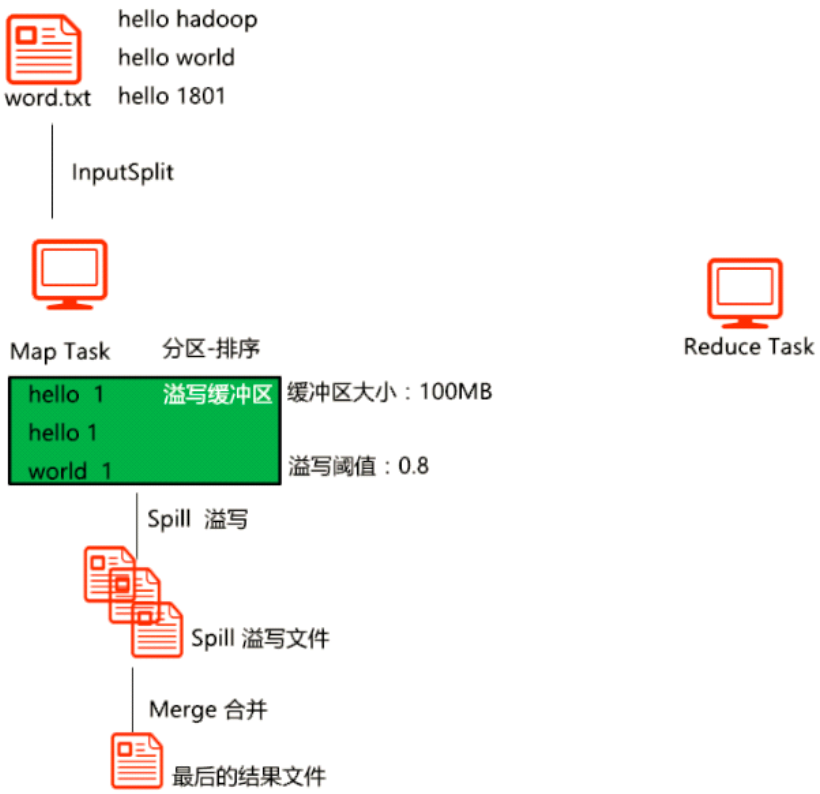
Shuffle

2018年6月15日 10:14

Shuffle-Map阶段



知识点

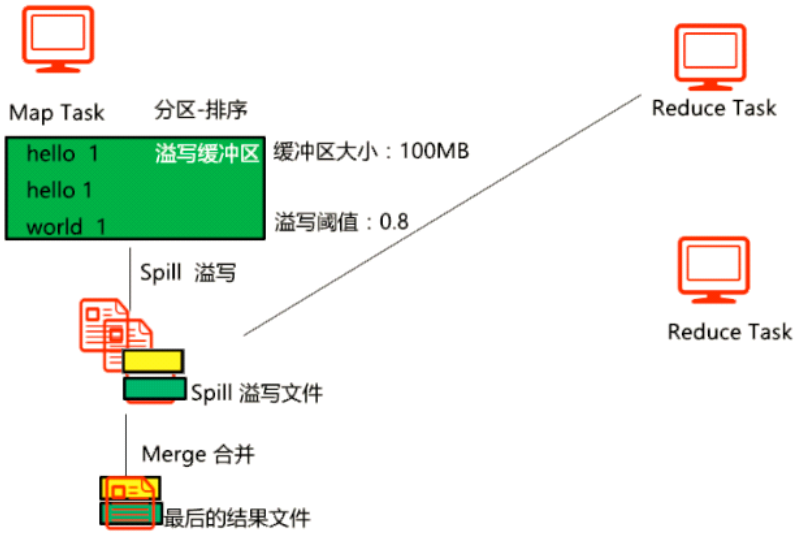
1.Map Task的输出k v，一开始会进入溢写缓冲区中，对数据做处理，比如分区、排序等操作。

2.有几个Map Task，就有几个对应的溢写缓冲区

3.溢写缓冲区默认是100MB，溢写阈值：0.8。（都可通过配置文件调节）

4.当缓冲区中的数据达到溢写阈值时，会发生Spill溢写过程。把内存中数据溢写到磁盘的文件上。

5.第4步生成的文件，称为Spill溢写文件



6.每一个Spill文件里的数据都是已分好区，且排好序的

7.当Spill过程结束之后，会发生Merge过程。目的是将多个Spill合成最后的结果文件（Finaloutput）。

8.结果文件是一个已分好区，且已排序的文件。

9.Spill和Merge过程不一定会发生。

10.如果发生了Spill过程，最后存留在溢写缓冲区里数据，会Flush到文件中。目的是确保数据都落到文件中。

11.如果发生了Spill过程，但不一定会发生Merge。即如果只有一个Spill文件，则此文件就是最后的结果文件。

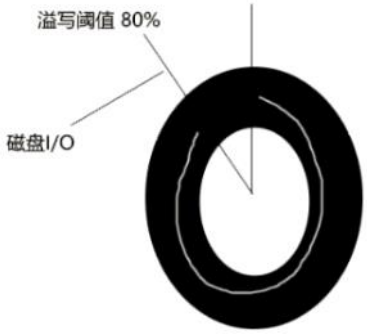
12.从性能调优的角度，可以加入Combiner中间过程，会减少数据在溢写缓冲区的存储，间接减少了Spill次数，即减少了磁盘的I/O次数。

13.如果加入了Combiner中间过程，在溢写缓冲区的处理阶段是一定会发生的。但是在Merge过程中，可能会发生。

14.Merge的Combiner不发生的条件：Spill文件的数量<3

15.从性能调优的角度，可以适当增大溢写缓冲区的大小，可以减少Spill的溢写次数。要根据服务的硬件情况来调节。一般服务器内存：32GB或64GB。结合集群的：slave节点数量+Job数量+每个Job的MapTask数量

