https://blog.csdn.net/qq\_44973159/category\_9946933.html

语言特点及概述

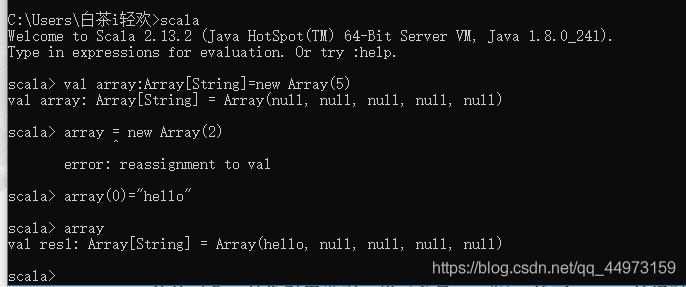
Scala 是一门满足现代软件工程师需求的语言；它是一门静态类型语言，支持混合范式；它也是一门运行在 JVM 之上的语言，语法简洁、优雅、灵活。Scala 拥有一套复杂的类型系统，Scala方言既能用于编写简短的解释脚本，也能用于构建大型复杂系统。这些只是它的一部分特性，下面就是其特性：

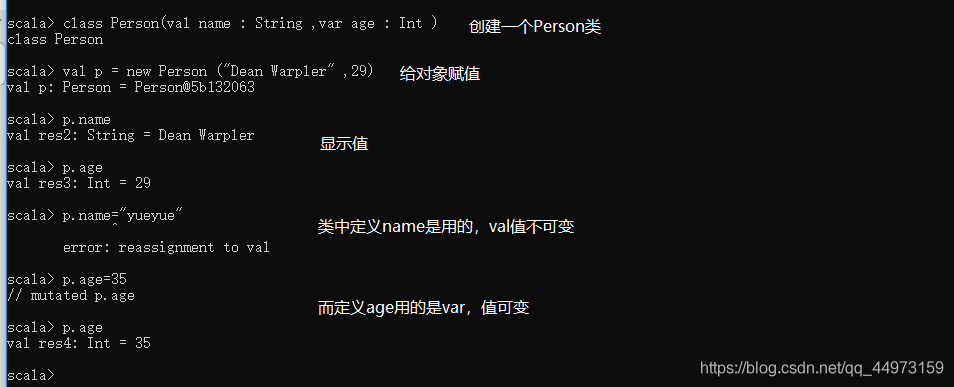
* **运行在 JVM 和 JavaScript 之上的语言**  
  Scala 不仅利用了 JVM 的高性能以及最优化性，Java 丰富的工具及类库生态系统也为其所用。不过 Scala 并不是只能运行在 JVM 之上！
* **静态类型**  
  在 Scala 语言中，静态类型（static typing）是构建健壮应用系统的一个工具。Scala 修正了 Java 类型系统中的一些缺陷，此外通过类型推演（type inference）也免除了大量的冗余代码。
* **混合式编程范式——面向对象编程**  
  Scala 完全支持面向对象编程（OOP）。Scala 引入特征（trait）改进了 Java 的对象模型。trait 能通过使用混合结构（mixin composition）简洁地实现新的类型。在 Scala 中，一切都是对象，即使是数值类型。
* **混合式编程范式——函数式编程**  
  Scala 完全支持函数式编程（FP），函数式编程已经被视为解决并发、大数据以及代码正确性问题的最佳工具。使用不可变值、被视为一等公民的函数、无副作用的函数、高阶函数以及函数集合，有助于编写出简洁、强大而又正确的代码。
* **复杂的类型系统**  
  Scala 对 Java 类型系统进行了扩展，提供了更灵活的泛型以及一些有助于提高代码正确性的改进。通过使用类型推演，Scala 编写的代码能够和动态类型语言编写的代码一样精简。
* **简洁、优雅、灵活的语法**  
  使用 Scala 之后，Java 中冗长的表达式不见了，取而代之的是简洁的 Scala 方言。Scala提供了一些工具，这些工具可用于构建领域特定语言（DSL），以及对用户友好的 API接口。
* **可扩展的架构**  
  使用 Scala，你能编写出简短的解释性脚本，并将其粘合成大型的分布式应用。以下四种语言机制有助于提升系统的扩展性：1) 使用 trait 实现的混合结构；2) 抽象类型成员和泛型；3) 嵌套类；4) 显式自类型（self type）。

**Scala的变量声明和数据类型详解**

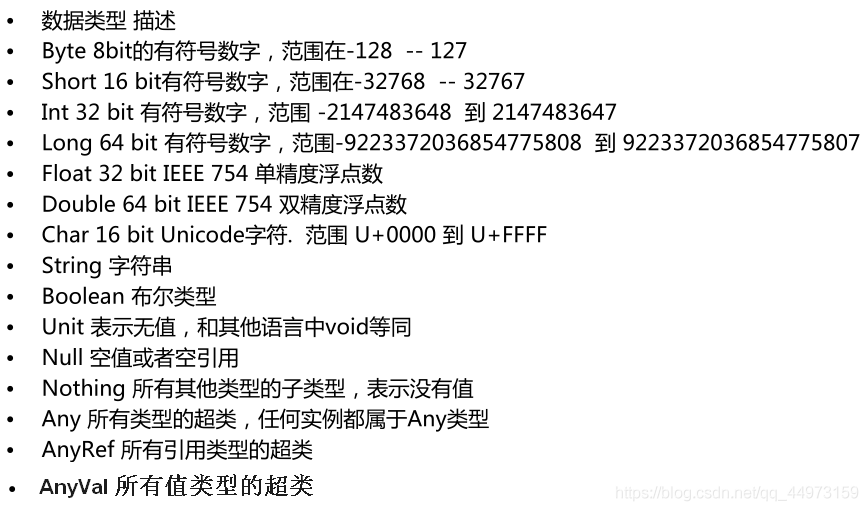
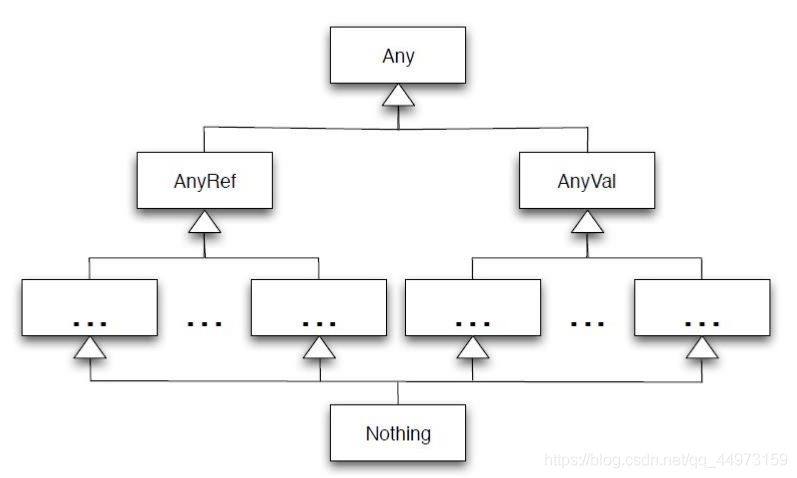
变量声明

在声明变量时，Scala 允许你决定该变量是不可变（只读）的，还是可变的（读写）。如下所示，不可变的“变量”用 val 关键字声明：  
val array: Array[String] = new Array(5)

Scala 的大部分变量事实上是指向堆内存对象的引用，这一点与 Java 一致。所以，以上代码中的 array 也是一个引用，它不能指向其他 Array，但所指向的 Array 中的元素是可变的，如下所示：  
  
一个 val 变量在声明时必须被初始化。  
类似地，一个可变变量用关键字 var 来声明。尽管由于该变量是可变变量，声明后可以再次对其赋值，也必须在声明的同时立即初始化：  
在这里插入图片描述  
在 Java 中， 所谓的原生类型， 即 char、byte、short、int、long、float、double 和boolean，与其他引用类型有着本质的不同。这些类型确实既不是对象，也没有引用，是“原始”值。Scala 尽力使其面向对象特性更加一致，因此这些类型在 Scala 中是包含有方法的对象，就像引用类型一样。然而，Scala 编译时将这些类型尽可能地转为原生类型，使你可以得到原生类型的运行效率.

用 val 和 var 声明变量时必须初始化这一规则，但存在少数例外情况。例如，这两个关键字均可以用在构造函数的参数中，这时候变量是该类的一个属性，因此显然不必在声明时进行初始化。此时如果用 val 声明，该属性是不可变的；如果用 var 声明，则该属性是可变的、  
  
var 和 val 关键字只标识引用本身是否可以指向另一个不同的对象，它们并未表明其所引用的对象是否可变。

为了减少可变性引起的bug，应该尽可能地使用不可变变量Val。

数据类型  
  
其中Any和AnyRef 、AnyVal 、Nothing 之间的关系如下图：  
  
空值之间的区别  
  
当给变量定义的时候，不给出类型就是一个Nothing  


代码：

package com.lzq.bilibili

//在Scala当中驼峰命名法，在Scala当中定义的类可以传递参数,有了参数就有了默认的构造,传参一定要指定类型,传递参数的时候会自动getset

//在类当中重写构造，构造中第一行需要调用默认构造

//在Scala当中，new Person时 出来方法不执行【除了构造方法】其他都执行

//在同一个文件当中class名称和Object名称一样时，这个类叫做这个对象的伴生类，这个对象叫做这个类的伴生对象，互相之间可以互相访问私有变量

class Person(pname: String, page: Int) {

val name = pname

val age = page

var sex = 'M'

println("=========Begin========")

def Say() = {

print("hello " + DataType.name)

