



Kotlin Language Documentation

Table of Contents

- 开始 4
 - 基本语法 4
 - 习惯用法 10
 - 编码规范 15
- 基础 16
 - 基本类型 16
 - 包 22
 - 控制流 23
 - 返回和跳转 27
- 类和对象 29
 - 类和继承 29
 - 属性和字段 35
 - 接口 39
 - 可见性修饰符 41
 - 扩展 44
 - 数据类 50
 - 泛型 52
 - 泛型函数 56
 - 泛型约束 56
 - 嵌套类 58
 - 枚举类 59
 - 对象表达式和对象声明 61
 - 委托 64
 - 委托属性 65
- 函数和Lambdas表达式 69
 - 函数 69
 - 高阶函数和lambda表达式 75
 - 内联函数 inline 80
- 其他 83

解构声明	83
集合	85
范围	87
类型的检查与转换	91
.This表达式	93
等式	94
运算符重载	95
Null 安全性	98
异常	101
注解	103
反射	108
Type-Safe Builders	111
动态类型	117
参考	118
互操作	120
在Kotlin中调用Java代码	120
Java调用Kotlin代码	128
工具	136
生成kotlin代码文档	136
使用 Maven	139
Using Ant	142
使用 Gradle	146
Kotlin and OSGi	150
常见问题	152
FAQ	152
对比Java	155
对比Scala	156

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

参阅 [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)  
}
```

参阅 [函数](#).

定义局部变量

常量（使用 `val` 关键字声明）：

```
val a: Int = 1  
val b = 1 // `Int` 类型自动推断  
val c: Int // 如果没有初始值，声明常量时，常量的类型不能省略  
c = 1 // 明确赋值
```

变量（使用 `var` 关键字声明）：

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

参阅 [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]}")  
}
```

参阅 [字符串模板](#).

使用条件判断

```
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
```

参阅 [if 表达式](#).

使用可空值及 `null` 检查

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? {
    // ...
}
```

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) {
        // 在空指针判断后，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
}

// 在空指针判断后, x 和 y 会自动转换为非空值
print(x * y)
```

参阅 [Null 安全性](#).

使用类型检查及自动类型转换

`is` 运算符用于类型判断: 检查某个实例是否是某类型。如果一个局部常量或者不可变的类成员变量已经判断出为某类型, 那么判断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` 在该条件判断分支内自动转换成 `String`
        return obj.length
    }

    // 在离开类型判断分支后, `obj` 仍然是 `Any` 类型
    return null
}
```

或者

```
fun getStringLength(obj: Any): Int? {
    if (obj !is String)
        return null

    // `obj` 在这一类型判断分支自动转换为 `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
}
```

参阅 [类](#) and [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])
```

参阅 [for循环](#).

Using a while loop

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

See [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")  
    }  
}
```

参阅 [when表达式](#).

使用区间 (range)

使用 `in` 运算符来检查某个数字是否在指定区间内:

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

检查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)
```

参阅 [区间 Range](#).

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) }
```

参阅 [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！

创建方便任务间传递的数据 DTO's (POJO's/POCO's)

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

会为 `Customer` 类提供以下功能：

- 所有属性的 getters （对于 `var` 定义的还有 setters）
- `equals()`
- `hashCode()`
- `toString()`
- `copy()`
- 所有属性的 `component1()`, `component2()`, ... 等等 (参阅[数据类](#))

函数的默认参数

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

过滤 list

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

或者可以更短：

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

String 内插

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

类型判断

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

遍历 map/pair型list

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

`k`、`v` 可以改成任意名字。

使用区间 (range)

```
for (i in 1..100) { ... }
for (x in 2..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 {
    // compute the string
}
```

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)
    }

    // Working with 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) { //draw a 100 pix square  
    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())  
}
```

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"  
  
}
```

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`

存储方式

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

另一方面，它保留了相等性：

```
val a: Int = 10000
print(a == a) // Prints 'true'
val boxedA: Int? = a
val anotherBoxedA: Int? = a
print(boxedA == anotherBoxedA) // Prints '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's `<<`)
- `shr(bits)` – 有符号右移 (Java's `>>`)
- `ushr(bits)` – 无符号右移 (Java's `>>>`)
- `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 中数组是不协变的（invariant）。这意味着 Kotlin 不让我们把 `Array<String>` 赋值给 `Array<Any>`，以防止可能的运行时失败（但是你可以使用 `Array<out Any>`，参见 [类型预测](#)）。

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，以及原生字符串白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)
    """
```

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

// 有 else
var max: Int
if (a > b)
    max = a
else
    max = b

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

`if` 的分支可以是代码块，最后的表达式作为该块的值：

```
val max = if (a > b) {
    print("Choose a")
    a
} else {
    print("Choose b")
    b
}
```

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

参见 [if 语法](#)。

When表达式

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

```
when (x) {
    1 -> print("x == 1")
    2 -> print("x == 2")
    else -> { // 注意这个块
        print("x is neither 1 nor 2")
    }
}
```

`when` 将它的参数和所有的分支条件顺序比较，直到某个分支满足条件。`when` 既可以被当做表达式使用也可以被当做语句使用。如果它被当做表达式，符合条件的分支的值就是整个表达式的值，如果当做语句使用，则忽略个别分支的值。（像 `if` 一样，每一个分支可以是一个代码块，它的值是块中最后的表达式的值。）

如果其他分支都不满足条件将会求值 `else` 分支。如果 `when` 作为一个表达式使用，则必须有 `else` 分支，除非编译器能够检测出所有的可能情况都已经覆盖了。

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

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

我们可以用任意表达式（而不只是常量）作为分支条件

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

我们也可以检测一个值在 (`in`) 或者不在 (`!in`) 一个[区间](#)或者集合中：

```
when (x) {
  in 1..10 -> print("x is in the range")
  in validNumbers -> print("x is valid")
  !in 10..20 -> print("x is outside the range")
  else -> print("none of the above")
}
```

另一种可能性是检测一个值是 (`is`) 或者不是 (`!is`) 一个特定类型的值。注意：由于[智能转换](#)，你可以访问该类型的方法和属性而无需任何额外的检测。

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

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

```
when {
  x.isOdd() -> print("x is odd")
  x.isEven() -> print("x is even")
  else -> print("x is funny")
}
```

参见 [when 语法](#)。

For循环

`for` 循环可以对任何提供迭代器（iterator）的对象进行遍历，语法如下：

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

循环体可以是一个代码块。

```
for (item: Int in ints) {
    // ...
}
```

如上所述，`for` 可以循环遍历任何提供了迭代器的对象。即：

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

这三个函数都需要标记为 `operator`。

对数组的 `for` 循环会被编译为并不创建迭代器的基于索引的循环。

如果你想要通过索引遍历一个数组或者一个 list，你可以这么做：

```
for (i in array.indices)
    print(array[i])
```

注意这种“在区间上遍历”会编译成优化的实现而不会创建额外对象。

或者你可以用库函数 `withIndex`：

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

参见 [for 语法](#)。

While循环

`while` 和 `do..while` 照常使用

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

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

参见 [while 语法](#).

循环中的Break和continue

在循环中 Kotlin 支持传统的 `break` 和 `continue` 操作符。参见[返回和跳转](#)。

返回和跳转

Kotlin 有三种结构化跳转操作符

- `return` 默认从最直接包围它的函数或者[匿名函数](#)返回。
- `break` 终止最直接包围它的循环。
- `continue` 继续下一次最直接包围它的循环。

Break和Continue标签

在 Kotlin 中任何表达式都可以用标签（[label](#)）来标记。标签的格式为标识符后跟 `@` 符号，例如：`abc@`、`fooBar@` 都是有效的标签（参见[语法](#)）。要为一个表达式加标签，我们只要在其前加标签即可。

```
loop@ for (i in 1..100) {  
    // ...  
}
```

现在，我们可以用标签限制 `break` 或者 `continue`：

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

标签限制的 `break` 跳转到刚好位于该标签指定的循环后面的执行点。`continue` 继续标签指定的循环的下一次迭代。

标签处返回

Kotlin 有函数字面量、局部函数和对象表达式。因此 Kotlin 的函数可以被嵌套。标签限制的 `return` 允许我们从外层函数返回。最重要的一个用途就是从 lambda 表达式中返回。回想一下我们这么写的时候：

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

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

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

现在，它只会从 lambda 表达式中返回。通常情况下使用隐式标签更方便。该标签与接受该 lambda 的函数同名。

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

或者，我们用一个[匿名函数](#)替代 lambda 表达式。匿名函数内部的 `return` 语句将从该匿名函数自身返回

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

当要返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；如果这个类没有类体，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) {  
}
```

如果这个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) {  
}
```

这个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。

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互通性](#)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{.keyword}` 关键字初始化其基类型，或委托给另一个构造函数做到这一点。注意，在这种情况下，不同的次构造函数可以调用基类型的不同的构造函数：

```
class MyView : View {
    constructor(ctx: Context) : super(ctx) {
    }

    constructor(ctx: Context, attrs: AttributeSet) : super(ctx, attrs) {
    }
}
```

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

覆盖成员

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

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

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

标记为 `override{}` 的成员本身是开放的，也就是说，它可以在子类中覆盖。如果你想禁止再次覆盖，使用 `final{}` 关键字：

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

等等！这样我怎么 hack 我的库？

我们这样设计继承和覆盖的方式(类和成员默认 `final`)，会让人很难继承第三方的类，因此很难进行 hack。

我们认为这不是一个劣势，原因如下：

- 最佳实践已经表明不应该使用这些 hacks
- 其他的有类似机制的语言（C++、C#）已经证明是成功的
- 如果人们实在想 hack，仍然有办法：你总可以使用 Java 进行 hack 再用 Kotlin 调用它（参见 [Java 互操作](#)），另外切面（Aspect）框架就是以此为目的的。

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重实现，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{.keyword}`。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. 这些声明是可变的,用关键字`var`或者使用只读关键字`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, 只需要使用名称引用即可, 就相当于Java中的公共字段:

```
fun copyAddress(address: Address): Address {  
    val result = Address() // there's no 'new' keyword in Kotlin  
    result.name = address.name // accessors are called  
    result.street = address.street  
    // ...  
    return result  
}
```

Getters 和 Setters

声明一个属性的完整语法

```
var <propertyName>: <PropertyType> [= <property_initializer>]  
    [<getter>]  
    [<setter>]
```

上面的定义中, 初始器(initializer)、getter 和 setter 都是可选的。属性类型(PropertyType)如果可以从初始器或者父类中推导出来, 也可以省略。

例如:

```
var allByDefault: Int? // error: explicit initializer required, default getter and  
    setter implied  
var initialized = 1 // has type Int, default getter and setter
```

~~注意公有的API(即**public**和**protected**)的属性, 类型是不做推导的。~~这么设计是为了防止改变初始器时不小心改变了公有API。比如: ~~

```
public val example = 1 // error: a public property must have a type specified  
explicitly
```

一个只读属性的语法和一个可变的语法有两方面的不同: 1、只读属性的用`val`开始代替`var` 2、只读属性不许 setter

```
val simple: Int? // has type Int, default getter, must be initialized in constructor
val inferredType = 1 // has type Int and a default getter
```

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

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

一个定义setter的例子:

```
var stringRepresentation: String
    get() = this.toString()
    set(value) {
        setDataFromString(value) // parses the string and assigns values to other properties
    }
```

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或注释的可见性,但是不需要改变默认的实现,您可以定义访问器而不定义它的实例:

```
var setterVisibility: String = "abc"
    private set // the setter is private and has the default implementation

var setterWithAnnotation: Any? = null
    @Inject set // annotate the setter with Inject
```

实际字段

在Kotlin不能有字段。然而,有时有必要使用一个字段在使用定制的访问器的时候。对于这些目的,Kotlin提供 自动支持,在属性名后面使用 `field` 标识符。

