# 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  
 }  
 }

## 委托的另外一种写法

private operator fun String.setValue(item: TestDelegateActivity11.Item, property: KProperty<\*>, value : String) {  
 // this = value  
 *println*(property) // var com.derry.lib\_delegate.simple4.Item.version222: kotlin.String【能够动态得到，所以可以反射】  
 // Kotlin的反射才能做到了【反射：修改 version222变量的值 为 value】  
 property.*javaField*?.*isAccessible* = true  
 property.*javaField*?.set(item, value)  
 // 只能修改 version222变量的值， 不能修改 version111变量的值  
}  
  
// getValue最容易的  
private operator fun String.getValue(item: TestDelegateActivity11.Item, property: KProperty<\*>)  
 = this + "自定义getValue" // == String本身 == "Version1 data"

class Item {  
 var version111 : String = "Version1 data"  
 var version222 : String by version111  
}

fun onDelegateBase(v : View) {  
 val item = Item()  
 *println*(item.version222)  
  
 *println*("item.version222:${item.version222} item.version111:${item.version111}")  
 item.version222 = "TTTTTT"  
 *println*("item.version222:${item.version222} item.version111:${item.version111}")  
}

## 监听新旧值

// *TODO 同学们注意，新增点（观察 新值 旧值）*class Simple1 {  
 var name: String by Delegates.observable("Test") **{** prop, old, new **->** *println*("旧值：$old -> 新值：$new")  
 **}**}

fun onDelegateBase(v : View) {  
 val simple1 = Simple1()  
 simple1.name = "Update1"  
 simple1.name = "Update2"  
}

# Lamda表达式

## 函数声明

函数的声明 用lambda去描述函数的声明

val method1 : () -> Unit  
  
val method2 : (Int, Int) -> Unit  
  
val method3: (String, Double) -> Any  
  
var method4 : (Int, Double, Long, String) -> Boolean  
  
var method5 : (Int, Int) -> Int

## 函数实现

val method01 : () -> Unit = **{** *println*("我是method01函数") **}**method01() // 调用函数 () == 操作符重载 invoke操作符  
method01.invoke() // 调用函数  
  
val method02 = **{** "我是method02函数" **}***println*(method02())  
  
val method03 = **{**str: String **->** *println*("你传递进来的值是:$str")**}**method03("Derry")  
  
val method04 = **{**number1: Int, number2:Int **->** number1.toString() + number2.toString()**}***println*(method04(1, 2))  
  
val method05 = **{**number1: Int, number2:Int **->** number1 + number2**}***println*(method05(1, 2))  
  
val method06 : (Int) -> String // 先声明  
method06 = fun(value: Int) : String = value.toString() // 后实现  
*println*(method06(88))  
  
// 用 先声明 后实现 来 自动推断  
val method07 : (Int) -> String // 先声明  
method07 = fun(value) = value.toString() // 后实现  
*println*(method07(99))  
  
// 声明 + 实现 一气呵成  
val method08: (Int) -> String /\*左边的是声明\*/ /\*右边是实现\*/ = **{**value**->** "$value aaa" **}***println*(method08(77))  
  
val method09 : (String, String) -> Unit = **{**aStr, bStr **->** *println*("aStr:$aStr, bStr:$bStr") **}**method09.invoke("AA", "BB")  
  
val method10 : (String) -> Unit = **{** /\*it ->\*/ // 如果你只有一个参数的话，如果你不写，默认就有it，他会自动给你增加it  
 *println*("你传递进来的值是:$**it**")  
**}**method10("李元霸")  
  
val method11 : (Int) -> Unit = **{** when(**it**) {  
 1 -> *println*("你传递进来的是一") // Unit  
 in 20..60 -> *println*("你传递的是 20 到 60 之内的数字") // Unit  
 else -> *println*("都不满足") // Unit  
 }  
**}**method11(60)  
  
val method11s : (Int) -> String = **{** when(**it**) {  
 1 -> "你传递进来的是一" // String  
 in 20..60 -> "你传递的是 20 到 60 之内的数字" // String  
 else -> "都不满足" // String  
 }  
**}***println*(method11s(60))

## 拒收

传入参数使用\_表示该参数已经被拒收。

val method13 : (Int, Int) -> Unit = **{**\_, number2**->** *println*("你传递的第二个参数是：$number2")  
**}**method13.invoke(100, 200) // 用户对 不用修改

## 函数接收所有参数类型

// (Any) -> Any  
val method14 = **{**str: Any **->** str**}***println*(method14(true))  
*println*(method14(57.654f))

## 函数调用invoke

val method15 /\*: (Char) -> Unit\*/ =  
 **{**sex : Char **->** *println*(if (sex == '男') "你是男生" else if (sex == '女') "你是女生" else "未知") **}**method15.invoke('男')

## 方法覆盖

// (Int) -> Unit  
var method16 = **{**number: Int **->** *println*("你的值是:$number")**}** // 被覆盖后，就失效了  
// 覆盖method16, 虽然你覆盖了，类型还是 (Int) -> Unit  
method16 = **{** *println*("覆盖的也能拿到值:$**it**") **}**method16(999)  
  
val method16ss : (String ? /\*= ""\*/, String) -> Unit = **{**str /\*= "default"\*/, str2**->** *println*("str:$str, str2:$str2")  
**}**method16ss("李元霸", "李连杰")  
method16ss(null, "李连杰")

## 传入什么参数就输出什么

// (Any) -> Any  
val method17 = **{**str: Any **->** *println*("你传递了:$str")  
 str // lambda最后一行作为返回值  
**}***println*(method17(34534))  
*println*(method17('C'))  
*println*(method17("DDDD"))

## 匿名扩展函数

// 给String增加 匿名扩展函数  
val method19 : String.() -> Unit = **{** // this == String本身 == 调用者本身 == "DDD"  
 *println*("你是$this")  
**}**"DDD".method19()  
  
val method20 : Int.(Int) -> String = **{** "两数相加的结果是:${this + **it**}" **}***println*(1.method20(100))  
*println*(method20(1, 100))  
  
val method21 : Double.(Double, Double) -> Unit = **{**d1, d2 **->** *println*("三数相加:${this + d1 + d2}")**}**method21.invoke(55.5, 66.6, 77.7)  
55.5.method21(66.6, 77.7)  
  
val method22 : Char.(String) -> Unit  
 = **{** *println*(if (this == '男') "先生你好" else if (this == '女') "女生你好" else **it**)**}**'男'.method22("未知人类")  
'A'.method22("未知人类")  
  
fun Char.method23(default: String) {  
 *println*(if (this == '男') "先生你好" else if (this == '女') "女生你好" else default)  
}  
'男'.*method23*("未知人类")  
'A'.*method23*("未知人类")

## 输出（返回类型）

fun t01() { *println*(1) } // 默认Unit，除非我们指定类型  
fun t02() {4646.5f} // 默认Unit  
fun t03() {true} // 默认Unit  
fun t04() : String {return "Derry"} // 指定String  
fun t05() : Int { return 99} // 指定Int  
  
fun s01() = **{}** // () -> Unit 函数返回一个函数  
fun s02() = **{** *println*("OK")**}** // () -> Unit 函数返回一个函数  
s02()()  
  
fun s03() : Char = *run* **{** 'A' **}** // Char run返回 {}里面的函数返回类型  
fun s04() : () -> Boolean = **{** true **}** // () -> Boolean 函数返回一个函数  
*println*(s03())  
*println*(s04()())

