****

**大数据技术原理及应用**

实验报告



学 院： 计算机科学与技术学院

专 业： 电力信息技术

报告名称： SparkRDD综合实践

学生姓名： 崔荣成 学号： 19108002

指导老师： 周 平

时 间： 2020年6月23日

目 录

[1、从集合创建RDD 2](#_Toc43970289)

[parallelize 2](#_Toc43970290)

[makeRDD 3](#_Toc43970291)

[2、从外部存储创建RDD 5](#_Toc43970292)

[textFile 5](#_Toc43970293)

[Spark算子：RDD基本转换操作(1)–map、flatMap、distinct 6](#_Toc43970294)

[map 6](#_Toc43970295)

[flatMap 7](#_Toc43970296)

[distinct 8](#_Toc43970297)

[Spark算子：RDD基本转换操作(2)–coalesce、repartition 8](#_Toc43970298)

[coalesce 8](#_Toc43970299)

[repartition 10](#_Toc43970300)

[Spark算子：RDD基本转换操作(3)–randomSplit、glom 11](#_Toc43970301)

[randomSplit 11](#_Toc43970302)

[glom 13](#_Toc43970303)

[Spark算子：RDD基本转换操作(4)–union、intersection、subtract 13](#_Toc43970304)

[union 13](#_Toc43970305)

[intersection 14](#_Toc43970306)

[subtract 16](#_Toc43970307)

[Spark算子：RDD基本转换操作(5)–mapPartitions、mapPartitionsWithIndex 17](#_Toc43970308)

[mapPartitions 17](#_Toc43970309)

[mapPartitionsWithIndex 18](#_Toc43970310)

[Spark算子：RDD基本转换操作(6)–zip 19](#_Toc43970311)

[zip 19](#_Toc43970312)

[Spark算子：RDD基本转换操作(7)–zipWithIndex、zipWithUniqueId 21](#_Toc43970313)

[zipWithIndex 21](#_Toc43970314)

[zipWithUniqueId 21](#_Toc43970315)

[Spark算子：RDD键值转换操作(1)–partitionBy、mapValues、flatMapValues 22](#_Toc43970316)

[partitionBy 22](#_Toc43970317)

[mapValues 23](#_Toc43970318)

[flatMapValues 23](#_Toc43970319)

[Spark算子：RDD键值转换操作(2)–combineByKey 24](#_Toc43970320)

[combineByKey 24](#_Toc43970321)

[Spark算子：RDD键值转换操作(3)–groupByKey、reduceByKey、 26](#_Toc43970322)

[groupByKey 26](#_Toc43970323)

[Spark算子：RDD键值转换操作(5)–leftOuterJoin、rightOuterJoin、subtractByKey 28](#_Toc43970324)

[leftOuterJoin 28](#_Toc43970325)

[rightOuterJoin 29](#_Toc43970326)

[subtractByKey 29](#_Toc43970327)

[Spark算子：RDD行动Action操作(1)–first、count、reduce、collect 30](#_Toc43970328)

[first 30](#_Toc43970329)

[count 31](#_Toc43970330)

[reduce 31](#_Toc43970331)

[又一例 32](#_Toc43970332)

[collect 33](#_Toc43970333)

[Spark算子：RDD行动Action操作(2)–take、top、takeOrdered 33](#_Toc43970334)

[take 33](#_Toc43970335)

[top 34](#_Toc43970336)

[takeOrdered 35](#_Toc43970337)

[Spark算子：RDD行动Action操作(3)–lookup 36](#_Toc43970338)

[lookup 36](#_Toc43970339)

[Spark算子：RDD行动Action操作(4)–countByKey、foreach、foreachPartition、sortBy 37](#_Toc43970340)

[countByKey 37](#_Toc43970341)

[foreach 37](#_Toc43970342)

[sortBy 38](#_Toc43970343)

[Spark算子：RDD行动Action操作(5)–saveAsTextFile、saveAsSequenceFile、saveAsObjectFile 39](#_Toc43970344)

[saveAsTextFile 39](#_Toc43970345)

[saveAsObjectFile 40](#_Toc43970346)

# 1、从集合创建RDD

## parallelize

def parallelize[T](seq: Seq[T], numSlices: Int = defaultParallelism)

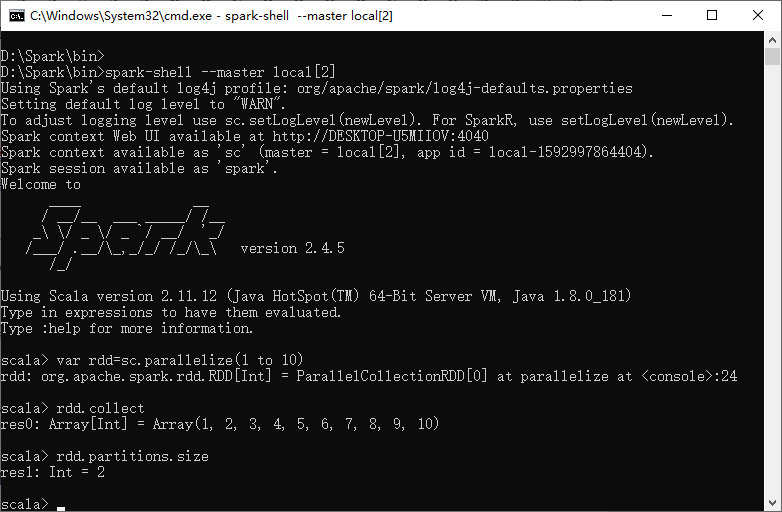
(implicit arg0: ClassTag[T]): RDD[T]

从一个Seq集合创建RDD。

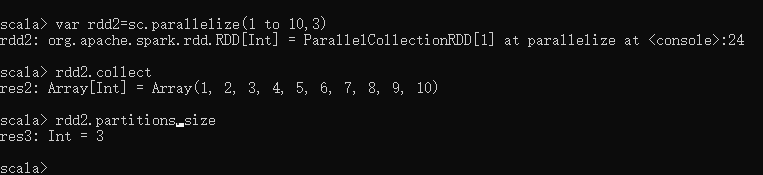
参数1：Seq集合，必须。

参数2：分区数，默认为该Application分配到的资源的CPU核数

|  |
| --- |
| **scala> var rdd = sc.parallelize(1 to 10)**  rdd: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[2] at parallelize at :21    scala> rdd.collect  res3: Array[Int] = Array(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)    scala> rdd.partitions.size  res4: Int = 15 |



|  |
| --- |
| //设置RDD为3个分区  **scala> var rdd2 = sc.parallelize(1 to 10,3)**  rdd2: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[3] at parallelize at :21    scala> rdd2.collect  res5: Array[Int] = Array(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)  scala> rdd2.partitions.size  res6: Int = 3 |



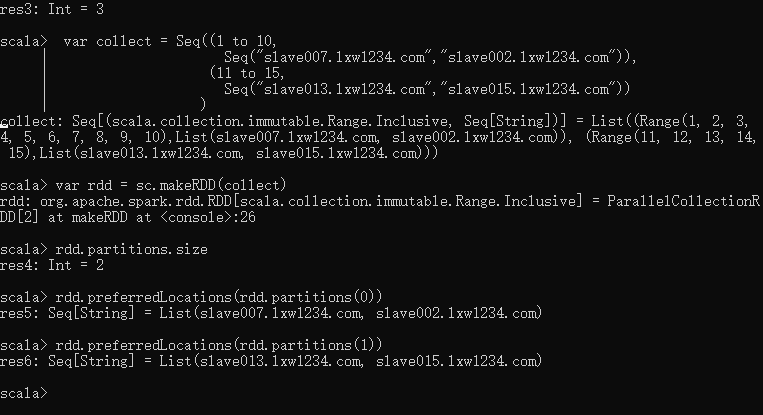
## makeRDD

def makeRDD[T](seq: Seq[(T, Seq[String])])(implicit arg0: ClassTag[T]): RDD[T]

该用法可以指定每一个分区的preferredLocations。

|  |
| --- |
| **scala> var collect = Seq((1 to 10,**  **Seq("slave007.lxw1234.com","slave002.lxw1234.com")),**  **(11 to 15,**  **Seq("slave013.lxw1234.com","slave015.lxw1234.com"))**  **)**  collect: Seq[(scala.collection.immutable.Range.Inclusive, Seq[String])] = List((Range(1, 2, 3, 4, 5, 6, 7, 8, 9, 10),  List(slave007.lxw1234.com, slave002.lxw1234.com)), (Range(11, 12, 13, 14, 15),List(slave013.lxw1234.com, slave015.lxw1234.com)))    **scala> var rdd = sc.makeRDD(collect)**  rdd: org.apache.spark.rdd.RDD[scala.collection.immutable.Range.Inclusive] = ParallelCollectionRDD[6] at makeRDD at :23    scala> rdd.partitions.size  res33: Int = 2    scala> rdd.preferredLocations(rdd.partitions(0))  res34: Seq[String] = List(slave007.lxw1234.com, slave002.lxw1234.com)    scala> rdd.preferredLocations(rdd.partitions(1))  res35: Seq[String] = List(slave013.lxw1234.com, slave015.lxw1234.com) |

