### L1.IDEA部署Kotlin环境

## IDEA环境部署

### 1.1 下载IDEA编译器下载最新版的就可以了

<https://www.jetbrains.com/idea/>

### 1.2 安装IDEA插件

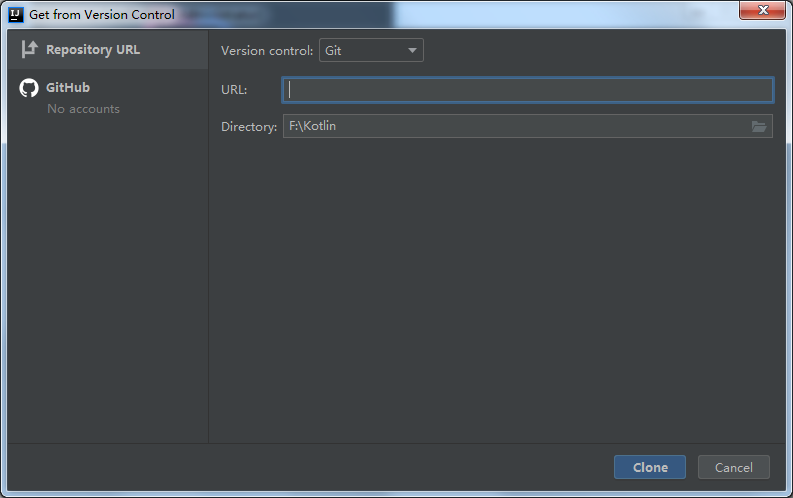
<https://zhile.io/2020/11/18/jetbrains-eval-reset-da33a93d.html>

### 1.3 获取教程项目

完成以上部署后获取教程项目：

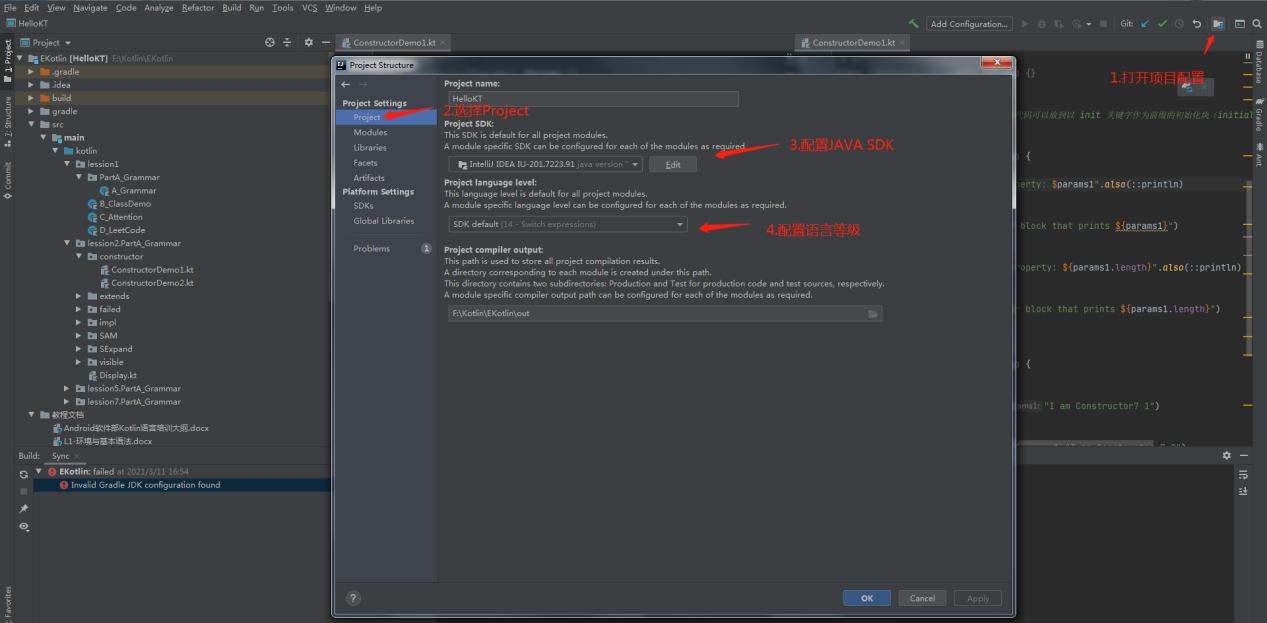


输入: <https://github.com/JustGank/EKotlin.git>



正常Build获取即可。

### 1.4 获取完成配置项目



配置完成后项目不报错，代表环境部署成功。

此时就具备了Kotlin语法的实验环境。

## 二、基本语法介绍

此部分作为Kotlin的大体语法展示单元，主要是展示Kotlin在常规编程中的使用方式。此部分讲解时需要体现特性和重点，部分细节可以放到后面单元专项中进行讲解。

### 2.1 包的定义与导入

包的声明应处于源文件顶部：

package my.demo

import kotlin.text.\*

此部分和JAVA类似，不用特殊关注

### 2.2 程序的入口点

Kotlin 应用程序的入口点是 main 函数。

fun main() {

println("Hello world!")

