# Kotlin——初级篇（二）：变量、常量、注释

在Kotlin中的变量、常量以及注释多多少少和Java语言是有着不同之处的。不管是变量、常量的定义方式，还是注释的使用。下面详细的介绍Kotlin中的变量、常量、注释的使用。以及和Java的对比。 如果您还没有搭建环境和不了解Kotlin的，请看我的上一篇博文[Kotlin——初级篇（一）：开发环境搭建](https://link.juejin.im/?target=http%3A%2F%2Fwww.cnblogs.com%2FJetictors%2Fp%2F7723018.html)

## 目录

## 一、Kotlin之变量用法

kotlin变量的声明方式与Java中声明变量有很大的区别，而且必须使用var或val关键字。其中：

* var: 用此关键字声明的变量表示可变变量，即可读且可写。相当于Java中普通变量
* val: 用此关键字声明的变量表示不可变变量，即可读且不可写。相当于Java中用final修饰的变量

### 1.1、基础用法

* 定义格式： **关键字 变量名: 数据类型 = xxx**

例：这是在顶层声明的时候

//立即初始化

var var\_a: Int = 10

//推导出类型

var var\_b = 5

//没有初始化的时候，必须声明类型

var var\_c: Float

var\_c = 12.3f

var\_c += 1

println("var\_a => $var\_a \t var\_b => $var\_b \t var\_a => $var\_c")

//立即初始化

val val\_a: Int = 100

//推导出类型

val val\_b = 50

//没有初始化的时候，必须声明类型

val val\_c: Int

val\_c = 1

// val\_c += 1 因为c是常量，所以这句代码是会报错的

println("val\_a => $val\_a \t val\_b => $val\_b \t val\_c => $val\_c")

复制代码

打印结果为：

var\_a => 10 var\_b => 5 var\_a => 13.3

val\_a => 100 val\_b => 50 val\_c => 1

复制代码

* 其中。var和val是Kotlin中定义变量必须使用的关键字。
* 每一行代码的结束可以省略掉分号;，这一点是和Java不同的地方。当然，第一次写可能会有一点不习惯。
* print()与println()都是打印方法，后者打印完成之后会换一行。此两个方法和Java的打印方法是一模一样的。
* $符号表示引用的意思。这里理解为**字符串模板**，在后续的[数据类型](https://link.juejin.im/?target=https%3A%2F%2Fjuejin.im%2Fpost%2F5a36020b6fb9a0451543f5c8)章节下的字符串类型中会讲解到。

### 1.2、在类中声明以及声明可空变量

**1.2.1、类中声明变量**

上面的是演示变量的基础定义。而且只有在顶层声明的情况下是可以不用实例化的。但是在实际开发当中，一般都是在一个类中去定义变量，这种情况被称为声明类的属性。在后面[Kotlin——中级篇（二）：属性与字段详解](https://link.juejin.im/?target=https%3A%2F%2Fjuejin.im%2Fpost%2F5a6378266fb9a01ca10b00e4)章节会详细的讲解。这里讲解其声明的特点并实例分析。

其特点如下：**必须初始化，如果不初始化，需使用lateinit关键字。参见1.3节。**

例：

class Test1{

// 定义属性

var var\_a : Int = 0

val val\_a : Int = 0

// 初始化

init {

var\_a = 10

// val\_a = 0 为val类型不能更改。

println("var\_a => $var\_a \t val\_a => $val\_a")

}

}

Test1()

复制代码

输出结果为：

var\_a => 10 val\_a => 0

复制代码

其中：在上面的例子中，大家只要关心类中声明变量的用法就可以了。其他的不用关心，这里只是为了能看到程序运行结果。

**1.2.2、声明可空变量**

在Java中，当我们声明一个变量不必关心这个变量是否为空，在使用这个变量的时候几乎上都会判断其是否为空增加程序的安全性。这样的习惯是极好的。但是无形中也增加了一定的代码量。有时候这样的代码还极有可能是无用的废代码。然而在Kotlin中当我们可以确定这个属性或变量一定不为空时，我们就用上面讲解到的去定义变量。否则就把它声明为可空变量。

可空变量的特点：

* 在声明的时候一定用标准的声明格式定义。不能用可推断类型的简写。
* 变量类型后面的?符号不能省略。不然就和普通的变量没区别了。
* 其初始化的值可以为null或确定的变量值。

定义：

var/val 变量名 ： 类型? = null/确定的值

复制代码

例：

class Test2{

// 声明可空变量

var var\_a : Int? = 0

val val\_a : Int? = null

init {

var\_a = 10

// val\_a = 0 为val类型不能更改。

println("var\_a => $var\_a \t val\_a => $val\_a")

}

}

Test2()

复制代码

输出结果为：

var\_a => 10 val\_a => null

复制代码

关于可空变量的定义就上面那么多。但是在使用可空变量的时候就没有Java中那么复杂了。关于可空变量的使用请参见我的[Kotlin——初级篇（六）： 可空类型、空安全（null）、类型转换等特性总结](https://link.juejin.im/?target=https%3A%2F%2Fjuejin.im%2Fpost%2F5a5b06f26fb9a01cb42c5206)这篇文章。

### 1.3、后期初始化与延迟初始化

在1.2节中，当在类中定义一个变量（属性）的时候是必须初始化的。这在平时的实际开发中能满足大部分的需求。但是还是有一些特殊的场景中不能满足。比如说：Android开发中对组件变量的声明与赋值，以及在使用Dagger2注解变量等。这就需要Kotlin中特有的后期初始化属性来满足这个需求了。当然这里还为大家讲解延迟初始化，在实际的开发中也是很有用处的。

**1.3.1、后期初始化属性**

声明后期初始化属性的特点：

* 使用lateinit关键字
* 必须是可读且可写的变量，即用var声明的变量
* 不能声明于可空变量。
* 不能声明于基本数据类型变量。例：Int、Float、Double等，注意：String类型是可以的。
* 声明后，在使用该变量前必须赋值，不然会抛出UninitializedPropertyAccessException异常。

实例讲解：举一个Android中常见的例子

// 声明组件

private lateinit var mTabLayout : TabLayout

lateinit var a : Int // 会报错。因为不能用于基本数据类型。

// 赋值

mTabLayout = find(R.id.home\_tab\_layout)

// 使用

mTabLayout.setupWithViewPager(mViewPager)

复制代码

**1.3.2、延迟初始化属性**

所谓延迟初始化即：指当程序在第一次使用到这个变量（属性）的时候在初始化。

声明延迟初始化属性的特点：

* 使用lazy{}高阶函数，不能用于类型推断。且该函数在变量的数据类型后面，用by链接。
* 必须是只读变量，即用val声明的变量。

实例讲解：同样是Android中常见的例子

// 声明一个延迟初始化的字符串数组变量

private val mTitles : Array<String> by lazy {

arrayOf(

ctx.getString(R.string.tab\_title\_android),

ctx.getString(R.string.tab\_title\_ios),

ctx.getString(R.string.tab\_title\_h5)

)

}

// 声明一个延迟初始化的字符串

private val mStr : String by lazy{

"我是延迟初始化字符串变量"

}

复制代码

## 二、Kotlin之常量的用法

Kotlin中声明常量的方式和在Java中声明常量的方式有很大的区别。这里举例说明：

Kotlin中使用val时候对应的Java代码：

Kotlin中的 val numA = 6 等价于 Java中的：public final int numA = 6

复制代码

很显然，Kotlin中只用val修饰还不是常量，它只能是一个不能修改的变量。那么常量怎么定义呢？其实很简单，在val关键字前面加上const关键字。

即：

const val NUM\_A = 6

复制代码

其特点：**const只能修饰val，不能修饰var**

**声明常量的三种正确方式**

1. 在顶层声明
2. 在object修饰的类中声明，在kotlin中称为**对象声明**，它相当于Java中一种形式的单例类
3. 在伴生对象中声明

举例说明：

// 1. 顶层声明

const val NUM\_A : String = "顶层声明"

// 2. 在object修饰的类中

object TestConst{

const val NUM\_B = "object修饰的类中"

}

// 3. 伴生对象中

class TestClass{

companion object {

const val NUM\_C = "伴生对象中声明"

}

}

fun main(args: Array<String>) {

println("NUM\_A => $NUM\_A")

println("NUM\_B => ${TestConst.NUM\_B}")

println("NUM\_C => ${TestClass.NUM\_C}")

}

复制代码

输出结果为：

NUM\_A => 顶层声明

NUM\_B => object修饰的类中

NUM\_C => 伴生对象中声明

复制代码

## 三、Kotlin之注释

Kotlin中的注释几乎和Java没什么区别。唯一的区别在于Kotlin中的多行注释中可以嵌套多行注释，而Java中是不能的。

* 单行注释

两个斜杠开头表示单行注释（//）

举例：

// 1. 单行注释

复制代码

* 多行注释（块注释）

以斜杠加星号开头（/\*），同时以星号加斜杠结尾（\*/），中间这是要注释的代码块！

举例：

/\*

2. 多行注释（块注释）

我是多行注释

\*/

复制代码

* 多行注释嵌套

kotlin中块注释的级联使用，其实个人觉得块注释的嵌套使用的意义不大，不过从视觉上确实能给人一种层次感

举例：

/\*

第一层块注释

/\*

第二层块注释

/\*

第三层快注释

这种注释方式在java中是不支持的，但是在kotlin中是支持的。算是一个亮点吧（貌似意义不大）。

\*/

\*/

\*/

复制代码

注：在Java中使用上面的注释代码直接报错。

* 类注释、方法注释

和Java是一样的

举例：

/\*\*

\* 3. 方法的注释（同java一样）

\*/

fun testMethodNote(){

}

**Kotlin——初级篇（三）：数据类型详解**

任意一种开发语言都有其数据类型，并且数据类型对于一门开发语言来说是最基本的构成，同时也是最基础的语法。当然，Kotlin也不例外。Kotlin的数据类型和Java是大致相同的，但是他们的写法不同，并且类型之间的转换也存在着差异。下面为大家详细说明并举例。

**目录**

**一、数值类型**

**1、Kotlin中的数字的内置类型（接近与Java），其关键字为：**

* Byte=> 字节 => 8位
* Short => 短整型 => 16位
* Int => 整型 => 32位
* Long => 长整型 => 64位
* Float => 浮点型 => 32位
* Double => 双精度浮点型 => 64位

例：

var a: Byte = 2

var b: Short = 2

var c: Int = 2

var d: Long = 2L //长整型由大写字母L标记

var e: Float = 2f //单精度浮点型由小写字母f或大写字符F标记

var f: Double = 2.0

println(" a => $a \n b => $b \n c => $c \n d => $d \n e => $e \n f => $f);

复制代码

输出结果为：

a => 2

b => 2

c => 2

d => 2

e => 2.0

f => 2.0

复制代码

**2、进制数**

* 二进制数
* 八进制数（Kotlin不支持）
* 十进制数
* 十六进制数

例：

var g = 0x0F //十六进制数

var h = 0b00001011 //二进制数

var k = 123 //十进制数

// ps：Kotlin不支持八进制数

println(" g => $g \n h => $h \n k => $k);

复制代码

输出结果为：

g => 15

h => 11

k => 123

复制代码

**3、数字类型字面常量的下划线**

**作用：分割数字进行分组，使数字常量更易读**

例：

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

println("oneMillion => $oneMillion")

println("creditCardNumber => $creditCardNumber")

println("socialSecurityNumber => $socialSecurityNumber")

println("hexBytes => $hexBytes")

println("bytes => $bytes")

复制代码

输出结果为：

oneMillion => 1000000

creditCardNumber => 1234567890123456

socialSecurityNumber => 999999999

hexBytes => 4293713502

bytes => 3530134674

复制代码

**4、装箱与拆箱**

* **装箱与拆箱**
* 在Kotlin中，存在数字的装箱，但是不存在拆箱。因为Kotlin是没有基本数据类型的，Kotlin是万般皆对象的原则。故**不存在和Java中的类似int是数据类型，Integer是整型的引用类型。**

在Kotlin中要实现装箱操作。首先要了解可空引用。即类似Int?(只限数值类型)这样的。

例：

val numValue: Int = 123

//装箱的过程，其实装箱之后其值是没有变化的

val numValueBox: Int? = numValue

println("装箱后： numValueBox => $numValueBox")

复制代码

输出结果为：

装箱后： numValueBox => 123

复制代码

* **两个数值的比较**

判断两个数值是否相等（==）,判断两个数值在内存中的地址是否相等（===）,其实上面的装箱操作之后其内存中的地址根据其数据类型的数值范围而定。

例：

val numValue: Int = 128

val numValueBox: Int? = numValue

/\*

比较两个数字

\*/

var result: Boolean

result = numValue == numValueBox

println("numValue == numValueBox => $result") // => true,其值是相等的

result = numValue === numValueBox

/\*

上面定义的变量是Int类型，大于127其内存地址不同，反之则相同。

这是`kotlin`的缓存策略导致的，而缓存的范围是` -128 ~ 127 `。

故，下面的打印为false

\*/

println("numValue === numValueBox => $result")

复制代码

输出结果为：

numValue == numValueBox => true

numValue === numValueBox => false

复制代码

**Ps:各位可以试试将变量numValue的值改为在-128 ~ 127这个区间的数字试试**

**5、转换**

* **显式转换**
* 较小的类型不会被隐式转换为更大的类型，故而系统提供了显式转换。提供的显式转换方法如下：
  1. toByte() => 转换为字节型
  2. toShort() => 转换为短整型
  3. toInt() => 转换为整型
  4. toLong() => 转换为长整型
  5. toFloat() => 转换为浮点型
  6. toDouble() => 转换为双精度浮点型
  7. toChar() => 转换为字符型
  8. toString() => 转换为字符串型

例：

var numA: Int = 97

println(numA.toByte())

println(numA.toShort())

println(numA.toInt())

println(numA.toLong())

println(numA.toFloat())

println(numA.toDouble())

println(numA.toChar())

println(numA.toString())

复制代码

输出结果为：

97

97

97

97.0

97.0

97

a

97

复制代码

* **隐式转换**

类型是从上下文推断出来的，即算术运算则被重载为适当的转换

例：

// 30L + 12 -> Long + Int => Long

val num = 30L + 12

print(num)

复制代码

输出结果为：

42

复制代码

**6、位运算符**

* Kotlin中对于按位操作，和Java是有很大的差别的。Kotlin中没有特殊的字符，但是只能命名为可以以中缀形式调用的函数，下列是按位操作的完整列表(仅适用于整形（Int）和长整形（Long）)：
  1. shl(bits) => 有符号向左移 (类似Java的<<)
  2. shr(bits) => 有符号向右移 (类似Java的>>)
  3. ushr(bits) => 无符号向右移 (类似Java的>>>)
  4. and(bits) => 位运算符 and (同Java中的按位与)
  5. or(bits) => 位运算符 or (同Java中的按位或)
  6. xor(bits) => 位运算符 xor (同Java中的按位异或)
  7. inv() => 位运算符 按位取反 (同Java中的按位取反)

下面附上Kotlin中关于位操作符的源码：

/\*\* Shifts this value left by [bits]. \*/

public infix fun shl(bitCount: Int): Int

/\*\* Shifts this value right by [bits], filling the leftmost bits with copies of the sign bit. \*/

public infix fun shr(bitCount: Int): Int

/\*\* Shifts this value right by [bits], filling the leftmost bits with zeros. \*/

public infix fun ushr(bitCount: Int): Int

/\*\* Performs a bitwise AND operation between the two values. \*/

public infix fun and(other: Int): Int

/\*\* Performs a bitwise OR operation between the two values. \*/

public infix fun or(other: Int): Int

/\*\* Performs a bitwise XOR operation between the two values. \*/

public infix fun xor(other: Int): Int

/\*\* Inverts the bits in this value. \*/

public fun inv(): Int

复制代码

例：

/\*

位运算符

支持序列如下：shl、shr、ushr、and、or、xor、inv

\*/

var operaNum: Int = 4

var shlOperaNum = operaNum shl(2)

var shrOperaNum = operaNum shr(2)

var ushrOperaNum = operaNum ushr(2)

var andOperaNum = operaNum and(2)

var orOperaNum = operaNum or(2)

var xorOperaNum = operaNum xor(2)

var invOperaNum = operaNum.inv()

println("shlOperaNum => $shlOperaNum \n " +

"shrOperaNum => $shrOperaNum \n " +

"ushrOperaNum => $ushrOperaNum \n " +

"andOperaNum => $andOperaNum \n " +

"orOperaNum => $orOperaNum \n " +

"xorOperaNum => $xorOperaNum \n " +

"invOperaNum => $invOperaNum")

复制代码

输出结果为：

shlOperaNum => 16

shrOperaNum => 1

ushrOperaNum => 1

andOperaNum => 0

orOperaNum => 6

xorOperaNum => 6

invOperaNum => -5

复制代码

**二、布尔类型（Boolean）**

**1、关键字**

Boolean关键字表示布尔类型，并且其值有true和false

例：

var isNum: Boolean

isNum = false

println("isNum => $isNum")

复制代码

输出结果为：

isNum => false

复制代码

**2、逻辑操作符（与Java相同）**

* ' || ' => 逻辑或（或者）
* ' && ' => 逻辑与（并且）
* ' ! ' => 逻辑非（取反）

例：

/\*

操作运算符

' || ' => 逻辑或（或者）

' && ' => 逻辑与（并且）

' ! ' => 逻辑非（取反）

\*/

var a: Boolean = false

var b: Boolean = true

var result: Boolean

/\* 逻辑或操作 \*/

if (a || b){

result = a || b

println("a || b => $result")

}

/\* 逻辑与操作 \*/

if (a && b){

result = a && b

println("a && b => $result")

}

/\* 逻辑非操作 \*/

result = !a

println("!a => $result")

result = !b

println("!b => $result")

复制代码

输出结果为：

isNum => false

a || b => true

!a => true

!b => false

复制代码

**三、字符型（Char）**

**1、关键字**

Char为表示字符型，字符变量用单引号（‘ ’）表示。并且不能直接视为数字，不过可以显式转换为数字。

例：

var char1: Char

char = 'a'

//char1 = 1 => 这句代码会直接出错

println("char1 => $char1")

复制代码

输出结果为：

char1 => a

复制代码

**2、显示转换为其他类型**

字符型的变量不仅可以转换为数字，同时也可转换为其他类型

例：

var var1 = char1.toByte()

var var2 = char1.toInt()

var var3 = char1.toString()

var var4 = char1.toFloat()

var var5 = char1.toShort()

println("var1 => $var1 \n var2 => $var2 \n var3 => $var3 \n var4 => $var4 \n var5 => $var5")

复制代码

输出结果为：

var1 => 97

var2 => 97

var3 => a

var4 => 97.0

var5 => 97

复制代码

PS:除了可以转换类型外，当变量为英文字母时还支持大小写转换。

例：

/\*

当字符变量为英文字母时，大小写的转换

\*/

var charA: Char = 'a'

var charB: Char = 'B'

var charNum: Char = '1'

var result: Char

// 转换为大写

result = charA.toUpperCase()

println("result => $result")

// 转换为小写

result = charB.toLowerCase()

println("result => $result")

//当字符变量不为英文字母时，转换无效

result = charNum.toLowerCase()

println("result => $result")

复制代码

输出结果为：

result => A

result => b

result => 1

复制代码

**3、字符转义** 同Java一样，使用某些特殊的字符时，要使用转义。下列是支持的转义序列：

* \t => 表示制表符
* \n => 表示换行符
* \b => 表示退格键（键盘上的Back建）
* \r => 表示键盘上的Enter键
* \\ => 表示反斜杠
* \' => 表示单引号
* \" => 表示双引号
* \$ => 表示美元符号，如果不转义在kotlin中就表示变量的引用了
* 其他的任何字符请使用Unicode转义序列语法。例：'\uFF00'

例：

println("\n 换行符")

println("\t 制表符")

println(" \b 退格键")

println("\r Enter键同样换行")

println('\\')

println('\'')

println('\"')

println('\$')

println('\uFF01')

复制代码

输出结果为：

换行符

制表符

退格键

Enter键同样换行

\

'

"

$

！

复制代码

**四、字符串类型（String）**

**1、关键字**

String表示字符串类型。其是不可变的。所以字符串的元素可以通过索引操作的字符：str[index]来访问。可以使用for循环迭代字符串： 其中str[index]中的str为要目标字符串，index为索引

例：

val str: String = "kotlin"

println("str => $str")

//迭代

for (s in str){

print(s)

print("\t")

}

复制代码

输出结果为：

str => kotlin

k o t l i n

复制代码

**2、 字符串字面量**

在Kotlin中， 字符串字面量有两种类型：

* 包含转义字符的字符串 转义包括（\t、\n等）,不包含转义字符串的也同属此类型
* 包含任意字符的字符串 由三重引号（""" .... """）表示

例：

// 类型1：

var str1: String = "hello\t\tkotlin"

println(str1)

str1 = "hello kotlin"

println(str1)

// 类型2：

val str2 = """ fun main(args: Array<String>){

println("我是三重引号引用的字符串，我可以包含任意字符")

} """

println(str2)

复制代码

输出结果为：

hello kotlin

hello kotlin

fun main(args: Array<String>){

println("我是三重引号引用的字符串，我可以包含任意字符")

}

复制代码

PS： 可以使用trimMargin()函数删除前导空格 ，默认使用符号(|)作为距前缀，当然也可以使用其他字符。例：右尖括号（>）、左尖括号（<）等。

例：

val str3: String = """

> I`m like Kotlin .

> I`m like Java .

> I`m like Android .

> I`m like React-Native.

""".trimMargin(">")

println(str3)

复制代码

输出结果为：

I`m like Kotlin .

I`m like Java .

I`m like Android .

I`m like React-Native.

