scala集合操作

scala spark

Val、def、lazy定义变量的区别

Val：获取一次，并立即执行(严格执行)

Def：在声明的时候赋值

Lazy：惰性执行，也就是说赋值的时候不执行，等到需要的时候再执行

```
package top.xiesen.bd14
object DefTest {
  def sumInt(x: Int, y: Int) = {
    println("执行sumInt方法")
    x + y
  }
  def main(args: Array[String]): Unit = {
    // val类型的变量,在声明时就会把右边表达式的结果计算并赋值给val变量，一旦赋值，右边的表达式不再计算
```

```
    val v = sumInt(3,5)
    println(s"打印val对象v: $v")
    println(s"第二次打印val对象v: $v")

    // def类型的变量，在声明赋值时，右边的表达式是不会计算结果的，在变量类型每次被调用的时候，等号右边的表达式都会被计算一次
    def d = sumInt(3,5)
    println("变量d已经赋值了")
    println(s"打印def对象d: $d")
    println(s"第二次打印def对象d: $d")

    // lazy定义的变量，在声明赋值时，等号右边的表达式不会马上计算结果
    // lazy在对象第一次被调用时，等号右边的表达式会被计算一次，并且被赋值一次
    // 后续的对lazy对象的再次调用，右边的表达式不会再进行计算
    lazy val l = sumInt(3,5)
    println("变量l已经被定义赋值过了")
    println(s"打印变量lazy的值l: $l")
    println(s"第二次打印变量lazy的值l: $l")
  }
}
```

List

reduce

`val reducerList = list6.reduce((x1, x2) => x1 + x2)` Reduce是对list6中的所有元素进行迭代计算的函数，reduce计算结束之后相当于把集合中的每一个元素按照迭代函数，迭代计算聚合起来

```
val list6 = List(1, 2, 3, 4, 5)
val reducerList = list6.reduce((x1, x2) => x1 + x2)
val reduceListMax = list6.reduce((x1,x2)=> if(x1 > x2) x1 else x2)
println(s"list6的总和 = $reducerList" )
println(s"list6的最大值 = $reduceListMax" )
```

fold

```
val foldResult = list6.fold(0)((x1,x2) => x1 + x2)
println(s"list6中元素的总和是: $foldResult")
```

foldLeft

```
// 把list6中的元素聚合成一个字符串，字符串中包含有每一个元素
val strFoldResult = list6.foldLeft("")((c,x) => s"${if(c == "") "" else ","}$x")
println(s"fold实现mkString: $list6")
```

aggregate

> 先进行单个元素之间的迭代，再进行分区之间的聚合

```
val strAggrateResult = list6.aggregate("")(
  (c, x) => s"$c${if (c == "") "" else ","}$x"
  , (c1, c2) => s"$c1,$c2"
)
println(s"aggregate实现mkString: $strAggrateResult")
```

```
val sumAggrateResult = list6.aggregate(0)(
  (x1,x2) => x1 + x2
  ,(c1,c2) => c1 + c2
)
println("aggregate实现list6中元素的总和是: " + sumAggrateResult)
```

常用函数

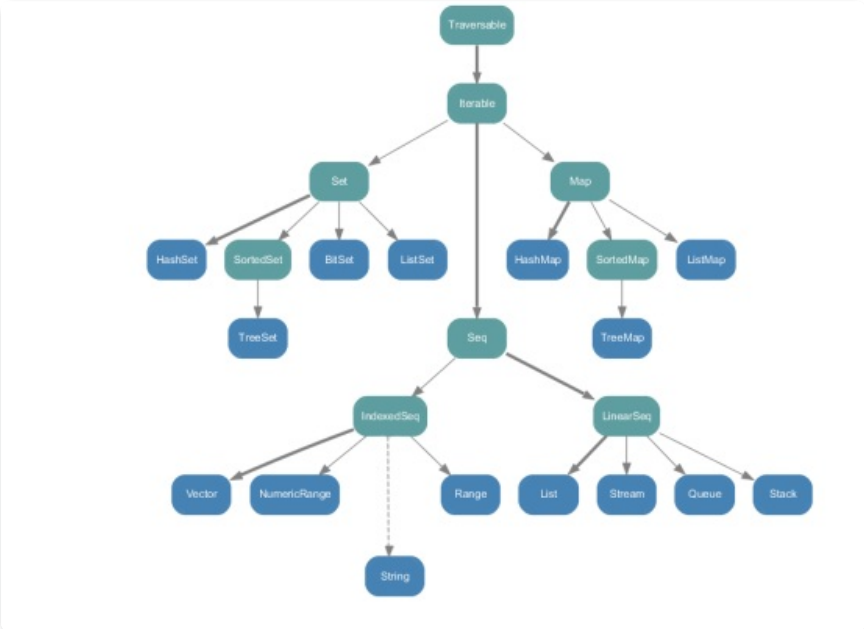
```
// 把a b c d 字符构建一个list
val list7 = "a" :: "b" :: "c" :: "d" :: Nil
println(list7)
// 获取list的头部
println(list7.head)
// 获取除第一个元素之外的元素列表
println(list7.tail)
// 获取list的最后一个元素
println(list7.last)
// 获取list除了最后一个元素，其他的元素列表
println(list7.init)

// 将list6按照奇数偶数分成两组
val groupResult = list6.groupBy(x => if(x % 2 == 1) "奇数" else "偶数")
println(s"将list6按照奇数偶数分成两组 : $groupResult")

// 拉链操作,当两个list的元素数量不一样时，在结果集中直接被丢弃
val list8 = List(1,2,3,4)
val list9 = List("a","b","c","d")
println(list8 zip list9)
```