16.溢写缓冲区也叫环写缓冲区（环形缓冲区），注意：溢写阈值的参数可调，但是不要调成100%。目的是为了避免产生写阻塞时间。此外，环形缓冲区的好处是每个MapTask重复利用同一块内存地址空间，可以减少内存碎片的产生，提高内存使用率，而且从GC角度来看，可以减少full gc发生的次数。



建议：看GC回收算法以及GC收集器，《深度理解Java虚拟机：JVM高级特性与最佳实践》（第2版）看第二章

17.可以开启Map Task的压缩机制，将最后的结果文件做压缩。好处可以减少网络数据的传输。

18.当Merge过程结束后，所有的Spill文件被删除

19.有几个Map Task ，就有几个最后结果文件。

20.最后的结果文件存到服务节点的本地磁盘上。

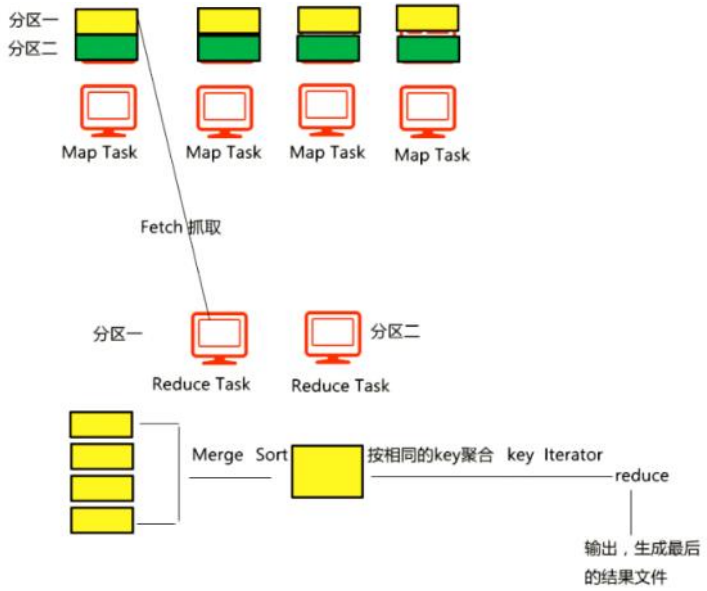
21.虽然一个Map Task处理的切片数据是128MB(满的情况），但是不能凭输入的数据大小来判断map的输出大小，要根据实际的业务代码来判断。

