Scala核心编程

-数据结构

尚硅谷研究院

数据结构(下)-集合操作

- **集合元素的映射-map映射操作**

看一个实际需求

要求：请将List(3,5,7) 中的所有元素都 * 2 ，将其结果放到一个新的集合中返回，即返回一个新的List(6,10,14), 请编写程序实现.

使用传统的方法解决

- 传统方法优点分析
- 传统方法缺点分析

```
val list1 = List(3, 5, 7)
var list2 = List[Int]()
for (item <- list1) { //遍历
list2 = list2 :+ item * 2
}
println(list2)
```

- 集合元素的映射-map映射操作

map映射操作

上面提出的问题，其实就是一个关于集合元素映射操作的问题。
在Scala中可以通过map映射操作来解决：将集合中的每一个元素通过指定功能（函数）映射（转换）成新的结果集合这里其实就是所谓的将函数作为参数传递给另外一个函数,这是函数式编程的特点

以HashSet为例说明

```
def map[B](f: (A) ⇒ B): HashSet[B]   //map函数的签名
```

1) 这个就是map映射函数集合类型都有
2) [B] 是泛型
3) map 是一个高阶函数(可以接受一个函数的函数，就是高阶函数)，可以接收函数 f: (A) => B 后面详解(先简单介绍下.)
4) HashSet[B] 就是返回的新的集合

- 集合元素的映射-map映射操作

使用map映射函数来解决

```
val list1 = List(3, 5, 7)
def f1(n1: Int): Int = {
2 * n1
}
val list2 = list1.map(f1)
println(list2)
```

高阶函数基本使用

```
object TestHighOrderDef {
  def main(args: Array[String]): Unit = {
    val res = test(sum, 6.0)
    println("res=" + res)
  }
  def test(f: Double => Double, n1: Double) = {
    f(n1)
  }
  def sum(d: Double): Double = {
    d + d
  }}
```

为了进一步理解，我们在举一个高阶函数的案例

```
def main(args: Array[String]): Unit = {
test2(sayOK)}
def test2(f: () => Unit) = {
f()}
def sayOK() = {
println("sayOKKK...")}
```

- 集合元素的映射-map映射操作

深刻理解map映射函数的机制-模拟实现

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

  val list1 = List(3, 5, 7)
  def f1(n1: Int): Int = {
    println("xxx")
    2 * n1
  }
  val list2 = list1.map(f1)
  println(list2)

  val myList = MyList()
  val myList2 = myList.map(f1)
  println("myList2=" + myList2)
  println("myList=" + myList.list1)

}
```

```
class MyList {
  var list1 = List(3, 5, 7)
  var list2 = List[Int]()
  def map(f:Int=>Int): List[Int] = {
    for (item<-list1) {
      list2 = list2 :+ f(item)
    }
    list2
  }
}
object MyList {
  def apply(): MyList = new MyList()
}
```

- **集合元素的映射-map映射操作**

课堂练习

请将 val names = List("Alice", "Bob", "Nick") 中的所有单词，全部转成字母大写，返回到新的List集合中.

```
val names = List("Alice", "Bob", "Nick")
def upper(s:String): String = {
s.toUpperCase
}
val names2 = names.map(upper)
println("names=" + names2)
```

- 集合元素的映射-map映射操作

flatmap映射：flat即压扁，压平，扁平化映射

flatmap：flat即压扁，压平，扁平化，效果就是将集合中的每个元素的子元素映射到某个函数并返回新的集合。

看一个案例：

```
val names = List("Alice", "Bob", "Nick")
def upper( s : String ) : String = {
    s. toUpperCase
}
//注意：每个字符串也是char集合
println(names.flatMap(upper))
```

- 集合元素的过滤-filter

filter：将符合要求的数据(筛选)放置到新的集合中

应用案例：将 val names = List("Alice", "Bob", "Nick") 集合中首字母为'A'的筛选到新的集合。

思考：如果这个使用传统的方式，如何完成?

