**1、Spark编程模型**

**1.1 术语定义**

**应用程序（Application）**： 基于Spark的用户程序，包含了一个Driver Program 和集群中多个的Executor；

**驱动程序（Driver Program）**：运行Application的main()函数并且创建SparkContext，通常用SparkContext代表Driver Program；

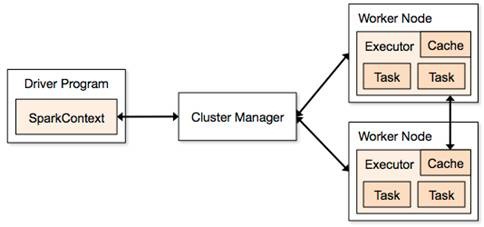
**执行单元（Executor）**： 是为某Application运行在Worker Node上的一个进程，该进程负责运行Task，并且负责将数据存在内存或者磁盘上，每个Application都有各自独立的Executors；

**集群管理程序（Cluster Manager）：** 在集群上获取资源的外部服务(例如：Standalone、Mesos或Yarn)；

**操作（Operation）**：作用于RDD的各种操作分为Transformation和Action；

**1.2 模型组成**

Spark应用程序可分两部分：Driver部分和Executor部分

[](http://images0.cnblogs.com/blog/107289/201508/111511034107511.jpg)

**1.2.1 Driver部分**

Driver部分主要是对SparkContext进行配置、初始化以及关闭。初始化SparkContext是为了构建Spark应用程序的运行环境，在初始化SparkContext，要先导入一些Spark的类和隐式转换；在Executor部分运行完毕后，需要将SparkContext关闭。

**1.2.2 Executor部分**

Spark应用程序的Executor部分是对数据的处理，数据分三种：

***1.2.2.1 原生数据***

包含原生的输入数据和输出数据

对于输入原生数据，Spark目前提供了两种：

  Scala集合数据集：如Array(1,2,3,4,5)，Spark使用parallelize方法转换成RDD

  Hadoop数据集：Spark支持存储在hadoop上的文件和hadoop支持的其他文件系统，如本地文件、HBase、SequenceFile和Hadoop的输入格式。例如Spark使用txtFile方法可以将本地文件或HDFS文件转换成RDD

对于输出数据，Spark除了支持以上两种数据，还支持scala标量

  生成Scala标量数据，如count（返回RDD中元素的个数）、reduce、fold/aggregate；返回几个标量，如take（返回前几个元素）。

  生成Scala集合数据集，如collect（把RDD中的所有元素倒入 Scala集合类型）、lookup（查找对应key的所有值）。

  生成hadoop数据集，如saveAsTextFile、saveAsSequenceFile

***1.2.2.2 RDD***

RDD具体在下一节中详细描述，RDD提供了四种算子：

输入算子：将原生数据转换成RDD，如parallelize、txtFile等

转换算子：最主要的算子，是Spark生成DAG图的对象，转换算子并不立即执行，在触发行动算子后再提交给driver处理，生成DAG图 -->  Stage --> Task  --> Worker执行。

缓存算子：对于要多次使用的RDD，可以缓冲加快运行速度，对重要数据可以采用多备份缓存。

行动算子：将运算结果RDD转换成原生数据，如count、reduce、collect、saveAsTextFile等。

***1.2.2.3 共享变量***

在Spark运行时，一个函数传递给RDD内的patition操作时，该函数所用到的变量在每个运算节点上都复制并维护了一份，并且各个节点之间不会相互影响。但是在Spark Application中，可能需要共享一些变量，提供Task或驱动程序使用。Spark提供了两种共享变量：

**广播变量（Broadcast Variables）**：可以缓存到各个节点的共享变量，通常为只读

– 广播变量缓存到各个节点的内存中，而不是每个 Task

– 广播变量被创建后，能在集群中运行的任何函数调用

– 广播变量是只读的，不能在被广播后修改

– 对于大数据集的广播， Spark 尝试使用高效的广播算法来降低通信成本

使用方法：

*val broadcastVar = sc.broadcast(Array(1, 2, 3))*

累计器：只支持加法操作的变量，可以实现计数器和变量求和。用户可以调用SparkContext.accumulator(v)创建一个初始值为v的累加器，而运行在集群上的Task可以使用“+=”操作，但这些任务却不能读取；只有驱动程序才能获取累加器的值。

使用方法：

*val accum = sc.accumulator(0)*

*sc.parallelize(Array(1, 2, 3, 4)).foreach(x => accum  + = x)*

*accum.value*

*val num=sc.parallelize(1 to 100)*

**2、RDD**

**2.1 术语定义**

弹性分布式数据集（RDD）： Resillient Distributed Dataset，Spark的基本计算单元，可以通过一系列算子进行操作（主要有Transformation和Action操作）；

有向无环图（DAG）：Directed Acycle graph，反应RDD之间的依赖关系；

有向无环图调度器（DAG Scheduler）：根据Job构建基于Stage的DAG，并提交Stage给TaskScheduler；

任务调度器（Task Scheduler）：将Taskset提交给worker（集群）运行并回报结果；

窄依赖（Narrow dependency）：子RDD依赖于父RDD中固定的data partition；

宽依赖（Wide Dependency）：子RDD对父RDD中的所有data partition都有依赖。

**2.2 RDD概念**

RDD是Spark的最基本抽象,是对分布式内存的抽象使用，实现了以操作本地集合的方式来操作分布式数据集的抽象实现。RDD是Spark最核心的东西，它表示已被分区，不可变的并能够被并行操作的数据集合，不同的数据集格式对应不同的RDD实现。RDD必须是可序列化的。RDD可以cache到内存中，每次对RDD数据集的操作之后的结果，都可以存放到内存中，下一个操作可以直接从内存中输入，省去了MapReduce大量的磁盘IO操作。这对于迭代运算比较常见的机器学习算法, 交互式数据挖掘来说，效率提升非常大。