复制代码

**3、字符串模板**

使用字符串模板的符号为（$）。在$符号后面加上变量名或大括号中的表达式

例：

val text1: String = "我来了！"

var text2: String = "$text1 kotlin"

var text3: String = "$text2 ${text1.length} 哈哈！！！！"

println(text1)

println(text2)

println(text3)

复制代码

输出结果为：

我来了！

我来了！ kotlin

我来了！ kotlin 4 哈哈！！！！

复制代码

**五、数组型（Array）**

* Kotlin中数组由Array<T>表示，可以去看看源码实现，里面就几个方法
* 创建数组的3个函数
  1. arrayOf()
  2. arrayOfNulls()
  3. 工厂函数（Array()）

**1、arrayOf()**

创建一个数组，参数是一个可变参数的泛型对象

例：

var arr1 = arrayOf(1,2,3,4,5) //等价于[1,2,3,4,5]

for (v in arr1){

print(v)

print("\t")

}

var arr2 = arrayOf("0","2","3",'a',32.3f)

for (v in arr2){

print(v)

print("\t")

}

复制代码

输出结果为：

1 2 3 4 5

0 2 3 a 32.3

复制代码

**2、arrayOfNulls()**

用于创建一个指定数据类型且可以为空元素的给定元素个数的数组

例：

var arr3 = arrayOfNulls<Int>(3)

//如若不予数组赋值则arr3[0]、arr3[1]、arr3[2]皆为null

for(v in arr3){

print(v)

print("\t")

}

println()

//为数组arr3赋值

arr3[0] = 10

arr3[1] = 20

arr3[2] = 30

for(v in arr3){

print(v)

print("\t")

}

复制代码

输出结果为：

null null null

10 20 30

复制代码

**3、工厂函数**

* 使用一个工厂函数Array()，它使用数组大小和返回给定其索引的每个数组元素的初始值的函数。
* Array() => 第一个参数表示数组元素的个数，第二个参数则为使用其元素下标组成的表达式

例：

var arr4 = Array(5,{index -> (index \* 2).toString() })

for (v in arr4){

print(v)

print("\t")

}

复制代码

输出结果为：

0 2 4 6 8

复制代码

**4、原始类型数组**

* Kotlin还有专门的类来表示原始类型的数组，没有装箱开销，它们分别是：
  1. ByteArray => 表示字节型数组
  2. ShortArray => 表示短整型数组
  3. IntArray => 表示整型数组
  4. LongArray => 表示长整型数组
  5. BooleanArray => 表示布尔型数组
  6. CharArray => 表示字符型数组
  7. FloatArray => 表示浮点型数组
  8. DoubleArray => 表示双精度浮点型数组
* PS: **Kotlin中不支持字符串类型这种原始类型数组，可以看源码Arrays.kt这个类中并没有字符串数组的声明。而源码中StringArray.kt这个类并不是声明字符串型数组的。**

下面的例子只演示了几种，其他的类似。 例：

var intArr: IntArray = intArrayOf(1,2,3,4,5)

for (number in intArr){

print(number)

print("\t")

}

println()

var charArr: CharArray = charArrayOf('a','1','b','c','3','d')

for (char in charArr){

print(char)

print("\t")

}

println()

var longArr: LongArray = longArrayOf(12L,1254L,123L,111L)

for (long in longArr){

print(long)

print("\t")

}

复制代码

输出结果为：

1 2 3 4 5

a 1 b c 3 d

12 1254 123 111

**Kotlin——初级篇（四）：控制语句（if、for、while、when、do...while、跳转语句）详解**

在前面 的章节中讲解了Kotlin语言中的数据类型、变量与常量的定义。不了解请参见前面的内容：

1. [Kotlin——初级篇（三）：数据类型详解](https://link.juejin.im/?target=https%3A%2F%2Fjuejin.im%2Fpost%2F5a36020b6fb9a0451543f5c8)。
2. [Kotlin——初级篇（二）：变量、常量、注释](https://link.juejin.im/?target=https%3A%2F%2Fjuejin.im%2Fpost%2F5a39ef7af265da4311205967)。

下面详细为大家讲解Kotlin中的控制语句使用。不得不说其和Java中还是有很多不一样的地方。

**目录**

**一、if语句**

在Kotlin中的if语句和Java还是还是有一定的区别的，它能在Java中更灵活，除了能实现Java写法外，还可以实现表达式（实现三元运算符），及作为一个块的运用。

**1、传统写法（同Java写法一样）**

例：

var numA = 2

if (numA == 2){

println("numA == $numA => true")

}else{

println("numA == $numA => false")

}

复制代码

输出结果为：

numA == 2 => true

复制代码

**2、Kotlin中的三元运算符**

* 在Kotlin中其实是不存在三元运算符(condition ? then : else)这种操作的。
* 那是因为if语句的特性(if表达式会返回一个值)故而不需要三元运算符。

例：

// 在Java中可以这么写，但是Kotlin中直接会报错。

// var numB: Int = (numA > 2) ? 3 : 5

// kotlin中直接用if..else替代。例：

var numB: Int = if ( numA > 2 ) 3 else 5 // 当numA大于2时输出numB的值为3，反之为5

println("numB = > $numB")

复制代码

输出结果为：

numB = > 3

复制代码

由上可以看出，Kotlin中的if可以作为一个表达式并返回一个值。

**3、作为一个块结构，并且最后一句表达式为块的值**

例：

var numA: Int = 2

var numC: Int = if (numA > 2){

numA++

numA = 10

println("numA > 2 => true")

numA

}else if (numA == 2){

numA++

numA = 20

println("numA == 2 => true")

numA

}else{

numA++

numA = 30

println("numA < 2 => true")

numA

}

// 根据上面的代码可以看出，每一个if分支里面都是一个代码块，并且返回了一个值。根据条件numC的值应该为20

println("numC => $numC")

复制代码

输出结果为：

numA == 2 => true

numC => 20

复制代码

**二、for语句**

* Kotlin废除了Java中的for(初始值;条件；增减步长)这个规则。但是Kotlin中对于for循环语句新增了其他的规则，来满足刚提到的规则。
* for循环提供迭代器用来遍历任何东西
* for循环数组被编译为一个基于索引的循环，它不会创建一个迭代器对象

**1、新增的规则，去满足for(初始值;条件;增减步长)这个规则**

* **1.1、递增**

关键字：until 范围：until[n,m) => 即大于等于n,小于m

例：

// 循环5次，且步长为1的递增

for (i in 0 until 5){

print("i => $i \t")

}

复制代码

输出结果为

i => 0 i => 1 i => 2 i => 3 i => 4

复制代码

* **1.2、递减**
* 关键字：downTo
* 范围：downTo[n,m] => 即小于等于n,大于等于m ,n > m

例：

// 循环5次，且步长为1的递减

for (i in 15 downTo 11){

print("i => $i \t")

}

复制代码

输出结果为：

i => 15 i => 14 i => 13 i => 12 i => 11

复制代码

**1.3、符号（' .. '） 表示递增的循环的另外一种操作**

* 使用符号( '..').
* 范围：..[n,m]=> 即大于等于n，小于等于m
* 和until的区别，一是简便性。二是范围的不同。

例：

print("使用 符号`..`的打印结果\n")

for (i in 20 .. 25){

print("i => $i \t")

}

println()

print("使用until的打印结果\n")

for (i in 20 until 25){

print("i => $i \t")

}

复制代码

输出结果为：

使用 符号`..`的打印结果

i => 20 i => 21 i => 22 i => 23 i => 24 i => 25

使用until的打印结果

i => 20 i => 21 i => 22 i => 23 i => 24

复制代码

**1.4、设置步长**

关键字：step

例：

for (i in 10 until 16 step 2){

print("i => $i \t")

}

复制代码

输出结果为：

i => 10 i => 12 i => 14

复制代码

**2、迭代**

* for循环提供一个迭代器用来遍历任何东西。
* for循环数组被编译为一个基于索引的循环，它不会创建一个迭代器对象

**2.1、遍历字符串**

此用法在数据类型章节中的字符串类型中用到过。还不甚清楚的可以查看 [Kotlin——初级篇（三）：数据类型详解](https://link.juejin.im/?target=https%3A%2F%2Fjuejin.im%2Fpost%2F5a36020b6fb9a0451543f5c8)。

例：

for (i in "abcdefg"){

print("i => $i \t")

}

复制代码

输出结果为：

i => a i => b i => c i => d i => e i => f i => g

复制代码

**2.2、遍历数组**

此用法在数据类型章节中的数组类型中用到过。还不甚清楚的可以查看 [Kotlin——初级篇（三）：数据类型详解](https://link.juejin.im/?target=https%3A%2F%2Fjuejin.im%2Fpost%2F5a36020b6fb9a0451543f5c8)。

例：

var arrayListOne = arrayOf(10,20,30,40,50)

for (i in arrayListOne){

print("i => $i \t")

}

复制代码

输出结果为：

i => 10 i => 20 i => 30 i => 40 i => 50

复制代码

**2.3、使用数组的indices属性遍历**

例：

var arrayListTwo = arrayOf(1,3,5,7,9)

for (i in arrayListTwo.indices){

println("arrayListTwo[$i] => " + arrayListTwo[i])

}

复制代码

输出结果为：

arrayListTwo[0] => 1

arrayListTwo[1] => 3

arrayListTwo[2] => 5

arrayListTwo[3] => 7

arrayListTwo[4] => 9

复制代码

**2.4、使用数组的withIndex()方法遍历**

例：

var arrayListTwo = arrayOf(1,3,5,7,9)

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

println("index => $index \t value => $value")

}

复制代码

输出结果为：

index => 0 value => 1

index => 1 value => 3

index => 2 value => 5

index => 3 value => 7

index => 4 value => 9

复制代码

**2.5、使用列表或数组的扩展函数遍历**

* 数组或列表有一个成员或扩展函数iterator()实现了Iterator<T>接口，且该接口提供了next()与hasNext()两个成员或扩展函数
* 其一般和while循环一起使用

1. 可以查看Array.kt这个类。可以看见其中的iterator()函数，而这个函数实现了Iterator接口。
2. /\*\*
3. \* Creates an iterator for iterating over the elements of the array.
4. \*/
5. public operator fun iterator(): Iterator<T>

复制代码

1. 查看Iterator.kt这个接口类，这个接口提供了hasNext()函数和next()函数。
2. public interface Iterator<out T> {
3. /\*\*
4. \* Returns the next element in the iteration.
5. \*/
6. public operator fun next(): T
7. /\*\*
8. \* Returns `true` if the iteration has more elements.
9. \*/
10. public operator fun hasNext(): Boolean
11. }

复制代码

例：

var arrayListThree = arrayOf(2,'a',3,false,9)

var iterator: Iterator<Any> = arrayListThree.iterator()

while (iterator.hasNext()){

println(iterator.next())

}

复制代码

输出结果为：

2

a

3

false

9

复制代码

终上所述就是for循环语句常用的用法。

**三、when语句**

* 在Kotlin中已经废除掉了Java中的switch语句。而新增了when(exp){}语句。
* when语句不仅可以替代掉switch语句，而且比switch语句更加强大

**3.1、when语句实现switch语句功能**

例：

when(5){

1 -> {

println("1")

}

2 -> println("2")

3 -> println("3")

5 -> {

println("5")

}

else -> {

println("0")

}

}

复制代码

输出结果为：

5

复制代码

**3.2、和逗号结合使用，相当于switch语句中的不使用break跳转语句**

例：

when(1){

// 即x = 1,2,3时都输出1。

1 , 2 , 3 -> {

println("1")

}

5 -> {

println("5")

}

else -> {

println("0")

}

}

复制代码

输出结果为：

1

复制代码

**3.3、条件可以使用任意表达式，不仅局限于常量**

相当于if表达式的用法。

例：

var num:Int = 5

when(num > 5){

true -> {

println("num > 5")

}

false ->{

println("num < 5")

}

else -> {

println("num = 5")

}

}

复制代码

输出结果为：

num < 5

复制代码

**3.4、 检查值是否存在于集合或数组中**

* 操作符：
  1. （in） 在
  2. (!in) 不在
* 限定:只适用于数值类型

例：

var arrayList = arrayOf(1,2,3,4,5)

when(1){

in arrayList.toIntArray() -> {

println("1 存在于 arrayList数组中")

}

in 0 .. 10 -> println("1 属于于 0~10 中")

!in 5 .. 10 -> println("1 不属于 5~10 中")

else -> {

println("都错了 哈哈！")

}

}

复制代码

输出结果为：

元素`1`存在于 arrayList数组中

复制代码

其中，符号( .. )表示至的意思。如例子中的0 .. 10就表示0至10或者0到10。

**3.5、检查值是否为指定类型的值**

* 操作符
  1. 是（is）
  2. 不是（!is）
* 值得注意的是，Kotlin的智能转换可以访问类型的方法和属性

例：

when("abc"){

is String -> println("abc是一个字符串")

else -> {

println("abc不是一个字符串")

}

}

// 智能转换

var a: Int = 2

when(a){

!is Int -> {

println("$a 不是一个Int类型的值")

}

else -> {

a = a.shl(2)

println("a => $a")

}

}

复制代码

输出结果为：

abc是一个字符串

a => 8

复制代码

**3.6、不使用表达式的when语句**

表示为最简单的布尔表达式

例：

var array = arrayOfNulls<String>(3)

when{

true -> {

for (i in array){

print(" $i \t")

}

println()

}

else -> {

}

}

复制代码

输出结果为：

null null null

复制代码

综上所述，为Kotlin中when控制语句的常见用法。可以看出它的强大。以及便利性。不仅可以替代掉Java语句中的swicth语句。甚至可以替换掉if语句。

**四、while语句**

* 其同Java中的while循环一样。在此不做累述。
* 定义格式：

while(exp){ 其中exp为表达式

...

}

复制代码

例：

var num = 5

var count = 1

while (num < 10){

println("num => $num")

println("循环了$count 次")

count++

num++

}

复制代码

输出结果为：

num => 5

循环了1 次

num => 6

循环了2 次

num => 7

循环了3 次

num => 8

循环了4 次

num => 9

循环了5 次

复制代码

**五、do...while语句**

* 其同Java中的do...while循环一样。在此不做累述。
* 定义格式：

do(exp){ // 其中exp为表达式

...

}(while)

复制代码

例：

var num = 5

var count = 1

do {

println("num => $num")

println("循环了$count 次")

count++

num++

}while (num < 10)

复制代码

输出结果为：

num => 5

循环了1 次

num => 6

循环了2 次

num => 7

循环了3 次

num => 8

循环了4 次

num => 9

循环了5 次

复制代码

**PS: do{...}while(exp)与while(exp){...}最大的区别是do{...}while(exp)最少执行一次**，这点也是和Java相同的

例：

var num = 5

var count = 1

do {

println("num => $num")

println("循环了$count 次")

count++

num++

}while (num < 5)

复制代码

输出结果为：

num => 5

循环了1 次

复制代码

**六、跳转语句（return、break、continue）**

其同Java中的跳转语句一样。在此不做累述。

**1、return语句**

默认情况下，从最近的封闭函数或匿名函数返回。

例：

fun returnExample(){

var str: String = ""

if (str.isBlank()){

println("我退出了该方法")

return

}

}

复制代码

输出结果为：

我退出了该方法

复制代码

**2、break语句**

作用：终止最近的闭合循环。

例：

var count: Int = 1

for (i in 1 until 10){

if (i == 5){

println("我在第$i 次退出了循环")

break

}

count++

}

println("我循环了多少次：count => $count")

复制代码

输出结果为：

我在第5 次退出了循环

我循环了多少次：count => 5

复制代码

**3、continue语句**

前进到最近的封闭循环的下一个步骤(迭代)。

例：

for (i in 1 until 10){

if (i == 5){

println("我跳过了第$i 次循环")

continue

}

println("i => $i")

}

复制代码

输出结果为：

i => 1

i => 2

i => 3

i => 4

我跳过了第5 次循环

i => 6

i => 7

i => 8

i => 9

# Kotlin——初级篇（五）：操作符与操作符重载一

本篇文章为大家详细的介绍Koltin特有的操作符重载。或许对于有编程经验的朋友来说，操作符这个词绝对不陌生，就算没有任何编辑基础的朋友，数学中的算数运算符也绝不陌生。例如（+、-、\*、/、>、<、>=、<=）等。而算数运算符是编程语言中的一种操作符而已。就算你没有任何基础，也请你详细的看完这篇文章，我相信你会很有收获的。

## 目录

## 一、约定

所谓预定：即指Kotlin允许我们为自己的类型提供预定义的一组操作符的实现。这些操作符具有固定的符号表示（如 + 或 \*)和固定的优先级。为实现这样的操作符,我们为相应的**操作类型**提供了一个固定名字的[函数](https://link.juejin.im/?target=https%3A%2F%2Fjuejin.im%2Fpost%2F5a6377425188257329148665)。这样的技术，称为**约定**

因为由类实现的接口集是固定的，而Kotlin不能为了实现其他接口而修改现有的类，因此一般通过**扩展函数的机制**来实现为现有的类增添新的**约定方法**，从而适应任何现有的Java类。

## 二、操作符与操作符重载

根据操作数据个数的不同，分为两种操作类型：

1. 一元操作：即指操作数只有一个的情况
2. 二元操作：即指操作数存在二两或多个的情况。**特别说明**：在存在多个操作数的情况下，会用**复合运算**或拆分为多个运算。

### 2.1、一元操作

一元操作:即指一个操作数的情况，

##### 2.1.1、简单的一元操作运算

这里分为三种情况有三种一元操作:

* + 表示为操作数实现一个正号的意思，其操作数为数值型
* - 表示为操作数实现一个负号的意思，其操作数为数值型
* ! 表示取反的意思，其操作数为boolean类型

提供一个表格直观的展示：

| **操作符** | **重载** |
| --- | --- |
| +a | a.unaryPlus() |
| -a | a.unaryMinus() |
| !a | a.not() |

例：

var a = 1

var b = -2

var c = true

var d = false

// 操作符实现

println("+a = ${+a}\t -a = ${-a}\t !c = ${!c}")

println("+b = ${+b}\t -b = ${-b}\t !d = ${!d}")

// 操作符重载实现

println("+a = ${a.unaryPlus()}\t -a = ${a.unaryMinus()}\t !c = ${c.not()}")

println("+b = ${b.unaryPlus()}\t -b = ${b.unaryMinus()}\t !d = ${d.not()}")

复制代码

输出结果为：

+a = 1 -a = -1 !c = false

+b = -2 -b = 2 !d = true

+a = 1 -a = -1 !c = false

+b = -2 -b = 2 !d = true

复制代码

##### 2.1.2、复杂的一元操作

复杂的一元操作符即指，对操作数进行自增、自减操作。和Java是一样的

这里主要有4种情况：

* 后缀自增：表示为操作数进行自增操作，其操作数为数值型。例如：a++
* 后缀自减：表示为操作数进行自减操作，其操作数为数值型。例如：a--
* 前缀自增：表示为操作数进行自增操作，其操作数为数值型。例如：++a
* 前缀自减：表示为操作数进行自增操作，其操作数为数值型。例如：--a

提供一个表格直观的展示：

| **操作符** | **重载** | **表示** |
| --- | --- | --- |
| a++ | a.inc() | a = a.also{ a.inc() } |
| a-- | a.dec() | a = a.also{ a.dec() } |
| ++a | a.inc() | a = a.inc().also{ a = it } |
| --a | a.dec() | a = a.dec().also{ a = it } |

解释：操作符++的重载为inc(),操作符--的重载为dec()。但是前缀操作和后缀操作是有着明显的区别的：

* 后缀操作是第一次调用的时候不执行自身。在第二次开始进行自增或自减操作。
* 前缀操作是第一次调用的时候就执行自增或自减操作

实例：

var a = 10

var b = 10

var c = 10

var d = 10

// 操作符实现

println("a++ = ${a++} \t b-- = ${b--} \t ++c = ${++c} \t --d = ${--d}")

// 操作符重载方式实现，或许你看不明白上表中代码，不过这没关系，你只要记住上面前缀与后缀操作的区别就行

a.also { a.inc() }

b.also { b.dec() }

c.inc().also { c = it }

d.dec().also { d = it }

println("a = $a \t b = $b \t c = $c \t d = $d")

复制代码

输出结果为：

a++ = 10 b-- = 10 ++c = 11 --d = 9

a = 10 b = 10 c = 11 d = 9

复制代码

### 2.2 二元操作

二元操作：即指操作数存在二两或多个的情况。

#### 2.2.1、简单的二元操作

简单的二元操作有：

* a + b，表示两个操作数相加，值得注意的是若某一个操作数为String类型时。其返回值为String类型，当且仅当两个操作数都为数值型时，其返回值才会数值型。
* a - b，表示两个操作数相减，返回值为数值型
* a \* b，表示两个操作数相乘，返回值为数值型
* a / b，表示两个操作数相除，返回值为数值型
* a % b，表示两个操作数相除后的余数，官方称之为模，即a模以b 。返回值为Int型
* a .. b，表示范围（区间），这里不详细说明，在下面一点的区间操作符一起讲解。

这里提供一个表格直观的展示：

| **操作符** | **重载** |
| --- | --- |
| a + b | a.plus(b) |
| a - b | a.minus(b) |
| a \* b | a.tiems(b) |
| a / b | a.div(b) |
| a % b | a.rem(b) 或 a.mod(b) |
| a .. b | a.rangTo(b) |

这里值得注意的是：a % b的重载为a.rem()或a.mod()。不过a.mod()是Koltin1.0版本的重载方法，现在已经**弃用**了，Koltin1.1以及以上版本使用a.rem()重载方法

例

// 简单的二元操作

val a = 10

val b = 2

val c = "2"

val d = "Kotlin"

// 操作符实现

println("a + d = " + a + d)

println("c + d = " + c + d)

println("a + b = ${a + b} \t a - b = ${a - b} \t a \* b = ${a \* b} \t a / b = ${a / b} \t a % b = ${a % b}")

// 操作符重载实现

// println("a + d = ${a + d}") 错误：字符串模板限制只能为数值型

println("a + b = ${a.plus(b)} \t a - b = ${a.minus(b)} \t a \* b = ${a.times(b)} \t a / b = ${a.div(b)} \t a % b = ${a.rem(b)}")

// println(a.plus(d)) 错误：因为第一个操作数`a`限制了其plus()方法的参数，

// println(d.plus(a)) 正确：因为plus()方法的参数为超（Any）类型

复制代码

输出结果为：

a + d = 10Kotlin

c + d = 2Kotlin

a + b = 12 a - b = 8 a \* b = 20 a / b = 5 a % b = 0

a + b = 12 a - b = 8 a \* b = 20 a / b = 5 a % b = 0

复制代码

#### 2.2.2、复合二元操作

复合的二元操作有：

* a += b，表示第一个操作数的的值为第一个操作数加上第二个操作数，值得注意的是若某一个操作数为String类型时。其返回值为String类型，当且仅当两个操作数都为数值型时，其返回值才会数值型。
* a -= b，表示第一个操作数的的值为第一个操作数减去第二个操作数，返回值为数值型
* a \*= b，表示第一个操作数的的值为第一个操作数乘以第二个操作数，返回值为数值型
* a /= b，表示第一个操作数的的值为第一个操作数除以第二个操作数，返回值为数值型
* a %= b，表示第一个操作数的的值为第一个操作数模以第二个操作数 。返回值为Int型

这里提供一个表格直观的展示：

| **操作符** | **表示** | **重载** |
| --- | --- | --- |
| a += b | a = a + b | a = a.plus(b) |
| a -= b | a = a - b | a = a.minus(b) |
| a \*= b | a = a \* b | a = a.tiems(b) |
| a /= b | a = a / b | a = a.div(b) |
| a %= b | a = a % b | a = a.rem(b) |

例: 操作符实现

var b = 2

var a = 10

var c = "Kotlin"

// 主要演示字符串的+=

c += a 等价于 c = c.plus(a)

print("c = $c \t")

a += b 等价于 a = a.plus(b)

print("a = $a \t")

a = 10

a -= b 等价于 a = a.minus(b)

print("a = $a \t")

a = 10

a \*= b 等价于 a = a.tiems(b)

print("a = $a \t")

a = 10

a /= b 等价于 a = a.div(b)

print("a = $a \t")

a = 10

a % b 等价于 a = a.rem(b)

print("a = $a \t")

复制代码

输出结果为：

c = Kotlin10 a = 12 a = 8 a = 20 a = 5 a = 0

复制代码

或许你会说这里为什么没有Kotlin的版本呢？你在看官方文档或者其他人一些博客文章的时候可能有这样a += b <=> a.plusAssign()的操作。但是我告诉你a.plusAssign()不是这样用的，你可以看源码知道primitives.kt文件中肯本就不存在plusAssign()这个方法。因为Koltin中**赋值不是表达式**。即 a += b <=> a = a + b在Kotlin中是a = a.plus(b)。不过数组与集合是同时存在plus()和plusAssign()这两个函数的。

还有一点就是：如果我的第一个操作数定义为val(不可变)类型时，a += b这个表达式会编译出错。

上面说到了在源码primitievs.kt文件中不存在plusAssign()、minusAssign()、timesAssign()、divAssign()、remAssign()这些方法。那为什么官方文档上会存在呢？这里这里不做详解，但是我会在自定义重载操作符方法的时候给大家说明,请大家详细的往下看，一些更高级的操作

### 2.3、位运算操作

位运算操作：即对一个数进行位移运算。关于这个操作符的重载函数，我在前面讲解数据类型章节的时候已经讲解过，这里就不多做累述了。没有看过的朋友请参见[Kotlin——初级篇（三）：数据类型详解](https://link.juejin.im/?target=https%3A%2F%2Fjuejin.im%2Fpost%2F5a36020b6fb9a0451543f5c8)

### 2.4、区间操作

区间操作符：即是符号..。值得注意的是这个操作符在Java中是不存在的，且两个操作数都是整型

| **操作符** | **表示** | **重载** |
| --- | --- | --- |
| a .. b | a 到 b 中间的值 | a.rangeTo(b) |

这个操作符一般用于for循环中，在条件判断中偶尔也会用到。

例：

val a = 1

val b = 5

// 操作符实现

val s = 3 in a .. b // true,因为3在区间[1,5]之内

println("s = $s")

for (index in a .. b){

print("index = $index \t")

}

// 操作符重载方式实现

val t = 3 in a.rangeTo(b)

println("t = $t")

for (index in a.rangeTo(b)){

print("index = $index \t")

}

复制代码

输出结果为：

s = true

index = 1 index = 2 index = 3 index = 4 index = 5

t = true

index = 1 index = 2 index = 3 index = 4 index = 5

复制代码

当然了，这些实例都是极其简单的。我在[Kotlin——初级篇（四）：控制语句讲解](https://link.juejin.im/?target=https%3A%2F%2Fjuejin.im%2Fpost%2F5a369ccaf265da4325296247)这篇文章也是讲到过的。

# Kotlin——初级篇（六）： 可空类型、空安全、非空断言、类型转换等特性总结

在我们熟知的Java中，定义一个变量可以默认不赋值，因为Java的系统会给我们默认赋一个默认值，并且Java可定义一个赋值为null的变量，这样在使用这个变量的时候都会去显示判断该变量是否为null。从代码的简洁性以及代码的阅读性来说，就差了Koltin一筹了，那么Kotlin定义一个变量可为null的变量怎么定义呢？下面针对Kotlin的这些特性，作出一个详细的讲解。

## 目录

## 一、可空类型、空安全

在前面的变量、常量中我们已经讲解到了变量的定义。这里不作详述。若你有兴趣，请参见[Kotlin——初级篇（二）：变量、常量、注释](https://link.juejin.im/?target=https%3A%2F%2Fjuejin.im%2Fpost%2F5a39ef7af265da4311205967)

### 1.1、定义一个可空类型的变量

定义一个可空类型的变量的格式为：修饰符 变量名 ： 类型? = 值

这里为了演示，定义变量和定义可空变量的区别，会提到定义变量的代码。

例：

// 定义一个不可为空的变量，用var修饰的变量可以被重新赋值，用val修饰的变量则不能，但是不能赋值为null

var a : Int = 12

val b : Int = 13

a = 20

// a = null 不能复制为null

// b = 20 不能被重新赋值

if(a == null){

// 这样的判断毫无意义，因为变量a永远不可能null

}

/\*

定义可空类型的变量,即变量可以被赋值为null

定义格式为：修饰符 变量名 ： 类型? = 值

\*/

var nullA : Int? = 12

val nullB : Int? = 13

nullA = null

if(nullA == null){

println("nullA = $nullA")

}

复制代码

可以看出：变量 nullA 的值为 null

分析：要定义一个可空类型的变量时，即在定义变量的类型后面加上?符号就行了。在使用的时候，记住要判断该段该变量是否为空，这个操作在Java中经常会用到...，如果定义一个不可为空类型的变量时，则判断将毫无意义，因为这个变量永远不会为空。

### 1.2、判断可空类型的两种使用方式

在上面我们提到，可空类型需要判断在使用，这里介绍除了if ... else...之外的其他方式

**1.2.1、if...else...判断**

例：

var str : String? = "123456"

str = null

if (str == null){

println("变量str为空")

}else{

println("str.length => ${str.length}")

}

复制代码

输出结果为：

变量str为空

复制代码

**1.2.2、使用符号?.判断**

* 该符号的用法为：可空类型变量?.属性/方法。如果可空类型变量为null时，返回null
* 这种用法大量用于**链式操作**的用法中，能有效避免空引用异常（NullPointException），因为只要链式其中的一个为null，则整个表达式都为null

例：

var str : String? = "123456"

str = null

println(str?.length) // 当变量str为null时，会返回空(null)

复制代码

输出结果为：

null

复制代码

**1.2.3、链式调用**

?.这种符号去判断是否为null，在Kotlin中使用的地方是很多，特别是对于链式调用来说体验性更好。

例：这里简单写一个建造者模式，来模拟?.在链式调用中的用法

class Test{

class Builder{

private var name : String? = "Tom"

private var age : Int? = 0

private var sex : String? = "男"

fun setName(name : String) : Builder?{

this.name = name

return this

}

fun setAge(age : Int) : Builder?{

this.age = age

return this

}

fun setSex(sex: String?) : Builder?{

this.sex = sex

return this

}

override fun toString(): String {

return "Builder(name=$name, age=$age, sex=$sex)"

}

}

}

fun main(args: Array<String>) {

val builder : Test.Builder? = Test.Builder().setName("Lily")?.setSex("nv")?.setAge(10)

println(builder.toString())

}

复制代码

输出结果为：

Builder(name=Lily, age=10, sex=女)

复制代码

如果你上面的代码看不懂可以看下的代码，你可以看下面的例子：顶一个可空类型的字符串的长度加5再减去10

val testStr : String? = null

val result = testStr?.length?.plus(5)?.minus(10)

println(result)

复制代码

可以看出输出结果为: null

**1.2.4、函数中使用可空类型的情况下**

当一个函数/方法有返回值时，如果方法中的代码使用?.去返回一个值，那么方法的返回值的类型后面也要加上?符号

例：

fun funNullMethod() : Int? {

val str : String? = "123456"

return str?.length

}

复制代码

输出结果为：

6

复制代码

**1.2.5、let操作符**

* let操作符的作用：当时用符号?.验证的时候忽略掉null
* let的用法：变量?.let{ ... }

例：排除掉数组中的空元素

val arrTest : Array<Int?> = arrayOf(1,2,null,3,null,5,6,null)

// 传统写法

for (index in arrTest) {

if (index == null){

continue

}

println("index => $index")

}

// let写法

for (index in arrTest) {

index?.let { println("index => $it") }

}

复制代码

输出结果为：

index => 1

index => 2

index => 3

index => 5

index => 6

复制代码

注意：上面的两种写法的效果是相同的，可以看出使用let{}高阶函数减少了好几行代码

## Evils操作符

Evils其实不是一个操作符，而是evil的复数，而evil的意思在这里可以理解为屏蔽、安全的操作符，这样的操作符有三种：

1. ?: 这个操作符表示在判断一个可空类型时，会返回一个我们自己设定好的默认值.
2. !! 这个操作符表示在判断一个可空类型时，会显示的抛出空引用异常（NullPointException）.
3. as? 这个操作符表示为安全的类型转换.

