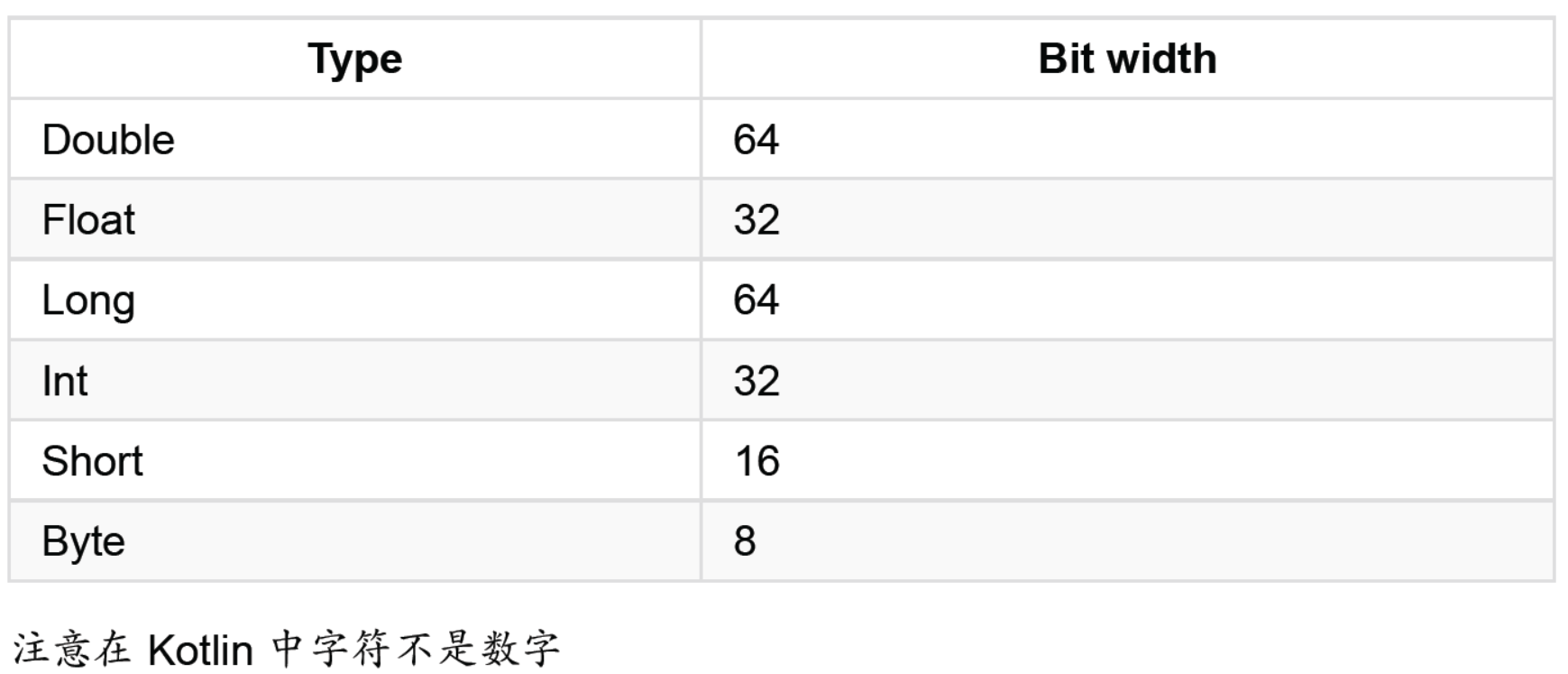
# 第一部分 Kotlin基础篇

## 一 新建Kotlin项目

## 二 Kotlin基础

### 1 kotlin数据类型

#### 1 数字类型



数字的表示方式和java一样

数值常量字面值有以下几种:

十进制: 123

Long 类型用大写 L 标记: 123L

十六进制: 0x0F

二进制: 0b00001011

基本类型

104

注意: 不支持八进制

Kotlin 同样支持浮点数的常规表示方法:

