



链滴

spark 算子详解 -----Transformation 算子介绍

作者: [18582596683](#)

原文链接: <https://ld246.com/article/1543743581730>

来源网站: [链滴](#)

许可协议: [署名-相同方式共享 4.0 国际 \(CC BY-SA 4.0\)](#)

一、Value数据类型的Transformation算子

1.输入分区与输出分区一对一类型的算子

1.1.map算子

功能：map是对RDD中的每个元素都执行一个指定的函数来产生一个新的RDD，任何原RDD中的元在新RDD中都有且仅有一个元素与之对应。

源码：

```
>
/***
 * Return a new RDD by applying a function to all elements of this RDD.
 */
def map[U: ClassTag](f: T => U): RDD[U] = withScope {
  val cleanF = sc.clean(f)
  new MapPartitionsRDD[U, T](this, (context, pid, iter) => iter.map(cleanF))
}
```

示例：

```
>
scala> val a = sc.parallelize(1 to 10,2)
a: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[0] at parallelize at <console>:24
>
scala> val b = a.map(_ * 2)
b: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[1] at map at <console>:25
>
scala> a.collect
res0: Array[Int] = Array(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
>
scala> b.collect
res1: Array[Int] = Array(2, 4, 6, 8, 10, 12, 14, 16, 18, 20)
```

1.2.flatMap算子

功能：将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，生成MapPartitionsRDD。

源码：

```
>
/***
 * Return a new RDD by first applying a function to all elements of this * RDD, and then flattening the results.
 */
def flatMap[U: ClassTag](f: T => TraversableOnce[U]): RDD[U] = withScope {
  val cleanF = sc.clean(f)
  new MapPartitionsRDD[U, T](this, (context, pid, iter) => iter.flatMap(cleanF))
}
```

示例：

```
>
```

```
scala> val a = sc.parallelize(1 to 5)
a: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[2] at parallelize at <console>:24
>
scala> val b = a.flatMap(x => 1 to x)
b: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[3] at flatMap at <console>:25
>
scala> b.collect
res2: Array[Int] = Array(1, 1, 2, 1, 2, 3, 1, 2, 3, 4, 1, 2, 3, 4, 5)
```

1.3.mapPartitions算子

功能：mapPartitions是map的一个变种。map的输入函数是应用于RDD中每个元素，而mapPartitions的输入函数是应用于每个分区。mapPartitions获取么个分区的迭代器，在函数中通过这个分区整体迭代器对整个分区的元素进行操作。

源码：

```
>
/** 
 * Return a new RDD by applying a function to each partition of this RDD. ** `preservesPartitioning` indicates whether the input function preserves the partitioner, which
 * should be `false` unless this is a pair RDD and the input function doesn't modify the keys.
 */def mapPartitions[U: ClassTag](
  f: Iterator[T] => Iterator[U],
  preservesPartitioning: Boolean = false): RDD[U] = withScope {
  val cleanedF = sc.clean(f)
  new MapPartitionsRDD(
    this,
    (context: TaskContext, index: Int, iter: Iterator[T]) => cleanedF(iter),
    preservesPartitioning)
}
```

示例：

```
>
scala> val a = sc.parallelize(1 to 6, 3)
a: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[6] at parallelize at <console>:24
>
scala> def doubleFunc(iter: Iterator[Int]): Iterator[(Int, Int)] = {
  var res = List[(Int, Int)]()
  while (iter.hasNext) {
    val cur = iter.next;
    res ::= (cur, cur*2)
  }
  res.iterator
}
doubleFunc: (iter: Iterator[Int])Iterator[(Int, Int)]
>
scala> val result = a.mapPartitions(doubleFunc)
result: org.apache.spark.rdd.RDD[(Int, Int)] = MapPartitionsRDD[7] at mapPartitions at <console>:27
>
scala> println(result.collect().mkString)
(2,4)(1,2)(4,8)(3,6)(6,12)(5,10)
```

1.4.mapPartitionsWithIndex算子

功能：函数作用同mapPartitions，不过提供了两个参数，第一个参数为分区的索引，第二个参数为入函数，即对每个分区操作的函数。

源码：

```
>
/**
 * Return a new RDD by applying a function to each partition of this RDD, while tracking the i
dex * of the original partition. ** `preservesPartitioning` indicates whether the input function
reserves the partitioner, which
 * should be `false` unless this is a pair RDD and the input function doesn't modify the keys.
 */
def mapPartitionsWithIndex[U: ClassTag](
  f: (Int, Iterator[T]) => Iterator[U],
  preservesPartitioning: Boolean = false): RDD[U] = withScope {
  val cleanedF = sc.clean(f)
  new MapPartitionsRDD(
    this,
    (context: TaskContext, index: Int, iter: Iterator[T]) => cleanedF(index, iter),
    preservesPartitioning)
}
```

示例：

```
>
scala> val a = sc.parallelize(1 to 9, 3)
a: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[4] at parallelize at <console>:24
>
scala> def mapPartIndexFunc(i1:Int,iter: Iterator[Int]):Iterator[(Int,Int)] ={
    |   val result = List[(Int, Int)]()
    |   var i = 0
    |   while(iter.hasNext){
    |     i += iter.next()
    |   }
    |   result::(i1, i).iterator
    | }
mapPartIndexFunc: (i1: Int, iter: Iterator[Int])Iterator[(Int, Int)]
>
scala> val b = a.mapPartitionsWithIndex(mapPartIndexFunc)
b: org.apache.spark.rdd.RDD[(Int, Int)] = MapPartitionsRDD[5] at mapPartitionsWithIndex at <
onsole>:27
>
scala> b.foreach(println(_))
(0,6)
(1,15)
(2,24)
```

1.5.glm算子

功能：将每个分区内的元素组成一个数组，分区数不变。

源码：

```
>
```

```

/**
 * Return an RDD created by coalescing all elements within each partition into an array.
 */
def glom(): RDD[Array[T]] = withScope {
  new MapPartitionsRDD[Array[T], T](this, (context, pid, iter) => Iterator(iter.toArray))
}

```

示例：

```

>
scala> val a = sc.parallelize(1 to 9, 3)
a: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[2] at parallelize at <console>:24
>
scala> a.collect
res2: Array[Int] = Array(1, 2, 3, 4, 5, 6, 7, 8, 9)
>
scala> val b = a.glom
b: org.apache.spark.rdd.RDD[Array[Int]] = MapPartitionsRDD[3] at glom at <console>:25
>
scala> b.collect
res3: Array[Array[Int]] = Array(Array(1, 2, 3), Array(4, 5, 6), Array(7, 8, 9))

```

1.6.randomSplit算子

功能：根据weight（权重值）将一个RDD划分成多个RDD，权重越高划分得到的元素较多的几率就越大。

- 1.需要注意的是第一个参数weight数组内数据的加和应为1。
- 2.第二个参数seed是可选参数，作为random的种子，如果每次随机的种子相同，生成的随机数序列是相同的。

源码：

```

>
/** 
 * Randomly splits this RDD with the provided weights. ** @param weights weights for splits,
will be normalized if they don't sum to 1
 * @param seed random seed
** @return split RDDs in an array
*/def randomSplit(
  weights: Array[Double],
  seed: Long = Utils.random.nextLong): Array[RDD[T]] = {
  require(weights.forall(_ >= 0),
    s"Weights must be nonnegative, but got ${weights.mkString("[", ", ", ", ", "]")}")
  require(weights.sum > 0,
    s"Sum of weights must be positive, but got ${weights.mkString("[", ", ", ", "]")}")
}

>
withScope {
  val sum = weights.sum
  val normalizedCumWeights = weights.map(_ / sum).scanLeft(0.0d)(_ + _)
  normalizedCumWeights.sliding(2).map { x =>
    randomSampleWithRange(x(0), x(1), seed)
  }.toArray
}
}