```
var counter = 0 // the initializer value is written directly to the backing field
    set(value) {
        if (value >= 0)
            field = value
    }
```

`field` 标识符只能用在属性的访问器内。

编译器会查看访问器的内部,如果他们使用了实际字段(或者访问器使用默认实现),那么将会生成一个实际字段,否则不会生成。

例如,下面的情况下,就没有实际字段:

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

支持属性

如果你的需求不符合这套“隐式的实际字段”方案，那么总可以使用“后背支持属性”(backing property)的方法：

```
private var _table: Map<String, Int>? = null
public val table: Map<String, Int>
    get() {
        if (_table == null)
            _table = HashMap() // Type parameters are inferred
        return _table ?: throw AssertionError("Set to null by another thread")
    }
```

从各种角度看，这和Java中定义Bean属性的方式一样。因为访问私有的属性的getter和setter函数，无寒函数调用开销。

Compile-Time Constants

Properties the value of which is known at compile time can be marked as *compile time constants* using the `const` modifier. Such properties need to fulfil the following requirements:

- Top-level or member of an `object`
- Initialized with a value of type `String` or a primitive type
- No custom getter

Such properties can be used in annotations:

```
const val SUBSYSTEM_DEPRECATED: String = "This subsystem is deprecated"

@Deprecated(SUBSYSTEM_DEPRECATED) fun foo() { ... }
```

Late-Initialized Properties

Normally, properties declared as having a non-null type must be initialized in the constructor. However, fairly often this is not convenient. For example, properties can be initialized through dependency injection, or in the setup method of a unit test. In this case, you cannot supply a non-null initializer in the constructor, but you still want to avoid null checks when referencing the property inside the body of a class.

To handle this case, you can mark the property with the `lateinit` modifier:

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

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

    @Test fun test() {
        subject.method() // dereference directly
    }
}
```

The modifier can only be used on `var` properties declared inside the body of a class (not in the primary constructor), and only when the property does not have a custom getter or setter. The type of the property must be non-null, and it must not be a primitive type.

Accessing a `lateinit` property before it has been initialized throws a special exception that clearly identifies the property being accessed and the fact that it hasn't been initialized.

重写属性

查看 [Overriding Members](#)

委托属性

从支持域最常见类型的属性只读(写入)。另一方面,使用自定义getter和setter属性可以实现任何方法行为。介于两者之间,有一些常见的模式属性是如何工作的。一个例子:lazy values,从映射读取关键字,访问一个数据库,访问通知侦听器,等等。

像常见的行为可以从函数库调用像[*delegated properties*](#)。

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 `abstract`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() {  
        // optional body  
    }  
}
```

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() {  
        // body  
    }  
}
```

接口属性

You can declare properties in interfaces. A property declared in an interface can either be abstract, or it can provide implementations for accessors. Properties declared in interfaces can't have backing fields, and therefore accessors declared in interfaces can't reference them.

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

解决重写 (Override) 冲突

实现多个接口时，可能会遇到接口方法名同名的问题。

```

interface A {
    fun foo() { print("A") }
    fun bar()
}

interface B {
    fun foo() { print("B") }
    fun bar() { print("bar") }
}

class C : A {
    override fun bar() { print("bar") }
}

class D : A, B {
    override fun foo() {
        super<A>.foo()
        super<B>.foo()
    }
}

```

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 *abstract*，因为没有方法体时默认为 *abstract*)。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，构造方法，和它们的setter方法都可以用_visibility modifiers_来做修饰。(getter一直与属性有着相同的可见性.)

在Kotlin中有以下四个可见性修饰符：

- `private` — 只有在声明的范围及其方法可见(在同一模块);
- `protected` — (只适用于类/接口成员)和“private”一样,但也在子类可见;
- `internal` — (在默认情况下使用)在同一个模块中可见(如果声明范围的所有者是可见的);
- `public` — 随处可见(如果声明范围的所有者是可见的).

注意: 函数 *with expression bodies* 所有的属性声明 `public` 必须始终显式指定返回类型。这是必需的，这样我们就不会随意改变一个类型,仅通过改变实现公共API的一部分。

```
public val foo: Int = 5    // explicit return type required
public fun bar(): Int = 5  // explicit return type required
public fun bar() {}       // block body: return type is Unit and can't be changed
                           accidentally, so not required
```

下面将解释不同类型的声明范围。

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：

```
// file name: example.kt
package foo

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

- 如果你不指定任何可见性修饰符，那么默认情况下使用 `internal` 修饰，这意味着你们将声明在同一个模块中可见;
- 如果你声明 `private`，只会是这个包及其子包内可见的，并且只在相同的模块;
- 如果你声明 `public`，随处可见。
- `protected` 不适用于顶层声明。

例子：

```
// file name: example.kt
package foo

private fun foo() {} // visible inside this package and subpackaged

public var bar: Int = 5 // property is visible everywhere
    private set         // setter is visible only in this package and subpackages

internal val baz = 6    // visible inside the same module, the modifier can be omitted
```

类和接口

当一个类中声明：

- `private` 意味着这个类只在内部可见(包含所有成员)。
- `protected` — 和 `private` 一样+在子类可见。
- `internal` — 任何客户端 *inside this module* 谁看到声明类，其 `internal` 成员在里面；
- `public` — 任何客户端看到声明类看到其 `public` 成员。

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

例子:

```
open class Outer {
    private val a = 1
    protected val b = 2
    val c = 3 // internal by default
    public val d: Int = 4 // return type required

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

class Subclass : Outer() {
    // a is not visible
    // b, c and d are visible
    // Nested and e are visible
}

class Unrelated(o: Outer) {
    // o.a, o.b are not visible
    // o.c and o.d are visible (same module)
    // Outer.Nested is not visible, and Nested::e is not visible either
}
```

构造函数

指定一个类的可见性的主构造函数,使用以下语法(注意你需要添加一个显示构造函数{::keyword} keyword):

```
class C private constructor(a: Int) { ... }
```

这里的构造函数是私有的。不像其他的声明，在默认情况下，所有构造函数是 `public`，这实际上等于他们是随处可见，其中的类是可见(即内部类的构造函数是唯一可见在同一模块内)。

局部声明

局部变量，函数和类不能有可见性修饰符。

Modules

The `internal` visibility modifier means that the member is visible with the same module. More specifically, a module is a set of Kotlin files compiled together:

- an IntelliJ IDEA module;
- a Maven or Gradle project;
- a set of files compiled with one invocation of the `Ant` task.

扩展

Kotlin和c#、Gosu一样，能够扩展一个类的新功能，而无需继承类或使用任何类型的设计模式，如装饰者。这通过特殊的声明叫做`_extensions_`。Kotlin支持`_extension functions_` 和 *extension properties*.

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。下面是为 `MutableList<Int>` 添加一个 `swap` 方法：

```
fun MutableList<Int>.swap(index1: Int, index2: Int) {
    val tmp = this[index1] // 'this' corresponds to the list
    this[index1] = this[index2]
    this[index2] = tmp
}
```

这个`this`关键字在扩展方法内接受对应的对象（在点符号以前传过来的）现在，我们可以像一个其他方法一样调用 `MutableList<Int>`：

```
val l = mutableListOf(1, 2, 3)
l.swap(0, 2) // 'this' inside 'swap()' will hold the value of 'l'
```

当然，这个方法像这样 `MutableList<T>`，我们可以使用泛型：

```
fun <T> MutableList<T>.swap(index1: Int, index2: Int) {
    val tmp = this[index1] // 'this' corresponds to the list
    this[index1] = this[index2]
    this[index2] = tmp
}
```

在接收类型表达式中，我们要在方法名可用前声明泛型类型参数。参见[Generic functions](#).

扩展的静态解析

扩展不能真正的修改他们继承的类。通过定义一个扩展，你不能在类内插入新成员，仅仅是通过该类的实例用点表达式去调用这个新函数。

我们想强调下扩展方法是被静态分发的，即他们不是接收类型的虚方法。This means that the extension function being called is determined by the type of the expression on which the function is invoked, not by the type of the result of evaluating that expression at runtime. For example:

```

open class C

class D: C()

fun C.foo() = "c"

fun D.foo() = "d"

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

printFoo(D())

```

This example will print “c”, because the extension function being called depends only on the declared type of the parameter `c`, which is the `C` class.

If a class has a member function, and an extension function is defined which has the same receiver type, the same name and is applicable to given arguments, the **member always wins**. For example:

```

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

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

```

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

Nullable接收者

注意扩展可被定义为可空的接收类型。这样的扩展可以被对象变量调用，即使他的值是null，你可以在方法体内检查 `this == null`，这也允许你 在没有检查null的时候调用Kotlin中的toString(): 检查发生在扩展方法的内部的时候

```

fun Any?.toString(): String {
    if (this == null) return "null"
    // after the null check, 'this' is autocast to a non-null type, so the toString()
    below
    // resolves to the member function of the Any class
    return toString()
}

```

扩展属性

和方法相似，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，因此对扩展来说是无效的 属性是有[backing field](#).这就是为什么初始化其不允许有 扩展属性。他们的行为只能显式的使用 getters/setters.

例子:

```
val Foo.bar = 1 // error: initializers are not allowed for extension properties
```

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 { } // will be called "Companion"  
}  
  
fun MyClass.Companion.foo() {  
    // ...  
}
```

就像伴生对象的其他普通成员，只用用类名作为限定符去调用他们

```
MyClass.foo()
```

扩展范围

大多数时候，我们定义扩张方法在顶层，即直接在包里

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

使用一个定义的包之外的扩展，我们需要导入他的头文件：

```
package com.example.usage  
  
import foo.bar.goo // importing all extensions by name "goo"  
// or  
import foo.bar.* // importing everything from "foo.bar"  
  
fun usage(baz: Baz) {  
    baz.goo()  
}
```

更多信息参见[Imports](#)

Declaring Extensions as Members

Inside a class, you can declare extensions for another class. Inside such an extension, there are multiple *implicit receivers* - objects members of which can be accessed without a qualifier. The instance of the class in which the extension is declared is called *dispatch receiver*, and the instance of the receiver type of the extension method is called *extension receiver*.

```
class D {  
    fun bar() { ... }  
}  
  
class C {  
    fun baz() { ... }  
  
    fun D.foo() {  
        bar()    // calls D.bar  
        baz()    // calls C.baz  
    }  
  
    fun caller(d: D) {  
        d.foo()  // call the extension function  
    }  
}
```

In case of a name conflict between the members of the dispatch receiver and the extension receiver, the extension receiver takes precedence. To refer to the member of the dispatch receiver you can use the [qualified this syntax](#).

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

Extensions declared as members can be declared as `open` and overridden in subclasses. This means that the dispatch of such functions is virtual with regard to the dispatch receiver type, but static with regard to the extension receiver type.

```

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() // call the extension function
    }
}

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

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

C().caller(D()) // prints "D.foo in C"
C1().caller(D()) // prints "D.foo in C1" - dispatch receiver is resolved virtually
C().caller(D1()) // prints "D.foo in C" - extension receiver is resolved statically

```

Motivation

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-classes的不愉快的部分是这样写代码的：

```

// 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得到：

```

// 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。我们希望它能更好点

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

但是我们不希望实现 List 类内所有可能的方法，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是处理数据的类。在这些类里的标准功能经常是 衍生自数据。在Kotlin中，这叫做 *数据类* 并标记为 `data`：

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

编译器自动从在主构造函数定义的全部特性中得到以下成员：

- `equals()` / `hashCode()` 对，
- `toString()` 格式是 `"User(name=John, age=42)"`，
- [componentN\(\) functions](#) 对应按声明顺序出现的所有属性，
- `copy()` 函数（见下面）。

如果有某个函数被明确地定义在类里或者被继承，编译器就不会生成这个函数。

To ensure consistency and meaningful behavior of the generated code, data classes have to fulfil the following requirements:

- The primary constructor needs to have at least one parameter;
- All primary constructor parameters need to be marked as `val` or `var`;
- Data classes cannot be abstract, open, sealed or inner;
- Data classes may not extend other classes (but may implement interfaces).

