Kotlin 官方参考文档

中文版

目录

Kotlin 官方参考文档 中文版

本书是 Kotlin 官方文档的参考（reference）部分的中文翻译，内容来自 Kotlin 中文站项目。

本书会与 Kotlin 中文站及 JetBrains 的 Kotlin 官方站准同步更新。

请在这里反馈问题。

参考

提供关于 Kotlin 语言的完整参考以及标准库。

从哪开始

这个参考是为让你很容易地在几个小时内学习 Kotlin 而设计的。 先从基本语法开始，然后再到更高级主题。 阅读时你可以在在线 IDE 中尝试代码示例。

一旦你认识到 Kotlin 是什么样的，尝试自己解决一些 Kotlin 心印交互式编程练习。 如果你不确定如何解决一个心印，或者你正在寻找一个更优雅的解决方案，看看 Kotlin 习惯用法。

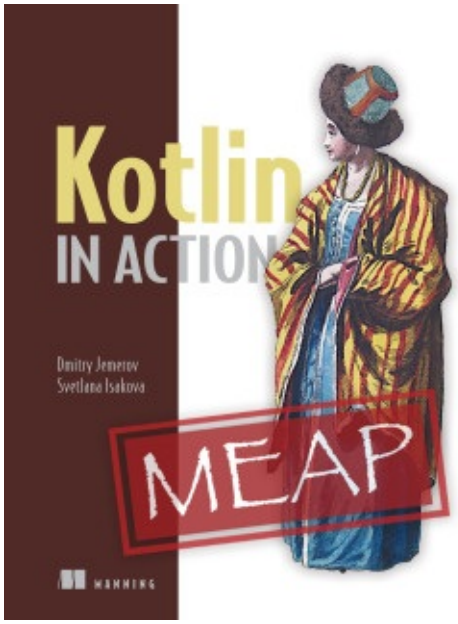
> 心印：Koan，佛教用语，不建议译作“公案”——译者注

离线浏览

下载离线文档 PDF 文件。

参考书

Kotlin in Action



Kotlin in Action 是由 Kotlin 开发团队的 Dmitry Jemerov 和 Svetlana Isakova 合著的 一本关于 Kotlin 的书。这本书目前可通过 MEAP 计划获得，它允许你在本书编写中 逐章去读、而当编写完成时获得本书最终版。

使用优惠码“39jemerov”这本书可优惠 39%。

Kotlin for Android Developers

Kotlin for Android Developers 是由 Antonio Leiva 所著展示 如何使用 Kotlin 从头创建一个 Android 应用程序的一本书。

Modern Web Development with Kotlin

Modern Web Development with Kotlin 是一本由Denis Kalinin撰写的关于 Kotlin Web 开发的书。它涵盖了刚好足够入门的基础知识，但主要集中于语言的使用实践方面。特别是，它指引你使用流行的后端与前端技术完成构建一个技术聚焦的 Web 应用程序的过程。

开始

开始熟悉 Kotlin 基础包括操作、编码习惯和习惯用法。

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 {
    print(a + b)
}
```

`Unit` 返回类型可以省略：

```
fun printSum(a: Int, b: Int) {
  print(a + b)
}
```

参见函数。

定义局部变量

一次赋值（只读）的局部变量:

```
val a: Int = 1
val b = 1  // 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 = 1      // 明确赋值
```

可变变量：

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

参见属性和字段。

注释

正如 Java 和 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。

参见生成 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

```
fun main(args: Array<String>) {
    if (args.size == 0) return

    print("First argument: ${args[0]}")
}
```

参见字符串模板。

使用条件表达式

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

使用 `if` 作为表达式:

```
fun max(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 main(args: Array<String>) {
    if (args.size < 2) {
        print("Two integers expected")
        return
    }

    val x = parseInt(args[0])
    val y = parseInt(args[1])

    // 直接使用 `x * y` 可能会报错，因为他们可能为 null
    if (x != null && y != null) {
        // 在空检测后，x 和 y 会自动转换为非空值（non-nullable）
        print(x * y)
    }
}
```

或者

```
    // ...
    if (x == null) {
        print("Wrong number format in '${args[0]}'")
        return
    }
    if (y == null) {
        print("Wrong number format in '${args[1]}'")
        return
    }

    // 在空检测后，x 和 y 会自动转换为非空值
    print(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 和 类型转换。

使用 `for` 循环

```
fun main(args: Array<String>) {
    for (arg in args) {
        print(arg)
    }
}
```

或者

```
for (i in args.indices) {
    print(args[i])
}
```

参见for循环。

Using a `while` loop

```
fun main(args: Array<String>) {
    var i = 0
    while (i < args.size) {
        print(args[i++])
    }
}
```

参见while 循环。

使用 `when` 表达式

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

```
if (x in 1..y-1) {
    print("OK")
}
```

检测某个数字是否在指定区间外:

```
if (x !in 0..array.lastIndex) {
    print("Out")
}
```

区间内迭代:

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

参见区间。

使用集合

对集合进行迭代:

```
for (name in names) {
    println(name)
}
```

使用 `in` 运算符来判断集合内是否包含某实例：

```
if (text in names) { // 会调用 names.contains(text)
    print("Yes")
}
```

使用 lambda 表达式来过滤（filter）和变换（map）集合：

```
names
        .filter { it.startsWith("A") }
        .sortedBy { it }
        .map { it.toUpperCase() }
        .forEach { print(it) }
```

参见高阶函数及Lambda表达式。

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 (参阅数据类)

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 }
```

String 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 data = ...
val email = data["email"] ?: throw IllegalStateException("Email is missing!")
```

if not null 执行代码

```
val data = ...

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

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 JsonSyntaxExc
eption {
//      ...

inline fun <reified T: Any> Gson.fromJson(json): 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

此页面包含当前 Kotlin 语言的编码风格

命名风格

如果拿不准的时候，默认使用Java的编码规范，比如：

- 使用驼峰法命名（并避免命名含有下划线）
- 类型名以大写字母开头
- 方法和属性以小写字母开头
- 使用 4 个空格缩进
- 公有函数应撰写函数文档，这样这些文档才会出现在 Kotlin Doc 中

冒号

类型和超类型之间的冒号前要有一个空格，而实例和类型之间的冒号前不要有空格：

```
interface Foo<out T : Any> : Bar {
    fun foo(a: Int): T
}
```

Lambda表达式

在lambda表达式中, 大括号左右要加空格，分隔参数与代码体的箭头左右也要加空格 。
lambda表达应尽可能不要写在圆括号中

```
list.filter { it > 10 }.map { element -> element * 2 }
```

在非嵌套的短lambda表达式中，最好使用约定俗成的默认参数 `it` 来替代显式声明参数名 。
在嵌套的有参数的lambda表达式中，参数应该总是显式声明。

Unit

如果函数返回 Unit 类型，该返回类型应该省略：

```
fun foo() { // 省略了 ": Unit"

}
```

函数还是属性

很多场合无参的函数可与只读属性互换。 尽管语义相近，也有一些取舍的风格约定。

底层算法优先使用属性而不是函数：

- 不会抛异常
- `O(1)` 复杂度
- 计算廉价（或缓存第一次运行）
- 不同调用返回相同结果

基础

基础内容

基本类型

在 Kotlin 中，所有东西都是对象，在这个意义上讲所以我们可以在任何变量上调用成员函数和属性。有些类型是内置的，因为他们的实现是优化过的。但是用户看起来他们就像普通的类。本节我们会描述大多数这些类型：数字、字符、布尔和数组。

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`

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。

注意数字装箱不会保留同一性:

```
val a: Int = 10000
print(a === a) // 输出 'true'
val boxedA: Int? = a
val anotherBoxedA: Int? = a
print(boxedA === anotherBoxedA) // !!!输出 'false'!!!
```

另一方面，它保留了相等性:

```
val a: Int = 10000
print(a == a) // 输出 'true'
val boxedA: Int? = a
val anotherBoxedA: Int? = a
print(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(a == b) // 惊！这将打印 "false" 鉴于 Long 的 equals() 检测其他部分也是 Long
```

所以同一性还有相等性都会在所有地方悄无声息地失去。

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
```

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`

缺乏隐式类型转换并不显著，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` 和 `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

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` 、 `\'` 、 `\"` 、 `\\` 和 `\$` 。 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`*** 和 ***`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` 和 `set` 函数（按照运算符重载约定这会转变为 `[]` ）和 `size` 属性，以及一些其他有用的成员函数：

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

    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、元素都为空的数组。

另一个选项是用接受数组大小和一个函数参数的工厂函数，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() })
```

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

注意: 与 Java 不同的是，Kotlin 中数组是不协变的（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。这些类和 `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]
```

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)
}
```

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，以及原生字符串可以包含换行和任意文本。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。

原生字符串 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和任何其他字符:

```
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
val s = "i = $i" // 求值结果为 "i = 10"
```

或者用花括号扩起来的任意表达式:

```
val s = "abc"
val str = "$s.length is ${s.length}" // 求值结果为 "abc.length is 3"
```

原生字符串和转义字符串内部都支持模板。 如果你需要在原生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。

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 将它的参数和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。

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和属性而无需 任何额外的检测。

```
val hasPrefix = 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")
}
```

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，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`。

对数组的 `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) {
    print(array[i])
}
```

注意这种“在区间上遍历”会编译成优化的实现而不会创建额外对象。

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 和 ***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和continue

在循环中 Kotlin 支持传统的 ***break*** 和 ***continue*** 操作符。参见返回和跳转。

返回和跳转

Kotlin 有三种结构化跳转操作符

- `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。

Break和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() {
    ints.forEach {
        if (it == 0) return
        print(it)
    }
}
```

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() {
    ints.forEach lit@ {
        if (it == 0) return@lit
        print(it)
    }
}
```

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() {
    ints.forEach {
        if (it == 0) return@forEach
        print(it)
    }
}
```

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() {
    ints.forEach(fun(value: Int) {
        if (value == 0) return
        print(value)
    })
}
```

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)` ”。

类和对象

所有对象相关的

- 类和继承
- 属性和字段
- 接口
- 可见性修饰符
- 扩展
- 数据类
- 泛型
- 嵌套类
- 枚举类
- 对象
- 委托
- 委托属性

类和继承

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）和由大括号包围的类体构成。类头和类体都是可选的； 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和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（和可选的类型参数）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：

```
class Customer(name: String) {
    init {
        logger.info("Customer initialized with value ${name}")
    }
}
```

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)
    }
}
```

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 = "")
> ```
>
> {:.info}

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、内部类和匿名内部类的类实例在嵌套类中有述。

类成员

类可以包含

- 构造函数和初始化块
- 函数
- 属性
- 嵌套类和内部类
- 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
```

`Any` 不是 `java.lang.Object`；尤其是，它除了 `equals()`、`hashCode()`和 `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)
```

如果该类有一个主构造函数，其基类型可以（并且必须）用（基类型的）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)
}
```

类上的 ***open*** 标注与 Java 中 ***final*** 相反，它允许其他类从这个类继承。默认情况下，在 Kotlin 中所有的类都是 final，对应于 Effective Java书中的第 17 条：要么为继承而设计，并提供文档说明，要么就禁止继承。

覆盖成员

我们之前提到过，Kotlin 力求清晰显式。与 Java 不同，Kotlin 需要显式标注可覆盖的成员（我们称之为开放）和覆盖后的成员：

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

Derived.v() 函数上必须加上 **override**标注。如果没写，编译器将会报错。如果函数没有标注 **open** 如 `Base.nv()`，则子类中不允许定义相同签名的函数，不论加不加 **override**。在一个 **final** 类中（没有用 **open** 标注的类），开放成员是禁止的。

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() {}
}
```

属性覆盖与方法覆盖类似。 注：你可以用 `override` 关键字作为主构造函数属性声明的一部分：

```
open class Foo {
    open val x: Int get { ... }
}

class Bar1(override val x: Int) : Foo() {

}
```

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。

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` 和 `B` 没问题，并且 `a()` 和 `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

类和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。

密封类

密封类用来表示受限的类层次结构：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，但是所以子类声明都必须嵌套在这个密封类声明内部。

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

值得注意的是一个密封类的子类的继承者（间接继承）可以在任何地方声明，不一定要在 这个密封类声明内部。

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。

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

属性和字段

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*** 。

```
public class Address {
    public var name: String = ...
    public var street: String = ...
    public var city: String = ...
    public var state: String? = ...
    public 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 和 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基类中推断出来，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
```

我们可以编写自定义的访问器，非常像普通函数，刚好在属性声明内部。这里有一个自定义 getter 的例子:

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

一个自定义的 setter 的例子:

```
var stringRepresentation: String
    get() = this.toString()
    set(value) {
        setDataFromString(value) // 解析字符串并赋值给其他属性
    }
```

按照惯例，setter 参数的名称是 `value` ，但是如果你喜欢你可以选择一个不同的名称。

如果你需要改变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

Kotlin 中类不能有字段。然而，当使用自定义访问器时，有时有一个幕后字段（backing field）有时是必要的。为此 Kotlin 提供 一个自动幕后字段，它可通过使用 `field` 标识符访问。

```
var counter = 0 // 此初始器值直接写入到幕后字段
  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` 的一个成员
- 用 `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() { ... }
```

惰性初始化属性

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（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。该属性必须是非空类型，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。

覆盖属性

参见 覆盖成员

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 property: Int // 抽象的

    val propertyWithImplementation: String
        get() = "foo"

    fun foo() {
        print(property)
    }
}

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

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()
    }
}
```

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。*D* 可以不用重写 *bar()*，因为它实现了 *A* 和 *B*，因而可以自动继承 *B* 中 *bar()* 的实现， 但是两个接口都定义了方法 *foo()*，为了告诉编译器 *D* 会继承谁的方法，必须在 *D* 中重写 *foo()*。

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`。

下面将根据声明范围的不同来解释。

包名

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。

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

当一个类内部声明：

- `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 或者 Gradle 项目；
- 一次 <kotlinc> Ant 任务执行所编译的一套文件。

扩展

Kotlin 同 C# 和 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 和 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的实例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和一个扩展函数，而这两个函数又有相同的接收者类型、相同的名字 并且都适用给定的参数，这种情况总是取成员函数。 例如：

```
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

和函数类似，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 扩展函数和属性：

```
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()
```

扩展的范围

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 // 以名字 "goo" 导入所有扩展
                   // 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
    }
}
```

对于分发接收者和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")
    }
}

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
```

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)), Col
lections.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）。

如果这些函数中的任何一个在类体中显式定义或继承自其基类型，则不会生成该函数。

为了确保生成的代码的一致性和有意义的行为，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；
- 数据类不能扩展其他类（但可以实现接口）。

> 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)
> ```

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)
```

数据类和解构声明

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` 和 `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。

