

**数学与信息学院学生实验报告**

**实验课程名称：** 大数据技术概论 **教师： 林鑫泓 \_\_**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **实验项目名称** | **MapReduce初级编程实践** | | | **实验成绩** |  |
| **学生姓名** |  | **学 号** |  | **年级专业班级** |  |
| **小组成员** | **无** | | | **实验日期** |  |

# 1. 实验目的和要求

## 1.1 实验目的

* **通过实验掌握基本的MapReduce编程方法；**
* **掌握用MapReduce解决一些常见的数据处理问题，包括数据去重、数据排序和数据挖掘等。**

## 1.2 实验软硬件环境

* **操作系统：Ubuntu19.04**
* **Hadoop版本：3.2.1**

## 1.3 实验要求

**①****编程实现文件合并和去重操作**

**②编写程序实现对输入文件的排序**

**③** **对给定的表格进行信息挖掘**

# 2. 实验记录

## 2.1 编程实现文件合并和去重操作

对于两个输入文件，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。下面是输入文件和输出文件的一个样例供参考。

输入文件A的样例如下：

|  |
| --- |
| 20170101 x  20170102 y  20170103 x  20170104 y  20170105 z  20170106 x |

输入文件B的样例如下：

|  |
| --- |
| 20170101 y  20170102 y  20170103 x  20170104 z  20170105 y |

根据输入文件A和B合并得到的输出文件C的样例如下：

|  |
| --- |
| 20170101 x  20170101 y  20170102 y  20170103 x  20170104 y  20170104 z  20170105 y  20170105 z  20170106 x |

### **2.1.1 问题分析及思路**

问题要求是实现去重和合并功能，为了让输入文件中的数据只出现一次，然后shuffle过程会有合并相同key值，就可以在map过程中将输出的Key设置数据，然后value设置为空；shuffle过程中合并完，数据就只出现一次，即为key；最后Reduce过程中输出。

### **2.1.2 代码**

**import** java.io.IOException;

**import** org.apache.hadoop.conf.Configuration;

**import** org.apache.hadoop.fs.Path;

**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;

**public** **class** Test1 {

//重载 map 函数

//输入，key为Object类型，value为text类型；输出key，value都为为text类型

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

**private** **static** Text *text* = **new** Text();//用text类型存储

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

*text* = value;//将输入中的 value 复制到输出数据的 key 上，并直接输出。

context.write(*text*, **new** Text(""));//key为一行数据，value为空值

}

}

//重载 reduce 函数，直接将输入中的 key 复制到输出数据的 key 上

//用一行作为key，value是空，在reduce时进行汇总，只有一个key，即一行只出现一次，value还是空。实现了去重。

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

//<key,value-list>输入

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

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

}

}

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

Configuration conf = **new** Configuration();

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

//直接设置输入参数 设置程序运行参数

String[] otherArgs = **new** String[] { "/user/chengrui/input", "/user/chengrui/output" };

**if** (otherArgs.length != 2) {//没有指定参数

System.***err***.println("Usage: wordcount <in><out>");

System.*exit*(2);

}

Job job = Job.*getInstance*(conf, "去重");//设置环境参数

job.setJarByClass(Test1.**class**);//设置整个程序的类名

job.setMapperClass(Map.**class**);//添加Mapper类

job.setCombinerClass(Reduce.**class**);//添加combiner类

job.setReducerClass(Reduce.**class**);//添加reducer类

job.setOutputKeyClass(Text.**class**);//设置输出类型

job.setOutputValueClass(Text.**class**);//设置输出类型

FileInputFormat.*addInputPath*(job, **new** Path(otherArgs[0]));//设置输入文件

FileOutputFormat.*setOutputPath*(job, **new** Path(otherArgs[1]));//设置输出文件

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

}

}

### **2.1.3 实验过程**

本地创建两个文件1.txt，2.txt，内容为后面要上传到hadoop的输入文件。

文件内容如下

1.txt：

|  |
| --- |
| 20170101 x  20170102 y  20170103 x  20170104 y  20170105 z  20170106 x |

