# 第1章 MapReduce概述

## 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**

可以实现上千台服务器集群并发工作，提供数据处理能力。

### 1.2.2 缺点

**1）不擅长实时计算**

MapReduce无法像MySQL一样，在毫秒或者秒级内返回结果。

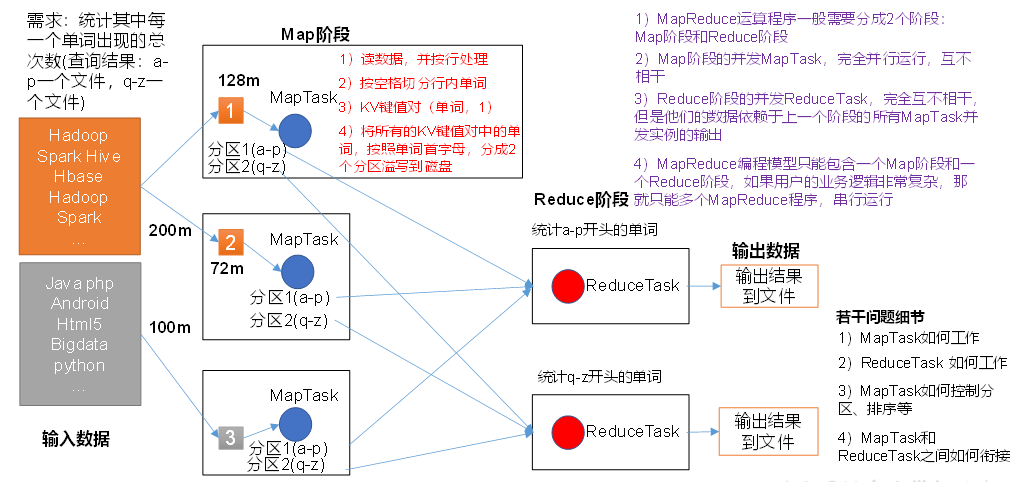
**2）不擅长流式计算**

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）不擅长DAG（有向无环图）计算(spark擅长)**

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ReduceTask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。

总结：分析WordCount数据流走向深入理解MapReduce核心思想。

## 1.4 MapReduce进程

一个完整的MapReduce程序在分布式运行时有三类实例进程：

（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 官方WordCount源码

采用反编译工具反编译源码，发现WordCount案例有Map类、Reduce类和驱动类。且数据的类型是Hadoop自身封装的序列化类型。

## 1.6 常用数据序列化类型

|  |  |
| --- | --- |
| **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 |
| Null | NullWritable |

## 1.7 MapReduce编程规范

用户编写的程序分成三个部分：Mapper、Reducer和Driver。

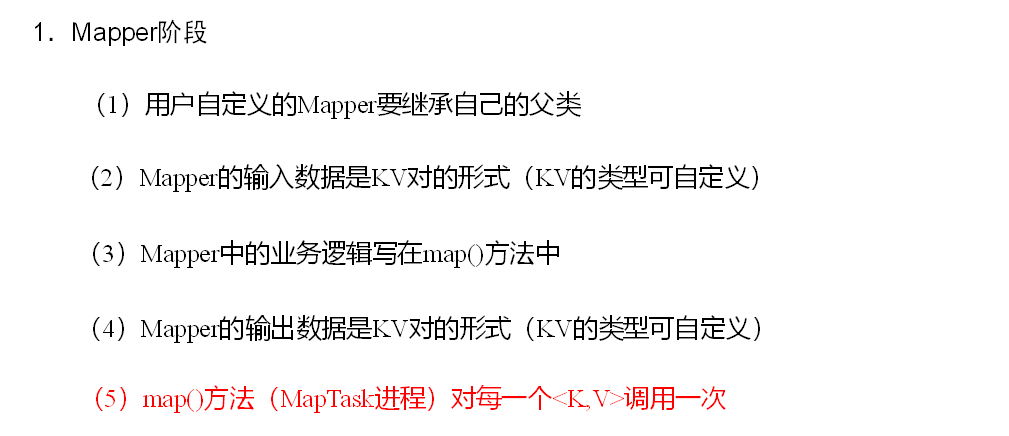


图1-7 MapReduce的编程规范

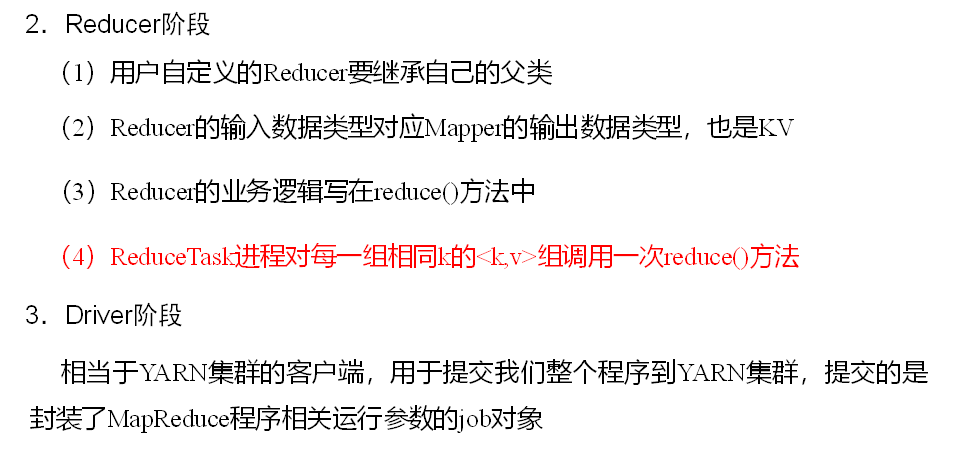


图1-8 MapReduce的编程规范

## 1.8 WordCount案例实操

**1）需求**

在给定的文本文件中统计输出每一个单词出现的总次数

**2）需求分析**

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

**3）环境准备**

（1）创建maven工程

（2）在pom.xml文件中添加如下依赖

<dependencies>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.12</version>

</dependency>

<dependency>

<groupId>org.apache.logging.log4j</groupId>

<artifactId>log4j-slf4j-impl</artifactId>

<version>2.12.0</version>

</dependency>

<dependency>

<groupId>org.apache.hadoop</groupId>

<artifactId>hadoop-client</artifactId>

<version>3.1.3</version>

</dependency>

</dependencies>

（2）在项目的src/main/resources目录下，新建一个文件，命名为“log4j2.xml”，在文件中填入。

<?xml version="1.0" encoding="UTF-8"?>

<Configuration status="error" strict="true" name="XMLConfig">

<Appenders>

<!-- 类型名为Console，名称为必须属性 -->

<Appender type="Console" name="STDOUT">

<!-- 布局为PatternLayout的方式，

输出样式为[INFO] [2018-01-22 17:34:01][org.test.Console]I'm here -->

<Layout type="PatternLayout"

pattern="[%p] [%d{yyyy-MM-dd HH:mm:ss}][%c{10}]%m%n" />

</Appender>

</Appenders>

<Loggers>

<!-- 可加性为false -->

<Logger name="test" level="info" additivity="false">

<AppenderRef ref="STDOUT" />

</Logger>

<!-- root loggerConfig设置 -->

<Root level="info">

<AppenderRef ref="STDOUT" />

</Root>

</Loggers>

</Configuration>

**4）编写程序**

（1）编写Mapper类

package com.user1.mapreduce.wordcount;

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.user1.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>{

int sum;

IntWritable v = new IntWritable();

@Override

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

// 1 累加求和

sum = 0;

for (IntWritable count : values) {

sum += count.get();

}

// 2 输出

v.set(sum);

context.write(key,v);

}

}

（3）编写Driver驱动类

package com.user1.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 获取配置信息以及获取job对象

Configuration conf = new Configuration();

Job job = Job.getInstance(conf);

// 2 关联本Driver程序的jar

job.setJarByClass(WordCountDriver.class);

// 3 关联Mapper和Reducer的jar

job.setMapperClass(WordCountMapper.class);

job.setReducerClass(WordCountReducer.class);

// 4 设置Mapper输出的kv类型

job.setMapOutputKeyClass(Text.class);

job.setMapOutputValueClass(IntWritable.class);

// 5 设置最终输出kv类型

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job

boolean result = job.waitForCompletion(true);

System.exit(result ? 0 : 1);

}

}

**5）本地测试**

（1）需要首先配置好HADOOP\_HOME变量以及Windows运行依赖

（2）在IDEA/Eclipse上运行程序

**6）集群上测试**

（0）用maven打jar包，需要添加的打包插件依赖

<build>

<plugins>

<plugin>

<artifactId>maven-compiler-plugin</artifactId>

<version>3.6.1</version>

<configuration>

<source>1.8</source>

<target>1.8</target>

</configuration>

</plugin>

<plugin>

<artifactId>maven-assembly-plugin</artifactId>

<configuration>

<descriptorRefs>

<descriptorRef>jar-with-dependencies</descriptorRef>

</descriptorRefs>

</configuration>

<executions>

<execution>

<id>make-assembly</id>

<phase>package</phase>

<goals>

<goal>single</goal>

</goals>

</execution>

</executions>

</plugin>

</plugins>

</build>

注意：如果工程上显示红叉。在项目上右键->maven->Reimport即可。

（1）将程序打成jar包，然后拷贝到Hadoop集群中

步骤详情：右键->Run as->maven install。等待编译完成就会在项目的target文件夹中生成jar包。如果看不到。在项目上右键->Refresh，即可看到。修改不带依赖的jar包名称为wc.jar，并拷贝该jar包到Hadoop集群。

（2）启动Hadoop集群

（3）执行WordCount程序

[user1@hadoop102 software]$ hadoop jar wc.jar

com.user1.wordcount.WordCountDriver /user/user1/input /user/user1/output

**7）在Windows上向集群提交任务**

（1）添加必要配置信息

public class WordCountDriver {

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

// 1 获取配置信息以及封装任务

Configuration conf = new Configuration();

//设置HDFS NameNode的地址

conf.set("fs.defaultFS", "hdfs://hadoop102:9820");

//指定MapReduce运行在Yarn上

conf.set("mapreduce.framework.name","yarn");

//指定mapreduce可以在远程集群运行

conf.set("mapreduce.app-submission.cross-platform","true");

//指定Yarn resourcemanager的位置

conf.set("yarn.resourcemanager.hostname","hadoop103");

Job job = Job.getInstance(conf);

// 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 设置最终输出kv类型

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);

}

}

（2）编辑任务配置

1）检查第一个参数Main class是不是我们要运行的类的全类名，如果不是的话一定要修改！

2）在VM options后面加上 ：-DHADOOP\_USER\_NAME=user1

3）在Program arguments后面加上两个参数分别代表输入输出路径，两个参数之间用空格隔开。如：hdfs://hadoop102:9820/input hdfs://hadoop102:9820/output

（3）打包，并将Jar包设置到Driver中

public class WordCountDriver {

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

// 1 获取配置信息以及封装任务

Configuration conf = new Configuration();

conf.set("fs.defaultFS", "hdfs://hadoop102:9820");

conf.set("mapreduce.framework.name","yarn");

conf.set("mapreduce.app-submission.cross-platform","true");

conf.set("yarn.resourcemanager.hostname","hadoop103");

Job job = Job.getInstance(conf);

// 2 设置jar加载路径

//job.setJarByClass(WordCountDriver.class);

//远程向集群提交任务,因为集群上没有自己写的代码,因此要设置下jar包

job.setJar("D:\IdeaProjects\mapreduce\target\mapreduce-1.0-SNAPSHOT.jar");

// 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 设置最终输出kv类型

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);

