

**数据处理**

|  |  |  |
| --- | --- | --- |
| **学 号：** | **16281002** | |
| **姓 名：** | **杜永坤** | |
| **专 业：** | **计算机科学与技术** | |
| **学 院：** | **计算机与信息技术学院** | |
|  |  |  |
|  |  |  |
| **提交日期：** | **2019年05月31日** | |

目录

[1 实验目的 3](#_Toc10170855)

[2 实验环境 3](#_Toc10170856)

[3 实验说明 3](#_Toc10170857)

[4 实验内容 4](#_Toc10170858)

[1.MapReduce数据处理 4](#_Toc10170859)

[1）编程实现文件合并和去重操作；对于每行至少具有三个字段的两个输入文件，即文件A和文件B，请编写MapReduce程序，对两个文件进行合并，并剔除其中重复的内容，得到一个新的输出文件C。 4](#_Toc10170860)

[2）编写程序实现对输入文件的排序；现在有多个输入文件，每个文件中的每行内容均为一个整数。要求读取所有文件中的整数，进行升序排序后，输出到一个新的文件中，输出的数据格式为每行两个整数，第一个数字为第二个整数的排序位次，第二个整数为原待排列的整数。 8](#_Toc10170861)

[2.SPARK数据处理 14](#_Toc10170862)

[1）使用SPARK shell编程实现以下指定功能： 14](#_Toc10170863)

[2）编程熟悉Spark中的健值对操作，利用Spark的API完成wordcount实验，即统计一段文本中每个单词的出现总数。 19](#_Toc10170864)

[5 附录 23](#_Toc10170865)

**实验三**

**数据处理**实验

# 实验目的

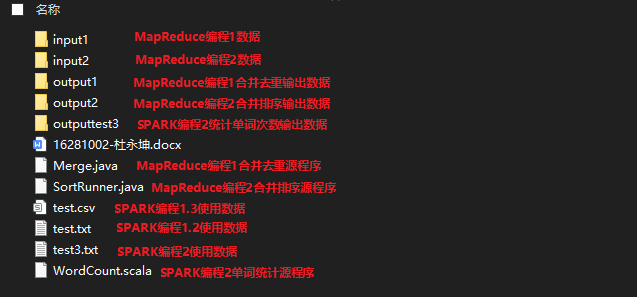
1. 理解MapReduce、SPARK在Hadoop大数据平台中的作用；
2. 掌握基本的MapReduce编程方法；
3. 了解MapReduce解决一些常见的数据处理问题；
4. 掌握基本的Spark shell编程；
5. 掌握基本的Spark API编程；

# 实验环境

实验平台：基于实验一搭建的虚拟机Hadoop大数据实验平台上的MapReduce、SPARK集群；

编程语言：JAVA、Scala；

# 实验说明

压缩包目录下包括实验报告，源代码，以及实验的输出结果：

# 实验内容

## 1.MapReduce数据处理

### 1）编程实现文件合并和去重操作；对于每行至少具有三个字段的两个输入文件，即文件A和文件B，请编写MapReduce程序，对两个文件进行合并，并剔除其中重复的内容，得到一个新的输出文件C。

**实验过程**

Merge.java:

import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.fs.FileSystem;

import org.apache.hadoop.fs.Path;

import org.apache.hadoop.io.Text;

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

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 java.io.IOException;

public class Merge {

//Map类，继承自Mapper类--一个抽象类

public static class Map extends Mapper<Object, Text, Text, Text>

{

private static Text text = new Text();

//重写map方法

public void map(Object key, Text value, Context content) throws IOException, InterruptedException

{

text = value

//底层通过Context content传递信息（即key value）

content.write(text, new Text(""));

}

}

//Reduce类，继承自Reducer类--一个抽象类

public static class Reduce extends Reducer<Text, Text, Text, Text>

{

public void reduce(Text key, Iterable<Text> values, Context context) throws IOException, InterruptedException

{

//对于所有的相同的key，只写入一个，相当于对于所有Iterable<Text> values，只执行一次write操作

context.write(key, new Text(""));

