大数据技术之Hadoop（MapReduce）

# **一 MapReduce入门**

## **1.1 MapReduce定义**

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

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

## **1.2 MapReduce优缺点**

### **1.2.1 优点**

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

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

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

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

### **1.2.2 缺点**

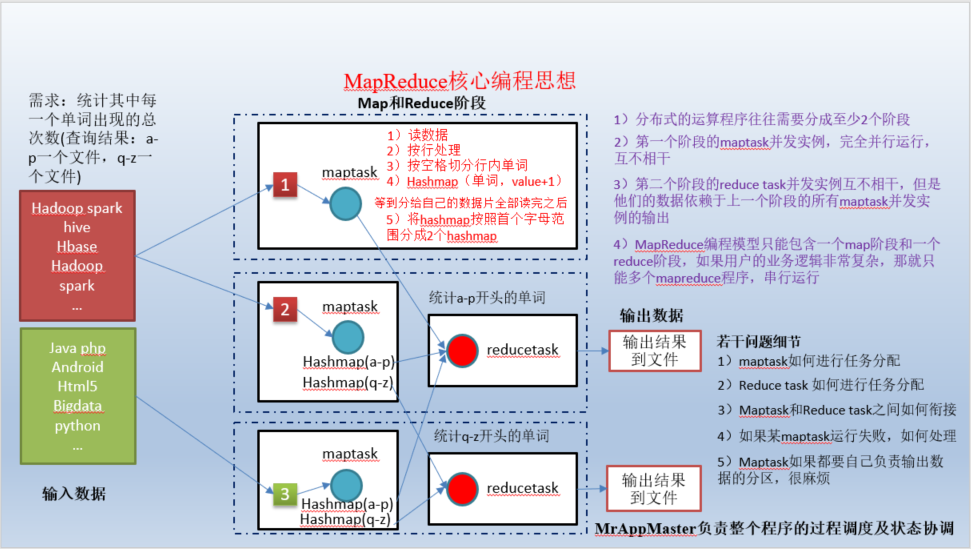
**MapReduce不擅长做实时计算、流式计算、DAG（有向无环图）计算。**

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

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

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

## **1.3 MapReduce核心思想**



1）分布式的运算程序往往需要分成至少2个阶段。

2）第一个阶段的maptask并发实例，完全并行运行，互不相干。

3）第二个阶段的reduce task并发实例互不相干，但是他们的数据依赖于上一个阶段的所有maptask并发实例的输出。

4）MapReduce编程模型只能包含一个map阶段和一个reduce阶段，如果用户的业务逻辑非常复杂，那就只能多个mapreduce程序，串行运行。

## **1.4 MapReduce进程**

一个完整的mapreduce程序在分布式运行时有三类实例进程：

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 MapReduce编程规范**

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

1）Mapper阶段

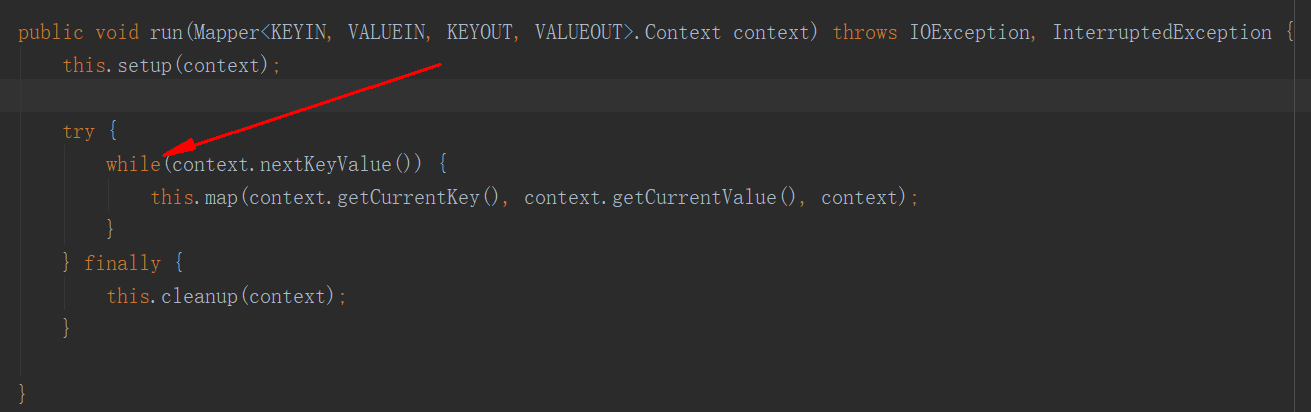
（1）用户自定义的Mapper要继承自己的父类

（2）Mapper的输入数据是KV对的形式（KV的类型可自定义）

（3）Mapper中的业务逻辑写在map()方法中

（4）Mapper的输出数据是KV对的形式（KV的类型可自定义）

（5）map()方法（maptask进程）对每一个<K,V>调用一次



2）Reducer阶段

（1）用户自定义的Reducer要继承自己的父类

（2）Reducer的输入数据类型对应Mapper的输出数据类型，也是KV

（3）Reducer的业务逻辑写在reduce()方法中

（4）Reducetask进程对每一组相同k的<k,v>组调用一次reduce()方法

3）Driver阶段

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

# **二 Hadoop序列化**

## **2.1 为什么要序列化？**

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

## **2.2 什么是序列化？**

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

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

## **2.3 为什么不用Java的序列化？**

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

## **2.4 为什么序列化对Hadoop很重要？**

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

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

1）紧凑：紧凑的格式能让我们充分利用网络带宽，而带宽是数据中心最稀缺的资

2）快速：进程通信形成了分布式系统的骨架，所以需要尽量减少序列化和反序列化的性能开销，这是基本的；

3）可扩展：协议为了满足新的需求变化，所以控制客户端和服务器过程中，需要直接引进相应的协议，这些是新协议，原序列化方式能支持新的协议报文；

4）互操作：能支持不同语言写的客户端和服务端进行交互；

## **2.5 常用数据序列化类型**

常用的数据类型对应的hadoop数据序列化类型

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

## **2.6 自定义bean对象实现序列化接口（Writable）**

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

（1）必须实现Writable接口

（2）反序列化时，需要反射调用空参构造函数，所以必须有空参构造

|  |
| --- |
| **public** FlowBean() {  **super**();  } |

（3）重写序列化方法

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

（4）重写反序列化方法

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

（5）注意反序列化的顺序和序列化的顺序完全一致

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

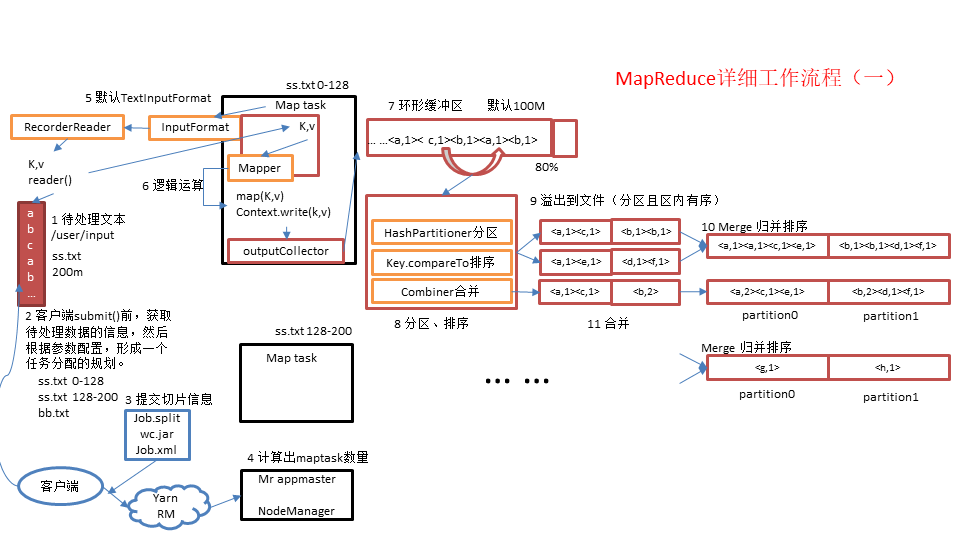
（7）如果需要将自定义的bean放在key中传输，则还需要实现comparable接口，因为mapreduce框中的shuffle过程一定会对key进行排序。

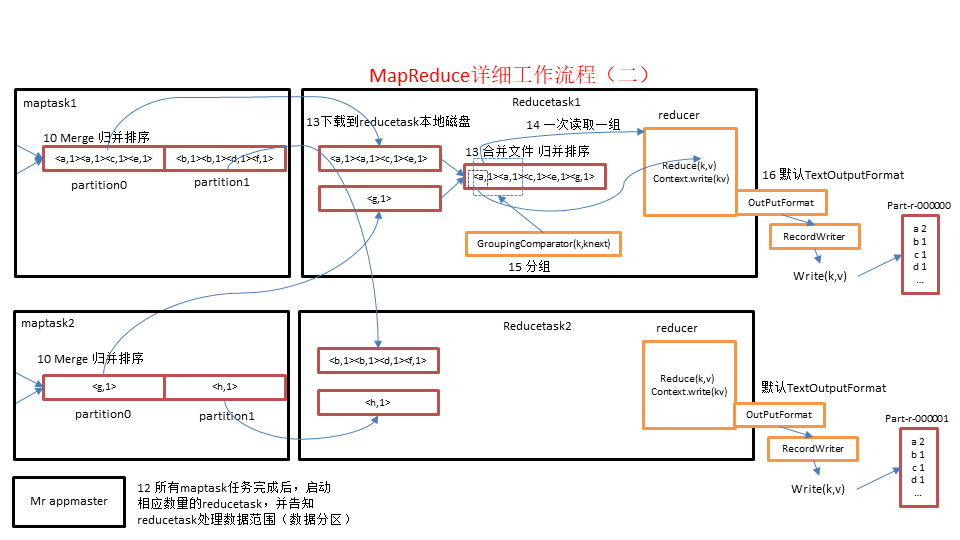
|  |
| --- |
| @Override  **public** **int** compareTo(FlowBean o) {  // 倒序排列，从大到小  **return** **this**.sumFlow > o.getSumFlow() ? -1 : 1;  } |

# **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最全工作流程，但是shuffle过程只是从第7步开始到第15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

Math.max(minSize, Math.min(maxSize, blockSize))

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中默认的切片机制：

（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的280行中，计算切片大小的逻辑：Math.max(minSize, Math.min(maxSize, blockSize));

切片主要由这几个值来运算决定

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

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

因此，默认情况下，切片大小=blocksize。

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 |

注：在看number of splits时，和最大值(MaxSplitSize)有关、总体规律就是和低于最大值是一片、高于最大值1.5倍+，则为两片；高于最大值2倍以上则向下取整，比如文件大小65MB，切片最大值为4MB,那么切片为16个.总体来说，切片差值不超过1个，不影响整体性能

4）案例实操

### **3.2.4 InputFormat接口实现类**

MapReduce任务的输入文件一般是存储在HDFS里面。输入的文件格式包括：基于行的日志文件、二进制格式文件等。这些文件一般会很大，达到数十GB，甚至更大。那么MapReduce是如何读取这些数据的呢？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K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)  (20,Intelligent learning engine)  (49,Learning more convenient)  (75,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。

计算公式：字符个数+空格+偏移量（自占一位）1

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）。

job.setInputFormatClass(KeyValueTextInputFormat.class);

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：

|  |
| --- |
| (Rich，learning form)  (Intelligent，learning engine)  (Learning，more convenient)  (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(20)，如果不整除，切片数=商+1。

以下是一个示例，仍然以上面的4行输入为例。

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

 例如，如果N是2，则每个输入分片包含两行。开启2个maptask。

|  |
| --- |
| (0,Rich learning form)  (19,Intelligent learning engine) |

另一个 mapper 则收到后两行：

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

        这里的键和值与TextInputFormat生成的一样。

### **3.2.5 自定义InputFormat**

1）概述

（1）自定义一个类继承FileInputFormat。

（2）改写RecordReader，实现一次读取一个完整文件封装为KV。

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

2）案例实操

详见7.4小文件处理（自定义InputFormat）。

## **3.3 MapTask工作机制**

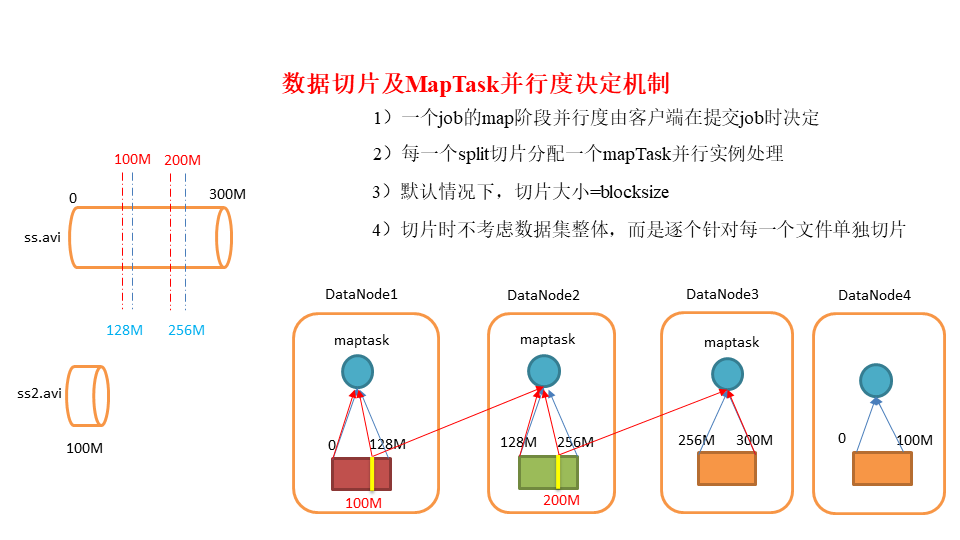
### **3.3.1 并行度决定机制**

1）问题引出

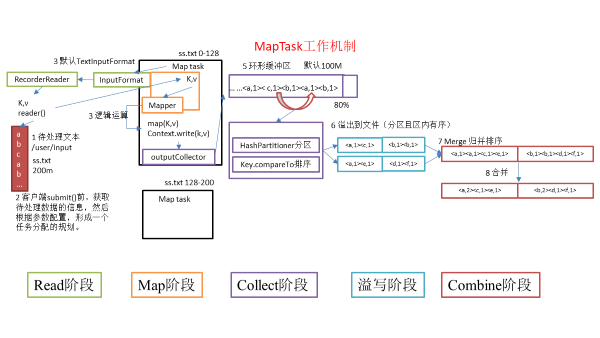
maptask的并行度决定map阶段的任务处理并发度，进而影响到整个job的处理速度。那么，mapTask并行任务是否越多越好呢？

2）MapTask并行度决定机制

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



### **3.3.2 MapTask工作机制**



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

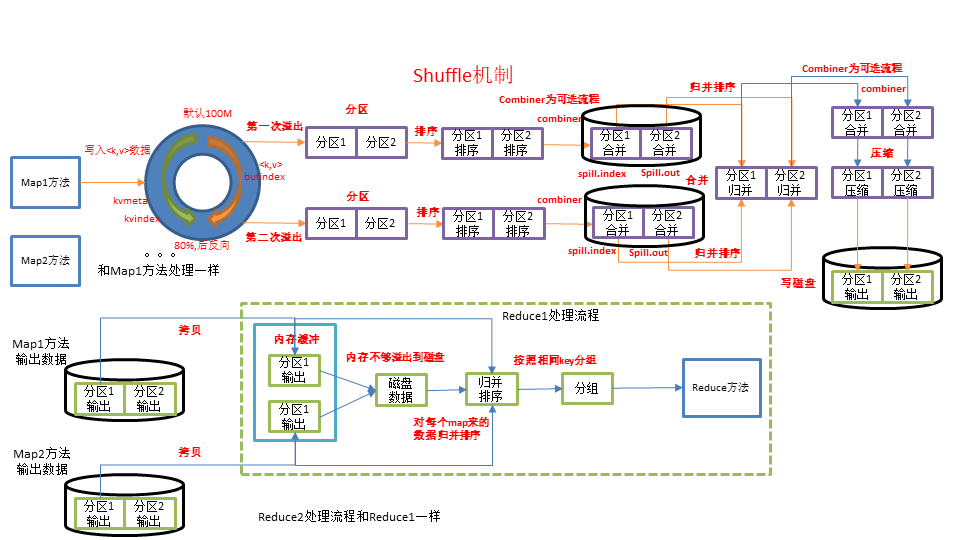
让每个MapTask最终只生成一个数据文件，可避免同时打开大量文件和同时读取大量小文件产生的随机读取带来的开销。