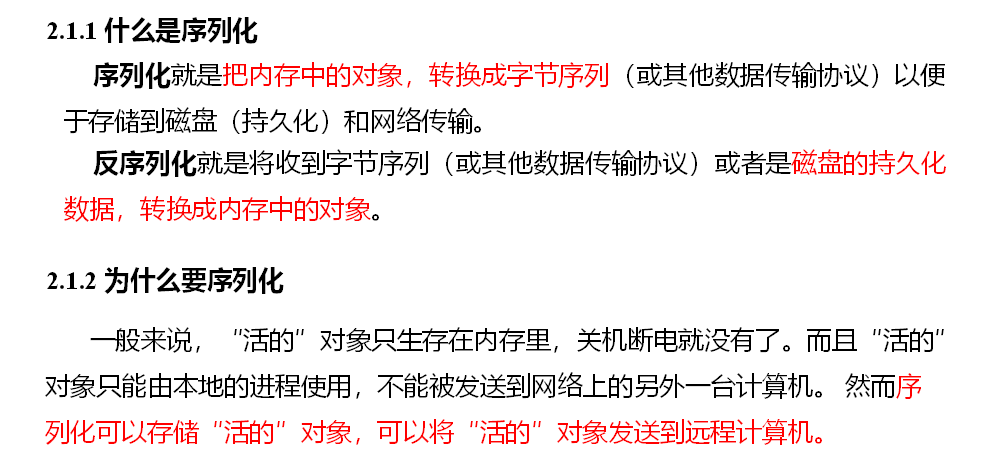
}

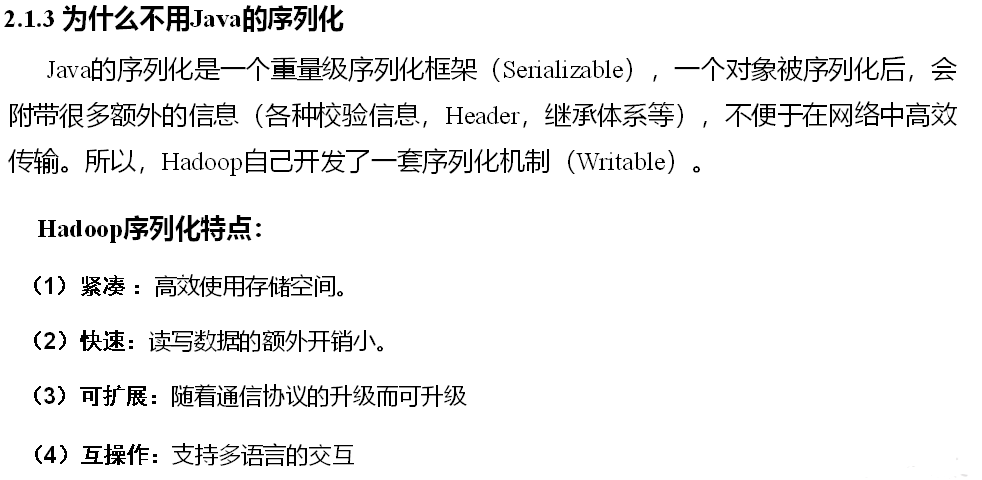
}

（4）提交并查看结果

# 第2章 Hadoop序列化

## 2.1 序列化概述





## 2.2 自定义bean对象实现序列化接口（Writable）

在企业开发中往往常用的基本序列化类型不能满足所有需求，比如在Hadoop框架内部传递一个bean对象，那么该对象就需要实现序列化接口。

具体实现bean对象序列化步骤如下7步。

（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

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过程要求对key必须能排序。详见后面排序案例。

@Override

public int compareTo(FlowBean o) {

// 倒序排列，从大到小

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

}

## 2.3 序列化案例实操

**1）需求**

统计每一个手机号耗费的总上行流量、总下行流量、总流量

（1）输入数据

|  |
| --- |
| 1 13736230513 192.196.100.1 www.atguigu.com 2481 24681 200  2 13846544121 192.196.100.2 264 0 200  3 13956435636 192.196.100.3 132 1512 200  4 13966251146 192.168.100.1 240 0 404  5 18271575951 192.168.100.2 www.atguigu.com 1527 2106 200  6 84188413 192.168.100.3 www.atguigu.com 4116 1432 200  7 13590439668 192.168.100.4 1116 954 200  8 15910133277 192.168.100.5 www.hao123.com 3156 2936 200  9 13729199489 192.168.100.6 240 0 200  10 13630577991 192.168.100.7 www.shouhu.com 6960 690 200  11 15043685818 192.168.100.8 www.baidu.com 3659 3538 200  12 15959002129 192.168.100.9 www.atguigu.com 1938 180 500  13 13560439638 192.168.100.10 918 4938 200  14 13470253144 192.168.100.11 180 180 200  15 13682846555 192.168.100.12 www.qq.com 1938 2910 200  16 13992314666 192.168.100.13 www.gaga.com 3008 3720 200  17 13509468723 192.168.100.14 www.qinghua.com 7335 110349 404  18 18390173782 192.168.100.15 www.sogou.com 9531 2412 200  19 13975057813 192.168.100.16 www.baidu.com 11058 48243 200  20 13768778790 192.168.100.17 120 120 200  21 13568436656 192.168.100.18 www.alibaba.com 2481 24681 200  22 13568436656 192.168.100.19 1116 954 200 |

（2）输入数据格式：

|  |
| --- |
| 7 13560436666 120.196.100.99 1116 954 200  id 手机号码 网络ip 上行流量 下行流量 网络状态码 |

（3）期望输出数据格式

|  |
| --- |
| 13560436666 1116 954 2070  手机号码 上行流量 下行流量 总流量 |

**2）需求分析**



**3）编写MapReduce程序**

（1）编写流量统计的Bean对象

package com.user1.mapreduce.writable;

import org.apache.hadoop.io.Writable;

import java.io.DataInput;

import java.io.DataOutput;

import java.io.IOException;

//1 继承Writable接口

public class FlowBean implements Writable {

private long upFlow; //上行流量

private long downFlow; //下行流量

private long sumFlow; //总流量

//2 提供无参构造

public FlowBean() {

}

//3 提供三个参数的getter和setter方法

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;

}

public void setSumFlow() {

this.sumFlow = this.upFlow + this.downFlow;

}

//4 实现序列化和反序列化方法,注意顺序一定要保持一致

@Override

public void write(DataOutput out) throws IOException {

out.writeLong(upFlow);

out.writeLong(downFlow);

out.writeLong(sumFlow);

}

@Override

public void readFields(DataInput in) throws IOException {

this.upFlow = in.readLong();

this.downFlow = in.readLong();

this.sumFlow = in.readLong();

}

//5 重写ToString

@Override

public String toString() {

return upFlow + "\t" + downFlow + "\t" + sumFlow;

}

}

（2）编写Mapper类

package com.user1.mapreduce.writable;

import org.apache.hadoop.io.LongWritable;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapreduce.Mapper;

import java.io.IOException;

public class FlowMapper extends Mapper<LongWritable, Text, Text, FlowBean> {

private Text outK = new Text();

private FlowBean outV = new FlowBean();

@Override

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

//1 获取一行数据,转成字符串

String line = value.toString();

//2 切割数据

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

//3 抓取我们需要的数据:手机号,上行流量,下行流量

String phone = split[1];

String up = split[split.length - 3];

String down = split[split.length - 2];

//4 封装outK outV

outK.set(phone);

outV.setUpFlow(Long.parseLong(up));

outV.setDownFlow(Long.parseLong(down));

outV.setSumFlow();

//5 写出outK outV

context.write(outK, outV);

}

}

（3）编写Reducer类

package com.user1.mapreduce.writable;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapreduce.Reducer;

import java.io.IOException;

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

private FlowBean outV = new FlowBean();

@Override

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

long totalUp = 0;

long totalDown = 0;

//1 遍历values,将其中的上行流量,下行流量分别累加

for (FlowBean flowBean : values) {

totalUp += flowBean.getUpFlow();

totalDown += flowBean.getDownFlow();

}

//2 封装outKV

outV.setUpFlow(totalUp);

outV.setDownFlow(totalDown);

outV.setSumFlow();

//3 写出outK outV

context.write(key,outV);

}

}

（4）编写Driver驱动类

package com.user1.mapreduce.writable;

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;

import java.io.IOException;

public class FlowDriver {

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

//1 获取job对象

Configuration conf = new Configuration();

Job job = Job.getInstance(conf);

//2 关联本Driver类

job.setJarByClass(FlowDriver.class);

//3 关联Mapper和Reducer

job.setMapperClass(FlowMapper.class);

job.setReducerClass(FlowReducer.class);

//4 设置Map端输出KV类型

job.setMapOutputKeyClass(Text.class);

job.setMapOutputValueClass(FlowBean.class);

//5 设置程序最终输出的KV类型

job.setOutputKeyClass(Text.class);

job.setOutputValueClass(FlowBean.class);

//6 设置程序的输入输出路径

FileInputFormat.setInputPaths(job, new Path("D:\\inputflow"));

FileOutputFormat.setOutputPath(job, new Path("D:\\flowoutput"));

//7 提交Job

boolean b = job.waitForCompletion(true);

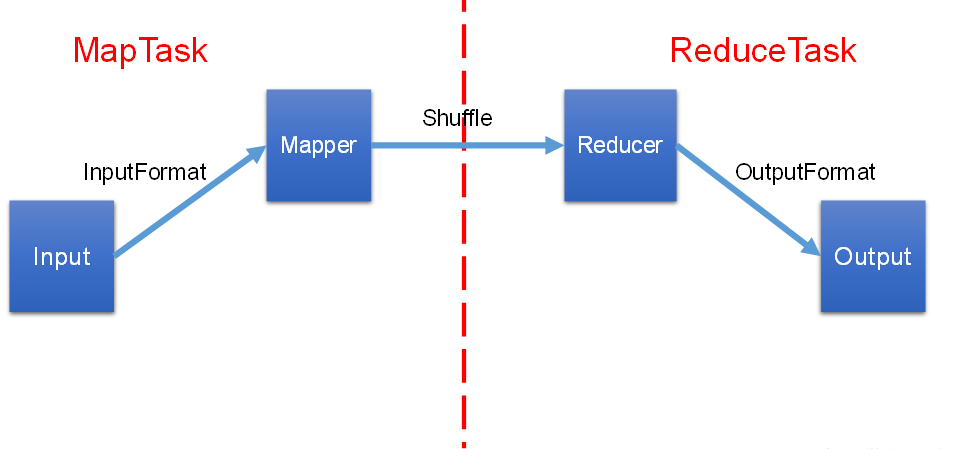
System.exit(b ? 0 : 1);

}

}

# 第3章 MapReduce框架原理

## 3.1 InputFormat数据输入



### 3.1.1 切片与MapTask并行度决定机制

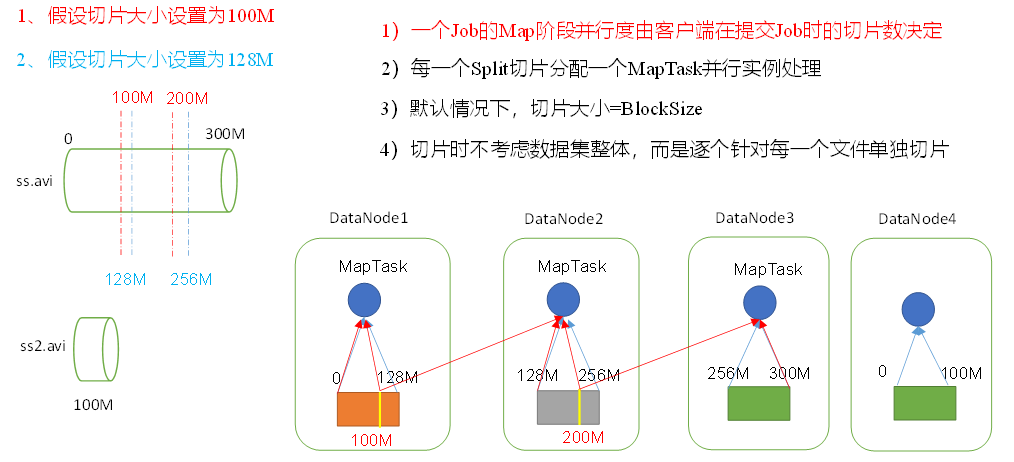
**1）问题引出**

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。

**2）MapTask并行度决定机制**

**数据块：**Block是HDFS物理上把数据分成一块一块。数据块是HDFS存储数据单位。

**数据切片：**数据切片只是在逻辑上对输入进行分片，并不会在磁盘上将其切分成片进行存储。数据切片是MapReduce程序计算输入数据的单位，一个切片会对应启动一个MapTask。



### 3.1.2 Job提交流程源码和切片源码详解

**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）判断是本地运行环境还是yarn集群运行环境

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);

