# Spark的算子详细介绍

Spark 算子大致可以分为以下两类:

* Transformation 变换/转换算子：这种变换并不触发提交作业，完成作业中间过程处理。

　　Transformation 操作是延迟计算的，也就是说从一个RDD 转换生成另一个 RDD 的转换操作不是马上执行，需要等到有 Action 操作的时候才会真正触发运算。

* Action 行动算子：这类算子会触发 SparkContext 提交 Job 作业。

　　 Action 算子会触发 Spark 提交作业（Job），并将数据输出 Spark系统。

## RDD构造

### 1、使用内存构造RDD

使用的方法：makeRDD和parallelize方法，如下代码所示：

|  |
| --- |
| ////////////////////////////\* 1、使用makeRDD创建RDD \*/  /\* List \*/  val rdd01 = sc.makeRDD(List(1,2,3,4,5,6))  val r01 = rdd01.map { x => x \* x }  println(r01.collect().mkString(","))  /\* Array \*/  val rdd02 = sc.makeRDD(Array(1,2,3,4,5,6))  val r02 = rdd02.filter { x => x < 5}  println(r02.collect().mkString(","))  /////////////////////////////\*2、使用parallelize创建RDD  /\*List\*/  val rdd03 = sc.parallelize(List(1,2,3,4,5,6), 1)  val r03 = rdd03.map { x => x + 1 }  println(r03.collect().mkString(","))  /\* Array \*/  val rdd04 = sc.parallelize(Array(1,2,3,4,5,6), 1)  val r04 = rdd04.filter { x => x > 3 }  println(r04.collect().mkString(",")) |

  大家看到了RDD本质就是一个数组，因此构造数据时候使用的是List（链表）和Array（数组）类型。

### **2、通过文件系统构造RDD**

代码如下所示：

|  |
| --- |
| val rdd:RDD[String] = sc.textFile("file:///D:/sparkdata.txt", 1)  val r:RDD[String] = rdd.flatMap { x => x.split(",") }  println(r.collect().mkString(",")) |

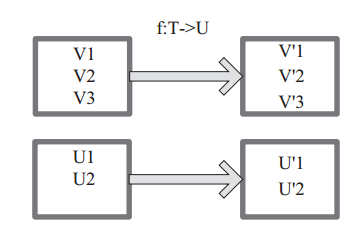
　　这里例子使用的是本地文件系统，所以文件路径协议前缀是file://。

## Transformations 算子

### map

将原来RDD的每个数据项通过map中的用户自定义函数f映射转变为一个新的元素。

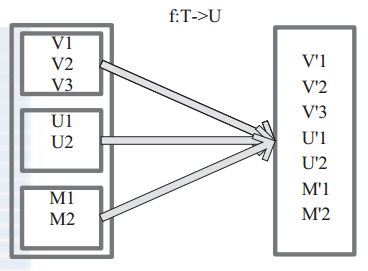
  下图中每个方框表示一个RDD分区，左侧的分区经过用户自定义函数f:T->U映射为右侧的新RDD分区。但是，实际只有等到 Action算子触发后，这个f函数才会和其他函数在一个stage中对数据进行运算。在下图中的第一个分区，数据记录V1输入f，通过 f 转换输出为转换后的分区中的数据记录V'1。



|  |
| --- |
| println("======map操作======")  println(rddInt.map(x => x + 1).collect().mkString(","))  println("======map操作======") |

### flatMap

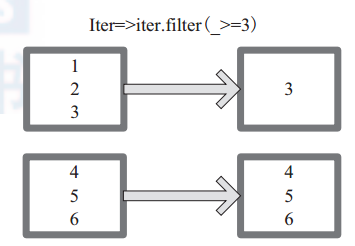
将原来RDD中的每个元素通过函数f转换为新的元素，并将生成的RDD的每个集合中的元素合并为一个集合，内部创建 FlatMappedRDD(this，sc.clean(f))。  
　　下图表示RDD的一个分区，进行flatMap函数操作，flatMap中 传入的函数为f:T->U，T和U可以是任意的数据类型。将分区中的数据通过用户自定义函数f转换为新的数据。外部大方框可以认为是一个RDD分区，小方框代表一个集合。V1、V2、V3在一个集合作为RDD 的一个数据项，可能存储为数组或其他容器，转换为V'1、V'2、V'3 后，将原来的数组或容器结合拆散，拆散的数据形成为RDD中的数据项。



|  |
| --- |
| println("======flatMap操作======")  println(rddFile.flatMap { x => x.split(",") }.first())  println("======flatMap操作======") |

1. **mapPartitions**

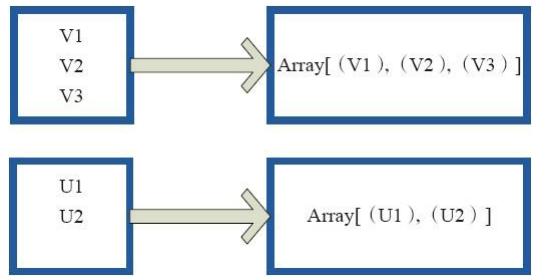
mapPartitions函数获取到每个分区的迭代器，在函数中通 过这个分区整体的迭代器对整个分区的元素进行操作。内部实现 是生成MapPartitionsRDD。下图中的方框代表一个RDD分区，用户通过函数f(iter)=>iter.filter(\_>=3)对分区中所有数据进行过滤，大于和等于3的数据保留。一个方块代表一个RDD分区，含有1、2、3的分区过滤只剩下元素3。



|  |
| --- |
| **val** rdd1 = sc.parallelize(*List*(1,2,8,3,3,4,5,6,7,7),2) rdd1.mapPartitions(x=>x.drop(1)).foreach(*println*) rdd1.mapPartitions(x=>x.dropWhile(x=>x>2)) |

