Hadoop实战

HDFS把节点分成两类：NameNode和DataNode。NameNode是唯一的，程序与之通信，然后从DataNode上存取文件。这些操作是透明的，与普通的文件系统API没有区别。

MapReduce则是JobTracker节点为主，分配工作以及负责和用户程序通信。

HDFS和MapReduce实现是完全分离的，并不是没有HDFS就不能MapReduce运

WinStar 陈光跃

配置完成Hadoop之后，我们可以试着运行hadoop本身自带的例子，WordCount：

Hadoop的Helloword

1、在HDFS文件系统里新建一个文件夹：

$HADOOP_HOME/bin/hadoop fs -mkdir -p /WordCount/input

检查是否创建成功

$HADOOP_HOME/bin/hadoop fs -ls /WordCount

2、创建一个txt文件，里面写上一段英文句子

vim /data/words.txt

3、将txt文件拷入到HDFS文件系统的input里

$HADOOP_HOME/bin/hadoopfs -put /data/words.txt /WordCount/input

4、运行WordCount：

$HADOOP_HOME/bin/hadoop jar $HADOOP_HOME/share/hadoop/mapreduce/hadoop-mapreduce-examples-3.0.3-source.jar org.apache.hadoop.examples.WordCount /WordCount/input /WordCount/output

注意：output文件夹不可以存在，所以再次运行需要先删掉

运行

结果

```
successfully
2018-06-29 10:45:58,222 INFO mapreduce.Job: Counters: 53
        File System Counters
                FILE: Number of bytes read=295
                FILE: Number of bytes written=404955
                FILE: Number of read operations=0
                FILE: Number of large read operations=0
                FILE: Number of write operations=0
                HDFS: Number of bytes read=272
                HDFS: Number of bytes written=181
                HDFS: Number of read operations=8
                HDFS: Number of large read operations=0
                HDFS: Number of write operations=2
        Job Counters
                Launched map tasks=1
                Launched reduce tasks=1
                Data-local map tasks=1
                Total time spent by all maps in occupied slots (ms)=7141
                Total time spent by all reduces in occupied slots (ms)=5428
                Total time spent by all map tasks (ms)=7141
                Total time spent by all reduce tasks (ms)=5428
                Total vcore-milliseconds taken by all map tasks=7141
                Total vcore-milliseconds taken by all reduce tasks=5428
                Total megabyte-milliseconds taken by all map tasks=7312384
root@master:/home/cgy
                Failed Shuffles=0
                Merged Map outputs=1
                GC time elapsed (ms)=153
                CPU time spent (ms)=1600
                Physical memory (bytes) snapshot=390660096
                Virtual memory (bytes) snapshot=5203202048
                Total committed heap usage (bytes)=230821888
                Peak Map Physical memory (bytes)=239681536
                Peak Map Virtual memory (bytes)=2598219776
                Peak Reduce Physical memory (bytes)=150978560
                Peak Reduce Virtual memory (bytes)=2604982272
        Shuffle Errors
                BAD_ID=0
                CONNECTION=0
                IO_ERROR=0
                WRONG_LENGTH=0
                WRONG_MAP=0
                WRONG_REDUCE=0
        File Input Format Counters
                Bytes Read=163
        File Output Format Counters
                Bytes Written=181
```