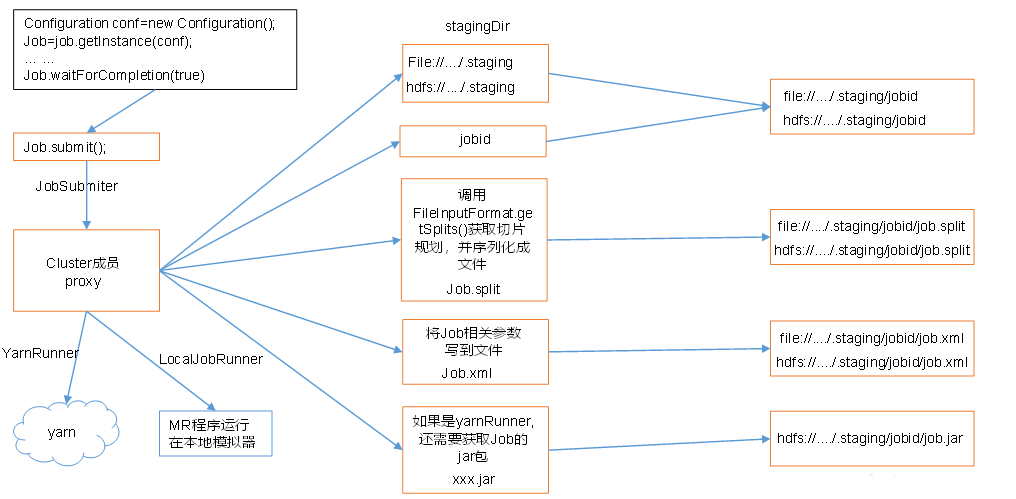
// 5）向Stag路径写XML配置文件

writeConf(conf, submitJobFile);

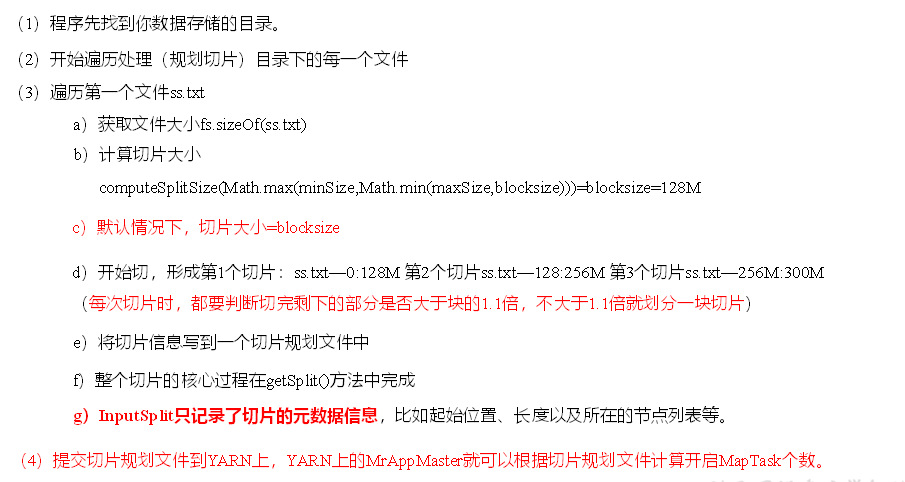
conf.writeXml(out);

// 6）提交Job,返回提交状态

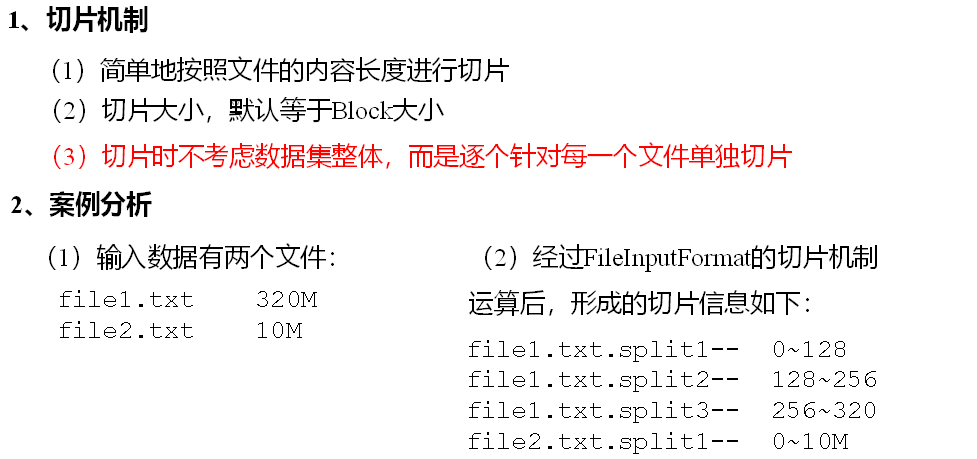
status = submitClient.submitJob(jobId, submitJobDir.toString(), job.getCredentials());



**2）FileInputFormat切片源码解析(input.getSplits(job))**



### 3.1.3 FileInputFormat切片机制





### 3.1.4 TextInputFormat

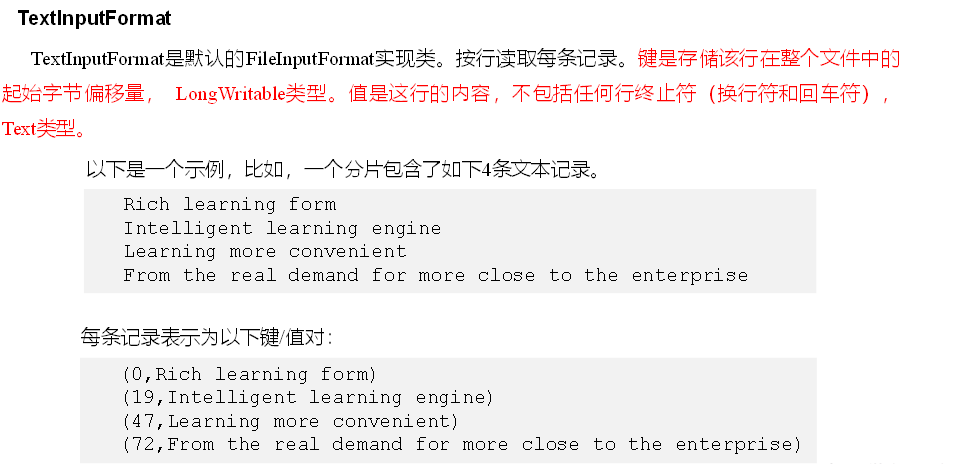


图3-8 TextInputFormat

### 3.1.5 CombineTextInputFormat切片机制

**1）应用场景：**

CombineTextInputFormat用于小文件过多的场景，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处理。

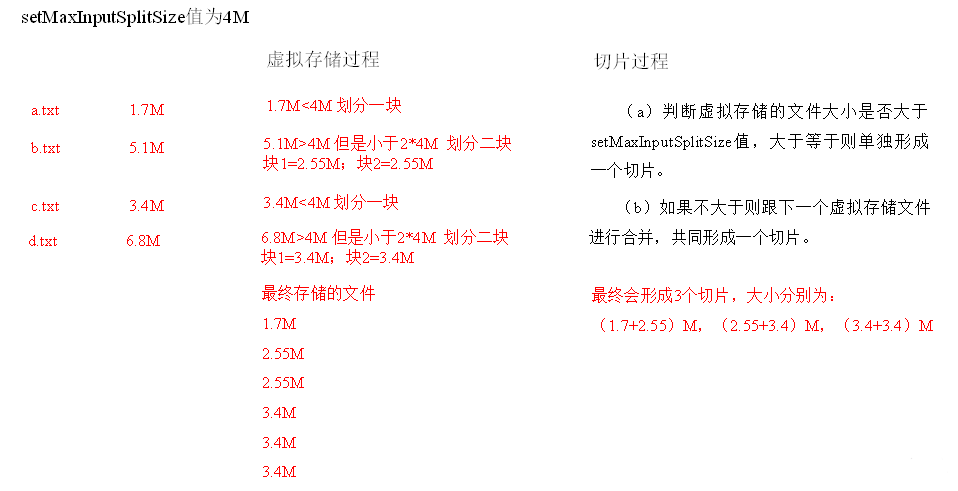
**2）虚拟存储切片最大值设置**

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

注意：虚拟存储切片最大值设置最好根据实际的小文件大小情况来设置具体的值。

**3）切片机制**

生成切片过程包括：虚拟存储过程和切片过程二部分。



（1）虚拟存储过程：

将输入目录下所有文件大小，依次和设置的setMaxInputSplitSize值比较，如果不大于设置的最大值，逻辑上划分一个块。如果输入文件大于设置的最大值且大于两倍，那么以最大值切割一块；当剩余数据大小超过设置的最大值且不大于最大值2倍，此时将文件均分成2个虚拟存储块（防止出现太小切片）。

例如setMaxInputSplitSize值为4M，输入文件大小为8.02M，则先逻辑上分成一个4M。剩余的大小为4.02M，如果按照4M逻辑划分，就会出现0.02M的小的虚拟存储文件，所以将剩余的4.02M文件切分成（2.01M和2.01M）两个文件。

（2）切片过程：

（a）判断虚拟存储的文件大小是否大于setMaxInputSplitSize值，大于等于则单独形成一个切片。

（b）如果不大于则跟下一个虚拟存储文件进行合并，共同形成一个切片。

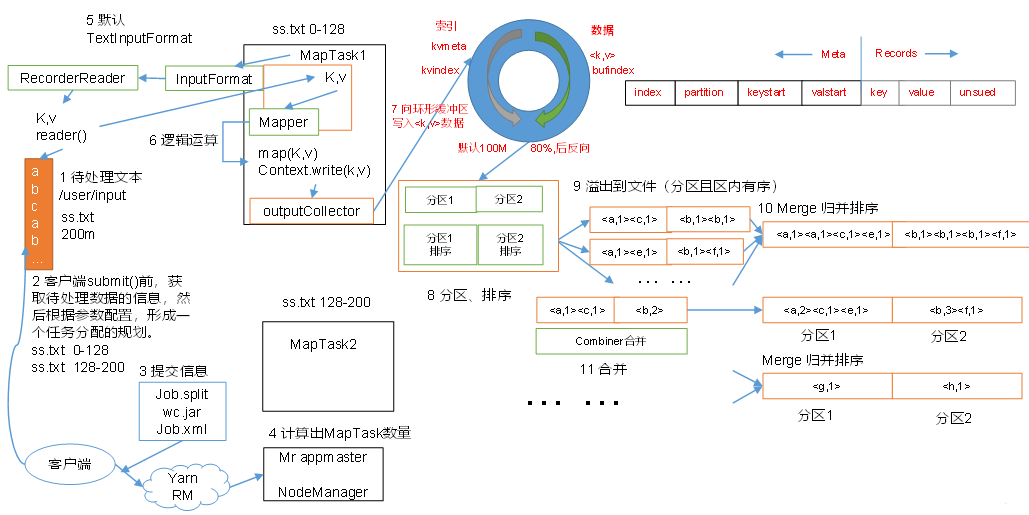
（c）测试举例：有4个小文件大小分别为1.7M、5.1M、3.4M以及6.8M这四个小文件，则虚拟存储之后形成6个文件块，大小分别为：