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 解释了该问题——第28条：利用有限制通配符来提升 *API* 的灵活性。首先，Java 中的泛型是不协变的，这意味着 `List<String>` 并不是 `List<Object>` 的子类型。为什么这样？如果 List 不是不协变的，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25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 T` 表示此方法接受 `T` 的一些子类型对象的集合，而不是 `T` 本身。这意味着我们可以安全地从其中（该集合中的元素是 T 的子类的实例）读取 `T`，但不能写入，因为我们不知道什么对象符合那个未知的 `T` 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：

```
abstract class Source<out T> {
    abstract 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。逆变类的一个很好的例子是 `Comparable` ：

```
abstract class Comparable<in T> {
    abstract 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

声明一个类型参数 T 为 *out* 非常方便，并且在使用处运用子类型也没有问题。是的，当问题中的类能够仅限于返回 `T` 时，但是如果它不能呢？ 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) // 错误：期望 (Array<Any>, Array<Any>)
```

这里我们遇到同样熟悉的问题：`Array <T>` 在 `T` 上是不协变的，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>`，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>`，其中 `T` 是一个具有上界 `TUpper` 的不协变类型参数，`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)
```

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> cloneWhenGreater(list: List<T>, threshold: T): List<T>
    where T : Comparable,
          T : Cloneable {
  return list.filter { it > threshold }.map { it.clone() }
}
```

嵌套类

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。

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。

每个枚举常量都具有在枚举类声明中获取其名称和位置的属性：

```
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。

对象表达式和对象声明

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：

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

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

单例模式是一种非常有用的模式，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。

要引用该对象，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
```

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()
    }
}
```

当然，在 JVM 平台，如果使用 `@JvmStatic` 注解，你可以将伴生对象的成员生成为真正的静态方法和字段。更详细信息请参见Java 互操作性一节 。

对象表达式和对象声明之间的语义差异

对象表达式和对象声明之间有一个重要的语义差别：

- 对象表达式是在使用他们的地方立即执行（及初始化）的
- 对象声明是在第一次被访问到时延迟初始化的
- 伴生对象的初始化是在相应的类被加载（解析）时，与 Java 静态初始化器的语义相匹配

委托

类委托

委托模式已经证明是实现继承的一个很好的替代方式， 而 Kotlin 可以零样板代码地原生支持它。 类 `Derived` 可以继承一个接口 `Base` ，并将其所有共有的方法委托给一个指定的对象：

```
interface Base {
    fun print()
}

class BaseImpl(val x: Int) : Base {
    override fun print() { print(x) }
}

class Derived(b: Base) : Base by b

fun main(args: Array<String>) {
    val b = BaseImpl(10)
    Derived(b).print() // 输出 10
}
```

`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。

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()` （和 `set()` ）会被委托给它的 `getValue()` 和 `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（和 `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.
```

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：

- 接收者 —— 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，
- 元数据 —— 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：

- 接收者 —— 同 `getValue()`，
- 元数据 —— 同 `getValue()`，
- 新的值 —— 必须和属性同类型或者是它的超类型。

`getValue()` 或/和 `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。

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 lamda 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(args: Array<String>) {
    println(lazyValue)
    println(lazyValue)
}
```

This example prints:

```
computed!
Hello
Hello
```

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和相关的开销。

可观察属性 Observable

`Delegates.observable()` 接受两个参数：初始值和修改时处理程序（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、旧值和新值：

```
import kotlin.properties.Delegates

class User {
    var name: String by Delegates.observable("<no name>") {
        prop, old, new ->
        println("$old -> $new")
    }
}

fun main(args: Array<String>) {
    val user = User()
    user.name = "first"
    user.name = "second"
}
```

这个例子输出：

```
<no name> -> first
first -> 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
}
```

函数和 Lambda 表达式

所有函数相关的

- 函数
- Lambda 表达式
- 内联函数

函数

函数声明

Kotlin 中的函数使用 `fun` 关键字声明

```
fun double(x: Int): Int {
}
```

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
```

中缀表示法

函数还可以用中缀表示法调用，当

- 他们是成员函数或扩展函数
- 他们只有一个参数
- 他们用 `infix` 关键字标注

```
// 给 Int 定义扩展
infix fun Int.shl(x: Int): Int {
……
}

// 用中缀表示法调用扩展函数

1 shl 2

// 等同于这样

1.shl(2)
```

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
}
```

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 = '_')
```

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)
```

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。

```
tailrec fun findFixPoint(x: Double = 1.0): Double
        = if (x == Math.cos(x)) 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，产生0.7390851332151607的结果。最终代码相当于这种更传统风格的代码：

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

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。

高阶函数和 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。 这种函数的一个很好的例子是 `lock()` ，它接受一个锁对象和一个函数，获取锁，运行函数并释放锁：

```
fun <T> lock(lock: Lock, body: () -> T): T {
    lock.lock()
    try {
        return body()
    }
    finally {
        lock.unlock()
    }
}
```

让我们来检查上面的代码： `body` 拥有函数类型： `() -> T` ， 所以它应该是一个不带参数并且返回 `T` 类型值的函数。 它在 ***try*** -代码块内部调用、被 `lock` 保护，其结果由 `lock()` 函数返回。

如果我们想调用 `lock()` 函数，我们可以把另一个函数传给它作为参数（参见函数引用）：

```
fun toBeSynchronized() = sharedResource.operation()

val result = lock(lock, ::toBeSynchronized)
```

通常会更方便的另一种方式是传一个 lambda 表达式：

```
val result = lock(lock, { sharedResource.operation() })
```

Lambda 表达式在下文会有更详细的描述，但为了继续这一段，让我们看一个简短的概述：

- lambda 表达式总是被大括号括着，
- 其参数（如果有的话）在 `->` 之前声明（参数类型可以省略），
- 函数体（如果存在的话）在 `->` 后面。

在 Kotlin 中有一个约定，如果函数的最后一个参数是一个函数，那么该参数可以在圆括号之外指定：

```
lock (lock) {
    sharedResource.operation()
}
```

高阶函数的另一个例子是 `map()` ：

```
fun <T, R> List<T>.map(transform: (T) -> R): List<R> {
    val result = arrayListOf<R>()
    for (item in this)
        result.add(transform(item))
    return result
}
```

该函数可以如下调用:

```
val doubled = ints.map { it -> it * 2 }
```

请注意，如果 lambda 是该调用的唯一参数，则调用中的圆括号可以完全省略。

`it` ：单个参数的隐式名称

另一个有用的约定是，如果函数字面值只有一个参数， 那么它的声明可以省略（连同 `->`），其名称是 `it` 。

```
ints.map { it * 2 }
```

这些约定可以写LINQ-风格的代码:

```
strings.filter { it.length == 5 }.sortBy { it }.map { it.toUpperCase() }
```

内联函数

使用内联函数有时能提高高阶函数的性能。

Lambda 表达式和匿名函数

一个 lambda 表达式或匿名函数是一个“函数字面值”，即一个未声明的函数， 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，换句话说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。写成函数的话，它相当于

```
fun compare(a: String, b: String): Boolean = a.length < b.length
```

函数类型

对于接受另一个函数作为参数的函数，我们必须为该参数指定函数类型。 例如上述函数 `max` 定义如下：

```
fun <T> max(collection: Collection<T>, less: (T, T) -> Boolean): T? {
    var max: T? = null
    for (it in collection)
        if (max == null || less(max, it))
            max = it
    return max
}
```

参数 `less` 的类型是 `(T, T) -> Boolean` ，即一个接受两个类型 `T` 的参数并返回一个布尔值的函数： 如果第一个参数小于第二个那么该函数返回 true。

在上面第 4 行代码中， `less` 作为一个函数使用：通过传入两个 `T` 类型的参数来调用。

如上所写的是就函数类型，或者可以有命名参数，如果你想文档化每个参数的含义的话。

```
val compare: (x: T, y: T) -> Int = ...
```

Lambda 表达式语法

Lambda 表达式的完整语法形式，即函数类型的字面值如下：

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

lambda 表达式总是被大括号括着， 完整语法形式的参数声明放在括号内，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 }
```

一个 lambda 表达式只有一个参数是很常见的。 如果 Kotlin 可以自己计算出签名，它允许我们不声明唯一的参数，并且将隐含地 为我们声明其名称为 `it` ：

```
ints.filter { it > 0 } // 这个字面值是“(it: Int) -> Boolean”类型的
```

请注意，如果一个函数接受另一个函数作为最后一个参数，lambda 表达式参数可以在 圆括号参数列表之外传递。 参见 callSuffix 的语法。

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表达式和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)
```

带接收者的函数字面值

Kotlin 提供了使用指定的 接收者对象 调用函数字面值的功能。在函数字面值的函数体中，可以调用该接收者对象上的方法而无需任何额外的限定符。这类似于扩展函数，它允你在函数体内访问接收者对象的成员。其用法的最重要的示例之一是类型安全的 Groovy-风格构建器。

这样的函数字面值的类型是一个带有接收者的函数类型：

```
sum : Int.(other: Int) -> Int
```

该函数字面值可以这样调用，就像它是接收者对象上的一个方法一样：

```
1.sum(2)
```

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。

```
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 lock<T>(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，但是如果我们使用得当（不内联大函数），它将在 性能上有所提升，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：

```
fun foo() {
    inlineFunction {
        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：

```
myTree.findParentOfType(MyTreeNodeType::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：

```
myTree.findParentOfType<MyTreeNodeType>()
```

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。

其他

各种知识点

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().itera
tor()
operator fun <K, V> Map.Entry<K, V>.component1() = getKey()
operator fun <K, V> Map.Entry<K, V>.component2() = getValue()
```

因此你可以在 ***for***-循环中对映射（以及数据类实例的集合等）自由使用解构声明。

集合

与大多数语言不同，Kotlin 区分可变集合和不可变集合（lists、sets、maps 等）。精确控制什么时候集合可编辑有助于消除 bug 和设计良好的 API。

预先了解一个可变集合的只读 视图 和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。和 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()` 。 创建 map 在非性能关键代码中可以用一个简单的惯用法： `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。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。

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 和 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。

Map 遵循同样模式。它们可以容易地实例化和访问，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)
```

范围

范围表达式是由“rangeTo”函数组成的，操作符的形式是 `..` 由 ***in*** 和 ***!in*** 补充。 范围被定义为任何可比类型,但是用于原生类型有更优化的实现。下面是使用范围的例子

```
if (i in 1..10) { // equivalent of 1 <= i && i <= 10
    println(i)
}
```

整型范围（ `IntRange` , `LongRange` , `CharRange` ）有一个额外的功能:他们可以遍历。 编译者需要关心的转换是简单模拟Java的索引 ***for*** 循环,不用担心越界。例如：

```
for (i in 1..4) print(i) // prints "1234"

for (i in 4..1) print(i) // prints nothing
```

你想要遍历数字颠倒顺序吗?这很简单。您可以使用标准库里面的 `downTo()` 函数

```
for (i in 4 downTo 1) print(i) // prints "4321"
```

是否可以任意进行数量的迭代,而不必每次的变化都是1呢?当然， `step()` 函数可以实现

```
for (i in 1..4 step 2) print(i) // prints "13"

for (i in 4 downTo 1 step 2) print(i) // prints "42"
```

To create a range which does not include its end element, you can use the `until` function:

```
for (i in 1 until 10) { // i in [1, 10), 10 is excluded
     println(i)
}
```

它是如何工作的

Ranges implement a common interface in the library: `ClosedRange<T>` .

`ClosedRange<T>` 在数学意义上表示一个间隔,是对比较类型的定义。 它有两个端点:‘start’和‘endInclusive’,这是包含在范围内。 主要的操作是 `contains` ,通常用 ***in*** */!in* {: .keyword }操作符内。

Integral type progressions (`IntProgression` , `LongProgression` , `CharProgression`) denote an arithmetic progression. Progressions are defined by the `first` element, the `last` element and a non-zero `increment` . The first element is `first` , subsequent elements are the previous element plus `increment` . The `last` element is always hit by iteration unless the progression is empty.

A progression is a subtype of `Iterable<N>` , where `N` is `Int` , `Long` or `Char` respectively, so it can be used in `for` -loops and functions like `map` , `filter` , etc. 迭代 `Progression` 与 Java/JavaScript的基于索引的 `for` 循环等价:

```
for (int i = first; i != last; i += increment) {
  // ...
}
```

对于整型, `..` 操作符创建一个对象既实现了 `ClosedRange` 也实现了 `Progression` 。 For example, `IntRange` implements `ClosedRange<Int>` and extends `IntProgression` , thus all operations defined for `IntProgression` are available for `IntRange` as well. `downTo()` 和 `step()` 函数的结果一直是 `Progression` 。

Progressions are constructed with the `fromClosedRange` function defined in their companion objects:

```
    IntProgression.fromClosedRange(start, end, increment)
```

The `last` element of the progression is calculated to find maximum value not greater than the `end` value for positive `increment` or minimum value not less than the `end` value for negative `increment` such that `(last - first) % increment == 0` .

