# 七．MapReduce实战

## 7.1 WordCount案例

### 7.1.1 需求1：统计一堆文件中单词出现的个数

0）需求：在一堆给定的文本文件中统计输出每一个单词出现的总次数

1）数据准备：



2）分析

按照mapreduce编程规范，分别编写Mapper，Reducer，Driver。



3）编写程序

（1）编写mapper类

|  |
| --- |
| package com.bigdata.mapreduce;  import java.io.IOException;  import org.apache.hadoop.io.IntWritable;  import org.apache.hadoop.io.LongWritable;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Mapper;  public class WordcountMapper extends Mapper<LongWritable, Text, Text, IntWritable>{    Text k = new Text();  IntWritable v = new IntWritable(1);    @Override  protected void map(LongWritable key, Text value, Context context)  throws IOException, InterruptedException {    // 1 获取一行  String line = value.toString();    // 2 切割  String[] words = line.split(" ");    // 3 输出  for (String word : words) {    k.set(word);  context.write(k, v);  }  }  } |

（2）编写reducer类

|  |
| --- |
| package com.bigdata.mapreduce.wordcount;  import java.io.IOException;  import org.apache.hadoop.io.IntWritable;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Reducer;  public class WordcountReducer extends Reducer<Text, IntWritable, Text, IntWritable>{  @Override  protected void reduce(Text key, Iterable<IntWritable> value,  Context context) throws IOException, InterruptedException {    // 1 累加求和  int sum = 0;  for (IntWritable count : value) {  sum += count.get();  }    // 2 输出  context.write(key, new IntWritable(sum));  }  } |

（3）编写驱动类

|  |
| --- |
| package com.bigdata.mapreduce.wordcount;  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.lib.input.FileInputFormat;  import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;  public class WordcountDriver {  public static void main(String[] args) throws IOException, ClassNotFoundException, InterruptedException {  // 1 获取配置信息  Configuration configuration = new Configuration();  Job job = Job.getInstance(configuration);  // 2 设置jar加载路径  job.setJarByClass(WordcountDriver.class);  // 3 设置map和Reduce类  job.setMapperClass(WordcountMapper.class);  job.setReducerClass(WordcountReducer.class);  // 4 设置map输出  job.setMapOutputKeyClass(Text.class);  job.setMapOutputValueClass(IntWritable.class);  // 5 设置Reduce输出  job.setOutputKeyClass(Text.class);  job.setOutputValueClass(IntWritable.class);    // 6 设置输入和输出路径  FileInputFormat.setInputPaths(job, new Path(args[0]));  FileOutputFormat.setOutputPath(job, new Path(args[1]));  // 7 提交  boolean result = job.waitForCompletion(true);  System.exit(result ? 0 : 1);  }  } |

4）集群上测试

（1）将程序打成jar包，然后拷贝到hadoop集群中。

（2）启动hadoop集群

（3）执行wordcount程序

[bigdata@hadoop102 software]$ hadoop jar wc.jar com.bigdata.wordcount.WordcountDriver /user/bigdata/input /user/bigdata/output1

5）本地测试

（1）在windows环境上配置HADOOP\_HOME环境变量。

（2）在eclipse上运行程序

（3）注意：如果eclipse打印不出日志，在控制台上只显示

|  |
| --- |
| 1.log4j:WARN No appenders could be found for logger (org.apache.hadoop.util.Shell).  2.log4j:WARN Please initialize the log4j system properly.  3.log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more info. |

需要在项目的src目录下，新建一个文件，命名为“log4j.properties”，在文件中填入

|  |
| --- |
| log4j.rootLogger=INFO, stdout  log4j.appender.stdout=org.apache.log4j.ConsoleAppender  log4j.appender.stdout.layout=org.apache.log4j.PatternLayout  log4j.appender.stdout.layout.ConversionPattern=%d %p [%c] - %m%n  log4j.appender.logfile=org.apache.log4j.FileAppender  log4j.appender.logfile.File=target/spring.log  log4j.appender.logfile.layout=org.apache.log4j.PatternLayout  log4j.appender.logfile.layout.ConversionPattern=%d %p [%c] - %m%n |

