# Transformations 和action常见算子

## Transformations 算子：

### 1）map（根据输入参数（函数）返回新的RDD）

将原来 RDD 的每个数据项通过 map 中的用户自定义函数 f 映射转变为一个新的元素。源码中 map 算子相当于初始化一个 RDD， 新 RDD 叫做 MappedRDD(this, sc.clean(f))。

     图 1中每个方框表示一个 RDD 分区，左侧的分区经过用户自定义函数 f:T->U 映射为右侧的新 RDD 分区。但是，实际只有等到 Action算子触发后，这个 f 函数才会和其他函数在一个stage 中对数据进行运算。在图 1 中的第一个分区，数据记录 V1 输入 f，通过 f 转换输出为转换后的分区中的数据记录 V’1。

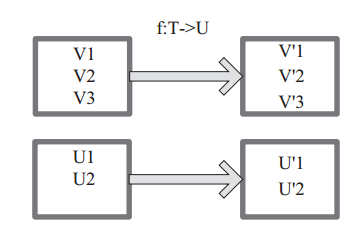


图1  map 算子对 RDD 转换

|  |
| --- |
| scala> **val rdd1 = sc.parallelize(Array("a","a","a","b","b","c"))**  rdd1: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[28] at parallelize at <console>:24  scala> **rdd1.map(x=>(x,1)).collect**  res2: Array[(String, Int)] = Array((a,1), (a,1), (a,1), (b,1), (b,1), (c,1))  scala> **rdd1.map((\_,1)).collect**  res5: Array[(String, Int)] = Array((a,1), (a,1), (a,1), (b,1), (b,1), (c,1)) |

### flatMap（压平）

     将原来 RDD 中的每个元素通过函数 f 转换为新的元素，并将生成的 RDD 的每个集合中的元素合并为一个集合，内部创建 FlatMappedRDD(this，sc.clean(f))。

图 2 表 示 RDD 的 一 个 分 区 ，进 行 flatMap函 数 操 作， flatMap 中 传 入 的 函 数 为 f:T->U， T和 U 可以是任意的数据类型。将分区中的数据通过用户自定义函数 f 转换为新的数据。外部大方框可以认为是一个 RDD 分区，小方框代表一个集合。 V1、 V2、 V3 在一个集合作为 RDD 的一个数据项，可能存储为数组或其他容器，转换为V’1、 V’2、 V’3 后，**将原来的数组或容器结合拆散**，**拆散的数据形成为 RDD 中的数据项**。

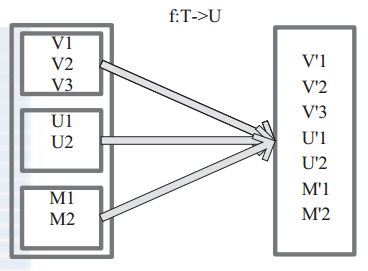


图2    flatMap 算子对 RDD 转换

|  |
| --- |
| **scala> val rdd1 = sc.parallelize(Array("hello world","hello spark","hello scala","hello hive","hello hadoop","hello hbase"))**  **rdd1: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[39] at parallelize at <console>:24**  **scala> rdd1.collect**  **res11: Array[String] = Array(hello world, hello spark, hello scala, hello hive, hello hadoop, hello hbase)**  **scala> rdd1.flatMap(x=>x.split(" ")).collect**  **res12: Array[String] = Array(hello, world, hello, spark, hello, scala, hello, hive, hello, hadoop, hello, hbase)**  **scala> rdd1.flatMap(\_.split(" ")).collect**  **res13: Array[String] = Array(hello, world, hello, spark, hello, scala, hello, hive, hello, hadoop, hello, hbase)** |

### 3）mapPartitions（对分区进行map操作）

      mapPartitions 函数获取到每个分区的迭代器，在函数中通过这个分区整体的迭代器对整个分区的元素进行操作。内部实现是生成MapPartitionsRDD。图3中的方框代表一个RDD分区。图3中，用户通过函数f(iter)=>iter.filter(\_>=3)对分区中所有数据进行过滤，大于和等于3的数据保留。一个方块代表一个RDD分区，含有1、2、3的分区过滤只剩下元素3。

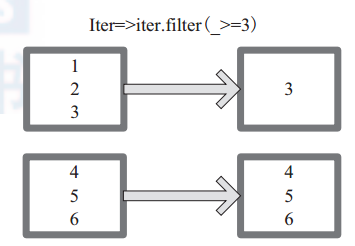


图3  mapPartitions 算子对 RDD 转换

mapPartitions是map的一个变种，它们都可进行分区的并行处理。

    两者的主要区别是调用的粒度不一样：map的输入变换函数是应用于RDD中每个元素，而mapPartitions的输入函数是应用于每个分区。

假设一个rdd有10个元素，分成3个分区。如果使用map方法，map中的输入函数会被调用10次；而使用mapPartitions方法的话，其输入函数会只会被调用3次，每个分区调用1次。

|  |
| --- |
| //生成10个元素3个分区的rdd a，元素值为1~10的整数（1 2 3 4 5 6 7 8 9 10），sc为SparkContext对象  val a = sc.parallelize(1 to 10, 3)  #查看该rdd的分区数量  a.partitions.length  //定义两个输入变换函数，它们的作用均是将rdd a中的元素值翻倍  //map的输入函数，其参数e为rdd元素值  def myfuncPerElement(e:Int):Int = {  println("e="+e)  e\*2  } |
| //mapPartitions的输入函数。iter是分区中元素的迭代子，返回类型也要是迭代子  def myfuncPerPartition ( iter : Iterator [Int] ) : Iterator [Int] = {  println("run in partition")  var res = for (e <- iter ) yield e\*2  res  }  val b = a.map(myfuncPerElement).collect  val c = a.mapPartitions(myfuncPerPartition).collect |

在**[Spark](http://lib.csdn.net/base/spark" \o "Apache Spark知识库" \t "https://blog.csdn.net/xingzhiqing/article/details/_blank)** shell中运行上述代码，可看到打印了3次run in partition，打印了10次e=。

 从输入函数（myfuncPerElement、myfuncPerPartition）层面来看，map是推模式，数据被推到myfuncPerElement中；mapPartitons是拉模式，myfuncPerPartition通过迭代子从分区中拉数据。

  这两个方法的另一个区别是在[大数据](http://lib.csdn.net/base/hadoop" \o "Hadoop知识库" \t "https://blog.csdn.net/xingzhiqing/article/details/_blank)集情况下的资源初始化开销和批处理处理，如果在myfuncPerPartition和myfuncPerElement中都要初始化一个耗时的资源，然后使用，比如[数据库](http://lib.csdn.net/base/mysql" \o "MySQL知识库" \t "https://blog.csdn.net/xingzhiqing/article/details/_blank)连接。在上面的例子中，myfuncPerPartition只需初始化3个资源（3个分区每个1次），而myfuncPerElement要初始化10次（10个元素每个1次），显然在大数据集情况下（数据集中元素个数远大于分区数），mapPartitons的开销要小很多，也便于进行批处理操作。

 mapPartitionsWithIndex和mapPartitons类似，只是其参数多了个分区索引号。

|  |
| --- |
| mapPartitionsWithIndex : 把每个partition中的分区号和对应的值拿出来, 看源码  val func = (index: Int, iter: Iterator[(Int)]) => {  iter.toList.map(x => "[partID:" + index + ", val: " + x + "]").iterator  }  val rdd1 = sc.parallelize(List(1,2,3,4,5,6,7,8,9), 2)  #查看该rdd的分区数量  rdd1.partitions.length  rdd1.mapPartitionsWithIndex(func).collect |

### 4）glom

　　glom函数将每个分区形成一个数组，内部实现是返回的GlommedRDD。 图4中的每个方框代表一个RDD分区。图4中的方框代表一个分区。 该图表示含有V1、 V2、 V3的分区通过函数glom形成一数组Array[（V1），（V2），（V3）]。