22.Map Task的输出结果有两类收集器：①DirectMapOutputCollector 在没有reducer组件的情况下使

用

②MapOutputBuffer 在有reducer组件的情况收集，在这个类中，包含了Spill 、溢写缓冲区相关的对象

Shuffle-Reduce阶段



知识点

1.当Map阶段接收，reduce会Fetch自己分区的数据

2.reduce 的Fetch结束后，会进行Merge 和Sort

3.Merge和Sort结束后，会发生reduce，按相同key聚合，形成key iterator传给开发者

4.Fetch线程数默认是5个，此参数可以调节。一般的做法是让此线程数接近或等于map task 数量。达到并行抓取的目的。

Shuffle（洗牌）过程

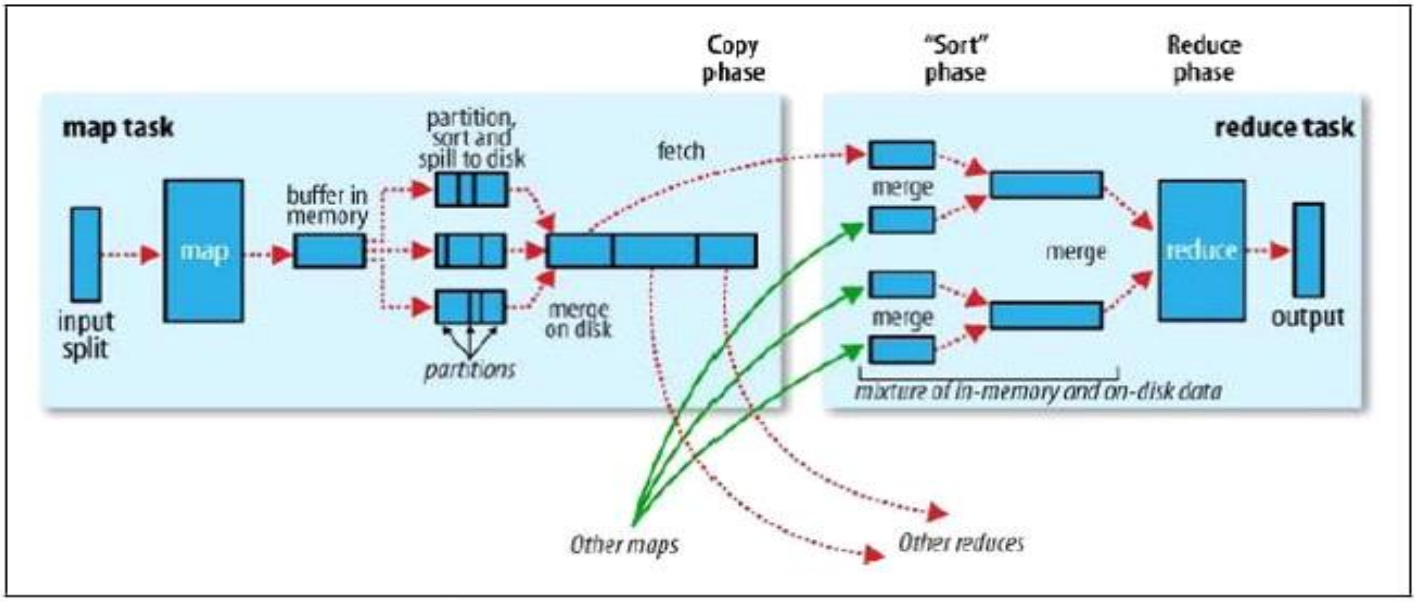


Figure 6-4. Shuffle and sort in MapReduce

Shuffle是MR框架最核心的的过程。

Hadoop常见参数控制+调优策略

配置所在文件	参数	参数默认值	作用
hdfs-site.xml	dfs.namenode.support.allow.format 格式化指令：hadoop namenode -format 作用是生成新的Fsimage文件和Edits文件。也会清空之前的元数据	true	NN是否允许被格式化？在生产系统，把它设置为false，阻止任何格式化操作在一个运行的DFS上。 建议初次格式化后，修改配置禁止，改成false
hdfs-site.xml	dfs.heartbeat.interval	3	DN的心跳间隔，秒 在集群网络通信状态不好的时候，适当调大
hdfs-site.xml	dfs.blocksize 切块大小 ①物理切块 ：真切，真的生成文件块 HDFS来实现的物理切块 ②逻辑切片：用切片对象来封装。 MapReduce来计算切片	134217728	块大小，默认是128MB 必须得是1024(page size)的整数倍
hdfs-site.xml	dfs.namenode.checkpoint.period 可以通过指令手动合并： hadoop dfsadmin -rollEdits	3600	edits和fsimage文件合并周期阈值，默认1小时
hdfs-site.xml	dfs.stream-buffer-size	4096	文件流缓存大小。需要是硬件page大小的整数倍。 在读写操作时，数据缓存大小。 注意：是1024的整数倍 注意和core-default.xml中指定文件类型的缓存是不同的，这个是dfs共用的
mapred-site.xml	mapreduce.task.io.sort.mb	100	任务内部排序缓冲区大小，默认是100MB 此参数调大，能够减少Spil溢写次数，减少磁盘I/O 建议：250MB~400MB
mapred-site.xml	mapreduce.map.sort.spill.percent	0.8	Map阶段溢写文件的阈值。不建议修改此值

mapred-site.xml	mapreduce.reduce.shuffle.parallelcopies	5	Reduce Task 启动的并发拷贝数据的线程数 建议，尽可能等于或接近于Map任务数量，达到并行抓取的效果
mapred-site.xml	mapreduce.job.reduce.slowstart.completedmaps	0.05	当Map任务数量完成率在5%时，Reduce任务启动，这个参数建议不要轻易改动，如果Map任务总量非常大时，可以将此参数调低，让reduce更早开始工作。
mapred-site.xml	io.sort.factor	10	文件合并（Merge）因子，如果文件数量太多，可以适当调大，从而减少I/O次数
mapred-site.xml	mapred.compress.map.output	false	是否对Map的输出结果文件进行压缩，默认是不压缩。但是如果Map的结果文件很大，可以开启压缩，在Reduce的远程拷贝阶段可以**节省网络带宽。**
mapred-site.xml	mapred.map.tasks.speculative.execution	true	**启动map任务的推测执行机制** 推测执行是Hadoop对“拖后腿”的任务的一种优化机制，当一个作业的某些任务运行速度明显慢于同作业的其他任务时，Hadoop会在另一个节点上为“慢任务”启动一个备份任务，这样两个任务同时处理一份数据，而Hadoop最终会将优先完成的那个任务的结果作为最终结果，并将另一个任务杀掉。 启动推测执行机制的目的是更快的完成job， 但是在集群计算资源紧张时，比如同时在运行很多个job，启动推测机制可能会带来相反效果。如果是这样，就改成false。 对于这个参数的控制，轻易不要改动。
mapred-site.xml	mapred.reduce.tasks.speculative.execution	true	

MR调优策略

Map Task和Reduce Task调优的一个原则就是

1.减少数据的传输量

2.尽量使用内存

3.减少磁盘I/O的次数

4.增大任务并行数

5.除此之外还有根据自己集群及网络的实际情况来调优。

MapReduce源码

2016年10月14日　　23:46

MapReduce计算框架重要类体系图

--环境结构

|--------JobConf
|--------JobClient
|--------InputFormat
|--------FileInputFormat
|--------TextInputFormat
|--------RecordReader
|--------LineRecordReader

|--------OutPutFormat
|--------FileOutputFormat
|--------TextOutputFormat
|--------RecordReader
|--------LineRecordWriter

--数据处理引擎

|--------MapperTask. run ( )
|--------MapperRunner
|--------MapOutputCollector
|--------MapOutputBuffer
|--------SpillThread
|--------MapTask.mergeParts()

|--------ReduceTask.run()
|--------MapOutPutCollector
|--------MapOutputCopier

JobConf介绍

JobConf是用户描述一个job的接口。下面的信息是MapReduce过程中一些较关键的定制信息：

InputFormat	将输入的数据集切割成小数据集 InputSplits，每一个 InputSplit 将由一个 Mapper 负责处理。此外 InputFormat 中还提供一个 RecordReader 的实现，将一个 InputSplit 解析成 <key,value> 对提供给 map 函数。	TextInputFormat (针对文本文件，按行将文本文件切割成 InputSplits，并用 LineRecordReader 将 InputSplit 解析成 <key,value> 对，key 是行在文件中的位置，value 是文件中的一行)	SequenceFileInputFormat
OutputFormat	提供一个 RecordWriter 的实现，负责输出最终结果	TextOutputFormat (用 LineRecordWriter 将最终结果写成纯文件文件,每个 <key,value> 对一行，key 和 value 之间用 tab 分隔)	SequenceFileOutputFormat
OutputKeyClass	输出的最终结果中 key 的类型	LongWritable	
OutputValueClass	输出的最终结果中 value 的类型	Text	
MapperClass	Mapper 类，实现 map 函数，完成输入的 <key,value> 到中间结果的映射	IdentityMapper (将输入的 <key,value> 原封不动的输出为中间结果)	LongSumReducer, LogRegexMapper, InverseMapper
CombinerClass	实现 combine 函数，将中间结果中的重复 key 做合并	null (不对中间结果中的重复 key 做合并)	
ReducerClass	Reducer 类，实现 reduce 函数，对中间结果做合并，形成最终结果	IdentityReducer (将中间结果直接输出为最终结果)	AccumulatingReducer, LongSumReducer
InputPath	设定 job 的输入目录，job 运行时会处理输入目录下的所有文件	null	
OutputPath	设定 job 的输出目录，job 的最终结果会写入输出目录下	null	
MapOutputKeyClass	设定 map 函数输出的中间结果中 key 的类型	如果用户没有设定的话，使用 OutputKeyClass	
MapOutputValueClass	设定 map 函数输出的中间结果中 value 的类型	如果用户没有设定的话，使用 OutputValuesClass	
OutputKeyComparator	对结果中的 key 进行排序时的使用的比较器	WritableComparable	
PartitionerClass	对中间结果的 key 排序后，用此 Partition 函数将其划分为R份,每份由一个 Reducer 负责处理。	HashPartitioner (使用 Hash 函数做 partition)	KeyFieldBasedPartitioner PipesPartitioner

