# Kotlin基础

## 变量

使用“val”或者是“var”关键字作为开头，后面跟“变量名称”，接着是“变量类型”和“赋值语句”

在Java中，如果我们要声明变量，我们必须要声明它的类型，后面跟着变量的名称和对应的值，然后以分号结尾。

// Integer age = 30;

各部分的意义如下：

关键字 变量类型  
 ↓ ↓ \*/  
 var age : Int = 30 /\*  
 ↑ ↑  
 变量名 变量值

分号：在Kotlin中，不需要分号，若要隔开，换行隔开，或，;隔开

var age1 : Int = 100 ; var age2 : Int = 100  
var age3 : Int = 100  
var age4 : Int = 100

类型推导：在Kotlin中，变量类型可以省略不写，会通过值推导出类型

var age5 = 100 // 背后隐式代码 var age5 : Int = 100 --- 默认推导类型为： Int

多用val：val是只读，var是可读可改，只要是不再修改，都用val

var age6 = 100  
age6 = 101 // var可改  
*println*(age6) // var可读  
  
val age7 = 100  
// age7 = 101 // val不可改，编译器报错  
*println*(age7) // val只读

// TODO 基础类型

基础类型：在Java中，有两种类型，整型int和Integer，前者是原始类型（基本类型），后者是包装类型（引用对象）。

/\*  
int i = 0; // 原始类型  
Integer j = 1; // 包装类型  
\*/

在Kotlin中，没有原始类型这个概念，Kotlin一切都是对象

// Double kotlin class  
// Float kotlin class  
// Long kotlin class  
// Int kotlin class  
// Short kotlin class  
// Char kotlin class  
// Boolean kotlin class

Kotlin基础类型的良苦用心：

Java的基础类型并不是完全面向对象，因为它存在原始类型，而原始类型并不属于对象。

而Kotlin则不一样，它从语言设计的层面上就规避了这个问题，基础类型则是完全面向对象的。

val d1 : Double = 100.toDouble().toLong().toDouble()

空安全：由于 基础类型则是完全面向对象的，那么对象就有可能为空，那么空安全机制就应运而生了

// val d2 : Double = null // 非可空类型，不能赋值为null，编译不通过  
val d3 : Double? = null // 可空类型，能赋值为null，编译通过

可空类型(可能编译后 包装类型) > 非可空类型 (编译后一直都是基本类型)

包装类型 = 基本类型 先这样理解

为空类型： 1.可以存null值。 2.可以存正常值。

非可空类型：1.只能存正常值。 不能给null值，编译不通过

var d4 : Double = 6588776.7  
var d5 : Double ? = null  
// d4 = d5 // 编译不通过，非可空类型 不能 接收 可空类型 小 = 大  
// d5 = d4 // 编译通过，可空类型 可以 接收 非可空类型 大 = 小  
  
// d4 = d5 // 编译不通过 的 解决方案（靠Kotlin编译期的监测）  
if (d5 != null) d4 = d5

Java隐式转换 VS Kotlin显示转换

int intValue = 9999; 短 小  
long longValue = intValue; 长 大  
  
longValue = intValue 1 有问题2（切点，自动转，你看不到，隐式的，有隐患，没有把这个转的过程暴露出来）

Kotlin推荐显示转换（好处：1.开发者自己转换 2.可读性特别强 3.一切都是透明的）

val intValue = 666  
val longValue : Long = intValue.toLong() // intValue.toLong这个是开发者写的，是开发者的责任

Boolean类型

val isOk = true  
val isNotOk = false  
*println*(isOk)  
*println*(isNotOk)

Char类型

val c = 'A'  
// val code : Int = c // 编译器报错  
val code : Int = c.toInt() // 必须透明化，可读性强，一切都需要暴露出来  
*println*(code) // 65

String字符串摸版

val name = "Derry"  
*print*("你的名字是:$name!")  
  
val array = *arrayOf*("Java", "Kotlin")  
*print*("你在学习什么${array[1]}语言呀!")  
  
