before

Kotlin初谈

Kotlin 是一个用于现代多平台应用的静态编程语言 ，由 JetBrains 开发。 Kotlin可以编译成Java字节码，也可以编译成JavaScript，方便在没有JVM的设备上运行。

kotlin支持面对对象和函数式两种编程风格

优点

- 简洁: 大大减少样板代码的数量。
- 安全: 避免空指针异常等整个类的错误。
- 互操作性: 充分利用 JVM、Android 和浏览器的现有库。完全兼容Java
- 工具友好: 可用任何 Java IDE 或者使用命令行构建。

啊！缺点？ 我们kt没有缺点

缺点

- 编译比较慢，自动为属性生成很多的get/set方法
- apk会变大

基本数据类型

- java把基本数据类型和引用数据类型做了区分。一个基本数据类型（如int）的变量直接储存了它的值，而一个引用类型（如String）的变量储存的是指向包含该对象的内存地址的引用因此基本数据类型的值能够更加高效的储存和传递，但是你不能够对这些值调用方法，或者是把他们加入到集合中。java提供了特殊的包装类型（比如 java.lang，
  Integer），在你需要对象的时候对基本数据英型进行葜装。因此，你不能用

```
collection<int>
```

来定义一个整数的集合，而必须用

```
Collection<Integer>
```

来定义。

- 但是Kotlin 并不区分基本数据类型和包装类型，你使用的永远是同一个类型（比如：

```
val i: Int = 1
val list: List<Int>= listof(1, 2, 3)
```

这样很方便。此外,你还能对一个数字类型的值调用方法。例如下面这段代码中,
使用了标准库的函数 coerceIn 来把值限制在特定范围内：

```
fun showProgress(progress: Int){
val percent = progress.coerceIn(0, 100)
println("We're ${percent}" done!")
>>> showProgress(146)
We're 100% done!
```

扩展

> 如果**基本数据**类型和**引用类型**是一样的，是不是意味着 Kotlin 使用对象来表示所有的数字？这样不是非常低效吗？确实低效，所以 Kotlin 并没有这样做。
>
> 在运行时，数字类型会尽可能地使用最高效的方式来表示。大多数情况下:对于变量、属性、参数和返回类型———Kotlin 的 Int 类型会被**编译**成 Java 基本数据类型 int。
>
> 唯一不可行的例外是**泛型类**，比如**集合**。用作泛型类型参数的基本数据类型会被编译成对应的 Java 包装类型。例如，Int 类型被用作集合类的类型参数时，集合类将会保存对应包装类型 java.lang.Integer 的实例。
>
> 对应到 Java 基本数据类型的类型完整列表如下：
>
> - 整数类型——Byte、Short、Int、Long
> - 浮点数类型——Float、Double
> - 字符类型——Char
> - 布尔类型——Boolean

KT的数据类型基本和java是相同的区别就是上面那些类容

基本语法

定义常量与变量

可变变量定义：var 关键字

```
var <标识符> : <类型> = <初始化值>
```

不可变变量定义：val 关键字，只能赋值一次的变量(类似Java中final修饰的变量)

```
val <标识符> : <类型> = <初始化值>
```

常量与变量都可以没有初始化值,但是在引用前必须初始化

编译器支持自动类型判断,即声明时可以不指定类型,由编译器判断。

循环

关于while 和do while 与java没有什么不同的 就不在赘述了

在Kotlin中想遍历1-100的数值可以这样写：

```
for (index in 1..100){
            print(index)
        }
```

这样写是正序遍历，如果想倒序遍历就该使用标准库中定义的 `downTo()` 函数：

```
for (index in 100 downTo 1){
            print(index)
        }
```

想不使用1作为遍历的步长，可以使用`step()`函数：

```
 for (index in 1..100 step 2){
            print(index)//会输出1..3..5......
        }
```

要创建一个不包含末尾元素的区间：

```
for (index in 1 until 10){
            println(index)//输出0..9
        }
```

遍历一个数组/列表，想同时取出下标和元素：

```
 val array = arrayOf("a", "b", "c")
        for ((index,e) in array.withIndex()){
            println("下标=$index----元素=$e")
        }

```

遍历一个数组/列表，只取出下标:

```
val array = arrayOf("a", "b", "c")
        for (index in array.indices){
            println("index=$index")//输出0，1，2
        }
```

遍历取元素：

```
val array = arrayOf("a", "b", "c")
        for (element in array){
            println("element=$element")//输出a,b,c
        }
```

迭代map

- kotlin 迭代 map 的结构： for ((a,b) in map)
- a代表的是 map 的key，b代表的是 map 的 value，a和b是变量，自己命名即可
- ..语法可以用于创建字符区间。例如 for (c in 'A'..'F')

```
fun main(args: Array<String>){
    val binaryReps = TreeMap<Char, String>()
    for (c in 'A'..'F'){//创建字符区间
        val binary = Integer.toBinaryString(c.toInt())  //将 ASCII 码转化成二进制
        binaryReps[c] = binary //根据 key 为c 把 binary 存到 map 中
    }
    for ((letter, binary) in binaryReps){//迭代 map，把 key 和 value 赋值给变量
        println("$letter = $binary")

}
```

异常

- KT并不区分受检异常和未受检异常，所以对开发者来说我们可以既可以选择处理异常亦可以选择不处理异常 并不会出现java中那种强制让人处理异常的情况
- try是可以作为表达式的 也就是说它又返回值

使用 *throw*-表达式来抛出异常：

```
throw Exception("Hi There!")
```

try 是一个表达式，即它可以有一个返回值：

这意味在某些时候可以优化我们处理异常的方式

```
val a: Int? = try {
    parseInt(input)
} catch (e: NumberFormatException)
{  null }
```

