

Spark 编程模型 (上)

--概念及 Shell 试验

目录

1 SPARK编程模型	3
1.1 术语定义.....	3
1.2 模型组成.....	3
1.2.1 Driver部分.....	3
1.2.2 Executor部分.....	4
2 RDD	5
2.1 术语定义.....	5
2.2 RDD概念.....	6
2.2.1 RDD的特点.....	6
2.2.2 RDD基础数据类型.....	6
2.2.3 例子：控制台日志挖掘.....	8
2.3 转换与操作.....	8
2.3.1 转换.....	9
2.3.2 操作.....	10
2.4 依赖类型.....	10
2.5 RDD缓存.....	11
3 RDD动手实验	13
3.1 启动SPARK SHELL.....	13
3.1.1 启动Hadoop.....	13
3.1.2 启动Spark.....	13
3.1.3 启动Spark Shell.....	14
3.2 转换与操作.....	15
3.2.1 并行化集合例子演示.....	15
3.2.2 Shuffle操作例子演示.....	17
3.2.3 文件例子读取.....	19
3.2.4 搜狗日志查询例子演示.....	21
3.2.5 缓存例子演示.....	错误！未定义书签。

Spark 编程模型 (上)

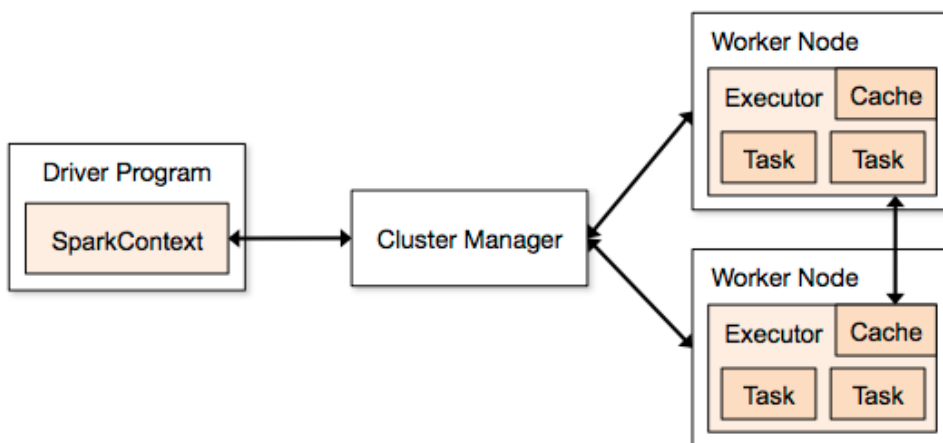
1 Spark 编程模型

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



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 partition 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))
```

- **累加器** : 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)** : Resilient 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。

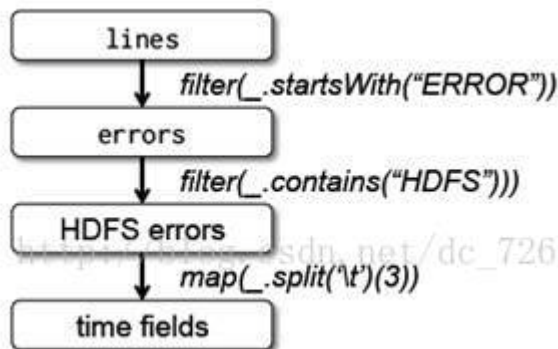
此外, 通过 Transformation 可以将 HadoopRDD 等转换成 FilterRDD(依赖一个父 RDD 产生) 和 JoinedRDD(依赖所有父 RDD) 等。

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：

```
1 lines = spark.textFile("hdfs://...")
2 errors = lines.filter(_.startsWith("ERROR"))
3 errors.persist()
4
5 // Count errors mentioning MySQL
6 errors.filter(_.contains("MySQL")).count()
7
8 // Return time fields of errors mentioning
9 // HDFS as an array (assuming time is field
10 // number 3 in a tab-separated format):
11 errors.filter(_.contains("HDFS"))
12     .map(_.split('\t')(3))
13     .collect()
```

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。



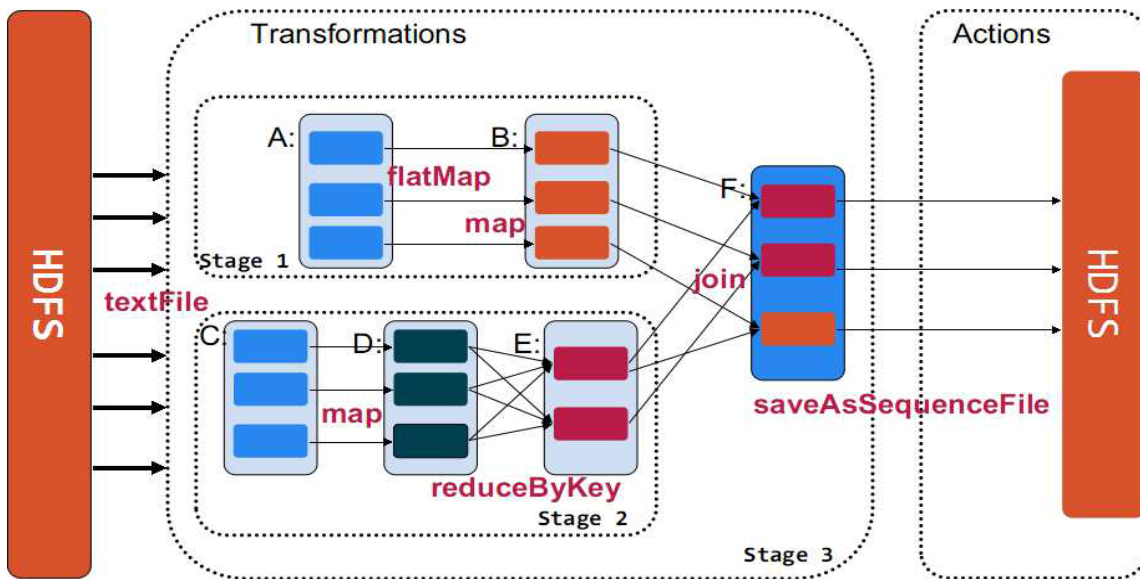
这里要注意的是前面曾经提到过的 Spark 的延迟处理。Spark 调度器会将 filter 和 map 这两个转换保存到管道，然后一起发送给结点去计算。

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。



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
---------------	---

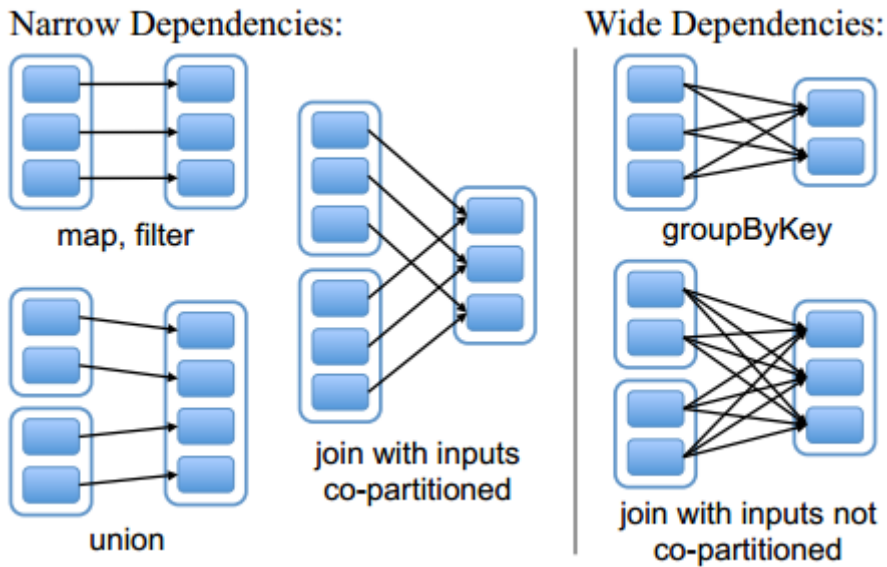
2.3.2 操作

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）。



- 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。

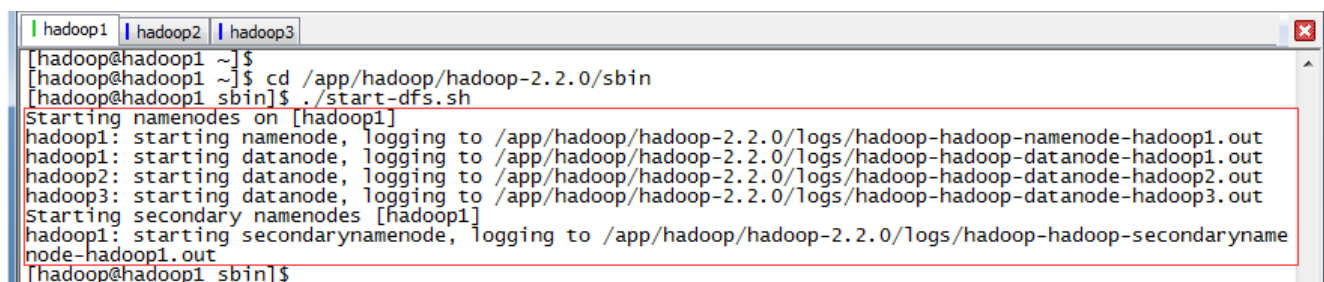
3.1 启动 Spark Shell

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
```

