Hadoop 集群（第 6 期）

——WordCount 运行详解

1、MapReduce理论简介

1.1 MapReduce编程模型

MapReduce 采用“分而治之”的思想，把对大规模数据集的操作，分发给一个主节点管理下的各个分节点共同完成，然后通过整合各个节点的中间结果，得到最终结果。简单地说，MapReduce 就是“任务的分解与结果的汇总”。

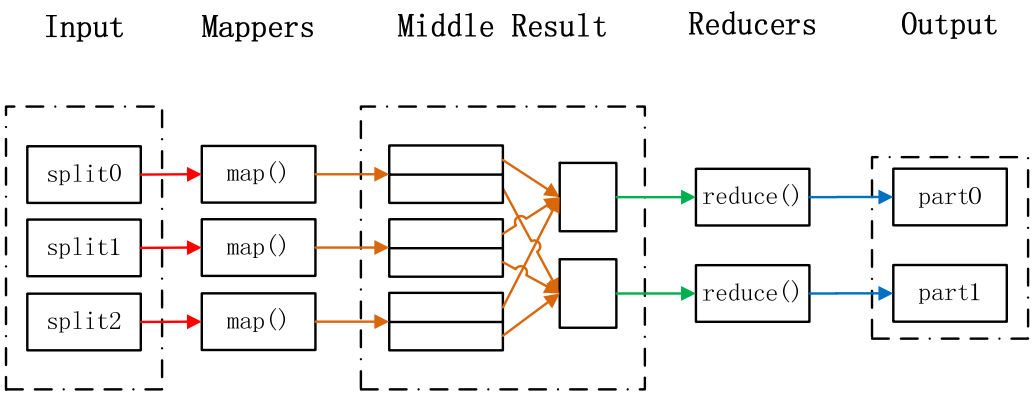
在 Hadoop 中，用于执行 MapReduce 任务的机器角色有两个：一个是 JobTracker；另一个是 TaskTracker，JobTracker 是用于调度工作的，TaskTracker 是用于执行工作的。一个 Hadoop 集群中只有一台 JobTracker。

在分布式计算中，MapReduce 框架负责处理了并行编程中分布式存储、工作调度、负载均衡、容错均衡、容错处理以及网络通信等复杂问题，把处理过程高度抽象为两个函数：map 和 reduce，map 负责把任务分解成多个任务，reduce 负责把分解后多任务处理的结果汇总起来。

需要注意的是，用 MapReduce 来处理的数据集（或任务）必须具备这样的特点：待处理的数据集可以分解成许多小的数据集，而且每一个小数据集都可以完全并行地进行处理。

1.2 MapReduce处理过程

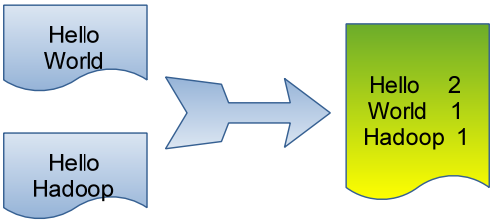
在 Hadoop 中，每个 MapReduce 任务都被初始化为一个 Job，每个 Job 又可以分为两种阶段：map 阶段和 reduce 阶段。这两个阶段分别用两个函数表示，即 map 函数和 reduce 函数。map 函数接收一个<key,value>形式的输入，然后同样产生一个<key,value>形式的中间输出，Hadoop 函数接收一个如<key,(list of values)>形式的输入，然后对这个 value 集合进行处理，每个 reduce 产生 0 或 1 个输出，reduce 的输出也是<key,value>形式的。



MapReduce 处理大数据集的过程

2、运行WordCount程序

单词计数是最简单也是最能体现 MapReduce 思想的程序之一，可以称为 MapReduce 版“Hello World”，该程序的完整代码可以在 Hadoop 安装包的“src/examples”目录下找到。单词计数主要完成功能是：统计一系列文本文件中每个单词出现的次数，如下图所示。



