# 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>caseclassclass Bread(brand:String, price:Int)

definedclassclass 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>caseclassclass Bread(brand:String, price:Int)

definedclassclass 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](….)

只限于原生类型

## 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));

起始breeze.status中也有

## 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"))

## 注意Joda的DateTime

Val ts :Timestamp = …

New DateTime(ts.getTime) 是少8小时的，也就是DateTime认为1971-01-01 00:00:00是开始时间，早于其他的。

## 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{

varmutablemutable: 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

## 类中Multi load function不能有默认参数的问题

如class A{

Def a(param1: String, param2: String, param3: Int = 0)

Def a(param1: String) // 不会出现问题

Def a(param1: String, param2: String) // 不会出现问题

Def a(param1: String, param2: String, param3) // 会出现问题

}

## Case class长度超限肿么办

一般不能超限（22）

如果函数中需要超限，可以嵌套case class

## Breeze.linalg.DenseMatrix怎样实例化

它的设计参数有点冗余，实例化起来有点繁琐，需要约定一些规则。

val u = Array.range(0, 6)

println(u.mkString(","))

println(new BDM(3, 2, u, 0, 2, true))

println(new BDM(2, 3, u, 0, 2, false))

第一个参数为要生成矩阵的行数

第二个参数为要生成矩阵的列数

第三个参数为要生成矩阵的数据源，是个Array[Double]

第四个参数为Array起始录入元素的位置——起始没有录入元素，只是添加了索引，这里说是录入是为了形象一点

第五个参数为每录入多少个进入下一个行或者列（如果是非转置就是没录入多少个进入下一列，转置这是行）(这个参数有点冗余,可以设计为如果转置该参数等于第一个参数——行数，否则等于第二个参数——列数)

第六个参数为是否转置，转置就是按行录入，不转置就是按列录入.

另外也要注意他只是存的索引。因此mat.data是数据，但是按列排列的还是按行排列的仍然未知，还需要考虑mat.transpose。这样设计好处是增加了速度，不用直接每个元素遍历，可以解决一些大矩阵和稀疏矩阵的问题，缺点就是取data时容易掉坑里。例如“spark和breeze的DenseVector有以下特性，改变其data，则对应的Vector也变化”

## Spark.mllib.linalg和breeze.linalg不一致时报错不对的坑

Error:(86, 27) could not find implicit value for parameter op: breeze.linalg.operators.OpMulMatrix.Impl2[breeze.linalg.DenseMatrix[Double],org.apache.spark.mllib.linalg.Vector,That]

val a = Aq \* beta \* data // (num \* num) \* (num \* p) \* (p \* 1) = num \* 1维

其实根本原因在于把Spark.mllib.linalg的Vector当成了breeze.linalg的Vector。

## spark和breeze的DenseVector有以下特性，改变其data，则对应的Vector也变化

val weight: linalg.DenseVector = new linalg.DenseVector(Array(3.0, -3.0, 2.0, 0.1, 0.2, 0.1, 0.47, -0.61, 0.14))

weight.values(0) = 100.0

println(weight) // 也变为100

可见DenseVector虽然不能update（spark的DenseVector）但能通过update其data实现。

## 对象实例化中的可变变量的注意事项

1. 方法调用到可变变量

用的是新的可变变量

1. 方法调用到可变变量生成的变量

有问题用的是旧的可变变量生成的变量

1. lazy的可变变量生成的变量

## scala的注解

1. scala自带的注解

在scala的anotation的meta中

@Unchecked

一般是在模式匹配的时候用到的，告诉编译器有些地方不用"检查"了。如前所述，List[String @ unchecked]。



1. 自定义注解
2. 如何生成API文档

## Spark中feature的一些学习

1. row中不用get得到元素的方法

dataset.select($(inputCol)).map { case Row(v: Vector) => v }

1. 这个写法很炫——在aggregate中以类的形式进行

def fit(dataset: RDD[Vector]): IDFModel = {

val idf = dataset.treeAggregate(new IDF.DocumentFrequencyAggregator(

minDocFreq = minDocFreq))(

seqOp = (df, v) => df.add(v),

combOp = (df1, df2) => df1.merge(df2)

).idf()

new IDFModel(idf)

}

/\*\* Document frequency aggregator. \*/

class DocumentFrequencyAggregator(val minDocFreq: Int) extends Serializable {

/\*\* number of documents \*/

private var m = 0L //这里m进行了一个计数统计

/\*\* document frequency vector \*/

private var df: BDV[Long] = \_ //还可以这样写！！！

def this() = this(0)

/\*\* Adds a new document. \*/

def add(doc: Vector): this.type = { // 注意add和merge返回的都是this.type

if (isEmpty) {

df = BDV.zeros(doc.size)

}

doc match {

case SparseVector(size, indices, values) =>

val nnz = indices.size

var k = 0

while (k < nnz) {

if (values(k) > 0) {

df(indices(k)) += 1L

}

k += 1

}

case DenseVector(values) =>

val n = values.size

var j = 0

while (j < n) {

if (values(j) > 0.0) {

df(j) += 1L

}

j += 1

}

case other =>

throw new UnsupportedOperationException(

s"Only sparse and dense vectors are supported but got ${other.getClass}.")

}

m += 1L

this

}

/\*\* Merges another. \*/

def merge(other: DocumentFrequencyAggregator): this.type = {

if (!other.isEmpty) {

m += other.m

if (df == null) {

df = other.df.copy

} else {

df += other.df

}

}

this

}

private def isEmpty: Boolean = m == 0L

/\*\* Returns the current IDF vector. \*/

def idf(): Vector = {

if (isEmpty) {

throw new IllegalStateException("Haven't seen any document yet.")

}

val n = df.length

val inv = new Array[Double](n)

var j = 0

while (j < n) {

/\*

\* If the term is not present in the minimum

\* number of documents, set IDF to 0. This

\* will cause multiplication in IDFModel to

\* set TF-IDF to 0.

\*

\* Since arrays are initialized to 0 by default,

\* we just omit changing those entries.

\*/

if (df(j) >= minDocFreq) {

inv(j) = math.log((m + 1.0) / (df(j) + 1.0))

}

j += 1

}

Vectors.dense(inv)

}

}

}

1. 直接建立word -> index的映射以获得语料库，从而获得词频统计向量——非常容易栈溢出。

解决方式：选定一个很大的数，将“hash + 取非负模”作为这个词的index

## 超出akka.frame.size

org.apache.spark.SparkException: Job aborted due to stage failure: Serialized task 87:20 was 579652764 bytes, which exceeds max allowed: spark.akka.frameSize (134217728 bytes) - reserved (204800 bytes). Consider increasing spark.akka.frameSize or using broadcast variables for large values.

