# Scala语法拾遗

## foldLeft等高级函数

应用场景：

val featureMap\_rdd: RDD[((Long, String), mutable.HashMap[Any, Long])] = new\_rdd.mapValues{

case (property\_arr, filsiz) => {

property\_arr.foldLeft(new mutable.HashMap[Any, Long]()){

case (muMap, property) if property != "" && property != -1L => {

muMap += property -> (muMap.getOrElse(property, 0L) + 1L)

}

muMap += "filsiz" -> (muMap.getOrElse("filsiz", 0L) + math.log(filsiz).toLong)

}

}

}

我们不想然所有的property参与foldLeft而需要加一个判断，这一点aggregate不行。

## Array[Array[T]]的转置重写

object Transposer{

implicit class TransArr[T](val matrix: Array[Array[T]]){

def transposeee(): Seq[Seq[T]] =

{

Array.range(0, matrix.head.length).map(i => matrix.view.map(\_(i)))

}

}

implicit class TransSeq[T](val matrix: Seq[Seq[T]]){

def transposeee(): Seq[Seq[T]] =

{

Array.range(0, matrix.head.length).map(i => matrix.view.map(\_(i)))

}

}

}

val matrix = Seq(Seq(0, 1, 0), Seq(0, 0, 1), Seq(1, 0, 0))

matrix.foreach(arr => println(arr.mkString(", ")))

// 转置

import Transposer.\_

matrix.transposeee().foreach(arr => println(arr.mkString(", ")))

## Array.slice源码

def slice(from: Int, until: Int): Repr = {

val lo = math.max(from, 0)

val hi = math.min(math.max(until, 0), length)

val elems = math.max(hi - lo, 0)

val b = newBuilder

b.sizeHint(elems)

var i = lo

while (i < hi) {

b += self(i)

i += 1

}

b.result

}

左闭右开

## Concat

## Option在高阶函数中的应用

## equals、eq和==方法

### 定义

* final def ==(arg0: Any): Boolean  
  The expression x == that is equivalent to if (x eq null) that eq null else x.equals(that).
* final def eq(arg0: AnyRef): Boolean  
  Tests whether the argument (that) is a reference to the receiver object (this).
* def equals(arg0: Any): Boolean  
  The equality method for reference types.

### 理解

简言之，equals方法是检查**值**是否相等，而eq方法检查的是**引用**是否相等。所以如果比较的对象是null那么==调用的是eq，不是null的情况调用的是equals。  
equals和eq在null比较中的区别

equals在比较null时是不安全的，而eq可以，进而==也可以

val a = null

val b = null

// println(a.equals(b)) // not compile, NullPointerException

println(a.eq(b))

println(a == b)

### equals和eq在其他对象比较中的区别

常见的scala内置类都包含

#### case class

在java中如果要对两个对象进行值比较，那么必须要实现equals 和hashCode方法。而在scala中为开发者提供了case class，默认实现了equals 和hashCode方法。

scala> case class Bread(brand:String, price:Int)

defined class Bread

scala> val b1 = Bread("BreadTalk", 50)

b1: Bread = Bread(BreadTalk,50)

scala> val b2 = Bread("BreadTalk", 60)

b2: Bread = Bread(BreadTalk,60)

scala> b1 eq b2

res2: Boolean = false

scala> b1 equals b2

res3: Boolean = true

而对于Array或者Map对象不能简单点使用equals进行**值**比较，要通过sameElements方法，例如：

scala> val a1 = Array("x", "y")

a1: Array[String] = Array(x, y)

scala> val a2 = Array("x", "y")

a2: Array[String] = Array(x, y)

scala> a1 equals a2

res4: Boolean = false

scala> a1 eq a2

res5: Boolean = false

scala> a1 sameElements a2

res6: Boolean = true

scala> val m1 = Map(1->"x", 2->"y")

m1: scala.collection.immutable.Map[Int,String] = Map(1 -> x, 2 -> y)

