



Kotlin Language Documentation

Table of Contents

| | |
|----------------------|----|
| 开始 | 4 |
| 基本语法 | 4 |
| Kotlin特点 | 9 |
| 编码约定 | 13 |
| 基础 | 14 |
| 基本类型 | 14 |
| 包 | 19 |
| 控制流 | 20 |
| 返回和跳转 | 23 |
| 类和对象 | 25 |
| 类和继承 | 25 |
| 属性和字段 | 31 |
| 接口 | 35 |
| 可见性修饰符 | 37 |
| 扩展 | 39 |
| 数据类 | 42 |
| 泛型 | 44 |
| 泛型方法 | 47 |
| 泛型约束 | 48 |
| 嵌套类 | 49 |
| 枚举类 | 50 |
| 对象表达式和对象声明 | 52 |
| 委托 | 55 |
| 委托属性 | 56 |
| 函数和Lambad表达式 | 60 |
| 函数 | 60 |
| 高阶函数和lambda表达式 | 65 |
| 内联函数 inline | 69 |
| 其他 | 72 |

| | |
|---------------------------|-----|
| 多重申明 | 72 |
| 范围 | 74 |
| 类型的检查与转换 | 79 |
| This表达式 | 81 |
| 等式 | 82 |
| 运算符重载 | 83 |
| Null 安全性 | 86 |
| 异常 | 89 |
| 注解 | 91 |
| 反射 | 94 |
| 动态类型 | 102 |
| 参考 | 103 |
| 互操作 | 105 |
| Java交互 | 105 |
| 工具 | 115 |
| 生成kotlin代码文档 | 115 |
| 使用 Maven | 117 |
| Using Ant | 121 |
| Using Griffon | 124 |
| 使用 Gradle | 125 |
| Kotlin and OSGi | 129 |
| 常见问题 | 131 |
| FAQ | 131 |
| Comparison to Java | 133 |
| Comparison to Scala | 134 |

开始

基本语法

定义包

包的声明应处于文件顶部：

```
package my.demo

import java.util.*

// ...
```

包的结构并不需要与文件夹路径完全匹配：源代码可以在文件系统的任意位置

参阅 [Packages](#).

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，返回值自动推断的函数：

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

不确定返回值 `Unit` 的函数：

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

`Unit` 类型的返回，在函数定义中可以省略：

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

参阅 [Functions](#).

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 // `Int` 类型自动推断（5 默认是 `Int`）
x += 1
```

参阅 [Properties And Fields](#).

使用字符串模板

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

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

参阅 [String templates](#).

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-expressions](#).

Using nullable values and checking for `null`

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 A reference must be explicitly marked as nullable when `null` value is possible.

返回 `null` 假如 `str` 的内容不是数字：

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

返回值可以是 `null` 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 会自动变成非空(non-nullable)值
        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 会自动变成非空值
print(x * y)
```

参阅 [Null-safety](#).

Using type checks and automatic casts

`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, 无需强制转换

```
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` 在一下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 [Classes](#) and [Type casts](#).

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 (ranges)

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 Ranges](#).

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(`.contains`)某实例 `in` :

```
if (text in names) // 自动调用 names.contains(text)
    print("Yes")
```

使用字面量函数(方便的 Higher-order 函数)来过滤(filter)和变换(map)集合:

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

参阅 [Higher-order函数及Lambda表达式](#).

Kotlin特点

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，pull一个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：

- getters (还有 setters)，所有以 `var's` 标记的类属性都会自动生成
- `equals()`
- `hashCode()`
- `toString()`
- `copy()`
- `component1()` , `component2()` , ..., 所有属性都会生成 (参阅 [Data classes](#))

函数默认参数

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

过滤链表

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

或者可以更短：

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

String内插入String

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

类型判断

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

遍历 map/list 中的键值对

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

k, v 可以改成任意名字.

使用区间 ranges

```
for (i in 1..100) { ... }  
for (x in 2..10) { ... }
```

只读链表

```
val list = listOf("a", "b", "c")
```

只读表

```
val map = mapOf("a" to 1, "b" to 2, "c" to 3)
```

表的访问与赋值

```
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)
```

files?.size 等价于 if (files != null) files.size else null

If not null and else 缩写

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

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

if null 缩写

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

if not null 缩写

```
val data = ...

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

when表达式具有返回值

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

‘try/catch’ 表达式也具有返回值

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

    // Working with result
}
```

‘if’ 表达式你猜

```

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，可以轻松实现类**Builder**模式的代码风格

```

fun arrayOfMinusOnes(size: Int) {
    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kotlin风格一起使用的时候，能简化代码，比如和 **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")
}

```

编码约定

此页面包含当前 Kotlin 语言的代码风格

命名风格

如果拿不准的时候，默认使用Java的代码风格，比如：

- 使用驼峰法命名 (避免命名含有下划线)
- 类、接口、枚举等类型名字以大写字母打头
- 方法和属性使用小写字母打头
- 使用4个空格作为缩进
- `public` 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

类和超类、接口等以大写字母开头的类型与冒号之间应留有空格，实例这种以小写字母开头的与冒号之间不留空格：

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

Lambda表达式

在lambda表达式中, 大括号旁要加空格, 箭头 `->` 旁也要加空格以区分参数与代码体 lambda表达应尽可能不要写在括号中

```
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` 来替代自定义的明确的参数名 在嵌套的lambda表达式中，参数应明确声明

Unit

如果函数返回 Unit 类型，那么返回的类型忽略掉，不必明确写出：

```
fun foo() { // ": Unit" 忽略了  
}
```

基础

基本类型

在Kotlin中,所有东西都是对象,所以我们可以调用成员函数和属性的任何变量对象。有些类型是内置的,他们的实现被优化过,但是用户看起来他们就像普通的类。本节我们会描述这些类型: numbers, characters, booleans 和 arrays.

Numbers

Kotlin处理numbers和Java很接近,但是并不完全相同。例如,对于numbers没有隐式扩大转换(如java中int可以隐式变为long),在一些情况下文字的使用有所不同。

对于numbers Kotlin提供了如下的内置类型 (与Java很相近):

| Type | Bitwidth |
|--------|----------|
| Double | 64 |
| Float | 32 |
| Long | 64 |
| Int | 32 |
| Short | 16 |
| Byte | 8 |

注意在kotlin中 characters 不是 numbers

字面量

下面是一些常量的写法:

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

注意: 不支持8进制

Kotlin 同样支持浮点数的常规表示方法:

- Doubles `123.5`, `123.5e10`
- Floats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的原始类型,除非我们需要一个可空的引用 (例如int?) 或泛型。后者情况下数字被装箱 (指的是赋值的时候把实例复制了一下, 不是相同实例)。

装箱数字不会保存它的实例:

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

另一方面它们值相等:

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

显示转换

由于不同的存储方式小的类型并不是大类型的子类型。如果它们的话, 就会出现下述问题 (下面的代码不能通过编译) :

```
// Hypothetical code, does not actually compile:
val a: Int? = 1 // A boxed Int (java.lang.Integer)
val b: Long? = a // implicit conversion yields a boxed Long (java.lang.Long)
print(a == b) // Surprise! This prints "false" as Long's equals() check for other part
to be Long as well
```

假设这样是可以的, 这里我们就悄无声息的丢掉了一些数据.

因此较小的类型不能隐式转换为较大的类型。因此我们不能声明一个 `Byte` 类型给一个 `Int` 变量, 在不进行显示转换的情况下。

```
val b: Byte = 1 // OK, literals are checked statically
val i: Int = b // ERROR
```

我们可以显示转换的去扩大类型

```
val i: Int = b.toInt() // OK: explicitly widened
```

每个number类型支持如下的转换:

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

失去隐式类型转换, 其实并没有带来多少困扰, 因为使用字面量的时候是没有代价的, 因为字面量的类型是推导出来的; 另一方面, 算数运算操作都针对不同类型的参数做好了重载, 比如:

```
val l = 1.toLong() + 3 // Long + Int => Long
```

运算符

Kotlin支持标准的算数操作符，并在相应的类上定义为成员函数（但编译器会针对运算进行优化，将函数调用优化成直接的算数操作）。查看 [Operator overloading](#).

对于按位操作(bitwise operation)，没有特别的符号来表示，而是直接使用命名函数：

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

这是完整的位运算操作 (只能对 `Int` 或者 `Long` 使用):

- `shl(bits)` – signed shift left (Java's `<<`)
- `shr(bits)` – signed shift right (Java's `>>`)
- `ushr(bits)` – unsigned shift right (Java's `>>>`)
- `and(bits)` – bitwise and
- `or(bits)` – bitwise or
- `xor(bits)` – bitwise xor
- `inv()` – bitwise inversion

Characters

Characters 用 `Char` 来表示. 像对待numbers那样就行

```
fun check(c: Char) {  
    if (c == 1) { // ERROR: incompatible types  
        // ...  
    }  
}
```

用单引号表示一个Character，例如: `'1'`, `'\n'`, `'\uFF00'`. 我们可以调用显示转换把Character转换为 `Int`

```
fun decimalDigitValue(c: Char): Int {  
    if (c !in '0'..'9')  
        throw IllegalArgumentException("Out of range")  
    return c.toInt() - '0'.toInt() // Explicit conversions to numbers  
}
```

像numbers, characters是被装箱当使用一个可空的引用. 这样实例不会被保存。

Booleans

类型 `Boolean` 有两个值: `true` 和 `false`.

Booleans使用nullable时候Boolean也会被装箱.

内置对Booelan的操作

- `||` – 短路或
- `&&` – 短路与

数组

数组在Kotlin中使用 `Array` 类来表示, `Array` 类定义了`set`和`get`函数(使用时可以用 `[]`, 通过符号重载的约定转换), 和 `size` 等等一些有用的成员函数:

```
class Array<T> private () {
    fun 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 `array()` 来创建一个包含数值的数组, `array(1, 2, 3)` 创建了 `array [1, 2, 3]`. 或者, `arrayOfNulls()` 可以创建一个指定大小, 元素都为空的数组。
或者使用函数来创建一个数组:

```
// Creates an Array<String> with values ["0", "1", "4", "9", "16"]
val asc = Array(5, {i -> (i * i).toString()})
```

综上, `[]` 操作符代表了成员函数 `get()` 和 `set()` .