}

def this(qname: String, qage: Int, qsex: Char) {

this(qname, qage)

this.sex = qsex

}

println("=========End========")

}

//object相当于java当中的单例，定义的全是静态的。相当于java的工具类 Object当中不能够传递参数

object DataType {

//object会在class之前进行调用

println("###### Object ######")

val name = "yueyue"

def apply(i: Int, s: String) = {

println("i = " + i + ", s =" + s)

}

def main(args: Array[String]): Unit = {

// if else分支语句

// val age = 15

// if (age < 18) {

// println("未成年")

// } else if (age == 18) {

// println("刚成年")

// } else {

// println("已经成年")

// }

// for循环

// val a = 1 to (10,2)

// println(a)

// for(i<- 1 to 10){

// println(i)

// }

// 嵌套for循环 方式一

// for (i <- 1 to 9) {

// for (j <- 1 to 9) {

// if (i >= j) {

// print(i + "\*" + j + "=" + i \* j + " ")

// }

// }

// println()

// }

//方式二

// for (i <- 1 to 9;j <- 1 to 9) {

// if (i >= j) {

// print(j + "\*" + i + "=" + i \* j + " ")

// }

// if(i==j ){

// println()

// }

// }

//在for语句当中添加条件

// for ( i <- 1 to 1000 if(i> 500) if(i%3 ==0)){

// print(i+" ")

// }

//while循环

//var i = 1

// while(i<10){

// println(s"第 $i 次")

// i=i+1

// //或者 i += 1 但是不能使用 ++

// }

//改成do while循环

// do {

// println(s"第 $i 次")

// i = i + 1

// }

// while (i < 10)

//直接使用变量获取值

// val res = for ( i <- 1 to 100 if(i> 50) if(i%3 ==0)) yield i

// println(res)

//DataType(1000, "String s")

//对Object进行传参，需要调用apply方法，否则是不可以传参的

//实例化这个类 Person p = new Person()

//val person = new Person(pname = "月月鸟", page = 25)

// println(person.name)

// println(person.age)

// person.Say()

// val person1 = new Person(qname = "yueyue1", qage = 30, qsex = 'M')

// print(person1.sex)

// val a = 100

// println(a)

// var b =200

// b=300

// println(b)

}

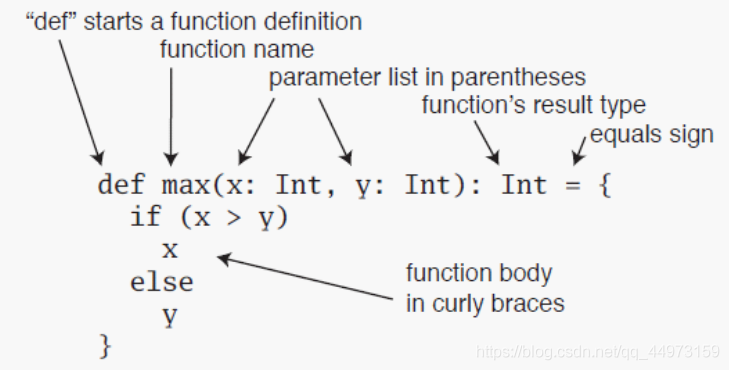
}

总结

* 建议类名首字母大写 ，方法首字母小写，类和方法命名建议符合驼峰命名法。
* scala 中的object是单例对象，相当于java中的工具类，可以看成是定义静态的方法的类。object不可以传参数。另：Trait不可以传参数
* scala中的class类默认可以传参数，默认的传参数就是默认的构造函数。重置必须要调用默认的构造函数。
* class 类属写构造函数的时候，性自带getter ，setter方法。
* 使用object时，不用new,使用class时要new ,并且new的时候，class中除了方法不执行，其他都执行。
* 如果在同一个文件中，object对象和class类的名称相同，则这个对象就是这个类的伴生对象，这个类就是这个对象的伴生类。可以互相访问私有变量。

# Scala的方法定义与函数详解

➢ **方法定义**

  
1.普通方法

def sum(a: Int, b: Int): Int = {

//return a + b

a + b

}

➢ **注意点**

1. 方法体的返回值可以省略，会自动推断
2. 方法体最后的返回值可以使用return，如果使用了return那么一定要指定返回类型
3. 方法体中没有用return。默认将方法体当中最后一行计算的结果当做返回值返回，
4. 方法体当中，如果只有一行，大括号可以省略
5. 如果定义方法，省略了方法名称和 = 无论结果是什么，结果丢弃，返回控制Unit

2.递归方法

在声明方法的时候需要给定方法的返回类型：以及其中的if else语句当中只有一条子语句的话，大括号可省略

def recursion(num: Int): Int = {

if (num == 1) 1 else num \* recursion(num - 1)

}

3.参数有默认值的方法

def Default\_parameters(a: Int = 10, b: Int = 20) = {

a + b

}

在调用过程当中如果只想改变一个参数值，重新赋值即可

println(Default\_parameters(b=54))

4.参数可变的方法

如下代码所示：传递参数的时候在后面加上一个\* 表示可传递多个参数，随后对参数进行打印输出可以使用最简单的for循环，foreach循环使用 => 匿名函数。等等

def Variable\_parameters(s: String\*) = {

// =>匿名函数

//s.foreach(ele => {println(ele)})

//当foreach当中的参数只用了一次可以使用下划线 "\_" 进行代替

//s.foreach(println(\_))

//或者再进行省略

s.foreach(println)

// for(ele <- s){

// println(ele)

// }

}

调用函数传递所需参数：

Variable\_parameters("hello","world","changsha")

5.匿名函数

()=>{} 参数类型=>返回类型，自动推断，直接调用即可。

def Anonymous\_function = (a: Int, b: Int) => {

a + b

}

6.嵌套方法

在一个方法当中再定义一个方法，实现嵌套方法。

def nesting(x: Int): Int = {

def nesting1(y: Int): Int = {

if (y == 1) 1 else y \* nesting1(y - 1)

}

}

nesting1(x)

}

7.偏应用函数

参数过多且参数大部分是不变的，就可以使用偏应用函数对其进行简化。

def application(date: Date, log: String): Any = {

println("Date is " + date + ", log is " + log)

}

调用方法：

val date = new Date()

application(date, "a")

application(date, "b")

application(date, "c")

上方调用还是显得有所冗长，直接使用一个方法直接简化：

val date = new Date()

def app=application(date,\_:String)

app("a")

8.1 高阶函数 ：方法的参数是函数

也就是将第一个参数改成一个函数进行传递

//sum函数用于调用高阶函数传递的函数

def sum1(a: Int, b: Int) = {

a + b

}

def parameter\_fun(f: (Int, Int) => Int, s: String) = {

val i: Int = f(100, 200);

i + " " + s

}

调用：

var res= parameter\_fun(sum1,"string")

println(res)

或者使用一个匿名函数。

var res = parameter\_fun((a:Int,b:Int)=>{a\*b},"string")

println(res)

8.2 高阶函数 ：方法的返回是函数

返回类型需要给出，而不能使用推断类型，需要根据情况而定，

def return\_fun(a: String): (String, String) => String = {

def return\_fun1(s1: String, s2: String): String = {

s1 + " " + s2 + " " + a

}

//return\_fun1

return\_fun1 \_ //使用这种方式就不需要给出返回类型

}

方法调用：在函数当中又包含了一个函数。

println(return\_fun("a")("b", "c"))

8.3 高阶函数 ：方法的参数和返回都是函数

这就是前两者的结合：

def neutralization(f: (Int, Int) => Int): (String, String) => String = {

val i: Int = f(1, 2)

def neutralization1(s1: String, s2: String): String = {

s1 + " " + s2 + " " + i

}

neutralization1

}

使用到一个匿名函数进行调用：

val res= neutralization((a,b)=>{a+b})("hello","changsha")

println(res)

9.柯里化函数

是对高阶函数的一个简化

def Currie(a: Int, b: Int)(c: Int, d: Int): Int = {

a + b + c + d

}

println(Currie(1, 2)(4, 3))

# Scala字符串String详解

string操作方法举例

➢ 比较:equals  
➢ 比较忽略大小写:equalsIgnoreCase  
➢ indexOf：如果字符串中有传入的assci码对应的值，返回下标

object String {

def main(args: Array[String]): Unit = {

val s="yueyue"

val s1="YUEYUE"

println(s.equals(s1)) //返回false

println(s.equalsIgnoreCase(s1)) //返回true

println(s.indexOf(101)) //101对应的是字母e,而e对应的索引为2

}

}

而所有String相关的方法见下：

char charAt(int index)

返回指定位置的字符 从0开始

int compareTo(Object o)

比较字符串与对象

int compareTo(String anotherString)

按字典顺序比较两个字符串

int compareToIgnoreCase(String str)

按字典顺序比较两个字符串，不考虑大小写

String concat(String str)

将指定字符串连接到此字符串的结尾

boolean contentEquals(StringBuffer sb)

将此字符串与指定的 StringBuffer 比较。

static String copyValueOf(char[] data)

返回指定数组中表示该字符序列的 String

static String copyValueOf(char[] data, int offset, int count)

返回指定数组中表示该字符序列的 String

boolean endsWith(String suffix)

测试此字符串是否以指定的后缀结束

boolean equals(Object anObject)

将此字符串与指定的对象比较

boolean equalsIgnoreCase(String anotherString)

将此 String 与另一个 String 比较，不考虑大小写

byte getBytes()

使用平台的默认字符集将此 String 编码为 byte 序列，并将结果存储到一个新的 byte 数组中

byte[] getBytes(String charsetName

使用指定的字符集将此 String 编码为 byte 序列，并将结果存储到一个新的 byte 数组中

void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)

将字符从此字符串复制到目标字符数组

int hashCode()