}

每一个文件中都可以有 fun main()的存在。这样可以很方便的让我们在每个文件内部完成展示和调试。

### 2.3函数

带有两个 Int 参数、返回 Int 的函数：

fun sum1(a: Int, b: Int): Int {

return a + b

}

Kotlin是先声明变量名，再声明变量类型。

将表达式作为函数体、返回值类型自动推断的函数：

fun sum2(a: Int, b: Int) = a + b

函数返回无意义的值,此处可以理解为Void,Unit也可以不写：

fun printSum(a: Int, b: Int): Unit {

println("sum of $a and $b is ${a + b}")

}

### 2.4变量

定义只读局部变量使用关键字 val 定义。**只能为其赋值一次。**

val a: Int = 1 // 立即赋值

val b = 2 // 自动推断出 `Int` 类型

val c: Int // 如果没有初始值类型不能省略

c = 3 // 明确赋值

可重新赋值的变量使用 var 关键字：

var x = 5 // 自动推断出 `Int` 类型

x += 1

顶层变量：

val PI = 3.14

var x = 0

fun incrementX() {

x += 1

}

### 2.5 注释

与大多数现代语言一样，Kotlin 支持单行（或行末）与多行（块）注释。

// 这是一个行注释

/\* 这是一个多行的

块注释。 \*/

Kotlin 中的块注释可以嵌套。

/\* 注释从这里开始

/\* 包含嵌套的注释 \*/

并且在这里结束。 \*/

### 2.6 字符串模板

此处就是$引用，直接引用变量时使用$即可，使用变量的某个变量时可以使用 {} 将要输入的内容包起来。

var a = 1

// 模板中的简单名称：

val s1 = "a is $a"

a = 2

// 模板中的任意表达式：

val s2 = "${s1.replace("is", "was")}, but now is $a"

### 2.7 条件表达式

下面两种写法等价：

fun maxOf(a: Int, b: Int): Int {

if (a > b) {

return a

} else {

return b

}

}

fun maxOf(a: Int, b: Int) = if (a > b) a else b（简单逻辑更推荐）

### 2.8 空值与 null 检测

**当某个变量的值可以为 null 的时候，必须在声明处的类型后添加 “？” 来标识该引用可为空。**

如果 str 的内容不是数字返回 null：

fun parseInt(str: String): Int? {}

使用返回可空值的函数:

fun printProduct(arg1: String, arg2: String) {

val x = parseInt(arg1)

val y = parseInt(arg2)

// 直接使用 `x \* y` 会导致编译错误，因为它们可能为 null

if (x != null && y != null) {

// 在空检测后，x 与 y 会自动转换为非空值（non-nullable）

println(x \* y)

}

else {

println("'$arg1' or '$arg2' is not a number")

}

if (x == null) {

println("Wrong number format in arg1: '$arg1'")

return

}

if (y == null) {

println("Wrong number format in arg2: '$arg2'")

return

}

// 在空检测后，x 与 y 会自动转换为非空值

println(x \* y)

}

### 2.9 类型检测与自动类型转换

is 运算符检测一个表达式是否某类型的一个实例。 如果一个不可变的局部变量或属性已经判断出为某类型，那么检测后的分支中可以直接当作该类型使用，无需显式转换：

fun getStringLength(obj: Any): Int? {

if (obj is String) {

// `obj` 在该条件分支内自动转换成 `String`

return obj.length

}

// 在离开类型检测分支后，`obj` 仍然是 `Any` 类型

return null

}

写法1：

fun getStringLength(obj: Any): Int? {

if (obj !is String) return null

// `obj` 在这一分支自动转换为 `String`

return obj.length

}

写法2：

fun getStringLength(obj: Any): Int? {

// `obj` 在 `&&` 右边自动转换成 `String` 类型

if (obj is String && obj.length > 0) {

return obj.length

}

return null

}

### 2.10 for 循环

val items = listOf("apple", "banana", "kiwifruit")

for (item in items) {

println(item)

}

有索引的遍历：

val items = listOf("apple", "banana", "kiwifruit")

for (index in items.indices) {

println("item at $index is ${items[index]}")

}

### 2.11 while 循环

val items = listOf("apple", "banana", "kiwifruit")

var index = 0

while (index < items.size) {

println("item at $index is ${items[index]}")

index++

}

### 2.12 when 表达式

fun describe(obj: Any): String =

when (obj) {

1 -> "One"

"Hello" -> "Greeting"

is Long -> "Long"

!is String -> "Not a string"

else -> "Unknown"

}

### 2.13 使用区间（range）

使用 in 运算符来检测某个数字是否在指定区间内，区间是包含收尾元素的！：

val x = 10

val y = 9

if (x in 1..y+1) {

println("fits in range")

}

检测某个数字是否在指定区间外:

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

if (-1 !in 0..list.lastIndex) {

println("-1 is out of range")

}

if (list.size !in list.indices) {

println("list size is out of valid list indices range, too")

}

区间迭代:

for (x in 1..5) {

print(x)

}

或数列迭代：

//代表从1开始 每次+2 一直到9

for (x in 1..10 step 2) {

print(x)

}

输出：13579

//在9-0这个逆序区间 每次减3

for (x in 9 downTo 0 step 3) {

print(x)

}

输出：9630

### 2.14 集合

对集合进行迭代:

for (item in items) {

println(item)

}

使用 in 运算符来判断集合内是否包含某实例：

//这是代替 if-else if-else if 的一种语法糖

val items = setOf("apple", "banana", "kiwifruit")

when {

"orange" in items -> println("juicy")

"apple" in items -> println("apple is fine too")

}

使用 lambda 表达式来过滤（filter）与映射（map）集合：

val fruits = listOf("banana", "avocado", "apple", "kiwifruit")

fruits.filter { it.startsWith("a") }

.sortedBy { it }

.map { it.toUpperCase() }

.forEach { println(it) }

### 2.15 创建基本类及其实例

fun main() {

val rectangle = Rectangle(5.0, 2.0)

val triangle = Triangle(3.0, 4.0, 5.0)

println("Area of rectangle is ${rectangle.calculateArea()}, its perimeter is ${rectangle.perimeter}")

println("Area of triangle is ${triangle.calculateArea()}, its perimeter is ${triangle.perimeter}")