注意: 与Java不同的是, Kotlin中数组不可变. 这意味着我们不能声明 `Array<String>` 到 `Array<Any>`, 否则可能会产生一个运行时错误(但是你可以使用 `Array<out Any>`, 查看 [Type Projections](#)).

Kotlin有专门的类来表示原始类型的数组, 避免了装箱开销: `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 = intArray(1, 2, 3)
x[0] = x[1] + x[2]
```

字符串

字符串用 `String` 表示。字符串是不可变的。

字符串的原始字符可以使用操作符访问: `s[i]` . 字符串可以使用 `for` 循环遍历:

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

字符串字面量

Kotlin有两种类型的字符串: 转义字符串可能由转义字符、原生字符串、换行和任意文本.转义字符串很像java的String:

```
val s = "Hello, world!\n"
```

转义方式采用传统的反斜杠.

原生字符串使用三个引号(“”)包括, 内部没有转义, 可以包含换行和任何其他文本:

```
val text = ""
for (c in "foo")
    print(c)
""
```

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 \$ 符号开始，包含一个简单的名称：

```
val i = 10
val s = "i = $i" // evaluates to "i = 10"
```

或者用花括号扩起来，内部可以是任意表达式：

```
val s = "abc"
val str = "$s.length is ${s.length}" // evaluates to "abc.length is 3"
```

校对BY 空白

包

源文件通常以包声明开头:

```
package foo.bar

fun baz() {}

class Goo {}

// ...
```

源文件所有的(无论是类或者函数)被包声明覆盖. 所以 `baz()` 的全名是 `foo.bar.baz`, `Goo` 的全名是 `foo.bar.Goo`.

如果没有明确声明文件属于“default”且包没有名称.

导入

除了模块定义的默认导入之外, 每个源文件也可以声明自己的导入。导入语句的语法定义描述在[grammar](#).

可以导入一个单独的名称, 如.

```
import foo.Bar // Bar is now accessible without qualification
```

也可以导入一个作用域下的所有内容(包、类、对象等):

```
import foo.* // everything in 'foo' becomes accessible
```

如果出现名称冲突, 可以使用 `as` 关键字来重命名导入的名称:

```
import foo.Bar // Bar is accessible
import bar.Bar as bBar // bBar stands for 'bar.Bar'
```

可见性和包嵌套

如果顶层声明是`private`, 它将是私有的(查看 [Visibility Modifiers](#)). 尽管Kotlin中可以包嵌套, 如包 `foo.bar` 是 `foo` 的一个成员, 但是`private` 仅仅可以被它的子包所见.

注意外部包成员不是默认引入的, 例如, 在 `foo.bar` 包的文件中我们不能在不引入的情况下访问 `foo`.

控制流

If表达式

在Kotlin中, `if`是一个表达式,它会返回一个值. 因此就不需要三元运算符 (如 `?` 三元表达式), 因为使用 `if` 就可以了。

```
// Traditional usage
var max = a
if (a < b)
    max = b

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

// As expression
val max = if (a > b) a else b
```

`if`的分支可以是代码段, 最后一行的表达式作为段的返回值:

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

当`if`仅仅有一个分支, 或者其中一个分支的返回结果 `Unit`, 它的类型 `Unit` .

See the [grammar for if](#).

When表达式

`when` 替代了c语言风格的switch操作符. 最简单的形式如下:

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

`when` 将它的参数和所有的分支条件进行比较, 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, 则忽略单个分支的值。(就像`if`, 每一个分支可以是一个代码块, 它的值是最后的表达式的值.)

`else` 分支将被执行如果其他分支都不满足条件。 如果 `when` 作为一个表达式被使用, `else` 分支是必须的, 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")
}
```

我们可以在判断分支条件的地方使用任何表达式, 而不仅仅是常量(和switch不同):

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

我们也可以检查一个值 `in` 或者 `!in` 一个 [范围](#) 或者集合:

```
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")
}
```

另一种用法是可以检查一个值 `is` 或者 `!is` 某种特定类型. 注意, 由于 [smart casts](#), 你可以访问该类型的方法和属性而不用额外的检查。

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

`when` 也可以用来替代 `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值, 而当一个分支的条件返回 `true` 时, 则调用该分支:

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

查看 [grammar for when](#).

For循环

`for` 循环可以对任何提供迭代器(iterator)的集合进行遍历, 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) {  
    // ...  
}
```

像上面提到的一样, `for`可以循环遍历任何提供了迭代器的集合。例如:

- 有一个成员函数或者扩展函数 `iterator()` ,它返回一个类型
- 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` .

如果你想要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“遍历一个范围”的函数会被编译器优化, 不会产生额外的对象。

See the [grammar for for](#).

While循环

`while` 和 `do..while` 的使用方法和其他语言一致

```
while (x > 0) {  
    x--  
}  
  
do {  
    val y = retrieveData()  
} while (y != null) // y is visible here!
```

查看 [grammar for while](#).

Break和continue在循环中的使用

在循环中Kotlin支持传统的`break`和`continue`操作符. 查看[Returns and jumps](#).

返回和跳转

Kotlin 有三种跳出结构

- `return`.默认情况下, 从最近的一个封闭的方法或者 [方法表达式](#)跳出.
- `break`.终止最近的封闭循环
- `continue`.直接进入循环体的下次循环

中断和继续标签

在Kotlin中任何表达式都可以用 `label` (标签) 来标记。

label的格式是被 '@' 标识符标记, 例如: `abc@`, `fooBar@` 都是有效的label (参见[语法](#))

你可以在一个方法前面放一个label。 `kotlin loop@ for (i in 1..100) { // ... }`

现在, 我们可以将label与 `break` 或者 `continue` 一起使用:

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

`break` 执行后将跳转到标记处。

`continue` 将进入循环体的下次循环

返回标签

在Kotlin里, 函数字面量、局部函数和对象表达式等函数都可以被嵌套在一起 适当的返回方式允许我们从外部方法返回值

带标签的 `return`, 最重要的一个用途, 就是让我们可以从函数字面量中返回。

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

这个 `return` 表达式从最近的封闭的方法中返回, 例如 'foo'。

(注意, 非全局的返回只支持内部方法, 参见[内联方法](#).) 如果我们只是需要跳出内部方法, 我们必须标记它并且返回这个标签

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

现在只是从内部方法返回。有时候用匿名的标签将会更加方便 像这样和方法同名的标签是可以的

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

通常，我们用一个[方法表达式](#)替代内部匿名方法。在方法内部声明一个`return`将从其内部返回

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

当要返回一个值得时候，推荐使用描述性的返回，例如：`kotlin return@a 1`

意思是“返回被标记为‘@a’值是‘1’的标签，而不是像‘(@a 1)’的一个标签表达式”

被命名的方法自动被定义成为标签

```
fun outer() {  
    fun inner() {  
        return@outer // the label @outer was defined automatically  
    }  
}
```

翻译BY S_arige

类和对象

类和继承

类

类声明Kotlin使用关键字`class` `{:.keyword}`

```
class Invoice {  
}
```

这个类声明被花括号包围，包括类名、类头(指定其类型参数,主构造函数等)和这个类的主干。类头和主干都是可选的； 如果这个类没有主干，花括号可以被省略。

```
class Empty
```

构造

在Kotlin中的类可以有主构造函数和一个或多个二级构造函数。主构造函数是类头的一部分:它跟在这个类名后面（和可选的类型参数）

```
class Person constructor(firstName: String) {  
}
```

如果这个主构造函数没有任何注解或者可见的修饰符，这个`constructor`关键字可以被省略

```
class Person(firstName: String) {  
}
```

这个主构造函数不能包含任何的代码。初始化的代码可以被放置在**initializer blocks**（初始的模块），以`init`为前缀作为关键字 `{:.keyword}`

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

请注意，主构造的参数可以在初始化模块中使用。它们也可以在类体内声明初始化的属性：

```
class Customer(name: String) {  
    val customerKey = name.toUpperCase()  
}
```

事实上，声明属性和初始化主构造函数,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或者是只读的

If the constructor has annotations or visibility modifiers, the `constructor` keyword is required, and the modifiers go before it: 如果构造函数有注解或可见性修饰符，这个`constructor`需要被关键字修饰。

