



Kotlin 语言文档

目录

开始	4
基本语法	4
习惯用法	9
编码规范	13
Kotlin 1.1 的新特性	14
基础	21
基本类型	21
包	25
控制流	26
返回和跳转	29
类和对象	31
类和继承	31
属性和字段	36
接口	39
可见性修饰符	41
扩展	43
数据类	47
泛型	48
嵌套类	53
枚举类	54
对象表达式和对象声明	55
委托	58
委托属性	59
函数和 Lambda 表达式	62
函数	62
高阶函数和 lambda 表达式	67
内联函数	71
其他	74
解构声明	74
集合	76
区间	77
类型的检查与转换	79
This 表达式	81
相等性	82
操作符重载	83
空安全	86
异常	88
注解	90

反射	94
类型安全的构建器	97
参考	101
Grammar	101
Notation	101
Semicolons	101
Syntax	101
Lexical structure	109
Java 互操作	110
在 Kotlin 中调用 Java 代码	110
Java 调用 Kotlin	116
JavaScript	122
动态类型	122
JavaScript 互操作性	123
JavaScript 反射	125
工具	126
生成kotlin代码文档	126
使用 Maven	129
Using Ant	132
使用 Gradle	135
Kotlin 与 OSGi	139
常见问题	140
FAQ	140
与 Java 比较	142
与 Scala 比较	143

开始

基本语法

定义包

包的声明应处于源文件顶部：

```
package my.demo

import java.util.*

// ...
```

目录与包的结构无需匹配：源代码可以在文件系统的任意位置。

参见[包](#)。

定义函数

带有两个 `Int` 参数、返回 `Int` 的函数：

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

将表达式作为函数体、返回值类型自动推断的函数：

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

函数返回无意义的值：

```
fun printSum(a: Int, b: Int): Unit {
    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(参见[数据类](#))

函数的默认参数

```
fun foo(a: Int = 0, b: String = "") { ... }
```

过滤 list

```
val positives = list.filter { x -> x > 0 }
```

或者可以更短:

```
val positives = list.filter { it > 0 }
```

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 JsonSyntaxException {
//         ...
//     }
// }

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

Kotlin 1.1 的新特性

Kotlin 1.1 目前[提供测试版](#)。在这里你可以找到该版本中提供的新功能列表。请注意,任何新功能都可能会在 Kotlin 1.1 发布之前更改。

JavaScript

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

协程 (实验性的)

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

协程实际上是一个轻量级的线程,可以暂停并稍后恢复。协程由协程构建器函数启动、并使用特殊挂起函数挂起。例如, `future` 启动一个协程,当你使用 `await` 时,挂起协程的执行而执行等待的操作,并且当等待的操作完成时恢复该协程的执行(可能在不同的线程上)。

标准库通过 `yield` 和 `yieldAll` 函数使用协程来支持惰性生成序列。在这样的序列中,在取回每个元素之后暂停返回序列元素的代码块,并在请求下一个元素时恢复。这里有一个例子:

```
val seq = buildSequence {
    println("Yielding 1")
    yield(1)
    println("Yielding 2")
    yield(2)
    println("Yielding a range")
    yieldAll(3..5)
}

for (i in seq) {
    println("Generated $i")
}
```

这将输出:

```
Yielding 1
Generated 1
Yielding 2
Generated 2
Yielding a range
Generated 3
Generated 4
Generated 5
```

`future/await` 的实现是作为外部库提供的 ([kotlinx.coroutines](#))。这里有一个显示其用法的例子:

```
future {
    val original = asyncLoadImage("...original...") // 创建一个 Future
    val overlay = asyncLoadImage("...overlay...")   // 创建一个 Future
    ...
    // 当等待两图片加载是挂起
    // 当他们都加载完成后运行 `applyOverlay(...)`
    return applyOverlay(original.await(), overlay.await())
}
```

`kotlinx.coroutines` 的 `future` 实现依赖于 `CompletableFuture`,因此需要 JDK 8,但它也提供了可移植的 `defer` 原语,并且可以构建其他实现。

其 [KEEP 文档](#)提供了 协程功能性的一个扩展描述。

请注意,协程目前还是一个实验性的功能,这意味着 Kotlin 团队不承诺在最终的 1.1 版本时保持该功能的向后兼容性。

其他语言功能

类型别名

类型别名允许你为现有类型定义备用名称。这对于泛型类型(如集合)以及函数类型最有用。这里有几个例子:

```
typealias FileTable<K> = MutableMap<K, MutableList<File>>

typealias MouseEventHandler = (Any, MouseEvent) -> Unit
```

更多详细信息请参阅其 [KEEP](#)。

已绑定的可调用引用

现在可以使用 `::` 操作符来获取指向特定对象实例的方法或属性的[成员引用](#)。以前这只能用 lambda 表达式表示。这里有一个例子：

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

更多详细信息请参阅其 [KEEP](#)。

密封类和数据类

Kotlin 1.1 删除了一些对 Kotlin 1.0 中已存在的密封类和数据类的限制。现在你可以在同一个文件中的任何地方定义一个密封类的子类，而不只是以作为密封类嵌套类的方式。数据类现在可以扩展其他类。这可以用来友好且清晰地定义一个表达式类的层次结构：

```
sealed class Expr

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

fun eval(expr: Expr): Double = when (expr) {
    is Const -> expr.number
    is Sum -> eval(expr.e1) + eval(expr.e2)
    NotANumber -> Double.NaN
}
```

更多详细信息请参阅[密封类](#) 及 [数据类](#)的 KEEP。

lambda 表达式中的解构

现在可以使用[解构声明](#)语法来解开传递给 lambda 表达式的参数。这里有一个例子：

```
map.mapValues { (key, value) -> "$value!" }
```

更多详细信息请参阅其 [KEEP](#)。

下划线用于未使用参数

对于具有多个参数的 lambda 表达式，可以使用 `_` 字符替换不使用的参数的名称：

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

这也适用于[解构声明](#)：

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

更多详细信息请参阅其 [KEEP](#)。

数字字面值中的下划线

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

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

更多详细信息请参阅其 [KEEP](#)。

对于属性的更短语法

对于没有自定义访问器、或者将 `getter` 定义为表达式主体的属性,现在可以省略属性的类型:

```
val name = ""

val lazyName get() = ""
```

对于这两个属性,编译器会推断其属性类型是字符串。

内联属性访问器

如果属性没有幕后字段,现在可以使用 `inline` 修饰符来标记该属性访问器。这些访问器的编译方式与[内联函数](#)相同。

```
val foo: Foo
    inline get() = Foo()
```

更多详细信息请参阅其 [KEEP](#)。

局部委托属性

现在可以对局部变量使用[委托属性](#)语法。一个可能的用途是定义一个延迟求值的局部变量:

```
fun foo() {
    val data: String by lazy { /* 计算该数据 */ }
    if (needData()) {
        println(data) // 数据在此处计算
    }
}
```

更多详细信息请参阅其 [KEEP](#)。

委托属性绑定的拦截

对于[委托属性](#),现在可以使用 `provideDelegate` 操作符拦截委托到属性的绑定。例如,如果我们想要在绑定之前检查属性名称,我们可以这样写:

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

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

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

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

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

泛型枚举值访问

现在可以用泛型的方式来枚举枚举类的值:

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

print(enumValues<RGB>().joinToString { it.name }) // 输出 RED, GREEN, BLUE
```


对于 DSL 中隐式接收者的作用域控制

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

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

在 Kotlin 1.0 中, 传递给 `td` 的 lambda 表达式中的代码可以访问三个隐式接收者: 传递给 `table`、`tr` 和 `td` 的。这允许你调用在上下文中没有意义的方法——例如在 `td` 里面调用 `tr`, 从而在 `<td>` 中放置一个 `<tr>` 标签。

在 Kotlin 1.1 中, 你可以限制这种情况, 以使只有在 `td` 的隐式接收者上定义的方法 会在传给 `td` 的 lambda 表达式中可用。你可以通过定义标记有 `@DslMarker` 元注解的注解 并将其应用于标记类的基类：

```
@DslMarker
annotation class HtmlTagMarker

@HtmlTagMarker
abstract class Tag(val name: String) { ... }

class TD() : Tag("td") { ... }

fun Tag.td(init: TD.() -> Unit) {
}
```

现在, 传递给 `td` 函数的 lambda 表达式 `init` 的隐式接收者是一个用 `@HtmlTagMarker` 注解过的类, 因此也具有此注解的類型的外部接收者会被阻止。

更多详细信息请参阅其 [KEEP](#)。

rem 操作符

`mod` 操作符现已弃用, 而使用 `rem` 取代。动机参见[这个问题](#)。

标准库

字符串到数字的转换

在 `String` 类中有一些新的扩展, 用来将它转换为数字, 而不会在无效数字上抛出异常: `String.toIntOrNull(): Int?`、`String.toDoubleOrNull(): Double?` 等。

```
val port = System.getenv("PORT")?.toIntOrNull() ?: 80
```

还有整数转换函数, 如 `Int.toString()`、`String.toInt()`、`String.toIntOrNull()`, 每个都有一个带有 `radix` 参数的重载, 它允许指定转换的基数 (2 到 36)。

onEach()

`onEach` 是一个小、但对于集合和序列很有用的扩展函数, 它允许对操作链中的集合/序列的每个元素执行一些操作, 可能带有副作用。对于迭代其行为像 `forEach` 但是也进一步返回可迭代实例。对于序列它返回一个包装序列, 它在元素迭代时延迟应用给定的动作。

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

takeIf()、takeUnless() 和 also()

这些是适用于任何接收者的三个通用扩展函数。

`also` 就像 `apply`: 它接受接收者、做一些动作、并返回该接收者。二者区别是在 `apply` 内部的代码块中接收者是 `this`, 而在 `also` 内部的代码块中是 `it` (并且如果你想的话, 你可以给它另一个名字)。当你不想掩盖来自外部作用域的 `this` 时这很方便:

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

`takeIf` 就像单个值的 `filter`。它检查接收者是否满足该谓词,并在满足时返回该接收者否则不满足时返回 `null`。结合 `elvis`-操作符和及早返回,它允许编写如下结构:

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

val index = input.indexOf(keyword).takeIf { it >= 0 } ?: error("keyword not found")
// 对输入字符串中的关键字索引做些事情,假定它找到
```

`takeUnless` 与 `takeIf` 相同,只是它采用了反向谓词。当它 不满足谓词时返回接收者,否则返回 `null`。因此,上面的示例之一可以用 `takeUnless` 重写如下:

```
val index = input.indexOf(keyword).takeUnless { it < 0 } ?: error("keyword not found")
```

当你有一个可调用的引用而不是 `lambda` 时,使用也很方便:

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

groupingBy()

此 API 可以用于按照键对集合进行分组,并同时折叠每个组。例如,它可以用于 计算文本中字符的频率:

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

Map.toMap() 和 Map.toMutableMap()

这两函数可以用来简易复制映射:

```
class ImmutablePropertyBag(map: Map<String, Any>) {
    private val mapCopy = map.toMap()
}
```

Map.minus(key)

运算符 `plus` 提供了一种将键值对添加到只读映射中以生成新映射的方法,但是没有一种简单的方法来做相反的操作:从映射中删除一个键采用不那么直接的方式如 `Map.filter()` 或 `Map.filterKeys()`。现在运算符 `minus` 填补了这个空白。有 4 个可用的重载:用于删除单个键、键的集合、键的序列和键的数组。

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

minOf() 和 maxOf()

这些函数可用于查找两个或三个给定值中的最小和最大值,其中值是原生数字或 `Comparable` 对象。每个函数还有一个重载,它接受一个额外的 `Comparator` 实例,如果你想比较自身不可比的对象的话。