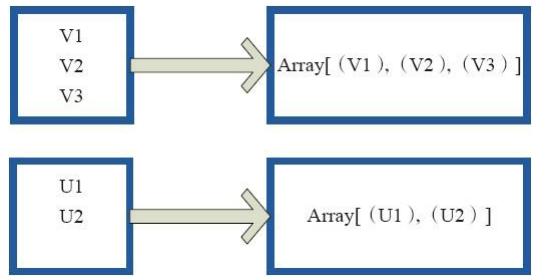
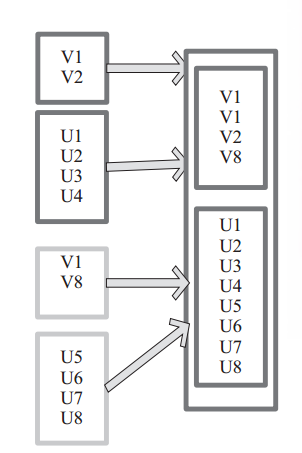


　图 4   glom算子对RDD转换

### union（合并）

      **使用union函数时需要保证两个RDD元素的数据类型相同，返回的RDD数据类型和被合并的RDD元素数据类型相同，并不进行去重操作，保存所有元素。如果想去重可以使用distinct()。同时Spark还提供更为简洁的使用union的API，通过++符号相当于union函数操作。**  
   图5中左侧大方框代表两个RDD，大方框内的小方框代表RDD的分区。右侧大方框代表合并后的RDD，大方框内的小方框代表分区。

含有V1、V2、U1、U2、U3、U4的RDD和含有V1、V8、U5、U6、U7、U8的RDD合并所有元素形成一个RDD。V1、V1、V2、V8形成一个分区，U1、U2、U3、U4、U5、U6、U7、U8形成一个分区。  
  
　　图5 union算子对RDD转换

|  |
| --- |
| scala> val rdd01 = sc.makeRDD(List(1,3,5,3))  rdd01: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[53] at makeRDD at <console>:24  scala> val rdd02 = sc.makeRDD(List(2,4,5,1))  rdd02: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[54] at makeRDD at <console>:24  scala> rdd01.union(rdd02).collect()  res27: Array[Int] = Array(1, 3, 5, 3, 2, 4, 5, 1)  scala> rdd01.union(rdd02).distinct.collect()  res31: Array[Int] = Array(4, 1, 5, 2, 3) |

### 6）cartesian（笛卡尔积）

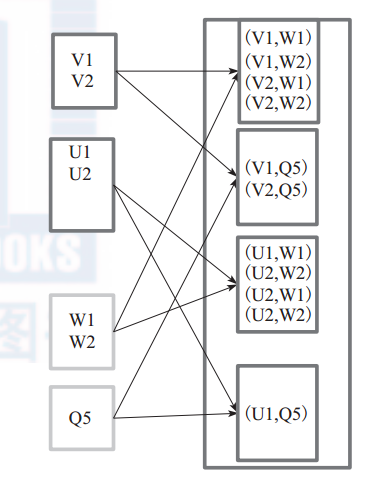
       对两个RDD内的所有元素 进行笛卡尔积操作。操作后，内部实现返回CartesianRDD。图6中左侧大方框代表两个RDD，大方框内的小方框代表RDD的分区。右侧大方框代表合并后的RDD，大方框内的小方框代表分区。图6中的大方框代表RDD，大方框中的小方框代表RDD分区。  
   例如：V1和另一个RDD中的W1、W2、Q5进行笛卡尔积运算形成(V1,W1)、(V1,W2)、(V1,Q5)。  
   

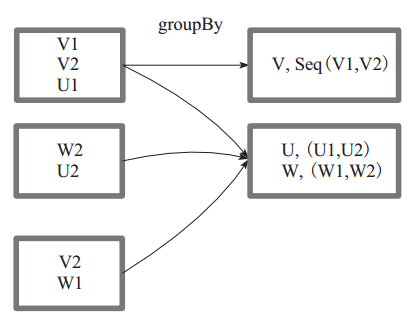
    图6 cartesian算子对RDD转换

|  |
| --- |
| **scala> val rdd01 = sc.makeRDD(List(1,3,5,3))**  **rdd01: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[81] at makeRDD at <console>:24**  **scala> val rdd02 = sc.makeRDD(List(2,4,5,1))**  **rdd02: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[82] at makeRDD at <console>:24**  **scala> rdd01.collect**  **res38: Array[Int] = Array(1, 3, 5, 3)**  **scala> rdd02.collect**  **res39: Array[Int] = Array(2, 4, 5, 1)**  **scala> rdd01.cartesian(rdd02).collect()**  **res37: Array[(Int, Int)] = Array((1,2), (1,4), (3,2), (3,4), (1,5), (1,1), (3,5), (3,1), (5,2), (5,4), (3,2), (3,4), (5,5), (5,1), (3,5), (3,1))** |

### 7）groupBy（根据key进行分组）

　　groupBy：将元素通过函数生成相应的Key，数据就转化为Key-Value格式，之后将Key相同的元素分为一组。  
　　函数实现如下：  
　　1）将用户函数预处理：  
　　val cleanF=sc.clean(f)  
　　2）对数据map进行函数操作，最后再进行groupByKey分组操作。

   this.map(t=>(cleanF(t),t)).groupByKey(p)  
　　其中，p确定了分区个数和分区函数，也就决定了并行化的程度。

　　图7中方框代表一个RDD分区，相同key的元素合并到一个组。例如V1和V2合并为V，Value为V1,V2。形成V,Seq(V1,V2)。  
  
图7 groupBy算子对RDD转换

|  |
| --- |
| **scala> val rdd1 = sc.parallelize(List(("tom", 1), ("jerry", 2), ("kitty", 3)))**  **rdd1: org.apache.spark.rdd.RDD[(String, Int)] = ParallelCollectionRDD[0] at parallelize at <console>:24**  **scala> val rdd2 = sc.parallelize(List(("jerry", 9), ("tom", 8), ("shuke", 7)))**  **rdd2: org.apache.spark.rdd.RDD[(String, Int)] = ParallelCollectionRDD[1] at parallelize at <console>:24**  **scala> val rdd3 = rdd1 union rdd2**  **rdd3: org.apache.spark.rdd.RDD[(String, Int)] = UnionRDD[2] at union at <console>:28**  **scala> rdd3.groupByKey**  **res0: org.apache.spark.rdd.RDD[(String, Iterable[Int])] = ShuffledRDD[3] at groupByKey at <console>:31**  **scala> rdd3.groupByKey.collect**  **res1: Array[(String, Iterable[Int])] = Array((tom,CompactBuffer(1, 8)), (jerry,CompactBuffer(2, 9)), (shuke,CompactBuffer(7)), (kitty,CompactBuffer(3)))**  **scala> rdd3.groupByKey.map(x=>(x.\_1,x.\_2.sum))**  **res2: org.apache.spark.rdd.RDD[(String, Int)] = MapPartitionsRDD[6] at map at <console>:31**  **scala> rdd3.groupByKey.map(x=>(x.\_1,x.\_2.sum)).collect**  **res3: Array[(String, Int)] = Array((tom,9), (jerry,11), (shuke,7), (kitty,3))** |

#WordCount, 第二个效率低

sc.textFile("/root/words.txt").flatMap(x=>x.split(" ")).map((\_,1)).reduceByKey(\_+\_).sortBy(\_.\_2,false).collect

sc.textFile("/root/words.txt").flatMap(x=>x.split(" ")).map((\_,1)).groupByKey.map(t=>(t.\_1, t.\_2.sum)).collect