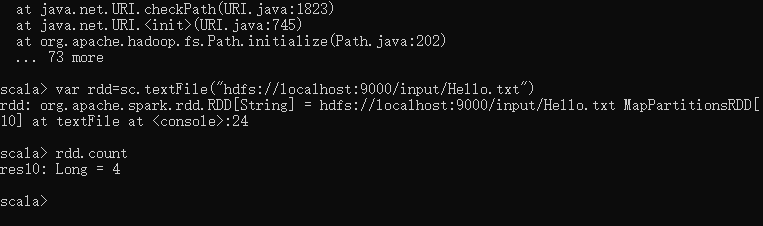
指定分区的优先位置，对后续的调度优化有帮助。



# 2、从外部存储创建RDD

## textFile

|  |
| --- |
| //从hdfs文件创建.  scala> **var rdd = sc.textFile("hdfs:///tmp/lxw1234/1.txt")**  rdd: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[26] at textFile at :21    scala> rdd.count  res48: Long = 4 |



|  |
| --- |
| //从本地文件创建  **scala> var rdd = sc.textFile("file:///etc/hadoop/conf/core-site.xml")**  rdd: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[28] at textFile at :21    scala> rdd.count  res49: Long = 97 |



# Spark算子：RDD基本转换操作(1)–map、flatMap、distinct

## map

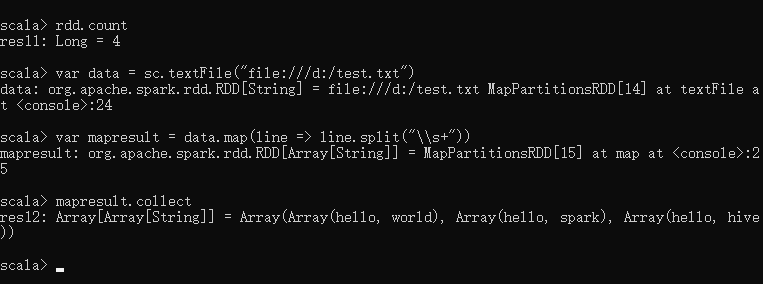
将一个RDD中的每个数据项，通过map中的函数映射变为一个新的元素。

输入分区与输出分区一对一，即：有多少个输入分区，就有多少个输出分区。

|  |
| --- |
| hadoop fs -cat /tmp/lxw1234/1.txt  hello world  hello spark  hello hive |



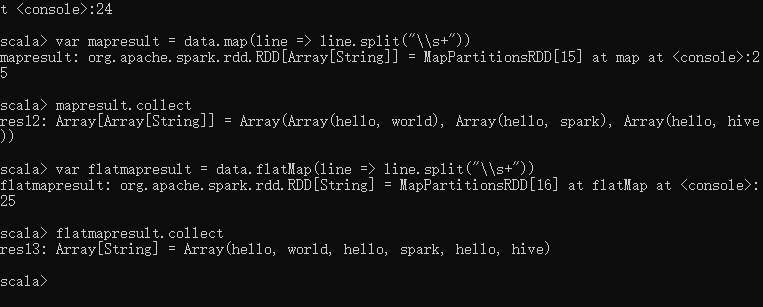
|  |
| --- |
| //读取HDFS文件到RDD  scala> var data = sc.textFile("**file:///**d:/test.txt")  data: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[1] at textFile at :21  //使用map算子  scala> **var mapresult = data.map(line => line.split("\\s+"))**  mapresult: org.apache.spark.rdd.RDD[Array[String]] = MapPartitionsRDD[2] at map at :23    //运算map算子结果  scala> mapresult.collect  res0: Array[Array[String]] = Array(Array(hello, world), Array(hello, spark), Array(hello, hive)) |



## flatMap

**属于Transformation算子，第一步和map一样，最后将所有的输出分区合并成一个。**

|  |
| --- |
| //使用flatMap算子  **scala> var flatmapresult = data.flatMap(line => line.split("\\s+"))**  flatmapresult: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[3] at flatMap at :23    //运算flagMap算子结果  scala> flatmapresult.collect  res1: Array[String] = Array(hello, world, hello, spark, hello, hive) |

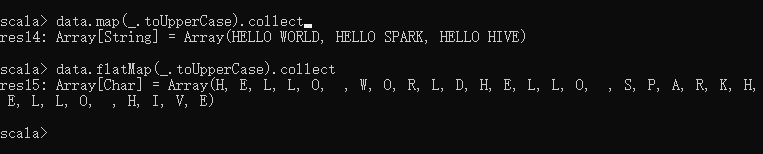


使用flatMap时候需要注意：

flatMap会将字符串看成是一个字符数组。

看下面的例子：

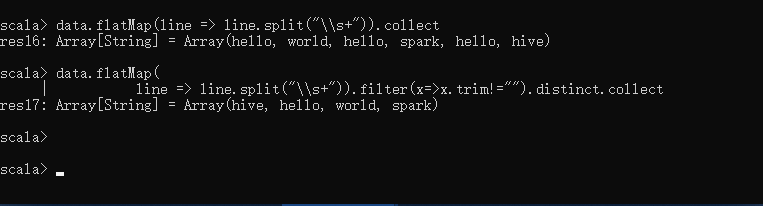
|  |
| --- |
| scala> data.map(\_.toUpperCase).collect  res32: Array[String] = Array(HELLO WORLD, HELLO SPARK, HELLO HIVE, HI SPARK)  scala> data.flatMap(\_.toUpperCase).collect  res33: Array[Char] = Array(H, E, L, L, O, , W, O, R, L, D, H, E, L, L, O, , S, P, A, R, K, H, E, L, L, O, , H, I, V, E, H, I, , S, P, A, R, K) |



## distinct

对RDD中的元素进行去重操作。

|  |
| --- |
| scala> data.flatMap(line => line.split("\\s+")).collect  res61: Array[String] = Array(hello, world, hello, spark, hello, hive, hi, spark)    **scala> data.flatMap(**  **line => line.split("\\s+")).filter(x=>x.trim!="").distinct.collect**  res62: Array[String] = Array(hive, hello, world, spark, hi) |



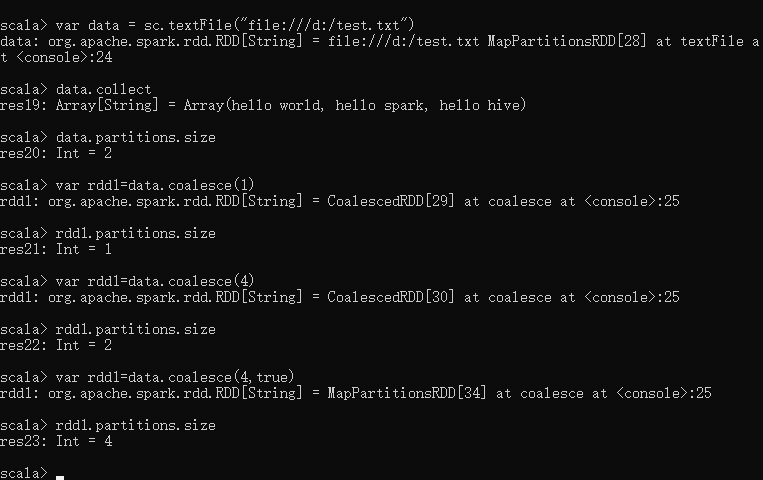
# Spark算子：RDD基本转换操作(2)–coalesce、repartition

## coalesce

该函数用于将RDD进行重分区，使用HashPartitioner。

以下面的例子来看：

|  |
| --- |
| scala> var data = sc.textFile("/tmp/lxw1234/1.txt")  data: org.apache.spark.rdd.RD0D[String] = MapPartitionsRDD[53] at textFile at :21    scala> data.collect  res37: Array[String] = Array(hello world, hello spark, hello hive, hi spark)    scala> data.partitions.size  res38: Int = 2 //RDD data默认有两个分区    scala> var rdd1 = data.coalesce(1)  rdd1: org.apache.spark.rdd.RDD[String] = CoalescedRDD[2] at coalesce at :23    scala> rdd1.partitions.size  res1: Int = 1 //rdd1的分区数为1    scala> var rdd1 = data.coalesce(4)  rdd1: org.apache.spark.rdd.RDD[String] = CoalescedRDD[3] at coalesce at :23    scala> rdd1.partitions.size  res2: Int = 2 //如果重分区的数目大于原来的分区数，那么必须指定shuffle参数为true，//否则，分区数不便    scala> var rdd1 = data.coalesce(4,true)  rdd1: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[7] at coalesce at :23    scala> rdd1.partitions.size  res3: Int = 4 |

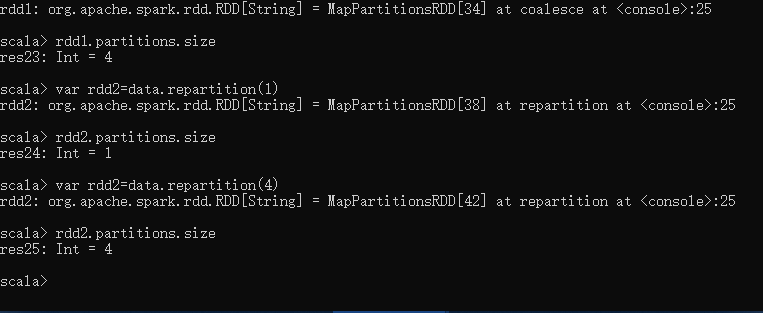


## repartition

def repartition(numPartitions: Int)(implicit ord: Ordering[T] = null): RDD[T]