```
val list1 = listOf("a", "b")
val list2 = listOf("x", "y", "z")
val minSize = minOf(list1.size, list2.size)
val longestList = maxOf(list1, list2, compareBy { it.size })
```

类似数组的列表实例化函数

类似于 `Array` 构造函数,现在有创建 `List` 和 `MutableList` 实例的函数,并通过 调用 `lambda` 表达式来初始化每个元素:

```
List(size) { index -> element }
MutableList(size) { index -> element }
```

Map.getValue()

Map 上的这个扩展函数返回一个与给定键相对应的现有值,或者抛出一个异常,提示找不到该键。如果该映射是用 withDefault 生成的,这个函数将返回默认值,而不是抛异常。

```
val map = mapOf("key" to 42)
// 返回不可空 Int 值 42
val value: Int = map.getValue("key")
// 抛出 NoSuchElementException
map.getValue("key2")

val mapWithDefault = map.withDefault { k -> k.length }
// 返回 4
val value2 = mapWithDefault.getValue("key2")
```

抽象集合

这些抽象类可以在实现 Kotlin 集合类时用作基类。对于实现只读集合,有 AbstractCollection、AbstractList、AbstractSet 和 AbstractMap,而对于可变集合,有 AbstractMutableCollection、AbstractMutableList、AbstractMutableSet 和 AbstractMutableMap。在 JVM 上,这些抽象可变集合从 JDK 的抽象集合继承了大部分的功能。

JVM 后端

Java 8 字节码支持

Kotlin 现在可以选择生成 Java 8 字节码(命令行选项 -jvm-target 1.8 或者 Ant/Maven/Gradle 中的相应选项)。目前这并不改变字节码的语义(特别是,接口和 lambda 表达式中的默认方法的生成与 Kotlin 1.0 中完全一样),但我们计划在以后进一步使用它。

Java 8 标准库支持

现在有支持在 Java 7 和 8 中新添加的 JDK API 的标准库的独立版本。如果你需要访问新的 API,请使用 kotlin-stdlib-jre7 和 kotlin-stdlib-jre8 maven artifacts,而不是标准的 kotlin-stdlib。这些构件是在 kotlin-stdlib 之上的微小扩展,它们将它作为传递依赖项带到项目中。

字节码中的参数名

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

常量内联

编译器现在将 const val 属性的值内联到使用它们的位置。

可变闭包变量

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

javax.scripting 支持

Kotlin 现在与 [javax.script API](#) (JSR-223) 集成。关于使用 API 的示例项目参见[这里](#)。

JavaScript 后端

统一的标准库

Kotlin 标准库的大部分目前可以从代码编译成 JavaScript 来使用。特别是,关键类如集合 (ArrayList、HashMap 等)、异常 (IllegalArgumentException 等) 以及其他几个关键类 (StringBuilder、Comparator) 现在都定义在 kotlin 包下。在 JVM 平台上,一些名称是相应 JDK 类的类型别名,而在 JS 平台上,这些类在 Kotlin 标准库中实现。

更好的代码生成

JavaScript 后端现在生成更加可静态检查的代码,这对 JS 代码处理工具(如 minifiers、optimisers、linters 等)更加友好。

external 修饰符

如果你需要以类型安全的方式从 Kotlin 访问 JavaScript 实现的类, 你可以使用 `external` 修饰符写一个 Kotlin 声明。(在 Kotlin 1.0 中, 使用了 `@native` 注解。) 与 JVM 目标平台不同, JS 平台允许对类和属性使用外部修饰符。例如, 可以按以下方式声明 DOM `Node` 类:

```
external class Node {
    val firstChild: Node

    fun appendChild(child: Node): Node

    fun removeChild(child: Node): Node

    // 等等
}
```

改进的导入处理

现在可以更精确地描述应该从 JavaScript 模块导入的声明。如果在外部声明上添加 `@JsModule("<module-name>")` 注解, 它会在编译期间正确导入到模块系统 (CommonJS或AMD)。例如, 使用 CommonJS, 该声明会通过 `require(...)` 函数导入。此外, 如果要声明作为模块或全局 JavaScript 对象导入, 可以使用 `@JsNonModule` 注解。

例如, 以下是将 JQuery 导入 Kotlin 模块的方法:

```
@JsNonModule
@JsName("$")
external abstract class JQuery {
    fun toggle(duration: Int = 0): JQuery
    fun click(handler: (Event) -> Unit): JQuery
}

@JsModule("jquery")
@JsNonModule
@JsName("$")
external fun JQuery(selector: String): JQuery
```

在这种情况下, JQuery 将作为名为 `jquery` 的模块导入。或者, 它可以用作 `$`-对象, 这取决于 Kotlin 编译器配置使用哪个模块系统。

你可以在应用程序中使用如下所示的这些声明:

```
fun main(args: Array<String>) {
    JQuery(".toggle-button").click {
        JQuery(".toggle-panel").toggle(300)
    }
}
```

基础

基本类型

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
        // ...
    }
}
```

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

Kotlin 也有无装箱开销的专门的类来表示原生类型数组: `ByteArray`、`ShortArray`、`IntArray` 等等。这些类和 `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)
}
```

字符串面值

Kotlin 有两种类型的字符串面值: 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)  
    })  
}
```

当要返回一个返回值的时候,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

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 = "")
```

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。**

覆盖方法

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() {}
}
```

覆盖属性

属性覆盖与方法覆盖类似;在超类中声明然后在派生类中重新声明的属性必须以 **override** 开头,并且它们必须具有兼容的类型。每个声明的属性可以由具有初始化器的属性或者具有 `getter` 方法的属性覆盖。

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

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

你也可以用一个 `var` 属性覆盖一个 `val` 属性,但反之则不行。这是允许的,因为一个 `val` 属性本质上声明了一个 `getter` 方法,而将其覆盖为 `var` 只是在子类中额外声明一个 `setter` 方法。

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

```
interface Foo {
    val count: Int
}

class Bar1(override val count: Int) : Foo

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

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[覆盖属性](#)

委托属性

最常见的一类属性就是简单地从幕后字段中读取(以及可能的写入)。另一方面,使用自定义 getter 和 setter 可以实现属性的任何行为。介于两者之间,属性如何工作有一些常见的模式。一些例子:惰性值、通过键值从映射读取、访问数据库、访问时通知侦听器等等。

这些常见行为可以通过使用[委托属性](#)实现为库。

接口

Kotlin 的接口与 Java 8 类似,既包含抽象方法的声明,也包含 实现。与抽象类不同的是,接口无法保存状态。它可以有 属性但必须声明为抽象或提供访问器实现。

使用关键字 `interface` 来定义接口

```
interface MyInterface {  
    fun bar()  
    fun foo() {  
        // 可选的方法体  
    }  
}
```

实现接口

一个类或者对象可以实现一个或多个接口。

```
class Child : MyInterface {  
    override fun bar() {  
        // 方法体  
    }  
}
```

接口中的属性

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

```
interface MyInterface {  
    val prop: Int // 抽象的  
  
    val propertyWithImplementation: String  
    get() = "foo"  
  
    fun foo() {  
        print(prop)  
    }  
}  
  
class Child : MyInterface {  
    override val prop: Int = 29  
}
```

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`。

下面将根据声明作用域的不同来解释。

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

对于类内部声明的成员:

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

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

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

例子:

```

open class Outer {
    private val a = 1
    protected open val b = 2
    internal val c = 3
    val d = 4 // 默认 public

    protected class Nested {
        public val e: Int = 5
    }
}

class Subclass : Outer() {
    // a 不可见
    // b、c、d 可见
    // Nested 和 e 可见

    override val b = 5 // 'b' 为 protected
}

class Unrelated(o: Outer) {
    // o.a、o.b 不可见
    // o.c 和 o.d 可见 (相同模块)
    // Outer.Nested 不可见, Nested::e 也不可见
}

```

构造函数

要指定一个类的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型的变量用点表达式去调用这个新函数。

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

```
open class C

class D: C()

fun C.foo() = "c"

fun D.foo() = "d"

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

printFoo(D())
```

这个例子会输出 "c",因为调用的扩展函数只取决于 参数 `c` 的声明类型,该类型是 `C` 类。

如果一个类定义有一个成员函数和一个扩展函数,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()
```

扩展的作用域

大多数时候我们在顶层定义扩展, 即直接在包里:

```
package foo.bar  
  
fun Baz.foo() { ... }
```

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)), Collections.max(list))

```

这些类名总是碍手碍脚的,我们可以通过静态导入达到这样效果:

```

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

```

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

```

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

```

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

数据类

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

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

编译器自动从主构造函数中声明的所有属性导出以下成员:

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

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**不型变的**,这意味着 `List<String>` **并不是** `List<Object>` 的子类型。为什么这样? 如果 List 不是**不型变的**,它就没比 Java 的数组好到哪去,因为如下代码会通过编译然后导致运行时异常:

```
// Java  
List<String> strs = new ArrayList<String>();  
List<Object> objs = strs; // !!!即将来临的问题的原因就在这里。Java 禁止这样!  
objs.add(1); // 这里我们把一个整数放入一个字符串列表  
String s = strs.get(0); // !!! ClassCastException:无法将整数转换为字符串
```

因此,Java 禁止这样的事情以保证运行时的安全。但这样会有一些影响。例如,考虑 `Collection` 接口中的 `addAll()` 方法。该方法的签名应该是什么? 直觉上,我们会这样:

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

但随后,我们将无法做到以下简单的事情(这是完全安全):

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

(在 Java 中,我们艰难地学到了这个教训,参见 [Effective Java](#), 第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-Extends,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解:**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**不型变的**,因此 `Array <Int>` 和 `Array <Any>` 都不是另一个的子类型。为什么? 再次重复,因为 `copy` **可能**做坏事,也就是说,例如它可能尝试写一个 `String` 到 `from`,并且如果我们实际上传递一个 `Int` 的数组,一段时间后将会抛出一个 `ClassCastException` 异常。

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

```

fun copy(from: Array<out Any>, to: Array<Any>) {
    // ...
}

```

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

你也可以使用 **in** 投影一个类型:

```

fun fill(dest: Array<in String>, value: String) {
    // ...
}

```

`Array<in String>` 对应于 Java 的 `Array<? super String>`,也就是说,你可以传递一个 `CharSequence` 数组或一个 `Object` 数组给 `fill()` 函数。

星投影

有时你想说,你对类型参数一无所知,但仍然希望以安全的方式使用它。这里的安全方式是定义泛型类型的这种投影,该泛型类型的每个具体实例化将是该投影的子类型。

Kotlin 为此提供了所谓的**星投影**语法:

- 对于 `Foo <out T>`,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 DataManager {
    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：

```
DataManager.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 lambda 表达式并记录结果, 后续调用 `get()` 只是返回记录的结果。

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

fun main(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

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，并且你传递一个 lambda 表达式作为相应的参数，你可以在圆括号之外指定它：

```
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，如果函数参数值只有一个参数，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“函数数字值”，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，即函数数字值。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。如果推断出的该 lambda 的返回类型不是 `Unit`，那么该 lambda 主体中的最后一个（或可能是单个）表达式会视为返回值。

如果我们把所有可选标注都留下，看起来如下：

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

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
```

我们可以使用[限定的返回](#)语法从 lambda 显式返回一个值。否则，将隐式返回最后一个表达式的值。因此，以下两个片段是等价的：

```
ints.filter {
    val shouldFilter = it > 0
    shouldFilter
}

ints.filter {
    val shouldFilter = it > 0
    return@filter shouldFilter
}
```

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*,即在外围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)
```

带接收者的函数数字面值

Kotlin 提供了使用指定的 *接收者对象* 调用函数数字面值的功能。在函数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 lambda 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

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 做事情
}
```