### 7.1.2 需求3：对每一个maptask的输出局部汇总（Combiner）

0）需求：统计过程中对每一个maptask的输出进行局部汇总，以减小网络传输量即采用Combiner功能。



1）数据准备：



**方案一**

1）增加一个WordcountCombiner类继承Reducer

|  |
| --- |
| **package** com.bigdata.mr.combiner;  **import** java.io.IOException;  **import** org.apache.hadoop.io.IntWritable;  **import** org.apache.hadoop.io.Text;  **import** org.apache.hadoop.mapreduce.Reducer;  **public** **class** WordcountCombiner **extends** Reducer<Text, IntWritable, Text, IntWritable>{  @Override  **protected** **void** reduce(Text key, Iterable<IntWritable> values,  Context context) **throws** IOException, InterruptedException {  // 1 汇总  **int** count = 0;  **for**(IntWritable v :values){  count += v.get();  }  // 2 写出  context.write(key, **new** IntWritable(count));  }  } |

2）在WordcountDriver驱动类中指定combiner

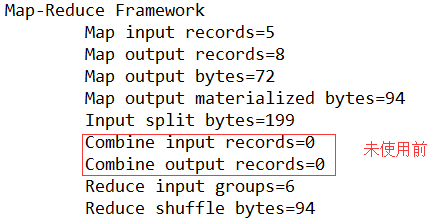
|  |
| --- |
| // 9 指定需要使用combiner，以及用哪个类作为combiner的逻辑  job.setCombinerClass(WordcountCombiner.class); |

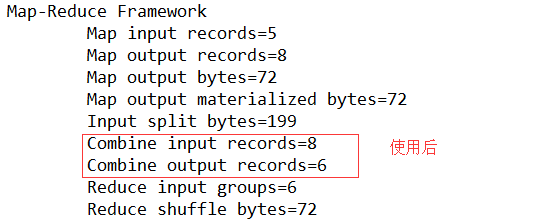
**方案二**

1）将WordcountReducer作为combiner在WordcountDriver驱动类中指定

|  |
| --- |
| // 指定需要使用combiner，以及用哪个类作为combiner的逻辑  job.setCombinerClass(WordcountReducer.**class**); |

运行程序





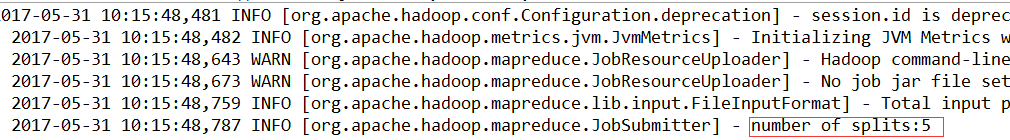
### 7.1.4 需求4：大量小文件的切片优化（CombineTextInputFormat）

0）需求：将输入的大量小文件合并成一个切片统一处理。

1）输入数据：准备5个小文件

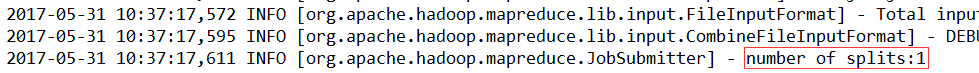
2）实现过程

（1）不做任何处理，运行需求1中的wordcount程序，观察切片个数为5



（2）在WordcountDriver中增加如下代码，运行程序，并观察运行的切片个数为1

|  |
| --- |
| // 如果不设置InputFormat，它默认用的是TextInputFormat.class  job.setInputFormatClass(CombineTextInputFormat.**class**);  CombineTextInputFormat.*setMaxInputSplitSize*(job, 4194304);// 4m  CombineTextInputFormat.*setMinInputSplitSize*(job, 2097152);// 2m |