## **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号不同的运营商输出到不同文件中（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。

&是为了不出现负数，也可以用 hashcode()%num + num来代替，但是位运算性能比较高。

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);    partition = 4;    // 2 判断是哪个省  **if** ("136".equals(preNum)) {  partition = 0;  }**else** **if** ("137".equals(preNum)) {  partition = 1;  }**else** **if** ("138".equals(preNum)) {  partition = 2;  }**else** **if** ("139".equals(preNum)) {  partition = 3;  }  **return** partition;  }  } |

（2）在job驱动中，设置自定义partitioner：

|  |
| --- |
| job.setPartitionerClass(CustomPartitioner.class); |

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

|  |
| --- |
| job.setNumReduceTasks(5); |

3）注意：

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

如果1<reduceTask的数量<getPartition的结果数，则有一部分分区数据无处安放，会Exception；

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

例如：假设自定义分区数为5，则

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

（2）job.setNumReduceTasks(2);会报错

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

4）案例实操

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

按照手机号不同的运营商输出到不同文件中

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

### **3.4.3 WritableComparable排序**

排序是MapReduce框架中最重要的操作之一。Map Task和Reduce Task均会对数据（按照key）进行排序。该操作属于Hadoop的默认行为。任何应用程序中的数据均会被排序，而不管逻辑上是否需要。默认排序是按照字典顺序排序，且实现该排序的方法是快速排序。

对于Map Task，它会将处理的结果暂时放到一个缓冲区中，当缓冲区使用率达到一定阈值后，再对缓冲区中的数据进行一次排序，并将这些有序数据写到磁盘上，而当数据处理完毕后，它会对磁盘上所有文件进行一次合并，以将这些文件合并成一个大的有序文件。

对于Reduce Task，它从每个Map Task上远程拷贝相应的数据文件，如果文件大小超过一定阈值，则放到磁盘上，否则放到内存中。如果磁盘上文件数目达到一定阈值，则进行一次合并以生成一个更大文件；如果内存中文件大小或者数目超过一定阈值，则进行一次合并后将数据写到磁盘上。当所有数据拷贝完毕后，Reduce Task统一对内存和磁盘上的所有数据进行一次合并。

每个阶段的默认排序

1）排序的分类：

（1）部分排序：

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

（2）全排序：

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

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

（3）辅助排序：（GroupingComparator分组）

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

（4）二次排序：

在自定义排序过程中，如果compareTo中的判断条件为两个即为二次排序。

2）自定义排序WritableComparable

（1）原理分析

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

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

（2）案例实操

详见7.2.3 需求3：将统计结果按照总流量倒序排序（排序）

详见7.2.4 需求4：不同省份输出文件内部排序（部分排序）

### **3.4.4 GroupingComparator分组（辅助排序）**

1）对reduce阶段的数据根据某一个或几个字段进行分组。

2）案例实操

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

### **3.4.5 Combiner合并**

0）在分布式的架构中，分布式文件系统HDFS，和分布式运算程序编程框架mapreduce。

HDFS:不怕大文件，**怕很多小文件**

mapreduce :**怕数据倾斜**

那么mapreduce是如果解决多个小文件的问题呢？

mapreduce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照**文件**规划切片，不管有多少个小文件，都会是单独的切片，都**会交给一个maptask**，这样，如果有大量的小文件

就会产生大量的maptask，处理效率极端底下

（2）优化策略

最好的方法：在数据处理的最前端（预处理、采集），就将小文件合并成大文件，在上传到HDFS做后续的分析

补救措施：如果已经是大量的小文件在HDFS中了，可以使用另一种inputformat来做切片（CombineFileInputformat），它的切片逻辑跟**TextInputformat**

**注：CombineTextInputFormat是CombineFileInputformat的子类**

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了

//如果不设置InputFormat，它默认的用的是TextInputFormat.class

/\*CombineTextInputFormat为系统自带的组件类

\* setMinInputSplitSize 中的2048是表示n个小文件之和不能大于2048

\* setMaxInputSplitSize 中的4096是 当满足setMinInputSplitSize中的2048情况下 在满足n+1个小文件之和不能大于4096

\*/

job.setInputFormatClass(CombineTextInputFormat.class);

CombineTextInputFormat.setMinInputSplitSize(job, 2048);

CombineTextInputFormat.setMaxInputSplitSize(job, 4096);

1）输入数据：准备5个小文件

2）实现过程

（1）不做任何处理，运行需求1中的wordcount程序，观察切片个数为5

（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, 4\*1024\*1024);// 4m  CombineTextInputFormat.*setMinInputSplitSize*(job, 2\*1024\*1024);// 2m |

注：在看number of splits时，和最大值(MaxSplitSize)有关、总体规律就是和低于最大值是一片、高于最大值1.5倍+，则为两片；高于最大值2倍以上则向下取整，比如文件大小65MB，切片最大值为4MB,那么切片为16个.总体来说，切片差值不超过1个，不影响整体性能

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); |

## **3.5 ReduceTask工作机制**

1）设置ReduceTask并行度（个数）

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

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

2）注意

（1）reducetask=0 ，表示没有reduce阶段，输出文件个数和map个数一致。

（2）reducetask默认值就是1，所以输出文件个数为一个。

（3）如果数据分布不均匀，就有可能在reduce阶段产生数据倾斜

（4）reducetask数量并不是任意设置，还要考虑业务逻辑需求，有些情况下，需要计算全局汇总结果，就只能有1个reducetask。

（5）具体多少个reducetask，需要根据集群性能而定。

（6）如果分区数不是1，但是reducetask为1，是否执行分区过程。答案是：不执行分区过程。因为在maptask的源码中，执行分区的前提是先判断reduceNum个数是否大于1。不大于1肯定不执行。

3）实验：测试reducetask多少合适。

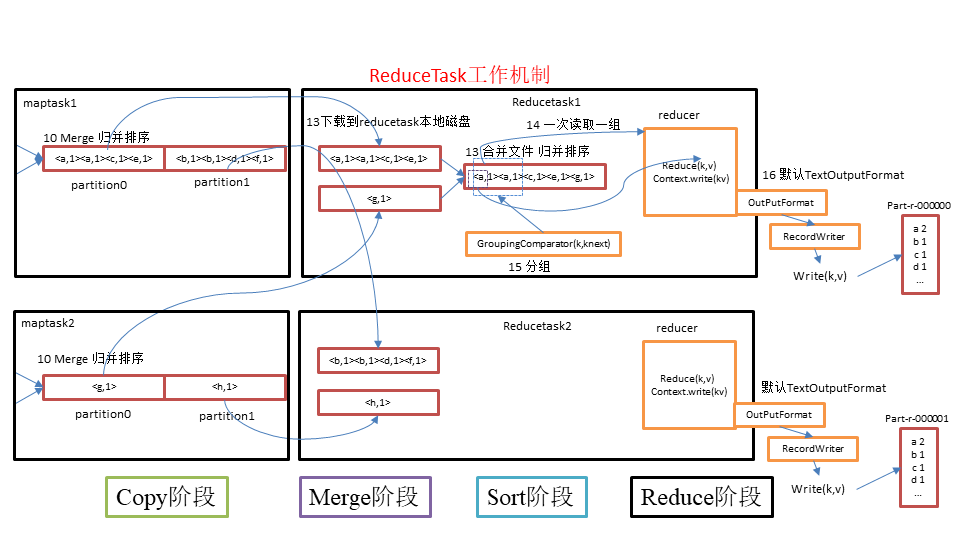
（1）实验环境：1个master节点，16个slave节点：CPU:8GHZ，内存: 2G

（2）实验结论：

表1 改变reduce task （数据量为1GB）

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

4）ReduceTask工作机制



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

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

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

（4）Reduce阶段：reduce()函数将计算结果写到HDFS上。

## **3.6 OutputFormat数据输出**

### **3.6.1 OutputFormat接口实现类**

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

1）文本输出TextOutputFormat

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

2）SequenceFileOutputFormat

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

3）自定义OutputFormat

根据用户需求，自定义实现输出。

### **3.6.2 自定义OutputFormat**

为了实现控制最终文件的输出路径，可以自定义OutputFormat。

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

1）自定义OutputFormat步骤

（1）自定义一个类继承FileOutputFormat。

（2）改写recordwriter，具体改写输出数据的方法write()。

2）实操案例：

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

## **3.7 Join多种应用**

### **3.7.1 Map join（Distributedcache分布式缓存）**

1）使用场景：一张表十分小、一张表很大。

2）解决方案

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

3）具体办法：采用distributedcache

（1）在mapper的setup阶段，将文件读取到缓存集合中。

（2）在驱动函数中加载缓存。

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

4）实操案例：

详见7.6.2需求2：map端表合并（Distributedcache）

### **3.7.2 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）案例实操**

详见7.6.1 需求1：reduce端表合并（数据倾斜）

## **3.8 数据清洗（ETL）**

1）概述

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

2）实操案例

详见7.7 日志清洗（数据清洗）。

## **3.9 计数器应用**

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

1）API

（1）采用枚举的方式统计计数

enum MyCounter{MALFORORMED,NORMAL}

//对枚举定义的自定义计数器加1

context.getCounter(MyCounter.MALFORORMED).increment(1);

（2）采用计数器组、计数器名称的方式统计

context.getCounter("counterGroup", "countera").increment(1);

组名和计数器名称随便起，但最好有意义。

（3）计数结果在程序运行后的控制台上查看。

2）案例实操

详见7.7 日志清洗（数据清洗）。

## **3.10 MapReduce开发总结**

在编写mapreduce程序时，需要考虑的几个方面：

1）输入数据接口：InputFormat

默认使用的实现类是：TextInputFormat

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

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

NlineInputFormat按照指定的行数N来划分切片。

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

用户还可以自定义InputFormat。

2）逻辑处理接口：Mapper

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

3）Partitioner分区

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

如果业务上有特别的需求，可以自定义分区。

4）Comparable排序

当我们用自定义的对象作为key来输出时，就必须要实现WritableComparable接口，重写其中的compareTo()方法。

部分排序：对最终输出的每一个文件进行内部排序。

全排序：对所有数据进行排序，通常只有一个Reduce。

二次排序：排序的条件有两个。

5）Combiner合并

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

6）reduce端分组：Groupingcomparator

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

利用上述这个机制，我们可以实现一个高效的分组取最大值的逻辑。

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

7）逻辑处理接口：Reducer

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

8）输出数据接口：OutputFormat

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

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

用户还可以自定义OutputFormat。

# **四 Hadoop数据压缩**

## **4.1 概述**

压缩技术能够有效减少底层存储系统（HDFS）读写字节数。压缩提高了网络带宽和磁盘空间的效率。在Hadoop下，尤其是数据规模很大和工作负载密集的情况下，使用数据压缩显得非常重要。在这种情况下，I/O操作和网络数据传输要花大量的时间。还有，Shuffle与Merge过程同样也面临着巨大的I/O压力。

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

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

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

注意：压缩特性运用得当能提高性能，但运用不当也可能降低性能。

基本原则：

（1）运算密集型的job，少用压缩

（2）IO密集型的job，多用压缩

## **4.2 MR支持的压缩编码**

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

为了支持多种压缩/解压缩算法，Hadoop引入了编码/解码器，如下表所示

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

压缩性能的比较

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

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

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

## **4.3 压缩方式选择**

### **4.3.1 Gzip压缩**

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

缺点：不支持split。

应用场景：当每个文件压缩之后在140M以内的（1个块大小内），都可以考虑用gzip压缩格式。例如说一天或者一个小时的日志压缩成一个gzip文件，运行mapreduce程序的时候通过多个gzip文件达到并发。hive程序，streaming程序，和java写的mapreduce程序完全和文本处理一样，压缩之后原来的程序不需要做任何修改。

### **4.3.2 Bzip2压缩**

优点：支持split；具有很高的压缩率，比gzip压缩率都高；hadoop本身支持，但不支持native(java和c互操作的API接口)；在linux系统下自带bzip2命令，使用方便。

缺点：压缩/解压速度慢；不支持native。

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

### **4.3.3 Lzo压缩**

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

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

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

### **4.3.4 Snappy压缩**

优点：高速压缩速度和合理的压缩率。

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

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

## **4.4 压缩位置选择**

压缩可以在MapReduce作用的任意阶段启用。

## **4.5 压缩配置参数**

要在Hadoop中启用压缩，可以配置如下参数：

|  |  |  |  |
| --- | --- | --- | --- |
| 参数 | 默认值 | 阶段 | 建议 |
| io.compression.codecs  （在core-site.xml中配置） | org.apache.hadoop.io.compress.DefaultCodec, org.apache.hadoop.io.compress.GzipCodec, org.apache.hadoop.io.compress.BZip2Codec | 输入压缩 | Hadoop使用文件扩展名判断是否支持某种编解码器 |
| mapreduce.map.output.compress（在mapred-site.xml中配置） | false | mapper输出 | 这个参数设为true启用压缩 |
| mapreduce.map.output.compress.codec（在mapred-site.xml中配置） | org.apache.hadoop.io.compress.DefaultCodec | mapper输出 | 使用LZO或snappy编解码器在此阶段压缩数据 |
| mapreduce.output.fileoutputformat.compress（在mapred-site.xml中配置） | false | reducer输出 | 这个参数设为true启用压缩 |
| mapreduce.output.fileoutputformat.compress.codec（在mapred-site.xml中配置） | org.apache.hadoop.io.compress. DefaultCodec | reducer输出 | 使用标准工具或者编解码器，如gzip和bzip2 |
| mapreduce.output.fileoutputformat.compress.type（在mapred-site.xml中配置） | RECORD | reducer输出 | SequenceFile输出使用的压缩类型：NONE和BLOCK |

## **4.6 压缩实战**

压缩案例详见7.10 压缩/解压缩。

# **五 Yarn**

## **5.1 Hadoop1.x和Hadoop2.x架构区别**

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

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

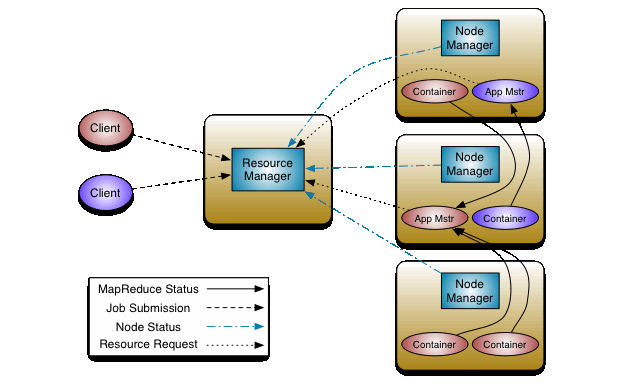
## **5.2 Yarn概述**

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

## **5.3 Yarn基本架构**

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

## **5.4 Yarn工作机制**



### **5.4.1 名词解释：**

1. 资源：在 YARN 的语境下，资源特指计算资源，包括 CPU 和内存。计算机的每个进程都会占用一定的 CPU 和内存，任务需要先向 RM 申请到资源后才能获准在 NM 上启动自己的进程。

2. 队列：YARN 将整个集群的资源划分为队列，每个用户的任务必须提交到指定队列。同时限制每个队列的大小，防止某个用户的任务占用整个集群，影响了其他用户的使用。

3. Vcore & Mem：逻辑 CPU 和逻辑内存，每个 NM 会向 RM 汇报自己有多少 vcore 和内存可用，具体数值由集群管理员配置。比如一台48核，128G内存的机器，可以配置40vcore，120G内存，意为可以对外提供这么多资源。具体数值可能根据实际情况有所调整。每个 NM 的逻辑资源加起来，就是整个集群的总资源量。