默认 double： 123.5 、 123.5e10

Float 用 f 或者 F 标记: 123.5f

kotlin中数字类型没有隐式转换，必须显示转换

每个数字类型支持如下的转换:

toByte(): Byte

toShort(): Short

toInt(): Int

toLong(): Long

toFloat(): Float

toDouble(): Double

toChar(): Char

运算

Kotlin支持数字运算的标准集，运算被定义为相应的类成员（但编译器会将函数调

用优化为相应的指令）。 参见运算符重载。

位运算，没有特殊字符来表示，而只可用中缀方式调用命名函数，例如:

val x = (1 shl 2) and 0x000FF000

这是完整的位运算列表（只用于 Int 与 Long ）：

shl(bits) – 有符号左移 (Java 的 << )

shr(bits) – 有符号右移 (Java 的 >> )

ushr(bits) – 无符号右移 (Java 的 >>> )

and(bits) – 位与

or(bits) – 位或

xor(bits) – 位异或

inv() – 位非

数比较

相等性检测： a == b 与 a != b

比较操作符： a < b 、 a > b 、 a <= b 、 a >= b

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

#### 2 字符

字符用 Char 类型表示。它们不能直接当作数字

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

我们可以显式把字符转换为 Int 数字：

toInt()//转换

当需要可空引用时，像数字、字符会被装箱。装箱操作不会保留同一性。

#### 3 布尔

布尔用 Boolean 类型表示，它有两个值： true 与 false 。若需要可空引用布尔会被装箱。

内置的布尔运算有：

|| – 短路逻辑或

&& – 短路逻辑与

! - 逻辑非

#### 4 数组

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

我们可以使用库函数 arrayOf() 来创建一个数组并传递元素值给它，这样

arrayOf(1, 2, 3) 创建了 array [1, 2, 3]。 或者，库函数 arrayOfNulls()

可以用于创建一个指定大小的、所有元素都为空的数组。

另一个选项是用接受数组大小以及一个函数参数的 Array 构造函数，用作参数的

函数能够返回给定索引的每个元素初始值：

fun main(args: Array<String>) {

//sampleStart

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

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

asc.forEach { println(it) }

//sampleEnd

}

如上所述， [] 运算符代表调用成员函数 get() 与 set() 。

注意: 与 Java 不同的是，Kotlin 中数组是不型变的（invariant）。这意味着 Kotlin不让我们把 Array<String> 赋值给 Array<Any> ，以防止可能的运行时失败（但是你可以使用 Array<out Any> , 参见类型投影）。

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

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

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

#### 5 字符串

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

fun main(args: Array<String>) {

val str = "abcd"

//sampleStart

for (c in str) {

println(c)

}

//sampleEnd

}

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

Kotlin 有两种类型的字符串字面值: 转义字符串可以有转义字符，以及原始字符串可以包含换行以及任意文本。转义字符串很像 Java 字符串:

val s = "Hello, world!\n"

转义采用传统的反斜杠方式。参见上面的 字符 查看支持的转义序列。

原始字符串 使用三个引号（ """ ）分界符括起来，内部没有转义并且可以包含换

行以及任何其他字符:

val text = """

for (c in "foo")

print(c)

"""

字符串模板

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

fun main(args: Array<String>) {

//sampleStart

val i = 10

println("i = $i") // 输出“i = 10”

//sampleEnd

}

或者用花括号括起来的任意表达式:

fun main(args: Array<String>) {

//sampleStart

val s = "abc"

println("$s.length is ${s.length}") // 输出“abc.length is 3”

//sampleEnd

}

#### 6 引用类型

除以上基本类型之外，其他由类定义的数据类型都是引用类型。

当然String也是引用类型，基本类型的可空类型由于涉及到装箱，也可以看做引用类型。这一点和java类似。

### 2 包

包的概念和命名和java一样，只是kotlin中不要求文件目录和包一一匹配

与 Java 不同，Kotlin 没有单独的“import static”语法； 所有这些声明都用 import关键字导入

### 3 控制流

控制流主要是if、when、for、while。相比java，kotlin用when替代了switch，但是when比switch更加强大

#### 1 If 表达式

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

// 传统用法

var max = a

if (a < b) max = b

// With else

var max: Int

if (a > b) {

max = a

} else {

max = b

}

// 作为表达式

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

if 的分支可以是代码块，最后的表达式作为该块的值：

val max = if (a > b) {

print("Choose a")

a

} else {

print("Choose b")

b

}

如果你使用 if 作为表达式而不是语句（例如：返回它的值或者把它赋给变量），该表达式需要有 else 分支。

#### 2 when 表达式

when 取代了类 C 语言的 switch 操作符。其最简单的形式如下：

when (x) {

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

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

else -> { // 注意这个块

print("x is neither 1 nor 2")