## 函数返回类型

// k01返回的是什么类型? 答： 函数返回一个函数 (Int) -> Unit  
fun k01() = **{**n1: Int **->** *println*("你输入的类型是int?${if (n1 is Int) "你是Int类型 $n1" else "你不是Int类型"}")**}**k01()(89)  
  
// k02返回的是什么类型? 答：(Int, Int) -> Char  
fun k02() : (Int, Int) -> Char = **{** n1: Int, n2:Int **->** *println*("两数相加:${n1+n2}")  
 true  
 'A'  
**}**// 划重点，想写多行直接敲回车，为什么要换行：为了编译器检查  
  
// *TODO 第二节课直播讲的内容，继续Lambda （以上节课基础 为根基的）*val methodX1 : (String) -> Int = fun(str) : Int {  
 return str.length // 不是我们之前学习的Lambda体{}, 这个是方法体了，需要写return  
}  
// 一般开发不会用 fun关键字 + 声明处  
  
val methodX2 = **{**str: String **->** str.length**}**val methodX2s : (String) -> Int = **{ it**.length **}**// 哪种写法比较通用点? = {str : String -> str.length} 因为可以偷懒  
  
// Function1<String, Int> 等价于 (String) -> Int  
val methodX3 : Function1<String, Int> = **{**str: String **->** str.length**}***println*(methodX3("Derry"))  
  
val methodX4 : Function2<Boolean, String, Unit> = **{**isOK: Boolean, str: String **->** *println*("isOK:$isOK")**}**methodX4(true, "DerryOK")  
  
// 一般是 Lambda + 函数 == 高阶函数  
  
// fun aa{} 与 var aa2={}有啥区别?  
fun aa() {}  
val aa2 = **{}**// aa就是一个函数，实打实的函数  
// aa2是接收一个匿名函数的变量而已，这个变量 可以执行这个匿名函数  
// 共同点：他们的现象是一样的，所以有同学就纠结了  
val aa3 = aa2 // 都属于函数引用的概念  
val aa4 = ::aa // ::的含义 就是把这个实打实的函数，变成 函数引用 所以可以赋值传递 给另外一个变量  
  
// 前面为什么用val方法不应该是fun吗？  
// 我们一直在写匿名函数，把匿名函数给 val method01 变量  
// KT函数是一等公民  
  
// 全栈语言： 服务器，iOS，Android，Windows，JS，Gradle Groovy build.gradle.kts ...  
// native层  
  
// 函数中的函数，就是高阶函数， 函数如果有Lambda就属于高阶函数  
  
// KT中的 基本上都是表达式 包括 if 可以灵活返回  
// Java中，基本上都是 语句 包括 if 执行体 不可以返回  
  
var a = if (true) 111 else 222

## 返回函数的函数

fun onDelegateBase(v : View) {  
 // （int，int）->String这个应该是返回值类型声明吧  
 // 考试：函数返回一个函数  
 val fun4 = fun(n1aaa: Int, n2aaa: Int) // fun4函数本身  
  
 : (Int, Int) -> String // fun4函数的函数 以返回值返回了这个函数  
  
 = **{**n1, n2 **->** "两个数相加:${n1 + n2} n1aaa:$n1aaa" **}** // fun4函数的函数 以返回值返回了这个函数 的 实现  
  
 *println*(fun4(100, 100)(1000, 1000))  
  
 // 考试：你掌握了，你的lambda基本上每一对手了  
 val k01 : (String) -> (String) -> (Boolean) -> (Int) -> (String) -> Int =  
 **{** it: String **->  
 {** it: String **->  
 {** it: Boolean **->  
 {** it: Int **->  
 {** it: String **->** 99  
 **}  
 }  
 }  
 }  
 }** *println*(k01("AAA")("BBB")(true)(888)("Derry"))  
   
}

# 高阶函数

高阶函数：函数中有lambda就是属于高阶函数， 函数的函数就是高阶函数

// show01 返回的是什么类型？ String 因为Lambda是返回String  
fun show01(number: Int, lambda : (Int) -> String) : String = lambda.invoke(number)  
  
// 需求：三数相乘 三数相加 三数相除 由用户指定，我只定义规则（高阶函数）  
fun show02(number1: Int, number2: Int, number3: Int, lambda: (Int, Int, Int) -> Unit) =  
 lambda(number1, number2, number3) // 调用Lambda而已

fun onDelegateBase(v : View) {  
 var r : String = *show01*(99) **{** /\*it: Int ->\*/  
 "Str it:$**it**"  
 **}** *println*(r)  
  
 // 三数相乘  
 *show02*(1, 2, 3) **{** n1, n2, n3 **->** *println*(n1 \* n2 \* n3)  
 **}** // 三数相加  
 *show02*(1, 2, 300) **{** n1, n2, n3 **->** *println*(n1 + n2 + n3)  
 **}** // 三数相除  
 *show02*(100, 2, 2) **{** n1, n2, n3 **->** *println*(n1 / n2 / n3)  
 **}**}

## 高阶函数登录的例子

// 登录功能 高阶函数完成  
private fun loginEngine(userName: String, userPwd: String) {  
 // 使用高阶函数  
 *loginCheck*(userName, userPwd) **{** name, pwd **->** // 例如：服务器发起 request  
 if (name == "Derry" && pwd == "123456") {  
 *println*("恭喜${name}登录成功")  
 } else {  
 *println*("不恭喜${name}登录失败")  
 }  
 **}**}  
  
// 高阶函数 规则  
private fun loginCheck(userName: String, userPwd: String, checkResult: (String, String) -> Unit) {  
 if (userName.*isEmpty*() || userPwd.*isEmpty*()) {  
 return  
 }  
  
 if (userName == null || userPwd == null) {  
 return  
 }  
  
 // 做 name pwd 校验工作的  
 // ...  
 // 省略了 几万行代码  
  
 // 检查通过了，用户名 与 密码 OK  
 checkResult(userName, userPwd)  
}

fun onDelegateBase(v : View) {  
 *loginEngine*("Derry", "123456")  
}

## 高阶函数登录例子

private fun loginEngine(userName: String, userPwd: String, responseResult: (Boolean) -> Unit) {  
 if (userName == "Derry" && userPwd == "123456") {  
 responseResult(true)  
 } else {  
 responseResult(false)  
 }  
}

fun onDelegateBase(v : View) {  
 // 站在用户角度上  
 *loginEngine*("Derry 23", "123456") **{** if (**it**) *println*("最终登录的结果是：登录成功") else *println*("最终登录的结果是：登录失败!!")  
 **}**}

