# Scala语法拾遗

## Scala中Map[(String, Int)]应该写为Map[String, Int]

## 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.ofDim的用法

def lagMat(x: Array[Double], maxLag: Int, includeOriginal: Boolean): Array[Array[Double]] = {

val numObservations = x.length

val numRows = numObservations - maxLag

val initialLag = if (includeOriginal) 1 else 0

val numCols = maxLag + initialLag

val lagMat = Array.ofDim[Double](numRows, numCols)

for (i <- 0 until numRows) {

for (j <- 0 until numCols) {

lagMat(i)(j) = x(i + maxLag - j + initialLag - 1)

}

}

lagMat

}

## 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

## Array.range(i, i + step + 1).reverse要比(i to i + step + 1).reverse多费一次循环

因为Range的reverse调用其实直接首尾对调就可以了，而Array不行。

## Array的flatten用法

val arr3: Array[Array[Array[String]]] = Array(

Array(

Array("A", "B", "C", "D"),

Array("E", "F", "G", "H"),

Array("I", "J", "K", "L")

),

Array(

Array("M", "N", "O", "P"),

Array("Q", "I", "S", "T"),

Array("U", "V", "W", "X"),

Array("Y", "Z", "1", "2")

),

Array(

Array("3", "4", "5", "6"),

Array("7", "8", "9", "0")

)

)

val flattenArr: Array[Array[String]] = arr3.flatten

println(flattenArr.length)

flattenArr.foreach(x => println(x.mkString(",")))

>9

>A,B,C,D

>E,F,G,H

>I,J,K,L

>M,N,O,P

>Q,I,S,T

>U,V,W,X

>Y,Z,1,2

>3,4,5,6

>7,8,9,0

可见：flatten是flatten的最外层。此时flatMap不行。Map+flatten不能由flatMap取代

应用：时间序列变量矩阵的滞后

在sparkts中有关于timeSeries的方阵滞后，但里面的元素是基于spark.mllib.distributed.linag中的Matrix的，想写一个基于breeze的DenseMatrix的。实现的效果和timeSeries中的lag类似：

/\*\*

\* Example input TimeSeries:

\* time a b

\* 4 pm 1 6

\* 5 pm 2 7

\* 6 pm 3 8

\* 7 pm 4 9

\* 8 pm 5 10

\*

\* With maxLag 2, includeOriginals = true and TimeSeries.laggedStringKey, we would get:

\* time a lag1(a) lag2(a) b lag1(b) lag2(b)

\* 6 pm 3 2 1 8 7 6

\* 7 pm 4 3 2 9 8 7

\* 8 pm 5 4 3 10 9 8

\*/

## 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)

})

}

## Spark中的DenseMatrix和RowMatrix

DenseMatrix是单节点的,是可序列化的,而RowMatrix不行.

也就是你可以构造一个RDD[DenseMatrix]但不能构造一个RDD[RowMatrix]

## Spark RDD中的zipWithIndex和sortWith等考虑到了分区的问题，也就是对全局sort和zipWithIndex

SortWith还需要验证

## Scala中ArrayBuffer等方法的+=和++=以及+=:和++=:

注意没有++,不像Array

++=加的是一个TraversableOnce即可

## Spark中top默认为降序,

Spark中top默认为降序,同时自带排序(内部实现的是takeOrdered), 一个误区是SortBy(升序).top可以实现升序取top, 其实sortBy排完序之后对top没有影响，top仍然会排序一次，此时仍然取的是降序的top，要想取升序，top后传入一个隐式的Ordering类top(4)(Ordering[U].reverse)或Ordering.by[((U, Int), Int)](f)

例如top(3)(Ordering.by[(String, Int), Int](\_.\_2))

## Breeze DenseMatrix的(::, k)方法

Breeze DenseMatrix的(::, k)方法，取得的虽然变为了DenseVector, 但调用data时仍然是DenseMatrix的data,DenseVector只是实现了对data的一个索引，要调用toArray才会得到真正了元素（此时元素才对应）

同样注意val ts\_p: SliceMatrix[Int, Int, Double] = lagData(::, (0 until K).map(k => index(k, p)))是个SliceMatrix，仍然是源Matrix加一个索引，并没有真正是一个对象，如果想要是一个全新的对象，也就是可以和其他相乘，应该加上toDenseMatrix.