2.1 准备工作

现在以“hadoop”普通用户登录“Master.Hadoop”服务器。

1）创建本地示例文件

首先在“/home/hadoop”目录下创建文件夹“file”。

```
[hadoop@Master ~]$ ll
总用量 141372
-rw-r--r--. 1 hadoop hadoop 59468784  2月 27 03:31 hadoop-1.0.0.tar.gz
-rw-r--r--. 1 hadoop hadoop 85292206  2月 27 03:29 jdk-6u31-linux-i586.bin
[hadoop@Master ~]$ mkdir ~/file
[hadoop@Master ~]$ ll
总用量 141376
drwxrwxr-x. 2 hadoop hadoop     4096  3月  2 05:31 file
-rw-r--r--. 1 hadoop hadoop 59468784  2月 27 03:31 hadoop-1.0.0.tar.gz
-rw-r--r--. 1 hadoop hadoop 85292206  2月 27 03:29 jdk-6u31-linux-i586.bin
[hadoop@Master ~]$
```

接着创建两个文本文件 file1.txt 和 file2.txt，使 file1.txt 内容为“Hello World”，而 file2.txt 的内容为“Hello Hadoop”。

```
[hadoop@Master ~]$ cd file
[hadoop@Master file]$ echo "Hello World" > file1.txt
[hadoop@Master file]$ echo "Hello Hadoop" > file2.txt
[hadoop@Master file]$ ll
总用量 8
-rw-rw-r--. 1 hadoop hadoop 12  3月  2 05:35 file1.txt
-rw-rw-r--. 1 hadoop hadoop 13  3月  2 05:36 file2.txt
[hadoop@Master file]$ more file1.txt
Hello World
[hadoop@Master file]$ more file2.txt
Hello Hadoop
[hadoop@Master file]$
```

2）在 HDFS 上创建输入文件夹

```
[hadoop@Master ~]$ hadoop fs -mkdir input
[hadoop@Master ~]$ hadoop fs -ls
Found 1 items
drwxr-xr-x   - hadoop supergroup          0 2012-03-02 05:41 /user/hadoop/input
[hadoop@Master ~]$
```

3）上传本地 file 中文件到集群的 input 目录下

```
[hadoop@Master ~]$ hadoop fs -put ~/file/file*.txt input
[hadoop@Master ~]$ hadoop fs -ls input
Found 2 items
-rw-r--r--   1 hadoop supergroup         12 2012-03-02 05:45 /user/hadoop/input/file1.txt
-rw-r--r--   1 hadoop supergroup         13 2012-03-02 05:45 /user/hadoop/input/file2.txt
[hadoop@Master ~]$
```

2.2 运行例子

1）在集群上运行 WordCount 程序

备注：以 input 作为输入目录，output 目录作为输出目录。

已经编译好的 WordCount 的 Jar 在“/usr/hadoop”下面，就是“hadoop-examples-1.0.0.jar”，所以在下面执行命令时记得把路径写全了，不然会提示找不到该 Jar 包。

```
[hadoop@Master ~]$ ll /usr/hadoop | grep jar
-rw-rw-r--.  1 hadoop hadoop    6840 12月 16 00:39 hadoop-ant-1.0.0.jar
-rw-rw-r--.  1 hadoop hadoop 3740200 12月 16 00:39 hadoop-core-1.0.0.jar
-rw-rw-r--.  1 hadoop hadoop  142465 12月 16 00:39 hadoop-examples-1.0.0.jar
-rw-rw-r--.  1 hadoop hadoop 2530737 12月 16 00:39 hadoop-test-1.0.0.jar
-rw-rw-r--.  1 hadoop hadoop  287776 12月 16 00:39 hadoop-tools-1.0.0.jar
[hadoop@Master ~]$ hadoop jar /usr/hadoop/hadoop-examples-1.0.0.jar wordcount input output
```

输入、输出文件夹

执行“jar”命令　“WordCount”所在Jar包　程序主类名

2）MapReduce 执行过程显示信息

```
[hadoop@Master ~]$ hadoop jar /usr/hadoop/hadoop-examples-1.0.0.jar wordcount input output
12/03/02 06:07:36 INFO input.FileInputFormat: Total input paths to process : 2
12/03/02 06:07:36 INFO mapred.JobClient: Running job: job_201202292213_0002
12/03/02 06:07:37 INFO mapred.JobClient:  map 0% reduce 0%
12/03/02 06:07:51 INFO mapred.JobClient:  map 50% reduce 0%
12/03/02 06:07:52 INFO mapred.JobClient:  map 100% reduce 0%
12/03/02 06:08:06 INFO mapred.JobClient:  map 100% reduce 100%
12/03/02 06:08:11 INFO mapred.JobClient: Job complete: job_201202292213_0002
12/03/02 06:08:11 INFO mapred.JobClient: Counters: 29
```