MapTask介绍

MapTask启动和执行一个Map任务，初始化、启动map任务，调用的是run()方法。

run()方法源码骨架：

run方法里调用了runOldMapper（ ）方法。这个方法是用于执行用户自定义Mapper类里的Map（)方法。（ M/R 2.0版本调用的是runNewMapper()方法 ）

runOldMapper（ ）方法源码骨架：

```
InputSplit inputSplit ……
RecordReader<INKEY,INVALUE> in ……
MapRunnable<INKEY,INVALUE,OUTKEY,OUTVALUE> runner
=ReflectionUtils.newInstance(job.getMapRunnerClass(), job);
```

MapOutputCollector<OUTKEY, OUTVALUE> collector……

方法作用释义：

1.这个方法会解析job里的InputSplit信息，InputSplit信息里封装了一个Map任务处理的文件切片信息，如文件切片所在的节点位置信息，处理的start位置以及处理的长度length（可以类比Zebra的FileSpilt）

2.拿到文件切片信息之后，怎么来读取文件内容呢？——>解析job指定的RecordReader，Hadoop默认的是LineRecordReader（按行读取，把每行头位置信息的偏移量作为key，每行的内容作为Value)。

3.MapRunnalbe的作用是，根据反射机制，拿到用户（程序员）自定义的Mapper类，拿到这个类之后，就可以拿到用户的输出map<key,value>了。

细节说明：MapRunnable是一个接口，那实际上这里runner对象是它的实现类：MappRunner类的对象。

MapRunner介绍

MapperRunner类源码骨架：

```
private Mapper<K1, V1, K2, V2> mapper;
public void configure(JobConf job) {
    this.mapper =
ReflectionUtils.newInstance(job.getMapperClass(), job);
 }
 public void run(RecordReader<K1, V1> input, OutputCollector<K2,
V2> output,Reporter reporter){
      while (input.next(key, value)) {
            mapper.map(key, value, output);
          }
      }
```

方法释义：MapRunner通过指定的Redcord读取器，读取input split的内容（默认是一行一行读）。每读取到一行，就调用一次map方法，并获得每次map方法的输出map<k,v>

MapOutputCollector介绍

4.MapOutputCollector是一个接口,作用是收集每次调用map后得到的新的kv对，然后把他们spill到文件或者放到内存，以做进一步的处理，比如分区、排序，combine等（就是shuffle过程）。
即MapOutputCollector是处理输出map的类。
细节说明：MapOutputCollector 其中有两个子类：MapOutputBuffer和DirectMapOutputCollector。 DirectMapOutputCollector用在不需要Reduce阶段的时候。（比如在做利润排序案例的时候，没有reducer）

如果Mapper后续有reduce任务，系统会使用MapOutputBuffer做为实现类，MapOutputBuffer使用了一个缓冲区对输出map对进行缓存。
默认缓存大小是100mb，当缓存中数据量达到80%的时候，发生spill过程，溢写到本地磁盘上。

MapOutputBuffer介绍

MapOutputBuffer骨架源码：

```
public static class MapOutputBuffer<K extends Object, V extends Object> implements MapOutputCollector<K, V>, IndexedSortable {
    private int partitions; //分区数量，reduce的数量，如果用户不设置，默认是1个
   private Class<K> keyClass;//输出map的key类型
    private Class<V> valClass;//输出map的value类型
    private RawComparator<K> comparator;//比较器，用于排序
  final SpillThread spillThread = new SpillThread();//是一个线程类，负责spill溢写到磁盘的过程
   final BlockingBuffer bb = new BlockingBuffer();//用于存放map输出结果的缓冲区

  public synchronized void collect(K key, V value, final int partition){
    startSpill();//开始溢写线程，这个方法会启动SpillThread线程开始溢写。溢写的条件是缓冲区里的数据达到总大小的80%。默认是缓冲区是100mb。这个是可以在配置文件里配置的
    keySerializer.serialize(key);//将输出map<K,V>的Key值序列化
```

```
    valSerializer.serialize(value);//将输出map<K,V>的Value值序列
    化，key和value序列化后，将其数据溢写到本地文件里


  }
  //溢写具体是由SpilThread这个线程类来负责的
   protected class SpillThread extends Thread {
    sortAndSpill();//需要spill时调用函数sortAndSpill，按照
    partition和key做排序。默认使用的是快速排序QuickSort算法。使
    用的比较器，默认是WritableComparable。Hadoop的基本数据类
    型都是实现了这个WritableComparable接口。

   }

//用户在结束map处理后，已经没有数据再输出到缓冲区，但缓存中还有
数据没有刷到磁盘上，需要将缓存中的数据 flush到磁盘上，这个动作就
是由MapOutputBuffer的flush来完成。
  public void flush(){

  }
}
```

