Kotlin入门和使用（讲稿）

zhangxiaoke@douban.com 2016.03.24

Intro

Java 有哪些问题?

- 空引用（Null references）：连空引用的发明者都成这是个 billion-dollar 错误（参见)。不论你费多大的功夫，你都无法避免它。因为 Java 的类型系统就是不安全的。
- 原始类型（Raw types）：我们在开发的时候总是会为了保持兼容性而卡在范型原始类型的问题上，我们都知道要努力避免 raw type 的警告，但是它们毕竟是在语言层面上的存在，这必定会造成误解和不安全因素。
- 协变数组（Covariant arrays）：你可以创建一个 string 类型的数组和一个 object 型的数组，然后把 string 数组分配给 object 数组。这样的代码可以通过编译，但是一旦你尝试在运行时分配一个数给那个数组的时候，他就会在运行时抛出异常。
- Java 8 存在高阶方法（ higher-order functions ），但是他们是通过 SAM 类型 实现的。SAM 是一个单个抽象方法，每个函数类型都需要一个对应的接口。如果你想要创建一个并不存在的 lambda 的时候或者不存着对应的函数类型的时候，你要自己去创建函数类型作为接口。
- 泛型中的通配符：诡异的泛型总是难以操作，难以阅读，书写，以及理解。对编译器而言，异常检查也变得很困难。
- 不够灵活，缺乏扩展能力：我们不能给不是我们自己写的 types、classes 或者 interfaces 增加新的方法。长时间以来，我们都会采用 util 类，杂乱无章地堆砌着我们代码或者或者揉在同一个 util package 里面。如果这是解决方案的话，它肯定不理想。
- 语法繁琐，不够简洁：Java 肯定不是最简洁的语言。这件事本身不是件坏事，但是事实上存在太多的常见的冗余。这会带来潜在的错误和缺陷。在这之前，我们还要处理安卓 API 带来的问题。

Features

- Lambdas
- Data classes
- Function literals

- Extension functions
- Null safety
- Smart casts
- String templates
- Properties
- Class delegation
- Type inference
- Range expressions

Kotlin 有几个核心的目标：

- 简约：帮你减少实现同一个功能的代码量。
- 易懂：让你的代码更容易阅读，同时易于理解。
- 安全：移除了你可能会犯错误的功能。
- 通用：基于 JVM 和 Javascript，你可以在很多地方运行。
- 互操作性：这就意味着 Kotlin 和 Java 可以相互调用，目标是 100% 兼容。

Kotlin的特性

刚才我们提到过的这些缺陷，Kotlin 通常直接移除了那些特性。同时它也加了一些新的特性：

- Lambda 表达式
- 数据类 (Data classes)
- 函数字面量和内联函数（Function literals & inline functions）
- 函数扩展 (Extension functions)
- 空安全（Null safety）
- 智能转换（Smart casts）
- 字符串模板（String templates）
- 主构造函数（Primary constructors）
- 类委托（Class delegation）
- 类型推断（Type inference）
- 单例（Singletons）
- 声明点变量（Declaration-site variance）
- 区间表达式（Range expressions）

我们将在这篇文章里提及以上大多数特性。Kotlin 之所以能跟随者 JVM 的生态系统不断地进步，是因为他没有任何限制。它编译出来的正是 JVM 字节码。在 JVM 看来，它就跟其他语言一样样的。事实上，如果你在 IntelliJ 或者 Android Studio 上用 Kotlin 的插件，它自带里一个字节码查看器，可以显示每个方法生成的 JVM 字节码。

Syntax

Variables

```kotlin
// hello.kt
package com.kotlin.demo

val a: Int = 1
val b = 1
val c: Int
c = 1
// c = 2
var x = 5
x += 1

// val text: String = "Hello, World"
val text = "Hello, World"

// val ints: Array<Int> = arrayOf<Int>(1,2,3,4)
val ints = arrayOf(1,2,3,4)

var a: String = "abc"
a = null // compilation error

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

类型声明

- 包的声明应处于源文件顶部，目录与包的结构无需匹配，源代码可以在文件系统的任意位置。
- 在 Kotlin 里，得把参数名放在前面，参数类型放在后面，用一个冒号隔开。
- 常量（使用 val 关键字声明），相当于Java里的final，如果没有初始值，声明常量时，常量的类型不能省略。
- 变量（使用 var 关键字声明），类型可以省略，自动推断出 Int 类型
- 正如 Java 和 JavaScript，Kotlin 支持行注释及块注释，与 Java 不同的是，Kotlin 的块注释可以嵌套。
- 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。

类型推导

你可能在其他语言中看到过类型推导。在 Java 里，我们需要自己声明类型，变量名，以及数值。

在 Kotlin 里，顺序有些不一样，你先声明变量名，然后是类型，然后是分配值。很多情况下，你不需要声明类型。一个字符串字面量足以指明这是个字符串类型。字符，整形，长整形，单浮点数，双浮点数，布尔值都是可以无需显性声明类型的。

只要 Kotlin 可以推导，这个规则同样适用与其他一些类型。通常，如果是局部变量，当你在声明一个值或者变量的时候你不需要指明类型。在一些无法推导的场景里，你才需要用完整的声明变量语法指明变量类型。

Define Function

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

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

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

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

fun main(args: Array<String>) {
    println("Hello, World!")
}
```