### glom

glom函数将每个分区形成一个数组，内部实现是返回的GlommedRDD。 图4中的每个方框代表一个RDD分区。图4中的方框代表一个分区。 该图表示含有V1、 V2、 V3的分区通过函数glom形成一数组Array[（V1），（V2），（V3）]。



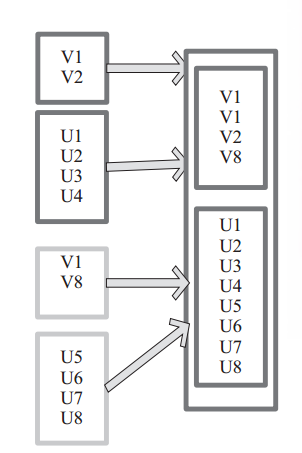
|  |
| --- |
| val rdd1 = sc.parallelize(List(1,2,8,3,3,4,5,6,7,7),2)  val rdd2 = rdd1.glom()  for(a<- rdd2){for(x<- a) {println(x) } } |

### union

使用union函数时需要保证两个RDD元素的数据类型相同，返回的RDD数据类型和被合并的RDD元素数据类型相同，并不进行去重操作，保存所有元素。如果想去重可以使用 distinct()。同时Spark还提供更为简洁的使用union的API，通过 ++ 符号相当于union函数操作。

图中左侧大方框代表两个RDD，大方框内的小方框代表RDD的分区。右侧大方框代表合并后的RDD，大方框内的小方框代表分区。

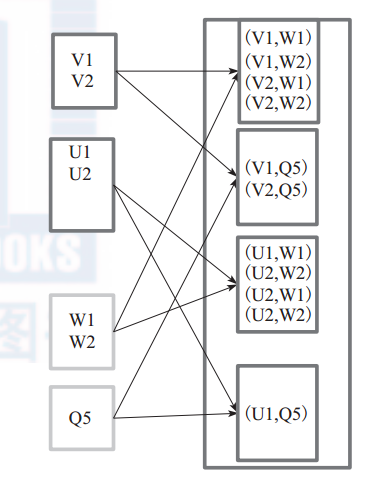
含有V1、V2、U1、U2、U3、U4的RDD和含有V1、V8、U5、U6、U7、U8的RDD合并所有元素形成一个RDD。V1、V1、V2、V8形成一个分区，U1、U2、U3、U4、U5、U6、U7、U8形成一个分区。



|  |
| --- |
| println("======union操作======")  println(rdd01.union(rdd02).collect().mkString(","))  println("======union操作======") |

### Cartesion

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CartesianRDD。图中左侧大方框代表两个RDD，大方框内的小方框代表RDD的分区。右侧大方框代表合并后的RDD，大方框内的小方框代表分区。图6中的大方框代表RDD，大方框中的小方框代表RDD分区。  
  例如：V1和另一个RDD中的W1、W2、Q5进行笛卡尔积运算形成 (V1,W1)、(V1,W2)、(V1,Q5)。



|  |
| --- |
| \* cartesian操作 \*/  println("======cartesian操作======")  println(rdd01.cartesian(rdd02).collect().mkString(","))  println("======cartesian操作======") |

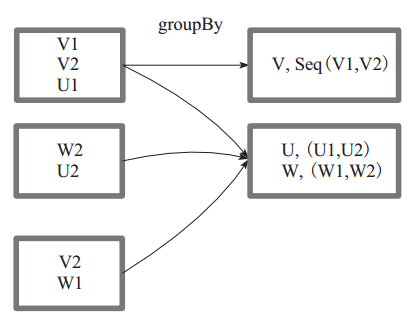
### groupBy

groupBy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相同的元素分为一组。  
函数实现如下：  
1）将用户函数预处理：  
　val cleanF = sc.clean(f)  
2）对数据map进行函数操作，最后再进行groupByKey分组操作。

 this.map(t => (cleanF(t), t)).groupByKey(p)  
　其中，p确定了分区个数和分区函数，也就决定了并行化的程度。

　图中方框代表一个RDD分区，相同key的元素合并到一个组。

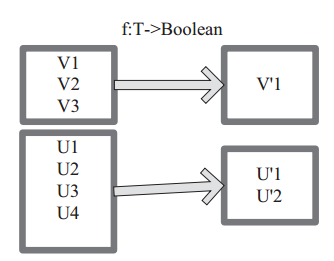
例如 V1和V2合并为V,Value为V1,V2。形成 V,Seq(V1,V2)。



|  |
| --- |
| **val** rdd = sc.parallelize(*List*(**"Hello"**,**"HADOOP"**,**"hbase"**,**"hive"**,**"flume"**)) rdd.groupBy(\_.length).foreach(x=>*println*(x)) |

