Kotlin

1.基本介绍

1.Kotlin的发展历程

- 2011年，JetBrains 发布了 Kotlin 的第一个版本，并在2012年将其开源。
- 2016年 Kotlin 发布了1.0正式版，代表着 Kotlin 语言已经足够成熟和稳定了，并且 JetBrains 也在自家的旗舰 IDE 开发工具 IntelliJ IDEA 中加入了 Kotlin 的支持。
- 2017年 Google 宣布 Kotlin 正式成为 Android 开发一级语言，并且 Android Studio 也加入了对 Kotlin 的支持。
- 2019年Google正式宣布了Kotlin First，未来提供的官方API也将会以Kotlin版本为主。

2.Kotlin相比于Java的优势

- 语法更加简洁，对于同样的功能，使用 Kotlin 开发的代码量可能会比使用 Java 开发的减少50%甚至更多。
- 语法更加高级，相比于 Java 比较老旧的语法，Kotlin 增加了很多现代高级语言的语法特性，使得开发效率大大提升。
- 语言更加安全，Kotlin 几乎杜绝了空指针这个全球崩溃率最高的异常。
- 和 Java 是100%兼容的，Kotlin 可以直接调用使用 Java 编写的代码，也可以无缝使用 Java 第三方的开源库。这使得 Kotlin 在加入了诸多新特性的同时，还继承了 Java 的全部财富。

3.Kotlin的工作原理

Kotlin 可以做到和 Java 100%兼容，这主要是得益于 Java 虚拟机的工作机制。

其实 Java 虚拟机并不会直接和你编写的 Java 代码打交道，而是和编译之后生成的 class 文件打交道。

而Kotlin也有一个自己的编译器，它可以将Kotlin代码也编译成同样规格的class文件。

Java 虚拟机不会关心 class 文件是从 Java 编译来的，还是从 Kotlin 编译来的，只要是符合规格的 class 文件，它都能识别。

也正是这个原因，JetBrains 才能以一个第三方公司的身份设计出一门用来开发Android应用程序的编程语言。

4.如何运行Kotlin代码

1. 第一种方法是使用 IntelliJ IDEA。
2. 第二种方法是在线运行 Kotlin 代码。JetBrains 专门提供了一个可以在线运行 Kotlin 代码的网站，地址是：https://play.kotlinlang.org。
3. 第三种方法是使用 Android Studio。在任意 Android 工程中创建一个 Kotlin 文件，并编写一个 `main()` 函数，即可运行`main()`函数中的Kotlin代码。

如果是旧项目,可以在任意包下,`new` 一个 `Kotlin class`,Android studio 就会提示 `Kotlin not configured`,按照提示操作即可,或者按照以下步骤手动添加

一、安装插件

> 使用AndroidStudio -> File -> Settings -> Plugins -> Browse repositories -> 搜索 kotlin ，然后选择哪个名字为 Kotlin 的安装就好，其他的任何都可以不要。

二、gradle添加依赖

Module的`build.gradle`

```
apply plugin: 'com.android.application'
apply plugin: 'kotlin-android'//这里添加
apply plugin: 'kotlin-android-extensions'//这里添加

android {
    compileSdkVersion 25
    buildToolsVersion "25.0.2"
    ...
}
dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    compile "org.jetbrains.kotlin:kotlin-stdlib-jre7:$kotlin_version"//主要是这两个依赖
    compile "org.jetbrains.kotlin:kotlin-reflect:$kotlin_version"//主要是这两个依赖
}
```

Project的`build.gradle`

```
buildscript {
    ext.kotlin_version = '1.3.71'
    repositories {
        maven {url 'http://maven.aliyun.com/nexus/content/groups/public/'}
    }
    dependencies {
        classpath 'com.android.tools.build:gradle:2.2.3'
        classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin_version" //重点是这个
    }
}
```

2.变量

1.变量声明

- **val（value的简写的简写）用来声明一个不可变的变量，这种变量在初始赋值之后就再也不能重新赋值，对应 Java 中的 final 变量,应该优先使用 val。**
- **var（variable的简写的简写）用来声明一个可变的变量，这种变量在初始赋值之后仍然可以再被重新赋值复制，对应 Java 中的非 final 变量。**

```
fun main() {
    val a = 10
    var b = 5
    b = b + 3
    println("a = " + a)
    println("b = " + b)
}
```

PS: 每一行代码的结束可以省略掉分号`;`，这一点是和`Java`不同的地方。当然，第一次写可能会有一点不习惯。

- **Kotlin 完全抛弃了 Java 中的基本数据类型，全部使用了对象数据类型。在 Java 中 int 是整型变量的关键字，而在 Kotlin 中 Int 变成了一个类，它拥有自己的方法和继承结构。**

Java基本数据类型	Kotlin对象数据类型	数据类型说明
int	Int	整型
long	Long	长整型
short	Short	短整型
float	Float	单精度浮点型
double	Double	双精度浮点型
boolean	Boolean	布尔型
char	Char	字符型
byte	Byte	字节型

- **字符串内嵌表达式(字符串模版)**

在Kotlin中，我们可以直接将表达式写在字符串里面，即使是构建非常复杂的字符串，也会变得很简单。

Kotlin中字符串内嵌表达式的语法规则如下：

```
 val person = Person("张三", 18)
 "hello, ${person.name}. nice to meet you!"
```

当表达式中仅有一个变量的时候，还可以将两边的大括号省略：

```
val name = "张三"
"hello, $name. nice to meet you!"
```

- **自定义get() 和 set()**

  这里讲解属性的自定义`getter()`与`setter()`。由上面可知，使用`val`修饰的属性，是不能有`setter()`的。而使用`var`修饰的属性可以同时拥有自定义的`getter()`与`setter()`。通过两个实例来说明:

例1：用`val`修饰的属性自定义情况

```
class Mime{
    // size属性
    private val size = 0

    // 即isEmpty这个属性，是判断该类的size属性是否等于0
    val isEmpty : Boolean
        get() = this.size == 0

    // 另一个例子
    val num = 2
        get() = if (field > 5) 10 else 0
}

// 测试
fun main(args: Array<String>) {
    val mime = Mime()
    println("isEmpty = ${mime.isEmpty}")
    println("num = ${mime.num}")
}
```

输出结果为：

```
isEmpty = true
num = 0
```

例2：用`var`修饰的属性自定义情况

```
class Mime{

    var str1 = "test"
        get() = field   // 这句可以省略，kotlin默认实现的方式
        set(value){
            field = if (value.isNotEmpty()) value else "null"
        }

    var str2 = ""
        get() = "随意怎么修改都不会改变"
        set(value){
            field = if (value.isNotEmpty()) value else "null"
        }
}

// 测试
fun main(args: Array<String>) {
    val mime = Mime()

    println("str = ${mime.str1}")
    mime.str1 = ""
    println("str = ${mime.str1}")
    mime.str1 = "kotlin"
    println("str = ${mime.str1}")

    println("str2 = ${mime.str2}")
    mime.str2 = ""
```

```
    println("str2 = ${mime.str2}")
    mime.str2 = "kotlin"
    println("str2 = ${mime.str2}")
}
```

输出结果为：

```
str = test
str = null
str = kotlin
str2 = 随意怎么修改都不会改变
str2 = 随意怎么修改都不会改变
str2 = 随意怎么修改都不会改变
```

经过上面的实例，总结出了以下几点：

> 1. 使用了`val`修饰的属性，不能有`setter()`.
> 2. 不管是`val`还是`var`修饰的属性，只要存在`getter()`,其值再也不会变化
> 3. 使用`var`修饰的属性，可以省略掉`getter()`,不然`setter()`毫无意义。当然`get() = field`除外。而`get() = field`是`Koltin`默认的实现，是可以省略这句代码的。

故而，在实际的项目开发中，这个自定义的`getter`与`setter`的意义不是太大。

2.变量的延迟初始化

Koltin中属性在声明的同时也要求要被初始化，否则会报错。 例如以下代码:

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

可是有的时候，我并不想声明一个类型可空的对象(因为调用太麻烦了)，而且我也没办法在对象一声明的时候就为它初始化，那么这时就需要用到Kotlin提供的**延迟初始化**。
Kotlin 中有两种延迟初始化的方式。一种是**lateinit var**，一种是**by lazy**.

