Hadoop Map/Reduce教程

目录

1. 目的

这篇教程从用户的角度出发，全面地介绍了Hadoop Map/Reduce框架的各个方面。

2. 先决条件

请先确认Hadoop被正确安装、配置和正常运行中。更多信息见：

- Hadoop快速入门对初次使用者。
- Hadoop集群搭建对大规模分布式集群。

3. 概述

Hadoop Map/Reduce是一个使用简易的软件框架，基于它写出来的应用程序能够运行在由上千个商用机器组成的大型集群上，并以一种可靠容错的方式并行处理上T级别的数据集。

一个Map/Reduce 作业（job） 通常会把输入的数据集切分为若干独立的数据块，由map任务（task）以完全并行的方式处理它们。框架会对map的输出先进行排序， 然后把结果输入给reduce任务。通常作业的输入和输出都会被存储在文件系统中。 整个框架负责任务的调度和监控，以及重新执行已经失败的任务。

通常，Map/Reduce框架和分布式文件系统是运行在一组相同的节点上的，也就是说，计算节点和存储节点通常在一起。这种配置允许框架在那些已经存好数据的节点上高效地调度任务，这可以使整个集群的网络带宽被非常高效地利用。

Map/Reduce框架由一个单独的master JobTracker 和每个集群节点一个slave TaskTracker共同组成。master负责调度构成一个作业的所有任务，这些任务分布在不同的slave上，master监控它们的执行，重新执行已经失败的任务。而slave仅负责执行由master指派的任务。

应用程序至少应该指明输入/输出的位置（路径），并通过实现合适的接口或抽象类提供map和reduce函数。再加上其他作业的参数，就构成了作业配置（job configuration）。然后，Hadoop的 job client提交作业（jar包/可执行程序等）和配置信息给JobTracker，后者负责分发这些软件和配置信息给slave、调度任务并监控它们的执行，同时提供状态和诊断信息给job-client。

虽然Hadoop框架是用JavaTM实现的，但Map/Reduce应用程序则不一定要用 Java来写。

- Hadoop Streaming是一种运行作业的实用工具，它允许用户创建和运行任何可执行程序 （例如：Shell工具）来做为mapper和reducer。
- Hadoop Pipes是一个与SWIG兼容的C++ API （没有基于JNITM技术），它也可用于实现Map/Reduce应用程序。

4．输入与输出

Map/Reduce框架运转在<key，value> 键值对上，也就是说， 框架把作业的输入看为是一组<key，value> 键值对，同样也产出一组 <key，value> 键值对做为作业的输出，这两组键值对的类型可能不同。

框架需要对key和value的类(classes)进行序列化操作， 因此，这些类需要实现Writable接口。 另外，为了方便框架执行排序操作，key类必须实现WritableComparable接口。

一个Map/Reduce 作业的输入和输出类型如下所示：

(input) <k1，v1> -> map -> <k2，v2> -> combine -> <k2，v2> -> reduce -> <k3，v3> (output)

5．例子：WordCount v1.0

在深入细节之前，让我们先看一个Map/Reduce的应用示例，以便对它们的工作方式有一个初步的认识。

WordCount是一个简单的应用，它可以计算出指定数据集中每一个单词出现的次数。

这个应用适用于 单机模式， 伪分布式模式 或 完全分布式模式 三种Hadoop安装方式。

5.1．源代码

	WordCount.java
1.	package org.myorg;
2.	
3.	import java.io.IOException;
4.	import java.util.*;

5.	
6.	import org.apache.hadoop.fs.Path;
7.	import org.apache.hadoop.conf.*;
8.	import org.apache.hadoop.io.*;
9.	import org.apache.hadoop.mapred.*;
10.	import org.apache.hadoop.util.*;
11.	
12.	public class WordCount {
13.	
14.	public static class Map extends MapReduceBase implements Mapper<LongWritable, Text, Text, IntWritable> {
15.	private final static IntWritable one = new IntWritable(1);
16.	private Text word = new Text();
17.	
18.	public void map(LongWritable key, Text value, OutputCollector<Text, IntWritable> output, Reporter reporter) throws IOException {
19.	String line = value.toString();
20.	StringTokenizer tokenizer = new StringTokenizer(line);
21.	while (tokenizer.hasMoreTokens()) {
22.	word.set(tokenizer.nextToken());
23.	output.collect(word, one);
24.	}

25.	}
26.	}
27.	
28.	public static class Reduce extends MapReduceBase implements Reducer<Text, IntWritable, Text, IntWritable> {
29.	public void reduce(Text key, Iterator<IntWritable> values, OutputCollector<Text, IntWritable> output, Reporter reporter) throws IOException {
30.	int sum = 0;
31.	while (values.hasNext()) {
32.	sum += values.next().get();
33.	}
34.	output.collect(key, new IntWritable(sum));
35.	}
36.	}
37.	
38.	public static void main(String[] args) throws Exception {
39.	JobConf conf = new JobConf(WordCount.class);
40.	conf.setJobName("wordcount");
41.	
42.	conf.setOutputKeyClass(Text.class);
43.	conf.setOutputValueClass(IntWritable.class);
44.	

45.	conf.setMapperClass(Map.class);
46.	conf.setCombinerClass(Reduce.class);
47.	conf.setReducerClass(Reduce.class);
48.	
49.	conf.setInputFormat(TextInputFormat.class);
50.	conf.setOutputFormat(TextOutputFormat.class);
51.	
52.	FileInputFormat.setInputPaths(conf, new Path(args[0]));
53.	FileOutputFormat.setOutputPath(conf, new Path(args[1]));
54.	
55.	JobClient.runJob(conf);
57.	}
58.	}
59.	