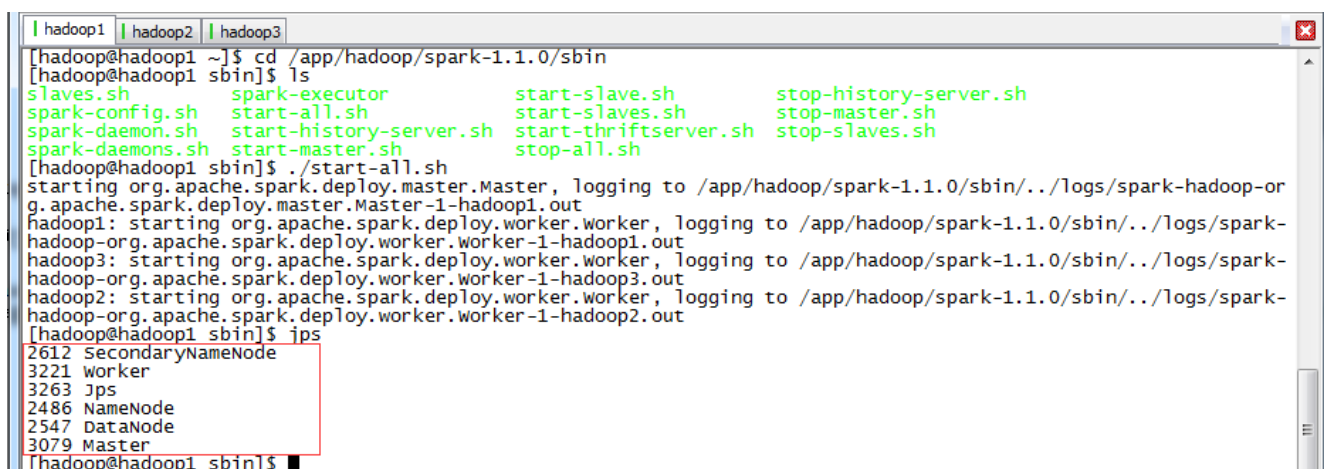


```
hadoop1 | hadoop2 | hadoop3
[hadoop@hadoop1 ~]$
[hadoop@hadoop1 ~]$ cd /app/hadoop/hadoop-2.2.0/sbin
[hadoop@hadoop1 sbin]$ ./start-dfs.sh
Starting namenodes on [hadoop1]
hadoop1: starting namenode, logging to /app/hadoop/hadoop-2.2.0/logs/hadoop-hadoop-namenode-hadoop1.out
hadoop1: starting datanode, logging to /app/hadoop/hadoop-2.2.0/logs/hadoop-hadoop-datanode-hadoop1.out
hadoop2: starting datanode, logging to /app/hadoop/hadoop-2.2.0/logs/hadoop-hadoop-datanode-hadoop2.out
hadoop3: starting datanode, logging to /app/hadoop/hadoop-2.2.0/logs/hadoop-hadoop-datanode-hadoop3.out
Starting secondary namenodes [hadoop1]
hadoop1: starting secondarynamenode, logging to /app/hadoop/hadoop-2.2.0/logs/hadoop-hadoop-secondaryname
node-hadoop1.out
[hadoop@hadoop1 sbin]$
```

3.1.2 启动 Spark

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

```
./start-all.sh
```



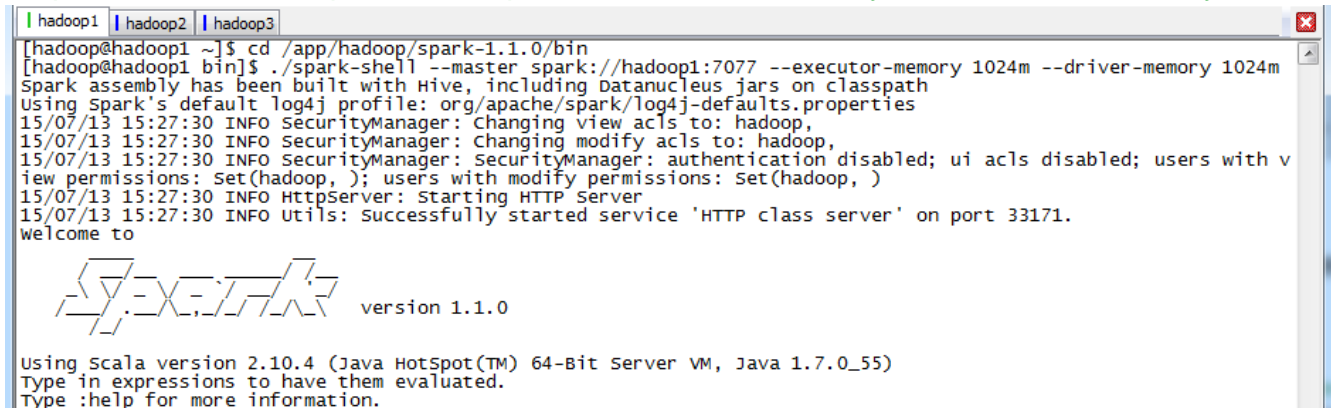
```
hadoop1 | hadoop2 | hadoop3
[hadoop@hadoop1 ~]$ cd /app/hadoop/spark-1.1.0/sbin
[hadoop@hadoop1 sbin]$ ls
slaves.sh          spark-executor      start-slave.sh      stop-history-server.sh
spark-config.sh   start-all.sh       start-slaves.sh     stop-master.sh
spark-daemon.sh   start-history-server.sh start-thriftserver.sh stop-slaves.sh
spark-daemons.sh start-master.sh     stop-all.sh
[hadoop@hadoop1 sbin]$ ./start-all.sh
starting org.apache.spark.deploy.master.Master, logging to /app/hadoop/spark-1.1.0/sbin/../logs/spark-hadoop-or
g.apache.spark.deploy.master.Master-1-hadoop1.out
hadoop1: starting org.apache.spark.deploy.worker.worker, logging to /app/hadoop/spark-1.1.0/sbin/../logs/spark-
hadoop-org.apache.spark.deploy.worker.worker-1-hadoop1.out
hadoop3: starting org.apache.spark.deploy.worker.worker, logging to /app/hadoop/spark-1.1.0/sbin/../logs/spark-
hadoop-org.apache.spark.deploy.worker.worker-1-hadoop3.out
hadoop2: starting org.apache.spark.deploy.worker.worker, logging to /app/hadoop/spark-1.1.0/sbin/../logs/spark-
hadoop-org.apache.spark.deploy.worker.worker-1-hadoop2.out
[hadoop@hadoop1 sbin]$ jps
2612 SecondaryNameNode
3221 worker
3263 Jps
2486 NameNode
2547 DataNode
3079 Master
[hadoop@hadoop1 sbin]$
```

