链滴

spark 算子详解 ------Action 算子介绍

作者：18582596683

原文链接：https://ld246.com/article/1545028745879

来源网站：链滴

一、无输出的算子

1.foreach算子

功能：对 RDD 中的每个元素都应用 f 函数操作，无返回值。

源码：

>

```
/**
 * Applies a function f to all elements of this RDD.
 */
def foreach(f: T => Unit): Unit = withScope {
  val cleanF = sc.clean(f)
  sc.runJob(this, (iter: Iterator[T]) => iter.foreach(cleanF))
}
```

示例：

>

```
scala> val rdd1 = sc.parallelize(1 to 9)
rdd1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[20] at parallelize at <console>:2
```

>

```
scala>  rdd1.foreach(x => printf("%d ", x))
1 2 3 4 5 6 7 8 9
```

2.foreachPartition算子

功能：该函数和foreach类似，不同的是,foreach是直接在每个partition中直接对iterator执行foreac操作,传入的function只是在foreach内部使用,
而foreachPartition是在每个partition中把iterator给传入的function,让function自己对iterator进行理（可以避免内存溢出）。

>

简单来说，foreach的iterator是针对的rdd中的元素，而foreachPartition的iterator是针对的分区本。

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 rdd1 = sc.parallelize(1 to 9, 2)
rdd1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[23] at parallelize at <console>:2

>
scala> rdd1.foreachPartition(x => printf("%s ", x.size))
4 5
```

二、输出到HDFS等文件系统的算子

1.saveAsTextFile算子

功能：该函数将数据输出，以文本文件的形式写入本地文件系统或者HDFS等。Spark将对每个元素调toString方法，将数据元素转换为文本文件中的一行记录。若将文件保存到本地文件系统，那么只会存在executor所在机器的本地目录。

源码：

```
>
/**
 * Save this RDD as a text file, using string representations of elements.
 */
def saveAsTextFile(path: String): Unit = withScope {
  // https://issues.apache.org/jira/browse/SPARK-2075
 //
 // NullWritable is a `Comparable` in Hadoop 1.+, so the compiler cannot find an implicit
 // Ordering for it and will use the default `null`. However, it's a `Comparable[NullWritable]`
 // in Hadoop 2.+, so the compiler will call the implicit `Ordering.ordered` method to create an

 // Ordering for `NullWritable`. That's why the compiler will generate different anonymous
 // classes for `saveAsTextFile` in Hadoop 1.+ and Hadoop 2.+.
 //
 // Therefore, here we provide an explicit Ordering `null` to make sure the compiler generate
 // same bytecodes for `saveAsTextFile`.  val nullWritableClassTag = implicitly[ClassTag[NullWr
table]]
  val textClassTag = implicitly[ClassTag[Text]]
  val r = this.mapPartitions { iter =>
    val text = new Text()
  iter.map { x =>
      text.set(x.toString)
 (NullWritable.get(), text)
   }
 }  RDD.rddToPairRDDFunctions(r)(nullWritableClassTag, textClassTag, null)
  .saveAsHadoopFile[TextOutputFormat[NullWritable, Text]](path)
}
```

示例：

```
>
scala> val rdd1 = sc.parallelize(1 to 9, 2)
rdd1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[26] at parallelize at <console>:2

>
scala> rdd1.saveAsTextFile("file:///opt/app/test/saveAsTextFileTest.txt")
```

2.saveAsObjectFile算子

功能：该函数用于将RDD以ObjectFile形式写入本地文件系统或者HDFS等。

源码：

```
>
/**
 * Save this RDD as a SequenceFile of serialized objects.
 */
def saveAsObjectFile(path: String): Unit = withScope {
  this.mapPartitions(iter => iter.grouped(10).map(_.toArray))
  .map(x => (NullWritable.get(), new BytesWritable(Utils.serialize(x))))
  .saveAsSequenceFile(path)
}
```

示例：

```
>
scala> val rdd1 = sc.parallelize(Array(("a", 1), ("b", 2), ("c", 3), ("d", 5), ("a", 4)), 2)
rdd1: org.apache.spark.rdd.RDD[(String, Int)] = ParallelCollectionRDD[40] at parallelize at <co
sole>:24
>
scala> rdd1.saveAsObjectFile("file:///opt/app/test/saveAsObejctFileTest.txt")
```

3.saveAsHadoopFile算子

功能：该函数将RDD存储在HDFS上的文件中,可以指定outputKeyClass、outputValueClass以及压格式,每个分区输出一个文件。

源码：