}

}

//main方法

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

final String OUTPUT\_PATH = "output1";

Configuration conf = new Configuration();

// conf.set("fs.defaultFS", "hdfs://localhost:9000");

Path path = new Path(OUTPUT\_PATH);

//加载配置文件

FileSystem fileSystem = path.getFileSystem(conf);

//输出目录若存在则删除

if (fileSystem.exists(new Path(OUTPUT\_PATH)))

{

fileSystem.delete(new Path(OUTPUT\_PATH),true);

}

//指定输入输出目录

String[] otherArgs = new String[]{"input1","output1"};

if (otherArgs.length != 2)

{

System.err.println("路径出错");

System.exit(2);

}

//一些初始化

Job job = Job.getInstance(conf,"Merge");

job.setJarByClass(Merge.class);

job.setMapperClass(Map.class); //初始化为自定义Map类

job.setReducerClass(Reduce.class); //初始化为自定义Reduce类

job.setOutputKeyClass(Text.class); //指定输出的key的类型，Text相当于String类

job.setOutputValueClass(Text.class); //指定输出的Value的类型，Text相当于String类

FileInputFormat.addInputPath(job, new Path(otherArgs[0])); //FileInputFormat指将输入的文件（若大于64M）进行切片划分，每个split切片对应一个Mapper任务

FileOutputFormat.setOutputPath(job, new Path(otherArgs[1]));

System.exit(job.waitForCompletion(true) ? 0 : 1);

}

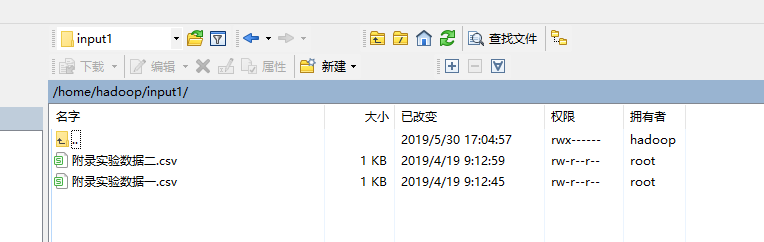
}

编译运行之前现在hadoop用户根目录下创建input1和output1目录。将

附录实验数据一：MapReduce编程一使用的数据文件A；

附录实验数据二：MapReduce编程一使用的数据文件B；

两个文件放到input1目录中：



然后编译运行：

编译

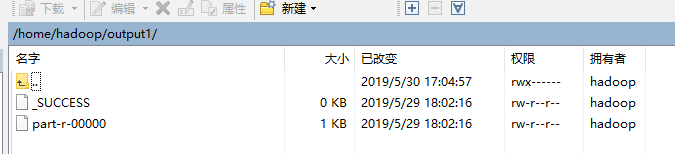
javac -cp /usr/local/hadoop-2.6.5/share/hadoop/common/\*:/usr/local/hadoop-2.6.5/share/hadoop/common/lib/\*:/usr/local/hadoop-2.6.5/share/hadoop/hdfs/lib/\*:/usr/local/hadoop-2.6.5/share/hadoop/hdfs/\*:/usr/local/hadoop-2.6.5/share/hadoop/mapreduce/\*:/usr/local/hadoop-2.6.5/share/hadoop/mapreduce/lib/\*:/usr/local/hadoop-2.6.5/share/hadoop/yarn/lib/\*:/usr/local/hadoop-2.6.5/share/hadoop/yarn/\*: Merge.java

