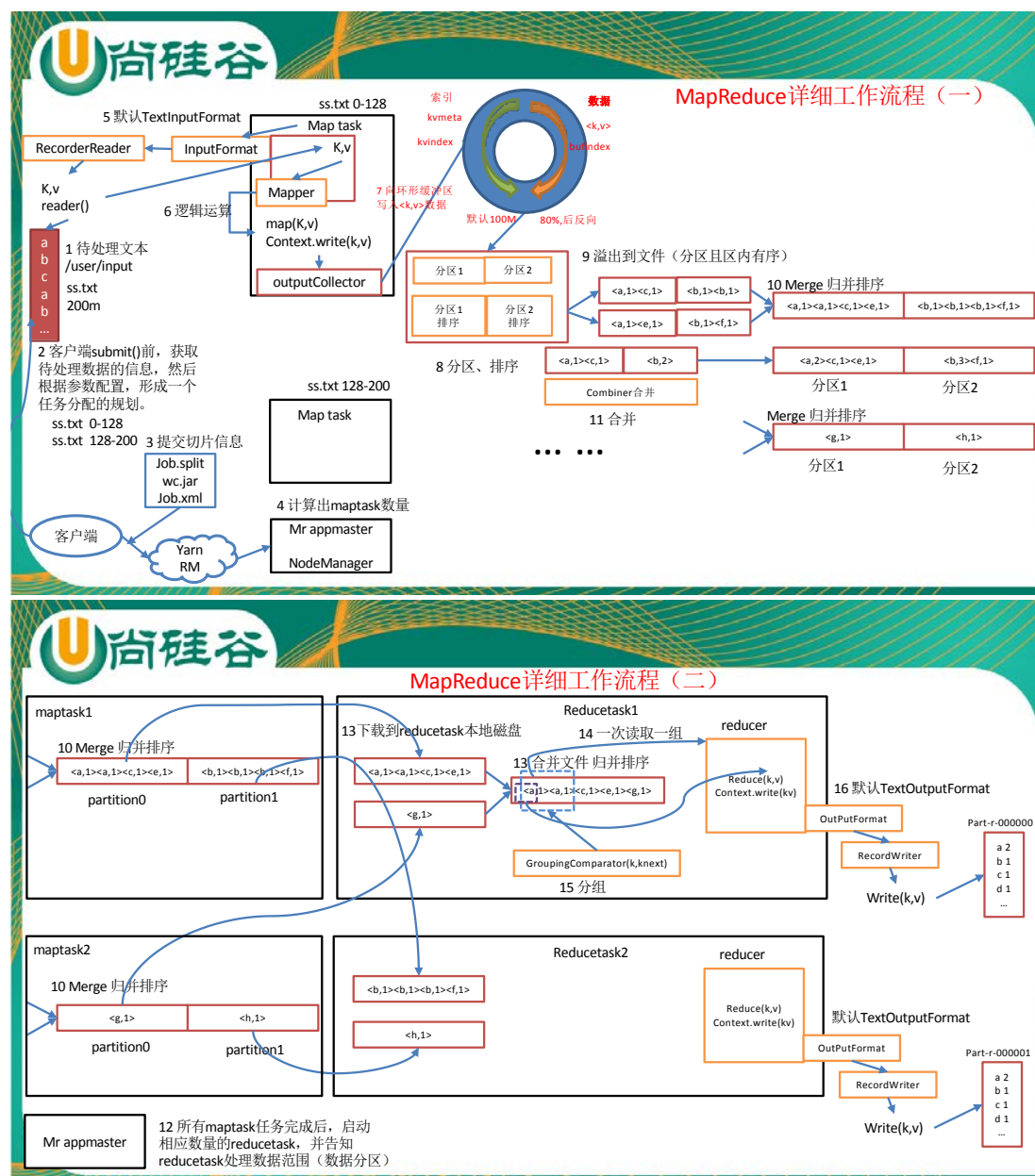


三 MapReduce 框架原理

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) `reducetask` 根据自己的分区号, 去各个 `maptask` 机器上取相应的结果分区数据

