# 转换算子(transform)

## Value类型

### Map(func)

|  |
| --- |
| 底层实现,用的MapPartitionsRDD  作用:返回一个新的RDD, 该RDD由每一个输入元素经过func函数转化完成.  val rdd = sc.makeRDD(Array(1,2,3))  rdd.map(x=>(x,1))  ((1,1), (2,1), (3,1)) |

### flatMap(func)

|  |
| --- |
| 底层实现,用的MapPartitionsRDD  作用: 类似于map, 但是每一个输入元素可以被映射为0或者多个输入元素(所以func应该返回一个序列, 而不是单一元素)  val rdd = sc.makeRDD(List(“coffee panda”, “happy panda”, “anger panda”))  rdd.flatMap(x=>x.split(“ ”))  Array(coffee, panda, hadppy, panda, anger, panda) |

### Map和flatMap的区别

|  |
| --- |
| 1. 从字面上来看, flatMap比Map多一个flat功能, flat意为压平 2. 实质上来说, flatMap就是先Map, 再flat.   举例说明  val rdd = sc.makeRDD(List(“coffee panda”, “happy panda”, “anger panda”))  rdd.map(x=>x.split(“ ”)) // RDD[Array[String]]  rdd.flatMap(x=>x.split(“ ”)) // RDD[String]  从上面看 flatMap把Array压平成一个一个字符串 |

### filter(func)

|  |
| --- |
| 底层实现,用的MapPartitionsRDD  作用: 过滤, 返回一个新的RDD, 该RDD由经过func函数计算后返回值为true的输入元素组成.  val rdd = sc.makeRDD(List(1,2,3,4))  rdd.filter(x=>x>2) // Array(3,4) |

### mapPartitions(func)

|  |
| --- |
| 底层实现,用的MapPartitionsRDD  作用: 类似于map, 但独立地在RDD的每一个分区上运行, 因此类型为T的RDD上运行时, func的函数类型必须是Iterator[T]=>Iterator[U]. 假设有N个元素, M个分区, 那么map的函数将被调用N次, 而mapPartitions被调用M次, mapPartitions函数一次处理所有分区.  val rdd = sc.makeRDD(Array(1,2,3))  rdd.mapPartitions(x=>x.map(x=>(x,1)))  Array((1,1), (2,1), (3,1)) |

### mapPartitionsWithIndex(func)

|  |
| --- |
| 底层实现,用的MapPartitionsRDD  作用: 类似于mapPartitions, 但func带有一个整数参数表示分片的索引值, 因此在类型为T的RDD上运行时, func的函数类型必须是(Int, Interator[T]) => Iterator[U];  val rdd = sc.makeRDD(List(1,2,3,4))  rdd.mapPartitionsWithIndex((index, item) => (item.map((index,\_))))  Array[(Int, Int)] = Array((0,1), (0,2), (1,3), (1,4), (1,5), (2,6), (2,7), (3,8), (3,9), (3,10)) |

### Map和mapPartitions以及mapPartitionsWithIndex的区别

|  |
| --- |
| 相同点   1. 这3个函数, 底层都是用MapPartitionsRDD实现的. 2. 无论mapPartitions和mapPartitionsWithIndex, 最后都是需要map去转换   不同点   1. mapPartitions通过把值按照分区传输, 在需要返回值的情况下, Executor与Driver连接的开销会减少(Driver一次性把整个分区发过来, 然后一次性处理完毕发送;)但是在某些情况下, 例如分区数据过大, 容易造成OOM. 2. Map是把值, 一个一个从Driver发送到Executor, 这样的好处是, Executor不用占据太大的内存. 3. mapPartitionsWithIndex, 实在mapPartitions的基础上, 把分区的索引值来回传递. 好处与mapPartitionsWithIndex一样.   Note:  这3个函数, 都是可以指定分区的, 默认以local[\*]为主. |

### MapPartitionsRDD

底层实现,用的MapPartitionsRDD

### glom

|  |
| --- |
| 作用: 将每一个分区形成一个数组, 形成新的RDD类型RDD[Array[T]]  举例  val rdd = sc.makeRDD(List(1,2,3,4,5,6,7,8,9,10))  rdd.glom.collect//  Array[Array[Int]] = Array(Array(1, 2), Array(3, 4, 5), Array(6, 7), Array(8, 9, 10)) |

### groupBy(func)

|  |
| --- |
| 作用: 分组, 按照传入函数的返回值进行分组. 即将相同key对应的值放入在一个迭代器.  举例  val rdd = sc.makeRDD(1 to 4)  rdd.groupBy(\_%2)  Array[(Int, Iterable[Int])] = Array((0,CompactBuffer(2, 4)), (1,CompactBuffer(1, 3))) |