```
>
/**
 * Output the RDD to any Hadoop-supported file system, using a Hadoop `OutputFormat` cla
s
 * supporting the key and value types K and V in this RDD.
 *
 * @note We should make sure our tasks are idempotent when speculation is enabled, i.e. do
 * not use output committer that writes data directly.
 * There is an example in https://issues.apache.org/jira/browse/SPARK-10063 to show the bad

 * result of using direct output committer with speculation enabled. */def saveAsHadoopFile(
  path: String,
  keyClass: Class[_],
  valueClass: Class[_],
  outputFormatClass: Class[_ <: OutputFormat[_, _]],
  conf: JobConf = new JobConf(self.context.hadoopConfiguration),
  codec: Option[Class[_ <: CompressionCodec]] = None): Unit = self.withScope {
  // Rename this as hadoopConf internally to avoid shadowing (see SPARK-2038).
  val hadoopConf = conf
  hadoopConf.setOutputKeyClass(keyClass)
  hadoopConf.setOutputValueClass(valueClass)
  conf.setOutputFormat(outputFormatClass)
  for (c <- codec) {
  hadoopConf.setCompressMapOutput(true)
  hadoopConf.set("mapreduce.output.fileoutputformat.compress", "true")
  hadoopConf.setMapOutputCompressorClass(c)
  hadoopConf.set("mapreduce.output.fileoutputformat.compress.codec", c.getCanonicalName

  hadoopConf.set("mapreduce.output.fileoutputformat.compress.type",
  CompressionType.BLOCK.toString)
  }
>
  // Use configured output committer if already set
  if (conf.getOutputCommitter == null) {
  hadoopConf.setOutputCommitter(classOf[FileOutputCommitter])
  }
>
  // When speculation is on and output committer class name contains "Direct", we should wa
n
 // users that they may loss data if they are using a direct output committer.  val speculationE
abled = self.conf.getBoolean("spark.speculation", false)
  val outputCommitterClass = hadoopConf.get("mapred.output.committer.class", "")
  if (speculationEnabled && outputCommitterClass.contains("Direct")) {
  val warningMessage =
  s"$outputCommitterClass may be an output committer that writes data directly to " +
       "the final location. Because speculation is enabled, this output committer may " +
       "cause data loss (see the case in SPARK-10063). If possible, please use an output " +
       "committer that does not have this behavior (e.g. FileOutputCommitter)."
  logWarning(warningMessage)
  }
>
  FileOutputFormat.setOutputPath(hadoopConf,
  SparkHadoopWriterUtils.createPathFromString(path, hadoopConf))
  saveAsHadoopDataset(hadoopConf)
}
```

示例:

```
>
val rdd1 = sc.parallelize(Array(("a", 1), ("b", 2), ("c", 3), ("d", 5), ("a", 4)), 2)
rdd1.saveAsHadoopFile("hdfs://192.168.199.201:8020/test",classOf[ClassTag[Text]],classOf[In
Writable],classOf[TextOutputFormat[Text,IntWritable]])
```

4.saveAsSequenceFile算子

功能：该函数用于将RDD以Hadoop SequenceFile的形式写入本地文件系统或者HDFS等。

源码:

```
>
/**
 * Output the RDD as a Hadoop SequenceFile using the Writable types we infer from the RDD
s key
 * and value types. If the key or value are Writable, then we use their classes directly;
 * otherwise we map primitive types such as Int and Double to IntWritable, DoubleWritable, e
c,
 * byte arrays to BytesWritable, and Strings to Text. The `path` can be on any Hadoop-support
d
 * file system.
 */
```