在JVM中，如果生成的类需要含有一个无参的构造函数，则所有的属性必须有默认值。（查看 [Constructors](#)）。

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

复制

在很多情况下，我们我们需要对一些属性做修改而其他的不变。这就是 `copy()` 这个方法的来源。对于上文的 `User` 类，应该是这么实现这个方法的

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

也可以这么写

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

数据类和多重声明

`_成员方法_`用于使数据类可以[多声明](#)：

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

标准数据类

在标准库提供了 `Pair` 和 `Triple`。在很多情况下，即使命名数据类是一个更好的设计选择，因为这能让代码可读性更强。

泛型

与Java相似，Kotlin中的类也具有类型参数，如：

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

一般而言，创建类的实例时，我们需要声明参数的类型，如：

```
val box: Box<Int> = Box<Int>(1)
```

但当参数类型可以从构造函数参数等途径推测时，在创建的过程中可以忽略类型参数：

```
val box = Box(1) // 1 has type Int, so the compiler figures out that we are talking  
about Box<Int>
```

型变Variance

Java的变量类型中，最为精妙的是通配符（wildcards）类型(详见 [Java Generics FAQ](#))。但是Kotlin中并不具备该类型，替而代之的是：声明设置差异（declaration-site variance）与类型推测。

首先，我们考虑一下Java中的通配符（wildcards）的意义。该问题在文档 [Effective Java, Item 28: Use bounded wildcards to increase API flexibility](#)中给出了详细的解释。首先，Java中的泛型类型是不变的，即 `List<String>` 并不是 `List<Object>` 的子类型。原因在于，如果List是可变的，并不会 优于Java数组。因为如下代码在编译后会产生运行时异常：

```
// Java  
List<String> strs = new ArrayList<String>();  
List<Object> objs = strs; // !!! The cause of the upcoming problem sits here. Java  
prohibits this!  
objs.add(1); // Here we put an Integer into a list of Strings  
String s = strs.get(0); // !!! ClassCastException: Cannot cast Integer to String
```

因此，Java规定泛型类型不可变来保证运行时的安全。但这样规定也具有有一些影响。如，`Collection` 接口中的 `addAll()` 方法，该方法的签名应该是什么？直观地，我们这样定义：

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

但随后，我们便不能实现以下肯定安全的事：

```
// Java
void copyAll(Collection<Object> to, Collection<String> from) {
    to.addAll(from); // !!! Would not compile with the naive declaration of addAll:
                    //      Collection<String> is not a subtype of Collection<Object>
}
```

(更详细的解析参见[Effective Java](#), Item 25: *Prefer lists to arrays*)

以上正是为什么 `addAll()` 的方法签名如下的原因：

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

通配符类型 (wildcard) 的声明 `? extends T` 表明了该方法允许一类对象是 `T` 的子类型，而非必须得是 `T` 本身。这意味着我们可以安全地从元素 (`T` 的子类集合中的元素) 读取 `T`，同时由于我们并不知道 `T` 的子类型，所以不能写元素。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 (上限) 的通配符类型 (wildcard) 使得类型是**协变的 (covariant)**。

理解为什么这样做可以使得类型的表达更加简单的关键点在于：如果只能从集合中获取元素，那么就可以使用 `String` 集合，从中读取 `Object` 也没问题。反过来，如果只能向集合中放入元素，就可以用 `Object` 集合并向其中放入 `String`：在Java中有 `List<? super String>` 是 `List<Object>` 的超类。

后面的情况被称为“**抗变性 (contravariance)**”，这种性质是只可以调用方法时利用 `String` 为 `List<? super String>` 的参数。（例如，可以调用 `add(String)` 或者 `set(int, String)`），或者当调用函数返回 `List<T>` 中的 `T`，你获取的并非一个 `String` 而是一个 `Object`。

Joshua Bloch 成这类为只可以从 **Producers (生产者)** 处读取的对象，以及只可向 **Consumers (消费者)** 处写的对象。他表示：“为了最大化地保证灵活性，在输入参数时使用通配符类型来代表生产者或者消费者”，同时他也提出了以下术语：

PECS 代表生产者扩展，消费者超类 (roducer-Extends, Consumer-Super)。

注记：当使用一个生产者对象时，如 `List<? extends Foo>`，在该对象上不可调用 `add()` 或 `set()` 方法。但这不代表该对象是不变的。例如，可以调用 `clear()` 方法移除列表里的所有元素，因为 `clear()` 方法不含任何参数。通配符类型唯一保证的仅仅是**类型安全**。不可变性完全是另一个话题了。

声明处型变

假设有一个泛型接口 `Source<T>`，该接口中不存在将 `T` 作为参数的方法，只有返回值为 `T` 的方法：

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

那么，利用 `Source<Object>` 类型的对象向 `Source<String>` 实例中存入引用是极为安全的，因为不存在任何可以调用的消费者方法。但是Java并不知道这点，依旧禁止这样操作：

```
// Java
void demo(Source<String> strs) {
    Source<Object> objects = strs; // !!! Not allowed in Java
    // ...
}
```

为了修正这一点，我们需要声明对象的类型为 `Source<? extends Object>`，有一点无意义，因为我们可以像以前一下在该对象上调用所有相同的方法，所以复杂类型没有增加值。但是编译器并不可理解。

在Kotlin中，我们有一种途径向编译器解释该表达，称之为：**声明处型变**：我们可以标注源的**变量类型**为 `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**输出**修改：

```
abstract class Source<out T> {
    abstract fun nextT(): T
}

fun demo(strs: Source<String>) {
    val objects: Source<Any> = strs // This is OK, since T is an out-parameter
    // ...
}
```

常规是：当一个类 `C` 中类型为 `T` 的变量被声明为**输出**，它将仅出现于类 `C` 的成员**输出**-位置，反之使得 `C<Base>` 可以安全地成为 `C<Derived>` 的超类。

简而言之，称类 `C` 是参数 `T` 中的**协变的**，或 `T` 是一个**协变的**参数类型。我们可以认为 `C` 是 `T` 的一个生产者，同时不是 `T` 的消费者。

out修饰符叫做**型变注解**，同时由于它在参数类型位置被提供，所以我们讨论**声明处型变**。与Java的**使用处型变**相反，类型使用通配符使得类型协变。

另外除了**out**，Kotlin又补充了一项型变注解：**in**。它是的变量类型**反变**：只可以被消费而不可以 被生产。反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 has type Double, which is a subtype of Number
    // Thus, we can assign x to a variable of type Comparable<Double>
    val y: Comparable<Double> = x // OK!
}
```

我们认为**in**和**out**是自解释（他们早已成功地被应用于C#中），所以上文的解释并非必须的，并且读者可以从[The Existential Transformation: Consumer in, Producer out!](#) 中获取更加深入的理解:~)。

类型预测

使用处型变：类型预测

声明变量类型T为out是极为方便的，并且在运用子类型的过程中也没有问题。是的，当该类可以被仅限于返回T，但是如果不可以呢？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) { /* ... */ }
}
```

该类中的T 不仅不可以被co- 也不能被逆变。这造成了极大的不灵活性。考虑该方法：

```
fun copy(from: Array<Any>, to: Array<Any>) {
    assert(from.size == to.size)
    for (i in from.indices)
        to[i] = from[i]
}
```

该方法试图从一个数组中copy元素到另一个数组。我们尝试着在实际中运用它：

```
val ints: Array<Int> = arrayOf(1, 2, 3)
val any = Array<Any>(3)
copy(ints, any) // Error: expects (Array<Any>, Array<Any>)
```

这里，我们陷入了一个类似的问题：Array<T> 中的T 是不变的，所以不论是Array<Int> 或Array<Any> 都不是另一个的子类型。为什么？因为copy操作可能是不安全的行为，例如，它可能尝试向来源写一个String，如果我们真的将其转换为Int 数组，随后ClassCastException 异常可能会被抛出。

那么，我们唯一需要确保的是 copy() 不会执行任何不安全的操作。我们试图阻止它向来源写，我们可以：

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

我们称该做法为类型预测：源 并不仅是一个数组，并且可以要是可预测的。我们仅可调用返回类型为T 的方法，如上，我们只能调用 get() 方法。这就是我们使用使用位置可变性而非Java中的Array<? extends Object> 的更加明确简单的方法

同时，你也可以利用in预测输入类型：

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

Array<in String> 比对于Java中的Array<? super String>，例如，你可以向fill() 方法传递一个CharSequence 数组或者一个Object 数组。

星-预测 Star-Projections

有时，你试图说你并不知道任何类型声明的方法，但是仍旧想安全地使用他。这里的安全方法指我们需要对`out`-预测

Kotlin提供了所谓的星预测语法如下：

- For `Foo<out T>`, where `T` is a covariant type parameter with the upper bound `TUpper`, `Foo<*>` is equivalent to `Foo<out TUpper>`. It means that when the `T` is unknown you can safely *read* values of `TUpper` from `Foo<*>`.
- For `Foo<in T>`, where `T` is a contravariant type parameter, `Foo<*>` is equivalent to `Foo<in Nothing>`. It means there is nothing you can *write* to `Foo<*>` in a safe way when `T` is unknown.
- For `Foo<T>`, where `T` is an invariant type parameter with the upper bound `TUpper`, `Foo<*>` is equivalent to `Foo<out TUpper>` for reading values and to `Foo<in Nothing>` for writing values.

If a generic type has several type parameters each of them can be projected independently. For example, if the type is declared as `interface Function<in T, out U>` we can imagine the following star-projections:

- `Function<*, String>` means `Function<in Nothing, String>`;
- `Function<Int, *>` means `Function<Int, out Any?>`;
- `Function<*, *>` means `Function<in Nothing, out Any?>`.