val info = """  
 仍是雨夜  
 凝望窗外  
 沉默的天际  
 问苍天  
 可会知  
 心里的感觉  
 """  
*println*(info)  
  
val info2 = """  
 仍是雨夜  
 凝望窗外  
 沉默的天际  
 问苍天  
 可会知  
 心里的感觉  
 """.*trimIndent*()  
*println*(info2) // 去除缩进

数组 Ctrl + Alt + Shift + P 序列化了

val arrayInt = *arrayOf*(100000, 200000, 300000) // 类型会被推导为整型数组 Array<Int>  
val arrayString = *arrayOf*("张三", "李四", "王五") // 类型会被推导为字符串数组 Array<String>

数组和列表的长度在java中，数组长度使用 length ，列表长度使用size()方法获得

而在Kotlin中，这个问题统一采用size (array.size list.size)

*arrayOf*(100000, 200000, 300000).size  
*listOf*(100000, 200000, 300000).size

函数调用

derryFunction("Derry")

derryFunction2(name = "Derry") // 命名参数：明确可见的参数名，可读性更强

fun createStudent( name: String, age: Int, gender: Int, friendCount: Int, feedCount: Int, likeCount: Long, commentCount: Int) {}

createStudent("Zhangsan", 30, 1, 78, 2093, 10937, 3285) // 可读性差,Java的做法（建造者设计模式）

createStudent( name = "Zhangsan", age = 30, gender = 1, friendCount = 78, feedCount = 2093, likeCount = 10937, commentCount = 3285) // 可读性强，易维护

createStudent( feedCount = 2093, likeCount = 10937, commentCount = 3285, name = "Zhangsan", age = 30, gender = 1, friendCount = 78) // 顺序灵活，难写错

默认参数（例如：一个函数一百个参数，用户使用不想传那么多参数，怎么办？ Java采用建造者设计模式解决， Kotlin可以直接用 默认参数解决）

fun createTeacher(name: String, age: Int, gender: Int = 1, friendCount: Int = 0, feedCount: Int = 0, likeCount: Long = 0L, commentCount: Int = 0) {}

createTeacher( name = "李元霸", age = 30, commentCount = 3285) // 其他的参数不想传递，就启动默认参数

流程控制

if表达式（Java的if是语句，Kotlin的if是表达式 可以返回 灵活多变 并且有语句的功能）

val isSuccessful = true  
val responseResult = if (isSuccessful) "恭喜你，登录成功" else "不恭喜，你登录失败了!"  
*println*(responseResult)

Elvis表达式 ?: 在函数中妙用

fun getNameLength(name: String) : Int {  
 return if (name != null) name.length else -1  
}  
fun getNameLength2(name: String?) : Int{  
 return name ?.length ?: -1 // name等于null，就会返回-1， name有值 就会返回值长度  
 // xxx ?: 如果xxx是null 就会执行 ?: 后面的区域代码  
}  
*println*(getNameLength2("Derry"))

when表达式 （Java都是语句，Kotlin的if when 等等 是表达式 可以返回 灵活多变 并且有语句的功能）

val number = 1  
val message : String = when(number) {  
 1 -> "number等于一"  
 2 -> "number等于二"  
 else -> "number 不是一也不是二" // 如果去掉这行，会报错  
}  
*print*(message)

for循环特点

val numberArray = *arrayOf*(100, 200, 300, 400, 500, 600)  
for (i in numberArray) {  
 *println*(i)  
}  
  
*println*()  
  
for (i in 100..600) { // 包含了 100 到 600  
 *println*(i)  
}  
  
*println*()  
  
for (i in 600 *downTo* 100 *step* 2) { // 600 到 100  
 *println*(i)  
}

函数声明

Kotlin的函数 更加符合人类正常思维， 先有输入，再有输出

/\*  
关键字 函数名 参数类型 返回值类型  
 ↓ ↓ ↓ ↓ \*/  
fun derryFunction(name: String): String {  
 return "你的姓名是:$name!"  
}/\* ↑  
 花括号内为：函数体  
\*/