5.2. 用法

假设环境变量HADOOP_HOME对应安装时的根目录, HADOOP_VERSION对应Hadoop的当前安装版本, 编译WordCount.java来创建jar包, 可如下操作:

```
$ mkdir wordcount_classes
$ javac -classpath ${HADOOP_HOME}/hadoop-${HADOOP_VERSION}-core.jar -d
wordcount_classes WordCount.java
$ jar -cvf /usr/joe/wordcount.jar -C wordcount_classes/ .
```

假设:

- /usr/joe/wordcount/input - 是HDFS中的输入路径

- /usr/joe/wordcount/output - 是HDFS中的输出路径

用示例文本文件做为输入:

```
$ bin/hadoop dfs -ls /usr/joe/wordcount/input/
/usr/joe/wordcount/input/file01
/usr/joe/wordcount/input/file02
$ bin/hadoop dfs -cat /usr/joe/wordcount/input/file01
Hello World Bye World
$ bin/hadoop dfs -cat /usr/joe/wordcount/input/file02
Hello Hadoop Goodbye Hadoop
```

运行应用程序:

```
$ bin/hadoop jar /usr/joe/wordcount.jar org.myorg.WordCount
/usr/joe/wordcount/input /usr/joe/wordcount/output
```

输出是:

```
$ bin/hadoop dfs -cat /usr/joe/wordcount/output/part-00000
Bye 1
Goodbye 1
Hadoop 2
Hello 2
World 2
```

应用程序能够使用-files选项来指定一个由逗号分隔的路径列表，这些路径是task的当前工作目录。使用选项-libjars可以向map和reduce的classpath中添加jar包。使用-archives选项程序可以传递档案文件做为参数，这些档案文件会被解压并且在task的当前工作目录下会创建一个指向解压生成的目录的符号链接（以压缩包的名字命名）。 有关命令行选项的更多细节请参考 Commands manual。

使用-libjars和-files运行wordcount例子:

```
hadoop jar hadoop-examples.jar wordcount -files cachefile.txt -libjars
mylib.jar input output
```

5.3. 解释

WordCount应用程序非常直截了当。

Mapper(14-26行)中的map方法(18-25行)通过指定的 TextInputFormat(49行)一次处理一行。然后，它通过StringTokenizer 以空格为分隔符将一行切分为若干tokens，之后，输出< <word>，1> 形式的键值对。

对于示例中的第一个输入，map输出是:
< Hello，1>
< World，1>
< Bye，1>
< World，1>

第二个输入，map输出是:
< Hello，1>
< Hadoop，1>
< Goodbye，1>
< Hadoop，1>

关于组成一个指定作业的map数目的确定，以及如何以更精细的方式去控制这些map，我们将在教程的后续部分学习到更多的内容。

WordCount还指定了一个combiner (46行)。因此，每次map运行之后，会对输出按照key进行排序，然后把输出传递给本地的combiner（按照作业的配置与Reducer一样），进行本地聚合。

第一个map的输出是:
< Bye，1>
< Hello，1>
< World，2>

第二个map的输出是:
< Goodbye，1>
< Hadoop，2>
< Hello，1>

Reducer(28-36行)中的reduce方法(29-35行) 仅是将每个key（本例中就是单词）出现的次数求和。

因此这个作业的输出就是:
< Bye，1>
< Goodbye，1>

< Hadoop, 2>
< Hello, 2>
< World, 2>

代码中的run方法中指定了作业的几个方面， 例如：通过命令行传递过来的输入/输出路径、key/value的类型、输入/输出的格式等等JobConf中的配置信息。随后程序调用了JobClient.runJob(55行)来提交作业并且监控它的执行。

我们将在本教程的后续部分学习更多的关于JobConf， JobClient， Tool和其他接口及类(class)。

6. Map/Reduce - 用户界面

这部分文档为用户将会面临的Map/Reduce框架中的各个环节提供了适当的细节。这应该会帮助用户更细粒度地去实现、配置和调优作业。然而，请注意每个类/接口的javadoc文档提供最全面的文档；本文只是想起到指南的作用。

我们会先看看Mapper和Reducer接口。应用程序通常会通过提供map和reduce方法来实现它们。

然后，我们会讨论其他的核心接口，其中包括： JobConf，JobClient，Partitioner， OutputCollector，Reporter， InputFormat，OutputFormat等等。

最后，我们将通过讨论框架中一些有用的功能点（例如：DistributedCache，IsolationRunner等等）来收尾。

6.1. 核心功能描述

应用程序通常会通过提供map和reduce来实现 Mapper和Reducer接口，它们组成作业的核心。

6.1.1. Mapper

Mapper将输入键值对(key/value pair)映射到一组中间格式的键值对集合。

Map是一类将输入记录集转换为中间格式记录集的独立任务。 这种转换的中间格式记录集不需要与输入记录集的类型一致。一个给定的输入键值对可以映射成0个或多个输出键值对。

Hadoop Map/Reduce框架为每一个InputSplit产生一个map任务，而每个InputSplit是

由该作业的InputFormat产生的。

概括地说，对Mapper的实现者需要重写 JobConfigurable.configure(JobConf)方法，这个方法需要传递一个JobConf参数，目的是完成Mapper的初始化工作。然后，框架为这个任务的InputSplit中每个键值对调用一次 map(WritableComparable, Writable, OutputCollector, Reporter)操作。应用程序可以通过重写Closeable.close()方法来执行相应的清理工作。

输出键值对不需要与输入键值对的类型一致。一个给定的输入键值对可以映射成0个或多个输出键值对。通过调用 OutputCollector.collect(WritableComparable,Writable)可以收集输出的键值对。

应用程序可以使用Reporter报告进度，设定应用级别的状态消息，更新Counters（计数器），或者仅是表明自己运行正常。

框架随后会把与一个特定key关联的所有中间过程的值（value）分成组，然后把它们传给Reducer以产出最终的结果。用户可以通过 JobConf.setOutputKeyComparatorClass(Class)来指定具体负责分组的 Comparator。

Mapper的输出被排序后，就被划分给每个Reducer。分块的总数目和一个作业的reduce任务的数目是一样的。用户可以通过实现自定义的 Partitioner来控制哪个key被分配给哪个 Reducer。

用户可选择通过 JobConf.setCombinerClass(Class)指定一个combiner，它负责对中间过程的输出进行本地的聚集，这会有助于降低从Mapper到 Reducer数据传输量。

这些被排好序的中间过程的输出结果保存的格式是(key-len, key, value-len, value)，应用程序可以通过JobConf控制对这些中间结果是否进行压缩以及怎么压缩，使用哪种 CompressionCodec。

6.1.1.1. 需要多少个Map?