RDD 最适合那种在数据集上的所有元素都执行相同操作的批处理式应用。在这种情况下， RDD 只需记录血统中每个转换就能还原丢失的数据分区，而无需记录大量的数据操作日志。所以 RDD 不适合那些需要异步、细粒度更新状态的应用 ，比如 Web 应用的存储系统，或增量式的 Web 爬虫等。对于这些应用，使用具有事务更新日志和数据检查点的数据库系统更为高效。

**2.2.1 RDD的特点**

1.来源：一种是从持久存储获取数据，另一种是从其他RDD生成

2.只读：状态不可变，不能修改

3.分区：支持元素根据 Key 来分区 ( Partitioning ) ，保存到多个结点上，还原时只会重新计算丢失分区的数据，而不会影响整个系统

4.路径：在 RDD 中叫世族或血统 ( lineage ) ，即 RDD 有充足的信息关于它是如何从其他 RDD 产生而来的

5.持久化：可以控制存储级别（内存、磁盘等）来进行持久化

6.操作：丰富的动作 ( Action ) ，如Count、Reduce、Collect和Save 等

**2.2.2 RDD基础数据类型**

目前有两种类型的基础RDD：并行集合（Parallelized Collections）：接收一个已经存在的Scala集合，然后进行各种并行计算。 Hadoop数据集（Hadoop Datasets）：在一个文件的每条记录上运行函数。只要文件系统是HDFS，或者hadoop支持的任意存储系统即可。这两种类型的RDD都可以通过相同的方式进行操作，从而获得子RDD等一系列拓展，形成lineage血统关系图。

**1.** **并行化集合**

并行化集合是通过调用SparkContext的parallelize方法，在一个已经存在的Scala集合上创建的（一个Seq对象）。集合的对象将会被拷贝，创建出一个可以被并行操作的分布式数据集。例如，下面的解释器输出，演示了如何从一个数组创建一个并行集合。

例如：val rdd = sc.parallelize(Array(1 to 10)) 根据能启动的executor的数量来进行切分多个slice，每一个slice启动一个Task来进行处理。

val rdd = sc.parallelize(Array(1 to 10), 5) 指定了partition的数量

**2.** **Hadoop数据集**

Spark可以将任何Hadoop所支持的存储资源转化成RDD,如本地文件（需要网络文件系统，所有的节点都必须能访问到）、HDFS、Cassandra、HBase、Amazon S3等，Spark支持文本文件、SequenceFiles和任何Hadoop InputFormat格式。

（1）使用textFile()方法可以将本地文件或HDFS文件转换成RDD

支持整个文件目录读取，文件可以是文本或者压缩文件(如gzip等，自动执行解压缩并加载数据)。如textFile（”file:///dfs/data”）

支持通配符读取,例如：

*val rdd1 = sc.textFile("file:///root/access\_log/access\_log\*.filter");*

*val rdd2=rdd1.map(\_.split("t")).filter(\_.length==6)*

*rdd2.count()*

*......*

*14/08/20 14:44:48 INFO HadoopRDD: Input split: file:/root/access\_log/access\_log.20080611.decode.filter:134217728+20705903*

*......*

textFile()可选第二个参数slice，默认情况下为每一个block分配一个slice。用户也可以通过slice指定更多的分片，但不能使用少于HDFS block的分片数。

（2）使用wholeTextFiles()读取目录里面的小文件，返回（用户名、内容）对

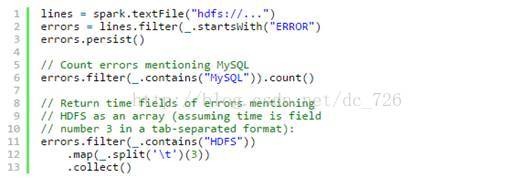
（3）使用sequenceFile[K,V]()方法可以将SequenceFile转换成RDD。SequenceFile文件是Hadoop用来存储二进制形式的key-value对而设计的一种平面文件(Flat File)。

（4）使用SparkContext.hadoopRDD方法可以将其他任何Hadoop输入类型转化成RDD使用方法。一般来说，HadoopRDD中每一个HDFS block都成为一个RDD分区。

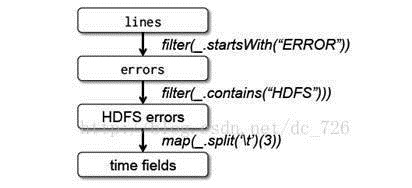
此外，通过Transformation可以将HadoopRDD等转换成FilterRDD(依赖一个父RDD产生）和JoinedRDD（依赖所有父RDD）等。

**2.2.3 例子：控制台日志挖掘**

假设网站中的一个 WebService 出现错误，我们想要从数以 TB 的 HDFS 日志文件中找到问题的原因，此时我们就可以用 Spark 加载日志文件到一组结点组成集群的RAM 中，并交互式地进行查询。以下是代码示例：

[](http://images0.cnblogs.com/blog/107289/201508/111511069577894.jpg)

首先行 1 从 HDFS 文件中创建出一个 RDD ，而行 2 则衍生出一个经过某些条件过滤后的 RDD 。行 3 将这个 RDD errors 缓存到内存中，然而第一个 RDD lines 不会驻留在内存中。这样做很有必要，因为 errors 可能非常小，足以全部装进内存，而原始数据则会非常庞大。经过缓存后，现在就可以反复重用 errors 数据了。我们这里做了两个操作，第一个是统计 errors 中包含 MySQL 字样的总行数，第二个则是取出包含 HDFS 字样的行的第三列时间，并保存成一个集合。