}

//抽象父类 “多边形” 直接声明构造方法

abstract class Shape(val sides: List<Double>) {

val perimeter: Double get() = sides.sum()

abstract fun calculateArea(): Double

}

interface RectangleProperties {

val isSquare: Boolean

}

//声明矩形类

class Rectangle(

var height: Double,

var length: Double

) : Shape(listOf(height, length, height, length)), RectangleProperties {

//重写接口属性

override val isSquare: Boolean get() = length == height

//重写面积计算

override fun calculateArea(): Double = height \* length

}

class Triangle(

var sideA: Double,

var sideB: Double,

var sideC: Double

) : Shape(listOf(sideA, sideB, sideC)) {

override fun calculateArea(): Double {

val s = perimeter / 2

return Math.sqrt(s \* (s - sideA) \* (s - sideB) \* (s - sideC))

}

}

## 三、习惯用法

### 3.1 创建 DTOs（POJOs/POCOs）

POJOs有时候也称作Plain Ordinary Java Objects，表示一个数据集合。

POJO只是一个普通的，已删除限制的Java Bean。 Java Bean必须满足以下要求：

1. 默认无参数构造函数

2. 对于值foo为可变属性，遵循get()方法和set()方法的Bean协议；如果值foo是不可变的，则不使用setFoo。

3.必须实现java的序列化(java.io.Serializable)

POJO不强制执行这些操作。顾名思义：在JDK下编译的对象可以被认为是一个普通的Java对象。没有应用服务器，没有基类，没有需要使用的接口。

**简答来说就是使用data class声明数据Bean。**

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

**声明后，编译器会为我们重写很多方法：**

* 所有属性的 getters （对于 var 定义的还有 setters）
* equals()
* hashCode()
* toString()
* copy()
* 所有属性的 component1()、 component2()……等等

### 3.2 函数的默认参数

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

使用带默认参数的方法，有默认参数的参数，其值是选写的。

### 3.3 过滤 list

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

或者可以更短:

val positives = list.filter { it > 0 }

### 3.4 检测元素是否存在于集合中

In在返回true，!in不在返回true

if ("john@example.com" in emailsList) { …… }

if ("jane@example.com" !in emailsList) { …… }

### 3.5字符串内插(同2.6)

println("Name $name")

### 3.6 类型判断

when (x) {

is Foo //-> ……

is Bar //-> ……

else //-> ……

}

### 3.7 遍历 map/pair型list

for ((k, v) in map) {

println("$k -> $v")

}

### 3.8 使用区间

for (i in 1..100) { …… }  **// 闭区间：包含 100**

for (i in 1 until 100) { …… } // **半开区间：不包含 100**

for (x in 2..10 step 2) { …… }

for (x in 10 downTo 1) { …… }

if (x in 1..10) { …… }

### 3.9 只读 list(只读代表声明后不能编辑)

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

### 3.10 只读 map(to 是 Pair 声明的一个中缀拓展函数)

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

### 3.11 访问  map（mutable代表可读写 ）

fun visitMap() {  
 val map = mutableMapOf("a" to 1, "b" to 2, "c" to 3)  
 map["key"] = 123  
 println(map["key"])  
}

### 3.13 延迟属性(使用时初始化变量值)

val p: String by lazy {

println(“Init p”)

“Hello!”

}

延迟属性Lazy 与 lateinit 区别

以下是lateinit var和by lazy { ... }委托属性之间的显著差异：

lazy { ... }代表只能用于val属性，而lateinit只能用于var，因为它不能编译到final字段，因此不能保证不变性;

lateinit var具有存储值的后备字段(backing field)，而by lazy { ... }创建一个委托对象，其中存储一次计算的值，将对代理实例的引用存储在类对象中，并为与委托实例一起使用的属性生成getter。

除了val之外，lateinit不能用于可空属性和Java原语类型(这是因为null用于未初始化的值);所以如果你需要在类中存在的支持字段，请使用lateinit; lateinit var可以从对象被看到的任何地方被初始化。从一个框架代码的内部，多个初始化方案是可能的单一类的不同对象。 by lazy { ... }反过来又定义了属性的唯一初始化器，只能通过覆盖子类中的属性进行更改。如果您希望以预先未知的方式从外部初始化属性，请使用lateinit。

另外，还有一个方法没有提到Delegates.notNull()，它适用于non-null属性的延迟初始化，包括Java原始类型的属性。

延迟属性Lazy 与 lateinit 使用总结

lateinit用于外部初始化：当需要外部资料通过调用方法初始化您的值时。 例如通过调用：

private lateinit var value: MyClass

fun init(externalProperties: Any) {

value = somethingThatDependsOn(externalProperties)}

而lazy当它只使用对象内部的依赖关系时。 嗯。静态变量的初始化挺适合这种方式的。

### 3.14 拓展函数

通过：类.拓展方法 声明

fun String.spaceToCamelCase() { …… }

### 3.15 创建单例（伴生对象）

object Resource {

val name = "Name"

}

里面的属性和方法都可以使用 类.的形式直接使用。

### 3.16 If not null 缩写

**当files为空时println会输出null**

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

println(files?.size)

// ！！断言会空针异常  
println(files!!.size)

### 3.17 If not null and else 缩写

？：类似于 三目原运算，当files为空的时候会输出empty

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

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

### 3.18 if null 执行一个语句

val values = ……

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

### 3.19 在可能会空的集合中取第一元素

val emails = …… // 可能会是空集合 emails 需要是集合类型

val mainEmail = emails.firstOrNull() ?: ""

### 3.20 if not null 执行代码

val value = ……

value?.let {

…… // 代码会执行到此处, 假如data不为null

}

### 3.21 映射可空值（如果非空的话）

val value = ……

val mapped = value?.let { transformValue(it) } ?: defaultValue

// 如果该值或其转换结果为空，那么返回 defaultValue。

### 3.22返回 when 表达式

fun transform(color: String): Int {

return when (color) {

"Red" -> 0

"Green" -> 1

"Blue" -> 2

else -> throw IllegalArgumentException("Invalid color param value")