该函数其实就是coalesce函数第二个参数为true的实现

|  |
| --- |
| scala> var rdd2 = data.repartition(1)  rdd2: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[11] at repartition at :23    scala> rdd2.partitions.size  res4: Int = 1    scala> var rdd2 = data.repartition(4)  rdd2: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[15] at repartition at :23    scala> rdd2.partitions.size  res5: Int = 4 |



# Spark算子：RDD基本转换操作(3)–randomSplit、glom

## randomSplit

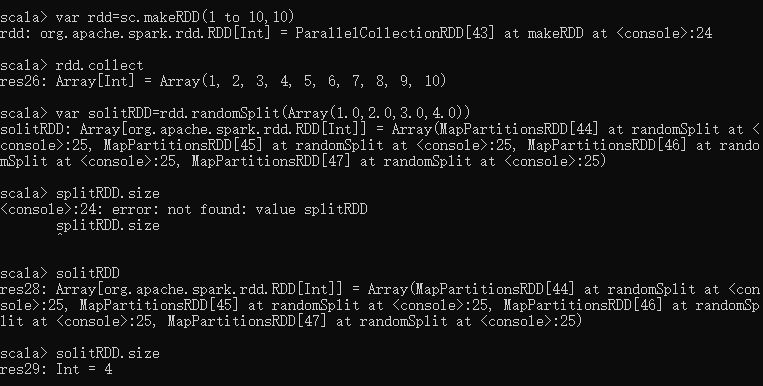
def randomSplit(weights: Array[Double], seed: Long = Utils.random.nextLong): Array[RDD[T]]

该函数根据weights权重，将一个RDD切分成多个RDD。

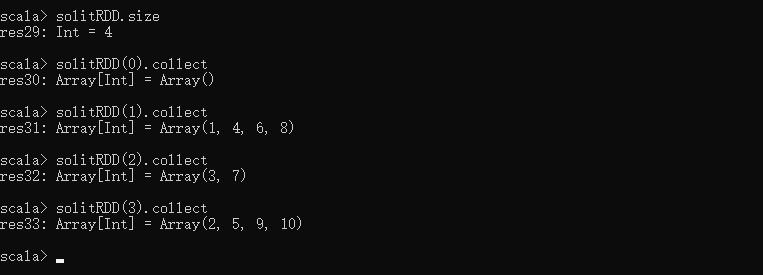
该权重参数为一个Double数组

第二个参数为random的种子，基本可忽略。

|  |
| --- |
| scala> var rdd = sc.makeRDD(1 to 10,10)  rdd: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[16] at makeRDD at :21    scala> rdd.collect  res6: Array[Int] = Array(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)    scala> var splitRDD = rdd.randomSplit(Array(1.0,2.0,3.0,4.0))  splitRDD: Array[org.apache.spark.rdd.RDD[Int]] = Array(MapPartitionsRDD[17] at randomSplit at :23,  MapPartitionsRDD[18] at randomSplit at :23,  MapPartitionsRDD[19] at randomSplit at :23,  MapPartitionsRDD[20] at randomSplit at :23)    //这里注意：randomSplit的结果是一个RDD数组  scala> splitRDD.size  res8: Int = 4  //由于randomSplit的第一个参数weights中传入的值有4个，因此，就会切分成4个RDD,  //把原来的rdd按照权重1.0,2.0,3.0,4.0，随机划分到这4个RDD中，权重高的RDD，划分到//的几率就大一些。  //注意，权重的总和加起来为1，否则会不正常 |



|  |
| --- |
| scala> splitRDD(0).collect  res10: Array[Int] = Array(1, 4)    scala> splitRDD(1).collect  res11: Array[Int] = Array(3)    scala> splitRDD(2).collect  res12: Array[Int] = Array(5, 9)    scala> splitRDD(3).collect  res13: Array[Int] = Array(2, 6, 7, 8, 10) |

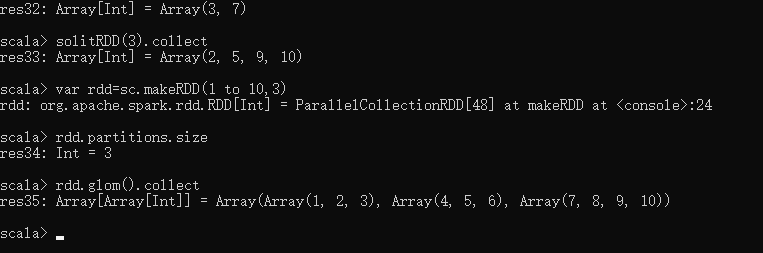


## glom

def glom(): RDD[Array[T]]

该函数是将RDD中每一个分区中类型为T的元素转换成Array[T]，这样每一个分区就只有一个数组元素。

|  |
| --- |
| scala> var rdd = sc.makeRDD(1 to 10,3)  rdd: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[38] at makeRDD at :21  scala> rdd.partitions.size  res33: Int = 3 //该RDD有3个分区  scala> rdd.glom().collect  res35: Array[Array[Int]] = Array(Array(1, 2, 3), Array(4, 5, 6), Array(7, 8, 9, 10))  //glom将每个分区中的元素放到一个数组中，这样，结果就变成了3个数组 |



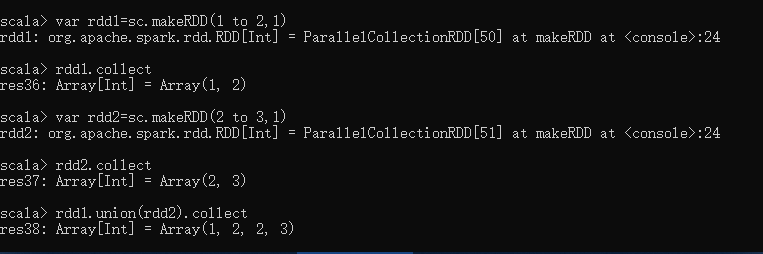
# Spark算子：RDD基本转换操作(4)–union、intersection、subtract

## union

def union(other: RDD[T]): RDD[T]

该函数比较简单，就是将两个RDD进行合并，不去重。

|  |
| --- |
| scala> var rdd1 = sc.makeRDD(1 to 2,1)  rdd1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[45] at makeRDD at :21    scala> rdd1.collect  res42: Array[Int] = Array(1, 2)    scala> var rdd2 = sc.makeRDD(2 to 3,1)  rdd2: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[46] at makeRDD at :21    scala> rdd2.collect  res43: Array[Int] = Array(2, 3)    scala> rdd1.union(rdd2).collect  res44: Array[Int] = Array(1, 2, 2, 3) |



## intersection

def intersection(other: RDD[T]): RDD[T]

def intersection(other: RDD[T], numPartitions: Int): RDD[T]

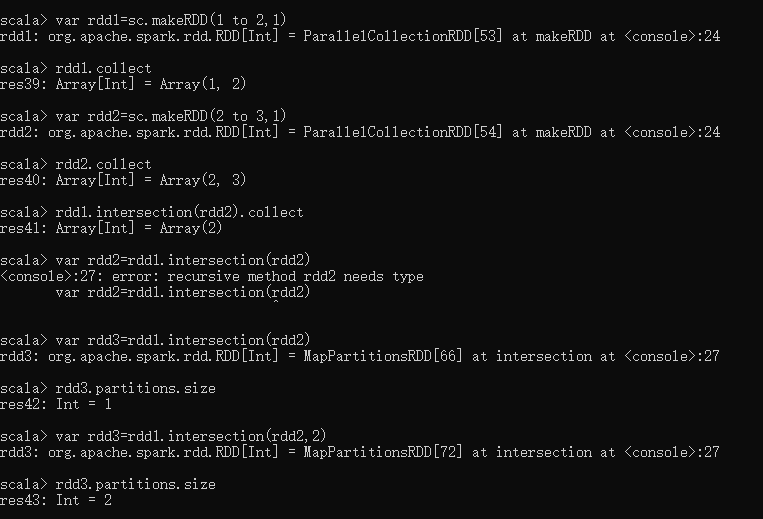
def intersection(other: RDD[T], partitioner: Partitioner)(implicit ord: Ordering[T] = null): RDD[T]

该函数返回两个RDD的交集，并且去重。

参数numPartitions指定返回的RDD的分区数。

参数partitioner用于指定分区函数

|  |
| --- |
| scala> var rdd1 = sc.makeRDD(1 to 2,1)  rdd1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[45] at makeRDD at :21    scala> rdd1.collect  res42: Array[Int] = Array(1, 2)    scala> var rdd2 = sc.makeRDD(2 to 3,1)  rdd2: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[46] at makeRDD at :21    scala> rdd2.collect  res43: Array[Int] = Array(2, 3)    scala> rdd1.intersection(rdd2).collect  res45: Array[Int] = Array(2)    scala> var rdd3 = rdd1.intersection(rdd2)  rdd3: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[59] at intersection at :25    scala> rdd3.partitions.size  res46: Int = 1    scala> var rdd3 = rdd1.intersection(rdd2,2)  rdd3: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[65] at intersection at :25    scala> rdd3.partitions.size  res47: Int = 2 |



## subtract

def subtract(other: RDD[T]): RDD[T]

