6. 了解RDD

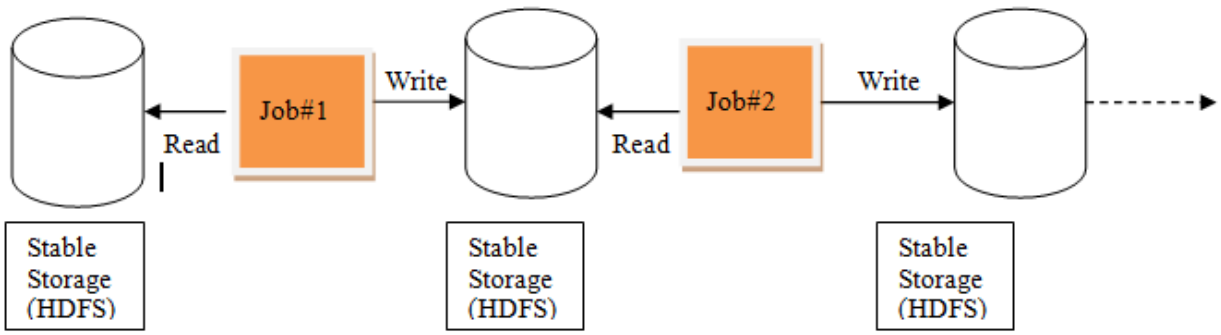
6.1 RDD的由来

RDD是Spark的基石，是实现Spark数据处理的核心抽象。那么RDD为什么会产生呢？

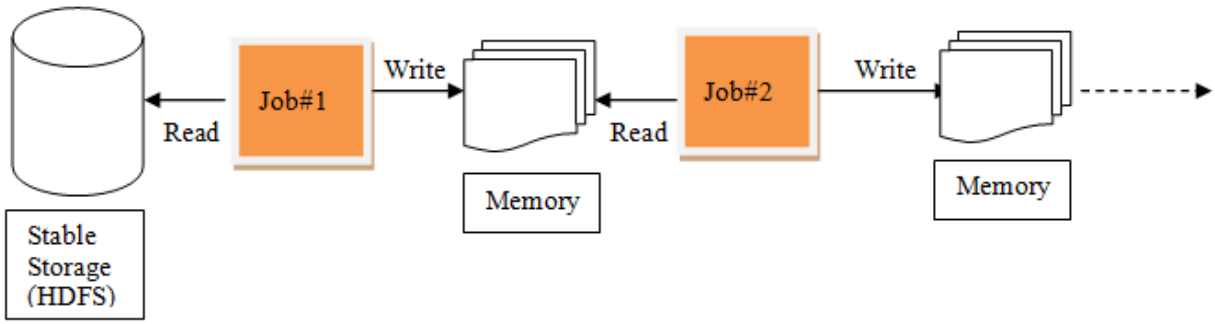
Hadoop的MapReduce是一种基于数据集的工作模式，面向数据，这种工作模式一般是从存储上加载数据集，然后操作数据集，最后写入物理存储设备。数据更多面临的是一次性处理。

MR的这种方式对数据领域两种常见的操作不是很高效。第一种是迭代式的算法。比如机器学习中ALS、凸优化梯度下降等。这些都需要基于数据集或者数据集的衍生数据反复查询反复操作。MR这种模式不太合适，即使多MR串行处理，性能和时间也是一个问题。数据的共享依赖于磁盘。另外一种是交互式数据挖掘，MR显然不擅长。

MR中的迭代：



Spark中的迭代：



我们需要一个效率非常快，且能够支持迭代计算和有效数据共享的模型，Spark应运而生。RDD是基于工作集的工作模式，更多的是面向工作流。

但是无论是MR还是RDD都应该具有类似位置感知、容错和负载均衡等特性。

6.2 RDD概念

RDD（Resilient Distributed Dataset）是一个**弹性分布式数据集**，是Spark中最基本的数据抽象，它代表一个**不可变、可分区、里面的元素可并行计算**的集合。

RDD具有数据流模型的特点：自动容错、位置感知性调度和可伸缩性。RDD允许用户在执行多个查询时显式地将工作集缓存在内存中，后续的查询能够重用工作集，这极大地提升了查询速度。

q1 弹性的含义

1) 自动进行内存和磁盘数据存储的切换

Spark优先把数据放到内存中，如果内存放不下，就会放到磁盘里面，程序进行自动的存储切换

2) 基于血统的高效容错机制

在RDD进行转换和动作的时候，会形成RDD的Lineage依赖链，当某一个RDD失效的时候，可以通过重新计算上游的RDD来重新生成丢失的RDD数据。

3) Task如果失败会自动进行特定次数的重试

RDD的计算任务如果运行失败，会自动进行任务的重新计算，默认次数是4次。

4) Stage如果失败会自动进行特定次数的重试

如果Job的某个Stage阶段计算失败，框架也会自动进行任务的重新计算，默认次数也是4次。

5) Checkpoint和Persist可主动或被动触发

RDD可以通过Persist持久化将RDD缓存到内存或者磁盘，当再次用到该RDD时直接读取就行。也可以将RDD进行检查点，检查点会将数据存储在HDFS中，该RDD的所有父RDD依赖都会被移除。

6) 数据调度弹性

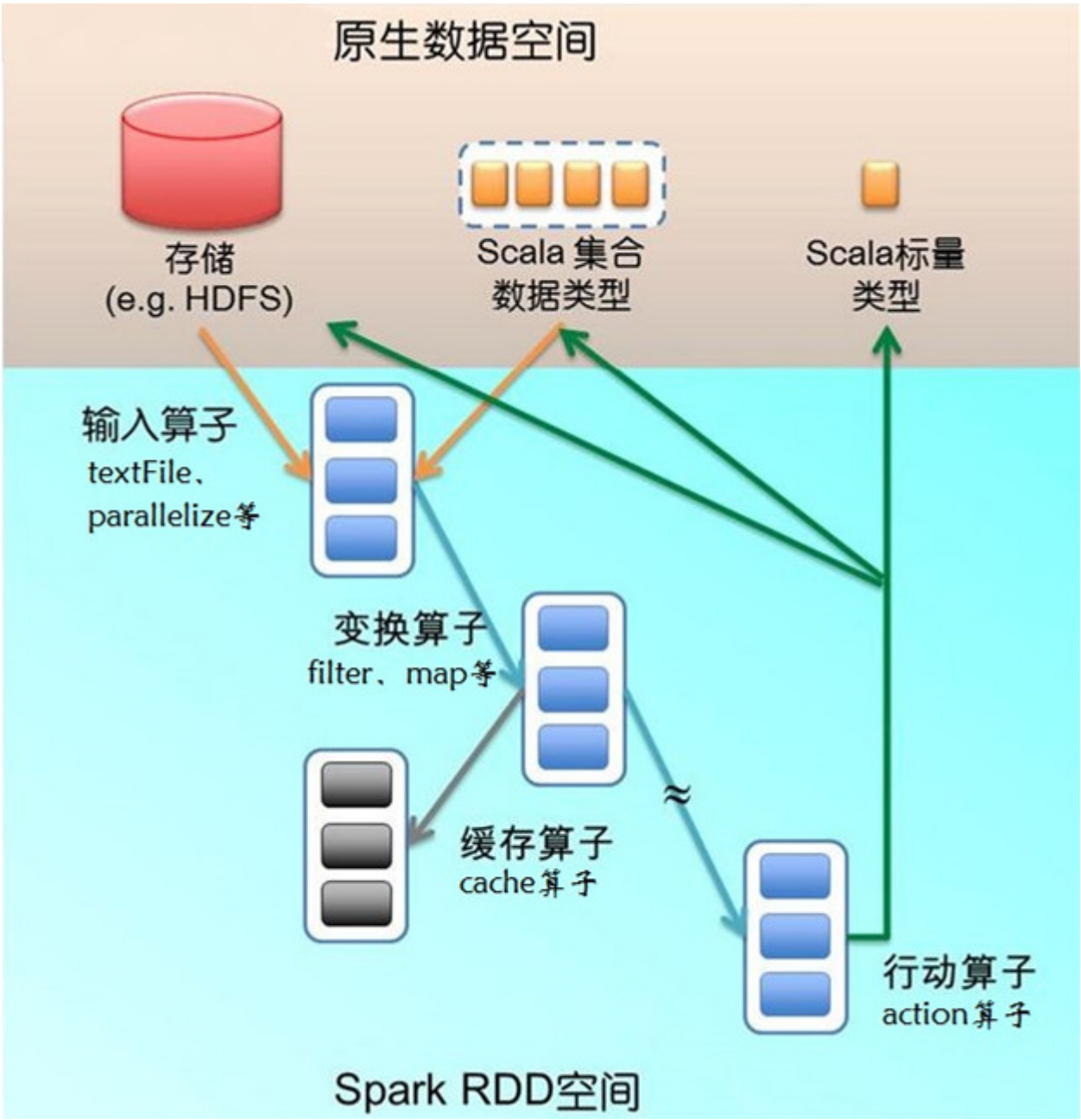
Spark把这个JOB执行模型抽象为通用的有向无环图DAG，可以将多Stage的任务串联或并行执行，调度引擎自动处理Stage的失败以及Task的失败。

7) 数据分片的高度弹性

可以根据业务的特征，动态调整数据分片的个数，提升整体的应用执行效率。

q2 RDD从哪来

RDD 可以从一个 Hadoop 文件系统（或者任何其它 Hadoop 支持的文件系统），或者一个在 driver program（驱动程序）中已存在的 Scala 集合，以及通过 transforming（转换）来创建一个 RDD。用户为了让它在整个并行操作中更高效的重用，也许会让 Spark persist（持久化）一个 RDD 到内存中。最后，RDD 会自动的从节点故障中恢复。



RDD支持两种操作:转化操作和行动操作。RDD 的转化操作是返回一个新的 RDD的操作，比如 map()和 filter()，而行动操作则是向驱动器程序返回结果或把结果写入外部系统的操作。比如 count() 和 first()。 Spark采用惰性计算模式，RDD只有第一次在一个行动操作中用到时，才会真正计算。Spark可以优化整个计算过程。默认情况下，Spark 的 RDD 会在你每次对它们进行行动操作时重新计算。如果想在多个行动操作中重用同一个 RDD，可以使用 RDD.persist() 让 Spark 把这个 RDD 缓存下来。

RDD里并不真正存储数据，对RDD进行操作，是在driver端转换成task，然后下发到Executor，将计算分散到多台集群上运行。

RDD相当于数据的一个代理，对代理进行操作，代理隔离了细节，使分布式编程类似于单机程序，无需关心任务调度，容错等。

spark提供了一个抽象叫RDD，它是集群上跨节点的元素集合，可以并行执行；

RDD可以通过Hadoop文件（或者其他Hadoop支持的文件系统）或者存在于驱动程序上的一个已知 的scala集合创建，并进行转换；

用户可以把RDD持久化到内存，在并行计算中高效实用；

RDD具有容错能力。

用户可以控制 RDDs 的两个方面：数据存储和分区.对于需要复用的 RDD, 用户可以明确的选择一个数据存储策略（比如内存缓存）. 他们也可以基于一个元素的 key 来为 RDD 所有的元素在机器节点间进行数据分区, 这样非常利于数据分布优化, 比如给两个数据集进行相同的 hash 分区, 然后进行 join, 可以提高 join 的性能.

通过调用 RDDs 的 persist 方法来缓存后续需要复用的 RDDs. Spark 默认是将缓存数据放在内存中, 但是如果内存不足的话则会写入到磁盘中. 用户可以通过 persist 的参数来调整缓存策略, 比如只将数据存储在磁盘中或者复制备份数据到多台机器. 最后, 用户可以为每一个 RDDs 的缓存设置优先级, 以达到哪个在内存中的 RDDs 应该首先写道磁盘中。

q3 为什么要抽象一个RDD

降低并行编程的难度。

6.3 RDD属性

RDD源码里有关RDD的属性

```
 * Internally, each RDD is characterized by five main properties:
 *
 *  - A list of partitions
 *  - A function for computing each split
 *  - A list of dependencies on other RDDs
 *  - Optionally, a Partitioner for key-value RDDs (e.g. to say that the RDD is hash-partitioned)
 *  - Optionally, a list of preferred locations to compute each split on (e.g. block locations for
 *    an HDFS file)
```

1）一组分片（Partition），即数据集的基本组成单位。对于RDD来说，每个分片都会被一个计算任务处理，并决定并行计算的粒度。用户可以在创建RDD时指定RDD的分片个数，如果没有指定，那么就会采用默认值。默认值就是程序所分配到的CPUCore的数目。

2）一个计算每个分区的函数。Spark中RDD的计算是以分片为单位的，每个RDD都会实现compute函数以达到这个目的。compute函数会对迭代器进行复合，不需要保存每次计算的结果。

3）RDD之间的依赖关系。RDD的每次转换都会生成一个新的RDD，所以RDD之间就会形成类似于流水线一样的前后依赖关系。在部分分区数据丢失时，Spark可以通过这个依赖关系重新计算丢失的分区数据，而不是对RDD的所有分区进行重新计算。

4）一个Partitioner，即RDD的分片函数。当前Spark中实现了两种类型的分片函数，一个是基于哈希的HashPartitioner，另外一个是基于范围的RangePartitioner。只有对于key-value的RDD，才会有Partitioner，非key-value的RDD的Parititioner的值是None。Partitioner函数不但决定了RDD本身的分片数量，也决定了parent RDD Shuffle输出时的分片数量。

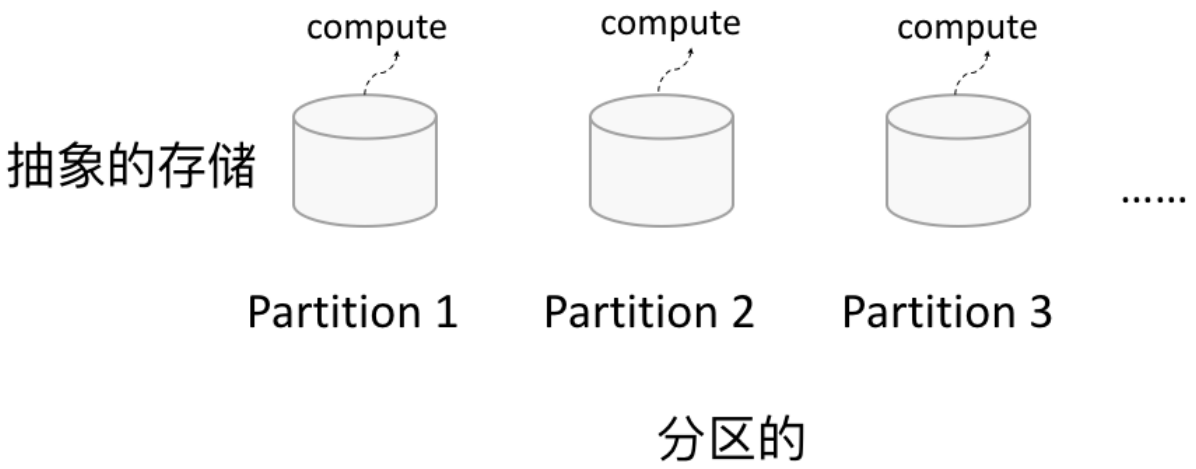
5）一个列表，存储存取每个Partition的优先位置（preferred location）。对于一个HDFS文件来说，这个列表保存的就是每个Partition所在的块的位置。按照“移动数据不如移动计算”的理念，Spark在进行任务调度的时候，会尽可能地将计算任务分配到其所要处理数据块的存储位置。