### sample(withReplacement, fraction, seed)

有值得深挖的部分, 后续回补.

|  |
| --- |
| 作用: 以指定的随机种子随机抽样出数量为fraction的数据, withReplacement表示抽出的数据是否放回.(true表示放回), seed表示随机数生成器的种子  举例  val rdd = sc.makeRDD(1 until 5)  rdd.sample(false, 0.1, 1)  Array[Int] = Array(4) |

### distinct([numTasks])

|  |
| --- |
| 作用: 对源RDD进行去重后返回一个新的RDD. 可以通过传入参数numTasks改变并行度.  举例:  val rdd = sc.makeRDD(List(1,2,3,5,3,4,2))  rdd.disinct // Array[Int] = Array(4, 1, 5, 2, 3)  rdd.distinct(2).glom.collect // Array[Array[Int]] = Array(Array(4, 2), Array(1, 3, 5)) |

### coalesce(numPartitions)

|  |
| --- |
| 作用: 缩减分区数, 用于大数据集过滤后, 提高小数据集的执行效率.  举例:  val rdd = sc.parallelize(1 to 16)  rdd.partitions.size //4  rdd.coalesce(3).partitions.size //3  Note:  coalesce(nums), 其缩减分区, 就是简单的把分区合并在一起. 另外不能扩张分区数 |

### repartition(numPartitions)

|  |
| --- |
| 作用: 根据分区数, 重新将数据shuffle到新的分区中.  举例:  val rdd = sc.parallelize(1 to 16)  rdd.repartition(5)  Array[Array[Int]] = Array(Array(12, 7, 16), Array(8, 1), Array(2, 9, 13), Array(5, 14, 3, 10), Array(6, 15, 4, 11))  Note:  这个操作涉及Shuffle, 速度较慢 |

### sortBy(func,[acending],[numPartitions])

|  |
| --- |
| 作用: 使用func先对数据进行处理, 按照处理后的数据比较结果排序, 默认正序.  举例:  val rdd = sc.parallelize(1 to 4)  rdd.sortBy(x=>x) // Array[Array[Int]] = Array(Array(1), Array(2), Array(3), Array(4))  rdd.sortBy(x=>x%2) // Array(Array(2, 4), Array(1, 3), Array())  Note:   1. 排序是按照func后的结果进行排序 2. 过程可能涉及Shuffle?后补 |

### pipe(command, [envVars])

|  |
| --- |
| 作用: 针对每个分区, 都执行一个shell脚本, 返回输出的RDD  Note:  脚本需要放在Worker节点可以访问到的位置 |

## 双Value类型

### union(otherDataset)

|  |
| --- |
| 作用: 对源RDD和参数RDD求并集后返回一个新的RDD.  举例:  val rdd1 = sc.parallelize(1 to 3)  val rdd2 = sc.parallelize(3 to 6)  rdd1.union(rdd2) // Array[Int] = Array(1, 2, 3, 3, 4, 5, 6)  Note:   1. 不删除重复数据 |

### subtract(otherDataset)

|  |
| --- |
| 作用: 计算差的函数, 去除RDD中与otherDataset相同的元素, 不同的保留下来.  举例:  val rdd1 = sc.parallelize(1 to 3)  val rdd2 = sc.parallelize(3 to 6)  rdd1.subtract(rdd2) // Array[Int] = Array(1, 2) |

### intersection(otherDataset)

|  |
| --- |
| 作用: 对两个RDD求交集, 返回新的RDD  举例:  val rdd1 = sc.parallelize(1 to 3)  val rdd2 = sc.parallelize(3 to 6)  rdd1.intersection(rdd2) // Array[Int] = Array(3) |

### cartesian(otherDataset)

|  |
| --- |
| 作用: 对两个RDD求笛卡尔积  举例:  val rdd1 = sc.parallelize(1 to 3)  val rdd2 = sc.parallelize(3 to 6)  rdd1.cartesian(rdd2) // Array[(Int, Int)] = Array((1,3), (1,4), (1,5), (1,6), (2,3), (2,4), (2,5), (2,6), (3,3), (3,4), (3,5), (3,6)) |

### zip(otherDataset)

|  |
| --- |
| 作用: 将两个RDD组合成key/value形式的RDD, 这里默认两个RDD的partitions数量以及元素的数量都相同, 否则会报错  举例:  val rdd1 = sc.parallelize(List('a','b','c','d'))  val rdd2 = sc.parallelize(List(3,2.5,2,1.5))  rdd1.zip(rdd2) // Array[(Char, Double)] = Array((a,3.0), (b,2.5), (c,2.0), (d,1.5))  Note:   1. 一般人称拉链形式. |