可能出现在：

1. 闭包传递的内容过大，超出了akka所对应的限制，exucter端
2. Collect过大，driver端

解决方法：

1. 调节akka.frameSize限制
2. 利用broadcast

Broadcast会将一个大的serializable的对象独立于Task发送，而在Task中仅仅存入该对象的引用。

1. 分次collect（如果是collect过大引起的话），或均匀分区后collect

## 避免超出akka.frameSize

**def** checkFrameSize(boxesNum: Long): Boolean = {  
**val** akkaFrameSize = util.*Try*(  
 data.sqlContext.sparkContext.getConf.getSizeAsMb("spark.akka.frameSize")).toOption  
**if** (akkaFrameSize.isDefined) {  
**val** size = (1 <<17).toLong \* akkaFrameSize.get // 17 = 20 - 3, a double = 2^3 Bytes  
size > boxesNum  
 } **else** {  
*println*("没有获得akka.frameSize的参数，未能判断等深或自定义的分箱边界是否超过结点间传输限制。")  
**true**}  
}

## 多态调整和协变

协变和多态类似，都是由子类型到基类型

1. 正常多态无法识别泛型

trait Car

class Volvo extends Car

case class Item[A](a: A)

val c1: Item[Car] = Item(new Volvo)

val c2: Item[Car] = new Item[Volvo](new Volvo) // not compile

println(c1)

println(c2) // not compile

1. 加入协变后可以

case class Item[+A](a: A)

val c1: Item[Car] = Item(new Volvo)

val c2: Item[Car] = new Item[Volvo](new Volvo) // compile

println(c1)

println(c2) //compile

case class Item[+A](a: A)

val c1: Item[Car] = Item(new Volvo)

val c2: Item[Car] = new Item[Volvo](new Volvo) // compile

println(c1)

println(c2) //compile

## 协变和逆变

一些规则：**一般的，函数声明的类型可以向上兼容但不能向下兼容。这里表述是自己的表述，向上兼容是指子类可以作为父类进行编译，不能向下兼容是指父类不能作为子类编译。**

示例：

1. 问题1

**class** Item[+A](**val** a: A){  
**def** test = **true**}  
**val** car1: Item[Car] = **new** Item(**new** Car)  
**val** volvo1: Item[Volvo] = **new** Item(**new** Volvo)  
**def** getCar(car: Item[Car]) = **true***println*(getCar(car1)) // 类型恰当  
*println*(getCar(volvo1)) // not compile， 需要Item[Volvo]但没法转为Item[Car], 多态这里没起作用  
// 解决方案 --协变

1. 解决方案——协变

**class** Item[+A](**val** a: A){  
**def** test = **true**}  
**val** car1: Item[Car] = **new** Item(**new** Car)  
**val** volvo1: Item[Volvo] = **new** Item(**new** Volvo)  
**def** getCar(car: Item[Car]) = **true***println*(getCar(car1)) // 类型恰当  
*println*(getCar(volvo1)) // compile， 需要Item[+A]中Volvo转为了父类型Car，子类Volvo向上兼容为基类Car，发生了协变

1. 问题2

**class** Item[A](**val** a: A){  
**def** test = **true**}  
**val** car1: Item[Car] = **new** Item(**new** Car)  
**val** volvo1: Item[Volvo] = **new** Item(**new** Volvo)  
  
**def** getVolvo(volvo: Item[Volvo]) = **true***println*(getVolvo(volvo1)) // 类型恰当  
*println*(getVolvo(car1)) // not compile， Item[Car]不是Item[Volvo]类型，基类不能兼容为子类。另外将Item变为协变也不行

1. 解决方案——逆变

**class** Item[-A](a: A){ // 注意此处a不能为val或var  
**def** test = **true**}  
**val** car1: Item[Car] = **new** Item(**new** Car)  
**val** volvo1: Item[Volvo] = **new** Item(**new** Volvo)  
  
**def** getVolvo(volvo: Item[Volvo]) = **true***println*(getVolvo(volvo1)) // 类型恰当  
*println*(getVolvo(car1)) // compiled，需要Item[+A]中Car转为了父类型Volvo，基类Car向下兼容为了子类Volvo，发生了逆变

注意事项：

1. 协变逆变都是单方向的，协变使得泛型能像多态一样可以向上兼容为父类型进行编译（有人也叫做向下兼容，只是角度问题），但反向不行。逆变可以使父类型作为子类型进行编译，但反向也不行。

## 上下界和ClassTag结合

今天利用上下界和ClassTag结合帮一个人解决了问题，自我感觉，帅呆了。

1. 原问题：



请问一下，这里怎么根据传入方法的类型设置返回值类型呢？

1. 解决

我觉得你的设计有点问题，既然外面需要判断一次类型为啥不直接写成两个函数呢，一个输出DF一个RDD。我觉得，如果不用泛型的话可以这样实现你要的需求：一种方法是以他们俩的父类Serializable作为输出类型；第二种是把RDD和DF装在某个类中作为变量在函数中操作。还有就是上面有些语法问题。

如果硬要用泛型的话，或许这样可以：

**import** org.apache.spark.rdd.RDD  
**import** org.apache.spark.sql.DataFrame  
  
**import** scala.reflect.ClassTag  
  
**val** df = *sqlc*.createDataFrame(*Seq*(  
 (0, "a"),  
(1, "b"),  
(2, "c")  
)).toDF("someId", "category")  
  
**def** getRdd(): RDD[String] = df.*rdd*.map(row => row.getAs[String](1))  
  
**def** getDataFrame(): DataFrame = df.select("category")  
  
**class** TwoTypeOutput[T <: Serializable : ClassTag]() {  
**def** generate(method: () =>T): T = {  
**type** method1 = () => RDD[String]  
**type** method2 = () => DataFrame  
  
 method **match** {  
**case** mth1: method1 => mth1()  
**case** mth2: method2 => mth2()  
**case** \_ =>**throw new** Exception("不支持其他类型的函数")  
 }  
 }  
}  
  
**new** TwoTypeOutput[DataFrame]().generate(getDataFrame).show(3)  
**new** TwoTypeOutput[RDD[String]]().generate(getRdd).take(3).foreach(*println*)

## scala的Double.NaN

scala的Double.NaN是一个特殊的类型，可以作为标记缺失值使用。同时也是可以比较大小的，虽然都是false。一下代码全是false。

*println*(Double.*NaN* <0.5)  
*println*(Double.*NaN* >0.5)  
*println*(Double.*NaN* == 0.5)  
*println*(Double.*NaN* == **null**)

## scala中String的toInt还是比较严格的

会先parseInt在如果给出“1.2”转为Int会报错。java.lang.NumberFormatException: For input string: "-1.2"。

**val**uu = "1E2".toDouble

不会报错但toInt不行。

## Array.distinct源码中对控制语句的使用

def distinct: Repr = {

val b = newBuilder

val seen = mutable.HashSet[A]()

for (x <- this) {

if (!seen(x)) {

b += x

seen += x

}

}

b.result

}

**使用了两个集合A，B，其中A参与了if守卫控制条件**

## Spark中aggregateByKey不能使用Array类型作为key

**Caused by: org.apache.spark.SparkException: Cannot use map-side combining with array keys.**

val rdd = df.rdd.map(r => {

**val key: Array[String]** = keyArray.map(name => r.get(getIndex(name)).asInstanceOf[String])

val typeName = r.get(getIndex(typeCol)).asInstanceOf[String]

val count = r.get(getIndex(countCol)).asInstanceOf[Long]

(key, (typeName, count))

}).aggregateByKey((Array.empty[(String, Long)], 0L))(

seqOp = {

case ((arr, sum), (name, count)) =>

(arr :+ (name, count), sum + count)

},

combOp = {

case ((arr1, sum1), (arr2, sum2)) => (arr1 ++ arr2, sum1 + sum2)

}

).flatMapValues {

所以虽然low，mkString为String可能还是需要的。

## Scala中\_的用法

1. 原理
2. 正常的替代就不说了
3. 二元替代

def setA(name: String, age: Int, check: (String, Int) => Boolean = **(\_, \_) => true**) = {

require(check(name, age), "error")

println(name)

println(age)

}

setA("John", 18)

## Int不是java.lang.object的子类

Any和AnyVal才是Int的父类型

Java.lang.object是AnyRef的父类型

HashMap[String, Object]始终无法将1加进去，还以为是泛型的问题（怀疑到set函数，自己定义了一个set函数，还调换了一下顺序， key和object的），最后才发现不是。

## Spark中进行mapPartition是Iterator另一种写法，认为很炫

val sentences: RDD[Array[Int]] = words.mapPartitions { **iter** =>

new Iterator[Array[Int]] {

**def hasNext**: Boolean = **iter.hasNext**

**def next()**: Array[Int] = {

val sentence = ArrayBuilder.make[Int]

var sentenceLength = 0

while (iter.hasNext && sentenceLength < MAX\_SENTENCE\_LENGTH) {

val word = bcVocabHash.value.get(**iter.next()**)

word match {

case Some(w) =>

sentence += w

sentenceLength += 1

case None =>

}

}

sentence.result()

}

}

}

## Spark.ml.Bucketizer中通过二分查找数据所在区间

private[feature] def binarySearchForBuckets(splits: Array[Double], feature: Double): Double = {

if (feature == splits.last) {

splits.length - 2

} else {

val idx = **ju.Arrays.binarySearch(splits, feature)**

if (idx >= 0) {

idx

} else {

val insertPos = -idx - 1

if (insertPos == 0 || insertPos == splits.length) {

throw new SparkException(s"Feature value $feature out of Bucketizer bounds" +

s" [${splits.head}, ${splits.last}]. Check your features, or loosen " +

s"the lower/upper bound constraints.")