Hadoop 命令会启动一个 JVM 来运行这个 MapReduce 程序，并自动获得 Hadoop 的配置，同时把类的路径（及其依赖关系）加入到 Hadoop 的库中。以上就是 Hadoop Job 的运行记录，从这里可以看到，这个 Job 被赋予了一个 ID 号：job_201202292213_0002，而且得知输入文件有两个（Total input paths to process : 2），同时还可以了解 map 的输入输出记录（record 数及字节数），以及 reduce 输入输出记录。比如说，在本例中，map 的 task 数量是 2 个，reduce

的 task 数量是一个。map 的输入 record 数是 2 个，输出 record 数是 4 个等信息。

2.3 查看结果

1）查看 HDFS 上 output 目录内容

```
[hadoop@Master ~]$ hadoop fs -ls output
Found 3 items
-rw-r--r--   1 hadoop supergroup          0 2012-03-02 06:08 /user/hadoop/output/_SUCCESS
drwxr-xr-x   - hadoop supergroup          0 2012-03-02 06:07 /user/hadoop/output/_logs
-rw-r--r--   1 hadoop supergroup         25 2012-03-02 06:07 /user/hadoop/output/part-r-00000
[hadoop@Master ~]$
```

从上图中知道生成了三个文件，我们的结果在“**part-r-00000**”中。

2）查看结果输出文件内容

```
[hadoop@Master ~]$ hadoop fs -cat output/part-r-00000
Hadoop  1
Hello   2
World   1
[hadoop@Master ~]$
```

3、WordCount源码分析

3.1 特别数据类型介绍

Hadoop 提供了如下内容的数据类型，这些数据类型都实现了 WritableComparable 接口，以便用这些类型定义的数据可以被序列化进行网络传输和文件存储，以及进行大小比较。

BooleanWritable：标准布尔型数值

ByteWritable：单字节数值

DoubleWritable：双字节数

FloatWritable：浮点数

IntWritable：整型数

LongWritable：长整型数

Text：使用 UTF8 格式存储的文本

NullWritable：当<key,value>中的 key 或 value 为空时使用

3.2 旧的WordCount分析

1）源代码程序

```
package org.apache.hadoop.examples;
```

```
import    java.io.IOException;
import    java.util.Iterator;
import    java.util.StringTokenizer;

import    org.apache.hadoop.fs.Path;
import    org.apache.hadoop.io.IntWritable;
import    org.apache.hadoop.io.LongWritable;
import    org.apache.hadoop.io.Text;
import    org.apache.hadoop.mapred.FileInputFormat;
import    org.apache.hadoop.mapred.FileOutputFormat;
import    org.apache.hadoop.mapred.JobClient;
import    org.apache.hadoop.mapred.JobConf;
import    org.apache.hadoop.mapred.MapReduceBase;
import    org.apache.hadoop.mapred.Mapper;
import    org.apache.hadoop.mapred.OutputCollector;
import    org.apache.hadoop.mapred.Reducer;
import    org.apache.hadoop.mapred.Reporter;
import    org.apache.hadoop.mapred.TextInputFormat;
import    org.apache.hadoop.mapred.TextOutputFormat;

public     class    WordCount
{

     public     static     class    Map    extends    MapReduceBase    implements
                Mapper<LongWritable, Text, Text, IntWritable>
     {
        private     final     static    IntWritable one =    new    IntWritable( 1 );
        private    Text    word =    new    Text();


           public     void    map(LongWritable key, Text value,
                      OutputCollector<Text, IntWritable> output, Reporter reporter)
                      throws    IOException
           {
                String line = value.toString();
                StringTokenizer tokenizer = new    StringTokenizer(line);
                while    (tokenizer.hasMoreTokens())
                {
                     word.set(tokenizer.nextToken());
                     output.collect(word, one);
                }
           }
     }
```

```
    public    static    class    Reduce    extends    MapReduceBase    implements
              Reducer<Text, IntWritable, Text, IntWritable>
    {
        public    void    reduce(Text key, Iterator<IntWritable> values,
                  OutputCollector<Text, IntWritable> output, Reporter reporter)
                  throws    IOException
        {
            int    sum =    0 ;
            while    (values.hasNext())
            {
                sum += values.next().get();
            }
            output.collect(key, new    IntWritable(sum));
        }
    }

    public    static    void    main(String[] args)    throws    Exception
    {
        JobConf conf = new    JobConf(WordCount. class );
        conf.setJobName("wordcount" );

        conf.setOutputKeyClass(Text.class );
        conf.setOutputValueClass(IntWritable.class );

        conf.setMapperClass(Map.class );
        conf.setCombinerClass(Reduce.class );
        conf.setReducerClass(Reduce.class );

        conf.setInputFormat(TextInputFormat.class );
        conf.setOutputFormat(TextOutputFormat.class );

        FileInputFormat.setInputPaths(conf, new    Path(args[ 0 ]));
        FileOutputFormat.setOutputPath(conf, new    Path(args[ 1 ]));

        JobClient.runJob(conf);
    }
}
```

