**SparkCore应用解析**

**教案**

**文档修订记录**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 文件状态：  [√ ]草稿  [ ]正式发布 | | **当前版本：** | | V1.1 | | | |
| **作 者：** | | 武玉飞 | | | |
| **审 核 人：** | |  | | | |
| **发布日期：** | | 20170518 | | | |
| **编制日期** | **版本** | | **状态** | **简要说明** | **作者** | **审核者** | **审核日期** |
| 20170518 | V1.1 | | A |  |  |  |  |
|  |  | |  |  |  |  |  |
|  |  | |  |  |  |  |  |
|  |  | |  |  |  |  |  |
|  |  | |  |  |  |  |  |
|  |  | |  |  |  |  |  |

说明：

1. 按修改时间先后倒序排列，最新修改的排在第一行。
2. 版本栏中填入版本编号或者更改记录编号。
3. 状态分为三种状态：A­——增加；M——修改；D——删除。
4. 在简要说明栏中填写变更的内容和变更的范围。
5. 表中所有日期格式为：YYYYMMDD。

**目 录**

[第1章 RDD概念 3](#_Toc492414571)

[1.1 RDD为什么会产生 3](#_Toc492414572)

[1.2 RDD概述 4](#_Toc492414573)

**[1.2.1](#_Toc492414574)** [什么是RDD 4](#_Toc492414574)

**[1.2.2](#_Toc492414575)** [RDD的属性 5](#_Toc492414575)

[1.3 RDD弹性 7](#_Toc492414576)

[1.4 RDD特点 8](#_Toc492414577)

**[1.4.1](#_Toc492414578)** [分区 8](#_Toc492414578)

**[1.4.2](#_Toc492414579)** [只读 9](#_Toc492414579)

**[1.4.3](#_Toc492414580)** [依赖 10](#_Toc492414580)

**[1.4.4](#_Toc492414581)** [缓存 11](#_Toc492414581)

**[1.4.5](#_Toc492414582)** [checkpoint 12](#_Toc492414582)

[第2章 RDD编程 13](#_Toc492414583)

[2.1 编程模型 13](#_Toc492414584)

[2.2 创建RDD 14](#_Toc492414585)

[2.3 RDD编程 16](#_Toc492414586)

**[2.3.1](#_Toc492414587)** [Transformation 16](#_Toc492414587)

**[2.3.2](#_Toc492414588)** [Action 43](#_Toc492414588)

**[2.3.3](#_Toc492414589)** [数值RDD的统计操作 50](#_Toc492414589)

**[2.3.4](#_Toc492414590)** [向RDD操作传递函数注意 51](#_Toc492414590)

**[2.3.5](#_Toc492414591)** [在不同RDD类型间转换 52](#_Toc492414591)

[2.4 RDD持久化 54](#_Toc492414592)

**[2.4.1](#_Toc492414593)** [RDD的缓存 54](#_Toc492414593)

**[2.4.2](#_Toc492414594)** [RDD缓存方式 54](#_Toc492414594)

[2.5 RDD检查点机制 55](#_Toc492414595)

**[2.5.1](#_Toc492414596)** [checkpoint 写流程 56](#_Toc492414596)

**[2.5.2](#_Toc492414597)** [checkpoint 读流程 58](#_Toc492414597)

[2.6 RDD的依赖关系 58](#_Toc492414598)

**[2.6.1](#_Toc492414599)** [窄依赖 59](#_Toc492414599)

**[2.6.2](#_Toc492414600)** [宽依赖 59](#_Toc492414600)

**[2.6.3](#_Toc492414601)** [Lineage 59](#_Toc492414601)

[2.7 DAG的生成 60](#_Toc492414602)

[2.8 RDD相关概念关系 61](#_Toc492414603)

[第3章 键值对RDD 62](#_Toc492414604)

[3.1 键值对RDD的转化操作 63](#_Toc492414605)

**[3.1.1](#_Toc492414606)** [转化操作列表 63](#_Toc492414606)

**[3.1.2](#_Toc492414607)** [聚合操作 65](#_Toc492414607)

**[3.1.3](#_Toc492414608)** [数据分组 67](#_Toc492414608)

**[3.1.4](#_Toc492414609)** [连接 67](#_Toc492414609)

**[3.1.5](#_Toc492414610)** [数据排序 68](#_Toc492414610)

[3.2 键值对RDD的行动操作 69](#_Toc492414611)

[3.3 键值对RDD的数据分区 69](#_Toc492414612)

**[3.3.1](#_Toc492414613)** [获取RDD的分区方式 69](#_Toc492414613)

**[3.3.2](#_Toc492414614)** [Hash分区方式 70](#_Toc492414614)

**[3.3.3](#_Toc492414615)** [Range分区方式 71](#_Toc492414615)

**[3.3.4](#_Toc492414616)** [自定义分区方式 71](#_Toc492414616)

**[3.3.5](#_Toc492414617)** [分区Shuffle优化 73](#_Toc492414617)

**[3.3.6](#_Toc492414618)** [基于分区进行操作 75](#_Toc492414618)

**[3.3.7](#_Toc492414619)** [从分区中获益的操作 76](#_Toc492414619)

[第4章 数据读取与保存主要方式 76](#_Toc492414620)

[4.1 文本文件输入输出 76](#_Toc492414621)

[4.2 JSON文件输入输出 77](#_Toc492414622)

[4.3 CSV文件输入输出 78](#_Toc492414623)

[4.4 SequenceFile文件输入输出 78](#_Toc492414624)

[4.5 对象文件输入输出 79](#_Toc492414625)

[4.6 Hadoop输入输出格式 80](#_Toc492414626)

[4.7 文件系统的输入输出 84](#_Toc492414627)

[4.8 数据库的输入输出 84](#_Toc492414628)

[第5章 RDD编程进阶 88](#_Toc492414629)

[5.1 累加器 88](#_Toc492414630)

[5.2 自定义累加器 90](#_Toc492414631)

[5.3 广播变量 92](#_Toc492414632)

# RDD概念

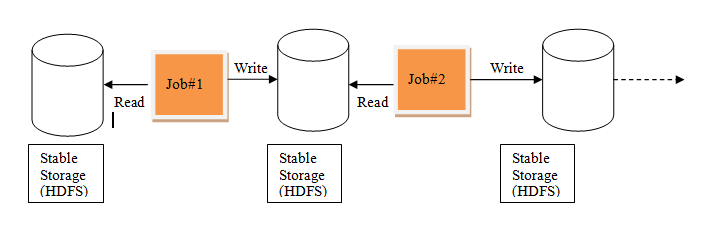
## RDD为什么会产生

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？

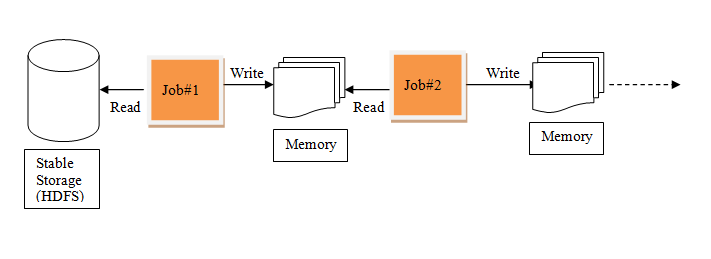
Hadoop的MapReduce是一种基于数据集的工作模式，面向数据，这种工作模式一般是从存储上加载数据集，然后操作数据集，最后写入物理存储设备。数据更多面临的是一次性处理。

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。

## RDD概述

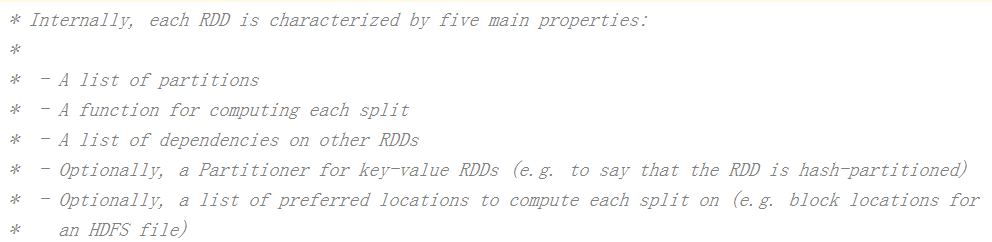
### 什么是RDD

RDD（Resilient Distributed Dataset）叫做分布式数据集，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。在 Spark 中，对数据的所有操作不外乎创建 RDD、转化已有RDD 以及调用 RDD 操作进行求值。每个 RDD 都被分为多个分区，这些分区运行在集群中的不同节点上。RDD 可以包含 Python、Java、Scala 中任意类型的对象， 甚至可以包含用户自定义的对象。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。

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()。

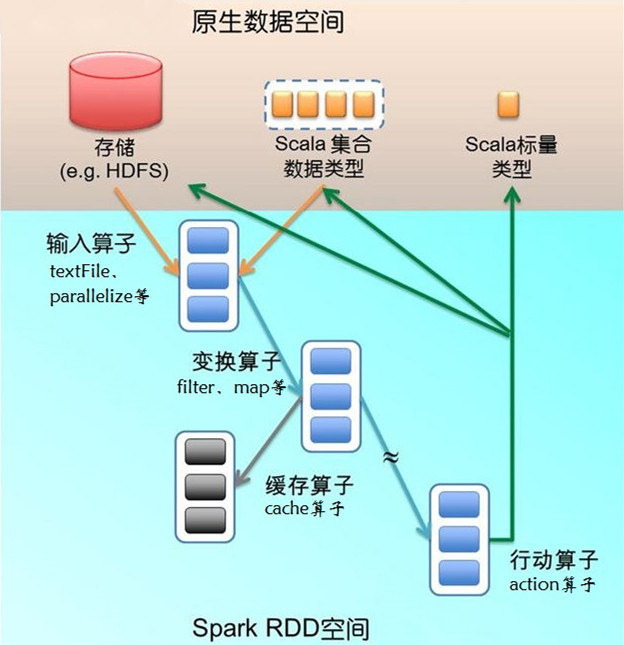
Spark采用惰性计算模式，RDD只有第一次在一个行动操作中用到时，才会真正计算。Spark可以优化整个计算过程。默认情况下，Spark 的 RDD 会在你每次对它们进行行动操作时重新计算。如果想在多个行动操作中重用同一个 RDD，可以使用 RDD.persist() 让 Spark 把这个 RDD 缓存下来。

### RDD的属性



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CPU Core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。

RDD是一个应用层面的逻辑概念。一个RDD多个分片。RDD就是一个元数据记录集，记录了RDD内存所有的关系数据。



Spark到底干了点啥？

## RDD弹性

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

1. 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。

1. 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。

1. 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。

1. 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。

1. 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。

1. 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。

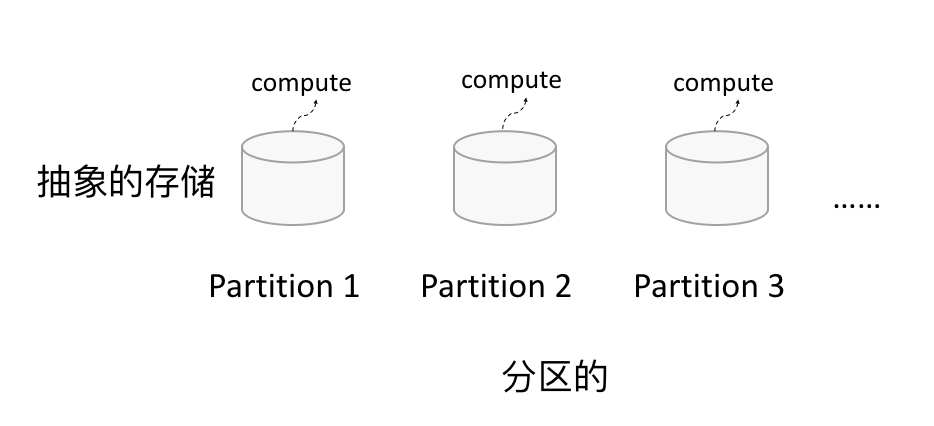
RDD全称叫做弹性分布式数据集(Resilient Distributed Datasets)，它是一种分布式的内存抽象，表示一个只读的记录分区的集合，它只能通过其他RDD转换而创建，为此，RDD支持丰富的转换操作(如map, join, filter, groupBy等)，通过这种转换操作，新的RDD则包含了如何从其他RDDs衍生所必需的信息，所以说RDDs之间是有依赖关系的。基于RDDs之间的依赖，RDDs会形成一个有向无环图DAG，该DAG描述了整个流式计算的流程，实际执行的时候，RDD是通过血缘关系(Lineage)一气呵成的，即使出现数据分区丢失，也可以通过血缘关系重建分区，总结起来，基于RDD的流式计算任务可描述为：从稳定的物理存储(如分布式文件系统)中加载记录，记录被传入由一组确定性操作构成的DAG，然后写回稳定存储。另外RDD还可以将数据集缓存到内存中，使得在多个操作之间可以重用数据集，基于这个特点可以很方便地构建迭代型应用(图计算、机器学习等)或者交互式数据分析应用。可以说Spark最初也就是实现RDD的一个分布式系统，后面通过不断发展壮大成为现在较为完善的大数据生态系统，简单来讲，Spark-RDD的关系类似于Hadoop-MapReduce关系。

## 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。

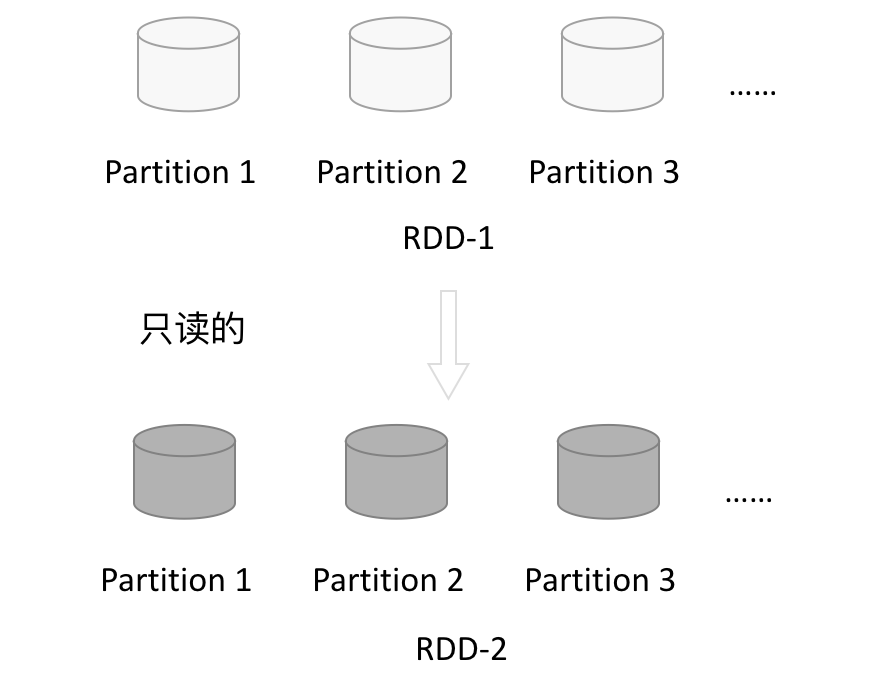
### 分区

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。

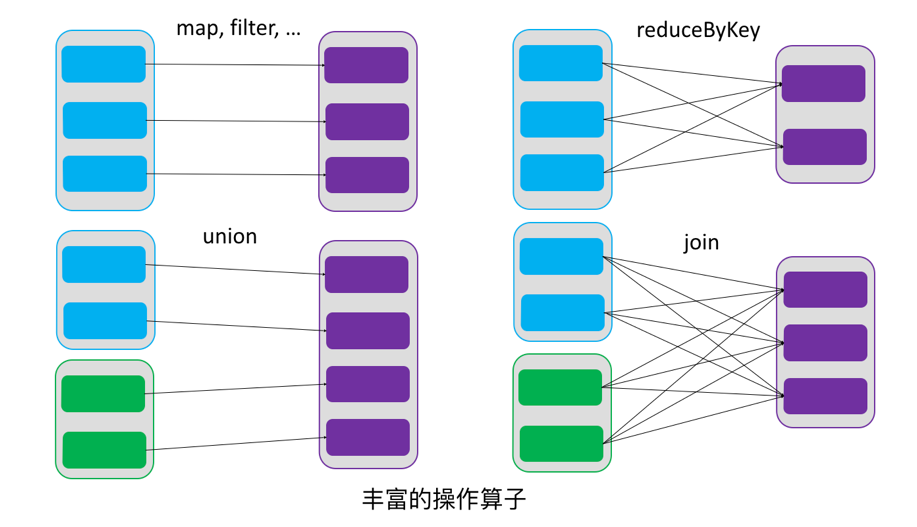


### 只读

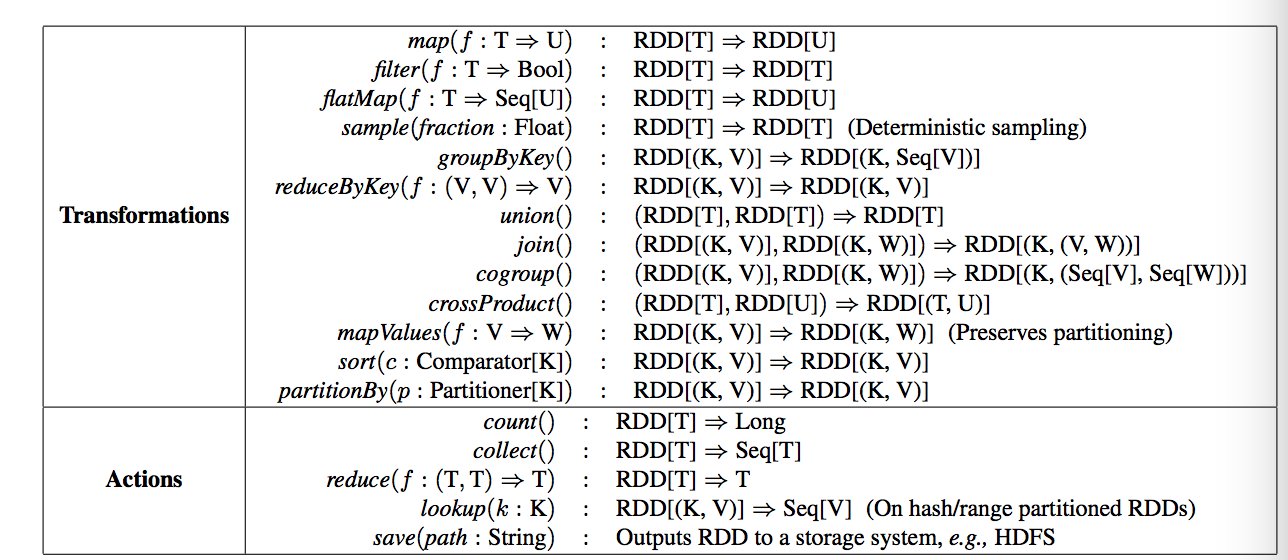
如下图所示，RDD是只读的，要想改变RDD中的数据，只能在现有的RDD基础上创建新的RDD。



