongClass>(),
    SomeOtherInterface,
    AndAnotherOne {

    fun foo() { ... }
}
```

为了将类头与类体分隔清楚，当类头很长时，可以在类头后放一空行（如上例所示）或者将左花括号放在独立行上：

```
class MyFavouriteVeryLongClassHolder :
    MyLongHolder<MyFavouriteVeryLongClass>(),
    SomeOtherInterface,
    AndAnotherOne
{
    fun foo() { ... }
}
```

构造函数参数使用常规缩进（4 个空格）。

> 理由：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
companion
inline
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`）。

注解格式化

注解通常放在单独的行上，在它们所依附的声明之前，并使用相同的缩进：

```
@Target(AnnotationTarget.PROPERTY)
annotation class JsonExclude
```

无参数的注解可以放在同一行：

```
@JsonExclude @JvmField
var x: String
```

无参数的单个注解可以与相应的声明放在同一行：

```
@Test fun foo() { …… }
```

文件注解

文件注解位于文件注释（如果有的话）之后、`package` 语句之前，并且用一个空白行与 `package` 分开（为了强调其针对文件而不是包）。

```
/** 授权许可、版权以及任何其他内容 */
@file:JvmName("FooBar")

package foo.bar
```

函数格式化

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）。

> 理由：与构造函数参数一致

对于由单个表达式构成的函数体，优先使用表达式形式。

```
fun foo(): Int {     // 不良
    return 1
}

fun foo() = 1        // 良好
```

表达式函数体格式化

如果函数的表达式函数体与函数声明不适合放在同一行，那么将 `=` 留在第一行。将表达式函数体缩进 4 个空格。

```
fun f(x: String) =
    x.length
```

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，那么在等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)
```

格式化控制流语句

如果 `if` 或 `when` 语句的条件有多行，那么在语句体外边总是使用大括号。将该条件的每个后续行相对于条件语句起始处缩进 4 个空格。将该条件的右圆括号与左花括号放在单独一行：

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

> 理由：对齐整齐并且将条件与语句体分隔清楚

将 `else`、`catch`、`finally` 关键字以及 do/while 循环的 `while` 关键字与之前的花括号放在相同的行上：

```
if (condition) {
    // 主体
} else {
    // else 部分
}

try {
    // 主体
} finally {
    // 清理
}
```

在 `when` 语句中，如果一个分支不止一行，可以考虑用空行将其与相邻的分支块分开：

```
private fun parsePropertyValue(propName: String, token: Token) {
    when (token) {
        is Token.ValueToken ->
            callback.visitValue(propName, token.value)

        Token.LBRACE -> { // ……
        }
    }
}
```

将短分支放在与条件相同的行上，无需花括号。

```
when (foo) {
    true -> bar() // 良好
    false -> { baz() } // 不良
}
```

方法调用格式化

在较长参数列表的左括号后添加一个换行符。按 4 个空格缩进参数。将密切相关的多个参数分在同一行。

```
drawSquare(
    x = 10, y = 10,
    width = 100, height = 100,
    fill = true
)
```

在分隔参数名与值的 `=` 左右留空格。

链式调用换行

当对链式调用换行时，将 `.` 字符或者 `?.` 操作符放在下一行，并带有单倍缩进：

```
val anchor = owner
    ?.firstChild!!
    .siblings(forward = true)
    .dropWhile { it is PsiComment || it is PsiWhiteSpace }
```

调用链的第一个调用通常在换行之前，当然如果能让代码更有意义也可以忽略这点。

Lambda 表达式格式化

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，应该尽可能将其放在圆括号外边传入。

```
list.filter { it > 10 }
```

如果为 lambda 表达式分配一个标签，那么不要在该标签与左花括号之间留空格：

```
fun foo() {
    ints.forEach lit@{
        // ……
    }
}
```

在多行的 lambda 表达式中声明参数名时，将参数名放在第一行，后跟箭头与换行符：

```
appendCommaSeparated(properties) { prop ->
    val propertyValue = prop.get(obj)  // ……
}
```

如果参数列表太长而无法放在一行上，请将箭头放在单独一行：

```
foo {
   context: Context,
   environment: Env
   ->
   context.configureEnv(environment)
}
```

文档注释

对于较长的文档注释，将开头 `/**` 放在一个独立行中，并且后续每行都以星号开头：

```
/**
 * 这是一条多行
 * 文档注释。
 */
```

简短注释可以放在一行内：

```
/** 这是一条简短文档注释。 */
```

通常，避免使用 `@param` 与 `@return` 标记。而是将参数与返回值的描述直接合并到文档注释中，并在提到参数的任何地方加上参数链接。只有当需要不适合放进主文本流程的冗长描述时才应使用 `@param` 与 `@return`。

```
// 避免这样：

/**
 * Returns the absolute value of the given number.
 * @param number The number to return the absolute value for.
 * @return The absolute value.
 */
fun abs(number: Int) = ……

// 而要这样：

/**
 * Returns the absolute value of the given [number].
 */
fun abs(number: 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

如果函数返回 Unit，那么应该省略返回类型：

```
fun foo() { // 这里省略了“: Unit”

}
```

分号

尽可能省略分号。

字符串模版

将简单变量传入到字符串模版中时不要使用花括号。只有用到更长表达式时才使用花括号。

```
println("$name has ${children.size} children")
```

语言特性的惯用法

不可变性

优先使用不可变（而不是可变）数据。初始化后未修改的局部变量与属性，总是将其声明为 `val` 而不是 `var` 。

总是使用不可变集合接口（`Collection`，`List`，`Set`，`Map`）来声明无需改变的集合。使用工厂函数创建集合实例时，尽可能选用返回不可变集合类型的函数：

```
// 不良：使用可变集合类型作为无需改变的值
fun validateValue(actualValue: String, allowedValues: HashSet<String>) { …… }

// 良好：使用不可变集合类型
fun validateValue(actualValue: String, allowedValues: Set<String>) { …… }

// 不良：arrayListOf() 返回 ArrayList<T>，这是一个可变集合类型
val allowedValues = arrayListOf("a", "b", "c")

// 良好：listOf() 返回 List<T>
val allowedValues = listOf("a", "b", "c")
```

默认参数值

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：

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

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，始终应该显式声明参数。

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。

命名参数

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，请使用命名参数语法，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)
```

使用条件语句

优先使用 `try`、`if` 与 `when` 的表达形式。例如：

```
return if (x) foo() else bar()

return when(x) {
    0 -> "zero"
    else -> "nonzero"
}
```

优先选用上述代码而不是：

```
if (x)
    return foo()
else
    return bar()

when(x) {
    0 -> return "zero"
    else -> return "nonzero"
}
```

`if` 还是 `when`

二元条件优先使用 `if` 而不是 `when`。不要使用

```
when (x) {
    null -> ……
    else -> ……
}
```

而应使用 `if (x == null) …… else ……`

如果有三个或多个选项时优先使用 `when`。

在条件中使用可空的 `Boolean` 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。

使用循环

优先使用高阶函数（`filter`、`map` 等）而不是循环。例外：`forEach`（优先使用常规的 `for` 循环，除非 `forEach` 的接收者是可空的或者 `forEach` 用做长调用链的一部分。）

当在使用多个高阶函数的复杂表达式与循环之间进行选择时，请了解每种情况下所执行操作的开销并且记得考虑性能因素。

区间上循环

使用 `until` 函数在一个区间上循环：

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

使用字符串

优先使用字符串模板而不是字符串拼接。

优先使用多行字符串而不是将 `\n` 转义序列嵌入到常规字符串字面值中。

如需在多行字符串中维护缩进，当生成的字符串不需要任何内部缩进时使用 `trimIndent`，而需要内部缩进时使用 `trimMargin`：

```
assertEquals(
    """
    Foo
    Bar
    """.trimIndent(),
    value
)

val a = """if(a > 1) {
          |    return a
          |}""".trimMargin()
```

函数还是属性

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

使用扩展函数

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。

使用中缀函数

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。

例如：

```
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：

```
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)
}
```

使用作用域函数 apply/with/run/also/let

Kotlin 提供了一系列用来在给定对象上下文中执行代码块的函数。要选择正确的函数，请考虑以下几点：

— 是否在块中的多个对象上调用方法，或者将上下文对象的实例作为参数传递?如果是，那么使用以 `it` 而不是 `this` 形式访问上下文对象的函数之一（`also` 或 `let`）。如果在代码块中根本没有用到接收者，那么使用 `also`。

```
// 上下文对象是“it”
class Baz {
    var currentBar: Bar?
    val observable: Observable

    val foo = createBar().also {
        currentBar = it                     // 访问 Baz 的属性
        observable.registerCallback(it)     // 将上下文对象作为参数传递
    }
}

// 代码块中未使用接收者
val foo = createBar().also {
    LOG.info("Bar created")
}

// 上下文对象是“this”
class Baz {
    val foo: Bar = createBar().apply {
        color = RED     // 只访问 Bar 的属性
        text = "Foo"
    }
}
```

— 调用的结果是什么?如果结果需是该上下文对象，那么使用 `apply` 或 `also`。如果需要从代码块中返回一个值，那么使用 `with`、`let` 或者 `run`

```
// 返回值是上下文对象
class Baz {
    val foo: Bar = createBar().apply {
        color = RED     // 只访问 Bar 的属性
        text = "Foo"
    }
}


// 返回值是代码块的结果
class Baz {
    val foo: Bar = createNetworkConnection().let {
        loadBar()
    }
}
```

— 上下文对象是否可空，或者是否作为调用链的结果求值而来的?如果是，那么使用 `apply`、`let` 或者 `run`。否则，使用 `with` 或者 `also`。

```
// 上下文对象可空
person.email?.let { sendEmail(it) }

// 上下文对象非空且可直接访问
with(person) {
    println("First name: $firstName, last name: $lastName")
}
```

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）

基础

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、字符以及布尔值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、字符、布尔值、数组与字符串。

数字

Kotlin 处理数字在某种程度上接近 Java，但是并不完全相同。例如，对于数字没有隐式拓宽转换（如 Java 中 `int` 可以隐式转换为 `long` ——译者注），另外有些情况的字面值略有不同。

Kotlin 提供了如下的内置类型来表示数字（与 Java 很相近）：

Type	Bit width
Double	64
Float	32
Long	64
Int	32
Short	16
Byte	8

注意在 Kotlin 中字符不是数字

字面常量

数值常量字面值有以下几种:

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

注意: 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`

数字字面值中的下划线（自 1.1 起）

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
```

表示方式

在 Java 平台数字是物理存储为 JVM 的原生类型，除非我们需要一个可空的引用（如 `Int?`）或泛型。后者情况下会把数字装箱。

注意数字装箱不一定保留同一性：

```
val a: Int = 10000
println(a === a) // 输出“true”
val boxedA: Int? = a
val anotherBoxedA: Int? = a
println(boxedA === anotherBoxedA) // !!!输出“false”!!!
```

另一方面，它保留了相等性：

```
val a: Int = 10000
println(a == a) // 输出“true”
val boxedA: Int? = a
val anotherBoxedA: Int? = a
println(boxedA == anotherBoxedA) // 输出“true”
```

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
```

所以相等性会在所有地方悄无声息地失去，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。这意味着在不进行显式转换的情况下我们不能把 `Byte` 型值赋给一个 `Int` 变量。

```
val b: Byte = 1 // OK, 字面值是静态检测的
val i: Int = b // 错误
```

我们可以显式转换来拓宽数字

```
val i: Int = b.toInt() // OK:显式拓宽
print(i)
```

每个数字类型支持如下的转换:

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

缺乏隐式类型转换很少会引起注意,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,运算被定义为相应的类成员(但编译器会将函数调用优化为相应的指令)。参见运算符重载。

对于位运算,没有特殊字符来表示,而只可用中缀方式调用命名函数,例如:

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

这是完整的位运算列表(只用于 `Int` 与 `Long`):

- `shl(bits)` – 有符号左移 (Java 的 `<<`)
- `shr(bits)` – 有符号右移 (Java 的 `>>`)
- `ushr(bits)` – 无符号右移 (Java 的 `>>>`)
- `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`

字符

字符用 `Char` 类型表示。它们不能直接当作数字

```
fun check(c: Char) {
    if (c == 1) { // 错误:类型不兼容
        // ……
    }
}
```

字符字面值用单引号括起来：`'1'`。特殊字符可以用反斜杠转义。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'`。

我们可以显式把字符转换为 `Int` 数字：

```
fun decimalDigitValue(c: Char): Int {
    if (c !in '0'..'9')
        throw IllegalArgumentException("Out of range")
    return c.toInt() - '0'.toInt() // 显式转换为数字
}
```

当需要可空引用时，像数字、字符会被装箱。装箱操作不会保留同一性。

布尔

布尔用 `Boolean` 类型表示，它有两个值：`true` 与 `false`。

若需要可空引用布尔会被装箱。

内置的布尔运算有：

— `||` – 短路逻辑或

— `&&` – 短路逻辑与

— `!` - 逻辑非

数组

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 `[]`）以及 `size` 属性，以及一些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>
    // ……
}
```

我们可以使用库函数 `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，库函数 `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每个元素初始值：

```
// 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) }
```

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

注意: 与 Java 不同的是，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）。

Kotlin 也有无装箱开销的专门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]
```

无符号整型

> 无符号类型自 Kotlin 1.3 起才可用，并且目前是*实验性的*。详见下文

Kotlin 为无符号整数引入了以下类型：

— `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

无符号类型支持其对应有符号类型的大多数操作。

> 请注意，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

无符号类型是使用另一个实验性特性（即内联类）实现的。

特化的类

与原生类型相同，每个无符号类型都有相应的为该类型特化的表示数组的类型：

— `kotlin.UByteArray`：无符号字节数组

— `kotlin.UShortArray`：无符号短整型数组

— `kotlin.UIntArray`：无符号整型数组

— `kotlin.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。

此外，区间与数列也支持 `UInt` 与 `ULong`（通过这些类 `kotlin.ranges.UIntRange`、`kotlin.ranges.UIntProgression`、`kotlin.ranges.ULongRange`、`kotlin.ranges.ULongProgression`）

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/Long）：

— 后缀 `u` 与 `U` 将字面值标记为无符号。确切类型会根据预期类型确定。如果没有提供预期的类型，会根据字面值大小选择 `UInt` 或者 `ULong`

```
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
```

— 后缀 `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
```

无符号整型的实验性状态

无符号类型的设计是实验性的，这意味着这个特性改进很快并且没有给出兼容性保证。当在 Kotlin 1.3+ 中使用无符号算术时，会报出警告表明这个特性是实验性的。如需移除警告，必须选择加入（opt-in）无符号类型的实验性使用。

选择加入无符号整型有两种可行的方式：将 API 标记为实验性的，或者无需标记。

— 如需传播实验性，要么使用 `@ExperimentalUnsignedTypes` 标注使用了无符号整型的声明，要么将 `-Xexperimental=kotlin.ExperimentalUnsignedTypes` 传给编译器（请注意，后者会使所编译的模块内的*所有*声明都具实验性）

— 如需选择加入而不传播实验性，要么使用 `@UseExperimental(ExperimentalUnsignedTypes::class)` 注解标注声明，要么将 `-Xuse-experimental=kotlin.ExperimentalUnsignedTypes` 传给编译器

你的客户是否必须选择使用你的 API 取决于你，不过请记住，无符号整型是一个实验性特性，因此使用它们的 API 可能会因语言的变更而发生突然破坏。

技术细节也参见实验性 API KEEP。

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。

字符串

字符串用 `String` 类型表示。字符串是不可变的。字符串的元素——字符可以使用索引运算符访问：`s[i]`。可以用 `for` 循环迭代字符串：

```
for (c in str) {
    println(c)
}
```

可以用 `+` 操作符连接字符串。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：

```
val s = "abc" + 1
println(s + "def")
```

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：转义字符串可以有转义字符，以及原始字符串可以包含换行以及任意文本。转义字符串很像 Java 字符串：

```
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)
"""
```

你可以通过 `trimMargin()` 函数去除前导空格：

```
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

字符串可以包含*模板表达式*，即一些小段代码，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，由一个简单的名字构成：

```
val i = 10
println("i = $i") // 输出“i = 10”
```

或者用花括号括起来的任意表达式：

```
val s = "abc"
println("$s.length is ${s.length}") // 输出“abc.length is 3”
```

原始字符串与转义字符串内部都支持模板。如果你需要在原始字符串中表示字面值 `$` 字符（它不支持反斜杠转义），你可以用下列语法：

```
val price = """
${'$'}9.99
"""
```

包

源文件通常以包声明开头:

```
package foo.bar

fun baz() { ... }
class Goo { ... }

// ……
```

源文件所有内容(无论是类还是函数)都包含在声明的包内。所以上例中 `baz()` 的全名是 `foo.bar.baz`、`Goo` 的全名是 `foo.bar.Goo`。

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.* (自 1.1 起)
- 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指令。导入语法在语法中讲述。

可以导入一个单独的名字,如.

```
import foo.Bar // 现在 Bar 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 foo.* // “foo”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 foo.Bar // Bar 可访问
import bar.Bar as bBar // bBar 代表“bar.Bar”
```

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。

与 Java 不同，Kotlin 没有单独的“import static”语法；所有这些声明都用 `import` 关键字导入。

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）。

控制流：if、when、for、while

If 表达式

在 Kotlin 中，`if`是一个表达式，即它会返回一个值。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。

```
// 传统用法
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`的分支可以是代码块，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
}
```

如果你使用 `if` 作为表达式而不是语句（例如：返回它的值或者把它赋给变量），该表达式需要有 `else` 分支。

参见 `if` 语法。

When 表达式

`when` 取代了类 C 语言的 switch 操作符。其最简单的形式如下：

```
when (x) {
    1 -> print("x == 1")
    2 -> print("x == 2")
    else -> { // 注意这个块
        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 既可以被当做表达式使用也可以被当做语句使用。如果它被当做表达式，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。(像 if 一样，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，则必须有 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]。

如果很多分支需要用相同的方式处理，则可以把多个分支条件放在一起，用逗号分隔：

```
when (x) {
    0, 1 -> print("x == 0 or x == 1")
    else -> print("otherwise")
}
```

我们可以用任意表达式 (而不只是常量) 作为分支条件

```
when (x) {
    parseInt(s) -> print("s encodes x")
    else -> print("s does not encode x")
}
```

我们也可以检测一个值在 (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")
}
```

另一种可能性是检测一个值是 (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。

```
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")
    x.isEven() -> print("x is even")
    else -> print("x is funny")
}
```

Since Kotlin 1.3, it is possible to capture when subject in a variable using following syntax:

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

Scope of variable, introduced in `when` subject, is restricted to `when` body.

参见 when 语法。

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。语法如下:

```
for (item in collection) print(item)
```

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。即:

- 有一个成员函数或者扩展函数 `iterator()`,它的返回类型
  - 有一个成员函数或者扩展函数 `next()`,并且
  - 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:

```
for (i in 1..3) {
    println(i)
}
for (i in 6 downTo 0 step 2) {
    println(i)
}
```

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:

```
for (i in array.indices) {
    println(array[i])
}
```

或者你可以用库函数 `withIndex`:

```
for ((index, value) in array.withIndex()) {
    println("the element at $index is $value")
}
```

参见for 语法。

While 循环

while 与 do..while 照常使用

```
while (x > 0) {
    x--
}

do {
  val y = retrieveData()
} while (y != null) // y 在此处可见
```

参见while 语法.

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。

返回和跳转

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（`label`）来标记。标签的格式为标识符后跟 `@` 符号，例如：`abc@`、`fooBar@`都是有效的标签（参见语法）。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，我们可以用标签限制 `break` 或者`continue`：

```
loop@ for (i in 1..100) {
    for (j in 1..100) {
        if (……) break@loop
    }
}
```

标签限制的 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。

标签处返回

Kotlin 有函数字面量、局部函数和对象表达式。因此 Kotlin 的函数可以被嵌套。标签限制的 `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：

```
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")
}
```

这个 return 表达式从最直接包围它的函数即 foo 中返回。(注意，这种非局部的返回只支持传给内联函数的 lambda 表达式。) 如果我们需要从 lambda 表达式中返回，我们必须给它加标签并用以限制 return。

```
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，即 forEach 循环
        print(it)
    }
    print(" done with explicit label")
}
```

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。该标签与接受该 lambda 的函数同名。

```
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，即 forEach 循环
        print(it)
    }
    print(" done with implicit label")
}
```

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

```
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，即 forEach 循环
        print(value)
    })
    print(" done with anonymous function")
}
```

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：

```
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")
}
```

当要返一个回值的时候，解析器优先选用标签限制的 return，即

```
return@a 1
```

意为“从标签 @a 返回 1”，而不是“返回一个标签标注的表达式 (@a 1)”。

类与对象

类与继承

类

Kotlin 中使用关键字 `class` 声明类

```
class Invoice { ... }
```

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：

```
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}")
    }
}
```

请注意，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.toUpperCase()
}
```

事实上，声明属性以及从主构造函数初始化属性，Kotlin 有简洁的语法：

```
class Person(val firstName: String, val lastName: String, var age: Int) { …… }
```

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) { …… }
```

更多详情，参见可见性修饰符

次构造函数

类也可以声明前缀有 `constructor`的**次构造函数**：

```
class Person {
    constructor(parent: Person) {
        parent.children.add(this)
    }
}
```

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) {
    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：

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

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

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,你需要声明一个带有非默认可见性的空的主构造函数:

```
class DontCreateMe private constructor () { ... }
```

> **注意**:在 JVM 上,如果主构造函数的所有的参数都有默认值,编译器会生成 一个额外的无参构造函数,它将使用默认值。这使得 Kotlin 更易于使用像 Jackson 或者 JPA 这样的通过无参构造函数创建类的实例的库。

```
class Customer(val customerName: String = "")
```

创建类的实例

要创建一个类的实例,我们就像普通函数一样调用构造函数:

```
val invoice = Invoice()

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

注意 Kotlin 并没有 new 关键字。

创建嵌套类、内部类与匿名内部类的类实例在嵌套类中有述。

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

在 Kotlin 中所有类都有一个共同的超类 `Any`,这对于没有超类型声明的类是默认超类:

```
class Example // 从 Any 隐式继承
```

> 注意:Any 并不是 `java.lang.Object`;尤其是,它除了 `equals()`、`hashCode()` 与 `toString()` 外没有任何成员。更多细节请查阅Java互操作性部分。

要声明一个显式的超类型,我们把类型放到类头的冒号之后:

```
open class Base(p: Int)

class Derived(p: Int) : Base(p)
```

如果派生类有一个主构造函数，其基类型可以（并且必须）用基类的主构造函数参数就地初始化。

如果类没有主构造函数，那么每个次构造函数必须使用 super 关键字初始化其基类型，或委托给另一个构造函数做到这一点。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

我们之前提到过，Kotlin 力求清晰显式。与 Java 不同，Kotlin 对于可覆盖的成员（我们称之为*开放*）以及覆盖后的成员需要显式修饰符：

```
open class Base {
    open fun v() { ... }
    fun nv() { ... }
}
class Derived() : Base() {
    override fun v() { ... }
}
```

Derived.v() 函数上必须加上 override 修饰符。如果没写，编译器将会报错。如果函数没有标注 open 如 `Base.nv()`，那么子类中不允许定义相同签名的函数，不论加不加 **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，也就是说，它可以在子类中覆盖。如果你想禁止再次覆盖，使用 final 关键字：

```
open class AnotherDerived() : Base() {
    final override fun v() { ... }
}
```

覆盖属性

属性覆盖与方法覆盖类似；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 getter 方法的属性覆盖。

```
open class Foo {
    open val x: Int get() { …… }
}

class Bar1 : Foo() {
    override val x: Int = ……
}
```

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 getter 方法，而将其覆盖为 `var` 只是在子类中额外声明一个 setter 方法。

请注意，你可以在主构造函数中使用 `override` 关键字作为属性声明的一部分。

```
interface Foo {
    val count: Int
}

class Bar1(override val count: Int) : Foo

class Bar2 : Foo {
    override var count: Int = 0
}
```

派生类初始化顺序

在构造派生类的新实例的过程中，第一步完成其基类的初始化（在之前只有对基类构造函数参数的求值），因此发生在派生类的初始化逻辑运行之前。

```
open class Base(val name: String) {

    init { println("Initializing Base") }

    open val size: Int =
        name.length.also { println("Initializing size in Base: $it") }
}

class Derived(
    name: String,
    val lastName: String
) : Base(name.capitalize().also { println("Argument for Base: $it") }) {

    init { println("Initializing Derived") }

    override val size: Int =
        (super.size + lastName.length).also { println("Initializing size in Derived: $it")
}
}
```

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。如果在基类初始化逻辑中（直接或通过另一个覆盖的 `open` 成员的实现间接）使用了任何一个这种属性，那么都可能导致不正确的行为或运行时故障。设计一个基类时，应该避免在构造函数、属性初始化器以及 `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 Foo {
    open fun f() { println("Foo.f()") }
    open val x: Int get() = 1
}

class Bar : Foo() {
    override fun f() {
        super.f()
        println("Bar.f()")
    }

    override val x: Int get() = super.x + 1
}
```

在一个内部类中访问外部类的超类，可以通过由外部类名限定的 `super` 关键字来实现：`super@Outer`：

```
class Bar : Foo() {
    override fun f() { /* …… */ }
    override val x: Int get() = 0

    inner class Baz {
        fun g() {
            super@Bar.f() // 调用 Foo 实现的 f()
            println(super@Bar.x) // 使用 Foo 实现的 x 的 getter
        }
    }
}
```

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）。为了表示采用从哪个超类型继承的实现，我们使用由尖括号中超类型名限定的 `super`，如 `super<Base>`：

```
open class A {
    open fun f() { print("A") }
    fun a() { print("a") }
}

interface B {
    fun f() { print("B") } // 接口成员默认就是“open”的
    fun b() { print("b") }
}

class C() : A(), B {
    // 编译器要求覆盖 f():
    override fun f() {
        super<A>.f() // 调用 A.f()
        super<B>.f() // 调用 B.f()
  }
}
```

同时继承 `A` 与 `B` 没问题，并且 `a()` 与 `b()` 也没问题因为 `C` 只继承了每个函数的一个实现。但是 `f()` 由 `C` 继承了两个实现，所以我们**必须**在 `C` 中覆盖 `f()` 并且提供我们自己的实现来消除歧义。

抽象类

类以及其中的某些成员可以声明为 `abstract`。抽象成员在本类中可以不用实现。需要注意的是，我们并不需要用 `open` 标注一个抽象类或者函数——因为这不言而喻。

我们可以用一个抽象成员覆盖一个非抽象的开放成员

```
open class Base {
    open fun f() {}
}

abstract class Derived : Base() {
    override abstract fun f()
}
```

伴生对象

与 Java 或 C# 不同，在 Kotlin 中类没有静态方法。在大多数情况下，它建议简单地使用包级函数。

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，你就可以使用像在 Java/C# 中调用静态方法相同的语法来调用其成员，只使用类名作为限定符。

属性与字段

声明属性

Kotlin的类可以有属性。属性可以用关键字var 声明为可变的，否则使用只读关键字val。

```
class Address {
    var name: String = ……
    var street: String = ……
    var city: String = ……
    var state: String? = ……
    var zip: String = ……
}
```

要使用一个属性，只要用名称引用它即可，就像 Java 中的字段：

```
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
}
```

Getters 与 Setters

声明一个属性的完整语法是

```
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（或者从其 getter 返回值，如下文所示）中推断出来，也可以省略。

例如:

```
var allByDefault: Int? // 错误:需要显式初始化器,隐含默认 getter 和 setter
var initialized = 1 // 类型 Int、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` 开始代替 `var`

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
```

我们可以为属性定义自定义的访问器。如果我们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（这让我们可以实现计算出的属性）。以下是一个自定义 getter 的示例：

```
val isEmpty: Boolean
    get() = this.size == 0
```

如果我们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。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。

自 Kotlin 1.1 起，如果可以从 getter 推断出属性类型，则可以省略它：

```
val isEmpty get() = this.size == 0  // 具有类型 Boolean
```

如果你需要改变一个访问器的可见性或者对其注解，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:

```
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 类中不能直接声明字段。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 // 注意:这个初始器直接为幕后字段赋值
    set(value) {
        if (value >= 0) field = value
    }
```

`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，下面的情况下，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")
    }
```

从各方面看，这正是与 Java 相同的方式。因为通过默认 getter 和 setter 访问私有属性会被优化，所以不会引入函数调用开销。

编译期常量

已知值的属性可以使用 `const` 修饰符标记为 *编译期常量*。这些属性需要满足以下要求：

- 位于顶层或者是 `object` 声明 或 `companion object` 的一个成员
- 以 `String` 或原生类型值初始化
- 没有自定义 getter

这些属性可以用在注解中：

```
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查。

为处理这种情况，你可以用 `lateinit` 修饰符标记该属性：

```
public class MyTest {
    lateinit var subject: TestSubject

    @SetUp fun setup() {
        subject = TestSubject()
    }

    @Test fun test() {
        subject.method()  // 直接解引用
    }
}
```

该修饰符只能用于在类体中的属性（不是在主构造函数中声明的 `var` 属性，并且仅当该属性没有自定义 getter 或 setter 时），而自 Kotlin 1.2 起，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（自 1.2 起）

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，即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。

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）。另一方面，使用自定义 getter 和 setter 可以实现属性的任何行为。介于两者之间，属性如何工作有一些常见的模式。一些例子：惰性值、通过键值从映射读取、访问数据库、访问时通知侦听器等等。

这些常见行为可以通过使用*委托属性*实现为库。

接口

Kotlin 的接口与 Java 8 类似，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

你可以在接口中定义属性。在接口中声明的属性要么是抽象的，要么提供访问器的实现。在接口中声明的属性不能有幕后字段（backing field），因此接口中声明的访问器不能引用它们。

```
interface MyInterface {
    val prop: Int // 抽象的

    val propertyWithImplementation: String
        get() = "foo"

    fun foo() {
        print(prop)
    }
}

class Child : MyInterface {
    override val prop: Int = 29
}
```

接口继承

一个接口可以从其他接口派生，从而既提供基类型成员的实现也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。例如

```
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，因为没有方法体时默认为抽象）。因为 *C* 是一个实现了 *A* 的具体类，所以必须要重写 *bar()* 并实现这个抽象方法。

然而，如果我们从 *A* 和 *B* 派生 *D*，我们需要实现我们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。

可见性修饰符

类、对象、接口、构造函数、方法、属性和它们的 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`。如果没有显式指定修饰符的话，默认可见性是 `public`。

以下解释了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、对象和接口可以在顶层声明，即直接在包内：

```
// 文件名:example.kt
package foo

fun baz() { ... }
class Bar { ... }
```

- 如果你不指定任何可见性修饰符，默认为 `public`，这意味着你的声明将随处可见；
- 如果你声明为 `private`，它只会在声明它的文件内可见；
- 如果你声明为 `internal`，它会在相同模块内随处可见；
- `protected` 不适用于顶层声明。

注意：要使用另一包中可见的顶层声明，仍需将其导入进来。

例如：

```
// 文件名:example.kt
package foo

private fun foo() { …… } // 在 example.kt 内可见

public var bar: Int = 5 // 该属性随处可见
    private set         // setter 只在 example.kt 内可见

internal val baz = 6    // 相同模块内可见
```

类和接口

对于类内部声明的成员：

- `private` 意味着只在这个类内部（包含其所有成员）可见；
- `protected` —— 和 `private` 一样 + 在子类中可见。
- `internal` —— 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` —— 能见到类声明的任何客户端都可见其 `public` 成员。

注意 对于Java用户：Kotlin 中外部类不能访问内部类的 private 成员。

如果你覆盖一个 `protected` 成员并且没有显式指定其可见性，该成员还会是 `protected` 可见性。

例子:

```
open class Outer {
    private val a = 1
    protected open val b = 2
    internal 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
}

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

要指定一个类的的主构造函数的可见性，使用以下语法（注意你需要添加一个显式 `constructor` 关键字）：

```
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

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：

- 一个 IntelliJ IDEA 模块；
- 一个 Maven 项目；
- 一个 Gradle 源集（例外是 `test` 源集可以访问 `main` 的 internal 声明）；

— 一次 `<kotlinc>` Ant 任务执行所编译的一套文件。

扩展

Kotlin 同 C# 与 Gosu 类似，能够扩展一个类的新功能而无需继承该类或使用像装饰者这样的任何类型的设计模式。这通过叫做 *扩展* 的特殊声明完成。Kotlin 支持 *扩展函数* 与 *扩展属性*。

扩展函数

声明一个扩展函数，我们需要用一个 *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，我们对任意 `MutableList<Int>` 调用该函数了：

```
val l = mutableListOf(1, 2, 3)
l.swap(0, 2) // “swap()”内部的“this”得到“l”的值
```

当然，这个函数对任何 `MutableList<T>` 起作用，我们可以泛化它：

```
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，我们要在函数名前声明泛型参数。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，你并没有在一个类中插入新成员，仅仅是可以通过该类型的变量用点表达式去调用这个新函数。

我们想强调的是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。这意味着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：

```
open class C

class D: C()

fun C.foo() = "c"

fun D.foo() = "d"

fun printFoo(c: C) {
    println(c.foo())
}

printFoo(D())
```

这个例子会输出 "c"，因为调用的扩展函数只取决于参数 `c` 的声明类型，该类型是 `C` 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，都适用给定的参数，这种情况**总是取成员函数**。例如：

```
class C {
    fun foo() { println("member") }
}

fun C.foo() { println("extension") }
```

如果我们调用 `C` 类型 `c` 的 `c.foo()`，它将输出“member”，而不是“extension”。

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：

```
class C {
    fun foo() { println("member") }
}

fun C.foo(i: Int) { println("extension") }
```

调用 `C().foo(1)` 将输出 "extension"。

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`，这能让你在没有检测 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

与函数类似，Kotlin 支持扩展属性：

```
val <T> List<T>.lastIndex: Int
    get() = size - 1
```

注意：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 getters/setters 定义。

例如：

```
val Foo.bar = 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：

```
class MyClass {
    companion object { }  // 将被称为 "Companion"
}

fun MyClass.Companion.foo() { …… }
```

就像伴生对象的其他普通成员，只需用类名作为限定符去调用他们

```
MyClass.foo()
```

扩展的作用域

大多数时候我们在顶层定义扩展，即直接在包里：

```
package foo.bar

fun Baz.goo() { …… }
```

要使用所定义包之外的一个扩展，我们需要在调用方导入它：

```
package com.example.usage

import foo.bar.goo // 导入所有名为“goo”的扩展
                   // 或者
import foo.bar.*   // 从“foo.bar”导入一切

fun usage(baz: Baz) {
    baz.goo()
}
```

更多信息参见导入

扩展声明为成员

在一个类内部你可以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 D {
    fun bar() { …… }
}

class C {
    fun baz() { …… }

    fun D.foo() {
        bar()   // 调用 D.bar
        baz()   // 调用 C.baz
    }

    fun caller(d: D) {
        d.foo()   // 调用扩展函数
    }
}
```

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 C {
    fun D.foo() {
        toString()         // 调用 D.toString()
        this@C.toString()  // 调用 C.toString()
    }
}
```

声明为成员的扩展可以声明为 `open` 并在子类中覆盖。这意味着这些函数的分发对于分发接收者类型是虚拟的，但对于扩展接收者类型是静态的。

```
open class D { }

class D1 : D() { }

open class C {
    open fun D.foo() {
        println("D.foo in C")
    }

    open fun D1.foo() {
        println("D1.foo in C")
    }

    fun caller(d: D) {
        d.foo()   // 调用扩展函数
    }
```

```
}

class C1 : C() {
    override fun D.foo() {
        println("D.foo in C1")
    }

    override fun D1.foo() {
        println("D1.foo in C1")
    }
}

fun main() {
    C().caller(D())   // 输出 "D.foo in C"
    C1().caller(D())  // 输出 "D.foo in C1" —— 分发接收者虚拟解析
    C().caller(D1())  // 输出 "D.foo in C" —— 扩展接收者静态解析
}
```

关于可见性的说明

扩展的可见性与相同作用域内声明的其他实体的可见性相同。例如：

— 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，那么该扩展不能访问接收者的 `private` 成员。

动机

在Java中，我们将类命名为“*Utils”：`FileUtils`、`StringUtils` 等，著名的 `java.util.Collections` 也属于同一种命名方式。关于这些 Utils-类的不愉快的部分是代码写成这样：

```
// Java
Collections.swap(list, Collections.binarySearch(list,
    Collections.max(otherList)),
    Collections.max(list));
```

这些类名总是碍手碍脚的，我们可以通过静态导入达到这样效果：

```
// Java
swap(list, binarySearch(list, max(otherList)), max(list));
```

这会变得好一点，但是我们并没有从 IDE 强大的自动补全功能中得到帮助。如果能这样就更好了：

```
// Java
list.swap(list.binarySearch(otherList.max()), list.max());
```

但是我们不希望在 `List` 类内实现这些所有可能的方法，对吧?这时候扩展将会帮助我们。

数据类

我们经常创建一些只保存数据的类。在这些类中，一些标准函数往往是从数据机械推导而来的。在 Kotlin 中，这叫做 *数据类* 并标记为 `data`：

```
data class User(val name: String, val age: Int)
```

编译器自动从主构造函数中声明的所有属性导出以下成员：

- `equals()`/`hashCode()` 对；
- `toString()` 格式是 `"User(name=John, age=42)"`；
- componentN() 函数 按声明顺序对应于所有属性；
- `copy()` 函数（见下文）。

为了确保生成的代码的一致性以及有意义的行为，数据类必须满足以下要求：

- 主构造函数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；
- （在1.1之前）数据类只能实现接口。

此外，成员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；
- 从一个已具 `copy(……)` 函数且签名匹配的类型派生一个数据类在 Kotlin 1.2 中已弃用，并且会在 Kotlin 1.3 中禁用。
- 不允许为 `componentN()` 以及 `copy()` 函数提供显式实现。

自 1.1 起，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，则所有的属性必须指定默认值。（参见构造函数）。

```
data class User(val name: String = "", val age: Int = 0)
```

在类体中声明的属性

请注意，对于那些自动生成的函数，编译器只使用在主构造函数内部定义的属性。如需在生成的实现中排出一个属性，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。

```
val person1 = Person("John")
val person2 = Person("John")
person1.age = 10
person2.age = 20
```

复制

在很多情况下，我们需要复制一个对象改变它的一些属性，但其余部分保持不变。`copy()` 函数就是为此而生成。对于上文的 `User` 类，其实现会类似下面这样：

```
fun copy(name: String = this.name, age: Int = this.age) = User(name, age)
```

这让我们可以写：

```
val jack = User(name = "Jack", age = 1)
val olderJack = jack.copy(age = 2)
```

数据类与解构声明

为数据类生成的 *Component 函数* 使它们可在解构声明中使用：

```
val jane = User("Jane", 35)
val (name, age) = jane
println("$name, $age years of age") // 输出 "Jane, 35 years of age"
```

标准数据类

标准库提供了 `Pair` 与 `Triple`。尽管在很多情况下命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

密封类用来表示受限的类继承结构：当一个值为有限集中的类型、而不能有任何其他类型时。在某种意义上，他们是枚举类的扩展：枚举类型的值集合也是受限的，但每个枚举常量只存在一个实例，而密封类的一个子类可以有可包含状态的多个实例。

要声明一个密封类，需要在类名前面添加 `sealed` 修饰符。虽然密封类也可以有子类，但是所有子类都必须在与密封类自身相同的文件中声明。（在 Kotlin 1.1 之前，该规则更加严格：子类必须嵌套在密封类声明的内部）。

```
sealed class Expr
data class Const(val number: Double) : Expr()
data class Sum(val e1: Expr, val e2: Expr) : Expr()
object NotANumber : Expr()
```

（上文示例使用了 Kotlin 1.1 的一个额外的新功能：数据类扩展包括密封类在内的其他类的可能性。）

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。

密封类不允许有非-`private` 构造函数（其构造函数默认为 `private`）。

请注意，扩展密封类子类的类（间接继承者）可以放在任何位置，而无需在同一个文件中。

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 eval(expr: Expr): Double = when(expr) {
    is Const -> expr.number
    is Sum -> eval(expr.e1) + eval(expr.e2)
    NotANumber -> Double.NaN
    // 不再需要 `else` 子句，因为我们已经覆盖了所有的情况
}
```

泛型

与 Java 类似，Kotlin 中的类也可以有类型参数：

```
class Box<T>(t: T) {
    var value = t
}
```

一般来说，要创建这样类的实例，我们需要提供类型参数：

```
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，允许省略类型参数：

```
val box = Box(1) // 1 具有类型 Int，所以编译器知道我们说的是 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，它有两个其他的东西：声明处型变（declaration-site variance）与类型投影（type projections）。

首先，让我们思考为什么 Java 需要那些神秘的通配符。在《Effective Java》第三版 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。为什么这样？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然后导致运行时异常：

```
// Java
List<String> strs = new ArrayList<String>();
List<Object> objs = strs; // !!!即将来临的问题的原因就在这里。Java 禁止这样！
objs.add(1); // 这里我们把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
```

因此，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，我们会这样：

```
// Java
interface Collection<E> …… {
  void addAll(Collection<E> items);
}
```

但随后，我们将无法做到以下简单的事情（这是完全安全）：

```
// 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，我们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 的一些子类型*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，该限制可以让 `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）**。

理解为什么这个技巧能够工作的关键相当简单：如果只能从集合中获取项目，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* 项目，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**。

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)`），当然如果调用函数返回 `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**，并称那些你只能**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-Extens，消费者-Super（Producer-Extends, Consumer-Super）。

注意：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()`。但这并不意味着该对象是**不可变的**：例如，没有什么阻止你调用 `clear()` 从列表中删除所有项目，因为 `clear()` 根本无需任何参数。通配符（或其他类型的型变）保证的唯一的事情是**类型安全**。不可变性完全是另一回事。

声明处型变

假设有一个泛型接口 `Source<T>`，该接口中不存在任何以 `T` 作为参数的方法，只是方法返回 `T` 类型值：