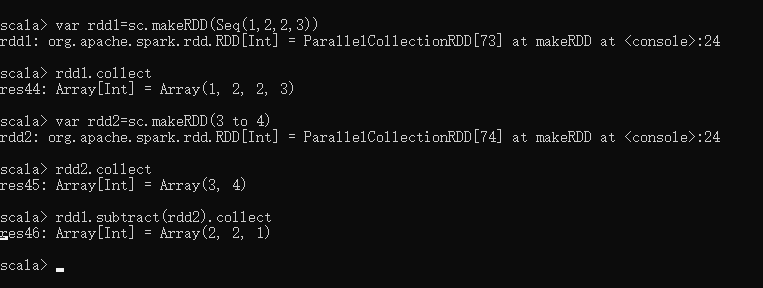
def subtract(other: RDD[T], numPartitions: Int): RDD[T]

def subtract(other: RDD[T], partitioner: Partitioner)(implicit ord: Ordering[T] = null): RDD[T]

该函数类似于intersection，但返回在RDD中出现，并且不在otherRDD中出现的元素，不去重。

参数含义同intersection

|  |
| --- |
| scala> var rdd1 = sc.makeRDD(Seq(1,2,2,3))  rdd1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[66] at makeRDD at :21    scala> rdd1.collect  res48: Array[Int] = Array(1, 2, 2, 3)    scala> var rdd2 = sc.makeRDD(3 to 4)  rdd2: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[67] at makeRDD at :21    scala> rdd2.collect  res49: Array[Int] = Array(3, 4)    scala> rdd1.subtract(rdd2).collect  res50: Array[Int] = Array(1, 2, 2) |



# Spark算子：RDD基本转换操作(5)–mapPartitions、mapPartitionsWithIndex

## mapPartitions

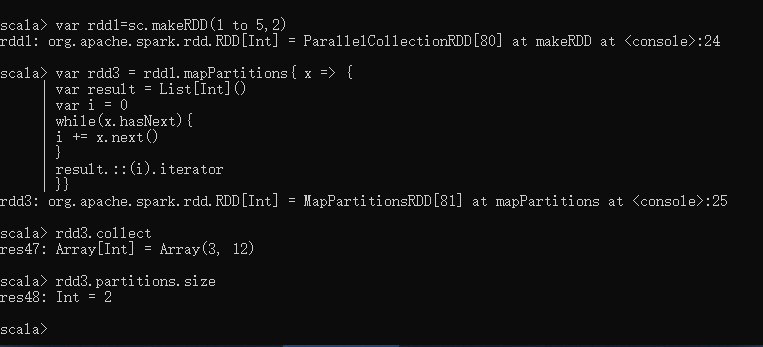
def mapPartitions[U](f: (Iterator[T]) => Iterator[U], preservesPartitioning: Boolean = false)(implicit arg0: ClassTag[U]): RDD[U]

该函数和map函数类似，只不过映射函数的参数由RDD中的每一个元素变成了RDD中每一个分区的迭代器。如果在映射的过程中需要频繁创建额外的对象，使用mapPartitions要比map高效的过。

比如，将RDD中的所有数据通过JDBC连接写入数据库，如果使用map函数，可能要为每一个元素都创建一个connection，这样开销很大，如果使用mapPartitions，那么只需要针对每一个分区建立一个connection。

参数preservesPartitioning表示是否保留父RDD的partitioner分区信息。

|  |
| --- |
| var rdd1 = sc.makeRDD(1 to 5,2)  //rdd1有两个分区  scala>var rdd3 = rdd1.mapPartitions{ x => {  var result = List[Int]()  var i = 0  while(x.hasNext){  i += x.next()  }  result.::(i).iterator  }}  rdd3: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[84] at mapPartitions at :23    //rdd3将rdd1中每个分区中的数值累加  scala> rdd3.collect  res65: Array[Int] = Array(3, 12)  scala> rdd3.partitions.size  res66: Int = 2 |

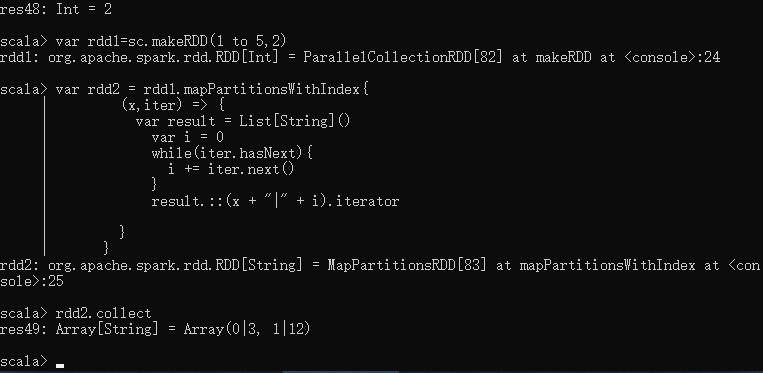


## mapPartitionsWithIndex

def mapPartitionsWithIndex[U](f: (Int, Iterator[T]) => Iterator[U], preservesPartitioning: Boolean = false)(implicit arg0: ClassTag[U]): RDD[U]

函数作用同mapPartitions，不过提供了两个参数，第一个参数为分区的索引。

|  |
| --- |
| var rdd1 = sc.makeRDD(1 to 5,2)  //rdd1有两个分区  var rdd2 = rdd1.mapPartitionsWithIndex{  (x,iter) => {  var result = List[String]()  var i = 0  while(iter.hasNext){  i += iter.next()  }  result.::(x + "|" + i).iterator    }  }  //rdd2将rdd1中每个分区的数字累加，并在每个分区的累加结果前面加了分区索引  scala> rdd2.collect  res13: Array[String] = Array(0|3, 1|12) |



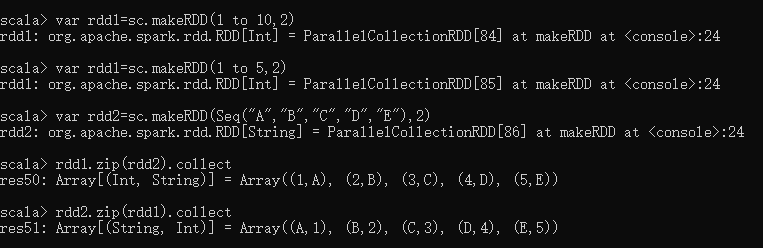
# Spark算子：RDD基本转换操作(6)–zip

## zip

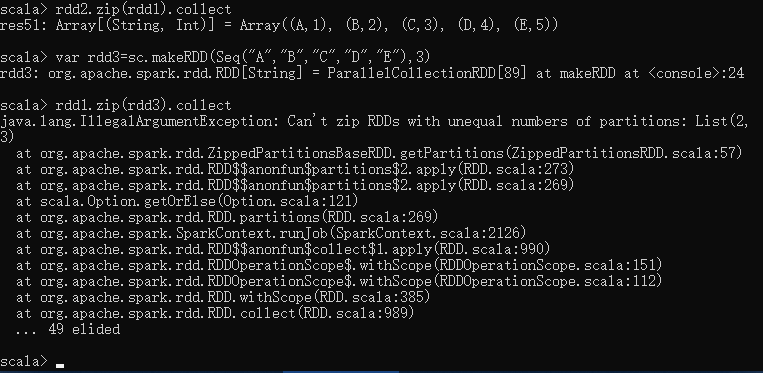
def zip[U](other: RDD[U])(implicit arg0: ClassTag[U]): RDD[(T, U)]

zip函数用于将两个RDD组合成Key/Value形式的RDD,这里默认两个RDD的partition数量以及元素数量都相同，否则会抛出异常。

|  |
| --- |
| scala> var rdd1 = sc.makeRDD(1 to 10,2)  rdd1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[0] at makeRDD at :21    scala> var rdd1 = sc.makeRDD(1 to 5,2)  rdd1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[1] at makeRDD at :21    scala> var rdd2 = sc.makeRDD(Seq("A","B","C","D","E"),2)  rdd2: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[2] at makeRDD at :21    scala> rdd1.zip(rdd2).collect  res0: Array[(Int, String)] = Array((1,A), (2,B), (3,C), (4,D), (5,E))    scala> rdd2.zip(rdd1).collect  res1: Array[(String, Int)] = Array((A,1), (B,2), (C,3), (D,4), (E,5)) |



|  |
| --- |
| scala> var rdd3 = sc.makeRDD(Seq("A","B","C","D","E"),3)  rdd3: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[5] at makeRDD at :21    scala> rdd1.zip(rdd3).collect  java.lang.IllegalArgumentException: Can't zip RDDs with unequal numbers of partitions  //如果两个RDD分区数不同，则抛出异常 |



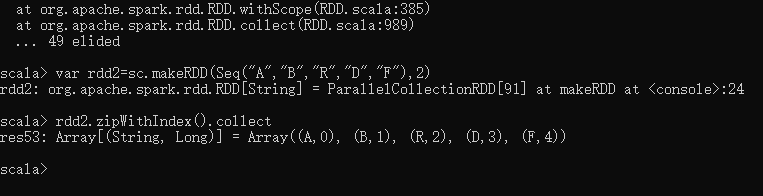
# Spark算子：RDD基本转换操作(7)–zipWithIndex、zipWithUniqueId

## zipWithIndex

def zipWithIndex(): RDD[(T, Long)]