一些实用函数

rangeTo()

整型得 `rangeTo()` 运算符只要简单地调用构造函数 `*Range` 类,例如:

```
class Int {
    //...
    operator fun rangeTo(other: Long): LongRange = LongRange(this, other)
    //...
    operator fun rangeTo(other: Int): IntRange = IntRange(this, other)
    //...
}
```

Floating point numbers (`Double` , `Float`) do not define their `rangeTo` operator, and the one provided by the standard library for generic `Comparable` types is used instead:

```
    public operator fun <T: Comparable<T>> T.rangeTo(that: T): ClosedRange<T>
```

The range returned by this function cannot be used for iteration.

downTo()

`downTo()` 的扩展函数可以为任何数字整型对定义,这里有两个例子:

```
fun Long.downTo(other: Int): LongProgression {
    return LongProgression.fromClosedRange(this, other, -1.0)
}

fun Byte.downTo(other: Int): IntProgression {
    return IntProgression.fromClosedRange(this, other, -1)
}
```

reversed()

定义 `reversed()` 扩展函数是为了每个 `*Progression` 类定义的,它们返回反向的级数。

```
fun IntProgression.reversed(): IntProgression {
    return IntProgression.fromClosedRange(last, first, -increment)
}
```

step()

`step()` 扩展函数是为每个 `*Progression` 类定义的, 他们返回级数与都修改了 `step` 值(函数参数)。 注意,step值必需总是正的，因此这个函数从不改变的迭代方向。

```
fun IntProgression.step(step: Int): IntProgression {
    if (step <= 0) throw IllegalArgumentException("Step must be positive, was: $step")
    return IntProgression.fromClosedRange(first, last, if (increment > 0) step else -s
tep)
}

fun CharProgression.step(step: Int): CharProgression {
    if (step <= 0) throw IllegalArgumentException("Step must be positive, was: $step")
    return CharProgression.fromClosedRange(first, last, step)
}
```

Note that the `last` value of the returned progression may become different from the `last` value of the original progression in order to preserve the invariant `(last - first) % increment == 0`. Here is an example:

```
(1..12 step 2).last == 11  // progression with values [1, 3, 5, 7, 9, 11]
(1..12 step 3).last == 10  // progression with values [1, 4, 7, 10]
(1..12 step 4).last == 9   // progression with values [1, 5, 9]
```

类型的检查与转换

`is` 和 `!is` 运算符

我们可以使用 `is` 或者它的否定 `!is` 运算符检查一个对象在运行中是否符合所给出的类型：

```
if (obj is String) {
    print(obj.length)
}

if (obj !is String) { // same as !(obj is String)
    print("Not a String")
}
else {
    print(obj.length)
}
```

智能转换

在很多情况下，在Kotlin有时不用使用明确的转换运算符，因为编译器会在需要的时候自动为了不变的值和输入（安全）而使用 `is` 进行监测：

```
fun demo(x: Any) {
    if (x is String) {
        print(x.length) // x is automatically cast to String
    }
}
```

如果错误的检查导致返回，编译器会清楚地转换为一个正确的：

```
    if (x !is String) return
    print(x.length) // x is automatically cast to String
```

或者在右边是 `&&` 和 `||` ：

```
    // x is automatically cast to string on the right-hand side of `||`
    if (x !is String || x.length == 0) return

    // x is automatically cast to string on the right-hand side of `&&`
    if (x is String && x.length > 0) {
        print(x.length) // x is automatically cast to String
    }
```

这些智能转换在 `when` -expressions 和 `while` -loops 也一样：

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

Note that smart casts do not work when the compiler cannot guarantee that the variable cannot change between the check and the usage. More specifically, smart casts are applicable according to the following rules:

- `val` local variables - always;
- `val` properties - if the property is private or internal or the check is performed in the same module where the property is declared. Smart casts aren't applicable to open properties or properties that have custom getters;
- `var` local variables - if the variable is not modified between the check and the usage and is not captured in a lambda that modifies it;
- `var` properties - never (because the variable can be modified at any time by other code).

“不安全”的转换运算符

通常，如果转换是不可能的，转换运算符会抛出一个异常。于是，我们称之为不安全的。在Kotlin这种不安全的转换会出现在插入运算符 `as` (see operator precedence)：

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

记住 `null` 不能被转换为不可为空的 `String` 。例如，如果 `y` 是空，则这段代码会抛出异常。为了匹配Jave的转换语义，我们不得不在右边拥有可空的类型，就像：

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

“安全的”（可为空的）转换运算符

为了避免异常的抛出，一个可以使用安全的转换运算符——`as?`，它可以在失败时返回一个`null`：

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

记住尽管事实是右边的`as?`可使一个不为空的`String`类型的转换结果为可空的。

This表达式

为了记录下当前的接收者我们使用 `this` 表达式:

- 在一个类成员中, `this` 指的是当前类对象。
- 在一个扩展函数或者带有接收者字面函数, `this` 表示左边的接收者.

如果 `this` 没有应用者，则指向的是最内层的闭合范围。为了在其它范围中返回 this ，需要使用标签：

`this` 限定

{:#限定}

为了在范围外部访问 `this` (一个类, 或者扩展函数, 或者带标签的带接收者的字面函数 我们使用 `this@label` 作为label： on the scope `this` is meant to be from:

```
class A { // implicit label @A
    inner class B { // implicit label @B
        fun Int.foo() { // implicit label @foo
            val a = this@A // A's this
            val b = this@B // B's this

            val c = this // foo()'s receiver, an Int
            val c1 = this@foo // foo()'s receiver, an Int

            val funLit = lambda@ fun String.() {
                val d = this // funLit's receiver
            }


            val funLit2 = { s: String ->
                // foo()'s receiver, since enclosing lambda expression
                // doesn't have any receiver
                val d1 = this
            }
        }
    }
}
```

相等性

Kotlin中有两种类型的相等性:

- 引用相等(两个引用指向相同的对象)
- 结构相等 (`equals()`)

引用相等

引用相等使用 `===` 操作符判断(它的否定是 `!==`). `a === b` 只有当 `a` 和 `b` 指向同一个对象才返回true。

结构相等

结构相等使用 `==` 操作符判断(它的否定是 `!=`). 通常, `a == b` 表达式被翻译为：

```
a?.equals(b) ?: (b === null)
```

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 的是不是 `null` 引用。

注意当与 `null` 比较时完全没有必要为优化你的代码而将 `a == null` 写成 `a === null` 编译器会自动帮你做的。

操作符重载

Kotlin允许我们实现一些我们自定义类型的运算符实现。这些操作符有固定的表示（像 `+` 或者 `*`），和固定的优先级。为实现这样的操作符，我们提供了固定名字的成员函数 或扩展函数，比如二元操作符的左值和一元操作符的参数类型。 Functions that overload operators need to be marked with the `operator` modifier.

转换

这里我们描述了一些常用操作符的重载。

一元操作符

表达式	翻译为
`+a`	`a.unaryPlus()`
`-a`	`a.unaryMinus()`
`!a`	`a.not()`

这张表解释了当编译器运行时，比如，表达式 `+a` ，是这样运行的：

- 决定 `a` 的类型, 假设为 `T` 。
- 寻找接收者是 `T` 带有 `operator` 修饰的无参函数 `unaryPlus()` ,例如一个成员方法或者扩展方法。
- 如果找不到或者不明确就返回一个错误。
- 如果函数是当前函数或返回类型是 `R` 则表达式 `+a` 是 `R` 类型。

注意 这些操作符和其它的一样, 都被优化为基本类型并且不会产生多余的开销。

表达式	翻译为
`a++`	`a.inc()` + 见下方
`a--`	`a.dec()` + 见下方

这些操作符允许修改接收者和返回类型。

> `inc()/dec()` 不应该改变接收对象.
> "修改接收者"你应该修改接收者变量而非对象。 {:.note}

编译器是这样解决有后缀的操作符的比如 `a++`：

- 决定 `a` 的类型, 假设为 `T` 。

- 查找接收类型为 `T` 带有 `operator` 修饰的无参数函数`inc()。
- 如果返回类型为 `R` ,那么 `R` 为 `T` 子类型.

计算表达式的步骤是：

- 把 `a` 的值存在 `a0` 中,
- 把 `a.inc()` 结果作用于 `a` ,
- 把 `a0` 作为表达式的结果.

a-- 的运算步骤也是一样的。

对于前缀操作符 `++a` 和 `--a` 的解决方式也是一样的, 步骤是:

- 把 `a.inc()` 作用于 `a` ,
- 返回新值 `a` 作为表达式结果。

二元操作符

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.mod(b)`
`a..b`	`a.rangeTo(b)`

编译器只是解决了该表中翻译为列的表达式。

Expression	Translated to
`a in b`	`b.contains(a)`
`a !in b`	`!b.contains(a)`

in 和 !in 的产生步骤是一样的，但参数顺序是相反的。 {:#in}

符号	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)`

方括号被转换为 get set 函数。

符号	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)`

括号被转换为带有正确参数的 `invoke` 函数。

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.modAssign(b)`

{:#assignments}

在分配 a+= b时编译器是下面这样实现的:

- 右边函数是否可用。
  - 对应的二元函数是否 (如 `plus()` 和 `plusAssign()`)也可用, 不可用就报告错误.
  - 确定它的返回值是 `Unit` 类型, 否则报告错误。
  - 生成 `a.plusAssign(b)` *否则试着生成 `a = a + b` 代码 (这里包含类型检查: `a + b` 一定要是 `a` 的子类型).

注意: assignments在Kotlin中不是表达式. {:#Equals}

表达式	翻译为
`a == b`	`a?.equals(b) ?: (b === null)`
`a != b`	`!(a?.equals(b) ?: (b === null))`

注意: `===` 和 `!==` (实例检查)不能重载, 所以没有转换方式。

The `==` 操作符有点特殊: 它被翻译成一个复杂的表达式，用于筛选 `null` 值，而且 `null == null` 是 `true` 。

符号	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 `compareTo` 的调用，这个函数需要返回 `Int` 值

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（Nullable）和不可空（Non-Null） 类型

Kotlin 的类型系统致力于消除空引用异常的危险，又称《上亿美元的错误》。

许多编程语言，包括 Java 中最常见的错误就是访问空引用的成员变量，导致空引用异常。在 Java 中，将等同于 `NullPointerException` 或简称 `NPE` 。

Kotlin 类型系统的目的就是从我们的代码中消除 `NullPointerException` 。 `NPE` 的原因可能是

- 显式调用 `throw NullPointerException()`
- Usage of the `!!` operator that is described below
- 外部 Java 代码引起
- 对于初始化，有一些数据不一致 (比如一个还没初始化的 `this` 用于构造函数的某个地方)

在 Kotlin 中，类型系统是要区分一个引用是否可以是 ***`null`*** （nullable references）或者不可以，即 不可空引用（non-null references）。例如，常见的 `String` 就不能够为 ***`null`*** ：

```
var a: String = "abc"
a = null // 编译错误
```

若是想要允许 `null` ，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
```

现在，如果你调用/访问一个 `a` 方法/属性（译者注： `a` 是一个不可空类型）的一个方法，它保证不会造成 `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` 的相同属性，这将是不安全的，同时编译器会报错：

```
val l = b.length // 错误：变量 b 可能为 null
```

可是我仍然需要访问这些属性，对吧？这里有一些方式可以这么做：

使用条件语句检测是否为 `null`

首先，你可以明确地检查 `b` 是否为 `null`，并分别处理两种选择：

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

编译器会跟踪所执行的检查信息，然后允许你在 `if` 中调用 `length`
同时，也支持更复杂（更智能）的条件：

```
if (b != null && b.length > 0) {
    print("String of length ${b.length}")
} else {
    print("Empty string")
}
```

需要注意的是这仅适用其中 `b` 是不可变的（i.e. a local variable which is not modified between the check and the usage or a member `val` which has a backing field and is not overridable）情况，否则在 检查之后它可能它为空导致异常。

安全的调用

你的第二个选择是安全的操作符，写作 `?.` ：

```
b?.length
```

如果 `b` 是非空的，就会返回 `b.length` ，否则返回 `null`，这个表达式的类型就是 `Int?`.

安全调用在链式调用的时候十分有用。例如，如果Bob，一个雇员，可被分配给一个部门（或不），这反过来又可以获得 Bob 的部门负责人的名字（如果有的话），我们这么写：

```
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`。

To perform a certain operation only for non-null values, you can use the safe call operator together with `let`:

```
val listWithNulls: List<String?> = listOf("A", null)
for (item in listWithNulls) {
     item?.let { println(it) } // prints A and ignores null
}
```

Elvis 操作符

当我们有一个可以为空的变量 `r`，我们可以说「如果 `r` 非空，我们使用它；否则使用某个非空的值：

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

对于完整的 ***if*** -表达式, 可以换成 Elvis 操作符来表达, 写作 `?:`：

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

如果 `?:` 的左边表达式是非空的， elvis 操作符就会返回左边的结果, 否则返回右边的内容。请注意，仅在左侧为空的时候，右侧表达式才会进行计算。

注意, 因为 ***throw*** 和 ***return*** 在 Kotlin 中都是一种表达式，它们也可以用在 Elvis 操作符的右边。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

第三种操作的方式是给 NPE 爱好者的。我们可以写 `b!!` ，这样就会返回一个不可空的 `b` 的值（例如：在我们例子中的 `String`）或者如果 `b` 是空的，就会抛出 `NPE` 异常：

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

因此，如果你想要一个 `NPE` ，你可以使用它。but you have to ask for it explicitly, and it does not appear out of the blue.

安全转型

转型的时候，可能会经常出现 `ClassCastException` 。 所以，现在可以使用安全转型，当转型不成功的时候，它会返回 ***null*** ：

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

Collections of Nullable Type

If you have a collection of elements of a nullable type and want to filter non-null elements, you can do so by using `filterNotNull`.

```
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` 的子类。 每一个异常都含有一条信息、栈回溯信息和一个可选的原因。

可以使用 ***`throw`*** -expression 来抛出一个异常。

```
throw MyException("Hi There!")
```

使用 `*try*{: .keyword }-expression` 来捕获一个异常。

```
try {
    // some code
}
catch (e: SomeException) {
    // handler
}
finally {
    // optional finally block
}
```

可以有零或多个 ***`catch`*** 块。 ***`finally`*** 块可以省略。 ***`catch`*** 和 ***`finally`*** 块应该至少出现一个。

Try是一个表达式

`try` 是一个表达式，比如，它可以有一个返回值。

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

`try` -expression 的返回值是 ***`try`*** 中 最后一个表达式或者是 ***`catch`*** 块中最后一个表达式。 ***`finally`*** 块中的内容不会影响到表达式的结果。

受检的异常

Kotlin中没有受检的异常。这是有很多原因的，但是我们会提供一个简单的示例。

以下示例是JDK中 `StringBuilder` 类实现中的

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

What does this signature say? It says that every time I append a string to something (a `StringBuilder` , some kind of a log, a console, etc.) I have to catch those `IOExceptions` . Why? Because it might be performing IO (`Writer` also implements `Appendable`)... So it results into this kind of code all over the place:

```
try {
    log.append(message)
}
catch (IOException e) {
    // Must be safe
}
```

这样做是没有好处的，参阅 Effective Java,Item 65: 不要忽略异常。

Bruce Eckel在Does Java need Checked Exceptions? 中指出:

> 通过一些小程序测试得出的结论是规范的异常会提高开发者的生产效率和提高代码质量，但是大型软件项目的经验告诉我们一个不同的结论 - 这会降低生产效率并且不会对代码质量有明显提高。

相关引用：

- Java's checked exceptions were a mistake (Rod Waldhoff)
- The Trouble with Checked Exceptions (Anders Hejlsberg)

Java 互操作性

请在 Java Interoperability section 异常章节中参阅Java交互相关信息。

注解

注解的声明

注解是连接元数据以及代码的。为了声明注解，把 ***annotation*** 这个关键字放在类前面：

```
annotation class Fancy
```

Additional attributes of the annotation can be specified by annotating the annotation class with meta-annotations:

- `@Target` specifies the possible kinds of elements which can be annotated with the annotation (classes, functions, properties, expressions etc.);
- `@Retention` specifies whether the annotation is stored in the compiled class files and whether it's visible through reflection at runtime (by default, both are true);
- `@Repeatable` allows using the same annotation on a single element multiple times;
- `@MustBeDocumented` specifies that the annotation is part of the public API and should be included in the class or method signature shown in the generated API documentation.