2.txt：

|  |
| --- |
| 20170101 y  20170102 y  20170103 x  20170104 z  20170105 y |

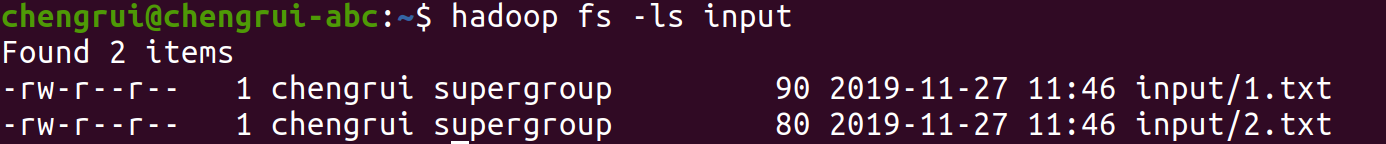
Hadoop创建输入目录



把本地文件上传到hadoop的输入目录中

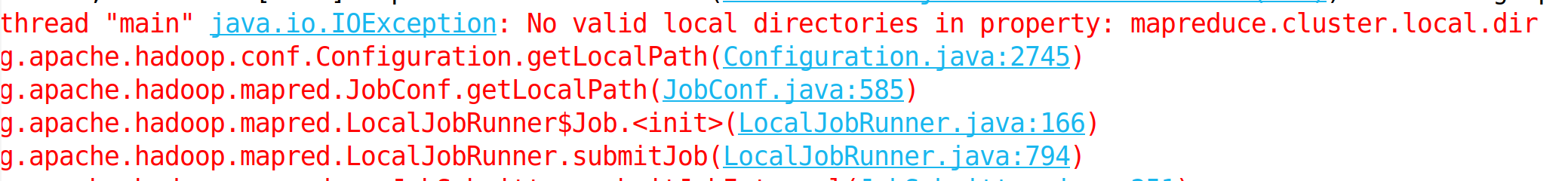


查看input，上传成功

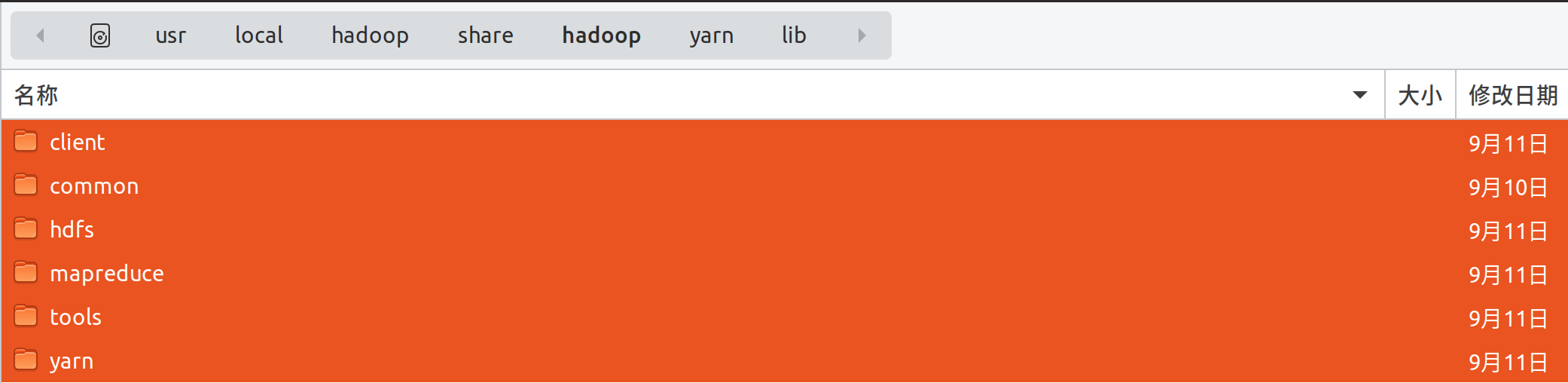


运行代码：

发现报错：

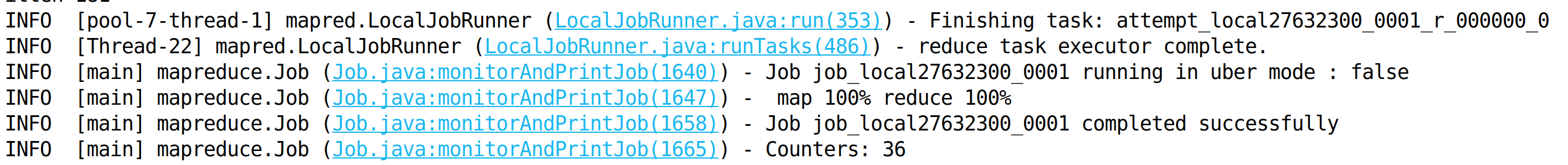


原因是包没导全，mapreduce需要的包不仅仅需要当初hadoop基础操作的那些包，需要导入下列目录和其lib子目录下的所有包

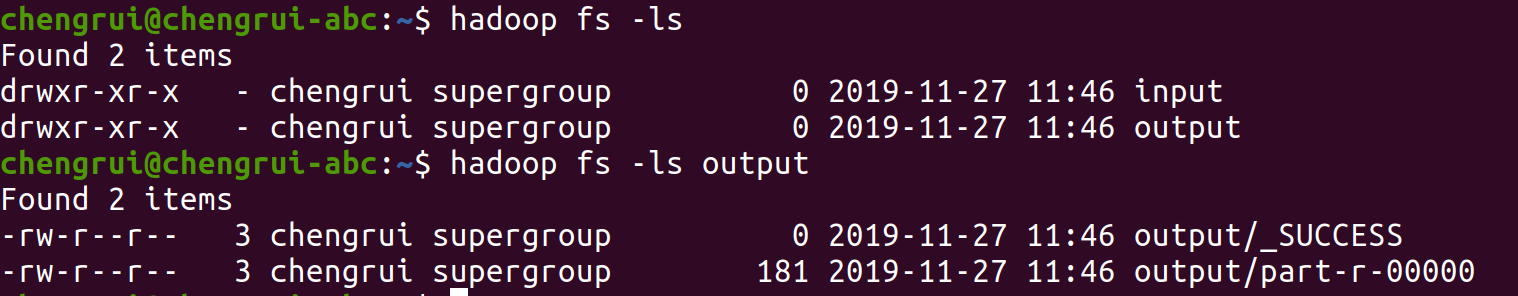


导入后，再次执行代码：

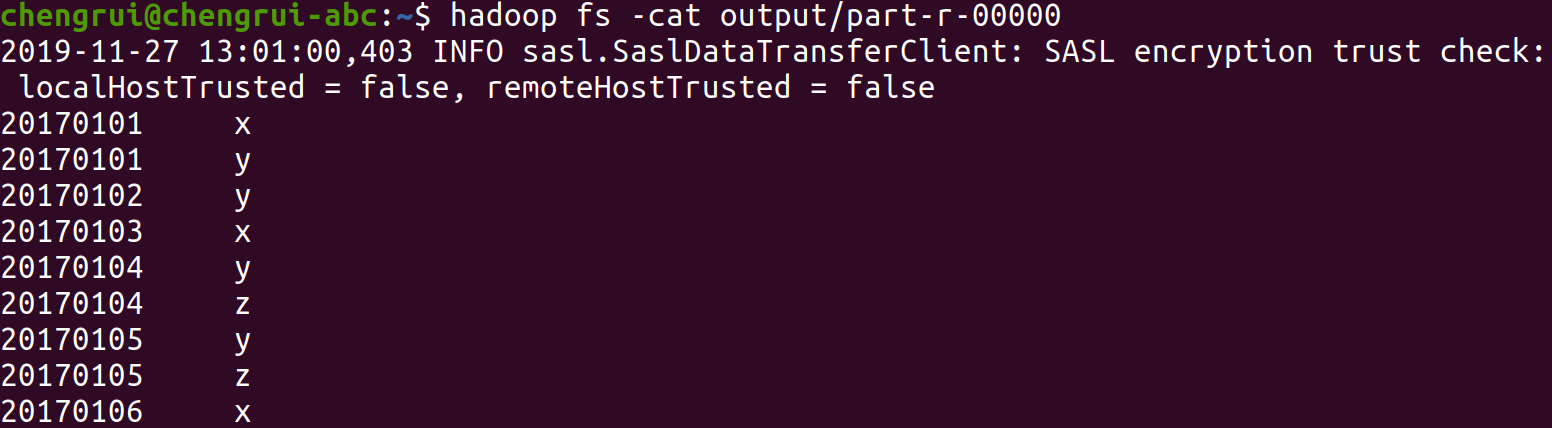
成功，由于控制台输出很多数据，于是只截取部分



执行hadoop fs -ls，发现生成了一个output目录，目录下有两个文件。



使用cat查看文件内容，发现\_SUCCESS为空，而实验结果在part-r-00000中。



## 2.2 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。下面是输入文件和输出文件的一个样例供参考。

输入文件1的样例如下：

|  |
| --- |
| 33  37  12  40 |

输入文件2的样例如下：

|  |
| --- |
| 4  16  39  5 |
|  |

输入文件3的样例如下：

|  |
| --- |
| 1  45  25 |

根据输入文件1、2和3得到的输出文件如下：

|  |
| --- |
| 1 1  2 4  3 5  4 12  5 16  6 25  7 33  8 37  9 39  10 40  11 45 |

### **2.2.1 问题分析及思路**

问题要求是排序，在MapReduce过程中是有自动排序的，具体过程如下：

首先将读入的数据在Map中转化成IntWritable型，然后作为Key值输出（value任意）；其次需要重写partition类，保证整体有序，具体做法是用输入数据的最大值除以系统partition数据的商作为分割数据的边界增量，也就是说分割数据的边界的最大值为此商的1倍、2倍至numPartition-1倍，这样就能保证执行partition后的数据是整体有序的；然后Reduce获得<Key,value-list>之后，根据value-list中元素的个数决定输出次数。

### **2.2.2 代码**

**import** java.io.IOException;

**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.Partitioner;

**import** org.apache.hadoop.mapreduce.Reducer;

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

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

**public** **class** Test2 {

//map 函数

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

**private** **static** IntWritable *data* = **new** IntWritable();

//hadoop接口，等同Integer整型变量。