定义函数

- 声明函数的关键字是 fun，fun 后面跟的是函数的名称，然后括号包裹起来的是函数参数，这个跟 Java 类似。
- 这是带有两个 Int 参数、返回 Int 的函数。可以将表达式作为函数体、返回值类型自动推断的函数。函数返回无意义的值，Unit 返回类型可以省略。Kotlin的函数和Java类似，但不需要定义在类内部。
- 函数的返回类型在最后，这个跟 Java 放在前面形式不太一样。如果一个函数没有返回任何类型，可以返回一个 Unit 类型，当然也可以省略。调用 Kotlin 标准库中的函数 println 就能打印 Hello World 出来，实际上它最终调用了 Java 的 system.out.println。

If Expressions

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

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

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

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

If表达式

- 在 Kotlin 中，if是一个表达式，即它会返回一个值。 因此就不需要三元运算符（条件 ? 然后 : 否则），因为普通的 if 就能胜任这个角色。
- if的分支可以是代码块，最后的表达式作为该块的值。如果你使用 if 作为表达式而不是语句（例如：返回它的值或者 把它赋给变量），该表达式需要有 else 分支。

Loop

```
fun forLoop1(args: Array<String>) {
    for (arg in args) {
        print(arg)
    }
}

fun forLoop2(args: Array<String>) {
    for (i in args.indices) {
        print(args[i])
    }
}

fun whileLoop1(args: Array<String>) {
    var i = 0
    while (i < args.size)
        print(args[i++])
}
```

FOR循环

- for 循环可以对任何提供迭代器（iterator）的对象进行遍历，循环体可以是一个代码块。
- for 可以循环遍历任何提供了迭代器的对象。即：
    - 有一个成员函数或者扩展函数 iterator()，它的返回类型
    - 有一个成员函数或者扩展函数 next()，并且
    - 有一个成员函数或者扩展函数 hasNext() 返回 Boolean
- 对数组的 for 循环会被编译为并不创建迭代器的基于索引的循环，注意这种“在区间上遍历”会编译成优化的实现而不会创建额外对象，如果你想要通过索引遍历一个数组或者一个 list，你可以用.indices或array.withIndex。
- while 和 do..while 照常使用。

When

```
when (x) {
  0, 1 -> print("x == 0 or x == 1")
  3 -> print("x == 3")
  4 -> print("x == 4")
  else -> print("otherwise")
}

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")
}
```

When语句和表达式

- when 取代了类 C 语言的 switch 操作符
- when 将它的参数和所有的分支条件顺序比较，直到某个分支满足条件
- when 既可以被当做表达式使用也可以被当做语句使用
  - 如果它被当做表达式， 符合条件的分支的值就是整个表达式的值
  - 如果当做语句使用， 则忽略个别分支的值

- 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
- 我们可以用任意表达式（而不只是常量）作为分支条件
- 我们也可以检测一个值在（in）或者不在（!in）一个区间或者集合中
- 另一种可能性是检测一个值是（is）或者不是（!is）一个特定类型的值
- when 也可以用来取代 if-else if链
- 如果不提供参数，所有的分支条件都是简单的布尔表达式，而当一个分支的条件为真时则执行该分支

Returns

```
return.
break.
continue.

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

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

返回和跳转

- Kotlin 有三种结构化跳转操作符
    - return.默认从最直接包围它的函数或者匿名函数返回。
    - break.终止最直接包围它的循环。
    - continue.继续下一次最直接包围它的循环。
- 在 Kotlin 中任何表达式都可以用标签（label）来标记。
- 我们可以用标签限制 break 或者continue，标签限制的 break 跳转到刚好位于该标签指定的循环后面的执行点。 continue 继续标签指定的循环的下一次迭代。
- Kotlin 有函数字面量、局部函数和对象表达式。因此 Kotlin 的函数可以被嵌套。 标签限制的 return 允许我们从外层函数返回。 最重要的一个用途就是从 lambda 表达式中返回。
- 这个 return 表达式从最直接包围它的函数即 foo 中返回。 （注意，这种非局部的返回只支持传给内联函数的 lambda 表达式。） 如果我们需要从 lambda 表达式中返回，我们必须给它加标签并用以限制 return。

Strings

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

val s = "Hello, world!\n"

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

val s = "abc"
val str = "$s.length is ${s.length}"

var args = arrayOf("Cat", "Dog", "Rabbit")
print("Hello ${args[0]}")
```

字符串说明

- 字符串用 String 类型表示。字符串是不可变的。 字符串的元素——字符可以使用索引运算符访问: s[i]。可以用 for 循环迭代字符串。
- Kotlin 有两种类型的字符串字面值: 转义字符串可以有转义字符，以及原生字符串白可以包含换行和任意文本。转义字符串很像 Java 字符串。
- 原生字符串 使用三个引号（"""）分界符括起来，内部没有转义并且可以包含换行和任何其他字符。
- 字符串可以包含模板表达式 ，即一些小段代码，会求值并把结果合并到字符串中。 模板表达式以美元符（$）开头，由一个简单的名字构成，或者用花括号扩起来的任意表达式。
- 原生字符串和转义字符串内部都支持模板。 如果你需要在原生字符串中表示字面值 $ 字符（它不支持反斜杠转义），你可以用三引号语法。
- 字符串字面值用单引号括起来: '1'。 特殊字符可以用反斜杠转义。 支持这几个转义序列：\t、 \b、\n、\r、\'、"、\ 和 $。 编码其他字符要用 Unicode 转义序列语法：'\uFF00'。