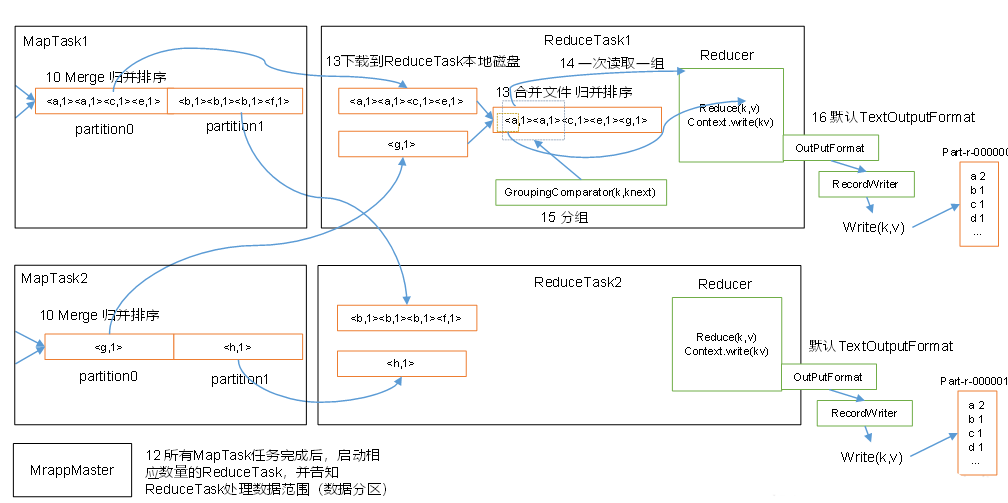
1.7M，（2.55M、2.55M），3.4M以及（3.4M、3.4M）

最终会形成3个切片，大小分别为：

（1.7+2.55）M，（2.55+3.4）M，（3.4+3.4）M

## 3.2 MapReduce工作流程





上面的流程是整个MapReduce最全工作流程，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会抓取到同一个分区的来自不同MapTask的结果文件，ReduceTask会将这些文件再进行合并（归并排序）

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

**注意：**

（1）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。

（2）缓冲区的大小可以通过参数调整，参数：mapreduce.task.io.sort.mb默认100M。

（3）源码解析流程

=================== MapTask ===================

context.write(k, NullWritable.get()); //自定义的map方法的写出，进入

output.write(key, value);

//MapTask727行，收集方法，进入两次

collector.collect(key, value,partitioner.getPartition(key, value, partitions));

HashPartitioner(); //默认分区器

collect() //MapTask1082行 map端所有的kv全部写出后会走下面的close方法

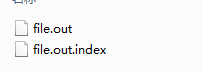
close() //MapTask732行

collector.flush() // 溢出刷写方法，MapTask735行，提前打个断点，进入

sortAndSpill() //溢写排序，MapTask1505行，进入

sorter.sort() QuickSort //溢写排序方法，MapTask1625行，进入

mergeParts(); //合并文件，MapTask1527行，进入



collector.close();

=================== ReduceTask ===================

if (isMapOrReduce()) //reduceTask324行，提前打断点

initialize() // reduceTask333行,进入

init(shuffleContext); // reduceTask375行,走到这需要先给下面的打断点

totalMaps = job.getNumMapTasks(); // ShuffleSchedulerImpl第120行，提前打断点

merger = createMergeManager(context); //合并方法，Shuffle第80行

// MergeManagerImpl第232 235行，提前打断点

this.inMemoryMerger = createInMemoryMerger(); //内存合并

this.onDiskMerger = new OnDiskMerger(this); //磁盘合并

eventFetcher.start(); //开始抓取数据，Shuffle第107行，提前打断点

eventFetcher.shutDown(); //抓取结束，Shuffle第141行，提前打断点

copyPhase.complete(); //copy阶段完成，Shuffle第151行

taskStatus.setPhase(TaskStatus.Phase.SORT); //开始排序阶段，Shuffle第152行

sortPhase.complete(); //排序阶段完成，即将进入reduce阶段 reduceTask382行

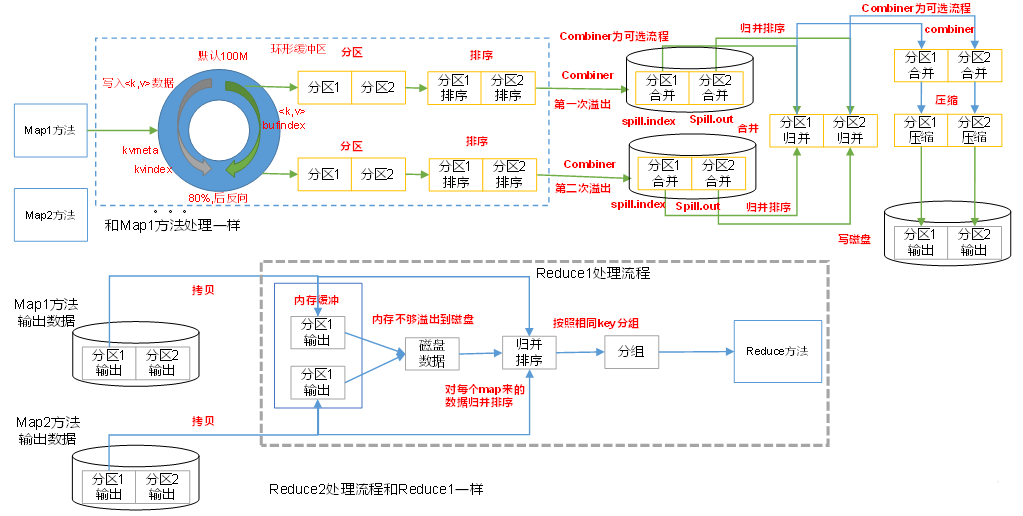
reduce(); //reduce阶段调用的就是我们自定义的reduce方法，会被调用多次

cleanup(context); //reduce完成之前，会最后调用一次Reducer里面的cleanup方法

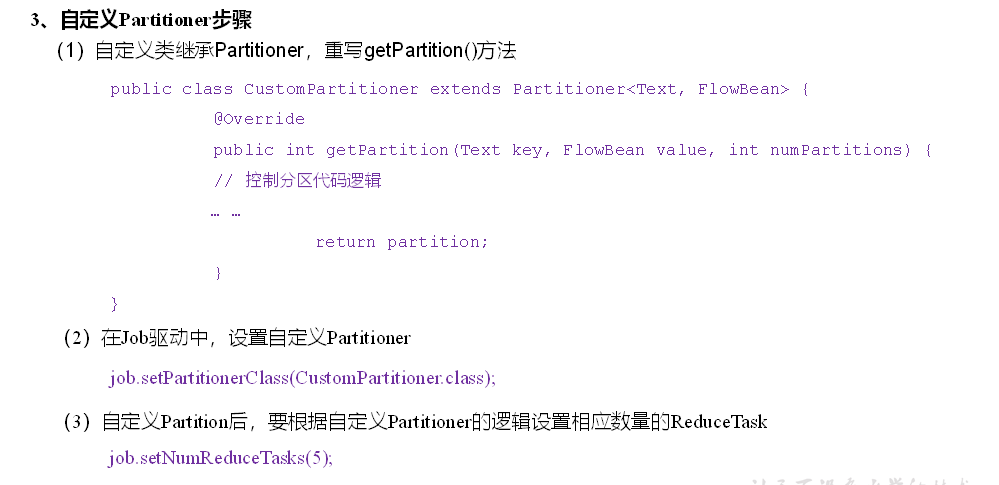
## 3.3 Shuffle机制

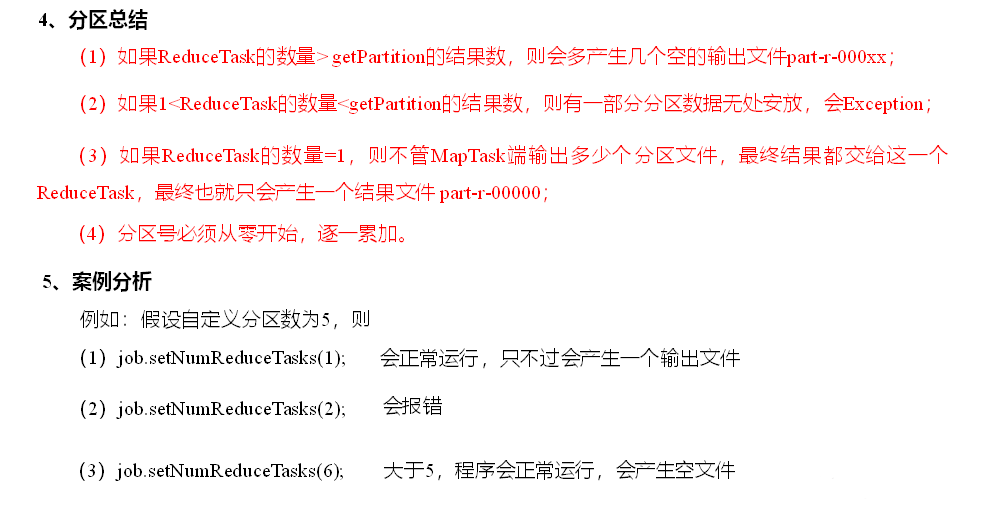
### 3.3.1 Shuffle机制

Map方法之后，Reduce方法之前的数据处理过程称之为Shuffle。



### 3.3.2 Partition分区





### 3.3.3 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）

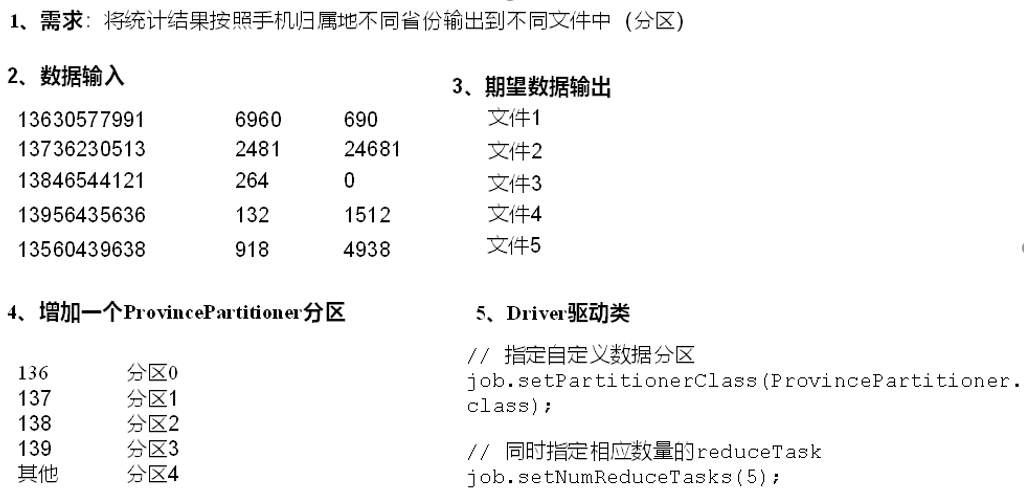
（1）输入数据

|  |
| --- |
| 1 13736230513 192.196.100.1 www.atguigu.com 2481 24681 200  2 13846544121 192.196.100.2 264 0 200  3 13956435636 192.196.100.3 132 1512 200  4 13966251146 192.168.100.1 240 0 404  5 18271575951 192.168.100.2 www.atguigu.com 1527 2106 200  6 84188413 192.168.100.3 www.atguigu.com 4116 1432 200  7 13590439668 192.168.100.4 1116 954 200  8 15910133277 192.168.100.5 www.hao123.com 3156 2936 200  9 13729199489 192.168.100.6 240 0 200  10 13630577991 192.168.100.7 www.shouhu.com 6960 690 200  11 15043685818 192.168.100.8 www.baidu.com 3659 3538 200  12 15959002129 192.168.100.9 www.atguigu.com 1938 180 500  13 13560439638 192.168.100.10 918 4938 200  14 13470253144 192.168.100.11 180 180 200  15 13682846555 192.168.100.12 www.qq.com 1938 2910 200  16 13992314666 192.168.100.13 www.gaga.com 3008 3720 200  17 13509468723 192.168.100.14 www.qinghua.com 7335 110349 404  18 18390173782 192.168.100.15 www.sogou.com 9531 2412 200  19 13975057813 192.168.100.16 www.baidu.com 11058 48243 200  20 13768778790 192.168.100.17 120 120 200  21 13568436656 192.168.100.18 www.alibaba.com 2481 24681 200  22 13568436656 192.168.100.19 1116 954 200 |

（2）期望输出数据

手机号136、137、138、139开头都分别放到一个独立的4个文件中，其他开头的放到一个文件中。

**2）需求分析**



**3）在案例2.4的基础上，增加一个分区类**

package com.user1.mapreduce.partitioner;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapreduce.Partitioner;

public class ProvincePartitioner extends Partitioner<Text, FlowBean> {

@Override

public int getPartition(Text text, FlowBean flowBean, int numPartitions) {

//获取手机号前三位prePhone

String phone = text.toString();

String prePhone = phone.substring(0, 3);