## Key-Value类型

### partitionBy(分区器)

|  |
| --- |
| 作用: 对双值RDD进行分区操作, 如果原有的分区和现在一致, 就不进行分区, 否则通过Shuffle生成对应的分区.  举例:  val rdd = sc.parallelize(Array((1,"aaa"),(2,"bbb"),(3,"ccc"),(4,"ddd")),4)  rdd.partitionBy(new org.apache.spark.HashPartitioner(2))  Note:   1. 分区器可以重写. 后补重写相关过程 |

### groupByKey()

|  |
| --- |
| 作用: 对双值RDD的Key进行聚合, 返回sequence  举例:  val rdd = sc.makeRDD(List("hello world","hello Spark", "hello Scala"))  val pairRDD = rdd.flatMap(x=>x.split(" ")).map(x=>(x,1))  pariRDD.groupByKey() // Array[(String, Iterable[Int])] = Array((Spark,CompactBuffer(1)), (hello,CompactBuffer(1, 1, 1)), (Scala,CompactBuffer(1)), (world,CompactBuffer(1)))  //从这里可以看出, 这不是我们想要的, 还需要经过map的转化  pariRDD.groupByKey().map(x=>(x.\_1,x.\_2.sum)).collect  // Array[(String, Int)] = Array((Spark,1), (hello,3), (Scala,1), (world,1)) |

### reduceByKey(func,[numTasks])

|  |
| --- |
| 作用: 在一个(K,V)的RDD上调用, 返回一个(K,V)的RDD, 使用指定的reduce函数, 将相同key的值聚合在一起, reduce任务的个数可以通过第二个可选的参数来设置  举例:  val rdd = sc.makeRDD(List("hello world","hello Spark", "hello Scala"))  val pairRDD = rdd.flatMap(x=>x.split(" ")).map(x=>(x,1))  pairRDD.reduceByKey((x,y)=>x+y).collect  /002FArray[(String, Int)] = Array((Spark,1), (hello,3), (Scala,1), (world,1)) |

### reduceByKey和groupByKey的区别

|  |
| --- |
| 1. reduceByKey: 按照key进行聚合, 在Shuffle之前有combine(预聚合)操作, 返回结果是RDD(K,V). 2. groupByKey: 按照key进行分组, 直接进行shuffle. 3. reduceByKey更推荐使用, 但是注意逻辑, 比如在取count就有问题, 依业务来.(后补) 4. groupByKey在shuffle的时候做聚合, reduceByKey在shuffle之前就聚合了, 所以reduceByKey更推荐使用. |

### aggregateByKey

|  |
| --- |
| 1. 作用: 在KV对的RDD中, 按key将value进行分组合并. 合并时, 将每个value和初始值作为Seq函数的参数, 按行计算, 返回一个新的KV对. 然后再将结果按照key进行合并, 最后将每个分组的combine函数进行计算(先将前两个value进行计算, 将返回结果和下一个value传给combine函数, 以此类推), 将key与计算结果作为一个新的KV对输出. 2. 参数: (zeroValue:U,[partitioner:Partitioner])(SeqOp: (U,V)=>U, combOp: (U,U)=>U))   zeroValue: 给每一个分区中的每一个key一个初始值  seqOp: 函数用于在每一个分区中用初始值逐步迭代value  combOp: 函数用于合并每个分区中的结果   1. 举例:   val pairRDD = spark.sparkContext.parallelize(List(("a",3),("a",2),("c",4),("b",3),("c",6),("c",8)), 2)  // Array[Array[(String, Int)]] = Array(Array((a,3), (a,2), (c,4)), Array((b,3), (c,6), (c,8)))  pairRDD.aggregateByKey(0)(math.max(\_,\_),\_+\_).collect  // Array[(String, Int)] = Array((b,3), (a,3), (c,12))   1. Note: zreoValue, 是初始值. |

### foldByKey

|  |
| --- |
| 1. 作用: aggregateByKey的简化操作, 当分区内操作与分区间操作相同时, 代码可以少写(暂时只看到这个用途) 2. 参数: (zeroValue:V)(func: (V,V)=>V): RDD[(K,V)]] 3. 举例:   **val** pairRDD = spark.sparkContext.parallelize(*List*((**"a"**,3),(**"a"**,2),(**"c"**,4),(**"b"**,3),(**"c"**,6),(**"c"**,8)), 2)  **val** res = pairRDD.foldByKey(0)(\_+\_)  res.foreach(*println*)  // (a,5) (b,3) (c,18) |