```
// Java
interface Source<T> {
  T nextT();
}
```

那么，在 `Source <Object>` 类型的变量中存储 `Source <String>` 实例的引用是极为安全的——没有消费者-方法可以调用。但是 Java 并不知道这一点，并且仍然禁止这样操作：

```
// Java
void demo(Source<String> strs) {
  Source<Object> objects = strs; // !!!在 Java 中不允许
  // ……
}
```

为了修正这一点，我们必须声明对象的类型为 `Source<? extends Object>`，这是毫无意义的，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**：我们可以标注 `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，我们提供 **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，他们说类 `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。你可以认为 `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，所以我们讲**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释：**in**。它使得一个类型参数**逆变**：只可以被消费而不可以被生产。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，我们可以将 x 赋给类型为 Comparable <Double> 的变量
    val y: Comparable<Double> = x // OK!
}
```

我们相信 **in** 和 **out** 两词是自解释的（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，并且可以将其改写为更高的目标：

[存在性（The Existential）](#) **转换：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* 非常方便，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) {
    fun get(index: Int): T { …… }
    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>` 和 `Array <Any>` 都不是另一个的子类型。为什么？再次重复，因为 copy **可能**做坏事，也就是说，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，一段时间后将会抛出一个 `ClassCastException` 异常。

那么，我们唯一要确保的是 `copy()` 不会做任何坏事。我们想阻止它**写**到 `from`，我们可以：

```
fun copy(from: Array<out Any>, to: Array<Any>) { …… }
```

这里发生的事情称为**类型投影**：我们说 `from` 不仅仅是一个数组，而是一个受限制的（**投影的**）数组：我们只可以调用返回类型为类型参数 `T` 的方法，如上，这意味着我们只能调用 `get()`。这就是我们的**使用处型变**的用法，并且是对应于 Java 的 `Array<? extends Object>`、但使用更简单些的方式。

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将是该投影的子类型。

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>`。这意味着当 `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>`。这意味着当 `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>`，我们可以想象以下星投影：

- `Function<*, String>` 表示 `Function<in Nothing, String>`；
- `Function<Int, *>` 表示 `Function<Int, out Any?>`；
- `Function<*, *>` 表示 `Function<in Nothing, out Any?>`。

注意：星投影非常像 Java 的原始类型，但是安全。

泛型函数

不仅类可以有类型参数。函数也可以有。类型参数要放在函数名称**之前**：

```
fun <T> singletonList(item: T): List<T> {
    // ……
}

fun <T> T.basicToString() : String {  // 扩展函数
    // ……
}
```

要调用泛型函数，在调用处函数名**之后**指定类型参数即可：

```
val l = singletonList<Int>(1)
```

可以省略能够从上下文中推断出来的类型参数，所以以下示例同样适用：

```
val l = singletonList(1)
```

泛型约束

能够替换给定类型参数的所有可能类型的集合可以由**泛型约束**限制。

上界

最常见的约束类型是与 Java 的 *extends* 关键字对应的 **上界**：

```
fun <T : Comparable<T>> sort(list: List<T>) {  …… }
```

冒号之后指定的类型是**上界**：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<HashMap<Int, String>> 的子类型
```

默认的上界（如果没有声明）是 `Any?`。在尖括号中只能指定一个上界。如果同一类型参数需要多个上界，我们需要一个单独的 **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() }
}
```

类型擦除

Kotlin 为泛型声明用法执行的类型安全检测仅在编译期进行。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
```

内部类

类可以标记为 `inner` 以便能够访问外部类的成员。内部类会带有一个对外部类的对象的引用：

```
class Outer {
    private val bar: Int = 1
    inner class Inner {
        fun foo() = bar
    }
}

val demo = Outer().Inner().foo() // == 1
```

参见限定的 `this` 表达式以了解内部类中的 `this` 的消歧义用法。

匿名内部类

使用对象表达式创建匿名内部类实例：

```
window.addMouseListener(object: MouseAdapter() {

    override fun mouseClicked(e: MouseEvent) { …… }

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

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，你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

枚举类的最基本的用法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。枚举常量用逗号分隔。

初始化

因为每一个枚举都是枚举类的实例，所以他们可以是这样初始化过的：

```
enum class Color(val rgb: Int) {
        RED(0xFF0000),
        GREEN(0x00FF00),
        BLUE(0x0000FF)
}
```

匿名类

枚举常量也可以声明自己的匿名类：

```
enum class ProtocolState {
    WAITING {
        override fun signal() = TALKING
    },

    TALKING {
        override fun signal() = WAITING
    };

    abstract fun signal(): ProtocolState
}
```

及相应的方法、以及覆盖基类的方法。注意，如果枚举类定义任何成员，要使用分号将成员定义中的枚举常量定义分隔开，就像在 Java 中一样。

枚举条目不能包含内部类以外的嵌套类型（已在 Kotlin 1.2 中弃用）。

在枚举类中实现接口

一个枚举类可以实现接口（但不能从类继承），可以为所有条目提供统一的接口成员实现，也可以在相应匿名类中为每个条目提供各自的实现。只需将接口添加到枚举类声明中即可，如下所示：

```
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)
}
```

使用枚举常量

就像在 Java 中一样，Kotlin 中的枚举类也有合成方法允许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()` 方法将抛出 `IllegalArgumentException` 异常。

自 Kotlin 1.1 起，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：

```
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。

对象表达式与对象声明

有时候，我们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。Java 用*匿名内部类*处理这种情况。Kotlin 用*对象表达式*和*对象声明*对这个概念稍微概括了下。

对象表达式

要创建一个继承自某个（或某些）类型的匿名类的对象，我们会这么写：

```
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，则必须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
}
```

任何时候，如果我们只需要“一个对象而已”，并不需要特殊超类型，那么我们可以简单地写：

```
fun foo() {
    val adHoc = object {
        var x: Int = 0
        var y: Int = 0
    }
    print(adHoc.x + adHoc.y)
}
```

请注意，匿名对象可以用作只在本地和私有作用域中声明的类型。如果你使用匿名对象作为公有函数的返回类型或者用作公有属性的类型，那么该函数或属性的实际类型会是匿名对象声明的超类型，如果你没有声明任何超类型，就会是 `Any`。在匿名对象中添加的成员将无法访问。

```
class C {
    // 私有函数,所以其返回类型是匿名对象类型
    private fun foo() = object {
        val x: String = "x"
    }

    // 公有函数,所以其返回类型是 Any
    fun publicFoo() = object {
        val x: String = "x"
    }

    fun bar() {
        val x1 = foo().x        // 没问题
        val x2 = publicFoo().x  // 错误:未能解析的引用“x”
    }
}
```

就像 Java 匿名内部类一样,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。(与 Java 不同的是,这不仅限于 final 变量。)

```
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(继 Scala 之后)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。

如需引用该对象,我们直接使用其名称即可:

```
DataProviderManager.registerDataProvider(……)
```

这些对象可以有超类型：

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

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

注意：对象声明不能在局部作用域（即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
```

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（无论是否命名）的引用：

```
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。

内联类

> 内联类仅在 Kotlin 1.3 之后版本可用，目前还是*实验性的*。关于详情请参见下文

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，它通过在类的前面定义一个 `inline` 修饰符来声明：

```
inline class Password(val value: String)
```

内联类必须含有唯一的一个属性在主构造函数中初始化。在运行时，将使用这个唯一属性来表示内联类的实例（关于运行时的内部表达请参阅下文）：

```
// 不存在 'Password' 类的真实实例对象
// 在运行时，'securePassword' 仅仅包含 'String'
val securePassword = Password("Don't try this in production")
```

这就是内联类的主要特性，它灵感来源于 “inline” 这个名称：类的数据被 “内联” 到该类使用的地方（类似于内联函数中的代码被内联到该函数调用的地方）。

成员

内联类支持普通类中的一些功能。特别是，内联类可以声明属性与函数：

```
inline class Name(val s: String) {
    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
}
```

然而，内联类的成员也有一些限制：

- 内联类不能含有 `init` 代码块
- 内联类不能含有 `inner` 类
- 内联类不能含有幕后字段
  - 因此，内联类只能含有简单的计算属性（不能含有延迟初始化/委托属性）

继承

内联类允许去继承接口

```
interface Printable {
    fun prettyPrint(): String
}

inline class Name(val s: String) : Printable {
    override fun prettyPrint(): String = "Let's $s!"
}

fun main() {
    val name = Name("Kotlin")
    println(name.prettyPrint()) // 仍然会作为一个静态方法被调用
}
```

禁止内联类参与到类的继承关系结构中。这就意味着内联类不能继承其他的类而且必须是 `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

inline class Foo(val i: Int) : I

fun asInline(f: Foo) {}
fun <T> asGeneric(x: T) {}
fun asInterface(i: I) {}
fun asNullable(i: Foo?) {}

fun <T> id(x: T): T = x

fun main() {
    val f = Foo(42)

    asInline(f)    // 拆箱操作: 用作 Foo 本身
    asGeneric(f)   // 装箱操作: 用作泛型类型 T
    asInterface(f) // 装箱操作: 用作类型 I
    asNullable(f)  // 装箱操作: 用作不同于 Foo 的可空类型 Foo?

    // 在下面这里例子中,'f' 首先会被装箱(当它作为参数传递给 'id' 函数时)然后又被拆箱(当它从'id'函数中被返回时)
    // 最后, 'c' 中就包含了被拆箱后的内部表达(也就是 '42'), 和 'f' 一样
    val c = id(f)
}
```

因为内联类既可以表示为基础类型有可以表示为包装器，引用相等对于内联类而言毫无意义，因此这也是被禁止的。

名字修饰

由于内联类被编译为其基础类型，因此可能会导致各种模糊的错误，例如意想不到的平台签名冲突：

```
inline 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。

> 请注意在 Java 中 - 是一个 *无效的* 符号，也就是说在 Java 中不能调用使用内联类作为形参的函数。

内联类与类型别名

初看起来，内联类似乎与类型别名非常相似。实际上，两者似乎都引入了一种新的类型，并且都在运行时表示为基础类型。

然而，关键的区别在于类型别名与其基础类型(以及具有相同基础类型的其他类型别名)是 *赋值兼容* 的，而内联类却不是这样。

换句话说，内联类引入了一个真实的新类型，与类型别名正好相反，类型别名仅仅是为现有的类型取了个新的替代名称(别名)：

```
typealias NameTypeAlias = String
inline 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("") // 正确：传递基础类型的实参替代函数中别名类型的形参
    acceptNameInlineClass("") // 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
}
```

内联类的实验性状态

内联类的设计目前是实验性的，这就是说此特性是正在 *快速变化*的，并且不保证其兼容性。在 Kotlin 1.3+ 中使用内联类时，将会得到一个警告，来表明此特性还是实验性的。

要想移除警告，你必须通过对 `kotlinc` 指定 `-XXLanguage:+InlineClasses` 参数来选择使用该实验性的特性。

在 Gradle 中启用内联类：

```
compileKotlin {
    kotlinOptions.freeCompilerArgs += ["-XXLanguage:+InlineClasses"]
}
```

关于详细信息，请参见编译器选项。关于多平台项目的设置，请参见使用 Gradle 构建多平台项目章节。

在 Maven 中启用内联类

```
<configuration>
    <args>
        <arg>-XXLanguage:+InlineClasses</arg>
    </args>
</configuration>
```

关于详细信息，请参见指定编译器选项。

进一步讨论

关于其他技术详细信息和讨论，请参见内联类的语言提议

委托

属性委托

属性委托在单独一页中讲：属性委托。

由委托实现

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”：

```
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)
}
```

委托属性

有一些常见的属性类型，虽然我们可以在每次需要的时候手动实现它们，但是如果能够为大家把他们只实现一次并放入一个库会更好。例如包括：

- 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任何的接口，但是需要提供一个 `getValue()` 函数（与 `setValue()`——对于 var 属性）。例如：

```
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.")
    }
}
```

当我们从委托到一个 `Delegate` 实例的 `p` 读取时，将调用 `Delegate` 中的 `getValue()` 函数，所以它第一个参数是读出 `p` 的对象、第二个参数保存了对 `p` 自身的描述（例如你可以取它的名字）。例如：

```
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。

请注意，自 Kotlin 1.1 起你可以在函数或代码块中声明一个委托属性，因此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

`lazy()` 是接受一个 lambda 并返回一个 `Lazy <T>` 实例的函数，返回的实例可以作为实现延迟属性的委托：第一次调用 `get()` 会执行已传递给 `lazy()` 的 lambda 表达式并记录结果，后续调用 `get()` 只是返回记录的结果。

```
val lazyValue: String by lazy {
    println("computed!")
    "Hello"
}

fun main() {
    println(lazyValue)
    println(lazyValue)
}
```

默认情况下，对于 lazy 属性的求值是**同步锁的（synchronized）**：该值只在一个线程中计算，并且所有线程会看到相同的值。如果初始化委托的同步锁不是必需的，这样多个线程可以同时执行，那么将 `LazyThreadSafetyMode.PUBLICATION` 作为参数传递给 `lazy()` 函数。而如果你确定初始化将总是发生在单个线程，那么你可以使用 `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

`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"
}
```

如果你想能够截获一个赋值并“否决”它，就使用 `vetoable()` 取代 `observable()`。在属性被赋新值生效*之前*会调用传递给 `vetoable` 的处理程序。

把属性储存在映射中

一个常见的用例是在一个映射（map）里存储属性的值。这经常出现在像解析 JSON 或者做其他“动态”事情的应用中。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
))
```

委托属性会从这个映射中取值（通过字符串键——属性的名称）：

```
println(user.name) // Prints "John Doe"
println(user.age)  // Prints 25
```

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（自 1.1 起）

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

这里我们总结了委托对象的要求。

对于一个**只读**属性(即 `val` 声明的),委托必须提供一个名为 `getValue` 的函数,该函数接受以下参数:

- `thisRef` —— 必须与 *属性所有者* 类型(对于扩展属性——指被扩展的类型)相同或者是它的超类型;
- `property` —— 必须是类型 `KProperty<*>` 或其超类型。

这个函数必须返回与属性相同的类型(或其子类型)。

对于一个**可变**属性(即 `var` 声明的),委托必须*额外*提供一个名为 `setValue` 的函数,该函数接受以下参数:

- `thisRef` —— 同 `getValue()`;
- `property` —— 同 `getValue()`;
- new value —— 必须与属性同类型或者是它的超类型。

`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。

委托类可以实现包含所需 `operator` 方法的 `ReadOnlyProperty` 或 `ReadWriteProperty` 接口之一。这俩接口是在 Kotlin 标准库中声明的:

```
interface ReadOnlyProperty<in R, out T> {
    operator fun getValue(thisRef: R, property: KProperty<*>): T
}

interface ReadWriteProperty<in R, T> {
    operator fun getValue(thisRef: R, property: KProperty<*>): T
    operator fun setValue(thisRef: R, property: KProperty<*>, value: T)
}
```

翻译规则

在每个委托属性的实现的背后,Kotlin 编译器都会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。

请注意，直接在代码中引用绑定的可调用引用的语法 `this::prop` 自 Kotlin 1.1 起才可用。

提供委托（自 1.1 起）

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。

`provideDelegate` 的一个可能的使用场景是在创建属性时（而不仅在其 getter 或 setter 中）检查属性一致性。

例如，如果要在绑定之前检查属性名称，可以这样写：

```
class ResourceDelegate<T> : ReadOnlyProperty<MyUI, T> {
    override fun getValue(thisRef: MyUI, property: KProperty<*>): T { ... }
}

class ResourceLoader<T>(id: ResourceID<T>) {
    operator fun provideDelegate(
            thisRef: MyUI,
            prop: KProperty<*>
    ): ReadOnlyProperty<MyUI, T> {
        checkProperty(thisRef, prop.name)
        // 创建委托
        return ResourceDelegate()
    }

    private fun checkProperty(thisRef: MyUI, name: String) { …… }
}

class MyUI {
    fun <T> bindResource(id: ResourceID<T>): ResourceLoader<T> { …… }

    val image by bindResource(ResourceID.image_id)
    val text by bindResource(ResourceID.text_id)
}
```

`provideDelegate` 的参数与 `getValue` 相同：

- `thisRef` —— 必须与 *属性所有者* 类型（对于扩展属性——指被扩展的类型）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：

```
// 检查属性名称而不使用“provideDelegate”功能
class MyUI {
    val image by bindResource(ResourceID.image_id, "image")
    val text by bindResource(ResourceID.text_id, "text")
}

fun <T> MyUI.bindResource(
        id: ResourceID<T>,
        propertyName: String
): ReadOnlyProperty<MyUI, T> {
   checkProperty(this, propertyName)
   // 创建委托
}
```

在生成的代码中，会调用 `provideDelegate` 方法来初始化辅助的 `prop$delegate` 属性。比较对于属性声明 `val prop: Type by MyDelegate()` 生成的代码与上面（当 `provideDelegate` 方法不存在时）生成的代码：

```
class C {
    var prop: Type by MyDelegate()
}

// 这段代码是当“provideDelegate”功能可用时
// 由编译器生成的代码：
class C {
    // 调用“provideDelegate”来创建额外的“delegate”属性
    private val prop$delegate = MyDelegate().provideDelegate(this, this::prop)
    var prop: Type
        get() = prop$delegate.getValue(this, this::prop)
        set(value: Type) = prop$delegate.setValue(this, this::prop, value)
}
```

请注意，`provideDelegate` 方法只影响辅助属性的创建，并不会影响为 getter 或 setter 生成的代码。

函数与 Lambda 表达式

函数

函数声明

Kotlin 中的函数使用 fun 关键字声明：

```
fun double(x: Int): Int {
    return 2 * x
}
```

函数用法

调用函数使用传统的方法：

```
val result = double(2)
```

调用成员函数使用点表示法：

```
Sample().foo() // 创建类 Sample 实例并调用 foo
```

参数

函数参数使用 Pascal 表示法定义，即 *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) { …… }
```

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。与其他语言相比，这可以减少重载数量：

```
fun read(b: Array<Byte>, off: Int = 0, len: Int = b.size) { …… }
```

默认值通过类型后面的 **=** 及给出的值来定义。

覆盖方法总是使用与基类型方法相同的默认参数值。当覆盖一个带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命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
```

不过如果最后一个 lambda 表达式参数从括号外传给函数函数调用，那么允许默认参数不传值：

```
fun foo(bar: Int = 0, baz: Int = 1, qux: () -> Unit) { …… }

foo(1) { println("hello") } // 使用默认值 baz = 1
foo { println("hello") }    // 使用两个默认值 bar = 0 与 baz = 1
```

命名参数

可以在调用函数时使用命名的函数参数。当一个函数有大量的参数或默认参数时这会非常方便。

给定以下函数：

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

我们可以使用默认参数来调用它：

```
reformat(str)
```

然而，当使用非默认参数调用它时，该调用看起来就像：

```
reformat(str, true, true, false, '_')
```

使用命名参数我们可以使代码更具有可读性：

```
reformat(str,
    normalizeCase = true,
    upperCaseFirstLetter = true,
    divideByCamelHumps = false,
    wordSeparator = '_'
)
```

并且如果我们不需要所有的参数：

```
reformat(str, wordSeparator = '_')
```

当一个函数调用混用位置参数与命名参数时，所有位置参数都要放在第一个命名参数之前。例如，允许调用 `f(1, y = 2)` 但不允许 `f(x = 1, 2)`。

可以通过使用**星号**操作符将可变数量参数（vararg）以命名形式传入：

```
fun foo(vararg strings: String) { …… }

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

请注意，在调用 Java 函数时不能使用命名参数语法，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

如果一个函数不返回任何有用的值，它的返回类型是 `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`，在这种情况下它是可选的。Kotlin 不推断具有块代码体的函数的返回类型，因为这样的函数在代码体中可能有复杂的控制流，并且返回类型对于读者（有时甚至对于编译器）是不明显的。

可变数量的参数（Varargs）

函数的参数（通常是最后一个）可以用 `vararg` 修饰符标记：

```
fun <T> asList(vararg ts: T): List<T> {
    val result = ArrayList<T>()
    for (t in ts) // ts is an Array
        result.add(t)
    return result
}
```

允许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，即上例中的 `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，可以使用命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。

当我们调用 `vararg`-函数时，我们可以一个接一个地传参，例如 `asList(1, 2, 3)`，或者，如果我们已经有一个数组并希望将其内容传给该函数，我们使用**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)
```

中缀表示法

标有 `infix` 关键字的函数也可以使用中缀表示法（忽略该调用的点与圆括号）调用。中缀函数必须满足以下要求：

- 它们必须是成员函数或扩展函数；
- 它们必须只有一个参数；
- 其参数不得接受可变数量的参数且不能有默认值。

```
infix fun Int.shl(x: Int): Int { …… }

// 用中缀表示法调用该函数
1 shl 2

// 等同于这样
1.shl(2)
```

中缀函数调用的优先级低于算术操作符、类型转换以及 rangeTo 操作符。以下表达式是等价的：

— `1 shl 2 + 3` 与 `1 shl (2 + 3)`

— `0 until n * 2` 与 `0 until (n * 2)`

— `xs union ys as Set<*>` 与 `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` 与 `a && (b xor c)`

— `a xor b in c` 与 `(a xor b) in c`

完整的优先级层次结构请参见其语法参考。

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`；不能像常规方法调用那样省略。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

在 Kotlin 中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# 或 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: Set<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（即闭包）的局部变量，所以在上例中，*visited* 可以是局部变量：

```
fun dfs(graph: Graph) {
    val visited = HashSet<Vertex>()
    fun dfs(current: Vertex) {
        if (!visited.add(current)) return
        for (v in current.neighbors)
            dfs(v)
    }

    dfs(graph.vertices[0])
}
```

成员函数

成员函数是在类或对象内部定义的函数：

```
class Sample() {
    fun foo() { print("Foo") }
}
```

成员函数以点表示法调用：

```
Sample().foo() // 创建类 Sample 实例并调用 foo
```

关于类和覆盖成员的更多信息参见类和继承。

泛型函数

函数可以有泛型参数，通过在函数名前使用尖括号指定：

```
fun <T> singletonList(item: T): List<T> { …… }
```

关于泛型函数的更多信息参见泛型。

内联函数

内联函数在这里讲述。

扩展函数

扩展函数在其自有章节讲述。

高阶函数和 Lambda 表达式

高阶函数和 Lambda 表达式在其自有章节讲述。

尾递归函数

Kotlin 支持一种称为尾递归的函数式编程风格。这允许一些通常用循环写的算法改用递归函数来写，而无堆栈溢出的风险。当一个函数用 `tailrec` 修饰符标记并满足所需的形式时，编译器会优化该递归，留下一个快速而高效的基于循环的版本：

```
val eps = 1E-10 // "good enough", could be 10^-15

tailrec fun findFixPoint(x: Double = 1.0): Double
        = if (Math.abs(x - Math.cos(x)) < eps) x else findFixPoint(Math.cos(x))
```

这段代码计算余弦的不动点（fixpoint of cosine），这是一个数学常数。它只是重复地从 1.0 开始调用 Math.cos，直到结果不再改变，对于这里指定的 `eps` 精度会产生 0.7390851332151611 的结果。最终代码相当于这种更传统风格的代码：

```
val eps = 1E-10 // "good enough", could be 10^-15

private fun findFixPoint(): Double {
    var x = 1.0
    while (true) {
        val y = Math.cos(x)
        if (Math.abs(x - y) < eps) return x
        x = Math.cos(x)
    }
}
```

要符合 `tailrec` 修饰符的条件的话，函数必须将其自身调用作为它执行的最后一个操作。在递归调用后有更多代码时，不能使用尾递归，并且不能用在 try/catch/finally 块中。目前尾部递归只在 JVM 后端中支持。

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、作为参数传递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操作函数。

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。

一个不错的示例是集合的函数式风格的 `fold`，它接受一个初始累积值与一个接合函数，并通过将当前累积值与每个集合元素连续接合起来代入累积值来构建返回值：

```
fun <T, R> Collection<T>.fold(
    initial: R,
    combine: (acc: R, nextElement: T) -> R
): R {
    var accumulator: R = initial
    for (element: T in this) {
        accumulator = combine(accumulator, element)
    }
    return accumulator
}
```

在上述代码中，参数 `combine` 具有函数类型 `(R, T) -> R`，因此 `fold` 接受一个函数作为参数，该函数接受类型分别为 `R` 与 `T` 的两个参数并返回一个 `R` 类型的值。在 `for`-循环内部调用该函数，然后将其返回值赋值给 `accumulator`。

为了调用 `fold`，需要传给它一个函数类型的实例作为参数，而在高阶函数调用处，（下文详述的）lambda 表达 式广泛用于此目的。

```
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)
```

以下各节会更详细地解释上文提到的这些概念。

函数类型

Kotlin 使用类似 `(Int) -> String` 的一系列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，即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。
- 挂起函数属于特殊种类的函数类型，它的表示法中有一个 `suspend` 修饰符，例如 `suspend () -> Unit` 或者 `suspend A.(B) -> C`。

函数类型表示法可以选择性地包含函数的参数名：`(x: Int, y: Int) -> Point`。这些名称可用于表明参数的含义。

> 如需将函数类型指定为可空，请使用圆括号：`((Int, Int) -> Int)?`。
>
> 函数类型可以使用圆括号进行接合：`(Int) -> ((Int) -> Unit)`
>
> 箭头表示法是右结合的，`(Int) -> (Int) -> Unit` 与前述示例等价，但不等于 `((Int) -> (Int)) -> Unit`。

还可以通过使用类型别名给函数类型起一个别称：

```
typealias ClickHandler = (Button, ClickEvent) -> Unit
```

函数类型实例化

有几种方法可以获得函数类型的实例：

- 使用函数字面值的代码块，采用以下形式之一：
  - lambda 表达式：`{ a, b -> a + b }`，
  - 匿名函数：`fun(s: String): Int { return s.toIntOrNull() ?: 0 }`

  带有接收者的函数字面值可用作带有接收者的函数类型的值。
- 使用已有声明的可调用引用：
  - 顶层、局部、成员、扩展函数：`::isOdd`、`String::toInt`，
  - 顶层、成员、扩展属性：`List<Int>::size`，

— 构造函数：`::Regex`

这包括指向特定实例成员的绑定的可调用引用：`foo::toString`。

— 使用实现函数类型接口的自定义类的实例：

```
class IntTransformer: (Int) -> Int {
    override operator fun invoke(x: Int): Int = TODO()
}

val intFunction: (Int) -> Int = IntTransformer()
```

如果有足够信息，编译器可以推断变量的函数类型：

```
val a = { i: Int -> i + 1 } // 推断出的类型是 (Int) -> Int
```

带与不带接收者的函数类型*非字面*值可以互换，其中接收者可以替代第一个参数，反之亦然。例如，`(A, B) -> C` 类型的值可以传给或赋值给期待 `A.(B) -> C` 的地方，反之亦然：

```
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
```

> 请注意，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：

```
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
```

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”，即未声明的函数，但立即做为表达式传递。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，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，即函数字面值，它等价于以下命名函数：

```
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 = { 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 `->` 符号之后。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。

如果我们把所有可选标注都留下，看起来如下：

```
val sum: (Int, Int) -> Int = { x, y -> x + y }
```

将 lambda 表达式传给最后一个参数

在 Kotlin 中有一个约定：如果函数的最后一个参数接受函数，那么作为相应参数传入的 lambda 表达式可以放在圆括号之外：

```
val product = items.fold(1) { acc, e -> acc * e }
```

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是很常见的。

如果编译器自己可以识别出签名，也可以不用声明唯一的参数并忽略 `->`。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

我们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.toUpperCase() }
```

下划线用于未使用的变量（自 1.1 起）

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（自 1.1 起）

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

上面提供的 lambda 表达式语法缺少的一个东西是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。其函数体可以是表达式（如上所示）或代码块：

```
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，而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`）。

请注意，匿名函数参数总是在括号内传递。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*，即在外部作用域中声明的变量。与 Java 不同的是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 提供了调用带有接收者（提供*接收者对象*）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，这将非常有用。

```
val sum = fun Int.(other: Int): Int = this + other
```

当接收者类型可以从上下文推断时，lambda 表达式可以用作带接收者的函数字面值。One of the most important examples of their usage is type-safe builders:

```
class HTML {
    fun body() { …… }
}

fun html(init: HTML.() -> Unit): HTML {
    val html = HTML()  // 创建接收者对象
    html.init()        // 将该接收者对象传给该 lambda
    return html
}

html {       // 带接收者的 lambda 由此开始
    body()   // 调用该接收者对象的一个方法
}
```

内联函数

使用高阶函数会带来一些运行时的效率损失：每一个函数都是一个对象，并且会捕获一个闭包。即那些在函数体内会访问到的变量。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。即 `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()
}
```

这个不是我们从一开始就想要的吗？

为了让编译器这么做，我们需要使用 `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；不过如果我们使用得当（即避免内联过大函数），性能上会有所提升，尤其是在循环中的“超多态（megamorphic）”调用处。

禁用内联

如果你只想被（作为参数）传给一个内联函数的 lamda 表达式中只有一些被内联，你可以用 `noinline` 修饰符标记一些函数参数：

```
inline fun foo(inlined: () -> Unit, noinline notInlined: () -> Unit) { …… }
```

可以内联的 lambda 表达式只能在内联函数内部调用或者作为可内联的参数传递，但是 `noinline` 的可以以任何我们喜欢的方式操作：存储在字段中、传送它等等。

需要注意的是，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，则可以用 `@Suppress("NOTHING_TO_INLINE")` 注解关掉该警告）。

非局部返回

在 Kotlin 中，我们可以只使用一个正常的、非限定的 `return` 来退出一个命名或匿名函数。这意味着要退出一个 lambda 表达式，我们必须使用一个标签，并且在 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 foo() {
    ordinaryFunction {
        return // 错误:不能使 `foo` 在此处返回
    }
}
```

但是如果 lambda 表达式传给的函数是内联的，该 return 也可以内联，所以它是允许的：

inline fun inlined(block: () -> Unit) { println("hi!") }

```
fun foo() {
    inlined {
        return // OK:该 lambda 表达式是内联的
    }
}
```

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。我们习惯了在循环中用这种结构，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。为了标识这种情况，该 lambda 表达式参数需要用 `crossinline` 修饰符标记：

```
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

有时候我们需要访问一个作为参数传给我们的一个类型：

```
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?
}
```

在这里我们向上遍历一棵树并且检查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)
```

我们真正想要的只是传一个类型给该函数，即像这样调用它：

```
treeNode.findParentOfType<MyTreeNode>()
```

为能够这么做，内联函数支持*具体化的类型参数*，于是我们可以这样写：

```
inline fun <reified T> TreeNode.findParentOfType(): T? {
    var p = parent
    while (p != null && p !is T) {
        p = p.parent
    }
    return p as T?
}
```

我们使用 `reified` 修饰符来限定类型参数，现在可以在函数内部访问它了，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` 现在都能用了。此外，我们还可以按照上面提到的方式调用它：`myTree.findParentOfType<MyTreeNodeType>()`。

虽然在许多情况下可能不需要反射，但我们仍然可以对一个具体化的类型参数使用它：

```
inline fun <reified T> membersOf() = T::class.members

fun main(s: Array<String>) {
    println(membersOf<StringBuilder>().joinToString("\n"))
}
```

普通的函数（未标记为内联函数的）不能有具体化参数。不具有运行时表示的类型（例如非具体化的类型参数或者类似于 `Nothing` 的虚构类型）不能用作具体化的类型参数的实参。

相关底层描述，请参见规范文档。

内联属性（自 1.1 起）

`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 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查其函数体。

多平台程序设计

Platform-Specific Declarations

> Multiplatform projects are an experimental feature in Kotlin 1.2 and 1.3. All of the language and tooling features described in this document are subject to change in future Kotlin versions.

One of the key capabilities of Kotlin's multiplatform code is a way for common code to depend on platform-specific declarations. In other languages, this can often be accomplished by building a set of interfaces in the common code and implementing these interfaces in platform-specific modules. However, this approach is not ideal in cases when you have a library on one of the platforms that implements the functionality you need, and you'd like to use the API of this library directly without extra wrappers. Also, it requires common declarations to be expressed as interfaces, which doesn't cover all possible cases.

As an alternative, Kotlin provides a mechanism of *expected and actual declarations*. With this mechanism, a common module can define *expected declarations*, and a platform module can provide *actual declarations* corresponding to the expected ones. To see how this works, let's look at an example first. This code is part of a common module:

```
package org.jetbrains.foo

expect class Foo(bar: String) {
    fun frob()
}

fun main() {
    Foo("Hello").frob()
}
```

And this is the corresponding JVM module:

```
package org.jetbrains.foo

actual class Foo actual constructor(val bar: String) {
    actual fun frob() {
        println("Frobbing the $bar")
    }
}
```

This illustrates several important points:

— An expected declaration in the common module and its actual counterparts always have exactly the same fully qualified name.
— An expected declaration is marked with the `expect` keyword; the actual declaration is marked with the `actual` keyword.
— All actual declarations that match any part of an expected declaration need to be marked as `actual`.
— Expected declarations never contain any implementation code.

Note that expected declarations are not restricted to interfaces and interface members. In this example, the expected class has a constructor and can be created directly from common code. You can apply the `expect` modifier to other declarations as well, including top-level declarations and annotations:

```
// Common
expect fun formatString(source: String, vararg args: Any): String

expect annotation class Test

// JVM
actual fun formatString(source: String, vararg args: Any) =
    String.format(source, *args)

actual typealias Test = org.junit.Test
```

The compiler ensures that every expected declaration has actual declarations in all platform modules that implement the corresponding common module, and reports an error if any actual declarations are missing. 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 common 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>
```

Building Multiplatform Projects with Gradle

> Multiplatform projects are an experimental feature in Kotlin 1.2 and 1.3. All of the language and tooling features described in this document are subject to change in future Kotlin versions.

This document describes how Kotlin multiplatform projects are configured and built using Gradle. Only Gradle versions 4.7 and above can be used, older Gradle versions are not supported.

Gradle Kotlin DSL support has not been implemented yet for multiplatform projects, it will be added in the future updates. Please use the Groovy DSL in the build scripts.

Setting up a Multiplatform Project

You can create a new multiplatform project in the IDE by selecting one of the multiplatform project templates in the New Project dialog under the "Kotlin" section.

For example, if you choose "Kotlin (Multiplatform Library)", a library project is created that has three targets, one for the JVM, one for JS, and one for the Native platform that you are using. These are configured in the `build.gradle` script in the following way:

```
plugins {
    id 'org.jetbrains.kotlin.multiplatform' version '1.3.11'
}

repositories {
    mavenCentral()
}

kotlin {
    targets {
        fromPreset(presets.jvm, 'jvm')
        fromPreset(presets.js, 'js')
        fromPreset(presets.mingwX64, 'mingw')
    }

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

The three targets are created from the presets that provide some default configuration. There are presets for each of the supported platforms.

The source sets and their dependencies are then configured as follows:

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

kotlin {
    targets { /* ... */ }

    sourceSets {
```

```
        commonMain {
            dependencies {
                implementation 'org.jetbrains.kotlin:kotlin-stdlib-common'
            }
        }
        commonTest {
            dependencies {
                implementation 'org.jetbrains.kotlin:kotlin-test-common'
                implementation 'org.jetbrains.kotlin:kotlin-test-annotations-common'
            }
        }
        jvmMain {
            dependencies {
                implementation 'org.jetbrains.kotlin:kotlin-stdlib-jdk8'
            }
        }
        jvmTest {
            dependencies {
                implementation 'org.jetbrains.kotlin:kotlin-test'
                implementation 'org.jetbrains.kotlin:kotlin-test-junit'
            }
        }
        jsMain { /* ... */ }
        jsTest { /* ... */ }
        mingwMain { /* ... */ }
        mingwTest { /* ... */ }
    }
}
```

These are the default source set names for the production and test sources for the targets configured above. The source sets `commonMain` and `commonTest` are included into production and test compilations, respectively, of all targets. Note that the dependencies for common source sets `commonMain` and `commonTest` are the common artifacts, and the platform libraries go to the source sets of the specific targets.

The details on project structure and the DSL can be found in the following sections.

Gradle Plugin

To setup a multiplatform project from scratch, first, apply the `kotlin-multiplatform` plugin to the project by adding the following to the `build.gradle` file:

```
plugins {
    id 'org.jetbrains.kotlin.multiplatform' version '1.3.11'
}
```

This creates the `kotlin` extension at the top level. You can then access it in the build script for:

— setting up the targets for multiple platforms (no targets are created by default);

— configuring the source sets and their dependencies;

Setting up Targets

A target is a part of the build responsible for compiling, testing, and packaging a piece of software aimed for one of the supported platforms.

As the platforms are different, targets are built in different ways as well and have various platform-specific settings. The Gradle plugin bundles a number of presets for the supported platforms. A preset can be used to create a target by just providing a name as follows:

```
kotlin {
    targets {
        fromPreset(presets.jvm, 'jvm6') // Create a JVM target by the name 'jvm6'

        fromPreset(presets.linuxX64, 'linux') {
            /* You can specify additional settings for the 'linux' target here */
        }
    }
}
```

Building a target requires compiling Kotlin once or multiple times. Each Kotlin compilation of a target may serve a different purpose (e.g. production code, tests) and incorporate different source sets. The compilations of a target may be accessed in the DSL, for example, to get the task names, dependency files and compilation outputs:

```
kotlin {
    targets {
        fromPreset(presets.jvm, 'jvm6') {
            def mainKotlinTaskName = compilations.main.compileKotlinTaskName
            def mainOutputs = compilations.main.output
            def testRuntimeClasspath = compilations.test.runtimeDependencyFiles
        }
    }
}
```

To modify the Kotlin compiler options of a compilation, use the compilation's task which can be found by its name:

```
kotlin {
    targets {
        fromPreset(presets.jvm, 'jvm8') {
            // Configure a single target's compilations (main and test)
            compilations.all {
                tasks[compileKotlinTaskName].kotlinOptions {
                    jvmTarget = '1.8'
                }
            }
        }

        /* ... */

        // Configure all compilations of all targets:
        all {
            compilations.all {
                tasks[compileKotlinTaskName].kotlinOptions {
                    allWarningsAsErrors = true
                }
            }
        }
    }
}
```

All of the targets may share some of the sources and may have platform-specific sources in their compilations as well. See Configuring source sets for details.

Some targets may require additional configuration. For Android and iOS examples, see the Multiplatform Project: iOS and Android tutorial.

Supported platforms

There are target presets that one can apply using `fromPreset(presets.<presetName>, '<targetName>')`, as described above, for the following target platforms:

- `jvm` for Kotlin/JVM. Note: `jvm` targets do not compile Java;
- `js` for Kotlin/JS;
- `android` for Android applications and libraries. Note that one of the Android Gradle plugins should be applied as well;
- Kotlin/Native target presets (see the notes below):
  - `androidNativeArm32` and `androidNativeArm64` for Android NDK;
  - `iosArm32`, `iosArm64`, `iosX64` for iOS;
  - `linuxArm32Hfp`, `linuxMips32`, `linuxMipsel32`, `linuxX64` for Linux;
  - `macosX64` for MacOS;
  - `mingwX64` for Windows;

    `wasm32` for WebAssembly.

Note that some of the Kotlin/Native targets require an appropriate host machine to build on.

Configuring source sets

A Kotlin source set is a collection of Kotlin sources, along with their resources, dependencies, and language settings, which may take part in Kotlin compilations of one or more targets.

If you apply a target preset, some source sets are created and configured by default. See Default Project Layout.

The source sets are configured within a `sourceSets { ... }` block of the `kotlin { ... }` extension:

```
kotlin {
    targets { /* ... */ }

    sourceSets {
        foo { /* ... */ } // create or configure a source set by the name 'foo'
        bar { /* ... */ }
    }
}
```

> Note: creating a source set does not link it to any target. Some source sets are predefined and thus compiled by default. However, custom source sets always need to be explicitly directed to the compilations. See: Connecting source sets.

The source set names are case-sensitive. When referring to a default source set, make sure the name prefix matches a target's name, for example, a source set `iosX64Main` for a target `iosX64`.

A source set by itself is platform-agnostic, but it can be considered platform-specific if it is only compiled for a single platform. A source set can, therefore, contain either common code shared between the platforms or platform-specific code.

To add Kotlin source directories and resources to a source set, use its `kotlin` and `resources` `SourceDirectorySet`s:

```
kotlin {
    sourceSets {
        commonMain {
            kotlin.srcDir('src')
            resources.srcDir('res')
        }
    }
}
```

Connecting source sets

Kotlin source sets may be connected with the 'depends on' relation, so that if a source set `foo` depends on a source set `bar` then:

— whenever `foo` is compiled for a certain target, `bar` takes part in that compilation as well and is also compiled into the same target binary form, such as JVM class files or JS code;

— the resources of `bar` are always processed and copied along with the resources of `foo`;

— sources of `foo` 'see' the declarations of `bar`, including the `internal` ones, and the dependencies of `bar`, even those specified as `implementation` dependencies;

— `foo` may contain platform-specific implementations for the expected declarations of `bar`;

— the language settings of `foo` and `bar` should be consistent;

Circular source set dependencies are prohibited.

The source sets DSL can be used to define these connections between the source sets:

```
kotlin {
    sourceSets {
        commonMain { /* ... */ }
        allJvm {
            dependsOn commonMain
            /* ... */
        }
    }
}
```

Custom source sets created in addition to the default ones should be explicitly included into the dependencies hierarchy to be able to use declarations from other source sets and, most importantly, to take part in compilations. Most often, they need a `dependsOn commonMain` or `dependsOn commonTest` statement, and some of the default platform-specific source sets should depend on the custom ones, directly or indirectly:

```
kotlin {
    targets {
        fromPreset(presets.mingwX64, 'windows')
        fromPreset(presets.linuxX64, 'linux')
        /* ... */
    }
    sourceSets {
        desktopTest { // custom source set with tests for the two targets
            dependsOn commonTest
            /* ... */
        }
        windowsTest { // default test source set for target 'windows'
            dependsOn desktopTest
            /* ... */
        }
        linuxTest { // default test source set for target 'linux'
            dependsOn desktopTest
        }
        /* ... */
    }
}
```

Adding Dependencies

To add a dependency to a source set, use a `dependencies { ... }` block of the source sets DSL. Four kinds of dependencies are supported:

— `api` dependencies are used both during compilation and at runtime and are exported to library consumers. If any types from a dependency are used in the public API of the current module, then it should be an `api` dependency;

— `implementation` dependencies are used during compilation and at runtime for the current module, but are not exposed for compilation of other modules depending on the one with the `implementation` dependency. The `implementation` dependency kind should be used for dependencies needed for the internal logic of a module. If a module is an endpoint application which is not published, it may use `implementation` dependencies instead of `api` ones.

— `compileOnly` dependencies are only used for compilation of the current module and are available neither at runtime nor during compilation of other modules. These dependencies should be used for APIs which have a third-party implementation available at runtime.

— `runtimeOnly` dependencies are available at runtime but are not visible during compilation of any module.

Dependencies are specified per source set as follows:

```
kotlin {
    sourceSets {
        commonMain {
            dependencies {
                api 'com.example:foo-metadata:1.0'
            }
        }
        jvm6Main {
            dependencies {
                api 'com.example:foo-jvm6:1.0'
            }
        }
    }
}
```

Note that for the IDE to correctly analyze the dependencies of the common sources, the common source sets need to have corresponding dependencies on the Kotlin metadata packages in addition to the platform-specific artifact dependencies of the platform-specific source sets. Usually, an artifact with a suffix `-common` (as in `kotlin-stdlib-common`) or `-metadata` is required when using a published library (unless it is published with Gradle metadata, as described below).

However, a `project('...')` dependency on another multiplatform project is resolved to an appropriate target automatically. It is enough to specify a single `project('...')` dependency in a source set's dependencies, and the compilations that include the source set will receive a corresponding platform-specific artifact of that project, given that it has a compatible target.

Likewise, if a multiplatform library is published in the experimental Gradle metadata publishing mode and the project is set up to consume the metadata as well, then it is enough to specify a dependency only once, for the common source set. Otherwise, each platform-specific source set should be provided with a corresponding platform module of the library, in addition to the common module, as shown above.

An alternative way to specify the dependencies is to use the Gradle built-in DSL at the top level with the configuration names following the pattern `<sourceSetName><DependencyKind>`:

```
dependencies {
    commonMainApi 'com.example:foo-common:1.0'
    jvm6MainApi 'com.example:foo-jvm6:1.0'
}
```

Language settings

The language settings for a source set can be specified as follows:

```
kotlin {
    sourceSets {
        commonMain {
            languageSettings {
                languageVersion = '1.3' // possible values: '1.0', '1.1', '1.2', '1.3'
                apiVersion = '1.3' // possible values: '1.0', '1.1', '1.2', '1.3'
                enableLanguageFeature('InlineClasses') // language feature name
                useExperimentalAnnotation('kotlin.ExperimentalUnsignedTypes') //
annotation FQ-name
                progressiveMode = true // false by default
            }
        }
    }
}
```

It is possible to configure the language settings of all source sets at once:

```
kotlin.sourceSets.all {
    languageSettings {
        progressiveMode = true
    }
}
```

Language settings of a source set affect how the sources are analyzed in the IDE. Due to the current limitations, in a Gradle build, only the language settings of the compilation's default source set are used.

The language settings are checked for consistency between source sets depending on each other. Namely, if `foo` depends on `bar`:

- `foo` should set `languageVersion` that is greater than or equal to that of `bar`;
- `foo` should enable all unstable language features that `bar` enables (there's no such requirement for bugfix features);
- `foo` should use all experimental annotations that `bar` uses;
- `apiVersion`, bugfix language features, and `progressiveMode` can be set arbitrarily;

Default Project Layout

By default, each project contains two source sets, `commonMain` and `commonTest`, where one can place all the code that should be shared between all of the target platforms. These source sets are added to each production and test compilation, respectively.

Then, once a target is added, default compilations are created for it:

- `main` and `test` compilations for JVM, JS, and Native targets;
- a compilation per Android variant, for Android targets;

For each compilation, there is a default source set under the name composed as `<targetName><CompilationName>`. This default source set participates in the compilation, and thus it should be used for the platform-specific code and dependencies, and for adding other source sets to the compilation by the means of 'depends on'. For example, a project with targets `jvm6` (JVM) and `nodeJs` (JS) will have source sets: `commonMain`, `commonTest`, `jvm6Main`, `jvm6Test`, `nodeJsMain`, `nodeJsTest`.

Numerous use cases are covered by just the default source sets and don't require custom source sets.

Each source set by default has its Kotlin sources under `src/<sourceSetName>/kotlin` directory and the resources under `src/<sourceSetName>/resources`.

In Android projects, additional Kotlin source sets are created for each Android source set. If the Android target has a name `foo`, the Android source set `bar` gets a Kotlin source set counterpart `fooBar`. The Kotlin compilations, however, are able to consume Kotlin sources from all of the directories `src/bar/java`, `src/bar/kotlin`, and `src/fooBar/kotlin`. Java sources are only read from the first of these directories.

Running Tests

Running tests in a Gradle build is currently supported by default for JVM, Android, Linux, Windows and macOS; JS and other Kotlin/Native targets need to be manually configured to run the tests with an appropriate environment, an emulator or a test framework.

A test task is created under the name `<targetName>Test` for each target that is suitable for testing. Run the `check` task to run the tests for all targets.

As the `commonTest` default source set is added to all test compilations, tests and test tools that are needed on all target platforms may be placed there.

The `kotlin.test` API is availble for multiplatform tests. Add the `kotlin-test-common` and `kotlin-test-annotations-common` dependencies to `commonTest` to use `DefaultAsserter` and `@Test` / `@Ignore` / `@BeforeTest` / `@AfterTest` annotations in the common tests.

For JVM targets, use `kotlin-test-junit` or `kotlin-test-testng` for the corresponding asserter implementation and annotations mapping.

For Kotlin/JS targets, add `kotlin-test-js` as a test dependency. At this point, test tasks for Kotlin/JS are created but do not run tests by default; they should be manually configured to run the tests with a JavaScript test framework.

Kotlin/Native targets do not require additional test dependencies, and the `kotlin.test` API implementations are built-in.

Publishing a Multiplatform Library

> The set of target platforms is defined by a multiplatform library author, and they should provide all of the platform-specific implementations for the library. Adding new targets for a multiplatform library at the consumer's side is not supported.

A library built from a multiplatform project may be published to a Maven repository with the Gradle `maven-publish` plugin, which can be applied as follows:

```
plugins {
    /* ... */
    id 'maven-publish'
}
```

Once this plugin is applied, default publications are created for each of the targets that can be built on the current host. This requires `group` and `version` to be set in the project. The default artifact IDs follow the pattern `<projectName>-<targetNameToLowerCase>`, for example `sample-lib-nodejs` for a target named `nodeJs` in a project `sample-lib`.

Also, an additional publication is added by default which contains serialized Kotlin declarations and is used by the IDE to analyze multiplatform libraries.

By default, a sources JAR is added to each publication in addition to its main artifact. The sources JAR contains the sources used by the `main` compilation of the target.

The Maven coordinates can be altered and additional artifact files may be added to the publication within the `targets { ... }` block:

```
kotlin {
    targets {
        fromPreset(presets.jvm, 'jvm6') {
            /* ... */
            mavenPublication {
                artifactId = 'sample-lib-jvm'
                artifact(jvmJavadocJar)
            }
        }
    }
}
```

Experimental metadata publishing mode

An experimental publishing and dependency consumption mode can be enabled by adding `enableFeaturePreview('GRADLE_METADATA')` to the root project's `settings.gradle` file. With Gradle metadata enabled, an additional publication is added which references the target publications as its variants. The artifact ID of this publication matches the project name.

> Gradle metadata publishing is an experimental Gradle feature which is not guaranteed to be backward-compatible. Future Gradle versions may fail to resolve a dependency to a library published with current versions of Gradle metadata. Library authors are recommended to use it to publish experimental versions of the library alongside with the stable publishing mechanism until the feature is considered stable.

If a library is published with Gradle metadata enabled and a consumer enables the metadata as well, the consumer may specify a single dependency on the library in a common source set, and a corresponding platform-specific variant will be chosen, if available, for each of the compilations. Consider a `sample-lib` library built for the JVM and JS and published with experimental Gradle metadata. Then it is enough for the consumers to add `enableFeaturePreview('GRADLE_METADATA)` and specify a single dependency:

```
kotlin {
    targets {
        fromPreset(presets.jvm, 'jvm6')
        fromPreset(presets.js, 'nodeJs')
    }
    sourceSets {
        commonMain {
            dependencies {
                api 'com.example:sample-lib:1.0'
                // is resolved to `sample-lib-jvm` for JVM, `sample-lib-js` for JS
            }
        }
    }
}
```

Disambiguating Targets

It is possible to have more than one target for a single platform in a multiplatform library. For example, these targets may provide the same API and differ in the libraries they cooperate with at runtime, like testing frameworks or logging solutions.

However, dependencies on such a multiplatform library may be ambiguous and may thus fail to resolve under certain conditions:

— A `project('...')` dependency on a multiplatform library is used. Replace it with a `project(path: '...', configuration: '...')` dependency. Use the appropriate target's runtime elements configuration, such as `jvm6RuntimeElements`. Due to the current limitations, this dependency should be placed in a top-level `dependencies { ... }` block rather than in a source set's dependencies.

— A published library dependency is used. If a library is published with experimental Gradle metadata, one can still replace the single dependency with unambiguous dependencies on its separate target modules, as if it had no experimental Gradle metadata.

— In both of the cases above, another solution is to mark the targets with a custom attribute. This, however, must be done on both the library author and the consumer sides, and it's the library author's responsibility to communicate the attribute and its values to the consumers;

Add the following symmetrically, to both the library and the consumer projects. The consumer may only need to mark a single target with the attribute:

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

   kotlin {
       targets {
           fromPreset(presets.jvm, 'junit') {
               attributes.attribute(testingFrameworkAttribute,
'junit')
           }
           fromPreset(presets.jvm, 'testng') {
               attributes.attribute(testingFrameworkAttribute,
'testng')
           }
       }
   }
```

Using Kotlin/Native Targets

It is important to note that some of the Kotlin/Native targets may only be built with an appropriate host machine:

— Linux targets may only be built on a Linux host;

— Windows targets require a Windows host;

— macOS and iOS targets can only be built on a macOS host;

— Android Native targets require a Linux or macOS host;

A target that is not supported by the current host is ignored during build and therefore not published. A library author may want to set up builds and publishing from different hosts as required by the library target platforms.

Kotlin/Native output kinds

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 run or used as a native library.

To link a binary in addition to the Kotlin/Native library, add one or more of the `outputKinds`, which can be:

— `executable` for an executable program;

— `dynamic` for a dynamic library;

— `static` for a static library;

— `framework` for an Objective-C framework (only supported for macOS and iOS targets)

This can be done as follows:

```
kotlin {
    targets {
        fromPreset(presets.linuxX64, 'linux') {
            compilations.main.outputKinds 'executable' // could be 'static', 'dynamic'
        }
    }
}
```

This creates additional link tasks for the debug and release binaries. The tasks can be accessed after project evaluation from the compilation as, for example, `getLinkTask('executable', 'release')` or `getLinkTaskName('static', 'debug')`. To get the binary file, use `getBinary`, for example, as `getBinary('executable', 'release')` or `getBinary('static', 'debug')`.

CInterop support

Since Kotlin/Native provides interoperability with native languages, there is a DSL allowing one to configure this feature for a specific compilation.

A compilation can interact with several native libraries. Interoperability with each of them can be configured in the `cinterops` block of the compilation:

```
// In the scope of a Kotlin/Native target's compilation:
cinterops {
    myInterop {
        // 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 to look for headers.
        includeDirs {
            // Directories for header search (an analogue of the -I<path> compiler
option).
            allHeaders 'path1', 'path2'

            // Additional directories to search headers listed in the 'headerFilter' def-
file option.
            // -headerFilterAdditionalSearchPrefix command line option analogue.
            headerFilterOnly 'path1', 'path2'
        }
        // A shortcut for includeDirs.allHeaders.
        includeDirs "include/directory", "another/directory"
    }

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

Often it's necessary to specify target-specific linker options for a binary which uses a native library. It can be done using the `linkerOpts` DSL method of a Kotlin/Native compilation:

```
compilations.main {
    linkerOpts '-L/lib/search/path -L/another/search/path -lmylib'
}
```

其他

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。我们已经声明了两个新变量：`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* 的另一个例子。（参见像 `+` 和 `*`、`for`-循环等操作符）。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。

请注意，`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：当你写：

```
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：从函数中返回两个变量

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。

注意: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:解构声明和映射

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().iterator()
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(自 1.1 起)

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(自 1.1 起)

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!" }
```

集合：List、Set、Map

与大多数语言不同，Kotlin 区分可变集合与不可变集合（lists、sets、maps 等）。精确控制什么时候集合可编辑有助于消除 bug 以及设计良好的 API。

预先了解一个可变集合的只读 *视图* 与一个真正的不可变集合之间的区别是很重要的。它们都容易创建，但类型系统不能表达它们的差别，所以由你来跟踪（是否相关）。

Kotlin 的 `List<out T>` 类型是一个提供只读操作如 `size`、`get` 等的接口。与 Java 类似，它继承自 `Collection<T>` 进而继承自 `Iterable<T>`。改变 list 的方法是由 `MutableList<T>` 加入的。这一模式同样适用于 `Set<out T>/MutableSet<T>` 及 `Map<K, out V>/MutableMap<K, V>`。

我们可以看下 list 及 set 类型的基本用法：

```
val numbers: MutableList<Int> = mutableListOf(1, 2, 3)
val readOnlyView: List<Int> = numbers
println(numbers)        // 输出 "[1, 2, 3]"
numbers.add(4)
println(readOnlyView)   // 输出 "[1, 2, 3, 4]"
readOnlyView.clear()    // -> 不能编译

val strings = hashSetOf("a", "b", "c", "c")
assert(strings.size == 3)
```

Kotlin 没有专门的语法结构创建 list 或 set。要用标准库的方法，如 `listOf()`、`mutableListOf()`、`setOf()`、`mutableSetOf()`。在非性能关键代码中创建 map 可以用一个简单的惯用法来完成：`mapOf(a to b, c to d)`。

注意上面的 `readOnlyView` 变量（译者注：与对应可变集合变量 `numbers`）指向相同的底层 list 并会随之改变。如果一个 list 只存在只读引用，我们可以考虑该集合完全不可变。创建一个这样的集合的一个简单方式如下：

```
val items = listOf(1, 2, 3)
```

目前 `listOf` 方法是使用 array list 实现的，但是未来可以利用它们知道自己不能变的事实，返回更节约内存的完全不可变的集合类型。

注意这些类型是协变的。这意味着，你可以把一个 `List<Rectangle>` 赋值给 `List<Shape>` 假定 `Rectangle` 继承自 `Shape`（集合类型与元素类型具有相同的继承关系）。对于可变集合类型这是不允许的，因为这将导致运行时故障：你可能向 `List<Shape>` 中添加一个 `Circle`，而在程序的其他地方创建了一个其中含有 `Circle` 的 List`。

有时你想给调用者返回一个集合在某个特定时间的一个快照，一个保证不会变的：

```
class Controller {
    private val _items = mutableListOf<String>()
    val items: List<String> get() = _items.toList()
}
```

这个 `toList` 扩展方法只是复制列表项，因此返回的 list 保证永远不会改变。

List 与 set 有很多有用的扩展方法值得熟悉：

```
val items = listOf(1, 2, 3, 4)
items.first() == 1
items.last() == 4
items.filter { it % 2 == 0 }   // 返回 [2, 4]

val rwList = mutableListOf(1, 2, 3)
rwList.requireNoNulls()        // 返回 [1, 2, 3]
if (rwList.none { it > 6 }) println("No items above 6")  // 输出“No items above 6”
val item = rwList.firstOrNull()
```

…… 以及所有你所期望的实用工具，例如 sort、zip、fold、reduce 等等。

非常值得注意的是，对只读集合返回修改后的集合的操作（例如 `+`、`filter`、`drop` 等）并不会以原子方式创建其结果，因此如果没有合适的同步机制，在不同的线程中使用其结果是不安全的。

Map 遵循同样模式。它们可以容易地实例化与访问，像这样：

```
val readWriteMap = hashMapOf("foo" to 1, "bar" to 2)
println(readWriteMap["foo"])  // 输出“1”
val snapshot: Map<String, Int> = HashMap(readWriteMap)
```

区间

区间表达式由具有操作符形式 `..` 的 `rangeTo` 函数辅以 `in` 和 `!in` 形成。区间是为任何可比较类型定义的，但对于整型原生类型，它有一个优化的实现。以下是使用区间的一些示例：

```
if (i in 1..10) { // 等同于 1 <= i && i <= 10
    println(i)
}
```

整型区间（`IntRange`、`LongRange`、`CharRange`）有一个额外的特性：它们可以迭代。编译器负责将其转换为类似 Java 的基于索引的 `for`-循环而无额外开销：

```
for (i in 1..4) print(i)
```

如果你想倒序迭代数字呢？也很简单。你可以使用标准库中定义的 `downTo()` 函数：

```
for (i in 4 downTo 1) print(i)
```

能否以不等于 1 的任意步长迭代数字？当然没问题，`step()` 函数有助于此：

```
for (i in 1..4 step 2) print(i)

for (i in 4 downTo 1 step 2) print(i)
```

要创建一个不包括其结束元素的区间，可以使用 `until` 函数：

```
for (i in 1 until 10) {
    // i in [1, 10) 排除了 10
    println(i)
}
```

它是如何工作的

区间实现了该库中的一个公共接口：`ClosedRange<T>`。

`ClosedRange<T>` 在数学意义上表示一个闭区间，它是为可比较类型定义的。它有两个端点：`start` 和 `endInclusive` 他们都包含在区间内。其主要操作是 `contains`，通常以 `in`/`!in` 操作符形式使用。

整型数列（`IntProgression`、`LongProgression`、`CharProgression`）表示等差数列。数列由 `first` 元素、`last` 元素和非零的 `step` 定义。第一个元素是 `first`，后续元素是前一个元素加上 `step`。`last` 元素总会被迭代命中，除非该数列是空的。

数列是 `Iterable<N>` 的子类型，其中 `N` 分别为 `Int`、`Long` 或者 `Char`，所以它可用于 `for`-循环以及像 `map`、`filter` 等函数中。对 `Progression` 迭代相当于 Java/JavaScript 的基于索引的 `for`-循环：

```
for (int i = first; i != last; i += step) {
  // ……
}
```

对于整型类型，`..` 操作符创建一个同时实现 `ClosedRange<T>` 和 `*Progression` 的对象。例如，`IntRange` 实现了 `ClosedRange<Int>` 并扩展自 `IntProgression`，因此为 `IntProgression` 定义的所有操作也可用于 `IntRange`。`downTo()` 和 `step()` 函数的结果总是一个 `*Progression`。

数列由在其伴生对象中定义的 `fromClosedRange` 函数构造：

```
IntProgression.fromClosedRange(start, end, step)
```

数列的 `last` 元素这样计算：对于正的 `step` 找到不大于 `end` 值的最大值、或者对于负的 `step` 找到不小于 `end` 值的最小值，使得 `(last - first) % increment == 0`。

一些实用函数

rangeTo()

整型类型的 `rangeTo()` 操作符只是调用 `*Range` 类的构造函数，例如：

```
class Int {
    //……
    operator fun rangeTo(other: Long): LongRange = LongRange(this, other)
    //……
    operator fun rangeTo(other: Int): IntRange = IntRange(this, other)
    //……
}
```

浮点数（`Double`、`Float`）未定义它们的 `rangeTo` 操作符，而使用标准库提供的泛型 `Comparable` 类型的操作符：

```
    public operator fun <T: Comparable<T>> T.rangeTo(that: T): ClosedRange<T>
```

该函数返回的区间不能用于迭代。

downTo()

扩展函数 `downTo()` 是为任何整型类型对定义的，这里有两个例子：

```
fun Long.downTo(other: Int): LongProgression {
    return LongProgression.fromClosedRange(this, other.toLong(), -1L)
}

fun Byte.downTo(other: Int): IntProgression {
    return IntProgression.fromClosedRange(this.toInt(), other, -1)
}
```

reversed()

扩展函数 `reversed()` 是为每个 `*Progression` 类定义的，并且所有这些函数返回反转后的数列：

```
fun IntProgression.reversed(): IntProgression {
    return IntProgression.fromClosedRange(last, first, -step)
}
```

step()

扩展函数 `step()` 是为每个 `*Progression` 类定义的，所有这些函数都返回带有修改了 `step` 值（函数参数）的数列。步长（step）值必须始终为正，因此该函数不会更改迭代的方向：

```
fun IntProgression.step(step: Int): IntProgression {
    if (step <= 0) throw IllegalArgumentException("Step must be positive, was: $step")
    return IntProgression.fromClosedRange(first, last, if (this.step > 0) step else -step)
}

fun CharProgression.step(step: Int): CharProgression {
    if (step <= 0) throw IllegalArgumentException("Step must be positive, was: $step")
    return CharProgression.fromClosedRange(first, last, if (this.step > 0) step else -
step)
}
```

请注意，返回数列的 `last` 值可能与原始数列的 `last` 值不同，以便保持不变式 `(last - first) % step == 0` 成立。这里是一个例子：

```
(1..12 step 2).last == 11  // 值为 [1, 3, 5, 7, 9, 11] 的数列
(1..12 step 3).last == 10  // 值为 [1, 4, 7, 10] 的数列
(1..12 step 4).last == 9   // 值为 [1, 5, 9] 的数列
```

类型的检查与转换“is”与“as”

`is` 与 `!is` 操作符

我们可以在运行时通过使用 `is` 操作符或其否定形式 `!is` 来检查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

在许多情况下，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`-检查以及显式转换，并在需要时自动插入（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查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 `&&` 和 `||` 的右侧：

```
    // `||` 右侧的 x 自动转换为字符串
    if (x !is String || x.length == 0) return

    // `&&` 右侧的 x 自动转换为字符串
    if (x is String && x.length > 0) {
        print(x.length) // x 自动转换为字符串
    }
```

这些 *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，当编译器不能保证变量在检查和使用之间不可改变时，智能转换不能用。更具体地，智能转换能否适用根据以下规则：

- `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，或者该检查在声明属性的同一模块中执行。智能转换不适用于 open 的属性或者具有自定义 getter 的属性；
- `var` 局部变量——如果变量在检查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，我们称之为*不安全的*。Kotlin 中的不安全转换由中缀操作符 `as`（参见operator precedence）完成：

```
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，即如果 `y` 为空，上面的代码会抛出一个异常。为了匹配 Java 转换语义，我们必须在转换右边有可空类型，就像：

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

“安全的”（可空）转换操作符

为了避免抛出异常，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，泛型类型的实例并无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。

为此，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。例如：

```
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>>() // 破坏类型安全!
```

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，并且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` 的映射保存到该文件
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>
```

编译器会对最后一行的类型转换产生一个警告。该类型转换不能在运行时完全检测，并且不能保证映射中的值是“Int”。

为避免未受检类型转换，可以重新设计程序结构：在上例中，可以使用具有类型安全实现的不同接口 `DictionaryReader<T>` 与 `DictionaryWriter<T>`。可以引入合理的抽象，将未受检的类型转换从调用代码移动到实现细节中。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诸如 `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
```

在 JVM 平台中，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。

This 表达式

为了表示当前的 *接收者* 我们使用 `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）我们使用`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
            }
        }
    }
}
```

相等性

Kotlin 中有两种类型的相等性：

— 结构相等（用 `equals()` 检查）；

— 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。按照惯例，像 `a == b` 这样的表达式会翻译成：

```
a?.equals(b) ?: (b === null)
```

也就是说如果 `a` 不是 `null` 则调用 `equals(Any?)` 函数，否则（即 `a` 是 `null`）检查 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`]https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/-any/equals.html) 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。

浮点数相等性

当相等性检测的两个操作数都是静态已知的（可空或非空的）`Float` 或 `Double` 类型时，该检测遵循 IEEE 754 浮点数运算标准。

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。

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。对于运行时表示为原生类型的值（例如 `Int`），`===` 相等检测等价于 `==` 检测。

操作符重载

Kotlin 允许我们为自己的类型提供预定义的一组操作符的实现。这些操作符具有固定的符号表示（如 `+` 或 `*`）和固定的优先级。为实现这样的操作符，我们为相应的类型（即二元操作符左侧的类型和一元操作符的参数类型）提供了一个固定名字的成员函数或扩展函数。重载操作符的函数需要用 `operator` 修饰符标记。

另外，我们描述为不同操作符规范操作符重载的约定。

一元操作

一元前缀操作符

表达式	翻译为
+a	a.unaryPlus()
-a	a.unaryMinus()
!a	a.not()

这个表是说，当编译器处理例如表达式 `+a` 时，它执行以下步骤：

- 确定 `a` 的类型，令其为 `T`；
- 为接收者 `T` 查找一个带有 `operator` 修饰符的无参函数 `unaryPlus()`，即成员函数或扩展函数；
- 如果函数不存在或不明确，则导致编译错误；
- 如果函数存在且其返回类型为 `R`，那就表达式 `+a` 具有类型 `R`；

注意 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()`；

— 检查函数的返回类型是 `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 `a.inc()` 结果赋值给 `a`；

— 把 `a0` 作为表达式的结果返回。

对于 `a--`，步骤是完全类似的。

对于*前缀*形式 `++a` 和 `--a` 以相同方式解析，其步骤是：

— 把 `a.inc()` 结果赋值给 `a`；

— 把 `a` 的新值作为表达式结果返回。

二元操作

算术运算符

表达式	翻译为
`a + b`	`a.plus(b)`
`a - b`	`a.minus(b)`
`a * b`	`a.times(b)`
`a / b`	`a.div(b)`
`a % b`	`a.rem(b)`、`a.mod(b)`（已弃用）
`a..b`	`a.rangeTo(b)`

对于此表中的操作，编译器只是解析成*翻译为*列中的表达式。

请注意，自 Kotlin 1.1 起支持 `rem` 运算符。Kotlin 1.0 使用 `mod` 运算符，它在 Kotlin 1.1 中被弃用。

示例

下面是一个从给定值起始的 Counter 类的示例，它可以使用重载的 `+` 运算符来增加计数：

```
data class Counter(val dayIndex: Int) {
    operator fun plus(increment: Int): Counter {
        return Counter(dayIndex + increment)
    }
}
```

“In”操作符

表达式	翻译为
`a in b`	`b.contains(a)`
`a !in b`	`!b.contains(a)`

对于 `in` 和 `!in`，过程是相同的，但是参数的顺序是相反的。

索引访问操作符

表达式	翻译为
`a[i]`	`a.get(i)`
`a[i, j]`	`a.get(i, j)`
`a[i_1, ……, i_n]`	`a.get(i_1, ……, i_n)`
`a[i] = b`	`a.set(i, b)`
`a[i, j] = b`	`a.set(i, j, b)`
`a[i_1, ……, i_n] = b`	`a.set(i_1, ……, i_n, b)`

方括号转换为调用带有适当数量参数的 `get` 和 `set`。

调用操作符

表达式	翻译为
`a()`	`a.invoke()`
`a(i)`	`a.invoke(i)`
`a(i, j)`	`a.invoke(i, j)`
`a(i_1, ……, i_n)`	`a.invoke(i_1, ……, i_n)`

圆括号转换为调用带有适当数量参数的 `invoke`。

广义赋值

表达式	翻译为
`a += b`	`a.plusAssign(b)`
`a -= b`	`a.minusAssign(b)`
`a *= b`	`a.timesAssign(b)`
`a /= b`	`a.divAssign(b)`
`a %= b`	`a.remAssign(b)`,`a.modAssign(b)`(已弃用)

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,那么报告错误(模糊),
  - 确保其返回类型是 `Unit`,否则报告错误,
  - 生成 `a.plusAssign(b)` 的代码;
- 否则试着生成 `a = a + b` 的代码(这里包含类型检查:`a + b` 的类型必须是 `a` 的子类型)。

注意:赋值在 Kotlin 中*不是*表达式。

相等与不等操作符

表达式	翻译为
`a == b`	`a?.equals(b) ?: (b === null)`
`a != b`	`!(a?.equals(b) ?: (b === null))`

这些操作符只使用函数 `equals(other: Any?): Boolean`,可以覆盖它来提供自定义的相等性检测实现。不会调用任何其他同名函数(如 `equals(other: Foo)`)。

注意:`===` 和 `!==`(同一性检查)不可重载,因此不存在对他们的约定。

这个 `==` 操作符有些特殊:它被翻译成一个复杂的表达式,用于筛选 `null` 值。`null == null` 总是 true,对于非空的 `x`,`x == null` 总是 false 而不会调用 `x.equals()`。

比较操作符

表达式	翻译为
`a > b`	`a.compareTo(b) > 0`
`a < b`	`a.compareTo(b) < 0`
`a >= b`	`a.compareTo(b) >= 0`
`a <= b`	`a.compareTo(b) <= 0`

所有的比较都转换为对 `compareTo` 的调用,这个函数需要返回 `Int` 值

属性委托操作符

`provideDelegate`、`getValue` 以及 `setValue` 操作符函数已在委托属性中描述。

命名函数的中缀调用

我们可以通过中缀函数的调用 来模拟自定义中缀操作符。

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 的类型系统旨在从我们的代码中消除 `NullPointerException`。NPE 的唯一可能的原因可能是：

- 显式调用 `throw NullPointerException()`；
- 使用了下文描述的 `!!` 操作符；
- 有些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；
  - 用于具有错误可空性的 Java 互操作的泛型类型，例如一段 Java 代码可能会向 Kotlin 的 `MutableList<String>` 中加入 `null`，这意味着应该使用 `MutableList<String?>` 来处理它；
  - 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`：

```
var a: String = "abc"
a = null // 编译错误
```

如果要允许为空，我们可以声明一个变量为可空字符串，写作 `String?`：

```
var b: String? = "abc"
b = null // ok
print(b)
```

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`，这样你就可以放心地使用：

```
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
```

但是我们还是需要访问该属性，对吧?有几种方式可以做到。

在条件中检查 null

首先，你可以显式检查 `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查的信息，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：

```
val b = "Kotlin"
if (b != null && b.length > 0) {
    print("String of length ${b.length}")
} else {
    print("Empty string")
}
```

请注意，这只适用于 `b` 是不可变的情况（即在检查和使用之间没有修改过的局部变量，或者不可覆盖并且有幕后字段的 `val` 成员），因为否则可能会发生在检查之后 `b` 又变为 `null` 的情况。

安全的调用

你的第二个选择是安全调用操作符，写作 `?.`：

```
val a = "Kotlin"
val b: String? = null
println(b?.length)
println(a?.length)
```

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，如果一个员工 Bob 可能会（或者不会）分配给一个部门，并且可能有另外一个员工是该部门的负责人，那么获取 Bob 所在部门负责人（如果有的话）的名字，我们写作：

```
bob?.department?.head?.name
```

如果任意一个属性（环节）为空，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：

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

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空都会跳过赋值，而右侧的表达式根本不会求值：

```
// 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

当我们有一个可空的引用 `r` 时，我们可以说“如果 `r` 非空，我使用它；否则使用某个非空的值 `x`”：

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

除了完整的 `if`-表达式，这还可以通过 Elvis 操作符表达，写作 `?:`：

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

如果 `?:` 左侧表达式非空，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，当且仅当左侧为空时，才会对右侧表达式求值。

请注意，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。这可能会非常方便，例如，检查函数参数：

```
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，若该值为空则抛出异常。我们可以写 `b!!`，这会返回一个非空的 `b` 值（例如：在我们例子中的 `String`）或者如果 `b` 为空，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()
```

异常

异常类

Kotlin 中所有异常类都是 `Throwable` 类的子孙类。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：

```
throw Exception("Hi There!")
```

使用 `try`-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` 块至少应该存在一个。

Try 是一个表达式

`try` 是一个表达式，即它可以有一个返回值：

```
val a: Int? = try { parseInt(input) } 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。

受检的异常

Kotlin 没有受检的异常。这其中有很多原因，但我们会提供一个简单的例子。

以下是 JDK 中 `StringBuilder` 类实现的一个示例接口：

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

这个签名是什么意思？它是说，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）上时我就必须捕获那些 `IOException`。为什么？因为它可能正在执行 IO 操作（`Writer` 也实现了 `Appendable`）…… 所以它导致这种代码随处可见的出现：

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

这并不好，参见《Effective Java》第三版 第 77 条：*不要忽略异常*。

Bruce Eckel 在《Java 是否需要受检的异常?》(Does Java need Checked Exceptions?) 中指出：

> 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。

其他相关引证：

— 《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)

Nothing 类型

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` 表达式的类型是特殊类型 `Nothing`。该类型没有值，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，编译器会知道执行不会超出该调用：

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

可能会遇到这个类型的另一种情况是类型推断。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。

注解

注解声明

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、表达式等）；
- `@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.VALUE_PARAMETER, AnnotationTarget.EXPRESSION)
@Retention(AnnotationRetention.SOURCE)
@MustBeDocumented
annotation class Fancy
```

用法

```
@Fancy class Foo {
    @Fancy fun baz(@Fancy foo: Int): Int {
        return (@Fancy 1)
    }
}
```

如果需要对类的主构造函数进行标注，则需要在构造函数声明中添加 `constructor` 关键字，并将注解添加到其前面：

```
class Foo @Inject constructor(dependency: MyDependency) { …… }
```

你也可以标注属性访问器：

```
class Foo {
    var x: MyDependency? = null
        @Inject set
}
```

构造函数

注解可以有接受参数的构造函数。

```
annotation class Special(val why: String)

@Special("example") class Foo {}
```

允许的参数类型有：

- 对应于 Java 原生类型的类型（Int、Long等）；
- 字符串；
- 类（`Foo::class`）；
- 枚举；
- 其他注解；
- 上面已列类型的数组。

注解参数不能有可空类型，因为 JVM 不支持将 `null` 作为注解属性的值存储。

如果注解用作另一个注解的参数，则其名称不以 @ 字符为前缀：

```
annotation class ReplaceWith(val expression: String)

annotation class Deprecated(
        val message: String,
        val replaceWith: ReplaceWith = ReplaceWith(""))

@Deprecated("This function is deprecated, use === instead", ReplaceWith("this === other"))
```

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 代码能够正常看到该注解及参数 。

```

import kotlin.reflect.KClass

annotation class Ann(val arg1: KClass<*>, val arg2: KClass<out Any>)

@Ann(String::class, Int::class) class MyClass
```

Lambda 表达式

注解也可以用于 lambda 表达式。它们会被应用于生成 lambda 表达式体的 `invoke()` 方法上。这对于像 Quasar 这样的框架很有用，该框架使用注解进行并发控制。

```
annotation class Suspendable

val f = @Suspendable { Fiber.sleep(10) }
```

注解使用处目标

当对属性或主构造函数参数进行标注时，从相应的 Kotlin 元素生成的 Java 元素会有多个，因此在生成的 Java 字节码中该注解有多个可能位置 。如果要指定精确地指定应该如何生成该注解，请使用以下语法：

```
class Example(@field:Ann val foo,    // 标注 Java 字段
              @get:Ann val bar,      // 标注 Java getter
              @param:Ann val quux)   // 标注 Java 构造函数参数
```

可以使用相同的语法来标注整个文件。要做到这一点，把带有目标 `file` 的注解放在文件的顶层、package 指令之前或者在所有导入之前（如果文件在默认包中的话）：

```
@file:JvmName("Foo")

package org.jetbrains.demo
```

如果你对同一目标有多个注解，那么可以这样来避免目标重复——在目标后面添加方括号并将所有注解放在方括号内：

```
class Example {
    @set:[Inject VisibleForTesting]
    var collaborator: Collaborator
}
```

支持的使用处目标的完整列表为：

- `file`；
- `property`（具有此目标的注解对 Java 不可见）；
- `field`；
- `get`（属性 getter）；
- `set`（属性 setter）；
- `receiver`（扩展函数或属性的接收者参数）；
- `param`（构造函数参数）；
- `setparam`（属性 setter 参数）；
- `delegate`（为委托属性存储其委托实例的字段）。

要标注扩展函数的接收者参数，请使用以下语法：

```
fun @receiver:Fancy String.myExtension() { ... }
```

如果不指定使用处目标，则根据正在使用的注解的 `@Target` 注解来选择目标 。如果有多个适用的目标，则使用以下列表中的第一个适用目标：

- `param`；
- `property`；

— `field`.