3.1.3 启动 Spark Shell

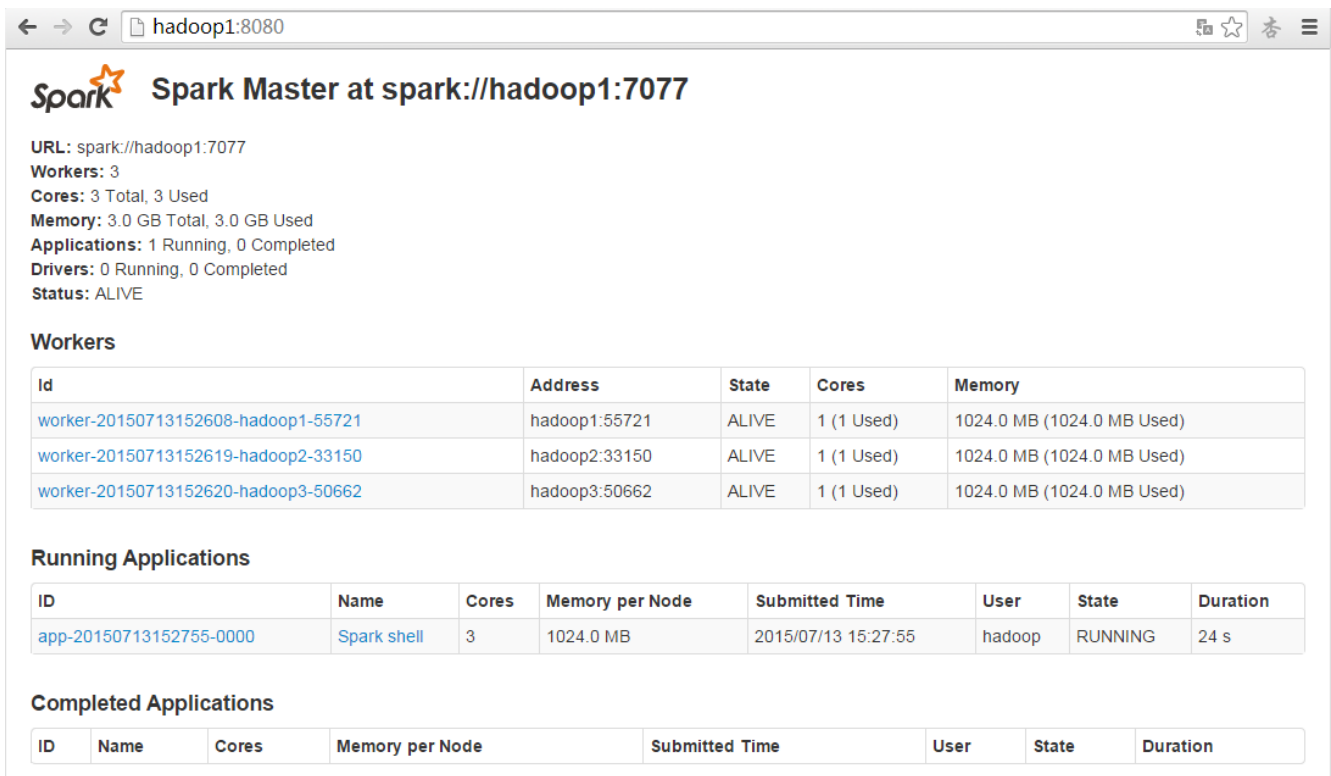
在 spark 客户端 (这里在 hadoop1 节点) , 使用 spark-shell 连接集群 , 各个 Excetor 分配的核数和内存可以根据需要进行指定

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

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



启动后查看启动情况 , 如下图所示 :