为使其能用, 我们应该

- 通过提供一个 `iterator()` 函数将映射表示为一个值的序列,
- 通过提供函数 `component1()` 和 `component2()` 来将每个元素呈现为一对。

当然事实上, 标准库提供了这样的扩展:

```
operator fun <K, V> Map<K, V>.iterator(): Iterator<Map.Entry<K, V>> = entrySet().iterator()
operator fun <K, V> Map.Entry<K, V>.component1() = getKey()
operator fun <K, V> Map.Entry<K, V>.component2() = getValue()
```

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

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()`。在非性能关键代码中创建 map 可以用一个简单的 *惯用法* 来完成: `mapOf(a to b, c to d)`

注意上面的 `readOnlyView` 变量 (译者注: 与对应可变集合变量 `numbers`) 指向相同的底层 list 并会随之改变。如果一个 list 只存在只读引用, 我们可以考虑该集合完全不可变。创建一个这样的集合的一个简单方式如下:

```
val items = listOf(1, 2, 3)
```

目前 `listOf` 方法是使用 array list 实现的, 但是未来可以利用它们知道自己不能变的事实, 返回更节约内存的完全不可变的集合类型。

注意这些类型是 *协变的*。这意味着, 你可以把一个 `List<Rectangle>` 赋值给 `List<Shape>` 假定 `Rectangle` 继承自 `Shape`。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.requireNotNulls() // 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)
```

区间

区间表达式由具有操作符形式 `..` 的 `rangeTo` 函数辅以 `in` 和 `!in` 形成。区间是为任何可比较类型定义的,但对于整型原生类型,它有一个优化的实现。以下是使用区间的一些示例

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

整型区间 (`IntRange`、`LongRange`、`CharRange`) 有一个额外的特性:它们可以迭代。编译器负责将其转换为类似 Java 的基于索引的 `for`-循环而无额外开销。

```
for (i in 1..4) print(i) // 输出“1234”

for (i in 4..1) print(i) // 什么都不输出
```

如果你想倒序迭代数字呢?也很简单。你可以使用标准库中定义的 `downTo()` 函数

```
for (i in 4 downTo 1) print(i) // 输出“4321”
```

能否以不等于 1 的任意步长迭代数字?当然没问题, `step()` 函数有助于此

```
for (i in 1..4 step 2) print(i) // 输出“13”

for (i in 4 downTo 1 step 2) print(i) // 输出“42”
```

要创建一个不包括其结束元素的区间,可以使用 `until` 函数:

```
for (i in 1 until 10) { // i in [1, 10) 排除了 10
    println(i)
}
```

它是如何工作的

区间实现了该库中的一个公共接口: `ClosedRange<T>`。

`ClosedRange<T>` 在数学意义上表示一个闭区间,它是为可比较类型定义的。它有两个端点: `start` 和 `endInclusive` 他们都包含在区间内。其主要操作是 `contains`,通常以 `in`/`!in` 操作符形式使用。

整型数列 (`IntProgression`、`LongProgression`、`CharProgression`) 表示等差数列。数列由 `first` 元素、`last` 元素和非零的 `increment` 定义。第一个元素是 `first`,后续元素是前一个元素加上 `increment`。`last` 元素总会被迭代命中,除非该数列是空的。

数列是 `Iterable<N>` 的子类型,其中 `N` 分别为 `Int`、`Long` 或者 `Char`,所以它可用于 `for`-循环以及像 `map`、`filter` 等函数中。对 `Progression` 迭代相当于 Java/JavaScript 的基于索引的 `for`-循环:

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

对于整型类型, `..` 操作符创建一个同时实现 `ClosedRange<T>` 和 `*Progression` 的对象。例如, `IntRange` 实现了 `ClosedRange<Int>` 并扩展自 `IntProgression`,因此为 `IntProgression` 定义的所有操作也可用于 `IntRange`。`downTo()` 和 `step()` 函数的结果总是一个 `*Progression`。

数列由在其伴生对象中定义的 `fromClosedRange` 函数构造:

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

数列的 `last` 元素这样计算:对于正的 `increment` 找到不大于 `end` 值的最大值、或者对于负的 `increment` 找到不小于 `end` 值的最小值,使得 `(last - first) % increment == 0`。

一些实用函数

rangeTo()

整型类型的 `rangeTo()` 操作符只是调用 `*Range` 类的构造函数,例如:

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

浮点数 (`Double`、`Float`) 未定义它们的 `rangeTo` 操作符,而使用标准库提供的泛型 `Comparable` 类型的操作符:

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

该函数返回的区间不能用于迭代。

downTo()

扩展函数 `downTo()` 是为任何整型类型对定义的,这里有两个例子:

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

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

reversed()

扩展函数 `reversed()` 是为每个 `*Progression` 类定义的,并且所有这些函数返回反转后的数列。

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

step()

扩展函数 `step()` 是为每个 `*Progression` 类定义的,所有这些函数都返回带有修改了 `step` 值(函数参数)的数列。步长(`step`)值必须始终为正,因此该函数不会更改迭代的方向。

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

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

请注意,返回数列的 `last` 值可能与原始数列的 `last` 值不同,以便保持不变式 `(last - first) % increment == 0` 成立。这里是一个例子:

```
(1..12 step 2).last == 11 // 值为 [1, 3, 5, 7, 9, 11] 的数列
(1..12 step 3).last == 10 // 值为 [1, 4, 7, 10] 的数列
(1..12 step 4).last == 9  // 值为 [1, 5, 9] 的数列
```

类型的检查与转换

is 和 !is 操作符

我们可以在运行时通过使用 `is` 操作符或其否定形式 `!is` 来检查对象是否符合给定类型：

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

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

智能转换

在许多情况下,不需要在 Kotlin 中使用显式转换操作符,因为编译器跟踪 不可变值的 `is` -检查,并在需要时自动插入(安全的)转换：

```
fun demo(x: Any) {
    if (x is String) {
        print(x.length) // x 自动转换为字符串
    }
}
```

编译器足够聪明,能够知道如果反向检查导致返回那么该转换是安全的：

```
if (x !is String) return
print(x.length) // x 自动转换为字符串
```

或者在 `&&` 和 `||` 的右侧：

```
// `||` 右侧的 x 自动转换为字符串
if (x !is String || x.length == 0) return

// `&&` 右侧的 x 自动转换为字符串
if (x is String && x.length > 0) {
    print(x.length) // x 自动转换为字符串
}
```

这些 智能转换 用于 [when-表达式](#) 和 [while-循环](#) 也一样：

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

请注意,当编译器不能保证变量在检查和使用之间不可改变时,智能转换不能用。更具体地,智能转换能否适用根据以下规则：

- `val` 局部变量——总是可以；
- `val` 属性——如果属性是 `private` 或 `internal`,或者该检查在声明属性的同一模块中执行。智能转换不适用于 `open` 的属性或者具有自定义 `getter` 的属性；
- `var` 局部变量——如果变量在检查和使用之间没有修改、并且没有在会修改它的 `lambda` 中捕获；
- `var` 属性——决不可能(因为该变量可以随时被其他代码修改)。

“不安全的”转换操作符

通常,如果转换是不可能的,转换操作符会抛出一个异常。因此,我们称之为 *不安全的*。Kotlin 中的不安全转换由中缀操作符 `as` (参见[operator precedence](#)) 完成:

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

请注意,`null` 不能转换为 `String` 因该类型不是[可空的](#),即如果 `y` 为空,上面的代码会抛出一个异常。为了匹配 Java 转换语义,我们必须在转换右边有可空类型,就像:

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

“安全的”(可空)转换操作符

为了避免抛出异常,可以使用安全转换操作符 `as?`,它可以在失败时返回 `null`:

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

请注意,尽管事实上 `as?` 的右边是一个非空类型的 `String`,但是其转换的结果是可空的。

This 表达式

为了表示当前的 *接收者* 我们使用 `this` 表达式：

- 在 *类* 的成员中, `this` 指的是该类的当前对象
- 在 *扩展函数* 或者 *带接收者的函数数字面值* 中, `this` 表示在点左侧传递的 *接收者* 参数。

如果 `this` 没有限定符, 它指的是最内层的包含它的作用域。要引用其他作用域中的 `this`, 请使用 *标签限定符*：

限定的 `this`

要访问来自外部作用域的 `this` (一个 *类* 或者 *扩展函数*, 或者带标签的 *带接收者的函数数字面值*) 我们使用 `this@label`, 其中 `@label` 是一个代指 `this` 来源的标签：

```
class A { // 隐式标签 @A
    inner class B { // 隐式标签 @B
        fun Int.foo() { // 隐式标签 @foo
            val a = this@A // A 的 this
            val b = this@B // B 的 this

            val c = this // foo() 的接收者, 一个 Int
            val c1 = this@foo // foo() 的接收者, 一个 Int

            val funLit = lambda@ fun String.() {
                val d = this // funLit 的接收者
            }

            val funLit2 = { s: String ->
                // foo() 的接收者, 因为它包含的 lambda 表达式
                // 没有任何接收者
                val d1 = this
            }
        }
    }
}
```

相等性

Kotlin 中有两种类型的相等性：

- 引用相等 (两个引用指向同一对象)
- 结构相等 (用 `equals()` 检查)

引用相等

引用相等由 `===` (以及其否定形式 `!==`) 操作判断。`a === b` 当且仅当 `a` 和 `b` 指向同一个对象时求值为 `true`。

结构相等

结构相等由 `==` (以及其否定形式 `!=`) 操作判断。按照惯例, 像 `a == b` 这样的表达式会翻译成

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

也就是说如果 `a` 不是 `null` 则调用 `equals(Any?)` 函数, 否则 (即 `a` 是 `null`) 检查 `b` 是否与 `null` 引用相等。

请注意, 当与 `null` 显式比较时完全没必要优化你的代码: `a == null` 会被自动转换为 `a === null`。

操作符重载

Kotlin 允许我们为自己的类型提供预定义的一组操作符的实现。这些操作符具有固定的符号表示（如 `+` 或 `*`）和固定的[优先级](#)。为实现这样的操作符，我们为相应的类型（即二元操作符左侧的类型和一元操作符的参数类型）提供了一个固定名字的[成员函数](#)或[扩展函数](#)。重载操作符的函数需要用 `operator` 修饰符标记。

约定

在这里我们描述为不同操作符规范操作符重载的约定。

一元操作

表达式	翻译为
<code>+a</code>	<code>a.unaryPlus()</code>
<code>-a</code>	<code>a.unaryMinus()</code>
<code>!a</code>	<code>a.not()</code>


这个表是说，当编译器处理例如表达式 `+a` 时，它执行以下步骤：

- 确定 `a` 的类型，令其为 `T`。
- 为接收者 `T` 查找一个带有 `operator` 修饰符的无参函数 `unaryPlus()`，即成员函数或扩展函数。
- 如果函数不存在或不明确，则导致编译错误。
- 如果函数存在且其返回类型为 `R`，那就表达式 `+a` 具有类型 `R`。

注意 这些操作以及所有其他操作都针对[基本类型](#)做了优化，不会为它们引入函数调用的开销。

表达式	翻译为
<code>a++</code>	<code>a.inc()</code> + 见下文
<code>a--</code>	<code>a.dec()</code> + 见下文

这些操作应该改变它们的接收者并且（可选地）返回一个值。

 **`inc()/dec()` 不应该改变接收者对象。**
“改变接收者”我们的指的是 *该接收者变量值* 而不是接收者对象指向。

编译器执行以下步骤来解析 *后缀形式* 的操作符，例如 `a++`：

- 确定 `a` 的类型，令其为 `T`。
- 查找一个适用于类型为 `T` 的接收者的、带有 `operator` 修饰符的无参函数 `inc()`。
- 如果该函数返回类型 `R`，那么 `R` 必须是 `T` 的子类型。

计算表达式的步骤是：

- 把 `a` 的初始值存储到临时存储 `a0` 中，
- 把 `a.inc()` 结果赋值给 `a`，
- 把 `a0` 作为表达式的结果返回。

对于 `a--`，步骤是完全类似的。

对于 *前缀形式* `++a` 和 `--a` 以相同方式解析，其步骤是：

- 把 `a.inc()` 结果赋值给 `a`，
- 把 `a` 的新值作为表达式结果返回。

二元操作

表达式	翻译为
<code>a + b</code>	<code>a.plus(b)</code>

表达式	翻译为
b	us(b)
a * b	a.times(b)
a / b	a.div(b)
a % b	a.mod(b)
a..b	a.rangeTo(b)

对于此表中的操作,编译器只是解析成翻译为列中的表达式。

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

对于 in 和 !in,过程是相同的,但是参数的顺序是相反的。

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)

方括号转换为调用带有适当数量参数的 get 和 set。

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)

圆括号转换为调用带有适当数量参数的 invoke。

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)

对于赋值操作,例如 a += b,编译器执行以下步骤:

- 如果右列的函数可用
 - 如果相应的二元函数(即 plusAssign() 对应于 plus())也可用,那么报告错误(模糊)。
 - 确保其返回类型是 Unit,否则报告错误。
 - 生成 a.plusAssign(b) 的代码
- 否则试着生成 a = a + b 的代码(这里包含类型检查: a + b 的类型必须是 a 的子类型)。

注意:赋值在 Kotlin 中不是表达式。

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

注意: `===` 和 `!==` (同一性检查)不可重载,因此不存在对他们的约定

这个 `==` 操作符有些特殊:它被翻译成一个复杂的表达式,用于筛选 `null` 值,而 `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对 `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

可空类型与非空类型

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

许多编程语言(包括 Java)中最常见的陷阱之一是访问空引用的成员,导致空引用异常。在 Java 中,这等同于 `NullPointerException` 或简称 `NPE`。

Kotlin 的类型系统旨在从我们的代码中消除 `NullPointerException`。NPE 的唯一可能的原因可能是

- 显式调用 `throw NullPointerException()`
- 使用了下文描述的 `!!` 操作符
- 外部 Java 代码导致的
- 对于初始化,有一些数据不一致(如一个未初始化的 `this` 用于构造函数的某个地方)

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

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

如果要允许为空,我们可以声明一个变量为可空字符串,写作 `String?`:

```
var b: String? = "abc"
b = null // ok
```

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

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

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

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

但是我们还是需要访问该属性,对吧?有几种方式可以做到。

在条件中检查 `null`

首先,你可以显式检查 `b` 是否为 `null`,并分别处理两种可能:

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

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

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

请注意,这只适用于 `b` 是不可变的情况(即在检查和使用之间没有修改过的局部变量,或者不可覆盖并且有幕后字段的 `val` 成员),因为否则可能会发生在检查之后 `b` 又变为 `null` 的情况。

安全的调用

你的第二个选择是安全调用操作符,写作 `?.`:

```
b?.length
```

如果 `b` 非空,就返回 `b.length`,否则返回 `null`,这个表达式的类型是 `Int?`。

安全调用在链式调用中很有用。例如,如果一个员工 Bob 可能会(或者不会)分配给一个部门,并且可能有另外一个员工是该部门的负责人,那么获取 Bob 所在部门负责人(如果有的话)的名字,我们写作:

```
bob?.department?.head?.name
```

如果任意一个属性(环节)为空,这个链式调用就会返回 `null`。

如果要只对非空值执行某个操作,安全调用操作符可以与 `let` 一起使用:

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

Elvis 操作符

当我们有一个可空的引用 `r` 时,我们可以说“如果 `r` 非空,我使用它;否则使用某个非空的值 `x`”:

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

除了完整的 `if`-表达式,这还可以通过 Elvis 操作符表达,写作 `?:` :

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

如果 `?:` 左侧表达式非空,elvis 操作符就返回其左侧表达式,否则返回右侧表达式。请注意,当且仅当左侧为空时,才会对右侧表达式求值。

请注意,因为 `throw` 和 `return` 在 Kotlin 中都是表达式,所以它们也可以用在 elvis 操作符右侧。这可能会非常方便,例如,检查函数参数:

```
fun foo(node: Node): String? {
    val parent = node.getParent() ?: return null
    val name = node.getName() ?: throw IllegalArgumentException("name expected")
    // .....
}
```

!! 操作符

第三种选择是为 NPE 爱好者准备的。我们可以写 `b!!`,这会返回一个非空的 `b` 值(例如:在我们例子中的 `String`)或者如果 `b` 为空,就会抛出一个 NPE 异常:

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

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

安全的类型转换

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

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

可空类型的集合

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

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

异常

异常类

Kotlin 中所有异常类都是 `Throwable` 类的子孙类。每个异常都有消息、堆栈回溯信息和可选的原因。

使用 `throw`-表达式来抛出异常。

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

使用 `try`-表达式来捕获异常。

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

可以有零到多个 `catch` 块。`finally` 块可以省略。但是 `catch` 和 `finally` 块至少应该存在一个。

Try 是一个表达式

`try` 是一个表达式,即它可以有一个返回值。

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

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

受检的异常

Kotlin 没有受检的异常。这其中有很多原因,但我们会提供一个简单的例子。

以下是 JDK 中 `StringBuilder` 类实现的一个示例接口

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

这个签名是什么意思? 它是说,每次我追加一个字符串到一些东西(一个 `StringBuilder`、某种日志、一个控制台等)上时 我就必须捕获那些 `IOException`。为什么?因为它可能正在执行 IO 操作(`Writer` 也实现了 `Appendable`)…… 所以它导致这种代码随处可见的出现:

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

这并不好,参见[《Effective Java》](#)第 65 条:不要忽略异常。

Bruce Eckel 在[《Java 是否需要受检的异常?》\(Does Java need Checked Exceptions?\)](#)中指出:

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

其他相关引证:

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

Java 互操作性

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

注解

注解声明

注解是将元数据附加到代码的方法。要声明注解, 请将 `annotation` 修饰符放在类的前面:

```
annotation class Fancy
```

注解的附加属性可以通过用元注解标注注解类来指定:

- `@Target` 指定可以用 该注解标注的元素的可能的类型 (类、函数、属性、表达式等);
- `@Retention` 指定该注解是否 存储在编译后的 class 文件中, 以及它在运行时能否通过反射可见 (默认都是 true);
- `@Repeatable` 允许 在单个元素上多次使用相同的该注解;
- `@MustBeDocumented` 指定 该注解是公有 API 的一部分, 并且应该包含在 生成的 API 文档中显示的类或方法的签名中。

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

用法

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

如果需要对类的主构造函数进行标注, 则需要在构造函数声明中添加 `constructor` 关键字, 并将注解添加到其前面:

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

你也可以标注属性访问器:

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

构造函数

注解可以有接受参数的构造函数。

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

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

允许的参数类型有:

- 对应于 Java 原生类型的类型 (Int、Long等);
- 字符串;
- 类 (Foo::class);
- 枚举;
- 其他注解;
- 上面已列类型的数组。

如果注解用作另一个注解的参数,则其名称不以 @ 字符为前缀:

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

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

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

如果需要将一个类指定为注解的参数,请使用 Kotlin 类 ([KClass](#))。Kotlin 编译器会自动将其转换为 Java 类,以便 Java 代码能够正常看到该注解和参数。

```
import kotlin.reflect.KClass

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

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

Lambda 表达式

注解也可以用于 lambda 表达式。它们会被应用于生成 lambda 表达式体的 `invoke()` 方法上。这对于像 [Quasar](#) 这样的框架很有用,该框架使用注解进行并发控制。

```
annotation class Suspendable

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

注解使用处目标

当对属性或主构造函数参数进行标注时,从相应的 Kotlin 元素生成的 Java 元素会有多个,因此在生成的 Java 字节码中该注解有多个可能位置。如果要指定精确地指定应该如何生成该注解,请使用以下语法:

```
class Example(@field:Ann val foo,    // 标注 Java 字段
              @get:Ann val bar,     // 标注 Java getter
              @param:Ann val quux)  // 标注 Java 构造函数参数
```

可以使用相同的语法来标注整个文件。要做到这一点,把带有目标 `file` 的注解放在文件的顶层、`package` 指令之前或者在所有导入之前(如果文件在默认包中的话):

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

package org.jetbrains.demo
```

如果你对同一目标有多个注解,那么可以这样来避免目标重复——在目标后面添加方括号 并将所有注解放在方括号内:

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

支持的使用处目标的完整列表为:

- `file`
- `property` (具有此目标的注解对 Java 不可见)
- `field`
- `get` (属性 getter)
- `set` (属性 setter)
- `receiver` (扩展函数或属性的接收者参数)

- `param` (构造函数参数)
- `setparam` (属性 setter 参数)
- `delegate` (为委托属性存储其委托实例的字段)

要标注扩展函数的接收者参数, 请使用以下语法:

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

如果不指定使用处目标, 则根据正在使用的注解的 `@Target` 注解来选择目标。如果有多个适用的目标, 则使用以下列表中的第一个适用目标:

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

Java 注解

Java 注解与 Kotlin 100% 兼容:

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

class Tests {
    // 将 @Rule 注解应用于属性 getter
    @get:Rule val tempFolder = TemporaryFolder()

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

因为 Java 编写的注解没有定义参数顺序, 所以不能使用常规函数调用 语法来传递参数。相反, 你需要使用命名参数语法。

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

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

就像在 Java 中一样, 一个特殊的情况是 `value` 参数; 它的值无需显式名称指定。

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

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

如果 Java 中的 `value` 参数具有数组类型, 它会成为 Kotlin 中的一个 `vararg` 参数:

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

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

对于具有数组类型的其他参数,你需要显式使用 `arrayOf` :

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

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

注解实例的值会作为属性暴露给 Kotlin 代码。

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

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

反射

反射是这样的一组语言和库功能,它允许在运行时自省你的程序的结构。Kotlin 让语言中的函数和属性做为一等公民、并对其自省(即在运行时获悉一个名称或者一个属性或函数的类型)与简单地使用函数式或响应式风格紧密相关。

⚠ 在 Java 平台上,使用反射功能所需的运行时组件作为单独的 JAR 文件(kotlin-reflect.jar)分发。这样做是为了减少不使用反射功能的应用程序所需的运行时库的大小。如果你需要使用反射,请确保该.jar文件添加到项目的 classpath 中。

类引用

最基本的反射功能是获取 Kotlin 类的运行时引用。要获取对静态已知的 Kotlin 类的引用,可以使用类字面值语法:

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

该引用是 `KClass` 类型的值。

请注意,Kotlin 类引用与 Java 类引用不同。要获得 Java 类引用,请在 `KClass` 实例上使用 `.java` 属性。

函数引用

当我们有一个命名函数声明如下:

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

我们可以很容易地直接调用它(`isOdd(5)`),但是我们也可以把它作为一个值传递。例如传给另一个函数。为此,我们使用 `::` 操作符:

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

这里 `::isOdd` 是函数类型 `(Int) -> Boolean` 的一个值。

当上下文中已知函数期望的类型时, `::` 可以用于重载函数。例如:

```
fun isOdd(x: Int) = x % 2 != 0
fun isOdd(s: String) = s == "brillig" || s == "slithy" || s == "tove"

val numbers = listOf(1, 2, 3)
println(numbers.filter(::isOdd)) // 引用到 isOdd(x: Int)
```

或者,你可以通过将方法引用存储在具有显式指定类型的变量中来提供必要的上下文:

```
val predicate: (String) -> Boolean = ::isOdd // 引用到 isOdd(x: String)
```