MapTask.mergeParts()方法源码骨架：

```
private void mergeParts() {
      final Path[] filename = new Path[numSpills];//记录了生成的
Spill文件信息
  //循环得到每个Spill文件名字以及长度
      for(int i = 0; i < numSpills; i++) {
        filename[i] = mapOutputFile.getSpillFile(i);
        finalOutFileSize +=
rfs.getFileStatus(filename[i]).getLen() ;

        }
 //将多个Spill文件内容通过finalOut输出流写出到finalOutputFile文
件，这个文件就是最后的结果文件
//此外，merge后的文件是一个已分区且已排好序的文件
    FSDataOutputStream finalOut=rfs.create(finalOutputFile,
true, 4096);
```

```
    sortPhase.addPhases(partitions);
     Writer<K, V> writer = new Writer<K, V>(job, finalOut,
keyClass, valClass, codec,
                                spilledRecordsCounter);
    //如果用户未指定有Combineg过程，直接执行merge合并
    //如果用户指定了有Combine过程，但是如果spill文件数量小于3，
在merger阶段也不会发生combine
      if (combinerRunner == null || numSpills
< minSpillsForCombine) {
            Merger.writeFile(kvIter, writer, reporter, job);
          } else {
//如果用户指定了Comibine，并spill文件数量大于3，则会发生Combine，
然后进行merge
            combineCollector.setWriter(writer);
            combinerRunner.combine(kvIter, combineCollector);
          }
          sortPhase.startNextPhase();

        finalOut.close();
//Merge之后，把Spill文件删掉
        for(int i = 0; i < numSpills; i++) {
          rfs.delete(filename[i],true);
        }
      }
```

ReduceTask介绍

ReduceTask用于初始化和执行Reduce任务。

初始化和启动reduce任务的方法入口是：run()方法

run()方法源码骨架：

```
void run(final JobConf job, final TaskUmbilicalProtocol
umbilical){
   //定义了reduce一共有三个阶段，分别是：copy、sort、reduce阶
段
    copyPhase = getProgress().addPhase("copy");
       sortPhase  = getProgress().addPhase("sort");
       reducePhase = getProgress().addPhase("reduce");

    TaskReporter reporter = startReporter(umbilical);//设置并启动
```

report进程以便和TaskTracker进程通信，汇报reudce任务的执行情况

Class combinerClass = conf.getCombinerClass();//获取CombinerClass

shuffleConsumerPlugin.init(shuffleContext);//shuffleConsumerPlugin.init()方法用于初始化copy阶段所依赖的运行环境，以及创建Merge管理器。copy阶段就是从各个Map任务服务器那里，去拷贝map的输出文件自己分区的数据到reduce任务节点上。这一过程，我们也称之为Fetch。当reduce节点拿到所有的输出文件之后，如果文件数量太多，（hadoop默认是的数量时10）会进行文件的合并，这个过程称为merge。并且在文件合并时，会按key进行排序。

rIter = shuffleConsumerPlugin.run();//shuffleConsumerPlugin.run()方法就是执行fetch过程、merge过程、以及sort过程。并且，把最后的结果封装成:[key1，Iterable<>] [key2,Iterable<>]这样的形式，封装到rIter 对象里。

mapOutputFilesOnDisk.clear();//reduce把各个map任务节点上的文件merger完后之后会生成新文件，则原来的那些就没用了，所以删掉

sortPhase.complete(); //sortpahse完成，copyphase阶段是在先于sortphase完成

setPhase(TaskStatus.Phase.REDUCE); //更新当前状态，进入reduce阶段。

statusUpdate(umbilical);//通过rpc告知JobTracker当前reduce任务的执行阶段。

Class keyClass = job.getMapOutputKeyClass();//获取输出key值类型

Class valueClass = job.getMapOutputValueClass(); //获取reduce输出value类型

RawComparator comparator = job.getOutputValueGroupingComparator();//获取比较器

//这个是执行reduce 合并的方法入口

runOldReducer(job, umbilical, reporter, rIter, comparator, keyClass, valueClass);

done(umbilical, reporter);//以上三个阶段都完成后，reduce任务结束，做一些资源清理工作，并最后向JobTracker发送一次统计报告，然

后结束Reporter通信线程。

}

执行Reduce任务的方法入口是：runOldReducer()方法 （ M/R 1.x版本API ）

---

runOldReducer()方法源码骨架：

```
void runOldReducer(){

 Reducer<INKEY,INVALUE,OUTKEY,OUTVALUE> reducer =
ReflectionUtils.newInstance(job.getReducerClass(), job);//得到用户定义的Reduce实现类
String finalName = getOutputName(getPartition());//得到分区数量，hadoop默认分区是1。
RecordWriter<OUTKEY, OUTVALUE> finalOut =.....//确定reduce结果输出器，Hadoop默认是LineRecordWriter，输出形式：reduce输出map.key  Tab  reduce输出map.value
reduce输出map.key  Tab  reduce输出map.value .....

 OutputCollector<OUTKEY,OUTVALUE> collector =  new
OutputCollector<OUTKEY,OUTVALUE>() {
        public void collect(OUTKEY key, OUTVALUE value)
        {
          finalOut.write(key, value);//
         }
      };
ReduceValuesIterator<INKEY,INVALUE> values=rIter；//将rIter里的数据拿出来。
while (values.more()) {
         reducer.reduce(values.getKey(), values, collector,
reporter);
         values.nextKey();
         }//循环执行reduce.reduce方法，并把每次reduce的输出map结果通过Collector，写出到结果文件里。直到所有遍历完所有的key值后，退出reduce方法。
```

```
    reducer.close();//reduce阶段结束
}
```

复制线程(MapOutputCopier)