3）主方法 Main 分析

```
public    static    void    main(String[] args)    throws    Exception
    {
        JobConf conf = new    JobConf(WordCount. class );
        conf.setJobName("wordcount" );
```

```
        conf.setOutputKeyClass(Text.class );
        conf.setOutputValueClass(IntWritable.class );

        conf.setMapperClass(Map.class );
        conf.setCombinerClass(Reduce.class );
        conf.setReducerClass(Reduce.class );

        conf.setInputFormat(TextInputFormat.class );
        conf.setOutputFormat(TextOutputFormat.class );

        FileInputFormat.setInputPaths(conf, new    Path(args[ 0 ]));
        FileOutputFormat.setOutputPath(conf, new    Path(args[ 1 ]));

        JobClient.runJob(conf);
    }
```

首先讲解一下 **Job** 的**初始化过程**。**main** 函数调用 **Jobconf** 类来对 **MapReduce Job** 进行初始化，然后调用 **setJobName()**方法命名这个 **Job**。对 Job 进行合理的命名有助于**更快**地找到 Job，以便在 JobTracker 和 Tasktracker 的页面中对其进行**监视**。

```
JobConf conf = new    JobConf(WordCount. class );
conf.setJobName("wordcount" );
```

接着设置 Job 输出结果<key,value>的中 key 和 value 数据类型，因为结果是<单词,个数>，所以 key 设置为“Text”类型，相当于 Java 中 String 类型。Value 设置为“IntWritable”，相当于 Java 中的 int 类型。

```
conf.setOutputKeyClass(Text.class );
conf.setOutputValueClass(IntWritable.class );
```

然后设置 Job 处理的 Map（拆分）、Combiner（中间结果合并）以及 Reduce（合并）的相关处理类。这里用 Reduce 类来进行 Map 产生的中间结果合并，避免给网络数据传输产生压力。

```
conf.setMapperClass(Map.class );
conf.setCombinerClass(Reduce.class );
conf.setReducerClass(Reduce.class );
```

接着就是调用 setInputPath()和 setOutputPath()设置输入输出路径。

```
conf.setInputFormat(TextInputFormat.class );
conf.setOutputFormat(TextOutputFormat.class );
```

（1）InputFormat 和 InputSplit

InputSplit 是 Hadoop 定义的用来传送给每个单独的 map 的数据，InputSplit 存储的并非数据本身，而是一个分片长度和一个记录数据位置的数组。生成 InputSplit 的方法可以通过 InputFormat()来设置。

当数据传送给 map 时，map 会将输入分片传送到 InputFormat，InputFormat 则调用方法 getRecordReader()生成 RecordReader，RecordReader 再通过 creatKey()、creatValue()方法创建可供 map 处理的<key,value>对。简而言之，InputFormat()方法是用来生成可供 map 处理的<key,value>对的。