注记：星预测很像Java中的raw类型，但是比raw类型更加安全。

泛型函数

不仅仅是类可以有类型参数，函数也可以有。类型参数放置在函数名前：

```
fun <T> singletonList(item: T): List<T> {  
    // ...  
}  
  
fun <T> T.basicToString() : String { // extension function  
    // ...  
}
```

如果参数类型是从调用方显式传递而来，那么它要在函数名之后被声明：

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

泛型约束

集合的所有可能类型可以被给定的被约束的泛型约束参数类型替代。

上界

约束最常见的类型是上界相比较于Java中的`extends`关键字：


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

在冒号之后被声明的是**上界**：代替 `T` 的仅可为 `Comparable<T>` 的子类型。例如：

```
sort(listOf(1, 2, 3)) // OK. Int is a subtype of Comparable<Int>
sort(listOf(HashMap<Int, String>())) // Error: HashMap<Int, String> is not a subtype of
Comparable<HashMap<Int, String>>
```

默认的上界（如果没有声明）是 `Any?`。只能有一个上界可以在尖括号中被声明。如果相同的类型参数需要多个上界，我们需要分割符 **where**-子句，如：

```
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

在类的内部可以嵌套其他的类

```
class Outer {  
    private val bar: Int = 1  
    class Nested {  
        fun foo() = 2  
    }  
}  
  
val demo = Outer.Nested().foo() // == 2
```

内部类

为了能被外部类访问一个类可以被标记为内部类（“inner” 关键词）。内部类会带有一个来自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
```

参阅[this-expressions.html](https://www.scala-lang.org/doc/2.12/this-expressions.html)中“this”关键词用法来学习在内部类中如何消除“this”关键词的歧义。

枚举类

枚举类的最基本应用是实现类型安全的多项目集合。

```
enum class Direction {  
    NORTH, SOUTH, WEST, EAST  
}
```

其中每一个常量（NORTH, SOUTH.....）都是一个对象。每一个常量用逗号“,”分隔。

初始化

因为每一条枚举（RED, GREEN.....）都是枚举类的实例，所以他们可以被初始化。

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

匿名类

枚举实例也可以被声明为他们自己的匿名类，并同时包含他们相应原本的方法和覆盖基本方法。

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

注意如果枚举类定义了任何成员，你需要像JAVA一样把枚举实例的定义和成员定义用分号分开。

枚举实例的用法

像JAVA一样，枚举类在Kotlin中有合成方法。它允许列举枚举实例并且通过名称返回枚举实例。下面是应用实例 (假设枚举实例名称是 EnumClass):

```
EnumClass.valueOf(value: String): EnumClass  
EnumClass.values(): Array<EnumClass>
```

如果特定的对象名无法对应任何一个定义在枚举类中的枚举常量，`valueOf()` 方法会抛出一个异常 `IllegalArgumentException`。

每一个枚举常量在枚举类定义时都有一个属性去获得他们的名字和位置。

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

枚举常量也可以实现[Comparable](#) 接口。他们会依照在枚举类中的定义先后以自然顺序排列。

对象表达式和对象声明

有些时候我们需要创建一个对某些类做了轻微改变的一个对象，而不用为了它显式地定义一个新的子类。Java把这种情况处理为*匿名内部类*。在Kotlin稍微推广了这个概念，称它们为*对象表达式*和*对象声明*。

对象表达式

创建一个继承自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) {  
        // ...  
    }  
})
```

如果父类型有一个构造函数，合适的构造函数参数必须传递给它。多个父类型用逗号隔开，跟在冒号后面：

```
open class A(x: Int) {  
    public open val y: Int = x  
}  
  
interface B {...}  
  
val ab = object : A(1), B {  
    override val y = 15  
}
```

或许，我们需要的仅是无父类的一个对象，那么我们可以简单地写为：

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

就像Java的匿名内部类，在对象表达式里代码可以访问封闭的作用域（但与Java不同的是，它能访问非final修饰的变量）

```

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，而在Kotlin（在Scala之后）中很容易就能声明一个单例。

```

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

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

```

这被称为*对象声明*。如果有一个`object`关键字在名字前面，这不能再被称为一个_表达式_。我们不能把这样的东西赋值给变量，但我们可以通过它的名字来引用它。这样的对象可以有父类型：

```

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

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

```

NOTE: 对象声明不能是本地的（即直接嵌套在函数里面），但它们可以被嵌套进另外的对象声明或者非内部类里。

伴生对象

一个对象声明在一个类里可以标志上`companion`这个关键字：

```
class MyClass {
    companion object Factory {
        fun create(): MyClass = MyClass()
    }
}
```

伴生对象的成员可以使用类名称作为限定符来调用：

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

使用 `companion` 关键字时候，伴生对象的名称可以省略：

```
class MyClass {
    companion object {
    }
}

val x = MyClass.Companion
```

注意，虽然伴生对象的成员在其他语言中看起来像静态成员，但在运行时它们 仍然是实体的实例成员，举例来说，我们能用它实现接口：

```
interface Factory<T> {
    fun create(): T
}

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

然而，在JVM中，如果你使用 `@JvmStatic` 注解，你可以让伴生对象的成员生成为实际存在的静态方法和域。可以从[Java interoperability](#) 这里 查看详情。

对象表达式与对象声明语义上的不同

这是一个在对象表达式与对象声明上重要的不同之处：

- 当对象声明被第一次访问的时候,它会被**延迟 (lazily)** 初始化
- 当对象表达式被用到的时候，它会被**立即执行**（并且初始化）

委托属性

有一些种类的属性，虽然我们可以在每次需要的时候手动实现它们，但是如果能够把他们只实现一次 并放入一个库 同时又能够一直使用它们那会更好。例如：

- 延迟属性 (lazy properties) : 数值只在第一次被访问的时候计算。
- 可观察属性 (observable properties) : 监听器得到关于这个特性变化的通知,
- 把所有属性储存在一个map中，而不是每个在单独的字段里。

为了支持这些(或者其他)例子，Kotlin 采用 *委托属性*：

```
class Example {  
    var p: String by Delegate()  
}
```

语法是: `val/var <property name>: <Type> by <expression>`. 在 `by` 后面的表达式是 *委托*, 因为 `get()` (和 `set()`) 相当于属性会被委托给它的 `getValue()` 和 `setValue()` 方法。特性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`'s)。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.")  
    }  
}
```

当我们读取一个 `Delegate` 的委托实例 `p`，`Delegate` 中的 `getValue()` 就被调用，所以它第一变量就是从 `p` 读取的实例，第二个变量代表 `p` 自身的描述。(例如你可以用它的名字)。下面是例子：

```
val e = Example()  
println(e.p)
```

打印结果：

Example@33a17727, thank you for delegating 'p' to me!

类似的，当我们给 `p` 赋值，`setValue()` 函数就被调用。前两个参数是一样的，第三个参数保存着将要被赋予的值：

```
e.p = "NEW"
```

打印结果：

NEW has been assigned to 'p' in Example@33a17727.

属性委托要求

这里我们总结委托对象的要求。

对于一个 **只读** 属性 (如 `val`), 一个委托一定会提供一个 `getValue` 函数来获取下面的参数:

- 接收者 — 必须与_属性所有者_类型相同或者是其父类(对于扩展属性, 类型范围允许扩大),
- 包含数据 — 一定要是 `KProperty<*>` 的类型或它的父类型,

这个函数必须返回同样的类型作为属性 (或者子类型)

对于一个 **可变** 属性 (如 `var`), 一个委托需要额外地提供一个函数 `setValue` 来获取下面的参数:

- 接收者 — 同 `getValue()`,
- 包含数据 — 同 `getValue()`,
- 新的值 — 必须和属性同类型或者是他的父类型。

`getValue()` 或/和 `setValue()` 函数可能会作为代理类的成员函数或者扩展函数来提供。当你需要代理一个属性给一个不是原来就提供这些函数的对象的时候, 后者更为方便。两种函数都需要用 `operator` 关键字来进行标记

标准委托

标准库中对于一些有用的委托提供了工厂 (factory) 方法。

延迟属性 Lazy

函数 `lazy()` 接受一个 lambda 然后返回一个可以作为实现延迟属性的委托 `Lazy<T>` 实例来: 第一次对于 `get()` 的调用会执行 (之前) 传递到 `lazy()` 的 lambda 表达式并记录结果, 后面的 `get()` 调用会直接返回记录的结果。

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

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

默认地, 对于 lazy 属性的计算是 **同步锁 (synchronized)** 的: 这个值只在一个线程被计算, 并且所有的线程会看到相同的值。如果初始化代理的同步锁不是必须的, 以至于多个线程可以同步地执行, 那么将 `LazyThreadSafetyMode.PUBLICATION` 作为一个变量传递给 `lazy()` 函数。而且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()` 需要两个参数: 初始值和 handler。这个 handler 会在每次我们给赋值的时候被调用 (在工作完成前)。它有三个参数: 一个被赋值的属性, 旧的值和新的值:

```
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
```

如果你想有能力来截取和“否决”它分派的事件，就使用 `vetoable()` 取代 `observable()`。被传递给 `vetoable` 的 handler 会在属性被赋新的值_之前_执行

~~### 非空 Not-Null~~

有时候我们有一个非空的值 `var`，但是我们却没有合适的值去给构造器去初始化。例如，它必须被之后初始化。问题是在 Kotlin 中你不能有一个没有被初始化的非抽象属性：

```
class Foo {
    var bar: Bar // ERROR: must be initialized
}
```

我们可以用 `null` 去初始化它，但是我们不得不在每次使用前检查一下。

`Delegates.notNull()` 可以解决这个问题：

```
class Foo {
    var bar: Bar by Delegates.notNull()
}
```

如果这个属性在首次写入前进行读取，它就会抛出一个异常，写入后就正常了。

把属性储存在map中

一个参加的用例是在一个 map 里存储属性的值。这经常出现在解析 JSON 或者做其他的“动态”的事情应用里头。在这样的情况下，你需要使用 map 的实例本身作为代理用于代理属性

```
class User(val map: Map<String, Any?>) {
    val name: String by map
    val age: Int by map
}
```

在这个例子中，构造函数会接收一个map参数：

```
val user = User(mapOf(
    "name" to "John Doe",
    "age" to 25
))
```

委托会从这个map中取值 (通过string类型的key，就是属性的名字):

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

对于 `var`的变量，我们可以把只读的 `Map` 换成 `MutableMap` 就可以了

```
class MutableUser(val map: MutableMap<String, Any?>) {
    var name: String by map
    var age: Int by map
}
```

函数和Lambdas表达式

函数

函数声明

在Kotlin中，函数声明使用关键字 `fun`

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

函数用途

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达式

```
Sample().foo() // create instance of class Sample and calls foo
```

中缀表示法

函数还可以用中缀表示法，当

- 他们是成员函数 或者 [扩展函数](#)
- 他们只有一个参数
- They are marked with the `infix` keyword

```
// Define extension to Int
infix fun Int.shl(x: Int): Int {
    ...
}

// call extension function using infix notation

1 shl 2

// is the same as

1.shl(2)
```

参数

函数参数是使用Pascal表达式，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(缺省参数)

函数参数有默认值,当对应的参数是省略。与其他语言相比可以减少数量的过载。

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

默认值定义使用后`** = **`类型的值。

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非常方便。

给出下面的函数：

```
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这个

```
reformat(str)
```

然而，调用非默认时，调用类似于

```
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 = '_'
)
```

如果我们不需要所有的参数

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

Note that the named argument syntax cannot be used when calling Java functions, because Java bytecode does not always preserve names of function parameters.

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`。这个值不需要显式地返回

```
fun printHello(name: String?): Unit {
    if (name != null)
        println("Hello ${name}")
    else
        println("Hi there!")
    // `return Unit` or `return` is optional
}
```

`Unit` 返回类型声明也是可选的。上面的代码等同于

```
fun printHello(name: String?) {
    ...
}
```

单个表达式函数

当一个函数返回单个表达式，花括号可以省略并且主体由 `** ==` 符号之后指定

```
fun double(x: Int): Int = x * 2
```

显式地声明返回类型 [可选](#) 时，这可以由编译器推断

```
fun double(x: Int) = x * 2
```

显式地返回类型

函数模块体必须显式地指定返回类型，除非是用于返回 `Unit`，[在这种情况下](#)，它是可选的。Kotlin不推断返回类型与函数在模块体的功能，因为这些功能可能在模块体有复杂的控制流程，对于读者（有时甚至编译器）来说返回类型将不明显。

数量可变的参数(可变参数)

函数的（通常最后一个）参数可以使用 `'vararg'` 修饰：

```
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  
}
```

允许可变参数传递给函数：

```
val list = asList(1, 2, 3)
```

内部函数 `vararg` 类型 `T` 是可见的 `array T`，即上面的例子中的 `ts` 变量是 `Array<out T>` 类型。

只有一个参数可以标注为 `vararg`。If a `vararg` parameter is not the last one in the list, values for the following parameters can be passed using the named argument syntax, or, if the parameter has a function type, by passing a lambda outside parentheses.

当我们调用 `vararg` 函数，我们可以一个接一个传递参数，例如 `asList(1, 2, 3)` 或者，如果我们已经有了一个数组并希望将其内容传递给函数，我们使用 **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(函数范围)

在Kotlin中函数可以在文件顶级声明，这意味着您不需要像一些语言如Java、C#或Scala那样创建一个类来持有一个函数。此外除了顶级函数功能，Kotlin函数也可以在局部声明，作为成员函数和扩展函数。

局部函数

Kotlin提供局部函数,即一个函数在另一个函数中


