目录

[Kotlin的数据类型 1](#_Toc484618474)

[基本数据类型 1](#_Toc484618475)

[数组 2](#_Toc484618476)

[字符串 2](#_Toc484618477)

[集合 3](#_Toc484618478)

[泛型 3](#_Toc484618479)

[声明处型变 4](#_Toc484618480)

[使用处型变 4](#_Toc484618481)

[可见度修饰符 5](#_Toc484618482)

[反射 5](#_Toc484618483)

[声明属性 5](#_Toc484618486)

[可空类型和不可空类型 6](#_Toc484618487)

[局部变量 6](#_Toc484618488)

[定义函数 7](#_Toc484618490)

[匿名函数和函数对象 7](#_Toc484618491)

[高阶函数 8](#_Toc484618492)

[局部函数 8](#_Toc484618493)

[内联函数 9](#_Toc484618494)

[默认参数 9](#_Toc484618495)

[中缀函数 10](#_Toc484618496)

[扩展函数 10](#_Toc484618497)

[泛型函数 12](#_Toc484618498)

[可变参数 12](#_Toc484618499)

[if表达式 12](#_Toc484618500)

[when表达式 13](#_Toc484618501)

[类型检查及自动类型转换 13](#_Toc484618502)

[强制类型转换 14](#_Toc484618503)

[循环 14](#_Toc484618504)

[结构跳跃表达式 14](#_Toc484618505)

[区间表达式 15](#_Toc484618506)

[解构声明 15](#_Toc484618507)

[包Package 15](#_Toc484618508)

[类class 16](#_Toc484618509)

[次级构造 16](#_Toc484618510)

[类的继承 16](#_Toc484618512)

[接口interface 18](#_Toc484618516)

[对象表达式(匿名内部类) 20](#_Toc484618517)

[伴生对象 20](#_Toc484618518)

[对象声明 21](#_Toc484618519)

[异常 21](#_Toc484618520)

Kotlin官方英文文档<http://kotlinlang.org/docs/reference/>

## Kotlin的数据类型

### 基本数据类型

Kotlin使用几个特殊的类表示基本数据类型,将java中的基本数据类型首字母大写即为这些类.虽然使用特殊的类表示,不过物理存储时还是作为JVM原生类型存储,当作为可空引用或泛型时会把基本数据类型装箱.

在kotlin中有两种相等性: 使用 === 和 !== 判断两个引用是否指向同一个对象

使用 == 和 != 判断其结构是否相等, == 会调用equals进行比较.

#### 数字和字符

十六进制为0x 开头;二进制为0b开头,Kotlin中字符不再是数字,无法直接当成数字使用.

可以在数字中可以添加下划线提高可读性: var a = 1\_000\_000

Kotlin对于数字较小的类型不能隐式转换为较大的类型,可以显式转换来拓宽数字,较小类型不是较大类型的子类型,不同数据类型无法使用 == 来进行比较,不同类型进行运行会拓展为更大的类型.

var a = 1  
var b: Long  
//b = a 直接进行赋值会报错  
b = a.toLong() //需要进行显式转换

//println(b == a) 不同数据类型无法使用 == 进行比较  
*println*(b.equals(a)) //但是使用equals进行结果为false

每个数字类型和字符都可以使用方法转换为其他类型:

**toByte(): Byte toShort(): Short toInt(): Int**

**toLong(): Long toFloat(): Float toDouble(): Double**

**toChar(): Char**

数字装箱时若装箱的值超过-128-127时,不再保留同一性,不过会保留相等性

val a = 1000  
val b: Int? = a  
val c: Int? = a  
*println*(b === c) //false  
*println*(b == c) //true

**位运算**: 在java中的位运算符无法在kotlin中识别,Int和Long的位运算需要使用如下方法,都是中辍方法, . 和后面参数的括号可以省略: 比如 1 and 2,

**shl(bits)**  有符号左移 << **shr(bits)**  有符号右移 >>

**ushr(bits)**  无符号右移 >>> **and(bits)**  与 &

**or(bits)**  或 | **xor(bits)**  异或 ^

**inv()**  非 !

#### 布尔

用**Boolean**表示,Kotlin中的布尔运算只保留了: 短路或 || 、短路与 && 、非 !

### 数组

用**Array<T>**类表示,操作还是使用 [索引] 进行操作

可以使用库函数提供的方法创建数组:

**arrayof(各元素)**  创建指定元素的数组

var a = arrayOf("1", 2, '2') //这个数组是object类型的

**arrayOfNulls(size)**  创建一个指定大小、元素都为空的数组

var a2 = *arrayOfNulls*<Int>(1);

可以使用Array类的构造方法:

**Array(size, init函数)** 创建一个指定大小并按照init函数为每个元素赋初始值的数组.

var arr = Array(5, fun(i: Int): Int { //这是匿名函数的形式  
 return i \* i  
})

var arr2 = Array(5, **{** i **->** i \* i **}**) //lambda表达式的形式

**原生类型数组** : 使用Array<T>类存储基本数据类型时,会将基本数据类型进行装箱.每个基本数据类型都有无装箱开销的专门类来表示原生类型数组,这些类的使用方法和Array完全相同.比如:**IntArray**

### 字符串

还是用String表示,但每个元素可以使用 [索引] 访问,可以使用for循环直接迭代字符串.

