

Scala集合&高阶函数

一 内容回顾

1.1 教学重点

1. Scala变量与类型
2. Scala类型转换
3. Scala条件与循环控制
4. Scala方法与函数定义

1.2 教学难点

1. 使用val与var定义变量的区别
2. Scala类型层次结构图
3. 方法与函数的区别

二 教学目标

- 1 1. 熟练使用定长Array和边长Array
- 2 2. 熟练可变Map和不可变Map
- 3 3. 熟练Tuple元组
- 4 4. 熟练使用List列表和Set集合
- 5 5. 熟练使用集合中的重要方法
- 6 6. 了解并行化集合
- 7 7. 熟练写出scala版本的WordCount
- 8 8. 了解Java和Scala集合的互相转换
- 9 9. 熟练使用匿名函数
- 10 10. 熟练使用函数作为方法参数和返回值
- 11 11. 了解闭包
- 12 12. 了解柯里化

三 教学导读

暂无

四 教学内容

4.1 集合（scala.collection）

集合是一种用来存储各种对象和数据的容器。Scala 集合分为可变的和不可变的集合。

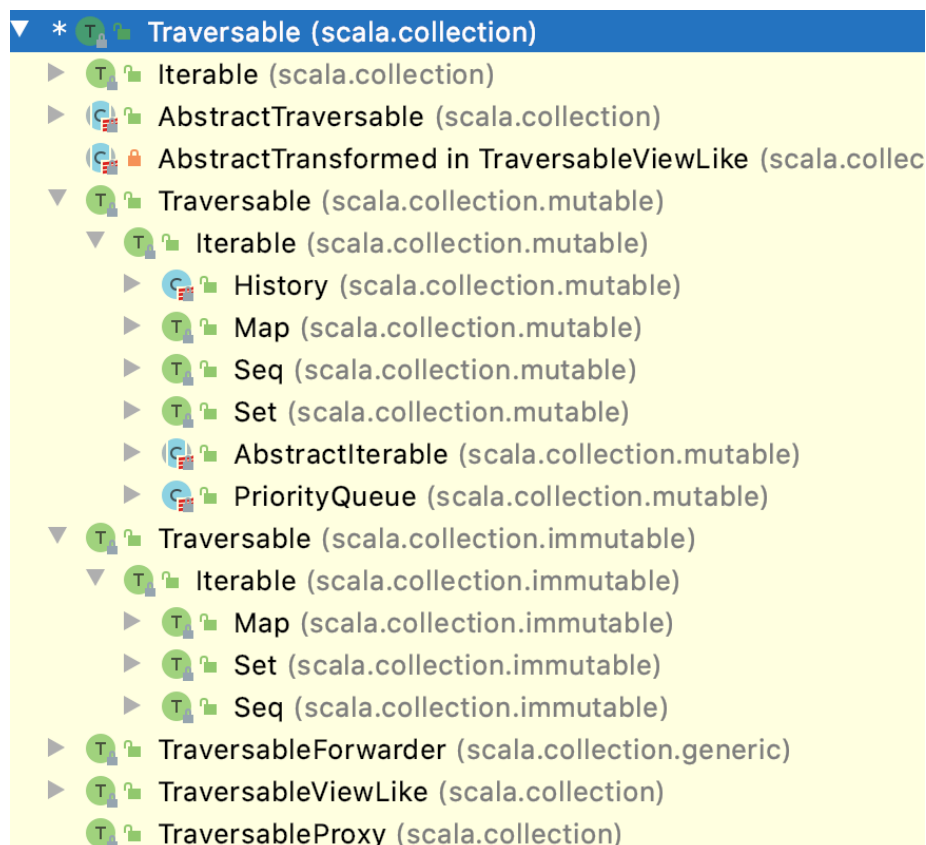
- 1 1. 不可变集合可以安全的并发访问。
- 2 2. 可变集合可以在适当的地方被更新或扩展。这意味着你可以修改，添加，移除一个集合的元素。

不可变集合，相比之下，永远不会改变。不过，你仍然可以模拟添加，移除或更新操作。但是这些操作将在每一种情况下都返回一个新的集合，同时使原来的集合不发生改变。

scala集合两个主要的包：

```
1 # 不可变集合
2   scala.collection.immutable (Scala默认采用不可变集合)
3 # 可变集合
4   scala.collection.mutable
5
```

Scala的集合有三大类：**序列Seq、集Set、映射Map**，所有的集合都扩展自Iterable特质（暂理解为接口），意味着集合的基本特点是支持迭代遍历的。



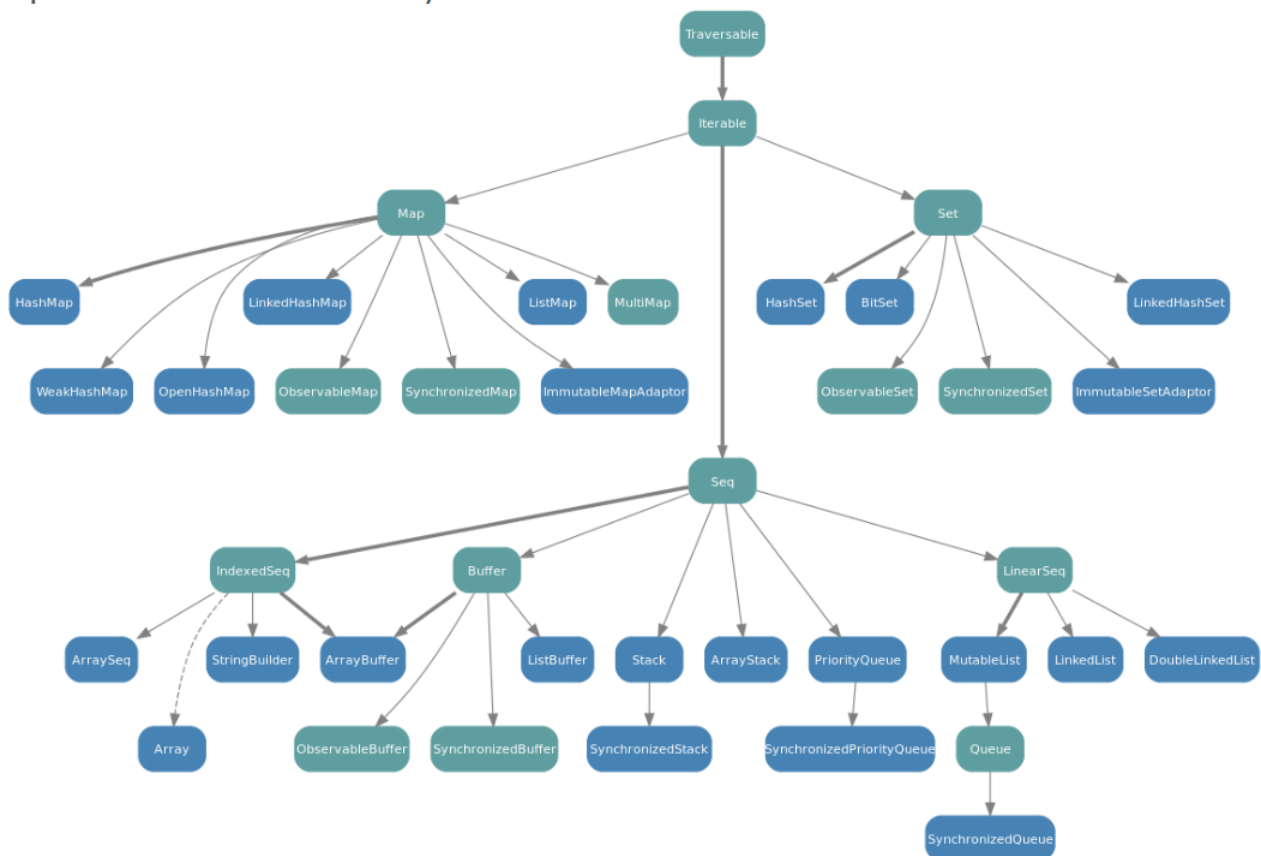
分类	描述
Seq	序列。元素以线性方式存储，集合中可以存放重复对象，。参考 API文档
Set	集（数据集，区别于集合）。集中的对象不按特定的方式排序，并且没有重复对象。 参考 API文档
Map	一种把键对象和值对象映射的集合，它的每一个元素都包含一对键对象和值对象。 参考 API文档

对于可变与不可变集合，Seq、Set、Map又有不同的实现方式，下二图详细描述了其继承关系。[官方文档](#)

Graph for immutable hierarchy:



Graph for mutable hierarchy:



4.1.1 数组

数组元素内容要求类型一致。按照是否可扩容分为两种：

- 1 **# Array**
- 2 - 定长数组，数组不可扩容 `scala.Array`
- 3 **# ArrayBuffer**
- 4 - 变长数组，数组可扩容 `scala.collection.mutable.ArrayBuffer`

4.1.1.1 定长数组Array

Java中创建定长数组

```
1  int[] data = new int[3]; /*定义长度为3的数组*/
2  data[0] = 1; // 第一个元素
3  data[1] = 2; // 第二个元素
4  data[2] = 3; // 第三个元素
5
6  //也可以创建时初始化数组元素
7  int[] data=new int[] {1,2,3};
```

同样，Scala中也有两种创建一个定长数组的方式：

- 使用new关键字创建一个定长数组， `var arr=new Array[Int](3)`
- 直接使用Array创建并初始化一个数组，注意不再使用new关键字。 `var arr=Array(1,2,3)`

```
scala> var arr= new Array[Int](10)
arr: Array[Int] = Array(0, 0, 0, 0, 0, 0, 0, 0, 0, 0)

scala> var arr1 = Array[Int](10,20,30)
arr1: Array[Int] = Array(10, 20, 30)

scala> var arr2 = Array(10,20,30)
arr2: Array[Int] = Array(10, 20, 30)

scala> var arr3 = Array()
arr3: Array[Nothing] = Array()

scala> arr3(0)=1
<console>:13: error: type mismatch;
 found   : Int(1)
 required: Nothing
    arr3(0)=1
           ^
```