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。

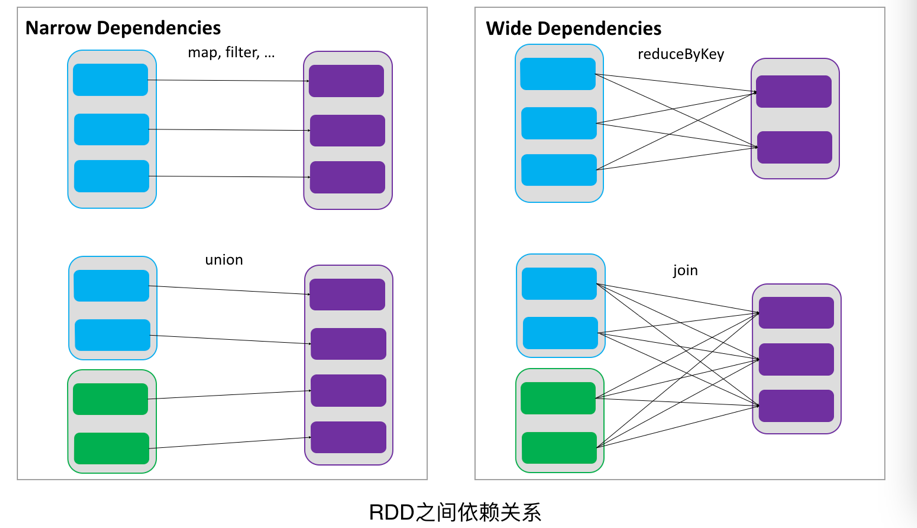


RDD的操作算子包括两类，一类叫做transformations，它是用来将RDD进行转化，构建RDD的血缘关系；另一类叫做actions，它是用来触发RDD的计算，得到RDD的相关计算结果或者将RDD保存的文件系统中。下图是RDD所支持的操作算子列表。

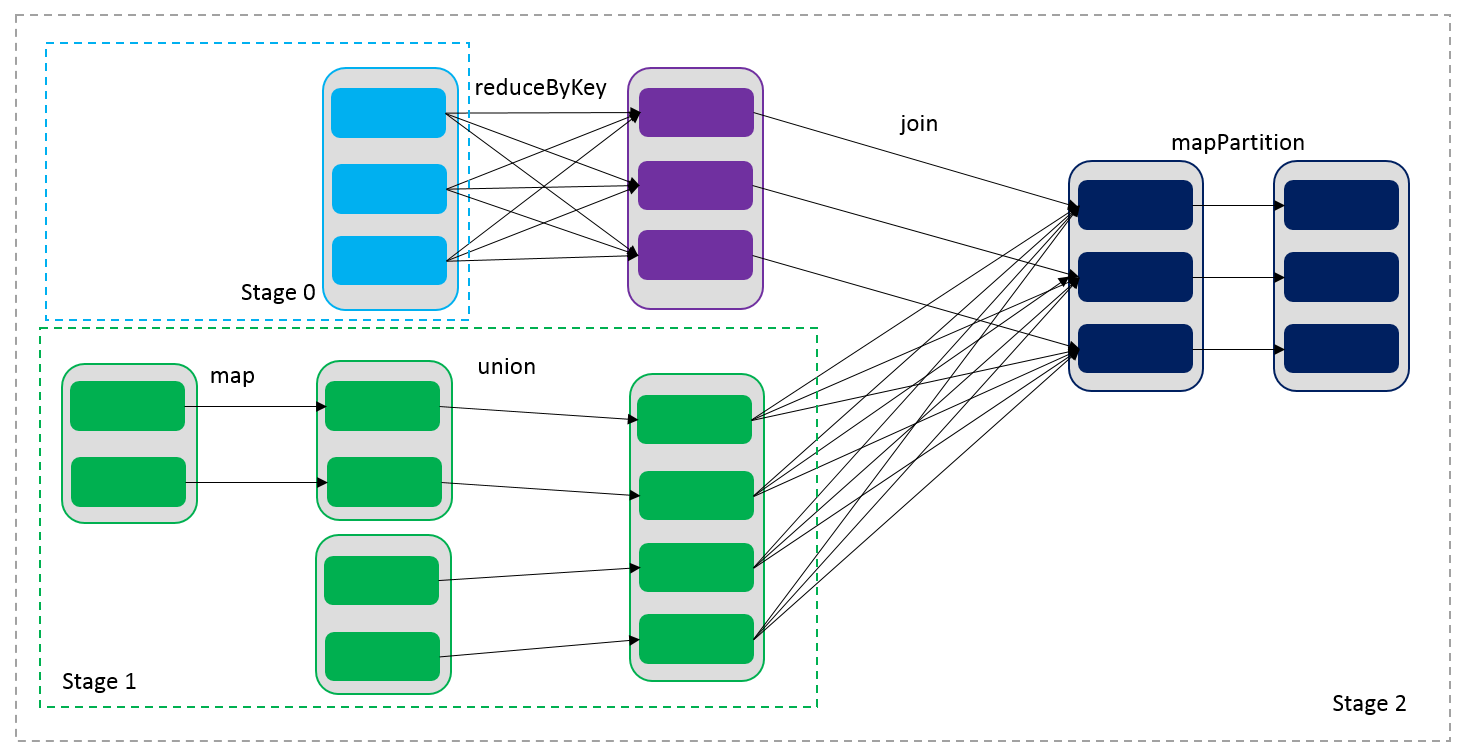


### 依赖

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。

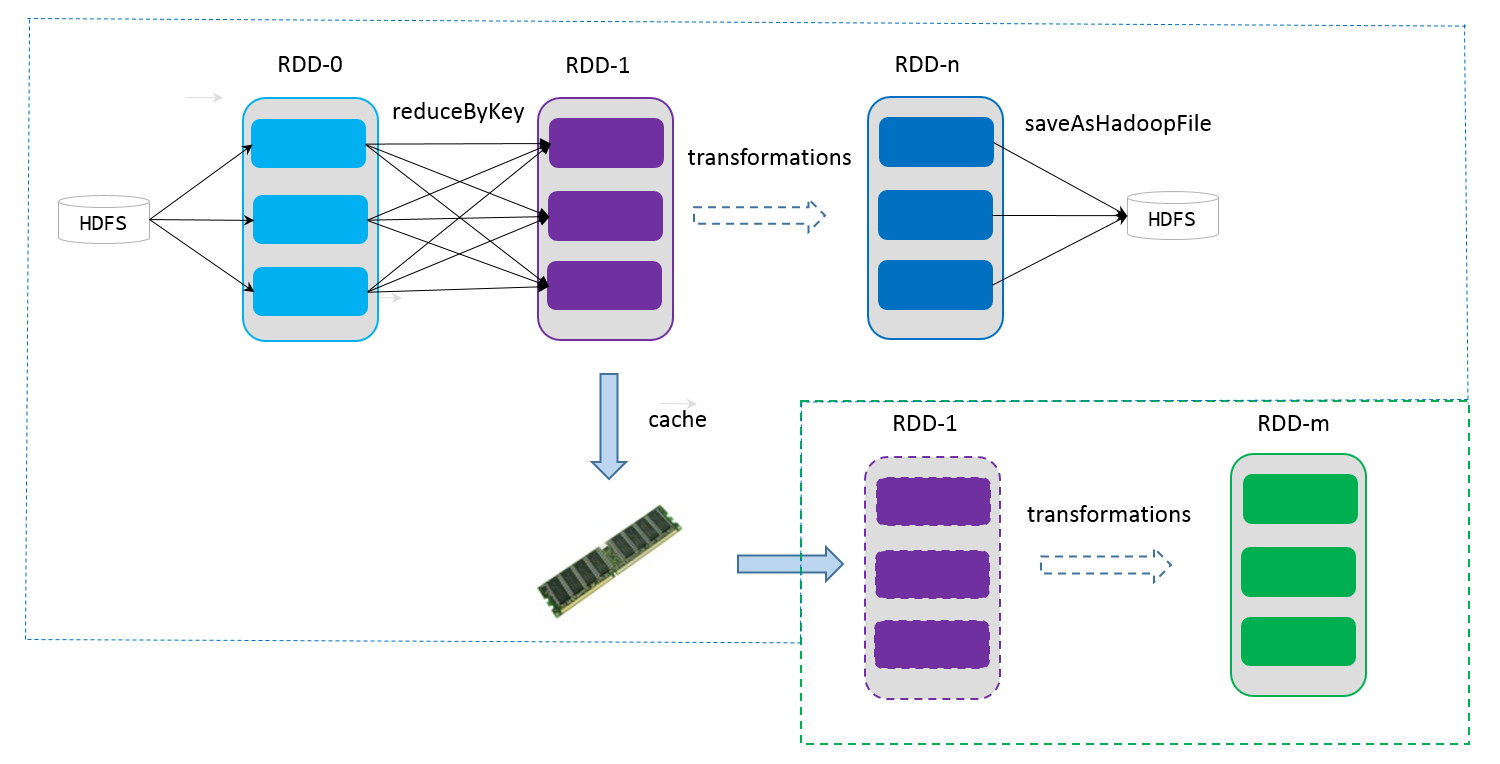


通过RDDs之间的这种依赖关系，一个任务流可以描述为DAG(有向无环图)，如下图所示，在实际执行过程中宽依赖对应于Shuffle(图中的reduceByKey和join)，窄依赖中的所有转换操作可以通过类似于管道的方式一气呵成执行(图中map和union可以一起执行)。



### 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。



### 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。

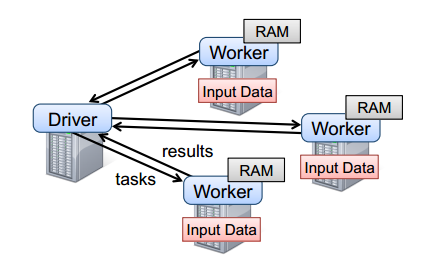
给定一个RDD我们至少可以知道如下几点信息：1、分区数以及分区方式；2、由父RDDs衍生而来的相关依赖信息；3、计算每个分区的数据，计算步骤为：1）如果被缓存，则从缓存中取的分区的数据；2）如果被checkpoint，则从checkpoint处恢复数据；3）根据血缘关系计算分区的数据。

# RDD编程

## 编程模型

在Spark中，RDD被表示为对象，通过对象上的方法调用来对RDD进行转换。经过一系列的transformations定义RDD之后，就可以调用actions触发RDD的计算，action可以是向应用程序返回结果(count, collect等)，或者是向存储系统保存数据(saveAsTextFile等)。在Spark中，只有遇到action，才会执行RDD的计算(即延迟计算)，这样在运行时可以通过管道的方式传输多个转换。

要使用Spark，开发者需要编写一个Driver程序，它被提交到集群以调度运行Worker，如下图所示。Driver中定义了一个或多个RDD，并调用RDD上的action，Worker则执行RDD分区计算任务。



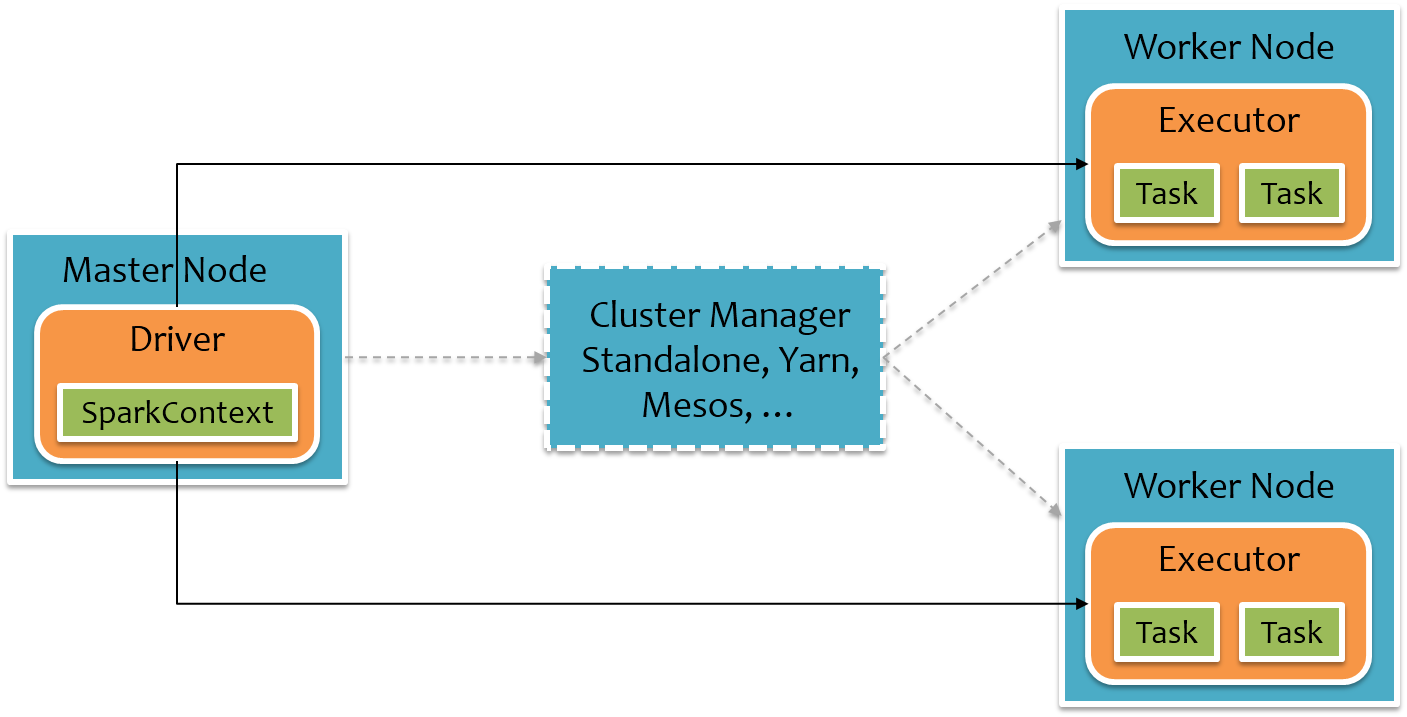
Dirver是啥？

SparkContext是啥？

Executor是啥？

Master是啥？

Worker是啥？



## 创建RDD

在Spark中创建RDD的创建方式大概可以分为三种：（1）、从集合中创建RDD；（2）、从外部存储创建RDD；（3）、从其他RDD创建。

1. 由一个已经存在的Scala集合创建，集合并行化。

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

　　而从集合中创建RDD，[Spark](https://www.iteblog.com/archives/tag/spark/" \t "_blank)主要提供了两种函数：parallelize和makeRDD。我们可以先看看这两个函数的声明：

|  |  |
| --- | --- |
| |  | | --- | | def parallelize[T: ClassTag](        seq: Seq[T],        numSlices: Int = defaultParallelism): RDD[T]    def makeRDD[T: ClassTag](        seq: Seq[T],        numSlices: Int = defaultParallelism): RDD[T]    def makeRDD[T: ClassTag](seq: Seq[(T, Seq[String])]): RDD[T] | |

　　我们可以从上面看出makeRDD有两种实现，而且第一个makeRDD函数接收的参数和parallelize完全一致。其实第一种makeRDD函数实现是依赖了parallelize函数的实现，来看看Spark中是怎么实现这个makeRDD函数的：

|  |  |
| --- | --- |
| |  | | --- | | def makeRDD[T: ClassTag](      seq: Seq[T],      numSlices: Int = defaultParallelism): RDD[T] = withScope {    parallelize(seq, numSlices)  } | |

　　我们可以看出，这个makeRDD函数完全和parallelize函数一致。但是我们得看看第二种makeRDD函数函数实现了，它接收的参数类型是Seq[(T, Seq[String])]，Spark文档的说明是：

Distribute a local Scala collection to form an RDD, with one or more location preferences (hostnames of Spark nodes) for each object. Create a new partition for each collection item.

原来，这个函数还为数据提供了位置信息，来看看我们怎么使用：

|  |  |
| --- | --- |
| |  | | --- | | scala> val guigu1= sc.parallelize(List(1,2,3))  guigu1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[10] at parallelize at <console>:21    scala> val guigu2 = sc.makeRDD(List(1,2,3))  guigu2: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[11] at makeRDD at <console>:21    scala> val seq = List((1, List("Hello", "World", "Spark")),       | (2, List("Atguigu", "JiaoYu")))  seq: List[(Int, List[String])] = List((1,List(Hello, World, Spark)),   (2,List(Atguigu, JiaoYu)))    scala> val guigu3 = sc.makeRDD(seq)  guigu3: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[12] at makeRDD at <console>:23    scala> guigu3.preferredLocations(guigu3.partitions(1))  res26: Seq[String] = List(Atguigu, JiaoYu)    scala> guigu3.preferredLocations(guigu3.partitions(0))  res27: Seq[String] = List(Hello, World, Spark)    scala> guigu1.preferredLocations(guigu1.partitions(0))  res28: Seq[String] = List() | |

我们可以看到，makeRDD函数有两种实现，第一种实现其实完全和parallelize一致；而第二种实现可以为数据提供位置信息，而除此之外的实现和parallelize函数也是一致的，如下：

|  |  |
| --- | --- |
| |  | | --- | | def parallelize[T: ClassTag](      seq: Seq[T],      numSlices: Int = defaultParallelism): RDD[T] = withScope {    assertNotStopped()    new ParallelCollectionRDD[T](this, seq, numSlices, Map[Int, Seq[String]]())  }    def makeRDD[T: ClassTag](seq: Seq[(T, Seq[String])]): RDD[T] = withScope {    assertNotStopped()    val indexToPrefs = seq.zipWithIndex.map(t => (t.\_2, t.\_1.\_2)).toMap    new ParallelCollectionRDD[T](this, seq.map(\_.\_1), seq.size, indexToPrefs)  } | |

都是返回ParallelCollectionRDD，而且这个makeRDD的实现不可以自己指定分区的数量，而是固定为seq参数的size大小。

1. 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、HBase等

|  |
| --- |
| scala> val atguigu = sc.textFile("hdfs://master01:9000/RELEASE")  atguigu: org.apache.spark.rdd.RDD[String] = hdfs://master01:9000/RELEASE MapPartitionsRDD[4] at textFile at <console>:24 |