Set

> Set是不重复的集合，在集合中是无序的。Set的操作基本上是和List中的函数是相同的



Set的声明和字面量

```
// set的声明和字面量
val set1 = Set(1, 2, 3, 4, 5, 4, 5)
```

```
println(set1)
// set是无序、不重复，因此不能使用index进行取值
println(set1(0))
```

Set集合遍历

```
// set遍历
for (i <- set1) println(i)
println("----------------------")
set1.foreach(x => println(x))
```

可变set集合

```
val mSet = scala.collection.mutable.Set(3,2,4)
mSet.add(456)
println(mSet)
mSet.add(3)
println(mSet)
mSet += 1
println(mSet)
mSet.remove(3)
println(mSet)
```

Map

Map声明，字面量

```
// map声明，字面量，取值
val map1 = Map(1 -> "a", 2 -> "b", 3 -> "c")
println(map1)
val map2 = Map((1,"a"),(2,"b"),(3,"c"))
println(map2)
```

Map取值

```
// map取值
println(map1(1))
println(map1(2))

val map3 = Map("a" -> 1,"b" -> 2, "c" -> 3)
println(map3("a"))
```

Map遍历

```
// 遍历
// 方式一
map3.foreach(
  x => println(s"key: ${x._1},value:${x._2}")
)
println("-----------------")
// 方式二
for(x <- map3){
```

```
  println(s"key: ${x._1}, value: ${x._2}")
}
println("------------")
// 方式三
for((k,v) <- map3){
  println(s"key: ${k}, value: ${v}")
}
println("-------------")
// 方式四
for(ks <- map3.keySet){
  println(s"key: ${ks}, value: ${map3(ks)}")
}
```

通过get获取key的value值

```
// get方法，获取指定key的value值
val map4 = Map("zhang" -> 80, "li" -> 85, "liu" -> 90)
println(map4("zhang"))
println(map4.get("zhang"))
// 小括号获取key的value的值，如果key不存在，程序会抛出 java.util.NoSuchElementException 异常
// 建议使用get获取key的value的值
//    println(map4("aaa"))
println(map4.get("wang"))
```

Map ++

```
// map ++
println(map3 ++ map4)
println(map3 ++ Map("c" -> 100))
```

Map中判断是否包含key

```
// 判断map4中是否包含key: "zhang"
println(map4.contains("zhang"))
```

Map count

```
// 计算出value代表key长度(value的值=key的长度)的kv对的数据量
val map5 = Map("apple" -> 5, "pear" -> 4, "peach" -> 5, "tomato" -> 6, "banana" -> 7)
val count1 = map5.count(x => x._1.length == x._2)
println(s"value的值=key的长度 的数量:  $count1")
// 计算出key长度是5的kv对数量
val count2 = map5.count(x => x._1.length == 5)
println(s"key长度是5的kv对数量: $count2")
```

Map drop

```
// drop
println(map5)
println(map5.drop(2))
println(map5.dropRight(2))
```

Map filter

```
// 过滤掉key长度等于5的kv对
val filter1 = map5.filterKeys(x => x.length != 5)
println("使用filterKeys过滤掉key长度等于5的kv对,结果为 : " + filter1)

val filter2 = map5.filter(x => x._1.length != 5)
println("使用filter过滤掉key长度等于5的kv对,结果为 : " + filter2)
```

flatMap 展平

```
//flatMap 展平
val map6 = Map("a" -> List(1, 2, 3), "b" -> List(4, 5, 6))
println(map6)
val flatMap1 = map6.flatMap(x => x._2)
println(flatMap1)

val map7 = Map("zhang" -> List("zhangfei" -> "shu", "zhangliao" -> "wei"), "liu" -> Li
st("liubei" -> "shu", "liuzhang" -> "zhong"))
println(map7)
val flatMap2 = map7.flatMap(x => x._2)
println(flatMap2)
```

Map groupby

```
val map8 = Map("zhang fei" -> "shu", "zhang liao" -> "wei", "liu bei" -> "shu", "liu z
hang" -> "zhong")
// 按照国家分成N组
val result = map8.groupBy(x => x._2)
println("按照国家分成N组,结果为: "+result)
// 按照姓氏分组
val result1 = map8.groupBy(x => x._1.split("\\s")(0))
println("按照姓氏分组,结果是: " + result1)
```

Head tail init last

```
// head tail init last 在递归操作时使用
val  head1 = map8.head
println(head1)
val tail1 = map8.tail
println(tail1)
val init1 = map8.init
println(init1)
val last1 = map8.last
println(last1)
```