如果我们需要使用类的成员函数或扩展函数,它需要是限定的。例如 `String::toCharArray` 为类型 `String` 提供了一个扩展函数: `String().toCharArray() -> CharArray`。

示例:函数组合

考虑以下函数:

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

它返回一个传给它的两个函数的组合: `compose(f, g) = f(g(*))`。现在,你可以将其应用于可用引用:

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

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

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

属性引用

要把属性作为 Kotlin 中的一等对象来访问,我们也可以使用 `::` 运算符:

```
var x = 1

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

表达式 `::x` 求值为 `KProperty<Int>` 类型的属性对象,它允许我们使用 `get()` 读取它的值,或者使用 `name` 属性来获取属性名。更多信息请参见[关于 KProperty 类的文档](#)。

对于可变属性,例如 `var y = 1`,`::y` 返回 `KMutableProperty<Int>` 类型的一个值,该类型有一个 `set()` 方法。

属性引用可以用在不需要参数的函数处:

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

要访问属于类的成员的属性,我们这样限定它:

```
class A(val p: Int)

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

与 Java 反射的互操作性

在 Java 平台上,标准库包含反射类的扩展,它提供了与 Java 反射对象之间映射(参见 `kotlin.reflect.jvm` 包)。例如,要查找一个用作 Kotlin 属性 getter 的幕后字段或 Java 方法,可以这样写:

```
import kotlin.reflect.jvm.*

class A(val p: Int)

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

要获得对应于 Java 类的 Kotlin 类,请使用 `.kotlin` 扩展属性:

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

构造函数引用

构造函数可以像方法和属性那样引用。他们可以用于期待这样的函数类型对象的任何地方：它与该构造函数接受相同参数并且返回相应类型的对象。通过使用 `::` 操作符并添加类名来引用构造函数。考虑下面的函数，它期待一个无参并返回 `Foo` 类型的函数参数：

```
class Foo

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

使用 `::Foo`，类 `Foo` 的零参数构造函数，我们可以这样简单地调用它：

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


类型安全的构建器

[构建器 \(builder\)](#) 的概念在 *Groovy* 社区中非常热门。构建器允许以半声明 (semi-declarative) 的方式定义数据。构建器很适合用来[生成 XML](#)、[布局 UI 组件](#)、[描述 3D 场景](#)以及其他更多功能……

对于很多情况下, Kotlin 允许检查类型的构建器, 这使得它们比 Groovy 自身的动态类型实现更具吸引力。

对于其余的情况, Kotlin 支持动态类型构建器。

一个类型安全的构建器示例

考虑下面的代码:

```
import com.example.html.* // 参见下文声明

fun result(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"}

            // 一个具有属性和文本内容的元素
            a(href = "http://kotlinlang.org") {"Kotlin"}

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

            // 以下代码生成的内容
            p {
                for (arg in args)
                    +arg
            }
        }
    }
```

这是完全合法的 Kotlin 代码。你可以[在这里](#)在线运行上文代码 (修改它并在浏览器中运行)。

实现原理

让我们来看看 Kotlin 中实现类型安全构建器的机制。首先, 我们需要定义我们想要构建的模型, 在本例中我们需要建模 HTML 标签。用一些类就可以轻易完成。例如, `HTML` 是一个描述 `<html>` 标签的类, 也就是说它定义了像 `<head>` 和 `<body>` 这样的子标签。(参见[下文](#)它的声明。)

现在, 让我们回想下为什么我们可以在代码中这样写:

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

`html` 实际上是一个函数调用, 它接受一个 [lambda 表达式](#) 作为参数。该函数定义如下:

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

这个函数接受一个名为 `init` 的参数,该参数本身就是一个函数。该函数的类型是 `HTML.() -> Unit`,它是一个 **带接收者的函数类型**。这意味着我们需要向函数传递一个 `HTML` 类型的实例(接收者),并且我们可以在函数内部调用该实例的成员。该接收者可以通过 `this` 关键字访问:

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

(`head` 和 `body` 是 `HTML` 的成员函数。)

现在,像往常一样,`this` 可以省略掉了,我们得到的东西看起来已经非常像一个构建器了:

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

那么,这个调用做什么?让我们看看上面定义的 `html` 函数的主体。它创建了一个 `HTML` 的新实例,然后通过调用作为参数传入的函数来初始化它(在我们的示例中,归结为在 `HTML` 实例上调用 `head` 和 `body`),然后返回此实例。这正是构建器所应做的。

`HTML` 类中的 `head` 和 `body` 函数的定义与 `html` 类似。唯一的区别是,它们将构建的实例添加到包含 `HTML` 实例的 `children` 集合中:

```
fun head(init: Head.() -> Unit) : Head {
    val head = Head()
    head.init()
    children.add(head)
    return head
}

fun body(init: Body.() -> Unit) : Body {
    val body = Body()
    body.init()
    children.add(body)
    return body
}
```

实际上这两个函数做同样的事情,所以我们可以有一个泛型版本, `initTag` :

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

所以,现在的函数很简单:

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

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

并且我们可以使用它们来构建 `<head>` 和 `<body>` 标签。

这里要讨论的另一件事是如何向标签体中添加文本。在上例中我们这样写到

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

所以基本上,我们只是把一个字符串放进一个标签体内部,但在它前面有一个小的 `+`,所以它是一个函数调用,调用一个前缀 `unaryPlus()` 操作。该操作实际上是由一个扩展函数 `unaryPlus()` 定义的,该函数是 `TagWithText` 抽象类(`Title` 的父类)的成员:

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

所以,在这里前缀 `+` 所做的事情是把一个字符串包装到一个 `TextElement` 实例中,并将其添加到 `children` 集合中,以使其成为标签树的一个适当的部分。

所有这些都在上面构建器示例顶部导入的包 `com.example.html` 中定义。在下一节中,你可以阅读这个包的完整定义。

com.example.html 包的完整定义

这就是 `com.example.html` 包的定义(只有上面例子中使用的元素)。它构建一个 HTML 树。代码中大量使用了[扩展函数](#)和[带接收者的 lambda 表达式](#)。

```
package com.example.html

interface Element {
    fun render(builder: StringBuilder, indent: String)
}

class TextElement(val text: String) : Element {
    override fun render(builder: StringBuilder, indent: String) {
        builder.append("$indent$text\n")
    }
}

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

Grammar

Notation

This section informally explains the grammar notation used below.

Symbols and naming

Terminal symbol names start with an uppercase letter, e.g. **SimpleName**.

Nonterminal symbol names start with lowercase letter, e.g. **kotlinFile**.

Each *production* starts with a colon (:).

Symbol definitions may have many productions and are terminated by a semicolon (;).

Symbol definitions may be prepended with *attributes*, e.g. `start` attribute denotes a start symbol.

EBNF expressions

Operator `|` denotes *alternative*.

Operator `*` denotes *iteration* (zero or more).

Operator `+` denotes *iteration* (one or more).

Operator `?` denotes *option* (zero or one).

alpha { beta } denotes a nonempty *beta*-separated list of *alpha*'s.

Operator `` `++` means that no space or comment allowed between operands.

Semicolons

Kotlin provides "semicolon inference": syntactically, subsentences (e.g., statements, declarations etc) are separated by the pseudo-token [SEMI](#), which stands for "semicolon or newline". In most cases, there's no need for semicolons in Kotlin code.

Syntax

Relevant pages: [Packages](#)

```
start
kotlinFile
: preamble toplevelObject*
;
start
script
: preamble expression*
;
preamble
(used by script, kotlinFile)
: fileAnnotations? packageHeader? import*
;
fileAnnotations
(used by preamble)
: fileAnnotation*
;
fileAnnotation
(used by fileAnnotations)
: "@" "file" ":" ("[" unescapedAnnotation+ "]" | unescapedAnnotation)
;
packageHeader
(used by preamble)
: modifiers "package" SimpleName{"."} SEMI?
;
See Packages

import
(used by preamble)
: "import" SimpleName{"."} (":" ".*" | "as" SimpleName)? SEMI?
;
See Imports
```

```

toplevelObject
(used by kotlinFile)
: class
: object
: function
: property
;

```

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, controlStructureBody, 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)
: 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](#)

```
controlStructureBody
  (used by whenEntry, for, if, doWhile, while)
: block
: blockLevelExpression
;
if
  (used by atomicExpression)
: "if" "(" expression ")" controlStructureBody SEMI? ("else" controlStructureBody)?
;
try
  (used by atomicExpression)
: "try" block catchBlock* finallyBlock?
;
catchBlock
  (used by try)
: "catch" "(" annotations SimpleName ":" userType ")" block
;
finallyBlock
  (used by try)
: "finally" block
;
loop
  (used by atomicExpression)
: for
: while
: doWhile
;
for
  (used by loop)
: "for" "(" annotations (multipleVariableDeclarations | variableDeclarationEntry) "in" expression ")" controlStructureBody
;
while
  (used by loop)
: "while" "(" expression ")" controlStructureBody
;
doWhile
  (used by loop)
: "do" controlStructureBody "while" "(" expression ")"
;
;
```

Expressions

Precedence

Precedence	Title	Symbols
Highest	Postfix	++, --, ., ?., ?

Precedence	Infix Type RHS	Symbols
		., as, as?, !, LabelDefinition@@
	Multiplicative	*, /, %
	Additive	+, -
	Range	..
	Infix function	SimpleName
	Elvis	?:
	Named checks	in, !in, is, !is
	Comparison	<, >, <=, >=
	Equality	==, \! ==
	Conjunction	&&
	Disjunction	
Lowest	Assignment	=, +=, -=, *=, /=, %=

Rules

```

expression
(used by for, atomicExpression, longTemplate, whenCondition, functionBody, doWhile, property, script, explicitDelegation, jump, while,
arrayAccess, blockLevelExpression, if, when, valueArguments, functionParameter)
: disjunction (assignmentOperator disjunction)*
;
disjunction
(used by expression)
: conjunction ("||" conjunction)*
;
conjunction
(used by disjunction)
: equalityComparison ("&&" equalityComparison)*
;
equalityComparison
(used by conjunction)
: comparison (equalityOperation comparison)*
;
comparison
(used by equalityComparison)
: namedInfix (comparisonOperation namedInfix)*
;
namedInfix
(used by comparison)
: elvisExpression (inOperation elvisExpression)*
: elvisExpression (isOperation 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
:
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
: blockLevelExpression
:
blockLevelExpression
(used by statement, controlStructureBody)
: annotations ("\\n")+ expression
:
multiplicativeOperation
(used by multiplicativeExpression)
: "*" : "/" : "%"
:
additiveOperation
(used by additiveExpression)
: "+" : "-"
:
inOperation
(used by namedInfix)
: "in" : "!in"
:
typeOperation
(used by typeRHS)
: "as" : "as?" : "."
:
isOperation
(used by namedInfix)
: "is" : "!is"
:
comparisonOperation
(used by comparison)
: "<" : ">" : ">=" : "<="
:
equalityOperation
(used by equalityComparison)
: "!=" : "=="
:

```

