

# 尚硅谷大数据技术之 Hadoop (MapReduce)

(作者：大海哥)

官网：[www.atguigu.com](http://www.atguigu.com)

版本：V1.3

## 一 MapReduce 入门

### 1.1 MapReduce 定义

Mapreduce 是一个分布式运算程序的编程框架，是用户开发“基于 hadoop 的数据分析应用”的核心框架。

Mapreduce 核心功能是将用户编写的业务逻辑代码和自带默认组件整合成一个完整的分布式运算程序，并发运行在一个 hadoop 集群上。

### 1.2 MapReduce 优缺点

#### 1.2.1 优点

1) **MapReduce 易于编程**。它简单的实现一些接口，就可以完成一个分布式程序，这个分布式程序可以分布到大量廉价的 PC 机器上运行。也就是说你写一个分布式程序，跟写一个简单的串行程序是一模一样的。就是因为这个特点使得 MapReduce 编程变得非常流行。

2) **良好的扩展性**。当你的计算资源不能得到满足的时候，你可以通过简单的增加机器来扩展它的计算能力。

3) **高容错性**。MapReduce 设计的初衷就是使程序能够部署在廉价的 PC 机器上，这就要求它具有很高的容错性。比如其中一台机器挂了，它可以把上面的计算任务转移到另外一个节点上运行，不至于这个任务运行失败，而且这个过程不需要人工参与，而完全是由 Hadoop 内部完成的。

4) **适合 PB 级以上海量数据的离线处理**。这里加红字体离线处理，说明它适合离线处理而不适合在线处理。比如像毫秒级别的返回一个结果，MapReduce 很难做到。

#### 1.2.2 缺点

**MapReduce 不擅长做实时计算、流式计算、DAG（有向图）计算。**

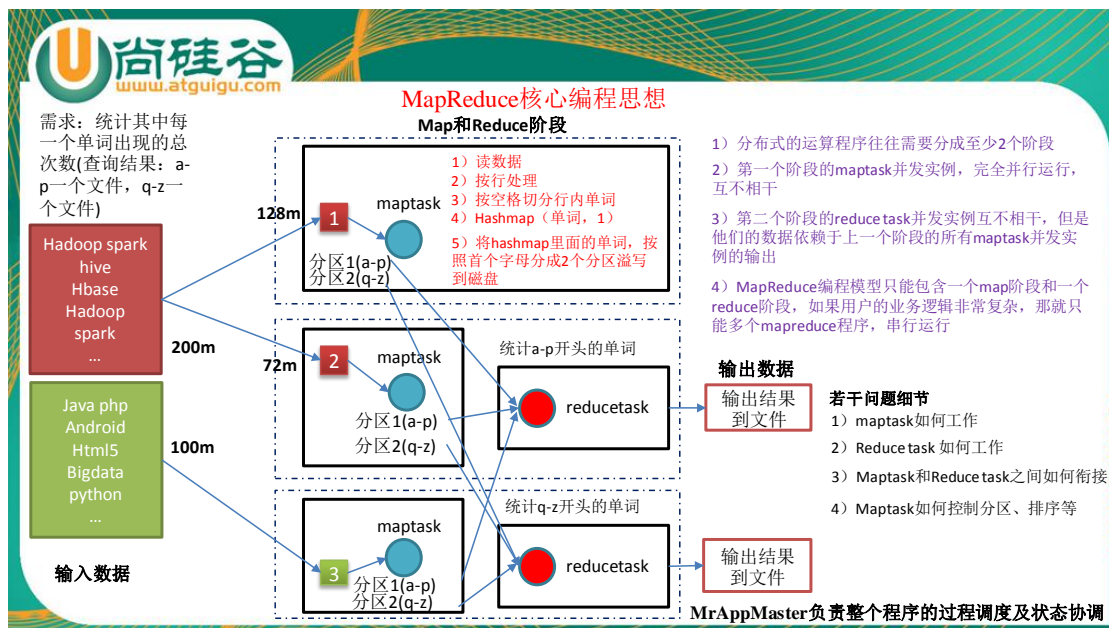
1) **实时计算**。MapReduce 无法像 Mysql 一样，在毫秒或者秒级内返回结果。

【更多 Java、HTML5、Android、python、大数据 资料下载，可访问尚硅谷（中国）官网 [www.atguigu.com](http://www.atguigu.com) 下载区】

2) 流式计算。流式计算的输入数据是动态的，而 MapReduce 的输入数据集是静态的，不能动态变化。这是因为 MapReduce 自身的设计特点决定了数据源必须是静态的。

3) DAG (有向图) 计算。多个应用程序存在依赖关系，后一个应用程序的输入为前一个的输出。在这种情况下，MapReduce 并不是不能做，而是使用后，每个 MapReduce 作业的输出结果都会写入到磁盘，会造成大量的磁盘 IO，导致性能非常的低下。

## 1.3 MapReduce 核心思想



- 1) 分布式的运算程序往往需要分成至少 2 个阶段。
- 2) 第一个阶段的 maptask 并发实例，完全并行运行，互不相干。
- 3) 第二个阶段的 reduce task 并发实例互不相干，但是他们的数据依赖于上一个阶段的所有 maptask 并发实例的输出。
- 4) MapReduce 编程模型只能包含一个 map 阶段和一个 reduce 阶段，如果用户的业务逻辑非常复杂，那就只能多个 mapreduce 程序，串行运行。

## 1.4 MapReduce 进程

一个完整的 mapreduce 程序在分布式运行时三类实例进程：

- 1) MrAppMaster：负责整个程序的过程调度及状态协调。
- 2) MapTask：负责 map 阶段的整个数据处理流程。
- 3) ReduceTask：负责 reduce 阶段的整个数据处理流程。

## 1.5 MapReduce 编程规范（八股文）

用户编写的程序分成三个部分：Mapper，Reducer，Driver(提交运行 mr 程序的客户端)

### 1) Mapper 阶段

- (1) 用户自定义的 Mapper 要继承自己的父类
- (2) Mapper 的输入数据是 KV 对的形式 (KV 的类型可自定义)
- (3) Mapper 中的业务逻辑写在 map()方法中
- (4) Mapper 的输出数据是 KV 对的形式 (KV 的类型可自定义)
- (5) map()方法 (maptask 进程) 对每一个<K,V>调用一次

### 2) Reducer 阶段

- (1) 用户自定义的 Reducer 要继承自己的父类
- (2) Reducer 的输入数据类型对应 Mapper 的输出数据类型，也是 KV
- (3) Reducer 的业务逻辑写在 reduce()方法中
- (4) Reducetask 进程对每一组相同 k 的<k,v>组调用一次 reduce()方法

### 3) Driver 阶段

整个程序需要一个 Driver 来进行提交，提交的是一个描述了各种必要信息的 job 对象

### 4) 案例实操

详见 7.1.1 统计一堆文件中单词出现的个数 (WordCount 案例)。

## 二 Hadoop 序列化

### 2.1 为什么要序列化?

一般来说，“活的”对象只生存在内存里，关机断电就没有了。而且“活的”对象只能由本地的进程使用，不能被发送到网络上的另外一台计算机。然而序列化可以存储“活的”对象，可以将“活的”对象发送到远程计算机。

### 2.2 什么是序列化?

序列化就是把内存中的对象，转换成字节序列（或其他数据传输协议）以便于存储（持久化）和网络传输。

反序列化就是将收到字节序列（或其他数据传输协议）或者是硬盘的持久化数据，转换成内存中的对象。

## 2.3 为什么不用 Java 的序列化？

Java 的序列化是一个重量级序列化框架（Serializable），一个对象被序列化后，会附带很多额外的信息（各种校验信息，header，继承体系等），不便于在网络中高效传输。所以，hadoop 自己开发了一套序列化机制（Writable），精简、高效。

## 2.4 为什么序列化对 Hadoop 很重要？

因为 Hadoop 在集群之间进行通讯或者 RPC 调用的时候，需要序列化，而且要求序列化要快，且体积要小，占用带宽要小。所以必须理解 Hadoop 的序列化机制。

序列化和反序列化在分布式数据处理领域经常出现：进程通信和永久存储。然而 Hadoop 中各个节点的通信是通过远程调用（RPC）实现的，那么 RPC 序列化要求具有以下特点：

- 1) 紧凑：紧凑的格式能让我们充分利用网络带宽，而带宽是数据中心最稀缺的资源
- 2) 快速：进程通信形成了分布式系统的骨架，所以需要尽量减少序列化和反序列化的性能开销，这是基本的；
- 3) 可扩展：协议为了满足新的需求变化，所以控制客户端和服务端过程中，需要直接引进相应的协议，这些是新协议，原序列化方式能支持新的协议报文；
- 4) 互操作：能支持不同语言写的客户端和服务端进行交互；

## 2.5 常用数据序列化类型

常用的数据类型对应的 hadoop 数据序列化类型

Java 类型	Hadoop Writable 类型
boolean	BooleanWritable
byte	ByteWritable
int	IntWritable
float	FloatWritable
long	LongWritable
double	DoubleWritable
string	Text
map	MapWritable
array	ArrayWritable

## 2.6 自定义 bean 对象实现序列化接口（Writable）

1) 自定义 bean 对象要想序列化传输，必须实现序列化接口，需要注意以下 7 项。

(1) 必须实现 Writable 接口

(2) 反序列化时，需要反射调用空参构造函数，所以必须有空参构造

【更多 Java、HTML5、Android、python、大数据 资料下载，可访问尚硅谷（中国）官网 [www.atguigu.com](http://www.atguigu.com) 下载区】

```
public FlowBean() {  
    super();  
}
```

(3) 重写序列化方法

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

(4) 重写反序列化方法

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

(5) 注意反序列化的顺序和序列化的顺序完全一致

(6) 要想把结果显示在文件中，需要重写 `toString()`，可用“\t”分开，方便后续用。

(7) 如果需要将自定义的 bean 放在 key 中传输，则还需要实现 `comparable` 接口，因为 mapreduce 框中的 shuffle 过程一定会对 key 进行排序。

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

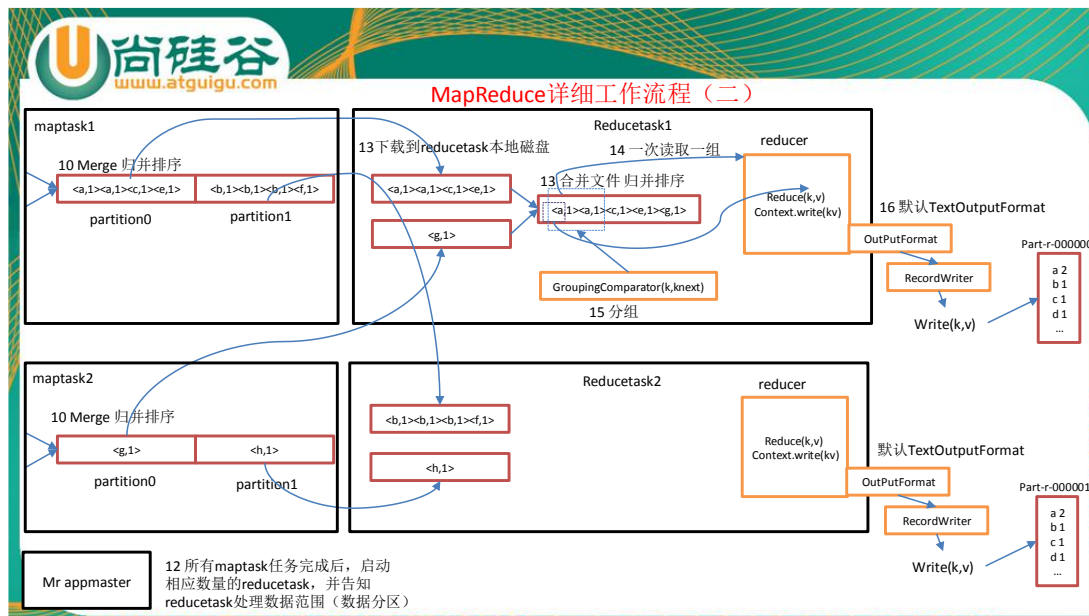
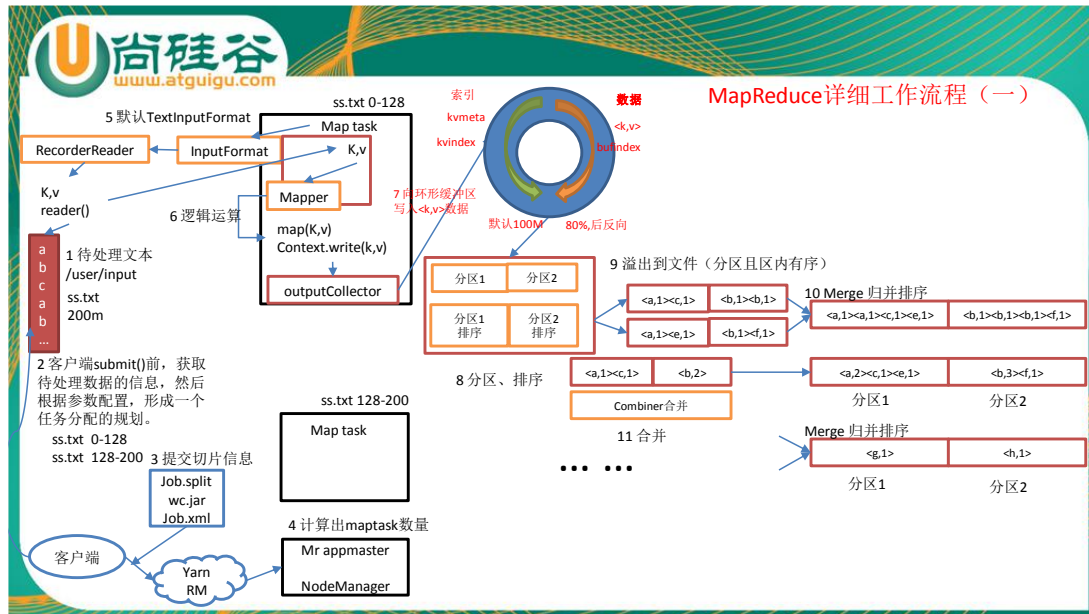
2) 案例实操

详见 7.2.1 统计每一个手机号耗费的总上行流量、下行流量、总流量（序列化）。

## 三 MapReduce 框架原理

### 3.1 MapReduce 工作流程

1) 流程示意图



## 2) 流程详解

上面的流程是整个 mapreduce 最全流程，但是 shuffle 过程只是从第 7 步开始到第 16 步结束，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) 在溢出过程中，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) reducerTask 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

【更多 Java、HTML5、Android、python、大数据 资料下载，可访问尚硅谷（中国）官网 [www.atguigu.com](http://www.atguigu.com) 下载区】



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)

### 3) 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。

## 3.2 InputFormat 数据输入

### 3.2.1 Job 提交流程和切片源码详解

#### 1) job 提交流程源码详解

```
waitForCompletion()

submit();

// 1 建立连接

connect();

// 1) 创建提交 job 的代理

new Cluster(getConfiguration());

// (1) 判断是本地 yarn 还是远程

initialize(jobTrackAddr, conf);

// 2 提交 job

submitter.submitJobInternal(Job.this, cluster)

// 1) 创建给集群提交数据的 Stag 路径

Path jobStagingArea = JobSubmissionFiles.getStagingDir(cluster, conf);

// 2) 获取 jobid , 并创建 job 路径

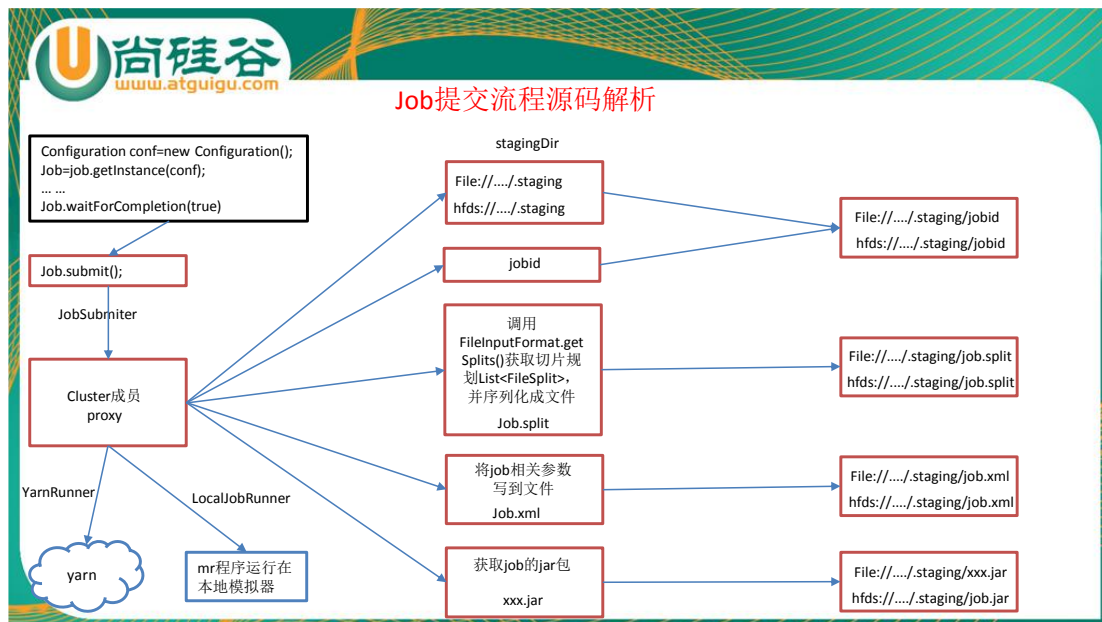
JobID jobId = submitClient.getNewJobID();

// 3) 拷贝 jar 包到集群

copyAndConfigureFiles(job, submitJobDir);

rUploader.uploadFiles(job, jobSubmitDir);
```

```
// 4) 计算切片, 生成切片规划文件  
writeSplits(job, submitJobDir);  
  
maps = writeNewSplits(job, jobSubmitDir);  
  
input.getSplits(job);  
  
// 5) 向 Stag 路径写 xml 配置文件  
writeConf(conf, submitJobFile);  
  
conf.writeXml(out);  
  
// 6) 提交 job, 返回提交状态  
  
status = submitClient.submitJob(jobId, submitJobDir.toString(),  
job.getCredentials());
```



## 2) FileInputFormat 源码解析(input.getSplits(job))

- (1) 找到你数据存储的目录。
- (2) 开始遍历处理（规划切片）目录下的每一个文件
- (3) 遍历第一个文件 ss.txt
  - a) 获取文件大小 fs.sizeOf(ss.txt);
  - b) 计算切片大小

$\text{computeSliteSize}(\text{Math.max}(\text{minSize}, \text{Math.min}(\text{maxSize}, \text{blocksize}))) = \text{blocksize} = 128\text{M}$

c) 默认情况下，切片大小=blocksize



d) 开始切, 形成第 1 个切片: ss.txt—0:128M 第 2 个切片 ss.txt—128:256M 第 3 个切片 ss.txt—256M:300M (每次切片时, 都要判断切完剩下的部分是否大于块的 1.1 倍, 不大于 1.1 倍就划分一块切片)

e) 将切片信息写到一个切片规划文件中

f) 整个切片的核心过程在 `getSplit()` 方法中完成。

g) 数据切片只是在逻辑上对输入数据进行分片, 并不会再磁盘上将其切分成分片进行存储。`InputSplit` 只记录了分片的元数据信息, 比如起始位置、长度以及所在的节点列表等。

h) 注意: block 是 HDFS 物理上存储的数据, 切片是对数据逻辑上的划分。

(4) 提交切片规划文件到 yarn 上, yarn 上的 `MrAppMaster` 就可以根据切片规划文件计算开启 `maptask` 个数。

### 3.2.2 FileInputFormat 切片机制

1) `FileInputFormat` 中默认的分片机制:

- (1) 简单地按照文件的内容长度进行切片
- (2) 切片大小, 默认等于 block 大小
- (3) 切片时不考虑数据集整体, 而是逐个针对每一个文件单独切片

比如待处理数据有两个文件:

file1.txt	320M
file2.txt	10M

经过 `FileInputFormat` 的分片机制运算后, 形成的切片信息如下:

file1.txt.split1--	0~128
file1.txt.split2--	128~256
file1.txt.split3--	256~320
file2.txt.split1--	0~10M

2) `FileInputFormat` 切片大小的参数配置

通过分析源码, 在 `FileInputFormat` 中, 计算切片大小的逻辑: `Math.max(minSize, Math.min(maxSize, blockSize));`

切片主要由这几个值来运算决定

`mapreduce.input.fileinputformat.split.minsize=1` 默认值为 1

`mapreduce.input.fileinputformat.split.maxsize= Long.MAXValue` 默认值 `Long.MAXValue`

因此, 默认情况下, 切片大小=blocksize。

【更多 Java、HTML5、Android、python、大数据 资料下载, 可访问尚硅谷(中国)官网 [www.atguigu.com](http://www.atguigu.com) 下载区】

maxsize（切片最大值）：参数如果调得比 blocksize 小，则会让切片变小，而且就等于配置的这个参数的值。

minsize（切片最小值）：参数调的比 blockSize 大，则可以让切片变得比 blockSize 还大。

### 3) 获取切片信息 API

```
// 根据文件类型获取切片信息
FileSplit inputSplit = (FileSplit) context.getInputSplit();
// 获取切片的文件名称
String name = inputSplit.getPath().getName();
```

## 3.2.3 CombineTextInputFormat 切片机制

关于大量小文件的优化策略

1) 默认情况下 TextInputFormat 对任务的切片机制是按文件规划切片，不管文件多小，都会是一个单独的切片，都会交给一个 maptask，这样如果有大量小文件，就会产生大量的 maptask，处理效率极其低下。

### 2) 优化策略

(1) 最好的办法，在数据处理系统的最前端（预处理/采集），将小文件先合并成大文件，再上传到 HDFS 做后续分析。

(2) 补救措施：如果已经是大量小文件在 HDFS 中了，可以使用另一种 InputFormat 来做切片（CombineTextInputFormat），它的切片逻辑跟 TextFileInputFormat 不同：它可以将多个小文件从逻辑上规划到一个切片中，这样，多个小文件就可以交给一个 maptask。

(3) 优先满足最小切片大小，不超过最大切片大小

```
CombineTextInputFormat.setMaxInputSplitSize(job, 4194304); // 4m
```

```
CombineTextInputFormat.setMinInputSplitSize(job, 2097152); // 2m
```

举例：0.5m+1m+0.3m+5m=2m + 4.8m=2m + 4m + 0.8m

### 3) 具体实现步骤

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

### 4) 案例实操

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

### 3.2.4 InputFormat 接口实现类

MapReduce 任务的输入文件一般是存储在 HDFS 里面。输入的文件格式包括：基于行的日志文件、二进制格式文件等。这些文件一般会很大，达到数十 GB，甚至更大。那么 MapReduce 是如何读取这些数据呢？下面我们首先学习 InputFormat 接口。

InputFormat 常见的接口实现类包括：TextInputFormat、KeyValueTextInputFormat、NLineInputFormat、CombineTextInputFormat 和自定义 InputFormat 等。

#### 1) TextInputFormat

TextInputFormat 是默认的 InputFormat。每条记录是一行输入。键是 LongWritable 类型，存储该行在整个文件中的字节偏移量。值是这行的内容，不包括任何行终止符（换行符和回车符）。