```
@Target(AnnotationTarget.CLASS, AnnotationTarget.FUNCTION,
        AnnotationTarget.VALUE_PARAMETER, AnnotationTarget.EXPRESSION)
@Retention(AnnotationRetention.SOURCE)
@MustBeDocumented
annotation class Fancy
```

用途

```
@Fancy class Foo {
    @Fancy fun baz(@Fancy foo: Int): Int {
        return (@Fancy 1)
    }
}
```

如果你需要注解类的主构造方法，你需要给构造方法的声明添加 ***constructor*** 这个关键字，还有在前面添加注解：

```
class Foo @Inject constructor(dependency: MyDependency) {
    // ...
}
```

你也可以注解属性访问器：

```
class Foo {
    var x: MyDependency? = null
        @Inject set
}
```

构造方法

注解可以有参数的构造方法。

```
annotation class Special(val why: String)

@Special("example") class Foo {}
```

Allowed parameter types are:

- types that correspond to Java primitive types (Int, Long etc.);
- strings;
- classes (`Foo::class`);
- enums;
- other annotations;
- arrays of the types listed above.

If an annotation is used as a parameter of another annotation, its name is not prefixed with the @ character:

```
annotation class ReplaceWith(val expression: String)

annotation class Deprecated(
        val message: String,
        val replaceWith: ReplaceWith = ReplaceWith(""))

@Deprecated("This function is deprecated, use === instead", ReplaceWith("this === othe
r"))
```

If you need to specify a class as an argument of an annotation, use a Kotlin class (KClass). The Kotlin compiler will automatically convert it to a Java class, so that the Java code will be able to see the annotations and arguments normally.

```
import kotlin.reflect.KClass

annotation class Ann(val arg1: KClass<*>, val arg2: KClass<out Any?>)

@Ann(String::class, Int::class) class MyClass
```

Lambdas

注解也可以用在lambda表达式中。这将会应用到 lambda 生成的 `invoke()` 方法 。这对 Quasar框架很有用， 在这个框架中注解被用来并发控制

```
annotation class Suspendable

val f = @Suspendable { Fiber.sleep(10) }
```

Annotation Use-site Targets

When you're annotating a property or a primary constructor parameter, there are multiple Java elements which are generated from the corresponding Kotlin element, and therefore multiple possible locations for the annotation in the generated Java bytecode. To specify how exactly the annotation should be generated, use the following syntax:

```
class Example(@field:Ann val foo,    // annotate Java field
              @get:Ann val bar,      // annotate Java getter
              @param:Ann val quux)   // annotate Java constructor parameter
```

The same syntax can be used to annotate the entire file. To do this, put an annotation with the target `file` at the top level of a file, before the package directive or before all imports if the file is in the default package:

```
@file:JvmName("Foo")

package org.jetbrains.demo
```

If you have multiple annotations with the same target, you can avoid repeating the target by adding brackets after the target and putting all the annotations inside the brackets:

```
class Example {
     @set:[Inject VisibleForTesting]
     var collaborator: Collaborator
}
```

The full list of supported use-site targets is:

- `file`
- `property` (annotations with this target are not visible to Java)
- `field`
- `get` (property getter)
- `set` (property setter)
- `receiver` (receiver parameter of an extension function or property)
- `param` (constructor parameter)
- `setparam` (property setter parameter)
- `delegate` (the field storing the delegate instance for a delegated property)

To annotate the receiver parameter of an extension function, use the following syntax:

```
fun @receiver:Fancy String.myExtension() { }
```

If you don't specify a use-site target, the target is chosen according to the `@Target` annotation of the annotation being used. If there are multiple applicable targets, the first applicable target from the following list is used:

- `param`
- `property`
- `field`

Jave注解

Java注解是百分百适用于Kotlin：

```
import org.junit.Test
import org.junit.Assert.*
import org.junit.Rule
import org.junit.rules.*

class Tests {
    // apply @Rule annotation to property getter
    @get:Rule val tempFolder = TemporaryFolder()

    @Test fun simple() {
        val f = tempFolder.newFile()
        assertEquals(42, getTheAnswer())
    }
}
```

因为在Java里，注释的参数顺序不是明确的，你不能使用常规的方法 调用语法传递的参数。相反的，你需要使用指定的参数语法。

```
// Java
public @interface Ann {
    int intValue();
    String stringValue();
}
```

```
// Kotlin
@Ann(intValue = 1, stringValue = "abc") class C
```

就像在Jave里一样，需要一个特殊的参数是' value`参数;它的值可以使用不明确的名称来指定。

```
// Java
public @interface AnnWithValue {
    String value();
}
```

```
// Kotlin
@AnnWithValue("abc") class C
```

如果在Java中 `value` 参数是array类型，在Kotlin中必须使用 `vararg` 这个参数。

```
// Java
public @interface AnnWithArrayValue {
    String[] value();
}
```

```kotlin
// Kotlin
@AnnWithArrayValue("abc", "foo", "bar") class C
```

For other arguments that have an array type, you need to use `arrayOf` explicitly:

```kotlin
// Java
public @interface AnnWithArrayMethod {
    String[] names();
}
```

```kotlin
// Kotlin
@AnnWithArrayMethod(names = arrayOf("abc", "foo", "bar")) class C
```

注解实例的值被视为Kotlin的属性。

```kotlin
// Java
public @interface Ann {
    int value();
}
```

```kotlin
// Kotlin
fun foo(ann: Ann) {
    val i = ann.value
}
```

反射

反射是一系列语言和库的特性，允许在运行时获取你代码结构。 把函数和属性作为语言的一等公民，反射它们（如获名称或者属性类型或者 方法）和使用函数式编程或反应是编程风格很像。

> 在Java平台，使用反射特性所需的运行时组件作为一个单独的 Jar 文件(`kotlin-reflect.jar`).这样做减小了不使用反射的应用程序库的大小. 如果你确实要使用反射,请确保该文件被添加到了项目路径. {:.note}

类引用

最基本的反射特性就是得到运行时的类引用。要获取引 用并使之成为静态类可以使用字面类语法:

```
val c = MyClass::class
```

引用是KClass类型.~~你可以使用 `KClass.properties` 和 `KClass.extensionProperties` 来获得类和父类所有属性引用的列表。~~

注意Kotlin类引用不完全与Java类引用一致.查看Java interop section 详细信息。

函数引用

我们有一个像下面这样的函数声明:

```
fun isOdd(x: Int) = x % 2 != 0
```

我们可以直接调用(`isOdd(5)`), 也可以把它作为一个值传给其他函数. 我们使用 `::` 操作符实现:

```
val numbers = listOf(1, 2, 3)
println(numbers.filter(::isOdd)) // prints [1, 3]
```

这里 `::isOdd` 是一个函数类型的值 `(Int) -> Boolean` .

`::` can be used with overloaded functions when the expected type is known from the context. For example:

```
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)) // refers to isOdd(x: Int)
```

Alternatively, you can provide the necessary context by storing the method reference in a variable with an explicitly specified type:

```
val predicate: (String) -> Boolean = ::isOdd   // refers to isOdd(x: String)
```

如果我们需要使用类成员或者一个扩展方法，它必须是有权访问的, 例如 `String::toCharArray` 带着一个 `String` : `String.() -> CharArray` 类型扩展函数.

例子: 函数组合

考量以下方法：

```
fun <A, B, C> compose(f: (B) -> C, g: (A) -> B): (A) -> C {
    return { x -> f(g(x)) }
}
```

它返回一个由俩个传递进去的函数的组合。 现在你可以把它用在可调用的引用上了：

```
fun length(s: String) = s.length

val oddLength = compose(::isOdd, ::length)
val strings = listOf("a", "ab", "abc")

println(strings.filter(oddLength)) // Prints "[a, abc]"
```

属性引用

我们同样可以用 `::` 操作符来访问Kotlin中的顶级类的属性：

```
var x = 1

fun main(args: Array<String>) {
    println(::x.get()) // prints "1"
    ::x.set(2)
    println(x)         // prints "2"
}
```

表达式 `::x` 推断为 `KProperty<Int>` 类型的属性对象,它允许我们 使用 `get()` 函数来读它的值或者使用 `name` 属性来得到它的值。更多请 查看docs on the `KProperty` class.

对于可变属性,例如 `var y = 1` , `::y` 返回值类型是 `KMutableProperty<Int>` , 它有一个 `set()` 方法.

A property reference can be used where a function with no parameters is expected:

```
val strs = listOf("a", "bc", "def")
println(strs.map(String::length)) // prints [1, 2, 3]
```

To access a property that is a member of a class, we qualify it:

```
class A(val p: Int)

fun main(args: Array<String>) {
    val prop = A::p
    println(prop.get(A(1))) // prints "1"
}
```

对于扩展属性:

```
val String.lastChar: Char
    get() = this[length - 1]

fun main(args: Array<String>) {
    println(String::lastChar.get("abc")) // prints "c"
}
```

与Java反射调用

在 java 平台上，标准库包括反射类的扩展，提供了到 java 反射 对象的映射(参看 `kotlin.reflect.jvm` 包)。 比如，想找到一个备用字段或者 java getter 方法，你可以这样写：

```
import kotlin.reflect.jvm.*

class A(val p: Int)

fun main(args: Array<String>) {
    println(A::p.javaGetter) // prints "public final int A.getP()"
    println(A::p.javaField)  // prints "private final int A.p"
}
```

To get the Kotlin class corresponding to a Java class, use the `.kotlin` extension property:

```
fun getKClass(o: Any): KClass<Any> = o.javaClass.kotlin
```

构造函数引用

构造函数可以像属性和方法那样引用. 它们可以使用在任何一个函数类型的对象的地方， 期望得到相同参数的构造函数，并返回一个适当类型的对象. 构造函数使用 `::` 操作符加类名引用. 考虑如下函数， 需要一个无参数函数返回类型是 `Foo`：

```
class Foo

fun function(factory : () -> Foo) {
    val x : Foo = factory()
}
```

Using `::Foo`, the zero-argument constructor of the class Foo, 我们可以简单的这样使用:

```
function(::Foo)
```

Type-Safe Builders

构建器(builders)的概念在*Groovy*社区非常热门。使用构建器我们可以用半声明(semi-declarative)的方式定义数据。构建器非常适合用来生成XML，组装UI组件，描述3D场景，以及很多其他功能...

很多情况下，Kotlin允许检查类型的构建器，这样比Groovy本身提供的构建器更有吸引力。

其他情况下，Kotlin也支持动态类型的构建器。

一个类型安全的构建器的示例

考虑下面的代码~~。这段代码是从这里摘出来并稍作修改的~~：

```
import com.example.html.* // see declarations below

fun result(args: Array<String>) =
    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"}

            // an element with attributes and text content
            a(href = "http://kotlinlang.org") {+"Kotlin"}

            // mixed content
            p {
                +"This is some"
                b {+"mixed"}
                +"text. For more see the"
                a(href = "http://kotlinlang.org") {+"Kotlin"}
                +"project"
            }
            p {+"some text"}

            // content generated by
            p {
                for (arg in args)
                    +arg
            }
        }
    }
```

这是一段完全合法的Kotlin代码。 这里可以在线运行这段代码（在你的浏览器中修改它）。

构建器的实现原理

让我们一步一步了解Kotlin中的类型安全构建器是如何实现的。 首先我们需要定义构建的模型，在这里我们需要构建的是HTML标签的模型。 用一些类就可以轻易实现。 比如 `HTML` 是一个类，描述 `<html>` 标签；它定义了子标签 `<head>` 和 `<body>` 。 (查看它的定义下方.)

现在我们先回忆一下我们在构建器代码中这么声明:

```
html {
 // ...
}
```

`html` 实际上是一个函数，其参数是一个lambda 表达式。 这个函数定义如下：

```
fun html(init: HTML.() -> Unit): HTML {
    val html = HTML()
    html.init()
    return html
}
```

这个函数定义一个叫做 `init` 的参数，本身是个函数。 The type of the function is `HTML.() -> Unit` , which is a *function type with receiver*. This means that we need to pass an instance of type `HTML` (a *receiver*) to the function, and we can call members of that instance inside the function. The receiver can be accessed through the ***`this`*** keyword:

```
html {
    this.head { /* ... */ }
    this.body { /* ... */ }
}
```

(`head` 和 `body` 都是 `HTML` 类的成员函数)

现在，和平时一样， ***`this`*** 可以省略掉，所以我们就可以得到一段已经很有构建器风格的代码：:

```
html {
    head { /* ... */ }
    body { /* ... */ }
}
```

那么，这个调用做了什么？让我们看看上面定义的 `html` 函数的函数体。它新建了一个 `HTML` 对象，接着调用传入的函数来初始化它，（在我们上面的 `HTML` 例子中，在 `html` 对象上调用了 `body()` 函数），接着返回this实例。 这正是构建器所应做的。

HTML类里定义的 `head` 和 `body` 函数的定义类似于 `html` 函数。唯一的区别是，它们将新建的实力先添加到html的children属性上，再返回：

```
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的是完全相同的事情，所以我们可以定义一个泛型函数 `initTag` ：

```
    protected fun <T : Element> initTag(tag: T, init: T.() -> Unit): T {
        tag.init()
        children.add(tag)
        return tag
    }
```

现在我们的函数变成了这样:

```
fun head(init: Head.() -> Unit) = initTag(Head(), init)

fun body(init: Body.() -> Unit) = initTag(Body(), init)
```

我们可以使用它们来构建 `<head>` 和 `<body>` 标签.

