# 1. MapReduce 基本原理

MapReduce是一种编程模型，用于大规模数据集的分布式运算

## MapReduce 通俗解释

图书馆要清点图书数量，有10个书架，管理员为了加快统计速度，找来了10个同学，每个同学负责统计一个书架的图书数量

张同学 统计 书架1
王同学 统计 书架2
刘同学 统计 书架3
.....

过了一会儿，10个同学陆续到管理员这汇报自己的统计数字，管理员把各个数字加起来，就得到了图书总数

这个过程就可以理解为MapReduce的工作过程

### MapReduce中有两个核心操作

（1）map

管理员分配哪个同学统计哪个书架，每个同学都进行相同的“统计”操作，这个过程就是map

（2）reduce

每个同学的结果进行汇总，这个过程就是reduce

## MapReduce 工作过程拆解

下面通过一个经典案例（单词统计）看MapReduce是如何工作的

有一个文本文件，被分成了4份，分别放到了4台服务器中存储

Text 1: the weather is good

Text 2: today is good

Text 3: good weather is good

Text 4: today has good weather

现在要统计出每个单词的出现次数

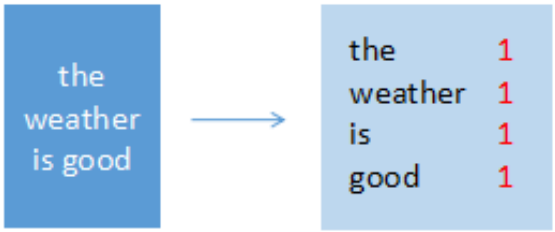
## 处理过程

（1）拆分单词

- map节点 1

输入：“the weather is good”

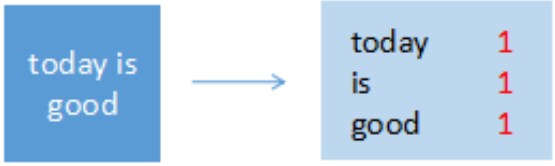
输出：(the, 1), (weather, 1), (is, 1), (good, 1)




- map节点 2

输入：“today is good”

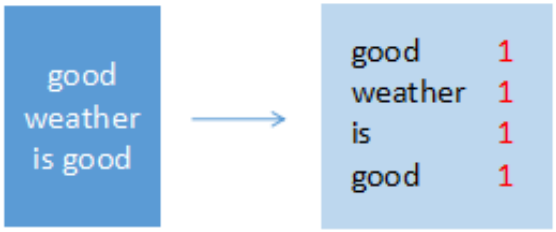
输出：(today, 1), (is, 1), (good, 1)




- map节点 3

输入：“good weather is good”

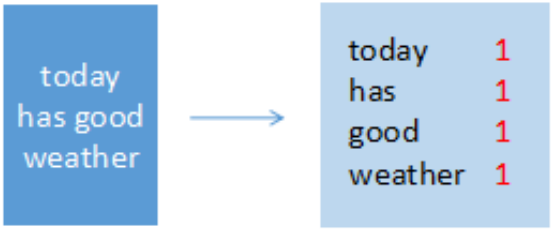
输出：(good, 1), (weather, 1), (is, 1), (good, 1)




- map节点 4

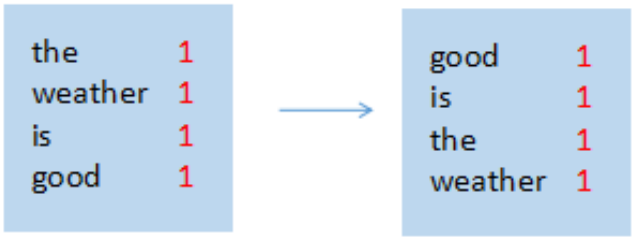
输入：“today has good weather”

输出：(today, 1), (has, 1), (good, 1), (weather, 1)



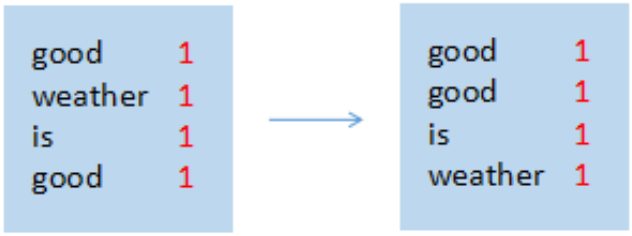

（2）排序

- map节点 1

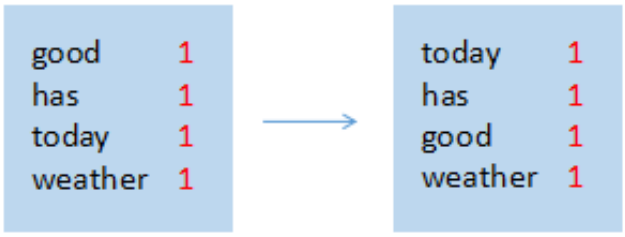



- map节点 2


- map节点 3




- map节点 4




(3) 合并

- map节点 1


- map节点 2


- map节点 3

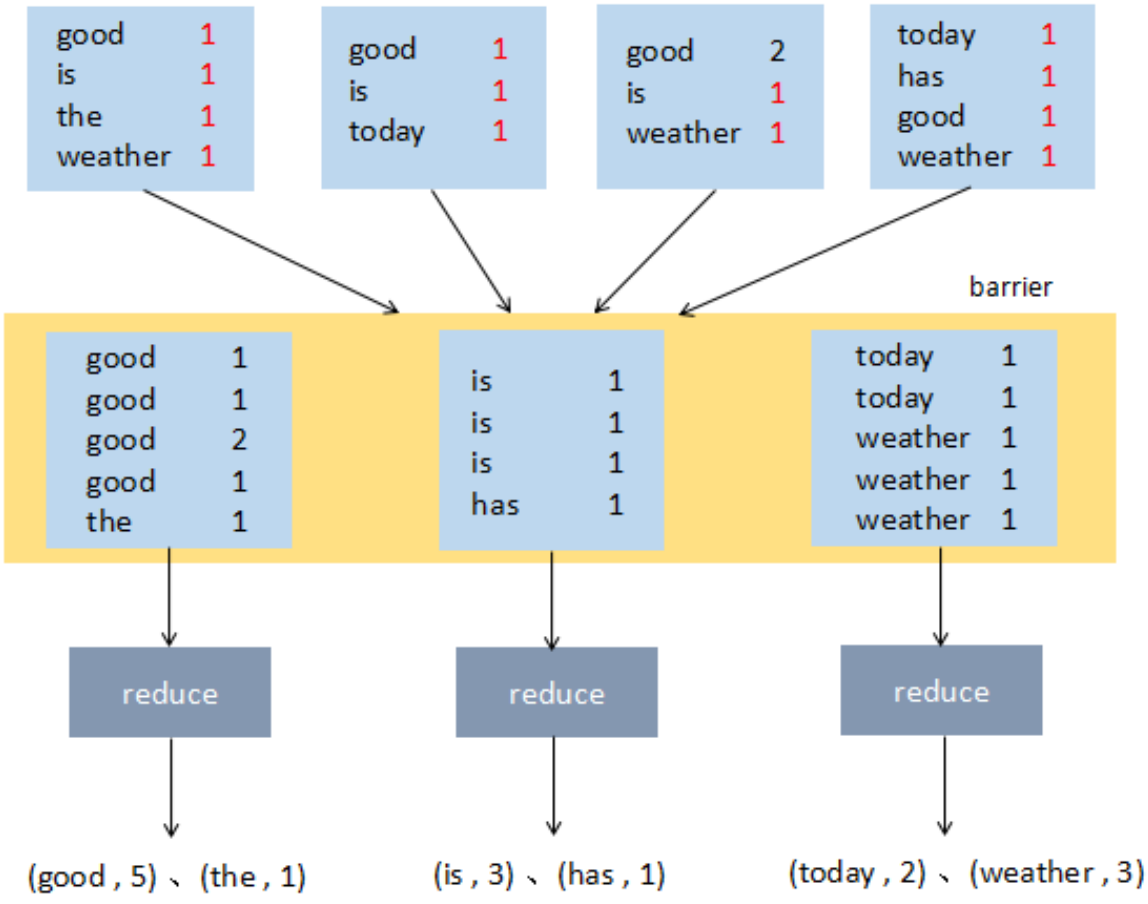

- map节点 4


（4）汇总统计

每个map节点都完成以后，就要进入reduce阶段了

例如使用了3个reduce节点，需要对上面4个map节点的结果进行重新组合，比如按照26个字母分成3段，分配给3个reduce节点

reduce节点进行统计，计算出最终结果




这就是最基本的 mapreduce 处理流程

# MapReduce 编程思路

了解了mapredude的工作过程，我们思考一下用代码实现时需要做哪些工作?

1. 在4个服务器中启动4个map任务
2. 每个map任务读取目标文件，每读一行就拆分一下单词，并记下来此单词出现了一次
3. 目标文件的每一行都处理完成后，需要把单词进行排序
4. 在3个服务器上启动reduce任务
5. 每个reduce获取一部分map的处理结果
6. reduce任务进行汇总统计，输出最终的结果数据

可以看到这个过程是很复杂的，这还只是非常粗略的过程，实际上要复杂得多

但不用担心，MapReduce 是一个非常优秀的编程模型，已经把绝大多数的工作做完了，我们**只需要关心2个部分**：

1. map 处理逻辑 - 对传进来的一行数据如何处理？输出什么信息？
2. reduce 处理逻辑 - 对传进来的map处理结果如果处理？输出什么信息？

编写好这两个核心业务逻辑之后，只需要几行简单的代码把map和reduce装配成一个job，然后提交给hadoop集群就可以了

至于其他的复杂细节，例如如何启动 map任务 和 reduce任务、如何读取文件、如对map结果排序、如何把map结果数据分配给reduce、reduce如何把最终结果保存到文件 等等，MapReduce框架都帮我们做好了，而且还支持很多自定义扩展配置，例如 如何读文件、如何组织map或者reduce的输出结果 等等，后面的示例中会有介绍

# 2. MapReduce 入门示例 - WordCount 单词统计

WordCount 是非常好的入门示例，相当于 helloword，下面就开发一个 wordcount 的 mapreduce 程序，体验实际开发方式

## 2.1 安装 Hadoop 实践环境

您可以选择`自己搭建环境`，也可以使用`打包好的 hadoop 环境`（版本 2.7.3）

这个hadoop环境实际上是一个虚机镜像，所以需要安装 virtualbox 虚拟机、vagrant 镜像管理工具，和我做的 hadoop 镜像，然后用这个镜像启动虚机就可以了，下面是具体操作步骤：

1）安装 virtualbox

下载地址

```
https://www.virtualbox.org/wiki/Downloads
```

2）安装 vagrant

因为官网下载较慢，我上传到了云盘

windows版

```
链接: https://pan.baidu.com/s/1pKKQGHl 密码: eykr
```

Mac版

```
链接: https://pan.baidu.com/s/1slts9yt 密码: aig4
```

安装完成后，在命令行终端下就可以使用 vagrant 命令

3）下载 hadoop 镜像

```
链接: https://pan.baidu.com/s/1bpaisnd 密码: pn6c
```

4）启动

加载 hadoop 镜像

```
vagrant box add {自定义镜像名称} {镜像所在路径}
```

例如您想命名为 `hadoop`，镜像下载后的路径为 `d:\hadoop.box`，加载命令就是这样：

```
vagrant box add hadoop d:\hadoop.box
```

创建工作目录，例如 `d:\hdfstest`

进入此目录，初始化

```
cd d:\hdfstest
vagrant init hadoop
```

启动虚机

```
vagrant up
```

启动完成后，就可以使用SSH客户端登录 hadoop 服务器虚机了

```
IP  127.0.0.1
端口 2222
用户名 root
密码 vagrant
```

在 hadoop 服务器中启动 hdfs 和 yarn，之后就可以运行 mapreduce 程序了

```
start-dfs.sh
start-yarn.sh
```

## 2.2 创建项目

> 流程是在本机开发，然后打包，上传到 hadoop 服务器上运行

新建项目目录 `wordcount`，其中新建文件 `pom.xml`，内容：

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="h
ttp://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http:
//maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>demo.mr</groupId>
    <artifactId>mapreduce-wordcount</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>jar</packaging>

    <name>mapreduce-wordcount</name>
    <url>http://maven.apache.org</url>

    <properties>
```

```
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>

    <dependencies>
        <!-- https://mvnrepository.com/artifact/commons-beanutils/commons-beanutils -->
        <dependency>
            <groupId>commons-beanutils</groupId>
            <artifactId>commons-beanutils</artifactId>
            <version>1.9.3</version>
        </dependency>

        <!-- https://mvnrepository.com/artifact/org.apache.hadoop/hadoop-common -->
        <dependency>
            <groupId>org.apache.hadoop</groupId>
            <artifactId>hadoop-common</artifactId>
            <version>2.7.3</version>
        </dependency>
        <!-- https://mvnrepository.com/artifact/org.apache.hadoop/hadoop-hdfs -->
        <dependency>
            <groupId>org.apache.hadoop</groupId>
            <artifactId>hadoop-hdfs</artifactId>
            <version>2.7.3</version>
        </dependency>
        <!-- https://mvnrepository.com/artifact/org.apache.hadoop/hadoop-mapreduce-client-common -->
        <dependency>
            <groupId>org.apache.hadoop</groupId>
            <artifactId>hadoop-mapreduce-client-common</artifactId>
            <version>2.7.3</version>
        </dependency>
        <!-- https://mvnrepository.com/artifact/org.apache.hadoop/hadoop-mapreduce-client-core -->
        <dependency>
            <groupId>org.apache.hadoop</groupId>
            <artifactId>hadoop-mapreduce-client-core</artifactId>
            <version>2.7.3</version>
        </dependency>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>3.8.1</version>
```

```
            <scope>test</scope>
        </dependency>
    </dependencies>
</project>
```

然后创建源码目录 `src/main/java`

现在的目录结构

```
├── pom.xml
├── src
│   └── main
│       └── java
```

## 2.3 代码

**mapper程序：** src/main/java/WordcountMapper.java

内容：

```
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> {
    @Override
    protected void map(LongWritable key, Text value, Context co
ntext)
            throws IOException, InterruptedException {

        // 得到输入的每一行数据
        String line = value.toString();

        // 通过空格分割
        String[] words = line.split(" ");

        // 循环遍历 输出
        for (String word : words) {
            context.write(new Text(word), new IntWritable(1));
```

```
        }
    }
}
```

这里定义了一个 mapper 类，其中有一个 map 方法

mapreduce框架每读到一行数据，就会调用一次这个 map 方法

map 的处理流程就是接收一个 key value 对儿，然后进行业务逻辑处理，最后输出一个 key value 对儿