Map输出复制器ReduceCopier在其内部采用多线程的方式来从其它的TaskTracker上通过http协议请求的复制属于自己的Map任务输出结果。至于复制线程的个数可在配置文件mapred-site.xml中配置，对应的配置项：mapred.reduce.parallel.copies，为了提高reduce及整个Hadoop的效率，这个值应该设置和该作业的Map任务数差不多。每一个复制线程都从scheduledCopies 中获取一个任务来执行，在接受目标TaskTracker传过来的Map输出数据时，它会根据当前内存情况决定将该数据存放在村内还是磁盘。

自定义格式输入

需求说明：

map函数输入key变为IntWritable类型，并且表示的是当前处理的行数。

map函数输入value为Text类型，表示每行内容

AuthFormat代码:

```
public class AuthFormat extends FileInputFormat<IntWritable, Text>{

    @Override

    public RecordReader<IntWritable, Text> createRecordReader(InputSplit split,

    TaskAttemptContext context)

              throws IOException, InterruptedException {


        return new AuthReader();

    }


}
```

AuthReader代码：

```
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.IntWritable;

import org.apache.hadoop.io.Text;
```

```
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;

import org.apache.hadoop.util.LineReader;


public class AuthReader extends RecordReader<IntWritable, Text>{

    private FileSplit fs ;

    private LineReader lineReader ;

    private IntWritable key ;

    private Text value ;

    //--定义一个计数器，记录本次读取到了多少行

    int count = 0;


    @Override

    public void initialize(InputSplit split, TaskAttemptContext context) throws

    IOException, InterruptedException {

        this.fs = (FileSplit) split;

        //--获取文件路径

        Path path = fs.getPath();

        //--获取文件系统

        Configuration conf = context.getConfiguration();

        FileSystem fileSystem = path.getFileSystem(conf);

        //--通过文件系统读取文件得到流

        FSDataInputStream in = fileSystem.open(path);
```

```
        //--将流包装为LineReader方便按行读取
        lineReader = new LineReader(in);
    }

    @Override
    public boolean nextKeyValue() throws IOException, InterruptedException {
        //--要返回的键，本次读取的行数
        key = new IntWritable();
        //--要返回的值，本次读取到的内容
        value = new Text();
        //--定义一个temp临时记录内容
        Text temp = new Text();
        int len = lineReader.readLine(temp);
        //--判断是否读取到了数据
        if(len == 0){
            //--表示没有行数据可读，则不再执行 nextKeyValue()方法
            return false;

        }else{
            //--读到了数据，将数据追加到value中
            //可以这样写：value=tmp;
            //也可以像下面这样写
            value.append(temp.getBytes(), 0, temp.getLength());
            //--计数器加1，表明读取到了一行内容
            count++;
```

```
            key.set(count);
            return true;
            }
}

@Override
public IntWritable getCurrentKey() throws IOException, InterruptedException {
     return key;
}

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

@Override
public float getProgress() throws IOException, InterruptedException {
     return 0;
}

@Override
public void close() throws IOException {
     if(lineReader != null)lineReader.close();
}
```

```
}
```

AuthMapper代码：

```
public class AuthMapper extends Mapper<IntWritable,Text,IntWritable,Text>{
    @Override
    protected void map(IntWritable key, Text value, Mapper<IntWritable, Text,
    IntWritable, Text>.Context context)
            throws IOException, InterruptedException {
        context.write(key, value);
    }

}
```

AuthDriver代码：

```
public class AuthDriver {
    public static void main(String[] args) throws Exception {
        Configuration conf=new Configuration();
        Job job=Job.getInstance(conf);
        job.setJarByClass(AuthDriver.class);
        job.setMapperClass(AuthMapper.class);
        job.setMapOutputKeyClass(IntWritable.class);
        job.setMapOutputValueClass(Text.class);
        job.setInputFormatClass(AuthFormat.class);

        FileInputFormat.setInputPaths(job,new
        Path("hdfs://192.168.234.21:9000/word"));
        FileOutputFormat.setOutputPath(job,new
```

```
            Path("hdfs://192.168.234.21:9000/word/result"));
            job.waitForCompletion(true);
        }
}
```

自定义格式计算个人成绩

案例数据：

```
tom
math 90
english 98
jary
math 78
english 87
rose
math 87
english 90
tom
math 67
english 87
jary
math 59
english 80
rose
math 79
english 60
```

要求整理成如下数据格式：

```
jary  math 59 english 80
jary  math 78 english 87
```

```
rose   math 79 english 60

rose   math 87 english 90

tom    math 67  english 87

tom    math 90  english 98
```

AuthInputFormat代码：

```
public class AuthFormat extends FileInputFormat<Text, Text>{

     @Override

     public RecordReader<Text, Text> createRecordReader(InputSplit split,

     TaskAttemptContext context)

               throws IOException, InterruptedException {

          return new AuthReader();

     }

}
```

AuthReader代码：

```
public class AuthReader extends RecordReader<Text, Text>{

     private LineReader lineReader;

     private Text key;

     private Text value;
```

```
@Override
public void initialize(InputSplit split, TaskAttemptContext context) throws
IOException, InterruptedException {
     FileSplit fileSplit=(FileSplit) split;
     Path path=fileSplit.getPath();
     FileSystem fs=path.getFileSystem(context.getConfiguration());
     FSDataInputStream in=fs.open(path);
     lineReader=new LineReader(in);

}

@Override
public boolean nextKeyValue() throws IOException, InterruptedException {
     key=new Text();
     value=new Text();
     Text tmp=new Text();
     int result=lineReader.readLine(tmp);
     if(result==0){
          return false;
     }
     else{
         //先读取姓名
```

```
            key.set(tmp);
        }
        //读取姓名后，读取姓名的下两行内容，然后把两行内容用空格 拼接成一行内
        容。
        //这样，最后形成的key/value 就变成了  姓名/对应的学科成绩
        for(int i=0;i<2;i++){
            lineReader.readLine(tmp);
            value.append(tmp.getBytes(),0,tmp.getLength());
            value.append(" ".getBytes(),0,1);
        }
        return true;
    }

    @Override
    public Text getCurrentKey() throws IOException, InterruptedException {

        return key;
    }

    @Override
    public Text getCurrentValue() throws IOException, InterruptedException {

        return value;
    }
```

```
	@Override
	public float getProgress() throws IOException, InterruptedException {

		return 0;
	}

	@Override
	public void close() throws IOException {
		if(lineReader!=null){
			lineReader.close();
		}

	}

}
```