Basic Types

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

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

数据类型

- 在 Kotlin 中，所有东西都是对象，在这个意义上讲所以我们可以在任何变量上调用成员函数和属性。有些类型是内置的，因为他们的实现是优化过的。但是用户看起来他们就像普通的类。本节我们会描述大多数这些类型：数字、字符、布尔和数组。
- Kotlin 处理数字在某种程度上接近 Java，但是并不完全相同。例如，对于数字没有隐式拓宽转换（如 Java 中 int 可以隐式转换为long——译者注），另外有些情况的字面值略有不同。
- 注意在 Kotlin 中字符不是数字。
- 在 Java 平台数字是物理存储为 JVM 的原生类型，除非我们需要一个可空的引用（如 Int?）或泛型。 后者情况下会把数字装箱。
- 由于不同的表示方式，较小类型并不是较大类型的子类型。 如果它们是的话，就会出现下述问题。

```
// 假想的代码，实际上并不能编译：
val a: Int? = 1 // 一个装箱的 Int (java.lang.Integer)
val b: Long? = a // 隐式转换产生一个装箱的 Long (java.lang.Long)
print(a == b) // 惊！这将打印 "false" 鉴于 Long 的 equals() 检测其他部分也是 Long
```

- 因此较小的类型不能隐式转换为较大的类型。 这意味着在不进行显式转换的情况下我们不能把 Byte 型值赋给一个 Int 变量。
- 缺乏隐式类型转换并不显著，因为类型会从上下文推断出来，而算术运算会有重载做适当转换。例如

```
val l = 1L + 3 // Long + Int => Long
```

编码风格

- 使用驼峰不要使用下划线
- 类型名是用大写字母开头
- 方法和属性名小写开头
- 使用四个空格的缩进
- 公开方法应该有文档
- 冒号前后有空格，大括号两边有空格，箭头两边有空格，举例

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

class: center, middle

Classes and Objects

Classes

```
class Invoice {
}

class Empty

class Person(name: String) {
}

class Person(name: String) {
    val customName = name.toUpperCase()
}

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

定义一个类

- 类的定义要通过 class 关键字，跟 Java 里的一样，关键字后是类名。Kotlin 有一个主构造函数，我们可以直接将构造函数参数列表写在类的声明处，还可以直接用 var 或者 val 关键字将参数声明为成员变量（又称：类属性）

- 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。
- 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）。
- 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。
- 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。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) {
  // ...
}
```

Constructors

```
class Person constructor(name: String) {
    val fixName = name.toUpperCase()
}

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

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

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

构造函数

Kotlin 中，类可以拥有多个构造函数，这一点跟 Java 类似。但你也可以有一个主构造函数。下面的例子是我们从上面的例子里衍生出来的，在函数头里添加了一个主构造函数

当然，更好的方法是：直接在主构造函数里定义这些属性，定义的方法是在参数名前加上 var 或者 val 关键字，val 是代表属性是常量。

在主构造函数里，可以直接用这些参数变量赋值给类的属性，或者用构造代码块来实现初始化。

- 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为关键字作为前缀
- 与普通属性一样,主构造函数中声明的属性可以是 可变的（var）或者是只读的（val）
- 如果构造函数有注解或可见性修饰符，这个constructor关键字是必需的
- 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
- 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
- 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

Inheritance

```
class Example // Implicitly inherits from Any

open class Base(p: Int)

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

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

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

类的继承

- 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
- 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。
- 要声明一个明确的父类，我们把类型放到类头冒号之后，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函

数。

Overriding Members

```
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()
  }
}
```

成员覆盖

- 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。）
- 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
- 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类型，我们使用super{：.keyword}在尖括号，如规范的父名super
- 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。
- 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。

Properties

```
public class Address {
  public val code:Int = 10015
  public var name: String = ...
  public var city: String = ...
  public var state: String? = ...
  public var zip: String = ...
}

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
}

const val DEPRECATED: String = "deprecated"
const val SOCKET_TIMEOUT = 30*1000L
```

类的属性

- Kotlin的类可以有属性. 这些声明是可变的,用关键字var或者使用只读关键字val
- 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
- 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化器时不小心改变了公有API。

常量属性的要求

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

Getters and Setters

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

var stringRepresentation: String
  get() = this.toString()
  set(value) {
    setDataFromString(value)
  }

var setterVisibility: String = "abc"
// Initializer required, not a nullable type
  private set // the setter is private
```

Getter和Setter

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

Interface

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

    val propertyWithImplementation: String
        get() = "foo"

    fun bar()
    fun foo() {
      // optional body
    }
}

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

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

接口定义

- 使用关键字 interface 来定义接口。Kotlin 的接口与 Java 8 类似，既包含抽象方法的声明，也包含 实现。与抽象类不同的是，接口无法保存状态。它可以有 属性但必须声明为 abstract或提供访问器实现。
- 一个类或者对象可以实现一个或多个接口。
- 实现多个接口时，可能会遇到接口方法名同名的问题。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()。

Data Class

```kotlin
// equals()/hashCode()/toString()/componentN()/copy()