Java 注解

Java 注解与 Kotlin 100% 兼容：

```
import org.junit.Test
import org.junit.Assert.*
import org.junit.Rule
import org.junit.rules.*

class Tests {
    // 将 @Rule 注解应用于属性 getter
    @get:Rule val tempFolder = TemporaryFolder()

    @Test fun simple() {
        val f = tempFolder.newFile()
        assertEquals(42, getTheAnswer())
    }
}
```

因为 Java 编写的注解没有定义参数顺序，所以不能使用常规函数调用语法来传递参数。相反，你需要使用命名参数语法：

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

```
// Kotlin
@AnnWithValue("abc") class C
```

数组作为注解参数

如果 Java 中的 `value` 参数具有数组类型，它会成为 Kotlin 中的一个 `vararg` 参数：

```
// Java
public @interface AnnWithArrayValue {
    String[] value();
}
```

```kotlin
// Kotlin
@AnnWithArrayValue("abc", "foo", "bar") class C
```

对于具有数组类型的其他参数，你需要显式使用数组字面值语法（自 Kotlin 1.2 起）或者 `arrayOf(……)`：

```kotlin
// Java
public @interface AnnWithArrayMethod {
    String[] names();
}
```

```kotlin
// Kotlin 1.2+:
@AnnWithArrayMethod(names = ["abc", "foo", "bar"])
class C

// 旧版本 Kotlin:
@AnnWithArrayMethod(names = arrayOf("abc", "foo", "bar"))
class D
```

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：

```kotlin
// Java
public @interface Ann {
    int value();
}
```

```kotlin
// Kotlin
fun foo(ann: Ann) {
    val i = ann.value
}
```

反射

反射是这样的一组语言和库功能，它允许在运行时自省你的程序的结构。Kotlin 让语言中的函数和属性做为一等公民、并对其自省（即在运行时获悉一个名称或者一个属性或函数的类型）与简单地使用函数式或响应式风格紧密相关。

> 在 Java 平台上，使用反射功能所需的运行时组件作为单独的 JAR 文件（`kotlin-reflect.jar`）分发。这样做是为了减少不使用反射功能的应用程序所需的运行时库的大小。如果你需要使用反射，请确保该 .jar文件添加到项目的 classpath 中。

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。

请注意，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（自 1.1 起）

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}" }
```

你可以获取对象的精确类的引用，例如 `GoodWidget` 或 `BadWidget`，尽管接收者表达式的类型是 `Widget`。

可调用引用

函数、属性以及构造函数的引用，除了作为自省程序结构外，还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

当我们有一个命名函数声明如下：

```
fun isOdd(x: Int) = x % 2 != 0
```

我们可以很容易地直接调用它（`isOdd(5)`），但是我们也可以将其作为一个函数类型的值，例如将其传给另一个函数。为此，我们使用 `::` 操作符：

```
val numbers = listOf(1, 2, 3)
println(numbers.filter(::isOdd))
```

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 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)
```

或者，你可以通过将方法引用存储在具有显式指定类型的变量中来提供必要的上下文：

```
val predicate: (String) -> Boolean = ::isOdd   // 引用到 isOdd(x: String)
```

如果我们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`。

请注意，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（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(*))`。现在，你可以将其应用于可调用引用：

```
fun length(s: String) = s.length

val oddLength = compose(::isOdd, ::length)
val strings = listOf("a", "ab", "abc")

println(strings.filter(oddLength))
```

属性引用

要把属性作为 Kotlin中 的一等对象来访问，我们也可以使用 `::` 运算符：

```
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，它允许我们使用 `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)
}
```

属性引用可以用在不需要参数的函数处：

```
val strs = listOf("a", "bc", "def")
println(strs.map(String::length))
```

要访问属于类的成员的属性，我们这样限定它：

```
class A(val p: Int)
val prop = A::p
println(prop.get(A(1)))
```

对于扩展属性：

```
val String.lastChar: Char
    get() = this[length - 1]

fun main() {
    println(String::lastChar.get("abc"))
}
```

与 Java 反射的互操作性

在Java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。他们可以用于期待这样的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，我们可以这样简单地调用它：

```
function(::Foo)
```

构造函数的可调用引用的类型也是 KFunction<out R> 的子类型之一，取决于其参数个数。

绑定的函数与属性引用（自 1.1 起）

你可以引用特定对象的实例方法：

```
val numberRegex = "\\d+".toRegex()
println(numberRegex.matches("29"))

val isNumber = numberRegex::matches
println(isNumber("29"))
```

取代直接调用方法 `matches` 的是我们存储其引用。这样的引用会绑定到其接收者上。它可以直接调用（如上例所示）或者用于任何期待一个函数类型表达式的时候：

```
val numberRegex = "\\d+".toRegex()
val strings = listOf("abc", "124", "a70")
println(strings.filter(numberRegex::matches))
```

比较绑定的类型和相应的未绑定类型的引用。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：

```
val prop = "abc"::length
println(prop.get())
```

自 Kotlin 1.2 起，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

inner 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
```

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）。以下是构建器的一些示例应用场景：

— 使用 Kotlin 代码生成标记语言，例如 HTML 或 XML；

— 以编程方式布局 UI 组件：Anko；

— 为 Web 服务器配置路由：Ktor。

一个类型安全的构建器示例

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 = "http://kotlinlang.org") {+"Kotlin"}

            // 混合的内容
            p {
                +"This is some"
                b {+"mixed"}
                +"text. For more see the"
                a(href = "http://kotlinlang.org") {+"Kotlin"}
                +"project"
```

```
            }
            p {+"some text"}

            // 以下代码生成的内容
            p {
                for (arg in args)
                    +arg
            }
        }
    }
```

这是完全合法的 Kotlin 代码。你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

让我们来看看 Kotlin 中实现类型安全构建器的机制。首先，我们需要定义我们想要构建的模型，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，也就是说它定义了像 `<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我们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*。这意味着我们需要向函数传递一个 HTML 类型的实例（*接收者*），并且我们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，我们得到的东西看起来已经非常像一个构建器了：

```
html {
    head { …… }
    body { …… }
}
```

那么，这个调用做什么？让我们看看上面定义的 `html` 函数的主体。它创建了一个 `HTML` 的新实例，然后通过调用作为参数传入的函数来初始化它（在我们的示例中，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，所以我们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，现在我们的函数很简单：

```
fun head(init: Head.() -> Unit) = initTag(Head(), init)

fun body(init: Body.() -> Unit) = initTag(Body(), init)
```

并且我们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。在上例中我们这样写到：

```
html {
    head {
        title {+"XML encoding with Kotlin"}
    }
    // ……
}
```

所以基本上，我们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（自 1.1 起）

使用 DSL 时，可能会遇到上下文中可以调用太多函数的问题。我们可以调用 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，在 Kotlin 1.1 中引入了一种控制接收者作用域的特殊机制。

为了使编译器开始控制标记，我们只是必须用相同的标记注解来标注在 DSL 中使用的所有接收者的类型。例如，对于 HTML 构建器，我们声明一个注解 `@HTMLTagMarker`：

```
@DslMarker
annotation class HtmlTagMarker
```

如果一个注解类使用 `@DslMarker` 注解标注，那么该注解类称为 DSL 标记。

在我们的 DSL 中，所有标签类都扩展了相同的超类 `Tag`。只需使用 `@HtmlTagMarker` 来标注超类就足够了，之后，Kotlin 编译器会将所有继承的类视为已标注：

```
@HtmlTagMarker
abstract class Tag(val name: String) { …… }
```

我们不必用 `@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。

请注意,`@DslMarker` 注解在 Kotlin 1.1 起才可用。

```
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")
```

```
        builder.append("$indent</$name>\n")
    }

    private fun renderAttributes(): String {
        val builder = StringBuilder()
        for ((attr, value) in attributes) {
            builder.append(" $attr=\"$value\"")
        }
        return builder.toString()
    }

    override fun toString(): String {
        val builder = StringBuilder()
        render(builder, "")
        return builder.toString()
    }
}

abstract class TagWithText(name: String) : Tag(name) {
    operator fun String.unaryPlus() {
        children.add(TextElement(this))
    }
}

class HTML : TagWithText("html") {
    fun head(init: Head.() -> Unit) = initTag(Head(), init)

    fun body(init: Body.() -> Unit) = initTag(Body(), init)
}

class Head : TagWithText("head") {
    fun title(init: Title.() -> Unit) = initTag(Title(), init)
}

class Title : TagWithText("title")

abstract class BodyTag(name: String) : TagWithText(name) {
    fun b(init: B.() -> Unit) = initTag(B(), init)
    fun p(init: P.() -> Unit) = initTag(P(), init)
    fun h1(init: H1.() -> Unit) = initTag(H1(), init)
    fun a(href: String, init: A.() -> Unit) {
        val a = initTag(A(), init)
        a.href = href
    }
}

class Body : BodyTag("body")
class B : BodyTag("b")
class P : BodyTag("p")
class H1 : BodyTag("h1")

class A : BodyTag("a") {
    var href: String
        get() = attributes["href"]!!
        set(value) {
            attributes["href"] = value
        }
}

fun html(init: HTML.() -> Unit): HTML {
```

```
    val html = HTML()
    html.init()
    return html
}
```

类型别名

类型别名为现有类型提供替代名称。如果类型名称太长，你可以另外引入较短的名称，并使用新的名称替代原类型名。

它有助于缩短较长的泛型类型。例如，通常缩减集合类型是很有吸引力的：

```
typealias NodeSet = Set<Network.Node>

typealias FileTable<K> = MutableMap<K, MutableList<File>>
```

你可以为函数类型提供另外的别名：

```
typealias MyHandler = (Int, String, Any) -> Unit

typealias Predicate<T> = (T) -> Boolean
```

你可以为内部类和嵌套类创建新名称：

```
class A {
    inner class Inner
}
class B {
    inner class Inner
}

typealias AInner = A.Inner
typealias BInner = B.Inner
```

类型别名不会引入新类型。它们等效于相应的底层类型。当你在代码中添加 `typealias Predicate<T>` 并使用 `Predicate<Int>` 时，Kotlin 编译器总是把它扩展为 `(Int) -> Boolean`。因此，当你需要泛型函数类型时，你可以传递该类型的变量，反之亦然：

```
typealias Predicate<T> = (T) -> Boolean

fun foo(p: Predicate<Int>) = p(42)

fun main() {
    val f: (Int) -> Boolean = { it > 0 }
    println(foo(f)) // 输出 "true"

    val p: Predicate<Int> = { it > 0 }
    println(listOf(1, -2).filter(p)) // 输出 "[1]"
}
```

参考

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查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查一个值**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`
  - 检查一个值具有指定类型
  - 在 when 表达式中用于上述目的
- `!is`
  - 检查一个值**不**具有指定类型
  - 在 when 表达式中用于上述目的
- `null` 是表示不指向任何对象的对象引用的常量
- `object` 同时声明一个类及其实例
- `package` 指定当前文件的包
- `return` 从最近层的函数或匿名函数返回
- `super`
  - 引用一个方法或属性的超类实现
  - 在次构造函数中调用超类构造函数
- `this`
  - 引用当前接收者
  - 在次构造函数中调用同一个类的另一个构造函数
- `throw` 抛出一个异常
- `true` 指定布尔类型的“真”值
- `try` 开始一个异常处理块
- `typealias` 声明一个类型别名
- `val` 声明一个只读属性或局部变量
- `var` 声明一个可变属性或局部变量
- `when` 开始一个 when 表达式（执行其中一个给定分支）
- `while` 开始一个 while 循环（前置条件的循环）

软关键字

以下符号在适用的上下文中充当关键字，而在其他上下文中可用作标识符：

- `by`
  - 将接口的实现委托给另一个对象
  - 将属性访问器的实现委托给另一个对象
- `catch` 开始一个处理指定异常类型的块

- `constructor` 声明一个主构造函数或次构造函数
- `delegate` 用作注解使用处目标
- `dynamic` 引用一个 Kotlin/JS 代码中的动态类型
- `field` 用作注解使用处目标
- `file` 用作注解使用处目标
- `finally` 开始一个当 try 块退出时总会执行的块
- `get`
  - 声明属性的 getter
  - 用作注解使用处目标
- `import` 将另一个包中的声明导入当前文件
- `init` 开始一个初始化块
- `param` 用作注解使用处目标
- `property` 用作注解使用处目标
- `receiver` 用作注解使用处目标
- `set`
  - 声明属性的 setter
  - 用作注解使用处目标
- `setparam` 用作注解使用处目标
- `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：

- `actual` 表示多平台项目中的一个平台相关实现
- `abstract` 将一个类或成员标记为抽象
- `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` 将一个声明标记为不是在 Kotlin 中实现（通过 JNI 访问或者在 JavaScript 中实现）
— `final` 禁止成员覆盖
— `infix` 允许以中缀表示法调用函数
— `inline` 告诉编译器在调用处内联传给它的函数和 lambda 表达式
— `inner` 允许在嵌套类中引用外部类实例
— `internal` 将一个声明标记为在当前模块中可见
— `lateinit` 允许在构造函数之外初始化非空属性
— `noinline` 关闭传给内联函数的 lambda 表达式的内联
— `open` 允许一个类子类化或覆盖成员
— `operator` 将一个函数标记为重载一个操作符或者实现一个约定
— `out` 将类型参数标记为协变
— `override` 将一个成员标记为超类成员的覆盖
— `private` 将一个声明标记为在当前类或文件中可见
— `protected` 将一个声明标记为在当前类及其子类中可见
— `public` 将一个声明标记为在任何地方可见
— `reified` 将内联函数的类型参数标记为在运行时可访问
— `sealed` 声明一个密封类（限制子类化的类）
— `suspend` 将一个函数或 lambda 表达式标记为挂起式（可用做协程）
— `tailrec` 将一个函数标记为尾递归（允许编译器将递归替换为迭代）
— `vararg` 允许一个参数传入可变数量的参数

特殊标识符

以下标识符由编译器在指定上下文中定义，并且可以用作其他上下文中的常规标识符：

— `field` 用在属性访问器内部来引用该属性的幕后字段
— `it` 用在 lambda 表达式内部来隐式引用其参数

操作符和特殊符号

Kotlin 支持以下操作符和特殊符号：

— `+`、`-`、`*`、`/`、`%` —— 数学操作符
  — `*` 也用于将数组传递给 vararg 参数
— `=`

    - 赋值操作符
    - 也用于指定参数的默认值
- `+=`、`-=`、`*=`、`/=`、`%=` —— 广义赋值操作符
- `++`、`--` —— 递增与递减操作符
- `&&`、`||`、`!` —— 逻辑“与”、“或”、“非”操作符（对于位运算，请使用相应的中缀函数）
- `==`、`!=` —— 相等操作符（对于非原生类型会翻译为调用 `equals()`）
- `===`、`!==` —— 引用相等操作符
- `<`、`>`、`<=`、`>=` —— 比较操作符（对于非原生类型会翻译为调用 `compareTo()`）
- `[`、`]` —— 索引访问操作符（会翻译为调用 `get` 与 `set`）
- `!!` 断言一个表达式非空
- `?.` 执行安全调用（如果接收者非空，就调用一个方法或访问一个属性）
- `?:` 如果左侧的值为空，就取右侧的值（elvis 操作符）
- `::` 创建一个成员引用或者一个类引用
- `..` 创建一个区间
- `:` 分隔声明中的名称与类型
- `?` 将类型标记为可空
- `->`
    - 分隔 lambda 表达式的参数与主体
    - 分隔在函数类型中的参数类型与返回类型声明
    - 分隔 when 表达式分支的条件与代码体
- `@`
    - 引入一个注解
    - 引入或引用一个循环标签
    - 引入或引用一个 lambda 表达式标签
    - 引用一个来自外部作用域的 “this” 表达式
    - 引用一个外部超类
- `;` 分隔位于同一行的多个语句
- `$` 在字符串模版中引用变量或者表达式
- `_`
    - 在 lambda 表达式中代替未使用的参数
    - 在解构声明中代替未使用的参数

Grammar

Notation

This section informally explains the grammar notation used below.

Symbols and naming

Terminal symbol names start with an uppercase letter, e.g. **SimpleName**.
Nonterminal symbol names start with lowercase letter, e.g. **kotlinFile**.
Each *production* starts with a colon (**:**).
Symbol definitions may have many productions and are terminated by a semicolon (**;**).
Symbol definitions may be prepended with *attributes*, e.g. `start` attribute denotes a start symbol.

EBNF expressions

Operator `|` denotes *alternative*.
Operator `*` denotes *iteration* (zero or more).
Operator `+` denotes *iteration* (one or more).
Operator `?` denotes *option* (zero or one).
alpha `{` beta `}` denotes a nonempty *beta*-separated list of *alpha*'s.
Operator ``++'' means that no space or comment allowed between operands.

Semicolons

Kotlin provides "semicolon inference": syntactically, subsentences (e.g., statements, declarations etc) are separated by the pseudo-token SEMI, which stands for "semicolon or newline". In most cases, there's no need for semicolons in Kotlin code.

Syntax

Relevant pages: Packages

```
start
kotlinFile
  : preamble topLevelObject*
  ;
start
script
  : preamble expression*
  ;
preamble
(used by script, kotlinFile)
  : fileAnnotations? packageHeader? import*
  ;
fileAnnotations
(used by preamble)
  : fileAnnotation*
  ;
fileAnnotation
(used by fileAnnotations)
  : "@" "file" ":" ("[" unescapedAnnotation+ "]" | unescapedAnnotation)
  ;
packageHeader
(used by preamble)
```

: modifiers "package" SimpleName{"."} SEMI?
;

See Packages

import
(used by preamble)
: "import" SimpleName{"."} ("." "*" | "as" SimpleName)? SEMI?
;

See Imports

topLevelObject
(used by kotlinFile)
: class
: object
: function
: property
: typeAlias
;
typeAlias
(used by memberDeclaration, declaration, topLevelObject)
: modifiers "typealias" SimpleName typeParameters? "=" type
;

Classes

See Classes and Inheritance

class
(used by memberDeclaration, declaration, topLevelObject)
: modifiers ("class" | "interface") SimpleName
typeParameters?
primaryConstructor?
(":" annotations delegationSpecifier{","})?
typeConstraints
(classBody? | enumClassBody)
;
primaryConstructor
(used by class, object)
: (modifiers "constructor")? ("(" functionParameter{","} ")")
;
classBody
(used by objectLiteral, enumEntry, class, companionObject, object)
: ("{" members "}")?
;
members
(used by enumClassBody, classBody)
: memberDeclaration*
;
delegationSpecifier
(used by objectLiteral, class, companionObject, object)
: constructorInvocation
: userType
: explicitDelegation
;
explicitDelegation
(used by delegationSpecifier)
: userType "by" expression
;
typeParameters
(used by typeAlias, class, property, function)
: "<" typeParameter{","} ">"
;
typeParameter
(used by typeParameters)
: modifiers SimpleName (":" userType)?
;

See Generic classes

typeConstraints
(used by class, property, function)
: ("where" typeConstraint{","})?
;
typeConstraint
(used by typeConstraints)

: annotations SimpleName ":" type
;

See Generic constraints

Class members

memberDeclaration
(used by members)
: companionObject
: object
: function
: property
: class
: typeAlias
: anonymousInitializer
: secondaryConstructor
;
anonymousInitializer
(used by memberDeclaration)
: "init" block
;
companionObject
(used by memberDeclaration)
: modifiers "companion" "object" SimpleName? (":" delegationSpecifier{","})? classBody?
;
valueParameters
(used by secondaryConstructor, function)
: "(" functionParameter{","}? ")"
;
functionParameter
(used by valueParameters, primaryConstructor)
: modifiers ("val" | "var")? parameter ("=" expression)?
;
block
(used by catchBlock, anonymousInitializer, secondaryConstructor, functionBody, controlStructureBody, try, finallyBlock)
: "{" statements "}"
;
function
(used by memberDeclaration, declaration, topLevelObject)
: modifiers "fun"
typeParameters?
(type ".")?
SimpleName
typeParameters? valueParameters (":" type)?
typeConstraints
functionBody?
;
functionBody
(used by getter, setter, function)
: block
: "=" expression
;
variableDeclarationEntry
(used by for, lambdaParameter, property, multipleVariableDeclarations)
: SimpleName (":" type)?
;
multipleVariableDeclarations
(used by for, lambdaParameter, property)
: "(" variableDeclarationEntry{","} ")"
;
property
(used by memberDeclaration, declaration, topLevelObject)
: modifiers ("val" | "var")
typeParameters?
(type ".")?
(multipleVariableDeclarations | variableDeclarationEntry)
typeConstraints
("by" | "=" expression SEMI?)?
(getter? setter? | setter? getter?) SEMI?
;

See Properties and Fields

getter

(used by property)
 : modifiers "get"
 : modifiers "get" "(" ")" (":" type)? functionBody
 ;
setter
(used by property)
 : modifiers "set"
 : modifiers "set" "(" modifiers (SimpleName | parameter) ")" functionBody
 ;
parameter
(used by functionType, setter, functionParameter)
 : SimpleName ":" type
 ;
object
(used by memberDeclaration, declaration, topLevelObject)
 : "object" SimpleName primaryConstructor? (":" delegationSpecifier{","})? classBody?
secondaryConstructor
(used by memberDeclaration)
 : modifiers "constructor" valueParameters (":" constructorDelegationCall)? block
 ;
constructorDelegationCall
(used by secondaryConstructor)
 : "this" valueArguments
 : "super" valueArguments
 ;

See Object expressions and Declarations

Enum classes

See Enum classes

enumClassBody
(used by class)
 : "{" enumEntries (";" members)? "}"
 ;
enumEntries
(used by enumClassBody)
 : (enumEntry{","} ","? ";"?)?
 ;
enumEntry
(used by enumEntries)
 : modifiers SimpleName ("(" arguments ")")? classBody?
 ;

Types

See Types

type
(used by namedInfix, simpleUserType, getter, atomicExpression, whenCondition, property, typeArguments, function, typeAlias, parameter, functionType, variableDeclarationEntry, lambdaParameter, typeConstraint)
 : typeModifiers typeReference
 ;
typeReference
(used by typeReference, nullableType, type)
 : "(" typeReference ")"
 : functionType
 : userType
 : nullableType
 : "dynamic"
 ;
nullableType
(used by typeReference)
 : typeReference "?"
 ;
userType
(used by typeParameter, catchBlock, callableReference, typeReference, delegationSpecifier, constructorInvocation, explicitDelegation)
 : simpleUserType{"."}
 ;
simpleUserType
(used by userType)

: SimpleName ("<" (optionalProjection type | "*"){","} ">")?
;
optionalProjection
(used by simpleUserType)
: varianceAnnotation
;
functionType
(used by typeReference)
: (type ".")? "(" parameter{","}? ")" "->" type?
;

Control structures

See Control structures

controlStructureBody
(used by whenEntry, for, if, doWhile, while)
: block
: blockLevelExpression
;
if
(used by atomicExpression)
: "if" "(" expression ")" controlStructureBody SEMI? ("else" controlStructureBody)?
;
try
(used by atomicExpression)
: "try" block catchBlock* finallyBlock?
;
catchBlock
(used by try)
: "catch" "(" annotations SimpleName ":" userType ")" block
;
finallyBlock
(used by try)
: "finally" block
;
loop
(used by atomicExpression)
: for
: while
: doWhile
;
for
(used by loop)
: "for" "
(" annotations (multipleVariableDeclarations | variableDeclarationEntry) "in" expression ")" controlStructureBody
;
while
(used by loop)
: "while" "(" expression ")" controlStructureBody
;
doWhile
(used by loop)
: "do" controlStructureBody "while" "(" expression ")"
;

Expressions

Precedence

Precedence	Title	Symbols
Highest	Postfix	`++`, `--`, `.`, `?.`, `?`
	Prefix	`-`, `+`, `++`, `--`, `!`, `labelDefinition@`
	Type RHS	`:`, `as`, `as?`
	Multiplicative	`*`, `/`, `%`
	Additive	`+`, `-`
	Range	`..`
	Infix function	`SimpleName`
	Elvis	`?:`
	Named checks	`in`, `!in`, `is`, `!is`
	Comparison	`<`, `>`, `<=`, `>=`
	Equality	`==`, `\!==`
	Conjunction	`&&`
	Disjunction	`||`
Lowest	Assignment	`=`, `+=`, `-=`, `*=`, `/=`, `%=`

Rules

expression
(used by for, atomicExpression, longTemplate, whenCondition, functionBody, doWhile, property, script, explicitDelegation, jump, while, arrayAccess, blockLevelExpression, if, when, valueArguments, functionParameter)
 : disjunction (assignmentOperator disjunction)*
 ;
disjunction
(used by expression)
 : conjunction ("||" conjunction)*
 ;
conjunction
(used by disjunction)
 : equalityComparison ("&&" equalityComparison)*
 ;
equalityComparison
(used by conjunction)
 : comparison (equalityOperation comparison)*
 ;
comparison
(used by equalityComparison)
 : namedInfix (comparisonOperation namedInfix)*
 ;
namedInfix
(used by comparison)
 : elvisExpression (inOperation elvisExpression)*
 : elvisExpression (isOperation type)?
 ;
elvisExpression
(used by namedInfix)
 : infixFunctionCall ("?:" infixFunctionCall)*
 ;
infixFunctionCall
(used by elvisExpression)
 : rangeExpression (SimpleName rangeExpression)*
 ;
rangeExpression
(used by infixFunctionCall)
 : additiveExpression (".." additiveExpression)*
 ;
additiveExpression
(used by rangeExpression)
 : multiplicativeExpression (additiveOperation multiplicativeExpression)*
 ;
multiplicativeExpression
(used by additiveExpression)
 : typeRHS (multiplicativeOperation typeRHS)*

;
typeRHS
(used by multiplicativeExpression)
: prefixUnaryExpression (typeOperation prefixUnaryExpression)*
;
prefixUnaryExpression
(used by typeRHS)
: prefixUnaryOperation* postfixUnaryExpression
;
postfixUnaryExpression
(used by prefixUnaryExpression, postfixUnaryOperation)
: atomicExpression postfixUnaryOperation*
: callableReference postfixUnaryOperation*
;
callableReference
(used by postfixUnaryExpression)
: (userType "?"*)? "::" SimpleName typeArguments?
;
atomicExpression
(used by postfixUnaryExpression)
: "(" expression ")"
: literalConstant
: functionLiteral
: "this" labelReference?
: "super" ("<" type ">")? labelReference?
: if
: when
: try
: objectLiteral
: jump
: loop
: collectionLiteral
: SimpleName
;
labelReference
(used by atomicExpression, jump)
: "@" ++ LabelName
;
labelDefinition
(used by prefixUnaryOperation, annotatedLambda)
: LabelName ++ "@"
;
literalConstant
(used by atomicExpression)
: "true" | "false"
: stringTemplate
: NoEscapeString
: IntegerLiteral
: HexadecimalLiteral
: CharacterLiteral
: FloatLiteral
: "null"
;
stringTemplate
(used by literalConstant)
: "\"" stringTemplateElement* "\""
;
stringTemplateElement
(used by stringTemplate)
: RegularStringPart
: ShortTemplateEntryStart (SimpleName | "this")
: EscapeSequence
: longTemplate
;
longTemplate
(used by stringTemplateElement)
: "${" expression "}"
;
declaration
(used by statement)
: function
: property
: class
: typeAlias
: object
;
statement
(used by statements)

: declaration
: blockLevelExpression
;
blockLevelExpression
(used by statement, controlStructureBody)
: annotations ("\n")+ expression
;
multiplicativeOperation
(used by multiplicativeExpression)
: "*" : "/" : "%"
;
additiveOperation
(used by additiveExpression)
: "+" : "-"
;
inOperation
(used by namedInfix)
: "in" : "!in"
;
typeOperation
(used by typeRHS)
: "as" : "as?" : ":"
;
isOperation
(used by namedInfix)
: "is" : "!is"
;
comparisonOperation
(used by comparison)
: "<" : ">" : ">=" : "<="
;
equalityOperation
(used by equalityComparison)
: "!=" : "=="
;
assignmentOperator
(used by expression)
: "="
: "+=" : "-=" : "*=" : "/=" : "%="
;
prefixUnaryOperation
(used by prefixUnaryExpression)
: "-" : "+"
: "++" : "--"
: "!"
: annotations
: labelDefinition
;
postfixUnaryOperation
(used by postfixUnaryExpression)
: "++" : "--" : "!!"
: callSuffix
: arrayAccess
: memberAccessOperation postfixUnaryExpression
;
callSuffix
(used by constructorInvocation, postfixUnaryOperation)
: typeArguments? valueArguments annotatedLambda
: typeArguments annotatedLambda
;
annotatedLambda
(used by callSuffix)
: ("@" unescapedAnnotation)* labelDefinition? functionLiteral
memberAccessOperation
(used by postfixUnaryOperation)
: "." : "?." : "?"
;
typeArguments
(used by callSuffix, callableReference, unescapedAnnotation)
: "<" type{","} ">"
;
valueArguments
(used by callSuffix, constructorDelegationCall, unescapedAnnotation)
: "(" (SimpleName "=")? "*"? expression{","} ")"
;
jump
(used by atomicExpression)
: "throw" expression

: "return" ++ labelReference? expression?
: "continue" ++ labelReference?
: "break" ++ labelReference?
;
functionLiteral
(used by atomicExpression, annotatedLambda)
: "{" statements "}"
: "{" lambdaParameter{","} "->" statements "}"
;
lambdaParameter
(used by functionLiteral)
: variableDeclarationEntry
: multipleVariableDeclarations (":" type)?
;
statements
(used by block, functionLiteral)
: SEMI* statement{SEMI+} SEMI*
;
constructorInvocation
(used by delegationSpecifier)
: userType callSuffix
;
arrayAccess
(used by postfixUnaryOperation)
: "[" expression{","} "]"
;
objectLiteral
(used by atomicExpression)
: "object" (":" delegationSpecifier{","})? classBody
;
collectionLiteral
(used by atomicExpression)
: "[" element{","}? "]"
;

When-expression

See When-expression

when
(used by atomicExpression)
: "when" ("(" expression ")")? "{"
whenEntry*
"}"
;
whenEntry
(used by when)
: whenCondition{","} "->" controlStructureBody SEMI
: "else" "->" controlStructureBody SEMI
;
whenCondition
(used by whenEntry)
: expression
: ("in" | "!in") expression
: ("is" | "!is") type
;

Modifiers

modifiers
(used by typeParameter, getter, packageHeader, class, property, function, typeAlias, secondaryConstructor, enumEntry, setter, companionObject, primaryConstructor, functionParameter)
: (modifier | annotations)*
;
typeModifiers
(used by type)
: (suspendModifier | annotations)*
;
modifier
(used by modifiers)
: classModifier
: accessModifier
: varianceAnnotation
: memberModifier
: parameterModifier

: typeParameterModifier
: functionModifier
: propertyModifier
;
classModifier
(used by modifier)
: "abstract"
: "final"
: "enum"
: "open"
: "annotation"
: "sealed"
: "data"
;
memberModifier
(used by modifier)
: "override"
: "open"
: "final"
: "abstract"
: "lateinit"
;
accessModifier
(used by modifier)
: "private"
: "protected"
: "public"
: "internal"
;
varianceAnnotation
(used by modifier, optionalProjection)
: "in"
: "out"
;
parameterModifier
(used by modifier)
: "noinline"
: "crossinline"
: "vararg"
;
typeParameterModifier
(used by modifier)
: "reified"
;
functionModifier
(used by modifier)
: "tailrec"
: "operator"
: "infix"
: "inline"
: "external"
: suspendModifier
;
propertyModifier
(used by modifier)
: "const"
;
suspendModifier
(used by typeModifiers, functionModifier)
: "suspend"
;

Annotations

annotations
(used by catchBlock, prefixUnaryOperation, blockLevelExpression, for, typeModifiers, class, modifiers, typeConstraint)
: (annotation | annotationList)*
;
annotation
(used by annotations)
: "@" (annotationUseSiteTarget ":")? unescapedAnnotation
;
annotationList
(used by annotations)
: "@" (annotationUseSiteTarget ":")? "[" unescapedAnnotation+ "]"
;

annotationUseSiteTarget
(used by annotation, annotationList)
 : "field"
 : "file"
 : "property"
 : "get"
 : "set"
 : "receiver"
 : "param"
 : "setparam"
 : "delegate"
 ;
unescapedAnnotation
(used by annotation, fileAnnotation, annotatedLambda, annotationList)
 : SimpleName{"."} typeArguments? valueArguments?
 ;

Lexical structure

helper
Digit
(used by IntegerLiteral, HexDigit)
 : ["0".."9"];
IntegerLiteral
(used by literalConstant)
 : Digit (Digit | "_")*
FloatLiteral
(used by literalConstant)
 : <Java double literal>;
helper
HexDigit
(used by RegularStringPart, HexadecimalLiteral)
 : Digit | ["A".."F", "a".."f"];
HexadecimalLiteral
(used by literalConstant)
 : "0x" HexDigit (HexDigit | "_")*;
CharacterLiteral
(used by literalConstant)
 : <character as in Java>;

See Basic types

NoEscapeString
(used by literalConstant)
 : <"""-quoted string>;
RegularStringPart
(used by stringTemplateElement)
 : <any character other than backslash, quote, $ or newline>
ShortTemplateEntryStart:
 : "$"
EscapeSequence:
 : UnicodeEscapeSequence | RegularEscapeSequence
UnicodeEscapeSequence:
 : "\u" HexDigit{4}
RegularEscapeSequence:
 : "\" <any character other than newline>

See String templates

SEMI
(used by whenEntry, if, statements, packageHeader, property, import)
 : <semicolon or newline>;
SimpleName
(used by typeParameter, catchBlock, simpleUserType, atomicExpression, LabelName, packageHeader, class, object, infixFunctionCall, function, typeAlias, parameter, callableReference, variableDeclarationEntry, stringTemplateElement, enumEntry, setter, import, companionObject, valueArguments, unescapedAnnotation, typeConstraint)
 : <java identifier>
 : "`" <java identifier> "`"
 ;

See Java interoperability

LabelName
(used by labelReference, labelDefinition)
 : "@" SimpleName;

See Returns and jumps

Java 互操作

在 Kotlin 中调用 Java 代码

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 和 Setter

遵循 Java 约定的 getter 和 setter 的方法（名称以 `get` 开头的无参数方法和以 `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。例如：

```
import java.util.Calendar

fun calendarDemo() {
    val calendar = Calendar.getInstance()
    if (calendar.firstDayOfWeek == Calendar.SUNDAY) {  // 调用 getFirstDayOfWeek()
        calendar.firstDayOfWeek = Calendar.MONDAY      // 调用ll setFirstDayOfWeek()
    }
    if (!calendar.isLenient) {                         // 调用 isLenient()
        calendar.isLenient = true                      // 调用 setLenient()
    }
}
```

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 目前不支持只写（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 中会被特别对待并称为 *平台类型*。对这种类型的空检查会放宽，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）
```

当我们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
```

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），或者我们可以选择我们期望的类型（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
```

如果我们选择非空类型，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。当我们把平台值传递给期待非空值等的 Kotlin 函数时，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（在错误信息中、参数信息中等）显示他们，所以我们用一个助记符来表示他们：

— `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`）
- 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`）。

你可以在 Kotlin 编译器源代码中找到完整的列表。

注解类型参数

可以标注泛型类型的类型参数，以便同时为其提供可空性信息。例如，考虑这些 Java 声明的注解：

```
@NotNull
Set<@NotNull String> toSet(@NotNull Collection<@NotNull String> elements) { …… }
```

在 Kotlin 中可见的是以下签名：

```
fun toSet(elements: (Mutable)Collection<String>) : (Mutable)Set<String> { …… }
```

请注意 `String` 类型参数上的 `@NotNull` 注解。如果没有的话，类型参数会是平台类型：

```
fun toSet(elements: (Mutable)Collection<String!>) : (Mutable)Set<String!> { …… }
```

标注类型参数适用于针对 Java 8 或更高版本环境，并且要求可空性注解支持 `TYPE_USE` 目标（`org.jetbrains.annotations` 15 或以上版本支持）。

> 注：由于当前的技术限制，IDE 无法正确识别用作依赖的已编译 Java 库中类型参数上的这些注解。

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。

自 Kotlin 1.1.50 起，也支持自定义可空限定符（KEEP-79）（见下文）。

类型限定符别称（自 1.1.50 起）

如果一个注解类型同时标注有 `@TypeQualifierNickname` 与 JSR-305 `@Nonnull`（或者它的其他别称，如 `@CheckForNull`），那么该注解类型自身将用于 检索精确的可空性，且具有与该可空性注解相同的含义：

```
@TypeQualifierNickname
@Nonnull(when = When.ALWAYS)
@Retention(RetentionPolicy.RUNTIME)
public @interface MyNonnull {
}

@TypeQualifierNickname
@CheckForNull // 另一个类型限定符别称的别称
@Retention(RetentionPolicy.RUNTIME)
public @interface MyNullable {
}

interface A {
    @MyNullable String foo(@MyNonnull String x);
    // 在 Kotlin(严格模式)中:`fun foo(x: String): String?`

    String bar(List<@MyNonnull String> x);
    // 在 Kotlin(严格模式)中:`fun bar(x: List<String>!): String!`
}
```

类型限定符默认值（自 1.1.50 起）

`@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；以及
- `ElementType.TYPE_USE`（自 1.1.60 起）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。

```java
@Nonnull
@TypeQualifierDefault({ElementType.METHOD, ElementType.PARAMETER})
public @interface NonNullApi {
}

@Nonnull(when = When.MAYBE)
@TypeQualifierDefault({ElementType.METHOD, ElementType.PARAMETER, ElementType.TYPE_USE})
public @interface NullableApi {
}

@NullableApi
interface A {
    String foo(String x); // fun foo(x: String?): String?

    @NotNullApi // 覆盖来自接口的默认值
    String bar(String x, @Nullable String y); // fun bar(x: String, y: String?): String

    // 由于 `@NullableApi` 具有 `TYPE_USE` 元素类型,
    // 因此认为 List<String> 类型参数是可空的:
    String baz(List<String> x); // fun baz(List<String?>?): String?

    // “x”参数仍然是平台类型,因为有显式
    // UNKNOWN 标记的可空性注解:
    String qux(@Nonnull(when = When.UNKNOWN) String x); // fun baz(x: String!): String?
}
```

注意: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:

```java
// 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(自 1.1.60 起)

库的维护者可以使用 `@UnderMigration` 注解(在单独的构件 `kotlin-annotations-jvm` 中提供)来定义可为空性类型限定符的迁移状态。

`@UnderMigration(status = ...)` 中的状态值指定了编译器如何处理 Kotlin 中注解类型的不当用法(例如,使用 `@MyNullable` 标注的类型值作为非空值):

- `MigrationStatus.STRICT` 使注解像任何纯可空性注解一样工作,即对不当用法报错并影响注解声明内的类型在 Kotlin 中的呈现;
- 对于 `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;而
- `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 {}
```

注意: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}`(自 1.1.60 起)覆盖 `@UnderMigration` 注解的行为。用户可能对库的迁移状态有不同的看法:他们可能希望在官方迁移状态为 `WARN` 时报错误,反之亦然,他们可能希望推迟错误报告直到他们完成迁移。
- `-Xjsr305=@<fq.name>:{strict|warn|ignore}`(自 1.1.60 起)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。

对于 kotlin 1.1.50+/1.2 版本,其默认行为等同于 `-Xjsr305=warn`。`strict` 值应认为是实验性的(以后可能添加更多检测)。

已映射类型

Kotlin 特殊处理一部分 Java 类型。这样的类型不是“按原样”从 Java 加载，而是 *映射* 到相应的 Kotlin 类型。映射只发生在编译期间，运行时表示保持不变。Java 的原生类型映射到相应的 Kotlin 类型（请记住平台类型）：

Java 类型	Kotlin 类型
byte	kotlin.Byte
short	kotlin.Short
int	kotlin.Int
long	kotlin.Long
char	kotlin.Char
float	kotlin.Float
double	kotlin.Double
boolean	kotlin.Boolean

一些非原生的内置类型也会作映射：

Java 类型	Kotlin 类型
java.lang.Object	kotlin.Any!
java.lang.Cloneable	kotlin.Cloneable!
java.lang.Comparable	kotlin.Comparable!
java.lang.Enum	kotlin.Enum!
java.lang.Annotation	kotlin.Annotation!
java.lang.Deprecated	kotlin.Deprecated!
java.lang.CharSequence	kotlin.CharSequence!
java.lang.String	kotlin.String!
java.lang.Number	kotlin.Number!
java.lang.Throwable	kotlin.Throwable!

Java 的装箱原始类型映射到可空的 Kotlin 类型：

Java type	Kotlin type
java.lang.Byte	kotlin.Byte?
java.lang.Short	kotlin.Short?
java.lang.Integer	kotlin.Int?
java.lang.Long	kotlin.Long?
java.lang.Character	kotlin.Char?
java.lang.Float	kotlin.Float?
java.lang.Double	kotlin.Double?
java.lang.Boolean	kotlin.Boolean?

请注意，用作类型参数的装箱原始类型映射到平台类型：例如，`List<java.lang.Integer>` 在 Kotlin 中会成为 `List<Int!>`。

集合类型在 Kotlin 中可以是只读的或可变的，因此 Java 集合类型作如下映射：（下表中的所有 Kotlin 类型都驻留在 `kotlin.collections` 包中）：

Java 类型	Kotlin 只读类型	Kotlin 可变类型	加载的平台类型
`Iterator<T>`	`Iterator<T>`	`MutableIterator<T>`	`(Mutable)Iterator<T>!`
`Iterable<T>`	`Iterable<T>`	`MutableIterable<T>`	`(Mutable)Iterable<T>!`
`Collection<T>`	`Collection<T>`	`MutableCollection<T>`	`(Mutable)Collection<T>!`
`Set<T>`	`Set<T>`	`MutableSet<T>`	`(Mutable)Set<T>!`
`List<T>`	`List<T>`	`MutableList<T>`	`(Mutable)List<T>!`
`ListIterator<T>`	`ListIterator<T>`	`MutableListIterator<T>`	`(Mutable)ListIterator<T>!`
`Map<K, V>`	`Map<K, V>`	`MutableMap<K, V>`	`(Mutable)Map<K, V>!`
`Map.Entry<K, V>`	`Map.Entry<K, V>`	`MutableMap.MutableEntry<K,V>`	`(Mutable)Map.(Mutable)Entry<K, V>!`

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>!`

注意：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，我们会执行一些转换：

- 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<*>!`，即 `List<out Any?>!`。

和 Java 一样，Kotlin 在运行时不保留泛型，即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。即 `ArrayList<Integer>()` 和 `ArrayList<Character>()` 是不能区分的。这使得执行 `is`-检测不可能照顾到泛型。Kotlin 只允许 `is`-检测星投影的泛型类型：

```
if (a is List<Int>) // 错误：无法检查它是否真的是一个 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 不允许我们把一个 `Array<String>` 赋值给一个 `Array<Any>`，从而避免了可能的运行时故障。Kotlin 也禁止我们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 中你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)
}
```

即使当我们使用索引定位时，也不会引入任何开销：

```
for (i in array.indices) {// 不会创建迭代器
    array[i] += 2
}
```

最后，in-检测也没有额外开销：

```
if (i in array.indices) { // 同 (i >= 0 && i < array.size)
    print(array[i])
}
```

Java 可变参数

Java 类有时声明一个具有可变数量参数（varargs）的方法来使用索引：

```
public class JavaArrayExample {

    public void removeIndicesVarArg(int... indices) {
        // 在此编码……
    }
}
```

在这种情况下，你需要使用展开运算符 `*` 来传递 `IntArray`：

```
val javaObj = JavaArrayExample()
val array = intArrayOf(0, 1, 2, 3)
javaObj.removeIndicesVarArg(*array)
```

目前无法传递 `null` 给一个声明为可变参数的方法。

操作符

由于 Java 无法标记用于运算符语法的方法，Kotlin 允许具有正确名称和签名的任何 Java 方法作为运算符重载和其他约定（`invoke()` 等）使用。不允许使用中缀调用语法调用 Java 方法。

受检异常

在 Kotlin 中，所有异常都是非受检的，这意味着编译器不会强迫你捕获其中的任何一个。因此，当你调用一个声明受检异常的 Java 方法时，Kotlin 不会强迫你做任何事情：

```
fun render(list: List<*>, to: Appendable) {
    for (item in list) {
        to.append(item.toString()) // Java 会要求我们在这里捕获 IOException
    }
}
```

对象方法

当 Java 类型导入到 Kotlin 中时，类型 `java.lang.Object` 的所有引用都成了 `Any`。而因为 `Any` 不是平台指定的，它只声明了 `toString()`、`hashCode()` 和 `equals()` 作为其成员，所以为了能用到 `java.lang.Object` 的其他成员，Kotlin 要用到扩展函数。

wait()/notify()

类型 `Any` 的引用没有提供 `wait()` 与 `notify()` 方法。通常不鼓励使用它们，而建议使用 `java.utl.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
```

上面的代码使用了自 Kotlin 1.1 起支持的绑定的类引用。你也可以使用 `javaClass` 扩展属性：

```
val fooClass = foo.javaClass
```

clone()

要覆盖 `clone()`，需要继承 `kotlin.Cloneable`：

```
class Example : Cloneable {
    override fun clone(): Any { …… }
}
```

不要忘记《Effective Java》第三版 的第 13 条: *谨慎地改写clone*。

finalize()

要覆盖 `finalize()`，所有你需要做的就是简单地声明它，而不需要 `override` 关键字：

```
class C {
    protected fun finalize() {
        // 终止化逻辑
    }
}
```

根据 Java 的规则，`finalize()` 不能是 `private` 的。

从 Java 类继承

在 kotlin 中，类的超类中最多只能有一个 Java 类（以及按你所需的多个 Java 接口）。

访问静态成员

Java 类的静态成员会形成该类的“伴生对象”。我们无法将这样的“伴生对象”作为值来传递，但可以显式访问其成员，例如：

```
if (Character.isLetter(a)) { …… }
```

要访问已映射到 Kotlin 类型的 Java 类型的静态成员，请使用 Java 类型的完整限定名：`java.lang.Integer.bitCount(foo)`。

Java 反射

Java 反射适用于 Kotlin 类，反之亦然。如上所述，你可以使用 `instance::class.java`，`ClassName::class.java` 或者 `instance.javaClass` 通过 `java.lang.Class` 来进入 Java 反射。

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

就像 Java 8 一样，Kotlin 支持 SAM 转换。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") })
```

请注意，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。

还要注意，此功能只适用于 Java 互操作；因为 Kotlin 具有合适的函数类型，所以不需要将函数自动转换为 Kotlin 接口的实现，因此不受支持。

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。

Java 中调用 Kotlin

Java 可以轻松调用 Kotlin 代码。

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.foo.bar` 包内的 `example.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.foo.bar.ExampleKt` 的 Java 类的静态方法。