```

assignmentOperator
(used by expression)
: "="
: "+=" : "-=" : "*=" : "/=" : "%="
;
prefixUnaryOperation
(used by prefixUnaryExpression)
: "-" : "+"
: "++" : "--"
: "!"
: annotations
: labelDefinition
;
postfixUnaryOperation
(used by postfixUnaryExpression)
: "++" : "--" : "!!"
: callSuffix
: arrayAccess
: memberAccessOperation postfixUnaryExpression
;
callSuffix
(used by constructorInvocation, postfixUnaryOperation)
: typeArguments? valueArguments annotatedLambda
: typeArguments annotatedLambda
;
annotatedLambda
(used by callSuffix)
: ("@" unescapedAnnotation)* labelDefinition? functionLiteral
;
memberAccessOperation
(used by postfixUnaryOperation)
: "." : "?" : "?"
;
typeArguments
(used by callSuffix, callableReference, unescapedAnnotation)
: "<" type "{" type ">"
;
valueArguments
(used by callSuffix, constructorDelegationCall, unescapedAnnotation)
: "(" (SimpleName "=")? "*" expression "{" type "}" ")"
;
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 "{" type "}" "]"
;
objectLiteral
(used by atomicExpression)
: "object" ("{" delegationSpecifier "{" type "}" } classBody
;

```

Pattern matching

See [When-expression](#)

```

when
(used by atomicExpression)
: "when" ("(" expression ")")? "{"
  whenEntry*
  "}"
;
whenEntry
(used by when)
: whenCondition "{" type "}" "->" controlStructureBody SEMI
: "else" "->" controlStructureBody 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
: parameterModifier
: typeParameterModifier
: functionModifier
: propertyModifier
: annotations
;

classModifier
(used by modifierKeyword)
: "abstract"
: "final"
: "enum"
: "open"
: "annotation"
: "sealed"
: "data"
;

memberModifier
(used by modifierKeyword)
: "override"
: "open"
: "final"
: "abstract"
: "lateinit"
;

accessModifier
(used by modifierKeyword)
: "private"
: "protected"
: "public"
: "internal"
;

varianceAnnotation
(used by modifierKeyword, optionalProjection)
: "in"
: "out"
;

parameterModifier
(used by modifierKeyword)
: "noinline"
: "crossinline"
: "vararg"
;

typeParameterModifier
(used by modifierKeyword)
: "reified"
;

functionModifier
(used by modifierKeyword)
: "tailrec"
: "operator"
: "infix"
: "inline"
: "external"
;

propertyModifier
(used by modifierKeyword)
: "const"
;
```

Annotations

```
annotations
(used by catchBlock, prefixUnaryOperation, blockLevelExpression, 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"
: "setparam"
;
unescapedAnnotation
(used by annotation, fileAnnotation, annotatedLambda, annotationList)
: SimpleName{""} typeArguments? valueArguments?
;

```

Lexical structure

```

helper
Digit
(used by IntegerLiteral, HexDigit)
: ["0".."9"];
IntegerLiteral
(used by literalConstant)
: Digit+?
FloatLiteral
(used by literalConstant)
: <Java double literal>;
helper
HexDigit
(used by RegularStringPart, HexadecimalLiteral)
: Digit["A".."F", "a".."f"];
HexadecimalLiteral
(used by literalConstant)
: "0x" HexDigit+;
CharacterLiteral
(used by literalConstant)
: <character as in Java>;
See Basic types

```

```

NoEscapeString
(used by literalConstant)
: <""""-quoted string>;
RegularStringPart
(used by stringTemplateElement)
: <any character other than backslash, quote, $ or newline>
ShortTemplateEntryStart:
: "$"
EscapeSequence:
: UnicodeEscapeSequence | RegularEscapeSequence
UnicodeEscapeSequence:
: "\u" HexDigit{4}
RegularEscapeSequence:
: "\\" <any character other than newline>
See String templates

```

```

SEMI
(used by whenEntry, if, statements, packageHeader, property, import)
: <semicolon or newline>;
SimpleName
(used by typeParameter, catchBlock, simpleUserType, atomicExpression, LabelName, packageHeader, class, object, functionLiteral,
infixFunctionCall, function, parameter, callableReference, FieldName, variableDeclarationEntry, stringTemplateElement, setter,
enumEntry, import, valueArguments, unescapedAnnotation, typeConstraint)
: <java identifier>
: "" <java identifier> "`"
;
See Java interoperability

```

```

FieldName
(used by atomicExpression)
: "$" SimpleName;
LabelName
(used by labelReference, labelDefinition)
: "@" SimpleName;
See Returns and jumps

```

Java 互操作

在 Kotlin 中调用 Java 代码

Kotlin 在设计时就考虑了 Java 互操作性。可以从 Kotlin 中自然地调用现存的 Java 代码,并且在 Java 代码中也可以很顺利地调用 Kotlin 代码。在本节中我们会介绍从 Kotlin 中调用 Java 代码的一些细节。

几乎所有 Java 代码都可以使用而没有任何问题

```
import java.util.*

fun demo(source: List<Int>) {
    val list = ArrayList<Int>()
    // “for”-循环用于 Java 集合:
    for (item in source) {
        list.add(item)
    }
    // 操作符约定同样有效:
    for (i in 0..source.size() - 1) {
        list[i] = source[i] // 调用 get 和 set
    }
}
```

Getter 和 Setter

遵循 Java 约定的 getter 和 setter 的方法 (名称以 `get` 开头的无参数方法和以 `set` 开头的单参数方法) 在 Kotlin 中表示为属性。例如:

```
import java.util.Calendar

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

请注意,如果 Java 类只有一个 setter,它在 Kotlin 中不会作为属性可见,因为 Kotlin 目前不支持只写 (set-only) 属性。

返回 void 的方法

如果一个 Java 方法返回 void,那么从 Kotlin 调用时中返回 `Unit`。万一有人使用其返回值,它将由 Kotlin 编译器在调用处赋值,因为该值本身是预先知道的(是 `Unit`)。

将 Kotlin 中关键字的 Java 标识符进行转义

一些 Kotlin 关键字在 Java 中是有效标识符:`in`、`object`、`is` 等等。如果一个 Java 库使用了 Kotlin 关键字作为方法,你仍然可以通过反引号 (‘) 字符转义它来调用该方法

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

空安全和平台类型

Java 中的任何引用都可能是 `null`,这使得 Kotlin 对来自 Java 的对象要求严格空安全是不现实的。Java 声明的类型在 Kotlin 中会被特别对待并称为 *平台类型*。对这种类型的空检查会放宽,因此它们的安全保证与在 Java 中相同(更多请参见[下文](#))。

考虑以下示例:

```
val list = ArrayList<String>() // 非空 (构造函数结果)
list.add("Item")
val size = list.size() // 非空 (原生 int)
val item = list[0] // 推断为平台类型 (普通 Java 对象)
```

当我们调用平台类型变量的方法时, Kotlin 不会在编译时报告可空性错误, 但在运行时调用可能会失败, 因为空指针异常或者 Kotlin 生成的阻止空值传播的断言:

```
item.substring(1) // 允许, 如果 item == null 可能会抛出异常
```

平台类型是不可标示的, 意味着不能在语言中明确地写下它们。当把一个平台值赋值给一个 Kotlin 变量时, 可以依赖类型推断 (该变量会具有推断出的平台类型, 如上例中 `item` 所具有的类型), 或者我们可以选择我们期望的类型 (可空或非空类型均可):

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

如果我们选择非空类型, 编译器会在赋值时触发一个断言。这防止 Kotlin 的非空变量保存空值。当我们把平台值传递给期待非空值等的 Kotlin 函数时, 也会触发断言。总的来说, 编译器尽力阻止空值通过程序向远传播 (尽管鉴于泛型的原因, 有时这不可能完全消除)。

平台类型表示法

如上所述, 平台类型不能在程序中显式表述, 因此在语言中没有相应语法。然而, 编译器和 IDE 有时需要 (在错误信息中、参数信息中等) 显示他们, 所以我们用一个助记符来表示他们:

- `T!` 表示“`T` 或者 `T?`”,
- `(Mutable)Collection<T>!` 表示“可以可变或不可变、可空或不可空的 `T` 的 Java 集合”,
- `Array<(out) T>!` 表示“可空或者不可空的 `T` (或 `T` 的子类型) 的 Java 数组”

可空性注解

具有可空性注解的 Java 类型并不表示为平台类型, 而是表示为实际可空或非空的 Kotlin 类型。编译器支持多种可空性注解, 包括:

- [JetBrains](#) (`org.jetbrains.annotations` 包中的 `@Nullable` 和 `@NotNull`)
- `Android` (`com.android.annotations` 和 `android.support.annotations`)
- `JSR-305` (`javax.annotation`)
- `FindBugs` (`edu.umd.cs.findbugs.annotations`)
- `Eclipse` (`org.eclipse.jdt.annotation`)
- `Lombok` (`lombok.NonNull`)。

你可以在 [Kotlin 编译器源代码](#) 中找到完整的列表。

已映射类型

Kotlin 特殊处理一部分 Java 类型。这样的类型不是“按原样”从 Java 加载, 而是映射到相应的 Kotlin 类型。映射只发生在编译期间, 运行时表示保持不变。Java 的原生类型映射到相应的 Kotlin 类型 (请记住[平台类型](#)):

Java 类型	Kotlin 类型
byte	kotlin.Byte
short	kotlin.Short
int	kotlin.Int
long	kotlin.Long
char	kotlin.Char
float	kotlin.Float
double	kotlin.Double
boolean	kotlin.Boolean

一些非原生的内置类型也会作映射:

Java 类型	Kotlin 类型
java.lang.Object	kotlin.Any!
java.lang.Cloneable	kotlin.Cloneable!
java.lang.Comparable	kotlin.Comparable!
java.lang.Enum	kotlin.Enum!
java.lang.Annotation	kotlin.Annotation!
java.lang.Deprecated	kotlin.Deprecated!
java.lang.Void	kotlin.Nothing!
java.lang.CharSequence	kotlin.CharSequence!
java.lang.String	kotlin.String!
java.lang.Number	kotlin.Number!
java.lang.Throwable	kotlin.Throwable!

集合类型在 Kotlin 中可以是只读的或可变的,因此 Java 集合类型作如下映射:(下表中的所有 Kotlin 类型都驻留在 `kotlin` 包中)

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

Java 的数组按下文所述映射:

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

Kotlin 中的 Java 泛型

Kotlin 的泛型与 Java 有点不同(参见[泛型](#))。当将 Java 类型导入 Kotlin 时,我们会执行一些转换:

- Java 的通配符转换成类型投影
 - `Foo<? extends Bar>` 转换成 `Foo<out Bar!>!`
 - `Foo<? super Bar>` 转换成 `Foo<in Bar!>!`
- Java 的原始类型转换成星投影
 - `List` 转换成 `List<*>!`,即 `List<out Any?>!`

和 Java 一样,Kotlin 在运行时不保留泛型,即对象不携带传递到他们构造器中的那些类型参数的实际类型。即 `ArrayList<Integer>()` 和 `ArrayList<Character>()` 是不能区分的。这使得执行 `is`-检测不可能照顾到泛型。Kotlin 只允许 `is`-检测星投影的泛型类型:

```
if (a is List<Int>) // 错误:无法检查它是否真的是一个 Int 列表
// but
if (a is List<*>) // OK:不保证列表的内容
```

Java 数组

与 Java 不同,Kotlin 中的数组是不型变的。这意味着 Kotlin 不允许我们把一个 `Array<String>` 赋值给一个 `Array<Any>`,从而避免了可能的运行时故障。Kotlin 也禁止我们把一个子类的数组当做超类的数组传递给 Kotlin 的方法,但是对于 Java 方法,这是允许的(通过 `Array<(out) String>!` 这种形式的[平台类型](#))。

Java 平台上,数组会使用原生数据类型以避免装箱/拆箱操作的开销。由于 Kotlin 隐藏了这些实现细节,因此需要一个变通方法来与 Java 代码进行交互。对于每种原生类型的数组都有一个特化的类 (`IntArray`、`DoubleArray`、`CharArray` 等等)来处理这种情况。它们与 `Array` 类无关,并且会编译成 Java 原生类型数组以获得最佳性能。

假设有一个接受 `int` 数组索引的 Java 方法:

```
public class JavaArrayExample {  
  
    public void removeIndices(int[] indices) {  
        // 在此编码.....  
    }  
}
```

