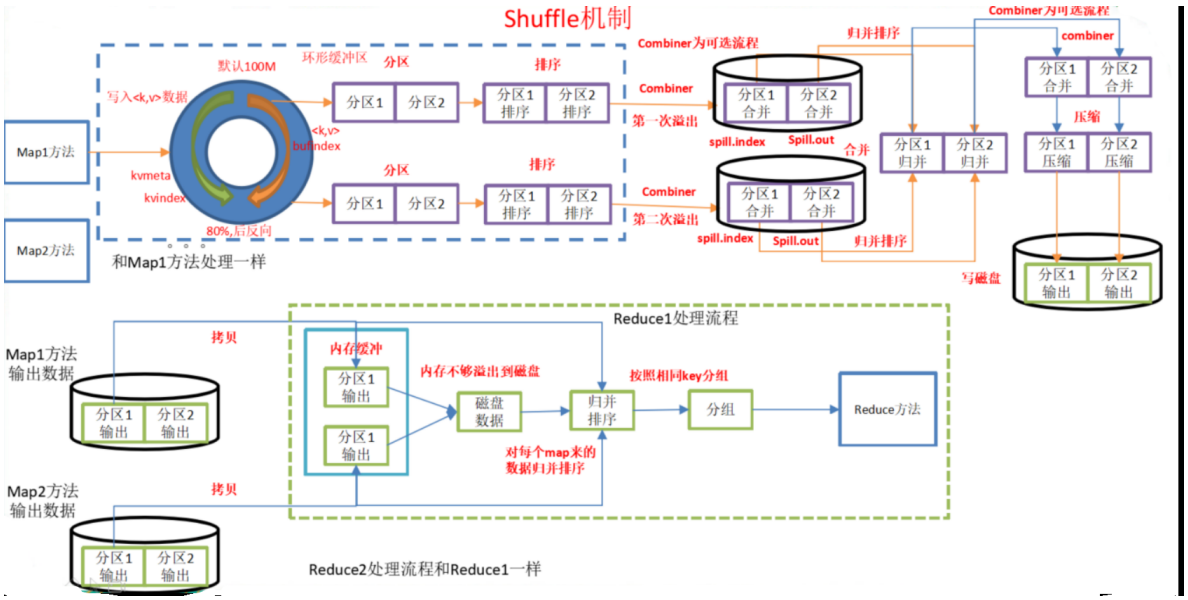
> 凡是打不到你的，终将使你变得更加强大

MapReduce之Shuffle机制

> 前面的文档中已经提到过shuffle了，不过那只是介绍了整个shuffle的流程，还没有介绍如何改变shuffle过程来满足我们的业务需求。

回顾Shuffle机制

> Mapreduce确保每个Reducer的输入都是按key排序的。系统执行排序的过程（即将Mapper输出作为输入传给Reducer）称为Shuffle



具体Shuffle过程详解，如下：

1）MapTask收集我们的map()方法输出的kv对，放到内存缓冲区中

2）从内存缓冲区不断溢出本地磁盘文件，可能会溢出多个文件

3）多个溢出文件会被合并成大的溢出文件

4）在溢出过程及合并的过程中，都要调用Partitioner进行分区和针对key进行排序

5）ReduceTask根据自己的分区号，去各个MapTask机器上取相应的结果分区数据

6）ReduceTask会取到同一个分区的来自不同MapTask的结果文件，ReduceTask会将这些文件再进行合并（归并排序）

7）合并成大文件后，Shuffle的过程也就结束了，后面进入ReduceTask的逻辑运算过程（从文件中取出一个一个的键值对Group，调用用户自定义的reduce()方法）

注意：

> Shuffle中的缓冲区大小会影响到MapReduce程序的执行效率，原则上说，缓冲区越大，磁盘io的次数越少，执行速度就越快。缓冲区的大小可以通过参数调整，参数：io.sort.mb 默认100M。