## 7.2 流量汇总案例

### 7.2.1 需求1：统计手机号耗费的总上行流量、下行流量、总流量（序列化）

1）需求：

统计每一个手机号耗费的总上行流量、下行流量、总流量

2）数据准备



输入数据格式：

|  |
| --- |
| 1363157993055 13560436666 C4-17-FE-BA-DE-D9:CMCC 120.196.100.99 18 15 1116 954 200  手机号码 上行流量 下行流量 |

输出数据格式

|  |
| --- |
| 1356·0436666 1116 954 2070  手机号码 上行流量 下行流量 总流量 |

3）分析

基本思路：

Map阶段：

（1）读取一行数据，切分字段

（2）抽取手机号、上行流量、下行流量

（3）以手机号为key，bean对象为value输出，即context.write(手机号,bean);

Reduce阶段：

（1）累加上行流量和下行流量得到总流量。

（2）实现自定义的bean来封装流量信息，并将bean作为map输出的key来传输

（3）MR程序在处理数据的过程中会对数据排序(map输出的kv对传输到reduce之前，会排序)，排序的依据是map输出的key

所以，我们如果要实现自己需要的排序规则，则可以考虑将排序因素放到key中，让key实现接口：WritableComparable。

然后重写key的compareTo方法。

4）编写mapreduce程序

（1）编写流量统计的bean对象

|  |
| --- |
| package com.bigdata.mapreduce.flowsum;  import java.io.DataInput;  import java.io.DataOutput;  import java.io.IOException;  import org.apache.hadoop.io.Writable;  // 1 实现writable接口  public class FlowBean implements Writable{  private long upFlow ;  private long downFlow;  private long sumFlow;    //2 反序列化时，需要反射调用空参构造函数，所以必须有  public FlowBean() {  super();  }  public FlowBean(long upFlow, long downFlow) {  super();  this.upFlow = upFlow;  this.downFlow = downFlow;  this.sumFlow = upFlow + downFlow;  }    //3 写序列化方法  @Override  public void write(DataOutput out) throws IOException {  out.writeLong(upFlow);  out.writeLong(downFlow);  out.writeLong(sumFlow);  }    //4 反序列化方法  //5 反序列化方法读顺序必须和写序列化方法的写顺序必须一致  @Override  public void readFields(DataInput in) throws IOException {  this.upFlow = in.readLong();  this.downFlow = in.readLong();  this.sumFlow = in.readLong();  }  // 6 编写toString方法，方便后续打印到文本  @Override  public String toString() {  return upFlow + "\t" + downFlow + "\t" + sumFlow;  }  public long getUpFlow() {  return upFlow;  }  public void setUpFlow(long upFlow) {  this.upFlow = upFlow;  }  public long getDownFlow() {  return downFlow;  }  public void setDownFlow(long downFlow) {  this.downFlow = downFlow;  }  public long getSumFlow() {  return sumFlow;  }  public void setSumFlow(long sumFlow) {  this.sumFlow = sumFlow;  }  } |

（2）编写mapper

|  |
| --- |
| package com.bigdata.mapreduce.flowsum;  import java.io.IOException;  import org.apache.hadoop.io.LongWritable;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Mapper;  public class FlowCountMapper extends Mapper<LongWritable, Text, Text, FlowBean>{    FlowBean v = new FlowBean();  Text k = new Text();    @Override  protected void map(LongWritable key, Text value, Context context)  throws IOException, InterruptedException {    // 1 获取一行  String line = value.toString();    // 2 切割字段  String[] fields = line.split("\t");    // 3 封装对象  // 取出手机号码  String phoneNum = fields[1];  // 取出上行流量和下行流量  long upFlow = Long.parseLong(fields[fields.length - 3]);  long downFlow = Long.parseLong(fields[fields.length - 2]);    v.set(downFlow, upFlow);    // 4 写出  context.write(new Text(phoneNum), new FlowBean(upFlow, downFlow));  }  } |