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

String text = value.toString();//一行内容

*data*.set(Integer.*parseInt*(text));//将内容转化为整型

context.write(*data*, **new** IntWritable(1));//输出key为数据，value为1

}

}

//reduce 函数

//输入的 key（即数字）复制到输出的 value 上，然后根据输入的 value-list中元素的个数决定 key 的输出次数

**public** **static** **class** Reduce **extends** Reducer<IntWritable, IntWritable, IntWritable, IntWritable> {

**private** **static** IntWritable *line\_num* = **new** IntWritable(1);

**public** **void** reduce(IntWritable key, Iterable<IntWritable> values, Context context)//输入，key为数字，value为list(用来计算个数)

**throws** IOException, InterruptedException {

**for** (IntWritable val : values) {//value-list元素个数决定 key 的输出次数

context.write(*line\_num*, key);//（行数，数字），输出多次

*line\_num* = **new** IntWritable(*line\_num*.get() + 1);//行数+1

}

}

}

// Partition 函数，根据输入数据的最大值和 MapReduce 框架中Partition的数量

// 将输入数据按照大小分块的边界，然后根据输入数值和边界的关系返回对应的 Partiton ID

// 简单来讲就是按int型重新Partition分块

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

**public** **int** getPartition(IntWritable key, IntWritable value, **int** num\_Partition) {

**int** Maxnumber = 65223;// int 型的最大数值

**int** bound = Maxnumber / num\_Partition + 1;//分界

**int** keynumber = key.get();//数据

**for** (**int** i = 0; i < num\_Partition; i++) {//找分界

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

**return** i;//返回对应的 Partiton ID

}

}

**return** -1;

}

}

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

Configuration conf = **new** Configuration();

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

String[] otherArgs = **new** String[] { "/user/chengrui/input","/user/chengrui/output" }; // 直接设置输入参数