Kotlin的字符串分为转义字符串和原始字符串:

**转义字符串**和java字符串相同,使用 " 括起来,其中可以使用反斜杠对字符进行转义.

**原始字符串**使用 """ 括起来,其中不用转义,可以包含任意字符,会保留 """ 中的所有内容

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

原生字符串可以使用trimMargin()方法去除其中的所有空格,默认使用 | 作为每一行的开始,会将|前的所有空格和|移除,可以传入参数将其替换.比如 trimMargin(">")

#### 字符串模板表达式

在字符串中可以使用模板表达式加入一段代码,会将代码进行求值后合并到字符串.转义字符串和原生字符串都支持模板表达式

模板表达式使用 $ (美元符号) 开头:

当只有 $ 时,其后最多跟着一个属性,会将属性的值合并到字符串 var h = "a = $a"

若要包含代码块,使用 ${}会将花括号中代码运行后合并到字符串 var h = "a = ${a + 2}

## 集合

在kotlin中,集合的继承结构和父类的功能发生了改变,分为可变集合和不可变集合.集合中添加了更多的方便的方法.

List<out T> 提供只读操作,继承自 Collection<T> 进而继承自 Iterable<T>,改变 list 的方法由 MutableList<T> 加入的.

set和map也有相似结构: Set<out T>/MutableSet<T> Map<K, out V>/MutableMap<K, V>.

创建list或set用标准库的方法listOf()、mutableListOf()、setOf()、mutableSetOf()等

创建map在非性能关键代码中可以使用 mapof(a to b,...)

## 泛型

java中的泛型设计为不可变不具有继承关系,比如List<String>不是List<Object>子类型,因为如果具有继承关系,当List<String>转为List<Object>后,向其中添加Integer再从之前的List<String>中读取存入的Integer即会产生错误.

不能转换类型避免了这种错误的发生,但是因为泛型不具有继承关系,有些理所当然的操作变为不可用,比如将一个泛型为子类的集合作为参数传递给参数为泛型为父类的集合的一个方法时是不可用的,比如addAll方法如果参数为addAll(Collection<E> c),则List<Object>的对象调用时不能addAll(List<String>).

为了解决泛型不可变引起的这些问题java设计了类型通配符 **?** 可以指定上界和下界: ? extends T 和 ? super T,当只有 ? 时相当于 ? extends Object,**他们只能作为对象接收方引用以限定范围内的类型为泛型的集合对象.**

**? extends T** 表示上界,使用 ? extends T作为泛型的集合可以指向泛型为T或T的子类的集合对象,因为指向的集合是未知的,只要是T或T的子类都可以,如果他指向的是一个子类却添加了父类的对象或父类对象的其他子类对象就会产生错误,即无法得知添加的类型是否符合指向的类型,因此以该通配符为泛型的集合被设计为不能进行添加操作,只能取出T类型的实例,称为生产者.

**? super T** 表示下界,使用 ? super T作为泛型的集合可以指向泛型为T或T的父类的集合对象,因为指向的集合也是未知的,只要是T或T的父类都可以,如果他指向的是一个子类却添加了父类的对象也会产生错误,为了保证指向的集合不会存储到错误的类型,以该通配符为泛型的集合只能添加T的实例即最下层的对象的实例,但是在取出时无法知道指向的集合中存储的到底是哪个类型,所以读取时只能读取为最上层的超类Object类型的实例,被称为消费者.

Kotlin中,普通的泛型与java没有什么区别,类型通配符通过Declaration-site variance(声明处型变)和Type projections(类型投影,使用处型变)实现.

### 声明处型变

在泛型第一处声明的地方,使用 out 关键字修饰一个泛型,可以保证该类型只会被从集合中生产出而不能被添加到集合中. 使用 in 关键字修饰一个泛型,可以保证该类型只会被消费而不会被生产

**out相当于java的? extends T,可以指向泛型为T或T的子类的该类的对象**

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

fun demo(strs: Source<String>) {  
 val objects: Source<Any> = strs  
}

**in 相当于java的? super T,可以指向泛型为T或T的父类的该类的对象**

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

fun demo(x: Comparable<Number>) {  
 val y: Comparable<Double> = x  
}

### 使用处型变

在第一处声明的地方使用型变会让该类仅限于生产或消费T,如果需要只是在某处使用的位置保证该类只会被生产或消费就可以在使用处添加型变.

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

当不添加out修饰符时,此时from只能接收泛型为Any的数组,因为如果可以接收到其他泛型的数组无法保证不会被存入错误的类型,此时添加out修饰,可以保证from只会生产其中存储的类型,可以接收泛型为Any和其子类的所有数组.

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

这里dest可以传入泛型为CharSequence或object的数组,向其中存入String类型是安全的.