（3）编写reducer

|  |
| --- |
| package com.bigdata.mapreduce.flowsum;  import java.io.IOException;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Reducer;  public class FlowCountReducer extends Reducer<Text, FlowBean, Text, FlowBean> {  @Override  protected void reduce(Text key, Iterable<FlowBean> values, Context context)  throws IOException, InterruptedException {  long sum\_upFlow = 0;  long sum\_downFlow = 0;  // 1 遍历所用bean，将其中的上行流量，下行流量分别累加  for (FlowBean flowBean : values) {  sum\_upFlow += flowBean.getSumFlow();  sum\_downFlow += flowBean.getDownFlow();  }  // 2 封装对象  FlowBean resultBean = new FlowBean(sum\_upFlow, sum\_downFlow);    // 3 写出  context.write(key, resultBean);  }  } |

（4）编写驱动

|  |
| --- |
| package com.bigdata.mapreduce.flowsum;  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.lib.input.FileInputFormat;  import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;  public class FlowsumDriver {  public static void main(String[] args) throws IllegalArgumentException, IOException, ClassNotFoundException, InterruptedException {    // 1 获取配置信息，或者job对象实例  Configuration configuration = new Configuration();  Job job = Job.getInstance(configuration);  // 6 指定本程序的jar包所在的本地路径  job.setJarByClass(FlowsumDriver.class);  // 2 指定本业务job要使用的mapper/Reducer业务类  job.setMapperClass(FlowCountMapper.class);  job.setReducerClass(FlowCountReducer.class);  // 3 指定mapper输出数据的kv类型  job.setMapOutputKeyClass(Text.class);  job.setMapOutputValueClass(FlowBean.class);  // 4 指定最终输出的数据的kv类型  job.setOutputKeyClass(Text.class);  job.setOutputValueClass(FlowBean.class);    // 5 指定job的输入原始文件所在目录  FileInputFormat.setInputPaths(job, new Path(args[0]));  FileOutputFormat.setOutputPath(job, new Path(args[1]));  // 7 将job中配置的相关参数，以及job所用的java类所在的jar包， 提交给yarn去运行  boolean result = job.waitForCompletion(true);  System.exit(result ? 0 : 1);  }  } |

### 7.2.2 需求2：将统计结果按照手机归属地不同省份输出到不同文件中（Partitioner）

0）需求：将统计结果按照手机归属地不同省份输出到不同文件中（分区）

1）数据准备



2）分析

（1）Mapreduce中会将map输出的kv对，按照相同key分组，然后分发给不同的reducetask。默认的分发规则为：根据key的hashcode%reducetask数来分发

（2）如果要按照我们自己的需求进行分组，则需要改写数据分发（分组）组件Partitioner

自定义一个CustomPartitioner继承抽象类：Partitioner

（3）在job驱动中，设置自定义partitioner： job.setPartitionerClass(CustomPartitioner.class)

3）在需求1的基础上，增加一个分区类

|  |
| --- |
| package com.bigdata.mapreduce.flowsum;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Partitioner;  public class ProvincePartitioner extends Partitioner<Text, FlowBean> {  @Override  public int getPartition(Text key, FlowBean value, int numPartitions) {  // 1 获取电话号码的前三位  String preNum = key.toString().substring(0, 3);    int partition = 4;    // 2 判断是哪个省  if ("136".equals(preNum)) {  partition = 0;  }else if ("137".equals(preNum)) {  partition = 1;  }else if ("138".equals(preNum)) {  partition = 2;  }else if ("139".equals(preNum)) {  partition = 3;  }  return partition;  }  } |