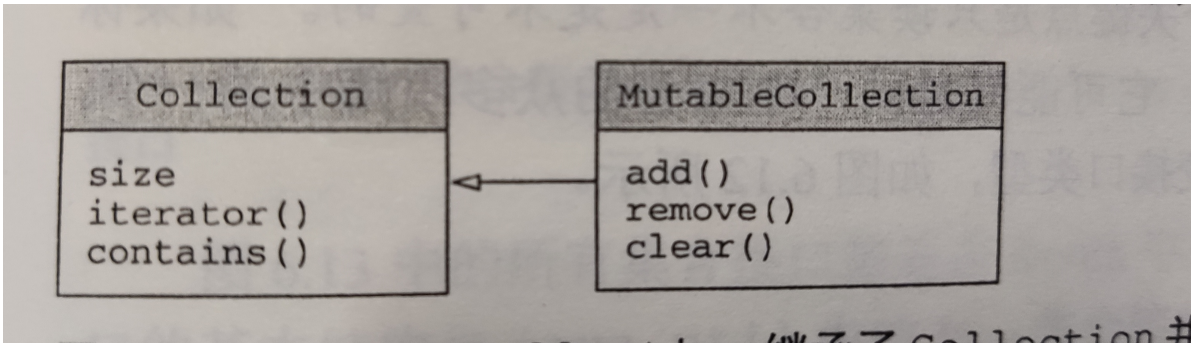
集合

kt当中并没有自己专门的集合类 它全部使用的是java的集合

- 原因是因为 这样可以更好的与java代码进行交互 而且java集合框架本身就比较优秀（个人想法）

只读集合和可变集合

- Kotlin的集合设计与Java不同的是，它把访问集合数据的接口和修改集合数据的接口分开了。这种区别存在于最基础的使用集合的接口之中，
- Kotlin.collections.Collection。使用这个接口，可以遍历集合中的元素、获取集合的大小、判断集合中是否包含某个元素，以及执行其他从该集合中读取数据的操作。但是这个接口没有任何添加或移除元素的方法。
- 使用Kotlin.collections.MutableCollections接口可以修改集合中的数据。它继承了普通的Kotlin.collections.Collection接口，还提供了方法来添加和移除元素、清空集合等。



集合类型	只读	可变
List	listOf	mutableListOf arrayListOf
Set	setOf	mutableSetOf hashSetOf linkedSetOf sortedSetOf
Map	mapOf	mutableMapOf hashMapOf linkedMapOf sortedMapOf

关于集合的各种新操作 也就是kt为集合添加的新功能 我在这里给一篇博客

集合操作符

函数

为了增强代码的可读性，Kotlin 允许我们使用命名参数，即在调用某函数的时候，可以将函数参数名一起标明，从而明确地表达该参数的含义与作用，但是在指定了一个参数的名称后，之后的所有参数都需要标明名称

命名参数

代码预览

```
fun main(args: Array<String>) {
    //错误，在指定了一个参数的名称后，之后的所有参数都需要标明名称
    //compute(index = 110, "leavesC")
    compute(index = 120, value = "leavesC")
    compute(130, value = "leavesC")

}

fun compute(index: Int, value: String) {

}
```

默认参数值

可以在声明函数的时候指定参数的默认值，从而避免创建重载的函数

代码预览

```
fun main(args: Array<String>) {
    compute(age = 24)
    compute(age = 24, name = "leavesC")
    compute(age = 24, value = 90, name = "leavesC")
    compute(value = 90, age = 24, name = "leavesC")
}

fun compute(name: String = "leavesC", age: Int, value: Int = 100) {

}
```

可变参数

代码预览

```
fun main(args: Array<String>) {
    compute()
    compute("leavesC")
    compute("leavesC", "叶应是叶")
    compute("leavesC", "叶应是叶", "叶")
}

fun compute(vararg name: String) {
    name.forEach { println(it) }
}
```

局部函数

Kotlin 支持在函数中嵌套函数，被嵌套的函数称为局部函数

代码预览

```
fun main(args: Array<String>) {
    compute("leavesC", "country")
}

fun compute(name: String, country: String) {
    fun check(string: String) {
        if (string.isEmpty()) {
            throw IllegalArgumentException("参数错误")
        }
    }
    check(name)
    check(country)
}
```

顶层函数和属性

他们出现的目的是为了消除java中的静态工具类

那么kt为什么要这门干南?

概念: 我们知道在Java中有静态函数和静态属性概念，它们一般作用就是为了提供一个全局共享访问区域和方法。我们一般的习惯的写法就是写一个类包裹一些static修饰的方法，然后在外部访问的直接利用**类名.方法名**访问。

问题: 我们都知道静态函数内部是不包含状态的，也就是所谓的纯函数，它的输入仅仅来自于它的参数列表，而它的输出也仅仅依赖于它参数列表。我们设想一下这样开发情景，有时候我们并不想利用实例对象来调用函数，所以我们一般会往静态函数容器类中添加静态函数，如此反复，这样无疑是让这个类容器膨胀。

解决: 在Kotlin中则认为一个函数或方法有时候并不是属于任何一个类，它可以独立存在。所以在Kotlin中类似静态函数和静态属性会去掉外层类的容器，一个函数或者属性可以直接定义在一个Kotlin文件的顶层中，在使用的地方只需要import这个函数或属性即可。如果你的代码还存在很多以"Util"后缀结尾的工具类，是时候去掉了。

在顶层文件中定义一个函数:

```
package com.mikyou.kotlin.top

import java.math.BigDecimal


//这个顶层函数不属于任何一个类，不需要类容器，不需要static关键字
fun add(a:Int,b:Int)=a+b

//测试顶层函数，实际上Kotlin中main函数和Java不一样，它可以不存在任何类容器中，可以直接定义在一个Kotlin 文件中
//另一方面也解释了Kotlin中的main函数不需要了static关键字，实际上它自己就是个顶层函数。
fun main(args: Array<String>) {
    println("文件大小: ${formateFileSize(15582.0)}")
}
```

实质原理

- 通过以上例子我们思考一下顶层函数在JVM中是怎么运行的，如果你仅仅是在Kotlin中使用这些顶层函数，那么可以不用细究。但是如果你是Java和Kotlin混合开发模式，那么你就有必要深入内部原理。我们都知道Kotlin和Java互操作性是很强的，所以就衍生出了一个问题:在Kotlin中定义的顶层函数。我们都知道在java中我们也是可以调用的

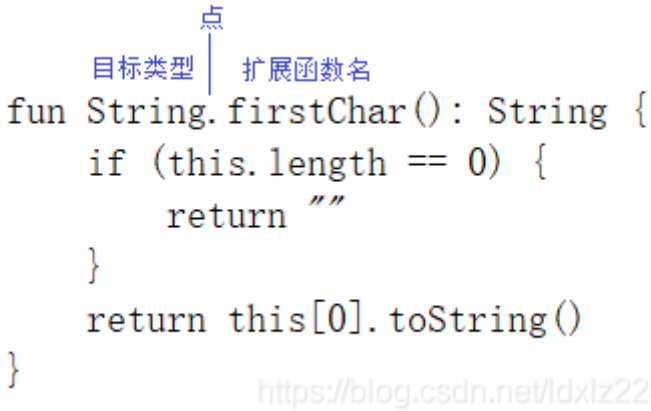
  展示

  Kotlin中的顶层函数反编译成的Java中的容器类名一般是顶层文件名+“Kt”后缀作为类名，但是也是可以自定义的。也就是说顶层文件名和生成容器类名没有必然的联系。**通过Kotlin中的@file: JvmName("自定义生成类名")注解就可以自动生成对应Java调用类名，注意需要放在文件顶部，在package声明的前面**