#### 一般约束

除了out进行约束,在尖括号内的泛型后可以使用 **:** 为泛型设置上限,相当于java的 ? extends T,在<>内只可指定一个上限,使用where子句指定多个上限,多个上限时只有一个上限可以为类,其他为上限为接口,没指定上限时默认为Any?

fun <T : CharSequence> method(list: MutableList<T>) {}

fun <T> method2(list: MutableList<String>) whereT:MyClass, T : MyInterface {}

## 可见度修饰符

private: 在包内使用时只会在包含该声明的文件内部可见,在类中使用时只对该类可见

protected: 只在子类中可见

internal: 在包内使用时在同一个模块中到处可见,在类中使用时模块中可以获取到类的就能获取

public: 在包内使用时任何地方都可见,在类中使用时可以获取到类的都可以获取

当没有使用可见性修饰符时默认为public,在子类中进行覆盖时若不另外声明修饰符与父类中相同.

## 反射

### 类的反射

kotlin中类的运行时的引用为KClass,要获取一个类或对象的运行时引用使用字面类语法 **:: class**

比如Juser是一个普通类,jj是它的实例对象

获取KClass实例:

var aaa: KClass<JUser> = JUser::class  
var bbb: KClass<out JUser> = jj::class

Class对象在kotlin也可以使用

要获取到Class对象,可以使用KClass的.java属性或使用实例的.javaClass属性

var ccc: Class<JUser> = JUser::class.java  
var ddd: Class<JUser> = jj.*javaClass*

Class实例具有.kotlin属性为KClass对象,KClass具有.java属性为Class对象.

### 函数反射

:: 运算符用于反射方法 **:: 函数名**, 当方法为某个类的成员方法时使用 **类名 :: 函数名**

反射到的函数的数据类型表示为 **类名.(参数类型) -> 返回值类型** 没有类名时省略类名,没有参数时保留(),多个参数类型使用逗号分开,没有返回值类型是返回值类型为Unit. 意思是这个函数接收n个参数类型的参数,返回返回值类型的值. 反射到的函数对象可以像函数一样直接调用.

val toCA: String.() -> CharArray = String::toCharArray toCA("123")

## 声明属性

使用 val声明常量,使用var声明变量.声明的属性默认为不可空属性,若要声明可空属性需要在数据类型后加上? var a : String?

### 可空类型和不可空类型

Kotlin的类型系统对空指针做了特殊的处理,对于一个可空属性在使用时编译器会强行报错(即使看起来这个属性不会是空).对于不可空类型,在使用时需要进行处理.kotlin提供了一些特殊的操作符进行可空类型的操作,比如非空判断可以通过编译: var b = if (a != null) a.length else -1

#### 安全操作符和Elvis操作符

在对一个可空属性进行调用时 使用安全操作符 **?.** ,当这个属性不为空时会正常调用,否则返回null.

安全操作符可以在连续调用中进行使用,直到遇到一个属性为空时就返回null,比如: a?.b?.c?.d

需要只对可空类型的非空值进行操作,可以使用**let运算符**:

val list: List<String?> = *listOf*("1", "2", null, "4")  
for (item in list) {  
 item?.*let* **{** *println*(item) **} //当item为空时,let中的内容不会执行**}

要在可空类型为空的时候进行一些其他操作,可以使用**Elvis操作符**,它相当于if else的效果:

*println*(item ?: "null")

//相当于 println(if (item != null) item else "null")

#### 抛出空指针异常

要像以前一样抛出空指针异常时,可以使用 **!!** 运算符,调用时使用 **!!.**

*println*(item!!)  
println(item!!.*toInt*())

### 局部变量

具有初始值直接进行赋值,变量的类型会自动推断出,可以省略;若没有初始值需要声明类型.

val a = 1 //直接进行赋值,val为只读变量  
var b: Int //声明指定类型的变量

b = 2

### 属性

**关于属性和字段**:属性和字段这个概念是在c#中比较明显的一个内容,在java中并没有表现的比较明显,在java中,可以将成员变量为字段,get和set方法为属性.在kotlin中官方称为属性,没有字段,这里把属性理解为java中的字段(成员变量)更容易理解,但是不完全是java中的成员变量.

可以在类中或包中声明属性,包中声明的属性使用全包名引用,类中声明的属性通过类的实例对象引用.

var修饰的属性默认具有get和set方法,可以自定义get和set方法.赋值会自动调用set方法,调用属性会自动调用其get方法. val修饰的属性只具有get方法.

将常量使用const修饰可以作为**编译时常数**,编译时常数只能是基本数据类型和String

将属性使用lateinit修饰这个属性为**延期初始化属性**,不必在声明时就进行初始化赋值.

**完整的声明一个属性**为: var 变量名 : 类型 = 初始化

get() =

set(value) { //这里一般使用value,也可以换其他名称

}

在声明属性时get和set方法是可选的,如果可以从初始化和get推断出类型,类型也是可选的.

set和get方法前可以添加注解不必实现其主体,也可以添加修饰符,get的修饰符必须与属性的修饰符相同,set的修饰符必须和属性的修饰符相同或权限更低.

