



Kotlin 语言参考文档 中文版 (2016年04月)

原作者: JetBrains 公司
中文版译者: 李 颖(liying.cn.2010@gmail.com)

目录

前言	4
关于翻译	4
参考文档	4
相关书籍	4
入门	6
基本语法	6
惯用法	13
编码规约	19
基础	20
基本类型	20
包	27
控制流	29
返回与跳转	33
类与对象	35
类与继承	35
属性(Property)与域(Field)	42
接口	46
可见度修饰符	48
扩展	51
数据类	57
泛型	59
泛型函数	64
泛型约束(Generic constraint)	64
嵌套类(Nested Class)	66
枚举类	67
对象表达式(Object Expression)与对象声明(Object Declaration)	69
委托(Delegation)	73
委托属性(Delegated Property)	74
函数与 Lambda 表达式	78

函数	78
高阶函数与 Lambda 表达式	85
内联函数(Inline Function)	90
其他	94
解构声明(Destructuring Declaration)	94
集合(Collection)	96
值范围(Range)	98
类型检查与类型转换	101
this 表达式	103
相等判断	104
操作符重载(Operator overloading)	105
Null 值安全性	109
异常(Exception)	112
注解(Annotation)	114
反射	119
类型安全的构建器(Type-Safe Builder)	123
动态类型(Dynamic Type)	129
参考	130
与 Java 的互操作性	133
在 Kotlin 中调用 Java 代码	133
在 Java 中调用 Kotlin	142
工具	150
为 Kotlin 代码编写文档	150
使用 Maven	154
使用 Ant	157
使用 Gradle	161
Kotlin 与 OSGi	165
FAQ	167
FAQ	167
与 Java 比较	170
与 Scala 比较	172

前言

关于翻译

本文是 Kotlin 语言参考文档的中文翻译版.

原文

网址: <https://kotlinlang.org/docs/reference/>

代码库: <https://github.com/JetBrains/kotlin-web-site>

中文翻译版

网址: <http://www.liying-cn.net/kotlin/docs/reference/>

代码库: <https://github.com/LiYing2010/kotlin-web-site>

翻译者: 李 颖 (liying.cn.2010@gmail.com)

关于本文档的任何问题, 欢迎与译者联系.

参考文档

关于 Kotlin 语言和 [标准库](#) 的完整参考文档.

从哪里开始

本参考文档的目标是帮助你在几个小时之内很容易地学习 Kotlin. 请从 [基本语法](#) 开始, 然后继续阅读更高级的内容. 阅读本文档时, 你可以在 [online IDE](#) 中试验文档中的例子程序.

当你大致了解 Kotlin 之后, 你可以前往 [Kotlin Koans](#), 这里是一些交互式编程练习题, 你可以试着自己解决这些问题. 如果你不清楚如何解决某个练习题, 或者想要寻找更优雅的解决方式, 可以参考 [Kotlin 惯用法](#).

离线阅读

你可以下载本参考文档的 [PDF 版](#).

相关书籍

Kotlin 实战(Kotlin in Action)

□ [Kotlin 实战\(Kotlin in Action\)](#) 是一本关于 Kotlin 的书, 作者是 Dmitry Jemerov 和 Svetlana Isakova, 他们是 Kotlin 开发组的成员. 本书目前可以通过 MEAP 计划获取, 通过这种方式, 你可以在本书撰写过程中就提前阅读到各章内容, 并在本书最终完成之后得到完整版.

购买本书时使用 Coupon 号码 ‘39jemerov’ , 可获得 39% 的折扣.

面向 Android 的 Kotlin 手册(Kotlin for Android Developers)

□ [面向 Android 的 Kotlin 手册\(Kotlin for Android Developers\)](#) 由 Antonio Leiva 撰写, 本书向你展示如何使用 Kotlin 语言从零开始创建一个 Android 应用程序.

入门

基本语法

定义包

包的定义应该在源代码文件的最上方:

```
package my.demo

import java.util.*

// ...
```

源代码所在的目录结构不必与包结构保持一致: 源代码文件可以放置在文件系统的任意位置.

参见 [包](#).

定义函数

以下函数接受两个 `Int` 类型参数, 并返回 `Int` 类型结果:

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

以下函数使用表达式语句作为函数体, 返回类型由自动推断决定:

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

以下函数不返回有意义的结果:

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

返回值为 `Unit` 类型时, 可以省略:

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

参见 [函数](#).

定义局部变量

一次性赋值 (只读) 的局部变量:

```
val a: Int = 1  
val b = 1 // 变量类型自动推断为 `Int` 类型  
val c: Int // 没有初始化语句时, 必须明确指定类型  
c = 1 // 明确赋值
```

值可变的变量:

```
var x = 5 // 变量类型自动推断为 `Int` 类型  
x += 1
```

参见 [属性\(Property\)与域\(Field\)](#).

注释

与 Java 和 JavaScript 一样, Kotlin 支持行末注释, 也支持块注释.

```
// 这是一条行末注释  
  
/* 这是一条块注释  
   可以包含多行内容. */
```

与 Java 不同, Kotlin 中的块注释允许嵌套.

关于文档注释的语法, 详情请参见 [Kotlin 代码中的文档](#).

使用字符串模板

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

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

参见 [字符串模板](#).

使用条件表达式

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

以表达式的形式使用 `if`:

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

参见 [if 表达式](#).

使用可为 null 的值, 以及检查 `null`

当一个引用可能为 `null` 值时, 对应的类型声明必须明确地标记为可为 `null`.

当 `str` 中的字符串内容不是一个整数时, 返回 `null`:

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

以下示例演示如何使用一个返回值可为 `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` 会导致错误, 因为它们可能为 null.
    if (x != null && y != null) {
        // 在进行过 null 值检查之后, x 和 y 的类型会被自动转换为非 null 变量
        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
}

// 在进行过 null 值检查之后, x 和 y 的类型会被自动转换为非 null 变量
print(x * y)

```

参见 [Null 值安全](#).

使用类型检查和自动类型转换

`is` 运算符可以检查一个表达式的值是不是某个类型的实例. 如果对一个不可变的局部变量或属性进行过类型检查, 那么之后的代码就不必再对它进行显式地类型转换, 而可以直接将它当作需要的类型来使用:

```
fun getStringLength(obj: Any): Int? {
    if (obj is String) {
        // 在这个分支中, `obj` 的类型会被自动转换为 `String`
        return obj.length
    }

    // 在类型检查所影响的分支之外, `obj` 的类型仍然是 `Any`
    return null
}
```

或者

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

    // 在这个分支中, `obj` 的类型会被自动转换为 `String`
    return obj.length
}
```

甚至可以

```
fun getStringLength(obj: Any): Int? {
    // 在 `&&` 运算符的右侧, `obj` 的类型会被自动转换为 `String`
    if (obj is String && obj.length > 0)
        return obj.length

    return null
}
```

参见 [类](#) 和 [类型转换](#).

使用 for 循环

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

或者

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

参见 [for 循环](#).

使用 while 循环

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

参见 [while 循环](#).

使用 when 表达式

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

参见 [when expression](#).

使用范围值

使用 `in` 运算符检查一个数值是否在某个范围之内:

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

检查一个数值是否在某个范围之外:

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

在一个值范围内进行遍历迭代:

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

参见 [范围](#).

使用集合(Collection)

在一个集合上进行遍历迭代:

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

使用 `in` 运算符检查一个集合是否包含某个对象:

```
if (text in names) // 将会调用 names.contains(text) 方法
    print("Yes")
```

使用 Lambda 表达式, 对集合元素进行过滤和变换:

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

参见 [高阶函数与 Lambda 表达式](#).

惯用法

本章介绍 Kotlin 中的一些常见的习惯用法. 如果你有自己的好的经验, 可以将它贡献给我们. 你可以你的修正提交到 git, 并发起一个 Pull Request.

创建 DTO 类(或者叫 POJO/POCO 类)

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

以上代码将创建一个 `Customer` 类, 其中包含以下功能:

- 所有属性的 getter 函数(对于 `var` 型属性还有 setter 函数)
- `equals()` 函数
- `hashCode()` 函数
- `toString()` 函数
- `copy()` 函数
- 所有属性的 `component1()`, `component2()`, ... 函数(参见 [数据类](#))

对函数参数指定默认值

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

过滤 List 中的元素

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

甚至可以写得更短:

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

在字符串内插入变量值

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

类型实例检查

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

使用成对变量来遍历 Map, 这种语法也可以用来遍历 Pair 组成的 List

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

上例中的 `k`, `v` 可以使用任何的变量名.

使用数值范围

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

只读 List

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

只读 Map

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

访问 Map

```
println(map["key"])
map["key"] = value
```

延迟计算(Lazy)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(Singleton)

```
object Resource {
    val name = "Name"
}
```

If not null 的简写表达方式

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

println(files?.size)
```

If not null ... else 的简写表达方式

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

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

当值为 null 时, 执行某个语句

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

当值不为 null 时, 执行某个语句

```
val data = ...

data?.let {
    ... // 这个代码段将在 data 不为 null 时执行
}
```

在函数的 return 语句中使用 when 语句

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

将 ‘try/catch’ 用作一个表达式

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

    // 使用 result
}
```

将 ‘if’ 用作一个表达式

```
fun foo(param: Int) {
    val result = if (param == 1) {
        "one"
    } else if (param == 2) {
        "two"
    } else {
        "three"
    }
}
```

返回值为 Unit 类型的多个方法, 可以通过 Builder 风格的方式来串联调用

```
fun arrayOfMinusOnes(size: Int): IntArray {
    return IntArray(size).apply { fill(-1) }
}
```

使用单个表达式来定义一个函数

```
fun theAnswer() = 42
```


以上代码等价于:

```
fun theAnswer(): Int {  
    return 42  
}
```

这种用法与其他惯用法有效地结合起来, 可以编写出更简短的代码. 比如. 可以与 `when`-表达式结合起来:

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

在同一个对象实例上调用多个方法(‘with’ 语句)

```
class Turtle {  
    fun penDown()  
    fun penUp()  
    fun turn(degrees: Double)  
    fun forward(pixels: Double)  
}  
  
val myTurtle = Turtle()  
with(myTurtle) { // 描绘一个边长 100 像素的正方形  
    penDown()  
    for(i in 1..4) {  
        forward(100.0)  
        turn(90.0)  
    }  
    penUp()  
}
```

类似 Java 7 中针对资源的 try 语句

```
val stream = Files.newInputStream(Paths.get("/some/file.txt"))  
stream.bufferedReader().use { reader ->  
    println(reader.readText())  
}
```

对于需要泛型类型信息的函数, 可以使用这样的简便形式

```
// public final class Gson {  
// ...  
// public <T> T fromJson(JsonElement json, Class<T> classOfT) throws JsonSyntaxException {  
// ...
```

```
inline fun <reified T: Any> Gson.fromJson(json): T = this.fromJson(json, T::class.java)
```

编码规约

本章介绍 Kotlin 语言目前的编码风格.

命名风格

如果对某种规约存在疑问, 请默认使用 Java 编码规约, 比如:

- 在名称中使用驼峰式大小写(并且不要在名称中使用下划线)
- 类型首字母大写
- 方法和属性名称首字母小写
- 语句缩进使用 4 个空格
- public 函数应该编写文档, 这些文档将出现在 Kotlin Doc 中

冒号

当冒号用来分隔类型和父类型时, 冒号之前要有空格, 当冒号用来分隔类型和实例时, 冒号之前不加空格:

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

Lambda 表达式

在 Lambda 表达式中, 大括号前后应该加空格, 分隔参数与函数体的箭头符号前后也应该加空格. 如果有可能, 将 Lambda 表达式作为参数传递时, 应该尽量放在圆括号之外.

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

在简短, 并且无嵌套的 Lambda 表达式中, 推荐使用惯用约定的 `it` 作为参数名, 而不要明确地声明参数. 在有参数, 并且嵌套的 Lambda 表达式中, 应该明确地声明参数.

Unit

如果函数的返回值为 Unit 类型, 那么返回值的类型声明应当省略:

```
fun foo() { // 此处省略了 ": Unit"  
  
}
```

基础

基本类型

在 Kotlin 中, 一切都是对象, 这就意味着, 我们可以对任何变量访问它的成员函数和属性. 有些数据类型是内建的(built-in), 因为对它们的实现进行了优化, 但对于使用者来说内建类型与普通类没有区别. 本节我们将介绍大部分内建类型: 数值, 字符, 布尔值, 以及数组.

数值

Kotlin 处理数值的方式与 Java 类似, 但并不完全相同. 比如, 对数值不会隐式地扩大其值范围, 而且在某些情况下, 数值型的字面值(literal)也与 Java 存在轻微的不同.

Kotlin 提供了以下内建类型来表达数值(与 Java 类似):

类型	位宽度
Double	64
Float	32
Long	64
Int	32
Short	16
Byte	8

注意, 字符在 Kotlin 中不是数值类型.

字面值常数(Literal Constant)

对于整数值, 有以下几种类型的字面值常数:

- 10进制数: `123`
 - Long 类型需要大写的 `L` 来标识: `123L`
- 16进制数: `0x0F`
- 2进制数: `0b00001011`

注意: 不支持8进制数的字面值.

Kotlin 还支持传统的浮点数值表达方式:

- 无标识时默认为 Double 值: 123.5 , 123.5e10
- Float 值需要用 f 或 F 标识: 123.5f

内部表达

在 Java 平台中, 数值的物理存储使用 JVM 的基本类型来实现, 但当我们表达一个可为 null 的数值引用时(比如, Int?), 或者涉及到泛型时, 我们就不能使用基本类型了. 这种情况下数值会被装箱(box)为数值对象.

注意, 数值对象的装箱(box)并不保持对象的同一性(identity):

```
val a: Int = 10000
print(a === a) // 打印结果为 'true'
val boxedA: Int? = a
val anotherBoxedA: Int? = a
print(boxedA === anotherBoxedA) // !!!打印结果为 'false'!!!
```

但是, 装箱(box)会保持对象内容相等(equality):

```
val a: Int = 10000
print(a == a) // 打印结果为 'true'
val boxedA: Int? = a
val anotherBoxedA: Int? = a
print(boxedA == anotherBoxedA) // 打印结果为 'true'
```

显式类型转换

由于数据类型内部表达方式的差异, 较小的数据类型不会被看作较大数据类型的子类型(subtype). 如果小数据类型是大数据类型的子类型, 那么我们将会遇到以下问题:

```
// 以下为假想代码, 实际上是无法编译的:
val a: Int? = 1 // 装箱后的 Int (java.lang.Integer)
val b: Long? = a // 这里进行隐式类型转换, 产生一个装箱后的 Long (java.lang.Long)
print(a == b) // 结果与你期望的相反! 这句代码打印的结果将是 "false", 因为 Long 的 equals() 方法会检查比较对象, 要求对方也是一个 Long 对象
```

这样, 不仅不能保持同一性(identity), 而且还在所有发生隐式类型转换的地方, 保持内容相等(equality)的能力也静悄悄地消失了.

由于存在以上问题, Kotlin 不会将较小的数据类型隐式地转换为较大的数据类型. 也就是说, 如果不进行显式类型转换, 我们就不能将一个 Byte 类型值赋给一个 Int 类型的变量.

```
val b: Byte = 1 // 这是 OK 的, 因为编译器会对字面值进行静态检查
val i: Int = b // 这是错误的
```

我们可以使用显式类型转换, 来将数值变为更大的类型

```
val i: Int = b.toInt() // 这是 OK 的: 我们明确地扩大了数值的类型
```

所有的数值类型都支持以下类型转换方法:

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

Kotlin 语言中缺少了隐式类型转换的能力, 这个问题其实很少会引起使用者的注意, 因为类型可以通过代码上下文自动推断出来, 而且数学运算符都进行了重载(overload), 可以适应各种数值类型的参数, 比如

```
val l = 1L + 3 // Long 类型 + Int 类型, 结果为 Long 类型
```

运算符(Operation)

Kotlin 对数值类型支持标准的数学运算符(operation), 这些运算符都被定义为相应的数值类上的成员函数(但编译器会把对类成员函数的调用优化为对应的运算指令). 参见 [运算符重载](#).

对于位运算符, 没有使用特别的字符来表示, 而只是有名称的普通函数, 但调用这些函数时, 可以将函数名放在运算数的中间(即中缀表示法), 比如:

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

以下是位运算符的完整列表(只适用于 `Int` 类型和 `Long` 类型):

- `shl(bits)` – 带符号左移 (等于 Java 的 `<<`)
- `shr(bits)` – 带符号右移 (等于 Java 的 `>>`)
- `ushr(bits)` – 无符号右移 (等于 Java 的 `>>>`)
- `and(bits)` – 按位与(and)
- `or(bits)` – 按位或(or)
- `xor(bits)` – 按位异或(xor)

`inv()` - 按位取反

字符

字符使用 `Char` 类型表达. 字符不能直接当作数值使用

```
fun check(c: Char) {  
    if (c == 1) { // 错误: 类型不兼容  
        // ...  
    }  
}
```

字符的字面值(literal)使用单引号表达: `'1'`. 特殊字符使用反斜线转义表达. Kotlin 支持的转义字符包括: `\t`, `\b`, `\n`, `\r`, `\'`, `\"`, `\\` 以及 `\$`. 其他任何字符, 都可以使用 Unicode 转义表达方式: `'\uFF00'`.

我们可以将字符显式地转换为 `Int` 型数值:

```
fun decimalDigitValue(c: Char): Int {  
    if (c !in '0'..'9')  
        throw IllegalArgumentException("Out of range")  
    return c.toInt() - '0'.toInt() // 显式转换为数值  
}
```

与数值型一样, 当需要一个可为 `null` 的字符引用时, 字符会被装箱(box)为对象. 装箱操作不保持对象的同一性(identity).

布尔值

`Boolean` 类型用来表示布尔值, 有两个可能的值: `true` 和 `false`.

当需要一个可为 `null` 的布尔值引用时, 布尔值也会被装箱(box).

布尔值的内建运算符有

- `||` - 或运算(会进行短路计算)
- `&&` - 与运算(会进行短路计算)
- `!` - 非运算

数组

Kotlin 中的数组通过 `Array` 类表达, 这个类拥有 `get` 和 `set` 函数(这些函数通过运算符重载转换为 `[]` 运算符), 此外还有 `size` 属性, 以及其他一些有用的成员函数:

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

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

要创建一个数组, 我们可以使用库函数 `arrayOf()`, 并向这个函数传递一些参数来指定数组元素的值, 所以 `arrayOf(1, 2, 3)` 将创建一个数组, 其中的元素为 `[1, 2, 3]`. 或者, 也可以使用库函数 `arrayOfNulls()` 来创建一个指定长度的数组, 其中的元素全部为 `null` 值.

另一种方案是使用一个工厂函数, 第一个参数为数组大小, 第二个参数是另一个函数, 这个函数接受数组元素下标作为自己的输入参数, 然后返回这个下标对应的数组元素的初始值:

```
// 创建一个 Array<String>, 其中的元素为 ["0", "1", "4", "9", "16"]
val asc = Array(5, { i -> (i * i).toString() })
```

我们在前面提到过, `[]` 运算符可以用来调用数组的成员函数 `get()` 和 `set()`.

注意: 与 Java 不同, Kotlin 中数组的类型是不可变的. 所以 Kotlin 不允许将一个 `Array<String>` 赋值给一个 `Array<Any>`, 否则可能会导致运行时错误(但你可以使用 `Array<out Any>`, 参见 [类型投射](#)).

Kotlin 中也有专门的类来表达基本数据类型的数组: `ByteArray`, `ShortArray`, `IntArray` 等等, 这些数组可以避免数值对象装箱带来的性能损耗. 这些类与 `Array` 类之间不存在继承关系, 但它们的方法和属性是一致的. 各个基本数据类型的数组类都有对应的工厂函数:

```
val x: IntArray = intArrayOf(1, 2, 3)
x[0] = x[1] + x[2]
```

字符串

字符串由 `String` 类型表示. 字符串的内容是不可变的. 字符串中的元素是字符, 可以通过下标操作符来访问: `s[i]`. 可以使用 `for` 循环来遍历字符串:

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

字符串的字面值(literal)

Kotlin 中存在两种字符串面值: 一种称为转义字符串(escaped string), 其中可以包含转义字符, 另一种成为原生字符串(raw string), 其内容可以包含换行符和任意文本. 转义字符串(escaped string) 与 Java 的字符串非常类似:

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

转义字符使用通常的反斜线方式表示. 关于 Kotlin 支持的转义字符, 请参见上文的 [字符](#) 小节.

原生字符串(raw string)由三重引号表示(`"""`), 其内容不转义, 可以包含换行符和任意字符:

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

你可以使用 [trimMargin\(\)](#) 函数来删除字符串的前导空白(leading whitespace):

```
val text = """
| Tell me and I forget.
| Teach me and I remember.
| Involve me and I learn.
| (Benjamin Franklin)
""".trimMargin()
```

默认情况下, 会使用 `|` 作为前导空白的标记前缀, 但你可以通过参数指定使用其它字符, 比如 `trimMargin(">")`.

字符串模板

字符串内可以包含模板表达式, 也就是说, 可以包含一小段代码, 这段代码会被执行, 其计算结果将被拼接为字符串内容的一部分. 模板表达式以 `$` 符号开始, `$` 符号之后可以是一个简单的变量名:

```
val i = 10
val s = "i = $i" // 计算结果为 "i = 10"
```

`$` 符号之后也可以是任意的表达式, 由大括号括起:

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

原生字符串(raw string)和转义字符串(escaped string)内都支持模板. 由于原生字符串无法使用反斜线转义表达方式, 如果你想在字符串内表示 `$` 字符本身, 可以使用以下语法:

```
val price = """  
${'$'}9.99  
"""
```

包

源代码文件的开始部分可以是包声明:

```
package foo.bar

fun baz() {}

class Goo {}

// ...
```

源代码内的所有内容(比如类, 函数)全部都包含在所声明的包之内. 因此, 上面的示例代码中, `baz()` 函数的完整名称将是 `foo.bar.baz`, `Goo` 类的完整名称将是 `foo.bar.Goo`.

如果没有指定包, 那么源代码文件中的内容将属于 “default” 包, 这个包没有名称.

导入(Import)

除默认导入(Import)的内容之外, 各源代码可以包含自己独自の `import` 指令. `import` 指令的语法请参见 [语法](#).

我们可以导入一个单独的名称, 比如

```
import foo.Bar // 导入后 Bar 就可以直接访问, 不必指定完整的限定符
```

也可以导入某个范围(包, 类, 对象, 等等)之内所有可访问的内容:

```
import foo.* // 导入后 'foo' 内的一切都可以访问了
```

如果发生了名称冲突, 我们可以使用 `as` 关键字, 给重名实体指定新的名称(新名称仅在当前范围内有效):

```
import foo.Bar // 导入后 Bar 可以访问了
import bar.Bar as bBar // 可以使用新名称 bBar 来访问 'bar.Bar'
```

`import` 关键字不仅可以用来导入类; 还可以用来导入其他声明:

- 顶级(top-level) 函数和属性;
- [对象声明](#) 中定义的函数和属性;
- [枚举常数](#)

与 Java 不同, Kotlin 没有单独的 “import static” 语法; 所有这些声明都使用通常的 `import` 关键字来表达.

顶级(top-level) 声明的可见度

如果一个顶级(top-level) 声明被标注为 `private`, 它将成为私有的, 只有在它所属的文件内可以访问(参见 [可见度修饰符](#)).