执行

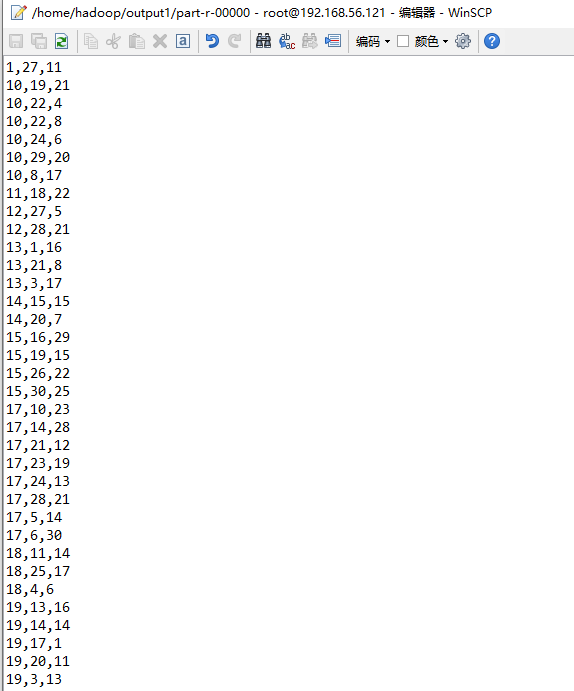
java -cp /usr/local/hadoop-2.6.5/share/hadoop/common/\*:/usr/local/hadoop-2.6.5/share/hadoop/common/lib/\*:/usr/local/hadoop-2.6.5/share/hadoop/hdfs/lib/\*:/usr/local/hadoop-2.6.5/share/hadoop/hdfs/\*:/usr/local/hadoop-2.6.5/share/hadoop/mapreduce/\*:/usr/local/hadoop-2.6.5/share/hadoop/mapreduce/lib/\*:/usr/local/hadoop-2.6.5/share/hadoop/yarn/lib/\*:/usr/local/hadoop-2.6.5/share/hadoop/yarn/\*: Merge

**结果：**

查看output1目录会发现目录下多出文件：



说明程序运行成功，结果在part-r-00000文件中：



### 2）编写程序实现对输入文件的排序；现在有多个输入文件，每个文件中的每行内容均为一个整数。要求读取所有文件中的整数，进行升序排序后，输出到一个新的文件中，输出的数据格式为每行两个整数，第一个数字为第二个整数的排序位次，第二个整数为原待排列的整数。

**实验过程**

SortRunner.java:

import org.apache.hadoop.io.Text;

import org.apache.hadoop.io.LongWritable;

import org.apache.hadoop.mapreduce.Mapper;

import java.io.IOException;

import org.apache.hadoop.io.IntWritable;

import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.fs.Path;

import org.apache.hadoop.mapreduce.Job;

import org.apache.hadoop.mapreduce.Partitioner;

import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;

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

import org.apache.hadoop.util.GenericOptionsParser;

import org.apache.hadoop.mapreduce.Mapper;

import org.apache.hadoop.mapreduce.Reducer;

import org.apache.hadoop.fs.FileSystem;

public class SortRunner {

public static class Partition extends Partitioner<IntWritable, IntWritable> {

@Override

public int getPartition(IntWritable key, IntWritable value,

int numPartitions) {

int MaxNumber = 65223;

int bound = MaxNumber / numPartitions + 1;

int keynumber = key.get();

for (int i = 0; i < numPartitions; i++) {

if (keynumber < bound \* i && keynumber >= bound \* (i - 1))

return i - 1;

}

return 0;

}

}

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

private static IntWritable data = new IntWritable();

public void map(Object key, Text value, Context context)

throws IOException, InterruptedException {

String line = value.toString();

data.set(Integer.parseInt(line));

context.write(data, new IntWritable(1));

}

}

public static class SortReducer extends Reducer<IntWritable, IntWritable, IntWritable, IntWritable>{

private static IntWritable linenum = new IntWritable(1);

public void reduce(IntWritable key, Iterable<IntWritable> values,

Context context) throws IOException, InterruptedException {

for (IntWritable val : values) {

context.write(linenum, key);

linenum = new IntWritable(linenum.get() + 1);

}

}

}

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

// TODO Auto-generated method stub

final String OUTPUT\_PATH = "output2";

Configuration conf = new Configuration();

// conf.set("fs.defaultFS", "hdfs://localhost:9000");

Path path = new Path(OUTPUT\_PATH);

//加载配置文件

FileSystem fileSystem = path.getFileSystem(conf);