- **lateinit var**

```
private lateinit var name: String
```

`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。

`lateinit var`只能用来修饰类属性，不能用来修饰局部变量

`lateinit var`不能声明于可空变量。

`lateinit var` 不能声明于基本数据类型变量。例：`Int`、`Float`、`Double`等，注意：`String`类型是可以的。

Kotlin相信当开发者显式使用 `lateinit var` 关键字的时候，他一定也会在后面某个合理的时机将该属性对象初始化,假如没有初始化就调用,不然会抛出`UninitializedPropertyAccessException`异常。

另外,我们还可以通过代码判断一个全局变量是否已经完成了初始化,就能避免对同一对象重复的初始化:

```
if (!::name.isInitialized) {
    name = "zhangsan"
}
```

`::name.isInitialized` 可以判断 `name` 变量是否已经初始化了,然后我们对结果取反,如果没有初始化,那么我们对 `name` 进行初始化.

- **by lazy**

`by lazy` 的作用是懒加载,把想要延迟执行的代码放入到 `by lazy` 代码块中,在一开始的时候就不会去执行,只有当第一次调用被 `by lazy` 修饰的变量的时候,代码块中的代码才会执行.

`by lazy` 可以用于类属性和局部变量, `by lazy` 本身是一种属性委托。 写法如下：

```
val zhangsan by lazy {
    Person("zhangsan", 18)
}
```

`by lazy` 并不是连在一起的关键字,属性委托的关键字是 `by` , `lazy` 是一个高阶函数 .

在 `lazy` 函数中会创建并返回一个 `Delegate` 对象,当我们调用 `zhangsan` 属性的时候,实际上我们调用的的 `Delegate` 对象的 `getValue()` 方法,然后 `getValue()` 会调用 `lazy` 函数传入的 `Lambda` 表达式,这样表达式中的代码就可以得到执行,并且调用 `zhangsan` 属性后,得到的值就是 `Lambda` 表达式最后一行的返回值 .

3.常量

`Kotlin` 中声明常量的方式和在 `Java` 中声明常量的方式有很大的区别。这里举例说明：

`Kotlin` 中使用 `val` 时候对应的 `Java` 代码：

```
Kotlin中的 val numA = 6   等价于  Java中的：public final int numA = 6
```

很显然， `Kotlin` 中只用 `val` 修饰还不是常量，它只能是一个不能修改的变量。那么常量怎么定义呢？其实很简单，在 `val` 关键字前面加上 `const` 关键字。

即：

```
const val NUM_A = 6
```

其特点： `const` **只能修饰** `val` **，不能修饰** `var`

声明常量的三种正确方式

> 1. 在顶层声明
> 2. 在 `object` 修饰的类中声明，在 `kotlin` 中称为**对象声明**，它相当于 `Java` 中一种形式的单例类
> 3. 在伴生对象中声明

举例说明：

```
// 1. 顶层声明
const val NUM_A : String = "顶层声明"

// 2. 在object修饰的类中
object TestConst{
    const val NUM_B = "object修饰的类中"
}

// 3. 伴生对象中
```

```
class TestClass{
    companion object {
        const val NUM_C = "伴生对象中声明"
    }
}
```

3.函数

1.基本语法

定义一个函数的语法规则如下：

```
fun methodName(param1: Int, param2: Int): String {
      return (param1 + param2).toString()
}
```

`fun`（function的简写）是定义函数的关键字，无论你定义什么函数，都一定要使用fun来声明。
紧跟在 `fun` 后面的是函数名
函数名后面的一对括号中，可以声明该函数接收什么参数。
括号后面的部分是可选的，用于声明该函数会返回什么类型的数据。如果不需要返回任何数据，这部分可以不写。
两个大括号之间的内容就是函数体，可以在这里编写函数的具体逻辑。

2.单行代码函数

当一个函数的函数体中只有一行代码时，可以使用单行代码函数的语法糖：

```
fun methodName(param1: Int, param2: Int)= (param1 + param2).toString()
```

使用这种写法，可以直接将唯一的一行代码写在函数定义的尾部，中间用等号连接即可。
`return` 关键字也可以省略，等号足以表达返回值的意思。
Kotlin 还拥有出色的类型推导机制，可以自动推导出返回值的类型。

3.函数的参数默认值

`Kotlin` 允许在定义函数的时候给任意参数设定一个默认值，这样当调用此函数时就不会强制要求调用方为此参数传值，在没有传值的情况下会自动使用参数的默认值。语法格式如下：

```
fun printParams(num: Int,str: String = "hello world") {
    println("num is $num , str is $str")
}
```

这里给 `printParams()` 函数的第二个参数设定了一个默认值，这样当调用 `printParams()` 函数时，可以选择给第二个参数 `str` 传值，也可以选择不传，在不传的情况下就会自动使用默认值,可以减少方法的重载

4.使用键值对传参(具名参数)

我们将上面的`printParams()`函数参数调换一个顺序:

```
fun printParams(str: String = "hello world", num: Int) {
    println("num is $num , str is $str")
}
```

我们尝试调用上面定义的 `printParams()` 函数:

```
printParams(123)
```

我们会发现编译器是会报错的,因为 `printParams()` 函数定义的第一个参数是 `String` 类型的,编译器会认为我们要将一个 `Int` 类型的值赋值给一个 `String` 类型的参数,会报类型不匹配的错误

`Kotlin` 提供了一个机制,就是可以通过键值对的方式传参,从而不必像传统写法那样按照参数定义的顺序来传参,比如调用 `printParams()` 函数,我们可以这么写:

```
printParams(num = 123,str = "world")
```

此时哪个参数在前在后都无所谓, `Kotlin` 会准确的将参数匹配上,使用这种键值对的方式后,我们就可以省略掉 `str` 参数了,代码如下:

```
printParams(num = 123)
```

5.不定长参数和多返回值

- **不定长度参数**

例如 java `public void println(String.. args) { }` 可以这样定义不定个数参数的函数。
Kotlin 通过关键字 `vararg` 实现这个功能，如示例：

```
fun prints(vararg strings: String) {
  for (string in strings){
    println(string)
  }
}

prints("a", "b", "c")
```

如果已经有一个数组， 可以通过关键字 `*` 传递数组。

```
val strings = arrayOf("a", "b", "c", "d", "e")
prints(*strings)
```