## RDD编程

RDD一般分为数值RDD和键值对RDD，本章不进行具体区分，先统一来看，下一章会对键值对RDD做专门说明。

### Transformation

RDD中的所有转换都是延迟加载的，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：**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 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 |
| **../../../../../../Desktop/20161123221628837.**   |  | | --- | | scala> var source = sc.parallelize(1 to 10)  source: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[8] at parallelize at <console>:24  scala> source.collect()  res7: Array[Int] = Array(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)  scala> val mapadd = source.map(\_ \* 2)  mapadd: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[9] at map at <console>:26  scala> mapadd.collect()  res8: Array[Int] = Array(2, 4, 6, 8, 10, 12, 14, 16, 18, 20) | | | | | |
| 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 |
| |  | | --- | | scala> var sourceFilter = sc.parallelize(Array("xiaoming","xiaojiang","xiaohe","dazhi"))  sourceFilter: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[10] at parallelize at <console>:24  scala> val filter = sourceFilter.filter(\_.contains("xiao"))  filter: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[11] at filter at <console>:26  scala> sourceFilter.collect()  res9: Array[String] = Array(xiaoming, xiaojiang, xiaohe, dazhi)  scala> filter.collect()  res10: Array[String] = Array(xiaoming, xiaojiang, xiaohe) | | | | | |
| 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）** |
| **../../../../../../Desktop/20161123221615509.**   |  | | --- | | scala> val sourceFlat = sc.parallelize(1 to 5)  sourceFlat: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[12] at parallelize at <console>:24  scala> sourceFlat.collect()  res11: Array[Int] = Array(1, 2, 3, 4, 5)  scala> val flatMap = sourceFlat.flatMap(1 to \_)  flatMap: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[13] at flatMap at <console>:26  scala> flatMap.collect()  res12: Array[Int] = Array(1, 1, 2, 1, 2, 3, 1, 2, 3, 4, 1, 2, 3, 4, 5) | | | | | |
| 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]。假设有N个元素，有M个分区，那么map的函数的将被调用N次,而mapPartitions被调用M次,一个函数一次处理所有分区 |
| |  | | --- | | scala> val rdd = sc.parallelize(List(("kpop","female"),("zorro","male"),("mobin","male"),("lucy","female")))  rdd: org.apache.spark.rdd.RDD[(String, String)] = ParallelCollectionRDD[16] at parallelize at <console>:24  scala> :paste  // Entering paste mode (ctrl-D to finish)  def partitionsFun(iter : Iterator[(String,String)]) : Iterator[String] = {  var woman = List[String]()  while (iter.hasNext){  val next = iter.next()  next match {  case (\_,"female") => woman = next.\_1 :: woman  case \_ =>  }  }  woman.iterator  }  // Exiting paste mode, now interpreting.  partitionsFun: (iter: Iterator[(String, String)])Iterator[String]  scala> val result = rdd.mapPartitions(partitionsFun)  result: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[17] at mapPartitions at <console>:28  scala> result.collect()  res13: Array[String] = Array(kpop, lucy) | | | | | |
| 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, Interator[T]) => Iterator[U] |
| |  | | --- | | scala> val rdd = sc.parallelize(List(("kpop","female"),("zorro","male"),("mobin","male"),("lucy","female")))  rdd: org.apache.spark.rdd.RDD[(String, String)] = ParallelCollectionRDD[18] at parallelize at <console>:24  scala> :paste  // Entering paste mode (ctrl-D to finish)  def partitionsFun(index : Int, iter : Iterator[(String,String)]) : Iterator[String] = {  var woman = List[String]()  while (iter.hasNext){  val next = iter.next()  next match {  case (\_,"female") => woman = "["+index+"]"+next.\_1 :: woman  case \_ =>  }  }  woman.iterator  }  // Exiting paste mode, now interpreting.  partitionsFun: (index: Int, iter: Iterator[(String, String)])Iterator[String]  scala> val result = rdd.mapPartitionsWithIndex(partitionsFun)  result: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[19] at mapPartitionsWithIndex at <console>:28  scala> result.collect()  res14: Array[String] = Array([0]kpop, [3]lucy) | | | | | |
| sample(withReplacement, fraction, seed) | | | | 以指定的随机种子随机抽样出数量为fraction的数据，withReplacement表示是抽出的数据是否放回，true为有放回的抽样，false为无放回的抽样，seed用于指定随机数生成器种子。例子从RDD中随机且有放回的抽出50%的数据，随机种子值为3（即可能以1 2 3的其中一个起始值） |
| |  | | --- | | scala> val rdd = sc.parallelize(1 to 10)  rdd: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[20] at parallelize at <console>:24  scala> rdd.collect()  res15: Array[Int] = Array(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)  scala> var sample1 = rdd.sample(true,0.4,2)  sample1: org.apache.spark.rdd.RDD[Int] = PartitionwiseSampledRDD[21] at sample at <console>:26  scala> sample1.collect()  res16: Array[Int] = Array(1, 2, 2, 7, 7, 8, 9)  scala> var sample2 = rdd.sample(false,0.2,3)  sample2: org.apache.spark.rdd.RDD[Int] = PartitionwiseSampledRDD[22] at sample at <console>:26  scala> sample2.collect()  res17: Array[Int] = Array(1, 9) | | | | | |
| takeSample | | | | 和Sample的区别是：takeSample返回的是最终的结果集合。 |
|  | | | | |
| union(otherDataset) | | | | 对源RDD和参数RDD求并集后返回一个新的RDD |
| |  | | --- | | scala> val rdd1 = sc.parallelize(1 to 5)  rdd1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[23] at parallelize at <console>:24  scala> val rdd2 = sc.parallelize(5 to 10)  rdd2: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[24] at parallelize at <console>:24  scala> val rdd3 = rdd1.union(rdd2)  rdd3: org.apache.spark.rdd.RDD[Int] = UnionRDD[25] at union at <console>:28  scala> rdd3.collect()  res18: Array[Int] = Array(1, 2, 3, 4, 5, 5, 6, 7, 8, 9, 10) | | | | | |
| intersection(otherDataset) | | | | 对源RDD和参数RDD求交集后返回一个新的RDD |
| |  | | --- | | scala> val rdd1 = sc.parallelize(1 to 7)  rdd1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[26] at parallelize at <console>:24  scala> val rdd2 = sc.parallelize(5 to 10)  rdd2: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[27] at parallelize at <console>:24  scala> val rdd3 = rdd1.intersection(rdd2)  rdd3: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[33] at intersection at <console>:28  scala> rdd3.collect()  [Stage 15:=============================> (2 + 2) res19: Array[Int] = Array(5, 6, 7) | | | | | |
| distinct([numTasks])) | | | | 对源RDD进行去重后返回一个新的RDD. 默认情况下，只有8个并行任务来操作，但是可以传入一个可选的numTasks参数改变它。 |
| |  | | --- | | scala> val distinctRdd = sc.parallelize(List(1,2,1,5,2,9,6,1))  distinctRdd: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[34] at parallelize at <console>:24  scala> val unionRDD = distinctRdd.distinct()  unionRDD: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[37] at distinct at <console>:26  scala> unionRDD.collect()  [Stage 16:> (0 + 4) [Stage 16:=============================> (2 + 2) res20: Array[Int] = Array(1, 9, 5, 6, 2)  scala> val unionRDD = distinctRdd.distinct(2)  unionRDD: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[40] at distinct at <console>:26  scala> unionRDD.collect()  res21: Array[Int] = Array(6, 2, 1, 9, 5) | | | | | |
| partitionBy | | 对RDD进行分区操作，如果原有的partionRDD和现有的partionRDD是一致的话就不进行分区，  否则会生成ShuffleRDD. | | |
| |  | | --- | | scala> val rdd = sc.parallelize(Array((1,"aaa"),(2,"bbb"),(3,"ccc"),(4,"ddd")),4)  rdd: org.apache.spark.rdd.RDD[(Int, String)] = ParallelCollectionRDD[44] at parallelize at <console>:24  scala> rdd.partitions.size  res24: Int = 4  scala> var rdd2 = rdd.partitionBy(new org.apache.spark.HashPartitioner(2))  rdd2: org.apache.spark.rdd.RDD[(Int, String)] = ShuffledRDD[45] at partitionBy at <console>:26  scala> rdd2.partitions.size  res25: Int = 2 | | | | | |
| 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，reduce任务的个数可以通过第二个可选的参数来设置 |
| |  | | --- | | scala> val rdd = sc.parallelize(List(("female",1),("male",5),("female",5),("male",2)))  rdd: org.apache.spark.rdd.RDD[(String, Int)] = ParallelCollectionRDD[46] at parallelize at <console>:24  scala> val reduce = rdd.reduceByKey((x,y) => x+y)  reduce: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[47] at reduceByKey at <console>:26  scala> reduce.collect()  res29: Array[(String, Int)] = Array((female,6), (male,7)) | | | | | |
| groupByKey | groupByKey也是对每个key进行操作，但只生成一个sequence。 | | | |
| |  | | --- | | scala> val words = Array("one", "two", "two", "three", "three", "three")  words: Array[String] = Array(one, two, two, three, three, three)  scala> val wordPairsRDD = sc.parallelize(words).map(word => (word, 1))  wordPairsRDD: org.apache.spark.rdd.RDD[(String, Int)] = MapPartitionsRDD[4] at map at <console>:26  scala> val group = wordPairsRDD.groupByKey()  group: org.apache.spark.rdd.RDD[(String, Iterable[Int])] = ShuffledRDD[5] at groupByKey at <console>:28  scala> group.collect()  res1: Array[(String, Iterable[Int])] = Array((two,CompactBuffer(1, 1)), (one,CompactBuffer(1)), (three,CompactBuffer(1, 1, 1)))  scala> group.map(t => (t.\_1, t.\_2.sum))  res2: org.apache.spark.rdd.RDD[(String, Int)] = MapPartitionsRDD[6] at map at <console>:31  scala> res2.collect()  res3: Array[(String, Int)] = Array((two,2), (one,1), (three,3))  scala> val map = group.map(t => (t.\_1, t.\_2.sum))  map: org.apache.spark.rdd.RDD[(String, Int)] = MapPartitionsRDD[7] at map at <console>:30  scala> map.collect()  res4: Array[(String, Int)] = Array((two,2), (one,1), (three,3)) | | | | | |
| combineByKey[C](  createCombiner: V => C,  mergeValue: (C, V) => C,  mergeCombiners: (C, C) => C) | 对相同K，把V合并成一个集合.  **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。 | | | |
| |  | | --- | | scala> val scores = Array(("Fred", 88), ("Fred", 95), ("Fred", 91), ("Wilma", 93), ("Wilma", 95), ("Wilma", 98))  scores: Array[(String, Int)] = Array((Fred,88), (Fred,95), (Fred,91), (Wilma,93), (Wilma,95), (Wilma,98))  scala> val input = sc.parallelize(scores)  input: org.apache.spark.rdd.RDD[(String, Int)] = ParallelCollectionRDD[52] at parallelize at <console>:26  scala> val combine = input.combineByKey(  | (v)=>(v,1),  | (acc:(Int,Int),v)=>(acc.\_1+v,acc.\_2+1),  | (acc1:(Int,Int),acc2:(Int,Int))=>(acc1.\_1+acc2.\_1,acc1.\_2+acc2.\_2))  combine: org.apache.spark.rdd.RDD[(String, (Int, Int))] = ShuffledRDD[53] at combineByKey at <console>:28  scala> val result = combine.map{  | case (key,value) => (key,value.\_1/value.\_2.toDouble)}  result: org.apache.spark.rdd.RDD[(String, Double)] = MapPartitionsRDD[54] at map at <console>:30  scala> result.collect()  res33: Array[(String, Double)] = Array((Wilma,95.33333333333333), (Fred,91.33333333333333)) | | | | | |
| aggregateByKey(zeroValue:U,[partitioner: Partitioner])(seqOp: (U, V) => U,combOp: (U, U) => U) | | | | 在kv对的RDD中，，按key将value进行分组合并，合并时，将每个value和初始值作为seq函数的参数，进行计算，返回的结果作为一个新的kv对，然后再将结果按照key进行合并，最后将每个分组的value传递给combine函数进行计算（先将前两个value进行计算，将返回结果和下一个value传给combine函数，以此类推），将key与计算结果作为一个新的kv对输出。  seqOp函数用于在每一个分区中用初始值逐步迭代value，combOp函数用于合并每个分区中的结果 |
| **例如：分一个分区，以key为1的分区为例，0先和3比较得3，3在和2比较得3，3在和4比较得4，所以整个key为1的组最终结果为（1，4），同理，key为2的最终结果为（2，3），key为3的为（3，8）.**  **如果分三个分区，前两个是一个分区，中间两个是一个分区，最后两个是一个分区，第一个分区的最终结果为（1，3），第二个分区为（1，4）（2，3），最后一个分区为（3，8），combine后为 (3,8), (1,7), (2,3)**  **../../../../../../Desktop/New%20Mockup%**   |  | | --- | | scala> val rdd = sc.parallelize(List((1,3),(1,2),(1,4),(2,3),(3,6),(3,8)),3)  rdd: org.apache.spark.rdd.RDD[(Int, Int)] = ParallelCollectionRDD[12] at parallelize at <console>:24  scala> val agg = rdd.aggregateByKey(0)(math.max(\_,\_),\_+\_)  agg: org.apache.spark.rdd.RDD[(Int, Int)] = ShuffledRDD[13] at aggregateByKey at <console>:26  scala> agg.collect()  res7: Array[(Int, Int)] = Array((3,8), (1,7), (2,3))  scala> agg.partitions.size  res8: Int = 3  scala> val rdd = sc.parallelize(List((1,3),(1,2),(1,4),(2,3),(3,6),(3,8)),1)  rdd: org.apache.spark.rdd.RDD[(Int, Int)] = ParallelCollectionRDD[10] at parallelize at <console>:24  scala> val agg = rdd.aggregateByKey(0)(math.max(\_,\_),\_+\_).collect()  agg: Array[(Int, Int)] = Array((1,4), (3,8), (2,3)) | | | | | |
| foldByKey(zeroValue: V)(func: (V, V) => V): RDD[(K, V)] | | | | **aggregateByKey的简化操作，seqop和combop相同** |
| |  | | --- | | scala> val rdd = sc.parallelize(List((1,3),(1,2),(1,4),(2,3),(3,6),(3,8)),3)  rdd: org.apache.spark.rdd.RDD[(Int, Int)] = ParallelCollectionRDD[91] at parallelize at <console>:24  scala> val agg = rdd.foldByKey(0)(\_+\_)  agg: org.apache.spark.rdd.RDD[(Int, Int)] = ShuffledRDD[92] at foldByKey at <console>:26  scala> agg.collect()  res61: Array[(Int, Int)] = Array((3,14), (1,9), (2,3)) | | | | | |
| 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 |
| |  | | --- | | scala> val rdd = sc.parallelize(Array((3,"aa"),(6,"cc"),(2,"bb"),(1,"dd")))  rdd: org.apache.spark.rdd.RDD[(Int, String)] = ParallelCollectionRDD[14] at parallelize at <console>:24  scala> rdd.sortByKey(true).collect()  res9: Array[(Int, String)] = Array((1,dd), (2,bb), (3,aa), (6,cc))  scala> rdd.sortByKey(false).collect()  res10: Array[(Int, String)] = Array((6,cc), (3,aa), (2,bb), (1,dd)) | | | | | |
| sortBy(func,[ascending], [numTasks]) | | | | 与sortByKey类似，但是更灵活,可以用func先对数据进行处理，按照处理后的数据比较结果排序。 |
| |  | | --- | | scala> val rdd = sc.parallelize(List(1,2,3,4))  rdd: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[21] at parallelize at <console>:24  scala> rdd.sortBy(x => x).collect()  res11: Array[Int] = Array(1, 2, 3, 4)  scala> rdd.sortBy(x => x%3).collect()  res12: Array[Int] = Array(3, 4, 1, 2) | | | | | |
| 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 |
| |  | | --- | | scala> val rdd = sc.parallelize(Array((1,"a"),(2,"b"),(3,"c")))  rdd: org.apache.spark.rdd.RDD[(Int, String)] = ParallelCollectionRDD[32] at parallelize at <console>:24  scala> val rdd1 = sc.parallelize(Array((1,4),(2,5),(3,6)))  rdd1: org.apache.spark.rdd.RDD[(Int, Int)] = ParallelCollectionRDD[33] at parallelize at <console>:24  scala> rdd.join(rdd1).collect()  res13: Array[(Int, (String, Int))] = Array((1,(a,4)), (2,(b,5)), (3,(c,6))) | | | | | |
| cogroup(otherDataset, [numTasks]) | | | | 在类型为(K,V)和(K,W)的RDD上调用，返回一个(K,(Iterable<V>,Iterable<W>))类型的RDD |
| |  | | --- | | scala> val rdd = sc.parallelize(Array((1,"a"),(2,"b"),(3,"c")))  rdd: org.apache.spark.rdd.RDD[(Int, String)] = ParallelCollectionRDD[37] at parallelize at <console>:24  scala> val rdd1 = sc.parallelize(Array((1,4),(2,5),(3,6)))  rdd1: org.apache.spark.rdd.RDD[(Int, Int)] = ParallelCollectionRDD[38] at parallelize at <console>:24  scala> rdd.cogroup(rdd1).collect()  res14: Array[(Int, (Iterable[String], Iterable[Int]))] = Array((1,(CompactBuffer(a),CompactBuffer(4))), (2,(CompactBuffer(b),CompactBuffer(5))), (3,(CompactBuffer(c),CompactBuffer(6))))  scala> val rdd2 = sc.parallelize(Array((4,4),(2,5),(3,6)))  rdd2: org.apache.spark.rdd.RDD[(Int, Int)] = ParallelCollectionRDD[41] at parallelize at <console>:24  scala> rdd.cogroup(rdd2).collect()  res15: Array[(Int, (Iterable[String], Iterable[Int]))] = Array((4,(CompactBuffer(),CompactBuffer(4))), (1,(CompactBuffer(a),CompactBuffer())), (2,(CompactBuffer(b),CompactBuffer(5))), (3,(CompactBuffer(c),CompactBuffer(6))))  scala> val rdd3 = sc.parallelize(Array((1,"a"),(1,"d"),(2,"b"),(3,"c")))  rdd3: org.apache.spark.rdd.RDD[(Int, String)] = ParallelCollectionRDD[44] at parallelize at <console>:24  scala> rdd3.cogroup(rdd2).collect()  [Stage 36:> (0 + 0) res16: Array[(Int, (Iterable[String], Iterable[Int]))] = Array((4,(CompactBuffer(),CompactBuffer(4))), (1,(CompactBuffer(d, a),CompactBuffer())), (2,(CompactBuffer(b),CompactBuffer(5))), (3,(CompactBuffer(c),CompactBuffer(6)))) | | | | | |
| cartesian(otherDataset) | | | | **笛卡尔积** |
| |  | | --- | | scala> val rdd1 = sc.parallelize(1 to 3)  rdd1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[47] at parallelize at <console>:24  scala> val rdd2 = sc.parallelize(2 to 5)  rdd2: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[48] at parallelize at <console>:24  scala> rdd1.cartesian(rdd2).collect()  res17: Array[(Int, Int)] = Array((1,2), (1,3), (1,4), (1,5), (2,2), (2,3), (2,4), (2,5), (3,2), (3,3), (3,4), (3,5)) | | | | | |
| pipe(command, [envVars]) | | | | **对于每个分区，都执行一个perl或者shell脚本，返回输出的RDD** |
| **Shell脚本**  #!/bin/sh  echo "AA"  while read LINE; do  echo ">>>"${LINE}  done    **注意：shell脚本需要集群中的所有节点都能访问到。**   |  | | --- | | scala> val rdd = sc.parallelize(List("hi","Hello","how","are","you"),1)  rdd: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[50] at parallelize at <console>:24  scala> rdd.pipe("/home/bigdata/pipe.sh").collect()  res18: Array[String] = Array(AA, >>>hi, >>>Hello, >>>how, >>>are, >>>you)  scala> val rdd = sc.parallelize(List("hi","Hello","how","are","you"),2)  rdd: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[52] at parallelize at <console>:24  scala> rdd.pipe("/home/bigdata/pipe.sh").collect()  res19: Array[String] = Array(AA, >>>hi, >>>Hello, AA, >>>how, >>>are, >>>you)  pipe.sh:  #!/bin/sh  echo "AA"  while read LINE; do  echo ">>>"${LINE}  done | | | | | |
| coalesce(numPartitions) | | | | **缩减分区数，用于大数据集过滤后，提高小数据集的执行效率。** |
| |  | | --- | | scala> val rdd = sc.parallelize(1 to 16,4)  rdd: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[54] at parallelize at <console>:24  scala> rdd.partitions.size  res20: Int = 4  scala> val coalesceRDD = rdd.coalesce(3)  coalesceRDD: org.apache.spark.rdd.RDD[Int] = CoalescedRDD[55] at coalesce at <console>:26  scala> coalesceRDD.partitions.size  res21: Int = 3 | | | | | |
| repartition(numPartitions) | | | | 根据分区数，从新通过网络随机洗牌所有数据。 |
| |  | | --- | | scala> val rdd = sc.parallelize(1 to 16,4)  rdd: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[56] at parallelize at <console>:24  scala> rdd.partitions.size  res22: Int = 4  scala> val rerdd = rdd.repartition(2)  rerdd: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[60] at repartition at <console>:26  scala> rerdd.partitions.size  res23: Int = 2  scala> val rerdd = rdd.repartition(4)  rerdd: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[64] at repartition at <console>:26  scala> rerdd.partitions.size  res24: Int = 4 | | | | | |
| repartitionAndSortWithinPartitions(partitioner) | | | | repartitionAndSortWithinPartitions函数是repartition函数的变种，与repartition函数不同的是，repartitionAndSortWithinPartitions在给定的partitioner内部进行排序，性能比repartition要高。 |
|  | | | |  |
| glom | | | | 将每一个分区形成一个数组，形成新的RDD类型时RDD[Array[T]] |
| |  | | --- | | scala> val rdd = sc.parallelize(1 to 16,4)  rdd: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[65] at parallelize at <console>:24  scala> rdd.glom().collect()  res25: Array[Array[Int]] = Array(Array(1, 2, 3, 4), Array(5, 6, 7, 8), Array(9, 10, 11, 12), Array(13, 14, 15, 16)) | | | | | |
| mapValues | | | 针对于(K,V)形式的类型只对V进行操作 | |
| |  | | --- | | scala> val rdd3 = sc.parallelize(Array((1,"a"),(1,"d"),(2,"b"),(3,"c")))  rdd3: org.apache.spark.rdd.RDD[(Int, String)] = ParallelCollectionRDD[67] at parallelize at <console>:24  scala> rdd3.mapValues(\_+"|||").collect()  res26: Array[(Int, String)] = Array((1,a|||), (1,d|||), (2,b|||), (3,c|||)) | | | | | |
| subtract | | | 计算差的一种函数去除两个RDD中相同的元素，不同的RDD将保留下来 | |
| |  | | --- | | scala> val rdd = sc.parallelize(3 to 8)  rdd: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[70] at parallelize at <console>:24  scala> val rdd1 = sc.parallelize(1 to 5)  rdd1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[71] at parallelize at <console>:24  scala> rdd.subtract(rdd1).collect()  res27: Array[Int] = Array(8, 6, 7) | | | | | |
|  | | | | |