[](http://images0.cnblogs.com/blog/107289/201508/111511109267747.jpg)

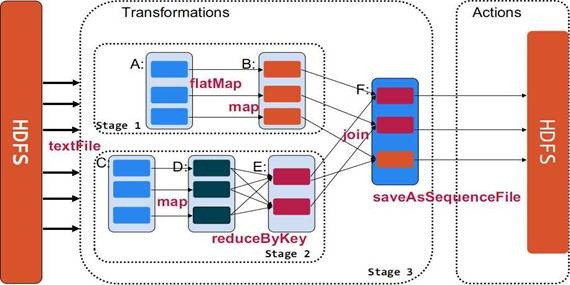
这里要注意的是前面曾经提到过的 Spark 的延迟处理。 Spark 调度器会将 filter 和 map 这两个转换保存到管道，然后一起发送给结点去计算。

**2.3 转换与操作**

对于RDD可以有两种计算方式：转换（返回值还是一个RDD）与操作（返回值不是一个RDD）

转换(Transformations) (如：map, filter, groupBy, join等)，Transformations操作是Lazy的，也就是说从一个RDD转换生成另一个RDD的操作不是马上执行，Spark在遇到Transformations操作时只会记录需要这样的操作，并不会去执行，需要等到有Actions操作的时候才会真正启动计算过程进行计算。

操作(Actions) (如：count, collect, save等)，Actions操作会返回结果或把RDD数据写到存储系统中。Actions是触发Spark启动计算的动因。

[](http://tech.uc.cn/wp-content/uploads/2013/09/SparkTA11.jpg)

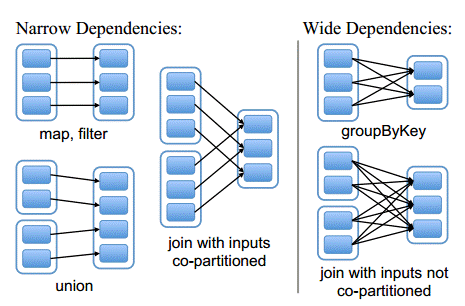
**2.3.1 转换**

|  |  |
| --- | --- |
| reduce(func) | 通过函数func聚集数据集中的所有元素。Func函数接受2个参数，返回一个值。这个函数必须是关联性的，确保可以被正确的并发执行 |
| collect() | 在Driver的程序中，以数组的形式，返回数据集的所有元素。这通常会在使用filter或者其它操作后，返回一个足够小的数据子集再使用，直接将整个RDD集Collect返回，很可能会让Driver程序OOM |
| count() | 返回数据集的元素个数 |
| take(n) | 返回一个数组，由数据集的前n个元素组成。注意，这个操作目前并非在多个节点上，并行执行，而是Driver程序所在机器，单机计算所有的元素(Gateway的内存压力会增大，需要谨慎使用） |
| first() | 返回数据集的第一个元素（类似于take（1） |
| saveAsTextFile(path) | 将数据集的元素，以textfile的形式，保存到本地文件系统，hdfs或者任何其它hadoop支持的文件系统。Spark将会调用每个元素的toString方法，并将它转换为文件中的一行文本 |
| saveAsSequenceFile(path) | 将数据集的元素，以sequencefile的格式，保存到指定的目录下，本地系统，hdfs或者任何其它hadoop支持的文件系统。RDD的元素必须由key-value对组成，并都实现了Hadoop的Writable接口，或隐式可以转换为Writable（Spark包括了基本类型的转换，例如Int，Double，String等等） |
| foreach(func) | 在数据集的每一个元素上，运行函数func。这通常用于更新一个累加器变量，或者和外部存储系统做交互 |

**2.3.2 操作**

|  |  |
| --- | --- |
| map(func) | 返回一个新的分布式数据集，由每个原元素经过func函数转换后组成 |
| filter(func) | 返回一个新的数据集，由经过func函数后返回值为true的原元素组成 |
| flatMap(func) | 类似于map，但是每一个输入元素，会被映射为0到多个输出元素（因此，func函数的返回值是一个Seq，而不是单一元素） |
| flatMap(func) | 类似于map，但是每一个输入元素，会被映射为0到多个输出元素（因此，func函数的返回值是一个Seq，而不是单一元素） |
| sample(withReplacement,  frac, seed) | 根据给定的随机种子seed，随机抽样出数量为frac的数据 |
| union(otherDataset) | 返回一个新的数据集，由原数据集和参数联合而成 |
| groupByKey([numTasks]) | 在一个由（K,V）对组成的数据集上调用，返回一个（K，Seq[V])对的数据集。注意：默认情况下，使用8个并行任务进行分组，你可以传入numTask可选参数，根据数据量设置不同数目的Task |
| reduceByKey(func,  [numTasks]) | 在一个（K，V)对的数据集上使用，返回一个（K，V）对的数据集，key相同的值，都被使用指定的reduce函数聚合到一起。和groupbykey类似，任务的个数是可以通过第二个可选参数来配置的。 |
| join(otherDataset,  [numTasks]) | 在类型为（K,V)和（K,W)类型的数据集上调用，返回一个（K,(V,W))对，每个key中的所有元素都在一起的数据集 |
| groupWith(otherDataset,  [numTasks]) | 在类型为（K,V)和(K,W)类型的数据集上调用，返回一个数据集，组成元素为（K, Seq[V], Seq[W]) Tuples。这个操作在其它框架，称为CoGroup |
| cartesian(otherDataset) | 笛卡尔积。但在数据集T和U上调用时，返回一个(T，U）对的数据集，所有元素交互进行笛卡尔积。 |
| flatMap(func) | 类似于map，但是每一个输入元素，会被映射为0到多个输出元素（因此，func函数的返回值是一个Seq，而不是单一元素） |