```
class Customer public inject constructor(name: String) { ... }
```

更多请查看[Visibility Modifiers](#)

扩展构造函数

类也可以拥有被称为“二级构造函数”(为了实现Kotlin向Java一样拥有多个构造函数)，通常被加上前缀“constructor”

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

如果类有一个主构造函数,每个二级构造函数需要委托给主构造函数,直接或间接地通过另一个二级函数。委托到另一个使用同一个类的构造函数用`this`关键字

```
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构造函数（原发性或继发性），这将有一个生成的主构造函数不带参数。构造函数的可见性是`public`。如果你不希望你的类有一个公共构造函数，你需要声明与非缺省可见一个空的主构造函数：

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

注意在JVM上，如果所有的主构造函数的参数有默认值，编译器会产生一个额外的参数的构造函数，将使用默认值。这使得更易于使用kotlin与通过参数构造函数创建类的实例，如使用Jackson或JPA库的时候。

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

创建类的实例

要创建一个类的实例，我们调用构造函数，就好像它是普通的函数：

```
val invoice = Invoice()

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

注意Kotlin不能有“new”关键字

类成员

类可以包括

- 构造和初始化模块
- [函数](#)
- [属性](#)
- [匿名和内部类](#)
- [对象声明](#)

继承

在Kotlin所有的类中都有一个共同的父类 `Any`，这是一个默认的父类且没有父类型声明：

```
class Example // Implicitly inherits from Any
```

`Any` 不属于 `java.lang.Object` ;特别是，它并没有任何其他任何成员，甚至连 `equals()`，`hashCode()` 和 `toString()` 都没有。

请参阅[Java的互操作性](#)更多的细节部分。

要声明一个明确的父类，我们把类型放到类头冒号之后：

```
open class Base(p: Int)

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

如上所见，父类可以（并且必须）在声明继承的地方，用原始构造函数初始化。

如果类没有主构造，那么每个次级构造函数初始化基本类型 使用`super{ :keyword}`关键字，或委托给另一个构造函数做到这一点。注意，在这种情况下，不同的二级构造函数可以调用基类型的不同的构造：

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

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

父类上的`open{ :keyword}`标注可以理解为Java中`final{ :keyword}`的反面，它允许其他类 从这个类中继承。默认情况下，在Kotlin所有的类都是`final`，对应于 [Effective Java](#) 书中的17条：设计并显示标注继承，否则就禁止它。

覆盖成员

我们之前提到过，Kotlin力求清晰显式。不像Java中，Kotlin需要明确的 标注覆盖的成员（我们称之为`open`）和重写的函数。（继承父类并覆盖父类函数时，Kotlin要求父类必须有`open`标注，被覆盖的函数必须有`open`标注，并且子类的函数必须加`override`标注。）：

```
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。如果父类的这个函数没有标注`open`，则子类中不允许定义同名函数，不论加不加`override`。在一个`final`类中（即没有声明`open`的类），函数上也不允许加`open`标注。

成员标记为`override{:.keyword}`的本身是开放的，也就是说，它可以在子类中重写。如果你想禁止重写的，使用`final{:.keyword}`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：比如某些情况下可以使用Java进行hack，再用Kotlin调用；或者使用面向切面的框架(Aspect)。（请参阅[Java的互操作](#)）

重写的规则

在Kotlin中，实现继承的调用通过以下规则： 如果一个类继承父类成员的多种实现方法，可以直接在子类中引用， 它必须重写这个成员，并提供其自己的实现（当然也可以使用父类的）。 为了表示从中继承的实现而采取的父亲类型，我们使用`super{:.keyword}`在尖括号，如规范的父亲名 `super<Base>`：

```

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

interface B {
    fun f() { print("B") } // interface members are 'open' by default
    fun b() { print("b") }
}

class C() : A(), B {
    // The compiler requires f() to be overridden:
    override fun f() {
        super<A>.f() // call to A.f()
        super<B>.f() // call to B.f()
    }
}

```

类C同时继承A和B是可以的，而且我们在调用a()和b()函数时没有任何问题，因为他们在C的基类中只有一个实现。但是f()函数则在A,B中都有实现，所以我们必须在C中覆盖f()，并且提供我们的实现以消除歧义。

抽象类

类和其中的某些实现可以声明为`abstract{ : .keyword}`。抽象成员在本类中可以不用实现。。因此，当一些子类继承一个抽象的成员，它并不算是一个实现：

```

abstract class A {
    abstract fun f()
}

interface B {
    open fun f() { print("B") }
}

class C() : A(), B {
    // We are not required to override f()
}

```

Note that we do not need to annotate an abstract class or function with `open` – it goes without saying.

We can override a non-abstract open member with an abstract one

需要注意的是，我们并不需要标注一个抽象类或者函数为`open` - 因为这不言而喻。

我们可以重写一个`open`非抽象成员使之为抽象的。

```

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

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

```

同伴对象

在Kotlin中，不像Java或C#，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调用的函数，而不依赖一个类的实例，但需要访问的内部一个类（例如，一个工厂方法），你可以写为[对象声明]（object_declarations.html）中的一员里面的那个类。

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

更具体地讲，如果你声明一个[同伴对象](#)在你的类中，你就可以在Java/C#中调用与它的成员方法相同的语法的静态方法，只使用类名作为一个修饰语。

翻译BY S_arige

属性和字段

声明属性

Kotlin的属性. 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" // Initializer required, not a nullable type
    private set // the setter is private and has the default implementation

var setterWithAnnotation: Any?
    @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` 符号。

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

上面的 `field` 字段只能用在访问属性

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.

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。更多信息在[这里](#)。

翻译BY 空白

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。它的属性必须声明为 `abstract`。

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 {  
    fun bar() {  
        // body  
    }  
}
```

接口属性

接口只能定义无状态（stateless）的属性。

```
interface MyInterface {  
    val property: Int // abstract  
  
    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。

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。

这通过特殊的声明调用`_extensions_`.现在，Kotlin支持`_extension functions_` 和 *extension properties*.

扩展方法

声明一个扩展方法，我们需要用一个 *receiver type*,也就是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(x: Int, y: Int) {
    val tmp = this[x] // 'this' corresponds to the list
    this[x] = this[y]
    this[y] = 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(x: Int, y: Int) {
    val tmp = this[x] // 'this' corresponds to the list
    this[x] = this[y]
    this[y] = 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去调用这个新方法。

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。如果有一个成员方法和相同类型的扩展方法都适用于给定的参数，**成员方法总是赢**.例如：

```
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接受者

注意扩展可被定义为null的接受类型。这样的扩展被称为对象变量。即使他的值是null，你可以在方法体内检查 `this == null`，这也允许你调用Kotlin中的`toString()`在没有检查null的时候：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张方法和属性

```
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，只需用类名作为限定符去调用他们 `kotlin 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](#)

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。

翻译By S_arige

数据类

我们经常创建一些只是处理数据的类。在这些类里的功能经常是衍生自他们所持有的数据。在Kotlin中，这样的类可以被称为 `data`：

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

这被叫做一个 *数据类*。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()` ,
- `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()` 方法 .

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

差异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，取而代之的是：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，带上界的通配符类型 (wildcard) 是的类型更加多样性。

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 s` 集合，从中读取 `Object s`。反过来，如果只能向集合中放入元素，就可以获取 `Object s`并向其中放入 `String s`：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。

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，与可变性并不冲突。

声明位置变量

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：**声明变量位置**：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> {
    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**可变**，或 T 是一个**可变**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。

输出改变被成为**输出注释**，同时由于它在参数类型位置被提供，所以我们讨论**声明变量位置**。与Java的**使用变量位置**，通配符是的类型可变的用法相反。

另外针对**输出**，Kotlin又补充了一项变量注释：**输入**。它是的变量类型**反变**：只可以被消费而不可以被生产。例如一个反变类 Comparable：

```

abstract class Comparable<in T> {
    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**输入**和**输出**是自解释（他们可以成功地被应用于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

使用变量位置声明的类型预测

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？例如：

```

class Array<T>(val length: 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.length == to.length)
    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> = array(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**输出-预测**（对象并没有使用任何未知类型），预测具有相对参数的上界，例如大多数情况下的 out Any?。Kotlin提供了一种简单的方法，称为**星预测**：Foo<*> 代表 Foo<out Bar>，Bar 是 Foo 参数类型的上界。

笔记：星预测很像Java中的raw类型，但是比raw类型更加安全。

泛型方法

不仅仅是类具有类型参数，方法也具有。通常，我们将类型参数放置在方法名的后面，并用尖括号引起：

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

但是，对于 [Extension functions](#) 可能并不需要在确定接收到参数类型之前声明他的类型变量。所以，Kotlin允许以下表达方法：

```
fun <T> T.basicToString(): String {
    return typeinfo.typeinfo(this) + "@ " + System.identityHashCode(this)
}
```

如果参数类型是从调用方传递而来，那么它仅仅只能在方法名之后被声明：

```
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(list(1, 2, 3)) // OK. Int is a subtype of Comparable<Int>  
sort(list(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() }  
}
```

翻译By 咩咩

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://kotlinlang.org/docs/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。

翻译By EasonZhou

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方法去获得他们的名字和位置。

