> 要么庸俗，要么孤独

RDD转换算子

单RDD处理

map算子

作用：返回一个新的RDD，该RDD由每一个输入元素经过func函数转换后组成。

需求：创建一个数值型的RDD，将所有元素加上1返回形成新的RDD。

(1) 创建RDD

```
scala> val rdd=sc.parallelize(0 to 9)
```

(2) 将所有元素+1

```
scala> val result=rdd.map(_+1)
```

(3) 查看结果

```
scala> result.collect
result:Array[Int]=Array(1，2，3，4，5，6，7，8，9，10)
```

mapPartitions算子

作用：mapPartitions和map作用类似，只不过作用RDD的范围不同，map是作用在RDD中的每个元素上，而mapPartitions作用的是RDD的分区，一次性处理整个分区，如果RDD有三个分区，则调用三次这个算子。

需求：创建一个具有三个分区的数值型RDD，然后对其中的元素都乘3。

(1) 创建有三个分区的RDD，可以使用rdd.partitions查看分区数

```
scala> val rdd=sc.parallelize(0 to 9，3)
```

(2) 将所有的元素乘3

```
scala> val result=rdd.mapPartitions(data=>data.map(_*3))
```

(3) 查看结果

```
scala> result.collect
result:Array[Int]=Array(0，3，6，9，12，15，18，21，24，27)
```

map和mapPartition区别

> 1.map()：每次处理一条数据。
>
> 2.mapPartition()：每次处理一个分区的数据，这个分区的数据处理完后，原RDD中分区的数据才能释放，可能导致OOM。
>
> 建议：当内存空间较大的时候建议使用mapPartition()，以提高处理效率。

mapPartitionsWithIndex算子

作用：这个算子的作用和上一个算子作用类似，只是在计算分区的时候都会带上分区的索引。

需求：创建一个数值型RDD，然后输出每个数值对应的分区。

(1) 创建RDD

```
scala> val rdd=sc.parallelize(0 to 6)
```

(2) 将分区索引和数据进行关联

```
scala> val result=rdd.mapPartitionsWithIndex((index，items)=>items.map((index，
_)))
```

(3) 查看结果

```
scala> result.collect
result:Array[(Int，Int)]=Array((0，0)，(0，1)，(0，2)，(1，3)，(1，4)，(1，5)，(1，
6))
```

flatMap算子

作用：这个算子的作用类似于map，但是它是对输入的每一个元素映射为0，1，或者多个元素的输出。

需求：创建一个数值型RDD，输出每个数值它的平方和立方。

(1) 创建RDD

```
scala> val rdd=sc.parallelize(0to3)
```

(2) 将每个元素映射为它的平发和立方

```
scala> val result=rdd.flatMap(x=>Array(x*x，x*x*x))
```

(3) 查看结果

```
scala> result.collect
result:Array[Int]=Array(0，0，1，1，4，8，9，27)
```

groupBy算子

作用：将返回的数组进行分组

需求：创建一个数值型RDD，按照奇偶性进行分组

(1) 创建RDD

```
scala> val rdd=sc.parallelize(0to3)
```

(2) 将数值按照奇偶性分组

```
scala> val result=rdd.groupBy(x=>if(x%2==0)"偶)"else"奇")
```

(3) 查看结果

```
scala> result.collect
result:Array[(String，Iterable[Int])]=Array((偶)，CompactBuffer(0，2，4，6))，(奇，
CompactBuffer(1，3，5)))
```

filter算子

作用：将RDD中的元素进行过滤返回条件为True的元素

需求：创建一个字符型RDD，返回带spark的字符串【有问题都可以私聊我WX：focusbigdata，或者关注我的公众号：FocusBigData，注意大小写】

(1) 创建RDD

```
scala> val rdd=sc.parallelize(Array("spark-basic"，"spark-
mlib"，"hadoop"，"yarn"))
```

(2) 过滤出带spark的字符串

```
scala> val result=rdd.filter(x=>x.contains("spark"))
```

(3) 查看结果

```
scala> result.collect
result:Array[String]=Array(spark-basic，spark-mlib)
```

distinct算子

作用：对RDD中的元素进行去重操作

需求：创建一个数值型RDD对其进行去重

(1) 创建RDD

```
scala> val rdd=sc.parallelize(Array(0，0，1，1，2，2))
```

(2) 过滤出带spark的字符串

```
scala> val result=rdd.distinct()
```

(3) 查看结果

```
scala> result.collect

result:Array[Int]=Array(0，2，1)
```

sample算子

作用：算子的参数列表为withPeplacement，fraction，seed。

l withReplacement：表示是抽出的数据是否放回，true为有放回的抽样，false为无放回的抽样。

l fraction：表示以指定的随机种子随机抽样出比例为fraction的数据。抽取的数量是：size*fraction，但是结果并不能保证结果的比例准确。

l seed：用于指定随机数生成器种子，默认传入当前时间戳。

需求：对数值型RDD进行不放回抽样和放回抽样。

(1) 创建RDD

```
scala> val rdd=sc.parallelize(0 to 9)
```

(2) 不放回抽样

```
scala> rdd.sample(false，0.5).collect
res2:Array[Int]=Array(0，1，2，4，5，9)
```