2）在驱动函数中增加自定义数据分区设置和reduce task设置

|  |
| --- |
| package com.bigdata.mapreduce.flowsum;  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.lib.input.FileInputFormat;  import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;  public class FlowsumDriver {  public static void main(String[] args) throws IllegalArgumentException, IOException, ClassNotFoundException, InterruptedException {    // 1 获取配置信息，或者job对象实例  Configuration configuration = new Configuration();  Job job = Job.getInstance(configuration);  // 6 指定本程序的jar包所在的本地路径  job.setJarByClass(FlowsumDriver.class);  // 2 指定本业务job要使用的mapper/Reducer业务类  job.setMapperClass(FlowCountMapper.class);  job.setReducerClass(FlowCountReducer.class);  // 3 指定mapper输出数据的kv类型  job.setMapOutputKeyClass(Text.class);  job.setMapOutputValueClass(FlowBean.class);  // 4 指定最终输出的数据的kv类型  job.setOutputKeyClass(Text.class);  job.setOutputValueClass(FlowBean.class);  // 8 指定自定义数据分区  job.setPartitionerClass(ProvincePartitioner.class);  // 9 同时指定相应数量的reduce task  job.setNumReduceTasks(5);    // 5 指定job的输入原始文件所在目录  FileInputFormat.setInputPaths(job, new Path(args[0]));  FileOutputFormat.setOutputPath(job, new Path(args[1]));  // 7 将job中配置的相关参数，以及job所用的java类所在的jar包， 提交给yarn去运行  boolean result = job.waitForCompletion(true);  System.exit(result ? 0 : 1);  }  } |

### 7.2.3 需求3：将统计结果按照总流量倒序排序（全排序）

排序的分类：

（1）部分排序：

MapReduce根据输入记录的键对数据集排序。保证输出的每个文件内部排序。

（2）全排序：

如何用Hadoop产生一个全局排序的文件？最简单的方法是使用一个分区。但该方法在处理大型文件时效率极低，因为一台机器必须处理所有输出文件，从而完全丧失了MapReduce所提供的并行架构。

替代方案：首先创建一系列排好序的文件；其次，串联这些文件；最后，生成一个全局排序的文件。主要思路是使用一个分区来描述输出的全局排序。例如：可以为上述文件创建3个分区，在第一分区中，记录的单词首字母a-g，第二分区记录单词首字母h-n, 第三分区记录单词首字母o-z。

0）需求

根据需求1产生的结果再次对总流量进行排序。

1）数据准备



2）分析

（1）把程序分两步走，第一步正常统计总流量，第二步再把结果进行排序

（2）context.write(总流量，手机号)

（3）FlowBean实现WritableComparable接口重写compareTo方法

|  |
| --- |
| @Override  **public** **int** compareTo(FlowBean o) {  // 倒序排列，从大到小  **return** **this**.sumFlow > o.getSumFlow() ? -1 : 1;  } |

3）代码实现

（1）FlowBean对象在在需求1基础上增加了比较功能

|  |
| --- |
| **package** com.bigdata.mapreduce.sort;  **import** java.io.DataInput;  **import** java.io.DataOutput;  **import** java.io.IOException;  **import** org.apache.hadoop.io.WritableComparable;  **public** **class** FlowBean **implements** WritableComparable<FlowBean> {  **private** **long** upFlow;  **private** **long** downFlow;  **private** **long** sumFlow;  // 反序列化时，需要反射调用空参构造函数，所以必须有  **public** FlowBean() {  **super**();  }  **public** FlowBean(**long** upFlow, **long** downFlow) {  **super**();  **this**.upFlow = upFlow;  **this**.downFlow = downFlow;  **this**.sumFlow = upFlow + downFlow;  }  **public** **void** set(**long** upFlow, **long** downFlow) {  **this**.upFlow = upFlow;  **this**.downFlow = downFlow;  **this**.sumFlow = upFlow + downFlow;  }  **public** **long** getSumFlow() {  **return** sumFlow;  }  **public** **void** setSumFlow(**long** sumFlow) {  **this**.sumFlow = sumFlow;  }  **public** **long** getUpFlow() {  **return** upFlow;  }  **public** **void** setUpFlow(**long** upFlow) {  **this**.upFlow = upFlow;  }  **public** **long** getDownFlow() {  **return** downFlow;  }  **public** **void** setDownFlow(**long** downFlow) {  **this**.downFlow = downFlow;  }  /\*\*  \* 序列化方法  \* **@param** out  \* **@throws** IOException  \*/  @Override  **public** **void** write(DataOutput out) **throws** IOException {  out.writeLong(upFlow);  out.writeLong(downFlow);  out.writeLong(sumFlow);  }  /\*\*  \* 反序列化方法 注意反序列化的顺序和序列化的顺序完全一致  \* **@param** in  \* **@throws** IOException  \*/  @Override  **public** **void** readFields(DataInput in) **throws** IOException {  upFlow = in.readLong();  downFlow = in.readLong();  sumFlow = in.readLong();  }  @Override  **public** String toString() {  **return** upFlow + "\t" + downFlow + "\t" + sumFlow;  }  @Override  **public** **int** compareTo(FlowBean o) {  // 倒序排列，从大到小  **return** **this**.sumFlow > o.getSumFlow() ? -1 : 1;  }  } |