## 高阶函数和扩展函数

### 给泛型增加匿名扩展函数

给泛型增加匿名扩展函数，相当于对泛型进行扩展，泛型类型就等于this，如果泛型用作lamda表达式的参数，那么lamda表达式也会隐式持有this。

fun <T> T.myRunOK(mm: T.(Float) -> Boolean) {  
 // this == T本身 == 调用者本身 == "Derry"  
 mm(346.56f) // 调用Lambda  
}

上述例子lamda的参数有T.(Float)， 这说明之后lamda表达式会隐式持有this，这个this的类型和调用者的类型是相同的。

例如下面的例子，this类型是String

"Derry".*myRunOK* **{** true  
**}**

下面的例子this类型是Int

99.*myRunOK* **{** true  
**}**

### 给泛型添加具名拓展函数

给泛型增加具体名称扩展函数，该函数内部会持有和泛型一致的this。增加的这个函数反编译方法类型是public static，全局都可以公开访问。

fun <T> T.abc() {  
 // this == T本身 == 调用者本身 == "Derry"  
 *println*("abc我是:$this")  
}

下面的调用分别输出了对应this的类型

777.*abc*() // abc我是:777  
commonOK().*abc*() // abc我是:88  
436.564f.*abc*() // abc我是:436.564  
"sfdasf".*abc*() // abc我是:sfdasf

### 泛型拓展函数，lamda表示式参数

这里简化一下上述的实现，给泛型T添加一个拓展函数，参数是一个函数类型。

fun <T> T.derry(mm: () -> Unit) {  
 mm()  
}

调用方式：

"AAAA".*derry* **{  
  
}**

上述的函数类型是无参，返回Unit: () -> Unit

### 泛型拓展函数，lamda表示式有参数

fun <T> T.derry2(mm: (Double) -> Unit) {  
 mm(547546.56)  
}

"AAAA".*derry2* **{** // it == Double == 547546.56  
 *println*("derry2 it: $**it**")  
**}**

### lamda表示式匿名拓展String函数参数

// 匿名函数扩展 { 持有this == String == "李元霸" }  
fun <T> T.derry3(mm: String.(Double) -> Unit) {  
 "李元霸".mm(547546.56)  
}

"AAAA".*derry3* **{** // 匿名函数扩展 { 持有this == String == "李元霸" }  
 /\*this == "李元霸"  
 it == 547546.56\*/  
**}**

### lamda表示式匿名拓展String函数参数

fun <T> T.derry4(mm: T.(Double) -> Unit) {  
 // this == T本身 == this 123/A  
  
 // this.mm(547546.56) // 这个才是他的真身  
  
 mm(547546.56)  
}

123.*derry4* **{** // this == 123本身  
 *println*("derry4 我是 :$this")  
**}**

### 泛型拓展函数，lamda表示式，添加T参数

当泛型T作为lamda表达式参数时，如果并未对T实现匿名拓展函数，那么lamda表达式实现处就没有this，只有it参数。

fun <T> T.derry5(mm: (T) -> Unit) {  
 // this == T本身 == this 调用者  
 mm(this)  
}

456.*derry5* **{** // it == 456本身  
 *println*("derry5 我是 :$**it**")  
**}**

### 泛型匿名拓展函数理解

匿名拓展还是的写法：mm : T.() -> Unit。这里的意思是对T类型进行拓展还是，这个函数他的类型是()->Unit，它是匿名的，因为T.()并没有指定他的名称，这样定义之后，其实T类型里面就有了一个方法，方法类型是()->Unit。但是这个方法并没有实现，需要调用方来实现这个具体的方法，这样这个方法当然也就有this，指向调用者的具体类型。参数mm是方法名称，它用来标记改函数类型的一个变量名称。

## 高阶函数演化过程

### 单个lamda表达式

对于单个lamda表达式的高阶函数

// *TODO 1 历史演化过程： 1 2 3 4 【参数一:有默认参数，参数二:lambda输入是参数一】*fun show(name: String = "Derry", lambda: (String) -> Unit) = lambda(name)

过程1：

show("Zhangsan", lambda = **{** *println*("输出:$**it**")  
**}**)

过程2，可以直接去掉指定的lamda表达式名称

show("Zhangsan", **{** *println*("输出:$**it**")  
**}**)

过程3，lamda表达式作为最后一个参数，可以提取到函数外部

show("Zhangsan") **{** *println*("输出:$**it**")  
**}**

过程4，当只有一个输入参数时，可以直接省略，用it代替

show **{** *println*("输出:$**it**")  
**}**

### 多个lamda表达式

对于多个lamda表达式的高阶函数

// *TODO 3 多个Lambda 历史演化过程： 1 2 3 4 【参数一、二，是不同的lambda，输入为Int】*fun show2(lamba1: (Int) -> Unit, lamba2: (Int) -> Unit) {  
 lamba1(100)  
 lamba2(200)  
}

过程1

show2(lamba1 = **{** *println*("lamba1输出:$**it**")  
**}**, lamba2 = **{** *println*("lamba2输出:$**it**")  
**}**)

过程2

show2(**{** *println*("lamba1输出:$**it**")  
**}**, **{** *println*("lamba2输出:$**it**")  
**}**)

过程3

show2(**{** *println*("lamba1输出:$**it**")  
**}**) **{** *println*("lamba2输出:$**it**")  
**}**

过程4，可以直接指定输入的lamda表达式的输入，不需要限定顺序

show2(lamba2 = **{** *println*("lamba2输出:$**it**")  
**}**, lamba1 = **{** *println*("lamba1输出:$**it**")  
**}**)

### 多个函数参数，调用时指定输入的参数

对于下面的函数

// *TODO 5 写一个函数 name，age，sex，sex\_，\_age，sxe: Char*fun show3(name: String, age: Int, sex: Char, sex\_ : Char, sxe: Char) {}

可以在调用时指定参数，而不限定参数顺序

show3(sex = '女', sex\_ = '男', sxe = '女', age = 99, name = "Derry")

## 函数引用类型

高阶函数匿名函数体写法：

// *TODO 2 匿名函数体写法，规则 输入为整形，输出为字符串*showAction **{** "我的值是:$**it**"  
**}**

一般源码的做法：

// *TODO 3 一般源码的做法(具名函数，可以随意传递)*showAction(::lambdaImpl)

// *TODO 5 :: 把这个函数变成函数引用，就可以传递赋值给变量了 Function 到 (Int) 到 Int.() 最后把变量给高阶函数执行*val r1 : Function1<Int, String> = ::lambdaImpl  
val r2 : (Int) -> String = r1 // it  
val r3 : Int.() -> String = r2 // Int.() 等价于 (Int) Int.() Int.属于来源，会自动把来源作为第一个参数 this

*showAction*(r3)

// *TODO 4 一般源码的做法(具名函数，可以随意传递) 【写一个函数，输入为整形，输出为字符串】*fun lambdaImpl(value: Int) : String = "我的值是:$value"  
  
// *TODO 1 写一个高阶函数，lambda 输入为整形，输出为字符串，然后打印lamdba*fun showAction(action: (Int) -> String) {  
 *println*(action(88)) // 调用 打印Lambda  
}

上述Int.() -> String，最终会反编译成(Int) -> String，它会把Int来源默认当做第一个参数传入，所以Int.() -> String和(Int) -> String是完全等价的。