Hadoop 预定义了多种方法将不同类型的输入数据转化为 map 能够处理的<key,value>对，它们都继承自 InputFormat，分别是：

```
InputFormat
    |
    |---BaileyBorweinPlouffe.BbpInputFormat
    |---ComposableInputFormat
    |---CompositeInputFormat
    |---DBInputFormat
    |---DistSum.Machine.AbstractInputFormat
    |---FileInputFormat
        |---CombineFileInputFormat
        |---KeyValueTextInputFormat
        |---NLineInputFormat
        |---SequenceFileInputFormat
        |---TeraInputFormat
        |---TextInputFormat
```

其中 TextInputFormat 是 Hadoop 默认的输入方法，在 TextInputFormat 中，每个文件（或其一部分）都会单独地作为 map 的输入，而这个是继承自 FileInputFormat 的。之后，每行数据都会生成一条记录，每条记录则表示成<key,value>形式：

- key 值是每个数据的记录在数据分片中字节偏移量，数据类型是 LongWritable；
- value 值是每行的内容，数据类型是 Text。

（2）OutputFormat

每一种输入格式都有一种输出格式与其对应。默认的输出格式是 TextOutputFormat，这种输出方式与输入类似，会将每条记录以一行的形式存入文本文件。不过，它的键和值可以是任意形式的，因为程序内容会调用 toString()方法将键和值转换为 String 类型再输出。

3）Map 类中 map 方法分析

```
public static class Map extends MapReduceBase implements
        Mapper<LongWritable, Text, Text, IntWritable>
{
    private final static IntWritable one = new IntWritable( 1 );
    private Text word = new Text();
```

```
        public    void   map(LongWritable key, Text value,
                  OutputCollector<Text, IntWritable> output, Reporter reporter)
                  throws   IOException
        {
            String line = value.toString();
            StringTokenizer tokenizer = new   StringTokenizer(line);
            while   (tokenizer.hasMoreTokens())
            {
                word.set(tokenizer.nextToken());
                output.collect(word, one);
            }
        }
    }
```

Map 类继承自 **MapReduceBase**，并且它实现了 **Mapper 接口**，此接口是一个**规范类型**，它有 4 种形式的参数，分别用来指定 map 的**输入** key 值类型、**输入** value 值类型、**输出** key 值类型和**输出** value 值类型。在本例中，因为使用的是 TextInputFormat，它的输出 key 值是 LongWritable 类型，输出 value 值是 Text 类型，所以 map 的输入类型为<LongWritable,Text>。在本例中需要输出<word,1>这样的形式，因此输出的 key 值类型是 Text，输出的 value 值类型是 IntWritable。

实现此接口类还需要实现 map 方法，map 方法会具体负责对输入进行操作，在本例中，map 方法对输入的行以空格为单位进行切分，然后使用 **OutputCollect** 收集输出的<word,1>。

4）Reduce 类中 reduce 方法分析

```
public    static    class   Reduce   extends   MapReduceBase   implements
          Reducer<Text, IntWritable, Text, IntWritable>
{
    public    void   reduce(Text key, Iterator<IntWritable> values,
              OutputCollector<Text, IntWritable> output, Reporter reporter)
              throws   IOException
    {
        int   sum =   0 ;
        while   (values.hasNext())
        {
            sum += values.next().get();
        }
        output.collect(key, new   IntWritable(sum));
    }
}
```

Reduce 类也是继承自 **MapReduceBase** 的，需要实现 Reducer 接口。Reduce 类以 map 的输出作为输入，因此 Reduce 的输入类型是<Text，Intwritable>。而 Reduce 的输出是**单词**

和它的数目，因此，它的输出类型是<Text,IntWritable>。Reduce 类也要实现 reduce 方法，在此方法中，reduce 函数将输入的 key 值作为输出的 key 值，然后将获得多个 value 值加起来，作为输出的值。