```
name(): String  
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。

翻译By EasonZhou

对象表达式和对象声明

有些时候我们需要创建一个对象对某些类做稍微改变，而不用为了它明确定义一个新的子类。

Java把这处理为*匿名内部类*。在Kotlin稍微归纳为*对象表达式*和*对象声明*。

对象表达式

创建一个继承自一些类型的内部类的对象，我们可以这么写：

```
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被传递下去。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  
}
```

或许如果我们只需要“仅仅是一个对象”，没有父类的，我们可以简单这么写：

```
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，在对象表达里代码可以使变量与作用域联系起来（与Java不同的是，这不是受final变量限制的。）

```

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）中使得单例模式很容易声明。

```

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“表达”。我们不能把它归于变量，但我们可以通过它的名字来指定它。这些对象可以有父类型：

```

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

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

```

NOTE: 对象声明不能是本地的（例如：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被称为使用类名称作为限定符：

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

The name of the companion object can be omitted, in which case the name `Companion` will be used:

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
```

记住，虽然伴生对象的成员在其他语言中看起来像静态成员，但在日常使用中它们仍然是实体的实例成员，而且比如说能继承接口：

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

然而，在JVM中你可以有些产生自真正的静态方法和域的伴生对象的成员，如果你使用 `@platformStatic` 注解。可以从[Java interoperability](#) 这里查看详情。

对象表达式与对象声明区别

这是一个重要的的不同在对象表达式与对象声明上

- 对象声明被**lazily**初始化，当被第一次访问的时候
- 对对象表达被**立即**执行（被初始化），当它被用到的时候

翻译By Wahchi

委托

委托类

[委托模式](#)是实现继承的一个有效方式。

Kotlin原生支持它。

一个类 `Derived` 可以从一个接口 `Base` 继承并且委托所有的共有方法为具体对象。

```
interface Base {  
    fun print()  
}  
  
class BaseImpl(val x: Int) : Base {  
    override fun print() { print(x) }  
}  
  
class Derived(b: Base) : Base by b  
  
fun main() {  
    val b = BaseImpl(10)  
    Derived(b).print() // prints 10  
}
```

在父类 `Derived` 中的 `by`-语句表示 `b` 将会被 储存在 `Derived` 的内部对象中

并且编译器会把所有 `Base` 的方法生成给最终的 `b`。

翻译By EasonZhou

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之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。
- 可控性 (observable properties)：监听器得到关于这个特性变化的通知，
- 把所有特性储存在一个图型结构中，而不是分开每一条。

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()) 协同的属性会被委托给它。特性委托不必实现任何的接口，但是需要提供一个 get() 方法(和 set()) — 对于 var's)。例如：

```
class Delegate {  
  
    fun get(thisRef: Any?, property: PropertyMetadata): String {  
        return "$thisRef, thank you for delegating '${property.name}' to me!"  
    }  
  
    fun set(thisRef: Any?, property: PropertyMetadata, 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 get() 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!

相同的，当我们给 p 赋值，set() 方法就被调用。前两个参数是一样的，第三个参数代表被赋予的值：

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

打印结果：

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

属性委托要求

下面见到介绍委托对象的要求。

对于一个 *只读* 属性 (如 val)，一个委托一定会提供一个 get 函数来处理下面的参数：

- 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 属性包含数据 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`), 一个委托需要提供_额外_的函数 `set` 来获取下面的参数:

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

标准委托

标准库中 `kotlin.properties.Delegates` 对象对于一些有用的委托提供了工厂 (factory) 方法。

延迟属性 `Lazy`

函数 `Delegates.lazy()` 会在接受一个变量而后返回一个执行延迟属性的委托: 第一个调用 `get()` 执行变量传递到 `lazy()` 并记录结果, 后来的 `get()` 调用只会返回记录的结果。

```
import kotlin.properties.Delegates

val lazy: String by Delegates.lazy {
    println("computed!")
    "Hello"
}

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

By default, the evaluation of lazy properties is **synchronized**: the value is computed only in one thread, and all threads will see the same value. If the synchronization of initialization delegate is not required, so that multiple threads can execute it simultaneously, pass `LazyThreadSafetyMode.PUBLICATION` as a parameter to the `lazy()` function. And if you're sure that the initialization will always happen on a single thread, you can use `LazyThreadSafetyMode.NONE` mode, which doesn't incur any thread-safety guarantees and the related overhead.

如果你需要 线程安全, 使用 `blockingLazy()`: 它会进行同样的操作, 但是能够保证数值将会只在一个线程中计算, 同时所有线程会看到同样的数值。

观察者 `Observable`

`Delegates.observable()` 需要两个参数: 初始值和修改后的处理(handler)。这个 handler 会在每次赋值的时候被属性调用 (在工作完成前)。它有三个变量: 一个被赋值的属性, 旧的值和新的值:

```

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

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

```

结果:

```

<no name> -> first
first -> second

```

如果你想截取它的分配并取消它, 就使用 `vetoable()` 取代 `observable()`.

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

`Delegates.mapVal()` 用到一个map的实例, 同时返回一个从map中以把属性名作为关键字读取属性的委托。这有很多在程序中应用的例子, 例如解析JSON数据或者做其他动态的事情: `kotlin class User(val map: Map<String, Any?>) { val name: String by Delegates.mapVal(map) val age: Int by Delegates.mapVal(map) }`

在这个例子中, 构造函数持有一个map:

```

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

```

委托会从这个图中取值 (通过属性的名字, 也就是string关键字):

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

对于 *var's* 我们可以使用 `mapVar()` (注意这里需要一个 `MutableMap` 而不是只读的 `Map`).

翻译By EasonZhou

函数和Lambad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 {  
}
```

参数

函数参数是使用Pascal符号定义,即 *name: type*.

参数用逗号隔开。每个参数必须显式类型。

```
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。

参数命名

函数参数可以在调用函数时被命名。这是非常方便的，当一个函数有大量的参数或默认的。

给出下面的函数：

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

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 = '_')
```

Unit返回函数

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

在某些情况下,一个显式地返回类型是必需的:

- 函数表达式是公共的或是受保护的.这些都被认为是公共API的一部分.由于没有显式地返回类型使得它有可能更容易更改类型.这就是为什么显式地类型都需要相同的原因[属性](#).
- 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,返回类型将不明显的阅读器(有时甚至为编译器)。

可变的参数(可变参数)

函数的最后一个参数可以使用`vararg`注释

```
fun asList<T>(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 `ts` 变量在上面的例子是 `Array<out T>` 类型。

只有一个参数可以标注为 `vararg`.这可能是最后一个参数或前一个最后的,如果最后一个参数类型(允许一个lambda括号外传递)

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

函数作用域(函数范围)

在Kotlin中,函数可以在文件头部声明,这意味着您不需要创建一个类来保存一个函数,类似的语言如Java, C # 或Scala。此外除了头部函数功能, Kotlin函数也可以在局部声明,作为成员函数和扩展功能。

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, 关闭),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])
}
```

从外部函数使用局部函数甚至可以返回[正确的表达式](#)

```
fun reachable(from: Vertex, to: Vertex): Boolean {
    val visited = HashSet<Vertex>()
    fun dfs(current: Vertex) {
        // here we return from the outer function:
        if (current == to) return@reachable true
        // And here -- from local function:
        if (!visited.add(current)) return
        for (v in current.neighbors)
            dfs(v)
    }

    dfs(from)
    return false // if dfs() did not return true already
}
```

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,在函数之后使用尖括号和在参数值之前。

```
fun singletonArray<T>(item: T): Array<T> {
    return Array<T>(1, {item})
}
```

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](#)

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符号

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

插入表示法

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](#)
- 他们有一个参数

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

// call extension function using infix notation

1 shl 2

// is the same as

1.shl(2)
```

翻译By Jacky Xu

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 lock<T>(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 [函数数字量](#) (通常被称为 *lambda* 表达式):

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

函数数字量 [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:

- 一个函数数字面值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, 那么我们可以省略括号

```
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}
```

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，用 `it` 表示。

```
ints map {it * 2} // Infix call + Implicit 'it'
```

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。

函数数字面量和函数表达式

一个函数数字面量和函数表达式是一个“匿名函数”，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，也就是函数数字面量。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 max<T>(collection: Collection<out 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，目的就是能通过[命名参数](#)调用。

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

函数数字面量语法

函数数字面量的全部语法形式，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}
```

一个函数数字面值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参数类型。函数体后面有一个 `->` 符号。如果我们把所有的可选注释都留了出来,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,一个函数数字面量只有一个参数。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,该函数数字面量参数可以放在括号外的参数列表。语法细则详见 [call-Suffix](#)。

函数表达式

上述函数数字面量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: 函数表达式。

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

函数表达式看起来很像是一个正则函数声明,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正则函数方式相同,只是如果参数类型能沟通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)
```

函数表达式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,函数表达式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函数数字面量。

另一个函数数字面量和函数表达式区别是 [non-local returns](#) 的行为。一个带标签的 `return{:.keyword}` 语句,总是在用 `fun` 关键词声明的函数中返回。这意味着函数数字面中的 `return{:.keyword}` 将在函数闭包中返回。然而函数表达式 `return*{:.keyword}` 的就是在函数表达式中返回。