该函数将RDD中的元素和这个元素在RDD中的ID（索引号）组合成键/值对。

|  |
| --- |
| scala> var rdd2 = sc.makeRDD(Seq("A","B","R","D","F"),2)  rdd2: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[34] at makeRDD at :21    scala> rdd2.zipWithIndex().collect  res27: Array[(String, Long)] = Array((A,0), (B,1), (R,2), (D,3), (F,4)) |



## zipWithUniqueId

def zipWithUniqueId(): RDD[(T, Long)]

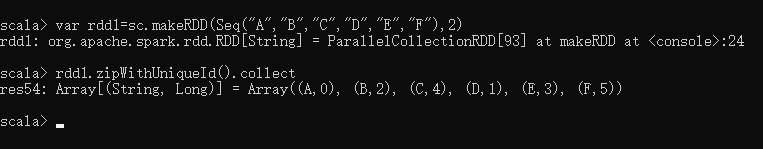
该函数将RDD中元素和一个唯一ID组合成键/值对，该唯一ID生成算法如下：

每个分区中第一个元素的唯一ID值为：该分区索引号，

每个分区中第N个元素的唯一ID值为：(前一个元素的唯一ID值) + (该RDD总的分区数)

看下面的例子：

|  |
| --- |
| scala> var rdd1 = sc.makeRDD(Seq("A","B","C","D","E","F"),2)  rdd1: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[44] at makeRDD at :21  //rdd1有两个分区，  scala> rdd1.zipWithUniqueId().collect  res32: Array[(String, Long)] = Array((A,0), (B,2), (C,4), (D,1), (E,3), (F,5))  //总分区数为2  //第一个分区第一个元素ID为0，第二个分区第一个元素ID为1  //第一个分区第二个元素ID为0+2=2，第一个分区第三个元素ID为2+2=4  //第二个分区第二个元素ID为1+2=3，第二个分区第三个元素ID为3+2=5 |



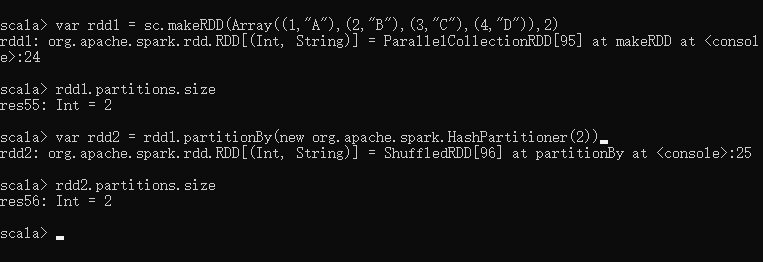
# Spark算子：RDD键值转换操作(1)–partitionBy、mapValues、flatMapValues

## partitionBy

def partitionBy(partitioner: Partitioner): RDD[(K, V)]

该函数根据partitioner函数生成新的ShuffleRDD，将原RDD重新分区。

|  |
| --- |
| scala> var rdd1 = sc.makeRDD(Array((1,"A"),(2,"B"),(3,"C"),(4,"D")),2)  rdd1: org.apache.spark.rdd.RDD[(Int, String)] = ParallelCollectionRDD[23] at makeRDD at :21    scala> rdd1.partitions.size  res20: Int = 2    //使用partitionBy重分区  scala> var rdd2 = rdd1.partitionBy(new org.apache.spark.HashPartitioner(2))  rdd2: org.apache.spark.rdd.RDD[(Int, String)] = ShuffledRDD[25] at partitionBy at :23    scala> rdd2.partitions.size  res23: Int = 2 |

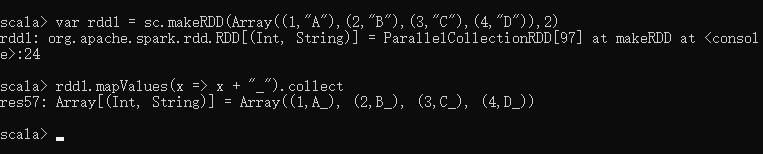


## mapValues

def mapValues[U](f: (V) => U): RDD[(K, U)]

同基本转换操作中的map，只不过mapValues是针对[K,V]中的V值进行map操作。

|  |
| --- |
| scala> var rdd1 = sc.makeRDD(Array((1,"A"),(2,"B"),(3,"C"),(4,"D")),2)  rdd1: org.apache.spark.rdd.RDD[(Int, String)] = ParallelCollectionRDD[27] at makeRDD at :21    scala> rdd1.mapValues(x => x + "\_").collect  res26: Array[(Int, String)] = Array((1,A\_), (2,B\_), (3,C\_), (4,D\_)) |

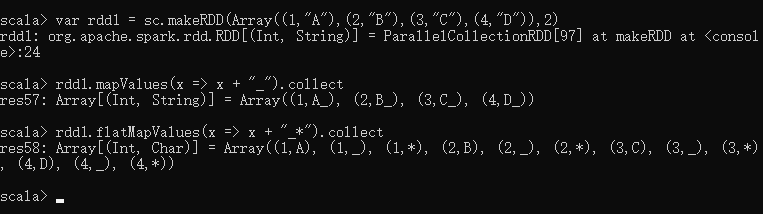


## flatMapValues

def flatMapValues[U](f: (V) => TraversableOnce[U]): RDD[(K, U)]

同基本转换操作中的flatMap，只不过flatMapValues是针对[K,V]中的V值进行flatMap操作。

|  |
| --- |
| scala> rdd1.flatMapValues(x => x + "\_").collect  res36: Array[(Int, Char)] = Array((1,A), (1,\_), (2,B), (2,\_), (3,C), (3,\_), (4,D), (4,\_)) |



# Spark算子：RDD键值转换操作(2)–combineByKey

## combineByKey

def combineByKey[C](createCombiner: (V) => C, mergeValue: (C, V) => C, mergeCombiners: (C, C) => C): RDD[(K, C)]

该函数用于将RDD[K,V]转换成RDD[K,C],这里的V类型和C类型可以相同也可以不同。

其中的参数：

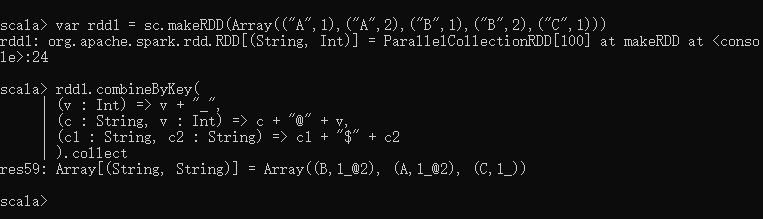
createCombiner：组合器函数，用于将V类型转换成C类型，输入参数为RDD[K,V]中的V,输出为C

mergeValue：合并值函数，将一个C类型和一个V类型值合并成一个C类型，输入参数为(C,V)，输出为C

mergeCombiners：合并组合器函数，用于将两个C类型值合并成一个C类型，输入参数为(C,C)，输出为C

看下面例子：

|  |
| --- |
| scala> var rdd1 = sc.makeRDD(Array(("A",1),("A",2),("B",1),("B",2),("C",1)))  scala> rdd1.combineByKey(  (v : Int) => v + "\_",  (c : String, v : Int) => c + "@" + v,  (c1 : String, c2 : String) => c1 + "$" + c2  ).collect  res60: Array[(String, String)] = Array((A,2\_$1\_), (B,1\_$2\_), (C,1\_)) |



其中三个映射函数分别为：

createCombiner: (V) => C

(v : Int) => v + “\_” //在每一个V值后面加上字符\_，返回C类型(String)

mergeValue: (C, V) => C

(c : String, v : Int) => c + “@” + v //合并C类型和V类型，中间加字符@,返回C(String)

mergeCombiners: (C, C) => C

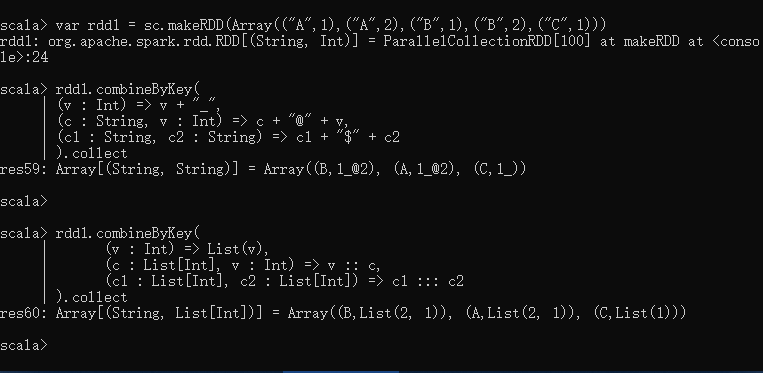
(c1 : String, c2 : String) => c1 + “$” + c2 //合并C类型和C类型，中间加$，返回C(String)

其他参数为默认值。

最终，将RDD[String,Int]转换为RDD[String,String]。

再看例子：

|  |
| --- |
| rdd1.combineByKey(  (v : Int) => List(v),  (c : List[Int], v : Int) => v :: c,  (c1 : List[Int], c2 : List[Int]) => c1 ::: c2  ).collect  res65: Array[(String, List[Int])] = Array((A,List(2, 1)), (B,List(2, 1)), (C,List(1)))  最终将RDD[String,Int]转换为RDD[String,List[Int]]。 |