课上练习：使用Array定义一个长度不变的数组

```
1  object ArrayDemo {
2      def main(args: Array[String]){
3          //初始化一个长度为8的定长数组
4          val arr1 = new Array[Int](8)
```

```
5 //会有初始化零值: Array[Int] = Array(0,0,0,0,0,0,...)
6 //直接打印定长数组, 内容为数组的hashCode值
7 println(arr1)
8 //将数组转换成数组缓冲, 就可以看到原数组中的内容了
9 //toBuffer会将数组转换成长数组缓冲
10 println(arr1.toBuffer)
11
12 //注意: 如果不使用new获取数组, 相当于调用了数组的apply方法, 直接为数组赋值
13 //通过一组初始化值定义定长数组
14 val arr2 = Array[Int](10,20,30)
15 //输出数组元素值
16 println(arr2.toBuffer)
17
18 //定义一个长度为3的定长数组
19 val arr3 = Array("hadoop", "storm", "spark")
20 //使用()来访问元素
21 println(arr3(2))
22
23 //包含10个整数的数组, 初始化值为0
24 val nums = new Array[Int](10)
25 //遍历数组
26 for(i <- 0 until nums.length)
27     print(s"$i:${nums(i)} ") //s字符串插值
28 println()
29
30 //包含10个字符串的数组, 初始化值为null
31 val strs0 = new Array[String](10)
32 for(i <- 0 until strs0.length)
33     print(s"$i:${strs0(i)} ")
34     println()
35
36 //赋初值的字符串数组
```



```

37     val strs1 = Array("hello" ,"world")
38     for(i <- 0 until strs1.length)
39         print(s"$i:${strs1(i)} ")
40         println()
41
42     //访问并修改元素值
43     strs1(0) = "byebye"
44     for(i <- 0 until strs1.length)
45         print(s"$i:${strs1(i)} ")
46         println()
47
48     }
49 }
50
51

```

输出结果：

```

[I@47f37ef1
ArrayBuffer(0, 0, 0, 0, 0, 0, 0, 0)
ArrayBuffer(10)
10
spark
0:0 1:0 2:0 3:0 4:0 5:0 6:0 7:0 8:0 9:0
0:null 1:null 2:null 3:null 4:null 5:null 6:null 7:null 8:null 9:null
0:hello 1:world |
0:byebye 1:world

```

定长数组是不可变集合吗？

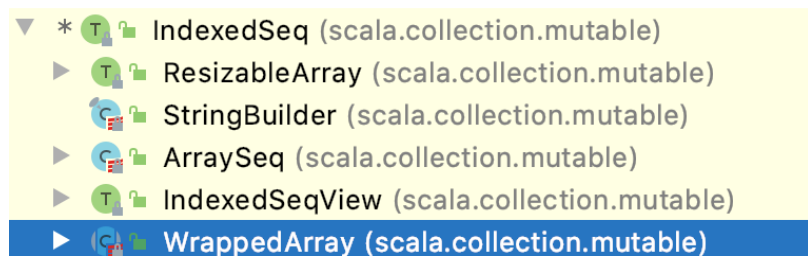
不是。定长数组是可变集合的一种，内容可变，但是其长度不可变。

```
1 # 扩展：为什么定长数组是可变集合？
2   Array本身不属于scala集合成员，从前面类继承图中也可发现这一点，
   在可变集合图中IndexedSeq有一条虚线指向了Array，说明并不是直接继
   承关系。
3   一般将Array归为集合是因为Scala默认将Array隐式转换为
   WrappedArray，而WrappedArray实现了IndexedSeq特质。
4   从这一点上来说，String与WrappedString也有异曲同工之妙，可以发
   现在不可变集合中，String与IndexedSeq也是虚线连接，也就是说在
   Scala中，String可以当集合处理。参考下文中的描述，请自行通过源码验
   证。
```

Array被隐式转换为

WrappedArray (scala.collection.mutable.WrappedArray) ，间接拥有了集合的特征。

```
1 //Predef.scala 提示：所有scala文件默认导入Predef对象
2 implicit def genericWrapArray[T](xs: Array[T]):
   WrappedArray[T] =
3     if (xs eq null) null
4     else WrappedArray.make(xs)
5
```



4.1.1.2 变长数组ArrayBuffer

定义变长数组的方式:

```

scala> import scala.collection.mutable.ArrayBuffer
import scala.collection.mutable.ArrayBuffer

scala> var arr1=ArrayBuffer[Int]()
arr1: scala.collection.mutable.ArrayBuffer[Int] = ArrayBuffer()

scala> arr1.length
res2: Int = 0

scala> var arr2=ArrayBuffer[Int](10)
arr2: scala.collection.mutable.ArrayBuffer[Int] = ArrayBuffer(10)

scala> arr2.length
res3: Int = 1

scala> var arr3=ArrayBuffer(10)
arr3: scala.collection.mutable.ArrayBuffer[Int] = ArrayBuffer(10)

scala> arr3.length
res4: Int = 1

scala> var arr4=new ArrayBuffer[Int]()
arr4: scala.collection.mutable.ArrayBuffer[Int] = ArrayBuffer()

scala> arr4.length
res5: Int = 0

scala> var arr5=new ArrayBuffer[Int](10)
arr5: scala.collection.mutable.ArrayBuffer[Int] = ArrayBuffer()

scala> arr5.length
res6: Int = 0

```

使用 ArrayBuffer定义长度按需变化的数组。

```

1  import scala.collection.mutable.ArrayBuffer
2
3  object VarArrayDemo {
4      def main(args: Array[String]){
5          //定义一个空的可变长Int型数组
6          val nums = ArrayBuffer[Int]()
7
8          //在尾端添加元素
9          nums += 1
10
11         //在尾端添加多个元素
12         nums += (2,3,4,5)
13
14         //使用++=在尾端添加任何集合
15         nums ++= Array(6,7,8)
16
17         //这些操作符，有相应的 -= ， --=可以做数组的删减，用法同+=，
        ++=

```

```
18
19 //使用append追加一个或者多个元素
20     nums.append(1)
21     nums.append(2,3)
22
23 //在下标2之前插入元素
24     nums.insert(2,20)
25     nums.insert(2,30,30)
26
27 //移除最后2元素
28     nums.trimEnd(2)
29 //移除最开始的一个或者多个元素
30     nums.trimStart(1)
31
32 //从下标2出移除一个或者多个元素
33     nums.remove(2)
34     nums.remove(2,2)
35
36 //使用增强for循环进行数组遍历
37     for(elem <- nums)
38         println(elem)
39
40 //基于下标访问使用增强for循环进行数组遍历
41     for(i <- 0 until nums.length)
42         println(nums(i))
43 }
44 }
```

执行结果：

```
1 2 3 4 5 6 7 8
1 2 30 30 20 3 4 5 6
1 2 3 4 5 6
```

4.1.1.3 定长数组与变长数组的转换

```
1 arr1.toBuffer //转为变长
2 arr2.toArray  //转为定长
```

4.1.1.4 遍历数组

1.增强for循环,参见变长数组的代码

```
1 //使用增强for循环进行数组遍历
2 for(elem <- nums)
3   println(elem)
```

2.使用until生成脚标, `0 until 10` 包含0但不包含10

```
1 //基于下标访问使用增强for循环进行数组遍历
2 for(i <- 0 until nums.length)
3   println(nums(i))
```

4.1.1.5 数组元素处理

课上练习:

```
1 # 现有数组包含元素1-9
2 - 1.对每个元素乘2后输出
3 - 2.对每个偶数乘2后输出
```

分析：使用yield关键字将原始的数组进行转换会产生一个新的数组，原始的数组不变。也可以通过高阶函数filter、map完成，这些高阶函数类似于Java中的Lambda表达式，很明显高阶函数更灵活，推荐使用高阶函数实现

```
1 object ArrayTransfer {
2
3     def main(args: Array[String]): Unit = {
4
5         //使用for推导式生成一个新的数组
6         val a = Array(1, 2, 3, 4, 5, 6, 7, 8, 9)
7         val res1 = for(elem <- a) yield 2*elem
8         for(elem <- res1)
9             print(elem+" ")
10        println()
11
12        //对原数组元素过滤后生成一个新的数组
13        //将偶数取出乘以10后再生成一个新的数组
14        val res2 = for(elem <- a if elem%2 == 0)yield
15        2*elem
16        for(elem <- res2)
17            print(elem+" ")
18        println()
19
20        //使用filter和map转换出新的数组
21        val res3 = a.filter(_ % 2 == 0).map(2 * _)
22        for(elem <- res3)
23            print(elem+" ")
24        println()
25    }
```

执行结果：

```
2 4 6 8 10 12 14 16 18
4 8 12 16
4 8 12 16
```

4.1.1.6 数组常用方法

在Scala中，数组上的某些方法对数组进行相应的操作非常方便。

```
1 object ArrayAlgorithm {
2   def main(args: Array[String]): Unit = {
3
4     val arr = Array(9,1,3,4,2,7,5,6,8)
5     println(arr.sum) // 数组求和:45
6     println(arr.max) // 数组求最大值:9
7     println(arr.min) // 数组求最小值:1
8     println(arr.sorted.toBuffer) // 数组排序:默认升序, 返回
    一个新的数组
9     println(arr.sortWith(_<_).toBuffer) // 可以自定义排序
    规则, 返回一个新的数组
10  }
11 }
```

4.1.1.7 数组进阶

课上练习

```
1 //创建二维数组
2 def main(args: Array[String]) {
3     var myMatrix = ofDim[Int](3,3)
4
5     // 创建矩阵
```

```

6      for (i <- 0 to 2) {
7          for ( j <- 0 to 2) {
8              myMatrix(i)(j) = j;
9          }
10     }
11
12     // 打印二维阵列
13     for (i <- 0 to 2) {
14         for ( j <- 0 to 2) {
15             print(" " + myMatrix(i)(j));
16         }
17         println();
18     }
19     //不规则多维数组
20     val multiDimArr = new Array[Array[Int]](3)
21     multiDimArr2(0) = new Array[Int] (2)
22     multiDimArr2(1) = new Array[Int] (3)
23     multiDimArr2(2) = new Array[Int] (4)
24     multiDimArr2(1)(2)(3) = 1
25
26 }

```

4.1.2 元组(Tuple)

Scala Tuple表示固定元素的组合，元组可以装着多个不同类型的值，是不同类型的值的聚集。Tuple是Scala中非常重要的一种数据结构，后面会大量使用。其特点包括：

- 1 1.最多支持22个元素组合，分别对应类型Tuple1~Tuple22，相应也称为一元组（一般不用）、二元组、三元组...
- 2 2.元组可以容纳不同类型的元素
- 3 3.元组不可变

特别地：二元组可以表示Map中的一个元素，Map是K/V对偶的集合，对偶是元组的最简单形式。

4.1.2.1 创建访问元组

创建元组：使用小括号 `()` 将多个元素括起来，元素之间用逗号分隔，元素的类型和个数不超过22。

访问组元：使用 `_1`，`_2`，`_3` 等形式访问组元，注意下标从1开始。

第二种方式定义的元组也可以通过 `a`，`b`，`c` 去访问组元。

```
scala> val t = ("qianfeng", "hadoop", 1)
t: (String, String, Int) = (qianfeng, hadoop, 1)
scala> val t, (a,b,c) = ("qianfeng", "hadoop", 1)
t: (String, String, Int) = (qianfeng, hadoop, 1)
a: String = qianfeng
b: String = hadoop
c: Int = 1
scala> val t_1 = t._1
t_1: String = qianfeng
```