返回此字符串的哈希码

16

int indexOf(int ch)

返回指定字符在此字符串中第一次出现处的索引（输入的是ascii码值）

int indexOf(int ch, int fromIndex)

返返回在此字符串中第一次出现指定字符处的索引，从指定的索引开始搜索

int indexOf(String str)

返回指定子字符串在此字符串中第一次出现处的索引

int indexOf(String str, int fromIndex)

返回指定子字符串在此字符串中第一次出现处的索引，从指定的索引开始

String intern()

返回字符串对象的规范化表示形式

int lastIndexOf(int ch)

返回指定字符在此字符串中最后一次出现处的索引

int lastIndexOf(int ch, int fromIndex)

返回指定字符在此字符串中最后一次出现处的索引，从指定的索引处开始进行反向搜索

int lastIndexOf(String str)

返回指定子字符串在此字符串中最右边出现处的索引

int lastIndexOf(String str, int fromIndex)

返回指定子字符串在此字符串中最后一次出现处的索引，从指定的索引开始反向搜索

int length()

返回此字符串的长度

boolean matches(String regex)

告知此字符串是否匹配给定的正则表达式

boolean regionMatches(boolean ignoreCase, int toffset, String other, int ooffset, int len)

测试两个字符串区域是否相等

28

boolean regionMatches(int toffset, String other, int ooffset, int len)

测试两个字符串区域是否相等

String replace(char oldChar, char newChar)

返回一个新的字符串，它是通过用 newChar 替换此字符串中出现的所有 oldChar 得到的

String replaceAll(String regex, String replacement

使用给定的 replacement 替换此字符串所有匹配给定的正则表达式的子字符串

String replaceFirst(String regex, String replacement)

使用给定的 replacement 替换此字符串匹配给定的正则表达式的第一个子字符串

String[] split(String regex)

根据给定正则表达式的匹配拆分此字符串

String[] split(String regex, int limit)

根据匹配给定的正则表达式来拆分此字符串

boolean startsWith(String prefix)

测试此字符串是否以指定的前缀开始

boolean startsWith(String prefix, int toffset)

测试此字符串从指定索引开始的子字符串是否以指定前缀开始。

CharSequence subSequence(int beginIndex, int endIndex)

返回一个新的字符序列，它是此序列的一个子序列

String substring(int beginIndex)

返回一个新的字符串，它是此字符串的一个子字符串

String substring(int beginIndex, int endIndex)

返回一个新字符串，它是此字符串的一个子字符串

char[] toCharArray()

将此字符串转换为一个新的字符数组

String toLowerCase()

使用默认语言环境的规则将此 String 中的所有字符都转换为小写

String toLowerCase(Locale locale)

使用给定 Locale 的规则将此 String 中的所有字符都转换为小写

String toString()

返回此对象本身（它已经是一个字符串！）

String toUpperCase()

使用默认语言环境的规则将此 String 中的所有字符都转换为大写

String toUpperCase(Locale locale)

使用给定 Locale 的规则将此 String 中的所有字符都转换为大写

String trim()

删除指定字符串的首尾空白符

static String valueOf(primitive data type x)

返回指定类型参数的字符串表示形式

# Scala 中的 Array 数组 详解

Scala 语言中提供的数组是用来存储固定大小的同类型元素，

1. 一维数组的声明与遍历

使用以下方式声明数组。

val arr = Array[String]("hello","world")

或者

val arr1 = new Array[Int](5)

使用for循环进行遍历或者foreach方法

for (ele<-arr){

println(ele)

}

或者

arr1.foreach(println)

测试如下图：  
  
2. 二维数组的声明与遍历

创建二维数组

val arr2 = new Array[Array[Int]](3)

对二维数组进行赋值

arr2(0) = Array[Int](4, 5, 6)

arr2(1) = Array[Int](4, 5, 6)

arr2(2) = Array[Int](4, 5, 6)

使用双重for循环进行遍历输出

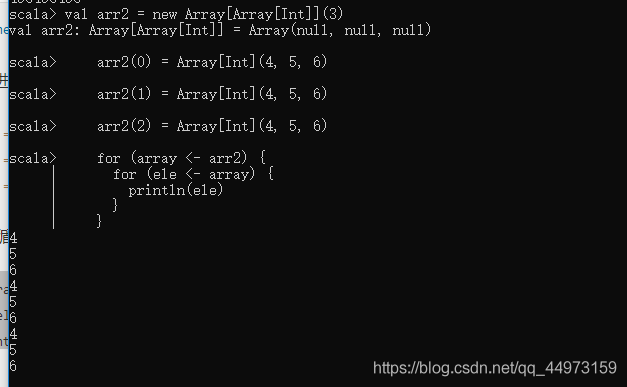
for (array <- arr2) {

for (ele <- array) {

println(ele)

}

}

测试如下图所示：  
  
3. 可变长度数组 ArrayBuffer

使用ArrayBuffer之前要导入相对应的包：

import scala.collection.mutable.ArrayBuffer

创建数组并且对数组进行遍历

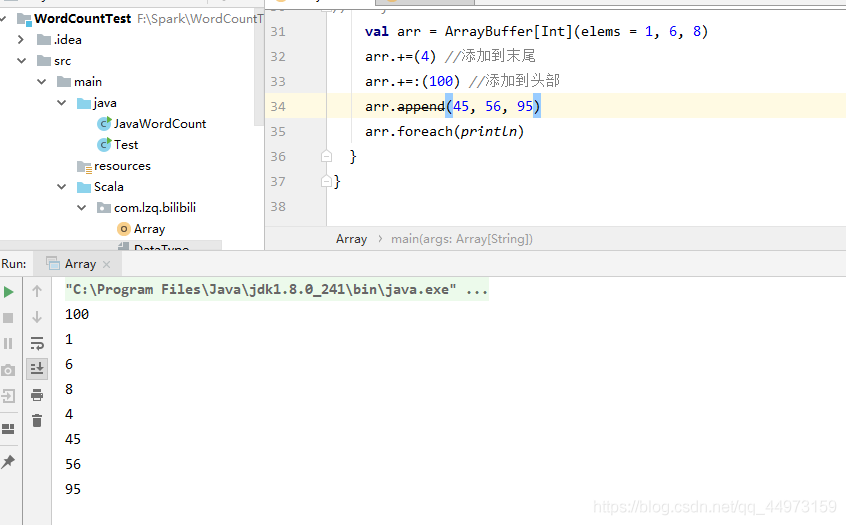
val arr = ArrayBuffer[Int](elems = 1, 6, 8)

arr.+=(4) //添加到末尾

arr.+=:(100) //添加到头部

arr.append(45, 56, 95)

arr.foreach(println)

测试效果如下图所示  
  
数组其余方法详解

| **序号** | **方法** | **详解** |
| --- | --- | --- |
| 1 | def apply( x: T, xs: T\* ): Array[T] | 创建指定对象 T 的数组, T 的值可以是 Unit, Double, Float, Long, Int, Char, Short, Byte, Boolean。 |
| 2 | def concat[T]( xss: Array[T]\* ): Array[T] | 合并数组 |
| 3 | def copy( src: AnyRef, srcPos: Int, dest: AnyRef, destPos: Int, length: Int ): Unit | 复制一个数组到另一个数组上。相等于 Java’s System.arraycopy(src, srcPos, dest, destPos, length)。 |
| 4 | def empty[T]: Array[T] | 返回长度为 0 的数组 |
| 5 | def iterate[T]( start: T, len: Int )( f: (T) => T ): Array[T] | 返回指定长度数组，每个数组元素为指定函数的返回值。以上实例数组初始值为 0，长度为 3，计算函数为a=>a+1：Array.iterate(0,3)(a=>a+1) res1: Array[Int] = Array(0, 1, 2) |
| 6 | def fill [T] ( n: Int )(elem: => T): Array[T] | 返回数组，长度为第一个参数指定，同时每个元素使用第二个参数进行填充。 |
| 7 | def fill[T] ( n1: Int, n2: Int )( elem: => T ): Array[Array[T]] | 返回二数组，长度为第一个参数指定，同时每个元素使用第二个参数进行填充。 |
| 8 | def ofDim[T] ( n1: Int ): Array[T] | 创建指定长度的数组 |
| 9 | def ofDim[T] ( n1: Int, n2: Int ): Array[Array[T]] | 创建二维数组 |
| 10 | def ofDim[T] ( n1: Int, n2: Int, n3: Int ): Array[Array[Array[T]]] | 创建三维数组 |
| 11 | def range( start: Int, end: Int, step: Int ): Array[Int] | 创建指定区间内的数组，step 为每个元素间的步长 |
| 12 | def range( start: Int, end: Int ): Array[Int] | 创建指定区间内的数组 |
| 13 | def tabulate[T] ( n: Int )(f: (Int)=> T): Array[T] | 返回指定长度数组，每个数组元素为指定函数的返回值，默认从 0 开始。以下实例返回 3 个元素：Array.tabulate(3)(a => a + 5) res0: Array[Int] = Array(5, 6, 7) |
| 14 | def tabulate [T] ( n1: Int, n2: Int )( f: (Int, Int ) => T): Array[Array[T]] | 返回指定长度的二维数组，每个数组元素为指定函数的返回值，默认从 0 开始。 |