scala> val m2 = Map(1->"x", 2->"y")

m2: scala.collection.immutable.Map[Int,String] = Map(1 -> x, 2 -> y)

scala> m1 sameElements m2

res7: Boolean = true

scala> val m3 = Map(1->"x", 2->"z")

m3: scala.collection.immutable.Map[Int,String] = Map(1 -> x, 2 -> z)

scala> m1 sameElements m3

res8: Boolean = false

如果Array中存的是对象，也是一样的，例如

scala> case class Bread(brand:String, price:Int)

defined class Bread

scala> val b1 = Bread("BreadTalk", 50)

b1: Bread = Bread(BreadTalk,50)

scala> val b2 = Bread("BreadTalk", 50)

b2: Bread = Bread(BreadTalk,50)

scala> val b3 = Bread("BreadTalk", 60)

b3: Bread = Bread(BreadTalk,60)

scala> val a1 = Array(b1)

a1: Array[Bread] = Array(Bread(BreadTalk,50))

scala> val a2 = Array(b2)

a2: Array[Bread] = Array(Bread(BreadTalk,50))

scala> val a3 = Array(b3)

a3: Array[Bread] = Array(Bread(BreadTalk,60))

scala> a1 equals a2

res0: Boolean = false

scala> a1 sameElements a2

res1: Boolean = true

scala> a1 equals a3

res2: Boolean = false

scala> a1 sameElements a3

res3: Boolean = false  
  
主要内容转载自：https://www.jianshu.com/p/7b2b19d2fe7d，部分原创

## 一般类的深拷贝问题

深拷贝和浅拷贝的区别就是一个赋值是引用,另一个赋值直接将值赋予对象.java创建的一般对象进行赋值是浅拷贝。

### 示例

class Params {

var values: String = ""

var separator: String = "##||##"

/\*\* 将新的参数添加进来 \*/

def append(newParam: String): this.type = {

this.values = if(values.equals("")){

newParam

} else {

this.values + separator + newParam}

this

}

/\*\* 设定分隔不同节点参数的分隔符 \*/

def setSeparator(separator: String): this.type = {

this.separator = separator

this

}

/\*\* 直接更新本节点的参数 \*/

def update(params: String): this.type = {

this.values = params

this

}

/\*\* 创建一个copy方法，区别引用赋值，解决算子多次执行自我append的问题 \*/

def copy: Params = {

val newParams = new Params

newParams.update(this.values)

newParams

}

}

val a = new Params

a.append("aa")

val b = a

b.append("bb")

val c = a

c.append("cc")

println(a.values)

println(b.values)

println(c.values)

> aa##||##bb##||##cc

> aa##||##bb##||##cc

> aa##||##bb##||##cc

我们发现三个引用的是一个对象

val a = new Params

a.append("aa")

val b = a.copy

b.append("bb")

val c = a.copy

c.append("cc")

println(a.values)

println(b.values)

println(c.values)

> aa

> aa##||##bb

> aa##||##cc

上述代码可以放到一个情景中就是:

def eat(food: string){

a.append("eat" + food)

}

def watch(book: string){

a.append("watch" + book)

}

def warning{

if(a != ""){

}

}

每天执行一次

想监控每天吃的和看的，如果和日历上昨天吃的和看的发生了变化就警告。此时的场景是适合浅拷贝的，否则每天执行一次，会发生一下场景：eat：A，wacht：B，eat：A，wacht：B，eat：A，wacht：B，不断累加，每天都会比历史日志多出来eat：A，wacht：B，因而永远不会和历史日子一致，虽然我每天吃的和看的偶相同

## Scala.specialized

Class DenseMatrix[@specialized (Double, Int, Float, Long) V](….)