```

示例：

```
>
scala> val a = sc.parallelize(1 to 9)
a: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[26] at parallelize at <console>:24
>
scala> val b= a.randomSplit(Array(0.2,0.3,0.5))
b: Array[org.apache.spark.rdd.RDD[Int]] = Array(MapPartitionsRDD[27] at randomSplit at <co
sole>:25, MapPartitionsRDD[28] at randomSplit at <console>:25, MapPartitionsRDD[29] at ra
domSplit at <console>:25)
>
scala> b.size
res20: Int = 3
>
scala> b(0).collect
res21: Array[Int] = Array(2, 3, 8)
>
scala> b(1).collect
res22: Array[Int] = Array(1, 5, 9)
>
scala> b(2).collect
res23: Array[Int] = Array(4, 6, 7)
>
>下面是测试相同的种子会生成相同的结果
scala> val c= a.randomSplit(Array(0.2,0.8), 2)
c: Array[org.apache.spark.rdd.RDD[Int]] = Array(MapPartitionsRDD[30] at randomSplit at <co
sole>:25, MapPartitionsRDD[31] at randomSplit at <console>:25)
>
scala> c(0).collect
res25: Array[Int] = Array(2, 3, 7)
>
scala> c(1).collect
res26: Array[Int] = Array(1, 4, 5, 6, 8, 9)
>
scala> val d= a.randomSplit(Array(0.2,0.8), 2)
d: Array[org.apache.spark.rdd.RDD[Int]] = Array(MapPartitionsRDD[32] at randomSplit at <co
sole>:25, MapPartitionsRDD[33] at randomSplit at <console>:25)
>
scala> d(0).collect
res27: Array[Int] = Array(2, 3, 7)
>
scala> d(1).collect
res28: Array[Int] = Array(1, 4, 5, 6, 8, 9)
>
scala> val e= a.randomSplit(Array(0.2,0.8), 3)
e: Array[org.apache.spark.rdd.RDD[Int]] = Array(MapPartitionsRDD[34] at randomSplit at <co
sole>:25, MapPartitionsRDD[35] at randomSplit at <console>:25)
>
scala> e(0).collect
res29: Array[Int] = Array(1, 5, 9)
>
scala> e(1).collect
res30: Array[Int] = Array(2, 3, 4, 6, 7, 8)
```

2.输入分区与输出分区多对一类型的算子

2.1.union算子

功能：求两个算子的并集，并且不去重，需要保证两个 RDD 元素的数据类型相同。

源码：

```
>
/**
 * Return the union of this RDD and another one. Any identical elements will appear multiple
 * times (use `distinct()` to eliminate them).
 */
def union(other: RDD[T]): RDD[T] = withScope {
  sc.union(this, other)
}
```

示例：

```
>
scala> val a = sc.parallelize(1 to 5)
a: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[36] at parallelize at <console>:24
>
scala> val b = sc.parallelize(3 to 8)
b: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[37] at parallelize at <console>:24
>
scala> val c = a.union(b)
c: org.apache.spark.rdd.RDD[Int] = UnionRDD[38] at union at <console>:27
>
scala> c.collect
res31: Array[Int] = Array(1, 2, 3, 4, 5, 3, 4, 5, 6, 7, 8)
```

2.2.cartesian算子

功能：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Cartesia RDD。

源码：

```
>
/**
 * Return the Cartesian product of this RDD and another one, that is, the RDD of all pairs of
 * elements (a, b) where a is in `this` and b is in `other`.
 */
def cartesian[U: ClassTag](other: RDD[U]): RDD[(T, U)] = withScope {
  new CartesianRDD(sc, this, other)
}
```

示例：

```
>
scala> val rdd2 = sc.parallelize(5 to 9,1)
rdd2: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[37] at parallelize at <console>:2
>
scala> val rdd3 = rdd1.cartesian(rdd2)
rdd3: org.apache.spark.rdd.RDD[(Int, Int)] = CartesianRDD[38] at cartesian at <console>:27
>
scala> rdd3.collect
```

```
res15: Array[(Int, Int)] = Array((1,5), (1,6), (1,7), (1,8), (1,9), (2,5), (2,6), (2,7), (2,8), (2,9), (3,5), (3,6), (3,7), (3,8), (3,9))
```

3.输入分区与输出分区多对多类型的算子

3.1.groupBy算子

功能：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 相同的元素分一组。

源码：

```
>
/**
 * Return an RDD of grouped items. Each group consists of a key and a sequence of elements
 * mapping to that key. The ordering of elements within each group is not guaranteed, and
 * may even differ each time the resulting RDD is evaluated.
 *
 * @note This operation may be very expensive. If you are grouping in order to perform an
 * aggregation (such as a sum or average) over each key, using `PairRDDFunctions.aggregate
 * yKey` or `PairRDDFunctions.reduceByKey` will provide much better performance.
 */
def groupBy[K](f: T => K)(implicit kt: ClassTag[K]): RDD[(K, Iterable[T])] = withScope {
  groupBy[K](f, defaultPartitioner(this))
}
```

示例：

```
>
scala> val rdd1 = sc.parallelize(1 to 9, 3)
rdd1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[39] at parallelize at <console>:2

>
scala> val rdd2 = rdd1.groupBy(x => { if (x % 2 == 0) "even" else "odd" })
rdd2: org.apache.spark.rdd.RDD[(String, Iterable[Int])] = ShuffledRDD[41] at groupBy at <con
ole>:25
>
scala> rdd2.collect
res17: Array[(String, Iterable[Int])] = Array((even,CompactBuffer(2, 4, 6, 8)), (odd,CompactBuffe
(1, 3, 5, 7, 9)))
```

3.2.coalesce算子

功能：该函数用于将RDD进行重分区，默认不进行shuffle。

- 如果分区数减少，默认不进行shuffle，此时父RDD和子RDD之间是窄依赖。比如：1000个分区被新设置成10个分区，这样不会发生shuffle。
- 如果分区数量增大时，比如Rdd的原分区数是100，想设置成1000，此时需要把shuffle设置成true行，因为如果设置成false，不会进行shuffle操作，此时父RDD和子RDD之间是窄依赖，这时并不会增加RDD的分区。

源码：

```
>
/**
```

```

* Return a new RDD that is reduced into `numPartitions` partitions.
*
* This results in a narrow dependency, e.g. if you go from 1000 partitions
* to 100 partitions, there will not be a shuffle, instead each of the 100
* new partitions will claim 10 of the current partitions. If a larger number
* of partitions is requested, it will stay at the current number of partitions.
*
* However, if you're doing a drastic coalesce, e.g. to numPartitions = 1,
* this may result in your computation taking place on fewer nodes than
* you like (e.g. one node in the case of numPartitions = 1). To avoid this,
* you can pass shuffle = true. This will add a shuffle step, but means the
* current upstream partitions will be executed in parallel (per whatever
* the current partitioning is).
*
* @note With shuffle = true, you can actually coalesce to a larger number
* of partitions. This is useful if you have a small number of partitions,
* say 100, potentially with a few partitions being abnormally large. Calling
* coalesce(1000, shuffle = true) will result in 1000 partitions with the
* data distributed using a hash partitioner. The optional partition coalescer
* passed in must be serializable.
*/
def coalesce(numPartitions: Int, shuffle: Boolean = false,
  partitionCoalescer: Option[PartitionCoalescer] = Option.empty)
(implicit ord: Ordering[T] = null)
: RDD[T] = withScope {
  require(numPartitions > 0, s"Number of partitions ($numPartitions) must be positive.")
  if (shuffle) {
    /** Distributes elements evenly across output partitions, starting from a random partition. */
    val distributePartition = (index: Int, items: Iterator[T]) => {
      var position = new Random(hashing.byteswap32(index)).nextInt(numPartitions)
      items.map { t =>
        // Note that the hash code of the key will just be the key itself. The HashPartitioner
        // will mod it with the number of total partitions. position = position + 1
        (position, t)
      }
    } : Iterator[(Int, T)]
  }
  >
  // include a shuffle step so that our upstream tasks are still distributed
  new CoalescedRDD(
    new ShuffledRDD[Int, T, T](
      mapPartitionsWithIndexInternal(distributePartition, isOrderSensitive = true),
      new HashPartitioner(numPartitions),
      numPartitions,
      partitionCoalescer).values
    ) else {
      new CoalescedRDD(this, numPartitions, partitionCoalescer)
    }
}