//输出目录若存在则删除

if (fileSystem.exists(new Path(OUTPUT\_PATH)))

{

fileSystem.delete(new Path(OUTPUT\_PATH),true);

}

//指定输入输出目录

String[] otherArgs = new String[]{"input2","output2"};

if (otherArgs.length != 2)

{

System.err.println("路径出错");

System.exit(2);

}

Job job = new Job(conf, "SortRunner");

job.setJarByClass(SortRunner.class);

job.setMapperClass(SortMapper.class);

job.setPartitionerClass(Partition.class);

job.setReducerClass(SortReducer.class);

job.setOutputKeyClass(IntWritable.class);

job.setOutputValueClass(IntWritable.class);

FileInputFormat.addInputPath(job, new Path(otherArgs[0])); //FileInputFormat指将输入的文件（若大于64M）进行切片划分，每个split切片对应一个Mapper任务

FileOutputFormat.setOutputPath(job, new Path(otherArgs[1]));

System.exit(job.waitForCompletion(true) ? 0 : 1);

}

}

在hadoop用户根目录下创建input2和output2目录：

然后将实验数据：

附录实验数据三：MapReduce编程二使用的数据文件A；

附录实验数据四：MapReduce编程二使用的数据文件B；

放到目录input2目录下

做完上面的工作之后进行编译执行：

编译

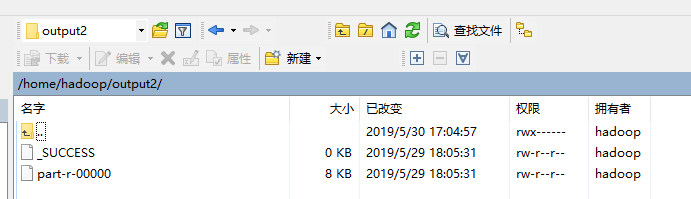
javac -cp /usr/local/hadoop-2.6.5/share/hadoop/common/\*:/usr/local/hadoop-2.6.5/share/hadoop/common/lib/\*:/usr/local/hadoop-2.6.5/share/hadoop/hdfs/lib/\*:/usr/local/hadoop-2.6.5/share/hadoop/hdfs/\*:/usr/local/hadoop-2.6.5/share/hadoop/mapreduce/\*:/usr/local/hadoop-2.6.5/share/hadoop/mapreduce/lib/\*:/usr/local/hadoop-2.6.5/share/hadoop/yarn/lib/\*:/usr/local/hadoop-2.6.5/share/hadoop/yarn/\*: SortRunner.java

执行

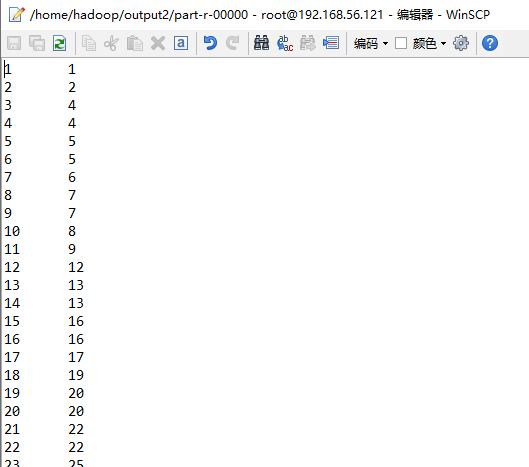
java -cp /usr/local/hadoop-2.6.5/share/hadoop/common/\*:/usr/local/hadoop-2.6.5/share/hadoop/common/lib/\*:/usr/local/hadoop-2.6.5/share/hadoop/hdfs/lib/\*:/usr/local/hadoop-2.6.5/share/hadoop/hdfs/\*:/usr/local/hadoop-2.6.5/share/hadoop/mapreduce/\*:/usr/local/hadoop-2.6.5/share/hadoop/mapreduce/lib/\*:/usr/local/hadoop-2.6.5/share/hadoop/yarn/lib/\*:/usr/local/hadoop-2.6.5/share/hadoop/yarn/\*: SortRunner