//定义一个分区号变量partition,根据prePhone设置分区号

int partition;

if("136".equals(prePhone)){

partition = 0;

}else if("137".equals(prePhone)){

partition = 1;

}else if("138".equals(prePhone)){

partition = 2;

}else if("139".equals(prePhone)){

partition = 3;

}else {

partition = 4;

}

//最后返回分区号partition

return partition;

}

}

**4）在驱动函数中增加自定义数据分区设置和ReduceTask设置**

package com.user1.mapreduce.partitioner;

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;

import java.io.IOException;

public class FlowDriver {

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

//1 获取job对象

Configuration conf = new Configuration();

Job job = Job.getInstance(conf);

//2 关联本Driver类

job.setJarByClass(FlowDriver.class);

//3 关联Mapper和Reducer

job.setMapperClass(FlowMapper.class);

job.setReducerClass(FlowReducer.class);

//4 设置Map端输出数据的KV类型

job.setMapOutputKeyClass(Text.class);

job.setMapOutputValueClass(FlowBean.class);

//5 设置程序最终输出的KV类型

job.setOutputKeyClass(Text.class);

job.setOutputValueClass(FlowBean.class);

//8 指定自定义分区器

job.setPartitionerClass(ProvincePartitioner.class);

//9 同时指定相应数量的ReduceTask

job.setNumReduceTasks(5);

//6 设置输入输出路径

FileInputFormat.setInputPaths(job, new Path("D:\\inputflow"));

FileOutputFormat.setOutputPath(job, new Path("D\\partitionout"));

//7 提交Job

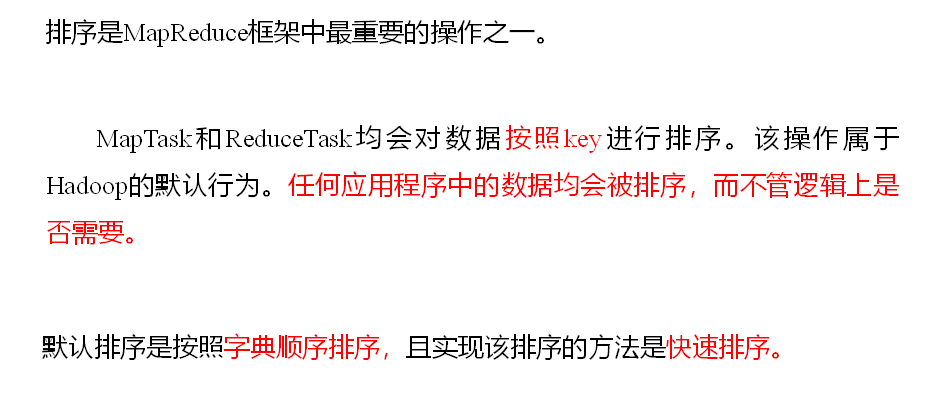
boolean b = job.waitForCompletion(true);

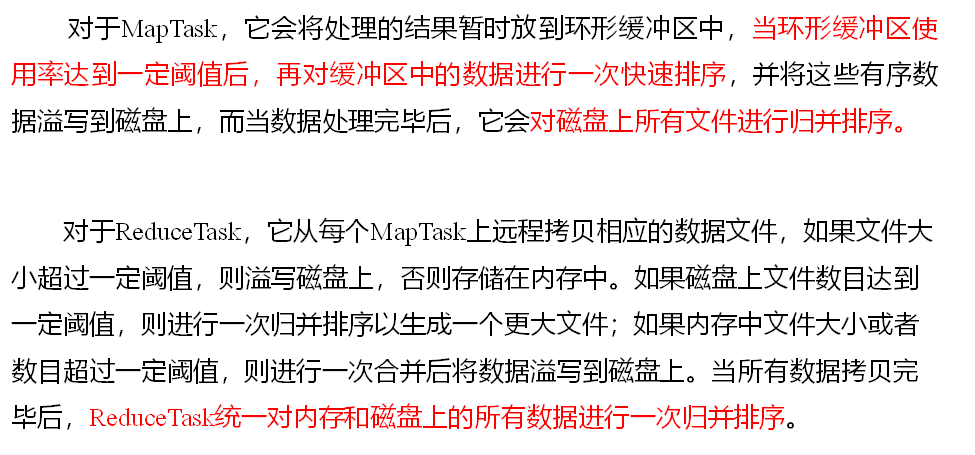
System.exit(b ? 0 : 1);

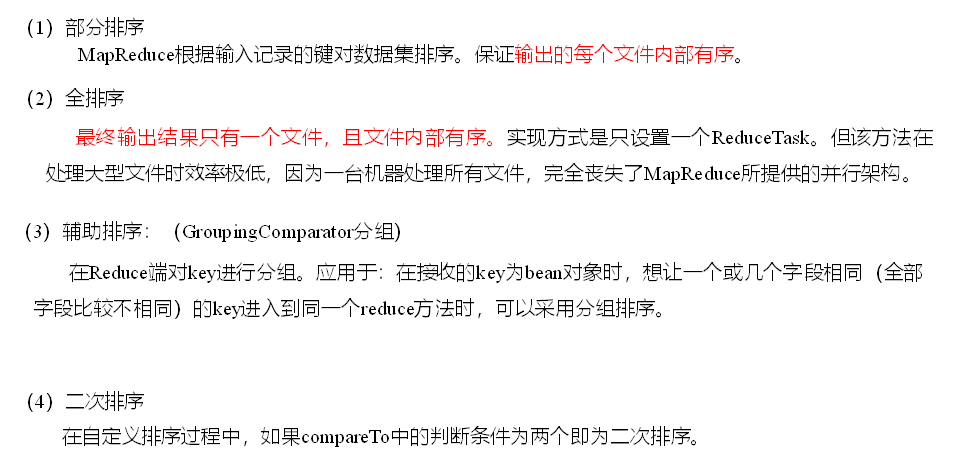
}

}

### 3.3.4 WritableComparable排序







**自定义排序WritableComparable原理分析**

bean对象做为key传输，需要实现WritableComparable接口重写compareTo方法，就可以实现排序。

@Override

public int compareTo(FlowBean bean) {

int result;

// 按照总流量大小，倒序排列

if (this.sumFlow > bean.getSumFlow()) {

result = -1;

}else if (this.sumFlow < bean.getSumFlow()) {

result = 1;

}else {

result = 0;

}

return result;

}

### 3.3.5 WritableComparable排序案例实操（全排序）

**1）需求**

根据案例2.3序列化案例产生的结果再次对总流量进行倒序排序。

（1）输入数据

原始数据 第一次处理后的数据

如上

（2）期望输出数据

13509468723 7335 110349 117684

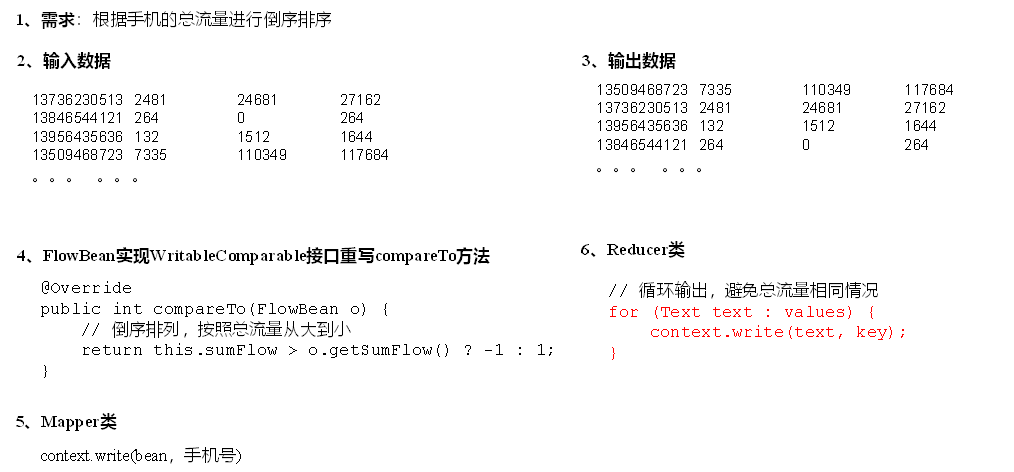
13736230513 2481 24681 27162

13956435636 132 1512 1644

13846544121 264 0 264

。。。 。。。

**2）需求分析**



**3）代码实现**

（1）FlowBean对象在在需求1基础上增加了比较功能

package com.user1.mapreduce.writablecompable;

import org.apache.hadoop.io.WritableComparable;

import java.io.DataInput;

import java.io.DataOutput;

import java.io.IOException;

public class FlowBean implements WritableComparable<FlowBean> {

private long upFlow; //上行流量

private long downFlow; //下行流量

private long sumFlow; //总流量

//提供无参构造

public FlowBean() {

}

//生成三个属性的getter和setter方法

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;

}

public void setSumFlow() {

this.sumFlow = this.upFlow + this.downFlow;

}

//实现序列化和反序列化方法,注意顺序一定要一致

@Override

public void write(DataOutput out) throws IOException {

out.writeLong(this.upFlow);

out.writeLong(this.downFlow);

out.writeLong(this.sumFlow);

}

@Override

public void readFields(DataInput in) throws IOException {

this.upFlow = in.readLong();

this.downFlow = in.readLong();

this.sumFlow = in.readLong();

}

//重写ToString,最后要输出FlowBean

@Override

public String toString() {

return upFlow + "\t" + downFlow + "\t" + sumFlow;

}

@Override

public int compareTo(FlowBean o) {

//按照总流量比较,倒序排列

if(this.sumFlow > o.sumFlow){

return -1;

}else if(this.sumFlow < o.sumFlow){

return 1;

}else {

return 0;

}

}

}

（2）编写Mapper类

package com.user1.mapreduce.writablecompable;

import org.apache.hadoop.io.LongWritable;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapreduce.Mapper;

import java.io.IOException;

public class FlowMapper extends Mapper<LongWritable, Text, FlowBean, Text> {

private FlowBean outK = new FlowBean();

private Text outV = new Text();

@Override

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

//1 获取一行数据

String line = value.toString();

//2 按照"\t",切割数据

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

//3 封装outK outV

outK.setUpFlow(Long.parseLong(split[1]));

outK.setDownFlow(Long.parseLong(split[2]));

outK.setSumFlow();

outV.set(split[0]);

//4 写出outK outV

context.write(outK,outV);

}

}

（3）编写Reducer类

package com.user1.mapreduce.writablecompable;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapreduce.Reducer;

import java.io.IOException;

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

@Override

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

//遍历values集合,循环写出,避免总流量相同的情况

for (Text value : values) {

//调换KV位置,反向写出

context.write(value,key);

}

}

}

（4）编写Driver类

package com.user1.mapreduce.writablecompable;

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;

import java.io.IOException;

public class FlowDriver {

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

//1 获取job对象

Configuration conf = new Configuration();

Job job = Job.getInstance(conf);

//2 关联本Driver类

job.setJarByClass(FlowDriver.class);

//3 关联Mapper和Reducer

job.setMapperClass(FlowMapper.class);

job.setReducerClass(FlowReducer.class);

//4 设置Map端输出数据的KV类型

job.setMapOutputKeyClass(FlowBean.class);

job.setMapOutputValueClass(Text.class);

//5 设置程序最终输出的KV类型

job.setOutputKeyClass(Text.class);

job.setOutputValueClass(FlowBean.class);

//6 设置输入输出路径

FileInputFormat.setInputPaths(job, new Path("D:\\inputflow2"));

FileOutputFormat.setOutputPath(job, new Path("D:\\comparout"));

//7 提交Job

boolean b = job.waitForCompletion(true);

System.exit(b ? 0 : 1);

}

}

### 3.3.6 WritableComparable排序案例实操（区内排序）

**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）需求分析**

基于前一个需求，增加自定义分区类，分区按照省份手机号设置。



**3）案例实操**

（1）增加自定义分区类

package com.user1.mapreduce.partitionercompable;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapreduce.Partitioner;

public class ProvincePartitioner2 extends Partitioner<FlowBean, Text> {

@Override

public int getPartition(FlowBean flowBean, Text text, int numPartitions) {

//获取手机号前三位

String phone = text.toString();

String prePhone = phone.substring(0, 3);