```
val names = List("Alice", "Bob", "Nick")
def startA(s:String): Boolean = {
s.startsWith("A")
}
val names2 = names.filter(startA)
println("names=" + names2)
```

- 化简

看一个需求:

val list = List(1, 20, 30, 4 ,5) , 求出list的和.

化简:

化简：将二元函数引用于集合中的函数,。

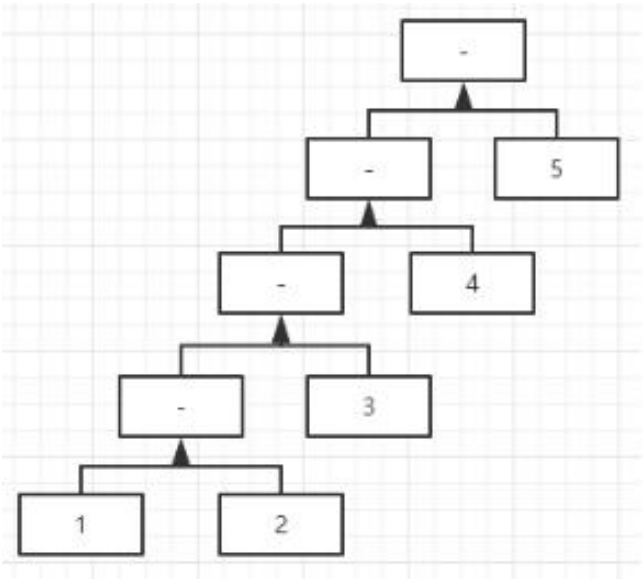
上面的问题当然可以使用遍历list方法来解决，这里我们使用scala的化简方式来完成。[案例演示+代码说明]

```
val list = List(1, 20, 30, 4, 5)
def sum(n1: Int, n2: Int): Int = {
n1 + n2
}
val res = list.reduceLeft(sum)
println("res=" + res)
```

```
//说明
1)  def reduceLeft[B >: A](@deprecatedName('f) op: (B, A) => B): B
2)  reduceLeft(f) 接收的函数需要的形式为 op: (B, A) => B): B
3)  reduceleft(f) 的运行规则是 从左边开始执行将得到的结果返回给第一个参
4)   然后继续和下一个元素运行，将得到的结果继续返回给第一个参数，继
5)  即: //((((1 + 2)  + 3) + 4) + 5) = 15
```

- 化简

reduceLefft(_ + _)这个函数的执行逻辑如图:



说明: .reduceRight(_ - _)反之同理

- 化简

课堂练习题

1) 分析下面的代码输出什么结果

```
val list = List(1, 2, 3, 4 ,5)
    def minus( num1 : Int, num2 : Int ): Int = {
      num1 - num2
    }
println(list.reduceLeft(minus)) // 输出?
println(list.reduceRight(minus)) //输出?
println(list.reduce(minus)) //是哪个，看源码秒懂
```

2) 使用化简的方法求出 List(3,4,2,7,5) 最小的值

- 折叠

基本介绍

fold函数将上一步返回的值作为函数的第一个参数继续传递参与运算，直到list中的所有元素被遍历。

1) 可以把reduceLeft看做简化版的foldLeft。
   如何理解:

```
def reduceLeft[B >: A](@deprecatedName('f) op: (B, A) => B): B =
  if (isEmpty) throw new UnsupportedOperationException("empty.reduceLeft")
  else tail.foldLeft[B](head)(op)
```

   大家可以看到. reduceLeft就是调用的foldLeft[B](head)，并且是默认从集合的**head元素**开始操作的。

2) 相关函数：fold，foldLeft，foldRight，可以参考reduce的相关方法理解

- 折叠

应用案例

看下面代码看看输出什么，并分析原因.

```
// 折叠
val list = List(1, 2, 3, 4)
def minus( num1 : Int, num2 : Int ): Int = {
  num1 - num2
}
println(list.foldLeft(5)(minus)) // 函数的柯里化
println(list.foldRight(5)(minus)) //
```

- 折叠

foldLeft和foldRight 缩写方法分别是：/:和:

```
val list4 = List(1, 9, 2, 8)
def minus(num1: Int, num2: Int): Int = {
num1 - num2
}
var i6 = (1 /: list4) (minus) // =等价=> list4.foldLeft(1)(minus)
println(i6) // 输出?
i6 = (100 /: list4) (minus)
println(i6) // 输出?
i6 = (list4 :\ 10) (minus) // list4.foldRight(10)(minus)
println(i6) // 输出?
```

- 扫描

基本介绍

扫描，即对某个集合的所有元素做fold操作，但是会把产生的所有中间结果放置于一个集合中保存

应用实例