} else {

insertPos - 1

}

}

}

}

## Scala中string.split的方法

很奇怪，需要看一下：

val vocabSizeString = "2^18"

vocabSizeString.split("^").foreach(println) // not work

vocabSizeString.split(‘^’).foreach(println) // work

## spark中SparseVector、DenseVector的最大长度

两者的最大长度一样，都是2147483647，即Int的最大值。小于等于这个值是能够运行的，比如下面的代码。超出这个值会抛出异常，但只是Int导致的异常：integer number too large.

因此SparseVector和DenseVector的区别只是存储方式和运算机制不同。

1. 如果一个记录超过50%的记录为0值或空值，SparseVector占用空间更小；

具体的计算方式是：

A dense vector needs 8 \* size + 8 bytes, while a sparse vector needs 12 \* nnz + 20 bytes.

DenseVector(Array()) 一个Double 64位，8字节，一个Array占8字节，一个Vector对象8字节，总共 8\*size + 8 + 8 + 各种引用等

SparseVector(size, indices, values) 一个 Int 32位， 4字节，对应一个Double64位8字节，一个size Int类型4字节，两个Array的引用各8字节，一个Vector对象8字节，总共(4+8)\*nonZeroNum + 4 + 8\*2 + 8。

因此Vector有种方法叫做compressed

def compressed: Vector = {

val nnz = numNonzeros

// A dense vector needs 8 \* size + 8 bytes, while a sparse vector needs 12 \* nnz + 20 bytes.

if (1.5 \* (nnz + 1.0) < size) {

toSparse

} else {

toDense

}

}

1. 如果一个记录超过50%的记录为0值或空值，SparseVector进行运算更快捷；
2. 它们在最终存储数据的大小上没有区别，都是Int.MaxValue。

import org.apache.spark.mllib.linalg.Vectors

import org.apache.spark.sql.DataFrame

val data = Seq(

(-1.0, Vectors.dense(Array(1.0, 3.0, 5.0))),

(0.0, Vectors.dense(Array(1.0))),

(2.0, Vectors.sparse(2147483647, Array(1, 3), Array(0.0, 1.0)))

)

val dataFrame = sqlc.createDataFrame(data).toDF("features", "notEqualLengthVector")

dataFrame.show()

## 多列的udf如何写

**VectorAssembler中的源码：**

// Data transformation.  
**val** assembleFunc = *udf* { r: Row =>  
 VectorAssembler.*assemble*(r.toSeq: \_\*)  
}  
**val** args = $(*inputCols*).map { c =>  
 schema(c).dataType **match** {  
**case** DoubleType => dataset(c)  
**case** \_: VectorUDT => dataset(c)  
**case** \_: NumericType | BooleanType => dataset(c).cast(DoubleType).as(s"**$**{c}\_double\_**$**uid")  
 }  
}  
  
dataset.select(*col*("\*"), assembleFunc(*struct*(args : \_\*)).as($(*outputCol*), metadata))

## SparkSQL中列名的敏感词问题

有些特殊符合或者敏感词会被parser解析为语句或者一些语法。

比如“first”，比如“$”

可以加反引号“`”，对于有反引号的列名可以加两个反引号“``”.

在源码中org.apache.spark.sql.catalyst.parser.PostProcessor中有这些匹配模式。

scala> df.select(expr("```x```.a"), expr("```x```")("b")).printSchema

## map + case中省略match

data.select(colName).*rdd*.map {  
**case** *Row*(sv: SparseVector) => sv.toDense  
**case** *Row*(dv: DenseVector) => dv  
**case** \_ =>**throw new** Exception("没有识别到vector类型")  
}

真牛逼

## 列名中有点，出现outOFindexException -1

描述：原数据中有引号“””，将列名中的引号换为“.”，报错

解析：

是由于将.解析的问题，加入反引号“`”，

发现drop加反引号没效果，于是不能selectExpr(“\*”, “`oldName` as `newName`”).drop(“`oldNmae`”)

**val** exprSQL = **for** (sField <- schema) **yield**{  
  
**if** (sField.dataType == StringType) {  
**val** colName = sField.name  
*println*(i)  
**val** strTemp = **try**{  
**if** (i == 1) {  
**val** head = colName.getBytes()  
**if** (head(0) == (-17) && head(1) == (-69) && head(2) == (-65)) head.drop(3).map(\_.toChar).mkString **else** colName  
 } **else** colName  
 }**catch**{  
**case** e: Exception =>**throw new** Exception(s"在**$**colName 处失败，此时i = **$**i, 错误**$**{e.getMessage}")  
 }  
// val newName=colName.substring(colName.indexOf("\"")+1,colName.lastIndexOf("\""))  
 // val newName = colName.trim.slice(1, colName.length - 1)  
**val** newName = **try**{  
 func(strTemp.trim)  
 }**catch** {  
**case** e: Exception =>**throw new** Exception(s"在此处失败，错误在**$**{strTemp.trim},错误**$**{e.getMessage}")  
 }  
  
 (colName, newName)  
 }**else**{  
 ("f", "f")  
 }  
}  
  
inputDf = inputDf.selectExpr(exprSQL.map{**case** (colN, newN) =>"`"+ colN + "`" + " as " + "`" + newN + "`"}:\_\*)  
  
inputDf.show

## spark SQL中过滤出非索引类的数值

需要为大于等于0的自然数，可以是double类型，但数值必须相等。

## SparkSQL对sql树中count的优化

1. 如果执行count语句，只执行触及到count的语法树（一般也就是生成count的表的语句，因为count操作collect的withCallback为false，不会触发df.queryExecution.executedPlan的遍历执行），如下面的例子。
2. 如果执行collect类操作，执行所有的语法树。
3. 例子如下：
4. **val** rawDataFrame = *sqlc*.createDataFrame(  
   *Array*(  
    (0.0, 1, 32f, Vectors.*dense*(*Array*(1, 2, 0).map(\_.toDouble)), *Array*(*Array*("s"), *Array*("d"))),  
   (1.2, 1, 27f, Vectors.*dense*(*Array*(0, 1, 1).map(\_.toDouble)), *Array*(*Array*("s"), *Array*("d"))),  
   (1.0, 1, 22f, Vectors.*dense*(*Array*(0, 1, 0).map(\_.toDouble)), *Array*(*Array*("s"), *Array*("d"))),  
   (1.0, 1, 25f, Vectors.*dense*(*Array*(1, 1, 1).map(\_.toDouble)), *Array*(*Array*("s"), *Array*("d"))),  
   (-0.1, 1, 23f, Vectors.*dense*(*Array*(0, 2, 1).map(\_.toDouble)), *Array*(*Array*("s"), *Array*("d")))  
    )  
   ).toDF("label", "intType", "floatType", "feature", "arrayArray")  
     
     
   **val** inputCol = "label"  
   rawDataFrame.select(inputCol).filter(s"**$**inputCol != floor(**$**inputCol) or **$**inputCol< 0").show()  
     