}

}

when 将它的参数与所有的分支条件顺序比较，直到某个分支满足条件。 when既可以被当做表达式使用也可以被当做语句使用。如果它被当做表达式， 符合条件的分支的值就是整个表达式的值，如果当做语句使用， 则忽略个别分支的值。（像if 一样，每一个分支可以是一个代码块，它的值是块中最后的表达式的值。）如果其他分支都不满足条件将会求值 else 分支。 如果 when 作为一个表达式使用，则必须有 else 分支， 除非编译器能够检测出所有的可能情况都已经覆盖了［例如，对于 枚举（ enum ）类条目与密封（ sealed ）类子类型］。

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

when (x) {

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

else -> print("otherwise")

}

我们可以用任意表达式（而不只是常量）作为分支条件

when (x) {

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

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

}

我们也可以检测一个值在（ in ）或者不在（ !in ）一个区间或者集合中：控制流

when (x) {

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

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

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

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

}

另一种可能性是检测一个值是（ is ）或者不是（ !is ）一个特定类型的值。注意： 由于智能转换，你可以访问该类型的方法与属性而无需任何额外的检测。

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

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

else -> false

}

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

when {

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

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

else -> print("x is funny")

}

#### 3 for 循环

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

循环体可以是一个代码块。

for (item: Int in ints) {

// ……

}

如上所述， for 可以循环遍历任何提供了迭代器的对象。即：

有一个成员函数或者扩展函数 iterator() ，它的返回类型

有一个成员函数或者扩展函数 next() ，并且

有一个成员函数或者扩展函数 hasNext() 返回 Boolean 。

这三个函数都需要标记为 operator 。

如需在数字区间上迭代，请使用区间表达式:

fun main(args: Array<String>) {

//sampleStart

for (i in 1..3) {

println(i)

}

for (i in 6 downTo 0 step 2) {

println(i)

}

//sampleEnd

}

对区间或者数组的 for 循环会被编译为并不创建迭代器的基于索引的循环。

如果你想要通过索引遍历一个数组或者一个 list，你可以这么做：

fun main(args: Array<String>) {

val array = arrayOf("a", "b", "c")

//sampleStart

for (i in array.indices) {

println(array[i])

}

//sampleEnd

}

或者你可以用库函数 withIndex ：

#### 4 while 循环

while循环，do while,break,continue这几个与java中无区别

### 4 返回与跳转

Kotlin 有三种结构化跳转表达式：

return 。默认从最直接包围它的函数或者匿名函数返回。

break 。终止最直接包围它的循环。

continue 。继续下一次最直接包围它的循环。

所有这些表达式都可以用作更大表达式的一部分：

val s = person.name ?: return

这些表达式的类型是 Nothing 类型。

#### 1 Break 与 Continue 标签

在 Kotlin 中任何表达式都可以用标签（ label ）来标记。 标签的格式为标识符后跟 @ 符号，例如： abc@ 、 fooBar@ 都是有效的标签（参见语法）。 要为一个表达式加标签，我们只要在其前加标签即可。

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

// ……

}

现在，我们可以用标签限制 break 或者 continue ：

fun testBreakLabel(){  
 loop@for(i in 1..100){  
 for (j in 1..100){  
 *println*(" i = $i j = $j")  
 if (i== 2 && j == 2){  
// //默认退出的是内层循环,最直接包围它的循环  
// break  
 //退出到标签循环  
 break@loop  
 }  
 }  
 }  
}

标签限制的 break 跳转到刚好位于该标签指定的循环后面的执行点。 continue

继续标签指定的循环的下一次迭代。

#### 2 return标签

Kotlin 有函数字面量、局部函数和对象表达式。因此 Kotlin 的函数可以被嵌套。 标签限制的 return 允许我们从外层函数返回。 最重要的一个用途就是从 lambda表达式中返回。回想一下我们这么写的时候：

fun testReturnLabel(){

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

if (it == 3){

//直接return testReturnLabel的调用

return

}

println("i = $it")

}

println("testReturnLabel end")

}

程序输出

i = 1

i = 2

这个 return 表达式从最直接包围它的函数即 foo 中返回。 （注意，这种非局部的返回只支持传给内联函数的 lambda 表达式。） 如果我们需要从 lambda 表达式中返回，我们必须给它加标签并用以限制 return 。

fun testReturnLabel(){

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

if (it == 3){

//直接return testReturnLabel的调用

return@it

}

println("i = $it")