}

}

### 3.23 “try/catch”表达式

fun test() {

val result = try {

count()

} catch (e: ArithmeticException) {

throw IllegalStateException(e)

}

}

### 3.24 “if”表达式

fun foo(param: Int) {

val result = if (param == 1) {

"one"

} else if (param == 2) {

"two"

} else {

"three"

}

}

### 3.25 返回类型为 Unit 的方法的 Builder 风格用法

Fill()函数 是简单的将集合内的所有元素替换成指定元素

fun arrayOfMinusOnes(size: Int): IntArray {

//生成一个大小为Size的集合 让里面的元素都是-1

return IntArray(size).apply { fill(-1) }

}

### 3.26 单表达式函数

fun theAnswer() = 42

等价于

fun theAnswer(): Int {

return 42

}

单表达式函数与其它惯用法一起使用能简化代码，例如和 when 表达式一起使用：

fun transform(color: String): Int = when (color) {

"Red" -> 0

"Green" -> 1

"Blue" -> 2

else -> throw IllegalArgumentException("Invalid color param value")

}

### 3.27 对一个对象实例调用多个方法 （with）

没有实现Builder的类，可以使用类于Builder的写法，让程序更加简洁。

class Turtle {

fun penDown()

fun penUp()

fun turn(degrees: Double)

fun forward(pixels: Double)

}

val myTurtle = Turtle()

with(myTurtle) { // 画一个 100 像素的正方形

penDown()

for (i in 1..4) {

forward(100.0)

turn(90.0)

}

penUp()

}

### 3.28 配置对象的属性（apply）

val myRectangle = Rectangle().apply { //直接对参数赋值

length = 4

breadth = 5

color = 0xFAFAFA

}

这对于配置未出现在对象构造函数中的属性非常有用。

### 3.29 对于需要泛型信息的泛型函数的适宜形式

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

Reified：这个是为了满足inline特性而设计的语法糖，因为给函数使用内联之后，编译器会用其函数体来替换掉函数调用，而如果该函数里面有泛型就可能会出现编译器不懂该泛型的问题，所以引入reified，使该泛型被智能替换成对应的类型

### 3.30 使用可空布尔

val b: Boolean? = ……

if (b == true) {

……

} else {

// `b` 是 false 或者 null

}

### 3.31 交换两个变量

Also代表做这个操作的同时，也做另外一个操作。

var a = 1

var b = 2

a = b.also { b = a }

### 3.32 TODO()：将代码标记为不完整

Kotlin 的标准库有一个 TODO() 函数，该函数总是抛出一个 NotImplementedError。 其返回类型为 Nothing，因此无论预期类型是什么都可以使用它。 还有一个接受原因参数的重载：

fun calcTaxes(): BigDecimal = TODO("Waiting for feedback from accounting")

IntelliJ IDEA 的 kotlin 插件理解 TODO() 的语言，并且会自动在 TODO 工具窗口中添加代码指示。

## 四、基本类型

### 4.1 数字

Kotlin 提供了一组表示数字的内置类型。 对于整数，有四种不同大小的类型，因此值的范围也不同。

| 类型 | 大小（比特数） | 最小值 | 最大值 |
| --- | --- | --- | --- |
| Byte | 8 | -128 | 127 |
| Short | 16 | -32768 | 32767 |
| Int | 32 | -2,147,483,648 (-231) | 2,147,483,647 (231 - 1) |
| Long | 64 | -9,223,372,036,854,775,808 (-263) | 9,223,372,036,854,775,807 (263 - 1) |

**所有以未超出 Int 最大值的整型值初始化的变量都会推断为 Int 类型**。如果初始值超过了其最大值，那么推断为 Long 类型。 如需显式指定 Long 型值，请在该值后追加 L 后缀。

val one = 1 // Int

val threeBillion = 3000000000 // Long

val oneLong = 1L // Long

val oneByte: Byte = 1

对于浮点数，Kotlin 提供了 Float 与 Double 类型。 根据 [IEEE 754 标准](https://zh.wikipedia.org/wiki/IEEE_754)， 两种浮点类型的十进制位数（即可以存储多少位十进制数）不同。 Float 反映了 IEEE 754 单精度，而 Double 提供了双精度。

| 类型 | 大小（比特数） | 有效数字比特数 | 指数比特数 | 十进制位数 |
| --- | --- | --- | --- | --- |
| Float | 32 | 24 | 8 | 6-7 |
| Double | 64 | 53 | 11 | 15-16 |

对于以小数初始化的变量，编译器会推断为 Double 类型。 如需将一个值显式指定为 Float 类型，请添加 f 或 F 后缀。 如果这样的值包含多于 6～7 位十进制数，那么会将其舍入。

val pi = 3.14 // Double

val e = 2.7182818284 // Double

val eFloat = 2.7182818284f // Float，实际值为 2.7182817

请注意，与一些其他语言不同，Kotlin 中的数字没有隐式拓宽转换。 **例如，具有 Double 参数的函数只能对 Double 值调用，而不能对 Float、 Int 或者其他数字值调用**。

fun printDouble(d: Double) { print(d) }

val i = 1

val d = 1.1

val f = 1.1f

printDouble(d)

// printDouble(i) // 错误：类型不匹配

// printDouble(f) // 错误：类型不匹配

### 4.2 字面常量

数值常量字面值有以下几种:

十进制: 123

Long 类型用大写 L 标记: 123L

十六进制: 0x0F

二进制: 0b00001011

**注意: 不支持八进制**

Kotlin 同样支持浮点数的常规表示方法:

默认 double：123.5、123.5e10

Float 用 f 或者 F 标记: 123.5f

### 4.2 数字字面值中的下划线（自 1.1 起）

你可以使用下划线使数字常量更易读：

val oneMillion = 1\_000\_000

val creditCardNumber = 1234\_5678\_9012\_3456L

val socialSecurityNumber = 999\_99\_9999L

val hexBytes = 0xFF\_EC\_DE\_5E

val bytes = 0b11010010\_01101001\_10010100\_10010010

### 4.3 表示方式

在 Java 平台数字是物理存储为 JVM 的原生类型，除非我们需要一个可空的引用（如 Int?）或泛型。 后者情况下会把数字装箱。

注意数字装箱不一定保留同一性，保留相等性:

for (i in 120..130) {  
 val boxedA: Int? = i  
 val anotherBoxedA: Int? = i  
 println("i =$i boxedA === anotherBoxedA is ${boxedA === anotherBoxedA} boxedA == anotherBoxedA s ${boxedA == anotherBoxedA}") // true  
}

运行结果：

i =120 boxedA === anotherBoxedA is true boxedA == anotherBoxedA s true

i =121 boxedA === anotherBoxedA is true boxedA == anotherBoxedA s true

i =122 boxedA === anotherBoxedA is true boxedA == anotherBoxedA s true

i =123 boxedA === anotherBoxedA is true boxedA == anotherBoxedA s true

i =124 boxedA === anotherBoxedA is true boxedA == anotherBoxedA s true

i =125 boxedA === anotherBoxedA is true boxedA == anotherBoxedA s true

i =126 boxedA === anotherBoxedA is true boxedA == anotherBoxedA s true

i =127 boxedA === anotherBoxedA is true boxedA == anotherBoxedA s true

i =128 boxedA === anotherBoxedA is false boxedA == anotherBoxedA s true

i =129 boxedA === anotherBoxedA is false boxedA == anotherBoxedA s true

i =130 boxedA === anotherBoxedA is false boxedA == anotherBoxedA s true

这里的不一定，形容是不准确的，这个Int的内部实现有关。

从日志可以看出127以内：boxedA === anotherBoxedA is true

这是同一性，因为Int内部已经创建完了这些对象，他们只是引用了已经创建完的Int内部对象。从128 开始是从新创建对象，这时就不在具备同一性了。

### 4.4 显式转换

由于不同的表示方式，较小类型并不是较大类型的子类型。 如果它们是的话，就会出现下述问题：

// 假想的代码，实际上并不能编译：

val a: Int? = 1 // 一个装箱的 Int (java.lang.Integer)

val b: Long? = a // 隐式转换产生一个装箱的 Long (java.lang.Long)

print(b == a) // 惊！这将输出“false”鉴于 Long 的 equals() 会检测另一个是否也为 Long

所以相等性会在所有地方悄无声息地失去，更别说同一性了。

因此较小的类型不能隐式转换为较大的类型。 这意味着在不进行显式转换的情况下我们不能把 Byte 型值赋给一个 Int 变量。

val b: Byte = 1 // OK, 字面值是静态检测的

val i: Int = b // 错误

我们可以显式转换来拓宽数字

**val i: Int = b.toInt() // OK：显式拓宽**

print(i)

**每个数字类型支持如下的转换:**

toByte(): Byte

toShort(): Short

toInt(): Int

toLong(): Long

toFloat(): Float

toDouble(): Double

toChar(): Char

**同时String也支持这些转换方法，可以参考StringNumberConversions.kt文件。**

缺乏隐式类型转换很少会引起注意，因为类型会从上下文推断出来，而算术运算会有重载做适当转换，例如：

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

### 4.5 运算

Kotlin支持数字运算的标准集（+ - \* / %），运算被定义为相应的类成员（但编译器会将函数调用优化为相应的指令）。

#### 4.5.1整数除法

请注意，整数间的除法总是返回整数。会丢弃任何小数部分。例如：

val x = 5 / 2

//println(x == 2.5) // ERROR: Operator '==' cannot be applied to 'Int' and 'Double'

println(x == 2)

对于任何两个整数类型之间的除法来说都是如此。

val x = 5L / 2

println(x == 2L) //输出true

如需返回浮点类型，请将其中的一个参数显式转换为浮点类型。

val x = 5 / 2.toDouble()

println(x == 2.5)

#### 4.5.2位运算

对于位运算，没有特殊字符来表示，而只可用中缀方式调用具名函数，例如:

val x = (1 shl 2) and 0x000FF000

这是完整的位运算列表（只用于 Int 与 Long）：

* shl(bits) – 有符号左移
* shr(bits) – 有符号右移
* ushr(bits) – 无符号右移
* and(bits) – 位与
* or(bits) – 位或
* xor(bits) – 位异或
* inv() – 位非

### 4.6浮点数比较

本节讨论的浮点数操作如下：

相等性检测：a == b 与 a != b

比较操作符：a < b、 a > b、 a <= b、 a >= b

区间实例以及区间检测：a..b、 x in a..b、 x !in a..b

当其中的操作数 a 与 b 都是静态已知的 Float 或 Double 或者它们对应的可空类型（声明为该类型，或者推断为该类型，或者[智能类型转换](https://www.kotlincn.net/docs/reference/typecasts.html" \l "%E6%99%BA%E8%83%BD%E8%BD%AC%E6%8D%A2)的结果是该类型），两数字所形成的操作或者区间遵循 IEEE 754 浮点运算标准。

然而，为了支持泛型场景并提供全序支持，当这些操作数并非静态类型为浮点数（例如是 Any、 Comparable<……>、 类型参数）时，这些操作使用为 Float 与 Double 实现的不符合标准的 equals 与 compareTo，这会出现：

认为 NaN 与其自身相等

认为 NaN 比包括正无穷大（POSITIVE\_INFINITY）在内的任何其他元素都大

认为 -0.0 小于 0.0

### 4.7 字符

字符用 Char 类型表示。**它们不能直接当作数字**

var c: Char=’c’

if (c == 1) { // 错误：类型不兼容

Java可以

char a='a';  
if(a==100){}

字符字面值用单引号括起来: '1'。 特殊字符可以用反斜杠转义。 支持这几个转义序列：\t、 \b、\n、\r、\'、\"、\\ 与 \$。 编码其他字符要用 Unicode 转义序列语法：'\uFF00'。