   **val** transform = NullableFunctions.*udf*((d: Double) =>**if**(d <0) **throw new** Exception("数值错误") **else** 1.0)  
     
   **import** org.apache.spark.sql.functions.*col  
     
   println*(rawDataFrame.withColumn("newName", transform(*col*("label"))).count()) // 不报错  
     
   rawDataFrame.withColumn("newName", transform(*col*("label"))).show() // 报错
5. 那么问题来了，超大数据的转换，如果我要测transform操作的性能，如何才能测出来了？

## 如何创建一个Exception

Spark.sql. org.apache.spark.sql.types.DataTypeParser提供了一个不错的思路

*/\*\* The exception thrown from the* [[*DataTypeParser*]]*. \*/***private**[sql] **class** DataTypeException(message: String) **extends** Exception(message)

## Scala对于package object中对伴生对象不太支持

我在一个package object中定义了一个伴生对象，结果编译出现版本问题

Compile 2.10.6

Compile 2.10.6

诸如此类一堆

后来放到一个目录结构的package中，没有出现问题。

## SparkSQL中Array<String>类型的列在Row中是以WrapperArray形式存在的

存储的值是：WrappedArray(1.0)以1.0为例

存储的类型是：scala.collection.mutable.WrappedArray$ofRef

因此：

1）注意匹配的时候要用Seq<String>，而不要用Array.

2）它也区分不了Seq[String]还是Seq[Double]的，都match的上，所以一下代码是有问题的

Df.rdd.match{

row =>

row.get(0) match {

case Seq[String] =>

case Seq[Double] =>

}

} // not really work

需要通过schema判定。

## ArrayBuilder的用法

val values = ArrayBuilder.make[Double]

values += 1.0

values.result

好处，可变对象节省空间

## Spark上一个分区中的数据在下一个分区中用到怎么办？

1. 可以将数据放入buffer中，aggregate以buffer对象aggregate

Aggregate是以partitionid的形式进行的吗？

1. 如果用到的数据条目一定，可以先取每个分区的最后若干个，collect，再mapPartitionWithIndex，这个使用范围比3）广
2. 如果用到的数据条目一定，可以zipWithIndex，然后一个减掉1，join。这个适用于上下文联系的情况
3. 可以遍历分区数，对每个分区collect，获得最后一个元素，用于下一个分区的进行。
4. 交叠分区,设计一个重分区机制,将两个分区交叠用到了备份两份,分别发到不同分区中

## Spark添加自增列

rawDataFrame.select(*col*("\*"), *monotonicallyIncreasingId*().as(idCol))

可惜该自增列不是连续的id，每个分区内是连续的，但分区外不是。



## 一个想法——结点间数据传输

利用join实现分区所在结点间的数据传输

## Spark rdd直接进行runJob

**val** rdd: RDD[Int] = *sc*.parallelize(Array.*range*(0, 200), 5)  
**val** lth = rdd.partitions.length  
**val** u = *sc*.runJob(rdd, (it: Iterator[Int]) => it.take(5).toArray, *Range*(lth - 2, lth))  
 u.foreach(arr =>  
*println*(arr.mkString(","))  
 )

## cala中Map后面加mapValue无法序列化？

**//** KLInfoMethod

**val** sidesElements: collection.Map[Int, (LagInfoForPartition, Int)] = collection.Map(0 -> (*LagInfoForPartition*(*Array*(0.0), *Array*(1.0)), 50))  
sidesElements.foreach(*println*)  
  
**val** headTailInfo: LagInfo = *sc*.broadcast(**new** LagInfo(sidesElements.mapValues(\_.\_1), sidesElements.mapValues(\_.\_2), range, range)).value

// 报错

object not serializable (class: scala.collection.immutable.MapLike$$anon$2, value: Map(0 -> LagInfoForPartition([D@1acd2a14,[D@5432be12)))

//解决

**val**headTailInfo: LagInfo = *sc*.broadcast(**new** LagInfo(sidesElements.mapValues(\_.\_1).map(*identity*), sidesElements.mapValues(\_.\_2).map(*identity*), range, range)).value

// mapValue后面再加一个map，将其由MappedValues变为Map类型

## scala中Iterator进行操作后发生变化Map后面加mapValue无法序列化？

**val** iter: Iterator[Int] = Array.*range*(0, 10).toIterator  
*println*("length1", iter.length) // (length,10)  
*println*("length2", iter.length) // (length2,0)  
  
**val** iter2 = Array.*range*(0, 10).toIterator  
  
*println*("try1", iter2.toArray.mkString(",")) // (try1,0,1,2,3,4,5,6,7,8,9)  
*println*("try2", iter2.toArray.mkString(",")) // (try2,)

这是一个非常容易入坑的变化，因此注意，遇到iterator尽量不要转为数组等进行操作而通过while循环进行操作，或者先一次性转为Array等类型，再进行操作**//** KLInfoMethod

**val** sidesElements: collection.Map[Int, (LagInfoForPartition, Int)] = collection.Map(0 -> (*LagInfoForPartition*(*Array*(0.0), *Array*(1.0)), 50))  
sidesElements.foreach(*println*)  
  
**val** headTailInfo: LagInfo = *sc*.broadcast(**new** LagInfo(sidesElements.mapValues(\_.\_1), sidesElements.mapValues(\_.\_2), range, range)).value

// 报错

object not serializable (class: scala.collection.immutable.MapLike$$anon$2, value: Map(0 -> LagInfoForPartition([D@1acd2a14,[D@5432be12)))

//解决

**val**headTailInfo: LagInfo = *sc*.broadcast(**new** LagInfo(sidesElements.mapValues(\_.\_1).map(*identity*), sidesElements.mapValues(\_.\_2).map(*identity*), range, range)).value

// mapValue后面再加一个map，将其由MappedValues变为Map类型

## 遇到一个不熟悉的问题java.lang.ClassCastException

网上说这是父类不能作为子类出现

不过我的代码里并没有出现这样的问题，我的代码本地能够运行但平台http://192.168.11.102:6091/app/#/home运行不了，我的代码里只有一个case class里面有一些方法能够生成自身类型，可能这就是问题，最后有人升级了一下平台我的代码竟然又能运行了。