}

println("testReturnLabel end")

}

一般比标签与lambda 函数同名

fun testReturnLabel(){

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

if (it == 3){

//直接return testReturnLabel的调用

//return

return@forEach

}

println("i = $it")

}

println("testReturnLabel end")

}

输出结果

i = 1

i = 2

i = 4

i = 5

testReturnLabel end

如果对于匿名函数，匿名函数内部的 return 语句将从该匿名函数自身返回

fun testReturnLabel2(){

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

if (value == 3){

//直接return testReturnLabel的调用

return

}

println("i = $value")

})

println("testReturnLabel2 end")

}

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

fun testReturnLabel3(){

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

if (it == 3){

//return lint 类似于在lint处continue了

return@lint

}

println("i = $it")

}}

println("testReturnLabel3 end")

}

输出

i = 1

i = 2

testReturnLabel3 end

## 三 kotlin类型系统

### 1 可空性

#### 1 可空类型

kotlin中引入了**Type?**来表示Type类型是可能为null,这样从编译上避免了很多NPE的发生。

Type? = Type or null

问号?可以加载任何类型后面，表示这个类型的变量可能为null

没有问号?表示这种类型的变量不为null。

对可空类型的操作，kotlin中考虑了NPE的情况，做了一些限制

//可空类型

var name:String? = null

//非空类型

var personName:String = "娃哈哈"

对可空类型的变量调用它的方法必须做null判断，如果左了判断编译器会记住判断结果

var length:Int = if (name != null) name.length else 0

可空类型变量不能直接赋值给非空类型变量

// error

personName = name;

可空类型的值不能传递给非空类型参数

fun getLength(s:String):Int = s.length

//error

getLength(name)

但是可以把非空类型赋值给可空类型变量

//ok

name = personName;

完整实例如下：

fun main(args: Array<String>) {

//可空类型

var name:String? = null

//非空类型

var personName:String = "娃哈哈"

var length:Int = if (name != null) name.length else 0

//error

getLength(name)

//ok

//name = personName;

// error

personName = name;

}

fun getLength(s:String):Int = s.length

#### 2 安全调用运算符”?.”

安全调用运算符允许把一次null检查和一次方法或者属性调用合并成一个操作。

Xxx?.xxx = if(Xxx != null) Xxx.xxx else null

注意调用结果是可空类型的

我们来改写前面的例子

var len:Int? = name?.length

对于方法的调用也类似，就不举例子了。

#### 3 Elvis运算符 “?:”

与”?.”运算符类似,”?:”接受两个运算数，如果第一个不为null，表达式就是第一个值，否则取第二个值

Xxx?:xxx = if(Xxx != null) Xxx else xxx

注意”?:”经常与”?.”运算符一起调用，因为我们经常有这样一种调用:

如果Xxx不为null，就调用Xxx.A，否则返回一个默认值

实例如下：

var length2:Int = name?.length?:0

单独使用”?:”一般用于可空类型赋值给非空类型

//可空类型

var name:String? = null

//非空类型

var personName:String = "娃哈哈"

personName = name?:""

#### 4 安全转换符”as?”

as用来进行类型转换，在java中如果类型不匹配将会抛异常，在kotlin中为了减少这种异常抛出，如果认为类型不匹配使用as?可以直接返回null。从而减少了异常的发生

xxx as? Type = if(xxx is Type) xxx as Type else null

as?经常和?:一起使用，见下面的例子

class People(val name:String,val age:Int){

override fun equals(other: Any?): Boolean {

var another = other as? People?:return false

return another.name == name && another.age == age

}

override fun hashCode(): Int {

return age.hashCode()\*31 + name.hashCode()

}

}

#### 5 非空断言”!!”

非空断言主要用作将任何类型转换为非空类型，如果被转换的值是null,将会抛出NPE异常。

xxx!! = if(xxx != null) xxx else throw NPE

实例如下：

fun ignoreNulls(s:String?){

var str = s!!

println(str.length)

}

如果s为null，将会在var str = s!!抛出NPE异常

## 四 函数

## 五 类与对象

### 1 类与继承

kotlin中定义类如下：

class ClassName [constructor][(params:Type,…)] [{ ... }]