**if** (otherArgs.length != 2) {

System.***err***.println("Usage: wordcount <in><out>");

System.*exit*(2);

}

Job job = Job.*getInstance*(conf, "合并排序");

job.setJarByClass(Test2.**class**);

job.setMapperClass(Map.**class**);

job.setReducerClass(Reduce.**class**);

job.setPartitionerClass(Partition.**class**);

job.setOutputKeyClass(IntWritable.**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);

}

}

### **2.2.3 实验过程**

**文件内容：**

1.txt：

|  |
| --- |
| 33  37  12  40 |

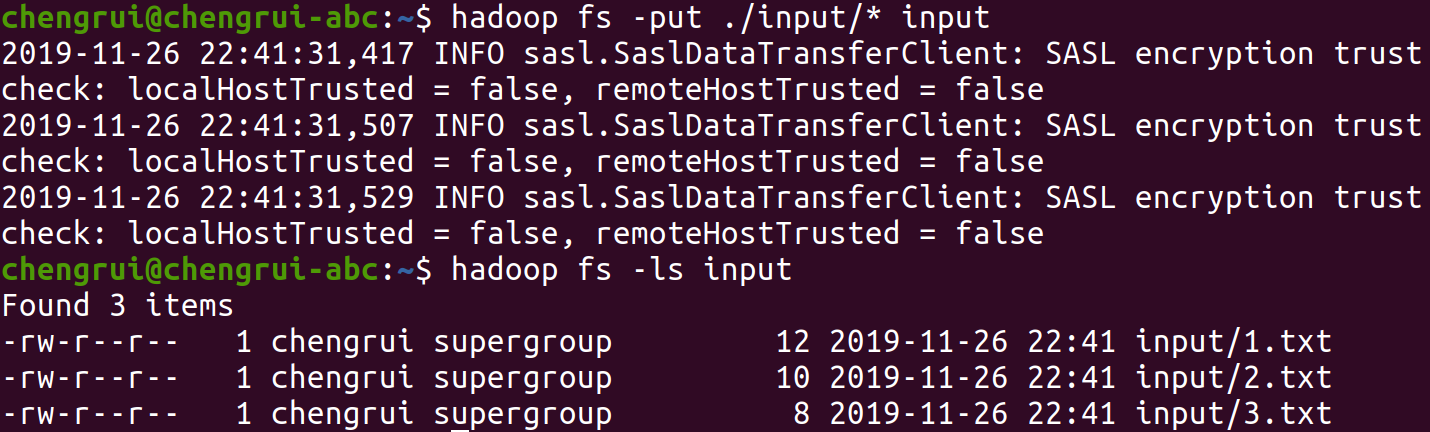
2.txt：

|  |
| --- |
| 4  16  39 |
| 5 |

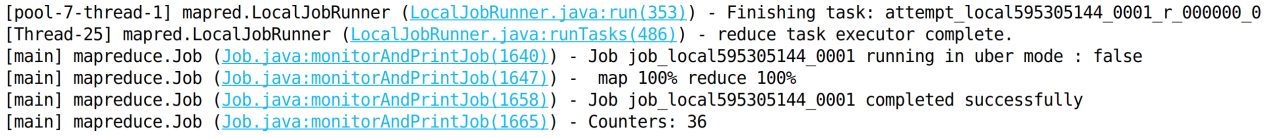
3.txt:

|  |
| --- |
| 1  45  25 |

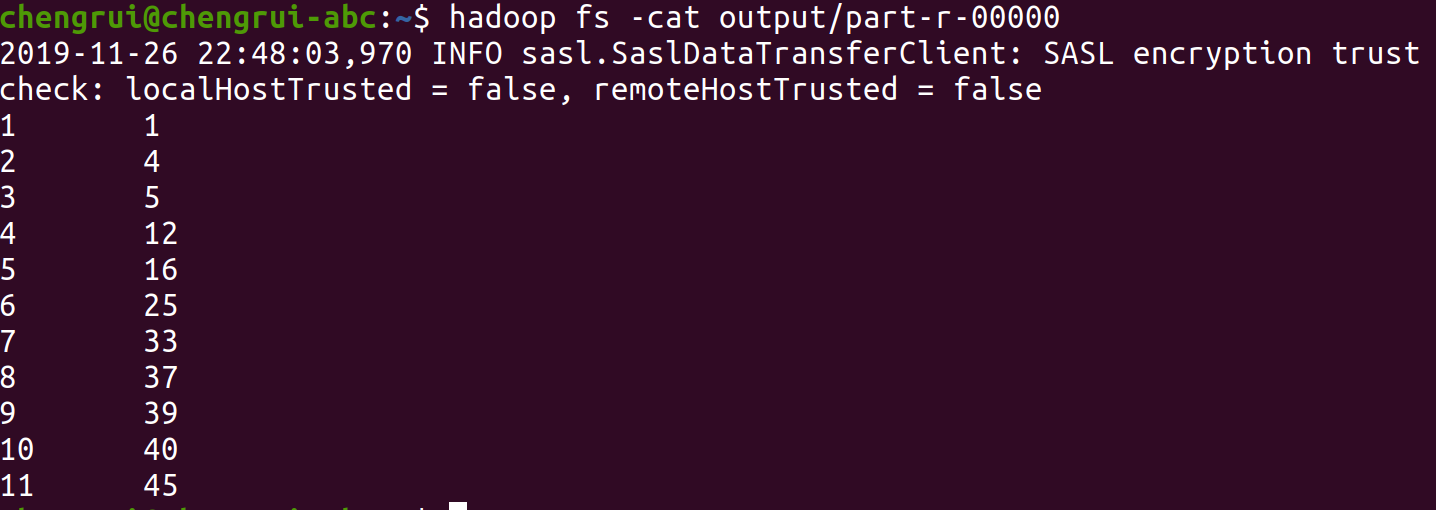
**上传输入文件:**



**运行代码，控制台部分截图：**



**查看结果，输出文件的内容如下**





## 2.3 对给定的表格进行信息挖掘

下面给出一个child-parent的表格，要求挖掘其中的父子辈关系，给出祖孙辈关系的表格。

输入文件内容如下：

|  |
| --- |
| child parent  Steven Lucy  Steven Jack  Jone Lucy  Jone Jack  Lucy Mary  Lucy Frank  Jack Alice  Jack Jesse  David Alice  David Jesse  Philip David  Philip Alma  Mark David  Mark Alma |

输出文件内容如下：

|  |
| --- |
| grandchild grandparent  Steven Alice  Steven Jesse  Jone Alice  Jone Jesse  Steven Mary  Steven Frank  Jone Mary  Jone Frank  Philip Alice  Philip Jesse  Mark Alice  Mark Jesse |

### **2.3.1 问题分析及思路**

问题是要数据挖掘，从父子关系表中，找出爷孙关系。那么，首先，map阶段，把数据正反各输出一次，即把每一行的key-value输入，同时也把value-key输入，这样经过shuffle过程，可以得到key后面的value就为key的父辈和子辈，然后其父辈和子辈就自然构成了祖孙关系。为了标记哪些是孙子辈，哪些是爷爷辈，在Map阶段加上前缀。Reduce阶段根据前缀进行分类,把祖辈和孙辈分别存在不同数组中，然后输出。

例如：

Steven Jack

Jack Alice

正反输出一次，加上前缀：

Steven +Jack

Jack +Alice

Jack -Steven

Alice -Jack

可见+后的和-后的构成爷孙关系

### **2.3.3 代码**

**import** java.io.IOException;

**import** java.util.\*;

**import** org.apache.hadoop.conf.Configuration;

**import** org.apache.hadoop.fs.Path;

**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;

**public** **class** Test3 {

//Map 将输入文件按照空格分割成 child 和 parent

//正反各输出一次，小辈加上前缀”-“，长辈加上前缀”+”

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

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

String child\_name = **new** String();

String parent\_name = **new** String();

String line = value.toString();

**int** i = 0;

**while** (line.charAt(i) != ' ') {//记录‘ ’的位置

i++;

}

String[] values = { line.substring(0, i), line.substring(i + 1) };

//child ，parent

**if** (values[0].compareTo("child") != 0) {//第二行开始，第一行是表头

child\_name = values[0];

parent\_name = values[1];

context.write(**new** Text(parent\_name), **new** Text("-" + child\_name));//正反各输出一次

context.write(**new** Text(child\_name), **new** Text("+" + parent\_name));//父辈前加+

}

}

}

//把同一个key的儿子和父辈输出，即为祖关系

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

**private** **static** **int** *linenum* = 0;

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

**if** (*linenum* == 0) {

context.write(**new** Text("grandchild"), **new** Text("grandparent"));

++*linenum*;

}

ArrayList<Text> grandChild = **new** ArrayList<Text>();

ArrayList<Text> grandParent = **new** ArrayList<Text>();

**for** (Text val : values) {

String s = val.toString();

**if** (s.startsWith("-")) {//孙子

grandChild.add(**new** Text(s.substring(1)));

} **else** {//祖辈

grandParent.add(**new** Text(s.substring(1)));

}

}

**for** (Text text1 : grandChild) {

**for** (Text text2 : grandParent) {

context.write(text1, text2);

}

}

}

}

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

Configuration conf = **new** Configuration();

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

String[] otherArgs = **new** String[] { "/user/chengrui/input", "/user/chengrui/output" };

**if** (otherArgs.length != 2) {

System.***err***.println("Usage: wordcount <in><out>");

System.*exit*(2);

}

Job job = Job.*getInstance*(conf, "relationship");

job.setJarByClass(Test3.**class**);

job.setMapperClass(Map.**class**);

job.setReducerClass(Reduce.**class**);

job.setOutputKeyClass(Text.**class**);

job.setOutputValueClass(Text.**class**);

FileInputFormat.*addInputPath*(job, **new** Path(otherArgs[0]));