创建元组

访问元组的第一个组元

另一种定义方式:

```
1 | val tuple3 = new Tuple3(1, 3.14, "Fred")
```

元组的实际类型取决于它的元素的类型，比如：

- `(99, "runoob")` 实际类型是 `Tuple2[Int, String]`
- `('u', 'r', "the", 1, 4, "me")` 实际类型为 `Tuple6[Char, Char, String, Int, Int, String]`

目前 Scala 支持的元组最大长度为 22。对于更大长度你可以使用集合，或者样例类（case class）。

4.1.2.2 元组访问

```
1 //注意元组元素的访问有下划线，并且访问下标从1开始
2 val value1 = tuple3._3
3 println(value1)
4 // 按照索引访问元组的第一个元素，从0开始
5 val value2 = tuple3.productElement(0)
6 println(value2)
```

4.1.2.3 元组遍历

因为元组本身不是集合成员，所以元组不能作为for生成器的表达式。但元组实现了 `productIterator()` 方法，该方法可以生成元组的迭代器 `Iterator`（继承自 `TraversableOnce`），迭代器提供了遍历集合的方法。这也是通常我们将Tuple视为集合的原因。

方式1:

```
1 for (elem <- tuple1.productIterator) {
2     print(elem)
3 }
4 println()
```

方式2:

```
1 tuple1.productIterator.foreach(i => println(i))
  //foreach是高阶函数
2 tuple1.productIterator.foreach(print(_))
```

4.1.2.4 拉链操作

元组拉链指通过zip方法把两个元组中的每个元素依次咬合，生成新元组，以便后续处理。

```
scala> val names = Array("tom", "jerry", "kitty")
names: Array[String] = Array(tom, jerry, kitty)

scala> val scores = Array(2, 3, 4)
scores: Array[Int] = Array(2, 3, 4)

scala> names.zip(scores)
res13: Array[(String, Int)] = Array((tom, 2), (jerry, 3), (kitty, 4))

scala> names.zip(scores)
res14: Array[(String, Int)] = Array((tom, 2), (jerry, 3), (kitty, 4))
```

注意：如果两个数组的元素个数不一致，拉链操作后生成的数组的长度为较小的那个数组的元素个数

4.1.2.5 拉链扩展

1、**zip**函数将传进来的两个参数中相应位置上的元素组成一个pair数组。如果其中一个参数元素比较长，那么多余的参数会被删掉。

```
scala> val numbers = Seq(0, 1, 2, 3, 4)
numbers: Seq[Int] = List(0, 1, 2, 3, 4)

scala> val series = Seq(0, 1, 1, 2, 3)
series: Seq[Int] = List(0, 1, 1, 2, 3)

scala> numbers.zip(series)
res14: Seq[(Int, Int)] = List((0, 0), (1, 1), (2, 1), (3, 2), (4, 3))
```

2、**zipAll** 函数和上面的zip函数类似，但是如果其中一个元素个数比较少，那么将用默认的元素填充。

```
scala> val xs = List(1, 2, 3)
xs: List[Int] = List(1, 2, 3)

scala> val ys = List('a', 'b')
ys: List[Char] = List(a, b)

scala> val zs = List("I", "II", "III", "IV")
zs: List[String] = List(I, II, III, IV)

scala> val x = 0
x: Int = 0

scala> val y = '_'
y: Char = _

scala> val z = ""
z: String = _

scala> xs.zipAll(ys, x, y)
res15: List[(Int, Char)] = List((1, a), (2, b), (3, _))

scala> ys.zipAll(zs, x, y)
res16: List[(AnyVal, Any)] = List((a, I), (b, II), (0, III), (0, IV))

scala> xs.zipAll(zs, x, z)
res17: List[(Int, String)] = List((1, I), (2, II), (3, III), (0, IV))
```

3、zipWithIndex函数将元素和其所在的位置索引组成一个pair。

该方法把集合中的每个元素和该元素的索引进行一个拉链操作

```
1 scala> series.zipWithIndex
2 res11: Seq[(Int, Int)] = List((0,0), (1,1), (2,2),
   (3,3), (4,4), (5,5), (8,6), (15,7))
3
4 //如果索引值想从指定的位置开始, 使用zip
5 scala> series.zip(Stream from 1)
6 res14: Seq[(Int, Int)] = List((0,1), (1,2), (2,3),
   (3,4), (4,5), (5,6), (8,7), (15,8))
7
8 scala> for ((value,index) <- series.zip(Stream from 1))
   println(index+" "+value)
9 1 0
10 2 1
11 3 2
12 4 3
13 5 4
14 6 5
15 7 8
16 8 15
17
```

4、unzip函数可以将一个元组的列表转变成一个列表的元组

```

1 scala> val numbers = Seq(0,1,2,3,4)
2 numbers: Seq[Int] = List(0, 1, 2, 3, 4)
3
4 scala> val series = Seq(0,1,1,2,3)
5 series: Seq[Int] = List(0, 1, 1, 2, 3)
6
7 scala> numbers zip series
8 res16: Seq[(Int, Int)] = List((0,0), (1,1), (2,1),
   (3,2), (4,3))
9
10 scala> res16 unzip
11 warning: there were 1 feature warning(s); re-run with -
   feature for details
12 res17: (Seq[Int], Seq[Int]) = (List(0, 1, 2, 3,
   4),List(0, 1, 1, 2, 3))

```

4.1.3 映射 (Map)

在Scala中，把哈希表这种数据结构叫做映射。Scala中的Map存储的内容是键值对(key-value)，Map区分可变Map (scala.collection.mutable.Map) 和不可变Map (scala.collection.immutable.Map)。不可变的Map（仅TreeMap）支持有序，而可变的Map是无序的。

4.1.3.1 构建映射

Map中的元素为二元组，可用两种方式表示。

```

1 ("a", 1)
2 "a" -> 1

```

课上练习

```

1 //构建一个不可变的Map，默认即为不可变Map
2 //其中的元素其实是Tuple2
3 val scores = Map("zhangsan" -> 90, "lisi" -> 80, "wangwu" -> 70)
4
5 //使用元组方式构建
6 val scores = Map(("zhangsan", 90), ("lisi", 80),
7   ("wangwu", 70))
8
9 //构建一个可变的map，注意包名
10 val scores =
11   scala.collection.mutable.Map(("zhangsan", 90),
12     ("lisi", 80), ("wangwu", 70))

```

4.1.3.2 访问映射中的值

根据键获取map中对应的值，可以有以下三种方法，推荐使用getOrElse方法。

```

1 //如果key存在，则返回对应的值
2 //如果key不存在，则抛出异常
3 [java.util.NoSuchElementException]
4 //在Java中,如果key不存在则返回null
5 scala> val score1 = scores("lisi")
6 score1: Int = 80
7
8 //使用contains方法检查是否存在key对应的值
9 //使用contains先判断在取值，可以防止异常，并加入相应的处理逻辑
10 // 返回Boolean, true或者false
11 // 如果key存在，则返回true
12 // 如果key不存在，则返回false
13 map4.contains("B")

```

```

14 scala> val score2 = if(scores.contains("lisi"))
    scores("lisi") else 0
15 score2: Int = 80
16
17 //使用get方法取值, 返回Option对象, Some或者None
18 //如果返回some, 可以进一步通过get方法取回相应的值
19 //如果返回None, 通过get方法取值, 抛出异常
    java.util.NoSuchElementException: None.get
20 var map4 = mutable.Map( ("A", 1), ("B", "北京"), ("C",
    3) )
21 println(map4.get("A")) //Some
22 println(map4.get("A").get) //得到Some在取出
23
24 //使用getOrElse()取值
25 //def getOrElse[V1 >: V](key: K, default: => V1)
26 //如果key存在, 返回key对应的值。
27 //如果key不存在, 返回默认值。
28 scala> val score3 = scores.getOrElse("lisi",0)
29 score3: Int = 80

```

4.1.3.3 修改可变Map信息

遍历访问map

```

1 //修改键对应的值
2 //可变的map才能修改
3 //key存在, 则修改对应的值, key不存在, 则添加键值对
4 scala> scores("lisi") = 100
5
6 scala> scores
7 res23: scala.collection.mutable.Map[String,Int] =
    Map(lisi -> 100, zhangsan -> 90, wangwu -> 70)
8
9 scala> scores.update("lisi",50)

```

```
10
11 scala> scores
12 res25: scala.collection.mutable.Map[String,Int] =
    Map(lisi -> 50, zhangsan -> 90, wangwu -> 70)
13
14 //添加单个元素-方式1, 如果key存在, 则修改相应key的值。
15 scala> scores("zhaoliu") = 88
16
17 scala> scores
18 res27: scala.collection.mutable.Map[String,Int] =
    Map(lisi -> 50, zhangsan -> 90, zhaoliu -> 88, wangwu -
    > 70)
19 //添加单个元素-方式2
20 scala> scores +=("tom"->77)
21 res28: scores.type = Map(lisi -> 50, zhangsan -> 90,
    tom -> 77, zhaoliu -> 88, wangwu -> 70)
22
23 //添加多个元素-方式1
24 scala> scores = scores + ("tom"->77,"jerry"->88)
25 scala> scores +=("tom"->77,"jerry"->88)
26 res28: scores.type = Map(lisi -> 50, zhangsan -> 90,
    tom -> 77, zhaoliu -> 88, wangwu -> 7,jerry -> 88)
27 scala> val scores2 = Map(("za",90),("lq",80),("wg",70))
28 scores2: scala.collection.immutable.Map[String,Int] =
    Map(za -> 90, lq -> 80, wg -> 70)
29 //添加多个元素-方式2
30 scala> scores ++= scores2
31 res30: scores.type = Map(lisi -> 50, zhangsan -> 90, lq
    -> 80, tom -> 77, zhaoliu -> 88, za -> 90, wg -> 70,
    wangwu -> 70)
32 //移除键值对
33 scala> scores -= "lisi"
```



```

34 res31: scala.collection.mutable.Map[String,Int] =
    Map(zhangsan -> 90, tom -> 77, lq -> 80, zhaoliu -> 88,
    za -> 90, wg -> 70, wangwu -> 70)
35 //移除多个键一
36 scala> scores--List("zhangsan","tom")
37 res33: scala.collection.mutable.Map[String,Int] =
    Map(lisi -> 50, lq -> 80, zhaoliu -> 88, za -> 90, wg -
    > 70, wangwu -> 70)
38 //移除多个键二
39 scala> scores-("lisi","lq")
40 res34: scala.collection.mutable.Map[String,Int] =
    Map(zhangsan -> 90, tom -> 77, zhaoliu -> 88, za -> 90,
    wg -> 70, wangwu -> 70)
41
42