4. MinResources & MaxResources：为了使每个队列都能得到一定的资源，同时又不浪费集群的空闲资源，队列的资源设置都是“弹性”的。每个队列都有 min 和 max 两个资源值，min 表示只要需求能达到，集群一定会提供这么多资源；如果资源需求超过了 min 值而同时集群仍有空闲资源，则仍然可以满足；但又限制了资源不能无限申请以免影响其他任务，资源的分配不会超过 max 值。

5. Container：任务申请到资源后在 NM 上启动的进程统称 Container。比如在 MapReduce 中可以是 Mapper 或 Reducer，在 Spark 中可以是 Driver 或 Executor。

### **5.4.2工作机制简化版**

1. 用户使用客户端向 RM 提交一个任务job，同时指定提交到哪个队列和需要多少资源。用户可以通过每个计算引擎的对应参数设置，如果没有特别指定，则使用默认设置。

2. RM 在收到任务提交的请求后，先根据资源和队列是否满足要求选择一个 NM，通知它启动一个特殊的 container，称为 ApplicationMaster（AM），后续流程由它发起。

3. AM 向 RM 注册后根据自己任务的需要，向 RM 申请 container，包括数量、所需资源量、所在位置等因素。

4. 如果队列有足够资源，RM 会将 container 分配给有足够剩余资源的 NM，由 AM 通知 NM 启动 container。

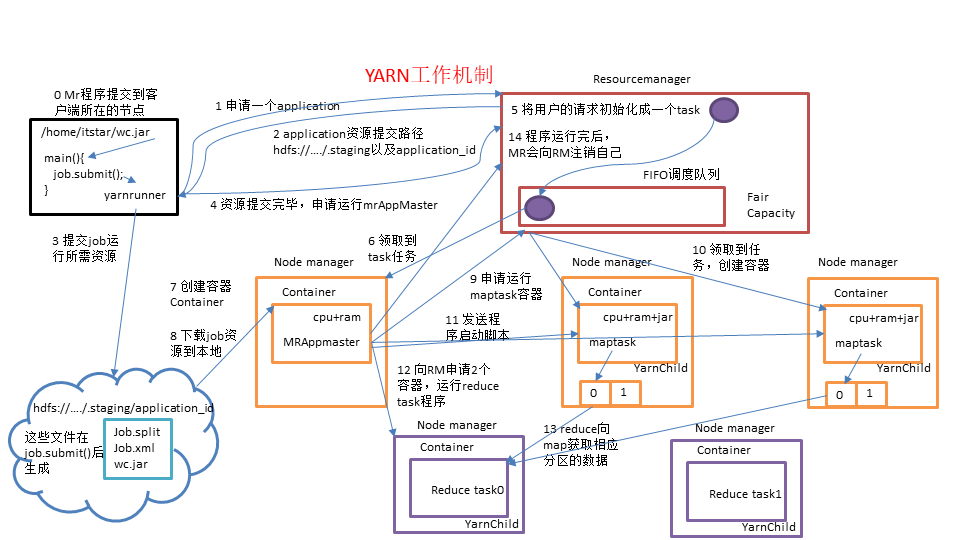
5. container 启动后执行具体的任务，处理分给自己的数据。NM 除了负责启动 container，还负责监控它的资源使用状况以及是否失败退出等工作，如果 container 实际使用的内存超过申请时指定的内存，会将其杀死，保证其他 container 能正常运行。

6. 各个 container 向 AM 汇报自己的进度，都完成后，AM 向 RM 注销任务并退出，RM 通知 NM 杀死对应的 container，任务结束。

**container设置多少资源合适？**

如果 container 内存设置得过低，而实际使用的内存较多，则可能会被 YARN 在运行过程中杀死，无法正常运行。而如果 container 内部线程并发数较多而 vcore 设置的较少，则可能会被分配到一个 load 已经比较高的机器上，导致运行缓慢。所以需要预估单个 container 处理的数据量对应的内存，同时 vcore 数设置的不应该比并发线程数低。

### **5.4.3 Yarn复杂运行机制**



2）工作机制详解

（0）Mr程序提交到客户端所在的节点。

（1）Yarnrunner向Resourcemanager申请一个Application。

（2）rm将该应用程序的资源路径返回给yarnrunner。

（3）该程序将运行所需资源提交到HDFS上。

（4）程序资源提交完毕后，申请运行mrAppMaster。

（5）RM将用户的请求初始化成一个task。

（6）其中一个NodeManager领取到task任务。

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

（8）Container从HDFS上拷贝资源到本地。

（9）MRAppmaster向RM 申请运行maptask资源。

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

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

（12）MrAppMaster等待所有maptask运行完毕后，向RM申请容器，运行reduce task。

（13）reduce task向maptask获取相应分区的数据。

（14）程序运行完毕后，MR会向RM申请注销自己。

## **5.5 作业提交全过程**

作业提交全过程详解

（1）作业提交

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

第1步：client向RM申请一个作业id。

第2步：RM给client返回该job资源的提交路径和作业id。

第3步：client提交jar包、切片信息和配置文件到指定的资源提交路径。

第4步：client提交完资源后，向RM申请运行MrAppMaster。

（2）作业初始化

第5步：当RM收到client的请求后，将该job添加到容量调度器中。

第6步：某一个空闲的NM领取到该job。

第7步：该NM创建Container，并产生MRAppmaster。

第8步：下载client提交的资源到本地。

（3）任务分配

第9步：MrAppMaster向RM申请运行多个maptask任务资源。

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

（4）任务运行

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

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

第13步：reduce task向maptask获取相应分区的数据。

第14步：程序运行完毕后，MR会向RM申请注销自己。

（5）进度和状态更新

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

（6）作业完成

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

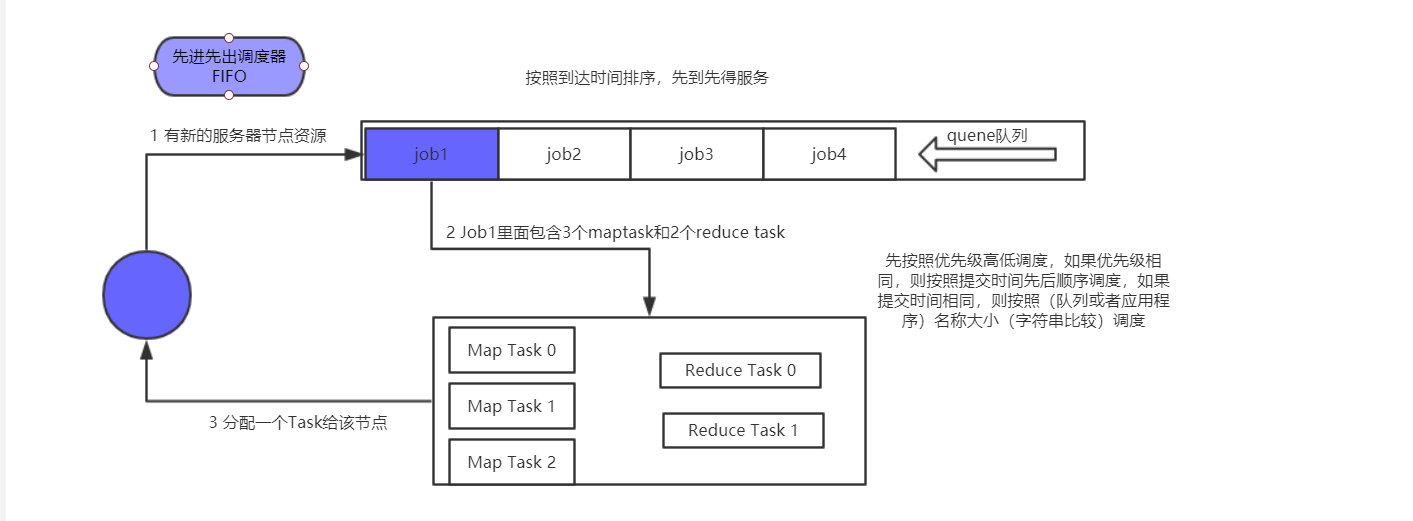
## **5.6 资源调度器**

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

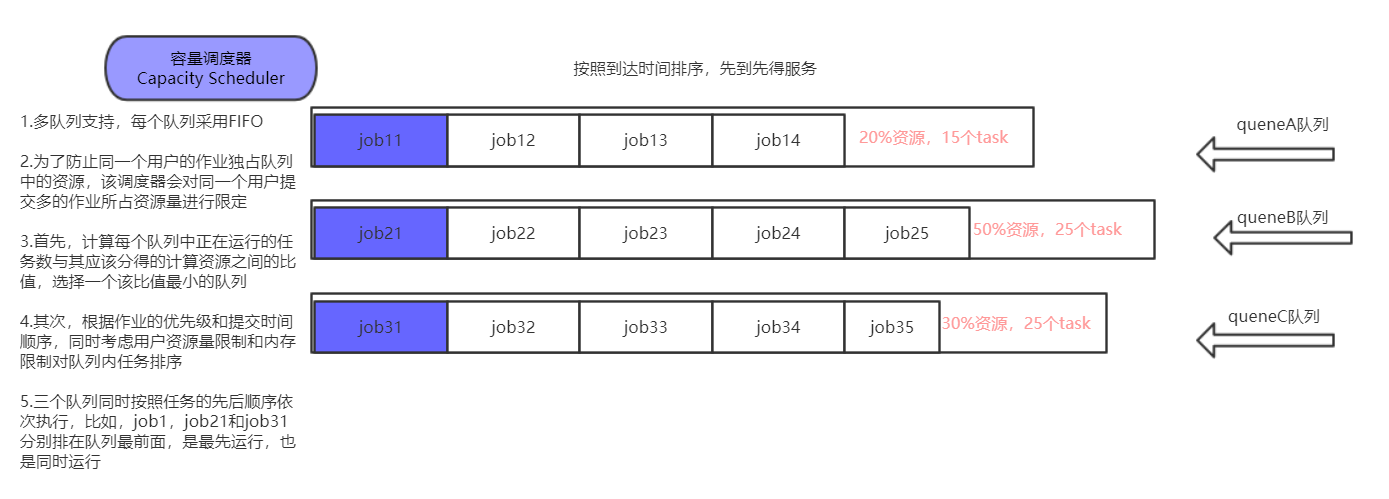
1. **先进先出调度器（FIFO）**



**优点:**调度算法简单，JobTracker（job提交任务后发送得地方）工作负担轻。

**缺点:**忽略了不同作业的需求差异。例如如果类似对海量数据进行统计分析的作业长期占据计算资源，那么在其后提交的交互型作业有可能迟迟得不到处理，从而影响到用户的体验。

1. **容量调度器（Capacity Scheduler）===>Yahoo开发**



1.多队列支持，每个队列采用FIFO

2.为了防止同一个用户的作业独占队列中的资源，该调度器会对同一个用户提交多的作业所占资源量进行限定

3.首先，计算每个队列中正在运行的任务数与其应该分得的计算资源之间的比值，选择一个该比值最小的队列

4.其次，根据作业的优先级和提交时间顺序，同时考虑用户资源量限制和内存限制对队列内任务排序

5.三个队列同时按照任务的先后顺序依次执行，比如，job11，job21和job31分别排在队列最前面，是最先运行，也是同时运行

该调度**默认情况下不支持优先级**，但是可以在配置文件中开启此选项，如果支持优先级，调度算法就是带有优先级的FIFO。

**不支持优先级抢占**，一旦一个作业开始执行，在执行完之前它的资源不会被高优先级作业所抢占。

对队列中同一用户提交的作业能够获得的资源百分比进行了限制以使同属于一用户的作业不能出现独占资源的情况。

**3）公平调度器（Fair Scheduler）===>Facebook开发**



1.支持多队列多用户，每个队列中的资源量可以配置，同一个队列中的作业公平共享队列中所有资源

2.比如有三个队列A，B，C.每个队列中的job按照优先级分配资源，优先级越高分配的资源越多，但是每个job都分配到资源以确保公平。在资源有限的情况下，每个job理想情况下，获得的计算资源与实际获得的计算资源存在一种差距，这个差距叫做缺额。同一个队列，job的资源缺额越大，越先获得的资源优先执行，作业是按照缺额的高低来先后执行的，而且可以看到上图有多个作业同时运行

## **5.7 任务的推测执行**

推测执行(Speculative Execution)是指在集群环境下运行MapReduce，可能是程序Bug，负载不均或者其他的一些问题，导致在一个JOB下的多个TASK速度不一致，比如有的任务已经完成，但是有些任务可能只跑了10%，根据木桶原理，这些任务将成为整个JOB的短板，如果集群启动了推测执行，这时为了最大限度的提高短板，Hadoop会为该task启动备份任务，让speculative task与原始task同时处理一份数据，哪个先运行完，则将谁的结果作为最终结果，并且在运行完成后Kill掉另外一个任务。

1）作业完成时间取决于最慢的任务完成时间

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

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

2）推测执行机制：

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

3）执行推测任务的前提条件

（1）每个task只能有一个备份任务；

（2）当前job已完成的task必须不小于0.05（5%）

（3）开启推测执行参数设置，mapred-site.xml文件中默认是打开的。

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

4）不能启用推测执行机制情况

（1）任务间存在严重的负载倾斜；

（2）特殊任务，比如任务向数据库中写数据。

# **六 Hadoop企业优化**

## **6.1 MapReduce 跑的慢的原因**

Mapreduce 程序效率的瓶颈在于两点：

1）计算机性能

CPU、内存、磁盘健康、网络

2）I/O 操作优化

（1）数据倾斜

（2）map和reduce数设置不合理

（3）map运行时间太长，导致reduce等待过久

（4）小文件过多

（5）大量的不可分块的超大文件

（6）spilt次数过多

（7）merge次数过多等。

## **6.2 MapReduce优化方法**

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

### **6.2.1 数据输入**

（1）合并小文件：在执行mr任务前将小文件进行合并，大量的小文件会产生大量的map任务，增大map任务装载次数，而任务的装载比较耗时，从而导致mr运行较慢。

（2）采用CombineTextInputFormat来作为输入，解决输入端大量小文件场景。

### **6.2.2 Map阶段**

**1）减少溢写（spill）次数：**通过调整io.sort.mb及sort.spill.percent参数值，增大触发spill的内存上限，减少spill次数，从而减少磁盘IO。

**2）减少合并（merge）次数：**通过调整io.sort.factor参数，增大merge的文件数目，减少merge的次数，从而缩短mr处理时间。

3）在map之后，**不影响业务逻辑前提下，先进行combine处理**，减少 I/O。

### **6.2.3 Reduce阶段**

**1）合理设置map和reduce数**：两个都不能设置太少，也不能设置太多。太少，会导致task等待，延长处理时间；太多，会导致 map、reduce任务间竞争资源，造成处理超时等错误。

**2）设置map、reduce共存：**调整slowstart.completedmaps参数，使map运行到一定程度后，reduce也开始运行，减少reduce的等待时间。

**3）规避使用reduce：**因为reduce在用于连接数据集的时候将会产生大量的网络消耗。

**4）合理设置reduce端的buffer：**默认情况下，数据达到一个阈值的时候，buffer中的数据就会写入磁盘，然后reduce会从磁盘中获得所有的数据。也就是说，buffer和reduce是没有直接关联的，中间多个一个写磁盘->读磁盘的过程，既然有这个弊端，那么就可以通过参数来配置，使得buffer中的一部分数据可以直接输送到reduce，从而减少IO开销：mapred.job.reduce.input.buffer.percent，默认为0.0。当值大于0的时候，会保留指定比例的内存读buffer中的数据直接拿给reduce使用。这样一来，设置buffer需要内存，读取数据需要内存，reduce计算也要内存，所以要根据作业的运行情况进行调整。

### **6.2.4 IO传输**

**1）采用数据压缩的方式**，减少网络IO的的时间。安装使用Snappy和LZO压缩编码器。

**2）使用SequenceFile二进制文件。**

### **6.2.5 数据倾斜问题**

1）数据倾斜现象

数据频率倾斜——某一个区域的数据量要远远大于其他区域。

数据大小倾斜——部分记录的大小远远大于平均值。

2）如何收集倾斜数据

在reduce方法中加入记录map输出键的详细情况的功能。

|  |
| --- |
| public static final String MAX\_VALUES = "skew.maxvalues";  private int maxValueThreshold;    @Override  public void configure(JobConf job) {       maxValueThreshold = job.getInt(MAX\_VALUES, 100);  }  @Override  public void reduce(Text key, Iterator<Text> values,                       OutputCollector<Text, Text> output,                       Reporter reporter) throws IOException {       int i = 0;       while (values.hasNext()) {           values.next();           i++;       }       if (++i > maxValueThreshold) {           log.info("Received " + i + " values for key " + key);       }  } |