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

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

}

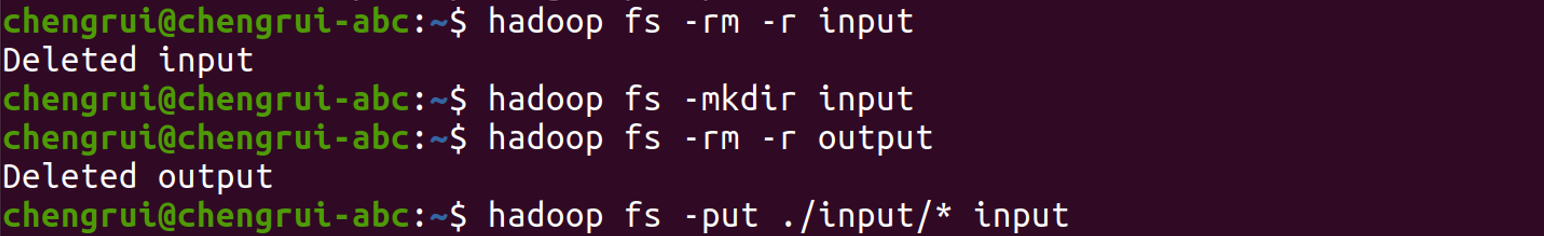
}

### **2.3.3 实验过程**

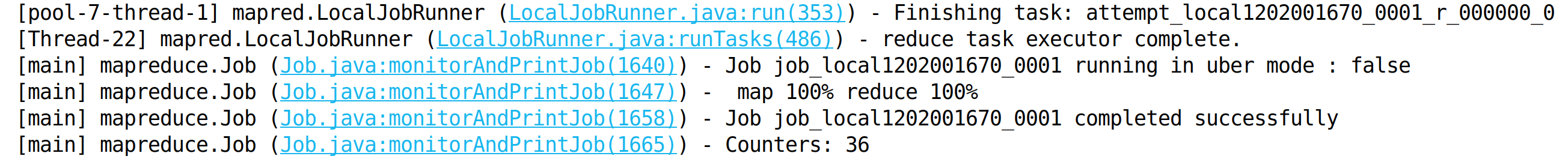
文件内容：

|  |
| --- |
| child parent  Steven Lucy  Steven Jack  Jone Lucy  Jone Jack  Lucy Mary  Lucy Frank  Jack Alice  Jack Jesse  David Alice  David Jesse  Philip David  Philip Alma  Mark David  Mark Alma |

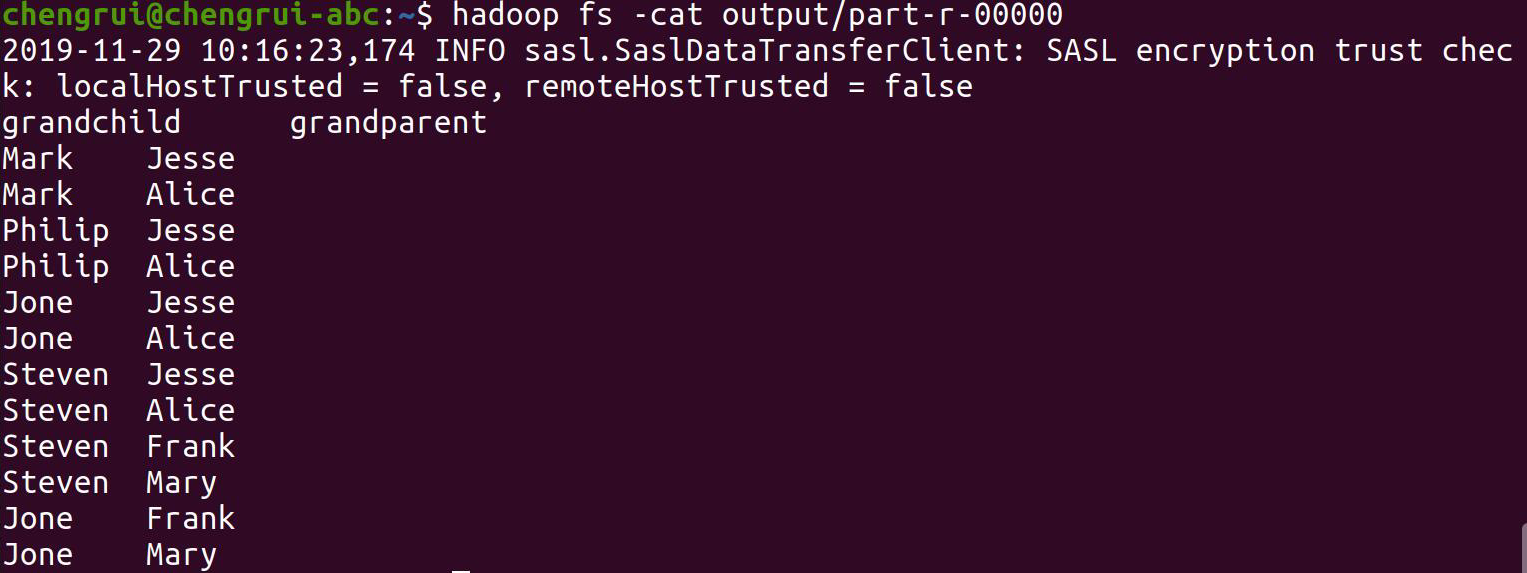
删除原有的输入输出目录，重新创建输入目录，把输入文件推上去。



运行代码：



查看结果：



# 3. 实验总结

## 3.1 编程实现文件合并和去重操作

（1）为了让输入文件中的数据只出现一次，所以在map过程中将输出的Key设置为数据，然后value设置为空；这样shuffle过程中合并完，数据就只出现一次，即为key；最后Reduce过程中直接输出key，就可以做到去重。具体过程在代码注释里有详细写到。

## 3.2 编写程序实现对输入文件的排序

（1）MapReduce过程中的默认排序，是按照Key值进行排序，如果Key为封装int的IntWritable类型，那么MapReduce按照数字大小对Key排序；如果Key为封装String的Text类型，那么MapReduce按照字典顺序对字符串排序。

（2）具体过程

首先将读入的数据在Map中转化成IntWritable型，然后作为Key值输出（value任意）；其次需要重写partition类，保证整体有序，具体做法是用输入数据的最大值除以系统partition数据的商作为分割数据的边界增量，也就是说分割数据的边界的最大值为此商的1倍、2倍至numPartition-1倍，这样就能保证执行partition后的数据是整体有序的；然后Reduce获得<Key,value-list>之后，根据value-list中元素的个数决定输出次数。