Map的数目通常是由输入数据的大小决定的，一般就是所有输入文件的总块（block）数。

Map正常的并行规模大致是每个节点（node）大约10到100个map，对于CPU 消耗较小的map任务可以设到300个左右。由于每个任务初始化需要一定的时间，因此，比较合理的情况是map执行的时间至少超过1分钟。

这样，如果你输入10TB的数据，每个块（block）的大小是128MB，你将需要大约

82,000个map来完成任务，除非使用 setNumMapTasks(int)（注意：这里仅仅是对框架进行了一个提示(hint)，实际决定因素见这里）将这个数值设置得更高。

6.1.2. Reducer

Reducer将与一个key关联的一组中间数值集归约（reduce）为一个更小的数值集。

用户可以通过 JobConf.setNumReduceTasks(int)设定一个作业中reduce任务的数目。

概括地说，对Reducer的实现者需要重写 JobConfigurable.configure(JobConf)方法，这个方法需要传递一个JobConf参数，目的是完成Reducer的初始化工作。然后，框架为成组的输入数据中的每个<key，(list of values)>对调用一次 reduce(WritableComparable, Iterator, OutputCollector, Reporter)方法。之后，应用程序可以通过重写Closeable.close()来执行相应的清理工作。

Reducer有3个主要阶段：shuffle、sort和reduce。

6.1.2.1. Shuffle

Reducer的输入就是Mapper已经排好序的输出。在这个阶段，框架通过HTTP为每个Reducer获得所有Mapper输出中与之相关的分块。

6.1.2.2. Sort

这个阶段，框架将按照key的值对Reducer的输入进行分组 （因为不同mapper的输出中可能会有相同的key）。

Shuffle和Sort两个阶段是同时进行的；map的输出也是一边被取回一边被合并的。

Secondary Sort

如果需要中间过程对key的分组规则和reduce前对key的分组规则不同，那么可以通过 JobConf.setOutputValueGroupingComparator(Class)来指定一个Comparator。再加上 JobConf.setOutputKeyComparatorClass(Class)可用于控制中间过程的key如何被分组，所以结合两者可以实现按值的二次排序。

6.1.2.3. Reduce

在这个阶段，框架为已分组的输入数据中的每个 <key，(list of values)>对调用一次 reduce(WritableComparable, Iterator, OutputCollector, Reporter)方法。

Reduce任务的输出通常是通过调用 OutputCollector.collect(WritableComparable, Writable)写入 文件系统的。

应用程序可以使用Reporter报告进度，设定应用程序级别的状态消息，更新Counters（计数器），或者仅是表明自己运行正常。

Reducer的输出是没有排序的。

6.1.2.4. 需要多少个Reduce?

Reduce的数目建议是0.95或1.75乘以（<no. of nodes> * mapred.tasktracker.reduce.tasks.maximum）。

用0.95，所有reduce可以在maps一完成时就立刻启动，开始传输map的输出结果。用1.75，速度快的节点可以在完成第一轮reduce任务后，可以开始第二轮，这样可以得到比较好的负载均衡的效果。

增加reduce的数目会增加整个框架的开销，但可以改善负载均衡，降低由于执行失败带来的负面影响。

上述比例因子比整体数目稍小一些是为了给框架中的推测性任务（speculative-tasks）或失败的任务预留一些reduce的资源。

6.1.2.5. 无Reducer

如果没有归约要进行，那么设置reduce任务的数目为零是合法的。

这种情况下，map任务的输出会直接被写入由 setOutputPath(Path)指定的输出路径。框架在把它们写入FileSystem之前没有对它们进行排序。

6.1.3. Partitioner

Partitioner用于划分键值空间（key space）。

Partitioner负责控制map输出结果key的分割。Key（或者一个key子集）被用于产生分区，通常使用的是Hash函数。分区的数目与一个作业的reduce任务的数目是一样的。因此，它控制将中间过程的key（也就是这条记录）应该发送给m个reduce任务中的哪一个来进行reduce操作。

HashPartitioner是默认的 Partitioner。

6.1.4. Reporter

Reporter是用于Map/Reduce应用程序报告进度，设定应用级别的状态消息， 更新Counters（计数器）的机制。

Mapper和Reducer的实现可以利用Reporter 来报告进度，或者仅是表明自己运行正常。在那种应用程序需要花很长时间处理个别键值对的场景中，这种机制是很关键的，因为框架可能会以为这个任务超时了，从而将它强行杀死。另一个避免这种情况发生的方式是，将配置参数mapred.task.timeout设置为一个足够高的值（或者干脆设置为零，则没有超时限制了）。

应用程序可以用Reporter来更新Counter（计数器）。

6.1.5. OutputCollector

OutputCollector是一个Map/Reduce框架提供的用于收集 Mapper或Reducer输出数据的通用机制 （包括中间输出结果和作业的输出结果）。

Hadoop Map/Reduce框架附带了一个包含许多实用型的mapper、reducer和partitioner的类库。

6.2. 作业配置

JobConf代表一个Map/Reduce作业的配置。

JobConf是用户向Hadoop框架描述一个Map/Reduce作业如何执行的主要接口。框架会按照JobConf描述的信息忠实地去尝试完成这个作业，然而：

- 一些参数可能会被管理者标记为 final，这意味它们不能被更改。
- 一些作业的参数可以被直截了当地进行设置（例如： setNumReduceTasks(int)），而另一些参数则与框架或者作业的其他参数之间微妙地相互影响，并且设置起来比较复杂（例如： setNumMapTasks(int)）。