Partition分区

需求

> 将统计结果按照手机号前三位不同输出到不同文件中（分区）

需求分析

> 默认分区是根据key的hashCode对Reduce Tasks个数取模得到的，用户没法控制哪个key存储到哪个分区

```
public class HashPartitioner<k, v> extends Partitioner<K, v>
public int getPartition（k key, v value, int numReduceTasks） {
//注意这里是根据key的值进行hash然后与上整数最大值,最后对分区数取值
 return （key.hashCode（）& Integer. MAX_VALUE ）% numReduceTasks
 }
```

> 如果想自定义分区，就要继承Partitioner，重写getPartition()方法，自己定义分区规则，然后再驱动中设置这个分区类，还要设置对应的ReduceTask个数

分区总结
（1）如果 Reduce Task的数量> getPartition的结果数，则会多产生几个空的输出文件part-r-000Xx
（2）如果1< ReduceTask的数量< getPartition的结果数，则有一部分分区数据无处安放，会Exception；
（3）如果 ReduceTask的数量=1，则不管 Map Task端输出多少个分区文件，最终结果都交给这一个Reduce Task，最终也就只会产生一个结果文件 part-I0000

自定义Partitioner

Partitioner代码

```
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);

        // 驱动中分区数也要设置正确!
        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;
    }
}
```

Driver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.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 {

        // 输入输出路径需要根据自己电脑上实际的输入输出路径设置
        args = new String[]{"e:/output1","e:/output2"};

        // 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);
    }
}
```

WritableComparable排序

> 排序是 Mapreduce框架中最重要的操作之一。
> MapTask和 Reducetask均会对数据**按照key**进行排序。(如果我们不想按照key排序呢)

> 该操作属于Hadoop的默认行为。任何应用程序中的数据均会被排序，而不管逻辑上是否需要。
> 默认排序是按照字典顺序排序，且实现该排序的方法是快速排序。

排序发生的时机

> - 对于 MapTask，它会将处理的结果暂时放到一个缓冲区中，当缓冲区使用率达到一定阈值后，再对**缓冲区中的数据进行一次排序**，并将这些有序数据写到磁盘上，而当数据处理完毕后，它会对磁盘上所有文件进行一次合并，以将这些文件合并成一个大的有序文件
> - 对于 ReduceTask，它从每个 MapTask上远程拷贝相应的数据文件，如果文件大小超过一定阈值，则放到磁盘上，否则放到内存中。如果磁盘上文件数目达到定阈值，则进行一次合并以生成一个更大文件；如果内存中文件大小或者数目超过一定阈值，则进行一次合并后将数据写到磁盘上。当所有数据拷贝完毕后， Reduce Task统一**对内存和磁盘上的所有数据进行一次归并排序。**

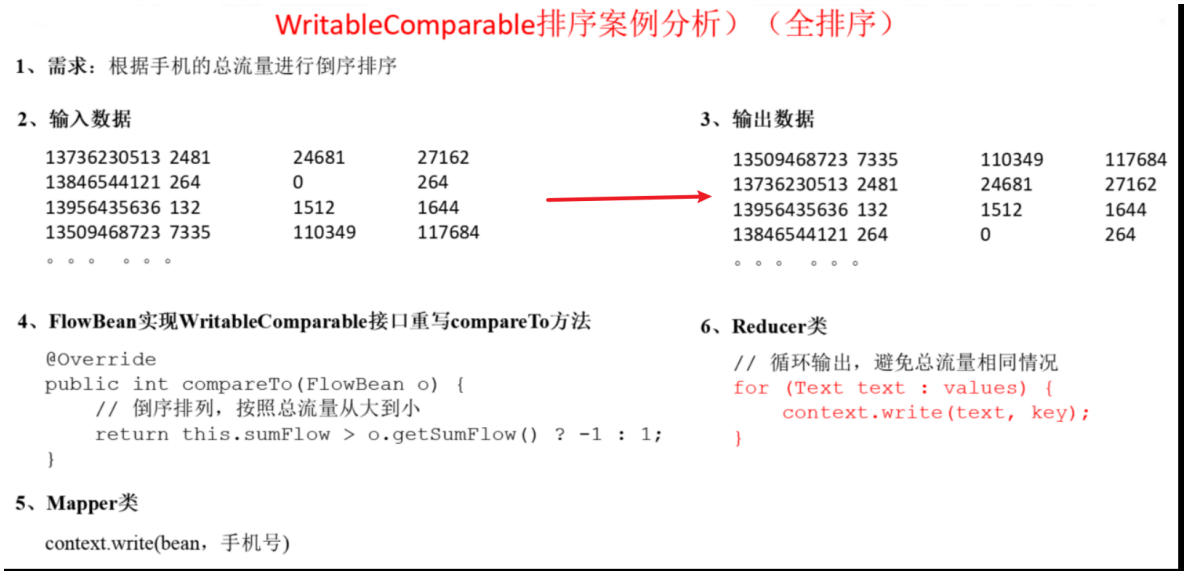
排序的分类

> （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有序。
> （2）全排序最终输出结果只有一个文件，且文件内部有序。实现方式是只设置一个 Reduce Task。但该方法在处理大型文件时效率极低，因为一台机器处理所有文件，完全丧失了Mapreduce所提供的并行架构。
> （3）辅助排序：（ Grouping Comparator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。**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 compare To中的判断条件为两个即为二次排序

自定义排序WritableComparable【全排序】

（1）创建类实现WritableComparable接口

（2）重写接口中compareTo方法即可



FlowBean代码