单一表达式函数

fun derryFunction2(name: String) : String = "你的姓名是:$name!"  
fun derryFunction3(name: String) = "你的姓名是:$name!" // 通过返回值类型推导出，返回类型String

函数类型

函数类型是 (输入类型) -> 输出类型

1. 下面的函数类型是什么

例1

() -> Unit 输入没有 -> 输出没有

val method01 : () -> Unit = **{** *println*("method01 run") **}**method01()

例2 (String) -> Boolean

val method02 : (String) -> Boolean = **{**str: String **->** true // Boolean  
**}***println*(method02("Derry"))

例3 (String) -> Int

val method04 : (String) -> Int = **{** str: String **->** str.length // Int  
**}**

例4 (Int,Int) -> Int

val method05 = **{**n1: Int, n2: Int **->** n1 / n2 // Int  
**}**

例5 (Any) -> Any Any == Object

val method06 = **{**any: Any **->** any // Any  
**}***println*(method06(5465))  
*println*(method06("Derry"))  
*println*(method06(true))

例6 (Char)-> Unit

val method07 = **{** sex: Char **->** Unit  
 *println*(if(sex == '男') "先生你好" else if(sex == '女') "女士你好" else "人妖") // Unit  
**}**method07('男')  
method07('女')  
method07('A')

例7： (Int, Int, Int) -> Int

val method08 : (Int, Int, Int) -> Int = **{**n1: Int, n2:Int, n3:Int **->** n1 + n2 + n3  
**}**

例8：(Int, String, Float) -> String

val method09 = **{**n1: Int, n2:String, n3:Float **->** "你的姓名:$n2,你的年龄:$n1,你的体重:$n3" // String  
**}**

例9：(List<String>) -> Unit

val method10 = **{** str: List<String> **->** for(i in str) {  
 *println*("你的每一条是:$i") // Unit  
 }  
**}**

例10：(() -> Unit) -> Unit

val method11 = **{** a1: () -> Unit **->** a1() // 调用a1函数 调用a1会返回 a1 的 Unit  
**}**  
method11 **{** *println*("我是a1函数的实现，我被调用了")  
**}**

例11：(String, String) -> Unit

val method12 = **{** userName: String, userPwd: String **->  
  
}**

例12：(String, String, (Boolean) -> Unit) -> Unit

val method13 = **{** userName: String, userPwd: String, a : (Boolean) -> Unit **->** if(userName == "Derry" && userPwd == "123456") {  
 a(true) // 登录成功 调用 a 方法传入 true  
 } else {  
 a(false) // 登录失败 调用 a 方法传入 false  
 }  
**}**  
method13("Derry2", "123456") **{** if (**it**) *println*("恭喜你，登录成功") else *println*("不恭喜，登录失败")  
**}**

例13：(() -> Unit) -> Unit

val method14 = **{** a : () -> Unit **->** a() // 调用 a函数  
**}**method14 **{** *println*("a函数 实现被执行了")  
**}**

例14： 考点 method15函数 再返回一个函数

val method15 = **{** str: String **->** // 函数 输入 输出 (Int) -> String  
 **{** n1: Int **->** "n1:$n1 我是String类型的字符串" // String  
 **}  
}**// 为什么是两个括号 第一个括号 调用method15函数而已  
// 第二个括号 调用method15函数 返回的函数  
*println*(method15("Derry")(999))

# Kotlin 委托

委托的含义，将接口的实现委托给参数，这样就不用实现具体接口类

## 类委托1

具体例子如下：

interface DB {  
 fun save()  
}  
  
class SqlDB() : DB {  
 override fun save() = *println*("save to sql")  
}  
  
class MySqlDB() : DB {  
 override fun save() = *println*("save to MySqlDB")  
}  
  
class OracleDB() : DB {  
 override fun save() = *println*("save to Oracle")  
}

使用方法：

fun onDelegateBase(v : View) {  
 CreateDBAction(SqlDB()).save()  
 CreateDBAction(OracleDB()).save()  
 CreateDBAction(MySqlDB()).save()  
}