3）减少数据倾斜的方法

**方法1：抽样和范围分区**

可以通过对原始数据进行抽样得到的结果集来预设分区边界值。

**方法2：自定义分区**

基于输出键的背景知识进行自定义分区。例如，如果map输出键的单词来源于一本书。且其中某几个专业词汇较多。那么就可以自定义分区将这这些专业词汇发送给固定的一部分reduce实例。而将其他的都发送给剩余的reduce实例。

**方法3：Combine**

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

**方法4：采用Map Join，尽量避免Reduce Join。**

### **6.2.6 常用的调优参数**

1）资源相关参数

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

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

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

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

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

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

2）容错相关参数(mapreduce性能优化)

|  |  |
| --- | --- |
| 配置参数 | 参数说明 |
| mapreduce.map.maxattempts | 每个Map Task最大重试次数，一旦重试参数超过该值，则认为Map Task运行失败，默认值：4。 |
| mapreduce.reduce.maxattempts | 每个Reduce Task最大重试次数，一旦重试参数超过该值，则认为Map Task运行失败，默认值：4。 |
| mapreduce.task.timeout | Task超时时间，经常需要设置的一个参数，该参数表达的意思为：如果一个task在一定时间内没有任何进入，即不会读取新的数据，也没有输出数据，则认为该task处于block状态，可能是卡住了，也许永远会卡主，为了防止因为用户程序永远block住不退出，则强制设置了一个该超时时间（单位毫秒），默认是600000。如果你的程序对每条输入数据的处理时间过长（比如会访问数据库，通过网络拉取数据等），建议将该参数调大，该参数过小常出现的错误提示是“AttemptID:attempt\_14267829456721\_123456\_m\_000224\_0 Timed out after 300 secsContainer killed by the ApplicationMaster.”。 |

## **6.3 HDFS小文件优化方法**

### **6.3.1 HDFS小文件弊端**

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

### **6.3.2 解决方案**

1）Hadoop Archive:

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

2）Sequence file：

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

3）CombineFileInputFormat：

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

4）开启JVM重用

对于大量小文件Job，可以开启JVM重用会减少45%运行时间。

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

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

# **七 MapReduce实战**

## **7.1 WordCount案例**

### **7.1.1 需求1：统计一堆文件中单词出现的个数**

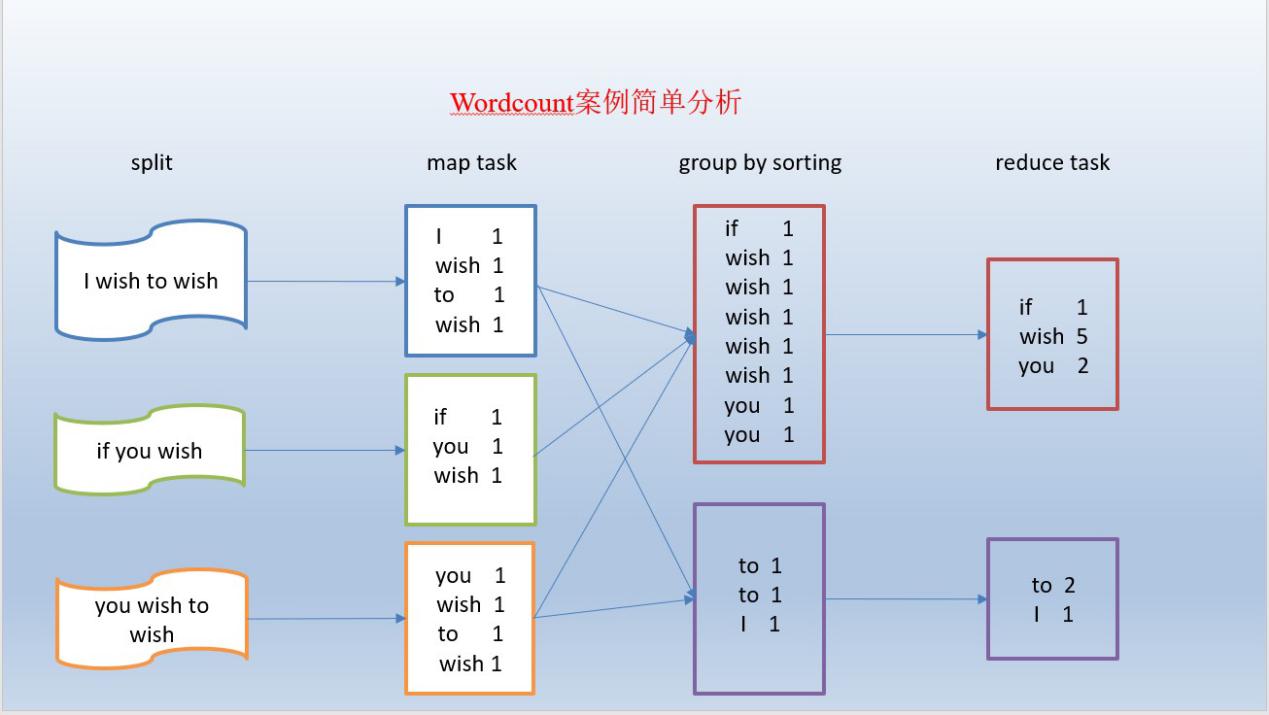
0）需求：在一堆给定的文本文件中统计输出每一个单词出现的总次数

1）数据准备：

2）分析

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





3）编写程序

（1）编写mapper类

|  |
| --- |
| package com.itstar.mapreduce;  import java.io.IOException;  import org.apache.hadoop.io.IntWritable;  import org.apache.hadoop.io.LongWritable;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Mapper;  public class WordCountMapper extends Mapper<LongWritable, Text, Text, IntWritable>{    Text k = new Text();  IntWritable v = new IntWritable(1);    @Override  protected void map(LongWritable key, Text value, Context context)  throws IOException, InterruptedException {    // 1 获取一行  String line = value.toString();    // 2 切割  String[] words = line.split(" ");    // 3 输出  for (String word : words) {    k.set(word);  context.write(k, v);  }  }  } |

（2）编写reducer类

|  |
| --- |
| package com.itstar.mapreduce.wordcount;  import java.io.IOException;  import org.apache.hadoop.io.IntWritable;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Reducer;  public class WordCountReducer extends Reducer<Text, IntWritable, Text, IntWritable>{  @Override  protected void reduce(Text key, Iterable<IntWritable> value,  Context context) throws IOException, InterruptedException {    // 1 累加求和  int sum = 0;  for (IntWritable count : value) {  sum += count.get();  }    // 2 输出  context.write(key, new IntWritable(sum));  }  } |

（3）编写驱动类

|  |
| --- |
| package com.itstar.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 {  String[] args=new String{“输入路径”,”输出路径”};  // 1 获取配置信息  Configuration configuration = new Configuration();  Job job = Job.getInstance(configuration);  // 2 设置jar加载路径  job.setJarByClass(WordcountDriver.class);  // 3 设置map和Reduce类  job.setMapperClass(WordcountMapper.class);  job.setReducerClass(WordcountReducer.class);  // 4 设置map输出  job.setMapOutputKeyClass(Text.class);  job.setMapOutputValueClass(IntWritable.class);  // 5 设置Reduce输出  job.setOutputKeyClass(Text.class);  job.setOutputValueClass(IntWritable.class);    // 6 设置输入和输出路径  FileInputFormat.setInputPaths(job, new Path(args[0]));  FileOutputFormat.setOutputPath(job, new Path(args[1]));  // 7 提交  boolean result = job.waitForCompletion(true);  System.exit(result ? 0 : 1);  }  } |

4）集群上测试

（1）将程序打成jar包，然后拷贝到hadoop集群中。

（2）启动hadoop集群

（3）执行wordcount程序

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

5）本地测试

（1）在windows环境上配置HADOOP\_HOME环境变量。

（2）在idea上运行程序

（3）注意：如果idea打印不出日志，在控制台上只显示

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

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

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

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

0）分析

1）自定义分区

|  |
| --- |
| **package** com.itstar.mapreduce.wordcount;  **import** org.apache.hadoop.io.IntWritable;  **import** org.apache.hadoop.io.Text;  **import** org.apache.hadoop.mapreduce.Partitioner;  **public** **class** WordCountPartitioner **extends** Partitioner<Text, IntWritable>{  @Override  **public** **int** getPartition(Text key, IntWritable value, **int** numPartitions) {    // 1 获取单词key  String firWord = key.toString().substring(0, 1);  **//**String -> int  int result = Integer.valueOf(firWord);  // 2 根据奇数偶数分区  **if** (result % 2 == 0) {  **return** 0;  }**else** {  **return** 1;  }  }  } |

2）在驱动中配置加载分区，设置reducetask个数

|  |
| --- |
| job.setPartitionerClass(WordCountPartitioner.**class**);  job.setNumReduceTasks(2); |

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

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

1）数据准备：

**方案一**

1）增加一个WordcountCombiner类继承Reducer

|  |
| --- |
| **package** com.itstar.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

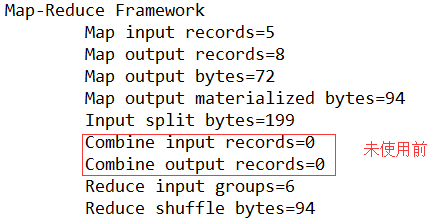
|  |
| --- |
| // 9 指定需要使用combiner，以及用哪个类作为combiner的逻辑  job.setCombinerClass(WordcountCombiner.class); |

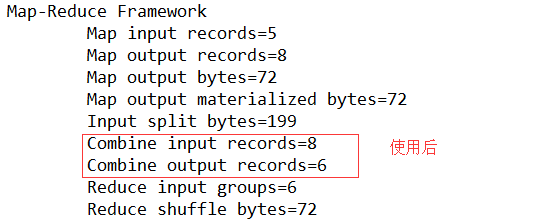
**方案二**

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





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

0）在分布式的架构中，分布式文件系统HDFS，和分布式运算程序编程框架mapreduce。

HDFS:不怕大文件，**怕很多小文件**

mapreduce :**怕数据倾斜**

那么mapreduce是如果解决多个小文件的问题呢？

mapreduce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照文件规划切片，不管有多少个小文件，都会是单独的切片，都**会交给一个maptask**，这样，如果有大量的小文件

就会产生大量的maptask，处理效率极端底下

（2）优化策略

最好的方法：在数据处理的最前端（预处理、采集），就将小文件合并成大文件，在上传到HDFS做后续的分析

补救措施：如果已经是大量的小文件在HDFS中了，可以使用另一种inputformat来做切片（CombineFileInputformat），它的切片逻辑跟**TextInputformat**

**注：combineTextInputFormat是CombineFileInputformat的子类**

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了

//如果不设置InputFormat，它默认的用的是TextInputFormat.class

/\*CombineTextInputFormat为系统自带的组件类

\* setMinInputSplitSize 中的2048是表示n个小文件之和不能大于2048

\* setMaxInputSplitSize 中的4096是 当满足setMinInputSplitSize中的2048情况下 在满足n+1个小文件之和不能大于4096

\*/

job.setInputFormatClass(CombineTextInputFormat.class);

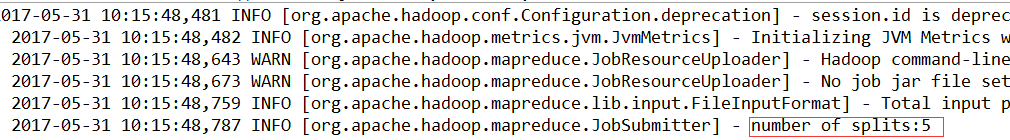
CombineTextInputFormat.setMinInputSplitSize(job, 2048);

CombineTextInputFormat.setMaxInputSplitSize(job, 4096);

1）输入数据：准备5个小文件

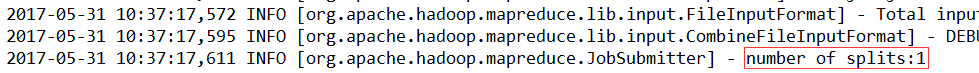
2）实现过程

（1）不做任何处理，运行需求1中的wordcount程序，观察切片个数为5



（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 |



注：

|  |  |  |  |
| --- | --- | --- | --- |
| 文件大小 < MinSplit < MaxSplit | number of splits:1 |  |  |
| MinSplit < 文件大小 < MaxSplit | number of splits:1 |  |  |
| MaxSplit < 文件大小 < 2\*MaxSplit | number of splits:2 |  |  |
| 2 \* MaxSplit < 文件大小 | number of splits:3 |  |  |
| 测试大小 | 最大MB | 文件大小和最大值倍数 | Splits |
| 4.97MB | 3MB | 1.65倍 | 2 |
| 4.1MB | 3MB | 1.36 | 1 |
| 6.51 | 3MB | 2.17 | 3 |

## **7.2 流量汇总案例**

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

1）需求：

统计每一个手机号耗费的总上行流量、下行流量、总流量

**2）数据准备**

输入数据格式：

|  |
| --- |
| **数据格式：**时间戳、电话号码、基站的物理地址、访问网址的ip、网站域名、数据包、接包数、上行/传流量、下行/载流量、响应码 |

输出数据格式

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

3）分析

基本思路：

Map阶段：

（1）读取一行数据，切分字段

（2）抽取手机号、上行流量、下行流量

（3）以手机号为key，bean对象为value输出，即context.write(手机号,bean);

Reduce阶段：

（1）累加上行流量和下行流量得到总流量。

（2）实现自定义的bean来封装流量信息，并将bean作为map输出的key来传输

（3）MR程序在处理数据的过程中会对数据排序(map输出的kv对传输到reduce之前，会排序)，排序的依据是map输出的key

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

然后重写key的compareTo方法。

4）编写mapreduce程序

（1）编写流量统计的bean对象

|  |
| --- |
| import java.io.DataInput;  import java.io.DataOutput;  import java.io.IOException;  import org.apache.hadoop.io.Writable;  // 1 实现writable接口  public class FlowBean implements Writable{  private long upFlow ;  private long downFlow;  private long sumFlow;    //2 反序列化时，需要反射调用空参构造函数，所以必须有  public FlowBean() {  super();  }  public FlowBean(long upFlow, long downFlow) {  super();  this.upFlow = upFlow;  this.downFlow = downFlow;  this.sumFlow = upFlow + downFlow;  }    //3 写序列化方法  @Override  public void write(DataOutput out) throws IOException {  out.writeLong(upFlow);  out.writeLong(downFlow);  out.writeLong(sumFlow);  }    //4 反序列化方法  //5 反序列化方法读顺序必须和写序列化方法的写顺序必须一致  @Override  public void readFields(DataInput in) throws IOException {  this.upFlow = in.readLong();  this.downFlow = in.readLong();  this.sumFlow = in.readLong();  }  // 6 编写toString方法，方便后续打印到文本  @Override  public String toString() {  return upFlow + "\t" + downFlow + "\t" + sumFlow;  }  public long getUpFlow() {  return upFlow;  }  public void setUpFlow(long upFlow) {  this.upFlow = upFlow;  }  public long getDownFlow() {  return downFlow;  }  public void setDownFlow(long downFlow) {  this.downFlow = downFlow;  }  public long getSumFlow() {  return sumFlow;  }  public void setSumFlow(long sumFlow) {  this.sumFlow = sumFlow;  }  } |

（2）编写mapper

|  |
| --- |
| import java.io.IOException;  import org.apache.hadoop.io.LongWritable;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Mapper;  public class FlowCountMapper extends Mapper<LongWritable, Text, Text, FlowBean>{    FlowBean v = new FlowBean();  Text k = new Text();    @Override  protected void map(LongWritable key, Text value, Context context)  throws IOException, InterruptedException {    // 1 获取一行  String line = value.toString();    // 2 切割字段  String[] fields = line.split("\t");    // 3 封装对象  // 取出手机号码  String phoneNum = fields[1];  // 取出上行流量和下行流量  long upFlow = Long.parseLong(fields[fields.length - 3]);  long downFlow = Long.parseLong(fields[fields.length - 2]);    v.set(downFlow, upFlow);    // 4 写出  context.write(new Text(phoneNum), new FlowBean(upFlow, downFlow));  }  } |