```

4.1.3.4 遍历map

```

1 //遍历
2 //返回一个set集合
3 scala> val res = scores.keySet
4 res: scala.collection.Set[String] = Set(lisi, zhangsan,
    lq, tom, zhaoliu, za, wg, wangwu)
5
6 scala> for(elem <- res)
7     | print(elem + " ")
8 lisi zhangsan lq tom zhaoliu za wg wangwu
9 //返回Map中所有key的迭代器
10 scala> val ite = scores.keys
11 ite: Iterable[String] = Set(lisi, zhangsan, lq, tom,
    zhaoliu, za, wg, wangwu)
12
13 scala> for (item <- ite)

```

```

14         | print(item + " ")
15 lisi zhangsan lq tom zhaoliu za wg wangwu
16
17 //返回Map中所有值的迭代器
18 scala> val values = scores.values
19 values: Iterable[Int] = HashMap(50, 90, 80, 77, 88, 90,
20 70, 70)
21
22 //返回键值对
23 scala> for (item <- scores)
24         | print(item+" ")
25 (lisi,50) (zhangsan,90) (lq,80) (tom,77) (zhaoliu,88)
26 (za,90) (wg,70) (wangwu,70)
27 //使用k、v表示二元组中的键和值
28 scala> for ((k,v) <- scores)
29         | print(k+": "+v+" ")
30 lisi:50  zhangsan:90  lq:80  tom:77  zhaoliu:88  za:90
31 wg:70  wangwu:70

```

4.1.3.5 HashMap

HashMap是最常用的数据结构之一，查找和增删元素具有O(1)的时间复杂度。同时HashMap也是默认的Map类型。

```

1 //可变Map
2 scala> var map4 = mutable.Map( ("A", 1), ("B", "北京"),
  ("C", 3) )
3 map4: scala.collection.mutable.Map[String,Any] = Map(A
  -> 1, C -> 3, B -> 北京)
4 scala> map4.getClass
5 res208: Class[_ <:
  scala.collection.mutable.Map[String,Any]] = class
  scala.collection.mutable.HashMap
6
7 //不可变Map
8 scala> var map5 = Map( ("A", 1), ("B", "北京"), ("C", 3)
  , ("D", 4), ("E", 5))
9 map5: scala.collection.immutable.Map[String,Any] =
  Map(E -> 5, A -> 1, B -> 北京, C -> 3, D -> 4)
10
11 scala> map5.getClass
12 res211: Class[_ <:
  scala.collection.immutable.Map[String,Any]] = class
  scala.collection.immutable.HashMap$HashTrieMap

```

可变HashMap操作示例:

```

1 import scala.collection.mutable
2
3 object MutMapDemo extends App{
4   val map1 = new mutable.HashMap[String, Int]()
5   //向map中添加数据
6   map1("spark") = 1
7   map1 += (("hadoop", 2)) //注意外面括号, 等价于map1.+=(
  ("hadoop", 2))
8   map1.put("storm", 3)
9   println(map1)

```

```

10
11 //从map中移除元素
12 map1 -= "spark"
13 map1.remove("hadoop")
14 println(map1)
15 }

```

4.1.3.6 TreeMap

返回按特定顺序排列的元素集合。

```

1 import scala.collection.immutable.TreeMap
2 var tm = TreeMap(3 -> 'x', 1 -> 'x', 4 -> 'x')
3 tm += (2 -> 'x')
4 tm //Map(1 -> x, 2 -> x, 3 -> x, 4 -> x)
5

```

4.1.4 列表 (List)

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，其次列表是链表结构，而数组不是。

- 1 1.列表中的元素类型必须相同。
- 2 2.列表是有序的。
- 3 3.列表是不可变的，内容及长度都不可变。

4.1.4.1 List构造

注意：任何对List的修改操作都会返回新的List，而原List不会变化。

```

1 //空列表

```

```

2 //可以使用Nil构建一个空列表 Nil=List[Nothing]
3 val empty = Nil
4 //也可以使用空元素构建一个空列表
5 val empty : List[Nothing] = List()
6
7 scala> val fruit = List("apples", "oranges", "pears")
8 fruit: List[String] = List(apples, oranges, pears)
9 scala> val diag3 =
10     |      List(
11     |          List(1, 0, 0),
12     |          List(0, 1, 0),
13     |          List(0, 0, 1)
14     |      )
15 diag3: List[List[Int]] = List(List(1, 0, 0), List(0, 1,
0), List(0, 0, 1))
16 scala> val empty = List()
17 empty: List[Nothing] = List()
18
19 //所有的列表可以看做是由两个基础构造快Nil和中缀操作符::构成, ::
表示从列表前段扩展。
20 //在Scala中列表要么为空 (Nil表示空列表) 要么是一个head元素加上
一个tail列表。
21 //9 :: List(5, 2) :: 操作符是将给定的头和尾创建一个新的列表
22 //注意: :: 操作符是右结合的, 如9 :: 5 :: 2 :: Nil相当于 9 ::
(5 :: (2 :: Nil))
23
24 scala> val fruit = "apples" :: ("oranges" :: ("pears"
:: Nil))
25 fruit: List[String] = List(apples, oranges, pears)
26 scala> val diag3 = (1 :: (0 :: (0 :: Nil))) :: (0 :: (1
:: (0 :: Nil))) :: (0 :: (0 :: (1 :: Nil))) :: Nil
27 diag3: List[List[Int]] = List(List(1, 0, 0), List(0, 1,
0), List(0, 0, 1))

```

```

28 scala> val empty = Nil
29 empty: scala.collection.immutable.Nil.type = List()
30 //注意: :: 操作符是右结合的, 如9 :: 5 :: 2 :: Nil相当于 9 ::
    (5 :: (2 :: Nil))
31 scala> val nums = 1::(2::(3::(4::Nil)))
32 nums: List[Int] = List(1, 2, 3, 4)
33 scala> val nums = 1 :: 2 :: 3 :: 4 :: Nil
34 nums: List[Int] = List(1, 2, 3, 4)
35 //注意是否加Nil的区别
36 scala> val nums = 1 :: 2 :: 3 :: 4
37 <console>:7: error: value :: is not a member of Int
38     val nums = 1 :: 2 :: 3 :: 4
39                     ^
40
41 scala> val nums = 1 :: 2 :: 3 :: 4 :: list0
42 nums: List[Int] = List(1, 2, 3, 4, 5, 6)
43 //列表中的元素可以是不同类型
44 scala> val nums = 1 :: 2 :: 3 :: 4 :: list0 :: Nil
45 nums: List[Any] = List(1, 2, 3, 4, List(5, 6))

```

4.1.4.2 List的访问

```

1 scala> val nums = 1 :: 2 :: 3 :: 4 :: list0 :: Nil
2 nums: List[Any] = List(1, 2, 3, 4, List(5, 6))
3
4 scala> nums(1)
5 res22: Any = 2
6
7 scala> nums(3)
8 res23: Any = 4

```

4.1.4.3 List的遍历

```
1  /**
2   * List的各种遍历方式
3   */
4  val lst = List(1,2,3,4,5);
5  print("foreach遍历: ")
6  lst.foreach { x => print(x+",") } //foreach遍历,这个是传统遍历,新手不熟无奈之下可以用它
7  println("")
8
9  var temp = lst.map { x => x+1 } //遍历,与foreach的区别是返回值为List 【B】
10 println("map遍历: "+temp.mkString(", "));
11
12
13 var temp1 = lst.reduceLeft((sum,i)=>sum +i) //遍历,返回值类型是一个与集合相同的Int
14 println("reduce遍历返回Int: "+temp1);
15
16 var temp2 = lst.foldLeft(List[Int]())((x,y)=>y::x); //遍历,返回值是自定义类型
17 //ps fold类函数还可以改成 : \ , /:的形式,代码精简了不少,但是可读性却减低了例如
18 println("foldLeft遍历返回自定义类型: "+temp2.mkString(", "));
19
20 var temp3=( List[Int]() /: lst){(m,c)=>c::m} //遍历,实现反转
21 println("foldLeft遍历实现反转: "+temp2.mkString(", "));
22
```

4.1.4.4 List的追加

```
1  object ImmutListDemo {
2
3      def main(args: Array[String]) {
4          //创建一个不可变的集合
5          val lst1 = List(1,2,3)
6          //将0插入到lst1的前面生成一个新的List ::和+: 右结合
7          // ::为右结合操作符
8          // 将元素追加到集合开头
9          val lst2 = 0 :: lst1
10         val lst3 = lst1.::(0)
11         val lst4 = 0 +: lst1
12         val lst5 = lst1.+: (0)
13         //将一个元素添加到lst1的后面产生一个新的集合, :+左结合
14         val lst6 = lst1 :+ 3
15         val lst0 = List(4,5,6)
16         //将2个list合并成一个新的List
17         val lst7 = lst1 ++ lst0
18         //将lst1插入到lst0前面生成一个新的集合
19         val lst8 = lst1 ++: lst0
20         //将lst0插入到lst1前面生成一个新的集合
21         val lst9 = lst1.:::(lst0)
22         println(lst9)
23     }
24
25 }
26
```

4.1.4.5 List的基本操作

对列表的所有操作可以表达为一下三种

head 返回列表的第一个元素。tail 返回除第一个元素之外所有元素组成的列表。

isEmpty 返回列表是否为空。

注意：其中tail head作用在空列表上，会报异常。

```
1 val list0 = List(1,2,3)
2 val list1 = List(4,5,6)
3 // head:返回列表第一个元素
4 list0.head // Int = 1
5 // tail:返回除了第一个元素之外的其他元素，以列表返回
6 list0.tail // List[Int] = List(2, 3)
7 // isEmpty:判断列表是否为空，为空时返回true
8 list0.isEmpty // Boolean = false
9 // concat:连接列表，返回一个新的集合
10 List.concat(list0,list1) // List[Int] = List(1, 2, 3,
11 4,
12 5, 6)
13 // fill:使用数个相同的元素创建一个列表
14 List.fill(10)(2) // 重复次数:10, 重复元素:2
15 // reverse:将列表顺序反转，返回一个新的集合
16 list0.reverse
17 //求列表长度
18 scala> List(1,2,3).length
19 res11: Int = 3
```