### Action

**常用的Action：**

|  |  |
| --- | --- |
| **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** |
| |  | | --- | | scala> val rdd1 = sc.makeRDD(1 to 10,2)  rdd1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[85] at makeRDD at <console>:24  scala> rdd1.reduce(\_+\_)  res50: Int = 55  scala> val rdd2 = sc.makeRDD(Array(("a",1),("a",3),("c",3),("d",5)))  rdd2: org.apache.spark.rdd.RDD[(String, Int)] = ParallelCollectionRDD[86] at makeRDD at <console>:24  scala> rdd2.reduce((x,y)=>(x.\_1 + y.\_1,x.\_2 + y.\_2))  res51: (String, Int) = (adca,12) | | |
| collect() | **在驱动程序中，以数组的形式返回数据集的所有元素** |
| **../../../../../../Desktop/20161123221808041.** | |
| 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) | **返回前几个的排序** |
|  | |
| aggregate (zeroValue: U)(seqOp: (U, T) ⇒ U, combOp: (U, U) ⇒ U) | **aggregate函数将每个分区里面的元素通过seqOp和初始值进行聚合，然后用combine函数将每个分区的结果和初始值(zeroValue)进行combine操作。这个函数最终返回的类型不需要和RDD中元素类型一致。** |
| **../../../../../../Desktop/New%20Mockup%**   |  | | --- | | scala> var rdd1 = sc.makeRDD(1 to 10,2)  rdd1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[88] at makeRDD at <console>:24  scala> rdd1.aggregate(1)(  | {(x : Int,y : Int) => x + y},  | {(a : Int,b : Int) => a + b}  | )  res56: Int = 58  scala> rdd1.aggregate(1)(  | {(x : Int,y : Int) => x \* y},  | {(a : Int,b : Int) => a + b}  | )  res57: Int = 30361 | | |
| fold(num)(func) | 折叠操作，aggregate的简化操作，seqop和combop一样。 |
| |  | | --- | | scala> var rdd1 = sc.makeRDD(1 to 4,2)  rdd1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[90] at makeRDD at <console>:24  scala> rdd1.aggregate(1)(  | {(x : Int,y : Int) => x + y},  | {(a : Int,b : Int) => a + b}  | )  res59: Int = 13  scala> rdd1.fold(1)(\_+\_)  res60: Int = 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** |
|  | |
| 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) | **用于将RDD中的元素序列化成对象，存储到文件中。** |
|  | |
| 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 rdd = sc.parallelize(List((1,3),(1,2),(1,4),(2,3),(3,6),(3,8)),3)  rdd: org.apache.spark.rdd.RDD[(Int, Int)] = ParallelCollectionRDD[95] at parallelize at <console>:24  scala> rdd.countByKey()  res63: scala.collection.Map[Int,Long] = Map(3 -> 2, 1 -> 3, 2 -> 1) | | |
| foreach(func) | **在数据集的每一个元素上，运行函数func进行更新。** |
| **../../../../../../Desktop/20161123221819674.**   |  | | --- | | scala> var rdd = sc.makeRDD(1 to 10,2)  rdd: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[107] at makeRDD at <console>:24  scala> var sum = sc.accumulator(0)  warning: there were two deprecation warnings; re-run with -deprecation for details  sum: org.apache.spark.Accumulator[Int] = 0  scala> rdd.foreach(sum+=\_)  scala> sum.value  res68: Int = 55  scala> rdd.collect().foreach(println)  1  2  3  4  5  6  7  8  9  10 | | |

### 数值RDD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 |

### 向RDD操作传递函数注意

Spark 的大部分转化操作和一部分行动操作，都需要依赖用户传递的函数来计算。 在 Scala 中，我们可以把定义的内联函数、方法的引用或静态方法传递给 Spark，就像 Scala 的其他函数式 API 一样。我们还要考虑其他一些细节，比如所传递的函数及其引用 的数据需要是可序列化的(实现了 Java 的 Serializable 接口)。 传递一个对象的方法或者字段时，会包含对整个对象的引用。

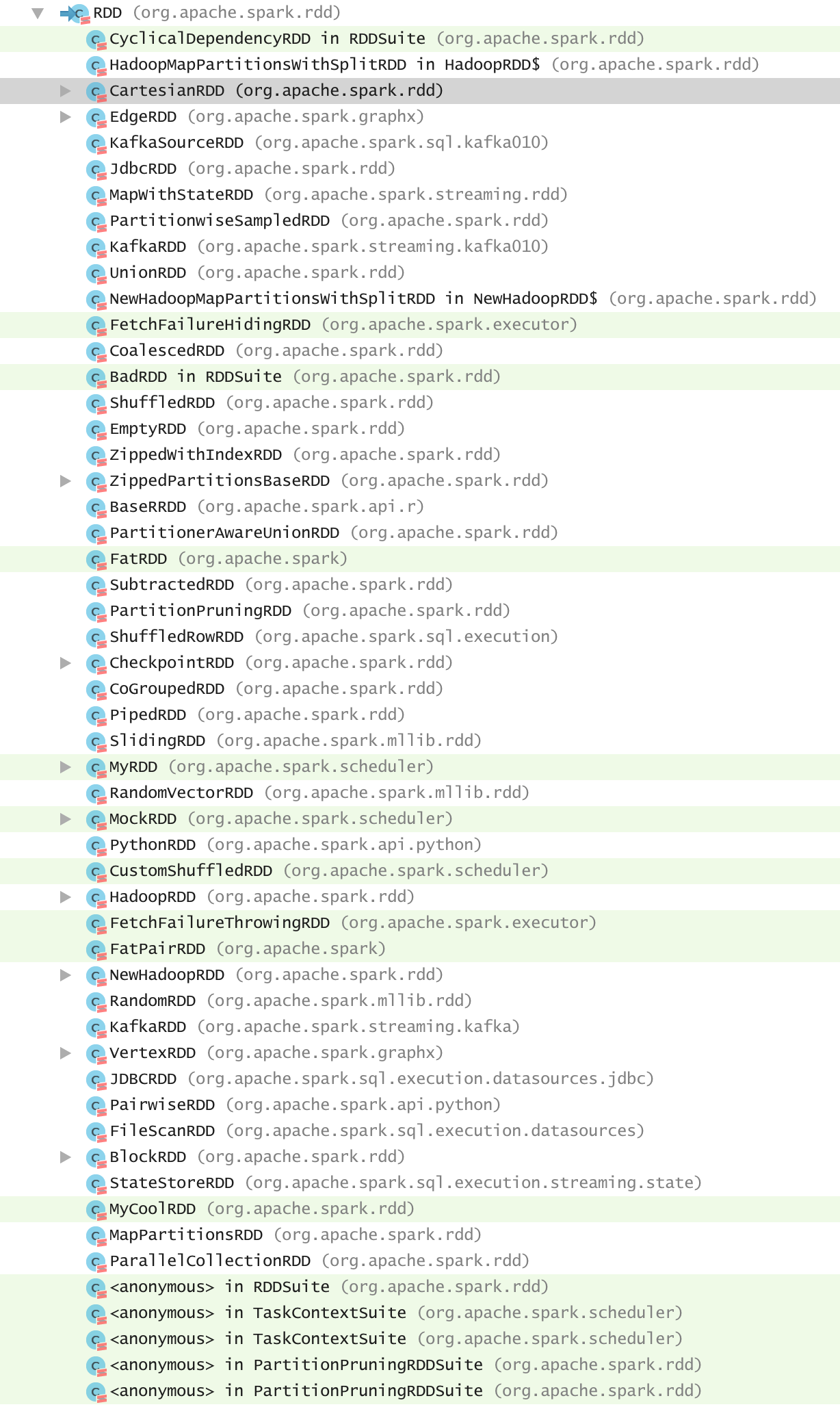
|  |
| --- |
| **class** SearchFunctions(**val** query: String) extends java.io.Serializable{  **def** isMatch(s: String): Boolean = {  s.contains(query)  }  **def** getMatchesFunctionReference(rdd: org.apache.spark.rdd.RDD[String]): org.apache.spark.rdd.RDD[String] = {  *// 问题:"isMatch"表示"this.isMatch"，因此我们要传递整个"this"* rdd.filter(isMatch)  }  **def** getMatchesFieldReference(rdd: org.apache.spark.rdd.RDD[String]): org.apache.spark.rdd.RDD[String] = {  *// 问题:"query"表示"this.query"，因此我们要传递整个"this"*  *rdd.filter(x => x.*contains*(query))* }  **def** getMatchesNoReference(rdd: org.apache.spark.rdd.RDD[String]): org.apache.spark.rdd.RDD[String] = {  *// 安全:只把我们需要的字段拿出来放入局部变量中*  *val query\_ = this.query* rdd.filter(x => x.contains(query\_))  }  } |

如果在 Scala 中出现了 NotSerializableException，通常问题就在于我们传递了一个不可序列 化的类中的函数或字段。

### 在不同RDD类型间转换

有些函数只能用于特定类型的 RDD，比如 mean() 和 variance() 只能用在数值 RDD 上， 而 join() 只能用在键值对 RDD 上。在 Scala 和 Java 中，这些函数都没有定义在标准的 RDD 类中，所以要访问这些附加功能，必须要确保获得了正确的专用 RDD 类。

在 Scala 中，将 RDD 转为有特定函数的 RDD(比如在 RDD[Double] 上进行数值操作)是 由隐式转换来自动处理的。



## RDD持久化

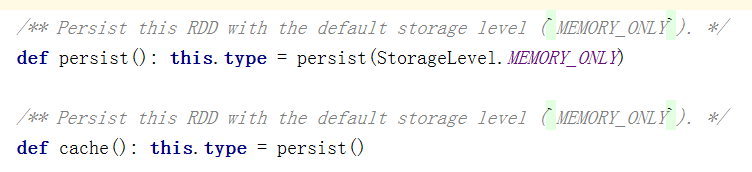
### RDD的缓存

Spark速度非常快的原因之一，就是在不同操作中可以在内存中持久化或缓存个数据集。当持久化某个RDD后，每一个节点都将把计算的分片结果保存在内存中，并在对此RDD或衍生出的RDD进行的其他动作中重用。这使得后续的动作变得更加迅速。RDD相关的持久化和缓存，是Spark最重要的特征之一。可以说，缓存是Spark构建迭代式算法和快速交互式查询的关键。如果一个有持久化数据的节点发生故障，Spark 会在需要用到缓存的数据时重算丢失的数据分区。如果 希望节点故障的情况不会拖累我们的执行速度，也可以把数据备份到多个节点上。

### RDD缓存方式

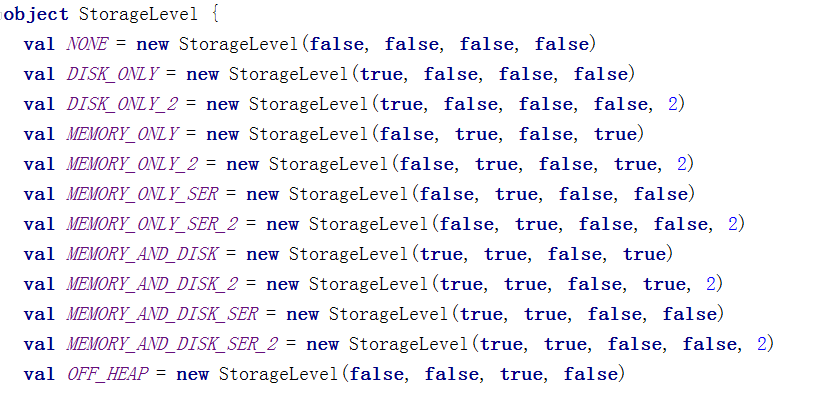
RDD通过persist方法或cache方法可以将前面的计算结果缓存，默认情况下 persist() 会把数据以序列化的形式缓存在 JVM 的堆空 间中。

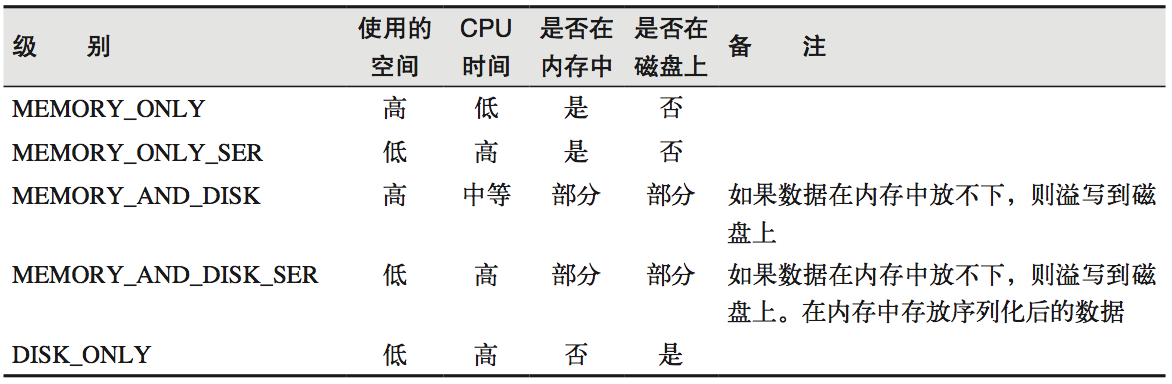
但是并不是这两个方法被调用时立即缓存，而是触发后面的action时，该RDD将会被缓存在计算节点的内存中，并供后面重用。



通过查看源码发现cache最终也是调用了persist方法，默认的存储级别都是仅在内存存储一份，Spark的存储级别还有好多种，存储级别在object StorageLevel中定义的。

|  |
| --- |
| scala> val rdd = sc.makeRDD(1 to 10)  rdd: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[19] at makeRDD at <console>:25  scala> val nocache = rdd.map(\_.toString+"["+System.currentTimeMillis+"]")  nocache: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[20] at map at <console>:27  scala> val cache = rdd.map(\_.toString+"["+System.currentTimeMillis+"]")  cache: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[21] at map at <console>:27  scala> cache.cache  res24: cache.type = MapPartitionsRDD[21] at map at <console>:27  scala> nocache.collect  res25: Array[String] = Array(1[1505479375155], 2[1505479374674], 3[1505479374674], 4[1505479375153], 5[1505479375153], 6[1505479374675], 7[1505479375154], 8[1505479375154], 9[1505479374676], 10[1505479374676])  scala> nocache.collect  res26: Array[String] = Array(1[1505479375679], 2[1505479376157], 3[1505479376157], 4[1505479375680], 5[1505479375680], 6[1505479376159], 7[1505479375680], 8[1505479375680], 9[1505479376158], 10[1505479376158])  scala> nocache.collect  res27: Array[String] = Array(1[1505479376743], 2[1505479377218], 3[1505479377218], 4[1505479376745], 5[1505479376745], 6[1505479377219], 7[1505479376747], 8[1505479376747], 9[1505479377218], 10[1505479377218])  scala> cache.collect  res28: Array[String] = Array(1[1505479382745], 2[1505479382253], 3[1505479382253], 4[1505479382748], 5[1505479382748], 6[1505479382257], 7[1505479382747], 8[1505479382747], 9[1505479382253], 10[1505479382253])  scala> cache.collect  res29: Array[String] = Array(1[1505479382745], 2[1505479382253], 3[1505479382253], 4[1505479382748], 5[1505479382748], 6[1505479382257], 7[1505479382747], 8[1505479382747], 9[1505479382253], 10[1505479382253])  scala> cache.collect  res30: Array[String] = Array(1[1505479382745], 2[1505479382253], 3[1505479382253], 4[1505479382748], 5[1505479382748], 6[1505479382257], 7[1505479382747], 8[1505479382747], 9[1505479382253], 10[1505479382253])  cache.persist(org.apache.spark.storage.StorageLevel.MEMORY\_ONLY) |



在存储级别的末尾加上“\_2”来把持久化数据存为两份 

缓存有可能丢失，或者存储存储于内存的数据由于内存不足而被删除，RDD的缓存容错机制保证了即使缓存丢失也能保证计算的正确执行。通过基于RDD的一系列转换，丢失的数据会被重算，由于RDD的各个Partition是相对独立的，因此只需要计算丢失的部分即可，并不需要重算全部Partition。

注意：使用 Tachyon可以实现堆外缓存

## RDD检查点机制

Spark中对于数据的保存除了持久化操作之外，还提供了一种检查点的机制，检查点（本质是通过将RDD写入Disk做检查点）是为了通过lineage做容错的辅助，lineage过长会造成容错成本过高，这样就不如在中间阶段做检查点容错，如果之后有节点出现问题而丢失分区，从做检查点的RDD开始重做Lineage，就会减少开销。检查点通过将数据写入到HDFS文件系统实现了RDD的检查点功能。

cache 和 checkpoint 是有显著区别的，  缓存把 RDD 计算出来然后放在内存中，但是RDD 的依赖链（相当于数据库中的redo 日志）， 也不能丢掉， 当某个点某个 executor 宕了，上面cache 的RDD就会丢掉， 需要通过 依赖链重放计算出来， 不同的是， checkpoint 是把 RDD 保存在 HDFS中， 是多副本可靠存储，所以依赖链就可以丢掉了，就斩断了依赖链， 是通过复制实现的高容错。