6.4 RDD特点

RDD表示只读的分区的数据集，对RDD进行改动，只能通过RDD的转换操作，由一个RDD得到一个新的RDD，新的RDD包含了从其他RDD衍生所必需的信息。RDDs之间存在依赖，RDD的执行是按照血缘关系延时计算的。如果血缘关系较长，可以通过持久化RDD来切断血缘关系。

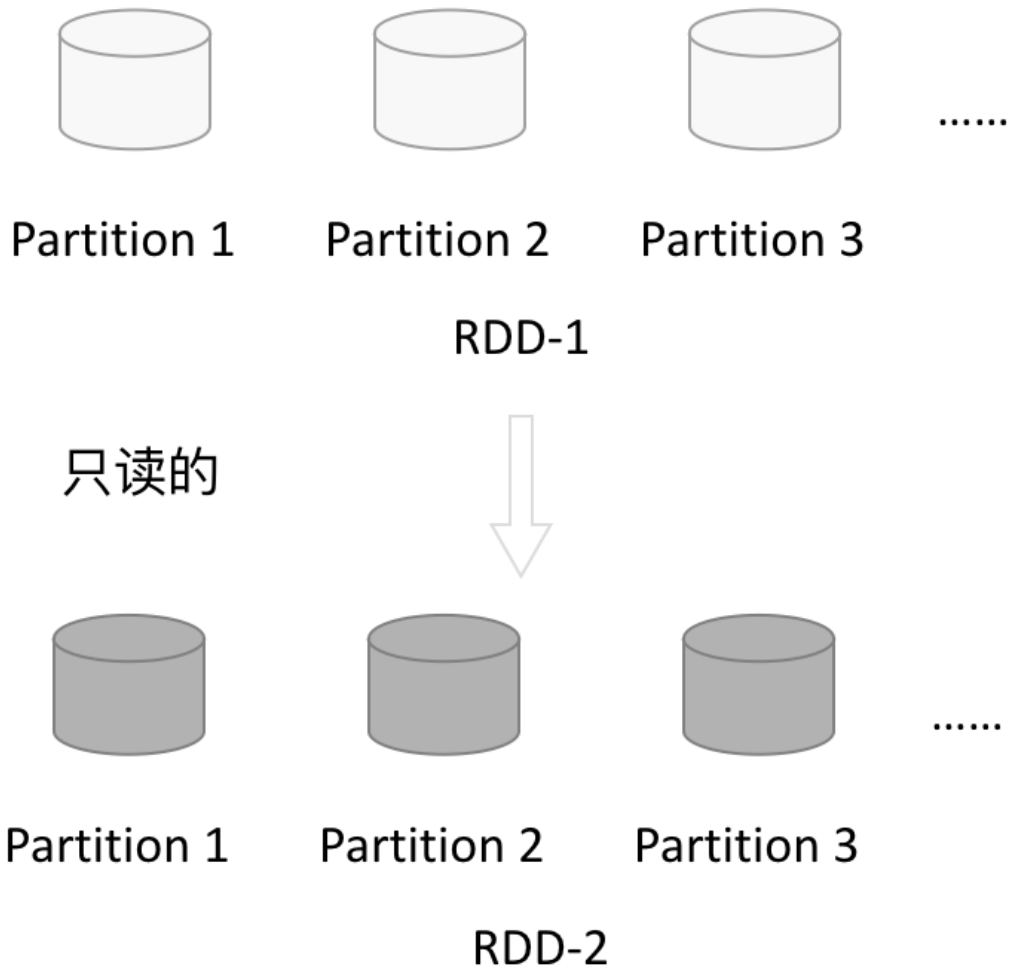
6.4.1 分区

RDD逻辑上是分区的，每个分区的数据是抽象存在的，计算的时候会通过一个compute函数得到每个分区的数据。如果RDD是通过已有的文件系统构建，则compute函数是读取指定文件系统中的数据，如果RDD是通过其他RDD转换而来，则compute函数是执行转换逻辑将其他RDD的数据进行转换。

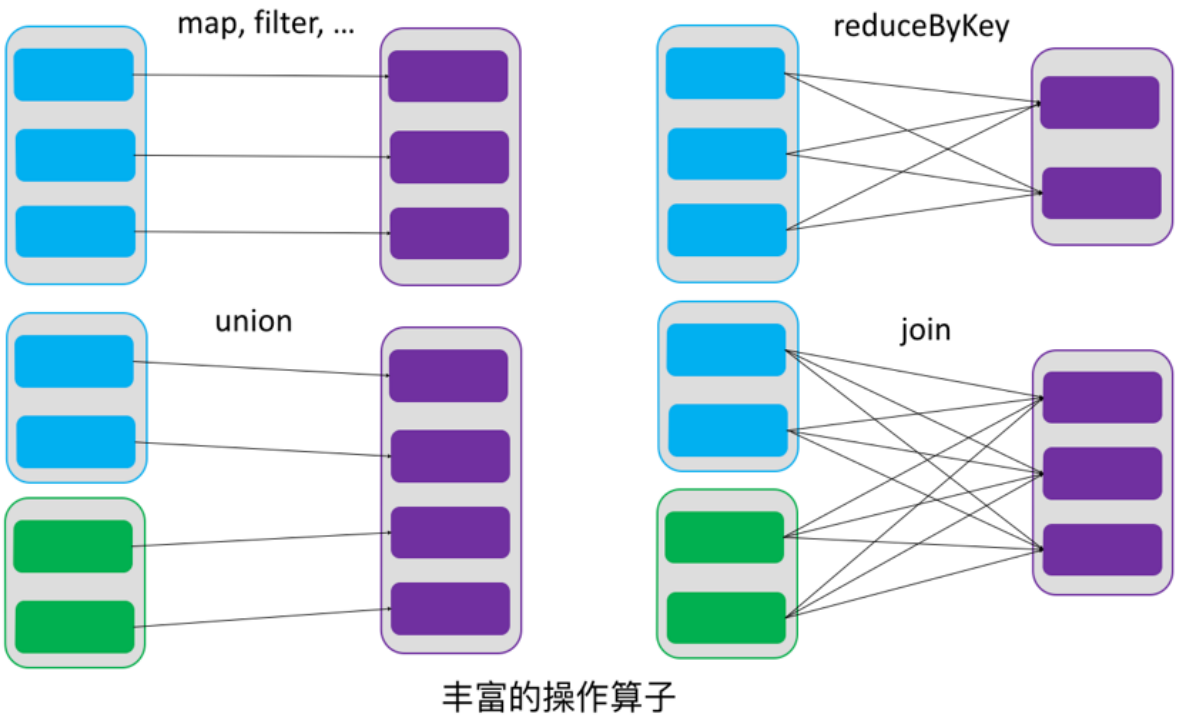


1.4.2 只读

如下图所示，RDD是只读的，要想改变RDD中的数据，只能在现有的RDD基础上创建新的RDD。



由一个RDD转换到另一个RDD，可以通过丰富的操作算子实现，不再像MapReduce那样只能写map和reduce了，如下图所示。

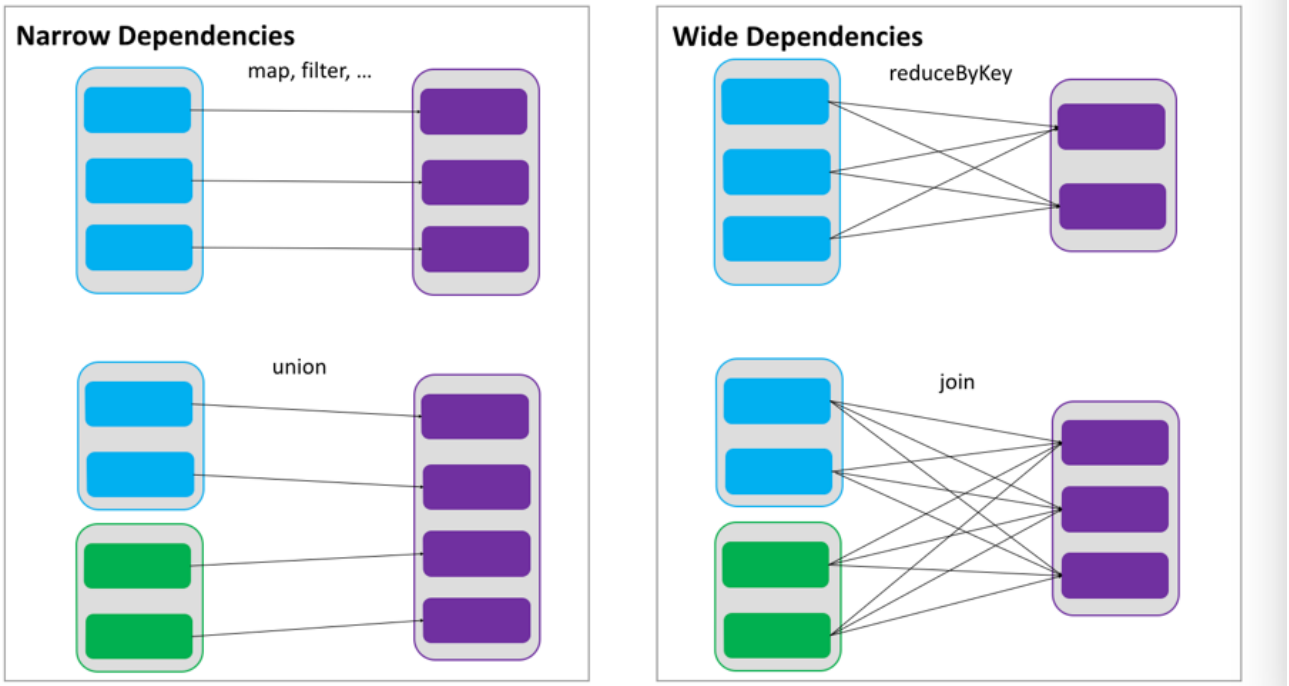


RDD的操作算子包括两类，一类叫做transformations，它是用来将RDD进行转化，构建RDD的血缘关系；另一类叫做actions，它是用来触发RDD的计算，得到RDD的相关计算结果或者将RDD保存的文件系统中。下图是RDD所支持的操作算子列表。

Transformations	*map*(*f* : T ⇒ U)	:	RDD[T] ⇒ RDD[U]
	filter(*f* : T ⇒ Bool)	:	RDD[T] ⇒ RDD[T]
	flatMap(*f* : T ⇒ Seq[U])	:	RDD[T] ⇒ RDD[U]
	sample(*fraction* : Float)	:	RDD[T] ⇒ RDD[T] (Deterministic sampling)
	groupByKey()	:	RDD[(K, V)] ⇒ RDD[(K, Seq[V])]
	reduceByKey(*f* : (V, V) ⇒ V)	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	union()	:	(RDD[T], RDD[T]) ⇒ RDD[T]
	join()	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (V, W))]
	cogroup()	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (Seq[V], Seq[W]))]
	crossProduct()	:	(RDD[T], RDD[U]) ⇒ RDD[(T, U)]
	mapValues(*f* : V ⇒ W)	:	RDD[(K, V)] ⇒ RDD[(K, W)] (Preserves partitioning)
	sort(*c* : Comparator[K])	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	partitionBy(*p* : Partitioner[K])	:	RDD[(K, V)] ⇒ RDD[(K, V)]
Actions	*count*()	:	RDD[T] ⇒ Long
	collect()	:	RDD[T] ⇒ Seq[T]
	reduce(*f* : (T, T) ⇒ T)	:	RDD[T] ⇒ T
	lookup(*k* : K)	:	RDD[(K, V)] ⇒ Seq[V] (On hash/range partitioned RDDs)
	save(*path* : String)	:	Outputs RDD to a storage system, *e.g.*, HDFS

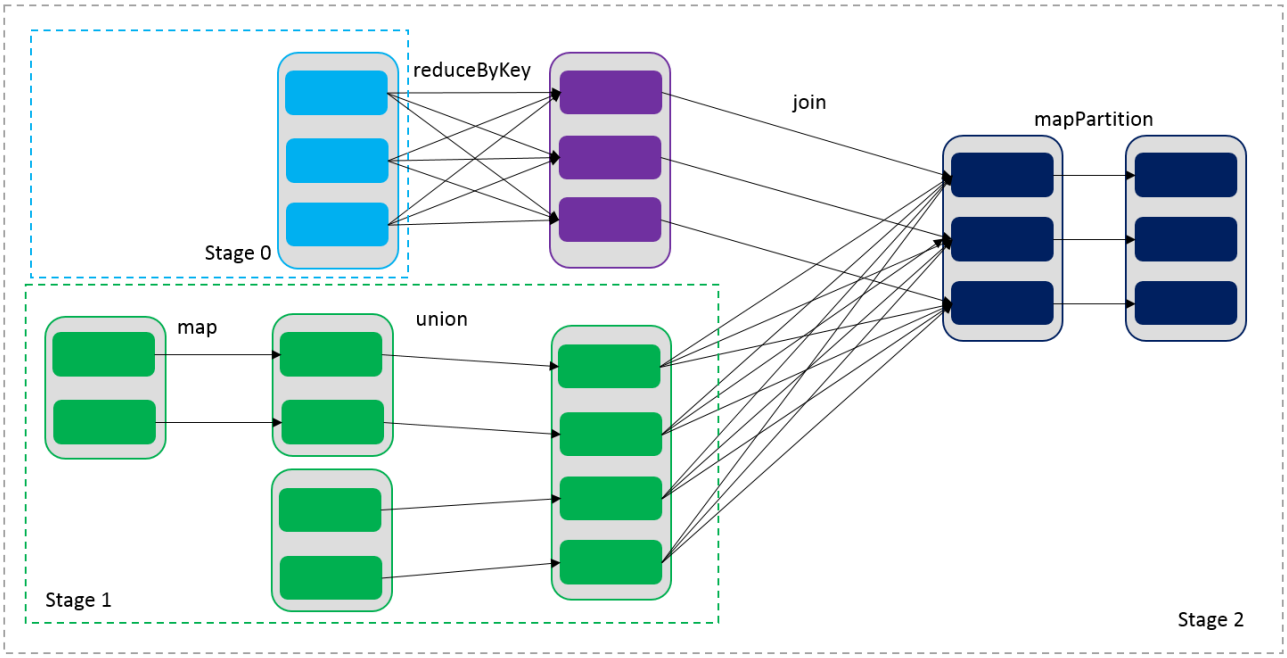
1.4.3 依赖

RDDs通过操作算子进行转换，转换得到的新RDD包含了从其他RDDs衍生所必需的信息，RDDs之间维护着这种血缘关系，也称之为依赖。如下图所示，依赖包括两种，一种是窄依赖，RDDs之间分区是一一对应的，另一种是宽依赖，下游RDD的每个分区与上游RDD(也称之为父RDD)的每个分区都有关，是多对多的关系。