3.3 新的WordCount分析

1）源代码程序

```
package org.apache.hadoop.examples;

import java.io.IOException;
import java.util.StringTokenizer;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.util.GenericOptionsParser;

public class WordCount {

   public static class TokenizerMapper
        extends Mapper<Object, Text, Text, IntWritable>{

     private final static IntWritable one = new IntWritable(1);
     private Text word = new Text();

     public void map(Object key, Text value, Context context
                     ) throws IOException, InterruptedException {
       StringTokenizer itr = new StringTokenizer(value.toString());
       while (itr.hasMoreTokens()) {
         word.set(itr.nextToken());
         context.write(word, one);
       }
     }
   }

   public static class IntSumReducer
        extends Reducer<Text,IntWritable,Text,IntWritable> {
```

```
    private IntWritable result = new IntWritable();

    public void reduce(Text key, Iterable<IntWritable> values,
                       Context context
                       ) throws IOException, InterruptedException {
      int sum = 0;
      for (IntWritable val : values) {
        sum += val.get();
      }
      result.set(sum);
      context.write(key, result);
    }
  }

  public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    String[] otherArgs = new GenericOptionsParser(conf, args).getRemainingArgs();
    if (otherArgs.length != 2) {
      System.err.println("Usage: wordcount <in> <out>");
      System.exit(2);
    }
    Job job = new Job(conf, "word count");
    job.setJarByClass(WordCount.class);
    job.setMapperClass(TokenizerMapper.class);
    job.setCombinerClass(IntSumReducer.class);
    job.setReducerClass(IntSumReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    FileInputFormat.addInputPath(job, new Path(otherArgs[0]));
    FileOutputFormat.setOutputPath(job, new Path(otherArgs[1]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
```

1）Map 过程

```
public static class TokenizerMapper
        extends Mapper<Object, Text, Text, IntWritable>{

    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(Object key, Text value, Context context
                    ) throws IOException, InterruptedException {
```

```
        StringTokenizer itr = new StringTokenizer(value.toString());
        while (itr.hasMoreTokens()) {
          word.set(itr.nextToken());
          context.write(word, one);
        }
      }
    }
```

Map 过程需要继承 org.apache.hadoop.mapreduce 包中 **Mapper** 类，并**重写**其 map 方法。通过在 map 方法中添加两句把 key 值和 value 值输出到控制台的代码，可以发现 map 方法中 value 值存储的是文本文件中的一行（以回车符为行结束标记），而 key 值为该行的首字母相对于文本文件的首地址的偏移量。然后 StringTokenizer 类将每一行拆分成为一个个的单词，并将<word,1>作为 map 方法的结果输出，其余的工作都交有 **MapReduce 框架**处理。

2）Reduce 过程

```
public static class IntSumReducer
        extends Reducer<Text,IntWritable,Text,IntWritable> {
    private IntWritable result = new IntWritable();

    public void reduce(Text key, Iterable<IntWritable> values,
                       Context context
                       ) throws IOException, InterruptedException {
      int sum = 0;
      for (IntWritable val : values) {
        sum += val.get();
      }
      result.set(sum);
      context.write(key, result);
    }
  }
```

Reduce 过程需要继承 org.apache.hadoop.mapreduce 包中 **Reducer** 类，并**重写**其 reduce 方法。Map 过程输出<key,values>中 key 为单个单词，而 values 是对应单词的计数值所组成的列表，Map 的输出就是 Reduce 的输入，所以 reduce 方法只要遍历 values 并求和，即可得到某个单词的总次数。

3）执行 MapReduce 任务

```
public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    String[] otherArgs = new GenericOptionsParser(conf, args).getRemainingArgs();
    if (otherArgs.length != 2) {
      System.err.println("Usage: wordcount <in> <out>");
```

```
        System.exit(2);
    }
    Job job = new Job(conf, "word count");
    job.setJarByClass(WordCount.class);
    job.setMapperClass(TokenizerMapper.class);
    job.setCombinerClass(IntSumReducer.class);
    job.setReducerClass(IntSumReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    FileInputFormat.addInputPath(job, new Path(otherArgs[0]));
    FileOutputFormat.setOutputPath(job, new Path(otherArgs[1]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
```

在 MapReduce 中，由 Job 对象负责管理和运行一个计算任务，并通过 Job 的一些方法对任务的参数进行相关的设置。此处设置了使用 TokenizerMapper 完成 Map 过程中的处理和使用 IntSumReducer 完成 Combine 和 Reduce 过程中的处理。还设置了 Map 过程和 Reduce 过程的输出类型：key 的类型为 Text，value 的类型为 IntWritable。任务的输出和输入路径则由命令行参数指定，并由 FileInputFormat 和 FileOutputFormat 分别设定。完成相应任务的参数设定后，即可调用 **job.waitForCompletion()**方法执行任务。