```
def saveAsSequenceFile(
  path: String,
  codec: Option[Class[_ <: CompressionCodec]] = None): Unit = self.withScope {
  def anyToWritable[U <% Writable](u: U): Writable = u
>
  // TODO We cannot force the return type of `anyToWritable` be same as keyWritableClass a
d
  // valueWritableClass at the compile time. To implement that, we need to add type paramet
rs to
 // SequenceFileRDDFunctions. however, SequenceFileRDDFunctions is a public class so it will
be a
 // breaking change.  val convertKey = self.keyClass != _keyWritableClass
  val convertValue = self.valueClass != _valueWritableClass
>
  logInfo("Saving as sequence file of type " +
    s"(${_keyWritableClass.getSimpleName},${_valueWritableClass.getSimpleName})" )
  val format = classOf[SequenceFileOutputFormat[Writable, Writable]]
  val jobConf = new JobConf(self.context.hadoopConfiguration)
  if (!convertKey && !convertValue) {
  self.saveAsHadoopFile(path, _keyWritableClass, _valueWritableClass, format, jobConf, codec)
  } else if (!convertKey && convertValue) {
  self.map(x => (x._1, anyToWritable(x._2))).saveAsHadoopFile(
  path, _keyWritableClass, _valueWritableClass, format, jobConf, codec)
  } else if (convertKey && !convertValue) {
  self.map(x => (anyToWritable(x._1), x._2)).saveAsHadoopFile(
  path, _keyWritableClass, _valueWritableClass, format, jobConf, codec)
  } else if (convertKey && convertValue) {
  self.map(x => (anyToWritable(x._1), anyToWritable(x._2))).saveAsHadoopFile(
  path, _keyWritableClass, _valueWritableClass, format, jobConf, codec)
  }
}
```

示例：

```
>
scala> val rdd1 = sc.parallelize(Array(("a", 1), ("b", 2), ("c", 3), ("d", 5), ("a", 4)), 2)
rdd1: org.apache.spark.rdd.RDD[(String, Int)] = ParallelCollectionRDD[38] at parallelize at <co
sole>:24
>
scala> rdd1.saveAsSequenceFile("file:///opt/app/test/saveAsSequenceFileTest1.txt")
```

5.saveAsHadoopDataset算子

功能：该函数使用旧的Hadoop API将RDD输出到任何Hadoop支持的存储系统，例如Hbase,为该存系统使用Hadoop JobConf 对象。

源码：

```
>
/**
 * Output the RDD to any Hadoop-supported storage system, using a Hadoop JobConf object
for
 * that storage system. The JobConf should set an OutputFormat and any output paths requir
d
 * (e.g. a table name to write to) in the same way as it would be configured for a Hadoop
 * MapReduce job.
 */
def saveAsHadoopDataset(conf: JobConf): Unit = self.withScope {
  val config = new HadoopMapRedWriteConfigUtil[K, V](new SerializableJobConf(conf))
  SparkHadoopWriter.write(
  rdd = self,
  config = config)
}
```

示例：
>

```
val rdd1 = sc.parallelize(Array(("a", 1), ("b", 2), ("c", 3), ("d", 5), ("a", 4)), 2)
var jobConf = new JobConf()
jobConf.setOutputFormat(classOf[TextOutputFormat[Text,IntWritable]])
jobConf.setOutputKeyClass(classOf[Text])
jobConf.setOutputValueClass(classOf[IntWritable])
jobConf.set("mapred.output.dir","/test/")
rdd1.saveAsHadoopDataset(jobConf)
```

6.saveAsNewAPIHadoopFile算子

功能：该函数用于将RDD数据保存到HDFS上，使用新版本Hadoop API。用法基本同saveAsHadoopile。

源码：
>

```
/**
 * Output the RDD to any Hadoop-supported file system, using a new Hadoop API `OutputFomat`
  * (mapreduce.OutputFormat) object supporting the key and value types K and V in this RDD.
 */
def saveAsNewAPIHadoopFile(
  path: String,
  keyClass: Class[_],
  valueClass: Class[_],
  outputFormatClass: Class[_ <: NewOutputFormat[_, _]],
  conf: Configuration = self.context.hadoopConfiguration): Unit = self.withScope {
  // Rename this as hadoopConf internally to avoid shadowing (see SPARK-2038).
  val hadoopConf = conf
  val job = NewAPIHadoopJob.getInstance(hadoopConf)
  job.setOutputKeyClass(keyClass)
  job.setOutputValueClass(valueClass)
  job.setOutputFormatClass(outputFormatClass)
  val jobConfiguration = job.getConfiguration
  jobConfiguration.set("mapreduce.output.fileoutputformat.outputdir", path)
  saveAsNewAPIHadoopDataset(jobConfiguration)
}
```

示例：
>