以下是一个示例，比如，一个分片包含了如下 4 条文本记录。

Rich learning form
Intelligent learning engine
Learning more convenient
From the real demand for more close to the enterprise

每条记录表示为以下键/值对：

(0, Rich learning form)
(19, Intelligent learning engine)
(47, Learning more convenient)
(72, From the real demand for more close to the enterprise)

很明显，键并不是行号。一般情况下，很难取得行号，因为文件按字节而不是按行切分为分片。

#### 2) KeyValueTextInputFormat

每一行均为一条记录，被分隔符分割为 key，value。可以通过在驱动类中设置 `conf.set(KeyValueLineRecordReader.KEY_VALUE_SEPERATOR, " ");` 来设定分隔符。默认分隔符是 tab (\t)。

以下是一个示例，输入是一个包含 4 条记录的分片。其中——>表示一个（水平方向的）制表符。

```
line1 ——>Rich learning form
line2 ——>Intelligent learning engine
line3 ——>Learning more convenient
line4 ——>From the real demand for more close to the enterprise
```

每条记录表示为以下键/值对：

```
(line1,Rich learning form)
(line2,Intelligent learning engine)
(line3,Learning more convenient)
(line4,From the real demand for more close to the enterprise)
```

此时的键是每行排在制表符之前的 Text 序列。

### 3) NLineInputFormat

如果使用 NLineInputFormat, 代表每个 map 进程处理的 **InputSplit** 不再按 **block** 块去划分, 而是按 **NLineInputFormat** 指定的行数 **N** 来划分。即输入文件的总行数/**N**=切片数, 如果不整除, 切片数=商+1。

以下是一个示例, 仍然以上面的 4 行输入为例。

```
Rich learning form
Intelligent learning engine
Learning more convenient
From the real demand for more close to the enterprise
```

例如, 如果 N 是 2, 则每个输入分片包含两行。开启 2 个 maptask。

```
(0,Rich learning form)
(19,Intelligent learning engine)
```

另一个 mapper 则收到后两行:

```
(47,Learning more convenient)
(72,From the real demand for more close to the enterprise)
```

这里的键和值与 TextInputFormat 生成的一样。

## 3.2.5 自定义 InputFormat

### 1) 概述

(1) 自定义一个类继承 FileInputFormat。

【更多 Java、HTML5、Android、python、大数据 资料下载, 可访问尚硅谷(中国)官网 [www.atguigu.com](http://www.atguigu.com) 下载区】

(2) 改写 RecordReader，实现一次读取一个完整文件封装为 KV。

(3) 在输出时使用 SequenceFileOutPutFormat 输出合并文件。

## 2) 案例实操

详见 7.4 小文件处理（自定义 InputFormat）。

## 3.3 MapTask 工作机制

### 3.3.1 并行度决定机制

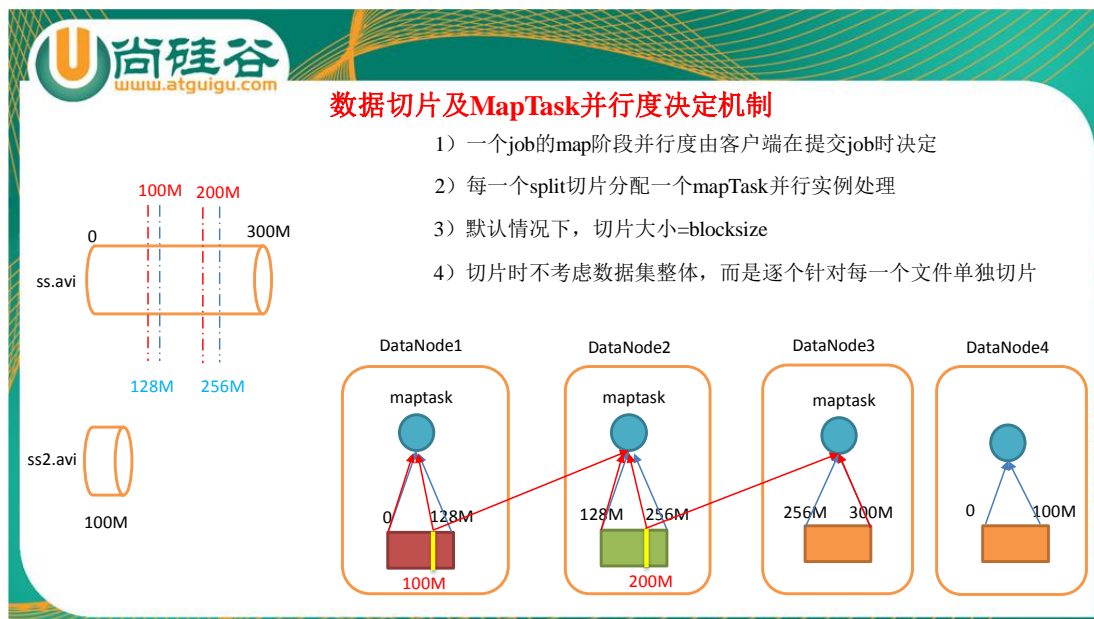
#### 1) 问题引出

maptask 的并行度决定 map 阶段的任务处理并发度，进而影响到整个 job 的处理速度。

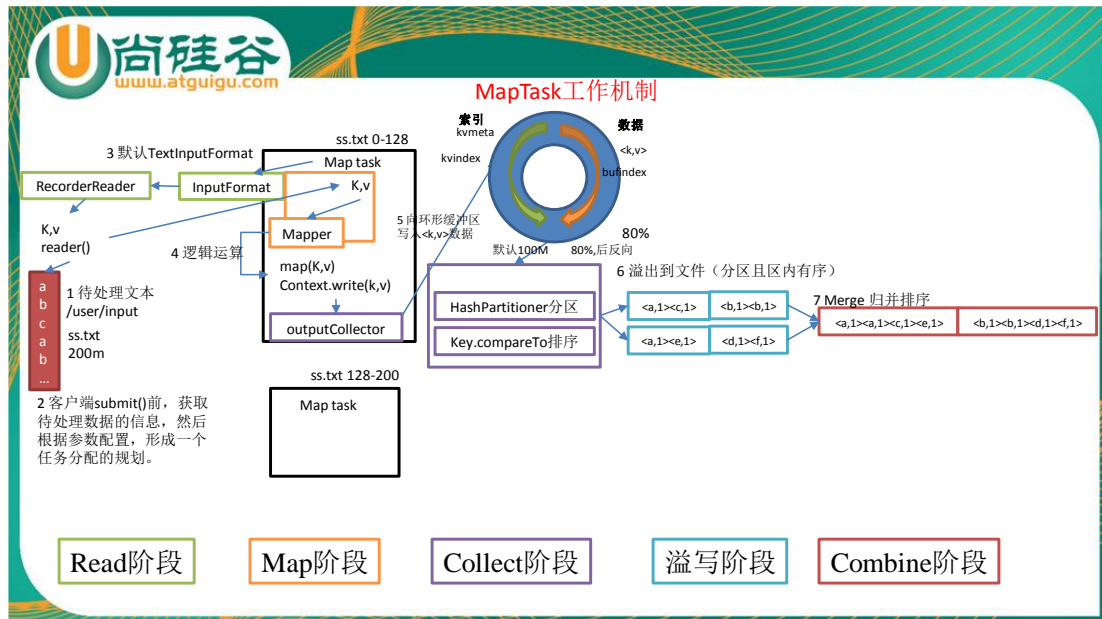
那么，mapTask 并行任务是否越多越好呢？

#### 2) MapTask 并行度决定机制

一个 job 的 map 阶段 MapTask 并行度（个数），由客户端提交 job 时的切片个数决定。



### 3.3.2 MapTask 工作机制



（1）Read 阶段：Map Task 通过用户编写的 `RecordReader`，从输入 `InputSplit` 中解析出一个一个 `key/value`。

（2）Map 阶段：该节点主要是将解析出的 `key/value` 交给用户编写 `map()` 函数处理，并产生一系列新的 `key/value`。

（3）Collect 收集阶段：在用户编写 `map()` 函数中，当数据处理完成后，一般会调用 `OutputCollector.collect()` 输出结果。在该函数内部，它会将生成的 `key/value` 分区（调用 `Partitioner`），并写入一个环形内存缓冲区中。

（4）Spill 阶段：即“溢写”，当环形缓冲区满后，MapReduce 会将数据写到本地磁盘上，生成一个临时文件。需要注意的是，将数据写入本地磁盘之前，先要对数据进行一次本地排序，并在必要时对数据进行合并、压缩等操作。

溢写阶段详情：

步骤 1：利用快速排序算法对缓存区内的数据进行排序，排序方式是，先按照分区编号 `partition` 进行排序，然后按照 `key` 进行排序。这样，经过排序后，数据以分区为单位聚集在一起，且同一分区内所有数据按照 `key` 有序。

步骤 2：按照分区编号由小到大依次将每个分区中的数据写入任务工作目录下的临时文件 `output/spillN.out`（`N` 表示当前溢写次数）中。如果用户设置了 `Combiner`，则写入文件之前，对每个分区中的数据进行一次聚集操作。

步骤 3：将分区数据的元信息写到内存索引数据结构 `SpillRecord` 中，其中每个分区的元信息包括 `kvindex` 和 `buindex`。

【更多 Java、HTML5、Android、python、大数据 资料下载，可访问尚硅谷（中国）官网 [www.atguigu.com](http://www.atguigu.com) 下载区】



信息包括在临时文件中的偏移量、压缩前数据大小和压缩后数据大小。如果当前内存索引大小超过 1MB，则将内存索引写到文件 `output/spillN.out.index` 中。

(5) Combine 阶段：当所有数据处理完成后，MapTask 对所有临时文件进行一次合并，以确保最终只会生成一个数据文件。

当所有数据处理完后，MapTask 会将所有临时文件合并成一个大文件，并保存到文件 `output/file.out` 中，同时生成相应的索引文件 `output/file.out.index`。

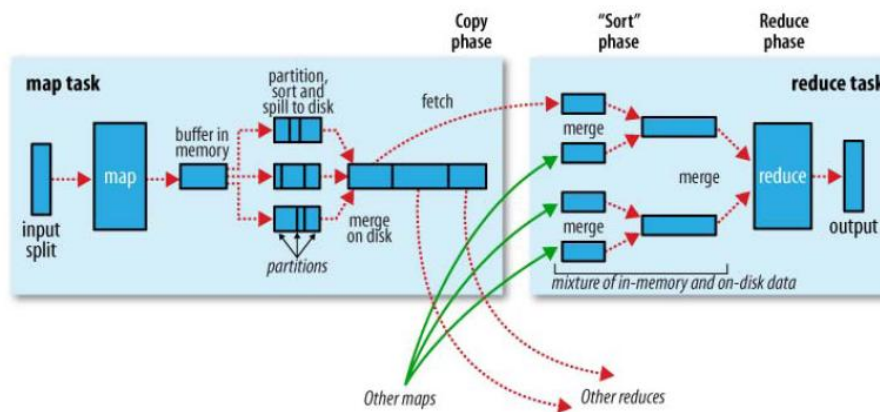
在进行文件合并过程中，MapTask 以分区为单位进行合并。对于某个分区，它将采用多轮递归合并的方式。每轮合并 `io.sort.factor`（默认 100）个文件，并将产生的文件重新加入待合并列表中，对文件排序后，重复以上过程，直到最终得到一个大文件。

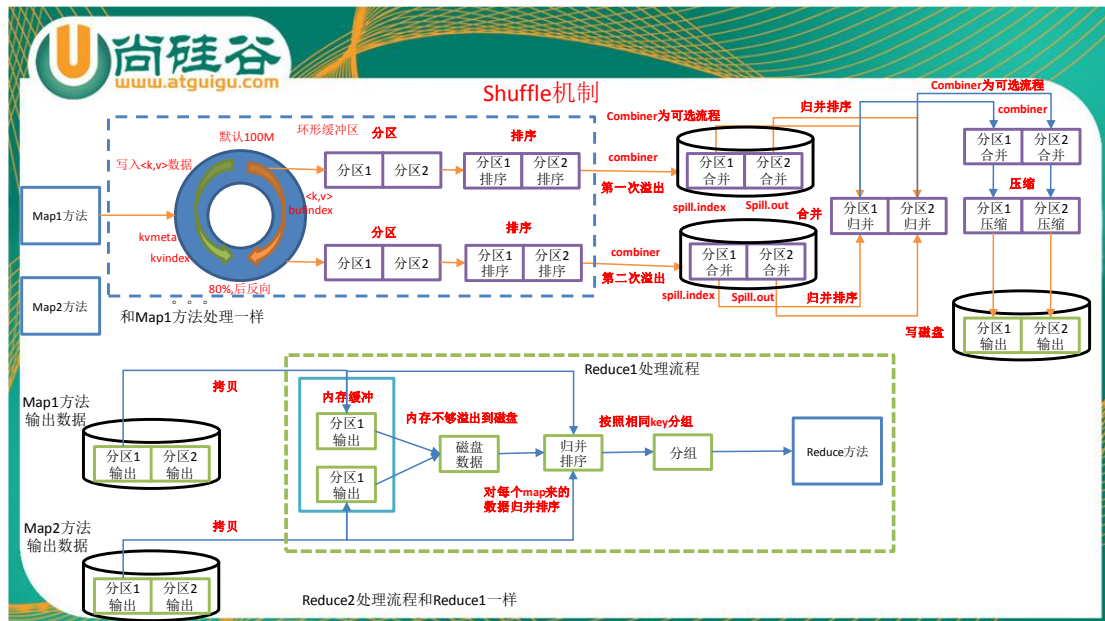
让每个 MapTask 最终只生成一个数据文件，可避免同时打开大量文件和同时读取大量小文件产生的随机读取带来的开销。

## 3.4 Shuffle 机制

### 3.4.1 Shuffle 机制

Mapreduce 确保每个 reducer 的输入都是按键排序的。系统执行排序的过程（即将 map 输出作为输入传给 reducer）称为 shuffle。





### 3.4.2 Partition 分区

0) 问题引出：要求将统计结果按照条件输出到不同文件中（分区）。比如：将统计结果按照手机归属地不同省份输出到不同文件中（分区）

1) 默认 partition 分区

```
public class HashPartitioner<K, V> extends Partitioner<K, V> {
    public int getPartition(K key, V value, int numReduceTasks) {
        return (key.hashCode() & Integer.MAX_VALUE) % numReduceTasks;
    }
}
```

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

2) 自定义 Partitioner 步骤

(1) 自定义类继承 Partitioner，重写 getPartition()方法

```
public class ProvincePartitioner extends Partitioner<Text, FlowBean> {

    @Override
    public int getPartition(Text key, FlowBean value, int numPartitions) {

        // 1 获取电话号码的前三位
        String preNum = key.toString().substring(0, 3);

        int partition = 4;

        // 2 判断是哪个省
```

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

(2) 在 job 驱动中，设置自定义 partitioner:

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

(3) 自定义 partition 后，要根据自定义 partitioner 的逻辑设置相应数量的 reduce task

```
job.setNumReduceTasks(5);
```

3) 注意:

如果 reduceTask 的数量 > getPartition 的结果数，则会多产生几个空的输出文件 part-r-000xx;

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;

如果 reduceTask 的数量 = 1，则不管 mapTask 端输出多少个分区文件，最终结果都交给这一个 reduceTask，最终也就只会产生一个结果文件 part-r-00000;

例如：假设自定义分区数为 5，则

(1) job.setNumReduceTasks(1); 会正常运行，只不过会产生一个输出文件

(2) job.setNumReduceTasks(2); 会报错

(3) job.setNumReduceTasks(6); 大于 5，程序会正常运行，会产生空文件

4) 案例实操

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

详见 7.1.2 需求 2: 把单词按照 ASCII 码奇偶分区 (Partitioner)

### 3.4.3 WritableComparable 排序

排序是 MapReduce 框架中最重要的操作之一。Map Task 和 Reduce Task 均会对数据（按照 key）进行排序。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，  
【更多 Java、HTML5、Android、python、大数据 资料下载，可访问尚硅谷（中国）官网 [www.atguigu.com](http://www.atguigu.com) 下载区】

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。**

对于 Map Task，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 Reduce Task，它从每个 Map Task 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一定阈值，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合并。

### 每个阶段的默认排序

#### 1) 排序的分类:

##### (1) 部分排序:

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

##### (2) 全排序:

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

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

##### (3) 辅助排序：(GroupingComparator 分组)

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

##### (4) 二次排序:

在自定义排序过程中，如果 compareTo 中的判断条件为两个即为二次排序。

## 2) 自定义排序 WritableComparable

### (1) 原理分析

bean 对象实现 **WritableComparable** 接口重写 **compareTo** 方法，就可以实现排序

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

### (2) 案例实操

详见 7.2.3 需求 3：将统计结果按照总流量倒序排序（排序）

详见 7.2.4 需求 4：不同省份输出文件内部排序（部分排序）

## 3.4.4 GroupingComparator 分组（辅助排序）

1) 对 reduce 阶段的数据根据某一个或几个字段进行分组。

2) 案例实操

详见 7.3 求出每一个订单中最贵的商品（GroupingComparator）

## 3.4.5 Combiner 合并

1) combiner 是 MR 程序中 Mapper 和 Reducer 之外的一种组件。

2) combiner 组件的父类就是 Reducer。

3) combiner 和 reducer 的区别在于运行的位置：

Combiner 是在每一个 maptask 所在的节点运行；

Reducer 是接收全局所有 Mapper 的输出结果；

4) combiner 的意义就是对每一个 maptask 的输出进行局部汇总，以减小网络传输量。

5) combiner 能够应用的前提是不能影响最终的业务逻辑，而且，combiner 的输出 kv 应该跟 reducer 的输入 kv 类型要对应起来。

Mapper

3 5 7 ->(3+5+7)/3=5

2 6 ->(2+6)/2=4

Reducer

(3+5+7+2+6)/5=23/5      不等于      (5+4)/2=9/2

6) 自定义 Combiner 实现步骤：

【更多 Java、HTML5、Android、python、大数据 资料下载，可访问尚硅谷（中国）官网 [www.atguigu.com](http://www.atguigu.com) 下载区】

(1) 自定义一个 combiner 继承 Reducer，重写 reduce 方法

```
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
        int count = 0;
        for(IntWritable v : values){
            count = v.get();
        }
        // 2 写出
        context.write(key, new IntWritable(count));
    }
}
```

(2) 在 job 驱动类中设置：

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

7) 案例实操

详见 7.1.3 需求 3：对每一个 maptask 的输出局部汇总 (Combiner)

## 3.5 ReduceTask 工作机制

1) 设置 ReduceTask 并行度 (个数)

reducetask 的并行度同样影响整个 job 的执行并发度和执行效率，但与 maptask 的并发数由切片数决定不同，Reducetask 数量的决定是可以直接手动设置：

```
//默认值是 1，手动设置为 4
job.setNumReduceTasks(4);
```

2) 注意

- (1) reducetask=0，表示没有 reduce 阶段，输出文件个数和 map 个数一致。
- (2) reducetask 默认值就是 1，所以输出文件个数为一个。
- (3) 如果数据分布不均匀，就有可能在 reduce 阶段产生数据倾斜
- (4) reducetask 数量并不是任意设置，还要考虑业务逻辑需求，有些情况下，需要计算全局汇总结果，就只能有 1 个 reducetask。
- (5) 具体多少个 reducetask，需要根据集群性能而定。
- (6) 如果分区数不是 1，但是 reducetask 为 1，是否执行分区过程。答案是：不执行分区过程。

因为在 maptask 的源码中，执行分区的前提是先判断 reduceNum 个数是否大于 1。  
【更多 Java、HTML5、Android、python、大数据 资料下载，可访问尚硅谷（中国）官网 [www.atguigu.com](http://www.atguigu.com) 下载区】



不大于 1 肯定不执行。

3) 实验：测试 `reducetask` 多少合适。

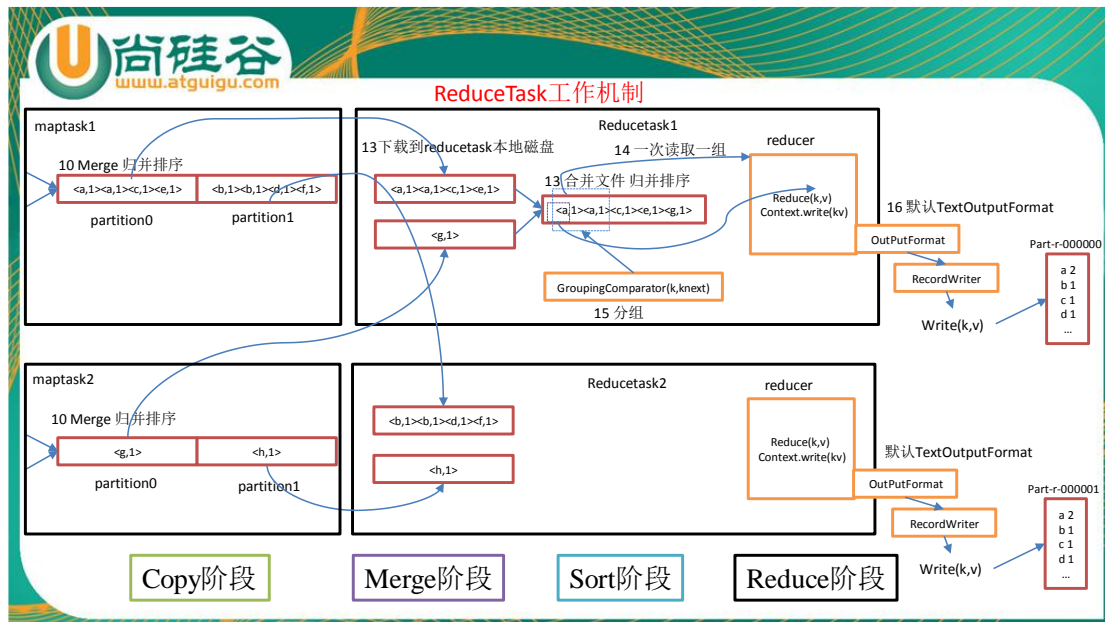
(1) 实验环境：1 个 master 节点，16 个 slave 节点：CPU:8GHZ，内存: 2G

(2) 实验结论：

表 1 改变 `reduce task` （数据量为 1GB）

Map task =16										
Reduce task	1	5	10	15	16	20	25	30	45	60
总时间	892	146	110	92	88	100	128	101	145	104

4) `ReduceTask` 工作机制



(1) Copy 阶段：`ReduceTask` 从各个 `MapTask` 上远程拷贝一片数据，并针对某一片数据，如果其大小超过一定阈值，则写到磁盘上，否则直接放到内存中。

(2) Merge 阶段：在远程拷贝数据的同时，`ReduceTask` 启动了两个后台线程对内存和磁盘上的文件进行合并，以防止内存使用过多或磁盘上文件过多。

(3) Sort 阶段：按照 MapReduce 语义，用户编写 `reduce()` 函数输入数据是按 key 进行聚集的一组数据。为了将 key 相同的数据聚在一起，Hadoop 采用了基于排序的策略。由于各个 `MapTask` 已经实现对自己的处理结果进行了局部排序，因此，`ReduceTask` 只需对所有数据进行一次归并排序即可。

(4) Reduce 阶段：`reduce()` 函数将计算结果写到 HDFS 上。

## 3.6 OutputFormat 数据输出

### 3.6.1 OutputFormat 接口实现类

OutputFormat 是 MapReduce 输出的基类，所有实现 MapReduce 输出都实现了 OutputFormat 接口。下面我们介绍几种常见的 OutputFormat 实现类。