- **返回多个值**

Kotlin内置了 Pair 和 Triple， 可以返回2个值和3个值,比如以下例子:

```
fun multiReturnValues():Triple<Int,Long,Double>{
    return Triple(1,3L,10.0)
}
//方式1
val multiReturnValues = multiReturnValues()
val first = multiReturnValues.first
val second = multiReturnValues.second
val third = multiReturnValues.third
println("first:$first  second:$second  third:$third")
//方式2,解构声明(一个解构声明创造了多个变量)
val (a, b, c) = multiReturnValues()
println("a:$a  b:$b  c:$c")
```

我们可以看到,其实 Kotlin 也只能返回一个返回值,只是内置了 Pair 和 Triple,让我们可以模拟返回多返回值

- **解构声明**

所谓的解构声明就是将一个对象解构(destructure)为多个变量，也就是意味着一个解构声明会一次性创建多个变量.简单的来说，一个解构声明有两个动作：

- 声明了多个变量
- 将对象的属性值赋值给相应的变量

2. 将对象的属性值赋值给相应的变量

比如，有个数据类Person，其有name和age两个属性

```
data class Person(var name: String, var age: Int) {
}
```

当我们对Person的实例使用解构声明时，可以这样做：

```
var person: Person = Person("Jone", 20)
var (name, age) = person  //按照形参的顺序,不会去匹配形参的命名,即name和age调换,那么name就会被赋值20

println("name: $name, age: $age")// 打印：name: Jone, age: 20
```

6.标准函数

`Kotlin`的标准函数指的是 `Standard.kt` 文件中定义的函数,任何 `Kotlin` 代码都可以自由地调用所有的标准函数,常用的标准扩展函数`run`,`with`, `let` , `also` 和 `apply`,它们都是适用于任何对象的通用扩展函数。但是对于`run`,`with`,`let`,`also`和`apply`这五个函数他们的用法及其相似，以至于我们无法确定去选择使用哪一个。那么现在我们就来聊一下这五个函数它们的使用方法，它们的不同之处以及在什么场景下去使用。

1.run

`run` 函数分为两种,`run{}` 和 `T.run{}`

- **run{}**

如下代码:

```
run {
    使用子命名空间运行代码段
    返回最后的对象结果
}

fun test(){
    var animal = "cat"
    run {
        val animal = "dog"
        println(animal)   // dog
    }
    println(animal)       //cat
}
```

在这个简单的 `test` 函数当中我们拥有一个单独的作用域，在 `run` 函数中能够重新定义一个 animal 变量，并且它的作用域只存在于 `run` 函数当中。

目前对于这个 `run` 函数看起来貌似没有什么用处，但是在 `run` 函数当中它不仅仅只是一个作用域，他还有一个返回值。他会返回在这个作用域当中的最后一个对象。

```
val i = run {
    if (true) 1 else 0
}
println(i) //1
```

- **T.run{}**

```
T.run {
    this 指代 T 对象
    可直接访问 T 对象内部成员
    返回最后的对象结果
}

val student = Student(name = "李四")
val s = student.run {
    this.name = "王五"  //this.可以省略
    age = 1
    sno = "abc"
    ...
    2
}
println(s)  //2
println(student.name) //王五
```

2.with

`with` 接收两个参数,第一个参数可以是任意类型的对象,第二个是 `Lambda` 表达式, `with` 会在 `Lambda` 表达式中提供第一个参数对象的上下文,并使用 `Lambda` 表达式的最后一行作为返回值返回,示例代码如下:

```
with(T) {
    this 指代 T 对象
    可直接访问 T 对象内部成员
    返回最后的对象结果
}

val s = with(student) {
    this.name = "王五"  //this.可以省略
    age = 1
    sno = "abc"
    2
}
println(s)    //2
println(student.name)  //王五
```

通过代码可以看出,`with` 和 `T.run` 是很相似的

3.let

用法和 `with`,`T.run` 类似,更多的情况用于辅助判空

```
T.let {
    默认 it 指代 T 对象
    更显式区分调用 T 对象成员或外部变量
    返回最后的对象结果
}

val student = Student(name = "李四")
val s = student.let {
    it.name = "王五"
    it.age = 1
    it.sno = "abc"
    2
}
println(s)    //2
println(student.name)  //王五
```

4.Apply

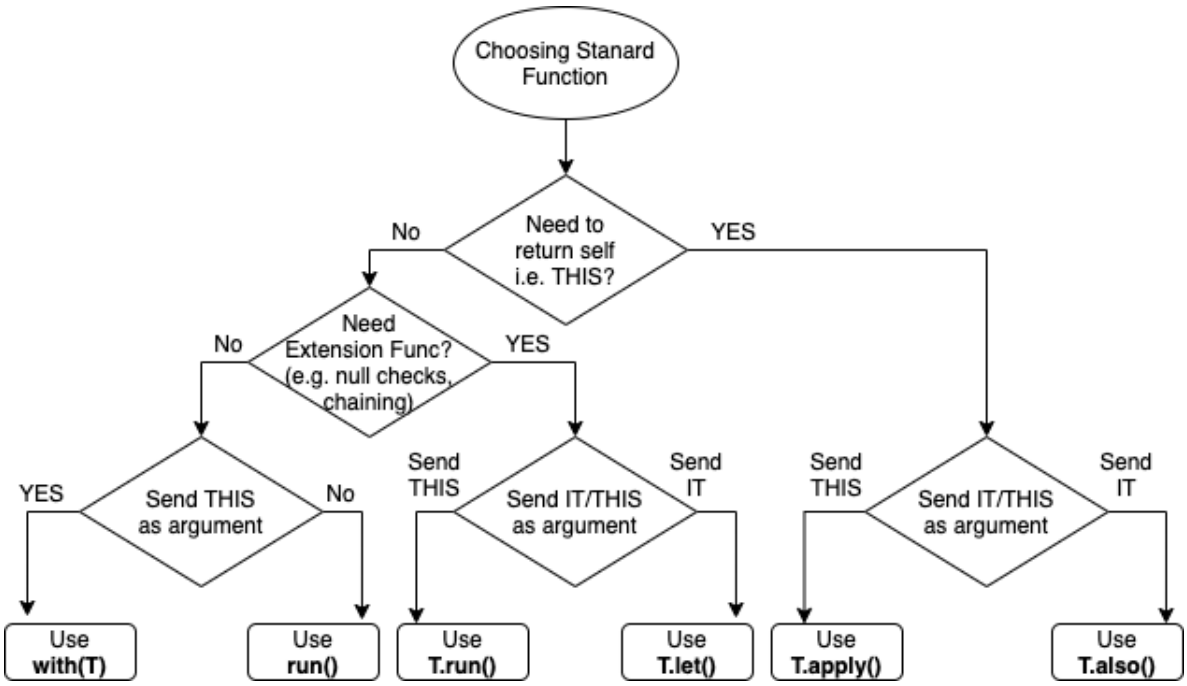
```
T.apply {
    this 指代 T 对象
    可直接访问 T 对象内部成员
    返回 T 对象本身
}

val student = Student(name = "李四")
val s = student.apply {
    this.name = "王五"
    age = 1
    sno = "abc"
    2
}
println(s == student) //true
println(student.name) //王五
```