```
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的局部变量（即闭包），所以在上面的例子，the *visited*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])
}
```

Member Functions

成员函数

成员函数是一个函数,定义在一个类或对象里

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

成员函数调用点符号

```
Sample().foo() // creates instance of class Sample and calls foo
```

有关类信息和主要成员查看[Classes](#) 和 [Inheritance](#)

重载函数

函数可以有泛型参数，通过在函数前使用尖括号指定。

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

有关重载函数更多信息请查看 [Generics](#)

内联函数

内联函数解释 [here](#)

扩展函数

扩展函数解释 [their own section](#)

高阶函数和Lambdas表达式

高阶函数和Lambdas表达式中有详细解释 [their own section](#)

Tail recursive functions

Kotlin supports a style of functional programming known as [tail recursion](#). This allows some algorithms that would normally be written using loops to instead be written using a recursive function, but without the risk of stack overflow. When a function is marked with the `tailrec` modifier and meets the required form the compiler optimises out the recursion, leaving behind a fast and efficient loop based version instead.

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

This code calculates the fixpoint of cosine, which is a mathematical constant. It simply calls `Math.cos` repeatedly starting at 1.0 until the result doesn't change any more, yielding a result of 0.7390851332151607. The resulting code is equivalent to this more traditional style:

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

To be eligible for the `tailrec` modifier, a function must call itself as the last operation it performs. You cannot use tail recursion when there is more code after the recursive call, and you cannot use it within try/catch/finally blocks. Currently tail recursion is only supported in the JVM backend.

高阶函数和lambda表达式

高阶函数

高阶函数是一种能用函数作为参数或者返回值为函数的一种函数。 `lock()` 是高阶函数中一个比较好的例子，它接收一个lock对象和一个函数，获得锁，运行传入的函数，并释放锁：

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

我们分析一下上面的代码：函数 `body` 拥有[函数类型](#): `() -> T` 所以body应该是一个不带参数并且返回 `T` 类型的值的函数。它在`try`代码块中调用，被 `lock` 保护的，当 `lock()` 函数被调用时返回他的值。

如果我们想调用 `lock()` 函数，我们可以把另一个函数传递给它作为参数(详见 [函数引用](#)):

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

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

另外一种更为便捷的方式是传入一个 [lambda 字面量](#):

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

lambda 表达式 [这里有](#)更详细的描述, 但是为了继续这一段，让我们看到一个简短的概述：

- 一个 lambda 表达式总是被大括号包围着。
- 其参数（如果有的话）被声明在 `->` 之前（参数类型可以省略）
- 函数体在 `->` 后面 (如果存在的话).

在Kotlin中, 如果函数的最后一个参数是一个函数，那么该参数可以在括号外指定：

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

另一个高阶函数的例子是 `map()` ([MapReduce](#)):

```
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
}
```

这个函数可以如下调用:

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

Note that the parentheses in a call can be omitted entirely if the lambda is the only argument to that call.

还有一个有用的公约是: 如果函数数字面量有一个参数, 那它的声明可以省略(连同 `->`), 用 `it` 表示。

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

这些约定可以写成 [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“函数数字面值”, 即 一个未声明的函数, 但却立即写为表达式。思考下面的例子:

```
max(strings, { a, b -> a.length() < b.length() })
```

`max` 函数是一个高阶函数, 也就是说 他的第二个参数是一个函数. 这个参数是一个表达式, 但它本身也是一个函数, 也就是函数数字面值. 写成一个函数的话, 它相当于

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

函数类型

对于一个接收一个函数作为参数的函数, 我们必须为该参数指定一个函数类型。譬如上述 `max` 函数定义如下:

```
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` 是一个 `(T, T) -> Boolean` 类型的函数, 也就是说 `less` 函数接收两个 `T` 类型的参数并返回一个 `Boolean` 值: 如果第一个比第二个小就返回 `True` .

在第四行代码里, `less` 被用作作为一个函数: 它传入两个 `T` 类型的参数.

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

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

Lambda表达式语法

Lambda 表达式的全部语法形式, 也就是函数类型的字面量, 譬如下面的代码:

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

一个函数Lambda表达式总是被大括号包围着, 在括号内有全部语法形式中的参数声明并且有可选的类型注解, 函数体后面有一个 `->` 符号。如果我们把所有的可选注解都留了出来, 那剩下的是什么样子的:

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

这是非常常见的, 一个 lambda 表达式只有一个参数。如果Kotlin能自己计算出自己的数字签名, 我们就可以不去声明这个唯一的参数。并且用 `it` 进行隐式声明。

```
ints.filter { it > 0 } // this literal is of type '(it: Int) -> Boolean'
```

请注意, 如果函数取另一个函数作为最后一个参数, 该 lambda 表达式参数可以放在 括号外的参数列表。语法细则详见 [callSuffix](#).

匿名函数

上述 lambda 表达式的语法还少了一个东西: 能够指定函数的返回 类型。在大多数情况下, 这是不必要的。因为返回类型可以被自动推断出来. 然而, 如果你需要明确的指定。你需要一个替代语法: *匿名函数*.

```
fun(x: Int, y: Int): Int = x + y
```

匿名函数看起来很像一个普通函数声明, 只是名字被省略了。内容 也是一个表达式 (如上面的代码) 或者代码块:

```
fun(x: Int, y: Int): Int {
    return x + y
}
```

指定的参数和返回类型与指定一个普通函数方式相同，只是如果参数 类型能通过上下文推断出来，那么该参数类型是可以省略的：

```
ints.filter(fun(item) = item > 0)
```

匿名函数的返回类型推断法只适用于常规函数：具有表达式体并且必须明确指定函数体有代码（或者假定 `Unit`）块的返回类型能够被自动推断出来。

请注意，匿名函数参数始终在圆括号内传递。允许在 函数括号外使用的速记语法只针对于 lambda 函数。

另一个 lambda 表达式和匿名函数区别是 [non-local returns](#) 的行为。一个不带标签的 `return{ .keyword }` 语句 总是在用 `fun` 关键词声明的函数中返回。这意味着 lambda 表达式中的 `return{ .keyword }` 将在函数闭包中返回。然而匿名函数 `return*{ .keyword }` 就是在匿名函数自身中返回。

闭包

一个 lambda 表达式或者匿名函数（以及一个 [本地函数](#) 本地函数和一个 [对象表达式](#)）可以访问他的_闭包_，即声明在外范围内的变量。与java不同，在闭包中捕获的变量可以被修改：

```
var sum = 0
ints.filter { it > 0 }.forEach {
    sum += it
}
print(sum)
```

带接收者得函数数字面值

kotlin提供了使用一个特定的 *receiver*对象 来调用一个函数的能力. 在函数体内部，你可以调用 接受者对象 的方法而不需要任何额外的限定符。这和 扩展函数 有点类似，它允你在函数体内访问接收器对象的成员。

他们的一个最重要的例子是[Type-safe Groovy-style builders](#)的使用。

这样的函数数字面量的类型是一个带receiver的函数类型

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

如果函数是一个在receiver对象上的方法，那么这个函数可以被调用

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

匿名函数的语法允许你直接指定函数的receiver的类型 如果你需要用一個receiver声明一个函数类型的变量，并且在后面用到它，那么这个语法就很有用

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

当receiver类型能够被从上下文推断的时候，Lambda表达式能够被用于带receiver的函数数字面量

```
class HTML {  
    fun body() { ... }  
}  
  
fun html(init: HTML.() -> Unit): HTML {  
    val html = HTML() // create the receiver object  
    html.init()        // pass the receiver object to the lambda  
    return html  
}  
  
html {                // lambda with receiver begins here  
    body()             // calling a method on the receiver object  
}
```

内联函数 inline

使用[高阶函数](#)会带来一些运行时间效率的损失：每一个函数都是一个对象，并且都会捕获一个闭包。即那些在函数体内会被访问的变量。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。我们通过下面的示例函数来分析上面这些内容。如，`lock()` 函数可以被很容易地在调用点被内联。考虑下面的例子：

```
lock(1) { foo() }
```

编译器没有为参数创建一个函数对象和产生一个调用点。取而代之，编译器产生了下面的代码：

```
1.lock()
try {
    foo()
}
finally {
    1.unlock()
}
```

这个不是我们在一开始时候想要的么？

为了让编译器做这个，我们需要在 `lock()` 函数前面加上 `inline` 修饰符：

```
inline fun lock<T>(lock: Lock, body: () -> T): T {
    // ...
}
```

`inline` 修饰符会影响函数体本身以及传递过来的lambdas: 所有的这些会被内联到 调用点。

内联本身有时会引起生成的代码数量增加，但是如果我们使用得当(不要内联大的函数)。它将在 性能上有所提升，尤其是在超多态(megamorphic)调用点的循环中。

禁止内联 (noinline)

为了预防 有时候你只希望被（作为参数）传递到一个内联函数的lambdas 只有一些被内联，你可以用 `noinline` 修饰符标记你的参数：

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

可以内联的lambdas只能在内联函数内部被调用或者被当作一个可内联的参数传递。但是通过 `noinline` 我们可以把它变化成任何的方式：储存在指定地点，传递它等等。

需要注意的是，如果一个内联函数没有可以内联的函数参数并且没有 [泛型变量](#)，编译器会产生一个警告。因为内联这样的函数 很可能是无意义的(你可以supress这个警告如果你确定内联是必须的)。

非本地 (Non-local) 的返回

在Kotlin中，我们只能使用一个普通的，无限制的 `return` 来退出一个有名的函数或者匿名函数。这个意味着为了退出一个lambda,我们不得不用一个 [标签](#),同时 在一个 lambda中 `return` 单独出现时不允许的，因为一个lambda不可能让外围的函数返回：

```
fun foo() {
    ordinaryFunction {
        return // ERROR: can not make `foo` return here
    }
}
```

但是如果lambda所传递到的函数是内联的，那么return也会被内联。于是下面就是允许的：

```
fun foo() {
    inlineFunction {
        return // OK: the lambda is inlined
    }
}
```

这样的返回(在lambda中，但是退出封闭的函数)被称作 *non-local* 返回。我们把 这种排序构造用在循环中。通常这些内联函数是封闭的：

```
fun hasZeros(ints: List<Int>): Boolean {
    ints.forEach {
        if (it == 0) return true // returns from hasZeros
    }
    return false
}
```

注意有些内联函数可能会不直接地在函数体中调用那些被作为参数传递过来的lambdas， 而从其他执行上下文中调用，比如一个本地对象或者一个嵌套函数。在这种情况下， *non-local*的控制流程 也会被lambdas禁止。为了标识这种情况，lambda参数需要 以 `crossinline` 修饰：

```
inline fun f(crossinline body: () -> Unit) {
    val f = object: Runnable {
        override fun run() = body()
    }
    // ...
}
```

`break` 和 `continue` 在内联的lambdas还不能使用，但是我们正在计划支持他们。

泛型变量

有时候我们需要访问一个作为参数传递过来的一个类型：

```
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，但是函数调用点十分不优雅：