#### 1) 文本输出 TextOutputFormat

默认的输出格式是 TextOutputFormat，它把每条记录写为文本行。它的键和值可以是任意类型，因为 TextOutputFormat 调用 toString()方法把它们转换为字符串。

#### 2) SequenceFileOutputFormat

SequenceFileOutputFormat 将它的输出写为一个顺序文件。如果输出需要作为后续 MapReduce 任务的输入，这便是一种好的输出格式，因为它的格式紧凑，很容易被压缩。

#### 3) 自定义 OutputFormat

根据用户需求，自定义实现输出。

### 3.6.2 自定义 OutputFormat

为了实现控制最终文件的输出路径，可以自定义 OutputFormat。

要在一个 mapreduce 程序中根据数据的不同输出两类结果到不同目录，这类灵活的输出需求可以通过自定义 outputformat 来实现。

#### 1) 自定义 OutputFormat 步骤

- (1) 自定义一个类继承 FileOutputFormat。
- (2) 改写 recordwriter，具体改写输出数据的方法 write()。

#### 2) 实操案例：

详见 7.5 修改日志内容及自定义日志输出路径（自定义 OutputFormat）。

## 3.7 Join 多种应用

### 3.7.1 Reduce join

#### 1) 原理：

Map 端的主要工作：为来自不同表(文件)的 key/value 对打标签以区别不同来源的记录。然后用连接字段作为 key，其余部分和新加的标志作为 value，最后进行输出。

Reduce 端的主要工作：在 reduce 端以连接字段作为 key 的分组已经完成，我们只需要在每一个分组当中将那些来源于不同文件的记录(在 map 阶段已经打标志)分开，最后进行合并

【更多 Java、HTML5、Android、python、大数据 资料下载，可访问尚硅谷（中国）官网 [www.atguigu.com](http://www.atguigu.com) 下载区】

并就 ok 了。

## 2) 该方法的缺点

这种方式的缺点很明显就是会造成 map 和 reduce 端也就是 **shuffle 阶段出现大量的数据传输，效率很低。**

## 3) 案例实操

详见 7.6.1 需求 1: reduce 端表合并（数据倾斜）

## 3.7.2 Map join (Distributedcache 分布式缓存)

1) 使用场景：一张表十分小、一张表很大。

## 2) 解决方案

在 map 端缓存多张表，提前处理业务逻辑，这样增加 map 端业务，减少 reduce 端数据的压力，尽可能的减少数据倾斜。

## 3) 具体办法：采用 distributedcache

(1) 在 mapper 的 setup 阶段，将文件读取到缓存集合中。

(2) 在驱动函数中加载缓存。

```
job.addCacheFile(new URI("file:/e:/mapjoincache/pd.txt")); // 缓存普通文件到 task 运行节点
```

## 4) 实操案例：

详见 7.6.2 需求 2: map 端表合并 (Distributedcache)

## 3.8 数据清洗 (ETL)

### 1) 概述

在运行核心业务 Mapreduce 程序之前，往往要先对数据进行清洗，清理掉不符合用户要求的数据。清理的过程往往只需要运行 mapper 程序，不需要运行 reduce 程序。

### 2) 实操案例

详见 7.7 日志清洗（数据清洗）。

## 3.9 计数器应用

Hadoop 为每个作业维护若干内置计数器，以描述多项指标。例如，某些计数器记录已处理的字节数和记录数，使用户可监控已处理的输入数据量和已产生的输出数据量。

### 1) API

【更多 Java、HTML5、Android、python、大数据 资料下载，可访问尚硅谷（中国）官网 [www.atguigu.com](http://www.atguigu.com) 下载区】

(1) 采用枚举的方式统计计数

```
enum MyCounter{MALFORORMED,NORMAL}  
  
//对枚举定义的自定义计数器加 1  
  
context.getCounter(MyCounter.MALFORORMED).increment(1);
```

(2) 采用计数器组、计数器名称的方式统计

```
context.getCounter("counterGroup", "countera").increment(1);  
  
组名和计数器名称随便起，但最好有意义。
```

(3) 计数结果在程序运行后的控制台上查看。

## 2) 案例实操

详见 7.7 日志清洗（数据清洗）。

## 3.10 MapReduce 开发总结

在编写 mapreduce 程序时，需要考虑的几个方面：

### 1) 输入数据接口：InputFormat

默认使用的实现类是：TextInputFormat

TextInputFormat 的功能逻辑是：一次读一行文本，然后将该行的起始偏移量作为 key，行内容作为 value 返回。

KeyValueTextInputFormat 每一行均为一条记录，被分隔符分割为 key，value。默认分隔符是 tab (\t)。

NlineInputFormat 按照指定的行数 N 来划分切片。

CombineTextInputFormat 可以把多个小文件合并成一个切片处理，提高处理效率。

用户还可以自定义 InputFormat。

### 2) 逻辑处理接口：Mapper

用户根据业务需求实现其中三个方法：map()    setup()    cleanup ()

### 3) Partitioner 分区

有默认实现 HashPartitioner，逻辑是根据 key 的哈希值和 numReduces 来返回一个分区号；key.hashCode()&Integer.MAXVALUE % numReduces

如果业务上有特别的需求，可以自定义分区。

### 4) Comparable 排序

当我们用自定义的对象作为 key 来输出时，就必须要实现 WritableComparable 接口，重  
【更多 Java、HTML5、Android、python、大数据 资料下载，可访问尚硅谷（中国）官网 [www.atguigu.com](http://www.atguigu.com) 下载区】

写其中的 `compareTo()` 方法。

部分排序：对最终输出的每一个文件进行内部排序。

全排序：对所有数据进行排序，通常只有一个 `Reduce`。

二次排序：排序的条件有两个。

#### 5) Combiner 合并

`Combiner` 合并可以提高程序执行效率，减少 `io` 传输。但是使用时必须不能影响原有的业务处理结果。

#### 6) reduce 端分组：Groupingcomparator

`reduceTask` 拿到输入数据（一个 `partition` 的所有数据）后，首先需要对数据进行分组，其分组的默认原则是 `key` 相同，然后对每一组 `kv` 数据调用一次 `reduce()` 方法，并且将这一组 `kv` 中的第一个 `kv` 的 `key` 作为参数传给 `reduce` 的 `key`，将这一组数据的 `value` 的迭代器传给 `reduce()` 的 `values` 参数。

利用上述这个机制，我们可以实现一个高效的分组取最大值的逻辑。

自定义一个 `bean` 对象用来封装我们的数据，然后改写其 `compareTo` 方法产生倒序排序的效果。然后自定义一个 `Groupingcomparator`，将 `bean` 对象的分组逻辑改成按照我们的业务分组 `id` 来分组（比如订单号）。这样，我们要取的最大值就是 `reduce()` 方法中传进来 `key`。

#### 7) 逻辑处理接口：Reducer

用户根据业务需求实现其中三个方法：`reduce()`   `setup()`   `cleanup()`

#### 8) 输出数据接口：OutputFormat

默认实现类是 `TextOutputFormat`，功能逻辑是：将每一个 `KV` 对向目标文本文件中输出为一行。

`SequenceFileOutputFormat` 将它的输出写为一个顺序文件。如果输出需要作为后续 `MapReduce` 任务的输入，这便是一种好的输出格式，因为它的格式紧凑，很容易被压缩。

用户还可以自定义 `OutputFormat`。

## 四 Hadoop 数据压缩

### 4.1 概述

压缩技术能够有效减少底层存储系统（`HDFS`）读写字节数。压缩提高了网络带宽和磁盘空间的效率。在 `Hadoop` 下，尤其是数据规模很大和工作负载密集的情况下，使用数据压

【更多 `Java`、`HTML5`、`Android`、`python`、大数据 资料下载，可访问尚硅谷（中国）官网 [www.atguigu.com](http://www.atguigu.com) 下载区】

缩显得非常重要。在这种情况下，I/O 操作和网络数据传输要花大量的时间。还有，Shuffle 与 Merge 过程同样也面临着巨大的 I/O 压力。

鉴于磁盘 I/O 和网络带宽是 Hadoop 的宝贵资源，数据压缩对于节省资源、最小化磁盘 I/O 和网络传输非常有帮助。不过，尽管压缩与解压操作的 CPU 开销不高，其性能的提升和资源的节省并非没有代价。

如果磁盘 I/O 和网络带宽影响了 MapReduce 作业性能，在任意 MapReduce 阶段启用压缩都可以改善端到端处理时间并减少 I/O 和网络流量。

压缩 Mapreduce 的一种优化策略：通过压缩编码对 Mapper 或者 Reducer 的输出进行压缩，以减少磁盘 IO，提高 MR 程序运行速度（但相应增加了 cpu 运算负担）。

注意：压缩特性运用得当能提高性能，但运用不当也可能降低性能。

基本原则：

(1) 运算密集型的 job，少用压缩

(2) IO 密集型的 job，多用压缩

## 4.2 MR 支持的压缩编码

压缩格式	hadoop 自带?	算法	文件扩展名	是否可切分	换成压缩格式后，原来的程序是否需要修改
DEFAULT	是，直接使用	DEFAULT	.deflate	否	和文本处理一样，不需要修改
Gzip	是，直接使用	DEFAULT	.gz	否	和文本处理一样，不需要修改
bzip2	是，直接使用	bzip2	.bz2	是	和文本处理一样，不需要修改
LZO	否，需要安装	LZO	.lzo	是	需要建索引，还需要指定输入格式
Snappy	否，需要安装	Snappy	.snappy	否	和文本处理一样，不需要修改

为了支持多种压缩/解压缩算法，Hadoop 引入了编码/解码器，如下表所示

压缩格式	对应的编码/解码器
DEFLATE	org.apache.hadoop.io.compress.DefaultCodec
gzip	org.apache.hadoop.io.compress.GzipCodec
bzip2	org.apache.hadoop.io.compress.BZip2Codec
LZO	com.hadoop.compression.lzo.LzopCodec
Snappy	org.apache.hadoop.io.compress.SnappyCodec

压缩性能的比较



压缩算法	原始文件大小	压缩文件大小	压缩速度	解压速度
gzip	8.3GB	1.8GB	17.5MB/s	58MB/s
bzip2	8.3GB	1.1GB	2.4MB/s	9.5MB/s
LZO	8.3GB	2.9GB	49.3MB/s	74.6MB/s

<http://google.github.io/snappy/>

On a single core of a Core i7 processor in 64-bit mode, Snappy **compresses** at about **250 MB/sec** or more and **decompresses** at about **500 MB/sec** or more.

## 4.3 压缩方式选择

### 4.3.1 Gzip 压缩

优点：压缩率比较高，而且压缩/解压速度也比较快；hadoop 本身支持，在应用中处理 gzip 格式的文件就和直接处理文本一样；大部分 linux 系统都自带 gzip 命令，使用方便。

缺点：不支持 split。

应用场景：当每个文件压缩之后在 130M 以内的（1 个块大小内），都可以考虑用 gzip 压缩格式。例如说一天或者一个小时的日志压缩成一个 gzip 文件，运行 mapreduce 程序的时候通过多个 gzip 文件达到并发。hive 程序，streaming 程序，和 java 写的 mapreduce 程序完全和文本处理一样，压缩之后原来的程序不需要做任何修改。

### 4.3.2 Bzip2 压缩

优点：支持 split；具有很高的压缩率，比 gzip 压缩率都高；hadoop 本身支持，但不支持 native；在 linux 系统下自带 bzip2 命令，使用方便。

缺点：压缩/解压速度慢；不支持 native。

应用场景：适合对速度要求不高，但需要较高的压缩率的时候，可以作为 mapreduce 作业的输出格式；或者输出之后的数据比较大，处理之后的数据需要压缩存档减少磁盘空间并且以后数据用得比较少的情况；或者对单个很大的文本文件想压缩减少存储空间，同时又需要支持 split，而且兼容之前的应用程序（即应用程序不需要修改）的情况。

### 4.3.3 Lzo 压缩

优点：压缩/解压速度也比较快，合理的压缩率；支持 split，是 hadoop 中最流行的压缩格式；可以在 linux 系统下安装 lzop 命令，使用方便。

缺点：压缩率比 gzip 要低一些；hadoop 本身不支持，需要安装；在应用中对 lzo 格式的文件需要做一些特殊处理（为了支持 split 需要建索引，还需要指定 inputformat 为 lzo 格式）。

【更多 Java、HTML5、Android、python、大数据 资料下载，可访问尚硅谷（中国）官网 [www.atguigu.com](http://www.atguigu.com) 下载区】

应用场景：一个很大的文本文件，压缩之后还大于 200M 以上的可以考虑，而且单个文件越大，lzo 优点越越明显。

## 4.3.4 Snappy 压缩

优点：高速压缩速度和合理的压缩率。

缺点：不支持 split；压缩率比 gzip 要低；hadoop 本身不支持，需要安装；

应用场景：当 Mapreduce 作业的 Map 输出的数据比较大的时候，作为 Map 到 Reduce 的中间数据的压缩格式；或者作为一个 Mapreduce 作业的输出和另外一个 Mapreduce 作业的输出。

## 4.4 压缩位置选择

压缩可以在 MapReduce 作用的任意阶段启用。



## 4.5 压缩配置参数

要在 Hadoop 中启用压缩，可以配置如下参数：

参数	默认值	阶段	建议
io.compression.codecs (在 core-site.xml 中配置)	org.apache.hadoop.io.compress.DefaultCodec, org.apache.hadoop.io.compress.GzipCodec, org.apache.hadoop.io.compress.BZip2Codec	输入压缩	Hadoop 使用文件扩展名判断是否支持某种编解码器

【更多 Java、HTML5、Android、python、大数据 资料下载，可访问尚硅谷（中国）官网 [www.atguigu.com](http://www.atguigu.com) 下载区】

mapreduce.map.output.compress(在 mapred-site.xml 中配置)	false	mapper 输出	这个参数设为 true 启用压缩
mapreduce.map.output.compress.codec (在 mapred-site.xml 中配置)	org.apache.hadoop.io.compress.DefaultCodec	mapper 输出	使用 LZO 或 snappy 编解码器在此阶段压缩数据
mapreduce.output.fileoutputformat.compress (在 mapred-site.xml 中配置)	false	reducer 输出	这个参数设为 true 启用压缩
mapreduce.output.fileoutputformat.compress.codec(在 mapred-site.xml 中配置)	org.apache.hadoop.io.compress.DefaultCodec	reducer 输出	使用标准工具或者编解码器, 如 gzip 和 bzip2
mapreduce.output.fileoutputformat.compress.type (在 mapred-site.xml 中配置)	RECORD	reducer 输出	SequenceFile 输出使用的压缩类型: NONE 和 BLOCK

## 4.6 压缩实战

压缩案例详见 7.10 压缩/解压缩。

# 五 Yarn

## 5.1 Hadoop1.x 和 Hadoop2.x 架构区别

在 Hadoop1.x 时代, Hadoop 中的 MapReduce 同时处理业务逻辑运算和资源的调度, 耦合性较大。

【更多 Java、HTML5、Android、python、大数据 资料下载, 可访问尚硅谷(中国)官网 [www.atguigu.com](http://www.atguigu.com) 下载区】

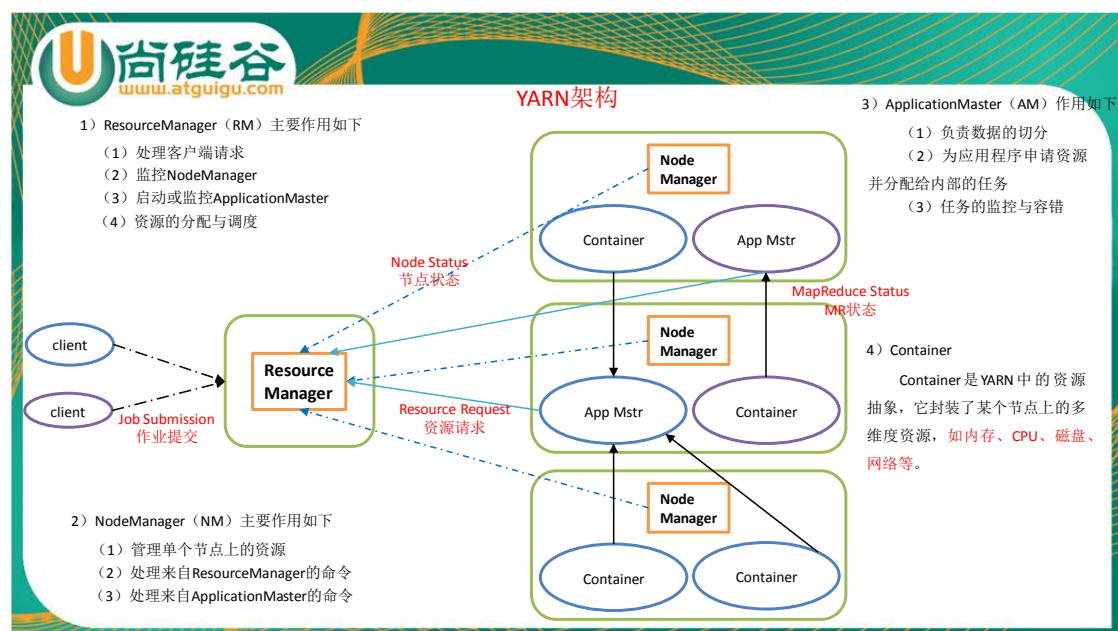
在 Hadoop2.x 时代，增加了 Yarn。Yarn 只负责资源的调度，MapReduce 只负责运算。

## 5.2 Yarn 概述

**Yarn** 是一个资源调度平台，负责为运算程序提供服务器运算资源，相当于一个分布式的操作系统平台，而 **MapReduce** 等运算程序则相当于运行于操作系统之上的应用程序。

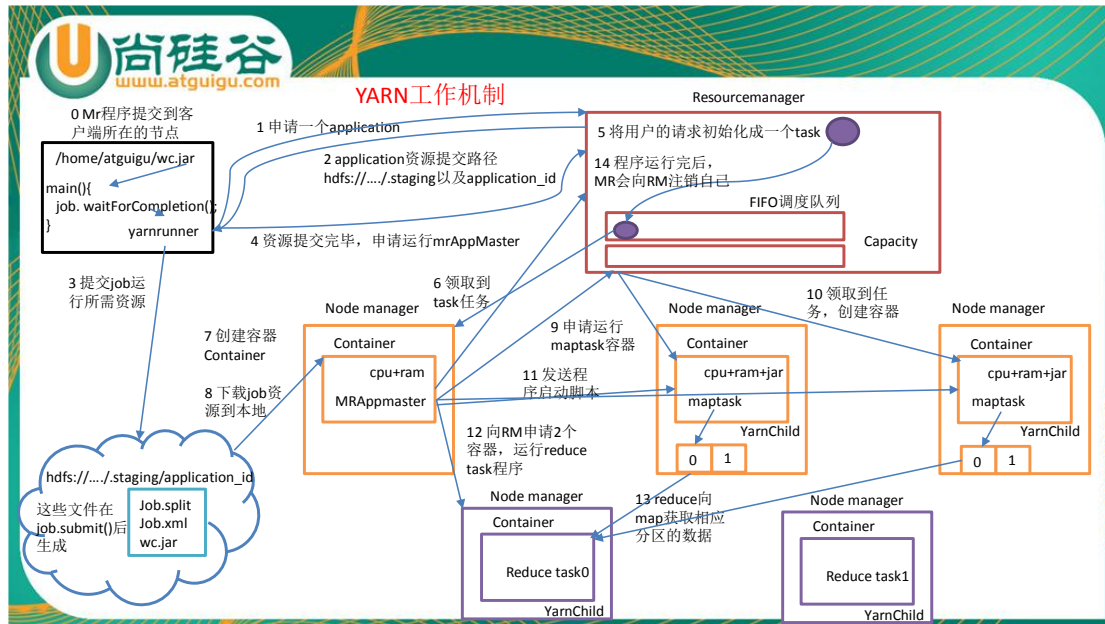
## 5.3 Yarn 基本架构

YARN 主要由 ResourceManager、NodeManager、ApplicationMaster 和 Container 等组件构成。



## 5.4 Yarn 工作机制

### 1) Yarn 运行机制

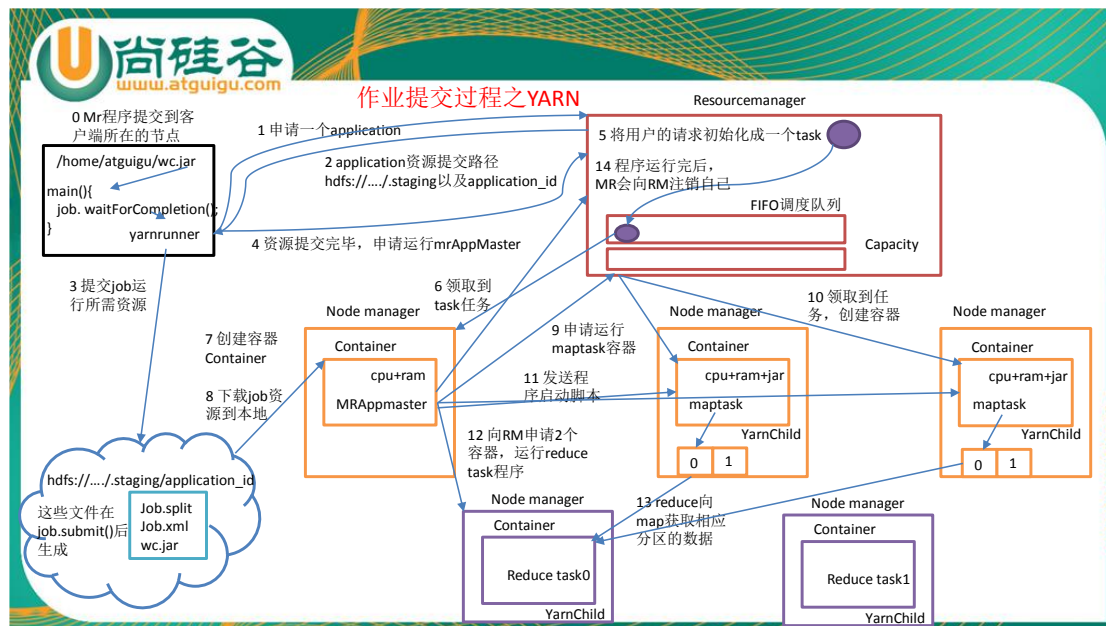


## 2) 工作机制详解

- (0) Mr 程序提交到客户端所在的节点。
- (1) Yarnrunner 向 Resourcemanager 申请一个 Application。
- (2) rm 将该应用程序的资源路径返回给 yarnrunner。
- (3) 该程序将运行所需资源提交到 HDFS 上。
- (4) 程序资源提交完毕后，申请运行 mrAppMaster。
- (5) RM 将用户的请求初始化成一个 task。
- (6) 其中一个 NodeManager 领取到 task 任务。
- (7) 该 NodeManager 创建容器 Container，并产生 MRAppmaster。
- (8) Container 从 HDFS 上拷贝资源到本地。
- (9) MRAppmaster 向 RM 申请运行 maptask 资源。
- (10) RM 将运行 maptask 任务分配给另外两个 NodeManager，另两个 NodeManager 分别领取任务并创建容器。
- (11) MR 向两个接收到任务的 NodeManager 发送程序启动脚本，这两个 NodeManager 分别启动 maptask，maptask 对数据分区排序。
- (12) MrAppMaster 等待所有 maptask 运行完毕后，向 RM 申请容器，运行 reduce task。
- (13) reduce task 向 maptask 获取相应分区的数据。
- (14) 程序运行完毕后，MR 会向 RM 申请注销自己。