如果类型是这个：Int.(String) -> Unit，那么最终kotlin会处理成(Int, String) -> Unit

## 模拟android 点击事件

fun onDelegateBase(v : View) {  
 // 模拟安卓点击事件  
 val func = AndroidClickListener<String>()  
 func.touchListeners() // 在没有添加任何事件的时候，触发点击事件看看，相当于点击button  
  
 // 匿名函数作为事件  
 func.addListener("Derry1") **{** *println*("事件被触发了 执行了 值是:$**it**")  
 **}** func.addListener("Derry2") **{** *println*("事件被触发了 执行了 值是:$**it**")  
 **}** func.addListener("Derry3") **{** *println*("事件被触发了 执行了 值是:$**it**")  
 **}** // 触发事件  
 func.touchListeners()  
  
 *println*()  
  
 // 具名函数作为事件  
 func.addListener("Derry4", ::show1)  
  
 // 具名函数事件随意传递  
 val result : (Any ?) -> Unit = ::show1  
 func.addListener("Derry5", result)  
  
 val result2 : (Any ?) -> Unit = **{** *println*("result2 事件被触发了 执行了 值是:$**it**")  
 **}** func.addListener("Derry6", result2)  
  
 // 触发事件  
 func.touchListeners()  
  
 // Lambda 不支持泛型 ，用Any代替， Lambda不支持 默认参数  
  
 val v1 = ::a // 把a函数 实打实的函数，变成函数引用 ::  
}  
  
// *TODO 模拟安卓点击事件*class AndroidClickListener<T> {  
  
 // 定义集合1 元素类型为 Lambda 输入泛型? 输出无  
 private val actions = *arrayListOf*< (T ?) -> Unit >() // 每一个元素 都是事件 (T ?) -> Unit  
  
 // 定义集合2 元素类型为 泛型?  
 private val values = *arrayListOf*<T ?>() // 每一个元素 都是 事件需要的数据  
  
 // (T) it  
 // T.() this  
  
 // 定义addListener 参数1 是集合2的元素， 参数2 是集合1的元素， 方法内保存两个元素  
 fun addListener(value: T?, action: (T ?) -> Unit) {  
 actions += action // add 添加用户自定义的事件 到 集合  
 values += value // add 添加用户自定义的数据 到 集合  
 }  
  
 // 定义 模拟点击事件 点击 touchListeners，先判断集合1 有没有事件，在执行集合1里面所有元素  
 fun touchListeners() {  
 if (actions.isEmpty()) {  
 *println*("你还没有添加任何事件呀")  
 return  
 }  
  
 actions.*forEachIndexed* **{** index : Int, action : (T ?) -> Unit **->** action.invoke(values[index]) // 执行所有元素 == lambda规则 == 用户自定义事件体  
 **}** }  
}  
  
// 定义具名函数作为事件  
// action: (T ?) -> Unit  
fun <T> show1(value : T ?) : Unit {  
 *println*("show1 事件被触发了 执行了 值是:$value")  
}  
  
fun a() {}

## 实现RxJava链式调用

实现模拟RxJava链式调用的功能，下面分为几个版本的具体实现

### 基础版本

使用中转站就是为了转化（保存）我们的create 和 map 等等的数据，高阶函数内部持有隐式it参数

*/\*\*  
 \* 测试1  
 \*/*fun onDelegateBase(v : View) {  
 create **{** // 没有输入，意味着，没有it 没有this  
 "Derry"  
 64654  
 6546.65  
 'A'  
 6466.5f // 最后一行作为输出 输出给下一个环节 map  
 **}**.map **{** it : Float -> // it == 6466.5f  
 123  
 **}**.map **{** it: Int ->  
 true  
 **}**.map **{** it: Boolean ->  
 65466.6  
 **}**.consumer **{** it: Double ->  
 *println*("消费:$it") // 没有输出，为什么，因为直接消费就完成了  
 **}** create **{** "Derry"  
 **}**.map **{  
 it**.length  
 **}**.map **{** "内容的长度是:$**it**"  
 **}**.map **{** "【$**it**】"  
 **}**.consumer **{** *println*("消费:$**it**")  
 **}**}  
  
// 中转站目的： 保存最新的item 给map输入用 给map输出保存到item  
  
// 中转站 保存你要流向下去的数据  
class Helper<T>(private var item: T) // var item: T 这个item是一直在更新的  
{  
 // item == 6466.5f  
 // item == 123  
  
 // map可以链式调用  
 fun<R> map(action: (T) -> R) : Helper<R>  
 {  
 val newItem: R = action(item)  
 // newItem == 123  
 return Helper<R>(newItem)  
 }  
  
 // 思路：消费 只需要 上一个操作符的返回类型就可以了，我不需要返回，所以不会设计R  
 fun consumer(action: (T) -> Unit) = action(item)  
}  
  
// 思路：输入你不需要考虑，因为是最后一行作为流向输出的数据  
// 思路：中转站就是为了转化（保存）我们的create 和 map 等等的数据  
fun <R> create(action: () -> R) : Helper<R>  
{  
 val r: R = action() // r == 6466.5f  
 return Helper<R>(r)  
}

### 版本2

高阶函数内部持有this指针

*/\*\*  
 \* 测试1  
 \*/*fun onDelegateBase(v : View) {  
 create **{** // 没有输入，意味着，没有it 没有this  
 "Derry"  
 64654  
 6546.65  
 'A'  
 6466.5f // 最后一行作为输出 输出给下一个环节 map  
 **}**.map **{** 123  
 **}**.map **{** true  
 **}**.map **{** 65466.6  
 **}**.consumer **{** *println*("消费:$this") // 没有输出，为什么，因为直接消费就完成了  
 **}** create **{** "Derry"  
 **}**.map **{** length  
 **}**.map **{** "内容的长度是:$this"  
 **}**.map **{** "【$this】"  
 **}**.consumer **{** *println*("消费:$this")  
 **}**}  
  
// 中转站目的： 保存最新的item 给map输入用 给map输出保存到item  
// 中转站 保存你要流向下去的数据  
class Helper<T>(private var item: T) // var item: T 这个item是一直在更新的  
{  
 // item == 6466.5f  
 // item == 123  
  
 // map可以链式调用  
 fun<R> map(action: T.() -> R) : Helper<R>  
 {  
 val newItem: R = action(item)  
 // newItem == 123  
 return Helper<R>(newItem)  
 }  
  
 // 思路：消费 只需要 上一个操作符的返回类型就可以了，我不需要返回，所以不会设计R  
 fun consumer(action: T.() -> Unit) = action(item)  
}  
  
// 思路：输入你不需要考虑，因为是最后一行作为流向输出的数据  
// 思路：中转站就是为了转化（保存）我们的create 和 map 等等的数据  
fun <R> create(action: () -> R) : Helper<R>  
{  
 val r: R = action() // r == 6466.5f  
 return Helper<R>(r)  
}