```
myTree.findParentOfType(MyTreeNodeType::class.java)
```

我们真正想要的只是简单的传递一个类型给函数，例如这样调用：

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

为了实现这个，内联函数支持 *泛型变量*，于是我们就可以这样写：

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

我们用泛型修饰符 *reified* 来修饰这个类型参数，现在它就像一个普通的类一样，能在函数内部被正常访问了。因为这个函数是内联的，不需要反射，一般的运算符 *!is* 和 *as* 就可以工作了。此外，我们可以像上面说的调用它：

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

虽然反射在很多情况下不一定需要，我们还是可以在泛型参数里使用它：

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

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

普通的函数 (没有被标记为内联) 不能够有泛型参数。一个没有一个运行时表示的类型 (例如一个非泛型参数或者一个虚构类型如 *Nothing*) 不能被用做泛型参数的变量。

更详细解释参考 [spec document](#).

其他

解构声明

有时把一个对象_解构_成很多变量很比较方便，比如：

```
val (name, age) = person
```

这种语法叫做_解构声明_。一个解构声明同时创造多个变量。我们申明了两个新变量：`name` 和 `age` ,并且可以独立使用他们。

```
println(name)
println(age)
```

一个解构声明会被向下编译成下面的代码：

```
val name = person.component1()
val age = person.component2()
```

`component1()` 和 `component2()` 函数是 *principle of conventions* widely 在Kotlin 中的另一个例子。(参考运算符如 `+` , `*` , `for-loops` 等) 任何可以被放在解构声明右边的和组件函数的需求数字都可以调用它。当然，这里可以有更多的如 `component3()` 和 `component4()` .

解构声明对 `for-loops`有效：

Destructuring declarations also work in `for-loops`: when you say

```
for ((a, b) in collection) { ... }
```

变量 `a` 和 `b` 从调用从 `component1()` 和 `component2()` 返回的集合`collection`中的对象。

例：从函数中返回两个变量

让我们从一个函数中返回两个变量。例如，一个结果对象和一些排序的状态。在Kotlin中一个简单的实现方式是申明一个[data class](#)并且返回他的实例：

```
data class Result(val result: Int, val status: Status)
fun function(...): Result {
    // computations

    return Result(result, status)
}

// Now, to use this function:
val (result, status) = function(...)
```

既然数据类自动申明 `componentN()` 函数，解构声明在这里也有效。

NOTE: 我们也可用标准类 `Pair` 并且让 `function()` 返回 `Pair<Int, Status>`，但是如果让数据合理命名通常还是更好。

例: 解构和映射。

可能最好的遍历一个映射的方式就是这样：

```
for ((key, value) in map) {
    // do something with the key and the value
}
```

为了实现这个，我们需要

- 通过提供一个 `iterator()` 迭代函数来表示一系列有序值来表示映射。
- 把每个元素标识为一对函数 `component1()` 和 `component2()`。

当然，标准库中提供了这一扩展：

```
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()
```

于是你可以自由的使用解构声明 `for-loops` 来操作映射(也可以用在数据类实例的集合等)。

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)           // 打印 "[1, 2, 3]"
numbers.add(4)
println(readOnlyView)      // 打印 "[1, 2, 3, 4]"
readOnlyView.clear()       // -> 不能编译

val strings = hashSetOf("a", "b", "c", "c")
assert(strings.size == 3)
```

Kotlin 没有专门的语法结构创建 list 或 set。要用标准库的方法，如 `listOf()`、`mutableListOf()`、`setOf()`、`mutableSetOf()`。创建 map 在非性能关键代码中可以用一个简单的[惯用法](#)：`mapOf(a to b, c to d)`

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

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

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

注意这些类型是[协变的](#)。这意味着，你可以把一个 `List<Rectangle>` 赋值给 `List<Shape>` 假定 `Rectangle` 继承自 `Shape`。对于可变集合类型这是不允许的，因为这将导致运行时故障。

有时你想给调用者返回一个集合在某个特定时间的一个快照，一个保证不会变的：

```
class Controller {
    private val _items = mutableListOf<String>()
    val items: List<String> get() = _items.toList()
}
```

这个 `toList` 扩展方法只是复制列表项，因此返回的 list 保证永远不会改变。

List 和 set 有很多有用的扩展方法值得熟悉：

```
val items = listOf(1, 2, 3, 4)
items.first == 1
items.last == 4
items.filter { it % 2 == 0 } // 返回 [2, 4]
rwList.requireNotNulls()
if (rwList.none { it > 6 }) println("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(map["foo"])
val snapshot: Map<String, Int> = HashMap(readWriteMap)
```

范围

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

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

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

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

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

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

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

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

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

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

它是如何工作的

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

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

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

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

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

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

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

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

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

~~###常见的接口定义~~

有两种基本接口:Range和Progression。

Range 接口定义了一个范围或一个数学意义上的区间。它有两个端点,start 和end,并且contains()函数检查是否包含一个给定的数字范围 (也可以作为in/!in操作符)。“开始”和“结束”是包含在范围内。如果start=end,范围包含一个确定的元素。如果 start > end,范围是空的。

```
interface Range<T : Comparable<T>> {
    val start: T
    val end: T
    fun contains(element: T): Boolean
}
```

Progression定义了一种等差算法。它有 start(进程中的第一个元素), end(被包含的最后一个元素)和increment (每个进程元素和以前的区别,非零)。但它的主要特征是,可以遍历过程,所以这是Iterable的子类。end最后一个元素不是必须的,如 start < end && increment < 0 or start > end && increment > 0.

```
interface Progression<N : Number> : Iterable<N> {
    val start: N
    val end: N
    val increment: Number // not N, because for Char we'll want it to be negative
    // fun iterator(): Iterator<N> is defined in Iterable interface
}
```

迭代'Progression'相当于一个索引for{:.Java关键字}循环:

```
// if increment > 0
for (int i = start; i <= end; i += increment) {
    // ...
}

// if increment < 0
for (int i = start; i >= end; i += increment) {
    // ...
}
```

类的实现

为了避免不必要的重复,让我们只考虑一个数字类型,Int。对于其他类型的数量实现是一样的。注意,可以使用这些类的构造函数创建实例,而更方便使用的rangeTo()(这个名字,或作为..操作符), downTo(), reversed()和step()等实用的函数,以后介绍。

IntProgression 类很简单快捷:

```
class IntProgression(override val start: Int, override val end: Int, override val
increment: Int): Progression<Int> {
    override fun iterator(): Iterator<Int> = IntProgressionIteratorImpl(start, end,
increment) // implementation of iterator is obvious
}
```

IntRange 有点复杂:它的实现类是 Progression<Int>和Range<Int>,因为它是自然的遍历的(默认增量为1整数和浮点类型):

```
class IntRange(override val start: Int, override val end: Int): Range<Int>,
Progression<Int> {
    override val increment: Int
        get() = 1
    override fun contains(element: Int): Boolean = start <= element && element <= end
    override fun iterator(): Iterator<Int> = IntProgressionIteratorImpl(start, end,
increment)
}
```

ComparableRange 也很简单(请记住,比较转换是 compareTo()):

```
class ComparableRange<T : Comparable<T>>(override val start: T, override val end: T):
Range<T> {
    override fun contains(element: T): Boolean = start <= element && element <= end
}
```

一些实用函数

rangeTo()

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

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

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

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

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

downTo()

downTo() 的扩展函数可以为任何数字整型对定义,这里有两个例子:

```

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

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

```

reversed()

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

```

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

```

step()

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

```

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

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

```

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

```

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

```

类型的检查与转换

is 和 !is运算符

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

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

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

智能转换

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

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

如果错误的检查导致返回，编译器会清楚地转换为一个正确的：

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

或者在右边是 `&&` 和 `||`：

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

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

这些智能转换在 [when-expressions](#) 和 [while-loops](#) 也一样：

```

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

```

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

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

“不安全”的转换运算符

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

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

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

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

“安全的”（可为空的）转换运算符

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

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

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

This表达式

为了记录下当前的接受者我们使用`this`表达式:

- 在一个[类](#)成员中, `this`指的是当前类对象。
- 在一个[扩展函数](#)或者[带有接收者字面函数](#), `this`表示左边的接收者。

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

`this`使用范围

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

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

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

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

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

等式

Kotlin中有两种类型的等于:

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

引用相等

引用相等使用 `===` 操作符判断(它的否定是 `!==`). `a === b` 只有当 `a` 和 `b` 指向同一个对象才返回`true`。> ~~另外, 你可以使用内联函数 `identityEquals()` 判断引用相等: > ~~~> kotlin > `a.identityEquals(b)` > // or > `a.identityEquals b` // infix call > > 当且仅当 `a` 和 `b` 指向同一个对象返回`true`。

结构相等

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

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

就是说如果 `a` 不是 `null` 则调用 `equals(Any?)` 函数, 否则 (即 `a` 是 `null`) 检查 `b` 的是不是 `null` 引用。

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

运算符重载

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

转换

这里我们描述了一些常用运算符的重载。

一元运算符

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


这张表解释了当编译器运行时，比如，表达式 `+a`，是这样运行的：

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

注意 这些操作符和其它的一样, 都被优化为[基本类型](#)并且不会产生多余的开销。

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

这些操作符允许修改接收者和返回类型。

 **`inc()/dec()` 不应该改变接收对象.**
“修改接受者”你应该修改接收者变量而非对象。

编译器是这样解决有[后缀](#)的操作符的比如 `a++`：

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

计算表达式的步骤是：

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

a- 的运算步骤也是一样的。

对于前缀运算符 ++a 和 --a 的解决方式也是一样的, 步骤是:

- 把 a.inc() 作用于 a ,
- 返回新值 a 作为表达式结果。

二元操作符

表达式	翻译为
a + b	a.plus(b)
a - b	a.minus(b)
a * b	a.times(b)
a / b	a.div(b)
a % b	a.mod(b)
a..b	a.rangeTo(b)

编译器只是解决了该表中翻译为列的表达式。

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

in 和 !in 的产生步骤是一样的, 但参数顺序是相反的。

符号	翻译为
a[i]	a.get(i)
a[i, j]	a.get(i, j)
a[i_1, ..., i_n]	a.get(i_1, ..., i_n)
a[i] = b	a.set(i, b)
a[i, j] = b	a.set(i, j, b)
a[i_1, ..., i_n] = b	a.set(i_1, ..., i_n, b)

方括号被转换为 get set 函数。

符号	翻译为
a()	a.invoke()
a(i)	a.invoke(i)
a(i, j)	a.invoke(i, j)
a(i_1, ..., i_n)	a.invoke(i_1, ..., i_n)

括号被转换为带有正确参数的 invoke 函数。

表达式	翻译为
-----	-----

表达式	翻译为
<code>a += b</code>	<code>a.plusAssign(b)</code>
<code>a -= b</code>	<code>a.minusAssign(b)</code>
<code>a *= b</code>	<code>a.timesAssign(b)</code>
<code>a /= b</code>	<code>a.divAssign(b)</code>
<code>a %= b</code>	<code>a.modAssign(b)</code>

在分配 `a += b` 时编译器是下面这样实现的:

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

注意: assignments 在 Kotlin 中不是表达式。

表达式	翻译为
<code>a == b</code>	<code>a?.equals(b) ?: b === null</code>
<code>a != b</code>	<code>!(a?.equals(b) ?: b === null)</code>

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

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

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

所有的比较都转换为 `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。

Null 安全性

可空 (Nullable) 和不可空 (Non-Null) 类型

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

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

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

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

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

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

若是想要允许 `null`，我们可以声明一个变量为可空字符串，写作 `String?`：

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

现在，如果你调用/访问一个 `a` 方法/属性（译者注：`a` 是一个不可空类型）的一个方法，它保证不会造成 `NPE`，这样你就可以放心地使用：

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

但是如果你想访问 `b` 的相同属性，这将是不安全的，同时编译器会报错：

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

可是我仍然需要访问这些属性，对吧？这里有一些方式可以这么做：

使用条件语句检测是否为 `null`

首先，你可以明确地检查 `b` 是否为 `null`，并分别处理两种选择：

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

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

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

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

安全的调用

你的第二个选择是安全的操作符，写作 `?.`：

```
b?.length
```

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

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

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

如果任意一个属性（环节）为空，这个链式调用就会返回 `null`。

Elvis 操作符

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

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

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

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

如果 `?:` 的左边表达式是非空的，elvis 操作符就会返回左边的结果，否则返回右边的内容。

请注意，仅在左侧为空的时候，右侧表达式才会进行计算。

注意，因为 `throw` 和 `return` 在 Kotlin 中都是一种表达式，它们也可以用在 Elvis 操作符的右边。非常方便，例如，检查函数参数：

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

!! 操作符

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

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

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

安全转型

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

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

异常

异常类

Kotlin中所有异常类都是 `Exception` 类的子累。每一个异常都含有一条信息、栈回溯信息和一个可选选项。可以使用 `throw-expression` 来抛出一个异常。

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

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

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

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

Try是一个表达式

`try` 是一个表达式，比如，它可以有一个返回值。

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

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

可检查异常

Kotlin中没有可检查异常。这是有很多原因的，但是我们会提供一个简单的示例。以下示例是JDK中 `StringBuilder` 类实现中的

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

这一行代码告诉我们，每次当我们调用`append`向 `StringBuilder` 追加字符串时，都需要捕获 `IOExceptions` 异常。因为它可能会操作IO(`Writer` 也实现了 `Appendable` 接口)...