```
// example.kt
package demo

class Foo

fun bar() { …… }
```

```
// Java
new demo.Foo();
demo.ExampleKt.bar();
```

可以使用 `@JvmName` 注解修改生成的 Java 类的类名：

```kotlin
@file:JvmName("DemoUtils")

package demo

class Foo

fun bar() { ... }
```

```kotlin
// Java
new demo.Foo();
demo.DemoUtils.bar();
```

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。

```kotlin
// oldutils.kt
@file:JvmName("Utils")
@file:JvmMultifileClass

package demo

fun foo() { ... }
```

```kotlin
// newutils.kt
@file:JvmName("Utils")
@file:JvmMultifileClass

package demo

fun bar() { ... }
```

```kotlin
// Java
demo.Utils.foo();
demo.Utils.bar();
```

实例字段

如果需要在 Java 中将 Kotlin 属性作为字段暴露，那就需要使用 `@JvmField` 注解对其标注。该字段将具有与底层属性相同的可见性。如果一个属性有幕后字段（backing field）、非私有、没有 `open` / `override` 或者 `const` 修饰符并且不是被委托的属性，那么你可以用 `@JvmField` 注解该属性。

```kotlin
class C(id: String) {
    @JvmField val ID = id
}
```

```kotlin
// Java
class JavaClient {
    public String getID(C c) {
        return c.ID;
    }
}
```

延迟初始化的属性（在Java中）也会暴露为字段。该字段的可见性与 `lateinit` 属性的 setter 相同。

静态字段

在命名对象或伴生对象中声明的 Kotlin 属性会在该命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
```

在命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
```

用 `const` 标注的（在类中以及在顶层的）属性在 Java 中会成为静态字段：

```
// 文件 example.kt

object Obj {
    const val CONST = 1
}

class C {
    companion object {
        const val VERSION = 9
    }
}

const val MAX = 239
```

在 Java 中：

```
int c = Obj.CONST;
int d = ExampleKt.MAX;
int v = C.VERSION;
```

静态方法

如上所述，Kotlin 将包级函数表示为静态方法。Kotlin 还可以为命名对象或伴生对象中定义的函数生成静态方法，如果你将这些函数标注为 `@JvmStatic` 的话。如果你使用该注解，编译器既会在相应对象的类中生成静态方法，也会在对象自身中生成实例方法。例如：

```
class C {
    companion object {
        @JvmStatic fun foo() {}
        fun bar() {}
    }
}
```

现在，`foo()` 在 Java 中是静态的，而 `bar()` 不是：

```
C.foo(); // 没问题
C.bar(); // 错误:不是一个静态方法
C.Companion.foo(); // 保留实例方法
C.Companion.bar(); // 唯一的工作方式
```

对于命名对象也同样：

```
object Obj {
    @JvmStatic fun foo() {}
    fun bar() {}
}
```

在 Java 中：

```
Obj.foo(); // 没问题
Obj.bar(); // 错误
Obj.INSTANCE.bar(); // 没问题,通过单例实例调用
Obj.INSTANCE.foo(); // 也没问题
```

`@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。

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）；

— `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 中的命名函数在字节码中有另外一个 JVM 名称。最突出的例子是由于*类型擦除*引发的：

```
fun List<String>.filterValid(): List<String>
fun List<Int>.filterValid(): List<Int>
```

这两个函数不能同时定义，因为它们的 JVM 签名是一样的：`filterValid(Ljava/util/List;)Ljava/util/List;`。如果我们真的希望它们在 Kotlin 中用相同名称，我们需要用 `@JvmName` 去标注其中的一个（或两个），并指定不同的名称作为参数：

```
fun List<String>.filterValid(): List<String>

@JvmName("filterValidInt")
fun List<Int>.filterValid(): List<Int>
```

在 Kotlin 中它们可以用相同的名称 `filterValid` 来访问，而在 Java 中，它们分别是 `filterValid` 和 `filterValidInt`。

同样的技巧也适用于属性 `x` 和函数 `getX()` 共存：

```
val x: Int
    @JvmName("getX_prop")
    get() = 15

fun getX() = 10
```

如需在没有显式实现 getter 与 setter 的情况下更改属性生成的访问器方法的名称，可以使用 `@get:JvmName` 与 `@set:JvmName`：

```
@get:JvmName("x")
@set:JvmName("changeX")
var x: Int = 23
```

生成重载

通常，如果你写一个有默认参数值的 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 Foo @JvmOverloads constructor(x: Int, y: Double = 0.0) {
    @JvmOverloads fun f(a: String, b: Int = 0, c: String = "abc") { …… }
}
```

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:
Foo(int x, double y)
Foo(int x)

// 方法
void f(String a, int b, String c) { }
void f(String a, int b) { }
void f(String a) { }
```

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

如上所述，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。于是如果我们有一个这样的 Kotlin 函数：

```
// example.kt
package demo

fun foo() {
    throw IOException()
}
```

然后我们想要在 Java 中调用它并捕捉这个异常：

```
// Java
try {
  demo.Example.foo();
}
catch (IOException e) { // 错误:foo() 未在 throws 列表中声明 IOException
  // ……
}
```

因为 `foo()` 没有声明 `IOException`，我们从 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 foo() {
    throw IOException()
}
```

空安全性

当从 Java 中调用 Kotlin 函数时，没人阻止我们将 `null` 作为非空参数传递。这就是为什么 Kotlin 给所有期望非空参数的公有函数生成运行时检测。这样我们就能在 Java 代码里立即得到 `NullPointerException`。

型变的泛型

当 Kotlin 的类使用了声明处型变，有两种选择可以从 Java 代码中看到它们的用法。让我们假设我们有以下类和两个使用它的函数：

```
class Box<out T>(val value: T)

interface Base
class Derived : Base

fun boxDerived(value: Derived): Box<Derived> = Box(value)
fun unboxBase(box: Box<Base>): Base = box.value
```

一种看似理所当然地将这俩函数转换成 Java 代码的方式可能会是：

```
Box<Derived> boxDerived(Derived value) { …… }
Base unboxBase(Box<Base> box) { …… }
```

问题是，在 Kotlin 中我们可以这样写 `unboxBase(boxDerived("s"))`，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，我们按以下这样定义 `unboxBase`：

```
Base unboxBase(Box<? extends Base> box) { …… }
```

这里我们使用 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` 我们生成 `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，我们不生成通配符，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) { …… }
```

注意：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>`。

如果我们在默认不生成通配符的地方需要通配符，我们可以使用 `@JvmWildcard` 注解：

```
fun boxDerived(value: Derived): Box<@JvmWildcard Derived> = Box(value)
// 将被转换成
// Box<? extends Derived> boxDerived(Derived value) { …… }
```

另一方面，如果我们根本不需要默认的通配符转换，我们可以使用`@JvmSuppressWildcards`

```
fun unboxBase(box: Box<@JvmSuppressWildcards Base>): Base = box.value
// 会翻译成
// Base unboxBase(Box<Base> box) { …… }
```

注意：`@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() { …… }
```

JavaScript

动态类型

> 在面向 JVM 平台的代码中不支持动态类型

作为一种静态类型的语言，Kotlin仍然需要与无类型或松散类型的环境（例如 JavaScript生态系统）进行互操作。为了方便这些使用场景，语言中有 `dynamic` 类型可用：

```
val dyn: dynamic = ……
```

`dynamic` 类型基本上关闭了 Kotlin 的类型检查系统：

- 该类型的值可以赋值给任何变量或作为参数传递到任何位置；
- 任何值都可以赋值给 `dynamic` 类型的变量，或者传递给一个接受 `dynamic` 作为参数的函数；
- `null`-检查对这些值是禁用的。

`dynamic` 最特别的特性是，我们可以对 `dynamic` 变量调用**任何**属性或以任意参数调用**任何**函数：

```
dyn.whatever(1, "foo", dyn) // “whatever”在任何地方都没有定义
dyn.whatever(*arrayOf(1, 2, 3))
```

在 JavaScript 平台上，该代码将按照原样编译：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 注解为要调用的函数分配明确的名称。

动态调用总是返回 `dynamic` 作为结果，所以我们可以自由地这样链接调用：

```
dyn.foo().bar.baz()
```

当我们把一个 lambda 表达式传给一个动态调用时，它的所有参数默认都是 `dynamic` 类型的：

```
dyn.foo {
    x -> x.bar() // x 是 dynamic
}
```

使用 `dynamic` 类型值的表达式会按照原样转换为 JavaScript，并且不使用 Kotlin 运算符约定。支持以下运算符：

- 二元：`+`、`-`、`*`、`/`、`%`、`>`、`<`、`>=`、`<=`、`==`、`!=`、`===`、`!==`、`&&`、`||`
- 一元
  - 前置：`-`、`+`、`!`
  - 前置及后置：`++`、`--`
- 赋值：`+=`、`-=`、`*=`、`/=`、`%=`
- 索引访问：
  - 读：`d[a]`，多于一个参数会出错
  - 写：`d[a1] = a2`，`[]` 中有多于一个参数会出错

`in`、`!in` 以及 `..` 操作对于 `dynamic` 类型的值是禁用的。

更多技术说明请参见规范文档。

Kotlin 中调用 JavaScript

Kotlin 已被设计为能够与 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查类型。你可以通过动态类型在 Kotlin 中自由地与 JavaScript 交流，但是如果你想要 Kotlin 类型系统的全部威力，你可以为 JavaScript 库创建 Kotlin 头文件。

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")
}
```

`js` 的参数必须是字符串常量。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"
```

`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、函数或属性的实现由开发人员提供，因此不会尝试从声明中生成任何 JavaScript 代码。这意味着你应该省略 `external` 声明内容的代码体。例如：

```
external fun alert(message: Any?): Unit

external class Node {
    val firstChild: Node

    fun append(child: Node): Node

    fun removeChild(child: Node): Node

    // 等等
}

external val window: Window
```

请注意，嵌套的声明会继承 `external` 修饰符，即在 `Node` 类中，我们在成员函数和属性之前并不放置 `external`。

`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。即：

```
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。要描述来自上例中的 `MyClass`，你可以这样写：

```
external class MyClass {
    companion object {
        fun sharedMember()
    }

    fun ownMember()
}
```

声明可选参数

一个外部函数可以有可选参数。JavaScript 实现实际上如何计算这些参数的默认值，是 Kotlin 所不知道的，因此在 Kotlin 中不可能使用通常的语法声明这些参数。你应该使用以下语法：

```
external fun myFunWithOptionalArgs(x: Int,
    y: String = definedExternally,
    z: Long = definedExternally)
```

这意味着你可以使用一个必需参数和两个可选参数来调用 `myFunWithOptionalArgs`（它们的默认值由一些 JavaScript 代码算出）。

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` 类并用非 `external` 类扩展它。例如：

```
external open class HTMLElement : Element() {
    /* 成员 */
}

class CustomElement : HTMLElement() {
    fun foo() {
        alert("bar")
    }
}
```

有一些限制：

1. 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。
2. 不能覆盖一个使用默认参数的函数。

请注意，你无法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` 方法时，只需传递实际具有这些方法的对象。对于静态类型的 Kotlin，你可以使用接口来表达这点，例如：

```
external interface HasFooAndBar {
    fun foo()

    fun bar()
}

external fun myFunction(p: HasFooAndBar)
```

外部接口的另一个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：

1. 它们不能在 `is` 检查的右侧使用。
2. `as` 转换为外部接口总是成功（并在编译时产生警告）。
3. 它们不能作为具体化类型参数传递。
4. 它们不能用在类的字面值表达式（即 `I::class`）中。

JavaScript 中调用 Kotlin

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，你应该记住一些微妙的事情。

用独立的 JavaScript 隔离声明

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 。所以如果你把模块命名为 `myModule`，那么所有的声明都可以通过 `myModule` 对象在 JavaScript 中可用。例如：

```
fun foo() = "Hello"
```

可以在 JavaScript 中这样调用：

```
alert(myModule.foo());
```

这不适用于当你将 Kotlin 模块编译为 JavaScript 模块时（关于这点的详细信息请参见 JavaScript 模块）。在这种情况下，不会有一个包装对象，而是将声明作为相应类型的 JavaScript 模块对外暴露。例如，对于 CommonJS 的场景，你应该写：

```
alert(require('myModule').foo());
```

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 中使用完整限定的名称。例如：

```
package my.qualified.packagename

fun foo() = "Hello"
```

可以在 JavaScript 中这样调用：

```
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：

```
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，Kotlin 编译器不会对 `external` 声明应用这种修饰，因此你不必在其上使用 `@JsName`。值得注意的另一个例子是从外部类继承的非外部类。在这种情况下，任何被覆盖的函数也不会被修饰。

`@JsName` 的参数需要是一个常量字符串字面值，该字面值是一个有效的标识符。任何尝试将非标识符字符串传递给 `@JsName` 时，编译器都会报错。以下示例会产生编译期错误：

```
@JsName("new C()")   // 此处出错
external fun newC()
```

在 JavaScript 中表示 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），即以下代码能够工作：

  ```
  fun f() {
      val x: Int = 23
      val y: Any = x
      println(y as Float)
  }
  ```

- Kotlin 保留了 `kotlin.Int`、`kotlin.Byte`、`kotlin.Short`、`kotlin.Char` 和 `kotlin.Long` 的溢出语义。
- JavaScript 中没有 64 位整数，所以 `kotlin.Long` 没有映射到任何 JavaScript 对象，它是由一个 Kotlin 类模拟的。
- `kotlin.String` 映射到 JavaScript String。
- `kotlin.Any` 映射到 JavaScript Object（即 `new Object()`、`{}` 等）。

— `kotlin.Array` 映射到 JavaScript Array。

— Kotlin 集合（即 `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。

自 1.1.50 版起，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 UInt16Array

— `kotlin.LongArray` 会映射为 JavaScript 中具有 `$type$ == "LongArray"` 属性的 `kotlin.Long` 的数组。

JavaScript 模块

Kotlin 允许你将 Kotlin 项目编译为热门模块系统的 JavaScript 模块。以下是可用选项的列表：

1. 无模块（Plain）。不为任何模块系统编译。像往常一样，你可以在全局作用域中以其名称访问模块。默认使用此选项。
2. 异步模块定义（AMD，Asynchronous Module Definition），它尤其为 require.js 库所使用。
3. CommonJS 约定，广泛用于 node.js/npm（`require` 函数和 `module.exports` 对象）
4. 统一模块定义（UMD，Unified Module Definitions），它与 *AMD* 和 *CommonJS* 兼容，并且当在运行时 *AMD* 和 *CommonJS* 都不可用时，作为“plain”使用。

选择目标模块系统

选择目标模块系统的方式取决于你的构建环境：

在 IntelliJ IDEA 中

设置每个模块：打开“File → Project Structure…”，在“Modules”中找到你的模块并选择其下的“Kotlin”facet。在 “Module kind”字段中选择合适的模块系统。

为整个项目设置：打开“File → Settings”，选择“Build, Execution, Deployment”→“Compiler”→“Kotlin compiler”。在 “Module kind”字段中选择合适的模块系统。

在 Maven 中

要选择通过 Maven 编译时的模块系统，你应该设置 `moduleKind` 配置属性，即你的 `pom.xml` 应该看起来像这样：

```
<plugin>
    <artifactId>kotlin-maven-plugin</artifactId>
    <groupId>org.jetbrains.kotlin</groupId>
    <version>${kotlin.version}</version>
    <executions>
        <execution>
            <id>compile</id>
            <goals>
                <goal>js</goal>
            </goals>
        </execution>
    </executions>
    <!-- 插入这些行 -->
    <configuration>
        <moduleKind>commonjs</moduleKind>
    </configuration>
    <!-- 插入文本结束 -->
</plugin>
```

可用值包括：`plain`、`amd`、`commonjs`、`umd`。

在 Gradle 中

要选择通过 Gradle 编译时的模块系统，你应该设置 `moduleKind` 属性，即

```
compileKotlin2Js.kotlinOptions.moduleKind = "commonjs"
```

可用的值类似于 Maven。

@JsModule 注解

要告诉 Kotlin 一个 `external` 类、包、函数或者属性是一个 JavaScript 模块，你可以使用 `@JsModule` 注解。考虑你有以下 CommonJS 模块叫“hello”：

```
module.exports.sayHello = function(name) { alert("Hello, " + name); }
```

你应该在 Kotlin 中这样声明：

```
@JsModule("hello")
external fun sayHello(name: String)
```

将 @JsModule 应用到包

一些 JavaScript 库导出包（命名空间）而不是函数和类。从 JavaScript 角度讲，它是一个具有一些成员的对象，这些成员*是*类、函数和属性。将这些包作为 Kotlin 对象导入通常看起来不自然。编译器允许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 */ }
}
```

重要提示：标有 `@file:JsModule` 注解的文件无法声明非外部成员。下面的示例会产生编译期错误：

```
@file:JsModule("extModule")
package ext.jspackage.name

external fun foo()

fun bar() = "!" + foo() + "!" // 此处报错
```

导入更深的包层次结构

在前文示例中，JavaScript 模块导出单个包。但是，一些 JavaScript 库会从模块中导出多个包。Kotlin 也支持这种场景，尽管你必须为每个导入的包声明一个新的 `.kt` 文件。

例如，让我们的示例更复杂一些：

```
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

当一个声明具有 `@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，你可以将这些文件复制到项目的静态资源，并通过 `<script>` 元素包含。要告诉 Kotlin，可以在非模块环境中使用一个 `@JsModule` 声明，你应该放置 `@JsNonModule` 声明。例如，给定 JavaScript 代码：

```
function topLevelSayHello(name) { alert("Hello, " + name); }
if (module && module.exports) {
    module.exports = topLevelSayHello;
}
```

可以这样描述：

```
@JsModule("hello")
@JsNonModule
@JsName("topLevelSayHello")
external fun sayHello(name: String)
```

备注

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，因此你可以使用上述任何模块系统。也可以在 NPM 上使用 [kotlin 包]

JavaScript 反射

目前，JavaScript 不支持完整的 Kotlin 反射 API。唯一支持的该 API 部分是 `::class` 语法，它允许你引用一个实例的类或者与给定类型相对应的类。一个 `::class` 表达式的值是一个只能支持 simpleName 和 isInstance 成员的精简版 KClass 实现。

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。

示例：

```
class A
class B
class C

inline fun <reified T> foo() {
    println(T::class.simpleName)
}

val a = A()
println(a::class.simpleName)  // 获取一个实例的类;输出“A”
println(B::class.simpleName)  // 获取一个类型的类;输出“B”
println(B::class.js.name)     // 输出“B”
foo<C>()                      // 输出“C”
```

JavaScript DCE

自版本 1.1.4 起，Kotlin/JS 包含了一个无用代码消除（DCE，dead code elimination）工具。该工具允许在生成的 JS 中删除未使用的属性、函数和类。出现未使用的声明有这几种可能情况：

— 函数可以内联并且从未直接调用（除少数情况之外，这总会出现）。

— 你所使用的共享库提供了比实际需要更多的功能/函数。例如，标准库（`kotlin.js`）包含用于操作列表、数组、字符序列、DOM 适配器等的函数/功能，这些一起提供了大约 1.3 mb 的文件。一个简单的“Hello, world”应用程序只需要控制台程序，整个文件只有几千字节。

无用代码消除通常也称为“tree shaking”。

如何使用

DCE 工具目前对 Gradle 可用。

要激活 DCE 工具，请将以下这行添加到 `build.gradle` 中：

```
apply plugin: 'kotlin-dce-js'
```

请注意，如果你正在使用多项目构建，那么应该将插件应用在作为应用程序入口点的主项目。

默认情况下，可以在路径 `$BUILD_DIR/min/` 中找到生成的一组 JavaScript文件（你的应用程序与所有依赖关系），其中 `$BUILD_DIR` 是生成 JavaScript 的路径（通常是 `build/classes/main`）。

配置

要在主源集上配置 DCE，可以使用 `runDceKotlinJs` 任务（以及用于其他源集对应的 `runDce<sourceSetName>KotlinJs`）。

有时你直接在 JavaScript 中使用一个 Kotlin 声明，而被 DCE 给去除掉了。你可能想保留这个声明。为此，你可以在 `build.gradle` 中使用以下语法：

```
runDceKotlinJs.keep "declarationToKeep"[, "declarationToKeep", ...]
```

其中 `declarationToKeep` 具有以下语法：

```
moduleName.dot.separated.package.name.declarationName
```

例如，考虑一个模块命名为 `kotlin-js-example`，它在 `org.jetbrains.kotlin.examples` 包中包含一个名为 `toKeep` 的函数。使用以下这行：

```
runDceKotlinJs.keep "kotlin-js-example_main.org.jetbrains.kotlin.examples.toKeep"
```

请注意，如果函数具有参数，它的名称会被修饰，因此在 keep 指令中应该使用修饰后的名称。

开发模式

运行 DCE 在每次构建时会额外花费一些时间，而且输出大小在开发过程中无关紧要。可以通过 DCE 任务的 `dceOptions.devMode` 标志使 DCE 工具跳过实际的无效代码消除从而缩短开发构建时间。

例如，如需根据自定义条件禁用 `main` 源集的 DCE 并且总是禁用 `test` 代码的 DCE，请将下述几行添加到构建脚本中：

```
runDceKotlinJs.dceOptions.devMode = isDevMode
runDceTestKotlinJs.dceOptions.devMode = true
```

示例

显示如何将 Kotlin 与 DCE 及 webpack 集成并得到一个小的捆绑的完整示例，可以在这里找到。

注意事项

- 对于 1.1.x 版本，DCE 工具是一个 *实验性的* 特性。这并不意味着我们要删除它，或者它不能用于生产。这意味着我们可能更改配置参数的名称、默认设置等等。
- 目前，如果你的项目是共享库，那么不应使用 DCE 工具。它只适用于开发应用程序（可能使用共享库）时。原因是：DCE 不知道库的哪些部分会被用户的应用程序所使用。
- DCE 不会通过删除不必要的空格及缩短标识符来执行代码压缩（丑化）。对于此目的，你应该使用现有的工具，如 UglifyJS（https://github.com/mishoo/UglifyJS2）或者 Google Closure Compiler（https://developers.google.com/closure/compiler/）。

原生

Kotlin/Native 中的并发

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 with message passing
- Object subgraph ownership transfer
- Object subgraph freezing
- Object subgraph detachment
- Raw shared memory using C globals
- Coroutines for blocking operations (not covered in this document)

Workers

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 (i.e. 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, program will likely crash unexpectedly, so consider that last resort in optimizing, not a general purpose mechanism.

For a more complete example please refer to the workers example in the Kotlin/Native repository.

Object transfer and freezing

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 which has no 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 - 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 autofreezing 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. Combining it with raw memory sharing it allows side channel object transfer between concurrent threads, if the worker mechanisms are insufficient for a particular task.

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 in it references to detached object subgraphs. Consider the following .def file:

```
package = global

---
typedef struct {
  int version;
  void* kotlinObject;
} SharedData;

SharedData sharedData;
```

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 threads keeps 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 threads 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

Combined, these mechanisms allow natural race-freeze programming with code reuse across platforms in MPP projects.

Kotlin/Native 中的不可变性

Kotlin/Native implements strict mutability checks, ensuring the important invariant that the object is either immutable or accessible from the single thread at that moment in time (`mutable XOR global`).

Immutability is a runtime property in Kotlin/Native, and can be applied to an arbitrary object subgraph using the `kotlin.native.concurrent.freeze` function. It makes all the objects reachable from the given one immutable, such a transition is a one-way operation (i.e., objects cannot be unfrozen later). Some naturally immutable objects such as `kotlin.String`, `kotlin.Int`, and other primitive types, along with `AtomicInt` and `AtomicReference` are frozen by default. If a mutating operation is applied to a frozen object, an `InvalidMutabilityException` is thrown.

To achieve `mutable XOR global` invariant, all globally visible state (currently, `object` singletons and enums) are automatically frozen. If object freezing is not desired, a `kotlin.native.ThreadLocal` annotation can be used, which will make the object state thread local, and so, mutable (but the changed state is not visible to other threads).

Top level/global variables of non-primitive types are by default accessible in the main thread (i.e., the thread which initialized *Kotlin/Native* runtime first) only. Access from another thread will lead to an `IncorrectDereferenceException` being thrown. To make such variables accessible in other threads, you can use either the `@ThreadLocal` annotation, and mark the value thread local or `@SharedImmutable`, which will make the value frozen and accessible from other threads.

Class `AtomicReference` can be used to publish the changed frozen state to other threads, and so build patterns like shared caches.

Kotlin/Native 库

Kotlin compiler specifics

To produce a library with the Kotlin/Native compiler use the `-produce library` or `-p library` flag. For example:

```
$ kotlinc foo.kt -p library -o bar
```

the above command will produce a `bar.klib` with the compiled contents of `foo.kt`.

To link to a library use the `-library <name>` or `-l <name>` flag. For example:

```
$ kotlinc qux.kt -l bar
```

the above command will produce a `program.kexe` out of `qux.kt` and `bar.klib`

cinterop tool specifics

The **cinterop** tool produces `.klib` wrappers for native libraries as its main output. For example, using the simple `libgit2.def` native library definition file provided in your Kotlin/Native distribution

```
$ cinterop -def  samples/gitchurn/src/main/c_interop/libgit2.def -compilerOpts -
I/usr/local/include -o libgit2
```

we will obtain `libgit2.klib`.

See more details in INTEROP.md

klib utility

The **klib** library management utility allows you to inspect and install the libraries.

The following commands are available.

To list library contents:

```
$ klib contents <name>
```

To inspect the bookkeeping details of the library

```
$ klib info <name>
```

To install the library to the default location use

```
$ klib install <name>
```

To remove the library from the default repository use

```
$ klib remove <name>
```

All of the above commands accept an additional `-repository <directory>` argument for specifying a repository different to the default one.

```
$ klib <command> <name> -repository <directory>
```

Several examples

First let's create a library. Place the tiny library source code into `kotlinizer.kt`:

```
package kotlinizer
val String.kotlinized
    get() = "Kotlin $this"
```

```
$ kotlinc kotlinizer.kt -p library -o kotlinizer
```

The library has been created in the current directory:

```
$ ls kotlinizer.klib
kotlinizer.klib
```

Now let's check out the contents of the library:

```
$ klib contents kotlinizer
```

We can install `kotlinizer` to the default repository:

```
$ klib install kotlinizer
```

Remove any traces of it from the current directory:

```
$ rm kotlinizer.klib
```

Create a very short program and place it into a `use.kt`:

```
import kotlinizer.*

fun main(args: Array<String>) {
    println("Hello, ${"world".kotlinized}!")
}
```

Now compile the program linking with the library we have just created:

```
$ kotlinc use.kt -l kotlinizer -o kohello
```

And run the program:

```
$ ./kohello.kexe
Hello, Kotlin world!
```

Have fun!

Advanced topics

Library search sequence

When given a `-library foo` flag, the compiler searches the `foo` library in the following order:

```
* Current compilation directory or an absolute path.

* All repositories specified with `-repo` flag.

* Libraries installed in the default repository (For now the default is  `~/.konan`, however it could be changed by setting **KONAN_DATA_DIR** environment variable).

* Libraries installed in `$installation/klib` directory.
```

The library format

Kotlin/Native libraries are zip files containing a predefined directory structure, with the following layout:

foo.klib when unpacked as **foo/** gives us:

```
- foo/
  - targets/
    - $platform/
      - kotlin/
        - Kotlin compiled to LLVM bitcode.
      - native/
        - Bitcode files of additional native objects.
    - $another_platform/
      - There can be several platform specific kotlin and native pairs.
  - linkdata/
    - A set of ProtoBuf files with serialized linkage metadata.
  - resources/
    - General resources such as images. (Not used yet).
  - manifest - A file in *java property* format describing the library.
```

An example layout can be found in `klib/stdlib` directory of your installation.

平台库

Overview

To provide access to user's native operating system services, `Kotlin/Native` distribution includes a set of prebuilt libraries specific to each target. We call them **Platform Libraries**.

POSIX bindings

For all `Unix` or `Windows` based targets (including `Android` and `iPhone`) we provide the `posix` platform lib. It contains bindings to platform's implementation of `POSIX` standard.

To use the library just

```
import platform.posix.*
```

The only target for which it is not available is WebAssembly.

Note that the content of `platform.posix` is NOT identical on different platforms, in the same way as different `POSIX` implementations are a little different.

Popular native libraries

There are many more platform libraries available for host and cross-compilation targets. `Kotlin/Native` distribution provides access to `OpenGL`, `SDL`, `zlib` and other popular native libraries on applicable platforms.

On Apple platforms `objc` library is provided for interoperability with Objective-C.

Inspect the contents of `dist/klib/platform/$target` of the distribution for the details.

Availability by default

The packages from platform libraries are available by default. No special link flags need to be specified to use them. `Kotlin/Native` compiler automatically detects which of the platform libraries have been accessed and automatically links the needed libraries.

On the other hand, the platform libs in the distribution are merely just wrappers and bindings to the native libraries. That means the native libraries themselves (`.so`, `.a`, `.dylib`, `.dll` etc) should be installed on the machine.

Examples

`Kotlin/Native` installation provides a wide spectrum of examples demonstrating the use of platform libraries. See samples for details.

Kotlin/Native 互操作

Introduction

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.

— create a `.def` file describing what to include into bindings

— use the `cinterop` tool to produce Kotlin bindings

— run *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 a separate document OBJC_INTEROP.md.

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 standartized 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/main/c_interop/libgit2.def \
 -compilerOpts -I/usr/local/include -o libgit2
```

Compile the client:

```
../../dist/bin/kotlinc src/main/kotlin \
 -library libgit2 -o GitChurn
```

Run the client:

```
./GitChurn.kexe ../..
```

Creating bindings for a new library

To create bindings for a new library, start by 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 -compilerOpts -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).

The output of a config script with `--libs` will be passed as a `-linkedArgs` `kotlinc` flag value (quoted) when compiling.

Selecting 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.

Filtering 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 <SomeLbrary/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.

Filtering by module maps

Some libraries have proper `module.modulemap` or `module.map` files in its 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:

```
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 = -DFOO=bar
linkerOpts = -lpng
```

Target-specific options, only applicable to the certain target can be specified as well, such as

```
 compilerOpts = -DBAR=bar
 compilerOpts.linux_x64 = -DFOO=foo1
 compilerOpts.mac_x64 = -DFOO=foo2
```

and so, C headers on Linux will be analyzed with `-DBAR=bar -DFOO=foo1` and on macOS with `-DBAR=bar -DFOO=foo2`. Note that any definition file option can have both common and the platform-specific part.

Adding 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;
}
```

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.

Including 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:

```
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.

Using 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 are mapped to types having fields available via the dot notation, i.e. `someStructInstance.field1`.

— `typedef` are represented as `typealias`.

Also, any C type has the Kotlin type representing the lvalue of this type, i.e., the value located in memory rather than a simple immutable self-contained value. Think C++ references, as a similar concept. For structs (and `typedef`s to structs) this representation is the main one and has the same name as the struct itself, for Kotlin enums it 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

```
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.

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
}
```

Passing 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)
```

Working with the strings

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.cstr.ptr }
    menu = new_menu("Menu".cstr.ptr, items.toCValues().ptr)
    ...
}
```

In this example, all values passed to the C API `new_menu()` have a lifetime of the innermost `memScope` it belongs to. Once the control flow leaves the `memScoped` scope the C pointers become invalid.

Passing and receiving structs by value

When a C function takes or returns a struct `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.

Note that some function types are not supported currently. For example, it is not possible to get a pointer to a function that receives or returns structs by value.

If the callback doesn't run in the main thread, it is mandatory to init the *Kotlin/Native* runtime by calling `kotlin.native.initRuntimeIfNeeded()`.

Passing 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:

```
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 `FOO` 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 FOO(arg);
}
```

Definition file hints

The `.def` file supports several options for adjusting the generated bindings.

— `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

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).toInt()

            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.

Kotlin/Native 与 Swift/Objective-C 互操作

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 e.g. "Using cinterop" in Gradle plugin documentation. 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 "Targets and output kinds" section in Gradle plugin documentation). See calculator sample for an example.

Mappings

The table below shows how Kotlin concepts are mapped to Swift/Objective-C and vice versa.

Kotlin	Swift	Objective-C	Notes
`class`	`class`	`@interface`	note
`interface`	`protocol`	`@protocol`	
`constructor/create`	Initializer	Initializer	note
Property	Property	Property	note
Method	Method	Method	note note
`@Throws`	`throws`	`error:(NSError**)error`	note
Extension	Extension	Category member	note
`companion` member <-	Class method or property	Class method or property	
`null`	`nil`	`nil`	
`Singleton`	`Singleton()`	`[Singleton singleton]`	note
Primitive type	Primitive type / `NSNumber`		note
`Unit` return type	`Void`	`void`	
`String`	`String`	`NSString`	
`String`	`NSMutableString`	`NSMutableString`	note
`List`	`Array`	`NSArray`	
`MutableList`	`NSMutableArray`	`NSMutableArray`	
`Set`	`Set`	`NSSet`	
`MutableSet`	`NSMutableSet`	`NSMutableSet`	note
`Map`	`Dictionary`	`NSDictionary`	
`MutableMap`	`NSMutableDictionary`	`NSMutableDictionary`	note
Function type	Function type	Block pointer type	note

Name translation

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.

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.

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

```
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. In this case all Kotlin exceptions (except for instances of `Error`, `RuntimeException` and subclasses) are translated into a Swift error/`NSError`.

Note that the opposite reversed translation is not implemented yet: Swift/Objective-C error-throwing methods aren't imported to Kotlin as exception-throwing.

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 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 factory method, i.e. as `[MySingleton mySingleton]` in Objective-C and `MySingleton()` in Swift.

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 `${prefix}Int` 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. `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

```
foo {
    bar($0 as! Int32)
    return KotlinUnit()
}
```

Casting between mapped types

When writing Kotlin code, an object may need to be converted from a Kotlin type to the equivalent Swift/Objective-C type (or vice versa). In this case a plain old Kotlin cast can be used, 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. Currently a class that adopts the Kotlin protocol should inherit `NSObject` (either directly or indirectly). Note that all Kotlin classes do inherit `NSObject`, so a Swift/Objective-C subclass of Kotlin class can adopt the Kotlin protocol.

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.

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 INTEROP.md for an example case where the library uses some plain C features (e.g. unsafe pointers, structs etc.).

Kotlin/Native Gradle 插件

IMPORTANT NOTICE

This document describes Kotlin/Native experimental Gradle plugin, which is not the plugin yet supported by IDE or in multiplatform projects. See MPP Gradle plugin documentation for more information.

Overview

You may use the Gradle plugin to build *Kotlin/Native* projects. Builds of the plugin are available at the Gradle plugin portal, so you can apply it using Gradle plugin DSL:

```
plugins {
    id "org.jetbrains.kotlin.platform.native" version "1.3.0-rc-146"
}
```

You also can get the plugin from a Bintray repository. In addition to releases, this repo contains old and development versions of the plugin which are not available at the plugin portal. To get the plugin from the Bintray repo, include the following snippet in your build script:

```
buildscript {
   repositories {
       mavenCentral()
       maven {
           url "https://dl.bintray.com/jetbrains/kotlin-native-dependencies"
       }
   }

   dependencies {
       classpath "org.jetbrains.kotlin:kotlin-native-gradle-plugin:1.3.0-rc-146"
   }
}

apply plugin: 'org.jetbrains.kotlin.platform.native'
```

By default the plugin downloads the Kotlin/Native compiler during the first run. If you have already downloaded the compiler manually you can specify the path to its root directory using `konan.home` project property (e.g. in `gradle.properties`).

```
konan.home=/home/user/kotlin-native-0.8
```

In this case the compiler will not be downloaded by the plugin.

Source management

Source management in the `kotlin.platform.native` plugin is uniform with other Kotlin plugins and is based on source sets. A source set is a group of Kotlin/Native source which may contain both common and platform-specific code. The plugin provides a top-level script block `sourceSets` allowing you to configure source sets. Also it creates the default source sets `main` and `test` (for production and test code respectively).

By default the production sources are located in `src/main/kotlin` and the test sources - in `src/test/kotlin`.

```
sourceSets {
    // Adding target-independent sources.
    main.kotlin.srcDirs += 'src/main/mySources'

    // Adding Linux-specific code. It will be compiled in Linux binaries only.
    main.target('linux_x64').srcDirs += 'src/main/linux'
}
```

Targets and output kinds

By default the plugin creates software components for the main and test source sets. You can access them via the `components` container provided by Gradle or via the `component` property of a corresponding source set:

```
// Main component.
components.main
sourceSets.main.component

// Test component.
components.test
sourceSets.test.component
```

Components allow you to specify:

— Targets (e.g. Linux/x64 or iOS/arm64 etc)
— Output kinds (e.g. executable, library, framework etc)
— Dependencies (including interop ones)

Targets can be specified by setting a corresponding component property:

```
components.main {
    // Compile this component for 64-bit MacOS, Linux and Windows.
    targets = ['macos_x64', 'linux_x64', 'mingw_x64']
}
```

The plugin uses the same notation as the compiler. By default, test component uses the same targets as specified for the main one.

Output kinds can also be specified using a special property:

```
components.main {
    // Compile the component into an executable and a Kotlin/Native library.
    outputKinds = [EXECUTABLE, KLIBRARY]
}
```

All constants used here are available inside a component configuration script block. The plugin supports producing binaries of the following kinds:

— `EXECUTABLE` - an executable file;

— `KLIBRARY` - a Kotlin/Native library (*.klib);

— `FRAMEWORK` - an Objective-C framework;

— `DYNAMIC` - shared native library;

— `STATIC` - static native library.

Also each native binary is built in two variants (build types): `debug` (debuggable, not optimized) and `release` (not debuggable, optimized). Note that Kotlin/Native libraries have only `debug` variant because optimizations are preformed only during compilation of a final binary (executable, static lib etc) and affect all libraries used to build it.

Compile tasks

The plugin creates a compilation task for each combination of the target, output kind, and build type. The tasks have the following naming convention:

`compile<ComponentName><BuildType><OutputKind><Target>KotlinNative`

For example `compileDebugKlibraryMacos_x64KotlinNative`, `compileTestDebugKotlinNative`.

The name contains the following parts (some of them may be empty):

— `<ComponentName>` - name of a component. Empty for the main component.

— `<BuildType>` - `Debug` or `Release`.

— `<OutputKind>` - output kind name, e.g. `Executabe` or `Dynamic`. Empty if the component has only one output kind.

— `<Target>` - target the component is built for, e.g. `Macos_x64` or `Wasm32`. Empty if the component is built only for one target.

Also the plugin creates a number of aggregate tasks allowing you to build all the binaries for a build type (e.g. `assembleAllDebug`) or all the binaries for a particular target (e.g. `assembleAllWasm32`).

Basic lifecycle tasks like `assemble`, `build`, and `clean` are also available.

Running tests

The plugin builds a test executable for all the targets specified for the `test` component. If the current host platform is included in this list the test running tasks are also created. To run tests, execute the standard lifecycle `check` task:

```
./gradlew check
```

Dependencies

The plugin allows you to declare dependencies on files and other projects using traditional Gradle's mechanism of configurations. The plugin supports Kotlin multiplatform projects allowing you to declare the `expectedBy` dependencies

```
dependencies {
    implementation files('path/to/file/dependencies')
    implementation project('library')
    testImplementation project('testLibrary')
    expectedBy project('common')
}
```

It's possible to depend on a Kotlin/Native library published earlier in a maven repo. The plugin relies on Gradle's metadata support so the corresponding feature must be enabled. Add the following line in your `settings.gradle`:

```
enableFeaturePreview('GRADLE_METADATA')
```

Now you can declare a dependency on a Kotlin/Native library in the traditional `group:artifact:version` notation:

```
dependencies {
    implementation 'org.sample.test:mylibrary:1.0'
    testImplementation 'org.sample.test:testlibrary:1.0'
}
```

Dependency declaraion is also possible in the component block:

```
components.main {
    dependencies {
        implementation 'org.sample.test:mylibrary:1.0'
    }
}

components.test {
    dependencies {
        implementation 'org.sample.test:testlibrary:1.0'
    }
}
```

Using cinterop

It's possible to declare a cinterop dependency for a component:

```
components.main {
    dependencies {
        cinterop('mystdio') {
            // src/main/c_interop/mystdio.def is used as a def file.

            // Set up compiler options
            compilerOpts '-I/my/include/path'

            // It's possible to set up different options for different targets
            target('linux') {
                compilerOpts '-I/linux/include/path'
            }
        }
    }
}
```

Here an interop library will be built and added in the component dependencies.

Often it's necessary to specify target-specific linker options for a Kotlin/Native binary using an interop. It can be done using the `target` script block:

```
components.main {
    target('linux') {
        linkerOpts '-L/path/to/linux/libs'
    }
}
```

Also the `allTargets` block is available.