4.1.4.6 List的常用方法

```
1 //与head, tail对应的last init
2 scala> val abcde = List('a', 'b', 'c', 'd', 'e')
3 abcde: List[Char] = List(a, b, c, d, e)
4 scala> abcde.last
```

```

5 res12: Char = e
6 scala> abcde.init
7 res13: List[Char] = List(a, b, c, d)
8
9 //反转列表
10 scala> abcde.reverse
11 res14: List[Char] = List(e, d, c, b, a)
12
13 //tail,init的泛化
14 //提取列表的前n个元素
15 scala> abcde take 2
16 res15: List[Char] = List(a, b)
17 //丢弃列表的前n个元素
18 scala> abcde drop 2
19 res16: List[Char] = List(c, d, e)
20 scala> abcde splitAt 2
21 res17: (List[Char], List[Char]) = (List(a, b), List(c,
    d, e))
22 scala> List(abcde take 2, abcde drop 2)
23 res18: List[List[Char]] = List(List(a, b), List(c, d,
    e))
24
25 //拉链操作
26 scala> abcde zip List(1,2,3,4,5)
27 res22: List[(Char, Int)] = List((a,1), (b,2), (c,3),
    (d,4), (e,5))

```

4.1.4.7 List的模式拆分

列表可以使用模式匹配进行拆分。

```

1 scala> val fruit = List("apples", "oranges", "pears")
2 fruit: List[String] = List(apples, oranges, pears)
3 //给出的元素个数与列表的元素个数一致，否则会报错。

```

```

4 scala> val List(a, b, c) = fruit
5 a: String = apples
6 b: String = oranges
7 c: String = pears
8 //在元素个数不可知的情况下,最好使用::匹配。
9 scala> val a :: b :: rest = fruit
10 a: String = apples
11 b: String = oranges
12 rest: List[String] = List(pears)
13
14 //合并操作
15 scala> List(1, 2) ::: List(3, 4, 5)
16 res8: List[Int] = List(1, 2, 3, 4, 5)
17 scala> List(1, 2) ::: List(3, 4, 5) ::: List(6,7,8)
18 res9: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8)
19 scala> List(1, 2) ::: (List(3, 4, 5) ::: List(6,7,8))
20 res10: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8)

```

4.1.4.8 List的高阶方法

```

1 //映射: map, flatMap, foreach
2 scala> List(1,2,3).map(_+1)
3 res23: List[Int] = List(2, 3, 4)
4
5 scala> val words = List("the", "quick", "brown", "fox")
6 words: List[String] = List(the, quick, brown, fox)
7 scala> words.map(_.length)
8 res24: List[Int] = List(3, 5, 5, 3)
9 scala> words map (_.toList.reverse.toString)
10 res25: List[String] = List(List(e, h, t), List(k, c, i,
    u, q), List(n, w, o, r, b), List(x, o, f))
11 scala> words map (_.toList.reverse.mkString)
12 res26: List[String] = List(eht, kciuq, nworb, xof)

```

```
13 scala> words map (_.toList)
14 res27: List[List[Char]] = List(List(t, h, e), List(q,
    u, i, c, k), List(b, r, o, w, n), List(f, o, x))
15 scala> words flatMap (_.toList)
16 res28: List[Char] = List(t, h, e, q, u, i, c, k, b, r,
    o, w, n, f, o, x)
17 scala> var sum = 0
18 sum: Int = 0
19 scala> List(1, 2, 3, 4, 5) foreach (sum += _)
20
21 //过滤 filter, partition, find, takeWhile, dropWhile,
    span
22 scala> List(1, 2, 3, 4, 5) filter (_ % 2 == 0)
23 res37: List[Int] = List(2, 4)
24 scala> words filter (_.length == 3)
25 res38: List[java.lang.String] = List(the, fox)
26 scala> List(1, 2, 3, 4, 5) partition (_ % 2 == 0)
27 res30: (List[Int], List[Int]) = (List(2, 4), List(1, 3,
    5))
28
29 scala> List(1,2,3,4,5)find(_%2==0)
30 res31: Option[Int] = Some(2)
31 scala> List(1, 2, 3, 4, 5) find (_ <= 0)
32 res32: Option[Int] = None
33
34 scala> List(1, 2, 3, -4, 5) takeWhile (_ > 0)
35 res33: List[Int] = List(1, 2, 3)
36 scala> words dropWhile (_ startsWith "t")
37 res34: List[String] = List(quick, brown, fox)
38 scala> List(1, 2, 3, -4, 5) span (_ > 0)
39 res35: (List[Int], List[Int]) = (List(1, 2, 3), List(-4,
    5))
40 //排序sortWith
```

```
41 scala> words.sortWith(_.length > _.length)
42 res40: List[String] = List(quick, brown, the, fox)
```

4.1.4.9 可变列表ListBuffer

ListBuffer与List的区别在于：修改ListBuffer改变的是本身。

```
1  import scala.collection.mutable.ListBuffer
2
3  object MutListDemo extends App{
4      //构建一个可变列表，初始有3个元素1,2,3
5      val lst0 = ListBuffer[Int](1,2,3)
6      //创建一个空的可变列表
7      val lst1 = new ListBuffer[Int]
8      //向lst1中追加元素，注意：没有生成新的集合
9      lst1 += 4
10     lst1.append(5)
11
12     //将lst1中的元素追加到lst0中， 注意：没有生成新的集合
13     lst0 ++= lst1
14
15     //将lst0和lst1合并成一个新的ListBuffer 注意：生成了一个集合
16     val lst2= lst0 ++ lst1
17
18     //将元素追加到lst0的后面生成一个新的集合
19     val lst3 = lst0 :+ 5
20 }
```

4.1.5 Set

Set中的元素不可重复，无序（TreeSet除外）。

4.1.5.1 创建集合

不可变Set

```
1 import scala.collection.immutable.HashSet
2 val set0=Set(1,2,3,4,5)
3 println(set0.getClass)
  //scala.collection.immutable.HashSet
4
5 val set1 = new HashSet[Int]() //默认的Set
6 // 可以使用加号追加元素, 会产生一个新的集合, 原有set不变
7 val set2 = set1 + 1
8 // set集合中不会出现重复的元素
9 val set3 = set2 ++ Set(1,2,3)
10 val set4 = Set(1,2,3,4,5,6)
11 // set集合中的元素是无序的
12 print(set4)
13
14
15 // SortedSet中是有序的
16 val
  set5=scala.collection.immutable.TreeSet(1,1,10,5,6,7,8,
  11,13)
17
```

可变Set

```
1 import scala.collection.mutable
2
3 object MutSetDemo extends App{
4
5     val mutableSet = Set(1,2,3)
6     println(mutableSet.getClass.getName) //
  scala.collection.mutable.HashSet

```

```

7
8 //创建一个可变的HashSet
9 val set1 = new mutable.HashSet[Int]()
10 //向HashSet中添加元素
11 set1 += 2
12 //add等价于+=
13 set1.add(4)
14 //删除一个元素,如果删除的对象不存在,则不生效,也不会报错
15
16 set1 -= 5
17 set1.remove(2)
18 println(set1)
19 }

```

4.1.5.2 集合基本操作

Scala集合有三个基本操作：

- `head` 返回集合第一个元素
- `tail` 返回一个集合，包含除了第一元素之外的其他元素
- `isEmpty` 在集合为空时返回true

对于Scala集合的任何操作都可以使用这三个基本操作来表达。实例如下：

```

1 object Test {
2     def main(args: Array[String]) {
3         val site = Set("1000phone", "Google", "Baidu")
4         val nums: Set[Int] = Set()
5
6         println( "第一网站是 : " + site.head )
7         println( "最后一个网站是 : " + site.tail )
8         println( "查看列表 site 是否为空 : " + site.isEmpty
9     )
10        println( "查看 nums 是否为空 : " + nums.isEmpty )
11    }
12 }

```

查找集合中最大与最小元素 你可以使用 **Set.min** 方法来查找集合中的最小元素，使用 **Set.max** 方法查找集合中的最大元素。实例如下：

```

1 object Test {
2     def main(args: Array[String]) {
3         val num = Set(5,6,9,20,30,45)
4         //set集合遍历
5         for(x <- num) {
6             println(x)
7         }
8         // 查找集合中最大与最小元素
9         println( "Set(5,6,9,20,30,45) 集合中的最小元素是 : "
10    + num.min )
11        println( "Set(5,6,9,20,30,45) 集合中的最大元素是 : "
12    + num.max )
13    }
14 }

```


4.1.5.3 两个set之间的常用操作

你可以使用 `++` 运算符或 `Set.++()` 方法来连接两个集合。如果元素有重复的就会移除重复的元素。实例如下：

```
1 object Test {
2     def main(args: Array[String]) {
3         val site1 = Set("1000phone", "Google", "Baidu")
4         val site2 = Set("Faceboook", "Taobao")
5
6         // ++ 作为运算符使用
7         var site = site1 ++ site2
8         println( "site1 ++ site2 : " + site )
9
10        // ++ 作为方法使用
11        site = site1.++(site2)
12        println( "site1.++(site2) : " + site )
13    }
14 }
```

执行以上代码，输出结果为：

```
1 $ vim Test.scala
2 $ scala Test.scala
3 site1 ++ site2 : Set(Faceboook, Taobao, Google, Baidu,
4 1000phone)
5 site1.++(site2) : Set(Faceboook, Taobao, Google, Baidu,
6 1000phon)
7
```

4.1.5.4 交集

你可以使用 **Set.&** 方法或 **Set.intersect** 方法来查看两个集合的交集元素。实例如下：

```
1 object Test {
2     def main(args: Array[String]) {
3         val num1 = Set(5,6,9,20,30,45)
4         val num2 = Set(50,60,9,20,35,55)
5
6         // 交集
7         println( "num1.&(num2) : " + num1.&(num2) )
8         println( "num1.intersect(num2) : " +
9 num1.intersect(num2) )
10    }
```