4、WordCount处理过程

本节将对 WordCount 进行更详细的讲解。详细执行步骤如下：

1）将文件拆分成 splits，由于测试用的文件较小，所以每个文件为一个 split，并将文件按行分割形成<key,value>对，如图 4-1 所示。这一步由 MapReduce 框架自动完成，其中偏移量（即 key 值）包括了回车所占的字符数（Windows 和 Linux 环境会不同）。

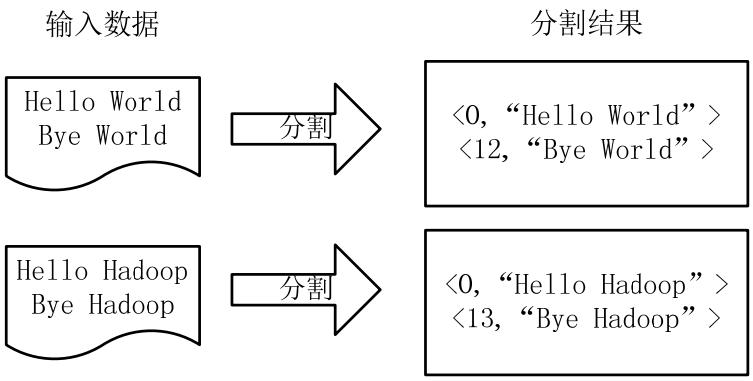


图 4-1 分割过程

2）将分割好的<key,value>对交给用户定义的 map 方法进行处理，生成新的<key,value>对，如图 4-2 所示。

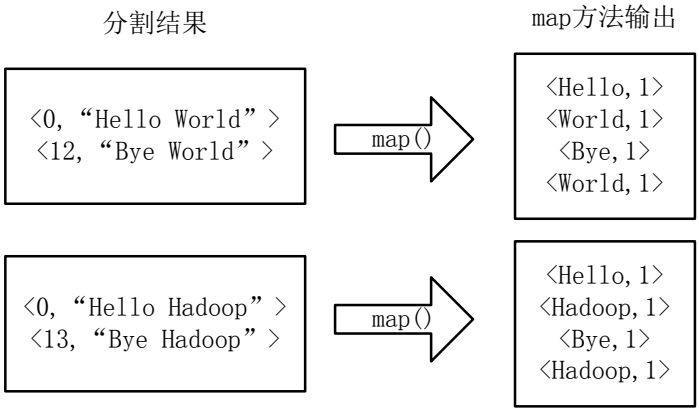


图 4-2 执行 map 方法

3）得到 map 方法输出的<key,value>对后，Mapper 会将它们按照 key 值进行排序，并执行 Combine 过程，将 key 至相同 value 值累加，得到 Mapper 的最终输出结果。如图 4-3 所示。

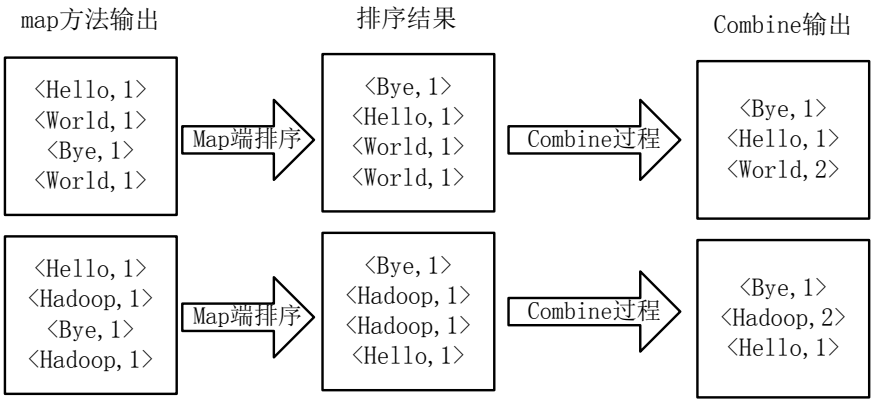


图 4-3 Map 端排序及 Combine 过程

4）Reducer 先对从 Mapper 接收的数据进行排序，再交由用户自定义的 reduce 方法进行处理，得到新的<key,value>对，并作为 WordCount 的输出结果，如图 4-4 所示。