另外一个需要讨论的是如何给标签添加文本内容。在上面的例子里我们使用了如下的方式：

```
html {
    head {
        title {+"XML encoding with Kotlin"}
    }
    // ...
}
```

所以基本上，我们直接在标签体中添加文字，但前面需要在前面加一个 `+` 符号。 事实上这个符号是用一个扩展函数 `unaryPlus()` 来定义的。 `unaryPlus()` 是抽象类 `TagWithText` (`Title` 的父类)的成员函数。

```
fun String.unaryPlus() {
    children.add(TextElement(this))
}
```

所以，前缀 `+` 所做的事情是把字符串用 `TextElement` 对象包裹起来，并添加到children集合上，这样就正确加入到标签树中了。

所有这些都定义在包 `com.example.html` 里，上面的构建器例子在代码顶端导入了。 下一节里你可以详细的浏览这个名字空间中的所有定义。

包 `com.example.html` 的完整定义

下面是包 `com.example.html` 的定义（只列出了上面的例子中用到的元素）。它可以生成一个HTML树。 代码中大量使用了扩展函数和 带接收者的lambda技术

```
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")
    }
}

abstract class Tag(val name: String) : Element {
    val children = arrayListOf<Element>()
    val attributes = hashMapOf<String, String>()

    protected fun <T : Element> initTag(tag: T, init: T.() -> Unit): T {
        tag.init()
        children.add(tag)
        return tag
    }

    override fun render(builder: StringBuilder, indent: String) {
        builder.append("$indent<$name${renderAttributes()}>\n")
        for (c in children) {
            c.render(builder, indent + "  ")
        }
        builder.append("$indent</$name>\n")
```

```
    }

    private fun renderAttributes(): String {
        val builder = StringBuilder()
        for (a in attributes.keys) {
            builder.append(" $a=\"${attributes[a]}\"")
        }
        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") {
    public var href: String
```

```
        get() = attributes["href"]!!
        set(value) {
            attributes["href"] = value
        }
}

fun html(init: HTML.() -> Unit): HTML {
    val html = HTML()
    html.init()
    return html
}
```

参考

API 和语法

- API 参考
- 语法

API参考

- 参见 http://kotlinlang.org/api/latest/jvm/stdlib/

语法

Notation

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"` `":"` (`"["` annotationEntry+ `"]"` | annotationEntry)
;

packageHeader

(used by preamble)

: modifiers `"package"` SimpleName{ `"."` } SEMI?
;

import

(used by preamble, package)

: `"import"` SimpleName{ `"."` } (`"."` `"*"` | `"as"` SimpleName)? SEMI?
;

See Imports

toplevelObject

(used by package, kotlinFile)

: package
: class
: object

: function
: property
;

package

(used by toplevelObject)

: `"package"` SimpleName{ `"."` } `"{"`
import
toplevelObject
`"}"`
;

See Packages

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, object)

: (`"{"` members `"}"`)?
;

members

(used by enumClassBody, classBody)

: memberDeclaration*
;

delegationSpecifier

(used by objectLiteral, class, object)

: constructorInvocation
: userType
: explicitDelegation
;

explicitDelegation

(used by delegationSpecifier)

: userType `"by"` expression
;

typeParameters

(used by 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"`
;

valueParameters

(used by secondaryConstructor, function)

: `"("` functionParameter{ `","` }? `")"`
;

functionParameter

(used by valueParameters, primaryConstructor)

: modifiers (`"val"` | `"var"`)? parameter (`"="` expression)?
;

initializer

(used by enumEntry)

: annotations constructorInvocation
;

block

(used by catchBlock, anonymousInitializer, secondaryConstructor, functionBody, try,finallyBlock)

: `"{"` statements `"}"`
;

function

(used by memberDeclaration, declaration, toplevelObject)

: modifiers `"fun"` typeParameters?
(type `"."` | annotations)?
SimpleName
typeParameters? valueParameters (`":"` type)?
typeConstraints
functionBody?
;

functionBody

(used by getter, setter, function)

: block
: `"="` expression
;

variableDeclarationEntry

(used by for, property, multipleVariableDeclarations)

: SimpleName (`":"` type)?
;

multipleVariableDeclarations

(used by for, property)

: `"("` variableDeclarationEntry{ `","` } `")"`
;

property

(used by memberDeclaration, declaration, toplevelObject)

: modifiers (`"val"` | `"var"`)
typeParameters? (type `"."` | annotations)?
(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*
;

enumEntries

(used by enumClassBody)

: (enumEntry `","` ?)?
;

enumEntry

(used by enumEntries)

: modifiers SimpleName ((`":"` initializer) | (`"("` arguments `")"`))? classBody?
;

Types

See Types

type

(used by isRHS, simpleUserType, parameter, functionType, atomicExpression, getter,variableDeclarationEntry, property, typeArguments, typeConstraint, function)

: annotations typeDescriptor
;

typeDescriptor

(used by nullableType, typeDescriptor, type)

: `"("` typeDescriptor `")"`
: functionType
: userType
: nullableType
: `"dynamic"`
;

nullableType

(used by typeDescriptor)

: typeDescriptor `"?"`

userType

(used by typeParameter, catchBlock, callableReference, typeDescriptor,delegationSpecifier, constructorInvocation, explicitDelegation)

: (`"package"` `"."`)? simpleUserType{ `"."` }
;

simpleUserType

(used by userType)

: SimpleName (`"<"` (optionalProjection type | `"*"`){ `","` } `">"`)?
;

optionalProjection

(used by simpleUserType)

: varianceAnnotation
;

functionType

(used by typeDescriptor)

: (type `"."`)? `"("` (parameter | modifiers type){ `","` } `")"` `"->"` type?
;

Control structures

See Control structures

if

(used by atomicExpression)

: `"if"` `"("` expression `")"` expression SEMI? (`"else"` expression)?
;

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 `")"` expression
;

while

(used by loop)

: `"while"` `"("` expression `")"` expression
;

doWhile

(used by loop)

: `"do"` expression `"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, whenEntry, arrayAccess,statement, 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 isRHS)?
;

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
: SimpleName

: FieldName
: `"package"`
;

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 `"}"`
;

isRHS

(used by namedInfix, whenCondition)

: type
;

declaration

(used by statement)

: function
: property
: class
: object
;

statement

(used by statements)

: declaration
: 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)

: (`"@"` annotationEntry)* 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 `"}"`
: `"{"` (modifiers SimpleName){ `","` } `"->"` statements `"}"`
;

statements

(used by block, functionLiteral)

: SEMI *statement{SEMI+} SEMI*
;

constructorInvocation

(used by delegationSpecifier, initializer)

: userType callSuffix
;

arrayAccess

(used by postfixUnaryOperation)

: `"["` expression{ `","` } `"]"`
;

objectLiteral

(used by atomicExpression)

: `"object"` (`":"` delegationSpecifier{ `","` })? classBody
;

Pattern matching

See When-expression

when

(used by atomicExpression)

: `"when"` (`"("` expression `")"`)? `"{"`
whenEntry*
`"}"`
;

whenEntry

(used by when)

: whenCondition{ `","` } `"->"` expression SEMI
: `"else"` `"->"` expression SEMI
;

whenCondition

(used by whenEntry)

: expression
: (`"in"` | `"!in"`) expression
: (`"is"` | `"!is"`) isRHS
;

Modifiers

modifiers

(used by typeParameter, getter, packageHeader, class, property, functionLiteral,function, functionType, secondaryConstructor, setter, enumEntry, companionObject,primaryConstructor, functionParameter)

: modifier*
;

modifier

(used by modifiers)

: modifierKeyword
;

modifierKeyword

(used by modifier)

: classModifier
: accessModifier
: varianceAnnotation
: memberModifier
: annotations
;

classModifier

(used by modifierKeyword)

: `"abstract"`
: `"final"`
: `"enum"`
: `"open"`
: `"annotation"`
;

memberModifier

(used by modifierKeyword)

: `"override"`
: `"open"`
: `"final"`
: `"abstract"`
;

accessModifier

(used by modifierKeyword)

: `"private"`
: `"protected"`
: `"public"`

: `"internal"`
;

varianceAnnotation

(used by modifierKeyword, optionalProjection)

: `"in"`
: `"out"`
;

Annotations

annotations

(used by catchBlock, prefixUnaryOperation, for, modifierKeyword, class, property,type, typeConstraint, function, initializer)

: (annotation | annotationList)*
;

annotation

(used by annotations)

: `"@"` (annotationUseSiteTarget `":"`)? unescapedAnnotation
;

annotationList

(used by annotations)

: `"@"` (annotationUseSiteTarget `":"`)? `"["` unescapedAnnotation+ `"]"`
;

annotationUseSiteTarget

(used by annotation, annotationList)

: `"file"`
: `"field"`
: `"property"`
: `"get"`
: `"set"`

: `"param"`
: `"sparam"`
;

unescapedAnnotation

(used by annotation, annotationList)

: SimpleName{ `"."` } typeArguments? valueArguments?
;

Lexical structure

helper

Digit

(used by IntegerLiteral, HexDigit)

: [`"0"` .. `"9"`];

IntegerLiteral

(used by literalConstant)

: Digit+?

FloatLiteral

(used by literalConstant)

: \;

helper

HexDigit

(used by RegularStringPart, HexadecimalLiteral)

: Digit | [`"A"` .. `"F"` , `"a"` .. `"f"`];

HexadecimalLiteral

(used by literalConstant)

: `"0x"` HexDigit+;

CharacterLiteral

(used by literalConstant)

: \;

See Basic types

NoEscapeString

(used by literalConstant)

: \<"""-quoted string>;

RegularStringPart

(used by stringTemplateElement)

: \
ShortTemplateEntryStart:
: `"$"`
EscapeSequence:
: UnicodeEscapeSequence | RegularEscapeSequence
UnicodeEscapeSequence:
: `"\u"` HexDigit{4}
RegularEscapeSequence:
: `"\"` \

See String templates

SEMI

(used by whenEntry, if, statements, packageHeader, property, import)

: \;

SimpleName

(used by typeParameter, catchBlock, simpleUserType, atomicExpression, LabelName,package, packageHeader, class, object, functionLiteral, infixFunctionCall, function,parameter, callableReference, FieldName, variableDeclarationEntry,stringTemplateElement, setter, enumEntry, import, valueArguments,unescapedAnnotation, typeConstraint)

: \
: `"`` ` ``"` \ `"`` ` ``"`
;

See Java interoperability

FieldName

(used by atomicExpression)

: `"$"` SimpleName;

LabelName

(used by labelReference, labelDefinition)

: `"@"` SimpleName;

See Returns and jumps

Java 互操作

关于 Kotlin 和 Java 互操作性你需要知道的。

在Kotlin中调用Java代码

Kotlin 在设计时就是以与 java 交互为中心的。现存的 Java 代码可以在 kotlin 中使用，并且 Kotlin 代码也可以 在 Java 中流畅运行。这节我们会讨论在 kotlin 中调用 Java 代码的细节。

基本所有的 Java 代码都可以运行

```
import java.util.*

fun demo(source: List<Int>) {
    val list = ArrayList<Int>()
    // 'for'-loops work for Java collections:
    for (item in source) {
        list.add(item)
    }
    // Operator conventions work as well:
    for (i in 0..source.size() - 1) {
        list[i] = source[i] // get and set are called
    }
}
```

Getters 和 Setters

若一个属性的getter/setter方法按照约定的规范进行命名(getter方法以 `get` 开头不带参数/setter方法以 `set` 开头 带一个参数)，那么在Kotlin中可以直接对这个属性进行访问。如：

```
import java.util.Calendar

fun calendarDemo() {
    val calendar = Calendar.getInstance()
    if (calendar.firstDayOfWeek == Calendar.SUNDAY) {  // call getFirstDayOfWeek()
        calendar.firstDayOfWeek = Calendar.MONDAY       // call setFirstDayOfWeek()
    }
}
```

需要注意的是，对于只有setter方法的属性，kotlin目前还不支持对它的直接访问。

返回void的方法

如果一个Java方法返回void，那么在Kotlin中，它会返回 `Unit` 。 万一有人使用它的返回值，Kotlin的编译器会在调用的地方赋值， 因为这个值本身已经提前可以预知了(这个值就是 `Unit`)。

将Java代码中与Kotlin关键字冲突的标识符进行转义

一些Kotlin的关键字在Java中是合法的标识符: ***in***, ***object***, ***is***, 等等. 如果一个Java库在方法中使用了Kotlin关键字,你仍然可以使用这个方法 使用反引号(`)转义来避免冲突。

```
foo.`is`(bar)
```

Null安全性和平台类型

Java中的所有引用都可能是 ***null*** 值，这使得Kotlin严格的null控制对来自Java的对象来说变得不切实际。 在Kotlin中Java声明类型被特别对待叫做*platform types*.这种类型的Null检查是不严格的， 所以他们还维持着同Java中一样的安全性 (更多参见下面)。

考虑如下例子:

```
val list = ArrayList<String>() // non-null (constructor result)
list.add("Item")
val size = list.size() // non-null (primitive int)
val item = list[0] // platform type inferred (ordinary Java object)
```

当我们调用平台类型的变量上的方法时，Kotlin不会在编译阶段报出可能为空的错误， 但在运行时，会产生空指针异常，或者是断言失败的错误。后者是因为kotlin为了阻止null值传播会生成非空断言语句。

```
item.substring(1) // 允许, 如果item为空会抛异常
```

平台类型是不可转义的，也就是说我们不能在程序里把他们写出来。 当把一个平台数值赋值给kotlin变量的时候（变量会有一个推断出来的平台类型， 上面的例子里就是 `item` 的类型），我们可以用类型推断，或者指定我们期望的类型（nullable和non-null类型都可以）：

```
val nullable: String? = item // 允许，没有问题
val notNull: String = item // 允许，运行时可能失败
```

如果我们指定了一个非空类型，编译器会在赋值前额外生成一个断言。这样Kotlin的非空变量就不会有 空值。当把平台数值传递给只接受非空数值的kolin函数的时候，也同样会生成这个断言， 编译器尽可能的阻止空置在程序里传播。（因为泛型的存在， 有时也不能百分百的阻止）

平台类型的概念

如上所述，平台类型不能在程序里显式的出现， 所以没有针对他们的语法。 然而，编译器和IDE有时需要显式他们(如在错误信息，参数信息中)，所以我们用 一个好记的标记来表示他们：

- `T!` 表示"`T` 或者 `T?`"
- `(Mutable)Collection<T>!` 表示"`T` 的java集合，可变的或不可变的，可空的或非空的"
- `Array<(out) T>!` 表示"`T` (或 `T` 的子类)的java数组，可空的或非空的"

可空性注解

Java types which have nullability annotations are represented not as platform types, but as actual nullable or non-null Kotlin types. The compiler supports several flavors of nullability annotations, including:

- JetBrains (`@Nullable` and `@NotNull` from the `org.jetbrains.annotations` package)
- Android (`com.android.annotations` and `android.support.annotations`)
- JSR-305 (`javax.annotation`)
- FindBugs (`edu.umd.cs.findbugs.annotations`)
- Eclipse (`org.eclipse.jdt.annotation`)
- Lombok (`lombok.NonNull`).

You can find the full list in the Kotlin compiler source code.