（3）编写reducer

|  |
| --- |
| import java.io.IOException;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Reducer;  public class FlowCountReducer extends Reducer<Text, FlowBean, Text, FlowBean> {  @Override  protected void reduce(Text key, Iterable<FlowBean> values, Context context)  throws IOException, InterruptedException {  long sum\_upFlow = 0;  long sum\_downFlow = 0;  // 1 遍历所用bean，将其中的上行流量，下行流量分别累加  for (FlowBean flowBean : values) {  sum\_upFlow += flowBean.getSumFlow();  sum\_downFlow += flowBean.getDownFlow();  }  // 2 封装对象  FlowBean resultBean = new FlowBean(sum\_upFlow, sum\_downFlow);    // 3 写出  context.write(key, resultBean);  }  } |

（4）编写驱动

|  |
| --- |
| import java.io.IOException;  import org.apache.hadoop.conf.Configuration;  import org.apache.hadoop.fs.Path;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Job;  import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;  import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;  public class FlowsumDriver {  public static void main(String[] args) throws IllegalArgumentException, IOException, ClassNotFoundException, InterruptedException {    // 1 获取配置信息，或者job对象实例  Configuration configuration = new Configuration();  Job job = Job.getInstance(configuration);  // 6 指定本程序的jar包所在的本地路径  job.setJarByClass(FlowsumDriver.class);  // 2 指定本业务job要使用的mapper/Reducer业务类  job.setMapperClass(FlowCountMapper.class);  job.setReducerClass(FlowCountReducer.class);  // 3 指定mapper输出数据的kv类型  job.setMapOutputKeyClass(Text.class);  job.setMapOutputValueClass(FlowBean.class);  // 4 指定最终输出的数据的kv类型  job.setOutputKeyClass(Text.class);  job.setOutputValueClass(FlowBean.class);    // 5 指定job的输入原始文件所在目录  FileInputFormat.setInputPaths(job, new Path(args[0]));  FileOutputFormat.setOutputPath(job, new Path(args[1]));  // 7 将job中配置的相关参数，以及job所用的java类所在的jar包， 提交给yarn去运行  boolean result = job.waitForCompletion(true);  System.exit(result ? 0 : 1);  }  } |

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

0）需求：将统计结果按照手机归属地不同省份输出到不同文件中（分区）

1）数据准备

2）分析

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

3）在需求1的基础上，增加一个分区类

|  |
| --- |
| 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);  //注：如果设置的分区数小于下面的分区数，如3、则最后一个分区混数据分区  //注：如何设置的分区数大于下面的分区数，如5，则报错  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.itstar.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 FlowsumDriver {  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(FlowsumDriver.class);  // 2 指定本业务job要使用的mapper/Reducer业务类  job.setMapperClass(FlowCountMapper.class);  job.setReducerClass(FlowCountReducer.class);  // 3 指定mapper输出数据的kv类型  job.setMapOutputKeyClass(Text.class);  job.setMapOutputValueClass(FlowBean.class);  // 4 指定最终输出的数据的kv类型  job.setOutputKeyClass(Text.class);  job.setOutputValueClass(FlowBean.class);  // 8 指定自定义数据分区  job.setPartitionerClass(ProvincePartitioner.class);  // 9 同时指定相应数量的reduce task  job.setNumReduceTasks(5);    // 5 指定job的输入原始文件所在目录  FileInputFormat.setInputPaths(job, new Path(args[0]));  FileOutputFormat.setOutputPath(job, new Path(args[1]));  // 7 将job中配置的相关参数，以及job所用的java类所在的jar包， 提交给yarn去运行  boolean result = job.waitForCompletion(true);  System.exit(result ? 0 : 1);  }  } |

### **7.2.3 需求3：将统计结果按照总流量倒序排序（全排序）**

0）需求

根据需求1产生的结果再次对总流量进行排序。

1）数据准备

2）分析

（1）把程序分两步走，第一步正常统计总流量，第二步再把结果进行排序

（2）context.write(总流量，手机号)

（3）FlowBean实现WritableComparable接口重写compareTo方法

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

3）代码实现

（1）FlowBean对象在在需求1基础上增加了比较功能

|  |
| --- |
| **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.itstar.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 k= 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]);    k.set(upFlow, downFlow);  v.set(phoneNbr);    // 4 输出  context.write(k, v);  }  } |

（3）编写reducer

|  |
| --- |
| package com.itstar.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.itstar.mapreduce.sort;  import java.io.IOException;  import org.apache.hadoop.conf.Configuration;  import org.apache.hadoop.fs.Path;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Job;  import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;  import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;  public class FlowCountSortDriver {  public static void main(String[] args) throws ClassNotFoundException, IOException, InterruptedException {    // 1 获取配置信息，或者job对象实例  Configuration configuration = new Configuration();  Job job = Job.getInstance(configuration);  // 6 指定本程序的jar包所在的本地路径  job.setJarByClass(FlowCountSortDriver.class);  // 2 指定本业务job要使用的mapper/Reducer业务类  job.setMapperClass(FlowCountSortMapper.class);  job.setReducerClass(FlowCountSortReducer.class);  // 3 指定mapper输出数据的kv类型  job.setMapOutputKeyClass(FlowBean.class);  job.setMapOutputValueClass(Text.class);  // 4 指定最终输出的数据的kv类型  job.setOutputKeyClass(Text.class);  job.setOutputValueClass(FlowBean.class);  // 5 指定job的输入原始文件所在目录  FileInputFormat.setInputPaths(job, new Path(args[0]));  FileOutputFormat.setOutputPath(job, new Path(args[1]));    // 7 将job中配置的相关参数，以及job所用的java类所在的jar包， 提交给yarn去运行  boolean result = job.waitForCompletion(true);  System.exit(result ? 0 : 1);  }  } |

### **7.2.4 需求4：不同省份输出文件内部排序（部分排序）**

1）需求

要求每个省份手机号输出的文件中按照总流量内部排序。

2）分析：

基于需求3，增加自定义分区类即可。

3）案例实操

（1）增加自定义分区类

|  |
| --- |
| import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Partitioner;  public class ProvincePartitioner extends Partitioner<FlowBean, Text> {  @Override  public int getPartition(FlowBean key, Text value, int numPartitions) {    // 1 获取手机号码前三位  String preNum = value.toString().substring(0, 3);    int partition = 4;    // 2 根据手机号归属地设置分区  if ("136".equals(preNum)) {  partition = 0;  }else if ("137".equals(preNum)) {  partition = 1;  }else if ("138".equals(preNum)) {  partition = 2;  }else if ("139".equals(preNum)) {  partition = 3;  }  return partition;  }  } |

（2）在驱动类中添加分区类

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

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

1）需求

有如下订单数据

|  |  |  |
| --- | --- | --- |
| 订单id | 商品id | 成交金额 |
| 0000001 | Pdt\_01 | 222.8 |
| 0000001 | Pdt\_06 | 25.8 |
| 0000002 | Pdt\_03 | 522.8 |
| 0000002 | Pdt\_04 | 122.4 |
| 0000002 | Pdt\_05 | 722.4 |
| 0000003 | Pdt\_01 | 222.8 |
| 0000003 | Pdt\_02 | 33.8 |

现在需要求出每一个订单中最贵的商品。

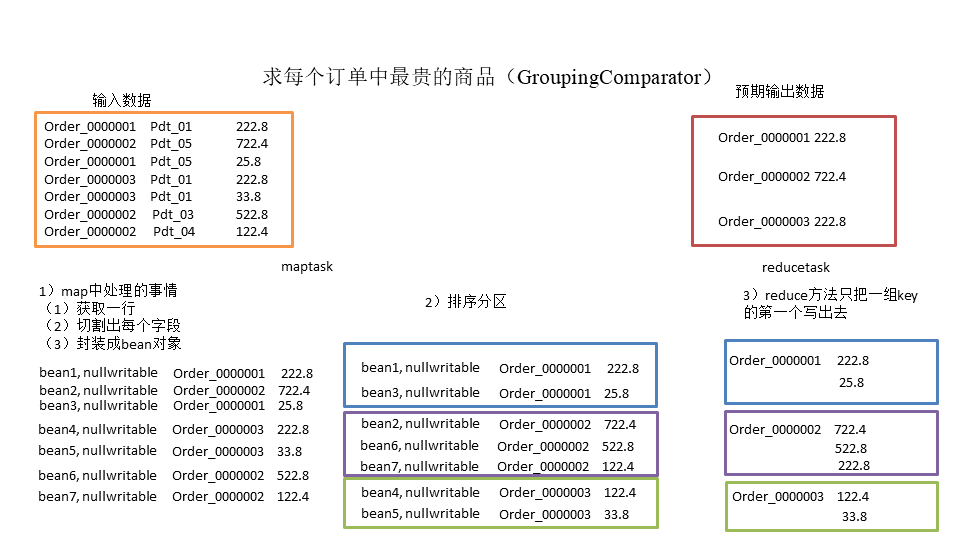
2）输入数据

输出数据预期：

3）分析

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

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



4）代码实现

（1）定义订单信息OrderBean

|  |
| --- |
| import lombok.AllArgsConstructor;  import lombok.Getter;  import lombok.NoArgsConstructor;  import lombok.Setter;  import org.apache.hadoop.io.WritableComparable;  import java.io.DataInput;  import java.io.DataOutput;  import java.io.IOException;  @Getter  @Setter  @AllArgsConstructor  @NoArgsConstructor  public class OrderBean implements WritableComparable<OrderBean> {  // 订单id号  private int order\_id;  // 价格  private double price;  @Override  public String toString() {  return order\_id + "\t" + price;  }  @Override  public int compareTo(OrderBean o) {  int result;  if (this.order\_id > o.getOrder\_id()) {  result = 1;  } else if (this.order\_id < o.getOrder\_id()) {  result = -1;  } else {  result = this.price > o.getPrice() ? -1 : 1;  }  return result;  }  @Override  public void write(DataOutput out) throws IOException {  out.writeInt(order\_id);  out.writeDouble(price);  }  @Override  public void readFields(DataInput in) throws IOException {  in.readInt();  in.readDouble();  }  } |

（2）编写OrderSortMapper

|  |
| --- |
| 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

|  |
| --- |
| **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

|  |
| --- |
| **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

|  |
| --- |
| **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

|  |
| --- |
| **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 {  // 1 获取配置信息  Configuration conf = **new** Configuration();  Job job = Job.*getInstance*(conf);  // 2 设置jar包加载路径  job.setJarByClass(OrderDriver.**class**);  // 3 加载map/reduce类  job.setMapperClass(OrderMapper.**class**);  job.setReducerClass(OrderReducer.**class**);  // 4 设置map输出数据key和value类型  job.setMapOutputKeyClass(OrderBean.**class**);  job.setMapOutputValueClass(NullWritable.**class**);  // 5 设置最终输出数据的key和value类型  job.setOutputKeyClass(OrderBean.**class**);  job.setOutputValueClass(NullWritable.**class**);  // 6 设置输入数据和输出数据路径  FileInputFormat.*setInputPaths*(job, **new** Path(args[0]));  FileOutputFormat.*setOutputPath*(job, **new** Path(args[1]));  // 10 设置reduce端的分组  job.setGroupingComparatorClass(OrderGroupingComparator.**class**);  // 7 设置分区  job.setPartitionerClass(OrderPartitioner.**class**);  // 8 设置reduce个数  job.setNumReduceTasks(3);  // 9 提交  **boolean** result = job.waitForCompletion(**true**);  System.*exit*(result ? 0 : 1);  }  } |

## **7.4 小文件处理案例（自定义InputFormat）**

1）需求

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

2）输入数据

最终预期文件格式：

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InputFromat

|  |
| --- |
| 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 isSplitable(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

|  |
| --- |
| 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

|  |
| --- |
| 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

|  |
| --- |
| 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

|  |
| --- |
| import java.io.IOException;  import org.apache.hadoop.conf.Configuration;  import org.apache.hadoop.fs.Path;  import org.apache.hadoop.io.BytesWritable;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Job;  import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;  import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;  import org.apache.hadoop.mapreduce.lib.output.SequenceFileOutputFormat;  public class SequenceFileDriver {  public static void main(String[] args) throws IOException, ClassNotFoundException, InterruptedException {    args = new String[] { "e:/input/inputinputformat", "e:/output1" };  Configuration conf = new Configuration();  Job job = Job.getInstance(conf);  job.setJarByClass(SequenceFileDriver.class);  job.setMapperClass(SequenceFileMapper.class);  job.setReducerClass(SequenceFileReducer.class);  // 设置输入的inputFormat  job.setInputFormatClass(WholeFileInputformat.class);  // 设置输出的outputFormat  job.setOutputFormatClass(SequenceFileOutputFormat.class);  job.setMapOutputKeyClass(Text.class);  job.setMapOutputValueClass(BytesWritable.class);    job.setOutputKeyClass(Text.class);  job.setOutputValueClass(BytesWritable.class);  FileInputFormat.setInputPaths(job, new Path(args[0]));  FileOutputFormat.setOutputPath(job, new Path(args[1]));  boolean result = job.waitForCompletion(true);  System.exit(result ? 0 : 1);  }  } |

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

1）需求

过滤输入的log日志中是否包含itstar

（1）包含itstar的网站输出到e:/itstar.log

（2）不包含itstar的网站输出到e:/other.log

2）输入数据

输出预期：

3）具体程序：

（1）自定义一个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

|  |
| --- |
| 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 itstarOut = null;  FSDataOutputStream otherOut = null;  public FilterRecordWriter(TaskAttemptContext job) {  // 1 获取文件系统  FileSystem fs;  try {  fs = FileSystem.get(job.getConfiguration());  // 2 创建输出文件路径  Path itstarPath = new Path("e:/itstar.log");  Path otherPath = new Path("e:/other.log");  // 3 创建输出流  itstarOut = fs.create(itstarPath);  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“itstar”输出到不同文件  if (key.toString().contains("itstar")) {  itstarOut.write(key.toString().getBytes());  } else {  otherOut.write(key.toString().getBytes());  }  }  @Override  public void close(TaskAttemptContext context) throws IOException, InterruptedException {  // 关闭资源  if (itstarOut != null) {  itstarOut.close();  }    if (otherOut != null) {  otherOut.close();  }  }  } |

（3）编写FilterMapper

|  |
| --- |
| 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

|  |
| --- |
| 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

|  |
| --- |
| import org.apache.hadoop.conf.Configuration;  import org.apache.hadoop.fs.Path;  import org.apache.hadoop.io.NullWritable;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Job;  import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;  import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;  public class FilterDriver {  public static void main(String[] args) throws Exception {  args = **new** String[] { "e:/input/inputoutputformat", "e:/output2" };  Configuration conf = new Configuration();  Job job = Job.getInstance(conf);  job.setJarByClass(FilterDriver.class);  job.setMapperClass(FilterMapper.class);  job.setReducerClass(FilterReducer.class);  job.setMapOutputKeyClass(Text.class);  job.setMapOutputValueClass(NullWritable.class);    job.setOutputKeyClass(Text.class);  job.setOutputValueClass(NullWritable.class);  // 要将自定义的输出格式组件设置到job中  job.setOutputFormatClass(FilterOutputFormat.class);  FileInputFormat.setInputPaths(job, new Path(args[0]));  // 虽然我们自定义了outputformat，但是因为我们的outputformat继承自fileoutputformat  // 而fileoutputformat要输出一个\_SUCCESS文件，所以，在这还得指定一个输出目录  FileOutputFormat.setOutputPath(job, new Path(args[1]));  boolean result = job.waitForCompletion(true);  System.exit(result ? 0 : 1);  }  } |