### 简化版本1

上面实现的一个简化版本

*/\*\*  
 \* 测试1  
 \*/*fun onDelegateBase(v : View) {  
 create **{** "Derry"  
 **}**.map **{** length  
 **}**.map **{** "内容的长度是:$this"  
 **}**.map **{** "【$this】"  
 **}**.consumer **{** *println*("消费:$this")  
 **}**}  
  
class Helper<T>(private var item: T) {  
 fun<R> map(action: T.() -> R) = Helper(action(item))  
 fun consumer(action: T.() -> Unit) = action(item)  
}  
fun <R> create(action: () -> R) = Helper(action())

### 简化版本2

说明

fun <R> create(action: () -> R) = Helper(action())

create函数返回Helper对象，该对象传入的是action的返回值

class Helper<T>(var item: T)

定义个Helper的类

fun<T, R> Helper<T>.map(action: T.() -> R) = Helper(action(item))

对Helper类定义map的扩展函数，该函数接收函数类型，输入类型是T，返回类型是R。因为是对Helper的扩展函数，所以map方法可以直接访问item，这个item就是上一步create方法产生的Helper时传入的参数。该函数类型有返回值，返回值作为下一个Helper生成时的参数。

fun<T> Helper<T>.consumer(action: T.() -> Unit) = action(item)

该函数只有输入类型，没有输出，它接收上一步map高阶函数的返回值作为参数，在该函数内部消耗时间，所以没有返回值。

*/\*\*  
 \* 测试1  
 \*/*fun onDelegateBase(v : View) {  
 create **{** "Derry"  
 **}**.*map* **{** length  
 **}**.*map* **{** "内容的长度是:$this"  
 **}**.*map* **{** "【$this】"  
 **}**.*consumer* **{** *println*("消费:$this")  
 **}**}  
  
class Helper<T>(var item: T)  
fun<T, R> Helper<T>.map(action: T.() -> R) = Helper(action(item))  
fun<T> Helper<T>.consumer(action: T.() -> Unit) = action(item)  
fun <R> create(action: () -> R) = Helper(action())

### 简化版本3

*/\*\*  
 \* 测试1  
 \*/*fun onDelegateBase(v : View) {  
 create **{** "Derry"  
 **}**.*map* **{** length  
 **}**.*map* **{** "内容的长度是:$this"  
 **}**.*map* **{** "【$this】"  
 **}**.*consumer* **{** // 不需要返回，直接消费，就不需要 R，所以是 Unit  
 *println*("消费:$this")  
 **}**}  
  
fun <R> create(action: () -> R) : R = action() // 把 "Derry"保存到R泛型里面去  
fun<T, R> T.map(action: T.() -> R) : R = action(this) // this == "Derry" R=length 5  
fun<T> T.consumer(action: T.() -> Unit) = action(this) // this == length 5

上面是使用对泛型扩展进行实现的，具体的步骤如下：

1. create函数返回”Derry“字符串，由于map已经对T进行拓展，所以这里返回的字符串可以调用map方法
2. 调用map方法时，action函数持有调用者的this变量，所以map内部可以直接使用this，并且action有返回值，返回值也是字符串，可以继续调用map方法。
3. 上一步map返回的结果最后调用consumer方法， consumer的action函数也持有this，所以可以直接进行输出。

## 实现apply方法

官方提供给我们使用的apply内置函数是为了实现链式调用，虽然它有返回值，但是不提倡使用变量接收它的返回值。类似下面这样：

// 错误的用法  
val r2 : File = File("D:\\a.txt").*apply* **{** /\*this.\*/*readLines*()  
 'A'  
 true // 无法以最后一行作为返回值  
**}**.*apply* **{** setWritable(true)  
**}**.*apply* **{** setReadable(true)  
**}**.*apply* **{** // ...  
**}**

实现自定义的apply函数

*/\*\*  
 \* 测试1  
 \*/*fun onDelegateBase(v : View) {  
 // *TODO 我们自己写的* "Derry".*mApply* **{** ""  
 true  
 9  
 *println*(this)  
 **}**.*mApply* **{ }**.*mApply* **{ }**.*mApply* **{ }** val r1s : String = "Derry".*mApply* **{  
  
 }**.*mApply* **{  
  
 }**.*mApply* **{  
  
 }** val r2s: Int = 123.*mApply* **{  
  
 }** val r3: Boolean = true.*mApply* **{  
  
 }**.*mApply* **{  
  
 }**}  
  
private inline fun <I> I.mApply(lambda: I.() -> Unit) : I {  
 // lambda(this) // 他的真身  
 lambda() // 调用lambda // this可以省略的  
  
 return this // this == I == 调用者本身  
}

上述实现自定义apply函数需要注意点

1. 只要是高阶函数，必须用inline修饰，为什么，因为内部会做lambda优化

研究下高阶函数使用inline和不使用inline时的性能问题分析

1. I.mApply对泛型进行函数扩展，所有类型都可以使用mApply方法
2. I.() 对泛型进行匿名函数拓展，lamda实现处持有this
3. 返回值I，返回调用者本身，可以实现链式调用
4. lambda()，调用lambda表达式

## Kotlin函数式编程

合并2个list为一个map

zip(): 合并两个集合

map(): 变换每一个元素

*/\*\*  
 \* 测试1  
 \*/*fun onTest1(v : View) {  
 // 1.定义name集合  
 val names = *listOf*("Zhangsan", "Lisi", "Wangwu")  
 // 2.定义age集合  
 val ages = *listOf*(20, 21, 22)  
  
 names.*zip*(ages).*toMap*().*map* **{** "you name:${**it**.key}, you age:${**it**.value}" **}**.*map* **{** *println*(**it**) **}** // 简化  
 names.*zip*(ages).*toMap*().*map* **{** *println*("you name:${**it**.key}, you age:${**it**.value}") **}**}  
  
*/\*\*  
 \* 测试1  
 \*/*fun onTest2(v : View) {  
 *listOf*("Zhangsan", "Lisi", "Wangwu").*zip*(*listOf*(20, 21, 22)).*toMap*().*map* **{** *println*("you name:${**it**.key}, you age:${**it**.value}") **}**}

## 自定义forEach循环

fun onDelegateBase(v : View) {  
 // *TODO 官方的  
 listOf*("AAA", "BBB", "CCC")  
 .*forEach* **{** *println*(**it**)  
 true // 无返回  
 **}** *println*()  
  
 // *TODO 我们的  
 listOf*("AAA", "BBB", "CCC")  
 .*mForEach* **{** *println*(**it**)  
 true // 无返回  
 **}** *println*()  
  
 *listOf*("AAA", "BBB", "CCC")  
 .*mForEach2* **{** *println*(**it**)  
 true // 无返回  
 **}** *println*()  
  
 *listOf*(111, 222, 333)  
 .*mForEach3* **{** *println*(this)  
 true // 无返回  
 **}** *println*()  
  
 *listOf*(555, 666, 777)  
 .*mForEach4* **{** *println*(this)  
 true // 无返回  
 **}**}  
  
// Set List Map .... 父类 Iterable<E>  
private inline fun<E> Iterable<E>.mForEach(lambda: (E) -> Unit) {  
 // this == Iterable<E> == 调用者本身 List Set Map  
 for(item: E in this) lambda(item)  
}  
  
