kotlin

- 简介
  - 介绍
    - Kotlin 是一个基于JVM 的新的编程语言，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已正式成为Android官方支持开发语言
  - 学习链接
    - https://www.cnblogs.com/Jetictors/articles/9227498.html
      Kotlin教程——史上最全面、最详细的…
    - https://www.kotlincn.net/
      Kotlin 语言中文站
- 变量、常量、注释
  - 变量
    - var：用此关键字声明的变量表示可变变量，即可读且可写，相当于Java中普通变量。
      - //立即初始化
        var var_a: Int = 10

        //推导出类型
        var var_b = 5

        //没有初始化的时候，必须声明类型
        var var_c: Float
        var_c = 12.3f
    - val：用此关键字声明的变量表示不可变变量，即可读且不可写，相当于Java中用final修饰的变量。
    - 类中声明变量
      - 必须初始化，如果不初始化，需使用lateinit关键字。
      - class Test1{
        // 定义属性

```
        var var_a : Int = 0
        val val_a : Int = 0
        // 初始化
        init {
            var_a = 10
            // val_a = 0 为val类型不能更改。
        }
    }
```

- 声明可空变量
  - 可空变量的特点
    - 1、在声明的时候一定用标准的声明格式定义，不能用可推断类型的简写。
      2、变量类型后面的?符号不能省略。
      3、其初始化的值可以为null或确定的变量值。
  - var/val 变量名 : 类型? = null/确定的值
- 后期初始化
  - 后期初始化属性的特点
    - 1、使用lateinit关键字。
      2、必须是可读且可写的变量，即用var声明的变量。
      3、不能声明于可空变量。
      4、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。
      5、声明后，在使用该变量前必须赋值，不然会抛出UninitializedPropertyAccessException异常。
    - // 声明组件
      private lateinit var mTabLayout : TabLayout

      // 会报错。因为不能用于基本数据类型。
      lateinit var a : Int

      // 赋值
      mTabLayout = find(R.id.home_tab_layout)

      // 使用
      mTabLayout.setupWithViewPager(mViewPager)
- 延迟初始化
  - 延迟初始化属性的特点
    - 1、使用lazy{}高阶函数，不能用于类型推断。且该函数在变量的数据类型后面，用by链接。
      2、必须是只读变量，即用val声明的变量。
      3、指当程序在第一次使用到这个变量（属性）的时候再初始化。
    - // 声明一个延迟初始化的字符串数组变量

```
    private val mTitles : Array<String> by lazy {
        arrayOf(
                ctx.getString(R.string.tab_title_android),
                ctx.getString(R.string.tab_title_ios),
                ctx.getString(R.string.tab_title_h5)
        )
    }

    // 声明一个延迟初始化的字符串
    private val mStr : String by lazy{
        "我是延迟初始化字符串变量"
    }
```

- 常量
  - 定义
    - 在val关键字前面加上const关键字。
  - 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。

```
// 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"
    }
}

fun main(args: Array<String>) {
    println("NUM_A => $NUM_A")
    println("NUM_B => ${TestConst.NUM_B}")
    println("NUM_C => ${TestClass.NUM_C}")
}
```

- 每一行代码的结束可以省略掉分号;
- 注释
  - 单行注释
    - // 这是一个单行注释
  - 多行注释（块注释）
    - /* 这是一个多行
      注释。 */
  - 多行注释嵌套
    - /*
      第一层块注释
      /*
      第二层块注释
      /*
      第三层快注释
      */
      */
      */
  - 类注释、方法注释
    - /**
      * 3. 方法的注释（同java一样）
      */
- 数据类型 73
- 控制语句
  - 条件控制
    - if语句
      - 1. 除了能实现Java写法外，可以实现表达式（实现三元运算符），及作为一个块的运用。
        2. 可以有返回值的，if语句每一个条件中最后一行代码的返回值。
      - 3种写法
        - 1、传统写法
          - 例子

```
fun main(args: Array<String>) {
    var numA = 2
    if (numA == 2){
        println("numA == $numA ")
    }else{
        println("numA == $numA ")
    }
}
```

```
//结果
numA == 2
```

- 2、三元运算符
  - 1. 在Kotlin中其实是不存在三元运算符(condition ? then : else)这种操作的。
    2. 那是因为if语句的特性(if表达式会返回一个值)故而不需要三元运算符。
  - 例子

```
// 在Java中可以这么写，但是Kotlin中直接会报错。
// var numB: Int = (numA > 2) ? 3 : 5

// kotlin中直接用if..else替代。例：
var numB: Int = if ( numA > 2 ) 3 else 5 // 当numA大于2时输出numB的值为3，反之为5
println("numB = > $numB")
```

- 3、块结构
  - 1. 表达式的结果为块的值。
  - 例子

```
fun main(args: Array<String>) {
    val num1 : Int = 2
    val num2 : Int = 3
    val value : Int = if(num1 > num2){
        num1
    }else {
        num2
    }
    println("$value")
}

//结果
3
```