```

示例:

```

>
scala> val a = sc.parallelize(1 to 9, 3)
a: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[8] at parallelize at <console>:24
>
```

```
scala> a.partitions.size
res11: Int = 3
>
scala> val b = a.coalesce(1)
b: org.apache.spark.rdd.RDD[Int] = CoalescedRDD[9] at coalesce at <console>:25
>
scala> b.partitions.size
res12: Int = 1
>
scala> val c = a.coalesce(4)
c: org.apache.spark.rdd.RDD[Int] = CoalescedRDD[10] at coalesce at <console>:25
>
scala> c.partitions.size
res13: Int = 3
>
scala> val d = a.coalesce(4, true)
d: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[14] at coalesce at <console>:25
>
scala> d.partitions.size
res14: Int = 4
```

3.3.repartition算子

功能： repartition方法其实就是调用了coalesce方法,shuffle设置为true的情况。

源码：

```
>
/**
 * Return a new RDD that has exactly numPartitions partitions. ** Can increase or decrease the
 * level of parallelism in this RDD. Internally, this uses * a shuffle to redistribute data. ** If you are
 * decreasing the number of partitions in this RDD, consider using `coalesce`,
 * which can avoid performing a shuffle. ** TODO Fix the Shuffle+Repartition data loss issue
 * described in SPARK-23207.
 */
def repartition(numPartitions: Int)(implicit ord: Ordering[T] = null): RDD[T] = withScope {
  coalesce(numPartitions, shuffle = true)
}
```

示例：

```
>
scala> val a = sc.parallelize(1 to 9, 3)
a: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[15] at parallelize at <console>:24
>
scala> val b = a.repartition(1)
b: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[19] at repartition at <console>:25
>
scala> b.partitions.size
res15: Int = 1
>
scala> val c = a.repartition(4)
c: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[23] at repartition at <console>:25
>
scala> c.partitions.size
res16: Int = 4
```

4.输出分区为输入分区子集型的算子

4.1.filter算子

功能：filter 是对RDD中的每个元素都执行一个指定的函数来过滤产生一个新的RDD。任何原RDD中元素在新RDD中都有且只有一个元素与之对应。

源码：

```
/**  
 * Return a new RDD containing only the elements that satisfy a predicate.  
 */  
def filter(f: T => Boolean): RDD[T] = withScope {  
    val cleanF = sc.clean(f)  
    new MapPartitionsRDD[T, T](  
        this,  
        (context, pid, iter) => iter.filter(cleanF),  
        preservesPartitioning = true)  
}
```

示例：

```
>  
scala> val rdd1 = sc.parallelize(1 to 9, 3)  
rdd1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[42] at parallelize at <console>:2  
  
>  
scala> val rdd2 = rdd1.filter(_ % 2 == 0)  
rdd2: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[43] at filter at <console>:25  
>  
scala> rdd2.collect  
res18: Array[Int] = Array(2, 4, 6, 8)
```

4.2.distinct算子

功能：distinct将RDD中的元素进行去重操作。

源码：

```
>  
/**  
 * Return a new RDD containing the distinct elements in this RDD.  
 */  
def distinct(): RDD[T] = withScope {  
    distinct(partitions.length)  
}
```

示例：

```
>  
scala> c.collect  
res31: Array[Int] = Array(1, 2, 3, 4, 5, 3, 4, 5, 6, 7, 8)  
>  
scala> val d = c.distinct()  
d: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[41] at distinct at <console>:25  
>  
scala> d.collect
```

```
res32: Array[Int] = Array(8, 1, 2, 3, 4, 5, 6, 7)
```

4.3.intersection算子

功能：求两个RDD的交集。

源码：

```
>
/** 
 * Return the intersection of this RDD and another one. The output will not contain any duplicate
 * elements, even if the input RDDs did.
 *
 * @note This method performs a shuffle internally.
 */
def intersection(other: RDD[T]): RDD[T] = withScope {
  this.map(v => (v, null)).cogroup(other.map(v => (v, null)))
    .filter { case (_, (leftGroup, rightGroup)) => leftGroup.nonEmpty && rightGroup.nonEmpty }
    .keys
}
```

示例：

```
>
scala> val a = sc.parallelize(1 to 5)
a: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[42] at parallelize at <console>:24
>
scala> val b = sc.parallelize(3 to 8)
b: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[43] at parallelize at <console>:24
>
scala> val c = a.intersection(b)
c: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[49] at intersection at <console>:27
>
scala> c.collect
res33: Array[Int] = Array(4, 5, 3)
```

4.4.subtract算子

功能：求两个RDD的差集。

源码：

```
>
/** 
 * Return an RDD with the elements from `this` that are not in `other`.
 * * Uses `this` partitioner/partition size, because even if `other` is huge, the resulting
 * RDD will be <= us.
 */
def subtract(other: RDD[T]): RDD[T] = withScope {
  subtract(other, partitioner.getOrElse(new HashPartitioner(partitions.length)))
}
```

示例：

```
>
scala> val a = sc.parallelize(1 to 5)
```

```
a: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[42] at parallelize at <console>:24
>
scala> val b = sc.parallelize(3 to 8)
b: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[43] at parallelize at <console>:24
>
scala> val d = a.subtract(b)
d: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[53] at subtract at <console>:27
>
scala> d.collect
res34: Array[Int] = Array(1, 2)
```

4.5.sample算子

功能：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。

源码：

```
>
/** 
 * Return a sampled subset of this RDD.
 *
 * @param withReplacement can elements be sampled multiple times (replaced when sample
out)
 * @param fraction expected size of the sample as a fraction of this RDD's size
 * without replacement: probability that each element is chosen; fraction must be [0, 1]
 * with replacement: expected number of times each element is chosen; fraction must be gre
ter
 * than or equal to 0 * @param seed seed for the random number generator
 *
 * @note This is NOT guaranteed to provide exactly the fraction of the count
 * of the given [[RDD]].
 */
def sample(
    withReplacement: Boolean,
    fraction: Double,
    seed: Long = Utils.random.nextLong): RDD[T] = {
    require(fraction >= 0,
        s"Fraction must be nonnegative, but got ${fraction}")
    >
    withScope {
        require(fraction >= 0.0, "Negative fraction value: " + fraction)
        if (withReplacement) {
            new PartitionwiseSampledRDD[T, T](this, new PoissonSampler[T](fraction), true, seed)
        } else {
            new PartitionwiseSampledRDD[T, T](this, new BernoulliSampler[T](fraction), true, seed)
        }
    }
}
```

示例：

```
>
scala> val rdd1 = sc.parallelize(1 to 9, 3)
rdd1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[44] at parallelize at <console>:2
```

```

>
scala> val rdd2 = rdd1.sample(false, 0.3)
rdd2: org.apache.spark.rdd.RDD[Int] = PartitionwiseSampledRDD[45] at sample at <console>:
5
>
scala> rdd2.collect
res20: Array[Int] = Array(5, 8, 9)