//定义一个分区号变量partition,根据prePhone设置分区号

int partition;

if("136".equals(prePhone)){

partition = 0;

}else if("137".equals(prePhone)){

partition = 1;

}else if("138".equals(prePhone)){

partition = 2;

}else if("139".equals(prePhone)){

partition = 3;

}else {

partition = 4;

}

//最后返回分区号partition

return partition;

}

}

（2）在驱动类中添加分区类

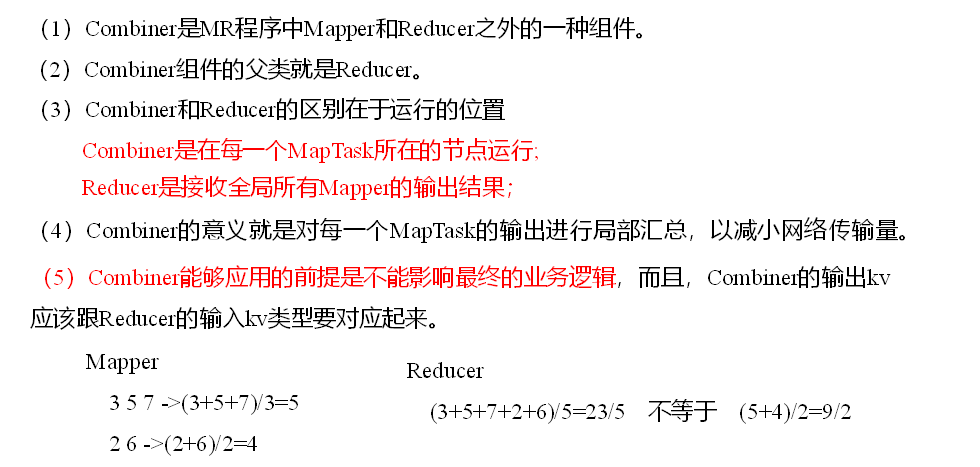
// 设置自定义分区器

job.setPartitionerClass(ProvincePartitioner2.class);

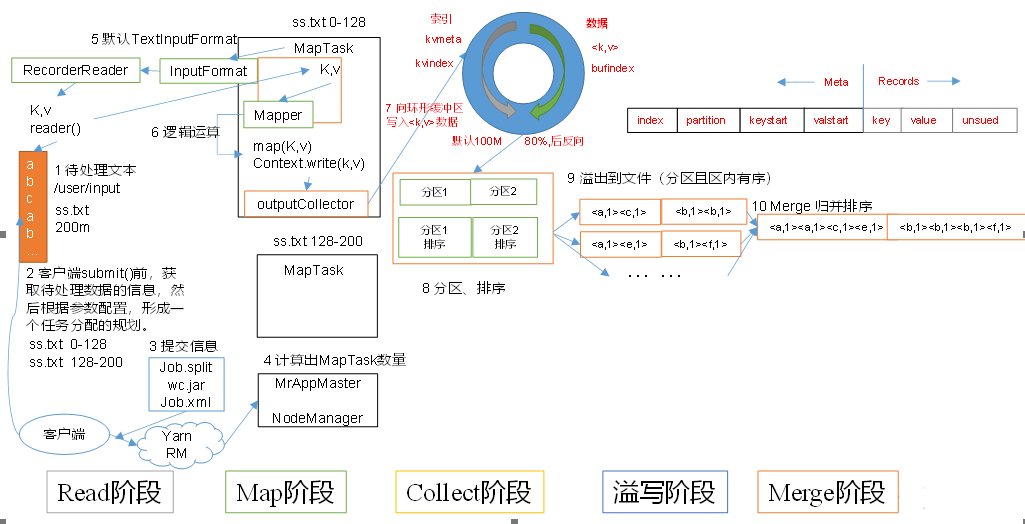
// 设置对应的ReduceTask的个数

job.setNumReduceTasks(5);

### 3.3.7 Combiner合并



## 3.4 MapTask工作机制



（1）Read阶段：MapTask通过InputFormat获得的RecordReader，从输入InputSplit中解析出一个个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。

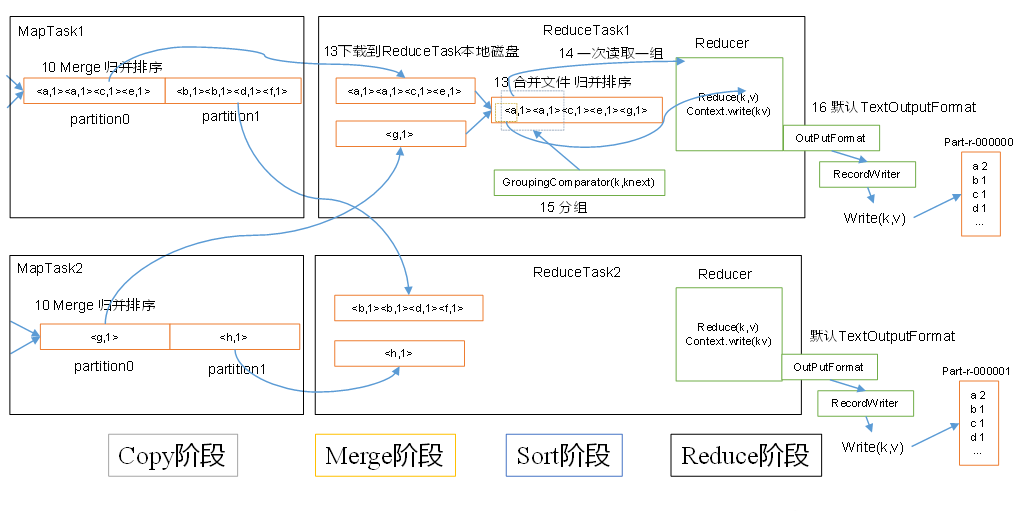
（5）Merge阶段：当所有数据处理完成后，MapTask对所有临时文件进行一次合并，以确保最终只会生成一个数据文件。

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mapreduce.task.io.sort.factor（默认10）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。

## 3.5 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。

**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）实验：测试ReduceTask多少合适**

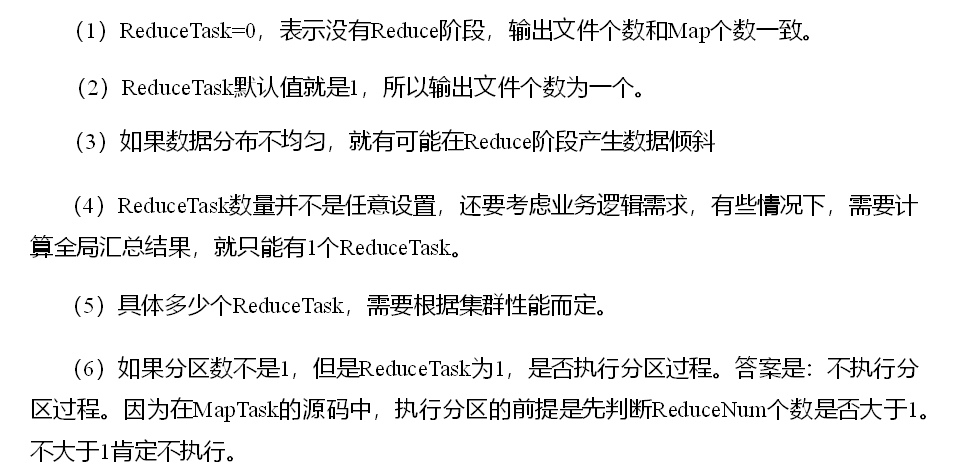
（1）实验环境：1个Master节点，16个Slave节点：CPU:8GHZ，内存: 2G

（2）实验结论：

表 改变ReduceTask （数据量为1GB）

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| MapTask =16 | | | | | | | | | | |
| ReduceTask | 1 | 5 | 10 | 15 | 16 | 20 | 25 | 30 | 45 | 60 |
| 总时间 | 892 | 146 | 110 | 92 | 88 | 100 | 128 | 101 | 145 | 104 |

**3）注意事项**

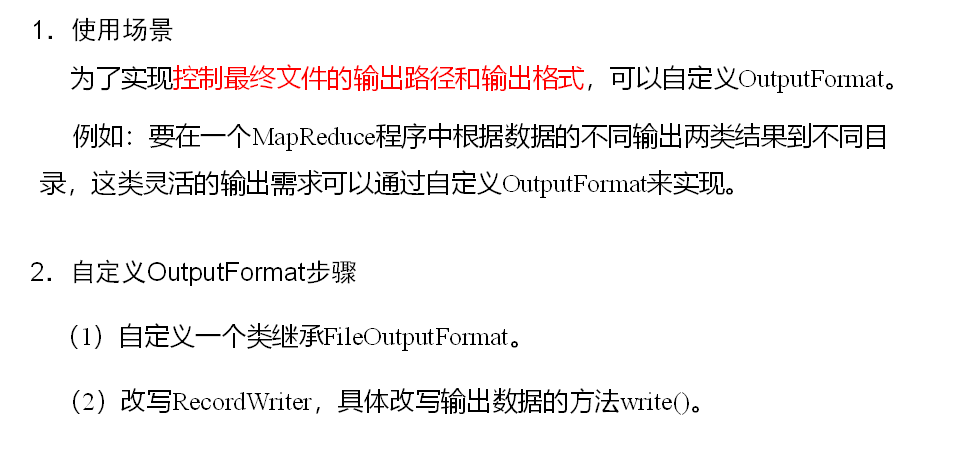


## 3.6 OutputFormat数据输出

### 3.6.1 OutputFormat接口实现类

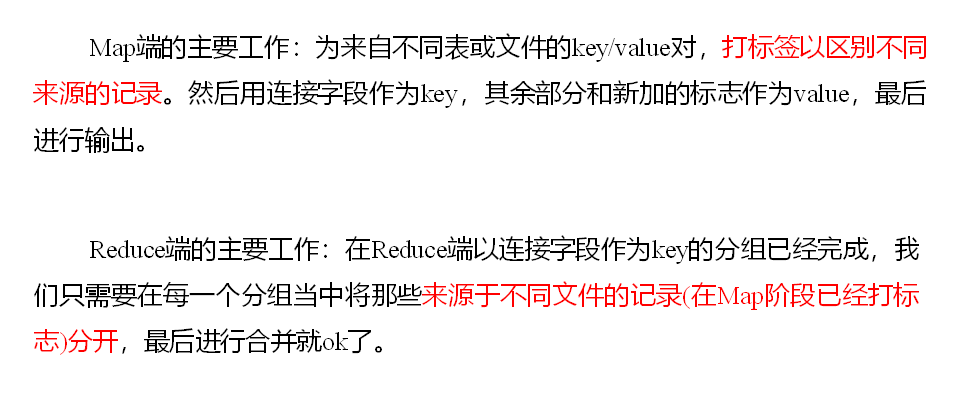


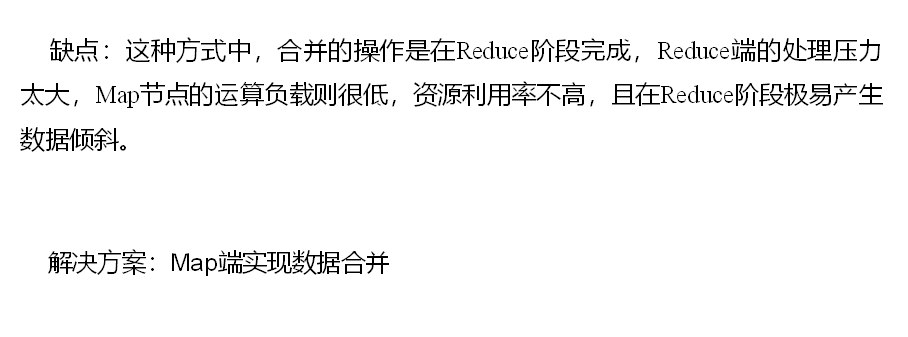
### 3.6.2 自定义OutputFormat



## 3.7 Join多种应用

### 3.7.1 Reduce Join





### 3.7.2 Map Join

**1）使用场景**

Map Join适用于一张表十分小、一张表很大的场景。

**2）优点**

思考：在Reduce端处理过多的表，非常容易产生数据倾斜。怎么办？

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Driver驱动类中加载缓存。

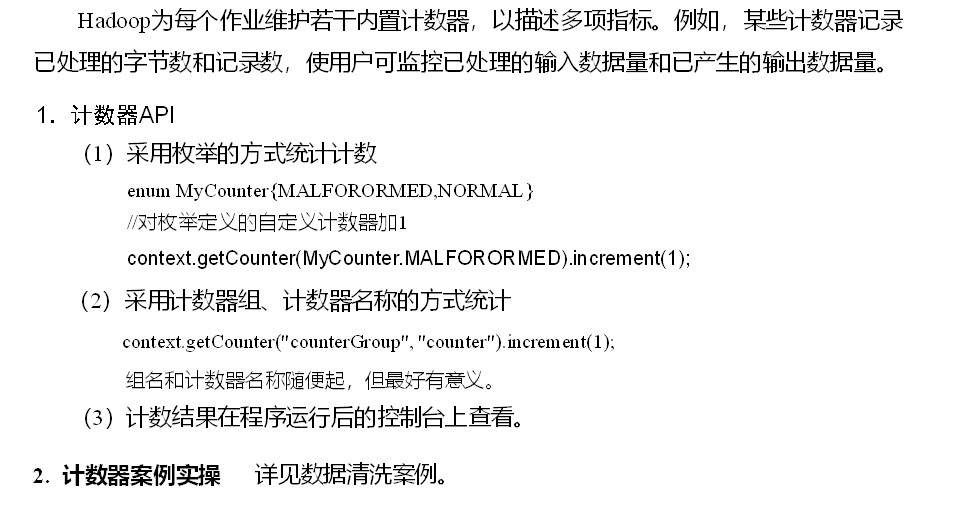
//缓存普通文件到Task运行节点。

job.addCacheFile(new URI("file:///e:/cache/pd.txt"));

