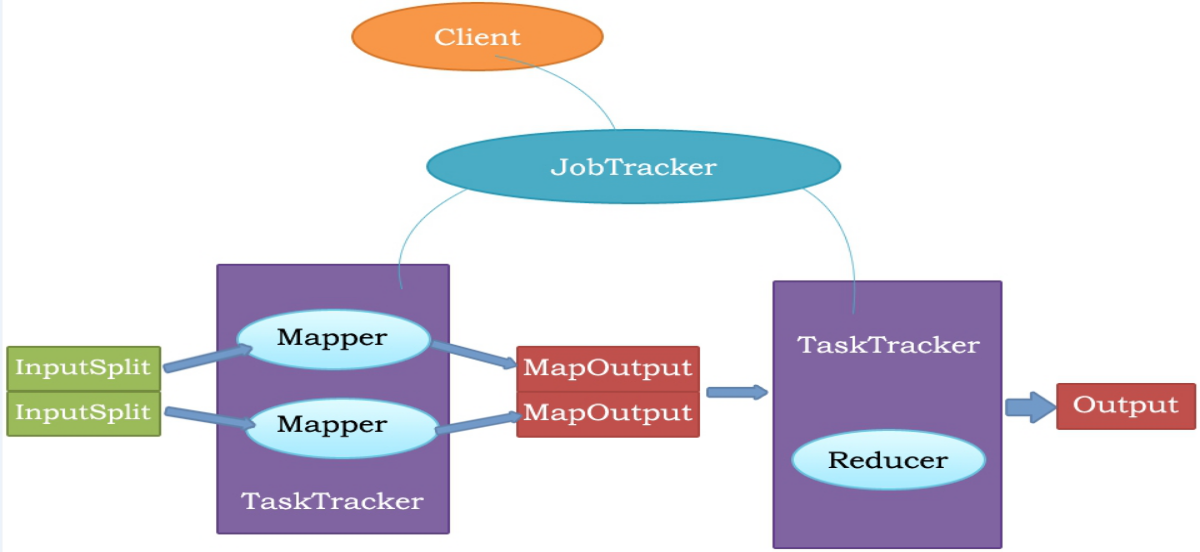
# **[MapReduce的原理及执行过程](https://www.cnblogs.com/ahu-lichang/p/6645074.html)**

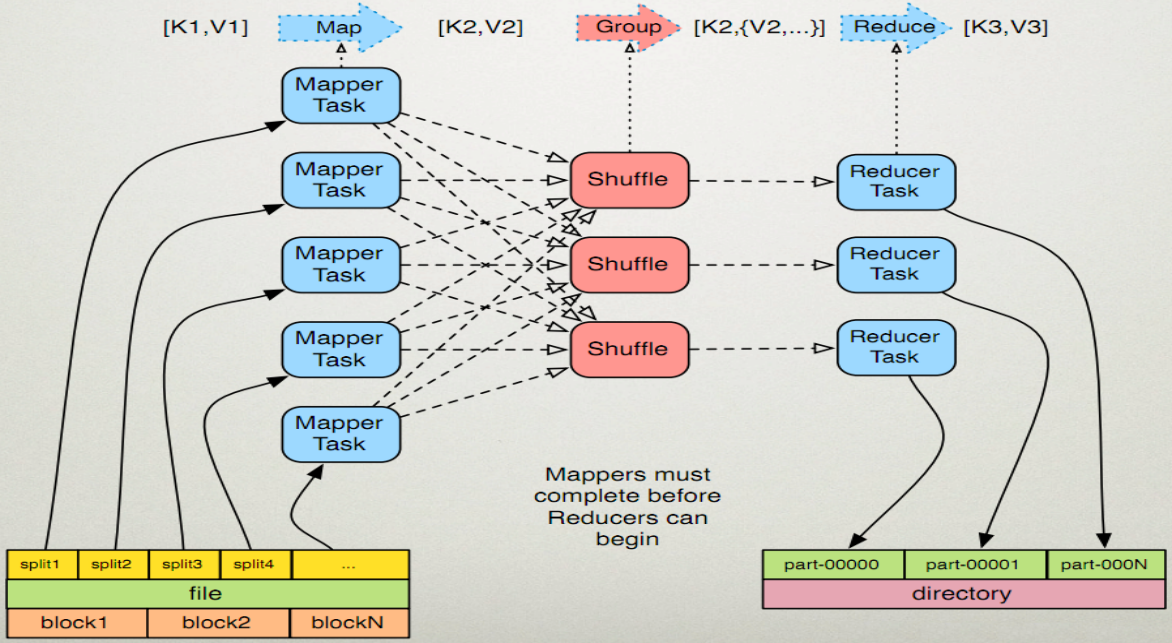
## 一、MapReduce简介

1. MapReduce是一种分布式计算模型，是Google提出的，主要用于搜索领域，解决海量数据的计算问题。
2. MR有两个阶段组成：Map和Reduce，用户只需实现map()和reduce()两个函数，即可实现分布式计算。

## 二、MapReduce执行流程



## **三、MapReduce原理**



## **四、MapReduce的执行步骤：**

1、Map任务处理

1.1 读取HDFS中的文件。每一行解析成一个<k,v>。每一个键值对调用一次map函数。

<0,hello you>

<10,hello me>

1.2 覆盖map()，接收1.1产生的<k,v>，进行处理，转换为新的<k,v>输出。

<hello,1><you,1><hello,1><me,1>

　　1.3 对1.2输出的<k,v>进行分区。默认分为一个区。详见《[Partitioner](http://www.cnblogs.com/ahu-lichang/p/6657895.html)》