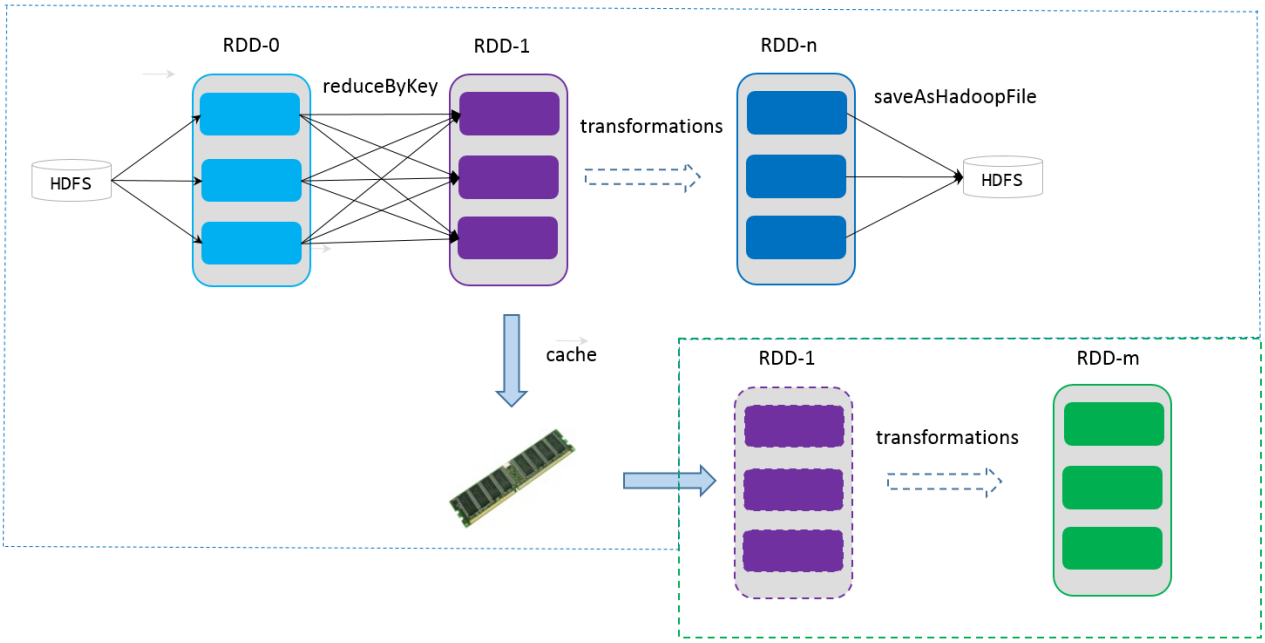
RDD之间依赖关系

通过RDDs之间的这种依赖关系，一个任务流可以描述为DAG(有向无环图)，如下图所示，在实际执行过程中宽依赖对应于Shuffle(图中的reduceByKey和join)，窄依赖中的所有转换操作可以通过类似于管道的方式一气呵成执行(图中map和union可以一起执行)。



1.4.4 缓存

如果在应用程序中多次使用同一个RDD，可以将该RDD缓存起来，该RDD只有在第一次计算的时候会根据血缘关系得到分区的数据，在后续其他地方用到该RDD的时候，会直接从缓存处取而不用再根据血缘关系计算，这样就加速后期的重用。如下图所示，RDD-1经过一系列的转换后得到RDD-n并保存到hdfs，RDD-1在这一过程中会有个中间结果，如果将其缓存到内存，那么在随后的RDD-1转换到RDD-m这一过程中，就不会计算其之前的RDD-0了。



1.4.5 checkpoint

虽然RDD的血缘关系天然地可以实现容错，当RDD的某个分区数据失败或丢失，可以通过血缘关系重建。但是对于长时间迭代型应用来说，随着迭代的进行，RDDs之间的血缘关系会越来越长，一旦在后续迭代过程中出错，则需要通过非常长的血缘关系去重建，势必影响性能。为此，RDD支持checkpoint将数据保存到持久化的存储中，这样就可以切断之前的血缘关系，因为checkpoint后的RDD不需要知道它的父RDDs了，它可以从checkpoint处拿到数据。

给定一个RDD我们至少可以知道如下几点信息：1、分区数以及分区方式；2、由父RDDs衍生而来的相关依赖信息；3、计算每个分区的数据，计算步骤为：1）如果被缓存，则从缓存中取的分区的数据；2）如果被checkpoint，则从checkpoint处恢复数据；3）根据血缘关系计算分区的数据。

6.5 RDD和普通集合的对比

RDD里面计入的是描述信息（从哪里读取数据，以后对数据如何计算），RDD的方法分为两类Transformation(lazy)，Action(生成task，并发送到executor执行)。

scala集合中存储的时真正要计算的数据、执行方法后立即返回结果。

一系列分区

每一个输入切片会有一个函数作用在上面

RDD之间存在依赖关系（是父RDD调用什么方法，传入那些函数得到的）

（可选）RDD中如果存储的是KV，shuffle是会有一个分区器，默认是hash partitioner

（可选) 如果读取的是HDFS数据，那么会有一个最优位置。

6.6 创建RDD

1）由一个Scala集合或数组以并行化的方式创建。

一般用来测试或者实验

```
val rdd1 =sc.parallelize(Array(1,2,3,4,5,6,7,8))
```

RDD一旦创建，分布式的数据集就可以执行并行计算

```
rdd1.reduce((a,b)=>a+b)
```

RDD创建的过程中，一个重要参数是把数据集切分的分区数。Spark针对集群中的每一个分区执行一个任务（task），一般，集群中每一个CPU分配2-4个分区，Spark会基于集群自动设置分区数目，但是，也可以手动指定，比如：

```
sc.parallelize(data, 10)
```

注意：

通过一个集合创建一个RDD，也可以使用

```
sc.makeRDD(Array(1,2,3,4,5,6,7,8))
```

通过查看源码发现makeRDD和parallelize的关系。

2）由外部存储系统的数据集创建，包括本地的文件系统，还有所有Hadoop支持的数据集，比如HDFS、Cassandra、HDFS, Cassandra, HBase, Amazon S3等，Spark支持文本文件、序列文件（SequenceFiles are flat files consisting of binary key/value pairs."Flat"which means it has no structure for indexing and there are usually no structural relationships between the records.）以及任何Hadoop支持的数据集。

```
//textFile需要一个表示文件地址的URI参数，该文件可以是本地文件，或者 hdfs://, s3a://等
//textFile读取文件的结果是以文件每一行作为一个元素的集合
//val rdd1 =sc.textFile("hdfs://node01/words.txt")
val distFile = sc.textFile("data.txt")
distFile.map(s => s.length).reduce((a, b) => a + b)
```

使用Spark读取文件需要注意的点：

- 如果读取的是本地文件，该文件必须是worker节点可以存取的，通过拷贝文件到所有的worker节点或者使用基于挂载的共享文件系统；
- spark中所有参数是文件的方法，包括文本文件，都支持目录、压缩文件和通配符文件，比如可以使用

```
textFile("/my/directory")
textFile("/my/directory/*.txt")
textFile("/my/directory/*.gz")
```

- textFile 方法也有一个可选参数控制文件的分区数量，默认情况下，Spark文件的每一个block生成一个partition分区（HDFS默认快大小为128MB），但是也可以传入一个更大的值获取更多的分区，但是无法获取少于block个数的分区个数。

3）使用已有的RDD，通过Transformation生成一个新的RDD

扩展:查看RDD:

方式一使用collect()

```
myRDD.collect().foreach(println)
```

方式二使用take(number)

```
myRDD.take(n).foreach(println)
```

6.7. 基础知识

RDD的算子分两种类型：Transformation和Action，具有Transformation属性的算子，具有惰性，不会直接计算结果，具有Action属性的算子，触发算子的执行。

6.7.1 基本认识

```
val lines = sc.textFile("data.txt")
val lineLengths = lines.map(s => s.length)
val totalLength = lineLengths.reduce((a, b) => a + b)
```

第一行定义了一个RDD，数据集不会加载到内存，或者不会有基于该数据集的操作，lines仅仅是一个指向该文件的一个指针；

第二行定义了lineLengths作为map转换函数的结果，由于惰性，lineLengths 不会立即计算；

最后，调用reduce，它是一个action算子，这是Spark把计算分成多个任务在各个机上执行，每台机器执行自己这部分数据的map和reduction，返回结果给驱动程序。

如果，之后还打算使用lineLengths，可以添加：

```
lineLengths.persist()
```

在reduce之前，可以把lineLengths在首次计算后保存在内存。

6.7.2. 传入函数

Spark 有很多API需要提供函数式的传入参数，有两种方式推荐：

- 匿名函数，在执行代码较少的情况下使用；
- 使用全局单例对象中的静态方法；

```
object MyFunctions {
  def func1(s: String): String = { ... }
}
myRdd.map(MyFunctions.func1)
```

Note that while it is also possible to pass a reference to a method in a class instance (as opposed to a singleton object), this requires sending the object that contains that class along with the method. For example, consider:

```
class MyClass {
  def func1(s: String): String = { ... }
  def doStuff(rdd: RDD[String]): RDD[String] = { rdd.map(func1) }
}
```

Here, if we create a new MyClass instance and call doStuff on it, the map inside there references the func1 method of that MyClass instance, so the whole object needs to be sent to the cluster. It is similar to writing rdd.map(x => this.func1(x)).

In a similar way, accessing fields of the outer object will reference the whole object:

```
class MyClass {
  val field = "Hello"
  def doStuff(rdd: RDD[String]): RDD[String] = { rdd.map(x => field + x) }
}
```

is equivalent to writing rdd.map(x => this.field + x), which references all of this. To avoid this issue, the simplest way is to copy field into a local variable instead of accessing it externally:

```
def doStuff(rdd: RDD[String]): RDD[String] = {
  val field_ = this.field
  rdd.map(x => field_ + x)
}
```

6.7.3. 理解闭包

Spark的难点之一是在集群中执行代码时，理解变量和方法的作用范围以及生命周期。

```
var counter = 0
var rdd = sc.parallelize(data)
// Wrong: Don't do this!!
rdd.foreach(x => counter += x)
println("Counter value: " + counter)
```

本地和集群模式

上面的代码行为是不确定的，并且可能无法按预期正常工作。执行作业时，Spark 会分解 RDD 操作到每个 executor 中的 task 里。在执行之前，Spark 计算任务的 closure（闭包）。闭包是指 executor 要在RDD上进行计算时必须对执行节点可见的那些变量和方法（在这里是foreach()）。闭包被序列化并被发送到每个 executor。

闭包的变量副本发给每个 executor ，当 counter 被 foreach 函数引用的时候，它已经不再是 driver node 的 counter 了。虽然在 driver node 仍然有一个 counter 在内存中，但是对 executors 已经不可见。executor 看到的只是序列化的闭包一个副本。所以 counter 最终的值还是 0，因为对 counter 所有的操作均引用序列化的 closure 内的值。

在 local 本地模式，在某些情况下的 foreach 功能实际上是同一 JVM 上的驱动程序中执行，并会引用同一个原始的 counter 计数器，实际上可能更新.

为了确保这些类型的场景明确的行为应该使用的 Accumulator 累加器。当一个执行的任务分配到集群中的各个 worker 结点时，Spark 的累加器是专门提供安全更新变量的机制。

打印 RDD 的 elements

另一种常见的语法用于打印 RDD 的所有元素使用 rdd.foreach(println) 或 rdd.map(println)。在一台机器上，这将产生预期的输出和打印 RDD 的所有元素。然而，在集群 cluster 模式下，stdout 输出正在被执行写操作 executors 的 stdout 代替，而不是在一个驱动程序上，因此 stdout 的 driver 程序不会显示这些！要打印 driver 程序的所有元素，可以使用的 collect() 方法首先把 RDD 放到 driver 程序节点上: rdd.collect().foreach(println)。这可能会导致 driver 程序耗尽内存，因为 collect() 获取整个 RDD 到一台机器; 如果你只需要打印 RDD 的几个元素，一个更安全的方法是使用 take(): rdd.take(100).foreach(println)。

6.7.4. 键值对

虽然大多数 Spark 操作工作在包含任何类型对象的 RDDs 上，只有少数特殊的操作可用于 Key-Value 对的 RDDs. 最常见的是分布式 “shuffle” 操作，如通过元素的 key 来进行 grouping 或 aggregating 操作.

例如，下面的代码使用的 Key-Value 对的 reduceByKey 操作统计文本文件中每一行出现了多少次:

```
val lines = sc.textFile("data.txt")
val pairs = lines.map(s => (s, 1))
val counts = pairs.reduceByKey((a, b) => a + b)
```

我们也可以使用 counts.sortByKey() ，例如，在对按字母顺序排序，最后 counts.collect() 把他们作为一个数据对象返回给 driver 程序。

注意: 当在 key-value 对操作中使用自定义的 objects 作为 key 时, 必须确保有一个自定义的 equals() 方法和 hashCode() 方法。

7.分区

7.1为什么要分区

分区的目标之一就是减少数据通过网络传输的量。

最大化的利用分区有益于执行作业的效率，因为分区后可以并行执行大量task在不同集群节点上。

7.2 如何进行分区

默认情况下，spark对每一个RDD会自动计算分区，但是我们也可以在创建RDD的时候指定分区数。

7.2.1 系统分区

对于键值对数据进行分区，可以基于分区器，有两种分区器可供选择

哈希分区：分区基于存储对象的哈希码，目前使用的计算公式是：

partition = hashCode % numberOfPartitions

范围分区：通过数值范围分区。

比如我们有一个list，包含1到100,100个数，如果我们想存储5个分区中，范围分区首先计算各个范围 (5: [1-20], [21-40], [41-60], [61-80], [81-100])，然后，数据会根据数据的Key值匹配范围值，例如，key值为5的数据放在范围[1-20]分区内，key值为59的数据放在范围[41-60]分区内。

如何范围的数目低于期望的分区数，分区数目会减少。

7.2.2 自定义分区器

implementations of abstract class org.apache.spark.Partitioner

7.2.3 重新分区

coalesce(numPartitions) - 减少分区数，该方法在很多分区数据较少的情况下可以使用，通过减少分区提高执行效率. 如果允许shuffle，不仅可以减少分区，也可以增加，同时shuffle会因为数据分区导致数据通过网络移动，有可能导致性能风险。 repartition(numPartitions)

repartitionAndSortWithinPartitions(partitioner)在shuffle的过程中重新指定分区器

7.3 分区的属性

一个RDD有多个分区分散在不同节点上；

一个分区的数据都来自于某一个RDD；

一个分区的数据不可能来自于两个不同的RDD；

一个分区必定存储于某一台机器（节点）上，在一个节点上被处理；

一台机器（节点）可以存储多个不同分区（每个CPU分配2-4个分区）；

7.4 确定分区数目

分区数目的多少回导致什么问题？

分区数目代表了并行度的

太多小的分区会导致任务调度的代价巨大，同时资源释放导致垃圾回收的压力巨大；

太少的分区导致并行度降低

确定一个合理的分区数目，考虑以下几点：

一般分区数目在100到10000之间

最小阈值 = 2*集群中总核数

最大阈值 = 任务在100ms内能处理完毕。

最后一个stage的RDD决定了分区的数目，但是coalesce 或者 repartition会改变这个数目。

7.5 分布式数据分区

从HDFS中读文件，一个分区对应一个hdfs文件分片，因此Spark 使用已有的机制对HDFS数据进行分区。

如果是将Driver端的Scala集合并行化创建RDD，没有指定RDD的分区数目，分区数等于集群中分配给该应用的总核数。

如果是从HDFS中读取数据创建RDD，并且设置了最新分区数量是1，那么RDD的分区数据即是输入切片的数据，如果不设置最小分区的数量，spark调用textfile时默认传入2，那么RDD的分区数量大于等于切片的数量。

block和split的理解: 两者是从不同的角度来定义的：HDFS以固定大小的block为基本单位存储数据（分布式文件系统，实际存储角度，物理存储单位），而MapReduce以split作为处理单位（编程模型角度，逻辑单位）。