### filter

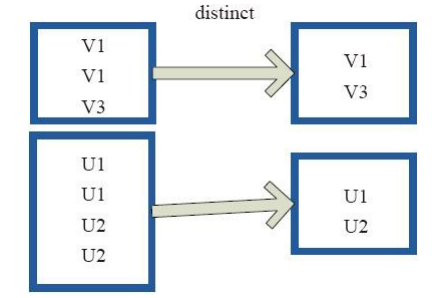
filter函数功能是对元素进行过滤，对每个元素应用f函数，返回值为true的元素在RDD中保留，返回值为false的元素将被过滤掉。内部实现相当于生成FilteredRDD(this，sc.clean(f))。  
下面代码为函数的本质实现：  
    deffilter(f:T=>Boolean):RDD[T]=newFilteredRDD(this,sc.clean(f))  
　　图中每个方框代表一个RDD分区，T可以是任意的类型。通过用户自定义的过滤函数f，对每个数据项操作，将满足条件、返回结果为true的数据项保留。例如，过滤掉V2和V3保留了V1，为区分命名为 V'1。



|  |
| --- |
| println(rddInt.filter(x => x > 4).collect().mkString(",")) |

### Distinct

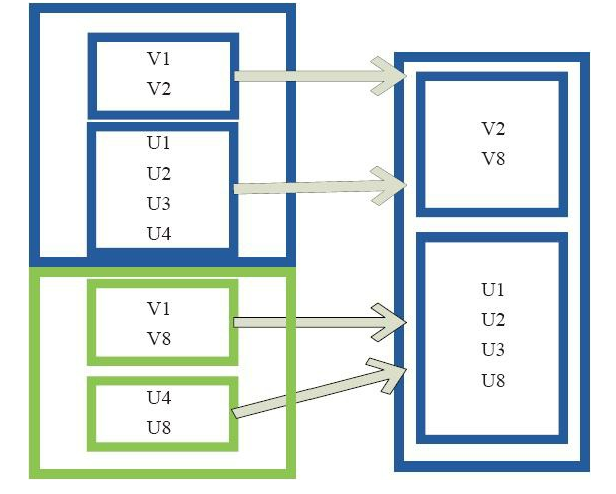
**d**istinct将RDD中的元素进行去重操作。图中的每个方框代表一个RDD分区，通过distinct函数，将数据去重。例如，重复数据V1、 V1去重后只保留一份V1。



|  |
| --- |
| println(rddInt.distinct().collect().mkString(",")) |

### subtract

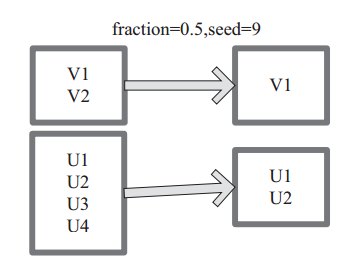
subtract相当于进行集合的差操作，RDD1去除RDD1和RDD 2交集中的所有元素。图中左侧的大方框代表两个RDD，大方框内的小方框代表RDD的分区。右侧大方框代表合并后的RDD，大方框内的小方框代表分区。 V1在两个RDD中均有，根据差集运算规则，新RDD不保留，V2在第一个RDD有，第二个RDD没有，则在新RDD元素中包含V2。



|  |
| --- |
| println(rdd01.subtract(rdd02).collect().mkString(",")) |

### Sample

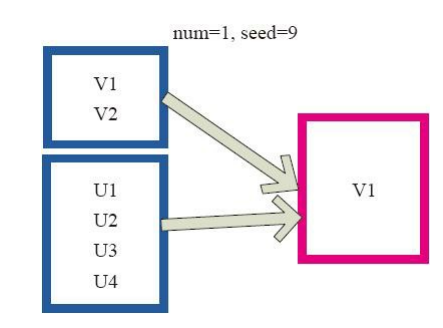
sample将 RDD这个集合内的元素进行采样，获取所有元素的子集。用户可以设定是否有放回的抽样、百分比、随机种子，进而决定采样方式。内部实现是生成SampledRDD(withReplacement， fraction,seed)。  
函数参数设置：  
 　　withReplacement=true，表示有放回的抽样。  
 　　withReplacement=false，表示无放回的抽样。  
　　图中的每个方框是一个RDD分区。通过sample函数，采样50%的数据。V1、V2、U1、U2、U3、U4采样出数据V1和U1、U2形成新的RDD。



|  |
| --- |
| **val** rdd = sc.makeRDD(*List*(1,24,7,6,3,1,2,1,9,8,12,23,34,55,61,19,18),2)rdd.sample(**true**,1,0).foreach(*println*) |

### takeSample

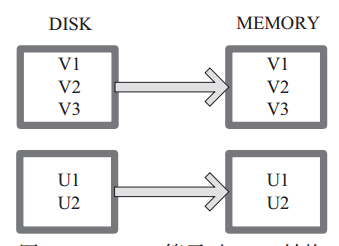
takeSample函数和上面的sample函数是一个原理，但是不使用相对比例采样，而是按设定的采样个数进行采样，同时返回结果不再是RDD，而是相当于对采样后的数据进行Collect（），返回结果的集合为单机的数组。  
　　图中左侧的方框代表分布式的各个节点上的分区，右侧方框代表单机上返回的结果数组。通过takeSample对数据采样，设置为采样一份数据，返回结果为V1。



|  |
| --- |
| **val** rdd = sc.makeRDD(*List*(1,24,7,6,3,1,2,1,9,8,12,23,34,55,61,19,18),2)rdd.takeSample(**true**,10,0).foreach(*println*) |

