Day26 & Day27_Scala & 函数 & 数组 & 集合

大数据-张军锋 Day26 Day27 Scala 函数 数组 集合

- Day26 & Day27_Scala & 函数 & 数组 & 集合
  - 函数
    - 函数的定义
    - 函数的类型
    - 函数字面量（匿名函数）
      - def与val的区别
  - def & val & lazy定义变量的区别
    - val
    - def
    - lazy
  - Array
    - 重点
    - 定长数组
    - 变长数组：数组缓冲，ArrayBuffer
      - Array和ArrayBuffer的转换
    - 遍历Array和ArrayBuffer
    - 数组转换
    - 常用算法
    - 多维数组
  - List
    - 常用操作符
    - 常用变换操作
      - map
      - flatMap & flatten
      - reduce
      - reduceLeft & reduceRight
      - fold & foldLeft & foldRight & aggregate
      - sortBy & sortWith & sorted
      - filter & filterNot
      - count & endsWith
      - diff & union & intersect
      - distinct
      - head & last & tail & init
      - groupBy & grouped
      - scan
      - scanLeft & scanRight
      - take & takeRight & takeWhile
      - drop & dropRight & dropWhile
      - span & splitAt & partition
      - padTo
      - combinations & permutations

函数

函数的定义

函数定义和对象的定义一样，编译器可以通过返回值
因此，**绝大多数的函数定义，都不写返回值类型**
只有一种情况例外：**递归函数**


```
def functionName(x: Int, y: Int): Int = {
  x + y
}
```

在scala中因为函数是对象，因此它的定义方式有很多
下面的这个函数定义是一个过程函数的定义，**过程函数是没有返回值的，即返回值是Unit**
只要函数这么定义它的返回值就是Unit，就算在函数体内return也不会返回结果

```
def functionName(x: Int, y: Int) {
  x + y
}
```

> 如果定义函数时把函数的类型指定为Unit，那么不管该函数的语句块最后一句结果是什么，该函数的返回值始终是Unit( )

函数的类型

因为scala中函数是一等公民，因此它和对象一样也有自己的类型
因为函数中涉及的类型包括参数的类型，返回值类型，因此**函数的类型就用参数类型和返回值类型来共同定义**

```
def functionName(x: Int, y: Int) = {
  x + y
}
```

如上函数有两个参数都是Int,有一个返回值也是Int,那么在scala中的函数类型描述是:
`(Int,Int)=> Int`
其中 **=> 符号分割参数定义（输入）类型和返回值（输出）类型**

函数字面量（匿名函数）

函数的字面量也是用=>来进行定义，它的左边是参数（输入），右边是返回值（输出）

```
val plusIntVal: (Int, Int) => Int = (x, y) => x + y
val plusIntVal1 = (x: Int, y: Int) => x + y
```

val定义的函数名称，后面不加小括号代表的是对函数对象的引用，后面添加小括号代表的是对函数对象的调用

def与val的区别

- def定义的函数不可以当做对象来被传递
- val定义的函数可以被当做函数来被传递

def & val & lazy定义变量的区别

val

- val类型的变量,在声明时就会把右边的表达式的结果计算并赋值给val变量
- 一旦赋值，右边的表达式就不再计算

```
def sumInt(x: Int, y: Int) = {
  println("执行sumInt方法")
  x + y
}

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

输出结果

```
执行sumInt方法
赋值成功
打印val对象v:8
第二次打印val对象v:8
```

def

- def类型的变量，在声明赋值时，右边的表达式是不会马上计算结果的
- 在def类型变量每一次被调用的时候，等号右边的表达式都会被重新计算一次

```
def d = sumInt(3, 5)
println("赋值成功")
println(s"打印def对象d:$d")
println(s"第二次打印def对象d:$d")
```

输出结果

```
赋值成功
执行sumInt方法
打印def对象d:8
执行sumInt方法
第二次打印def对象d:8
```

lazy

- lazy定义的变量，在声明变量时，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的时候，等号右边的表达式会被调用一次，并赋值给lazy对象
- 后续的对lazy对象的再次调用，右边的表达式将不再会被重新计算

```
lazy val l = sumInt(3, 5)
println("赋值成功")
println(s"打印lazy对象l:$l")
println(s"第二次打印lazy对象l:$l")
```

输出结果

```
赋值成功
执行sumInt方法
打印lazy对象l:8
第二次打印lazy对象l:8
```

Array

Array是可变（元素可变）的，它和java的数字T[ ]是对应的
数组是定长的，定义的时候必须指定长度

重点

- 长度固定则使用Array，若长度有可能变化则使用ArrayBuffer
- 提供初始值时不要使用new
- 用( )来访问元素，而不是[ ]来访问元素
- 用`for（elem <- arr）`来遍历元素
- 用`for（elem <- arr if …）… yield …`来将原数组转为新数组
- Scala数组和Java数组可以互操作；用ArrayBuffer，使用scala.collection.JavaConversions中的转换函数

定长数组

