# RDD上的算子

RDD上的所有的算子示例：

<http://homepage.cs.latrobe.edu.au/zhe/ZhenHeSparkRDDAPIExamples.html>

## join leftOuterjoin rightOuterJoin cogroup

#join 只能作用于 k,v RDD[(Int,Int)]

相当于SQL中的内关联join，只返回两个RDD根据K可以关联上的结果

在类型为（K,V)和（K,W)类型的数据集上调用时，返回一个相同key对应的所有元素对在一起的(K, (V, W))数据集

leftOuterJoin

rightOuterJoin

val rdd3 = rdd1.leftOuterJoin(rdd2) RDD[(String, (V, Option[W]))]

val rdd3 = rdd1.rightOuterJoin(rdd2) RDD[(String, (Option[V], W))]

#cogroup

在类型为（K,V)和（K,W)的数据集上调用，返回一个(K, (Iterable[V], Iterable[W]))元组的数据集。这个操作也可以称之为groupwith

相当于SQL中的全外关联full outer join，返回左右RDD中的记录，关联不上的为空。

|  |
| --- |
| **val** sc: SparkContext = *MySpark*(**this**.getClass.getSimpleName)   *// K - V* **val** rdd1: RDD[(String, Double)] = sc.makeRDD(*List*((**"reba"**, 9000.0), (**"nazha"**, 8000.0), (**"ruhua"**, 10000.0)))   *// K - W* **val** rdd2: RDD[(String, Int)] = sc.makeRDD(*List*((**"reba"**, 7), (**"nazha"**, 8), (**"yangmi"**, 3)))   *// 默认分区数不变* **val** join: RDD[(String, (Double, Int))] = rdd1.join(rdd2)   *// join.foreach(println)   // RDD[(K, (V, Option[W]))] 右边可能关联不上* **val** result1: RDD[(String, (Double, Option[Int]))] = rdd1.leftOuterJoin(rdd2)  *// 求当月的出场费总额* result1.mapValues(tp => tp.\_2.getOrElse(0) \* tp.\_1)   *// .foreach(println)    println*(**"----------------------"**)  *// RDD[(K,(Option[V],W))]* rdd1.rightOuterJoin(rdd2)  *//.foreach(println)    // cogroup* **val** cogroup: RDD[(String, (Iterable[Double], Iterable[Int]))] = rdd1.cogroup(rdd2)  *// cogroup.foreach(println)* cogroup.mapValues(tp=>{  tp.\_1.sum \* tp.\_2.sum  }).foreach(*println*) |

## 笛卡尔积

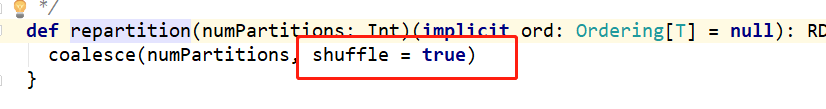
cartesian笛卡尔积

|  |
| --- |
| scala> val rdd1 = sc.makeRDD(List("tom","cat","jim"))  rdd1: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[0] at makeRDD at <console>:24  scala> val rdd2 = sc.makeRDD(List(1,3))  rdd2: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[1] at makeRDD at <console>:24  scala> val rdd3 = rdd1.ca  cache canEqual cartesian  scala> val rdd3 = rdd1.cartesian(rdd2)  rdd3: org.apache.spark.rdd.RDD[(String, Int)] = CartesianRDD[2] at cartesian at <console>:28  scala> rdd3.collect  [Stage 0:> (0 + 14) / 14[Stage 0:==> (6 + 13) / 14[Stage 0:=================================> (87 + 12) / 14[Stage 0:================================================> (130 + 12) / 14 res0: Array[(String, Int)] = Array((tom,1), (tom,3), (cat,1), (cat,3), (jim,1), (jim,3)) |