扩展函数与属性

到了这里我们就可以明白为什么 kt使用的是java的集合 但是却多了很多对集合的操作　其原理就是扩展函数

扩展函数语法就是



https://blog.csdn.net/ldxlz224

```kotlin
fun String.firstChar(): String {
    if (this.length == 0) {
        return ""
    }
    return this[0].toString()
}

fun String.lastChar(): String {
    if (this.length == 0) {
        return ""
    }
    return this[this.length - 1].toString()
}
```

然后在可以直接调用

```kotlin
package com.demo.kotlin


object Text {

    @JvmStatic
    fun main(args: Array<String>) {
        println("abc".firstChar())
        println("qwe".lastChar())
    }
}

//a
//e
```

为MutableList扩展一个firstElement属性

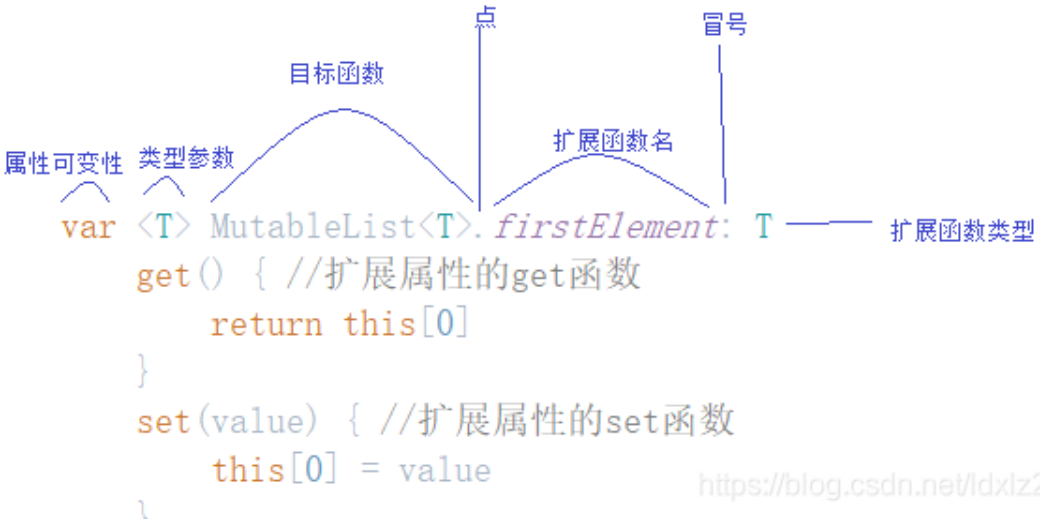
```kotlin
var <T> MutableList<T>.firstElement: T
    get() { //扩展属性的get函数
        return this[0]
    }
    set(value) { //扩展属性的set函数
        this[0] = value
    }
```

调用如下

```
object Text {
    @JvmStatic
    fun main(args: Array<String>) {
        val list = mutableListOf(1, 2, 3, 4, 5, 6, 7)
        println(list.firstElement)
        list.firstElement = 100
        println(list)
    }
}
```

输出如下

```
1
[100, 2, 3, 4, 5, 6, 7]
```



这图片写错了应该是扩展属性名

实现原理

扩展属性和扩展函数的本质是以静态导入的方式来实现的。

- 我也不是很了解 不过大佬们可以自行百度了解啦

中缀调用

可以以以下形式创建一个 Map 变量

代码预览

```
fun main(args: Array<String>) {
    val maps = mapOf(1 to "leavesC", 2 to "ye", 3 to "czy")
    maps.forEach { key, value -> println("key is : $key , value is : $value") }
}
```

使用 “to” 来声明 map 的 key 与 value 之间的对应关系，这种形式的函数调用被称为中缀调用

中缀调用可以与只有一个参数的函数一起使用，无论是普通的函数还是扩展函数。中缀符号需要通过infix修饰符来进行标记

代码预览

```
fun main(args: Array<String>) {
    val pair = 10 test "leavesC"
    val pair2 = 1.2 test 20
    println(pair2.javaClass) //class kotlin.Pair
}

infix fun Any.test(other: Any) = Pair(this, other)
```

解构声明

有时会有把一个对象解构成多个变量的需求，在 Kotlin 中这种语法称为解构声明

例如，以下例子将 Person 变量结构为了两个新变量：name 和 age，并且可以独立使用它们

代码预览

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

fun main(args: Array<String>) {
    val (name, age) = Person("leavesC", 24)
    println("Name: $name , age: $age")
    //Name: leavesC , age: 24
}
```

一个解构声明会被编译成以下代码：

代码预览

```
    val name = person.component1()
    val age = person.component2()
```

解构声明也可以用在 for 循环中

代码预览

```
    val list = listOf(Person("leavesC", 24), Person("leavesC", 25))
    for ((name, age) in list) {
        println("Name: $name , age: $age")
    }
```

遍历 map 同样适用

代码预览

```
    val map = mapOf("leavesC" to 24, "ye" to 25)
    for ((name, age) in map) {
        println("Name: $name , age: $age")
    }
```

同样也适用于 lambda 表达式

代码预览

```
    val map = mapOf("leavesC" to 24, "ye" to 25)
    map.mapKeys { (key, value) -> println("key : $key , value : $value") }
```

如果在解构声明中不需要某个变量，那么可以用下划线取代其名称，此时不会调用相应的componentN()操作符函数

代码预览

```
val map = mapOf("leavesC" to 24, "ye" to 25)
for ((_, age) in map) {
    println("age: $age")
}
```

类

构造方法

- java中的构造方法

```
/**
 * java person 类
 */
public class Person {
    String name;
    int age;

    public Person() {
    }

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```

特点：

1、方法名与类名相同

2、不定义返回值类型

Kotlin中的构造方法

Kotlin中讲构造方法独立了出来，使用关键字constructor**来表示。同时，Kotlin中将构造方法分为了两类：**主构造方法和次构造方法

主构造方法，每个类最多有1个

主构造方法在类后面声明👇

- 空参/有参主构造方法

```
/**
 * kotlin 企鹅类
 */
class Penguin constructor() {

    var name:String?=null //名称
    var age:Int=0  //年龄
    var weight:Int=100  //体重

}
```

```
/**
 * kotlin 企鹅类
 */
class Penguin constructor(name: String,age:Int) {

    var name:String?=null //名称
    var age:Int=0  //年龄
    var weight:Int=100  //体重

}
```

- constructor关键字可以省略

```kotlin
/**
 * kotlin 企鹅类
 */
class Penguin () {

    var name:String?=null //名称
    var age:Int=0  //年龄
    var weight:Int=100  //体重

}
```

```kotlin
/**
 * kotlin 企鹅类
 */
class Penguin (name: String,age:Int) {

    var name:String?=null //名称
    var age:Int=0  //年龄
    var weight:Int=100  //体重

}
```