执行以上代码，输出结果为：

```
1 num1.&(num2) : Set(20, 9)
2 num1.intersect(num2) : Set(20, 9)
```

```
1 import scala.collection.immutable.HashSet
2 val set1 = Set(5,6,9,20,30,45)
3 val set2 = Set(50,60,9,20,35,55)
4 // contains:是否包含某一元素, 包含返回true, 不包含返回false
5 set1.contains(10) // Boolean = false
6 // &:交集运算
7 set1 & set2
8 set1.intersect(set2) // Set(20,9)
9 // &~:差集运算
10 set1 &~ set2
11 set1.diff(set2) // Set(5, 6, 45, 30)
12 // union:并集运算
```

```

13 set1.union(set2)
14 // scala.collection.immutable.Set[Int] = Set(5, 20, 6,
15 60, 9, 45, 35, 50, 55, 30)
16 // count:计算满足指定条件的集合元素个数
17 val fun = (x:Int) => x > 10
18 set1.count(fun) // Int = 3
19 // iterator:获得一个迭代器，可用于集合遍历
20 val it = set1.iterator
21 while(it.hasNext){
22     println(it.next())
23 }
24 // size:返回集合元素的数量
25 // splitAt:将集合拆分为两个容器，第一个由前n个元素组成，第二个由
26 剩下的元素组成
27 set1.splitAt(5) // 返回一个元组
28 // take:返回前n个元素
29 // takeRight:返回后n个元
30 // 可以使用to{type}快速转换为其他集合类型

```

4.2 集合应用

4.2.1 集合的重要函数（重点）

```

1  /*
2     常用方法
3     基本上集合中都可以使用,及个别是使用不上
4     在Spark中常用集合:  Array List 元组  Map  Set
5  */
6  object CollectionMethodDemo {
7      def main(args: Array[String]): Unit = {
8          /*
9              map遍历集合处理集合中数据并返回一个全新集合

```

10 `filter`遍历集合提供一个函数指定过滤规则,对集合中数据继续过滤,满足需求的会被过滤出来形成一个新的集合

11 聚合函数(求和)`sum` 将集合中每一个元素相加求出最终和(具备可加性) --> 数值

12 `max`和`min` 最大值和最小值(了解)

13 `foreach`遍历集合主要是打印集合中数据(也可以操作集合中数据但是不能新的集合) --> 输出

14 `sorted` 排序 --> 升序 降序 可以考虑 `reverse`(翻转)

15 `sortwith` 需要传入比较参数,这个参数比较会决定是升序还是降序

16 第一个参数 > 第二个参数 降序

17 第一个参数 < 第二个参数 升序

18 `ps`:这些方法`Array`和`List`基本上都可以使用

19 `Map`和`Set`选择使用 元组也是选择使用

20 `*/`

21 `//1.Flatten` 可以将存储在集合内部的集合压平

22 `val list:List[List[Int]] =`

`List(List(1,2,3),List(4,5,6),List(7,8,9))`

23 `//List(1,2,3,4,5,6,7,8,9)`

24 `val flatten: List[Int] = list.flatten`

25 `println(flatten)`

26 `//set`集合中提供了特殊的操作

27 `//交集,并集和差集`

28 `val num1 = Set(5,6,9,20,30,45)`

29 `val num2= Set(50,60,9,20,35,55)`

30 `//交集`

31 `println(num1.intersect(num2))`

32 `//并集`

33 `println(num1.union(num2))`

34 `//差集`

35 `println(num1.diff(num2))`

36

```

37      //forall 对集合中的元素条件进行过滤,只有当所有元素都满足要
      要求是,才会返回true 否则是false
38      val list1 = List(1,2,3,4,5)
39      val bool:Boolean = list1.forall(_ < 3)
40
41      //Partition 分区
42      //ps:Scala中只能模拟,spark中是可以做到这个分区
43      //根据传入的函数 将是数据吸入到不同存储位置中
44      val list2 = List(1,2,3,4,5,6)
45      //返回值类型是一个元组
46      val tuple: (List[Int], List[Int]) =
list2.partition(_%2 ==0)
47      val list2_1 = tuple._1
48
49      //2个聚合函数
50      //1. fold 求和 比较特殊需要连个参数 一个是默认值,另外一个
      是计算逻辑
51      // 当前fold是典型的 柯里化
52      val list3 = List(1,2,3,4,5)
53      //计算
54      //[A1 >: A](z: A1)(op: (A1, A1) => A1): A1
55      //现在属于是单线程,效果是看出来, 使用par的时候回出现多线程
      状态,
56      val sum = list3.fold(2)((res,n)=>res+n)
57      println(sum)
58      //fold有连个变种 foldleft 和 foldright
59      //2.reduce 直接求和 通过计算逻辑 进行集合中数据的计算
60      val list4 = List(1,2,3,4,5)
61      val sum2 = list4.reduce((x,y)=>x+y)
62      //reduce也有两个变种 reduceleft 和 reduceright
63  }
64 }
65

```

4.2.2 并行集合

Scala为了充分使用多核CPU，提供了并行集合（有别于前面的串行集合），用于多核环境的并行计算。

主要用到的算法有：

```
1 # Divide and conquer : 分治算法, Scala通过splitters,  
  combiners等抽象层来实现, 主要原理是将计算工作分解很多任务, 分发给  
  一些处理器去完成, 并将它们处理结果合并返回  
2  
3 # Work stealin: 算法, 主要用于任务调度负载均衡 (load-  
  balancing), 通俗点完成自己的所有任务之后, 发现其他人还有活没干  
  完, 主动 (或被安排) 帮他人一起干, 这样达到尽早干完的目的。  
4
```

1) 打印1~5

```
1 (1 to 5).foreach(println(_))  
2 println()  
3 (1 to 5).par.foreach(println(_)) //paraller
```

2) 查看并行集合中元素访问的线程

```
1 val result1 = (0 to 10000).map{case _ =>  
  Thread.currentThread.getName}.distinct  
2 val result2 = (0 to 10000).par.map{case _ =>  
  Thread.currentThread.getName}.distinct  
3 println(result1)  
4 println(result2)
```

4.2.3 WordCount

```
1  /**
2   * 单词统计
3   */
4  object Scala_WordCount {
5   def main(args: Array[String]): Unit = {
6     //数据
7     val line = List("hello tom hello jerry", "hello
8 xiaobai hello", "hello tom")
9     // 要求将数据进行处理得到的结果
10
11     //List("hello", "tom", "hello", "jerry", "hello", "xiaobai"
12     , "hello", "hello", "tom")
13
14     /*flatten+Map实现
15     val words: List[Array[String]] = line.map(_.split("
16 "))
17     val flatten: List[String] = words.flatten
18     println(flatten)
19     */
20
21     //1.flatMap --> flatten+Map 具备在遍历处理数据同时,将数
22     据将数据进行扁平化处理
23     //在处理数据的时候一定要返回一个集
24     val words: List[String] = line.flatMap(_.split(" "))
25     println(words)
26     // 可以仿照MR中Map阶段,对单词拼接形成kv键值对 单词,1
27     val tuples: List[(String, Int)] = words.map((_, 1))
28     println(tuples)
29     //MR中kv键值对会发送到Reduce然后会出现相同key为一组计算一
30     次reduce
```

```

25      //因为处理是单词,仿照MR中Reduce端处理逻辑,相同key在一
    起,Scala中提供groupBy根据传入参数进行分组
26      //返回值时一个Map
27      val grouped: Map[String, List[(String, Int)]] =
    tuples.groupBy(_._1)
28      println(grouped)
29      //Map的组成是 key 是单词 value是具体的单词对应元组 其中元
    组是使用List存储
30      //此时只需要知道key所对应value中List元素的个数,就可以知道
    当前单词的个数了
31      //mapValues是处理Map集合中value的值,只操作value,原则是
    对应key的value值,返回值是一个Map[key是原有的可以,value是计算结
    果之后的value]
32      val sumed: Map[String, Int] =
    grouped.mapValues(_._2.size)
33      println(sumed)
34      val list: List[(String, Int)] = sumed.toList
35      /**
36       * sortBy分为两种一种是Scala版本 另外一种是Spark版本
37       * Scala版本在没有使用隐式转换自定义的前提下,sortBy只能升
    序不能降序
38       * Spark版本中sortBy有两个参数 一个是根据谁来排序, 第二个
    参数是Boolean类型 true是升序 false降序
39       */
40      //参数是一个比较原则
41      val sorted: List[(String, Int)] = list.sortBy(_._2)
42
43      //topN 降序排序
44      val top1 = sorted take 1
45      println(sorted)
46
47      //简化版本
48      //line.flatMap(_._2.split(" ")).map(_._1)

```



```

49      //
      .groupBy(_._1).mapValues(_._size).toList.sortBy(_._2).take(1)
50
51    }
52
53  }

```

4.2.4 Java和Scala集合的互操作

```

1  Scala中的集合和Java中集合互相转换  引用一个包
2  import scala.collection.JavaConverters._
3  val list:java.util.List[Int]=List(1,2,3,4,5).asJava //将
   Scala集合转换为Java集合
4  val buffer: mutable.Buffer[Int] = list.asScala //将Java集
   合转换为Scala
5  //toXXX 转换成你要的集合即可 --> xxx是集合名字 例如Array

```

4.3 高阶函数(重点)

4.3.1 高阶函数的概念

如果一个函数的传入参数为函数或者返回值是函数，则该函数即为高阶函数。前面提到的集合中大部分函数都属于高阶函数：

```

1  - map、flatMap
2  - filter
3  - fold、reduce
4  - forall、foreach
5  - partition
6  - groupBy
7  - sortBy

```


总结他们都有共同的特点：函数的参数是一个函数。

4.3.2 传入参数为函数

Scala中，函数是头等公民，和数字一样。不仅可以调用，还可以在变量中存放函数，也可以作为参数传入函数，或者作为函数的返回值。

```
scala> val arr = Array(1, 2, 3)
arr: Array[Int] = Array(1, 2, 3)

scala> val fun = (x: Int) => x*2
fun: Int => Int = <function1>

scala> arr.map(fun)  函数作为参数传入其他函数
res0: Array[Int] = Array(2, 4, 6)
```


4.3.3 传入参数为匿名函数

在Scala中，你不需要给每一个函数命名，就像不必给每个数字命名一样，将函数赋给变量的函数叫做匿名函数

```
scala> val arr = Array(1, 2, 3)
arr: Array[Int] = Array(1, 2, 3)

scala> val fun = (x: Int) => x*2
fun: Int => Int = <function1>


scala> arr.map(fun)
res0: Array[Int] = Array(2, 4, 6)

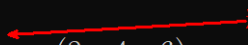
scala> arr.map((x: Int) => x*2)  直接将一个匿名函数作为参数传递给另一个函数
res1: Array[Int] = Array(2, 4, 6)
```

还可以

```
scala> arr.map(fun)
res0: Array[Int] = Array(2, 4, 6)

scala> arr.map((x: Int) => x*2 )
res1: Array[Int] = Array(2, 4, 6)

scala> arr.map( x => x*2 )  精简方式1
res2: Array[Int] = Array(2, 4, 6)

scala> arr.map( _*2 )  精简方式2
res3: Array[Int] = Array(2, 4, 6)
```