控制流

if 表达式

在 Kotlin 中, `if` 是一个表达式, 也就是说, 它有返回值. 因此, Kotlin 中没有三元运算符(条件 ? then 分支返回值 : else 分支返回值), 因为简单的 `if` 表达式完全可以实现同样的任务.

```
// if 的传统用法
var max = a
if (a < b)
    max = b

// 使用 else 分支的方式
var max: Int
if (a > b)
    max = a
else
    max = b

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

`if` 的分支可以是多条语句组成的代码段, 代码段内最后一个表达式的值将成为整个代码段的返回值:

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

如果你将 `if` 作为表达式来使用(比如, 将它的值作为函数的返回值, 或将它的值赋值给一个变量), 而不是用作普通的流程控制语句, 这种情况下 `if` 表达式必须有 `else` 分支.

参见 [if 语法](#).

when 表达式

`when` 替代了各种 C 风格语言中的 `switch` 操作符. 最简单的形式如下例:

```

when (x) {
    1 -> print("x == 1")
    2 -> print("x == 2")
    else -> { // 注意, 这里是代码段
        print("x is neither 1 nor 2")
    }
}

```

`when` 语句会将它的参数与各个分支逐个匹配, 直到找到某个分支的条件成立. `when` 可以用作表达式, 也可以用作流程控制语句. 如果用作表达式, 满足条件的分支的返回值将成为整个表达式的值. 如果用作流程控制语句, 各个分支的返回值将被忽略. (与 `if` 类似, 各个分支可以是多条语句组成的代码段, 代码段内最后一个表达式的值将成为整个代码段的值.)

如果其他所有分支的条件都不成立, 则会执行 `else` 分支. 如果 `when` 被用作表达式, 则必须有 `else` 分支, 除非编译器能够证明其他分支的条件已经覆盖了所有可能的情况.

如果对多种条件需要进行相同的处理, 那么可以对一个分支指定多个条件, 用逗号分隔:

```

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

```

在分支条件中, 我们可以使用任意的表达式(而不仅仅是常数值)

```

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

```

我们还可以使用 `in` 或 `lin` 来检查一个值是否属于一个 [范围](#), 或者检查是否属于一个集合:

```

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")
}

```

还可以使用 `is` 或 `lis` 来检查一个值是不是某个类型. 注意, 由于 Kotlin 的 [智能类型转换](#) 功能, 进行过类型判断之后, 你就可以直接访问这个类型的方法和属性, 而不必再进行显式的类型检查.

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

`when` 也可以用来替代 `if-else if` 串. 如果没有指定参数, 那么所有的分支条件都应该是单纯的布尔表达式, 当条件的布尔表达式值为 `true` 时, 就会执行对应的分支:

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

参见 [when 语法](#).

for 循环

任何值, 只要能够产生一个迭代器(iterator), 就可以使用 `for` 循环进行遍历. 语法如下:

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

循环体可以是多条语句组成的代码段.

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

前面提到过, 凡是能够产生迭代器(iterator)的值, 都可以使用 `for` 进行遍历, 也就是说, 遍历对象需要满足以下条件:

- 存在一个成员函数- 或扩展函数 `iterator()`, 它的返回类型应该:
 - 存在一个成员函数- 或扩展函数 `next()`, 并且
 - 存在一个成员函数- 或扩展函数 `hasNext()`, 它的返回类型为 `Boolean` 类型.

上述三个函数都需要标记为 `operator`.

`for` 循环遍历数组时, 会被编译为基于数组下标的循环, 不会产生迭代器(iterator)对象.

如果你希望使用下标变量来遍历数组或 `List`, 可以这样做:

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

注意, 上例中的 “在数值范围内的遍历” 会在编译期间进行优化, 运行时不会产生额外的对象实例.

或者, 你也可以使用 `withIndex` 库函数:

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

参见 [for 语法](#).

while 循环

`while` 和 `do..while` 的功能与其他语言一样:

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

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

参见 [while 语法](#).

循环的中断(break)与继续(continue)

Kotlin 的循环支持传统的 `break` 和 `continue` 操作符. 参见 [返回与跳转](#).

返回与跳转

Kotlin 中存在 3 种程序流程跳出操作符

- `return`. 默认行为是, 从最内层的函数或 [匿名函数](#) 中返回.
- `break`. 结束最内层的循环.
- `continue`. 在最内层的循环中, 跳转到下一次循环.

Break 和 Continue 的位置标签

Kotlin 中的任何表达式都可以用 `label` 标签来标记. 标签的形式与标识符相同, 后面附加一个 `@` 符号, 比如: `abc@`, `fooBar@` 都是合法的标签(参见 [语法](#)). 要给一个表达式标记标签, 我们只需要将标签放在它之前:

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

然后, 我们就可以使用标签来限定 `break` 或 `continue` 的跳转对象:

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

通过标签限定后, `break` 语句, 将会跳转到这个标签标记的循环语句之后. `continue` 语句则会跳转到循环语句的下一循环.

使用标签控制 return 的目标

在 Kotlin 中, 通过使用字面值函数(function literal), 局部函数(local function), 以及对象表达式(object expression), 允许实现函数的嵌套. 通过标签限定的 `return` 语句, 可以从一个外层函数中返回. 最重要的使用场景是从 Lambda 表达式中返回. 回忆一下我们曾经写过以下代码:

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

这里的 `return` 会从最内层的函数中返回, 也就是. 从 `foo` 函数返回. (注意, 这种非局部的返回(non-local return), 仅对传递给 [内联函数\(inline function\)](#) 的 Lambda 表达式有效.) 如果需要从 Lambda 表达式返回, 我们必须对它标记一个标签, 然后使用这个标签来指明 `return` 的目标:

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

这样, `return` 语句就只从 Lambda 表达式中返回. 通常, 使用隐含标签会更方便一些, 隐含标签的名称与 Lambda 表达式被传递去的函数名称相同.

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

或者, 我们也可以使用 [匿名函数](#) 来替代 Lambda 表达式. 匿名函数内的 `return` 语句会从匿名函数内返回.

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

当 `return` 语句指定了返回值时, 源代码解析器会将这样的语句优先识别为使用标签限定的 `return` 语句, 也就是说:

```
return@a 1
```

含义是 “返回到标签 `@a` 处, 返回值为 `1`”, 而不是 “返回一个带标签的表达式 `(@a 1)`”.

类与对象

类与继承

类

Kotlin 中的类使用 `class` 关键字定义:

```
class Invoice {  
}
```

类的定义由以下几部分组成: 类名, 类头部(指定类的类型参数, 主构造器, 等等.), 以及由大括号括起的类主体部分. 类的头部和主体部分都是可选的; 如果类没有主体部分, 那么大括号也可以省略.

```
class Empty
```

构造器

Kotlin 中的类可以有一个 **主构造器** (primary constructor), 以及一个或多个 **次构造器** (secondary constructor). 主构造器是类头部的一部分, 位于类名称(以及可选的类型参数)之后.

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

如果主构造器没有任何注解(annotation), 也没有任何可见度修饰符, 那么 `constructor` 关键字可以省略:

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

主构造器中不能包含任何代码. 初始化代码可以放在 **初始化代码段** (initializer block) 中, 初始化代码段使用 `init` 关键字作为前缀:

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

注意, 主构造器的参数可以在初始化代码段中使用. 也可以在类主体定义的属性初始化代码中使用:

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

实际上, Kotlin 有一种简洁语法, 可以通过主构造器来定义属性并初始化属性值:

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

与通常的属性一样, 主构造器中定义的属性可以是可变的(**var**), 也可以是只读的(**val**).

如果构造器有注解, 或者有可见度修饰符, 这时 **constructor** 关键字是必须的, 注解和修饰符要放在它之前:

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

详情请参见 [可见度修饰符](#).

次级构造器(secondary constructor)

类还可以声明 **次级构造器** (secondary constructor), 使用 **constructor** 关键字作为前缀:

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

如果类有主构造器, 那么每个次级构造器都必须委托给主构造器, 要么直接委托, 要么通过其他次级构造器间接委托. 委托到同一个类的另一个构造器时, 使用 **this** 关键字实现:

```
class Person(val name: String) {
    constructor(name: String, parent: Person) : this(name) {
        parent.children.add(this)
    }
}
```

如果一个非抽象类没有声明任何主构造器和次级构造器, 它将带有一个自动生成的, 无参数的主构造器. 这个构造器的可见度为 `public`. 如果不希望你的类带有 `public` 的构造器, 你需要声明一个空的构造器, 并明确设置其可见度:

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

注意: 在 JVM 中, 如果主构造器的所有参数都指定了默认值, 编译器将会产生一个额外的无参数构造器, 这个无参数构造器会使用默认参数值来调用既有的构造器. 有些库(比如 Jackson 或 JPA) 会使用无参数构造器来创建对象实例, 这个特性将使得 Kotlin 比较容易与这种库协同工作.

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

创建类的实例

要创建一个类的实例, 我们需要调用类的构造器, 调用方式与使用通常的函数一样:

```
val invoice = Invoice()

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

注意, Kotlin 没有 `new` 关键字.

类成员

类中可以包含以下内容:

- 构造器和初始化代码块
- [函数](#)
- [属性](#)
- [嵌套类和内部类](#)
- [对象声明](#)

继承

Kotlin 中所有的类都有一个共同的超类 `Any`, 如果类声明时没有指定超类, 则默认为 `Any`:

```
class Example // 隐含地继承自 Any
```

`Any` 不是 `java.lang.Object` ; 尤其要注意, 除 `equals()` , `hashCode()` 和 `toString()` 之外, 它没有任何成员. 详情请参见 [与 Java 的互操作性](#).

要明确声明类的超类, 我们在类的头部添加一个冒号, 冒号之后指定超类:

```
open class Base(p: Int)

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

如果类有主构造器, 那么可以(而且必须)在主构造器中使用主构造器的参数来初始化基类.

如果类没有主构造器, 那么所有的次级构造器都必须使用 `super` 关键字来初始化基类, 或者委托到另一个构造器, 由被委托的构造器来初始化基类. 注意, 这种情况下, 不同的次级构造器可以调用基类中不同的构造器:

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

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

类上的 `open` 注解(annotation) 与 Java 的 `final` 正好相反: 这个注解表示允许从这个类继承出其他子类. 默认情况下, Kotlin 中所有的类都是 `final` 的, 这种设计符合 [Effective Java](#), 一书中的第 17 条原则: *允许继承的地方, 应该明确设计, 并通过文档注明, 否则应该禁止继承*.

成员的覆盖

我们在前面提到过, 我们很注意让 Kotlin 中的一切都明白无误. 而且与 Java 不同, Kotlin 要求明确地注解来标识允许被子类覆盖的成员(我们称之为 `open`), 而且也要求明确地注解来标识对超类成员的覆盖:

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

对于 `Derived.v()` 必须添加 `override` 注解. 如果遗漏了这个注解, 编译器将会报告错误. 如果一个函数没有标注 `open` 注解, 比如上例中的 `Base.nv()`, 那么在子类中声明一个同名同参的方法将是非法的, 无论是否添加 `override` 注解, 都不可以. 在一个 `final` 类(比如, 一个没有添加 `open` 注解的类)中, 声明 `open` 成员是禁止的.

当一个子类成员标记了 `override` 注解来覆盖父类成员时, 覆盖后的子类成员本身也将是 `open` 的, 也就是说, 子类成员可以被自己的子类再次覆盖. 如果你希望禁止这种再次覆盖, 可以使用 `final` 关键字:

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

等一下! 这样一来, 我如何才能 hack 我用到的那些类库呢?!

你可能曾经习惯于从类库中的某个类继承一个子类, 然后覆盖掉类库设计者并不期望你覆盖的某些方法, 以这种比较肮脏的手段来 hack 类库. 但在 Kotlin 中, 类与成员默认都是 `final` 的, 我们针对继承和覆盖问题所选择的这种设计原则, 将会带来一个问题, 就是前面所说的那种类库 hack 方式将会变得比较困难.

但我们认为这并不是一个缺点, 理由如下:

- 程序设计的最佳实践原则认为, 你本来就不应该使用这种 hack 手段
- 其他语言(比如 C++, C#)也使用了类似的原则, 大家使用起来并未遇到问题
- 如果有人确实希望 hack, 仍然存在其他方法: 你可以用 Java 来编写你的 hack 代码, 然后通过 Kotlin 来调用(参见 [与 Java 的互操作性](#)). 另外 Aspect 框架就是为这类目的设计的, 你可以使用 Aspect 框架来解决这类问题.

覆盖的规则

在 Kotlin 中, 类继承中的方法实现问题, 遵守以下规则: 如果一个类从它的直接超类中继承了同一个成员的多个实现, 那么这个子类必须覆盖这个成员, 并提供一个自己的实现(可以使用继承得到的多个实现中的某一个). 为了表示使用的方法是从哪个超类继承得到的, 我们使用 `super` 关键字, 将超类名称放在尖括号类, 比如, `super<Base>`:

```

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

interface B {
    fun f() { print("B") } // 接口的成员默认是 'open' 的
    fun b() { print("b") }
}

class C() : A(), B {
    // 编译器要求 f() 方法必须覆盖:
    override fun f() {
        super<A>.f() // 调用 A.f()
        super<B>.f() // 调用 B.f()
    }
}

```

同时继承 A 和 B 是合法的, 而且函数 a() 和 b() 的继承也不存在问题, 因为对于这两个函数, C 类都只继承得到了唯一的一个实现. 但对函数 f() 的继承就发生了问题, 因为 C 类从超类中继承得到了两个实现, 因此在 C 类中我们必须覆盖函数 f(), 并提供我们自己的实现, 这样才能消除歧义.

抽象类

类本身, 或类中的部分成员, 都可以声明为 **abstract** 的. 抽象成员在类中不存在具体的实现. 注意, 我们不必对抽象类或抽象成员标注 open 注解 – 因为它显然必须是 open 的.

我们可以使用抽象成员来覆盖一个非抽象的 open 成员:

```

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

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

```

同伴对象(Companion Object)

与 Java 或 C# 不同, Kotlin 的类没有静态方法(static method). 大多数情况下, 建议使用包级函数(package-level function)替代静态方法.

如果你需要写一个函数, 希望使用者不必通过类的实例来调用它, 但又需要访问类的内部信息(比如, 一个工厂方法), 你可以将这个函数写为这个类之内的一个 [对象声明](#) 的成员, 而不是类本身的成员.

具体来说, 如果你在类中声明一个 [同伴对象](#), 那么只需要使用类名作为限定符就可以调用同伴对象的成员了, 语法与 Java/C# 中调用类的静态方法一样.

封闭类(Sealed Class)

封闭类(Sealed class)用来表示对类阶层的限制, 可以限定一个值只允许是某些指定的类型之一, 而不允许是其他类型. 感觉上, 封闭类是枚举类(enum class)的一种扩展: 枚举类的值也是有限的, 但每一个枚举值常数都只存在唯一的一个实例, 封闭类则不同, 它允许的子类类型是有限的, 但子类可以有多个实例, 每个实例都可以包含它自己的状态数据.

要声明一个封闭类, 需要将 `sealed` 修饰符放在类名之前. 封闭类可以有子类, 但所有的子类声明都必须嵌套在封闭类的声明部分之内.

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

注意, 从封闭类的子类再继承的子类(间接继承者)可以放在任何地方, 不必在封闭类的声明部分之内.

使用封闭类的主要好处在于, 当使用 [when expression](#) 时, 可以验证分支语句覆盖了所有的可能情况, 因此就不必通过 `else` 分支来处理例外情况.

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

属性(Property)与域(Field)

声明属性

Kotlin 中的类可以拥有属性. 可以使用 `var` 关键字声明为可变(mutable)属性, 也可以使用 `val` 关键字声明为只读属性.

```
public class Address {  
    public var name: String = ...  
    public var street: String = ...  
    public var city: String = ...  
    public var state: String? = ...  
    public var zip: String = ...  
}
```

使用属性时, 只需要简单地通过属性名来参照它, 和使用 Java 中的域变量(field)一样:

```
fun copyAddress(address: Address): Address {  
    val result = Address() // Kotlin 中没有 'new' 关键字  
    result.name = address.name // 将会调用属性的访问器方法  
    result.street = address.street  
    // ...  
    return result  
}
```

取值方法(Getter)与设值方法(Setter)

声明属性的完整语法是:

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

其中的初始化器(initializer), 取值方法(getter), 以及设值方法(setter)都是可选的. 如果属性类型可以通过初始化器自动推断得到, 或者可以通过这个属性覆盖的基类成员属性推断得到, 则属性类型的声明也可以省略.

示例:

```
var allByDefault: Int? // 错误: 需要明确指定初始化器, 此处会隐含地使用默认的取值方法和设值方法  
var initialized = 1 // 属性类型为 Int, 使用默认的取值方法和设值方法
```

只读属性声明的完整语法与可变属性有两点不同: 由 `val` 开头, 而不是 `var`, 并且不允许指定设值方法:

```
val simple: Int? // 属性类型为 Int, 使用默认的取值方法, 属性值必须在构造器中初始化
val inferredType = 1 // 属性类型为 Int, 使用默认的取值方法
```

我们可以编写自定义的访问方法, 与普通的函数很类似, 访问方法的位置就在属性定义体之内. 下面是一个自定义取值方法的示例:

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

自定义设值方法的示例如下:

```
var stringRepresentation: String
    get() = this.toString()
    set(value) {
        setDataFromString(value) // 解析字符串内容, 并将解析得到的值赋给对应的其他属性
    }
```

Kotlin 的编程惯例是, 设值方法的参数名称为 `value`, 但如果你喜欢, 也可以选择使用不同的名称.

如果你需要改变属性访问方法的可见度, 或者需要对其添加注解, 但又不需要修改它的默认实现, 你可以定义这个方法, 但不定义它的实现体:

```
var setterVisibility: String = "abc"
    private set // 设值方法的可见度为 private, 并使用默认实现

var setterWithAnnotation: Any? = null
    @Inject set // 对设值方法添加 Inject 注解
```

属性的后端域变量(Backing Field)

Kotlin 的类不能拥有域变量. 但是, 使用属性的自定义访问器时, 有时需要后端域变量(backing field). 为了这种目的, Kotlin 提供了一种自动的后端域变量, 可以通过 `field` 标识符来访问:

```
var counter = 0 // 初始化给定的值将直接写入后端域变量中
    set(value) {
        if (value >= 0)
            field = value
    }
```

`field` 标识符只允许在属性的访问器函数内使用.

编译器会检查属性访问器的函数体, 如果使用了后端域变量(或者, 如果没有指定访问器的函数体, 使用了默认实现), 编译器就会生成一个后端域变量, 否则, 就不会生成后端域变量.

比如, 下面的情况不会存在后端域变量:

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

后端属性(Backing Property)

如果你希望实现的功能无法通过这种 “隐含的后端域变量” 方案来解决, 你可以使用 *后端属性(backing property)* 作为替代方案:

```
private var _table: Map<String, Int>? = null
public val table: Map<String, Int>
    get() {
        if (_table == null)
            _table = HashMap() // 类型参数可以自动推断得到, 不必指定
        return _table ?: throw AssertionError("Set to null by another thread")
    }
```

不管从哪方面看, 这种方案都与 Java 中完全相同, 因为后端私有属性的取值方法与设值方法都使用默认实现, 我们对这个属性的访问将被编译器优化, 变为直接读写后端域变量, 因此不会发生不必要的函数调用, 导致性能损失.

编译期常数值

如果属性值在编译期间就能确定, 则可以使用 `const` 修饰符, 将属性标记为_编译期常数值(compile time constants). 这类属性必须满足以下所有条件:

- 必须是顶级属性, 或者是一个 `object` 的成员
- 值被初始化为 `String` 类型, 或基本类型(primitive type)
- 不存在自定义的取值方法

这类属性可以用在注解内:

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

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

延迟初始化属性(Late-Initialized Property)

通常, 如果属性声明为非 `null` 数据类型, 那么属性值必须在构造器内初始化. 但是, 这种限制很多时候会带来一些不便. 比如, 属性值可以通过依赖注入来进行初始化, 或者在单元测试代码的 `setup` 方法中初始化. 这种情况下, 你就无法在构造器中为属性编写一段非 `null` 值的初始化代码, 但你仍然希望在类内参照这个属性时能够避免 `null` 值检查.

要解决这个问题, 你可以为属性添加一个 `lateinit` 修饰符:

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

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

    @Test fun test() {
        subject.method() // 直接访问属性
    }
}
```

这个修饰符只能用于 `var` 属性, 而且只能是声明在类主体部分之内的属性(不可以是主构造器中声明的属性), 而且属性不能有自定义的取值方法和设值方法. 属性类型必须是非 `null` 的, 而且不能是基本类型.

在一个 `lateinit` 属性被初始化之前访问它, 会抛出一个特别的异常, 这个异常将会指明被访问的属性, 以及它没有被初始化这一错误.

属性的覆盖

参见 [成员的覆盖](#)

委托属性(Delegated Property)

最常见的属性只是简单地读取(也有可能会写入)一个后端域变量. 但是, 通过使用自定义的取值方法和设值方法, 我们可以实现属性任意复杂的行为. 在这两种极端情况之间, 还存在一些非常常见的属性工作模式. 比如: 属性值的延迟加载, 通过指定的键值(key)从 `map` 中读取数据, 访问数据库, 属性被访问时通知监听器, 等等.

这些常见行为可以使用 [委托属性\(delegated property\)](#), 以库的形式实现.

接口

Kotlin 中的接口与 Java 8 非常类似. 接口中可以包含抽象方法的声明, 也可以包含方法的实现. 接口与抽象类的区别在于, 接口不能存储状态数据. 接口可以有属性, 但这些属性必须是抽象的, 或者必须提供访问器的自定义实现.

接口使用 `interface` 关键字来定义:

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

实现接口

类或者对象可以实现一个或多个接口:

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

接口中的属性

你可以在接口中定义属性. 接口中声明的属性要么是抽象的, 要么提供访问器的自定义实现. 接口中声明的属性不能拥有后端域变量(backing field), 因此, 在接口中定义的属性访问器也不能访问属性的后端域变量.

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

解决覆盖冲突(overriding conflict)

当我们为一个类指定了多个超类, 可能会导致我们对同一个方法继承得到了多个实现. 比如:

```
interface A {  
    fun foo() { print("A") }  
    fun bar()  
}  
  
interface B {  
    fun foo() { print("B") }  
    fun bar() { print("bar") }  
}  
  
class C : A {  
    override fun bar() { print("bar") }  
}  
  
class D : A, B {  
    override fun foo() {  
        super<A>.foo()  
        super<B>.foo()  
    }  
}
```

接口 *A* 和 *B* 都定义了函数 *foo()* 和 *bar()*. 它们也都实现了 *foo()*, 但只有 *B* 实现了 *bar()* (在 *A* 中 *bar()* 没有标记为 *abstract*, 因为在接口中, 如果没有定义函数体, 则函数默认为 *abstract*). 现在, 如果我们从 *A* 派生一个实体类 *C*, 显然, 我们必须覆盖函数 *bar()*, 并提供一个实现. 如果我们从 *A* 和 *B* 派生出 *D*, 我们就不必覆盖函数 *bar()*, 因为我们从超类中继承得到了它的唯一一个实现. 但是对于函数 *foo()*, 我们继承得到了两个实现, 因此编译器不知道应该选择使用哪个实现, 因此编译器强制要求我们覆盖函数 *foo()*, 明确地告诉编译器, 我们究竟希望做什么.

可见度修饰符

类, 对象, 接口, 构造器, 函数, 属性, 以及属性的设值方法, 都可以使用_可见度修饰符_(属性的取值方法永远与属性本身的可见度一致, 因此不需要控制其可见度。) Kotlin 中存在 4 种可见度修饰符: `private`, `protected`, `internal` 以及 `public`. 如果没有明确指定修饰符, 则使用默认的可见度 `public`.

在不同的范围内, 这些可见度的含义是不同的, 详情请看下文.

包

函数, 属性, 类, 对象, 接口, 都可以声明为”顶级的(top-level)”, 也就是说, 直接声明在包之内:

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

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

- 如果你不指定任何可见度修饰符, 默认会使用 `public`, 其含义是, 你声明的东西在任何位置都可以访问;
- 如果你将声明的东西标记为 `private`, 那么它将只在同一个源代码文件内可以访问;
- 如果标记为 `internal`, 那么它将在同一个模块(module)内的任何位置都可以访问;
- 对于顶级(top-level)声明, `protected` 修饰符是无效的.