- 如果加权限修饰符，那么放到constructor前面，此时，就不能省略constructor关键字了

```kotlin
/**
 * kotlin 企鹅类
 */
class Penguin private constructor () {

    var name:String?=null //名称
    var age:Int=0  //年龄
    var weight:Int=100  //体重

}
```

```kotlin
/**
 * kotlin 企鹅类
 */
class Penguin private constructor (name: String,age:Int) {

    var name:String?=null //名称
    var age:Int=0  //年龄
    var weight:Int=100  //体重

}
```

- 再次强调，**当有了权限修饰符时，就不能省略constructor关键字了**

```kotlin
/**
 * kotlin 企鹅类
 */
class Penguin private  () {

    var name:String?=null //名称
    var age:Int=0  //年龄
    var weight:Int=100  //体重

}
```

```kotlin
/**
 * kotlin 企鹅类
 */
class Penguin private  (name: String,age:Int) {

    var name:String?=null //名称
    var age:Int=0  //年龄
    var weight:Int=100  //体重

}
```

到这里，有关主构造方法的**声明语法**就讲完了
接下来我们会有另外一个问题，怎么用？

当使用有参构造方法时，我们怎么使用主构造方法中的参数呢？

> java中的使用

```java
/**
 * java person 类
 */
public class Person {
    String name;
    int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```

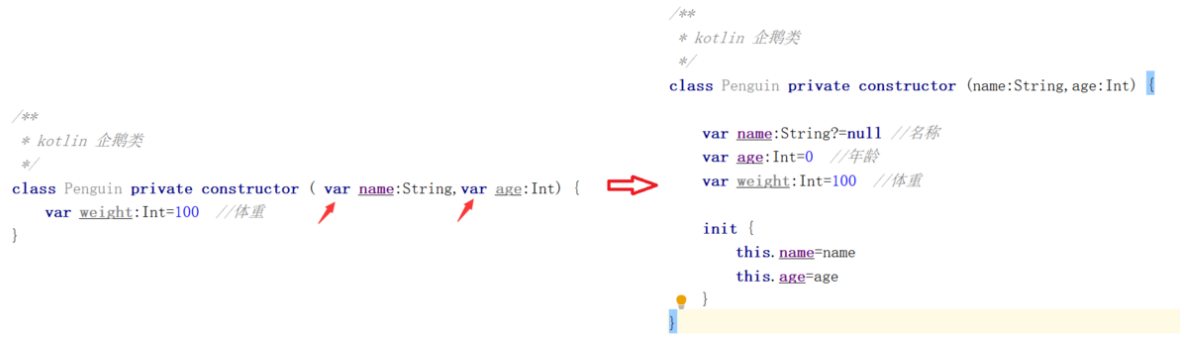
Kotlin的主构造方法是没有方法体的，这就意味着我们无法在主构造方法中进行任何操作。
但Kotlin为我们**提供了一个`init`代码块**，这个代码块的执行顺序**在主构造方法之后**，次构造方法之前，我们可以在这个代码块中进行各种初始化的操作,包括访问主构造方法中的参数👇

```kotlin
/**
* kotlin 企鹅类
*/
class Penguin private constructor (name:String,age:Int) {

   var name:String?=null //名称
   var age:Int=0  //年龄
   var weight:Int=100  //体重

   //kotlin为我们提供了一个init代码块,
   //init代码块的执行顺序在主构造方法之后
   //我们可以在init代码块中进行各种初始化操作
   //init代码块中可以访问主构造方法中的变量
   init {
       this.name=name
       this.age=age
   }
}
```

虽然可以初始化变量 但是我们是kotlin语言所以 应该有更加简单的初始化方式 这个太繁琐了

在一个类中声明属性，通过构造方法传值进行初始化这种操作有没有很频繁，有没有很繁琐，虽然IDE给我们提供了快捷键，虽然你的手速也很快，但是要是能把这个过程“省略”掉就更完美了。
Kotlin让提供了这样的功能

> 主构造方法中声明属性



左侧的写法跟右侧的写法效果完全相同。

var换成val也可以👇

```
/**
 * kotlin 企鹅类
 */
class Penguin private constructor ( val name:String,val age:Int) {
    var weight:Int=100  //体重
}
```

当主构造器和次构造器同时存在时，次构造器必须直接或者间接调用主构造器

```
/**
 * kotlin 企鹅类
 */
class Penguin constructor(name: String) {
    var weight:Int=100  //体重
    var age:Int=0
    var name:String?=null

    //空参次构造器,调用了下面的次要构造方法
    constructor():this("奔波儿霸",10)

    //有参次构造器,调用了主构造方法
    constructor(name:String,age:Int):this(name){
        this.name=name
        this.age=age
    }
}
```

构造方法调用构造方法的方式如👆代码所示，使用`:this`来调用

执行顺序上，主构造器>init代码块>次构造器

这个执行顺序还是比较重要的 在某些时候我们可能需要

修饰符

open final abstract 默认未final的

类修饰符

修饰符	说明
final	不能被继承
open	可以被继承
abstract	抽象类
enum	枚举类
data	数据类
sealed	密封类
annotation	注解类

成员修饰符

修饰符	说明
override	重写函数
open	可被重写
final	不能被重写
abstract	抽象函数
iateinit	后期初始化

访问权限修饰符

修饰符	类成员	顶层声明
public	所有地方可见	所有地方可见
internal	模块中可见	模块中可见
protected	子类中可见	
private	类中可见	文件中可见

只是这个protected 与java有区别 java是同包访问

kt只是类和子类可见

- 修饰符对于大家来说都比较简单 我们看看就可以了

嵌套类内部类

相当于`java`中的静态内部类（有`static`关键字修饰的内部类）

由于相当于Java中的静态内部类，所以不能访问到`nameStr`变量，但是可以访问另一个嵌套类

```
//嵌套类
class OutClass1{
    private var nameStr:String = " print nameStr"
    class InnerClass{
        fun innerMethod() = "inner method print"
    }
}
fun main() {
    //外部想调用InnerClass的方法，可以直接调用
    var inner1:String = OutClass1.InnerClass().innerMethod();
}
```

局部嵌套类

定义在方法中的嵌套类

```
class OuterClass3{
    fun getName(): String{
        class GetClass{
            var name = "print getName"
        }
        return GetClass().name
    }
}
```

内部类

相当于`java`中的非静态内部类（没有`static`关键字修饰的内部类）

定义好内部类后，内部类会持有一个外部类的引用

内部类需要关键字`inner`声明

内部类可以直接调用外部类的属性，需要用`this@`