4.3.4 传入参数为方法

在Scala中，方法和函数是不一样的，最本质的区别是函数可以作为参数传递到方法中。

```
1 case class WeeklyWeatherForecast(temperatures:
  Seq[Double]) {
2
3   private def convertCtoF(temp: Double) = temp * 1.8 +
4     32
5   //方法convertCtoF作为参数传入
6   def forecastInFahrenheit: Seq[Double] =
7     temperatures.map(convertCtoF)
8 }
```

4.3.5 返回值为函数

```
1 //返回值为函数类型: (String, String) => String
2 def urlBuilder(ssl: Boolean, domainName: String):
3   (String, String) => String = {
4     val schema = if (ssl) "https://" else "http://"
5     (endpoint: String, query: String) =>
6       s"$schema$domainName/$endpoint?$query"
7   }
8
9 val domainName = "www.example.com"
10 def getURL = urlBuilder(ssl=true, domainName)
11 val endpoint = "users"
12 val query = "id=1"
13 val url = getURL(endpoint, query) //
14   "https://www.example.com/users?id=1": String
```

4.3.6 闭包

闭包是一个函数，返回值依赖于声明在函数外部的一个或多个变量。

函数体内可以方法相应作用域内的任何变量。

闭包通常来讲可以简单的认为是可以访问一个函数里面局部变量的另外一个函数。

普通函数：

```
1 | val multiplier = (i:Int) => i * 10
```

函数体内有一个变量 i，它作为函数的一个参数。

```
1 | val multiplier = (i:Int) => i * factor
```

在 multiplier 中有两个变量：i 和 factor。其中的一个 i 是函数的形式参数，在 multiplier 函数被调用时，i 被赋予一个新的值。然而，factor 不是形式参数，而是自由变量，考虑下面代码：

```
1 | var factor = 3
2 | val multiplier = (i:Int) => i * factor
```

这里我们引入一个自由变量 factor，这个变量定义在函数外面。

这样定义的函数变量 multiplier 成为一个"闭包"，因为它引用到函数外面定义的变量，定义这个函数的过程是将这个自由变量捕获而构成一个封闭的函数。

```

1 object Test {
2     def main(args: Array[String]) {
3         println( "multiplier(1) value = " + multiplier(1)
4         )
5         println( "multiplier(2) value = " + multiplier(2)
6         )
7     }
8     var factor = 3
9     val multiplier = (i:Int) => i * factor
10 }

```

4.3.7 柯里化

Currying函数，指的是，将原来接收两个参数的一个函数，转换为两个函数，第一个函数接收原先的第一个参数，然后返回接收原先第二个参数的第二个函数。

在函数调用的过程中，就变为了两个函数连续调用的形式

下面先给出一个普通的非柯里化的函数定义，实现一个加法函数：

```

1 scala> def plainOldSum(x:Int,y:Int) = x + y
2 plainOldSum: (x: Int, y: Int)Int
3 scala> plainOldSum(1,2)
4 res0: Int = 3

```

使用“柯里化”技术，把函数定义为多个参数列表：

```

1 scala> def curriedSum(x:Int)(y:Int) = x + y
2 curriedSum: (x: Int)(y: Int)Int
3 scala> curriedSum (1)(2)
4 res0: Int = 3

```

当你调用 `curriedSum (1)(2)` 时，实际上是依次调用两个普通函数（非柯里化函数），第一次调用使用一个参数 `x`，返回一个函数类型的值，第二次使用参数 `y` 调用这个函数类型的值，我们使用下面两个分开的定义在模拟 `curriedSum` 柯里化函数：

首先定义第一个函数：

```
1 scala> def first(x:Int) = (y:Int) => x + y
2 first: (x: Int)Int => Int
```

然后我们使用参数1调用这个函数来生成第二个函数。

```
1 scala> val second=first(1)
2 second: Int => Int = <function1>
3 scala> second(2)
4 res1: Int = 3
```

`first`，`second`的定义演示了柯里化函数的调用过程，它们本身和 `curriedSum` 没有任何关系，但是我们可以使用 `curriedSum` 来定义 `second`，如下：

```
1 scala> val onePlus = curriedSum(1)_
2 onePlus: Int => Int = <function1>
```

下划线“`_`”作为第二参数列表的占位符，这个定义的返回值为一个函数，当调用时会给调用的参数加一。

```
1 scala> onePlus(2)
2 res2: Int = 3
```

通过柯里化，你还可以定义多个类似 `onePlus` 的函数，比如 `twoPlus`

```
1 scala> val twoPlus = curriedSum(2) _
2 twoPlus: Int => Int = <function1>
3 scala> twoPlus(2)
4 res3: Int = 4
```

```
scala> val mul =(x: Int, y: Int) => x*y
mul: (Int, Int) => Int = <function2>

scala> val mulOneAtTime = (x: Int) => ((y: Int) => x*y) 柯里化
mulOneAtTime: Int => (Int => Int) = <function1>

scala> mulOneAtTime(6)(7)
res4: Int = 42

scala> def mulOneTime1(x: Int)(y: Int) = x*y 简写的柯里化
mulOneTime1: (x: Int)(y: Int)Int

scala> mulOneTime1(6)(7)
res5: Int = 42
```

五 实战应用

5.1 实战案例一

创建10个整数的随机列表，分别求奇数位与偶数位之和。

```
1 如: List(10, 9, 8, 7, 6, 5, 4, 3, 2, 1): 奇数位之各为
   10+8+6+4+2, 偶数位之和为9+7+5+3+1
```

分析

```
1 # 生成随机数集合
2 # 考虑使用zipWithIndex实现, 注意index以0开始, 变换使满足题意
3 # 拉链后按index奇偶分组, 然后组内求和
```

关键代码

```
1 import scala.Math
2 import scala.collection.mutable.ListBuffer
3 //初始化列表
4 //nums是Vector集合，类似Array，有着连续的内存空间
5 val nums=for(i <- 1 to 10) yield
  (Math.random()*100).toInt
6
7 //拉链操作
8 val rs=nums.zipWithIndex.groupBy(x=>x._2 % 2).map(kvs=>
  (kvs._1,kvs._2.unzip._1.sum))
9
10 //输出
11 for((k,v)<-rs){
12   if(k==0)
13     println(s"$nums 奇数位之和: $v")
14   else
15     println(s"$nums 偶数位之和: $v")
16 }
17
18 /*****
19 *****/
20 //拉链操作分解
21 /*
22 scala> nums.zipWithIndex //拉链
23 res92: scala.collection.immutable.IndexedSeq[(Int,
24   Int)] = Vector((67,0), (30,1), (26,2), (63,3), (23,4),
25   (19,5), (44,6), (59,7), (36,8), (67,9))
26
27 scala> nums.zipWithIndex.groupBy(x=>x._2 % 2) //按index
28 分2组
```



```

25 res93:
   scala.collection.immutable.Map[Int,scala.collection.imm
   mutable.IndexedSeq[(Int, Int)]] = Map(1 ->
   Vector((30,1), (63,3), (19,5), (59,7), (67,9)), 0 ->
   Vector((67,0), (26,2), (23,4), (44,6), (36,8)))
26
27 scala> nums.zipWithIndex.groupBy(x=>x._2 % 2).map(kvs=>
   (kvs._1,kvs._2.unzip)) //反拉链
28 res94: scala.collection.immutable.Map[Int,
   (scala.collection.immutable.IndexedSeq[Int],
   scala.collection.immutable.IndexedSeq[Int])] = Map(1 ->
   (Vector(30, 63, 19, 59, 67),Vector(1, 3, 5, 7, 9)), 0 -
   > (Vector(67, 26, 23, 44, 36),Vector(0, 2, 4, 6, 8)))
29
30 scala> nums.zipWithIndex.groupBy(x=>x._2 % 2).map(kvs=>
   (kvs._1,kvs._2.unzip._1))//去除index
31 res95:
   scala.collection.immutable.Map[Int,scala.collection.imm
   mutable.IndexedSeq[Int]] = Map(1 -> Vector(30, 63, 19,
   59, 67), 0 -> Vector(67, 26, 23, 44, 36))
32
33 scala> nums.zipWithIndex.groupBy(x=>x._2 % 2).map(kvs=>
   (kvs._1,kvs._2.unzip._1.sum))//求和
34 res96: scala.collection.immutable.Map[Int,Int] = Map(1
   -> 238, 0 -> 196)
35
36 //输出, 其中0表示奇数位, 1表示偶数位
37 //Map(1 -> 238, 0 -> 196)
38 */
39 /*****
   *****/

```

5.2 实战案例二

现有全国疫情信息 `yq.csv` 如下，其描述了各地每日新增、疑似人数情况。请使用Scala统计每日新增最多的城市。

```
1 dt,province,adds,possibles
2 2020-5-12,beijing,5,8
3 2020-5-12,shanghai,3,6
4 2020-5-13,beijing,3,5
5 2020-5-13,shanghai,2,1
```

分析

```
1 # 思考下面SQL
2 select max(adds) from yq group by dt
3 上面语句可以求出每日最多新增数据，无法获得每日新增最多的城市。可
  以改进如下：
4 select max(concat(adds,province)) from yq group by dt
5 结论：将adds与province合并，但仍然按照adds最大值。
6 # Scala思路
7 - 读取文件，使用Tuple存储每行，注意跳过表头
8   scala.io.Source.fromFile("a.txt").getLines
9 - 筛选参与计算的字段：dt、province、adds
10 - 格式化参与计算的字段：(dt,(adds,province))
11 - 按dt分组：
12   ("2020-5-12",List((5,"beijing"),(3,"shanghai")))
13   ("2020-5-13",List((3,"beijing"),(2,"shanghai")))
14 - 组内按adds降序排序后取top1
15
```

关键代码

```

1  val
   source=scala.io.Source.fromFile("/users/ly/temp/yq.csv"
   ).getLines
2  source.next //跳过表头
3  val yq=source.toList.filter(_.length>0).map(line=>{
4      val fields=line.split(",")
5      (fields(0),(fields(2),fields(1))) //筛选并格式化参与计算
   的字段: (dt,(adds,province))
6  })
7  //分组排序
8  val rs=yq.groupBy(row=>row._1).map(kvs=>
   (kvs._1,kvs._2.map(_._2).sortWith(_._1>_._1).head))
9
10 //输出
11 for((k,v) <- rs){
12     println(s"日期: $k Top 1: ${v._2} 新增: ${v._1}")
13 }
14
15 /*****
   *****/
16 //分组排序步骤分解
17 /**
18 // 按dt分组
19 scala> yq.groupBy(row=>row._1)
20 res122:
   scala.collection.immutable.Map[String,List[(String,
   (String, String))]] = Map(2020-5-13 -> List((2020-5-13,
   (3,beijing)), (2020-5-13,(2,shanghai))), 2020-5-12 ->
   List((2020-5-12,(5,beijing)), (2020-5-12,
   (3,shanghai))))
21
22 //分组后的结果