（2）编写mapper

|  |
| --- |
| package com.bigdata.mapreduce.sort;  import java.io.IOException;  import org.apache.hadoop.io.LongWritable;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Mapper;  public class FlowCountSortMapper extends Mapper<LongWritable, Text, FlowBean, Text>{  FlowBean bean = new FlowBean();  Text v = new Text();  @Override  protected void map(LongWritable key, Text value, Context context)  throws IOException, InterruptedException {  // 1 获取一行  String line = value.toString();    // 2 截取  String[] fields = line.split("\t");    // 3 封装对象  String phoneNbr = fields[0];  long upFlow = Long.parseLong(fields[1]);  long downFlow = Long.parseLong(fields[2]);    bean.set(upFlow, downFlow);  v.set(phoneNbr);    // 4 输出  context.write(bean, v);  }  } |

（3）编写reducer

|  |
| --- |
| package com.bigdata.mapreduce.sort;  import java.io.IOException;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Reducer;  public class FlowCountSortReducer extends Reducer<FlowBean, Text, Text, FlowBean>{  @Override  protected void reduce(FlowBean key, Iterable<Text> values, Context context)  throws IOException, InterruptedException {    // 循环输出，避免总流量相同情况  for (Text text : values) {  context.write(text, key);  }  }  } |

（4）编写driver

|  |
| --- |
| package com.bigdata.mapreduce.sort;  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.lib.input.FileInputFormat;  import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;  public class FlowCountSortDriver {  public static void main(String[] args) throws ClassNotFoundException, IOException, InterruptedException {    // 1 获取配置信息，或者job对象实例  Configuration configuration = new Configuration();  Job job = Job.getInstance(configuration);  // 6 指定本程序的jar包所在的本地路径  job.setJarByClass(FlowCountSortDriver.class);  // 2 指定本业务job要使用的mapper/Reducer业务类  job.setMapperClass(FlowCountSortMapper.class);  job.setReducerClass(FlowCountSortReducer.class);  // 3 指定mapper输出数据的kv类型  job.setMapOutputKeyClass(FlowBean.class);  job.setMapOutputValueClass(Text.class);  // 4 指定最终输出的数据的kv类型  job.setOutputKeyClass(Text.class);  job.setOutputValueClass(FlowBean.class);  // 5 指定job的输入原始文件所在目录  FileInputFormat.setInputPaths(job, new Path(args[0]));  FileOutputFormat.setOutputPath(job, new Path(args[1]));    // 7 将job中配置的相关参数，以及job所用的java类所在的jar包， 提交给yarn去运行  boolean result = job.waitForCompletion(true);  System.exit(result ? 0 : 1);  }  } |

### 7.2.4 需求4：不同省份输出文件内部排序（部分排序）

1）需求

要求每个省份手机号输出的文件中按照总流量内部排序。

2）分析：

基于需求3，增加自定义分区类即可。

3）案例实操

（1）增加自定义分区类

|  |
| --- |
| package com.bigdata.mapreduce.sort;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Partitioner;  public class ProvincePartitioner extends Partitioner<FlowBean, Text> {  @Override  public int getPartition(FlowBean key, Text value, int numPartitions) {    // 1 获取手机号码前三位  String preNum = value.toString().substring(0, 3);    int partition = 4;    // 2 根据手机号归属地设置分区  if ("136".equals(preNum)) {  partition = 0;  }else if ("137".equals(preNum)) {  partition = 1;  }else if ("138".equals(preNum)) {  partition = 2;  }else if ("139".equals(preNum)) {  partition = 3;  }  return partition;  }  } |