### Filter（过滤）

    filter函数功能是对元素进行过滤，对每个元素应用f函数，返回值为true的元素在RDD中保留，返回值为false的元素将被过滤掉。内部实现相当于生成FilteredRDD(this，sc.clean(f))。  
  下面代码为函数的本质实现：  
   deffilter(f:T=>Boolean):RDD[T]=newFilteredRDD(this,sc.clean(f))  
　　图8中每个方框代表一个RDD分区，T可以是任意的类型。通过用户自定义的过滤函数f，对每个数据项操作，将满足条件、返回结果为true的数据项保留。例如，过滤掉V2和V3保留了V1，为区分命名为V’1。

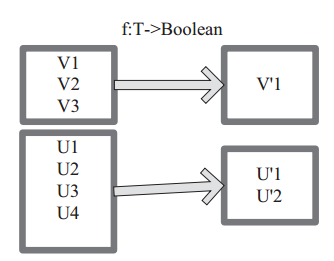


图8 filter算子对RDD转换

|  |
| --- |
| **# 或者 sc.parallelize((1 to 9).toList)**  **# 或者 sc.parallelize((1 to 9).toArray[Int])**  **# 或者 sc.parallelize(List(1,2,3,4,5,6,7,8,9))**  **# 或者 sc.parallelize(Array(1,2,3,4,5,6,7,8,9))**  **scala> val rdd1=sc.parallelize(Array(1,2,3,4,5,6,7,8,9))**  **1,2,3,4,5,6,7,8,9**  **scala> rdd1.filter(\_ > 3).collect().mkString(",").foreach(print)**  **4,5,6,7,8,9** |

### 9）Distinct（去重）

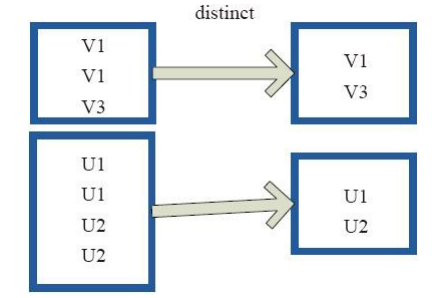
　　distinct将RDD中的元素进行去重操作。图9中的每个方框代表一个RDD分区，通过distinct函数，将数据去重。 例如，重复数据V1、 V1去重后只保留一份V1。  


　图9  distinct算子对RDD转换

|  |
| --- |
| **scala> val rdd1 = sc.parallelize(Array("a","b","c","d","b","a"), 2)**  **rdd1: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[14] at parallelize at <console>:24**  **scala> rdd1.distinct().collect**  **res6: Array[String] = Array(d, b, a, c)**  **scala> println(rdd1.distinct().collect().mkString(","))**  **d,b,a,c** |

### 10）subtract（求差集，在两个集合中都存在的，不会被纳入到新的RDD）

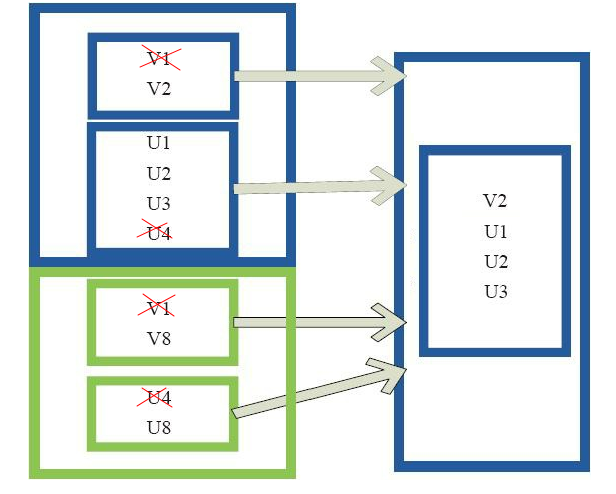
　　subtract相当于进行集合的差操作，RDD1去除RDD1和RDD2交集中的所有元素。图10中左侧的大方框代表两个RDD，大方框内的小方框代表RDD的分区。 右侧大方框代表合并后的RDD，大方框内的小方框代表分区。V1、U4在两个RDD中均有，根据差集运算规则，新RDD不保留，V2、U1、U2、U3在第一个RDD有，第二个RDD没有，则在新RDD元素中包含V2、U1、U2、U3。  
　　

　　图10   subtract算子对RDD转换

|  |
| --- |
| **scala> val rdd1=sc.parallelize(Array("a","b","c","d"))**  **rdd1: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[25] at parallelize at <console>:24**  **scala> val rdd2=sc.parallelize(Array("a","b","e","f"))**  **rdd2: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[26] at parallelize at <console>:24**  **scala> println(rdd1.subtract(rdd2).collect().mkString(","))**  **d,c**  **scala> println(rdd2.subtract(rdd1).collect().mkString(","))**  **f,e** |

说明：**rdd1.subtract(rdd2)和rdd2.subtract(rdd1) 取得的结果是不一样的**

**谁调用subtract方法，谁的元素去除相同的之后，就会被保留下来**

### 11）Sample

       sample 将 RDD 这个集合内的元素进行采样，获取所有元素的子集。用户可以设定是否有放回的抽样、百分比、随机种子，进而决定采样方式。内部实现是生成 SampledRDD(withReplacement， fraction， seed)。  
　　函数参数设置：  
‰ 　　withReplacement=true，表示有放回的抽样。  
‰ 　　withReplacement=false，表示无放回的抽样。  
　　图11中的每个方框是一个RDD分区。通过sample函数，采样50%的数据。V1、V2、U1、U2、U3、U4采样出数据V1和U1、U2形成新的RDD。

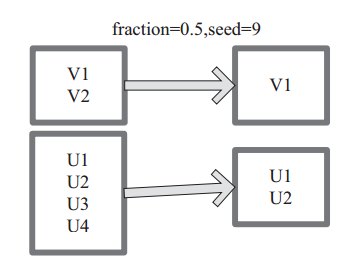


　图11 sample算子对RDD转换

### **12）takeSample**

　　takeSample函数和上面的sample函数是一个原理，但是不使用相对比例采样，而是按设定的采样个数进行采样，同时返回结果不再是RDD，而是相当于对采样后的数据进行  
Collect（），返回结果的集合为单机的数组。  
　　图12中左侧的方框代表分布式的各个节点上的分区，右侧方框代表单机上返回的结果数组。 通过takeSample对数据采样，设置为采样一份数据，返回结果为V1。

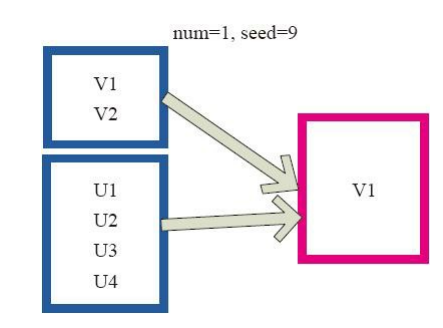


图12 takeSample算子对RDD转换

### Cache

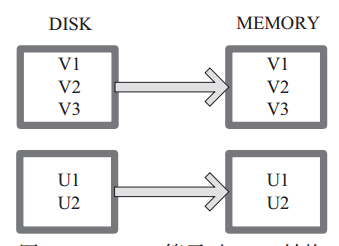
     cache 将 RDD 元素从磁盘缓存到内存。 相当于 persist(MEMORY\_ONLY) 函数的功能。  
     图13 中每个方框代表一个 RDD 分区，左侧相当于数据分区都存储在磁盘，通过 cache 算子将数据缓存在内存。  
      

　图 13 Cache 算子对 RDD 转换

### **Persist**

      persist 函数对 RDD 进行缓存操作。数据缓存在哪里依据 StorageLevel 这个枚举类型进行确定。 有以下几种类型的组合（见10）， DISK 代表磁盘，MEMORY 代表内存， SER 代表数据是否进行序列化存储。

　　下面为函数定义， StorageLevel 是枚举类型，代表存储模式，用户可以通过图 14-1 按需进行选择。  
　　persist(newLevel:StorageLevel)  
　　图 14-1 中列出persist 函数可以进行缓存的模式。例如，MEMORY\_AND\_DISK\_SER 代表数据可以存储在内存和磁盘，并且以序列化的方式存储，其他同理。  