访问器可自定义,在访问器中直接使用属性名进行引用会出现StackOverflowError(因为引用属性名称去会自动调用自己的get set方法,结果就是无限循环),这时就需要一个中间人作为支持字段提供在访问器中的访问,kotlin提供了标识符field作为支持字段,field只能在属性访问器中进行使用.

如果属性使用至少一个访问器的默认实现,或者自定义访问器通过标识符引用它,则将为field属性生成支持字段,其他情况这个属性不具有支持字段,**不具有支持字段的属性无法直接初始化**.

var x : String = "1"   
 get() = **field** //实际上这就是get和set的默认实现,对支持字段进行操作  
set(value) {  
 **field** = value  
 }

## 定义函数

1. 使用关键字 fun 声明一个函数,函数可以声明在包下,类中,函数中.

fun 方法名(变量名 : 变量类型,…) : 返回类型 {

return 返回值

}

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

}

当无返回值时可以将返回类型省略.( 也可以将返回类型写为 Unit,Unit类型表示无意义)

2. **函数的函数体可以直接为一个表达式**,会使用表达式的结果自动推断出函数的返回类型:

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

### 匿名函数和函数对象

kotlin中很多地方需要将方法作作为参数传入,可以使用**匿名函数**和**通过反射获取方法的引用**.

函数的类型的表现形式为: **类名.(参数类型) -> 返回值类型.**

匿名函数只是去掉了函数名, 可以使用lambda表达式简单的写出来:

var arr = Array(5, **{** i **->** i \* i **}**) //相当于:

var arr = Array(5, fun(i: Int): Int {

return i \* i  
})

当使用完整的匿名函数表达时可以不声明其类型,当使用lambda表达式时,必须声明其类型.

val d = fun(a: Int) = a > 0

var e: (Int) -> Boolean = **{ it** > 0 **}**

使用反射中的 **类名::运算符** 获取方法的引用,当没有类名时直接省略类名.

fun method(i: Int): Int {  
 return i \* i  
}

var x = ::method  
var arr2 = Array(5, x)

获取函数的引用之后可以直接使用引用对函数进行调用 x(2)

#### lambda表达式

在kotlin中,lambda表达式主要用于表示匿名函数,一个lambda表达式表达在花括号{}内,如果这个函数有参数, 在 -> 之前声明参数,省略参数的类型,函数体在 -> 后.

有一个公约,当函数的参数为一个时,这个参数默认名称为**it** ,可以省略其参数的声明和 -> ,直接在{}中书写函数体.

### 高阶函数

在kotlin中函数也是对象,可以将函数作为对象作为参数和返回值,将函数作为参数或返回函数的函数就是高阶函数. 比如Array的构造函数就是一个高阶函数(size: Int, init: (Int) -> T)

如果一个高阶函数的最后一个参数是需要传入一个函数,这个参数可以在括号外指定.

var array = Array(10) **{** fun(it: Int): String = it.toString() **}**  //匿名函数形式  
var array2 = Array(10) **{** it.toString() **}** //lambda表达式形式

### 局部函数

局部函数是在函数的内部声明的函数. **局部函数 和lambda表达式和 匿名函数 以及** [**对象表达式**](#_对象表达式(匿名内部类)) **都可以访问外部函数的局部变量,并且与java中不同的是可以修改其值,称为闭包.**

fun method(a: Int) {  
 val b = 1  
 fun method() {  
 *println*(a + b)  
 }  
 method()  
}

### 内联函数

使用高阶函数会带来一些运行时间效率的损失,因为每个函数作为一个对象,都会捕获一个闭包,即在函数体中那些访问的变量,对于函数对象和类的内存分配以及虚拟调用会引入运行时间开销.

通过内联lambda表达式可以消除这种开销,通过使用 inline 修饰函数可以让函数作为一个内联函数,如果参数中有多个函数,希望一些函数不被内联,可以使用 noinline修饰这个参数,inline还可用于属性的get和set方法的修饰.

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

**非本地返回**

当使用lambda时,如果需要从lambda中返回,需要添加一个标签([结构跳跃表达式](#_结构跳跃表达式)),在lambda中是不能让外围函数返回的,但是当内联后,return也会被内联,因此可以在lambda中返回外围函数.

不过有的内联函数不会在函数体中调用被传入的lambda,是从其他的上下文中进行调用,比如将传入的函数对象放入对象表达式或局部函数,然后通过对象表达式的对象或通过调用局部函数间接进行了调用,这种情况下,本地返回也会被禁止,在这种情况若还需要进行非本地返回需要将参数使用crossinline进行修饰.

比如在这个函数中,传入的body函数被放在对象表达式中.