## 修改分区数量的算子

repartition(分区数量)

coalesce(分区数量)





repartition会对数据进行重写的分发（shuffle） 同一个分区的数据，会被分发到不同的分区中去。

coalesce 默认是没有进行shuffle的，所以当用coalesce来扩大分区的数量，是失败的，分区数量不变。

|  |
| --- |
| scala> val rdd1 = sc.makeRDD(List(1,3,5,6,7,8),3)  rdd1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[3] at makeRDD at <console>:24  scala> rdd1.repartition(1)  res1: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[7] at repartition at <console>:27  scala> rdd1.repartition(2)  res2: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[11] at repartition at <console>:27  scala> rdd1.coalesce(2)  res3: org.apache.spark.rdd.RDD[Int] = CoalescedRDD[12] at coalesce at <console>:27  scala> res2.partitions.size  res4: Int = 2  scala> res3.partitions.size  res5: Int = 2  scala> val f =(i:Int,it:Iterator[Int])=>  | it.map(t=> s"p=$i,v=$t")  f: (Int, Iterator[Int]) => Iterator[String] = <function2>  scala> res2.mapPartitionsWithIndex(f).collect  res6: Array[String] = Array(p=0,v=5, p=0,v=1, p=0,v=7, p=1,v=3, p=1,v=8, p=1,v=6)  scala> res3.mapPartitionsWithIndex(f).collect  res7: Array[String] = Array(p=0,v=1, p=0,v=3, p=1,v=5, p=1,v=6, p=1,v=7, p=1,v=8)  scala> rdd1.repartition(10)  res8: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[18] at repartition at <console>:27  scala> rdd1.coalesce(10)  res9: org.apache.spark.rdd.RDD[Int] = CoalescedRDD[19] at coalesce at <console>:27  scala> res8.partitions.size  res10: Int = 10  scala> res9.partitions.size  res11: Int = 3 |

总结：

repartition(10) = rdd1.coalesce(10,true)

repartition会对数据进行重新的shuffle。 coalesce主要用于合并分区，不会进行数据的shuffle。

实际使用：

如果数据需要shuffle，选择 repartition。

repartition 常用于 扩大分区数量。 提升任务的并行度。

coalesce常用于合并分区（**减少分区数量**） 不能用于扩大分区数量。除非加shuffle为true。

|  |
| --- |
| rdd1.sortByKey(**false**).coalesce(1).foreach(*println*) |

## aggregate,aggregateByKey

aggregate 是action算子，aggregateByKey 是转换算子。



第一个参数，是初始值。 初始值 ，参与分区内聚合， 还参与全局聚合

第二个参数： 是两个函数参数。第一个函数，表示分区内聚合，第二个函数，全局聚合。

|  |
| --- |
| scala> val rdd1 = sc.makeRDD(List(1,3,4,5),2)  rdd1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[24] at makeRDD at <console>:24  scala> rdd1.aggregate(0)(\_+\_,\_+\_)  res14: Int = 13  scala> rdd1.aggregate(0)(\_+\_,\_+\_)  res15: Int = 13  scala> rdd1.aggregate(10)(\_+\_,\_+\_)  res16: Int = 43  scala> rdd1.aggregate(10)((a,b)=>math.max(a,b),\_+\_)  res17: Int = 30  scala> val rdd2 = sc.parallelize(List("a","b","c","d","e","f"),2)  rdd2: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[25] at parallelize at <console>:24  scala> rdd2.aggregate("")(\_ ++ \_, \_ ++ \_)  res18: String = defabc  scala> rdd2.aggregate("")(\_ ++ \_, \_ ++ \_)  res19: String = abcdef |

aggregateByKey：

转换类的算子



第一个参数：初始值 初始值 只参与分区聚合

第二个参数： 两个函数 第一个是分区聚合函数 第二个函数 是全局聚合函数

|  |
| --- |
| scala> val pairRDD = sc.parallelize(List( ("cat",2), ("cat", 5), ("mouse", 4),("cat", 12), ("dog", 12), ("mouse", 2)), 2)  pairRDD: org.apache.spark.rdd.RDD[(String, Int)] = ParallelCollectionRDD[26] at parallelize at <console>:24  scala> pairRDD.aggregate  aggregate aggregateByKey  scala> pairRDD.aggregateByKey(0)(\_+\_,\_+\_)  res20: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[27] at aggregateByKey at <console>:27  scala> res20.collect  res21: Array[(String, Int)] = Array((dog,12), (cat,19), (mouse,6))  scala> pairRDD.aggregateByKey(10)(\_+\_,\_+\_).collect  res22: Array[(String, Int)] = Array((dog,22), (cat,39), (mouse,26))  scala> cat 17 mouse 14 dog 22 cat 22 mouse 12 |