通常，JobConf会指明Mapper、Combiner(如果有的话)、 Partitioner、Reducer、InputFormat和 OutputFormat的具体实现。JobConf还能指定一组输入文件（setInputPaths(JobConf, Path...) /addInputPath(JobConf, Path)）和（setInputPaths(JobConf, String) /addInputPaths(JobConf, String)）以及输出文件应该写在哪儿（setOutputPath(Path)）。

JobConf可选择地对作业设置一些高级选项，例如：设置Comparator； 放到DistributedCache上的文件；中间结果或者作业输出结果是否需要压缩以及怎么压缩；利用用户提供的脚本(setMapDebugScript(String)/setReduceDebugScript(String)) 进行调试；作业是否允许预防性（speculative）任务的执行(setMapSpeculativeExecution(boolean))/(setReduceSpeculativeExecution(boolean))；每个任务最大的尝试次数（setMaxMapAttempts(int)/setMaxReduceAttempts(int)）；一个作业能容忍的任务失败的百分比(setMaxMapTaskFailuresPercent(int)/setMaxReduceTaskFailuresPercent(int)) ；等等。

当然，用户能使用 set(String, String)/get(String, String) 来设置或者取得应用程序需要的任意参数。然而，DistributedCache的使用是面向大规模只读数据的。

6.3. 任务的执行和环境

TaskTracker是在一个单独的jvm上以子进程的形式执行 Mapper/Reducer任务（Task）的。

子任务会继承父TaskTracker的环境。用户可以通过JobConf中的mapred.child.java.opts配置参数来设定子jvm上的附加选项，例如： 通过-Djava.library.path=<> 将一个非标准路径设为运行时的链接用以搜索共享库，等等。如果mapred.child.java.opts包含一个符号@taskid@， 它会被替换成map/reduce的taskid的值。

下面是一个包含多个参数和替换的例子，其中包括：记录jvm GC日志； JVM JMX代理程序以无密码的方式启动，这样它就能连接到jconsole上，从而可以查看子进程的内存和线程，得到线程的dump；还把子jvm的最大堆尺寸设置为512MB， 并为子jvm的java.library.path添加了一个附加路径。

```
<property>
  <name>mapred.child.java.opts</name>
  <value>
     -Xmx512M -Djava.library.path=/home/mycompany/lib -verbose:gc
-Xloggc:/tmp/@taskid@.gc
     -Dcom.sun.management.jmxremote.authenticate=false
-Dcom.sun.management.jmxremote.ssl=false
  </value>
```

</property>

用户或管理员也可以使用mapred.child.ulimit设定运行的子任务的最大虚拟内存。mapred.child.ulimit的值以（KB)为单位，并且必须大于或等于-Xmx参数传给JavaVM的值，否则VM会无法启动。

注意：mapred.child.java.opts只用于设置task tracker启动的子任务。为守护进程设置内存选项请查看 cluster_setup.html

${mapred.local.dir}/taskTracker/是task tracker的本地目录， 用于创建本地缓存和job。它可以指定多个目录（跨越多个磁盘），文件会半随机的保存到本地路径下的某个目录。当job启动时，task tracker根据配置文档创建本地job目录，目录结构如以下所示：

- ${mapred.local.dir}/taskTracker/archive/ :分布式缓存。这个目录保存本地的分布式缓存。因此本地分布式缓存是在所有task和job间共享的。
- ${mapred.local.dir}/taskTracker/jobcache/$jobid/ ： 本地job目录。
  - ${mapred.local.dir}/taskTracker/jobcache/$jobid/work/: job指定的共享目录。各个任务可以使用这个空间做为暂存空间，用于它们之间共享文件。这个目录通过job.local.dir 参数暴露给用户。这个路径可以通过API JobConf.getJobLocalDir()来访问。它也可以被做为系统属性获得。因此，用户（比如运行streaming）可以调用System.getProperty("job.local.dir")获得该目录。
  - ${mapred.local.dir}/taskTracker/jobcache/$jobid/jars/: 存放jar包的路径，用于存放作业的jar文件和展开的jar。job.jar是应用程序的jar文件，它会被自动分发到各台机器，在task启动前会被自动展开。使用api JobConf.getJar() 函数可以得到job.jar的位置。使用JobConf.getJar().getParent()可以访问存放展开的jar包的目录。
  - ${mapred.local.dir}/taskTracker/jobcache/$jobid/job.xml： 一个job.xml文件，本地的通用的作业配置文件。
  - ${mapred.local.dir}/taskTracker/jobcache/$jobid/$taskid： 每个任务有一个目录task-id，它里面有如下的目录结构：
    - ${mapred.local.dir}/taskTracker/jobcache/$jobid/$taskid/job.xml: 一个job.xml文件，本地化的任务作业配置文件。任务本地化是指为该task设定特定的属性值。这些值会在下面具体说明。
    - ${mapred.local.dir}/taskTracker/jobcache/$jobid/$taskid/output 一个存放中间过程的输出文件的目录。它保存了由framwork产生的临时map reduce数据，比如map的输出文件等。

- ${mapred.local.dir}/taskTracker/jobcache/$jobid/$taskid/work: task的当前工作目录。
- ${mapred.local.dir}/taskTracker/jobcache/$jobid/$taskid/work/tmp: task的临时目录。（用户可以设定属性mapred.child.tmp 来为map和reduce task设定临时目录。缺省值是./tmp。如果这个值不是绝对路径， 它会把task的工作路径加到该路径前面作为task的临时文件路径。如果这个值是绝对路径则直接使用这个值。 如果指定的目录不存在，会自动创建该目录。之后，按照选项 -Djava.io.tmpdir='临时文件的绝对路径'执行java子任务。 pipes和streaming的临时文件路径是通过环境变量TMPDIR='the absolute path of the tmp dir'设定的）。 如果mapred.child.tmp有./tmp值，这个目录会被创建。