已映射类型

Kotlin特殊处理一部分java类型。这些类型不是通过as或is来直接转换，而是映射到了指定的kotlin类型上。 映射只发生在编译期间，运行时仍然是原来的类型。 java的原生类型映射成如下kotlin类型（记得 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`

{:.zebra}

一些非原生类型也会作映射：

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.Void`	`kotlin.Nothing!`
`java.lang.CharSequence`	`kotlin.CharSequence!`
`java.lang.String`	`kotlin.String!`
`java.lang.Number`	`kotlin.Number!`
`java.lang.Throwable`	`kotlin.Throwable!`

{:.zebra}

集合类型在Kotlin里可以是只读的或可变的，因此Java集合类型作如下映射：（下表所有的Kotlin类型都在 `Kotlin` 包里）

Java类型	Kotlin只读类型	Kotlin可变类型	加载的斗
`Iterator<T>`	`Iterator<T>`	`MutableIterator<T>`	`(Mutable)Iter`
`Iterable<T>`	`Iterable<T>`	`MutableIterable<T>`	`(Mutable)Iter`
`Collection<T>`	`Collection<T>`	`MutableCollection<T>`	`(Mutable)Coll`
`Set<T>`	`Set<T>`	`MutableSet<T>`	`(Mutable)Set<`
`List<T>`	`List<T>`	`MutableList<T>`	`(Mutable)List`
`ListIterator<T>`	`ListIterator<T>`	`MutableListIterator<T>`	`(Mutable)List`
`Map<K, V>`	`Map<K, V>`	`MutableMap<K, V>`	`(Mutable)Map<`
`Map.Entry<K, V>`	`Map.Entry<K, V>`	`MutableMap.MutableEntry<K,V>`	`(Mutable)Map.` `(Mutable)Entry`

{:.zebra}

Java数组的映射在这里提到过 below：

Java类型	Kotlin类型
`int[]`	`kotlin.IntArray!`
`String[]`	`kotlin.Array<(out) String>!`

{:.zebra}

Kotlin中的Java泛型

Kotlin的泛型和Java的有些不同（详见 Generics）。当引入java类型的时候，我们作如下转换：

- Java的通配符转换成类型投射
  - `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号投射
  - `List` 转换成 `List<*>!`，也就是 `List<out Any?>!`

和Java一样，Kotlin在运行时不保留泛型，即对象不知道传递到他们构造器中的那些参数的的实际类型。 也就是，`ArrayList<Integer>()` 和 `ArrayList<Character>()` 是区分不出来的。 这意味着，不可能用 ***is*** -来检测泛型。 Kotlin只允许用 ***is*** -来检测星号投射的泛型类型:

```
if (a is List<Int>) // 错误: 不能检测是否是一个Int的List
// but
if (a is List<*>) // 可以：不保证list里面的内容类型
```

Java数组

和Java不同，Kotlin里的数组不是协变的。Kotlin不允许我们把 `Array<String>` 赋值给 `Array<Any>` ，从而避免了可能的运行时错误。Kotlin也禁止我们把一个子类的数组当做父类的数组传递进Kotlin的方法里。但是对Java方法，这是允许的（考虑这种形式的平台类型 platform types `Array<(out) String>!` ）。

Java平台上，原生数据类型的数组被用来避免封箱/开箱的操作开销。由于Kotlin隐藏了这些实现细节，就得有一个变通方法和Java代码交互。每个原生类型的数组都有一个特有类(specialized class)来处理这种问题(`IntArray` , `DoubleArray` , `CharArray` ...)。它们不是 `Array` 类，而是被编译成java的原生数组，来获得最好的性能。

假设有一个Java方法，它接受一个表示索引的int数组作参数

```
public class JavaArrayExample {

    public void removeIndices(int[] indices) {
        // code here...
    }
}
```

在Kotlin里你可以这样传递一个原生数组:

```
val javaObj = JavaArrayExample()
val array = intArrayOf(0, 1, 2, 3)
javaObj.removeIndices(array)  // passes int[] to method
```

当编译成jvm字节码的时候，编译器会优化对数组的访问，确保不会产生额外的负担。

```
val array = arrayOf(1, 2, 3, 4)
array[x] = array[x] * 2 // 不会生成对get() 和 set()的调用
for (x in array) { // 不会创建迭代器
    print(x)
}
```

即便是用索引遍历数组。

```
for (i in array.indices) {// 不会创建迭代器
    array[i] += 2
}
```

最后， `in` -检测也没有额外负担。

```
if (i in array.indices) { // 和 (i >= 0 && i < array.size) 一样
    print(array[i])
}
```

Java Varargs

Java类也会这样声明方法，表示参数是可变参数。

```
public class JavaArrayExample {

    public void removeIndices(int... indices) {
        // code here...
    }
}
```

这种情况，你需要用展开操作符 `*` 来传递 `IntArray`：

```
val javaObj = JavaArray()
val array = intArrayOf(0, 1, 2, 3)
javaObj.removeIndicesVarArg(*array)
```

目前无法传递 ***null*** 给一个变参的方法。

操作符

虽然Java不能自定义操作符重载，但Kotlin允许任意 使用方法名合法的方法与标示符进行操作符重载，也可以自定义其它约定（如 `invoke()` 等）。 但调用Java代码的时候，使用中缀语法(`infix call syntax`)是不被允许的。

受检异常

在Kotlin里，所有的异常都是非受检的, 也就是说，编译器不会强制你去捕捉任何异常。 因此，你调用一个声明了异常的java方法的时候，kotlin不会强制你作处理。

```
fun render(list: List<*>, to: Appendable) {
    for (item in list) {
        to.append(item.toString()) // Java里会让你在这里捕捉IOException
    }
}
```

对象方法

当java类型被引入到kotlin里时，所有的 `java.lang.Object` 类型引用，会被转换成 `Any` 。 因为 `Any` 不是平台独有的，它仅声明了三个成员方法： `toString()` , `hashCode()` 和 `equals()` ， 所以为了能用到 `java.lang.Object` 的其他方法，kotlin采用了扩展函数。

wait()/notify()

Effective Java 第69条善意的提醒了要用concurrency类而不是 `wait()` 和 `notify()` 。 因此， `Any` 不提供这两个方法。 你一定要用的话，就把它转换成 `java.lang.Object` 。

```
(foo as java.lang.Object).wait()
```

getClass()

获取一个对象的类型信息，我们可以用javaClass这个扩展属性。

```
val fooClass = foo.javaClass
```

用javaClass()，而不是java里的写法 `Foo.class` 。

```
val fooClass = javaClass<Foo>()
```

clone()

要重写 `clone()` ，扩展 `kotlin.Cloneable` ：

```
class Example : Cloneable {
  override fun clone(): Any { ... }
}
```

不要忘了 Effective Java,第11条: 谨慎的重写克隆。

finalize()

要重载 `finalize()` , 你要做的仅仅是声明它，不需要 ***`override`*** 关键字：

```
class C {
    protected fun finalize() {
        // 具体逻辑
    }
}
```

根据 Java 的规则， `finalize()` 不能为 ***`private`*** 。

从 Java类的继承

在kotlin里，超类里最多只能有一个java类(java接口数目不限)。这个java类必须放在超类列表的最前面。

访问静态成员

java类的静态成员就是它们的 “伴生对象”。我们无法将这样的“伴生对象”当作数值来传递， 但可以显式的访问它们，比如：

```
if (Character.isLetter(a)) {
    // ...
}
```

Java 反射

Java反射可以用在kotlin类上，反之亦然。前面提过，你可以 `instance.javaClass` 或者 `ClassName::class.java` 开始基于 `java.lang.Class` 的java反射操作。

Other supported cases include acquiring a Java getter/setter method or a backing field for a Kotlin property, a `KProperty` for a Java field, a Java method or constructor for a `KFunction` and vice versa.

SAM(单抽象方法) 转换

就像 Java 8 那样，Kotlin 支持 SAM 转换，这意味着 Kotlin 函数字面量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 能够跟这个 Kotlin 函数的参数类型匹配的上。

你可以这样创建SAM接口的实例：

```
val runnable = Runnable { println("This runs in a runnable") }
```

...在方法调用里:

```
val executor = ThreadPoolExecutor()
// Java签名: void execute(Runnable command)
executor.execute { println("This runs in a thread pool") }
```

如果 Java 类有多个接受函数接口的方法，你可以用一个 适配函数来把闭包转成你需要的 SAM 类型。编译器也会在必要时生成这些适配函数。

```
executor.execute(Runnable { println("This runs in a thread pool") })
```

注意SAM的转换只对接口有效，对抽象类无效，即使它们就只有一个 抽象方法。

还要注意这个特性只针对和 Java 的互操作；因为 Kotlin 有合适的函数类型，把函数自动转换成 Kotlin 接口的实现是没有必要的，也就没有支持了。

在Kotlin中使用JNI

如果要声明一个使用本机代码(C 或者 C++)实现的方法，你需要给它加上 `external` 标识符(等同于Java里的 `native`)

```
external fun foo(x: Int): Double
```

余下的工作和Java完全一样

Java调用Kotlin代码

Java可以轻松调用Kotlin代码。

属性

属性getters被转换成 *get*-方法，setters转换成*set*-方法。

包级别的函数

`example.kt` 文件中 `org.foo.bar` 包内声明的所有的函数和属性，都会被放到一个叫 `org.foo.bar.ExampleKt` 的java类里。

```
// example.kt
package demo

class Foo

fun bar() {
}
```

```
// Java
new demo.Foo();
demo.ExampleKt.bar();
```

可以使用 `@JvmName` 注解自定义生成的Java 类的类名：

```
@file:JvmName("DemoUtils")

package demo

class Foo

fun bar() {
}
```

```
// 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，类名相同或者有相同的 `@JvmName` 注解）通常会报错，然而，可以在每个文件添加 `@JvmMultifileClass` 注解，可以让编译器生成一个统一的带有特殊名字的类，这个类包含了对应这些文件中所有的声明。

```
// oldutils.kt
@file:JvmName("Utils")
@file:JvmMultifileClass

package demo

fun foo() {
}
```

```
// newutils.kt
@file:JvmName("Utils")
@file:JvmMultifileClass

package demo

fun bar() {
}
```

```
// Java
demo.Utils.foo();
demo.Utils.bar();
```

实例字段

如果在 Java 需要像字段一样调用一个 Kotlin 的属性，你需要使用 `@JvmField` 注解。这个字段与属性具有相同的可见性。属性符合有幕后字段（backing field）、非私有、没有 `open`，`override` 或者 `const` 修饰符、不是被委托的属性这些条件才可以使用 `@JvmField` 注解。

```
class C(id: String) {
    @JvmField val ID = id
}
```

```
// Java
class JavaClient {
    public String getID(C c) {
        return c.ID;
    }
}
```

延迟初始化 的属性（在Java中）也可以被作为字段调用，字段的可见性和 `lateinit` 属性的 setter 相同。

静态字段

在一个命名对象或者伴生对象中声明的Koltin属性会持有静态幕后字段（backing fields），这些字段存在于该命名对象或者伴生对象中的。

通常，这些字段都是private的，但是他们可以通过以下方式暴露出来。

- `@JvmField` 注解;
- `lateinit` 修饰符;
- `const` 修饰符.

用 `@JvmField` 注解该属性可以生成一个与该属性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);
// public static final field in Key class
```

在命名对象或者伴生对象中的一个延迟初始化 的属性都有一个静态实际字段，字段和该属性的setter也有相同的可见性。

```
object Singleton {
    lateinit var provider: Provider
}
```

```
// Java
Singleton.provider = new Provider();
// public static non-final field in Singleton class
```

使用 `const` 注解可以将 Kotlin 属性转换成 Java 中的静态字段。

```
// file example.kt

object Obj {
    const val CONST = 1
}

class C {
    companion object {
        const val VERSION = 9
    }
}

const val MAX = 239
```

In Java:

```
int c = Obj.CONST;
int d = ExampleKt.MAX;
int v = C.VERSION;
```

静态方法

正如上面所说，Kotlin 自动为包级函数生成了静态方法 在Kotlin 中，还可以通过 `@JvmStatic` 注解在命名对象或者伴生对象中定义的函数来生成对应的静态方法。 例如：

```
class C {
    companion object {
        @JvmStatic fun foo() {}
        fun bar() {}
    }
}
```

现在， `foo()` 在java里就是静态的了，而 `bar()` 不是：

```
C.foo(); // 没问题
C.bar(); // 错误: 不是一个静态方法
```

同样的，命名对象：

```
object Obj {
    @JvmStatic fun foo() {}
    fun bar() {}
}
```

Java 里：

```
Obj.foo(); // 没问题
Obj.bar(); // 错误
Obj.INSTANCE.bar(); // 对单例的方法调用
Obj.INSTANCE.foo(); // 也行
```

通过使用 `@JvmStatic` 注解对象的属性或伴生对象，使对应的getter 和 setter 方法在这个对象或者包含这个伴生对象的类中也成为静态成员。

用@JvmName解决签名冲突

有时我们想让一个 Kotlin 里的命名函数在字节码里有另外一个 JVM 名字。 最突出的例子就是类名型擦除:

```
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;`. 如果我们真的相让它们在 Kotlin里用同一个名字，我们需要用 `@JvmName` 去注释它们中的一个（或两个），指定的另外一个名字当参数：

```
fun List<String>.filterValid(): List<String>

@JvmName("filterValidInt")
fun List<Int>.filterValid(): List<Int>
```

在Kotlin里它们可以都用 `filterValid` 来访问，但是在Java里，它们是 `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
```

生成重载

通常，如果你写一个有默认参数值的 Kotlin 方法，在 Java 里，只会有一个有完整参数的签名。如果你要暴露多个重载给java调用者，你可以使用 @JvmOverloads 注解。

```
@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。在上面这个例子里，生成下面的方法：

```
// Java
void f(String a, int b, String c) { }
void f(String a, int b) { }
void f(String a) { }
```

构造函数，静态函数等也能用这个标记。但他不能用在抽象方法上，包括 接口中的方法。

注意一下，Secondary Constructors 描述过，如果一个类的所有构造函数参数都有默认 值，会生成一个公开的无参构造函数。这就算 没有@JvmOverloads 注解也有效。

受检异常

上面说过，kotlin没有受检异常。 所以，通常，kotlin函数的java签名没有声明抛出异常。 于是如果我们有一个kotlin函数：

```
// example.kt
package demo

fun foo() {
    throw IOException()
}
```

然后我们想要在java里调用它，捕捉这个异常：

```
// Java
try {
  demo.Example.foo();
}
catch (IOException e) { // 错误: foo() 没有声明 IOException
  // ...
}
```

因为 `foo()` 没有声明 `IOException`，java编译器报了错误信息。 为了解决这个问题，要在kotlin里使用 `@throws` 标记。

```
@Throws(IOException::class)
fun foo() {
    throw IOException()
}
```

Null安全性

当从 Java 中调用 Kotlin 函数时，没人阻止我们传递 ***`null`*** 给一个非空参数。 这就是为什么 Kotlin 给所有期望非空参数的公开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` 。

可变泛型

当Kotlin 的类使用了 declaration-site variance，从Java 的角度看起来有两种用法，比如我们下面涉及到的这种用法的类和两个函数。

```
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函数转换成 Java 代码的方式可能会是这样：

```
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"))` ，但是这样的写法在 Java 中是无法通过的，因为在 Java 中 `Box` 的泛型参数 `T` 是不可变的， `Box<Derived>` 实际上并不是 `Box<Base>` 的子类。如果在 Java 中要编译通过我们需要像下面这样定义 `unboxBase` ：

```
Base unboxBase(Box<? extends Base> box) { ... }
```

我们在这里通过使用 Java 的 通配符类型 (`? extends Base`) 去模拟declaration-site variance，因为在 Java 中只能这么做。

当作为参数的时候，为了让 Kotlin 的 API 工作，针对 `Box` 我们将Koltin 中的 `Box<Super>` 在 Java 中生成为 `Box<? extends Super>` (`Foo` 将生成为 `Foo<? super Bar>`)。当作为返回值的时候，我们不需要生成通配符类型，因为如果生成通配符在 Java 中还需要做其他操作来转换(这是常见的 Java 代码风格)。因此上面例子中的函数实际上会被转换成下面的代码：