```

4.6.takeSample算子

功能：和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。

源码：

```

>
/** 
 * Return a fixed-size sampled subset of this RDD in an array
 *
 * @param withReplacement whether sampling is done with replacement
 * @param num size of the returned sample
 * @param seed seed for the random number generator
 * @return sample of specified size in an array
 *
 * @note this method should only be used if the resulting array is expected to be small, as
 * all the data is loaded into the driver's memory.
 */
def takeSample(
  withReplacement: Boolean,
  num: Int,
  seed: Long = Utils.random.nextLong): Array[T] = withScope {
  val numStDev = 10.0
>
  require(num >= 0, "Negative number of elements requested")
  require(num <= (Int.MaxValue - (numStDev * math.sqrt(Int.MaxValue))).toInt,
    "Cannot support a sample size > Int.MaxValue - " +
    s"$numStDev * math.sqrt(Int.MaxValue)")
>
  if (num == 0) {
    new Array[T](0)
  } else {
    val initialCount = this.count()
    if (initialCount == 0) {
      new Array[T](0)
    } else {
      val rand = new Random(seed)
      if (!withReplacement && num >= initialCount) {
        Utils.randomizeInPlace(this.collect(), rand)
      } else {
        val fraction = SamplingUtils.computeFractionForSampleSize(num, initialCount,
          withReplacement)
        var samples = this.sample(withReplacement, fraction, rand.nextInt()).collect()
>
        // If the first sample didn't turn out large enough, keep trying to take samples;
      }
    }
  }
}

```

```
// this shouldn't happen often because we use a big multiplier for the initial size var numIter  
= 0  
while (samples.length < num) {  
logWarning(s"Needed to re-sample due to insufficient sample size. Repeat #$numIters")  
samples = this.sample(withReplacement, fraction, rand.nextInt()).collect()  
numIters += 1  
}  
Utils.randomizeInPlace(samples, rand).take(num)  
}  
}  
}
```

示例：

```
>  
scala> val rdd1 = sc.parallelize(1 to 9, 3)  
rdd1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[48] at parallelize at <console>:2  
  
>  
scala> val rdd2 = rdd1.takeSample(false, 4)  
rdd2: Array[Int] = Array(3, 1, 2, 9)
```

5.Cache型的算子

5.1.persist算子

功能：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。可以缓存到内存或者磁盘。

源码：

```
>  
/**  
 * Set this RDD's storage level to persist its values across operations after the first time  
 * it is computed. This can only be used to assign a new storage level if the RDD does not  
 * have a storage level set yet. Local checkpointing is an exception.  
 */  
def persist(newLevel: StorageLevel): this.type = {  
  if (isLocallyCheckpointed) {  
    // This means the user previously called localCheckpoint(), which should have already  
    // marked this RDD for persisting. Here we should override the old storage level with // one  
    // that is explicitly requested by the user (after adapting it to use disk). persist(LocalRDDCheckpo  
    ntData.transformStorageLevel(newLevel), allowOverride = true)  
  } else {  
    persist(newLevel, allowOverride = false)  
  }  
}
```

缓存等级：

```
>  
StorageLevel.DISK_ONLY  
StorageLevel.DISK_ONLY_2  
StorageLevel.MEMORY_ONLY  
StorageLevel.MEMORY_ONLY_2  
StorageLevel.MEMORY_AND_DISK  
StorageLevel.MEMORY_AND_DISK_2
```

StorageLevel.OFF_HEAP

5.2.cache算子

功能：将 RDD 元素从磁盘缓存到内存。相当于 persist(MEMORY_ONLY) 函数的功能。

源码：

```
>
/**
 * Persist this RDD with the default storage level (`MEMORY_ONLY`).
 */
def cache(): this.type = persist()
```

二、Key-Value数据类型的Transformation算子

1.输入分区与输出分区一对一类型的算子

1.1.mapValues算子

功能：该函数用于处理key-value的Value，原RDD中的Key保持不变，与新的Value一起组成新的RD中的元素。因此，该函数只适用于元素为key-value对的RDD。

源码：

```
>
/**
 * Pass each value in the key-value pair RDD through a map function without changing the keys;
 * this also retains the original RDD's partitioning.
 */
def mapValues[U](f: V => U): RDD[(K, U)] = self.withScope {
  val cleanF = self.context.clean(f)
  new MapPartitionsRDD[(K, U), (K, V)](self,
    (context, pid, iter) => iter.map { case (k, v) => (k, cleanF(v)) },
    preservesPartitioning = true)
}
```

示例：

```
>
scala> val rdd1 = sc.parallelize(List(("A", 1), ("B", 2), ("C", 3), ("D", 4)), 2)
rdd1: org.apache.spark.rdd.RDD[(String, Int)] = ParallelCollectionRDD[6] at parallelize at <console>:24
>
scala> val rdd2 = rdd1.mapValues(10 + _)
rdd2: org.apache.spark.rdd.RDD[(String, Int)] = MapPartitionsRDD[7] at mapValues at <console>:25
>
scala> rdd2.collect
res4: Array[(String, Int)] = Array((A,11), (B,12), (C,13), (D,14))
```

1.2.flatMapValues算子

功能： flatMapValues类似于mapValues，不同的在于flatMapValues应用于元素为KV对的RDD中Value。每个元素的Value被输入函数映射为一系列的值，然后这些值再与原RDD中的Key组成一系列的KV对。

源码：

```
>
/** 
 * Pass each value in the key-value pair RDD through a flatMap function without changing the
 * keys; this also retains the original RDD's partitioning.
 */
def flatMapValues[U](f: V => TraversableOnce[U]): RDD[(K, U)] = self.withScope {
  val cleanF = self.context.clean(f)
  new MapPartitionsRDD[(K, U), (K, V)](self,
    (context, pid, iter) => iter.flatMap { case (k, v) =>
      cleanF(v).map(x => (k, x))
    },
    preservesPartitioning = true)
}
```

示例：

```
>
scala> val rdd1 = sc.parallelize(List(("A", 1), ("B", 2), ("C", 3), ("D", 4)), 2)
rdd1: org.apache.spark.rdd.RDD[(String, Int)] = ParallelCollectionRDD[8] at parallelize at <con
sole>:24
>
scala> val rdd2 = rdd1.flatMapValues(1 to _)
rdd2: org.apache.spark.rdd.RDD[(String, Int)] = MapPartitionsRDD[9] at flatMapValues at <co
sole>:25
>
scala> rdd2.collect
res5: Array[(String, Int)] = Array((A,1), (B,1), (B,2), (C,1), (C,2), (C,3), (D,1), (D,2), (D,3), (D,4))
```

1.3.sortByKey算子

功能：该函数用于对Key-Value形式的RDD进行排序。

源码：

```
>
/** 
 * Return a new RDD by applying a function to each partition of this RDD, while tracking the i
dex
 * of the original partition.
 *
 * `preservesPartitioning` indicates whether the input function preserves the partitioner, which
 * should be `false` unless this is a pair RDD and the input function doesn't modify the keys.
 */
def mapPartitionsWithIndex[U: ClassTag](
  f: (Int, Iterator[T]) => Iterator[U],
  preservesPartitioning: Boolean = false): RDD[U] = withScope {
  val cleanedF = sc.clean(f)
  new MapPartitionsRDD(
    this,
    (context: TaskContext, index: Int, iter: Iterator[T]) => cleanedF(index, iter),
```

```
    preservesPartitioning)
}
```