```
val array = Array(1, 2, 3, 4)
array(0) = 666
val array = new Array[Int](10)
```

- **复杂对象类型在数组定义时被初始化为null，数值型被初始化为0**
- **数组定义之后长度不能被改变，但数组内容是可以改变的**

> 在JVM中，Scala的Array以Java数组方式实现。示例中的数组在JVM中的类型为java.lang.String[].Int, Double 或其他与java中基本类型对应的数组都是基本类型数组。如Array(2,3,5,7,11)在JVM中就是一个int[]

变长数组：数组缓冲，ArrayBuffer

对于长度需要变化的数组，Java有ArrayList，C++有Vector，Scala中有ArrayBuffer
要使用ArrayBuffer，先要引入scala.collection.mutable.ArrayBuffer

```
//ArrayBuffer的声明和字面量
val ab1 = ArrayBuffer(1, 2, 3, 4)
val ab2 = new ArrayBuffer[Int](2)
val ab3 = new ArrayBuffer[String]()

//获取和修改内容
println(ab1(1))
ab1(0) = 23

//在尾端添加多个元素，以括号包起来
ab2.+=(123, 456)

//可以用 ++= 操作符追加任何集合
ab2.++= Array(8, 13, 21)

//在首端添加元素生成新的集合
var ab4 = ab2.+:(789)

//修改指定位置的元素值
ab2.update(1, 666)

//移除最后3个元素
ab1.trimEnd(3)

//drop不是在原有的数组上操作，而是生成新的数组
println(ab2.drop(1))
println(ab2.dropRight(1))
```

在ArrayBuffer的尾端添加或移除元素是一个高效的操作（”amortized constant time“，固定时间）。
在任意位置插入或移除元素时，效率较低——在那位置之后的元素都要被平移。

```
//在索引为2的地方插入元素，后面可以跟多个元素
ab2.insert(2, 111, 222)

//移除指定位置的元素，第二个参数是要移除多少元素（1个可以不写）
ab2.remove(2)
ab2.remove(2, 2)
```

Array和ArrayBuffer的转换

```
array.toArray

array.toBuffer
```

遍历Array和ArrayBuffer

在Java和C++中，数组和数组列表/向量有一些语法上的不同，Scala则更加统一。大多数时候，可以用相同的代码处理这两种数据结构。

```
for(i <- 0 until array.length)
    println(i + ":" +a(i))
until返回所有小于但不包括上限的数字
to返回所有小于等于上限的数字

//设置步长，即每次循环i的值自增多少
0 until (array.length, 2)
//Range(0,2,4,...)

(0 until array.length).reverse
//Range(...,2,1,0)

for(elem <- array)
    println(elem)
```

数组转换

从一个数组（或数组转换）出发，以某种方式对它进行转换，这些转换动作**不会修改原始数组，而是产生一个全新的数组。**
缓冲数据转换后产生的仍然是缓冲数组

```
val a = Array(2,3,5,7,11)
val result = for(elem <- a) yield 2*elem
//result是Array（4,6,10,14,22)

for(elem <- a if elem % 2 == 0) yield 2* elem
//对每个偶数元素翻倍，并丢弃奇数元素
```

```
//给定一个整数的数组缓冲，移除第一个负数之外的所有负数。
val ab = ArrayBuffer(1, 2, -3, -4, -5, 6, 7, 8)
var bool = true
val newAb = for(i <- 0 until ab.length if bool||ab(i)>=0)yield{
  if(ab(i)<0){
    bool=false
    //遇到第一个负数时置first = false,以后再遇到负数，根据 first||a(i)>=0
就直接跳过了
  }
  ab(i)
}
println(newAb)
//Vector(1, 2, -3, 6, 7, 8)
```

常用算法

sum方法 & max方法

要使用sum方法，元素类型必须是数值类型

```
val ab = ArrayBuffer(1, 2, -3, -4, -5, 6, 7, 8)
println(ab.sum)          //12
println(ab.max)          //8
ArrayBuffer("Mary","had","a","little","lamb").max    //"little"
```

sorted方法

sorted方法将Array或ArrayBuffer排序并返回经过排序的Array或ArrayBuffer，这个过程不会修改原始版本：

```
val b = ArrayBuffer(1,7,2,9)
val bSorted = b.sorted //b没有改变，bSorted是ArrayBuffer(1,2,7,9)
```

sortWith方法

```
val bDescending = b.sortWith(_>_)       //ArrayBuffeer(9,7,2,1)
```

可以直接对一个数组排序，但不能对数组缓冲排序：

```
val a = Array(1,7,2,9)
scala.util.Sorting,quickSort(a)
//a : Array(1,2,7,9)
```

mkString方法

```
a.mkString(" and ")
//"1 and 2 and 7 and 9"
a.mkString("<", "," , ">")
```

与toString相比