是因为平台脚本不支持，但jar中支持。

## 将类型像示例一样进行匹配操作

参考源码

目录：

org.apache.spark.sql.catalyst.ScalaReflection

*/\*\* Returns a catalyst DataType and its nullability for the given Scala Type using reflection. \*/***def** schemaFor(tpe: `Type`): Schema = ScalaReflectionLock.synchronized {  
**val** className = getClassNameFromType(tpe)  
 tpe **match** {  
**case** t **if** Utils.*classIsLoadable*(className) &&  
 Utils.*classForName*(className).isAnnotationPresent(*classOf*[SQLUserDefinedType]) =>  
// Note: We check for classIsLoadable above since Utils.classForName uses Java reflection,  
 // whereas className is from Scala reflection. This can make it hard to find classes  
 // in some cases, such as when a class is enclosed in an object (in which case  
 // Java appends a '$' to the object name but Scala does not).  
**val** udt = Utils.*classForName*(className)  
 .getAnnotation(*classOf*[SQLUserDefinedType]).udt().newInstance()  
*Schema*(udt, nullable = **true**)  
**case** t **if** t <:< localTypeOf[Option[\_]] =>  
**val** TypeRef(\_, \_, Seq(optType)) = t  
*Schema*(schemaFor(optType).dataType, nullable = **true**)  
**case** t **if** t <:< localTypeOf[Array[Byte]] =>*Schema*(BinaryType, nullable = **true**)  
**case** t **if** t <:< localTypeOf[Array[\_]] =>  
**val** TypeRef(\_, \_, Seq(elementType)) = t  
**val** *Schema*(dataType, nullable) = schemaFor(elementType)  
*Schema*(*ArrayType*(dataType, containsNull = nullable), nullable = **true**)  
**case** t **if** t <:< localTypeOf[Seq[\_]] =>  
**val** TypeRef(\_, \_, Seq(elementType)) = t  
**val** *Schema*(dataType, nullable) = schemaFor(elementType)  
*Schema*(*ArrayType*(dataType, containsNull = nullable), nullable = **true**)  
**case** t **if** t <:< localTypeOf[Map[\_, \_]] =>  
**val** TypeRef(\_, \_, Seq(keyType, valueType)) = t  
**val** *Schema*(valueDataType, valueNullable) = schemaFor(valueType)  
*Schema*(*MapType*(schemaFor(keyType).dataType,  
valueDataType, valueContainsNull = valueNullable), nullable = **true**)  
**case** t **if** t <:< localTypeOf[Product] =>  
**val** formalTypeArgs = t.typeSymbol.asClass.typeParams  
**val** TypeRef(\_, \_, actualTypeArgs) = t  
**val** constructorSymbol = t.member(nme.CONSTRUCTOR)  
**val** params = **if** (constructorSymbol.isMethod) {  
 constructorSymbol.asMethod.paramss  
 } **else** {  
// Find the primary constructor, and use its parameter ordering.  
**val**primaryConstructorSymbol: Option[Symbol] = constructorSymbol.asTerm.alternatives.find(  
 s => s.isMethod && s.asMethod.isPrimaryConstructor)  
**if** (primaryConstructorSymbol.isEmpty) {  
 sys.*error*("Internal SQL error: Product object did not have a primary constructor.")  
 } **else** {  
 primaryConstructorSymbol.get.asMethod.paramss  
 }  
 }  
*Schema*(*StructType*(  
 params.head.map { p =>  
**val** *Schema*(dataType, nullable) =  
 schemaFor(p.typeSignature.substituteTypes(formalTypeArgs, actualTypeArgs))  
*StructField*(p.name.toString, dataType, nullable)  
 }), nullable = **true**)  
**case** t **if** t <:< localTypeOf[String] =>*Schema*(StringType, nullable = **true**)  
**case** t **if** t <:< localTypeOf[java.sql.Timestamp] =>*Schema*(TimestampType, nullable = **true**)  
**case** t **if** t <:< localTypeOf[java.sql.Date] =>*Schema*(DateType, nullable = **true**)  
**case** t **if** t <:< localTypeOf[BigDecimal] =>*Schema*(DecimalType.*SYSTEM\_DEFAULT*, nullable = **true**)  
**case** t **if** t <:< localTypeOf[java.math.BigDecimal] =>  
*Schema*(DecimalType.*SYSTEM\_DEFAULT*, nullable = **true**)  
**case** t **if** t <:< localTypeOf[Decimal] =>*Schema*(DecimalType.*SYSTEM\_DEFAULT*, nullable = **true**)  
**case** t **if** t <:< localTypeOf[java.lang.Integer] =>*Schema*(IntegerType, nullable = **true**)  
**case** t **if** t <:< localTypeOf[java.lang.Long] =>*Schema*(LongType, nullable = **true**)  
**case** t **if** t <:< localTypeOf[java.lang.Double] =>*Schema*(DoubleType, nullable = **true**)  
**case** t **if** t <:< localTypeOf[java.lang.Float] =>*Schema*(FloatType, nullable = **true**)  
**case** t **if** t <:< localTypeOf[java.lang.Short] =>*Schema*(ShortType, nullable = **true**)  
**case** t **if** t <:< localTypeOf[java.lang.Byte] =>*Schema*(ByteType, nullable = **true**)  
**case** t **if** t <:< localTypeOf[java.lang.Boolean] =>*Schema*(BooleanType, nullable = **true**)  
**case** t **if** t <:< definitions.IntTpe =>*Schema*(IntegerType, nullable = **false**)  
**case** t **if** t <:< definitions.LongTpe =>*Schema*(LongType, nullable = **false**)  
**case** t **if** t <:< definitions.DoubleTpe =>*Schema*(DoubleType, nullable = **false**)  
**case** t **if** t <:< definitions.FloatTpe =>*Schema*(FloatType, nullable = **false**)  
**case** t **if** t <:< definitions.ShortTpe =>*Schema*(ShortType, nullable = **false**)  
**case** t **if** t <:< definitions.ByteTpe =>*Schema*(ByteType, nullable = **false**)  
**case** t **if** t <:< definitions.BooleanTpe =>*Schema*(BooleanType, nullable = **false**)  
**case** other =>  
**throw new** UnsupportedOperationException(s"Schema for type **$**other is not supported")  
 }  
}

用的地方：

**def** udf[RT: TypeTag, A1: TypeTag](f: Function1[A1, RT]): UserDefinedFunction = {  
**val** inputTypes = *Try*(ScalaReflection.schemaFor(typeTag[A1]).dataType :: *Nil*).getOrElse(*Nil*)  
*UserDefinedFunction*(f, ScalaReflection.schemaFor(typeTag[RT]).dataType, inputTypes)  
}

## spark自定义分区

spark提供自定义分区

abstract class Partitioner extends Serializable {

def numPartitions: Int

def getPartition(key: Any): Int

}

**def numPartitions: Int**：这个方法需要返回你想要创建分区的个数；

**def getPartition(key: Any): Int**：这个函数需要对输入的key做计算，然后返回该key的分区ID，范围一定是0到numPartitions-1；

**equals()**：这个是Java标准的判断相等的函数，之所以要求用户实现这个函数是因为Spark内部会比较两个RDD的分区是否一样。

## Spark自定义分区解决交叠分区问题

## Scala中yyyy-MM-dd'T'HH:mm:ssXXX格式时间的解析

**val** sdf1 = **new** SimpleDateFormat("yyyy-MM-dd'T'HH:mm:ssXXX")  
**val** sdf2 = **new** SimpleDateFormat("yyyy-MM-dd HH:mm:ss")  
**val** date = sdf1.parse("2018-01-22T09:12:43Z")  
*println*(sdf2.format(date))  
  
**val** time1 = "2018-01-22T09:12:43Z"  
**val** dt = **new** DateTime(**new** java.text.SimpleDateFormat("yyyy-MM-dd'T'HH:mm:ssXXX").parse(time1).getTime)  
*println*(dt)

试了下用“yyyy-MM-dd'T'HH:mm:ssZ”反而没有成功，但明明写的Z代表的是UTC时区的啊。

## Scala中正则表达式

定义的正则表达式其实是个函数（或者说类），需要加参数。

**val** dataType = "integer"  
  
**val** pattern = "(?i)(int|integer)".r  
dataType **match** {  
**case** pattern(\_) =>*println*("this is int type")  
**case** "double" =>*println*("this is double type")  
}

## spark性能优化之reduceByKey+Partitioner



## Spark coalesce不好用

shuffle=false时只是将大分区变为小分区，而且必须变，shuffle为true时可以小变大。因此无法自动将空分区搞出来，除非你大致知道数据量最多为10个分区合适，否则会产生不必要的shuffle。

**val** rdd2 = rdd.filter(dv => dv.data(0) != 1)  
 .coalesce(1, **false**)  
*println*("-"\*80)  
rdd2.foreachPartition(iter =>*println*(iter.mkString(",")))  
  
rdd2.partitions.map(\_.index).foreach(*println*)

## spark中runJob需要重新广播或者重新赋值然后闭包传递

方式1:val s =>闭包传递到各个分区 => runJob调用 // 这样调用的是最开始传递的值

方式2: var s =>闭包传递到各个分区 => runJob调用 => s改变 => runJob调用 // 这样调用的是内部的值

## Spark中shuffle的保序操作

### 问题：

一个partitionBy后的ShuffleRDD进行下一步操作，连续执行两次得到不同的结果。猜想是shuffleRDD并没有保序。

val rdd2 = partitionMaterial.partitionBy(new OverLapPartitioner(4))

rdd2.mapPartitionsWithIndex {

case (index, iter) =>

Iterator((index, Some(iter.toArray.head)))