　图 14-1  persist 算子对 RDD 转换

　　图 14-2 中方框代表 RDD 分区。 disk 代表存储在磁盘， mem 代表存储在内存。数据最初全部存储在磁盘，通过 persist(MEMORY\_AND\_DISK) 将数据缓存到内存，但是有的分区无法容纳在内存，将含有 V1、 V2、 V3 的RDD存储到磁盘，将含有U1，U2的RDD仍旧存储在内存。

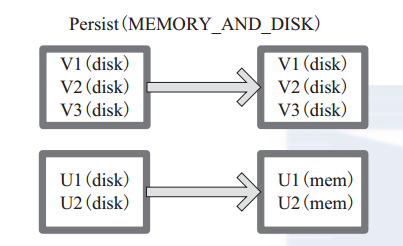


图 14-2   Persist 算子对 RDD 转换

### **mapValues**

     mapValues：针对（Key，Value）型数据中的Value进行Map操作，而不对Key进行处理。

    图15中的方框代表 RDD 分区。a=>a+2 代表对 (V1,1) 这样的 Key Value 数据对，数据只对Value中的 1 进行加 2 操作，返回结果为 3。

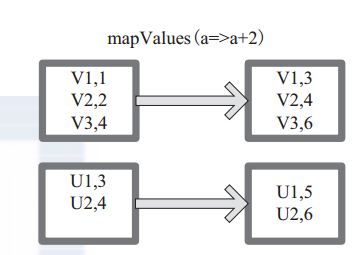


　图15 mapValues算子RDD对转换

### **combineByKey**

　　下面代码为 combineByKey 函数的定义：  
　　combineByKey[C](createCombiner:(V) C,  
　　mergeValue:(C, V) C,  
　　mergeCombiners:(C, C) C,  
　　partitioner:Partitioner,  
　　mapSideCombine:Boolean=true,  
　　serializer:Serializer=null):RDD[(K,C)]

说明：  
‰ 　　createCombiner： V => C， C 不存在的情况下，比如通过 V 创建 seq C。  
‰　　 mergeValue： (C， V) => C，当 C 已经存在的情况下，需要 merge，比如把 item V  
加到 seq C 中，或者叠加。  
　　 mergeCombiners： (C， C) => C，合并两个 C。  
‰ 　　partitioner： Partitioner, Shuff le 时需要的 Partitioner。  
‰ 　　mapSideCombine ： Boolean = true，为了减小传输量，很多 combine 可以在 map  
端先做，比如叠加，可以先在一个 partition 中把所有相同的 key 的 value 叠加，  
再 shuff le。  
‰ 　　serializerClass： String = null，传输需要序列化，用户可以自定义序列化类：

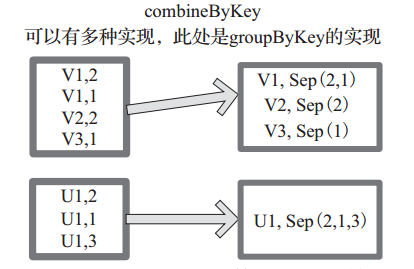
　　例如，相当于将元素为 (Int， Int) 的 RDD 转变为了 (Int， Seq[Int]) 类型元素的 RDD。图 16中的方框代表 RDD 分区。如图，通过 combineByKey， 将 (V1,2)， (V1,1)数据合并为（ V1,Seq(2,1)）。  
　　

　　图 16  comBineByKey 算子对 RDD 转换

### **reduceByKey**

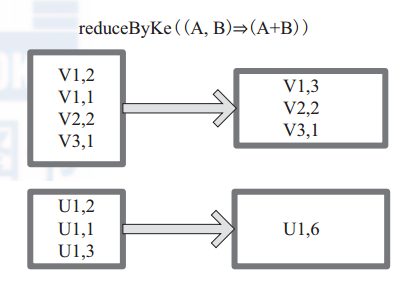
   reduceByKey是比combineByKey更简单的一种情况，只是两个值合并成一个值，（Int，Int V）to（Int，Int C），比如叠加。所以createCombiner reduceBykey很简单，就是直接返回 v，而 mergeValue和 mergeCombiners 逻辑是相同的，没有区别。  
    函数实现：  
    def reduceByKey(partitioner: Partitioner, func: (V, V) => V): RDD[(K, V)]= {  
 combineByKey[V]((v: V) => v, func, func, partitioner)  
 }  
　　图17中的方框代表RDD分区。通过用户自定义函数(A,B)=>(A+B)函数，将相同key的数据(V1,2)和(V1,1)的value相加运算，结果为（V1,3）。  
   

　图17 reduceByKey算子对RDD转换

### partitionBy

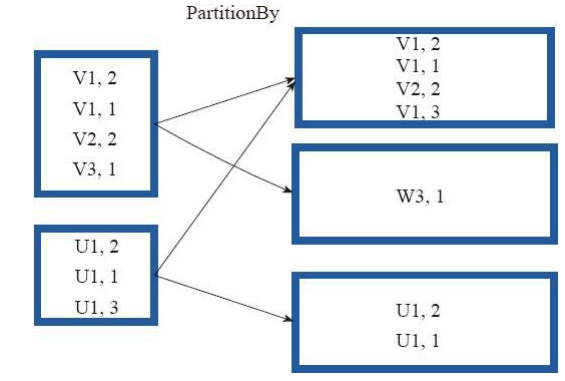
　　partitionBy函数对RDD进行分区操作。  
　　函数定义如下。  
　　partitionBy（partitioner：Partitioner）  
　　如果原有RDD的分区器和现有分区器（partitioner）一致，则不重分区，如果不一致，则相当于根据分区器生成一个新的ShuffledRDD。  
　　图18中的方框代表RDD分区。通过新的分区策略将原来在不同分区的V1、V2数据都合并到了一个分区。  


　图18 partitionBy算子对RDD转换

### **Cogroup**

 　　cogroup函数将两个RDD进行协同划分，cogroup函数的定义如下。  
　　cogroup[W]（other：RDD[（K，W）]，numPartitions：Int）：RDD[（K，（Iterable[V]，Iterable[W]））]  
　　对在两个RDD中的Key-Value类型的元素，每个RDD相同Key的元素分别聚合为一个集合，并且返回两个RDD中对应Key的元素集合的迭代器。  
　　（K，（Iterable[V]，Iterable[W]））  
　其中，Key和Value，Value是两个RDD下相同Key的两个数据集合的迭代器所构成的元组。  
　　图19中的大方框代表RDD，大方框内的小方框代表RDD中的分区。将RDD1中的数据（U1，1）、（U1，2）和RDD2中的数据（U1，2）合并为（U1，（（1，2），（2）））。