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

//...  
}

#### 泛型变量

普通的泛型声明这个泛型只能作为参数类型使用,在方法体中无法使用,在内联函数中可以使用 reified 将泛型修饰,这个泛型就像普通的类一样,可以在函数内部正常访问.普通函数无法使用泛型变量

inline fun <reified T> ArrayList<T>.method(map: ArrayList<Any>) {  
 map.*filterIsInstance*<T>().*forEach* **{** add(**it**) **}**}

### 默认参数

函数的参数可以进行初始化赋值作为默认参数,默认值在类型后使用 = 进行赋值

当方法被重写时,其默认参数不变,但是在重写的方法中默认参数必须被省略

当方法所需的参数具有多个时,传入的参数使用 参数名 = 值,可以只传入需要传入的参数.

调用Java函数时不能使用命名参数语法,因为Java字节码不总是保留函数参数的名称.

open class A {  
 open fun print(a: Int = 1, b: Int) {  
 *println*("$a , $b")  
 }  
}  
  
class B : A() {  
 override fun print(a: Int, b: Int) {  
 super.print(a, b)  
 }  
}

调用:

val a = A()  
a.print(b = 2)  
  
val b = B()  
b.print(a = 3,b = 2)

#### 类型别名

当一个类型的名称太长时,可以使用typealias关键字为一个类型声明一个别名.类型别名只能声明在包结构下,不能声明在类或方法中.

比如上面的函数对象的类型: typealias Predicate<T> = (T) -> Boolean

val p: Predicate<Int> = **{ it** > 0 **}**

*println*(p(42)) // true  
*println*(*listOf*(1, -2).*filter*(p)) // prints "[1]"

### 中缀函数

使用**infix** 修饰的函数在调用时可以省略 . 和 () ,比如位运算的函数都是中缀函数

**infix** **fun** Int.shl(x: Int): Int { ... }

调用: var a = 1 shl 2 //相当于 1.shl(2)

### 扩展函数

kotlin提供了扩展函数功能,在不继承或使用设计模式的情况下扩充一个类的方法,然后使用该类的实例调用该方法,在扩展方法中,this代表调用这个方法的对象 fun 类名/接口名.方法名(参数) {}

比如在kotlin中String类拥有了toInt等系列方法就是通过扩展函数实现:

public fun String.toInt(): Int = Integer.parseInt(this)

扩展函数通过静态调度,被调用的扩展函数由调用该函数的表达式的类型决定.

若类的扩展函数与类的成员方法重名并具有相同参数,则优先选择成员方法,扩展函数可以重载成员方法.

#### 扩展属性

和方法一样,也可以添加扩展属性,但是由于并不是真正将属性插入到类中,扩展属性因此无法具有支持字段,扩展属性不支持扩展初始化器,只能通过明确的get和set方法定义,其他的特性与一般属性一致.

var 类名/方法名.属性名:类型

get() = ...

set(value) {

...

}

比如为集合添加一个最后的索引的只读属性:

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

扩展函数和扩展属性一般直接在包下定义,在其他包中使用时需要进行导入.

#### 在一个类中为另一个类扩展

在这种扩展中可以在没有限定符(this)的情况下访问被扩展的类的成员,在方法中还可以访问自己的方法,添加扩展的类称作调度接收者,被扩展的类称作扩展接收者.

但是这种扩展在调度接收者之外后扩展接收者就无法调用到,所以需要提供一个调用方法.

当两方具有完全一样的方法时,优先选择扩展接收者中的方法,如果想要调用到调度接收者中的该方法,可以使用 @this类名 进行调用.

class D {  
 fun dMethod() {}  
}  
  
class C {  
 fun cMethod() {}  
  
 fun D.exMethod() {  
 dMethod()  
 cMethod()  
 toString()  
 this@C.toString()  
 }  
  
 fun caller(d: D) { //需要提供一个调用方法  
 d.*exMethod*()  
 }  
}

### 泛型函数

和java一样在函数上也可以使用泛型代替一些类型,泛型添加在方法名前,在使用时要在方法名后方指定泛型类型,要是可以从推导出泛型类型,可以省略.

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

扩展函数也可以使用泛型,相当于对所有类型扩展了该方法:

fun <T> T.basicToString(): String {  
 ....

}

### 可变参数

kotlin中使用**vararg**声明一个可变参数,一般声明在方法中参数的最后一个,**可变参数以数组的形式传递,**可以在for循环中使用in获取可变参数中的每一个值.若已经有一个数组想要将其内容传递给函数,可以使用**扩展运算符 \***

fun <T> toList(vararg items: T): List<T> {  
 val l = ArrayList<T>()  
 l.*addAll*(items)  
 return l  
}

val d = *arrayOf*(4, 4, 3)  
*println*(*toList*(\*d, 1, 2, 3, 4))

## if表达式

Kotlin中if**可以是一个语句也可以是一个表达式**,作为语句时格式与java相同,作为表达式时可以返回一个值,并且必须含有一个else分支,如果表达式是一个代码块,最后的表达式作为该块的值(Kotlin中不再有三元表达式)

var max = if (a > b) {  
 val c = a  
 c \* 2 + 1  
} else {  
 val d = b  
 d / 2 + 1  
}