### Cache

cache将RDD元素从磁盘缓存到内存。相当persist(MEMORY\_ONLY) 函数的功能。  
   图中每个方框代表一个RDD分区，左侧相当于数据分区都存储在磁盘，通过cache算子将数据缓存在内存。



|  |
| --- |
| **val** rdd = sc.makeRDD(*List*(1,24,7,6,3,1,2,1,9,8,12,23,34,55,61,19,18),2) **val** rdd1 = rdd.map(\_\*3).cache() |

### Persist

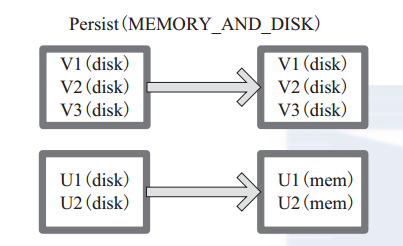
persist函数对RDD进行缓存操作。数据缓存在哪里依据 StorageLevel这个枚举类型进行确定。有以下几种类型的组合（见10），DISK代表磁盘，MEMORY代表内存，SER代表数据是否进行序列化存储。

下面为函数定义，StorageLevel是枚举类型，代表存储模式，用户可以通过图 14-1 按需进行选择。

persist(newLevel:StorageLevel)  
　　图中列出persist函数可以进行缓存的模式。例如，MEMORY\_AND\_DISK\_SER代表数据可以存储在内存和磁盘，并且以序列化的方式存储，其他同理。



图中方框代表RDD分区。disk代表存储在磁盘，mem代表存储在内存。数据最初全部存储在磁盘，通过persist(MEMORY\_AND\_DISK)将数据缓存到内存，但是有的分区无法容纳在内存，将含有V1、V2、 V3 的RDD存储到磁盘，将含有U1，U2的RDD仍旧存储在内存。

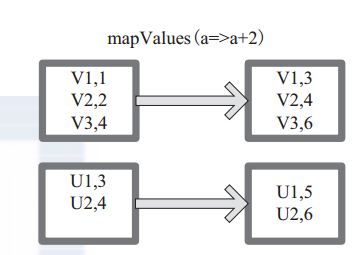


|  |
| --- |
| **val** rdd = sc.makeRDD(*List*(1,24,7,6,3,1,2,1,9,8,12,23,34,55,61,19,18),2) **val** rdd1 = rdd.map(\_\*3).persist(StorageLevel.*DISK\_ONLY\_2*) |

### mapValues

mapValues：针对（Key，Value）型数据中的Value进行Map操作，而不对Key进行处理。

图中的方框代表RDD分区。a=>a+2代表对(V1,1)这样的 Key Value数据对，数据只对Value中的1进行加2操作，返回结果为3。



|  |
| --- |
| **val** rdd = sc.parallelize(*List*(**"Hello"**,**"HADOOP"**,**"hbase"**,**"hive"**,**"flume"**)) *//val rdd = sc.makeRDD(List(1,24,7,6,3,1,2,1,9,8,12,23,34,55,61,19,18),2)* **val** rdd1 = rdd.map(x=>(x.length,x)).mapValues(\_+**"e"**).foreach(*println*) |

### combineByKey

我们先来看一下combineByKey函数的定义：

|  |
| --- |
| **def combineByKey[C](createCombiner: (V) => C,**  **mergeValue: (C, V) => C,**  **mergeCombiners: (C, C) => C): RD** |

****createCombiner****: 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。

****mergeValue:**** 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。

****mergeCombiners:**** 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。

如下解释下3个重要的函数参数：

createCombiner: V => C ，这个函数把当前的值作为参数，此时我们可以对其做些附加操作(类型转换)并把它返回 (这一步类似于初始化操作)

mergeValue: (C, V) => C，该函数把元素V合并到之前的元素C(createCombiner)上 (这个操作在每个分区内进行)

mergeCombiners: (C, C) => C，该函数把2个元素C合并 (这个操作在不同分区间进行)

**下面我们看一个求平均值的例子：**

|  |
| --- |
| val initialScores = Array(("Fred", 88.0), ("Fred", 95.0), ("Fred", 91.0), ("Wilma", 93.0), ("Wilma", 95.0), ("Wilma", 98.0))  val d1 = sc.parallelize(initialScores)  type MVType = (Int, Double) //定义一个元组类型(科目计数器,分数)  d1.combineByKey(  score => (1, score),  (c1: MVType, newScore) => (c1.\_1 + 1, c1.\_2 + newScore),  (c1: MVType, c2: MVType) => (c1.\_1 + c2.\_1, c1.\_2 + c2.\_2)  ).map { case (name, (num, socre)) => (name, socre / num) }.collect |