闭包

一个函数数字面量或表达式(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)
```

扩展函数表达式

除了常规函数，kotlin支持扩展函数。拓展函数字面量是很有用处的，并且支持表达式。他们的一个最重要的例子是[Type-safe Groovy-style builders](#)的使用。

扩展函数表达式不同于一般的函数表达式，它有一个接收器类型规范。

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

接收器类型可仅在函数表达式中指定，而不是在函数字面量中。函数字面量能作为拓展函数表达式，但是只能在接收器类型能够通过上下文推断出来的时候。

扩展函数表达式的类型是接收器函数类型：

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

该函数可以被称为一个点或中缀形式（因为它只有一个参数）：

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

翻译By Airoyee

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。例如：一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)和虚拟调用引入运行时间开销。但是在许多情况下通过内联化匿名函数可以降低这类的开销。我们通过下面的示例函数来分析上面这些内容。如，`lock()` 函数可以被很容易的在申明调用时被内联化。

```
lock(l) {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：

```
lock.lock()
try {
    foo()
}
finally {
    lock.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`：

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

`inline` 关键字注释会影响函数体本身以及传递过来的lambdas: 这些所都会被内联到调用点。

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，尤其是在极其巨大的调用循环中。

禁止内联@noinline

为了预防有时候你只希望一部分的lambdas传递到一个内联函数后背内联，你可以把一些你的函数变量用 `@noinline` 标记：

```
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变量并且没有[泛型变量](#)，编译器会产生一个警告。因为内联一个这样的函数很可能是无意义的(你可以无视这个警告如果你确定内联是必须的)。

非本地的返回

在Kotlin中，我们只能使用一个普通的，无限制的 `return` 来退出一个命名的函数或者函数表达式。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不可能让完整的函数返回：

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

但是如果函数和lambda被传到一个内联函数中，return也可以被内联。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，但是退出闭包的函数)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闭合的：

```
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从在函数体中调用被作为参数传递过来的lambdas，而从其他执行语句中调用，像一个本地对象或者一个嵌套函数。在这种情况下，非本地的控制流程也会被lambdas禁止。为了标识这种情况，lambda参数需要以 `InlineOptions.ONLY_LOCAL_RETURN` 标注：

```
inline fun f(inlineOptions(InlineOption.ONLY_LOCAL_RETURN) 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变量类型：

```
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
}
```

在这里我们访问一个树并且检查每个节点是不是特定的类型。这都没有问题，但是访问点并不理想：

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

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`表示变量类型，于是现在它在函数内部就可以像正常函数一样的访问了。既然这个函数是内联的，没有必须的反射，一般的运算符 `!is` 和 `as` 就开始工作。同时，我们也可以这样调用它：

```
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：`javaClass()` 给我们权限去访问它：

```
inline fun methodsOf<reified T>() = javaClass<T>().getMethods()  
  
fun main(s: Array<String>) {  
    println(methodsOf<String>().joinToString("\n"))  
}
```

正常的函数 (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](#).

翻译By EasonZhou

其他

多重申明

有时把一个对象_分解_成很多变量很比较方便，比如：

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

这种语法叫做_多重申明_。一个多重申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)
```

一个多重申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) 任何可以被放在多重分配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()` .

多重申明对 `for`-loops有效：

```
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, 多重申明在这里也有效。

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: 多重申明和图。

可能最好的遍历一个图的方式就是这样:

```
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图。
- 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:

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

于是你可以自由的使用多重申明 `for-loops` 来操作图(也可以用在数据类实例的集合等)。

翻译By EasonZhou

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,但是最好用于数字的比较。下面是使用范围的例子

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

if (x !in 1.0..3.0) println(x)

if (str in "island".. "isle") println(str)
```

数值范围有一个额外的功能: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

for (x in 1.0..2.0) print("$x ") // prints "1.0 2.0 "
```

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"

for (i in 1.0..2.0 step 0.3) print("$i ") // prints "1.0 1.3 1.6 1.9 "
```

它是如何工作的

在库里有两个接口: `Range<T>` 和 `Progression<N>`。

`Range<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:‘开始’和‘结束’,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。

`Progression<N>` 表示一个等差数列,是数字类型定义。它有“开始”,“结束”和一个非零的“增量”。 `Progression<N>` 是 `Iterable<N>` 的子类,所以它可以用在 `for` { `.keyword` } 循环中或者 `map`, `filter` 等函数中。第一个元素是 `start`, 下一个元素等于前面加上 `increment`。迭代 `Progression` 与Java/JavaScript的 `for` 循环相同:

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

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

对于数字, `..` 操作符创建一个对象既包含 `Range` 也包含 `Progression`。由于 `downTo()` 和 `step()` 函数所以一直是 `Progression`。

范围指标

使用范例

```
// Checking if value of comparable is in range. Optimized for number primitives.
if (i in 1..10) println(i)

if (x in 1.0..3.0) println(x)

if (str in "island".. "isle") println(str)

// Iterating over arithmetical progression of numbers. Optimized for number primitives
// (as indexed for-loop in Java).
for (i in 1..4) print(i) // prints "1234"

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

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

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

for (i in (1..4).reversed()) print(i) // prints "4321"

for (i in (1..4).reversed() step 2) print(i) // prints "42"

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

for (x in 1.0..2.0) print("$x ") // prints "1.0 2.0 "

for (x in 1.0..2.0 step 0.3) print("$x ") // prints "1.0 1.3 1.6 1.9 "

for (x in 2.0 downTo 1.0 step 0.3) print("$x ") // prints "2.0 1.7 1.4 1.1 "

for (str in "island".. "isle") println(str) // error: string range cannot be iterated
over
```

###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
    sometimes
    // 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` `IntRange`是有点复杂:它的实现类是 `Progression<Int>` 和 `Range<Int>`,因为它是自然的遍历的(默认增量值为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()

定义 `rangeTo()` 函数,只要简单地调用构造函数 `*Range` 类,例如:

```

class Int {
    //...
    fun rangeTo(other: Byte): IntRange = IntRange(this, other)
    //...
    fun rangeTo(other: Int): IntRange = IntRange(this, other)
    //...
}

```

downTo()

`downTo()` 的扩展函数可以为任何数字类型定义,这里有两个例子:

```

fun Long.downTo(other: Double): DoubleProgression {
    return DoubleProgression(this, other, -1.0)
}

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

```

reversed()

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

```

fun IntProgression.reversed(): IntProgression {
    return IntProgression(end, start, -increment)
}

fun IntRange.reversed(): IntProgression {
    return IntProgression(end, start, -1)
}

```

step()

step() 扩展函数是为每个 *Range 和 *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(start, end, if (increment > 0) step else -step)
}

fun IntRange.step(step: Int): IntProgression {
    if (step <= 0) throw IllegalArgumentException("Step must be positive, was: $step")
    return IntProgression(start, end, step)
}

```

翻译By 空白

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 Array<Int> -> print(x.sum())
}
```

“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。

翻译By [Wahchi](#)

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[扩展字面函数](#), `this`表示左边的接受者.

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[扩展字面函数](#) 我们使用 `this@label` 作为[label](#):

```
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 {String.() ->
        val d = this // funLit's receiver
        val d1 = this@lambda // funLit's receiver
      }

      val funLit2 = { (s: String) ->
        // foo()'s receiver, since enclosing function literal
        // 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：

```
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
```

如果 `a` 不是 `null` 则调用 `equals(Any?)` 函数，否则检查 `b` 是否等于 `null`。

注意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](#)和[扩展函数](#)，比如二元运算符的左值 and 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.plus()</code> |
| <code>-a</code> | <code>a.minus()</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 `plus()`，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 |
|--------------------|---------------------------|
| <code>a + b</code> | <code>a.plus(b)</code> |
| <code>a - b</code> | <code>a.minus(b)</code> |
| <code>a * b</code> | <code>a.times(b)</code> |
| <code>a / b</code> | <code>a.div(b)</code> |
| <code>a % b</code> | <code>a.mod(b)</code> |
| <code>a..b</code> | <code>a.rangeTo(b)</code> |

编译器只是解决了该表中翻译为列的表达式。

| Expression | Translated to |
|----------------------|-----------------------------|
| <code>a in b</code> | <code>b.contains(a)</code> |
| <code>a !in b</code> | <code>!b.contains(a)</code> |

`in` 和 `!in` 的产生步骤是一样的, 但参数顺序是相反的。

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

方括号被转换为 `get set` 函数。

| 标志 | 翻译为 |
|-------------------------------|--------------------------------------|
| <code>a(i)</code> | <code>a.invoke(i)</code> |
| <code>a(i, j)</code> | <code>a.invoke(i, j)</code> |
| <code>a(i_1, ..., i_n)</code> | <code>a.invoke(i_1, ..., i_n)</code> |

括号被转换为带有正确参数的 `invoke` 函数。

| 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> |

| 表达式 | 翻译为 |
|---------------------|-----------------------------|
| <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。

`==` 运算符有两点不同:

- 它被翻译成一个复杂的表达式, 用于筛选空值, 而且 `null == null` 是真。
- 它需要带有特定签名的函数, 而不仅仅是特定名称的函数, 像下面这样:

```
fun equals(other: Any?): Boolean
```

或者用相同的参数列表和返回类型的扩展函数。

| 标志 | 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，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，叫作 `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()`
- 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 `a`（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()
```

但是如果你想调用 `b` 的一些方法，这将是 unsafe 的，同时编译器会报错：

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

可是我仍然需要调用这些方法，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")
```

需要注意的是其中 `b` 是不可变的，这仅适用（例如，不可变的局部变量，`val` 成员，并且是不可重写的），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.