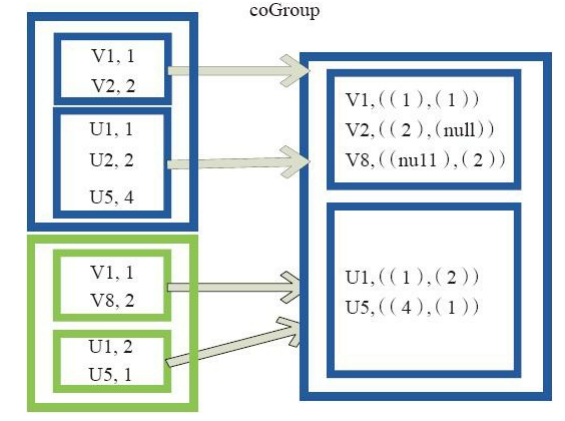


　图19 Cogroup算子对RDD转换

### **Join**

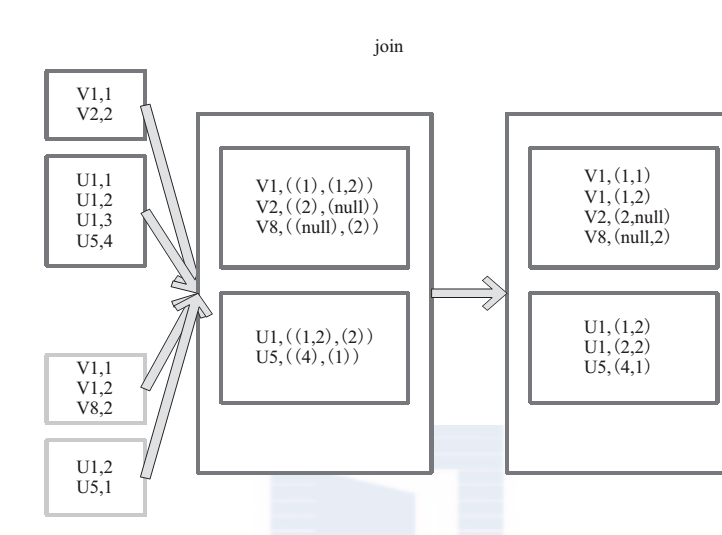
    join对两个需要连接的RDD进行cogroup函数操作，将相同key的数据能够放到一个分区，在cogroup操作之后形成的新RDD对每个key下的元素进行笛卡尔积的操作，返回的结果再展平，对应key下的所有元组形成一个集合。最后返回RDD[(K，(V，W))]。  
　　下面代码为join的函数实现，本质是通过cogroup算子先进行协同划分，再通过flatMapValues将合并的数据打散。  
    this.cogroup(other,partitioner).flatMapValues{case(vs,ws)=> for(v<-vs;w<-ws)yield(v,w)}  
图20是对两个RDD的join操作示意图。大方框代表RDD，小方框代表RDD中的分区。函数对相同key的元素，如V1为key做连接后结果为(V1,(1,1))和(V1,(1,2))。  


　图20 join算子对RDD转换

### **eftOutJoin和rightOutJoin**

　　LeftOutJoin（左外连接）和RightOutJoin（右外连接）相当于在join的基础上先判断一侧的RDD元素是否为空，如果为空，则填充为空。如果不为空，则将数据进行连接运算，并返回结果。  
下面代码是leftOutJoin的实现。  
if（ws.isEmpty）{  
vs.map（v=>（v，None））  
}else{  
for（v<-vs；w<-ws）yield（v，Some（w））  
}

## 2、Actions算子

　　本质上在Action算子中通过SparkContext进行了提交作业的runJob操作，触发了RDDDAG的执行。  
例如，Action算子collect函数的代码如下，感兴趣的读者可以顺着这个入口进行源码剖析：  
  
/\*\*  
\* Return an array that contains all of the elements in this RDD.  
\*/  
def collect(): Array[T] = {  
/\* 提交 Job\*/  
val results = sc.runJob(this, (iter: Iterator[T]) => iter.toArray)  
Array.concat(results: \_\*)  
}

### **foreach**

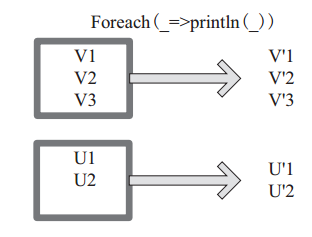
　　foreach对RDD中的每个元素都应用f函数操作，不返回RDD和Array，而是返回Uint。图22表示foreach算子通过用户自定义函数对每个数据项进行操作。本例中自定义函数为println()，控制台打印所有数据项。  
　　

图22foreach算子对RDD转换

### **saveAsTextFile**

　　函数将数据输出，存储到HDFS的指定目录。

下面为saveAsTextFile函数的内部实现，其内部通过调用saveAsHadoopFile进行实现：  
**this.map(x => (NullWritable.get(), new Text(x.toString))).saveAsHadoopFile[TextOutputFormat[NullWritable, Text]](path)**  
将RDD中的每个元素映射转变为(null，x.toString)，然后再将其写入HDFS。  
图23中左侧方框代表RDD分区，右侧方框代表HDFS的Block。通过函数将RDD的每个分区存储为HDFS中的一个Block。

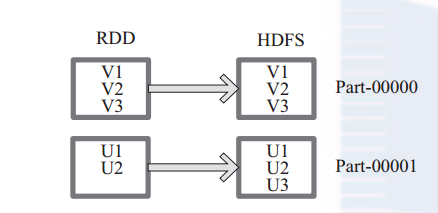


图23 saveAsHadoopFile算子对RDD转换

### **saveAsObjectFile**

　　saveAsObjectFile将分区中的每10个元素组成一个Array，然后将这个Array序列化，映射为（Null，BytesWritable（Y））的元素，写入HDFS为SequenceFile的格式。  
　　下面代码为函数内部实现。  
　　map（x=>（NullWritable.get（），new BytesWritable（Utils.serialize（x））））  
　　图24中的左侧方框代表RDD分区，右侧方框代表HDFS的Block。通过函数将RDD的每个分区存储为HDFS上的一个Block。

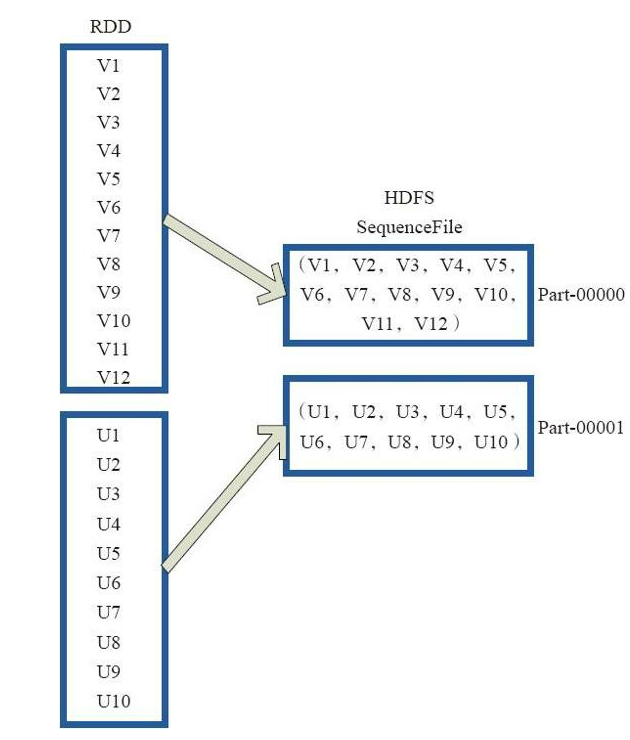


图24saveAsObjectFile算子对RDD转换

### **Collect**

　　collect相当于toArray，toArray已经过时不推荐使用，collect将分布式的RDD返回为一个单机的scalaArray数组。在这个数组上运用scala的函数式操作。  
　　图25中左侧方框代表RDD分区，右侧方框代表单机内存中的数组。通过函数操作，将结果返回到Driver程序所在的节点，以数组形式存储。

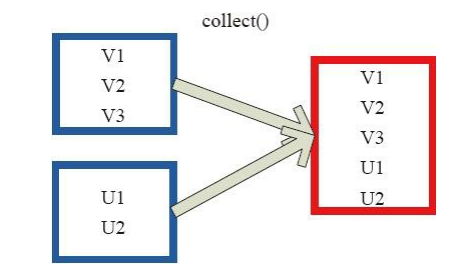


图25 Collect算子对RDD转换

### **collectAsMap**

　　collectAsMap对（K，V）型的RDD数据返回一个单机HashMap。对于重复K的RDD元素，后面的元素覆盖前面的元素。  
　　图26中的左侧方框代表RDD分区，右侧方框代表单机数组。数据通过collectAsMap函数返回给Driver程序计算结果，结果以HashMap形式存储。