**2.4 依赖类型**

在 RDD 中将依赖划分成了两种类型：窄依赖 (Narrow Dependencies) 和宽依赖 (Wide Dependencies) 。窄依赖是指 父 RDD 的每个分区都只被子 RDD 的一个分区所使用 。相应的，那么宽依赖就是指父 RDD 的分区被多个子 RDD 的分区所依赖。例如， Map 就是一种窄依赖，而 Join 则会导致宽依赖 ( 除非父 RDD 是 hash-partitioned，见下图 ) 。  
[](http://images0.cnblogs.com/blog/107289/201508/111511215678897.gif)

窄依赖（Narrow Dependencies ）

  子RDD 的每个分区依赖于常数个父分区（即与数据规模无关）

  输入输出一对一的算子，且结果RDD 的分区结构不变，主要是map 、flatMap

  输入输出一对一，但结果RDD 的分区结构发生了变化，如union 、coalesce

  从输入中选择部分元素的算子，如filter 、distinct 、subtract 、sample

宽依赖（Wide Dependencies ）

  子RDD 的每个分区依赖于所有父RDD 分区

  对单个RDD 基于Key 进行重组和reduce，如groupByKey 、reduceByKey ；

  对两个RDD 基于Key 进行join 和重组，如join

**2.5 RDD缓存**

Spark可以使用 persist 和 cache 方法将任意 RDD 缓存到内存、磁盘文件系统中。缓存是容错的，如果一个 RDD 分片丢失，可以通过构建它的 transformation自动重构。被缓存的 RDD 被使用的时，存取速度会被大大加速。一般的executor内存60%做 cache， 剩下的40%做task。

Spark中，RDD类可以使用cache() 和 persist() 方法来缓存。cache()是persist()的特例，将该RDD缓存到内存中。而persist可以指定一个StorageLevel。StorageLevel的列表可以在StorageLevel 伴生单例对象中找到：

*object StorageLevel {*

*val NONE = new StorageLevel(false, false, false, false)*

*val DISK\_ONLY = new StorageLevel(true, false, false, false)*

*val DISK\_ONLY\_2 = new StorageLevel(true, false, false, false, 2)*

*val MEMORY\_ONLY = new StorageLevel(false, true, false, true)*

*val MEMORY\_ONLY\_2 = new StorageLevel(false, true, false, true, 2)*

*val MEMORY\_ONLY\_SER = new StorageLevel(false, true, false, false)*

*val MEMORY\_ONLY\_SER\_2 = new StorageLevel(false, true, false, false, 2)*

*val MEMORY\_AND\_DISK = new StorageLevel(true, true, false, true)*

*val MEMORY\_AND\_DISK\_2 = new StorageLevel(true, true, false, true, 2)*

*val MEMORY\_AND\_DISK\_SER = new StorageLevel(true, true, false, false)*

*val MEMORY\_AND\_DISK\_SER\_2 = new StorageLevel(true, true, false, false, 2)*

*val OFF\_HEAP = new StorageLevel(false, false, true, false) // Tachyon*

*}*

*// 其中，StorageLevel 类的构造器参数如下：*

*class StorageLevel private(  private var useDisk\_ : Boolean,  private var useMemory\_ : Boolean,  private var useOf*

Spark的不同StorageLevel ，目的满足内存使用和CPU效率权衡上的不同需求。我们建议通过以下的步骤来进行选择：

如果你的RDDs可以很好的与默认的存储级别(MEMORY\_ONLY)契合，就不需要做任何修改了。这已经是CPU使用效率最高的选项，它使得RDDs的操作尽可能的快；

如果不行，试着使用MEMORY\_ONLY\_SER并且选择一个快速序列化的库使得对象在有比较高的空间使用率的情况下，依然可以较快被访问；

尽可能不要存储到硬盘上，除非计算数据集的函数，计算量特别大，或者它们过滤了大量的数据。否则，重新计算一个分区的速度，和与从硬盘中读取基本差不多快；

如果你想有快速故障恢复能力，使用复制存储级别(例如：用Spark来响应web应用的请求)。所有的存储级别都有通过重新计算丢失数据恢复错误的容错机制，但是复制存储级别可以让你在RDD上持续的运行任务，而不需要等待丢失的分区被重新计算；

如果你想要定义你自己的存储级别(比如复制因子为3而不是2)，可以使用StorageLevel 单例对象的apply()方法；

在不使用cached RDD的时候，及时使用unpersist方法来释放它。

**3、RDD动手实战**

在这里我们将对RDD的转换与操作进行动手实战，首先通过实验我们能够观测到转换的懒执行，并通过toDebugString()去查看RDD的LineAge，查看RDD在运行过程中的变换过程，接着演示了从文件读取数据并进行大数据经典的单词计数实验，最后对搜狗提供的搜索数据进行查询，在此过程中演示缓存等操作。

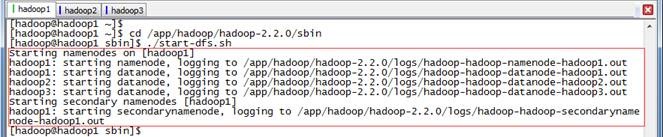
**3.1 启动Spark Shell**

**3.1.1 启动Hadoop**

在随后的实验中将使用到HDFS文件系统，需要进行启动

*$cd /app/hadoop/hadoop-2.2.0/sbin*

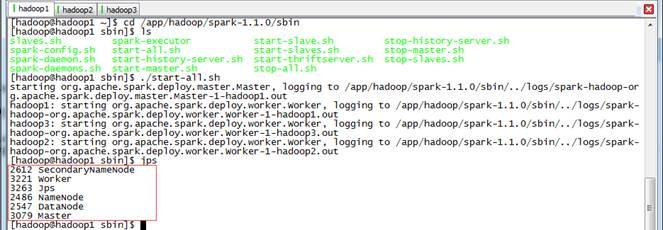
*$./start-dfs.sh*

[](http://images0.cnblogs.com/blog/107289/201508/111511283173235.jpg)