```
val rdd1 = sc.parallelize(Array(("a", 1), ("b", 2), ("c", 3), ("d", 5), ("a", 4)), 2)
rdd1.saveAsNewAPIHadoopFile("hdfs://192.168.199.201:8020/test",classOf[Text],classOf[IntWrtable],classOf[output.TextOutputFormat[Text,IntWritable]])
```

7.saveAsNewAPIHadoopDataset算子

原文链接：spark 算子详解 ------Action 算子介绍

功能：使用新的Hadoop API将RDD输出到任何Hadoop支持的存储系统，例如Hbase,为该存储系统用Hadoop Configuration对象。Conf设置一个OutputFormat和任何需要的输出路径(如要写入的表)，就像为Hadoop MapReduce作业配置的那样。

源码：

>

```
/**
 * Output the RDD to any Hadoop-supported storage system with new Hadoop API, using a
adoop
 * Configuration object for that storage system. The Conf should set an OutputFormat and an

 * output paths required (e.g. a table name to write to) in the same way as it would be
 * configured for a Hadoop MapReduce job.
 *
 * @note We should make sure our tasks are idempotent when speculation is enabled, i.e. do
 * not use output committer that writes data directly.
 * There is an example in https://issues.apache.org/jira/browse/SPARK-10063 to show the bad

 * result of using direct output committer with speculation enabled.
 */
def saveAsNewAPIHadoopDataset(conf: Configuration): Unit = self.withScope {
  val config = new HadoopMapReduceWriteConfigUtil[K, V](new SerializableConfiguration(co
f))
  SparkHadoopWriter.write(
  rdd = self,
  config = config)
}
```

示例：

>

```
val rdd1 = sc.parallelize(Array(("a", 1), ("b", 2), ("c", 3), ("d", 5), ("a", 4)), 2)
  var jobConf = new JobConf()
  jobConf.setOutputFormat(classOf[TextOutputFormat[Text,IntWritable]])
  jobConf.setOutputKeyClass(classOf[Text])
  jobConf.setOutputValueClass(classOf[IntWritable])
  jobConf.set("mapred.output.dir","/test/")
  rdd1.saveAsNewAPIHadoopDataset(jobConf)
```

三、输出scala集合和数据类型的算子

1.first算子

功能：返回RDD中的第一个元素，不排序。

源码：

>

```
/**
 * Return the first element in this RDD.
 */
def first(): T = withScope {
  take(1) match {
  case Array(t) => t
    case _ => throw new UnsupportedOperationException("empty collection")
  }
}
```

示例：

```
>
scala> val rdd1 = sc.parallelize(1 to 9)
rdd1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[0] at parallelize at <console>:24
>
scala> val rdd2 = rdd1.first()
rdd2: Int = 1
>
scala> print(rdd2)
1
```

2.count算子

功能：返回RDD中的元素数量。

源码：

```
>
/**
 * Return the number of elements in the RDD.
 */
def count(): Long = sc.runJob(this, Utils.getIteratorSize _).sum
```

示例：

```
>
scala> val rdd1 = sc.parallelize(1 to 9)
rdd1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[1] at parallelize at <console>:24
>
scala> println(rdd1.count())
9
```

3.reduce算子

功能：将RDD中元素两两传递给输入函数，同时产生一个新值，新值与RDD中下一个元素再被传递给入函数，直到最后只有一个值为止。

源码：

```
>
/**
 * Reduces the elements of this RDD using the specified commutative and
 * associative binary operator.
 */
def reduce(f: (T, T) => T): T = withScope {
  val cleanF = sc.clean(f)
  val reducePartition: Iterator[T] => Option[T] = iter => {
  if (iter.hasNext) {
  Some(iter.reduceLeft(cleanF))
  } else {
  None
  }
 }  var jobResult: Option[T] = None
```

```
  val mergeResult = (index: Int, taskResult: Option[T]) => {
  if (taskResult.isDefined) {
  jobResult = jobResult match {
  case Some(value) => Some(f(value, taskResult.get))
  case None => taskResult
      }
 } }  sc.runJob(this, reducePartition, mergeResult)
  // Get the final result out of our Option, or throw an exception if the RDD was empty
  jobResult.getOrElse(throw new UnsupportedOperationException("empty collection"))
}
```

示例:

```
>
scala> val rdd1 = sc.parallelize(1 to 9)
rdd1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[2] at parallelize at <console>:24
>
scala> val rdd2 = rdd1.reduce((x,y) => x + y)
rdd2: Int = 45
```

4.collect算子

功能：将一个RDD以一个Array数组形式返回其中的所有元素。

源码:

```
>
/**
 * Return an array that contains all of the elements in this RDD.
 *
 * @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 collect(): Array[T] = withScope {
  val results = sc.runJob(this, (iter: Iterator[T]) => iter.toArray)
  Array.concat(results: _*)
}
```

示例:

```
>
scala> val rdd1 = sc.parallelize(1 to 9)
rdd1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[3] at parallelize at <console>:24
>
scala> rdd1.collect
res3: Array[Int] = Array(1, 2, 3, 4, 5, 6, 7, 8, 9)
```

5.take算子

功能：返回一个包含数据集前n个元素的数组（从0下标到n-1下标的元素），不排序。

源码:

```
>
/**
 * Take the first num elements of the RDD. It works by first scanning one partition, and use the
 * results from that partition to estimate the number of additional partitions needed to satisfy
 * the limit.
 *
 * @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.
 *
 * @note Due to complications in the internal implementation, this method will raise
 * an exception if called on an RDD of `Nothing` or `Null`.
 */
def take(num: Int): Array[T] = withScope {
  val scaleUpFactor = Math.max(conf.getInt("spark.rdd.limit.scaleUpFactor", 4), 2)
  if (num == 0) {
  new Array[T](0)
  } else {
  val buf = new ArrayBuffer[T]
  val totalParts = this.partitions.length
    var partsScanned = 0
  while (buf.size < num && partsScanned < totalParts) {
  // The number of partitions to try in this iteration. It is ok for this number to be
 // greater than totalParts because we actually cap it at totalParts in runJob.  var numPartsToT
y = 1L
  val left = num - buf.size
      if (partsScanned > 0) {
  // If we didn't find any rows after the previous iteration, quadruple and retry.
 // Otherwise, interpolate the number of partitions we need to try, but overestimate // it by 5
%. We also cap the estimation in the end.  if (buf.isEmpty) {
  numPartsToTry = partsScanned * scaleUpFactor
       } else {
  // As left > 0, numPartsToTry is always >= 1
  numPartsToTry = Math.ceil(1.5 * left * partsScanned / buf.size).toInt
          numPartsToTry = Math.min(numPartsToTry, partsScanned * scaleUpFactor)
  }
 }
  val p = partsScanned.until(math.min(partsScanned + numPartsToTry, totalParts).toInt)
  val res = sc.runJob(this, (it: Iterator[T]) => it.take(left).toArray, p)
>
  res.foreach(buf ++= _.take(num - buf.size))
  partsScanned += p.size
    }
>
  buf.toArray
  }
}
```

示例:

```
>
scala> val rdd1 = sc.parallelize(1 to 9)
rdd1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[4] at parallelize at <console>:24
>
scala> val rdd2 = rdd1.take(3)
rdd2: Array[Int] = Array(1, 2, 3)
```

6.top算子

功能：从按降序排列的RDD中获取前N个元素，或者有可选的key函数决定顺序，返回一个数组。

源码：

```
>
/**
 * Returns the top k (largest) elements from this RDD as defined by the specified
 * implicit Ordering[T] and maintains the ordering. This does the opposite of
 * [[takeOrdered]]. For example:
 * {{{
  *   sc.parallelize(Seq(10, 4, 2, 12, 3)).top(1)
 *   // returns Array(12)
 *
 *   sc.parallelize(Seq(2, 3, 4, 5, 6)).top(2)
 *   // returns Array(6, 5)
 * }}}
 *
 * @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.
 *
 * @param num k, the number of top elements to return
 * @param ord the implicit ordering for T
 * @return an array of top elements
 */def top(num: Int)(implicit ord: Ordering[T]): Array[T] = withScope {
  takeOrdered(num)(ord.reverse)
}
```

示例：

```
>
scala> val rdd1 = sc.parallelize(1 to 9)
rdd1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[5] at parallelize at <console>:24
>
scala> val rdd2 = rdd1.top(3)
rdd2: Array[Int] = Array(9, 8, 7)
```

7.takeOrdered算子

功能：返回RDD中前n个元素，并按默认顺序排序（升序）或者按自定义比较器顺序排序。

源码：