```
a.toString
//"[I@73e5"
//这里调用的是来自Java的没有意义的toString方法
b.toString
//"ArrayBuffer(1,7,2,9)"
//toString方法报告了类型，便于调试
```

filter

```
//过滤掉带有a的水果
val fruita = Array("apple", "tomato", "peach", "watermallon", "ber
y")
val noAFruita = fruita.filter(x => !(x.contains("a")))    //bery

//搜索是否存在长度等于5的水果，返回布尔
val length5Fruita = fruita.exists(x => x.length == 5)     //true
```

多维数组

通过数组的数组实现多维数组的定义：

```
//定义2行3列数组
var multiDimArr=Array(Array(1,2,3),Array(2,3,4))
multiDimArr: Array[Array[Int]] = Array(Array(1, 2, 3), Array(2, 3,
4))

//获取第一行第三列元素
multiDimArr(0)(2)
res99: Int = 3

//多维数组的遍历
for(i <- multiDimArr)
    println( i.mkString(","))//1,2,32,3,4
```

List

List的声明、遍历和上文Array一样，在此就不多做解释了

常用操作符

`++` 从列表的尾部添加另外一个列表
`++:` 在列表的头部添加一个列表
`+:` 在列表的头部添加一个元素
`:+` 在列表的尾部添加一个元素
`::` 在列表的头部添加一个元素
`:::` 在列表的头部添加另外一个列表
`:\` B 与foldRight等价

```
val left = List(1,2,3)
val right = List(4,5,6)

//以下操作等价
left ++ right   // List(1,2,3,4,5,6)
left ++: right  // List(1,2,3,4,5,6)
right.++:(left)    // List(1,2,3,4,5,6)
right.:::(left)  // List(1,2,3,4,5,6)

//以下操作等价
0 +: left    //List(0,1,2,3)
left.+:(0)   //List(0,1,2,3)

//以下操作等价
left :+ 4    //List(1,2,3,4)
left.:+(4)   //List(1,2,3,4)

//以下操作等价
0 :: left       //List(0,1,2,3)
left.::(0)      //List(0,1,2,3)
```

> 看到这里大家应该跟我一样有一点晕吧，怎么这么多奇怪的操作符，这里给大家一个提示，**任何以冒号结果的操作符，都是右绑定的**，即 0 :: List(1,2,3) = List(1,2,3).::(0) = List(0,1,2,3) 从这里可以看出**操作::其实是右边List的操作符，而非左边Int类型的操作符**

常用变换操作

map

```
map[B](f: (A) ⇒ B): List[B]
```

定义一个变换,把该变换应用到列表的每个元素中,原列表不变，返回一个新的列表数据

- 平方变换

```
val nums = List(1,2,3)
val square = (x: Int) => x*x
val squareNums1 = nums.map(num => num*num)     //List(1,4,9)
val squareNums2 = nums.map(math.pow(_,2))      //List(1,4,9)
val squareNums3 = nums.map(square)             //List(1,4,9)
```

- 保存文本数据中的某几列

```
val text = List("Homeway,25,Male","XSDYM,23,Female")
val usersList = text.map(_.split(",")(0))
val usersWithAgeList = text.map(line => {
    val fields = line.split(",")
    val user = fields(0)
    val age = fields(1).toInt
    (user,age)
})
```

flatMap & flatten

```
对列表的列表进行平坦化操作
flatten: flatten[B]: List[B]
```

```
map之后对结果进行flatten
flatMap: flatMap[B](f: (A) ⇒ GenTraversableOnce[B]): List[B]
```

定义一个变换f, 把f应用列表的每个元素中，每个f返回一个列表，最终把所有列表连结起来。

```
val text = List("A,B,C","D,E,F")
val textMapped = text.map(_.split(",").toList)
// List(List("A","B","C"),List("D","E","F"))

val textFlattened = textMapped.flatten
// List("A","B","C","D","E","F")

val textFlatMapped = text.flatMap(_.split(",").toList)
// List("A","B","C","D","E","F")
```

reduce

```
reduce[A1 >: A](op: (A1, A1) ⇒ A1): A1
```

定义一个变换f, f把两个列表的元素合成一个，遍历列表，最终把列表合并成单一元素

- 列表求和

```
val nums = List(1,2,3)
val sum1 = nums.reduce((a,b) => a+b)    //6
val sum2 = nums.reduce(_+_)             //6
val sum3 = nums.sum                     //6
```

reduceLeft & reduceRight

```
reduceLeft: reduceLeft[B >: A](f: (B, A) ⇒ B): B

reduceRight: reduceRight[B >: A](op: (A, B) ⇒ B): B
```

reduceLeft从列表的左边往右边应用reduce函数，reduceRight从列表的右边往左边应用reduce函数

```
val nums = List(2.0,2.0,3.0)
val resultLeftReduce = nums.reduceLeft(math.pow)
// = pow( pow(2.0,2.0) , 3.0) = 64.0

val resultRightReduce = nums.reduceRight(math.pow)
// = pow(2.0, pow(2.0,3.0)) = 256.0
```

fold & foldLeft & foldRight & aggregate

fold & aggregate