```
components.main {
    // Configure all targets.
    allTargets {
        linkerOpts '-L/path/to/libs'
    }
}
```

Publishing

In the presence of `maven-publish` plugin the publications for all the binaries built are created. The plugin uses Gradle metadata to publish the artifacts so this feature must be enabled (see the dependencies section).

Now you can publish the artifacts with the standard Gradle `publish` task:

```
./gradlew publish
```

Only `EXECUTABLE` and `KLIBRARY` binaries are published currently.

The plugin allows you to customize the pom generated for the publication with the `pom` code block available for every component:

```
components.main {
    pom {
        withXml {
            def root = asNode()
            root.appendNode('name', 'My library')
            root.appendNode('description', 'A Kotlin/Native library')
        }
    }
}
```

Serialization plugin

The plugin is shipped with a customized version of the `kotlinx.serialization` plugin. To use it you don't have to add new buildscript dependencies, just apply the plugins and add a dependency on the serialization library:

```
apply plugin: 'org.jetbrains.kotlin.platform.native'
apply plugin: 'kotlinx-serialization-native'

dependencies {
    implementation 'org.jetbrains.kotlinx:kotlinx-serialization-runtime-native'
}
```

The the example project for details.

DSL example

In this section a commented DSL is shown. See also the example projects that use this plugin, e.g. Kotlinx.coroutines, MPP http client

```
plugins {
    id "org.jetbrains.kotlin.platform.native" version "1.3.0-rc-146"
}

sourceSets.main {
    // Plugin uses Gradle's source directory sets here,
    // so all the DSL methods available in SourceDirectorySet can be called here.
    // Platform independent sources.
    kotlin.srcDirs += 'src/main/customDir'

    // Linux-specific sources
    target('linux').srcDirs += 'src/main/linux'
}

components.main {

    // Set up targets
    targets = ['linux_x64', 'macos_x64', 'mingw_x64']

    // Set up output kinds
    outputKinds = [EXECUTABLE, KLIBRARY, FRAMEWORK, DYNAMIC, STATIC]

    // Specify custom entry point for executables
    entryPoint = "org.test.myMain"
```

```
    // Target-specific options
    target('linux_x64') {
        linkerOpts '-L/linux/lib/path'
    }

    // Targets independent options
    allTargets {
        linkerOpts '-L/common/lib/path'
    }

    dependencies {

        // Dependency on a published Kotlin/Native library.
        implementation 'org.test:mylib:1.0'

        // Dependency on a project
        implementation project('library')

        // Cinterop dependency
        cinterop('interop-name') {
            // Def-file describing the native API.
            // The default path is src/main/c_interop/<interop-name>.def
            defFile project.file("deffile.def")

            // Package to place the Kotlin API generated.
            packageName 'org.sample'

            // Options to be passed to compiler and linker by cinterop tool.
            compilerOpts 'Options for native stubs compilation'
            linkerOpts 'Options for native stubs'

            // Additional headers to parse.
            headers project.files('header1.h', 'header2.h')

            // Directories to look for headers.
            includeDirs {
                // All objects accepted by the Project.file method may be used with both
options.

                // Directories for header search (an analogue of the -I<path> compiler
option).
                allHeaders 'path1', 'path2'

                // Additional directories to search headers listed in the 'headerFilter'
def-file option.
                // -headerFilterAdditionalSearchPrefix command line option analogue.
                headerFilterOnly 'path1', 'path2'
            }
            // A shortcut for includeDirs.allHeaders.
            includeDirs "include/directory" "another/directory"

            // Pass additional command line options to the cinterop tool.
            extraOpts '-shims', 'true'

            // Additional configuration for Linux.
            target('linux') {
                compilerOpts 'Linux-specific options'
            }
        }
```

```
    }

    // Additional pom settings for publication.
    pom {
        withXml {
            def root = asNode()
            root.appendNode('name', 'My library')
            root.appendNode('description', 'A Kotlin/Native library')
        }
    }

    // Additional options passed to the compiler.
    extraOpts '--time'
}
```

调试

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

Producing binaries with debug info with Kotlin/Native compiler

To produce binaries with the Kotlin/Native compiler it's sufficient to use the `-g` option on the command line.
Example:

```
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
hello.kt:2, address = 0x00000001000012e4
(lldb) r
Process 28473 launched: '/Users/minamoto/ws/.git-trees/debugger-fixes/terminator.kexe'
(x86_64)
Process 28473 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = breakpoint 1.1
    frame #0: 0x00000001000012e4 terminator.kexe`kfun:main(kotlin.Array<kotlin.String>) at
hello.kt:2
   1    fun main(args: Array<String>) {
-> 2      println("Hello world")
   3      println("I need your clothes, your boots and your motocycle")
   4    }
(lldb) n
Hello world
Process 28473 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = step over
    frame #0: 0x00000001000012f0 terminator.kexe`kfun:main(kotlin.Array<kotlin.String>) at
hello.kt:3
   1    fun main(args: Array<String>) {
   2      println("Hello world")
-> 3      println("I need your clothes, your boots and your motocycle")
   4    }
(lldb)
```

Breakpoints

Modern debuggers provide several ways to set a breakpoint, see below for a tool-by-tool breakdown:

lldb

— by name

```
(lldb) b -n kfun:main(kotlin.Array<kotlin.String>)
Breakpoint 4: where = terminator.kexe`kfun:main(kotlin.Array<kotlin.String>) + 4 at
hello.kt:2, address = 0x00000001000012e4
```

-n is optional, this flag is applied by default

— by location (filename, line number)

```
(lldb) b -f hello.kt -l 1
Breakpoint 1: where = terminator.kexe`kfun:main(kotlin.Array<kotlin.String>) + 4 at
hello.kt:2, address = 0x00000001000012e4
```

— by address

```
(lldb) b -a 0x00000001000012e4
Breakpoint 2: address = 0x00000001000012e4
```

— by regex, you might find it useful for debugging generated artifacts, like lambda etc. (where used `#` symbol in name).

```
3: regex = 'main\(', locations = 1
  3.1: where = terminator.kexe`kfun:main(kotlin.Array<kotlin.String>) + 4 at hello.kt:2,
address = terminator.kexe[0x00000001000012e4], unresolved, hit count = 0
```

gdb

— by regex

```
(gdb) rbreak main(
Breakpoint 1 at 0x1000109b4
struct ktype:kotlin.Unit &kfun:main(kotlin.Array<kotlin.String>);
```

— by name **unusable**, because `:` is a separator for the breakpoint by location

```
(gdb) b kfun:main(kotlin.Array<kotlin.String>)
No source file named kfun.
Make breakpoint pending on future shared library load? (y or [n]) y
Breakpoint 1 (kfun:main(kotlin.Array<kotlin.String>)) pending
```

— by location

```
(gdb) b hello.kt:1
Breakpoint 2 at 0x100001704: file /Users/minamoto/ws/.git-trees/hello.kt, line 1.
```

— by address

```
(gdb) b *0x100001704
Note: breakpoint 2 also set at pc 0x100001704.
Breakpoint 3 at 0x100001704: file /Users/minamoto/ws/.git-trees/hello.kt, line 2.
```

Stepping

Stepping functions works mostly the same way as for C/C++ programs

Variable inspection

Variable inspections for var variables works out of the box for primitive types. For non-primitive types there are custom pretty printers for lldb in `konan_lldb.py`:

```
λ cat main.kt | nl
     1  fun main(args: Array<String>) {
     2      var x = 1
     3      var y = 2
     4      var p = Point(x, y)
     5      println("p = $p")
     6  }

     7  data class Point(val x: Int, val y: Int)

λ lldb ./program.kexe -o 'b main.kt:5' -o
(lldb) target create "./program.kexe"
Current executable set to './program.kexe' (x86_64).
(lldb) b main.kt:5
Breakpoint 1: where = program.kexe`kfun:main(kotlin.Array<kotlin.String>) + 289 at
main.kt:5, address = 0x000000000040af11
(lldb) r
Process 4985 stopped
```

```
* thread #1, name = 'program.kexe', stop reason = breakpoint 1.1
    frame #0: program.kexe`kfun:main(kotlin.Array<kotlin.String>) at main.kt:5
   2        var x = 1
   3        var y = 2
   4        var p = Point(x, y)
-> 5        println("p = $p")
   6    }
   7
   8    data class Point(val x: Int, val y: Int)

Process 4985 launched: './program.kexe' (x86_64)
(lldb) fr var
(int) x = 1
(int) y = 2
(ObjHeader *) p = 0x00000000007643d8
(lldb) command script import dist/tools/konan_lldb.py
(lldb) fr var
(int) x = 1
(int) y = 2
(ObjHeader *) p = Point(x=1, y=2)
(lldb) p p
(ObjHeader *) $2 = Point(x=1, y=2)
(lldb)
```

Getting representation of the object variable (var) could also be done using the built-in runtime function `Konan_DebugPrint` (this approach also works for gdb, using a module of command syntax):

```
0:b-debugger-fixes:minamoto@unit-703(0)# cat ../debugger-plugin/1.kt | nl -p
     1  fun foo(a:String, b:Int) = a + b
     2  fun one() = 1
     3  fun main(arg:Array<String>) {
     4    var a_variable = foo("(a_variable) one is ", 1)
     5    var b_variable = foo("(b_variable) two is ", 2)
     6    var c_variable = foo("(c_variable) two is ", 3)
     7    var d_variable = foo("(d_variable) two is ", 4)
     8    println(a_variable)
     9    println(b_variable)
    10    println(c_variable)
    11    println(d_variable)
    12  }
0:b-debugger-fixes:minamoto@unit-703(0)# lldb ./program.kexe -o 'b -f 1.kt -l 9' -o r
(lldb) target create "./program.kexe"
Current executable set to './program.kexe' (x86_64).
(lldb) b -f 1.kt -l 9
Breakpoint 1: where = program.kexe`kfun:main(kotlin.Array<kotlin.String>) + 463 at 1.kt:9,
address = 0x0000000100000dbf
(lldb) r
(a_variable) one is 1
Process 80496 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = breakpoint 1.1
    frame #0: 0x0000000100000dbf program.kexe`kfun:main(kotlin.Array<kotlin.String>) at
1.kt:9
   6      var c_variable = foo("(c_variable) two is ", 3)
   7      var d_variable = foo("(d_variable) two is ", 4)
   8      println(a_variable)
-> 9      println(b_variable)
   10     println(c_variable)
   11     println(d_variable)
   12   }

Process 80496 launched: './program.kexe' (x86_64)
(lldb) expression -- Konan_DebugPrint(a_variable)
(a_variable) one is 1(KInt) $0 = 0
(lldb)
```

Known issues

— performance of Python bindings.

Note: 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.

Q: How do I run my program?

A: 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.

Q: What is Kotlin/Native memory management model?

A: 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.

Q: How do I create a shared library?

A: Use the `-produce dynamic` compiler switch, or `compilations.main.outputKinds 'DYNAMIC'` in Gradle, i.e.

```
targets {
    fromPreset(presets.iosArm64, 'mylib') {
       compilations.main.outputKinds 'DYNAMIC'
    }
}
```

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 `samples/python_extension` for an example of using such a shared object to provide a bridge between Python and Kotlin/Native.

Q: How do I create a static library or an object file?

A: Use the `-produce static` compiler switch, or `compilations.main.outputKinds 'STATIC'` in Gradle, i.e.

```
targets {
    fromPreset(presets.iosArm64, 'mylib') {
       compilations.main.outputKinds 'STATIC'
    }
}
```

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.

Q: How do I run Kotlin/Native behind a corporate proxy?

A: 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.

Q: How do I specify a custom Objective-C prefix/name for my Kotlin framework?

A: Use the `-module_name` compiler option or matching Gradle DSL statement, i.e.

```
targets {
    fromPreset(presets.iosArm64, 'myapp') {
       compilations.main.outputKinds 'FRAMEWORK'
       compilations.main.extraOpts '-module_name', 'TheName'
    }
}
```

Q: How do I enable bitcode for my Kotlin framework?

A: Use either `-Xembed-bitcode` or `-Xembed-bitcode-marker` compiler option or matching Gradle DSL statement, i.e.

```
targets {
    fromPreset(presets.iosArm64, 'myapp') {
       compilations.main.outputKinds 'FRAMEWORK'
       compilations.main.extraOpts '-Xembed-bitcode' // for release binaries
       // or '-Xembed-bitcode-marker' for debug binaries
}
```

These options have nearly the same effect as clang's `-fembed-bitcode` / `-fembed-bitcode-marker` and swiftc's `-embed-bitcode` / `-embed-bitcode-marker`.

Q: Why do I see `InvalidMutabilityException`?

A: 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).

Q: How do I make a singleton object mutable?

A: 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.

Q: How can I compile my project against the Kotlin/Native master?

A: We release dev builds frequently, usually at least once a week. You can check the list of available versions. But if we recently fixed an issue and you want to check it before a release is done, you can do:

- For the CLI, you can compile using gradle as stated in the README (and if you get errors, you can try to do a `./gradlew clean`):
- For Gradle, you can use Gradle composite builds like this:

协程

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

其他参考资料

— 使用协程进行 UI 编程指南

— 响应式流与协程指南

— 协程设计文档（KEEP）

— 完整的 kotlinx.coroutines API 参考文档

目录

协程基础

这一部分包括基础的协程概念。

你的第一个协程程序

运行以下代码：

```
import kotlinx.coroutines.*

fun main() {
    GlobalScope.launch { // 在后台启动一个新的协程并继续
        delay(1000L) // 无阻塞的等待1秒钟(默认时间单位是毫秒)
        println("World!") // 在延迟后打印输出
    }
    println("Hello,") // 主线程的协程将会继续等待
    Thread.sleep(2000L) // 阻塞主线程2秒钟来保证 JVM 存活
}
```

> 你可以点击这里获得完整代码

代码运行的结果：

```
Hello,
World!
```

本质上，协程是轻量级的线程。它们在 CoroutineScope 上下文中和 launch *协程构建器* 一起被启动。这里我们在 GlobalScope 中启动了一些新的协程，存活时间是指新的协程的存活时间被限制在了整个应用程序的存活时间之内。

你可以使用一些协程操作来替换一些线程操作，比如：用 `GlobalScope.launch { …… }` 替换 `thread { …… }` 用 `delay(……)` 替换 `Thread.sleep(……)` 。尝试一下。

如果你开始使用 `GlobalScope.launch` 来替换 `thread`，编译器将会抛出错误：

```
Error: Kotlin: Suspend functions are only allowed to be called from a coroutine or another suspend function
```

这是因为 delay 是一个特别的 *挂起函数*，它不会造成线程阻塞，但是 *挂起* 函数只能在协程中使用。

桥接阻塞与非阻塞的世界

第一个例子中在同一段代码中包含了 *非阻塞的* `delay(……)` 和 *阻塞的* `Thread.sleep(……)`。这非常容易让我们记混哪个是阻塞的、哪个是非阻塞的。来一起使用显式的阻塞 runBlocking 协程构建器：

```
import kotlinx.coroutines.*

fun main() {
    GlobalScope.launch { // 在后台启动一个新的协程并继续
        delay(1000L)
        println("World!")
    }
    println("Hello,") // 主线程中的代码会立即执行
    runBlocking {     // 但是这个函数阻塞了主线程
        delay(2000L)  // ……我们延迟2秒来保证 JVM 的存活
    }
}
```

> 你可以点击这里来获得完整代码

结果是相似的，但是这些代码只使用了非阻塞的函数 delay。在主线程中调用了 `runBlocking`，*阻塞* 会持续到 `runBlocking` 中的协程执行完毕。

这个例子可以使用更多的惯用方法来重写，使用 `runBlocking` 来包装 main 函数：

```
import kotlinx.coroutines.*

fun main() = runBlocking<Unit> { // 开始执行主协程
    GlobalScope.launch { // 在后台开启一个新的协程并继续
        delay(1000L)
        println("World!")
    }
    println("Hello,") // 主协程在这里会立即执行
    delay(2000L)      // 延迟2秒来保证 JVM 存活
}
```

> 你可以点击这里获得完整代码

这里的 `runBlocking<Unit> { …… }` 作为一个适配器被用来启动最高优先级的主协程。我们显式的声明 `Unit` 为返回值类型，因为Kotlin中的 `main` 函数返回 `Unit` 类型。

这也是一种使用挂起函数来实现单元测试的方法：

```
class MyTest {
    @Test
    fun testMySuspendingFunction() = runBlocking<Unit> {
        // 这里我们可以使用挂起函数来实现任何我们喜欢的断言风格
    }
}
```

等待一个任务

延迟一段时间来等待另一个协程开始工作并不是一个好的选择。让我们显式地等待（使用非阻塞的方法）一个后台 Job 执行结束：

```
val job = GlobalScope.launch { // 启动一个新的协程并保持对这个任务的引用
    delay(1000L)
    println("World!")
}
println("Hello,")
job.join() // 等待直到子协程执行结束
```

> 你可以点击这里来获得完整代码

现在，结果仍然相同，但是主协程与后台任务的持续时间没有任何关系。这样写会更好。

结构化的并发

这里还有一些东西我们期望的写法被使用在协程的实践中。当我们使用 `GlobalScope.launch` 时我们创建了一个最高优先级的协程。甚至，虽然它是轻量级的，但是它在运行起来的时候仍然消耗了一些内存资源。甚至如果我们失去了一个对新创建的协程的引用，它仍然会继续运行。如果一段代码在协程中挂起（举例来说，我们错误的延迟了太长时间），如果我们启动了太多的协程，是否会导致内存溢出？如果我们手动引用所有的协程和 join 是非常容易出错的。

这有一个更好的解决办法。我们可以在你的代码中使用结构化并发。用来代替在 GlobalScope 中启动协程，就像我们使用线程时那样（线程总是全局的），我们可以在一个具体的作用域中启动协程并操作。

在我们的例子中，我们有一个被转换成使用 runBlocking 的协程构建器的 `main` 函数，每一个协程构建器，包括 `runBlocking`，在它代码块的作用域内添加一个CoroutineScope 实例。在这个作用域内启动的协程不需要明确的调用 `join`，因为一个外围的协程（我们的例子中的 `runBlocking`）只有在它作用域内所有协程执行完毕后才会结束。从而，我们可以使我们的示例更简单：

```
import kotlinx.coroutines.*

fun main() = runBlocking { // this: CoroutineScope
    launch { // 在 runBlocking 作用域中启动一个新协程
        delay(1000L)
        println("World!")
    }
    println("Hello,")
}
```

你可以点击这里来获取完整代码

作用域构建器

除了由上面多种构建器提供的协程作用域，也是可以使用 coroutineScope 构建器来声明你自己的作用域的。它启动了一个新的协程作用域并且在所有子协程执行结束后并没有执行完毕。runBlocking 和 coroutineScope 主要的不同之处在于后者在等待所有的子协程执行完毕时并没有使当前线程阻塞。

```
import kotlinx.coroutines.*

fun main() = runBlocking { // this: CoroutineScope
    launch {
        delay(200L)
        println("Task from runBlocking")
    }

    coroutineScope { // 创建一个新的协程作用域
        launch {
            delay(500L)
            println("Task from nested launch")
        }

        delay(100L)
        println("Task from coroutine scope") // 该行将在嵌套启动之前执行打印
    }

    println("Coroutine scope is over") // 该行将在嵌套结束之后才会被打印
}
```

你可以点击这里来获得完整代码

提取函数重构

让我们在 `launch { …… }` 中提取代码块并分离到另一个函数中。当你在这段代码上展示“提取函数”函数的时候，你得到了一个新的函数并用 `suspend` 修饰。这是你的第一个 *挂起函数* 。挂起函数可以像一个普通的函数一样使用内部协程，但是它们拥有一些额外的特性，反过来说，使用其它的挂起函数，比如这个例子中的 `delay`，可以使协程暂停执行。

```
import kotlinx.coroutines.*

fun main() = runBlocking {
    launch { doWorld() }
    println("Hello,")
}

// 你的第一个挂起函数
suspend fun doWorld() {
    delay(1000L)
    println("World!")
}
```

> 你可以点击这里来获得完整代码

但是如果提取函数包含了一个调用当前作用域的协程构建器？在这个例子中仅仅使用 `suspend` 来修饰提取出来的函数是不够的。在 `CoroutineScope` 调用 `doWorld` 方法是一种解决方案，但它并非总是适用，因为它不会使API看起来更清晰。惯用的解决方法是使 `CoroutineScope` 在一个类中作为一个属性并包含一个目标函数，或者使它外部的类实现 `CoroutineScope` 接口。作为最后的手段，CoroutineScope(coroutineContext) 也是可以使用的，但是这样的结构是不安全的，因为你将无法在这个作用域内控制方法的执行。只有私有的API可以使用这样的写法。

协程是轻量级的

运行下面的代码：

```
import kotlinx.coroutines.*

fun main() = runBlocking {
    repeat(100_000) { // 启动大量的协程
        launch {
            delay(1000L)
            print(".")
        }
    }
}
```

> 你可以点击这里来获得完整代码

它启动了100,000个协程，并且每秒钟每个协程打印一个点。现在，尝试使用线程来这么做。将会发生什么？（大多数情况下你的代码将会抛出内存溢出错误）

像守护线程一样的全局协程

下面的代码在 GlobalScope 中启动了一个长时间运行的协程，它在1秒内打印了“I'm sleeping”两次并且延迟一段时间后在main函数中返回：

```
GlobalScope.launch {
    repeat(1000) { i ->
        println("I'm sleeping $i ...")
        delay(500L)
    }
}
delay(1300L) // 在延迟之后结束程序
```

> 你可以点击这里来获取完整代码

你可以运行这个程序并在命令行中看到它打印出了如下三行：

```
I'm sleeping 0 ...
I'm sleeping 1 ...
I'm sleeping 2 ...
```

在 GlobalScope 中启动的活动中的协程就像守护线程一样，不能使它们所在的进程保活。

目录

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：

```
val job = launch {
    repeat(1000) { i ->
        println("I'm sleeping $i ...")
        delay(500L)
    }
}
delay(1300L) // 延迟一段时间
println("main: I'm tired of waiting!")
job.cancel() // 取消该任务
job.join() // 等待任务执行结束
println("main: Now I can quit.")
```

> 你可以点击这里获得完整代码

程序执行后的输出如下：

```
I'm sleeping 0 ...
I'm sleeping 1 ...
I'm sleeping 2 ...
main: I'm tired of waiting!
main: Now I can quit.
```

一旦 main 函数调用了 `job.cancel`，我们在其它的协程中就看不到任何输出，因为它被取消了。这里也有一个可以使 Job 挂起的函数 cancelAndJoin 它合并了对 cancel 以及 join 的调用。

取消是协作的

协程的取消是 *协作* 的。一段协程代码必须协作才能被取消。所有 `kotlinx.coroutines` 中的挂起函数都是 *可被取消的* 。它们检查协程的取消，并在取消时抛出 CancellationException。然而，如果协程正在执行计算任务，并且没有检查取消的话，那么它是不能被取消的，就如如下示例代码所示：

```
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("I'm sleeping ${i++} ...")
            nextPrintTime += 500L
        }
    }
}
delay(1300L) // 等待一段时间
println("main: I'm tired of waiting!")
job.cancelAndJoin() // 取消一个任务并且等待它结束
println("main: Now I can quit.")
```

你可以点击这里获得完整代码

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，任务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

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。

```
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("I'm sleeping ${i++} ...")
            nextPrintTime += 500L
        }
    }
}
delay(1300L) // 等待一段时间
println("main: I'm tired of waiting!")
job.cancelAndJoin() // 取消该任务并等待它结束
println("main: Now I can quit.")
```

你可以点击这里获得完整代码

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：

```
val job = launch {
    try {
        repeat(1000) { i ->
            println("I'm sleeping $i ...")
            delay(500L)
        }
    } finally {
        println("I'm running finally")
    }
}
delay(1300L) // 延迟一段时间
println("main: I'm tired of waiting!")
job.cancelAndJoin() // 取消该任务并且等待它结束
println("main: Now I can quit.")
```

> 你可以点击这里获得完整代码

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：

```
I'm sleeping 0 ...
I'm sleeping 1 ...
I'm sleeping 2 ...
main: I'm tired of waiting!
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、取消一个任务、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：

```
val job = launch {
    try {
        repeat(1000) { i ->
            println("I'm sleeping $i ...")
            delay(500L)
        }
    } finally {
        withContext(NonCancellable) {
            println("I'm running finally")
            delay(1000L)
            println("And I've just delayed for 1 sec because I'm non-cancellable")
        }
    }
}
delay(1300L) // 延迟一段时间
println("main: I'm tired of waiting!")
job.cancelAndJoin() // 取消该任务并等待它结束
println("main: Now I can quit.")
```

你可以点击这里获得完整代码

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：

```
withTimeout(1300L) {
    repeat(1000) { i ->
        println("I'm sleeping $i ...")
        delay(500L)
    }
}
```

你可以点击这里获得完整代码

运行后得到如下输出：

```
I'm sleeping 0 ...
I'm sleeping 1 ...
I'm sleeping 2 ...
Exception in thread "main" kotlinx.coroutines.TimeoutCancellationException: Timed out
waiting for 1300 ms
```

withTimeout 抛出了 `TimeoutCancellationException`，它是 CancellationException 的子类。我们之前没有在控制台上看到堆栈跟踪信息的打印。这是因为在被取消的协程中 `CancellationException` 被认为是协程执行结束的正常原因。然而，在这个示例中我们在 `main` 函数中正确地使用了 `withTimeout`。

由于取消只是一个例外，所有的资源都使用常用的方法来关闭。如果你需要做一些各类使用超时的特别的额外操作，可以使用类似 withTimeout 的 withTimeoutOrNull 函数，并把这些会超时的代码包装在 `try {...} catch (e: TimeoutCancellationException) {...}` 代码块中，而 withTimeoutOrNull 通过返回 `null` 来进行超时操作，从而替代抛出一个异常：

```
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")
```

> 你可以点击这里获得完整代码

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
```

目录

通道(实验性的)

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。

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。

```
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!")
```

> 你可以点击这里获得完整代码

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。

```
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!")
```

> 你可以点击这里获得完整代码

构建通道生产者

协程生成一系列元素的模式很常见。这是 *生产者——消费者* 模式的一部分，并且经常能在并发的代码中看到它。你可以将生产者抽象成一个函数，并且使通道作为它的参数，但这与必须从函数中返回结果的常识相违悖。

这里有一个名为 produce 的便捷的协程构建器，可以很容易的在生产者端正确工作，并且我们使用扩展函数 consumeEach 在消费者端替代 `for` 循环：

```
val squares = produceSquares()
squares.consumeEach { println(it) }
println("Done!")
```

> 你可以点击这里获得完整代码

管道

管道是一种一个协程在流中开始生产可能无穷多个元素的模式：

```
fun CoroutineScope.produceNumbers() = produce<Int> {
    var x = 1
    while (true) send(x++) // 在流中开始从1生产无穷多个整数
}
```

并且另一个或多个协程开始消费这些流，做一些操作，并生产了一些额外的结果。在下面的例子中，对这些数字仅仅做了平方操作：

```
fun CoroutineScope.square(numbers: ReceiveChannel<Int>): ReceiveChannel<Int> = produce {
    for (x in numbers) send(x * x)
}
```

主要的代码启动并连接了整个管道：

```
val numbers = produceNumbers() // 从1开始生产整数
val squares = square(numbers) // 对整数做平方
for (i in 1..5) println(squares.receive()) // 打印前5个数字
println("Done!") // 我们的操作已经结束了
coroutineContext.cancelChildren() // 取消子协程
```

> 你可以点击这里获得完整代码

> 所有创建了协程的函数被定义在了 CoroutineScope 的扩展上，所以我们可以依靠结构化并发来确保没有常驻在我们的应用程序中的全局协程。

使用管道的素数

让我们来展示一个极端的例子——在协程中使用一个管道来生成素数。我们开启了一个数字的无限序列。

```
fun CoroutineScope.numbersFrom(start: Int) = produce<Int> {
    var x = start
    while (true) send(x++) // 开启了一个无限的整数流
}
```

在下面的管道阶段中过滤了来源于流中的数字，删除了所有可以被给定素数整除的数字。

```
fun CoroutineScope.filter(numbers: ReceiveChannel<Int>, prime: Int) = produce<Int> {
    for (x in numbers) if (x % prime != 0) send(x)
}
```

现在我们开启了一个从2开始的数字流管道，从当前的通道中取一个素数，并为每一个我们发现的素数启动一个流水线阶段：

```
numbersFrom(2) -> filter(2) -> filter(3) -> filter(5) -> filter(7) ……
```

下面的例子打印了前十个素数，在主线程的上下文中运行整个管道。直到所有的协程在该主协程 runBlocking 的作用域中被启动完成。我们不必使用一个显式的列表来保存所有被我们已经启动的协程。我们使用 cancelChildren 扩展函数在我们打印了前十个素数以后来取消所有的子协程。

```
var cur = numbersFrom(2)
for (i in 1..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
```

> 你可以点击这里获得完整代码

这段代码的输出如下：

```
2
3
5
7
11
13
17
19
23
29
```

注意，你可以在标准库中使用 `buildIterator` 协程构建器来构建一个相似的管道。使用 `buildIterator` 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 `buildSequence` / `buildIterator`，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者协程。在这个示例中，它们只是打印它们的 id 和接收到的数字：

```
fun CoroutineScope.launchProcessor(id: Int, channel: ReceiveChannel<Int>) = launch {
    for (msg in channel) {
        println("Processor #$id received $msg")
    }
}
```

现在让我们启动五个处理者协程并让它们工作将近一秒。看看发生了什么：

```
val producer = produceNumbers()
repeat(5) { launchProcessor(it, producer) }
delay(950)
producer.cancel() // 取消协程处理器从而将它们全部杀死
```

> 你可以点击这里获得完整代码

该输出将类似于如下所示，尽管接收的是处理器的 id 但每个整数也许会不同：

```
Processor #2 received 1
Processor #4 received 2
Processor #0 received 3
Processor #1 received 4
Processor #3 received 5
Processor #2 received 6
Processor #4 received 7
Processor #0 received 8
Processor #1 received 9
Processor #3 received 10
```

注意，取消生产者协程并关闭它的通道，因此通过正在执行的处理者协程通道来终止迭代。

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。如果其中一个处理者协程执行失败，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）：

```
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
```

你可以点击这里获得完整代码

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（aka rendezvous（这句话应该是个俚语，意思好像是“又是约会”的意思，不知道怎么翻））。如果发送先被调用，则它将被挂起直到接收被调用，如果接收先被调用，它将被挂起直到发送被调用。

Channel() 工厂函数与 produce 建造器通过一个可选的参数 `capacity` 来指定 *缓冲区大小*。缓冲允许发送者在被挂起前发送多个元素，就像 `BlockingQueue` 有指定的容量一样，当缓冲区被占满的时候将会引起阻塞。

看看如下代码的表现：

```
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
```

你可以点击这里获得完整代码

使用缓冲通道并给 capacity 参数传入 *四* 它将打印 “sending” *五* 次：

```
Sending 0
Sending 1
Sending 2
Sending 3
Sending 4
```

前四个元素被加入到了缓冲区并且发送者在试图发送第五个元素的时候被挂起。

通道是公平的

发送和接收操作是 *公平的* 并且尊重调用它们的多个协程。它们遵守先进先出原则，可以看到第一个协程调用 `receive` 并得到了元素。在下面的例子中两个协程 “乒” 和 "乓" 都从共享的“桌子”通道接收到这个“球”元素。

```
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
    }
}
```

你可以点击这里得到完整代码

“乒”协程首先被启动，所以它首先接收到了球。甚至虽然“乒” 协程在将球发送会桌子以后立即开始接收，但是球还是被“乓” 协程接收了，因为它一直在等待着接收球：

```
ping Ball(hits=1)
pong Ball(hits=2)
ping Ball(hits=3)
pong Ball(hits=4)
```

注意，有时候通道执行时由于执行者的性质而看起来不那么公平。点击这个提案来查看更多细节。

计时器通道

计时器通道是一种特别的会合通道，每次经过特定的延迟都会从该通道进行消费并产生 `Unit`。虽然它看起来似乎没用，它被用来构建分段来创建复杂的基于时间的 produce 管道和进行窗口化操作以及其它时间相关的处理。可以在 select 中使用计时器通道来进行“打勾”操作。

使用工厂方法 ticker 来创建这些通道。为了表明不需要其它元素，请使用 ReceiveChannel.cancel 方法。

现在让我们看看它是如何在实践中工作的：

```
import kotlinx.coroutines.*
import kotlinx.coroutines.channels.*

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") // 初始尚未经过的延迟

    nextElement = withTimeoutOrNull(50) { tickerChannel.receive() } // 所有随后到来的元素都经过了100浩渺的延迟
    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
}
```

> 你可以点击这里获得完整代码

它的打印如下：

```
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。

目录

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：

```
val time = measureTimeMillis {
    val one = doSomethingUsefulOne()
    val two = doSomethingUsefulTwo()
    println("The answer is ${one + two}")
}
println("Completed in $time ms")
```

> 你可以点击这里获得完整代码

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。

```
val time = measureTimeMillis {
    val one = async { doSomethingUsefulOne() }
    val two = async { doSomethingUsefulTwo() }
    println("The answer is ${one.await() + two.await()}")
}
println("Completed in $time ms")
```

> 你可以点击这里获得完整代码

它的打印输出如下：

```
The answer is 42
Completed in 1017 ms
```

这里快了两倍，因为我们使用两个协程进行并发。注意，使用协程进行并发总是显式的。

惰性启动的 async

使用一个可选的参数 `start` 并传值 CoroutineStart.LAZY，可以对 async 进行惰性操作。只有当结果需要被 await 或者如果一个 start 函数被调用，协程才会被启动。运行下面的示例：

```
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")
```

> 你可以点击这里获得完整代码

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被执行，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，如果我们在 `println` 中调用了 await 并且在这个协程中省略调用了 start，接下来 await 会开始执行协程并且等待协程执行结束，因此我们会得到顺序的行为，但这不是惰性启动的预期用例。当调用挂起函数计算值的时候 `async(start = CoroutineStart.LAZY)` 用例是标准的 `lazy` 函数的替换方案。

async 风格的函数

我们可以定义异步风格的函数来 *异步* 的调用 `doSomethingUsefulOne` 和 `doSomethingUsefulTwo` 并使用 async 协程建造器并带有一个显式的 GlobalScope 引用。我们给这样的函数的名称中加上“Async”后缀来突出表明：事实上，它们只做异步计算并且需要使用延期的值来获得结果。

```
// somethingUsefulOneAsync 函数的返回值类型是 Deferred<Int>
fun somethingUsefulOneAsync() = GlobalScope.async {
    doSomethingUsefulOne()
}

// somethingUsefulTwoAsync 函数的返回值类型是 Deferred<Int>
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被使用。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：

```
// 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")
}
```

> 你可以点击这里获得完整代码

> 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。

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被中止，将会发生什么。通常情况下，一个全局的异常处理者会捕获这个异常，将异常打印成日记并报告给开发者，但是反之该程序将会继续执行其它操作。但是这里我们的 `somethingUsefulOneAsync` 仍然在后台执行，尽管如此，启动它的那次操作也会被终止。这个程序将不会进行结构化并发，如下一小节所示。

使用 async 的结构化并发

让我们使用使用 async 的并发这一小节的例子并且提取出一个函数并发的调用 `doSomethingUsefulOne` 与 `doSomethingUsefulTwo` 并且返回它们两个的结果之和。由于 async 被定义为了 CoroutineScope 上的扩展，我们需要将它写在作用域内，并且这是 coroutineScope 函数所提供的：

```
suspend fun concurrentSum(): Int = coroutineScope {
    val one = async { doSomethingUsefulOne() }
    val two = async { doSomethingUsefulTwo() }
     one.await() + two.await()
}
```

这种情况下，如果在 `concurrentSum` 函数内部发生了错误，并且它抛出了一个异常，所有在作用域中启动的协程都将会被取消。

```
val time = measureTimeMillis {
    println("The answer is ${concurrentSum()}")
}
println("Completed in $time ms")
```

> 你可以点击这里获得完整代码

从上面的 main 函数的输出可以看出，我们仍然可以同时执行这两个操作：

```
The answer is 42
Completed in 1017 ms
```

取消始终通过协程的层次结构来进行传递：

```
import kotlinx.coroutines.*

fun main() = runBlocking<Unit> {
    try {
        failedConcurrentSum()
    } catch(e: ArithmeticException) {
        println("Computation failed with ArithmeticException")
    }
}

suspend fun failedConcurrentSum(): Int = coroutineScope {
    val one = async<Int> {
        try {
            delay(Long.MAX_VALUE) // 模拟一个长时间的运算
            42
        } finally {
            println("First child was cancelled")
        }
    }
    val two = async<Int> {
        println("Second child throws an exception")
        throw ArithmeticException()
    }
        one.await() + two.await()
}
```

你可以点击这里这里获得完整代码

注意，当第一个子协程失败的时候第一个 `async` 是如何等待父线程被取消的：

```
Second child throws an exception
First child was cancelled
Computation failed with ArithmeticException
```

目录

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

协程上下文包括了一个 *协程调度器* (请参见 CoroutineDispatcher)，它确定了相应的协程在执行时使用一个或多个线程。协程调度器可以将协程的执行局限在指定的线程中，调度它运行在线程池中或让它不受限的运行。

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：

```
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}")
}
```

你可以点击这里获得完整代码

它执行后得到了如下输出(也许顺序会有所不同):

```
Unconfined            : I'm working in thread main
Default               : I'm working in thread DefaultDispatcher-worker-1
newSingleThreadContext: I'm working in thread MyOwnThread
main runBlocking      : I'm working in thread main
```

当调用 `launch { …… }` 时不传参数,它从启动了它的 CoroutineScope 中承袭了上下文(以及调度器)。在这个案例中,它从 `main` 线程中的 `runBlocking` 主协程承袭了上下文。

Dispatchers.Unconfined 是一个特殊的调度器且似乎也运行在 `main` 线程中,但实际上,它是一种不同的机制,这会在后文中讲到。

该默认调度器,当协程在 GlobalScope 中启动的时候被使用,它代表 Dispatchers.Default 使用了共享的后台线程池,所以 `GlobalScope.launch { …… }` 也可以使用相同的调度器——`launch(Dispatchers.Default) { …… }`。

newSingleThreadContext 为协程的运行启动了一个新的线程。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,或存储在一个顶级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 协程调度器在被调用的线程中启动协程,但是这只有直到程序运行到第一个挂起点的时候才行。挂起后,它将在完全由该所运行的线程中恢复挂起被调用的函数。非受限的调度器是合适的,当协程没有消耗 CPU 时间或更新共享数据(比如UI界面)时它被限制在了指定的线程中。

另一方面,默认的,一个调度器承袭自外部的 CoroutineScope。而 runBlocking 协程的默认调度器,特别是,被限制在调用它的线程,因此承袭它在限制有可预测的 FIFO 调度的线程的执行上是非常有效果的。

```
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}")
}
```

> 你可以点击这里获得完整代码

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
```

因此，该协程从 `runBlocking {……}` 协程中承袭了上下文并在主线程中执行，同时使用非受限调度器的协程从被执行 delay 函数的默认执行者线程中恢复。

> 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被用在通常的代码中。

调试协程与线程

协程可以在一个线程上挂起并恢复其它线程。甚至一个单线程的调度器是非常难以弄清楚协程何时，在哪里，正在做什么的。使用通常的方法来调试应用程序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。使用协程时，单独的线程名称不会给出很多上下文，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：

```
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()}")
```

> 你可以点击这里获得完整代码

这里有三个协程，其中主协程是 (#1) —— `runBlocking`，而另外两个协程计算延期的值 `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
```

该 `log` 函数将线程的名字打印在了方括号中，你可以看到是 `main` 线程，但是当前正在执行的协程的标识符被附加到它上面。当调试模式开启的时候该标识符被连续赋值给所有已经被创建的协程。

你可以在 newCoroutineContext 函数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）：

```
newSingleThreadContext("Ctx1").use { ctx1 ->
    newSingleThreadContext("Ctx2").use { ctx2 ->
        runBlocking(ctx1) {
            log("Started in ctx1")
            withContext(ctx2) {
                log("Working in ctx2")
            }
            log("Back to ctx1")
        }
    }
}
```

> 你可以点击这里获得完整代码

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，你可以在下面的输出中看到：

```
[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。

上下文中的任务

协程的 Job 是它上下文中的一部分。协程可以在它所属的上下文中使用 `coroutineContext[Job]` 表达式来取回它：

```
println("My job is ${coroutineContext[Job]}")
```

> 你可以点击这里获得完整代码

当它运行于调试模式时将处理一些任务：

```
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 将会成为父协程任务的 *子* 任务。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。

然而，当 GlobalScope 被用来启动一个协程时，它与作用域无关且是独立被启动的。

```
// 启动一个协程来处理某种传入请求(request)
val request = launch {
    // 孵化了两个子任务, 其中一个通过 GlobalScope 启动
    GlobalScope.launch {
        println("job1: I run in GlobalScope 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 cancelled")
    }
}
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?")
```

> 你可以点击这里获得完整代码

这段代码的输出如下：

```
job1: I run in GlobalScope 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以及不必使用 Job.join 在最后的时候等待它们：

```
// 启动一个协程来处理某种传入请求(request)
val request = launch {
    repeat(3) { i -> // 启动少量的子任务
        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")
```

你可以点击这里获得完整代码

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

协程日志会频繁记录的时候以及当你只是需要来自相同协程的关联日志记录，自动分配 id 是非常棒的。然而，当协程与执行一个明确的请求或与执行一些显式的后台任务有关的时候，出于调试的目的给它明确的命名是更好的做法。CoroutineName 上下文元素可以给线程像给函数命名一样命名。它在协程被执行且调试模式被开启时将显示线程的名字。

下面的例子演示了这一概念：

```
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()}")
```

你可以点击这里获得完整代码

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：

```
launch(Dispatchers.Default + CoroutineName("test")) {
    println("I'm working in thread ${Thread.currentThread().name}")
}
```

你可以点击这里获得完整代码

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
```

通过显式任务取消

让我们把有关上下文、子协程以及任务的知识梳理一下。假设我们的应用程序中有一个在生命周期中的对象，但这个对象并不是协程。假如，我们写了一个 Android 应用程序并在上下文中启动了多个协程来为 Android activity 进行异步操作来拉取以及更新数据，或作动画等。当 activity 被销毁的时候这些协程必须被取消以防止内存泄漏。

我们通过创建一个 Job 的实例来管理协程的生命周期，并让它与我们的 activity 的生命周期相关联。当一个 activity 被创建的时候一个任务 (job) 实例被使用 Job() 工厂函数创建，并且当这个 activity 被销毁的时候它也被取消，就像下面这样：

```
class Activity : CoroutineScope {
    lateinit var job: Job

    fun create() {
        job = Job()
    }

    fun destroy() {
        job.cancel()
    }
    // 继续运行……
```

我们也可以在 `Actvity` 类中实现 CoroutineScope 接口。我们只需提供一个覆盖的 CoroutineScope.coroutineContext 属性来在作用域中为协程指定上下文。我们结合所需要的调度器（我们在这个例子中使用 Dispatchers.Default）和任务：