　　1.4 对不同分区中的数据进行排序（按照k）、分组。分组指的是相同key的value放到一个集合中。　排序后：**<hello,1> <hello,1> <me,1> <you,1>**  分组后：**<hello,{1,1}><me,{1}><you,{1}>**

　　1.5 （可选）对分组后的数据进行归约。详见《[Combiner](http://www.cnblogs.com/ahu-lichang/p/6657572.html)》

2、Reduce任务处理

　　2.1 多个map任务的输出，按照不同的分区，通过网络copy到不同的reduce节点上。（**shuffle**）详见《[shuffle过程分析](http://www.cnblogs.com/ahu-lichang/p/6665242.html)》

　　2.2 对多个map的输出进行合并、排序。覆盖reduce函数，接收的是分组后的数据，实现自己的业务逻辑，　**<hello,2> <me,1> <you,1>**

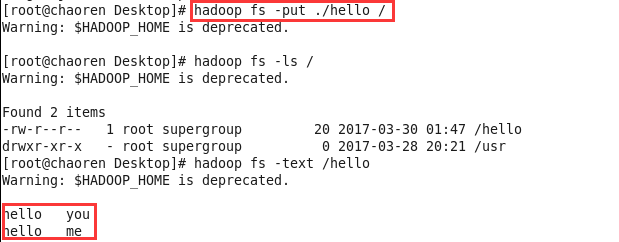
　　　　处理后，产生新的<k,v>输出。

　　2.3 对reduce输出的<k,v>写到HDFS中。

## **五、Java代码实现**

注：要导入org.apache.hadoop.fs.FileUtil.java。

1、先创建一个hello文件，上传到HDFS中



2、然后再编写代码，实现文件中的单词个数统计（代码中被注释掉的代码，是可以省略的，不省略也行）

package mapreduce;

import java.net.URI;

import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.fs.FileSystem;

import org.apache.hadoop.fs.Path;

import org.apache.hadoop.io.LongWritable;

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.input.TextInputFormat;

import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;

public class WordCountApp {

static final String INPUT\_PATH = "hdfs://chaoren:9000/hello";

static final String OUT\_PATH = "hdfs://chaoren:9000/out";

public static void main(String[] args) throws Exception {

Configuration conf = new Configuration();

FileSystem fileSystem = FileSystem.get(new URI(INPUT\_PATH), conf);

Path outPath = new Path(OUT\_PATH);

if (fileSystem.exists(outPath)) {

fileSystem.delete(outPath, true);

}

Job job = new Job(conf, WordCountApp.class.getSimpleName());

// 1.1指定读取的文件位于哪里

FileInputFormat.setInputPaths(job, INPUT\_PATH);

// 指定如何对输入的文件进行格式化，把输入文件每一行解析成键值对

//job.setInputFormatClass(TextInputFormat.class);

// 1.2指定自定义的map类

job.setMapperClass(MyMapper.class);

// map输出的<k,v>类型。如果<k3,v3>的类型与<k2,v2>类型一致，则可以省略

//job.setOutputKeyClass(Text.class);

//job.setOutputValueClass(LongWritable.class);

// 1.3分区

//job.setPartitionerClass(org.apache.hadoop.mapreduce.lib.partition.HashPartitioner.class);

// 有一个reduce任务运行

//job.setNumReduceTasks(1);

// 1.4排序、分组

// 1.5归约

// 2.2指定自定义reduce类

job.setReducerClass(MyReducer.class);

// 指定reduce的输出类型

job.setOutputKeyClass(Text.class);

job.setOutputValueClass(LongWritable.class);

// 2.3指定写出到哪里

FileOutputFormat.setOutputPath(job, outPath);

// 指定输出文件的格式化类

//job.setOutputFormatClass(TextOutputFormat.class);

// 把job提交给jobtracker运行

job.waitForCompletion(true);

}

/\*\*

\*

\* KEYIN 即K1 表示行的偏移量

\* VALUEIN 即V1 表示行文本内容

\* KEYOUT 即K2 表示行中出现的单词

\* VALUEOUT 即V2 表示行中出现的单词的次数，固定值1

\*

\*/

static class MyMapper extends

Mapper<LongWritable, Text, Text, LongWritable> {

protected void map(LongWritable k1, Text v1, Context context)

throws java.io.IOException, InterruptedException {

String[] splited = v1.toString().split("\t");

for (String word : splited) {

context.write(new Text(word), new LongWritable(1));

}

};

}

/\*\*

\* KEYIN 即K2 表示行中出现的单词

\* VALUEIN 即V2 表示出现的单词的次数

\* KEYOUT 即K3 表示行中出现的不同单词

\* VALUEOUT 即V3 表示行中出现的不同单词的总次数

\*/

static class MyReducer extends

Reducer<Text, LongWritable, Text, LongWritable> {

protected void reduce(Text k2, java.lang.Iterable<LongWritable> v2s,

Context ctx) throws java.io.IOException,

InterruptedException {

long times = 0L;

for (LongWritable count : v2s) {

times += count.get();

}

ctx.write(k2, new LongWritable(times));

};

}

}

3、运行成功后，可以在Linux中查看操作的结果