```
import java.io.DataInput;
import java.io.DataOutput;
import java.io.IOException;
import org.apache.hadoop.io.WritableComparable;

public class FlowBean implements WritableComparable<FlowBean> {
```

```
    // 映射读入数据
    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，按照总流量排序
        return this.sumFlow > o.getSumFlow() ? -1 : 1;
    }
}
```

Mapper代码

```
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);
        }
    }
```

Reducer代码

```
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);
        }
    }
}
```

Driver代码

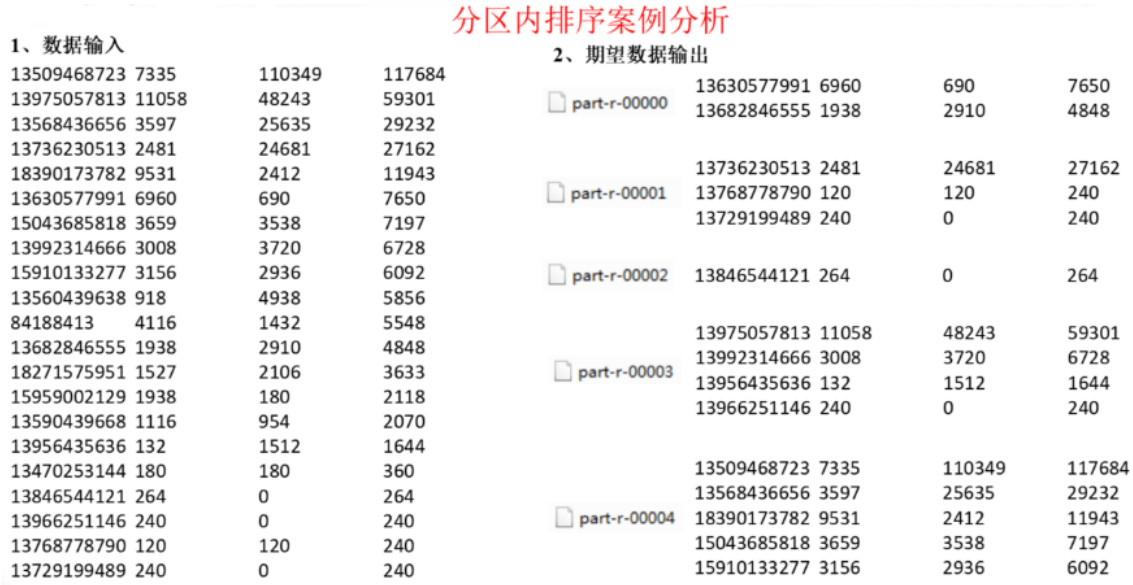
```
import java.io.IOException;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class FlowCountSortDriver {

    public static void main(String[] args) throws ClassNotFoundException,
IOException, InterruptedException {

        // 输入输出路径需要根据自己电脑上实际的输入输出路径设置
        args = new String[]{"e:/output1","e:/output2"};