}.collect().foreach(println)

println("-"\*80)

rdd2.mapPartitionsWithIndex {

case (index, iter) =>

Iterator((index, Some(iter.toArray.head)))

}.collect().foreach(println)

**第一次**

**(0,Some((8589934591,G)))**

**(1,Some((17179869183,N)))**

**(2,Some((17179869184,N)))**

**(3,Some((25769803776,T)))**

**--------------------------------------------------------------**

**第二次**

**(0,Some((8589934591,G)))**

**(1,Some((8589934592,G)))**

**(2,Some((17179869184,N)))**

**(3,Some((25769803776,T)))**

执行十次，绝大部分顺序都不同

### 验证：

为了验证，改为cache数据，此时不继续计算上一个stage（这个stage中shuffle是partitionBy操作，是无序的）而直接取数据。

执行十次，每次顺序一样。

### 分析：



因此shuffle后的shuffleRDD每个分片中的数据顺序由shuffle这个stage中的shuffle reader和shuffer writer的执行先后顺序有关。而partitionBy先后顺序是受资源状态，数据分布是否在一个worker上等因素影响的，也就是执行顺序是受随机因素影响的。

因此rdd2.partionBy执行两次，数据分区内的顺序很可能不同。注意这里可能更不易发现：

**Val rdd2 = rdd1.partitionBy()**

**Rdd2.mapPartion().collect() // 执行第一次**

**Rdd2.mapPartion().collect() // 执行第一次**

### 扩展：

什么操作时保序的，也就是不收shuffle过程影响的？

1. 首先map类操作是:map/filter/flatMap
2. sortBy也是
3. sc.textFile，sc.parallelize也是
4. coalesce(shuffle=false)是
5. countByKey
6. 最后一些action操作也是保序的：show, collect,count,runJob

哪些不是？

1. partitionBy
2. join
3. cogroup
4. reduceByKey
5. groupByKey

待验证：

1. limit

## spark sql createDataFrame时分区数200



创建方式

*sqlc*.createDataFrame(seq).toDF("id", "dayTime", "x", "y", "z")

## scala元组赋值变量名不能大写



大写时表示模式匹配。

## 如何保证RDD没有空分区

目前应该无法保证，因为coalesce也无法保证，即使你给他的是刚好的非空分区数目，他仍然可能弄出非空分区，因为它的步骤不是先去空分区再拉数据，而是选好分区id然后拉数据。

## Monoid并行算法框架

Scala有一个框架monoid（翻译：幺半群）可以进行并行算法的开发。其中的并行化函数需要满足：结合特性（associativity），与恒等值（identity）。

满足这两个特性后，可以进而满足左折叠和右折叠的相等（如foldLeft和foldRight），交换律。等等。

## Lazy的另一个好处

Lazy还有一个天然的好处，在类内部变量的传递上

**class** Lzy(mp: Map[String, Int]) {  
**def** rdd(key: String): Option[Int] = mp.get(key)  
}  
**var** map1 = *Map*[String, Int]()  
**lazy val** z = **new** Lzy(map1)  
map1 += ("a" ->10)  
*println*("a", z.rdd("a")) // 拿到，因为此时z没有被实例化  
map1 += ("b" ->20)  
*println*("b", z.rdd("b")) // 拿不到，因为此时z被实例化了已经

## spark broadcast和闭包的区别

1. broadcast不收akkaFrameSize限制
2. 开发过程中，会遇到需要在算子函数中使用外部变量的场景（尤其是大变量，比如100M以上的大集合），那么此时就应该使用Spark的广播（Broadcast）功能来提升性能。广播后的变量，会保证每个Executor的内存中，只驻留一份变量副本，而Executor中的task执行时共享该Executor中的那份变量副本

## Spark中的快速HashMap

**import**scala.reflect.ClassTag  
  
*/\*\*  
 \* A fast hash map implementation for nullable keys. This hash map supports insertions and updates,  
 \* but not deletions. This map is about 5X faster than java.util.HashMap, while using much less  
 \* space overhead.  
 \*  
 \* Under the hood, it uses our OpenHashSet implementation.  
 \*  
 \* NOTE: when using numeric type as the value type, the user of this class should be careful to  
 \* distinguish between the 0/0.0/0L and non-exist value  
 \*/***private**[spark]  
**class** OpenHashMap[K : ClassTag, @specialized(Long, Int, Double) V: ClassTag](

注意：0这里貌似是被作为了空值。

## Scala反射

**object**A {  
**def** fun1(a: String, b: Int): String = "success, the result is: " + "a, " + b  
  
**def** fun2(): Unit = *print*("fun2")  
  
**val** *t* = 1  
}

**val** a = A  
  
**val** classA = Class.*forName*("com.zzjz.deepinsight.core.MBLCYC.A")  
**val** method = classA.getDeclaredMethod("fun1", *classOf*[String], *classOf*[Int])  
**val** res = method.invoke(classA, "5", **new** Integer(0)) // 如果是Int的话,要用new Integer(5)  
  
*println*(res)  
  
**val** arr: Array[Any] = *Array*(a, 0)  
**val** res2 = method.invoke(arr(0), "5", **new** Integer(1))  
*println*(res2)

/\*  
如果类A如下定义:  
class A(a:String)  
则需要在反射时候添加构造器  
val cons = classA.getConstructors  
val newInst = cons(0).newInstance("你好")  
method.invoke(newInst, "哈哈")  
如果在构造其中不需要输入参数则  
 class A  
 method.invoke(classA.newInstance(), "5", new Integer(1))  
\*/

## SimpleDateFormat有识别上界

*println*(Long.*MinValue* >>1)  
*println*(**new** SimpleDateFormat("yyyy-MM-dd HH:mm:ss").format(**new** Date(-45478727387904L)))