示例:

```
>
scala> val rdd1 = sc.parallelize(List(("A", 1), ("B", 3), ("C", 2)))
rdd1: org.apache.spark.rdd.RDD[(String, Int)] = ParallelCollectionRDD[5] at parallelize at <console>:24
>
scala> val rdd2 = sc.parallelize(List(("B", 2), ("D", 1), ("E", 2)))
rdd2: org.apache.spark.rdd.RDD[(String, Int)] = ParallelCollectionRDD[6] at parallelize at <console>:24
>
scala> val rdd3 = rdd1 union rdd2
rdd3: org.apache.spark.rdd.RDD[(String, Int)] = UnionRDD[7] at union at <console>:27
    ^
>
scala> val rdd5 = rdd3.sortByKey(true)
rdd5: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[11] at sortByKey at <console>:25
>
scala> rdd5.collect
res3: Array[(String, Int)] = Array((A,1), (B,3), (B,2), (C,2), (D,1), (E,2))
```

1.4.sortBy算子

功能: sortBykey的升级版, 可以指定按key或者value排序。

源码:

```
>
/** 
 * Return this RDD sorted by the given key function.
 */
def sortBy[K](
  f: (T) => K,
  ascending: Boolean = true,
  numPartitions: Int = this.partitions.length)
(implicit ord: Ordering[K], ctag: ClassTag[K]): RDD[T] = withScope {
  this.keyBy[K](f)
  .sortByKey(ascending, numPartitions)
  .values
}
```

示例:

```
>
scala> val rdd1 = sc.parallelize(Array(("a",1),("b",2),("c",3),("a",4),("d",5),("b",6),("e",7),("c",8),("d",9)))
rdd1: org.apache.spark.rdd.RDD[(String, Int)] = ParallelCollectionRDD[12] at parallelize at <console>:24
>
scala> val rdd2 = rdd1.reduceByKey(_ + _)
rdd2: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[13] at reduceByKey at <console>:25
>
scala> val rdd3 = rdd2.sortBy(_.value, false)
```

```
rdd3: org.apache.spark.rdd.RDD[(String, Int)] = MapPartitionsRDD[18] at sortBy at <console>:5
>
scala> rdd3.collect
res4: Array[(String, Int)] = Array((d,14), (c,11), (b,8), (e,7), (a,5))
```

1.5.zip算子

功能：zip函数用于将两个非key-value的RDD，通过以一对对应的关系压缩为key-value的RDD，两个RDD的分区数需要相同，分区中的元素个数也要相等。

源码：

```
>
/** 
 * Return a new RDD by applying a function to each partition of this RDD, while tracking the index
 * of the original partition.
 *
 * `preservesPartitioning` indicates whether the input function preserves the partitioner, which
 * should be `false` unless this is a pair RDD and the input function doesn't modify the keys.
 */
def mapPartitionsWithIndex[U: ClassTag](
  f: (Int, Iterator[T]) => Iterator[U],
  preservesPartitioning: Boolean = false): RDD[U] = withScope {
  val cleanedF = sc.clean(f)
  new MapPartitionsRDD(
    this,
    (context: TaskContext, index: Int, iter: Iterator[T]) => cleanedF(index, iter),
    preservesPartitioning)
}
```

示例：

```
>
scala> val a = sc.makeRDD(List(1,2,3))
a: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[19] at makeRDD at <console>:24
>
scala> val b = sc.makeRDD(List("a","b","c"))
b: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[20] at makeRDD at <console>:2

>
scala> val c = a.zip(b)
c: org.apache.spark.rdd.RDD[(Int, String)] = ZippedPartitionsRDD2[21] at zip at <console>:27
>
scala> c.collect
res5: Array[(Int, String)] = Array((1,a), (2,b), (3,c))
```

1.6.zipPartitions算子

功能：zipPartitions函数将多个RDD按照partition组合成为新的RDD，该函数需要组合的RDD具有相同的分区数，但对于每个分区内的元素数量没有要求。

源码：

```
>
```

```

/** 
 * Return a new RDD by applying a function to each partition of this RDD, while tracking the index
 * of the original partition.
 *
 * `preservesPartitioning` indicates whether the input function preserves the partitioner, which
 * should be `false` unless this is a pair RDD and the input function doesn't modify the keys.
 */
def mapPartitionsWithIndex[U: ClassTag](
  f: (Int, Iterator[T]) => Iterator[U],
  preservesPartitioning: Boolean = false): RDD[U] = withScope {
  val cleanedF = sc.clean(f)
  new MapPartitionsRDD(
    this,
    (context: TaskContext, index: Int, iter: Iterator[T]) => cleanedF(index, iter),
    preservesPartitioning)
}

```

示例：

```

>
scala> val a = sc.parallelize(1 to 9, 3)
a: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[4] at parallelize at <console>:24
>
scala> def mapPartIndexFunc(i1:Int,iter: Iterator[Int]):Iterator[(Int,Int)] = {
    |   val result = List[(Int, Int)]()
    |   var i = 0
    |   while(iter.hasNext){
    |     i += iter.next()
    |   }
    |   result::(i1, i).iterator
    | }
mapPartIndexFunc: (i1: Int, iter: Iterator[Int])Iterator[(Int, Int)]
>
scala> val b = a.mapPartitionsWithIndex(mapPartIndexFunc)
b: org.apache.spark.rdd.RDD[(Int, Int)] = MapPartitionsRDD[5] at mapPartitionsWithIndex at <onsole>:27
>
scala> b.foreach(println(_))
(0,6)
(1,15)
(2,24)

```

1.7.zipWithIndex算子

功能：该函数将RDD中的元素和这个元素在RDD中的ID（索引号）组合成键/值对。

源码：

```

>
/** 
 * Zips this RDD with its element indices. The ordering is first based on the partition index
 * and then the ordering of items within each partition. So the first item in the first
 * partition gets index 0, and the last item in the last partition receives the largest index.
 * This is similar to Scala's zipWithIndex but it uses Long instead of Int as the index type.
 * This method needs to trigger a spark job when this RDD contains more than one partitions.

```

```

    * * @note Some RDDs, such as those returned by groupBy(), do not guarantee order of
    * elements in a partition. The index assigned to each element is therefore not guaranteed,
    * and may even change if the RDD is reevaluated. If a fixed ordering is required to guarantee
    * the same index assignments, you should sort the RDD with sortByKey() or save it to a file.
    */
def zipWithIndex(): RDD[(T, Long)] = withScope {
  new ZippedWithIndexRDD(this)
}

```

示例：

```

>
scala> val a = sc.parallelize(1 to 5,2)
a: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[24] at parallelize at <console>:24
>
scala> val b = a.zipWith
zipWithIndex  zipWithUniqueId
>
scala> val b = a.zipWithIndex()
b: org.apache.spark.rdd.RDD[(Int, Long)] = ZippedWithIndexRDD[25] at zipWithIndex at <con
ole>:25
>
scala> b.collect
res6: Array[(Int, Long)] = Array((1,0), (2,1), (3,2), (4,3), (5,4))

```

1.8.zipWithUniqueId算子

功能：该函数将RDD中元素和一个唯一ID组合成键/值对，该唯一ID生成算法如下：

每个分区中第一个元素的唯一ID值为：该分区索引号；

每个分区中第N个元素的唯一ID值为：(前一个元素的唯一ID值) + (该RDD总的分区数)；

源码：

```

>
/** 
 * Zips this RDD with generated unique Long ids. Items in the kth partition will get ids k, n+k,
 * 2*n+k, ..., where n is the number of partitions. So there may exist gaps, but this method
 * won't trigger a spark job, which is different from [[org.apache.spark.rdd.RDD#zipWithIndex]]
 *
 * @note Some RDDs, such as those returned by groupBy(), do not guarantee order of
 * elements in a partition. The unique ID assigned to each element is therefore not guaranteed
 * and may even change if the RDD is reevaluated. If a fixed ordering is required to guarantee
 * the same index assignments, you should sort the RDD with sortByKey() or save it to a file.
 */
def zipWithUniqueId(): RDD[(T, Long)] = withScope {
  val n = this.partitions.length.toLong
  this.mapPartitionsWithIndex { case (k, iter) =>
    Utils.getIteratorZipWithIndex(iter, 0L).map { case (item, i) =>
      (item, i * n + k)
    }
  }
}