### 2.1、?:操作符

当我们定义了一个可空类型的变量时，如果该变量不为空，则使用，反之使用另外一个不为空的值

例：

val testStr : String? = null

var length = 0

// 例： 当testStr不为空时，输出其长度，反之输出-1

// 传统写法

length = if (testStr != null) testStr.length else -1

// ?: 写法

length = testStr?.length ?: -1

println(length)

复制代码

输出结果为：

-1

复制代码

分析：此操作符一般和?.操作符连用。当且仅当?:左边的表达式为null时，才会执行?:右边的表达式。

### 2.2、!!操作符

!!操作符可谓是给爱好空引用异常（NullPointException）的开发者使用，因为在使用一个可空类型变量时，在该变量后面加上!!操作符，会显示的抛出NullPointException异常

例：

val testStr : String? = null

println(testStr!!.length)

复制代码

输出结果为：

可以看出，在未做空判断的情况下直接使用操作符!!的情况下，抛出了**空异常**

### 2.3、as?操作符

其实这里是指as操作符，表示类型转换，如果不能正常转换的情况下使用as?操作符。当使用as操作符的使用不能正常的转换的情况下会抛出类型转换（ClassCastException）异常，而使用as?操作符则会返回null,但是不会抛出异常

**2.3.1、使用as**

例：

// 会抛出ClassCastException异常

val num1 : Int? = "Koltin" as Int

println("nun1 = $num1")

复制代码

输出结果为：

**2.3.2、使用as?**

例：

val num2 : Int? = "Koltin" as? Int

println("nun2 = $num2)

复制代码

输出结果为：

num2 = null

复制代码

## 参考

[Kotlin 空安全](https://link.juejin.im/?target=http%3A%2F%2Fwww.yiibai.com%2Fkotlin%2Fnull-safety.html)  
[Koltin官网文档](https://link.juejin.im/?target=https%3A%2F%2Fkotlinlang.org%2Fdocs%2Freference%2Fnull-safety.html)

## 总结

这一章在实际的项目开发当中用到的地方是很多的，如果用好了各种空安全的操作符，估计你的项目中就不会抛出以及异常了，在这里我做了一个总结，希望会对各位有所帮助：

* 项目中会抛出空引用（NullPointerException）异常的情况
  1. 在可空类型变量的使用时，用了!!操作符
  2. 显式抛出空引用异常 throw NullPointerException()
  3. 外部 Java 代码导致的
  4. 对于初始化，有一些数据不一致(如一个未初始化的 this 用于构造函数的某个地方)
* 项目中会抛出类型转换（ClassCastException）异常的情况
  1. 在类型转换中使用了as操作符
  2. 使用了toXXX()方法不能转换的情况下
  3. 外部 Java 代码导致的
* 尽量避免使用的操作符
  1. 尽可能的不要使用!!操作符，多使用?:、?.操作符，以及let{}函数
  2. 尽可能的使用as?操作符去替换掉as,在不确定是否可以安全转换的情况下不使用toXXX()方法

# Kotlin——初级篇（七）：函数基础总结

对于Kotlin中的函数来说，和JavaScript或者Lua这些语言很像，它有着这些语言的特性。但是也与这些语言就着许多不同之处。或许你更了解Java语言，然而对于Java语言来说，它不是不是闭包这个特性的。由于在这篇文章主要是讲解函数的基本使用，故而只会讲解到函数的定义、函数的返回值、函数的参数等。剩下的部分我会在这个系列的高级篇会给大家奉上。

## 目录

## 一、函数的声明及基本使用

在Kotlin中函数的声明，同JavaScript相似。

### 1.1、函数的声明

* Kotlin中的函数声明关键字为：fun
* 定义格式为：可见性修饰符 fun 函数名(参数名 ：类型,...) : 返回值{}

这里的可见性修饰符请参见我的另一篇文章[Kotlin——中级篇（三）：可见性修饰符详解](https://link.juejin.im/?target=https%3A%2F%2Fjuejin.im%2Fpost%2F5a3293ec51882531926ebfe6)，至于返回值以及参数请继续往下面看

例：定义一个最基本的函数

fun basis(){

...

}

复制代码

注意：

* 上面的例子中没有可见性修饰符，那是因为Kotlin中默认为public可见性修饰符
* ()圆括号必须存在，即使是没有参数的情况下
* {}大括号必须存在，即使是没有函数体的时候，不过在Kotlin中有一个特例就是，函数具备返回值的时候，如果只用一个表达式就可以完成这个函数，则可以使用**单表达式函数**。在下面会详细的给大家讲解
* 在函数没有返回值时可以省略其返回值

### 1.2、成员函数

成员函数是指在类或对象中的内部函数。你可以参见我的另一篇文章[Kotlin——中级篇（一）：类（class）详解](https://link.juejin.im/?target=https%3A%2F%2Fjuejin.im%2Fpost%2F5a3297de6fb9a045055e295e)

例：声明一个类，在类中在定义一个方法，这个方法就是这个类的成员函数

class Test{

fun foo(){}

}

复制代码

### 1.3、函数的使用

函数的使用分为两种：

* 普通的使用
* 成员函数的使用

例：

// 普通的使用

basis()

// 如果函数有返回值

val x = basis()

// 成员函数的使用：先初始化对象，在根据对象使用`中缀符号(.)`调用其成员函数

Test().foo()

// 如果函数有返回值

val x = Test().foo()

复制代码

## 二、函数的返回值

在Kotlin中，函数的返回值类型可以分为：

* Unit类型：该类型即无返回值的情况，可以省略。
* 其他类型： 显示返回类型的情况

### 2.1、Unit类型

* 这个类型可以理解为函数无返回值。

例：

fun unitFun() : Unit{

println("我是返回值为Unit的函数，Unit可省略")

return

// return Unit 可省略

// 或者 return 可省略

}

等价于

fun unitFun(){

println("我是返回值为Unit的函数，Unit可省略")

}

复制代码

当无返回值时，使用平时Java语法写普通函数一样即可。不需显示写成Unit。这里只是做一个介绍而已

### 2.2 、其他显示返回类型的返回值

这个类型可以理解为，该函数具有返回值，并且返回值类型不能省略，并且return也不能省略

例： 定义一个返回类型为Int的函数

fun returnFun() : Int{

return 2

}

复制代码

## 三、函数的参数

对于函数中的参数来说，这里主要分为几个模块来讲解：

* 具有参数的基本函数声明
* 默认参数，即参数具有默认值
* 命名参数，当传递参数的时候显示使用参数名 = 参数值
* 可变数量的参数：即参数的个数不定。

### 3.1、具有参数的函数定义

定义一个具有参数的函数，使用Pascal表示法定义，即为:name : type。其中的参数必须具有显示的参数类型，并且参数与参数之间用逗号(,)隔开。

例：

fun funArgs(numA : Int, numB : Float){

println("numA = $numA \t numB = $numB")

}

fun main(args: Array<String>) {

funArgs(1,10f)

}

复制代码

### 3.2、默认参数

对于默认参数，即使指一个函数中的参数具有默认值，这样在使用该函数的时候，可以省略一部分参数，可以减少函数的重载

例：定义个具有默认参数的函数

fun defArgs(numA : Int = 1, numB : Float = 2f, numC : Boolean = false){

println("numA = $numA \t numB = $numB \t numC = $numC")

}

fun main(args: Array<String>) {

// 默认参数的函数使用

defArgs()

defArgs(1,10f,true)

}

复制代码

输出结果为：

numA = 1 numB = 2.0 numC = false

numA = 1 numB = 10.0 numC = true

复制代码

可以看出：当具有默认参数时，可以对是参数有默认值的参数不传递参数值。

不过这里有一点值得注意的是： 当该函数是一个成员函数时,并且该函数时覆写继承类中的方法时，则该成员函数必须从签名中省略该函数的默认值。其实这一点不必过于看重，因为在你覆写方法时，编辑器都默认会帮你实现的..

举例说明：

open class A{

open fun foo(i : Int = 10){}

}

class B : A(){

override fun foo(i: Int) {

super.foo(i)

println(i)

}

}

fun main(args: Array<String>) {

B().foo()

}

复制代码

输出结果为：10

如果你对类继承不了解，可以参见我的另一篇文章[Kotlin——中级篇（四）：继承类详解](https://link.juejin.im/?target=https%3A%2F%2Fjuejin.im%2Fpost%2F5a6303fb51882573467d0fbc)

### 3.3、命名参数

即在使用函数时显示使用参数名 = 参数值这种方式传递参数

例：这里举例一个**普通的具有参数函数与具有默认参数函数的结合**使用

fun callFun(str : String,

isTrue : Boolean = false,

numA : Int = 2,

numB: Float = 2.0f,

numC : Int = 2){}

复制代码

可以看出，上面实例中的str参数是不具备默认值的。

1. 我们可以这样使用它：
2. callFun("str") // 这样是可以的，因为后面的参数都具有默认值

复制代码

1. 当使用非默认参数调用它时，使用起来就和Java函数一样：
2. callFun("str",true,2,2.0,2) // 这样阅读性很差，因为你除了看函数的定义外，你不知道这些参数的含义

复制代码

1. 使用命名参数我们可以使代码更具有可读性：
2. callFun("str",isTrue = true,numA = 3, numB = 3.0f, numC = 3)

复制代码

1. 当我们不需要所有的参数时：
2. callFun("str",isTrue = true)

复制代码

不过当我们使用命名参数时，虽然可以提高代码的阅读性，及在我们使用第3、4时有一个坑，就是Java中不支持这种写法的。而在实际的项目开发中，几乎上都是和Java语言混用的。那么这种用法就不适合了。所有这里还是推荐使用1、2两点的用法。这是只是介绍Kotlin的这种便利性而已。

### 3.4、可变数量参数

* 当一个函数中的参数是不定数量的个数并且是同一个类型，则可是使用vararg修饰符去修饰这个变量，则被vararg修饰的参数相当于一个固定类型的数组。
* 声明格式：fun 函数名(vararg 参数名 ： 类型，...) ：返回值{}

例：

fun varargFun(numA: Int, vararg str : String){

// ...

}

复制代码

其中，既然vararg修饰的变量相当于一个固定类型的数组，则我们可以对它进行一些高级的操作。这里只举例一些简单的用法，如果你有兴趣，可以参见我的另一篇文章[Kotlin——高级篇（五）：集合之常用操作符汇总](https://link.juejin.im/?target=https%3A%2F%2Fjuejin.im%2Fpost%2F5b1f7699f265da6e155d5965)

例：

fun varargFun(numA: Int, vararg str : String){

// 遍历

for (s in str) {

}

// 获取元素

// str[index]

// str.component1() ... str.component5()

// 或者其高阶函数用法

// str.map { }

// str.filter { }

// str.sortBy { }

}

复制代码

在传递参数值时，我们可以一个一个参数的传递，或者可以直接传递一个当前定义类型的数组。不过在传递数组时，请使用伸展操作符( \* )。

例：

普通传递 : varargFun(1,"aaa","bbb","ccc","ddd","fff")

数组传递：

val strArray = arrayOf("aaa","bbb","ccc","ddd","fff")

varargFun(1,\*strArray)

复制代码

## 四、单表达式函数

* 上面提到了单表达式函数。这里详细的讲解一下。因为在结构体非常简单的情况下，使用单表达式函数还是能提高代码的阅读性的，并且可以少些两行代码。
* 单表达式函数：即函数具备返回值的时候，可以省略花括号并且在=赋值符号之后指定代码体，而函数的返回值是有编辑器自动推断的。

例：

// 无参数的情况

fun test1() = 2 // 自动推断为：返回类型为Int

// 有参数的情况

fun test2(num : Int) = num \* 2 // 自动推断为：返回类型为Int

// 或者

fun test3(x : Float, y : Int = 2) = x \* y // 和默认参数一起使用，返回值为Float型

fun main(args: Array<String>) {

println(test1())

println(test2(2))

println(test3(2f))

}

复制代码

输出结果为：

2

4

4.0

复制代码

## 总结

对于Kotlin中函数/方法的基本用法就讲解到这里，在后续的章节中，会为大家奉上函数的高级用法以及高阶函数的使用，并且会讲到其和lambda表达式的连用。 这一篇讲解的内容及其简单，如果您有编程经验您只要主要关注可变个数参数的函数，以及单表达式函数就可以额。

[源代码](https://link.juejin.im/?target=https%3A%2F%2Fgithub.com%2FJetictors%2FKotlinLearn%2Fblob%2Fmaster%2Fsrc%2Fcom%2Fkotlin%2Fleran%2Ffunction%2FBasisFunctionDemo.kt)

# Kotlin——初级篇（八）：关于字符串（String）常用操作汇总

在前面讲解Kotlin数据类型的时候，提到了字符串类型，当然关于其定义在前面的章节中已经讲解过了。对Kotlin中的数据类型不清楚的同学。请参考[Kotlin——初级篇（三）：数据类型详解](https://link.juejin.im/?target=https%3A%2F%2Fjuejin.im%2Fpost%2F5a36020b6fb9a0451543f5c8)这篇文章。  
在这篇文章中，会对字符串（String）的常用操作做出一个讲解。比如，字符串的截取、查找、获取长度、替换等等...

## 目录

## 一、字符串查找

字符串查找功能在实际开发中，只会用到几个简单的函数，这里我只说明几个常用的。

### 1.1、获取第一个元素

val str = "kotlin very good"

str.first() <=> str[0] <=> str.get(0)

复制代码

其还实现了查找等于某一个字符的第一个元素,即first{}高阶函数

val str = "kotlin very good"

// 如果未查找到满足条件的元素，会抛出NoSuchElementException异常

str.first{ it == 'o' }

复制代码

还有一个firstOrNull()函数，它和first()函数的区别在于如果字符串为空串时，first()函数会抛出异常，而firstOrNull()函数会返回null。当然还有firstOrNull{}函数，它和first{}函数的区别在于如果字符串为空串时，first{}函数会抛出异常，而firstOrNull{}函数会返回null

### 1.2、 获取最后一个元素

val str = ...

str.last() <==> str.get(lastIndex) <==> str[lastIndex]

其中 lastIndex 是一个拓展属性，其实现是 length - 1

复制代码

同时还实现了查找等于某一个字符的最后一个元素,即last{}高阶函数

val str = "kotlin very good"

// 如果未查找到满足条件的元素，会抛出NoSuchElementException异常

str.last{ it == 'o' } // 其实从源码中我们发现其是对原字符串反转之后再进行遍历查找满足条件的元素。

复制代码

lastOrNull()同firstOrNUll()一样。lastOrNull{}同firstOrNUll{}一样。

### 1.3、查找元素

其实查找元素提供了两个高阶函数find{}、findLast{}。不过他们都是对firstOrNull()或lastOrNull()函数进行处理。这里不多作累述。

### 1.3、 查找对应元素的下标

这里提供查找对应元素下标的几个函数：

* indexOf() : 查找某一个元素或字符串在原字符串中第一次出现的下标。
* indexLastOf() : 查找某一个元素或字符串在原字符串中最后一次出现的下标。
* indexOfFirst{} : 同indexOf()
* indexOfLast{} : 同indexLastOf()

例：

println(str.indexOfFirst { it == 'o' })

println(str.indexOfLast { it == 'o' })

println(str.indexOf('o',0))

println(str.indexOf("very",0))

println(str.lastIndexOf('o'))

println(str.lastIndexOf("good"))

复制代码

输出结果为：

1

14

1

7

14

12

复制代码

## 二、字符串截取

如果你有Java或者其他语言的编程基础。相信您对字符串的截取应该不会陌生。您可以继续的往下看，就当作是对字符串截取只是的一个巩固。当然，您也可以直接的跳过本节内容，因为在Kotlin中，字符串的截取的函数subString()是调用了Java中的subString()函数。

在Kotlin中除了调用subString()函数外，还可以调用subSequence()函数，有兴趣的朋友可以去看看源码。

### 2.1、 用subString()函数截取

我们看一看subString()函数的源码

@kotlin.internal.InlineOnly

public inline fun String.substring(startIndex: Int): String = (this as java.lang.String).substring(startIndex)

@kotlin.internal.InlineOnly

public inline fun String.substring(startIndex: Int, endIndex: Int): String

= (this as java.lang.String).substring(startIndex, endIndex)

public fun String.substring(range: IntRange): String = substring(range.start, range.endInclusive + 1)

复制代码

从以上的源码中我们可以看出，Kotlin中是使用了Java中的subString()函数

其中：

* startIndex参数：截取字符串的开始下标
* endIndex参数：截取字符串的结束下标
* rang参数，是指一个IntRang类型

实例：

val str = "Kotlin is a very good programming language"

println("s = ${str.substring(10)}") // 当只有开始下标时，结束下标为length - 1

println(str.substring(0,15))

println(str.substring(IntRange(0,15)))

复制代码

输出结果为：

a very good programming language

Kotlin is a ver

Kotlin is a very

复制代码

注意：

* 使用subString(startIndex,endIndex)和subString(rang)时的区别。可以从上边的结果结合源码可以看出来。
* 切记下标越界的情况。即StringIndexOutOfBoundsException异常

### 2.2、用subSequence()函数截取

在Kotlin中除了使用上面讲解到的使用subString()截取字符串外，还可以使用subSequence()函数截取。

我们看一看其源码实现：

public fun subSequence(startIndex: Int, endIndex: Int): CharSequence

public fun CharSequence.subSequence(range: IntRange): CharSequence = subSequence(range.start, range.endInclusive + 1)

复制代码

从源码中我们可以看出，其大致和subString()函数一样，但是其不提供只传递startIndex的情况

实例：

val str = "Kotlin is a very good programming language"

println(str.subSequence(0,15))

println(str.subSequence(IntRange(0,15)))

复制代码

输出结果为：

Kotlin is a ver

Kotlin is a very

复制代码

## 三、字符串替换

同上述字符串截取一样，如果您有编程经验，可跳过此节。不过对于字符串替换功能，Kotlin除了实现Java中的replace()ha函数外，还提供了另外的replaceFirst()、replaceAfter()、replaceBefore()、replaceIndent()、等函数。下面会对这些函数意义进行实例讲解。

### 3.1、replace()函数

replace()函数提供了4个重载函数。他们能实现不同的功能

#### 3.1.1、 replace(oldChar,newChar,ignoreCase = false)

其中：

* 作用 ： 把原字符串中的某一个字符全部替换成新的字符。然后返回新的字符串
* 参数说明：

1. oldChar: 需要替换的字符
2. newChar: 新的字符
3. ignoreCase : 是否引用Java中的replace()函数。默认值为false，即用Java的replace()函数

例：

// 把字符`a`全部替换为`A`

val str = "Kotlin is a very good programming language"

println(str.replace('a','A'))

复制代码

输出结果为：

Kotlin is A very good progrAmming lAnguAge

复制代码

#### 3.1.2、 replace(oldValue,newValue,ignoreCase = false)

其中：

* 作用 ： 把原字符串中的某一个字符全部替换成新的字符。然后返回新的字符串
* 参数说明：

1. oldValue: 需要替换的字符串
2. newValue: 新的字符串
3. ignoreCase : 是否引用Java中的replace()函数。默认值为false，即用Java的replace()函数

例：

// 把字符串`Kotlin`替换成字符串`Java`

val str = "Kotlin is a very good programming language"

println(str.replace("Kotlin","Java"))

复制代码

输出结果为：

Java is a very good programming language

复制代码

#### 3.1.3、 replace(regex,replacement)

其中：

* 作用 ： 根据定义的正则规则去匹配源字符串，把满足规则的字符串替换成新的字符串。
* 参数说明：

1. regex: 正则表达式
2. replacement: 新的字符串

例：

// 正则的规则为检测数字，如果为数字则替换成字符串`kotlin`

val str = "1234a kotlin 5678 3 is 4"

println(str.replace(Regex("[0-9]+"),"kotlin"))

复制代码

输出结果为：

kotlina kotlin kotlin kotlin is kotlin

复制代码

#### 3.1.4、replace(regex: Regex, noinline transform: (MatchResult) -> CharSequence)

其中：

* 作用 ： 根据定义的正则规则去匹配源字符串，把满足规则的字符串通过transform{}高阶函数映射的新字符串替换。
* 参数说明：

1. regex: 正则表达式
2. transform: 高阶函数

例：

val str = "1234a kotlin 5678 3 is 4"

val newStr = str.replace(Regex("[0-9]+"),{

"abcd "

})

复制代码

输出结果为：

abcd abcd abcd abcd a kotlin abcd abcd abcd abcd abcd is abcd

复制代码

可以从上述的两个函数中看到两个函数的区别。关于高阶函数、正则表达式的知识点可以先自行看资料理解。小生会在后续的章节中补上相关的内容...