>>-4611686018427387904

>>0528-10-31 04:10:12

## scala中Long和Int比较有时好像会有问题

## spark rdd进行累加出的一个糗的错误

rdd.reduce((v1, v2) => if(v1 > 2) 1 else 0 + if(v2 > 2) 1 else 0)想求rdd中大于2的数的个数

结果发现不是0就是1，一直找错误，最后发现reduce的问题，recduce中两个元素有一个是递归的结果。。。

## java/scala时间解析基本规则

### 模式说明

　d 月中的某一天。一位数的日期没有前导零。

　dd 月中的某一天。一位数的日期有一个前导零。

　ddd 周中某天的缩写名称，在 AbbreviatedDayNames 中定义。

　dddd 周中某天的完整名称，在 DayNames 中定义。

　M 月份数字。一位数的月份没有前导零。

　MM 月份数字。一位数的月份有一个前导零。

　MMM 月份的缩写名称，在 AbbreviatedMonthNames 中定义。

　MMMM 月份的完整名称，在 MonthNames 中定义。

　y 不包含纪元的年份。如果不包含纪元的年份小于 10，则显示不具有前导零的年份。

　yy 不包含纪元的年份。如果不包含纪元的年份小于 10，则显示具有前导零的年份。

　yyyy 包括纪元的四位数的年份。

　gg 时期或纪元。如果要设置格式的日期不具有关联的时期或纪元字符串，则忽略该模式。

　h 12 小时制的小时。一位数的小时数没有前导零。

　hh 12 小时制的小时。一位数的小时数有前导零。

　H 24 小时制的小时。一位数的小时数没有前导零。

　HH 24 小时制的小时。一位数的小时数有前导零。

　m 分钟。一位数的分钟数没有前导零。

　mm 分钟。一位数的分钟数有一个前导零。

　s 秒。一位数的秒数没有前导零。

　ss 秒。一位数的秒数有一个前导零。

　f 秒的小数精度为一位。其余数字被截断。

　ff 秒的小数精度为两位。其余数字被截断。

　fff 秒的小数精度为三位。其余数字被截断。

　ffff 秒的小数精度为四位。其余数字被截断。

　fffff 秒的小数精度为五位。其余数字被截断。

　ffffff 秒的小数精度为六位。其余数字被截断。

　fffffff 秒的小数精度为七位。其余数字被截断。

　t 在 AMDesignator 或 PMDesignator 中定义的 AM/PM 指示项的第一个字符（如果存在）。

　tt 在 AMDesignator 或 PMDesignator 中定义的 AM/PM 指示项（如果存在）。

　z 时区偏移量（“+”或“-”后面仅跟小时）。一位数的小时数没有前导零。例如，太平洋标准时间是“-8”。

　zz 时区偏移量（“+”或“-”后面仅跟小时）。一位数的小时数有前导零。例如，太平洋标准时间是“-08”。

　zzz 完整时区偏移量（“+”或“-”后面跟有小时和分钟）。一位数的小时数和分钟数有前导零。例如，太平洋标准时间是“-08:00”。

　: 在 TimeSeparator 中定义的默认时间分隔符。

　/ 在 DateSeparator 中定义的默认日期分隔符。

　% c 其中 c 是格式模式（如果单独使用）。如果格式模式与原义字符或其他格式模式合并，则可以省略“%”字符。

　\ c 其中 c 是任意字符。照原义显示字符。若要显示反斜杠字符，请使用“\\”。

### 知识点

GMT表示格林威治时间Greenwich Mean Time， GMT+08表示北京市区，一般java在中国的默认时区为北京时区，Z用来匹配GMT

## SparkContext缓存清理机制

spark会自动清理弱引用（ContextCleaner为RDD、shuffle、broadcast、*accumulator、*Checkpoint维持了一个弱引用，当相关对象不可达时，就会将对象插入referenceQueue中。有一个单独的线程来处理这个队列中的对象。）中长期没有用到的缓存，注意RDD的引用仍在。

1.6版本的源码是这样的：

**private def** persist(newLevel: StorageLevel, allowOverride: Boolean): **this**.**type** = {  
// TODO: Handle changes of StorageLevel  
**if** (*storageLevel* != StorageLevel.*NONE* && newLevel != *storageLevel* && !allowOverride) {  
**throw new** UnsupportedOperationException(  
"Cannot change storage level of an RDD after it was already assigned a level")  
 }  
// If this is the first time this RDD is marked for persisting, register it  
 // with the SparkContext for cleanups and accounting. Do this only once.  
**if**(*storageLevel* == StorageLevel.*NONE*) {  
 sc.cleaner.foreach(\_.registerRDDForCleanup(**this**))  
 sc.persistRDD(**this**)  
 }  
*storageLevel* = newLevel  
**this**}

注意会注册到Cleanup。注意看注释，当弱引用对象（checkpoint，cache等）变得weak reacheable是会加入到cleanup队列

*/\*\*  
 \* A WeakReference associated with a CleanupTask.  
 \*  
 \* When the referent object becomes only weakly reachable, the corresponding  
 \* CleanupTaskWeakReference is automatically added to the given reference queue.  
 \*/***private class** CleanupTaskWeakReference(  
**val** task: CleanupTask,  
referent: AnyRef,  
referenceQueue: ReferenceQueue[AnyRef])  
**extends** WeakReference(referent, referenceQueue)

一般大约每30分钟gc一次

**private val** *periodicGCInterval* =  
 sc.conf.getTimeAsSeconds("spark.cleaner.periodicGC.interval", "30min")

### 整个的清理机制

1）清理元数据

MetadataCleaner对象中有一个定时器，用于清理下列的元数据信息：

MAP\_OUTPUT\_TRACKER：Maptask的输出元信息

SPARK\_CONTEXT：persistentRdds中的rdd

HTTP\_BROADCAST, http广播的元数据

BLOCK\_MANAGER：blockmanager中存储的数据

SHUFFLE\_BLOCK\_MANAGER：shuffle的输出数据

BROADCAST\_VARS：Torrent方式广播broadcast的元数据

2）清理真实数据

spark会自动清理弱引用（ContextCleaner为RDD、shuffle、broadcast、*accumulator、*Checkpoint维持了一个弱引用，当相关对象不可达时，就会将对象插入referenceQueue中，ContextCleaner进行处理。