## Spark中隐式转换及其在breeze. Linalg中的应用

### 隐式转换及其基本应用

## breeze.linalg中一些比较好的用法

mllib中的DenseMatrix[T]需要T，而在DenseMatrix [T]上面添加了Matrix，从而实现了对任意T类型（当然需要@special中的数据）都可以调用的方法，写函数时传递的类型直接是Matrix，不用根据T的类型进行多次声明。而breeze中不行，必须声明类型，这是breeze的弊端。breeze的优势在于它的函数编程用了大量的隐式类，虽然不好懂，但调用起来比较方便。

## Scala中的toInt的限制

不能超过2^31 - 1（2147483647 ），越界会自动进行位运算的补全

## Scala中NaN、None、null、nothing的用法

1. 这四个一个比一个更加抽象

NaN是一些数值类型的缺失对象，用于在计算时需要传递某类型但又有无效值或者空值的情况。

None是Option[T]类型的缺失对象，如果以Option封装某一类型，None可以用于无效或空值的情况

Null是引用类型的空值，可以表示集合类型，String等的空值

Nothing是任何对象的空值，一般只作为泛型时使用。如要新建一个空集合Array.empty[nothing]后面可以填入任何类型

## 控制流语句的使用

### 整体的感想

是计算机出身的程序员在控制流使用方面很擅长，数学出身的对高级函数和矩阵运算很擅长。为了弥补不足从今天起开始学习一些常用的控制流语句架构。

### While先加还是后加的问题

While(p < 200){

Arr += p

P +=1

}

While(p < 200){

P +=1

Arr += p

}

### While中i的步长不固定的问题

以线性缺失值补全为例

def fillLinear(values: BV[Double]): BDV[Double] = {

val result = values.copy.toArray

var i = 1

while (i < result.length - 1) {

val rangeStart = i

while (i < result.length - 1 && result(i).isNaN) {

i += 1

}

val before = result(rangeStart - 1)

val after = result(i)

if (i != rangeStart && !before.isNaN && !after.isNaN) {

val increment = (after - before) / (i - (rangeStart - 1))

for (j <- rangeStart until i) {

result(j) = result(j - 1) + increment

}

}

i += 1

}

new BDV[Double](result)

}

不过这种缺失值补全的时间复杂度过高（大约是O(n\*m)），还可以变为O（n）

## org.apache.commons.math3.distribution

### 求一些标准分布的分位数值

以正态分布为例

NormalDistribution normalDistribution = new NormalDistribution();

double difference = normalDistribution.inverseCumulativeProbability(1.0D - alpha) \* FastMath.sqrt(1.0D / (double)numberOfTrials \* mean \* (1.0D - mean));

## mutable.Map的+=问题

val categories = scala.collection.mutable.Map.empty[String, Double]

categories += tup

val categories = scala.collection.mutable.Map[String, Double]

categories += tup // not compile

## DataFrame列名的问题

1）DataFrame中的DataType和String类型交互的问题

DataType中有几个方法可以使用

1. 由string到DataType

def fromCaseClassString(string: String): DataType = CaseClassStringParser(string)

支持一下类型之间的转换

|"StringType" ^^^ StringType

| "FloatType" ^^^ FloatType

| "IntegerType" ^^^ IntegerType

| "ByteType" ^^^ ByteType

| "ShortType" ^^^ ShortType

| "DoubleType" ^^^ DoubleType

| "LongType" ^^^ LongType

| "BinaryType" ^^^ BinaryType

| "BooleanType" ^^^ BooleanType

| "DateType" ^^^ DateType

| "DecimalType()" ^^^ DecimalType.USER\_DEFAULT

| fixedDecimalType

| "TimestampType" ^^^ TimestampType

1. 由DataType到String

This.typeName

注意是lowerCase

## 实现类内参数变化的两个方法

1. 通过var s不断对s进行赋值
2. 在对象中创建var的变量进行更改，更为好看

class ColumnInfo(var name: String, var dataType: String){

def checkName(schema: StructType, paste: String = "\_"): this.type = {

var colName = this.name

while(schema.fieldNames contains colName){

colName += paste

}

this.name = colName

this

}

}

Class TimeColInfo(…) extends ColumnInfo(…)