## **7.6 MapReduce中多表合并案例**

1）需求：

订单数据表t\_order：

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

商品信息表t\_product

|  |  |
| --- | --- |
| pid | pname |
| 01 | 小米 |
| 02 | 华为 |
| 03 | 格力 |

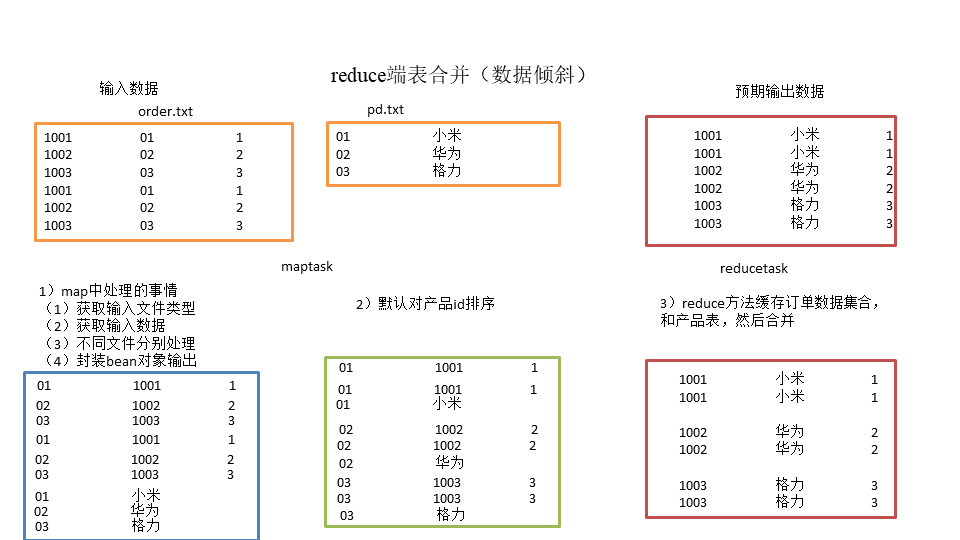
将商品信息表中数据根据商品pid合并到订单数据表中。

最终数据形式：

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

### **7.6.1 需求1：Reduce端表合并（数据倾斜）**

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



1）创建商品和订合并后的bean类

|  |
| --- |
| 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 setPname(String pname) {  this.pname = pname;  }  @Override  public void write(DataOutput out) throws IOException {  out.writeUTF(order\_id);  out.writeUTF(p\_id);  out.writeInt(amount);  out.writeUTF(pname);  out.writeUTF(flag);  }  @Override  public void readFields(DataInput in) throws IOException {  this.order\_id = in.readUTF();  this.p\_id = in.readUTF();  this.amount = in.readInt();  this.pname = in.readUTF();  this.flag = in.readUTF();  }  @Override  public String toString() {  return order\_id + "\t" + pname + "\t" + amount + "\t" ;  }  } |

2）编写TableMapper程序

|  |
| --- |
| **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

|  |
| --- |
| 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

|  |
| --- |
| import org.apache.hadoop.conf.Configuration;  import org.apache.hadoop.fs.Path;  import org.apache.hadoop.io.NullWritable;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Job;  import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;  import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;  public class TableDriver {  public static void main(String[] args) throws Exception {  // 1 获取配置信息，或者job对象实例  Configuration configuration = new Configuration();  Job job = Job.getInstance(configuration);  // 2 指定本程序的jar包所在的本地路径  job.setJarByClass(TableDriver.class);  // 3 指定本业务job要使用的mapper/Reducer业务类  job.setMapperClass(TableMapper.class);  job.setReducerClass(TableReducer.class);  // 4 指定mapper输出数据的kv类型  job.setMapOutputKeyClass(Text.class);  job.setMapOutputValueClass(TableBean.class);  // 5 指定最终输出的数据的kv类型  job.setOutputKeyClass(TableBean.class);  job.setOutputValueClass(NullWritable.class);  // 6 指定job的输入原始文件所在目录  FileInputFormat.setInputPaths(job, new Path(args[0]));  FileOutputFormat.setOutputPath(job, new Path(args[1]));  // 7 将job中配置的相关参数，以及job所用的java类所在的jar包， 提交给yarn去运行  boolean result = job.waitForCompletion(true);  System.exit(result ? 0 : 1);  }  } |

3）运行程序查看结果

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

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

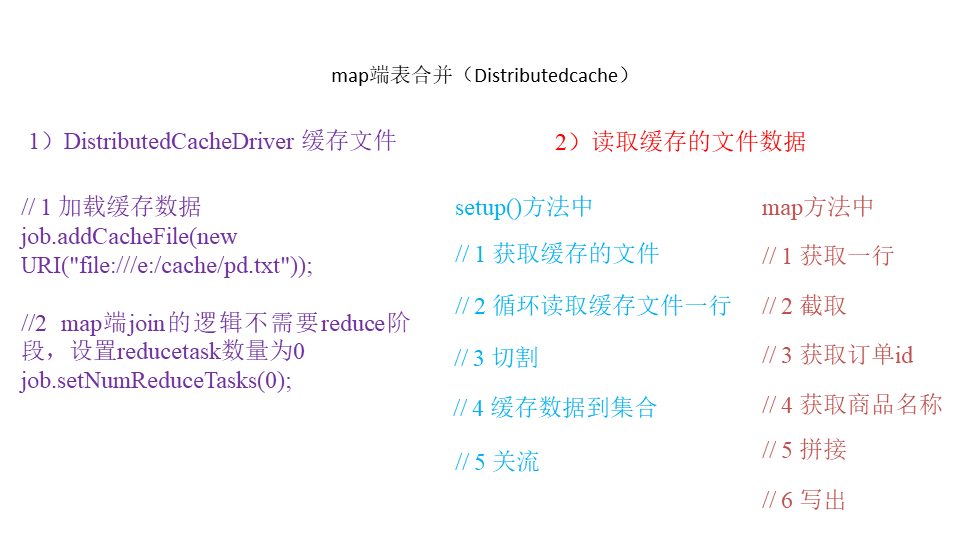
解决方案： map端实现数据合并

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

1）分析

适用于关联表中有小表的情形；

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



2）实操案例

（1）先在驱动模块中添加缓存文件

|  |
| --- |
| import java.net.URI;  import org.apache.hadoop.conf.Configuration;  import org.apache.hadoop.fs.Path;  import org.apache.hadoop.io.NullWritable;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Job;  import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;  import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;  public class DistributedCacheDriver {  public static void main(String[] args) throws Exception {  // 1 获取job信息  Configuration configuration = new Configuration();  Job job = Job.getInstance(configuration);  // 2 设置加载jar包路径  job.setJarByClass(DistributedCacheDriver.class);  // 3 关联map  job.setMapperClass(DistributedCacheMapper.class);    // 4 设置最终输出数据类型  job.setOutputKeyClass(Text.class);  job.setOutputValueClass(NullWritable.class);  // 5 设置输入输出路径  FileInputFormat.setInputPaths(job, new Path(args[0]));  FileOutputFormat.setOutputPath(job, new Path(args[1]));  // 6 加载缓存数据  job.addCacheFile(**new** URI("file:///e:/inputcache/pd.txt"));    // 7 map端join的逻辑不需要reduce阶段，设置reducetask数量为0  job.setNumReduceTasks(0);  // 8 提交  boolean result = job.waitForCompletion(true);  System.exit(result ? 0 : 1);  }  } |

（2）读取缓存的文件数据

|  |
| --- |
| import java.io.BufferedReader;  import java.io.FileInputStream;  import java.io.IOException;  import java.io.InputStreamReader;  import java.util.HashMap;  import java.util.Map;  import org.apache.commons.lang.StringUtils;  import org.apache.hadoop.io.LongWritable;  import org.apache.hadoop.io.NullWritable;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Mapper;  public class DistributedCacheMapper extends Mapper<LongWritable, Text, Text, NullWritable>{  Map<String, String> pdMap = new HashMap<>();    @Override  protected void setup(Mapper<LongWritable, Text, Text, NullWritable>.Context context)  throws IOException, InterruptedException {  // 1 获取缓存的文件  BufferedReader reader = **new** BufferedReader(**new** InputStreamReader(**new** FileInputStream("pd.txt"),"UTF-8"));    String line;  while(StringUtils.isNotEmpty(line = reader.readLine())){  // 2 切割  String[] fields = line.split("\t");    // 3 缓存数据到集合  pdMap.put(fields[0], fields[1]);  }    // 4 关流  reader.close();  }    Text k = new Text();    @Override  protected void map(LongWritable key, Text value, Context context)  throws IOException, InterruptedException {  // 1 获取一行  String line = value.toString();    // 2 截取  String[] fields = line.split("\t");    // 3 获取产品id  String pId = fields[1];    // 4 获取商品名称  String pdName = pdMap.get(pId);    // 5 拼接  k.set(line + "\t"+ pdName);    // 6 写出  context.write(k, NullWritable.get());  }  } |

## **7.7 日志清洗案例**

### **7.7.1 简单解析版**

1）需求：

去除日志中按照空格切分，字段长度小于等于11的日志。

2）输入数据

3）实现代码：

（1）编写LogMapper

|  |
| --- |
| **package** com.itstar.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.itstar.mapreduce.weblog;  **import** org.apache.hadoop.conf.Configuration;  **import** org.apache.hadoop.fs.Path;  **import** org.apache.hadoop.io.NullWritable;  **import** org.apache.hadoop.io.Text;  **import** org.apache.hadoop.mapreduce.Job;  **import** org.apache.hadoop.mapreduce.lib.input.FileInputFormat;  **import** org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;  **public** **class** LogDriver {  **public** **static** **void** main(String[] args) **throws** Exception {  args = **new** String[] { "e:/input/inputlog", "e:/output1" };  // 1 获取job信息  Configuration conf = **new** Configuration();  Job job = Job.*getInstance*(conf);  // 2 加载jar包  job.setJarByClass(LogDriver.**class**);  // 3 关联map  job.setMapperClass(LogMapper.**class**);  // 4 设置最终输出类型  job.setOutputKeyClass(Text.**class**);  job.setOutputValueClass(NullWritable.**class**);  // 设置reducetask个数为0  job.setNumReduceTasks(0);  // 5 设置输入和输出路径  FileInputFormat.*setInputPaths*(job, **new** Path(args[0]));  FileOutputFormat.*setOutputPath*(job, **new** Path(args[1]));  // 6 提交  job.waitForCompletion(**true**);  }  } |

### **7.7.2 复杂解析版**

1）需求：

对web访问日志中的各字段识别切分

去除日志中不合法的记录（1.字段小于等于11不合法 2.状态码大于等于400）

根据统计需求，生成各类访问请求过滤数据

2）输入数据

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.itstar.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.itstar.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.itstar.mapreduce.log;  **import** org.apache.hadoop.conf.Configuration;  **import** org.apache.hadoop.fs.Path;  **import** org.apache.hadoop.io.NullWritable;  **import** org.apache.hadoop.io.Text;  **import** org.apache.hadoop.mapreduce.Job;  **import** org.apache.hadoop.mapreduce.lib.input.FileInputFormat;  **import** org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;  **public** **class** LogDriver {  **public** **static** **void** main(String[] args) **throws** Exception {  // 1 获取job信息  Configuration conf = **new** Configuration();  Job job = Job.*getInstance*(conf);  // 2 加载jar包  job.setJarByClass(LogDriver.**class**);  // 3 关联map  job.setMapperClass(LogMapper.**class**);  // 4 设置最终输出类型  job.setOutputKeyClass(Text.**class**);  job.setOutputValueClass(NullWritable.**class**);  // 5 设置输入和输出路径  FileInputFormat.*setInputPaths*(job, **new** Path(args[0]));  FileOutputFormat.*setOutputPath*(job, **new** Path(args[1]));  // 6 提交  job.waitForCompletion(**true**);  }  } |

## **7.8 倒排索引案例（多job串联）**

0）需求：有大量的文本（文档、网页），需要建立搜索索引

（1）第一次预期输出结果

|  |
| --- |
| itstar--a.txt 3  itstar--b.txt 2  itstar--c.txt 2  pingping--a.txt 1  pingping--b.txt 3  pingping--c.txt 1  ss--a.txt 2  ss--b.txt 1  ss--c.txt 1 |

（2）第二次预期输出结果

|  |
| --- |
| itstar c.txt-->2 b.txt-->2 a.txt-->3  pingping c.txt-->1 b.txt-->3 a.txt-->1  ss c.txt-->1 b.txt-->1 a.txt-->2 |

1）第一次处理

（1）第一次处理，编写OneIndexMapper

|  |
| --- |
| import java.io.IOException;  import org.apache.hadoop.io.IntWritable;  import org.apache.hadoop.io.LongWritable;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Mapper;  import org.apache.hadoop.mapreduce.lib.input.FileSplit;  public class OneIndexMapper extends Mapper<LongWritable, Text, Text , IntWritable>{    String name;  Text k = new Text();  IntWritable v = new IntWritable();    @Override  protected void setup(Context context)  throws IOException, InterruptedException {  // 获取文件名称  FileSplit split = (FileSplit) context.getInputSplit();    name = split.getPath().getName();  }    @Override  protected void map(LongWritable key, Text value, Context context)  throws IOException, InterruptedException {  // 1 获取1行  String line = value.toString();    // 2 切割  String[] fields = line.split(" ");    for (String word : fields) {  // 3 拼接  k.set(word+"--"+name);  v.set(1);    // 4 写出  context.write(k, v);  }  }  } |

（2）第一次处理，编写OneIndexReducer

|  |
| --- |
| package com.itstar.mapreduce.index;  import java.io.IOException;  import org.apache.hadoop.io.IntWritable;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Reducer;  public class OneIndexReducer extends Reducer<Text, IntWritable, Text, IntWritable>{    @Override  protected void reduce(Text key, Iterable<IntWritable> values,  Context context) throws IOException, InterruptedException {    int count = 0;  // 1 累加求和  for(IntWritable value: values){  count +=value.get();  }    // 2 写出  context.write(key, new IntWritable(count));  }  } |

（3）第一次处理，编写OneIndexDriver

|  |
| --- |
| package com.itstar.mapreduce.index;  import org.apache.hadoop.conf.Configuration;  import org.apache.hadoop.fs.Path;  import org.apache.hadoop.io.IntWritable;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Job;  import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;  import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;  public class OneIndexDriver {  public static void main(String[] args) throws Exception {  args = **new** String[] { "e:/input/inputoneindex", "e:/output5" };  Configuration conf = new Configuration();  Job job = Job.getInstance(conf);  job.setJarByClass(OneIndexDriver.class);  job.setMapperClass(OneIndexMapper.class);  job.setReducerClass(OneIndexReducer.class);  job.setMapOutputKeyClass(Text.class);  job.setMapOutputValueClass(IntWritable.class);    job.setOutputKeyClass(Text.class);  job.setOutputValueClass(IntWritable.class);  FileInputFormat.setInputPaths(job, new Path(args[0]));  FileOutputFormat.setOutputPath(job, new Path(args[1]));  job.waitForCompletion(true);  }  } |

（4）查看第一次输出结果

|  |
| --- |
| itstar--a.txt 3  itstar--b.txt 2  itstar--c.txt 2  pingping--a.txt 1  pingping--b.txt 3  pingping--c.txt 1  ss--a.txt 2  ss--b.txt 1  ss--c.txt 1 |

2）第二次处理

（1）第二次处理，编写TwoIndexMapper

|  |
| --- |
| import java.io.IOException;  import org.apache.hadoop.io.LongWritable;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Mapper;  public class TwoIndexMapper extends Mapper<LongWritable, Text, Text, Text>{  Text k = new Text();  Text v = new Text();    @Override  protected void map(LongWritable key, Text value, Context context)  throws IOException, InterruptedException {    // 1 获取1行数据  String line = value.toString();    // 2用“--”切割  String[] fields = line.split("--");    k.set(fields[0]);  v.set(fields[1]);    // 3 输出数据  context.write(k, v);  }  } |

（2）第二次处理，编写TwoIndexReducer

|  |
| --- |
| import java.io.IOException;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Reducer;  public class TwoIndexReducer extends Reducer<Text, Text, Text, Text> {  @Override  protected void reduce(Text key, Iterable<Text> values, Context context) throws IOException, InterruptedException {  // itstar a.txt 3  // itstar b.txt 2  // itstar c.txt 2  // itstar c.txt-->2 b.txt-->2 a.txt-->3  StringBuilder sb = new StringBuilder();  // 1 拼接  for (Text value : values) {  sb.append(value.toString().replace("\t", "-->") + "\t");  }  // 2 写出  context.write(key, new Text(sb.toString()));  }  } |