5.also

```
T.also {
    默认 it 指代 T 对象
    更显式区分调用 T 对象成员或外部变量
    返回 T 对象本身
}
val student = Student(name = "李四")
val s = student.also {
    it.name = "王五"
    it.age = 1
    it.sno = "abc"
    2
}
println(s == student) //true
println(student.name) //王五
```

6.总结



我们可以看出在这五个扩展函数当中,就是接收者和返回值的不同。对于`with`,`T.run`,`T.apply`接收者是`this`，而`T.let`和`T.also`接受者是~；对于`with`,`run`,`T.run`,`T.let`返回值是作用域的最后一个对象（`this`）,而`T.apply`和`T.also`返回值是调用者本身(`itself`)。

7.静态方法

在`Java`中定义一个静态方法很简单,只需要在方法上声明一个关键字`static`就可以了:

```
public class Fragment{
    public static void doSomeThing(){
        //do something
    }
}
```

`Kotlin`的静态方法:

```
class Util {
    companion object {

        fun doAction() {
            //do action
        }
    }
}
```

`Kotlin`中可以这么调用:

```
 Util.doAction()
```

这个时候如果你尝试在`Java`代码中调用`doAction()`,你会发现`doAction()`不存在,因为它并不是真正的静态方法,而是`companion object`关键字在`Util`类的内部创建了一个伴生类,`doAction`调用的是伴生类中的`doAction`方法,只不过`kotlin`保证`Util`中始终只会保证存在一个伴生类对象.

如果我们真的需要在 `Java` 代码中调用 `doAction()`,那么我们可以在`companion object`中的方法上添加`@JvmStatic` 注解,那么 `Kotlin` 编译器就会将这些方法编译成为真正的静态方法,在 Java代码中就可以调用,如下图:

```
class Util {
    companion object {

        @JvmStatic
        fun doAction() {
            //do action
        }
    }
}
```

注意:`@JvmStatic` 只能在单例类(`Object`) 或者 `companion object` 中的方法上,如果添加在普通方法上,会直接提示语法错误

8.顶层函数(Top level function)

除了静态函数，Kotlin 还有更方便的东西：「`top-level declaration` 顶层声明」。其实就是把属性和函数的声明不写在 `class` 里面，这个在 Kotlin 里是允许的,我们要定义一个顶层函数,首先要先创建一个Kotlin 文件,注意创建文件名的时候,文件类型要选 `File`,假设我们给文件名起名叫 `Helper` 然后在新建的类中,我们可以定义如下函数:

```
// 属于 package，不在 class/object 内
fun topLevelFuncion() {
}
```

这样写的属性和函数，不属于任何 `class`，而是直接属于 `package`，它和静态变量、静态函数一样是全局的，但用起来更方便：你在其他 Kotlin 代码中调用的时候，就连类名都不用写：

```
topLevelFunction()
```

但是,如果我们在 Java 代码中在直接调用 `topLevelFunction()`,你会发现找不到 `topLevelFunction()` 方法,因为我们刚才给文件名起名叫 `Helper`,Kotlin 编译器会给我们自动创建了一个叫 `HelperKt` 的 `Java` 类,`topLevelFuncion()` 是以静态方法的形式在 `HelperKt` 文件中的,所以我们在 Java 中,使用如下代码调用:

```
 HelperKt.toplevelFunction();
```

9.扩展函数

扩展函数可以在已有类中添加新的方法，不会对原类做修改，扩展函数定义形式：

```
fun ClassName.methodName(param1: Int,param2: Int): Int{
    return 0
}
```

相比于定义一个普通的函数,定义扩展函数只需要在函数名前面加上一个 `ClassName.` 的语法结构,就表示将函数添加到指定类当中了.

想象一下,我们需要有这么这个需求: 一段字符串可能包含字母,数字,和特殊符号,我们希望能统计到字符串中的字符数量.按照 Java 的编程思维,我们会写出如下函数:

```
object StringUtil {
    fun letterCount(str: String): Int {
        var count = 0
        for (char in str) { //遍历字符串中的字符
            if (char.isLetter()) {
                count++
            }
        }
        return count
    }
}
```

当我们需要统计某个字符串的数量时,需要编写如下代码:

```
val str = "abc123!#@"
val count = StringUtil.letterCount(str)
```

但是我们有了扩展函数的后,可以使用面向对象的思维来实现这个功能,比如将 `letterCount()` 函数添加到 String 类中

由于我们希望向 `String` 添加一个扩展函数,因此需要创建一个 `String.kt` 的文件,文件名没有固定要求格式,但是为了便于查找,建议向哪个类添加扩展函数,就定义一个同名的 Kotlin 文件;并且,建议定义为顶层方法,这样可以让扩展函数拥有全局的访问阈,现在我们向 `String.kt` 添加如下代码:

```
fun String.letterCount(): Int {
    var count = 0
    for (char in this) {
        if (char.isLetter()) {
            count++
        }
    }
    return count
}
```

我们将 `letterCount()` 定义为 String 类的扩展函数,那么函数中就自动拥有了 String 实例的上下文,因此 `letterCount()` 就不再需要接收一个字符串参数了,直接遍历 `this` 即可, `this` 即代表字符串本身.

定义好了扩展函数,统计某个字符串中的字符数量,只需要这样写即可:

```
val count = "abc123!#@".letterCount()
```

除了 `String` 类之外,你还可以向任何类中添加扩展函数,Kotlin 基本对此没有限制.如果能利用好扩展函数这个功能,将会大幅提升代码质量和开发效率.

10.运算符重载

什么是运算符重载？

简单来说，就是 Kotlin 通过调用自己代码中定义特定的函数名的函数（成员函数或者扩展函数），并且用 `operator` 修饰符标记，来实现特定的语言结构，例如如果你在一个类上面定义了一个特定函数命名 `plus` 的函数，那么按照 Kotlin 的约定，可用在这个类的实例上使用 `+` 运算符,结构如下:

```
class Obj {
    operator fun plus(obj: Obj): Obj {
        //处理相加逻辑
    }
}
```

如上代码中,关键字 `operator` 和 `plus` 都是固定不变的,接收的参数类型和返回值类型可以视你的逻辑自行设定.