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

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

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

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

数据类/POJO

- 我们经常创建一些只是处理数据的类。在这些类里的标准功能经常是 衍生自数据。在Kotlin中，这叫做数据类 并标记为data。
- 编译器自动从在主构造函数定义的全部特性中得到以下成员：
    - equals()/hashCode() 对，
    - toString() 格式是 "User(name=John, age=42)"，
    - componentN() functions 对应按声明顺序出现的所有属性，
    - copy() 函数（见下面）。
- 如果有某个函数被明确地定义在类里或者被继承，编译器就不会生成这个函数。
- 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。
- 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。
- 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。

Destructuring Declarations

```kotlin
data class Person(val name:String, val age:Int){}

val person = Person("John",20)
val (name, age) = person

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

for ((a, b) in collection) { ... }
for ((key, value) in map) {
}

val (result, status) = function(...)

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

解构声明

- 有时把一个对象*解构*成很多变量很比较方便，这种语法叫做*解构声明*。一个解构声明同时创造多个变量。 我们申明了两个新变量：name 和 age,并且可以独立使用他们。
- component1() 和 component2() 函数是 principle of conventions widely 在Kotlin 中的另一个例子。 (参考运算符如 + ， *, for-loops 等) 任何可以被放在解构声明右边的和组件函数的需求数字都可以调用它。当然，这里可以有更多的如 component3() 和 component4()。
- 变量 a 和 b 从调用从 component1() 和 component2() 返回的集合collection中的对象。
- 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。
- 可能最好的遍历一个映射的方式就是这样，实现这个接口，于是你可以自由的使用解构声明 for-loops 来操作映射(也可以用在数据类实例的集合等)。

Nested and Inner Classes

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

val demo = Outer.Nested().foo() // == 2
```

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

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

嵌套类和内部类

- 在类的内部可以嵌套其他的类，相当于Java里的static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

Enum

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

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

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

  TALKING {
    override fun signal() = WAITING
  };

  abstract fun signal(): ProtocolState
}
```

枚举

- 枚举类的最基本应用是实现类型安全的多项目集合。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。
- 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。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。
- 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)。

- 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。

Object

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

// Singleton
object Resource {
    val name = "Name"
}

open class A(x: Int) {
  public open val y: Int = x
}

interface B {...}

val ab = object : A(1), B {
  override val y = 15
}
```

对象和单例

- 有些时候我们需要创建一个对某些类做了轻微改变的一个对象，而不用为了它显式地定义一个新的子类。 Java把这种情况处理为匿名内部类。 在Kotlin稍微推广了这个概念，称它们为对象表达式和对象声明。
- 对象表达式：如果父类型有一个构造函数，合适的构造函数参数必须传递给它。 多个父类型用逗号隔开，跟在冒号后面。
- 或许，我们需要的仅是无父类的一个对象，那么我们可以简单地写为adHoc这种。
- 就像Java的匿名内部类，在对象表达式里代码可以访问封闭的作用域 （但与Java不同的是，它能访问非final修饰的变量）。
- 对象声明：单例模式是一种非常有用的模式，而在Kotilin（在Scala之后）中很容易就能声明一个单例。DataProviderManager被称为对象声明。如果有一个object关键字在名字前面，这不能再被称为一个*表达式*。 我们不能把这样的东西赋值给变量，但我们可以通过它的名字来引用它。

对象表达式与对象声明语义上的不同

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

Companion

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

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

val x = MyClass.Companion
```

伴生对象

Kotlin 移除了 static 的概念。通常用 companion object 来实现类似功能。

- 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
- 伴生对象的成员可以使用类名称作为限定符来调用
- 使用companion关键字时候，伴生对象的名称可以省略
- 注意，虽然伴生对象的成员在其他语言中看起来像静态成员，但在运行时它们 仍然是实体的实例成员，举例来说，我们能用它实现接口
- 然而，在JVM中，如果你使用@JvmStatic注解，你可以让伴生对象的成员生成为实际存在的静态方法和域

Class Delegation

```kotlin
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
}
```

类的委托

委托是一个大家都知道的设计模式，Kotlin 把委托视为很重要的语言特性。

- 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。
- 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，并且编译器会生成所有的用于转发给b的Base的方法。

Delegated Properties

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

class Delegate {
  operator fun getValue(thisRef: Any?,
        property: KProperty<*>): String {
    return "$thisRef, thank you for delegating
        '${property.name}' to me!"
  }

  operator fun setValue(thisRef: Any?,
        property: KProperty<*>, value: String) {
    println("$value has been assigned to
        '${property.name} in $thisRef.'")
  }
}

val e = Example()
println(e.p)
// Example@33a17727, thank you for delegating ‘p’ to me!

e.p = "NEW"
// NEW has been assigned to ‘p’ in Example@33a17727.
```

委托属性

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

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

当我们读取一个Delegate的委托实例 p , Delegate中的getValue()就被调用, 所以它第一变量就是我们从 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)。

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

属性委托要求

这里我们总结委托对象的要求。

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

- 接收者 — 必须与*属性所有者*类型相同或者是其父类(对于扩展属性，类型范围允许扩大),
- 包含数据 — 一定要是 KProperty<*> 的类型或它的父类型,
- 这个函数必须返回同样的类型作为属性（或者子类型）

