Day6 3月5日

软件51 庞建业 2151601012

运行wordcount程序

```
[root@hmaster ~]# cd /hadoopDev/

[root@hmaster hadoopDev]# vi t1.txt

[root@hmaster hadoopDev]# vi t2.txt
```

把本地文件放到hdfs存储空间:

```
[root@hmaster hadoopDev]# hadoop fs -mkdir  /input

[root@hmaster hadoopDev]# hadoop fs -put t1.txt t2.txt /input

[root@hmaster hadoopDev]# hadoop fs -ls /input
```

Found 2 items

```
-rw-r--r--   2 root supergroup         59 2018-03-03 16:54 /input/t1.txt

-rw-r--r--   2 root supergroup         33 2018-03-03 16:54 /input/t2.txt
```

```
[root@hmaster hadoopDev]# cd /hadoop/share/hadoop/mapreduce/
```

运行wordcount程序，/input是第一个参数，就是我们需要统计单词的文本文件. /output是第二个参数，会自动创建

```
[root@hmaster mapreduce]# hadoop jar hadoop-mapreduce-examples-3.0.0.jar wordcount /input
/output
```

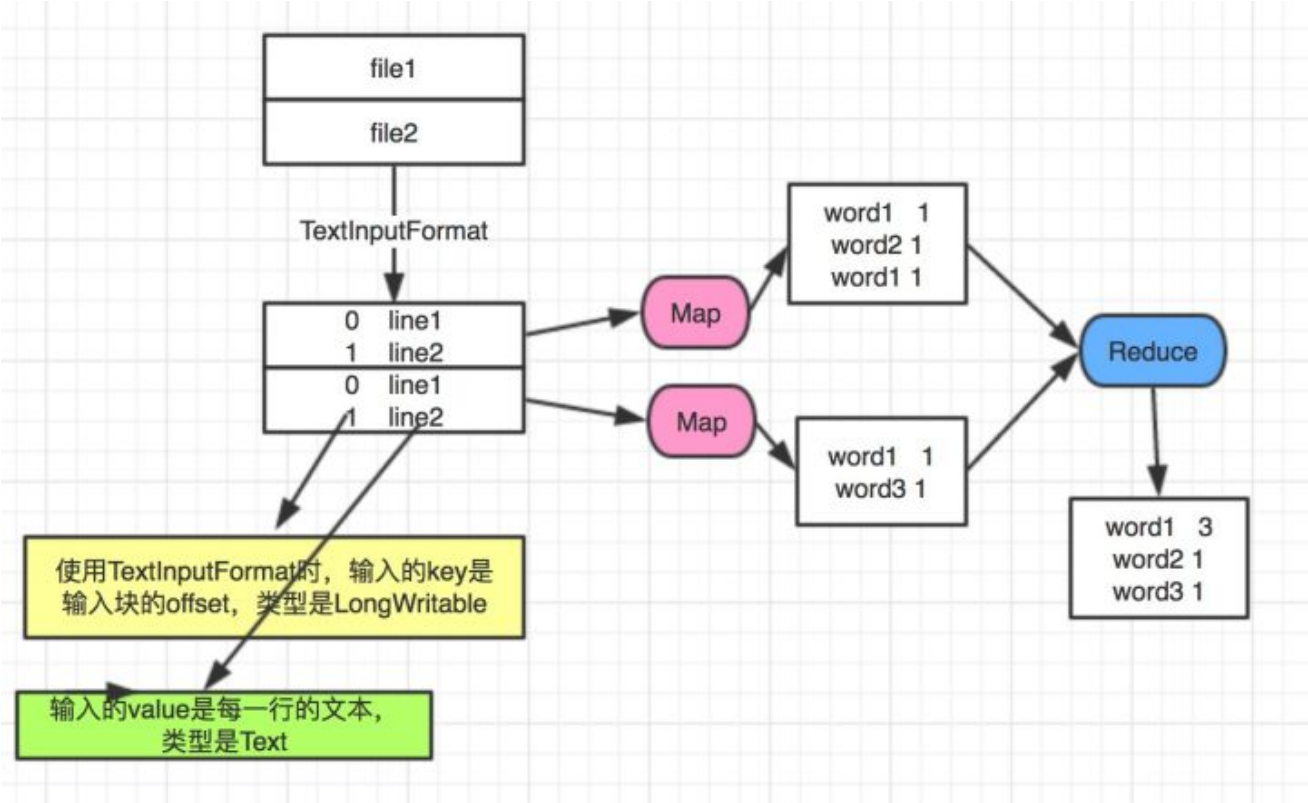
```
[root@hmaster mapreduce]# hadoop fs -ls /output
```