## 5.5 作业提交全过程

### 1) 作业提交过程之 YARN



作业提交全过程详解

#### (1) 作业提交

第0步：client 调用 `job.waitForCompletion` 方法，向整个集群提交 MapReduce 作业。

第1步：client 向 RM 申请一个作业 id。

第2步：RM 给 client 返回该 job 资源的提交路径和作业 id。

第3步：client 提交 jar 包、切片信息和配置文件到指定的资源提交路径。

第4步：client 提交完资源后，向 RM 申请运行 `MrAppMaster`。

#### (2) 作业初始化

第5步：当 RM 收到 client 的请求后，将该 job 添加到容量调度器中。

第6步：某一个空闲的 NM 领取到该 job。

第7步：该 NM 创建 Container，并产生 `MRAppmaster`。

第8步：下载 client 提交的资源到本地。

#### (3) 任务分配

第9步：`MrAppMaster` 向 RM 申请运行多个 `maptask` 任务资源。

第10步：RM 将运行 `maptask` 任务分配给另外两个 NodeManager，另两个 NodeManager 分别领取任务并创建容器。

#### (4) 任务运行

【更多 Java、HTML5、Android、python、大数据 资料下载，可访问尚硅谷（中国）官网 [www.atguigu.com](http://www.atguigu.com) 下载区】



第 11 步: MR 向两个接收到任务的 NodeManager 发送程序启动脚本, 这两个 NodeManager 分别启动 maptask, maptask 对数据分区排序。

第 12 步: MrAppMaster 等待所有 maptask 运行完毕后, 向 RM 申请容器, 运行 reduce task。

第 13 步: reduce task 向 maptask 获取相应分区的数据。

第 14 步: 程序运行完毕后, MR 会向 RM 申请注销自己。

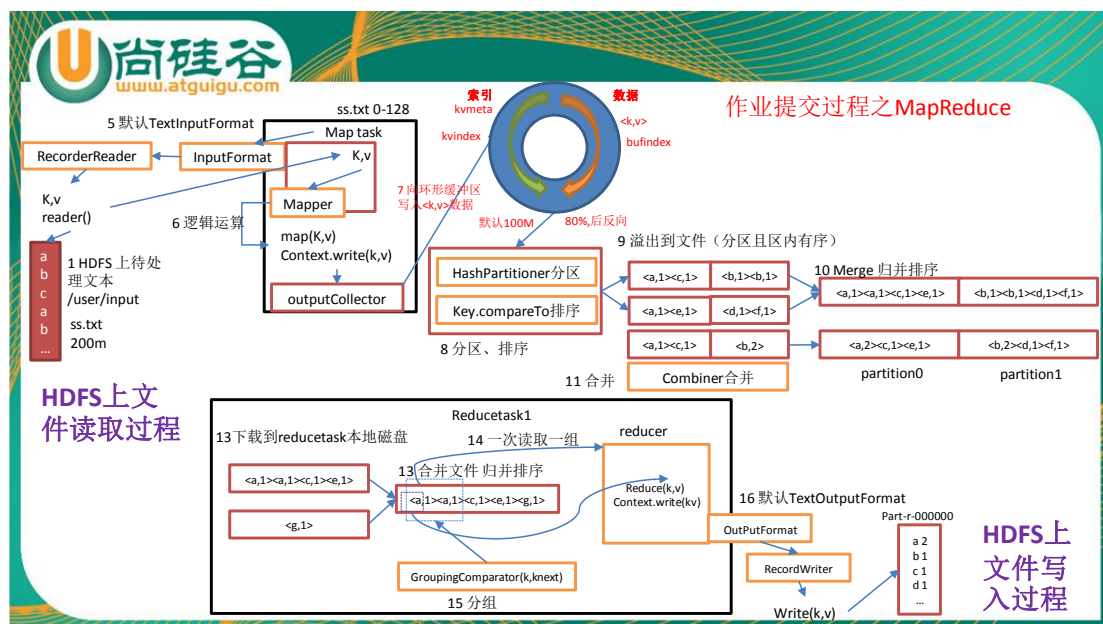
#### (5) 进度和状态更新

YARN 中的任务将其进度和状态(包括 counter)返回给应用管理器, 客户端每秒(通过 `mapreduce.client.progressmonitor.pollinterval` 设置)向应用管理器请求进度更新, 展示给用户。

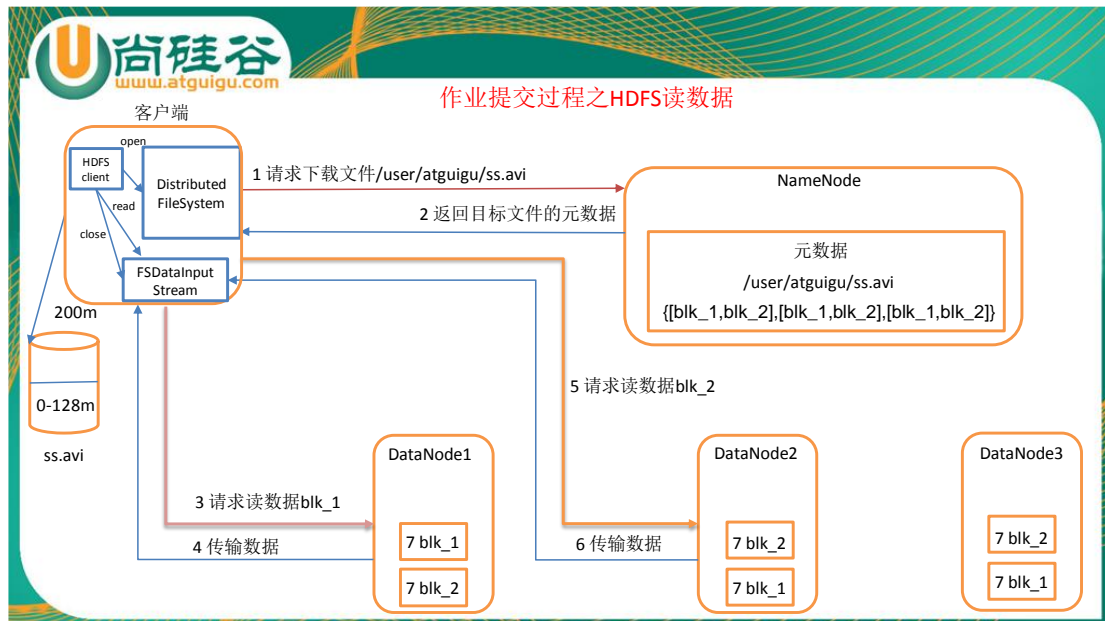
#### (6) 作业完成

除了向应用管理器请求作业进度外, 客户端每 5 分钟都会通过调用 `waitForCompletion()` 来检查作业是否完成。时间间隔可以通过 `mapreduce.client.completion.pollinterval` 来设置。作业完成之后, 应用管理器和 container 会清理工作状态。作业的信息会被作业历史服务器存储以备之后用户核查。

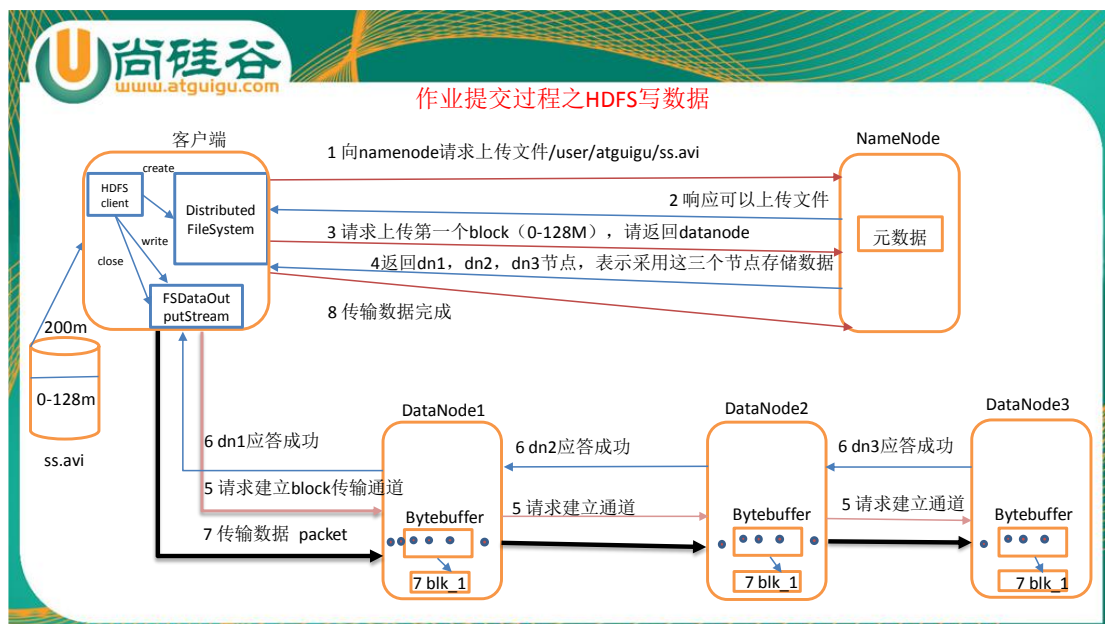
### 2) 作业提交过程之 MapReduce



### 3) 作业提交过程之读数据



#### 4) 作业提交过程之写数据



## 5.6 资源调度器

目前，Hadoop 作业调度器主要有三种：FIFO、Capacity Scheduler 和 Fair Scheduler。

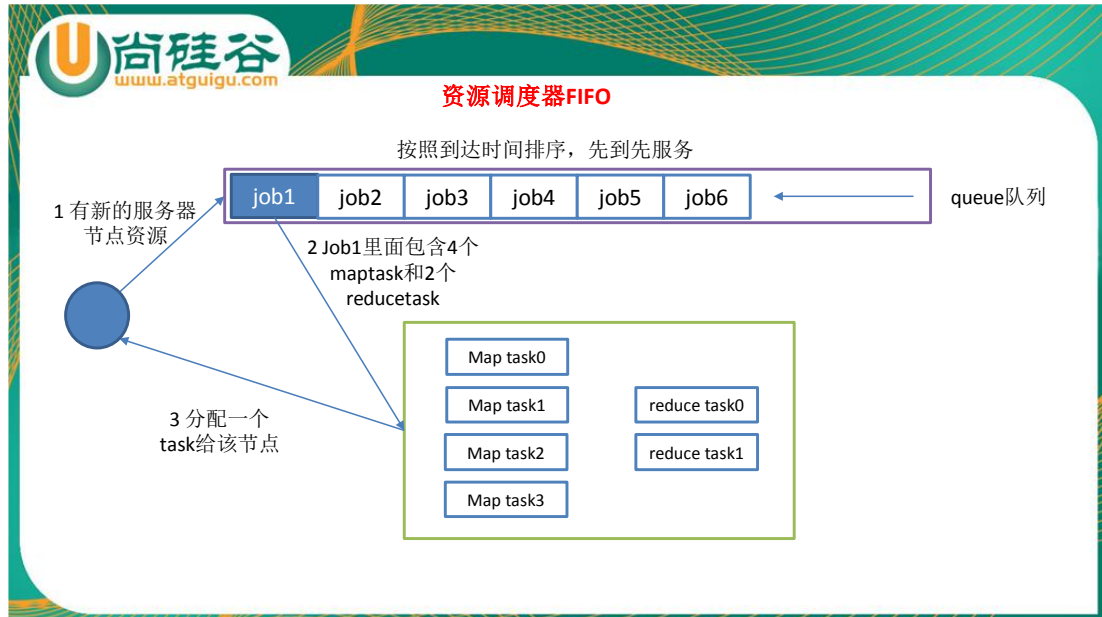
Hadoop2.7.2 默认的资源调度器是 Capacity Scheduler。

具体设置详见：yarn-default.xml 文件

```
<property>
  <description>The class to use as the resource scheduler.</description>
  <name>yarn.resourcemanager.scheduler.class</name>
  <value>org.apache.hadoop.yarn.server.resourcemanager.scheduler.capacity.CapacityScheduler</value>
</property>
```

</property>

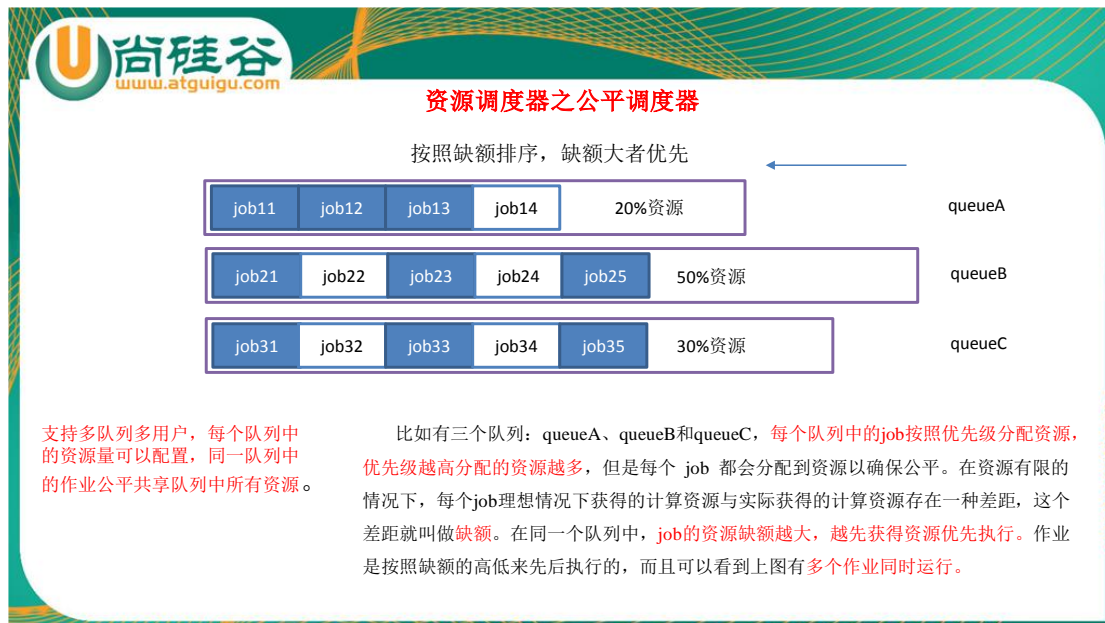
### 1) 先进先出调度器 (FIFO)



### 2) 容量调度器 (Capacity Scheduler)



### 3) 公平调度器 (Fair Scheduler)



## 5.7 任务的推测执行

### 1) 作业完成时间取决于最慢的任务完成时间

一个作业由若干个 Map 任务和 Reduce 任务构成。因硬件老化、软件 Bug 等，某些任务可能运行非常慢。

典型案例：系统中有 99% 的 Map 任务都完成了，只有少数几个 Map 老是进度很慢，完不成，怎么办？

### 2) 推测执行机制：

发现拖后腿的任务，比如某个任务运行速度远慢于任务平均速度。为拖后腿任务启动一个备份任务，同时运行。谁先运行完，则采用谁的结果。

### 3) 执行推测任务的前提条件

- (1) 每个 task 只能有一个备份任务；
- (2) 当前 job 已完成的 task 必须不小于 0.05 (5%)
- (3) 开启推测执行参数设置。Hadoop2.7.2 mapred-site.xml 文件中默认是打开的。

```
<property>
  <name>mapreduce.map.speculative</name>
  <value>true</value>
  <description>If true, then multiple instances of some map tasks
may be executed in parallel.</description>
</property>

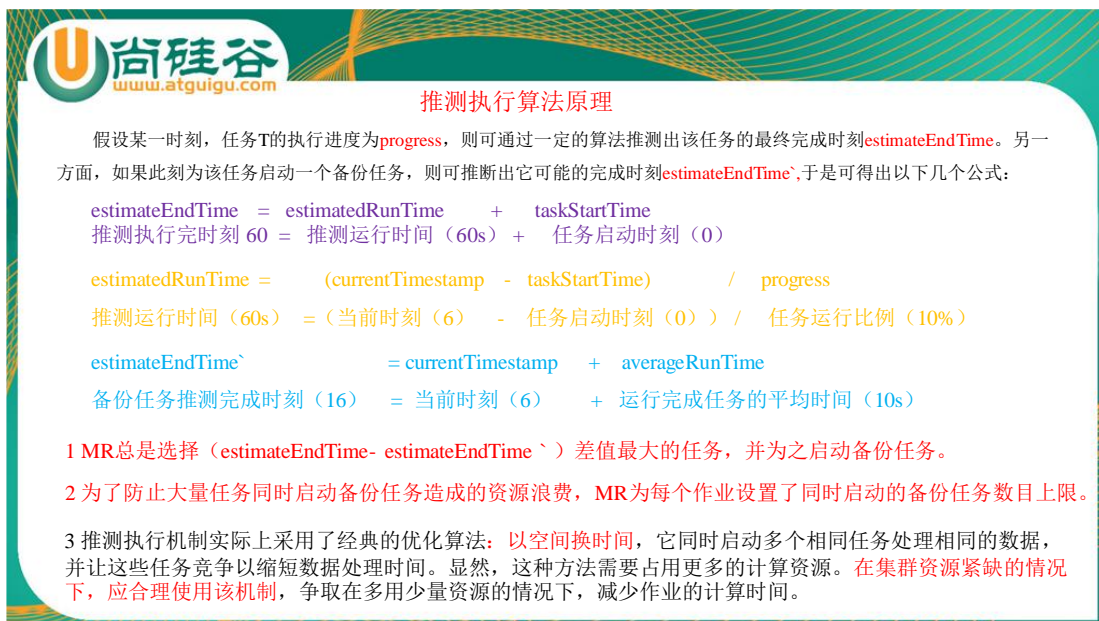
<property>
```

```
<name>mapreduce.reduce.speculative</name>
<value>true</value>
<description>If true, then multiple instances of some reduce tasks
may be executed in parallel.</description>
</property>
```

#### 4) 不能启用推测执行机制情况

- (1) 任务间存在严重的负载倾斜;
- (2) 特殊任务, 比如任务向数据库中写数据。

#### 5) 算法原理:



**推测执行算法原理**

假设某一时刻, 任务T的执行进度为progress, 则可通过一定的算法推测出该任务的最终完成时刻`estimateEndTime`。另一方面, 如果此刻为该任务启动一个备份任务, 则可推断出它可能的完成时刻`estimateEndTime``, 于是可得出以下几个公式:

$$\begin{aligned} \text{estimateEndTime} &= \text{estimatedRunTime} + \text{taskStartTime} \\ \text{推测执行完时刻 } 60 &= \text{推测运行时间 (60s)} + \text{任务启动时刻 (0)} \\ \text{estimatedRunTime} &= (\text{currentTimestamp} - \text{taskStartTime}) / \text{progress} \\ \text{推测运行时间 (60s)} &= (\text{当前时刻 (6)} - \text{任务启动时刻 (0)}) / \text{任务运行比例 (10\%)} \\ \text{estimateEndTime`} &= \text{currentTimestamp} + \text{averageRunTime} \\ \text{备份任务推测完成时刻 (16)} &= \text{当前时刻 (6)} + \text{运行完成任务的平均时间 (10s)} \end{aligned}$$

1 MR总是选择 $(\text{estimateEndTime} - \text{estimateEndTime`})$ 差值最大的任务, 并为之启动备份任务。

2 为了防止大量任务同时启动备份任务造成的资源浪费, MR为每个作业设置了同时启动的备份任务数目上限。

3 推测执行机制实际上采用了经典的优化算法: 以空间换时间, 它同时启动多个相同任务处理相同的数据, 并让这些任务竞争以缩短数据处理时间。显然, 这种方法需要占用更多的计算资源。在集群资源紧缺的情况下, 应合理使用该机制, 争取在多用少量资源的情况下, 减少作业的计算时间。

## 六 Hadoop 企业优化

### 6.1 MapReduce 跑的慢的原因

Mapreduce 程序效率的瓶颈在于两点:

#### 1) 计算机性能

CPU、内存、磁盘健康、网络

#### 2) I/O 操作优化

- (1) 数据倾斜
- (2) map 和 reduce 数设置不合理
- (3) map 运行时间太长, 导致 reduce 等待过久
- (4) 小文件过多
- (5) 大量的不可分块的超大文件



- (6) spill 次数过多
- (7) merge 次数过多等。

## 6.2 MapReduce 优化方法

MapReduce 优化方法主要从六个方面考虑：数据输入、Map 阶段、Reduce 阶段、IO 传输、数据倾斜问题和常用的调优参数。

### 6.2.1 数据输入

- (1) 合并小文件：在执行 mr 任务前将小文件进行合并，大量的小文件会产生大量的 map 任务，增大 map 任务装载次数，而任务的装载比较耗时，从而导致 mr 运行较慢。
- (2) 采用 CombineTextInputFormat 来作为输入，解决输入端大量小文件场景。

### 6.2.2 Map 阶段

- 1) **减少溢写 (spill) 次数**：通过调整 io.sort.mb 及 sort.spill.percent 参数值，增大触发 spill 的内存上限，减少 spill 次数，从而减少磁盘 IO。
- 2) **减少合并 (merge) 次数**：通过调整 io.sort.factor 参数，增大 merge 的文件数目，减少 merge 的次数，从而缩短 mr 处理时间。
- 3) 在 map 之后，不影响业务逻辑前提下，先进行 combine 处理，减少 I/O。

### 6.2.3 Reduce 阶段

- 1) **合理设置 map 和 reduce 数**：两个都不能设置太少，也不能设置太多。太少，会导致 task 等待，延长处理时间；太多，会导致 map、reduce 任务间竞争资源，造成处理超时等错误。
- 2) **设置 map、reduce 共存**：调整 slowstart.completedmaps 参数，使 map 运行到一定程度后，reduce 也开始运行，减少 reduce 的等待时间。
- 3) **规避使用 reduce**：因为 reduce 在用于连接数据集的时候将会产生大量的网络消耗。
- 4) **合理设置 reduce 端的 buffer**：默认情况下，数据达到一个阈值的时候，buffer 中的数据就会写入磁盘，然后 reduce 会从磁盘中获得所有的数据。也就是说，buffer 和 reduce 是没有直接关联的，中间多个一个写磁盘->读磁盘的过程，既然有这个弊端，那么就可以通过参数来配置，使得 buffer 中的一部分数据可以直接输送到 reduce，从而减少 IO 开销：mapred.job.reduce.input.buffer.percent，默认为 0.0。当值大于 0 的时候，会保留指定比例的内存读 buffer 中的数据直接拿给 reduce 使用。这样一来，设置 buffer 需要内存，读取数据需【更多 Java、HTML5、Android、python、大数据 资料下载，可访问尚硅谷（中国）官网 [www.atguigu.com](http://www.atguigu.com) 下载区】