对于文件中的一行记录，可能会划分到不同的block中，也可能划分到不同的split中。

split是逻辑上的概念，它只包含一些元数据信息，比如数据起始位置、数据长度、数据所在节点等，它的划分方法完全由用户自己决定。split的多少决定Map Task的数目，因为每个split交给一个Map Task处理。

大小关系：>,=,<均有可能。

7.6 TextFile分区窥探

toDebugString to see the lineage of RDDs **getNumPartitions** to, shockingly, get the number of partitions **glom** to see clearly how your data are partitioned

源码跟踪：

```
val lines:RDD[String] = sc.textFile("hdfs://192.168.56.3:9000/")
===>
  def textFile(
```

```
    path: String,
    minPartitions: Int = defaultMinPartitions): RDD[String] = withScope {
  assertNotStopped()
  hadoopFile(path, classOf[TextInputFormat], classOf[LongWritable], classOf[Text],
    minPartitions).map(pair => pair._2.toString).setName(path)
}
===>hadoopFile到
class HadoopRDD[K, V](
  sc: SparkContext,
  broadcastedConf: Broadcast[SerializableConfiguration],
  initLocalJobConfFuncOpt: Option[JobConf => Unit],
  inputFormatClass: Class[_ <: InputFormat[K, V]],
  keyClass: Class[K],
  valueClass: Class[V],
  minPartitions: Int)
extends RDD[(K, V)](sc, Nil) with Logging {

if (initLocalJobConfFuncOpt.isDefined) {
  sparkContext.clean(initLocalJobConfFuncOpt.get)
}
===>lines.count()到HadoopRDD.scala中getPartitions断点
  override def getPartitions: Array[Partition] = {
  val jobConf = getJobConf()
  // add the credentials here as this can be called before SparkContext initialized
  SparkHadoopUtil.get.addCredentials(jobConf)
  val inputFormat = getInputFormat(jobConf)
  val inputSplits = inputFormat.getSplits(jobConf, minPartitions)
  val array = new Array[Partition](inputSplits.size)
  for (i <- 0 until inputSplits.size) {
    array(i) = new HadoopPartition(id, i, inputSplits(i))
  }
  array
}


 ===>FileInputFormat中getSplits实现

      /** Splits files returned by {@link #listStatus(JobConf)} when
 * they're too big.*/
public InputSplit[] getSplits(JobConf job, int numSplits)
  throws IOException {
  Stopwatch sw = new Stopwatch().start();
  FileStatus[] files = listStatus(job);

  // Save the number of input files for metrics/loadgen
  job.setLong(NUM_INPUT_FILES, files.length);
  long totalSize = 0;                           // compute total size
  for (FileStatus file: files) {                // check we have valid files
    if (file.isDirectory()) {
      throw new IOException("Not a file: "+ file.getPath());
    }
    totalSize += file.getLen();

  }
```

```
long goalSize = totalSize / (numSplits == 0 ? 1 : numSplits);
long minSize = Math.max(job.getLong(org.apache.hadoop.mapreduce.lib.input.
  FileInputFormat.SPLIT_MINSIZE, 1), minSplitSize);

// generate splits
ArrayList<FileSplit> splits = new ArrayList<FileSplit>(numSplits);
NetworkTopology clusterMap = new NetworkTopology();
for (FileStatus file: files) {
  Path path = file.getPath();
  long length = file.getLen();
  if (length != 0) {
    FileSystem fs = path.getFileSystem(job);
    BlockLocation[] blkLocations;
    if (file instanceof LocatedFileStatus) {
      blkLocations = ((LocatedFileStatus) file).getBlockLocations();
    } else {
      blkLocations = fs.getFileBlockLocations(file, 0, length);
    }
    if (isSplitable(fs, path)) {
      long blockSize = file.getBlockSize();
      long splitSize = computeSplitSize(goalSize, minSize, blockSize);

      long bytesRemaining = length;
      while (((double) bytesRemaining)/splitSize > SPLIT_SLOP) {
        String[][] splitHosts = getSplitHostsAndCachedHosts(blkLocations,
            length-bytesRemaining, splitSize, clusterMap);
        splits.add(makeSplit(path, length-bytesRemaining, splitSize,
            splitHosts[0], splitHosts[1]));
        bytesRemaining -= splitSize;
      }

      if (bytesRemaining != 0) {
        String[][] splitHosts = getSplitHostsAndCachedHosts(blkLocations, length
            - bytesRemaining, bytesRemaining, clusterMap);
        splits.add(makeSplit(path, length - bytesRemaining, bytesRemaining,
            splitHosts[0], splitHosts[1]));
      }
    } else {
      String[][] splitHosts = getSplitHostsAndCachedHosts(blkLocations,0,length,clusterMap);
      splits.add(makeSplit(path, 0, length, splitHosts[0], splitHosts[1]));
    }
  } else {
    //Create empty hosts array for zero length files
    splits.add(makeSplit(path, 0, length, new String[0]));
  }
}
sw.stop();
if (LOG.isDebugEnabled()) {
  LOG.debug("Total # of splits generated by getSplits: " + splits.size()
      + ", TimeTaken: " + sw.elapsedMillis());
}

return splits.toArray(new FileSplit[splits.size()]);
```

```
    }
```

7. Transformation

1）RDD中的所有转换都是**延迟加载**的，具有懒惰的属性。也就是说，它们并不会直接计算结果。相反的，它们只是记住这些应用到基础数据集（例如一个文件）上的转换动作。只有当发生一个要求返回结果给Driver的动作时，这些转换才会真正运行。这种设计让Spark更加有效率地运行。

常用的Transformation算子：

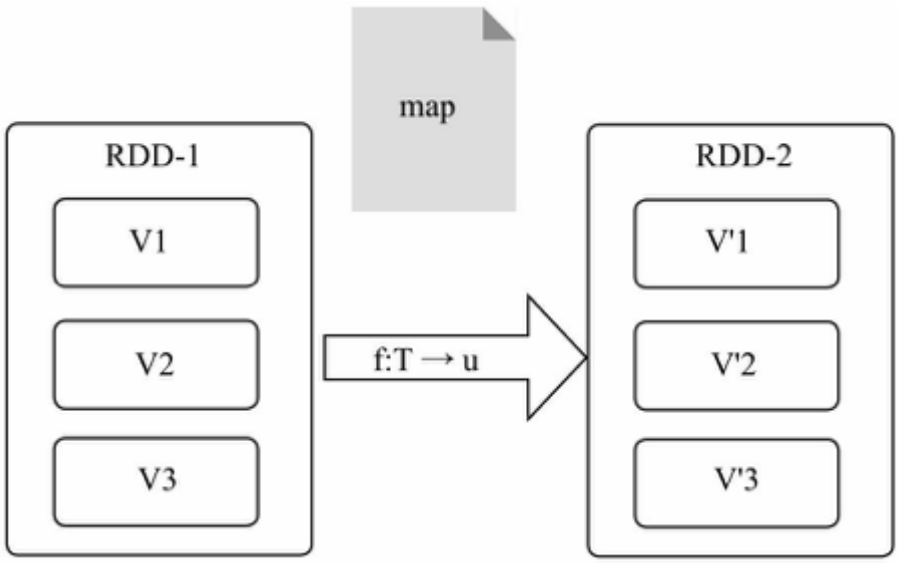
转换	含义
map(func)	返回一个新的RDD，该RDD由每一个输入元素经过func函数转换后组成
filter(func)	返回一个新的RDD，该RDD由经过func函数计算后返回值为true的输入元素组成
flatMap(func)	类似于map，但是每一个输入元素可以被映射为0或多个输出元素（所以func应该返回一个序列，而不是单一元素）
mapPartitions(func)	类似于map，但独立地在RDD的每一个分片上运行，因此在类型为T的RDD上运行时，func的函数类型必须是Iterator[T] => Iterator[U]
mapPartitionsWithIndex(func)	类似于mapPartitions，但func带有一个整数参数表示分片的索引值，因此在类型为T的RDD上运行时，func的函数类型必须是 (Int, Iterator[T]) => Iterator[U]
sample(withReplacement, fraction, seed)	根据fraction指定的比例对数据进行采样，可以选择是否使用随机数进行替换，seed用于指定随机数生成器种子
union(otherDataset)	对源RDD和参数RDD求并集后返回一个新的RDD
intersection(otherDataset)	对源RDD和参数RDD求交集后返回一个新的RDD
distinct([numTasks]))	对源RDD进行去重后返回一个新的RDD
groupByKey([numTasks])	在一个(K,V)的RDD上调用，返回一个(K, Iterator[V])的RDD
reduceByKey(func, [numTasks])	在一个(K,V)的RDD上调用，返回一个(K,V)的RDD，使用指定的reduce函数，将相同key的值聚合到一起，与groupByKey类似，reduce任务的个数可以通过第二个可选的参数来设置
aggregateByKey(zeroValue)(seqOp, combOp, [numTasks])	
sortByKey([ascending], [numTasks])	在一个(K,V)的RDD上调用，K必须实现Ordered接口，返回一个按照key进行排序的(K,V)的RDD
sortBy(func,[ascending], [numTasks])	与sortByKey类似，但是更灵活
join(otherDataset, [numTasks])	在类型为(K,V)和(K,W)的RDD上调用，返回一个相同key对应的所有元素对在一起的(K,(V,W))的RDD
cogroup(otherDataset, [numTasks])	在类型为(K,V)和(K,W)的RDD上调用，返回一个(K,(Iterable,Iterable))类型的RDD

转换	含义
cartesian(otherDataset)	笛卡尔积
coalesce(numPartitions)	重新分区
repartition(numPartitions)	重新分区
repartitionAndSortWithinPartitions(partitioner)	

5.1map

对RDD中的所有元素施加一个函数映射，返回一个新的RDD，该RDD由原RDD中的每个元素 经过function转换后组成

```
def map[U](f: (T) ⇒ U)(implicit arg0: ClassTag[U]): RDD[U]
```



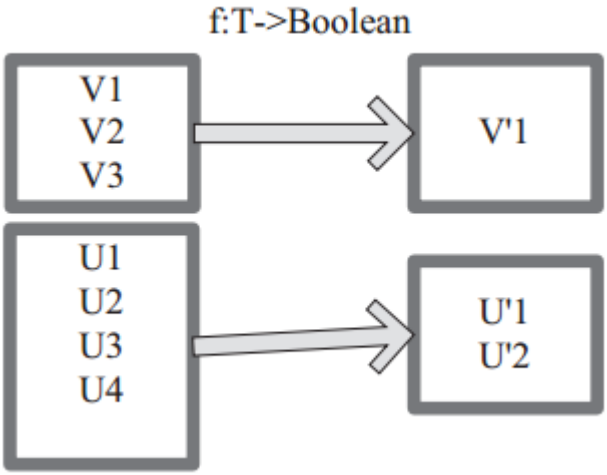
使用：

```
scala> val a = sc.parallelize(List("dog", "salmon", "salmon", "rat", "elephant"), 3)
a: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[0] at parallelize at <console>:27

scala> val b = a.map(_.length)
b: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[1] at map at <console>:29

scala> val c = a.zip(b)
c: org.apache.spark.rdd.RDD[(String, Int)] = ZippedPartitionsRDD2[2] at zip at <console>:31

scala> c.collect
res0: Array[(String, Int)] = Array((dog,3), (salmon,6), (salmon,6), (rat,3), (elephant,8))


//问题：不使用zip函数怎么解决？
```

5.2 filter

该RDD由原RDD中满足函数中的条件的元素组成

```
def filter(f: (T) ⇒ Boolean): RDD[T]
```



使用：

```
scala> val a = sc.parallelize(1 to 10, 3)
a: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[0] at parallelize at <console>:27

scala> val b = a.filter(_ % 2 == 0)
b: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[1] at filter at <console>:29

scala> b.collect
res0: Array[Int] = Array(2, 4, 6, 8, 10)
```

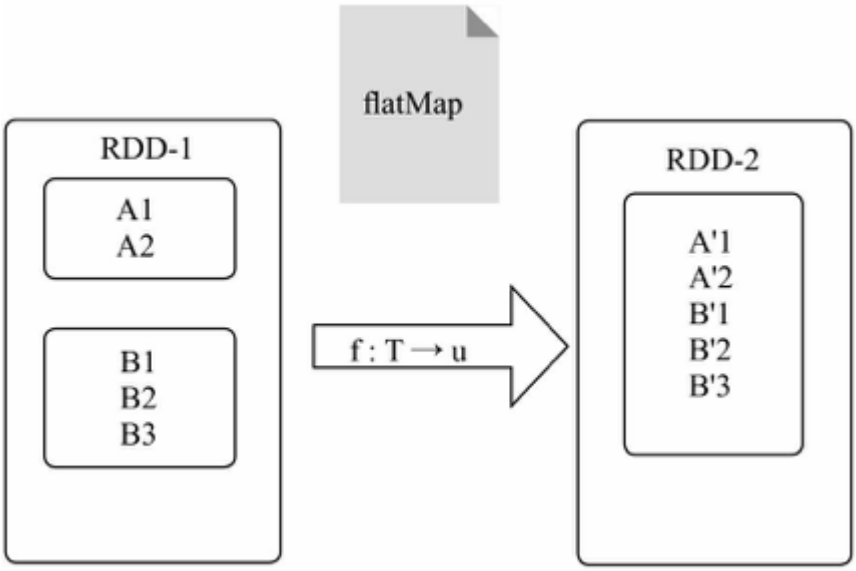
对比filterByRange [Ordered]（主要针对键值对）

```
scala> sc.parallelize(List((2,"cat"),(5,"dog"),(3,"monkey"),(6,"chick")))
res1: org.apache.spark.rdd.RDD[(Int, String)] = ParallelCollectionRDD[2] at parallelize at
<console>:28

scala> res1.filter
filter       filterWith

scala> res1.filterByRange(4,6)
res2: org.apache.spark.rdd.RDD[(Int, String)] = MapPartitionsRDD[3] at filterByRange at
<console>:30

scala> res2.collect
res3: Array[(Int, String)] = Array((5,dog), (6,chick))
```

5.3 flatMap

```
def flatMap[U](f: (T) ⇒ TraversableOnce[U])(implicit arg0: ClassTag[U]): RDD[U]
```

先map再flat



使用：

```
scala> val a = sc.parallelize(1 to 10, 5)
a: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[3] at parallelize at <console>:27

scala> a.flatMap(1 to _).collect
res1: Array[Int] = Array(1, 1, 2, 1, 2, 3, 1, 2, 3, 4, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 6, 1, 2, 3,
4, 5, 6, 7, 1, 2, 3, 4, 5, 6, 7, 8, 1, 2, 3, 4, 5, 6, 7, 8, 9, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

scala> sc.parallelize(List(1, 2, 3), 2).flatMap(x => List(x, x, x)).collect
res2: Array[Int] = Array(1, 1, 1, 2, 2, 2, 3, 3, 3)

scala> val x  = sc.parallelize(1 to 10, 3)
x: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[7] at parallelize at <console>:27

scala> x.flatMap(List.fill(scala.util.Random.nextInt(10))(_)).collect
res3: Array[Int] = Array(1, 2, 2, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 4, 5, 5, 5, 5, 5, 5, 5, 5, 6, 6,
6, 6, 6, 6, 7, 7, 9, 9, 9, 10, 10, 10, 10, 10)

scala> val rdd0 = sc.parallelize(List(List("a b c","d e f"),List("h i j ","k l m n ")))
rdd0: org.apache.spark.rdd.RDD[List[String]] = ParallelCollectionRDD[0] at parallelize at
<console>:2
//思考：两个flatMap的异同
scala> val rdd1 =rdd0.flatMap(_.flatMap(_.split(" ")))
rdd1: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[1] at flatMap at <console>:29

scala> val res= rdd1.collect
res: Array[String] = Array(a, b, c, d, e, f, h, i, j, k, l, m, n)
```