Found 2 items

```
-rw-r--r--   2 root supergroup          0 2018-03-03 17:08 /output/_SUCCESS

-rw-r--r--   2 root supergroup         78 2018-03-03 17:08 /output/part-r-00000
```

如果重新执行，需要把/output删除

```
hadoop fs -rm -rf /output
```



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：

- JobTracker用于调度工作的，一个Hadoop集群中只有一个JobTracker，位于master。
- TaskTracker用于执行工作，位于各slave上。

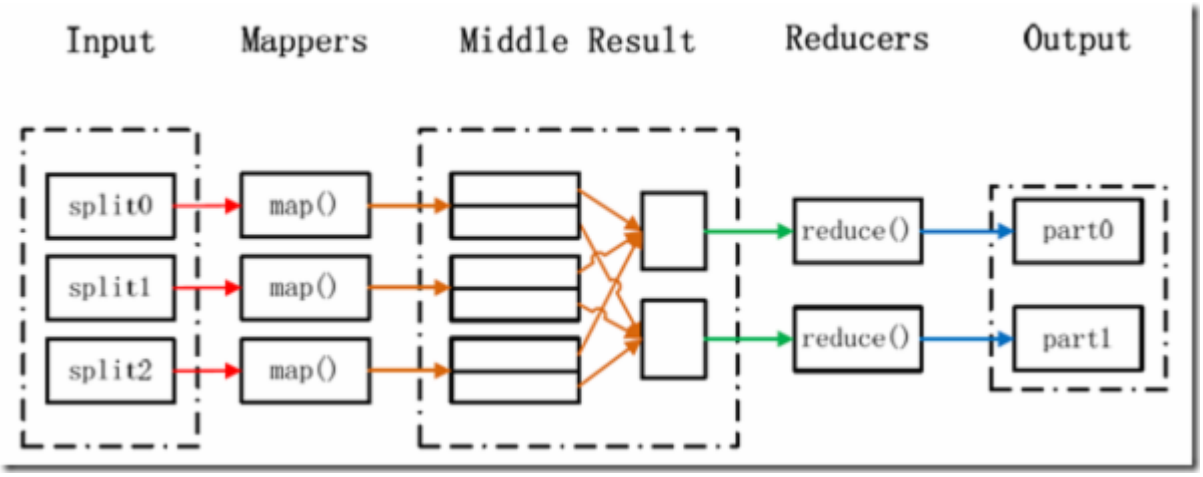
在分布式计算中，MapReduce框架负责处理了并行编程中分布式存储、工作调度、负载均衡、容错均衡、容错处理以及网络通信等复杂问题，把处理过程高度抽象为两个函数：map和reduce，map负责把任务分解成多个任务，reduce负责把分解后多任务处理的结果汇总起来。

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。

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。

- map：`(K1, V1)` ——> `list(K2, V2)`
- reduce：`(K2, list(V2))` ——> `list(K3, V3)`



WordCount

只包含三个文件：一个 Map 的 Java 文件，一个 Reduce 的 Java 文件，一个负责调用的主程序 Java 文件。

在当前用户的主文件夹下创建`wordcount_01/`目录，在该目录下再创建`src/`和`classes/`。 src 目录存放 Java 的源代码，classes 目录存放编译结果。

```
public class WordCount {

    public static class Map extends 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, Context context)
                throws IOException, InterruptedException {
            String line = value.toString();
            StringTokenizer tokenizer = new StringTokenizer(line);
            while (tokenizer.hasMoreTokens()) {
                word.set(tokenizer.nextToken());
                context.write(word, one);
            }
        }
    }

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

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

    public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();

        Job job = new Job(conf, "wordcount");
        job.setJarByClass(WordCount.class);

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

        job.setMapperClass(Map.class);
        job.setReducerClass(Reduce.class);
        /**
         * 设置一个本地combine,可以极大的消除本节点重复单词的计数,减小网络传输的开销
         */
        job.setCombinerClass(Reduce.class);

        job.setInputFormatClass(TextInputFormat.class);
        job.setOutputFormatClass(TextOutputFormat.class);

        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        job.waitForCompletion(true);
    }
}
```