## 算子的总结

转换类的算子：

RDD之间的依赖

普通的算子： map fliter flatMap

数据是一对一的，

shuffle类的算子：

分区内的数据会进行重新的分发

reduceByKey join distinct

action类的算子：

算子，作用于RDD[k]

算子： 必须作用于 RDD[K,V]

# 全局TopK和分组的TopK

## 全局TopK

|  |
| --- |
| **object** FacTeacher {   *// 取Top3* **val** *topN* = 3   **def** main(args: Array[String]): Unit = {  **val** sc = *MySpark*(**this**.getClass.getSimpleName)   *// 1,读数据* **val** file: RDD[String] = sc.textFile(**"f:/mrdata/teacher.log"**)   *// 2，数据预处理 提取出 学科名称 teachername* **val** splitRdd: RDD[((String, String), Int)] = file.map(str => {  *// 协议 host:port URL  // http://bigdata.edu360.cn/laozhang* **val** index: Int = str.lastIndexOf(**"/"**)   *// 获取老师的名称* **val** tName = str.substring(index + 1)   **val** url: URL = **new** URL(str.substring(0, index))  *// 获取hostName* **val** host = url.getHost() *// bigdata.edu360.cn  // 分隔符需要进行转义* **val** split = host.split(**"\\."**)  *// 获取到学科名称* **val** subject = split(0)   *// 把数据组装成嵌套元组* ((subject, tName), 1)  })    *// 分组聚合* **val** result: RDD[((String, String), Int)] = splitRdd.reduceByKey(\_ + \_)   *// result.foreach(println)  // 取topk* **val** finRes: Array[((String, String), Int)] = result.sortBy(\_.\_2, **false**).take(*topN*)   finRes.foreach(*println*)  sc.stop()  } } |

## 分组的TopK-groupBy

|  |
| --- |
| **object** SubFacTeacher {   *// 取Top3* **val** *topN* = 2   **def** main(args: Array[String]): Unit = {  **val** sc = *MySpark*(**this**.getClass.getSimpleName)   *// 1,读数据* **val** file: RDD[String] = sc.textFile(**"f:/mrdata/teacher.log"**)   *// 2，数据预处理 提取出 学科名称 teachername* **val** splitRdd: RDD[((String, String), Int)] = file.map(str => {  *// 协议 host:port URL  // http://bigdata.edu360.cn/laozhang* **val** index: Int = str.lastIndexOf(**"/"**)   *// 获取老师的名称* **val** tName = str.substring(index + 1)   **val** url: URL = **new** URL(str.substring(0, index))  *// 获取hostName* **val** host = url.getHost() *// bigdata.edu360.cn  // 分隔符需要进行转义* **val** split = host.split(**"\\."**)  *// 获取到学科名称* **val** subject = split(0)   *// 把数据组装成嵌套元组* ((subject, tName), 1)  })    *// 分组聚合* **val** result: RDD[((String, String), Int)] = splitRdd.reduceByKey(\_ + \_)   *// 再根据学科进行分组 ,之后，进行组内聚合* **val** groupRdd: RDD[(String, Iterable[((String, String), Int)])] = result.groupBy(\_.\_1.\_1)   **val** result2: RDD[(String, List[((String, String), Int)])] = groupRdd.mapValues(it => {  *// 这里的sortBy 是本地集合的API* it.toList.sortBy(-\_.\_2).take(*topN*)  })   **val** result3 = result2.mapValues(lst => {   lst.map { **case** ((\_, tName), cnts) => (tName, cnts) }  })  result3.foreach(*println*)    sc.stop()  } } |