```
//内部类
class OutClass2{
    private var nameStr:String = " print nameStr"
    inner class InnerClass{
        //内部类中调用外部类的
        fun innerMethod() = this@OutClass2.nameStr
    }
}
fun main() {
    //外部类调用有inner关键字声明的内部类需要创建实例去调用
    var inner2:String = OutClass2().InnerClass().innerMethod()
```

内部类变量的引用方式

- Kotlin`访问外部类变量的方式：`this@OuterClass.name
- Java`访问外部类变量的方式：`OuterClass.this.name

```
class OuterClass4 {
    val name = "name 1"
    inner class NameClass2 {
```

```
        val name = "name 2"
        fun getName() {
            val name = "name 3"
            //外部类中的name调用
            println("${this@OuterClass4.name}")      //打印 name 1
            //当前类中的name调用
            println("${this.name}")                  //打印 name 2
            //当前方法的name调用
            println("${name}")                       //打印 name 3
        }
    }
}
```

数据类

https://www.jianshu.com/p/82145082192a/

object

定义一个类并同时创建个实例(也就是一个类对象

- 使用场景
    - 对象声明为单例的一种方式
    - 伴生对象可以持有工厂方法和其他与这个类相关，但是在调用时并不依赖实例的方法。他们的成员可以通过类名来访问
    - 对象表达式用来替代java的匿名内部类

使用object创建单例模式 就不能够再次访问它的构造方法 因为这样是没有意义的

伴生对象

—工厂方法和静态成员的地盘

伴生对象里的init代码块就相当于Java中的静态代码块。在类加载的时候会优先执行且只会执行一次。

也可以实现接口

在对象声明的前面加上**companion**关键字就生成了伴生对象。作用就是为其所在的外部类**模拟静态成员。**

每个类可以最多有一个半生对象；
伴生对象的成员类似于 Java 的静态成员；
使用 const 关键字修饰常量，类似于 Java 中的 static final修饰。
可以使用 @JvmField 和 @JvmStatic 类似于 Java 中调用静态属性和静态方法；
伴生对象可以扩展属性和扩展方法。

- kt没有static关键字 所以在大多数的情况下 推荐的是去使用顶层函数的方式 来替代java中的静态工具类 但是这样也有一个缺点 那就是顶层函数并不能访问类的private成员 所以就有了伴生对象的说法 它是实现工厂模式的理想选择

语法： （ObjectName可省略）

```
companion object ObjectName : [0~N个父类型] {
    //伴生对象类体
}
```

- 每个类最多定义一个伴生对象；
- 伴生对象**相当于**外部类的对象，可以直接通过外部类名访问伴生对象的成员；
- 由于kotlin取消了static关键字，伴生对象是为了弥补kotlin没有static关键字修饰的静态成员的不足；
- 虽然伴生对象是为其所在对象模拟静态成员，但是伴生对象成员依然属于伴生对象本身的成员，而不属于其外部类的成员。

伴生对象名称可以省略，省略伴生对象名称后，如果想获取伴生对象本身，可以通过Companion获取。

```
fun main() {
    println(OuterClass.name)//伴生对象属性
    OuterClass.companionFun()//调用伴生对象方法
    OuterClass.Companion//通过Companion获取伴生对象本身
}

class OuterClass {
    companion object {
        val name = "伴生对象属性"
        fun companionFun() {
            println("调用伴生对象方法")
        }
    }
}
```

为伴生对象扩展成员，如果伴生对象有名字，则通过“**外部类.伴生对象名字.成员**”的方式扩展；

如果伴生对象没名字，则通过“**外部类.Companion.成员**”的方式扩展

```
fun main() {
    println(OuterClass.name)//伴生对象属性
    OuterClass.companionFun()//调用伴生对象方法

    println(OuterClass.extraParam)//为伴生对象扩展属性
    OuterClass.test()//为伴生对象扩展方法
}

class OuterClass {
    companion object {
        val name = "伴生对象属性"
        fun companionFun() {
            println("调用伴生对象方法")
        }
    }
}

/**
 * 为伴生对象扩展方法
 */
fun OuterClass.Companion.test() {
    println("为伴生对象扩展方法")
}
```

```
24
25 /**
26  * 为伴生对象扩展属性
27  */
28 val OuterClass.Companion.extraParam: String
29     get() = "为伴生对象扩展属性"
```

对象表达式

是针对java匿名内部类的替代品 并且增加了实现多接口的能力 和修改在创建对象的作用域中定义的变量的能力等功能

对象表达式的语法格式如下：

```
1 object [: ``0``~N个父类型]{
2  ``//对象表达式的类体部分
3 }
```

规则：

- 对象表达式不能是抽象类，因为系统在创建对象表达式时会立即创建对象。因此不允许将对象表达式定义成抽象类。
- 对象表达式不能定义构造器。但对象表达式可以定义初始化块，可以通过初始化块来完成构造器需要完成的事情。
- 对象表达式可以包含内部类，不能包含嵌套类。

```
1  interface Outputable {      //这是一个接口嘛
2    fun output(msg: String)
3  }
4
5  abstract class Product(var price: Double) {//抽象类咯
6    abstract val name: String
7    abstract fun printInfo()
8  }
9
10 fun main(args: Array<String>) {
11   //指定一个父类型（接口）的对象表达式
12   var ob1 = object : Outputable {
13     override fun output(msg: String) {
14       for (i in 1..6) {
15         println("<h${i}>${msg}</h${i}>")
16       }
17     }
18   }
19   ob1.output("随便输出点什么吧")
20   println("-----------------------------------------------")
21   //指定零个父类型的对象表达式
22   var ob2 = object {
23     //初始化块
24     init {
25       println("初始化块")
26     }
27
28     //属性
29     var name = "Kotlin"
30
```

```
    //方法
    fun test() {
      println("test方法")
    }

    //只能包含内部类，不可以包含嵌套类
    inner class Inner
  }
  println(ob2.name)
  ob2.test()
  println("------------------------------------------------")
  //指定两个父类型的对象表达式
  var ob3 = object : Outputable, Product(1.23) {
    override fun output(msg: String) {
      println("输出信息：${msg}")
    }

    override val name: String
      get() = "激光打印机"

    override fun printInfo() {
      println("高速极光打印机们支持自动双面打印！")
    }
  }
  println(ob3.name)
  ob3.output("Kotlin慢慢学")
  ob3.printInfo()
}
```

接口

Kotlin对于接口的设计和Java并不完全相同，它增强了接口的功能，比如如下两个：**接口方法支持默认实现、接口中支持抽象属性。**例子如下：

```
// Kotlin代码
interface Flyer {
    val speed: Int
    val height
        get() = 1000
    fun kond()
    fun fly() {
        println("I can fly")
    }
}
```

Kotlin的这个设计应该是向Java8看齐，因为Java 8引入了一个新特性——接口方法支持默认实现，默认实现方法fly()转换为Java代码如下：