下面的属性是为每个task执行时使用的本地参数，它们保存在本地化的任务作业配置文件里：

名称	类型	描述
mapred.job.id	String	job id
mapred.jar	String	job目录下job.jar的位置
job.local.dir	String	job指定的共享存储空间
mapred.tip.id	String	task id
mapred.task.id	String	task尝试id
mapred.task.is.map	boolean	是否是map task
mapred.task.partition	int	task在job中的id
map.input.file	String	map读取的文件名
map.input.start	long	map输入的数据块的起始位置偏移
map.input.length	long	map输入的数据块的字节数
mapred.work.output.dir	String	task临时输出目录

task的标准输出和错误输出流会被读到TaskTracker中，并且记录到${HADOOP_LOG_DIR}/userlogs

DistributedCache 可用于map或reduce task中分发jar包和本地库。子jvm总是把 当前工作目录 加到 java.library.path 和 LD_LIBRARY_PATH。 因此，可以通过 System.loadLibrary或 System.load装载缓存的库。有关使用分布式缓存加载共享库的细节请参考 native_libraries.html

6.4. 作业的提交与监控

JobClient是用户提交的作业与JobTracker交互的主要接口。

JobClient 提供提交作业，追踪进程，访问子任务的日志记录，获得Map/Reduce集群状态信息等功能。

作业提交过程包括:

1. 检查作业输入输出样式细节
2. 为作业计算InputSplit值。
3. 如果需要的话，为作业的DistributedCache建立必须的统计信息。
4. 拷贝作业的jar包和配置文件到FileSystem上的Map/Reduce系统目录下。
5. 提交作业到JobTracker并且监控它的状态。

作业的历史文件记录到指定目录的"_logs/history/"子目录下。这个指定目录由 hadoop.job.history.user.location设定，默认是作业输出的目录。因此默认情况下，文件会存放在mapred.output.dir/_logs/history目录下。用户可以设置 hadoop.job.history.user.location为none来停止日志记录。

用户使用下面的命令可以看到在指定目录下的历史日志记录的摘要。

```
$ bin/hadoop job -history output-dir
```

这个命令会打印出作业的细节，以及失败的和被杀死的任务细节。
要查看有关作业的更多细节例如成功的任务、每个任务尝试的次数（task attempt）等，可以使用下面的命令

```
$ bin/hadoop job -history all output-dir
```

用户可以使用 OutputLogFilter 从输出目录列表中筛选日志文件。

一般情况，用户利用JobConf创建应用程序并配置作业属性， 然后用 JobClient 提交作业并监视它的进程。

6.4.1. 作业的控制

有时候，用一个单独的Map/Reduce作业并不能完成一个复杂的任务，用户也许要链接

多个Map/Reduce作业才行。这是容易实现的，因为作业通常输出到分布式文件系统上的，所以可以把这个作业的输出作为下一个作业的输入实现串联。

然而，这也意味着，确保每一作业完成(成功或失败)的责任就直接落在了客户身上。在这种情况下，可以用的控制作业的选项有：

- runJob(JobConf)：提交作业，仅当作业完成时返回。
- submitJob(JobConf)：只提交作业，之后需要你轮询它返回的 RunningJob句柄的状态，并根据情况调度。
- JobConf.setJobEndNotificationURI(String)：设置一个作业完成通知，可避免轮询。

6.5. 作业的输入

InputFormat 为Map/Reduce作业描述输入的细节规范。

Map/Reduce框架根据作业的InputFormat来：

1. 检查作业输入的有效性。
2. 把输入文件切分成多个逻辑InputSplit实例， 并把每一实例分别分发给一个Mapper。
3. 提供RecordReader的实现，这个RecordReader从逻辑InputSplit中获得输入记录，这些记录将由Mapper处理。

基于文件的InputFormat实现（通常是 FileInputFormat的子类） 默认行为是按照输入文件的字节大小，把输入数据切分成逻辑分块（logical InputSplit ） 。 其中输入文件所在的FileSystem的数据块尺寸是分块大小的上限。下限可以设置 mapred.min.split.size 的值。

考虑到边界情况，对于很多应用程序来说，很明显按照文件大小进行逻辑分割是不能满足需求的。 在这种情况下，应用程序需要实现一个RecordReader来处理记录的边界并为每个任务提供一个逻辑分块的面向记录的视图。

TextInputFormat 是默认的InputFormat。

如果一个作业的Inputformat是TextInputFormat， 并且框架检测到输入文件的后缀是.gz和.lzo，就会使用对应的CompressionCodec自动解压缩这些文件。 但是需要注意，上述带后缀的压缩文件不会被切分，并且整个压缩文件会分给一个mapper来处理。

6.5.1. InputSplit

InputSplit 是一个单独的Mapper要处理的数据块。

一般的InputSplit 是字节样式输入，然后由RecordReader处理并转化成记录样式。

FileSplit 是默认的InputSplit。 它把 map.input.file 设定为输入文件的路径，输入文件是逻辑分块文件。

6.5.2. RecordReader

RecordReader 从InputSlit读入<key, value>对。

一般的，RecordReader 把由InputSplit 提供的字节样式的输入文件，转化成由Mapper处理的记录样式的文件。 因此RecordReader负责处理记录的边界情况和把数据表示成keys/values对形式。

6.6. 作业的输出

OutputFormat 描述Map/Reduce作业的输出样式。

Map/Reduce框架根据作业的OutputFormat来:

1. 检验作业的输出，例如检查输出路径是否已经存在。
2. 提供一个RecordWriter的实现，用来输出作业结果。 输出文件保存在FileSystem上。

TextOutputFormat是默认的 OutputFormat。

6.6.1. 任务的Side-Effect File

在一些应用程序中，子任务需要产生一些side-file，这些文件与作业实际输出结果的文件不同。

在这种情况下，同一个Mapper或者Reducer的两个实例（比如预防性任务）同时打开或者写 FileSystem上的同一文件就会产生冲突。因此应用程序在写文件的时候需要为每次任务尝试（不仅仅是每次任务，每个任务可以尝试执行很多次）选取一个独一无二的文件名(使用attemptid，例如task_200709221812_0001_m_000000_0)。

为了避免冲突，Map/Reduce框架为每次尝试执行任务都建立和维护一个特殊的${mapred.output.dir}/_temporary/_${taskid}子目录，这个目录位于本次尝试执行任务输出结果所在的FileSystem上，可以通过 ${mapred.work.output.dir}来访问这个子目录。 对于成功完成的任务尝试，只有