上述的代码标识一个 `Obj` 对象可以和 另一个 `Obj` 对象相加,返回一个新的 `Obj` 对象,对应的调用方式如下:

```
val obj1 = Obj()
val obj2 = Obj()
val obj3 = obj1 + obj2
```

这是Kotlin给我们提供的语法糖,在编译的时候,会被转换为 obj1.plus(obj2) 的调用方式,所以,在 java 代码中调用的时候,需要这么调用: `obj1.plus(obj2);`

比如以下例子:

```
class Money() {
    var value = 0

    operator fun plus(money: Money): Money {
        val sum = value + money.value
        val newMoney = Money()
        newMoney.value = sum
        return newMoney
    }

    operator fun plus(i: Int): Money {
        val sum = value + i
        val newMoney = Money()
        newMoney.value = sum
        return newMoney
    }
}
```

调用如下:

```
val money1 = Money()
money1.value = 8
val money2 = Money()
money2.value = 5
val money3 = money1 + money2
println("money3.value = ${money3.value}")  //打印结果: money3.value = 13
```

不光是 plus (+),包括 minus (-) ,times (*) ,div(/) 等,都可以重载

4.逻辑控制

1. if条件语句

Kotlin中 的 `if` 语句相比于 Java 有一个额外的功能：它是可以有返回值的，返回值就是`if`语句每一个条件中最后一行代码的返回值。

```
fun largerNumber(num1: Int, num2: Int): Int {
    val value = if (num1 > num2) {
        num1
    } else {
        num2
    }
    return value
}
```

上述代码，`value` 其实是一个多余的变量，我们可以直接将 `if` 语句返回，这样代码将会变得更加精简，如下所示：

```
fun largerNumber(num1: Int, num2: Int): Int {
    return if (num1 > num2) {
        num1
    } else {
        num2
    }
}
```

当一个函数只有一行代码时，可以省略函数体部分，直接将这一行代码使用等号串连在函数定义的尾部。虽然 `largerNumber()` 函数不止只有一行代码，但是它和只有一行代码的作用是相同的，只是 `return` 了一下 if 语句的返回值而已，符合该语法糖的使用条件。那么我们就可以将代码进一步精简：

```
fun largerNumber(num1: Int, num2: Int) = if (num1 > num2) {
    num1
} else {
    num2
}
```

最后，还可以将上述代码再精简一下，直接压缩成一行代码：

```
fun largerNumber(num1: Int, num2: Int) = if (num1 > num2) num1 else num2
```

顺便说一下,Kotlin 是没有三元表达式的,但是可以使用精简的 if 函数代替

2.when条件语句

1.精确匹配

当需要判断的条件非常多的时候，可以考虑使用 `when` 语句来替代 `if` 语句。

```
    fun getScore(name: String): Int {
        var value: Int
        when (name) {
            "Tom" -> {
                value = 86
            }
            "Jim" -> {
                value = 86
```

```
            }
            "Jack" -> {
                value = 95
            }
            "Lily" -> {
                value = 100
            }
            else -> {
                value = 0
            }
        }
        return value
    }
```

`when`的执行逻辑里,即 `{}` 只有一行代码时, `{}` 可以省略,,精简如下:

```
    fun getScore(name: String): Int {
        var value: Int
        when (name) {
            "Tom" -> value = 86
            "Jim" -> value = 86
            "Jack" -> value = 95
            "Lily" -> value = 100
            else -> value = 0
        }
        return value
    }
```

`when` 也可以有返回值的,可以再精简:

```
    fun getScore(name: String): Int {
        return when (name) {
            "Tom" -> 86
            "Jim" -> 86
            "Jack" -> 95
            "Lily" -> 100
            else -> 0
        }
    }
```

`when` 也可以使用单行代码函数的语法糖,最后精简如下:

```
  fun getScore(name: String): Int = when (name) {
        "Tom" -> 86
        "Jim" -> 86
        "Jack" -> 95
        "Lily" -> 100
        else -> 0
    }
```

2.类型匹配

除了精确匹配之外，`when` 语句还允许进行类型匹配。

```
fun checkNumber(num: Number) {
    when (num) {
        is Int -> println("number is Int")
        is Double -> println("number is Double")
        else -> println("number not support")
    }
}
```

3.与逗号结合使用

相当于switch语句中的不使用break跳转语句

例：

```
when(1){
     // 即x = 1,2,3时都输出1。
    1 , 2 , 3 -> {
        println("1")
    }
    5 -> {
        println("5")
    }
    else -> {
        println("0")
    }
}
```

4.条件可以使用任意表达式

条件可以使用任意表达式，不仅局限于常量,相当于`if`表达式的用法:

```
var num:Int = 5
when(num > 5){
    true -> {
        println("num > 5")
    }
    false ->{
        println("num < 5")
    }
    else -> {
        println("num = 5")
    }
}
```

5.检查值是否存在于集合或数组中

- 操作符：
    1. `(in)` 在
    2. `(!in)` 不在
- 限定:只适用于数值类型

```
var arrayList = arrayOf(1,2,3,4,5)
when(1){
    in arrayList.toIntArray() -> {
        println("1 存在于 arrayList数组中")
    }
    in 0 .. 10 -> println("1 属于于 0~10 中")
    !in 5 .. 15 -> println("1 不属于 5~15 中")
    else -> {
        println("都没匹配上")
    }
}
```

3.for-in循环语句

1.基本使用

我们可以使用如下 Kotlin 代码来表示一个区间：

```
val range = 0..10
```

上述代码表示创建了一个0到10的区间，并且两端都是闭区间，这意味着 0 到 10 这两个端点都是包含在区间中的，用数学的方式表达出来就是[0, 10]。
也可以使用 `until` 关键字来创建一个左闭右开的区间：

```
val range = 0 until 10
```

上述代码表示创建了一个 0 到 10 的左闭右开区间，它的数学表达方式是[0, 10)。有了区间之后，我们就可以通过 `for-in` 循环来遍历这个区间：

```
fun main() {
    for (i in range) {
        println(i)
    }
    //println(range.joinToString())//快速打印区间内的值的方法
}
```

可以创建一个连续的区间:

```
 val rangeDouble = 0.0 .. 10.0
```

一般用于判断某值是否在区间内:

```
 if(3.0 in rangeDouble){
      println("3.0 in $rangeDouble")
 }
 if(11.0 !in rangeDouble){
      println("11.0 not in $rangeDouble")
 }
```

指定字符串区间,同样可以使用`in`,或者是遍历:

```
 val charRange = 'A'..'Z'
```

2.指定步长

如果你想跳过其中的一些元素，可以使用 `step` 关键字：

```
fun main() {
    for (i in 0 until 10 step 2) {
        println(i)
    }
}
//0,2,4,6,8
```

3.降序区间

如果你想创建一个降序的区间，可以使用 `downTo` 关键字：

```
fun main() {
    for (i in 10 downTo 1) {
        println(i)
    }
}
//10,9,8,7,6,5,4,3,2,1
```

5.面向对象编程

1.类与对象

可以使用如下代码定义一个类，以及声明它所拥有的字段和函数：

```
class Person {
    var name = ""
    var age = 0

    fun eat() {
        println(name + " is eating. He is " + age + " years old.")
    }
}
```

然后使用如下代码创建对象，并对对象进行操作：

```
fun main() {
    val p = Person()
    p.name = "Jack"
    p.age = 19
    p.eat()
}
```

2.继承

Kotlin 中一个类默认是不可以被继承的，如果想要让一个类可以被继承，需要主动声明 `open` 关键字：

```
open class Person {
    ...
}
```

要让另一个类去继承 `Person` 类，则需要使用冒号关键字：

```
class Student : Person() {
    var sno = ""
    var grade = 0
}
```

3.接口

Kotlin 中定义接口的关键字和 Java 中是相同的，都是使用的 interface：

```
interface Study {
    fun readBooks()
    fun doHomework()
}
```

而 Kotlin 中实现接口的关键字变量了冒号，和继承使用的是同样的关键字：

```
class Student(val name: String, val age: Int) : Study {
    override fun readBooks() {
        println(name + " is reading.")
    }

    override fun doHomework() {
        println(name + " is doing homework.")
    }
}
```

4.数据类

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

Kotlin 会根据数据类的主构造函数中的参数将 `equals()、hashCode()、toString()` 等固定且无实际逻辑意义的方法自动生成，从而大大简少了开发的工作量.