**3.1.2 启动Spark**

*$cd /app/hadoop/spark-1.1.0/sbin*

*$./start-all.sh*

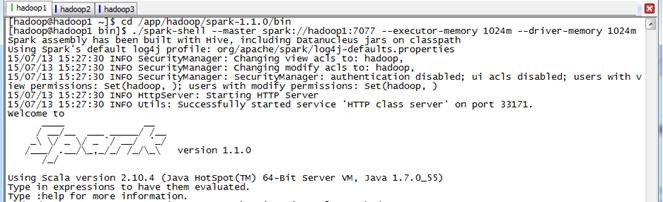
[](http://images0.cnblogs.com/blog/107289/201508/111511356456229.jpg)

**3.1.3 启动Spark Shell**

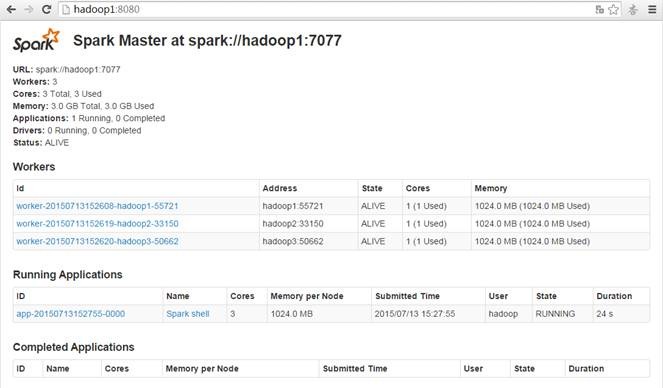
在spark客户端（这里在hadoop1节点)，使用spark-shell连接集群，各个Excetor分配的核数和内存可以根据需要进行指定

*$cd /app/hadoop/spark-1.1.0/bin*

*$./spark-shell --master spark://hadoop1:7077 --executor-memory 1024m --driver-memory 1024m*

[](http://images0.cnblogs.com/blog/107289/201508/111511426922010.jpg)

启动后查看启动情况，如下图所示：

[](http://images0.cnblogs.com/blog/107289/201508/111511472707978.jpg)

**3.2 上传测试数据**

搜狗日志数据可以从http://download.labs.sogou.com/dl/q.html 下载，其中完整版大概2GB左右，文件中字段分别为：访问时间\t用户ID\t[查询词]\t该URL在返回结果中的排名\t用户点击的顺序号\t用户点击的URL。其中SogouQ1.txt、SogouQ2.txt、SogouQ3.txt分别是用head -n 或者tail -n 从SogouQ数据日志文件中截取，分别包含100万，200万和1000万笔数据，这些测试数据也放在该系列配套资源的data\sogou目录下。

搜狗日志数据放在data\sogou下，把该目录下的SogouQ1.txt、SogouQ2.txt和SogouQ3.txt解压，然后通过下面的命令上传到HDFS中/sogou目录中

*cd /home/hadoop/upload/*

*ll sogou*

*tar -zxf \*.gz*

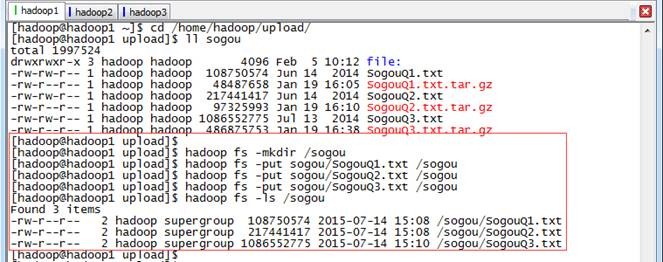
*hadoop fs -mkdir /sogou*

*hadoop fs -put sogou/SogouQ1.txt /sogou*

*hadoop fs -put sogou/SogouQ2.txt /sogou*

*hadoop fs -put sogou/SogouQ3.txt /sogou*

*hadoop fs -ls /sogou*

*[](http://images0.cnblogs.com/blog/107289/201508/111511564891129.jpg)*

**3.3 转换与操作**

**3.3.1 并行化集合例子演示**

在该例子中通过parallelize方法定义了一个从1~10的数据集，然后通过map(\_\*2)对数据集每个数乘以2，接着通过filter(\_%3==0)过滤被3整除的数字，最后使用toDebugString显示RDD的LineAge，并通过collect计算出最终的结果。

*val num=sc.parallelize(1 to 10)*

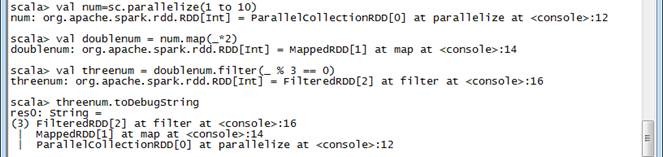
*val doublenum = num.map(\_\*2)*

*val threenum = doublenum.filter(\_ % 3 == 0)*

*threenum.toDebugString*

*threenum.collect*

在下图运行结果图中，我们可以看到RDD的LineAge演变，通过paralelize方法建立了一个ParalleCollectionRDD，使用map()方法后该RDD为MappedRDD，接着使用filter()方法后转变为FilteredRDD。