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

这样做是没有好处的，参阅 [Effective Java](#), Item 65: *不要忽略异常*。

Bruce Eckel在[Does Java need Checked Exceptions?](#) 中指出：

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

相关引用：

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

Java交互

请在 [Java Interoperability section](#) 异常章节中参阅Java交互相关信息。

注解

注解的声明

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

```
annotation class Fancy
```

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

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

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

用途

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

~~在很多情况下，@ 这个标志不是强制性使用的。它只是在当注解表达式或者本地声明时需要：~~

```
fancy class Foo {  
    fancy fun baz(fancy foo: Int): Int {  
        @fancy fun bar() { ... }  
        return (@fancy 1)  
    }  
}
```

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

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

你也可以注解属性访问器：

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

构造方法

注解可以有参数的构造方法。

```
annotation class Special(val why: String)  
  
@Special("example") class Foo {}
```

Allowed parameter types are:

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

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

```
public annotation class ReplaceWith(val expression: String)  
  
public annotation class Deprecated(  
    val message: String,  
    val replaceWith: ReplaceWith = ReplaceWith(""))  
  
@Deprecated("This function is deprecated, use === instead", ReplaceWith("this ===  
other"))
```

Lambdas

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

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

Annotation Use-site Targets

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

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

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

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

package org.jetbrains.demo
```

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

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

The full list of supported use-site targets is:

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

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

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

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

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

Java注解

Java注解是百分百适用于Kotlin:

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

class Tests {
    @Test fun simple() {
        assertEquals(42, getTheAnswer())
    }
}
```

~~Java注解也可像用import修饰符重新命名: ~~

```
import org.junit.Test as test

class Tests {
    test fun simple() {
        ...
    }
}
```

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

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

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

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

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

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

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

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

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

如果你需要像注解参数一样指定一个类，使用一个Kotlin的类吧([KClass](#))。Kotlin编译器会自动把它转换成Java类，使得Java代码能正常看到注解和参数。

```
import kotlin.reflect.KClass

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

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

注解实例的值被视为Kotlin的属性。

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

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

反射

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

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

类引用

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

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

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

注意Kotlin类引用不完全与Java类引用一致.查看[Java interop section](#) 详细信息。

函数引用

我们有一个像下面这样的函数声明:

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

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

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

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

注意现在 :: 不能被使用来重载函数. 将来, 我们计划 提供一个语法明确参数类型这样就可以使用明确的重载函数了。

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

例子: 函数组合

考量以下方法:

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

它返回一个由俩个传递进去的函数的组合。现在你可以把它用在可调用的引用上了:

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

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

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

属性引用

我们同样可以用 `::` 操作符来访问Kotlin中的顶级类的属性：

```
var x = 1

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

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

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

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

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

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

```
class A(val p: Int)

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

对于扩展属性:

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

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

与Java反射调用

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

```
import kotlin.reflect.jvm.*

class A(val p: Int)

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

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

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

构造函数引用

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

```
class Foo

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

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

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

Type-Safe Builders

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

很多情况下，Kotlin允许[检查类型](#)的构建器，这样比Groovy本身提供的构建器更有吸引力。

其他情况下，Kotlin也支持[动态类型](#)的构建器。

一个类型安全的构建器的示例

考虑下面的代码~~。这段代码是从[这里](#)摘出来并稍作修改的~~：

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

fun result(args: Array<String>) =
    html {
        head {
            title {"XML encoding with Kotlin"}
        }
        body {
            h1 {"XML encoding with Kotlin"}
            p {"this format can be used as an alternative markup to XML"}

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

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

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

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

构建器的实现原理

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

现在我们先回忆一下我们在构建器代码中这么声明：

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

`html` 实际上是一个函数，其参数是一个[lambda 表达式](#) 这个函数定义如下：

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

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

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

(`head` 和 `body` 都是 `HTML` 类的成员函数)

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

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

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

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


```

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

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

```

实际上这两个函数做的是完全相同的事情，所以我们可以定义一个泛型函数 `initTag`：

```

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

```

现在我们的函数变成了这样：

```

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

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

```

我们可以使用它们来构建 `<head>` 和 `<body>` 标签。

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

```

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

```

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

```

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

```

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

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

包 `com.example.html` 的完整定义

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

```
package com.example.html

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

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

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

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

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

    private fun renderAttributes(): String? {
        val builder = StringBuilder()
        for (a in attributes.keys) {
            builder.append(" $a=\"${attributes[a]}\"")
        }
        return builder.toString()
    }

    override fun toString(): String {
        val builder = StringBuilder()
        render(builder, "")
        return builder.toString()
    }
}
```

```

abstract class TagWithText(name: String) : Tag(name) {
    operator fun String.unaryPlus() {
        children.add(TextElement(this))
    }
}

class HTML() : TagWithText("html") {
    fun head(init: Head.() -> Unit) = initTag(Head(), init)

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

class Head() : TagWithText("head") {
    fun title(init: Title.() -> Unit) = initTag(Title(), init)
}

class Title() : TagWithText("title")

abstract class BodyTag(name: String) : TagWithText(name) {
    fun b(init: B.() -> Unit) = initTag(B(), init)
    fun p(init: P.() -> Unit) = initTag(P(), init)
    fun h1(init: H1.() -> Unit) = initTag(H1(), init)
    fun a(href: String, init: A.() -> Unit) {
        val a = initTag(A(), init)
        a.href = href
    }
}

class Body() : BodyTag("body")
class B() : BodyTag("b")
class P() : BodyTag("p")
class H1() : BodyTag("h1")

class A() : BodyTag("a") {
    public var href: String
        get() = attributes["href"]!!
        set(value) {
            attributes["href"] = value
        }
}

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

```

~~### 附录.让Java类更好~~

上面的代码中有一段很好的：

```
class A() : BodyTag("a") {
    var href: String
    get() = attributes["href"]!!
    set(value) { attributes["href"] = value }
}
```

我们访问映射(Map) attributes的方式，是把它当作“关联数组”(associate array)来访问的：用[]操作符。依照编译器的[惯例](#)它被翻译成get(K)和set(K, V)，正好。但是我们说过，attributes是一个JavaMap，也就是说，它没有set(K, V)函数。(译注：Java的映射中的函数是put(K, V))。在Kotlin中，这个问题很容易解决：

```
fun <K, V> Map<K, V>.set(key: K, value: V) = this.put(key, value)
```

所以我们只要给Map类添加一个[扩展函数](#)set(K, V)，并委托Map类原有的put(K, V)函数，就可以让Java类使用Kotlin的操作符号了。

动态类型

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

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

`dynamic` 类型关闭了Kotlin类型检查:

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

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

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

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

动态调用返回 `dynamic` 作为结果, 因此我们可以轻松实现链式调用:

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

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

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

更多细节, [查看](#).

参考

Grammar

We are working on revamping the Grammar definitions and give it some style! Until then, please check the [Grammar from the old site](#)

互操作

在Kotlin中调用Java代码

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

基本所有的 Java 代码都可以运行

```
import java.util.*

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

Getters and Setters

Methods that follow the Java conventions for getters and setters (no-argument methods with names starting with `get` and single-argument methods with names starting with `set`) are represented as properties in Kotlin. For example:

```
import java.util.Calendar

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

Note that, if the Java class only has a setter, it will not be visible as a property in Kotlin, because Kotlin does not support set-only properties at this time.

返回void的方法

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

将Java代码中与Kotlin关键字冲突的标识符进行转义

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

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

Null安全性和平台类型

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

考虑如下例子:

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

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

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

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

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

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

平台类型的概念

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

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

Nullability annotations

Java types which have nullability annotations are represented not as platform types, but as actual nullable or non-null Kotlin types. Currently, the compiler supports the [JetBrains flavor of the nullability annotations](#) (`@Nullable` and `@NotNull` from the `org.jetbrains.annotations` package).

映射类型

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

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

一些非原生类型也会作映射：

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

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

Java类型	Kotlin只读类型	Kotlin可变类型	加载的平台类型
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>!

Java类型	Kotlin只读类型	Kotlin可变类型	加载的平台类型
Iterator<T>	Iterable<T>	Iterable<T>	Iterable<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数组的映射在这里提到过 [below](#)：

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

Kotlin中的Java泛型

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

- Java的通配符转换成类型投射
 - `Foo<? extends Bar>` 转换成 `Foo<out Bar!>!`
 - `Foo<? super Bar>` 转换成 `Foo<in Bar!>!`
- Java的原始类型转换成星号投射
 - `List` 转换成 `List<*>!`，也就是 `List<out Any?>!`

和Java一样，Kotlin在运行时不保留泛型，即对象不知道传递到它们构造器中的那些参数的实际类型。~~Kotlin的范型就像Java一样不会在运行时保留信息，也就是对象不会携带传递到它们构造函数中的类型参数的信息。~~~也就是说，运行时无法区分 `ArrayList<Integer>()` 和 `ArrayList<Character>()`。~~也就 是，`ArrayList<Integer>()` 和 `ArrayList<Character>()` 是区分不出来的。这意味着，不可能用 `is`-来检测泛型。~~这就导致，无法使用`is`-检测范型。~~ Kotlin只允许用`is`-来检测星号投射的泛型类型：~~Kotlin只允许用`is`-检测星号投射的范型类型。~~

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

```
~~~ kotlin if (a is List<Int>) // 错误：无法检测是否是一个Int的List // but if (a is
List<*>) // 可以：不确保List里的内容
```

Java数组

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

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

假设有一个Java方法，它接收一个表示索引的int数组作参数

```
public class JavaArrayExample {

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

在Kotlin里你可以这样传递一个原生数组:

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

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

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

即便是用索引遍历数组。

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

最后, `in`-检测也没有额外负担。

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

Java Varargs

Java类也会这样声明方法, 表示参数是可变参数。

```
public class JavaArrayExample {

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

这种情况, 你需要用展开操作符 `*` 来传递 `IntArray` :

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

目前无法传递 `null` 给一个变参的方法。

Operators

Since Java has no way of marking methods for which it makes sense to use the operator syntax, Kotlin allows using any Java methods with the right name and signature as operator overloads and other conventions (`invoke()` etc.) Calling Java methods using the infix call syntax is not allowed.