建议给数据类一个默认的无参构造函数,以避免出现文中的问题. Android 避坑指南，发现了一个极度不安全的操作,文中的主要问题是后端没有返回 Android 所需要的字段,导致 Kotlin 空类型失效的问题

5.单例类

```
object Singleton {
    fun singletonTest() {
        println("singletonTest is called.")
    }
}
```

而调用单例类中的函数比较类似于Java中静态方法的调用方式：

```
Singleton.singletonTest()
```

这种写法虽然看上去像是静态方法的调用，但其实 Kotlin 在背后自动帮我们创建了一个 Singleton 类的实例，并且保证全局只会存在一个 Singleton 实例。

6.构造函数

和 Java 一样,Kotlin也有构造函数的概念,但是 Kotlin中,将构造函数分成了两种,**主构造函数**和**次构造函数**

- **主构造函数**

每个类都有默认的不带参数的主构造函数,当然可以显示的指定参数,主构造函数的特点是没有函数体,直接定义在类名后面即可,比如下面这种写法:

```
class Student(val sno: String, val grade: Int) : Person() {
    init {
        println("sno is $sno")
        println("grade is $grade")
    }
}
```

这里我们将学号和年级字段都放到了主构造函数中,表明在对`Student`实例化的时候,必须传入构造函数中的所有参数,比如:

```
val student = Student("a123",5)
```

主构造函数是没有函数体的,如果想在主构造函数中写一些逻辑,可以使用 Kotlin 提供的一个`init`结构体,逻辑可以写在`init`结构体中

仔细观察`Person`类后面的一对括号,为什么要写括号呢?这个涉及到了 Java中,子类的构造函数必须调用父类的构造函数,这个规定在 Kotlin 中也要遵守

看一下`Student`类,我们声明了一个主构造函数,根据继承特性的规定,我们必须调用父类的构造函数,可是主构造函数中并没有函数体,所以`Kotlin`使用了这么一个规定:`括号`。子类的主构造函数调用了父类的哪个构造函数,在继承的时候,通过括号来指定.

所以上面的`Student`继承`Person`中,因为`Person`没有显示的指明主构造函数的参数,所以只需要写一对括号:`()`.就调用了 Person 的主构造函数,

如果我们把 Person 改造一下:

```
open class Person(val name: String, val age: Int) {
}
```

`Student`中的代码就会报错:`No value passed for parameter 'age'`,因为Student 调用了 Person 的主构造函数,但是没有传入参数,Student 修改为如下:

```
class Student(val sno: String, val grade: Int, name: String, age: Int) :
Person(name = name, age = age) {
}
```

我们在`Studnet`主构造函数中,传入了`name`和`age`字段,然后传给`Person`的构造函数,注意:因为我们在主构造函数中声明成`val`或`var`的参数,将自动成为该类的字段,所以`name`和`age`不能在添加`val`或`var`关键字了,否则会和父类中的字段冲突

其实我们在Kotlin中几乎用不到次构造函数,因为 Kotlin 给我们提供了一个函数设定默认参数的功能,基本上可以替代次构造函数的作用,比如下面的 Student 类:

```
class Student(val sno: String = "a123", val grade: Int = 1, name: String, age:
Int = 18) : Person(name = name, age = age) {
}
```

我们默认给定了学号,年级和年龄,创建 Student 对象的时候,学号,年级和年龄就变得可传可不传,使用如下代码就可以创建一个名叫李四的 Student 对象:

```
Student(name = "李四")
```

- **次构造函数**

一个类可以有多个次构造函数，用`constructor`关键字声明。Kotlin 中规定，当一个类既有主构造函数又有次构造函数时，所有次构造函数都必须使用`this`关键字直接或间接的调用主构造函数,如下例子:

```
class Student(val sno: String = "a123", val grade: Int = 1, name: String, age:
Int = 18) : Person(name = name, age = age) {

    constructor(name: String, age: Int) : this("", 0, name, age)

    constructor() : this("", 0)
}
```

6. 数组型（Array）

- `Kotlin`中数组由`Array<T>`表示
- 创建数组的3个函数
    1. `arrayOf()`
    2. `arrayOfNulls()`
    3. 工厂函数（`Array()`）

1.arrayOf()

> 创建一个数组，参数是一个可变参数的泛型对象

例：

```
var arr1 = arrayOf(1, 2, 3, 4, 5) //等价于java [1,2,3,4,5]
println(arr1.joinToString())

var arr2 = arrayOf("0", "2", "3", 'a', 32.3f)
println(arr2.joinToString())
```

输出结果为：

```
1, 2, 3, 4, 5
0, 2, 3, a, 32.3
```

2.arrayOfNulls()

> 用于创建一个指定数据类型且可以为空元素的给定元素个数的数组

例：

```
var arr3 = arrayOfNulls<Int>(3)

//如若不予数组赋值则arr3[0]、arr3[1]、arr3[2]皆为null
println(arr3.joinToString())

//为数组arr3赋值
arr3[0] = 10
arr3[1] = 20
arr3[2] = 30
println(arr3.joinToString())
```

输出结果为：

```
null, null, null
10, 20, 30
```

3.工厂函数

- 使用一个工厂函数`Array()`，它使用数组大小和返回给定其索引的每个数组元素的初始值的函数。
- `Array()` => 第一个参数表示数组元素的个数，第二个参数则为使用其元素下标组成的表达式

例：

```
var arr4 = Array(5,{index -> (index * 2).toString() })
println(arr4.joinToString())
```

输出结果为:

```
0, 2, 4, 6, 8
```

4.原始类型数组

- `Kotlin`还有专门的类来表示原始类型的数组，没有装箱开销，它们分别是：
    1. `ByteArray` => 表示字节型数组
    2. `ShortArray` => 表示短整型数组
    3. `IntArray` => 表示整型数组
    4. `LongArray` => 表示长整型数组
    5. `BooleanArray` => 表示布尔型数组
    6. `CharArray` => 表示字符型数组
    7. `FloatArray` => 表示浮点型数组
    8. `DoubleArray` => 表示双精度浮点型数组
- PS: **`Kotlin`中不支持字符串类型这种原始类型数组，可以看源码`Arrays.kt`这个类中并没有字符串数组的声明。而源码中`StringArray.kt`这个类并不是声明字符串型数组的。**

下面的例子只演示了几种，其他的类似。
例：

```
var intArr: IntArray = intArrayOf(1,2,3,4,5)
println(intArr.joinToString())

var charArr: CharArray = charArrayOf('a','1','b','c','3','d')
println(charArr.joinToString())


var longArr: LongArray = longArrayOf(12L,1254L,123L,111L)
println(longArr.joinToString())
```

输入结果为:

```
1, 2, 3, 4, 5
a, 1, b, c, 3, d
12, 1254, 123, 111
```

7.集合

1.集合的类型

Kotlin 集合相对于 Java 的集合,增加了"**不可变**"集合的接口

	Kotlin	**Java**
不可变List (不可以添加或者删除元素)	**List**	List
可变List	MutableList	List
不可变Set (不可以添加或者删除元素)	**Set**	Set
可变Set	MutableSet	Set
不可变Map (不可以添加或者删除元素)	**Map<K,V>**	Map<K,V>
可变Map	MutableMap<K,V>	Map<K,V>