我们可以显式把字符转换为 Int 数字：

fun decimalDigitValue(c: Char): Int {

if (c !in '0'..'9')

throw IllegalArgumentException("Out of range")

return c.toInt() - '0'.toInt() // 显式转换为数字

}

当需要可空引用时，像数字、字符会被装箱。装箱操作不会保留同一性。

### 4.8 布尔

布尔用 Boolean 类型表示，它有两个值：true 与 false。

若需要可空引用布尔会被装箱。

内置的布尔运算有：

* || – 逻辑或
* && – 逻辑与
* ! - 逻辑非

### 4.9 数组

**数组在 Kotlin 中使用 Array 类来表示**，它定义了 get 与 set 函数（按照运算符重载约定这会转变为 []）以及 size 属性，以及一些其他有用的成员函数：

class Array<T> private constructor() {

val size: Int

operator fun get(index: Int): T

operator fun set(index: Int, value: T): Unit

operator fun iterator(): Iterator<T>

// ……

}

我们可以使用库函数 arrayOf() 来创建一个数组并传递元素值给它，这样 arrayOf(1, 2, 3) 创建了 array [1, 2, 3]。 或者，**库函数 arrayOfNulls() 可以用于创建一个指定大小的、所有元素都为空的数组。**

public fun <reified @PureReifiable T> arrayOfNulls(size: Int): Array<T?>

另一个选项是用接受数组大小以及一个函数参数的 Array 构造函数，用作参数的函数能够返回给定索引的每个元素初始值：

// 创建一个 Array<String> 初始化为 ["0", "1", "4", "9", "16"]

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

asc.forEach { println(it) }

如上所述，[] 运算符代表调用成员函数 get() 与 set()。

Kotlin 中数组是不型变的（invariant）。这意味着 Kotlin 不让我们把 Array<String> 赋值给 Array<Any>，以防止可能的运行时失败。

### 4.10 原生类型数组

Kotlin 也有无装箱开销的专门的类来表示原生类型数组: ByteArray、 ShortArray、IntArray 等等。这些类与 Array 并没有继承关系，但是它们有同样的方法属性集。它们也都有相应的工厂方法:

val x: IntArray = intArrayOf(1, 2, 3)

x[0] = x[1] + x[2]

// 大小为 5、值为 [0, 0, 0, 0, 0] 的整型数组

val arr = IntArray(5)

// 例如：用常量初始化数组中的值

// 大小为 5、值为 [42, 42, 42, 42, 42] 的整型数组

val arr = IntArray(5) { 42 }

// 例如：使用 lambda 表达式初始化数组中的值

// 大小为 5、值为 [0, 1, 2, 3, 4] 的整型数组（值初始化为其索引值）

var arr = IntArray(5) { it \* 1 }

### 4.11 无符号整型

Kotlin 为无符号整数引入了以下类型：

kotlin.UByte: 无符号 8 比特整数，范围是 0 到 255

kotlin.UShort: 无符号 16 比特整数，范围是 0 到 65535

kotlin.UInt: 无符号 32 比特整数，范围是 0 到 2^32 - 1

kotlin.ULong: 无符号 64 比特整数，范围是 0 到 2^64 - 1

无符号类型支持其对应有符号类型的大多数操作。

### 4.12 特化的类

与原生类型相同，每个无符号类型都有相应的为该类型特化的表示数组的类型：

kotlin.UByteArray: 无符号字节数组

kotlin.UShortArray: 无符号短整型数组

kotlin.UIntArray: 无符号整型数组

kotlin.ULongArray: 无符号长整型数组

与有符号整型数组一样，它们提供了类似于 Array 类的 API 而没有装箱开销。

此外，[区间与数列](https://www.kotlincn.net/docs/reference/ranges.html)也支持 UInt 与 ULong（通过这些类 kotlin.ranges.UIntRange、 kotlin.ranges.UIntProgression、 kotlin.ranges.ULongRange、 kotlin.ranges.ULongProgression）

### 4.13 字面值

为使无符号整型更易于使用，Kotlin 提供了用后缀标记整型字面值来表示指定无符号类型（类似于 Float/Long）：

后缀 u 与 U 将字面值标记为无符号。确切类型会根据预期类型确定。如果没有提供预期的类型，会根据字面值大小选择 UInt 或者 ULong

val b: UByte = 1u // UByte，已提供预期类型

val s: UShort = 1u // UShort，已提供预期类型

val l: ULong = 1u // ULong，已提供预期类型

val a1 = 42u // UInt：未提供预期类型，常量适于 UInt

val a2 = 0xFFFF\_FFFF\_FFFFu // ULong：未提供预期类型，常量不适于 UInt

后缀 uL 与 UL 显式将字面值标记为无符号长整型。

val a = 1UL // ULong，即使未提供预期类型并且常量适于 UInt

### 4.14 字符串

字符串用 String 类型表示。字符串是不可变的。 字符串的元素——字符可以使用索引运算符访问: s[i]。 可以用 for 循环迭代字符串:

for (c in str) {

println(c)

}

可以用 + 操作符连接字符串。这也适用于连接字符串与其他类型的值， 只要表达式中的第一个元素是字符串：

val s = "abc" + 1

println(s + "def")

请注意，在大多数情况下，优先使用[字符串模板](https://www.kotlincn.net/docs/reference/basic-types.html" \l "%E5%AD%97%E7%AC%A6%E4%B8%B2%E6%A8%A1%E6%9D%BF)（$）或原始字符串而不是字符串连接。