## when表达式

Kotlin使用when替换了switch,when比switch更加灵活,when语句中可以使用任意表达式作为分支条件,会将参数与所有分支顺序匹配,在匹配到对应的条件后,执行后方的代码块结束运行,当分支条件是类型检查时后方的代码块也会自动类型转换.

var x = 123  
val arr = *arrayOf*(1, 2, 3, 4)  
when (x) {  
 1 -> x = 123 //分支可以是和switch一样的元素比较  
 123, 234 -> { //分支可以与逗号组合

x = 2  
 }  
 is Int -> *println*(x) //类型检查

in arr -> *println*("in arr") //是否在范围或集合  
 else -> *println*("ww") //else的作用相当于switch的default  
}

和if语句一样,when语句也可以是一个表达式,其值是代码块中最后一个表达式的值,作为表达式时必须有else分支:

var y = when (x) {  
 1 -> 2  
 2 -> 3  
 else -> 4  
}

when后面可以不带上参数,这时分支需要是布尔表达式,条件为true时执行对应分支,然后结束代码块.

val items = *listOf*("1", "2", "3")  
when {  
 "1" in items -> *println*("1 in")  
 "2" in items -> *println*("2 in")  
 "4" in items -> *println*("4 in")  
}

## 类型检查及自动类型转换

在if和when中使用is可以用于类型判断,判断后的分支中会自动将类型转为对应类型,可以直接当做对应类型使用,省略了显式转换:

fun test(s: Any) {  
 if (s is String) {  
 val c = s[1] //此处s已经自动转换为String  
 s.length  
 }  
 //或者使用 !is  
 if (s !is String)  
 return  
  
 s.length //此处s也已经自动转换为String  
 val c = s[2]  
}

## 强制类型转换

在kotlin中使用as进行强制类型转换: val tv = findViewById(R.id.*tv*) as TextView

当as前的内容为空时会抛出异常,若想防止该异常可以将其变为可空类型:

val tv :TextView? = findViewById(R.id.*tv*) as TextView?

或者使用安全的强转 as? ,在失败的时候会返回null

var tv = findViewById(R.id.*tv*) as? TextView

## 循环

Kotlin中 for循环可以遍历任何提供了迭代器的类型.

for(变量名 in 迭代的内容) {

}

若需要获取迭代对象的索引,迭代对象一般提供了indices方法,其返回值是一个可迭代的索引集合.

**for** (i **in** array.indices)

可以使用withIndex()方法,该方法会返回迭代对象的索引和值的Map

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

Kotlin中 while 循环和 do{}white{} 与java中完全相同,正常使用.

## 结构跳跃表达式

和java中一样,Kotlin中有 return , break ,continue,默认情况下与java中并没有什么不同

但是在Kotlin中,这些表达式后都可以添加一个@标记,表示对指定的位置进行对应操作.

在循环中,可以对指定循环添加标签,在break或countinue后添加标签,表示对指定循环进行跳转

abc@ for (i in 1..100) {  
 for (j in 1..100) {  
 if (...)

break@abc

}  
}

在方法中,因为有些参数是需要传入一个函数,当传入的函数为匿名函数时,在匿名函数中的return会只返回该匿名函数,但是当传入的函数是一个lambda表达式时,return会返回外部的函数,如果想要只返回lambda表达式的函数,就需要添加标签,当不添加标签时,标签默认为方法名.

fun foreach(ints: Array<Int>) {  
 ints.*forEach* adb@ **{** if (**it** > 2) return@adb  
 **}**}

fun getArray(size: Int): IntArray {  
 return IntArray(2, **{**

if (it > 10) {  
 return@IntArray it  
 } else {  
 return@IntArray it / 2  
 }  
 **}**)  
}

## 区间表达式

使用..或者until表示一个从小到大的区间,前面的数比后面的数大时,区间没有元素.

1..10 1 until 10

或者使用downTo 表达一个从大到小的区间,前面的数比后面的小时,区间没有元素. 10 *downTo* 1

在区间后可以加上step表示以以任意步长迭代区间. i **in** 1..4 step 2

在for循环中 使用 in 可以迭代区间中的元素 for (i in 1 *until* 10)

在if 判断中 使用in可以判断元素是否在区间中 if (x in 2..10)

## 解构声明

可以声明多个变量然后将一个对象解构赋值给这些变量.从对象中获取值通过componentN()函数实现.

在遍历map时使用解构声明: for ((key, value) in map)

如果函数的返回值是一个可解构的对象,可以使用解构声明从返回值中获取两个数据,看起来就像返回了多个返回值: var (a,b) = method()

如果在结构声明中有一些值是不需要的,可以使用 \_ 进行填充

for((\_, value) in map)

## 包Package

包的声明与java的相同,但是在kotlin中,包中可以包含属性、方法和类, 导包的方式也与java没有什么不同,当导入的类或属性或方法同名时 可以使用as进行重命名.

import com.example.myapplication.sum as sum2

## 类class

类的声明还是使用class关键字,类的声明需要类名 类头 和类体,当没有类体时可以省略花括号

类头包括 **主构造函数** 与其 **修饰符和注解**. 类可以没有类头

class类名 修饰符和注解 constructor(参数:类型)

若没有修饰符和注解,可以省略constructor关键字,构造函数的修饰符默认为public;

构造函数中的参数在没有添加修饰符时默认为val为常量,在创建对象时进行赋值后,在类中无法再修改.