        // 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);
    }
}
```

自定义排序WritableComparable【区内排序】

需求

> 其实就是先按照手机号进行分区，然后在进行排序即可，这个和开窗函数中先Parttion By 在Order By是一样的，后面使用Hive写一条sql直接完成这些操作，不用再写mapreduce代码了



只要再全排序代码基础上加上分区的代码即可！

Partitioner代码

```
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
```

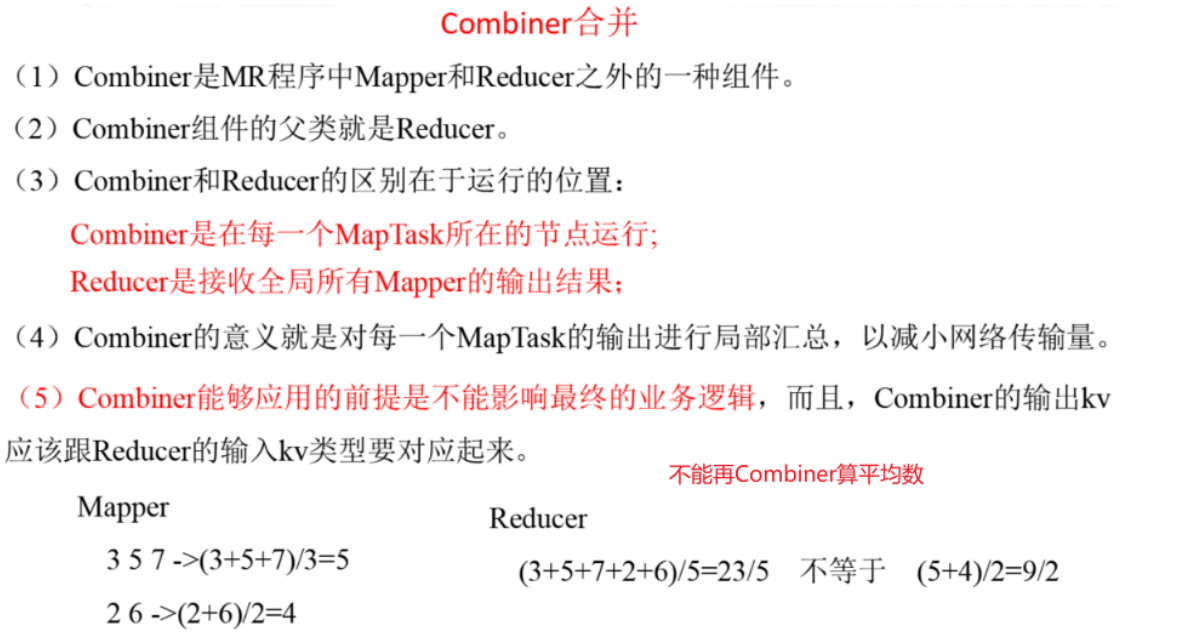
```
        if ("136".equals(preNum)) {
            partition = 0;
        }else if ("137".equals(preNum)) {
            partition = 1;
        }else if ("138".equals(preNum)) {
            partition = 2;
        }else if ("139".equals(preNum)) {
            partition = 3;
        }

        return partition;
    }
}
```

Driver代码

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

Combiner合并

> Combiner可以理解为小Reducer，只是运行的位置不同，它是为了减少网络的传输量而提前进行的聚合操作



自定义Combiner

Combiner代码

```
public class WordcountCombiner extends Reducer<Text, IntWritable,
Text,IntWritable>{
    @Override
    protected void reduce(Text key, Iterable<IntWritable> values,Context
context) throws IOException, InterruptedException {

        // 1 汇总操作,不要平均
```

```
        int count = 0;
        for(IntWritable v :values){
            count += v.get();
        }

        // 2 写出
        context.write(key, new IntWritable(count));
    }
}
```

Driver代码