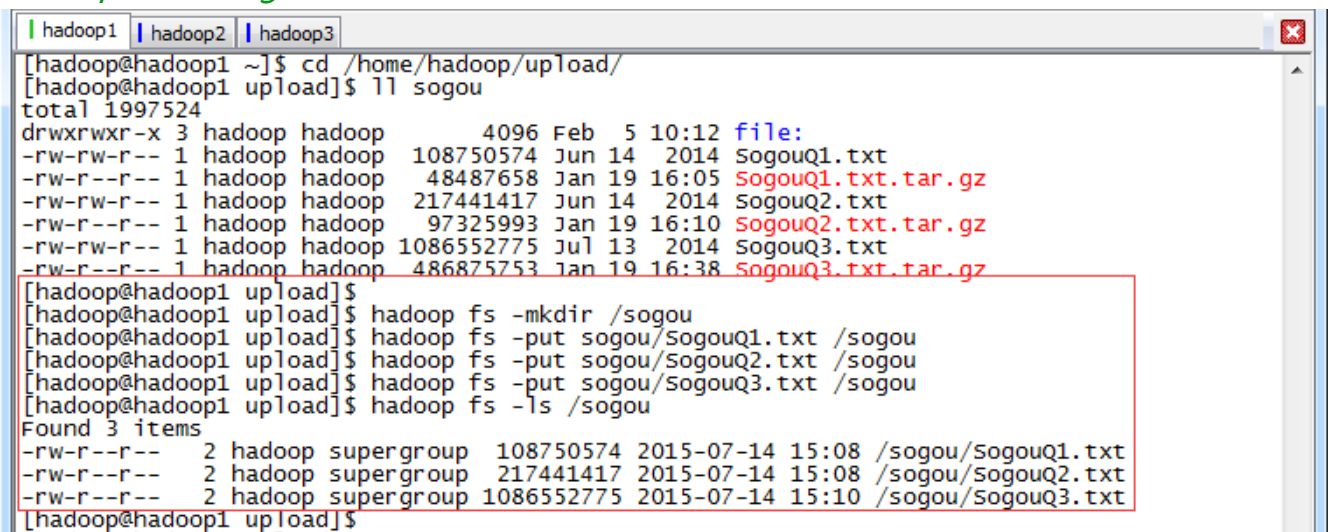
3.2 上传测试数据

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
```



```
hadoop1 | hadoop2 | hadoop3  
[hadoop@hadoop1 ~]$ cd /home/hadoop/upload/  
[hadoop@hadoop1 upload]$ ll sogou  
total 1997524  
drwxrwxr-x 3 hadoop hadoop 4096 Feb 5 10:12 file:  
-rw-rw-r-- 1 hadoop hadoop 108750574 Jun 14 2014 SogouQ1.txt  
-rw-r--r-- 1 hadoop hadoop 48487658 Jan 19 16:05 SogouQ1.txt.tar.gz  
-rw-rw-r-- 1 hadoop hadoop 217441417 Jun 14 2014 SogouQ2.txt  
-rw-r--r-- 1 hadoop hadoop 97325993 Jan 19 16:10 SogouQ2.txt.tar.gz  
-rw-rw-r-- 1 hadoop hadoop 1086552775 Jul 13 2014 SogouQ3.txt  
-rw-r--r-- 1 hadoop hadoop 486875753 Jan 19 16:38 SogouQ3.txt.tar.gz  
[hadoop@hadoop1 upload]$  
[hadoop@hadoop1 upload]$ hadoop fs -mkdir /sogou  
[hadoop@hadoop1 upload]$ hadoop fs -put sogou/SogouQ1.txt /sogou  
[hadoop@hadoop1 upload]$ hadoop fs -put sogou/SogouQ2.txt /sogou  
[hadoop@hadoop1 upload]$ hadoop fs -put sogou/SogouQ3.txt /sogou  
[hadoop@hadoop1 upload]$ hadoop fs -ls /sogou  
Found 3 items  
-rw-r--r-- 2 hadoop supergroup 108750574 2015-07-14 15:08 /sogou/SogouQ1.txt  
-rw-r--r-- 2 hadoop supergroup 217441417 2015-07-14 15:08 /sogou/SogouQ2.txt  
-rw-r--r-- 2 hadoop supergroup 1086552775 2015-07-14 15:10 /sogou/SogouQ3.txt  
[hadoop@hadoop1 upload]$
```

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。

```
scala> val num=sc.parallelize(1 to 10)
num: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[0] at parallelize at <console>:12

scala> val doublenum = num.map(_*2)
doublenum: org.apache.spark.rdd.RDD[Int] = MappedRDD[1] at map at <console>:14

scala> val threenum = doublenum.filter(_ % 3 == 0)
threenum: org.apache.spark.rdd.RDD[Int] = FilteredRDD[2] at filter at <console>:16

scala> threenum.toDebugString
res0: String =
(3) FilteredRDD[2] at filter at <console>:16
  | MappedRDD[1] at map at <console>:14
  | ParallelCollectionRDD[0] at parallelize at <console>:12
```

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

```
scala> threenum.collect
15/07/13 22:09:45 INFO SparkContext: Starting job: collect at <console>:19
15/07/13 22:09:45 INFO DAGScheduler: Got job 3 (collect at <console>:19) with 3 output partitions (allowLocal=false)
15/07/13 22:09:45 INFO DAGScheduler: Final stage: Stage 3(collect at <console>:19)
15/07/13 22:09:45 INFO DAGScheduler: Parents of final stage: List()
15/07/13 22:09:45 INFO DAGScheduler: Missing parents: List()
15/07/13 22:09:45 INFO DAGScheduler: Submitting stage 3 (FilteredRDD[2] at filter at <console>:16), which has no missing parents
15/07/13 22:09:45 INFO MemoryStore: ensureFreeSpace(1856) called with curMem=9162, maxMem=556038881
15/07/13 22:09:45 INFO MemoryStore: Block broadcast_3 stored as values in memory (estimated size 1856.0 B, free 530.3 MB)
15/07/13 22:09:45 INFO MemoryStore: ensureFreeSpace(1198) called with curMem=11018, maxMem=556038881
15/07/13 22:09:45 INFO MemoryStore: Block broadcast_3_piece0 stored as bytes in memory (estimated size 1198.0 B, free 530.3 MB)
15/07/13 22:09:45 INFO BlockManagerInfo: Added broadcast_3_piece0 in memory on hadoop1:58290 (size: 1198.0 B, free: 530.3 MB)
15/07/13 22:09:45 INFO BlockManagerMaster: Updated info of block broadcast_3_piece0
15/07/13 22:09:45 INFO DAGScheduler: Submitting 3 missing tasks from stage 3 (FilteredRDD[2] at filter at <console>:16)
15/07/13 22:09:45 INFO TaskSchedulerImpl: Adding task set 3.0 with 3 tasks
15/07/13 22:09:45 INFO TaskSetManager: Starting task 0.0 in stage 3.0 (TID 9, hadoop3, PROCESS_LOCAL, 1152 bytes)
15/07/13 22:09:45 INFO TaskSetManager: Starting task 1.0 in stage 3.0 (TID 10, hadoop1, PROCESS_LOCAL, 1152 bytes)
15/07/13 22:09:45 INFO TaskSetManager: Starting task 2.0 in stage 3.0 (TID 11, hadoop2, PROCESS_LOCAL, 1152 bytes)
15/07/13 22:09:45 INFO BlockManagerInfo: Added broadcast_3_piece0 in memory on hadoop3:54866 (size: 1198.0 B, free: 530.3 MB)
15/07/13 22:09:45 INFO BlockManagerInfo: Added broadcast_3_piece0 in memory on hadoop2:56954 (size: 1198.0 B, free: 530.3 MB)
15/07/13 22:09:45 INFO BlockManagerInfo: Added broadcast_3_piece0 in memory on hadoop1:52855 (size: 1198.0 B, free: 530.3 MB)
15/07/13 22:09:45 INFO TaskSetManager: Finished task 0.0 in stage 3.0 (TID 9) in 83 ms on hadoop3 (1/3)
15/07/13 22:09:45 INFO TaskSetManager: Finished task 2.0 in stage 3.0 (TID 11) in 92 ms on hadoop2 (2/3)
15/07/13 22:09:45 INFO TaskSetManager: Finished task 1.0 in stage 3.0 (TID 10) in 109 ms on hadoop1 (3/3)
15/07/13 22:09:45 INFO TaskSchedulerImpl: Removed TaskSet 3.0, whose tasks have all completed, from pool
15/07/13 22:09:45 INFO DAGScheduler: Stage 3 (collect at <console>:19) finished in 0.110 s
15/07/13 22:09:45 INFO SparkContext: Job finished: collect at <console>:19, took 0.126929098 s
res5: Array[Int] = Array(6, 12, 18)
```

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：

Spark Stages

Total Duration: 28 min
Scheduling Mode: FIFO
Active Stages: 0
Completed Stages: 12
Failed Stages: 0

Active Stages (0)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Shuffle Read	Shuffle Write
----------	-------------	-----------	----------	------------------------	-------	--------------	---------------

Completed Stages (12)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Shuffle Read	Shuffle Write
11	take at <console>:15	+details 2015/07/13 22:25:50	52 ms	2/2			
10	take at <console>:15	+details 2015/07/13 22:25:50	49 ms	1/1			
9	count at <console>:15	+details 2015/07/13 22:25:45	86 ms	3/3			
8	take at <console>:15	+details 2015/07/13 22:25:32	72 ms	2/2			
7	take at <console>:15	+details 2015/07/13 22:25:32	44 ms	1/1			

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

Details for Stage 9

Total task time across all tasks: 16 ms

Summary Metrics for 3 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Result serialization time	0 ms	0 ms	1 ms	1 ms	1 ms
Duration	5 ms	5 ms	5 ms	6 ms	6 ms
Time spent fetching task results	0 ms	0 ms	0 ms	0 ms	0 ms
Scheduler delay	71 ms	71 ms	86 ms	95 ms	95 ms

Aggregated Metrics by Executor

Executor ID	Address	Task Time	Total Tasks	Failed Tasks	Succeeded Tasks	Input	Shuffle Read	Shuffle Write	Shuffle Spill (Memory)	Shuffle Spill (Disk)
0	hadoop1:52855	92 ms	1	0	1	0.0 B	0.0 B	0.0 B	0.0 B	0.0 B
1	hadoop2:56954	0.1 s	1	0	1	0.0 B	0.0 B	0.0 B	0.0 B	0.0 B
2	hadoop3:54866	76 ms	1	0	1	0.0 B	0.0 B	0.0 B	0.0 B	0.0 B

Tasks

Index	ID	Attempt	Status	Locality Level	Executor	Launch Time	Duration	GC Time	Accumulators	Errors
0	21	0	SUCCESS	PROCESS_LOCAL	hadoop2	2015/07/13 22:25:45	5 ms			
1	22	0	SUCCESS	PROCESS_LOCAL	hadoop1	2015/07/13 22:25:45	6 ms			
2	23	0	SUCCESS	PROCESS_LOCAL	hadoop3	2015/07/13 22:25:45	5 ms			

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，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
```

```
scala> val kv1=sc.parallelize(List(("A",1),("B",2),("C",3),("A",4),("B",5)))
kv1: org.apache.spark.rdd.RDD[(String, Int)] = ParallelCollectionRDD[10] at parallelize at <console>:12

scala> kv1.sortByKey().collect
15/07/13 22:40:49 INFO SparkContext: Starting job: sortByKey at <console>:15
15/07/13 22:40:49 INFO DAGScheduler: Got job 17 (sortByKey at <console>:15) with 3 output partitions (allowLocal=false)
15/07/13 22:40:49 INFO DAGScheduler: Final stage: Stage 21(sortByKey at <console>:15)
15/07/13 22:40:49 INFO DAGScheduler: Parents of final stage: List()
15/07/13 22:40:49 INFO DAGScheduler: Missing parents: List()
15/07/13 22:40:49 INFO DAGScheduler: Submitting Stage 21 (MapPartitionsRDD[12] at sortByKey at <console>:15), which has no missing
15/07/13 22:40:49 INFO TaskSetManager: Finished task 2.0 in stage 22.0 (TID 62) in 109 ms on hadoop2 (1/3)
15/07/13 22:40:49 INFO TaskSetManager: Finished task 0.0 in stage 22.0 (TID 60) in 115 ms on hadoop3 (2/3)
15/07/13 22:40:49 INFO TaskSetManager: Finished task 1.0 in stage 22.0 (TID 61) in 130 ms on hadoop1 (3/3)
15/07/13 22:40:49 INFO TaskSchedulerImpl: Removed TaskSet 22.0, whose tasks have all completed, from pool
15/07/13 22:40:49 INFO DAGScheduler: Stage 22 (collect at <console>:15) finished in 0.134 s
15/07/13 22:40:49 INFO SparkContext: Job finished: collect at <console>:15, took 0.257030802 s
res15: Array[(String, Int)] = Array((A,4), (A,1), (B,2), (B,5), (C,3))
```