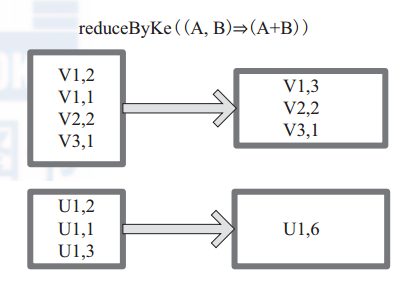
**参数含义的解释：**  
a 、score => (1, score)，我们把分数作为参数,并返回了附加的元组类型。 以"Fred"为列，当前其分数为88.0 =>(1,88.0)  1表示当前科目的计数器，此时只有一个科目。  
  
b、(c1: MVType, newScore) => (c1.\_1 + 1, c1.\_2 + newScore)，注意这里的c1就是createCombiner初始化得到的(1,88.0)。在一个分区内，我们又碰到了"Fred"的一个新的分数91.0。当然我们要把之前的科目分数和当前的分数加起来即c1.\_2 + newScore,然后把科目计算器加1即c1.\_1 + 1  
  
c、 (c1: MVType, c2: MVType) => (c1.\_1 + c2.\_1, c1.\_2 + c2.\_2)，注意"Fred"可能是个学霸,他选修的科目可能过多而分散在不同的分区中。所有的分区都进行mergeValue后,接下来就是对分区间进行合并了,分区间科目数和科目数相加分数和分数相加就得到了总分和总科目数。

|  |
| --- |
| res1: Array[(String, Double)] = Array((Wilma,95.33333333333333), (Fred,91.33333333333333)) |

### reduceByKey

reduceByKey 是比combineByKey更简单的一种情况，只是两个值合并成一个值，（ Int，Int V）to（Int，Int C），比如叠加。所以reduceBykey很简单，就是直接返回v。

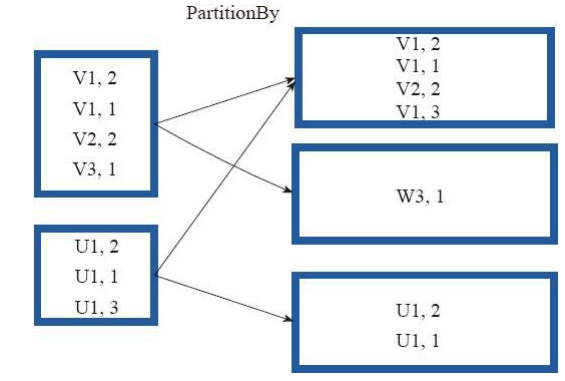
函数实现：  
  def reduceByKey(partitioner: Partitioner, func: (V, V) => V): RDD[(K, V)]  
= {combineByKey[V]((v: V) => v, func, func, partitioner)}  
　　图中的方框代表RDD分区。通过用户自定义函数(A,B) => (A + B) 函数，将相同key的数据(V1,2)和(V1,1)的value相加运算，结果为（ V1,3）。



|  |
| --- |
| **val** rdd = sc.parallelize(*List*(**"Hello"**,**"HADOOP"**,**"hbase"**,**"hive"**,**"flume"**)) *//val rdd = sc.makeRDD(List(1,24,7,6,3,1,2,1,9,8,12,23,34,55,61,19,18),2)* **val** rdd1 = rdd.map(x=>(x,x.length)).reduceByKey(\_+\_).foreach(*println*)  **val** rdd = sc.parallelize(*List*((**"Hello"**,(1,2)),(**"HADOOP"**,(1,4)),(**"hbase"**,(5,6)),(**"hive"**,(11,12)),(**"flume"**,(4,5)))) *//val rdd = sc.makeRDD(List(1,24,7,6,3,1,2,1,9,8,12,23,34,55,61,19,18),2)* **val** rdd1 = rdd.reduceByKey((x,y)=>(x.\_1+y.\_1,x.\_2+y.\_2)).foreach(*println*) |

### partitionBy

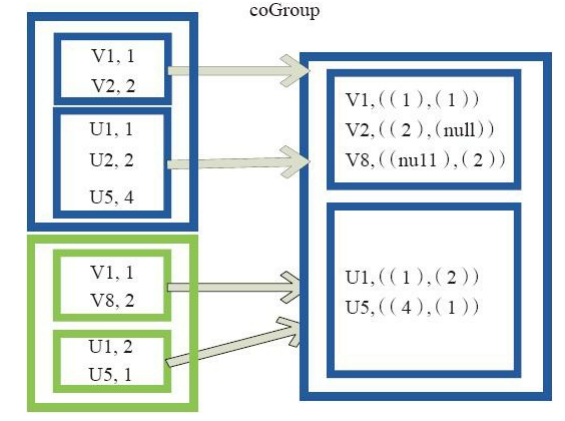
partitionBy函数对RDD进行分区操作。  
函数定义如下。  
partitionBy（partitioner：Partitioner）  
　　如果原有RDD的分区器和现有分区器（partitioner）一致，则不重分区，如果不一致，则相当于根据分区器生成一个新的ShuffledRDD。  
　　图中的方框代表RDD分区。通过新的分区策略将原来在不同分区的V1、V2数据都合并到了一个分区。



|  |
| --- |
| **var** rdd1 = sc.makeRDD(*Array*((1,**"A"**),(2,**"B"**),(3,**"C"**),(4,**"D"**)),2) *println*(rdd1.partitions.size) rdd1.foreachPartition(x=>*println*(x.toList)) **var** rdd2 = rdd1.partitionBy(**new** org.apache.spark.HashPartitioner(2)) *println*(rdd2.partitions.size) rdd2.foreachPartition(x=>*println*(x.toList)) |