要内存，reduce 计算也要内存，所以要根据作业的运行情况进行调整。

#### 6.2.4 IO 传输

- 1) 采用数据压缩的方式，减少网络 IO 的时间。安装 Snappy 和 LZ4 压缩编码器。
- 2) 使用 SequenceFile 二进制文件。

#### 6.2.5 数据倾斜问题

- 1) 数据倾斜现象

数据频率倾斜——某一个区域的数据量要远远大于其他区域。

数据大小倾斜——部分记录的大小远远大于平均值。

- 2) 如何收集倾斜数据

在 reduce 方法中加入记录 map 输出键的详细情况的功能。

```
public static final String MAX_VALUES = "skew.maxvalues";
private int maxThreshold;

@Override
public void configure(JobConf job) {
    maxThreshold = job.getInt(MAX_VALUES, 100);
}

@Override
public void reduce(Text key, Iterator<Text> values,
    OutputCollector<Text, Text> output,
    Reporter reporter) throws IOException {
    int i = 0;
    while (values.hasNext()) {
        values.next();
        i++;
    }

    if (i > maxThreshold) {
        log.info("Received " + i + " values for key " + key);
    }
}
```

- 3) 减少数据倾斜的方法

##### 方法 1: 抽样和范围分区

可以通过对原始数据进行抽样得到的结果集来预设分区边界值。

##### 方法 2: 自定义分区

基于输出键的背景知识进行自定义分区。例如，如果 map 输出键的单词来源于一本书。

【更多 Java、HTML5、Android、python、大数据 资料下载，可访问尚硅谷（中国）官网 [www.atguigu.com](http://www.atguigu.com) 下载区】



且其中某几个专业词汇较多。那么就可以自定义分区将这这些专业词汇发送给固定的一部分 reduce 实例。而将其他的都发送给剩余的 reduce 实例。

### 方法 3: Combine

使用 Combine 可以大量地减小数据倾斜。在可能的情况下, combine 的目的就是聚合并精简数据。

### 方法 4: 采用 Map Join, 尽量避免 Reduce Join。

## 6.2.6 常用的调优参数

### 1) 资源相关参数

(1) 以下参数是在用户自己的 mr 应用程序中配置就可以生效 (mapred-default.xml)

配置参数	参数说明
mapreduce.map.memory.mb	一个 Map Task 可使用的资源上限 (单位:MB), 默认为 1024。如果 Map Task 实际使用的资源量超过该值, 则会被强制杀死。
mapreduce.reduce.memory.mb	一个 Reduce Task 可使用的资源上限 (单位:MB), 默认为 1024。如果 Reduce Task 实际使用的资源量超过该值, 则会被强制杀死。
mapreduce.map.cpu.vcores	每个 Map task 可使用的最多 cpu core 数目, 默认值: 1
mapreduce.reduce.cpu.vcores	每个 Reduce task 可使用的最多 cpu core 数目, 默认值: 1
mapreduce.reduce.shuffle.parallelcopies	每个 reduce 去 map 中拿数据的并行数。默认值是 5
mapreduce.reduce.shuffle.merge.percent	buffer 中的数据达到多少比例开始写入磁盘。默认值 0.66
mapreduce.reduce.shuffle.input.buffer.percent	buffer 大小占 reduce 可用内存的比例。默认值 0.7
mapreduce.reduce.input.buffer.percent	指定多少比例的内存用来存放 buffer 中的数据, 默认值是 0.0

(2) 应该在 yarn 启动之前就配置在服务器的配置文件中才能生效 (yarn-default.xml)

配置参数	参数说明
------	------

【更多 Java、HTML5、Android、python、大数据 资料下载, 可访问尚硅谷 (中国) 官网 [www.atguigu.com](http://www.atguigu.com) 下载区】

yarn.scheduler.minimum-allocation-mb	1024	给应用程序 container 分配的最小内存
yarn.scheduler.maximum-allocation-mb	8192	给应用程序 container 分配的最大内存
yarn.scheduler.minimum-allocation-vcores	1	每个 container 申请的最小 CPU 核数
yarn.scheduler.maximum-allocation-vcores	32	每个 container 申请的最大 CPU 核数
yarn.nodemanager.resource.memory-mb	8192	给 containers 分配的最大物理内存

(3) shuffle 性能优化的关键参数, 应在 yarn 启动之前就配置好 (mapred-default.xml)

配置参数	参数说明
mapreduce.task.io.sort.mb      100	shuffle 的环形缓冲区大小，默认 100m
mapreduce.map.sort.spill.percent      0.8	环形缓冲区溢出的阈值，默认 80%

## 2) 容错相关参数(mapreduce 性能优化)

配置参数	参数说明
mapreduce.map.maxattempts	每个 Map Task 最大重试次数, 一旦重试参数超过该值, 则认为 Map Task 运行失败, 默认值: 4。
mapreduce.reduce.maxattempts	每个 Reduce Task 最大重试次数, 一旦重试参数超过该值, 则认为 Map Task 运行失败, 默认值: 4。
mapreduce.task.timeout	Task 超时时间, 经常需要设置的一个参数, 该参数表达的意思为: 如果一个 task 在一定时间内没有任何进入, 即不会读取新的数据, 也没有输出数据, 则认为该 task 处于 block 状态, 可能是卡住了, 也许永远会卡主, 为了防止因为用户程序永远 block 住不退出, 则强制设置了一个该超时时间 (单位毫秒), 默认是 600000。如果你的程序对每条输入数据的处理时间过长 (比如会访问数据库, 通过网络拉取数据等), 建议将该参数调大, 该参数过小常出现的错误提示是 “AttemptID:attempt_14267829456721_123456_m_000224_0 Timed out after 300 secsContainer killed by the ApplicationMaster。”。

## 6.3 HDFS 小文件优化方法

### 6.3.1 HDFS 小文件弊端

HDFS 上每个文件都要在 namenode 上建立一个索引, 这个索引的大小约为 150byte, 这样当小文件比较多时, 就会产生很多的索引文件, 一方面会大量占用 namenode 的内存空间, 另一方面就是索引文件过大是索引速度变慢。

### 6.3.2 解决方案

1) Hadoop Archive:

【更多 Java、HTML5、Android、python、大数据 资料下载, 可访问尚硅谷 (中国) 官网 [www.atguigu.com](http://www.atguigu.com) 下载区】

是一个高效地将小文件放入 HDFS 块中的文件存档工具，它能够将多个小文件打包成一个 HAR 文件，这样就减少了 namenode 的内存使用。

#### 2) Sequence file:

sequence file 由一系列的二进制 key/value 组成，如果 key 为文件名，value 为文件内容，则可以将大批小文件合并成一个大文件。

#### 3) CombineFileInputFormat:

CombineFileInputFormat 是一种新的 inputformat，用于将多个文件合并成一个单独的 split，另外，它会考虑数据的存储位置。

#### 4) 开启 JVM 重用

对于大量小文件 Job，可以开启 JVM 重用会减少 45% 运行时间。

JVM 重用理解：一个 map 运行一个 jvm，重用的话，在一个 map 在 jvm 上运行完毕后，jvm 继续运行其他 map。

具体设置：mapreduce.job.jvm.numtasks 值在 10-20 之间。

## 七 MapReduce 实战

### 7.1 WordCount 案例

#### 7.1.1 需求 1：统计一堆文件中单词出现的个数

0) 需求：在一堆给定的文本文件中统计输出每一个单词出现的总次数

##### 1) 数据准备:



hello.txt

##### 2) 分析

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

### 3.1.1 需求1: 统计一堆文件中单词出现的个数 (WordCount案例)

#### Mapper

// 1 将maptask传给我们的文本内容先转换成String

// 2 根据空格将这一行切分成单词

// 3 将单词输出为<单词, 1>

#### Reducer

// 1 汇总各个key的个数

// 2 输出该key的总次数

#### Driver

// 1 获取配置信息, 或者job对象实例

// 2 指定本程序的jar包所在的本地路径

// 3 关联mapper/Reducer业务类

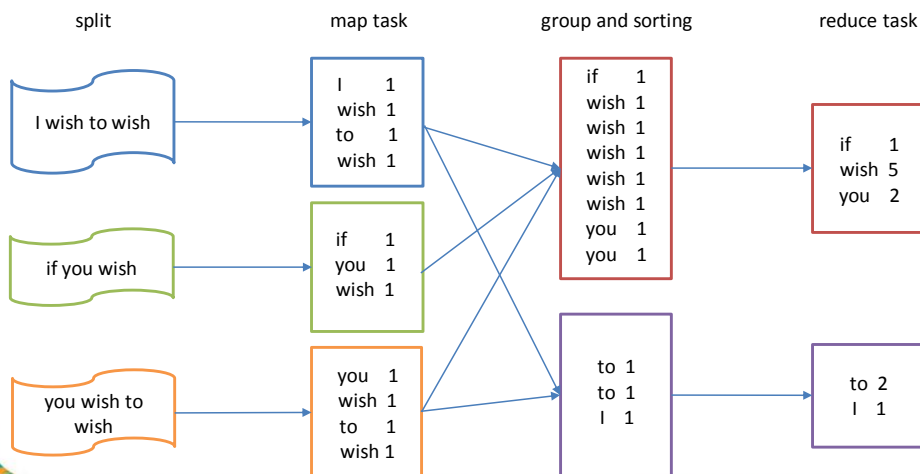
// 4 指定mapper输出数据的kv类型

// 5 指定最终输出的数据的kv类型

// 6 指定job的输入原始文件所在目录

// 7 提交

### Wordcount案例简单分析



## 3) 编写程序

### (1) 编写 mapper 类

```
package com.atguigu.mapreduce;
import java.io.IOException;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

public class WordcountMapper extends Mapper<LongWritable, Text, Text, IntWritable>{

    Text k = new Text();
```

```
IntWritable v = new IntWritable(1);

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

    // 1 获取一行
    String line = value.toString();

    // 2 切割
    String[] words = line.split(" ");

    // 3 输出
    for (String word : words) {

        k.set(word);
        context.write(k, v);
    }
}
```

## (2) 编写 reducer 类

```
package com.atguigu.mapreduce.wordcount;
import java.io.IOException;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

public class WordcountReducer extends Reducer<Text, IntWritable, Text, IntWritable>{

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

        // 1 累加求和
        int sum = 0;
        for (IntWritable count : value) {
            sum += count.get();
        }

        // 2 输出
        context.write(key, new IntWritable(sum));
    }
}
```

## (3) 编写驱动类

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

public class WordcountDriver {

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

        // 1 获取配置信息
        Configuration configuration = new Configuration();
        Job job = Job.getInstance(configuration);

        // 2 设置 jar 加载路径
        job.setJarByClass(WordcountDriver.class);

        // 3 设置 map 和 Reduce 类
        job.setMapperClass(WordcountMapper.class);
        job.setReducerClass(WordcountReducer.class);

        // 4 设置 map 输出
        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(IntWritable.class);

        // 5 设置 Reduce 输出
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);

        // 6 设置输入和输出路径
        FileInputFormat.setInputPaths(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        // 7 提交
        boolean result = job.waitForCompletion(true);

        System.exit(result ? 0 : 1);
    }
}
```

```
}  
}
```

#### 4) 集群上测试

- (1) 将程序打成 jar 包，然后拷贝到 hadoop 集群中。
- (2) 启动 hadoop 集群
- (3) 执行 wordcount 程序

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

#### 5) 本地测试

- (1) 在 windows 环境中配置 HADOOP\_HOME 环境变量。
- (2) 在 eclipse 上运行程序
- (3) 注意：如果 eclipse 打印不出日志，在控制台上只显示

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

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

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

### 7.1.2 需求 2：把单词按照 ASCII 码奇偶分区 (Partitioner)

#### 0) 分析





## 3.1.2 需求2: 把单词按照ASCII码奇偶分区 (Partitioner)

1) 自定义分区

// 1 获取单词key

// 2 根据奇数偶数分区

2) 在驱动中配置加载分区, 设置reducetask个数

job.setPartitionerClass(WordCountPartitioner.class);

job.setNumReduceTasks(2);

## 1) 自定义分区

```
package com.atguigu.mapreduce.wordcount;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Partitioner;

public class WordCountPartitioner extends Partitioner<Text, IntWritable>{

    @Override
    public int getPartition(Text key, IntWritable value, int numPartitions) {

        // 1 获取单词 key
        String firWord = key.toString().substring(0, 1);
        char[] charArray = firWord.toCharArray();
        int result = charArray[0];
        // int result = key.toString().charAt(0);

        // 2 根据奇数偶数分区
        if (result % 2 == 0) {
            return 0;
        } else {
            return 1;
        }
    }
}
```

## 2) 在驱动中配置加载分区, 设置 reducetask 个数

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

```
job.setNumReduceTasks(2);
```

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

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

 尚硅谷  
www.atguigu.com  
3.1.3 需求3：对每一个maptask的输出局部汇总（Combiner）

方案一	方案二
1) 增加一个WordcountCombiner类继承Reducer	1) 将 WordcountReducer 作为 combiner 在 WordcountDriver驱动类中指定
2) 在WordcountCombiner中 (1) 统计单词汇总 (2) 将统计结果输出	<pre>job.setCombinerClass(WordcountReducer.class);</pre>

1) 数据准备：



hello.txt

方案一

1) 增加一个 WordcountCombiner 类继承 Reducer

```
package com.atguigu.mr.combiner;
import java.io.IOException;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

public class WordcountCombiner extends Reducer<Text, IntWritable, Text, IntWritable>{

    @Override
    protected void reduce(Text key, Iterable<IntWritable> values,
        Context context) throws IOException, InterruptedException {
        // 1 汇总
        int count = 0;
        for(IntWritable v : values){
            count += v.get();
        }
    }
}
```

【更多 Java、HTML5、Android、python、大数据 资料下载，可访问尚硅谷（中国）官网 [www.atguigu.com](http://www.atguigu.com) 下载区】

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

2) 在 WordcountDriver 驱动类中指定 combiner

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

## 方案二

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

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

运行程序

```
Map-Reduce Framework  
  Map input records=5  
  Map output records=8  
  Map output bytes=72  
  Map output materialized bytes=94  
  Input split bytes=199  
  Combine input records=0  
  Combine output records=0  
  Reduce input groups=6  
  Reduce shuffle bytes=94
```

未使用前

```
Map-Reduce Framework  
  Map input records=5  
  Map output records=8  
  Map output bytes=72  
  Map output materialized bytes=72  
  Input split bytes=199  
  Combine input records=8  
  Combine output records=6  
  Reduce input groups=6  
  Reduce shuffle bytes=72
```

使用后

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

0) 需求: 将输入的大量小文件合并成一个切片统一处理。

1) 输入数据: 准备 5 个小文件

2) 实现过程

(1) 不做任何处理, 运行需求 1 中的 wordcount 程序, 观察切片个数为 5

```
2017-05-31 10:15:48,759 INFO [org.apache.hadoop.mapreduce.lib.input.FileInputFormat] - Total input p  
2017-05-31 10:15:48,787 INFO [org.apache.hadoop.mapreduce.JobSubmitter] - number of splits:5
```

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

【更多 Java、HTML5、Android、python、大数据 资料下载, 可访问尚硅谷 (中国) 官网 [www.atguigu.com](http://www.atguigu.com) 下载区】

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

```
2017-05-31 10:37:17,595 INFO [org.apache.hadoop.mapreduce.lib.input.CombineFileInputFormat] - DEB
2017-05-31 10:37:17,611 INFO [org.apache.hadoop.mapreduce.JobSubmitter] - number of splits:1
```

## 7.2 流量汇总案例

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

#### 1) 需求:

统计每一个手机号耗费的总上行流量、下行流量、总流量

#### 2) 数据准备



phone\_data.txt

输入数据格式:

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

输出数据格式

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

#### 3) 分析

基本思路:

Map 阶段:

- (1) 读取一行数据, 切分字段
- (2) 抽取手机号、上行流量、下行流量
- (3) 以手机号为 key, bean 对象为 value 输出, 即 context.write(手机号,bean);

Reduce 阶段:

- (1) 累加上行流量和下行流量得到总流量。
- (2) 实现自定义的 bean 来封装流量信息, 并将 bean 作为 map 输出的 key 来传输
- (3) MR 程序在处理数据的过程中会对数据排序(map 输出的 kv 对传输到 reduce 之前,

会排序), 排序的依据是 map 输出的 key

【更多 Java、HTML5、Android、python、大数据 资料下载, 可访问尚硅谷(中国)官网 [www.atguigu.com](http://www.atguigu.com) 下载区】

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

然后重写 key 的 compareTo 方法。

#### 4) 编写 mapreduce 程序

##### (1) 编写流量统计的 bean 对象

```
package com.atguigu.mapreduce.flowsun;
import java.io.DataInput;
import java.io.DataOutput;
import java.io.IOException;
import org.apache.hadoop.io.Writable;

// 1 实现 writable 接口
public class FlowBean implements Writable{

    private long upFlow ;
    private long downFlow;
    private long sumFlow;

    //2 反序列化时，需要反射调用空参构造函数，所以必须有
    public FlowBean() {
        super();
    }

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

    //3 写序列化方法
    @Override
    public void write(DataOutput out) throws IOException {
        out.writeLong(upFlow);
        out.writeLong(downFlow);
        out.writeLong(sumFlow);
    }

    //4 反序列化方法
    //5 反序列化方法读顺序必须和写序列化方法的写顺序必须一致
    @Override
```

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

// 6 编写 toString 方法，方便后续打印到文本
@Override
public String toString() {
    return upFlow + "\t" + downFlow + "\t" + sumFlow;
}

public long getUpFlow() {
    return upFlow;
}

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

public long getDownFlow() {
    return downFlow;
}

public void setDownFlow(long downFlow) {
    this.downFlow = downFlow;
}

public long getSumFlow() {
    return sumFlow;
}

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

## (2) 编写 mapper

```
package com.atguigu.mapreduce.flowsum;
import java.io.IOException;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;
```

```
public class FlowCountMapper extends Mapper<LongWritable, Text, Text, FlowBean>{

    FlowBean v = new FlowBean();
    Text k = new Text();

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

        // 1 获取一行
        String line = value.toString();

        // 2 切割字段
        String[] fields = line.split("\t");

        // 3 封装对象
        // 取出手机号码
        String phoneNum = fields[1];
        // 取出上行流量和下行流量
        long upFlow = Long.parseLong(fields[fields.length - 3]);
        long downFlow = Long.parseLong(fields[fields.length - 2]);

        v.set(downFlow, upFlow);

        // 4 写出
        context.write(new Text(phoneNum), new FlowBean(upFlow, downFlow));
    }
}
```

### (3) 编写 reducer

```
package com.atguigu.mapreduce.flowsum;
import java.io.IOException;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

public class FlowCountReducer extends Reducer<Text, FlowBean, Text, FlowBean> {

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

        long sum_upFlow = 0;
        long sum_downFlow = 0;
```



```
// 1 遍历所用 bean，将其中的上行流量，下行流量分别累加
for (FlowBean flowBean : values) {
    sum_upFlow += flowBean.getSumFlow();
    sum_downFlow += flowBean.getDownFlow();
}

// 2 封装对象
FlowBean resultBean = new FlowBean(sum_upFlow, sum_downFlow);

// 3 写出
context.write(key, resultBean);
}
}
```

#### (4) 编写驱动

```
package com.atguigu.mapreduce.flowsun;
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 FlowsunDriver {

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

        // 1 获取配置信息，或者 job 对象实例
        Configuration configuration = new Configuration();
        Job job = Job.getInstance(configuration);

        // 6 指定本程序的 jar 包所在的本地路径
        job.setJarByClass(FlowsunDriver.class);

        // 2 指定本业务 job 要使用的 mapper/Reducer 业务类
        job.setMapperClass(FlowCountMapper.class);
        job.setReducerClass(FlowCountReducer.class);

        // 3 指定 mapper 输出数据的 kv 类型
        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(FlowBean.class);
```

```
// 4 指定最终输出的数据的 kv 类型
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(FlowBean.class);

// 5 指定 job 的输入原始文件所在目录
FileInputFormat.setInputPaths(job, new Path(args[0]));
FileOutputFormat.setOutputPath(job, new Path(args[1]));

// 7 将 job 中配置的相关参数, 以及 job 所用的 java 类所在的 jar 包, 提交给
yarn 去运行
boolean result = job.waitForCompletion(true);
System.exit(result ? 0 : 1);
}
}
```

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

0) 需求: 将统计结果按照手机归属地不同省份输出到不同文件中 (分区)

1) 数据准备



phone\_data.txt

2) 分析

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

(2) 如果要按照我们自己的需求进行分组, 则需要改写数据分发 (分组) 组件 Partitioner 自定义一个 CustomPartitioner 继承抽象类: Partitioner

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

3) 在需求 1 的基础上, 增加一个分区类

```
package com.atguigu.mapreduce.flowsun;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Partitioner;

public class ProvincePartitioner extends Partitioner<Text, FlowBean> {

    @Override
    public int getPartition(Text key, FlowBean value, int numPartitions) {
```

```
// 1 获取电话号码的前三位
String preNum = key.toString().substring(0, 3);

int partition = 4;

// 2 判断是哪个省
if ("136".equals(preNum)) {
    partition = 0;
}else if ("137".equals(preNum)) {
    partition = 1;
}else if ("138".equals(preNum)) {
    partition = 2;
}else if ("139".equals(preNum)) {
    partition = 3;
}

return partition;
}
}
```

## 2) 在驱动函数中增加自定义数据分区设置和 reduce task 设置

```
package com.atguigu.mapreduce.flowsun;
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 FlowsunDriver {

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

        // 1 获取配置信息, 或者 job 对象实例
        Configuration configuration = new Configuration();
        Job job = Job.getInstance(configuration);

        // 6 指定本程序的 jar 包所在的本地路径
        job.setJarByClass(FlowsunDriver.class);

        // 2 指定本业务 job 要使用的 mapper/Reducer 业务类
        job.setMapperClass(FlowCountMapper.class);
    }
}
```

```
job.setReducerClass(FlowCountReducer.class);

// 3 指定 mapper 输出数据的 kv 类型
job.setMapOutputKeyClass(Text.class);
job.setMapOutputValueClass(FlowBean.class);

// 4 指定最终输出的数据的 kv 类型
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(FlowBean.class);

// 8 指定自定义数据分区
job.setPartitionerClass(ProvincePartitioner.class);
// 9 同时指定相应数量的 reduce task
job.setNumReduceTasks(5);

// 5 指定 job 的输入原始文件所在目录
FileInputFormat.setInputPaths(job, new Path(args[0]));
FileOutputFormat.setOutputPath(job, new Path(args[1]));

// 7 将 job 中配置的相关参数, 以及 job 所用的 java 类所在的 jar 包, 提交给
yarn 去运行
boolean result = job.waitForCompletion(true);
System.exit(result ? 0 : 1);
}
}
```

### 7.2.3 需求 3: 将统计结果按照总流量倒序排序 (全排序)

#### 0) 需求

根据需求 1 产生的结果再次对总流量进行排序。

#### 1) 数据准备



phone\_data.txt

#### 2) 分析

- (1) 把程序分两步走, 第一步正常统计总流量, 第二步再把结果进行排序
- (2) context.write(总流量, 手机号)
- (3) FlowBean 实现 WritableComparable 接口重写 compareTo 方法

```
@Override
public int compareTo(FlowBean o) {
    // 倒序排列, 从大到小
```

```
return this.sumFlow > o.getSumFlow() ? -1 : 1;  
}
```

### 3) 代码实现

(1) FlowBean 对象在需求 1 基础上增加了比较功能

```
package com.atguigu.mapreduce.sort;  
  
import java.io.DataInput;  
import java.io.DataOutput;  
import java.io.IOException;  
import org.apache.hadoop.io.WritableComparable;  
  
public class FlowBean implements WritableComparable<FlowBean> {  
  
    private long upFlow;  
    private long downFlow;  
    private long sumFlow;  
  