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

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

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

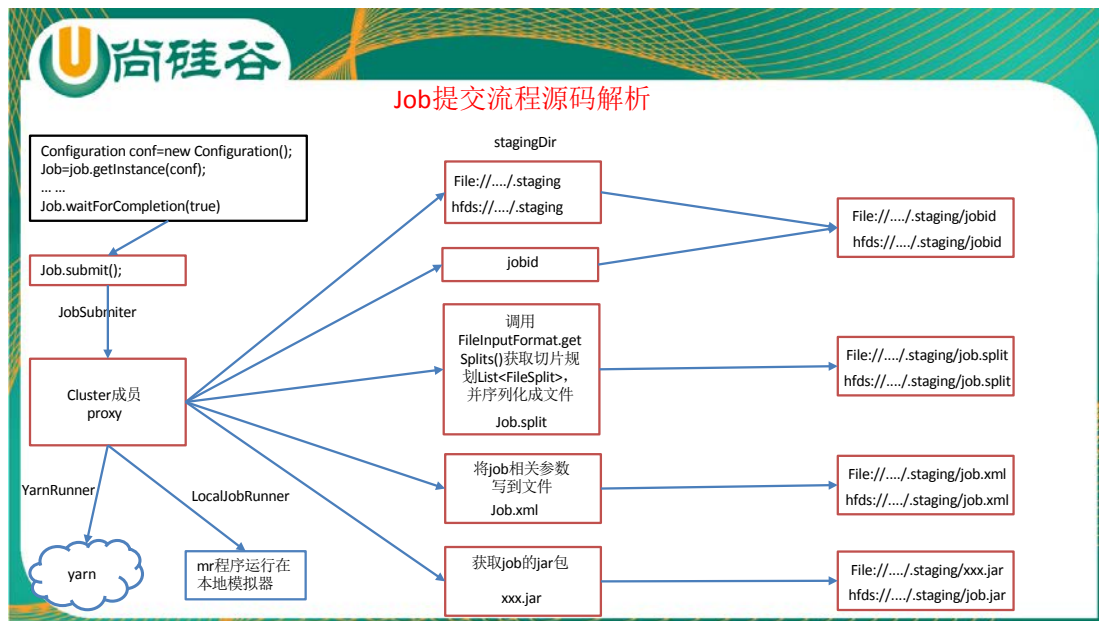
writeConf(conf, submitJobFile);

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

`computeSliteSize(Math.max(minSize,Math.min(maxSize,blocksize)))=blocksize=128M`

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 中默认切片机制：

(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 的切片机制运算后，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： $\text{Math.max}(\text{minSize}, \text{Math.min}(\text{maxSize}, \text{blockSize}))$;

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。

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
```

3.2.4 CombineTextInputFormat 案例实操

1) 需求：将输入的大量小文件合并成一个切片统一处理。

2) 输入数据: 准备 5 个小文件

3) 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

```
// 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] - DEBI
2017-05-31 10:37:17,611 INFO [org.apache.hadoop.mapreduce.JobSubmitter] - number of splits:1
```

3.2.5 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。

3.2.6 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.keyvaleTextInputFormat;
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);
    }
}

```

3.2.7 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

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
```

3.2.8 自定义 InputFormat

1) 概述

- (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。

3.2.9 自定义 InputFormat 案例实操

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，就将小文件或小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);

}

}
```