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

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

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

标准委托

标准库中对于一些有用的委托提供了工厂（factory）方法。

By Lazy

```
val lazyValue: String by lazy {
    Log.v("Lazy", "Lazy Init")
    "Hello, Lazy!"
}

fun lazyTest() {
    Log.d("Lazy", lazyValue)
    Log.d("Lazy", lazyValue)
}
```

```
V/Lazy: Lazy Init
D/Lazy: Hello, Lazy!
D/Lazy: Hello, Lazy!
```

延迟属性 Lazy

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

默认地，对于lazy属性的计算是同步锁（synchronized） 的: 这个值只在一个线程被计算，并且所有的线程会

看到相同的值。如果初始化代理的同步锁不是必须的，以至于多个线程可以同步地执行，那么将LazyThreadSafetyMode.PUBLICATION作为一个变量传递给lazy()函数。

而且如果你确定初始化将总是发生在单个线程，那么你可以使用 LazyThreadSafetyMode.NONE 模式, 它不会有任何线程安全的保证和相关的开销。

Observable

```
import kotlin.properties.Delegates

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

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

// output
// <no name> -> first
// first -> second
```

可观察属性 Observable

Delegates.observable() 需要两个参数：初始值和handler。 这个 handler 会在每次我们给赋值的时候被调用(在工作完成前). 它有三个参数:一个被赋值的属性，旧的值和新的值

这个例子输出：

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

By Map

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

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

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

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

// custom impl by Json
```

把属性储存在map中

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

在这个例子中，构造函数会接收一个map参数，委托会从这个map中取值 (通过string类型的key，就是属性的名字)，对于 var的变量，我们可以把只读的Map换成 MutableMap就可以了

Functions and Lambdas

Function Declarations

```kotlin
fun double(x: Int): Int {
}
val result = double(2)

infix fun Int.shl(x: Int): Int {
...
}
1 shl 2

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

fun printHello(name: String?): Unit {
    if (name != null)
        println("Hello ${name}")
}

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

函数声明

在Kotlin中，函数声明使用关键字 fun

函数用途

调用函数使用传统的方法，调用成员函数使用点表达式

中缀表示法

函数还可以用中缀表示法，当：1. 他们是成员函数 或者 扩展函数；2. 他们只有一个参数；3. 使用infix关键字声明

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。显式地声明返回类型可选时，这可以由编译器推断。

显式地返回类型

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

器）来说返回类型将不明显。

返回Unit的函数

如果一个函数不返回任何有用的值，它的返回类型是Unit。Unit是一种只有一个值 - Unit`。这个 值不需要显式地返回。Unit返回类型声明也是可选的，可以省略。

Function Arguments

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

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

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

参数

函数参数是使用Pascal表达式，即 name: type。参数用逗号隔开。每个参数必须有显式类型。

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。默认值定义使用后* * = * *类型的值。

命名参数

可以在调用函数时使用命名的函数参数。当一个函数有大量的参数或默认参数时这非常方便。使用命名参数我们可以使代码更具有可读性。可以省略部分参数。

Function Usage

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

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

val list = asList(1, 2, 3)

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

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

Sample().foo()
```

数量可变的参数(可变参数)

函数的（通常最后一个）参数可以使用'vararg`修饰。内部函数vararg类型T是可见的arrayT,即上面的例子中的ts变量是Array类型。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*）

函数作用域(函数范围)

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

局部函数

Kotlin提供局部函数,即一个函数在另一个函数中，局部函数可以访问外部函数的局部变量（即闭包），所以在上面的例子，the visited是局部变量.

成员函数

成员函数是一个函数,定义在一个类或对象里，成员函数调用点符号

Higher-order Functions

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

高阶函数

这是个新奇的术语，它指的是函数可以接收函数，或者函数可以返回函数。

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。

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)。

内联函数

使用内联函数有时能提高高阶函数的性能。

Higher-order Functions

```
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
}

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

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

val names = listOf("Jake", "Jesse", "Matt", "Alec")
val jakes = names.filter { it == "Jake" }
```

高阶函数

另一个高阶函数的例子是 map() (MapReduce)

这些约定可以写成 LINQ-风格 的代码:

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

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

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

Function Types

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

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

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。

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, 也就是函数字面量。

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 被用作为一个函数: 它传入两个T类型的参数.

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

Lambda Expressions

```
1  val sum = { x: Int, y: Int -> x + y }
2
3  val sum: (Int, Int) -> Int = { x, y -> x + y }
4
5  ints.filter { it > 0 }
6
7  fun(x: Int, y: Int): Int = x + y
8
9  fun(x: Int, y: Int): Int {
10   return x + y
11 }
12
13 ints.filter(fun(item) = item > 0)
14
15 // receier
16 val sum = fun Int.(other: Int): Int = this + other
17 sum : Int.(other: Int) -> Int
18 1.sum(2)
```

Lambda表达式

另外，函数表达式也被称作 lambdas 或者 closures。这里有一个最简单的函数表达式：{ it.toString() }。它是一段代码在 “it” 变量上调用了 two-string 函数。“it” 是个 built-in 的名字。当你在写这些函数表达式的时候，如果你只有一个参数传入这段代码，你可以用 “it” 引用，这只是一个你不需要定义参数的方法。

但是当你需要定义参数的时候，或者不止一个参数要定义的时候，语法就是这样的：{ x, y -> x + y }。我们可以创建一段代码，一个函数表达式，输入两个参数，然后把它们相加。如果我们愿意，我们可以显示定义类型。

Lambda表达式语法

Lambda 表达式的全部语法形式, 也就是函数类型的字面量, 譬如上面的sum的声明。

一个 lambda 表达式货匿名函数是一个“函数字面值”, 即 一个未声明的函数，但却立即写为表达式。

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

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

lambda 表达式

这里有更详细的描述, 但是为了继续这一段，让我们看到一个简短的概述：

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

匿名函数

上述 lambda 表达式的语法还少了一个东西： 能够指定函数的返回 类型。在大多数情况下, 这是不必要的。因为返回类型可以被自动推断出来. 然而，如果你需要要明确的指定。你需要一个替代语法。匿名函数看起来很像一个普通函数声明, 只是名字被省略了。

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

闭包

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

带接收者得函数字面值

这样的函数字面量的类型是一个带receiver的函数类型

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

Inline Functions