示例:

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

private fun foo() {} // 只在 example.kt 文件内可访问

public var bar: Int = 5 // 这个属性在任何地方都可以访问
    private set // 但它的设值方法只在 example.kt 文件内可以访问

internal val baz = 6 // 在同一个模块(module)内可以访问
```

类与接口

对于类内部的声明:

- `private` 表示只在这个类(以及它的所有成员)之内可以访问;
- `protected` — 与 `private` 一样, 另外在子类中也可以访问;
- `internal` — 在 本模块之内, 凡是能够访问到这个类的地方, 同时也能访问到这个类的 `internal` 成员;
- `public` — 凡是能够访问到这个类的地方, 同时也能访问这个类的 `public` 成员.

Java 使用者 请注意: 在 Kotlin 中, 外部类(outer class)不能访问其内部类(inner class)的 private 成员.

示例:

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

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

class Subclass : Outer() {
    // a 不可访问
    // b, c 和 d 可以访问
    // Nested 和 e 可以访问
}

class Unrelated(o: Outer) {
    // o.a, o.b 不可访问
    // o.c 和 o.d 可以访问(属于同一模块)
    // Outer.Nested 不可访问, Nested::e 也不可访问
}
```

构造器

要指定类的主构造器的可见度, 请使用以下语法(注意, 你需要明确添加一个 `constructor` 关键字):

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

这里构造器是 private 的. 所有构造器默认都是 `public` 的, 因此使得凡是访问到类的地方都可以访问到类的构造器(也就是说, 一个 `internal` 类的构造器只能在同一个模块内访问).

局部声明

局部变量, 局部函数, 以及局部类, 都不能指定可见度修饰符.

模块(Module)

`internal` 修饰符表示这个成员只能在同一个模块内访问. 更确切地说, 一个模块(module)是指一起编译的一组 Kotlin 源代码文件:

— 一个 IntelliJ IDEA 模块;

- 一个 Maven 工程, 或 Gradle 工程;
- 通过 Ant 任务的一次调用编译的一组文件.

扩展

与 C# 和 Gosu 类似, Kotlin 提供了向一个类扩展新功能的能力, 而且不必从这个类继承, 也不必使用任何设计模式, 比如 Decorator 模式之类. 这种功能是通过一种特殊的声明来实现的, Kotlin 中称为 *扩展(extension)*. Kotlin 支持 *扩展函数(extension function)* 和 *扩展属性(extension property)*.

扩展函数(Extension Function)

要声明一个扩展函数, 我们需要在函数名之前添加前缀, 表示这个函数的 *接收者类型(receiver type)*, 也就是说, 表明我们希望扩展的对象类型. 以下示例将为 `MutableList<Int>` 类型添加一个 `swap` 函数:

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

在扩展函数内, `this` 关键字指代接收者对象(receiver object)(也就是调用扩展函数时, 在点号之前指定的对象实例). 现在, 我们可以对任意一个 `MutableList<Int>` 对象调用这个扩展函数:

```
val l = mutableListOf(1, 2, 3)  
l.swap(0, 2) // 'swap()' 函数内的 'this' 将指向 'l' 的值
```

显然, 这个函数可以适用与任意元素类型的 `MutableList<T>`, 因此我们可以使用泛型, 将它的元素类型泛化:

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

我们在函数名之前声明了泛型的类型参数, 然后在接收者类型表达式中就可以使用泛型了. 参见 [泛型函数](#).

扩展函数是静态解析的

扩展函数并不会真正修改它所扩展的类. 定义扩展函数时, 其实并没有向类中插入新的成员方法, 而只是创建了一个新的函数, 并且可以通过点号标记法的形式, 对一个类的实例调用这个新函数.

我们希望强调一下, 扩展函数的调用派发过程是**静态的**, 也就是说, 它并不是接收者类型的虚拟成员. 这意味着, 调用扩展函数时, 具体被调用的函数是哪一个, 是通过调用函数的对象表达式的类型来决定的, 而不是在运行时刻表达式动态计算的最终结果类型决定的. 比如:

```

open class C

class D: C()

fun C.foo() = "c"

fun D.foo() = "d"

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

printFoo(D())

```

这段示例程序的打印结果将是 “c”，因为调用哪个函数, 仅仅是由参数 `c` 声明的类型决定, 这里参数 `c` 的类型为 `C` 类.

如果类中存在成员函数, 同时又在同一个类上定义了同名的扩展函数, 并且与调用时指定的参数匹配, 这种情况下 总是会优先使用成员函数. 比如:

```

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

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

```

如果我们对 `C` 类型的任意变量 `c` 调用 `c.foo()`, 结果会打印 “member”, 而不是 “extension” .

可为空的接收者(Nullable Receiver)

注意, 对可以为空的接收者类型也可以定义扩展. 这样的扩展函数, 即使在对象变量值为 `null` 时也可以调用, 在扩展函数的实现体之内, 可以通过 `this == null` 来检查接收者是否为 `null`. 在 Kotlin 中允许你调用 `toString()` 函数, 而不必检查对象是否为 `null`, 就是通过这个原理实现的: 对象是否为 `null` 的检查发生在扩展函数内部, 因此调用者不必再做检查.

```

fun Any?.toString(): String {
    if (this == null) return "null"
    // 进行过 null 检查后, 'this' 会被自动转换为非 null 类型, 因此下面的 toString() 方法
    // 会被解析为 Any 类的成员函数
    return toString()
}

```

扩展属性(Extension Property)

与扩展函数类似, Kotlin 也支持扩展属性:

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

注意, 由于扩展属性实际上不会向类添加新的成员, 因此无法让一个扩展属性拥有一个 [后端域变量](#). 所以, 对于扩展属性不允许存在初始化器. 扩展属性的行为只能通过明确给定的取值方法与设值方法来定义.

示例:

```
val Foo.bar = 1 // 错误: 扩展属性不允许存在初始化器
```

对同伴对象(Companion Object)的扩展

如果一个类定义了[同伴对象](#), 你可以对这个同伴对象定义扩展函数和扩展属性:

```
class MyClass {
    companion object {} // 通过 "Companion" 来引用这个同伴对象
}

fun MyClass.Companion.foo() {
    // ...
}
```

与同伴对象的常规成员一样, 可以只使用类名限定符来调用这些扩展函数和扩展属性:

```
MyClass.foo()
```

扩展的范围

大多数时候我们会在顶级位置定义扩展, 也就是说, 直接定义在包之下:

```
package foo.bar

fun Baz.goo() { ... }
```

要在扩展定义所在的包之外使用扩展, 我们需要在调用处 import 这个包:

```

package com.example.usage

import foo.bar.goo // 通过名称 "goo" 来导入扩展
// 或者
import foo.bar.* // 导入 "foo.bar" 包之下的全部内容

fun usage(baz: Baz) {
    baz.goo()
}

```

详情请参见 [导入](#).

将扩展定义为成员

在类的内部, 你可以为另一个类定义扩展. 在这类扩展中, 存在多个 *隐含接受者(implicit receiver)* - 这些隐含接收者的成员可以不使用限定符直接访问. 扩展方法的定义所在的类的实例, 称为*_派发接受者(dispatch receiver)*, 扩展方法的目标类型的实例, 称为*_扩展接受者(extension receiver)*.

```

class D {
    fun bar() { ... }
}

class C {
    fun baz() { ... }

    fun D.foo() {
        bar() // 这里将会调用 D.bar
        baz() // 这里将会调用 C.baz
    }

    fun caller(d: D) {
        d.foo() // 这里将会调用扩展函数
    }
}

```

当派发接受者与扩展接受者的成员名称发生冲突时, 扩展接受者的成员将会被优先使用. 如果想要使用派发接受者的成员, 请参见 [带限定符的 this 语法](#).

```

class C {
    fun D.foo() {
        toString() // 这里将会调用 D.toString()
        this@C.toString() // 这里将会调用 C.toString()
    }
}

```

以成员的形式定义的扩展函数, 可以声明为 `open`, 而且可以在子类中覆盖. 也就是说, 在这类扩展函数的派发过程中, 针对派发接受者是虚拟的(virtual), 但针对扩展接受者仍然是静态的(static).

```
open class D {
}

class D1 : D() {
}

open class C {
    open fun D.foo() {
        println("D.foo in C")
    }

    open fun D1.foo() {
        println("D1.foo in C")
    }

    fun caller(d: D) {
        d.foo() // 调用扩展函数
    }
}

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

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

C().caller(D()) // 打印结果为 "D.foo in C"
C1().caller(D()) // 打印结果为 "D.foo in C1" - 派发接受者的解析过程是虚拟的
C().caller(D1()) // 打印结果为 "D.foo in C" - 扩展接受者的解析过程是静态的
```

使用扩展的动机

在 Java 中, 我们通常会使用各种名为 “*Utils” 的工具类: `FileUtils`, `StringUtils` 等等. 著名的 `java.util.Collections` 也属于这种工具类. 这种工具类模式令人很不愉快的地方在于, 使用时代码会写成这种样子:

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

代码中反复出现的工具类类名非常烦人. 我们也可以使用静态导入(static import), 然后代码会变成这样:

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

这样略好了一点点, 但是没有了类名做前缀, 就导致我们无法利用 IDE 强大的代码自动补完功能. 如果我们能写下面这样的代码, 那不是很好吗:

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

但是我们又不希望将一切可能出现的方法在 `List` 类之内全部都实现出来, 对不对? 这恰恰就是 Kotlin 的扩展机制可以帮助我们解决的问题.

数据类

我们经常会创建一些数据类, 什么功能也没有, 而仅仅用来保存数据. 在这些类中, 某些常见的功能经常可以由类中保存的数据内容即可自动推断得到. 在 Kotlin 中, 我们将这样的类称为 *数据类*, 通过 `data` 关键字标记:

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

编译器会根据主构造器中声明的全部属性, 自动推断产生以下成员函数:

- `equals()` / `hashCode()` 函数对,
- `toString()` 函数, 输出格式为 `"User(name=John, age=42)"`,
- [componentN\(\)](#) 函数群, 这些函数与类的属性对应, 函数名中的数字 1 到 N, 与属性的声明顺序一致,
- `copy()` 函数 (详情见下文).

如果上述任意一个成员函数在类定义体中有明确的定义, 或者从基类继承得到, 那么这个成员函数不会自动生成.

为了保证自动生成的代码的行为一致, 并且有意义, 数据类必须满足以下所有要求:

- 主构造器至少要有有一个参数;
- 主构造器的所有参数必须标记为 `val` 或 `var`;
- 数据类不能是抽象类, `open` 类, 封闭(`sealed`)类, 或内部(`inner`)类;
- 数据类不能继承自任何其他类(但可以实现接口).

在 JVM 上, 如果自动生成的类需要拥有一个无参数的构造器, 那么需要为所有的属性指定默认值 (参见 [构造器](#)).

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

对象复制

我们经常会需要复制一个对象, 然后修改它的一部分属性, 但保持其他属性不变. 这就是自动生成的 `copy()` 函数所要实现的功能. 对于前面示例中的 `User` 类, 自动生成的 `copy()` 函数的实现将会是下面这样:

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

有了这个函数, 我们可以编写下面这样的代码:

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

数据类中成员数据的解构

编译器会为数据类生成 *组件函数(Component function)*, 有了这些组件函数, 就可以在 [解构声明\(destructuring declaration\)](#) 中使用数据类:

```
val jane = User("Jane", 35)
val (name, age) = jane
println("$name, $age years of age") // 打印结果将是 "Jane, 35 years of age"
```

标准库中的数据类

Kotlin 的标准库提供了 `Pair` 和 `Triple` 类可供使用. 但是, 大多数情况下, 使用有具体名称的数据了是一种更好的设计方式, 因为, 数据类可以为属性指定有含义的名称, 因此可以增加代码的可读性.

泛型

与 Java 一样, Kotlin 中的类也可以有类型参数:

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

通常, 要创建这样一个类的实例, 我们需要指定类型参数:

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

但是, 如果类型参数可以通过推断得到, 比如, 通过构造器参数类型, 或通过其他手段推断得到, 此时允许省略类型参数:

```
val box = Box(1) // 1 的类型为 Int, 因此编译器知道我们创建的实例是 Box<Int> 类型
```

类型变异(Variance)

Java 的类型系统中, 最微妙, 最难于理解和使用的部分之一, 就是它的通配符类型(wildcard type) (参见 [Java 泛型 FAQ](#)). Kotlin 中不存在这样的通配符类型. 它使用另外的两种东西: 声明处类型变异(declaration-site variance), 以及类型投射(type projection).

首先, 我们来思考一下为什么 Java 需要那些神秘的通配符类型. 这个问题已有详细的解释, 请参见 [Effective Java](#), 第 28 条: *为增加 API 的灵活性, 应该使用限定范围的通配符类型(bounded wildcard)*. 首先, Java 中的泛型类型是 **不可变的(invariant)**, 也就是说 `List<String>` 不是 `List<Object>` 的子类型. 为什么会这样? 因为, 如果 `List` 不是 **不可变的(invariant)**, 那么下面的代码将可以通过编译, 然后在运行时导致一个异常, 那么 `List` 就并没有任何优于 Java 数组的地方了:

```
// Java  
List<String> strs = new ArrayList<String>();  
List<Object> objs = strs; // !!! 导致后面问题的原因就在这里. Java 会禁止这样的代码!  
objs.add(1); // 在这里, 我们向 String 组成的 List 添加了一个 Integer 类型的元素  
String s = strs.get(0); // !!! ClassCastException: 无法将 Integer 转换为 String
```

由于存在这种问题, Java 禁止上面示例中的做法, 以便保证运行时刻的类型安全. 但这个原则背后存在一些隐含的影响. 比如, 我们来看看 `Collection` 接口的 `addAll()` 方法. 这个方法的签名应该是什么样的? 直觉地, 我们会将它定义为:

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

但是, 这样的定义会导致我们无法进行下面这种非常简单的操作(尽管这种操作是绝对安全的):

```
// Java
void copyAll(Collection<Object> to, Collection<String> from) {
    to.addAll(from); // !!! 如果 addAll 方法使用前面那种简单的定义, 这里的调用将无法通过编译:
    // 因为 Collection<String> 不是 Collection<Object> 的子类型
}
```

(在 Java 语言中, 我们通过非常痛苦的方式才学到了这个教训, 详情请参见 [Effective Java](#), 第 25 条: 尽量使用 List, 而不是数组)

正因为上面的问题, 所以 addAll() 的签名定义其实是这样的:

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

这里的 **通配符类型参数(wildcard type argument)** ? extends T 表示, 该方法接受的参数是一个集合, 集合元素的类型是 T 的某种子类型, 而不限于 T 本身. 这就意味着, 我们可以安全地从集合元素中读取 T (因为集合的元素是 T 的某个子类型的实例), 但不能写入到集合中去, 因为我们不知道什么样的对象实例才能与这个 T 的未知子类型匹配. 尽管有这样的限制, 作为回报, 我们得到了希望的功能: Collection<String> 是 Collection<? extends Object> 的子类型. 用“高级术语”来说, 指定了 extends 边界(上边界)的通配符类型, 使得我们的类型成为一种 **协变(covariant)** 类型.

要理解这种技巧的工作原理十分简单: 如果你只能从一个集合取得元素, 那么就可以使用一个 String 组成的集合, 并从中读取 Object 实例. 反过来, 如果你只能向集合放入元素, 那么就可以使用一个 Object 组成的集合, 并向其中放入 String: 在 Java 中, 我们可以使用 List<? super String>, 它是 List<Object> 的一个父类型.

上面的后一种情况称为 **反向类型变异(contravariance)**, 对于 List<? super String>, 你只能调用那些接受 String 类型参数的方法(比如, 可以调用 add(String), 或 set(int, String)), 而当你调用 List<T> 调用返回类型为 T 的方法时, 你得到的返回值将不会是 String 类型, 而只是 Object 类型.

Joshua Bloch 将那些只能读取的对象称为 **生产者(Producer)**, 将那些只能写入的对象称为 **消费者(Consumer)**. 他建议: “为尽量保证灵活性, 应该对代表生产者和消费者的输入参数使用通配符类型”, 他还提出了下面的记忆口诀:

PECS: 生产者(Producer)对应 Extends, 消费者(Consumer)对应 Super.

注意: 如果你使用一个生产者对象, 比如, List<? extends Foo>, 你将无法对这个对象调用 add() 或 set() 方法, 但这并不代表这个对象是 **值不变的(immutable)**: 比如, 你完全可以调用 clear() 方法来删除 List 内的所有元素, 因为 clear() 方法不需要任何参数. 通配符类型(或者其他任何的类型变异)唯一能够确保的仅仅是 **类型安全**. 对象值的不变性(Immutability)是与此完全不同的另一个问题.

声明处的类型变异(Declaration-site variance)

假设我们有一个泛型接口 `Source<T>` , 其中不存在任何接受 `T` 作为参数的方法, 仅有返回值为 `T` 的方法:

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

那么, 完全可以在 `Source<Object>` 类型的变量中保存一个 `Source<String>` 类型的实例 – 因为不存在对消费者方法的调用. 但 Java 不能理解这一点, 因此仍然禁止以下代码:

```
// Java
void demo(Source<String> strs) {
    Source<Object> objects = strs; // !!! 在 Java 中禁止这样的操作
    // ...
}
```

为了解决这个问题, 我们不得不将对象类型声明为 `Source<? extends Object>` , 其实是毫无意义的, 因为我们在这样修改之后, 我们所能调用的方法与修改之前其实是完全一样的, 因此, 使用这样复杂的类型声明并未带来什么好处. 但编译器并不理解这一点.

在 Kotlin 中, 我们有办法将这种情况告诉编译器. 这种技术称为 **声明处的类型变异(declaration-site variance)**: 我们可以对 `Source` 的 **类型参数** `T` 添加注解, 来确保 `Source<T>` 的成员函数只会 **返回** (生产) `T` 类型, 而绝不会消费 `T` 类型. 为了实现这个目的, 我们可以对 `T` 添加 **out** 修饰符:

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

fun demo(strs: Source<String>) {
    val objects: Source<Any> = strs // 这是 OK 的, 因为 T 是一个 out 类型参数
    // ...
}
```

一般规则是: 当 `C` 类的类型参数 `T` 声明为 **out** 时, 那么在 `C` 的成员函数中, `T` 类型只允许出现在 **输出** 位置, 这样的限制带来的回报就是, `C<Base>` 可以安全地用作 `C<Derived>` 的父类型.

用”高级术语”来说, 我们将 `C` 类称为, 在类型参数 `T` 上 **协变的(covariant)**, 或者说 `T` 是一个 **协变的(covariant)** 类型参数. 你可以将 `C` 类看作 `T` 类型对象的 **生产者**, 而不是 `T` 类型对象的 **消费者**.

out 修饰符称为 **协变注解(variance annotation)**, 而且, 由于这个注解出现在类型参数的声明处, 因此我们称之为 **声明处的类型变异(declaration-site variance)**. 这种方案与 Java 中的 **使用处类型变异(use-site variance)** 刚好相反, 在 Java 中, 是类型使用处的通配符产生了类型的协变.

除了 **out** 之外, Kotlin 还提供了另一种类型变异注解: **in**. 这个注解导致类型参数 **反向类型变异(contravariant)**: 这个类型将只能被消费, 而不能被生产. 反向类型变异的一个很好的例子是 `Comparable` :

```

abstract class Comparable<in T> {
    abstract fun compareTo(other: T): Int
}

fun demo(x: Comparable<Number>) {
    x.compareTo(1.0) // 1.0 类型为 Double, 是 Number 的子类型
    // 因此, 我们可以将 x 赋值给 Comparable<Double> 类型的变量
    val y: Comparable<Double> = x // OK!
}

```

我们认为 **in** 和 **out** 关键字的意义是十分明显的(同样的关键字已经在 C# 中经常使用了), 因此, 前面提到的记忆口诀也没有必要了, 为了一种崇高的理念, 我们可以将它改写一下:

存在主义 变形法则: 消费者进去, 生产者出来! :-)

译注: 上面两句翻译得不够好, 待校

类型投射(Type projection)

使用处的类型变异(Use-site variance): 类型投射(Type projection)

将声明类型参数 **T** 声明为 **out**, 就可以在使用时将它子类化, 这是十分方便的. 当我们的类 **能够** 局限为仅仅只返回 **T** 类型值的时候, 的确如此, 但如果不能呢? 关于这个问题, 一个很好的例子是 **Array** 类:

```

class Array<T>(val size: Int) {
    fun get(index: Int): T { /* ... */ }
    fun set(index: Int, value: T) { /* ... */ }
}

```

这个类对于类型参数 **T** 既不能协变, 也不能反向协变. 这就带来很大的不便. 我们来看看下面的函数:

```

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

```

这个函数应该将元素从一个 **Array** 复制到另一个 **Array**. 我们来试试使用一下这个函数:

```

val ints: Array<Int> = arrayOf(1, 2, 3)
val any = Array<Any>(3)
copy(ints, any) // 错误: 期待的参数类型是 (Array<Any>, Array<Any>)

```

在这里, 我们又遇到了熟悉的老问题: `Array<T>` 对于类型参数 `T` 是 **不可变的**, 因此 `Array<Int>` 和 `Array<Any>` 谁也不是谁的子类型. 为什么会这样? 原因与以前一样, 因为 `copy` 函数 **有可能** 会做一些不安全的操作, 也就是说, 这个函数可能会试图向 `from` 数组中 **写入**, 比如说, 一个 `String`, 这时假如我们传入的实际参数是一个 `Int` 的数组, 就会导致一个 `ClassCastException`.

所以, 我们需要确保的就是 `copy()` 函数不会做这类不安全的操作. 我们希望禁止这个函数向 `from` 数组写入数据, 我们可以这样声明:

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

这种声明在 Kotlin 中称为 **类型投射(type projection)**: 我们声明的含义是, `from` 不是一个单纯的数组, 而是一个被限制(投射)的数组: 我们只能对这个数组调用那些返回值为类型参数 `T` 的方法, 在这个例子中, 我们只能调用 `get()` 方法. 这就是我们实现 **使用处的类型变异(use-site variance)** 的方案, 与 Java 的 `Array<? extends Object>` 相同, 但略为简单一些.

你也可以使用 `in` 关键字来投射一个类型:

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

`Array<in String>` 与 Java 的 `Array<? super String>` 相同, 也就是说, 你可以使用 `CharSequence` 数组, 或者 `Object` 数组作为 `fill()` 函数的参数.

星号投射(Star-projection)

有些时候, 你可能想表示你并不知道类型参数的任何信息, 但是仍然希望能够安全地使用它. 这里所谓”安全地使用”是指, 对泛型类型定义一个类型投射, 要求这个泛型类型的所有的实体实例, 都是这个投射的子类型.

对于这个问题, Kotlin 提供了一种语法, 称为 **星号投射(star-projection)**:

- 假如类型定义为 `Foo<out T>`, 其中 `T` 是一个协变的类型参数, 上界(upper bound)为 `TUpper`, `Foo<*>` 等价于 `Foo<out TUpper>`. 它表示, 当 `T` 未知时, 你可以安全地从 `Foo<*>` 中 **读取** `TUpper` 类型的值.
- 假如类型定义为 `Foo<in T>`, 其中 `T` 是一个反向协变的类型参数, `Foo<*>` 等价于 `Foo<in Nothing>`. 它表示, 当 `T` 未知时, 你不能安全地向 `Foo<*>` 写入任何东西.
- 假如类型定义为 `Foo<T>`, 其中 `T` 是一个协变的类型参数, 上界(upper bound)为 `TUpper`, 对于读取值的场合, `Foo<*>` 等价于 `Foo<out TUpper>`, 对于写入值的场合, 等价于 `Foo<in Nothing>`.

如果一个泛型类型中存在多个类型参数, 那么每个类型参数都可以单独的投射. 比如, 如果类型定义为 `interface Function<in T, out U>`, 那么可以出现以下几种星号投射:

- `Function<*, String>`, 代表 `Function<in Nothing, String>`;

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

注意: 星号投射与 Java 的原生类型(raw type)非常类似, 但可以安全使用.

泛型函数

不仅类可以有类型参数. 函数一样可以有类型参数. 类型参数放在函数名称之前:

```
fun <T> singletonList(item: T): List<T> {  
    // ...  
}  
  
fun <T> T.basicToString(): String { // 扩展函数  
    // ...  
}
```

如果在调用处明确地传入了类型参数, 那么类型参数应该放在函数名称 之后:

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

泛型约束(Generic constraint)

对于一个给定的类型参数, 所允许使用的类型, 可以通过 **泛型约束(generic constraint)** 来限制.

上界(Upper bound)

最常见的约束是 **上界(upper bound)**, 与 Java 中的 `extends` 关键字相同:

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

冒号之后指定的类型就是类型参数的 **上界(upper bound)**: 对于类型参数 `T`, 只允许使用 `Comparable<T>` 的子类型. 比如:

```
sort(listOf(1, 2, 3)) // 正确: Int 是 Comparable<Int> 的子类型  
sort(listOf(HashMap<Int, String>())) // 错误: HashMap<Int, String> 不是  
Comparable<HashMap<Int, String>> 的子类型
```

如果没有指定, 则默认使用的上界是 `Any?`. 在定义类型参数的尖括号内, 只允许定义唯一一个上界. 如果同一个类型参数需要指定多个上界, 这时就需要使用单独的 **where** 子句:


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

嵌套类(Nested Class)

类可以嵌套在另一个类之内:

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

内部类(Inner class)

类可以使用 `inner` 关键字来标记, 然后就可以访问外部类(outer class)的成员. 内部类会保存一个引用, 指向外部类的对象实例:

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

在内部类中使用 `this` 关键字会产生歧义, 关于如何消除这种歧义, 请参见 [带限定符的 this 表达式](#).

枚举类

枚举类最基本的用法, 就是实现类型安全的枚举值:

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

每个枚举常数都是一个对象. 枚举常数之间用逗号分隔.

初始化

由于每个枚举值都是枚举类的一个实例, 因此枚举值可以初始化:

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

匿名类

枚举常数也可以定义它自己的匿名类:

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

枚举常数的匿名类可以有各自的方法, 还可以覆盖基类的方法. 注意, 与 Java 中一样, 如果枚举类中定义了什么成员, 你需要用分号将枚举常数的定义与枚举类的成员定义分隔开.

使用枚举常数

与 Java 中一样, Kotlin 中的枚举类拥有编译器添加的方法, 可以列出枚举类中定义的所有枚举常数值, 可以通过枚举常数值的名称字符串得到对应的枚举常数值. 这些方法的签名如下(这里假设枚举类名称为 `EnumClass`):

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

如果给定的名称不能匹配枚举类中定义的任何一个枚举常数值, `valueOf()` 方法会抛出 `IllegalArgumentException` 异常.

每个枚举常数值都拥有属性, 可以取得它的名称, 以及它在枚举类中声明的顺序:

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

枚举常数值还实现了 [Comparable](#) 接口, 枚举常数值之间比较时, 会使用枚举常数值在枚举类中声明的顺序作为自己的大小顺序.

对象表达式(Object Expression)与对象声明(Object Declaration)

有时我们需要创建一个对象, 这个对象在某个类的基础上略做修改, 但又不希望仅仅为了这一点点修改就明确地声明一个新类. Java 通过 *匿名内部类(anonymous inner class)* 来解决这种问题. Kotlin 使用 *对象表达式(object expression)* 和 *对象声明(object declaration)*, 对这个概念略做了一点泛化.

对象表达式(Object expression)

要创建一个继承自某个类(或多个类)的匿名类的对象, 我们需要写这样的代码:

```
window.addMouseListener(object : MouseAdapter() {
    override fun mouseClicked(e: MouseEvent) {
        // ...
    }

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

如果某个基类有构造器, 那么必须向构造器传递适当的参数. 通过冒号之后的逗号分隔的类型列表, 可以指定多个基类:

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

interface B {...}

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

如果, 我们 “只需要对象”, 而不需要继承任何有价值的基类, 我们可以简单地写:

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

与 Java 的匿名内部类(anonymous inner class)类似, 对象表达式内的代码可以访问创建这个对象的代码范围内的变量. (与 Java 不同的是, 被访问的变量不需要被限制为 final 变量.)

```

fun countClicks(window: JComponent) {
    var clickCount = 0
    var enterCount = 0

    window.addMouseListener(object : MouseAdapter() {
        override fun mouseClicked(e: MouseEvent) {
            clickCount++
        }

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

```

对象声明(Object declaration)

[单例模式](#) 是一种非常有用的模式, Kotlin (继 Scala 之后) 可以非常便利地声明一个单例:

```

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

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

```

这样的代码称为一个 *对象声明(object declaration)*. 如果在 `object` 关键字之后指定了一个名称, 那么它就不再是 *对象表达式* 了. 我们不能再将它赋值给一个变量, 但我们可以通过它的名称来引用它. 这样的对象也可以指定基类:

```

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

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

```

注意: 对象声明不可以是局部的(也就是说, 不可以直接嵌套在函数之内), 但可以嵌套在另一个对象声明之内, 或者嵌套在另一个非内部类(non-inner class)之内.

同伴对象(Companion Object)

一个类内部的对象声明, 可以使用 `companion` 关键字标记为同伴对象:

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

我们可以直接使用类名称作为限定符来访问同伴对象的成员:

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

同伴对象的名称可以省略, 如果省略, 则会使用默认名称 `Companion` :

```
class MyClass {  
    companion object {  
    }  
}  
  
val x = MyClass.Companion
```

注意, 虽然同伴对象的成员看起来很像其他语言中的类的静态成员(static member), 但在运行时期, 这些成员仍然是真实对象的实例的成员, 它们与静态成员是不同的, 举例来说, 它还可以实现接口:

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

但是, 如果使用 `@JvmStatic` 注解, 你可以让同伴对象的成员在 JVM 上被编译为真正的静态方法(static method)和静态域(static field). 详情请参见 [与 Java 的互操作性](#).

对象表达式与对象声明在语义上的区别

对象表达式与对象声明在语义上存在一个重要的区别:

- 对象声明是 **延迟(lazily)** 初始化的, 只会在首次访问时才会初始化

— 对象表达式则会在使用处 **立即** 执行(并且初始化)

委托(Delegation)

类的委托(Class Delegation)

[委托模式](#) 已被实践证明为类继承模式之外的另一种很好的替代方案, Kotlin 直接支持委托模式, 因此你不必再为了实现委托模式而手动编写那些无聊的例行公事的代码(boilerplate code)了. 比如, `Derived` 类可以继承 `Base` 接口, 并将 `Base` 接口所有的 public 方法委托给一个指定的对象:

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

`Derived` 类声明的基类列表中的 `by` 子句表示, `b` 将被保存在 `Derived` 的对象实例内部, 而且编译器将会生成继承自 `Base` 接口的所有方法, 并将调用转发给 `b`.

委托属性(Delegated Property)

有许多非常具有共性的属性, 虽然我们可以在每个需要这些属性的类中手工地实现它们, 但是, 如果能够只实现一次, 然后将它放在库中, 供所有需要的类使用, 那将会好很多. 这样的例子包括:

- 延迟加载属性(lazy property): 属性值只在初次访问时才会计算,
- 可观察属性(observable property): 属性发生变化时, 可以向监听器发送通知,
- 将多个属性保存在一个 map 内, 而不是保存在多个独立的域内.

为了解决这些问题(以及其它问题), Kotlin 允许 *委托属性(delegated property)*:

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

委托属性的语法是: `val/var <property name>: <Type> by <expression>`. 其中 `by` 关键字之后的表达式就是 *委托*, 属性的 `get()` 方法(以及 `set()` 方法) 将被委托给这个对象的 `getValue()` 和 `setValue()` 方法. 属性委托不必实现任何接口, 但必须提供 `getValue()` 函数(对于 `var` 属性, 还需要 `setValue()` 函数). 示例:

```
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.")  
    }  
}
```

如果属性 `p` 委托给一个 `Delegate` 的实例, 那么当我们读取属性值时, 就会调用到 `Delegate` 的 `getValue()` 函数, 此时函数收到的第一个参数将是我们访问的属性 `p` 所属的对象实例, 第二个参数将是 `p` 属性本身的描述信息(比如, 你可以从这里得到属性名称). For example:

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

这段代码的打印结果将是:

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

类似的, 当我们向属性 `p` 赋值时, 将会调用到 `setValue()` 函数. 这个函数收到的前两个参数与 `getValue()` 函数相同, 第三个参数将是即将赋给属性的新值:

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

这段代码的打印结果将是:

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

属性委托的前提条件

下面我们总结一下对属性委托对象的要求.

对于一个 **只读** 属性 (也就是说, **val** 属性), 它的委托必须提供一个名为 `getValue` 的函数, 这个函数接受以下参数:

- receiver — 这个参数的类型必须与 *属性所属的类* 相同, 或者是它的基类(对于扩展属性 — 这个参数的类型必须与被扩展的类型相同, 或者是它的基类),
- metadata — 这个参数的类型必须是 `KProperty<*>`, 或者是它的基类,

这个函数的返回值类型必须与属性类型相同(或者是它的子类型).

对于一个 **值可变(mutable)** 属性(也就是说, **var** 属性), 除 `getValue` 函数之外, 它的委托还必须 *另外再* 提供一个名为 `setValue` 的函数, 这个函数接受以下参数:

- receiver — 与 `getValue()` 函数的参数相同,
- metadata — 与 `getValue()` 函数的参数相同,
- new value — 这个参数的类型必须与属性类型相同, 或者是它的基类.

`getValue()` 和 `setValue()` 函数可以是委托类的成员函数, 也可以是它的扩展函数. 如果你需要将属性委托给一个对象, 而这个对象本来没有提供这些函数, 这时使用扩展函数会更便利一些. 这两个函数都需要标记为 `operator`.

标准委托

Kotlin 标准库中提供了一些工厂方法, 可以实现几种很有用的委托.

延迟加载(Lazy)

`lazy()` 是一个函数, 接受一个 Lambda 表达式作为参数, 返回一个 `Lazy<T>` 类型的实例, 这个实例可以作为一个委托, 实现延迟加载属性(lazy property): 第一次调用 `get()` 时, 将会执行 `lazy()` 函数受到的 Lambda 表达式, 然后会记住这次执行的结果, 以后所有对 `get()` 的调用都只会简单地返回以前记住的结果.

```

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

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

```

默认情况下, 延迟加载属性(lazy property)的计算是 **同步的(synchronized)**: 属性值只会在唯一一个线程内计算, 然后所有线程都将得到同样的属性值. 如果委托的初始化计算不需要同步, 多个线程可以同时执行初始化计算, 那么可以向 `lazy()` 函数传入一个 `LazyThreadSafetyMode.PUBLICATION` 参数. 相反, 如果你确信初始化计算只可能发生在一个线程内, 那么可以使用 `LazyThreadSafetyMode.NONE` 模式, 这种模式不会保持线程同步, 因此不会带来这方面的性能损失.

可观察属性(Observable)

`Delegates.observable()` 函数接受两个参数: 第一个是初始化值, 第二个是属性值变化事件的响应器(handler). 每次我们向属性赋值时, 响应器(handler)都会被调用(在属性赋值处理完成 之后). 响应器收到三个参数: 被赋值的属性, 赋值前的旧属性值, 以及赋值后的新属性值:

```

import kotlin.properties.Delegates

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

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

```

上面的示例代码打印的结果将是:

```

<no name> -> first
first -> second

```

如果你希望能够拦截属性的赋值操作, 并且还能够 “否决” 赋值操作, 那么不要使用 `observable()` 函数, 而应该改用 `vetoable()` 函数. 传递给 `vetoable` 函数的事件响应器, 会在属性赋值处理执行 之前 被调用.

将多个属性保存在一个 map 内

有一种常见的使用场景是将多个属性的值保存在一个 map 之内. 在应用程序解析 JSON, 或者执行某些“动态(dynamic)”任务时, 经常会出现这样的需求. 这种情况下, 你可以使用 map 实例本身作为属性的委托.

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

上例中, 类的构造器接受一个 map 实例作为参数:

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

委托属性将从这个 map 中读取属性值(使用属性名称字符串作为 key 值):

```
println(user.name) // 打印结果为: "John Doe"  
println(user.age) // 打印结果为: 25
```

如果不用只读的 Map, 而改用值可变的 MutableMap, 那么也可以用作 var 属性的委托:

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

函数与 Lambda 表达式

函数

函数声明

Kotlin 中使用 `fun` 关键字定义函数:

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

函数使用

函数的调用使用传统的方式:

```
val result = double(2)
```

调用类的成员函数时, 使用点号标记法(dot notation):

```
Sample().foo() // 创建一个 Sample 类的实例, 然后调用这个实例的 foo 函数
```

中缀标记法(Infix notation)

也可以使用中缀标记法(infix notation)来调用函数, 但需要满足以下条件:

- 是成员函数, 或者是[扩展函数](#)
- 只有单个参数
- 使用 `infix` 关键字标记

```
// 为 Int 类型定义扩展函数
infix fun Int.shl(x: Int): Int {
    ...
}

// 使用中缀标记法调用扩展函数

1 shl 2

// 上面的语句等价于

1.shl(2)
```

参数

函数参数的定义使用 Pascal 标记法, 也就是, *name: type* 的格式. 多个参数之间使用逗号分隔. 每个参数都必须明确指定类型.

```
fun powerOf(number: Int, exponent: Int) {
    ...
}
```

默认参数

函数参数可以指定默认值, 当参数省略时, 就会使用默认值. 与其他语言相比, 这种功能使得我们可以减少大量的 重载(overload)函数定义.

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

参数默认值的定义方法, 在参数类型之后, 添加 = 和默认值.

命名参数

调用函数时, 可以通过参数名来指定参数. 当函数参数很多, 或者存在默认参数时, 指定参数名是一种非常便利的功能.

比如, 对于下面这个函数:

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

我们可以使用默认参数来调用它:

```
reformat(str)
```

但是, 如果需要使用非默认的参数值调用它, 那么代码会成为这样:

```
reformat(str, true, true, false, ' ')
```

如果使用命名参数, 我们的代码可读性可以变得更高:

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

而且, 如果我们不需要指定所有的参数, 那么可以这样:

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

注意, 调用 Java 函数时, 不能使用这种命名参数语法, 因为 Java 字节码并不一定保留了函数参数的名称信息.

返回值为 Unit 的函数

如果一个函数不返回任何有意义的结果值, 那么它的返回类型为 `Unit`. `Unit` 类型只有唯一的一个值 - `Unit`. 在函数中, 不需要明确地返回这个值:


```
fun printHello(name: String?): Unit {
    if (name != null)
        println("Hello ${name}")
    else
        println("Hi there!")
    // 这里可以写 `return Unit` 或者 `return`, 都是可选的
}
```

返回类型 `Unit` 的声明本身也是可选的. 上例中的代码等价于:

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

单表达式函数(Single-Expression function)

如果一个函数返回单个表达式, 那么大括号可以省略, 函数体可以直接写在 `=` 之后:

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

如果编译器可以推断出函数的返回值类型, 那么返回值的类型定义是 [可选的](#):

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

明确指定返回值类型

如果函数体为多行语句组成的代码段, 那么就必须明确指定返回值类型, 除非这个函数打算返回 `Unit`, [这时返回类型的声明可以省略](#). 对于多行语句组成的函数, Kotlin 不会推断其返回值类型, 因为这样的函数内部可能存在复杂的控制流, 而且返回值类型对于代码的读者来说并不是那么一目了然(有些时候, 甚至对于编译器来说也很难判定返回值类型).

不定数量参数(Varargs)

一个函数的一个参数 (通常是参数中的最后一个) 可以标记为 `vararg`:

```
fun <T> asList(vararg ts: T): List<T> {
    val result = ArrayList<T>()
    for (t in ts) // ts 是一个 Array
        result.add(t)
    return result
}
```

调用时, 可以向这个函数传递不定数量的参数:

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

在函数内部, 类型为 `T` 的 `vararg` 参数会被看作一个 `T` 类型的数组, 也就是说, 上例中的 `ts` 变量的类型为 `Array<out T>`。

只有一个参数可以标记为 `vararg`。如果 `vararg` 参数不是函数的最后一个参数, 那么对于 `vararg` 参数之后的其他参数, 可以使用命名参数语法来传递参数值, 或者, 如果参数类型是函数, 可以在括号之外传递一个 Lambda 表达式。

调用一个存在 `vararg` 参数的函数时, 我们可以逐个传递参数值, 比如, `asList(1, 2, 3)`, 或者, 如果我们已经有了一个数组, 希望将它的内容传递给函数, 我们可以使用 **展开(spread)** 操作符(在数组之前加一个 `*`):

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

函数的范围

在 Kotlin 中函数可以定义在源代码的顶级范围内(top level), 这就意味着, 你不必象在 Java, C# 或 Scala 等等语言中那样, 创建一个类来容纳这个函数, 除顶级函数之外, Kotlin 中的函数也可以定义为局部函数, 成员函数, 以及扩展函数。

局部函数

Kotlin 支持局部函数, 也就是, 嵌套在另一个函数内的函数:

```
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中的局部变量(也就是, 闭包), 因此, 在上面的例子中, `visited` 可以定义为一个局部变量:

```
fun dfs(graph: Graph) {
    val visited = HashSet<Vertex>()
    fun dfs(current: Vertex) {
        if (!visited.add(current)) return
        for (v in current.neighbors)
            dfs(v)
    }

    dfs(graph.vertices[0])
}
```

成员函数

成员函数是指定义在类或对象之内的函数：

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

对成员函数的调用使用点号标记法

```
Sample().foo() // 创建 Sample 类的实例, 并调用 foo 函数
```

关于类, 以及成员覆盖, 详情请参见 [类](#) 和 [继承](#)

泛型函数

函数可以带有泛型参数, 泛型参数通过函数名之前的尖括号来指定：

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

关于泛型函数, 详情请参见 [泛型](#)

内联函数(Inline Function)

内联函数的详细解释在 [这里](#)

扩展函数

扩展函数的详细解释在 [单独的章节](#)

高阶函数(Higher-Order Function) 与 Lambda 表达式

高阶函数(Higher-Order Function) 与 Lambda 表达式的详细解释在 [单独的章节](#)

尾递归函数(Tail recursive function)

Kotlin 支持一种称为 [尾递归\(tail recursion\)](#) 的函数式编程方式. 这种方式使得某些本来需要使用循环来实现的算法, 可以改用递归函数来实现, 但同时不会存在栈溢出(stack overflow)的风险. 当一个函数标记为 `tailrec`, 并且满足要求的形式, 编译器就会对代码进行优化, 消除函数的递归调用, 产生一段基于循环实现的, 快速而且高效的代码.

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

上面的代码计算余弦函数的不动点(fixpoint), 结果应该是一个数学上的常数. 这个函数只是简单地从 1.0 开始不断地重复地调用 `Math.cos` 函数, 直到计算结果不再变化为止, 计算结果将是 0.7390851332151607. 编译器优化产生的代码等价于下面这种传统方式编写的代码:

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

要符合 `tailrec` 修饰符的要求, 函数必须在它执行的所有操作的最后一步, 递归调用它自身. 如果在这个递归调用之后还存在其他代码, 那么你不能使用尾递归, 而且你不能将尾递归用在 `try/catch/finally` 结构内. 尾递归目前只能用在 JVM 环境内.

高阶函数与 Lambda 表达式

高阶函数(Higher-Order Function)

高阶函数(higher-order function)是一种特殊的函数, 它接受函数作为参数, 或者返回一个函数. 这种函数的一个很好的例子就是 `lock()` 函数, 它的参数是一个锁对象(lock object), 以及另一个函数, 它首先获取锁, 运行对象函数, 然后再释放锁:

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

我们来分析一下上面的代码: `body` 参数是一个 [函数类型](#): `() -> T`, 因此它应该是一个函数, 没有参数, 返回一个 `T` 类型的值. `body` 函数在 `try` 块内被调用, 被 `lock` 锁保护住, 它的执行结果被 `lock()` 函数当作自己的结果返回.

如果我们要调用 `lock()` 函数, 我们需要将另一个函数传递给它作为参数(参见 [函数引用](#)):

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

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

另一种更常用的便捷方式是传递一个 [Lambda 表达式](#) 作为参数:

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

Lambda 表达式的详细介绍请参见 [后面的章节](#), 但为了继续本章的内容, 我们在这里做一点简单的介绍:

- Lambda 表达式用大括号括起,
- 它的参数(如果存在的话)定义在 `->` 之前 (参数类型可以省略),
- (如果存在 `->` 的话)函数体定义在 `->` 之后.

在 Kotlin 中有一种约定, 如果调用一个函数时, 最后一个参数是另一个函数, 那么这个参数可以写在括号之外:

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

高阶函数的另一个例子是 `map()` :

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

这个函数可以象这样调用:

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

注意, 调用函数时, 如果 Lambda 表达式是唯一的一个参数, 那么整个括号都可以省略.

另一个有用的约定是, 如果一个函数数字面值(function literal)只有唯一一个参数, 那么这个参数的声明可以省略(-> 也可以一起省略), 参数声明省略后, 将使用默认名称 `it` :

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

有了这样的约定, 我们就可以写出 [LINQ 风格](#) 的代码:

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

内联函数(Inline Function)

有些时候, 使用 [内联函数](#) 可以提高高阶函数的性能.

Lambda 表达式与匿名函数(Anonymous Function)

Lambda 表达式, 或者匿名函数, 是一种”函数数字面值(function literal)”, 也就是, 一个没有声明的函数, 但是立即作为表达式传递出去. 我们来看看下面的代码:

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

函数 `max` 是一个高阶函数, 也就是说, 它接受一个函数值作为第二个参数. 第二个参数是一个表达式, 本身又是另一个函数, 也就是说, 它是一个函数数字面量. 作为函数, 它等价于:

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

函数类型(Function Type)

对于接受另一个函数作为自己参数的函数, 我们必须针对这个参数指定一个函数类型. 比如, 前面提到的 `max` 函数, 它的定义如下:

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

参数 `less` 的类型是 `(T, T) -> Boolean`, 也就是, 它是一个函数, 接受两个 `T` 类型参数, 并且返回一个 `Boolean` 类型结果: 如果第一个参数小于第二个参数, 则返回 `true`, 否则返回 `false`.

在函数体, 第 4 行, `less` 被作为一个函数来使用: 这里调用了它, 传递给它两个 `T` 类型的参数.

函数类型的定义可以写作上面例子中那样, 如果你希望为各个参数编写文档, 解释其含义, 那么也可以指定参数名称.

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

Lambda 表达式的语法

Lambda 表达式的完整语法形式, 也就是, 函数类型的字面值, 如下:

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

Lambda 表达式包含在大括号之内, 在完整语法形式中, 参数声明在圆括号之内, 参数类型的声明可选, 函数体在 `->` 符号之后. 如果我们把所有可选的内容都去掉, 那么剩余的部分如下:

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

很多情况下 Lambda 表达式只有唯一一个参数. 如果 Kotlin 能够自行判断出 Lambda 表达式的参数定义, 那么它将允许我们省略唯一一个参数的定义, 并且会为我们隐含地定义这个参数, 使用的参数名为 `it`:

```
ints.filter { it > 0 } // 这个函数字面值的类型是 '(it: Int) -> Boolean'
```

注意, 如果一个函数接受另一个函数作为它的最后一个参数, 那么 Lambda 表达式作为参数时, 可以写在圆括号之外. 详细的语法请参见 [后缀调用](#).

匿名函数(Anonymous Function)

上面讲到的 Lambda 表达式语法, 还遗漏了一点, 就是可以指定函数的返回值类型. 大多数情况下, 不需要指定函数类型, 因为可以自动推断得到. 但是, 如果的确需要明确指定返回值类型, 你可以选择另一种语法: *匿名函数(anonymous function)*.

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

匿名函数看起来与通常的函数声明很类似, 区别在于省略了函数名. 函数体可以是一个表达式(如上例), 也可以是多条语句组成的代码段:

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

参数和返回值类型的声明与通常的函数一样, 但如果参数类型可以通过上下文推断得到, 那么类型声明可以省略:

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

对于匿名函数, 返回值类型的自动推断方式与通常的函数一样: 如果函数体是一个表达式, 那么返回值类型可以自动推断得到, 如果函数体是多条语句组成的代码段, 则返回值类型必须明确指定(否则被认为是 `Unit`).

注意, 匿名函数参数一定要在圆括号内传递. 允许将函数类型参数写在圆括号之外语法, 仅对 Lambda 表达式有效.

Lambda 表达式与匿名函数之间的另一个区别是, 它们的 [非局部返回\(non-local return\)](#) 的行为不同. 不使用标签的 `return` 语句总是从 `fun` 关键字定义的函数中返回. 也就是说, Lambda 表达式内的 `return` 将会从包含这个 Lambda 表达式的函数中返回, 而匿名函数内的 `return` 只会从匿名函数本身返回.

闭包(Closure)

Lambda 表达式, 匿名函数 (此外还有 [局部函数](#), [对象表达式](#)) 可以访问它的 *闭包*, 也就是, 定义在外层范围中的变量. 与 Java 不同, 闭包中捕获的变量是可以修改的(译注: Java 中必须为 `final` 变量):

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

带有接受者的函数字面值

Kotlin 提供了一种能力, 调用一个函数数字面值时, 可以指定一个 *接收者对象(receiver object)*. 在这个函数数字面值的函数体内部, 你可以调用接收者对象的方法, 而不必指定任何限定符. 这种能力与扩展函数很类似, 在扩展函数的函数体中, 你也可以访问接收者对象的成员. 这种功能最重要的例子之一就是 [类型安全的 Groovy 风格的生成器\(builder\)](#).

这样的函数数字面值, 它的类型是带接受者的函数类型:

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

这样的函数数字面值, 可以象接受者对象上的方法一样调用:

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

匿名函数语法允许你直接指定函数数字面值的接受者类型. 如果你需要声明一个带接受者的函数类型变量, 然后再将来的某个地方使用它, 那么这种功能就很有用.

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

如果接受者类型可以通过上下文自动推断得到, 那么 Lambda 表达式也可以用做带接受者的函数数字面值.