ScoreMapper代码：

```
public class ScoreMapper extends Mapper<Text, Text, Text, Text>{

	@Override
	protected void map(Text key, Text value, Mapper<Text, Text, Text,
	Text>.Context context)
			throws IOException, InterruptedException {
		context.write(key, value);
	}
}
```

ScoreDirver代码：

```
public class ScoreDriver {


	public static void main(String[] args) throws Exception {
		Configuration conf = new Configuration();
		Job job = Job.getInstance(conf, "JobName");
		job.setJarByClass(cn.tarena.hadoop.ScoreDriver.class);
		job.setMapperClass(cn.tarena.hadoop.ScoreMapper.class);


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


		job.setInputFormatClass(AuthInputFormat.class);


		FileInputFormat.setInputPaths(job, new
		Path("hdfs://192.168.234.21:9000/formatscore"));
		FileOutputFormat.setOutputPath(job, new
		Path("hdfs://192.168.234.21:9000/formatscore/result"));


		if (!job.waitForCompletion(true))
			return;
	}


}
```

自定义格式输出

需求说明

最后的输出结果，key和value间用|分割。每行输出完后，回车换行\r\n

```
jary|math 59 english 80
jary|math 78 english 87
rose|math 79 english 60
rose|math 87 english 90
tom|math 67  english 87
tom|math 90  english 98
```

AuthOutputFormat代码：

```
public class AuthOutputFormat<K,V> extends FileOutputFormat<K,V>{

	@Override
	public RecordWriter<K,V> gtetRecordWrier(TaskAttemptContext job) throws
	IOException, InterruptedException {

		//Get the default path and filename for the output format.
		//第二个参数：extension an extension to add to the filename
		Path path=super.getDefaultWorkFile(job, "");
		Configuration conf=job.getConfiguration();
		FileSystem fs=path.getFileSystem(conf);
		FSDataOutputStream out=fs.create(path);


		return new AuthWriter<K,V>(out,"|","\r\n");
	}

}
```

AuthWriter代码：

```
public class AuthWriter<K,V> extends RecordWriter<K,V>{

	private FSDataOutputStream out;
	private String keyValueSeparator;
	private String lineSeparator;


	public AuthWriter(FSDataOutputStream out, String keyValueSeparator, String
	lineSeparator) {
		this.out=out;
		this.keyValueSeparator=keyValueSeparator;
		this.lineSeparator=lineSeparator;
	}

	@Override
	public void write(K key, V value) throws IOException, InterruptedException {
		out.write(key.toString().getBytes());
```

```
            out.write(keyValueSeparator.getBytes());
            out.write(value.toString().getBytes());
            out.write(lineSeparator.getBytes());

        }

        @Override
        public void close(TaskAttemptContext context) throws IOException,
        InterruptedException {
            if(out!=null)out.close();

        }

    }
```

AuthDriver代码：

```
    public class ScoreDriver {

        public static void main(String[] args) throws Exception {
            Configuration conf = new Configuration();
            Job job = Job.getInstance(conf, "JobName");
            job.setJarByClass(cn.tarena.hadoop.ScoreDriver.class);
            job.setMapperClass(cn.tarena.hadoop.ScoreMapper.class);

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

            job.setInputFormatClass(AuthInputFormat.class);
            job.setOutputFormatClass(AuthOutputFormat.class);

            FileInputFormat.setInputPaths(job, new
            Path("hdfs://192.168.234.21:9000/formatscore"));
            FileOutputFormat.setOutputPath(job, new
            Path("hdfs://192.168.234.21:9000/formatscore/result"));

            if (!job.waitForCompletion(true))
                return;
        }

    }
```

MultipleInputs 多输入源

需求说明

有下面两个不同格式的文件，现在想通过一个MR来进行每个人的成绩统计工作。

注意，不能用两个MR，因为那相当于两个job任务。

tom	tom math 90 english 98
math 90	jary math 78 english 87
english 98	rose math 87 english 90
jary	tom math 67 english 87
math 78	jary math 59 english 80
english 87	rose math 79 english 60
rose	
math 87	
english 90	
tom	
math 67	
english 87	
jary	
math 59	
english 80	
rose	
math 79	
english 60	

ScoreDriver代码：

```
public class ScoreDriver {

	public static void main(String[] args) throws Exception {
		Configuration conf = new Configuration();
		Job job = Job.getInstance(conf, "JobName");
		job.setJarByClass(cn.tarena.hadoop.ScoreDriver.class);

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

		job.setInputFormatClass(AuthInputFormat.class);
		//需要注意，如果一个Mapper代码不能通用的解决，则需要分别指定。此时，就不能去设置
		//setMapperClass()了
		MultipleInputs.addInputPath(job, new
		Path("hdfs://192.168.234.21:9000/formatscore/format-
		score.txt"),AuthInputFormat.class,ScoreMapper.class);

		MultipleInputs.addInputPath(job, new
		Path("hdfs://192.168.234.21:9000/formatscore/format-
		score-1.txt"),TextInputFormat.class,ScoreMapper2.class);


		FileOutputFormat.setOutputPath(job, new
		Path("hdfs://192.168.234.21:9000/formatscore/result"));

		if (!job.waitForCompletion(true))
			return;
	}

}
```