### combineByKey

|  |
| --- |
| 1. 作用: 对相同的K, 把V合并成为一个集合. 2. 参数: (createCombine: V=>C, mergeValue: (C,V)=>C, mergeCombiners: (C,C)=>C) 3. 举例:   **val** pairRDD = spark.sparkContext.parallelize(*List*((**"a"**,3),(**"a"**,2),(**"c"**,4),(**"b"**,3),(**"c"**,6),(**"c"**,8)), 2)  **val** res = pairRDD.combineByKey((\_,1),(acc:(Int,Int), v)=>(acc.\_1+v, acc.\_2+1),(acc1:(Int,Int),acc2:(Int,Int))=>(acc1.\_1+acc2.\_1, acc1.\_2+acc2.\_2))  **val** finalres = res.map {  **case** (key, value) => (key, value.\_1 / value.\_2.toDouble) }  finalres.foreach(*println*)  //(a,2.5) (b,3.0) (c,6.0) |

### groupByKey、reduceByKey、aggregateByKey、foldByKey、combineByKey的区别

|  |
| --- |
| 相同点   1. 都是作用在pairRDD上的函数. 2. 底层都是调用combineByKeyWithClassTag这个方法     不同点   1. groupByKey 2. reduceByKey 3. aggregateByKey, 分区内操作和分区间操作可以不同. 4. foldByKey, 分区内和分区间操作相同. 5. combineByKey, 自由度最大, zeroValue的初始化, 分区内以及分区间的函数可以自己写. |

### sortByKey([ascending],[numPartitons])

|  |
| --- |
| 1. 作用: 在一个pairRDD上调用, 按照key进行排序 2. Note: sortByKey, 只在区内排序 |

### mapValues

|  |
| --- |
| 1. 作用: 针对(K,V)类型, 并对V进行操作 2. 举例:   val rdd = sc.makeRDD(List((1,"a"),(2,"b"),(3,"c")))  rdd.mapValues(\_+"111").collect  // Array[(Int, String)] = Array((1,a111), (2,b111), (3,c111)) |

### join

|  |
| --- |
| 后补 |

### cogroup(otherDataset,[numPartitions])

|  |
| --- |
| 后补 |

# 行动算子

## reduce(func)

|  |
| --- |
| 1. 作用: 通过func函数聚集RDD中的所有元素, 先聚合分区内的数据, 再聚合分区间数据. 2. 举例:   Demo1, 单值reduce  **val** rdd = spark.sparkContext.parallelize(1 to 10, 2)  rdd.reduce(\_+\_) //5  Demo2, 多值reduce  **val** rdd1 = spark.sparkContext.makeRDD(*Array*((**"a"**,1),(**"a"**,3),(**"c"**,3),(**"d"**,5)))  rdd1.reduce((x, y) => (x.\_1 + y.\_1, x.\_2 + y.\_2)) // (adca, 12), 可以看出来分区的影响, 并行作业. |

## collect()

|  |
| --- |
| 1. 作用: 在Driver中, 以数组的形式返回数据集中的所有元素 |

## count()

|  |
| --- |
| 1. 作用: 返回RDD中元素的个数 |

## first()

|  |
| --- |
| 1. 作用: 返回RDD中的第一个元素 |

## take(n)

|  |
| --- |
| 1. 作用: 返回一个由RDD的前n个元素组成的数组. |

## takeOrdered(n)

|  |
| --- |
| 1. 作用: 返回该RDD排序后的前n个元素组成的数组. |

## aggregate()

|  |
| --- |
|  |

## fold(num)(func)

|  |
| --- |
|  |

## saveAsTextFile(path)

|  |
| --- |
| 1. 作用: 将RDD中的元素以textfile的形式保存到HDFS文件系统或者其他支持的文件系统. 对于每个元素, Spark将会调用toString方法, 将它转换为文件中的文本. |

## saveAsSequenceFile(path)

|  |
| --- |
| 1. 作用: 将RDD中的元素以Hadoop sequencefile的形式保存到指定的目录下, 可以是HDFS或者其他hadoop支持的文件系统. |

## saveAsObjectFile(path)

|  |
| --- |
| 1. 作用: 将RDD中的元素序列化成对象, 存储到文件中. |

## countByKey()

|  |
| --- |
| 1. 作用: 针对(K,V)类型的RDD, 返回一个(K,Int)的map, 表示每一个key对应的元素个数. 2. 举例:   **val** rdd = spark.sparkContext.makeRDD(*List*(**"hello world"**,**"hello Spark"**, **"hello Scala"**))  **val** mapRDD = rdd.flatMap(x=>x.split(**" "**)).map(x=>(x,1))  mapRDD.countByKey.foreach(*println*)  // (hello,3) (world,1) (Spark,1) (Scala,1)   1. Note: 针对的是(K, V)类型的RDD, 如果非此类型, 则没有这个方法. |

## foreach(func)

|  |
| --- |
| 1. 作用: 在RDD的每一个元素上, 运行函数func进行更新. |