```
Mapper<LongWritable, Text, Text, IntWritable>
```

其中的4个类型分别是：输入 key 类型、输入 value 类型、输出 key 类型、输出 value 类型

mapreduce框架读到一行数据后以 key value 形式传进来，key 默认情况下是mr框架所读到一行文本的起始偏移量（Long 类型），value 默认情况下是mr框架所读到的一行的数据内容（String 类型）

输出也是 key value 形式的，是用户自定义逻辑处理完成后定义的key，用户自己决定用什么作为key，value 是用户自定义逻辑处理完成后的 value，内容和类型也是用户自己决定

此例中 输出 key 就是 word（字符串类型），输出 value 就是单词数量（整型）

这里的数据类型和我们常用的不一样，因为 mapreduce 程序的输出输出数据需要在不同机器间传输，所以必须是可序列化的，例如 Long 类型，hadoop 中定义了自己的可序列化类型 LongWritable，String 对应的是 Text，int 对应的是 IntWritable

**reduce程序：** src/main/java/WordCountReducer.java

```
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
s,
Context context) throws IOException, InterruptedException {

        Integer count = 0;
        for (IntWritable value : values) {
            count += value.get();
        }
        context.write(key, new IntWritable(count));
    }
}
```

这里定义了一个 Reducer类 和一个 reduce方法

```
Reducer<Text, IntWritable, Text, IntWritable>
```

4个类型分别指：输入 key 的类型、输入 value 的类型、输出 key 的类型、输出 value 的类型

需要注意，reduce方法接收的是：一个字符串类型的 key、一个可迭代的数据集

因为reduce任务读取到map任务处理结果是这样的：

```
(good,1) (good,1) (good,1) (good,1)
```

当传给 reduce方法 时，就变为：

```
key : good
value : (1,1,1,1)
```

所以，reduce 方法接收到的是同一个 key 的一组value

**主程序：** src/main/java/WordCountMapReduce.java

```
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;
```

```java
public class WordCountMapReduce {
    public static void main(String[] args) throws Exception{
        // 创建配置对象
        Configuration conf = new Configuration();

        // 创建job对象
        Job job = Job.getInstance(conf, "wordcount");

        // 设置运行job的类
        job.setJarByClass(WordCountMapReduce.class);

        // 设置 mapper 类
        job.setMapperClass(WordcountMapper.class);

        // 设置 reduce 类
        job.setReducerClass(WordCountReducer.class);

        // 设置 map 输出的 key value
        job.setMapOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);

        // 设置 reduce 输出的 key value
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);

        // 设置输入输出的路径
        FileInputFormat.setInputPaths(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        // 提交job
        boolean b = job.waitForCompletion(true);

        if(!b){
            System.out.println("wordcount task fail!");
        }
    }
}
```

这个 main 方法就是用来组装一个 job 并提交执行

## 2.4 编译打包

在 pom.xml 所在目录下执行打包命令：

```
mvn package
```

执行完成后，会自动生成 `target` 目录，其中有打包好的 jar 文件

现在项目文件结构

```
├── pom.xml
├── src
│   └── main
│       └── java
│           ├── WordCountMapReduce.java
│           ├── WordCountReducer.java
│           └── WordcountMapper.java
└── target
    ├── ...
    ├── mapreduce-wordcount-0.0.1-SNAPSHOT.jar
```

## 2.5 运行

先把 target 中的 jar 上传到 hadoop 服务器

然后在 hadoop 服务器的 hdfs 中准备测试文件（把 hadoop 所在目录下的txt文件都上传到 hdfs）

```
cd $HADOOP_HOME
hdfs dfs -mkdir -p /wordcount/input
hdfs dfs -put *.txt /wordcount/input
```

执行 wordcount jar

```
hadoop jar mapreduce-wordcount-0.0.1-SNAPSHOT.jar WordCountMapR
educe /wordcount/input /wordcount/output
```

执行完成后验证

```
hdfs dfs -cat /wordcount/output/*
```

可以看到单词数量统计结果

# 3. MapReduce 执行过程分析

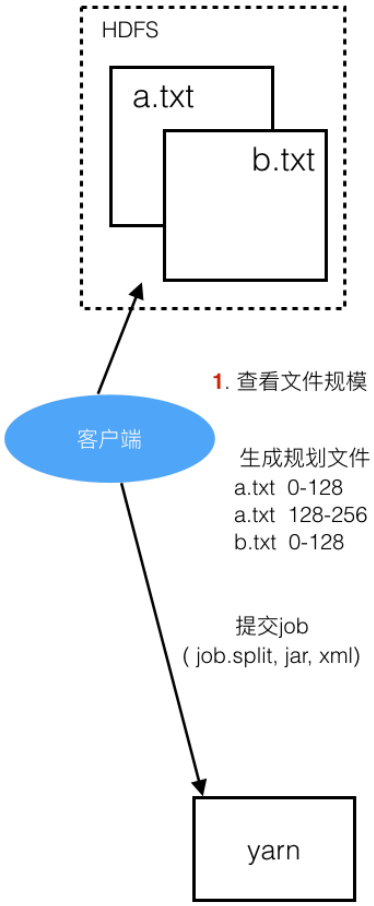
下面看一下从 job 提交到执行完成这个过程是怎样的

（1）客户端提交任务

client 提交任务时会先到 hdfs 中查看目标文件的大小，了解要获取的数据的规模，然后形成任务分配的规划，例如：

a.txt 0-128M 交给一个task，128-256M 交给一个task，b.txt 0-128M 交给一个task，128-256M 交给一个task ...，形成规划文件 job.split

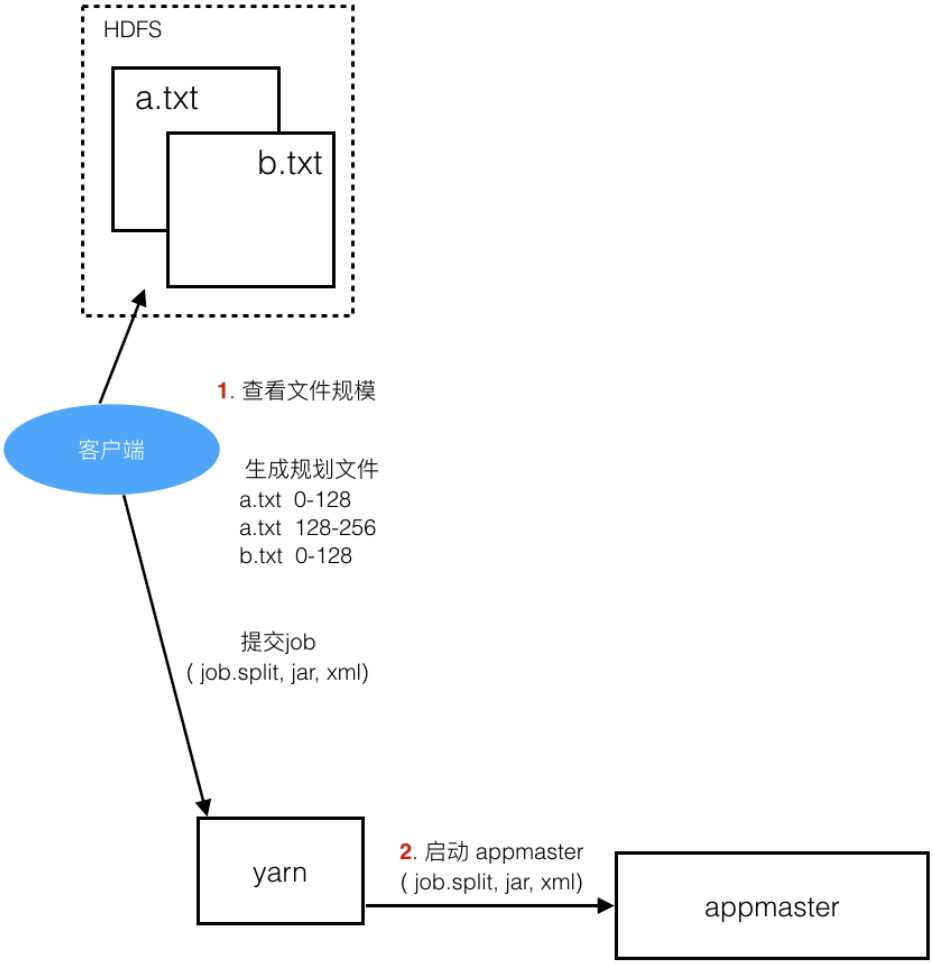
然后把规划文件 job.split、jar、配置文件xml 提交给 yarn（hadoop集群资源管理器，负责为任务分配合适的服务器资源）




（2）启动 appmaster

> appmaster 是本次 job 的主管，负责 maptask 和 reducetask 的启动、监控、协调管理工作

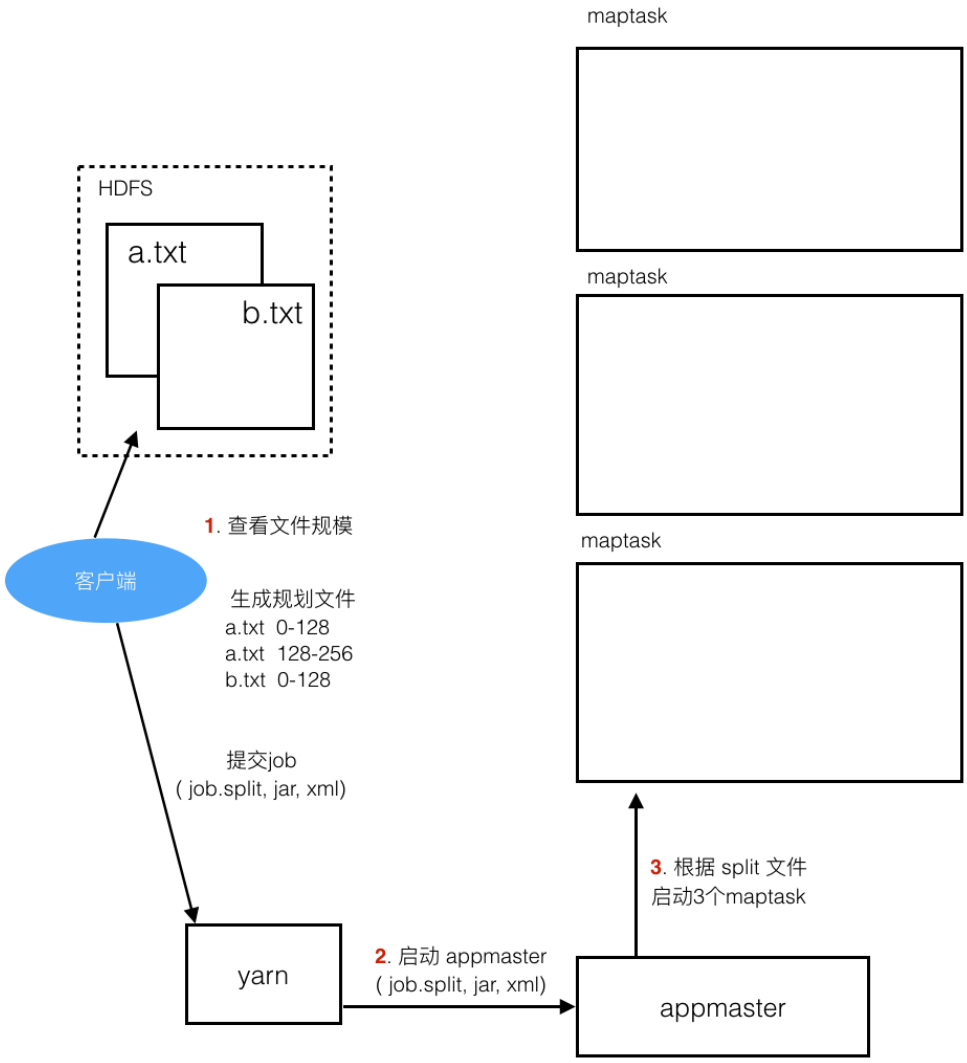
yarn 找一个合适的服务器来启动 appmaster，并他 job.split、jar、xml 交给他




（3）启动 maptask

appmaster 启动后，根据规划文件job.split中的分片信息启动 maptask，一个分片对应一个 maptask

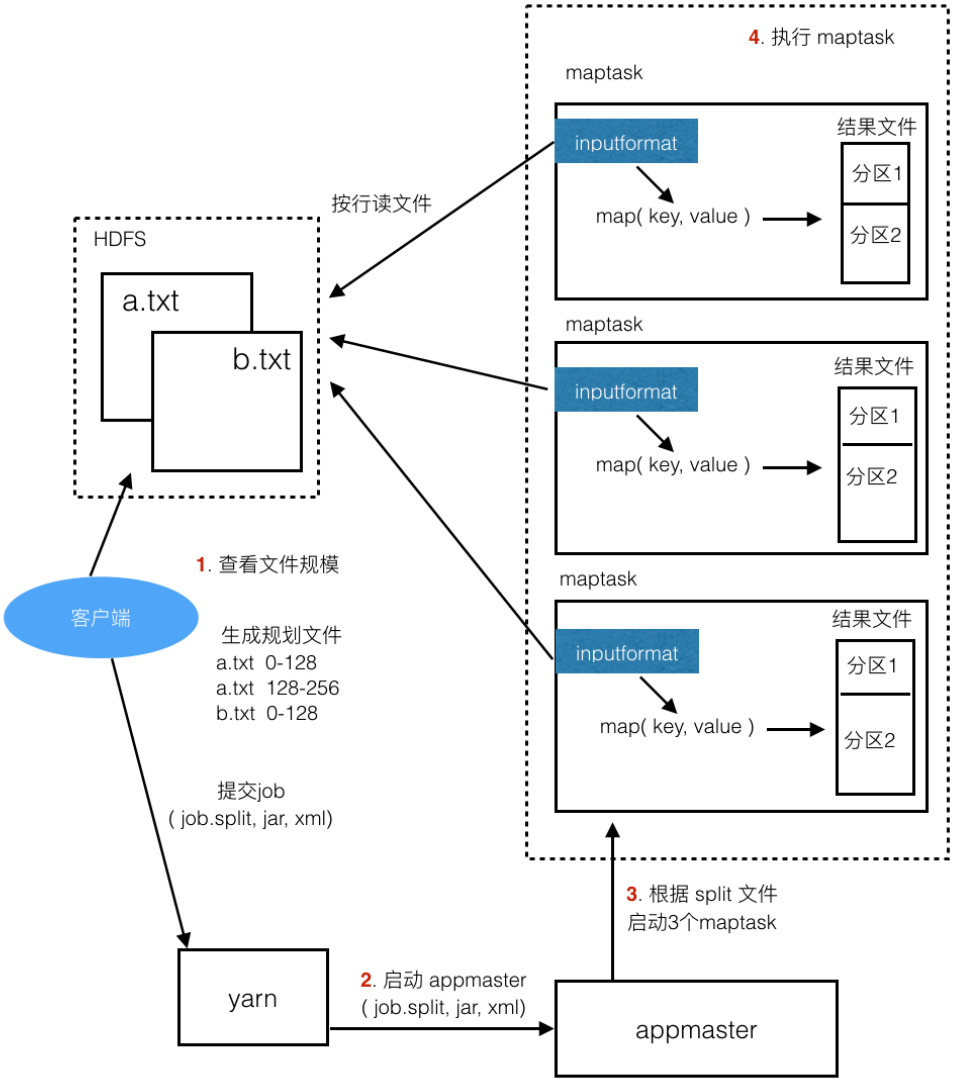
分配 maptask 时，会尽量让 maptask 在目标数据所在的 datanode 上执行




（4）执行 maptask

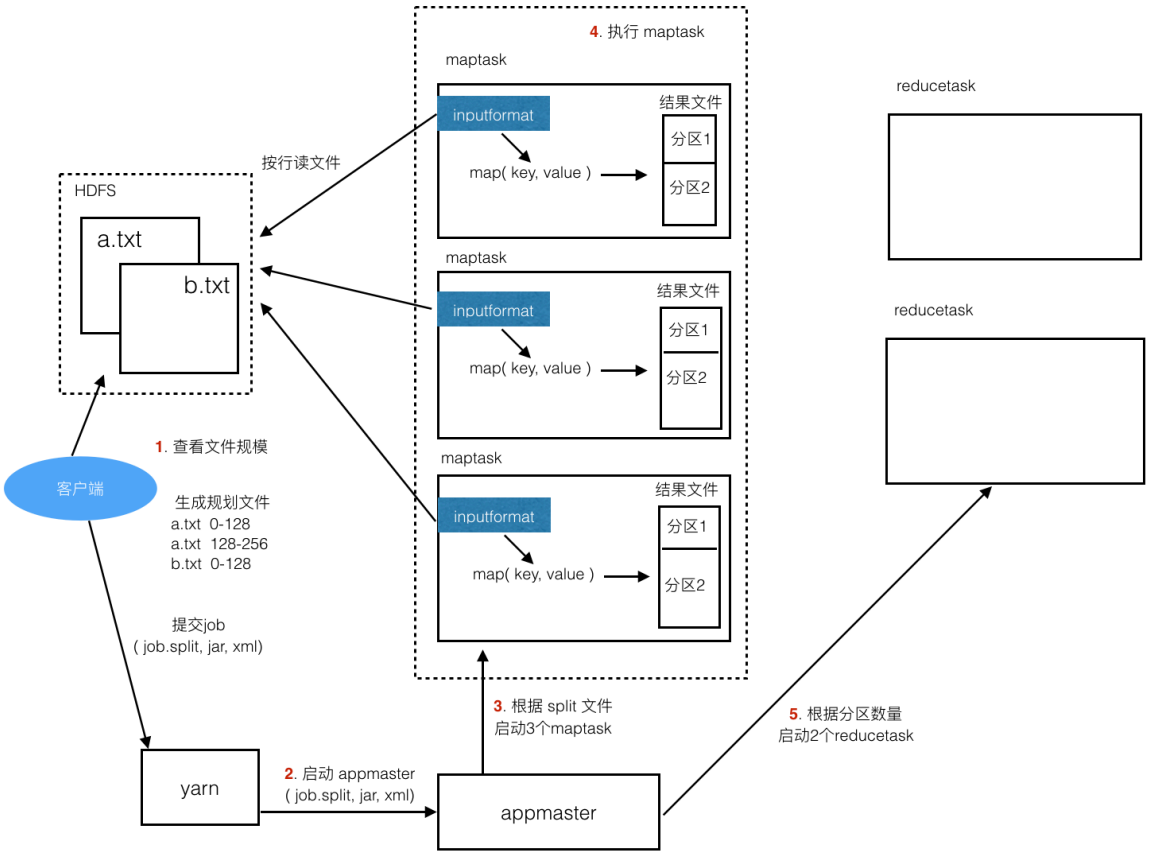
maptask 会一行行的读目标文件，交给我们写的map程序，读一行就调一次 map 方法，map 调用 context.write 把处理结果写出去，保存到本机的一个结果文件中，这个文件中的内容是分区且有序的

分区的作用就是定义哪些key在一组，一个分区对应一个reducer




（5）启动 reducetask

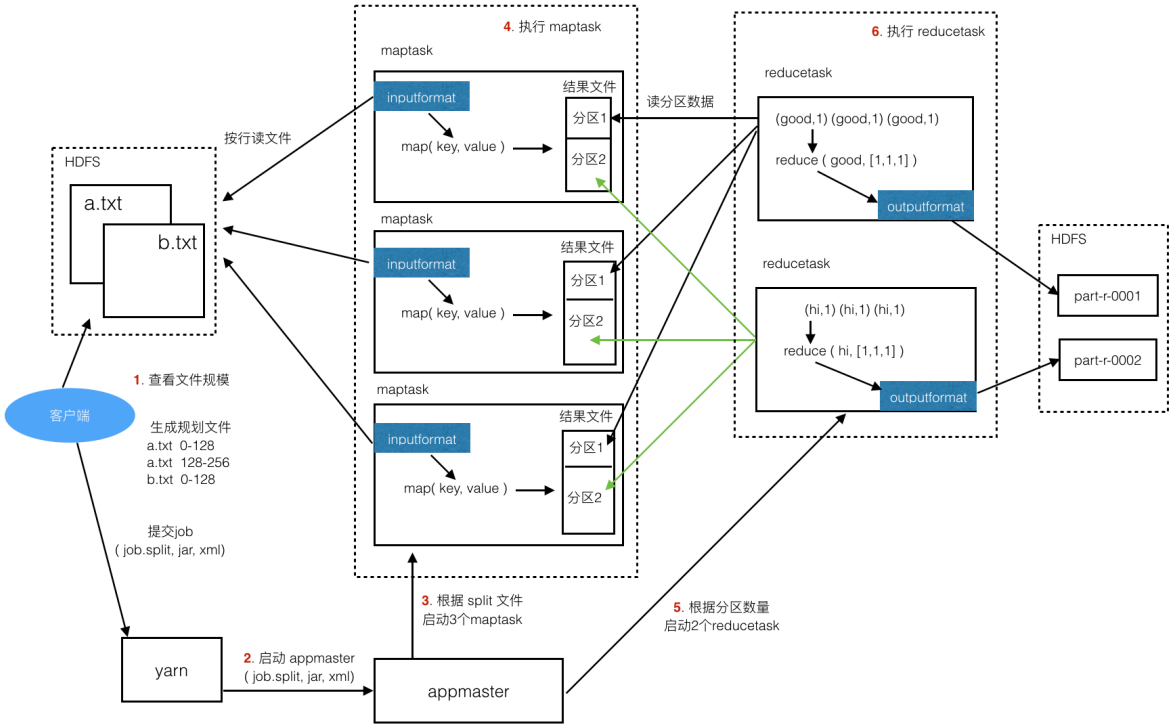
maptask 都运行完成后，appmaster 再启动 reducetask，maptask的结果中有几个分区就启动几个 reducetask




（6）执行 reducetask

reducetask 去读取 maptask 的结果文件中自己对应的那个分区数据，例如 reducetask_01 去读第一个分区中的数据

reducetask 把读到的数据按 key 组织好，传给 reduce 方法进行处理，处理结果写到指定的输出路径




# 4. 实例1 - 自定义对象序列化

## 4.1 需求与实现思路

### 4.1.1 需求

需要统计手机用户流量日志，日志内容示例：

| 手机号 | 上行流量 | 下行流量 |
| --- | --- | --- |
| 13726230501 | 200 | 1100 |
| 13396230502 | 300 | 1200 |
| 13897230503 | 400 | 1300 |
| 13897230503 | 100 | 300 |
| 13597230534 | 500 | 1400 |
| 13597230534 | 300 | 1200 |

要把同一个用户的 上行流量、下行流量 进行累加，并计算出总和

例如上面的 13897230503 有两条记录，就要对这两条记录进行累加，计算总和，得到：