```
fold[A1 >: A](z: A1)(op: (A1, A1) ⇒ A1): A1
```

带有初始值的reduce,从一个初始值开始，从左向右将两个元素合并成一个，最终把列表合并成单一元素。

```
val list = List(1, 2, 3, 4, 5, 6, 7, 8, 9)
val sumList = list.fold(0)((a, b) => a + b)//45

val sumAggrateResult = list.aggregate(0)(
  (c, x) => c + x
  , (c1, c2) => c1 + c2
)//45
```

foldLeft & aggregate

```
foldLeft[B](z: B)(f: (B, A) ⇒ B): B

aggregate[B](z: ⇒ B)(seqop: (B, A) ⇒ B, combop: (B, B) ⇒ B): B
```

带有初始值的reduceLeft

```
val list = List(1, 2, 3, 4, 5, 6, 7, 8, 9)
val strFoldResult = list.foldLeft("")((c, x) => s"$c${if (c == "")
"" else ","}$x")
//1,2,3,4,5,6,7,8,9

val strAggrateResult = list.aggregate("")(
  (c, x) => s"$c${if (c == "") "" else ","}$x"
  , (c1, c2) => s"$c1,$c2"
)
//1,2,3,4,5,6,7,8,9
```

foldRight

```
foldRight[B](z: B)(op: (A, B) ⇒ B): B
```

带有初始值的reduceRight

```
val nums = List(2,3,4)
val sum = nums.fold(1)(_+_)  // = 1+2+3+4 = 9

val nums = List(2.0,3.0)
val result1 = nums.foldLeft(4.0)(math.pow)
// = pow(pow(4.0,2.0),3.0) = 4096
val result2 = nums.foldRight(1.0)(math.pow)
// = pow(1.0,pow(2.0,3.0)) = 8.0
```

sortBy & sortWith & sorted

```
sortBy: sortBy[B](f: (A) ⇒ B)(implicit ord: math.Ordering[B]): Lis
t[A]
```

按照应用函数f之后产生的元素进行排序

sorted

```
按照元素自身进行排序
sorted[B >: A](implicit ord: math.Ordering[B]): List[A]
```

sortWith

```
使用自定义的比较函数进行排序
sortWith(lt: (A, A) ⇒ Boolean): List[A]
```

```
val nums = List(1,3,2,4)
val sorted = nums.sorted  //List(1,2,3,4)

val users = List(("HomeWay",25),("XSDYM",23))
val sortedByAge = users.sortBy{case(user,age) => age}
//List(("XSDYM",23),("HomeWay",25))
val sortedWith = users.sortWith{case(user1,user2) => user1._2 < use
r2._2} //List(("XSDYM",23),("HomeWay",25))
```

filter & filterNot

filter

```
filter(p: (A) ⇒ Boolean): List[A]
```

filterNot

```
filterNot(p: (A) ⇒ Boolean): List[A]
```

filter 保留列表中符合条件p的列表元素
filterNot，保留列表中不符合条件p的列表元素

```
val nums = List(1,2,3,4)
val odd = nums.filter( _ % 2 != 0) // List(1,3)
val even = nums.filterNot( _ % 2 != 0) // List(2,4)
```

count & endsWith

```
count(p: (A) ⇒ Boolean): Int

endsWith[B](that: GenSeq[B]): Boolean
```

count计算列表中所有满足条件p的元素的个数，等价于 filter(p).length
endsWith测试这个序列是否以给定的序列结束

```
val list = List(1, 2, 3, 4, 5, 6, 7, 8, 9)
val bt5Count = list4.count(x => x > 5)        //4
val isEndWithNum = list.endsWith(List(8, 9))    //true
```

diff & union & intersect

diff

```
保存列表中那些不在另外一个列表中的元素，即从集合中减去与另外一个集合的交集
diff(that: collection.Seq[A]): List[A]
```

union

```
与另外一个列表进行连结
union(that: collection.Seq[A]): List[A]
```

intersect

```
与另外一个集合的交集
intersect(that: collection.Seq[A]): List[A]
```

```
val nums1 = List(1,2,3)
val nums2 = List(2,3,4)
val diff1 = nums1 diff nums2        // List(1)
val diff2 = nums2.diff(num1)        // List(4)
val union1 = nums1 union nums2      // List(1,2,3,2,3,4)
val union2 = nums2 ++ nums1         // List(2,3,4,1,2,3)
val intersection = nums1 intersect nums2  //List(2,3)
```

distinct

```
保留列表中非重复的元素，相同的元素只会被保留一次
distinct: List[A]
```

```
val list = List("A","B","C","A","B")
val distincted = list.distinct      // List("A","B","C")
```

head & last & tail & init