2.集合的创建

使用如下代码可以初始化一个 List/ MutableList 集合

```
val list = listOf<String>("Apple", "Banana", "Orange", "Pear", "Grape")

//或者
val mutableList = mutableListOf("Apple", "Banana", "Orange", "Pear", "Grape")
//Kotlin有出色的类型推导机制,泛型声明<String>可以省略

//或者
 val stringList = ArrayList<String>()
//对应的java代码为
List<String> stringList = new ArrayList<>();
```

使用如下代码可以初始化一个 Set/MutableSet 集合：

```
val set = setOf("Apple", "Banana", "Orange", "Pear", "Grape")
//或者
val  mutableSet = mutableSetOf("Apple", "Banana", "Orange", "Pear", "Grape")
```

使用如下代码可以初始化一个 Map/MutableMap 集合：

```
val map = mapOf<String,Any>("name" to "张三", "age" to 18)
//或者
val mutableMap = mutableMapOf<String,Any>("name" to "张三", "age" to 18)
```

3.集合的读写

1. 添加

```
val stringList = ArrayList<String>()
for (i in 0 until 10) {
    stringList.add("num:$i")
}
//或者
for (i in 0 until 10) {
    stringList += "num:$i"
}
```

2. 删除

```
val stringList = ArrayList<String>()
for (i in 0 until 10) {
    stringList.remove("num:$i")
}
//或者
for (i in 0 until 10) {
    stringList -= "num:$i"
}
```

3. 读写

```
val stringList = ArrayList<String>()
stringList[5] ="Hello World"
val valueAt5 = stringList[5]

val map = HashMap<String, Int>()
map["hello"] = 10
println(map["hello"])
```

8.空指针检查

1.什么是空指针

不用我多说,大家也知道什么是空指针了,如下 java 代码

```
public void doStudy(Study study) {
    study.readBooks();
    study.doHomework();
}
```

上面的 java 代码安全吗?不一定，因为这要取决于调用方传入的参数是什么，如果我们向 `doStudy()` 方法传入了一个null参数，那么毫无疑问这里就会发生空指针异常。因此，更加稳妥的做法是在调用参数的方法之前先进行一个判空处理，如下所示：

```
public void doStudy(Study study) {
    if (study != null) {
        study.readBooks();
        study.doHomework();
    }
}
```

2.Kotlin中的可空类型系统

Kotlin中引入了一个可空类型系统的概念，它利用编译时判空检查的机制几乎杜绝了空指针异常。

```
fun doStudy(study: Study) {
    study.readBooks()
    study.doHomework()
}
```

这段代码看上去和刚才的 Java 版本并没有什么区别，但实际上它是没有空指针风险的，因为 Kotlin 默认所有的参数和变量都不可为空，所以这里传入的 Study 参数也一定不会为空，可以放心地调用它的任何函数。

Kotlin 提供了另外一套可为空的类型系统，就是在类名的后面加上一个问号。比如，`Int` 表示不可为空的整型，而 `Int?` 就表示可为空的整型；`String` 表示不可为空的字符串，而 `String?` 就表示可为空的字符串。

```
val a: Int? = null  //编译不报错
val b: Int = null   //编译报错,不允许为空
val c: Int = a      //编译报错,因为a可能为空
```

使用可为空的类型系统时，需要在编译时期就把所有的空指针异常都处理掉。

3.判空辅助工具

Kotlin提供了一系列的辅助工具，使开发者能够更轻松地进行判空处理。

- **1. `?.` 操作符**

`?.` 操作符表示当对象不为空时正常调用相应的方法，当对象为空时则什么都不做。比如：

```
if (a != null) {
    a.doSomething()
}
```

这段代码使用 `?.` 操作符就可以简化成：

```
a?.doSomething()
```

- **2. `?:`操作符**

`?:` 操作符表示如果左边表达式的结果不为空就返回左边表达式的结果，否则就返回右边表达式的结果。比如

```
val c = if (a ! = null) {
    a
} else {
    b
}
```

这段代码的逻辑使用 `?:` 操作符就可以简化成：

```
val c = a ?: b
```

- **3.!!操作符**

如果确信某个值确定不为空,那么可以使用 `!!` ,添加在对象后面,来强制忽略 Kotlin 的空类型检查,比如以下代码:

```
val a: Int? = null
val c: Int = a!!
```

这是一种有风险的写法,意在告诉 Kotlin,我确信这里不会为空,不需要空指针检查,但是如果出现问题,那么会抛出 `kotlin.KotlinNullPointerException`

- **4. 判空小技巧**

结合使用`?.`操作符和 `let` 函数也可以对多次重复调用的某个变量统一进行判空处理：

```
fun doStudy(study: Study?) {
    study?.let {
        it.readBooks()
        it.doHomework()
    }
}
```

- **5.as?操作符**

其实这里是指`as`操作符，表示类型转换，如果不能正常转换的情况下使用`as?`操作符。当使用`as`操作符的使用不能正常的转换的情况下会抛出`类型转换（ClassCastException）异常`，而使用`as?`操作符则会返回`null`,但是不会抛出异常

1. 使用 as

例：

```
// 会抛出ClassCastException异常
val num1 : Int? = "Koltin" as Int
println("nun1 = $num1")
```

2. 使用as?

例：

```
val num2 : Int? = "Koltin" as? Int
println("nun2 = $num2)
```

输出结果为：

```
num2 = null
```

9.Lambda

1.Lambda的定义

Lambda就是一小段可以作为参数传递的代码。正常情况下，我们向某个函数传参时只能传入变量，而借助Lambda却允许传入一小段代码。

我们来看一下Lambda表达式的语法结构：

```
{参数名1: 参数类型, 参数名2: 参数类型 -> 函数体}
```

首先最外层是一对大括号，如果有参数传入到Lambda表达式中的话，我们还需要声明参数列表，参数列表的结尾使用一个`->`符号，表示参数列表的结束以及函数体的开始，函数体中可以编写任意行代码，并且最后一行代码会自动作为Lambda表达式的返回值。

2.集合的函数式API

集合中的 map 函数是最常用的一种函数式API，它用于将集合中的每个元素都映射成一个另外的值，映射的规则在 Lambda 表达式中指定，最终生成一个新的集合。比如，这里我们希望让所有的水果名都变成大写模式，就可以这样写：

```
fun main() {
    val list = listOf("Apple", "Banana", "Orange", "Pear", "Grape",
"Watermelon")
    val newList = list.map({ fruit: String -> fruit.toUpperCase() })
    for (fruit in newList) {
        println(fruit)
    }
}
```

- 当 `Lambda` 参数是函数的最后一个参数时，可以将 `Lambda` 表达式移到函数括号的外面。
- 如果 `Lambda` 参数是函数的唯一一个参数的话，还可以将函数的括号省略。
- 由于 `Kotlin` 拥有出色的类型推导机制，`Lambda` 表达式中的参数列表其实在大多数情况下也不必声明参数类型。
- 当 `Lambda` 表达式的参数列表中只有一个参数时，也不必声明参数名，而是可以使用it关键字来代替。

因此，`Lambda`表达式的写法可以进一步简化成如下方式：

```
val newList = list.map { it.toUpperCase() }
```

3.Java函数式API

如果我们在 Kotlin 代码中调用了一个Java方法，并且该方法接收一个 Java 单抽象方法接口参数(即 SAM,Single Abstract Method)，就可以使用函数式 API。Java 单抽象方法接口指的是接口中只有一个待实现方法，如果接口中有多个待实现方法，则无法使用函数式API。

举个例子，Android中有一个极为常用的点击事件接口 `OnClickListener`，其定义如下：

```
public interface OnClickListener {
     void onClick(View v);
}
```

可以看到，这是一个单抽象方法接口。假设现在我们拥有一个按钮button的实例，就可以使用函数式API的写法来注册这个按钮的点击事件：

```
button.setOnClickListener { v ->
    //在这处理点击事件
}
//或者
button.setOnClickListener {
    //在这处理点击事件
}
```

10.高阶函数

1.高阶函数的定义

如果一个函数接收另一个函数作为参数,或者返回值的类型是另一个函数,那么该函数就称为高阶函数.