```
class HTML {
    fun body() { ... }
}

fun html(init: HTML.() -> Unit): HTML {
    val html = HTML() // 创建接受者对象
    html.init()       // 将接受者对象传递给 Lambda 表达式
    return html
}

html { // 带接受者的 Lambda 表达式从这里开始
    body() // 调用接受者对象上的一个方法
}
```

内联函数(Inline Function)

使用 [高阶函数](#) 在运行时会带来一些不利: 每个函数都是一个对象, 而且它还要捕获一个闭包, 也就是, 在函数体内部访问的那些外层变量. 内存占用(函数对象和类都会占用内存) 以及虚方法调用都会带来运行时的消耗.

但在很多情况下, 通过将 Lambda 表达式内联在使用处, 可以消除这些运行时消耗. 上文中的那些函数就是很好的例子. 也就是说, `lock()` 函数可以很容易地内联在调用处. 看看下面的例子:

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

编译器可以直接产生下面的代码, 而不必为参数创建函数对象, 然后再调用这个参数指向的函数:

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

这不就是我们最初期望的东西吗?

为了让编译器做到这点, 我们需要使用 `inline` 修饰符标记 `lock()` 函数:

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

`inline` 修饰符既会影响到函数本身, 也影响到传递给它的 Lambda 表达式: 这两者都会被内联到调用处.

函数内联也许会导致编译产生的代码尺寸变大, 但如果我们使用合理(不要内联太大的函数), 可以换来性能的提高, 尤其是在循环内发生的 “megamorphic” 函数调用. (译注: megamorphic 意义不明)

noinline

如果一个内联函数的参数中有多个 Lambda 表达式, 而你只希望内联其中的一部分, 你可以对函数的一部分参数添加 `noinline` 标记:

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

可内联的 Lambda 表达式只能在内联函数内部调用, 或者再作为可内联的参数传递给其他函数, 但 `noinline` 的 Lambda 表达式可以按照我们喜欢的方式任意使用: 可以保存在域内, 也可以当作参数传递, 等等.

注意. 如果一个内联函数不存在可以内联的函数类型参数, 而且没有 [实体化的类型参数](#), 编译器将会产生一个警告, 因为将这样的函数内联不太可能带来任何益处(如果你确信需要内联, 可以关闭这个警告).

非局部返回(Non-local return)

在 Kotlin 中, 使用无限定符的通常的 `return` 语句, 只能用来退出一个有名称的函数, 或匿名函数. 这就意味着, 要退出一个 Lambda 表达式, 我们必须使用一个 [标签](#), 无标签的 `return` 在 Lambda 表达式内是禁止使用的, 因为 Lambda 表达式不允许强制包含它的函数返回:

```
fun foo() {
    ordinaryFunction {
        return // 错误: 这里不允许让 `foo` 函数返回
    }
}
```

但是, 如果 Lambda 表达式被传递去的函数是内联函数, 那么 `return` 语句也可以内联, 因此 `return` 是允许的:

```
fun foo() {
    inlineFunction {
        return // OK: 这里的 Lambda 表达式是内联的
    }
}
```

这样的 `return` 语句(位于 Lambda 表达式内部, 但是退出包含 Lambda 表达式的函数)成为 *非局部(non-local)* 返回. 我们在循环中经常用到这样的结构, 而循环也常常就是包含内联函数的地方:

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

注意, 有些内联函数可能并不在自己的函数体内直接调用传递给它的 Lambda 表达式参数, 而是通过另一个执行环境来调用, 比如通过一个局部对象, 或者一个嵌套函数. 这种情况下, 在 Lambda 表达式内, 非局部的控制流同样是禁止的. 为了标识这一点, Lambda 表达式参数需要添加 `crossinline` 修饰符:

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

在内联的 Lambda 表达式中目前还不能使用 break 和 continue, 但我们计划将来支持它们

实体化的类型参数(Reified type parameter)

有些时候我们需要访问作为参数传递来的类型:

```
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
}
```

这里, 我们向上遍历一颗树, 然后使用反射来检查节点是不是某个特定的类型. 这些都没问题, 但这个函数的调用代码不太漂亮:

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

我们真正需要的, 只是简单地将一个类型传递给这个函数, 也就是说, 象这样调用它:

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

为了达到这个目的, 内联函数支持 *实体化的类型参数(reified type parameter)*, 使用这个功能我们可以将代码写成:

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

我们给类型参数添加了 `reified` 修饰符, 现在, 它可以在函数内部访问了, 就好象它是一个普通的类一样. 由于函数是内联的, 因此不必使用反射, 通常的操作符, 比如 `!is` 和 `as` 都可以正常工作了. 此外, 我们可以象前面提到的那样来调用这个函数: `myTree.findParentOfType<MyTreeNodeType>()`.

虽然很多情况下并不需要, 但我们仍然可以对一个实体化的类型参数使用反射:

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

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

通常的函数(没有使用 `inline` 标记的) 不能够使用实体化的类型参数. 一个没有运行时表现的类型(比如, 一个没有实体化的类型参数, 或者一个虚拟类型, 比如 `Nothing`) 不可以用作实体化的类型参数.

关于实体化类型参数的更底层的介绍, 请参见 [规格文档](#).

其他

解构声明(Destructuring Declaration)

有些时候, 能够将一个对象 *解构(destructure)* 为多个变量, 将会很方便, 比如:

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

这种语法称为 *解构声明(destructuring declaration)*. 一个解构声明会一次性创建多个变量. 上例中我们声明了两个变量: `name` 和 `age`, 并且可以独立地使用这两个变量:

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

解构声明在编译时将被分解为以下代码:

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

这里的 `component1()` 和 `component2()` 函数是 Kotlin 中广泛使用的 *约定原则(principle of convention)* 的又一个例子(其它例子请参见 `+` 和 `*` 操作符, `for` 循环, 等等.). 任何东西都可以作为解构声明右侧的被解构值, 只要可以对它调用足够数量的组件函数(component function). 当然, 还可以存在 `component3()` 和 `component4()` 等等.

注意, `componentN()` 函数需要标记为 `operator`, 才可以在解构声明中使用.

解构声明还可以使用在 `for` 循环中: 当我们说:

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

上面的代码将遍历集合中的所有元素, 然后对各个元素调用 `component1()` 和 `component2()` 函数, 变量 `a` 和 `b` 将得到 `component1()` 和 `component2()` 函数的返回值.

示例: 从一个函数返回两个值

举例来说, 假如我们需要从一个函数返回两个值. 比如, 一个是结果对象, 另一个是某种状态值. 在 Kotlin 中有一种紧凑的方法实现这个功能, 我们可以声明一个 [数据类](#), 然后返回这个数据类的一个实例:

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

    return Result(result, status)
}

// 现在, 可以这样使用这个函数:
val (result, status) = function(...)
```

由于数据类会自动声明 `componentN()` 函数, 因此可以在这里使用解构声明.

注意: 我们也可以使用标准库中的 `Pair` 类, 让上例中的 `function()` 函数返回一个 `Pair<Int, Status>` 实例. 但是, 给你的数据恰当地命名, 通常是一种更好的设计.

示例: 解构声明与 Map

遍历一个 map 的最好的方式可能就是:

```
for ((key, value) in map) {
    // 使用 key 和 value 执行某种操作
}
```

为了让上面的代码正确运行, 我们应该:

- 实现 `iterator()` 函数, 使得 map 成为多个值构成的序列,
- 实现 `component1()` 和 `component2()` 函数, 使得 map 内的每个元素成为一对值.

Kotlin 的标准库也的确实现了这些扩展函数:

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

因此, 你可以在对 map 的 `for` 循环中自由地使用解构声明(也可以在对数据类集合的 `for` 循环中使用解构声明).

集合(Collection)

与很多其他语言不同, Kotlin 明确地区分可变的和不可变的集合(list, set, map, 等等). 明确地控制集合什么时候可变什么时候不可变, 对于消除 bug 是很有帮助的, 也有助于设计出良好的 API.

理解可变集合的只读 *视图(view)* 与一个不可变的集合之间的差别, 是很重要的. 这两者都可以很容易地创建, 但类型系统无法区分这二者的差别, 因此记住哪个集合可变哪个不可变, 这是你自己的责任.

Kotlin 的 `List<out T>` 类型是一个只读的接口, 它提供的操作包括 `size`, `get` 等等. 与 Java 一样, 这个接口继承自 `Collection<T>`, `Collection<T>` 又继承自 `Iterable<T>`. 能够修改 List 内容的方法是由 `MutableList<T>` 接口添加的. 对于 `Set<out T>/MutableSet<T>` 以及 `Map<K, out V>/MutableMap<K, V>` 也是同样的模式.

通过下面的例子, 我们可以看看 list 和 set 类型的基本用法:

```
val numbers: MutableList<Int> = mutableListOf(1, 2, 3)
val readOnlyView: List<Int> = numbers
println(numbers)      // 打印结果为: "[1, 2, 3]"
numbers.add(4)
println(readOnlyView) // 打印结果为: "[1, 2, 3, 4]"
readOnlyView.clear()  // -> 无法编译

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

Kotlin 没有专门的语法用来创建 list 和 set. 你可以使用标准库中的方法, 比如 `listOf()`, `mutableListOf()`, `setOf()`, `mutableSetOf()`. 在并不极端关注性能的情况下, 创建 map 可以使用一个简单的 [惯用法](#): `mapOf(a to b, c to d)`

注意, `readOnlyView` 变量指向的其实是同一个 list 实例, 因此它的内容会随着后端 list 一同变化. 如果指向 list 的只有唯一一个引用, 而且这个引用是只读的, 那么我们可以这个集合完全是不可变的. 创建一个这样的集合的简单办法如下:

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

目前, `listOf` 方法是使用 array list 实现的, 但在将来, 这个方法会返回一个内存效率更高的, 完全不可变的集合类型, 以便尽量利用集合内容不可变这个前提.

注意, 只读集合类型是 [协变的\(covariant\)](#). 也就是说, 假设 `Rectangle` 继承自 `Shape`, 你可以将一个 `List<Rectangle>` 类型的值赋给一个 `List<Shape>` 类型变量. 但这对于可变的集合类型是不允许的, 因为可能导致运行时错误.

有时候, 你希望向调用者返回集合在某个时刻的一个快照, 而且这个快照保证不会变化:


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

`toList` 扩展方法只是单纯地复制 list 内的元素, 因此, 返回的 list 内容可以确保不会变化.

list 和 set 还有一些有用的扩展方法, 值得我们熟悉一下:

```
val items = listOf(1, 2, 3, 4)
items.first == 1
items.last == 4
items.filter { it % 2 == 0 } // 返回值为: [2, 4]
rwList.requireNotNulls()
if (rwList.none { it > 6 }) println("No items above 6")
val item = rwList.firstOrNull()
```

... 此外还有你所期望的各种工具方法, 比如 sort, zip, fold, reduce 等等.

Map 也遵循相同的模式. 可以很容易地创建和访问, 比如:

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

值范围(Range)

值范围表达式使用 `rangeTo` 函数来构造, 这个函数的操作符形式是 `..`, 另外还有两个相关操作符 `in` 和 `!in`. 任何可比较大小的数据类型(`comparable type`)都可以定义值范围, 但对于整数性的基本类型, 值范围的实现进行了特殊的优化. 下面是使用值范围的一些示例:

```
if (i in 1..10) { // 等价于: 1 <= i && i <= 10
    println(i)
}
```

整数性的值范围(`IntRange`, `LongRange`, `CharRange`) 还有一种额外的功能: 可以对这些值范围进行遍历. 编译器会负责将这些代码变换为 Java 中基于下标的 `for` 循环, 不会产生不必要的性能损耗.

```
for (i in 1..4) print(i) // 打印结果为: "1234"

for (i in 4..1) print(i) // 没有打印结果
```

如果你需要按反序遍历整数, 应该怎么办? 很简单. 你可以使用标准库中的 `downTo()` 函数:

```
for (i in 4 downTo 1) print(i) // 打印结果为: "4321"
```

可不可以使用 1 以外的任意步长来遍历整数? 没问题, `step()` 函数可以帮你实现:

```
for (i in 1..4 step 2) print(i) // 打印结果为: "13"

for (i in 4 downTo 1 step 2) print(i) // 打印结果为: "42"
```

值范围的工作原理

值范围实现了标准库中的一个共通接口: `ClosedRange<T>`.

`ClosedRange<T>` 表示数学上的一个闭区间(`closed interval`), 由可比较大小的数据类型(`comparable type`)构成. 这个区间包括两个端点: `start` 和 `endInclusive`, 这两个端点的值都包含在值范围内. 主要的操作是 `contains`, 主要通过 `in`/`!in` 操作符的形式来调用.

整数性类型的数列(`IntProgression`, `LongProgression`, `CharProgression`) 代表算术上的一个整数数列. 数列由 `first` 元素, `last` 元素, 以及一个非 0 的 `increment` 来定义. 第一个元素就是 `first`, 后续的所有元素等于前一个元素加上 `increment`. 除非数列为空, 否则遍历数列时一定会到达 `last` 元素.

数列是 `Iterable<N>` 的子类型, 这里的 `N` 分别代表 `Int`, `Long` 和 `Char`, 因此数列可以用在 `for` 循环内, 还可以用于 `map` 函数, `filter` 函数, 等等. 在 `Progression` 上的遍历等价于 Java/JavaScript 中基于下标的 `for` 循环:

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

对于整数性类型, `..` 操作符将会创建一个实现了 `ClosedRange<T>` 和 `*Progression` 接口的对象. 比如, `IntRange` 实现了 `ClosedRange<Int>`, 并继承 `IntProgression` 类, 因此 `IntProgression` 上定义的所有操作对于 `IntRange` 都有效. `downTo()` 和 `step()` 函数的结果永远是一个 `*Progression`.

要构造一个数列, 可以使用对应的类的同伴对象中定义的 `fromClosedRange` 函数:

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

数列的 `last` 元素会自动计算, 对于 `increment` 为正数的情况, 会求得一个不大于 `end` 的最大值, 对于 `increment` 为负数的情况, 会求得一个不小于 `end` 的最小值, 并且使得 `(last - first) % increment == 0`.

工具函数

`rangeTo()`

整数性类型上定义的 `rangeTo()` 操作符只是简单地调用 `*Range` 类的构造器, 比如:

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

浮点型数值(`Double`, `Float`) 没有定义自己的 `rangeTo` 操作符, 而是使用标准库为共通的 `Comparable` 类型提供的操作符:

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

这个函数返回的值范围不能用来遍历.

`downTo()`

`downTo()` 扩展函数可用于一对整数类型值, 下面是两个例子:

```

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

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

```

reversed()

对每个 `*Progression` 类都定义了 `reversed()` 扩展函数, 所有这些函数都会返回相反的数列.

```

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

```

step()

对每个 `*Progression` 类都定义了 `step()` 扩展函数, 所有这些函数都会返回使用新 `step` 值(由函数参数指定)的数列. 步长值参数要求永远是正数, 因此这个函数不会改变数列遍历的方向.

```

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

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

```

注意, 函数返回的数列的 `last` 值可能会与原始数列的 `last` 值不同, 这是为了保证 `(last - first) % increment == 0` 原则. 下面是一个例子:

```

(1..12 step 2).last == 11 // 数列中的元素为 [1, 3, 5, 7, 9, 11]
(1..12 step 3).last == 10 // 数列中的元素为 [1, 4, 7, 10]
(1..12 step 4).last == 9  // 数列中的元素为 [1, 5, 9]

```

类型检查与类型转换

is 与 !is 操作符

我们可以使用 `is` 操作符, 在运行时检查一个对象与一个给定的类型是否一致, 或者使用与它相反的 `!is` 操作符:

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

智能类型转换

很多情况下, 在 Kotlin 中你不必使用显式的类型转换操作, 因为编译器会对不可变的值追踪 `is` 检查, 然后在需要的时候自动插入(安全的)类型转换:

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

如果一个相反的类型检查导致了 `return`, 此时编译器足够智能, 可以判断出转换处理是安全的:

```
if (x !is String) return  
print(x.length) // x 被自动转换为 String 类型
```

在 `&&` 和 `||` 操作符的右侧也是如此:

```
// 在 `||` 的右侧, x 被自动转换为 String 类型  
if (x !is String || x.length == 0) return  
  
// 在 `&&` 的右侧, x 被自动转换为 String 类型  
if (x is String && x.length > 0)  
    print(x.length) // x 被自动转换为 String 类型
```

这种 *智能类型转换*(*smart cast*)对于 [when 表达式](#) 和 [while 循环](#) 同样有效:

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

注意, 在类型检查语句与变量使用语句之间, 假如编译器无法确保变量不会改变, 此时智能类型转换是无效的. 更具体地说, 必须满足以下条件时, 智能类型转换才有效:

- 局部的 **val** 变量 - 永远有效;
- **val** 属性 - 如果属性是 `private` 的, 或 `internal` 的, 或者类型检查处理与属性定义出现在同一个模块 (module) 内, 那么智能类型转换是有效的. 对于 `open` 属性, 或存在自定义 `get` 方法的属性, 智能类型转换是无效的;
- 局部的 **var** 变量 - 如果在类型检查语句与变量使用语句之间, 变量没有被改变, 而且它没有被 Lambda 表达式捕获并在 Lambda 表达式内修改它, 那么智能类型转换是有效的;
- **var** 属性 - 永远无效(因为其他代码随时可能改变变量值).

“不安全的” 类型转换操作符

如果类型转换不成功, 类型转换操作符通常会抛出一个异常. 因此, 我们称之为 *不安全的(unsafe)*. 在 Kotlin 中, 不安全的类型转换使用中缀操作符 **as** (参见 [操作符优先顺序](#)):

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

注意 `null` 不能被转换为 `String`, 因为这个类型不是 [可为 null 的\(nullable\)](#), 也就是说, 如果 `y` 为 `null`, 上例中的代码将抛出一个异常. 为了实现与 Java 相同的类型转换, 我们需要在类型转换操作符的右侧使用可为 `null` 的类型, 比如:

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

“安全的” (nullable) 类型转换操作

为了避免抛出异常, 你可以使用 *安全的* 类型转换操作符 **as?**, 当类型转换失败时, 它会返回 `null`:

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

注意, 尽管 **as?** 操作符的右侧是一个非 `null` 的 `String` 类型, 但这个转换操作的结果仍然是可为 `null` 的.

this 表达式

为了表示当前函数的 *接收者(receiver)*, 我们使用 `this` 表达式:

- 在 [类](#) 的成员函数中, `this` 指向这个类的当前对象实例
- 在 [扩展函数](#) 中, 或 [带接收者的函数面值\(function literal\)](#) 中, `this` 代表调用函数时, 在点号左侧传递的 *接收者* 参数.

如果 `this` 没有限定符, 那么它指向 *包含当前代码的最内层范围*. 如果想要指向其他范围内的 `this`, 需要使用 *标签限定符*:

带限定符的 `this`

为了访问更外层范围(比如 [类](#), 或 [扩展函数](#), 或有标签的 [带接受者的函数面值](#))内的 `this`, 我们使用 `this@label`, 其中的 `@label` 是一个 [标签](#), 代表我们想要访问的 `this` 所属的范围:

```
class A { // 隐含的标签 @A
  inner class B { // 隐含的标签 @B
    fun Int.foo() { // 隐含的标签 @foo
      val a = this@A // 指向 A 的 this
      val b = this@B // 指向 B 的 this

      val c = this // 指向 foo() 函数的接受者, 一个 Int 值
      val c1 = this@foo // 指向 foo() 函数的接受者, 一个 Int 值

      val funLit = lambda@ fun String.() {
        val d = this // 指向 funLit 的接受者
      }

      val funLit2 = { s: String ->
        // 指向 foo() 函数的接受者, 因为包含当前代码的 Lambda 表达式没有接受者
        val d1 = this
      }
    }
  }
}
```

相等判断

在 Kotlin 中存在两种相等判断:

- 引用相等 (两个引用指向同一个对象)
- 结构相等 (`equals()` 判断)

引用相等

引用相等使用 `===` 操作 (以及它的相反操作 `!==`) 来判断. 当, 且仅当, `a` 与 `b` 指向同一个对象时, `a === b` 结果为 `true`.

结构相等

结构相等使用 `==` 操作 (以及它的相反操作 `!=`) 来判断. 按照约定, `a == b` 这样的表达式将被转换为:

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

也就是说, 如果 `a` 不为 `null`, 将会调用 `equals(Any?)` 函数, 否则(也就是 `a` 为 `null`) 将会检查 `b` 是否指向 `null`.

注意, 当明确地与 `null` 进行比较时, 没有必要优化代码: `a == null` 将会自动转换为 `a === null`.

操作符重载(Operator overloading)

Kotlin 允许我们对数据类型的一组预定义的操作符提供实现函数。这些操作符的表达符号是固定(比如 `+` 或 `*`)，[优先顺序](#)也是固定的。要实现这些操作符，我们需要对相应的数据类型实现一个固定名称的 [成员函数](#) 或 [扩展函数](#)，这里所谓“相应的数据类型”，对于二元操作符，是指左侧操作数的类型，对于一元操作符，是指唯一一个操作数的类型。用于实现操作符重载的函数应该使用 `operator` 修饰符进行标记。

约定

下面我们介绍与各种操作符的重载相关的约定。

一元操作符

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


上表告诉我们说，当编译器处理一元操作符时，比如表达式 `+a`，它将执行以下步骤：

- 确定 `a` 的类型，假设为 `T`。
- 查找带有 `operator` 修饰符，无参数的 `unaryPlus()` 函数，而且函数的接受者类型为 `T`，也就是说，`T` 类型的成员函数或扩展函数。
- 如果这个函数不存在，或者找到多个，则认为是编译错误。
- 如果这个函数存在，并且返回值类型为 `R`，则表达式 `+a` 的类型为 `R`。

注意，这些操作符，以其其它所有操作符，都对 [基本类型](#) 进行了优化，因此不会发生函数调用，并由此产生性能损耗。

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

这些操作符应该改变它们的接受者，并且返回一个值(可选)。

 **`inc()/dec()` 不应该改变接受者对象的值。**

上文所谓“改变它们的接受者”，我们指的是改变 *接受者变量*，而不是改变接受者对象的值。

对于 *后缀形式*操作符，比如 `a++`，编译器解析时将执行以下步骤：

- 确定 `a` 的类型，假设为 `T`。
- 查找带有 `operator` 修饰符，无参数的 `inc()` 函数，而且函数的接受者类型为 `T`。

— 如果这个函数返回值类型为 `R`, 那么它必须是 `T` 的子类型.

计算这个表达式所造成的影响是:

- 将 `a` 的初始值保存到临时变量 `a0` 中,
- 将 `a.inc()` 的结果赋值给 `a`,
- 返回 `a0`, 作为表达式的计算结果值.

对于 `a--`, 计算步骤完全类似.

对于 前缀形式的操作符 `++a` 和 `--a`, 解析过程是一样的, 计算表达式所造成的影响是:

- 将 `a.inc()` 的结果赋值给 `a`,
- 返回 `a` 的新值, 作为表达式的计算结果值.

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>

对于上表中的操作符, 编译器只是简单地解析 翻译为 列中的表达式.

表达式	翻译为
<code>a in b</code>	<code>b.contains(a)</code>
<code>a !in b</code>	<code>!b.contains(a)</code>

对于 `in` 和 `!in` 操作符, 解析过程也是一样的, 但参数顺序被反转了.

符号	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>

方括号被翻译为, 使用适当个数的参数, 对 `get` 和 `set` 函数的调用.

符号	翻译为
<code>a()</code>	<code>a.invoke()</code>
<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>

圆括号被翻译为, 使用适当个数的参数, 调用 `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>
<code>a /= b</code>	<code>a.divAssign(b)</code>
<code>a %= b</code>	<code>a.modAssign(b)</code>

对于赋值操作符, 比如 `a += b`, 编译器执行以下步骤:

- 如果上表中右列的函数可用
 - 如果对应的二元操作函数(也就是说, 对于 `plusAssign()` 来说, `plus()` 函数) 也可用, 报告错误(歧义).
 - 确认函数的返回值类型为 `Unit`, 否则报告错误.
 - 生成 `a.plusAssign(b)` 的代码
- 否则, 尝试生成 `a = a + b` 的代码(这里包括类型检查: `a + b` 的类型必须是 `a` 的子类型).

注意: 在 Kotlin 中, 赋值操作 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 `!==` (同一性检查) 操作符不允许重载, 因此对这两个操作符不存在约定

`==` 操作符是特殊的: 它被翻译为一个复杂的表达式, 其中包括对 `null` 值的判断, 而且, `null == null` 的判断结果为 `true`.

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

所有的比较操作符都被翻译为对 `compareTo` 函数的调用, 这个函数的返回值必须是 `Int` 类型.

对命名函数的中缀式调用

使用 [中缀式函数调用](#), 我们可以模拟自定义的中缀操作符.

Null 值安全性

可为 null 的类型与不可为 null 的类型

Kotlin 类型系统的设计目标就是希望消除代码中 null 引用带来的危险, 也就是所谓的 [造成十亿美元损失的大错误](#).

在许多编程语言(包括 Java)中, 最常见的陷阱之一就是, 对一个指向 null 值的对象访问它的成员, 导致一个 null 引用异常. 在 Java 中就是 `NullPointerException`, 简称 NPE.

Kotlin 的类型系统致力于从我们的代码中消除 `NullPointerException`. 只有以下情况可能导致 NPE:

- 明确调用 `throw NullPointerException()`
- 使用 `!!` 操作符, 详情见后文
- 外部的 Java 代码导致这个异常
- 初始化过程中存在某些数据不一致 (在构造器中使用了未初始化的 `this`)

在 Kotlin 中, 类型系统明确区分可以指向 `null` 的引用 (可为 null 引用) 与不可以指向 `null` 的引用 (非 null 引用). 比如, 一个通常的 `String` 类型变量不可以指向 `null`:

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

要允许 null 值, 我们可以将变量声明为可为 null 的字符串, 写作 `String?`:

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

现在, 假如你对 `a` 调用方法或访问属性, 可以确信不会产生 NPE, 因此你可以安全地编写以下代码:

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

但如果你要对 `b` 访问同样的属性, 就不是安全的, 编译器会报告错误:

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

但我们仍然需要访问这个属性, 对不对? 有以下几种方法可以实现.

在条件语句中进行 null 检查

首先, 你可以明确地检查 `b` 是否为 `null`, 然后对 `null` 和非 `null` 的两种情况分别处理:

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

编译器将会追踪你执行过的检查, 因此允许在 `if` 内访问 `length` 属性. 更复杂的条件也是支持的:

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

注意, 以上方案需要的前提是, `b` 的内容不可变(也就是说, 对于局部变量的情况, 在 `null` 值检查与变量使用之间, 要求这个局部变量没有被修改, 对于类属性的情况, 要求是一个使用后端域变量的 `val` 属性, 并且不允许被后代类覆盖), 因为, 假如没有这样的限制的话, `b` 就有可能在检查之后被修改为 `null` 值.

安全调用

第二个选择方案是使用安全调用操作符, 写作 `? :`

```
b?.length
```

如果 `b` 不是 `null`, 这个表达式将会返回 `b.length`, 否则返回 `null`. 这个表达式本身的类型为 `Int?`.

安全调用在链式调用的情况下非常有用. 比如, 假如雇员 Bob, 可能被派属某个部门 Department (也可能不属于任何部门), 这个部门可能存在另一个雇员担任部门主管, 那么, 为了取得 Bob 所属部门的主管的名字, (如果存在的话), 我们可以编写下面的代码:

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

这样的链式调用, 只要属性链中任何一个属性为 `null`, 整个表达式就会返回 `null`.

Elvis 操作符

假设我们有一个可为 `null` 的引用 `r`, 我们可以用说, “如果 `r` 不为 `null`, 那么就使用它, 否则, 就使用某个非 `null` 的值 `x`”:

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

除了上例这种完整的 `if` 表达式之外, 还可以使用 Elvis 操作符来表达, 写作 `? :`

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

如果 `? :` 左侧的表达式值不是 `null`, Elvis 操作符就会返回它的值, 否则, 返回右侧表达式的值. 注意, 只有在左侧表达式值为 `null` 时, 才会计算右侧表达式.

注意, 由于在 Kotlin 中 `throw` 和 `return` 都是表达式, 因此它们也可以用在 Elvis 操作符的右侧. 这种用法可以带来很大的方便, 比如, 可以用来检查函数参数值是否合法:

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

!! 操作符

对于 NPE 的热爱者们来说, 还有第三个选择方案. 我们可以写 `b!!`, 对于 `b` 不为 `null` 的情况, 这个表达式将会返回这个非 `null` 的值(比如, 在我们的例子中就是一个 `String` 类型值), 如果 `b` 是 `null`, 这个表达式就会抛出一个 NPE:

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

所以, 如果你确实想要 NPE, 你可以抛出它, 但你必须明确地提出这个要求, 否则 NPE 不会在你没有注意的地方无声无息地出现.

安全的类型转换

如果对象不是我们期望的目标类型, 那么通常的类型转换就会导致 `ClassCastException`. 另一种选择是使用安全的类型转换, 如果转换不成功, 它将会返回 `null`:

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

异常(Exception)

异常类

Kotlin 中所有的异常类都是 `Throwable` 的后代类. 每个异常都带有一个错误消息, 调用堆栈, 以及可选的错误原因.

要抛出异常, 可以使用 `throw` 表达式:

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

要捕获异常, 可以使用 `try` 表达式:

```
try {  
    // 某些代码  
}  
catch (e: SomeException) {  
    // 异常处理  
}  
finally {  
    // 可选的 finally 代码段  
}
```

`try` 表达式中可以有 0 个或多个 `catch` 代码段. `finally` 代码段可以省略. 但是, `catch` 或 `finally` 代码段总计至少要出现一个.

`Try` 是一个表达式

`try` 是一个表达式, 也就是说, 它可以有返回值.

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

`try` 表达式的返回值, 要么是 `try` 代码段内最后一个表达式的值, 要么是 `catch` 代码段内最后一个表达式的值. `finally` 代码段的内容不会影响 `try` 表达式的结果值.

受控异常(Checked Exception)

Kotlin 中不存在受控异常(`checked exception`). 原因有很多, 我们举一个简单的例子.

下面的例子是 JDK 中 `StringBuilder` 类所实现的一个接口:

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


这个方法签名代表什么意思? 它说, 每次我想要将一个字符串追加到某个对象(比如, 一个 `StringBuilder`, 某种 `log`, 控制台, 等等), 我都必须要捕获 `IOException` 异常. 为什么? 因为这个对象有可能会执行 IO 操作(比如 `Writer` 类也会实现 `Appendable` 接口)… 因此就导致我们的程序中充满了这样的代码:

```
try {
    log.append(message)
}
catch (IOException e) {
    // 实际上前面的代码必然是安全的
}
```

这样的结果就很不好, 参见 [Effective Java](#), 第 65 条: *不要忽略异常*.

Bruce Eckel 在 [Java 需要受控异常吗?](#) 一文中说:

在小程序中的试验证明, 在方法定义中要求标明异常信息, 可以提高开发者的生产性, 同时提高代码质量, 但在大型软件中的经验则却指向一个不同的结论 – 生产性降低, 而代码质量改善不大, 或者根本没有改善.

另外还有其他一些这类讨论文章:

- [Java 的受控异常是一个错误](#) (Rod Waldhoff)
- [受控异常带来的问题](#) (Anders Hejlsberg)(译注, Borland Turbo Pascal 和 Delphi 的主要作者, 微软 .Net 概念的发起人之一, .Net 首席架构师)

与 Java 的互操作性

关于与 Java 的互操作性问题, 请参见 [与 Java 的互操作性](#) 中关于异常的小节.

注解(Annotation)

注解的声明

注解是用来为代码添加元数据(metadata)的一种手段. 要声明一个注解, 需要在类之前添加 `annotation` 修饰符:

```
annotation class Fancy
```

注解的其他属性, 可以通过向注解类添加元注解(meta-annotation)的方法来指定:

- `@Target` 指定这个注解可被用于哪些元素(类, 函数, 属性, 表达式, 等等.);
- `@Retention` 指定这个注解的信息是否被保存到编译后的 class 文件中, 以及在运行时是否可以通过反射访问到它(默认情况下, 这两个设定都是 true);
- `@Repeatable` 允许在单个元素上多次使用同一个注解;
- `@MustBeDocumented` 表示这个注解是公开 API 的一部分, 在自动产生的 API 文档的类或者函数签名中, 应该包含这个注解的信息.

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

使用

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

如果你需要对一个类的主构造器添加注解, 那么必须在构造器声明中添加 `constructor` 关键字, 然后在这个关键字之前添加注解:

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

也可以对属性的访问器函数添加注解:

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

构造器

注解可以拥有带参数的构造器.

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

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

允许使用的参数类型包括:

- 与 Java 基本类型对应的数据类型(Int, Long, 等等.);
- 字符串;
- 类 (Foo::class);
- 枚举;
- 其他注解;
- 由以上数据类型构成的数组.

如果一个注解被用作另一个注解的参数, 那么在它的名字之前不使用 @ 前缀:

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

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

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

Lambda 表达式

注解也可以用在 Lambda 上. 此时, Lambda 表达式的函数体内容将会生成一个 `invoke()` 方法, 注解将被添加到这个方法上. 这个功能对于 [Quasar](#) 这样的框架非常有用, 因为这个框架使用注解来进行并发控制.

```
annotation class Suspendable

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

注解的使用目标(Use-site Target)

当你为一个属性或一个主构造器的参数添加注解时, 从一个 Kotlin 元素会产生出多个 Java 元素, 因此在编译产生的 Java 字节码中, 你的注解存在多个可能的适用目标. 为了明确指定注解应该使用在哪个元素上, 可以使用以下语法:

```
class Example(@field:Ann val foo, // 对 Java 域变量添加注解
              @get:Ann val bar,   // 对属性的 Java get 方法添加注解
              @param:Ann val quux) // 对 Java 构造器参数添加注解
```

同样的语法也可以用来对整个源代码文件添加注解. 你可以添加一个目标为 `file` 的注解, 放在源代码文件的最顶端, `package` 指令之前, 如果这个源代码属于默认的包, 没有 `package` 指令, 则放在所有的 `import` 语句之前:

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

package org.jetbrains.demo
```

如果你有目标相同的多个注解, 那么可以在目标之后添加方括号, 然后将所有的注解放在方括号之内, 这样可以避免重复指定相同的目标:

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

Kotlin 支持的所有注解使用目标如下:

- `file`
- `property` (使用这个目标的注解, 在 Java 中无法访问)
- `field`
- `get` (属性的 `get` 方法)
- `set` (属性的 `set` 方法)
- `receiver` (扩展函数或扩展属性的接受者参数)
- `param` (构造器的参数)
- `setparam` (属性 `set` 方法的参数)
- `delegate` (保存代理属性的代理对象实例的域变量)

要对扩展函数的接受者参数添加注解, 请使用以下语法:

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

如果不指定注解的使用目标, 那么将会根据这个注解的 `@Target` 注解来自动选定使用目标. 如果存在多个可用的目标, 将会使用以下列表中的第一个:

- param
- property
- field

Java 注解

Kotlin 100% 兼容 Java 注解:

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

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

由于 Java 注解中没有定义参数的顺序, 因此不可以使用通常的函数调用语法来给注解传递参数. 相反, 你需要使用命名参数语法.

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

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

与 Java 一样, 有一个特殊情况就是 `value` 参数; 这个参数的值可以不使用明确的参数名来指定.

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

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

如果 Java 注解的 `value` 参数是数组类型, 那么在 Kotlin 中会变为 `vararg` 类型:

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

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

如果需要指定一个类作为注解的参数, 可以使用 Kotlin 类([KClass](#)). Kotlin 编译器会自动将它转换为 Java 类, 因此 Java 代码可以正常地访问到这个注解和它的参数.

```
import kotlin.reflect.KClass

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

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

Java 注解实例的值, 在 Kotlin 代码中可以通过属性的形式访问.

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

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

反射

反射是语言与库中的一组功能, 可以在运行时刻获取程序本身的信息. Kotlin 将函数和属性当作语言中的一等公民(first-class citizen), 而且, 通过反射获取它们的信息(也就是说, 在运行时刻得到一个函数或属性的名称和数据类型) 可以通过简单的函数式, 或交互式的编程方式实现.

⚠ 在 Java 平台上, 使用反射功能所需要的运行时组件是作为一个单独的 JAR 文件发布的(kotlin-reflect.jar). 这是为了对那些不使用反射功能的应用程序, 减少其运行库的大小. 如果你需要使用反射, 请注意将这个 .jar 文件添加到你的项目的 classpath 中.

类引用(Class Reference)

最基本的反射功能就是获取一个 Kotlin 类的运行时引用. 要得到一个静态的已知的 Kotlin 类的引用, 可以使用 *类字面值(class literal)* 语法:

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

类引用是一个 `KClass` 类型的值.

注意, Kotlin 的类引用不是一个 Java 的类引用. 要得到 Java 的类引用, 请使用 `KClass` 对象实例的 `.java` 属性.

函数引用(Function Reference)

假设我们有一个有名称的函数, 声明如下:

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

我们可以很容易地直接调用它(`isOdd(5)`), 但我们也可以将这个函数当作一个值来传递, 比如, 传给另一个函数作为参数. 为了实现这个功能, 我们使用 `::` 操作符:

```
val numbers = listOf(1, 2, 3)
println(numbers.filter(::isOdd)) // 打印结果为: [1, 3]
```

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

注意, 现在 `::` 操作符不能用于重载函数. 将来, 我们计划提供一种语法来指明函数的参数类型, 这样就可以在多个重载函数中选择我们希望引用的那一个.

如果我们需要使用一个类的成员函数, 或者一个扩展函数, 就必须使用限定符. 比如, `String::toCharArray` 指向 `String` 上的一个扩展函数, 函数类型为: `String() -> CharArray`.

示例: 函数组合

我们来看看下面的函数:

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

这个函数返回一个新的函数, 由它的两个参数代表的函数组合在一起构成: `compose(f, g) = f(g(*))` . 现在, 你可以使用可以执行的函数引用来调用这个函数:

```
fun length(s: String) = s.size  
  
val oddLength = compose(::isOdd, ::length)  
val strings = listOf("a", "ab", "abc")  
  
println(strings.filter(oddLength)) // 打印结果为: "[a, abc]"
```

属性引用(Property Reference)

在 Kotlin 中, 可以将属性作为一等对象来访问, 方法是使用 `::` 操作符:

```
var x = 1  
  
fun main(args: Array<String>) {  
    println(::x.get()) // 打印结果为: "1"  
    ::x.set(2)  
    println(x)         // 打印结果为: "2"  
}
```

表达式 `::x` 的计算结果是一个属性对象, 类型为 `KProperty<Int>`, 通过它 `get()` 方法可以得到属性值, 通过它的 `name` 属性可以得到属性名称. 详情请参见 [KProperty 类的 API 文档](#).

对于值可变的属性, 比如, `var y = 1`, `::y` 返回的属性对象的类型为 `KMutableProperty<Int>`, 它有一个 `set()` 方法.

属性引用可以用在所有使用无参函数的地方:

```
val str = listOf("a", "bc", "def")  
println(str.map(String::length)) // 打印结果为: [1, 2, 3]
```

要访问类的成员属性, 我们需要使用限定符:


```
class A(val p: Int)

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

与 Java 反射功能的互操作性

在 Java 平台上, Kotlin 的标准库包含了针对反射类的扩展函数, 这些反射类提供了与 Java 反射对象的相互转换功能(参见包 `kotlin.reflect.jvm`). 比如, 要查找一个 Kotlin 属性的后端域变量, 或者查找充当这个属性取值函数的 Java 方法, 你可以编写下面这样的代码:

```
import kotlin.reflect.jvm.*

class A(val p: Int)

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

要查找与一个 Java 类相对应的 Kotlin 类, 可以使用 `.kotlin` 扩展属性:

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

构造器引用(Constructor Reference)

与方法和属性一样, 也可以引用构造器. 构造器引用可以用于使用函数类型对象的地方, 但这个函数类型接受的参数应该与构造器相同, 返回值应该是构造器所属类的对象实例. 引用构造器使用 `::` 操作符, 再加上类名称. 我们来看看下面的函数, 它接受的参数是一个函数, 这个函数参数本身没有参数, 并返回 `Foo` 类型:

```
class Foo

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

使用 `::Foo` , 也就是 `Foo` 类的无参构造器的引用, 我们可以很简单地调用上面的函数:

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

类型安全的构建器(Type-Safe Builder)

[构建器\(builder\)](#) 的理念在 *Groovy* 开发社区非常流行. 使用构建器, 可以以一种半声明式的方式(semi-declarative way)来定义数据. 构建器非常适合于 [生成 XML](#), [控制 UI 组件布局](#), [描述 3D 场景](#), 等等等等...

在很多的使用场景下, Kotlin 可以创建一种 [类型检查](#) 的构建器, 这种功能使得 kotlin 中的构建器比 Groovy 自己的动态类型构建器更具吸引力.

如果无法使用带类型检查的构建器, Kotlin 也支持动态类型的构建器.

类型安全的构建器的示例

我们来看看以下代码:

```
import com.example.html.* // 具体的声明参见下文

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

            // 一个元素, 指定了属性, 还指定了其中的文本内容
            a(href = "http://kotlinlang.org") {+"Kotlin"}

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

            // 由程序生成的内容
            p {
                for (arg in args)
                    +arg
            }
        }
    }
```

上面是一段完全合法的 Kotlin 代码. 你可以在 [这个页面](#) 中在线验证这段代码(可以在浏览器中修改并运行它).

工作原理

我们来看看 Kotlin 中类型安全的构建器的实现机制. 首先, 我们要对我们想要构建的东西定义一组模型, 在这个示例中, 我们需要对 HTML 标签建模. 这个任务很简单, 只需要定义一组对象就可以了. 比如, `HTML` 是一个类, 负责描述 `<html>` 标签, 也就是说, 它可以定义子标签, 比如 `<head>` 和 `<body>`. (这个类的具体定义请参见[下文](#).)

现在, 回忆一下为什么我们可以写这样的代码:

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

`html` 实际上是一个函数调用, 它接受一个 [Lambda 表达式](#) 作为参数. 这个函数的定义如下:

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

这个函数只接受唯一一个参数, 名为 `init`, 这个参数本身又是一个函数, 其类型是 `HTML.() -> Unit`, 它是一个 *带接受者的函数类型*. 也就是说, 我们应该向这个函数传递一个 `HTML` 的实例(一个 *接收者*)作为参数, 而且在函数内, 我们可以调用这个实例的成员. 接受者可以通过 `this` 关键字来访问:

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

(`head` 和 `body` 是 `html` 类的成员函数.)

现在, `this` 关键字可以省略, 通常都是如此, 省略之后我们的代码就已经非常接近一个构建器了:

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

那么, 这个函数调用做了什么? 我们来看看上面定义的 `html` 函数体. 首先它创建了一个 `HTML` 类的新实例, 然后它调用通过参数得到的函数, 来初始化这个 `HTML` 实例 (在我们的示例中, 这个初始化函数对 `HTML` 实例调用了 `head` 和 `body` 方法), 然后, 这个函数返回这个 `HTML` 实例. 这正是构建器应该做的.

`HTML` 类中 `head` 和 `body` 函数的定义与 `html` 函数类似. 唯一的区别是, 这些函数会将自己创建的对象实例添加到自己所属的 `HTML` 实例的 `children` 集合中:

```

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

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

```

实际上这两个函数做的事情完全相同, 因此我们可以编写一个泛型化的函数, 名为 `initTag`:

```

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

```

然后, 这两个函数就变得很简单了:

```

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

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

```

现在我们可以使用这两个函数来构建 `<head>` 和 `<body>` 标签了.

还需要讨论的一个问题是, 我们如何在标签内部添加文本. 在上面的示例程序中, 我们写了这样的代码:

```

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

```

我们所作的, 仅仅只是将一个字符串放在一个标签之内, 但在字符串之前有一个小小的 `+`, 所以, 它是一个函数调用, 被调用的是前缀操作符函数 `unaryPlus()`. 这个操作符实际上是由扩展函数 `unaryPlus()` 定义的, 这个扩展函数是抽象类 `TagWithText` 的成员 (这个抽象类是 `Title` 类的祖先类):

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

所以, 前缀操作符 `+` 所作的, 是将一个字符串封装到 `TextElement` 的一个实例中, 然后将这个实例添加到 `children` 集合中, 然后这个字符串就会成为标签树中一个适当的部分。

以上所有类和函数都定义在 `com.example.html` 包中, 上面的构建器示例程序的最上部引入了这个包。在下一节中, 你可以读到这个包的完整定义。

com.example.html 包的完整定义

下面是 `com.example.html` 包的完整定义(但只包含上文示例程序使用到的元素)。它可以构建一个 HTML 树。这段代码大量使用了 [扩展函数](#) 和 [带接受者的 Lambda 表达式](#)。

```
package com.example.html

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

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

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

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

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

    private fun renderAttributes(): String? {
        val builder = StringBuilder()
    }
```

```

        for (a in attributes.keys) {
            builder.append(" $a=\"${attributes[a]}\"")
        }
        return builder.toString()
    }

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

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

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

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

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

class Title() : TagWithText("title")

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

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

class A() : BodyTag("a") {
    public var href: String

```

```
    get() = attributes["href"]!!  
    set(value) {  
        attributes["href"] = value  
    }  
}  
  
fun html(init: HTML() -> Unit): HTML {  
    val html = HTML()  
    html.init()  
    return html  
}
```


动态类型(Dynamic Type)

⚠️ 当编译目标平台为 JVM 时, 不支持动态类型

Kotlin 虽然是一种静态类型的语言, 但它仍然可以与无类型或松散类型的环境互操作, 比如各种 JavaScript 环境. 为了为这样的使用场景提供帮助, Kotlin 提供了 `dynamic` 类型:

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

简单来说, `dynamic` 类型关闭了 Kotlin 的类型检查:

- 这个类型的值可以赋值给任意变量, 也可以作为参数传递给任何函数,
- 任何值都可以复制给 `dynamic` 类型的变量, 也可以传递给函数的 `dynamic` 类型参数,
- 对这些值不做 `null` 检查.

`dynamic` 类型最特殊的功能是, 允许我们对 `dynamic` 类型变量访问它的 **任何** 属性, 还可以使用任意参数访问它的 **任何** 函数:

```
dyn.whatever(1, "foo", dyn) // 没有在任何地方定义过 'whatever'  
dyn.whatever(*arrayOf(1, 2, 3))
```

在 JavaScript 平台上, 这些代码会被”原封不动”地编译: Kotlin 代码中的 `dyn.whatever(1)`, 编译产生的 JavaScript 代码就是同样的 `dyn.whatever(1)`.

一个动态调用永远会返回一个 `dynamic` 的结果, 因此我们可以将这些调用自由地串联起来:

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

当我们向一个动态调用传递一个 Lambda 表达式作为参数时, Lambda 表达式的所有参数类型默认都是 `dynamic`:

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

关于更加深入的技术性介绍, 请参见 [规格文档](#).

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](#)

与 Java 的互操作性

在 Kotlin 中调用 Java 代码

Kotlin 的设计过程中就考虑到了与 Java 的互操作性. 在 Kotlin 中可以通过很自然的方式调用既有的 Java 代码, 反过来在 Java 中也可以很流畅地使用 Kotlin 代码. 本章中我们介绍在 Kotlin 中调用 Java 代码的一些细节问题.

大多数 Java 代码都可以直接使用, 没有任何问题:

```
import java.util.*

fun demo(source: List<Int>) {
    val list = ArrayList<Int>()
    // 对 Java 集合使用 'for' 循环:
    for (item in source)
        list.add(item)
    // 也可以对 Java 类使用 Kotlin 操作符:
    for (i in 0..source.size() - 1)
        list[i] = source[i] // 这里会调用 get 和 set 方法
}
```

Get 和 Set 方法

符合 Java 的 Get 和 Set 方法规约的方法(无参数, 名称以 `get` 开头, 或单个参数, 名称以 `set` 开头)在 Kotlin 中会被识别为属性. 比如:

```
import java.util.Calendar

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

注意, 如果 Java 类中只有 set 方法, 那么在 Kotlin 中不会被识别为属性, 因为 Kotlin 目前还不支持只写(set-only) 的属性.