经过上面的replace()函数的讲解。并且对几个重载函数进行了分析。相信大家对replace()函数中的参数都有大致的了解。并且下面的几个函数也是大致相同的操作。故而下面的几个函数操作，我只会举例说明其作用。关于参数的介绍就不作细究了

### 3.2、replaceFirst()函数

作用：把满足条件的第一个字符或字符串替换成新的字符或字符串

例：

val str = "Kotlin is a very good programming language"

println(str.replaceFirst('a','A'))

println(str.replaceFirst( "Kotlin","Java"))

复制代码

输出结果为：

Kotlin is A very good programming language

Java is a very good programming language

复制代码

### 3.3、replaceBefore()函数

作用：截取满足条件的第一个字符或字符串后面的字符串，包含满足条件字符或字符串自身，并在其前面加上新的字符串。

例：

val str = "Kotlin is a very good programming language"

println(str.replaceBefore('a',"AA"))

println(str.replaceBefore("Kotlin","Java"))

复制代码

输出结果为：

AAa very good programming language

JavaKotlin is a very good programming language

复制代码

### 3.4、replaceBeforeLast()函数

作用：截取满足条件的最后一个字符或字符串后面的字符串，包含满足条件字符或字符串自身，并在其前面加上新的字符串。

例：

val str = "Kotlin is a very good programming language"

println(str.replaceBeforeLast('a',"AA"))

println(str.replaceBeforeLast("Kotlin","Java"))

复制代码

输出结果为：

AAage

JavaKotlin is a very good programming language

复制代码

### 3.5、replaceAfter()函数

作用：截取满足条件的第一个字符或字符串前面的字符串，包含满足条件字符或字符串自身，并在其后面加上新的字符串。

例：

val str = "Kotlin is a very good programming language"

println(str.replaceAfter('a',"AA"))

println(str.replaceAfter("Kotlin","Java"))

复制代码

输出结果为：

Kotlin is aAA

KotlinJava

复制代码

### 3.6、replaceAfterLast()函数

作用：截取满足条件的最后一个字符或字符串前面的字符串，包含满足条件字符或字符串自身，并在其后面加上新的字符串。

例：

val str = "Kotlin is a very good programming language"

println(str.replaceAfterLast('a',"AA"))

println(str.replaceAfterLast("Kotlin","Java"))

复制代码

输出结果为：

Kotlin is a very good programming languaAA

KotlinJava

复制代码

## 四、字符串分割

同上一节一样，Kotlin除了实现Java中的split()函数之外，还提供了splitToSequence()函数来分割字符串。分割成功后都会返回一个字符串集合用于我们后续操作。

### 4.1、 split()函数

split()函数也是提供了4个重载函数。其中，用正则表达式为条件分割占用两个。用字符分割占用一个。用字符串分割占用一个。

#### 4.1.2、使用正则表达式分割

在Kotlin中使用正则表达式，使用的是Regex类，而Java中使用正则表达式使用的Pattern类。这里分别举例说明

例：

var str2 = "1 kotlin 2 java 3 Lua 4 JavaScript"

val list3 = str2.split(Regex("[0-9]+"))

for (str in list3){

print("$str \t")

}

println()

val list4 = str2.split(Pattern.compile("[0-9]+"))

for (str in list4){

print("$str \t")

}

复制代码

输出结果为：

kotlin java Lua JavaScript

kotlin java Lua JavaScript

复制代码

#### 4.1.2、使用字符或字符串分割

在实际的项目开发中，还是这种方式用的要多一些。不过这里值得注意的是，不管是用字符分割，还是字符串分割，其都是一个可变参数。即其参数的个数不定。

例：

val str1 = "Kotlin is a very good programming language"

val list1 = str1.split(' ')

for (str in list1){

print("$str \t")

}

println()

val list2 = str1.split(" ")

for (str in list2){

print("$str \t")

}

复制代码

输出结果为：

Kotlin is a very good programming language

Kotlin is a very good programming language

复制代码

这里举例一个可变参数情况的例子：

val str3 = "a b c d e f g h 2+3+4+5"

val list5 = str3.split(' ','+')

for (str in list5){

print("$str \t")

}

复制代码

输出结果为：

a b c d e f g h 2 3 4 5

复制代码

### 4.2、 splitToSequence()函数

此函数同样可以用字符串或字符去分割，其用法和上述的split()函数一样。这里就不多累述了...

## 五、其他

除了上面讲解到的几点外，还有很多常用的处理，比如检测字符串是否为空串，是否为null，获取字符串长度，字符串反转，统计，转换字符数组，获取指定下标的字符等等。

### 5.1、 获取字符串长度

在Kotlin中用两种获取字符串长度的方法。其实也是一种而已

1. 直接用length属性获取长度
2. 用count()函数获取，其实count()函数的做法也是返回了length长度。

实例：

val str = "kotlin very good"

// 1. 直接用length属性获取

println("str.length => ${str.length}")

// 2. 用count()函数获取

println("str.count() => ${str.count()}")

复制代码

输出结果为：

str.length => 16

str.count() => 16

复制代码

这里我们看一看count()函数的源码

/\*\*

\* Returns the length of this char sequence.

\* 其实也就是返回了length属性...

\*/

@kotlin.internal.InlineOnly

public inline fun CharSequence.count(): Int {

return length

}

复制代码

### 5.2、统计重复字符

上面讲到了count()函数，是返回了length属性获取字符串的长度。其实源码中还提供了一个叫count{}的高阶函数，用于统计字符串中重复某个字符的个数。

看一看他的源码

public inline fun CharSequence.count(predicate: (Char) -> Boolean): Int {

var count = 0

for (element in this) if (predicate(element)) count++

return count

}

复制代码

源码解释：该函数接受一个Boolean类型的Lambda表达式。然后循环这个字符串，如果我的条件成立，则变量count自增。循环完成之后返回重复的个数count

实例：

val str = "kotlin very good"

val count = str.count { it == 'o' }

println("count : $count")

复制代码

输出结果为：

count : 3

复制代码

### 5.3、 验证字符串

在实际的开发中，特别是Android开发中，会经常遇到验证输入框的内容是否为空串的时候。在Kotlin中，和Java一样提供了几个处理这种情况的函数。

下列几个函数都是处理字符串为空串或为空的：

* isEmpty() : 其源码是判断其length是等于0，若等于0则返回true,反之返回false。不能直接用于可空的字符串
* isNotEmpty() : 其源码是判断其length是否大于0，若大于0则返回true,反之返回false。不能直接用于可空的字符串
* isNullOrEmpty() : 其源码是判断该字符串是否为null或者其length是否等于0。
* isBlank() : 其源码是判断其length是否等于0,或者判断其包含的空格数是否等于当前的length。不能直接用于可空的字符串
* isNotBlank() : 其源码是对isBlank()函数取反。不能直接用于可空的字符串
* isNotOrBlank() : 其源码判断该字符串是否为null。或者调用isBlank()函数

实例：

val str : String? = "kotlin"

// 可以看出当str为可空变量的时候，isNullOrEmpty()和isNotOrBlank()可以不做直接调用而不做任何处理

，而其他的函数则不行

println(str?.isEmpty())

println(str?.isNotEmpty())

println(str.isNullOrEmpty())

println(str?.isBlank())

println(str?.isNotBlank())

println(str.isNullOrBlank())

复制代码

输出结果为：

false

true

false

false

true

false

复制代码

### 5.4、字符串连接

字符串链接，在Java中只能使用+来链接，当然除了StringBuilder、StringBuffer除外。而在Kotlin中除了使用+以外，还可以使用plus()函数。其接受任意类型。plus()函数是一个运算符重载函数。在前面的章节中讲解过了。若你对Kotlin中的云素福重载不了解，您可以查看我的另一篇文章：[Kotlin——初级篇（五）：操作符重载一](https://link.juejin.im/?target=https%3A%2F%2Fjuejin.im%2Fpost%2F5a4ce9865188257d6a7ef291)

实例讲解：

val oldStr = "kotlin"

println(oldStr.plus(" very good"))

println(oldStr + " very good")

复制代码

输出结果为：

kotlin very good

kotlin very good

复制代码

### 5.5、字符串反转

和数组一样，字符串也是可以反转元素的。直接使用reversed()函数即可。

例：

val str = "kotlin"

println("字符串反转：${str.reversed()}")

复制代码

输出结果：

字符串反转：niltok

复制代码

### 5.6、判断字符串的起始与结尾

在实际的开发中，判断字符串的起始与结尾是很常用的。

#### 5.6.1、 startsWith()

其中：

作用： 判断其字符串是否由某一个字符或字符串起始。 参数说明：

1. char : 起始字符
2. prefix : 起始字符串
3. ignoreCase : 是否调用Java中的此函数。默认为false
4. startIndex : 开始位置

例：

val str = "kotlin"

str.startsWith('k') // 是否有字符`k`起始

str.startsWith("Kot") // 是否由字符串`kot`起始

str.startsWith("lin",3) // 当起始位置为3时，是否由字符串`lin`起始

复制代码

输出结果为：

true

true

true

复制代码

#### 5.6.2、endsWith()

其中：

作用： 判断其字符串是否由某一个字符或字符串结尾。 参数说明：

1. char : 结尾字符
2. suffix : 结尾字符串
3. ignoreCase : 是否调用Java中的此函数。默认为false

例：

val str = "kotlin"

println(str.endsWith("lin")) // 是否由字符串`lin`结尾

println(str.endsWith('n')) // 是否由字符`n`结尾

复制代码

输出结果为：

true

true

复制代码

## 总结

在实际的项目开发中，字符串的处理使用的情况是很多的。特别是字符串的验证处理、替换、分割、截取。这也是我把这些知识点整理出来的原因。这些知识点是很基础，但同时也是很常用的。如果您有编程经验的话您就当对字符串的基础知识的巩固。  
关于kotlin中字符串的操作就写到这里，其实还有很多功能，虽然在开发中用到的地方不是特别多，但是我们最起码能应该有个了解。有兴趣的朋友可以去看看其源码的实现。

[源代码](https://link.juejin.im/?target=https%3A%2F%2Fgithub.com%2FJetictors%2FKotlinLeran%2Fblob%2Fmaster%2Fsrc%2Fcom%2Fkotlin%2Fleran%2Fbasis%2FOperatorStringDemo.kt)

**Kotlin——中级篇（一）：类（class）详解**

在任何一门面向对象编程的语言里，类（class）是非常基础、但也是非常重要的一项组成，通俗的说就是万般皆对象，而所说的对象就是我们生成的类。Kotlin也是如此，下面详细为大家介绍Kotlin中的类的组成结构、函数、声明等。

**目录**

**一、类的声明**

**1、关键字**

声明类的关键字为class

**2、声明格式**

class Test{

// 属性...

...

// 构造函数

...

// 函数

...

// 内部类

...

...

}

复制代码

其中：

**当类没有结构体的时候，大括号可以省略。即：**

class Test

复制代码

**二、类的构造函数**

* 在Kotlin中，允许有一个主构造函数和多个二级构造函数（辅助构造函数）。其中主构造函数是类头的一部分。
* 关键字或者构造函数名：constructor(参数)

**1、主构造函数**

* 主构造函数是类头的一部分，类名的后面跟上构造函数的关键字以及类型参数。

**1.1、举例说明：**

class Test constructor(num : Int){

...

}

复制代码

**等价于**

/\*

因为是默认的可见性修饰符且不存在任何的注释符

故而主构造函数constructor关键字可以省略

\*/

class Test(num: Int){

...

}

复制代码

**1.2、构造函数中的初始化代码块**

* 构造函数中不能出现其他的代码，只能包含初始化代码。包含在初始化代码块中。
* 关键字：init{...}
* 值得注意的是，init{...}中能使用构造函数中的参数

例：

fun main(args: Array<String>) {

// 类的实例化，会在下面讲解到，这里只是作为例子讲解打印结果

var test = Test(1)

}

class Test constructor(var num : Int){

init {

num = 5

println("num = $num")

}

}

复制代码

输出结果为：

num = 5

复制代码

其中，上面的constructor关键字是可以省略的。

**1.3、声明属性的简便方法**

* 即在主构造函数中声明。

例：

class Test(val num1 : Int, var num2 : Long, val str : String){

...

}

复制代码

则：相当于声明了3个属性。 其中，var表示变量（可读写），val表示常量（只读）。

**1.4、什么时候constructor可以省略**

* 在构造函数不具有注释符或者默认的可见性修饰符时，constructor关键字可以省略。
* 默认的可见性修饰符时public。可以省略不写。请参我的另外一篇文章[Kotlin——中级篇（三）：可见性修饰符详解](https://link.juejin.im/?target=https%3A%2F%2Fjuejin.im%2Fpost%2F5a3293ec51882531926ebfe6)

例：

// 类似下面两种情况的，都必须存在constructor关键字，并且在修饰符或者注释符后面。

class Test private constructor(num: Int){

}

class Test @Inject constructor(num: Int){

}

复制代码

**2、辅助（二级）构造函数**

* Kotlin中支持二级构造函数。它们以constructor关键字作为前缀。

**2.1、声明**

例：

class Test{

constructor(参数列表){

}

}

复制代码

**2.2、同时存在主构造函数和二级构造函数时的情况**

* 如果类具有主构造函数，则每个辅助构造函数需要通过另一个辅助构造函数直接或间接地委派给主构造函数。 使用this关键字对同一类的另一个构造函数进行委派：

例：

fun main(args: Array<String>) {

var test1 = Test(1)

var test2 = Test(1,2)

}

// 这里是为了代码清晰，故而没有隐藏constructor关键字

class Test constructor(num: Int){

init {

println("num = $num")

}

constructor(num : Int, num2: Int) : this(num) {

println(num + num2)

}

}

复制代码

说明：二级构造函数中的参数1(num)，是委托了主构造函数的参数num。

可以看出，当实例化类的时候只传1个参数的时候，只会执行init代码块中的代码。当传2个参数的时候，除了执行了init代码块中代码外，还执行了二级构造函数中的代码。

输出结果为：

num = 1

num = 1

3

复制代码

**2.3、当类的主构造函数都存在默认值时的情况**

* 在JVM上，如果类主构造函数的所有参数都具有默认值，编译器将生成一个额外的无参数构造函数，它将使用默认值。 这使得更容易使用Kotlin与诸如Jackson或JPA的库，通过无参数构造函数创建类实例。
* 同理可看出，当类存在主构造函数并且有默认值时，二级构造函数也适用

例：

fun main(args: Array<String>) {

var test = Test()

var test1 = Test(1,2)

var test2 = Test(4,5,6)

}

class Test constructor(num1: Int = 10 , num2: Int = 20){

init {

println("num1 = $num1\t num2 = $num2")

}

constructor(num1 : Int = 1, num2 : Int = 2, num3 : Int = 3) : this(num1 , num2){

println("num1 = $num1\t num2 = $num2 \t num3 = $num3")

}

}

复制代码

输出结果为：

num1 = 10 num2 = 20

num1 = 1 num2 = 2

num1 = 4 num2 = 5

num1 = 4 num2 = 5 num3 = 6

复制代码

说明： 当实例化无参的构造函数时。使用了参数的默认值。

**三、类的实例化**

* 创建一个类的实例，需要调用类的构造函数，就像它是一个常规函数一样：

例：

var test = Test()

var test1 = Test(1,2)

复制代码

其实在上面的例子中就实例化类的运用。 注意：**这里和Java不同的点是，没有new这个关键字**

**四、类的组成**

* 类的构成由构造函数和初始化代码块、属性（字段）、函数（方法）、内部类（嵌套类）、对象声明五部分组成

**1、构造函数和初始化代码块**

在此篇文章已经讲解，不清楚请再次熟悉下此篇文章中的**构造函数**模块。

**2、属性（字段）**

由于篇幅原因，请参见[Kotlin——中级篇（二）： 属性与字段详解](https://link.juejin.im/?target=https%3A%2F%2Fjuejin.im%2Fpost%2F5a6378266fb9a01ca10b00e4)章节。

**3、函数（方法）**

由于篇幅原因，请参见[Kotlin——初级篇（七）：函数基础总结](https://link.juejin.im/?target=https%3A%2F%2Fjuejin.im%2Fpost%2F5a6377425188257329148665)章节。

**4、内部类（嵌套类）**

由于篇幅原因，请参见[Kotlin——中级篇（七）：抽象类（abstract）、内部类（嵌套类）详解](https://link.juejin.im/?target=https%3A%2F%2Fjuejin.im%2Fpost%2F5a48a0e8518825455f2fa070)章节。

**5、对象声明**

由于篇幅原因，请参见[对象声明](https://juejin.im/post/5a3297de6fb9a045055e295e)章节。

**五、类的类别**

* Kotlin中的类可详细的分为：密封类、内部类（嵌套类）、抽象类、枚举类、接口类、数据类

**1、密封类**

由于篇幅原因，请参见[Kotlin——中级篇（六）：数据类（data）、密封类(sealed)详解](https://link.juejin.im/?target=https%3A%2F%2Fjuejin.im%2Fpost%2F5a37e4b45188253aea1f7219)章节。

**2、内部类（嵌套类）**

内部类、嵌套类已在[Kotlin——中级篇（七）：抽象类（abstract）、内部类（嵌套类）详解](https://link.juejin.im/?target=https%3A%2F%2Fjuejin.im%2Fpost%2F5a48a0e8518825455f2fa070)中讲解过了。

**3、抽象类**

由于篇幅原因，请参见[Kotlin——中级篇（七）：抽象类（abstract）、内部类（嵌套类）详解](https://link.juejin.im/?target=https%3A%2F%2Fjuejin.im%2Fpost%2F5a48a0e8518825455f2fa070)章节。

**4、枚举类**

由于篇幅原因，请参见[Kotlin——中级篇（五）：枚举类（Enum）、接口类（Interface）详解](https://link.juejin.im/?target=https%3A%2F%2Fjuejin.im%2Fpost%2F5a34c551518825552b3f9c91)章节。

**5、接口类**

由于篇幅原因，请参见[Kotlin——中级篇（五）：枚举类（Enum）、接口类（Interface）详解](https://link.juejin.im/?target=https%3A%2F%2Fjuejin.im%2Fpost%2F5a34c551518825552b3f9c91)章节。

**6、数据类**

由于篇幅原因，请参见[Kotlin——中级篇（六）：数据类（data）、密封类(sealed)详解](https://link.juejin.im/?target=https%3A%2F%2Fjuejin.im%2Fpost%2F5a37e4b45188253aea1f7219)章节。

**六、类的继承**

由于篇幅原因，请参见[Kotlin——中级篇（四）：继承类详解](https://link.juejin.im/?target=https%3A%2F%2Fjuejin.im%2Fpost%2F5a6303fb51882573467d0fbc)章节。

**七、总结**

关于类的介绍就到这里，不清楚的可以多看看文章。不过也没太大关系，此篇文章大多数都是理论性东西。自己用Kotlin多写写代码就理解、掌握了。

# Kotlin——中级篇（三）：可见性修饰符详解

在Kotlin中，不管是类，对象，接口，构造函数，函数，属性及其设置器都具有可见性修饰符。Kotlin中的可见性修饰符共四种。即public、protected、private、internal。在不同的场景声明不同的修饰符都有不同的可见性。下面详解介绍不同的修饰符在不同场景的范围。 这里值得注意的是，属性的Getter()函数的可见性具有与属性相同的可见性。

## 目录

## 一、 四种修饰符的说明

* 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 。对于模块的范围在下面会说明。
* 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。

## 二、在不同场景下各个修饰符声明的范围讲解

对于可见性修饰符来说，不管是Java还是Kotlin，在不同的场景下用可见性修饰符修饰过的变量、函数、属性等都有不同的引用范围。

### 2.1、在包(package)中声明，即顶层声明的情况

在顶层申明中分为下面两种情况：

* 在同一个文件中访问的情况
* 在不同文件中访问的情况

**2.1.1、测试同一个文件中声明不同的修饰符的范围**

**2.1.2、测试不同文件中声明不同的修饰符的范围**

**2.1.3、小结**

* 在顶层情况下声明的变量、方法、类、接口等，不能使用protected修饰符。
* public修饰符可省略，因为它是系统默认的。
* 在同一文件下，用任意修饰符都可以访问，当然此处除开protected修饰符
* 在不同文件下，只能访问public、internal修饰符修饰的变量、方法等
* 上边只演示了在顶层声明方法的情况，不过声明其他的也是一样的。

### 2.2、在类(class)和接口(interface)中声明的情况

* 在类中声明的情况和在接口中声明的情况是完全不相同的。

**2.2.1、测试在类中声明的情况**

**2.2.2、测试在接口中声明的情况**

**2.2.3、小结**

* 在类中声明属性、方法等，可以使用任意修饰符修饰。并且在类中的任意位置都能访问各个修饰符修饰的属性、方法等。
* 在类外，只能访问由public、internal修饰符修饰的属性、方法。此情况不管是不是在同一个文件中去操作。
* 在接口中声明的情况下，只能由public修饰符修饰属性。方法可由public、private两个修饰符去修饰，但是，用private修饰符修饰符修饰的方法不能被实现该接口的类重写。
* 为了使用接口中由public修饰符修饰的属性、方法等。只能定义一个类去实现该接口，然后重写其属性、方法。这样就能在可在类中或类外去操作。

### 2.3、在构造函数中声明的情况

看过我前几遍文章的都知道在Kotlin中分为主构造函数，和副级或多级构造函数，当然在Java中也是存在的，只不过我们更习惯称其为方法重载。对此还不甚清晰的，可以阅读我的上一篇文章[Kotlin——中级篇（一）：类(class)详解](https://link.juejin.im/?target=http%3A%2F%2Fwww.jianshu.com%2Fp%2Ff9b089d216c4)。 故而，当我们在构造函数中声明的情况下，如果是主构造函数的时候，是和在类(class)中声明的情况是一样的。如果是主构造函数的时候，则是和在局部声明的情况是一样的。

### 2.4、局部声明的情况

在局部声明的情况下，只能使用public修饰符修饰的变量。并且在变量只能在局部这个范围内使用，出了这个范围后就访问不到了。

## 三、与Java中的可见性修饰符的对比

* Java中的可见性修饰符和Koltin中的可见性修饰符还是很有几点不同之处的： 1、 四种修饰符的不同。Koltin的四种修饰符（public、interna、protected、private），Java的的四种修饰符（public、protected、privavte、default（即不使用任何修饰符））。 2、默认修饰符的不同。Kotlin的默认修饰符为public，Java的默认修饰符为default

# Kotlin——中级篇（四）：继承类详解

在前面的章节中，详细的详解了类的使用，但是由于篇幅的限制，关于类的很多特性都没有讲解到。今天在这篇文章中，详细的讲解Kotlin中类的特性。如果您对Kotlin中的类还没有一个整体的了解的话，请参见我上一篇文章[Kotlin——中级篇（一）：类（class）详解](https://link.juejin.im/?target=https%3A%2F%2Fjuejin.im%2Fpost%2F5a3297de6fb9a045055e295e)  
众所周知，Kotlin是一门面向对象的开发语言。那么他也有面向对象语言的特性。而面向对象的三大特性即封装、继承、多态。这是每一门面向对象语言否具有的特性。今天这一节会着重的讲解Kotlin的继承与Java的不同处和Kotlin独有的特点。

## 目录

## 一、面向对象的特征

面向对象的三大特征：封装、继承、多态

由于面向对象的三大特征太过于普通，而且这并不是Kotlin中特有的知识。在这里就不多做描述。

## 二、Kotlin继承类

在Kotlin中，继承这个特性除了定义关键字,以及所有的父类和Java语言不通之外，其他的其实无太大的差别。不过既然写到了这里，还是从始至终的写完这个特性，如果您有Java的基础，您可以当复习一遍。

### 2.1、超类(Any)

在Kotlin中，说有的类都是继承与Any类，这是这个没有父类型的类。即当我们定义各类时，它默认是继承与Any这个超类的

例：

class Demo // 这里定义了一个Demo类，即这个类默认是继承与超类的。

复制代码

因为Any这个类只是给我们提供了equals()、hashCode()、toString()这三个方法。我们可以看看Any这个类的源码实现：

package kotlin

/\*\*

\* The root of the Kotlin class hierarchy. Every Kotlin class has [Any] as a superclass.

\* 看这个源码注释：意思是任何一个Kotlin的类都继承与这个[Any]类

\*/

public open class Any {

// 比较: 在平时的使用中经常用到的equals()函数的源码就在这里额

public open operator fun equals(other: Any?): Boolean

// hashCode()方法：其作用是返回该对象的哈希值

public open fun hashCode(): Int

// toString()方法

public open fun toString(): String

}

复制代码

从源码可以我们看出，它直接属于kotlin这个包下。并且只定义了上面所示的三个方法。或许你具有Java的编程经验。在我们熟知的Java中，所有的类默认都是继承与Object类型的。而Object这个类除了比Any多了几个方法与属性外，没有太大的区别。不过他们并不是同一个类。这里就不多种讲解了....