调用 groupByKey()运行结果

```
15/07/13 22:44:06 INFO TaskSetManager: Finished task 2.0 in stage 27.0 (TID 77) in 108 ms on hadoop1 (3/3)
15/07/13 22:44:06 INFO TaskSchedulerImpl: Removed TaskSet 27.0, whose tasks have all completed, from pool
15/07/13 22:44:06 INFO DAGScheduler: Stage 27 (collect at <console>:15) finished in 0.090 s
15/07/13 22:44:06 INFO SparkContext: Job finished: collect at <console>:15, took 0.222751608 s
res17: Array[(String, Iterable[Int])] = Array((B,CompactBuffer(2, 5)), (C,CompactBuffer(3)), (A,CompactBuffer(4, 1)))
```

调用 reduceByKey ()运行结果

```
15/07/13 22:44:15 INFO TaskSetManager: Finished task 0.0 in stage 29.0 (TID 81) in 105 ms on hadoop3 (2/3)
15/07/13 22:44:15 INFO TaskSetManager: Finished task 2.0 in stage 29.0 (TID 83) in 105 ms on hadoop1 (3/3)
15/07/13 22:44:15 INFO DAGScheduler: Stage 29 (collect at <console>:15) finished in 0.109 s
15/07/13 22:44:15 INFO TaskSchedulerImpl: Removed TaskSet 29.0, whose tasks have all completed, from pool
15/07/13 22:44:15 INFO SparkContext: Job finished: collect at <console>:15, took 0.266078984 s
res18: Array[(String, Int)] = Array((B,7), (C,3), (A,5))
```

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。

Spark Stages

Total Duration: 41 min
 Scheduling Mode: FIFO
 Active Stages: 0
 Completed Stages: 31
 Failed Stages: 0

Active Stages (0)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Shuffle Read	Shuffle Write
----------	-------------	-----------	----------	------------------------	-------	--------------	---------------

Completed Stages (31)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Shuffle Read	Shuffle Write
29	collect at <console>:15	+details 2015/07/13 22:44:15	0.1 s	3/3		495.0 B	
30	parallelize at <console>:12	+details 2015/07/13 22:44:15	0.1 s	3/3			825.0 B
27	collect at <console>:15	+details 2015/07/13 22:44:06	90 ms	3/3		495.0 B	
28	parallelize at <console>:12	+details 2015/07/13 22:44:06	85 ms	3/3			825.0 B
25	collect at <console>:15	+details 2015/07/13 22:43:43	0.4 s	3/3		330.0 B	
26	parallelize at <console>:12	+details 2015/07/13 22:43:43	79 ms	3/3			825.0 B

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。

Details for Stage 30

Total task time across all tasks: 77 ms

Shuffle write: 825.0 B

Summary Metrics for 3 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Result serialization time	0 ms	0 ms	0 ms	0 ms	0 ms
Duration	19 ms	19 ms	19 ms	39 ms	39 ms
Time spent fetching task results	0 ms	0 ms	0 ms	0 ms	0 ms
Scheduler delay	63 ms	63 ms	89 ms	0.1 s	0.1 s
Shuffle Write	165.0 B	165.0 B	330.0 B	330.0 B	330.0 B

Aggregated Metrics by Executor

Executor ID	Address	Task Time	Total Tasks	Failed Tasks	Succeeded Tasks	Input	Shuffle Read	Shuffle Write	Shuffle Spill (Memory)	Shuffle Spill (Disk)
0	hadoop1:52855	0.1 s	1	0	1	0.0 B	0.0 B	330.0 B	0.0 B	0.0 B
1	hadoop2:56954	0.1 s	1	0	1	0.0 B	0.0 B	330.0 B	0.0 B	0.0 B
2	hadoop3:54866	0.1 s	1	0	1	0.0 B	0.0 B	165.0 B	0.0 B	0.0 B

Tasks

Index	ID	Attempt	Status	Locality Level	Executor	Launch Time	Duration	GC Time	Accumulators	Write Time	Shuffle Write	Errors
0	78	0	SUCCESS	PROCESS_LOCAL	hadoop3	2015/07/13 22:44:15	19 ms				165.0 B	
1	79	0	SUCCESS	PROCESS_LOCAL	hadoop2	2015/07/13 22:44:15	19 ms				330.0 B	
2	80	0	SUCCESS	PROCESS_LOCAL	hadoop1	2015/07/13 22:44:15	39 ms				330.0 B	