```

示例：

```

>
scala> val a = sc.parallelize(1 to 5,2)
a: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[0] at parallelize at <console>:24
>
scala> val b = a.zipWithUniqueId()
b: org.apache.spark.rdd.RDD[(Int, Long)] = MapPartitionsRDD[1] at zipWithUniqueId at <console>:25
>
scala> b.collect
collect  collectAsMap  collectAsync
>
scala> b.collect
res0: Array[(Int, Long)] = Array((1,0), (2,2), (3,1), (4,3), (5,5))
>
//总分区数为2`  

//第一个分区第一个元素ID为0, 第二个分区第一个元素ID为1`  

//第一个分区第二个元素ID为0+2=2, 第一个分区第三个元素ID为2+2=4`  

//第二个分区第二个元素ID为1+2=3, 第二个分区第三个元素ID为3+2=5`
```

2.对单个RDD或两个RDD聚集的算子

2.1.combineByKey算子

功能：该函数用于将RDD[K,V]转换成RDD[K,C],这里的V类型和C类型可以相同也可以不同。该函数有个参数：

第一个参数：给定一个初始值，用函数生成初始值。

第二个参数：combinbe聚合逻辑。

第三个参数：reduce端聚合逻辑。

源码：

```

>
/**  

 * Generic function to combine the elements for each key using a custom set of aggregation  

 * functions. This method is here for backward compatibility. It does not provide combiner  

 * classtag information to the shuffle.  

 *  

 * @see `combineByKeyWithClassTag`  

 */  

def combineByKey[C](  

  createCombiner: V => C,  

  mergeValue: (C, V) => C,  

  mergeCombiners: (C, C) => C,  

  partitioner: Partitioner,  

  mapSideCombine: Boolean = true,  

  serializer: Serializer = null): RDD[(K, C)] = self.withScope {  

  combineByKeyWithClassTag(createCombiner, mergeValue, mergeCombiners,  

  partitioner, mapSideCombine, serializer)(null)  

}
```

>
-----参数说明：

createCombiner：组合器函数，用于将V类型转换成C类型，输入参数为RDD[K,V]中的V,输出为C
mergeValue：合并值函数，将一个C类型和一个V类型值合并成一个C类型，输入参数为(C,V)，输出C

mergeCombiners: 分区合并组合器函数，用于将两个C类型值合并成一个C类型，输入参数为(C,C)输出为C

numPartitions: 结果RDD分区数，默认保持原有的分区数

partitioner: 分区函数，默认为HashPartitioner

mapSideCombine: 是否需要在Map端进行combine操作，类似于MapReduce中的combine，默为true

示例：

```
>
scala> val rdd1 = sc.parallelize(List(1,2,2,3,3,3,3,4,4,4,4), 2)
rdd1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[10] at parallelize at <console>:2

>
scala> val rdd2 = rdd1.map(_,_)
rdd2: org.apache.spark.rdd.RDD[(Int, Int)] = MapPartitionsRDD[11] at map at <console>:25
>
scala> val rdd3 = rdd2.combineByKey(-_, (x:Int, y:Int) => x + y, (x:Int, y:Int) => x + y)
rdd3: org.apache.spark.rdd.RDD[(Int, Int)] = ShuffledRDD[12] at combineByKey at <console>:5
>
scala> rdd2.collect
res6: Array[(Int, Int)] = Array((1,1), (2,1), (2,1), (3,1), (3,1), (3,1), (3,1), (4,1), (4,1), (4,1), (4,1))
>
scala> rdd3.collect
res7: Array[(Int, Int)] = Array((4,3), (2,0), (1,-1), (3,0))
>
```

在上述代码中，(1,1), (2,1), (2,1), (3,1), (3,1), (3,1) 被划分到第一个partition，(3,1), (4,1), (4,1), (4,1), (4,1) 被划分到第二个。于是有如下操作：

(1, 1): 由于只有1个，所以在值取负的情况下，自然输出(1, -1)

(2, 1): 由于有2个，第一个取负，第二个不变，因此combine后为(2, 0)

(3, 1): partition1中有3个，参照上述规则，combine后为(3, 1)，partition2中有1个，因此combine后为(3, -1)。在第二次combine时，不会有初始化操作，因此直接相加，结果为(3, 0)

(4, 1): 过程同上，结果为(4, 3)

2.2.reduceByKey算子

功能：reduceByKey就是对元素为KV对的RDD中Key相同的元素的Value进行reduce，因此，Key相同的多个元素的值被reduce为一个值，然后与原RDD中的Key组成一个新的KV对。

源码：

```
>
/**
 * Merge the values for each key using an associative and commutative reduce function. This
 * will
 * also perform the merging locally on each mapper before sending results to a reducer, similarly
 * to a "combiner" in MapReduce. Output will be hash-partitioned with the existing partitioner
 * parallelism level.
 */
def reduceByKey(func: (V, V) => V): RDD[(K, V)] = self.withScope {
  reduceByKey(defaultPartitioner(self), func)
}
```

示例:

```
>
scala> val rdd1 = sc.parallelize(List(("A", 1), ("B", 3), ("C", 2)))
rdd1: org.apache.spark.rdd.RDD[(String, Int)] = ParallelCollectionRDD[5] at parallelize at <console>:24
>
scala> val rdd2 = sc.parallelize(List(("B", 2), ("D", 1), ("E", 2)))
rdd2: org.apache.spark.rdd.RDD[(String, Int)] = ParallelCollectionRDD[6] at parallelize at <console>:24
>
scala> val rdd3 = rdd1 union rdd2
rdd3: org.apache.spark.rdd.RDD[(String, Int)] = UnionRDD[7] at union at <console>:27
>
scala> val rdd4 = rdd3.reduceByKey(_ + _)
rdd4: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[8] at reduceByKey at <console>:5
>
scala> rdd4.collect
res2: Array[(String, Int)] = Array((A,1), (B,5), (C,2), (D,1), (E,2))
```

2.3.partitionBy算子

功能: 该函数根据partitioner函数生成新的ShuffleRDD, 将原RDD重新分区。

源码:

```
>
/** 
 * Return a copy of the RDD partitioned using the specified partitioner.
 */
def partitionBy(partitioner: Partitioner): RDD[(K, V)] = self.withScope {
  if (keyClass.isArray && partitioner.isInstanceOf[HashPartitioner]) {
    throw new SparkException("HashPartitioner cannot partition array keys.")
  }
  if (self.partitioner == Some(partitioner)) {
    self
  } else {
    new ShuffledRDD[K, V, V](self, partitioner)
  }
}
```

示例:

```
>
scala> val rdd1 = sc.parallelize(List(("A", 1), ("B", 2), ("C", 3), ("D", 4)), 2)
rdd1: org.apache.spark.rdd.RDD[(String, Int)] = ParallelCollectionRDD[2] at parallelize at <console>:24
>
scala> val rdd2 = rdd1.glom()
rdd2: org.apache.spark.rdd.RDD[Array[(String, Int)]] = MapPartitionsRDD[3] at glom at <console>:25
>
scala> rdd2.collect
res1: Array[Array[(String, Int)]] = Array(Array((A,1), (B,2)), Array((C,3), (D,4)))
>
scala> val rdd3 = rdd1.partitionBy(new org.apache.spark.HashPartitioner(2))
```

```
rdd3: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[4] at partitionBy at <console>:25
>
scala> rdd3.partitions.size
res2: Int = 2
>
scala> val rdd4 = rdd3.glom
rdd4: org.apache.spark.rdd.RDD[Array[(String, Int)]] = MapPartitionsRDD[5] at glom at <console>:25
>
scala> rdd4.collect
res3: Array[Array[(String, Int)]] = Array(Array((B,2), (D,4)), Array((A,1), (C,3)))
```