Kotlin通过编译实际上生成了以下的代码：

public final class CreateDBAction implements DB {  
 // $FF: synthetic field  
 private final DB $$delegate\_0;  
  
 public CreateDBAction(@NotNull DB db) {  
 Intrinsics.checkParameterIsNotNull(db, "db");  
 super();  
 this.$$delegate\_0 = db;  
 }  
  
 @Override  
 public void save() {  
 this.$$delegate\_0.save();  
 }  
}

## 类委托2

委托示例2

interface IUSB {  
 fun insert()  
  
 /\*fun insert2()  
 fun insert3()  
 fun insert4()  
 fun insert5()  
 // 20\*/  
}  
  
class Mouse : IUSB {  
 override fun insert() = *println*("鼠标插入了USB")  
}  
  
class KeyBard : IUSB {  
 override fun insert() = *println*("键盘插入了USB")  
}  
  
// 委托含义解释：把接口的实现，委托给了参数iusb  
class CreateUSB(iusb: IUSB) : IUSB by iusb  
/\*  
 背后生成的代码：  
 public final class CreateUSB implements IUSB {  
  
 private IUSB $$delegate\_0;  
  
 public CreateDB(IUSB iusb) {  
 this.$$delegate\_0 = iusb;  
 }  
  
 @Override  
 public void insert() {  
 this.$$delegate\_0.save();  
 }  
 }  
 \*/  
  
// 不使用委托(全部都要自己完成实现) 如果使用委托 实现细节交给委托完成，我们不管了  
/\*class CreateUSB2(iusb: IUSB) : IUSB {  
  
 val iusb\_ = iusb  
  
 override fun insert() {  
 iusb\_.insert()  
 }  
  
 override fun insert2() {  
 *TODO("Not yet implemented")*  
 }  
  
 override fun insert3() {  
 *TODO("Not yet implemented")*  
 }  
  
 override fun insert4() {  
 *TODO("Not yet implemented")*  
 }  
  
 override fun insert5() {  
 *TODO("Not yet implemented")*  
 }  
}\*/

使用方法：

fun onDelegateBase(v : View) {  
 CreateUSB(Mouse()).insert() // new CreateUSB(new Mouse()).insert();  
 CreateUSB(KeyBard()).insert() // new CreateUSB(new KeyBard()).insert();  
}

## 类委托3

interface Database { // 接口  
 fun insert()  
}  
  
class MySql : Database { // 实现类  
 override fun insert() = *println*("insert run")  
}  
  
// by可以把database接口中的方法，委托给 (database: Database 委托对象 $$delegate\_0)  
// 委托类，必须是接口才行  
// 调用 处理 的细节，交给了 委托，帮我们完成  
class CreateDatabase(database: Database) : Database by database // 委托含义解释：把Database接口的实现 委托给了 参数database  
/\*  
 背后生成的代码：  
 public final class CreateDatabase implements Database {  
 private Database $$delegate\_0; // 委托者 就是 代理  
  
 public CreateDatabase(Database database) {  
 $$delegate\_0 = database;  
 }  
  
 // 重写了接口的方法， 将 insert 委托给了 $$delegate\_0.insert();  
 @Override  
 public void insert() {  
 $$delegate\_0.insert();  
 }  
 }  
\*/

fun onDelegateBase(v : View) {  
 // 传不同的委托对象进去，它就会有不同的行为。  
 CreateDatabase(MySql()).insert()  
}