```
//把  a b c d 字母构建一个List
val list = "a" :: "b" :: "c" :: "d" :: Nil
println(list)   //List(a, b, c, d)
//获取list的第一个元素
println(list.head)   //a
//获取list的最后一个元素
println(list.last)   //d
//获取list的第一个元素之外的其他的元素列表
println(list.tail)   //List(b, c, d)
//获取list除最后一个元素之外其他的元素列表
println(list.init)   //List(a, b, c)
```

groupBy & grouped

groupBy

```
将列表进行分组，分组的依据是应用f在元素上后产生的新元素
groupBy[K](f: (A) ⇒ K): Map[K, List[A]]
```

grouped

```
按列表按照固定的大小进行分组
grouped(size: Int): Iterator[List[A]]
```

```
val data = List(("HomeWay","Male"),("XSDYM","Femail"),("Mr.Wang","M
ale"))
val group1 = data.groupBy(_._2)
// = Map("Male" -> List(("HomeWay","Male"),("Mr.Wang","Male")),"Fem
ale" -> List(("XSDYM","Femail")))
val group2 = data.groupBy{case (name,sex) => sex}
// = Map("Male" -> List(("HomeWay","Male"),("Mr.Wang","Male")),"Fem
ale" -> List(("XSDYM","Femail")))
val fixSizeGroup = data.grouped(2).toList
// = Map("Male" -> List(("HomeWay","Male"),("XSDYM","Femail")),"Fem
ale" -> List(("Mr.Wang","Male")))
```

scan

```
scan[B >: A, That](z: B)(op: (B, B) ⇒ B)(implicit cbf: CanBuildFro
m[List[A], B, That]): That
```

由一个初始值开始，从左向右，进行积累的op操作，这个比较难解释，具体的看例子吧。

```
val nums = List(1,2,3)
val result = nums.scan(10)(_+_)
// List(10,10+1,10+1+2,10+1+2+3) = List(10,11,12,13)
```

scanLeft & scanRight

```
scanLeft: scanLeft[B, That](z: B)(op: (B, A) ⇒ B)(implicit bf: CanB
uildFrom[List[A], B, That]): That

scanRight: scanRight[B, That](z: B)(op: (A, B) ⇒ B)(implicit bf: Ca
nBuildFrom[List[A], B, That]): That
```

scanLeft: 从左向右进行scan函数的操作
scanRight：从右向左进行scan函数的操作

```
val nums = List(1.0,2.0,3.0)
val result = nums.scanLeft(2.0)(math.pow)
// List(2.0,pow(2.0,1.0), pow(pow(2.0,1.0),2.0),pow(pow(po
w(2.0,1.0),2.0),3.0) = List(2.0,2.0,4.0,64.0)
val result = nums.scanRight(2.0)(math.pow)
// List(2.0,pow(3.0,2.0), pow(2.0,pow(3.0,2.0)), pow(1.0,pow(2.0,po
w(3.0,2.0))) = List(1.0,512.0,9.0,2.0)
```

take & takeRight & takeWhile

```
提取列表的前n个元素
takeRight(n: Int): List[A]

提取列表的最后n个元素
takeRight(n: Int): List[A]

从左向右提取列表的元素，直到条件p不成立
takeWhile(p: (A) ⇒ Boolean): List[A]
```

```
val nums = List(1,1,1,1,4,4,4,4)
val left = nums.take(4)   // List(1,1,1,1)
val right = nums.takeRight(4) // List(4,4,4,4)
val headNums = nums.takeWhile( _ == nums.head)  // List(1,1,1,1)
```

drop & dropRight & dropWhile

```
丢弃前n个元素，返回剩下的元素
drop(n: Int): List[A]

丢弃最后n个元素，返回剩下的元素
dropRight(n: Int): List[A]

从左向右丢弃元素，直到条件p不成立
dropWhile(p: (A) ⇒ Boolean): List[A]
```

```
val nums = List(1,1,1,1,4,4,4,4)
val left = nums.drop(4)    // List(4,4,4,4)
val right = nums.dropRight(4)  // List(1,1,1,1)
val tailNums = nums.dropWhile( _ == nums.head)  // List(4,4,4,4)
```

span & splitAt & partition

```
从左向右应用条件p进行判断，直到条件p不成立，此时将列表分为两个列表
span(p: (A) ⇒ Boolean): (List[A], List[A])

将列表分为前n个，与，剩下的部分
splitAt(n: Int): (List[A], List[A])

将列表分为两部分，第一部分为满足条件p的元素，第二部分为不满足条件p的元素
partition(p: (A) ⇒ Boolean): (List[A], List[A])
```

```
val nums = List(1,1,1,2,3,2,1)
val (prefix,suffix) = nums.span( _ == 1)
// prefix = List(1,1,1), suffix = List(2,3,2,1)

val (prefix,suffix) = nums.splitAt(3)
// prefix = List(1,1,1), suffix = List(2,3,2,1)

val (prefix,suffix) = nums.partition( _ == 1)
// prefix = List(1,1,1,1), suffix = List(2,3,2)
```

padTo