```

```
23 scala> yq.groupBy(row=>row._1).map(kvs=>
    (kvs._1,kvs._2))
24 res123:
scala.collection.immutable.Map[String,List[(String,
(String, String))]] = Map(2020-5-13 -> List((2020-5-13,
(3,beijing)), (2020-5-13,(2,shanghai))), 2020-5-12 ->
List((2020-5-12,(5,beijing)), (2020-5-12,
(3,shanghai))))
25
26 //筛选分组后的(adds,province)
27 scala> yq.groupBy(row=>row._1).map(kvs=>
    (kvs._1,kvs._2.map(_._2)))
28 res124:
scala.collection.immutable.Map[String,List[(String,
String)]] = Map(2020-5-13 -> List((3,beijing),
(2,shanghai)), 2020-5-12 -> List((5,beijing),
(3,shanghai)))
29
30 //按adds降序
31 scala> yq.groupBy(row=>row._1).map(kvs=>
    (kvs._1,kvs._2.map(_._2).sortWith(_._1>_._1)))
32 res125:
scala.collection.immutable.Map[String,List[(String,
String)]] = Map(2020-5-13 -> List((3,beijing),
(2,shanghai)), 2020-5-12 -> List((5,beijing),
(3,shanghai)))
33
34 //取top 1
35 scala> yq.groupBy(row=>row._1).map(kvs=>
    (kvs._1,kvs._2.map(_._2).sortWith(_._1>_._1).head))
36 res126: scala.collection.immutable.Map[String,(String,
String)] = Map(2020-5-13 -> (3,beijing), 2020-5-12 ->
(5,beijing))
```

```

37
38  **/
39  /*****
    *****/

```

5.3 实战案例三

`orders.csv`: 数据字典 (订单id、订单日期、顾客id、订单状态)

```

1  order_id  order_date  order_customer_id order_status
2  1  2013-07-25  00:00:00  11599  CLOSED
3  2  2013-07-25  00:00:00  256  PENDING_PAYMENT
4  4  2013-07-25  00:00:00  8827  CLOSED

```

`order_item.csv`: 数据字典 (订单明细id、订单id、商品id、商品数量、总价、单价)

```

1  order_item_id order_item_order_id order_item_product_id
   order_item_quantity order_item_subtotal
   order_item_product_price
2  1  1  957  1  299.98  299.98
3  2  2  1073  1  199.99  199.99
4  3  2  502  5  250  50
5  4  2  403  1  129.99  129.99
6  5  4  897  2  49.98  24.99
7  6  4  365  5  299.95  59.99
8  7  4  502  3  150  50
9  8  4  1014  4  199.92  49.98
10

```

请完成如下指标统计:

```
1 # 统计未支付的订单数量
2 # 每个订单的商品列表（包含哪些商品）
3 # 每个订单的总金额
4 # 找出消费最多的客户
```

分析

```
1 # 统计未支付的订单数量
2 - 对orders订单状态过滤
3 # 每个订单的商品列表（包含哪些商品）
4 - 使用Map[Int,List[Int]]存储结果, key为订单id, value存储商品
  列表
5 # 每个订单的总金额
6 - 使用Map[Int,List[Double]]存储每个订单明细的总价, key为订单
  id, value存储订单明细总价
7 - 基于上一步结果, 对每个key的value进行sum
8 # 找出消费最多的客户
9 - 基于每个订单的总金额, 按用户分组求sum后排序
10
```

关键代码

```
1 //装载数据
2 val
  orderSource=scala.io.Source.fromFile("/users/ly/temp/or
  ders.csv").getLines
3 orderSource.next
4 val
  orderItemSource=scala.io.Source.fromFile("/users/ly/tem
  p/order_item.csv").getLines
5 orderItemSource.next
6 //解析订单表
```

```
7 val orders=orderSource.map(line=>{
8     val fields=line.split("\t")
9     (fields(0).toInt,fields(1),fields(2),fields(3))
10 }).toList
11 //解析订单明细表
12 val orderItems=orderItemSource.map(line=>{
13     val fields=line.split("\t")
14     (fields(0).toInt,
15     fields(1).toInt,
16     fields(2).toInt,
17     fields(3).toInt,
18     fields(4).toDouble,
19     fields(5).toDouble)
20 }).toList
21
22 /// 统计未支付的订单数量
23 val rs1=orders.filter(_._4=="PENDING_PAYMENT").size
24 println(s"未支付订单数量: $rs1")
25
26 /// 每个订单的商品列表（包含哪些商品）
27 import scala.collection.mutable.Map
28 val ordersMap=Map[Int,List[Int]]()
29 //初始化ordersMap
30 orders.foreach(order=>{
31     //该订单商品明细 List[Int] _3为商品id
32     val
33     products=orderItems.filter(_._2==order._1).map(_._3)
34     ordersMap.put(order._1,products)
35 })
36 for((k,v)<-ordersMap){
37     println(s""订单: $k 商品明细: ${v.mkString(",")}""")
38 }
```

```

39  //# 每个订单的总金额
40  val ordersPrice=Map[Int,List[Double]]()
41  orders.foreach(order=>{
42      //_5为某一个商品总金额
43      val total=orderItems.filter(_._2==order._1).map(_._5)
44      ordersPrice.put(order._1,total)
45  })
46  for((k,v)<-ordersPrice){
47      println(s""订单: $k 总金额: ${v.sum}""")
48  }
49
50  //# 找出消费最多的客户
51  //找出每个订单的客户与消费额
52  val customerDetails=orders.map(order=>{
53      val total=orderItems.filter(_._2==order._1).map(_._5)
54      (order._1,order._3,total.sum)
55  })
56  val (vip,totalAmount)=customerDetails.groupBy(_._2)
57      .map(kvs=>
58          (kvs._1,kvs._2.map(_._3).sum)).toList //转成List排序
59          .sortWith(_._2>_._2)
60          .head
61  //输出
62  println(s"vip是$vip, 消费总金额: $totalAmount")

```

六 教学总结

6.1 教学重点

1. 掌握集合操作 (Array、Tuple、List、Map、Set)
2. 掌握高阶函数的概念及应用

6.2 教学难点

1. 可变集合与不可变集合的区别
2. 高阶函数的应用
3. 柯里化函数
4. 闭包

七 课后作业

1. 使用Map实现WordCount

分析

- ```
1 # Map的设计：使用单词为key，计数器为value
2 - 第一种方式：每到来一个单词，使用List作为value，List追加1，最后
 对List求和
3 Map("hello"->List(1,1,...), "world"-
 >List(1,1,...), ...)
4 - 第二种方式：每到来一个单词，使用Int作为value，值累加1
5 Map("hello"->1+1+..., "world"->1+1+..., ...)
```

#### 关键代码

2.现有学生成绩数据a.txt文件，请统计各学科平均分、最低分、最高分。

```
1 Id gender Math English Physics
2 301610 male 80 64 78
3 301611 female 65 87 58
4 301612 female 44 71 77
5 301613 female 66 71 91
6 301614 female 70 71 100
7 301615 male 72 77 72
8 301616 female 73 81 75
9 301617 female 69 77 75
10 301618 male 73 61 65
11 301619 male 74 69 68
12 301620 male 76 62 76
13 301621 male 73 69 91
14 301622 male 55 69 61
15 301623 male 50 58 75
16 301624 female 63 83 93
17 301625 male 72 54 100
18 301626 male 76 66 73
19 301627 male 82 87 79
20 301628 female 62 80 54
21 301629 male 89 77 72
```

分析

```
1 # 读取文件api
2 scala.io.Source.fromFile("a.txt").getLines
3 # 文件每行以空格分隔，首先使用split切分，然后可以使用5元组表示每个
 学生的基本信息
4 ("301610","male",80,64,78)
5 # 将每门学科所有成绩分别使用一个List存储，List提供了sum、size、
 min、max方法
6 val math=List(80,64,44...)
7 val englist=List(64,87,71,...)
8 val physics=List(78,58,77,...)
9
```

关键代码

```
1
```

## 八 解决方案

### 8.1 应用场景

Scala在大数据中的应用场景：

```
1 # Scala in Spark
2 Spark通过Scala语言集成API提供RDD抽象。Scala是一种静态类型的
 函数式面向对象Java语言虚拟机。我们选择使用Scala是因为它的简洁性
 （对于交互式使用特别有用）和效率（由于静态类型）的结合。但是，RDD
 抽象并不需要函数式语言；通过使用类来表示用户的函数，也可以提供其他
 语言的RDD。
3 一个Spark Scala API WordCount代码片断：
```

```

4 textFile.flatMap(line => line.split(" "))
5 .map(word => (word, 1))
6 .reduceByKey((a, b) => a + b)
7 .foreach(println)
8
9 # Scala in Kafka
10 Kafka起初是由LinkedIn公司采用Scala语言开发的一个多分区、多副本且基于ZooKeeper协调的分布式消息系统，现已被捐献给 Apache 基金会。目前 Kafka 已经定位为一个分布式流式处理平台，它以高吞吐、可持久化、可水平扩展、支持流数据处理等多种特性而被广泛使用。目前越来越多的开源分布式处理系统如 Cloudera、Storm、Spark、Flink等都支持与Kafka集成。
11
12
13 # Scala in Flink
14 一个Flink Scala API WordCount代码片断：
15 val counts = text.flatMap {
16 _.toLowerCase.split("\\W+") filter { _.nonEmpty } }
17 .map { (_, 1) }
18 .keyBy(_._1)
19 .window(TumblingProcessingTimeWindows.of(Time.seconds
20 (5)))
21 .sum(1)

```

## 8.2 核心面试题

```
1 # var、val和def三个关键字之间的区别?
2 # 元组如何遍历?
3 # 数组、列表是可变的吗?
4 # Nil是什么?
5 # 如何对Map排序?
6 # 什么是高阶函数?
7
```

## 九 授课流程（教师版）

---

早自习（50分钟）

前课分享（表达能力 技术能力）（10分钟）

第一节（80分钟）（授课+实操）

休息（20分钟）

第二节（80分钟）（练习答疑）

前课分享（10分钟）

第三节（80分钟）（授课+实操）

休息（20分钟）

第四节（80分钟）（授课+实操）（可选）

休息（10分钟）

第五节（50分钟）（练习答疑）

晚自习（120分钟）（自由练习）