## 属性委托

委托类委托的是接口的方法，委托属性委托的是属性的 set get方法。

一个val属性，kotlin内部会为它自动生成get方法，表示只读。一个var属性，kotlin会为他生成get和set方法，表示可读可写。

下面是属性委托的一个例子：

class Simple04 {  
  
 // var 可读 可改 内部会有 set get val 只有get 只读  
 var floatValue : Float = 757567.65f  
 set(v) {  
 **field** = v  
 *println*("你设置了 floatValue哦 v:$v")  
 }  
 get() {  
 *println*("你获取了 floatValue哦")  
 return **field** }  
  
 var number : Float by ::floatValue  
 /\*set(v) { 相当于，隐式的有这段代码哦  
 field = v  
 println("你设置了floatValue哦 v:$v")  
 }  
 get() {  
 println("你获取了floatValue哦")  
 return field  
 }\*/  
   
}

fun onDelegateBase(v : View) {  
 val simple04 = Simple04()  
 simple04.number = 99999.0f // 我给number赋值【用户在赋值 number 调用 setNumber ---> 实例 ---> setFloatValue】  
 *println*(simple04.number) // 我读取number【/用户在读取 number 调用 getNumber ---> 实例 ---> getFloatValue】  
}

实际背后生成的代码是：

public final class Simple04 {  
 // 实例 方便访问 getFloatValue setFloatValue 的实例  
 private final KMutableProperty0 number$delegate = new Simple04$number$2((Simple04)this);  
  
 private float floatValue = 8768979.0F;  
  
 public final float getFloatValue() {  
 String var1 = "你获取了floatValue哦";  
 System.out.println(var1);  
 return this.floatValue;  
 }  
  
 public final void setFloatValue(float v) {  
 this.floatValue = v;  
 String var2 = "你设置了floatValue哦 v:" + v;  
 System.out.println(var2);  
 }  
  
 // 用户在读取 number 调用 getNumber ---> 实例 ---> getFloatValue  
 public final float getNumber() {  
 KProperty0 var1 = (KProperty0)this.number$delegate;  
 Object var3 = null;  
 return ((Number)var1.get()).floatValue();  
 }  
  
 // 用户在赋值 number 调用 setNumber ---> 实例 ---> setFloatValue  
 public final void setNumber(float var1) {  
 KMutableProperty0 var2 = this.number$delegate;  
 Object var4 = null;  
 Float var5 = var1;  
 var2.set(var5);  
 }  
}

## 懒加载

懒加载的意思是第一次使用的时候才会去初始化，如果已经初始化，下次使用直接使用初始化的对象，不重复进行初始化

*/\*\*  
 \* 测试  
 \*/*fun onDelegateBase(v : View) {  
 *println*("准备工作中...")  
 Thread.sleep(3000L)  
  
 *println*("开始请求中")  
 *println*(responseData) // responseData如果没有值，只有在用responseData的时候，才会加载，这就是懒加载  
 *println*(responseData) // responseData如果有值了，会直接返回  
 *println*(responseData) // responseData如果有值了，会直接返回  
}  
  
fun requestDownload() : String {  
 *println*("requestDownload run 》》》》》》")  
 Thread.sleep(2000L) // 模拟下载延时了  
 return "恭喜你，下载完成了"  
}  
  
val responseData : String by *lazy* **{** requestDownload() **}**

## 自定义委托

自定义委托中，对于var的属性，必须要有set和get方法

对于String类，set传入的参数类型是String，get返回类型是String

Get和set方法第一个参数必须是是包含本属性的类或者包含本属性类的父类。

标准写法如下：

class Owner {  
  
 // var （自定义委托中，必须有 set get）  
 // String （自定义委托中，get返回String，set传入String）  
 // get/set方法 第一个参数 必须包含 本类Owner或Owner父类  
 var text: String by Simple07()  
  
 var text2 : String by StringDelegate()  
}  
  
class Simple07 {  
  
 private var str: String = "Default"  
  
 operator fun getValue(owner: Owner, property: KProperty<\*>) : String {  
 *println*("getValue执行啦")  
 return str  
 }  
  
 operator fun setValue(owner: Owner, property: KProperty<\*>, value : String) {  
 *println*("setValue执行啦")  
 str = value  
 }  
}  
  
class StringDelegate() : ReadWriteProperty<Owner, String> {  
  
 private var str: String = "Default"  
  
 override fun getValue(thisRef: Owner, property: KProperty<\*>): String {  
 *println*("ReadWriteProperty getValue执行啦")  
 return str  
 }  
  
 override fun setValue(thisRef: Owner, property: KProperty<\*>, value: String) {  
 *println*("ReadWriteProperty setValue执行啦")  
 str = value  
 }  
  