//如果是集群运行,需要设置HDFS路径

job.addCacheFile(new URI("hdfs://hadoop102:9820/cache/pd.txt"));

## 3.8 计数器应用



## 3.9 数据清洗（ETL）

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。

**1）需求**

去除日志中字段个数小于等于11的日志。

（1）输入数据

（2）期望输出数据

每行字段长度都大于11。

**2）需求分析**

需要在Map阶段对输入的数据根据规则进行过滤清洗。

**3）实现代码**

（1）编写LogMapper类

package com.user1.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>{

@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 日志合法就直接写出

context.write(value, NullWritable.get());

}

// 2 封装解析日志的方法

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("ETL", "True").increment(1);

return true;

}else {

context.getCounter("ETL", "False").increment(1);

return false;

}

}

}

（2）编写LogDriver类

package com.user1.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 {

// 输入输出路径需要根据自己电脑上实际的输入输出路径设置

args = new String[] { "e:/input/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

boolean b = job.waitForCompletion(true);

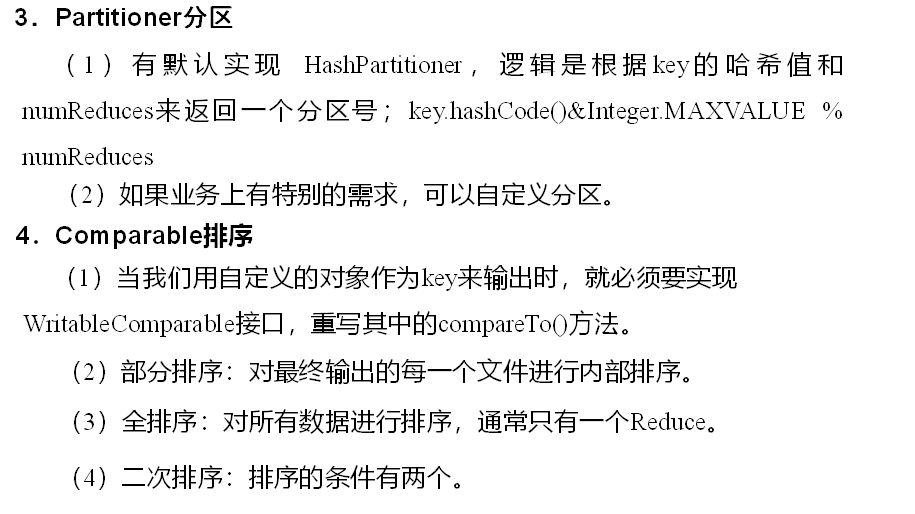
System.exit(b ? 0 : 1);

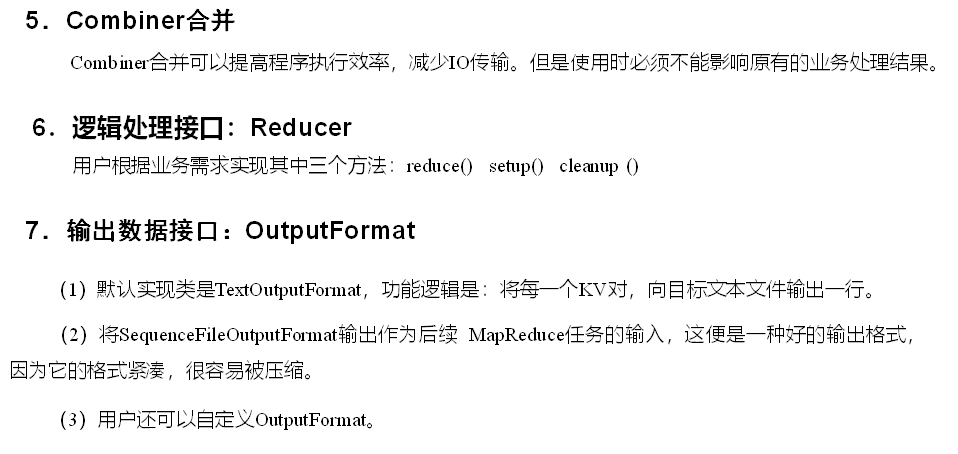
}

}

## 3.10 MapReduce开发总结





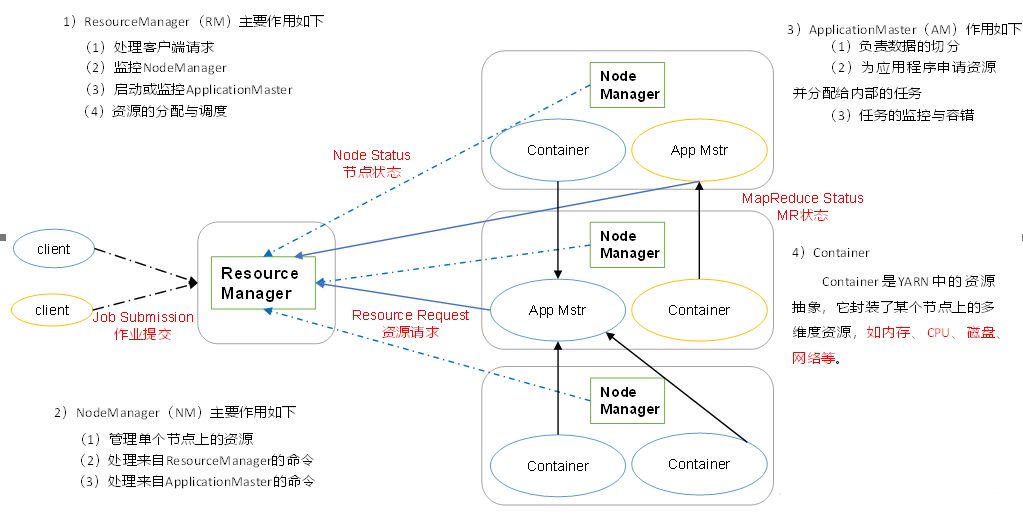


# 第4章 Yarn资源调度器

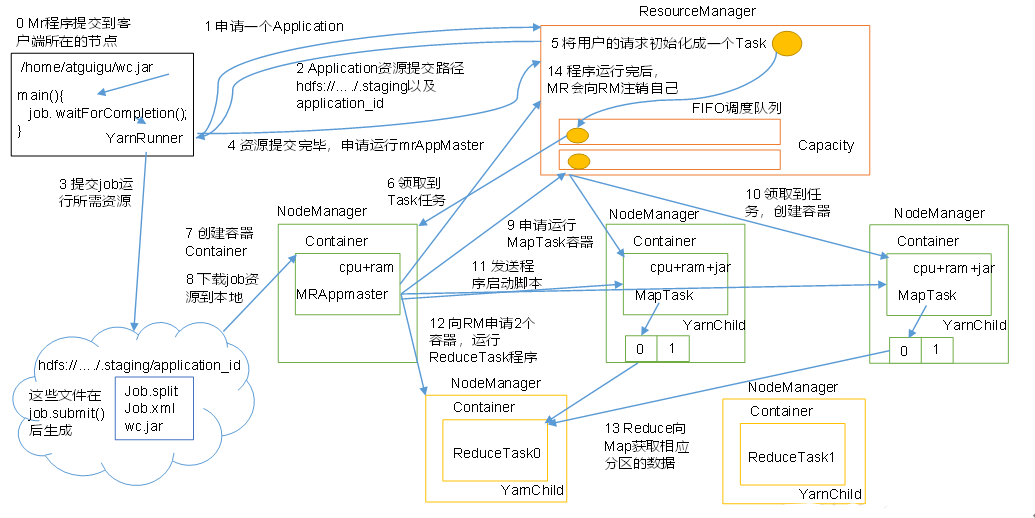
Yarn是一个资源调度平台，负责为运算程序提供服务器运算资源，相当于一个分布式的操作系统平台，而MapReduce等运算程序则相当于运行于操作系统之上的应用程序。

## 4.1 Yarn基本架构

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



## 4.2 Yarn工作机制



（1）MR程序提交到客户端所在的节点。

（2）YarnRunner向ResourceManager申请一个Application。

（3）RM将该应用程序的资源路径返回给YarnRunner。

（4）该程序将运行所需资源提交到HDFS上。

（5）程序资源提交完毕后，申请运行mrAppMaster。

（6）RM将用户的请求初始化成一个Task。

（7）其中一个NodeManager领取到Task任务。

（8）该NodeManager创建容器Container，并产生MRAppmaster。

（9）Container从HDFS上拷贝资源到本地。

（10）MRAppmaster向RM 申请运行MapTask资源。

（11）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。

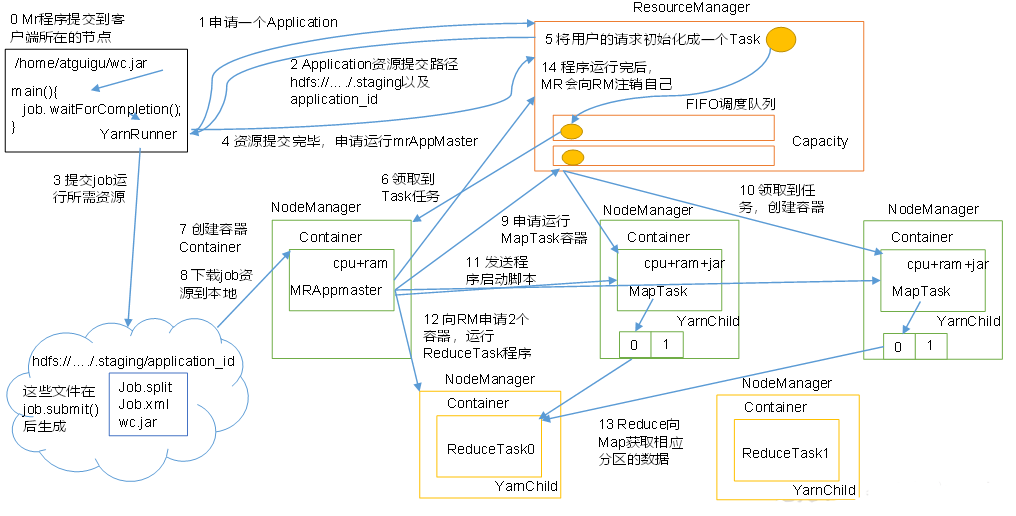
（12）MR向两个接收到任务的NodeManager发送程序启动脚本，这两个NodeManager分别启动MapTask，MapTask对数据分区排序。