2.4.groupByKey算子

功能：根据key值进行分组，groupByKey()方法的数据本身就是一种key-value类型的。

源码：

```
>
/** 
 * Group the values for each key in the RDD into a single sequence. Hash-partitions the
 * resulting RDD with the existing partitioner/parallelism level. The ordering of elements
 * within each group is not guaranteed, and may even differ each time the resulting RDD is *
valuated.
*
* @note This operation may be very expensive. If you are grouping in order to perform an
* aggregation (such as a sum or average) over each key, using `PairRDDFunctions.aggregate
yKey`
* or `PairRDDFunctions.reduceByKey` will provide much better performance.
*/
def groupByKey(): RDD[(K, Iterable[V])] = self.withScope {
  groupByKey(defaultPartitioner(self))
}
```

示例：

```
>
scala> val a = sc.makeRDD(Array(("A",1),("B",2),("C",1),("A",3)))
a: org.apache.spark.rdd.RDD[(String, Int)] = ParallelCollectionRDD[3] at makeRDD at <console>:24
>
scala> val b = a.groupByKey()
b: org.apache.spark.rdd.RDD[(String, Iterable[Int])] = ShuffledRDD[4] at groupByKey at <console>:25
>
scala> b.collect
res1: Array[(String, Iterable[Int])] = Array((A,CompactBuffer(1, 3)), (B,CompactBuffer(2)), (C,CompactBuffer(1)))
```

2.5.foldByKey算子

功能：该函数用于RDD[K,V]根据K将V做折叠、合并处理，其中的参数zeroValue表示先根据映射函数将zeroValue应用于V,进行初始化V,再将映射函数应用于初始化后的V。

源码：

```

>
/** 
 * Return a new RDD by applying a function to each partition of this RDD, while tracking the index * of the original partition. **`preservesPartitioning` indicates whether the input function reserves the partitioner, which
 * should be `false` unless this is a pair RDD and the input function doesn't modify the keys.
 */
def mapPartitionsWithIndex[U: ClassTag](
  f: (Int, Iterator[T]) => Iterator[U],
  preservesPartitioning: Boolean = false): RDD[U] = withScope {
  val cleanedF = sc.clean(f)
  new MapPartitionsRDD(
    this,
    (context: TaskContext, index: Int, iter: Iterator[T]) => cleanedF(index, iter),
    preservesPartitioning)
}

```

示例：

```

>
scala> val rdd1 = sc.parallelize(List(("A", 1), ("B", 2), ("C", 3), ("A", 4)), 2)
rdd1: org.apache.spark.rdd.RDD[(String, Int)] = ParallelCollectionRDD[13] at parallelize at <console>:24
>
scala> val rdd2 = rdd1.foldByKey(10)(_ + _)
rdd2: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[14] at foldByKey at <console>:25
>
scala> rdd2.collect
res8: Array[(String, Int)] = Array((B,12), (A,25), (C,13))
>
//将rdd1中每个key对应的V进行累加，注意zeroValue=10,需要先初始化V,映射函数为+操作，比如(A",1), ("A",4)，先将zeroValue应用于每个V,得到： ("A",1+10), ("A",4+10)，即： ("A",11), ("A",14)再将映射函数应用于初始化后的V，最后得到(A,11+14),即(A,25)

```

2.6.reduceByKeyLocally算子

功能：该函数将RDD[K,V]中每个K对应的V值根据映射函数来运算，运算结果映射到一个Map[K,V]中而不是RDD[K,V]。

源码：

```

>
/** 
 * Merge the values for each key using an associative and commutative reduce function, but return
 * the results immediately to the master as a Map. This will also perform the merging locally on
 * each mapper before sending results to a reducer, similarly to a "combiner" in MapReduce.
 */
def reduceByKeyLocally(func: (V, V) => V): Map[K, V] = self.withScope {
  val cleanedF = self.sparkContext.clean(func)
  >
  if (keyClass.isArray) {
    throw new SparkException("reduceByKeyLocally() does not support array keys")
  }
  >

```

```

val reducePartition = (iter: Iterator[(K, V)]) => {
  val map = new JHashMap[K, V]
  iter.foreach { pair =>
    val old = map.get(pair._1)
    map.put(pair._1, if (old == null) pair._2 else cleanedF(old, pair._2))
  }
  Iterator(map)
} : Iterator[JHashMap[K, V]]
>
val mergeMaps = (m1: JHashMap[K, V], m2: JHashMap[K, V]) => {
  m2.asScala.foreach { pair =>
    val old = m1.get(pair._1)
    m1.put(pair._1, if (old == null) pair._2 else cleanedF(old, pair._2))
  }
  m1
} : JHashMap[K, V]
>
self.mapPartitions(reducePartition).reduce(mergeMaps).asScala
}

```

示例：

```

>
scala> val rdd1 = sc.parallelize(List(("A", 1), ("B", 2), ("C", 3), ("A", 4)))
rdd1: org.apache.spark.rdd.RDD[(String, Int)] = ParallelCollectionRDD[15] at parallelize at <co
sole>:24
>
scala> val rdd2 = rdd1.reduceByKeyLocally((x,y) => x * y)
rdd2: scala.collection.Map[String,Int] = Map(A -> 4, B -> 2, C -> 3)

```

2.7.cogroup算子

功能：该函数用于将多个RDD中的同一个key对应的不同的value组合到一起。返回一个结果RDD，含了一个元组，元组里面的每一个key，对应多个RDD中匹配的value。

源码：

```

>
/** 
 * For each key k in `this` or `other1` or `other2` or `other3`,
 * return a resulting RDD that contains a tuple with the list of values
 * for that key in `this`, `other1`, `other2` and `other3`.
 */
def cogroup[W1, W2, W3](other1: RDD[(K, W1)],
  other2: RDD[(K, W2)],
  other3: RDD[(K, W3)],
  partitioner: Partitioner)
  : RDD[(K, (Iterable[V], Iterable[W1], Iterable[W2], Iterable[W3]))] = self.withScope {
  if (partitioner.isInstanceOf[HashPartitioner] && keyClass.isArray) {
    throw new SparkException("HashPartitioner cannot partition array keys.")
  }
  val cg = new CoGroupedRDD[K](Seq(self, other1, other2, other3), partitioner)
  cg.mapValues { case Array(vs, w1s, w2s, w3s) =>
    (vs.asInstanceOf[Iterable[V]],
     w1s.asInstanceOf[Iterable[W1]],
     w2s.asInstanceOf[Iterable[W2]],
     w3s.asInstanceOf[Iterable[W3]])
  }
}

```

```
w3s.asInstanceOf[Iterable[W3]]  
}  
}
```

示例:

```
>  
scala> val rdd1 = sc.parallelize(List(("A", 1), ("B", 2), ("C", 3), ("D", 4)))  
rdd1: org.apache.spark.rdd.RDD[(String, Int)] = ParallelCollectionRDD[17] at parallelize at <co  
sole>:24  
>  
scala> val rdd2 = sc.parallelize(List(("A", 5), ("B", 6), ("E", 7), ("F", 8)))  
rdd2: org.apache.spark.rdd.RDD[(String, Int)] = ParallelCollectionRDD[18] at parallelize at <co  
sole>:24  
>  
scala> val rdd3 = rdd1.cogroup(rdd2)  
rdd3: org.apache.spark.rdd.RDD[(String, (Iterable[Int], Iterable[Int]))] = MapPartitionsRDD[20]  
t cogroup at <console>:27  
>  
scala> rdd3.collect  
res10: Array[(String, (Iterable[Int], Iterable[Int]))] = Array((D,(CompactBuffer(4),CompactBuffer(  
)), (A,(CompactBuffer(1),CompactBuffer(5))), (E,(CompactBuffer(),CompactBuffer(7))), (B,(Com  
pactBuffer(2),CompactBuffer(6))), (F,(CompactBuffer(),CompactBuffer(8))), (C,(CompactBuffer(3),  
ompactBuffer()))))
```