```java
// Java代码
void fly();

public static final class DefaultImpls {
    public static void fly(Flyer $this) {
        String var1 = "I can fly";
        System.out.println(var1);
    }
}
```

也就是在接口中定义了静态内部类去实现。

抽象属性

Java中是没有抽象属性的，因为abstract只能修饰类和方法，所以接口的属性都是常量，不支持抽象属性。然而Kotlin作为更好的Java，它的接口支持抽象属性，是因为背后通过Java中的抽象方法来实现的，比如属性speed转换为Java代码如下：

```java
// Java代码
interface Flyer{
    public abstract int getSpeed();
}
```

1、Kotlin接口中的属性，同方法一样，若没有指定默认行为，则在实现该接口的类中必须对该属性进行初始化。

2、若要指定属性默认行为，需要像属性height一样用get()进行申明。

```
interface User{
    val nickname:String
}

class PrivateUser(override val nickname:String):User


class SubscribingUser(val email:String):User{

}
```

```
在接口中的属性既可以是抽象的,也可以有访问器的实现,
但不能有幕后字段(backing field),因此访问器不能引用它们。
    interface MyInterface {
        val prop: Int // 抽象abstract,不能初始化

        val property: String
            get() = "foo" // 有访问器的实现，非抽象

        fun foo() {
            print(prop)
        }
```

```
    }

    class Child : MyInterface {
        override val prop:
        Int = 29
    }
```

类型系统

kotlin中 ? 和 ?. 和 ?: 和 as? 和 !!的区别

? 可空类型

otlin和Java的类型系统之间的一个很重要的区别就是，Kotlin对可空类型的显示支持

也就是说你可以声明一个变量，并且使用可空类型?来表示这个变量是可以为null的

比如:

java:

```
int StrLen(String s){return s.length}
//这个函数并不安全，原因是传入的参数s如果是null，就会报空指针异常
```

kotlin:

```
fun StrLen(s:String?):Int = s.length  //不能直接调用length方法
//1、这里使用了可空类型?，?可以加载任何类型的后面来表示这个类型的变量可以为null
//2、可空类型的变量在使用的时候不能直接调用它的方法
//3、也不能把可空类型的值传给非空类型
/**
*如 val x:String?=null  var y:String = x//把可空类型的x赋值给非空类型的y会报错：Type mismatch
*同样也不能把一个可控类型的值传给拥有非空类型参数的函数如StrLen(x) 传给 StrLen(x:String)
**/
```

?. 安全调用运算符

```
fun StrLen(s:String?):Int = s.length  //不能直接调用length方法
//如果增加了null检查以后，就可以直接调用s.length了,如下:
fun StrLen(s:String?):Int  = if(s!=null) s.length else 0
//但是如果每个可空类型都这样检查会显得特别累赘，此时就用到了安全调用运算符?.

s?.length 就相当于 if(s!=null) s.length else null
//如果s不为空就执行方法,如果为空就返回null
```

?: Elvis运算符

（null合并运算符）

使用?:运算符可以设置当检查结果为空的时候的返回值

```
fun foo(s:String?){
     val t:String  =  s ?:  "" //如果?:左边的值不为空返回左边的值，如果为空返回""
}

//可以这样使用
a?. peroson?. name ?: "UnKnown" //如果?:左边为空则返回"UnKnown"
//和throw运算符同事使用
//如果不为空就返回name,如果为空就抛出一个有意义的错误,而不是NullPointException
val name = a?.person?.name ?: throw illegalargumentexception("UnKnown name")//如果name为空就会报自定义的异常，防止下面代码调用而直接报空指针异常
println(name.length)//如果name为空就会报空指针异常
```

as? 安全转换运算符

尝试把值转换成给定的类型，如果类型不合适就返回null

```
foo as? Type -> foo is  Type  retrun (foo as Type)
             -> foo !is Type  return null

//as?和?:联合使用
object as? Person ?: "not human"
object as? Person ?: return false
```

!! 非空断言

Kotlin不推荐使用非空断言，通常我们会用?:来防止程序运行时报空指针异常而崩溃

如果值为null就抛出NullPointerException空指针异常

```
var s:String = s!!  //如果s为null则会抛出空指针异常，并且异常会指向使用!!的这一行
println(s)//如果s为null则会抛出空指针异常
//使用断!!可以很方便的在抛出空指针异常的时候定位到异常的变量的位置
//但是千万不要连续使用断言!!
//student!!.person!!.name//如果报空指针异常了则无法判断到底是student为空还是person为空，所以不要连续使用断言!!
```

lateinit 和 by lazy

Koltin中属性在声明的同时也要求要被初始化，否则会报错，ex:

```
private var name0: String //报错
private var name1: String = "xiaoming" //不报错
private var name2: String? = null //不报错
```

但是我们在写代码的时候又不想给它整一个可空类型的对像

因为这样看起来很繁琐 但是在一开始的时候我们又不能让它初始化

那我们就必须使用kt的延迟初始化了

otlin中有两种延迟初始化的方式。一种是**lateinit var**，一种是**by lazy**。

lateinit 只用于变量 var:

lateinit var只能用来修饰类属性，不能用来修饰局部变量，并且只能用来修饰对象，不能用来修饰基本 类型(因为基本类型的属性在类加载后的准备阶段都会被初始化为默认值)。 lateinit var的作用也比较简单，就是让编译期在检查时不要因为属性变量未被初始化而报错。