另外，`!!` 是为了简明，和扩展模拟以前的标准库功能，定义如下：

```
inline fun <T : Any> T?.sure(): T =  
    if (this == null)  
        throw NullPointerException()  
    else  
        this
```

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
```

翻译 by [drakeet](#)

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。

翻译： [cx9527](#)

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
```

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 {}
```

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) }
```

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
}
```

翻译By [Wahchi](#)

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，它必须是可访问的, 它的返回类型是“extension function”，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 compose<A, B, C>(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.

访问一个类的属性成员, 我们这样修饰:

```

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"
}
```

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()
}
```

我们可以简单的这样使用:

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


[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://jetbrains.com/kotlin") {+"Kotlin"}

            // mixed content
            p {
                +"This is some"
                b {+"mixed"}
                +"text. For more see the"
                a(href = "http://jetbrains.com/kotlin") {+"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。(在IDEA中), 可以点击函数名称浏览他们的定义代码。 [这里](#).

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 {
    // ...
}
```

这实际上是一个函数, 其参数是一个[函数数字量](#)(查看[这个页面](#)的详细说明) 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。实际上，它是一个[扩展函数](#)，其接受者类型为 `HTML` (并且返回 `Unit`)。所以，当我们传入一个函数字面量作为参数时，它被认定为一个扩展函数，从而在内部就可以使用 `this` 引用了。

```
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，它们将新建的实例先添加到 `html` 的 `children` 属性上，再返回：

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

fun body(init: Body.() -> Unit) {
    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 initTag<T : Element>(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 `plus()` 来定义的. `plus()` 是抽象类 `TagWithText` (`Title` 的父类) 的成员函数.

```
fun String.plus() {
  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[扩展函数数字面量](#)技术

```
package com.example.html

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

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

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 initTag<T: Element>(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.keySet()) {
        builder.append(" $a=\"${attributes[a]}\"")
    }
    return builder.toString()
}
}

abstract class TagWithText(name: String): Tag(name) {
    fun String.plus() {
        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Java Map, 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(*array(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

Java交互

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。

在Kotlin中调用Java代码

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
}
```

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。

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.get(0) // platform type inferred (ordinary Java object)

```

When we call methods on variables of platform types, Kotlin does not issue nullability errors at compile time, but the call may fail at runtime, because of a null-pointer exception or an assertion that Kotlin generates to prevent nulls from propagating:

```

item.substring(1) // allowed, may throw an exception if item == null

```

Platform types are *non-denotable*, meaning that one can not write them down explicitly in the language. When a platform value is assigned to a Kotlin variable, we can rely on type inference (the variable will have an inferred platform type then, as `item` has in the example above), or we can choose the type that we expect (both nullable and non-null types are allowed):

```

val nullable: String? = item // allowed, always works
val notNull: String = item // allowed, may fail at runtime

```

If we choose a non-null type, the compiler will emit an assertion upon assignment. This prevents Kotlin's non-null variables from holding nulls. Assertions are also emitted when we pass platform values to Kotlin functions expecting non-null values etc. Overall, the compiler does its best to prevent nulls from propagating far through the program (although sometimes this is impossible to eliminate entirely, because of generics).

Notation for Platform Types

As mentioned above, platform types cannot be mentioned explicitly in the program, so there's no syntax for them in the language. Nevertheless, the compiler and IDE need to display them sometimes (in error messages, parameter info etc), so we have a mnemonic notation for them:

- `T!` means "`T` or `T?`",
- `(Mutable)Collection<T>!` means "Java collection of `T` may be mutable or not, may be nullable or not",
- `Array<(out) T>!` means "Java array of `T` (or a subtype of `T`), nullable or not"

Mapped types

Kotlin treats some Java types specially. Such types are not loaded from Java "as is", but are *mapped* to corresponding Kotlin types. The mapping only matters at compile time, the runtime representation remains unchanged. Java's primitive types are mapped to corresponding Kotlin types (keeping [platform types](#) in mind):

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

Some non-primitive built-in classes are also mapped:

| Java type | Kotlin type |
|-------------------------------------|-----------------------------------|
| <code>java.lang.Object</code> | <code>kotlin.Any!</code> |
| <code>java.lang.Cloneable</code> | <code>kotlin.Cloneable!</code> |
| <code>java.lang.Comparable</code> | <code>kotlin.Comparable!</code> |
| <code>java.lang.Enum</code> | <code>kotlin.Enum!</code> |
| <code>java.lang.Annotation</code> | <code>kotlin.Annotation!</code> |
| <code>java.lang.Deprecated</code> | <code>kotlin.deprecated!</code> |
| <code>java.lang.Void</code> | <code>kotlin.Nothing!</code> |
| <code>java.lang.CharSequence</code> | <code>kotlin.CharSequence!</code> |
| <code>java.lang.String</code> | <code>kotlin.String!</code> |
| <code>java.lang.Number</code> | <code>kotlin.Number!</code> |
| <code>java.lang.Throwable</code> | <code>kotlin.Throwable!</code> |

Collection types may be read-only or mutable in Kotlin, so Java's collections are mapped as follows (all Kotlin types in this table reside in the package `kotlin`):

| Java type | Kotlin read-only type | Kotlin mutable type | Loaded platform type |
|------------------------------------|------------------------------------|--|--|
| <code>Iterator<T></code> | <code>Iterator<T></code> | <code>MutableIterator<T></code> | <code>(Mutable)Iterator<T>!</code> |
| <code>Iterable<T></code> | <code>Iterable<T></code> | <code>MutableIterable<T></code> | <code>(Mutable)Iterable<T>!</code> |
| <code>Collection<T></code> | <code>Collection<T></code> | <code>MutableCollection<T></code> | <code>(Mutable)Collection<T>!</code> |
| <code>Set<T></code> | <code>Set<T></code> | <code>MutableSet<T></code> | <code>(Mutable)Set<T>!</code> |
| <code>List<T></code> | <code>List<T></code> | <code>MutableList<T></code> | <code>(Mutable)List<T>!</code> |
| <code>ListIterator<T></code> | <code>ListIterator<T></code> | <code>MutableListIterator<T></code> | <code>(Mutable)ListIterator<T>!</code> |
| <code>Map<K, V></code> | <code>Map<K, V></code> | <code>MutableMap<K, V></code> | <code>(Mutable)Map<K, V>!</code> |
| <code>Map.Entry<K, V></code> | <code>Map.Entry<K, V></code> | <code>MutableMap.MutableEntry<K, V></code> | <code>(Mutable)Map. (Mutable)Entry<K, V>!</code> |

Java's arrays are mapped as mentioned [below](#):

| Java type | Kotlin type |
|-----------------------|--|
| <code>int[]</code> | <code>kotlin.IntArray!</code> |
| <code>String[]</code> | <code>kotlin.Array<(out) String>!</code> |

Java generics in Kotlin

Kotlin's generics are a little different from Java's (see [Generics](#)). When importing Java types to Kotlin we perform some conversions:

- Java's wildcards are converted into type projections
 - `Foo<? extends Bar>` becomes `Foo<out Bar!>!`
 - `Foo<? super Bar>` becomes `Foo<in Bar!>!`
- Java's raw types are converted into star projections
 - `List` becomes `List<*>!`, i.e. `List<out Any?>!`

Like Java's, Kotlin's generics are not retained at runtime, i.e. objects do not carry information about actual type arguments passed to their constructors, i.e. `ArrayList<Integer>()` is indistinguishable from `ArrayList<Character>()`. This makes it impossible to perform `is`-checks that take generics into account. Kotlin only allows `is`-checks for star-projected generic types:

```
if (a is List<Int>) // Error: cannot check if it is really a List of Ints
// but
if (a is List<*>) // OK: no guarantees about the contents of the list
```

Java Arrays

Arrays in Kotlin are invariant, unlike Java. This means that Kotlin does not let us assign an `Array<String>` to an `Array<Any>`, which prevents a possible runtime failure. Passing an array of a subclass as an array of superclass to a Kotlin method is also prohibited, but for Java methods this is allowed (though [platform types](#) of the form `Array<(out) String>!`).

Arrays are used with primitive datatypes on the Java platform to avoid the cost of boxing/unboxing operations. As Kotlin hides those implementation details, a workaround is required to interface with Java code. There are specialized classes for every type of primitive array (`IntArray`, `DoubleArray`, `CharArray`, and so on) to handle this case. They are not related to the `Array` class and are compiled down to Java's primitive arrays for maximum performance.

Suppose there is a Java method that accepts an int array of indices:

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

To pass an array of primitive values you can do the following in Kotlin:

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

Java classes sometimes use a method declaration for the indices with a variable number of arguments (varargs).

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

In that case you need to use the spread operator `*` to pass the `IntArray`:

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

It's currently not possible to pass `null` to a method that is declared as varargs.