```
13797230503, 500, 1600, 2100
```

### 4.1.2 实现思路

- map

接收日志的一行数据，key 为行的偏移量，value 为此行数据

输出时，应以手机号为 key，value 应为一个整体，包括：上行流量、下行流量、总流量

手机号是字符串类型 Text，而这个整体不能用基本数据类型表示，需要我们自定义一个bean对象，并且要实现可序列化

```
key: 13897230503
value: < upFlow:100, dFlow:300, sumFlow:400 >
```

- reduce

接收一个手机号标识的 key，及这个手机号对应的bean对象集合

例如：

```
key:
13897230503

value:
< upFlow:400, dFlow:1300, sumFlow:1700 >,
< upFlow:100, dFlow:300, sumFlow:400 >
```

迭代bean对象集合，累加各项，形成一个新的bean对象，例如

```
< upFlow:400+100, dFlow:1300+300, sumFlow:1700+400 >
```

最后输出：

```
key: 13897230503
value: < upFlow:500, dFlow:1600, sumFlow:2100 >
```

## 4.2 代码实践

### 4.2.1 创建项目

新建项目目录 `serializebean`，其中新建文件 `pom.xml`，内容:

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="h
ttp://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http:
//maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>demo.mr</groupId>
    <artifactId>mapreduce-serializebean</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>jar</packaging>

    <name>mapreduce-serializebean</name>
    <url>http://maven.apache.org</url>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sour
ceEncoding>
    </properties>

    <dependencies>
        <!-- https://mvnrepository.com/artifact/commons-beanuti
ls/commons-beanutils -->
        <dependency>
            <groupId>commons-beanutils</groupId>
            <artifactId>commons-beanutils</artifactId>
            <version>1.9.3</version>
        </dependency>

        <!-- https://mvnrepository.com/artifact/org.apache.hado
op/hadoop-common -->
        <dependency>
            <groupId>org.apache.hadoop</groupId>
            <artifactId>hadoop-common</artifactId>
            <version>2.7.3</version>
        </dependency>
```

```
        <!-- https://mvnrepository.com/artifact/org.apache.hadoop/hadoop-hdfs -->
        <dependency>
            <groupId>org.apache.hadoop</groupId>
            <artifactId>hadoop-hdfs</artifactId>
            <version>2.7.3</version>
        </dependency>
        <!-- https://mvnrepository.com/artifact/org.apache.hadoop/hadoop-mapreduce-client-common -->
        <dependency>
            <groupId>org.apache.hadoop</groupId>
            <artifactId>hadoop-mapreduce-client-common</artifactId>
            <version>2.7.3</version>
        </dependency>
        <!-- https://mvnrepository.com/artifact/org.apache.hadoop/hadoop-mapreduce-client-core -->
        <dependency>
            <groupId>org.apache.hadoop</groupId>
            <artifactId>hadoop-mapreduce-client-core</artifactId>
            <version>2.7.3</version>
        </dependency>
    </dependencies>
</project>
```

然后创建源码目录 `src/main/java`

现在项目目录的文件结构

```
├── pom.xml
└── src
    └── main
        └── java
```

### 4.2.2 代码

**自定义bean：** src/main/java/FlowBean

```
import java.io.DataInput;
import java.io.DataOutput;
import java.io.IOException;
```

```java
import org.apache.hadoop.io.Writable;

public class FlowBean implements Writable {
    private long upFlow;
    private long dFlow;
    private long sumFlow;

    public FlowBean(){

    }

    public FlowBean(long upFlow, long dFlow){
        this.upFlow = upFlow;
        this.dFlow = dFlow;
        this.sumFlow = upFlow + dFlow;
    }

    public long getUpFlow() {
        return upFlow;
    }

    public void setUpFlow(long upFlow) {
        this.upFlow = upFlow;
    }

    public long getdFlow() {
        return dFlow;
    }

    public void setdFlow(long dFlow) {
        this.dFlow = dFlow;
    }

    public long getSumFlow() {
        return sumFlow;
    }

    public void setSumFlow(long sumFlow) {
        this.sumFlow = sumFlow;
    }

    public void write(DataOutput out) throws IOException {
        out.writeLong(upFlow);
        out.writeLong(dFlow);
        out.writeLong(sumFlow);
    }
```

```
    public void readFields(DataInput in) throws IOException {
        upFlow = in.readLong();
        dFlow = in.readLong();
        sumFlow = in.readLong();
    }

    @Override
    public String toString() {

        return upFlow + "\t" + dFlow + "\t" + sumFlow;
    }
}
```

**mapreduce程序：** src/main/java/FlowCount

```
import java.io.IOException;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.LongWritable;
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 FlowCount {
    static class FlowCountMapper extends Mapper<LongWritable, T
ext, Text, FlowBean> {
        @Override
        protected void map(LongWritable key, Text value, Mapper
<LongWritable, Text, Text, FlowBean>.Context context)
                throws IOException, InterruptedException {

            // 将一行内容转成string
            String line = value.toString();
            // 切分字段
            String[] fields = line.split("\t");
            // 取出手机号
            String phoneNbr = fields[0];
            // 取出上行流量下行流量
            long upFlow = Long.parseLong(fields[1]);
            long dFlow = Long.parseLong(fields[2]);
```

```
            context.write(new Text(phoneNbr), new FlowBean(upFl
ow, dFlow));
        }
    }

    static class FlowCountReducer extends Reducer<Text, FlowBea
n, Text, FlowBean> {
        @Override
        protected void reduce(Text key, Iterable<FlowBean> valu
es,
                Reducer<Text, FlowBean, Text, FlowBean>.Context
 context) throws IOException, InterruptedException {

            long sum_upFlow = 0;
            long sum_dFlow = 0;

            // 遍历所有bean，将其中的上行流量，下行流量分别累加
            for (FlowBean bean : values) {
                sum_upFlow += bean.getUpFlow();
                sum_dFlow += bean.getdFlow();
            }

            FlowBean resultBean = new FlowBean(sum_upFlow, sum_
dFlow);
            context.write(key, resultBean);
        }
    }

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

        Configuration conf = new Configuration();
        Job job = Job.getInstance(conf);

        // 指定本程序的jar包所在的本地路径
        job.setJarByClass(FlowCount.class);

        // 指定本业务job要使用的mapper/Reducer业务类
        job.setMapperClass(FlowCountMapper.class);
        job.setReducerClass(FlowCountReducer.class);

        //指定mapper输出数据的kv类型
        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(FlowBean.class);

        //指定最终输出的数据的kv类型
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(FlowBean.class);
```

```
        //指定job的输入原始文件所在目录
        FileInputFormat.setInputPaths(job, new Path(args[0]));
        //指定job的输出结果所在目录
        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        //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
        /*job.submit();*/
        boolean res = job.waitForCompletion(true);
        System.exit(res?0:1);
    }
}
```

### 4.2.3 编译打包

在 pom.xml 所在目录下执行打包命令：

```
mvn package
```

执行完成后，会自动生成 `target` 目录，其中有打包好的 jar 文件

现在项目文件结构

```
├── pom.xml
├── src
│   └── main
│       └── java
│           ├── FlowBean.java
│           └── FlowCount.java
└── target
    ├── ...
    ├── mapreduce-serializebean-0.0.1-SNAPSHOT.jar
```

### 4.2.4 运行

先把 target 中的 jar 上传到 hadoop 服务器

然后下载测试数据文件：

```
链接: https://pan.baidu.com/s/1skTABlr 密码: tjwy
```

上传到 hdfs

```
hdfs dfs -mkdir -p /flowcount/input
hdfs dfs -put flowdata.log /flowcount/input
```