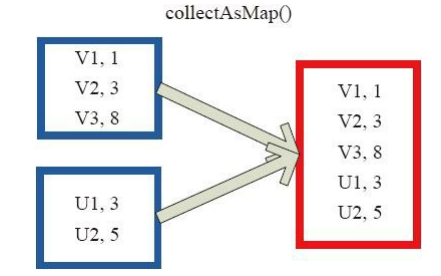


 图26CollectAsMap算子对RDD转换

### **reduceByKeyLocally**

　　实现的是先reduce再collectAsMap的功能，先对RDD的整体进行reduce操作，然后再收集所有结果返回为一个HashMap。

### **lookup**

下面代码为lookup的声明。  
lookup（key：K）：Seq[V]  
Lookup函数对（Key，Value）型的RDD操作，返回指定Key对应的元素形成的Seq。这个函数处理优化的部分在于，如果这个RDD包含分区器，则只会对应处理K所在的分区，然后返回由（K，V）形成的Seq。如果RDD不包含分区器，则需要对全RDD元素进行暴力扫描处理，搜索指定K对应的元素。  
　　图28中的左侧方框代表RDD分区，右侧方框代表Seq，最后结果返回到Driver所在节点的应用中。

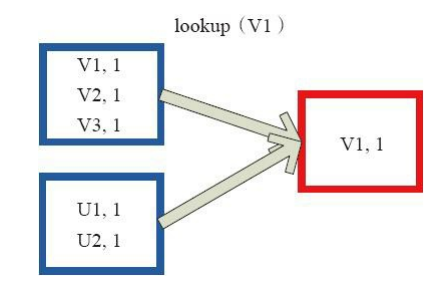


图28 lookup对RDD转换

### **count**

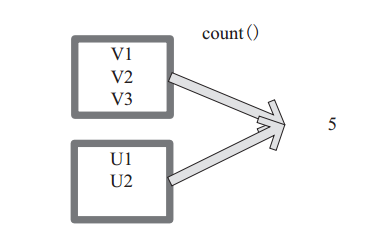
　　count返回整个RDD的元素个数。内部函数实现为：  
　　defcount():Long=sc.runJob(this,Utils.getIteratorSize\_).sum  
　　图29中，返回数据的个数为5。一个方块代表一个RDD分区。  


图29count对RDD算子转换

### **top**

top可返回最大的k个元素。函数定义如下。  
top（num：Int）（implicitord：Ordering[T]）：Array[T]

相近函数说明如下。  
·top返回最大的k个元素。  
·take返回最小的k个元素。  
·takeOrdered返回最小的k个元素，并且在返回的数组中保持元素的顺序。  
·first相当于top（1）返回整个RDD中的前k个元素，可以定义排序的方式Ordering[T]。  
返回的是一个含前k个元素的数组。

### **reduce**

　　reduce函数相当于对RDD中的元素进行reduceLeft函数的操作。函数实现如下。  
　　Some（iter.reduceLeft（cleanF））  
　　reduceLeft先对两个元素<K，V>进行reduce函数操作，然后将结果和迭代器取出的下一个元素<k，V>进行reduce函数操作，直到迭代器遍历完所有元素，得到最后结果。在RDD中，先对每个分区中的所有元素<K，V>的集合分别进行reduceLeft。每个分区形成的结果相当于一个元素<K，V>，再对这个结果集合进行reduceleft操作。  
　　例如：用户自定义函数如下。  
　　f：（A，B）=>（A.\_1+”@”+B.\_1，A.\_2+B.\_2）  
　　图31中的方框代表一个RDD分区，通过用户自定函数f将数据进行reduce运算。示例  
最后的返回结果为V1@[1]V2U！@U2@U3@U4，12。

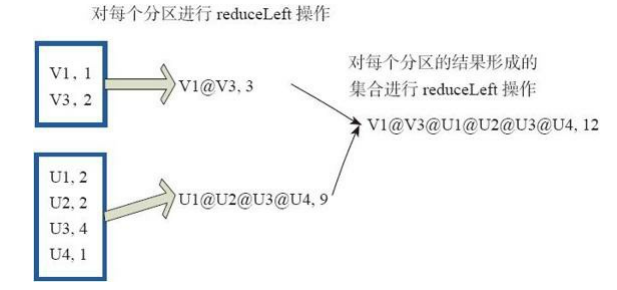


图31 reduce算子对RDD转换

### **fold**

　　fold和reduce的原理相同，但是与reduce不同，相当于每个reduce时，迭代器取的第一个元素是zeroValue。  
　　图32中通过下面的用户自定义函数进行fold运算，图中的一个方框代表一个RDD分区。读者可以参照reduce函数理解。  
　　fold（（”V0@”，2））（（A，B）=>（A.\_1+”@”+B.\_1，A.\_2+B.\_2））

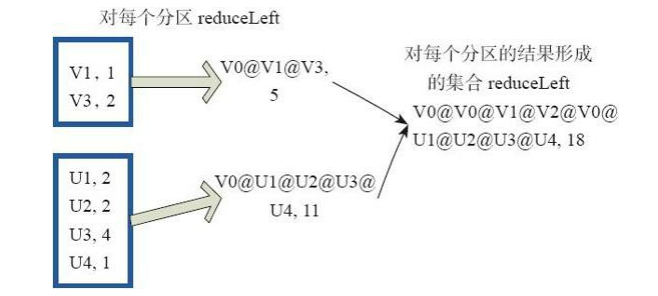


　图32 fold算子对RDD转换

### **aggregate**

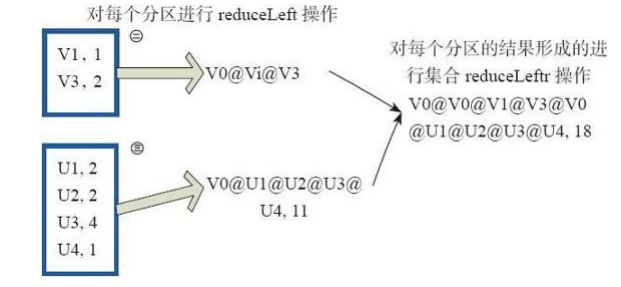
 　aggregate先对每个分区的所有元素进行aggregate操作，再对分区的结果进行fold操作。  
　aggreagate与fold和reduce的不同之处在于，aggregate相当于采用归并的方式进行数据聚集，这种聚集是并行化的。而在fold和reduce函数的运算过程中，每个分区中需要进行串行处理，每个分区串行计算完结果，结果再按之前的方式进行聚集，并返回最终聚集结果。  
　　函数的定义如下。  
aggregate[B]（z：B）（seqop：（B，A）=>B，combop：（B，B）=>B）：B  
　　图33通过用户自定义函数对RDD进行aggregate的聚集操作，图中的每个方框代表一个RDD分区。  
　　rdd.aggregate（”V0@”，2）（（A，B）=>（A.\_1+”@”+B.\_1，A.\_2+B.\_2）），（A，B）=>（A.\_1+”@”+B\_1，A.\_@+B\_.2））  
　　最后，介绍两个计算模型中的两个特殊变量。  
　　广播（broadcast）变量：其广泛用于广播MapSideJoin中的小表，以及广播大变量等场景。这些数据集合在单节点内存能够容纳，不需要像RDD那样在节点之间打散存储。  
Spark运行时把广播变量数据发到各个节点，并保存下来，后续计算可以复用。相比Hadoo的distributedcache，广播的内容可以跨作业共享。Broadcast的底层实现采用了BT机制。  


　图33 aggregate算子对RDD转换

②代表V。  
　　③代表U。  
　　accumulator变量：允许做全局累加操作，如accumulator变量广泛使用在应用中记录当前的运行指标的情景。

### **aggregateByKey**

|  |
| --- |
| **val pairRDD = sc.parallelize(List( ("cat",2), ("cat", 5), ("mouse", 4),("cat", 12), ("dog", 12), ("mouse", 2)), 2)**  **def func2(index: Int, iter: Iterator[(String, Int)]) : Iterator[String] = {**  **iter.toList.map(x => "[partID:" + index + ", val: " + x + "]").iterator**  **}**  **pairRDD.mapPartitionsWithIndex(func2).collect**  **pairRDD.aggregateByKey(0)(math.max(\_, \_), \_ + \_).collect**  **pairRDD.aggregateByKey(100)(math.max(\_, \_), \_ + \_).collect** |