    // 反序列化时，需要反射调用空参构造函数，所以必须有  
    public FlowBean() {  
        super();  
    }  
  
    public FlowBean(long upFlow, long downFlow) {  
        super();  
        this.upFlow = upFlow;  
        this.downFlow = downFlow;  
        this.sumFlow = upFlow + downFlow;  
    }  
  
    public void set(long upFlow, long downFlow) {  
        this.upFlow = upFlow;  
        this.downFlow = downFlow;  
        this.sumFlow = upFlow + downFlow;  
    }  
  
    public long getSumFlow() {  
        return sumFlow;  
    }  
  
    public void setSumFlow(long sumFlow) {  
        this.sumFlow = sumFlow;  
    }  
}
```

```
public long getUpFlow() {
    return upFlow;
}

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

public long getDownFlow() {
    return downFlow;
}

public void setDownFlow(long downFlow) {
    this.downFlow = downFlow;
}

/**
 * 序列化方法
 * @param out
 * @throws IOException
 */
@Override
public void write(DataOutput out) throws IOException {
    out.writeLong(upFlow);
    out.writeLong(downFlow);
    out.writeLong(sumFlow);
}

/**
 * 反序列化方法 注意反序列化的顺序和序列化的顺序完全一致
 * @param in
 * @throws IOException
 */
@Override
public void readFields(DataInput in) throws IOException {
    upFlow = in.readLong();
    downFlow = in.readLong();
    sumFlow = in.readLong();
}

@Override
public String toString() {
    return upFlow + "\t" + downFlow + "\t" + sumFlow;
}
```



```
}

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

### (2) 编写 mapper

```
package com.atguigu.mapreduce.sort;
import java.io.IOException;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

public class FlowCountSortMapper extends Mapper<LongWritable, Text, FlowBean, Text>{
    FlowBean bean = new FlowBean();
    Text v = new Text();

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

        // 1 获取一行
        String line = value.toString();

        // 2 截取
        String[] fields = line.split("\t");

        // 3 封装对象
        String phoneNbr = fields[0];
        long upFlow = Long.parseLong(fields[1]);
        long downFlow = Long.parseLong(fields[2]);

        bean.set(upFlow, downFlow);
        v.set(phoneNbr);

        // 4 输出
        context.write(bean, v);
    }
}
```

### (3) 编写 reducer

```
package com.atguigu.mapreduce.sort;
import java.io.IOException;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

public class FlowCountSortReducer extends Reducer<FlowBean, Text, Text, FlowBean>{

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

        // 循环输出，避免总流量相同情况
        for (Text text : values) {
            context.write(text, key);
        }
    }
}
```

#### (4) 编写 driver

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

public class FlowCountSortDriver {

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

        // 1 获取配置信息，或者 job 对象实例
        Configuration configuration = new Configuration();
        Job job = Job.getInstance(configuration);

        // 6 指定本程序的 jar 包所在的本地路径
        job.setJarByClass(FlowCountSortDriver.class);

        // 2 指定本业务 job 要使用的 mapper/Reducer 业务类
        job.setMapperClass(FlowCountSortMapper.class);
        job.setReducerClass(FlowCountSortReducer.class);
    }
}
```

```
// 3 指定 mapper 输出数据的 kv 类型
job.setMapOutputKeyClass(FlowBean.class);
job.setMapOutputValueClass(Text.class);

// 4 指定最终输出的数据的 kv 类型
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(FlowBean.class);

// 5 指定 job 的输入原始文件所在目录
FileInputFormat.setInputPaths(job, new Path(args[0]));
FileOutputFormat.setOutputPath(job, new Path(args[1]));

// 7 将 job 中配置的相关参数, 以及 job 所用的 java 类所在的 jar 包, 提交给
yarn 去运行
boolean result = job.waitForCompletion(true);
System.exit(result ? 0 : 1);
}
}
```

## 7.2.4 需求 4: 不同省份输出文件内部排序 (部分排序)

### 1) 需求

要求每个省份手机号输出的文件中按照总流量内部排序。

### 2) 分析:

基于需求 3, 增加自定义分区类即可。

### 3) 案例实操

#### (1) 增加自定义分区类

```
package com.atguigu.mapreduce.sort;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Partitioner;

public class ProvincePartitioner extends Partitioner<FlowBean, Text> {

    @Override
    public int getPartition(FlowBean key, Text value, int numPartitions) {

        // 1 获取手机号码前三位
        String preNum = value.toString().substring(0, 3);

        int partition = 4;
```

```
// 2 根据手机号归属地设置分区
if ("136".equals(preNum)) {
    partition = 0;
}else if ("137".equals(preNum)) {
    partition = 1;
}else if ("138".equals(preNum)) {
    partition = 2;
}else if ("139".equals(preNum)) {
    partition = 3;
}

return partition;
}
```

(2) 在驱动类中添加分区类

```
// 加载自定义分区类
job.setPartitionerClass(FlowSortPartitioner.class);

// 设置 Reducetask 个数
job.setNumReduceTasks(5);
```

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

1) 需求

有如下订单数据

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

现在要求出每一个订单中最贵的商品。

2) 输入数据



GroupingComparator.txt

输出数据预期:

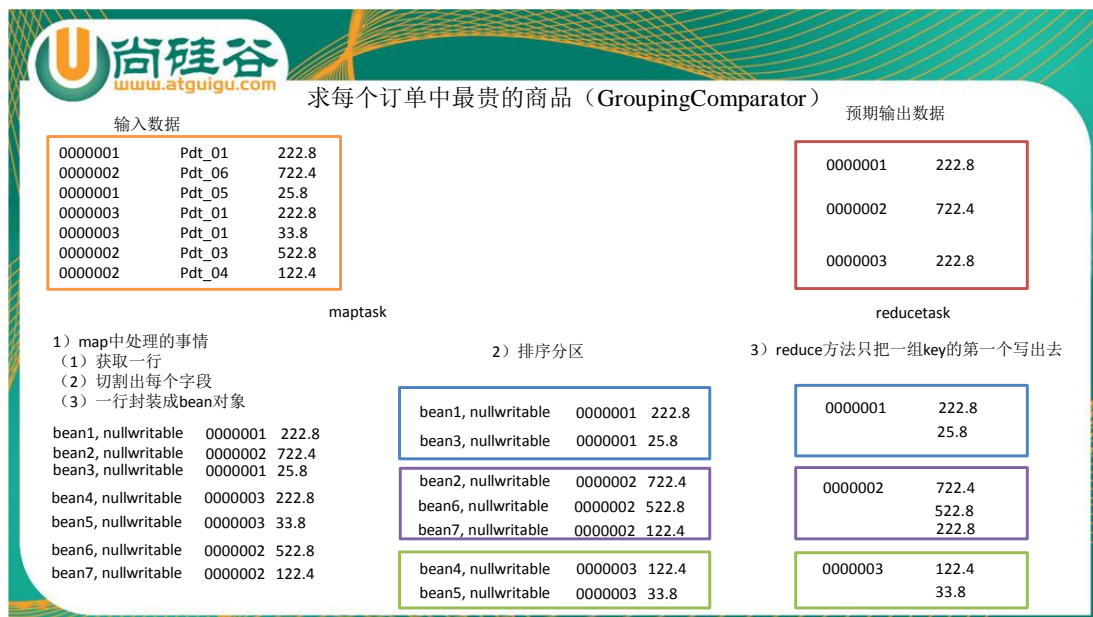
【更多 Java、HTML5、Android、python、大数据 资料下载，可访问尚硅谷（中国）官网 [www.atguigu.com](http://www.atguigu.com) 下载区】

 part-r-00000.txt       part-r-00001       part-r-00002

### 3) 分析

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

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



### 4) 代码实现

#### (1) 定义订单信息 OrderBean

```
package com.atguigu.mapreduce.order;
import java.io.DataInput;
import java.io.DataOutput;
import java.io.IOException;
import org.apache.hadoop.io.WritableComparable;

public class OrderBean implements WritableComparable<OrderBean> {

    private int order_id; // 订单 id 号
    private double price; // 价格

    public OrderBean() {
        super();
    }
}
```

```
public OrderBean(int order_id, double price) {
    super();
    this.order_id = order_id;
    this.price = price;
}

@Override
public void write(DataOutput out) throws IOException {
    out.writeInt(order_id);
    out.writeDouble(price);
}

@Override
public void readFields(DataInput in) throws IOException {
    order_id = in.readInt();
    price = in.readDouble();
}

@Override
public String toString() {
    return order_id + "\t" + price;
}

public int getOrder_id() {
    return order_id;
}

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

public double getPrice() {
    return price;
}

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

// 二次排序
@Override
public int compareTo(OrderBean o) {
```



```
int result;

if (order_id > o.getOrder_id()) {
    result = 1;
} else if (order_id < o.getOrder_id()) {
    result = -1;
} else {
    // 价格倒序排序
    result = price > o.getPrice() ? -1 : 1;
}

return result;
}
}
```

## (2) 编写 OrderSortMapper

```
package com.atguigu.mapreduce.order;
import java.io.IOException;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.NullWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

public class OrderMapper extends Mapper<LongWritable, Text, OrderBean, NullWritable> {
    OrderBean k = new OrderBean();

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

        // 1 获取一行
        String line = value.toString();

        // 2 截取
        String[] fields = line.split("\t");

        // 3 封装对象
        k.setOrder_id(Integer.parseInt(fields[0]));
        k.setPrice(Double.parseDouble(fields[2]));

        // 4 写出
        context.write(k, NullWritable.get());
    }
}
```

## (3) 编写 OrderSortPartitioner

```
package com.atguigu.mapreduce.order;
import org.apache.hadoop.io.NullWritable;
import org.apache.hadoop.mapreduce.Partitioner;

public class OrderPartitioner extends Partitioner<OrderBean, NullWritable> {

    @Override
    public int getPartition(OrderBean key, NullWritable value, int numReduceTasks) {

        return (key.getOrder_id() & Integer.MAX_VALUE) % numReduceTasks;
    }
}
```

## (4) 编写 OrderSortGroupingComparator

```
package com.atguigu.mapreduce.order;
import org.apache.hadoop.io.WritableComparable;
import org.apache.hadoop.io.WritableComparator;

public class OrderGroupingComparator extends WritableComparator {

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

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

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

        int result;
        if (aBean.getOrder_id() > bBean.getOrder_id()) {
            result = 1;
        } else if (aBean.getOrder_id() < bBean.getOrder_id()) {
            result = -1;
        } else {
            result = 0;
        }

        return result;
    }
}
```

```
}
```

#### (5) 编写 OrderSortReducer

```
package com.atguigu.mapreduce.order;
import java.io.IOException;
import org.apache.hadoop.io.NullWritable;
import org.apache.hadoop.mapreduce.Reducer;

public class OrderReducer extends Reducer<OrderBean, NullWritable, OrderBean, NullWritable> {

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

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

#### (6) 编写 OrderSortDriver

```
package com.atguigu.mapreduce.order;
import java.io.IOException;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.NullWritable;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class OrderDriver {

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

        // 1 获取配置信息
        Configuration conf = new Configuration();
        Job job = Job.getInstance(conf);

        // 2 设置 jar 包加载路径
        job.setJarByClass(OrderDriver.class);

        // 3 加载 map/reduce 类
        job.setMapperClass(OrderMapper.class);
        job.setReducerClass(OrderReducer.class);
    }
}
```

```
// 4 设置 map 输出数据 key 和 value 类型
job.setMapOutputKeyClass(OrderBean.class);
job.setMapOutputValueClass(NullWritable.class);

// 5 设置最终输出数据的 key 和 value 类型
job.setOutputKeyClass(OrderBean.class);
job.setOutputValueClass(NullWritable.class);

// 6 设置输入数据和输出数据路径
FileInputFormat.setInputPaths(job, new Path(args[0]));
FileOutputFormat.setOutputPath(job, new Path(args[1]));

// 10 设置 reduce 端的分组
job.setGroupingComparatorClass(OrderGroupingComparator.class);

// 7 设置分区
job.setPartitionerClass(OrderPartitioner.class);

// 8 设置 reduce 个数
job.setNumReduceTasks(3);

// 9 提交
boolean result = job.waitForCompletion(true);
System.exit(result ? 0 : 1);
}
}
```

## 7.4 小文件处理案例（自定义 InputFormat）

### 1) 需求

无论 hdfs 还是 mapreduce，对于小文件都有损效率，实践中，又难免面临处理大量小文件的场景，此时，就需要有相应解决方案。将多个小文件合并成一个文件 SequenceFile，SequenceFile 里面存储着多个文件，存储的形式为文件路径+名称为 key，文件内容为 value。

### 2) 输入数据



one.txt



two.txt



three.txt

最终预期文件格式：

part-r-00000

### 3) 分析

小文件的优化无非以下几种方式:

- (1) 在数据采集的时候, 就将小文件或大批数据合成大文件再上传 HDFS
- (2) 在业务处理之前, 在 HDFS 上使用 `mapreduce` 程序对小文件进行合并
- (3) 在 `mapreduce` 处理时, 可采用 `CombineTextInputFormat` 提高效率

### 4) 具体实现

本节采用自定义 `InputFormat` 的方式, 处理输入小文件的问题。

- (1) 自定义一个类继承 `FileInputFormat`
- (2) 改写 `RecordReader`, 实现一次读取一个完整文件封装为 KV
- (3) 在输出时使用 `SequenceFileOutPutFormat` 输出合并文件

### 5) 程序实现:

- (1) 自定义 `InputFormat`

```
package com.atguigu.mapreduce.inputformat;
import java.io.IOException;
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;

// 定义类继承 FileInputFormat
public class WholeFileInputformat extends FileInputFormat<NullWritable, BytesWritable>{

    @Override
    protected boolean isSplittable(JobContext context, Path filename) {
        return false;
    }

    @Override
    public RecordReader<NullWritable, BytesWritable> createRecordReader(InputSplit split,
TaskAttemptContext context)
        throws IOException, InterruptedException {

        WholeRecordReader recordReader = new WholeRecordReader();
    }
}
```

```
recordReader.initialize(split, context);

return recordReader;
}
}
```

## (2) 自定义 RecordReader

```
package com.atguigu.mapreduce.inputformat;
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 WholeRecordReader extends RecordReader<NullWritable, BytesWritable>{

    private Configuration configuration;
    private FileSplit split;

    private boolean processed = false;
    private BytesWritable value = new BytesWritable();

    @Override
    public void initialize(InputSplit split, TaskAttemptContext context) throws IOException,
    InterruptedException {

        this.split = (FileSplit)split;
        configuration = context.getConfiguration();
    }

    @Override
    public boolean nextKeyValue() throws IOException, InterruptedException {

        if (!processed) {
            // 1 定义缓存区
            byte[] contents = new byte[(int)split.getLength()];
```

```
FileSystem fs = null;
FSDataInputStream fis = null;

try {
    // 2 获取文件系统
    Path path = split.getPath();
    fs = path.getFileSystem(configuration);

    // 3 读取数据
    fis = fs.open(path);

    // 4 读取文件内容
    IOUtils.readFully(fis, contents, 0, contents.length);

    // 5 输出文件内容
    value.set(contents, 0, contents.length);
} catch (Exception e) {

} finally {
    IOUtils.closeStream(fis);
}

processed = true;

return true;
}

return false;
}

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

@Override
public BytesWritable getCurrentValue() throws IOException, InterruptedException {
    return value;
}

@Override
public float getProgress() throws IOException, InterruptedException {
    return processed? 1:0;
```



```
}

@Override
public void close() throws IOException {
}
}
```

### (3) SequenceFileMapper 处理流程

```
package com.atguigu.mapreduce.inputformat;
import java.io.IOException;
import org.apache.hadoop.io.BytesWritable;
import org.apache.hadoop.io.NullWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.lib.input.FileSplit;

public class SequenceFileMapper extends Mapper<NullWritable, BytesWritable, Text,
BytesWritable>{

    Text k = new Text();

    @Override
    protected void setup(Mapper<NullWritable, BytesWritable, Text, BytesWritable>.Context
context)
        throws IOException, InterruptedException {
        // 1 获取文件切片信息
        FileSplit inputSplit = (FileSplit) context.getInputSplit();
        // 2 获取切片名称
        String name = inputSplit.getPath().toString();
        // 3 设置 key 的输出
        k.set(name);
    }

    @Override
    protected void map(NullWritable key, BytesWritable value,
        Context context)
        throws IOException, InterruptedException {

        context.write(k, value);
    }
}
```

### (4) SequenceFileReducer 处理流程

```
package com.atguigu.mapreduce.inputformat;
```

```
import java.io.IOException;
import org.apache.hadoop.io.BytesWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

public class SequenceFileReducer extends Reducer<Text, BytesWritable, Text, BytesWritable> {

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

        context.write(key, values.iterator().next());
    }
}
```

#### (5) SequenceFileDriver 处理流程

```
package com.atguigu.mapreduce.inputformat;
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.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.lib.output.SequenceFileOutputFormat;

public class SequenceFileDriver {

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

        args = new String[] { "e:/input/inputinputformat", "e:/output1" };
        Configuration conf = new Configuration();

        Job job = Job.getInstance(conf);
        job.setJarByClass(SequenceFileDriver.class);
        job.setMapperClass(SequenceFileMapper.class);
        job.setReducerClass(SequenceFileReducer.class);

        // 设置输入的 inputFormat
        job.setInputFormatClass(WholeFileInputformat.class);
        // 设置输出的 outputFormat
```

```
job.setOutputFormatClass(SequenceFileOutputFormat.class);

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

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

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

boolean result = job.waitForCompletion(true);

System.exit(result ? 0 : 1);
}
}
```

## 7.5 过滤日志及自定义日志输出路径案例（自定义 OutputFormat）

### 1) 需求

过滤输入的 log 日志中是否包含 atguigu

(1) 包含 atguigu 的网站输出到 e:/atguigu.log

(2) 不包含 atguigu 的网站输出到 e:/other.log

### 2) 输入数据



log.txt

输出预期:



atguigu.log



other.log

### 3) 具体程序:

(1) 自定义一个 outputformat

```
package com.atguigu.mapreduce.outputformat;
import java.io.IOException;
import org.apache.hadoop.io.NullWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.RecordWriter;
import org.apache.hadoop.mapreduce.TaskAttemptContext;
```

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

public class FilterOutputFormat extends FileOutputFormat<Text, NullWritable>{

    @Override
    public RecordWriter<Text, NullWritable> getRecordWriter(TaskAttemptContext job)
        throws IOException, InterruptedException {

        // 创建一个 RecordWriter
        return new FilterRecordWriter(job);
    }
}
```

## (2) 具体的写数据 RecordWriter

```
package com.atguigu.mapreduce.outputformat;
import java.io.IOException;
import org.apache.hadoop.fs.FSDataOutputStream;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.NullWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.RecordWriter;
import org.apache.hadoop.mapreduce.TaskAttemptContext;

public class FilterRecordWriter extends RecordWriter<Text, NullWritable> {
    FSDataOutputStream atguiguOut = null;
    FSDataOutputStream otherOut = null;

    public FilterRecordWriter(TaskAttemptContext job) {
        // 1 获取文件系统
        FileSystem fs;

        try {
            fs = FileSystem.get(job.getConfiguration());

            // 2 创建输出文件路径
            Path atguiguPath = new Path("e:/atguigu.log");
            Path otherPath = new Path("e:/other.log");

            // 3 创建输出流
            atguiguOut = fs.create(atguiguPath);
            otherOut = fs.create(otherPath);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

```
    }  
}  
  
@Override  
public void write(Text key, NullWritable value) throws IOException,  
InterruptedException {  
  
    // 判断是否包含 “atguigu” 输出到不同文件  
    if (key.toString().contains("atguigu")) {  
        atguiguOut.write(key.toString().getBytes());  
    } else {  
        otherOut.write(key.toString().getBytes());  
    }  
}  
  
@Override  
public void close(TaskAttemptContext context) throws IOException,  
InterruptedException {  
    // 关闭资源  
    if (atguiguOut != null) {  
        atguiguOut.close();  
    }  
  
    if (otherOut != null) {  
        otherOut.close();  
    }  
}  
}
```

### (3) 编写 FilterMapper

```
package com.atguigu.mapreduce.outputformat;  
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 FilterMapper extends Mapper<LongWritable, Text, Text, NullWritable>{  
  
    Text k = new Text();  
  
    @Override  
    protected void map(LongWritable key, Text value, Context context)  
        throws IOException, InterruptedException {
```

```
// 1 获取一行
String line = value.toString();

k.set(line);

// 3 写出
context.write(k, NullWritable.get());
}
}
```

#### (4) 编写 FilterReducer

```
package com.atguigu.mapreduce.outputformat;
import java.io.IOException;
import org.apache.hadoop.io.NullWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

public class FilterReducer extends Reducer<Text, NullWritable, Text, NullWritable> {

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

        String k = key.toString();
        k = k + "\r\n";

        context.write(new Text(k), NullWritable.get());
    }
}
```

#### (5) 编写 FilterDriver

```
package com.atguigu.mapreduce.outputformat;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.NullWritable;
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 FilterDriver {
    public static void main(String[] args) throws Exception {

        args = new String[] { "e:/input/inputoutputformat", "e:/output2" };
    }
}
```

```
Configuration conf = new Configuration();

Job job = Job.getInstance(conf);

job.setJarByClass(FilterDriver.class);
job.setMapperClass(FilterMapper.class);
job.setReducerClass(FilterReducer.class);

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

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

// 要将自定义的输出格式组件设置到 job 中
job.setOutputFormatClass(FilterOutputFormat.class);

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

// 虽然我们自定义了 outputformat，但是因为我们的 outputformat 继承自
fileoutputformat
// 而 fileoutputformat 要输出一个_SUCCESS 文件，所以，在这还得指定一个输出目录
FileOutputFormat.setOutputPath(job, new Path(args[1]));

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

## 7.6 MapReduce 中多表合并案例

1) 需求:

订单数据表 t\_order:

id	pid	amount
1001	01	1
1002	02	2
1003	03	3



order.txt



商品信息表 t\_product

pid	pname
01	小米
02	华为
03	格力



pd.txt

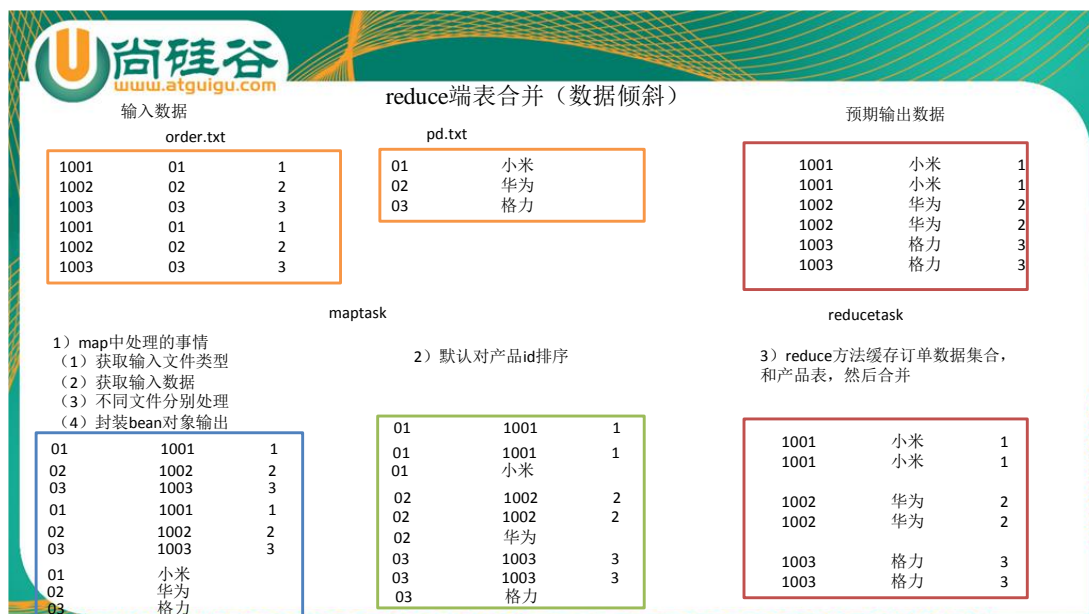
将商品信息表中数据根据商品 pid 合并到订单数据表中。

最终数据形式：

id	pname	amount
1001	小米	1
1004	小米	4
1002	华为	2
1005	华为	5
1003	格力	3
1006	格力	6

## 7.6.1 需求 1: Reduce 端表合并（数据倾斜）

通过将关联条件作为 map 输出的 key, 将两表满足 join 条件的数据并携带数据所来源的文件信息, 发往同一个 reduce task, 在 reduce 中进行数据的串联。



1) 创建商品和订合并后的 bean 类