# Scala 中的 List 列表详解

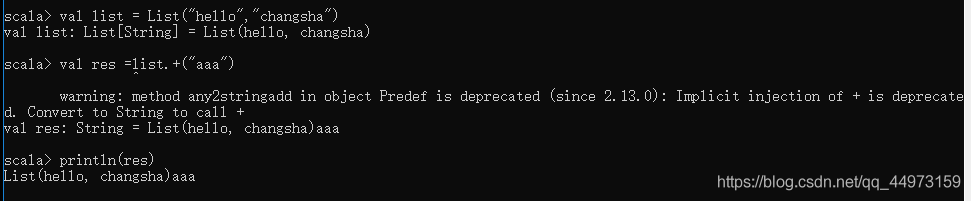
Scala 列表类似于数组，它们所有元素的类型都相同，但是它们也有所不同：列表是不可变的，值一旦被定义了就不能改变，其次列表 具有递归的结构（也就是链接表结构）而数组不是。

1. List 列表的声明与遍历

val list = List("hello","changsha")

val res =list.+("aaa")

print(res)

效果图如下：  
  
2. List 列表的map、flatMap函数

实例：

val nameList = List(

"hello hunan",

"hello changsha",

"hello mingzhen"

)

val mapResult:List[Array[String]] = nameList.map{

x => x.split(" ")

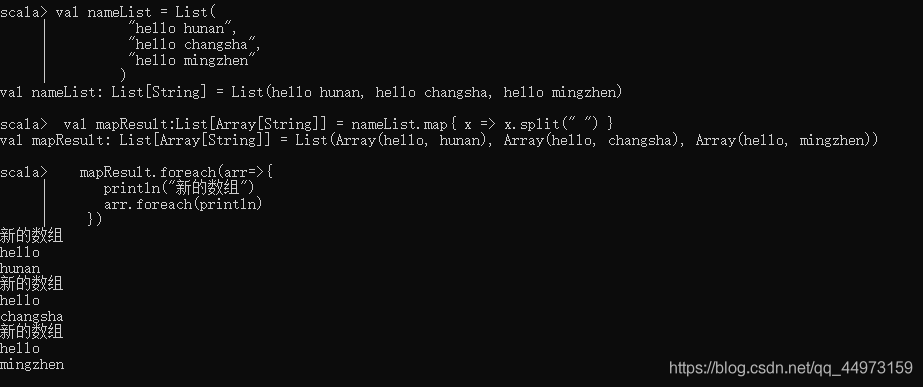
}

mapResult.foreach(arr=>{

println("新的数组")

arr.foreach(println)

})

  
使用flatMap

val nameList = List(

"hello hunan",

"hello changsha",

"hello mingzhen"

)

val flatmapResult:List[String] = nameList.flatMap{

x => x.split(" ")

}

flatmapResult.foreach(println)

效果图如下：  
  
3. List 列表的filter过滤函数

匹配上的会被留下，

val nameList = List(

"hello hunan",

"hello changsha",

"hello mingzhen"

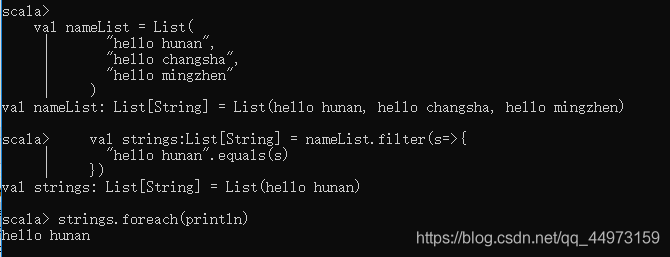
)

val strings:List[String] = nameList.filter(s=>{

"hello hunan".equals(s)

})

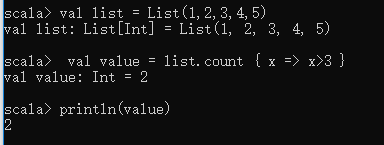
strings.foreach(println)

效果图如下：  
  
4. List 列表的 count 计数函数

val list = List(1,2,3,4,5)

scala> val value = list.count { x => x>3 }

println(value)

效果图如下：  
  
5. List 列表的可变

同理首先要导入相对应的包：

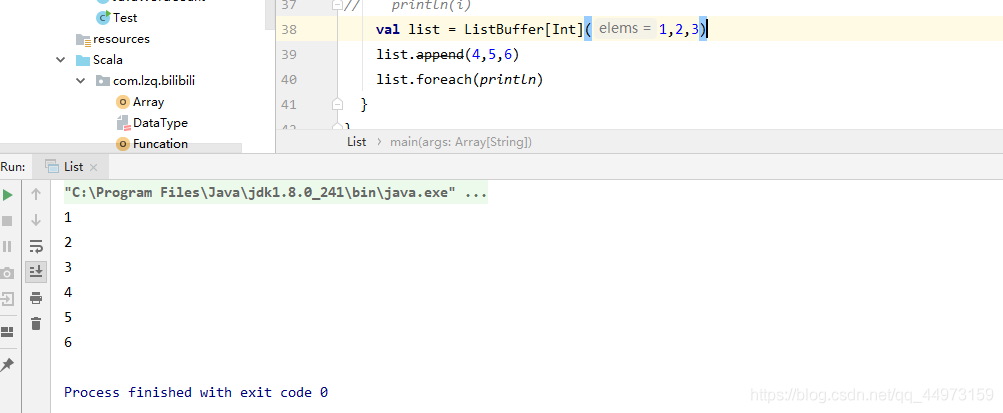
import scala.collection.mutable.ListBuffer

对列表进行添加元素：

val list = ListBuffer[Int](1,2,3)

list.append(4,5,6)

list.foreach(println)

  
6. List 列表其余的方法及函数

def +(elem: A): List[A]

前置一个元素列表

def ::(x: A): List[A]

在这个列表的开头添加的元素。

def :::(prefix: List[A]): List[A]

增加了一个给定列表中该列表前面的元素。

def ::(x: A): List[A]

增加了一个元素x在列表的开头

def addString(b: StringBuilder): StringBuilder

追加列表的一个字符串生成器的所有元素。

def addString(b: StringBuilder, sep: String): StringBuilder

追加列表的使用分隔字符串一个字符串生成器的所有元素。

def apply(n: Int): A

选择通过其在列表中索引的元素

def contains(elem: Any): Boolean

测试该列表中是否包含一个给定值作为元素。

def copyToArray(xs: Array[A], start: Int, len: Int): Unit

列表的副本元件阵列。填充给定的数组xs与此列表中最多len个元素，在位置开始。

def distinct: List[A]

建立从列表中没有任何重复的元素的新列表。

def drop(n: Int): List[A]

返回除了第n个的所有元素。

def dropRight(n: Int): List[A]

返回除了最后的n个的元素。

def dropWhile(p: (A) => Boolean): List[A]

丢弃满足谓词的元素最长前缀。

def endsWith[B](that: Seq[B]): Boolean

测试列表是否使用给定序列结束。

def equals(that: Any): Boolean

equals方法的任意序列。比较该序列到某些其他对象。

def exists(p: (A) => Boolean): Boolean

测试谓词是否持有一些列表的元素。

def filter(p: (A) => Boolean): List[A]

返回列表满足谓词的所有元素。

def forall(p: (A) => Boolean): Boolean

测试谓词是否持有该列表中的所有元素。

def foreach(f: (A) => Unit): Unit

应用一个函数f以列表的所有元素。

def head: A

选择列表的第一个元素

def indexOf(elem: A, from: Int): Int

经过或在某些起始索引查找列表中的一些值第一次出现的索引。

def init: List[A]

返回除了最后的所有元素

def intersect(that: Seq[A]): List[A]

计算列表和另一序列之间的多重集交集。

def isEmpty: Boolean

测试列表是否为空

def iterator: Iterator[A]

创建一个新的迭代器中包含的可迭代对象中的所有元素

def last: A

返回最后一个元素

def lastIndexOf(elem: A, end: Int): Int

之前或在一个给定的最终指数查找的列表中的一些值最后一次出现的索引

def length: Int

返回列表的长度

def map[B](f: (A) => B): List[B]

通过应用函数以g这个列表中的所有元素构建一个新的集合

def max: A

查找最大的元素

def min: A

查找最小元素

def mkString: String

显示列表的字符串中的所有元素

def mkString(sep: String): String

显示的列表中的字符串中使用分隔串的所有元素

def reverse: List[A]

返回新列表，在相反的顺序元素

def sorted[B >: A]: List[A]

根据排序对列表进行排序

def startsWith[B](that: Seq[B], offset: Int): Boolean

测试该列表中是否包含给定的索引处的给定的序列

def sum: A

概括这个集合的元素

def tail: List[A]

返回除了第一的所有元素

def take(n: Int): List[A]

返回前n个元素

def takeRight(n: Int): List[A]

返回最后n个元素

def toArray: Array[A]

列表以一个数组变换

def toBuffer[B >: A]: Buffer[B]

列表以一个可变缓冲器转换

def toMap[T, U]: Map[T, U]

此列表的映射转换

def toSeq: Seq[A]

列表的序列转换

def toSet[B >: A]: Set[B]

列表到集合变换

def toString(): String

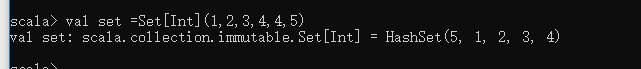
列表转换为字符串

# Scala 的 Set集合（不可重复）详解

Set定义

Set 当中元素是无序的，而且会去重，如下代码所示，定义了一个Set

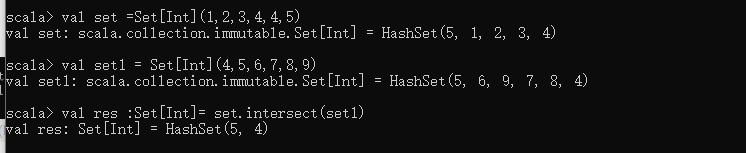
val set =Set[Int](1,2,3,4,4,5)

代码运行如下  
  
intersect 求交集

val set =Set[Int](1,2,3,4,4,5)