如果存在以下场景，则比较适合使用检查点机制：

1. DAG中的Lineage过长，如果重算，则开销太大（如在PageRank中）。
2. 在宽依赖上做Checkpoint获得的收益更大。

为当前RDD设置检查点。该函数将会创建一个二进制的文件，并存储到checkpoint目录中，该目录是用[Spark](https://www.iteblog.com/archives/tag/spark/" \t "_blank)Context.setCheckpointDir()设置的。在checkpoint的过程中，该RDD的所有依赖于父RDD中的信息将全部被移出。对RDD进行checkpoint操作并不会马上被执行，必须执行Action操作才能触发。

|  |
| --- |
| scala> val data = sc.parallelize(1 to 100 , 5)  data: org.apache.spark.rdd.RDD[Int] =  　　ParallelCollectionRDD[12] at parallelize at <console>:12    scala> sc.setCheckpointDir("hdfs://master01:9000/checkpoint")    scala> data.checkpoint    scala> data.count  scala> val ch1 = sc.parallelize(1 to 2)  ch1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[33] at parallelize at <console>:25  scala> val ch2 = ch1.map(\_.toString+"["+System.currentTimeMillis+"]")  ch2: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[36] at map at <console>:27  scala> val ch3 = ch1.map(\_.toString+"["+System.currentTimeMillis+"]")  ch3: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[37] at map at <console>:27  scala> ch3.checkpoint  scala> ch2.collect  res62: Array[String] = Array(1[1505480940726], 2[1505480940243])  scala> ch2.collect  res63: Array[String] = Array(1[1505480941957], 2[1505480941480])  scala> ch2.collect  res64: Array[String] = Array(1[1505480942736], 2[1505480942257])  scala> ch3.collect  res65: Array[String] = Array(1[1505480949080], 2[1505480948603])  scala> ch3.collect  res66: Array[String] = Array(1[1505480948683], 2[1505480949161])  scala> ch3.collect  res67: Array[String] = Array(1[1505480948683], 2[1505480949161]) |

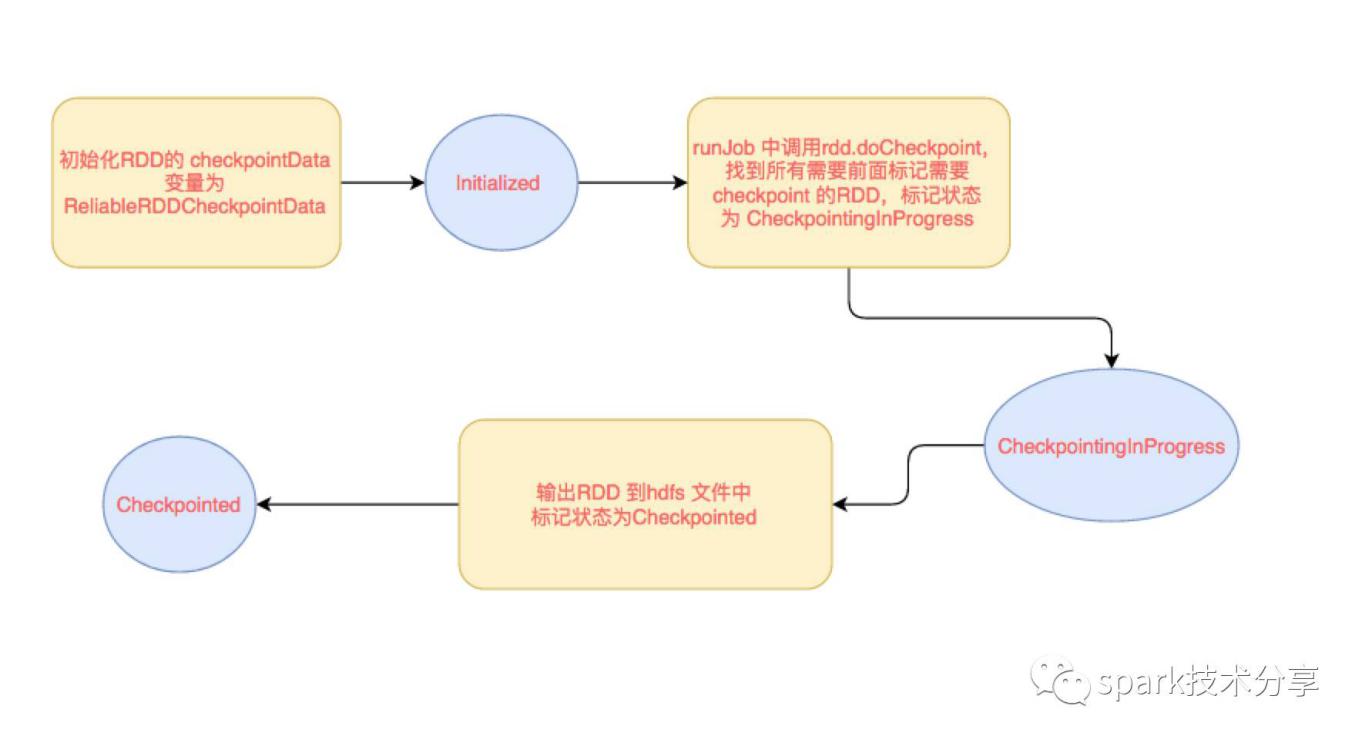


### checkpoint 写流程

RDD checkpoint 过程中会经过以下几个状态，

[ Initialized → marked for checkpointing → checkpointing in progress → checkpointed ]

转换流程如下



1. data.checkpoint 这个函数调用中， 设置的目录中， 所有依赖的 RDD 都会被删除， 函数必须在 job 运行之前调用执行， 强烈建议 RDD 缓存 在内存中（又提到一次，千万要注意哟）， 否则保存到文件的时候需要从头计算。初始化RDD的 checkpointData 变量为 ReliableRDDCheckpointData。  这时候标记为 Initialized 状态，
2. 在所有 job action 的时候， runJob 方法中都会调用 rdd.doCheckpoint ,  这个会向前递归调用所有的依赖的RDD， 看看需不需要  checkpoint。 需要需要 checkpoint， 然后调用  checkpointData.get.checkpoint()， 里面标记 状态为 CheckpointingInProgress，  里面调用具体实现类的 ReliableRDDCheckpointData 的 doCheckpoint 方法，
3. doCheckpoint -> writeRDDToCheckpointDirectory， 注意这里会把 job 再运行一次， 如果已经cache 了，就可以直接使用缓存中的 RDD 了， 就不需要重头计算一遍了（怎么又说了一遍），  这时候直接把RDD， 输出到 hdfs， 每个分区一个文件， 会先写到一个临时文件， 如果全部输出完，进行 rename ， 如果输出失败，就回滚delete。
4. 标记 状态为 Checkpointed， markCheckpointed方法中清除所有的依赖， 怎么清除依赖的呢， 就是 吧RDD 变量的强引用 设置为 null， 垃圾回收了，会触发 ContextCleaner 里面监听清除实际 BlockManager 缓存中的数据

### checkpoint 读流程

如果一个RDD 我们已经 checkpoint了那么是什么时候用呢， checkpoint 将 RDD 持久化到 HDFS 或本地文件夹，如果不被手动 remove 掉，是一直存在的，也就是说可以被下一个 driver program 使用。 比如 spark streaming 挂掉了， 重启后就可以使用之前 checkpoint 的数据进行 recover （这个流程我们在下面一篇文章会讲到） ，  当然在同一个 driver program 也可以使用。 我们讲下在同一个 driver program 中是怎么使用 checkpoint 数据的。

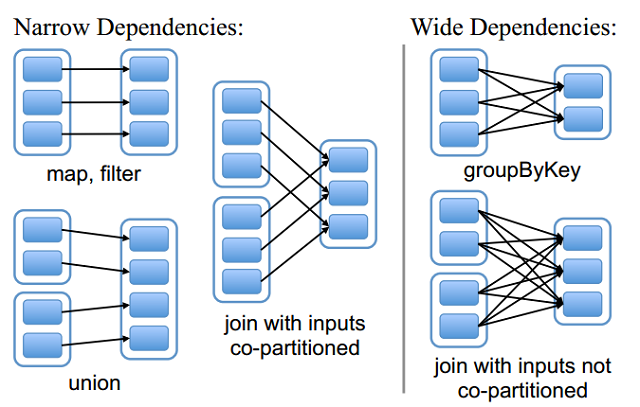
如果 一个 RDD 被checkpoint了， 如果这个 RDD 上有 action 操作时候，或者回溯的这个 RDD 的时候,这个 RDD 进行计算的时候，里面判断如果已经 checkpoint 过,  对分区和依赖的处理都是使用的 RDD 内部的 checkpointRDD 变量。

具体细节如下，

如果 一个 RDD 被checkpoint了，  那么这个 RDD 中对分区和依赖的处理都是使用的 RDD 内部的 checkpointRDD 变量， 具体实现是  ReliableCheckpointRDD 类型。 这个是在 checkpoint 写流程中创建的。依赖和获取分区方法中先判断是否已经checkpoint， 如果已经checkpoint了， 就斩断依赖，  使用ReliableCheckpointRDD， 来处理依赖和获取分区。  
 如果没有，才往前回溯依赖。  依赖就是没有依赖， 因为已经斩断了依赖， 获取分区数据就是读取 checkpoint 到 hdfs目录中不同分区保存下来的文件。

## RDD的依赖关系

RDD和它依赖的父RDD（s）的关系有两种不同的类型，即窄依赖（narrow dependency）和宽依赖（wide dependency）。



### 窄依赖

窄依赖指的是每一个父RDD的Partition最多被子RDD的一个Partition使用

总结：窄依赖我们形象的比喻为**独生子女**

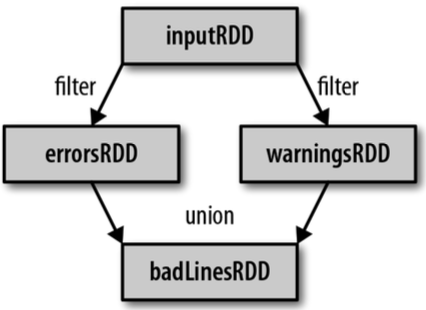
### 宽依赖

宽依赖指的是多个子RDD的Partition会依赖同一个父RDD的Partition，会引起shuffle

总结：宽依赖我们形象的比喻为**超生**

### Lineage

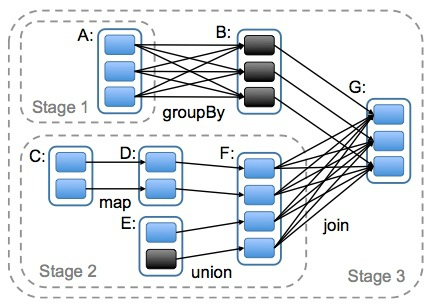
RDD只支持粗粒度转换，即在大量记录上执行的单个操作。将创建RDD的一系列Lineage（即血统）记录下来，以便恢复丢失的分区。RDD的Lineage会记录RDD的元数据信息和转换行为，当该RDD的部分分区数据丢失时，它可以根据这些信息来重新运算和恢复丢失的数据分区。



|  |
| --- |
| scala> val text = sc.textFile("README.md")  text: org.apache.spark.rdd.RDD[String] = README.md MapPartitionsRDD[1] at textFile at <console>:24  scala> val words = text.flatMap(\_.split)  split splitAt  scala> val words = text.flatMap(\_.split(" "))  words: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[2] at flatMap at <console>:26  scala> words.map((\_,1))  res0: org.apache.spark.rdd.RDD[(String, Int)] = MapPartitionsRDD[3] at map at <console>:29  scala> res0.reduceByKey  reduceByKey reduceByKeyLocally  scala> res0.reduceByKey(\_+\_)  res1: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[4] at reduceByKey at <console>:31  scala> res1.dependencies  res2: Seq[org.apache.spark.Dependency[\_]] = List(org.apache.spark.ShuffleDependency@6cfe48a4)  scala> res0.dependencies  res3: Seq[org.apache.spark.Dependency[\_]] = List(org.apache.spark.OneToOneDependency@6c9e24c4) |

## DAG的生成

DAG(Directed Acyclic Graph)叫做有向无环图，原始的RDD通过一系列的转换就就形成了DAG，根据RDD之间的依赖关系的不同将DAG划分成不同的Stage，对于窄依赖，partition的转换处理在Stage中完成计算。对于宽依赖，由于有Shuffle的存在，只能在parent RDD处理完成后，才能开始接下来的计算，因此宽依赖是划分Stage的依据。



## RDD相关概念关系



输入可能以多个文件的形式存储在HDFS上，每个File都包含了很多块，称为Block。

当Spark读取这些文件作为输入时，会根据具体数据格式对应的InputFormat进行解析，一般是将若干个Block合并成一个输入分片，称为InputSplit，注意InputSplit不能跨越文件。  
随后将为这些输入分片生成具体的Task。InputSplit与Task是一一对应的关系。

随后这些具体的Task每个都会被分配到集群上的某个节点的某个Executor去执行。

1. 每个节点可以起一个或多个Executor。
2. 每个Executor由若干core组成，每个Executor的每个core一次只能执行一个Task。
3. 每个Task执行的结果就是生成了目标RDD的一个partiton。

注意: 这里的core是虚拟的core而不是机器的物理CPU核，可以理解为就是Executor的一个工作线程。

而 Task被执行的并发度 = Executor数目 \* 每个Executor核数。

至于partition的数目：

1. 对于数据读入阶段，例如sc.textFile，输入文件被划分为多少InputSplit就会需要多少初始Task。
2. 在Map阶段partition数目保持不变。
3. 在Reduce阶段，RDD的聚合会触发shuffle操作，聚合后的RDD的partition数目跟具体操作有关，例如repartition操作会聚合成指定分区数，还有一些算子是可配置的。

RDD在计算的时候，每个分区都会起一个task，所以rdd的分区数目决定了总的的task数目。

申请的计算节点（Executor）数目和每个计算节点核数，决定了你同一时刻可以并行执行的task。

比如的RDD有100个分区，那么计算的时候就会生成100个task，你的资源配置为10个计算节点，每个2个核，同一时刻可以并行的task数目为20，计算这个RDD就需要5个轮次。

如果计算资源不变，你有101个task的话，就需要6个轮次，在最后一轮中，只有一个task在执行，其余核都在空转。

如果资源不变，你的RDD只有2个分区，那么同一时刻只有2个task运行，其余18个核空转，造成资源浪费。这就是在spark调优中，增大RDD分区数目，增大任务并行度的做法。

# 键值对RDD

键值对 RDD 是 Spark 中许多操作所需要的常见数据类型。 本章做特别讲解。除了在基础RDD类中定义的操作之外，Spark 为包含键值对类型的 RDD 提供了一些专有的操作 在PairRDDFunctions专门进行了定义。这些 RDD 被称为 pair RDD。

有很多种方式创建pair RDD，在输入输出章节会讲解。一般如果从一个普通的RDD转 为pair RDD时，可以调用map()函数来实现，传递的函数需要返回键值对。

|  |
| --- |
| **val** pairs = lines.map(x => (x.split(**" "**)(0), x)) |

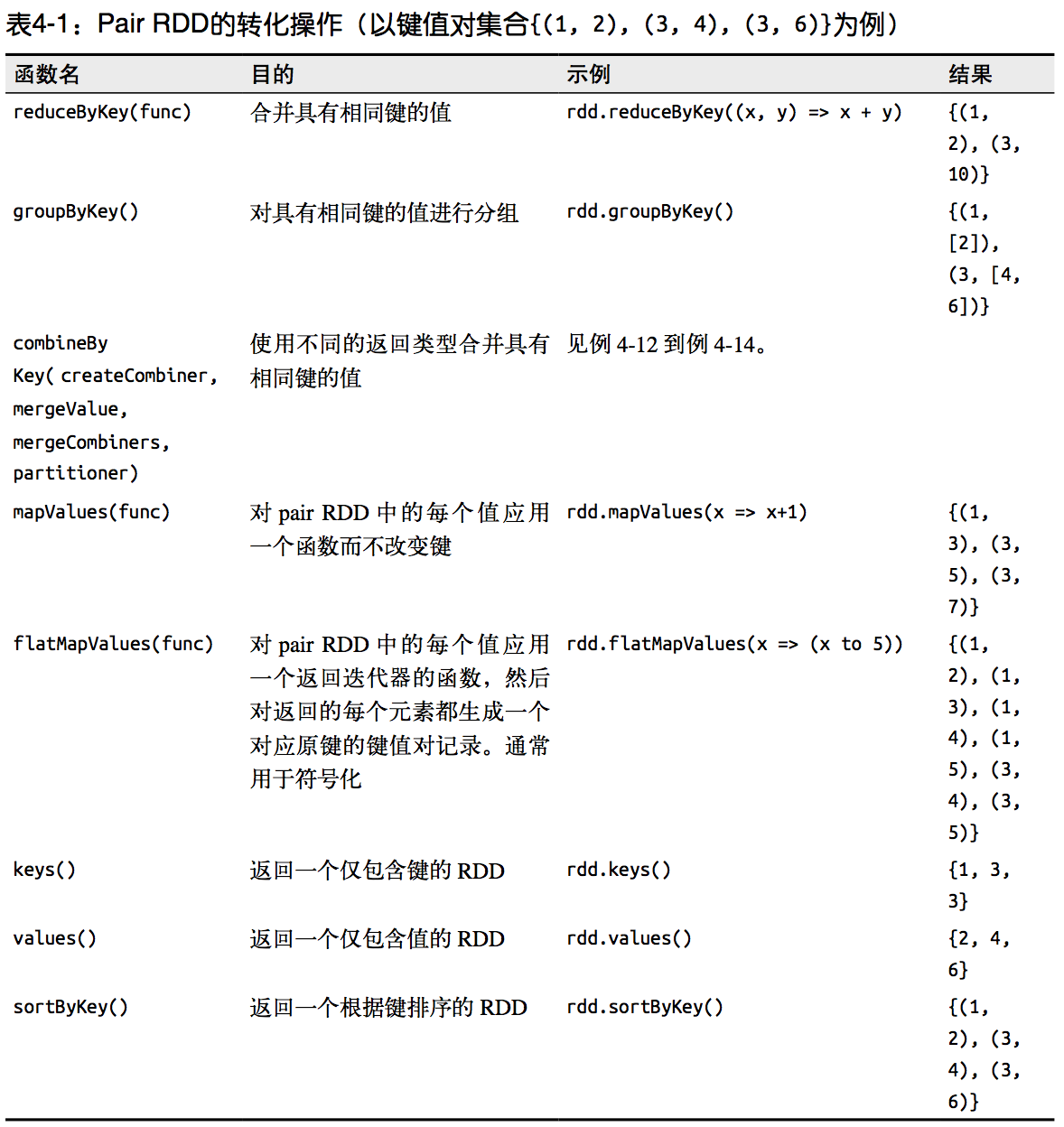
 

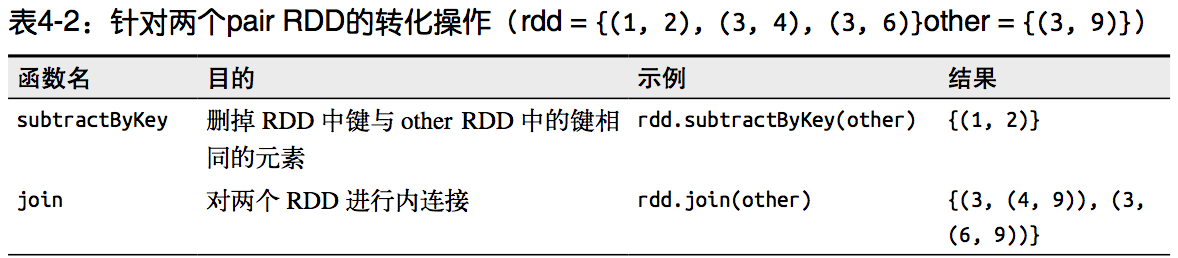
## 键值对RDD的转化操作

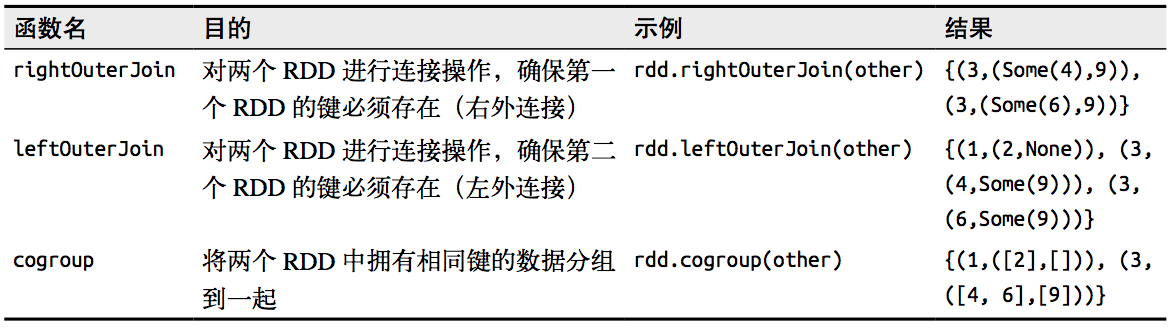
### 转化操作列表

上一章进行了练习，这一章会重点讲解。

针对一个Pair RDD的转化操作







### 聚合操作

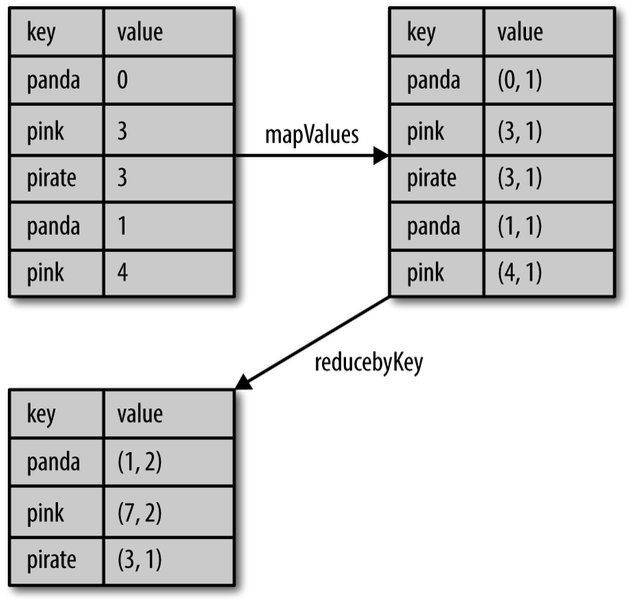
当数据集以键值对形式组织的时候，聚合具有相同键的元素进行一些统计是很常见的操 作。之前讲解过基础RDD上的fold()、combine()、reduce()等行动操作，pair RDD上则 有相应的针对键的转化操作。Spark 有一组类似的操作，可以组合具有相同键的值。这些 操作返回 RDD，因此它们是转化操作而不是行动操作。