受检异常

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

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

对象方法

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

wait()/notify()

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

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

getClass()

获取一个对象的类型信息，我们可以用`javaClass`这个扩展属性。

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

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

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

clone()

要重写 `clone()`，扩展 `kotlin.Cloneable`：

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

不要忘了 [Effective Java](#), 第11条: *谨慎的重写克隆*。

finalize()

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

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

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

java类的继承

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

访问静态成员

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

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

Java 反射

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

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

SAM(单抽象方法) 转换

就像 Java 8 那样, Kotlin 支持 SAM 转换, 这意味着 Kotlin 函数字面量可以被自动的转换成 只有一个非默认方法的 Java 接口的实现, 只要这个方法的参数类型 能够跟这个 Kotlin 函数的参数类型匹配的上。

你可以这样创建SAM接口的实例:

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

...在方法调用里:

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

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

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

注意SAM的转换只对接口有效，对抽象类无效，即使它们就只有一个 抽象方法。

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

Using JNI with Kotlin

To declare a function that is implemented in native (C or C++) code, you need to mark it with the `external` modifier:

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

The rest of the procedure works in exactly the same way as in Java.

Java调用Kotlin代码

Java可以轻松调用Kotlin代码。

属性

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

包级别的函数

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

```
// example.kt
package demo

class Foo

fun bar() {
}
```

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

可以使用 @JvmName 注解自定义生成的Java 类的类名：

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

package demo

class Foo

fun bar() {
}
```

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

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


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

package demo

fun foo() {
}
```

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

package demo

fun bar() {
}
```

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

实例字段

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

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

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

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

静态字段

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

通常，这些字段都是private的，但是他们可以通过以下方式暴露出来。

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

用 `@JvmField` 注解该属性可以生成一个与该属性相同可见性的静态字段。

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

```
// Java
Key.COMPARATOR.compare(key1, key2);
// public static final field in Key class
```

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

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

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

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

```
// file example.kt

object Obj {
    const val CONST = 1
}

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

const val MAX = 239
```

In Java:

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

静态方法

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

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

现在，`foo()` 在java里就是静态的了，而 `bar()` 不是：

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

同样的，命名对象：

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

Java 里：

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

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

用@JvmName解决签名冲突

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

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

这两个函数不能同时定义，因为它们的 JVM 签名是一样

的： `filterValid(Ljava/util/List;)Ljava/util/List;`。如果我们真的想让他们在 Kotlin 里用同一个名字，我们需要用 `@JvmName` 去注释它们中的一个（或两个），指定的另外一个名字当参数：

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

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

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

同样的技巧也适用于属性 `x` 和函数 `getX()` 共存：

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

fun getX() = 10
```

生成重载

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

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

对于每一个有默认值的参数，都会生成一个额外的重载，这个重载会把这个参数和它右边的所有参数都移除掉。在上面这个例子里，生成下面的方法：

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

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

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

受检异常

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

```
// example.kt
package demo

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

然后我们想要在java里调用它，捕捉这个异常：

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

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

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

Null安全性

当从 Java 中调用 Kotlin 函数时，没人阻止我们传递 `null` 给一个非空参数。这就是为什么 Kotlin 给所有期望非空参数的公开函数生成运行时检测。这样我们就能在 Java 代码里立即得到 `NullPointerException`。

可变泛型

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

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

interface Base
class Derived : Base

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

一种看似理所当然地将函数转换成 Java 代码的方式可能会是这样：

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

问题是，在Kotlin中我们可以这样写 `unboxBase(boxDerived("s"))`，但是这样的写法在Java中是无法通过的，因为在Java中Box的泛型参数T是不可变的，`Box<Derived>`实际上并不是`Box<Base>`的子类。如果在Java中要编译通过我们需要像下面这样定义`unboxBase`：[1]:# (The problem is that in Kotlin we can say, but in Java that would be impossible, because in Java the class `Box` is *invariant* in its parameter `T`, and thus `Box<Derived>` is not a subtype of `Box<Base>`. To make it work in Java we'd have to define `unboxBase` as follows:)

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

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

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

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

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

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

如果我们不想要默认生成的通配符，需要自己指定可以使用`@JvmWildcard`注解：

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

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

```
fun unboxBase(box: Box<@JvmSuppressWildcards Base>): Base = box.value
// 将被转换成
// Base unboxBase(Box<Base> box) { ... }
```

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

Nothing 类型的转换

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

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

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的参数。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

`@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。

使用 Maven

插件与版本

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

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

Milestone	Version
1.0.1 hotfix update 2	1.0.1-2
1.0.1 hotfix update	1.0.1-1
1.0.1	1.0.1
1.0 GA	1.0.0
Release Candidate	1.0.0-rc-1036
Beta 4	1.0.0-beta-4589
Beta 3	1.0.0-beta-3595
Beta 2	1.0.0-beta-2423
Beta	1.0.0-beta-1103
Beta Candidate	1.0.0-beta-1038
M14	0.14.449
M13	0.13.1514
M12.1	0.12.613
M12	0.12.200
M11.1	0.11.91.1
M11	0.11.91
M10.1	0.10.195
M10	0.10.4
M9	0.9.66
M8	0.8.11
M7	0.7.270
M6.2	0.6.1673
M6.1	0.6.602
M6	0.6.69
M5.3	0.5.998

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：

```

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

```

Maven 中需要引用 Kotlin 插件用于编码源码：

```

<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>

```

同时编译 Kotlin 与 Java 源码

编译混合代码时 Kotlin 编译器应先于 Java 的编译器被调用。在 Maven 中这表示 kotlin-maven-plugin 先于 maven-compiler-plugin 运行。

It could be done by moving Kotlin compilation to previous phase, process-sources（如果有更好的解决方案欢迎提出）：

```
<plugin>
  <artifactId>kotlin-maven-plugin</artifactId>
  <groupId>org.jetbrains.kotlin</groupId>
  <version>${kotlin.version}</version>

  <executions>
    <execution>
      <id>compile</id>
      <phase>process-sources</phase>
      <goals> <goal>compile</goal> </goals>
    </execution>

    <execution>
      <id>test-compile</id>
      <phase>process-test-sources</phase>
      <goals> <goal>test-compile</goal> </goals>
    </execution>
  </executions>
</plugin>
```

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	
includeRuntime	If output is a .jar file, whether Kotlin runtime library is included in the jar	No	true
moduleName	Name of the module being compiled	No	The name of the target (if specified) or the project

`kotlin2js` Attributes

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

main	Should compiler generated code call the main function	No
Name	Description	Required

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

KotlinConfigure Kotlin in Project action.

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"
    }
}
```

The correspondence between Kotlin releases and versions is displayed below:

里程碑	版本
1.0.1 hotfix update 2	1.0.1-2
1.0.1 hotfix update	1.0.1-1
1.0.1	1.0.1
1.0 GA	1.0.0
Release Candidate	1.0.0-rc-1036
Beta 4	1.0.0-beta-4589
Beta 3	1.0.0-beta-3595
Beta 2	1.0.0-beta-2423
Beta	1.0.0-beta-1103
Beta Candidate	1.0.0-beta-1038
M14	0.14.449
M13	0.13.1514
M12.1	0.12.613
M12	0.12.200
M11.1	0.11.91.1
M11	0.11.91
M10.1	0.10.195
M10	0.10.4
M9	0.9.66

里程碑	版本
M7	0.7.270
M6.2	0.6.1673
M6.1	0.6.602
M6	0.6.69
M5.3	0.5.998

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.

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"

```

OSGi

OSGi 支持查看 [Kotlin OSGi page](#).

例子

[Kotlin Repository](#) 包含的例子:

- [Kotlin](#)
- [Mixed Java and Kotlin](#)
- [Android](#)
- [JavaScript](#)

Kotlin and OSGi

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

Maven

To include the Kotlin OSGi bundle to a Maven project:

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

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

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

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

Gradle

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

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

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

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

FAQ

Why not just add required manifest options to all Kotlin libraries

Even though it is the most preferred way to provide OSGi support, unfortunately it couldn't be done for now due to so called [“package split” issue](#) that couldn't be easily eliminated and such a big change is not planned for now.

There is `Require-Bundle` feature but it is not the best option too and not recommended to use. So it was decided to make a separate artifact for OSGi.

常见问题

FAQ

常见问题

Kotlin是什么？

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

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

为什么要出一门新语言？

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

这个项目背后的主要设计目标是：

- 创建一个兼容 Java 的语言，
- 它的编译速度至少和 Java 一样快，
- 使它比 Java 更安全，即静态检测空指针解引用等常见陷阱，
- 通过支持变量类型推断、高阶函数（闭包）、扩展函数、mixin 以及一等公民的委托等等使它比 Java 更简洁；
- 并且，在保持有用级表现力（见上文）的前提下，使它比最成熟的竞品——Scala 更简单。

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。

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 互通性](#)。

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。安装说明请参阅这个[教程](#)。

有独立的编译器吗？

有。你可以从 [Github 上的发布页](#) 下载独立的编译器和其他构建工具。

Kotlin是函数式语言吗？

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

Kotlin支持泛型吗？

Kotlin 支持泛型。它也支持声明处型变和使用处型变。Kotlin 也没有通配符类型。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。

花括号是必需的吗？

是。

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、注解以及 语言引文（language quotations）。

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。有提供 CommonJS 和 AMD 支持的计划。

对比Java

Kotlin解决了一些Java中难搞的问题

Kotlin 修复了Java中一系列长期困扰我们的问题

- [类型系统](#)控制了空引用的发生.
- [没有原始类型](#)
- Kotlin中数组是[不变的](#)
- 相对于Java的SAM-conversions, Kotlin有更加合适的[函数类型](#)
- 使用没有通配符的[site variance](#)
- Kotlin 没有检查[异常](#)

Java有但是Kotlin没有的东东

- [检查异常](#)
- [原始类型](#) , 不是类
- [静态成员](#)
- [未私有化字段](#)
- [通配符类型](#)

Kotlin有但是Java没有的东东

- [Lambda 表达式](#) + [内联函数](#) = 高性能自定义结构
- [扩展函数](#)
- [Null 安全性](#)
- [智能类型转换](#)
- [字符串模板](#)
- [属性](#)
- [主构造方法](#)
- [一级委托](#)
- [变量和属性类型的类型推断](#)
- [单例](#)
- [声明点变化 & 类型预测](#)
- [范围表达式](#)
- [运算符重载](#)
- [伴生对象](#)
- [Data classes](#)
- [Separate interfaces for read-only and mutable collections](#)

对比Scala

The main goal of the Kotlin team is to create a pragmatic and productive programming language, rather than to advance the state of the art in programming language research. 稍作考虑, 如果你对Scala已相当得心应手, 你或许不需要再学习Kotlin

Scala有什么Kotlin没有的

- 隐式转换, 限定性因素, 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code中具体发生了什么
 - 为了做到功能多样性, Kotlin会使用[扩展函数](#).
- 重写类型成员
- 路径依赖性类型
- 宏
- 存在类型
 - [类型预测](#) 是一种非常特殊的情况
- 特性初始化的复杂逻辑
 - 参照 [类和接口](#)
- 自定义符号运算
 - 参照 [运算符重载](#)
- 嵌入式 XML
 - 参照 [类型安全 Groovy风格 builders](#)
- 结构类型
- 值类型
 - We plan to support [Project Valhalla](#) once it is released as part of the JDK
- Yield operator
- Actors
 - Kotlin supports [Quasar](#), a third-party framework for actor support on the JVM
- 并行集合
 - Kotlin supports Java 8 streams, which provide similar functionality

Kotlin有什么Scala没有的

- [零开销 空安全](#)
 - Scala 的 Option, 它是语法和运行时包装
- [智能转换](#)
- [Kotlin的内联函数助力非局部跳跃](#)
- [第一类 授权](#). Also implemented via 3rd party plugin: Autoproxy
- [Member references](#) (also supported in Java 8).