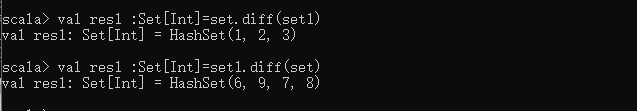
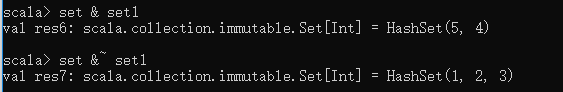
val set1 = Set[Int](4,5,6,7,8,9)

val res :Set[Int]= set.intersect(set1)

效果图如下：  
  
diff 求差集

val res1 :Set[Int]=set.diff(set1)

val res1 :Set[Int]=set1.diff(set)

效果图如下  
  
操作符操作  
  
filter过滤器

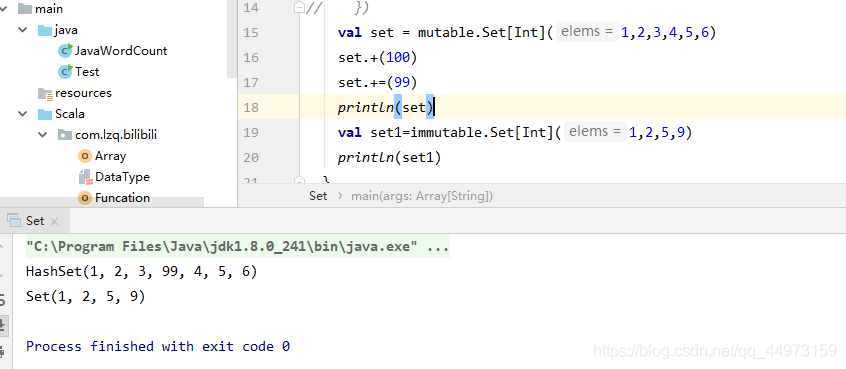
可以看到返回值的类型还是Set。  
在这里插入图片描述

可变长的Set

导入包

import scala.collection.mutable

import scala.collection.imutable

在这里插入图片描述  
而在使用Set的时候，要使用可变和不可变的时候，只需要在Set前面加上mutable或者imutable  
  
Set的其余用法

| **序号** | **方法** | **描述** |
| --- | --- | --- |
| 1 | def +(elem: A): Set[A] | 为集合添加新元素，x并创建一个新的集合，除非元素已存在 |
| 2 | def -(elem: A): Set[A] | 移除集合中的元素，并创建一个新的集合 |
| 3 | def contains(elem: A): Boolean | 如果元素在集合中存在，返回 true，否则返回 false。 |
| 4 | def &(that: Set[A]): Set[A] | 返回两个集合的交集 |
| 5 | def &~(that: Set[A]): Set[A] | 返回两个集合的差集 |
| 6 | def +(elem1: A, elem2: A, elems: A\*): Set[A] | 通过添加传入指定集合的元素创建一个新的不可变集合 |
| 7 | def ++(elems: A): Set[A] | 合并两个集合 |
| 8 | def -(elem1: A, elem2: A, elems: A\*): Set[A] | 通过移除传入指定集合的元素创建一个新的不可变集合 |
| 9 | def addString(b: StringBuilder): StringBuilder | 将不可变集合的所有元素添加到字符串缓冲区 |
| 10 | def addString(b: StringBuilder, sep: String): StringBuilder | 将不可变集合的所有元素添加到字符串缓冲区，并使用指定的分隔符 |
| 11 | def apply(elem: A) | 检测集合中是否包含指定元素 |
| 12 | def count(p: (A) => Boolean): Int | 计算满足指定条件的集合元素个数 |
| 13 | def copyToArray(xs: Array[A], start: Int, len: Int): Unit | 复制不可变集合元素到数组 |
| 14 | def diff(that: Set[A]): Set[A] | 比较两个集合的差集 |
| 15 | def drop(n: Int): Set[A]] | 返回丢弃前n个元素新集合 |
| 16 | def dropRight(n: Int): Set[A] | 返回丢弃最后n个元素新集合 |
| 17 | def dropWhile(p: (A) => Boolean): Set[A] | 从左向右丢弃元素，直到条件p不成立 |
| 18 | def equals(that: Any): Boolean | equals 方法可用于任意序列。用于比较系列是否相等。 |
| 19 | def exists(p: (A) => Boolean): Boolean | 判断不可变集合中指定条件的元素是否存在。 |
| 20 | def filter(p: (A) => Boolean): Set[A] | 输出符合指定条件的所有不可变集合元素。 |
| 21 | def find(p: (A) => Boolean): Option[A] | 查找不可变集合中满足指定条件的第一个元素 |
| 22 | def forall(p: (A) => Boolean): Boolean | 查找不可变集合中满足指定条件的所有元素 |
| 23 | def foreach(f: (A) => Unit): Unit | 将函数应用到不可变集合的所有元素 |
| 24 | def head: A | 获取不可变集合的第一个元素 |
| 25 | def init: Set[A] | 返回所有元素，除了最后一个 |
| 26 | def intersect(that: Set[A]): Set[A] | 计算两个集合的交集 |
| 27 | def isEmpty: Boolean | 判断集合是否为空 |
| 28 | def iterator: Iterator[A] | 创建一个新的迭代器来迭代元素 |
| 29 | def last: A | 返回最后一个元素 |
| 30 | def map[B](f: (A) => B): immutable.Set[B] | 通过给定的方法将所有元素重新计算 |
| 31 | def max: A | 查找最大元素 |
| 32 | def min: A | 查找最小元素 |
| 33 | def mkString: String | 集合所有元素作为字符串显示 |
| 34 | def mkString(sep: String): String | 使用分隔符将集合所有元素作为字符串显示 |
| 35 | def product: A | 返回不可变集合中数字元素的积。 |
| 36 | def size: Int | 返回不可变集合元素的数量 |
| 37 | def splitAt(n: Int): (Set[A], Set[A]) | 把不可变集合拆分为两个容器，第一个由前 n 个元素组成，第二个由剩下的元素组成 |
| 38 | def subsetOf(that: Set[A]): Boolean | 如果集合A中含有子集B返回 true，否则返回false |
| 39 | def sum: A | 返回不可变集合中所有数字元素之和 |
| 40 | def tail: Set[A] | 返回一个不可变集合中除了第一元素之外的其他元素 |
| 41 | def take(n: Int): Set[A] | 返回前 n 个元素 |
| 42 | def takeRight(n: Int):Set[A] | 返回后 n 个元素 |
| 43 | def toArray: Array[A] | 将集合转换为数组 |
| 44 | def toBuffer[B >: A]: Buffer[B] | 返回缓冲区，包含了不可变集合的所有元素 |
| 45 | def toList: List[A] | 返回 List，包含了不可变集合的所有元素 |
| 46 | def toMap[T, U]: Map[T, U] | 返回 Map，包含了不可变集合的所有元素 |
| 47 | def toSeq: Seq[A] | 返回 Seq，包含了不可变集合的所有元素 |
| 48 | def toString(): String | 返回一个字符串，以对象来表示 |

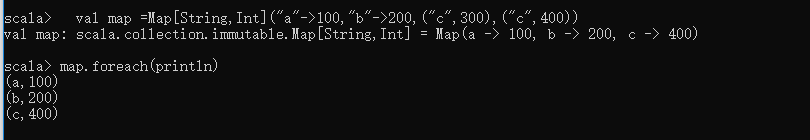
# Scala当中的Map 映射

key -> value 的语法形式实际上是用库中的隐式转换实现的，实际调用了 Map.apply 方法。Map.apply 方法的参数为一个两元素的元组（键值对）。

map 的参数是一个函数，以上示例展示了定义这个函数的两种方法。每个键值对（元组）都将被传入到该函数中。我们可以将其参数定义为一个两元素的元组，

map定义：其中当出现同一个key值的时候，后面的值会将前面的值进行覆盖。使用foreach进行遍历

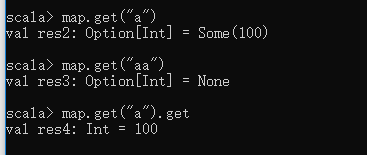
val map =Map[String,Int]("a"->100,"b"->200,("c",300),("c",400))

  
根据key返回value值，使用的是一个get,这个时候如果存在对应的值就会返回一个Some类型的值，反之则是一个None，这俩个类型都是Option的子类。最后再.get就会返回对应的基本类型。

map.get("a")

map.get("aa")

map.get("a").get

  
当对一个不存在的key获取他的值的时候，很显然是会报错的，这个时候可以使用 ,getOrElse()方法，表示当不存在的时候，返回这个方法里面的东西。

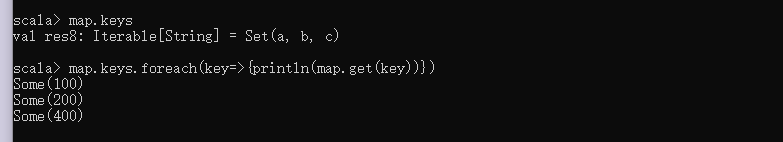
map.get("aa").getOrElse("不存在")

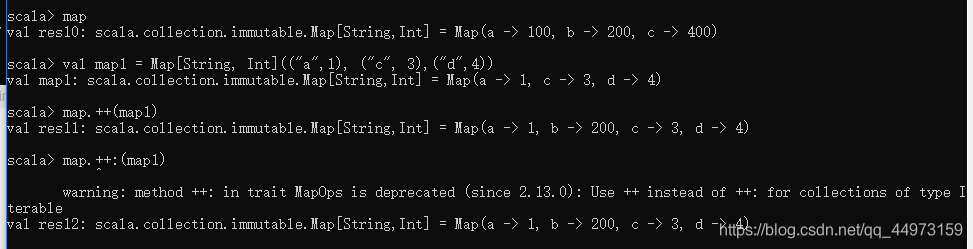
在这里插入图片描述  
获取所有的key，使用.keys方法，返回值的类型是一个 Iterable [String]  
根据所获取的key值打印出所对应的value值

map.keys.foreach(key => {

println(map.get(key))