3.3 MapTask 工作机制

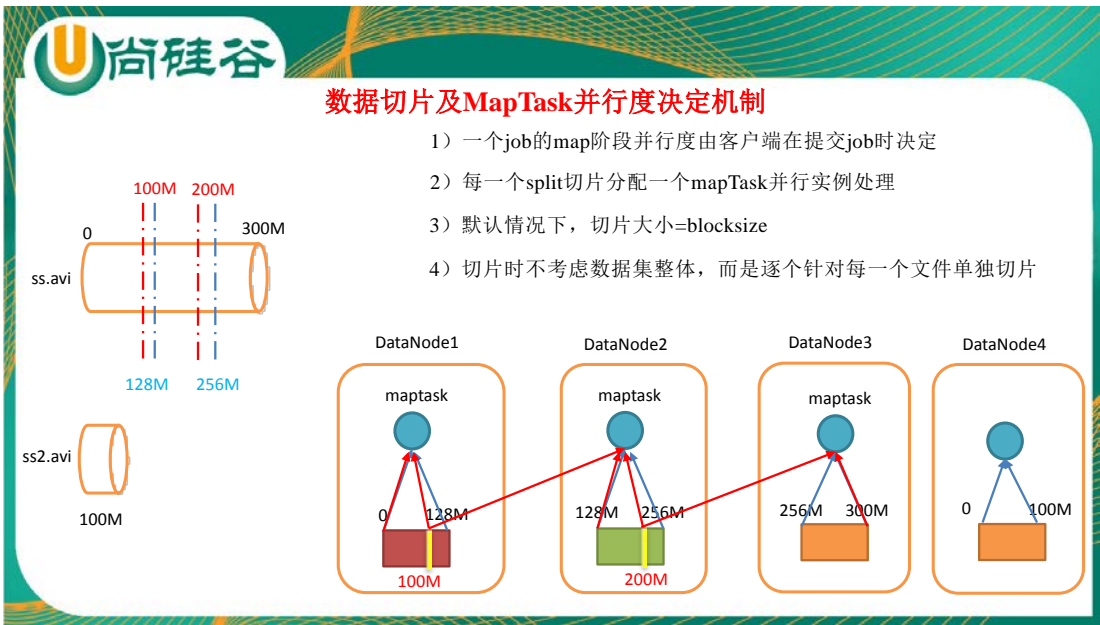
3.3.1 并行度决定机制

1) 问题引出

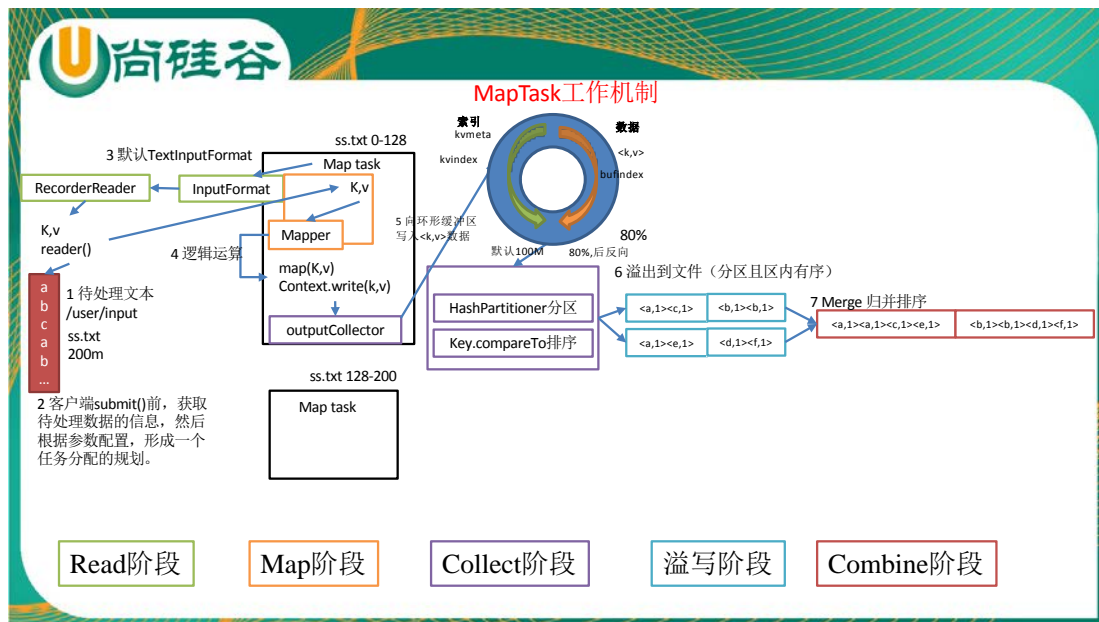
maptask 的并行度决定 map 阶段的任务处理并发度，进而影响到整个 job 的处理速度。
那么，mapTask 并行任务是否越多越好呢？