*[](http://images0.cnblogs.com/blog/107289/201508/111512023791895.jpg)*

在下图中使用collect方法时触发运行作业，通过任务计算出结果

*[](http://images0.cnblogs.com/blog/107289/201508/111512079102406.jpg)*

以下语句和collect一样，都会触发作业运行

*num.reduce (\_ + \_)*

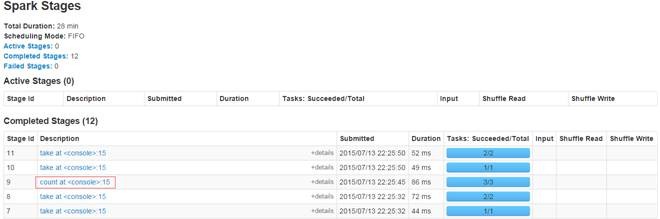
*num.take(5)*

*num.first*

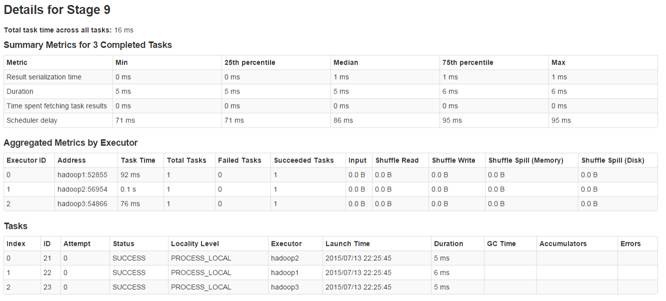
*num.count*

*num.take(5).foreach(println)*

运行的情况可以通过页面进行监控，在Spark Stages页面中我们可以看到运行的详细情况，包括运行的Stage id号、Job描述、提交时间、运行时间、Stage情况等，可以点击作业描述查看更加详细的情况：

[](http://images0.cnblogs.com/blog/107289/201508/111512136769443.jpg)

在这个页面上我们将看到三部分信息：作业的基本信息、Executor信息和Tasks的信息。特别是Tasks信息可以了解到作业的分片情况，运行状态、数据获取位置、耗费时间及所处的Executor等信息

[](http://images0.cnblogs.com/blog/107289/201508/111512182546411.jpg)

**3.3.2 Shuffle操作例子演示**

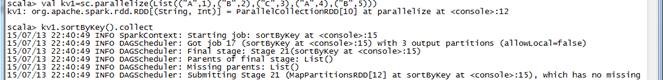
在该例子中通过parallelize方法定义了K-V键值对数据集合，通过sortByKey()进行按照Key值进行排序，然后通过collect方法触发作业运行得到结果。groupByKey()为按照Key进行归组，reduceByKey(\_+\_)为按照Key进行累和，这三个方法的计算和前面的例子不同，因为这些RDD类型为宽依赖，在计算过程中发生了Shuffle动作。

*val kv1=sc.parallelize(List(("A",1),("B",2),("C",3),("A",4),("B",5)))*

*kv1.sortByKey().collect*

*kv1.groupByKey().collect*

*kv1.reduceByKey(\_+\_).collect*

[](http://images0.cnblogs.com/blog/107289/201508/111512219576981.jpg)

[clip_image031](http://images0.cnblogs.com/blog/107289/201508/111512247544292.jpg)

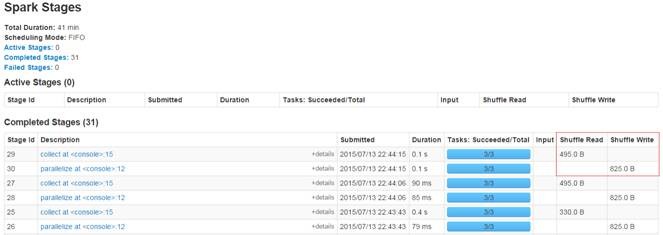
调用groupByKey()运行结果

[clip_image033](http://images0.cnblogs.com/blog/107289/201508/111512279894705.jpg)

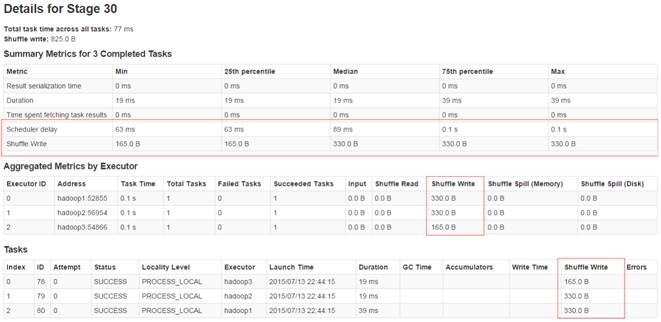
调用reduceByKey ()运行结果

[clip_image035](http://images0.cnblogs.com/blog/107289/201508/111512311149917.jpg)

我们在作业运行监控界面上能够看到：每个作业分为两个Stage，在第一个Stage中进行了Shuffle Write，在第二个Stage中进行了Shuffle Read。

[](http://images0.cnblogs.com/blog/107289/201508/111512335828730.jpg)

在Stage详细运行页面中可以观察第一个Stage运行情况，内容包括：Stage运行的基本信息、每个Executor运行信息和任务的运行信息，特别在任务运行中我们可以看到任务的状态、数据读取的位置、机器节点、耗费时间和Shuffle Write时间等。

[](http://images0.cnblogs.com/blog/107289/201508/111512389737582.jpg)