When compiling to JVM byte codes, the compiler optimizes access to arrays so that there's no overhead introduced:

```
val array = array(1, 2, 3, 4)
array[x] = array[x] * 2 // no actual calls to get() and set() generated
for (x in array) // no iterator created
    print(x)
```

Even when we navigate with an index, it does not introduce any overhead

```
for (i in array.indices) // no iterator created
    array[i] += 2
```

Finally, `in`-checks have no overhead either

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

Checked Exceptions

In Kotlin, all exceptions are unchecked, meaning that the compiler does not force you to catch any of them. So, when you call a Java method that declares a checked exception, Kotlin does not force you to do anything:

```
fun render(list: List<*>, to: Appendable) {
    for (item in list)
        to.append(item.toString()) // Java would require us to catch IOException here
}
```

Object Methods

When Java types are imported into Kotlin, all the references of the type `java.lang.Object` are turned into `Any`. Since `Any` is not platform-specific, it only declares `toString()`, `hashCode()` and `equals()` as its members, so to make other members of `java.lang.Object` available, Kotlin uses [extension functions](#).

`wait()/notify()`

[Effective Java](#) Item 69 kindly suggests to prefer concurrency utilities to `wait()` and `notify()`. Thus, these methods are not available on references of type `Any`. If you really need to call them, you can cast to `java.lang.Object`:

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

`getClass()`

To retrieve the type information from an object, we use the `javaClass` extension property.

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

Instead of Java's `Foo.class` use `javaClass()`.

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

clone()

To override `clone()`, your class needs to extend `kotlin.Cloneable`:

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

Do not forget about [Effective Java](#), Item 11: *Override clone judiciously*.

finalize()

To override `finalize()`, all you need to do is simply declare it, without using the `override` keyword:

```
class C {  
    protected fun finalize() {  
        // finalization logic  
    }  
}
```

According to Java's rules, `finalize()` must not be `private`.

Inheritance from Java classes

At most one Java-class (and as many Java interfaces as you like) can be a supertype for a class in Kotlin. This class must go first in the supertype list.

Accessing static members

Static members of Java classes form “companion objects” for these classes. We cannot pass such a “companion object” around as a value, but can access the members explicitly, for example

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

Java Reflection

Java reflection works on Kotlin classes and vice versa. As mentioned above, you can use `instance.javaClass` or `javaClass<ClassName>()` to enter Java reflection through `java.lang.Class`. You can then “convert” to Kotlin reflection by calling `.kotlin`:

```
val kClass = x.javaClass.kotlin
```

In much the same way you can convert from Kotlin reflection to Java: `ClassName::class.java` is the same as `javaClass<ClassName>()`. Other supported cases include acquiring a Java getter/setter method or a backing field for a Kotlin property, getting a containing `KPackage` instance for a Java class, and getting a `KProperty` for a Java field.

SAM Conversions

Just like Java 8, Kotlin supports SAM conversions. This means that Kotlin function literals can be automatically converted into implementations of Java interfaces with a single non-default method, as long as the parameter types of the interface method match the parameter types of the Kotlin function.

You can use this for creating instances of SAM interfaces:

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

...and in method calls:

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

If the Java class has multiple methods taking functional interfaces, you can choose the one you need to call by using an adapter function that converts a lambda to a specific SAM type. Those adapter functions are also generated by the compiler when needed.

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

Note that SAM conversions only work for interfaces, not for abstract classes, even if those also have just a single abstract method.

Also note that this feature works only for Java interop; since Kotlin has proper function types, automatic conversion of functions into implementations of Kotlin interfaces is unnecessary and therefore unsupported.

Calling Kotlin code from Java

Kotlin code can be called from Java easily.

Package-Level Functions

All the functions and properties declared inside a package `org.foo.bar` are put into a Java class named `org.foo.bar.BarPackage`.

```
package demo  
  
class Foo  
  
fun bar() {  
}
```

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

For the root package (the one that's called a "default package" in Java), a class named `_DefaultPackage` is created.

Static Methods and Fields

As mentioned above, Kotlin generates static methods for package-level functions. On top of that, it also generates static methods for functions defined in named objects or companion objects of classes and annotated as `@platformStatic`. For example:

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

Now, `foo()` is static in Java, while `bar()` is not:

```
C.foo(); // works fine
C.bar(); // error: not a static method
```

Same for named objects:

```
object Obj {
    platformStatic fun foo() {}
    fun bar() {}
}
```

In Java:

```
Obj.foo(); // works fine
Obj.bar(); // error
Obj.INSTANCE.bar(); // works, a call through the singleton instance
Obj.INSTANCE.foo(); // works too
```

Also, public properties defined in objects and companion objects, as well as top-level properties annotated with `const`, are turned into static fields in Java:

```
// file example.kt

object Obj {
    val CONST = 1
}

const val MAX = 239
```

In Java:

```
int c = Obj.CONST;
int d = ExampleKt.MAX;
```

Handling signature clashes with `@platformName`

Sometimes we have a named function in Kotlin, for which we need a different JVM name the byte code. The most prominent example happens due to *type erasure*:

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


These two functions can not be defined side-by-side, because their JVM signatures are the same:

`filterValid(Ljava/util/List;)Ljava/util/List;`. If we really want them to have the same name in Kotlin, we can annotate one (or both) of them with `@platformName` and specify a different name as an argument:

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

From Kotlin they will be accessible by the same name `filterValid`, but from Java it will be `filterValid` and `filterValidInt`.

The same trick applies when we need to have a property `x` alongside with a function `getX()`:

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

fun getX() = 10
```

Overloads Generation

Normally, if you write a Kotlin method with default parameter values, it will be visible in Java only as a full signature, with all parameters present. If you wish to expose multiple overloads to Java callers, you can use the `@jvmOverloads` annotation.

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

For every parameter with a default value, this will generate one additional overload, which has this parameter and all parameters to the right of it in the parameter list removed. In this example, the following methods will be generated:

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

The annotation also works for constructors, static methods etc. It can't be used on abstract methods, including methods defined in interfaces.

Note that, as described in [Secondary Constructors](#), if a class has default values for all constructor parameters, a public no-argument constructor will be generated for it. This works even if the `@jvmOverloads` annotation is not specified.

Checked Exceptions

As we mentioned above, Kotlin does not have checked exceptions. So, normally, the Java signatures of Kotlin functions do not declare exceptions thrown. Thus if we have a function in Kotlin like this:

```
package demo

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

And we want to call it from Java and catch the exception:

```
// Java
try {
    demo.DemoPackage.foo();
}
catch (IOException e) { // error: foo() does not declare IOException in the throws list
    // ...
}
```

we get an error message from the Java compiler, because `foo()` does not declare `IOException`. To work around this problem, use the `@throws` annotation in Kotlin:

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

Null-safety

When calling Kotlin functions from Java, nobody prevents us from passing `null` as a non-null parameter. That's why Kotlin generates runtime checks for all public functions that expect non-nulls. This way we get a `NullPointerException` in the Java code immediately.

Properties

Property getters are turned into *get*-methods, and setters – into *set*-methods.

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）。

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标记的一个例子：

```
/**
 * 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，你可以在括号中附上参数的值，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>`

用来标记一个类的特殊值，这个标记可以用来标记主构造函数，

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

用来标记一个方法抛出的异常。自从Kotlin没有异常检查，因此没有方法会抛出异常，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.

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 Maven V3

通过 *kotlin.version* 指定所要使用的 Kotlin 版本，有以下值可供选择：

- X.Y-SNAPSHOT：指定 X.Y 的值使用对应的最新开发版，当前只有一个开发版本号 0.1-SNAPSHOT。这是非稳定的版本的只适用于测试编译器的新特性。使用开发版需要在 pom 文件中[设置相应的 repository](#)。
- X.Y.Z：指定 X.Y.Z 值使用对应的稳定版本，这些版本发布在 Maven Central Repository 中，无需在 pom 文件中进行额外的设置。

以下是稳定版本对应的版本号 X.Y.Z 值：

| Milestone | Version |
|-----------|-----------|
| 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 |

设置开发版的 repository

使用 Kotlin 的开发版需要在 pom 文件中定义以下的 repository：

```

<repositories>
  <repository>
    <id>sonatype.oss.snapshots</id>
    <name>Sonatype OSS Snapshot Repository</name>
    <url>http://oss.sonatype.org/content/repositories/snapshots</url>
    <releases>
      <enabled>false</enabled>
    </releases>
    <snapshots>
      <enabled>true</enabled>
    </snapshots>
  </repository>
</repositories>

<pluginRepositories>
  <pluginRepository>
    <id>sonatype.oss.snapshots</id>
    <name>Sonatype OSS Snapshot Repository</name>
    <url>http://oss.sonatype.org/content/repositories/snapshots</url>
    <releases>
      <enabled>false</enabled>
    </releases>
    <snapshots>
      <enabled>true</enabled>
    </snapshots>
  </pluginRepository>
</pluginRepositories>

```

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>
      <phase>compile</phase>
      <goals> <goal>compile</goal> </goals>
    </execution>

    <execution>
      <id>test-compile</id>
      <phase>test-compile</phase>
      <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。

通过指定 pom 文件中 <phase> 标签的值为 process-sources 可实现以上目的。（如果有更好的方法欢迎提出）

```

<plugin>
  <artifactId>kotlin-maven-plugin</artifactId>
  <groupId>org.jetbrains.kotlin</groupId>
  <version>0.1-SNAPSHOT</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](#).