运行

```
hadoop jar mapreduce-serializebean-0.0.1-SNAPSHOT.jar FlowCount
 /flowcount/input /flowcount/output2
```

检查

```
hdfs dfs -cat /flowcount/output/*
```

# 5. 实例2 - 自定义分区

## 5.1 需求与实现思路

### 5.1.1 需求

还以上个例子的手机用户流量日志为例

| 手机号 | 上行流量 | 下行流量 |
| --- | --- | --- |
| 13726230501 | 200 | 1100 |
| 13396230502 | 300 | 1200 |
| 13897230503 | 400 | 1300 |
| 13897230503 | 100 | 300 |
| 13597230534 | 500 | 1400 |
| 13597230534 | 300 | 1200 |

在上个例子的统计需求基础上添加一个新需求：按省份统计，不同省份的手机号放到不同的文件里

例如 137 表示属于河北，138 属于河南，那么在结果输出时，他们分别在不同的文件中

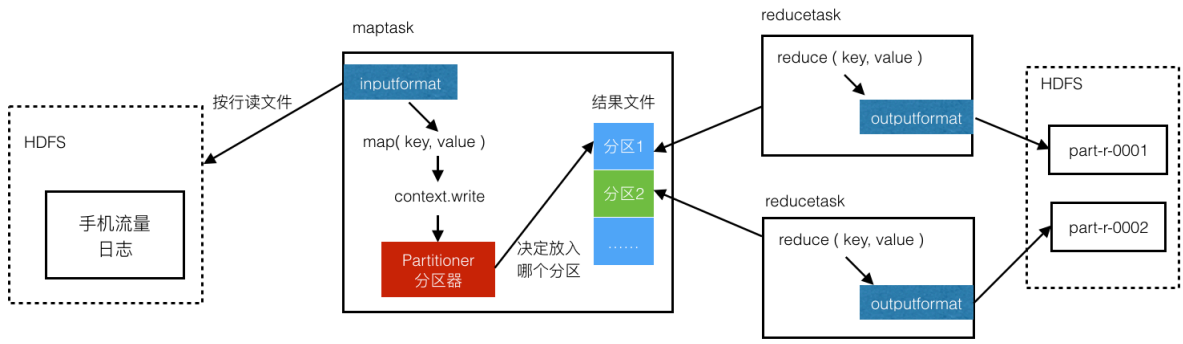
### 5.1.2 实现思路

map 和 reduce 的处理思路与上例相同，这里需要多做2步：

（1）自定义一个分区器 `Partitioner`

根据手机号判断属于哪个分区

有几个分区就有几个 reducetask，每个 reducetask 输出一个文件，那么，不同分区中的数据就写入了不同的结果文件中




（2）在 main 程序中指定使用我们自定义的 Partitioner 即可

## 5.2 代码实践

### 5.2.1 创建项目

新建项目目录 `custom_partion`，其中新建文件 `pom.xml`，内容:

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="h
ttp://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http:
//maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>demo.mr</groupId>
    <artifactId>mapreduce-custompartion</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>jar</packaging>
```

```
    <name>mapreduce-custompartion</name>
    <url>http://maven.apache.org</url>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sour
ceEncoding>
    </properties>

    <dependencies>
        <!-- https://mvnrepository.com/artifact/commons-beanuti
ls/commons-beanutils -->
        <dependency>
            <groupId>commons-beanutils</groupId>
            <artifactId>commons-beanutils</artifactId>
            <version>1.9.3</version>
        </dependency>

        <!-- https://mvnrepository.com/artifact/org.apache.hado
op/hadoop-common -->
        <dependency>
            <groupId>org.apache.hadoop</groupId>
            <artifactId>hadoop-common</artifactId>
            <version>2.7.3</version>
        </dependency>
        <!-- https://mvnrepository.com/artifact/org.apache.hado
op/hadoop-hdfs -->
        <dependency>
            <groupId>org.apache.hadoop</groupId>
            <artifactId>hadoop-hdfs</artifactId>
            <version>2.7.3</version>
        </dependency>
        <!-- https://mvnrepository.com/artifact/org.apache.hado
op/hadoop-mapreduce-client-common -->
        <dependency>
            <groupId>org.apache.hadoop</groupId>
            <artifactId>hadoop-mapreduce-client-common</artifac
tId>
            <version>2.7.3</version>
        </dependency>
        <!-- https://mvnrepository.com/artifact/org.apache.hado
op/hadoop-mapreduce-client-core -->
        <dependency>
            <groupId>org.apache.hadoop</groupId>
            <artifactId>hadoop-mapreduce-client-core</artifactI
d>
            <version>2.7.3</version>
        </dependency>
```

```
    </dependencies>
</project>
```

然后创建源码目录 `src/main/java`

现在项目目录的文件结构

```
├── pom.xml
└── src
    └── main
        └── java
```

### 5.2.2 代码

**自定义bean:** src/main/java/FlowBean.java

```
import java.io.DataInput;
import java.io.DataOutput;
import java.io.IOException;

import org.apache.hadoop.io.Writable;

public class FlowBean implements Writable {
    private long upFlow;
    private long dFlow;
    private long sumFlow;

    public FlowBean(){

    }

    public FlowBean(long upFlow, long dFlow){
        this.upFlow = upFlow;
        this.dFlow = dFlow;
        this.sumFlow = upFlow + dFlow;
    }

    public long getUpFlow() {
        return upFlow;
    }

    public void setUpFlow(long upFlow) {
        this.upFlow = upFlow;
    }
```

```
    public long getdFlow() {
        return dFlow;
    }

    public void setdFlow(long dFlow) {
        this.dFlow = dFlow;
    }

    public long getSumFlow() {
        return sumFlow;
    }

    public void setSumFlow(long sumFlow) {
        this.sumFlow = sumFlow;
    }

    public void write(DataOutput out) throws IOException {
        out.writeLong(upFlow);
        out.writeLong(dFlow);
        out.writeLong(sumFlow);
    }

    public void readFields(DataInput in) throws IOException {
        upFlow = in.readLong();
        dFlow = in.readLong();
        sumFlow = in.readLong();
    }

    @Override
    public String toString() {

        return upFlow + "\t" + dFlow + "\t" + sumFlow;
    }
}
```

**自定义分区器：**src/main/java/ProvincePartitioner.java

```
import java.util.HashMap;

import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Partitioner;

public class ProvincePartitioner extends Partitioner<Text, Flow
Bean>{
```

```
    public static HashMap<String, Integer> proviceDict = new Ha
shMap<String, Integer>();
    static{
        proviceDict.put("137", 0);
        proviceDict.put("133", 1);
        proviceDict.put("138", 2);
        proviceDict.put("135", 3);
    }

    @Override
    public int getPartition(Text key, FlowBean value, int numPa
rtitions) {
        String prefix = key.toString().substring(0, 3);
        Integer provinceId = proviceDict.get(prefix);

        return provinceId==null?4:provinceId;
    }

}
```

这段代码是本示例的重点，其中定义了一个 hashmap，假设其是一个数据库，定义了手机号和分区的关系

`getPartition` 取得手机号的前缀，到数据库中获取区号，如果没在数据库中，就指定其为‘其他分区’（用4代表）

**mapreduce程序：**src/main/java/FlowCount.java

```
import java.io.IOException;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.LongWritable;
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 FlowCount {
    static class FlowCountMapper extends Mapper<LongWritable, T
ext, Text, FlowBean> {
        @Override
        protected void map(LongWritable key, Text value, Mapper
```

```
<LongWritable, Text, Text, FlowBean>.Context context)
                throws IOException, InterruptedException {

            // 将一行内容转成string
            String line = value.toString();
            // 切分字段
            String[] fields = line.split("\t");
            // 取出手机号
            String phoneNbr = fields[0];
            // 取出上行流量下行流量
            long upFlow = Long.parseLong(fields[1]);
            long dFlow = Long.parseLong(fields[2]);

            context.write(new Text(phoneNbr), new FlowBean(upFl
ow, dFlow));
        }
    }

    static class FlowCountReducer extends Reducer<Text, FlowBea
n, Text, FlowBean> {
        @Override
        protected void reduce(Text key, Iterable<FlowBean> valu
es,
                Reducer<Text, FlowBean, Text, FlowBean>.Context
 context) throws IOException, InterruptedException {

            long sum_upFlow = 0;
            long sum_dFlow = 0;

            // 遍历所有bean，将其中的上行流量，下行流量分别累加
            for (FlowBean bean : values) {
                sum_upFlow += bean.getUpFlow();
                sum_dFlow += bean.getdFlow();
            }

            FlowBean resultBean = new FlowBean(sum_upFlow, sum_
dFlow);
            context.write(key, resultBean);
        }
    }

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

        Configuration conf = new Configuration();
        Job job = Job.getInstance(conf);

        // 指定本程序的jar包所在的本地路径
```

```
        job.setJarByClass(FlowCount.class);

        // 指定本业务job要使用的mapper/Reducer业务类
        job.setMapperClass(FlowCountMapper.class);
        job.setReducerClass(FlowCountReducer.class);

        // 指定我们自定义的数据分区器
        job.setPartitionerClass(ProvincePartitioner.class);
        // 同时指定相应“分区”数量的reducetask
        job.setNumReduceTasks(5);

        // 指定mapper输出数据的kv类型
        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(FlowBean.class);

        // 指定最终输出的数据的kv类型
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(FlowBean.class);

        // 指定job的输入原始文件所在目录
        FileInputFormat.setInputPaths(job, new Path(args[0]));
        // 指定job的输出结果所在目录
        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        // 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
        /* job.submit(); */
        boolean res = job.waitForCompletion(true);
        System.exit(res ? 0 : 1);
    }
}
```

main 程序中指定了使用自定义的分区器

```
job.setPartitionerClass(ProvincePartitioner.class);
```

### 5.2.3 编译打包

在 pom.xml 所在目录下执行打包命令：

```
mvn package
```

执行完成后，会自动生成 `target` 目录，其中有打包好的 jar 文件

现在项目文件结构

```
├── pom.xml
├── src
│   └── main
│       └── java
│           ├── FlowBean.java
│           ├── FlowCount.java
│           └── ProvincePartitioner.java
└── target
    ├── ...
    ├── mapreduce-custompartion-0.0.1-SNAPSHOT.jar
```

### 5.2.4 运行

先把 target 中的 jar 上传到 hadoop 服务器

运行

```
hadoop jar mapreduce-custompartion-0.0.1-SNAPSHOT.jar FlowCount
 /flowcount/input /flowcount/output-part
```

检查

```
hdfs dfs -ls /flowcount/output-part
```

# 6. 实例3 - 计算出每组订单中金额最大的记录

## 6.1 需求与实现思路

### 6.1.1 需求

有如下订单数据：

| 订单id | 商品id | 成交金额 |
| --- | --- | --- |
| Order_0000001 | Pdt_01 | 222.8 |
| Order_0000001 | Pdt_05 | 25.8 |

| | | |
|---|---|---|
| Order_0000002 | Pdt_03 | 522.8 |
| Order_0000002 | Pdt_04 | 122.4 |
| Order_0000003 | Pdt_01 | 222.8 |

需要求出每一个订单中成交金额最大的一笔交易

### 6.1.2 实现思路

先介绍一个概念 `GroupingComparator` 组比较器

通过 wordcount 来理解他的作用

wordcount 中 map 处理完成后的结果数据是这样的：

```
<good,1>
<good,1>
<good,1>
<is,1>
<is,1>
```

reducer 会把这些数据都读进来，然后进行分组，把 key 相同的放在一组，形成这样的形式：

```
<good, [1,1,1]>
<is, [1,1]>
```

然后对每一组数据调用一次 `reduce( key, Iterable, ...)` 方法

其中分组的操作就需要用到 `GroupingComparator`，对 key 进行比较，相同的放在一组

> 上例中的 `Partitioner` 是属于 `map` 端的，`GroupingComparator` 是属于 `reduce` 端的

下面看整体实现思路

（1）定义一个订单bean

属性包括：订单号、金额

```
{ itemid, amount }
```

要实现可序列化，与比较方法`compareTo`，比较规则：订单号不同的，按照订单号比较，相同的，按照金额比较

（2）定义一个 Partitioner

根据订单号的hashcode分区，可以保证订单号相同的在同一个分区，以便 reduce 中接收到同一个订单的全部记录

同分区的数据是有序的，这就用到了 bean 中的比较方法，可以让订单号相同的记录按照金额从大到小排序

在 map 方法中输出数据时，key 就是 bean，value 为 null

map的结果数据形式例如：

| 分区1 | 分区2 |
| --- | --- |
| <{ Order_0000001，222.8 }, null> | <{ Order_0000002，522.8 }, null> |
| <{ Order_0000001，25.8 }, null> | <{ Order_0000002，122.4 }, null> |
| <{ Order_0000003，222.8 }, null> | |

（3）定义一个 GroupingComparator

因为 map 的结果数据中 key 是 bean，不是普通数据类型，所以需要使用自定义的比较器来分组，就使用 bean 中的 订单号 来比较

例如读取到分区1的数据：

```
<{ Order_0000001, 222.8 }, null>,
<{ Order_0000001, 25.8 }, null>,
<{ Order_0000003, 222.8 }, null>
```

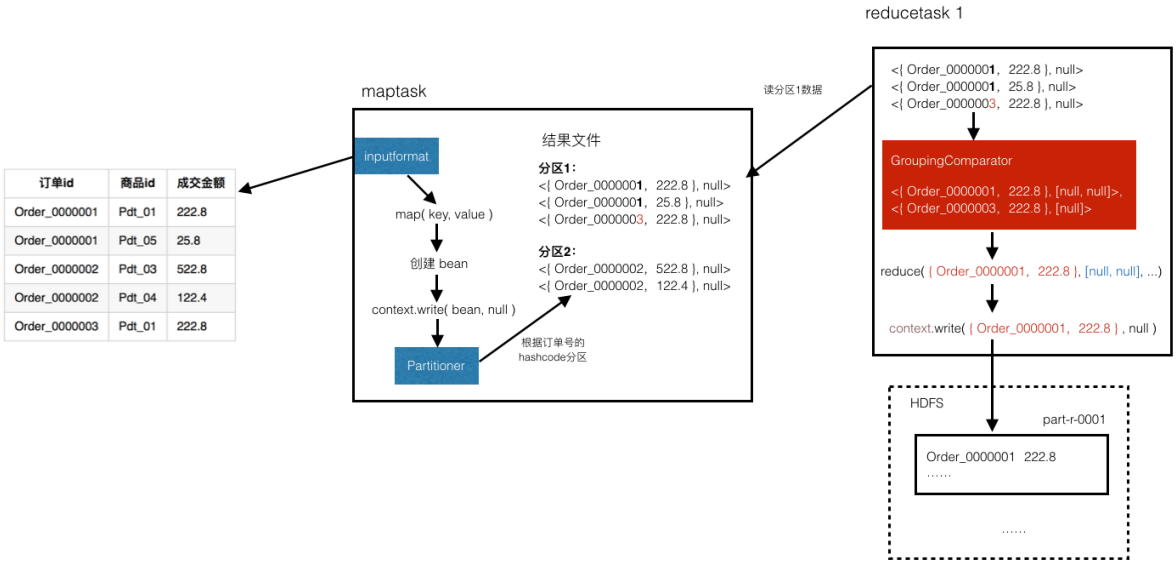
进行比较，前两条数据的订单号相同，放入一组，默认是以第一条记录的 key 作为这组记录的 key

分组后的形式如下：