reduceByKey() 与 reduce() 相当类似;它们都接收一个函数，并使用该函数对值进行合并。 reduceByKey() 会为数据集中的每个键进行并行的归约操作，每个归约操作会将键相同的值合 并起来。因为数据集中可能有大量的键，所以 reduceByKey() 没有被实现为向用户程序返回一 个值的行动操作。实际上，它会返回一个由各键和对应键归约出来的结果值组成的新的 RDD。

foldByKey() 则与 fold() 相当类似;它们都使用一个与 RDD 和合并函数中的数据类型相 同的零值作为初始值。与 fold() 一样，foldByKey() 操作所使用的合并函数对零值与另一 个元素进行合并，结果仍为该元素。

求均值操作：版本一

|  |
| --- |
| input.mapValues(x => (x, 1)).reduceByKey((x, y) => (x.\_1 + y.\_1, x.\_2 + y.\_2)).map{ **case** (key, value) => (key, value.\_1 / value.\_2.toFloat) } |



combineByKey() 是最为常用的基于键进行聚合的函数。大多数基于键聚合的函数都是用它 实现的。和 aggregate() 一样，combineByKey() 可以让用户返回与输入数据的类型不同的 返回值。

要理解 combineByKey()，要先理解它在处理数据时是如何处理每个元素的。由于 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。需要注意的是，这一过程会在每个分区中第一次出现各个 键时发生，而不是在整个 RDD 中第一次出现一个键时发生。

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。

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。

求均值：版本二

|  |
| --- |
| **val** result = input.combineByKey(  (v) => (v, 1),  (acc: (Int, Int), v) => (acc.\_1 + v, acc.\_2 + 1),  (acc1: (Int, Int), acc2: (Int, Int)) => (acc1.\_1 + acc2.\_1, acc1.\_2 + acc2.\_2) ).map{ **case** (key, value) => (key, value.\_1 / value.\_2.toFloat) }  result.collectAsMap().map(println(\_)) |

### 数据分组

如果数据已经以预期的方式提取了键，groupByKey() 就会使用 RDD 中的键来对数据进行 分组。对于一个由类型 K 的键和类型 V 的值组成的 RDD，所得到的结果 RDD 类型会是 [K, Iterable[V]]。

groupBy() 可以用于未成对的数据上，也可以根据除键相同以外的条件进行分组。它可以 接收一个函数，对源 RDD 中的每个元素使用该函数，将返回结果作为键再进行分组。

多个RDD分组，可以使用cogroup函数，cogroup() 的函数对多个共享同 一个键的 RDD 进行分组。对两个键的类型均为 K 而值的类型分别为 V 和 W 的 RDD 进行 cogroup() 时，得到的结果 RDD 类型为 [(K, (Iterable[V], Iterable[W]))]。如果其中的 一个 RDD 对于另一个 RDD 中存在的某个键没有对应的记录，那么对应的迭代器则为空。 cogroup() 提供了为多个 RDD 进行数据分组的方法。

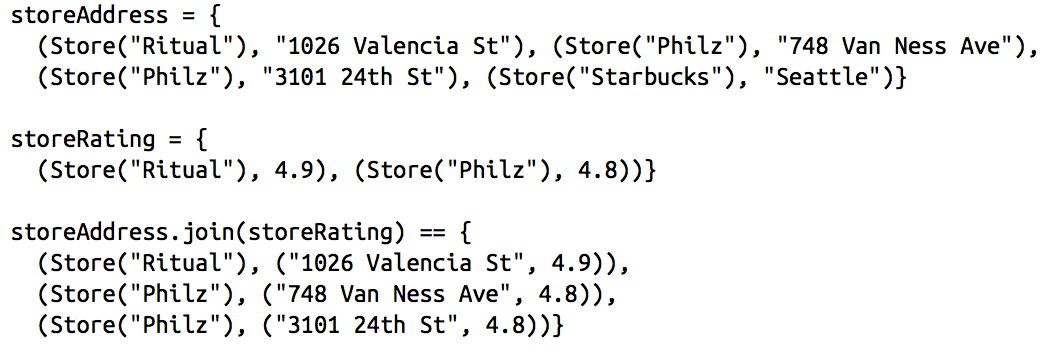
### 连接

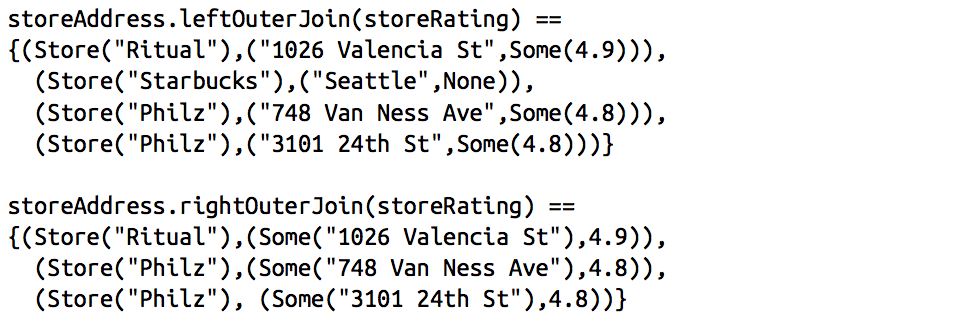
连接主要用于多个Pair RDD的操作，连接方式多种多样:右外连接、左外连接、交 叉连接以及内连接。

普通的 join 操作符表示内连接 2。只有在两个 pair RDD 中都存在的键才叫输出。当一个输 入对应的某个键有多个值时，生成的pair RDD会包括来自两个输入RDD的每一组相对应 的记录。

leftOuterJoin()产生的pair RDD中，源RDD的每一个键都有对应的记录。每个 键相应的值是由一个源 RDD 中的值与一个包含第二个 RDD 的值的 Option(在 Java 中为 Optional)对象组成的二元组。

rightOuterJoin() 几乎与 leftOuterJoin() 完全一样，只不过预期结果中的键必须出现在 第二个 RDD 中，而二元组中的可缺失的部分则来自于源 RDD 而非第二个 RDD。

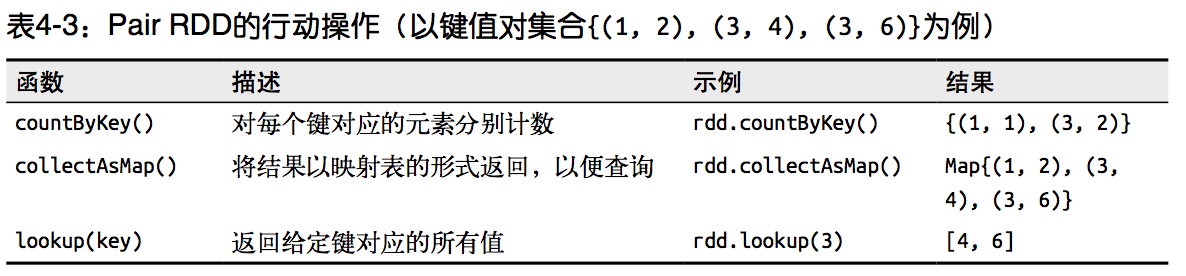




### 数据排序

sortByKey() 函数接收一个叫作 ascending 的参数，表示我们是否想要让结果按升序排序(默认值为 true)。

## 键值对RDD的行动操作



## 键值对RDD的数据分区

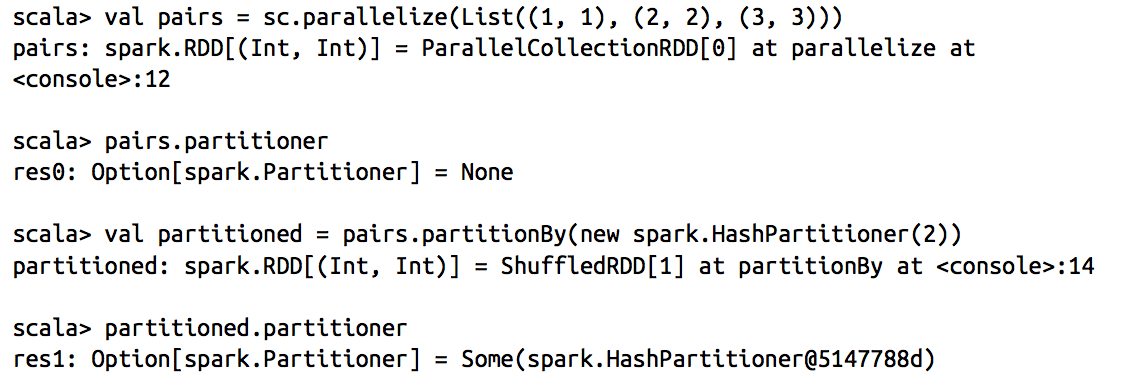
Spark目前支持Hash分区和Range分区，用户也可以自定义分区，Hash分区为当前的默认分区，Spark中分区器直接决定了RDD中分区的个数、RDD中每条数据经过Shuffle过程属于哪个分区和Reduce的个数

注意：

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

### 获取RDD的分区方式

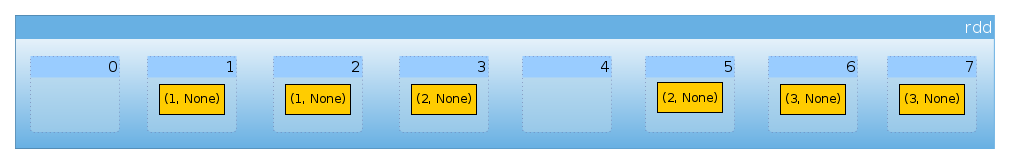
可以通过使用RDD的partitioner 属性来获取 RDD 的分区方式。它会返回一个 scala.Option 对象， 通过get方法获取其中的值。

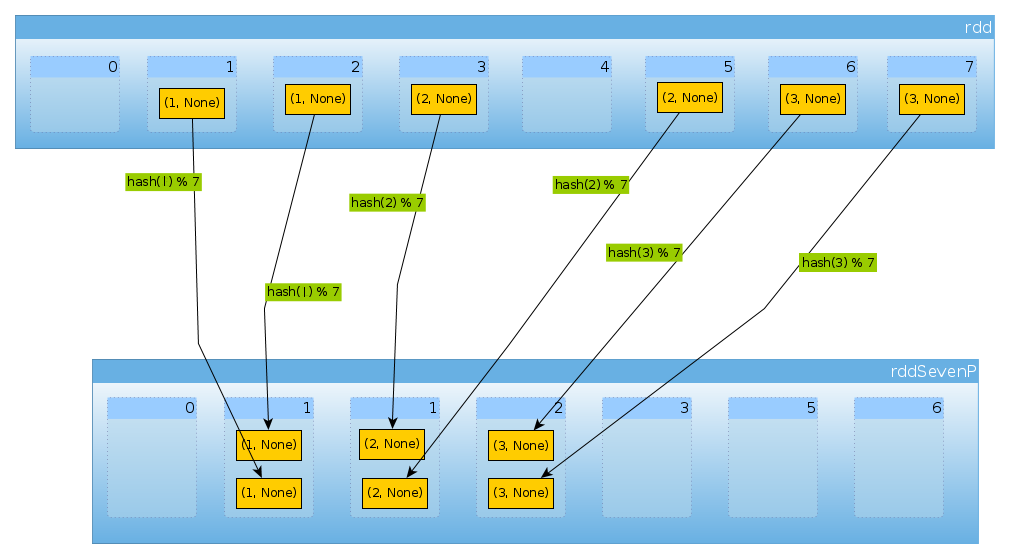


### Hash分区方式

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

|  |
| --- |
| scala> nopar.partitioner  res20: Option[org.apache.spark.Partitioner] = None  scala> val nopar = sc.parallelize(List((1,3),(1,2),(2,4),(2,3),(3,6),(3,8)),8)  nopar: org.apache.spark.rdd.RDD[(Int, Int)] = ParallelCollectionRDD[10] at parallelize at <console>:24  scala>nopar.mapPartitionsWithIndex((index,iter)=>{ Iterator(index.toString+" : "+iter.mkString("|")) }).collect  res0: Array[String] = Array("0 : ", 1 : (1,3), 2 : (1,2), 3 : (2,4), "4 : ", 5 : (2,3), 6 : (3,6), 7 : (3,8))  scala> val hashpar = nopar.partitionBy(new org.apache.spark.HashPartitioner(7))  hashpar: org.apache.spark.rdd.RDD[(Int, Int)] = ShuffledRDD[12] at partitionBy at <console>:26  scala> hashpar.count  res18: Long = 6  scala> hashpar.partitioner  res21: Option[org.apache.spark.Partitioner] = Some(org.apache.spark.HashPartitioner@7)  scala> hashpar.mapPartitions(iter => Iterator(iter.length)).collect()  res19: Array[Int] = Array(0, 3, 1, 2, 0, 0, 0) |





### Range分区方式

HashPartitioner分区弊端：可能导致每个分区中数据量的不均匀，极端情况下会导致某些分区拥有RDD的全部数据。

RangePartitioner分区优势：尽量保证每个分区中数据量的均匀，而且分区与分区之间是有序的，一个分区中的元素肯定都是比另一个分区内的元素小或者大；

但是分区内的元素是不能保证顺序的。简单的说就是将一定范围内的数映射到某一个分区内。

RangePartitioner作用：将一定范围内的数映射到某一个分区内，在实现中，分界的[算法](http://lib.csdn.net/base/datastructure" \t "_blank" \o "算法与数据结构知识库)尤为重要。用到了水塘抽样算法。

### 自定义分区方式

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

numPartitions: Int:返回创建出来的分区数。

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

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

假设我们需要将相同后缀的数据写入相同的文件，我们通过将相同后缀的数据分区到相同的分区并保存输出来实现。

|  |
| --- |
| **package** com.atguigu.spark  **import** org.apache.spark.{Partitioner, SparkConf, SparkContext}  */\*\*  \* Created by wuyufei on 04/09/2017.  \*/* **class** CustomerPartitioner(numParts:Int) **extends** Partitioner {   *//覆盖分区数* **override def** numPartitions: Int = numParts   *//覆盖分区号获取函数* **override def** getPartition(key: Any): Int = {  **val** ckey: String = key.toString  ckey.substring(ckey.length-1).toInt%numParts  } }  **object** CustomerPartitioner {  **def** main(args: Array[String]) {  **val** conf=**new** SparkConf().setAppName(**"partitioner"**)  **val** sc=**new** SparkContext(conf)   **val** data=sc.parallelize(*List*(**"aa.2"**,**"bb.2"**,**"cc.3"**,**"dd.3"**,**"ee.5"**))    data.map((\_,1)).partitionBy(**new** CustomerPartitioner(5)).keys.saveAsTextFile(**"hdfs://master01:9000/partitioner"**)  } } |

|  |
| --- |
| scala> val data=sc.parallelize(List("aa.2","bb.2","cc.3","dd.3","ee.5").zipWithIndex,2)  data: org.apache.spark.rdd.RDD[(String, Int)] = ParallelCollectionRDD[7] at parallelize at <console>:27  scala> data.collect  res4: Array[(String, Int)] = Array((aa.2,0), (bb.2,1), (cc.3,2), (dd.3,3), (ee.5,4))  scala> data.mapPartitionsWithIndex((index,iter)=>Iterator(index.toString +" : "+ iter.mkString("|"))).collect  res5: Array[String] = Array(0 : (aa.2,0)|(bb.2,1), 1 : (cc.3,2)|(dd.3,3)|(ee.5,4))  scala> :paste  // Entering paste mode (ctrl-D to finish)  class CustomerPartitioner(numParts:Int) extends org.apache.spark.Partitioner{  //覆盖分区数  override def numPartitions: Int = numParts  //覆盖分区号获取函数  override def getPartition(key: Any): Int = {  val ckey: String = key.toString  ckey.substring(ckey.length-1).toInt%numParts  }  }  // Exiting paste mode, now interpreting.  defined class CustomerPartitioner  scala> data.partitionBy(new CustomerPartitioner(4))  res7: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[9] at partitionBy at <console>:31  scala> res7.mapPartitionsWithIndex((index,iter)=>Iterator(index.toString +" : "+ iter.mkString("|"))).collect  res8: Array[String] = Array("0 : ", 1 : (ee.5,4), 2 : (aa.2,0)|(bb.2,1), 3 : (cc.3,2)|(dd.3,3)) |