})

  
俩个map映射合并  
其中.++是表示将map1往map里面合并，即map1覆盖map，而原本的.++:是表示map往map1里面添加，map覆盖map1，但是在这里弹出警告：trait MapOps中的method++：已弃用（从2.13.0开始）：对于Iterable类型的集合，请使用++而不是++：方法。

注： 如果一个包含 + 的表达式返回结果类型 String，而这并不是你所期望的结果。这可能是因为编译器认为这是该表达式唯一可行的解析方式，就把 + 两边的子表达式转为字符串，再相加。  
  
可变长的Map，导入对应的Map包，使用put方法进行添加元素

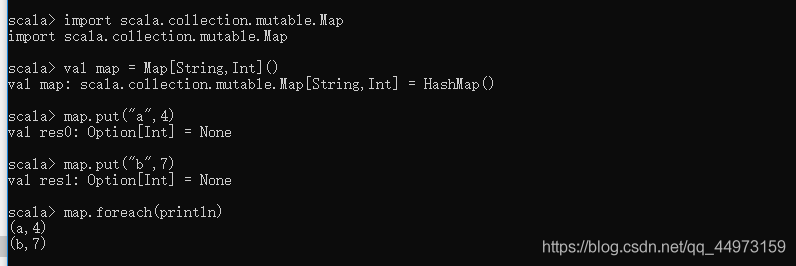
import scala.collection.mutable.Map

val map = Map[String,Int]()

map.put("a",4)

map.put("b",7)

map.foreach(println)

  
可变长度的map映射的filter过滤，当使用的是命令行的时候，没有换行输入的话，需要家伙少年宫分号进行隔开，在最后可以看到返回值的类型会是一个 scala.collection.mutable.Map[String,Int]

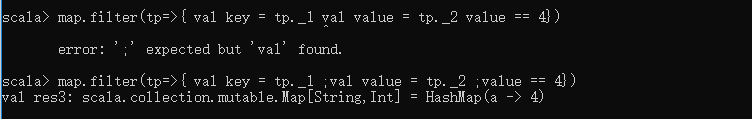
map.filter(tp => {

val key = tp.\_1

val value = tp.\_2

value == 4

})

  
其余的Map方法：如下表所示

| **序号** | **方法名** | **描述** |
| --- | --- | --- |
| 1 | def ++(xs: Map[(A, B)]): Map[A, B] | 返回一个新的 Map，新的 Map xs 组成 |
| 2 | def -(elem1: A, elem2: A, elems: A\*): Map[A, B] | 返回一个新的 Map, 移除 key 为 elem1, elem2 或其他 elems。 |
| 3 | def --(xs: GTO[A]): Map[A, B] | 返回一个新的 Map, 移除 xs 对象中对应的 key |
| 4 | def get(key: A): Option[B] | 返回指定 key 的值 |
| 5 | def iterator: Iterator[(A, B)] | 创建新的迭代器，并输出 key/value 对 |
| 6 | def addString(b: StringBuilder): StringBuilder | 将 Map 中的所有元素附加到StringBuilder，可加入分隔符 |
| 7 | def addString(b: StringBuilder, sep: String): StringBuilder | 将 Map 中的所有元素附加到StringBuilder，可加入分隔符 |
| 8 | def apply(key: A): B | 返回指定键的值，如果不存在返回 Map 的默认方法 |
| 10 | def clone(): Map[A, B] | 从一个 Map 复制到另一个 Map |
| 11 | def contains(key: A): Boolean | 如果 Map 中存在指定 key，返回 true，否则返回 false。 |
| 12 | def copyToArray(xs: Array[(A, B)]): Unit | 复制集合到数组 |
| 13 | def count(p: ((A, B)) => Boolean): Int | 计算满足指定条件的集合元素数量 |
| 14 | def default(key: A): B | 定义 Map 的默认值，在 key 不存在时返回。 |
| 15 | def drop(n: Int): Map[A, B] | 返回丢弃前n个元素新集合 |
| 16 | def dropRight(n: Int): Map[A, B] | 返回丢弃最后n个元素新集合 |
| 17 | def dropWhile(p: ((A, B)) => Boolean): Map[A, B] | 从左向右丢弃元素，直到条件p不成立 |
| 18 | def empty: Map[A, B] | 返回相同类型的空 Map |
| 19 | def equals(that: Any): Boolean | 如果两个 Map 相等(key/value 均相等)，返回true，否则返回false |
| 20 | def exists(p: ((A, B)) => Boolean): Boolean | 判断集合中指定条件的元素是否存在 |
| 21 | def filter(p: ((A, B))=> Boolean): Map[A, B] | 返回满足指定条件的所有集合 |
| 22 | def filterKeys(p: (A) => Boolean): Map[A, B] | 返回符合指定条件的的不可变 Map |
| 23 | def find(p: ((A, B)) => Boolean): Option[(A, B)] | 查找集合中满足指定条件的第一个元素 |
| 24 | def foreach(f: ((A, B)) => Unit): Unit | 将函数应用到集合的所有元素 |
| 25 | def init: Map[A, B] | 返回所有元素，除了最后一个 |
| 26 | def isEmpty: Boolean | 检测 Map 是否为空 |
| 27 | def keys: Iterable[A] | 返回所有的key/p> |
| 28 | def last: (A, B) | 返回最后一个元素 |
| 29 | def max: (A, B) | 查找最大元素 |
| 30 | def min: (A, B) | 查找最小元素 |
| 31 | def mkString: String | 集合所有元素作为字符串显示 |
| 32 | def product: (A, B) | 返回集合中数字元素的积。 |
| 33 | def remove(key: A): Option[B] | 移除指定 key |
| 34 | def retain(p: (A, B) => Boolean): Map.this.type | 如果符合满足条件的返回 true |
| 35 | def size: Int | 返回 Map 元素的个数 |
| 36 | def sum: (A, B) | 返回集合中所有数字元素之和 |
| 37 | def tail: Map[A, B] | 返回一个集合中除了第一元素之外的其他元素 |
| 38 | def take(n: Int): Map[A, B] | 返回前 n 个元素 |
| 39 | def takeRight(n: Int): Map[A, B] | 返回后 n 个元素 |
| 40 | def takeWhile(p: ((A, B)) => Boolean): Map[A, B] | 返回满足指定条件的元素 |
| 41 | def toArray: Array[(A, B)] | 集合转数组 |
| 42 | def toBuffer[B >: A]: Buffer[B] | 返回缓冲区，包含了 Map 的所有元素 |
| 43 | def toList: List[A] | 返回 List，包含了 Map 的所有元素 |
| 44 | def toSeq: Seq[A] | 返回 Seq，包含了 Map 的所有元素 |
| 45 | def toSet: Set[A] | 返回 Set，包含了 Map 的所有元素 |
| 46 | def toString(): String | 返回字符串对象 |

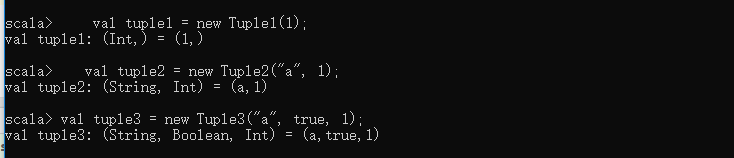
# Scala 的 Tuple 元组

元组的定义：与列表一样，与列表不同的是元组可以包含不同类型的元素。元组的值是通过将单个的值包含在圆括号中构成的。当定义的元组当中包含不同的数据类型，可看到返回值开始会包含其所对应的数据类型

val tuple1 = new Tuple1(1);

val tuple2 = new Tuple2("a", 1);

val tuple3 = new Tuple3("a", true, 1);

  
在元组当中最多可以写上22个元组的数据类型。虽然不会报错，但是不是元组类型了。  
使用.\_数字 进行获取元组当中所对应的值

tuple3.\_3

在这里插入图片描述  
对元组进行遍历，不存在for和foeach函数。需要使用到迭代器：

val res = tuple3.productIterator

使用到了迭代器就可以使用到while进行遍历：

while(res.hasNext){

println(res.next())

}

或者使用for或者foreach循环

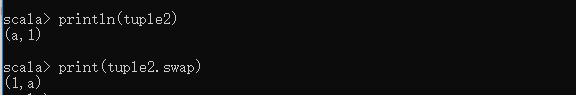
res.foreach(println)

在命令行当中使用foreach遍历没有返回结果，可在idea当中进行测试  
  
toString()方法：将元组当中的值放在一个括号里面进行返回：

println(tuple3.toString())

在这里插入图片描述  
在tuple2当中还存在一个swap函数，表示逆序显示

print(tuple2.swap)



# Scala 使用 Trait 接口

Scala Trait(特征) 相当于 Java 的接口，实际上它比接口还功能强大。与接口不同的是，它还可以定义属性和方法的实现。  
一般情况下Scala的类可以继承多个Trait，从结果来看就是实现了多重继承。Trait(特征) 定义的方式与类类似，但它使用的关键字是 trait。

案例实现  
1.定义trait，不能传递参数

trait read{

def read(name:String): Unit ={

println(name +" is reading ...")

}

}

trait listen{

def listen(name:String): Unit ={

println(name +" is listen ...")