构造两个文本文件, 把本地的两个文件拷贝到HDFS中:

```
➜  hadoop-examples git:(master) ✗ ./hadoop dfs -put wordcount-input/file* input
DEPRECATED: Use of this script to execute hdfs command is deprecated.
Instead use the hdfs command for it.

➜  hadoop-examples git:(master) ✗ ./hadoop dfs -ls input/
DEPRECATED: Use of this script to execute hdfs command is deprecated.
Instead use the hdfs command for it.

Found 2 items
-rw-r--r--   1 vonzhou supergroup         42 2016-12-03 23:17 input/file1
-rw-r--r--   1 vonzhou supergroup         43 2016-12-03 23:17 input/file2
```

编译程序得到jar:

```
mvn clean package
```

运行程序（指定main class的时候需要全包名限定）

查看执行的结果：

```
hadoop-examples git:(master) ✗ ./hadoop dfs -ls output
```

```
big     1
by      1
data    1
google  1
hadoop  2
hello   2
learning        1
papers  1
step    2
vonzhou 1
world   1
```

程序分析

Hadoop数据类型

Hadoop MapReduce操作的是键值对，但这些键值对并不是Integer、String等标准的Java类型。为了让键值对可以在集群上移动，Hadoop提供了一些实现了`WritableComparable`接口的基本数据类型，以便用这些类型定义的数据可以被**序列化**进行网络传输、文件存储与大小比较。

- 值：仅会被简单的传递，必须实现`Writable`或`WritableComparable`接口。
- 键：在Reduce阶段排序时需要进行比较，故只能实现`WritableComparable`接口。

下面是8个预定义的Hadoop基本数据类型，它们均实现了`WritableComparable`接口：

类	描述
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

流程

Map过程

```
public class TokenizerMapper extends Mapper<Object, Text, Text, IntWritable> {
    IntWritable one = new IntWritable(1);
    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。

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

其中的模板参数：第一个Object表示输入key的类型；第二个Text表示输入value的类型；第三个Text表示表示输出键的类型；第四个IntWritable表示输出值的类型。

作为map方法输入的键值对，其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

Reduce过程

```
public class IntSumReducer extends Reducer<Text, IntWritable, Text, IntWritable> {
    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** reduce方法。

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

其中模板参数同Map一样，依次表示是输入键类型，输入值类型，输出键类型，输出值类型。

```
public void reduce(Text key, Iterable<IntWritable> values, Context context)
```

reduce 方法的输入参数 key 为单个单词，而 values 是由各Mapper上对应单词的计数值所组成的列表（一个实现了 Iterable 接口的变量，可以理解成 values 里包含若干个 IntWritable 整数，可以通过迭代的方式遍历所有的值），所以只要遍历 values 并求和，即可得到某个单词出现的总次数。