```
<{ Order_0000001, 222.8 }, [null, null]>,
```

```
<{ Order_0000003, 222.8 }, [null]>
```

在 reduce 方法中收到的每组记录的 key 就是我们最终想要的结果，所以直接输出到文件就可以了




# 6.2 代码实践

## 6.2.1 创建项目

新建项目目录 `groupcomparator`，其中新建文件 `pom.xml`，内容:

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="h
ttp://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http:
//maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>demo.mr</groupId>
    <artifactId>mapreduce-groupcomparator</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>jar</packaging>

    <name>mapreduce-groupcomparator</name>
    <url>http://maven.apache.org</url>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sour
ceEncoding>
    </properties>
```

```
    <dependencies>
        <!-- https://mvnrepository.com/artifact/commons-beanuti
ls/commons-beanutils -->
        <dependency>
            <groupId>commons-beanutils</groupId>
            <artifactId>commons-beanutils</artifactId>
            <version>1.9.3</version>
        </dependency>

        <!-- https://mvnrepository.com/artifact/org.apache.hado
op/hadoop-common -->
        <dependency>
            <groupId>org.apache.hadoop</groupId>
            <artifactId>hadoop-common</artifactId>
            <version>2.7.3</version>
        </dependency>
        <!-- https://mvnrepository.com/artifact/org.apache.hado
op/hadoop-hdfs -->
        <dependency>
            <groupId>org.apache.hadoop</groupId>
            <artifactId>hadoop-hdfs</artifactId>
            <version>2.7.3</version>
        </dependency>
        <!-- https://mvnrepository.com/artifact/org.apache.hado
op/hadoop-mapreduce-client-common -->
        <dependency>
            <groupId>org.apache.hadoop</groupId>
            <artifactId>hadoop-mapreduce-client-common</artifac
tId>
            <version>2.7.3</version>
        </dependency>
        <!-- https://mvnrepository.com/artifact/org.apache.hado
op/hadoop-mapreduce-client-core -->
        <dependency>
            <groupId>org.apache.hadoop</groupId>
            <artifactId>hadoop-mapreduce-client-core</artifactI
d>
            <version>2.7.3</version>
        </dependency>
    </dependencies>
</project>
```

然后创建源码目录 `src/main/java`

现在项目目录的文件结构

```
├── pom.xml
└── src
    └── main
        └── java
```

### 6.2.2 代码

**自定义bean: ** src/main/java/OrderBean.java

```
import java.io.DataInput;
import java.io.DataOutput;
import java.io.IOException;

import org.apache.hadoop.io.DoubleWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.io.WritableComparable;

public class OrderBean implements WritableComparable<OrderBean>
 {

    private Text itemid;
    private DoubleWritable amount;

    public OrderBean() {
    }

    public OrderBean(Text id, DoubleWritable amount){
        this.set(id, amount);
    }

    public void set(Text id, DoubleWritable amount){
        this.itemid = id;
        this.amount = amount;
    }

    public Text getItemid() {
        return itemid;
    }

    public void setItemid(Text itemid) {
        this.itemid = itemid;
    }

    public DoubleWritable getAmount() {
```

```
        return amount;
    }

    public void setAmount(DoubleWritable amount) {
        this.amount = amount;
    }

    public void readFields(DataInput in) throws IOException {
        this.itemid = new Text(in.readUTF());
        this.amount = new DoubleWritable(in.readDouble());
    }

    public void write(DataOutput out) throws IOException {
        out.writeUTF(itemid.toString());
        out.writeDouble(amount.get());
    }

    public int compareTo(OrderBean o) {
        int ret = this.itemid.compareTo(o.getItemid());
        if(ret == 0){
            ret = -this.amount.compareTo(o.getAmount());
        }
        return ret;
    }
    @Override
    public String toString() {
        return itemid.toString() + "\t" + amount.get();
    }
}
```

**自定义分区器：** src/main/java/ItemIdPartitioner.java

```
import org.apache.hadoop.io.NullWritable;
import org.apache.hadoop.mapreduce.Partitioner;

public class ItemIdPartitioner extends Partitioner<OrderBean, N
ullWritable>{

    @Override
    public int getPartition(OrderBean bean, NullWritable value,
 int numReduceTasks) {
        //相同id的订单bean，会发往相同的partition
        //而且，产生的分区数，是会跟用户设置的reduce task数保持一致
        return (bean.getItemid().hashCode() & Integer.MAX_VALUE
) % numReduceTasks;
    }
```

```
}
```

**自定义比较器：** src/main/java/MyGroupingComparator.java

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

public class MyGroupingComparator extends WritableComparator {
    public MyGroupingComparator() {
        super(OrderBean.class, true);
    }

    @Override
    public int compare(WritableComparable a, WritableComparable
 b) {
        OrderBean ob1 = (OrderBean)a;
        OrderBean ob2 = (OrderBean)b;
        return ob1.getItemid().compareTo(ob2.getItemid());
    }
}
```

**mapreduce程序：** src/main/java/GroupSort.java

```
import java.io.IOException;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.DoubleWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.NullWritable;
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 GroupSort {
    static class SortMapper extends Mapper<LongWritable, Text,
OrderBean, NullWritable> {
        OrderBean bean = new OrderBean();

        @Override
```

```
        protected void map(LongWritable key, Text value, Contex
t context) throws IOException, InterruptedException {

            String line = value.toString();
            String[] fields = line.split(",");
            bean.set(new Text(fields[0]), new DoubleWritable(Do
uble.parseDouble(fields[2])));
            context.write(bean, NullWritable.get());
        }
    }

    static class SortReducer extends Reducer<OrderBean, NullWri
table, OrderBean, NullWritable> {
        @Override
        protected void reduce(OrderBean key, Iterable<NullWrita
ble> val, Context context)
                throws IOException, InterruptedException {

            context.write(key, NullWritable.get());
        }
    }

    public static void main(String[] args) throws Exception {
        // 创建任务
        Configuration conf = new Configuration();
        Job job = Job.getInstance(conf);
        job.setJarByClass(GroupSort.class);

        // 任务输出类型
        job.setOutputKeyClass(OrderBean.class);
        job.setOutputValueClass(NullWritable.class);

        // 指定 map reduce
        job.setMapperClass(SortMapper.class);
        job.setReducerClass(SortReducer.class);

        job.setGroupingComparatorClass(MyGroupingComparator.cla
ss);
        job.setPartitionerClass(ItemIdPartitioner.class);
        job.setNumReduceTasks(2);

        // 输入文件路径、输出路径
        FileInputFormat.setInputPaths(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        // 提交任务
        job.waitForCompletion(true);
```

```
    }
}
```

### 6.2.3 编译打包

在 pom.xml 所在目录下执行打包命令：

```
mvn package
```

执行完成后，会自动生成 `target` 目录，其中有打包好的 jar 文件

现在项目文件结构

```
├── pom.xml
├── src
│   └── main
│       └── java
│           ├── GroupSort.java
│           ├── ItemIdPartitioner.java
│           ├── MyGroupingComparator.java
│           └── OrderBean.java
└── target
    ├── ...
    ├── mapreduce-groupcomparator-0.0.1-SNAPSHOT.jar
```

### 6.2.4 运行

先把 target 中的 jar 上传到 hadoop 服务器

下载测试数据文件

```
链接: https://pan.baidu.com/s/1pKKlvh5 密码: 43xa
```

上传到 hdfs

```
hdfs dfs -put orders.txt /
```

运行

```
hadoop jar mapreduce-groupcomparator-0.0.1-SNAPSHOT.jar GroupSo
rt /orders.txt /outputOrders
```

检查

```
hdfs dfs -ls /outputOrders
hdfs dfs -cat /outputOrders/*
```

# 7. 实例4 - 合并多个小文件

## 7.1 需求与实现思路

### 7.1.1 需求

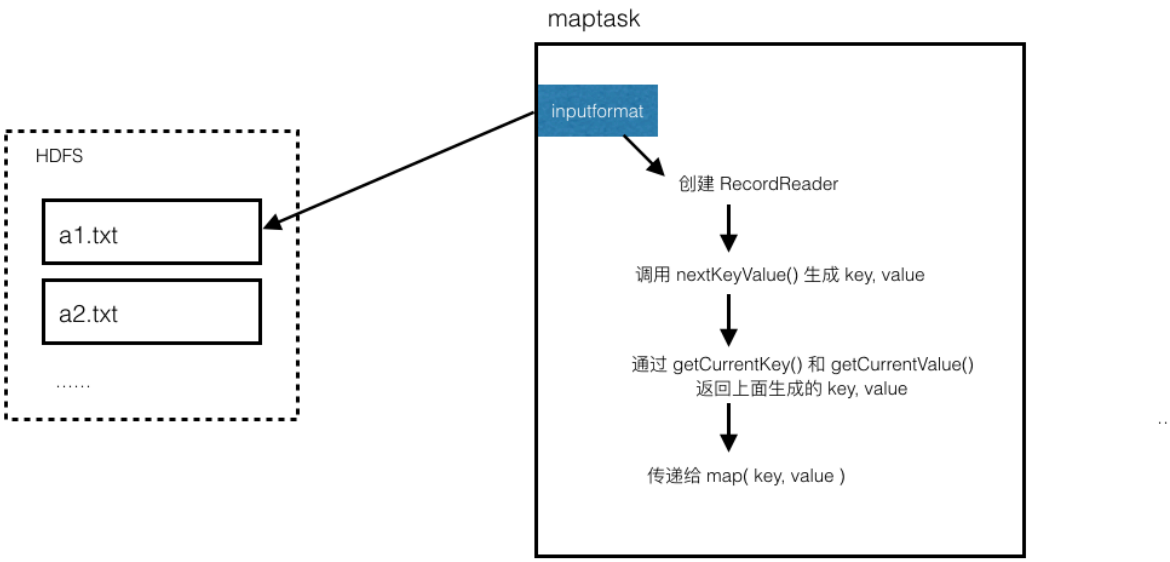
要计算的目标文件中有大量的小文件，会造成分配任务和资源的开销比实际的计算开销还大，这就产生了效率损耗

需要先把一些小文件合并成一个大文件

### 7.1.2 实现思路

文件的读取由 map 端负责，在前面的示意图中可以看到一个 `inputformat` 用来读取文件，然后以 key value 形式传递给 map 方法

我们要自定义文件的读取过程，就需要了解其细节流程：




所以我们需要自定义一个 `inputformat` 和 `RecordReader`

`inputformat` 使用我们自己的 `RecordReader`， `RecordReader` 负责实现一次读取一个完整文件封装为 key value

map 接收到文件内容，然后以文件名为 key，以文件内容为 value，向外输出

reduce 的任务非常简单，收到什么输出什么就可以了，所以可以省略，但是输出的格式要注意，要使用 `SequenceFileOutPutFormat`（用来输出对象）

因为 reduce 收到的 key value 都是对象，不是普通的文本，reduce 默认的输出格式是 TextOutputFormat，使用他的话，最终输出的内容就是对象ID，所以要使用 `SequenceFileOutPutFormat` 进行输出

# 7.2 代码实践

## 7.2.1 创建项目

新建项目目录 `inputformat`，其中新建文件 `pom.xml`，内容:

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="h
ttp://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http:
//maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>demo.mr</groupId>
```

```
    <artifactId>mapreduce-inputformat</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>jar</packaging>

    <name>mapreduce-inputformat</name>
    <url>http://maven.apache.org</url>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sour
ceEncoding>
    </properties>

    <dependencies>
        <!-- https://mvnrepository.com/artifact/commons-beanuti
ls/commons-beanutils -->
        <dependency>
            <groupId>commons-beanutils</groupId>
            <artifactId>commons-beanutils</artifactId>
            <version>1.9.3</version>
        </dependency>

        <!-- https://mvnrepository.com/artifact/org.apache.hado
op/hadoop-common -->
        <dependency>
            <groupId>org.apache.hadoop</groupId>
            <artifactId>hadoop-common</artifactId>
            <version>2.7.3</version>
        </dependency>
        <!-- https://mvnrepository.com/artifact/org.apache.hado
op/hadoop-hdfs -->
        <dependency>
            <groupId>org.apache.hadoop</groupId>
            <artifactId>hadoop-hdfs</artifactId>
            <version>2.7.3</version>
        </dependency>
        <!-- https://mvnrepository.com/artifact/org.apache.hado
op/hadoop-mapreduce-client-common -->
        <dependency>
            <groupId>org.apache.hadoop</groupId>
            <artifactId>hadoop-mapreduce-client-common</artifac
tId>
            <version>2.7.3</version>
        </dependency>
        <!-- https://mvnrepository.com/artifact/org.apache.hado
op/hadoop-mapreduce-client-core -->
        <dependency>
            <groupId>org.apache.hadoop</groupId>
```

```
            <artifactId>hadoop-mapreduce-client-core</artifactI
d>
            <version>2.7.3</version>
        </dependency>
    </dependencies>
</project>
```

然后创建源码目录 `src/main/java`

现在项目目录的文件结构

```
├── pom.xml
└── src
    └── main
        └── java
```

### 7.2.2 代码

**自定义inputform：** src/main/java/MyInputFormat.java

```
import java.io.IOException;
import java.io.Reader;

import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.BytesWritable;
import org.apache.hadoop.io.NullWritable;
import org.apache.hadoop.mapreduce.InputSplit;
import org.apache.hadoop.mapreduce.JobContext;
import org.apache.hadoop.mapreduce.RecordReader;
import org.apache.hadoop.mapreduce.TaskAttemptContext;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;

public class MyInputFormat extends FileInputFormat<NullWritable
, BytesWritable> {
    @Override
    protected boolean isSplitable(JobContext context, Path file
name) {
        // 设置每个小文件不可分片,保证一个小文件生成一个key-value键值对
        return false;
    }

    @Override
    public RecordReader<NullWritable, BytesWritable> createReco
rdReader(InputSplit split, TaskAttemptContext context)
```

```
            throws IOException, InterruptedException {

        MyRecordReader recordReader = new MyRecordReader();
        recordReader.initialize(split, context);
        return recordReader;
    }
}
```