（2）在驱动类中添加分区类

|  |
| --- |
| // 加载自定义分区类  job.setPartitionerClass(FlowSortPartitioner.**class**);  // 设置Reducetask个数  job.setNumReduceTasks(5); |

## 7.3 辅助排序和二次排序案例（GroupingComparator）

（3）辅助排序：（GroupingComparator分组）

Mapreduce框架在记录到达reducer之前按键对记录排序，但键所对应的值并没有被排序。甚至在不同的执行轮次中，这些值的排序也不固定，因为它们来自不同的map任务且这些map任务在不同轮次中完成时间各不相同。一般来说，大多数MapReduce程序会避免让reduce函数依赖于值的排序。但是，有时也需要通过特定的方法对键进行排序和分组等以实现对值的排序。

（4）二次排序：

在自定义排序过程中，如果compareTo中的判断条件为两个即为二次排序。

1）需求

有如下订单数据

|  |  |  |
| --- | --- | --- |
| 订单id | 商品id | 成交金额 |
| 0000001 | Pdt\_01 | 222.8 |
| 0000001 | Pdt\_06 | 25.8 |
| 0000002 | Pdt\_03 | 522.8 |
| 0000002 | Pdt\_04 | 122.4 |
| 0000002 | Pdt\_05 | 722.4 |
| 0000003 | Pdt\_01 | 222.8 |
| 0000003 | Pdt\_02 | 33.8 |

现在需要求出每一个订单中最贵的商品。

2）输入数据



输出数据预期：

3）分析

（1）利用“订单id和成交金额”作为key，可以将map阶段读取到的所有订单数据按照id分区，按照金额排序，发送到reduce。

（2）在reduce端利用groupingcomparator将订单id相同的kv聚合成组，然后取第一个即是最大值。

4）代码实现

（1）定义订单信息OrderBean

|  |
| --- |
| package com.bigdata.mapreduce.order;  import java.io.DataInput;  import java.io.DataOutput;  import java.io.IOException;  import org.apache.hadoop.io.WritableComparable;  public class OrderBean implements WritableComparable<OrderBean> {  private int order\_id; // 订单id号  private double price; // 价格  public OrderBean() {  super();  }  public OrderBean(int order\_id, double price) {  super();  this.order\_id = order\_id;  this.price = price;  }  @Override  public void write(DataOutput out) throws IOException {  out.writeInt(order\_id);  out.writeDouble(price);  }  @Override  public void readFields(DataInput in) throws IOException {  order\_id = in.readInt();  price = in.readDouble();  }  @Override  public String toString() {  return order\_id + "\t" + price;  }  public int getOrder\_id() {  return order\_id;  }  public void setOrder\_id(int order\_id) {  this.order\_id = order\_id;  }  public double getPrice() {  return price;  }  public void setPrice(double price) {  this.price = price;  }  // 二次排序  @Override  public int compareTo(OrderBean o) {  int result;  if (order\_id > o.getOrder\_id()) {  result = 1;  } else if (order\_id < o.getOrder\_id()) {  result = -1;  } else {  // 价格倒序排序  result = price > o.getPrice() ? -1 : 1;  }  return result;  }  } |

（2）编写OrderSortMapper

|  |
| --- |
| package com.bigdata.mapreduce.order;  import java.io.IOException;  import org.apache.hadoop.io.LongWritable;  import org.apache.hadoop.io.NullWritable;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Mapper;  public class OrderMapper extends Mapper<LongWritable, Text, OrderBean, NullWritable> {  OrderBean k = new OrderBean();    @Override  protected void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {    // 1 获取一行  String line = value.toString();    // 2 截取  String[] fields = line.split("\t");    // 3 封装对象  k.setOrder\_id(Integer.*parseInt*(fields[0]));  k.setPrice(Double.*parseDouble*(fields[2]));    // 4 写出  context.write(k, NullWritable.*get*());  }  } |