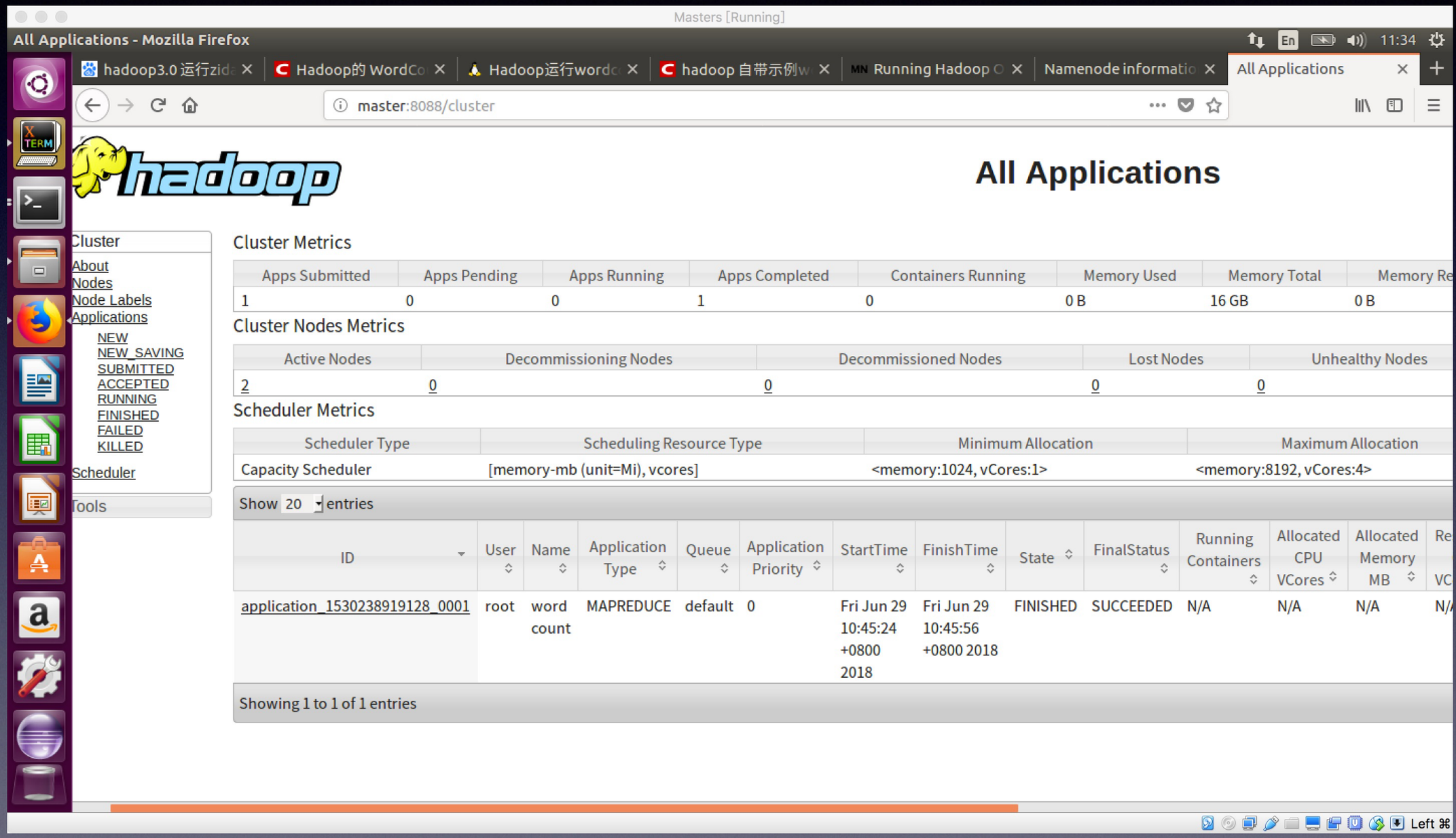
5、查看输出结果：

$HADOOP_HOME/bin/hadoop fs -cat /WordCount/output/part-r-00000

运行结果

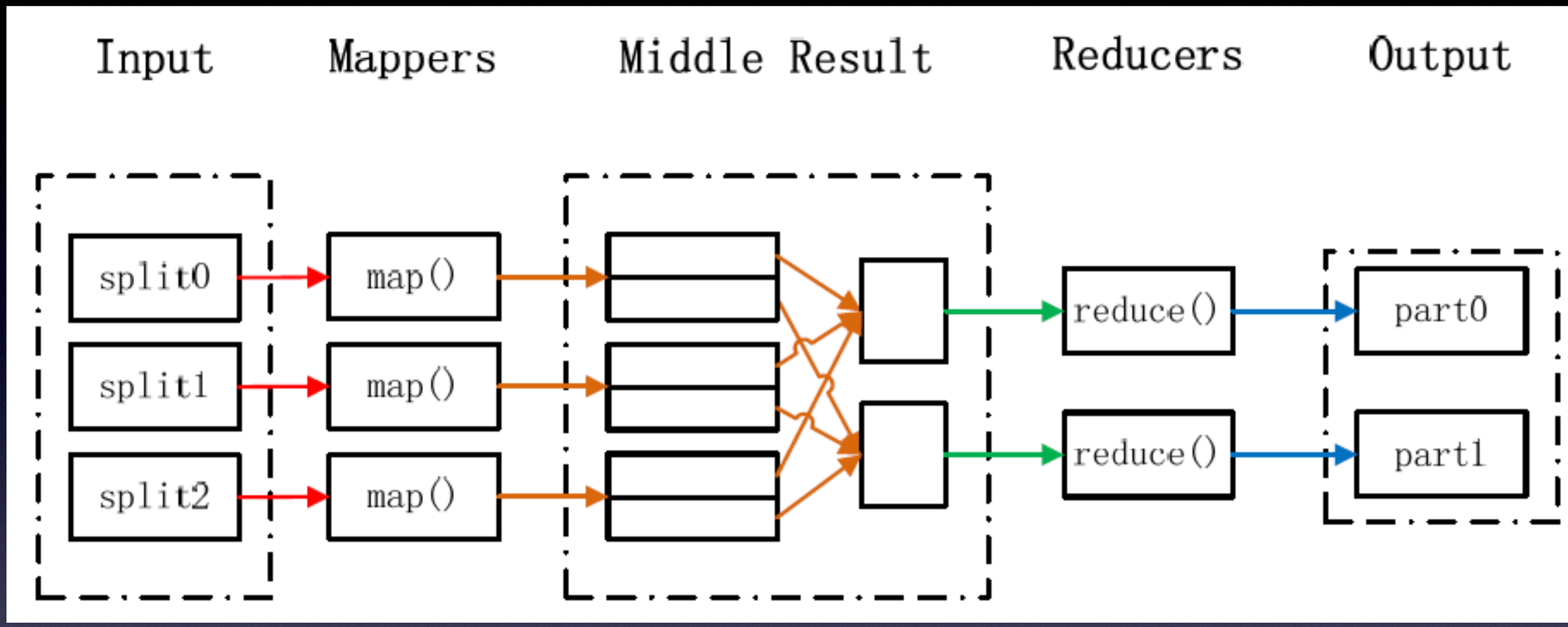
```
root@master:/home/cgy
t-r-00000
root@master:/home/cgy# $HADOOP_HOME/bin/hadoop fs -cat  /WordCount/output/par
-00000
,       4
.       1
be      1
can     1
first   2
hadoop  2
hello   1
i       1
in      1
is      2
it      2
it's    1
like    1
my      2
name    1
our     1
programming     1
so      1
start   1
step    1
the     1
```

6、Yarn查看进程



根据源码的编程详解

- WordCount使用的就是MapReducer的编程模型，是用java开源实现。
- MapReducer=Map+Reducer，其中Map负责任务的分配，Reducer负责执行工作并将结果汇总。



MapReducer执行过程

WinStar 陈光跃

WordCount:源码

```
 */
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);
      }
    }
  }
```

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。

```java
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> [<in>...] <out>");
    System.exit(2);
  }
  Job job = Job.getInstance(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);
  for (int i = 0; i < otherArgs.length - 1; ++i) {
    FileInputFormat.addInputPath(job, new Path(otherArgs[i]));
  }
  FileOutputFormat.setOutputPath(job,
    new Path(otherArgs[otherArgs.length - 1]));
  System.exit(job.waitForCompletion(true) ? 0 : 1);
 }
}
```

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。

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。

A Job object forms the specification of the job and gives you control over how the job is run. When we run this job on a Hadoop cluster, we will package the code into a JAR file (which Hadoop will distribute around the cluster). Rather than explicitly specifying the name of the JAR file, we can pass a class in the Job's setJarByClass() method, which Hadoop will use to locate the relevant JAR file by looking for the JAR file con− taining this class.