#### 4.14.1字符串字面值

Kotlin 有两种类型的字符串字面值: 转义字符串可以有转义字符， 以及原始字符串可以包含换行以及任意文本。以下是转义字符串的一个示例:

val s = "Hello, world!\n"

原始字符串 使用三个引号（"""）分界符括起来，**内部没有转义并且可以包含换行以及任何其他字符**:

val text = """

for (c in "foo")

print(c)

"""

字符串的格式就是“““ 中的格式。省略了转义字符，适合格式复杂的多行字符串。

你可以通过 [trimMargin()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.text/trim-margin.html) 函数去除前导空格：

val text = """

|Tell me and I forget.

|Teach me and I remember.

|Involve me and I learn.

|(Benjamin Franklin)

""".trimMargin()

默认 | 用作边界前缀，但你可以选择其他字符并作为参数传入，比如 trimMargin(">")。

#### 4.14.2 字符串模板

字符串字面值可以包含模板表达式 ，即一些小段代码，会求值并把结果合并到字符串中。 模板表达式以美元符（$）开头，由一个简单的名字构成:

val i = 10

println("i = $i") // 输出“i = 10”

或者用花括号括起来的任意表达式:

val s = "abc"

println("$s.length is ${s.length}")

原始字符串与转义字符串内部都支持模板。 如果你需要在原始字符串中表示字面值 $ 字符（它不支持反斜杠转义），你可以用下列语法：

val price = """

${'$'}9.99

"""

## 包与导入（无代码样例）

### 5.1 包

源文件通常以包声明开头:

package org.example

fun printMessage() { /\*……\*/ }

class Message { /\*……\*/ }

### 5.2 默认导入

有多个包会默认导入到每个 Kotlin 文件中：

* [kotlin.\*](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/index.html)
* [kotlin.annotation.\*](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.annotation/index.html) 注解
* [kotlin.collections.\*](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/index.html) 集合
* [kotlin.comparisons.\*](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.comparisons/index.html) 反射
* [kotlin.io.\*](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.io/index.html)
* [kotlin.ranges.\*](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.ranges/index.html)
* [kotlin.sequences.\*](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.sequences/index.html) 序列
* [kotlin.text.\*](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.text/index.html)

根据目标平台还会导入额外的包：

JVM: java.lang.\* [kotlin.jvm.\*](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.jvm/index.html)

JS: [kotlin.js.\*](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.js/index.html)

### 5.3 导入

除了默认导入之外，每个文件可以包含它自己的导入指令。 导入语法在[语法](https://www.kotlincn.net/docs/reference/grammar.html" \l "importHeader)中讲述。

可以导入一个单独的名字，如.

import org.example.Message // 现在 Message 可以不用限定符访问

也可以导入一个作用域下的所有内容（包、类、对象等）:

import org.example.\* // “org.example”中的一切都可访问

如果出现名字冲突，可以使用 as 关键字在本地重命名冲突项来消歧义：

import org.example.Message // Message 可访问

import org.test.Message as testMessage // testMessage 代表“org.test.Message”

关键字 import 并不仅限于导入类；也可用它来导入其他声明：

顶层函数及属性；

在[对象声明](https://www.kotlincn.net/docs/reference/object-declarations.html" \l "%E5%AF%B9%E8%B1%A1%E5%A3%B0%E6%98%8E)中声明的函数和属性;

[枚举常量](https://www.kotlincn.net/docs/reference/enum-classes.html)。

### 5.4 顶层声明的可见性

如果顶层声明是 private 的，它是声明它的文件所私有的。

### 5.5 可见性修饰符

**public**修饰符表示 公有 。此修饰符的范围最大。当不声明任何修饰符时，系统会默认使用此修饰符。

**internal**修饰符表示 模块 。就是Module内可见。

**protected**修饰符表示 私有+子类。值得注意的是，此修饰符不能用于顶层声明，在下面可以看到。

**private**修饰符表示 私有 。此修饰符的范围最小，即可见性范围最低。

## 控制流

注意这个流并不是协程中的流。

这里的流主要是逻辑流程的流如：if、when、for、while

### 6.1 If 表达式

在 Kotlin 中，if是一个表达式，即它会返回一个值。 因此就不需要三元运算符（条件 ? 然后 : 否则），因为普通的 if 就能胜任这个角色。

// 传统用法

var max = a

if (a < b) max = b

// With else

var max: Int

if (a > b) {

max = a

} else {

max = b

}

// 作为表达式

val max = if (a > b) a else b

**Kotlin不支持三目元运算符，其实三目元更简洁。**

if 的分支可以是代码块，最后的表达式作为该块的值：

val max = if (a > b) {

print("Choose a")

a

} else {

print("Choose b")

b

}

**如果你使用 if 作为表达式而不是语句（例如：返回它的值或者把它赋给变量），该表达式需要有 else 分支。也就是保证表达式会产生返回值。**

### 6.2 When 表达式

when 表达式取代了类 C 语言的 switch 语句。其最简单的形式如下：

when (x) {

1 -> print("x == 1")

2 -> print("x == 2")

else -> { // 注意这个块

print("x is neither 1 nor 2")

}

}

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

如果其他分支都不满足条件将会求值 else 分支。 如果 when 作为一个表达式使用，则必须有 else 分支， 除非编译器能够检测出所有的可能情况都已经覆盖了［例如，对于 [枚举（enum）类](https://www.kotlincn.net/docs/reference/enum-classes.html)条目与[密封（sealed）类](https://www.kotlincn.net/docs/reference/sealed-classes.html)子类型］。

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

when (x) {

0, 1 -> print("x == 0 or x == 1")

else -> print("otherwise")

}

我们可以用任意表达式（而不只是常量）作为分支条件

when (x) {

parseInt(s) -> print("s encodes x")

else -> print("s does not encode x")