使用外部的注解

Kotlin 使用外部注解为 Java 库提供精准的类型信息，通过 <configuration> 标签中的 annotationPaths 指定这些注解。

```
<plugin>
  <artifactId>kotlin-maven-plugin</artifactId>
  <groupId>org.jetbrains.kotlin</groupId>
  <version>0.1-SNAPSHOT</version>

  <configuration>
    <annotationPaths>
      <!--指向注解文件的根目录-->
      <annotationPath>${project.basedir}/src/main/resources/</annotationPath>
    </annotationPaths>
  </configuration>

  ...
```

例子

Maven 工程的例子可从 [Github 直接下载](#)

翻译 by [DemoJameson](#)

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>

```

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

kotlinc Attributes

| Name | Description | Required | Default Value |
|-------------------|---|----------|---------------|
| src | Kotlin source file or directory to compile | Yes | |
| output | Destination directory or .jar file name | Yes | |
| classpath | Compilation class path | No | |
| classpathref | Compilation class path reference | No | |
| stdlib | Path to "kotlin-runtime.jar" | No | "" |
| includeRuntimeJar | If output is a .jar file, whether Kotlin runtime library is included in the jar | No | true |

withKotlin attributes

| Name | Description | Required | Default Value |
|---------------------|------------------------------|----------|---------------|
| externalAnnotations | Path to external annotations | No | |

kotlin2js Attributes

| Name | Description | Required |
|--------------|---|----------|
| src | Kotlin source file or directory to compile | Yes |
| 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 |

Using Griffon

Griffon support is [provided externally](#)

使用 Gradle

插件和版本

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

我们希望通过 *kotlin.version*这样的形式来定义Kotlin的版本. 可能使用的形式是:

- X.Y-SNAPSHOT: 对应于每一个X.Y形式的发布版本, 在服务器上都会更新为对应的成功编译的版本. 这种版本是不稳定的, 并且推荐您仅用于测试编译器新功能. 现阶段所有的构建版本都以0.1-SNAPSHOT的形式发布. 通过使用快照方式, 我们需要 [在build.gradle文件中配置一个快照仓库](#).
- X.Y.Z: 对应于发布版本和里程碑版本 X.Y.Z, 是通过手动升级的. 这是稳定的版本. 正式版会在 Maven Central Repository中发布. 并不需要在build.gradle文件中额外配置.

里程碑和版本之间的对应关系如下:

| 里程碑 | 版本 |
|-------|-----------|
| 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 |

应用于JVM

为了在JVM中应用, Kotlin插件需要配置如下

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

在M11版本, 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
- 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

我们需要添加kotlin-gradle-plugin和Kotlin标准库的依赖:

```
buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath 'org.jetbrains.kotlin:kotlin-gradle-plugin:<version>'
    }
}

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

repositories {
    mavenCentral()
}

dependencies {
    compile 'org.jetbrains.kotlin:kotlin-stdlib:<version>'
}
```

使用快照版本

如果想要使用快照版本 (每日构建), 我们需要添加一个快照仓库并且将版本更改为0.1-SNAPSHOT:

```
buildscript {
    repositories {
        mavenCentral()
        maven {
            url 'http://oss.sonatype.org/content/repositories/snapshots'
        }
    }
    dependencies {
        classpath 'org.jetbrains.kotlin:kotlin-gradle-plugin:0.1-SNAPSHOT'
    }
}

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

repositories {
    mavenCentral()
    maven {
        url 'http://oss.sonatype.org/content/repositories/snapshots'
    }
}

dependencies {
    compile 'org.jetbrains.kotlin:kotlin-stdlib:0.1-SNAPSHOT'
}
```

使用外部注释

JDK和Android SDK的外部注释将自动配置. 如果想要为一些库添加更多的注解, 需要在Gradle脚本中添加下面这一行:

```
kotlinOptions.annotations = file('<path to annotations>')
```

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](#)

翻译By [ChiahaoLu](#)

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

Common Questions

What is Kotlin?

Kotlin is a statically typed language that targets the JVM and JavaScript. It is a general-purpose language intended for industry use.

It is developed by a team at JetBrains although it is an OSS language and has external contributors.

Why a new language?

At JetBrains, we've been developing for the Java platform for a long time, and we know how good it is. On the other hand, we know that the Java programming language has certain limitations and problems that are either impossible or very hard to fix due to backward-compatibility issues. We know that Java is going to stand long, but we believe that the community can benefit from a new statically typed JVM-targeted language free of the legacy trouble and having the features so desperately wanted by the developers.

The main design goals behind this project are

- To create a Java-compatible language,
- That compiles at least as fast as Java,
- Make it safer than Java, i.e. statically check for common pitfalls such as null pointer dereference,
- Make it more concise than Java by supporting variable type inference, higher-order functions (closures), extension functions, mixins and first-class delegation, etc;
- And, keeping the useful level of expressiveness (see above), make it way simpler than the most mature competitor – Scala.

How is it licensed?

Kotlin is an OSS language and is licensed under the Apache 2 OSS License. The IntelliJ Plug-in is also OSS.

It is hosted on GitHub and we happily accept contributors

Is it Java Compatible?

Yes. The compiler emits Java byte-code. Kotlin can call Java, and Java can call Kotlin. See [Java interoperability](#).

Is there tooling support?

Yes. There is an IntelliJ IDEA plugin that is available as an OSS project under the Apache 2 License. You can use Kotlin both in the [free OSS Community Edition and Ultimate Edition](#) of IntelliJ IDEA.

Is there Eclipse support?

Yes. Please refer to the [tutorial](#) for installation instructions.

Is there a standalone compiler?

Yes. You can download the standalone compiler and other builds tools from the [release page on GitHub](#)

Is Kotlin a Functional Language?

Kotlin is an Object-Orientated language. However it has support for higher-order functions as well as function literals and top-level functions. In addition, there are a good number of common functional language constructs in the standard Kotlin library (such as map, flatMap, reduce, etc.). Also, there's no clear definition on what a Functional Language is so we couldn't say Kotlin is one.

Does Kotlin support generics?

Kotlin supports generics. It also supports declaration-site variance and usage-site variance. Kotlin also does not have wildcard types. Inline functions support reified type parameters.

Are semicolons required?

No. They are optional.

Are curly braces required?

Yes.

Why have type declarations on the right?

We believe it makes the code more readable. Besides, it enables some nice syntactic features, for instance, it is easy to leave type annotations out. Scala has also proven pretty well this is not a problem.

Will right-handed type declarations effect tooling?

No. It won't. We can still implement suggestions for variable names, etc.

Is Kotlin extensible?

We are planning on making it extensible in a few ways: from inline functions to annotations and type loaders.

Can I embed my DSL into the language?

Yes. Kotlin provides a few features that help: Operator overloading, Custom Control Structures via inline functions, Infix function calls, Extension Functions, Annotations and language quotations.

What ECMAScript level does the JavaScript support?

Currently at 5.

Does the JavaScript back-end support module systems?

Yes. There are plans to provide CommonJS and AMD support.

Comparison to Java

Some Java issues addressed in Kotlin

Kotlin fixes a series of issues that Java suffers from

- Null references are [controlled by the type system](#).
- [No raw types](#)
- Arrays in Kotlin are [invariant](#)
- Kotlin has proper [function types](#), as opposed to Java's SAM-conversions
- [Use-site variance](#) without wildcards
- Kotlin does not have checked [exceptions](#)

What Java has that Kotlin does not

- [Checked exceptions](#)
- [Primitive types](#) that are not classes
- [Static members](#)
- [Non-private fields](#)
- [Wildcard-types](#)

What Kotlin has that Java does not

- [Function literals](#) + [Inline functions](#) = performant custom control structures
- [Extension functions](#)
- [Null-safety](#)
- [Smart casts](#)
- [String templates](#)
- [Properties](#)
- [Primary constructors](#)
- [First-class delegation](#)
- [Type inference for variable and property types](#)
- [Singletons](#)
- [Declaration-site variance & Type projections](#)
- [Range expressions](#)
- [Operator overloading](#)
- [Companion objects](#)

Comparison to Scala

The two main design goals for Kotlin are:

- Make compilation at least as fast as Java
- Keep useful level of expressiveness while maintaining the language simple as possible

Taking this into account, if you are happy with Scala, you probably do not need Kotlin

What Scala has that Kotlin does not

- Implicit conversions, parameters, etc
 - In Scala, sometimes it's very hard to tell what's happening in your code without using a debugger, because too many implicits get into the picture
 - To enrich your types with functions in Kotlin use [Extension functions](#).
- Overridable type members
- Path-dependent types
- Macros
- Existential types
 - [Type projections](#) are a very special case
- Complicated logic for initialization of traits
 - See [Classes and Inheritance](#)
- Custom symbolic operations
 - See [Operator overloading](#)
- Built-in XML
 - See [Type-safe Groovy-style builders](#)

Things that may be added to Kotlin later:

- Structural types
- Value types
- Yield operator
- Actors
- Parallel collections

What Kotlin has that Scala does not

- [Zero-overhead null-safety](#)
 - Scala has Option, which is a syntactic and run-time wrapper
- [Smart casts](#)
- [Kotlin's Inline functions facilitate Nonlocal jumps](#)
- [First-class delegation](#). Also implemented via 3rd party plugin: Autoproxy