（3）为什么需要重写partition类？

得到map产生的记录后， hadoop默认是根据散列值来派发，但是实际中，这并不能很高效或者按照我们要求的去执行任务。例如，经过partition处理后，一个节点的reducer分配到了20条记录，另一个却分配到了10W万条，这种情况效率就不太理想。另一方面，假设有两个reducer，他们处理数据后都是局部有序，然而不是全局有序的，所以需要我们自己定制partition来选择reducer。

（4）如何自定义partition？

只要自定义一个类，并且继承Partitioner类，重写其getPartition方法就好了，在使用的时候通过调用Job的setPartitionerClass指定一下即可。

（5）如何使用mapreduce来做全排序？

①使用一个partition，因为一个partition对应一个reduce的task，然而reduce的输入本来就是对key有序的，所以很自然地就产生了一个全排序文件。但是这种方法在处理大型文件时效率太低了。

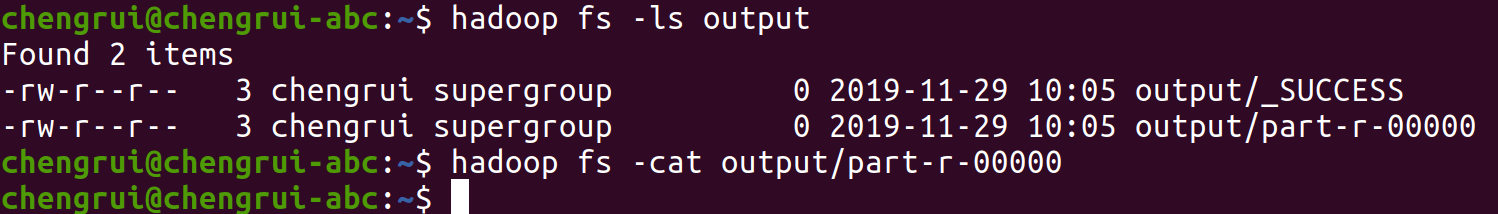
②如果是分多个partition，只要确保partition是有序的就行了。首先创建一系列排好序的文件；其次，串联这些文件；最后得到一个全局有序的文件。例如，这题是要按int类型分，int最大值为65223，所以就把它均分，每个部分对应一个partition。

## 3.3 对给定的表格进行信息挖掘

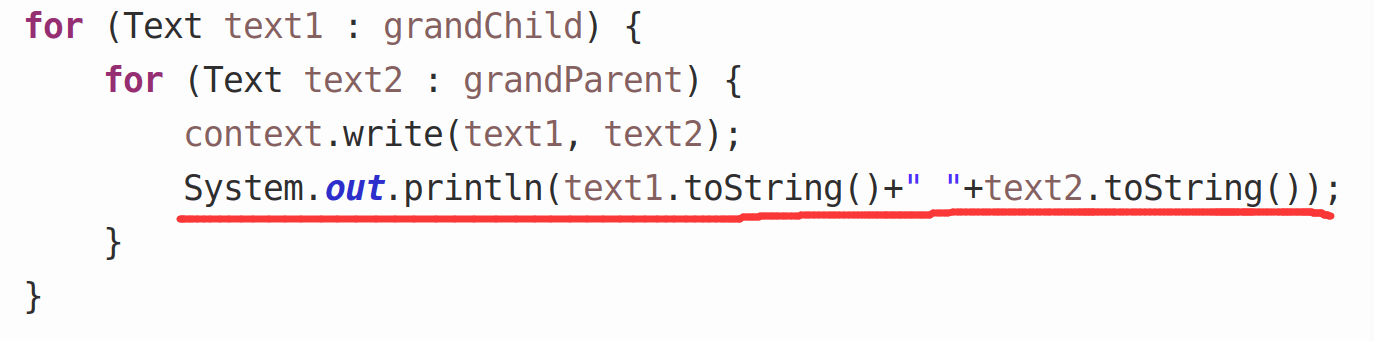
（1）问题是要数据挖掘，从父子关系表中，找出爷孙关系。那么，首先，map阶段，把数据正反各输出一次，即把每一行的key-value输入，同时也把value-key输入，这样经过shuffle过程，可以得到key后面的value就为key的父辈和子辈，然后其父辈和子辈就自然构成了祖孙关系。为了标记哪些是孙子辈，哪些是爷爷辈，在Map阶段加上前缀。Reduce阶段根据前缀进行分类,把祖辈和孙辈分别储存在不同数组中，然后输出。