[]表示可以省略

类声明由类名、类头（指定其类型参数、主构造函数等）以及由花括号包围的类体构成。**类头与类体**都是可选的； 如果一个类没有类体，可以省略花括号

class Empty

在 Kotlin 中的一个类可以有一个主构造函数以及一个或多个次构造函数。主构造函数是类头的一部分：它跟在类名（与可选的类型参数）后

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

如果主构造函数没有**任何注解或者可见性修饰符**，可以省略这个 constructor关键字。

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

主构造函数不能包含任何的代码。初始化的代码可以放到以 init 关键字作为前缀的初始化块（initializer blocks）中。

在实例初始化期间，初始化块按照它们出现在类体中的顺序执行，与属性初始化器交织在一起：

class User(var name:String,var age:Int){

init {

println("${this.javaClass.simpleName} init:$name $age")

}

}

请注意，主构造的参数可以在初始化块中使用。它们也可以在类体内声明的属性初始化器中使用：

class Customer(name: String) {

val customerKey = name.toUpperCase()

}

事实上，声明属性以及从主构造函数初始化属性，Kotlin 有简洁的语法：

class Person(val firstName: String, val lastName: String, var age: Int) { …… }

如果构造函数有注解或可见性修饰符，这个 constructor 关键字是必需的，并且这些修饰符在它前面：

class Customer public @SuppressWarnings constructor() {

}

次构造函数

3 构造函数

4 创建类实例

var user:User = User("heihei",20)

var user2:User2 = User2("美女",18)

var user1:User1 = User1()

## 六 Lambda编程

### 1 Lambda表达式定义

kotlin中lambda表达式定义如下：

{param1:Type,param2:Type… -> ….}

例子

{x:Int,y:String -> y.length + x}

就是一个lambda表达式

在kotlin中如果要直接运行lambda，可以使用run{}来运行

run{45}

### 2 kotlin中lambda表达式的约定

kotlin中对于lambda有很多简化的约定，我们来看一个例子

val people = listOf(User("张三",14),User("李四",16))

println("年纪最大:" + people.maxBy { it.age })

这里的lambda和我们看到的定义不太一样，如果完全不简写，那么完整的例子如下：

println("年纪最大:" + people.maxBy ({user:User -> user.age }))

那么最终为什么会简化成people.maxBy { it.age })呢，我们来看kotlin中lambda的约定

1 如果lambda表达式是函数调用的最后一个实参，它可以放在括号外面

如上所示，那么可以简化为：

println("年纪最大:" + people.maxBy(){user:User -> user.age })

我们看到，还可以简化

2 当lambda是函数唯一的实参时，可以去掉函数调用的括号

println("年纪最大:" + people.maxBy{user:User -> user.age })

3 如果lambda的参数的类型可以推导，那么可以省略参数的类型

println("年纪最大:" + people.maxBy{user -> user.age })

这里对people集合操作，其元素数据类型是User，因此可以省略不写

4 对于lambda中一个参数时，可以使用默认参数名称it来代替命名参数，并且lambda的参数列表可以简化，省略参数列表和->

println("年纪最大:" + people.maxBy { it.age })

到这里，我们可以看到的lambda形式就是在kotlin中经常出现的形式

上面介绍的是函数只有一个参数的情况，我们来看多个参数的情况

fun lambdaTest1(a:Int,b: (String) -> String){

println("$a + ${b(a.toString())}")

}

fun lambdaTest2(b: (String) -> String,a:Int){

println("$a + ${b(a.toString())}")

}

fun lambdaTest3(b: (arg1:String,arg2:Int) -> String,a:Int){

println("$a + ${b(a.toString(),100)}")

}

使用如下：

lambdaTest1(3){"hello:" + it}

lambdaTest2({"hello:" + it},5)

lambdaTest3({arg1, arg2 -> "hello:$arg1 + $arg2"},58)

输出结果如下：

3 + hello:3

5 + hello:5

58 + hello:58 + 100

### 3 Lambda在kotlin中的用途

1 作为单独的功能块单独使用

如下，定义两个数的和这样一个功能块

val sum = {a:Int,b:Int -> a+b}

println(sum(2,9))

作为单独的功能块，

2 作为函数参数的代码块

如上所示，例如lambdaTest1,LambdaTest2等

3 集合中使用lambda

kotlin中集合中的很多操作都支持lambda,例如maxBy,map,fliter等

4 高阶函数

### 4 Kotlin中集合常用函数式API

在kotlin中有一些常用的对集合进行操作的函数式API，掌握他们可以大大提高编程效率，这些API与java8中的

Stream API类似。下面主要讲解以下几大类

#### 1 filter 和 map

filter主要用作过滤器(对集合元素)，它的返回值也是一个集合，定义如下：

public inline fun <T> Iterable<T>.filter(predicate: (T) -> Boolean): List<T> {

return filterTo(ArrayList<T>(), predicate)