```
def minus( num1 : Int, num2 : Int ) : Int = {
num1 - num2
}
//5 (1,2,3,4,5) =>(5,4,2,-1,-5,-10)
val i8 = (1 to 5).scanLeft(5)(minus) //IndexedSeq[Int]
println(i8)
def add( num1 : Int, num2 : Int ) : Int = {
num1 + num2
}
//5 (1,2,3,4,5) =>(5,6,8, 11,15,20)
val i9 = (1 to 5).scanLeft(5)(add) //IndexedSeq[Int]
println(i9)
```

- 扫描

课堂练习

请写出下面的运行结果:

```
def test( num1 : Int, num2 : Int ) : Int = {
  num1 * num2
}
val i8 = (1 to 3).scanLeft(3)(test)
println(i8)
```


图片2.z

结果是: 3, 3, 6,18

- 集合综合应用案例

课堂练习1

val sentence = "AAAAAAAAAABBBBBBBBCCCCCDDDDDDD"

将**sentence** 中各个字符，通过foldLeft存放到 一个ArrayBuffer中

目的：理解flodLeft的用法.

```
val sentence = "AAAAAAAAAABBBBBBBBCCCCCDDDDDDD"

def putArry( arr : ArrayBuffer[Char], c : Char ): ArrayBuffer[Char] = {
arr.append(c)
arr
}
//创建val arr = ArrayBuffer[Char]()
val arr = ArrayBuffer[Char]()
sentence.foldLeft(arr)(putArry)
```

- 集合综合应用案例

课堂练习2

val sentence = "AAAAAAAAAABBBBBBBBCCCCCDDDDDDD"
使用映射集合，统计一句话中，各个字母出现的次数
提示：Map[Char, Int]()

1) 看看java如何实现

```
String sentence = "AAAAAAAAAABBBBBBBBCCCCCDDDDDDD";
Map<Character, Integer> charCountMap =
                new HashMap<Character, Integer>();
char[] cs = sentence.toCharArray();
for ( char c : cs ) {
    if ( charCountMap.containsKey(c) ) {
                Integer count = charCountMap.get(c);
                charCountMap.put(c, count + 1);
    } else {
                charCountMap.put(c, 1);}}
System.out.println(charCountMap);
```

- 集合综合应用案例

课堂练习2

val sentence = "AAAAAAAAAABBBBBBBBCCCCCDDDDDDD"
使用映射集合，统计一句话中，各个字母出现的次数
提示：Map[Char, Int]()

1) 使用scala的flodLeft折叠方式实现.

```
val sentence = "AAAAAAAAAABBBBBBBBCCCCCDD"
def charCount( map : Map[Char, Int], c : Char ): Map[Char, Int] = {
map + (c -> (map.getOrElse(c, 0) + 1))
}
val map2 = sentence.foldLeft(Map[Char, Int]())(charCount)
println(map2)
```

- 集合综合应用案例

课后练习3-大数据中经典的wordcount案例

val lines = *List*(**"atguigu han hello "**, **"atguigu han aaa aaa aaa ccc ddd uuu"**)

使用映射集合，**list**中，各个单词出现的次数，并按出现次数排序

提示：

代码：

```
val lines = List("atguigu han hello ", "atguigu han aaa aaa aaa ccc ddd uuu")

  val res1 = lines.flatMap(_.split(" "))
  println("res1=" + res1)
  // res1.map 说明
  //1. 使用map，返回对偶元组 形式为
  //List((hello,1), (tom,1), (hello,1), (jerry,1), (hello,1), (jerry,1), (hello,1), (kitty,1))
  val res2 = res1.map((_, 1))
  println("res2=" + res2)
  // res2.groupBy(_._1)
  //1. 分组的根据是以元素来分组
  //2. _._1 中的第一个 _ 表示取出的各个对偶元组比如 (hello,1)
  //3. _._1 中的_1, 表示对偶元组的第一个元素，比如 hello
  //4. 因此 _._1 表示我们分组的标准是按照对偶元组的第一个元素进行分组
  //5. 返回的形式为 Map(tom -> List((tom,1)), kitty -> List((kitty,1)), jerry -> List((jerry,1), (jerry,1)), hello -> List((hello,1), (hello,1), (hello,1), (hello,1)))
  val res3 = res2.groupBy(_._1)
  println("res3=" + res3)