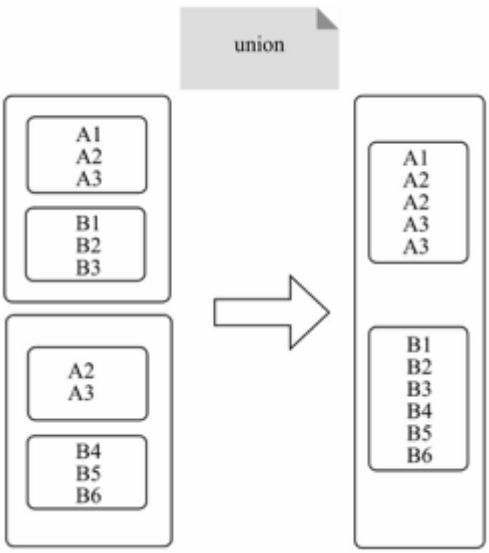
扩展：flatMapValues

```
scala> sc.parallelize(List((2,"a d"),(5,"e f")))
res5: org.apache.spark.rdd.RDD[(Int, String)] = ParallelCollectionRDD[5] at parallelize at
<console>:2

scala> res5.flatMapValues(_.split(" "))
res6: org.apache.spark.rdd.RDD[(Int, String)] = MapPartitionsRDD[6] at flatMapValues at
<console>:30

scala> res6.collect
res7: Array[(Int, String)] = Array((2,a), (2,d), (5,e), (5,f))
```

5.4 union, ++

```
def ++(other: RDD[T]): RDD[T]
def union(other: RDD[T]): RDD[T]
```

合并同一数据类型元素，但不去重。



使用

```
scala> val a = sc.parallelize(1 to 3,1)
a: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[0] at parallelize at <console>:27

scala> val b = sc.parallelize(5 to 7,1)
b: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[1] at parallelize at <console>:27

scala> (a++b).collect
res0: Array[Int] = Array(1, 2, 3, 5, 6, 7)
```

5.5 intersection

```
def intersection(other: RDD[T], numPartitions: Int): RDD[T]
```

返回两个差集，不含重复元素。

```
scala> val x = sc.parallelize(1 to 20)
x: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[3] at parallelize at <console>:27

scala> val y = sc.parallelize(10 to 30)
y: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[4] at parallelize at <console>:27

scala> val z = x.intersection(y)
z: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[10] at intersection at <console>:31

scala>

scala> z.collect
res1: Array[Int] = Array(16, 14, 12, 18, 20, 10, 13, 19, 15, 11, 17)
```

5.6 distinct

```
def distinct(): RDD[T]
def distinct(numPartitions: Int): RDD[T]
```

去重元素之后的RDD

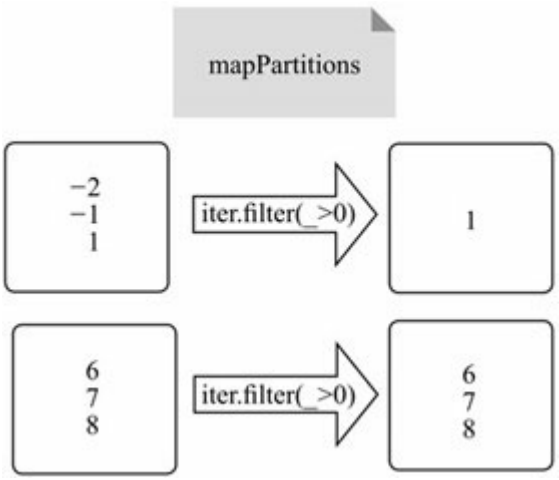
```
scala> val c = sc.parallelize(List("Gnu", "Cat", "Rat", "Dog", "Gnu", "Rat"), 2)
c: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[0] at parallelize at <console>:27

scala> c.distinct.collect
res0: Array[String] = Array(Dog, Cat, Gnu, Rat)

scala> val a = sc.parallelize(List(1,2,3,4,5,6,7,8,9,10))
a: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[5] at parallelize at <console>:27

scala> a.distinct(2).partitions.length
res3: Int = 2
```

```
scala> a.distinct(3).partitions.length
res4: Int = 3
```

5.7 mapPartitions

```
def mapPartitions[U](f: (Iterator[T]) ⇒ Iterator[U], preservesPartitioning: Boolean = false)
(implicit arg0: ClassTag[U]): RDD[U]
```

mapPartitions是map的一个变种。map的输入函数应用于RDD中的每个元素，而mapPartitions的输入函数应用于每个分区，也就是把每个分区中的内容作为整体来处理的。



使用：

```
scala> val rdd = sc.parallelize(List("dog","cat","chick","monkey"))
rdd: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[0] at parallelize at <console>:2

scala> rdd.mapPartitions(iter=>iter.filter(_.length>3))
res0: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[1] at mapPartitions at <console>:30

scala> res0.collect
res1: Array[String] = Array(chick, monkey)
```

5.8 mapPartitionsWithIndex

```
def mapPartitionsWithIndex[U](f: (Int, Iterator[T]) ⇒ Iterator[U], preservesPartitioning:
Boolean = false)(implicit arg0: ClassTag[U]): RDD[U]
```

map函数同样作用于整个分区，同时可以获取分区的index信息。

使用：

```
scala> val x = sc.parallelize(List(1,2,3,4,5,6,7,8,9,10), 3)
x: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[13] at parallelize at <console>:27

scala> def myfunc(index: Int, iter: Iterator[Int]) : Iterator[String] = {
     |   iter.map(x => index + ":" + x)
     | }
myfunc: (index: Int, iter: Iterator[Int])Iterator[String]

scala> x.mapPartitionsWithIndex(myfunc).collect()
res6: Array[String] = Array(0:1, 0:2, 0:3, 1:4, 1:5, 1:6, 2:7, 2:8, 2:9, 2:10)
```

5.9 比较map flatMap mapPartitions mapPartitionsWithIndex

Spark中，最基本的原则，就是每个task处理一个RDD的partition。

MapPartitions操作的优点：

如果是普通的map，比如一个partition中有1万条数据；ok，那么你的function要执行和计算1万次。

但是，使用MapPartitions操作之后，一个task仅仅会执行一次function，function一次接收所有 的partition数据。只要执行一次就可以了，性能比较高。

MapPartitions的缺点：可能会OOM。

如果是普通的map操作，一次function的执行就处理一条数据；那么如果内存不够用的情况下， 比如处理了1千条数据了，那么这个时候内存不够了，那么就可以将已经处理完的1千条数据从 内存里面垃圾回收掉，或者用其他方法，腾出空间来吧。

所以说普通的map操作通常不会导致内存的OOM异常。

在项目中，自己先去估算一下RDD的数据量，以及每个partition的量，还有自己分配给每个executor 的内存资源。看看一下子内存容纳所有的partition数据，行不行。如果行，可以试一下，能跑通就好。 性能肯定是有提升的。

```
//map和partition的区别：
scala> val rdd2 = rdd1.mapPartitions(_.map(_*10))
rdd2: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[1] ...

scala> rdd2.collect
res1: Array[Int] = Array(10, 20, 30, 40, 50, 60, 70)

scala> rdd1.map(_ * 10).collect
res3: Array[Int] = Array(10, 20, 30, 40, 50, 60, 70)

介绍mapPartition和map的区别，引出下面的内容：

mapPartitionsWithIndex
val func = (index: Int, iter: Iterator[(Int)]) => {
  iter.toList.map(x => "[partID:" +  index + ", val: " + x + "]").iterator
}
val rdd1 = sc.parallelize(List(1,2,3,4,5,6,7,8,9), 2)

rdd1.mapPartitionsWithIndex(func).collect
```

5.10 sample

```
def sample(withReplacement: Boolean, fraction: Double, seed: Long = Utils.random.nextLong):
RDD[T]
```

返回抽样得到的子集。

withReplacement为true时表示抽样之后还放回，可以被多次抽样，false表示不放回；fraction表示抽样比例；seed为随机数种子，比如当前时间戳

使用：

```
scala> val a = sc.parallelize(1 to 10000, 3)
a: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[0] at parallelize at <console>:27

scala> a.sample(false, 0.1, 0).count
res0: Long = 1032
```

5.11 keyBy

```
def keyBy[K](f: T => K): RDD[(K, T)]
```

通过在每个元素上应用一个函数生成键值对中的键，最后构成一个键值对元组。

使用：

```
scala> val a = sc.parallelize(List("dog", "salmon", "salmon", "rat", "elephant"), 3)
a: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[35] at parallelize at <console>:27

scala> val b = a.keyBy(_.length)
b: org.apache.spark.rdd.RDD[(Int, String)] = MapPartitionsRDD[36] at keyBy at <console>:29

scala> b.collect
res9: Array[(Int, String)] = Array((3,dog), (6,salmon), (6,salmon), (3,rat), (8,elephant))
```

5.12 mapValues

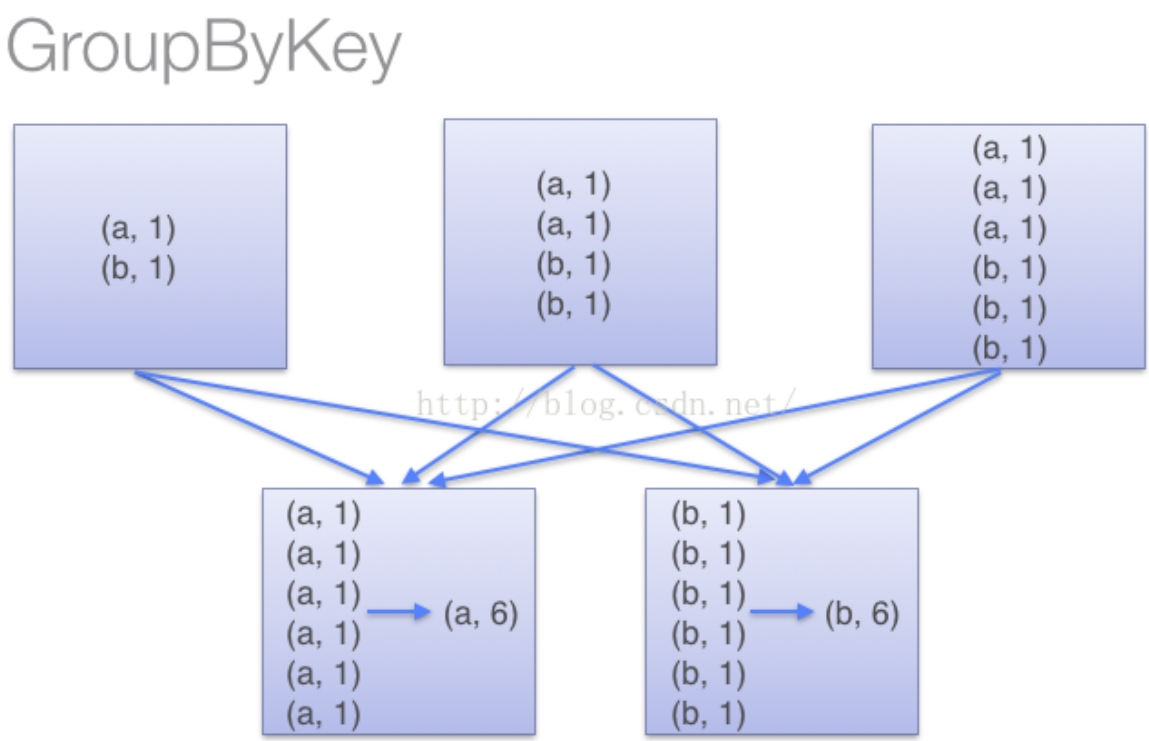
mapValues ：针对（Key， Value）型数据中的 Value 进行 Map 操作，而不对 Key 进行处理。

```
scala> val a = sc.parallelize(List("dog", "tiger", "lion", "cat", "panther", "eagle"), 2)
a: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[2] at parallelize at <console>:27

scala> val b = a.map(x => (x.length, x))
b: org.apache.spark.rdd.RDD[(Int, String)] = MapPartitionsRDD[3] at map at <console>:29

scala> b.mapValues("x" + _ + "x").collect
res2: Array[(Int, String)] = Array((3,xdogx), (5,xtigerx), (4,xlionx), (3,xcatx), (7,xpantherx),
(5,xeaglex))
```

5.13 groupByKey [Pair]

```
def groupByKey(): RDD[(K, Iterable[V])]
val words = Array("one", "two", "two", "three", "three", "three")
val wordPairsRDD = sc.parallelize(words).map(word => (word, 1))
val wordCountsWithGroup = wordPairsRDD .groupByKey() .map(t => (t._1, t._2.sum)) .collect()
```

按照key值进行分组



注意1：每次结果可能不同，因为发生shuffle

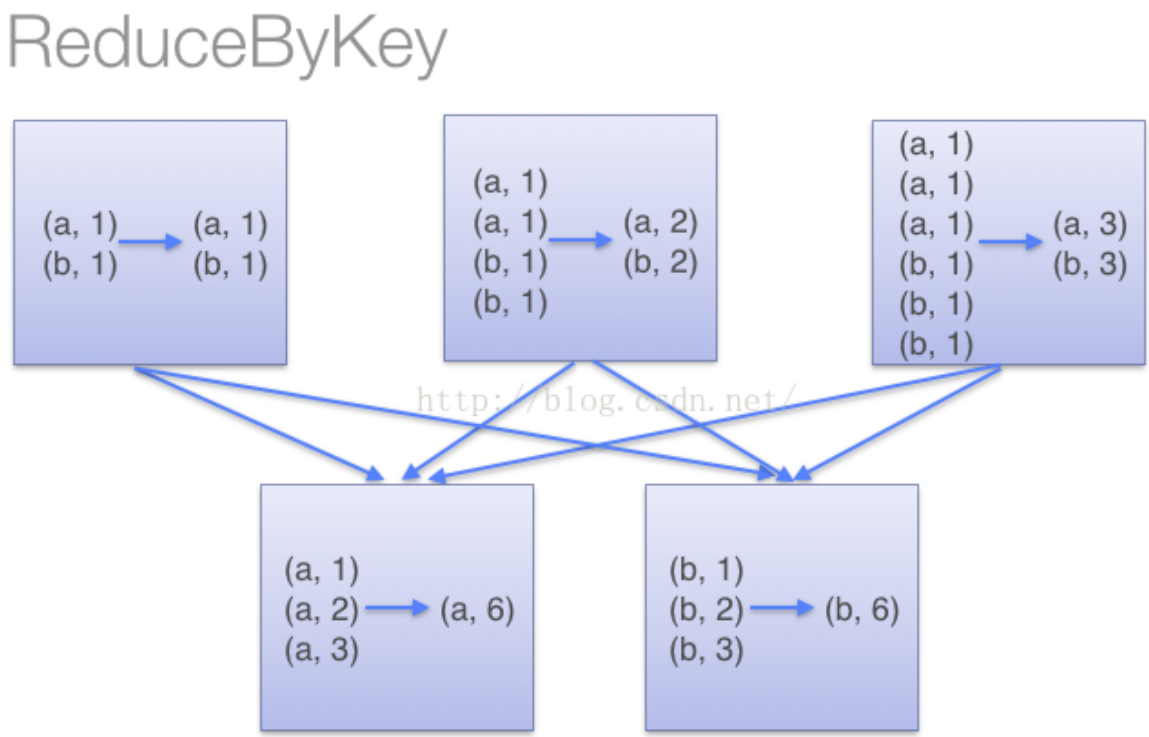
使用：

```
scala> val a = sc.parallelize(List("dog", "tiger", "lion", "cat", "spider", "eagle"), 2)
a: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[12] at parallelize at <console>:27

scala> val b = a.keyBy(_.length)
b: org.apache.spark.rdd.RDD[(Int, String)] = MapPartitionsRDD[13] at keyBy at <console>:29

scala> b.groupByKey.collect
res5: Array[(Int, Iterable[String])] = Array((4,CompactBuffer(lion)), (6,CompactBuffer(spider)),
(3,CompactBuffer(dog, cat)), (5,CompactBuffer(tiger, eagle)))
```

5.14 reduceByKey

```
def reduceByKey(func: (V, V) => V): RDD[(K, V)]
def groupByKey(): RDD[(K, Iterable[V])]
val words = Array("one", "two", "two", "three", "three", "three")
val wordPairsRDD = sc.parallelize(words).map(word => (word, 1))

val wordCountsWithReduce = wordPairsRDD .reduceByKey(_ + _) .collect()
```

对元素为KV对的RDD中Key相同的元素的Value进行reduce操作



使用：

```
scala> val a = sc.parallelize(List("dog", "cat", "owl", "gnu", "ant"), 2)
a: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[0] at parallelize at <console>:27

scala> val b = a.map(x => (x.length, x))
b: org.apache.spark.rdd.RDD[(Int, String)] = MapPartitionsRDD[1] at map at <console>:29

scala> b.reduceByKey(_ + _).collect
res0: Array[(Int, String)] = Array((3,dogcatowlgnuant))
```

5.15 foldByKey

foldByKey merges the values for each key using an associative function and a neutral "zero value".