在 Kotlin 中你可以这样传递一个原生类型的数组:

```
val javaObj = JavaArrayExample()  
val array = intArrayOf(0, 1, 2, 3)  
javaObj.removeIndices(array) // 将 int[] 传给方法
```

当编译为 JVM 字节代码时,编译器会优化对数组的访问,这样就不会引入任何开销:

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

即使当我们使用索引定位时,也不会引入任何开销

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

最后,`in`-检测也没有额外开销

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

Java 可变参数

Java 类有时声明一个具有可变数量参数 (`varargs`) 的方法来使用索引。

```
public class JavaArrayExample {  
  
    public void removeIndices(int... indices) {  
        // 在此编码.....  
    }  
}
```

在这种情况下,你需要使用展开运算符 `*` 来传递 `IntArray`:

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

目前无法传递 `null` 给一个声明为可变参数的方法。

操作符

由于 Java 无法标记用于运算符语法的方法,Kotlin 允许具有正确名称和签名的任何 Java 方法作为运算符重载和其他约定 (`invoke()` 等)使用。不允许使用中缀调用语法调用 Java 方法。

受检异常

在 Kotlin 中,所有异常都是非受检的,这意味着编译器不会强迫你捕获其中的任何一个。因此,当你调用一个声明受检异常的 Java 方法时,Kotlin 不会强迫你做任何事情:

```
fun render(list: List<*>, to: Appendable) {
    for (item in list) {
        to.append(item.toString()) // Java 会要求我们在这里捕获 IOException
    }
}
```

对象方法

当 Java 类型导入到 Kotlin 中时,类型 `java.lang.Object` 的所有引用都成了 `Any`。而因为 `Any` 不是平台指定的,它只声明了 `toString()`、`hashCode()` 和 `equals()` 作为其成员,所以为了能用 `java.lang.Object` 的其他成员,Kotlin 要用到[扩展函数](#)。

wait()/notify()

[Effective Java](#) 第 69 条善意地建议优先使用并发工具 (concurrency utilities) 而不是 `wait()` 和 `notify()`。因此,类型 `Any` 的引用不提供这两个方法。如果你真的需要调用它们的话,你可以将其转换为 `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
```

使用 `Foo::class.java` 而不是 Java 的写法 `Foo.class`。

```
val fooClass = Foo::class.java
```

clone()

要覆盖 `clone()`,需要继承 `kotlin.Cloneable` :

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

不要忘记 [Effective Java](#) 的第 11 条: *谨慎地改写 clone*。

finalize()

要覆盖 `finalize()`,所有你需要做的就是简单地声明它,而不需要 `override` 关键字:

```
class C {
    protected fun finalize() {
        // 终止化逻辑
    }
}
```

根据 Java 的规则, `finalize()` 不能是 `private` 的。

从 Java 类继承

在 kotlin 中,类的超类中最多只能有一个 Java 类(以及按你所需的多个 Java 接口)。

访问静态成员

Java 类的静态成员会形成该类的“伴生对象”。我们无法将这样的“伴生对象”作为值来传递，但可以显式访问其成员，例如：

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

Java 反射

Java 反射适用于 Kotlin 类，反之亦然。如上所述，你可以使用 `instance.javaClass` 或者 `ClassName::class.java` 通过 `java.lang.Class` 来进入 Java 反射。

其他支持的情况包括为一个 Kotlin 属性获取一个 Java 的 getter/setter 方法或者幕后字段、为一个 Java 字段获取一个 `KProperty`、为一个 `KFunction` 获取一个 Java 方法或者构造函数，反之亦然。

SAM 转换

就像 Java 8 一样，Kotlin 支持 SAM 转换。这意味着 Kotlin 函数面值可以被自动的转换成 只有一个非默认方法的 Java 接口的实现，只要这个方法的参数类型 能够与这个 Kotlin 函数的参数类型相匹配。

你可以这样创建 SAM 接口的实例：

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

……以及在方法调用中：

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

如果 Java 类有多个接受函数式接口的方法，那么可以通过使用 将 lambda 表达式转换为特定的 SAM 类型的适配器函数来选择需要调用的方法。这些适配器函数也会按需 由编译器生成。

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

请注意，SAM 转换只适用于接口，而不适用于抽象类，即使这些抽象类也只有一个 抽象方法。

还要注意，此功能只适用于 Java 互操作；因为 Kotlin 具有合适的函数类型，所以不需要将函数自动转换 为 Kotlin 接口的实现，因此不受支持。

在 Kotlin 中使用 JNI

要声明一个在本地 (C 或 C++) 代码中实现的函数，你需要使用 `external` 修饰符来标记它：

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

其余的过程与 Java 中的工作方式完全相同。

Java 调用 Kotlin

Java 可以轻松调用 Kotlin 代码。

属性

属性的 `getter` 会转换成 `get`-方法、`setter` 会转换成 `set`-方法。

包级函数

在 `org.foo.bar` 包内的 `example.kt` 文件中声明的所有的函数和属性,都会被放到一个名为 `org.foo.bar.ExampleKt` 的 java 类中。

```
// example.kt
package demo

class Foo

fun bar() {
}
```

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

可以使用 `@JvmName` 注解修改生成的 Java 类的类名:

```
@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(包名相同并且类名相同或者有相同的 `@JvmName` 注解)通常是错误的。然而,编译器能够生成一个单一的 Java 外观类,它具有指定的名称且包含来自所有文件中具有该名称的所有声明。要启用生成这样的外观,请在所有相关文件中使用 `@JvmMultifileClass` 注解。

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

package demo

fun foo() {
}
```

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

package demo

fun bar() {
}
```

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

实例字段

如果需要在 Java 中将 Kotlin 属性作为字段暴露,那就需要使用 `@JvmField` 注解对其标注。该字段将具有与底层属性相同的可见性。如果一个属性有幕后字段(backing field)、非私有、没有 `open / override` 或者 `const` 修饰符并且不是被委托的属性,那么你可以用 `@JvmField` 注解该属性。

```
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)也会暴露为字段。该字段的可见性与 `lateinit` 属性的 setter 相同。

静态字段

在命名对象或伴生对象中声明的 Kotlin 属性会在该命名对象或包含伴生对象的类中 具有静态幕后字段。

通常这些字段是私有的,但可以通过以下方式之一暴露出来:

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

使用 `@JvmField` 标注这样的属性使其成为与属性本身具有相同可见性的静态字段。

```
class Key(val value: Int) {
    companion object {
        @JvmField
        val COMPARATOR: Comparator<Key> = compareBy<Key> { it.value }
    }
}
```

```
// Java
Key.COMPARATOR.compare(key1, key2);
// Key 类中的 public static final 字段
```

在命名对象或者伴生对象中的一个[延迟初始化的](#)属性 具有与属性 setter 相同可见性的静态幕后字段。

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

```
// Java
Singleton.provider = new Provider();
// 在 Singleton 类中的 public static 非-final 字段
```

用 `const` 标注的(在类中以及在顶层的)属性在 Java 中会成为静态字段:

```
// 文件 example.kt

object Obj {
    const val CONST = 1
}

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

const val MAX = 239
```

在 Java 中:

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

静态方法

如上所述, Kotlin 会为包级函数生成静态方法。Kotlin 还可以为命名对象或伴生对象中定义的函数生成静态方法, 如果你将这些函数标注为 `@JvmStatic` 的话。例如:

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

现在, `foo()` 在 Java 中是静态的, 而 `bar()` 不是:

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

对于命名对象也同样:

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

在 Java 中:

```
Obj.foo(); // 没问题
Obj.bar(); // 错误
Obj.INSTANCE.bar(); // 没问题, 通过单例实例调用
Obj.INSTANCE.foo(); // 也没问题
```

`@JvmStatic` 注解也可以应用于对象或伴生对象的属性, 使其 `getter` 和 `setter` 方法在该对象或包含该伴生对象的类中是静态成员。

KClass

有时你需要调用有 `KClass` 类型参数的 Kotlin 方法。因为没有从 `Class` 到 `KClass` 的自动转换, 所以你必须通过调用 `Class<T>.kotlin` 扩展属性的等价形式来手动进行转换:

```
kotlin.jvm.JvmClassMappingKt.getKotlinClass(MainView.class)
```

用 `@JvmName` 解决签名冲突

有时我们想让一个 Kotlin 中的命名函数在字节码中有另外一个 JVM 名称。最突出的例子是由于类型擦除引发的：

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

这两个函数不能同时定义,因为它们的 JVM 签名是一样的: `filterValid(Ljava/util/List;)Ljava/util/List;`。如果我们真的希望它们在 Kotlin 中用相同名称,我们需要用 `@JvmName` 去标注其中的一个(或两个),并指定不同的名称作为参数:

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

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

在 Kotlin 中它们可以用相同的名称 `filterValid` 来访问,而在 Java 中,它们分别是 `filterValid` 和 `filterValidInt`。

同样的技巧也适用于属性 `x` 和函数 `getX()` 共存:

```
val x: Int
    @JvmName("getX_prop")
    get() = 15

fun getX() = 10
```

生成重载

通常,如果你写一个有默认参数值的 Kotlin 方法,在 Java 中只会有一个所有参数都存在的完整参数签名的方法可见,如果希望向 Java 调用者暴露多个重载,可以使用 `@JvmOverloads` 注解。

```
@JvmOverloads fun f(a: String, b: Int = 0, c: String = "abc") {
    ...
}
```

对于每一个有默认值的参数,都会生成一个额外的重载,这个重载会把这个参数和它右边的所有参数都移除掉。在上例中,会生成以下方法:

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

该注解也适用于构造函数、静态方法等。它不能用于抽象方法,包括在接口中定义的方法。

请注意,如[次构造函数](#)中所述,如果一个类的所有构造函数参数都有默认值,那么会为其生成一个公有的无参构造函数。这就算没有 `@JvmOverloads` 注解也有效。

受检异常

如上所述,Kotlin 没有受检异常。所以,通常 Kotlin 函数的 Java 签名不会声明抛出异常。于是如果我们有一个这样的 Kotlin 函数:

```
// example.kt
package demo

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

然后我们想要在 Java 中调用它并捕捉这个异常:

```
// Java
try {
    demo.Example.foo();
}
catch (IOException e) { // 错误:foo() 未在 throws 列表中声明 IOException
    // ...
}
```

因为 `foo()` 没有声明 `IOException`，我们从 Java 编译器得到了一个报错消息。为了解决这个问题，要在 Kotlin 中使用 `@Throws` 注解。

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

空安全性

当从 Java 中调用 Kotlin 函数时，没人阻止我们将 `null` 作为非空参数传递。这就是为什么 Kotlin 给所有期望非空参数的公有函数生成运行时检测。这样我们就能在 Java 代码里立即得到 `NullPointerException`。

型变的泛型

当 Kotlin 的类使用了[声明处型变](#)，有两种选择 可以从 Java 代码中看到它们的用法。让我们假设我们有以下类和两个使用它的函数：

```
class Box<out T>(val value: T)

interface Base
class Derived : Base

fun boxDerived(value: Derived): Box<Derived> = Box(value)
fun unboxBase(box: Box<Base>): Base = box.value
```

一种看似理所当然地将这俩函数转换成 Java 代码的方式可能会是：

```
Box<Derived> boxDerived(Derived value) { ... }
Base unboxBase(Box<Base> box) { ... }
```

问题是，在 Kotlin 中我们可以这样写 `unboxBase(boxDerived("s"))`，但是在 Java 中是行不通的，因为在 Java 中类 `Box` 在其泛型参数 `T` 上是不型变的，于是 `Box<Derived>` 并不是 `Box<Base>` 的子类。要使其在 Java 中工作，我们按以下这样定义 `unboxBase`：