## 分组的topK—过滤实现

|  |
| --- |
| **object** SubFacTeacher2 {   *// 取Top3* **val** *topN* = 2   **def** main(args: Array[String]): Unit = {  **val** sc = *MySpark*(**this**.getClass.getSimpleName)   *// 1,读数据* **val** file: RDD[String] = sc.textFile(**"f:/mrdata/teacher.log"**)   *// 2，数据预处理 提取出 学科名称 teachername* **val** splitRdd: RDD[((String, String), Int)] = file.map(str => {  *// 协议 host:port URL  // http://bigdata.edu360.cn/laozhang* **val** index: Int = str.lastIndexOf(**"/"**)   *// 获取老师的名称* **val** tName = str.substring(index + 1)   **val** url: URL = **new** URL(str.substring(0, index))  *// 获取hostName* **val** host = url.getHost() *// bigdata.edu360.cn  // 分隔符需要进行转义* **val** split = host.split(**"\\."**)  *// 获取到学科名称* **val** subject = split(0)   *// 把数据组装成嵌套元组* ((subject, tName), 1)  })    *// 提取出所有的学科数据* **val** subjectArr: Array[String] = splitRdd.map(\_.\_1.\_1).distinct().collect()    *// 可以做遍历 代码的耦合性高  // for (i <- Array("php", "javaee", "bigdata")) {* **for** (i <- subjectArr) {  *// php 过滤* **val** phpSubject = splitRdd.filter(t => i.equals(t.\_1.\_1))   *// 分组聚合* **val** phpRes: Array[((String, String), Int)] = phpSubject.reduceByKey(\_ + \_).sortBy(-\_.\_2).take(*topN*)   phpRes.foreach(*println*)  }   sc.stop()  } } |

# AB数据集的聚合

补充：

数据的切分：

|  |
| --- |
| **val** str = **"a,b,,," val** splits = str.split(**","**) *// 2 println*(splits.size) *// 2* **val** sp = str.split(**","**, 5) *// 是我们的给定数据str 有5个字段 println*(sp.size)  *// 按照字符串中，数据原有的字段进行切分* **val** split1 = str.split(**","**,-1) *println*(**s"split1=$**{split1.size}**"**)  *// 通过参数指定切分的长度为2* **val** split = str.split(**","**,2) *println*(split.size) *println*(split(0)) *// a println*(split(1)) *// b,,,* |

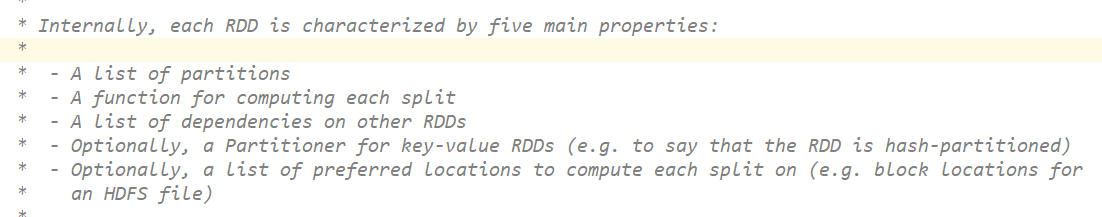
利用reduceByKey + leftOuterJoin实现AB数据集的聚合

|  |
| --- |
| **object** ABTest {  **def** main(args: Array[String]): Unit = {   **val** sc: SparkContext = *MySpark*(**this**.getClass.getSimpleName)   *// 读取数据  // u1 12 zs* **val** aFile = sc.textFile(**"f:/mrdata/joindata/a.txt"**)  *// u1 2016 9 m1* **val** bFile: RDD[String] = sc.textFile(**"f:/mrdata/joindata/b.txt"**)    *// 数据预处理 ---数据切分* **val** aSplitRdd: RDD[(String, String)] = aFile.map(str => {  **val** split = str.split(**" "**, 2)  **val** id = split(0)  **val** ageAndName = split(1)  *// 组成元组* (id, ageAndName)  })   **val** bSplitRdd: RDD[(String, List[String])] = bFile.map(str => {  **val** split = str.split(**" "**, 2)  **val** id = split(0)  **val** movie = split(1)  (id, *List*(movie))  })   *// 先对第二个数据进行进行处理 把相同key的数据聚合到一起* **val** breduceRdd: RDD[(String, List[String])] = bSplitRdd.reduceByKey(\_ ++ \_)   *// 对两个数据集进行 join* **val** abJoinRdd: RDD[(String, (String, Option[List[String]]))] = aSplitRdd.leftOuterJoin(breduceRdd)    **val** result: RDD[(String, String)] = abJoinRdd.mapValues {  **case** (ageAndName, movie) => {  *// 如果None 用 null null null* **val** movieData: String = movie **match** {  **case** None => **"null null null"** *// Some (v) => v 排序 ---》 拼接成字符串  // 按照年份的升序排序 ---> 拼接成字符串* **case** *Some*(lst) => lst.sortBy(\_.split(**" "**)(0).toInt).mkString(**" "**)  }  *// 拼接字符串* ageAndName + **","** + movieData  ageAndName.concat(**","**).concat(movieData)  }  }  *// 整理展示的结果格式* result.map { **case** (k, v) => k.concat(**" "**).concat(v) } foreach (*println*)    *// 都指定长度为2  /\*  (u3,Iterable("2012 3 m5"))  rddb.groupByKey()  rdda.leftOuterJoin(rddb)  rdda.cogroup(rddb)  rdda.union(rddb).groupByKey\*/    // 数据的聚合 \*\*\*   // 数据的整理输出* sc.stop()   } } |