```
job.setCombinerClass(WordcountCombiner.class);
```

GroupingComparator分组（辅助排序）

需求

> 有如下订单数据，要求找出每个订单中最贵的商品

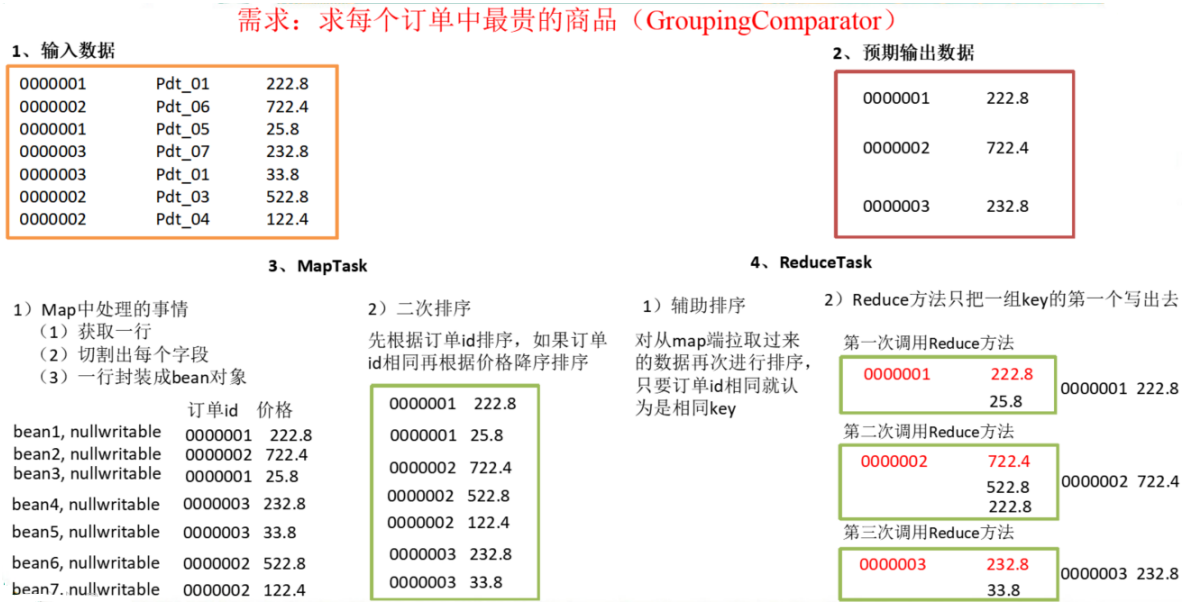
订单id	商品id	成交金额
0000001	Pdt_01	222.8
	Pdt_06	25.8
0000002	Pdt_03	522.8
	Pdt_04	122.4
	Pdt_05	722.4
0000003	Pdt_07	232.8
	Pdt_02	33.8

输出就是：

1 222.8

2 722.4

3 232.8



可以看出只要先二次排序然后再按照id分区输出第一个数据即可

OrderBean代码

> 二次排序

```
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;
    }
}
```

OrderSortMapper代码

```
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());
    }
}
```

OrderSortGroupingComparator代码

```
import org.apache.hadoop.io.WritableComparable;
import org.apache.hadoop.io.WritableComparator;

public class OrderGroupingComparator extends WritableComparator {

    protected OrderGroupingComparator() {
        super(OrderBean.class, true);
    }

    @Override
    public int compare(WritableComparable a, WritableComparable b) {

        OrderBean aBean = (OrderBean) a;
        OrderBean bBean = (OrderBean) b;

        int result;
        // 按照id进行排序
        if (aBean.getOrder_id() > bBean.getOrder_id()) {
            result = 1;
        } else if (aBean.getOrder_id() < bBean.getOrder_id()) {
            result = -1;
        } else {
            result = 0;
        }

        return result;
    }
}
```

OrderSortReducer代码

```
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());
    }
}
```

OrderSortDriver代码

```
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 {

// 输入输出路径需要根据自己电脑上实际的输入输出路径设置
        args  = new String[]{"e:/input/inputorder" , "e:/output1"};

        // 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]));
```

```
        // 8 设置reduce端的分组
    job.setGroupingComparatorClass(OrderGroupingComparator.class);

        // 7 提交
        boolean result = job.waitForCompletion(true);
        System.exit(result ? 0 : 1);
    }
}
```