> 无论多么巧妙回避战斗的展示，人生中总会有几次正面战斗的

RDD数据分区

> Spark目前支持Hash分区和Range分区，用户也可以自定义分区，Hash分区为当前的默认分区，Spark中分区器直接决定了RDD中分区的个数、RDD中每条数据经过Shuffle过程属于哪个分区和Reduce的个数

注意：

(1) 只有Key-Value类型的RDD才有分区器的，非Key-Value类型的RDD分区的值是None
(2) 每个RDD的分区ID范围：0~numPartitions-1，决定这个值是属于那个分区的。

查看RDD分区

（1）创建一个pairRDD

```
scala> val pairs = sc.parallelize(List((1,1),(2,2),(3,3)))
pairs: org.apache.spark.rdd.RDD[(Int, Int)] = ParallelCollectionRDD[3] at
parallelize at <console>:24
```

（2）查看RDD的分区器

```
scala> pairs.partitioner
res1: Option[org.apache.spark.Partitioner] = None
```

（3）导入HashPartitioner类

```
scala> import org.apache.spark.HashPartitioner
```

（4）使用HashPartitioner对RDD进行重新分区

```
scala> val partitioned = pairs.partitionBy(new HashPartitioner(2))
partitioned: org.apache.spark.rdd.RDD[(Int, Int)] = ShuffledRDD[4] at
partitionBy at <console>:27
```

（5）查看重新分后RDD的分区器

```
scala> partitioned.partitioner
res2: Option[org.apache.spark.Partitioner] =
Some(org.apache.spark.HashPartitioner@2)
```

Hash分区

> HashPartitioner分区的原理：对于给定的key，计算其hashCode，并除以分区的个数取余，如果余数小于0，则用余数+分区的个数（否则加0），最后返回的值就是这个key所属的分区ID

（1）查看分区器

```
scala> nopar.partitioner
res20: Option[org.apache.spark.Partitioner] = None
```

（2）创建RDD，8个分区

```
scala> val nopar = sc.parallelize(List((1,3),(1,2),(2,4),(2,3),(3,6),(3,8)),8)
nopar: org.apache.spark.rdd.RDD[(Int, Int)] = ParallelCollectionRDD[10] at
parallelize at <console>:24
```

（3）RDD拼接分区号，可以看出0号分区和4号分区无数据

```
scala>nopar.mapPartitionsWithIndex((index,iter)=>{ Iterator(index.toString+" :
"+iter.mkString("|")) }).collect
res0: Array[String] = Array("0 : ", 1 : (1,3), 2 : (1,2), 3 : (2,4), "4 : ", 5 :
(2,3), 6 : (3,6), 7 : (3,8))
```

（4）使用HashPartititioner对数据重新进行分区，变成7个分区

```
scala> val hashpar = nopar.partitionBy(new org.apache.spark.HashPartitioner(7))
hashpar: org.apache.spark.rdd.RDD[(Int, Int)] = ShuffledRDD[12] at partitionBy
at <console>:26
```

（5）查看分区器

```
scala> hashpar.partitioner
res21: Option[org.apache.spark.Partitioner] =
Some(org.apache.spark.HashPartitioner@7)
```

（6）查看每个分区长度【为什么结果是这样？给大家留个思考空间，不会的可以私聊我】

```
scala> hashpar.mapPartitions(iter => Iterator(iter.length)).collect()
res19: Array[Int] = Array(0, 2, 2, 2, 0, 0, 0)
```

Range分区

- HashPartitioner分区弊端：可能导致每个分区中数据量的不均匀，极端情况下会导致某些分区拥有RDD的全部数据。
- RangePartitioner作用：将一定范围内的数映射到某一个分区内，尽量保证**每个分区中数据量的均匀**，而且分区与分区之间是有序的，一个分区中的元素肯定都是比另一个分区内的元素小或者大，但是分区内的元素是不能保证顺序的。简单的说就是将一定范围内的数映射到某一个分区内。实现过程为：
  第一步：先从整个RDD中抽取出样本数据，将样本数据排序，计算出**每个分区的最大key值**，形成一个Array[KEY]类型的**数组变量rangeBounds**；
  第二步：判断**key在rangeBounds中所处的范围**，给出该key值在下一个RDD中的分区id下标；该分区器要求RDD中的**KEY类型必须是可以排序的**!!!很大的局限性（元组工作少用）

自定义分区

> 要实现自定义的分区器，需要继承org.apache.spark.Partitioner类并实现下面三个方法。

（1）numPartitions:Int:返回创建出来的分区数。

（2）getPartition(key:Any):Int:返回给定键的分区编号(0到分区数-1)。

（3）equals():Java判断相等性的标准方法。这个方法的实现非常重要，Spark需要用这个方法来检查你的分区器对象是否和其他分区器实例相同，这样Spark才可以判断两个RDD的分区方式是否相同。\

需求：将相同后缀的数据写入相同的文件，通过将相同后缀的数据分区到相同的分区并保存输出来实现。

（1）创建一个pairRDD

```
scala> val data=sc.parallelize(Array((1，1)，(2，2)，(3，3)，(4，4)，(5，5)，(6，
6)))

data:org.apache.spark.rdd.RDD[(Int，
Int)]=ParallelCollectionRDD[3]atparallelizeat<console>:24
```

（2）定义一个自定义分区类

```
scala> :paste

class CustomerPartitioner(numParts:Int) extends org.apache.spark.Partitioner{
//覆盖分区数
overridedefnumPartitions:Int=numParts

//覆盖分区号获取函数

override def getPartition(key:Any):Int={
val ckey:String=key.toString
ckey.substring(ckey.length-1).toInt%numParts
}}

defined classCustomerPartitioner
```

（3）将RDD使用自定义的分区类进行重新分区

```
scala> val par=data.partitionBy(new CustomerPartitioner(2))

par:org.apache.spark.rdd.RDD[(Int，
Int)]=ShuffledRDD[2]atpartitionByat<console>:27
```

（4）查看重新分区后的数据分布

```
scala> par.mapPartitionsWithIndex((index，items)=>items.map((index，_))).collect

res3:Array[(Int，(Int，Int))]=Array((0，(2，2))，(0，(4，4))，(0，(6，6))，(1，(1，
1))，(1，(3，3))，(1，(5，5)))
```

使用自定义的Partitioner是很容易的:只要把它传给partitionBy()方法即可。Spark中有许多依赖于数据混洗的方法，比如join()和groupByKey()，它们也可以接收一个可选的Partitioner对象来控制输出数据的分区方式。