### Cogroup

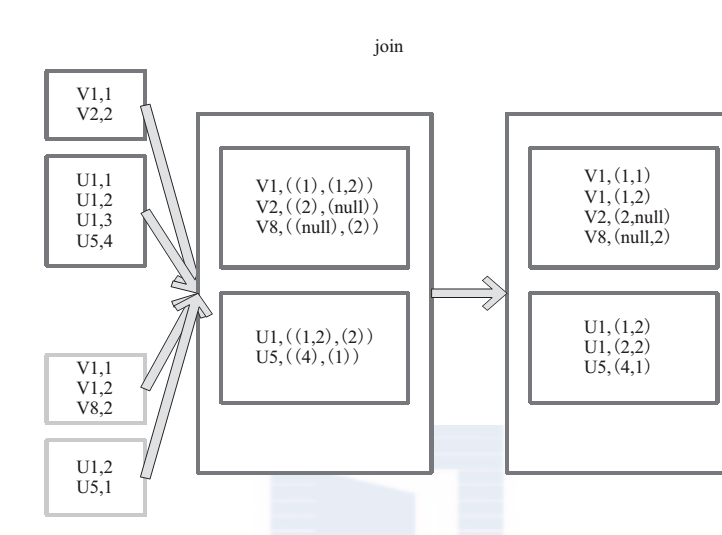
cogroup函数将两个RDD进行协同划分，cogroup函数的定义如下。  
　　cogroup[W]（other： RDD[（K， W）]， numPartitions： Int）： RDD[（K， （Iterable[V]， Iterable[W]））]  
　　对在两个RDD中的Key-Value类型的元素，每个RDD相同Key的元素分别聚合为一个集合，并且返回两个RDD中对应Key的元素集合的迭代器。  
　　（K， （Iterable[V]， Iterable[W]））  
　　其中，Key和Value，Value是两个RDD下相同Key的两个数据集合的迭代器所构成的元组。  
　　图中的大方框代表RDD，大方框内的小方框代表RDD中的分区。 将RDD1中的数据（U1，1）、（U1，2）和RDD2中的数据（U1，2）合并为（U1，（（1，2），（2）））。



|  |
| --- |
| **val** rdd1 = sc.parallelize(*List*((**"x01"**,1),(**"x02"**,2),(**"x01"**,3),(**"x03"**,5))) **val** rdd2 = sc.parallelize(*List*((**"x01"**,6),(**"x02"**,5),(**"x03"**,1),(**"x03"**,10))) **val** rdd3 = rdd1.cogroup(rdd2).foreach(*println*) |

### join

join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 cogroup 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key 下的所有元组形成一个集合。最后返回RDD[(K，(V，W))]。  
　　下面代码为join的函数实现，本质是通过cogroup 算子先进行 协同划分，再通过flatMapValues 将合并的数据打散。this.cogroup(other,partitioner).f latMapValues{case(vs,ws) => for(v<-vs;w<-ws)yield(v,w) }  
图是对两个RDD的oin 操作示意图。大方框代表 RDD，小方框代表 RDD 中的分区。函数对相同 key 的元素，如 V1 为 key 做连接后结果为 (V1,(1,1)) 和 (V1,(1,2))。



|  |
| --- |
| **val** rdd1 = sc.parallelize(*List*((**"x01"**,1),(**"x02"**,2),(**"x01"**,3),(**"x03"**,5)))  **val** rdd2 = sc.parallelize(*List*((**"x01"**,6),(**"x02"**,5),(**"x03"**,1),(**"x03"**,10))) rdd1.join(rdd2).foreach(*println*) |

### leftOutJoin和rightOutJoin

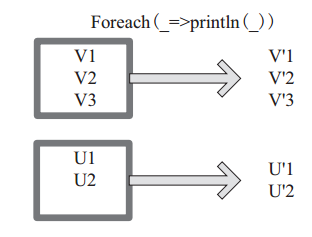
LeftOutJoin（左外连接）和RightOutJoin（右外连接）相当于在join的基础上先判断一侧的RDD元素是否为空，如果为空，则填充为空。 如果不为空，则将数据进行连接运算，并返回结果。

|  |
| --- |
| **val** rdd1 = sc.parallelize(*List*((**"x01"**,1),(**"x02"**,2),(**"x01"**,3),(**"x08"**,5)))  **val** rdd2 = sc.parallelize(*List*((**"x01"**,6),(**"x02"**,5),(**"x03"**,1),(**"x03"**,10))) rdd1.leftOuterJoin(rdd2).foreach(*println*)  rdd1.rightOuterJoin(rdd2).foreach(*println*) |

## 执行算子

　本质上在Action算子中通过SparkContext进行了提交作业runJob 操作，触发了RDD DAG的执行。  
例如，Action算子collect函数的代码如下，感兴趣的读者可以顺着这个入口进行源码剖析：  
/\*\*  
\* Return an array that contains all of the elements in this RDD.  
\*/  
def collect(): Array[T] = {  
/\* 提交 Job\*/  
val results = sc.runJob(this, (iter: Iterator[T]) => iter.toArray)  
Array.concat(results: \_\*)  
}

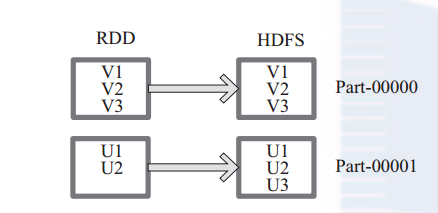
1. foreach

　　foreach对RDD中的每个元素都应用f函数操作，不返回RDD和Array，而是返回Uint。图表示foreach算子通过用户自定义函数对每个数据项进行操作。本例中自定义函数为println()，控制台打印所有数据项。  
　　