在下面进行了distinct、union、join和cogroup等操作中涉及到Shuffle过程

*val kv2=sc.parallelize(List(("A",4),("A",4),("C",3),("A",4),("B",5)))*

*kv2.distinct.collect*

*kv1.union(kv2).collect*

*val kv3=sc.parallelize(List(("A",10),("B",20),("D",30)))*

*kv1.join(kv3).collect*

*kv1.cogroup(kv3).collect*

**3.3.3文件例子读取**

这个是大数据经典的例子，在这个例子中通过不同方式读取HDFS中的文件，然后进行单词计数，最终通过运行作业计算出结果。本例子中通过toDebugString可以看到RDD的变化，

第一步   按照文件夹读取计算每个单词出现个数

在该步骤中RDD的变换过程为：HadoopRDD->MappedRDD-> FlatMappedRDD->MappedRDD->PairRDDFunctions->ShuffleRDD->MapPartitionsRDD

*val text = sc.textFile("hdfs://hadoop1:9000/class3/directory/")*

*text.toDebugString*

*val words=text.flatMap(\_.split(" "))*

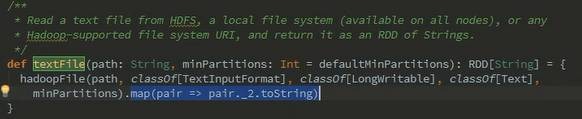
*val wordscount=words.map(x=>(x,1)).reduceByKey(\_+\_)*

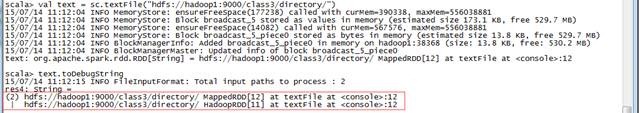
*wordscount.toDebugString*

*wordscount.collect*

RDD类型的变化过程如下：

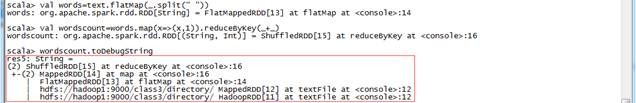
  首先使用textFile()读取HDFS数据形成MappedRDD，这里有可能有疑问，从HDFS读取的数据不是HadoopRDD，怎么变成了MappedRDD。回答这个问题需要从Spark源码进行分析，在sparkContext类中的textFile()方法读取HDFS文件后，使用了map()生成了MappedRDD。

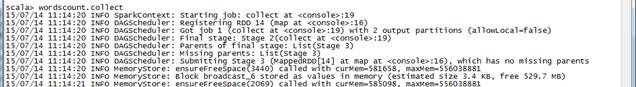
[](http://images0.cnblogs.com/blog/107289/201508/111512431141851.jpg)

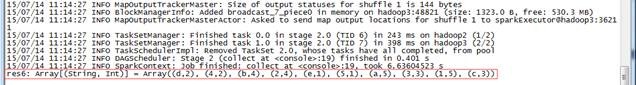
[](http://images0.cnblogs.com/blog/107289/201508/111512470043961.jpg)

  然后使用flatMap()方法对文件内容按照空格拆分单词，拆分形成FlatMappedRDD

  其次使用map(x=>(x(1),1))对上步骤拆分的单词形成（单词，1）数据对，此时生成的MappedRDD，最后使用reduceByKey()方法对单词的频度统计，由此生成ShuffledRDD，并由collect运行作业得出结果。

[](http://images0.cnblogs.com/blog/107289/201508/111512500046445.jpg)

[](http://images0.cnblogs.com/blog/107289/201508/111512556922442.jpg)

[](http://images0.cnblogs.com/blog/107289/201508/111513005358095.jpg)

第二步   按照匹配模式读取计算单词个数

*val rdd2 = sc.textFile("hdfs://hadoop1:9000/class3/directory/\*.txt")*

*rdd2.flatMap(\_.split(" ")).map(x=>(x,1)).reduceByKey(\_+\_).collect*

第三步   读取gz压缩文件计算单词个数

*val rdd3 = sc.textFile("hdfs://hadoop1:8000/class2/test.txt.gz")*

*rdd3.flatMap(\_.split(" ")).map(x=>(x,1)).reduceByKey(\_+\_).collect*

**3.3.4 搜狗日志查询例子演示**

搜狗日志数据可以从http://download.labs.sogou.com/dl/q.html 下载，其中完整版大概2GB左右，文件中字段分别为：访问时间\t用户ID\t[查询词]\t该URL在返回结果中的排名\t用户点击的顺序号\t用户点击的URL。其中SogouQ1.txt、SogouQ2.txt、SogouQ3.txt分别是用head -n 或者tail -n 从SogouQ数据日志文件中截取，分别包含100万，200万和1000万笔数据，这些测试数据也放在该系列配套资源的data\sogou目录下。

第一步   上传测试数据

搜狗日志数据放在data\sogou下，把该目录下的SogouQ1.txt、SogouQ2.txt和SogouQ3.txt解压，然后通过下面的命令上传到HDFS中/sogou目录中

*cd /home/hadoop/upload/*

*ll sogou*

*tar -zxf \*.gz*

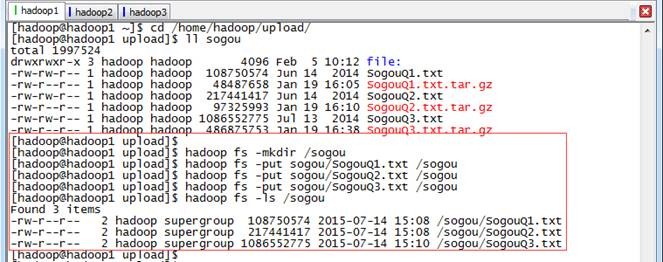
*hadoop fs -mkdir /sogou*

*hadoop fs -put sogou/SogouQ1.txt /sogou*

*hadoop fs -put sogou/SogouQ2.txt /sogou*

*hadoop fs -put sogou/SogouQ3.txt /sogou*

*hadoop fs -ls /sogou*

*[](http://images0.cnblogs.com/blog/107289/201508/111513076769146.jpg)*

第二步   查询搜索结果排名第1点击次序排在第2的数据

val rdd1 = sc.textFile("hdfs://hadoop1:9000/sogou/SogouQ1.txt")