# Spark算子：RDD键值转换操作(3)–groupByKey、reduceByKey、

## groupByKey

def groupByKey(): RDD[(K, Iterable[V])]

def groupByKey(numPartitions: Int): RDD[(K, Iterable[V])]

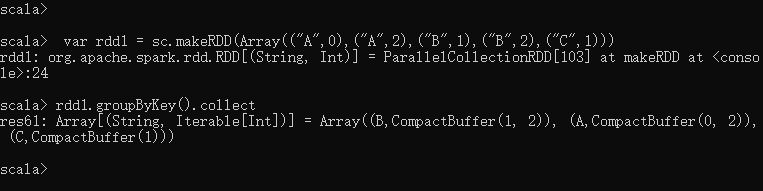
def groupByKey(partitioner: Partitioner): RDD[(K, Iterable[V])]

该函数用于将RDD[K,V]中每个K对应的V值，合并到一个集合Iterable[V]中，

参数numPartitions用于指定分区数；

参数partitioner用于指定分区函数；

|  |
| --- |
| scala> var rdd1 = sc.makeRDD(Array(("A",0),("A",2),("B",1),("B",2),("C",1)))  rdd1: org.apache.spark.rdd.RDD[(String, Int)] = ParallelCollectionRDD[89] at makeRDD at :21    scala> rdd1.groupByKey().collect  res81: Array[(String, Iterable[Int])] = Array((A,CompactBuffer(0, 2)), (B,CompactBuffer(2, 1)), (C,CompactBuffer(1))) |



reduceByKey

def reduceByKey(func: (V, V) => V): RDD[(K, V)]

def reduceByKey(func: (V, V) => V, numPartitions: Int): RDD[(K, V)]

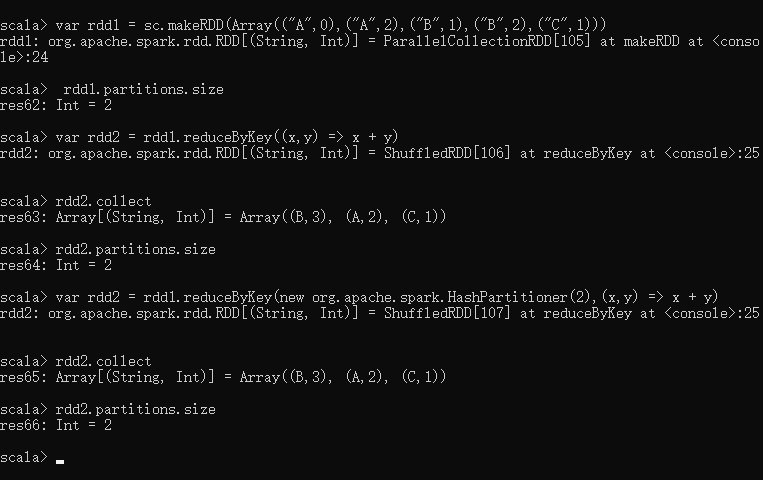
def reduceByKey(partitioner: Partitioner, func: (V, V) => V): RDD[(K, V)]

该函数用于将RDD[K,V]中每个K对应的V值根据映射函数来运算。

参数numPartitions用于指定分区数；

参数partitioner用于指定分区函数；

|  |
| --- |
| scala> var rdd1 = sc.makeRDD(Array(("A",0),("A",2),("B",1),("B",2),("C",1)))  rdd1: org.apache.spark.rdd.RDD[(String, Int)] = ParallelCollectionRDD[91] at makeRDD at :21    scala> rdd1.partitions.size  res82: Int = 15    scala> var rdd2 = rdd1.reduceByKey((x,y) => x + y)  rdd2: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[94] at reduceByKey at :23    scala> rdd2.collect  res85: Array[(String, Int)] = Array((A,2), (B,3), (C,1))    scala> rdd2.partitions.size  res86: Int = 15    scala> var rdd2 = rdd1.reduceByKey(new org.apache.spark.HashPartitioner(2),(x,y) => x + y)  rdd2: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[95] at reduceByKey at :23    scala> rdd2.collect  res87: Array[(String, Int)] = Array((B,3), (A,2), (C,1))    scala> rdd2.partitions.size  res88: Int = 2 |

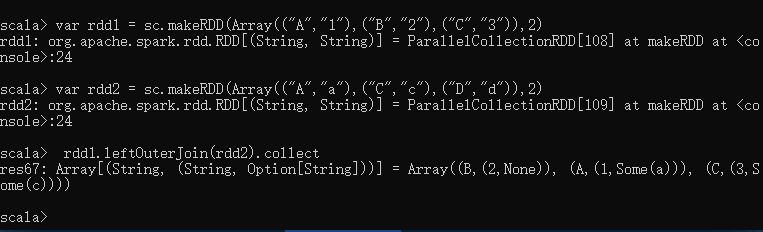


# Spark算子：RDD键值转换操作(5)–leftOuterJoin、rightOuterJoin、subtractByKey

## leftOuterJoin

leftOuterJoin类似于SQL中的左外关联left outer join，返回结果以前面的RDD为主，关联不上的记录为空。只能用于两个RDD之间的关联，如果要多个RDD关联，多关联几次即可。

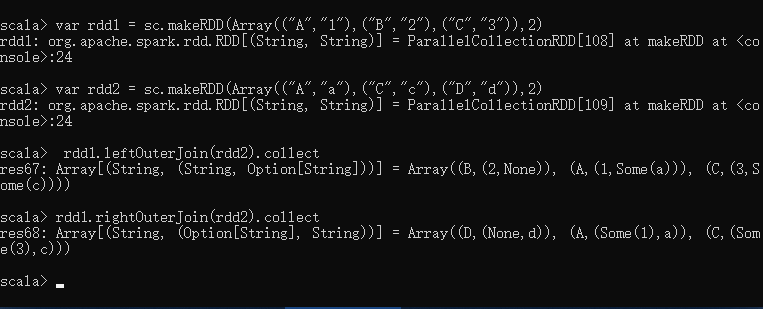
|  |
| --- |
| var rdd1 = sc.makeRDD(Array(("A","1"),("B","2"),("C","3")),2)  var rdd2 = sc.makeRDD(Array(("A","a"),("C","c"),("D","d")),2)    scala> rdd1.leftOuterJoin(rdd2).collect  res11: Array[(String, (String, Option[String]))] = Array((B,(2,None)), (A,(1,Some(a))), (C,(3,Some(c)))) |



## rightOuterJoin

rightOuterJoin类似于SQL中的有外关联right outer join，返回结果以参数中的RDD为主，关联不上的记录为空。只能用于两个RDD之间的关联，如果要多个RDD关联，多关联几次即可。

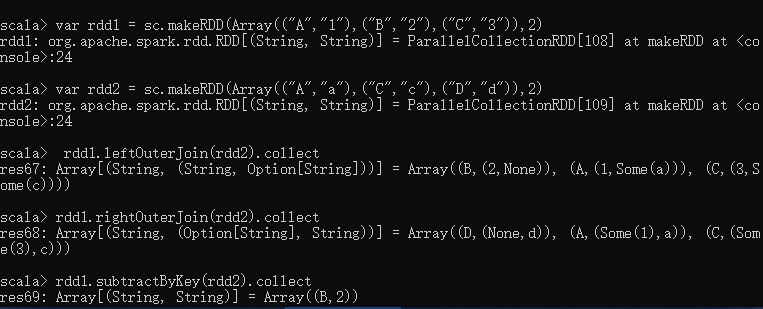
|  |
| --- |
| var rdd1 = sc.makeRDD(Array(("A","1"),("B","2"),("C","3")),2)  var rdd2 = sc.makeRDD(Array(("A","a"),("C","c"),("D","d")),2)  scala> rdd1.rightOuterJoin(rdd2).collect  res12: Array[(String, (Option[String], String))] = Array((D,(None,d)), (A,(Some(1),a)), (C,(Some(3),c))) |



## subtractByKey

subtractByKey和基本转换操作中的subtract类似，只不过这里是针对K的，返回在主RDD中出现，并且不在otherRDD中出现的元素。

|  |
| --- |
| var rdd1 = sc.makeRDD(Array(("A","1"),("B","2"),("C","3")),2)  var rdd2 = sc.makeRDD(Array(("A","a"),("C","c"),("D","d")),2)    scala> rdd1.subtractByKey(rdd2).collect  res13: Array[(String, String)] = Array((B,2)) |



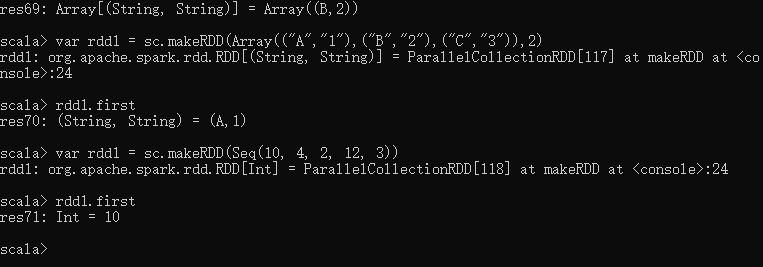
# Spark算子：RDD行动Action操作(1)–first、count、reduce、collect