1. saveAsTextFile

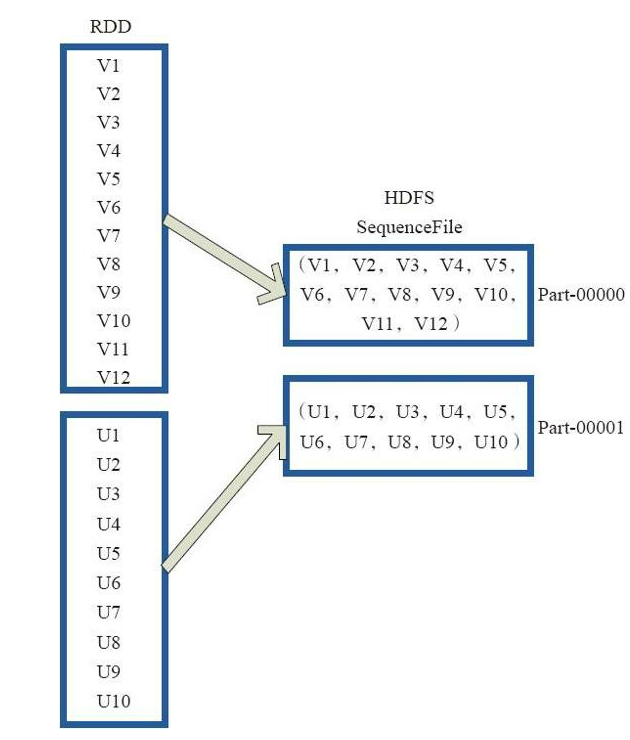
　　函数将数据输出，存储到HDFS的指定目录。

下面为 saveAsTextFile 函数的内部实现，其内部通过调用saveAsHadoopFile 进行实现：  
this.map(x => (NullWritable.get(), new Text(x.toString))).saveAsHadoopFile[TextOutputFormat[NullWritable, Text]](path)  
将RDD中的每个元素映射转变为(null， x.toString)，然后再将其写入 HDFS。  
　　图中左侧方框代表RDD分区,右侧方框代表HDFS的Block。通过函数将RDD的每个分区存储为HDFS中的一个 Block。



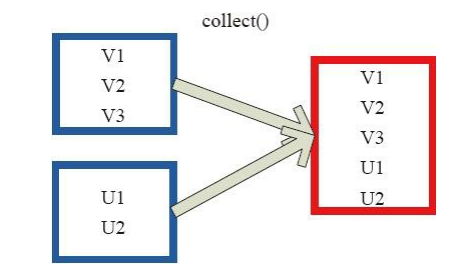
1. saveAsObjectFile

　　saveAsObjectFile将分区中的每10个元素组成一个Array，然后将这个Array序列化，映射为（Null，BytesWritable（Y））的元素，写入HDFS为SequenceFile的格式。  
　　下面代码为函数内部实现。  
　　map（x=>（NullWritable.get（），new BytesWritable（Utils.serialize（x））））  
　　图中的左侧方框代表RDD分区，右侧方框代表HDFS的Block。 通过函数将RDD的每个分区存储为HDFS上的一个Block。



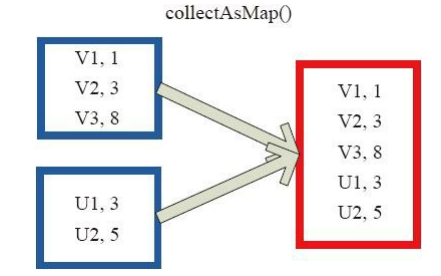
1. Collect

　　collect相当于toArray，toArray已经过时不推荐使用， collect 将分布式的 RDD 返回为一个单机的scala Array 数组。在这个数组上运用 scala 的函数式操作。  
　　图中左侧方框代表RDD分区，右侧方框代表单机内存中的数组。通过函数操作，将结果返回到 Driver 程序所在的节点，以数组形式存储。



1. collectAsMap

　　collectAsMap对（K，V）型的RDD数据返回一个单机HashMap。 对于重复K的RDD元素，后面的元素覆盖前面的元素。  
　　图中的左侧方框代表RDD分区，右侧方框代表单机数组。 数据通过collectAsMap函数返回给Driver程序计算结果，结果以HashMap形式存储。

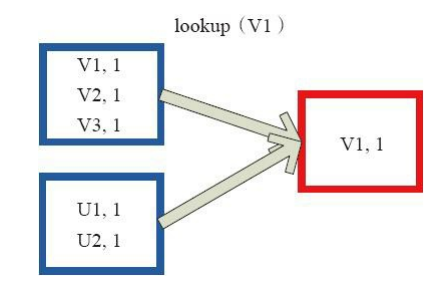


1. reduceByKeyLocally

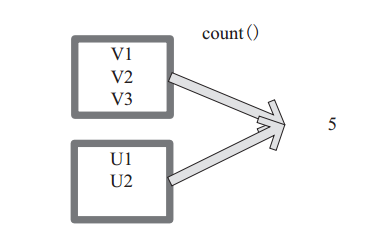
　　实现的是先reduce再collectAsMap的功能，先对RDD的整体进行reduce操作，然后再收集所有结果返回为一个HashMap。