```
// 作为返回类型 - 没有泛型
Box<Derived> boxDerived(Derived value) { ... }

// 作为参数 - 带有泛型
Base unboxBase(Box<? extends Base> box) { ... }
```

注意：如果参数类型是 final 的，就不用生成泛型了，比如，无论在什么地方 `Box<String>` 转换成 Java 代码始终还是 `Box<String>` ，

如果我们不想要默认生成的通配符，需要自己指定可以使用 `@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
// 将被转换成
// Base unboxBase(Box<Base> box) { ... }
```

注意： `@JvmSuppressWildcards` 不是只可以用在单独的类型参数上面，是可以是用在所有声明上，比如 函数，类等，其对应下面所有的泛型都不会自动转换为Java 中的通配符。

Nothing 类型的转换

`Nothing` 是一种特殊的类型，因为它在 Java 中没有类型相对应。事实上，每个 Java 的引用类型，包括 `java.lang.Void` 都可以接受 `null` 值，但是 Nothing 不行，因此在 Java 世界中没有什么可以代表这个类型，这就是为什么在Kotlin 中要生成原始类型需要使用 `Nothing` 。

```
fun emptyList(): List<Nothing> = listOf()
// 被转换为
// List emptyList() { ... }
```

JavaScript

Kotlin 与 JavaScript 合用

- 动态类型
- JavaScript 互操作性
- JavaScript 反射

动态类型

> The dynamic type is not supported in code targeting the JVM {:.note}

作为一个静态类型语言,Kotlin仍然可能会与无类型或者弱类型语言相互调用， 比如JavaScript,为了这方面使用可以使用 `dynamic` 类型。

```
val dyn: dynamic = ...
```

`dynamic` 类型关闭了Kotlin类型检查:

- 这样的类型可以分配任意变量或者在任意的地方作为参数传递,
- 任何值都可以分配为 `dynamic` 类型，或者作为参数传递给任何接受 `dynamic` 类型参数的函数，
- 这样的值不 `null` 检查。

`dynamic` 最奇特的特性就是可以在 `dynamic` 变量上调用任何属性或任何函数 ：

```
dyn.whatever(1, "foo", dyn) // 'whatever' is not defined anywhere
dyn.whatever(*arrayOf(1, 2, 3))
```

在JavaScript平台这段代码被编译为"as is": `dyn.whatever(1)` 在Kotlin中 `dyn.whatever(1)` 生成的 JavaScript 代码.

动态调用返回 `dynamic` 作为结果，因此我们可以轻松实现链式调用：

```
dyn.foo().bar.baz()
```

当给动态调用传递一个 lambda 表达式时，所有的参数默认都是 `dynamic` ：

```
dyn.foo {
    x -> x.bar() // x is dynamic
}
```

更多细节, 查看规范文档.

JavaScript Interoperability

JavaScript Modules

Since Kotlin version 1.0.4 you can compile your Kotlin projects to JS modules for popular module systems. Here is the list of available options:

1. Plain. Don't compile for any module system. As usual, you can access module `moduleName` via `kotlin.modules.moduleName`, or by just `moduleName` identifier put in the global scope. This option is used by default.
2. Asynchronous Module Definition (AMD), which is in particular used by require.js library.
3. CommonJS convention, widely used by node.js/npm (`require` function and `module.exports` object)
4. Unified Module Definitions (UMD), which is compatible with both *AMD* and *CommonJS*, and works as "plain" when neither *AMD* nor *CommonJS* is available.

Choosing the target module system depends on your build environment:

From IDEA

Open File -> Settings, select "Build, Execution, Deployment" -> "Compiler" -> "Kotlin compiler". Choose appropriate module system in "Module kind" field.

From Maven

To select module system when compiling via Maven, you should set `moduleKind` configuration property, i.e. your `pom.xml` should look like this:

```
<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>
    <!-- Insert these lines -->
    <configuration>
        <moduleKind>commonjs</moduleKind>
    </configuration>
    <!-- end of inserted text -->
</plugin>
```

Available values are: `plain`, `amd`, `commonjs`, `umd`.

From Gradle

To select module system when compiling via Gradle, you should set `moduleKind` property, i.e.

```
compileKotlin2Js.kotlinOptions.moduleKind = "commonjs"
```

Available values are similar to Maven

Notes

We ship `kotlin.js` standard library as a single file, which is itself compiled as an UMD module, so you can use it with any module system described above.

Although for now we don't support WebPack and Browserify directly, we tested `.js` files produced by Kotlin compiler with WebPack and Browserify, so Kotlin should work with these tools properly.

@JsName Annotation

In some cases (for example, to support overloads), the Kotlin compiler mangles the names of generated functions and attributes in JavaScript code. To control the generated names, you can use the `@JsName` annotation:

```
// Module 'kjs'

class Person(val name: String) {
    fun hello() {
        println("Hello $name!")
    }

    @JsName("helloWithGreeting")
    fun hello(greeting: String) {
        println("$greeting $name!")
    }
}
```

Now you can use this class from JavaScript in the following way:

```
var person = new kjs.Person("Dmitry");   // refers to module 'kjs'
person.hello();                          // prints "Hello Dmitry!"
person.helloWithGreeting("Servus");      // prints "Servus Dmitry!"
```

If we didn't specify the `@JsName` annotation, the name of the corresponding function would contain a suffix calculated from the function signature, for example `hello_61zpoe$`.

JavaScript Reflection

In Kotlin compiled to JavaScript, there's a property available on any object called `jsClass` which returns a `JsClass` instance. `JsClass` currently can do nothing more than providing a (non-qualified) name of the class. However, the `JsClass` instance itself is a reference to the constructor function. This can be used to interoperate with JS functions that expect a reference to a constructor.

To get a reference to a class, you can use the `::class` syntax. Full reflection API is currently not supported in Kotlin for JavaScript; the only available properties are `.simpleName` which returns the name of the class and `.js` which returns the corresponding `JsClass`.

Examples:

```
class A
class B
class C

inline fun <reified T> foo() {
    println(jsClass<T>().name)
}

println(A().jsClass.name)     // prints "A"
println(B::class.simpleName)  // prints "B"
println(B::class.js.name)     // prints "B"
foo<C>()                      // prints "C"
```

工具

生成kotlin代码文档

KDoc用来编写Kotlin代码文档（类似于java的 JavaDoc工具）。本质上来说，KDoc 结合了JavaDoc的标签块的句法和Markdown的语法来标记（来扩展Kotlin的特殊标记）。

Generating the Documentation

Kotlin's documentation generation tool is called Dokka. See the Dokka README for usage instructions.

Dokka has plugins for Gradle, Maven and Ant, so you can integrate documentation generation into your build process.

KDoc 语法

像JavaDoc一样，KDoc注释也 `/**` 开头和也 `*/` 结束,每一行注释可能都是也星号开头的， 但是并不作为注释内容的一部分。

按惯例来说，文档的第一段（到第一行空白行结束）是该文档元素的 总体描述，接下来的注释是详细描述

每一个块标记也新一行开始并且也 `@` 字符开头

这是用 KDoc 写类文档的一个例子：

```
/**
 * A group of *members*.
 *
 * This class has no useful logic; it's just a documentation example.
 *
 * @param T the type of a member in this group.
 * @property name the name of this group.
 * @constructor Creates an empty group.
 */
class Group<T>(val name: String) {
    /**
     * Adds a [member] to this group.
     * @return the new size of the group.
     */
    fun add(member: T): Int { ... }
}
```

块标签

KDoc现在支持如下的块标签：

`@param <name>`

代表一个函数的参数值或者一个类、属性或者函数的类型参数。 为了更好的区分描述中的参数值，如果你喜欢，你可以在参数名 括在方括号中，下面是两个符合条件的句法：

```
@param name description.
@param[name] description.
```

`@return`

函数的返回值

`@constructor`

类构造函数

`@receiver`

Documents the receiver of an extension function.

`@property <name>`

Documents the property of a class which has the specified name. This tag can be used for documenting properties declared in the primary constructor, where putting a doc comment directly before the property definition would be awkward.

`@throws <class>` , `@exception <class>`

用来标记一个方法抛出的异常。鉴于Kotlin没有异常检查， 因此不能期待所有可能异常都写出来，但是我们仍然可以使用这个标记来提示给这个类使用这一个 很好的信息。

`@sample <identifier>`

给当前的元素嵌入一个包含特殊名字的方法，为了能够包含例子 来展示这个元素是如何使用的。

`@see <identifier>`

给类或者方法加一个链接来查看 文档的信息

`@author`

文档编写人员的名字

`@since`

来指定什么版本引入了这个方法类

`@suppress`

不包含生成的文档中的元素。可用于不属于官方API的 模块的应用接口，但仍必须对外部可见。

> KDoc 不支持 `@deprecated` 这个标记. 请使用 `@Deprecated` 注释 {:.note}

内置Markup语法

内置Markup语法，KDoc使用了标准的Markdown 语法,来扩展了 它支持在代码中链接到其他元素的速记语法。

链接到元素

为了链接到其它元素（类，方法，属性和参数），把它的元素放在中括号中：

```
Use the method [foo] for this purpose.
```

If you want to specify a custom label for the link, use the Markdown reference-style syntax:

```
Use [this method][foo] for this purpose.
```

您还可以在链接中使用限定名。需要注意的是，不同于javadoc，合格的名字总是使用点字符分开的组件，即使在一个方法前：

```
Use [kotlin.reflect.KClass.properties] to enumerate the properties of the class.
```

如果被使用的元素内的元素被记录，则在链接的名称解析使用相同的规则。 特别是，这意味着，如果您已经导入一个名字到当前文件，在使用KDoc中您不需要完全限定它

注意KDoc在链接中没有解决重载成员的任何语法。自从Kotlin文档生成 工具把上所有的重载函数放在同一个页面之后，标识一个特定的重载函数 不需要链接的方式。

Module and Package Documentation

Documentation for a module as a whole, as well as packages in that module, is provided as a separate Markdown file, and the paths to that file is passed to Dokka using the `-include` command line parameter or the corresponding parameters in Ant, Maven and Gradle plugins.

Inside the file, the documentation for the module as a whole and for individual packages is introduced by the corresponding first-level headings. The text of the heading must be "Module `<module name>` " for the module, and "Package `<package qualified name>` " for a package.

Here's an example content of the file:

```
# Module kotlin-demo

The module shows the Dokka syntax usage.

# Package org.jetbrains.kotlin.demo

Contains assorted useful stuff.

## Level 2 heading

Text after this heading is also part of documentation for `org.jetbrains.kotlin.demo`

# Package org.jetbrains.kotlin.demo2

Useful stuff in another package.
```

使用 Maven

插件与版本

kotlin-maven-plugin 用于编译 Kotlin 源码与模块，当前只支持 Marven V3

通过 *kotlin.version* 指定所要使用的 Kotlin 版本，The correspondence between Kotlin releases and versions is displayed below:

Milestone	Version

依赖

Kotlin 提供了大量的标准库以供开发使用，需要在 pom 文件中设置以下依赖：

```
<dependencies>
    <dependency>
        <groupId>org.jetbrains.kotlin</groupId>
        <artifactId>kotlin-stdlib</artifactId>
        <version>${kotlin.version}</version>
    </dependency>
</dependencies>
```

仅编译 Kotlin 源码

在 <build> 标签中指定所要编译的 Kotlin 源码目录：

```
<build>
    <sourceDirectory>${project.basedir}/src/main/kotlin</sourceDirectory>
    <testSourceDirectory>${project.basedir}/src/test/kotlin</testSourceDirectory>
</build>
```

Maven 中需要引用 Kotlin 插件用于编码源码：

```
<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 源码

To compile mixed code applications Kotlin compiler should be invoked before Java compiler. In maven terms that means kotlin-maven-plugin should be run before maven-compiler-plugin using the following method, making sure that the kotlin plugin is above the maven-compiler-plugin in your pom.xml file.

```
<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>
                <!-- Replacing default-compile as it is treated specially by maven -->
                <execution>
                    <id>default-compile</id>
                    <phase>none</phase>
                </execution>
                <!-- Replacing default-testCompile as it is treated specially by maven
 -->
                <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>
```

Jar file

To create a small Jar file containing just the code from your module, include the following under `build->plugins` in your Maven pom.xml file, where `main.class` is defined as a property and points to the main Kotlin or Java class.

```
<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>
```

Self-contained Jar file

To create a self-contained Jar file containing the code from your module along with dependencies, include the following under `build->plugins` in your Maven pom.xml file, where `main.class` is defined as a property and points to the main Kotlin or Java class.

```
<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>
```

This self-contained jar file can be passed directly to a JRE to run your application:

```
java -jar target/mymodule-0.0.1-SNAPSHOT-jar-with-dependencies.jar
```

OSGi

OSGi支持查看 Kotlin OSGi page.

例子

Maven 工程的例子可从 Github 直接下载

Using Ant

Getting the Ant Tasks

Kotlin provides three tasks for Ant:

- kotlinc: Kotlin compiler targeting the JVM
- kotlin2js: Kotlin compiler targeting JavaScript
- withKotlin: Task to compile Kotlin files when using the standard *javac* Ant task

These tasks are defined in the *kotlin-ant.jar* library which is located in the *lib* folder for the Kotlin Compiler

Targeting JVM with Kotlin-only source

When the project consists of exclusively Kotlin source code, the easiest way to compile the project is to use the *kotlinc* task

```
<project name="Ant Task Test" default="build">
    <typedef resource="org/jetbrains/kotlin/ant/antlib.xml" classpath="${kotlin.lib}/k
otlin-ant.jar"/>

    <target name="build">
        <kotlinc src="hello.kt" output="hello.jar"/>
    </target>
</project>
```

where ${kotlin.lib} points to the folder where the Kotlin standalone compiler was unzipped.

Targeting JVM with Kotlin-only source and multiple roots

If a project consists of multiple source roots, use *src* as elements to define paths

```
<project name="Ant Task Test" default="build">
    <typedef resource="org/jetbrains/kotlin/ant/antlib.xml" classpath="${kotlin.lib}/k
otlin-ant.jar"/>

    <target name="build">
        <kotlinc output="hello.jar">
            <src path="root1"/>
            <src path="root2"/>
        </kotlinc>
    </target>
</project>
```

Targeting JVM with Kotlin and Java source

If a project consists of both Kotlin and Java source code, while it is possible to use *kotlinc*, to avoid repetition of task parameters, it is recommended to use *withKotlin* task

```
<project name="Ant Task Test" default="build">
    <typedef resource="org/jetbrains/kotlin/ant/antlib.xml" classpath="${kotlin.lib}/k
otlin-ant.jar"/>

    <target name="build">
        <delete dir="classes" failonerror="false"/>
        <mkdir dir="classes"/>
        <javac destdir="classes" includeAntRuntime="false" srcdir="src">
            <withKotlin/>
        </javac>
        <jar destfile="hello.jar">
            <fileset dir="classes"/>
        </jar>
    </target>
</project>
```

To specify additional command line arguments for `<withKotlin>`, you can use a nested `<compilerArg>` parameter. The full list of arguments that can be used is shown when you run `kotlinc -help`. You can also specify the name of the module being compiled as the `moduleName` attribute:

```
<withKotlin moduleName="myModule">
    <compilerarg value="-no-stdlib"/>
</withKotlin>
```

Targeting JavaScript with single source folder

```
<project name="Ant Task Test" default="build">
    <typedef resource="org/jetbrains/kotlin/ant/antlib.xml" classpath="${kotlin.lib}/k
otlin-ant.jar"/>

    <target name="build">
        <kotlin2js src="root1" output="out.js"/>
    </target>
</project>
```

Targeting JavaScript with Prefix, PostFix and sourcemap options

```
<project name="Ant Task Test" default="build">
    <taskdef resource="org/jetbrains/kotlin/ant/antlib.xml" classpath="${kotlin.lib}/k
otlin-ant.jar"/>

    <target name="build">
        <kotlin2js src="root1" output="out.js" outputPrefix="prefix" outputPostfix="po
stfix" sourcemap="true"/>
    </target>
</project>
```

Targeting JavaScript with single source folder and metaInfo option

The `metaInfo` option is useful, if you want to distribute the result of translation as a Kotlin/JavaScript library. If `metaInfo` was set to `true`, then during compilation additional JS file with binary metadata will be created. This file should be distributed together with the result of translation.