```
package com.atguigu.mapreduce.table;
import java.io.DataInput;
import java.io.DataOutput;
```

【更多 Java、HTML5、Android、python、大数据 资料下载, 可访问尚硅谷 (中国) 官网 [www.atguigu.com](http://www.atguigu.com) 下载区】

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

public class TableBean implements Writable {
    private String order_id; // 订单 id
    private String p_id; // 产品 id
    private int amount; // 产品数量
    private String pname; // 产品名称
    private String flag; // 表的标记

    public TableBean() {
        super();
    }

    public TableBean(String order_id, String p_id, int amount, String pname, String flag) {
        super();
        this.order_id = order_id;
        this.p_id = p_id;
        this.amount = amount;
        this.pname = pname;
        this.flag = flag;
    }

    public String getFlag() {
        return flag;
    }

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

    public String getOrder_id() {
        return order_id;
    }

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

    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;
}

@Override
public void write(DataOutput out) throws IOException {
    out.writeUTF(order_id);
    out.writeUTF(p_id);
    out.writeInt(amount);
    out.writeUTF(pname);
    out.writeUTF(flag);
}

@Override
public void readFields(DataInput in) throws IOException {
    this.order_id = in.readUTF();
    this.p_id = in.readUTF();
    this.amount = in.readInt();
    this.pname = in.readUTF();
    this.flag = in.readUTF();
}

@Override
public String toString() {
    return order_id + "\t" + pname + "\t" + amount + "\t" ;
}
}
```

## 2) 编写 TableMapper 程序

```
package com.atguigu.mapreduce.table;
import java.io.IOException;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.lib.input.FileSplit;

public class TableMapper extends Mapper<LongWritable, Text, Text, TableBean>{
    TableBean bean = new TableBean();
    Text k = new Text();

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

        // 1 获取输入文件类型
        FileSplit split = (FileSplit) context.getInputSplit();
        String name = split.getPath().getName();

        // 2 获取输入数据
        String line = value.toString();

        // 3 不同文件分别处理
        if (name.startsWith("order")) { // 订单表处理
            // 3.1 切割
            String[] fields = line.split("\t");

            // 3.2 封装 bean 对象
            bean.setOrder_id(fields[0]);
            bean.setP_id(fields[1]);
            bean.setAmount(Integer.parseInt(fields[2]));
            bean.setPname("");
            bean.setFlag("0");

            k.set(fields[1]);
        } else { // 产品表处理
            // 3.3 切割
            String[] fields = line.split("\t");

            // 3.4 封装 bean 对象
            bean.setP_id(fields[0]);
            bean.setPname(fields[1]);
```

```
        bean.setFlag("1");
        bean.setAmount(0);
        bean.setOrder_id("");

        k.set(fields[0]);
    }
    // 4 写出
    context.write(k, bean);
}
}
```

### 3) 编写 TableReducer 程序

```
package com.atguigu.mapreduce.table;
import java.io.IOException;
import java.util.ArrayList;
import org.apache.commons.beanutils.BeanUtils;
import org.apache.hadoop.io.NullWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

public class TableReducer extends Reducer<Text, TableBean, TableBean, NullWritable> {

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

        // 1 准备存储订单的集合
        ArrayList<TableBean> orderBeans = new ArrayList<>();
        // 2 准备 bean 对象
        TableBean pdBean = new TableBean();

        for (TableBean bean : values) {

            if ("0".equals(bean.getFlag())) { // 订单表
                // 拷贝传递过来的每条订单数据到集合中
                TableBean orderBean = new TableBean();
                try {
                    BeanUtils.copyProperties(orderBean, bean);
                } catch (Exception e) {
                    e.printStackTrace();
                }

                orderBeans.add(orderBean);
            } else { // 产品表
```

```
        try {
            // 拷贝传递过来的产品表到内存中
            BeanUtils.copyProperties(pdBean, bean);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

// 3 表的拼接
for(TableBean bean:orderBeans){
    bean.setPname (pdBean.getPname());

    // 4 数据写出去
    context.write(bean, NullWritable.get());
}
}
```

#### 4) 编写 TableDriver 程序

```
package com.atguigu.mapreduce.table;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.NullWritable;
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 TableDriver {

    public static void main(String[] args) throws Exception {
        // 1 获取配置信息，或者 job 对象实例
        Configuration configuration = new Configuration();
        Job job = Job.getInstance(configuration);

        // 2 指定本程序的 jar 包所在的本地路径
        job.setJarByClass(TableDriver.class);

        // 3 指定本业务 job 要使用的 mapper/Reducer 业务类
        job.setMapperClass(TableMapper.class);
        job.setReducerClass(TableReducer.class);

        // 4 指定 mapper 输出数据的 kv 类型
```

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

// 5 指定最终输出的数据的 kv 类型
job.setOutputKeyClass(TableBean.class);
job.setOutputValueClass(NullWritable.class);

// 6 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);
}
}
```

### 3) 运行程序查看结果

1001	小米	1
1001	小米	1
1002	华为	2
1002	华为	2
1003	格力	3
1003	格力	3

缺点: 这种方式中, 合并的操作是在 reduce 阶段完成, reduce 端的处理压力太大, map 节点的运算负载则很低, 资源利用率不高, 且在 reduce 阶段极易产生数据倾斜

解决方案: map 端实现数据合并

## 7.6.2 需求 2: Map 端表合并 (Distributedcache)

### 1) 分析

适用于关联表中有小表的情形;

可以将小表分发到所有的 map 节点, 这样, map 节点就可以在本地对自己所读到的大表数据进行合并并输出最终结果, 可以大大提高合并操作的并发度, 加快处理速度。



 尚硅谷  
www.atguigu.com

map端表合并 (Distributedcache)

1) DistributedCacheDriver 缓存文件	2) 读取缓存的文件数据														
<pre>// 1 加载缓存数据 job.addCacheFile(new URI("file:///e:/cache/pd.txt"));  //2 map端join的逻辑不需要reduce阶段, 设置reducesetask数量为0 job.setNumReduceTasks(0);</pre>	<table><tr><th>setup()方法中</th><th>map方法中</th></tr><tr><td>// 1 获取缓存的文件</td><td>// 1 获取一行</td></tr><tr><td>// 2 循环读取缓存文件一行</td><td>// 2 截取</td></tr><tr><td>// 3 切割</td><td>// 3 获取订单id</td></tr><tr><td>// 4 缓存数据到集合</td><td>// 4 获取商品名称</td></tr><tr><td>// 5 关流</td><td>// 5 拼接</td></tr><tr><td></td><td>// 6 写出</td></tr></table>	setup()方法中	map方法中	// 1 获取缓存的文件	// 1 获取一行	// 2 循环读取缓存文件一行	// 2 截取	// 3 切割	// 3 获取订单id	// 4 缓存数据到集合	// 4 获取商品名称	// 5 关流	// 5 拼接		// 6 写出
setup()方法中	map方法中														
// 1 获取缓存的文件	// 1 获取一行														
// 2 循环读取缓存文件一行	// 2 截取														
// 3 切割	// 3 获取订单id														
// 4 缓存数据到集合	// 4 获取商品名称														
// 5 关流	// 5 拼接														
	// 6 写出														

## 2) 实操案例

### (1) 先在驱动模块中添加缓存文件

```
package test;
import java.net.URI;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.NullWritable;
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 DistributedCacheDriver {

    public static void main(String[] args) throws Exception {
        // 1 获取 job 信息
        Configuration configuration = new Configuration();
        Job job = Job.getInstance(configuration);

        // 2 设置加载 jar 包路径
        job.setJarByClass(DistributedCacheDriver.class);

        // 3 关联 map
        job.setMapperClass(DistributedCacheMapper.class);

        // 4 设置最终输出数据类型
        job.setOutputKeyClass(Text.class);
```

```
job.setOutputValueClass(NullWritable.class);

// 5 设置输入输出路径
FileInputFormat.setInputPaths(job, new Path(args[0]));
FileOutputFormat.setOutputPath(job, new Path(args[1]));

// 6 加载缓存数据
job.addCacheFile(new URI("file:///e:/inputcache/pd.txt"));

// 7 map 端 join 的逻辑不需要 reduce 阶段，设置 reducetask 数量为 0
job.setNumReduceTasks(0);

// 8 提交
boolean result = job.waitForCompletion(true);
System.exit(result ? 0 : 1);
}
}
```

## (2) 读取缓存的文件数据

```
package test;
import java.io.BufferedReader;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.HashMap;
import java.util.Map;
import org.apache.commons.lang.StringUtils;
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 DistributedCacheMapper extends Mapper<LongWritable, Text, Text, NullWritable>{

    Map<String, String> pdMap = new HashMap<>();

    @Override
    protected void setup(Mapper<LongWritable, Text, Text, NullWritable>.Context context)
        throws IOException, InterruptedException {

        // 1 获取缓存的文件
        BufferedReader reader = new BufferedReader(new InputStreamReader(new
        FileInputStream("pd.txt"), "UTF-8"));
```

```
String line;
while(StringUtils.isNotEmpty(line = reader.readLine())){
    // 2 切割
    String[] fields = line.split("\t");

    // 3 缓存数据到集合
    pdMap.put(fields[0], fields[1]);
}

// 4 关流
reader.close();
}

Text k = new Text();

@Override
protected void map(LongWritable key, Text value, Context context)
    throws IOException, InterruptedException {
    // 1 获取一行
    String line = value.toString();

    // 2 截取
    String[] fields = line.split("\t");

    // 3 获取产品 id
    String pId = fields[1];

    // 4 获取商品名称
    String pdName = pdMap.get(pId);

    // 5 拼接
    k.set(line + "\t" + pdName);

    // 6 写出
    context.write(k, NullWritable.get());
}
}
```

## 7.7 日志清洗案例

### 7.7.1 简单解析版

1) 需求:

【更多 Java、HTML5、Android、python、大数据 资料下载，可访问尚硅谷（中国）官网 [www.atguigu.com](http://www.atguigu.com) 下载区】

去除日志中字段长度小于等于 11 的日志。

## 2) 输入数据



web.log

## 3) 实现代码:

### (1) 编写 LogMapper

```
package com.atguigu.mapreduce.weblog;
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 LogMapper extends Mapper<LongWritable, Text, Text, NullWritable>{

    Text k = new Text();

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

        // 1 获取 1 行数据
        String line = value.toString();

        // 2 解析日志
        boolean result = parseLog(line,context);

        // 3 日志不合法退出
        if (!result) {
            return;
        }

        // 4 设置 key
        k.set(line);

        // 5 写出数据
        context.write(k, NullWritable.get());
    }

    // 2 解析日志
```

```
private boolean parseLog(String line, Context context) {  
    // 1 截取  
    String[] fields = line.split(" ");  
  
    // 2 日志长度大于 11 的为合法  
    if (fields.length > 11) {  
        // 系统计数器  
        context.getCounter("map", "true").increment(1);  
        return true;  
    } else {  
        context.getCounter("map", "false").increment(1);  
        return false;  
    }  
}  
}
```

## (2) 编写 LogDriver

```
package com.atguigu.mapreduce.weblog;  
import org.apache.hadoop.conf.Configuration;  
import org.apache.hadoop.fs.Path;  
import org.apache.hadoop.io.NullWritable;  
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 LogDriver {  
  
    public static void main(String[] args) throws Exception {  
  
        args = new String[] { "e:/input/inputlog", "e:/output1" };  
  
        // 1 获取 job 信息  
        Configuration conf = new Configuration();  
        Job job = Job.getInstance(conf);  
  
        // 2 加载 jar 包  
        job.setJarByClass(LogDriver.class);  
  
        // 3 关联 map  
        job.setMapperClass(LogMapper.class);  
  
        // 4 设置最终输出类型  
        job.setOutputKeyClass(Text.class);  
    }  
}
```

```
job.setOutputValueClass(NullWritable.class);

// 设置 reducetask 个数为 0
job.setNumReduceTasks(0);

// 5 设置输入和输出路径
FileInputFormat.setInputPaths(job, new Path(args[0]));
FileOutputFormat.setOutputPath(job, new Path(args[1]));

// 6 提交
job.waitForCompletion(true);
}
}
```

### 7.7.2 复杂解析版

#### 1) 需求:

对 web 访问日志中的各字段识别切分

去除日志中不合法的记录

根据统计需求, 生成各类访问请求过滤数据

#### 2) 输入数据



web.log

#### 3) 实现代码:

(1) 定义一个 bean, 用来记录日志数据中的各数据字段

```
package com.atguigu.mapreduce.log;

public class LogBean {
    private String remote_addr; // 记录客户端的 ip 地址
    private String remote_user; // 记录客户端用户名称, 忽略属性 "-"
    private String time_local; // 记录访问时间与时区
    private String request; // 记录请求的 url 与 http 协议
    private String status; // 记录请求状态; 成功是 200
    private String body_bytes_sent; // 记录发送给客户端文件主体内容大小
    private String http_referer; // 用来记录从那个页面链接访问过来的
    private String http_user_agent; // 记录客户浏览器的相关信息

    private boolean valid = true; // 判断数据是否合法

    public String getRemote_addr() {
```

```
        return remote_addr;
    }

    public void setRemote_addr(String remote_addr) {
        this.remote_addr = remote_addr;
    }

    public String getRemote_user() {
        return remote_user;
    }

    public void setRemote_user(String remote_user) {
        this.remote_user = remote_user;
    }

    public String getTime_local() {
        return time_local;
    }

    public void setTime_local(String time_local) {
        this.time_local = time_local;
    }

    public String getRequest() {
        return request;
    }

    public void setRequest(String request) {
        this.request = request;
    }

    public String getStatus() {
        return status;
    }

    public void setStatus(String status) {
        this.status = status;
    }

    public String getBody_bytes_sent() {
        return body_bytes_sent;
    }
}
```



```
public void setBody_bytes_sent(String body_bytes_sent) {
    this.body_bytes_sent = body_bytes_sent;
}

public String getHttp_referer() {
    return http_referer;
}

public void setHttp_referer(String http_referer) {
    this.http_referer = http_referer;
}

public String getHttp_user_agent() {
    return http_user_agent;
}

public void setHttp_user_agent(String http_user_agent) {
    this.http_user_agent = http_user_agent;
}

public boolean isValid() {
    return valid;
}

public void setValid(boolean valid) {
    this.valid = valid;
}

@Override
public String toString() {
    StringBuilder sb = new StringBuilder();
    sb.append(this.valid);
    sb.append("\001").append(this.remote_addr);
    sb.append("\001").append(this.remote_user);
    sb.append("\001").append(this.time_local);
    sb.append("\001").append(this.request);
    sb.append("\001").append(this.status);
    sb.append("\001").append(this.body_bytes_sent);
    sb.append("\001").append(this.http_referer);
    sb.append("\001").append(this.http_user_agent);

    return sb.toString();
}
```

```
}
```

## (2) 编写 LogMapper 程序

```
package com.atguigu.mapreduce.log;
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 LogMapper extends Mapper<LongWritable, Text, Text, NullWritable>{
    Text k = new Text();

    @Override
    protected void map(LongWritable key, Text value, Context context)
        throws IOException, InterruptedException {
        // 1 获取 1 行
        String line = value.toString();

        // 2 解析日志是否合法
        LogBean bean = pressLog(line);

        if (!bean.isValid()) {
            return;
        }

        k.set(bean.toString());

        // 3 输出
        context.write(k, NullWritable.get());
    }

    // 解析日志
    private LogBean pressLog(String line) {
        LogBean logBean = new LogBean();

        // 1 截取
        String[] fields = line.split(" ");

        if (fields.length > 11) {
            // 2 封装数据
            logBean.setRemote_addr(fields[0]);
            logBean.setRemote_user(fields[1]);
            logBean.setTime_local(fields[3].substring(1));
        }
    }
}
```

```
        logBean.setRequest(fields[6]);
        logBean.setStatus(fields[8]);
        logBean.setBody_bytes_sent(fields[9]);
        logBean.setHttp_referer(fields[10]);

        if (fields.length > 12) {
            logBean.setHttp_user_agent(fields[11] + " " + fields[12]);
        } else {
            logBean.setHttp_user_agent(fields[11]);
        }

        // 大于 400, HTTP 错误
        if (Integer.parseInt(logBean.getStatus()) >= 400) {
            logBean.setValid(false);
        }
    } else {
        logBean.setValid(false);
    }

    return logBean;
}
}
```

### (3) 编写 LogDriver 程序

```
package com.atguigu.mapreduce.log;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.NullWritable;
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 LogDriver {
    public static void main(String[] args) throws Exception {
        // 1 获取 job 信息
        Configuration conf = new Configuration();
        Job job = Job.getInstance(conf);

        // 2 加载 jar 包
        job.setJarByClass(LogDriver.class);

        // 3 关联 map
        job.setMapperClass(LogMapper.class);
```

```
// 4 设置最终输出类型
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(NullWritable.class);

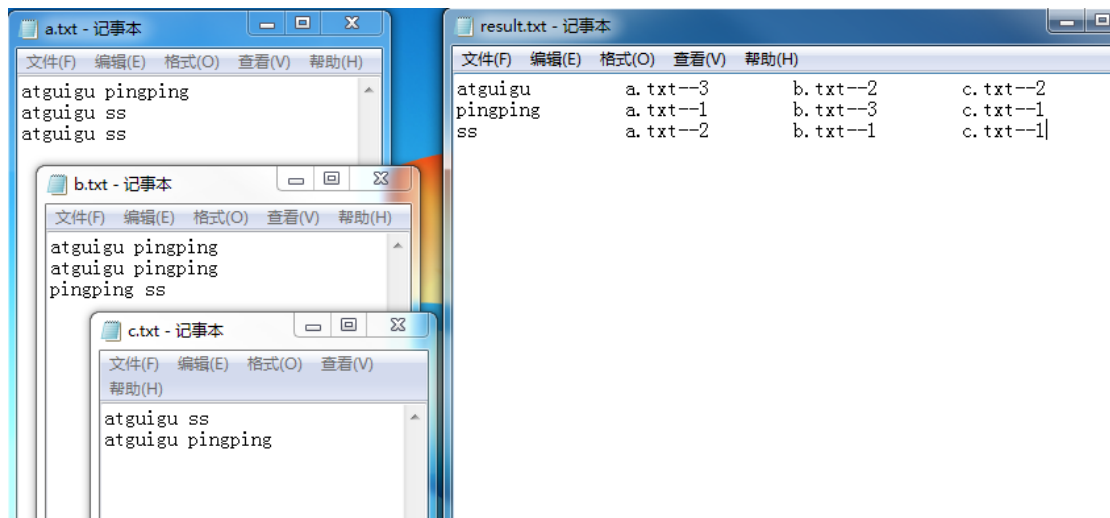
// 5 设置输入和输出路径
FileInputFormat.setInputPaths(job, new Path(args[0]));
FileOutputFormat.setOutputPath(job, new Path(args[1]));

// 6 提交
job.waitForCompletion(true);
}
```

## 7.8 倒排索引案例（多 job 串联）

0) 需求：有大量的文本（文档、网页），需要建立搜索索引

 a.txt     b.txt     c.txt



(1) 第一次预期输出结果

```
atguigu--a.txt 3
atguigu--b.txt 2
atguigu--c.txt 2
pingping--a.txt 1
pingping--b.txt 3
pingping--c.txt 1
ss--a.txt 2
ss--b.txt 1
ss--c.txt 1
```

(2) 第二次预期输出结果

```
atguigu c.txt-->2 b.txt-->2 a.txt-->3
pingping c.txt-->1 b.txt-->3 a.txt-->1
ss c.txt-->1 b.txt-->1 a.txt-->2
```

## 1) 第一次处理

(1) 第一次处理, 编写 OneIndexMapper

```
package com.atguigu.mapreduce.index;
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;
import org.apache.hadoop.mapreduce.lib.input.FileSplit;

public class OneIndexMapper extends Mapper<LongWritable, Text, Text , IntWritable>{

    String name;
    Text k = new Text();
    IntWritable v = new IntWritable();

    @Override
    protected void setup(Context context)
        throws IOException, InterruptedException {
        // 获取文件名称
        FileSplit split = (FileSplit) context.getInputSplit();

        name = split.getPath().getName();
    }

    @Override
    protected void map(LongWritable key, Text value, Context context)
        throws IOException, InterruptedException {
        // 1 获取 1 行
        String line = value.toString();

        // 2 切割
        String[] fields = line.split(" ");

        for (String word : fields) {
            // 3 拼接
            k.set(word+"--"+name);
            v.set(1);
        }
    }
}
```

```
        // 4 写出
        context.write(k, v);
    }
}
```

(2) 第一次处理，编写 OneIndexReducer

```
package com.atguigu.mapreduce.index;
import java.io.IOException;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

public class OneIndexReducer extends Reducer<Text, IntWritable, Text, IntWritable>{

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

        int count = 0;
        // 1 累加求和
        for(IntWritable value: values){
            count +=value.get();
        }

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

(3) 第一次处理，编写 OneIndexDriver

```
package com.atguigu.mapreduce.index;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class OneIndexDriver {

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

```
args = new String[] { "e:/input/inputoneindex", "e:/output5" };

Configuration conf = new Configuration();

Job job = Job.getInstance(conf);
job.setJarByClass(OneIndexDriver.class);

job.setMapperClass(OneIndexMapper.class);
job.setReducerClass(OneIndexReducer.class);

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

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

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

job.waitForCompletion(true);
}
```

(4) 查看第一次输出结果

```
atguigu--a.txt 3
atguigu--b.txt 2
atguigu--c.txt 2
pingping--a.txt 1
pingping--b.txt 3
pingping--c.txt 1
ss--a.txt 2
ss--b.txt 1
ss--c.txt 1
```

2) 第二次处理

(1) 第二次处理，编写 TwoIndexMapper

```
package com.atguigu.mapreduce.index;
import java.io.IOException;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

public class TwoIndexMapper extends Mapper<LongWritable, Text, Text, Text>{
    Text k = new Text();
```

```
Text v = new Text();

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

    // 1 获取 1 行数据
    String line = value.toString();

    // 2 用"--"切割
    String[] fields = line.split("--");

    k.set(fields[0]);
    v.set(fields[1]);

    // 3 输出数据
    context.write(k, v);
}
}
```

#### (2) 第二次处理，编写 TwoIndexReducer

```
package com.atguigu.mapreduce.index;
import java.io.IOException;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;
public class TwoIndexReducer extends Reducer<Text, Text, Text, Text> {