## 示例程序

Spark里的计算都是操作RDD进行，那么学习RDD的第一个问题就是如何构建RDD，构建RDD从数据来源角度分为两类：

第一类是从内存里直接读取数据，

第二类就是从文件系统里读取，当然这里的文件系统种类很多常见的就是HDFS以及本地文件系统了。

第一类方式从内存里构造RDD，使用的方法：makeRDD和parallelize方法，如下代码所示：

|  |
| --- |
| **/\*使用makeRDD创建RDD\*/**  **/\* List \*/**  **val rdd01 = sc.makeRDD(List(1,2,3,4,5,6))**  **val r01 = rdd01.map { x => x \* x }**  **println(r01.collect().mkString(","))**  **/\* Array \*/**  **val rdd02 = sc.makeRDD(Array(1,2,3,4,5,6))**  **val r02 = rdd02.filter { x => x < 5}**  **println(r02.collect().mkString(","))**    **val rdd03 = sc.parallelize(List(1,2,3,4,5,6), 1)**  **val r03 = rdd03.map { x => x + 1 }**  **println(r03.collect().mkString(","))**  **/\* Array \*/**  **val rdd04 = sc.parallelize(List(1,2,3,4,5,6), 1)**  **val r04 = rdd04.filter { x => x > 3 }**  **println(r04.collect().mkString(","))** |

**大家看到了RDD本质就是一个数组，因此构造数据时候使用的是List（链表）和Array（数组）类型。**

**第二类方式是通过文件系统构造RDD，代码如下所示：**

|  |
| --- |
| val rdd:RDD[String] = sc.textFile("file:///D:/sparkdata.txt", 1)  val r:RDD[String] = rdd.flatMap { x => x.split(",") }  println(r.collect().mkString(",")) |

## Transformations转化操作：

|  |
| --- |
| val rddInt:RDD[Int] = sc.makeRDD(List(1,2,3,4,5,6,2,5,1))  val rddStr:RDD[String] = sc.parallelize(Array("a","b","c","d","b","a"), 1)  val rddFile:RDD[String] = sc.textFile(path, 1)    val rdd01:RDD[Int] = sc.makeRDD(List(1,3,5,3))  val rdd02:RDD[Int] = sc.makeRDD(List(2,4,5,1))    /\* map操作 \*/  println("======map操作======")  println(rddInt.map(x => x + 1).collect().mkString(","))  println("======map操作======")  /\* filter操作 \*/  println("======filter操作======")  println(rddInt.filter(x => x > 4).collect().mkString(","))  println("======filter操作======")  /\* flatMap操作 \*/  println("======flatMap操作======")  println(rddFile.flatMap { x => x.split(",") }.first())  println("======flatMap操作======")  /\* distinct去重操作 \*/  println("======distinct去重======")  println(rddInt.distinct().collect().mkString(","))  println(rddStr.distinct().collect().mkString(","))  println("======distinct去重======")  /\* union操作 \*/  println("======union操作======")  println(rdd01.union(rdd02).collect().mkString(","))  println("======union操作======")  /\* intersection操作 \*/  println("======intersection操作======")  println(rdd01.intersection(rdd02).collect().mkString(","))  println("======intersection操作======")  /\* subtract操作 \*/  println("======subtract操作======")  println(rdd01.subtract(rdd02).collect().mkString(","))  println("======subtract操作======")  /\* cartesian操作 \*/  println("======cartesian操作======")  println(rdd01.cartesian(rdd02).collect().mkString(","))  println("======cartesian操作======") |

## Action行动操作：

|  |
| --- |
| **val rddInt:RDD[Int] = sc.makeRDD(List(1,2,3,4,5,6,2,5,1))**  **val rddStr:RDD[String] = sc.parallelize(Array("a","b","c","d","b","a"), 1)**    **/\* count操作 \*/**  **println("======count操作======")**  **println(rddInt.count())**  **println("======count操作======")**  **/\* countByValue操作 \*/**  **println("======countByValue操作======")**  **println(rddInt.countByValue())**  **println("======countByValue操作======")**  **/\* reduce操作 \*/**  **println("======countByValue操作======")**  **println(rddInt.reduce((x ,y) => x + y))**  **println("======countByValue操作======")**  **/\* fold操作 \*/**  **println("======fold操作======")**  **println(rddInt.fold(0)((x ,y) => x + y))**  **println("======fold操作======")**  **/\* aggregate操作 \*/**  **println("======aggregate操作======")**  **val res:(Int,Int) = rddInt.aggregate((0,0))((x,y) => (x.\_1 + x.\_2,y),(x,y) => (x.\_1 + x.\_2,y.\_1 + y.\_2))**  **println(res.\_1 + "," + res.\_2)**  **println("======aggregate操作======")**  **/\* foeach操作 \*/**  **println("======foeach操作======")**  **println(rddStr.foreach { x => println(x) })**  **println("======foeach操作======")** |