// run 就是执行你的逻辑 run{你的逻辑}  
private inline fun<T> Iterable<T>.mForEach2(lambda: (T) -> Unit) = *run* **{** for(item in this) lambda(item) **}**private inline fun<T> Iterable<T>.mForEach3(lambda: T.() -> Unit) = *run* **{** for(item in this) lambda(item) **}**private inline fun<T> Iterable<T>.mForEach4(lambda: T.() -> Unit) = r **{** for(item in this) lambda(item) **}**private inline fun<T> Iterable<T>.mForEach5(lambda: T.() -> Unit) =  
 r **{** for(item in this) lambda(item) **}**private inline fun <R> r(lambda : () -> R) = lambda()

## 内联inline

使用inline和不使用inline的区别：

使用inline编译器会做优化，相当于把代码拷贝到调用处

不使用inline内部会开辟对象，造成内存浪费，对性能有影响。

*/\*\*  
 \* 测试1  
 \*/*fun onDelegateBase(v : View) {  
 // 不加inline的效果  
 // show((Function0)null.INSTANCE); 对象开辟的浪费  
  
 // 加inline的效果  
 /\*String var2 = "AAA";  
 System.out.println(var2);\*/  
  
 show **{** *println*("AAA")  
 **}** // 相当于 C++ 宏替换 文本替换 inline会做优化，不需要你开辟空间浪费  
}  
  
// 高阶函数，尽量加inline  
inline fun show(lambda: () -> Unit) {  
 lambda()  
}

## 高阶函数总结

以下方法返回类型：(String, Int) -> String

fun study01() /\*: (String, Int) -> String\*/ = **{**name: String, age:Int **->** true  
 name  
 age  
 "我的姓名是:$name, 我的年龄是:$age" // 最后一行作为返回值 String  
**}**

以下方法返回类型：((Int, Int) -> String, String) -> Unit

// 函数返回函数，这个函数其实就是高阶函数  
// study02函数返回是什么类型？ ((Int, Int) -> String, String) -> Unit  
fun study02() = **{** lambdaAction: (Int, Int) -> String, studyInfo: String **->** false  
 '男'  
 val lambdaAction = lambdaAction(10, 20)  
 *println*("最后我组装的值是:$lambdaAction + $studyInfo") // 最后一行作为返回值 Unit  
**}**

以下方法返回类型：(String, Int) -> Boolean

// study033函数返回是什么类型？ (String, Int) -> Boolean  
fun study033() = **{**str: String, num: Int **->** true // 最后一行作为返回值，因为属于lambda范畴  
**}**

以下方法返回类型：(String, Int, (String) -> Unit, (Int) -> Unit) -> Unit

// study04函数返回是什么类型？ (String, Int, (String) -> Unit, (Int) -> Unit) -> Unit  
fun study04() = **{**str: String, num:Int, lambda1: (String) -> Unit, lambda2: (Int) -> Unit **->** lambda1(str)  
 lambda2(num)  
**}**

以下方法返回类型：(T1, T2, (T1) -> R1, (T2) -> R2) -> Unit

// study05函数返回是什么类型？(T1, T2, (T1) -> R1, (T2) -> R2) -> Unit  
fun <T1, T2, R1, R2> study05() = **{**str: T1, num: T2, lambda1 : (T1) -> R1, lambda2 : (T2) -> R2 **->** *println*("第一个lambda:${lambda1(str)} ${if (lambda1(str) is Int ) "你变换后是Int 值:${lambda1(str)}" else "你变换后不是Int"}")  
 *println*("第一个lambda:${lambda2(num)}")  
**}**

以下方法返回类型：(String, Int, (Int) -> Unit) -> Boolean

// study06函数返回是什么类型？ (String, Int, (Int) -> Unit) -> Boolean  
fun study06() : (String, Int, (Int) -> Unit ) -> Boolean = fun(str: String, num: Int, lambda : (Int) -> Unit) : Boolean {  
 str  
 num  
 return true // fun{} 此{} 是函数体，不是我们前面学习的那种lambda体， fun {} 是函数本身，不会以 最后一行变化二变化， 必须明确指定类型，否则默认就是 Unit  
}

以下方法返回类型：(String, Int, (Int) -> Int) -> Int

// study06s函数返回是什么类型？ (String, Int, (Int) -> Int) -> Int  
fun study06s() = **{** str: String, num: Int, lambda:(Int) -> Int **->** // -> 等价于理解 fun fun【1.明确指定类型替换默认隐式Unit，2.return 99】  
 str  
 num  
 99 // 最后一行作为返回值，因为属于lambda范畴  
**}**

## 手写内置函数

### Let函数

该函数的作用是：数默认当前对象作为闭包的it参数，并以最后一行代码作为返回值

fun onDelegateBase(v : View) {  
 // *TODO 官方的：* val r : Float = "Derry".*let* **{  
 it** // it == 调用者本身 == "Derry" 输入  
  
 ""  
 true  
 9  
 4354.545f // lambda最后一行作为返回值 Float 输出  
 **}** *println*(r)  
  
 // 我们自己写一套  
 val r2 : Boolean = "Derry".*mLet* **{** *println*("我的值是：$**it**") // 输入  
  
 false // lambda最后一行作为返回值 Boolean 输出  
 **}** *println*(r2)  
}  
  
// 特点： let 与 run 区别只有一点 let"(I)" {持有 it} run"I.()" {持有 this}  
// 1. I.mLet 万能类型.mLet 所有类型都可以调用 mLet函数  
// 2. -> O lambda里面最后一行是true，那么就是Boolean的返回  
// 3. (I) 让lambda持有it  
private inline fun <I, O> I.mLet(lambda: (I) -> O) : O = lambda(this) // I.xxx { this == I本身 }

### repeat函数

该函数的作用是：重复执行n次闭包内的代码

fun onDelegateBase(v : View) {  
 // *TODO 官方的：  
 repeat*(10) **{** *print*("下标是:$**it** ")  
 **}** *println*()  
  
 // *TODO 我们自己手写* mRepeat(10) **{** *print*("下标是:$**it** ")  
 **}** *println*()  
  
 // 我们自己扩展的，比官方的强大  
 10.*mRepeat2* **{** *print*("下标是:$**it** ")  
 **}**}  
  
private inline fun mRepeat(count: Int, action: (Int) -> Unit) {  
 for (item: Int in 0 *until* count) action(item)  
}  
  
private inline fun Int.mRepeat2(action: (Int) -> Unit) {  
 for (item: Int in 0 *until* this) action(item)  
}

### takeIf

该函数的作用是：如果takeIf里面返回true，则返回调用者自身this，否则返回null

fun onDelegateBase(v : View) {  
 // *TODO 官方的： true 返回 Derry， false会返回一个null* val r = "Derry".*takeIf* **{  
 it** // it == 调用者本身 == "Derry" 输入  
  
 false // false会返回一个null  
 true // true 返回 Derry 调用者本身  
 **}** *println*(r)  
  
 // *TODO 我们自己写一套* val r2 = "DDD".*mTakeIf* **{** true // it == 调用者本身 == "DDD" 输入  
 false  
 true  
 **}** *println*(r2)  
}  
  