${mapred.output.dir}/_temporary/_${taskid}下的文件会移动到${mapred.output.dir}。当然，框架会丢弃那些失败的任务尝试的子目录。这种处理过程对于应用程序来说是完全透明的。

在任务执行期间，应用程序在写文件时可以利用这个特性，比如 通过FileOutputFormat.getWorkOutputPath()获得${mapred.work.output.dir}目录， 并在其下创建任意任务执行时所需的side-file，框架在任务尝试成功时会马上移动这些文件，因此不需要在程序内为每次任务尝试选取一个独一无二的名字。

注意：在每次任务尝试执行期间，${mapred.work.output.dir} 的值实际上是${mapred.output.dir}/_temporary/_{$taskid}，这个值是Map/Reduce框架创建的。所以使用这个特性的方法是，在 FileOutputFormat.getWorkOutputPath() 路径下创建side-file即可。

对于只使用map不使用reduce的作业，这个结论也成立。这种情况下，map的输出结果直接生成到HDFS上。

6.6.2. RecordWriter

RecordWriter 生成<key, value> 对到输出文件。

RecordWriter的实现把作业的输出结果写到 FileSystem。

6.7. 其他有用的特性

6.7.1. Counters

Counters 是多个由Map/Reduce框架或者应用程序定义的全局计数器。 每一个Counter可以是任何一种 Enum类型。同一特定Enum类型的Counter可以汇集到一个组，其类型为Counters.Group。

应用程序可以定义任意(Enum类型)的Counters并且可以通过 map 或者 reduce方法中的 Reporter.incrCounter(Enum, long)或者 Reporter.incrCounter(String, String, long) 更新。之后框架会汇总这些全局counters。

6.7.2. DistributedCache

DistributedCache 可将具体应用相关的、大尺寸的、只读的文件有效地分布放置。

DistributedCache 是Map/Reduce框架提供的功能，能够缓存应用程序所需的文件 （包括文本，档案文件，jar文件等）。

应用程序在JobConf中通过url(hdfs://)指定需要被缓存的文件。 DistributedCache假定由hdfs://格式url指定的文件已经在 FileSystem上了。

Map-Redcue框架在作业所有任务执行之前会把必要的文件拷贝到slave节点上。 它运行高效是因为每个作业的文件只拷贝一次并且为那些没有文档的slave节点缓存文档。

DistributedCache 根据缓存文档修改的时间戳进行追踪。 在作业执行期间，当前应用程序或者外部程序不能修改缓存文件。

distributedCache可以分发简单的只读数据或文本文件，也可以分发复杂类型的文件例如归档文件和jar文件。归档文件(zip,tar,tgz和tar.gz文件)在slave节点上会被解档（un-archived）。 这些文件可以设置执行权限。

用户可以通过设置mapred.cache.{files|archives}来分发文件。 如果要分发多个文件，可以使用逗号分隔文件所在路径。也可以利用API来设置该属性: DistributedCache.addCacheFile(URI,conf)/ DistributedCache.addCacheArchive(URI,conf) and DistributedCache.setCacheFiles(URIs,conf)/ DistributedCache.setCacheArchives(URIs,conf) 其中URI的形式是 hdfs://host:port/absolute-path#link-name 在Streaming程序中，可以通过命令行选项 -cacheFile/-cacheArchive 分发文件。

用户可以通过 DistributedCache.createSymlink(Configuration)方法让 DistributedCache 在当前工作目录下创建到缓存文件的符号链接。 或者通过设置配置文件属性mapred.create.symlink为yes。 分布式缓存会截取URI的片段作为链接的名字。 例如，URI是 hdfs://namenode:port/lib.so.1#lib.so， 则在task当前工作目录会有名为lib.so的链接， 它会链接分布式缓存中的lib.so.1。

DistributedCache可在map/reduce任务中作为 一种基础软件分发机制使用。它可以被用于分发jar包和本地库（native libraries）。 DistributedCache.addArchiveToClassPath(Path, Configuration)和 DistributedCache.addFileToClassPath(Path, Configuration) API能够被用于 缓存文件和jar包，并把它们加入子jvm的classpath。也可以通过设置配置文档里的属性 mapred.job.classpath.{files|archives}达到相同的效果。缓存文件可用于分发和装载本地库。

6.7.3. Tool

Tool 接口支持处理常用的Hadoop命令行选项。

Tool 是Map/Reduce工具或应用的标准。应用程序应只处理其定制参数， 要把标准命令行选项通过 ToolRunner.run(Tool, String[]) 委托给 GenericOptionsParser处理。

Hadoop命令行的常用选项有：
```
-conf <configuration file>
-D <property=value>
-fs <local|namenode:port>
-jt <local|jobtracker:port>
```

6.7.4. IsolationRunner

IsolationRunner 是帮助调试Map/Reduce程序的工具。

使用IsolationRunner的方法是，首先设置 keep.failed.task.files属性为true （同时参考keep.task.files.pattern）。

然后，登录到任务运行失败的节点上，进入 TaskTracker的本地路径运行 IsolationRunner：
```
$ cd <local path>/taskTracker/${taskid}/work
$ bin/hadoop org.apache.hadoop.mapred.IsolationRunner ../job.xml
```

IsolationRunner会把失败的任务放在单独的一个能够调试的jvm上运行，并且采用和之前完全一样的输入数据。

6.7.5. Profiling

Profiling是一个工具，它使用内置的java profiler工具进行分析获得(2-3个)map或reduce样例运行分析报告。

用户可以通过设置属性mapred.task.profile指定系统是否采集profiler信息。 利用api JobConf.setProfileEnabled(boolean)可以修改属性值。如果设为true， 则开启profiling功能。profiler信息保存在用户日志目录下。缺省情况，profiling功能是关闭的。