  // x=>(x._1, x._2.size) 传入一个匿名函数，完成统计
  //1.x 表示传入的Map中的各个元素，比如 jerry -> List((jerry,1), (jerry,1))
  //2.x._1 表示 jerry
  //3.x._2.size，表示对 List((jerry,1), (jerry,1))求size,是多少就是多少
  //4.结果是 res4=Map(han -> 2, atguigu -> 2, hello -> 1)
  //5.到此结果就出来了，但是没有排序
  val res4 = res3.map(x=>(x._1, x._2.size))
  println("res4=" + res4 )

  // res4.toList.sortBy(_._2)
  //1. toList先将map转成 list,为了下一步排序
  //5. sortBy就是排序,以对偶元组的第二个值排序，就是大小排序
  val res5 = res4.toList.sortBy(_._2)
  println("res5=" + res5)

  //如果希望从大到小排序，执行reverse即可
  val res6 = res5.reverse
```

- **扩展-拉链(合并)**

基本介绍

在开发中，当我们需要将两个集合进行 **对偶元组合并**，可以使用拉链。

应用实例

```
// 拉链
val list1 = List(1, 2 ,3)
val list2 = List(4, 5, 6)

val list3 = list1.zip(list2) // (1,4),(2,5),(3,6)
println("list3=" + list3)
```

- 扩展-拉链(合并)

注意事项

1) 拉链的本质就是两个集合的合并操作，合并后每个元素是一个 对偶元组。
2) 操作的规则下图:

```
// 拉链
val list1 = List(1, 2, 3)
val list2 = List(4, 5, 6)
```

3) 如果两个集合个数不对应，会造成数据丢失。
4) 集合不限于List, 也可以是其它集合比如 Array
5) **如果要取出合并后的各个对偶元组的数据，可以遍历**

```
for(item<-list3){
 print(item._1 + " " + item._2) //取出时，按照元组的方式取出即可
}
```

- 扩展-迭代器

基本说明

通过iterator方法从集合获得一个迭代器，通过while循环和for表达式对集合进行遍历.(学习使用迭代器来遍历)

应用案例

```
val iterator = List(1, 2, 3, 4, 5).iterator // 得到迭代器
    println("--------遍历方式1 ----------------")
    while (iterator.hasNext) {
        println(iterator.next())
    }
    println("--------遍历方式2 for ----------------")
    for(enum <- iterator) {
      println(enum) //
    }
```

【案例演示+代码说明】

- 扩展-迭代器

应用案例小结

1) iterator 的构建实际是 **AbstractIterator 的一个匿名子类**，该子类提供了

```
/*
 def iterator: Iterator[A] = new AbstractIterator[A] {
var these = self
def hasNext: Boolean = !these.isEmpty
def next(): A =
*/
```

2) 该AbstractIterator 子类提供了 **hasNext next** 等方法.
3) 因此，我们可以使用 while的方式，使用hasNext next 方法变量

- 扩展-流 Stream

基本说明

stream是一个集合。这个集合，可以用于存放无穷多个元素，但是这无穷个元素并不会一次性生产出来，而是需要用到多大的区间，就会动态的生产，末尾元素遵循lazy规则(即：要使用结果才进行计算的) 。

创建Stream对象

➢ 案例:

```
def numsForm(n: BigInt) : Stream[BigInt] = n #:: numsForm(n + 1)
val stream1 = numsForm(1)
```

➢ 说明

1) Stream 集合存放的数据类型是BigInt
2) numsForm 是自定义的一个函数，函数名是程序员指定的。
3) 创建的集合的第一个元素是 n , 后续元素生成的规则是 n + 1
4) 后续元素生成的规则是可以程序员指定的 ， 比如 numsForm(n * 4)...

- 扩展-流 Stream

使用tail，会动态的向stream集合按规则生成新的元素