执行作业

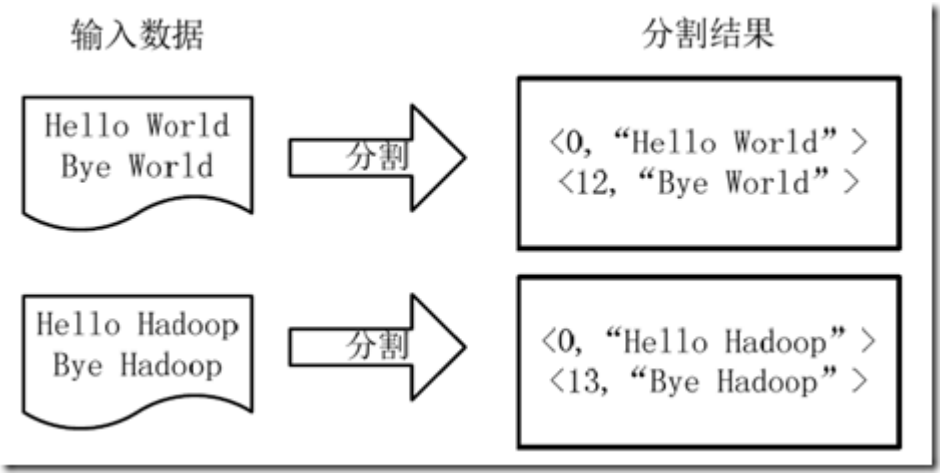
```
package com.lisong.hadoop;

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.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.util.GenericOptionsParser;

public class WordCount {
    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, "wordcount");
        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：

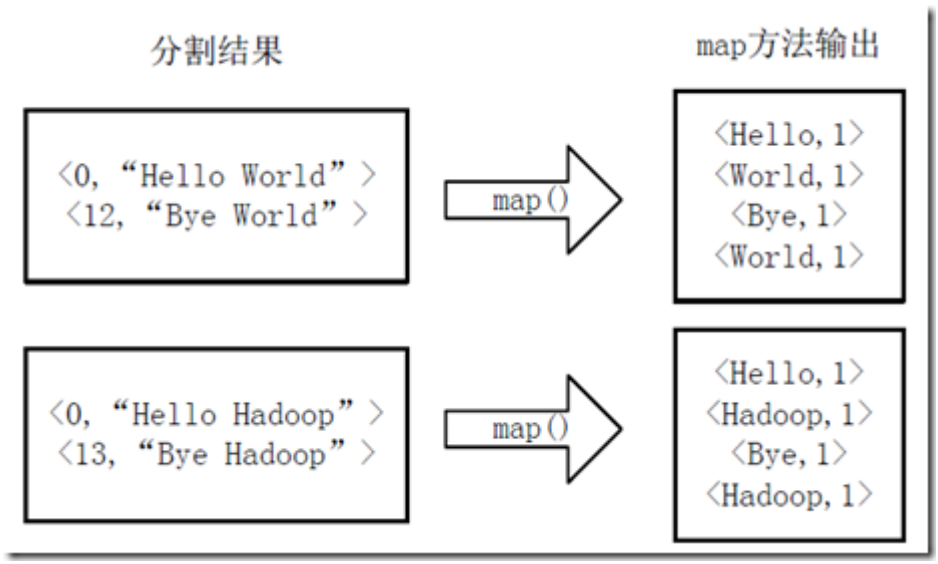
- 设置了使用`TokenizerMapper.class`完成Map过程中的处理，使用`IntSumReducer.class`完成Combine和Reduce过程中的处理。
- 还设置了Map过程和Reduce过程的输出类型：key的类型为Text，value的类型为IntWritable。

- 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。
    1. FileInputFormat类的很重要的作用就是将文件进行切分 split，并将 split 进一步拆分成key/value对
    2. FileOutputFormat类的作用是将处理结果写入输出文件。
- 完成相应任务的参数设定后，即可调用 `job.waitForCompletion()` 方法执行任务。

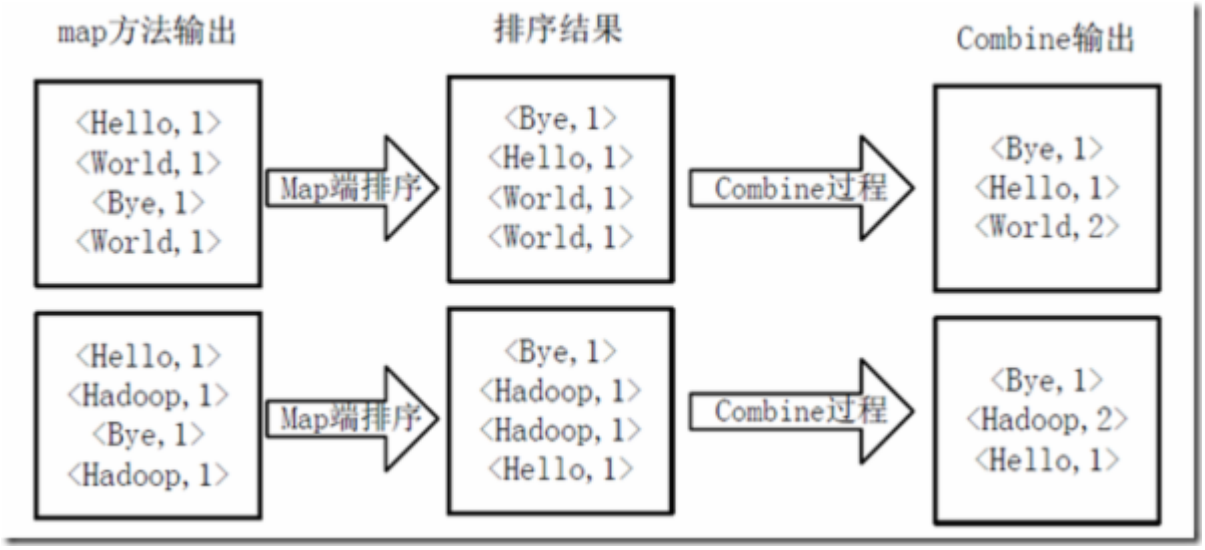
1）将文件拆分成splits，由于测试用的文件较小，所以每个文件为一个split，并将文件按行分割形成`<key,value>`对，key为偏移量（包括了回车符），value为文本行。这一步由MapReduce框架自动完成



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



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值相同的value值累加，得到Mapper的最终输出结果



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