# RDD上的5大特性

RDD是spark中的基本的计算模型。是一个抽象的概念。

RDD 抽象类



5大特性：

分别对应着RDD的成员属性或者方法。

## 分区列表：

RDD的数据集的基本组成单位。



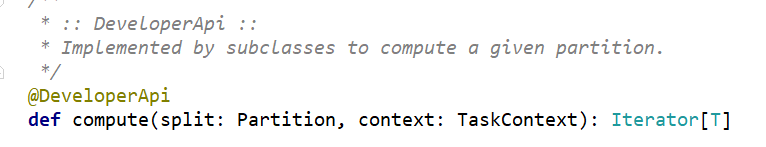
每一个RDD都有一到多个分区。

数据，

每一个分区，会记录读取的数据在哪里

## compute方法

计算方法。



用于计算的

MapPartitionsRDD中的compute 方法：



f: map((\_,1))

把父RDD中的数据，组装成iterator，然后直接传递到函数中。

**父RDD的数据到子RDD的数据的逻辑转换。**

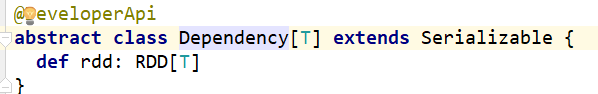
## RDD之间的依赖关系

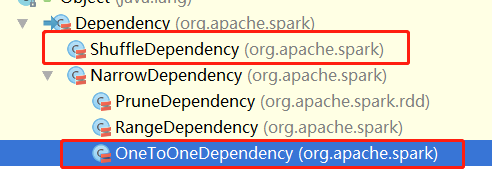


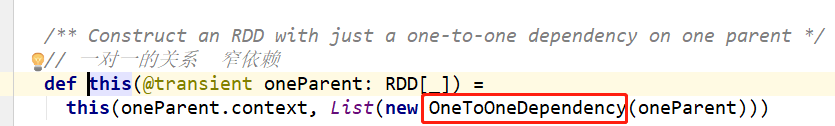
RDD之间的依赖关系分为两类：

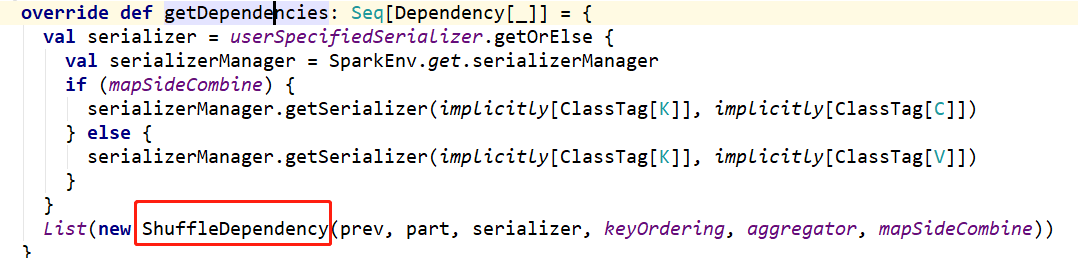
OneToOneDependency 窄依赖

ShuffleDependency 宽依赖







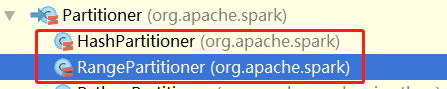


## 可选的，分区器



在RDD[K,V] 类型上才有分区器

RDD[K] 的分区器 None



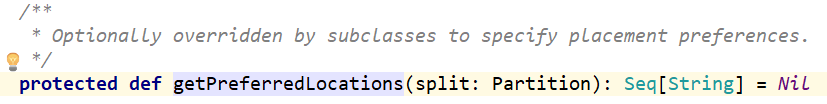
一种： HashPartitioner key 的hashcode % 分区数量