（3）编写OrderSortPartitioner

|  |
| --- |
| **package** com.bigdata.mapreduce.order;  **import** org.apache.hadoop.io.NullWritable;  **import** org.apache.hadoop.mapreduce.Partitioner;  **public** **class** OrderPartitioner **extends** Partitioner<OrderBean, NullWritable> {  @Override  **public** **int** getPartition(OrderBean key, NullWritable value, **int** numReduceTasks) {    **return** (key.getOrder\_id() & Integer.***MAX\_VALUE***) % numReduceTasks;  }  } |

（4）编写OrderSortGroupingComparator

|  |
| --- |
| **package** com.bigdata.mapreduce.order;  **import** org.apache.hadoop.io.WritableComparable;  **import** org.apache.hadoop.io.WritableComparator;  **public** **class** OrderGroupingComparator **extends** WritableComparator {  **protected** OrderGroupingComparator() {  **super**(OrderBean.**class**, **true**);  }  @SuppressWarnings("rawtypes")  @Override  **public** **int** compare(WritableComparable a, WritableComparable b) {  OrderBean aBean = (OrderBean) a;  OrderBean bBean = (OrderBean) b;  **int** result;  **if** (aBean.getOrder\_id() > bBean.getOrder\_id()) {  result = 1;  } **else** **if** (aBean.getOrder\_id() < bBean.getOrder\_id()) {  result = -1;  } **else** {  result = 0;  }  **return** result;  }  } |

（5）编写OrderSortReducer

|  |
| --- |
| **package** com.bigdata.mapreduce.order;  **import** java.io.IOException;  **import** org.apache.hadoop.io.NullWritable;  **import** org.apache.hadoop.mapreduce.Reducer;  **public** **class** OrderReducer **extends** Reducer<OrderBean, NullWritable, OrderBean, NullWritable> {  @Override  **protected** **void** reduce(OrderBean key, Iterable<NullWritable> values, Context context)  **throws** IOException, InterruptedException {    context.write(key, NullWritable.*get*());  }  } |

（6）编写OrderSortDriver

|  |
| --- |
| **package** com.bigdata.mapreduce.order;  **import** java.io.IOException;  **import** org.apache.hadoop.conf.Configuration;  **import** org.apache.hadoop.fs.Path;  **import** org.apache.hadoop.io.NullWritable;  **import** org.apache.hadoop.mapreduce.Job;  **import** org.apache.hadoop.mapreduce.lib.input.FileInputFormat;  **import** org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;  **public** **class** OrderDriver {  **public** **static** **void** main(String[] args) **throws** Exception, IOException {  // 1 获取配置信息  Configuration conf = **new** Configuration();  Job job = Job.*getInstance*(conf);  // 2 设置jar包加载路径  job.setJarByClass(OrderDriver.**class**);  // 3 加载map/reduce类  job.setMapperClass(OrderMapper.**class**);  job.setReducerClass(OrderReducer.**class**);  // 4 设置map输出数据key和value类型  job.setMapOutputKeyClass(OrderBean.**class**);  job.setMapOutputValueClass(NullWritable.**class**);  // 5 设置最终输出数据的key和value类型  job.setOutputKeyClass(OrderBean.**class**);  job.setOutputValueClass(NullWritable.**class**);  // 6 设置输入数据和输出数据路径  FileInputFormat.*setInputPaths*(job, **new** Path(args[0]));  FileOutputFormat.*setOutputPath*(job, **new** Path(args[1]));  // 10 设置reduce端的分组  job.setGroupingComparatorClass(OrderGroupingComparator.**class**);  // 7 设置分区  job.setPartitionerClass(OrderPartitioner.**class**);  // 8 设置reduce个数  job.setNumReduceTasks(3);  // 9 提交  **boolean** result = job.waitForCompletion(**true**);  System.*exit*(result ? 0 : 1);  }  } |