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`。

```
/**
 * Read a text file from HDFS, a local file system (available on all nodes), or any
 * Hadoop-supported file system URI, and return it as an RDD of Strings.
 */
def textFile(path: String, minPartitions: Int = defaultMinPartitions): RDD[String] = {
  hadoopFile(path, classOf[TextInputFormat], classOf[LongWritable], classOf[Text],
    minPartitions).map(pair => pair._2.toString)
}
```

```
scala> val text = sc.textFile("hdfs://hadoop1:9000/class3/directory/")
15/07/14 11:12:04 INFO MemoryStore: ensureFreespace(177238) called with curMem=390338, maxMem=556038881
15/07/14 11:12:04 INFO MemoryStore: Block broadcast_5 stored as values in memory (estimated size 173.1 KB, free 529.7 MB)
15/07/14 11:12:04 INFO MemoryStore: ensureFreespace(14082) called with curMem=567576, maxMem=556038881
15/07/14 11:12:04 INFO MemoryStore: Block broadcast_5_piece0 stored as bytes in memory (estimated size 13.8 KB, free 529.7 MB)
15/07/14 11:12:04 INFO BlockManagerInfo: Added broadcast_5_piece0 in memory on hadoop1:38368 (size: 13.8 KB, free: 530.2 MB)
15/07/14 11:12:04 INFO BlockManagerMaster: Updated info of block broadcast_5_piece0
text: org.apache.spark.rdd.RDD[String] = hdfs://hadoop1:9000/class3/directory/ MappedRDD[12] at textFile at <console>:12

scala> text.toDebugString
15/07/14 11:12:15 INFO FileInputFormat: Total input paths to process : 2
res4: String =
(2) hdfs://hadoop1:9000/class3/directory/ MappedRDD[12] at textFile at <console>:12
| hdfs://hadoop1:9000/class3/directory/ HadoopRDD[11] at textFile at <console>:12
```

- 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))` 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。

```
scala> val words=text.flatMap(_.split(" "))
words: org.apache.spark.rdd.RDD[String] = FlatMappedRDD[13] at flatMap at <console>:14

scala> val wordscount=words.map(x=>(x,1)).reduceByKey(_+_ )
wordscount: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[15] at reduceByKey at <console>:16

scala> wordscount.toDebugString
res5: String =
(2) ShuffledRDD[15] at reduceByKey at <console>:16
+- (2) MappedRDD[14] at map at <console>:16
| FlatMappedRDD[13] at flatMap at <console>:14
| | hdfs://hadoop1:9000/class3/directory/ MappedRDD[12] at textFile at <console>:12
| | hdfs://hadoop1:9000/class3/directory/ HadoopRDD[11] at textFile at <console>:12
```

```
scala> wordscount.collect
15/07/14 11:14:20 INFO SparkContext: Starting job: collect at <console>:19
15/07/14 11:14:20 INFO DAGScheduler: Registering RDD 14 (map at <console>:16)
15/07/14 11:14:20 INFO DAGScheduler: Got job 1 (collect at <console>:19) with 2 output partitions (allowLocal=false)
15/07/14 11:14:20 INFO DAGScheduler: Final stage: stage 2(collect at <console>:19)
15/07/14 11:14:20 INFO DAGScheduler: Parents of final stage: List(stage 3)
15/07/14 11:14:20 INFO DAGScheduler: Missing parents: List(stage 3)
15/07/14 11:14:20 INFO DAGScheduler: Submitting stage 3 (MappedRDD[14] at map at <console>:16), which has no missing parents
15/07/14 11:14:20 INFO MemoryStore: ensureFreespace(3440) called with curMem=581658, maxMem=556038881
15/07/14 11:14:20 INFO MemoryStore: Block broadcast_6 stored as values in memory (estimated size 3.4 KB, free 529.7 MB)
15/07/14 11:14:21 INFO MemoryStore: ensureFreespace(2069) called with curMem=585098, maxMem=556038881
```

```
15/07/14 11:14:27 INFO MapOutputTrackerMaster: Size of output statuses for shuffle 1 is 144 bytes
15/07/14 11:14:27 INFO BlockManagerInfo: Added broadcast_7_piece0 in memory on hadoop3:48821 (size: 1323.0 B, free: 530.3 MB)
15/07/14 11:14:27 INFO MapOutputTrackerMasterActor: Asked to send map output locations for shuffle 1 to sparkExecutor@hadoop3:3621
1
15/07/14 11:14:27 INFO TaskSetManager: Finished task 0.0 in stage 2.0 (TID 6) in 243 ms on hadoop2 (1/2)
15/07/14 11:14:27 INFO TaskSetManager: Finished task 1.0 in stage 2.0 (TID 7) in 398 ms on hadoop3 (2/2)
15/07/14 11:14:27 INFO TaskSchedulerImpl: Removed TaskSet 2.0, whose tasks have all completed, from pool
15/07/14 11:14:27 INFO DAGScheduler: Stage 2 (collect at <console>:19) finished in 0.401 s
15/07/14 11:14:27 INFO SparkContext: Job finished: collect at <console>:19, took 6.63604523 s
res6: Array[(String, Int)] = Array((d,2), (4,2), (b,4), (2,4), (e,1), (5,1), (a,5), (3,3), (1,5), (c,3))
```