1. lookup

下面代码为lookup的声明。  
lookup（key：K）：Seq[V]  
Lookup函数对（Key，Value）型的RDD操作，返回指定Key对应的元素形成的Seq。这个函数处理优化的部分在于，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）形成的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。  
　　图中的左侧方框代表RDD分区，右侧方框代表Seq，最后结果返回到Driver所在节点的应用中。



1. Count

　　count 返回整个 RDD 的元素个数。  
　　内部函数实现为：　　defcount():Long=sc.runJob(this,Utils.getIteratorSize\_).sum  
　　图 中，返回数据的个数为5。一个方块代表一个 RDD 分区。  


1. Top

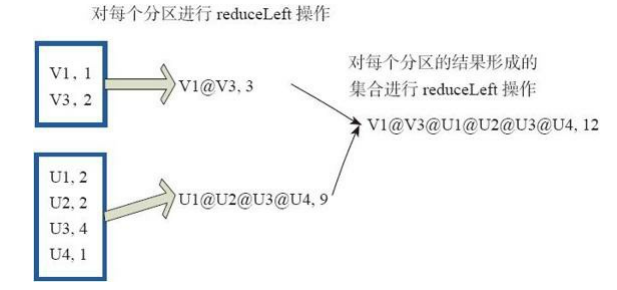
top可返回最大的k个元素。 函数定义如下。  
top（num：Int）（implicit ord：Ordering[T]）：Array[T]

相近函数说明如下。  
·top返回最大的k个元素。  
·take返回最小的k个元素。  
·takeOrdered返回最小的k个元素，并且在返回的数组中保持元素的顺序。  
·first相当于top（1）返回整个RDD中的前k个元素，可以定义排序的方式Ordering[T]。  
返回的是一个含前k个元素的数组。

1. reduce

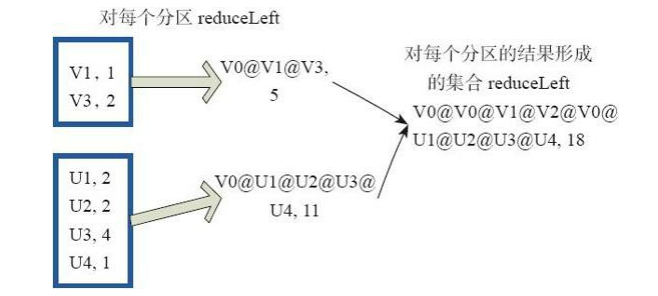
reduce函数相当于对RDD中的元素进行reduceLeft函数的操作。 函数实现如下。  
　　Some（iter.reduceLeft（cleanF））  
　　reduceLeft先对两个元素<K，V>进行reduce函数操作，然后将结果和迭代器取出的下一个元素<k，V>进行reduce函数操作，直到迭代器遍历完所有元素，得到最后结果。在RDD中，先对每个分区中的所有元素<K，V>的集合分别进行reduceLeft。 每个分区形成的结果相当于一个元素<K，V>，再对这个结果集合进行reduceleft操作。  
　　例如：用户自定义函数如下。  
　　f：（A，B）=>（A.\_1+"@"+B.\_1，A.\_2+B.\_2）  
　　图中的方框代表一个RDD分区，通过用户自定函数f将数据进行reduce运算。

示例最后的返回结果为V1@[1]V2U！@U2@U3@U4，12。



1. fold

　　fold和reduce的原理相同，但是与reduce不同，相当于每个reduce时，迭代器取的第一个元素是zeroValue。  
　　图中通过下面的用户自定义函数进行fold运算，图中的一个方框代表一个RDD分区。 读者可以参照reduce函数理解。  
　　fold（（"V0@"，2））（ （A，B）=>（A.\_1+"@"+B.\_1，A.\_2+B.\_2））



1. aggregate

 　　aggregate先对每个分区的所有元素进行aggregate操作，再对分区的结果进行fold操作。  
　　aggreagate与fold和reduce的不同之处在于，aggregate相当于采用归并的方式进行数据聚集，这种聚集是并行化的。而在fold和reduce函数的运算过程中，每个分区中需要进行串行处理，每个分区串行计算完结果，结果再按之前的方式进行聚集，并返回最终聚集结果。  
　　函数的定义如下。  
aggregate[B]（z： B）（seqop：（B，A） => B，combop：（B，B） => B）： B  
　　图通过用户自定义函数对RDD 进行aggregate的聚集操作，图中的每个方框代表一个RDD分区。  
　　[rdd.aggregate（"V0@"，2）（（A，B）=>（A.\_1+"@"+B.\_1，A.\_2+B.\_2）），（A，B）=>（A.\_1+"@"+B\_1，A.\_@+B\_.2））](mailto:rdd.aggregate（\"V0@\"，2）（（A，B）=>（A._1+\"@\"+B._1，A._2+B._2）），（A，B）=>（A._1+\"@\"+B_1，A._@+B_.2））)

|  |
| --- |
| **val** rddInt = sc.parallelize(*List*(1,2,3,4,5),2) **val** rddAggr1:(Int, Int)=rddInt.aggregate((0,0))((x,y)=> (x.\_1+y, x.\_2+1),(x,y)=>(x.\_1+y.\_1, x.\_2+y.\_2)) **val** rddAggr2:(Int) = rddInt.aggregate(0)((x,y)=>x+y,(x,y)=>x+y) *println*(rddAggr2.toString()) |

## 