## first

def first(): T

first返回RDD中的第一个元素，不排序。

|  |
| --- |
| scala> var rdd1 = sc.makeRDD(Array(("A","1"),("B","2"),("C","3")),2)  rdd1: org.apache.spark.rdd.RDD[(String, String)] = ParallelCollectionRDD[33] at makeRDD at :21    scala> rdd1.first  res14: (String, String) = (A,1)    scala> var rdd1 = sc.makeRDD(Seq(10, 4, 2, 12, 3))  rdd1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[0] at makeRDD at :21    scala> rdd1.first  res8: Int = 10 |

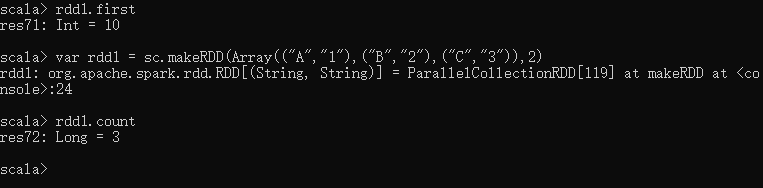


## count

def count(): Long

count返回RDD中的元素数量。

|  |
| --- |
| scala> var rdd1 = sc.makeRDD(Array(("A","1"),("B","2"),("C","3")),2)  rdd1: org.apache.spark.rdd.RDD[(String, String)] = ParallelCollectionRDD[34] at makeRDD at :21    scala> rdd1.count  res15: Long = 3 |

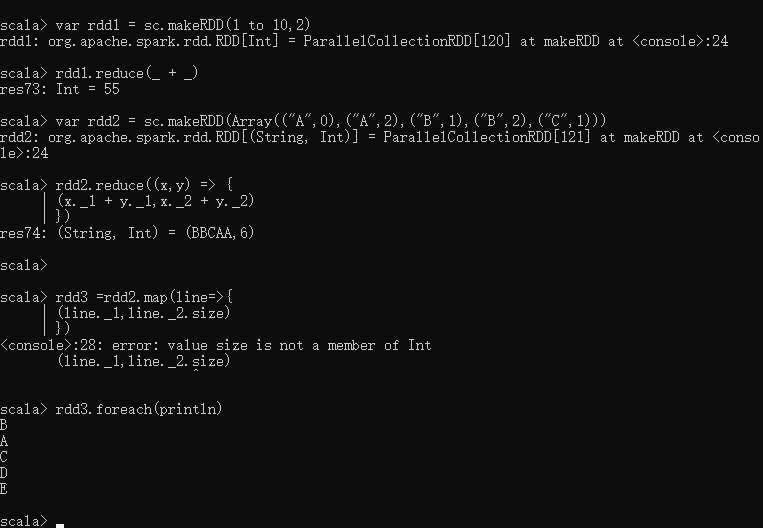


## reduce

def reduce(f: (T, T) ⇒ T): T

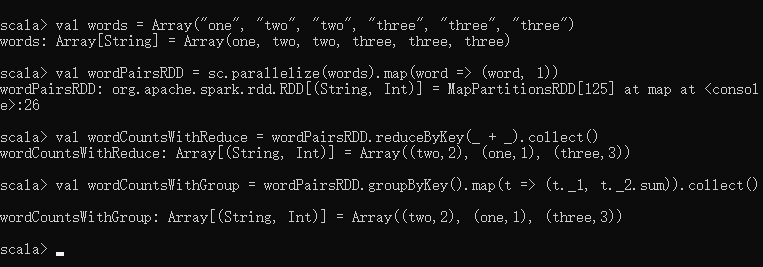
根据映射函数f，对RDD中的元素进行二元计算，返回计算结果。

|  |
| --- |
| scala> var rdd1 = sc.makeRDD(1 to 10,2)  rdd1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[36] at makeRDD at :21    scala> rdd1.reduce(\_ + \_)  res18: Int = 55    scala> var rdd2 = sc.makeRDD(Array(("A",0),("A",2),("B",1),("B",2),("C",1)))  rdd2: org.apache.spark.rdd.RDD[(String, Int)] = ParallelCollectionRDD[38] at makeRDD at :21  scala> rdd2.reduce((x,y) => {  (x.\_1 + y.\_1,x.\_2 + y.\_2)  })  res21: (String, Int) = (CBBAA,6)    val rdd3 =rdd2.map(line=>{  (line.\_1,line.\_2.size)  })  rdd3.foreach(println) |



## 又一例

|  |
| --- |
| val words = Array("one", "two", "two", "three", "three", "three")  val wordPairsRDD = sc.parallelize(words).map(word => (word, 1))  val wordCountsWithReduce = wordPairsRDD.reduceByKey(\_ + \_).collect()  val wordCountsWithGroup = wordPairsRDD.groupByKey().map(t => (t.\_1, t.\_2.sum)).collect() |

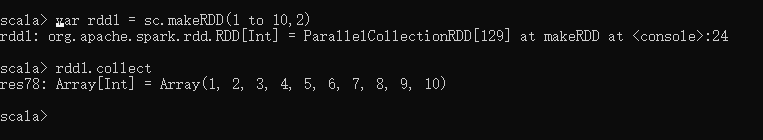


## collect

def collect(): Array[T]

collect用于将一个RDD转换成数组。

|  |
| --- |
| scala> var rdd1 = sc.makeRDD(1 to 10,2)  rdd1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[36] at makeRDD at :21    scala> rdd1.collect  res23: Array[Int] = Array(1, 2, 3, 4, 5, 6, 7, 8, 9, 10) |



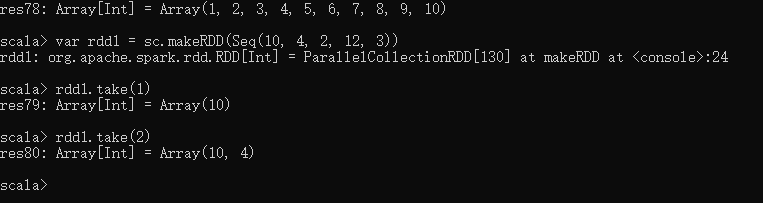
# Spark算子：RDD行动Action操作(2)–take、top、takeOrdered

## take

def take(num: Int): Array[T]

take用于获取RDD中从0到num-1下标的元素，不排序。

|  |
| --- |
| scala> var rdd1 = sc.makeRDD(Seq(10, 4, 2, 12, 3))  rdd1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[40] at makeRDD at :21    scala> rdd1.take(1)  res0: Array[Int] = Array(10)    scala> rdd1.take(2)  res1: Array[Int] = Array(10, 4) |

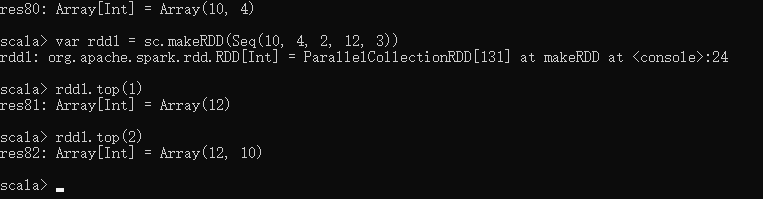


## top

def top(num: Int)(implicit ord: Ordering[T]): Array[T]

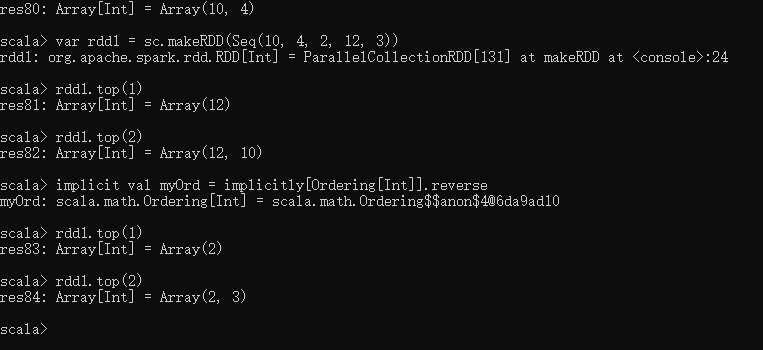
top函数用于从RDD中，按照默认（降序）或者指定的排序规则，返回前num个元素。

|  |
| --- |
| scala> var rdd1 = sc.makeRDD(Seq(10, 4, 2, 12, 3))  rdd1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[40] at makeRDD at :21    scala> rdd1.top(1)  res2: Array[Int] = Array(12)    scala> rdd1.top(2)  res3: Array[Int] = Array(12, 10) |



//指定排序规则

|  |
| --- |
| scala> implicit val myOrd = implicitly[Ordering[Int]].reverse  myOrd: scala.math.Ordering[Int] = scala.math.Ordering$$anon$4@767499ef    scala> rdd1.top(1)  res4: Array[Int] = Array(2)    scala> rdd1.top(2)  res5: Array[Int] = Array(2, 3) |