- when语句
  - 1. 在Kotlin中已经废除掉了Java中的switch语句，而新增了when(exp){}语句。
    2. when语句不仅可以替代掉switch语句，而且比switch语句更加强大。
  - 6种用法
    - 1、when语句实现switch语句功能
      - 例子

```
fun main(args: Array<String>) {
    when(3){
        1 -> {
            println("1")
        }
        2 -> println("2")
        3 -> println("3")
        else -> {
```

```
            println("0")
        }
    }
}

//结果
3
```

- 2、和逗号结合使用
  - 例子

```
fun main(args: Array<String>) {
    when(1){
        // 即x = 1,2,3时都输出1。
        1 , 2 , 3 -> {
            println("1")
        }
        else -> {
            println("0")
        }
    }
}

//结果
1
```

- 3、条件可以使用任意表达式，不仅局限于常量
  - 例子

```
fun main(args: Array<String>) {
    val num:Int = 2
    when(num > 1){
        true -> {
            println("num > 1")
        }
        false ->{
            println("num <= 1")
        }
    }
}

//结果
num > 1
```

- 4、检查值是否存在于集合或数组中
  - 1. 操作符：in在、!in不在。
    2. 只适用于数值类型。
  - 例子

```
fun main(args: Array<String>) {
    val arrayList = arrayOf(1, 2, 3, 4, 5)
    when (5) {
        in arrayList -> {
```

```
            println("存在arrayList中")
        }
        in 1..6 -> println("存在1到6中")
        else ->{
            println("不存在")
        }
    }
}

//结果
存在arrayList中
```

- 5、检查值是否为指定类型的值
  - 1. 操作符：is 是、!is不是。
  - 例子

```
when("a"){
    is String -> println("是一个字符串")
    else ->{
      println("不是一个字符串")
    }
 }

//结果
是一个字符串
```

- 6、不带参数的when语句
  - 1. 表示为最简单的布尔表达式。
  - 例子

```
fun main(args: Array<String>) {
    val name="Aom"
    when{
        name.startsWith("A")->println("1")
        name=="Tom" -> println("2")
        name=="Jack" -> println("3")
        else -> println("0")
    }
}

//结果
1
```

- 循环控制
  - for语句
    - 1. Kotlin废除了Java中的for(初始值;条件；增减步长)这个规则。但是Kotlin中对于for循环语句新增了其他的规则，来满足刚提到的规则。
      2. for循环提供迭代器用来遍历任何东西。
      3. for循环数组被编译为一个基于索引的循环，它不会创建一个迭代器对象。
    - 4个新增规则

- 1、递增
  - 1. 关键字：until
    2. 范围：until[n,m) => 即大于等于n,小于m
  - 例子

```
fun main(args: Array<String>) {
    // 循环5次，且步长为1的递增
    for (i in 0 until 5){
        print("i => $i \t")
    }
}

//结果
i => 0    i => 1    i => 2    i => 3    i => 4
```

- 2、递减
  - 1. 关键字：downTo
    2. 范围：downTo[n,m] => 即小于等于n,大于等于m ,n > m
  - 例子

```
fun main(args: Array<String>) {
    // 循环5次，且步长为1的递减
    for (i in 5 downTo 0){
        print("i => $i \t")
    }
}

//结果
i => 5    i => 4    i => 3    i => 2    i => 1    i => 0
```

- 3、' .. '
  - 1. 使用符号('..')。
    2. 表示递增的循环的另外一种操作。
    3. 范围：..[n,m]=> 即大于等于n，小于等于m。
    4. 和until的区别，一是简便性，二是范围的不同。
  - 例子

```
fun main(args: Array<String>) {
    // 循环5次，且步长为1的递减
    for (i in 1..5){
        print("i => $i \t")
    }
}

//结果
i => 1    i => 2    i => 3    i => 4    i => 5
```

- 4、步长
  - 例子

```
fun main(args: Array<String>) {
```

```
    // 循环5次，且步长为1的递减
    for (i in 1..5 step 2){
        print("i => $i \t")
    }
}

//结果
i => 1      i => 3      i => 5
```

- 5种迭代使用方式
  - 1、遍历字符串
    - 例子

```
fun main(args: Array<String>) {
  for (i in "abc"){
    print("$i \t")
  }
}

//结果
a  b  c
```

  - 2、遍历数组
    - 例子

```
fun main(args: Array<String>) {
  val arrayListOne = arrayOf(10,20,30)
  for (i in arrayListOne){
    print("$i \t")
  }
}

//结果
10  20  30
```

  - 3、使用数组的indices属性遍历
    - 例子

```
fun main(args: Array<String>) {
  val arrayListTwo = arrayOf(1,3,5)
  for (i in arrayListTwo.indices){
    print("${arrayListTwo[i]} \t")
  }
}

//结果
1  3  5
```

  - 4、使用数组的withIndex()方法遍历
    - 例子

```
fun main(args: Array<String>) {
  val arrayListTwo = arrayOf(1,3,5)
```

```
    for ((index,value) in arrayListTwo.withIndex()){
      println("$index \t $value")
    }
  }

  //结果
  0    1
  1    3
  2    5
```