(3) 放回抽样

```
scala> rdd.sample(true，2).collect
res3:Array[Int]=Array(0，2，2，4，4，5，5，5，5，8，8，9)
```

repartition算子

作用：根据指定的分区数重新shuffle数据。

需求：重新指定数值型RDD分区为6个。

(1) 创建RDD

```
scala> val rdd=sc.parallelize(0 to 11，3)
```

(2) 重新分区

```
scala> val result=rdd.repartition(6)
```

(3) 查看分区数

```
scala> result.partitions.length
result:Int=6
```

coalesce算子

作用：缩小RDD分区数，用于大数据过滤，提高小数据集的执行效率。

需求：减少数值型RDD分区为3个.

(1) 创建RDD

```
scala> val rdd=sc.parallelize(0to11，6)
```

(2) 重新分区

```
scala> val result=rdd.coalesce(3)
```

(3) 查看分区数

```
scala> result.partitions.length
result:Int=3
```

coalesce和repartition的区别

> 1. coalesce重新分区，可以选择是否进行shuffle过程。由参数shuffle:Boolean=false/true决定。
> 2. repartition实际上是调用的coalesce，进行shuffle。源码如下：
>
> def repartition(numPartitions:Int)(implicitord:Ordering[T]=null):RDD[T]=withScope{
> coalesce(numPartitions，shuffle=true)
> }

sortBy算子

作用：对RDD先进行处理然后排序，默认为正序。

需求：对数值型RDD进行排序。

(1) 创建RDD

```
scala> val rdd=sc.parallelize(Array(4，2，1，5，0，3))
```

(2) 正序排序

```
scala> val result=rdd.sortBy(x=>x).collect
result:Array[Int]=Array(0，1，2，3，4，5)
```

(3) 反序排序

```
scala> val result=rdd.sortBy(x=>x，false).collect
result:Array[Int]=Array(5，4，3，2，1，0)
```

多RDD处理

> 上面说的算子都是对单个RDD进行处理，但是如果遇到需要两个RDD的交集这种操作，那么就需要以下这些操作集合的算子了。

union算子

作用：将两个RDD求并集后返回一个新的RDD。

需求：创建两个RDD求并集。

(1) 创建两个RDD

```
scala> val rdd1=sc.parallelize(0to2)
scala> val rdd2=sc.parallelize(3to4)
```

(2) 求并集，查看结果

```
scala> rdd1.union(rdd2).collect
res1:Array[Int]=Array(0，1，2，3，4)

注意：union和++是等价操作。
```

subtarct算子

作用：将两个RDD求差集后返回一个新的RDD。

需求：创建两个RDD求差集。

(1) 创建两个RDD

```
scala> val rdd1=sc.parallelize(0to5)
scala> val rdd2=sc.parallelize(3to6)
```

(2) 求rdd1-rdd2差集，查看结果

```
scala> rdd1.subtract(rdd2).collect
res1:Array[Int]=Array(0，2，1)
```

(3) 求rdd2-rdd1差集，查看结果

```
scala> rdd2.subtract(rdd1).collect
res2:Array[Int]=Array(6)
```

intersection算子

作用：将两个RDD求交集后返回一个新的RDD。

需求：创建两个RDD求交集。

(1) 创建两个RDD

```
scala> val rdd1=sc.parallelize(0to5)
scala> val rdd2=sc.parallelize(3to6)
```

(2) 求并集，查看结果

```
scala> rdd1.intersection(rdd2).collect
res19:Array[Int]=Array(4，3，5)
```

cartesian算子

作用：计算两个RDD的笛卡尔积，避免使用。

需求：创建两个RDD求笛卡尔积。

(1) 创建两个RDD

```
scala> val rdd1=sc.parallelize(0to2)
scala> val rdd2=sc.parallelize(3to5)
```