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
```

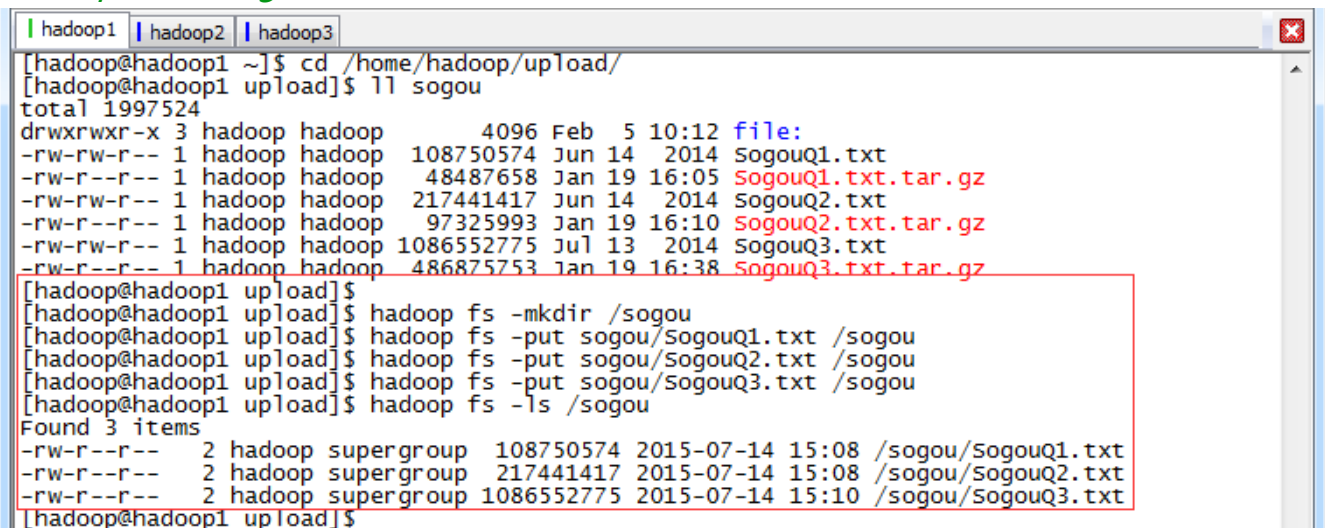
3.3.4 搜狗日志查询例子演示

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数据\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
```



```
[hadoop@hadoop1 ~]$ cd /home/hadoop/upload/  
[hadoop@hadoop1 upload]$ ll sogou  
total 1997524  
drwxrwxr-x 3 hadoop hadoop 4096 Feb  5 10:12 file:  
-rw-rw-r-- 1 hadoop hadoop 108750574 Jun 14  2014 SogouQ1.txt  
-rw-r--r-- 1 hadoop hadoop 48487658 Jan 19 16:05 SogouQ1.txt.tar.gz  
-rw-rw-r-- 1 hadoop hadoop 217441417 Jun 14  2014 SogouQ2.txt  
-rw-r--r-- 1 hadoop hadoop 97325993 Jan 19 16:10 SogouQ2.txt.tar.gz  
-rw-rw-r-- 1 hadoop hadoop 1086552775 Jul 13  2014 SogouQ3.txt  
-rw-r--r-- 1 hadoop hadoop 486875753 Jan 19 16:38 SogouQ3.txt.tar.gz  
[hadoop@hadoop1 upload]$  
[hadoop@hadoop1 upload]$ hadoop fs -mkdir /sogou  
[hadoop@hadoop1 upload]$ hadoop fs -put sogou/SogouQ1.txt /sogou  
[hadoop@hadoop1 upload]$ hadoop fs -put sogou/SogouQ2.txt /sogou  
[hadoop@hadoop1 upload]$ hadoop fs -put sogou/SogouQ3.txt /sogou  
[hadoop@hadoop1 upload]$ hadoop fs -ls /sogou  
Found 3 items  
-rw-r--r--  2 hadoop supergroup 108750574 2015-07-14 15:08 /sogou/SogouQ1.txt  
-rw-r--r--  2 hadoop supergroup 217441417 2015-07-14 15:08 /sogou/SogouQ2.txt  
-rw-r--r--  2 hadoop supergroup 1086552775 2015-07-14 15:10 /sogou/SogouQ3.txt  
[hadoop@hadoop1 upload]$
```

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 `MapppedRDD`；
- 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 `MapppedRDD`，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`；

```
scala> val rdd1 = sc.textFile("hdfs://hadoop1:9000/sogou/SogouQ1.txt")
15/07/14 15:16:55 INFO MemoryStore: ensureFreeSpace(138675) called with curMem=0, maxMem=556038881
15/07/14 15:16:55 INFO MemoryStore: Block broadcast_0 stored as values in memory (estimated size 135.4 KB, free 530.1 MB)
15/07/14 15:16:55 INFO MemoryStore: ensureFreeSpace(10090) called with curMem=138675, maxMem=556038881
15/07/14 15:16:55 INFO MemoryStore: Block broadcast_0_piece0 stored as bytes in memory (estimated size 9.9 KB, free 530.1 MB)
15/07/14 15:16:55 INFO BlockManagerInfo: Added broadcast_0_piece0 in memory on hadoop1:47093 (size: 9.9 KB, free: 530.3 MB)
15/07/14 15:16:55 INFO BlockManagerMaster: updated info of block broadcast_0_piece0
rdd1: org.apache.spark.rdd.RDD[String] = hdfs://hadoop1:9000/sogou/SogouQ1.txt MappedRDD[1] at textFile at <console>:12

scala> val rdd2=rdd1.map(_._split("\t")).filter(_._length==6)
rdd2: org.apache.spark.rdd.RDD[Array[String]] = FilteredRDD[3] at filter at <console>:14
```

- 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；

```
scala> rdd2.count()
15/07/14 15:18:57 INFO FileInputFormat: Total input paths to process : 1
15/07/14 15:18:57 INFO SparkContext: Starting job: count at <console>:17
15/07/14 15:18:57 INFO DAGScheduler: Got job 0 (count at <console>:17) with 2 output partitions (allowLocal=false)
15/07/14 15:18:57 INFO DAGScheduler: Final stage: stage 0(count at <console>:17)
15/07/14 15:18:57 INFO DAGScheduler: Parents of final stage: List()
15/07/14 15:18:57 INFO DAGScheduler: Missing parents: List()
15/07/14 15:18:57 INFO DAGScheduler: Submitting stage 0 (FilteredRDD[3] at filter at <console>:14), which has no missing paren
15/07/14 15:18:57 INFO MemoryStore: ensureFreeSpace(2760) called with curMem=148765, maxMem=556038881
15/07/14 15:18:57 INFO MemoryStore: Block broadcast_1 stored as values in memory (estimated size 2.7 KB, free 530.1 MB)
15/07/14 15:18:57 INFO MemoryStore: ensureFreeSpace(1710) called with curMem=151525, maxMem=556038881
15/07/14 15:18:57 INFO MemoryStore: Block broadcast_1_piece0 stored as bytes in memory (estimated size 1710.0 B, free 530.1 MB)
15/07/14 15:18:57 INFO BlockManagerInfo: Added broadcast_1_piece0 in memory on hadoop1:47093 (size: 1710.0 B, free: 530.3 MB)
15/07/14 15:18:57 INFO BlockManagerMaster: updated info of block broadcast_1_piece0
15/07/14 15:18:57 INFO DAGScheduler: Submitting 2 missing tasks from stage 0 (FilteredRDD[3] at filter at <console>:14)
15/07/14 15:18:57 INFO TaskSchedulerImpl: Adding task set 0.0 with 2 tasks
15/07/14 15:18:57 INFO TaskSetManager: Starting task 0.0 in stage 0.0 (TID 0, hadoop1, NODE_LOCAL, 1193 bytes)
15/07/14 15:18:57 INFO TaskSetManager: Starting task 1.0 in stage 0.0 (TID 1, hadoop2, NODE_LOCAL, 1193 bytes)
15/07/14 15:18:58 INFO ConnectionManager: Accepted connection from [hadoop1/192.168.10.111:44207]
15/07/14 15:18:58 INFO SendingConnection: Initiating connection to [hadoop1/192.168.10.111:55576]
15/07/14 15:18:58 INFO SendingConnection: Connected to [hadoop1/192.168.10.111:55576], 1 messages pending
15/07/14 15:18:58 INFO BlockManagerInfo: Added broadcast_1_piece0 in memory on hadoop1:55576 (size: 1710.0 B, free: 530.3 MB)
15/07/14 15:18:58 INFO ConnectionManager: Accepted connection from [hadoop2/192.168.10.112:37473]
15/07/14 15:18:58 INFO SendingConnection: Initiating connection to [hadoop2/192.168.10.112:35895]
15/07/14 15:18:58 INFO SendingConnection: Connected to [hadoop2/192.168.10.112:35895], 1 messages pending
15/07/14 15:18:58 INFO BlockManagerInfo: Added broadcast_1_piece0 in memory on hadoop2:35895 (size: 1710.0 B, free: 530.3 MB)
15/07/14 15:18:58 INFO BlockManagerInfo: Added broadcast_0_piece0 in memory on hadoop1:55576 (size: 9.9 KB, free: 530.3 MB)
15/07/14 15:18:59 INFO BlockManagerInfo: Added broadcast_0_piece0 in memory on hadoop2:35895 (size: 9.9 KB, free: 530.3 MB)
15/07/14 15:19:04 INFO TaskSetManager: Finished task 0.0 in stage 0.0 (TID 0) in 6972 ms on hadoop1 (1/2)
15/07/14 15:19:06 INFO TaskSetManager: Finished task 1.0 in stage 0.0 (TID 1) in 8464 ms on hadoop2 (2/2)
15/07/14 15:19:06 INFO TaskSchedulerImpl: Removed TaskSet 0.0, whose tasks have all completed, from pool
15/07/14 15:19:06 INFO DAGScheduler: stage 0 (count at <console>:17) finished in 8.484 s
15/07/14 15:19:06 INFO SparkContext: Job finished: count at <console>:17, took 8.619224474 s
res0: Long = 1000000
```