Map方法，对kv对进行映射转换

```
// Map方法，对kv对进行映射转换
val map9 = Map("小张" -> 3000, "小李" -> 4500, "小王" -> 5000, "小刘" -> 4000)
// 每个人工资加500
val upSalary = map9.map(x => (x._1, x._2 + 500))
println("map实现每个人工资加500，结果为: " + upSalary)
val upSalary1 = map9.mapValues(x => x + 500)
println("mapValues实现每个人工资加500，结果为: " + upSalary1)
```

```
// 工资大于4000位高收入，否则为低收入，在姓名钱打上高收入或低收入的标签，如[低收入]小张
val tagResult = map9.map(x => if (x._2 > 4000) ("[高收入]" ++ x._1, x._2) else ("[低收入]" ++ x._1, x._2))
println(tagResult)
```

key 最大最小值

```
val map1 = Map(1 -> "a", 2 -> "b", 3 -> "c")
// key 最大最小值
println(s"map1的最大key值: ${map1.max}, 最小key值: ${map1.min}")
val map9 = Map("小张" -> 3000, "小李" -> 4500, "小王" -> 5000, "小刘" -> 4000)
// maxby map9根据工资来返回最大值
val maxSalary1 = map9.maxBy(x => x._2)
println(s"最高工资: $maxSalary1")
val maxSalary2 = map9.map(x => (x._2,x._1)).max
println(s"最高工资: $maxSalary2")
```

Reduce

```
val map9 = Map("小张" -> 3000, "小李" -> 4500, "小王" -> 5000, "小刘" -> 4000)
// 计算map9月支出,把map9中的所有value加起来
// reduce需要保持输入输出一致
val salaryMonthly = map9.reduce((x1,x2) => {
  ("月支出",x1._2 + x2._2)
})
println(salaryMonthly)
```

Fold

```
val map9 = Map("小张" -> 3000, "小李" -> 4500, "小王" -> 5000, "小刘" -> 4000)
val salaryMonthlyByFold = map9.fold("月支出", 0)((c, x) => (c._1, c._2 + x._2))
println(s"Fold计算月支出,计算值: $salaryMonthlyByFold")
```

foldLeft,foldRight,aggregate

```
val map9 = Map("小张" -> 3000, "小李" -> 4500, "小王" -> 5000, "小刘" -> 4000)
// reduce和fold唯一的不同就是fold需要一个初值，他们的不足是都会受输入类型的限制，迭代计算的输入类型和输出类型一致
// foldLeft,foldRight,aggragate没有这个限制
val salaryMonthlyByFoldLeft = map9.foldLeft(0)((c, x) => c + x._2)
println(s"FoldLeft计算月支出,计算值: $salaryMonthlyByFoldLeft")

val salaryMonthlyByAggregate = map9.aggregate(0)(
  (c, x) => c + x._2
  , (c1, c2) => c1 + c2
)
println(s"aggregate计算月支出,计算值: $salaryMonthlyByAggregate")
```

可变map的声明

```
// 可变map的声明
val mMap1 = scala.collection.mutable.Map(1 -> "a", 2 -> "b", 3 -> "c")
println(mMap1)
```

```
println(mMap1(1))
println(mMap1.get(1))
mMap1.put(4,"d")
println(mMap1)
mMap1.remove(3)
println(mMap1)
mMap1 += (5 -> "e")
println(mMap1)
mMap1.update(1,"修改后的结果")
println(mMap1)
```

元组

元组的声明

```
// 元组的声明，字面量
val tuple1 = (1,2,"a",3.0,1231,true)
val person = (1,"小张",23,"15156987410","备注信息")
val pairTuple = ("a",1) // 等同于("a" -> 1)
println(tuple1)
println(person)
println(pairTuple)
```

元组的取值

```
// 元组位置是从1开始的
println(person._2)

// 元组也是不可变的，定义之后不能发生改变
//    person._2 = ""
val (one,two,three) = (1,"a",true)
println(one)
println(two)
println(three)
```

元组封装返回值

```
println(tupleTest01("Hello tuple"))
// tuple封装返回值
def tupleTest01(a: String) = {
  val value1 = s"return value1 $a"
  val value2 = s"return value1 $a"
  val value3 = s"return value1 $a"
  (value1,value2,value3)
}
```

OPtion、None、Some类型

> Option类是用来封装其他类型的对象，一般应用在方法的返回值上，以避免方法返回空值带来不必要的麻烦。Option是None和some的父类

None

> None没有任何返回值

```
object OptionTest {
  def main(args: Array[String]): Unit = {
    val value1 = getValue(1)
    val value2 = getValue(-1)
    println(s"$value1 , $value2")
    println(value1.get)

    val value3 = getValue(5)
    // getOrElse(0),参数指的是默认值
    println(value3.getOrElse(0))
  }
  def getValue(a: Int): Option[Int] = {
    if (a > 0) Some(a) else None
  }
}
```

some

> Some封装了返回值

```
// some可以封装任意类型，一般用在函数的返回值上
val some1 = Some("abc")
val some2 = Some(true)
println(some1)
println(some2)
```