运行结果：

查看Output2目录保存的输出文件：



结果保存在part-r-00000：



## 2.SPARK数据处理

### 1）使用SPARK shell编程实现以下指定功能：

#### （1）创建RDD，并熟悉RDD中的转换操作，行动操作，并给出相应实例

**实验过程**

首先启动spark，进入到scala环境:

spark-shell --master local[4]

下面的实验也是在scala> 环境下直接输入scala程序进行运行

创建RDD

val input = sc.parallelize(List(1, 2, 3, 4))

转换操作map操作：

实例：计算元素的平方值

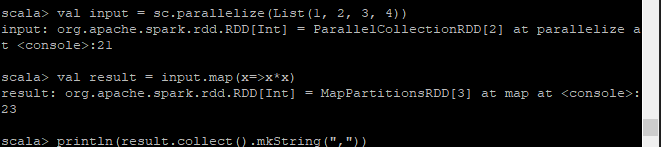
val result = input.map(x => x \* x)

行动操作collect操作：

实例：返回全部的元素

println(result.collect().mkString(","))

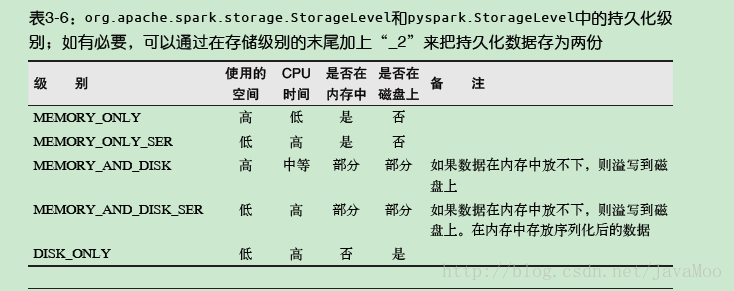
**实验结果：**





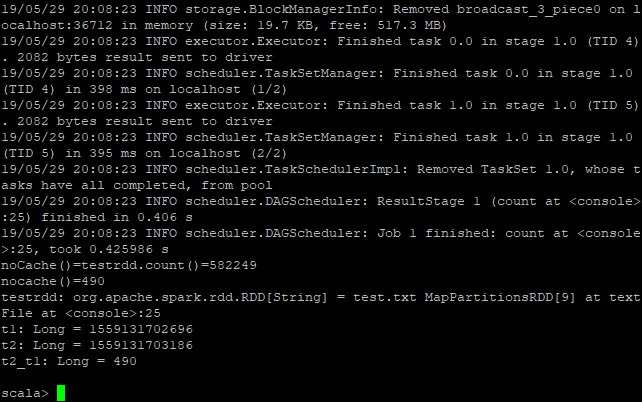
#### （2）了解如何将RDD中的计算过程进行持久化操作，并给出具体代码；

**实验过程**

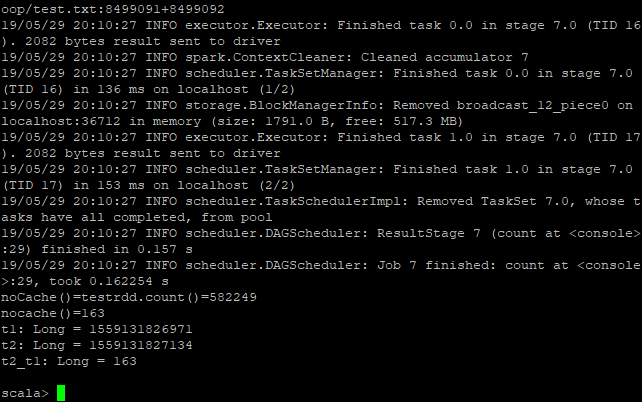
Spark RDD 是惰性求值的，而有时我们希望能多次使用同一个RDD。如果简单地对RDD 调用行动操作，Spark 每次都会重算RDD 以及它的所有依赖。这在迭代算法中消耗格外大。为了避免多次计算同一个RDD，可以让Spark 对数据进行持久化。出于不同的目的，我们可以为RDD 选择不同的持久化级别，持久化级别如下表