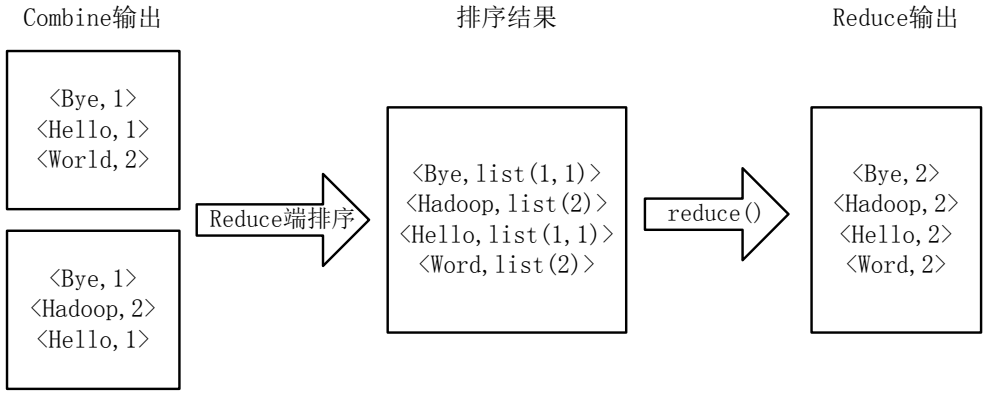


图 4-4 Reduce 端排序及输出结果

5、MapReduce新旧改变

Hadoop 最新版本的 MapReduce Release 0.20.0 的 API 包括了一个全新的 Mapreduce JAVA API，有时候也称为上下文对象。

新的 API 类型上不兼容以前的 API，所以，以前的应用程序需要重写才能使新的 API 发挥其作用 。

新的 API 和旧的 API 之间有下面几个明显的区别。

- 新的 API 倾向于使用抽象类，而不是接口，因为这更容易扩展。例如，你可以添加一个方法(用默认的实现)到一个抽象类而不需修改类之前的实现方法。在新的 API 中，Mapper 和 Reducer 是抽象类。
- 新的 API 是在 org.apache.hadoop.mapreduce 包(和子包)中的。之前版本的 API 则是放在 org.apache.hadoop.mapred 中的。
- 新的 API 广泛使用 context object(上下文对象)，并允许用户代码与 MapReduce 系统进行通信。例如，MapContext 基本上充当着 JobConf 的 OutputCollector 和 Reporter 的角色。
- 新的 API 同时支持"推"和"拉"式的迭代。在这两个新老 API 中，键/值记录对被推 mapper 中，但除此之外，新的 API 允许把记录从 map()方法中拉出，这也适用于 reducer。"拉"式的一个有用的例子是分批处理记录，而不是一个接一个。
- 新的 API 统一了配置。旧的 API 有一个特殊的 JobConf 对象用于作业配置，这是一个对于 Hadoop 通常的 Configuration 对象的扩展。在新的 API 中，这种区别没有了，所以作业配置通过 Configuration 来完成。作业控制的执行由 Job 类来负责，而不是 JobClient，它在新的 API 中已经荡然无存。

参考文献

感谢以下文章的编写作者，没有你们的铺路，我或许会走得很艰难，参考不分先后，贡献同等珍贵。

【1】Hadoop 实战——陆嘉恒——机械工业出版社

【2】实战 Hadoop——刘鹏——电子工业出版社

【3】Hadoop 上运行 WordCount 以及本地调试

地址：http://www.beoop.com/archives/244.html

【4】命令行运行 hadoop 实例 wordcount 程序

地址：http://blog.csdn.net/xw13106209/article/details/6862480

【5】Hadoop 示例程序 WordCount 运行及详解

地址：http://samuschen.iteye.com/blog/763940

【6】Hadoop 的安装与配置及示例 wordcount 的运行

地址：http://wenku.baidu.com/view/41eac9d850e2524de5187ef3.html