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

### 分区Shuffle优化

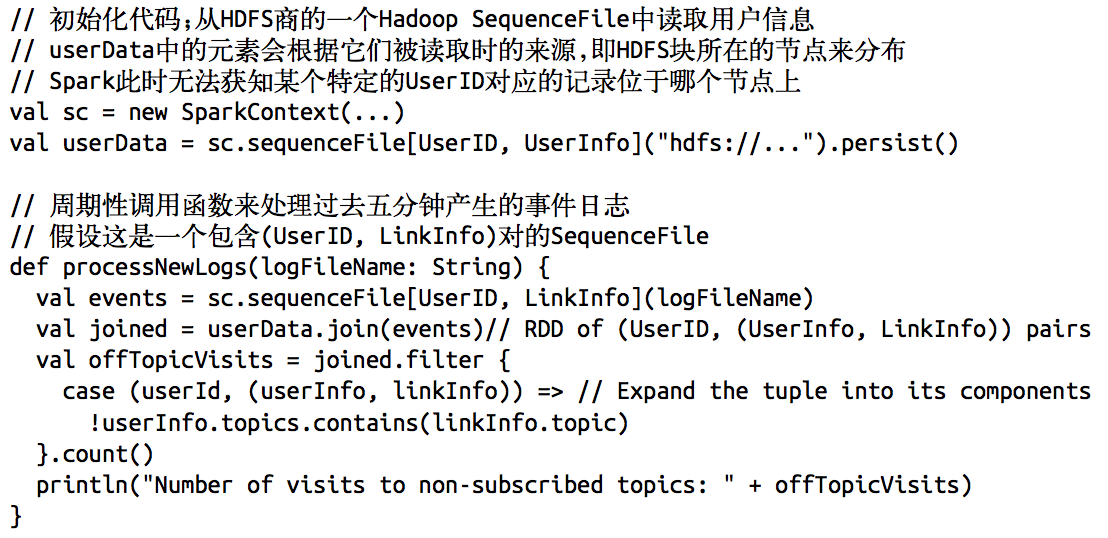
在分布式程序中， 通信的代价是很大的，因此控制数据分布以获得最少的网络传输可以极大地提升整体性能。

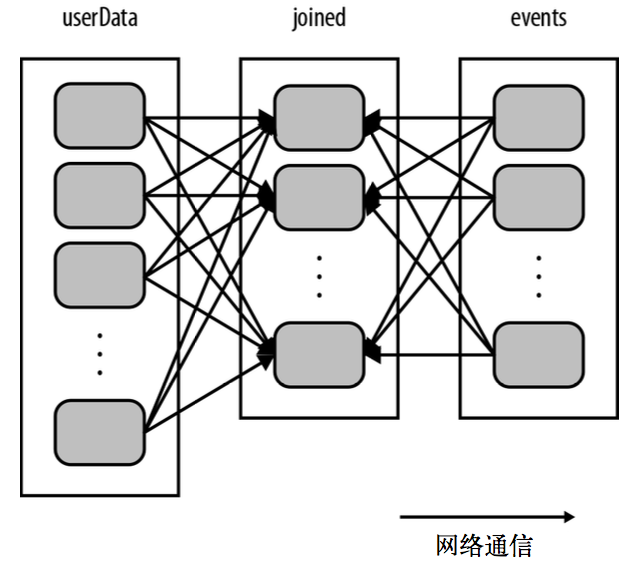
Spark 中所有的键值对 RDD 都可以进行分区。系统会根据一个针对键的函数对元素进行分 组。 主要有哈希分区和范围分区，当然用户也可以自定义分区函数。

通过分区可以有效提升程序性能。如下例子：

分析这样一个应用，它在内存中保存着一张很大的用户信息表—— 也就是一个由 (UserID, UserInfo) 对组成的 RDD，其中 UserInfo 包含一个该用户所订阅 的主题的列表。该应用会周期性地将这张表与一个小文件进行组合，这个小文件中存着过 去五分钟内发生的事件——其实就是一个由 (UserID, LinkInfo) 对组成的表，存放着过去 五分钟内某网站各用户的访问情况。例如，我们可能需要对用户访问其未订阅主题的页面 的情况进行统计。

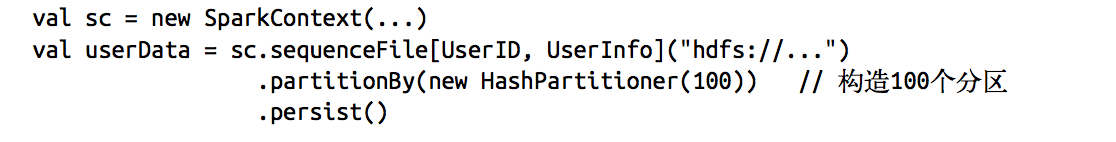
解决方案一：



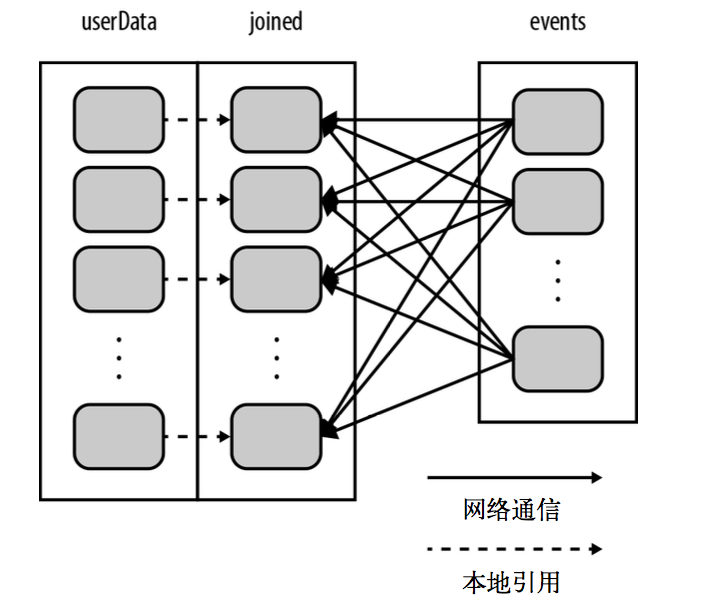


这段代码可以正确运行，但是不够高效。这是因为在每次调用 processNewLogs() 时都会用 到 join() 操作，而我们对数据集是如何分区的却一无所知。默认情况下，连接操作会将两 个数据集中的所有键的哈希值都求出来，将该哈希值相同的记录通过网络传到同一台机器 上，然后在那台机器上对所有键相同的记录进行连接操作。因为 userData 表比 每五分钟出现的访问日志表 events 要大得多，所以要浪费时间做很多额外工作:在每次调 用时都对 userData 表进行哈希值计算和跨节点数据混洗，降低了程序的执行效率。

优化方法：



我们在构 建 userData 时调用了 partitionBy()，Spark 就知道了该 RDD 是根据键的哈希值来分 区的，这样在调用 join() 时，Spark 就会利用到这一点。具体来说，当调用 userData. join(events) 时，Spark 只会对 events 进行数据混洗操作，将 events 中特定 UserID 的记 录发送到 userData 的对应分区所在的那台机器上。这样，需要通过网络传输的 数据就大大减少了，程序运行速度也可以显著提升了。



### 基于分区进行操作

基于分区对数据进行操作可以让我们避免为每个数据元素进行重复的配置工作。诸如打开 数据库连接或创建随机数生成器等操作，都是我们应当尽量避免为每个元素都配置一次的 工作。Spark 提供基于分区的 mapPartition 和 foreachPartition，让你的部分代码只对 RDD 的每个分区运行 一次，这样可以帮助降低这些操作的代价。

### 从分区中获益的操作

能够从数据分区中获得性能提升的操作有cogroup()、 groupWith()、join()、leftOuterJoin()、rightOuterJoin()、groupByKey()、reduceByKey()、 combineByKey() 以及 lookup()等。

# 数据读取与保存主要方式

## 文本文件输入输出

当我们将一个文本文件读取为 RDD 时，输入的每一行 都会成为RDD的一个元素。也可以将多个完整的文本文件一次性读取为一个pair RDD， 其中键是文件名，值是文件内容。

val input = sc.textFile("./README.md")

如果传递目录，则将目录下的所有文件读取作为RDD。

文件路径支持通配符。

通过wholeTextFiles()对于大量的小文件读取效率比较高，大文件效果没有那么高。

Spark通过saveAsTextFile() 进行文本文件的输出，该方法接收一个路径，并将 RDD 中的内容都输入到路径对应的文件中。Spark 将传入的路径作为目录对待，会在那个 目录下输出多个文件。这样，Spark 就可以从多个节点上并行输出了。

result.saveAsTextFile(outputFile)

|  |
| --- |
| scala> sc.textFile("./README.md")  res6: org.apache.spark.rdd.RDD[String] = ./README.md MapPartitionsRDD[7] at textFile at <console>:25  scala> val readme = sc.textFile("./README.md")  readme: org.apache.spark.rdd.RDD[String] = ./README.md MapPartitionsRDD[9] at textFile at <console>:24  scala> readme.collect()  res7: Array[String] = Array(# Apache Spark, "", Spark is a fast and general cluster...  scala> readme.saveAsTextFile("hdfs://master01:9000/test") |

## JSON文件输入输出

如果JSON文件中每一行就是一个JSON记录，那么可以通过将JSON文件当做文本文件来读取，然后利用相关的JSON库对每一条数据进行JSON解析。

|  |
| --- |
| scala> import org.json4s.\_  import org.json4s.\_  scala> import org.json4s.jackson.JsonMethods.\_  import org.json4s.jackson.JsonMethods.\_  scala> import org.json4s.jackson.Serialization  import org.json4s.jackson.Serialization  scala> var result = sc.textFile("examples/src/main/resources/people.json")  result: org.apache.spark.rdd.RDD[String] = examples/src/main/resources/people.json MapPartitionsRDD[7] at textFile at <console>:47  scala> implicit val formats = Serialization.formats(ShortTypeHints(List()))  formats: org.json4s.Formats{val dateFormat: org.json4s.DateFormat; val typeHints: org.json4s.TypeHints} = org.json4s.Serialization$$anon$1@61f2c1da  scala> result.collect().foreach(x => {var c = parse(x).extract[Person];println(c.name + "," + c.age)})  Michael,30  Andy,30  Justin,19 |

如果JSON数据是跨行的，那么只能读入整个文件，然后对每个文件进行解析。

JSON数据的输出主要是通过在输出之前将由结构 化数据组成的 RDD 转为字符串 RDD，然后使用 Spark 的文本文件 API 写出去。

说白了还是以文本文件的形式存，只是文本的格式已经在程序中转换为JSON。

## CSV文件输入输出

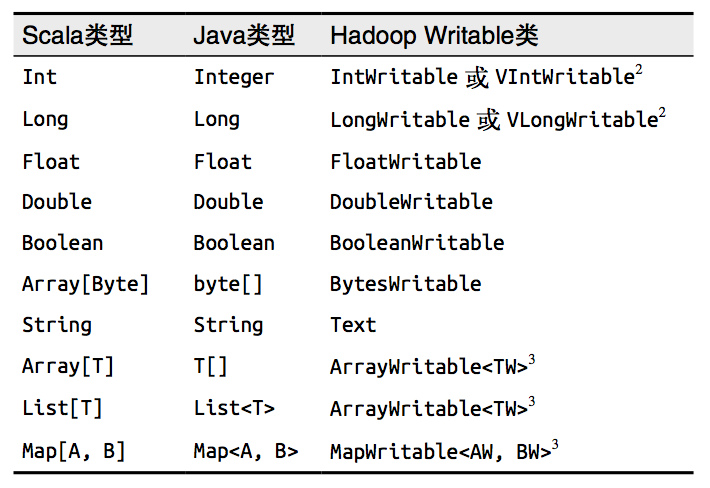
读取 CSV/TSV 数据和读取 JSON 数据相似，都需要先把文件当作普通文本文件来读取数据，然后通过将每一行进行解析实现对CSV的读取。

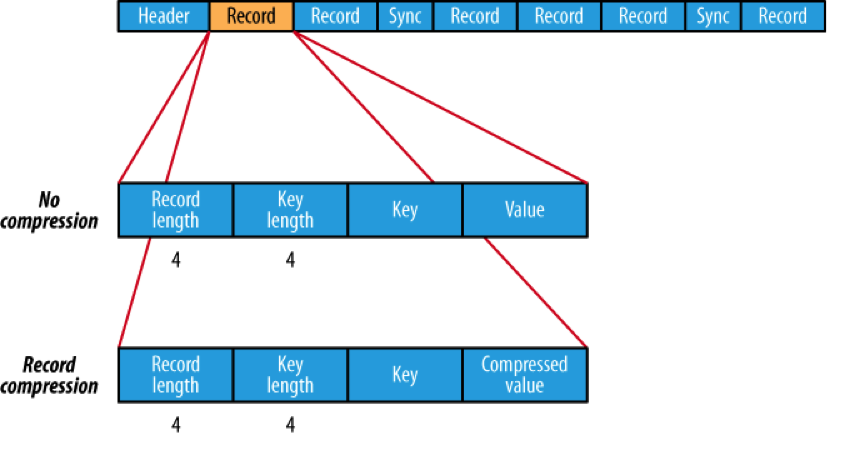
CSV/TSV数据的输出也是需要将结构化RDD通过相关的库转换成字符串RDD，然后使用 Spark 的文本文件 API 写出去。

## SequenceFile文件输入输出

 SequenceFile文件是[Hadoop](http://lib.csdn.net/base/hadoop" \t "_blank" \o "Hadoop知识库)用来存储二进制形式的key-value对而设计的一种平面文件(Flat File)。

Spark 有专门用来读取 SequenceFile 的接口。在 SparkContext 中，可以调用 sequenceFile[ keyClass, valueClass](path)。





|  |
| --- |
| scala> val data=sc.parallelize(List((2,"aa"),(3,"bb"),(4,"cc"),(5,"dd"),(6,"ee")))  data: org.apache.spark.rdd.RDD[(Int, String)] = ParallelCollectionRDD[16] at parallelize at <console>:24  scala> data.saveAsSequenceFile("hdfs://master01:9000/sequdata")  scala> val sdata = sc.sequenceFile[Int,String]("hdfs://master01:9000/sdata/p\*")  sdata: org.apache.spark.rdd.RDD[(Int, String)] = MapPartitionsRDD[19] at sequenceFile at <console>:24  scala> sdata.collect()  res14: Array[(Int, String)] = Array((2,aa), (3,bb), (4,cc), (5,dd), (6,ee)) |

可以直接调用 saveAsSequenceFile(path) 保存你的PairRDD，它会帮你写出数据。需要键和值能够自动转为Writable类型。

## 对象文件输入输出

对象文件是将对象序列化后保存的文件，采用Java的序列化机制。

可以通过objectFile[k,v](path) 函数接收一个路径，读取对象文件，返回对应的 RDD，也可以通过调用saveAsObjectFile() 实现对对象文件的输出。因为是序列化所以要指定类型。

|  |
| --- |
| scala> val data=sc.parallelize(List((2,"aa"),(3,"bb"),(4,"cc"),(5,"dd"),(6,"ee")))  data: org.apache.spark.rdd.RDD[(Int, String)] = ParallelCollectionRDD[20] at parallelize at <console>:24  scala> data.saveAsObjectFile("hdfs://master01:9000/objfile")  scala> import org.apache.spark.rdd.RDD  import org.apache.spark.rdd.RDD  scala> val objrdd:RDD[(Int,String)] = sc.objectFile[(Int,String)]("hdfs://master01:9000/objfile/p\*")  objrdd: org.apache.spark.rdd.RDD[(Int, String)] = MapPartitionsRDD[28] at objectFile at <console>:25  scala> objrdd.collect()  res20: Array[(Int, String)] = Array((2,aa), (3,bb), (4,cc), (5,dd), (6,ee)) |

## Hadoop输入输出格式

Spark的整个生态系统与Hadoop是完全兼容的,所以对于Hadoop所支持的文件类型或者数据库类型,Spark也同样支持.另外,由于Hadoop的API有新旧两个版本,所以Spark为了能够兼容Hadoop所有的版本,也提供了两套创建操作接口.对于外部存储创建操作而言,hadoopRDD和newHadoopRDD是最为抽象的两个函数接口,主要包含以下四个参数.

1. 输入格式(InputFormat): 制定数据输入的类型,如TextInputFormat等,新旧两个版本所引用的版本分别是org.apache.hadoop.mapred.InputFormat和org.apache.hadoop.mapreduce.InputFormat(NewInputFormat)
2. 键类型: 指定[K,V]键值对中K的类型
3. 值类型: 指定[K,V]键值对中V的类型
4. 分区值: 指定由外部存储生成的RDD的partition数量的最小值,如果没有指定,系统会使用默认值defaultMinSplits

其他创建操作的API接口都是为了方便最终的Spark程序开发者而设置的,是这两个接口的高效实现版本.例如,对于textFile而言,只有path这个指定文件路径的参数,其他参数在系统内部指定了默认值

**兼容旧版本HadoopAPI的创建操作**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **文件路径** | **输入格式** | **键类型** | **值类型** | **分区值** |
| textFile(path: String, minPartitions: Int = defaultMinPartitions) | path | TextInputFormat | LongWritable | Text | minSplits |
| hadoopFile[K, V, F <: InputFormat[K, V]](path: String, minPartitions: Int) (implicit km: ClassTag[K], vm: ClassTag[V], fm: ClassTag[F]): RDD[(K, V)] | path | F | K | V | minSplits |
| hadoopFile[K, V, F <: [K, V]](path: String) (implicit km: ClassTag[K], vm: ClassTag[V], fm: ClassTag[F]): RDD[(K, V)] | path | F | K | V | DefaultMinSplits |
| hadoopFile[K, V](path: String, inputFormatClass: Class[\_ <: InputFormat[K, V]], keyClass: Class[K], valueClass: Class[V], minPartitions: Int = defaultMinPartitions): RDD[(K, V)] | path | inputFormatClass | keyClass | valueClass | defaultMinPartitions |
| hadoopRDD[K, V](conf: JobConf, inputFormatClass: Class[\_ <: InputFormat[K, V]], keyClass: Class[K], valueClass: Class[V], minPartitions: Int = defaultMinPartitions): RDD[(K, V)] | n/a | inputFormatClass | keyClass | valueClass | defaultMinPartitions |
| sequenceFile[K, V](path: String, minPartitions: Int = defaultMinPartitions) (implicit km: ClassTag[K], vm: ClassTag[V], kcf: () ⇒ WritableConverter[K], vcf: () ⇒ WritableConverter[V]): RDD[(K, V)] | path | SequenceFileInputFormat[K,V] | K | V | defaultMinPartitions |
| objectFile[T](path: String, minPartitions: Int = defaultMinPartitions)(implicit arg0: ClassTag[T]): RDD[T] | path | SequenceFileInputFormat[NullWritable,BytesWritable] | NullWritable | BytesWritable | minSplits |

兼容新版本HadoopAPI的创建操作

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 文件路径 | 输入格式 | 键类型 | 值类型 | 分区值 |
| newAPIHadoopFile[K, V, F <: InputFormat[K, V]](path: String, fClass: Class[F], kClass: Class[K], vClass: Class[V], conf: Configuration = hadoopConfiguration): RDD[(K, V)] | path | F | K | V | n/a |
| newAPIHadoopFile[K, V, F <: InputFormat[K, V]](path: String)(implicit km: ClassTag[K], vm: ClassTag[V], fm: ClassTag[F]): RDD[(K, V)] | path | F | K | V | n/a |
| newAPIHadoopRDD[K, V, F <: InputFormat[K, V]](conf: Configuration = hadoopConfiguration, fClass: Class[F], kClass: Class[K], vClass: Class[V]): RDD[(K, V)] | n/a | F | K | V | n/a |

***注意:***

1.在Hadoop中以压缩形式存储的数据,不需要指定解压方式就能够进行读取,因为Hadoop本身有一个解压器会根据压缩文件的后缀推断解压算法进行解压.