一种： RangePartitioner key 的范围 分区的数量

如果需要自定义分区器：

就定义类 ，继承 Partitioner，重写抽象方法即可。

## 可选的，优先位置



理念： **移动数据不如移动计算**。

**优先，在有数据的节点上，启动计算任务。**

分区列表，分区器。

依赖关系

compute方法，优先位置。

### RDD中的分区器

只有RDD[K,V] 才有分区器 RDD[K] 分区器是None

HashPartitioner

RangePartitioner

reduceByKey HashParititoner

sortBy sortByKey RangePartitioner

HashPartitioner的判断相同的标准： 1，是否是同一个分区器；2 分区的数量是否相同

|  |
| --- |
| scala> val rdd1 = sc.makeRDD(List(1,2,3,12,15,18,21,27),3)  rdd1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[0] at makeRDD at <console>:24  scala> val rdd2 = rdd1.zipWI  zipWithIndex zipWithUniqueId  scala> val rdd2 = rdd1.zipWithIndex  rdd2: org.apache.spark.rdd.RDD[(Int, Long)] = ZippedWithIndexRDD[1] at zipWithIndex at <console>:26  scala> val rdd3 = rdd2.groupByKey()  rdd3: org.apache.spark.rdd.RDD[(Int, Iterable[Long])] = ShuffledRDD[2] at groupByKey at <console>:28  scala> val f = (i:Int,it:Iterator[(Int,Long)])=>  | it.map(t=> s"p=$i,v=${t.\_1}")  f: (Int, Iterator[(Int, Long)]) => Iterator[String] = <function2>  scala> rdd2.mapPartitionsWithIndex(f).collect  res0: Array[String] = Array(p=0,v=1, p=0,v=2, p=1,v=3, p=1,v=12, p=1,v=15, p=2,v=18, p=2,v=21, p=2,v=27)  scala> val f2 = (i:Int,it:Iterator[(Int, Iterable[Long])])=>  | it.map(t => s"p=$i,v= ${t.\_1}")  f2: (Int, Iterator[(Int, Iterable[Long])]) => Iterator[String] = <function2>  scala> rdd3.mapPartitionsWithIndex(f2).collect  res1: Array[String] = Array(p=0,v= 21, p=0,v= 15, p=0,v= 27, p=0,v= 18, p=0,v= 3, p=0,v= 12, p=1,v= 1, p=2,v= 2)  scala>  scala>  scala> val rdd5 = sc.makeRDD(List(11,101,10001,1001,9,9,11,121,141,9,9,11,11,11),3)  rdd5: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[5] at makeRDD at <console>:24  scala>  scala>  scala> val rdd6 = rdd5.zipWithIndex  rdd6: org.apache.spark.rdd.RDD[(Int, Long)] = ZippedWithIndexRDD[6] at zipWithIndex at <console>:26  scala> val rdd7 = rdd6.sortByKey()  rdd7: org.apache.spark.rdd.RDD[(Int, Long)] = ShuffledRDD[9] at sortByKey at <console>:28  scala> rdd6.mapPartitionsWithIndex(f).collect  res2: Array[String] = Array(p=0,v=11, p=0,v=101, p=0,v=10001, p=0,v=1001, p=1,v=9, p=1,v=9, p=1,v=11, p=1,v=121, p=1,v=141, p=2,v=9, p=2,v=9, p=2,v=11, p=2,v=11, p=2,v=11)  scala> rdd7.mapPartitionsWithIndex(f).collect  res3: Array[String] = Array(p=0,v=9, p=0,v=9, p=0,v=9, p=0,v=9, p=0,v=11, p=0,v=11, p=0,v=11, p=0,v=11, p=0,v=11, p=1,v=101, p=2,v=121, p=2,v=141, p=2,v=1001, p=2,v=10001)  scala> val rdd7 = rdd6.sortByKey()  rdd7: org.apache.spark.rdd.RDD[(Int, Long)] = ShuffledRDD[14] at sortByKey at <console>:28  scala> rdd7.mapPartitionsWithIndex(f).collect  res4: Array[String] = Array(p=0,v=9, p=0,v=9, p=0,v=9, p=0,v=9, p=0,v=11, p=0,v=11, p=0,v=11, p=0,v=11, p=0,v=11, p=1,v=101, p=2,v=121, p=2,v=141, p=2,v=1001, p=2,v=10001) |