在 Java 中,我们有像整形,布尔型等字段类型,而 Kotlin 添加了函数类型的概念.如果我们将这种函数类型,添加到一个函数的参数声明和返回值声明中,那么这个函数就是高阶函数.

高阶函数定义规则如下:

```
(String, Int) -> Unit
```

既然定义了一个函数类型,那么最关键的就是要声明该函数接收什么参数,以及它的返回值是什么.因此, `->` 左边的部分就是声明该函数接收什么类型的参数,多个参数用逗号隔开,不过没有任何参数,只需要写空括号就可以了;而 `->` 右边的部分,用于声明该函数的返回值类型是什么,如果没有返回值就使用 `Unit`, 相当于 Java 中的 `void`,注意这里的 `Unit` 不能忽略,函数类型声明总是需要一个显式地返回类型.

现在我们将上述的函数类型添加到某个函数的声明或者返回值上,那么这个函数就是一个高阶函数了,如下所示:

```
fun example(func: (String, Int) -> Unit) {
    func("小明", 18)
}
```

我们来定义一个符合上面函数形参的函数:

```
fun test(name: String, age: Int): Unit {
    println("$name 今年$age 岁")
}
```

`test()` 函数接收一个 String 类型和 Int 类型的变量,返回值是 `Unit`,符合 `example()` 的参数要求,下面我们来调用 `example()`:

```
example(::test)  //执行结果,会打印: 小明 今年18 岁
```

我们使用了 `::` 的写法,这是一种函数引用的方式,表示将 `test()` 函数(的对象)作为参数传递给 `example()` 函数,然后在 `example()` 中, `test()` 接收到了两个参数,并且执行了方法体中的代码

2.使用Lambda调用高阶函数

我们定义如下高阶函数:

```
fun num1AndNum2(num1: Int, num2: Int, operation: (Int, Int) -> Int): Int {
    val result = operation(num1, num2)
    return result
}
```

参数1和参数2无需过多解释,传入了两个整形值,第三个参数是接收一个两个整形参数并且返回值也是整型值的函数类型参数,在 `num1AndNum2()` 函数中,我们没有进行任何的运算操作,而是将 `num1` 和 `num2` 参数传给了第三个函数类型参数,并获取他的返回值,最终得到返回值类型.

高阶函数已经定义好了,我们该如何调用呢?由于 `num1AndNum2()` 函数接收一个函数类型的参数,因此我们得先定义与其函数类型匹配的的函数,于是我们添加如下代码:

```
fun plus(num1: Int, num2: Int): Int {
    return num1 + num2
}

fun minus(num1: Int, num2: Int): Int {
    return num1 - num2
}
```

有了上述函数之后,我们就可以调用 `num1AndNum2()` 函数了,如下代码:

```
fun main(){
    val num1 = 100
    val num2 = 80
    val result1 = num1AndNum2(num1, num2, ::plus)
    val result2 = num1AndNum2(num1, num2, ::minus)
    println("result1 is $result1")  //result1 is 180
    println("result2 is $result2")  //result2 is 20
}
```

打印结果和我们预期的结果是一致的.

使用这种函数引用(Function Reference)的方式虽然能够正常工作,但是每次调用任何高阶函数的时候,还得先定义一个与其类型匹配的的函数,太复杂了.

Kotlin 还支持使用其他方式来调用高阶函数,比如 Lambda 表达式,匿名函数,成员引用等,其中,Lambda表达式是最常见最普遍的调用方式.

上述代码使用 Lambda 表达式的写法来实现的话,代码如下:

```
fun main() {
    val num1 = 100
    val num2 = 80
    val result1 = num1AndNum2(num1, num2) { n1, n2 ->
        n1 + n2
    }
    val result2 = num1AndNum2(num1, num2) { n1, n2 ->
        n1 - n2
    }
    println("result1 is $result1")  //result1 is 180
    println("result2 is $result2")  //result2 is 20
}
```

可以看到, Lambda 同样可以完整的表达一个函数的参数声明和返回值声明(Lambda 最后一行会自动作为返回值),而且写法更加精简.

现在,我们可以把 `plus()` 和 `minus()` 函数删掉了,重新运行一下代码,会发现结果一模一样

3.完整的高阶函数

我们定义如下代码:

```
fun StringBuilder.build(Block: StringBuilder.() -> Unit): StringBuilder {
    Block()
    return this
}
```

我们给 `StringBuilder` 定义了一个 `build` 扩展函数,这个扩展函数接收一个函数类型的参数,并且 `build` 函数的返回值类型也是 `StringBuilder`.

注意:这个函数类型参数的声明方式和我们上面定义的语法有所不同:它在函数类型的前面加上了 `StringBuilder.` 的语法结构.其实这才是定义高阶函数完整的语法规则,在函数类型的前面加上 `ClassName.` 就表示这个函数类型是定义在哪个类当中的.

那么将函数类型定义到 `StringBuilder`类当中,有什么好处呢? 好处就是当我们调用 `build` 函数的时候,传入的 Lambda 表达式将自动拥有 `StringBuilder`的 上下文.

我们定义了一个吃水果的函数,如果用 Java 的思维,代码如下:

```
fun eat() {
    val list = listOf("apple", "banana", "orange", "pear", "grape")
    val sb = StringBuilder()
    sb.append("Start eating fruits.\n")
    for (fruit in list) {
        sb.append(fruit).append("\n")
    }
    sb.append("Ate all fruits.")
    println(sb.toString())
}
```

下面,我们用刚才定义好的 `build` 函数来简化 `StringBuilder` 构建字符串的方式了:

```
fun eat() {
    val list = listOf("apple", "banana", "orange", "pear", "grape")
    val result = StringBuilder().build {
        append("Start eating fruits.\n") //自动拥有 StringBuilder()的上下文,即新创建的 StringBuilder 对象,所以可以this.append,this可省略
        for (fruit in list) {
            append(fruit).append("\n")
        }
        append("Ate all fruits.")
    }
    println(result.toString())
}
```

如果构建的字符串越长,越复杂,那么使用这种方式,就更简洁,方便.

可以看到, `build` 函数的用法和 apply 的用法基本上一模一样,只不过我们编写的 build 函数只能用到 `StringBuilder` 类上,而 apply 可以使用在所有类中