如果用户设定使用profiling功能，可以使用配置文档里的属性mapred.task.profile.{maps|reduces} 设置要profile map/reduce task的范围。设置该属性值的api是 JobConf.setProfileTaskRange(boolean,String)。 范围的缺省值是0-2。

用户可以通过设定配置文档里的属性mapred.task.profile.params 来指定profiler配置参数。修改属性要使用api JobConf.setProfileParams(String)。当运行task时，如果字符串包含%s。 它会被替换成profileing的输出文件名。这些参数会在命令行里传递到子JVM中。缺省的profiling 参数是 -agentlib:hprof=cpu=samples,heap=sites,force=n,thread=y,verbose=n,file=%s。

6.7.6. 调试

Map/Reduce框架能够运行用户提供的用于调试的脚本程序。 当map/reduce任务失败时，用户可以通过运行脚本在任务日志（例如任务的标准输出、标准错误、系统日志以及作业配置文件）上做后续处理工作。用户提供的调试脚本程序的标准输出和标准错误会输出为诊断文件。如果需要的话这些输出结果也可以打印在用户界面上。

在接下来的章节，我们讨论如何与作业一起提交调试脚本。为了提交调试脚本， 首先要把这个脚本分发出去，而且还要在配置文件里设置。

6.7.6.1. 如何分发脚本文件:

用户要用 DistributedCache 机制来分发和链接脚本文件

6.7.6.2. 如何提交脚本:

一个快速提交调试脚本的方法是分别为需要调试的map任务和reduce任务设置 "mapred.map.task.debug.script" 和 "mapred.reduce.task.debug.script" 属性的值。这些属性也可以通过 JobConf.setMapDebugScript(String) 和 JobConf.setReduceDebugScript(String) API来设置。对于streaming， 可以分别为需要调试的map任务和reduce任务使用命令行选项-mapdebug 和 -reducedegug来提交调试脚本。

脚本的参数是任务的标准输出、标准错误、系统日志以及作业配置文件。在运行map/reduce失败的节点上运行调试命令是:
$script $stdout $stderr $syslog $jobconf

Pipes 程序根据第五个参数获得c++程序名。 因此调试pipes程序的命令是

$script $stdout $stderr $syslog $jobconf $program

6.7.6.3. 默认行为

对于pipes，默认的脚本会用gdb处理core dump， 打印 stack trace并且给出正在运行线程的信息。

6.7.7. JobControl

JobControl是一个工具，它封装了一组Map/Reduce作业以及他们之间的依赖关系。

6.7.8. 数据压缩

Hadoop Map/Reduce框架为应用程序的写入文件操作提供压缩工具，这些工具可以为map输出的中间数据和作业最终输出数据（例如reduce的输出）提供支持。它还附带了一些 CompressionCodec的实现，比如实现了 zlib和lzo压缩算法。 Hadoop同样支持gzip文件格式。

考虑到性能问题（zlib）以及Java类库的缺失（lzo）等因素，Hadoop也为上述压缩解压算法提供本地库的实现。更多的细节请参考 这里。

6.7.8.1. 中间输出

应用程序可以通过 JobConf.setCompressMapOutput(boolean)api控制map输出的中间结果，并且可以通过 JobConf.setMapOutputCompressorClass(Class)api指定CompressionCodec。

6.7.8.2. 作业输出

应用程序可以通过 FileOutputFormat.setCompressOutput(JobConf, boolean) api控制输出是否需要压缩并且可以使用 FileOutputFormat.setOutputCompressorClass(JobConf, Class)api指定CompressionCodec。

如果作业输出要保存成 SequenceFileOutputFormat格式，需要使用 SequenceFileOutputFormat.setOutputCompressionType(JobConf, SequenceFile.CompressionType)api，来设定 SequenceFile.CompressionType （i.e. RECORD / BLOCK - 默认是RECORD）。

7. 例子: WordCount v2.0

这里是一个更全面的WordCount例子，它使用了我们已经讨论过的很多Map/Reduce框架提供的功能。

运行这个例子需要HDFS的某些功能，特别是 DistributedCache相关功能。因此这个例子只能运行在 伪分布式 或者 完全分布式模式的 Hadoop上。

7.1. 源代码

	WordCount.java
1.	package org.myorg;
2.	
3.	import java.io.*;
4.	import java.util.*;
5.	
6.	import org.apache.hadoop.fs.Path;
7.	import org.apache.hadoop.filecache.DistributedCache;
8.	import org.apache.hadoop.conf.*;
9.	import org.apache.hadoop.io.*;
10.	import org.apache.hadoop.mapred.*;
11.	import org.apache.hadoop.util.*;
12.	
13.	public class WordCount extends Configured implements Tool {
14.	
15.	public static class Map extends MapReduceBase implements Mapper<LongWritable, Text, Text, IntWritable> {

16.	
17.	static enum Counters { INPUT_WORDS }
18.	
19.	private final static IntWritable one = new IntWritable(1);
20.	private Text word = new Text();
21.	
22.	private boolean caseSensitive = true;
23.	private Set<String> patternsToSkip = new HashSet<String>();
24.	
25.	private long numRecords = 0;
26.	private String inputFile;
27.	
28.	public void configure(JobConf job) {
29.	caseSensitive = job.getBoolean("wordcount.case.sensitive", true);
30.	inputFile = job.get("map.input.file");
31.	
32.	if (job.getBoolean("wordcount.skip.patterns", false)) {
33.	Path[] patternsFiles = new Path[0];
34.	try {
35.	patternsFiles = DistributedCache.getLocalCacheFiles(job);

```
36.         } catch (IOException ioe) {
37.           System.err.println("Caught exception while getting cached files: " + StringUtils.stringifyException(ioe));
38.         }
39.         for (Path patternsFile : patternsFiles) {
40.           parseSkipFile(patternsFile);
41.         }
42.       }
43.     }
44.
45.     private void parseSkipFile(Path patternsFile) {
46.       try {
47.         BufferedReader fis = new BufferedReader(new FileReader(patternsFile.toString()));
48.         String pattern = null;
49.         while ((pattern = fis.readLine()) != null) {
50.           patternsToSkip.add(pattern);
51.         }
52.       } catch (IOException ioe) {
53.         System.err.println("Caught exception while parsing the cached file '" + patternsFile + "' : " + StringUtils.stringifyException(ioe));
54.       }
```