// I.xxx { this == I == 调用者本身 }  
// mTakeIf函数返回什么类型？ I ?  
private inline fun <I> I.mTakeIf(action: (I) -> Boolean) =  
 if (action(this)) this else null

### thread

该函数的作用是：启动一个线程

fun onDelegateBase(v : View) {  
 // *TODO 官方的：* val r : Thread = *thread*(start = false) **{** *println*("我是异步线程 ${Thread.currentThread().*name*}")  
 **}** // 等会儿在启动，返回Thread的目的是为了下面这个  
 r.start()  
  
 // *TODO 自己的* mThread **{** *println*("我是异步线程 ${Thread.currentThread().*name*}")  
 **}**}  
  
private /\*inline\*/ fun mThread(  
 start: Boolean = true,  
 name: String ? = null,  
 /\*crossinline\*/ runAction: () -> Unit // crossinline 你的这个lambda不要给我内联 优化  
) : Thread {  
 val thread = object: Thread() {  
 override fun run() {  
 super.run()  
 runAction() // 在子线程里面调用lambda  
 }  
 }  
  
 if (start) thread.start()  
  
 // if (name != null) thread.name = name // 你还停留在Java的思维上  
 name ?. *let* **{** thread.*name* = **it }** // KT的写法  
  
 return thread  
}

# DSL

DSL(Domain Special Language): 领域特定语言

## 简单例子

模拟实现下面的功能

*layout* **{** *button* **{  
  
 }** *text* **{  
  
 }  
}**

class Helper // 中转站  
fun Helper.button(action: () -> Unit) { }  
fun Helper.text(action: () -> Unit) { }  
  
  
// 思路一：一旦你看到{} 这个就属于 lambda体，我们就需要为lambda实现体 定义lambda的规则  
// 思路二：{}里面有 button，text 我们需要让此lambda持有一个this==中转站，此this能够直接拿到 中转站（button， text）  
fun layout(action: Helper.() -> Unit) { }

## 模拟AndroidManifest.xml的解析

这个例子太复杂，不贴代码了，可以看demo里面有代码

## 简单例子泛型实现

使用泛型实现以下功能：

*layout* **{** *button* **{  
  
 }** *text* **{  
  
 }  
}**

具体代码实现：

fun onDelegateBase(v : View) {  
 // 1.{ 持有this == 泛型T （button， text） } 难题(T 持有到 button， text) ？ 攻克了难题  
 val r: String ? = layout<String> **{** // 3.由于你增加了一个泛型具体类型 String 给String增加了 两个函数 button text  
 this.*button* **{ }** this.*button* **{ }** // 省略this的写法  
 *button* **{}** *text* **{}  
 }** r.*button* **{ }** r.*text* **{ }** // *TODO ==============* layout<Double> **{** // this == Double == Double 给Double增加了 两个函数 button text  
 this.*button* **{ }** this.*text* **{ }  
 }**}  
  
fun<T> T.button(action: () -> Unit) { }  
fun <T> T.text(action: () -> Unit) { }  
// 2.此时 T 就已经 被扩展出来 两个函数 button text，意味着 这个T泛型 里面有 button text  
fun<T> layout(action: T.() -> Unit) : T ? { return null}

这种方式使用需要指定泛型的类型：

layout<Double> **{  
  
}**

因为layout函数指定了泛型类型，使用时需要对泛型进行实例化，接下来的例子实现不需要指定具体类型的layout方法。

## 简单例子泛型实现

实现上述例子中不需要指定具体类型的功能。

fun onDelegateBase(v : View) {  
 val r :Unit = layout **{** // this == Int == Int 都增加 button text 的 扩展函数  
  
 // 下面的代码，为什么可以点出来， Unit也能点  
 *button* **{ }**.*text* **{ }**.*button* **{ }** *text* **{ }**.*button* **{ }**.*text* **{ }  
  
 }**.*button* **{ }**.*text* **{ }**.*button* **{ }** // Unit也能点  
 r.*button* **{ }** r.*text* **{ }**}  
  
fun <T> T.button(action: () -> Unit) : Unit { }  
fun <T> T.text(action: () -> Unit) : Unit { }  
  
// 给 （任何类型，所有的类型），都增加 button text 的 扩展函数  
// fun layout(action: Any?.() -> Unit) : Unit { }  
// fun layout(action: Double.() -> Unit) : Unit { }  
fun layout(action: Int.() -> Unit) : Unit { } // Unit （button text）

下面2行代码实际上就已经给所有类型添加上了button和text的拓展函数，包括常用类型和Unit类型，所以即使返回Unit类型也能链式使用button和text方法。

## DSL实现Html语法规则

使用kotlin实现一套简单的Html规则，生成对应的html文件，能够在浏览器打开。代码太长，参考源码的DslHtmlActivity5。

# 协程

## 传统方式完成异步任务网络加载

亲参考demo源码

## 协程方式完成异步任务网络加载

GlobalScope 全局的，一般不会释放，所以基本不要使用

MainScope 需要自己取消

LifecycleScope 你不需要关心取消，他自动绑定Lifecycle生命周期了

ViewModelScope 你不需要关心取消，他自动绑定ViewModel生命周期了

核心代码如下：

main.launch(Dispatchers.Main) **{** // UI 线程  
 val result =  
 APIClient.instance.instanceRetrofit(WanAndroidAPI::class.*java*)  
 .loginActionCoroutine("Derry-vip", "123456") // 1.挂起出去执行异步操作 2.操作完成后恢复主线程  
  
 // 更新UI 因为这个括号里面是主线程，当然可以更新UI  
 Log.d(TAG, "errorMsg: ${result.data}")  
 textView.*text* = result.data.toString() // 更新控件 UI  
  
 mProgressDialog?.dismiss() // 隐藏加载框  
**}**

main.launch()协程提默认启动的是异步线程，所以需要添加标记Dispatchers.Main切换到主线程。当执行到以下代码时：

val result =  
 APIClient.instance.instanceRetrofit(WanAndroidAPI::class.*java*)  
 .loginActionCoroutine("Derry-vip", "123456") // 1.挂起出去执行异步操作 2.操作完成后恢复主线程

当前主线程会先切换到异步线程，执行网络操作并挂起，操作完成后切换到主线程，然后程序继续向下执行。

## 模拟多个访问请求串行访问

这里的意思就是假设需要请求A，B，C3个接口，请求完A接口返回折后再请求B接口，请求B接口返回之后再请求C接口，参考demo源码。

## 模拟多个访问请求串行访问（协程实现）

上面一步实现的功能可以使用协程进行实现，代码如下：

GlobalScope.*launch*(Dispatchers.Main) **{** // *TODO 先执行 异步请求1* var serverResponseInfo = requestLoadUser()  
 textView.*text* = serverResponseInfo // 更新UI  
 textView.setTextColor(Color.*GREEN*) // 更新UI  
  
 // *TODO 更新UI完成后，再去执行 异步请求2* serverResponseInfo = requestLoadUserAssets()  
 textView?.*text* = serverResponseInfo // 更新UI  
 textView?.setTextColor(Color.*BLUE*) // 更新UI  
  