```
    // 在 Activity 类中
    override val coroutineContext: CoroutineContext
        get() = Dispatchers.Default + job
    // 继续运行……
```

现在，在这个 `Activity` 的作用域中启动协程，且没有明确指定它们的上下文。在示例中，我们启动了十个协程并延迟不同的时间：

```
    // 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 ->
            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
```

在我们的 main 函数中我们创建了 activity，调用我们的 `doSomething` 测试函数，并在500毫秒后销毁 activity，所有已经启动了的协程都被取消了，如果我们等待的话可以确认没有任何东西被打印在屏幕上：

```
val activity = Activity()
activity.create() // 启动一个 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
```

> 你可以点击这里获得完整代码

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()` 中被单次调用了 `job.cancel()`。

线程局部数据

有时能够传递一些线程局部的数据很方便，但是，对于协程来说，它们不受任何特定线程的约束，所以很难手动的去实现它并且不写出大量的样板代码。

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：

```
threadLocal.set("main")
println("Pre-main, current thread: ${Thread.currentThread()}, thread local value:
'${threadLocal.get()}'")
val job = launch(Dispatchers.Default + threadLocal.asContextElement(value = "launch")) {
   println("Launch start, current thread: ${Thread.currentThread()}, thread local value:
'${threadLocal.get()}'")
    yield()
    println("After yield, current thread: ${Thread.currentThread()}, thread local value:
'${threadLocal.get()}'")
}
job.join()
println("Post-main, current thread: ${Thread.currentThread()}, thread local value:
'${threadLocal.get()}'")
```

> 你可以点击这里获得完整代码

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")`，无论协程执行在什么线程中都是没有问题的。因此，输出如（调试）所示：

```
Pre-main, current thread: Thread[main @coroutine#1,5,main], thread local value: 'main'
Launch start, current thread: Thread[DefaultDispatcher-worker-1 @coroutine#2,5,main],
thread local value: 'launch'
After yield, current thread: Thread[DefaultDispatcher-worker-2 @coroutine#2,5,main],
thread local value: 'launch'
Post-main, current thread: Thread[main @coroutine#1,5,main], thread local value: 'main'
```

`ThreadLocal` 具有一流的支持，可以与任何原始的 `kotlinx.coroutines` 一起使用。它有一个关键限制：当线程局部发生突变，新值不会传递到协程调用者中（作为上下文元素不能跟踪所有的 `ThreadLocal` 对象访问）并且下次挂起时更新的值将丢失。在协程中使用 withContext 线程局部的值，可以查看 asContextElement 的更多细节。

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，请参阅应实现的 ThreadContextElement 接口的文档。

目录

异常处理

这部分内容包括异常处理以及取消异常。我们已经知道当协程被取消的时候会在挂起点抛出 CancellationException，并且它在协程机制中被忽略了。但是如果一个异常在取消期间被抛出或多个子协程在同一个父协程中抛出异常将会发生什么？

异常的传播

协程构建器有两种风格：自动的传播异常（launch 以及 actor）或者将它们暴露给用户（async 以及 produce）。前者对待异常是不处理的，类似于 Java 的 `Thread.uncaughtExceptionHandler`，而后者依赖用户来最终消耗异常，比如说，通过 await 或 receive（produce 以及 receive 在通道中介绍过）。

可以通过一个在 GlobalScope 中创建新协程的简单示例来进行演示：

```
import kotlinx.coroutines.*

fun main() = runBlocking {
    val job = GlobalScope.launch {
        println("Throwing exception from launch")
        throw IndexOutOfBoundsException() // 我们将在控制台打印
Thread.defaultUncaughtExceptionHandler
    }
    job.join()
    println("Joined failed job")
    val deferred = GlobalScope.async {
        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")
    }
}
```

你可以点击这里获得完整代码

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
Joined failed job
Throwing exception from async
Caught ArithmeticException
```

CoroutineExceptionHandler

但是如果不想将所有的异常打印在控制台中呢? CoroutineExceptionHandler 上下文元素被用来将通用的 `catch` 代码块用于在协程中自定义日志记录或异常处理。它和使用 `Thread.uncaughtExceptionHandler` 很相似。

在 JVM 中可以重定义一个全局的异常处理者来将所有的协程通过 `ServiceLoader` 注册到 CoroutineExceptionHandler。全局异常处理者就如同 `Thread.defaultUncaughtExceptionHandler` 一样,在没有更多的指定的异常处理者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者中。

CoroutineExceptionHandler 仅在预计不会由用户处理的异常上调用, 所以在 async 构建器中注册它没有任何效果。

```
val handler = CoroutineExceptionHandler { _, exception ->
    println("Caught $exception")
}
val job = GlobalScope.launch(handler) {
    throw AssertionError()
}
val deferred = GlobalScope.async(handler) {
    throw ArithmeticException() // 没有打印任何东西,依赖用户去调用 deferred.await()
}
joinAll(job, deferred)
```

你可以点击这里获得完整代码

这段代码的输出如下：

```
Caught java.lang.AssertionError
```

取消与异常

取消与异常紧密相关。协程内部使用 `CancellationException` 来进行取消，这个异常会被所有的处理者忽略，所以那些可以被 `catch` 代码块捕获的异常仅仅应该被用来作为额外调试信息的资源。当一个协程在没有任何理由的情况下使用 Job.cancel 取消的时候，它会被终止，但是它不会取消它的父协程。无理由取消是父协程取消其子协程而非取消其自身的机制。

```
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()
```

你可以点击这里获得完整代码

这段代码的输出如下：

```
Cancelling child
Child is cancelled
Parent is not cancelled
```

如果协程遇到除 `CancellationException` 以外的异常，它将取消具有该异常的父协程。这种行为不能被覆盖，且它被用来提供一个稳定的协程层次结构来进行结构化并发而无需依赖 CoroutineExceptionHandler 的实现。且当所有的子协程被终止的时候，原本的异常被父协程所处理。

> 这也是为什么，在这个例子中，CoroutineExceptionHandler 总是被设置在由 GlobalScope 启动的协程中。将异常处理者设置在 runBlocking 主作用域内启动的协程中是没有意义的，尽管子协程已经设置了异常处理者，但是主协程也总是会被取消的。

```
val handler = CoroutineExceptionHandler { _, exception ->
    println("Caught $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 until all
children terminate")
                delay(100)
                println("The first child finished its non cancellable block")
            }
        }
    }
    launch { // 第二个子协程
        delay(10)
        println("Second child throws an exception")
        throw ArithmeticException()
    }
}
job.join()
```

> 你可以点击这里获得完整代码

这段代码的输出如下：

```
Second child throws an exception
Children are cancelled, but exception is not handled until all children terminate
The first child finished its non cancellable block
Caught java.lang.ArithmeticException
```

异常聚合

如果一个协程的多个子协程抛出异常将会发生什么？通常的规则是 “第一个异常赢得了胜利”，所以第一个被抛出的异常将会暴露给处理者。但也许这会是异常丢失的原因，比如说一个协程在 `finally` 块中抛出了一个异常。这时，多余的异常将会被压制。

> 其中一个解决方法是分别抛出异常，但是接下来 Deferred.await 应该有相同的机制来避免行为不一致并且会导致协程的实现细节（是否已将其部分工作委托给子协程）泄漏到异常处理者中。

```
import kotlinx.coroutines.*
import java.io.*

fun main() = runBlocking {
    val handler = CoroutineExceptionHandler { _, exception ->
        println("Caught $exception with suppressed
${exception.suppressed.contentToString()}")
    }
    val job = GlobalScope.launch(handler) {
        launch {
            try {
                delay(Long.MAX_VALUE)
            } finally {
                throw ArithmeticException()
            }
        }
        launch {
            delay(100)
            throw IOException()
        }
        delay(Long.MAX_VALUE)
    }
    job.join()
}
```

> 你可以点击这里获得完整代码

> 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。

这段代码的输出如下:

```
Caught java.io.IOException with suppressed [java.lang.ArithmeticException]
```

> 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,但将来会被修复。

取消异常是透明的并且会在默认情况下解包:

```
val handler = CoroutineExceptionHandler { _, exception ->
    println("Caught original $exception")
}
val job = GlobalScope.launch(handler) {
    val inner = launch {
        launch {
            launch {
                throw IOException()
            }
        }
    }
    try {
        inner.join()
    } catch (e: CancellationException) {
        println("Rethrowing CancellationException with original cause")
        throw e
    }
}
job.join()
```

> 你可以点击这里获得完整代码

这段代码的输出如下：

```
Rethrowing CancellationException with original cause
Caught original java.io.IOException
```

监督

正如我们之前研究的那样，取消是一种双向机制，在协程的整个层次结构之间传播。但是如果需要单向取消怎么办？

此类需求的一个良好示例是可以在其作用域内定义任务的的 UI 组件。如果任何一个 UI 的子任务执行失败了，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（并且它的任务也被取消了），由于它的结果不再被需要了，它有必要使所有的子任务执行失败。

另一个例子是服务进程孵化了一些子任务并且需要 *监督* 它们的执行，追踪它们的故障并在这些子任务执行失败的时候重启。

监督任务

SupervisorJob 可以被用于这些目的。它类似于常规的 Job，唯一的取消异常将只会向下传播。这是非常容易从示例中观察到的：

```
import kotlinx.coroutines.*

fun main() = runBlocking {
    val supervisor = SupervisorJob()
    with(CoroutineScope(coroutineContext + supervisor)) {
        // 启动第一个子任务--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("First child is failing")
            throw AssertionError("First child is cancelled")
        }
        // 启动第二个子任务
        val secondChild = launch {
            firstChild.join()
            // 取消了第一个子任务且没有传播给第二个子任务
            println("First child is cancelled: ${firstChild.isCancelled}, but second one
is still active")
            try {
                delay(Long.MAX_VALUE)
            } finally {
                // 但是取消了监督的传播
                println("Second child is cancelled because supervisor is cancelled")
            }
        }
        // 等待直到第一个子任务失败且执行完成
        firstChild.join()
        println("Cancelling supervisor")
        supervisor.cancel()
        secondChild.join()
    }
}
```

你可以点击这里获得完整代码

这段代码的输出如下：

```
First child is failing
First child is cancelled: true, but second one is still active
Cancelling supervisor
Second child is cancelled because supervisor is cancelled
```

监督作用域

对于*作用域*的并发，supervisorScope 可以被用来替代 coroutineScope 来实现相同的目的。它只会单向的传播并且当子任务自身执行失败的时候将它们全部取消。它也会在所有的子任务执行结束前等待，就像 coroutineScope 所做的那样。

```
import kotlin.coroutines.*
import kotlinx.coroutines.*

fun main() = runBlocking {
    try {
        supervisorScope {
            val child = launch {
                try {
                    println("Child is sleeping")
                    delay(Long.MAX_VALUE)
                } finally {
                    println("Child is cancelled")
                }
            }
            // 使用 yield 来给我们的子任务一个机会来执行打印
            yield()
            println("Throwing exception from scope")
            throw AssertionError()
        }
    } catch(e: AssertionError) {
        println("Caught assertion error")
    }
}
```

你可以点击这里获得完整代码

这段代码的输出如下：

```
Child is sleeping
Throwing exception from scope
Child is cancelled
Caught assertion error
```

监督协程中的异常

常规的任务和监督任务之间的另一个重要区别是异常处理。每一个子任务应该通过异常处理机制处理自身的异常。这种差异来自于子任务的执行失败不会传播给它的父任务的事实。

```
import kotlin.coroutines.*
import kotlinx.coroutines.*

fun main() = runBlocking {
    val handler = CoroutineExceptionHandler { _, exception ->
        println("Caught $exception")
    }
    supervisorScope {
        val child = launch(handler) {
            println("Child throws an exception")
            throw AssertionError()
        }
        println("Scope is completing")
    }
    println("Scope is completed")
}
```

> 你可以点击这里获得完整代码

这段代码的输出如下：

```
Scope is completing
Child throws an exception
Caught java.lang.AssertionError
Scope is completed
```

目录

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 *其中一个* 的数据。但是 select 表达式允许我们使用其 onReceive 子句 *同时* 从两者接收：

```
suspend fun selectFizzBuzz(fizz: ReceiveChannel<String>, buzz: ReceiveChannel<String>) {
    select<Unit> { // <Unit> 意味着该 select 表达式不返回任何结果
        fizz.onReceive { value ->  // 这是第一个 select 子句
            println("fizz -> '$value'")
        }
        buzz.onReceive { value ->  // 这是第二个 select 子句
            println("buzz -> '$value'")
        }
    }
}
```

让我们运行代码7次：

```
val fizz = fizz()
val buzz = buzz()
repeat(7) {
    selectFizzBuzz(fizz, buzz)
}
coroutineContext.cancelChildren() // 取消 fizz 和 buzz 协程
```

> 你可以点击这里获得完整代码

这段代码的执行结果如下：

```
fizz -> 'Fizz'
buzz -> 'Buzz!'
fizz -> 'Fizz'
fizz -> 'Fizz'
buzz -> 'Buzz!'
fizz -> 'Fizz'
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 onReceiveOrNull 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.onReceiveOrNull { value ->
            if (value == null)
                "Channel 'a' is closed"
            else
                "a -> '$value'"
        }
        b.onReceiveOrNull { value ->
            if (value == null)
                "Channel 'b' is closed"
            else
                "b -> '$value'"
        }
    }
```

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：

```
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()
```

你可以点击这里获得完整代码

这段代码的结果非常有趣，所以我们将在细节中分析它：

```
a -> 'Hello 0'
a -> 'Hello 1'
b -> 'World 0'
a -> 'Hello 2'
a -> 'Hello 3'
b -> 'World 1'
Channel 'a' is closed
Channel 'a' is closed
```

有几个结果可以通过观察得出。

首先，`select` *偏向于* 第一个子句，当可以同时选到多个子句时，第一个子句将被选中。在这里，两个通道都在不断地生成字符串，因此 `a` 通道作为 select 中的第一个子句获胜。然而因为我们使用的是无缓冲通道，所以 `a` 在其调用 send 时会不时地被挂起，进而 `b` 也有机会发送。

第二个观察结果是，当通道已经关闭时，会立即选择 onReceiveOrNull。

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：

```
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()
```

你可以点击这里获得完整代码

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。

```
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")
```

你可以点击这里获得完整代码

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 onReceiveOrNull 和 onAwait 子句放在同一个 `select` 中：

```
fun CoroutineScope.switchMapDeferreds(input: ReceiveChannel<Deferred<String>>) =
produce<String> {
    var current = input.receive() // 从第一个接收到的延迟值开始
    while (isActive) { // 循环直到被取消或关闭
        val next = select<Deferred<String>?> { // 从这个 select 中返回下一个延迟值或 null
            input.onReceiveOrNull { update ->
                update // 替换下一个要等待的值
            }
            current.onAwait { value ->
                send(value) // 发送当前延迟生成的值
                input.receiveOrNull() // 然后使用从输入通道得到的下一个延迟值
            }
        }
        if (next == null) {
            println("Channel was closed")
            break // 跳出循环
        } else {
            current = next
        }
    }
}
```

为了测试它，我们将用一个简单的异步函数，它在特定的延迟后返回特定的字符串：

```
fun CoroutineScope.asyncString(str: String, time: Long) = async {
    delay(time)
    str
}
```

main 函数只是启动一个协程来打印 `switchMapDeferreds` 的结果并向它发送一些测试数据：

```
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
```

> 你可以点击这里获得完整代码

这段代码的执行结果：

```
BEGIN
Replace
END
Channel was closed
```

目录

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)并发执行。这样就可以提出所有常见的并发问题。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 CoroutineScope.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 {
        val jobs = List(n) {
            launch {
                repeat(k) { action() }
            }
        }
        jobs.forEach { it.join() }
    }
    println("Completed ${n * k} actions in $time ms")
}
```

我们从一个非常简单的动作开始：在 GlobalScope 中使用多线程的 Dispatchers.Default 来递增一个共享的可变变量。

```
GlobalScope.massiveRun {
    counter++
}
println("Counter = $counter")
```

> 你可以点击这里获得完整代码

这段代码最后打印出什么结果?它不太可能打印出“Counter = 100000”,因为一千个协程在一百个线程中同时递增计数器但没有做并发处理。

> 注意:如果你运行程序的机器使用两个或者更少的 CPU,那么你 *将* 总是会看到 100000,因为线程池在这种情况下只有一个线程可运行。要重现这个问题,可以做如下的变动:

```
CoroutineScope(mtContext).massiveRun { // 在此及以下示例中使用刚才定义的上下文,而不是默认的
Dispatchers.Default
    counter++
}
println("Counter = $counter")
```

> 你可以点击这里获得完整代码

volatile 无济于事

有一种常见的误解:volatile 可以解决并发问题。让我们尝试一下:

```
import kotlinx.coroutines.*
import kotlin.system.*

suspend fun CoroutineScope.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 {
        val jobs = List(n) {
            launch {
                repeat(k) { action() }
            }
        }
        jobs.forEach { it.join() }
    }
    println("Completed ${n * k} actions in $time ms")
}

@Volatile // 在 Kotlin 中 `volatile` 是一个注解
var counter = 0

fun main() = runBlocking<Unit> {
    GlobalScope.massiveRun {
        counter++
    }
    println("Counter = $counter")
}
```

> 你可以点击这里获得完整代码

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 `AtomicInteger` 类：

```
GlobalScope.massiveRun {
    counter.incrementAndGet()
}
println("Counter = ${counter.get()}")
```

你可以点击这里获得完整代码

这是针对此类特定问题的最快解决方案。它适用于普通计数器、集合、队列和其他标准数据结构以及它们的基本操作。然而，它并不容易被扩展来应对复杂状态、或一些没有现成的线程安全实现的复杂操作。

以细粒度限制线程

限制线程 是解决共享可变状态问题的一种方案：对特定共享状态的所有访问权都限制在单个线程中。它通常应用于 UI 程序中：所有 UI 状态都局限于单个事件分发线程或应用主线程中。这在协程中很容易实现，通过使用一个单线程上下文：

```
GlobalScope.massiveRun { // 使用 DefaultDispathcer 运行每个协程
    withContext(counterContext) { // 但是把每个递增操作都限制在此单线程上下文中
        counter++
    }
}
println("Counter = $counter")
```

你可以点击这里获得完整代码

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 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。这里我们使用 CoroutineScope() 函数来切换协程上下文为 CoroutineScope：

```
CoroutineScope(counterContext).massiveRun { // 在单线程上下文中运行每个协程
    counter++
}
println("Counter = $counter")
```

你可以点击这里获得完整代码

这段代码运行更快而且打印出了正确的结果。

互斥

该问题的互斥解决方案：使用永远不会同时执行的 *关键代码块* 来保护共享状态的所有修改。在阻塞的世界中，你通常会为此目的使用 `synchronized` 或者 `ReentrantLock`。在协程中的替代品叫做 Mutex。它具有 lock 和 unlock 方法，可以隔离关键的部分。关键的区别在于 `Mutex.lock()` 是一个挂起函数，它不会阻塞线程。

还有 withLock 扩展函数，可以方便的替代常用的 `mutex.lock(); try { …… } finally { mutex.unlock() }` 模式：

```
GlobalScope.massiveRun {
    mutex.withLock {
        counter++
    }
}
println("Counter = $counter")
```

> 你可以点击这里获得完整代码

此示例中锁是细粒度的，因此会付出一些代价。但是对于某些必须定期修改共享状态的场景，它是一个不错的选择，但是没有自然线程可以限制此状态。

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，因该特征它被用于此处。

```
// 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() // 携带回复的请求
```

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：

```
val counter = counterActor() // 创建该 actor
GlobalScope.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
```

> 你可以点击这里获得完整代码

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。

工具

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 和 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：

```
/**
 * 一组*成员*。
 *
 * 这个类没有有用的逻辑; 它只是一个文档示例。
 *
 * @param T 这个组中的成员的类型。
 * @property name 这个组的名称。
 * @constructor 创建一个空组。
 */
class Group<T>(val name: String) {
    /**
     * 将 [member] 添加到这个组。
     * @return 这个组的新大小。
     */
    fun add(member: T): Int { …… }
}
```

块标签

KDoc 目前支持以下块标签（block tags）：

`@param <名称>`

用于函数的值参数或者类、属性或函数的类型参数。为了更好地将参数名称与描述分开，如果你愿意，可以将参数的名称括在方括号中。因此，以下两种语法是等效的：

```
@param name 描述。
@param[name] 描述。
```

`@return`

用于函数的返回值。

`@constructor`

用于类的主构造函数。

`@receiver`

用于扩展函数的接收者。

`@property <名称>`

用于类中具有指定名称的属性。这个标签可用于在主构造函数中声明的属性，当然直接在属性定义的前面放置 doc 注释会很别扭。

`@throws <类>,@exception <类>`

用于方法可能抛出的异常。因为 Kotlin 没有受检异常，所以也没有期望所有可能的异常都写文档，但是当它会为类的用户提供有用的信息时，仍然可以使用这个标签。

`@sample <标识符>`

将具有指定限定的名称的函数的主体嵌入到当前元素的文档中，以显示如何使用该元素的示例。

`@see <标识符>`

将到指定类或方法的链接添加到文档的**另请参见**块。

`@author`

指定要编写文档的元素的作者。

`@since`

指定要编写文档的元素引入时的软件版本。

`@suppress`

从生成的文档中排除元素。可用于不是模块的官方 API 的一部分但还是必须在对外可见的元素。

> KDoc 不支持 `@deprecated` 这个标签。作为替代，请使用 `@Deprecated` 注解。

内联标记

对于内联标记，KDoc 使用常规 Markdown 语法，扩展了支持用于链接到代码中其他元素的简写语法。

链接到元素

要链接到另一个元素（类、方法、属性或参数），只需将其名称放在方括号中：

```
为此目的，请使用方法 [foo]。
```

如果要为链接指定自定义标签（label），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：

```
# 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 注解处理

> 译注：kapt 即 Kotlin annotation processing tool（Kotlin 注解处理工具）缩写。

在 Kotlin 中通过 *kapt* 编译器插件支持注解处理器（参见JSR 269）。

简而言之，你可以在 Kotlin 项目中使用像 Dagger 或者 Data Binding 这样的库。

关于如何将 *kapt* 插件应用于 Gradle/Maven 构建中，请阅读下文。

在 Gradle 中使用

应用 `kotlin-kapt` Gradle 插件：

```
apply plugin: 'kotlin-kapt'
```

或者，你可以使用插件 DSL 应用它：

```
plugins {
    id "org.jetbrains.kotlin.kapt" version "1.3.11"
}
```

然后在 `dependencies` 块中使用 `kapt` 配置添加相应的依赖项：

```
dependencies {
    kapt 'groupId:artifactId:版本'
}
```

如果你以前使用 Android 支持作为注解处理器，那么以 `kapt` 取代 `annotationProcessor` 配置的使用。如果你的项目包含 Java 类，`kapt` 也会顾全到它们。

如果为 `androidTest` 或 `test` 源代码使用注解处理器，那么相应的 `kapt` 配置名为 `kaptAndroidTest` 和 `kaptTest`。请注意 `kaptAndroidTest` 和 `kaptTest` 扩展了 `kapt`，所以你可以只提供 `kapt` 依赖而它对生产和测试源代码都可用。

注解处理器参数

使用 `arguments {}` 块将参数传给注解处理器：

```
kapt {
    arguments {
        arg("key", "value")
    }
}
```

Java 编译器选项

Kapt 使用 Java 编译器来运行注解处理器。以下是将任意选项传给 javac 的方式：

```
kapt {
    javacOptions {
        // 增加注解处理器的最大错误次数
        // 默认为 100。
        option("-Xmaxerrs", 500)
    }
}
```

非存在类型校正

一些注解处理器（如 `AutoFactory`）依赖于声明签名中的精确类型。默认情况下，Kapt 将每个未知类型（包括生成的类的类型）替换为 `NonExistentClass`，但你可以更改此行为。将额外标志添加到 `build.gradle` 文件以启用在存根（stub）中推断出的错误类型：

```
kapt {
    correctErrorTypes = true
}
```

在 Maven 中使用

在 `compile` 之前在 kotlin-maven-plugin 中添加 `kapt` 目标的执行：

```
<execution>
    <id>kapt</id>
    <goals>
        <goal>kapt</goal>
    </goals>
    <configuration>
        <sourceDirs>
            <sourceDir>src/main/kotlin</sourceDir>
            <sourceDir>src/main/java</sourceDir>
        </sourceDirs>
        <annotationProcessorPaths>
            <!-- 在此处指定你的注解处理器。 -->
            <annotationProcessorPath>
                <groupId>com.google.dagger</groupId>
                <artifactId>dagger-compiler</artifactId>
                <version>2.9</version>
            </annotationProcessorPath>
        </annotationProcessorPaths>
    </configuration>
</execution>
```

你可以在 Kotlin 示例版本库 中找到一个显示使用 Kotlin、Maven 和 Dagger 的完整示例项目。

请注意，IntelliJ IDEA 自身的构建系统目前还不支持 kapt。当你想要重新运行注解处理时，请从“Maven Projects”工具栏启动构建。

在命令行中使用

Kapt 编译器插件已随 Kotlin 编译器的二进制发行版分发。

可以使用 kotlinc 选项 `Xplugin` 提供该 JAR 文件的路径来附加该插件：

```
-Xplugin=$KOTLIN_HOME/lib/kotlin-annotation-processing.jar
```

以下是可用选项的列表：

- `sources`（*必需*）：所生成文件的输出路径。
- `classes`（*必需*）：所生成类文件与资源的输出路径。
- `stubs`（*必需*）：存根文件的输出路径。换句话说，一些临时目录。
- `incrementalData`：二进制存根的输出路径。
- `apclasspath`（*可重复*）：注解处理器 JAR 包路径。如果有的多个 JAR 包就传多个 `apclasspath` 选项。
- `apoptions`：注解处理器选项的 base64 编码列表。详见 AP/javac options encoding。
- `javacArguments`：传给 javac 的选项的 base64 编码列表。详见 AP/javac options encoding。
- `processors`：逗号分隔的注解处理器全类名列表。如果指定，kapt 就不会尝试在 `apclasspath` 中查找注解处理器。
- `verbose`：启用详细输出。
- `aptMode`（*必需*）
  - `stubs` ——只生成注解处理所需的存根；
  - `apt` ——只运行注解处理；
  - `stubsAndApt` ——生成存根并运行注解处理。
- `correctErrorTypes`：参见下文。默认未启用。

插件选项格式为：`-P plugin:<plugin id>:<key>=<value>`。选项可以重复。

一个示例：

```
-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
```

生成 Kotlin 代码

Kapt 可生成 Kotlin 代码。是将生成的 Kotlin 源文件写入 `processingEnv.options["kapt.kotlin.generated"]` 所指定的目录，这些文件会与主源代码一起编译。

可以在 kotlin-examples Github 版本库中找到完整的示例。

请注意，对于所生成 Kotlin 文件，Kapt 不支持多轮处理。

AP/javac 选项编码

`apoptions` 与 `javacArguments` 命令行选项接受选项编码映射。这是自己编码选项的方式：

```
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())
}
```

使用 Gradle

为了用 Gradle 构建 Kotlin，你应该设置好 *kotlin-gradle* 插件，将其应用到你的项目中，并且添加 *kotlin-stdlib* 依赖。这些操作也可以在 IntelliJ IDEA 中通过调用 Project action 中的 Tools | Kotlin | Configure Kotlin 自动执行。

插件和版本

使用 `kotlin-gradle-plugin` 编译 Kotlin 源代码和模块.

要使用的 Kotlin 版本通常定义为 `kotlin_version` 属性：

```
buildscript {
    ext.kotlin_version = '1.3.11'

    repositories {
        mavenCentral()
    }

    dependencies {
        classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin_version"
    }
}
```

当通过 Gradle 插件 DSL 与 Gradle Kotlin DSL 使用 Kotlin Gradle 插件 1.1.1 及以上版本时，这不是必需的。

Building Kotlin Multiplatform Projects

Using the `kotlin-multiplatform` plugin for building multiplatform projects is described in Building Multiplatform Projects with Gradle.

针对 JVM

针对 JVM，需要应用 Kotlin 插件：

```
apply plugin: "kotlin"
```

或者，从 Kotlin 1.1.1 起，可以使用 Gradle 插件 DSL 来应用该插件：

```
plugins {
    id "org.jetbrains.kotlin.jvm" version "1.3.11"
}
```

在这个块中的 `version` 必须是字面值，并且不能从其他构建脚本中应用。

对于 Gradle Kotlin DSL，请按以下方式应用插件：

```
plugins {
    kotlin("jvm") version "1.3.11"
}
```

Kotlin 源代码可以与同一个文件夹或不同文件夹中的 Java 源代码混用。默认约定是使用不同的文件夹：

```
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：

```
sourceSets {
    main.kotlin.srcDirs += 'src/main/myKotlin'
    main.java.srcDirs += 'src/main/myJava'
}
```

对于 Gradle Kotlin DSL，请改用 `java.sourceSets { …… }` 配置源集。

针对 JavaScript

当针对 JavaScript 时，须应用不同的插件：

```
apply plugin: "kotlin2js"
```

这个插件只适用于 Kotlin 文件，因此建议将 Kotlin 和 Java 文件分开（如果是同一项目包含 Java 文件的情况）。与针对 JVM 一样，如果不使用默认约定，我们需要使用 *sourceSets* 来指定源代码文件夹：

```
sourceSets {
    main.kotlin.srcDirs += 'src/main/myKotlin'
}
```

除了输出的 JavaScript 文件，该插件默认会创建一个带二进制描述符的额外 JS 文件。如果你是构建其他 Kotlin 模块可以依赖的可重用库，那么该文件是必需的，并且应该与转换结果一起分发。其生成由 `kotlinOptions.metaInfo` 选项控制：

```
compileKotlin2Js {
    kotlinOptions.metaInfo = true
}
```

针对 Android

Android 的 Gradle 模型与普通 Gradle 有点不同，所以如果我们要构建一个用 Kotlin 编写的 Android 项目，我们需要用 *kotlin-android* 插件取代 *kotlin* 插件：

```
buildscript {
    ext.kotlin_version = '1.3.11'

    ……

    dependencies {
        classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin_version"
    }
}
apply plugin: 'com.android.application'
apply plugin: 'kotlin-android'
```

不要忘记配置标准库依赖关系。

Android Studio

如果使用 Android Studio，那么需要在 android 下添加以下内容：

```
android {
  ……

  sourceSets {
    main.java.srcDirs += 'src/main/kotlin'
  }
}
```

这让 Android Studio 知道该 kotlin 目录是源代码根目录，所以当项目模型加载到 IDE 中时，它会被正确识别。或者，你可以将 Kotlin 类放在 Java 源代码目录中，该目录通常位于 `src/main/java`。

配置依赖

除了上面显示的 `kotlin-gradle-plugin` 依赖之外，还需要添加 Kotlin 标准库的依赖：

```
repositories {
    mavenCentral()
}

dependencies {
    compile "org.jetbrains.kotlin:kotlin-stdlib"
}
```

如果针对 JavaScript，请使用 `compile "org.jetbrains.kotlin:kotlin-stdlib-js"` 替代之。

如果是针对 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 增加的额外的扩展函数。使用以下依赖之一来取代 `kotlin-stdlib`：

```
compile "org.jetbrains.kotlin:kotlin-stdlib-jdk7"
compile "org.jetbrains.kotlin:kotlin-stdlib-jdk8"
```

对于 Gradle Kotlin DSL，以下表示法的依赖关系与其等价：

```
dependencies {
    compile(kotlin("stdlib"))
    // 或者以下之一：
    compile(kotlin("stdlib-jdk7"))
    compile(kotlin("stdlib-jdk8"))
}
```

在 Kotlin 1.1.x 中，请使用 `kotlin-stdlib-jre7` 与 `kotlin-stdlib-jre8`。

如果你的项目中使用 Kotlin 反射或者测试设施，你也需要添加相应的依赖：

```
compile "org.jetbrains.kotlin:kotlin-reflect"
testCompile "org.jetbrains.kotlin:kotlin-test"
testCompile "org.jetbrains.kotlin:kotlin-test-junit"
```

或者，对于 Gradle Kotlin DSL：

```
compile(kotlin("reflect"))
testCompile(kotlin("test"))
testCompile(kotlin("test-junit"))
```

从 Kotlin 1.1.2 起，使用 `org.jetbrains.kotlin` group 的依赖项默认使用从已应用的插件获得的版本来解析。你可以用完整的依赖关系符号（如 `compile "org.jetbrains.kotlin:kotlin-stdlib:$kotlin_version"`，或者在 Gradle Kotlin DSL 中用 `kotlin("stdlib", kotlinVersion)`）手动提供其版本。

注解处理

请参见 Kotlin 注解处理工具（`kapt`）的描述。

增量编译

Kotlin 支持 Gradle 中可选的增量编译。增量编译跟踪构建之间源文件的改动，因此只有受这些改动影响的文件才会被编译。

从 Kotlin 1.1.1 起，默认启用增量编译。

有几种方法来覆盖默认设置：

1. 将 `kotlin.incremental=true` 或者 `kotlin.incremental=false` 行添加到一个 `gradle.properties` 或者一个 `local.properties` 文件中；
2. 将 `-Pkotlin.incremental=true` 或 `-Pkotlin.incremental=false` 添加到 Gradle 命令行参数。请注意，这样用法中，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将使增量缓存失效。

请注意，第一次构建不会是增量的。

Coroutines support

Coroutines support is an experimental feature in Kotlin 1.2, so the Kotlin compiler reports a warning when you use coroutines in your project. To turn off the warning, add the following block to your `build.gradle` file:

```
kotlin {
    experimental {
        coroutines 'enable'
    }
}
```

Or, with Gradle Kotlin DSL:

```
import org.jetbrains.kotlin.gradle.dsl.Coroutines
// ...

kotlin.experimental.coroutines = Coroutines.ENABLE
```

Module names

The Kotlin modules that the build produces are named accordingly to the `archivesBaseName` property of the project. If a project has a broad name like `lib` or `jvm`, which is common for subprojects, the Kotlin output files related to the module (`*.kotlin_module`) might clash with those from third-party modules with the same name. This causes problems when a project is packaged into a single archive (e.g. APK).

To avoid this, consider setting a unique `archivesBaseName` manually:

```
archivesBaseName = 'myExampleProject_lib'
```

With Gradle Kotlin DSL, it is:

```
setProperty("archivesBaseName", "myExampleProject_lib")
```

Gradle 构建缓存支持(自 1.2.20 起)

Kotlin 插件支持 Gradle 构建缓存(需要 Gradle 4.3 及以上版本;低版本则禁用缓存)。

由于注解处理器运行的任意代码可能不一定会将任务输入转换为输出、可能访问与修改 Gradle 未跟踪的文件等,因此默认不缓存 kapt 注解处理任务。要启用 kapt 缓存,请将以下列几行添加到构建脚本中:

```
kapt {
    useBuildCache = true
}
```

要禁用所有 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`)。

编译器选项

要指定附加的编译选项，请使用 Kotlin 编译任务的 `kotlinOptions` 属性。

当针对 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`。对于自定义源文件集（source set）这些任务称呼取决于 `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`、`compileReleaseUnitTestKotlin`。

当针对 JavaScript 时，这些任务分别称为 `compileKotlin2Js` 与 `compileTestKotlin2Js`，以及对于自定义源文件集称为 `compile＜Name＞Kotlin2Js`。

要配置单个任务，请使用其名称。示例：

```
compileKotlin {
    kotlinOptions.suppressWarnings = true
}

compileKotlin {
    kotlinOptions {
        suppressWarnings = true
    }
}
```

对于 Gradle Kotlin DSL，首先从项目的 `tasks` 中获取任务：

```
import org.jetbrains.kotlin.gradle.tasks.KotlinCompile
// ……

val compileKotlin: KotlinCompile by tasks

compileKotlin.kotlinOptions.suppressWarnings = true
```

相应地，为 JS 与 Common 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：

```
tasks.withType(org.jetbrains.kotlin.gradle.tasks.KotlinCompile::class.java).all {
    kotlinOptions { …… }
}
```

对于 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		false
`freeCompilerArgs`	附加编译器参数的列表		[]

JVM 与 JS 的公共属性

Name	Description	Possible values	Default value
apiVersion	只允许使用来自捆绑库的指定版本中的声明	"1.0"、"1.1"、"1.2"、"1.3"、"1.4 (EXPERIMENTAL)"	
languageVersion	提供与指定语言版本源代码兼容性	"1.0"、"1.1"、"1.2"、"1.3"、"1.4 (EXPERIMENTAL)"	

JVM 特有的属性

名称	描述	可能的值	默认值
javaParameters	为方法参数生成 Java 1.8 反射的元数据		false
jdkHome	要包含到 classpath 中的 JDK 主目录路径，如果与默认 JAVA_HOME 不同的话		
jvmTarget	生成的 JVM 字节码的目标版本（1.6 或 1.8），默认为 1.6	"1.6"、"1.8"	"1.6"
noJdk	不要在 classpath 中包含 Java 运行时		false
noReflect	不要在 classpath 中包含 Kotlin 反射实现		true
noStdlib	不要在 classpath 中包含 Kotlin 运行时		true

JS 特有的属性

名称	描述	可能的值	默认值
friendModulesDisabled	禁用内部声明导出		false
main	是否要调用 main 函数	"call"、"noCall"	"call"
metaInfo	使用元数据生成 .meta.js 与 .kjsm 文件。用于创建库		true
moduleKind	编译器生成的模块类型	"plain"、"amd"、"commonjs"、"umd"	"plain"
noStdlib	不使用捆绑的 Kotlin stdlib		true
outputFile	输出文件路径		
sourceMap	生成源代码映射（source map）		false
sourceMapEmbedSources	将源代码嵌入到源代码映射中	"never"、"always"、"inlining"	
sourceMapPrefix	源代码映射中路径的前缀		
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。

示例

以下示例显示了配置 Gradle 插件的不同可能性：

- Kotlin
- 混用 Java 与 Kotlin
- Android
- JavaScript

使用 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>1.3.11</kotlin.version>
</properties>
```

依赖

Kotlin 有一个广泛的标准库可用于应用程序。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>
```

如果你是针对 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）。

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>
            <artifactId>kotlin-maven-plugin</artifactId>
            <groupId>org.jetbrains.kotlin</groupId>
            <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 上面：

```xml
<build>
    <plugins>
        <plugin>
            <artifactId>kotlin-maven-plugin</artifactId>
            <groupId>org.jetbrains.kotlin</groupId>
            <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> <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。

协程支持

在 Kotlin 1.2 中协程支持是一项实验性的特性，因此当你在项目中使用协程时 Kotlin 编译器会报警告。可以将以下代码块添加到 `pom.xml` 文件中来关闭这一警告：

```
<configuration>
    <experimentalCoroutines>enable</experimentalCoroutines>
</configuration>
```

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：

```
<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：

```
 java -jar target/mymodule-0.0.1-SNAPSHOT-jar-with-dependencies.jar
```

针对 JavaScript

为了编译 JavaScript 代码，需要使用 `js` 和 `test-js` 目标来执行 `compile`：

```
<plugin>
    <groupId>org.jetbrains.kotlin</groupId>
    <artifactId>kotlin-maven-plugin</artifactId>
    <version>${kotlin.version}</version>
    <executions>
        <execution>
            <id>compile</id>
            <phase>compile</phase>
            <goals>
                <goal>js</goal>
            </goals>
        </execution>
        <execution>
            <id>test-compile</id>
            <phase>test-compile</phase>
            <goals>
                <goal>test-js</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

你还需要更改标准库依赖：

```
<groupId>org.jetbrains.kotlin</groupId>
<artifactId>kotlin-stdlib-js</artifactId>
<version>${kotlin.version}</version>
```

对于单元测试支持，你还需要添加对 `kotlin-test-js` 构件的依赖。

更多信息请参阅以 Maven 入门使用 Kotlin 与 JavaScript 教程。

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>
    <artifactId>kotlin-maven-plugin</artifactId>
    <groupId>org.jetbrains.kotlin</groupId>
    <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 和 JS 的公共属性

名称	属性名	描述	可能的值	默认值
nowarn		不生成警告	true、false	false
languageVersion	kotlin.compiler.languageVersion	提供与指定语言版本源代码兼容性	"1.0"、"1.1"、"1.2"、"1.3"、"1.4 (EXPERIMENTAL)"	
apiVersion	kotlin.compiler.apiVersion	只允许使用来自捆绑库的指定版本中的声明	"1.0"、"1.1"、"1.2"、"1.3"、"1.4 (EXPERIMENTAL)"	
sourceDirs		包含要编译源文件的目录		该项目源代码根目录
compilerPlugins		启用编译器插件		[]
pluginOptions		编译器插件的选项		[]
args		额外的编译器参数		[]

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"、"1.8"	"1.6"
jdkHome	kotlin.compiler.jdkHome	要包含到 classpath 中的 JDK 主目录路径，如果与默认 JAVA_HOME 不同的话		

JS 特有的属性

名称	属性名	描述	可能的值	默认值
outputFile		输出文件路径		
metaInfo		使用元数据生成 .meta.js 与 .kjsm 文件。用于创建库	true、false	true
sourceMap		生成源代码映射（source map）	true、false	false
sourceMapEmbedSources		将源代码嵌入到源代码映射中	"never"、"always"、"inlining"	"inlining"
sourceMapPrefix		源代码映射中路径的前缀		
moduleKind		编译器生成的模块类型	"plain"、"amd"、"commonjs"、"umd"	"plain"

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。

示例

一个示例 Maven 项目可以从 Github 版本库直接下载

使用 Ant

获取 Ant 任务

Kotlin 为 Ant 提供了三个任务：

- kotlinc: 针对 JVM 的 Kotlin 编译器；
- kotlin2js: 针对 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 编译器的 *lib* 文件夹中 需要 Ant 1.8.2+ 版本。