}

可以看到，返回的也是一个集合，并且类型与原类型相同，用法如下：

map主要对集合的元素做映射，可以映射成不同的类型，例如Int到String，定义如下

public inline fun <T, R> Iterable<T>.map(transform: (T) -> R): List<R> {

return mapTo(ArrayList<R>(collectionSizeOrDefault(10)), transform)

}

可以看到，返回值是映射成的新类型的集合

#### 2 all any count find

以上4个都是对集合的所有元素检查是否符合某个条件(或者它的变种，是否存在符合的元素等)

all检查集合中是否所有元素都符合某个条件

public inline fun <T> Iterable<T>.all(predicate: (T) -> Boolean): Boolean {

if (this is Collection && isEmpty()) return true

for (element in this) if (!predicate(element)) return false

return true

}

可以看到返回值是Boolean类型，都符合返回为true,有一个不符合为true

any检测集合中是否至少存在一个元素满足条件，如果是就返回true

public inline fun <T> Iterable<T>.any(predicate: (T) -> Boolean): Boolean {

if (this is Collection && isEmpty()) return false

for (element in this) if (predicate(element)) return true

return false

}

count 检查集合中有多少个元素满足该条件，返回值是Int

public inline fun <T> Iterable<T>.count(predicate: (T) -> Boolean): Int {

if (this is Collection && isEmpty()) return 0

var count = 0

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

return count

}

find找到集合中第一个满足条件的元素，返回值是集合元素类型，可能为null

@kotlin.internal.InlineOnly

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

return firstOrNull(predicate)

}

#### 3 groupBy

groupBy主要用作将集合按照给定的条件进行分组，即将集合转为map

public inline fun <T, K> Iterable<T>.groupBy(keySelector: (T) -> K): Map<K, List<T>> {

return groupByTo(LinkedHashMap<K, MutableList<T>>(), keySelector)

}

可以看到，返回的是一个map，key类型就是lambda的返回值类型，value是集合元素的List

#### 4 flatMap

flatMap先对集合中的某个元素映射成集合，再将所有映射成的集合做一个元素平铺，即做一个list

public inline fun <T, R> Iterable<T>.flatMap(transform: (T) -> Iterable<R>): List<R> {

return flatMapTo(ArrayList<R>(), transform)

}

返回集合元素映射的集合的list

## 七 高阶函数

### 1 高阶函数定义

在kotlin中任何以lambda或者函数引用为参数的函数，或者函数的返回值是lambda或者函数引用，就是高阶函数

### 2 函数类型

在kotlin中，函数都是有类型的。例如如下这个函数

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

return x+y

}

它的类型就是{(Int,Int) -> Int}，注意和lambda表达式的类型很像，只是多了一对括号。

因此学习高阶函数，我们首先要弄明白函数的类型

### 3 参数是lambda的高阶函数

fun main(args: Array<String>) {

//传递lambda表达式

highFunction1{

a,b -> a+b

}

highFunction1{

a,b -> a\*b

}

//直接传递函数

highFunction1(::multiply)

}

/\*\*

\* 操作两个数

\*/

fun highFunction1(operation:(Int,Int)->Int){

val result = operation(20,30)

println("result: $result")

}

/\*\*

\* 两个数相加

\*/

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

return x+y

}

/\*\*

\* 两个数相乘

\*/

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

return x \* y

}

可以看到，高阶函数可以直接传递lambda或者直接传递函数的引用

### 4 返回值是函数的高阶函数

/\*\*

\* 定义一个返回值是函数的高阶函数

\* @param name 入场

\* @return 返回一个函数或者lambda

\*/

fun highFunction2(name:String):(Int) -> Int{

if (name == "A"){

return ::returnFun

}

//返回lambda

return {a -> a + 10}

}

/\*\*

\* 作为高阶函数的返回函数

\*/

fun returnFun(a:Int):Int{

return a \* 100

}

使用如下：

//使用高阶函数

val res = highFunction2("A")

println(res(20)) //打印2000

val res2 = highFunction2("B")

println(res2(20)) //打印30