（3）第二次处理，编写TwoIndexDriver

|  |
| --- |
| import org.apache.hadoop.conf.Configuration;  import org.apache.hadoop.fs.Path;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Job;  import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;  import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;  public class TwoIndexDriver {  public static void main(String[] args) throws Exception {  args = **new** String[] { "e:/input/inputtwoindex", "e:/output6" };  Configuration config = new Configuration();  Job job = Job.getInstance(config);  job.setJarByClass(TwoIndexDriver.class);  job.setMapperClass(TwoIndexMapper.class);  job.setReducerClass(TwoIndexReducer.class);  job.setMapOutputKeyClass(Text.class);  job.setMapOutputValueClass(Text.class);    job.setOutputKeyClass(Text.class);  job.setOutputValueClass(Text.class);  FileInputFormat.setInputPaths(job, new Path(args[0]));  FileOutputFormat.setOutputPath(job, new Path(args[1]));  boolean result = job.waitForCompletion(true);  System.exit(result?0:1);  }  } |

（4）第二次查看最终结果

|  |
| --- |
| Itstar c.txt-->2 b.txt-->2 a.txt-->3  pingping c.txt-->1 b.txt-->3 a.txt-->1  ss c.txt-->1 b.txt-->1 a.txt-->2 |

## **7.9 找博客共同好友案例**

1）需求：

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

求出哪些人两两之间有共同好友，及他俩的共同好友都有谁？

2）需求分析：

先求出A、B、C、….等是谁的好友

第一次输出结果

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

第二次输出结果

|  |
| --- |
| A-B E C  A-C D F  A-D E F  A-E D B C  A-F O B C D E  A-G F E C D  …. |

3）代码实现：

（1）第一次Mapper

|  |
| --- |
| package com.itstar.mapreduce.friends;  import java.io.IOException;  import org.apache.hadoop.io.LongWritable;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Mapper;  public class OneShareFriendsMapper extends Mapper<LongWritable, Text, Text, Text>{    @Override  protected void map(LongWritable key, Text value, Mapper<LongWritable, Text, Text, Text>.Context context)  throws IOException, InterruptedException {  // 1 获取一行 A:B,C,D,F,E,O  String line = value.toString();    // 2 切割  String[] fields = line.split(":");    // 3 获取person和好友  String person = fields[0];  String[] friends = fields[1].split(",");    // 4写出去  for(String friend: friends){  // 输出 <好友，人>  context.write(new Text(friend), new Text(person));  }  }  } |

（2）第一次Reducer

|  |
| --- |
| package com.itstar.mapreduce.friends;  import java.io.IOException;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Reducer;  public class OneShareFriendsReducer extends Reducer<Text, Text, Text, Text>{    @Override  protected void reduce(Text key, Iterable<Text> values, Context context)  throws IOException, InterruptedException {    StringBuffer sb = new StringBuffer();  //1 拼接  for(Text person: values){  sb.append(person).append(",");  }    //2 写出  context.write(key, new Text(sb.toString()));  }  } |

（3）第一次Driver

|  |
| --- |
| package com.itstar.mapreduce.friends;  import org.apache.hadoop.conf.Configuration;  import org.apache.hadoop.fs.Path;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Job;  import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;  import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;  public class OneShareFriendsDriver {  public static void main(String[] args) throws Exception {  // 1 获取job对象  Configuration configuration = new Configuration();  Job job = Job.getInstance(configuration);    // 2 指定jar包运行的路径  job.setJarByClass(OneShareFriendsDriver.class);  // 3 指定map/reduce使用的类  job.setMapperClass(OneShareFriendsMapper.class);  job.setReducerClass(OneShareFriendsReducer.class);    // 4 指定map输出的数据类型  job.setMapOutputKeyClass(Text.class);  job.setMapOutputValueClass(Text.class);    // 5 指定最终输出的数据类型  job.setOutputKeyClass(Text.class);  job.setOutputValueClass(Text.class);    // 6 指定job的输入原始所在目录  FileInputFormat.setInputPaths(job, new Path(args[0]));  FileOutputFormat.setOutputPath(job, new Path(args[1]));    // 7 提交  boolean result = job.waitForCompletion(true);    System.exit(result?0:1);  }  } |

（4）第二次Mapper

|  |
| --- |
| package com.itstar.mapreduce.friends;  import java.io.IOException;  import java.util.Arrays;  import org.apache.hadoop.io.LongWritable;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Mapper;  public class TwoShareFriendsMapper extends Mapper<LongWritable, Text, Text, Text>{    @Override  protected void map(LongWritable key, Text value, Context context)  throws IOException, InterruptedException {  // A I,K,C,B,G,F,H,O,D,  B,c,d,f,g,h,i,k,o  // 友 人，人，人  String line = value.toString();  String[] friend\_persons = line.split("\t");  String friend = friend\_persons[0];  String[] persons = friend\_persons[1].split(",");  Arrays.sort(persons);  for (int i = 0; i < persons.length - 1; i++) {    for (int j = i + 1; j < persons.length; j++) {  // 发出 <人-人，好友> ，这样，相同的“人-人”对的所有好友就会到同1个reduce中去  context.write(new Text(persons[i] + "-" + persons[j]), new Text(friend));  }  }  }  } |

（5）第二次Reducer

|  |
| --- |
| package com.itstar.mapreduce.friends;  import java.io.IOException;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Reducer;  public class TwoShareFriendsReducer extends Reducer<Text, Text, Text, Text>{    @Override  protected void reduce(Text key, Iterable<Text> values, Context context)  throws IOException, InterruptedException {    StringBuffer sb = new StringBuffer();  for (Text friend : values) {  sb.append(friend).append(" ");  }    context.write(key, new Text(sb.toString()));  }  } |

（6）第二次Driver

|  |
| --- |
| package com.itstar.mapreduce.friends;  import org.apache.hadoop.conf.Configuration;  import org.apache.hadoop.fs.Path;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Job;  import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;  import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;  public class TwoShareFriendsDriver {  public static void main(String[] args) throws Exception {  // 1 获取job对象  Configuration configuration = new Configuration();  Job job = Job.getInstance(configuration);    // 2 指定jar包运行的路径  job.setJarByClass(TwoShareFriendsDriver.class);  // 3 指定map/reduce使用的类  job.setMapperClass(TwoShareFriendsMapper.class);  job.setReducerClass(TwoShareFriendsReducer.class);    // 4 指定map输出的数据类型  job.setMapOutputKeyClass(Text.class);  job.setMapOutputValueClass(Text.class);    // 5 指定最终输出的数据类型  job.setOutputKeyClass(Text.class);  job.setOutputValueClass(Text.class);    // 6 指定job的输入原始所在目录  FileInputFormat.setInputPaths(job, new Path(args[0]));  FileOutputFormat.setOutputPath(job, new Path(args[1]));    // 7 提交  boolean result = job.waitForCompletion(true);  System.exit(result?0:1);  }  } |

一个Driver串联

|  |
| --- |
| public class AllShareFriendsReducer {  public static void main(String[] args) throws IOException {  args = new String[]{"F:\\date\\A\\friends.txt","F:\\date\\A\\FA1","F:\\date\\A\\FA11"};  Configuration conf = new Configuration();  Job job1 = Job.getInstance(conf);  job1.setMapperClass(OneShareFriendsMapper.class);  job1.setReducerClass(OneShareFriendsReducer.class);  job1.setMapOutputKeyClass(Text.class);  job1.setMapOutputValueClass(Text.class);  job1.setOutputKeyClass(Text.class);  job1.setOutputValueClass(Text.class);  FileInputFormat.setInputPaths(job1, new Path(args[0]));  FileOutputFormat.setOutputPath(job1, new Path(args[1]));  Job job2 = Job.getInstance(conf);  job2.setMapperClass(TwoShareFriendsMapper.class);  job2.setReducerClass(TwoShareFriendsReducer.class);  job2.setMapOutputKeyClass(Text.class);  job2.setMapOutputValueClass(Text.class);  job2.setOutputKeyClass(Text.class);  job2.setOutputValueClass(Text.class);  FileInputFormat.setInputPaths(job2, new Path(args[1]));  FileOutputFormat.setOutputPath(job2, new Path(args[2]));  JobControl control = new JobControl("Andy");  ControlledJob ajob = new ControlledJob(job1.getConfiguration());  ControlledJob bjob = new ControlledJob(job2.getConfiguration());  bjob.addDependingJob(ajob);  control.addJob(ajob);  control.addJob(bjob);  Thread thread = new Thread(control);  thread.start();  while(!control.allFinished()){  Thread.sleep(1000);  }  System.exit(0);  }  }  往往会出现job线程还在执行，而main线程已经结束。因此我们需要加上下面这一行代码，通过判断job线程是否执行完毕，来决定是否退出jvm。通常job线程执行时间较长，因此我们让当前线程（main线程）在发现job线程没结束的情况下，稍微等他一秒钟。 |

## **7.10 压缩/解压缩案例**

### **7.10.1 数据流的压缩和解压缩**

CompressionCodec有两个方法可以用于轻松地压缩或解压缩数据。要想对正在被写入一个输出流的数据进行压缩，我们可以使用createOutputStream(OutputStreamout)方法创建一个CompressionOutputStream，将其以压缩格式写入底层的流。相反，要想对从输入流读取而来的数据进行解压缩，则调用createInputStream(InputStreamin)函数，从而获得一个CompressionInputStream，从而从底层的流读取未压缩的数据。

测试一下如下压缩方式：

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

|  |
| --- |
| import java.io.File;  import java.io.FileInputStream;  import java.io.FileNotFoundException;  import java.io.FileOutputStream;  import java.io.IOException;  import org.apache.hadoop.conf.Configuration;  import org.apache.hadoop.fs.Path;  import org.apache.hadoop.io.IOUtils;  import org.apache.hadoop.io.compress.CompressionCodec;  import org.apache.hadoop.io.compress.CompressionCodecFactory;  import org.apache.hadoop.io.compress.CompressionInputStream;  import org.apache.hadoop.io.compress.CompressionOutputStream;  import org.apache.hadoop.util.ReflectionUtils;  public class TestCompress {  public static void main(String[] args) throws Exception {  compress("e:/hello.txt","org.apache.hadoop.io.compress.BZip2Codec");  // decompress("e:/hello.txt.bz2");  }  /\*\*  \* 压缩方法  \*  \* @param fliename 文件路径+文件名  \* @param method 解码器  \*/  private static void compress(String filename, String method) throws Exception {  //创建输入流  FileInputStream fis = new FileInputStream(new File(filename));  //通过反射找到解码器的类  Class codeClass = Class.forName(method);  //通过反射工具类找到解码器对象,需要用到配置conf对象  CompressionCodec codec = (CompressionCodec) ReflectionUtils.newInstance(codeClass, new Configuration());  //创建输出流  FileOutputStream fos = new FileOutputStream(new File(filename + codec.getDefaultExtension()));  //获得解码器的输出对象  CompressionOutputStream cos = codec.createOutputStream(fos);  //流拷贝  IOUtils.copyBytes(fis,cos,5 \* 1024 \* 1024,false);  //关闭流  cos.close();  fos.close();  fis.close();  }  /\*\*  \* 解开压缩  \*  \* @param 文件路径+文件名  \* @param 后缀  \*/  private static void decompress(String filename, String decoded) throws Exception {  //获取factory实例  CompressionCodecFactory factory = new CompressionCodecFactory(new Configuration());  CompressionCodec codec = factory.getCodec(new Path(filename));  if (codec == null) {  System.out.println(filename);  return;  }  //解压缩的输入  CompressionInputStream cis = codec.createInputStream(new FileInputStream(new File(filename)));  //输出流  FileOutputStream fos = new FileOutputStream(new File(filename + "." + decoded));  //流拷贝  IOUtils.copyBytes(cis, fos, 5 \* 1024 \* 1024, false);  cis.close();  fos.close();  } |

### **7.10.2 Map输出端采用压缩**

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

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

|  |
| --- |
| **import** java.io.IOException;  **import** org.apache.hadoop.conf.Configuration;  **import** org.apache.hadoop.fs.Path;  **import** org.apache.hadoop.io.IntWritable;  **import** org.apache.hadoop.io.Text;  **import** org.apache.hadoop.io.compress.BZip2Codec;  **import** org.apache.hadoop.io.compress.CompressionCodec;  **import** org.apache.hadoop.io.compress.GzipCodec;  **import** org.apache.hadoop.mapreduce.Job;  **import** org.apache.hadoop.mapreduce.lib.input.FileInputFormat;  **import** org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;  **public** **class** WordCountDriver {  **public** **static** **void** main(String[] args) **throws** IOException, ClassNotFoundException, InterruptedException {  Configuration configuration = **new** Configuration();  // 开启map端输出压缩  configuration.setBoolean("mapreduce.map.output.compress", **true**);  // 设置map端输出压缩方式  configuration.setClass("mapreduce.map.output.compress.codec", BZip2Codec.**class**, CompressionCodec.**class**);  Job job = Job.*getInstance*(configuration);  job.setJarByClass(WordCountDriver.**class**);  job.setMapperClass(WordCountMapper.**class**);  job.setReducerClass(WordCountReducer.**class**);  job.setMapOutputKeyClass(Text.**class**);  job.setMapOutputValueClass(IntWritable.**class**);  job.setOutputKeyClass(Text.**class**);  job.setOutputValueClass(IntWritable.**class**);  FileInputFormat.*setInputPaths*(job, **new** Path(args[0]));  FileOutputFormat.*setOutputPath*(job, **new** Path(args[1]));  **boolean** result = job.waitForCompletion(**true**);  System.*exit*(result ? 1 : 0);  }  } |

2）Mapper保持不变

|  |
| --- |
| **package** com.itstar.mapreduce.compress;  **import** java.io.IOException;  **import** org.apache.hadoop.io.IntWritable;  **import** org.apache.hadoop.io.LongWritable;  **import** org.apache.hadoop.io.Text;  **import** org.apache.hadoop.mapreduce.Mapper;  **public** **class** WordCountMapper **extends** Mapper<LongWritable, Text, Text, IntWritable>{    @Override  **protected** **void** map(LongWritable key, Text value, Context context)  **throws** IOException, InterruptedException {  // 1 获取一行  String line = value.toString();  // 2 切割  String[] words = line.split(" ");  // 3 循环写出  **for**(String word:words){  context.write(**new** Text(word), **new** IntWritable(1));  }  }  } |

3）Reducer保持不变

|  |
| --- |
| **package** com.itstar.mapreduce.compress;  **import** java.io.IOException;  **import** org.apache.hadoop.io.IntWritable;  **import** org.apache.hadoop.io.Text;  **import** org.apache.hadoop.mapreduce.Reducer;  **public** **class** WordCountReducer **extends** Reducer<Text, IntWritable, Text, IntWritable>{    @Override  **protected** **void** reduce(Text key, Iterable<IntWritable> values,  Context context) **throws** IOException, InterruptedException {    **int** count = 0;  // 1 汇总  **for**(IntWritable value:values){  count += value.get();  }    // 2 输出  context.write(key, **new** IntWritable(count));  }  } |