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`。

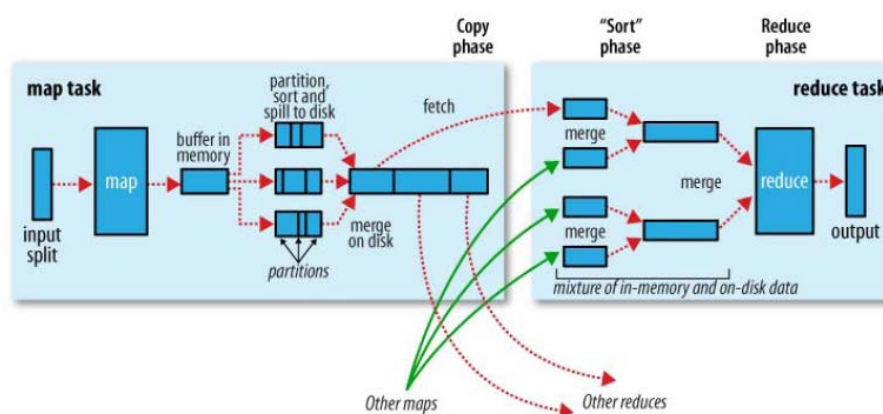
在进行文件合并过程中，MapTask 以分区为单位进行合并。对于某个分区，它将采用多轮递归合并的方式。每轮合并 `io.sort.factor`（默认 100）个文件，并将产生的文件重新加入待合并列表中，对文件排序后，重复以上过程，直到最终得到一个大文件。

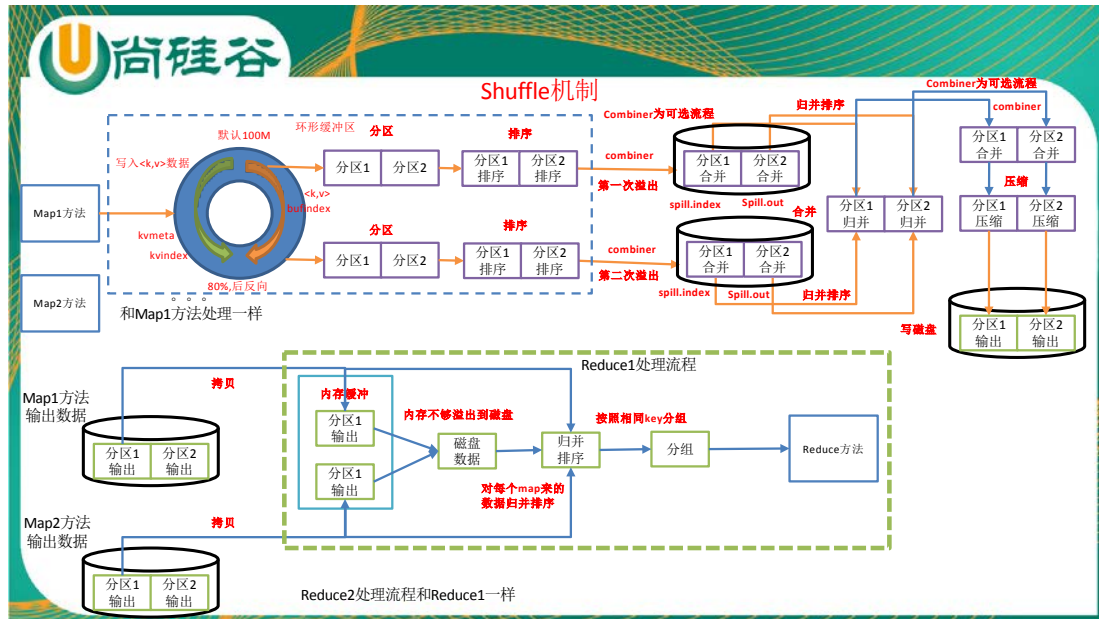
让每个 MapTask 最终只生成一个数据文件，可避免同时打开大量文件和同时读取大量小文件产生的随机读取带来的开销。

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

3.4.3 Partition 分区案例实操

1) 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）

2) 数据准备



phone_data.txt

3) 分析

(1) Mapreduce 中会将 map 输出的 kv 对，按照相同 key 分组，然后分发给不同的

reducetask。默认的分发规则为：根据 key 的 hashCode%reducetask 数来分发

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

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

4) 在案例 2.4 的基础上，增加一个分区类

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

public class ProvincePartitioner extends Partitioner<Text, FlowBean> {

    @Override
    public int getPartition(Text key, FlowBean value, int numPartitions) {
        // 1 获取电话号码的前三位
        String preNum = key.toString().substring(0, 3);

        int partition = 4;