3）不过要注意this.type用法

var newTimeCol: TimeColInfo = new TimeColInfo("timeCol", "long", Some("millisecond"))

newTimeCol = newTimeCol.checkName(data.schema) // 检查新建的时间列名是否存在于

如果改为：

class ColumnInfo(val name: String, val dataType: String){

def checkName(schema: StructType, paste: String = "\_"): ColumnInfo = {

var colName = this.name

while(schema.fieldNames contains colName){

colName += paste

}

New ColumnInfo(colName, val dataType)

}

}

var newTimeCol: TimeColInfo = new TimeColInfo("timeCol", "long", Some("millisecond"))

newTimeCol = newTimeCol.checkName(data.schema) //会认为是ColumnInfo而不是TimeColInfo

4）同时要注意var变量不能被覆盖

Error:(23, 32) overriding variable name in class ColumnInfo of type String;

variable name cannot override a mutable variable

class TimeColInfo(override var name: String,

## 可变对象



也就是说上述例子中可以写作

var newTimeCol: TimeColInfo = new TimeColInfo("timeCol", "long", Some("millisecond"))

newTimeCol.checkName(data.schema) //此时newTimeCol自己就已经发生了变化。而不用作为可变变量赋值

## Java. SimpleDateFormat牛逼的parse方法和JodaTime牛逼的toString方法

val timeFormat = new SimpleDateFormat("HH时mm分ss秒")

val timeStamp = timeFormat.parse("14时23分08秒").getTime

val timeFormat = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss")

val timeStamp = timeFormat.parse("2018-05-07 14:23:08").getTime

val dateTime = new DateTime(timeStamp)

val floorTime = dateTime.monthOfYear().roundFloorCopy()

println(floorTime.toString("yyyy-MM-dd HH:mm:ss"))

println(new DateTime(dateTime.getMillis - floorTime.getMillis).hourOfDay().roundFloorCopy().toString("dd日 HH时mm分ss秒"))

## scala继承中的变量问题

要点：继承一个类不能有声明的不可变变量

1. 在变量实例化参数中声明非private变量

缺点是必须为stable变量，不能变化，另外子类中必须继承该变量也必须为stable，限制了灵活性但增加了稳定性

Class A(val stable: Int){  
…

}

Class B(override val stable: Int, val newV: Double) extends A(stable){  
…

}

1. 声明private var变量或者在类内部声明一个var变量同时加入一个set方法

优点是类和子类都变成了可变变量,可以随时变更类内变量的值。缺点是有些不安全，需要在外部做一些判定。

Class A{

var mutable: Int = 0

def change (newInt: Int): this.type = {

this.mutable = new Int

this

}

}

Class B(val newV: Double) extends A{

…

}

达到如下效果

Val b = new B

b.change(2)

b.mutable // 2

## scala的定义this和apply方法的区别

最终区别在于

This

New A(param1, param2,…)

Apply

{New A}( param1, param2,…)

## 注意Floor并不等于整除加乘法

Math.floor(-2/3)结果为-1而-2/3结果为0，在时间分箱的时候碰到了这个不易察觉的坑

## Scala的尾递归的声明 scala.annotation.tailrec

尾递归即用即销，效果优于递归。

调用方式是

@ tailrec

Def rec() = {

…

…

…

rec

}

## Scala泛型的上界类型示例

abstract class GeneralizedLinearAlgorithm[M <: GeneralizedLinearModel]

extends Logging with Serializable {

def run(input: RDD[LabeledPoint]): M = {

…

}

}

比如logistic实现了LogisticWithSGDModel是GeneralizedLinearModel就可以输出