}

我们也可以检测一个值在（in）或者不在（!in）一个[区间](https://www.kotlincn.net/docs/reference/ranges.html)或者集合中：

when (x) {

in 1..10 -> print("x is in the range")

in validNumbers -> print("x is valid")

!in 10..20 -> print("x is outside the range")

else -> print("none of the above")

}

另一种可能性是检测一个值是（is）或者不是（!is）一个特定类型的值。注意： 由于[智能转换](https://www.kotlincn.net/docs/reference/typecasts.html" \l "%E6%99%BA%E8%83%BD%E8%BD%AC%E6%8D%A2)，你可以访问该类型的方法与属性而无需任何额外的检测。

fun hasPrefix(x: Any) = when(x) {

is String -> x.startsWith("prefix")

else -> false

}

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

when {

x.isOdd() -> print("x is odd")

y.isEven() -> print("y is even")

else -> print("x+y is odd.")

}

自 Kotlin 1.3 起，可以使用以下语法将 when 的主语（subject，译注：指 when 所判断的表达式）捕获到变量中：

fun Request.getBody() =

when (val response = executeRequest()) {

is Success -> response.body

is HttpError -> throw HttpException(response.status)

}

### 6.3 For 循环

for 循环可以对任何提供迭代器（iterator）的对象进行遍历，这相当于像 C# 这样的语言中的 foreach 循环。语法如下：

for (item in collection) print(item)

循环体可以是一个代码块。

for (item: Int in ints) {

// ……

}

如上所述，for 可以循环遍历任何提供了迭代器的对象。即：

有一个成员函数或者扩展函数 iterator()，它的返回类型

有一个成员函数或者扩展函数 next()，并且

有一个成员函数或者扩展函数 hasNext() 返回 Boolean。

这三个函数都需要标记为 operator。

如需在数字区间上迭代，请使用[区间表达式](https://www.kotlincn.net/docs/reference/ranges.html):

for (i in 1..3) {

println(i)

}

for (i in 6 downTo 0 step 2) {

println(i)

}

对区间或者数组的 for 循环会被编译为并不创建迭代器的基于索引的循环。

如果你想要通过索引遍历一个数组或者一个 list，你可以这么做：

for (i in array.indices) {

println(array[i])

}

或者你可以用库函数 withIndex：

for ((index, value) in array.withIndex()) {

println("the element at $index is $value")

}

### 6.4 While 循环

while 与 do..while 照常使用

while (x > 0) {

x--

}

do {

val y = retrieveData()

} while (y != null) // y 在此处可见

## 返回与跳转

Kotlin 有三种结构化跳转表达式：

**Return**：默认从最直接包围它的函数或者[匿名函数](https://www.kotlincn.net/docs/reference/lambdas.html" \l "%E5%8C%BF%E5%90%8D%E5%87%BD%E6%95%B0)返回。

**Break**：终止最直接包围它的循环。

**Continue**:继续下一次最直接包围它的循环。

所有这些表达式都可以用作更大表达式的一部分：

val s = person.name ?: return

### 7.1 Break 与 Continue 标签

**在 Kotlin 中任何表达式都可以用标签（label）来标记。**

标签的格式为标识符后跟 @ 符号，例如：abc@、fooBar@都是有效的标签（参见[语法](https://www.kotlincn.net/docs/reference/grammar.html" \l "label)）。 要为一个表达式加标签，我们只要在其前加标签即可。

loop@ for (i in 1..100) {

// ……

}

现在，我们可以用标签限制 break 或者continue：

loop@ for (i in 1..100) {

for (j in 1..100) {

if (……) break@loop

}

}

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

### 7.2 返回到标签

fun foo() {

listOf(1, 2, 3, 4, 5).forEach {

if (it == 3) return // 非局部直接返回到 foo() 的调用者

print(it)

}

println("this point is unreachable")

}

这个 return 表达式从最直接包围它的函数即 foo 中返回。 （注意，这种非局部的返回只支持传给[内联函数](https://www.kotlincn.net/docs/reference/inline-functions.html)的 lambda 表达式。） 如果我们需要从 lambda 表达式中返回，我们必须给它加标签并用以限制 return。

fun foo() {

listOf(1, 2, 3, 4, 5).forEach lit@{

if (it == 3) return@lit // 局部返回到该 lambda 表达式的调用者，即 forEach 循环

print(it)

}

print(" done with explicit label")

}

执行结果：1245 done with explicit label

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

fun foo() {

listOf(1, 2, 3, 4, 5).forEach {

if (it == 3) return@forEach // 局部返回到该 lambda 表达式的调用者，即 forEach 循环

print(it)

}

print(" done with implicit label")

}

或者，我们用一个[匿名函数](https://www.kotlincn.net/docs/reference/lambdas.html" \l "%E5%8C%BF%E5%90%8D%E5%87%BD%E6%95%B0)替代 lambda 表达式。

**匿名函数内部的 return 语句将从该匿名函数自身返回**

fun foo() {

listOf(1, 2, 3, 4, 5).forEach(fun(value: Int) {

if (value == 3) return **// 局部返回到匿名函数的调用者，即 forEach 循环**

print(value)

})

print(" done with anonymous function")

}

请注意，前文三个示例中使用的局部返回类似于在常规循环中使用 continue。并没有 break 的直接等价形式，不过可以通过增加另一层嵌套 lambda 表达式并从其中非局部返回来模拟：

fun foo() {

run loop@{

listOf(1, 2, 3, 4, 5).forEach {

if (it == 3) return@loop // 从传入 run 的 lambda 表达式非局部返回

print(it)

}

}

print(" done with nested loop")

}

执行结果：12 done with nested loop

当要返一个回值的时候，解析器优先选用标签限制的 return，即

return@a 1

意为“返回 1 到@a，而不是“返回一个标签标注的表达式而不是  (@a 1)”。