```
<project name="Ant Task Test" default="build">
    <typedef resource="org/jetbrains/kotlin/ant/antlib.xml" classpath="${kotlin.lib}/k
otlin-ant.jar"/>

    <target name="build">
        <!-- out.meta.js will be created, which contains binary descriptors -->
        <kotlin2js src="root1" output="out.js" metaInfo="true"/>
    </target>
</project>
```

References

Complete list of elements and attributes are listed below

Attributes common for kotlinc and kotlin2js

Name	Description	Required	Default Value
`src`	Kotlin source file or directory to compile	Yes	
`nowarn`	Suppresses all compilation warnings	No	false
`noStdlib`	Does not include the Kotlin standard library into the classpath	No	false
`failOnError`	Fails the build if errors are detected during the compilation	No	true

kotlinc Attributes

Name	Description	Required	Default Value
`output`	Destination directory or .jar file name	Yes	
`classpath`	Compilation class path	No	
`classpathref`	Compilation class path reference	No	
`includeRuntime`	If `output` is a .jar file, whether Kotlin runtime library is included in the jar	No	true
`moduleName`	Name of the module being compiled	No	The name of the target (if specified) or the project

kotlin2js Attributes

Name	Description	Required
`output`	Destination file	Yes
`library`	Library files (kt, dir, jar)	No
`outputPrefix`	Prefix to use for generated JavaScript files	No
`outputSuffix`	Suffix to use for generated JavaScript files	No
`sourcemap`	Whether sourcemap file should be generated	No
`metaInfo`	Whether metadata file with binary descriptors should be generated	No
`main`	Should compiler generated code call the main function	No

使用 Gradle

In order to build Kotlin with Gradle you should set up the *kotlin-gradle* plugin, apply it to your project and add *kotlin-stdlib* dependencies. Those actions may also be performed automatically in IntelliJ IDEA by invoking the Tools | Kotlin | Configure Kotlin in Project action.

You can also enable incremental compilation to make your builds faster.

插件和版本

使用 *kotlin-gradle-plugin* 编译Kotlin的源代码和模块.

要用的 Kotlin 版本通常是通过 *kotlin.version*属性来定义:

```
buildscript {
    ext.kotlin_version = '<version to use>'

    repositories {
      mavenCentral()
    }

    dependencies {
      classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin_version"
    }
}
```

Targeting the JVM

应用于JVM

为了在JVM中应用, Kotlin插件需要配置如下

```
apply plugin: "kotlin"
```

Kotlin源文件和Java源文件可以在同一个文件夹中存在, 也可以在不同文件夹中. 默认采用的是不同的文件夹:

```
project
    - src
        - main (root)
            - kotlin
            - java
```

如果不想使用默认选项，你需要更新对应的 *sourceSets* 属性

```
sourceSets {
    main.kotlin.srcDirs += 'src/main/myKotlin'
    main.java.srcDirs += 'src/main/myJava'
}
```

应用于 JavaScript

当应用于 JavaScript 的时候, 需要设置一个不同的插件:

```
apply plugin: "kotlin2js"
```

该插件仅作用于Kotlin文件，因此推荐使用这个插件来区分Kotlin和Java文件 (这种情况仅仅是同一工程中包含Java源文件的时候). 如果 不使用默认选项，又为了应用于 JVM，我们需要指定源文件夹使用 *sourceSets*

```
sourceSets {
    main.kotlin.srcDirs += 'src/main/myKotlin'
}
```

如果你想创建一个可重用的库, 使用 `kotlinOptions.metaInfo` 来生成额外的二进制形式的JS文件. 这个文件应该和编译结果一起分发.

```
compileKotlin2Js {
    kotlinOptions.metaInfo = true
}
```

应用于 Android

Android的 Gradle模型和传统的Gradle有些不同, 因此如果我们想要通过Kotlin来创建一个Android应用, 应该使用 *kotlin-android* 插件来代替 *kotlin*:

```
buildscript {
    ...
}
apply plugin: 'com.android.application'
apply plugin: 'kotlin-android'
```

Android Studio

如果你使用的是Android Studio, 下面的一些属性需要添加到文件中:

```
android {
  ...

  sourceSets {
    main.java.srcDirs += 'src/main/kotlin'
  }
}
```

上述属性可以使kotlin目录在Android Studio中作为源码根目录存在, 所以当项目模型加载到IDE可以被正确识别. Alternatively, you can put Kotlin classes in the Java source directory, typically located in `src/main/java`.

配置依赖

In addition to the kotlin-gradle-plugin dependency shown above, you need to add a dependency on the Kotlin standard library:

```
buildscript {
   ext.kotlin_version = '<version to use>'
  repositories {
    mavenCentral()
  }
  dependencies {
    classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin_version"
  }
}

apply plugin: "kotlin" // or apply plugin: "kotlin2js" if targeting JavaScript

repositories {
  mavenCentral()
}

dependencies {
  compile "org.jetbrains.kotlin:kotlin-stdlib:$kotlin_version"
}
```

If your project uses Kotlin reflection or testing facilities, you need to add the corresponding dependencies as well:

```
compile "org.jetbrains.kotlin:kotlin-reflect:$kotlin_version"
testCompile "org.jetbrains.kotlin:kotlin-test:$kotlin_version"
testCompile "org.jetbrains.kotlin:kotlin-test-junit:$kotlin_version"
```

Annotation processing

The Kotlin plugin supports annotation processors like *Dagger* or *DBFlow*. In order for them to work with Kotlin classes, add the respective dependencies using the `kapt` configuration in your `dependencies` block:

```
dependencies {
  kapt 'groupId:artifactId:version'
}
```

If you previously used the android-apt plugin, remove it from your `build.gradle` file and replace usages of the `apt` configuration with `kapt`. If your project contains Java classes, `kapt` will also take care of them. If you use annotation processors for your `androidTest` or `test` sources, the respective `kapt` configurations are named `kaptAndroidTest` and `kaptTest`.

Some annotation processing libraries require you to reference generated classes from within your code. For this to work, you'll need to add an additional flag to enable the *generation of stubs* to your build file:

```
kapt {
    generateStubs = true
}
```

Note, that generation of stubs slows down your build somewhat, which is why it's disabled by default. If generated classes are referenced only in a few places in your code, you can alternatively revert to using a helper class written in Java which can be seamlessly called from your Kotlin code.

For more information on `kapt` refer to the official blogpost.

Incremental compilation

Kotlin 1.0.2 introduced new experimental incremental compilation mode in Gradle. Incremental compilation tracks changes of source files between builds so only files affected by these changes would be compiled.

There are several ways to enable it:

1. add `kotlin.incremental=true` line either to a `gradle.properties` or a `local.properties` file;
2. add `-Pkotlin.incremental=true` to gradle command line parameters. Note that in this case the parameter should be added to each subsequent build (any build without this parameter invalidates incremental caches).

After incremental compilation is enabled, you should see the following warning message in your build log:

```
Using experimental kotlin incremental compilation
```

Note, that the first build won't be incremental.

OSGi

OSGi 支持查看 Kotlin OSGi page.

例子

Kotlin Repository 包含的例子:

- Kotlin
- Mixed Java and Kotlin
- Android
- JavaScript

Kotlin and OSGi

To enable Kotlin OSGi support you need to include `kotlin-osgi-bundle` instead of regular Kotlin libraries. It is recommended to remove `kotlin-runtime`, `kotlin-stdlib` and `kotlin-reflect` dependencies as `kotlin-osgi-bundle` already contains all of them. You also should pay attention in case when external Kotlin libraries are included. Most regular Kotlin dependencies are not OSGi-ready, so you shouldn't use them and should remove them from your project.

Maven

To include the Kotlin OSGi bundle to a Maven project:

```
    <dependencies>
         <dependency>
             <groupId>org.jetbrains.kotlin</groupId>
             <artifactId>kotlin-osgi-bundle</artifactId>
             <version>${kotlin.version}</version>
         </dependency>
     </dependencies>
```

To exclude the standard library from external libraries (notice that "star exclusion" works in Maven 3 only)

```
         <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

To include `kotlin-osgi-bundle` to a gradle project:

```
compile "org.jetbrains.kotlin:kotlin-osgi-bundle:$kotlinVersion"
```

To exclude default Kotlin libraries that comes as transitive dependencies you can use the following approach

```
dependencies {
 compile (
   [group: 'some.group.id', name: 'some.library', version: 'someversion'],
   .....) {
  exclude group: 'org.jetbrains.kotlin'
}
```

FAQ

Why not just add required manifest options to all Kotlin libraries

Even though it is the most preferred way to provide OSGi support, unfortunately it couldn't be done for now due to so called "package split" issue that couldn't be easily eliminated and such a big change is not planned for now. There is `Require-Bundle` feature but it is not the best option too and not recommended to use. So it was decided to make a separate artifact for OSGi.

常见问题

你想到的问题可能会在这里得到解答。

FAQ

常见问题

Kotlin是什么？

Kotlin 是目标平台为 JVM 和 JavaScript 的静态类型语言。它是一种旨在工业级使用的通用语言。

它是由 JetBrains 一个团队开发的，然而它是开源（OSS）语言并且也有外部贡献者。

为什么要出一门新语言？

在 JetBrains 我们已经在 Java 平台开发很长时间，并且我们知道它（Java）有多好。 另一方面，我们知道由于向后兼容性问题 Java 编程语言有一定的局限性和问题是不可能 或者很难解决的。我们知道 Java 还会延续很长时间， 但我们相信社区会从这个新的静态类型 JVM 平台语言中受益，它没有 遗留问题而有开发人员迫切想要特性。

Kotlin 这样设计背后的核心价值是使其

- 可互操作：Kotlin 可以与 Java 自由混搭，
- 安全：静态检查常见的陷阱（如：解引用空指针）来在编译期捕获错误，
- 可工具化：启用像 IDE、构建系统这样精确而高效的工具，
- “民主”：使语言的全部可供所有开发者使用（无需限制库的作者或者其他开发组使用一些功能的策略）。

如何授权？

Kotlin 是一种开源语言并在 Apache 2 开源软件许可下授权。它的 IntelliJ 插件也是开源软件。

它托管在 Github 上并且我们很乐意接受贡献者。

哪里可以获取 Kotlin 的高清徽标？

徽标可以在这里下载。请遵循压缩包内的 `readme.txt` 中的简单规则使用。

它兼容Java？

兼容。编译器生成的是 Java 字节码。Kotlin 可以调用 Java 并且 Java 也可以调用 Kotlin。参见与 Java 互操作性。

运行Kotlin代码所需的最低Java版本是哪个？

Kotlin 生成的字节码兼容 Java 6 以及更新版本。这确保 Kotlin 可以在像 Android 这样上一个所支持版本是 Java 6 的环境中使用。

有没有工具支持？

有。有一个作为 Apache 2 许可下开源项目的 IntelliJ IDEA 插件可用。 在 自由开源社区版和旗舰版的 IntelliJ IDEA 中都可以使用 Kotlin。

有没有Eclipse支持？

有。安装说明请参见这个教程。

有独立的编译器吗？

有。你可以从 Github 上的发布页下载独立的编译器和其他构建工具。

Kotlin是函数式语言吗？

Kotlin 是一种面向对象语言。不过它支持高阶函数以及 lambda 表达式和顶层函数。此外，在 Kotlin 标准库中还有很多一般函数式语言的设计（例如 map、flatMap、reduce 等）。当然，什么是函数式语言没有明确的定义，所以我们不能说 Kotlin 是其中之一。

Kotlin支持泛型吗？

Kotlin 支持泛型。它也支持声明处型变和使用处型变。Kotlin 没有通配符类型。内联函数支持具体化的类型参数。

分号是必需的吗？

不是。它们是可选的。

为什么类型声明在右侧？

我们相信这会使代码更易读。此外它启用了一些很好的语法特性，例如，很容易脱离类型注解。Scala 也已 很好地证明了这没有问题。

右侧类型声明会影响工具吗？

不会。我们仍然可以实现对变量名的建议等等。

Kotlin是可扩展的吗？

我们计划使其在这几个方面可扩展：从内联函数到注解和类型加载器。

我可以把我的DSL嵌入到语言里吗？

可以。Kotlin 提供了一些有助于此的特性：操作符重载、通过内联函数自定义控制结构、中缀函数调用、扩展函数、注解。

Kotlin for JavaScript 支持到 ECMAScript的什么水平？

目前到 5。

JavaScript后端支持模块系统吗？

支持。至少有提供 CommonJS 和 AMD 支持的计划。

与 Java 比较

Kotlin 解决了一些 Java 中的问题

Kotlin 通过以下措施修复了 Java 中一系列长期困扰我们的问题

- 空引用由类型系统控制。
- 无原始类型
- Kotlin 中数组是不协变的
- 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

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
- 变量和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
- 分离用于只读和可变集合的接口

与 Scala 比较

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
  - 参见类和接口
- 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
- Actors
  - Kotlin 支持Quasar——一个用于JVM上的actor支持的第三方框架
- 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，它是一个语法糖和运行时的包装器
- 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