返回值为 void 的方法

如果一个 Java 方法返回值为 void, 那么在 Kotlin 中调用时将返回 Unit . 如果, 在 Kotlin 中使用了返回值, 那么会由 Kotlin 编译器在调用处赋值, 因为返回值已经预先知道了(等于 Unit).

当 Java 标识符与 Kotlin 关键字重名时的转义处理

某些 Kotlin 关键字在 Java 中是合法的标识符: in, object, is, 等等. 如果 Java 类库中使用 Kotlin 的关键字作为方法名, 你仍然可以调用这个方法, 只要使用反引号(`)对方法名转义即可:

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

Null 值安全性与平台数据类型

Java 中的所有引用都可以为 null 值, 因此对于来自 Java 的对象, Kotlin 的严格的 null 值安全性要求就变得毫无意义了. Java 中定义的类型在 Kotlin 中会被特别处理, 被称为 *平台数据类型(platform type)*. 对于这些类型, Null 值检查会被放松, 因此对它们来说, 只提供与 Java 中相同的 null 值安全保证(详情参见[下文](#)).

我们来看看下面的例子:

```
val list = ArrayList<String>() // 非 null 值 (因为是构造器方法的返回结果)
list.add("Item")
val size = list.size() // 非 null 值 (因为基本类型 int)
val item = list[0] // 类型自动推断结果为平台类型 (通常的 Java 对象)
```

对于平台数据类型的变量, 当我们调用它的方法时, Kotlin 不会在编译时刻报告可能为 null 的错误, 但这个调用在运行时可能失败, 原因可能是发生 null 指针异常, 也可能是 Kotlin 编译时为防止 null 值错误而产生的断言, 在运行时导致失败:

```
item.substring(1) // 编译时允许这样的调用, 但在运行时如果 item == null 则可能抛出异常
```

平台数据类型是 *无法指示的(non-denotable)*, 也就是说不能在语言中明确指出这样的类型. 当平台数据类型的值赋值给 Kotlin 变量时, 我们可以依靠类型推断(这时变量的类型会被自动推断为平台数据类型, 比如上面示例程序中的变量 item 就是如此), 或者我们也可以选择我们期望的数据类型(可为 null 的类型和非 null 类型都允许):

```
val nullable: String? = item // 允许, 永远不会发生错误
val notNull: String = item // 允许, 但在运行时刻可能失败
```

如果我们选择使用非 null 类型, 那么编译器会在赋值处理之前输出一个断言(assertion). 它负责防止 Kotlin 中的非 null 变量指向一个 null 值. 当我们将平台数据类型的值传递给 Kotlin 函数的非 null 值参数时, 也会输出断言. 总之, 编译器会尽可能地防止 null 值错误在程序中扩散(然而, 有些时候由于泛型的存在, 不可能完全消除这种错误).

对平台数据类型的注解

上文中我们提到, 平台数据类型无法在程序中明确指出, 因此在 Kotlin 语言中没有专门的语法来表示这种类型. 然而, 有时编译器和 IDE 仍然需要表示这些类型(比如在错误消息中, 在参数信息中, 等等), 因此, 我们有一种助记用的注解:

- `T!` 代表 “`T` 或者 `T?`”,
- `(Mutable)Collection<T>!` 代表 “元素类型为 `T` 的 Java 集合, 内容可能可变, 也可能不可变, 值可能允许为 null, 也可能不允许为 null”,
- `Array<(out) T>!` 代表 “元素类型为 `T` (或 `T` 的子类型)的 Java 数组, 值可能允许为 null, 也可能不允许为 null”

可否为 null(Nullability) 注解

带有可否为 null(Nullability) 注解的 Java 类型在 Kotlin 中不会被当作平台数据类型, 而会被识别为可为 null 的, 或非 null 的 Kotlin 类型. 目前, 编译器支持 [JetBrains 风格的可否为 null 注解](#) (`org.jetbrains.annotations` 包中定义的 `@Nullable` 和 `@NotNull` 注解).

数据类型映射

Kotlin 会对某些 Java 类型进行特殊处理. 这些类型会被从 Java 中原封不动地装载进来, 但被 *映射* 为对应的 Kotlin 类型. 映射过程只会在编译时发生, 运行时的数据表达不会发生变化. Java 的基本数据类型会被映射为对应的 Kotlin 类型(但请注意 [平台数据类型](#) 问题):

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

有些内建类虽然不是基本类型, 也会被映射为对应的 Kotlin 类型:

Java 类型	Kotlin 类型
java.lang.Object	kotlin.Any!

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

集合类型在 Kotlin 中可能是只读的, 也可能是内容可变的, 因此 Java 的集合会被映射为以下类型(下表中所有的 Kotlin 类型都属于 `kotlin` 包):

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

Java 数据的映射如下, 详情参见 [下文](#):

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

在 Kotlin 中使用 Java 的泛型

Kotlin 的泛型与 Java 的泛型略有差异 (参见 [泛型](#)). 将 Java 类型导入 Kotlin 时, 我们进行以下变换:

— Java 的通配符会被变换为 Kotlin 的类型投射

— `Foo<? extends Bar>` 变换为 `Foo<out Bar>!`

— `Foo<? super Bar>` 变换为 `Foo<in Bar>!`

— Java 的原生类型(raw type) 转换为 Kotlin 的星号投射(star projection)

— `List` 变换为 `List<*>!`, 也就是 `List<out Any?>!`

与 Java 一样, Kotlin 的泛型信息在运行时不会保留, 也就是说, 创建对象时传递给构造器的类型参数信息, 在对象中不会保留下来, 所以, `ArrayList<Integer>()` 与 `ArrayList<Character>()` 在运行时刻是无法区分的. 这就导致无法进行带有泛型信息的 `is` 判断. Kotlin 只允许对星号投射(star projection)的泛型类型进行 `is` 判断:

```
if (a is List<Int>) // 错误: 无法判断它是不是 Int 构成的 List
// 但是
if (a is List<*>) // OK: 这里的判断不保证 List 内容的数据类型
```

Java 数组

与 Java 不同, Kotlin 中的数组是不可变的(invariant). 这就意味着, Kotlin 不允许我们将 `Array<String>` 赋值给 `Array<Any>`, 这样就可以避免发生运行时错误. 在调用 Kotlin 方法时, 如果参数声明为父类型的数组, 那么将子类型的数组传递给这个参数, 也是禁止的, 但对于 Java 的方法, 这是允许(通过使用 `Array<(out)String>!` 形式的[平台数据类型](#)).

在 Java 平台上, 会使用基本类型构成的数组, 以避免装箱(boxing)/拆箱(unboxing)操作带来的性能损失. 由于 Kotlin 会隐藏这些实现细节, 因此与 Java 代码交互时需要使用一个替代办法. 对于每种基本类型, 都存在一个专门的类(`IntArray`, `DoubleArray`, `CharArray`, 等等) 来解决这种问题. 这些类与 `Array` 类没有关系, 而且会被编译为 Java 的基本类型数组, 以便达到最好的性能.

假设有一个 Java 方法, 接受一个名为 `indices` 的参数, 类型是 `int` 数组:

```
public class JavaArrayExample {

    public void removeIndices(int[] indices) {
        // 方法代码在这里...
    }
}
```

为了向这个方法传递一个基本类型值构成的数组, 在 Kotlin 中你可以编写下面的代码:

```
val javaObj = JavaArrayExample()
val array = intArrayOf(0, 1, 2, 3)
javaObj.removeIndices(array) // 向方法传递 int[] 参数
```

编译输出 JVM 字节码时, 编译器会对数组的访问处理进行优化, 因此不会产生性能损失:

```
val array = arrayOf(1, 2, 3, 4)
array[x] = array[x] * 2 // 编译器不会产生对 get() 和 set() 方法的调用
for (x in array) // 不会创建迭代器(iterator)
    print(x)
```

即使我们使用下标来遍历数组, 也不会产生任何性能损失:

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

最后, `in` 判断也不会产生性能损失:

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

Java 的可变长参数(Varargs)

Java 类的方法声明有时会对 `indices` 使用可变长的参数定义(varargs).

```
public class JavaArrayExample {

    public void removeIndices(int... indices) {
        // 方法代码在这里...
    }
}
```

这种情况下, 为了将 `IntArray` 传递给这个参数, 需要使用展开(spread) `*` 操作符:

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

对于使用可变长参数的 Java 方法, 目前无法向它传递 `null` 参数.

操作符

由于 Java 中无法将方法标记为操作符重载方法, Kotlin 允许我们使用任何的 Java 方法, 只要方法名称和签名定义满足操作符重载的要求, 或者满足其他规约(`invoke()` 等等.) 使用中缀调用语法来调用 Java 方法是不允许的.

受控异常(Checked Exception)

在 Kotlin 中, 所有的异常都是不受控的(unchecked), 也就是说编译器不会强制要求你捕获任何异常. 因此, 当调用 Java 方法时, 如果这个方法声明了受控异常, Kotlin 不会要求你做任何处理:

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

Object 类的方法

当 Java 类型导入 Kotlin 时, 所有 `java.lang.Object` 类型的引用都会被转换为 `Any` 类型. 由于 `Any` 类与具体的实现平台无关, 因此它声明的成员方法只有 `toString()`, `hashCode()` 和 `equals()`, 所以, 为了补足 `java.lang.Object` 中的其他方法, Kotlin 使用了 [扩展函数](#).

wait()/notify()

[Effective Java](#) 第 69 条建议使用并发控制用的功能函数库, 而不要使用 `wait()` 和 `notify()` 方法. 因此, 对 `Any` 类型的引用不能使用这些方法. 如果你确实需要调用这些方法, 那么可以先将它变换为 `java.lang.Object` 类型:

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

getClass()

要得到一个对象的类型信息, 我们可以使用 `javaClass` 扩展属性.

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

对于类, 应该使用 `Foo::class.java`, 而不是 Java 中的 `Foo.class`.

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

clone()

要覆盖 `clone()` 方法, 你的类需要实现 `kotlin.Cloneable` 接口:

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

别忘了 [Effective Java](#), 第 11 条: *要正确地覆盖 clone 方法*.

finalize()

要覆盖 `finalize()` 方法, 你只需要声明它既可, 不必使用 `override` 关键字:

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

按照 Java 的规则, `finalize()` 不能是 `private` 方法.

继承 Java 的类

Kotlin 类的超类中, 最多只能指定一个 Java 类(Java 接口的数量没有限制).

访问静态成员(static member)

Java 类的静态成员(static member)构成这些类的” 同伴对象(companion object)”. 我们不能将这样的” 同伴对象” 当作值来传递, 但可以明确地访问它的成员, 比如:

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

Java 的反射

Java 的反射在 Kotlin 类中也可以使用, 反过来也是如此. 我们在上文中讲到, 你可以使用 `instance.javaClass` 或 `ClassName::class.java` 得到 `java.lang.Class`, 然后通过它就可以使用 Java 的反射功能.

此外还支持其他反射功能, 比如可以得到 Kotlin 属性对应的 Java get/set 方法或后端成员, 可以得到 Java 成员变量对应的 `KProperty`, 得到 `KFunction` 对应的 Java 方法或构造器, 或者反过来得到 Java 方法或构造器对应的 `KFunction`.

SAM 转换

与 Java 8 一样, Kotlin 支持 SAM(Single Abstract Method) 转换. 也就是说如果一个 Java 接口中仅有一个方法, 并且没有默认实现, 那么只要 Java 接口方法与 Kotlin 函数参数类型一致, Kotlin 的函数字面值就可以自动转换为这个接口的实现者.

你可以使用这个功能来创建 SAM 接口的实例:

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

…也可以用在方法调用中:

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

如果 Java 类中有多个同名的方法, 而且方法参数都可以接受函数式接口, 那么你可以使用一个适配器函数 (adapter function), 将 Lambda 表达式转换为某个具体的 SAM 类型, 然后就可以选择需要调用的方法. 编译器也会在需要的时候生成这些适配器函数.

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

注意, SAM 转换只对接口有效, 不能用于抽象类, 即使抽象类中仅有唯一一个抽象方法.

还应当注意, 这个功能只在 Kotlin 与 Java 互操作时有效; 由于 Kotlin 本身已经有了专门的函数类型, 因此没有必要将函数自动转换为 Kotlin 接口的实现者, Kotlin 也不支持这样的转换.

在 Kotlin 中使用 JNI(Java Native Interface)

要声明一个由本地代码(C 或者 C++)实现的函数, 你需要使用 `external` 修饰符标记这个函数:

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

剩下的工作与 Java 中完全相同.

在 Java 中调用 Kotlin

在 Java 中可以很容易地调用 Kotlin 代码.

属性

Property 的取值方法(getter)会被转换为 *get* 方法, 设值方法(setter)会被转换为 *set* 方法.

包级函数

在源代码文件 `example.kt` 的 `org.foo.bar` 包内声明的所有函数和属性, 都会被放在名为 `org.foo.bar.ExampleKt` 的 Java 类之内.

```
// example.kt
package demo

class Foo

fun bar() {
}
```

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

编译生成的 Java 类的名称, 可以通过 `@JvmName` 注解来改变:

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

package demo

class Foo

fun bar() {
}
```

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

如果多个源代码文件生成的 Java 类名相同(由于文件名和包名都相同, 或由于使用了相同的 `@JvmName` 注解)这样的情况通常会被认为是错误. 但是, 编译器可以使用指定的名称生成单个 Java Facade 类, 其中包含所有源代码文件的所有内容. 要生成这样的 Facade 类, 可以在多个源代码文件中使用 `@JvmMultifileClass` 注解.

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

package demo

fun foo() {
}
```

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

package demo

fun bar() {
}
```

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

实例的域

如果希望将一个 Kotlin 属性公开为 Java 中的一个域, 你需要对它添加 `@JvmField` 注解. 生成的域的可见度将与属性可见度一样. 要对属性使用 `@JvmField` 注解, 需要满足以下条件: 属性应该拥有后端域变量 (backing field), 不是 `private` 属性, 没有 `open`, `override` 或 `const` 修饰符, 并且不是委托属性 (delegated property).

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

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

[延迟初始化属性](#) 也会公开为 Java 中的域. 域的可见度将与属性的 `lateinit` 的设值方法可见度一样.

静态域

声明在命名对象(named object)或同伴对象(companion object)之内的 Kotlin 属性, 将会存在静态的后端域变量(backing field), 对于命名对象, 静态后端域变量存在于命名对象内, 对于同伴对象, 静态后端域变量存在在包含同伴对象的类之内.

通常这些静态的后端域变量是 private 的, 但可以使用以下方法来公开它:

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

如果对这样的属性添加 `@JvmField` 注解, 那么它的静态后端域变量可见度将会与属性本身的可见度一样.

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

```
// Java  
Key.COMPARATOR.compare(key1, key2);  
// 这里访问的是 Key 类中的 public static final 域
```

命名对象或同伴对象中的[延迟初始化属性](#)对应的静态的后端域变量, 其可见度将与属性的设值方法可见度一样.

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

```
// Java  
Singleton.provider = new Provider();  
// 这里访问的是 Singleton 类中的 public static 非 final 域
```

使用 `const` 修饰符的属性(无论定义在类中, 还是在顶级范围内(top level)) 会被转换为 Java 中的静态域:


```
// file example.kt

object Obj {
    const val CONST = 1
}

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

const val MAX = 239
```

在 Java 中:

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

静态方法

上文中我们提到, Kotlin 会为包级函数生成静态方法. 此外, 如果你对函数添加 `@JvmStatic` 注解, Kotlin 也可以为命名对象或同伴对象中定义的函数生成静态方法. 比如:

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

现在, `foo()` 在 Java 中是一个静态方法, 而 `bar()` 不是:

```
C.foo(); // 正确
C.bar(); // 错误: 不是静态方法
```

对命名对象也一样:

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

在 Java 中:

```
Obj.foo(); // 正确
Obj.bar(); // 错误
Obj.INSTANCE.bar(); // 正确, 这是对单体实例的一个方法调用
Obj.INSTANCE.foo(); // 也正确
```

`@JvmStatic` 注解也可以用于命名对象或同伴对象的属性, 可以使得属性的取值方法和设值方法变成静态方法, 对于命名对象, 这些静态方法在命名对象之内, 对于同伴对象, 这些静态方法在包含同伴对象的类之内.

使用 @JvmName 注解处理签名冲突

有时候我们在 Kotlin 中声明了一个函数, 但在 JVM 字节码中却需要一个不同的名称. 最显著的例子就是 *类型消除(type erasure)* 时的情况:

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

这两个函数是无法同时定义的, 因为它们产生的 JVM 代码的签名是完全相同的:

`filterValid(Ljava/util/List;)Ljava/util/List;`. 如果我们确实需要在 Kotlin 中给这两个函数定义相同的名称, 那么可以对其中一个(或两个)使用 `@JvmName` 注解, 通过这个注解的参数来指定一个不同的名称:

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

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

在 Kotlin 中, 可以使用相同的名称 `filterValid` 来访问这两个函数, 但在 Java 中函数名将是 `filterValid` 和 `filterValidInt`.

如果我们需要定义一个属性 `x`, 同时又定义一个函数 `getX()`, 这时也可以使用同样的技巧:

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

fun getX() = 10
```

重载函数的生成

通常, 如果在 Kotlin 中定义一个方法, 并指定了参数默认值, 这个方法在 Java 中只会存在带所有参数的版本. 如果你希望 Java 端的使用者看到不同参数的多个重载方法, 那么可以使用 `@JvmOverloads` 注解.

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

对于每个带有默认值的参数, 都会生成一个新的重载方法, 这个重载方法的签名将会删除这个参数, 以及右侧的所有参数. 上面的示例程序生成的 Java 方法如下:

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

这个注解也可以用于构造器, 静态方法, 等等. 但不能用于抽象方法, 包括定义在接口内的方法.

注意, 在 [次级构造器](#) 中介绍过, 如果一个类的构造器方法参数全部都指定了默认值, 那么会对这个类生成一个 public 的无参数构造器. 这个特性即使在没有使用 @JvmOverloads 注解时也是有效的.

受控异常(Checked Exception)

我们在上文中提到过, Kotlin 中不存在受控异常. 因此, Kotlin 函数在 Java 中的签名通常不会声明它抛出的异常. 因此, 假如我们有一个这样的 Kotlin 函数:

```
// example.kt
package demo

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

然后我们希望在 Java 中调用它, 并捕获异常:

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

这时 Java 编译器会报告错误, 因为 `foo()` 没有声明抛出 `IOException` 异常. 为了解决这个问题, 我们可以在 Kotlin 中使用 `@Throws` 注解:

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

Null值安全性

在 Java 中调用 Kotlin 函数时, 没有任何机制阻止我们向一个非 null 参数传递一个 null 值. 所以, Kotlin 编译时, 会对所有接受非 null 值参数的 public 方法产生一些运行时刻检查代码. 由于这些检查代码的存在, Java 端代码会立刻得到一个 `NullPointerException` 异常.

泛型的类型变异(Variant)

如果 Kotlin 类使用了 [声明处的类型变异\(declaration-site variance\)](#), 那么这些类在 Java 代码中看到的形式存在两种可能. 假设我们有下面这样的类, 以及两个使用这个类的函数:

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

interface Base
class Derived : Base

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

如果用最简单的方式转换为 Java 代码, 结果将是:

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

问题在于, 在 Kotlin 中我们可以这样: `unboxBase(boxDerived("s"))`, 但在 Java 中却不可以, 因为在 Java 中 `Box` 的类型参数 `T` 是 *不可变的(invariant)*, 因此 `Box<Derived>` 不是 `Box<Base>` 的子类型. 为了解决 Java 端的问题, 我们必须将 `unboxBase` 函数定义成这样:

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

这里我们使用了 Java 的 *通配符类型(wildcards type)* (`? extends Base`), 通过使用处类型变异(use-site variance)来模仿声明处的类型变异(declaration-site variance), 因为 Java 中只有使用处类型变异.

为了让 Kotlin 的 API 可以在 Java 中正常使用, 如果一个类 *被用作函数参数*, 那么对于定义了类型参数协变的 `Box` 类, `Box<Super>` 会生成为 Java 的 `Box<? extends Super>` (对于定义了类型参数反向协变的 `Foo` 类, 会生成为 Java 的 `Foo<? super Bar>`). 当类被用作返回值时, 编译产生的结果不会使用类型通配符, 否则 Java 端的使用者就不得不处理这些类型通配符(而且这是违反通常的 Java 编程风格的). 因此, 我们上面例子中的函数真正的输出结果是这样的:

```
// 返回值 - 没有类型通配符
Box<Derived> boxDerived(Derived value) { ... }

// 参数 - 有类型通配符
Base unboxBase(Box<? extends Base> box) { ... }
```

注意: 如果类型参数是 final 的, 那么生成类型通配符一般来说就没有意义了, 因此 `Box<String>` 永远是 `Box<String>`, 无论它出现在什么位置.

如果我们需要类型通配符, 但默认没有生成, 那么可以使用 `@JvmWildcard` 注解:

```
fun boxDerived(value: Derived): Box<@JvmWildcard Derived> = Box(value)
// 将被翻译为
// Box<? extends Derived> boxDerived(Derived value) { ... }
```

反过来, 如果默认生成了类型通配符, 但我们不需要它, 那么可以使用 `@JvmSuppressWildcards` 注解:

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

注意: `@JvmSuppressWildcards` 不仅可以用于单个的类型参数, 也可以用于整个函数声明或类声明, 这时它会使得这个函数或类之内的所有类型通配符都不产生.

Nothing 类型的翻译

`Nothing` 类型是很特殊的, 因为它在 Java 中没有对应的概念. 所有的 Java 引用类型, 包括 `java.lang.Void`, 都可以接受 `null` 作为它的值, 而 `Nothing` 甚至连 `null` 值都不能接受. 因此, 在 Java 的世界里无法准确地表达这个类型. 因此, Kotlin 会在使用 `Nothing` 类型参数的地方生成一个原生类型(raw type):