从上面源码中所产生的疑惑：类与函数前面都加上了open这个修饰符。那么这个修饰符的作用是什么呢？  
其实我们分析可以得出：既然Any类是所有类的父类，那么我们自己要定义一个继承类，跟着Any类的语法与结构就能定义一个继承类。故而，open修饰符是我们定义继承类的修饰符

### 2.2、定义

#### 2.2.1、继承类的基础使用

* 定义继承类的关键字为：open。不管是类、还是成员都需要使用open关键字。
* 定义格式为：
* open class 类名{
* ...
* open var/val 属性名 = 属性值
* ...
* open fun 函数名()
* ...
* }

复制代码

例：这里定义一个继承类Demo，并实现两个属性与方法,并且定义一个DemoTest去继承自Demo

open class Demo{

open var num = 3

open fun foo() = "foo"

open fun bar() = "bar"

}

class DemoTest : Demo(){

// 这里值得注意的是：Kotlin使用继承是使用`:`符号，而Java是使用extends关键字

}

fun main(args: Array<String>) {

println(DemoTest().num)

DemoTest().foo()

DemoTest().bar()

}

复制代码

输出结果为：

3

foo

bar

复制代码

分析：从上面的代码可以看出，DemoTest类只是继承了Demo类，并没有实现任何的代码结构。一样可以使用Demo类中的属性与函数。这就是继承的好处。

#### 2.2.2、继承类的构造函数

在上一篇文章中，讲解到了Kotlin类，可以有一个主构造函数，或者多个辅助函数。或者没有构造函数的情况。如果您对Kotlin的构造函数还不了解的情况，请阅读我的上一篇文章[Kotlin——中级篇（一）：类（class）详解](https://link.juejin.im/?target=https%3A%2F%2Fjuejin.im%2Fpost%2F5a3297de6fb9a045055e295e)。

这里当实现类无主构造函数，和存在主构造函数的情况。

* **无主构造函数**

当实现类无主构造函数时，则每个辅助构造函数必须使用super关键字初始化基类型，或者委托给另一个构造函数。 请注意，在这种情况下，不同的辅助构造函数可以调用基类型的不同构造函数

例：这里举例在Android中常见的自定义View实现，我们熟知，当我们指定一个组件是，一般实现继承类(基类型)的三个构造函数。

class MyView : View(){

constructor(context: Context) : super(context)

constructor(context: Context, attrs: AttributeSet?) : super(context, attrs)

constructor(context: Context, attrs: AttributeSet?, defStyleAttr: Int)

: super(context, attrs, defStyleAttr)

}

复制代码

可以看出，当实现类无主构造函数时，分别使用了super()去实现了基类的三个构造函数。

* **存在主构造函数**

当存在主构造函数时，主构造函数一般实现基类型中参数最多的构造函数，参数少的构造函数则用this关键字引用即可了。这点在[Kotlin——中级篇（一）：类（class）详解](https://link.juejin.im/?target=https%3A%2F%2Fjuejin.im%2Fpost%2F5a3297de6fb9a045055e295e)这篇文章是讲解到的。

例：同样以自定义组件为例子

class MyView(context: Context?, attrs: AttributeSet?, defStyleAttr: Int)

: View(context, attrs, defStyleAttr) {

constructor(context: Context?) : this(context,null,0)

constructor(context: Context?,attrs: AttributeSet?) : this(context,attrs,0)

}

复制代码

### 2.3、函数的重载与重写

在Kotlin中关于函数的重载与重写,和Java中是几乎是一样的，但是这里还是举例来说明一下。

**2.3.1、重写函数中的两点特殊用法**

不管是Java还是Kotlin，重写基类型里面的方法，则称为重写，或者是覆盖基类型方法。不过这里介绍两点Kotlin一点特殊的地方

1. 当基类中的函数，没有用open修饰符修饰的时候，实现类中出现的函数的函数名不能与基类中没有用open修饰符修饰的函数的函数名相同，不管实现类中的该函数有无override修饰符修饰。读着有点绕，直接看例子你就明白了。

例：

open class Demo{

fun test(){} // 注意，这个函数没有用open修饰符修饰

}

class DemoTest : Demo(){

// 这里声明一个和基类型无open修饰符修饰的函数，且函数名一致的函数

// fun test(){} 编辑器直接报红，根本无法运行程序

// override fun test(){} 同样报红

}

复制代码

1. 当一个类不是用open修饰符修饰时，这个类默认是final的。即：

class A{}

等价于

final class A{} // 注意，则的`final`修饰符在编辑器中是灰色的，因为Kotlin中默认的类默认是final的

复制代码

那么当一个基类去继承另外一个基类时，第二个基类不想去覆盖掉第一个基类的方法时，第二个基类的该方法使用final修饰符修饰。

例：

open class A{

open fun foo(){}

}

// B这个类继承类A，并且类B同样使用open修饰符修饰了的

open class B : Demo(){

// 这里使用final修饰符修饰该方法，禁止覆盖掉类A的foo()函数

final override fun foo(){}

}

复制代码

**2.3.2、方法重载**

在文章的开头提到了多态这个特性，方法的重载其实主要体现在这个地方。即函数名相同，函数的参数不同的情况。这一点和Java是相同的

这一点在继承类中同样有效：

例：

open class Demo{

open fun foo() = "foo"

}

class DemoTest : Demo(){

fun foo(str: String) : String{

return str

}

override fun foo(): String {

return super.foo()

}

}

fun main(args: Array<String>) {

println(DemoTest().foo())

DemoTest().foo("foo的重载函数")

}

复制代码

输出结果为：

foo

foo的重载函数

复制代码

### 2.4、重写属性

* 重写属性和重写方法其实大致是相同的，但是属性不能被重载。
* 重写属性即指：在基类中声明的属性，然后在其基类的实现类中重写该属性，该属性必须以override关键字修饰，并且其属性具有和基类中属性一样的类型。且可以重写该属性的值（Getter）

例：

open class Demo{

open var num = 3

}

class DemoTest : Demo(){

override var num: Int = 10

}

复制代码

**2.4.1、重写属性中，val与var的区别**

这里可以看出重写了num这个属性，并且为这个属性重写了其值为10  
但是，还有一点值得我们注意：当基类中属性的变量修饰符为val的使用，其实现类可以用重写属性可以用var去修饰。反之则不能。

例：

open class Demo{

open val valStr = "我是用val修饰的属性"

}

class DemoTest : Demo(){

/\*

\* 这里用val、或者var重写都是可以的。

\* 不过当用val修饰的时候不能有setter()函数，编辑器直接会报红的

\*/

// override val valStr: String

// get() = super.valStr

// override var valStr: String = ""

// get() = super.valStr

// override val valStr: String = ""

override var valStr: String = "abc"

set(value){field = value}

}

fun main(arge: Array<String>>){

println(DemoTest().valStr)

val demo = DemoTest()

demo.valStr = "1212121212"

println(demo.valStr)

}

复制代码

输出结果为：

abc

1212121212

复制代码

**2.4.2、Getter()函数慎用super关键字**

在这里值得注意的是，在实际的项目中在重写属性的时候不用get() = super.xxx,因为这样的话，不管你是否重新为该属性赋了新值，还是支持setter(),在使用的时候都调用的是基类中的属性值。

例： 继上面中的例子

class DemoTest : Demo(){

/\*

\* 这里介绍重写属性是，getter()函数中使用`super`关键字的情况

\*/

override var valStr: String = "abc"、

get() = super.valStr

set(value){field = value}

}

fun main(arge: Array<String>>){

println(DemoTest().valStr)

val demo = DemoTest()

demo.valStr = "1212121212"

println(demo.valStr)

}

复制代码

输出结果为：

我是用val修饰的属性

我是用val修饰的属性

复制代码

切记：重写属性的时候慎用super关键字。不然就是上面例子的效果

**2.4.3、在主构造函数中重写**

这一点和其实在接口类的文章中讲解过了，不清楚的可以去参见[Kotlin——中级篇（五）：枚举类（Enum）、接口类（Interface）详解](https://link.juejin.im/?target=https%3A%2F%2Fjuejin.im%2Fpost%2F5a34c551518825552b3f9c91)。

例：基类还是上面的例子

class DemoTest2(override var num: Int, override val valStr: String) : Demo()

fun main(args: Array<String>){

val demo2 = DemoTest2(1,"构造函数中重写")

println("num = ${demo2.num} \t valStr = ${demo2.valStr}")

}

复制代码

输出结果为：

num = 1 valStr = 构造函数中重写

复制代码

### 2.5、覆盖规则

这里的覆盖规则，是指实现类继承了一个基类，并且实现了一个接口类，当我的基类中的方法、属性和接口类中的函数重名的情况下，怎样去区分实现类到底实现哪一个中的属性或属性。 这一点和一个类同时实现两个接口类，而两个接口都用同样的属性或者函数的时候是一样的。在接口类这篇文章中已经讲解过，您可以参见[Kotlin——中级篇（五）：枚举类（Enum）、接口类（Interface）详解](https://link.juejin.im/?target=https%3A%2F%2Fjuejin.im%2Fpost%2F5a34c551518825552b3f9c91)。

例：

open class A{

open fun test1(){ println("基类A中的函数test1()") }

open fun test2(){println("基类A中的函数test2()")}

}

interface B{

fun test1(){ println("接口类B中的函数test1()") }

fun test2(){println("接口类B中的函数test2()")}

}

class C : A(),B{

override fun test1() {

super<A>.test1()

super<B>.test1()

}

override fun test2() {

super<A>.test2()

super<B>.test2()

}

}

复制代码

## 总结

对于Kotlin中**继承类**这一个知识点，在项目中用到的地方是很常见的。当你认真的学习完上面的内容，我相信你可以能很轻易的用于项目中，不过对一个类来说，继承的代价较高，当实现一个功能不必用到太多的集成属性的时候，可以用对象表达式这一个高级功能去替代掉继承。  
如果你有过其他面向对象语言的编程经验的话，你只要掌握其关键字、属性/函数重写、以及覆盖规则这三三个知识点就可以了。

# Kotlin——中级篇（六）：数据类（data）、密封类(sealed)详解

在前面几个章节章节中，详细的讲解了Koltin中的接口类（Interface）、枚举类（Enmu）,还不甚了解的可以查看我的上一篇文章[Kotlin——中级篇（五）：枚举类（Enum）、接口类（Interface）详解](https://link.juejin.im/?target=https%3A%2F%2Fjuejin.im%2Fpost%2F5a34c551518825552b3f9c91)。当然，在Koltin中，除了接口类、枚举类之外，还有抽象类、内部类、数据类以及密封类。在今天的章节中，为大家详细讲解数据类和密封类。在下一章节中，再为大家奉上Kotlin中的抽象类以及内部类的知识。如果还对Kotlin中类的分类还不清楚的可以查看我的另一篇博文[Kotlin——中级篇（一）：类（class）详解](https://link.juejin.im/?target=https%3A%2F%2Fjuejin.im%2Fpost%2F5a3297de6fb9a045055e295e)。

## 目录

## 一、数据类

* **在Java中，或者在我们平时的Android开发中，为了解析后台人员给我们提供的接口返回的Json字符串，我们会根据这个字符串去创建一个类或者实例对象，在这个类中，只包含了一些我们需要的数据，以及为了处理这些数据而所编写的方法。这样的类，在Kotlin中就被称为数据类**。

### 1、关键字

声明数据类的关键字为：data

**1.1、声明格式**

data class 类名(var param1 ：数据类型,...){}

复制代码

或者

data class 类名 可见性修饰符 constructor(var param1 : 数据类型 = 默认值,...)

复制代码

说明：

* data为声明数据类的关键字，必须书写在class关键字之前。
* 在没有结构体的时候，大括号{}可省略。
* 构造函数中必须存在至少一个参数，并且必须使用val或var修饰。这一点在下面数据类特性中会详细讲解。
* 参数的默认值可有可无。（若要实例一个无参数的数据类，则就要用到默认值）

例：

// 定义一个名为Person的数据类

data class Preson(var name : String,val sex : Int, var age : Int)

复制代码

**1.2、约定俗成的规定**

* 数据类也有其约定俗成的一些规定，这只是为增加代码的阅读性。

即，当构造函数中的参过多时，为了代码的阅读性，一个参数的定义占据一行。

例：

data class Person(var param1: String = "param1",

var param2: String = "param2",

var param3 : String,

var param4 : Long,

var param5 : Int = 2,

var param6 : String,

var param7 : Float = 3.14f,

var param8 : Int,

var param9 : String){

// exp

.

.

.

}

复制代码

**1.3、编辑器为我们做的事情**

当我们声明一个数据类时，编辑器自动为这个类做了一些事情，不然它怎么又比Java简洁呢。它会根据主构造函数中所定义的所有属性自动生成下列方法：

* 生成equals()函数与hasCode()函数
* 生成toString()函数，由类名（参数1 = 值1，参数2 = 值2，....）构成
* 由所定义的属性自动生成component1()、component2()、...、componentN()函数，其对应于属性的声明顺序。
* copy()函数。在下面会实例讲解它的作用。

其中，**当这些函数中的任何一个在类体中显式定义或继承自其基类型，则不会生成该函数**

### 2、数据类的特性

数据类有着和Kotlin其他类不一样的特性。除了含有其他类的一些特性外，还有着其独特的特点。并且也是数据类必须满足的条件：

* 主构造函数需要至少有一个参数
* 主构造函数的所有参数需要标记为 val 或 var；
* 数据类不能是抽象、开放、密封或者内部的；
* 数据类是可以实现接口的，如(序列化接口)，同时也是可以继承其他类的，如继承自一个密封类。

### 3、用实例说明其比Java的简洁性

**3.1、数据类的对比**

**Kotlin版：**

data class User(val name : String, val pwd : String)

复制代码

**Java版：**

public class User {

private String name;

private String pwd;

public User(){}

public User(String name, String pwd) {

this.name = name;

this.pwd = pwd;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public String getPwd() {

return pwd;

}

public void setPwd(String pwd) {

this.pwd = pwd;

}

@Override

public String toString() {

return "User{" +

"name='" + name + '\'' +

", pwd='" + pwd + '\'' +

'}';

}

}

复制代码

分析：实现同一个功能，从代码量来说，Koltin比Java少了很多行代码，比起更简洁。

**3.2、修改数据类属性**

例：修改User类的name属性

**Kotlin版：**

* Koltin要修改数据类的属性，则使用其独有的copy()函数。其作用就是：修改部分属性，但是保持其他不变

val mUser = User("kotlin","123456")

println(mUser)

val mNewUser = mUser.copy(name = "new Kotlin")

println(mNewUser)

复制代码

输出结果为：

User(name=kotlin, pwd=123456)

User(name=new Kotlin, pwd=123456)

复制代码

**Java版：**

User mUser = new User("Java","123456");

System.out.println(mUser);

mUser.setName("new Java");

System.out.println(mUser);

复制代码

输出结果为：

User{name='Java', pwd='123456'}

User{name='new Java', pwd='123456'}

复制代码

分析：从上面对两种方式的实现中可以看出，Kotlin是使用其独有的copy()函数去修改属性值，而Java是使用setXXX()去修改

### 4、解构声明

* 在前面讲到，Kotlin中定义一个数据类，则系统会默认自动根据参数的个数生成component1() ... componentN()函数。其...,componentN()函数就是用于解构声明的

val mUser = User("kotlin","123456")

val (name,pwd) = mUser

println("name = $name\tpwd = $pwd")

复制代码

输出结果为：

name = kotlin pwd = 123456

复制代码

### 5、系统标准库中的标准数据类

* 标准库提供了 Pair 和 Triple。尽管在很多情况下命名数据类是更好的设计选择， 因为它们通过为属性提供有意义的名称使代码更具可读性。
* 其实这两个类的源码部分不多，故而贴出这个类的源代码来分析分析

**5.1、源码分析**

@file:kotlin.jvm.JvmName("TuplesKt")

package kotlin

// 这里去掉了源码中的注释

public data class Pair<out A, out B>(

public val first: A,

public val second: B) : Serializable {

// toString()方法

public override fun toString(): String = "($first, $second)"

}

// 转换

public infix fun <A, B> A.to(that: B): Pair<A, B> = Pair(this, that)

// 转换成List集合

public fun <T> Pair<T, T>.toList(): List<T> = listOf(first, second)

// 这里去掉了源码中的注释

public data class Triple<out A, out B, out C>(

public val first: A,

public val second: B,

public val third: C ) : Serializable {

// toString()方法

public override fun toString(): String = "($first, $second, $third)"

}

// 转换成List集合

public fun <T> Triple<T, T, T>.toList(): List<T> = listOf(first, second, third)

复制代码

分析：从上面的源码可以看出，标准库中提供了两个标准的数据类，Pair类以及Triple类.其中：

* 两个类中都实现了toList()方法以及toString()方法。
* to()方法乃Pair类特有，作用是参数转换
* Pair类需要传递两个参数，Triple类需要传递三个参数。

**5.2、用法**

val pair = Pair(1,2) // 实例

val triple = Triple(1,2,3) // 实例

println("$pair \t $triple") // 打印：即调用了各自的toString()方法

println(pair.toList()) // 转换成List集合

println(triple.toList()) // 转换成List集合

println(pair.to(3)) // Pair类特有: 其作用是把参数Pair类中的第二个参数替换

复制代码

输出结果为：

(1, 2) (1, 2, 3)

[1, 2]

[1, 2, 3]

((1, 2), 3)

复制代码

## 二、密封类

密封类是用来表示受限的类继承结构。若还不甚清楚Kotlin的类继承，请参见我的上一篇文章[Kotlin——中级篇（四）：继承类详解](https://link.juejin.im/?target=https%3A%2F%2Fjuejin.im%2Fpost%2F5a6303fb51882573467d0fbc)。

### 1、什么是受限的类继承结构

* 所谓受限的类继承结构，即当类中的一个值只能是有限的几种类型，而不能是其他的任何类型。
* 这种受限的类继承结构从某种意义上讲，它相当于是枚举类的扩展。但是，我们知道Kotlin的枚举类中的枚举常量是受限的，因为每一个枚举常量只能存在一个实例。若对Kotlin中的枚举类不甚了解的，请参见我的另一篇文章[Kotlin——中级篇（五）：枚举类（Enum）、接口类（Interface）详解](https://link.juejin.im/?target=https%3A%2F%2Fjuejin.im%2Fpost%2F5a34c551518825552b3f9c91)。
* 但是其和枚举类不同的地方在于，密封类的一个子类可以有可包含状态的多个实例。
* 也可以说成，密封类是包含了一组受限的类集合，因为里面的类都是继承自这个密封类的。但是其和其他继承类（open）的区别在，密封类可以不被此文件外被继承，有效保护代码。但是，其密封类的子类的扩展是是可以在程序中任何位置的，即可以不在同一文件下。

上面的几点内容是密封类的特点，请详细的看下去，小生会对这几点内容进行详细的分析。

### 2、关键字

定义密封类的关键字：sealed

**2.1、声明格式**

sealed class SealedExpr()

复制代码

注意：**密封类是不能被实例化的**

即

val mSealedExpr = SealedExpr() // 这段代码是错误的，编译器直接会报错不能编译通过。

复制代码

既然密封类是不能实例化，那么我们要怎么使用，或者说它的作用是什么呢？请继续往下看

### 3、密封类的作用及其详细用法。

**3.1、作用**

用来表示受限的类继承结构。

例：

sealed class SealedExpr{

data class Person(val num1 : Int, val num2 : Int) : SealedExpr()

object Add : SealedExpr() // 单例模式

object Minus : SealedExpr() // 单例模式

}

// 其子类可以定在密封类外部，但是必须在同一文件中。`v1.1`之前只能定义在密封类内部

object NotANumber : SealedExpr()

复制代码

分析：即所定义的子类都必须继承于密封类，表示一组受限的类

**3.2、和普通继承类的区别**

* 我们知道普通的继承类使用open关键字定义，在项目中的类都可继承至该类。如果你对Koltin的继承类还不甚了解。请参见我的另一篇文章[Kotlin——中级篇（四）：继承类详解](https://link.juejin.im/?target=https%3A%2F%2Fjuejin.im%2Fpost%2F5a6303fb51882573467d0fbc)。
* 而密封类的子类必须是在密封类的内部或必须存在于密封类的同一文件。这一点就是上面提到的有效的代码保护。

**3.3、和枚举类的区别**

* 枚举类的中的每一个枚举常量都只能存在一个实例。而密封类的子类可以存在多个实例。

例：

val mPerson1 = SealedExpr.Person("name1",22)

println(mPerson1)

val mPerson2 = SealedExpr.Person("name2",23)

println(mPerson2)

println(mPerson1.hashCode())

println(mPerson2.hashCode())

复制代码

输出结果为：

Person(name=name1, age=22)

Person(name=name2, age=23)

-1052833328

-1052833296

复制代码

**3.4、其子类的类扩展实例**

* 在Kotlin支持扩展功能，其和C#、Go语言类似。这一点是Java没有的。如果你还对Koltin中的扩展功能还不甚清楚的。请参见我的另一篇博文[Kotlin——扩展功能详解](https://juejin.im/post/5a37e4b45188253aea1f7219)

为了演示密封类的子类的扩展是可以在项目中的任何位置这个功能，大家可以下载源码。源码链接在文章末尾会为大家奉上。 例：

// 其存在于SealedClassDemo.kt文件中

sealed class SealedExpr{

data class Person(val name : String, val age : Int) : SealedExpr()

object Add : SealedExpr()

companion object Minus : SealedExpr()

}

object NotANumber : SealedExpr()

其存在TestSealedDemo.kt文件中

fun <T>SealedExpr.Add.add(num1 : T, num2 : T) : Int{

return 100

}

fun main(args: Array<String>) {

println(SealedExpr.Add.add(1,2))

}

复制代码

输出结果为：

100

复制代码

说明：上面的扩展功能没有任何的意义，只是为了给大家展示密封类子类的扩展不局限与密封类同文件这一个功能而已。如果你还对Koltin中的扩展功能还不甚清楚的。请参见我的另一篇博文[Kotlin——扩展功能详解](https://juejin.im/post/5a37e4b45188253aea1f7219)

**3.5、使用密封类的好处**

* 有效的保护代码（上面已说明原因）
* 在使用when表达式 的时候，如果能够验证语句覆盖了所有情况，就不需要为该语句再添加一个else子句了。

例：

sealed class SealedExpr{

data class Person(val name : String, val age : Int) : SealedExpr()

object Add : SealedExpr()

companion object Minus : SealedExpr()

}

object NotANumber : SealedExpr()

fun eval(expr: SealedExpr) = when(expr){

is SealedExpr.Add -> println("is Add")

is SealedExpr.Minus -> println("is Minus")

is SealedExpr.Person -> println(SealedExpr.Person("Koltin",22))

NotANumber -> Double.NaN

}

复制代码

输出结果为：

is Minus

复制代码

## 三、总结

在实际的项目开发当中，数据类(data)类的用处是很多的，因为在开发APP时，往往会根据后台开发者所提供的接口返回的json而生成一个实体类，现在我们学习了数据类后，就不用再像Java一样写那么多代码了，即使是用编辑器提供的方法去自动生成。但是代码量上就能节省我们很多时间，并且也更加简洁。何乐而不为呢!密封类的情况在实际开发中不是很常见的。只有当时特殊的需求会用到的时候，才会使用密封类。当然我们还是要学习的。

# Kotlin——中级篇（七）：抽象类（abstract）、内部类（嵌套类）详解

在前面几个章节中，详细的介绍了Kotlin类的类别中的数据类、密封类、接口类以及枚举类。在这个章节中会对Koltin的抽象类和内部类作出一个详细的讲解。如果对上面所提到的类的类别还不是很清晰的，请阅读我的前几篇文章。  
[Kotlin——中级篇（六）：数据类（data）、密封类详解](https://link.juejin.im/?target=https%3A%2F%2Fjuejin.im%2Fpost%2F5a37e4b45188253aea1f7219)  
[Kotlin——中级篇（五）：枚举类（Enum）、接口类（Interface）详解](https://link.juejin.im/?target=https%3A%2F%2Fjuejin.im%2Fpost%2F5a34c551518825552b3f9c91)  
[Kotlin——中级篇（一）：类（class）详解](https://link.juejin.im/?target=https%3A%2F%2Fjuejin.im%2Fpost%2F5a3297de6fb9a045055e295e)

## 目录

## 一、抽象类

* 我们知道，在实际的开发程序的时候，一般都会写一个基类，封装常用方法、以及处理一些共有的逻辑，但是程序逻辑是根据每个类不同的功能实现不同的代码。而这个所谓的基类，一般都是一个抽象类。不管是Java还是Kotlin，实现其抽象类的作用就在于此。那么什么是抽象类呢，它是怎么定义的，它又要怎么使用呢？

### 1、抽象类的定义

抽象类，可以理解为类定义了一个模板。所有的子类都是根据这个模板是填充自己的代码。

**1.1、关键字**

* 声明一个抽象（类或函数）的关键字为：abstract

其中值得注意的是：抽象可以分为抽象类、抽象函数、抽象属性。而一个抽象类和普通类的区别在于抽象类除了可以有其自己的属性、构造函数、方法等组成部分，还包含了抽象函数以及抽象属性。

例：

abstract class Lanauage{

val TAG = this.javaClass.simpleName // 自身的属性

// 自身的函数

fun test() : Unit{

// exp

}

abstract var name : String // 抽象属性

abstract fun init() // 抽象方法

}

/\*\*

\* 抽象类Lanauage的实现类TestAbstarctA

\*/

class TestAbstarctA : Lanauage(){

override var name: String

get() = "Kotlin"

set(value) {}

override fun init() {

println("我是$name")

}

}

/\*\*

\* 抽象类Lanauage的实现类TestAbstarctB

\*/

class TestAbstarctB : Lanauage(){

override var name: String

get() = "Java"

set(value) {}

override fun init() {

println("我是$name")

}

}

fun main(args: Array<String>) {

// val lanauage = Lanauage() 是错误的，因为抽象类不能直接被实例化

val mTestAbstarctA = TestAbstarctA()

val mTestAbstarctB = TestAbstarctB()

println(mTestAbstarctA.name)

mTestAbstarctA.init()

println(mTestAbstarctB.name)

mTestAbstarctB.init()

}

复制代码

输出结果为：

Kotlin

我是Kotlin

Java

我是Java

复制代码

**1.2、小结**

* 抽象类本身具有普通类特性，以及组成部分。不过值得注意的是，**抽象类不能直接被实例化**
* 其抽象了类的子类必须全部重写带abstract修饰的属性和方法。
* 抽象成员只有定义，没有实现。都有abstract修饰符修饰。
* 抽象类是为其子类定义了一个模板。不同是类实现不同的功能