主构造函数没有方法体,初始化代码可以放在 init 初始化程序块中,主构造函数参数可以在类中使用.

class Demo constructor(var name: String, var age: Int) {  
 init {  
 //初始化操作

}  
}

### 次级构造

可以在类体中使用constructor 声明次级构造,若该类具有主构造函数,每个次级构造都需要通过this直接或间接的将参数传入主构造进行初始化.

constructor类名(参数:类型) : this(参数){

}

class Demo constructor(var name: String, var age: Int) {  
 constructor(name: String, age: Int, job: String) : this(name, age)  
 constructor(name: String, age: Int, job: String, price: Int) : this(name, age, job)  
}

如果需要对主构造或次构造中的参数设置默认值,格式为: 参数名 : 类型 = 值 ,类型不可省略.

### 创建类的对象

kotlin中没有new关键字,创建对象时直接调用其构造函数即可. var a = Demo("1", 2)

若一个类没有主构造也没有次级构造,可以不传入参数就创建该类对象;

若其没有主构造,有一个次级构造,那这个次级构造会作为主构造;

若有多个次级构造,其主构造实际上是一个空构造,但是无法不传入参数就创建该类对象.

若主构造中声明的所有属性都具有默认值,则该类实际上拥有一个空构造,可以使用空构造进行创建对象.

### 类的继承

Kotlin中的超类不再是Object而是Any,Any中只有equals()，hashCode()和toString().

如果要声明一个可继承的类,需要在class前面加上open修饰符,没有加上open修饰的类默认都无法继承,相当于被final修饰.子类继承时在主构造后使用 : 父类名

如果父类有无参构造,可以在主构造后使用 : 父类名() 进行父类的初始化;

如果父类没有无参构造,则子类构造必须传入参数进行初始化.如果子类没有主构造,每个次级构造需要使用super关键字初始化基类的构造,每个次级构造可以调用父类不同参数的构造,或调用自己的另一个初始化了基类构造的构造.

即: 每个子类的构造都需要间接或直接的访问到父类的某一个构造,和java中基本相同,只是写法不同.

class MyView : View {  
 constructor(context: Context) : super(context)  
 constructor(context: Context, attrs: AttributeSet) : this(context)  
 constructor(context: Context, attrs: AttributeSet, defStyleAttr: Int)  
 : super(context, attrs, defStyleAttr)  
}

#### 方法重写和属性覆盖

与类一样,类中可进行重写的方法和属性必须使用open修饰,未被open修饰的方法和属性默认不能进行重写,子类进行重写后重写的方法和属性必须使用override进行修饰,被override修饰的方法和属性默认是open的,若要防止被继续重写可以添加final修饰. 子类不能声明与父类中相同名称并不具有open修饰的方法和属性.属性覆盖时,即使父类的属性修饰为val,也可以使用var覆盖.

open class Demo2 {  
 open fun a() {  
 }  
 open var a: Int = 0  
  
}  
  
open class Demo3 : Demo2() {  
 override fun a() {  
 }  
 override var a: Int = 0;  
}

### 抽象类

抽象类和抽象方法使用abstract进行修饰,抽象类和抽象方法不需要open进行修饰即可继承和重写.

在抽象类中可以将父类中的非抽象方法重写为抽象方法.

### 数据类

在kotlin中,针对java中的model(bean类)提供了方便的实现,使用data修饰class创建一个数据类,数据类的主构造函数需要至少有一个参数,数据类不能抽象、开放和不能以内部类形式实现

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

数据类会自动生成equals、hashCode、toString和copy方法:

**equals、hashCode** : 当数据类中的数据是基本数据类型和字符串时,内容相同的数据类的实例会指向同一个hashCode,他们的equals也相同,若添加了其他数据类型,一般需要重写equals方法.

**copy**:可以方便的对一个实例中的数据进行修改,其实现方式类似于:

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

使用:

var a = User("111111111111", 33332)  
var c = a.copy(name = "2222")

**使用解构声明获取数据类中的数据:**

数据类中包含componentN()方法,可以使用解构声明获取数据类中的数据

var a = User("111111111111", 33332)  
var (d, e) = a //声明的d和e会按照a中声明的属性的顺序进行赋值.

### 内部类

可以在类中进行类的声明,但是普通的内部类声明,内部类不能访问外部类的属性和方法.使用**inner**修饰,即可访问外部类的属性和方法. **inner class {}**

非inner修饰的内部类在外部进行访问时需要创建该类的对象,使用inner修饰的内部类无法创建对象,访问时只能通过外部类的对象进行访问.

class E {  
 inner class Inner{  
 var a = 1  
 }  
}  
class F {  
 class Inner  
}

访问:

val a = E()  
*println*(a.Inner().a)  
val b = F.Inner()

## 接口interface

Kotlin中的接口可以包含抽象方法和具有方法体的方法,具有方法体的方法可以不被重写.