`createRecordReader` 方法中创建一个自定义的 reader

**自定义reader：** src/main/java/MyRecordReader.java

```
import java.io.IOException;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FSDataInputStream;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.BytesWritable;
import org.apache.hadoop.io.IOUtils;
import org.apache.hadoop.io.NullWritable;
import org.apache.hadoop.mapreduce.InputSplit;
import org.apache.hadoop.mapreduce.RecordReader;
import org.apache.hadoop.mapreduce.TaskAttemptContext;
import org.apache.hadoop.mapreduce.lib.input.FileSplit;

public class MyRecordReader extends RecordReader<NullWritable,
BytesWritable> {

    private FileSplit fileSplit;
    private Configuration conf;
    private BytesWritable value = new BytesWritable();
    private boolean processed = false;

    @Override
    public void close() throws IOException {
    }

    @Override
    public NullWritable getCurrentKey() throws IOException, Int
erruptedException {
        return NullWritable.get();
    }

    @Override
    public BytesWritable getCurrentValue() throws IOException,
```

```
InterruptedException {
        return value;
    }

    @Override
    public float getProgress() throws IOException, InterruptedE
xception {
        return processed ? 1.0f : 0.0f;
    }

    @Override
    public void initialize(InputSplit split, TaskAttemptContext
 context) throws IOException, InterruptedException {
        this.fileSplit = (FileSplit) split;
        this.conf = context.getConfiguration();
    }

    @Override
    public boolean nextKeyValue() throws IOException, Interrupt
edException {
        if (!processed) {
            byte[] contents = new byte[(int) fileSplit.getLengt
h()];
            Path file = fileSplit.getPath();
            FileSystem fs = file.getFileSystem(conf);
            FSDataInputStream in = null;
            try {
                in = fs.open(file);
                IOUtils.readFully(in, contents, 0, contents.len
gth);
                value.set(contents, 0, contents.length);
            } finally {
                IOUtils.closeStream(in);
            }
            processed = true;
            return true;
        }
        return false;
    }

}
```

其中有3个核心方法：`nextKeyValue`、`getCurrentKey`、`getCurrentValue`

`nextKeyValue` 负责生成要传递给 `map` 方法的 key 和 value

`getCurrentKey`、`getCurrentValue` 是实际获取 key 和 value 的

所以 `RecordReader` 的核心机制就是：通过 nextKeyValue 生成 key value，然后通过 getCurrentKey 和 getCurrentValue 来返回上面构造好的 key value

这里的 `nextKeyValue` 负责把整个文件内容作为 value

**mapreduce程序：** src/main/java/ManyToOne.java

```
import java.io.IOException;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.BytesWritable;
import org.apache.hadoop.io.NullWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.InputSplit;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.input.FileSplit;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.lib.output.SequenceFileOutpu
tFormat;

public class ManyToOne {
    static class FileMapper extends Mapper<NullWritable, BytesW
ritable, Text, BytesWritable> {
        private Text filenameKey;
        @Override
        protected void setup(Context context)
                throws IOException, InterruptedException {

            InputSplit split = context.getInputSplit();
            Path path = ((FileSplit) split).getPath();
            filenameKey = new Text(path.toString());
        }
        @Override
        protected void map(NullWritable key, BytesWritable valu
e, Context context)
                throws IOException, InterruptedException {
            context.write(filenameKey, value);
        }
    }

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

```
        Configuration conf = new Configuration();
        Job job = Job.getInstance(conf);
        job.setJarByClass(ManyToOne.class);

        job.setInputFormatClass(MyInputFormat.class);
        job.setOutputFormatClass(SequenceFileOutputFormat.class
);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(BytesWritable.class);
        job.setMapperClass(FileMapper.class);

        FileInputFormat.setInputPaths(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        job.waitForCompletion(true);
    }
}
```

main 程序中指定使用我们自定义的 `MyInputFormat`，输出使用 `SequenceFileOutputFormat`

### 7.2.3 编译打包

在 pom.xml 所在目录下执行打包命令：

```
mvn package
```

执行完成后，会自动生成 `target` 目录，其中有打包好的 jar 文件

现在项目文件结构

```
├── pom.xml
├── src
│   └── main
│       └── java
│           ├── ManyToOne.java
│           ├── MyInputFormat.java
│           └── MyRecordReader.java
└── target
    ├── ...
    ├── mapreduce-inputformat-0.0.1-SNAPSHOT.jar
```

### 7.2.4 运行

先把 target 中的 jar 上传到 hadoop 服务器

准备测试文件，把hadoop目录中的配置文件上传到 hdfs

```
hdfs dfs -mkdir /files
hdfs dfs -put $HADOOP_HOME/etc/hadoop/*.xml /files
```

运行

```
hadoop jar mapreduce-inputformat-0.0.1-SNAPSHOT.jar ManyToOne /
files /onefile
```

检查

```
hdfs dfs -ls /onefile
```

# 8. 实例5 - 分组输出到多个文件

## 8.1 需求与实现思路

### 8.1.1 需求

有如下订单数据：

| 订单id | 商品id | 成交金额 |
| --- | --- | --- |
| Order_0000001 | Pdt_01 | 222.8 |
| Order_0000001 | Pdt_05 | 25.8 |
| Order_0000002 | Pdt_05 | 325.8 |
| Order_0000002 | Pdt_03 | 522.8 |
| Order_0000002 | Pdt_04 | 122.4 |
| Order_0000003 | Pdt_01 | 222.8 |

| | | |
|---|---|---|
| Order_0000003 | Pdt_01 | 322.8 |

需要把相同订单id的记录放在一个文件中，并以订单id命名

### 8.1.2 实现思路

这个需求可以直接使用 `MultipleOutputs` 这个类来实现

默认情况下，每个 reducer 写入一个文件，文件名由分区号命名，例如 'part-r-00000'，而 MultipleOutputs 可以用 key 作为文件名，例如 ‘Order_0000001-r-00000’

所以，思路就是 map 中处理每条记录，以‘订单id’为 key，reduce 中使用 MultipleOutputs 进行输出，会自动以 key 为文件名，文件内容就是相同 key 的所有记录

例如 ‘Order_0000001-r-00000’ 中的内容就是：

```
Order_0000001,Pdt_05,25.8
Order_0000001,Pdt_01,222.8
```

## 8.2 代码实践

### 8.2.1 创建项目

新建项目目录 `multioutput`，其中新建文件 `pom.xml`，内容:

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="h
ttp://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http:
//maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>demo.mr</groupId>
    <artifactId>mapreduce-multipleOutput</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>jar</packaging>

    <name>mapreduce-multipleOutput</name>
    <url>http://maven.apache.org</url>
```

```
    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sour
ceEncoding>
    </properties>

    <dependencies>
        <!-- https://mvnrepository.com/artifact/commons-beanuti
ls/commons-beanutils -->
        <dependency>
            <groupId>commons-beanutils</groupId>
            <artifactId>commons-beanutils</artifactId>
            <version>1.9.3</version>
        </dependency>

        <!-- https://mvnrepository.com/artifact/org.apache.hado
op/hadoop-common -->
        <dependency>
            <groupId>org.apache.hadoop</groupId>
            <artifactId>hadoop-common</artifactId>
            <version>2.7.3</version>
        </dependency>
        <!-- https://mvnrepository.com/artifact/org.apache.hado
op/hadoop-hdfs -->
        <dependency>
            <groupId>org.apache.hadoop</groupId>
            <artifactId>hadoop-hdfs</artifactId>
            <version>2.7.3</version>
        </dependency>
        <!-- https://mvnrepository.com/artifact/org.apache.hado
op/hadoop-mapreduce-client-common -->
        <dependency>
            <groupId>org.apache.hadoop</groupId>
            <artifactId>hadoop-mapreduce-client-common</artifac
tId>
            <version>2.7.3</version>
        </dependency>
        <!-- https://mvnrepository.com/artifact/org.apache.hado
op/hadoop-mapreduce-client-core -->
        <dependency>
            <groupId>org.apache.hadoop</groupId>
            <artifactId>hadoop-mapreduce-client-core</artifactI
d>
            <version>2.7.3</version>
        </dependency>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
```

```
            <version>3.8.1</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
</project>
```

然后创建源码目录 `src/main/java`

现在项目目录的文件结构

```
├── pom.xml
└── src
    └── main
        └── java
```

### 8.2.2 代码

**mapreduce程序：** src/main/java/MultipleOutputTest.java

```
import java.io.IOException;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.NullWritable;
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.mapreduce.lib.output.MultipleOutputs;


public class MultipleOutputTest {
    static class MyMapper extends Mapper<LongWritable, Text, Te
xt, Text> {
        @Override
        protected void map(LongWritable key, Text value, Contex
t context) throws IOException, InterruptedException {

            String line = value.toString();
            String[] fields = line.split(",");
            context.write(new Text(fields[0]), value);
```

```
        }
    }

    static class MyReducer extends Reducer<Text, Text, NullWritable, Text> {
        private MultipleOutputs<NullWritable, Text> multipleOutputs;

        protected void setup(Context context) throws IOException, InterruptedException {
            multipleOutputs = new MultipleOutputs<NullWritable, Text>(context);
        }

        @Override
        protected void reduce(Text key, Iterable<Text> values, Context context)
                throws IOException, InterruptedException {

            for (Text value : values) {
                multipleOutputs.write(NullWritable.get(), value, key.toString());

            }
        }

        protected void cleanup(Context context) throws IOException, InterruptedException {
            multipleOutputs.close();
        }
    }

    public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        Job job = Job.getInstance(conf);
        job.setJarByClass(MultipleOutputTest.class);

        job.setMapperClass(MyMapper.class);
        job.setReducerClass(MyReducer.class);

        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(Text.class);

        job.setOutputKeyClass(NullWritable.class);
        job.setOutputValueClass(Text.class);

        FileInputFormat.setInputPaths(job, new Path(args[0]));
```

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

        // 提交任务
        job.waitForCompletion(true);
    }
}
```

### 8.2.3 编译打包

在 pom.xml 所在目录下执行打包命令：

```
mvn package
```

执行完成后，会自动生成 `target` 目录，其中有打包好的 jar 文件

现在项目文件结构

```
├── pom.xml
├── src
│   └── main
│       └── java
│           └── MultipleOutputTest.java
└── target
    ├── ...
    ├── mapreduce-multipleOutput-0.0.1-SNAPSHOT.jar
```

### 8.2.4 运行

先把 target 中的 jar 上传到 hadoop 服务器

然后运行

```
hadoop jar mapreduce-multipleOutput-0.0.1-SNAPSHOT.jar MultipleOutputTest /orders.txt /output-multi
```

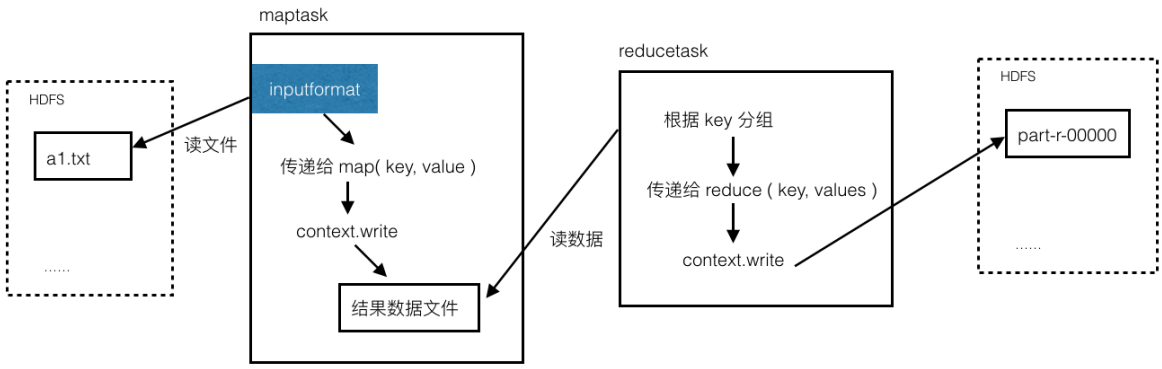
检查