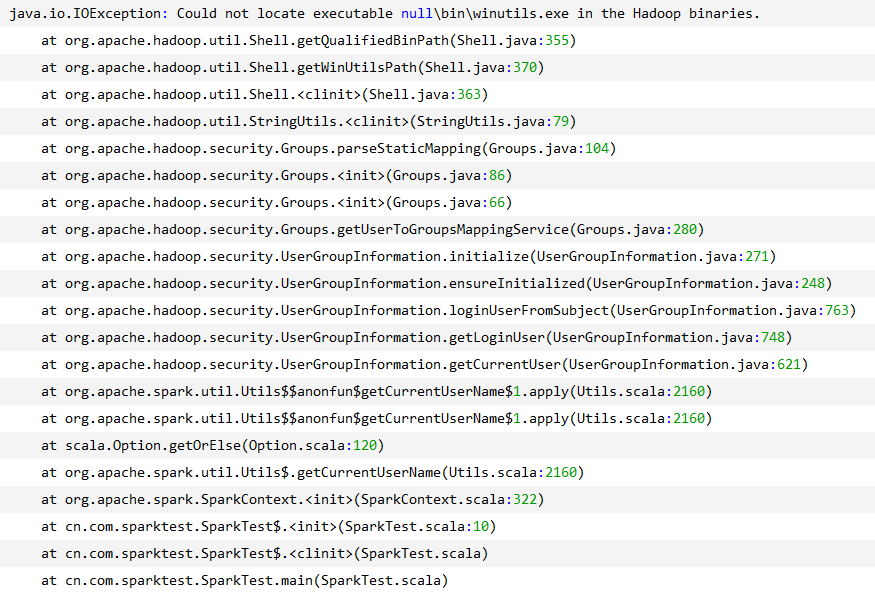
## 装有scala插件的eclipse里的完整代码：

|  |
| --- |
| package cn.com.sparktest    import org.apache.spark.SparkConf  import org.apache.spark.SparkConf  import org.apache.spark.SparkContext  import org.apache.spark.rdd.RDD    object SparkTest {    val conf:SparkConf = new SparkConf().setAppName("xtq").setMaster("local[2]")    val sc:SparkContext = new SparkContext(conf)      /\*\*     \* 创建数据的方式--从内存里构造数据(基础)     \*/    def createDataMethod():Unit = {      /\* 使用makeRDD创建RDD \*/      /\* List \*/      val rdd01 = sc.makeRDD(List(1,2,3,4,5,6))      val r01 = rdd01.map { x => x \* x }      println("===================createDataMethod:makeRDD:List=====================")      println(r01.collect().mkString(","))      println("===================createDataMethod:makeRDD:List=====================")      /\* Array \*/      val rdd02 = sc.makeRDD(Array(1,2,3,4,5,6))      val r02 = rdd02.filter { x => x < 5}      println("===================createDataMethod:makeRDD:Array=====================")      println(r02.collect().mkString(","))      println("===================createDataMethod:makeRDD:Array=====================")        /\* 使用parallelize创建RDD \*/      /\* List \*/      val rdd03 = sc.parallelize(List(1,2,3,4,5,6), 1)      val r03 = rdd03.map { x => x + 1 }      println("===================createDataMethod:parallelize:List=====================")      println(r03.collect().mkString(","))      println("===================createDataMethod:parallelize:List=====================")      /\* Array \*/      val rdd04 = sc.parallelize(List(1,2,3,4,5,6), 1)      val r04 = rdd04.filter { x => x > 3 }      println("===================createDataMethod:parallelize:Array=====================")      println(r04.collect().mkString(","))      println("===================createDataMethod:parallelize:Array=====================")    }      /\*\*     \* 创建Pair Map     \*/    def createPairRDD():Unit = {      val rdd:RDD[(String,Int)] = sc.makeRDD(List(("key01",1),("key02",2),("key03",3)))      val r:RDD[String] = rdd.keys      println("===========================createPairRDD=================================")      println(r.collect().mkString(","))      println("===========================createPairRDD=================================")    }      /\*\*     \* 通过文件创建RDD     \* 文件数据：     \*    key01,1,2.3            key02,5,3.7        key03,23,4.8        key04,12,3.9        key05,7,1.3     \*/    def createDataFromFile(path:String):Unit = {      val rdd:RDD[String] = sc.textFile(path, 1)      val r:RDD[String] = rdd.flatMap { x => x.split(",") }      println("=========================createDataFromFile==================================")      println(r.collect().mkString(","))      println("=========================createDataFromFile==================================")    }      /\*\*     \* 基本的RDD操作     \*/    def basicTransformRDD(path:String):Unit = {      val rddInt:RDD[Int] = sc.makeRDD(List(1,2,3,4,5,6,2,5,1))      val rddStr:RDD[String] = sc.parallelize(Array("a","b","c","d","b","a"), 1)      val rddFile:RDD[String] = sc.textFile(path, 1)        val rdd01:RDD[Int] = sc.makeRDD(List(1,3,5,3))      val rdd02:RDD[Int] = sc.makeRDD(List(2,4,5,1))        /\* map操作 \*/      println("======map操作======")      println(rddInt.map(x => x + 1).collect().mkString(","))      println("======map操作======")      /\* filter操作 \*/      println("======filter操作======")      println(rddInt.filter(x => x > 4).collect().mkString(","))      println("======filter操作======")      /\* flatMap操作 \*/      println("======flatMap操作======")      println(rddFile.flatMap { x => x.split(",") }.first())      println("======flatMap操作======")      /\* distinct去重操作 \*/      println("======distinct去重======")      println(rddInt.distinct().collect().mkString(","))      println(rddStr.distinct().collect().mkString(","))      println("======distinct去重======")      /\* union操作 \*/      println("======union操作======")      println(rdd01.union(rdd02).collect().mkString(","))      println("======union操作======")      /\* intersection操作 \*/      println("======intersection操作======")      println(rdd01.intersection(rdd02).collect().mkString(","))      println("======intersection操作======")      /\* subtract操作 \*/      println("======subtract操作======")      println(rdd01.subtract(rdd02).collect().mkString(","))      println("======subtract操作======")      /\* cartesian操作 \*/      println("======cartesian操作======")      println(rdd01.cartesian(rdd02).collect().mkString(","))      println("======cartesian操作======")    }      /\*\*     \* 基本的RDD行动操作     \*/    def basicActionRDD():Unit = {      val rddInt:RDD[Int] = sc.makeRDD(List(1,2,3,4,5,6,2,5,1))      val rddStr:RDD[String] = sc.parallelize(Array("a","b","c","d","b","a"), 1)        /\* count操作 \*/      println("======count操作======")      println(rddInt.count())      println("======count操作======")      /\* countByValue操作 \*/      println("======countByValue操作======")      println(rddInt.countByValue())      println("======countByValue操作======")      /\* reduce操作 \*/      println("======countByValue操作======")      println(rddInt.reduce((x ,y) => x + y))      println("======countByValue操作======")      /\* fold操作 \*/      println("======fold操作======")      println(rddInt.fold(0)((x ,y) => x + y))      println("======fold操作======")      /\* aggregate操作 \*/      println("======aggregate操作======")      val res:(Int,Int) = rddInt.aggregate((0,0))((x,y) => (x.\_1 + x.\_2,y),(x,y) => (x.\_1 + x.\_2,y.\_1 + y.\_2))      println(res.\_1 + "," + res.\_2)      println("======aggregate操作======")      /\* foeach操作 \*/      println("======foeach操作======")      println(rddStr.foreach { x => println(x) })      println("======foeach操作======")    }      def main(args: Array[String]): Unit = {      println(System.getenv("HADOOP\_HOME"))      createDataMethod()      createPairRDD()      createDataFromFile("file:///D:/sparkdata.txt")      basicTransformRDD("file:///D:/sparkdata.txt")      basicActionRDD()      /\*打印结果\*/      /\*D://hadoop  ===================createDataMethod:makeRDD:List=====================  1,4,9,16,25,36  ===================createDataMethod:makeRDD:List=====================  ===================createDataMethod:makeRDD:Array=====================  1,2,3,4  ===================createDataMethod:makeRDD:Array=====================  ===================createDataMethod:parallelize:List=====================  2,3,4,5,6,7  ===================createDataMethod:parallelize:List=====================  ===================createDataMethod:parallelize:Array=====================  4,5,6  ===================createDataMethod:parallelize:Array=====================  ===========================createPairRDD=================================  key01,key02,key03  ===========================createPairRDD=================================  key01,1,2.3,key02,5,3.7,key03,23,4.8,key04,12,3.9,key05,7,1.3  =========================createDataFromFile==================================  2,3,4,5,6,7,3,6,2  ======map操作======  ======filter操作======  5,6,5  ======filter操作======  ======flatMap操作======  key01  ======flatMap操作======  ======distinct去重======  4,6,2,1,3,5  ======distinct去重======  ======union操作======  1,3,5,3,2,4,5,1  ======union操作======  ======intersection操作======  1,5  ======intersection操作======  ======subtract操作======  3,3  ======subtract操作======  ======cartesian操作======  (1,2),(1,4),(3,2),(3,4),(1,5),(1,1),(3,5),(3,1),(5,2),(5,4),(3,2),(3,4),(5,5),(5,1),(3,5),(3,1)  ======cartesian操作======  ======count操作======  9  ======count操作======  ======countByValue操作======  Map(5 -> 2, 1 -> 2, 6 -> 1, 2 -> 2, 3 -> 1, 4 -> 1)  ======countByValue操作======  ======countByValue操作======  29  ======countByValue操作======  ======fold操作======  29  ======fold操作======  ======aggregate操作======  19,10  ======aggregate操作======  ======foeach操作======  a  b  c  d  b  a  ======foeach操作======\*/    }  } |

Spark执行时候我们需要构造一个SparkContenxt的环境变量，构造环境变量时候需要构造一个SparkConf对象，例如代码：setAppName("xtq").setMaster("local[2]")

appName就是spark任务名称，master为local[2]是指使用本地模式，启动2个线程完成spark任务。

在eclipse里运行spark程序时候，会报出如下错误：



该错误不会影响程序的运算，但总是让人觉得不舒服，这个问题是因为spark运行依赖于hadoop，可是在window下其实是无法安装hadoop，只能使用cygwin模拟安装，而新版本的hadoop在windows下使用需要使用winutils.exe，解决这个问题很简单，就是下载一个winutils.exe，注意下自己操作系统是32位还是64位，找到对应版本，然后放置在这样的目录下：

　　D:\hadoop\bin\winutils.exe

　　然后再环境变量里定义HADOOP\_HOME= D:\hadoop

　　环境变量的改变要重启eclipse，这样环境变量才会生效，这个时候程序运行就不会报出错误了。