### 2、抽象类的规则

* 在Kotlin中的抽象类在顶层定义的时候只能使用public可见性修饰符修饰。
* 抽象类中可以定义内部抽象类。
* 只能继承一个抽象类。
* 若要实现抽象类的实例化，需要依靠子类采用向上转型的方式处理。
* 抽象类可以继承自一个继承类，即抽象类可以作为子类。不过，抽象类建议不用open修饰符修饰，因为可以覆写抽象类的父类的函数。

例：

open class Base{

open fun init(){}

}

abstract class Lanauage : Base(){

val TAG = this.javaClass.simpleName // 自身的属性

// 自身的函数

fun test() : Unit{

// exp

}

abstract var name : String // 抽象属性

abstract override fun init() // 覆写父类的方法

abstract class Name(){} // 嵌套抽象类，可查看第二节中的嵌套类使用

}

fun main(args: Array<String>) {

// 若要实现抽象类的实例化，需要依靠子类采用向上转型的方式处理。

val mLanauage : Lanauage = TestAbstarctB()

}

复制代码

### 3、抽象类的实际应用

* 在Java的设计模式中，有一种设计模式叫模板设计模式，其定义为：
  + 定义一个操作中算法的骨架，而将一些步骤延迟到子类中，模板方法使得子类可以不改变算法的结构即可重定义该算法的某些特定步骤。
  + 通俗点的理解就是 ：完成一件事情，有固定的数个步骤，但是每个步骤根据对象的不同，而实现细节不同；就可以在父类中定义一个完成该事情的总方法，按照完成事件需要的步骤去调用其每个步骤的实现方法。每个步骤的具体实现，由子类完成。

Kotlin和Java是互通的，说明Kotlin也是支持这种设计模式的。  
如果你对Java中的**模板设计模式**不是很了解的，请参见[这篇文章](https://link.juejin.im/?target=https%3A%2F%2Fwww.cnblogs.com%2Fqq-361807535%2Fp%2F6854191.html)

## 二、内部类（嵌套类）

在实际开发中，用到内部类的地方是很多的。比如说：

* 对于Android开发来说，列表适配器（adapter）中的ViewHolder类，就是一个内部类。
* 根据后台开发人员提供的json字符串生成的对象中，也包含另外一个对象，这也是一个内部类。

### 1、嵌套类

上面提到的两种情况，是在开发中最常见的。当然，说到内部类，就必须世道另一个概念**嵌套类**，所谓的嵌套类：即指一个类可以嵌套在其他类中。

例：

class Other{ // 外部类

val numOuther = 1

class Nested { // 嵌套类

fun init(){

println("执行了init方法")

}

}

}

fun main(args: Array<String>) {

Other.Nested().init() // 调用格式为：外部类.嵌套类().嵌套类方法/属性

}

复制代码

输出结果为：

执行了init方法

复制代码

**注意：**

* 调用嵌套类的属性或方法的格式为：外部类.嵌套类().嵌套类方法/属性。在调用的时候嵌套类是需要实例化的。
* 嵌套类不能使用外部类的属性和成员

### 2、内部类

在上面的例子中讲解了嵌套类的使用，而内部类和嵌套类还是有一定的区别的，而且内部类是有特定的关键字去声明的。

#### 2.1、关键字

声明一个内部类使用inner关键字。 声明格式：inner class 类名(参数){}

例：

class Other{ // 外部类

val numOther = 1

inner class InnerClass{ // 嵌套内部类

val name = "InnerClass"

fun init(){

println("我是内部类")

}

}

}

fun main(args: Array<String>) {

Other().InnerClass().init() // 调用格式为：外部类().内部类().内部类方法/属性

}

复制代码

**注意：**

* 调用内部类的属性或方法的格式为：外部类().内部类().内部类方法/属性。在调用的时候嵌套类是需要实例化的。
* 内部类不能使用外部类的属性和成员

#### 2.2、匿名内部类

作为一名Android开发者，对匿名内部类都不陌生，因为在开发中，匿名内部类随处可见。比如说Button的OnClickListener，ListView的单击、长按事件等都用到了匿名内部类。 一般的使用方式为定义一个接口，在接口中定义一个方法。

例：

class Other{

lateinit private var listener : OnClickListener

fun setOnClickListener(listener: OnClickListener){

this.listener = listener

}

fun testListener(){

listener.onItemClick("我是匿名内部类的测试方法")

}

}

interface OnClickListener{

fun onItemClick(str : String)

}

fun main(args: Array<String>){

// 测试匿名内部类

val other = Other()

other.setOnClickListener(object : OnClickListener{

override fun onItemClick(str: String) {

// todo

println(str)

}

})

other.testListener()

}

复制代码

输出结果为：

我是匿名内部类的测试方法

复制代码

**遗留的问题**

在上面实现的匿名内部类是很常规的用法以及写法。在我们的实际开发当中也是大家熟知的写法。但是在我们实际开发当中，会引入lambda语法糖，让我们的项目支持lambda语法，简化代码量。在这里我也是想把用\*\*lambda语法实现匿名内部类实现的代码贴出来\*\*，但是考虑到很多朋友不知道lambda语法强大或者对lambda不了解，同时考虑到篇幅原因，故而在我的另一篇博文[Kotlin——高级篇（一）：Lambda表达式详解](https://link.juejin.im/?target=https%3A%2F%2Fjuejin.im%2Fpost%2F5ab9a5ccf265da239f076284)去讲解。

### 3、局部类

所谓局部类，这一点和Java是一致的。即定义在方法（函数）中的类。

例：

class Other{ // 外部类

val numOther = 1

fun partMethod(){

var name : String = "partMethod"

class Part{

var numPart : Int = 2

fun test(){

name = "test"

numPart = 5

println("我是局部类中的方法")

}

}

val part = Part()

println("name = $name \t numPart = " + part.numPart + "\t numOther = numOther")

part.test()

println("name = $name \t numPart = " + part.numPart + "\t numOther = numOther")

}

}

fun main(args: Array<String>) {

// 测试局部类

Other().partMethod()

}

复制代码

输出结果为：

name = partMethod numPart = 2 numOther = 1

我是局部类中的方法

name = test numPart = 5 numOther = 1

复制代码

**通过上面的实例：我们可以看出：**

* 局部类只能在定义该局部类的方法中使用。
* 定义在实例方法中的局部类可以访问外部类的所有变量和方法。但不能修改
* 局部类中的可以定义属性、方法。并且可以修改局部方法中的变量。

### 4、静态类

熟悉Java的朋友都知道Java的静态类，或者说用static修饰符修饰的类。但是在Kotlin中，是不存在static关键字的。那么我们怎样去实现一个静态类呢？

关于**静态类**的使用，以及静态类的语法。以及Koltin的单例模式实现。由于篇幅原因我在这里就不展示了。有兴趣的朋友请参见[kotlin中的object更像是语法糖](https://link.juejin.im/?target=https%3A%2F%2Fwww.jianshu.com%2Fp%2F157126615f77)。这篇文章是别的大牛诠释静态类以及单例实现很好的文章。后面我会出一篇详细的文章为大家讲解。

## 三、总结

在学完本篇博文中，你应该掌握抽象类的作用，掌握其和普通类、接口类、继承类的区别所在，了解实现抽象类的意义，或者说在项目中为什么要用抽象类去编写一个基类等。  
对于嵌套类和内部类而言，知道这两者的区别所在，和熟知他们在项目中用在什么地方就够了。对于静态类来说，常用的实现都是用其去实现一个单例模式。在Koltin的不像Java一样实现很多的工具类，因为Kotlin中的**扩展**功能很强大。可以用**扩展**去替换掉大部分的工具类。本篇文章主要是展示object的用法而已，详细的使用场景和用法会在后续的文章中为大家奉上。

# Kotlin——高级篇（一）：Lambda表达式详解

经过前面一系列对Kotlin讲解，相信大家已经能对Kotlin有了一个基本的认识。如果你又Java语言方面的编程经验，你可能已经不满足前面的基础语法了。从这篇文章起，就为大家讲解Kotlin语言中的高级操作。  
Lambda语法在Java中已经被广泛的运用，我们在开发Android中几乎上每一个项目也会在项目中接入Lambda插件，因为Lambda确实能简少很多的代码量。无独有偶，在Kotlin中也是Lambda语法的，在这篇文章中就详细的为大家讲解Lambda语法的编写与使用，同时会后面的[Kotlin——高级篇（二）：高阶函数详解与标准的高阶函数使用](https://link.juejin.im/?target=https%3A%2F%2Fjuejin.im%2Fpost%2F5b198c675188257d7a49b3ec)打下基础。

## 目录

## 一、Lambda介绍

在上面已经提到了在Java中已经被广泛的运用，但是也是在Java8的时候才支持这种Lambda表达式。在其他的编程语言中（例如：Scala语言）。而这种表达式是语法糖中的一种。值得庆幸的是，Kotlin一经开源成熟就已经支持这种语法。

Lambda表达式的本质其实是匿名函数，因为在其底层实现中还是通过匿名函数来实现的。但是我们在用的时候不必关心起底层实现。不过Lambda的出现确实是减少了代码量的编写，同时也是代码变得更加简洁明了。  
Lambda作为函数式编程的基础，其语法也是相当简单的。这里先通过一段简单的代码演示没让大家了解Lambda表达式的简洁之处。

例：

// 这里举例一个Android中最常见的按钮点击事件的例子

mBtn.setOnClickListener(object : View.OnClickListener{

override fun onClick(v: View?) {

Toast.makeText(this,"onClick",Toast.LENGTH\_SHORT).show()

}

})

复制代码

等价于

// 调用

mBtn.setOnClickListener { Toast.makeText(this,"onClick",Toast.LENGTH\_SHORT).show() }

复制代码

## 二、Lambda使用

关于Lambda的使用，我这里从两个方面讲解，一是先介绍Lambda表达式的特点，二是从Lambda的语法使用讲解。

### 2.1、Lambda表达式的特点

古人云：欲取之，先与之。

要学习Lambda表达式语法，必先了解其特点。我在这里先总结出Lambda表达式的一些特征。在下面讲解到Lambda语法与实践时大家就明白了。即：

* Lambda表达式总是被大括号括着
* 其参数(如果存在)在 -> 之前声明(参数类型可以省略)
* 函数体(如果存在)在 -> 后面。

#### 2.2、Lambda语法

为了让大家彻底的弄明白Lambda语法，我这里用三种用法来讲解。并且举例为大家说明

语法如下：

1. 无参数的情况 ：

val/var 变量名 = { 操作的代码 }

2. 有参数的情况

val/var 变量名 : (参数的类型，参数类型，...) -> 返回值类型 = {参数1，参数2，... -> 操作参数的代码 }

可等价于

// 此种写法：即表达式的返回值类型会根据操作的代码自推导出来。

val/var 变量名 = { 参数1 ： 类型，参数2 : 类型, ... -> 操作参数的代码 }

3. lambda表达式作为函数中的参数的时候，这里举一个例子：

fun test(a : Int, 参数名 : (参数1 ： 类型，参数2 : 类型, ... ) -> 表达式返回类型){

...

}

复制代码

实例讲解：

* 无参数的情况
* // 源代码
* fun test(){ println("无参数") }
* // lambda代码
* val test = { println("无参数") }
* // 调用
* test() => 结果为：无参数

复制代码

* 有参数的情况,这里举例一个两个参数的例子，目的只为大家演示
* // 源代码
* fun test(a : Int , b : Int) : Int{
* return a + b
* }
* // lambda
* val test : (Int , Int) -> Int = {a , b -> a + b}
* // 或者
* val test = {a : Int , b : Int -> a + b}
* // 调用
* test(3,5) => 结果为：8

复制代码

* lambda表达式作为函数中的参数的时候
* // 源代码
* fun test(a : Int , b : Int) : Int{
* return a + b
* }
* fun sum(num1 : Int , num2 : Int) : Int{
* return num1 + num2
* }
* // 调用
* test(10,sum(3,5)) // 结果为：18
* // lambda
* fun test(a : Int , b : (num1 : Int , num2 : Int) -> Int) : Int{
* return a + b.invoke(3,5)
* }
* // 调用
* test(10,{ num1: Int, num2: Int -> num1 + num2 }) // 结果为：18

复制代码

最后一个的实现可能大家难以理解，但请不要迷茫，你继续看下去，在下面的实践和高阶函数中会为大家介绍。

经过上面的实例讲解与语法的介绍，我们对其作出一个总结：

1. lambda表达式总是被大括号括着。
2. 定义完整的Lambda表达式如上面实例中的语法2，它有其完整的参数类型标注，与表达式返回值。当我们把一些类型标注省略的情况下，就如上面实例中的语法2的另外一种类型。当它推断出的返回值类型不为'Unit'时，它的返回值即为->符号后面代码的最后一个（或只有一个）表达式的类型。
3. 在上面例子中语法3的情况表示为：高阶函数，当Lambda表达式作为其一个参数时，只为其表达式提供了参数类型与返回类型，所以，我们在调用此高阶函数的时候我们要为该Lambda表达式写出它的具体实现。
4. invoke()函数：表示为通过函数变量调用自身，因为上面例子中的变量b是一个匿名函数。

### 3、Lambda实践

学会了上面讲解的语法只是，相信您已能大致的编写且使用lambda表达式了，不过只会上面简单的语法还不足以运用于实际项目中复杂的情况。下面从几个知识点讲解Lambda实践的要点。

### 3.1、it

* it并不是Kotlin中的一个关键字(保留字)。
* it是在当一个高阶函数中Lambda表达式的参数只有一个的时候可以使用it来使用此参数。it可表示为**单个参数的隐式名称**，是Kotlin语言约定的。

例1：

val it : Int = 0 // 即it不是`Kotlin`中的关键字。可用于变量名称

复制代码

例2：单个参数的隐式名称

// 这里举例一个语言自带的一个高阶函数filter,此函数的作用是过滤掉不满足条件的值。

val arr = arrayOf(1,3,5,7,9)

// 过滤掉数组中元素小于2的元素，取其第一个打印。这里的it就表示每一个元素。

println(arr.filter { it < 5 }.component1())

复制代码

例2这个列子只是给大家it的使用，filter高阶函数，在后面的[Kotlin——高级篇（四）：集合（Array、List、Set、Map）基础](https://link.juejin.im/?target=https%3A%2F%2Fjuejin.im%2Fpost%2F5ab7a9c4f265da2377196038)章节中会为大家详细讲解，这里不多做介绍。下面为我们自己写一个高阶函数去讲解it。关于高阶函数的定义与使用请参见[Kotlin——高级篇（二）：高阶函数详解与标准的高阶函数使用](https://link.juejin.im/?target=https%3A%2F%2Fjuejin.im%2Fpost%2F5b198c675188257d7a49b3ec)这篇文章。

例3：

fun test(num1 : Int, bool : (Int) -> Boolean) : Int{

return if (bool(num1)){ num1 } else 0

}

println(test(10,{it > 5}))

println(test(4,{it > 5}))

复制代码

输出结果为：

10

0

复制代码

代码讲解：上面的代码意思是，在高阶函数test中，其返回值为Int类型，Lambda表达式以num1位条件。其中如果Lambda表达式的值为false的时候返回0，反之返回num1。故而当条件为num1 > 5这个条件时，当调用test函数，num1 = 10返回值就是10，num1 = 4返回值就是0。

### 3.2、下划线（\_）

在使用Lambda表达式的时候，可以用下划线(\_)表示未使用的参数，表示不处理这个参数。

同时在遍历一个Map集合的时候，这当非常有用。

举例：

val map = mapOf("key1" to "value1","key2" to "value2","key3" to "value3")

map.forEach{

key , value -> println("$key \t $value")

}

// 不需要key的时候

map.forEach{

\_ , value -> println("$value")

}

复制代码

输出结果：

key1 value1

key2 value2

key3 value3

value1

value2

value3

复制代码

### 3.3 匿名函数

* 匿名函数的特点是可以明确指定其返回值类型。
* 它和常规函数的定义几乎相似。他们的区别在于，匿名函数没有函数名。

例：

fun test(x : Int , y : Int) : Int{ fun(x : Int , y : Int) : Int{

常规函数： return x + y 匿名函数： return x + y

} }

复制代码

在前面的[Kotlin——初级篇（七）：函数基础总结](https://link.juejin.im/?target=https%3A%2F%2Fjuejin.im%2Fpost%2F5a6377425188257329148665)我们讲解过单表达式函数。故而，可以简写成下面的方式。

常规函数 ： fun test(x : Int , y : Int) : Int = x + y

匿名函数 ： fun(x : Int , y : Int) : Int = x + y

复制代码

从上面的两个例子可以看出，匿名函数与常规函数的区别在于一个有函数名，一个没有。

实例演练：

val test1 = fun(x : Int , y : Int) = x + y // 当返回值可以自动推断出来的时候，可以省略，和函数一样

val test2 = fun(x : Int , y : Int) : Int = x + y

val test3 = fun(x : Int , y : Int) : Int{

return x + y

}

println(test1(3,5))

println(test2(4,6))

println(test3(5,7))

复制代码

输出结果为：

8

10

12

复制代码

从上面的代码我们可以总结出匿名函数与Lambda表达式的几点区别：

1. 匿名函数的参数传值，总是在小括号内部传递。而Lambda表达式传值，可以有省略小括号的简写写法。
2. 在一个不带标签的return语句中，匿名函数时返回值是返回自身函数的值，而Lambda表达式的返回值是将包含它的函数中返回。

### 3.4、带接收者的函数字面值

在kotlin中，提供了指定的接受者对象调用Lambda表达式的功能。在函数字面值的函数体中，可以调用该接收者对象上的方法而无需任何额外的限定符。它类似于扩展函数，它允你在函数体内访问接收者对象的成员。

* **匿名函数作为接收者类型**

匿名函数语法允许你直接指定函数字面值的接收者类型，如果你需要使用带接收者的函数类型声明一个变量，并在之后使用它，这将非常有用。

例：

val iop = fun Int.( other : Int) : Int = this + other

println(2.iop(3))

复制代码

输出结果为：

5

复制代码

* **Lambda表达式作为接收者类型**

要用Lambda表达式作为接收者类型的前提是**接收着类型可以从上下文中推断出来**。

例：这里用官方的一个例子做说明

class HTML {

fun body() { …… }

}

fun html(init: HTML.() -> Unit): HTML {

val html = HTML() // 创建接收者对象

html.init() // 将该接收者对象传给该 lambda

return html

}

html { // 带接收者的 lambda 由此开始

body() // 调用该接收者对象的一个方法

}

复制代码

### 3.5 闭包

* 所谓闭包，即是函数中包含函数，这里的函数我们可以包含(Lambda表达式，匿名函数，局部函数，对象表达式)。我们熟知，函数式编程是现在和未来良好的一种编程趋势。故而Kotlin也有这一个特性。
* 我们熟知，Java是不支持闭包的，Java是一种面向对象的编程语言，在Java中，对象是他的一等公民。函数和变量是二等公民。
* Kotlin中支持闭包，函数和变量是它的一等公民，而对象则是它的二等公民了。

实例：看一段Java代码

public class TestJava{

private void test(){

private void test(){ // 错误，因为Java中不支持函数包含函数

}

}

private void test1(){} // 正确，Java中的函数只能包含在对象中+

}

复制代码

实例：看一段Kotlin代码

fun test1(){

fun test2(){ // 正确，因为Kotlin中可以函数嵌套函数

}

}

复制代码

下面我们讲解Kotlin中几种闭包的表现形式。

#### 3.5.1、携带状态

例：让函数返回一个函数，并携带状态值

fun test(b : Int): () -> Int{

var a = 3

return fun() : Int{

a++

return a + b

}

}

val t = test(3)

println(t())

println(t())

println(t())

复制代码

输出结果：

7

8

9

复制代码

#### 3.5.2、引用外部变量，并改变外部变量的值

例：

var sum : Int = 0

val arr = arrayOf(1,3,5,7,9)

arr.filter { it < 7 }.forEach { sum += it }

println(sum)

复制代码

输出结果：

9

复制代码

### 3.6 在Android开发中为RecyclerView的适配器编写一个Item点击事件

class TestAdapter(val context : Context , val data: MutableList<String>)

: RecyclerView.Adapter<TestAdapter.TestViewHolder>(){

private var mListener : ((Int , String) -> Unit)? = null

override fun onBindViewHolder(holder: TestViewHolder?, position: Int) {

...

holder?.itemView?.setOnClickListener {

mListener?.invoke(position, data[position])

}

}

override fun onCreateViewHolder(parent: ViewGroup?, viewType: Int): TestViewHolder {

return TestViewHolder(View.inflate(context,layoutId,parent))

}

override fun getItemCount(): Int {

return data.size

}

fun setOnItemClickListener(mListener : (position : Int, item : String) -> Unit){

this.mListener = mListener

}

inner class TestViewHolder(itemView : View) : RecyclerView.ViewHolder(itemView)

}

// 调用

TestAdapter(this,dataList).setOnItemClickListener { position, item ->

Toast.makeText(this,"$position \t $item",Toast.LENGTH\_SHORT).show()

}

复制代码

## 总结

Lambda表达式是为我们减少了大量的代码，但是Lambda表达式是为后面的高阶函数章节打下基础，虽然在这篇文章中也提到了高阶函数，但是都是最基础的，在下一节中会为大家介绍自定义高阶函数与Kotlin自身中常用的高阶函数讲解。

在这一章节中，讲述了Lambda的语法、使用。以及Lambda表达式的一些特性与实践操作。当然还包含了匿名函数这一知识点。其中最重要的当属Lambda的实践操作。如果你看完这篇文章还不甚理解，请在仔细的阅读一遍并实际代码演练，因为在后面的高阶函数章节还会遇到。

# Kotlin——高级篇（二）：高阶函数详解与标准的高阶函数使用

在上面一个章节中，详细的讲解了Kotlin中关于Lambda表达式的语法以及运用，如果还您对其还不甚理解，请参见[Kotlin——高级篇（一）：Lambda表达式详解](https://link.juejin.im/?target=https%3A%2F%2Fjuejin.im%2Fpost%2F5ab9a5ccf265da239f076284)。在这篇文章中，多次提到了Kotlin中关于高阶函数的内容，故而在这一篇文章中会详解的对Kotlin高阶函数的方方面面。

## 目录

## 一、高阶函数介绍

在介绍高阶函数之前，或许您先应该了解Kotlin中，基础函数的使用与定义。您可以参见[Kotlin——初级篇（七）：函数基础总结](https://link.juejin.im/?target=https%3A%2F%2Fjuejin.im%2Fpost%2F5a6377425188257329148665)这边文章的用法。

在Kotlin中，高阶函数即指：将函数用作一个函数的参数或者返回值的函数。

### 1.1、将函数用作函数参数的情况的高阶函数

这里介绍字符串中的sumBy{}高阶函数。先看一看源码

// sumBy函数的源码

public inline fun CharSequence.sumBy(selector: (Char) -> Int): Int {

var sum: Int = 0

for (element in this) {

sum += selector(element)

}

return sum

}

复制代码

源码说明：

1. 大家这里可以不必纠结inline，和sumBy函数前面的CharSequence.。因为这是Koltin中的内联函数与扩展功能。在后面的章节中会给大家讲解到的。这里主要分析高阶函数，故而这里不多做分析。
2. 该函数返回一个Int类型的值。并且接受了一个selector()函数作为该函数的参数。其中，selector()函数接受一个Char类型的参数，并且返回一个Int类型的值。
3. 定义一个sum变量，并且循环这个字符串，循环一次调用一次selector()函数并加上sum。用作累加。其中this关键字代表字符串本身。

所以这个函数的作用是：**把字符串中的每一个字符转换为Int的值，用于累加，最后返回累加的值**

例：

val testStr = "abc"

val sum = testStr.sumBy { it.toInt() }

println(sum)

复制代码

输出结果为：

294 // 因为字符a对应的值为97,b对应98，c对应99，故而该值即为 97 + 98 + 99 = 294

复制代码

### 1.2、将函数用作一个函数的返回值的高阶函数。

这里使用官网上的一个例子来讲解。lock()函数，先看一看他的源码实现

fun <T> lock(lock: Lock, body: () -> T): T {

lock.lock()

try {

return body()

}

finally {

lock.unlock()

}

}

复制代码

源码说明：

1. 这其中用到了kotlin中泛型的知识点，这里赞不考虑。我会在后续的文章为大家讲解。
2. 从源码可以看出，该函数接受一个Lock类型的变量作为参数1，并且接受一个无参且返回类型为T的函数作为参数2.
3. 该函数的返回值为一个函数，我们可以看这一句代码return body()可以看出。

例：使用lock函数，下面的代码都是伪代码，我就是按照官网的例子直接拿过来用的

fun toBeSynchronized() = sharedResource.operation()

val result = lock(lock, ::toBeSynchronized)

复制代码

其中，::toBeSynchronized即为对函数toBeSynchronized()的引用，其中关于双冒号::的使用在这里不做讨论与讲解。

上面的写法也可以写作：

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

复制代码

### 1.3、高阶函数的使用

在上面的两个例子中，我们出现了str.sumBy{ it.toInt }这样的写法。其实这样的写法在前一章节Lambda使用中已经讲解过了。这里主要讲高阶函数中对Lambda语法的简写。

从上面的例子我们的写法应该是这样的：

str.sumBy( { it.toInt } )

复制代码

但是根据Kotlin中的约定，即当函数中只有一个函数作为参数，并且您使用了lambda表达式作为相应的参数，则可以省略函数的小括号()。故而我们可以写成：

str.sumBy{ it.toInt }

复制代码

还有一个约定，即当函数的最后一个参数是一个函数，并且你传递一个lambda表达式作为相应的参数，则可以在圆括号之外指定它。故而上面例2中的代码我们可写成：

val result = lock(lock){

sharedResource.operation()

}

复制代码

## 二、自定义高阶函数

我记得在上一章节中中我们写了一个例子：

// 源代码

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

return a + b

}

fun sum(num1 : Int , num2 : Int) : Int{

return num1 + num2

}

// 调用

test(10,sum(3,5)) // 结果为：18

// lambda

fun test(a : Int , b : (num1 : Int , num2 : Int) -> Int) : Int{

return a + b.invoke(3,5)

}

// 调用

test(10,{ num1: Int, num2: Int -> num1 + num2 }) // 结果为：18

复制代码

可以看出上面的代码中，直接在我的方法体中写死了数值，这在开发中是很不合理的，并且也不会这么写。上面的例子只是在阐述Lambda的语法。接下来我另举一个例子：

例：传入两个参数，并传入一个函数来实现他们不同的逻辑

例：

private fun resultByOpt(num1 : Int , num2 : Int , result : (Int ,Int) -> Int) : Int{

return result(num1,num2)

}

private fun testDemo() {

val result1 = resultByOpt(1,2){

num1, num2 -> num1 + num2

}

val result2 = resultByOpt(3,4){

num1, num2 -> num1 - num2

}

val result3 = resultByOpt(5,6){

num1, num2 -> num1 \* num2

}

val result4 = resultByOpt(6,3){

num1, num2 -> num1 / num2

}

println("result1 = $result1")

println("result2 = $result2")

println("result3 = $result3")

println("result4 = $result4")

}

复制代码

输出结果为：

result1 = 3

result2 = -1

result3 = 30

result4 = 2

复制代码

这个例子是根据传入不同的Lambda表达式，实现了两个数的+、-、\*、/。  
当然了，在实际的项目开发中，自己去定义高阶函数的实现是很少了，因为用系统给我们提供的高阶函数已经够用了。不过，当我们掌握了Lambda语法以及怎么去定义高阶函数的用法后。在实际开发中有了这种需求的时候也难不倒我们了。

## 三、常用的标准高阶函数介绍

下面介绍几个Kotlin中常用的标准高阶函数。熟练的用好下面的几个函数，能减少很多的代码量，并增加代码的可读性。下面的几个高阶函数的源码几乎上都出自Standard.kt文件

### 3.1、TODO函数

这个函数不是一个高阶函数，它只是一个抛出异常以及测试错误的一个普通函数。

此函数的作用：显示抛出NotImplementedError错误。NotImplementedError错误类继承至Java中的Error。我们看一看他的源码就知道了：

public class NotImplementedError(message: String = "An operation is not implemented.") : Error(message)

复制代码

TODO函数的源码

@kotlin.internal.InlineOnly

public inline fun TODO(): Nothing = throw NotImplementedError()

@kotlin.internal.InlineOnly

public inline fun TODO(reason: String): Nothing =

throw NotImplementedError("An operation is not implemented: $reason")

复制代码

举例说明：

fun main(args: Array<String>) {

TODO("测试TODO函数，是否显示抛出错误")

}

复制代码

输出结果为：

如果调用TODO()时，不传参数的，则会输出An operation is not implemented.