}

}

2.使用一个class类继承这些trait。

class Person1() extends read with listen{

//第一个继承使用extends 其余的都使用 with

}

3.调用类当中的接口：

object Trait1 {

def main(args: Array[String]): Unit = {

val p = new Person1()

p.read("张三")

p.listen("李四")

}

}

运行结果如下：  


# Scala 的 Match Case 模式匹配、偏函数与样式类

1.模式匹配

Scala 提供了强大的模式匹配机制，应用也非常广泛。

一个模式匹配包含了一系列备选项，每个都开始于关键字 case。

每个备选项都包含了一个模式及一到多个表达式。箭头符号 => 隔开了模式和表达式。

案例实现  
1.先在main方法中定义一个元组，并且获取到元组当中的每一个数据

val t = (1, 5, "a", true)

val res: Iterator[Any] = t.productIterator

2.定义一个匹配模式的方法，对不同的值进行匹配：

def MatchTest(o: Any): Unit = {

o match {

case 1 => println("value is 1")

case i: Int => println("type is Int , value is :" + i)

case s: String => println("type is String , value is :" + s)

case \_ => { //默认匹配，什么都匹配不上才匹配，往往放在最后

println("no match")

}

}

3.调用这个匹配模式，也就是在遍历数组的时候进行遍历：

res.foreach(MatchTest(\_))

或者

res.foreach(s => {

MatchTest(s)

})

注意点：

1. 模式匹配不仅可以匹配值还可以匹配类型
2. 从上到下顺序匹配，如果匹配到则不再往下匹配
3. 都匹配不上时，会匹配到case \_ ,相当于default
4. match 的最外面的”{ }”可以去掉看成一个语句

2.偏函数

如果一个方法中没有 match 只有case，这个函数可以定义成 PartialFunction 偏函数。偏函数定义时，不能使用括号传参，默认定义 PartialFunction 中传入一个值，匹配上了对应的case,返回一个值。PartialFunction 当中给定的类型是匹配的类型和返回值的类型。  
案例示例

def MyTest: PartialFunction[String, Int] = {

case "abc" => 3

case "a" => 1

case "b" => 2

case \_ => 0

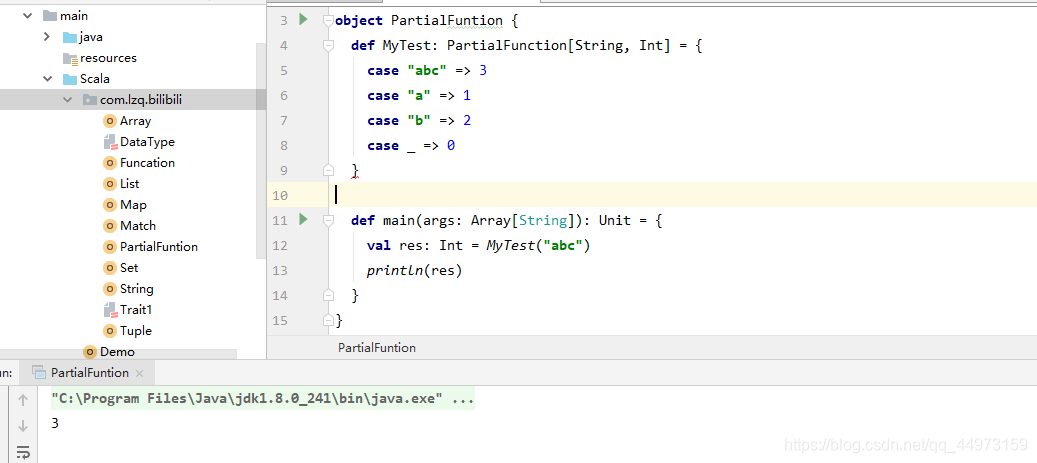
}

def main(args: Array[String]): Unit = {

val res: Int = MyTest("abc")

println(res)

}

对abc进行匹配，很显然返回值会是一个3  
  
3. 样式类

使用了case关键字的类定义就是样例类(case classes)，样例类是种特殊的类。实现了类构造参数的getter方法（构造参数默认被声明为val），当构造参数是声明为var类型的，它将帮你实现setter和getter方法。

注意点

1. 样例类默认帮你实现了toString,equals，copy和hashCode等方法。
2. 样例类可以new, 也可以不用new

代码示例

case class Person2(name:String,age:Int){

}

object Case\_classes {

def main(args: Array[String]): Unit = {

val p1 = new Person2(name = "zhangsan",age = 18)

val p2 = new Person2(name = "zhangsan",age = 18)

println(p1.equals(p2))

println(p1)

}

}



# Scala的隐式转换详解

隐式转换是在Scala编译器进行类型匹配时，如果找不到合适的类型，那么隐式转换会让编译器在作用范围内自动推导出来合适的类型。

1.隐式值与隐式参数  
隐式值是指在定义参数时前面加上implicit。隐式参数是指在定义方法时，方法中的部分参数是由implicit修饰【必须使用柯里化的方式，将隐式参数写在后面的括号中】。隐式转换作用就是：当调用方法时，不必手动传入方法中的隐式参数，Scala会自动在作用域范围内寻找隐式值自动传入。  
隐式值和隐式参数注意：

1. 同类型的参数的隐式值只能在作用域内出现一次，同一个作用域内不能定义多个类型一样的隐式值。
2. implicit 关键字必须放在隐式参数定义的开头
3. 一个方法只有一个参数是隐式转换参数时，那么可以直接定义implicit关键字修饰的参数，调用时直接创建类型不传入参数即可。
4. 一个方法如果有多个参数，要实现部分参数的隐式转换,必须使用柯里化这种方式,隐式关键字出现在后面，只能出现一次

def SayName(implicit name: String): Unit = {

println(name + " is a student ....")

}

def information(age: Int)(implicit name: String): Unit = {

println("name is " + name + ", age is " + age)

}

def main(args: Array[String]): Unit = {

implicit val name: String = "zhangsan"

SayName

information(19)

}

输出的结果：会自动寻找对应的函数，并且进行隐式转换。

zhangsan is a student ....

name is zhangsan, age is 19

2.隐式转换函数  
隐式转换函数是使用关键字implicit修饰的方法。当Scala运行时，假设如果A类型变量调用了method()这个方法，发现A类型的变量没有method()方法，而B类型有此method()方法，会在作用域中寻找有没有隐式转换函数将A类型转换成B类型，如果有隐式转换函数，那么A类型就可以调用method()这个方法。  
隐式转换函数注意：隐式转换函数只与函数的参数类型和返回类型有关，与函数名称无关，所以作用域内不能有相同的参数类型和返回类型的不同名称隐式转换函数。

class Anima(name: String) {

def canfly(): Unit = {

println(name + " can fly")

}

}

class Person3(pname: String) {

val name = pname;

}

//在object当中定义。

implicit def AnimaToPerson3(p: Person3): Anima = {

new Anima(p.name);

}

//在main函数当中定义一个类的变量进行访问方法。

val p = new Person3("person\_name");

p.canfly()

运行结果是：

person\_name can fly

3.隐式类

使用implicit关键字修饰的类就是隐式类。若一个变量A没有某些方法或者某些变量时，而这个变量A可以调用某些方法或者某些变量时，可以定义一个隐式类，隐式类中定义这些方法或者变量，隐式类中传入A即可。  
隐式类注意：

1. 隐式类必须定义在类，包对象，伴生对象中。
2. 隐式类的构造必须只有一个参数，同一个类，包对象，伴生对象中不能出现同类型构造的隐式类。

//在object当中定义隐式类。使用到前面的person3的参数

implicit class Person4(p: Person3) {

def ShowName(): Unit = {

println(p.name + " is ShowNaame")

}

}

//在mian方法中调用

val p3 = new Person3("preson3\_name")

p3.ShowName()

运行结果是：

preson3\_name is ShowNaame

# Scala 的通信模型 Actor

Actor Model是用来编写并行计算或分布式系统的高层次抽象（类似java中的Thread）让程序员不必为多线程模式下共享锁而烦恼,被用在Erlang 语言上, 高可用性99.9999999 % 一年只有31ms 宕机Actors将状态和行为封装在一个轻量的进程/线程中，但是不和其他Actors分享状态，每个Actors有自己的世界观，当需要和其他Actors交互时，通过发送事件和消息，发送是异步的，非堵塞的(fire-andforget)，发送消息后不必等另外Actors回复，也不必暂停，每个Actors有自己的消息队列，进来的消息按先来后到排列，这就有很好的并发策略和可伸缩性，可以建立性能很好的事件驱动系统。

Actor的特征：

* ActorModel是消息传递模型,基本特征就是消息传递
* 消息发送是异步的，非阻塞的
* 消息一旦发送成功，不能修改
* Actor之间传递时，自己决定决定去检查消息，而不是一直等待，是异步非阻塞的

在使用Actor通信模型之前，需要添加对应的jar包：scala-actors.jar [下载地址，单击前往](http://www.java2s.com/Code/Jar/s/Downloadscalaactorsjar.htm)添加jar包并且添加到环境当中。

随后创建代码：导入Actor包，使用一个class继承这个类，继承之后需要重写act方法，添加匹配。在main函数当中实例化一个对象，调用start方法进行启动，使用！进行发送消息。

package Actors

import scala.actors.Actor

class MyActor extends Actor{

override def act(): Unit = {

receive{

case s:String =>{println("type is String , s is " + s)}

case \_=> {println("no match ...")}

}

}

}

object Actor1 {

def main(args: Array[String]): Unit = {

val actor = new MyActor()

actor.start()