下面实验分别进行两次，第一次是没有调用persist() 方法来实现持久化操作打印出运行的时间，第二次调用了persist() 方法来实现持久化操作，打印出运行的时间。

**无持久化操作：**

val testrdd = sc.textFile("test.txt"); testrdd.count();val t1 = System.currentTimeMillis();println("noCache()=testrdd.count()=" + testrdd.count());val t2 = System.currentTimeMillis();val t2\_t1 = t2 - t1;println("nocache()=" + t2\_t1);

**持久化操作：**

val testrdd = sc.textFile("test.txt") .persist(StorageLevel.MEMORY\_ONLY());testrdd.count();val t1 = System.currentTimeMillis();println("noCache()=testrdd.count()=" + testrdd.count());val t2 = System.currentTimeMillis();val t2\_t1 = t2 - t1;println("nocache()=" + t2\_t1);

**结果分析**

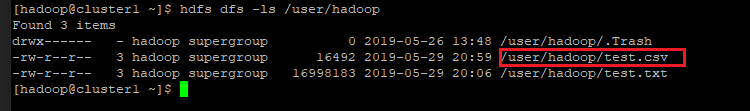
可以看到没有使用持久化操作的运行时间为490ms。而使用了持久化操作的运行时间为163ms，性能得到很大的提升，这是使用持久化带来的性能提升。

#### （3）了解Spark的数据读取与保存操作，尝试完成csv文件的读取和保存；

首先上传test.scv也就是

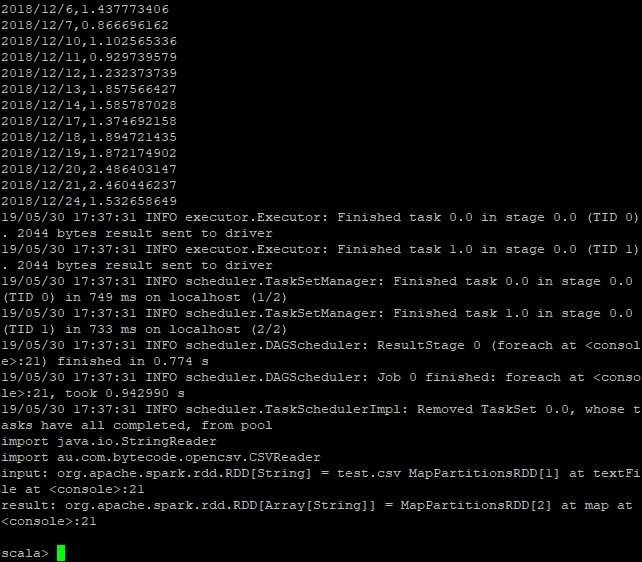
附录实验数据五：SPARK编程一使用的CSV数据文件；

上传到hdfs文件系统中：hdfa dfs -put test.csv /user/Hadoop/



输入下面的scala程序回车运行：

import java.io.StringReader;import au.com.bytecode.opencsv.CSVReader; val input=sc.textFile("test.csv");input.foreach(println);val result = input.map{line =>val reader = new CSVReader(new StringReader(line));reader.readNext();}

**实验结果：**

另一种实现方式是不能在scala环境直接运行，需要编写scala文件，在maven工程中或者 使用sbt 打包 Scala 程序，引入com.databricks.spark.csv对应的jar包才能实现：

import org.apache.spark.sql.SQLContext;val sqlContext = new SQLContext(sc);val df = sqlContext.load("com.databricks.spark.csv", Map("path" -> "test.csv", "header" -> "true"));df.select("日期", "数值").save("newcars.csv","com.databricks.spark.csv");

import org.apache.spark.sql.SQLContext;import com.databricks.spark.csv.\_;val sqlContext = new SQLContext(sc);val cars = sqlContext.csvFile("test.csv");cars.select("日期", "数值").saveAsCsvFile("newtest.csv")