针对 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}/kotlin-ant.jar"/>

    <target name="build">
        <kotlinc src="hello.kt" output="hello.jar"/>
    </target>
</project>
```

其中 `${kotlin.lib}` 指向解压缩 Kotlin 独立编译器所在文件夹。

针对 JVM 只用 Kotlin 源代码且多根

如果项目由多个源代码根组成，那么使用 *src* 作为元素来定义路径：

```
<project name="Ant Task Test" default="build">
    <typedef resource="org/jetbrains/kotlin/ant/antlib.xml"
classpath="${kotlin.lib}/kotlin-ant.jar"/>

    <target name="build">
        <kotlinc output="hello.jar">
            <src path="root1"/>
            <src path="root2"/>
        </kotlinc>
    </target>
</project>
```

针对 JVM 使用 Kotlin 和 Java 源代码

如果项目由 Kotlin 和 Java 源代码组成，虽然可以使用 *kotlinc* 来避免任务参数的重复，但是建议使用 *withKotlin* 任务：

```
<project name="Ant Task Test" default="build">
    <typedef resource="org/jetbrains/kotlin/ant/antlib.xml"
classpath="${kotlin.lib}/kotlin-ant.jar"/>

    <target name="build">
        <delete dir="classes" failonerror="false"/>
        <mkdir dir="classes"/>
        <javac destdir="classes" includeAntRuntime="false" srcdir="src">
            <withKotlin/>
        </javac>
        <jar destfile="hello.jar">
            <fileset dir="classes"/>
        </jar>
    </target>
</project>
```

还可以将正在编译的模块的名称指定为 `moduleName` 属性：

```
<withKotlin moduleName="myModule"/>
```

针对 JavaScript 用单个源文件夹

```
<project name="Ant Task Test" default="build">
    <typedef resource="org/jetbrains/kotlin/ant/antlib.xml"
classpath="${kotlin.lib}/kotlin-ant.jar"/>

    <target name="build">
        <kotlin2js src="root1" output="out.js"/>
    </target>
</project>
```

针对 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}/kotlin-ant.jar"/>

    <target name="build">
        <kotlin2js src="root1" output="out.js" outputPrefix="prefix"
outputPostfix="postfix" sourcemap="true"/>
    </target>
</project>
```

针对 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}/kotlin-ant.jar"/>

    <target name="build">
        <!-- 会创建 out.meta.js,其中包含二进制元数据 -->
        <kotlin2js src="root1" output="out.js" metaInfo="true"/>
    </target>
</project>
```

参考

元素和属性的完整列表如下：

kotlinc 和 kotlin2js 的公共属性

名称	描述	必需	默认值
src	要编译的 Kotlin 源文件或目录	是	
nowarn	禁止所有编译警告	否	false
noStdlib	不要将 Kotlin 标准库包含进 classpath	否	false
failOnError	在编译期间检测到错误时，会导致构建失败	否	true

kotlinc 属性

名称	描述	必需	默认值
output	目标目录或 .jar 文件名	是	
classpath	编译类路径	否	
classpathref	编译类路径引用	否	
includeRuntime	Kotlin 运行时库是否包含在 jar 中，如果 output 是 .jar 文件的话	否	true
moduleName	编译的模块的名称	否	目标（如果指定的话）或项目的名称

kotlin2js 属性

名称	描述	必需
output	目标文件	是
libraries	Kotlin 库的路径	否
outputPrefix	生成的 JavaScript 文件所用前缀	否
outputSuffix	生成的 JavaScript 文件所用后缀	否
sourcemap	是否要生成 sourcemap 文件	否
metaInfo	是否要生成具有二进制描述符的元数据文件	否
main	编译器是否生成调用 main 函数的代码	否

传递原始编译器参数

如需传递原始编译器参数，可以使用带 `value` 或 `line` 属性的 `<compilerarg>` 元素。可以放在 `<kotlinc>`、`<kotlin2js>` 与 `<withKotlin>` 任务元素内，如下所示：

```
<kotlinc src="${test.data}/hello.kt" output="${temp}/hello.jar">
    <compilerarg value="-Xno-inline"/>
    <compilerarg line="-Xno-call-assertions -Xno-param-assertions"/>
    <compilerarg value="-Xno-optimize"/>
</kotlinc>
```

当运行 `kotlinc -help` 时，会显示可以使用的参数的完整列表。

Kotlin 与 OSGi

要启用 Kotlin OSGi 支持，你需要引入 `kotlin-osgi-bundle` 而不是常规的 Kotlin 库。建议删除 `kotlin-runtime`、`kotlin-stdlib` 和 `kotlin-reflect` 依赖，因为 `kotlin-osgi-bundle` 已经包含了所有这些。当引入外部 Kotlin 库时你也应该注意。大多数常规 Kotlin 依赖不是 OSGi-就绪的，所以你不应该使用它们，且应该从你的项目中删除它们。

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：

```
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。

编译器插件

全开放编译器插件

Kotlin 有类及其默认为 `final` 的成员，这使得像 Spring AOP 这样需要类为 `open` 的框架与库用起来很不方便。这个 *all-open* 编译器插件会适配 Kotlin 以满足那些框架的需求，并使用指定的注解标注类而其成员无需显式使用 `open` 关键字打开。

例如，当你使用 Spring 时，你不需要打开所有的类，而只需要使用特定的注解标注，如 `@Configuration` 或 `@Service`。*All-open* 允许指定这些注解。

我们为全开放插件提供 Gradle 与 Maven 支持并有完整的 IDE 集成。

注意：对于 Spring，你可以使用 `kotlin-spring` 编译器插件（见下文）。

在 Gradle 中使用

将插件构件添加到 buildscript 依赖中并应用该插件：

```
buildscript {
    dependencies {
        classpath "org.jetbrains.kotlin:kotlin-allopen:$kotlin_version"
    }
}

apply plugin: "kotlin-allopen"
```

另一种方式是使用 `plugins` 块启用之：

```
plugins {
  id "org.jetbrains.kotlin.plugin.allopen" version "1.3.11"
}
```

然后指定会打开类的注解的列表：

```
allOpen {
    annotation("com.my.Annotation")
    // annotations("com.another.Annotation", "com.third.Annotation")
}
```

如果类（或任何其超类）标有 `com.my.Annotation` 注解，类本身及其所有成员会变为开放。

它也适用于元注解：

```
@com.my.Annotation
annotation class MyFrameworkAnnotation

@MyFrameworkAnnotation
class MyClass // 将会全开放
```

`MyFrameworkAnnotation` 已由全开放元注解 `com.my.Annotation` 标注，所以它也成了一个全开放注解。

在 Maven 中使用

下面是全开放与 Maven 一起使用的用法：

```
<plugin>
    <artifactId>kotlin-maven-plugin</artifactId>
    <groupId>org.jetbrains.kotlin</groupId>
    <version>${kotlin.version}</version>

    <configuration>
        <compilerPlugins>
            <!-- 或者 "spring" 对于 Spring 支持 -->
            <plugin>all-open</plugin>
        </compilerPlugins>

        <pluginOptions>
            <!-- 每个注解都放在其自己的行上 -->
            <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>
```

关于全开放注解如何工作的详细信息，请参考上面的“在 Gradle 中使用”一节。

Spring 支持

如果使用 Spring，可以启用 *kotlin-spring* 编译器插件而不是手动指定 Spring 注解。kotlin-spring 是在全开放之上的一层包装，并且其运转方式也完全相同。

与全开放一样，将该插件添加到 buildscript 依赖中：

```
buildscript {
    dependencies {
        classpath "org.jetbrains.kotlin:kotlin-allopen:$kotlin_version"
    }
}

apply plugin: "kotlin-spring" // 取代 "kotlin-allopen"
```

或者使用 Gradle 插件 DSL：

```
plugins {
  id "org.jetbrains.kotlin.plugin.spring" version "1.3.11"
}
```

在 Maven 中，则启用 `spring` 插件：

```
<compilerPlugins>
    <plugin>spring</plugin>
</compilerPlugins>
```

该插件指定了以下注解：`@Component`、`@Async`、`@Transactional`、`@Cacheable` 以及 `@SpringBootTest`。由于元注解的支持，标注有 `@Configuration`、`@Controller`、`@RestController`、`@Service` 或者 `@Repository` 的类会自动打开，因为这些注解标注有元注解 `@Component`。

当然，你可以在同一个项目中同时使用 `kotlin-allopen` 与 `kotlin-spring`。

请注意，如果使用 start.spring.io 服务生成的项目模板，那么默认会启用 `kotlin-spring` 插件。

在命令行中使用

全开放编译器插件的 JAR 包已随 Kotlin 编译器的二进制发行版分发。可以使用 kotlinc 选项 `Xplugin` 提供该 JAR 文件的路径来附加该插件：

```
-Xplugin=$KOTLIN_HOME/lib/allopen-compiler-plugin.jar
```

可以使用 `annotation` 插件选项或者启用“预设”来直接指定全开放注解。现在可用于全开放的唯一预设是 `spring`。

```
# 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
```

无参编译器插件

无参 (no-arg) 编译器插件为具有特定注解的类生成一个额外的零参数构造函数。

这个生成的构造函数是合成的，因此不能从 Java 或 Kotlin 中直接调用，但可以使用反射调用。

这允许 Java Persistence API（JPA）实例化一个类，就算它从 Kotlin 或 Java 的角度看没有无参构造函数（参见下面的 `kotlin-jpa` 插件的描述）。

在 Gradle 中使用

其用法非常类似于全开放插件。

添加该插件并指定注解的列表，这些注解一定会导致被标注的类生成无参构造函数。

```
buildscript {
    dependencies {
        classpath "org.jetbrains.kotlin:kotlin-noarg:$kotlin_version"
    }
}

apply plugin: "kotlin-noarg"
```

或者使用 Gradle 插件 DSL：

```
plugins {
  id "org.jetbrains.kotlin.plugin.noarg" version "1.3.11"
}
```

然后指定无参注解列表：

```
noArg {
    annotation("com.my.Annotation")
}
```

如果希望该插件在合成的构造函数中运行其初始化逻辑，请启用 `invokeInitializers` 选项。由于在未来会解决的 KT-18667 及 KT-18668，自 Kotlin 1.1.3-2 起，它被默认禁用。

```
noArg {
    invokeInitializers = true
}
```

在 Maven 中使用

```
<plugin>
    <artifactId>kotlin-maven-plugin</artifactId>
    <groupId>org.jetbrains.kotlin</groupId>
    <version>${kotlin.version}</version>

    <configuration>
        <compilerPlugins>
            <!-- 或者对于 JPA 支持用 "jpa" -->
            <plugin>no-arg</plugin>
        </compilerPlugins>

        <pluginOptions>
            <option>no-arg:annotation=com.my.Annotation</option>
            <!-- 在合成的构造函数中调用实例初始化器 -->
            <!-- <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

与 *kotlin-spring* 插件类似，*kotlin-jpa* 是在 *no-arg* 之上的一层包装。该插件自动指定了 `@Entity`、`@Embeddable` 与 `@MappedSuperclass` 这几个 *无参* 注解。

这是在 Gradle 中添加该插件的方法：

```
buildscript {
    dependencies {
        classpath "org.jetbrains.kotlin:kotlin-noarg:$kotlin_version"
    }
}

apply plugin: "kotlin-jpa"
```

或者使用 Gradle 插件 DSL：

```
plugins {
  id "org.jetbrains.kotlin.plugin.jpa" version "1.3.11"
}
```

在 Maven 中，则启用 `jpa` 插件：

```
<compilerPlugins>
    <plugin>jpa</plugin>
</compilerPlugins>
```

在命令行中使用

与全开放类似，将插件 JAR 文件添加到编译器插件类路径并指定注解或预设：

```
-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

编译器插件 *sam-with-receiver* 使所注解的 Java“单抽象方法”接口方法的第一个参数成为 Kotlin 中的接收者。这一转换只适用于当 SAM 接口作为 Kotlin 的 lambda 表达式传递时，对 SAM 适配器与 SAM 构造函数均适用（详见其文档）。

这里有一个示例：

```
public @interface SamWithReceiver {}

@SamWithReceiver
public interface TaskRunner {
    void run(Task task);
}
```

```
fun test(context: TaskContext) {
    val runner = TaskRunner {
        // 这里的“this”是“Task”的一个实例

        println("$name is started")
        context.executeTask(this)
        println("$name is finished")
    }
}
```

在 Gradle 中使用

除了事实上 sam-with-receiver 没有任何内置预设、并且需要指定自己的特殊处理注解列表外，其用法与 all-open 及 no-arg 相同。

```
buildscript {
    dependencies {
        classpath "org.jetbrains.kotlin:kotlin-sam-with-receiver:$kotlin_version"
    }
}

apply plugin: "kotlin-sam-with-receiver"
```

然后指定 SAM-with-receiver 的注解列表：

```
samWithReceiver {
    annotation("com.my.Annotation")
}
```

在 Maven 中使用

```
<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>
```

在 CLI 中使用

只需将该插件的 JAR 文件添加到编译器插件类路径中，并指定 sam-with-receiver 注解列表即可：

```
-Xplugin=$KOTLIN_HOME/lib/sam-with-receiver-compiler-plugin.jar
-P plugin:org.jetbrains.kotlin.samWithReceiver:annotation=com.my.SamWithReceiver
```

Code Style Migration Guide

Kotlin Coding Conventions and IntelliJ IDEA formatter

Kotlin Coding Conventions affect several aspects of writing idiomatic Kotlin, and a set of formatting recommendations aimed at improving Kotlin code readability is among them.

Unfortunately, the code formatter built into IntelliJ IDEA had to work long before this document was released and now has a default setup that produces different formatting from what is now recommended.

It may seem a logical next step to remove this obscurity by switching the defaults in IntelliJ IDEA and make formatting consistent with the Kotlin Coding Conventions. But this would mean that all the existing Kotlin projects will have a new code style enabled the moment the Kotlin plugin is installed. Not really the expected result for plugin update, right?

That's why we have the following migration plan instead:

- Enable the official code style formatting by default starting from Kotlin 1.3 and only for new projects (old formatting can be enabled manually)
- Authors of existing projects may choose to migrate to the Kotlin Coding Conventions
- Authors of existing projects may choose to explicitly declare using the old code style in a project (this way the project won't be affected by switching to the defaults in the future)
- Switch to the default formatting and make it consistent with Kotlin Coding Conventions in Kotlin 1.4

Differences between "Kotlin Coding Conventions" and "IntelliJ IDEA default code style"

The most notable change is in the continuation indentation policy. There's a nice idea to use the double indent for showing that a multi-line expression hasn't ended on the previous line. This is a very simple and general rule, but several Kotlin constructions look a bit awkward when they are formatted this way. In Kotlin Coding Conventions it's recommended to use a single indent in cases where the long continuation indent has been forced before

In practice, quite a bit of code is affected, so this can be considered a major code style update.

Migration to a new code style discussion

A new code style adoption might be a very natural process if it starts with a new project, when there's no code formatted in the old way. That is why starting from version 1.3, the Kotlin IntelliJ Plugin creates new projects with formatting from the Code Conventions document which is enabled by default.

Changing formatting in an existing project is a far more demanding task, and should probably be started with discussing all the caveats with the team.

The main disadvantage of changing the code style in an existing project is that the blame/annotate VCS feature will point to irrelevant commits more often. While each VCS has some kind of way to deal with this problem ("Annotate Previous Revision" can be used in IntelliJ IDEA), it's important to decide if a new style is worth all the effort. The practice of separating reformatting commits from meaningful changes can help a lot with later investigations.

Also migrating can be harder for larger teams because committing a lot of files in several subsystems may produce merging conflicts in personal branches. And while each conflict resolution is usually trivial, it's still wise to know if there are large feature branches currently in work.

In general, for small projects, we recommend converting all the files at once.

For medium and large projects the decision may be tough. If you are not ready to update many files right away you may decide to migrate module by module, or continue with gradual migration for modified files only.

Migration to a new code style

Switching to the Kotlin Coding Conventions code style can be done in `Settings → Editor → Code Style → Kotlin` dialog. Switch scheme to *Project* and activate `Set from... → Predefined Style → Kotlin Style Guide`.

In order to share those changes for all project developers `.idea/codeStyle` folder have to be committed to VCS.

If an external build system is used for configuring the project, and it's been decided not to share `.idea/codeStyle` folder, Kotlin Coding Conventions can be forced with an additional property:

In Gradle

Add **kotlin.code.style**=**official** property to the **gradle.properties** file at the project root and commit the file to VCS.

In Maven

Add **kotlin.code.style official** property to root **pom.xml** project file.

```
<properties>
  <kotlin.code.style>official</kotlin.code.style>
</properties>
```

Warning: having the **kotlin.code.style** option set may modify the code style scheme during a project import and may change the code style settings.

After updating your code style settings, activate “Reformat Code” in the project view on the desired scope.

For a gradual migration, it's possible to enable the *"File is not formatted according to project settings"* inspection. It will highlight the places that should be reformatted. After enabling the *"Apply only to modified files"* option, inspection will show formatting problems only in modified files. Such files are probably going to be committed soon anyway.

Store old code style in project

It's always possible to explicitly set the IntelliJ IDEA code style as the correct code style for the project. To do so please switch to the *Project* scheme in `Settings → Editor → Code Style → Kotlin` and select *"Kotlin obsolete IntelliJ IDEA codestyle"* in the *"Use defaults from:"* on the *Load* tab.

In order to share the changes across the project developers `.idea/codeStyle` folder, it has to be committed to VCS. Alternatively **kotlin.code.style**=**obsolete** can be used for projects configured with Gradle or Maven.

演进

Kotlin 演进

实用主义演进原则

> 语言的设计是石头铸造的（译注：原文为“cast in stone”，意为“最终定论，板上钉钉”，此处双关），
>
> 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更改（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更改，将相应内容标记为已弃用并 *在更改发生之前* 提供自动化的迁移工具。当语言发生更改之时，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更改的数量。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，只是带有_实验性_ 状态。实验性特性并不稳定，可以随时更改，选择使用它们的用户明确表示已准备好了应对未来的迁移问题。这些用户提供了宝贵的反馈，而我们收集这些反馈来迭代设计并使其坚如磐石。

不兼容的变更

如果从一个版本更新到另一个版本时，一些以前工作的代码不再工作，那么它是语言中的 *不兼容的变更*（有时称为“破坏性变更”）。在一些场景中“不再工作”的确切含义可能会有争议，但是它肯定包含以下内容：

— 之前编译运行正常的代码现在（编译或链接）失败并报错。这包括删除语言结构以及添加新的限制。
— 之前正常执行的代码现在抛异常了。

The less obvious cases that belong to the "grey area" include handling corner cases differently, throwing an exception of a different type than before, changing behavior observable only through reflection, changes in undocumented/undefined behavior, renaming binary artifacts, etc. Sometimes such changes are very important and affect migration experience dramatically, sometimes they are insignificant.

绝对不是不兼容的变更的一些示例包括

— Adding new warnings.
— Enabling new language constructs or relaxing limitations for existing ones.
— Changing private/internal APIs and other implementation details.

The principles of Keeping the Language Modern and Comfortable Updates suggest that incompatible changes are sometimes necessary, but they should be introduced carefully. Our goal is to make the users aware of upcoming changes well in advance to let them migrate their code comfortably.

Ideally, every incompatible change should be announced through a compile-time warning reported in the problematic code (usually referred to as a *deprecation warning*) and accompanied with automated migration aids. So, the ideal migration workflow goes as follows:

— Update to version A (where the change is announced)
  — See warnings about the upcoming change
  — Migrate the code with the help of the tooling
— Update to version B (where the change happens)
  — See no issues at all

In practice some changes can't be accurately detected at compile time, so no warnings can be reported, but at least the users will be notified through Release notes of version A that a change is coming in version B.

Dealing with compiler bugs

Compilers are complicated software and despite the best effort of their developers they have bugs. The bugs that cause the compiler itself to fail or report spurious errors or generate obviously failing code, though annoying and often embarrassing, are easy to fix, because the fixes do not constitute incompatible changes. Other bugs may cause the compiler to generate incorrect code that does not fail: e.g. by missing some errors in the source or simply generating wrong instructions. Fixes of such bugs are technically incompatible changes (some code used to compile fine, but now it won't any more), but we are inclined to fixing them as soon as possible to prevent the bad code patterns from spreading across user code. In our opinion, this serves the principle of Comfortable Updates, because fewer users have a chance of encountering the issue. Of course, this applies only to bugs that are found soon after appearing in a released version.

Decision Making

JetBrains, the original creator of Kotlin, is driving its progress with the help of the community and in accord with the Kotlin Foundation.

All changes to the Kotlin Programming Language are overseen by the Lead Language Designer (currently Andrey Breslav). The Lead Designer has the final say in all matters related to language evolution. Additionally, incompatible changes to fully stable components have to be approved to by the Language Committee designated under the Kotlin Foundation (currently comprised of Jeffrey van Gogh, William R. Cook and Andrey Breslav).

The Language Committee makes final decisions on what incompatible changes will be made and what exact measures should be taken to make user updates comfortable. In doing so, it relies on a set of guidelines available here.

Feature Releases and Incremental Releases

Stable releases with versions 1.2, 1.3, etc. are usually considered to be *feature releases* bringing major changes in the language. Normally, we publish *incremental releases*, numbered 1.2.20, 1.2.30, etc, in between feature releases.

Incremental releases bring updates in the tooling (often including features), performance improvements and bug fixes. We try to keep such versions compatible with each other, so changes to the compiler are mostly optimizations and warning additions/removals. Experimental features may, of course, be added, removed or changed at any time.

Feature releases often add new features and may remove or change previously deprecated ones. Feature graduation from experimental to stable also happens in feature releases.

EAP Builds

Before releasing stable versions, we usually publish a number of preview builds dubbed EAP (for "Early Access Preview") that let us iterate faster and gather feedback from the community. EAPs of feature releases usually produce binaries that will be later rejected by the stable compiler to make sure that possible bugs in the binary format survive no longer than the preview period. Final Release Candidates normally do not bear this limitation.

Experimental features

According to the Feedback Loop principle described above, we iterate on our designs in the open and release versions of the language where some features have the *experimental* status and *are supposed to change*. Experimental features can be added, changed or removed at any point and without warning. We make sure that experimental features can't be used accidentally by an unsuspecting user. Such features usually require some sort of an explicit opt-in either in the code or in the project configuration.

Experimental features usually graduate to the stable status after some iterations.

Status of different components

To check the stability status of different components of Kotlin (Kotlin/JVM, JS, Native, various libraries, etc), please consult this link.

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.

Library authors can use the @Deprecated and @Experimental 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 experimental 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 and -api-version flags that make a new version emulate the behaviour of an old one, for compatibility purposes. Normally, at least one previous version is supported. This effectively leaves a time span of two full feature release cycles for migration (which usually amounts to about two years). 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.

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).

不同组件的稳定性

依据组件的发展速度，可以有不同的稳定性模式：

- **快速流转（MF, Moving fast）**：即使在增量版本之间也不要期待任何兼容性，任何功能都可以在没有警告的情况下添加、删除或者更改。
- **有功能添加的增量版本（AIR, Additions in Incremental Releases）**：可以在增量版本中添加内容，应避免删除与更改行为，而如果必须要删改的话，应在之前的增量版本中预告。
- **稳定增量版本（SIR, Stable Incremental Releases）**：增量版本完全兼容，只会有优化与 bug 修复。可以在特性版本中进行任何更改。

- **完全稳定（FS, Fully Stable）**：增量版本完全兼容，特性版本兼容旧版。

对于相同的组件，源代码兼容性与二进制兼容性可以有不同的模式，例如，在二进制格式稳定之前，源代码语言可以达到完全稳定，反之亦然。

Kotlin 演进制度的条款只适用于已经达到完全稳定（FS）的组件。从那一刻起，不兼容的变更必须得到语言委员会的批注。

组件	进入该状态的版本	源代码兼容模式	二进制兼容模式
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**

Compatibility Guide for Kotlin 1.3

Keeping the Language Modern and *Comfortable Updates* are among the fundamental principles in Kotlin Language Design. The former says that constructions 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

One has to 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.

Evaluation order of constructor arguments regarding `<clinit>` call

Issue: KT-19532

Component: Kotlin/JVM

Incompatible change type: behavioral

Short summary: evaluation order with respect to class initialization is changed in 1.3

Deprecation cycle:

— <1.3: old behavior (see details in the Issue)

— >= 1.3: behavior changed, `-Xnormalize-constructor-calls=disable` can be used to temporarily revert to pre-1.3 behavior. Support for this flag is going to be removed in the next major release.

Missing getter-targeted annotations on annotation constructor parameters

Issue: KT-25287

Component: Kotlin/JVM

Incompatible change type: behavioral

Short summary: getter-target annotations on annotations constructor parameters will be properly written to classfiles in 1.3

Deprecation cycle:

— <1.3: getter-target annotations on annotation constructor parameters are not applied
— >=1.3: getter-target annotations on annotation constructor parameters are properly applied and written to the generated code

Missing errors in class constructor's `@get:` annotations

Issue: KT-19628

Component: Core language

Incompatible change type: Source

Short summary: errors in getter-target annotations will be reported properly in 1.3

Deprecation cycle:

— <1.2: compilation errors in getter-target annotations were not reported, causing incorrect code to be compiled fine.
— 1.2.x: errors reported only by tooling, the compiler still compiles such code without any warnings
— >=1.3: errors reported by the compiler too, causing erroneous code to be rejected

Nullability assertions on access to Java types annotated with `@NotNull`

Issue: KT-20830

Component: Kotlin/JVM

Incompatible change type: Behavioral

Short summary: nullability assertions for Java-types annotated with not-null annotations will be generated more aggressively, causing code which passes `null` here to fail faster.

Deprecation cycle:

— <1.3: the compiler could miss such assertions when type inference was involved, allowing potential `null` propagation during compilation against binaries (see Issue for details).

— >=1.3: the compiler generates missed assertions. This can case code which was (erroneously) passing `null`s here fail faster.
`-XXLanguage:-StrictJavaNullabilityAssertions` can be used to temporarily return to the pre-1.3 behavior. Support for this flag will be removed in the next major release.

Unsound smartcasts on enum members

Issue: KT-20772

Component: Core language

Incompatible change type: Source

Short summary: a smartcast on a member of one enum entry will be correctly applied to only this enum entry

Deprecation cycle:

— <1.3: a smartcast on a member of one enum entry could lead to an unsound smartcast on the same member of other enum entries.

— >=1.3: smartcast will be properly applied only to the member of one enum entry.
`-XXLanguage:-SoundSmartcastForEnumEntries` will temporarily return old behavior. Support for this flag will be removed in the next major release.

`val` backing field reassignment in getter

Issue: KT-16681

Components: Core language

Incompatible change type: Source

Short summary: reassignment of the backing field of `val`-property in its getter is now prohibited

Deprecation cycle:

— <1.2: Kotlin compiler allowed to modify backing field of `val` in its getter. Not only it violates Kotlin semantic, but also generates ill-behaved JVM bytecode which reassigns `final` field.

— 1.2.X: deprecation warning is reported on code which reassigns backing field of `val`

— >=1.3: deprecation warnings are elevated to errors

Array capturing before the `for`-loop where it is iterated

Issue: KT-21354

Component: Kotlin/JVM

Incompatible change type: Source

Short summary: if an expression in for-loop range is a local variable updated in a loop body, this change affects loop execution. This is inconsistent with iterating over other containers, such as ranges, character sequences, and collections.

Deprecation cycle:

— <1.2: described code patterns are compiled fine, but updates to local variable affect loop execution

— 1.2.X: deprecation warning reported if a range expression in a for-loop is an array-typed local variable which is assigned in a loop body

— 1.3: change behavior in such cases to be consistent with other containers

Nested classifiers in enum entries

Issue: KT-16310

Component: Core language

Incompatible change type: Source

Short summary: since Kotlin 1.3, nested classifiers (classes, object, interfaces, annotation classes, enum classes) in enum entries are prohibited

Deprecation cycle:

— <1.2: nested classifiers in enum entries are compiled fine, but may fail with exception at runtime
— 1.2.X: deprecation warnings reported on the nested classifiers
— >=1.3: deprecation warnings elevated to errors

Data class overriding `copy`

Issue: KT-19618

Components: Core language

Incompatible change type: Source

Short summary: since Kotlin 1.3, data classes are prohibited to override `copy()`

Deprecation cycle:

— <1.2: data classes overriding `copy()` are compiled fine but may fail at runtime/expose strange behavior
— 1.2.X: deprecation warnings reported on data classes overriding `copy()`
— >=1.3: deprecation warnings elevated to errors

Inner classes inheriting `Throwable` that capture generic parameters from the outer class

Issue: KT-17981

Component: Core language

Incompatible change type: Source

Short summary: since Kotlin 1.3, inner classes are not allowed to inherit `Throwable`

Deprecation cycle:

— <1.2: inner classes inheriting `Throwable` are compiled fine. If such inner classes happen to capture generic parameters, it could lead to strange code patterns which fail at runtime.
— 1.2.X: deprecation warnings reported on inner classes inheriting `Throwable`
— >=1.3: deprecation warnings elevated to errors

Visibility rules regarding complex class hierarchies with companion objects

Issues: KT-21515, KT-25333

Component: Core language

Incompatible change type: Source

Short summary: since Kotlin 1.3, rules of visibility by short names are stricter for complex class hierarchies involving companion objects and nested classifiers.

Deprecation cycle:

— <1.2: old visibility rules (see Issue for details)
— 1.2.X: deprecation warnings reported on short names which are not going to be accessible anymore. Tooling suggests automated migration by adding full name.
— >=1.3: deprecation warnings elevated to errors. Offending code should add full qualifiers or explicit imports

Non-constant vararg annotation parameters

Issue: KT-23153

Component: Core language

Incompatible change type: Source

Short summary: since Kotlin 1.3, setting non-constant values as vararg annotation parameters is prohibited

Deprecation cycle:

— <1.2: the compiler allows to pass non-constant value for vararg annotation parameter, but actually drops that value during bytecode generation, leading to non-obvious behavior
— 1.2.X: deprecation warnings reported on such code patterns
— >=1.3: deprecation warnings elevated to errors

Local annotation classes

Issue: KT-23277

Component: Core language

Incompatible change type: Source

Short summary: since Kotlin 1.3 local annotation classes are not supported

Deprecation cycle:

— <1.2: the compiler compiled local annotation classes fine
— 1.2.X: deprecation warnings reported on local annotation classes
— >=1.3: deprecation warnings elevated to errors

Smartcasts on local delegated properties

Issue: KT-22517

Component: Core language

Incompatible change type: Source

Short summary: since Kotlin 1.3 smartcasts on local delegated properties are not allowed

Deprecation cycle:

— <1.2: the compiler allowed to smartcast local delegated property, which could lead to unsound smartcast in case of ill-behaved delegates
— 1.2.X: smartcasts on local delegated properries are reported as deprecated (the compiler issues warnings)
— >=1.3: deprecation warnings elevated to errors

`mod` operator convention

Issues: KT-24197

Component: Core language

Incompatible change type: Source

Short summary: since Kotlin 1.3 declaration of `mod` operator is prohibited, as well as calls which resolve to such declarations

Deprecation cycle:

— 1.1.X, 1.2.X: report warnings on declarations of `operator mod`, as well as on calls which resolve to it
— 1.3.X: elevate warnings to error, but still allow to resolve to `operator mod` declarations
— 1.4.X: do not resolve calls to `operator mod` anymore

Passing single element to vararg in named form

Issues: KT-20588, KT-20589. See also KT-20171

Component: Core language

Incompatible change type: Source

Short summary: in Kotlin 1.3, assigning single element to vararg is deprecated and should be replaced with consecutive spread and array construction.

Deprecation cycle:

- <1.2: assigning one value element to vararg in named form compiles fine and is treated as assigning *single* element to array, causing non-obvious behavior when assigning array to vararg
- 1.2.X: deprecation warnings are reported on such assignments, users are suggested to switch to consecutive spread and array construction.
- 1.3.X: warnings are elevated to errors
- >= 1.4: change semantic of assigning single element to vararg, making assignment of array equivalent to the assignment of a spread of an array

Retention of annotations with target `EXPRESSION`

Issue: KT-13762

Component: Core language

Incompatible change type: Source

Short summary: since Kotlin 1.3, only `SOURCE` retention is allowed for annotations with target `EXPRESSION`

Deprecation cycle:

- <1.2: annotations with target `EXPRESSION` and retention other than `SOURCE` are allowed, but silently ignored at use-sites
- 1.2.X: deprecation warnings are reported on declarations of such annotations
- >=1.3: warnings are elevated to errors

Annotations with target `PARAMETER` shouldn't be applicable to parameter's type

Issue: KT-9580

Component: Core language

Incompatible change type: Source

Short summary: since Kotlin 1.3, error about wrong annotation target will be properly reported when annotation with target PARAMETER is applied to parameter's type

Deprecation cycle:

— <1.2: aforementioned code patterns are compiled fine; annotations are silently ignored and not present in the bytecode

— 1.2.X: deprecation warnings are reported on such usages

— >=1.3: warnings are elevated to errors

`Array.copyOfRange` throws an exception when indices are out of bounds instead of enlarging the returned array

Issue: KT-19489

Component: kotlin-stdlib (JVM)

Incompatible change type: Behavioral

Short summary: since Kotlin 1.3, ensure that the `toIndex` argument of `Array.copyOfRange`, which represents the exclusive end of the range being copied, is not greater than the array size and throw `IllegalArgumentException` if it is.

Deprecation cycle:

— <1.3: in case `toIndex` in the invocation of `Array.copyOfRange` is greater than the array size, the missing elements in range fill be filled with `null`s, violating soundness of the Kotlin type system.

— >=1.3: check that `toIndex` is in the array bounds, and throw exception if it isn't

Progressions of ints and longs with a step of `Int.MIN_VALUE` and `Long.MIN_VALUE` are outlawed and won't be allowed to be instantiated

Issue: KT-17176

Component: kotlin-stdlib (JVM)

Incompatible change type: Behavioral

Short summary: since Kotlin 1.3, prohibit step value for integer progressions being the minimum negative value of its integer type (`Long` or `Int`), so that calling `IntProgression.fromClosedRange(0, 1, step = Int.MIN_VALUE)` will throw `IllegalArgumentException`

Deprecation cycle:

— <1.3: it was possible to create an `IntProgression` with `Int.MIN_VALUE` step, which yields two values `[0, -2147483648]`, which is non-obvious behavior

— >=1.3: throw `IllegalArgumentException` if the step is the minimum negative value of its integer type

Check for index overflow in operations on very long sequences

Issue: KT-16097

Component: kotlin-stdlib (JVM)

Incompatible change type: Behavioral

Short summary: since Kotlin 1.3, make sure `index`, `count` and similar methods do not overflow for long sequences. See the Issue for the full list of affected methods.

Deprecation cycle:

— <1.3: calling such methods on very long sequences could produce negative results due to integer overflow

— >=1.3: detect overflow in such methods and throw exception immediately

Unify split by an empty match regex result across the platforms

Issue: KT-21049

Component: kotlin-stdlib (JVM)

Incompatible change type: Behavioral

Short summary: since Kotlin 1.3, unify behavior of `split` method by empty match regex across all platforms

Deprecation cycle:

— <1.3: behavior of described calls is different when comparing JS, JRE 6, JRE 7 versus JRE 8+

— >=1.3: unify behavior across the platforms

Discontinued deprecated artifacts in the compiler distribution

Issue: KT-23799

Component: other

Incompatible change type: Binary

Short summary: Kotlin 1.3 discontinues the following deprecated binary artifacts:

— `kotlin-runtime`: use `kotlin-stdlib` instead

— `kotlin-stdlib-jre7/8`: use `kotlin-stdlib-jdk7/8` instead

— `kotlin-jslib` in the compiler distribution: use `kotlin-stdlib-js` instead

Deprecation cycle:

— 1.2.X: the artifacts were marked as deprecated, the compiler reported warning on usage of those artifacts

— >=1.3: the artifacts are discontinued

Annotations in stdlib

Issue: KT-21784

Component: kotlin-stdlib (JVM)

Incompatible change type: Binary

Short summary: Kotlin 1.3 removes annotations from the package `org.jetbrains.annotations` from stdlib and moves them to the separate artifacts shipped with the compiler: `annotations-13.0.jar` and `mutability-annotations-compat.jar`

Deprecation cycle:

— <1.3: annotations were shipped with the stdlib artifact

— >=1.3: annotations ship in separate artifacts

常见问题

FAQ

Kotlin 是什么?

Kotlin 是一门针对 JVM、Android、JavaScript 以及原生平台的开源（OSS）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 1.3.11，发布于 2018-12-06。

Kotlin 是免费的吗?

是。Kotlin 是免费的，已经免费并会保持免费。它是遵循 Apache 2.0 许可证开发的，其源代码可以在 GitHub 上获得。

Kotlin 是面向对象还是函数式语言?

Kotlin 既具有面向对象又具有函数式结构。你既可以按 OO 风格也可以按 FP 风格使用，还可以混合使用两种风格。通过对诸如高阶函数、函数类型和 lambda 表达式等功能的一等支持，Kotlin 是一个很好的选择，如果你正在进行或探索函数式编程的话。

Kotlin 能给我超出 Java 语言的哪些优点?

Kotlin 更简洁。粗略估计显示，代码行数减少约 40%。它也更安全，例如对不可空类型的支持使应用程序不易发生 NPE。其他功能包括智能类型转换、高阶函数、扩展函数和带接收者的 lambda 表达式，提供了编写富于表现力的代码的能力以及易于创建 DSL 的能力。

Kotlin 与 Java 语言兼容吗?

兼容。Kotlin 与 Java 语言可以 100% 互操作，并且主要强调确保你现有的代码库可以与Kotlin 正确交互。你可以轻松地在 Java 中调用 Kotlin 代码以及在 Kotlin 中调用 Java 代码。这使得采用 Kotlin 更容易、风险更低。内置于 IDE 的自动化 Java 到 Kotlin 转换器可简化现有代码的迁移。

我可以用 Kotlin 做什么?

Kotlin 可用于任何类型的开发，无论是服务器端、客户端 Web 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?

原生 Kotlin（Kotlin/Native）目前正在准备中。它将 Kotlin 编译为可以无需 VM 运行的原生代码。有一个技术预览发布版，但它还不能用于生产，并且 1.0 我们还没有针对所有平台支持的计划。更多信息请查看 Kotlin/Native 博文公告。

哪些 IDE 支持 Kotlin?

所有主要的 Java IDE 都支持 Kotlin，包括 IntelliJ IDEA、Android Studio、Eclipse 和 NetBeans。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?

当针对JVM 平台时，Kotlin 生成 Java 兼容的字节码。当针对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。当针对原生平台时，Kotlin 会（通过 LLVM）生成平台相关的代码。

Kotlin 只针对 Java 6 吗?

不是。Kotlin 可以让你选择生成 Java 6 或者 Java 8 兼容的字节码。可以为较高版本的平台生成更优化的字节码。

Kotlin 难吗?

Kotlin 是受 Java、C#、JavaScript、Scala 以及 Groovy 等现有语言的启发。我们已经努力确保 Kotlin 易于学习，所以人们可以在几天之内轻松转向、阅读和编写 Kotlin。学习惯用的 Kotlin 和使用更多它的高级功能可能需要一点时间，但总体来说这不是一个复杂的语言。

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 还有 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 40+）。其首席语言设计师是 Andrey Breslav。除了核心团队，GitHub 上还有 100 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、在线尝试以及访问资源、参考文档和教程。

有没有关于 Kotlin 的书?

已经有一些关于 Kotlin 的书籍。其中包括由 Kotlin 团队成员 Dmitry Jemerov 和 Svetlana Isakova 合著的 Kotlin in Action、针对 Android 开发人员的 Kotlin for Android Developers。

Kotlin 有没有在线课程?

有一些 Kotlin 的课程，包括 Kevin Jones 的 Pluralsight Kotlin Course、Hadi Hariri 的 O' Reilly Course 以及 Peter Sommerhoff 的 Udemy Kotlin Course。

在 YouTube 和 Vimeo 上也有许多 Kotlin 演讲 的录像。

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 2018 年 10 月有近 20000 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 年在阿姆斯特丹举行。Kotlin 也会在全球不同地方举行大会。你可以在网站上找到即将到来的会谈列表。

Kotlin 上社交媒体吗？

上。最活跃的 Kotlin 帐号是 Twitter 上的。还有一个 Google+ 群组。

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。请遵循压缩包内 `guidelines.pdf` 中的简单规则。

与 Java 语言比较

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
- 静态成员
- 非私有化字段
- 通配符类型
- 三目操作符 `a ? b : c`

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
- 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

与 Scala 比较【官方已删除】

Kotlin 团队的主要目标是创建一种务实且高效的编程语言，而不是提高编程语言研究中的最新技术水平。考虑到这一点，如果你对 Scala 感到满意，那你很可能不需要 Kotlin。

Scala 有而 Kotlin没有的东西

- 隐式转换、参数……等等
  - 在 Scala 中，由于画面中有太多的隐式转换，有时不使用 debugger 会很难弄清代码中具体发生了什么
  - 在 Kotlin 中使用扩展函数来给类型扩充功能/函数（双关：functions）。
- 可覆盖的类型成员
- 路径依赖性类型
- 宏
- 存在类型
  - 类型投影是一种非常特殊的情况
- 特性（trait）初始化的复杂逻辑
  - 参见类与接口
- 自定义符号操作
  - 参见操作符重载
- 结构类型
- 值类型
  - 我们计划支持Project Valhalla当它作为 JDK 一部分发布时。
- yield 操作符与 actor
  - 参见协程
- 并行集合
  - Kotlin 支持 Java 8 streams，它提供了类似的功能

Kotlin 有而 Scala 没有的东西

- 零开销空安全
  - Scala 有 Option，它是一个语法糖以及运行时的包装器
- 智能转换
- Kotlin的内联函数便于非局部跳转
- 一等公民的委托。也通过第三方插件 Autoproxy 实现