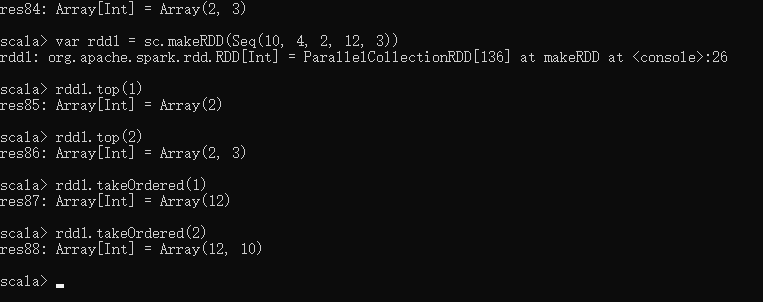


## takeOrdered

def takeOrdered(num: Int)(implicit ord: Ordering[T]): Array[T]

takeOrdered和top类似，只不过以和top相反的顺序返回元素。

|  |
| --- |
| scala> var rdd1 = sc.makeRDD(Seq(10, 4, 2, 12, 3))  rdd1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[40] at makeRDD at :21    scala> rdd1.top(1)  res4: Array[Int] = Array(2)    scala> rdd1.top(2)  res5: Array[Int] = Array(2, 3)    scala> rdd1.takeOrdered(1)  res6: Array[Int] = Array(12)    scala> rdd1.takeOrdered(2)  res7: Array[Int] = Array(12, 10) |

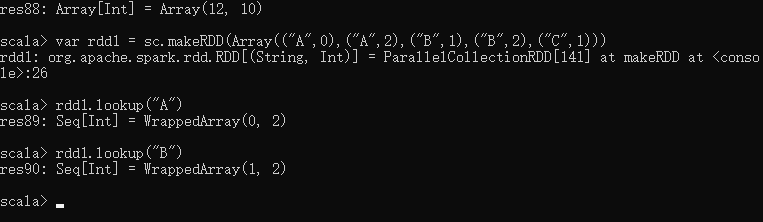


# Spark算子：RDD行动Action操作(3)–lookup

## lookup

lookup用于(K,V)类型的RDD,指定K值，返回RDD中该K对应的所有V值。

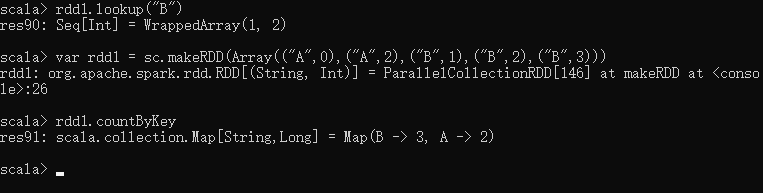
|  |
| --- |
| scala> var rdd1 = sc.makeRDD(Array(("A",0),("A",2),("B",1),("B",2),("C",1)))  rdd1: org.apache.spark.rdd.RDD[(String, Int)] = ParallelCollectionRDD[0] at makeRDD at :21    scala> rdd1.lookup("A")  res0: Seq[Int] = WrappedArray(0, 2)    scala> rdd1.lookup("B")  res1: Seq[Int] = WrappedArray(1, 2) |



# Spark算子：RDD行动Action操作(4)–countByKey、foreach、foreachPartition、sortBy

## countByKey

|  |
| --- |
| scala> var rdd1 = sc.makeRDD(Array(("A",0),("A",2),("B",1),("B",2),("B",3)))  scala> rdd1.countByKey  res5: scala.collection.Map[String,Long] = Map(A -> 2, B -> 3) |

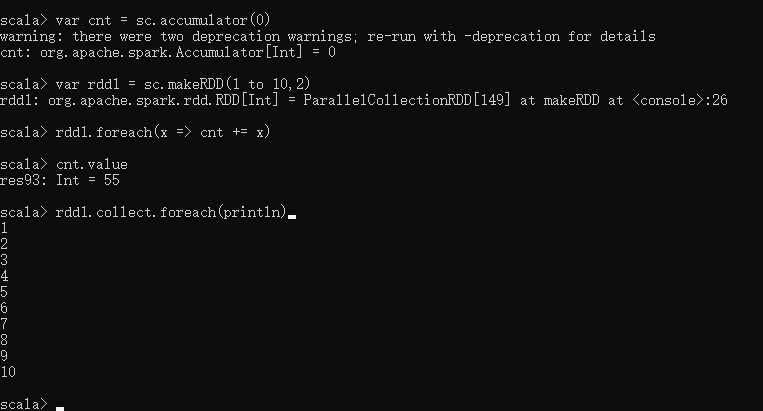


## foreach

def foreach(f: (T) ⇒ Unit): Unit

foreach用于遍历RDD,将函数f应用于每一个元素。

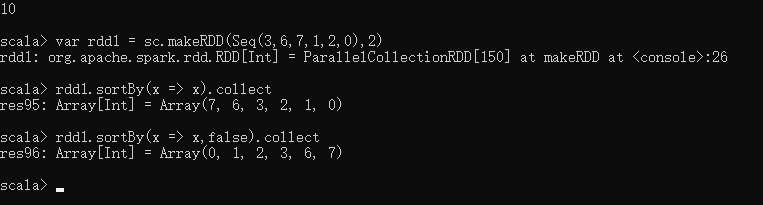
|  |
| --- |
| scala> var cnt = sc.accumulator(0)  cnt: org.apache.spark.Accumulator[Int] = 0    scala> var rdd1 = sc.makeRDD(1 to 10,2)  rdd1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[5] at makeRDD at :21  scala> rdd1.foreach(x => cnt += x)  scala> cnt.value  res51: Int = 55    scala> rdd1.collect.foreach(println) |



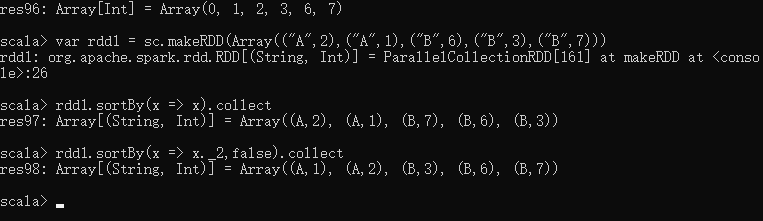
## sortBy

sortBy根据给定的排序k函数将RDD中的元素进行排序。

|  |
| --- |
| scala> var rdd1 = sc.makeRDD(Seq(3,6,7,1,2,0),2)    scala> rdd1.sortBy(x => x).collect  res1: Array[Int] = Array(0, 1, 2, 3, 6, 7) //默认升序    scala> rdd1.sortBy(x => x,false).collect  res2: Array[Int] = Array(7, 6, 3, 2, 1, 0) //降序 |



|  |
| --- |
| //RDD[K,V]类型  scala>var rdd1 = sc.makeRDD(Array(("A",2),("A",1),("B",6),("B",3),("B",7)))    scala> rdd1.sortBy(x => x).collect  res3: Array[(String, Int)] = Array((A,1), (A,2), (B,3), (B,6), (B,7))    //按照V进行降序排序  scala> rdd1.sortBy(x => x.\_2,false).collect  res4: Array[(String, Int)] = Array((B,7), (B,6), (B,3), (A,2), (A,1)) |



# Spark算子：RDD行动Action操作(5)–saveAsTextFile、saveAsSequenceFile、saveAsObjectFile

## saveAsTextFile

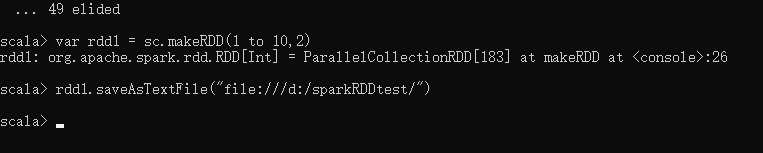
def saveAsTextFile(path: String): Unit

def saveAsTextFile(path: String, codec: Class[\_ <: CompressionCodec]): Unit

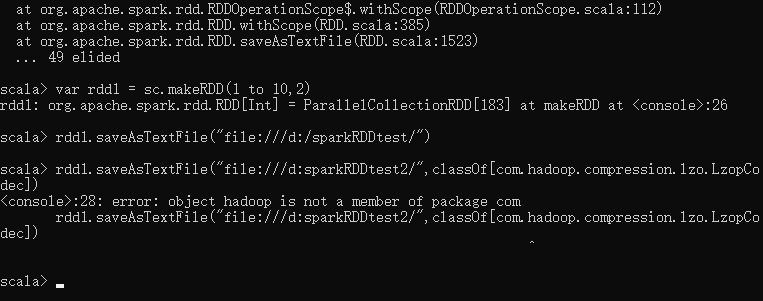
saveAsTextFile用于将RDD以文本文件的格式存储到文件系统中。

codec参数可以指定压缩的类名。

|  |
| --- |
| var rdd1 = sc.makeRDD(1 to 10,2)  **scala> rdd1.saveAsTextFile("hdfs://cdh5/tmp/lxw1234.com/") /**/保存到HDFS  注意：如果使用rdd1.saveAsTextFile(“file:///tmp/lxw1234.com”)将文件保存到本地文件系统，那么只会保存在Executor所在机器的本地目录。 |

|  |
| --- |
| //指定压缩格式保存  **rdd1.saveAsTextFile("hdfs://cdh5/tmp/lxw1234.com/",classOf[com.hadoop.compression.lzo.LzopCodec])** |



saveAsSequenceFile

saveAsSequenceFile用于将RDD以SequenceFile的文件格式保存到HDFS上。

用法同saveAsTextFile。

## saveAsObjectFile

def saveAsObjectFile(path: String): Unit

saveAsObjectFile用于将RDD中的元素序列化成对象，存储到文件中。

对于HDFS，默认采用SequenceFile保存。

|  |
| --- |
| var rdd1 = sc.makeRDD(1 to 10,2)  rdd1.saveAsObjectFile("hdfs://cdh5/tmp/lxw1234.com/") |