### 2）编程熟悉Spark中的健值对操作，利用Spark的API完成wordcount实验，即统计一段文本中每个单词的出现总数。

实验过程

WordCount.scala文件内容：

import org.apache.spark.SparkContext

import org.apache.spark.SparkContext.\_

import org.apache.spark.SparkConf

import scala.collection.Map

object WordCount {

def main(args: Array[String]) {

val inputFile = "test3.txt"

val conf = new SparkConf().setMaster("local[2]").setAppName("WordCount")

conf.set("spark.testing.memory", "500000000")

val sc = new SparkContext(conf)

val textFile = sc.textFile(inputFile)

val wordCount = textFile.flatMap(line => line.split(" ")).map(word => (word, 1)).reduceByKey((a, b) => a + b)

wordCount.foreach(println)

wordCount.saveAsTextFile("outputtest3")

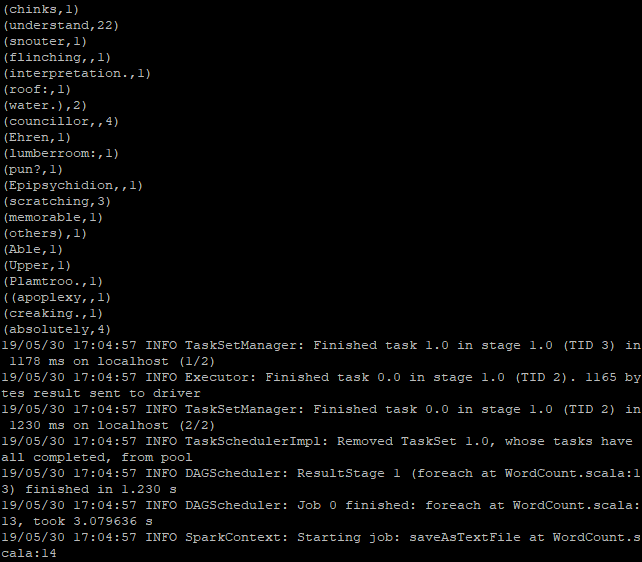
}

}

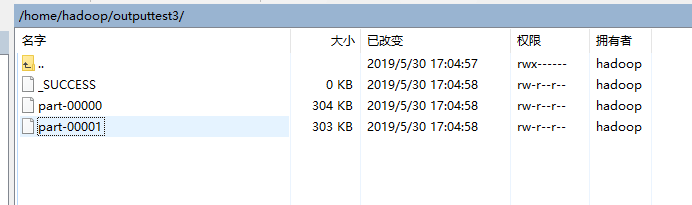
编译 scalac -cp /usr/local/spark-1.6.3-bin-hadoop2.6/lib/\*: WordCount.scala

运行 java -cp /usr/local/spark-1.6.3-bin-hadoop2.6/lib/\*: WordCount

**结果：**

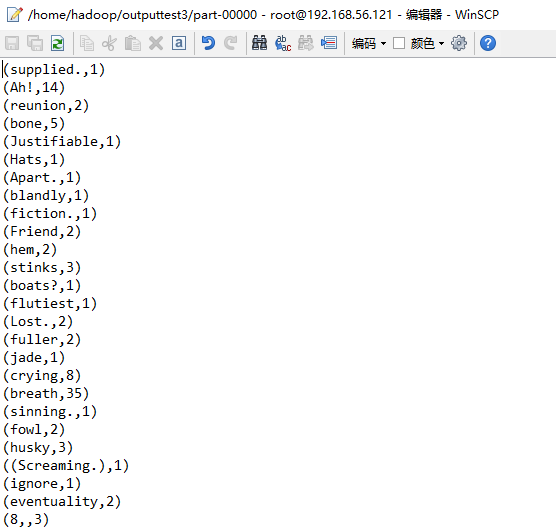


在hadoop用户根目录下outputtest3目录下：



由于输出内容太多，输出到两个文件中：

Part-r-00000：



Part-r-00001：