（2）关于job.setCombinerClass和job.setReducerClass

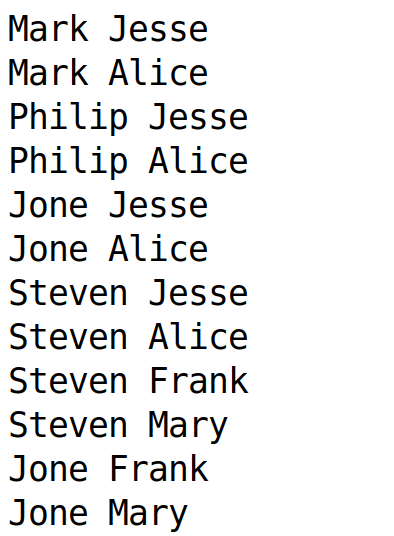
第一次运行完后，发现输出文件为空



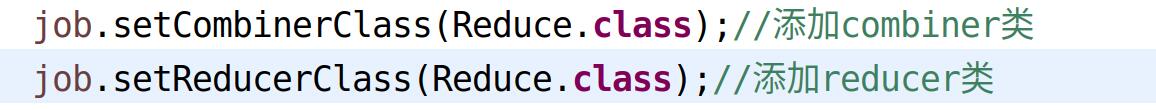
在reduce函数的输出下加了一条控制台输出用作测试

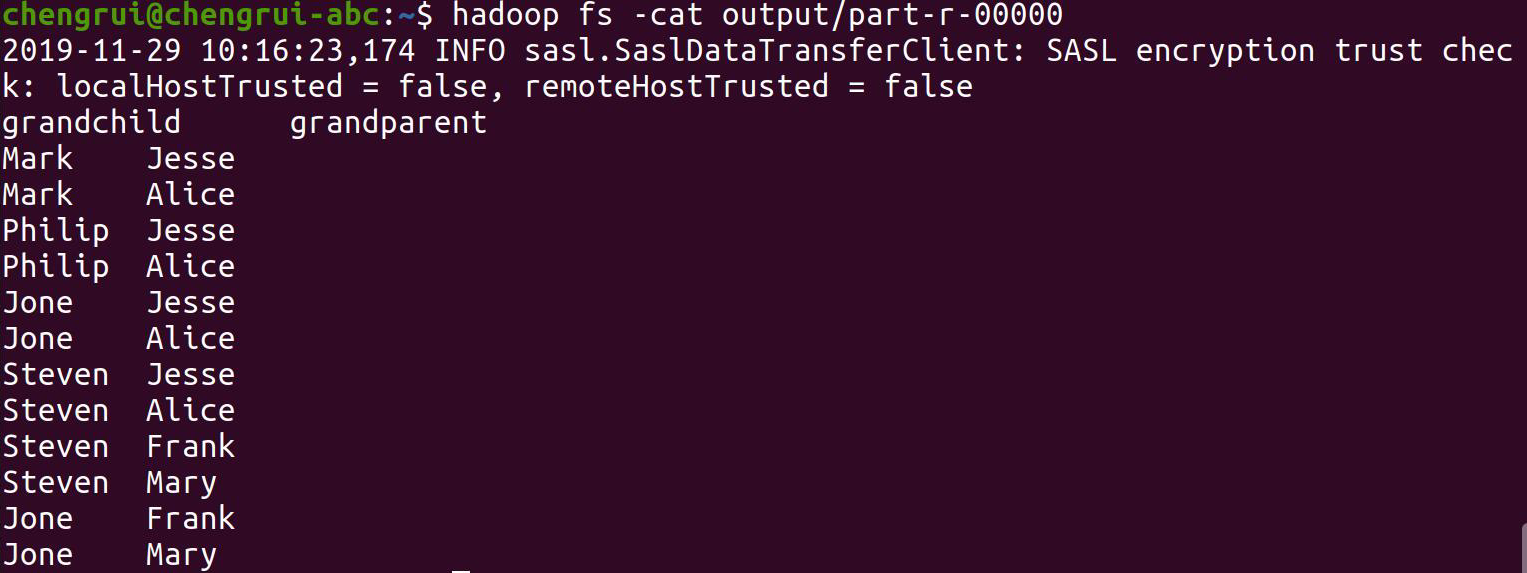


运行后发现，是有成功执行reduce的



然后经过一些尝试，我把job.setCombinerClass删了，发现成功。



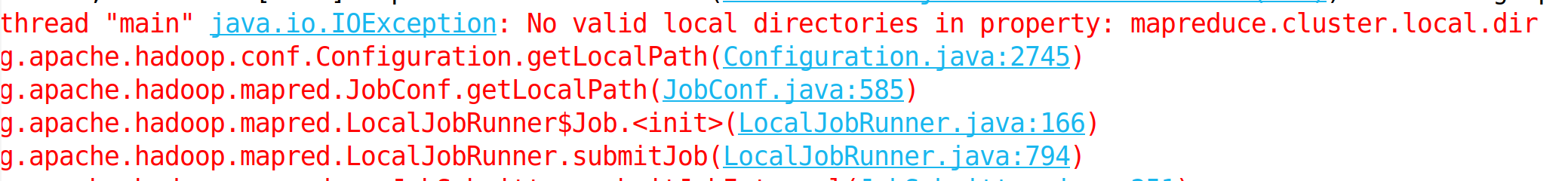


经过查阅资料，原因是：job.setCombinerClass和job.setReducerClass重复设置，会造成reduce 输出的key--value会当成map阶段的输出key--value再次输入到reduce中进行处理。

所以执行的流程为map(输出key--value)---->setCombinerClass(reduce)(输出key-value)---->reduce(key---value)  
所以我们reduce执行了两次，第一次执行Combiner reduce的输入为map的输出，第二次执行reduece的输入为第一次执行Combiner reduce的输出。然后经过第一次reduce后的数据，在第二次由于没有+和-，于是第二次reduce执行完没有输出，导致输出文件为空。

## 3.4 其他

使用eclipse运行mapreduce代码，需要导入足够的hadoop包。否则会报错。



mapreduce需要的包不仅仅需要当初hadoop基础操作的那些包，需要导入下列目录和其lib子目录下的所有包。