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

        return partition;
    }
}
```

2) 在驱动函数中增加自定义数据分区设置和 reduce task 设置

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

```

public class 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);
    }
}

```

3.4.4 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，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;
}
```

3.4.5 WritableComparable 排序案例实操

案例一

1) 需求

根据案例 2.4 产生的结果再次对总流量进行排序。

2) 数据准备



phone_data.txt

3) 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;
}
```

4) 代码实现

(1) FlowBean 对象在在需求 1 基础上增加了比较功能

```
package com.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);
    }
}
```


案例二

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:

基于前一个需求，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);
```

3.4.6 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：

(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);
```

3.4.7 Combiner 合并案例实操

1) 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。



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	<code>job.setCombinerClass(WordcountReducer.class);</code>
(1) 统计单词汇总	
(2) 将统计结果输出	

2) 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();
        }
        // 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

3.4.8 GroupingComparator 分组（辅助排序）

对 reduce 阶段的数据根据某一个或几个字段进行分组。

3.4.9 GroupingComparator 分组案例实操

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:



part-r-00000.txt



part-r-00001.txt




part-r-00002.txt

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。



求每个订单中最贵的商品 (GroupingComparator)

输入数据

0000001	Pdt_01	222.8
0000002	Pdt_06	722.4
0000001	Pdt_05	25.8
0000003	Pdt_01	222.8
0000003	Pdt_01	33.8
0000002	Pdt_03	522.8
0000002	Pdt_04	122.4

maptask

1) map中处理的事情

- (1) 获取一行
- (2) 切割出每个字段
- (3) 一行封装成bean对象

bean1, nullwritable	0000001	222.8
bean2, nullwritable	0000002	722.4
bean3, nullwritable	0000001	25.8
bean4, nullwritable	0000003	222.8
bean5, nullwritable	0000003	33.8
bean6, nullwritable	0000002	522.8
bean7, nullwritable	0000002	122.4

2) 排序分区

bean1, nullwritable	0000001	222.8
bean3, nullwritable	0000001	25.8

bean2, nullwritable	0000002	722.4
bean6, nullwritable	0000002	522.8
bean7, nullwritable	0000002	122.4

bean4, nullwritable	0000003	122.4
bean5, nullwritable	0000003	33.8

预期输出数据

0000001	222.8
0000002	722.4
0000003	222.8

reducetask

3) reduce方法只把一组key的第一个写出去

0000001	222.8
25.8	

0000002	722.4
522.8	
222.8	

0000003	122.4
33.8	

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);
}
}

```

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，就只能是 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。不大于 1 肯定不执行。

3) 实验：测试 `reducetask` 多少合适。

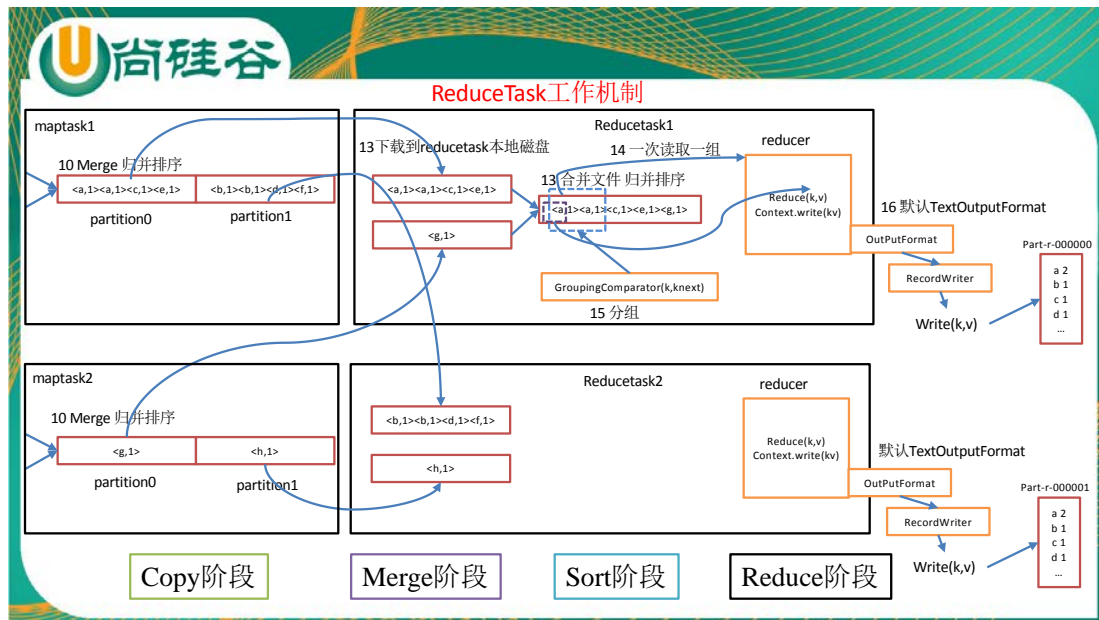
(1) 实验环境：1 个 master 节点，16 个 slave 节点：CPU:8GHZ，内存: 2G