```
>
/**
 * Returns the first k (smallest) elements from this RDD as defined by the specified
 * implicit Ordering[T] and maintains the ordering. This does the opposite of [[top]].
 * For example:
 * {{{
 *   sc.parallelize(Seq(10, 4, 2, 12, 3)).takeOrdered(1)
 *   // returns Array(2)
 *
 *   sc.parallelize(Seq(2, 3, 4, 5, 6)).takeOrdered(2)
 *   // returns Array(2, 3) * }}}
 *
 * @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.
 *
 * @param num k, the number of elements to return
 * @param ord the implicit ordering for T
 * @return an array of top elements
 */def takeOrdered(num: Int)(implicit ord: Ordering[T]): Array[T] = withScope {
  if (num == 0) {
  Array.empty
  } else {
  val mapRDDs = mapPartitions { items =>
      // Priority keeps the largest elements, so let's reverse the ordering.
  val queue = new BoundedPriorityQueue[T](num)(ord.reverse)
  queue ++= collectionUtils.takeOrdered(items, num)(ord)
  Iterator.single(queue)
  }
  if (mapRDDs.partitions.length == 0) {
  Array.empty
    } else {
  mapRDDs.reduce { (queue1, queue2) =>
        queue1 ++= queue2
        queue1
      }.toArray.sorted(ord)
  }
 }}
```

示例:

```
>
scala> val rdd1 = sc.makeRDD(Seq(5,4,2,1,3,6))
rdd1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[7] at makeRDD at <console>:24
>
scala> val rdd2 = rdd1.takeOrdered(3)
rdd2: Array[Int] = Array(1, 2, 3)
```

8.aggregate算子

功能：aggregate函数将每个分区里面的元素进行聚合（seqOp），然后用combine函数将每个分区结果和初始值(zeroValue)进行combine操作。这个函数最终返回的类型不需要和RDD中元素类型一
。

源码:

```
>
/**
 * Aggregate the elements of each partition, and then the results for all the partitions, using
 * given combine functions and a neutral "zero value". This function can return a different resu
t
 * type, U, than the type of this RDD, T. Thus, we need one operation for merging a T into an

 * and one operation for merging two U's, as in scala.TraversableOnce. Both of these functions
are
 * allowed to modify and return their first argument instead of creating a new U to avoid me
ory
 * allocation.
 *
 * @param zeroValue the initial value for the accumulated result of each partition for the
 * `seqOp` operator, and also the initial value for the combine results from
 *                   different partitions for the `combOp` operator - this will typically be the
 *                 neutral element (e.g. `Nil` for list concatenation or `0` for summation)
 * @param seqOp an operator used to accumulate results within a partition
 * @param combOp an associative operator used to combine results from different partitions
 */def aggregate[U: ClassTag](zeroValue: U)(seqOp: (U, T) => U, combOp: (U, U) => U): U = wi
hScope {
  // Clone the zero value since we will also be serializing it as part of tasks
  var jobResult = Utils.clone(zeroValue, sc.env.serializer.newInstance())
  val cleanSeqOp = sc.clean(seqOp)
  val cleanCombOp = sc.clean(combOp)
  val aggregatePartition = (it: Iterator[T]) => it.aggregate(zeroValue)(cleanSeqOp, cleanComb
p)
  val mergeResult = (index: Int, taskResult: U) => jobResult = combOp(jobResult, taskResult)
  sc.runJob(this, aggregatePartition, mergeResult)
  jobResult
}
```

示例:

>

```
scala> val rdd1 = sc.parallelize(1 to 9, 3)
rdd1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[11] at parallelize at <console>:2

》
scala>  val rdd2 =  rdd1.aggregate((0,0))(
     |      (acc,number) => (acc._1 + number, acc._2 + 1),
     |      (par1,par2) => (par1._1 + par2._1, par1._2 + par2._2)
     |    )
rdd2: (Int, Int) = (45,9)
```

9.fold算子

功能：通过op函数聚合各分区中的元素及合并各分区的元素，op函数需要两个参数，在开始时第一传入的参数为zeroValue,T为RDD数据集的数据类型，，其作用相当于SeqOp和comOp函数都相同的ggregate函数。

源码:

>