如果自定义分区器：

直接继承Partitioner，重写抽象方法。

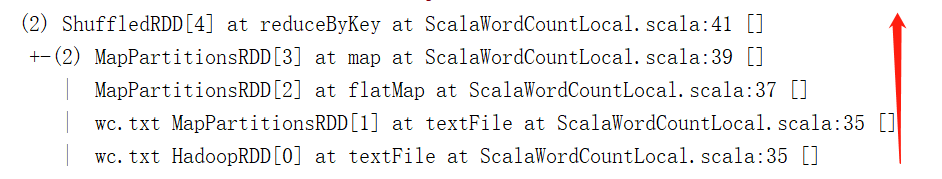
# 以wordcount为示例，查看spark任务的运行流程

## 一个wordcount中产生了几个rdd

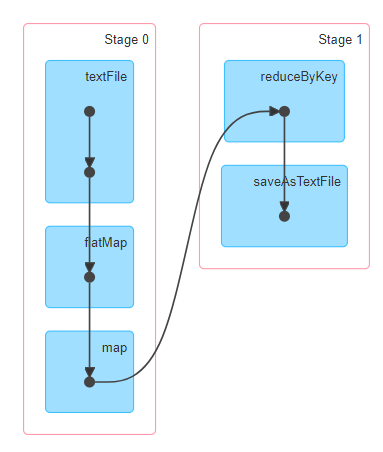
|  |
| --- |
| *// 创建SparkContext* **val** sc: SparkContext = **new** SparkContext(conf) *// 读取数据* **val** file: RDD[String] = sc.textFile(input) *// 切分并压平* **val** words: RDD[String] = file.flatMap(\_.split(**" "**)) *// 组装* **val** wordAndOne: RDD[(String, Int)] = words.map((\_, 1)) *// 分组聚合* **val** result: RDD[(String, Int)] = wordAndOne.reduceByKey(\_ + \_) *// 直接存储到hdfs中* result.saveAsTextFile(output) *// 释放资源* sc.stop() |

问题：

最简版的wordcount，一共产生了几个RDD?



这个wordcount的任务，一共产生了6个RDD



可以通过toDebugString 方法来查看rdd的依赖关系图。

*// 可以查看RDD的依赖关系  
println*(result.toDebugString)

## spark任务运行的数据流向图

|  |
| --- |
| scala> val rdd1 = sc.textFile("hdfs://hdp-01:9000/wordcount/input")  rdd1: org.apache.spark.rdd.RDD[String] = hdfs://hdp-01:9000/wordcount/input MapPartitionsRDD[1] at textFile at <console>:24  scala> val rdd2 = rdd1.flatMap(\_.split(" ")).map((\_,1))  rdd2: org.apache.spark.rdd.RDD[(String, Int)] = MapPartitionsRDD[3] at map at <console>:26  scala> val rdd3 = rdd2.reduceByKey(\_+\_,2)  rdd3: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[4] at reduceByKey at <console>:28  scala> rdd3.saveAsTextFile("hdfs://hdp-01:9000/wordcount/output11") |

逻辑图

## 任务调度的流程：

application🡪 job -🡪 stage 🡪 task

## 物理执行流程：

当执行我们自定义程序（main）的时候，并没有真正的执行程序。

Driver会记录rdd之间的依赖关系，每一个rdd传递了什么函数，记录rdd要去哪里读取数据。

当触发action算子的时候，真正开始执行任务。

DAG 🡪 切分stage -🡪 组装Task 🡪 把Task提交给executor去执行。

任务正式的开始按照我们的业务逻辑执行。

spark-submit 开始

今日总结：

rdd算子

rdd上的5大特性

分区器 –》 自定义分区器