### **7.10.3 Reduce输出端采用压缩**

基于workcount案例处理

1）修改驱动

|  |
| --- |
| **import** java.io.IOException;  **import** org.apache.hadoop.conf.Configuration;  **import** org.apache.hadoop.fs.Path;  **import** org.apache.hadoop.io.IntWritable;  **import** org.apache.hadoop.io.Text;  **import** org.apache.hadoop.io.compress.BZip2Codec;  **import** org.apache.hadoop.io.compress.DefaultCodec;  **import** org.apache.hadoop.io.compress.GzipCodec;  **import** org.apache.hadoop.io.compress.Lz4Codec;  **import** org.apache.hadoop.io.compress.SnappyCodec;  **import** org.apache.hadoop.mapreduce.Job;  **import** org.apache.hadoop.mapreduce.lib.input.FileInputFormat;  **import** org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;  **public** **class** WordCountDriver {  **public** **static** **void** main(String[] args) **throws** IOException, ClassNotFoundException, InterruptedException {    Configuration configuration = **new** Configuration();    Job job = Job.*getInstance*(configuration);    job.setJarByClass(WordCountDriver.**class**);    job.setMapperClass(WordCountMapper.**class**);  job.setReducerClass(WordCountReducer.**class**);    job.setMapOutputKeyClass(Text.**class**);  job.setMapOutputValueClass(IntWritable.**class**);    job.setOutputKeyClass(Text.**class**);  job.setOutputValueClass(IntWritable.**class**);    FileInputFormat.*setInputPaths*(job, **new** Path(args[0]));  FileOutputFormat.*setOutputPath*(job, **new** Path(args[1]));    // 设置reduce端输出压缩开启  FileOutputFormat.*setCompressOutput*(job, **true**);    // 设置压缩的方式  FileOutputFormat.*setOutputCompressorClass*(job, BZip2Codec.**class**);  // FileOutputFormat.setOutputCompressorClass(job, GzipCodec.class);  // FileOutputFormat.setOutputCompressorClass(job, DefaultCodec.class);    **boolean** result = job.waitForCompletion(**true**);    System.*exit*(result?1:0);  }  } |

1. Mapper和Reducer保持不变

## **7.11 两表Join**

### **未优化版本**

**Bean.java**

|  |
| --- |
| import org.apache.hadoop.io.WritableComparable;  import java.io.DataInput;  import java.io.DataOutput;  import java.io.IOException;  /\*  \* 人员和地址的通用bean  \*/  public class Bean implements WritableComparable<Bean> {  private String userNo = "";  private String userName = "";  private String addreNo = "";  private String addreName = "";  private int flag;  public Bean(Bean bean) {  this.userName = bean.getUserName();  this.userNo = bean.getUserNo();  this.addreName = bean.getAddreName();  this.addreNo = bean.getAddreNo();  this.flag = bean.getFlag();  }  public Bean() {  super();  // TODO Auto-generated constructor stub  }  public Bean(String userNo, String userName, String addreNo,  String addreName, int flag) {  super();  this.userNo = userNo;  this.userName = userName;  this.addreNo = addreNo;  this.addreName = addreName;  this.flag = flag;  }  public String getUserNo() {  return userNo;  }  public void setUserNo(String userNo) {  this.userNo = userNo;  }  public String getUserName() {  return userName;  }  public void setUserName(String userName) {  this.userName = userName;  }  public String getAddreNo() {  return addreNo;  }  public void setAddreNo(String addreNo) {  this.addreNo = addreNo;  }  public String getAddreName() {  return addreName;  }  public void setAddreName(String addreName) {  this.addreName = addreName;  }  public int getFlag() {  return flag;  }  public void setFlag(int flag) {  this.flag = flag;  }  @Override  public void write(DataOutput out) throws IOException {  out.writeUTF(userNo);  out.writeUTF(userName);  out.writeUTF(addreNo);  out.writeUTF(addreName);  out.writeInt(flag);  }  @Override  public void readFields(DataInput in) throws IOException {  this.userNo = in.readUTF();  this.userName = in.readUTF();  this.addreNo = in.readUTF();  this.addreName = in.readUTF();  this.flag = in.readInt();  }  @Override  public int compareTo(Bean arg0) {  // TODO Auto-generated method stub  return 0;  }  @Override  public String toString() {  return "userNo=" + userNo + ", userName=" + userName + ", addreNo="  + addreNo + ", addreName=" + addreName;  }  } |

**PersonAddrMap.java**

|  |
| --- |
| import org.apache.hadoop.io.IntWritable;  import org.apache.hadoop.io.LongWritable;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Mapper;  import java.io.IOException;  public class PersonAddrMap extends Mapper<LongWritable, Text, IntWritable, Bean> {  @Override  protected void map(LongWritable key, Text value,  Mapper<LongWritable, Text, IntWritable, Bean>.Context context)  throws IOException, InterruptedException {  String line = value.toString();  String str[] = line.split(" ");  if (str.length == 2) { //地区信息表  Bean bean = new Bean();  bean.setAddreNo(str[0]);  bean.setAddreName(str[1]);  bean.setFlag(0); // 0表示地区  context.write(new IntWritable(Integer.parseInt(str[0])), bean);  } else { //人员信息表  Bean bean = new Bean();  bean.setUserNo(str[0]);  bean.setUserName(str[1]);  bean.setAddreNo(str[2]);  bean.setFlag(1); // 1表示人员表  context.write(new IntWritable(Integer.parseInt(str[2])), bean);  }  }  } |

**PersonAddreRedu.java**

|  |
| --- |
| import org.apache.hadoop.io.IntWritable;  import org.apache.hadoop.io.NullWritable;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Reducer;  import java.io.IOException;  import java.util.ArrayList;  import java.util.List;  public class PersonAddreRedu extends Reducer<IntWritable, Bean, NullWritable,Text> {  @Override  protected void reduce(IntWritable key, Iterable<Bean> values,  Reducer<IntWritable, Bean, NullWritable, Text>.Context context)  throws IOException, InterruptedException {  Bean Addre = null;  List<Bean> peoples = new ArrayList<Bean>();  /\*  \* 如果values的第一个元素信息就是地址Addre的信息的话,  \* 我们就不再需要一个List来缓存person信息了,values后面的全是人员信息  \* 将减少巨大的内存空间  \*/  /\*  \* partitioner和shuffer的过程：  \* partitioner的主要功能是根据reduce的数量将map输出的结果进行分块,将数据送入到相应的reducer.  \* 所有的partitioner都必须实现partitioner接口并实现getPartition方法,该方法的返回值为int类型，并且取值范围在0~(numOfReducer-1),  \* 从而能将map的输出输入到对应的reducer中,对于某个mapreduce过程,hadoop框架定义了默认的partitioner为HashPartioner,  \* 该partitioner使用key的hashCode来决定将该key输送到哪个reducer;  \* shuffle将每个partitioner输出的结果根据key进行group以及排序,将具有相同key的value构成一个values的迭代器,并根据key进行排序分别调用  \* 开发者定义的reduce方法进行排序,因此mapreducer的所以key必须实现comparable接口的compareto()方法从而能实现两个key对象的比较  \*/  /\*  \* 我们需要自定义key的数据结构(shuffle按照key进行分组)来满足共同addreNo的情况下地址表的更小需求  \*  \*/  for (Bean bean : values) {  if (bean.getFlag() == 0) { // 表示地区表  Addre = new Bean(bean);  } else {  peoples.add(new Bean(bean)); //添加到peoplelist中  }  }  for (Bean peo : peoples) { // 给peoplelist添加地区名字  peo.setAddreName(Addre.getAddreName());  context.write(NullWritable.get(), new Text(peo.toString()));  }  }  } |

**PersonAddreMain.java**

|  |
| --- |
| import org.apache.hadoop.conf.Configuration;  import org.apache.hadoop.fs.Path;  import org.apache.hadoop.io.IntWritable;  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 PersonAddreMain {  public static void main(String[] args) throws Exception {  args = new String[] { "F:\\A\\join\\", "F:\\A\\out" };  Configuration conf = new Configuration();  Job job = Job.getInstance(conf);  job.setJarByClass(PersonAddreMain.class);  job.setMapperClass(PersonAddrMap.class);  job.setMapOutputKeyClass(IntWritable.class);  job.setMapOutputValueClass(Bean.class);  job.setReducerClass(PersonAddreRedu.class);  job.setOutputKeyClass(NullWritable.class);  job.setOutputValueClass(Text.class);  FileInputFormat.addInputPath(job, new Path(args[0]));  FileOutputFormat.setOutputPath(job, new Path(args[1]));  job.waitForCompletion(true);  }  } |

### **已优化版本**

**Bean.java**

|  |
| --- |
| import org.apache.hadoop.io.WritableComparable;  import java.io.DataInput;  import java.io.DataOutput;  import java.io.IOException;  /\*  \* 人员和地址的通用bean  \* 用作map输出的value  \*/  public class Bean implements WritableComparable<Bean> {  private String userNo = " ";  private String userName = " ";  private String addreNo = " ";  private String addreName = " ";  public Bean(Bean bean) {  this.userName = bean.getUserName();  this.userNo = bean.getUserNo();  this.addreName = bean.getAddreName();  this.addreNo = bean.getAddreNo();  }  public Bean() {  super();  // TODO Auto-generated constructor stub  }  public Bean(String userNo, String userName, String addreNo,  String addreName, int flag) {  super();  this.userNo = userNo;  this.userName = userName;  this.addreNo = addreNo;  this.addreName = addreName;  }  public String getUserNo() {  return userNo;  }  public void setUserNo(String userNo) {  this.userNo = userNo;  }  public String getUserName() {  return userName;  }  public void setUserName(String userName) {  this.userName = userName;  }  public String getAddreNo() {  return addreNo;  }  public void setAddreNo(String addreNo) {  this.addreNo = addreNo;  }  public String getAddreName() {  return addreName;  }  public void setAddreName(String addreName) {  this.addreName = addreName;  }  @Override  public void write(DataOutput out) throws IOException {  out.writeUTF(userNo);  out.writeUTF(userName);  out.writeUTF(addreNo);  out.writeUTF(addreName);  }  @Override  public void readFields(DataInput in) throws IOException {  this.userNo = in.readUTF();  this.userName = in.readUTF();  this.addreNo = in.readUTF();  this.addreName = in.readUTF();  }  @Override  public int compareTo(Bean arg0) {  // TODO Auto-generated method stub  return 0;  }  @Override  public String toString() {  return "userNo=" + userNo + ", userName=" + userName + ", addreNo="  + addreNo + ", addreName=" + addreName;  }  } |

**BeanKey.java**

|  |
| --- |
| import org.apache.hadoop.io.WritableComparable;  import java.io.DataInput;  import java.io.DataOutput;  import java.io.IOException;  /\*  \* map输出的key  \*/  public class BeanKey implements WritableComparable<BeanKey> {  private int AddreNo;  private boolean isPrimary; // true：address false：person  public BeanKey(int addreNo, boolean isPrimary) {  super();  this.AddreNo = addreNo;  this.isPrimary = isPrimary;  }  public BeanKey() {  super();  // TODO Auto-generated constructor stub  }  @Override  public void write(DataOutput out) throws IOException {  out.writeInt(AddreNo);  out.writeBoolean(isPrimary);  }  @Override  public void readFields(DataInput in) throws IOException {  this.AddreNo = in.readInt();  this.isPrimary = in.readBoolean();  }  // partitioner执行时调用hashcode()方法和compareTo()方法  // compareTo()方法作为shuffle排序的默认方法  @Override  public int hashCode() {  return this.AddreNo; // 按AddreNo进行分组  }  //用于排序,将相同的AddressNo的地址表和人员表,将地址表放到首位  @Override  public int compareTo(BeanKey o) {  if (this.AddreNo == o.getAddreNo()) { // 如果是同一个AddressNo的数据则判断是Person还是Address表  if (this.isPrimary == o.isPrimary()) { //如果属性相同属于同种类型的表,返回0  return 0;  } else {  return this.isPrimary ? -1 : 1; // true表示Address表 返回更小的值,将排至values队首  }  } else {  return this.AddreNo - o.getAddreNo() > 0 ? 1 : -1; //按AddressNo排序  }  }  public int getAddreNo() {  return AddreNo;  }  public void setAddreNo(int addreNo) {  AddreNo = addreNo;  }  public boolean isPrimary() {  return isPrimary;  }  public void setPrimary(boolean isPrimary) {  this.isPrimary = isPrimary;  }  } |

**PersonAddrMap.java**

|  |
| --- |
| import org.apache.hadoop.io.LongWritable;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Mapper;  import java.io.IOException;  /\*  \* map类使key,value分别进行处理  \*/  public class PersonAddreMap extends Mapper<LongWritable, Text, BeanKey, Bean> {  @Override  protected void map(LongWritable key, Text value,  Mapper<LongWritable, Text, BeanKey, Bean>.Context context)  throws IOException, InterruptedException {  String line = value.toString();  String str[] = line.split(" ");  if (str.length == 2) {  // Addre表  Bean Addre = new Bean();  Addre.setAddreNo(str[0]);  Addre.setAddreName(str[1]);  BeanKey AddreKey = new BeanKey();  AddreKey.setAddreNo(Integer.parseInt(str[0]));  AddreKey.setPrimary(true); // true表示地区表  context.write(AddreKey, Addre);  } else {  // Person表  Bean Person = new Bean();  Person.setUserNo(str[0]);  Person.setUserName(str[1]);  Person.setAddreNo(str[2]);  BeanKey PerKey = new BeanKey();  PerKey.setAddreNo(Integer.parseInt(str[2]));  PerKey.setPrimary(false);// false表示人员表  context.write(PerKey, Person);  }  }  } |

**PersonAddreRedu.java**

|  |
| --- |
| import org.apache.hadoop.io.NullWritable;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Reducer;  import java.io.IOException;  public class PersonAddreReduce extends Reducer<BeanKey, Bean, NullWritable, Text> {  @Override  protected void reduce(BeanKey key, Iterable<Bean> values,  Reducer<BeanKey, Bean, NullWritable, Text>.Context context)  throws IOException, InterruptedException {  Bean Addre = null;  int num = 0;  for (Bean bean : values) {  if (num == 0) {  Addre = new Bean(bean); // Address地址表为values的第一个值  num++;  } else {  // 其余全为person表  // 没有list数组,节省大量内存空间  bean.setAddreName(Addre.getAddreName());  context.write(NullWritable.get(), new Text(bean.toString()));  }  }  }  } |

**PersonAddreRedu.java**

|  |
| --- |
| import org.apache.hadoop.io.NullWritable;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Reducer;  import java.io.IOException;  public class PersonAddreReduce extends Reducer<BeanKey, Bean, NullWritable, Text> {  @Override  protected void reduce(BeanKey key, Iterable<Bean> values,  Reducer<BeanKey, Bean, NullWritable, Text>.Context context)  throws IOException, InterruptedException {  Bean Addre = null;  int num = 0;  for (Bean bean : values) {  if (num == 0) {  Addre = new Bean(bean); // Address地址表为values的第一个值  num++;  } else {  // 其余全为person表  // 没有list数组,节省大量内存空间  bean.setAddreName(Addre.getAddreName());  context.write(NullWritable.get(), new Text(bean.toString()));  }  }  }  } |

**PKFKCompartor.java**

|  |
| --- |
| import org.apache.hadoop.io.WritableComparable;  import org.apache.hadoop.io.WritableComparator;  /\*  \* 实现Group分组  \* shuffle的group过程默认的是使用的key(BeanKey)的compareTo()方法  \* 刚才我们添加的自定义的Key没有办法将具有相同AddressNo的地址和人员放到同一个group中(因为从compareTo()方法中可以看出他们是不相等的)  \* 我们需要的就是自己定义一个groupComparer就可以  \* 实现比较器  \*/  public class PKFKCompartor extends WritableComparator {  protected PKFKCompartor() {  super(BeanKey.class, true);  }  //两个BeanKey进行比较排序  @Override  public int compare(WritableComparable a, WritableComparable b) {  BeanKey a1 = (BeanKey) a;  BeanKey b1 = (BeanKey) b;  if (a1.getAddreNo() == b1.getAddreNo()) {  return 0;  } else {  return a1.getAddreNo() > b1.getAddreNo() ? 1 : -1;  }  }  } |

**PersonAddreMain.java**

|  |
| --- |
| 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 PersonAddreMain {  public static void main(String[] args) throws Exception {  args = new String[]{"F:\\A\\join\\", "F:\\A\\out\_Andy1"};  Configuration conf = new Configuration();  Job job = Job.getInstance(conf);  job.setJarByClass(PersonAddreMain.class);  //设置自定义的group  job.setGroupingComparatorClass(PKFKCompartor.class);  job.setMapperClass(PersonAddreMap.class);  job.setMapOutputKeyClass(BeanKey.class);  job.setMapOutputValueClass(Bean.class);  job.setReducerClass(PersonAddreRedu.class);  job.setOutputKeyClass(NullWritable.class);  job.setOutputValueClass(Text.class);  FileInputFormat.addInputPath(job, new Path(args[0]));  FileOutputFormat.setOutputPath(job, new Path(args[1]));  job.waitForCompletion(true);  }  } |