```
lock(l) { foo() }

// compiled
l.lock()
try {
  foo()
}
finally {
  l.unlock()
}

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

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

内联函数

内联函数和高阶函数经常一起见到。在某些场景下，当你用到泛型的时候，你可以给函数加上 inline 关键字。在编译时，它会用 lambda 表达式替换掉整个函数，整个函数的代码会成为内联代码。

使用高阶函数会带来一些运行时间效率的损失：每一个函数都是一个对象，并且都会捕获一个闭包。 即那些在函数体内会被访问的变量。 内存分配(对于函数对象和类)和虚拟调用会引入运行时间开销。

但是在许多情况下通过内联化 lambda 表达式可以消除这类的开销。 我们通过下面的示例函数来分析上面这些内容。如，lock() 函数可以被很容易地在调用点被内联。

inline修饰符会影响函数体本身以及传递过来的lambdas: 所有的这些会被内联到 调用点。内联本身有时会引起生成的代码数量增加，但是如果我们使用得当(不要内联大的函数)。它将在 性能上有所提升，尤其是在超多态(megamorphic)调用点的循环中。

禁止内联（noinline）

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

Extensions

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

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

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

fun Date.isTuesday(): Boolean {
  return getDay() == 2
}

val tuesday = date.isTuesday();

fun Int.biggerThanTen(): Boolean {
    return this > 10
}
```

声明一个扩展函数，我们需要用一个 接收者类型 也就是被扩展的类型来作为他的前缀。 下面是为MutableList添加一个swap方法，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。

函数扩展

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

函数扩展是 Kotlin 最强大的特性之一。

函数扩展可以是任何整形，字面量或者包装类型，也可以在标记为 final 的类上做类似操作。因为扩展函数不是真的给类增加代码，任何人都没有办法去修改一个类，它实际上是创建了一个静态方法，用语法糖来让扩展函数看着像是类自带的方法一样。

Kotlin 有扩展函数的概念。这不是 Kotlin 语言独有的，但是和其他语言里面我们看到的扩展又不太一样。如果我们在纯 Java 语言的环境下添加一个 date 的方法，我们需要写一个 utils 类或者 dates 类，然后增加一个静态方法。它接收一个实例，然后做些事情，可能会返回一个值。

Kotlin 的一个非常好的功能是，它会自动地转换有 getters 和 setters 综合属性的类型。所以我能够替换 getDay() 为 day，因为这个 day 的属性是存在的。它看起来像一个 field，但是实际上是个 property – getter 和 setter 的概念融合在了一起。

扩展的静态解析

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

我们想强调下扩展方法是被静态分发的，即他们不是接收类型的虚方法。

Nullable接收者

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

扩展属性

和方法相似，Kotlin支持扩展属性。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.

伴生对象的扩展

如果一个类定义有一个伴生对象 ，你也可以为伴生对象定义 扩展函数和属性，就像伴生对象的其他普通成员，只用用类名作为限定符去调用他们。

Stdlib

Kotlin标准库 The Kotlin Standard Library provides living essentials for everyday work with Kotlin. These include:

- 有用的高阶函数 Higher-order functions implementing idiomatic patterns (let, apply, use, synchronized, etc).
- 集合操作的扩展函数 Extension functions providing querying operations for collections (eager) and sequences (lazy).
- 字符串等工具函数 Various utilities for working with strings and char sequences.
- 对JDK文件/线程/IO等类的扩展 Extensions for JDK classes making it convenient to work with files, IO, and threading.

Types

```
Java                            Kotlin

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

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

Functions

```
fun assert(value: Boolean, lazyMessage: () -> Any)
fun check(value: Boolean, lazyMessage: () -> Any)
fun require(value: Boolean, lazyMessage: () -> Any)
fun error(message: Any): Nothing

fun <R> run(block: () -> R): R
fun <R> synchronized(lock: Any, block: () -> R): R
fun <T> lazy(initializer: () -> T): Lazy<T>
fun <T> lazyOf(value: T): Lazy<T>
fun <T, R> T.let(block: (T) -> R): R
fun <T> T.apply(block: T.() -> Unit): T
fun repeat(times: Int, action: (Int) -> Unit)
fun <T, R> with(receiver: T, block: T.() -> R): R
```

Functions

```
inline fun <T> T.apply(block: T.() -> Unit)
    : T { block(); return this }

inline fun <T, R> T.let(block: (T) -> R): R = block(this)

inline fun <T, R> with(receiver: T, block: T.() -> R)
    : R = receiver.block()

var u1: User? = null
val u2 = User("Alice", age = 20, desc = "Wonderful!")
u1?.apply { sayHello() }
u2.apply { eat(); sayHello() }
u1?.let {
    u2.show(it)
    u1.sayHello()
}
u2.let { println("user is valid!") }
with(u2) {
    println("do something on user")
    sayHello()
}
```

T.apply

Calls the specified function block with this value as its receiver and returns this value.