```
padTo(len: Int, elem: A): List[A]
```

将列表扩展到指定长度，长度不够的时候，使用elem进行填充，否则不做任何操作。

```
val nums = List(1,1,1)
val padded = nums.padTo(6,2)    // List(1,1,1,2,2,2)
```

combinations & permutations

```
取列表中的n个元素进行组合，返回不重复的组合列表，结果一个迭代器
combinations(n: Int): Iterator[List[A]]

对列表中的元素进行排列，返回不重得的排列列表，结果是一个迭代器
permutations: Iterator[List[A]]
```

```
val nums = List(1,1,3)
val combinations = nums.combinations(2).toList
//List(List(1,1),List(1,3))

val permutations = nums.permutations.toList
// List(List(1,1,3),List(1,3,1),List(3,1,1))
```

zip & zipAll & zipWithIndex & unzip & unzip3

```
zip[B](that: GenIterable[B]): List[(A, B)]
与另外一个列表进行拉链操作，将对应位置的元素组成一个pair，返回的列表长度为两个列表中短的那个

zipAll[B](that: collection.Iterable[B], thisElem: A, thatElem: B): List[(A, B)]
与另外一个列表进行拉链操作，将对应位置的元素组成一个pair，若列表长度不一致，自身列表比较短的话使用thisElem进行填充，对方列表较短的话使用thatElem进行填充

zipWithIndex: List[(A, Int)]
将列表元素与其索引进行拉链操作，组成一个pair

unzip[A1, A2](implicit asPair: (A) ⇒ (A1, A2)): (List[A1], List[A2])
解开拉链操作

unzip3[A1, A2, A3](implicit asTriple: (A) ⇒ (A1, A2, A3)): (List[A1], List[A2], List[A3])
3个元素的解拉链操作
```

```
val alphabet = List("A",B","C")
val nums = List(1,2)
val zipped = alphabet zip nums   //List(("A",1),("B",2))

val zippedAll = alphabet.zipAll(nums,"*",-1)   //List(("A",1),
("B",2),("C",-1))

val zippedIndex = alphabet.zipWithIndex  //List(("A",0),("B",1),
("C",3))

val (list1,list2) = zipped.unzip        //list1 = List("A","B"), li
st2 = List(1,2)

val (l1,l2,l3) = List((1, "one", '1'),(2, "two", '2'),(3, "three",
'3')).unzip3
//l1=List(1,2,3),l2=List("one","two","three"),l3=List('1','2','3')
```

slice

```
slice(from: Int, until: Int): List[A]
提取列表中从位置from到位置until(不含该位置)的元素列表
```

```
val nums = List(1,2,3,4,5)
val sliced = nums.slice(2,4)  //List(3,4)
```

sliding

```
sliding(size: Int, step: Int): Iterator[List[A]]
将列表按照固定大小size进行分组，步进为step，step默认为1,返回结果为迭代器
```

```
val nums = List(1,1,2,2,3,3,4,4)
val groupStep2 = nums.sliding(2,2).toList  //List(List(1,1),Lis
t(2,2),List(3,3),List(4,4))
val groupStep1 = nums.sliding(2).toList //List(List(1,1),Lis
t(1,2),List(2,2),List(2,3),List(3,3),List(3,4),List(4,4))
```

updated

```
updated(index: Int, elem: A): List[A]
对列表中的某个元素进行更新操作
```

```
val nums = List(1,2,3,3)
val fixed = nums.updated(3,4)  // List(1,2,3,4)
```

ListBuffer

ListBuffer基本上包括了前文所写的ArrayBuffer和List的共同点，在此就不多做说明了

Set

- Scala Set(集合)是没有重复的对象集合，**所有的元素都是唯一的**
- Set无序不重复，因此**不能使用索引取值**
- Scala 集合分为可变的和不可变的集合
  默认情况下，Scala 使用的是不可变集合，如果你想使用可变集合，需要引用 scala.collection.mutable.Set 包
  默认引用 scala.collection.immutable.Set
- 其他操作参考List

> **注意：** 虽然可变Set和不可变Set都有添加或删除元素的操作，但是有一个非常大的差别。对不可变Set进行操作，会产生一个新的set，原来的set并没有改变，这与List一样。 而对可变Set进行操作，改变的是该Set本身，与ListBuffer类似。

可变Set