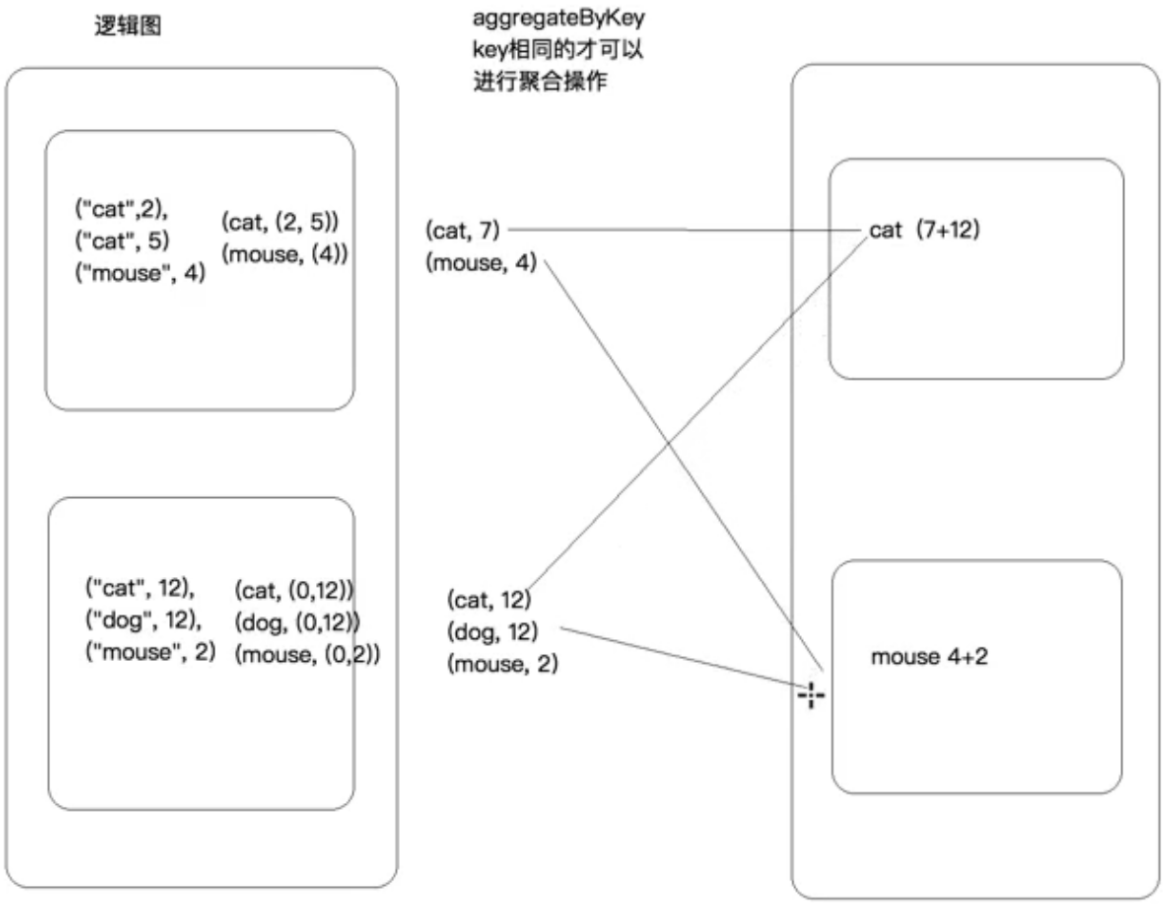
针对键值对的RDD进行聚合

```
scala> val a = sc.parallelize(List("dog", "cat", "owl", "gnu", "ant"), 2)
a: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[7] at parallelize at <console>:27

scala> val b = a.map(x => (x.length, x))
b: org.apache.spark.rdd.RDD[(Int, String)] = MapPartitionsRDD[8] at map at <console>:29

scala> b.foldByKey("")(_ + _).collect
res8: Array[(Int, String)] = Array((3,dogcatowlgnuant))

scala> b.foldByKey("QQ")(_ + _).collec
```

5.16 aggregateByKey

combineByKey can be used when you are combining elements but your return type differs from your input value type.

```
def aggregateByKeyU(seqOp: (U, V) ⇒ U, combOp: (U, U) ⇒ U)(implicit arg0: ClassTag[U]): RDD[(K,
U)]

def aggregateByKeyU(seqOp: (U, V) ⇒ U, combOp: (U, U) ⇒ U)(implicit arg0: ClassTag[U]): RDD[(K,
U)]

def aggregateByKeyU(seqOp: (U, V) ⇒ U, combOp: (U, U) ⇒ U)(implicit arg0: ClassTag[U]): RDD[(K,
U)]
```



使用：

```
val pairRDD = sc.parallelize(List( ("cat",2), ("cat", 5), ("mouse", 4),("cat", 12), ("dog", 12),
("mouse", 2)), 2)

// lets have a look at what is in the partitions
def myfunc(index: Int, iter: Iterator[(String, Int)]) : Iterator[String] = {
  iter.map(x => "[partID:" +  index + ", val: " + x + "]")
}
pairRDD.mapPartitionsWithIndex(myfunc).collect

res2: Array[String] = Array([partID:0, val: (cat,2)], [partID:0, val: (cat,5)], [partID:0, val:
(mouse,4)], [partID:1, val: (cat,12)], [partID:1, val: (dog,12)], [partID:1, val: (mouse,2)])

pairRDD.aggregateByKey(0)(math.max(_, _), _ + _).collect
res3: Array[(String, Int)] = Array((dog,12), (cat,17), (mouse,6))

pairRDD.aggregateByKey(100)(math.max(_, _), _ + _).collect
res4: Array[(String, Int)] = Array((dog,100), (cat,200), (mouse,200))
```

```
val pairRDD = sc.parallelize(List( ("cat",2), ("cat", 5), ("mouse", 4),("cat", 12), ("dog", 12),
("mouse", 2)), 2)
需求：统计猫狗耗子各有多少只？ 比较两种方法
pairRDD.aggregateByKey(0)(_+_,_+_).collect
pairRDD.reduceByKey(_+_).collect --也能实现

需求：把每个分区每种最多的动物取出来再进行对应的相加
def func2(index: Int, iter: Iterator[(String, Int)]) : Iterator[String] = {
  iter.toList.map(x => "[partID:" +  index + ", val: " + x + "]").iterator
}
pairRDD.mapPartitionsWithIndex(func2).collect
pairRDD.aggregateByKey(0)(math.max(_, _), _ + _).collect
pairRDD.aggregateByKey(100)(math.max(_, _), _ + _).collect
```

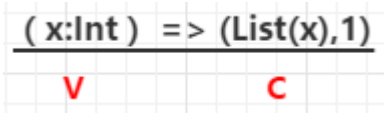
5.17 combineByKey

comineByKey(createCombiner,mergeValue,mergeCombiners,partitioner,mapSideCombine)

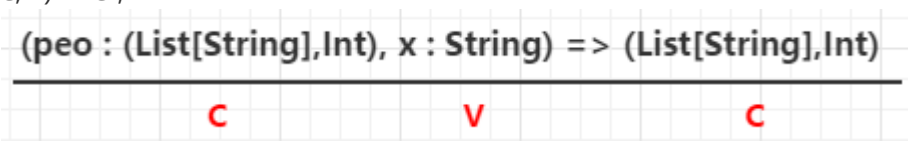
comineByKey(createCombiner,mergeValue,mergeCombiners,numPartitions)

comineByKey(createCombiner,mergeValue,mergeCombiners)

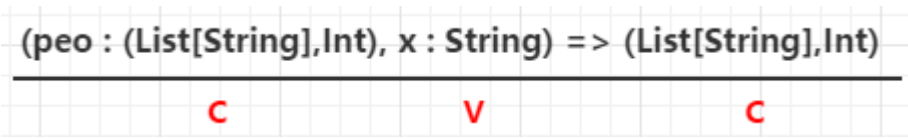
createCombiner:在第一次遇到Key时创建组合器函数，将RDD数据集中的V类型值转换C类型值（V => C），



mergeValue：合并值函数，再次遇到相同的Key时，将createCombiner道理的C类型值与这次传入的V类型值合并成一个C类型值（C,V）=>C，



mergeCombiners:合并组合器函数，将C类型值两两合并成一个C类型值



partitioner：使用已有的或自定义的分区函数，默认是HashPartitioner

mapSideCombine：是否在map端进行Combine操作,默认为true

注意前三个函数的参数类型要对应；第一次遇到Key时调用createCombiner，再次遇到相同的Key时调用mergeValue合并值

（例子）求各科平均成绩

```
scala> sc.setLogLevel("WARN")

scala> val inputrdd = sc.parallelize(Seq(
                        ("maths", 50), ("maths", 60),
                        ("english", 65),
                        ("physics", 66), ("physics", 61), ("physics", 87)),
```

```
                        1)
inputrdd: org.apache.spark.rdd.RDD[(String, Int)] = ParallelCollectionRDD[41] at parallelize at
<console>:27

scala> inputrdd.getNumPartitions
res55: Int = 1

scala> val reduced = inputrdd.combineByKey(
         (mark) => {
           println(s"Create combiner -> ${mark}")
           (mark, 1)
         },
         (acc: (Int, Int), v) => {
           println(s"""Merge value : (${acc._1} + ${v}, ${acc._2} + 1)""")
           (acc._1 + v, acc._2 + 1)
         },
         (acc1: (Int, Int), acc2: (Int, Int)) => {
           println(s"""Merge Combiner : (${acc1._1} + ${acc2._1}, ${acc1._2} + ${acc2._2})""")
           (acc1._1 + acc2._1, acc1._2 + acc2._2)
         }
     )
reduced: org.apache.spark.rdd.RDD[(String, (Int, Int))] = ShuffledRDD[42] at combineByKey at
<console>:29

scala> reduced.collect()
Create combiner -> 50
Merge value : (50 + 60, 1 + 1)
Create combiner -> 65
Create combiner -> 66
Merge value : (66 + 61, 1 + 1)
Merge value : (127 + 87, 2 + 1)
res56: Array[(String, (Int, Int))] = Array((maths,(110,2)), (physics,(214,3)), (english,(65,1)))

scala> val result = reduced.mapValues(x => x._1 / x._2.toFloat)
result: org.apache.spark.rdd.RDD[(String, Float)] = MapPartitionsRDD[43] at mapValues at
<console>:31

scala> result.collect()
res57: Array[(String, Float)] = Array((maths,55.0), (physics,71.333336), (english,65.0))
```

（例3）：统计男性和女生的个数，并以（性别，（名字，名字....），个数）的形式输出

```
object CombineByKey {

  def main(args: Array[String]) {

    val conf = new SparkConf().setMaster("local").setAppName("combinByKey")


    val sc = new SparkContext(conf)
```

```
    val people = List(("male", "Mobin"), ("male", "Kpop"), ("female", "Lucy"), ("male",
"Lufei"), ("female", "Amy"))

    val rdd = sc.parallelize(people)

    val combinByKeyRDD = rdd.combineByKey(

      (x: String) => (List(x), 1),

      (peo: (List[String], Int), x : String) => (x :: peo.1, peo.2 + 1),

      (sex1: (List[String], Int), sex2: (List[String], Int)) => (sex1.1 ::: sex2.1, sex1.2 +
sex2.2))

    combinByKeyRDD.foreach(println)

    sc.stop()

  }

}
```

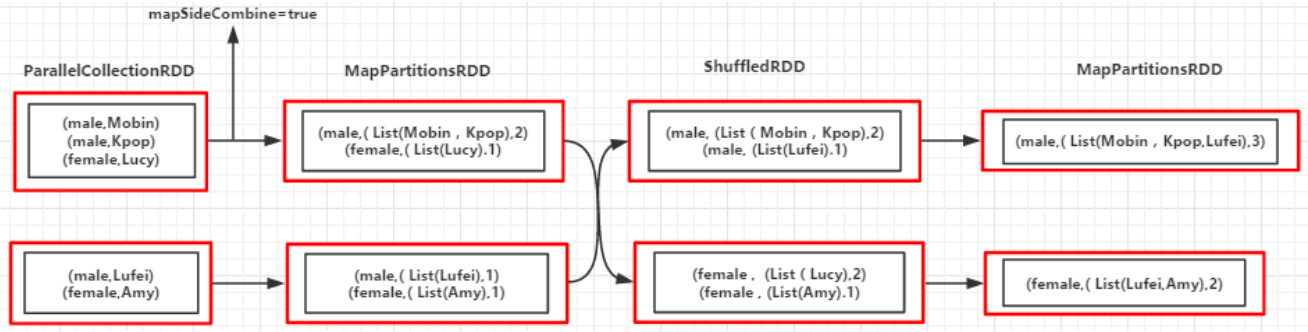
输出：

```
(male,(List(Lufei, Kpop, Mobin),3))
(female,(List(Amy, Lucy),2))
```

过程分解：

```
Partition1:

K="male"  -->  ("male","Mobin")  --> createCombiner("Mobin") =>  peo1 = (  List("Mobin") , 1 )

K="male"  -->  ("male","Kpop")  --> mergeValue(peo1,"Kpop") =>  peo2 = (  "Kpop"  ::  peo1_1 , 1
+ 1 )    //Key相同调用mergeValue函数对值进行合并

K="female"  -->  ("female","Lucy")  --> createCombiner("Lucy") =>  peo3 = (  List("Lucy") , 1 )

Partition2:

K="male"  -->  ("male","Lufei")  --> createCombiner("Lufei") =>  peo4 = (  List("Lufei") , 1 )

K="female"  -->  ("female","Amy")  --> createCombiner("Amy") =>  peo5 = (  List("Amy") , 1 )

Merger Partition:

K="male" --> mergeCombiners(peo2,peo4) => (List(Lufei,Kpop,Mobin))
```

```
K="female" --> mergeCombiners(peo3,peo5) => (List(Amy,Lucy))
```



（RDD依赖图）

5.01 比较reduceByKey和groupByKey，aggregateByKey

reduceByKey(func, numPartitions=None)

Merge the values for each key using an associative 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 numPartitions partitions, or the default parallelism level if numPartitions is not specified.