T.let

Calls the specified function block with this value as its argument and returns its result.

with

Calls the specified function block with the given receiver as its receiver and returns its result.

Arrays

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

fun iterator(): Iterator<T>
// ...
}

val asc = Array(5, { i -> (i * i).toString() })

val x: IntArray = arrayOf(1, 2, 3, 4, 5)
x[0] = x[1] + x[2]

val intArray2 = intArrayOf(2, 4, 6, 8, 10)
val boolArray1 = arrayOf(true, false, true, false, true)
val boolArray2 = booleanArrayOf(true, true, false, true)
val strArray1 = arrayOf("Cat", "Dog", "Rabbit")

// byteArrayOf(), charArrayOf(), shortArrayOf(),
// longArrayOf(), floatArrayOf()
```

Array Extensions

```kotlin
// create array
arrayOf()/arrayOfNulls()/emptyArray()/intArrayOf()

// functions
- get()/set()/iterator()/indices/lastIndex

// extension functions
- asIterable()/asList()/asSequence()/associate()
- distinct()/distinctBy()/drop()/dropLast()/binarySearch()

- find()/findLast()
- fill()/contains()/copyOf()/count()

- elementAt()/elementAtOrNull()/first()/last()
- all()/any()/filter()/filterNot()/filterTo()/flatten()

- forEach()/forEachIndexed()/map()/mapIndexed()/groupBy()
- intersect()()/joinTo()/joinToString()
```

Collections

```
val numbers: MutableList<Int> = mutableListOf(1, 2, 3)
println(numbers)        // prints "[1, 2, 3]"
numbers.add(4)

val readOnlyView: List<Int> = numbers
readOnlyView.clear()    // -> does not compile

val strings = hashSetOf("a", "b", "c", "c")

val items = listOf(1, 2, 3, 4)
items.last == 4
items.filter { it % 2 == 0 }   // Returns [2, 4]

if (rwList.none { it > 6 }) println("No items above 6")
val item = rwList.firstOrNull()

val readWriteMap = hashMapOf("foo" to 1, "bar" to 2)
println(map["foo"])
val snapshot: Map<String, Int> = HashMap(readWriteMap)
```

Ranges

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

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

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

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

// progression with values [1, 5, 9]
(1..12 step 4).last == 9
```

Collection Extensions

```
//Iterable, Collection, List, Set, Map

- iterator() // Iterable
- size/indices/count()/isEmpty()/contains() // Collection
- lastIndex/get()/indexOf()/listIterator()/subList() // List

// Extensions Functions for List
- toTypedArray()/toMutableList()
- isNotEmpty()/?orEmpty()/plus()/plusElement()
- asReversed()/binarySearch()/findLast()

- dropLast()/takeLast()/reduceRight()
- getOrNull()/elementAt()/elementAtOrNull()
- single()/slice()/first()/firstOrNull()/last()
- +/plus()

// MutableIterable, MutableCollection, MutableList

- add()/addAll()/remove()/removeAt()
- removeAll()/retainAll()/clear()
- reverse()/sort()/sortBy()/sortByDescending()/sortWith()
- +=/plusAssign() -=/minusAssign()
```

Generics

```
- 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?>!
```

```kotlin
// Array<out Any> -> Java: Array<? extends Object>
fun copy2(from: Array<out Any>, to: Array<Any>) {
    // ...
}

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

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

Kotlin的泛型

- 与Java相似，Kotlin中的类也具有类型参数，一般而言，创建类的实例时，我们需要声明参数的类型，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 tal
```

- 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并不是List的子类型。 原因在于，如果List是可变的，并不会 优于Java数组。

- 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表示为Collection<? extends Object>的子类型。 简而言之，带extends限定（上限）的通配符类型（wildcard）使得类型是协变的（covariant）。

- out修饰符叫做型变注解，同时由于它在参数类型位置被提供，所以我们讨论声明处型变。 与Java的使用处型变相反，类型使用通配符使得类型协变。另外除了out，Kotlin又补充了一项型变注释：in。它是的变量类型反变：只可以被消费而不可以 被生产。反变类的一个很好的例子是 Comparable

Kotlin中的Java泛型

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

- Java的通配符转换成类型投射
  - Foo<? extends Bar> 转换成 Foo!
  - Foo<? super Bar> 转换成 Foo!

- Java的原始类型转换成星号投射
  - List 转换成 List<*>!, 也就是 List!

和Java一样，Kotlin在运行时不保留泛型，即对象不知道传递到他们构造器中的那些参数的的实际类型。

Kotlin的范型就像Java一样不会在运行时保留信息，也就是对象不会携带传递到它们构造函数中的类型参数的信息。也就是说，运行时无法区分ArrayList() 和 ArrayList().

也就是，ArrayList() 和 ArrayList() 是区分不出来的。 这意味着，不可能用 is-来检测泛型。

这就导致，无法使用is-检测范型。~~ Kotlin只允许用is-来检测星号投射的泛型类型: Kotlin只允许用is-检测星号投射的范型类型。

Others

Smart Cast

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

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

// 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)
    // x is automatically cast to String
      print(x.length)

```

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

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

Type Cast

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

```
val x: String = y as String

val x: String? = y as String?