### 3.2 、run()函数

run函数这里分为两种情况讲解，因为在源码中也分为两个函数来实现的。采用不同的run函数会有不同的效果。

#### 3.2.1、run()

我们看下其源码：

public inline fun <R> run(block: () -> R): R {

contract {

callsInPlace(block, InvocationKind.EXACTLY\_ONCE)

}

return block()

复制代码

}

关于contract这部分代码小生也不是很懂其意思。在一些大牛的blog上说是其编辑器对上下文的推断。但是我也不知道对不对，因为在官网中，对这个东西也没有讲解到。不过这个单词的意思是契约，合同等等意思。我想应该和这个有关。在这里我就不做深究了。主要讲讲run{}函数的用法其含义。

这里我们只关心return block()这行代码。从源码中我们可以看出，run函数仅仅是执行了我们的block()，即一个Lambda表达式，而后返回了执行的结果。

**用法1：**

当我们需要执行一个代码块的时候就可以用到这个函数,并且这个代码块是独立的。即我可以在run()函数中写一些和项目无关的代码，因为它不会影响项目的正常运行。

例: 在一个函数中使用

private fun testRun1() {

val str = "kotlin"

run{

val str = "java" // 和上面的变量不会冲突

println("str = $str")

}

println("str = $str")

}

复制代码

输出结果：

str = java

str = kotlin

复制代码

**用法2：**

因为run函数执行了我传进去的lambda表达式并返回了执行的结果，所以当一个业务逻辑都需要执行同一段代码而根据不同的条件去判断得到不同结果的时候。可以用到run函数

例：都要获取字符串的长度。

val index = 3

val num = run {

when(index){

0 -> "kotlin"

1 -> "java"

2 -> "php"

3 -> "javaScript"

else -> "none"

}

}.length

println("num = $num")

复制代码

输出结果为：

num = 10

复制代码

当然这个例子没什么实际的意义。

#### 3.2.2、T.run()

其实T.run()函数和run()函数差不多，关于这两者之间的差别我们看看其源码实现就明白了：

public inline fun <T, R> T.run(block: T.() -> R): R {

contract {

callsInPlace(block, InvocationKind.EXACTLY\_ONCE)

}

return block()

}

复制代码

从源码中我们可以看出，block()这个函数参数是一个扩展在T类型下的函数。这说明我的block()函数可以可以使用当前对象的上下文。所以**当我们传入的lambda表达式想要使用当前对象的上下文的时候，我们可以使用这个函数。**

**用法：**

这里就不能像上面run()函数那样当做单独的一个代码块来使用。

例：

val str = "kotlin"

str.run {

println( "length = ${this.length}" )

println( "first = ${first()}")

println( "last = ${last()}" )

}

复制代码

输出结果为：

length = 6

first = k

last = n

复制代码

在其中，可以使用this关键字，因为在这里它就代码str这个对象，也可以省略。因为在源码中我们就可以看出，block() 就是一个T类型的扩展函数。

这在实际的开发当中我们可以这样用：

例： 为TextView设置属性。

val mTvBtn = findViewById<TextView>(R.id.text)

mTvBtn.run{

text = "kotlin"

textSize = 13f

...

}

复制代码

### 3.3 、with()函数

其实with()函数和T.run()函数的作用是相同的，我们这里看下其实现源码：

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

contract {

callsInPlace(block, InvocationKind.EXACTLY\_ONCE)

}

return receiver.block()

}

复制代码

这里我们可以看出和T.run()函数的源代码实现没有太大的差别。故而这两个函数的区别在于：

1. with是正常的高阶函数，T.run()是扩展的高阶函数。
2. with函数的返回值指定了receiver为接收者。

故而上面的T.run()函数的列子我也可用with来实现相同的效果：

例：

val str = "kotlin"

with(str) {

println( "length = ${this.length}" )

println( "first = ${first()}")

println( "last = ${last()}" )

}

复制代码

输出结果为：

length = 6

first = k

last = n

复制代码

为TextView设置属性，也可以用它来实现。这里我就不举例了。

在上面举例的时候，都是正常的列子，这里举一个特例：当我的对象可为null的时候，看两个函数之间的便利性。

例：

val newStr : String? = "kotlin"

with(newStr){

println( "length = ${this?.length}" )

println( "first = ${this?.first()}")

println( "last = ${this?.last()}" )

}

newStr?.run {

println( "length = $length" )

println( "first = ${first()}")

println( "last = ${last()}" )

}

复制代码

从上面的代码我们就可以看出，当我们使用对象可为null时，使用T.run()比使用with()函数从代码的可读性与简洁性来说要好一些。当然关于怎样去选择使用这两个函数，就得根据实际的需求以及自己的喜好了。

### 3.4、T.apply()函数

我们先看下T.apply()函数的源码：

public inline fun <T> T.apply(block: T.() -> Unit): T {

contract {

callsInPlace(block, InvocationKind.EXACTLY\_ONCE)

}

block()

return this

}

复制代码

从T.apply()源码中在结合前面提到的T.run()函数的源码我们可以得出,这两个函数的逻辑差不多，唯一的区别是T,apply执行完了block()函数后，返回了自身对象。而T.run是返回了执行的结果。

故而： T.apply的作用除了实现能实现T.run函数的作用外，还可以后续的再对此操作。下面我们看一个例子：

例：为TextView设置属性后，再设置点击事件等

val mTvBtn = findViewById<TextView>(R.id.text)

mTvBtn.apply{

text = "kotlin"

textSize = 13f

...

}.apply{

// 这里可以继续去设置属性或一些TextView的其他一些操作

}.apply{

setOnClickListener{ .... }

}

复制代码

或者：设置为Fragment设置数据传递

// 原始方法

fun newInstance(id : Int , name : String , age : Int) : MimeFragment{

val fragment = MimeFragment()

fragment.arguments.putInt("id",id)

fragment.arguments.putString("name",name)

fragment.arguments.putInt("age",age)

return fragment

}

// 改进方法

fun newInstance(id : Int , name : String , age : Int) = MimeFragment().apply {

arguments.putInt("id",id)

arguments.putString("name",name)

arguments.putInt("age",age)

}

复制代码

### 3.5、T.also()函数

关于T.also函数来说，它和T.apply很相似，。我们先看看其源码的实现：

public inline fun <T> T.also(block: (T) -> Unit): T {

contract {

callsInPlace(block, InvocationKind.EXACTLY\_ONCE)

}

block(this)

return this

}

复制代码

从上面的源码在结合T.apply函数的源码我们可以看出： T.also函数中的参数block函数传入了自身对象。故而这个函数的作用是用用block函数调用自身对象，最后在返回自身对象

这里举例一个简单的例子，并用实例说明其和T.apply的区别

例：

"kotlin".also {

println("结果：${it.plus("-java")}")

}.also {

println("结果：${it.plus("-php")}")

}

"kotlin".apply {

println("结果：${this.plus("-java")}")

}.apply {

println("结果：${this.plus("-php")}")

}

复制代码

他们的输出结果是相同的：

结果：kotlin-java

结果：kotlin-php

结果：kotlin-java

结果：kotlin-php

复制代码

从上面的实例我们可以看出，他们的区别在于，T.also中只能使用it调用自身,而T.apply中只能使用this调用自身。因为在源码中T.also是执行block(this)后在返回自身。而T.apply是执行block()后在返回自身。这就是为什么在一些函数中可以使用it,而一些函数中只能使用this的关键所在

### 3.6、T.let()函数

在前面讲解空安全、可空属性章节中，我们讲解到可以使用T.let()函数来规避空指针的问题。有兴趣的朋友可以去看看我的[otlin——初级篇（六）： 可空类型、空安全、非空断言、类型转换等特性总结](https://link.juejin.im/?target=https%3A%2F%2Fjuejin.im%2Fpost%2F5a5b06f26fb9a01cb42c5206)这篇文章。但是在这篇文章中，我们只讲到了它的使用。故而今天来说一下他的源码实现：

public inline fun <T, R> T.let(block: (T) -> R): R {

contract {

callsInPlace(block, InvocationKind.EXACTLY\_ONCE)

}

return block(this)

}

复制代码

从上面的源码中我们可以得出，它其实和T.also以及T.apply都很相似。而T.let的作用也不仅仅在使用空安全这一个点上。用T.let也可实现其他操作

例：

"kotlin".let {

println("原字符串：$it") // kotlin

it.reversed()

}.let {

println("反转字符串后的值：$it") // niltok

it.plus("-java")

}.let {

println("新的字符串：$it") // niltok-java

}

"kotlin".also {

println("原字符串：$it") // kotlin

it.reversed()

}.also {

println("反转字符串后的值：$it") // kotlin

it.plus("-java")

}.also {

println("新的字符串：$it") // kotlin

}

"kotlin".apply {

println("原字符串：$this") // kotlin

this.reversed()

}.apply {

println("反转字符串后的值：$this") // kotlin

this.plus("-java")

}.apply {

println("新的字符串：$this") // kotlin

}

复制代码

输出结果看是否和注释的结果一样呢：

原字符串：kotlin

反转字符串后的值：niltok

新的字符串：niltok-java

原字符串：kotlin

反转字符串后的值：kotlin

新的字符串：kotlin

原字符串：kotlin

反转字符串后的值：kotlin

新的字符串：kotlin

复制代码

### 3.7、T.takeIf()函数

从函数的名字我们可以看出，这是一个关于条件判断的函数,我们在看其源码实现：

public inline fun <T> T.takeIf(predicate: (T) -> Boolean): T? {

contract {

callsInPlace(predicate, InvocationKind.EXACTLY\_ONCE)

}

return if (predicate(this)) this else null

}

复制代码

从源码中我们可以得出这个函数的作用是：

传入一个你希望的一个条件，如果对象符合你的条件则返回自身，反之，则返回null。

例： 判断一个字符串是否由某一个字符起始，若条件成立则返回自身，反之，则返回null

val str = "kotlin"

val result = str.takeIf {

it.startsWith("ko")

}

println("result = $result")

复制代码

输出结果为：

result = kotlin

复制代码

### 3.8、T.takeUnless()函数

这个函数的作用和T.takeIf()函数的作用是一样的。只是和其的逻辑是相反的。即：传入一个你希望的一个条件，如果对象符合你的条件则返回null，反之，则返回自身。

这里看一看它的源码就明白了。

public inline fun <T> T.takeUnless(predicate: (T) -> Boolean): T? {

contract {

callsInPlace(predicate, InvocationKind.EXACTLY\_ONCE)

}

return if (!predicate(this)) this else null

}

复制代码

这里就举和T.takeIf()函数中一样的例子，看他的结果和T.takeIf()中的结果是不是相反的。

例：

val str = "kotlin"

val result = str.takeUnless {

it.startsWith("ko")

}

println("result = $result")

复制代码

输出结果为：

result = null

复制代码

### 3.8、repeat()函数

首先，我们从这个函数名就可以看出是关于重复相关的一个函数，再看起源码，从源码的实现来说明这个函数的作用：

public inline fun repeat(times: Int, action: (Int) -> Unit) {

contract { callsInPlace(action) }

for (index in 0..times - 1) {

action(index)

}

}

复制代码

从上面的代码我们可以看出这个函数的作用是：

根据传入的重复次数去重复执行一个我们想要的动作(函数)

例：

repeat(5){

println("我是重复的第${it + 1}次，我的索引为：$it")

}

复制代码

输出结果为：

我是重复的第1次，我的索引为：0

我是重复的第2次，我的索引为：1

我是重复的第3次，我的索引为：2

我是重复的第4次，我的索引为：3

我是重复的第5次，我的索引为：4

复制代码

### 3.9、lazy()函数

关于Lazy()函数来说，它共实现了4个重载函数，都是用于延迟操作，不过这里不多做介绍。因为在实际的项目开发中常用都是用于延迟初始化属性。而关于这一个知识点我在前面的变量与常量已经讲解过了。这里不多做介绍...

如果您有兴趣，可以去看看我的[Kotlin——初级篇（二）：变量、常量、注释的使用](https://link.juejin.im/?target=https%3A%2F%2Fjuejin.im%2Fpost%2F5a39ef7af265da4311205967)这篇文章。

## 四、对标准的高阶函数总结

关于重复使用同一个函数的情况一般都只有T.also、T.let、T.apply这三个函数。而这个三个函数在上面讲解这些函数的时候都用实例讲解了他们的区别。故而这里不做详细实例介绍。并且连贯着使用这些高阶函数去处理一定的逻辑，在实际项目中很少会这样做。一般都是单独使用一个，或者两个、三个这个连贯这用。但是在掌握了这些函数后，我相信您也是可以的。这里由于蝙蝠原因就不做实例讲解了..

关于他们之间的区别，以及他们用于实际项目中在一定的需求下到底该怎样去选择哪一个函数进行使用希望大家详细的看下他们的源码并且根据我前面说写的实例进行分析。

大家也可以参考这两篇文章：  
[掌握Kotlin标准函数：run, with, let, also and apply](https://link.juejin.im/?target=https%3A%2F%2Fjuejin.im%2Fpost%2F5a676159f265da3e3c6c4d82)  
[那些年，我们看不懂的那些Kotlin标准函数](https://link.juejin.im/?target=http%3A%2F%2Fyoungfeng.com%2F2018%2F04%2F27%2Fkotlin%2F%25E9%2582%25A3%25E4%25BA%259B%25E5%25B9%25B4%25EF%25BC%258C%25E6%2588%2591%25E4%25BB%25AC%25E7%259C%258B%25E4%25B8%258D%25E6%2587%2582%25E7%259A%2584%25E9%2582%25A3%25E4%25BA%259BKotlin%25E6%25A0%2587%25E5%2587%2586%25E5%2587%25BD%25E6%2595%25B0%2F)

## 总结

既然我们选择了Kotlin这门编程语言。那其高阶函数时必须要掌握的一个知识点，因为，在系统的源码中，实现了大量的高阶函数操作，除了上面讲解到的标准高阶函数外，对于字符串(String)以及集合等，都用高阶函数去编写了他们的一些常用操作。比如，元素的过滤、排序、获取元素、分组等等  
对于上面讲述到的标准高阶函数，大家一定要多用多实践，因为它们真的能在实际的项目开发中减少大量的代码编写量。

# Kotlin——高级篇（四）：集合（Array、List、Set、Map）基础

在实际的项目开发中，集合的运用可以说是多不胜数。不过Kotlin中的集合运用和Java中还是有很大的差别，他们两者之间，除了集合的类型相同以外，还包含集合的初始化的不同，以及Kotlin对于集合封装特别多的高阶函数以供我们能更简单、更快捷的编写代码。不过在讲解集合之前，我先会对Kotlin中的数组类型做出一个讲解，大家可以当做是对数组Array<T>的一个温故。

## 目录

## 一、数组类型

在Kotlin数组类型不是集合中的一种，但是它又和集合有着太多相似的地方。并且数组和集合可以互换。并且在初始化集合的时候也可以传入一个数组。用于数组类型在前面的章节已经讲解过了，这里就不在多做累述。有兴趣的朋友可以去看我前面关于数据类型的文章。  
[Kotlin——初级篇（三）：数据类型详解](https://link.juejin.im/?target=https%3A%2F%2Fjuejin.im%2Fpost%2F5a36020b6fb9a0451543f5c8)

这里只介绍几个常用的方法。其实在文章的后面，也对这些方法做出了讲解。

* 用arr[index]的获取元素。
* 用arr.component1() ... arr.component5()获取数组的前5个元素。同样适用与集合。
* 用arr.reverse()反转元素。
* 至于其他处理的元素，在文章的尾部都有说明。这也是我把数组类型与集合类型放在同一文章讲解的原因。

例1：使用componentX()函数

val arr = arrayOf("1",2,3,4)

println(arr.component1())

println(arr.component3())

// 程序崩溃，因为元素只有4个，所以在不确定元素个数的情况，慎用这些函数，还是使用遍历安全些。

println(arr.component5())

复制代码

输出结果：

1

3

复制代码

例2 ：反转元素

val arr = arrayOf("1",2,3,4)

arr.reverse()

// 文章后面会讲解forEach高阶函数。比for循环简洁多了

for (index in arr){

print("$index \t")

}

复制代码

输出结果：

4 3 2 1

复制代码

## 二、集合类型

* Kotlin中的集合和其他语言不同的是，Kotlin集合可分为可变和不可变集合。
* 在Kotlin中，集合类型包含三种类型：它们分别是：List、Set、Map,他们之间存在以下几个异同点：
  1. 它们都是接口，并不是实际的类。
  2. 它们只实现了isEmpty()、size、contains()等函数以及属性。
  3. List<E>和Set<E>都继承至Collection<out E>接口,且Collection<out E>继承于Iterable<out T>接口。而Map<K,V>是独立出来的一个接口。这一点和Java相同。
  4. 这三种集合类型分别有存在MutableList<E>、MutableSet<E>、MutableMap<K,V>接口，这些接口中提供了改变、操作集合的方法。例如add()、clear()、remove()等函数。

有以上三点我们可出，在定义集合类型变量的时候如果使用List<E>、Set<E>、Map<K,V>声明的时候该集合则是不可变集合，而使用MutableList<E>、MutableSet<E>、MutableMap<K,V>的时候该集合才是可变类型集合。这里我就不提供源码来分析了，有兴趣的可以看一看源码！源码在kotlin\collections\Collections.kt文件

下面对几个集合类型进行一一的讲解。

### 2.1、List类型

我们知道，一个接口是不能直接实例化的，那我们要初始化一个怎么做呢？其实Kotlin给我们提供了相应的标准库函数去处理。

* 声明并初始化List的集合：使用listOf(..)函数
* 声明并初始化MutableList的集合：使用mutableListOf(..)函数

例1：使用listOf()初始化不可变的List类型集合

val arr = arrayOf("1","2",3,4,5)

val list1 = listOf(1,2,"3",4,"5") // 随意创建

val list2 = listOf<String>("1","2","3","4","5") // 确定元素的值类型

val list3 = listOf(arr) // 可传入一个数组

以下代码是错误的。因为List<E>只能是不可变集合。而add、remove、clear等函数时MutableList中的函数

list1.add()

list1.remove

...