2.8.subtractByKey算子

功能: 类似于subtract, 删掉 RDD 中键与 other RDD 中的键相同的元素。

源码:

```
>  
/**  
 * Return an RDD with the pairs from `this` whose keys are not in `other`.  
 * * Uses `this` partitioner/partition size, because even if `other` is huge, the resulting  
 * RDD will be less than or equal to us.  
 */  
def subtractByKey[W: ClassTag](other: RDD[(K, W)]): RDD[(K, V)] = self.withScope {  
    subtractByKey(other, self.partitioner.getOrElse(new HashPartitioner(self.partitions.length)))  
}
```

示例:

```
>  
scala> val a = sc.makeRDD(Array(("A", "1"), ("B", "2"), ("C", "3")))  
a: org.apache.spark.rdd.RDD[(String, String)] = ParallelCollectionRDD[0] at makeRDD at <con  
sole>:24  
>  
scala> val b = sc.makeRDD(Array(("B", "4"), ("C", "5"), ("D", "6")))  
b: org.apache.spark.rdd.RDD[(String, String)] = ParallelCollectionRDD[1] at makeRDD at <con  
sole>:24  
>  
scala> val c = a.subtractByKey(b)  
c: org.apache.spark.rdd.RDD[(String, String)] = SubtractedRDD[2] at subtractByKey at <consol  
>:27  
>  
scala> c.collect
```

```
res0: Array[(String, String)] = Array((A,1))
```

###3.连接类型的算子

3.1.join算子

功能：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 c group 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 k y 下的所有元组形成一个集合。最后返回 RDD[(K, (V, W))].

源码：

```
>
/**
 * Return an RDD containing all pairs of elements with matching keys in `this` and `other`. Each
 * pair of elements will be returned as a (k, (v1, v2)) tuple, where (k, v1) is in `this` and
 * (k, v2) is in `other`. Uses the given Partitioner to partition the output RDD.
 */
def join[W](other: RDD[(K, W)], partitioner: Partitioner): RDD[(K, (V, W))] = self.withScope {
  this.cogroup(other, partitioner).flatMapValues( pair =>
    for (v <- pair._1.iterator; w <- pair._2.iterator) yield (v, w)
  )
}
```

示例：

```
>
scala> val rdd1 = sc.parallelize(List(("A", 1), ("B", 2), ("C", 3), ("D", 4)))
rdd1: org.apache.spark.rdd.RDD[(String, Int)] = ParallelCollectionRDD[21] at parallelize at <co
sole>:24
>
scala> val rdd2 = sc.parallelize(List(("A", 5), ("B", 6), ("E", 7), ("F", 8)))
rdd2: org.apache.spark.rdd.RDD[(String, Int)] = ParallelCollectionRDD[22] at parallelize at <co
sole>:24
>
scala> val rdd3 = rdd1.join(rdd2)
rdd3: org.apache.spark.rdd.RDD[(String, (Int, Int))] = MapPartitionsRDD[25] at join at <consol
>:27
>
scala> rdd3.collect
res12: Array[(String, (Int, Int))] = Array((A,(1,5)), (B,(2,6)))
```

3.2.leftOuterJoin算子

功能：leftOuterJoin类似于SQL中的左外关联left outer join，返回结果以前面的RDD为主，关联不的记录为空。只能用于两个RDD之间的关联。

源码：

```
>
/**
 * Perform a left outer join of `this` and `other`. For each element (k, v) in `this`, the
 * resulting RDD will either contain all pairs (k, (v, Some(w))) for w in `other`, or the
 * pair (k, (v, None)) if no elements in `other` have key k. Uses the given Partitioner to
 * partition the output RDD.
```

```

*/
def leftOuterJoin[W](
  other: RDD[(K, W)],
  partitioner: Partitioner): RDD[(K, (V, Option[W]))] = self.withScope {
  this.cogroup(other, partitioner).flatMapValues { pair =>
    if (pair._2.isEmpty) {
      pair._1.iterator.map(v => (v, None))
    } else {
      for (v <- pair._1.iterator; w <- pair._2.iterator) yield (v, Some(w))
    }
  }
}

```

示例：

```

>
scala> val rdd1 = sc.parallelize(List(("A", 1), ("B", 2), ("C", 3), ("D", 4)))
rdd1: org.apache.spark.rdd.RDD[(String, Int)] = ParallelCollectionRDD[26] at parallelize at <console>:24
>
scala> val rdd2 = sc.parallelize(List(("A", 5), ("B", 6), ("E", 7), ("F", 8)))
rdd2: org.apache.spark.rdd.RDD[(String, Int)] = ParallelCollectionRDD[27] at parallelize at <console>:24
>
scala> val rdd3 = rdd1.leftOuterJoin(rdd2)
rdd3: org.apache.spark.rdd.RDD[(String, (Int, Option[Int]))] = MapPartitionsRDD[30] at leftOuterJoin at <console>:27
>
scala> rdd3.collect
res13: Array[(String, (Int, Option[Int]))] = Array((D,(4,None)), (A,(1,Some(5))), (B,(2,Some(6))), (C,(3,None)))

```

3.3.rightOuterJoin算子

功能：rightOuterJoin类似于SQL中的有外关联right outer join，返回结果以参数中的RDD为主，关不上的记录为空。只能用于两个RDD之间的关联。

源码：

```

>
/** 
 * Perform a right outer join of `this` and `other`. For each element (k, w) in `other`, the
 * resulting RDD will either contain all pairs (k, (Some(v), w)) for v in `this`, or the
 * pair (k, (None, w)) if no elements in `this` have key k. Uses the given Partitioner to
 * partition the output RDD.
 */
def rightOuterJoin[W](other: RDD[(K, W)], partitioner: Partitioner)
  : RDD[(K, (Option[V], W))] = self.withScope {
  this.cogroup(other, partitioner).flatMapValues { pair =>
    if (pair._1.isEmpty) {
      pair._2.iterator.map(w => (None, w))
    } else {
      for (v <- pair._1.iterator; w <- pair._2.iterator) yield (Some(v), w)
    }
  }
}

```

示例：

```
>
scala> val rdd1 = sc.parallelize(List(("A", 1), ("B", 2), ("C", 3), ("D", 4)))
rdd1: org.apache.spark.rdd.RDD[(String, Int)] = ParallelCollectionRDD[31] at parallelize at <console>:24
>
scala> val rdd2 = sc.parallelize(List(("A", 5), ("B", 6), ("E", 7), ("F", 8)))
rdd2: org.apache.spark.rdd.RDD[(String, Int)] = ParallelCollectionRDD[32] at parallelize at <console>:24
>
scala> val rdd3 = rdd1.rightOuterJoin(rdd2)
rdd3: org.apache.spark.rdd.RDD[(String, (Option[Int], Int))] = MapPartitionsRDD[35] at rightOuterJoin at <console>:27
>
scala> rdd3.collect
res14: Array[(String, (Option[Int], Int))] = Array((A,Some(1),5), (E,None,7), (B,Some(2),6), (F,None,8))
```