- `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；

```
scala> val rdd3=rdd2.filter(_(3).toInt==1).filter(_(4).toInt==2)
rdd3: org.apache.spark.rdd.RDD[Array[String]] = FilteredRDD[7] at filter at <console>:16

scala> rdd3.count()
15/07/14 15:21:06 INFO SparkContext: Starting job: count at <console>:19
15/07/14 15:21:06 INFO DAGScheduler: Got job 2 (count at <console>:19) with 2 output partitions (allowLocal=false)
15/07/14 15:21:06 INFO DAGScheduler: Final stage: stage 2(count at <console>:19)
15/07/14 15:21:06 INFO DAGScheduler: Parents of final stage: List()
15/07/14 15:21:06 INFO DAGScheduler: Missing parents: List()
15/07/14 15:21:06 INFO DAGScheduler: Submitting stage 2 (FilteredRDD[7] at filter at <console>:16), which has no missing parents
15/07/14 15:21:06 INFO MemoryStore: ensureFreeSpace(2992) called with curMem=157987, maxMem=556038881
15/07/14 15:21:06 INFO MemoryStore: Block broadcast_3 stored as values in memory (estimated size 2.9 KB, free 530.1 MB)
15/07/14 15:21:06 INFO MemoryStore: ensureFreeSpace(1761) called with curMem=160979, maxMem=556038881
15/07/14 15:21:06 INFO MemoryStore: Block broadcast_3_piece0 stored as bytes in memory (estimated size 1761.0 B, free 530.1 MB)
15/07/14 15:21:06 INFO BlockManagerInfo: Added broadcast_3_piece0 in memory on hadoop1:47093 (size: 1761.0 B, free: 530.3 MB)
15/07/14 15:21:06 INFO BlockManagerMaster: updated info of block broadcast_3_piece0
15/07/14 15:21:06 INFO DAGScheduler: Submitting 2 missing tasks from stage 2 (FilteredRDD[7] at filter at <console>:16)
15/07/14 15:21:06 INFO TaskSchedulerImpl: Adding task set 2.0 with 2 tasks
15/07/14 15:21:06 INFO TaskSetManager: Starting task 0.0 in stage 2.0 (TID 4, hadoop1, NODE_LOCAL, 1193 bytes)
15/07/14 15:21:06 INFO TaskSetManager: Starting task 1.0 in stage 2.0 (TID 5, hadoop2, NODE_LOCAL, 1193 bytes)
15/07/14 15:21:06 INFO BlockManagerInfo: Added broadcast_3_piece0 in memory on hadoop1:55576 (size: 1761.0 B, free: 530.3 MB)
15/07/14 15:21:06 INFO BlockManagerInfo: Added broadcast_3_piece0 in memory on hadoop2:35895 (size: 1761.0 B, free: 530.3 MB)
15/07/14 15:21:08 INFO TaskSetManager: Finished task 0.0 in stage 2.0 (TID 4) in 2304 ms on hadoop1 (1/2)
15/07/14 15:21:08 INFO TaskSetManager: Finished task 1.0 in stage 2.0 (TID 5) in 2463 ms on hadoop2 (2/2)
15/07/14 15:21:08 INFO DAGScheduler: stage 2 (count at <console>:19) finished in 2.465 s
15/07/14 15:21:08 INFO TaskSchedulerImpl: Removed TaskSet 2.0, whose tasks have all completed, from pool
15/07/14 15:21:08 INFO SparkContext: Job finished: count at <console>:19, took 2.488170312 s
res2: Long = 19771
```

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

```
scala> rdd3.toDebugString
res3: String =
(2) FilteredRDD[7] at filter at <console>:16
| FilteredRDD[6] at filter at <console>:16
| FilteredRDD[3] at filter at <console>:14
| MappedRDD[2] at map at <console>:14
| hdfs://hadoop1:9000/sogou/SogouQ1.txt MappedRDD[1] at textFile at <console>:12
| hdfs://hadoop1:9000/sogou/SogouQ1.txt HadoopRDD[0] at textFile at <console>:12
```

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")
```

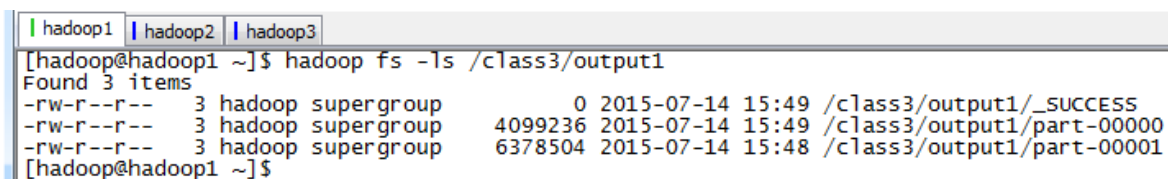
该命令运行的过程如下：

- 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：

```
scala> rdd4.toDebugString
res5: String =
(2) MappedRDD[14] at map at <console>:16
| ShuffledRDD[13] at sortByKey at <console>:16
+- (2) MappedRDD[10] at map at <console>:16
| ShuffledRDD[9] at reduceByKey at <console>:16
+- (2) MappedRDD[8] at map at <console>:16
| FilteredRDD[3] at filter at <console>:14
| MappedRDD[2] at map at <console>:14
| hdfs://hadoop1:9000/sogou/SogouQ1.txt MappedRDD[1] at textFile at <console>:12
| hdfs://hadoop1:9000/sogou/SogouQ1.txt HadoopRDD[0] at textFile at <console>:12
```

- 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
```



```
[hadoop@hadoop1 ~]$ hadoop fs -ls /class3/output1
Found 3 items
-rw-r--r--  3 hadoop supergroup          0 2015-07-14 15:49 /class3/output1/_SUCCESS
-rw-r--r--  3 hadoop supergroup  4099236 2015-07-14 15:49 /class3/output1/part-00000
-rw-r--r--  3 hadoop supergroup  6378504 2015-07-14 15:48 /class3/output1/part-00001
[hadoop@hadoop1 ~]$
```

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
```

```
[hadoop@hadoop1 upload]$ head result
(b3c94c37fb154d46c30a360c7941ff7e,676)
(cc7063efc64510c20bcdd604e12a3b26,613)
(955c6390c02797b3558ba223b8201915,391)
(b1e371de5729cdda9270b7ad09484c4f,337)
(6056710d9eafa569ddc800fe24643051,277)
(637b29b47fed3853e117aa7009a4b621,266)
(c9f4ff7790d0615f6f66b410673e3124,231)
(dca9034de17f6c34cfd56db13ce39f1c,226)
(82e53ddb484e632437039048c5901608,221)
(c72ce1164bcd263ba1f69292abdfdf7c,214)
```