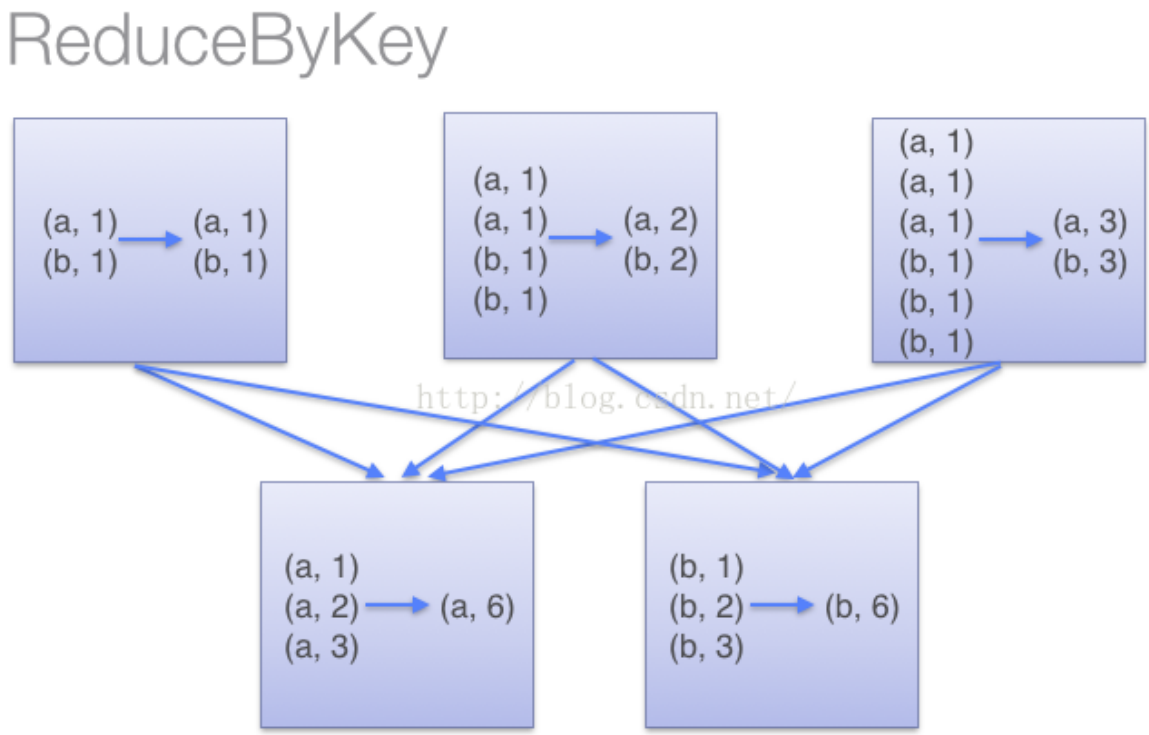
也就是，reduceByKey用于对每个key对应的多个value进行merge操作，最重要的是它能够在本地先进行merge操作，并且merge操作可以通过函数自定义。

groupByKey(numPartitions=None)

Group the values for each key in the RDD into a single sequence. Hash-partitions the resulting RDD with numPartitions partitions.** Note**: If you are grouping in order to perform an aggregation (such as a sum or average) over each key, using reduceByKey or aggregateByKey will provide much better performance.

也就是，groupByKey也是对每个key进行操作，但只生成一个sequence。需要特别注意“Note”中的话，它告诉我们：如果需要对sequence进行aggregation操作（注意，groupByKey本身不能自定义操作函数），那么，选择reduceByKey/aggregateByKey更好。这是因为groupByKey不能自定义函数，我们需要先用groupByKey生成RDD，然后才能对此RDD通过map进行自定义函数操作。

```
val words = Array("one", "two", "two", "three", "three", "three")
val wordPairsRDD = sc.parallelize(words).map(word => (word, 1))
val wordCountsWithReduce = wordPairsRDD.reduceByKey(_ + _)
val wordCountsWithGroup = wordPairsRDD.groupByKey().map(t => (t._1, t._2.sum))
```



reduceByKey 先在分区上进行合并，然后shuffle，最终得到一个结果。

5.18 sortByKey

```
def sortByKey(ascending: Boolean = true, numPartitions: Int = self.partitions.size): RDD[P]
```

使用：

```
scala> val a = sc.parallelize(List("dog", "cat", "owl", "gnu", "ant"), 2)
a: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[0] at parallelize at <console>:27

scala> val b = sc.parallelize(1 to a.count.toInt, 2)
b: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[1] at parallelize at <console>:29

scala> val c = a.zip(b)
c: org.apache.spark.rdd.RDD[(String, Int)] = ZippedPartitionsRDD2[2] at zip at <console>:31

scala> c.sortByKey(true).collect
res0: Array[(String, Int)] = Array((ant,5), (cat,2), (dog,1), (gnu,4), (owl,3))

scala> c.sortByKey(false).collect
res1: Array[(String, Int)] = Array((owl,3), (gnu,4), (dog,1), (cat,2), (ant,5))
```

5.19 sortBy

```
def sortBy[K](f: (T) ⇒ K, ascending: Boolean = true, numPartitions: Int = this.partitions.size)
(implicit ord: Ordering[K], ctag: ClassTag[K]): RDD[T]
```

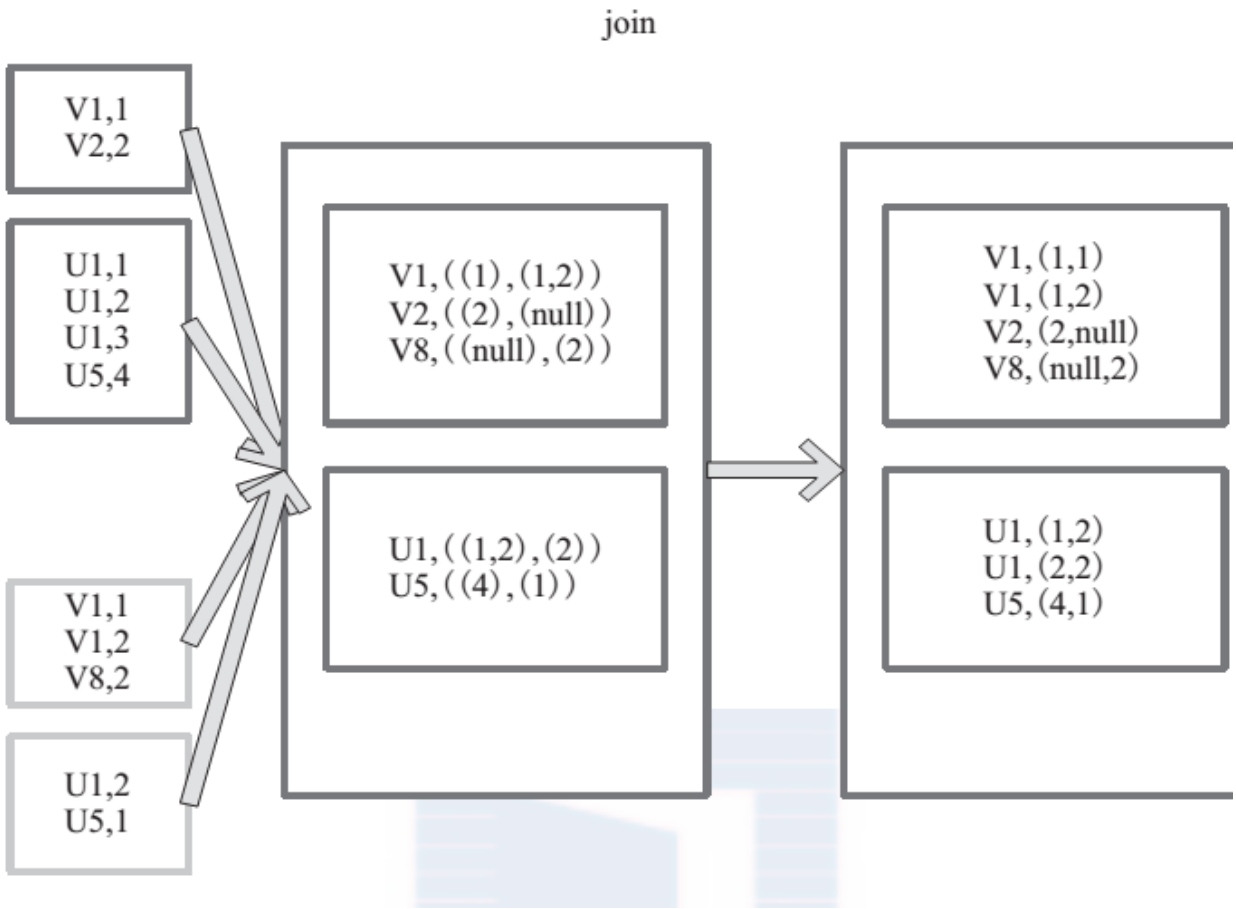
This function sorts the input RDD's data and stores it in a new RDD. The first parameter requires you to specify a function which maps the input data into the key that you want to sortBy.

使用：

```
scala> val y = sc.parallelize(Array(5, 7, 1, 3, 2, 1))
y: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[9] at parallelize at <console>:27

scala> y.sortBy(c => c, true).collect
res2: Array[Int] = Array(1, 1, 2, 3, 5, 7)

scala> y.sortBy(c => c, false).collect
res3: Array[Int] = Array(7, 5, 3, 2, 1, 1)

scala> val z = sc.parallelize(Array(("H", 10), ("A", 26), ("Z", 1), ("L", 5)))
z: org.apache.spark.rdd.RDD[(String, Int)] = ParallelCollectionRDD[22] at parallelize at
<console>:27

scala> z.sortBy(c => c._1, true).collect
res7: Array[(String, Int)] = Array((A,26), (H,10), (L,5), (Z,1))
```

5.7 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))]。



join算子（join，leftOuterJoin，rightOuterJoin）

1)只能通过PairRDD使用；

2) join算子操作的Tuple2<Object1, Object2>类型中，Object1是连接键;

join(otherDataset, [numTasks])是连接操作，将输入数据集(K,V)和另外一个数据集(K,W)进行Join， 得到(K,(V,W))；该操作是对于相同K的V和W集合进行笛卡尔积 操作，也即V和W的所有组合；

```
scala> val rdd1 = sc.parallelize(List(("Tom",1),("Jerry",2),("Kate",3)))
rdd1: org.apache.spark.rdd.RDD[(String, Int)] = ParallelCollectionRDD[2] at parallelize at
<console>:27

scala> val rdd2 = sc.parallelize(List(("Jerry",4),("shuke",5),("Tom",6)))
rdd2: org.apache.spark.rdd.RDD[(String, Int)] = ParallelCollectionRDD[3] at parallelize at
<console>:27

scala> val rdd3 = rdd1.join(rdd2)
rdd3: org.apache.spark.rdd.RDD[(String, (Int, Int))] = MapPartitionsRDD[6] at join at
<console>:31

scala> rdd3.collect
res0: Array[(String, (Int, Int))] = Array((Tom,(1,6)), (Jerry,(2,4)))

scala> val rdd4 = rdd1.leftOuterJoin(rdd2)

rdd4: org.apache.spark.rdd.RDD[(String, (Int, Option[Int]))] = MapPartitionsRDD[9] at
```

```
leftOuterJoin at <console>:31

scala> rdd4.collect
res1: Array[(String, (Int, Option[Int]))] = Array((Tom,(1,Some(6))), (Jerry,(2,Some(4))), (Kate,
(3,None)))

scala> val rdd5 = rdd1.rightOuterJoin(rdd2)
rdd5: org.apache.spark.rdd.RDD[(String, (Option[Int], Int))] = MapPartitionsRDD[12] at
rightOuterJoin at <console>:31

scala> rdd5.collect
res2: Array[(String, (Option[Int], Int))] = Array((Tom,(Some(1),6)), (Jerry,(Some(2),4)),
(shuke,(None,5)))
```

```
def join[W](other: RDD[(K, W)]): RDD[(K, (V, W))]
def join[W](other: RDD[(K, W)], numPartitions: Int): RDD[(K, (V, W))]
def join[W](other: RDD[(K, W)], partitioner: Partitioner): RDD[(K, (V, W))]
```

使用：

```
scala> val a = sc.parallelize(List("dog", "salmon", "salmon", "rat", "elephant"), 3)
a: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[28] at parallelize at <console>:27

scala> val b = a.keyBy(_.length)
b: org.apache.spark.rdd.RDD[(Int, String)] = MapPartitionsRDD[29] at keyBy at <console>:29

scala> val c =
sc.parallelize(List("dog","cat","gnu","salmon","rabbit","turkey","wolf","bear","bee"), 3)
c: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[30] at parallelize at <console>:27

scala> val d = c.keyBy(_.length)
d: org.apache.spark.rdd.RDD[(Int, String)] = MapPartitionsRDD[31] at keyBy at <console>:29

scala> b.join(d).collect
res8: Array[(Int, (String, String))] = Array((6,(salmon,salmon)), (6,(salmon,rabbit)), (6,
(salmon,turkey)), (6,(salmon,salmon)), (6,(salmon,rabbit)), (6,(salmon,turkey)), (3,(dog,dog)),
(3,(dog,cat)), (3,(dog,gnu)), (3,(dog,bee)), (3,(rat,dog)), (3,(rat,cat)), (3,(rat,gnu)), (3,
(rat,bee)))
```

5.20 repartition

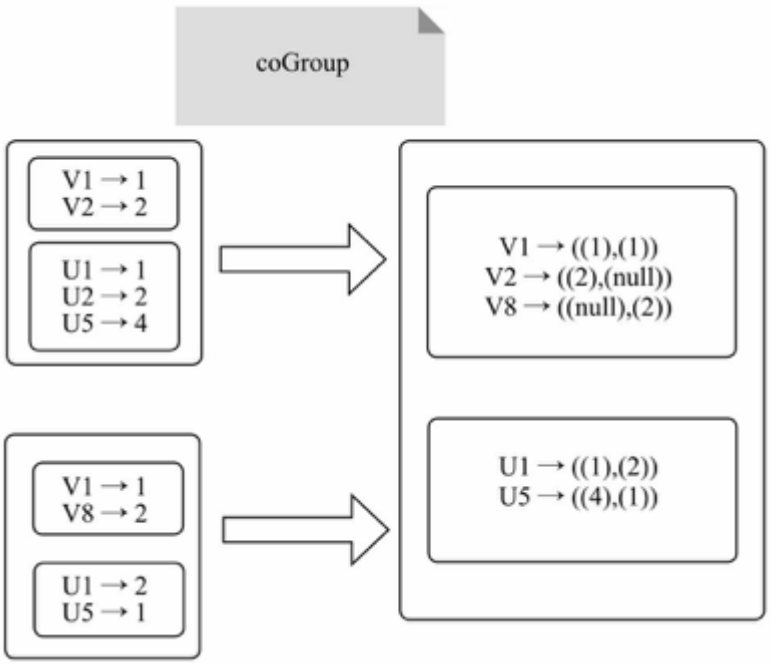
```
def repartition(numPartitions: Int)(implicit ord: Ordering[T] = null): RDD[T]
```