- 5、使用列表或数组的扩展函数遍历
  - 1. 数组或列表有一个成员或扩展函数iterator()实现了Iterator<T>接口，且该接口提供了next()与hasNext()两个成员或扩展函数。
    2. 其一般和while循环一起使用。
  - 例子

```
fun main(args: Array<String>) {
  val arrayListThree = arrayOf(2,'a')
  val iterator: Iterator<Any> = arrayListThree.iterator()

  while (iterator.hasNext()){
    println(iterator.next())
  }
}

//结果
2
a
```

- while语句
  - 1. 与Java中的while循环一样。
    2. 格式
    while(exp){ 其中exp为表达式
        ...
    }
  - 例子

```
fun main(args: Array<String>) {
  var num = 1
  while (num < 3){
    println("num => $num")
    num++
  }
}

//结果
num => 1
num => 2
```

- do...while语句
  - 1. 与Java中的do...while循环一样，最少执行一次。

2. 格式
do(exp){ // 其中exp为表达式
    ...
}(while)

- 例子

```
fun main(args: Array<String>) {
  var num = 3
  do {
    println("num => $num")
    num++
  }while (num < 3)
}

//结果
num => 3
```

- 跳转语句
  - return语句
    - 1. 默认情况下，从最近的封闭函数或匿名函数返回。
    - 例子

```
fun main(args: Array<String>) {
  returnExample()
}
fun returnExample(){
  val str: String = ""
  if (str.isBlank()){
    println("我退出了该方法")
    return
  }
  println("这里没打印")
}

//结果
我退出了该方法
```

  - break语句
    - 1. 终止最近的闭合循环。
    - 例子

```
fun main(args: Array<String>) {
  for (i in 1..5){
    println("i => $i")
    if (i == 2){
      println("我在 i = $i 时退出了循环")
      break
    }
  }
}
```

```
//结果
i => 1
i => 2
我在 i = 2 时退出了循环
```

- continue语句
  - 1. 前进到最近的封闭循环的下一个步骤(迭代)。
  - 例子

```
fun main(args: Array<String>) {
  for (i in 1..5){
    if (i == 2){
      println("我跳过了第 $i 次循环")
      continue
    }
    println("i => $i")
  }
}

//结果
i => 1
我跳过了第 2 次循环
i => 3
i => 4
i => 5
```

函数

函数的声明及基本使用

函数的声明

- 1. Kotlin中的函数声明关键字为：fun。
  2. 定义格式为：可见性修饰符 fun 函数名(参数名 ： 类型,...) : 返回值{}。
  4. Kotlin中默认为public可见性修饰符。
  5. ()圆括号必须存在，即使是没有参数的情况下。
  6. {}大括号必须存在，即使是没有函数体的时候，不过在Kotlin中有一个特例就是，函数具备返回值的时候，如果只用一个表达式就可以完成这个函数，则可以使用单表达式函数。
  7. 在函数没有返回值时可以省略其返回值。
- 定义一个最基本的函数

```
fun basis(){
    ...
}
```

成员函数

- 1. 成员函数是指在类或对象中的内部函数。
- 例子

```
class Test{
    fun foo(){}
```

```
}
```

- 函数的使用
  - 1. 函数的使用分为两种：
    （1）普通的使用。
    （2）成员函数的使用。
  - 例子

```
// 普通的使用
basis()
// 如果函数有返回值
val x = basis()

// 成员函数的使用：先初始化对象，在根据对象使用`中缀符号(.)`调用其成员函数
Test().foo()
// 如果函数有返回值
val x = Test().foo()
```

- 函数的返回值
  - 1. 在Kotlin中，函数的返回值类型可以分为：
    （1）Unit类型：该类型即无返回值的情况，可以省略。
    （2）其他类型： 显示返回类型的情况。
  - Unit类型
    - 例子

```
fun unitFun() : Unit{
    println("我是返回值为Unit的函数，Unit可省略")
    return

    // return Unit 可省略
    // 或者 return  可省略
}

等价于

fun unitFun(){
    println("我是返回值为Unit的函数，Unit可省略")
}
```

  - 其他类型
    - 1. 返回值类型不能省略。
    - 例子

```
fun returnFun() : Int{
    return 2
}
```

- 函数的参数
  - 具有参数的函数定义
    -

1. 定义一个具有参数的函数，使用Pascal 表示法定义，即为name : type。
2. 其中的参数必须具有显示的参数类型，并且参数与参数之间用逗号(,)隔开。

- 例子

```
fun funArgs(numA : Int, numB : Float){
    println("numA = $numA \t numB = $numB")
}

fun main(args: Array<String>) {
    funArgs(1,10f)
}
```

- 默认参数值
  - 1. 使函数中的参数具有默认值，这样在使用该函数的时候，可以省略一部分参数，可以减少函数的重载。
  - 例子

```
fun defArgs(numA : Int , numB : Boolean = false){
    println("numA  =  $numA \t numB = $numB ")
}

fun main(args: Array<String>) {
    // 默认参数的函数使用
    defArgs(1)
    defArgs(1,true)
}

//结果
numA = 1       numB = false
numA = 1       numB = true
```