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

“不安全”的转换运算符

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

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

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

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

“安全的”（可为空的）转换运算符

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

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

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

This

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

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

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

This的语义

为了记录下当前的接受者我们使用this表达式:

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

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

为了在范围外部访问this(一个类, 或者扩展函数, 或者带标签的带接收者的字面函数 我们使用this@label作为label

Null Safety

```kotlin
var a: String = "abc"
a = null // compilation error

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

val l = a.length
val l = b.length // error: variable 'b' can be null

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

val l: Int = if (b != null) b.length else -1
val l = b?.length ?: -1
val l = b!!.length() // npe
val aInt: Int? = a as? Int

val files = File("Test").listFiles()
println(files?.size)
println(files?.size ?: "empty")
```

可空（Nullable）和不可空（Non-Null） 类型

在 Kotlin 的类型体系里，有空类型和非空类型。类型系统识别出了 string 是一个非空类型，并且阻止编译器让它以空的状态存在。想要让一个变量为空，我们需要在声明后面加一个？号，同时赋值为 null。

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

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

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

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

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

安全的调用

你的第二个选择是安全的操作符，写作 ?. ： b?.length

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

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

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

如果任意一个属性（环节）为空，这个链式调用就会返回 null。

Elvis 操作符

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

`val l: Int = if (b != null) b.length else -1`
对于完整的 if-表达式, 可以换成 Elvis 操作符来表达, 写作 ?::

`val l = b?.length ?: -1`
如果 ?: 的左边表达式是非空的， elvis 操作符就会返回左边的结果, 否则返回右边的内容。 请注意，仅在左侧为空的时候，右侧表达式才会进行计算。

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

!! 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。

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
```

Exceptions

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

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

Reference

```
val c = MyClass::class

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

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

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

fun length(s: String) = s.size

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

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

类引用

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

val c = MyClass::class 引用是KClass类型.你可以使用KClass.properties 和KClass.extensionProperties 来获得类和父类所有属性引用的列表。

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

函数引用

我们有一个像下面这样的函数声明:

fun isOdd(x: Int) = x % 2 != 0 我们可以直接调用(isOdd(5)), 也可以把它作为一个值传给其他函数. 我们使用::操作符实现:

val numbers = listOf(1, 2, 3) println(numbers.filter(::isOdd)) // prints [1, 3] 这里 ::isOdd是一个函数类型的值 (Int) -> Boolean.

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

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

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。

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: center, middle

Java Interop

Calling Java from Kotlin

```kotlin
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
}

val calendar = Calendar.getInstance()
if (calendar.firstDayOfWeek == Calendar.SUNDAY) {
    calendar.firstDayOfWeek = Calendar.MONDAY
}
```

- 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)。

Null安全性和平台类型

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

平台类型的概念

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

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

映射类型

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

Java数组

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

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

Calling Java from Kotlin

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

val fooClass = foo.javaClass // foo.getClass()
val fooClass = javaClass<Foo>() // Foo.class

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

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

Java Varargs

Java类也会这样声明方法，表示参数是可变参数。这种情况，你需要用展开操作符 * 来传递 IntArray，目前无法传递 null 给一个变参的方法。

对象方法

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

Java 反射

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

java类的继承

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

访问静态成员

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

SAM(单抽象方法) 转换

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

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

注意SAM的转换只对接口有效，对抽象类无效，即使它们就只有一个 抽象方法。

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

Calling Kotlin from Java

```kotlin
@file:JvmName("DemoUtils")
package demo
class Foo

fun bar() {
}

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

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

可以使用 @JvmName 注解自定义生成的Java 类的类名

属性

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

包级别的函数

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

类名

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

实例字段

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

Calling Kotlin from Java

```
class C {
  companion object {
    const val VERSION = 1

    @JvmStatic fun foo() {}
    fun bar() {}

    @JvmField val COMPARATOR: Comparator<Key>
        = compareBy<Key> { it.value }
  }
}
```

```
C.foo(); // works fine
C.bar(); // error: not a static method
C.COMPARATOR.compare(key1, key2);
int v = C.VERSION;
```

静态字段

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

通常，这些字段都是private的，但是他们可以通过以下方式暴露出来。

- @JvmField 注解;
- lateinit 修饰符;

- const 修饰符.

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

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

静态方法

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

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

Calling Kotlin from Java

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

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

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

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

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

用@JvmName解决签名冲突

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

fun List.filterValid(): List fun List.filterValid(): List

这两个函数不能同时定义，因为它们的 JVM 签名是一样的：filterValid(Ljava/util/List;)Ljava/util/List;. 如果我们真的相让它们在 Kotlin里用同一个名字，我们需要用@JvmName去注释它们中的一个（或两个），指定的另外一个名字当参数

Kotlin里它们可以都用filterValid来访问，但是在Java里，它们是filterValid 和 filterValidInt. 同样的技巧也适用于属性 x 和函数 getX() 共存

生成重载

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

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

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

Null安全性

当从 Java 中调用 Kotlin 函数时，没人阻止我们传递 null 给一个非空参数。 这就是为什么 Kotlin 给所有期望非空参数的公开函数生成运行时检测。 这样我们就能在 Java 代码里立即得到 NullPointerException。

Nothing 类型的转换

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