This function changes the number of partitions to the number specified by the numPartitions parameter

```
scala> val rdd = sc.parallelize(List(1, 2, 10, 4, 5, 2, 1, 1, 1), 3)
rdd: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[37] at parallelize at <console>:27

scala> rdd.partitions.length
res10: Int = 3

scala> val rdd2 = rdd.repartition(5)
rdd2: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[41] at repartition at <console>:29

scala> rdd2.partitions.length
res11: Int = 5
```

5.21 cogroup

可以对多达3个RDD根据key进行分组，将每个Key相同的元素分别聚集为一个集合



```
scala> val a = sc.parallelize(List(1, 2, 1, 3), 1)
a: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[0] at parallelize at <console>:27

scala> val b = a.map((_, "b"))
b: org.apache.spark.rdd.RDD[(Int, String)] = MapPartitionsRDD[1] at map at <console>:29

scala> val c = a.map((_, "c"))
c: org.apache.spark.rdd.RDD[(Int, String)] = MapPartitionsRDD[2] at map at <console>:29

scala> b.cogroup(c).collect
res0: Array[(Int, (Iterable[String], Iterable[String]))] = Array((1,(CompactBuffer(b,
b),CompactBuffer(c, c))), (3,(CompactBuffer(b),CompactBuffer(c))), (2,
(CompactBuffer(b),CompactBuffer(c))))
```

5.02 比较coalesce, repartition

这两个算子都是用来调整分区个数的，其中repartition等价于 coalesce(numPartitions, shuffle = true).

```
def coalesce ( numPartitions : Int , shuffle : Boolean = false ): RDD [T]
def repartition ( numPartitions : Int ): RDD [T]
```

使用

```
scala> val rdd1 = sc.parallelize(1 to 10,10)
rdd1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[0] at parallelize at <console>:27

scala> rdd1.partitions.length
res0: Int = 10

scala> val rdd2 = rdd1.coalesce(3,false)
rdd2: org.apache.spark.rdd.RDD[Int] = CoalescedRDD[1] at coalesce at <console>:29

scala> rdd2.partitions.length
res1: Int = 3
```

```
repartition（重新分配分区），coalesce（（合并）重新分配分区并设置是否shuffle），
        partitionBy(根据partitioner函数生成新的ShuffleRDD，将原RDD重新分区)
val rdd1 = sc.parallelize(1 to 10, 10)
rdd1.repartition(5) --分区调整为5个
rdd1.partitions.length =5
coalesce：调整分区数量，参数一:要合并成几个分区，参数二：是否shuffle，false不会shuffle
val rdd2 = rdd1.coalesce(2, false)
val rdd1 = sc.parallelize(List(("jerry", 2), ("tom", 1), ("shuke", 2)), 3)
var rdd2 = rdd1.partitionBy(new org.apache.spark.HashPartitioner(2))
rdd2.partitions.length
```

8，Action算子

触发Spark作业的运行，真正触发转换算子的计算

常用的Action算子：

动作	含义
reduce(*func*)	通过func函数聚集RDD中的所有元素，这个功能必须是可交换且可并联的
collect()	在驱动程序中，以数组的形式返回数据集的所有元素
count()	返回RDD的元素个数
first()	返回RDD的第一个元素（类似于take(1)）
take(*n*)	返回一个由数据集的前n个元素组成的数组
takeSample(*withReplacement*,*num*, [*seed*])	返回一个数组，该数组由从数据集中随机采样的num个元素组成，可以选择是否用随机数替换不足的部分，seed用于指定随机数生成器种子
takeOrdered(*n*, *[ordering]*)	takeOrdered和top类似，只不过以和top相反的顺序返回元素
saveAsTextFile(*path*)	将数据集的元素以textfile的形式保存到HDFS文件系统或者其他支持的文件系统，对于每个元素，Spark将会调用toString方法，将它装换为文件中的文本
saveAsSequenceFile(*path*)	将数据集中的元素以Hadoop sequencefile的格式保存到指定的目录下，可以使HDFS或者其他Hadoop支持的文件系统。
saveAsObjectFile(*path*)	
countByKey()	针对(K,V)类型的RDD，返回一个(K,Int)的map，表示每一个key对应的元素个数。
foreach(*func*)	在数据集的每一个元素上，运行函数func进行更新。

6.1 reduce

```
def reduce(f: (T, T) => T): T
```

reduce将RDD中元素前两个传给输入函数，产生一个新的return值，新产生的return值与RDD中下一个元素（第三个元素）组成两个元素，再被传给输入函数，直到最后只有一个值为止。

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。

使用：

```
scala> val a = sc.parallelize(1 to 100,2)
a: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[49] at parallelize at <console>:27

scala> a.reduce(_+_)
res14: Int = 5050
```

6.2 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。

```
val a = sc.parallelize(List(1,2,3), 3)
a.fold(0)(_ + _)
res59: Int = 6
```

6.3 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。

用户可以定义两个reduce函数对RDD进行聚合，第一个reduce函数作用于分区，第二个reduce函数作用于不同分区。

```
val z = sc.parallelize(List(1,2,3,4,5,6), 2)

// lets first print out the contents of the RDD with partition labels
```

```
def myfunc(index: Int, iter: Iterator[(Int)]) : Iterator[String] = {
  iter.map(x => "[partID:" +  index + ", val: " + x + "]")
}

z.mapPartitionsWithIndex(myfunc).collect
res28: Array[String] = Array([partID:0, val: 1], [partID:0, val: 2], [partID:0, val: 3],
[partID:1, val: 4], [partID:1, val: 5], [partID:1, val: 6])

z.aggregate(0)(math.max(_, _), _ + _)
res40: Int = 9

// This example returns 16 since the initial value is 5
// reduce of partition 0 will be max(5, 1, 2, 3) = 5
// reduce of partition 1 will be max(5, 4, 5, 6) = 6
// final reduce across partitions will be 5 + 5 + 6 = 16
// note the final reduce include the initial value
z.aggregate(5)(math.max(_, _), _ + _)
res29: Int = 16


val z = sc.parallelize(List("a","b","c","d","e","f"),2)

//lets first print out the contents of the RDD with partition labels
def myfunc(index: Int, iter: Iterator[(String)]) : Iterator[String] = {
  iter.map(x => "[partID:" +  index + ", val: " + x + "]")
}

z.mapPartitionsWithIndex(myfunc).collect
res31: Array[String] = Array([partID:0, val: a], [partID:0, val: b], [partID:0, val: c],
[partID:1, val: d], [partID:1, val: e], [partID:1, val: f])

z.aggregate("")(_ + _, _+_)
res115: String = abcdef

// See here how the initial value "x" is applied three times.
//  - once for each partition
//  - once when combining all the partitions in the second reduce function.
z.aggregate("x")(_ + _, _+_)
res116: String = xxdefxabc

// Below are some more advanced examples. Some are quite tricky to work out.

val z = sc.parallelize(List("12","23","345","4567"),2)
z.aggregate("")((x,y) => math.max(x.length, y.length).toString, (x,y) => x + y)
res141: String = 42

z.aggregate("")((x,y) => math.min(x.length, y.length).toString, (x,y) => x + y)
res142: String = 11

val z = sc.parallelize(List("12","23","345",""),2)
z.aggregate("")((x,y) => math.min(x.length, y.length).toString, (x,y) => x + y)
res143: String = 10
```

```
第一个参数是分区里的每个元素相加，第二个参数是每个分区的结果再相加
rdd1.aggregate(0)(_+_, _+_)
需求：把每个分区的最大值取出来，再把各分区最大值相加
rdd1.aggregate(0)(math.max(_, _), _+_)
再看初始值设为10的结果
rdd1.aggregate(10)(math.max(_, _), _+_)
再看初始值设为2的结果
rdd1.aggregate(2)(math.max(_, _), _+_)

def func1(index: Int, iter: Iterator[(Int)]) : Iterator[String] = {
  iter.toList.map(x => "[partID:" +  index + ", val: " + x + "]").iterator
}
val rdd1 = sc.parallelize(List(1,2,3,4,5,6,7,8,9), 2)
rdd1.mapPartitionsWithIndex(func1).collect
rdd1.aggregate(0)(math.max(_, _), _ + _)
rdd1.aggregate(5)(math.max(_, _), _ + _)


val rdd2 = sc.parallelize(List("a","b","c","d","e","f"),2)
def func2(index: Int, iter: Iterator[(String)]) : Iterator[String] = {
  iter.toList.map(x => "[partID:" +  index + ", val: " + x + "]").iterator
}
rdd2.mapPartitionsWithIndex(func2).collect  --查看每个分区的元素
rdd2.aggregate("")(_ + _, _ + _)
查看初始值被应用了几次
rdd2.aggregate("=")(_ + _, _ + _)
如果设了三个分区，初始值被应用了几次？

val rdd3 = sc.parallelize(List("12","23","345","4567"),2)
rdd3.mapPartitionsWithIndex(func2).collect  --查看每个分区的元素
每次返回的值不一样，因为executor有时返回的慢，有时返回的快一些
rdd3.aggregate("")((x,y) => math.max(x.length, y.length).toString, (x,y) => x + y)

val rdd4 = sc.parallelize(List("12","23","345",""),2)
rdd4.mapPartitionsWithIndex(func2).collect  --查看每个分区的元素
为什么是01或10? 关键点："".length是"0",下次比较最小length就是1了
rdd4.aggregate("")((x,y) => math.min(x.length, y.length).toString, (x,y) => x + y)

val rdd5 = sc.parallelize(List("12","23","","345"),2)
rdd5.mapPartitionsWithIndex(func2).collect
rdd5.aggregate("")((x,y) => math.min(x.length, y.length).toString, (x,y) => x + y)
```

6.4 collect

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。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。

使用：

```
scala> val c = sc.parallelize(List("Gnu", "Cat", "Rat", "Dog", "Gnu", "Rat"), 2)
c: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[50] at parallelize at <console>:27

scala> c.collect
res15: Array[String] = Array(Gnu, Cat, Rat, Dog, Gnu, Rat)
```

6.5 collectAsMap [Pair]

类似collect，将键值对以map的形式搜集到driver端。

```
scala> val a = sc.parallelize(List(1, 2, 1, 3), 1)
a: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[0] at parallelize at <console>:27

scala> val b = a.zip(a)
b: org.apache.spark.rdd.RDD[(Int, Int)] = ZippedPartitionsRDD2[1] at zip at <console>:29

scala> b.collectAsMap
res0: scala.collection.Map[Int,Int] = Map(2 -> 2, 1 -> 1, 3 -> 3)
```

6.6 count

返回RDD的元素个数

```
scala> val c = sc.parallelize(List("Gnu", "Cat", "Rat", "Dog"), 2)
c: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[51] at parallelize at <console>:27

scala> c.count
res16: Long = 4
```

6.7 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。

```
```

6.8 take(n)

返回一个由数据集的前n个元素组成的数组

```
val b = sc.parallelize(List("dog", "cat", "ape", "salmon", "gnu"), 2)
b.take(2)
res18: Array[String] = Array(dog, cat)

val b = sc.parallelize(1 to 10000, 5000)
b.take(100)
res6: Array[Int] = Array(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20,
21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44,
45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68,
69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92,
93, 94, 95, 96, 97, 98, 99, 100)
```

6.9 takeSample(*withReplacement*,*num*, [*seed*])

返回一个数组，该数组由从数据集中随机采样的num个元素组成，可以选择是否用随机数替换不足的部分，seed用于指定随机数生成器种子

```
scala> val x = sc.parallelize(1 to 1000, 3)
x: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[0] at parallelize at <console>:27

scala> x.takeSample(true, 100, 1)
res0: Array[Int] = Array(176, 110, 806, 214, 977, 460, 274, 656, 977, 667, 863, 283, 575, 864,
90, 316, 923, 953, 500, 572, 421, 745, 199, 738, 829, 31, 604, 29, 439, 351, 820, 848, 675, 974,
210, 941, 764, 489, 586, 354, 160, 900, 7, 517, 485, 173, 188, 685, 416, 229, 938, 466, 496,
901, 291, 538, 223, 781, 39, 546, 297, 723, 452, 749, 737, 515, 868, 425, 197, 912, 487, 562,
330, 276, 363, 316, 457, 29, 584, 544, 987, 501, 565, 299, 823, 481, 555, 739, 768, 493, 889,
59, 102, 579, 475, 800, 445, 172, 189, 211)

scala> val x = sc.parallelize(1 to 10, 3)
x: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[2] at parallelize at <console>:27

scala> x.takeSample(true, 20, 1)
res1: Array[Int] = Array(7, 10, 9, 3, 7, 10, 7, 3, 1, 2, 10, 4, 10, 5, 10, 9, 2, 3, 7, 9)
```

6.10 takeOrdered(*n*, *[ordering]*)

takeOrdered和top类似，只不过以和top相反的顺序返回元素

```
val b = sc.parallelize(List("dog", "cat", "ape", "salmon", "gnu"), 2)
b.takeOrdered(2)
res19: Array[String] = Array(ape, cat)
```

6.11 first

返回RDD的第一个元素（类似于take(1)）

```
val c = sc.parallelize(List("Gnu", "Cat", "Rat", "Dog"), 2)
c.first
res1: String = Gnu
```

countByKey() 针对(K,V)类型的RDD，返回一个(K,Int)的map，表示每一个key对应的元素个数。

```
scala> val c = sc.parallelize(List((3, "Gnu"), (3, "Yak"), (5, "Mouse"), (3, "Dog")), 2)
c: org.apache.spark.rdd.RDD[(Int, String)] = ParallelCollectionRDD[4] at parallelize at
<console>:27

scala> c.countByKey
res2: scala.collection.Map[Int,Long] = Map(3 -> 3, 5 -> 1)
```

6.12 foreach(*func*)

在数据集的每一个元素上，运行函数func进行更新

```
scala> val c = sc.parallelize(List("cat", "dog", "tiger", "lion", "gnu", "crocodile", "ant",
"whale", "dolphin", "spider"), 3)
c: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[7] at parallelize at <console>:27

scala> c.foreach(x => println(x + "s are yummy"))
cats are yummy
dogs are yummy
tigers are yummy
lions are yummy
gnus are yummy
crocodiles are yummy
ants are yummy
whales are yummy
dolphins are yummy
spiders are yummy
```

注意：

1.foreach是作用在每个分区，结果输出到分区；

2.由于并行的原因，每个节点上打印的结果每次运行可能会不同。

6.13 saveAsTextFile(*path*)

将数据集的元素以textfile的形式保存到HDFS文件系统或者其他支持的文件系统，对于每个元素，Spark将会调用toString方法，将它装换为文件中的文本

```
val a = sc.parallelize(1 to 10000, 3)
a.saveAsTextFile("mydata_a")
```

9，RDD上的统计操作

Spark 对包含数值数据的 RDD 提供了一些描述性的统计操作。 Spark 的数值操作是通过流式算法实现的，允许以每次一个元素的方式构建出模型。这些 统计数据都会在调用 stats() 时通过一次遍历数据计算出来，并以 StatsCounter 对象返回。

```
scala> var rdd1 = sc.makeRDD(1 to 100)
rdd1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[42] at makeRDD at <console>:32

scala> rdd1.sum()
res34: Double = 5050.0

scala> rdd1.max()
res35: Int = 100
```