```
//创建Stream
def numsForm(n: BigInt) : Stream[BigInt] = n #:: numsForm(n + 1)
val stream1 = numsForm(1)
println(stream1) //
//取出第一个元素
println("head=" + stream1.head) //
println(stream1.tail) //
println(stream1) //?
```

注意： 如果使用流集合，就不能使用last属性，如果使用last集合就会进行无限循环

- 扩展-流 Stream

使用map映射stream的元素并进行一些计算

```
//创建Stream
def numsForm(n: BigInt) : Stream[BigInt] = n #:: numsForm(n + 1)
def multi(x:BigInt) : BigInt = {
x * x
}
println(numsForm(5).map(multi)) //? (25,?)
```

- 扩展-视图 View

基本介绍

Stream的懒加载特性，也可以对其他集合应用view方法来得到类似的效果，具有如下特点：

1) view方法产出一个**总是被懒执行**的集合。
2) view**不会缓存数据**，每次都要重新计算，比如遍历View时。

应用案例

请找到1-100 中，数字倒序排列 和它本身相同的所有数。(1 2, 11, 22, 33 ...)

```
def multiple(num: Int): Int = {
num}
def eq(i: Int): Boolean = {
i.toString.equals(i.toString.reverse)
}
//说明: 没有使用view
val viewSquares1 = (1 to 100)
.map(multiple)
.filter(eq)
println(viewSquares1)
//for (x <- viewSquares1) {}
//使用view
val viewSquares2 = (1 to 100)
.view
.map(multiple)
.filter(eq)
println(viewSquares2)
```

- 扩展-线程安全的集合

基本介绍

所有线程安全的集合都是以Synchronized开头的集合

SynchronizedBuffer

SynchronizedMap

SynchronizedPriorityQueue

SynchronizedQueue

SynchronizedSet

SynchronizedStack

- 扩展-并行集合

基本介绍

1) 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。

2) 主要用到的算法有：
Divide and conquer : 分治算法，Scala通过splitters(分解器)，combiners（组合器）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

Work stealin算法【学数学】，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

- 扩展-并行集合

应用案例

parallel(pærəlel并行)

1) 打印**1~5**

```
(1 to 5).foreach(println(_))
println()
(1 to 5).par.foreach(println(_))
```

2) 查看并行集合中元素访问的线程

```
val result1 = (0 to 100).map{case _ => Thread.currentThread.getName}
val result2 = (0 to 100).par.map{case _ => Thread.currentThread.getName}
println(result1)
println(result2)
```

- 扩展-操作符

基本介绍

这部分内容没有必要刻意去理解和记忆，语法使用的多了，自然就会熟练的使用，该部分内容了解一下即可。

操作符扩展

1) 如果想在变量名、类名等定义中使用语法关键字（保留字），可以配合反引号反引号 [案例演示]

   val `val` = 42

2) 中置操作符：**A 操作符 B** 等同于 **A.操作符(B)**

```
val n1 = 1
val n2 = 2
val r1 = n1 + n2
val r2 = n1.+(n2) //看Int的源码即可说明t
println("r1=" + r1 + " r2=" + r2)
val dog = new Dog
dog.+(90)
dog + 10
print(dog.age) // 101
```

- 扩展-操作符

操作符扩展

3) 后置操作符：A操作符 等同于 A.操作符，如果操作符定义的时候不带()则调用时不能加括号 [案例演示+代码说明]

```
// 操作符
val oper = new Operate
println(oper++)
println(oper.++)
```

```
class Operate {
  //定义函数/方法的时候，省略的()
  def ++ = "123"
}
```

4) 前置操作符，+、-、！、~等操作符A等同于A.unary_操作符 [案例演示]

```
class Operate {
  // 声明前置运算符
  //unary ： 一元运算符
  def unary_! = println("!!!!!!!")
}
```

```
// 操作符
val oper = new Operate
!oper //前置运算符
```

5) 赋值操作符，A 操作符= B 等同于 A = A 操作符 B ， 比如 A += B 等价 A = A + B

谢谢！ 欢迎收看！