```
fun emptyList(): List<Nothing> = listOf()
// 将被翻译为
// List emptyList() { ... }
```

工具

为 Kotlin 代码编写文档

为 Kotlin 代码编写文档使用的语言 (相当于 Java 中的 JavaDoc) 称为 **KDoc**. 本质上, KDoc 结合了 JavaDoc 和 Markdown, 它在块标签(block tag)使用 JavaDoc 语法(但做了扩展, 以便支持 Kotlin 特有的概念), Markdown 则用来表示内联标记(inline markup).

生成文档

Kotlin 的文档生成工具叫做 [Dokka](#). 关于它的使用方法请阅读 [Dokka README](#).

Dokka 有 plugin 可用于 Gradle, Maven 以及 Ant 构建环境, 因此你可以将 Kotlin 代码的文档生成集成到你的构建过程之内.

KDoc 语法

与 JavaDoc 一样, KDoc 以 `/**` 开始, 以 `*/` 结束. 文档中的每一行以星号开始, 星号本身不会被当作文档内容.

按照通常的习惯, 文档的第一段(直到第一个空行之前的所有文字)是对象元素的概要说明, 之后的内容则是详细说明.

每个块标签(block tag)都应该放在新的一行内, 使用 `@` 字符起始.

下面的例子是使用 KDoc 对一个类标注的文档:

```

/**
 * 由多个 *成员* 构成的一个组.
 *
 * 这个类没有任何有用的逻辑; 只是一个文档的示例.
 *
 * @param T 组内成员的类型.
 * @property name 组的名称.
 * @constructor 创建一个空的组.
 */
class Group<T>(val name: String) {
    /**
     * 向组添加一个 [成员].
     * @return 添加之后的组大小.
     */
    fun add(member: T): Int { ... }
}

```

块标签(Block Tag)

KDoc 目前支持以下块标签:

@param <name>

对一个函数的参数, 或一个类的类型参数标注文档. 如果你希望的话, 为了更好地区分参数名与描述文本, 可以将参数名放在方括号内. 所以下面两种语法是等价的:

@param name 描述.
 @param[name] 描述.

@return

对函数的返回值标注文档.

@constructor

对类的主构造器(primary constructor)标注文档.

@property <name>

对类中指定名称的属性标注文档. 这个标签可以用来标注主构造器中定义的属性, 如果将文档放在主构造器的属性声明之前会很笨拙, 因此可以使用标签来对指定的属性标注文档.

@throws <class>, @exception <class>

对一个方法可能抛出的异常标注文档. 由于 Kotlin 中不存在受控异常(`checked exception`), 因此也并不要求对所有的异常标注文档, 但如果异常信息对类的使用者很有帮助的话, 你可以使用这个标签来标注异常信息.

`@sample <identifier>`

为了演示对象元素的使用方法, 可以使用这个标签将指定名称的函数体嵌入到文档内.

`@see <identifier>`

这个标签会在文档的 **See Also** 部分, 添加一个指向某个类或方法的链接.

`@author`


标识对象元素的作者.

`@since`

标识对象元素最初引入这个软件时的版本号.

`@suppress`

将对象元素排除在文档之外. 有些元素, 不属于模块的正式 API 的一部分, 但站在代码的角度又需要被外界访问, 对这样的元素可以使用这个标签.

 KDoc 不支持 `@deprecated` 标签. 请使用 `@Deprecated` 注解来代替.

内联标记(Inline Markup)

对于内联标记(`inline markup`), KDoc 使用通常的 [Markdown](#) 语法, 但添加了一种缩写语法来生成指向代码内其他元素的链接.

指向元素的链接

要生成指向其他元素(类, 方法, 属性, 或参数)的链接, 只需要简单地将它的名称放在方括号内:

请使用 `[foo]` 方法来实现这个目的.

如果你希望对链接指定一个标签, 请使用 Markdown 参照风格(`reference-style`)语法:

请使用 `[这个方法][foo]` 来实现这个目的.

在链接中也可以使用带限定符的元素名称. 注意, 与 `JavaDoc` 不同, 限定符的元素名称永远使用点号来分隔各个部分, 包括方法名称之前的分隔符, 也是点号:

请使用 `[kotlin.reflect.KClass.properties]` 来列举一个类的属性.

链接中的元素名称使用的解析规则, 与这个名称出现在对象元素之内时的解析规则一样. 具体来说, 如果你在当前源代码文件中导入(import)了一个名称, 那么在 KDoc 注释内使用它时, 就不必再指定完整的限定符了.

注意, KDoc 没有任何语法可以解析链接内出现的重载函数. 由于 Kotlin 的文档生成工具会将所有重载函数的文档放在同一个页面之内, 因此不必明确指定某一个具体的重载函数, 链接也可以正常工作.

使用 Maven

插件与版本

kotlin-maven-plugin 插件用来在 maven 环境中编译 Kotlin 源代码和模块. 目前只支持 Maven v3.

可以通过 *kotlin.version* 变量来指定你希望使用的 Kotlin 版本. 下表是 Kotlin 的 Release 版本名称与版本号之间的对应关系:

Release 版本名	版本号
1.0.1 hotfix update 2	1.0.1-2
1.0.1 hotfix update	1.0.1-1
1.0.1	1.0.1
1.0 GA	1.0.0
Release Candidate	1.0.0-rc-1036
Beta 4	1.0.0-beta-4589
Beta 3	1.0.0-beta-3595
Beta 2	1.0.0-beta-2423
Beta	1.0.0-beta-1103
Beta Candidate	1.0.0-beta-1038
M14	0.14.449
M13	0.13.1514
M12.1	0.12.613
M12	0.12.200
M11.1	0.11.91.1
M11	0.11.91
M10.1	0.10.195
M10	0.10.4
M9	0.9.66
M8	0.8.11
M7	0.7.270
M6.2	0.6.1673
M6.1	0.6.602
M6	0.6.69
M5.3	0.5.998

依赖

Kotlin 有一个内容广泛的标准库, 可以在你的应用程序中使用. 请在 pom 文件中添加以下依赖设置:

```

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

```

编译 Kotlin 源代码

要编译 Kotlin 源代码, 请在 标签内指定源代码目录:

```

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

```

编译源代码时, 需要引用 Kotlin Maven 插件:

```

<plugin>
  <artifactId>kotlin-maven-plugin</artifactId>
  <groupId>org.jetbrains.kotlin</groupId>
  <version>${kotlin.version}</version>

  <executions>
    <execution>
      <id>compile</id>
      <goals> <goal>compile</goal> </goals>
    </execution>

    <execution>
      <id>test-compile</id>
      <goals> <goal>test-compile</goal> </goals>
    </execution>
  </executions>
</plugin>

```

编译 Kotlin 和 Java 的混合源代码

要编译混合源代码的应用程序, 需要在 Java 编译器之前调用 Kotlin 编译器. 用 Maven 的术语来说就是, kotlin-maven-plugin 应该在 maven-compiler-plugin 之前运行.

为了达到这个目的, 可以将 Kotlin 编译动作移动到比 Java 编译当作更前的 process-sources 步骤(phase) (如果你有更好的解决方法, 欢迎告诉我们):

```
<plugin>
  <artifactId>kotlin-maven-plugin</artifactId>
  <groupId>org.jetbrains.kotlin</groupId>
  <version>${kotlin.version}</version>

  <executions>
    <execution>
      <id>compile</id>
      <phase>process-sources</phase>
      <goals> <goal>compile</goal> </goals>
    </execution>

    <execution>
      <id>test-compile</id>
      <phase>process-test-sources</phase>
      <goals> <goal>test-compile</goal> </goals>
    </execution>
  </executions>
</plugin>
```

OSGi

关于对 OSGi 的支持, 请参见 [Kotlin 与 OSGi](#).

示例

我们提供了一个 Maven 工程示例, 可以 [通过 GitHub 仓库下载](#).

使用 Ant

安装 Ant Task

Kotlin 提供了 3 个 Ant Task:

- `kotlinc`: 面向 JVM 的 Kotlin 编译器
- `kotlin2js`: 面向 JavaScript 的 Kotlin 编译器
- `withKotlin`: 使用标准的 `javac` Ant Task 来编译 Kotlin 代码

这些 Task 定义在 `kotlin-ant.jar` 库文件内, 这个库文件位于 [Kotlin 编译器](#) 的 `lib` 文件夹内

面向 JVM, 编译纯 Kotlin 代码

如果工程内只包含 Kotlin 源代码, 这种情况下最简单的编译方法是使用 `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>
```

这里的 `${kotlin.lib}` 指向 Kotlin standalone 编译器解压缩后的文件夹。

面向 JVM, 编译包含多个根目录的纯 Kotlin 代码

如果工程中包含多个源代码根目录, 可以使用 `src` 元素来定义源代码路径:

```
<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>
```

面向 JVM, 编译 Kotlin 和 Java 的混合代码

如果工程包含 Kotlin 和 Java 的混合代码, 这时尽管也能够使用 *kotlinc*, 但为了避免重复指定 Task 参数, 推荐使用 *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>
```

要对 `<withKotlin>` 指定额外的命令行参数, 可以使用内嵌的 `<compilerArg>` 参数. 运行 `kotlinc -help` 命令, 可以看到参数的完整列表. 还可以通过 `moduleName` 属性来指定被编译的模块名称:

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

面向 JavaScript, 编译单个源代码文件夹

```
<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>
```

面向 JavaScript, 使用 Prefix, PostFix 和 sourcemap 选项

```
<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>
```

面向 JavaScript, 编译单个源代码文件夹, 使用 metaInfo 选项

如果你希望将编译结果当作一个 Kotlin/JavaScript 库发布, 可以使用 `metaInfo` 选项. 如果 `metaInfo` 设为 `true`, 那么编译时会额外创建带二进制元数据(binary metadata)的 JS 文件. 这个文件需要与编译结果一起发布.

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

  <target name="build">
    <!-- 会创建 out.meta.js 文件, 其中包含二进制描述符(binary descriptor) -->
    <kotlin2js src="root1" output="out.js" metaInfo="true"/>
  </target>
</project>
```

参照

完整的 Ant Task 元素和属性一览表如下:

kotlinc 和 kotlin2js 的共通属性

名称	说明	是否必须	默认值
src	需要编译的 Kotlin 源代码文件或源代码目录	是	
nowarn	屏蔽编译时的警告信息	否	false
noStdlib	不要将 Kotlin 标准库包含在 classpath 内	否	false
failOnError	如果编译过程中检测到错误, 是否让整个构建过程失败	否	true

kotlinc 的属性

名称	说明	是否必须	默认值
output	编译输出的目标目录, 或目标 .jar 文件名	是	

名称	说明	是否必须	默认值
classpath	编译时的 class path 值	是	
classpathref	编译时的 class path 参照	否	
includeRuntime	当 output 是 .jar 文件时, 是否将 Kotlin 运行库包含在这个 jar 内	否	true
moduleName	被编译的模块名称	否	编译目标的名称(如果有指定), 或工程名称

kotlin2js 的属性

名称	说明	是否必须
output	编译输出的目标文件	是
library	库文件(kt, dir, jar)	否
outputPrefix	生成 JavaScript 文件时使用的前缀	否
outputSuffix	生成 JavaScript 文件时使用的后缀	否
sourcemap	是否生成 sourcemap 文件	否
metaInfo	是否生成带二进制描述符(binary descriptor)的元数据(metadata)文件	否
main	编译器是否生成对 main 函数的调用代码	否

使用 Gradle

要使用 Gradle 编译 Kotlin 代码, 你需要 [设置 `kotlin-gradle plugin`](#), 将它 [应用](#) 到你的工程, 然后 [添加 `kotlin-stdlib` 依赖](#). 在 IntelliJ IDEA 中, 在 Project action 内选择 Tools -> Kotlin -> Configure Kotlin 也可以自动完成这些操作.

Plugin 与版本 Versions

`kotlin-gradle-plugin` 可以用来编译 Kotlin 源代码和模块.

使用的 Kotlin 版本通常定义为 `kotlin_version` 属性:

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

    repositories {
        mavenCentral()
    }

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

下表是 Kotlin 的 Release 版本名称与版本号之间的对应关系:

Milestone	Version
1.0.1 hotfix update 2	1.0.1-2
1.0.1 hotfix update	1.0.1-1
1.0.1	1.0.1
1.0 GA	1.0.0
Release Candidate	1.0.0-rc-1036
Beta 4	1.0.0-beta-4589
Beta 3	1.0.0-beta-3595
Beta 2	1.0.0-beta-2423
Beta	1.0.0-beta-1103
Beta Candidate	1.0.0-beta-1038
M14	0.14.449
M13	0.13.1514
M12.1	0.12.613
M12	0.12.200
M11.1	0.11.91.1
M11	0.11.91

Milestone	Version
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

编译到 JVM 平台

要编译到 JVM 平台, 需要应用(apply) Kotlin plugin:

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

Kotlin 源代码可以与 Java 源代码共存在同一个文件夹下, 也可以放在不同的文件夹下. 默认的约定是使用不同的文件夹:

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

如果不使用默认约定的文件夹结构, 那么需要修改相应的 *sourceSets* 属性:

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

编译到 JavaScript

编译到 JavaScript 时, 需要应用(apply)另一个 plugin:

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

这个 plugin 只能编译 Kotlin 源代码文件, 因此推荐将 Kotlin 和 Java 源代码文件放在不同的文件夹内(如果工程内包含 Java 文件的话). 与编译到 JVM 平台时一样, 如果不使用默认约定的文件夹结构, 我们需要使用 *sourceSets* 来指定文件夹目录:

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

如果希望创建一个可重复使用的 JavaScript 库, 请使用 `kotlinOptions.metaInfo` 生成一个带二进制描述符 (binary descriptor) 的 JS 文件. 这个文件需要与编译结果一起发布.

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

编译到 Android

Android 的 Gradle 模型与通常的 Gradle 略有区别, 因此如果我们想要编译一个使用 Kotlin 语言开发的 Android 工程, 就需要使用 *kotlin-android* plugin 而不是 *kotlin* plugin:

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

Android Studio

如果使用 Android Studio, 需要在 android 之下添加以下内容:

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

这些设置告诉 Android Studio, kotlin 目录是一个源代码根目录, 因此当工程模型装载进入 IDE 时, 就可以正确地识别这个目录.

配置依赖

除了上文讲到的 *kotlin-gradle-plugin* 依赖之外, 你还需要添加 Kotlin 标准库的依赖:

```

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

apply plugin: "kotlin" // 编译到 JavaScript 时, 应该是 apply plugin: "kotlin2js"

repositories {
    mavenCentral()
}

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

```

如果你的工程使用 Kotlin 的反射功能或测试功能, 那么还需要添加相应的依赖:

```

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

```

OSGi

关于对 OSGi 的支持, 请参见 [Kotlin 与 OSGi](#).

示例

[Kotlin 代码仓库](#) 中包含以下示例:

- [Kotlin](#)
- [Java 代码与 Kotlin 代码的混合](#)
- [Android](#)
- [JavaScript](#)

Kotlin 与 OSGi

要使用 Kotlin 的 OSGi 支持功能, 你需要使用 `kotlin-osgi-bundle`, 而不是通常的 Kotlin 库文件. 此外还建议你删除 `kotlin-runtime`, `kotlin-stdlib` 和 `kotlin-reflect` 依赖, 因为 `kotlin-osgi-bundle` 已经包含了这些库的内容. 此外还需要注意不要引用外部的 Kotlin 库文件. 大多数通常的 Kotlin 库依赖都不能用于 OSGi 环境, 因此你不应该使用它们, 要将它们从你的工程中删除.

Maven

在 Maven 工程中引入 Kotlin OSGi bundle:

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

从外部库中删除 Kotlin 的标准库(注意, exclusion 设置中星号只在 Maven 3 中有效):

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

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

Gradle

在 Gradle 工程中引入 `kotlin-osgi-bundle` :

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

通过传递依赖, 你可能会间接依赖到一些默认的 Kotlin 库, 你可以使用以下方法删除这些库:

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

FAQ

为什么不直接向所有的 Kotlin 库添加需要的 manifest 设值呢？

虽然这是提供 OSGi 支持时最优先的方法, 但很不幸, 目前我们无法做到这一点, 原因是所谓的 [“包分裂 \(package split\)” 问题](#), 这个问题很难解决, 所以目前我们不打算进行这样巨大的变更. 另外还有一种 `Require-Bundle` 功能, 但也不是最好的选择, 而且并不推荐采用这种方案. 因此我们决定为 OSGi 创建一个独立的库文件.

FAQ

FAQ

常见问题

什么是 Kotlin?

Kotlin 是一种针对 JVM 和 JavaScript 环境的静态类型语言. 它是一种面向软件产业实际应用的通用语言.

Kotlin 的开发者是 JetBrains 公司的一个团队, 但它是开源的, 而且还有 JetBrains 公司之外的贡献者.

为什么要开发一种新的语言?

在 JetBrains 公司, 我们已经在 Java 平台上进行了很多年的开发工作, 而且我们很了解 Java 的优点. 但是, 我们也认识到 Java 语言存在着一些问题, 而且由于向后兼容性的限制, 导致这些问题很难甚至不可能得到解决. 我们知道 Java 还会长期存在下去, 但我们相信, 假如开发一种针对 JVM 平台的静态类型的新语言, 丢掉那些历史遗留的包袱, 加上开发者们长期渴望的功能, 那么开发社区将会因此大大受益.

这个项目背后的主要涉及目标是:

- 创建一个与 Java 兼容的语言,
- 编译速度至少要与 Java 一样快,
- 新语言要比 Java 更加安全, 也就是说, 要对软件开发中的常见问题进行静态检查, 比如对空指针的访问问题,
- 新语言要比 Java 更简洁, 要支持变量类型推断, 高阶函数(闭包), 扩展函数, 代码混合(mixin), 委托(first-class delegation), 等等;
- 此外, 将新语言的表达能力控制在实用的程度(参见上文)(译注: 这里似乎应该是参见 “与 Scala 比较” 小节), 让它比最成熟的竞争者 - Scala - 更加简单.

Kotlin 使用什么样的许可证(license)?

Kotlin 是一个开源语言, 使用 Apache 2 OSS License. IntelliJ Plug-in 也是开源的.

Kotlin 的代码目前托管在 GitHub 上, 我们很欢迎大家贡献自己的代码.

Kotlin 与 Java 兼容吗?

是的. 编译器将会输出 Java 字节码. Kotlin 可以调用 Java, 反过来 Java 也可以调用 Kotlin. 请参见 [与 Java 的互操作性](#).

运行 Kotlin 代码需要的最低 Java 版本是多少?

Kotlin 编译产生的字节码兼容 Java 6 或更高版本. 因此 Kotlin 可以用于 Android 之类的环境, Android 目前支持的最高版本是 Java 6.

有针对 Kotlin 的开发工具吗?

是的. 有一个开源的 IntelliJ IDEA plugin, 使用 Apache 2 License. 在 IntelliJ IDEA 的 [免费的 OSS Community 版](#)和 [Ultimate 版](#) 中都可以使用 Kotlin.

有 Eclipse 环境的工具吗?

是的. 关于安装方法, 请参照 [教程](#).

有不依赖 IDE 环境的独立的(standalone)编译器吗?

是的. 你可以在 [GitHub 上的 Release 页面](#) 下载独立的编译器, 以及其他构建工具.

Kotlin 是函数式语言吗(Functional Language)?

Kotlin 是面向对象的语言. 但是它支持高阶函数, Lambda 表达式, 以及顶级(top-level)函数. 此外, Kotlin 的标准库中还存在函数式语言中常见的大量元素(比如 map, flatMap, reduce, 等等.). 而且, 关于函数式语言, 并没有一个清晰的定义, 所以我们并能说 Kotlin 是一种函数式语言.

Kotlin 支持泛型吗?

Kotlin 支持泛型. 它还支持声明处的类型变异(declaration-site variance)和使用处类型变异(usage-site variance). 而且 Kotlin 没有通配符类型. 内联函数(Inline function)支持实体化的类型参数(reified type parameter).

语句末尾需要分号(semicolon)吗?

不. 分号是可选的.

需要大括号(curly brace)吗?

是的.

为什么要将类型声明放在右侧?

我们认为这样可以提高代码的可读性. 此外, 这样还有助于实现一些很好的语法功能. 比如, 可以很容易地省略类型声明. Scala 也证明了这种设计没有问题.

类型声明放在右侧会不会对开发工具造成不好的影响?

不会. 我们照样可以实现变量名称的自动提示之类的功能.

Kotlin 可以扩展吗?

我们计划让它变得可扩展, 方法包括: 内联函数, 注解, 类型装载机(type loader).

我能将自己的 DSL 嵌入到 Kotlin 中吗?

可以. Kotlin 提供了一些特性可以帮助你: 操作符重载, 通过内联函数实现自定义的控制结构, 使用中缀(infix)语法调用函数, 扩展函数, 注解, 以及语言引用(language quotation).

JavaScript 支持的 ECMAScript 级别是多少?

目前是 5.

JavaScript 后端支持模块系统(module system)吗?

是的. 我们计划提供 CommonJS 和 AMD 支持.

与 Java 比较

Kotlin 中得到解决的一些 Java 问题

Java 中长期困扰的一系列问题, 在 Kotlin 得到了解决:

- Null 引用 [由类型系统管理](#).
- [没有原生类型\(raw type\)](#)
- Kotlin 中的数组是 [类型不可变的](#)
- 与 Java 中的 SAM 变换方案相反, Kotlin 中存在专门的 [函数类型\(function type\)](#)
- 不使用通配符的 [使用处类型变异\(Use-site variance\)](#)
- Kotlin 中不存在受控 [异常](#)

Java 中有, 而 Kotlin 中没有的东西

- [受控异常](#)
- 不是类的 [基本数据类型](#)
- [静态成员](#)
- [非私有的域\(Non-private field\)](#)
- [通配符类型\(Wildcard-type\)](#)

Kotlin 中有, 而 Java 中没有的东西

- [Lambda 表达式](#) + [内联函数](#) = 实现自定义的控制结构
- [扩展函数](#)
- [Null 值安全性](#)
- [类型智能转换](#)
- [字符串模板](#)
- [属性](#)
- [主构造器](#)
- [委托\(First-class delegation\)](#)
- [变量和属性的类型推断](#)
- [单例\(Singleton\)](#)
- [声明处类型变异\(Declaration-site variance\)](#) 和 [类型投射\(Type projection\)](#)
- [值范围表达式](#)
- [操作符重载](#)
- [同伴对象\(Companion object\)](#)
- [数据类](#)

— [集合的接口定义区分为只读集合与可变集合](#)

与 Scala 比较

Kotlin 的主要目标是创建一种实用的、高生产性的编程语言, 而不是验证编程语言理论研究的新结果. 从这一点考虑, 如果你喜欢 Scala, 那么你可能就不需要 Kotlin.

Scala 中有, 而 Kotlin 中没有的东西

- 隐式转换, 参数, 等等
 - 在 Scala 中, 有时不使用调试器简直很难搞清楚你的代码中究竟发生了什么, 因为在程序中出现了太多的隐含处理
 - 要对你的类型添加新的功能, 在 Kotlin 中请使用 [扩展函数](#).
- 可覆盖的类型成员(Overridable type member)
- 路径依赖类型(Path-dependent type)
- 宏
- 存在类型(Existential type)
 - [类型投射\(Type projection\)](#) 是一种很特殊的情况
- 特征(trait)初始化中的复杂逻辑
 - 参见 [类与继承](#)
- 自定义的符号化操作(symbolic operation)
 - 参见 [操作符重载](#)
- 内建的 XML
 - 参见 [类型安全的 Groovy 风格构建器\(Type-safe Groovy-style builder\)](#)
- 结构化类型(Structural type)
- 值类型(Value type)
 - 当 [Valhalla 项目](#) 作为 JDK 的一部分发布时, 我们计划支持这个项目
- Yield 操作符
- Actor
 - Kotlin 支持 [Quasar](#), 一个第三方框架, 可以在 JVM 上支持 actor
- 平行集合(Parallel collection)
 - Kotlin 支持 Java 8 的 stream, 它可以提供类似的功能

Kotlin 中有, 而 Scala 中没有的东西

- [无任何额外耗费的 Null 值安全性](#)
 - Scala 中有 Option, 它是一种语法上的和运行时的封装

- [类型智能转换](#)
- [Kotli 的内联函数可以帮助实现非局部的跳转\(Nonlocal jump\)](#)
- [委托\(First-class delegation\)](#). 可以使用第三方 plugin 实现: Autoproxy
- [成员引用](#) (Java 8 中也支持).