```
hdfs dfs -ls /output-multi
```

# 9. MapReduce 核心流程梳理

我们已经了解了 MapReduce 的大概流程：

（1）maptask 从目标文件中读取数据
（2）mapper 的 map 方法处理每一条数据，输出到文件中
（3）reducer 读取 map 的结果文件，进行分组，把每一组交给 reduce 方法进行处理，最后输出到指定路径



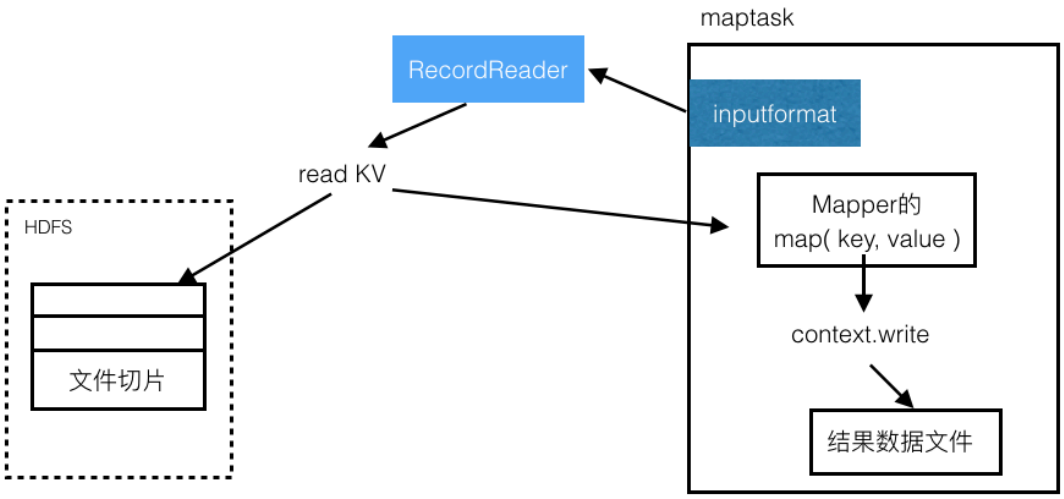

这是最基本的流程，有助于快速理解 MapReduce 的工作方式

通过上面的几个示例，我们已经接触了一些更深入的细节，例如 mapper 的 inputform 中还有 RecordReader、reducer 中还有 GroupingComparator

下面就看一下更加深入的处理流程

## maptask 中的处理流程

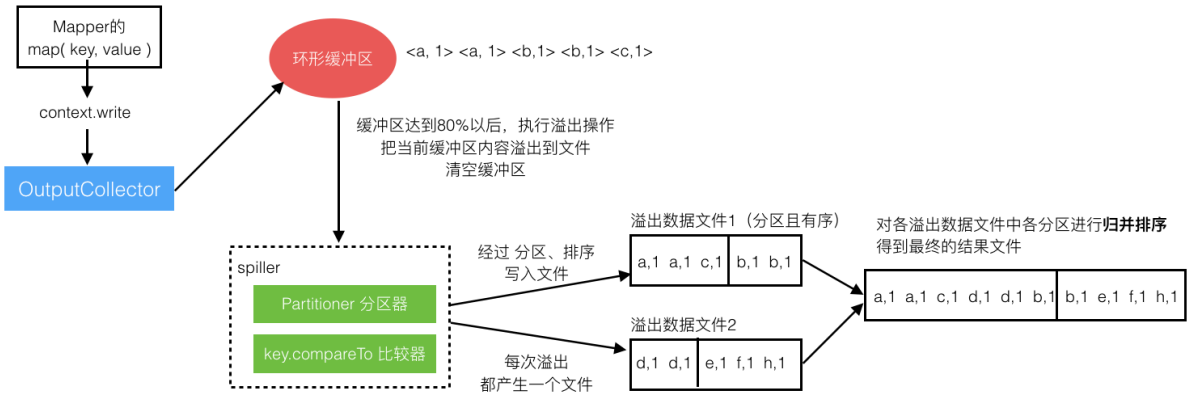
（1）读文件流程




目标文件会被按照规划文件进行切分，inputformat 调用 RecordReader 读取文件切片，RecordReader 会生成 key value 对儿，传递给 Mapper 的 map 方法

（2）写入结果文件的流程

从 Mapper 的 map 方法调用 `context.write` 之后，到形成‘结果数据文件’这个过程是比较复杂的




`context.write` 不是直接写入文件，而是把数据交给 `OutputCollector`，OutputCollector 把数据写入‘环形缓冲区’

‘环形缓冲区’ 中的数据会进行排序

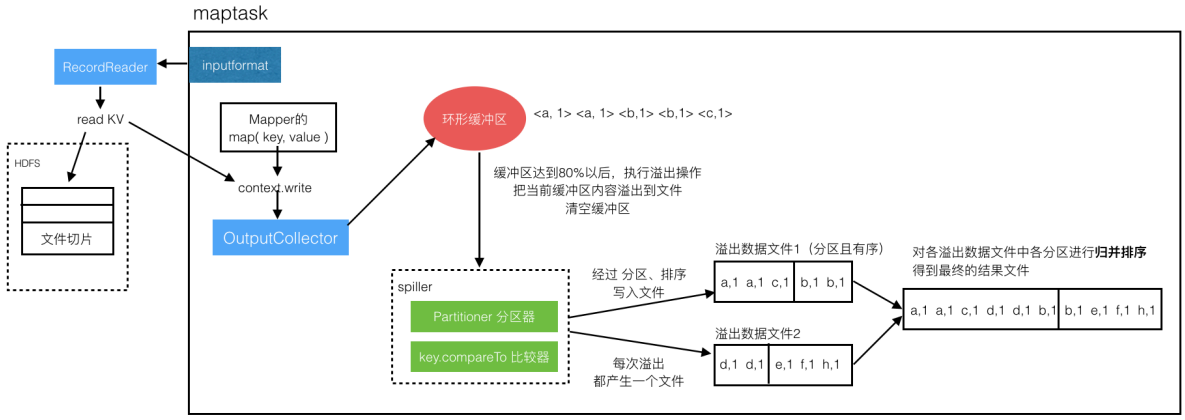
因为缓冲区的大小是有限制的，所以每当快满时（达到80%）就要把其中的数据写出去，这个过程叫做数据溢出

溢出到一个文件中，溢出过程会对这批数据进行分组、比较操作，然后写入文件，所以溢出文件中的数据是分好区的，并且是有序的

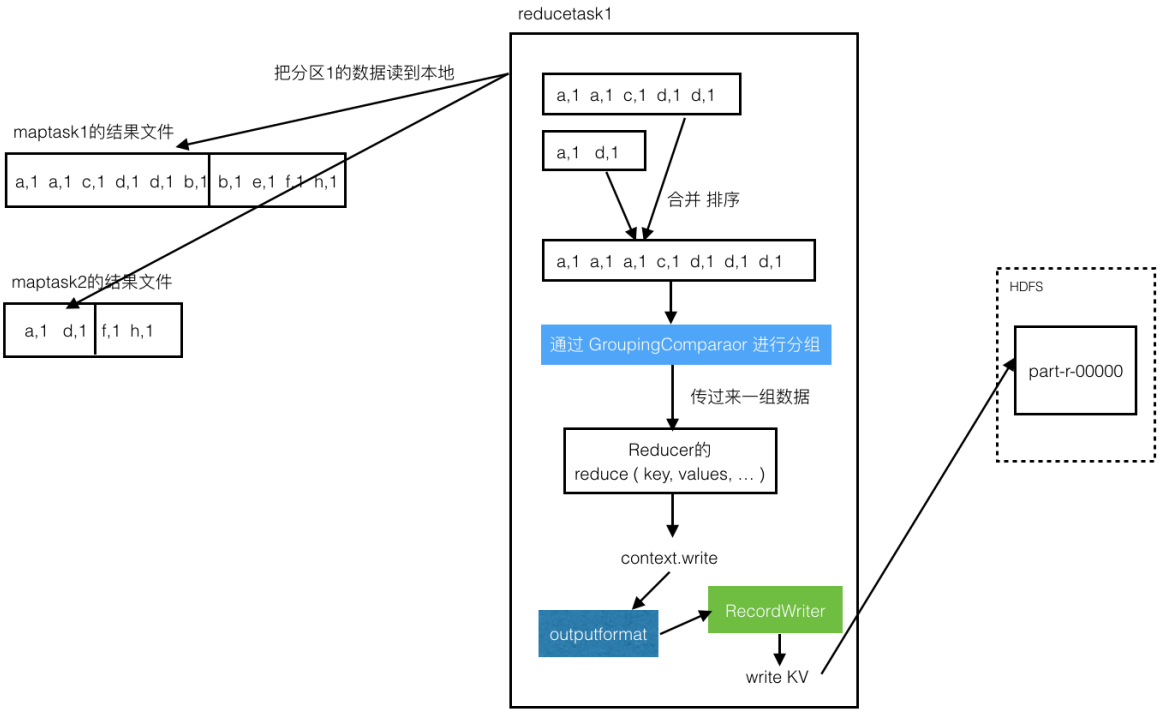
每次溢出都会产生一个溢出数据文件，所以会有多个

当 map 处理完全数据后，就会对各个溢出数据文件进行合并，每个文件中相同区的数据放在一起，并再次排序，最后得到一个整体的结果文件，其中是分区且有序的

这样就完成了map过程，读数据过程和写结果文件的过程联合起来如下图




# reducetask 的处理流程




reducetask 去读每个 maptask 产生的结果文件中自己所负责的分区数据，读到自己本地

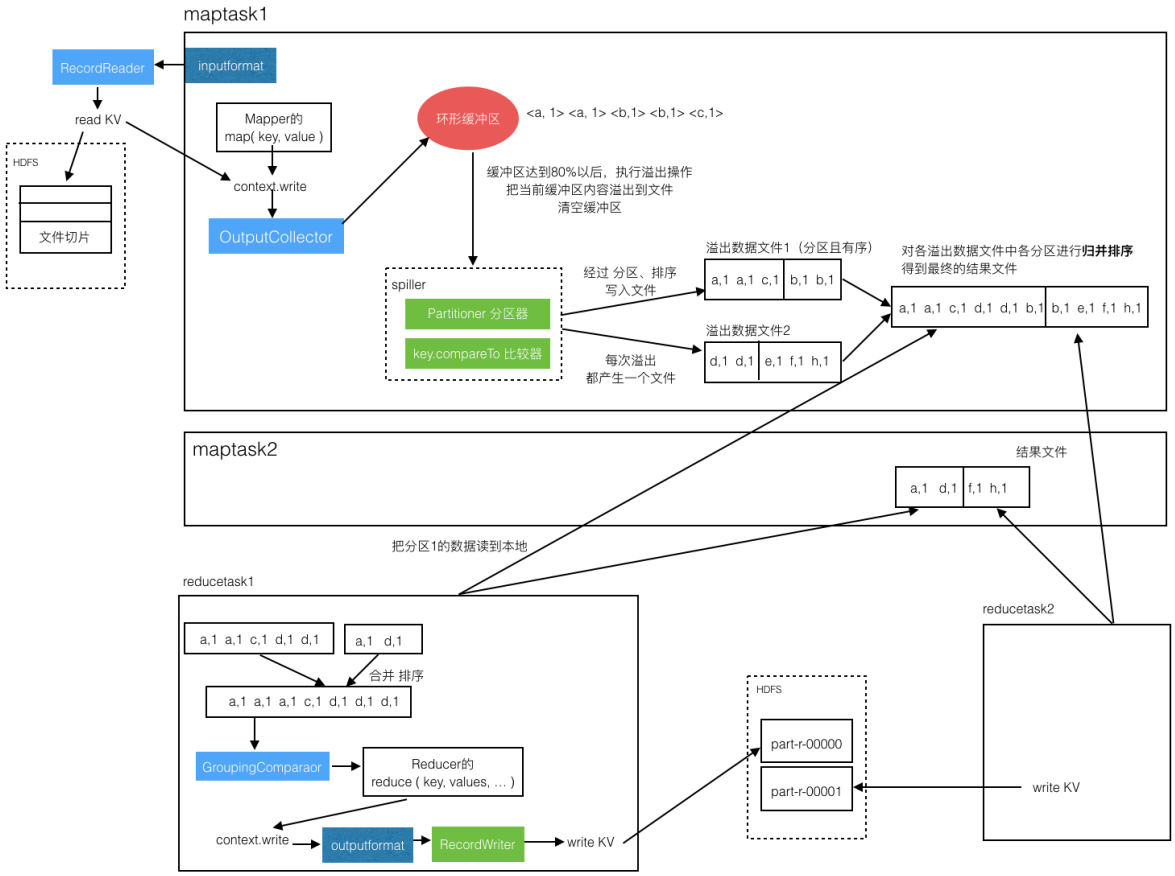
对多个数据文件进行合并排序，然后通过 GroupingComparaor 进行分组，把相

同 key 的数据放到一组

对每组数据调一次 reduce 方法，处理完成后写入目标路径文件

## 整体流程

把 map 和 reduce 的过程联合起来




# 10. 实例6 - join 操作

## 10.1 需求与实现思路

### 10.1.1 需求

有2个数据文件：订单数据、商品信息

订单数据表 order

| id | date | pid | amount |
| --- | --- | --- | --- |
| | | | |

| | | | |
|---|---|---|---|
| 1001 | 20170310 | P0001 | 2 |
| 1002 | 20170410 | P0001 | 3 |
| 1002 | 20170410 | P0002 | 3 |

商品信息表 product

| id | pname | category_id | price |
|---|---|---|---|
| P0001 | 小米5 | 1000 | 2 |
| P0002 | 锤子T1 | 1000 | 3 |

需要用 mapreduce 程序来实现下面这个SQL查询运算：

```
select o.id order_id, o.date, o.amount, p.id p_id, p.pname, p.c
ategory_id, p.price
from t_order o join t_product p on o.pid = p.id
```

### 10.1.2 实现思路

SQL的执行结果是这样的：

| order_id | date | p_id | pname | category_id | price |
|---|---|---|---|---|---|
| 1001 | 20170310 | P0001 | 小米5 | 1000 | 2 |
| 1002 | 20170410 | P0001 | 小米5 | 1000 | 2 |
| 1002 | 20170410 | P0002 | 锤子T1 | 1000 | 3 |

实际上就是给每条订单记录补充上商品表中的信息

实现思路：

（1）定义bean

把SQL执行结果中的各列封装成一个bean对象，实现序列化

bean中还要有一个另外的属性 flag，用来标识此对象的数据是订单还是商品

（2）map 处理

map 会处理两个文件中的数据，根据文件名可以知道当前这条数据是订单还是商品

对每条数据创建一个 bean 对象，设置对应的属性，并标识 flag（ 0 代表 order, 1 代表 product）

以join的关联项 'productid' 为 key，bean 为 value 进行输出

（3）reduce 处理

reduce 方法接收到 pid 相同的一组 bean 对象

遍历 bean 对象集合，如果 bean 是订单数据，就放入一个新的订单集合中，如果是商品数据，就保存到一个商品bean中

然后遍历那个新的订单集合，使用商品bean的数据对每个订单bean进行信息补全

这样就得到了完整的订单及其商品信息

# 10.2 代码实践

## 10.2.1 创建项目

新建项目目录 `jointest`，其中新建文件 `pom.xml`，内容:

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="h
ttp://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http:
//maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>demo.mr</groupId>
    <artifactId>mapreduce-jointest</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>jar</packaging>

    <name>mapreduce-jointest</name>
    <url>http://maven.apache.org</url>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sour
ceEncoding>
    </properties>
```

```
    <dependencies>
        <!-- https://mvnrepository.com/artifact/commons-beanuti
ls/commons-beanutils -->
        <dependency>
            <groupId>commons-beanutils</groupId>
            <artifactId>commons-beanutils</artifactId>
            <version>1.9.3</version>
        </dependency>

        <!-- https://mvnrepository.com/artifact/org.apache.hado
op/hadoop-common -->
        <dependency>
            <groupId>org.apache.hadoop</groupId>
            <artifactId>hadoop-common</artifactId>
            <version>2.7.3</version>
        </dependency>
        <!-- https://mvnrepository.com/artifact/org.apache.hado
op/hadoop-hdfs -->
        <dependency>
            <groupId>org.apache.hadoop</groupId>
            <artifactId>hadoop-hdfs</artifactId>
            <version>2.7.3</version>
        </dependency>
        <!-- https://mvnrepository.com/artifact/org.apache.hado
op/hadoop-mapreduce-client-common -->
        <dependency>
            <groupId>org.apache.hadoop</groupId>
            <artifactId>hadoop-mapreduce-client-common</artifac
tId>
            <version>2.7.3</version>
        </dependency>
        <!-- https://mvnrepository.com/artifact/org.apache.hado
op/hadoop-mapreduce-client-core -->
        <dependency>
            <groupId>org.apache.hadoop</groupId>
            <artifactId>hadoop-mapreduce-client-core</artifactI
d>
            <version>2.7.3</version>
        </dependency>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>3.8.1</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
```

```
</project>
```