(2) 求笛卡尔积，查看结果

```
scala> rdd1.cartesian(rdd2).collect
res1:Array[(Int，Int)]=Array((0，3)，(0，4)，(0，5)，(1，3)，(2，3)，(1，4)，(1，5)，
(2，4)，(2，5))
```

zip算子

作用：将两个RDD中的元素分别合并在一起，RDD的数量必须相等。

需求：创建两个RDD进行拉链操作。

(1) 创建两个RDD

```
scala> val rdd1=sc.parallelize(0to2)
scala> val rdd2=sc.parallelize(3to5)
```

(2) 求笛卡尔积，查看结果

```
scala> rdd1.zip(rdd2).collect
res1:Array[(Int，Int)]=Array((0，3)，(1，4)，(2，5))
```

大多数的算子操作是作用在任意类型的RDD上的，但是只有一些比较特殊的算子只能作用在kv类型的RDD上，这些算在大多涉及到shuffle操作。

Key-Value类型

partitionBy算子

作用：对pairRDD进行分区操作，如果原有的partionRDD的分区器和传入的分区器相同，则返回原pairRDD，否则会生成ShuffleRDD，即会产生shuffle过程。下面是partitionBy算子的源码：

```
def partitionBy(partitioner:Partitioner):RDD[(K，V)]=self.withScope{
if(self.partitioner==Some(partitioner)){
        self
}else{
    new ShuffledRDD[K，V，V](self，partitioner)
}
}
```

需求：创建一个pairRDD并对重新分区。

(1) 创建pairRDD

```
val rdd=sc.parallelize(Array((1，"a")，(2，"b")，(3，"c")))
```

(2) 查看分区数

```
scala> rdd.partitions.length
res0:Int=2
```

(3) 传入分区器，重新分区

```
scala> rdd.partitionBy(neworg.apache.spark.HashPartitioner(3))
res1:org.apache.spark.rdd.RDD[(Int，
String)]=ShuffledRDD[1]atpartitionByat<console>:27
```

(4) 重新查看分区数

```
scala> res1.partitions.length
res2:Int=3
```

reduceByKey算子

作用：把相同key的value聚合到一起。

需求：分别统计每个车辆品牌的数目。

(1) 通过List创建pairRDD

```
val rdd=sc.parallelize(List(("bmw", 1), ("honda", 5), ("benz", 5), ("bmw", 3),
("benz", 2)))
```

(2) 统计每个品牌的车辆数目

```
scala> val result=rdd.reduceByKey(_+_)
```

(3) 查看结果

```
scala> result.collect
result:Array[(String, Int)]=Array((honda, 5), (bmw, 4), (benz, 7))
```

注意：(+)可以被替换为(x, y)=>x+y，前者是后者的一种简写方式。

groupByKey算子

作用：按照key进行分组。

需求：对每个车辆品牌的进行分组。

(1) 通过List创建pairRDD

```
val rdd=sc.parallelize(List(("bmw", 1), ("honda", 1), ("benz", 1), ("bmw", 2),
("benz", 2)))
```

(2) 对每个车辆品牌的进行分组

```
scala> val result=rdd.groupByKey()
```

(3) 查看结果

```
scala> result.collect

result:Array[(String, Iterable[Int])]=Array((honda, CompactBuffer(1)), (bmw,
CompactBuffer(1, 2)), (benz, CompactBuffer(1, 2)))
```

注意：

a. 如果一个key对应的值太多很容易造成内存溢出。

b. 如果是要每个key上进行聚合操作，应该选择aggregateByKey或者reduceByKey，因为它们在进行分区的时候先进行聚合。

aggregateByKey算子

作用：下面是aggregateByKey算子的源码：

```
defaggregateByKey[U:ClassTag](zeroValue:U，numPartitions:Int)
       (seqOp:(U，V)=>U，combOp:(U，U)=>U):RDD[(K，U)]
```

aggregateByKey的三个参数解释：

第一个参数：每个分区中不同key的初始值。

第二个参数：分区内对相同key的值的自定义函数(需要带上第一个参数)。

第三个参数：不同分区中相同key的值的自定义函数。

需求：计算数值型RDD相同key的平均值。

(1) 创建具有两个分区的RDD

```
scala> val rdd=sc.parallelize(List(("a", 1), ("a", 2), ("b", 3), ("b", 5), ("c",
5), ("c", 5))，2)
```

(2) 调用aggregateByKey算子计算key的总和和个数

```
scala> val result=rdd.aggregateByKey((0，0))((acc，value)
=>(acc._1+value，acc._2+1)，(par1，par2)=>(par1._1+par2._1，par1._2+par2._2))

scala> result.collect
res1:Array[(String，(Int，Int))]=Array((b，(8，2))，(a，(3，2))，(c，(10，2)))
```