 // *TODO 更新UI完成后，再去执行 异步请求3* serverResponseInfo = requestLoadUserAssetsDetails()  
 textView?.*text* = serverResponseInfo // 更新UI  
 mProgressDialog?.dismiss() // 更新UI  
 textView?.setTextColor(Color.*RED*) // 更新UI  
**}**

## JetPack&MVVM&协程案例实战

参考源码

对于demo目录结构，该demo有以下几个目录

activity

viewmodel

repository

entity

api

其中activity通过ViewModel实现和UI的双向绑定，ViewModel通过访问repository请求相关的数据，response通过api从网络获取数据，置于为什么要再分一个reponsitory，而不统一在ViewModel进行处理，因为ViewModel只处理和UI相关的更新操作，如果统一到ViewModel处理，就违背了设计模式的单一职责。Reponsitory也可以通过其他方式获取数据，例如本地数据库，这样以后需要替换数据，只需要把reponsitory的获取数据方式切换到数据库获取就可以了，而其他地方不需要改动。

## 协程的挂起与恢复流程

协程挂起和恢复初步流程

1. 从当前协程体作用域Dispatchers.Main 协程依附的线程
2. 当执行到挂起函数时切换到 IO异步线程到，请求耗时操作
3. IO异步线程执行完成后，开始恢复到当前作用域Dispatchers.Main，切换到UI线程，执行挂起点下一步的代码，保证了挂起点代码恢复到 UI线程

## 协程挂起和恢复的执行原理

这部分的流程比较复杂，没有听懂，后面需要研究

# 泛型

## 泛型基础

泛型快速入门例子

fun onDelegateBase(v : View) {  
 // 1.完整写法  
 val myStudy : MyStudy<String> = MyStudy<String>("Derry is OK")  
 *println*(myStudy.item)  
  
 val myStudy2 : MyStudy<Boolean> = MyStudy<Boolean>(false)  
 *println*(myStudy2.item)  
  
 // 2.泛型 的 类型推断  
 val myStudy3 = MyStudy("Derry is OK") // 把T推断成 String  
 *println*(myStudy3.item)  
  
 val myStudy4 = MyStudy(true) // 把T推断成 Boolean  
 *println*(myStudy4.item)  
}  
  
// T? 可以传null，标准的写法  
class MyStudy<T>(type: T) {  
 var item: T = type  
}

## 泛型协变，逆变和不变

泛型协变：子类型限定，关键字 ? extends T，表示都是T的子类型

泛型逆变：超类型限定，关键字 ? super T，表示都是T或T的父类

泛型不变：没有协变和逆变就是不变。

如果我们只需要读取泛型数据（生成者），就可以把此泛型声明成协变 Java（? extends T） Kotlin out 协变，因为无法确定实际存放的是T的哪一个子类，所以不支持“存放”数据

如果我们只需要写入泛型数据（消费者），就可以把此泛型声明成逆变 Java（? super T） Kotlin in 逆变，因为无法确定所取得的数据具体是哪种类型，所以不能“获取”数据，

PECS原则： Producer Extends Consumer Super == 协变 与 逆变

public void onDelegateBase(View v) {  
 // *TODO ================= 协变* // Object[] = String[] 数组默认开启协变  
 // 之所以可以，是因为继承关系， 因为 默认的子父类继承关系，默认开启协变  
 String str = "";  
 Object obj;  
 obj = str; // fu = zi = 协变  
  
 List<Object> list1;  
 List<String> list2 = new ArrayList<String>();  
 // list1 = list2; // 因为List<Object> 不是 List<String>的父类， 因为这种情况下，默认禁止协变  
  
 // list1 = list2 假设是可以的，会有什么问题？  
 // list1.add(new Date()); 我存入进去的是Date Object类型  
 // String str = list2.get(0); 我取出来是 String 就会 出现类型转换不合理  
  
 List<? extends Object> lists1; // 【? extends】 能够 List<Object> 接收 List<String>， 手动开启协变  
 List<String> lists2 = new ArrayList<String>();  
 lists1 = lists2;  
  
 // 父类的引用List<父类> 接收 子类的对象List<子类> = 协变  
 // 父类 接收 子类 合理 == 协作的变化 协变  
 // 子类 接收 父类 不合理 == 逆行开车 逆向 逆变  
  
 // *TODO ================= 逆变* List<Object> list3 = new ArrayList<Object>();  
 // List<String> list2 = list1; // List<String> 无法接收 无法接纳 List<Object>， 默认禁止逆变  
  
 List<? super String> list4 = list3; // 【? super 】 能够让我们 List<String>子引用 接收 List<Object>父类型  
 // 逆变 Kotlin in  
}  
  
interface List1<E> {  
 void addAll(List1<E> items);  
}  
  
void copyAll(List1<Object> to, List1<String> from) {  
 // to = from; // 编译不通过： 因为 List1<String> 并不是 List1<Object> 的子类型  
}  
  
void copyAll2(List1<? extends Object> to, List1<String> from) { // ? extends 手动开启协变  
 to = from;  
}

## Kotlin 泛型协变和逆变

看demo

## 生产和消费

如果一个类使用out T修饰，那么它就是生产者，只能读不能写，就是泛型协变。

如果一个类使用in T修饰，那么它就是消费者，只能写不能读，就是泛型逆变。

如果一个类同时使用in 和 out 修饰，那么它就是生产者和消费者。其中in指定的类型只能用于传入参数，表示消费，out指定的类型只能用于返回，表示生产。

我们的普通代码没有 out 没有 in，默认就是不变。

## 类限定对全部方法生效

如果一个类使用了in或者out进行修饰，它对这个类中的所有方法都会生效，不需要每个方法都重新使用in或out进行定义。

## 泛型限定

使用T : Object关键字进行限定，表示T只能是Objcet或者Object的子类，参考demo。

## Kotlin注解

Klotlin 元注解：meta-annotation(元注解) 元注解描述注解的注解 @Target @Retention

下面是注解目标的类型

@Target(  
 AnnotationTarget.*CLASS*, // 作用域在类上  
 AnnotationTarget.*FUNCTION*, // 作用域在函数上  
 AnnotationTarget.*VALUE\_PARAMETER*, // 作用域在参数上  
 AnnotationTarget.*EXPRESSION*, // 作用域在表达式上  
 AnnotationTarget.*CONSTRUCTOR*, // 作用域在构造函数上  
 AnnotationTarget.*PROPERTY\_SETTER*, // 作用域在set上  
 AnnotationTarget.*PROPERTY\_GETTER* // 作用域在get上  
)

## Kotlin 反射

KT的反射，几乎目前不用，直接用KT的语法，调用Java的反射来玩

KT的反射，我个人认为，还不是很成熟，还不适合使用

Kotlin的反射：

1.第一次加载 比 Java慢， 第二次 第N次，才勉强达到Java反射性能

2.而外导入Kotlin反射包， xxxMB

3.学习成本高，全新一套内容

## 生成Html标签

参考demo

## 人员管理系统

参考demo

# 目录1

## 目录2

### 目录3

#### 目录4

##### 目录5