(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()`。

3.6.3 自定义 OutputFormat 案例实操

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;
```

```

FSDDataOutputStream 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);  
    }  
}
```



```
}  
}
```

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。**

3.7.2 Reduce join 案例实操

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

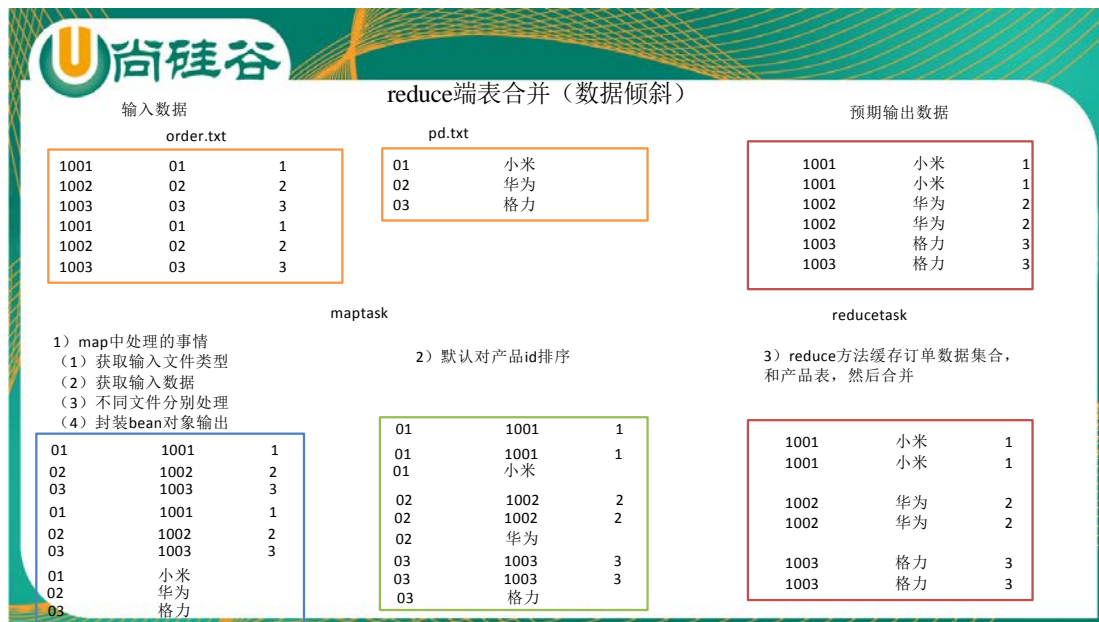
将商品信息表中数据根据商品 pid 合并到订单数据表中。

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

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;
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 setName(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

3.7.3 Map join

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



map端表合并 (Distributedcache)

1) DistributedCacheDriver 缓存文件

```
// 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reducetask数量为0
job.setNumReduceTasks(0);
```

2) 读取缓存的文件数据

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

3.7.4 Map join 案例实操

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。

2) 实现代码:

(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:/tableinput/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());
}
}

```

3.8 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

- (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。

3.9 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。

3.10 数据清洗案例实操

3.10.1 简单解析版

1) 需求：

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);
}
}

```

3.10.2 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);
}
```

3.11 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。