接口中可以声明属性,进行实现时与类不同,常量无法被覆盖为变量,反之也是.

常量可以提供访问器get,不能进行初始化赋值,若提供了get,实现类不必强行重写该常量并实现;若没有提供访问器get必须在实现类中对该常量进行实现;

声明的变量不能提供访问器,在实现类中必须进行实现.

接口中的属性和方法的访问只能通过其实现类的对象访问,即使是拥有访问器的常量.

接口的实现和类的继承的格式一样, 实现多个接口时使用逗号将接口名分开 : 类名,接口名,接口名

若实现的多个接口中有同名的具有方法体的方法 或 类中可重写的方法与接口中具有方法体同名,重写之后若要访问接口或父类中的该方法体,使用 super<类名/接口名>.方法名() 进行访问对应接口或父类中的该方法的方法体. super<IA>.m2()

interface IA {  
 val a: Int //声明的常量可以提供访问器,不必须被重写  
 get() = 2  
  
 var b: String //变量不能提供访问器,必须被重写  
 val c: String //声明的常量不提供访问器,必须被重写  
  
 fun m1() //抽象方法必须被重写  
  
 fun m2() { //具有方法体的方法可以不被重写  
 }  
}  
  
interface IB {  
 fun m2() {  
 }  
}  
  
class C : IA, IB {  
 override val a: Int  
 get() = 1  
  
 override val c: String = ""  
  
 override var b: String = ""  
  
 override fun m1() {  
 }

override fun m2() {  
 super<IA>.m2()  
 super<IB>.m2()  
 }  
}

## 对象表达式(匿名内部类)

**直接创建一个匿名内部类对象:** 关键字var a = object { var x = 1} println(a.x)

**接口的匿名内部类对象**:因为接口没有构造方法,所以并不需要初始化参数 : object : 接口名 {}

*tv*.setOnClickListener(object : View.OnClickListener {  
 override fun onClick(v: View?) {  
 *tv*.*text* = ""  
 }  
})

当需要传入的对象是函数式接口的对象时,可以使用lambda表达式进行表达:

*tv*.setOnClickListener **{** *tv*.*text* = "" **}**

**可继承类的匿名内部类对象:**类拥有构造函数,在声明一个匿名内部类时需要对父类的构造进行初始化:

object : 类名(初始化参数) {}

open class A(x: Int, y: Int) {  
 open val y: Int = x  
}  
  
val *aa* = object : A(1, 2) {  
 override val y = 3  
}

**在对象表达式中也可以访问可以访问封闭的作用域,并且不像java只限于final变量.**

匿名对象只能作为本地和私有声明中使用,如果作为公共函数的返回值或公共属性,其实例的实际类型是匿名对象的超类型,如果没有声明超类型其类型是Any,在匿名对象中添加的成员将都不能被访问到.

### 伴生对象

使用对象表达式将一个对象声明在一个类中时,可以使用companion关键字修饰作为一个伴生对象,伴生对象中的成员使用类名称进行调用,类似于java的静态方法(实际上在运行的时仍然是实体的实例成员),伴生对象可以实现接口和继承类.伴生对象的名称可以省略.

class B : A() {  
 companion object : A() {  
 fun method() {}  
 var a = 0  
 }  
}

调用伴生对象的成员: B.method() B.a

**伴生对象也可以扩展函数和扩展属性**,当声明的伴生对象有名称时,使用 类名.伴生对象名.扩展名 进行扩展,当伴生对象没有名称时,使用 类名. Companion.扩展名 进行扩展.

fun B.Companion.method2() { }

### 对象声明

可以使用object关键字声明一个对象,声明的对象可以具有超类型.

对象声明格式类似于class的声明,但是他不需要再去进行对象的创建,当他声明时即已经是一个对象,使用时直接使用他的名称.对象声明相当于一个类的单例模式.

除了不能在函数和内部类中进行对象声明,其他地方都可以进行对象声明.

object click : View.OnClickListener {  
 override fun onClick(v: View?) {  
 }  
}

*tv*.setOnClickListener(click)

## 异常

kotlin中在编译时没有进行异常检查,即不会强制要求对可能的异常进行捕获,比如io操作时,java会在编译期就报错要求捕获异常,但是kotlin不会.

**抛出异常**和java中一样使用throw,throw的类型是特殊的Nothing.

fun fail(message: String): Nothing {  
 throw IllegalArgumentException(message)  
}

**捕获异常**还是使用try cache finally,和java中的规则一样,不过kotlin中,try语句可以作为一个表达式,其返回的值是 try和cache中最后的一个表达式,finally中的代码不对返回的值影响:

var b = try {  
 "1B".*toInt*()  
} catch (E: NumberFormatException) {  
 "123"  
}