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

这里我们使用 Java 的[通配符类型](#)(`? extends Base`)来 通过使用处型变来模拟声明处型变，因为在 Java 中只能这样。

当它作为参数出现时，为了让 Kotlin 的 API 在 Java 中工作，对于协变定义的 `Box` 我们生成 `Box<Super>` 作为 `Box<? extends Super>`（或者对于逆变定义的 `Foo` 生成 `Foo<? super Bar>`）。当它是一个返回值时，我们不生成通配符，因为否则 Java 客户端将必须处理它们（并且它违反常用 Java 编码风格）。因此，我们的示例中的对应函数实际上翻译如下：

```
// 作为返回类型—没有通配符
Box<Derived> boxDerived(Derived value) { ... }

// 作为参数—有通配符
Base unboxBase(Box<? extends Base> box) { ... }
```

注意：当参数类型是 `final` 时，生成通配符通常没有意义，所以无论在什么地方 `Box<String>` 始终转换为 `Box<String>`。

如果我们在默认不生成通配符的地方需要通配符，我们可以使用 `@JvmWildcard` 注解：

```
fun boxDerived(value: Derived): Box<@JvmWildcard Derived> = Box(value)
// 将被转换成
// Box<? extends Derived> boxDerived(Derived value) { ... }
```


另一方面,如果我们根本不需要默认的通配符转换,我们可以使用 `@JvmSuppressWildcards`

```
fun unboxBase(box: Box<@JvmSuppressWildcards Base>): Base = box.value
// 会翻译成
// Base unboxBase(Box<Base> box) { ... }
```

注意: `@JvmSuppressWildcards` 不仅可用于单个类型参数,还可用于整个声明(如 函数或类),从而抑制其中的所有通配符。

Nothing 类型翻译

`Nothing` 是一种特殊的类型,因为它在 Java 中没有自然对应。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,这种类型不能在 Java 世界中 准确表示。这就是为什么在使用 `Nothing` 参数的地方 Kotlin 生成一个原始类型:

```
fun emptyList(): List<Nothing> = listOf()
// 会翻译成
// List emptyList() { ... }
```

JavaScript

动态类型

⚠️ 在面向 JVM 平台的代码中不支持动态类型

作为一种静态类型的语言, Kotlin 仍然需要与无类型或松散类型的环境 (例如 JavaScript 生态系统) 进行互操作。为了方便这些使用场景, 语言中有 `dynamic` 类型可用:

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

`dynamic` 类型基本上关闭了 Kotlin 的类型检查系统:

- 该类型的值可以赋值给任何变量或作为参数传递到任何位置,
- 任何值都可以赋值给 `dynamic` 类型的变量, 或者传递给一个接受 `dynamic` 作为参数的函数,
- `null` -检查对这些值是禁用的。

`dynamic` 最特别的特性是, 我们可以对 `dynamic` 变量调用**任何**属性或以任意参数调用**任何**函数:

```
dyn.whatever(1, "foo", dyn) // 'whatever' 在任何地方都没有定义
dyn.whatever(*arrayOf(1, 2, 3))
```

在 JavaScript 平台上, 该代码将按照原样编译: 在生成的 JavaScript 代码中, Kotlin 中的 `dyn.whatever(1)` 变为 `dyn.whatever(1)`。

动态调用总是返回 `dynamic` 作为结果, 所以我们可以自由地这样链接调用:

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

当我们把一个 lambda 表达式传给一个动态调用时, 它的所有参数默认都是 `dynamic` 类型的:

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

更多技术说明请参见[规范文档](#)。

JavaScript 互操作性

JavaScript 模块

从 Kotlin 1.0.4 版开始,你可以将 Kotlin 项目编译为热门模块系统的 JS 模块。以下是 可用选项的列表:

1. 无模块 (Plain)。不为任何模块系统编译。像往常一样,你可以通过 `kotlin.modules.moduleName` 访问模块 `moduleName`,或者直接通过 `moduleName` 标识符访问放在全局作用域内的。默认使用此选项。
2. [异步模块定义 \(AMD, Asynchronous Module Definition\)](#),它尤其为 `require.js` 库所使用。
3. [CommonJS](#) 约定,广泛用于 `node.js/npm` (`require` 函数和 `module.exports` 对象)
4. 统一模块定义 (UMD, Unified Module Definitions),它与 *AMD* 和 *CommonJS* 兼容,并且当 *AMD* 和 *CommonJS* 都不可用时,作为“plain”使用。

选择目标模块系统的方式取决于你的构建环境:

对于 IDEA

打开“File → Settings”,选择“Build, Execution, Deployment”→“Compiler”→“Kotlin compiler”。在“Module kind”字段中选择合适的模块系统。

对于 Maven

要选择通过 Maven 编译时的模块系统,你应该设置 `moduleKind` 配置属性,即你的 `pom.xml` 应该看起来像这样:

```
<plugin>
  <artifactId>kotlin-maven-plugin</artifactId>
  <groupId>org.jetbrains.kotlin</groupId>
  <version>${kotlin.version}</version>
  <executions>
    <execution>
      <id>compile</id>
      <goals>
        <goal>js</goal>
      </goals>
    </execution>
  </executions>
  <!-- 插入这些行 -->
  <configuration>
    <moduleKind>commonjs</moduleKind>
  </configuration>
  <!-- 插入文本结束 -->
</plugin>
```

可用值包括: `plain`、`amd`、`commonjs`、`umd`。

对于 Gradle

要选择通过 Gradle 编译时的模块系统,你应该设置 `moduleKind` 属性,即

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

可用的值类似于 Maven

注意事项

我们将 `kotlin.js` 标准库作为一个单独的文件,它本身编译为一个 UMD 模块,所以你可以与上述任何模块系统一起使用。

虽然现在我们没有直接支持 WebPack 和 Browserify,但是我们测试了由 Kotlin 编译器生成的 `.js` 文件与 WebPack 和 Browserify 一同使用,所以 Kotlin 应该能正确使用这些工具。

@JsName 注解

在某些情况下(例如,为了支持重载),Kotlin编译器会修饰(mangle) JavaScript 代码中生成的函数和属性的名称。如果要控制生成的名称,可以使用 `@JsName` 注解:

```
// 模块 'kjs'

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

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

现在你可以从 JavaScript 中用以下方式使用该类：

```
var person = new kjs.Person("Dmitry"); // 引用到模块 'kjs'
person.hello(); // 输出 "Hello Dmitry!"
person.helloWithGreeting("Servus"); // 输出 "Servus Dmitry!"
```

如果我们没有指定 `@JsName` 注解，相应函数的名称将包含从函数签名计算而来的后缀，例如 `hello_61zpoes$`。

JavaScript 反射

在 Kotlin 编译成的 JavaScript 中,有一个属性 可用于任何对象,名为 `jsClass`,它返回一个 `JsClass` 实例。`JsClass` 目前只能提供 (无限定的) 类的名称。然而, `JsClass` 实例本身是一个对构造函数的引用。这可以用于与期待构造函数的引用的 JS 函数互操作。

要获得对类的引用,可以使用 `::class` 语法。Kotlin for JavaScript 目前不支持完整的反射 API; 唯一可用的属性是 `.simpleName` 返回类的名称 以及 `.js` 返回相应的 `JsClass`。

示例:

```
class A
class B
class C

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

println(A().jsClass.name)      // 输出“A”
println(B::class.simpleName)  // 输出“B”
println(B::class.js.name)     // 输出“B”
foo<C>()                       // 输出“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

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 Maven V3

Define the version of Kotlin you want to use via *akotlin.version* property:

```
<properties>
  <kotlin.version>1.0.6</kotlin.version>
</properties>
```

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 插件用于编译源码:

```
<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}/kotlin-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}/kotlin-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}/kotlin-ant.jar"/>

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

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}/kotlin-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}/kotlin-ant.jar"/>

  <target name="build">
    <kotlin2js src="root1" output="out.js" outputPrefix="prefix" outputPostfix="postfix"
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}/kotlin-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	

Name	Description	Required	Default Value
includeRuntime	Whether to include a .jar file, whether Kotlin runtime library is included in the jar		
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.

Compiler Options

To specify additional compilation options, use the `kotlinOptions` property of a Kotlin compilation task. Examples:

```
compileKotlin {
    kotlinOptions.suppressWarnings = true
}

compileKotlin {
    kotlinOptions {
        suppressWarnings = true
    }
}
```

A complete list of options for the Gradle tasks follows:

Attributes common for 'kotlin' and 'kotlin2js'

Name	Description	Possible values	Default value
apiVersion	Allow to use declarations only from the specified version of bundled libraries	"1.0"	"1.0"
languageVersion	Provide source compatibility with specified language version	"1.0"	"1.0"
suppressWarnings	Generate no warnings		false
verbose	Enable verbose logging output		false
freeCompilerArgs	A list of additional compiler arguments		[]

Attributes specific for 'kotlin'

Name	Description	Possible values	Default value
includeRuntime	Include Kotlin runtime in to resulting .jar		false
jdkHome	Path to JDK home directory to include into classpath, if differs from default JAVA_HOME		
jvmTarget	Target version of the generated JVM bytecode, only 1.6 is supported	"1.6"	"1.6"
noJdk	Don't include Java runtime into classpath		false
noReflect	Don't include Kotlin reflection implementation into classpath		true
noStdlib	Don't include Kotlin runtime into classpath		true

Attributes specific for 'kotlin2js'

Name	Description	Possible values	Default value
kjsm	Generate kjsm-files (for creating libraries)		true
main	Whether a main function should be called	"call", "noCall"	"call"
metaInfo	Generate metadata		true
moduleKind	Kind of a module generated by compiler	"plain", "amd", "commonjs", "umd"	"plain"
noStdlib	Don't use bundled Kotlin stdlib		true
outputFile	Output file path		
sourceMap	Generate source map		false
target	Generate JS files for specific ECMA version	"v5"	"v5"

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 与 OSGi

要启用 Kotlin OSGi 支持,你需要引入 `kotlin-osgi-bundle` 而不是常规的 Kotlin 库。建议删除 `kotlin-runtime`、`kotlin-stdlib` 和 `kotlin-reflect` 依赖,因为 `kotlin-osgi-bundle` 已经包含了所有这些。当引入外部 Kotlin 库时你也应该注意。大多数常规 Kotlin 依赖不是 OSGi-就绪的,所以你不应该使用它们,且应该从你的项目中删除它们。

Maven

将 Kotlin OSGi 包引入到 Maven 项目中:

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

从外部库中排除标准库(注意“星排除”只在 Maven 3 中有效)

```
<dependency>
  <groupId>some.group.id</groupId>
  <artifactId>some.library</artifactId>
  <version>some.library.version</version>

  <exclusions>
    <exclusion>
      <groupId>org.jetbrains.kotlin</groupId>
      <artifactId>*</artifactId>
    </exclusion>
  </exclusions>
</dependency>
```

Gradle

将 `kotlin-osgi-bundle` 引入到 gradle 项目中:

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

要排除作为传递依赖的默认 Kotlin 库,你可以使用以下方法

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

FAQ

为什么不只是添加必需的清单选项到所有 Kotlin 库

尽管它是提供 OSGi 支持的最好的方式,遗憾的是现在做不到,是因为不能轻易消除的所谓的“[包拆分](#)”问题并且这么大的变化不可能现在规划。有 `Require-Bundle` 功能,但它也不是最好的选择,不推荐使用。所以决定为 OSGi 做一个单独的构件。

常见问题

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[不可变的](#)
- 相对于 Java 的 SAM-转换, Kotlin 有更合适的[函数类型](#)
- 没有通配符的[使用外型变](#)
- 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](#)
- [声明外型变 & 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