RDD：最终从各节点的blockmanager的memoryStore、diskStore中删除RDD数据

shuffle：删除driver中的mapstatuses关于该shuffleId的信息；删除所有节点中关于该shuffleId的所有分区的数据文件和索引文件

broadcast:最终从各节点的blockmanager的memoryStore、diskStore中删除broadcast数据

Checkpoint：清理checkpointDir目录下关于该rddId的文件

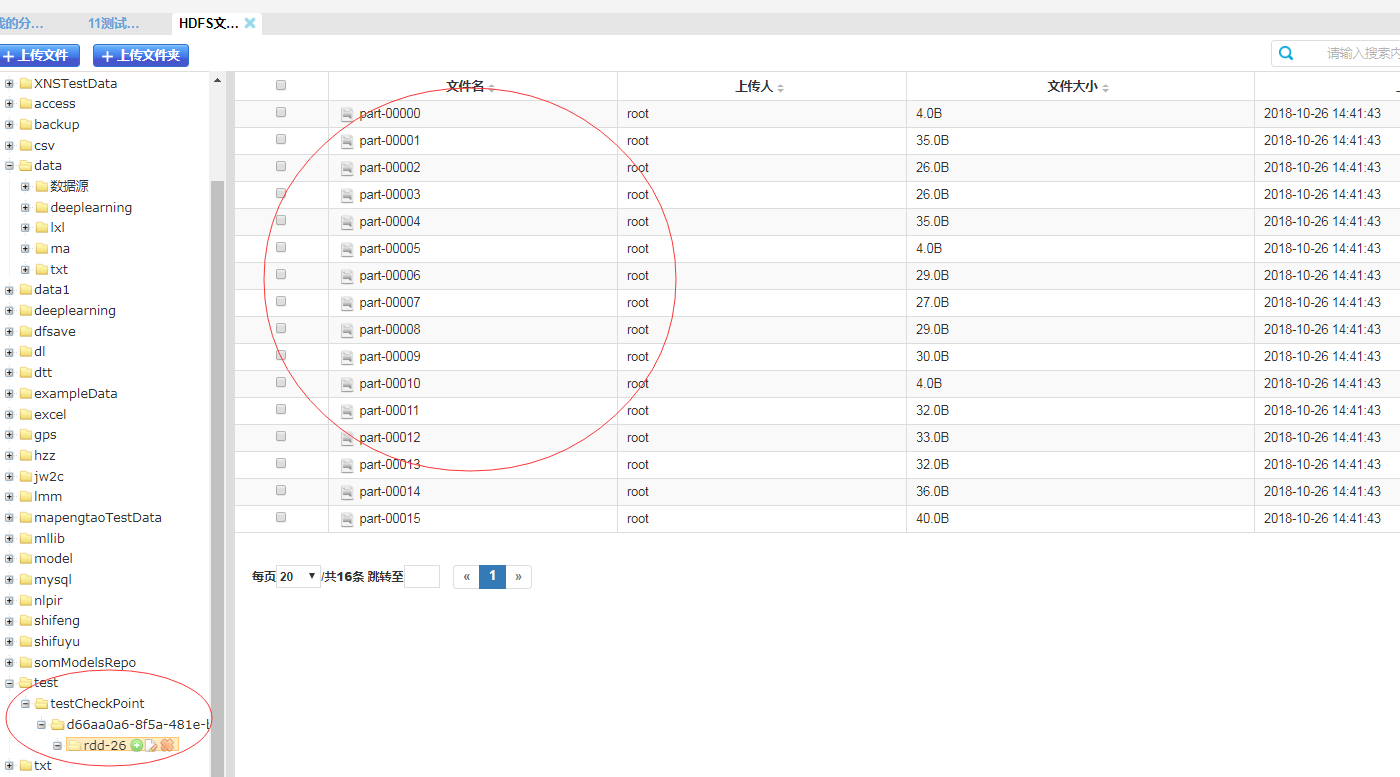
### unpersist

除了自动之外，unpersist也可以，防止某个过大的数据执行时占了太多空间，在cache并执行action后直接unpersist会为其他RDD的缓存流时间

### 实验一下——checkpoint的自动清除

1）14:40建一个RDD，checkpoint并触发它执行

**import** org.apache.spark.rdd.RDD  
*sc*.setCheckpointDir("/data/test/testCheckPoint/")  
**val** rdd1: RDD[String] = *sc*.parallelize(  
*Array*(  
"Mon Jan 08 18:22:03 CST 2018",  
"2018-01-08 18:22:03",  
"2018/01/08 18:22:03",  
"2018-01-08星期一 18:22:03",  
"2018-01-08Mon 18:22:03",  
"2018-01-08T10:22:03Z",  
"2018-01-08Mon 18:22:03",  
"2018-01-08 Mon 18:22:03",  
"2018-01-08T11:22:03+01:00",  
"2018-01-08T11:22:03.0+0100",  
"2018-01-08 11:22:03+01:00",  
"Mon, 08 Jan 2018 10:22:03 GMT",  
"Mon 08 Jan 2018 18:22:03 GMT+0800"  
)  
)  
rdd1.checkpoint()  
**val** cnt = rdd1.count()  
*println*(cnt)  
*println*(rdd1.sparkContext.getCheckpointDir)



内部是序列化后的字节。

2）15:40查看一次

## Spark sql collect\_list

**val** df = *sqlc*.createDataFrame(  
*Seq*(  
 (1, 0, "A"),  
(2, 2, "B"),  
(1, 2, "C"),  
(1, 2, "D"),  
(2, 0, "E"),  
(1, 0, "F")  
 )  
).toDF("col1", "col2", "col3")  
  
**import** org.apache.spark.sql.functions.{array, *coalesce*, *col*, collect\_list, collect\_set, first, min}  
  
df.groupBy(*col*("col1")).agg(*min*(*col*("col2")), *collect\_list*(*col*("col3"))).show()

|col1|min(col2)|collect\_list(col3)|

+----+---------+------------------+

| 1| 0| [A, C, D, F]|

| 2| 0| [B, E]|

+----+---------+------------------+

## Spark sql coalesce + outer join

**val** dataFrame1 = *sqlc*.createDataFrame(  
*Seq*(  
 ("Employee1", "salary100"),  
("Employee2", "salary50"),  
("Employee3", "salary200")  
 )).toDF("employee\_A", "salary\_A")  
  
  
**val** dataFrame2 = *sqlc*.createDataFrame(  
*Seq*(  
 ("Employee1", "salary100"),  
("Employee2", "salary50"),  
("Employee4", "salary600")  
 )).toDF("employee\_B", "salary\_B")  
  
**val** joined = dataFrame1  
 .join(dataFrame2, *col*("employee\_A") === *col*("employee\_B"), "outer")  
 .select(  
*coalesce*(*col*("employee\_A"), *col*("employee\_B")).as("employee"),  
*coalesce*(*col*("salary\_A"), *col*("salary\_B")).as("salary")  
 )  
  
joined.show()

实现类似这样的效果

id | name | age

1 | abc | 23

2 | xyz | 34

3 | pqr | 45

id | name | age

1 | lmn | 56

4 | efg | 37

-----

id | name | age

1 | lmn | 56

2 | xyz | 34

3 | pqr | 45

4 | efg | 37

## Java时间格式处理

### 时间规范

Symbol Meaning Presentation Examples

G era text AD;

u year year 2004;

y year-of-era year 2004;

D day-of-year number 189

M/L month-of-year number/text 7;

d day-of-month number 10

Q/q quarter-of-year number/text 3;

Y week-based-year year 1996;

w week-of-week-based-year number 27

W week-of-month number 4

E day-of-week text Tue;

e/c localized day-of-week number/text

F week-of-month number 3

a am-pm-of-day text PM

h clock-hour-of-am-pm (1-12) number

K hour-of-am-pm (0-11) number

k clock-hour-of-am-pm (1-24) number

H hour-of-day (0-23) number

m minute-of-hour number 30

s second-of-minute number 55

S fraction-of-second fraction 978

A milli-of-day number 1234

n nano-of-second number 987654321

N nano-of-day number 1234000000

V time-zone ID zone-id

z time-zone name zone-name

O localized zone-offset offset-O

X zone-offset ‘Z’ for

x zone-offset offset-x +0000;

Z zone-offset offset-Z +0000;

p pad next pad

‘ escape for text

“ single quote literal

[ optional section start

] optional section end

# reserved for future

{ reserved for future

} reserved for future

## Scala的forsome关键字以及下划线匹配泛型

forSome是Scala的一个关键字，不是函数。

在Scala的changeLog中，关于forSome的描述：

“It is now possible to define existential types using the new keyword forSome. An existential type has the form T forSome {Q} where Q is a sequence of value and/or type declarations. “

forSome用于下面的场景：

我们想对一些对象做一些操作，但是不关心对象内部的具体类型。或者说我们指向知道对象可以进行哪些操作，但是不关心它是什么以及可以进行哪些其他操作。比如我们想取List的第一个元素或者List的长度，但是不关心List是什么类型。因此我们需要知道该对象可以进行获取长度的操作，但是不关心其元素是什么类型：

Hence, there are useful existential types which hide some of the structure of the objects they represent but show enough structure to allow manipulations of the objects through operations the objects themselves provide

即Existential type做到了隐藏我们不关心的对象结构，同时暴露我们想要进行的操作，恰如其分地对外暴露协议。

def printFirst(x : Array[T] forSome {type T}) = println(x(0))

我们也可以使用泛型：

def printFirst[T](x : Array[T]) = println(x(0))

但是有些情况下我们可能不想使用方法泛型。

来自scala的术语表：

An existential type includes references to type variables that are unknown. For example, Array[T] forSome { type T } is an existential type. It is an array of T, where T is some completely unknown type. All that is assumed about T is that it exists at all. This assumption is weak, but it means at least that an Array[T] forSome { type T } is indeed an array and not a banana.

也就是说forSome只是表面类型存在，至于它是什么类型，不关心，Java中我们这样定义泛型：

class MyClass<?> {

...

}

也就是我们不关心泛型的类型，任何类型都可以。

另外可以指定某些类型,类似于Java中的

def addToFirst(x : Array[T] forSome {type T <: Integer}) = x(0) + 1

class MyClass<? extends Integer> {

...

}

一种更简洁的写法：

def addToFirst(x: Array[\_ <: Integer ]) = x(0) + 1

对于下面两个例子：

Array[T] forSome { type T; }

Array[T forSome { type T; }]

他们之间的区别非常大，第一个代表元素类型为任意类型的Array，第二个代表Array[Any]。即：T forSome {type T;} 等同于Any。

Type Any is equivalent to a for\_some { type a; }

再看：

Map[Class[T forSome { type T}], String] // Map[Class[Any],String]

Map[Class[T] forSome { type T}, String] // key为任意类型的Class

Map[Class[T], String] forSome { type T}

使用通配符：

Array[\_] // Array[T] forSome {type T}

Map[Class[\_] , String] //Map[Class[T] ,String] forSome {type T}

## spark rdd foreach

1）是action操作；

2）依然是并行化的；

3）并不费太多性能；