(3) 使用map算子计算平均值

```
scala> val avg=result.map(x=>(x._1，x._2._1/x._2._2))

#每个key的平均值
scala> avg.collect
res2:Array[(String，Int)]=Array((b，4)，(a，1)，(c，5))
```

combineByKey算子

作用：对于相同K的V值合并成一个集合

算子参数：

(createCombiner: V => C， mergeValue: (C， V) => C， mergeCombiners: (C， C) => C)

第一个参数：

combineByKey()会遍历分区中的所有元素，因此每个元素的键要么还没有遇到过，要么就和之前的某个元素的键相同。如果这是一个新的元素，combineByKey()会使用一个叫作createCombiner()的函数来创建那个键对应的累加器的初始值。

第二个参数：

如果这是一个在处理当前分区之前已经遇到的键，它会使用mergeValue()方法将该键的累加器对应的当前值与这个新的值进行合并。

第三个参数：

由于每个分区都是独立处理的，因此对于同一个键可以有多个累加器。如果有两个或者更多的分区都有对应同一个键的累加器，就需要使用用户提供的 mergeCombiners() 方法将各个分区的结果进行合并。

需求：计算数值型RDD相同key的平均值。

(1) 创建具有两个分区的RDD

```
scala>  val rdd= sc.parallelize(Array(("a", 1), ("a", 2), ("b", 3), ("b", 5),
("c", 5), ("c", 5)),2)
```

(2) 调用combineByKey算子计算key的总和和个数

```
scala>  val combine = rdd.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))
combine: org.apache.spark.rdd.RDD[(String, (Int, Int))] = ShuffledRDD[13] at
combineByKey at <console>:26

scala> combine.collect
res5: Array[(String, (Int, Int))] = Array((b,(8,2)), (a,(3,2)), (c,(10,2)))
```

(3) 使用map算子计算平均值

```
scala> val avg = combine.map{case (key,value) =>
(key,value._1/value._2.toDouble)}
avg:org.apache.spark.rdd.RDD[(String, Double)]= MapPartitionsRDD[14] at map at
<console>:28

scala> avg.collect
res6: Array[(String, Double)] = Array((b,4.0), (a,1.5), (c,5.0))
```

sortByKey算子

作用：返回一个按照Key进行排序的RDD，其中K必须实现Ordered接口。

需求：创建一个数值型RDD并对其排序。

(1) 创建一个数值型RDD

```
scala>  val rdd = sc.parallelize(Array((5,"a"),(3,"c"),(2,"b"),(6,"d")))
```

(2) 按照Key进行正序排序

```
scala> rdd.sortByKey(true).collect
res7: Array[(Int, String)] = Array((2,b), (3,c), (5,a), (6,d))
```

(3) 按照Key进行倒序排序

```
scala> rdd.sortByKey(false).collect
res8: Array[(Int, String)] = Array((6,d), (5,a), (3,c), (2,b))
```

mapValues算子

作用：对于（K，V）形式的类型只对V进行操作

需求：创建一个键值型RDD，将value后面加上后缀@

(1) 创建RDD

```
scala> val rdd = sc.parallelize(Array((1,"a"),(2,"b"),(3,"c")))
rdd: org.apache.spark.rdd.RDD[(Int, String)] = ParallelCollectionRDD[0] at
parallelize at <console>:24
```

(2) 对value字符串后面添加上@

```
scala> rdd.mapValues(_ + "@")
res0: org.apache.spark.rdd.RDD[(Int, String)] = MapPartitionsRDD[1] at mapValues
at <console>:27
```

(3) 查结果

```
scala> res0.collect
res1: Array[(Int, String)] = Array((1,a@), (2,b@), (3,c@))
```

join算子

作用：在类型为（K,V1）和（K,V2）的RDD上调用，返回一个（K,（V1，V2））

的RDD。

需求：创建键值型RDD，然后将key相同的数据合并到一起。

(1) 创建第一个RDD

```
scala> val rdd = sc.parallelize(Array((1,"a"),(2,"b"),(3,"c")))
rdd: org.apache.spark.rdd.RDD[(Int, String)] = ParallelCollectionRDD[2] at
parallelize at <console>:24
```

(2) 创建第二个RDD

```
scala> val rdd1 = sc.parallelize(Array((1,"x"),(2,"y"),(4,"z")))
rdd1: org.apache.spark.rdd.RDD[(Int, String)] = ParallelCollectionRDD[3] at
parallelize at <console>:24
```

(3) 合并两个RDD，并打印结果

```
scala> rdd.join(rdd1).collect()
res2: Array[(Int, (String, String))] = Array((2,(b,y)), (1,(a,x)))
```