}

*/\*\*  
 \* 测试  
 \*/*fun onDelegateBase(v : View) {  
 val o = Owner()  
 o.text = "DERRY"  
 *println*(o.text)  
  
 o.text2 = "AAAA"  
 *println*(o.text2)  
}

## 提供委托

当访问委托属性的时候，会自动调用委托类的provideDelegate的方法，然后它就不管其他了。

// *TODO 提供委托（provideDelegate）*class StringDelegateUpdate(var str: String = "Default") : ReadWriteProperty<Owner2, String> {  
  
 override fun getValue(thisRef: Owner2, property: KProperty<\*>): String {  
 *println*("ReadWriteProperty getValue执行啦")  
 return str  
 }  
  
 override fun setValue(thisRef: Owner2, property: KProperty<\*>, value: String) {  
 *println*("ReadWriteProperty setValue执行啦")  
 str = value  
 }  
  
}  
  
class SmartDelegator {  
  
 // 相当于是一个选择器，动态选择 是哪个 StringDelegateUpdate自定义委托  
 operator fun provideDelegate(  
 thisRef : Owner2,  
 property: KProperty<\*>) : ReadWriteProperty<Owner2, String> {  
  
 // 逻辑是自己决定的  
 return if (property.name.*contains*("aaa")) {  
 StringDelegateUpdate("bbb")  
 } else {  
 StringDelegateUpdate("aaa")  
 }  
 }  
}  
  
class Owner2 {  
 var aaa: String by SmartDelegator()  
 var bbb: String by SmartDelegator()  
}

*/\*\*  
 \* 测试  
 \*/*fun onDelegateBase(v : View) {  
 val owner = Owner2()  
 *println*(owner.aaa)  
 *println*(owner.bbb)  
}

## 委托应用

这里实现一个需求：内部可以修改，但是 外部只能读取

实现的原理是对外暴露不可修改的属性，内部通过委托给一个可写属性。

class Model2 {  
  
 // *TODO 需求：内部可以修改， 但是 外部只能读取* // List不可修改集合  
 val data : List<String> by :: \_data  
  
 // MutableList可以修改的集合  
 private val \_data : MutableList<String> = *mutableListOf*()  
  
 fun load() {  
 \_data.add("Hello") // 内部可以修改  
 }  
}

fun onDelegateBase(v : View) {  
 // 外部不能修改，只能读取  
 val model = Model2()  
 // model.data.add("Hello") // 报错，不可修改集合，压根就没有add方法  
 // model.\_data // 报错，被私有化了  
 *println*(model.data)  
}

## 控件委托

*/\*\*  
 \* 测试  
 \*/*fun onDelegateBase(v : View) {  
 // 对控件自定义委托（DataBinding）  
 var textView : TextView = findViewById(R.id.*tv\_title*)  
 var message : String? by textView  
 textView.*text* = "Derry修改了布局"  
 *println*("message:$message") // 布局 --> 数据message 一向  
 message = "数据发送改变"  
 *println*("textView:${textView.*text*}") // 数据 --> 布局 一向 = 双向绑定  
}  
  
// 自定义委托，提供者委托，可以更加直接的写法  
operator fun TextView.provideDelegate(value : Any?, property: KProperty<\*>) =  
 object: ReadWriteProperty<Any?, String?> {  
 override fun getValue(thisRef: Any?, property: KProperty<\*>): String? = *text* as String?  
 override fun setValue(thisRef: Any?, property: KProperty<\*>, value: String?) {  
 *text* = value  
 }  
 }

# 目录1

## 目录2

### 目录3

#### 目录4

##### 目录5