## Java.util.Random中的nextGussian和spark.mllib.random.RandomRDDs.normalRDD的差异

如果新建一个随机器，val rd = new java.util.Random(123L)放到分区中会使得每个分区中的随机数会重复。

NormalRDD解决了这个问题

## 一个状态监控和更新的类框架

**class** status(**var** stage: String, **var** stageStatus: Boolean) {  
 */\*\* 日志信息 \*/* **var** *logInfo*: String = stage + ":" + stageStatus.toString  
  
 */\*\* 状态更换函数 \*/* **def** replace(newStage: String, newStageStatus: Boolean): **this**.**type** = {  
 **this**.stage = newStage  
 **this**.stageStatus = newStageStatus  
 **val** newLogInfo: String = **this**.stage + ":" + **this**.stageStatus.toString  
 *logInfo* += " => " + newLogInfo  
 **this** }  
  
 */\*\* 状态更新函数 \*/* **def** update(newStage: String, newStageStatus: Boolean): **this**.**type** = {  
 **this**.stage = newStage  
 **this**.stageStatus = **this**.stageStatus && newStageStatus  
 **val** newLogInfo: String = **this**.stage + ":" + **this**.stageStatus.toString  
 *logInfo* += " => " + newLogInfo  
 **this** }  
}

## 对于view的应用

在连续的集合操作，尤其是两头小中间大（即中间步骤会产生较大集合）时view可以实现惰性运算，不会生成大量的中间集合，效率较高。

// view的应用

val a = 0 until 100000

val sumAll = a.zip(100000 until 200000).flatMap(x => Array.tabulate(x.\_1)(i => x.\_2)).sum

println(sumAll) // not compile, Exception: Out of GC

val sumAllView = a.view.zip(100000 until 200000).flatMap(x => Array.tabulate(x.\_1)(i => x.\_2)).sum

println(sumAllView)

## Scala中ArrayBuffer +=的问题

这是scala的一个缺陷

ArrayBuffer +=识别不了元组，会把元组当做映射来处理

描述：

val buff = ArrayBuffer.empty[(Int, Double)]

buff += (1, 1.0)不行

必须

Val v = (1, 1.0)

Buff += v

## StringBuilder, StringBuffer和String

1. 在性能方面

StringBuilder > StringBuffer > String

1. 在线程安全方面

StringBuffer > StringBuilder

## Scala.util.control.Breaks

Scala的break是创建一个BreakControl的异常（继承自Throwable）

import scala.util.control.Breaks.break

try{

if (true) break

}catch {

case e0: ControlThrowable => throw new Exception("流异常" + e0.getMessage) // 貌似两个没什么区别

case e1: Throwable => throw new Exception("break异常" + e1.getMessage)

}

## Spark序列化问题

## Spark中map(iterator => …)和mapPartition(iterator => …)

/\*\*

\* 特征列提取

\* ----

\* 按feature分组统计 => 根据给定的特征上限，分组取top => 提取出对应的特征和idf系数

\* ----

\* 同时发现feature + category => count效率太低, category作用又不高, 改为 => feature => count

\* 这里设定一个上界是为了防止GC过大和OM

\* @param bindDF 输入的长表

\*/

def run(bindDF: DataFrame, params: ArrayBuffer[Params4Bind])

: RDD[(String, Vector)] = {

require(sQLContext.isDefined, "SQLContext不能为空")

val ifLog = logarithm

/\*\* 个体的特征频率统计 --[(imsi, feature), count] \*/

val unitFrequency = bindDF.rdd.map(row => {

val tup = (util.Try(row.getAs[String](0)) getOrElse "",

util.Try(row.getAs[String](1)) getOrElse "")

(tup, 1)

}).filter{case (tup, \_) => tup.\_1.length \* tup.\_2.length > 0}.reduceByKey(\_ + \_)

/\*\* 整体的特征频率统计 + idf转换 --[(feature, count)]\*/

val ordering: Ordering[(String, Int)] = Ordering.by[(String, Int), Int](\_.\_2)

val idfFrequency: Map[String, Double] = unitFrequency.map {

case ((\_, featureV), countV) => (featureV, countV)