```
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)
```

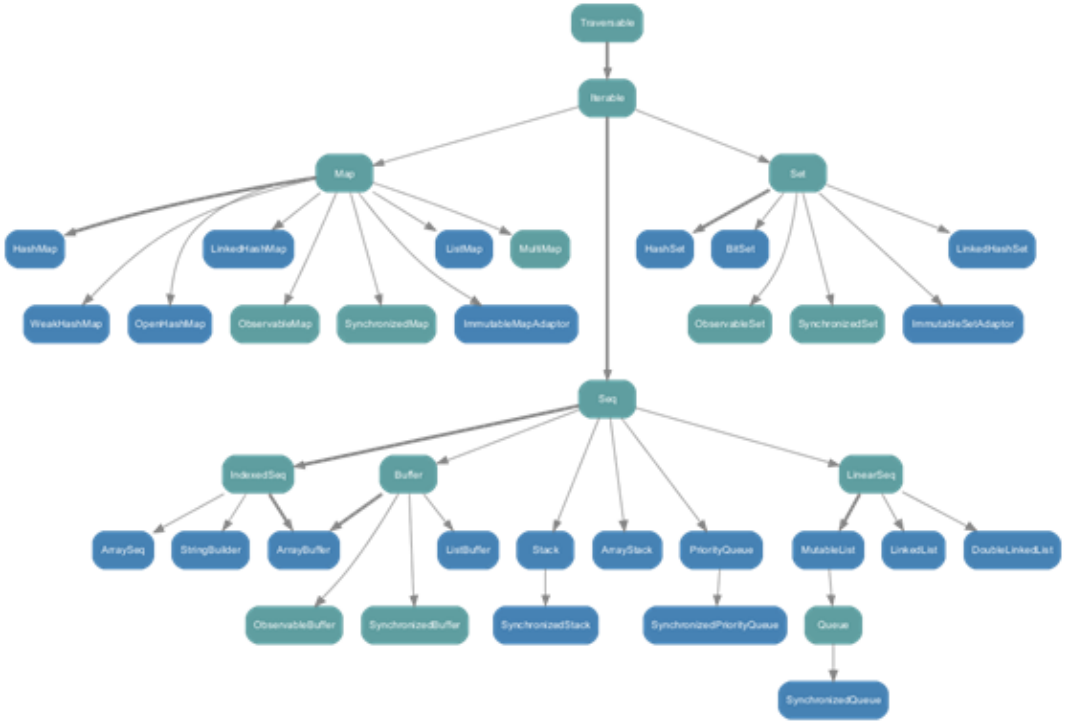
scala集合类的层次结构

scala.collection包中的集合类层次结构如下图：

These are all high-level abstract classes or traits, which generally have mutable as well as immutable implementations.

scala.collection.immutable包中的类层次结构:

scala.collection.mutable包中的类层次结构:



可变集合与不可变集合对应关系：

Map

Map是一种键值对的集合，一般将其翻译为映射

其实Map的操作和前文的也是差不多，这里直接贴代码了，不整理了

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

//map取值
println(map1(1))   //a

//遍历
map3.foreach(
  x => println(s"key:${x._1},value:${x._2}")
)
for (x <- map3) {
  println(s"key:${x._1},value:${x._2}")
}
for ((k, v) <- map3) {
  println(s"key:$k,value:$v")
}
for (ks <- map3.keySet) {
  println(s"key:$ks,value:${map3(ks)}")
}

//不可变的map无法重新给元素赋值
//map3("a") = 100

//get方法，获取指定key的value值
val map4 = Map("zhang" -> 16, "li" -> 20)
println(map4("zhang"))          //16
println(map4.get("zhang"))      //Some(16)
//小括号方法获取key的value值时，如果key不存在，程序会抛异常NoSuchElementExc
eption
//println(map4("wang")) // 获取没有的会报错
println(map4.get("wang"))       //None

//map ++
println(map3.++(map4))        //Map(a -> 1, b -> 2, zhang -> 16, li
-> 20)

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

val map5 = Map("apple" -> 5, "pear" -> 4, "peach" -> 5, "banana" ->
7)
//计算出value代表key长度的kv的数量
val count = map5.count(x => x._1.length == x._2)
println(count)          //3
//计算key长度是5的kv的数量
val count1 = map5.count(_._1.length == 5)
println(count1)             //2
```

```
//drop
println(map5)        //Map(apple -> 5, pear -> 4, peach -> 5, banan
a -> 7)
println(map5.drop(2))    //Map(peach -> 5, banana -> 7)
println(map5.dropRight(2))    //Map(apple -> 5, pear -> 4)
//过滤掉key长度等于5的kv对
val filter1 = map5.filterKeys(_.length != 5)
println(filter1)      //Map(pear -> 4, banana -> 7)

val filter2 = map5.filter(_._1.length != 5)
println(filter2)     //Map(pear -> 4, banana -> 7)

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

val map7 = Map("zhang" -> List("zhangfei" -> "shu", "zhangliao" ->
"wei"))
val flatMap1 = map7.flatMap(_._2)
println(flatMap1)        //Map(zhangfei -> shu, zhangliao -> wei)

val map8 = Map("zhang fei" -> "shu", "zhang liao" -> "wei", "cao ca
o" -> "wei", "sun quan" -> "wu")
//按照国家分成N组
val result = map8.groupBy(_._2)
println(result) //Map(wu -> Map(sun quan -> wu), wei -> Map(zhang l
iao -> wei, cao cao -> wei), shu -> Map(zhang fei -> shu))

//按照姓氏分组
val result1 = map8.groupBy(_._1.split("\\s")(0))
println(result1)//Map(sun -> Map(sun quan -> wu), zhang -> Map(zhan
g fei -> shu, zhang liao -> wei), cao -> Map(cao cao -> wei))

//head tail init last同样可以用

//map方法，对kv对进行映射转换
val map9 = Map("小张" -> 8000, "小李" -> 4000, "小王" -> 2000)

//每人工资涨500
val upSalary = map9.map(x => (x._1, x._2 + 500))
println(upSalary)//Map(小张 -> 8500, 小李 -> 4500, 小王 -> 2500)
val upSalary1 = map9.mapValues(_ + 500)
println(upSalary1)//Map(小张 -> 8500, 小李 -> 4500, 小王 -> 2500)

//工资大于4000的为高收入，否则为低收入
//在姓名前打上高收入或者低收入的标签
```

```
val tagResult = map9.map(x => (s"${if (x._2 > 4000) "[高收入]" else
"[低收入]"}${x._1}", x._2))
println(tagResult)//Map([高收入]小张 -> 8000, [低收入]小李 -> 4000, [低
收入]小王 -> 2000)

//key最大最小值
println(map1.max)  //(2,b)
//maxBy  map9根据工资来返回最大值
val maxSalary1 = map9.maxBy(_._2)
println(maxSalary1)   //(小张,8000)

val maxSalary2 = map9.map(x => (x._2, x._1)).max
println(maxSalary2)   //(8000,小张)

//把map中所有value加起来
val salaryMonth = map9.reduce((x1, x2) => ("月支出", x1._2 + x2._2))
println(salaryMonth)   //(月支出,14000)

val salaryMonthFold = map9.fold(("月支出", 0))((c, x) => (c._1, c._2
+ x._2))
println(salaryMonthFold)   //(月支出,14000)

//reduce和fold唯一不同是,fold需要一个初值,而reduce不需要,它们的不足是,都会受
到输入的类型限制,迭代函数
//的输入类型要和输出类型保持一致
//foldRight ,foldLeft , aggragate 没有这个限制
val salaryMonthlyFoldLeft = map9.foldLeft(0)((c, x) => c + x._2)
println(s"月支出计算:${salaryMonthlyFoldLeft}")//月支出计算:14000

val salaryMonthAggragate = map9.aggregate(0)(
  (c, x) => c + x._2
  , (c1, c2) => c1 + c2
)
println(s"月支出计算:${salaryMonthAggragate}")//月支出计算:14000
```