//actor ! "hello changsha"

actor ! 100

}

}

单方面发送消息，在此，我们可以定义俩个class类，使其互发消息。  
是用来一个样式类对发送消息的动作进行封装，封装发送给的那个对象和发送的消息。在里面的receive只会调用一次，所以在原先的基础上加上一个while循环始终调用，这个时候我们的case匹配的对象就不再是一个字符串类的值了，而是一个样式类的msg对象。

package Actors

import scala.actors.Actor

case class Msg(actor:Actor,msg: String){

}

//Actor之间的通信

class MyActor1 extends Actor{

override def act(): Unit = {

while(true){

receive{

case msg:Msg =>{println("type is String , s is " + msg.msg)

msg.actor ! " this is yueyue niao"

}

case \_=> {println("no match ...")}

}

}

}

}

class MyActor2(actor:Actor) extends Actor{

actor ! Msg(this,"this is huang yueyue ...")

override def act(): Unit = {

while(true){

receive{

case s:String =>{println("type is String , s is " + s)}

case \_=> {println("no match ...")}

}

}

}

}

object Actor2 {

def main(args: Array[String]): Unit = {

val actor1 = new MyActor1()

val actor2 = new MyActor2(actor1)

actor1.start()

actor2.start()

}

}

# Scala和Java分别实现的WordCount

在使用之前，对Maven进行配置：参考：[Java项目管理之Maven下载及安装](https://blog.csdn.net/qq_44973159/article/details/105777093) 和项目环境配置 [idea创建项目及配置详解和测试代码](https://blog.csdn.net/qq_44973159/article/details/105792078)  
之后建立一个data文件夹里面写一个word.txt文件，用于统计。

1.使用Scala实现单词统计

package WordCount

import org.apache.spark.rdd.RDD

import org.apache.spark.{SparkConf, SparkContext}

object SparkWC {

def main(args: Array[String]): Unit = {

val conf = new SparkConf() //设置Spark运行的名称、资源

//设置名称、

conf.setAppName("wordcount") //设置运行模式，使用local本地模式

conf.setMaster("local")

//创建Spark的上下文,通往Spark集群的唯一通道

val sc = new SparkContext(conf)

//返回类型是一个RDD类型

val lines: RDD[String] = sc.textFile("./data/word")

//按空格进行切开

val words: RDD[String] = lines.flatMap(line => {

line.split(" ")

}) //计数

val PairWord: RDD[(String, Int)] = words.map(word => {

new Tuple2(word, 1)

})

//先分组，再对value值进行计算

val res: RDD[(String, Int)] = PairWord.reduceByKey((v1: Int, v2: Int) => {

v1 + v2

})

//打印结果

res.foreach(one => {

println(one)

})

}

}

使用scala代码的时候，由于Scala的特性，代码还可以继续简化：

package WordCount

import org.apache.spark.rdd.RDD

import org.apache.spark.{SparkConf, SparkContext}

object SparkWC1 {

def main(args: Array[String]): Unit = {

val conf = new SparkConf().setAppName("wordcount").setMaster("local")

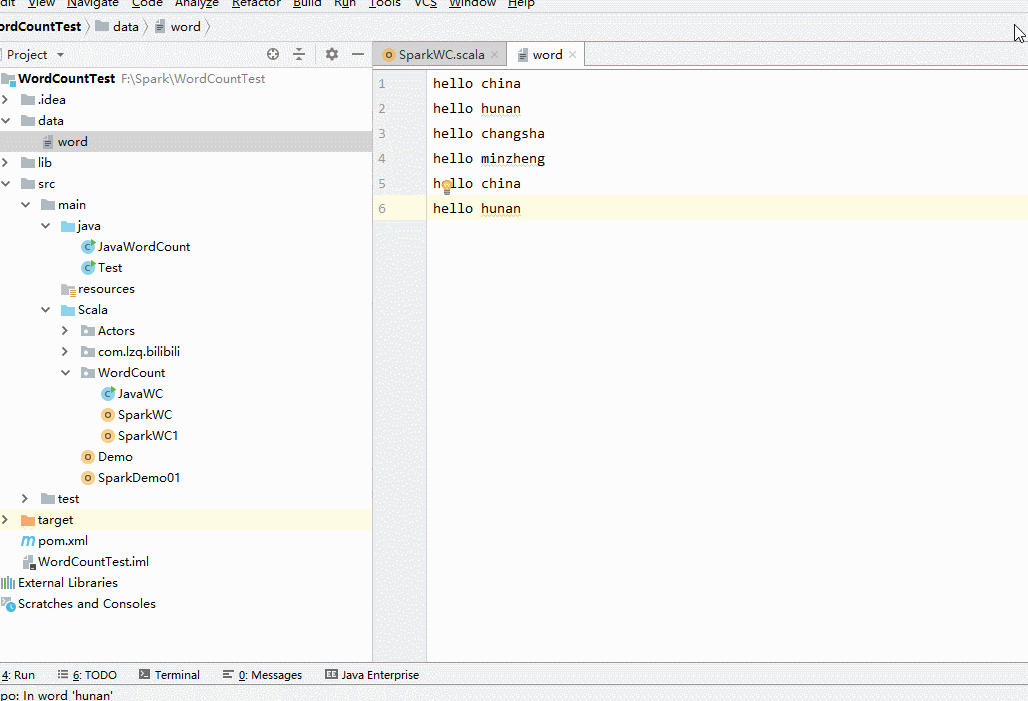
val sc = new SparkContext(conf)

sc.textFile("./data/word").flatMap(\_.split(" ")).map((\_, 1)).reduceByKey(\_ + \_).foreach(println)

sc.stop()

}

}

运行Scala代码，在控制台查看：  
  
2.使用Java实现单词统计

package WordCount;

import org.apache.spark.SparkConf;

import org.apache.spark.api.java.JavaPairRDD;

import org.apache.spark.api.java.JavaRDD;

import org.apache.spark.api.java.JavaSparkContext;

import org.apache.spark.api.java.function.FlatMapFunction;

import org.apache.spark.api.java.function.Function2;

import org.apache.spark.api.java.function.PairFunction;

import org.apache.spark.api.java.function.VoidFunction;

import org.apache.spark.sql.catalyst.plans.logical.Except;

import scala.Tuple2;

import java.util.Arrays;

import java.util.Iterator;

public class JavaWC {

public static void main(String[] args) {

SparkConf sparkConf = new SparkConf();

sparkConf.setAppName("javaWC");

sparkConf.setMaster("local");

JavaSparkContext javaSparkContext = new JavaSparkContext(sparkConf);

JavaRDD<String> lines = javaSparkContext.textFile("./data/word");

JavaRDD<String> words = lines.flatMap(new FlatMapFunction<String, String>() {

@Override

public Iterator<String> call(String lines) throws Exception {

return Arrays.asList(lines.split(" ")).iterator();

}

});

JavaPairRDD<String, Integer> PairWord = words.mapToPair(new PairFunction<String, String, Integer>() {

@Override

public Tuple2<String, Integer> call(String s) throws Exception {

return new Tuple2<String, Integer>(s, 1);

}

});

JavaPairRDD<String,Integer> res=PairWord.reduceByKey(new Function2<Integer, Integer, Integer>() {

@Override

public Integer call(Integer v1, Integer v2) throws Exception {

return v1 + v2;

}

});

res.foreach(new VoidFunction<Tuple2<String, Integer>>() {

@Override

public void call(Tuple2<String, Integer> stringIntegerTuple2) throws Exception {

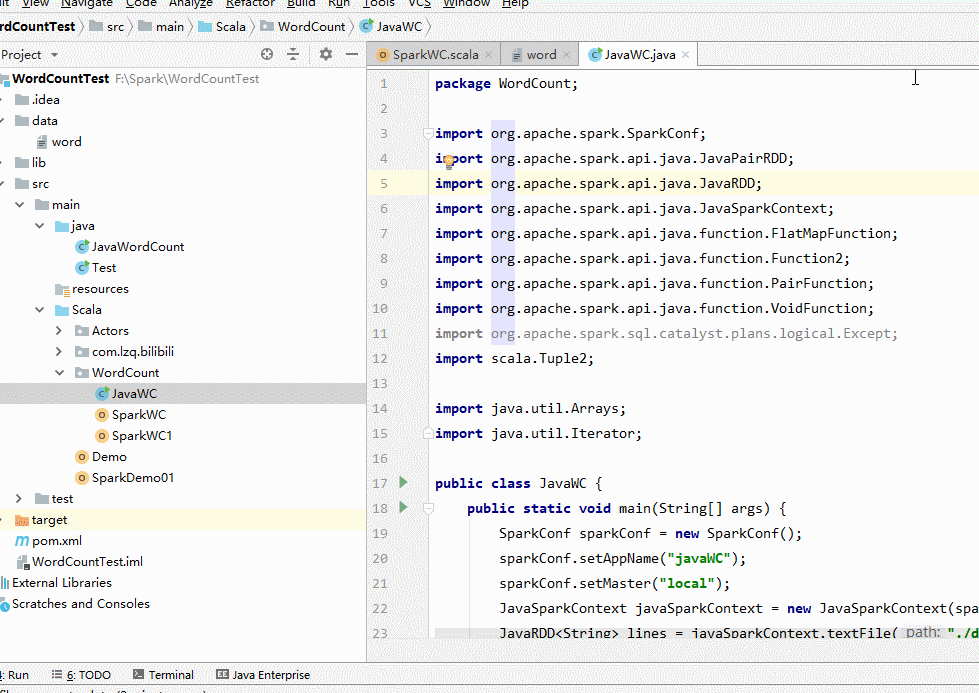
System.out.println(stringIntegerTuple2);

}

});

}

}

运行查看：效果是一样的：  
  
但是在这里很显然使用java代码实现代码过于冗余。不像Scala那样简洁。