2.如果用Spark从Hadoop中读取某种类型的数据不知道怎么读取的时候,上网查找一个使用map-reduce的时候是怎么读取这种这种数据的,然后再将对应的读取方式改写成上面的hadoopRDD和newAPIHadoopRDD两个类就行了.

***读取示例：***

|  |
| --- |
|  |

|  |
| --- |
| scala> val data = sc.parallelize(Array((30,"hadoop"), (71,"hive"), (11,"cat")))  data: org.apache.spark.rdd.RDD[(Int, String)] = ParallelCollectionRDD[47] at parallelize at <console>:35  scala> data.saveAsNewAPIHadoopFile("hdfs://master01:9000/output4/",classOf[LongWritable] ,classOf[Text] ,classOf[org.apache.hadoop.mapreduce.lib.output.TextOutputFormat[LongWritable, Text]]) |

对于RDD最后的归宿除了返回为集合和标量,也可以将RDD存储到外部文件系统或者数据库中,Spark系统与Hadoop是完全兼容的,所以MapReduce所支持的读写文件或者数据库类型,Spark也同样支持.另外,由于Hadoop的API有新旧两个版本,所以Spark为了能够兼容Hadoop所有的版本,也提供了两套API.

将RDD保存到HDFS中在通常情况下需要关注或者设置五个参数,即文件保存的路径,key值的class类型,Value值的class类型,RDD的输出格式(OutputFormat,如TextOutputFormat/SequenceFileOutputFormat),以及最后一个相关的参数codec(这个参数表示压缩存储的压缩形式,如DefaultCodec,Gzip,Codec等等)

***兼容旧版API***

|  |
| --- |
| saveAsObjectFile(path: String): Unit |
| saveAsTextFile(path: String, codec: Class[\_ <: CompressionCodec]): Unit |
| saveAsTextFile(path: String): Unit |
| saveAsHadoopFile[F <: OutputFormat[K, V]](path: String)(implicit fm: ClassTag[F]): Unit |
| saveAsHadoopFile[F <: OutputFormat[K, V]](path: String, codec: Class[\_ <: CompressionCodec])(implicit fm: ClassTag[F]): Unit |
| saveAsHadoopFile(path: String, keyClass: Class[\_], valueClass: Class[\_], outputFormatClass: Class[\_ <: OutputFormat[\_, \_]], codec: Class[\_ <: CompressionCodec]): Unit |
| saveAsHadoopDataset(conf: JobConf): Unit |

这里列出的API,前面6个都是saveAsHadoopDataset的简易实现版本,仅仅支持将RDD存储到HDFS中,而saveAsHadoopDataset的参数类型是JobConf,所以其不仅能够将RDD存储到HDFS中,也可以将RDD存储到其他数据库中,如Hbase,MangoDB,Cassandra等.

***兼容新版API***

|  |
| --- |
| saveAsNewAPIHadoopFile(path: String, keyClass: Class[\_], valueClass: Class[\_], outputFormatClass: Class[\_ <: OutputFormat[\_, \_]], conf: Configuration = self.context.hadoopConfiguration): Unit |
| saveAsNewAPIHadoopFile[F <: OutputFormat[K, V]](path: String)(implicit fm: ClassTag[F]): Unit |
| saveAsNewAPIHadoopDataset(conf: Configuration): Unit |

同样的,前2个API是saveAsNewAPIHadoopDataset的简易实现,只能将RDD存到HDFS中,而saveAsNewAPIHadoopDataset比较灵活.新版的API没有codec的参数,所以要压缩存储文件到HDFS中每需要使用hadoopConfiguration参数,设置对应mapreduce.map.output.compress.codec参数和mapreduce.map.output.compress参数.

注意:

1.如果不知道怎么将RDD存储到Hadoop生态的系统中,主要上网搜索一下对应的map-reduce是怎么将数据存储进去的,然后改写成对应的saveAsHadoopDataset或saveAsNewAPIHadoopDataset就可以了.

***写入示例：***

|  |
| --- |
|  |

|  |
| --- |
| scala> val read = sc.newAPIHadoopFile[LongWritable, Text, org.apache.hadoop.mapreduce.lib.input.TextInputFormat]("hdfs://master01:9000/output3/part\*", classOf[org.apache.hadoop.mapreduce.lib.input.TextInputFormat], classOf[LongWritable], classOf[Text])  read: org.apache.spark.rdd.RDD[(org.apache.hadoop.io.LongWritable, org.apache.hadoop.io.Text)] = hdfs://master01:9000/output3/part\* NewHadoopRDD[48] at newAPIHadoopFile at <console>:35  scala> read.map{case (k, v) => v.toString}.collect  res44: Array[String] = Array(30 hadoop, 71 hive, 11 cat) |

## 文件系统的输入输出

Spark 支持读写很多种文件系统， 像本地文件系统、Amazon S3、HDFS等。

## 数据库的输入输出

关系型数据库连接

支持通过Java JDBC访问关系型数据库。需要通过JdbcRDD进行，示例如下:

Mysql读取：

|  |
| --- |
| **def** main (args: Array[String] ) {  **val** sparkConf = **new** SparkConf ().setMaster (**"local[2]"**).setAppName (**"JdbcApp"**)  **val** sc = **new** SparkContext (sparkConf)   **val** rdd = **new** org.apache.spark.rdd.JdbcRDD (  sc,  () => {  Class.*forName* (**"com.mysql.jdbc.Driver"**).newInstance()  java.sql.DriverManager.getConnection (**"jdbc:mysql://master01:3306/rdd"**, **"root"**, **"hive"**)  },  **"select \* from rddtable where id >= ? and id <= ?;"**,  1,  10,  1,  r => (r.getInt(1), r.getString(2)))   println (rdd.count () )  rdd.foreach (println (\_) )  sc.stop () } |

Mysql写入：

|  |
| --- |
| **def** main(args: Array[String]) {  **val** sparkConf = **new** SparkConf().setMaster(**"local[2]"**).setAppName(**"HBaseApp"**)  **val** sc = **new** SparkContext(sparkConf)  **val** data = sc.parallelize(*List*(**"Female"**, **"Male"**,**"Female"**))   data.foreachPartition(insertData) }  **def** insertData(iterator: Iterator[String]): Unit = {  Class.forName ("com.mysql.jdbc.Driver").newInstance()  **val** conn = java.sql.DriverManager.getConnection(**"jdbc:mysql://master01:3306/rdd"**, **"root"**, **"hive"**)  iterator.foreach(data => {  **val** ps = conn.prepareStatement(**"insert into rddtable(name) values (?)"**)  ps.setString(1, data)   ps.executeUpdate()  }) } |

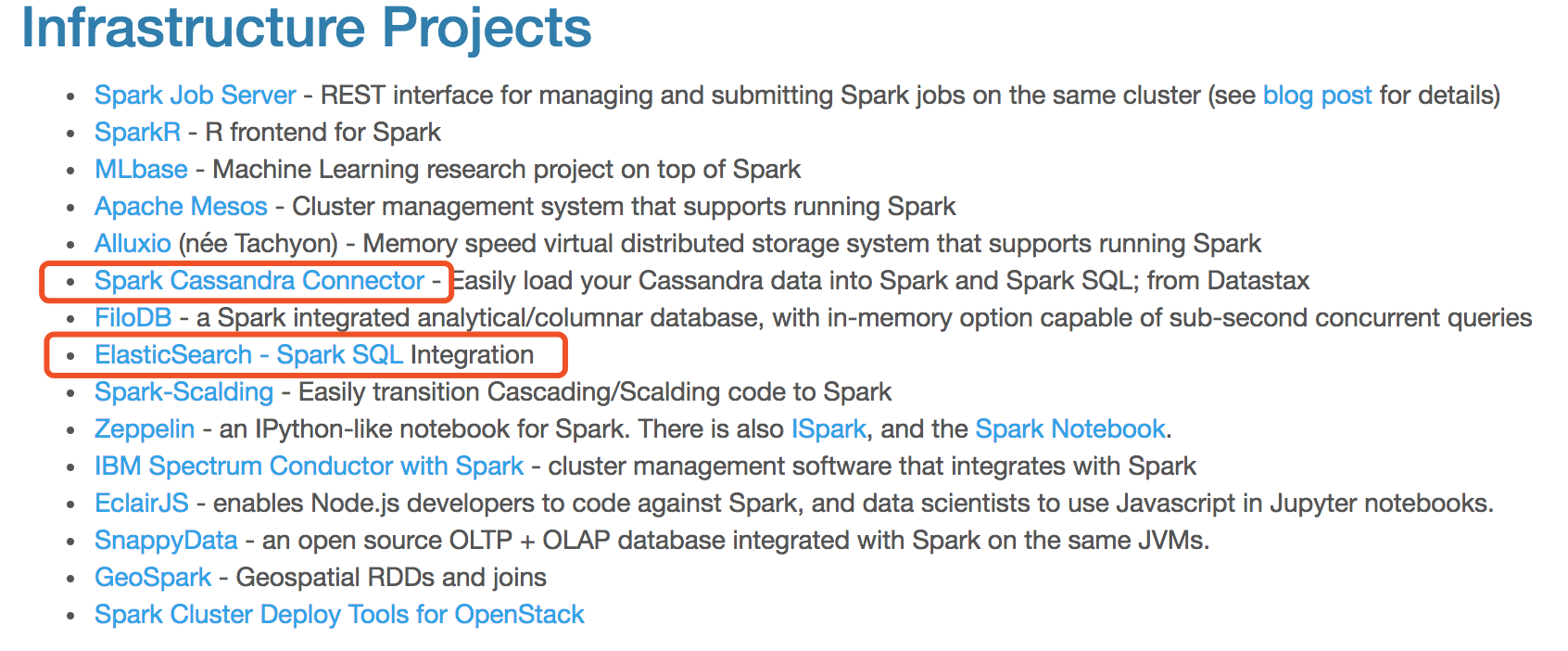
JdbcRDD 接收这样几个参数。

• 首先，要提供一个用于对数据库创建连接的函数。这个函数让每个节点在连接必要的配 置后创建自己读取数据的连接。

• 接下来，要提供一个可以读取一定范围内数据的查询，以及查询参数中lowerBound和 upperBound 的值。这些参数可以让 Spark 在不同机器上查询不同范围的数据，这样就不 会因尝试在一个节点上读取所有数据而遭遇性能瓶颈。

• 这个函数的最后一个参数是一个可以将输出结果从转为对操作数据有用的格式的函数。如果这个参数空缺，Spark会自动将每行结果转为一个对象数组。

***Cassandra数据库和ElasticSearch集成：***



***HBase数据库***

由于 org.apache.hadoop.hbase.mapreduce.TableInputFormat 类的实现，Spark 可以通过Hadoop输入格式访问HBase。这个输入格式会返回键值对数据，其中键的类型为org. apache.hadoop.hbase.io.ImmutableBytesWritable，而值的类型为org.apache.hadoop.hbase.client.Result。

HBase读取：

|  |
| --- |
| **def** main(args: Array[String]) {  **val** sparkConf = **new** SparkConf().setMaster(**"local[2]"**).setAppName(**"HBaseApp"**)  **val** sc = **new** SparkContext(sparkConf)   **val** conf = HBaseConfiguration.create()  *//HBase中的表名* conf.set(TableInputFormat.INPUT\_TABLE, **"fruit"**)   **val** hBaseRDD = sc.newAPIHadoopRDD(conf, classOf[TableInputFormat],  classOf[org.apache.hadoop.hbase.io.ImmutableBytesWritable],  classOf[org.apache.hadoop.hbase.client.Result])   **val** count = hBaseRDD.count()  println(**"hBaseRDD RDD Count:"**+ count)  hBaseRDD.cache()  hBaseRDD.foreach {  **case** (\_, result) =>  **val** key = Bytes.toString(result.getRow)  **val** name = Bytes.toString(result.getValue(**"info"**.getBytes, **"name"**.getBytes))  **val** color = Bytes.toString(result.getValue(**"info"**.getBytes, **"color"**.getBytes))  println(**"Row key:"** + key + **" Name:"** + name + **" Color:"** + color)  }  sc.stop() } |

HBase写入：

|  |
| --- |
| **def** main(args: Array[String]) {  **val** sparkConf = **new** SparkConf().setMaster(**"local[2]"**).setAppName(**"HBaseApp"**)  **val** sc = **new** SparkContext(sparkConf)   **val** conf = HBaseConfiguration.create()  **val** jobConf = **new** JobConf(conf)  jobConf.setOutputFormat(*classOf*[TableOutputFormat])  jobConf.set(TableOutputFormat.OUTPUT\_TABLE, **"fruit\_spark"**)   **val** fruitTable = TableName.valueOf(**"fruit\_spark"**)  **val** tableDescr = **new** HTableDescriptor(fruitTable)  tableDescr.addFamily(**new** HColumnDescriptor(**"info"**.getBytes))   **val** admin = **new** HBaseAdmin(conf)  **if** (admin.tableExists(fruitTable)) {  admin.disableTable(fruitTable)  admin.deleteTable(fruitTable)  }  admin.createTable(tableDescr)   **def** convert(triple: (Int, String, Int)) = {  **val** put = **new** Put(Bytes.toBytes(triple.\_1))  put.addImmutable(Bytes.toBytes(**"info"**), Bytes.toBytes(**"name"**), Bytes.toBytes(triple.\_2))  put.addImmutable(Bytes.toBytes(**"info"**), Bytes.toBytes(**"price"**), Bytes.toBytes(triple.\_3))  (**new** ImmutableBytesWritable, put)  }  **val** initialRDD = sc.parallelize(List((1,**"apple"**,11), (2,**"banana"**,12), (3,**"pear"**,13)))  **val** localData = initialRDD.map(convert)   localData.saveAsHadoopDataset(jobConf) } |

# RDD编程进阶

## 累加器

累加器用来对信息进行聚合，通常在向 Spark 传递函数时，比如使用 map() 函数或者用 filter() 传条件时，可以使用驱 动器程序中定义的变量，但是集群中运行的每个任务都会得到这些变量的一份新的副本， 更新这些副本的值也不会影响驱动器中的对应变量。 如果我们想实现所有分片处理时更新共享变量的功能，那么累加器可以实现我们想要的效果。

针对一个输入的日志文件，如果我们想计算文件中所有空行的数量，我们可以编写以下程序：

|  |
| --- |
| scala> val notice = sc.textFile("./NOTICE")  notice: org.apache.spark.rdd.RDD[String] = ./NOTICE MapPartitionsRDD[40] at textFile at <console>:32  scala> val blanklines = sc.accumulator(0)  warning: there were two deprecation warnings; re-run with -deprecation for details  blanklines: org.apache.spark.Accumulator[Int] = 0  scala> val tmp = notice.flatMap(line => {  | if (line == "") {  | blanklines += 1  | }  | line.split(" ")  | })  tmp: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[41] at flatMap at <console>:36  scala> tmp.count()  res31: Long = 3213  scala> blanklines.value  res32: Int = 171 |

累加器的用法如下所示。

通过在驱动器中调用SparkContext.accumulator(initialValue)方法，创建出存有初始值的累加器。返回值为 org.apache.spark.Accumulator[T] 对象，其中 T 是初始值 initialValue 的类型。

Spark闭包里的执行器代码可以使用累加器的 += 方法(在Java中是 add)增加累加器的值。

驱动器程序可以调用累加器的value属性(在Java中使用value()或setValue())来访问累加器的值。

注意：工作节点上的任务不能访问累加器的值。从这些任务的角度来看，累加器是一个只写变量。

对于要在行动操作中使用的累加器，Spark只会把每个任务对各累加器的修改应用一次。因此，如果想要一个无论在失败还是重复计 算时都绝对可靠的累加器，我们必须把它放在 foreach() 这样的行动操作中。转化操作中累加器可 能会发生不止一次更新。

## 自定义累加器

自定义累加器类型的功能在1.X版本中就已经提供了，但是使用起来比较麻烦，在2.0版本后，累加器的易用性有了较大的改进，而且官方还提供了一个新的抽象类：AccumulatorV2来提供更加友好的自定义类型累加器的实现方式。实现自定义类型累加器需要继承AccumulatorV2并至少覆写下例中出现的方法，下面这个累加器可以用于在程序运行过程中收集一些文本类信息，最终以Set[String]的形式返回。

|  |
| --- |
| **package** com.atguigu.spark  **import** org.apache.spark.util.AccumulatorV2 **import** org.apache.spark.{SparkConf, SparkContext} **import** scala.collection.JavaConversions.\_   */\*\*  \* Created by wuyufei on 05/09/2017.  \*/* **class** LogAccumulator **extends** org.apache.spark.util.AccumulatorV2[String, java.util.Set[String]] {  **private val** *\_logArray*: java.util.Set[String] = **new** java.util.HashSet[String]()   **override def** isZero: Boolean = {  *\_logArray*.isEmpty  }   **override def** reset(): Unit = {  *\_logArray*.clear()  }   **override def** add(v: String): Unit = {  *\_logArray*.add(v)  }   **override def** merge(other: org.apache.spark.util.AccumulatorV2[String, java.util.Set[String]]): Unit = {  other **match** {  **case** o: LogAccumulator => *\_logArray*.addAll(o.value)  }   }   **override def** value: java.util.Set[String] = {  java.util.Collections.*unmodifiableSet*(*\_logArray*)  }   **override def** copy():org.apache.spark.util.AccumulatorV2[String, java.util.Set[String]] = {  **val** newAcc = **new** LogAccumulator()  *\_logArray*.synchronized{  newAcc.*\_logArray*.addAll(*\_logArray*)  }  newAcc  } }  *// 过滤掉带字母的* **object** LogAccumulator {  **def** main(args: Array[String]) {  **val** conf=**new** SparkConf().setAppName(**"LogAccumulator"**)  **val** sc=**new** SparkContext(conf)   **val** accum = **new** LogAccumulator  sc.register(accum, **"logAccum"**)  **val** sum = sc.parallelize(*Array*(**"1"**, **"2a"**, **"3"**, **"4b"**, **"5"**, **"6"**, **"7cd"**, **"8"**, **"9"**), 2).filter(line => {  **val** pattern = **"""^-?(\d+)"""  val** flag = line.matches(pattern)  **if** (!flag) {  accum.add(line)  }  flag  }).map(\_.toInt).reduce(\_ + \_)   *println*(**"sum: "** + sum)  **for** (v <- accum.value) *print*(v + **""**)  *println*()  sc.stop()  } } |

## 广播变量

广播变量用来**高效分发较大的对象**。向所有工作节点发送一个 较大的只读值，以供一个或多个 Spark 操作使用。比如，如果你的应用需要向所有节点发 送一个较大的只读查询表，甚至是机器学习算法中的一个很大的特征向量，广播变量用起 来都很顺手。

传统方式下，Spark 会自动把闭包中所有引用到的变量发送到工作节点上。虽然这很方便，但也很低效。原因有二:首先，默认的任务发射机制是专门为小任务进行优化的；其次，事实上你可能会在多个并行操作中使用同一个变量，但是 Spark会为每个任务分别发送。

|  |
| --- |
| scala> val broadcastVar = sc.broadcast(Array(1, 2, 3))  broadcastVar: org.apache.spark.broadcast.Broadcast[Array[Int]] = Broadcast(35)  scala> broadcastVar.value  res33: Array[Int] = Array(1, 2, 3) |

使用广播变量的过程如下：

(1) 通过对一个类型 T 的对象调用 SparkContext.broadcast 创建出一个 Broadcast[T] 对象。 任何可序列化的类型都可以这么实现。

(2) 通过 value 属性访问该对象的值(在 Java 中为 value() 方法)。

(3) 变量只会被发到各个节点一次，应作为只读值处理(修改这个值不会影响到别的节点)。