（13）MrAppMaster等待所有MapTask运行完毕后，向RM申请容器，运行ReduceTask。

（14）ReduceTask向MapTask获取相应分区的数据。

（15）程序运行完毕后，MR会向RM申请注销自己。

## 4.3 作业提交全过程



作业提交全过程详解

（1）作业提交

第1步：Client调用job.waitForCompletion方法，向整个集群提交MapReduce作业。

第2步：Client向RM申请一个作业id。

第3步：RM给Client返回该job资源的提交路径和作业id。

第4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5步：Client提交完资源后，向RM申请运行MrAppMaster。

（2）作业初始化

第6步：当RM收到Client的请求后，将该job添加到容量调度器中。

第7步：某一个空闲的NM领取到该Job。

第8步：该NM创建Container，并产生MRAppmaster。

第9步：下载Client提交的资源到本地。

（3）任务分配

第10步：MrAppMaster向RM申请运行多个MapTask任务资源。

第11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

第12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13步：MrAppMaster等待所有MapTask运行完毕后，向RM申请容器，运行ReduceTask。

第14步：ReduceTask向MapTask获取相应分区的数据。

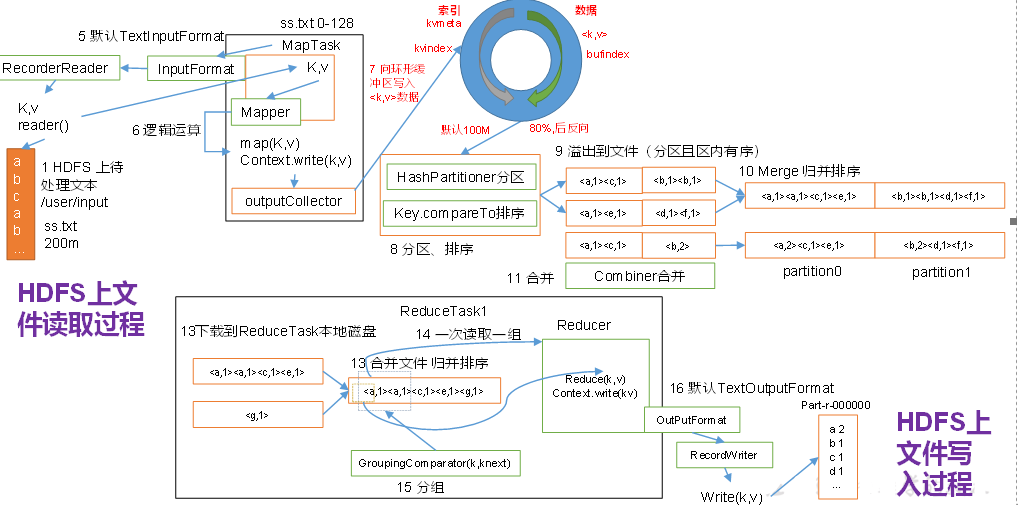
第15步：程序运行完毕后，MR会向RM申请注销自己。

（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秒都会通过调用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。



## 4.4 资源调度器

目前，Hadoop作业调度器主要有三种：FIFO、Capacity Scheduler和Fair Scheduler。Hadoop3.1.3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>

**1）先进先出调度器（FIFO）**

FIFO，即先来先服务，在该调度机制下，所有作业被统一提交到一个队列中，Hadoop按照提交顺序依次运行这些作业。

但随着Hadoop的普及，单个Hadoop集群的用户量越来越大，不同用户提交的应用程序往往具有不同的服务质量要求，典型的应用有以下几种：

批处理作业：这种作业往往耗时较长，对时间完成一般没有严格要求，如数据挖掘、机器学习等方面的应用程序。

交互式作业：这种作业期望能及时返回结果，如SQL查询（Hive）等。

生产性作业：这种作业要求有一定量的资源保证，如统计值计算、垃圾数据分析等。

此外，这些应用程序对硬件资源需求量也是不同的，如过滤、统计类作业一般为CPU密集型作业，而数据挖掘、机器学习作业一般为I/O密集型作业。因此，简单的FIFO调度策略不仅不能满足多样化需求，也不能充分利用硬件资源。

**2）容量调度器（Capacity Scheduler）**



Capacity Scheduler Capacity Scheduler 是Yahoo开发的多用户调度器，它以队列为单位划分资源，每个队列可设定一定比例的资源最低保证和使用上限，同时，每个用户也可设定一定的资源使用上限以防止资源滥用。而当一个队列的资源有剩余时，可暂时将剩余资源共享给其他队列。

总之，Capacity Scheduler 主要有以下几个特点：

①容量保证。管理员可为每个队列设置资源最低保证和资源使用上限，而所有提交到该队列的应用程序共享这些资源。

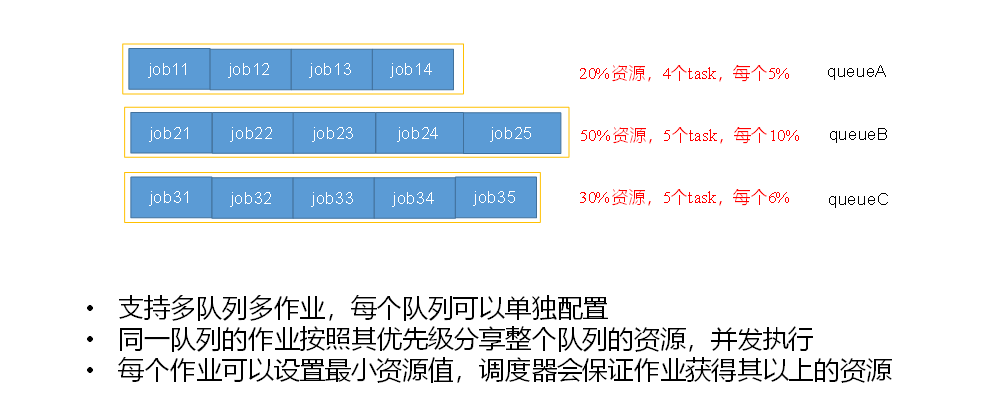
②灵活性，如果一个队列中的资源有剩余，可以暂时共享给那些需要资源的队列，而一旦该队列有新的应用程序提交，则其他队列借调的资源会归还给该队列。这种资源灵活分配的方式可明显提高资源利用率。

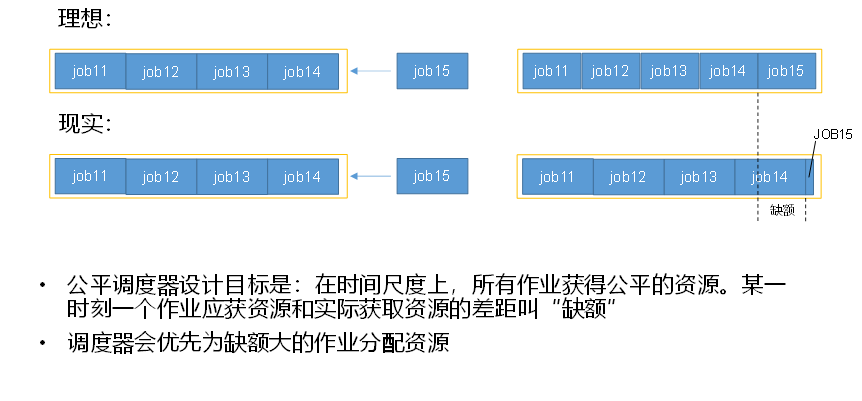
③多重租赁。支持多用户共享集群和多应用程序同时运行。为防止单个应用程序、用户或者队列独占集群中的资源，管理员可为之增加多重约束（比如单个应用程序同时运行的任务数等）。

④安全保证。每个队列有严格的ACL列表规定它的访问用户，每个用户可指定哪些用户允许查看自己应用程序的运行状态或者控制应用程序（比如杀死应用程序）。此外，管理员可指定队列管理员和集群系统管理员。

⑤动态更新配置文件。管理员可根据需要动态修改各种配置参数，以实现在线集群管理。

**3）公平调度器（Fair Scheduler）**





Fair Scheduler Fair Schedulere是Facebook开发的多用户调度器。

公平调度器的目的是让所有的作业随着时间的推移，都能平均地获取等同的共享资源。当有作业提交上来，系统会将空闲的资源分配给新的作业，每个任务大致上会获取平等数量的资源。和传统的调度策略不同的是它会让小的任务在合理的时间完成，同时不会让需要长时间运行的耗费大量资源的任务挨饿！

同Capacity Scheduler类似，它以队列为单位划分资源，每个队列可设定一定比例的资源最低保证和使用上限，同时，每个用户也可设定一定的资源使用上限以防止资源滥用；当一个队列的资源有剩余时，可暂时将剩余资源共享给其他队列。

当然，Fair Scheduler也存在很多与Capacity Scheduler不同之处，这主要体现在以下几个方面：

1. 资源公平共享。在每个队列中，Fair Scheduler 可选择按照FIFO、Fair或DRF策略为应用程序分配资源。其中，

**FIFO策略**

公平调度器每个队列资源分配策略如果选择FIFO的话，就是禁用掉每个队列中的Task共享队列资源，此时公平调度器相当于上面讲过的容量调度器。

**Fair策略**

Fair 策略(默认)是一种基于最大最小公平算法实现的资源多路复用方式，默认情况下，每个队列内部采用该方式分配资源。这意味着，如果一个队列中有两个应用程序同时运行，则每个应用程序可得到1/2的资源；如果三个应用程序同时运行，则每个应用程序可得到1/3的资源。

【扩展：】最大最小公平算法举例：

1. 不加权(关注点是job的个数)：

有一条队列总资源12个, 有4个job，对资源的需求分别是:

job1->1, job2->2 , job3->6, job4->5

第一次算: 12 / 4 = 3

job1: 分3 --> 多2个

job2: 分3 --> 多1个

job3: 分3 --> 差3个

job4: 分3 --> 差2个

第二次算: 3 / 2 = 1.5

job1: 分1

job2: 分2

job3: 分3 --> 差3个 --> 分1.5 --> 最终: 4.5

job4: 分3 --> 差2个 --> 分1.5 --> 最终: 4.5

第n次算: 一直算到没有空闲资源

1. 加权(关注点是job的权重)：

有一条队列总资源16，有4个job

对资源的需求分别是: job1->4 job2->2 job3->10 job4->4

每个job的权重为: job1->5 job2->8 job3->1 job4->2

第一次算: 16 / (5+8+1+2) = 1

job1: 分5 --> 多1

job2: 分8 --> 多6

job3: 分1 --> 少9

job4: 分2 --> 少2

第二次算: 7 / (1+2) = 7/3

job1: 分4

job2: 分2

job3: 分1 --> 分7/3 --> 少

job4: 分2 --> 分14/3(4.66) -->多2.66

第三次算:

job1: 分4

job2: 分2

job3: 分1 --> 分7/3 --> 分2.66

job4: 分4

第n次算: 一直算到没有空闲资源

**DRF策略**

DRF(Dominant Resource Fairness)，我们之前说的资源，都是单一标准，例如只考虑内存(也是yarn默认的情况)。但是很多时候我们资源有很多种，例如内存，CPU，网络带宽等，这样我们很难衡量两个应用应该分配的资源比例。

那么在YARN中，我们用DRF来决定如何调度：假设集群一共有100 CPU和10T 内存，而应用A需要(2 CPU, 300GB)，应用B需要(6 CPU, 100GB)。则两个应用分别需要A(2%CPU, 3%内存)和B(6%CPU, 1%内存)的资源，这就意味着A是内存主导的, B是CPU主导的，针对这种情况，我们可以选择DRF策略对不同应用进行不同资源（CPU和内存）的一个不同比例的限制。

②支持资源抢占。当某个队列中有剩余资源时，调度器会将这些资源共享给其他队列，而当该队列中有新的应用程序提交时，调度器要为它回收资源。为了尽可能降低不必要的计算浪费，调度器采用了先等待再强制回收的策略，即如果等待一段时间后尚有未归还的资源，则会进行资源抢占：从那些超额使用资源的队列中杀死一部分任务，进而释放资源。

yarn.scheduler.fair.preemption=true 通过该配置开启资源抢占。

③提高小应用程序响应时间。由于采用了最大最小公平算法，小作业可以快速获取资源并运行完成