*val rdd2=rdd1.map(\_.split("\t")).filter(\_.length==6)*

*rdd2.count()*

*val rdd3=rdd2.filter(\_(3).toInt==1).filter(\_(4).toInt==2)*

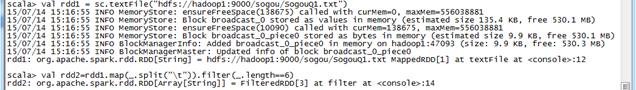
*rdd3.count()*

*rdd3.toDebugString*

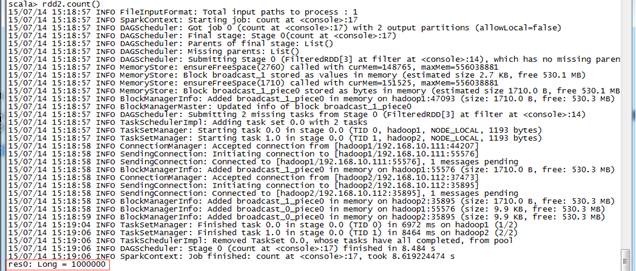
该命令运行的过程如下：

  首先使用textFile()读入SogouQ1.txt文件，读入后由HadoopRDD转变为MadppedRDD；

  然后通过rdd1.map(\_.split("\t"))对读入数据使用\t分隔符进行拆分，拆分后RDD类型不变即为MadppedRDD，对这些拆分后的数据使用filter(\_.length==6)过滤每行为6个字段的数据，这时数据变为FilteredRDD；

*[](http://images0.cnblogs.com/blog/107289/201508/111513183324525.jpg)*

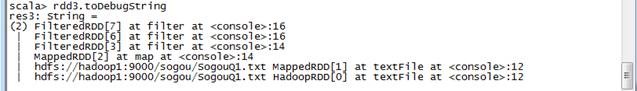
  运行rdd2.count()启动对rdd2计数的作业，通过运行结果可以看到该数据集为100条；

*[](http://images0.cnblogs.com/blog/107289/201508/111513263798432.jpg)*

  rdd2.filter(\_(3).toInt==1).filter(\_(4).toInt==2)表示对rdd2的数据的第4个字段搜索结果排名第一，第5个字段点击次序排在第二的数据进行过滤，通过count()方法运行作业得出最终的结果；

*[](http://images0.cnblogs.com/blog/107289/201508/111513356456569.jpg)*

使用toDebugString可以查看rdd3的RDD详细变换过程，如下图所示：

*[](http://images0.cnblogs.com/blog/107289/201508/111513396765637.jpg)*

第三步   Session查询次数排行榜并把结果保存在HDFS中

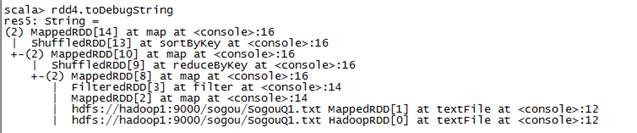
*val rdd4 = rdd2.map(x=>(x(1),1)).reduceByKey(\_+\_).map(x=>(x.\_2,x.\_1)). sortByKey(false).map(x=>(x.\_2,x.\_1))*

*rdd4.toDebugString*

*rdd4.saveAsTextFile("hdfs://hadoop1:9000/class3/output1")*

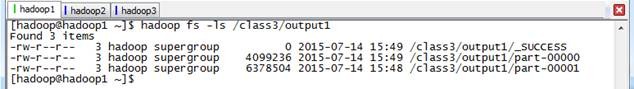
该命令运行的过程如下：

  rdd4的生成比较复杂，我们分步骤进行解析，轴线map(x=>(x(1),1))是获取每行的第二个字段（用户Session）计数为1，然后reduceByKey(\_+\_)是安排Key进行累和，即按照用户Session号进行计数求查询次数，其次map(x=>(x.\_2,x.\_1))是把Key和Value位置互换，为后面排序提供条件，使用sortByKey(false)对数据进行按Key值进行倒排，此时需要注意的是Key为查询次数，最后通过map(x=>(x.\_2,x.\_1)再次交换Key和Value位置，得到了（用户Session号，查询次数）结果。该过程RDD的变化如下图所示：

*[](http://images0.cnblogs.com/blog/107289/201508/111513422851408.jpg)*

  计算的结果通过如下命令可以查看到，可以看到由于输入数据存放在2个节点上，所以结果也分为两个文件

*hadoop fs -ls /class3/output1*

[](http://images0.cnblogs.com/blog/107289/201508/111513460985775.jpg)

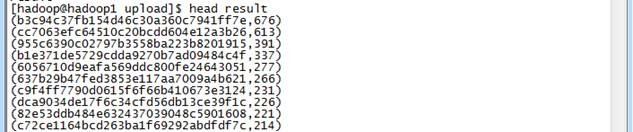
这是使用HDFS的getmerge合并这两个文件并进行查看

*$cd /app/hadoop/hadoop-2.2.0/bin*

*$hdfs dfs -getmerge hdfs://hadoop1:9000/class3/output1 /home/hadoop/upload/result*

*$cd /home/hadoop/upload/*

*$head result*

*[](http://images0.cnblogs.com/blog/107289/201508/111513490354745.jpg)*