55.	}
56.	
57.	public void map(LongWritable key, Text value, OutputCollector<Text, IntWritable> output, Reporter reporter) throws IOException {
58.	String line = (caseSensitive) ? value.toString() : value.toString().toLowerCase();
59.	
60.	for (String pattern : patternsToSkip) {
61.	line = line.replaceAll(pattern, "");
62.	}
63.	
64.	StringTokenizer tokenizer = new StringTokenizer(line);
65.	while (tokenizer.hasMoreTokens()) {
66.	word.set(tokenizer.nextToken());
67.	output.collect(word, one);
68.	reporter.incrCounter(Counters.INPUT_WORDS, 1);
69.	}
70.	
71.	if ((++numRecords % 100) == 0) {
72.	reporter.setStatus("Finished processing " + numRecords + " records " + "from the input file: " + inputFile);

73.	}
74.	}
75.	}
76.	
77.	public static class Reduce extends MapReduceBase implements Reducer<Text, IntWritable, Text, IntWritable> {
78.	public void reduce(Text key, Iterator<IntWritable> values, OutputCollector<Text, IntWritable> output, Reporter reporter) throws IOException {
79.	int sum = 0;
80.	while (values.hasNext()) {
81.	sum += values.next().get();
82.	}
83.	output.collect(key, new IntWritable(sum));
84.	}
85.	}
86.	
87.	public int run(String[] args) throws Exception {
88.	JobConf conf = new JobConf(getConf(), WordCount.class);
89.	conf.setJobName("wordcount");
90.	
91.	conf.setOutputKeyClass(Text.class);
92.	conf.setOutputValueClass(IntWritable.class);

```
93.
94.       conf.setMapperClass(Map.class);
95.       conf.setCombinerClass(Reduce.class);
96.       conf.setReducerClass(Reduce.class);
97.
98.       conf.setInputFormat(TextInputFormat.class);
99.       conf.setOutputFormat(TextOutputFormat.class);
100.
101.      List<String> other_args = new
ArrayList<String>();
102.      for (int i=0; i < args.length; ++i) {
103.        if ("-skip".equals(args[i])) {
104.          DistributedCache.addCacheFile(new
Path(args[++i]).toUri(), conf);
105.
conf.setBoolean("wordcount.skip.patterns",
true);
106.        } else {
107.          other_args.add(args[i]);
108.        }
109.      }
110.
111.      FileInputFormat.setInputPaths(conf,
new Path(other_args.get(0)));
112.      FileOutputFormat.setOutputPath(conf,
new Path(other_args.get(1)));
```

113.	
114.	JobClient.runJob(conf);
115.	return 0;
116.	}
117.	
118.	public static void main(String[] args) throws Exception {
119.	int res = ToolRunner.run(new Configuration(), new WordCount(), args);
120.	System.exit(res);
121.	}
122.	}
123.	

7.2. 运行样例

输入样例:

```
$ bin/hadoop dfs -ls /usr/joe/wordcount/input/
/usr/joe/wordcount/input/file01
/usr/joe/wordcount/input/file02
$ bin/hadoop dfs -cat /usr/joe/wordcount/input/file01
Hello World, Bye World!
$ bin/hadoop dfs -cat /usr/joe/wordcount/input/file02
Hello Hadoop, Goodbye to hadoop.
```

运行程序:

```
$ bin/hadoop jar /usr/joe/wordcount.jar org.myorg.WordCount
/usr/joe/wordcount/input /usr/joe/wordcount/output
```

输出:

```
$ bin/hadoop dfs -cat /usr/joe/wordcount/output/part-00000
```

```
Bye 1
Goodbye 1
Hadoop, 1
Hello 2
World! 1
World, 1
hadoop. 1
to 1
```

注意此时的输入与第一个版本的不同，输出的结果也有不同。

现在通过DistributedCache插入一个模式文件，文件中保存了要被忽略的单词模式。

```
$ hadoop dfs -cat /user/joe/wordcount/patterns.txt
\.
\,
\!
to
```

再运行一次，这次使用更多的选项：

```
$ bin/hadoop jar /usr/joe/wordcount.jar org.myorg.WordCount
-Dwordcount.case.sensitive=true /usr/joe/wordcount/input
/usr/joe/wordcount/output -skip /user/joe/wordcount/patterns.txt
```

应该得到这样的输出：

```
$ bin/hadoop dfs -cat /usr/joe/wordcount/output/part-00000
Bye 1
Goodbye 1
Hadoop 1
Hello 2
World 2
hadoop 1
```

再运行一次，这一次关闭大小写敏感性（case-sensitivity）：

```
$ bin/hadoop jar /usr/joe/wordcount.jar org.myorg.WordCount
-Dwordcount.case.sensitive=false /usr/joe/wordcount/input
/usr/joe/wordcount/output -skip /user/joe/wordcount/patterns.txt
```

输出:

```
$ bin/hadoop dfs -cat /usr/joe/wordcount/output/part-00000
bye 1
goodbye 1
hadoop 2
hello 2
world 2
```

7.3. 程序要点

通过使用一些Map/Reduce框架提供的功能, WordCount的第二个版本在原始版本基础上有了如下的改进:

- 展示了应用程序如何在Mapper (和Reducer)中通过configure方法 修改配置参数(28-43行)。
- 展示了作业如何使用DistributedCache 来分发只读数据。 这里允许用户指定单词的模式, 在计数时忽略那些符合模式的单词(104行)。
- 展示Tool接口和GenericOptionsParser处理Hadoop命令行选项的功能 (87-116, 119行)。
- 展示了应用程序如何使用Counters(68行), 如何通过传递给map (和reduce) 方法的Reporter实例来设置应用程序的状态信息(72行)。

Java和JNI是Sun Microsystems, Inc.在美国和其它国家的注册商标。