```
private lateinit var name:String

```

by lazy

而 lazy 只用于常量 val（应用于单例模式，而且当且仅当变量被第一次调用的时候，委托方法才会执 行）：

by lazy要求属性声明为 val ，即不可变变量，在java中相当于被 final 修饰。 这意味着该变量一旦初始化后就不允许再被修改值了(基本类型是值不能被修改，对象类型是引用不 能被修改)。 {} 内的操作就是返回唯一一次初始化的结果。 by lazy可以使用于类属性或者局部变量。

如果想要仔细探究它的原理 就走这里就好啦

原理窗口

类型参数

kt所有得泛型类和泛型函数的类型参数默认都是可空的

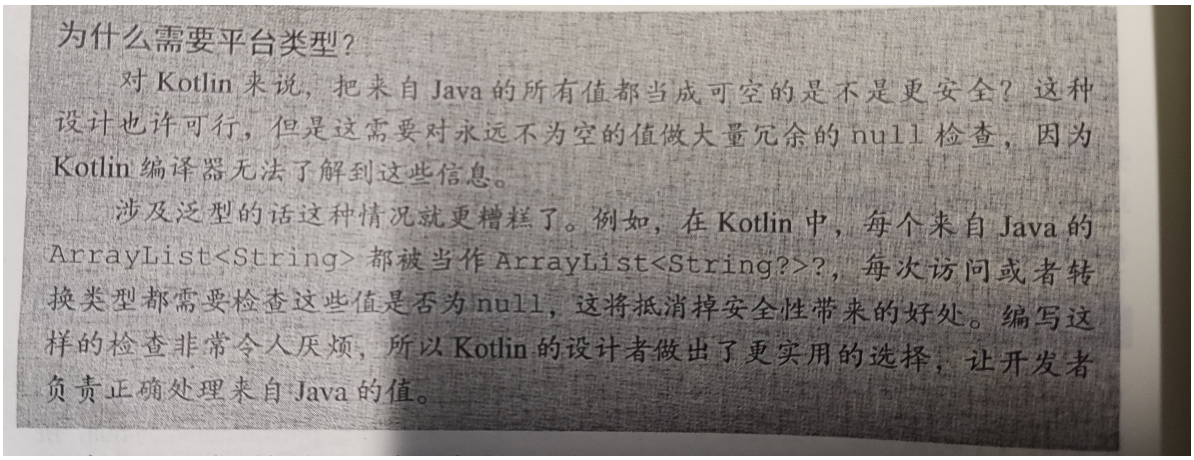
平台类型

平台类型本质上就是 Kotlin 不知道可空性信息的类型—所有 Java 引用类型在 Kotlin 中都表现为平台类型。当在 Kotlin 中处理平台类型的值的时候，它既可以被当做可空类型来处理，也可以被当做非空类型来操作。

平台类型的引入是 Kotlin 兼容 Java 时的一种权衡设计。试想下，如果所有来自 Java 的值都被看成非空，那么就容易写出比较危险的代码。反之，如果 Java 值都强制当做可空，则会导致大量的 `null` 检查。综合考量，平台类型是一种折中的设计方案。

平台类型的本质就是kt不知道可控信息的类型 既可以当作可空类型处理 也可以当作非空类型处理

为什么需要平台类型?


为什么需要平台类型？

对 Kotlin 来说，把来自 Java 的所有值都当成可空的是不是更安全？这种设计也许可行，但是这需要对永远不为空的值做大量冗余的 null 检查，因为 Kotlin 编译器无法了解到这些信息。

涉及泛型的话这种情况就更糟糕了。例如，在 Kotlin 中，每个来自 Java 的 ArrayList<String> 都被当作 ArrayList<String?>?，每次访问或者转换类型都需要检查这些值是否为 null，这将抵消掉安全性带来的好处。编写这样的检查非常令人厌烦，所以 Kotlin 的设计者做出了更实用的选择，让开发者负责正确处理来自 Java 的值。

```
error: null can not be a value of a non-null type Any
复制代码
```

```
    println(1 is Any)
        println(1 is Any?)
        println(null is Any)
        println(1 is Any?)
        println(Any() is Any?)
```

输出如下

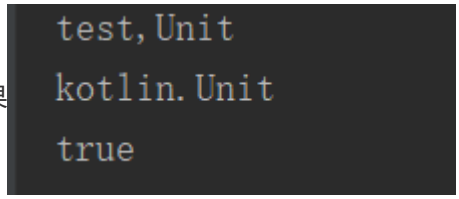
```
true
true
false
true
true
```

返回类型

当一个函数没有返回值的时候，我们用Unit来表示这个特征，而不是null，大多数时候我们不需要显示地返回Unit，或者生命一个函数的返回值是Unit，编译器会推断它。

```
fun unitExample() {
        println("test,Unit")
    }

    @JvmStatic
    fun main(args: Array<String>) {
        val helloUnit = unitExample()
        println(hellUnit)
        println(hellUnit is kotlin.Unit)
    }
```

输出结果



可以看出变量helloUnit的类型是kotlin.Unit类型。以下写法是等价的

```
fun unitExample():kotlin.Unit {
        println("test,Unit")
    }
    fun unitExample(){
        println("test,Unit")
        return kotlin.Unit
    }
    fun unitExample(){
        println("test,Unit")
    }
```

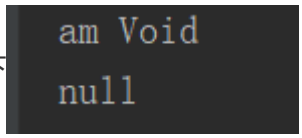
跟其他类型一样，Kotlin.Unit的类型是Any。如果是一个可空的Unit？那么父类型是Any？。

Nothing与Nothing?

在java中void不能是变量的类型，也不能作为值打印输出。但是在java中有个包装类Void是void的自动装箱类型。如果你想让一个方法的返回类型永远是null的话，可以把返回类型为这个大写的Void类型。

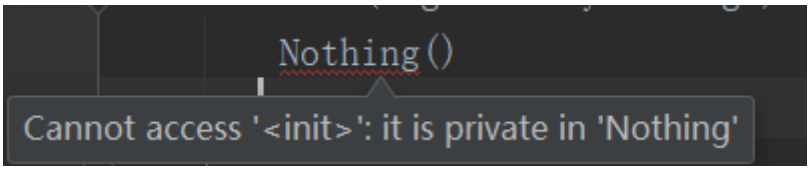
```java
public Void testV() {//声明类型是Void
        System.out.println("am Void");
        return null;//返回值只能是null
    }

    public static void main(String[] args) {
        JavaTest test = new JavaTest();
        Void aVoid = test.testV();
        System.out.println(aVoid);
    }
```

打印结果如下



这个Void对应的类型是Nothing?，其唯一可被访问的返回值也是null，Kotlin中类型层次结构最底层就是Nothing
Nothing的类定义如下

```kotlin
//Nothing的构造函数是private的，外界无法创建Nothing对象
public class Nothing private constructor()
```



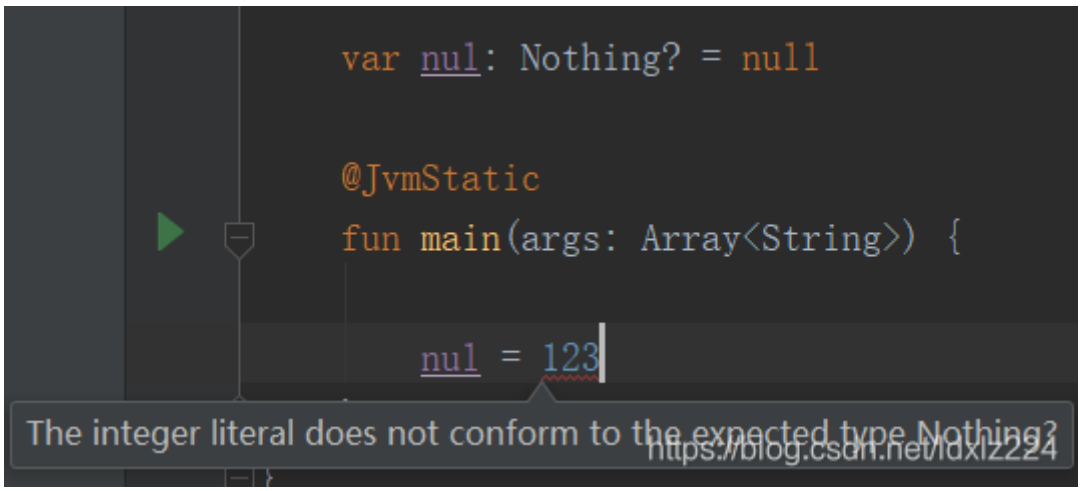
如果一个函数返回值是Nothing，那么这个函数永远不会有返回值。
但是我们可以使用Nothing来表达一个从来不存在的返回值。例如EmptyList中的get函数