Having constructed a Job object, we specify the input and output paths. An input path is specified by calling the static addInputPath() method on FileInputFormat, and it can be a single file, a directory (in which case, the input forms all the files in that direc− tory), or a file pattern. As the name suggests, addInputPath() can be called more than once to use input from multiple paths.

The output path (of which there is only one) is specified by the static setOutput Path() method on FileOutputFormat. It specifies a directory where the output files from the reduce function are written. The directory shouldn't exist before running the job because Hadoop will complain and not run the job. This precaution is to prevent data loss (it can be very annoying to accidentally overwrite the output of a long job with that of another).

Next, we specify the map and reduce types to use via the setMapperClass() and setReducerClass() methods.

The setOutputKeyClass() and setOutputValueClass() methods control the output types for the reduce function, and must match what the Reduce class produces. The map output types default to the same types, so they do not need to be set if the mapper produces the same types as the reducer (as it does in our case). However, if they are different, the map output types must be set using the setMapOutputKeyClass() and setMapOutputValueClass() methods.

The input types are controlled via the input format, which we have not explicitly set because we are using the default TextInputFormat.

After setting the classes that define the map and reduce functions, we are ready to run the job. The waitForCompletion() method on Job submits the job and waits for it to finish. The single argument to the method is a flag indicating whether verbose output is generated. When true, the job writes information about its progress to the console.

The return value of the waitForCompletion() method is a Boolean indicating success (true) or failure (false), which we translate into the program's exit code of 0 or 1.