然后创建源码目录 `src/main/java`

现在项目目录的文件结构

```
├── pom.xml
└── src
    └── main
        └── java
```

## 10.2.2 代码

**封装bean: ** src/main/java/InfoBean.java

```
import java.io.DataInput;
import java.io.DataOutput;
import java.io.IOException;

import org.apache.hadoop.io.Writable;

public class InfoBean implements Writable {

    private int order_id;
    private String dateString;
    private String p_id;
    private int amount;
    private String pname;
    private int category_id;
    private float price;

    // flag=0表示这个对象是封装订单表记录
    // flag=1表示这个对象是封装产品信息记录
    private String flag;

    public InfoBean() {
    }

    public void set(int order_id, String dateString, String p_i
d, int amount, String pname, int category_id, float price, Stri
ng flag) {
        this.order_id = order_id;
        this.dateString = dateString;
        this.p_id = p_id;
```

```
        this.amount = amount;
        this.pname = pname;
        this.category_id = category_id;
        this.price = price;
        this.flag = flag;
    }

    public int getOrder_id() {
        return order_id;
    }

    public void setOrder_id(int order_id) {
        this.order_id = order_id;
    }

    public String getDateString() {
        return dateString;
    }

    public void setDateString(String dateString) {
        this.dateString = dateString;
    }

    public String getP_id() {
        return p_id;
    }

    public void setP_id(String p_id) {
        this.p_id = p_id;
    }

    public int getAmount() {
        return amount;
    }

    public void setAmount(int amount) {
        this.amount = amount;
    }

    public String getPname() {
        return pname;
    }

    public void setPname(String pname) {
        this.pname = pname;
    }
```

```
    public int getCategory_id() {
        return category_id;
    }

    public void setCategory_id(int category_id) {
        this.category_id = category_id;
    }

    public float getPrice() {
        return price;
    }

    public void setPrice(float price) {
        this.price = price;
    }

    public String getFlag() {
        return flag;
    }

    public void setFlag(String flag) {
        this.flag = flag;
    }

    /**
     * private int order_id; private String dateString; private
 int p_id;
     * private int amount; private String pname; private int ca
tegory_id;
     * private float price;
     */
    public void write(DataOutput out) throws IOException {
        out.writeInt(order_id);
        out.writeUTF(dateString);
        out.writeUTF(p_id);
        out.writeInt(amount);
        out.writeUTF(pname);
        out.writeInt(category_id);
        out.writeFloat(price);
        out.writeUTF(flag);

    }

    public void readFields(DataInput in) throws IOException {
        this.order_id = in.readInt();
        this.dateString = in.readUTF();
        this.p_id = in.readUTF();
```

```
        this.amount = in.readInt();
        this.pname = in.readUTF();
        this.category_id = in.readInt();
        this.price = in.readFloat();
        this.flag = in.readUTF();

    }

    @Override
    public String toString() {
        return "order_id=" + order_id + ", dateString=" + dateS
tring + ", p_id=" + p_id + ", amount=" + amount + ", pname=" +
pname + ", category_id=" + category_id + ", price=" + price + "
, flag=" + flag;
    }
}
```

**mapreduce程序：** src/main/java/JoinMR.java

```
import java.io.IOException;
import java.util.ArrayList;

import org.apache.commons.beanutils.BeanUtils;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.NullWritable;
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.input.FileSplit;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class JoinMR {
    static class JoinMRMapper extends Mapper<LongWritable, Text
, Text, InfoBean> {
        InfoBean bean = new InfoBean();
        Text k = new Text();

        @Override
        protected void map(LongWritable key, Text value, Contex
t context) throws IOException, InterruptedException {

            String line = value.toString();
```

```
            String[] fields = line.split("\t");

            FileSplit inputSplit = (FileSplit) context.getInput
Split();
            String filename = inputSplit.getPath().getName();

            String pid = "";
            if (filename.startsWith("order")) {
                pid = fields[2];
                bean.set(Integer.parseInt(fields[0]), fields[1]
, pid, Integer.parseInt(fields[3]), "", 0, 0, "0");
            } else {
                pid = fields[0];
                bean.set(0, "", pid, 0, fields[1], Integer.pars
eInt(fields[2]), Float.parseFloat(fields[3]), "1");
            }

            k.set(pid);
            context.write(k, bean);
        }
    }

    static class JoinMRReducer extends Reducer<Text, InfoBean,
InfoBean, NullWritable> {
        @Override
        protected void reduce(Text pid, Iterable<InfoBean> bean
s, Context context)
                throws IOException, InterruptedException {

            InfoBean pdBean = new InfoBean();
            ArrayList<InfoBean> orderBeans = new ArrayList<Info
Bean>();

            try {
                for (InfoBean bean : beans) {
                    // product
                    if ("1".equals(bean.getFlag())) {
                        BeanUtils.copyProperties(pdBean, bean);
                    }else{
                        InfoBean odbean = new InfoBean();
                        BeanUtils.copyProperties(odbean, bean);
                        orderBeans.add(odbean);
                    }
                }
            } catch (Exception e) {

            }
```

```
            for(InfoBean bean : orderBeans){
                bean.setPname(pdBean.getPname());
                bean.setCategory_id(pdBean.getCategory_id());
                bean.setPrice(pdBean.getPrice());

                context.write(bean, NullWritable.get());
            }
        }
    }

    public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        Job job = Job.getInstance(conf);

        // 指定本程序的jar包所在的本地路径
        job.setJarByClass(JoinMR.class);

        // 指定本业务job要使用的mapper/Reducer业务类
        job.setMapperClass(JoinMRMapper.class);
        job.setReducerClass(JoinMRReducer.class);

        //指定mapper输出数据的kv类型
        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(InfoBean.class);

        //指定最终输出的数据的kv类型
        job.setOutputKeyClass(InfoBean.class);
        job.setOutputValueClass(NullWritable.class);

        //指定job的输入原始文件所在目录
        FileInputFormat.setInputPaths(job, new Path(args[0]));
        //指定job的输出结果所在目录
        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        //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
        /*job.submit();*/
        boolean res = job.waitForCompletion(true);
        System.exit(res?0:1);
    }
}
```

### 10.2.3 编译打包

在 pom.xml 所在目录下执行打包命令：

```
mvn package
```

执行完成后，会自动生成 `target` 目录，其中有打包好的 jar 文件

现在项目文件结构

```
├── pom.xml
├── src
│   └── main
│       └── java
│           ├── InfoBean.java
│           └── JoinMR.java
└── target
    ├── ...
    ├── mapreduce-jointest-0.0.1-SNAPSHOT.jar
```

### 10.2.4 运行

先把 target 中的 jar 上传到 hadoop 服务器

下载产品和订单的测试数据文件

```
链接: https://pan.baidu.com/s/1pLRnm47 密码: cg7x
```

```
链接: https://pan.baidu.com/s/1pLrvsfT 密码: j2zb
```

上传到 hdfs

```
hdfs dfs -mkdir -p /jointest/input
hdfs dfs -put order.txt /jointest/input
hdfs dfs -put product.txt /jointest/input
```

运行

```
hadoop jar joinmr.jar com.dys.mapreducetest.join.JoinMR /jointe
st/input /jointest/output
```

检查

```
hdfs dfs -cat /jointest/output/*
```

# 11. 实例7 - 计算出用户间的共同好友

## 11.1 需求与实现思路

### 11.1.1 需求

下面是用户的好友关系列表，每一行代表一个用户和他的好友列表

```
A:B,C,D,F,E,O
B:A,C,E,K
C:F,A,D,I
D:A,E,F,L
E:B,C,D,M,L
F:A,B,C,D,E,O,M
G:A,C,D,E,F
H:A,C,D,E,O
I:A,O
J:B,O
K:A,C,D
L:D,E,F
M:E,F,G
O:A,H,I,J
```

需要求出哪些人两两之间有共同好友，及他俩的共同好友都有谁?

例如从前2天记录中可以看出，C、E 是 A、B 的共同好友

最终的形式如下:

```
A-B     E C
A-C     D F
A-D     E F
A-E     D B C
A-F     O B C D E
```

### 11.1.2 实现思路

之前的示例中都是一个 mapreduce 计算出来的，这里我们使用2个 mapreduce 来实现

（1）第1个 mapreduce

- map

找出每个用户都是谁的好友，例如：

读一行 `A:B,C,D,F,E,O` （ A 的好友有这些，反过来拆开，这些人中的每一个都是 A 的好友）

输出 `<B,A> <C,A> <D,A> <F,A> <E,A> <O,A>`

再读一行 `B:A,C,E,K`

输出 `<A,B> <C,B> <E,B> <K,B>`

...

- reduce

key 相同的会分到一组，例如

`<C,A><C,B><C,E><C,F><C,G>.....`

key： `C`

value: `[ A, B, E, F, G ]`

意义是：C 是这些用户的好友

遍历 value 就可以得到：

```
A B 有共同好友 C
A E 有共同好友 C
...
B E 有共同好友 C
B F 有共同好友 C
```

输出：

```
<A-B,C>
<A-E,C>
```

```
<A-F,C>
<A-G,C>
<B-E,C>
<B-F,C>
.....
```

（2）第2个 mapreduce

对上一步的输出结果进行计算

- map

读出上一步的结果数据，组织成 key value 直接输出

例如：

读入一行 `<A-B,C>`

直接输出 `<A-B,C>`

- reduce

读入数据，key 相同的在一组

`<A-B,C><A-B,F><A-B,G>.......`

输出：

`A-B C,F,G,.....`

这样就得出了两个用户间的共同好友列表

# 11.2 代码实践

## 11.2.1 创建项目

新建项目目录 `jointest`，其中新建文件 `pom.xml`，内容:

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="h
ttp://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http:
//maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
```

```
    <groupId>demo.mr</groupId>
    <artifactId>mapreduce-friends</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>jar</packaging>

    <name>mapreduce-friends</name>
    <url>http://maven.apache.org</url>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sour
ceEncoding>
    </properties>

    <dependencies>
        <!-- https://mvnrepository.com/artifact/commons-beanuti
ls/commons-beanutils -->
        <dependency>
            <groupId>commons-beanutils</groupId>
            <artifactId>commons-beanutils</artifactId>
            <version>1.9.3</version>
        </dependency>

        <!-- https://mvnrepository.com/artifact/org.apache.hado
op/hadoop-common -->
        <dependency>
            <groupId>org.apache.hadoop</groupId>
            <artifactId>hadoop-common</artifactId>
            <version>2.7.3</version>
        </dependency>
        <!-- https://mvnrepository.com/artifact/org.apache.hado
op/hadoop-hdfs -->
        <dependency>
            <groupId>org.apache.hadoop</groupId>
            <artifactId>hadoop-hdfs</artifactId>
            <version>2.7.3</version>
        </dependency>
        <!-- https://mvnrepository.com/artifact/org.apache.hado
op/hadoop-mapreduce-client-common -->
        <dependency>
            <groupId>org.apache.hadoop</groupId>
            <artifactId>hadoop-mapreduce-client-common</artifac
tId>
            <version>2.7.3</version>
        </dependency>
        <!-- https://mvnrepository.com/artifact/org.apache.hado
op/hadoop-mapreduce-client-core -->
        <dependency>
```

```
            <groupId>org.apache.hadoop</groupId>
            <artifactId>hadoop-mapreduce-client-core</artifactI
d>
            <version>2.7.3</version>
        </dependency>
    </dependencies>
</project>
```

然后创建源码目录 `src/main/java`

现在项目目录的文件结构

```
├── pom.xml
└── src
    └── main
        └── java
```

### 11.2.2 代码

**第一步的 mapreduce 程序：** src/main/java/StepFirst.java

```
import java.io.IOException;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.LongWritable;
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 StepFirst {
    static class FirstMapper extends Mapper<LongWritable, Text,
 Text, Text> {
        @Override
        protected void map(LongWritable key, Text value, Mapper
<LongWritable, Text, Text, Text>.Context context)
                throws IOException, InterruptedException {

            String line = value.toString();
            String[] arr = line.split(":");
            String user = arr[0];
```

```
            String friends = arr[1];

            for(String friend : friends.split(",")){
                context.write(new Text(friend), new Text(user))
;
            }
        }
    }

    static class FirstReducer extends Reducer<Text, Text, Text,
 Text> {
        @Override
        protected void reduce(Text friend, Iterable<Text> users
, Context context)
                throws IOException, InterruptedException {

            StringBuffer buf = new StringBuffer();
            for(Text user : users){
                buf.append(user).append(",");
            }

            context.write(new Text(friend), new Text(buf.toStri
ng()));
        }
    }

    public static void main(String[] args) throws Exception {
        // 创建任务
        Configuration conf = new Configuration();
        Job job = Job.getInstance(conf);
        job.setJarByClass(StepFirst.class);

        // 任务输出类型
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(Text.class);

        // 指定 map reduce
        job.setMapperClass(FirstMapper.class);
        job.setReducerClass(FirstReducer.class);

        // 输入文件路径、输出路径
        FileInputFormat.setInputPaths(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        // 提交任务
        job.waitForCompletion(true);
```

```
    }
}
```

**第二步的 mapreduce 程序：** src/main/java/StepSecond.java

```
import java.io.IOException;
import java.util.Arrays;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.LongWritable;
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 StepSecond {
    static class SecondMapper extends Mapper<LongWritable, Text
, Text, Text> {
        @Override
        protected void map(LongWritable key, Text value, Mapper
<LongWritable, Text, Text, Text>.Context context)
                throws IOException, InterruptedException {

            String line = value.toString();
            String[] friend_users = line.split("\t");

            String friend = friend_users[0];
            String[] users = friend_users[1].split(",");

            Arrays.sort(users);

            for(int i=0; i<users.length - 1; i++){
                for(int j=i+1; j<users.length; j++){
                    // 这两个人有共同的好友
                    context.write(new Text(users[i] + "-" + use
rs[j]), new Text(friend));
                }
            }
        }
    }

    static class SecondReducer extends Reducer<Text, Text, Text
, Text> {
```

```
        @Override
        protected void reduce(Text user_user, Iterable<Text> fr
iends, Context context)
                throws IOException, InterruptedException {

            StringBuffer buf = new StringBuffer();
            for(Text friend : friends){
                buf.append(friend).append(" ");
            }

            context.write(user_user, new Text(buf.toString()));
        }
    }

    public static void main(String[] args) throws Exception {
        // 创建任务
        Configuration conf = new Configuration();
        Job job = Job.getInstance(conf);
        job.setJarByClass(StepSecond.class);

        // 任务输出类型
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(Text.class);

        // 指定 map reduce
        job.setMapperClass(SecondMapper.class);
        job.setReducerClass(SecondReducer.class);

        // 输入文件路径、输出路径
        FileInputFormat.setInputPaths(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        // 提交任务
        job.waitForCompletion(true);

    }
}
```

### 11.2.3 编译打包

在 pom.xml 所在目录下执行打包命令：

```
mvn package
```

执行完成后，会自动生成 `target` 目录，其中有打包好的 jar 文件

现在项目文件结构

```
├── pom.xml
├── src
│   └── main
│       └── java
│           ├── StepFirst.java
│           └── StepSecond.java
└── target
    ├── ...
    ├── mapreduce-friends-0.0.1-SNAPSHOT.jar
```

### 11.2.4 运行

先把 target 中的 jar 上传到 hadoop 服务器

下载测试数据文件

```
链接: https://pan.baidu.com/s/1o8fmfbG 密码: kbut
```

上传到 hdfs

```
hdfs dfs -mkdir -p /friends/input
hdfs dfs -put friendsdata.txt /friends/input
```

运行第一步

```
hadoop jar mapreduce-friends-0.0.1-SNAPSHOT.jar StepFirst /frie
nds/input/friendsdata.txt /friends/output01
```

运行第二步

```
hadoop jar mapreduce-friends-0.0.1-SNAPSHOT.jar StepSecond /fri
ends/output01/part-r-00000 /friends/output02
```

查看结果

```
hdfs dfs -ls /friends/output02
hdfs dfs -cat /friends/output02/*
```

# 12. 小结

MapReduce 的基础内容介绍完了，希望可以帮助您快速熟悉 MapReduce 的工作原理和开发方法

本内容出自公众号 `性能与架构` （yogoup），如有批评与建议（例如 内容有误、感觉有什么不足的地方、改进建议等），欢迎发送消息