```
/**
 * Aggregate the elements of each partition, and then the results for all the partitions, using a
 * given associative function and a neutral "zero value". The function
 * op(t1, t2) is allowed to modify t1 and return it as its result value to avoid object
 * allocation; however, it should not modify t2.
 *
 * This behaves somewhat differently from fold operations implemented for non-distributed
 * collections in functional languages like Scala. This fold operation may be applied to
 * partitions individually, and then fold those results into the final result, rather than
 * apply the fold to each element sequentially in some defined ordering. For functions
 * that are not commutative, the result may differ from that of a fold applied to a
 * non-distributed collection.
 *
 * @param zeroValue the initial value for the accumulated result of each partition for the `op`
 *                  operator, and also the initial value for the combine results from different
 *                  partitions for the `op` operator - this will typically be the neutral
 *                  element (e.g. `Nil` for list concatenation or `0` for summation)
 * @param op an operator used to both accumulate results within a partition and combine re
```

```
ults
 *                from different partitions */def fold(zeroValue: T)(op: (T, T) => T): T = withScope {
  // Clone the zero value since we will also be serializing it as part of tasks
  var jobResult = Utils.clone(zeroValue, sc.env.closureSerializer.newInstance())
  val cleanOp = sc.clean(op)
  val foldPartition = (iter: Iterator[T]) => iter.fold(zeroValue)(cleanOp)
  val mergeResult = (index: Int, taskResult: T) => jobResult = op(jobResult, taskResult)
  sc.runJob(this, foldPartition, mergeResult)
  jobResult
}
```

示例：

```
>
scala> val rdd1 = sc.parallelize(Array(("a", 1), ("b", 2), ("c", 3), ("d", 5), ("a", 4)), 2)
rdd1: org.apache.spark.rdd.RDD[(String, Int)] = ParallelCollectionRDD[13] at parallelize at <co
sole>:24
>
scala> val rdd2 = rdd1.fold(("e", 0))((val1, val2) => { if (val1._2 >= val2._2) val1 else val2})
rdd2: (String, Int) = (d,5)
>
scala> println(rdd2)
(d,5)
```

10.lookup算子

功能：该函数对（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

源码：

```
>
/**
 * Return the list of values in the RDD for key `key`. This operation is done efficiently if the
 * RDD has a known partitioner by only searching the partition that the key maps to.
 */
def lookup(key: K): Seq[V] = self.withScope {
  self.partitioner match {
  case Some(p) =>
     val index = p.getPartition(key)
  val process = (it: Iterator[(K, V)]) => {
  val buf = new ArrayBuffer[V]
  for (pair <- it if pair._1 == key) {
  buf += pair._2
       }
  buf
     } : Seq[V]
  val res = self.context.runJob(self, process, Array(index))
  res(0)
  case None =>
     self.filter(_._1 == key).map(_._2).collect()
  }
}
```

示例：

原文链接：spark 算子详解 ------Action 算子介绍

```
>
scala> val rdd1 = sc.parallelize(Array(("a", 1), ("b", 2), ("c", 3), ("d", 4), ("a", 5)), 2)
rdd1: org.apache.spark.rdd.RDD[(String, Int)] = ParallelCollectionRDD[14] at parallelize at <co
sole>:24
>
scala> val rdd2 = rdd1.lookup("a")
rdd2: Seq[Int] = WrappedArray(1, 5)
```

11.countByKey算子

功能：用于统计RDD[K,V]中每个K的数量，返回具有每个key的计数的（k，int）pairs的Map。

源码：

```
>
/**
 * Count the number of elements for each key, collecting the results to a local Map.
 *
 * @note This method should only be used if the resulting map is expected to be small, as
 * the whole thing is loaded into the driver's memory.
 * To handle very large results, consider using rdd.mapValues(_ => 1L).reduceByKey(_ + _), wh
ch * returns an RDD[T, Long] instead of a map.
 */
def countByKey(): Map[K, Long] = self.withScope {
  self.mapValues(_ => 1L).reduceByKey(_ + _).collect().toMap
}
```

示例：

```
>
scala> val rdd1 = sc.parallelize(Array(("a", 1), ("b", 2), ("c", 3), ("d", 4), ("a", 5)), 2)
rdd1: org.apache.spark.rdd.RDD[(String, Int)] = ParallelCollectionRDD[17] at parallelize at <co
sole>:24
>
scala> val rdd2 = rdd1.countByKey()
rdd2: scala.collection.Map[String,Long] = Map(d -> 1, b -> 1, a -> 2, c -> 1)
```

原文链接：spark 算子详解 ------Action 算子介绍