}.reduceByKey(\_ + \_).top(maxFeatureNum.toInt)(ordering).map {

case (featureWithLabel, tf) =>

(featureWithLabel, if(ifLog) 1.0 / math.log1p(tf) else 1.0 / (tf + 1.0))

}.toMap

val idfWithIndex: Map[String, Int] = idfFrequency.keySet.zipWithIndex.toMap

val sc = bindDF.sqlContext.sparkContext

val idfBC: Map[String, Double] = sc.broadcast(idfFrequency).value

val indexBC: Map[String, Int] = sc.broadcast(idfWithIndex).value

val size = idfBC.size

unitFrequency.map{case ((imsiV, featureV), countV) => (imsiV, (featureV, countV))}

.groupByKey()

.mapValues( iter => {

val iterLog = iter.map{case (f, c) => (f, if(ifLog) math.log1p(c) else c.toDouble)}

val freqSum = iterLog.map(\_.\_2).sum

val freqScores = iterLog.filter{

case (featureV, \_) => idfBC.contains(featureV)

}.map{

case (featureV, countV) => {

val index = indexBC(featureV)

val idf = idfBC(featureV)

(index, idf \* countV / freqSum)

}

}.toSeq

Vectors.sparse(size, freqScores)

})

}

对于里面的iterable类返回的也是iterable类，如果filter之后为空，返回的为一个空数组

(A,List(100))

(K,List())

例子1

GroupRdd中的是Iterable

val m = Array("A", "B", "C", "D", "E", "F", "G",

"H", "I", "J", "K", "L", "M", "N")

val arr = Array.tabulate(1000)(i => if(i < 3) (i, 100) else (i % 14, i / 50))

val rdd = sc.parallelize(arr, 10)

val groupRdd = rdd.map(arr => (m(arr.\_1), arr.\_2))

// groupRdd.groupByKey().foreach(println)

val newRdd = groupRdd.groupByKey().mapValues(iter => {

val filterIter = iter.filter(i => i >= 100)

filterIter

})

newRdd.foreach(println)

例子2

MapPartitions中的iter是Iterator

/\*\*

\* 找出大于100的值

\*/

groupRdd.mapPartitions(iterator => {

val result = ArrayBuffer.empty[(String, Int)]

while(iterator.hasNext){

if(iterator.next().\_2 >= 100)

result += iterator.next()

}

result.toIterator

}).foreach(println)

效果

(M,List())

(C,List(100))

例子3也说明其实对于基于iterable的迭代（包括List、Array、Iterator等）好处就是返回的数目不需要和原值一致。

*/\*\*  
 \* 例子3, mapPartition起到了过滤的作用  
 \*/***val** arr2 = *Array*(("A", 100), ("A", 1), ("A", 2), ("A", 3), ("A", 40),  
 ("B", 10), ("B", 1), ("B", 2), ("B", 3), ("B", 40),  
 ("C", 100), ("C", 1), ("C", 2), ("C", 3), ("C", 40))  
**val** rdd2 = *sc*.parallelize(arr2)  
**val** result = rdd2.mapPartitions(iterator => {  
 **val** result = ArrayBuffer.empty[(String, Int)]  
 **while**(iterator.hasNext){  
 **val** nt = iterator.next()  
 **if**(nt.\_2 >= 100){  
 result += nt  
 result += iterator.next()  
 result += iterator.next()  
 result += iterator.next()  
 }  
 }  
 result.toIterator  
})  
  
result.foreach(*println*)  
*println*(result.filter(\_.\_1 == "B").count())

(A,100)

(A,1)

(A,2)

(A,3)

(C,100)

(C,1)

(C,2)

(C,3)

小结；

1. 无论Iterable还是Iterator都可以返回长度任意（包括0）的Iterable、 Iterator
2. 当Iterable时如果返回0个则会返回一个List()
3. 当Iterator时如果一个分区返回0个值则在提交给Driver端DAGScheduler的分区数据信息中没有改分区，体现在代码中就是RDD没有该分区的值