可变Map

```
//可变map的声明定义
val mmap1 = scala.collection.mutable.Map(1 -> "a", 2 -> "b")
println(mmap1)          //Map(2 -> b, 1 -> a)
println(mmap1(1))       //a
println(mmap1.get(1))   //Some(a)
mmap1.put(3, "c")
println(mmap1)          //Map(2 -> b, 1 -> a, 3 -> c)
mmap1.remove(3)
println(mmap1)          //Map(2 -> b, 1 -> a)

mmap1 += (4 -> "d")
println(mmap1)          //Map(2 -> b, 4 -> d, 1 -> a)

mmap1.update(1,"修改后")
println(mmap1)          //Map(2 -> b, 4 -> d, 1 -> 修改后)
```

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

```
def main(args: Array[String]): Unit = {
  val value1 = getValue(3)
  val value2 = getValue(-3)
  println(value1)   //Some(3)
  println(value2)   //None
  println(value1.get)   //3
  val value3 = getValue(-1)
  // getOrElse(0),参数指的是默认值
  println(value3.getOrElse(0))   //0
}

def getValue(x: Int): Option[Int] = {
  if(x>0) Some(x)
  else None
}
```

Some

Some封装了返回值

```
val some1 = Some("abc")
val some2 = Some(true)
println(some1)          //Some(abc)
println(some2)          //Some(true)
```

元组

与列表一样，**元组也是不可变的**，但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

元组的声明、字面量、取值

```
//元组的声明、字面量、取值
val tuple = (1,2,"a",3.2,true)
val parson = (1,"小张",18)
val pairTuple = ("a",1)//等同于（"a"->1）
println(tuple)          //(1,2,a,3.2,true)
println(parson)         //(1,小张,18)
println(pairTuple)      //(a,1)
println(parson._2)      //小张

//元组也是不可变的，定义之后不能发生变化
//parson._2 = "li"
//元组可以进行多个变量定义和赋值
val list = List(1,2,3)
val (one,two,three) = (list,"a",true)
println(one)            //List(1, 2, 3)
println(two)            //a
println(three)          //true
```

元组的遍历

```
//没有循环遍历方法，下面的错误代码
for (i <- tuple) {
  println(i)
}
```

元组封装返回值

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