```kotlin
object EmptyList : List<Nothing> {

        override fun get(index: Int): Nothing {
            throw IndexOutOfBoundsException()
        }
 }
```

因为get永远不会反回值，而是直接抛出了异常，这个时候可以用Nothing作为get函数的返回值。
再例如Kotlin标准库里面的exitProcess()函数

```kotlin
@kotlin.internal.InlineOnly
public inline fun exitProcess(status: Int): Nothing {
    System.exit(status)
    throw RuntimeException("System.exit returned normally, while it was supposed to halt JVM.")
}
```

Unit与Nothing之间的区别是，Unit类型表达式计算结果返回值是Unit；Nothing类型表达式计算结果永远是不会反回的，与java中void相同。
Nothing?可以只包含一个值 null 。



Nothing?唯一允许的值是null，可被用作任何可空类型的空引用。

泛型

泛型擦除

关于泛型擦除的知识 我们直接看一篇博客（实际上是不想抄了）

https://www.imooc.com/article/details/id/287721

协变，逆变，抗变

首先让我们搞明白这三个名词的概念吧：

假设我们有如下两个类型集合

- 第一个集合为： `Animal`和`Dog`，`Dog`是`Animal`的子类

```
open class Animal
class Dog : Animal()
```

- 第二个集合为 `List<Animal>` `List<Dog>`

```
List<Animal>
List<Dog>
```

现在问题来了：由于`Dog`是`Animal`的子类，那么`List<Dog>`就是`List<Animal>`的子类这句话在Kotlin/Java中对吗?

相信大佬们都可以回答的出来，答案是**否定**的。我们这里要说的协变，逆变，抗变就是描述上面两个类型集合的关系的。

- 协变(Covariance)：`List<Dog>`是`List<Animal>`的子类型
- 逆变(Contravariance): `List<Animal>`是`List<Dog>`的子类型
- 抗变(Invariant): `List<Animal>`与`List<Dog>`没有任何继承关系

抗变

Java中泛型是**抗变**的，那就意味着`List<String>`不是`List<Object>`的子类型。因为如果不这样的话就会产生类型不安全问题。

例如下面代码可以通过编译的话，就会在运行时抛出异常

```
List<String> strs = new ArrayList<String>();
List<Object> objs = strs;
objs.add(1);
 // 尝试将Integer 转换为String,发生运行时异常 ClassCastException: Cannot cast Integer to String
```

所以上面的代码在编译时就会报错，这就保证了类型安全。

但值得注意的是Java中的**数组是协变的**，所以数组真的会遇到上面的问题，编译可以正常通过，但会发生运行时异常，所以在Java中要优先使用泛型集合。

```
String[] strs= new String[]{"ss007"};
 Object[] objs= strs;
 objs[0] = 1;

```

协变

抗变性会严重制约程序的灵活性，例如有如下方法`copyAll`,将一个`String`集合的内容**copy**到一个`Object`集合中，这是顺理成章的事。

```
// Java
void copyAll(Collection<Object> to, Collection<String> from) {
     to.addAll(from);
}
```

但是如果`Collection<E>`中的`addAll`方法签名如下的话，`copyAll`方法就通不过编译，因为通过上面的讲解，我们知道由于抗变性，`Collection<String>`不是`Collection<Object>`的子类，所以编译通不过。

```
boolean addAll(Collection<E> c);
```

那怎么办呢?

Java通过**通配符参数**(wildcard type argument)来解决, 把`addAll`的签名改成如下即可:

```
boolean addAll(Collection<? extends E> c);
```

逆变

同理有时我们需要将`Collection<Object>`传递给`Collection<String>`就使用`? super E`，其表示可以接收`E`或者`E`的父类，子类的位置却可以接收父类的实例，这就使得泛型类型发生了**逆变**

```
void m (List<? super String){
}
```

特性

当使用? extends E 时，只能调用传入参数的读取方法而无法调用其修改方法。
当使用? super E时，可以调用输入参数的修改方法，但调用读取方法的话返回值类型永远是Object，几乎没有用处。

是不是感觉不好理解，确实不好理解！让我们一起看下code吧，理解了Java的这块，Kotlin的In和out关键字就手到擒来了。

例如有如下一接口，其有两个方法，一个修改，一个读取。

下面是两个使用通配符的方法，注意看注释

```
//协变，可以接受BoxJ<Dog>类型的参数
 private Animal getOutAnimalFromBox(BoxJ<? extends Animal> box) {
       Animal animal = box.getAnimal();
      // box.putAnimal(某个类型) 无法调用该修改方法，因为无法确定 ？究竟是一个什么类型，没办法传入
       return animal;
  }

//逆变，可以接受BoxJ<Animal>类型的参数
 private void putAnimalInBox(BoxJ<? super Dog> box){
        box.putAnimal(new Dog());
        // 虽然可以调用读取方法，但返回的类型却是Object，因为我们只能确定 ？的最顶层基类是Object
        Object animal= box.getAnimal();
  }
```

	协变	逆变	不变		
Kotlin	只能作为消费者，只能读取不能添加。	只能作为生产者，只能添加，读取受限制	既可以添加，也可以读取		
java	<? extends T>只能作为消费者，只能读取不能添加。	<? super T>只能作为生产者，只能添加，读取受限制	既可以添加，也可以读取		

高阶函数

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

Lambda语法

```
  1. 无参数的情况 ：
    val/var 变量名 = { 操作的代码 }

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

    可等价于
    // 此种写法：即表达式的返回值类型会根据操作的代码自推导出来。
    val/var 变量名 = { 参数1 ：  类型，参数2 : 类型，... -> 操作参数的代码 }

    3. lambda表达式作为函数中的参数的时候，这里举一个例子：
    fun test(a : Int，参数名 : (参数1 ：  类型，参数2 : 类型，... ) -> 表达式返回类型){
        ...
    }
```

实例讲解：

- 无参数的情况