// 遍历

for(value in list1){

print("$value \t")

}

复制代码

输出结果：

1 2 3 4 5

复制代码

例2：使用mutableListOf()初始化可变的List类型集合

val arr = arrayOf("1",2,3,4)

val mutableList1 = mutableListOf(1,2,"3",4,"5") // 随意创建

val mutableList2 = mutableListOf<String>("1","2","3","4","5") // 确定元素的值类型

val mutableList3 = mutableListOf(arr) // 可传入一个数组

val mutableList : ArrayList<String> // 这里的ArrayList<>和Java里面的ArrayList一致

mutableList1.add("6") // 添加元素

mutableList1.add("7")

mutableList1.remove(1) // 删除某一元素

// 遍历

for(value in mutableList1){

print("$value \t")

}

mutableList1.clear() // 清空集合

复制代码

输出结果为：

2 3 4 5 6 7

复制代码

### 2.2、Set类型

Set类型集合的使用和List类型集合大致相同。这里不做详细的介绍，只讲解它和List类型集合不同的地方。

* 声明并初始化Set的集合：使用setOf(..)函数
* 声明并初始化MutableSet的集合：使用mutableSetOf(..)函数

例1： 声明并初始化

val set1 = setOf(1,2,"3","4","2",1,2,3,4,5)

val mutableSet1 = mutableSetOf(1,2,"3","4","2",1,2,3,4,5)

val mutableSet2 : HashSet<String> // 这里的HashSet<>和Java里面的HashSet<>一致

复制代码

例2 ：遍历集合，看效果与预计的有什么不同

// 遍历

for(value in set1){

print("$value \t")

}

复制代码

输出结果：

1 2 3 4 2 3 4 5

复制代码

在我们预计的效果中，遍历的结果应该为：1 2 3 4 2 1 2 3 4 5,但是结果却少了一个1 2。那么我们可以看出，Set类型集合会把重复的元素去除掉。这一点和Java是不谋而合的。这个特性也是Set类型集合与List集合类型的区别所在。

### 2.3、Map类型

Map<K,V>类型集合和List以及Set都有着差别。下面我们看Map类型集合的声明及初始化。

同前面两种类型一样，Map同样也分为不可变与可变集合。其中：

* 不可变的Map类型集合的初始化使用：mapOf()函数
* 可变的Map类型集合的初始化使用：mutableMapOf()函数

不过初始化和前面两种类型有差别，Map集合类型是一种以键-值对的方式出现。例：

// 以键值对的形式出现，键与值之间使用to

val map1 = mapOf("key1" to 2 , "key2" to 3)

val map2 = mapOf<Int,String>(1 to "value1" , 2 to "value2")

val mutableMap = mutableMapOf("key1" to 2 , "key1" to 3)

val hashMap = hashMapOf("key1" to 2 , "key1" to 3) // 同Java中的HashMap

map2.forEach{

key,value -> println("$key \t $value")

}

复制代码

输出结果为：

1 value1

2 value2

复制代码

注意：**当我们的键存在重复时，集合会过滤掉之前重复的元素**。

例：

val map = val map1 = mapOf("key1" to 2 , "key1" to 3 , "key1" to "value1" , "key2" to "value2")

map.forEach{

key,value -> println("$key \t $value")

}

复制代码

输出结果为：

key1 value1

key2 value2

复制代码

从上面的例子可以看出，当key值为key1时，元素只保留了最后一个元素。而过滤掉了之前key值相同的所有元素。

## 三、 集合类型的协变

试想一下，当一个集合赋值给另外一个集合时，这里以List<E>举例，如果两个集合的类型也就是E类型相同时，赋值是没有问题的。如果类型不同的情况，当E继承自M时。你就可以把List<E>赋值给List<M>了。这种情况称之为协变

我这里举两个例子

例1：

open class Person(val name : String , val age : Int){

override fun toString(): String {

return "Person(name='$name', age=$age)"

}

}

class Student(name: String, age : Int, cls : String) : Person(name, age)

// 注意：Any是kotlin中的超类，故而Student类也是继承自Any的。这里你可以换成Person类结果是相同的

var listPerson: List<Any>

val listStudent : List<Student> = listOf(Student("张三",12,"一班"),Student("王五",20,"二班"))

listPerson = listStudent

listPerson.forEach { println(it.toString()) }

复制代码

输出结果：

Person(name='张三', age=12)

Person(name='王五', age=20)

复制代码

例2：当集合的类型相同或有继承关系时，一个集合使用MutableList，一个集合使用List的情况。

var mutableListPerson: MutableList<Person>

val mutableListStudent : List<Student> = listOf(Student("张三",12,"一班"),Student("王五",20,"二班"))

mutableListPerson = mutableListStudent.toMutableList()

mutableListPerson.add(Person("a",15))

mutableListPerson.add(Person("b",45))

mutableListPerson.forEach { println(it.toString()) }

复制代码

输出结果为：

Person(name='张三', age=12)

Person(name='王五', age=20)

Person(name='a', age=15)

Person(name='b', age=45)

复制代码

看上面的实例2，使用了一个toMutableList()函数，其实这个函数的意思是把List转换成了MutableList。在以下的源码中我们可以看出：其实是实例化了一个ArrayList。

public fun <T> Collection<T>.toMutableList(): MutableList<T> {

return ArrayList(this)

}

public fun <T> Iterable<T>.toMutableList(): MutableList<T> {

if (this is Collection<T>)

return this.toMutableList()

return toCollection(ArrayList<T>())

}

复制代码

Set、Map集合的协变和上面的代码都相差不多，调用不同的转换函数罢了。除了toMutableList()函数以外，还有着toList()、toHashSet()、toSet()等等函数。这些函数都是在Iterable接口的拓展函数。大家有兴趣可以自己去看看源码，这里不做详细的累述。

## 四、一些常用的处理集合类型的拓展函数

除了上面讲到的toList()、toSet()、toHastSet()、toMutableList()、toSet()、toIntArray()等等拓展函数之外。还有一些常用的拓展的高阶函数。这里列举几个说明。并实例分析他们的作用。所有的源码都在kotlin\collections\\_Collections.kt文件。

不过这里由于文章篇幅的原因：这一节的内容会在下一章文章讲解。

请参见[Kotlin——高级篇（五）：集合之常用操作符汇总](https://link.juejin.im/?target=https%3A%2F%2Fjuejin.im%2Fpost%2F5b1f7699f265da6e155d5965)

## 总结

在这篇文章中，详细的讲解到了集合的几种类型的声明与使用，并且也对数组类型Array<T>温故了一遍。其实这篇文章的内容并不是很多，大家主要记住集合类型初始化的几个标准函数，以及集合的类型协变。在下一篇文章中会对处理集合与数组的常见函数做出一个讲解以及源码的剖析。

# Kotlin——高级篇（五）：集合之常用操作符汇总

在上一篇文章[Kotlin——高级篇（四）：集合（Array、List、Set、Map）基础](https://link.juejin.im/?target=https%3A%2F%2Fjuejin.im%2Fpost%2F5ab7a9c4f265da2377196038)中讲解到了数组Array<T>、集合(List、Set、Map)的定义与初始化。但是由于篇幅的原因，未讲解到操作他们的常用高阶函数。故而今天这篇文章详细的讲解这些函数。对他们的作用进行剖。并实例讲解他们的应用场景。当看完这篇文章，我相信你能对集合使用的得心应手。

## 目录

## 一、转换类

在上一篇文章中讲解数组(Array<T>)的与集合的时候也提到了关于集合转换的高阶函数。下面用源码的角度去分析并用实例去讲解其转换的过程。

### 1.1、转换为数组

当我们声明一个集合，可以把这个集合根据调用集合类相应的高阶函数来转换成相应的数组。集合类提供了toIntArray()、toDoubleArray()、toFloatArray()、toBetArray()等高阶函数去处理。下面提供一个函数的源码，其他函数的源码处理逻辑是相同的，有兴趣的朋友可以去看看这个源码类。

源码：

public fun Collection<Int>.toIntArray(): IntArray {

val result = IntArray(size)

var index = 0

for (element in this)

result[index++] = element

return result

}

复制代码

解释：从上面的源码，我们可以看出，该函数的返回类型时IntArray类型,即表示一个Int类型的数组。该函数的逻辑是定义一个Int类型的数组，然后遍历集合，把集合的元素一个一个的添加到定义的数组当中。

实例：

fun listToArray(){

val list = listOf<Int>(1,2,3,4,5,6) // 声明一个Int类型的List

val listArray = list.toIntArray() // 转换

println(list.javaClass.toString()) // 打印list的类型

println(listArray.javaClass.toString()) // 打印listArray的类型

println(listArray[1])

}

复制代码

输出结果：

变量list的类型为：class java.util.LinkedHashSet

变量listArray的类型为：class [I

2

复制代码

### 1.2、转换为集合

我们知道在Kotlin中，集合可分为不可变集合与可变集合。我们声明一个集合或者数组，可以转换成相应类型的集合。调用toXXX()转换成不可变集合。调用toMutableXXX()转换为可变集合。集合类提供了toList()、toMutableList()、toSet()、toMutableSet()、toHashSet()、toMap()等高阶函数去处理。同理也是从源码的角度去分析。

源码：这个是集合转集合的源码。数组转集合的源码还要简单些。

public fun <T> Iterable<T>.toList(): List<T> {

if (this is Collection) {

return when (size) {

0 -> emptyList()

1 -> listOf(if (this is List) get(0) else iterator().next())

else -> this.toMutableList()

}

}

return this.toMutableList().optimizeReadOnlyList()

}

复制代码

解释：首先判断是不是一个集合，如果是一个集合的情况，则根据集合的元素个数来执行相应的逻辑，当元素的个数为0时，返回一个空集合。当为1个的时候，用listOf()去初始化一个List集合。在这个新集合中去这个判断原集合的类型是不是返回集合的类型，如果是，则获取原集合的第一个元素作为新集合的元素返回。反之，则遍历原集合的元素。当原集合个数不为0或1时，使用toMutableList()转换成list。如果不是集合，则直接使用toMutableList()转换。这里的optimizeReadOnlyList()函数的逻辑即是上面原集合的逻辑。

解释起来很复杂，还是用代码说话吧...

实例：

// 数组转集合

fun arrayToList() {

val arr = arrayOf(1,3,5,7,9)

val list = arr.toList()

println("变量arr的类型为：${arr.javaClass}")

println("变量list的类型为：${list.javaClass}")

println(list[1])

}

// 集合转集合，这里用Set转List

fun listToList(){

val set = setOf(1)

val setTolist = set.toList()

println("变量set的类型为：${set.javaClass}")

println("变量setTolist的类型为：${setTolist.javaClass}")

println(setTolist[0])

}

复制代码

输出结果为：

变量arr的类型为：class [Ljava.lang.Integer;

变量list的类型为：class java.util.ArrayList

3

变量set的类型为：class java.util.Collections$SingletonSet

变量setTolist的类型为：class java.util.Collections$SingletonList

1

复制代码

关于集合转换的问题就讲到这里，有兴趣的同学可以去看看源码的实现。

## 二、操作类

关于集合操作类的函数大致分为六类，他们几乎上囊括了源码中实现的操作方法。下面对这6类操作符进行一一的讲解，不过这里就不对他们的源码进行分析了。大家有兴趣的话可以去看看他们的源码实现。几乎上都存在于\_Collections.kt这个文件中

### 2.1、元素操作符

元素操作符在这里包括：

* contains(元素) : 检查集合中是否包含指定的元素，若存在则返回true，反之返回false
* elementAt(index) : 获取对应下标的元素。若下标越界，会抛出IndexOutOfBoundsException（下标越界）异常，同get(index)一样
* elementAtOrElse(index,{...}) : 获取对应下标的元素。若下标越界，返回默认值，此默认值就是你传入的下标的运算值
* elementAtOrNull(index) : 获取对应下标的元素。若下标越界，返回null
* first() : 获取第一个元素，若集合为空集合，这会抛出NoSuchElementException异常
* first{} : 获取指定元素的第一个元素。若不满足条件，则抛出NoSuchElementException异常
* firstOrNull() : 获取第一个元素，若集合为空集合，返回null
* firstOrNull{} : 获取指定元素的第一个元素。若不满足条件，返回null
* getOrElse(index,{...}) : 同elementAtOrElse一样
* getOrNull(index) : 同elementAtOrNull一样
* last() : 同first()相反
* last{} : 同first{}相反
* lastOrNull{} : 同firstOrNull()相反
* lastOrNull() : 同firstOrNull{}相反
* indexOf(元素) : 返回指定元素的下标，若不存在，则返回-1
* indexOfFirst{...} : 返回第一个满足条件元素的下标，若不存在，则返回-1
* indexOfLast{...} : 返回最后一个满足条件元素的下标，若不存在，则返回-1
* single() : 若集合的长度等于0,则抛出NoSuchElementException异常，若等于1，则返回第一个元素。反之，则抛出IllegalArgumentException异常
* single{} : 找到集合中满足条件的元素，若元素满足条件，则返回该元素。否则会根据不同的条件，抛出异常。这个方法慎用
* singleOrNull() : 若集合的长度等于1,则返回第一个元素。否则，返回null
* singleOrNull{} : 找到集合中满足条件的元素，若元素满足条件，则返回该元素。否则返回null
* forEach{...} : 遍历元素。一般用作元素的打印
* forEachIndexed{index,value} : 遍历元素，可获得集合中元素的下标。一般用作元素以及下标的打印
* componentX() ： 这个函数在前面的章节中提过多次了。用于获取元素。其中的X只能代表1..5。详情可看下面的例子

例：

val list = listOf("kotlin","Android","Java","PHP","Python","IOS")

println(" ------ contains -------")

println(list.contains("JS"))

println(" ------ elementAt -------")

println(list.elementAt(2))

println(list.elementAtOrElse(10,{it}))

println(list.elementAtOrNull(10))

println(" ------ get -------")

println(list.get(2))

println(list.getOrElse(10,{it}))

println(list.getOrNull(10))

println(" ------ first -------")

println(list.first())

println(list.first{ it == "Android" })

println(list.firstOrNull())

println(list.firstOrNull { it == "Android" })

println(" ------ last -------")

println(list.last())

println(list.last{ it == "Android" })

println(list.lastOrNull())

println(list.lastOrNull { it == "Android" })

println(" ------ indexOf -------")

println(list.indexOf("Android"))

println(list.indexOfFirst { it == "Android" })

println(list.indexOfLast { it == "Android" })

println(" ------ single -------")

val list2 = listOf("list")

println(list2.single()) // 只有当集合只有一个元素时，才去用这个函数，不然都会抛出异常。

println(list2.single { it == "list" }) //当集合中的元素满足条件时，才去用这个函数，不然都会抛出异常。若满足条件返回该元素

println(list2.singleOrNull()) // 只有当集合只有一个元素时，才去用这个函数，不然都会返回null。

println(list2.singleOrNull { it == "list" }) //当集合中的元素满足条件时，才去用这个函数，不然返回null。若满足条件返回该元素

println(" ------ forEach -------")

list.forEach { println(it) }

list.forEachIndexed { index, it -> println("index : $index \t value = $it") }

println(" ------ componentX -------")

println(list.component1()) // 等价于`list[0] <=> list.get(0)`

println(list.component2()) // 等价于`list[1] <=> list.get(1)`

println(list.component3()) // 等价于`list[2] <=> list.get(2)`

println(list.component4()) // 等价于`list[3] <=> list.get(3)`

println(list.component5()) // 等价于`list[4] <=> list.get(4)`

复制代码

输出结果为：

------ contains -------

false

------ elementAt -------

Java

10

null

------ get -------

Java

10

null

------ first -------

kotlin

Android

kotlin

Android

------ last -------

IOS

Android

IOS

Android

------ indexOf -------

1

1

1

------ single -------

list

list

list

list

------ forEach -------

kotlin

Android

Java

PHP

Python

IOS

index : 0 value = kotlin

index : 1 value = Android

index : 2 value = Java

index : 3 value = PHP

index : 4 value = Python

index : 5 value = IOS

------ componentX -------

kotlin

Android

Java

PHP

Python

复制代码

### 2.2、顺序操作符

顺序操作符包括：

* reversed() : 反序。即和初始化的顺序反过来。
* sorted() : 自然升序。
* sortedBy{} : 根据条件升序，即把不满足条件的放在前面，满足条件的放在后面
* sortedDescending() : 自然降序。
* sortedByDescending{} : 根据条件降序。和sortedBy{}相反

其中从源码的角度说：

1. sorted()函数的核心是sort()函数。但这个sort()函数我不能直接调用罢了。
2. reversed()函数调用的是Java中的reversed()函数。
3. 其余的3个函数实现的核心为sortWith()函数。故而我们也可以用sortWith()来实现其他的排序方式

例：

val list1 = listOf(-1,-3,1,3,5,6,7,2,4,10,9,8)

// 反序

println(list1.reversed())

// 升序

println(list1.sorted())

// 根据条件升序，即把不满足条件的放在前面，满足条件的放在后面

println(list1.sortedBy { it % 2 == 0})

// 降序

println(list1.sortedDescending())

// 根据条件降序，和`sortedBy{}`相反

println(list1.sortedByDescending { it % 2 == 0 })

复制代码

输出结果为：

[8, 9, 10, 4, 2, 7, 6, 5, 3, 1, -3, -1]

[-3, -1, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

[-1, -3, 1, 3, 5, 7, 9, 6, 2, 4, 10, 8]

[10, 9, 8, 7, 6, 5, 4, 3, 2, 1, -1, -3]

[6, 2, 4, 10, 8, -1, -3, 1, 3, 5, 7, 9]

复制代码

### 2.3、映射操作符

映射操作符包括：

* map{...} : 把每个元素按照特定的方法进行转换，组成一个新的集合。
* mapNotNull{...} : 同map{}函数的作用相同，只是过滤掉转换之后为null的元素
* mapIndexed{index,result} : 把每个元素按照特定的方法进行转换，只是其可以操作元素的下标(index)，组成一个新的集合。
* mapIndexedNotNull{index,result} : 同mapIndexed{}函数的作用相同，只是过滤掉转换之后为null的元素
* flatMap{...} : 根据条件合并两个集合，组成一个新的集合。
* groupBy{...} : 分组。即根据条件把集合拆分为为一个Map<K,List<T>>类型的集合。具体看实例

例：

val list1 = listOf("kotlin","Android","Java","PHP","JavaScript")

println(list1.map { "str-".plus(it) })

println(list1.mapNotNull { "str-".plus(it) })

println(list1.mapIndexed { index, str ->

index.toString().plus("-").plus(str)

})

println(list1.mapIndexedNotNull { index, str ->

index.toString().plus("-").plus(str)

})

println( list1.flatMap { listOf(it,"new-".plus(it)) })

println(list1.groupBy { if (it.startsWith("Java")) "big" else "latter" })

复制代码

输出结果为：

[str-kotlin, str-Android, str-Java, str-PHP, str-JavaScript]

[str-kotlin, str-Android, str-Java, str-PHP, str-JavaScript]

[0-kotlin, 1-Android, 2-Java, 3-PHP, 4-JavaScript]

[0-kotlin, 1-Android, 2-Java, 3-PHP, 4-JavaScript]

[kotlin, new-kotlin, Android, new-Android, Java, new-Java, PHP, new-PHP, JavaScript, new-JavaScript]

{latter=[kotlin, Android, PHP], big=[Java, JavaScript]}

复制代码

### 2.4、过滤操作符

过滤操作符包括：

* filter{...} : 把不满足条件的元素过滤掉
* filterIndexed{...} : 和filter{}函数作用类似，只是可以操作集合中元素的下标（index）
* filterNot{...} : 和filter{}函数的作用相反
* filterNotNull() : 过滤掉集合中为null的元素。
* take(num) : 返回集合中前num个元素组成的集合
* takeWhile{...} : 循环遍历集合，从第一个元素开始遍历集合，当第一个出现不满足条件元素的时候，退出遍历。然后把满足条件所有元素组成的集合返回。
* takeLast(num) : 返回集合中后num个元素组成的集合
* takeLastWhile{...} : 循环遍历集合，从最后一个元素开始遍历集合，当第一个出现不满足条件元素的时候，退出遍历。然后把满足条件所有元素组成的集合返回。
* drop(num) : 过滤集合中前num个元素
* dropWhile{...} : 相同条件下，和执行takeWhile{...}函数后得到的结果相反
* dropLast(num) : 过滤集合中后num个元素
* dropLastWhile{...} : 相同条件下，和执行takeLastWhile{...}函数后得到的结果相反
* distinct() : 去除重复元素
* distinctBy{...} : 根据操作元素后的结果去除重复元素
* slice : 过滤掉所有不满足执行下标的元素。

例：

val list1 = listOf(-1,-3,1,3,5,6,7,2,4,10,9,8)

val list2 = listOf(1,3,4,5,null,6,null,10)

val list3 = listOf(1,1,5,2,2,6,3,3,7,4,4,8)

println(" ------ filter -------")

println(list1.filter { it > 1 })

println(list1.filterIndexed { index, result ->

index < 5 && result > 3

})

println(list1.filterNot { it > 1 })

println(list2.filterNotNull())

println(" ------ take -------")

println(list1.take(5))

println(list1.takeWhile { it < 5 })

println(list1.takeLast(5))

println(list1.takeLastWhile { it > 5 })

println(" ------ drop -------")

println(list1.drop(5))

println(list1.dropWhile { it < 5 })

println(list1.dropLast(5))

println(list1.dropLastWhile { it > 5 })

println(" ------ distinct -------")

println(list3.distinct())

println(list3.distinctBy { it + 2 })

println(" ------ slice -------")

println(list1.slice(listOf(1,3,5,7)))

println(list1.slice(IntRange(1,5)))

复制代码

输出结果为：

------ filter -------

[3, 5, 6, 7, 2, 4, 10, 9, 8]

[5]

[-1, -3, 1]

[1, 3, 4, 5, 6, 10]

------ take -------

[-1, -3, 1, 3, 5]

[-1, -3, 1, 3]

[2, 4, 10, 9, 8]

[10, 9, 8]

------ drop -------

[6, 7, 2, 4, 10, 9, 8]

[5, 6, 7, 2, 4, 10, 9, 8]

[-1, -3, 1, 3, 5, 6, 7]

[-1, -3, 1, 3, 5, 6, 7, 2, 4]

------ distinct -------

[1, 5, 2, 6, 3, 7, 4, 8]

[1, 5, 2, 6, 3, 7, 4, 8]

------ slice -------

[-3, 3, 6, 2]

[-3, 1, 3, 5, 6]

复制代码

### 2.5、生产操作符

生产操作符包括：

* plus() : 合并两个集合中的元素，组成一个新的集合。也可以使用符号+
* zip : 由两个集合按照相同的下标组成一个新集合。该新集合的类型是：List<Pair>
* unzip : 和zip的作用相反。把一个类型为List<Pair>的集合拆分为两个集合。看下面的例子
* partition : 判断元素是否满足条件把集合拆分为有两个Pair组成的新集合。

例：

val list1 = listOf(1,2,3,4)

val list2 = listOf("kotlin","Android","Java","PHP","JavaScript")

// plus() 和 `+`一样

println(list1.plus(list2))

println(list1 + list2)

// zip

println(list1.zip(list2))

println(list1.zip(list2){ // 组成的新集合由元素少的原集合决定

it1,it2-> it1.toString().plus("-").plus(it2)

})

// unzip

val newList = listOf(Pair(1,"Kotlin"),Pair(2,"Android"),Pair(3,"Java"),Pair(4,"PHP"))

println(newList.unzip())

// partition

println(list2.partition { it.startsWith("Ja") })

复制代码

输出结果为：

[1, 2, 3, 4, kotlin, Android, Java, PHP, JavaScript]

[1, 2, 3, 4, kotlin, Android, Java, PHP, JavaScript]

[(1, kotlin), (2, Android), (3, Java), (4, PHP)]

[1-kotlin, 2-Android, 3-Java, 4-PHP]

([1, 2, 3, 4], [Kotlin, Android, Java, PHP])

([Java, JavaScript], [kotlin, Android, PHP])

复制代码

### 2.6、统计操作符

统计操作符包括：

* any() : 判断是不是一个集合，若是，则在判断集合是否为空，若为空则返回false,反之返回true,若不是集合，则返回hasNext
* any{...} : 判断集合中是否存在满足条件的元素。若存在则返回true,反之返回false
* all{...} : 判断集合中的所有元素是否都满足条件。若是则返回true,反之则返回false
* none() : 和any()函数的作用相反
* none{...} : 和all{...}函数的作用相反
* max() : 获取集合中最大的元素，若为空元素集合，则返回null
* maxBy{...} : 获取方法处理后返回结果最大值对应那个元素的初始值，如果没有则返回null
* min() : 获取集合中最小的元素，若为空元素集合，则返回null
* minBy{...} : 获取方法处理后返回结果最小值对应那个元素的初始值，如果没有则返回null
* sum() : 计算出集合元素累加的结果。
* sumBy{...} : 根据元素运算操作后的结果，然后根据这个结果计算出累加的值。
* sumByDouble{...} : 和sumBy{}相似，不过sumBy{}是操作Int类型数据，而sumByDouble{}操作的是Double类型数据
* average() : 获取平均数
* reduce{...} : 从集合中的第一项到最后一项的累计操作。
* reduceIndexed{...} : 和reduce{}作用相同，只是其可以操作元素的下标(index)
* reduceRight{...} : 从集合中的最后一项到第一项的累计操作。
* reduceRightIndexed{...} : 和reduceRight{}作用相同，只是其可以操作元素的下标(index)
* fold{...} : 和reduce{}类似，但是fold{}有一个初始值
* foldIndexed{...} : 和reduceIndexed{}类似，但是foldIndexed{}有一个初始值
* foldRight{...} : 和reduceRight{}类似，但是foldRight{}有一个初始值
* foldRightIndexed{...} : 和reduceRightIndexed{}类似，但是foldRightIndexed{}有一个初始值

例：

val list1 = listOf(1,2,3,4,5)

println(" ------ any -------")

println(list1.any())

println(list1.any{it > 10})

println(" ------ all -------")

println(list1.all { it > 2 })

println(" ------ none -------")

println(list1.none())

println(list1.none{ it > 2})

println(" ------ max -------")

println(list1.max())

println(list1.maxBy { it + 2 })

println(" ------ min -------")

println(list1.min()) // 返回集合中最小的元素

println(list1.minBy { it + 2 })

println(" ------ sum -------")

println(list1.sum())

println(list1.sumBy { it + 2 })

println(list1.sumByDouble { it.toDouble() })

println(" ------ average -----")

println(list1.average())

println(" ------ reduce -------")

println(list1.reduce { result, next -> result + next})

println(list1.reduceIndexed { index, result, next ->

index + result + next

})

println(list1.reduceRight { result, next -> result + next })

println(list1.reduceRightIndexed {index, result, next ->

index + result + next

})

println(" ------ fold -------")

println(list1.fold(3){result, next -> result + next})

println(list1.foldIndexed(3){index,result, next ->

index + result + next

})

println(list1.foldRight(3){result, next -> result + next})

println(list1.foldRightIndexed(3){index,result, next ->

index + result + next

})

复制代码

输出结果为：

------ any -------

true

false

------ all -------

false

------ none -------

false

false

------ max -------

5

5

------ min -------

1

1

------ sum -------

15

25

15.0

------ average -----

3.0

------ reduce -------

15

25

15

21

------ fold -------

18

28

18

28

复制代码

## 总结

这篇文章篇幅很长，同时内容也很多。对于操作符讲解的不是很全，主要是介绍常用的操作符。并且对他们的含义即实现进行讲解。但是我相信您学习完上面的内容，对于集合的操作应该是得心印手了。同时了解了这些操作符是怎么实现的。

关于Kotlin中对于集合的操作符，我推荐几篇别人的文章：  
[Kotlin系列之常用操作符](https://link.juejin.im/?target=https%3A%2F%2Fblog.csdn.net%2Ffine1938768839%2Farticle%2Fdetails%2F73550989)  
[Kotlin入门第二课：集合操作](https://link.juejin.im/?target=https%3A%2F%2Fwww.cnblogs.com%2Ftgyf%2Fp%2F6892551.html%3Futm_source%3Ditdadao%26utm_medium%3Dreferral)  
[Kotlin集合与它的操作符们](https://link.juejin.im/?target=https%3A%2F%2Fwww.2cto.com%2Fkf%2F201611%2F567100.html)  
上面几篇文章对于Kotlin的全部操作符的介绍与使用讲解的很全面。并且都有例子去实现。

在这最后希望您能给个关注，因为您的关注，是我继续写文章最好的动力。

[源代码](https://link.juejin.im/?target=https%3A%2F%2Fgithub.com%2FJetictors%2FKotlinLearn%2Fblob%2Fmaster%2Fsrc%2Fcom%2Fkotlin%2Fleran%2Fother%2FOperaCollectionDemo.kt)