    @Override
    protected void reduce(Text key, Iterable<Text> values, Context context) throws
    IOException, InterruptedException {
        // atguigu a.txt 3
        // atguigu b.txt 2
        // atguigu c.txt 2

        // atguigu c.txt-->2 b.txt-->2 a.txt-->3

        StringBuilder sb = new StringBuilder();
        // 1 拼接
        for (Text value : values) {
            sb.append(value.toString().replace("\t", "-->") + "\t");
        }
        // 2 写出
        context.write(key, new Text(sb.toString()));
    }
}
```



```
}
```

(3) 第二次处理, 编写 TwoIndexDriver

```
package com.atguigu.mapreduce.index;
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 TwoIndexDriver {

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

        args = new String[] { "e:/input/inputtwoindex", "e:/output6" };

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

        job.setJarByClass(TwoIndexDriver.class);
        job.setMapperClass(TwoIndexMapper.class);
        job.setReducerClass(TwoIndexReducer.class);

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

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

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

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

    }

}
```

(4) 第二次查看最终结果

```
atguigu c.txt-->2 b.txt-->2 a.txt-->3
pingping c.txt-->1 b.txt-->3 a.txt-->1
ss c.txt-->1 b.txt-->1 a.txt-->2
```

## 7.9 找博客共同好友案例

1) 需求:

以下是博客的好友列表数据,冒号前是一个用户,冒号后是该用户的所有好友(数据中的好友关系是单向的)



friends.txt

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?

2) 需求分析:

先求出 A、B、C、....等是谁的好友

第一次输出结果

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

第二次输出结果

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

B-C A  
B-D A E  
B-E C  
B-F E A C  
B-G C E A  
B-H A E C  
B-I A  
B-K C A  
B-L E  
B-ME  
B-O A  
C-D A F  
C-E D  
C-F D A  
C-G D F A  
C-H D A  
C-I A  
C-K A D  
C-L D F  
C-MF  
C-O I A  
D-E L  
D-F A E  
D-G E A F  
D-H A E  
D-I A  
D-K A  
D-L E F  
D-M F E  
D-O A  
E-F D M C B  
E-G C D  
E-H C D  
E-J B  
E-K C D  
E-L D  
F-G D C A E  
F-H A D O E C  
F-I O A  
F-J B O  
F-K D C A  
F-L E D  
F-ME

```
F-O A
G-H D C E A
G-I A
G-K D A C
G-L D F E
G-M E F
G-O A
H-I O A
H-J O
H-K A C D
H-L D E
H-M E
H-O A
I-J O
I-K A
I-O A
K-L D
K-O A
L-M E F
```

### 3) 代码实现:

#### (1) 第一次 Mapper

```
package com.atguigu.mapreduce.friends;
import java.io.IOException;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

public class OneShareFriendsMapper extends Mapper<LongWritable, Text, Text, Text>{

    @Override
    protected void map(LongWritable key, Text value, Mapper<LongWritable, Text, Text,
Text>.Context context)
        throws IOException, InterruptedException {
        // 1 获取一行 A:B,C,D,F,E,O
        String line = value.toString();

        // 2 切割
        String[] fields = line.split(":");

        // 3 获取 person 和好友
        String person = fields[0];
        String[] friends = fields[1].split(",");
```

```
// 4 写出去
for(String friend: friends){
    // 输出 <好友, 人>
    context.write(new Text(friend), new Text(person));
}
}
```

### (2) 第一次 Reducer

```
package com.atguigu.mapreduce.friends;
import java.io.IOException;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

public class OneShareFriendsReducer extends Reducer<Text, Text, Text, Text>{

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

        StringBuffer sb = new StringBuffer();
        //1 拼接
        for(Text person: values){
            sb.append(person).append(",");
        }

        //2 写出
        context.write(key, new Text(sb.toString()));
    }
}
```

### (3) 第一次 Driver

```
package com.atguigu.mapreduce.friends;
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 OneShareFriendsDriver {

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

```
// 1 获取 job 对象
Configuration configuration = new Configuration();
Job job = Job.getInstance(configuration);

// 2 指定 jar 包运行的路径
job.setJarByClass(OneShareFriendsDriver.class);

// 3 指定 map/reduce 使用的类
job.setMapperClass(OneShareFriendsMapper.class);
job.setReducerClass(OneShareFriendsReducer.class);

// 4 指定 map 输出的数据类型
job.setMapOutputKeyClass(Text.class);
job.setMapOutputValueClass(Text.class);

// 5 指定最终输出的数据类型
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(Text.class);

// 6 指定 job 的输入原始所在目录
FileInputFormat.setInputPaths(job, new Path(args[0]));
FileOutputFormat.setOutputPath(job, new Path(args[1]));

// 7 提交
boolean result = job.waitForCompletion(true);

System.exit(result?0:1);
}
}
```

#### (4) 第二次 Mapper

```
package com.atguigu.mapreduce.friends;
import java.io.IOException;
import java.util.Arrays;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

public class TwoShareFriendsMapper extends Mapper<LongWritable, Text, Text, Text>{

    @Override
    protected void map(LongWritable key, Text value, Context context)
        throws IOException, InterruptedException {
        // A I, K, C, B, G, F, H, O, D,
```

```
// 友 人, 人, 人
String line = value.toString();
String[] friend_persons = line.split("\\t");

String friend = friend_persons[0];
String[] persons = friend_persons[1].split(",");

Arrays.sort(persons);

for (int i = 0; i < persons.length - 1; i++) {

    for (int j = i + 1; j < persons.length; j++) {
        // 发出 <人-人, 好友> , 这样, 相同的“人-人”对的所有好友就会到
同 1 个 reduce 中去
        context.write(new Text(persons[i] + "-" + persons[j]), new Text(friend));
    }
}
}
```

#### (5) 第二次 Reducer

```
package com.atguigu.mapreduce.friends;
import java.io.IOException;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

public class TwoShareFriendsReducer extends Reducer<Text, Text, Text, Text>{

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

        StringBuffer sb = new StringBuffer();

        for (Text friend : values) {
            sb.append(friend).append(" ");
        }

        context.write(key, new Text(sb.toString()));
    }
}
```

#### (6) 第二次 Driver

```
package com.atguigu.mapreduce.friends;
```

```
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 TwoShareFriendsDriver {

    public static void main(String[] args) throws Exception {
        // 1 获取 job 对象
        Configuration configuration = new Configuration();
        Job job = Job.getInstance(configuration);

        // 2 指定 jar 包运行的路径
        job.setJarByClass(TwoShareFriendsDriver.class);

        // 3 指定 map/reduce 使用的类
        job.setMapperClass(TwoShareFriendsMapper.class);
        job.setReducerClass(TwoShareFriendsReducer.class);

        // 4 指定 map 输出的数据类型
        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(Text.class);

        // 5 指定最终输出的数据类型
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(Text.class);

        // 6 指定 job 的输入原始所在目录
        FileInputFormat.setInputPaths(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));

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

## 7.10 压缩/解压缩案例

### 7.10.1 数据流的压缩和解压缩

CompressionCodec 有两个方法可以用于轻松地压缩或解压缩数据。要想对正在被写入

【更多 Java、HTML5、Android、python、大数据 资料下载，可访问尚硅谷（中国）官网 [www.atguigu.com](http://www.atguigu.com) 下载区】



一个输出流的数据进行**压缩**，我们可以使用 `createOutputStream(OutputStreamout)`方法创建一个 `CompressionOutputStream`，将其以压缩格式写入底层的流。相反，要想对从输入流读取而来的数据进行**解压缩**，则调用 `createInputStream(InputStreamin)` 函数，从而获得一个 `CompressionInputStream`，从而从底层的流读取未压缩的数据。

测试一下如下压缩方式：

DEFLATE	org.apache.hadoop.io.compress.DefaultCodec
gzip	org.apache.hadoop.io.compress.GzipCodec
bzip2	org.apache.hadoop.io.compress.BZip2Codec

```
package com.atguigu.mapreduce.compress;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IOUtils;
import org.apache.hadoop.io.compress.CompressionCodec;
import org.apache.hadoop.io.compress.CompressionCodecFactory;
import org.apache.hadoop.io.compress.CompressionInputStream;
import org.apache.hadoop.io.compress.CompressionOutputStream;
import org.apache.hadoop.util.ReflectionUtils;

public class TestCompress {

    public static void main(String[] args) throws Exception {
        compress("e:/hello.txt", "org.apache.hadoop.io.compress.BZip2Codec");
        // decompress("e:/hello.txt.bz2");
    }

    // 压缩
    private static void compress(String filename, String method) throws Exception {

        // 1 获取输入流
        FileInputStream fis = new FileInputStream(new File(filename));

        Class codecClass = Class.forName(method);

        CompressionCodec codec = (CompressionCodec)
```

```
ReflectionUtils.newInstance(codecClass, new Configuration());

    // 2 获取输出流
    FileOutputStream fos = new FileOutputStream(new File(filename
+codec.getDefaultExtension()));
    CompressionOutputStream cos = codec.createOutputStream(fos);

    // 3 流的对拷
    IOUtils.copyBytes(fis, cos, 1024*1024*5, false);

    // 4 关闭资源
    fis.close();
    cos.close();
    fos.close();
}

// 解压缩
private static void decompress(String filename) throws FileNotFoundException,
IOException {

    // 0 校验是否能解压缩
    CompressionCodecFactory factory = new CompressionCodecFactory(new
Configuration());
    CompressionCodec codec = factory.getCodec(new Path(filename));

    if (codec == null) {
        System.out.println("cannot find codec for file " + filename);
        return;
    }

    // 1 获取输入流
    CompressionInputStream cis = codec.createInputStream(new FileInputStream(new
File(filename)));

    // 2 获取输出流
    FileOutputStream fos = new FileOutputStream(new File(filename + ".decoded"));

    // 3 流的对拷
    IOUtils.copyBytes(cis, fos, 1024*1024*5, false);

    // 4 关闭资源
    cis.close();
    fos.close();
}
```

```
}  
}
```

### 7.10.2 Map 输出端采用压缩

即使你的 MapReduce 的输入输出文件都是未压缩的文件，你仍然可以对 map 任务的中间结果输出做压缩，因为它要写在硬盘并且通过网络传输到 reduce 节点，对其压缩可以提高很多性能，这些工作只要设置两个属性即可，我们来看下代码怎么设置：

1) 给大家提供的 hadoop 源码支持的压缩格式有：**BZip2Codec**、**DefaultCodec**

```
package com.atguigu.mapreduce.compress;  
import java.io.IOException;  
import org.apache.hadoop.conf.Configuration;  
import org.apache.hadoop.fs.Path;  
import org.apache.hadoop.io.IntWritable;  
import org.apache.hadoop.io.Text;  
import org.apache.hadoop.io.compress.BZip2Codec;  
import org.apache.hadoop.io.compress.CompressionCodec;  
import org.apache.hadoop.io.compress.GzipCodec;  
import org.apache.hadoop.mapreduce.Job;  
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;  
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;  
  
public class WordCountDriver {  
  
    public static void main(String[] args) throws IOException, ClassNotFoundException,  
        InterruptedException {  
  
        Configuration configuration = new Configuration();  
  
        // 开启 map 端输出压缩  
        configuration.setBoolean("mapreduce.map.output.compress", true);  
        // 设置 map 端输出压缩方式  
        configuration.setClass("mapreduce.map.output.compress.codec", BZip2Codec.class,  
            CompressionCodec.class);  
  
        Job job = Job.getInstance(configuration);  
  
        job.setJarByClass(WordCountDriver.class);  
  
        job.setMapperClass(WordCountMapper.class);  
        job.setReducerClass(WordCountReducer.class);
```

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

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

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

boolean result = job.waitForCompletion(true);

System.exit(result ? 1 : 0);
}
}
```

## 2) Mapper 保持不变

```
package com.atguigu.mapreduce.compress;
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 context)
        throws IOException, InterruptedException {
        // 1 获取一行
        String line = value.toString();
        // 2 切割
        String[] words = line.split(" ");
        // 3 循环写出
        for(String word:words){
            context.write(new Text(word), new IntWritable(1));
        }
    }
}
```

## 3) Reducer 保持不变

```
package com.atguigu.mapreduce.compress;
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> values,
        Context context) throws IOException, InterruptedException {

        int count = 0;
        // 1 汇总
        for(IntWritable value:values){
            count += value.get();
        }

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

### 7.10.3 Reduce 输出端采用压缩

基于 wordcount 案例处理

#### 1) 修改驱动

```
package com.atguigu.mapreduce.compress;
import java.io.IOException;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.io.compress.BZip2Codec;
import org.apache.hadoop.io.compress.DefaultCodec;
import org.apache.hadoop.io.compress.GzipCodec;
import org.apache.hadoop.io.compress.Lz4Codec;
import org.apache.hadoop.io.compress.SnappyCodec;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class WordCountDriver {

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

```
Configuration configuration = new Configuration();

Job job = Job.getInstance(configuration);

job.setJarByClass(WordCountDriver.class);

job.setMapperClass(WordCountMapper.class);
job.setReducerClass(WordCountReducer.class);

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

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

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

// 设置 reduce 端输出压缩开启
FileOutputFormat.setCompressOutput(job, true);

// 设置压缩的方式
FileOutputFormat.setOutputCompressorClass(job, BZip2Codec.class);
// FileOutputFormat.setOutputCompressorClass(job, GzipCodec.class);
// FileOutputFormat.setOutputCompressorClass(job, DefaultCodec.class);

boolean result = job.waitForCompletion(true);

System.exit(result?1:0);
}
```

2) Mapper 和 Reducer 保持不变（详见 7.10.2）

## 7.11 KeyValueTextInputFormat 使用案例

1) 需求：统计输入文件中每一行的第一个单词相同的行数。

2) 输入文件：

banzhang ni hao

xihuan hadoop banzhang dc

banzhang ni hao

xihuan hadoop banzhang dc

## 3) 输出

banzhang 2

xihuan 2

## 4) 代码实现

## (1) 编写 mapper

```
package com.atguigu.mapreduce.KeyValueTextInputFormat;
import java.io.IOException;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

public class KVTextMapper extends Mapper<Text, Text, Text, LongWritable>{

    final Text k = new Text();
    final LongWritable v = new LongWritable();

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

        // banzhang ni hao
        // 1 设置 key 和 value
        // banzhang
        k.set(key);
        // 设置 key 的个数
        v.set(1);

        // 2 写出
        context.write(k, v);
    }
}
```

## (2) 编写 reducer

```
package com.atguigu.mapreduce.KeyValueTextInputFormat;
import java.io.IOException;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

public class KVTextReducer extends Reducer<Text, LongWritable, Text, LongWritable>{

    LongWritable v = new LongWritable();
```

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

    long count = 0L;
    // 1 汇总统计
    for (LongWritable value : values) {
        count += value.get();
    }

    v.set(count);

    // 2 输出
    context.write(key, v);
}
}
```

### (3) 编写 Driver

```
package com.atguigu.mapreduce.keyvalueTextInputFormat;
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.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.input.KeyValueLineRecordReader;
import org.apache.hadoop.mapreduce.lib.input.KeyValueTextInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class MyDriver {

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

        Configuration conf = new Configuration();
        // 设置切割符
        conf.set(KeyValueLineRecordReader.KEY_VALUE_SEPERATOR, " ");
        // 获取 job 对象
        Job job = Job.getInstance(conf);

        // 设置 jar 包位置, 关联 mapper 和 reducer
        job.setJarByClass(MyDriver.class);
    }
}
```



```
job.setMapperClass(MyMapper.class);
job.setOutputValueClass(LongWritable.class);

// 设置 map 输出 kv 类型
job.setMapOutputKeyClass(Text.class);
job.setMapOutputValueClass(LongWritable.class);

// 设置最终输出 kv 类型
job.setReducerClass(MyReducer.class);
job.setOutputKeyClass(Text.class);

// 设置输入输出数据路径
FileInputFormat.setInputPaths(job, new Path(args[0]));

// 设置输入格式
job.setInputFormatClass(KeyValueTextInputFormat.class);

// 设置输出数据路径
FileOutputFormat.setOutputPath(job, new Path(args[1]));

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

## 7.12 NLineInputFormat 使用案例

- 1) 需求：根据每个输入文件的行数来规定输出多少个切片。例如每三行放入一个切片中。
- 2) 输入数据：

```
banzhang ni hao
xihuan hadoop banzhang dc
banzhang ni hao
xihuan hadoop banzhang dc
banzhang ni hao
xihuan hadoop banzhang dc
banzhang ni hao
xihuan hadoop banzhang dc
banzhang ni hao
xihuan hadoop banzhang dcbanzhang ni hao
xihuan hadoop banzhang dc
```

- 3) 输出结果：

Number of splits:4

【更多 Java、HTML5、Android、python、大数据 资料下载，可访问尚硅谷（中国）官网 [www.atguigu.com](http://www.atguigu.com) 下载区】

## 4) 代码实现:

## (1) 编写 mapper

```
package com.atguigu.mapreduce.nline;
import java.io.IOException;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

public class NLineMapper extends Mapper<LongWritable, Text, Text, LongWritable>{

    private Text k = new Text();
    private LongWritable v = new LongWritable(1);

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

        // 1 获取一行
        final String line = value.toString();

        // 2 切割
        final String[] splited = line.split(" ");

        // 3 循环写出
        for (int i = 0; i < splited.length; i++) {

            k.set(splited[i]);

            context.write(k, v);
        }
    }
}
```

## (2) 编写 Reducer

```
package com.atguigu.mapreduce.nline;
import java.io.IOException;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

public class NLineReducer extends Reducer<Text, LongWritable, Text, LongWritable>{

    LongWritable v = new LongWritable();
```

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

    long count = 0l;
    // 1 汇总
    for (LongWritable value : values) {
        count += value.get();
    }

    v.set(count);

    // 2 输出
    context.write(key, v);
}
}
```

### (3) 编写 driver

```
package com.atguigu.mapreduce.nline;
import java.io.IOException;
import java.net.URISyntaxException;
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.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.input.NLineInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class NLineDriver {

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

        // 获取 job 对象
        Configuration configuration = new Configuration();
        Job job = Job.getInstance(configuration);

        // 设置每个切片 InputSplit 中划分三条记录
        NLineInputFormat.setNumLinesPerSplit(job, 3);

        // 使用 NLineInputFormat 处理记录数
```

```
job.setInputFormatClass(NLineInputFormat.class);

// 设置 jar 包位置, 关联 mapper 和 reducer
job.setJarByClass(NLineDriver.class);
job.setMapperClass(NLineMapper.class);
job.setReducerClass(NLineReducer.class);

// 设置 map 输出 kv 类型
job.setMapOutputKeyClass(Text.class);
job.setMapOutputValueClass(LongWritable.class);

// 设置最终输出 kv 类型
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(LongWritable.class);

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

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

## 5) 结果查看

### (1) 输入数据

```
banzhang ni hao
xihuan hadoop banzhang dc
banzhang ni hao
xihuan hadoop banzhang dc
banzhang ni hao
xihuan hadoop banzhang dc
banzhang ni hao
xihuan hadoop banzhang dc
banzhang ni hao
xihuan hadoop banzhang dcbanzhang ni hao
xihuan hadoop banzhang dc
```

### (2) 输出结果的切片数:

```
WARN [org.apache.hadoop.mapreduce.JobResourceUploader] - Hadoop command-line
WARN [org.apache.hadoop.mapreduce.JobResourceUploader] - No job jar file set
INFO [org.apache.hadoop.mapreduce.lib.input.FileInputFormat] - Total input p
INFO [org.apache.hadoop.mapreduce.JobSubmitter] - number of splits:4
INFO [org.apache.hadoop.mapreduce.JobSubmitter] - Submitting tokens for job:
INFO [org.apache.hadoop.mapreduce.Job] - The url to track the job: http://lo
INFO [org.apache.hadoop.mapreduce.Job] - Running job: job_local998538859_000
INFO [org.apache.hadoop.mapred.LocalJobRunner] - OutputCommitter set in conf
INFO [org.apache.hadoop.mapreduce.lib.output.FileOutputCommitter] - File Out
```

## 八 常见错误

1) 导包容易出错。尤其 Text 和 CombineTextInputFormat。

2) Mapper 中第一个输入的参数必须是 LongWritable 或者 NullWritable, 不可以是 IntWritable。  
报的错误是类型转换异常。

3) java.lang.Exception: java.io.IOException: Illegal partition for 13926435656 (4), 说明 partition 和 reducetask 个数没对上, 调整 reducetask 个数。

4) 如果分区数不是 1, 但是 reducetask 为 1, 是否执行分区过程。答案是: 不执行分区过程。  
因为在 maptask 的源码中, 执行分区的前提是先判断 reduceNum 个数是否大于 1。不大于 1 肯定不执行。

5) 在 Windows 环境编译的 jar 包导入到 linux 环境中运行,

```
hadoop jar wc.jar com.atguigu.mapreduce.wordcount.WordCountDriver /user/atguigu/
/user/atguigu/output
```

报如下错误:

```
Exception in thread "main" java.lang.UnsupportedClassVersionError:
com/atguigu/mapreduce/wordcount/WordCountDriver : Unsupported major.minor version 52.0
```

原因是 Windows 环境用的 jdk1.7, linux 环境用的 jdk1.8。

解决方案: 统一 jdk 版本。

6) 缓存 pd.txt 小文件案例中, 报找不到 pd.txt 文件

原因: 大部分为路径书写错误。还有就是要检查 pd.txt.txt 的问题。还有个别电脑写相对路径找不到 pd.txt, 可以修改为绝对路径。

7) 报类型转换异常。

通常都是在驱动函数中设置 map 输出和最终输出时编写错误。

Map 输出的 key 如果没有排序, 也会报类型转换异常。

8) 集群中运行 wc.jar 时出现了无法获得输入文件。

【更多 Java、HTML5、Android、python、大数据 资料下载, 可访问尚硅谷(中国)官网 [www.atguigu.com](http://www.atguigu.com) 下载区】

原因：wordcount 案例的输入文件不能放用 hdfs 集群的根目录。

9) 出现了如下相关异常

```
Exception in thread "main" java.lang.UnsatisfiedLinkError:
org.apache.hadoop.io.nativeio.NativeIO$Windows.access0(Ljava/lang/String;I)Z
```

```
at org.apache.hadoop.io.nativeio.NativeIO$Windows.access0(Native Method)
```

```
at org.apache.hadoop.io.nativeio.NativeIO$Windows.access(NativeIO.java:609)
```

```
at org.apache.hadoop.fs.FileUtil.canRead(FileUtil.java:977)
```

```
java.io.IOException: Could not locate executable null\bin\winutils.exe in the Hadoop binaries.
```

```
at org.apache.hadoop.util.Shell.getQualifiedBinPath(Shell.java:356)
```

```
at org.apache.hadoop.util.Shell.getWinUtilsPath(Shell.java:371)
```

```
at org.apache.hadoop.util.Shell.<clinit>(Shell.java:364)
```

解决方案：拷贝 hadoop.dll 文件到 windows 目录 C:\Windows\System32。个别同学电脑还需要修改 hadoop 源码。

10) 自定义 outputformat 时，注意在 recordWriter 中的 close 方法必须关闭流资源。否则输出的文件内容中数据为空。

```
@Override
public void close(TaskAttemptContext context) throws IOException,
InterruptedException {
    if (atguigufos != null) {
        atguigufos.close();
    }
    if (otherfos != null) {
        otherfos.close();
    }
}
```