AuthInputformat代码：

```
public class AuthInputFormat extends FileInputFormat<Text, Text>{

	@Override
	public RecordReader<Text, Text> createRecordReader(InputSplit split, TaskAttemptContext
	context)
```

```
            throws IOException, InterruptedException {

        return new AuthReader();
    }

}
```

AuthReader代码：

```
public class AuthReader extends RecordReader<Text, Text>{

    private LineReader lineReader;
    private Text key;
    private Text value;



    @Override
    public void initialize(InputSplit split, TaskAttemptContext context) throws IOException,
    InterruptedException {
        FileSplit fileSplit=(FileSplit) split;
        Path path=fileSplit.getPath();
        FileSystem fs=path.getFileSystem(context.getConfiguration());
        FSDataInputStream in=fs.open(path);
        lineReader=new LineReader(in);

    }

    @Override
    public boolean nextKeyValue() throws IOException, InterruptedException {
        key=new Text();
        value=new Text();
        Text tmp=new Text();
        int result=lineReader.readLine(tmp);
        if(result==0){
            return false;
        }
        else{
            //先读取姓名
```

```
            key.set(tmp);
        }
        //读取姓名后，读取姓名的下两行内容，然后把两行内容用空格 拼接成一行内容。
        //这样，最后形成的key/value 就变成了  姓名/对应的学科成绩
        for(int i=0;i<2;i++){
            lineReader.readLine(tmp);
            value.append(tmp.getBytes(),0,tmp.getLength());
            value.append(" ".getBytes(),0,1);
        }
        return true;
    }

    @Override
    public Text getCurrentKey() throws IOException, InterruptedException {

        return key;
    }

    @Override
    public Text getCurrentValue() throws IOException, InterruptedException {

        return value;
    }

    @Override
    public float getProgress() throws IOException, InterruptedException {

        return 0;
    }

    @Override
    public void close() throws IOException {
        if(lineReader!=null){
            lineReader.close();
        }

    }
```

}

ScoreMapper代码：

```
public class ScoreMapper extends
Mapper<Text, Text, Text, Text>{


     @Override
     protected void map(Text key, Text
     value, Mapper<Text, Text, Text,
     Text>.Context context)
          throws IOException,
          InterruptedException {
          context.write(key, value);
     }
}
```

ScoreMapper2代码：

```
class ScoreMapper2 extends
Mapper<LongWritable,Text, Text,Text>{
     @Override
     protected void map(LongWritable key, Text
     value, Mapper<LongWritable, Text, Text,
     Text>.Context context)
          throws IOException, InterruptedException
          {
          String name=value.toString().split(" ")
          [0];
          String
          score=value.toString().split(name)[1];
         context.write(new Text(name), new
     Text(score));
     }
}
```

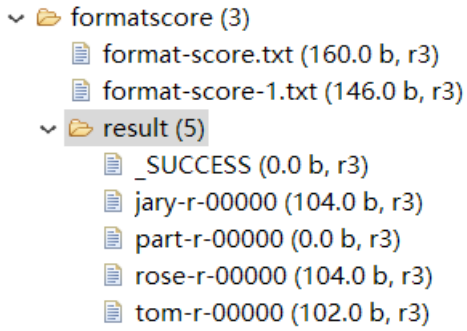
MultipleOutputs多输出源

一个reduce有多个输出结果文件。按规则将结果写到对应的结果文件里

- 源文件：

```
tom math 90 english 98
jary math 78 english 87
rose math 87 english 90
tom math 67 english 87
jary math 59 english 80
rose math 79 english 60
```

- 最后的结果：



- 某个结果文件示意：

```
jary  math 59 english 80
jary  math 78 english 87
jary math 59 english 80
jary math 78 english 87
```

- ***ScoreMapper代码略***

- ***ScoreReducer代码：***

```
public class ScoreReducer extends Reducer<Text, Text, Text, Text>{

    private MultipleOutputs<Text, Text> mos;

    @Override
    protected void reduce(Text name, Iterable<Text> scores, Reducer<Text, Text,
    Text, Text>.Context context)
            throws IOException, InterruptedException {
```

```
            for(Text score:scores){
                if(name.toString().equals("jary")){
                    mos.write("jary",name,score);
                }
                if(name.toString().equals("rose")){
                    mos.write("rose",name,score);
                }
                if(name.toString().equals("tom")){
                    mos.write("tom", name,score);
                }
            }

        }
        @Override
        protected void setup(Reducer<Text, Text, Text, Text>.Context context) throws
        IOException, InterruptedException {
            mos=new MultipleOutputs<>(context);
        }

    }
```

ScoreDriver代码：

```
public class ScoreDriver {

    public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        Job job = Job.getInstance(conf, "JobName");
        job.setJarByClass(cn.tarena.hadoop.ScoreDriver.class);


        // TODO: specify a reducer
        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(Text.class);
        job.setReducerClass(ScoreReducer.class);

        job.setInputFormatClass(AuthInputFormat.class);

        MultipleInputs.addInputPath(job, new
        Path("hdfs://192.168.234.21:9000/formatscore/format-
```

```
        score.txt"),AuthInputFormat.class,ScoreMapper.class);
        MultipleInputs.addInputPath(job, new
        Path("hdfs://192.168.234.21:9000/formatscore/format-
        score-1.txt"),TextInputFormat.class,ScoreMapper2.class);


        MultipleOutputs.addNamedOutput(job, "jary", AuthOutputFormat.class,
        Text.class, Text.class);
        MultipleOutputs.addNamedOutput(job, "tom", AuthOutputFormat.class,
        Text.class, Text.class);
        MultipleOutputs.addNamedOutput(job, "rose", AuthOutputFormat.class,
        Text.class, Text.class);

        FileOutputFormat.setOutputPath(job, new
        Path("hdfs://192.168.234.21:9000/formatscore/result"));

        if (!job.waitForCompletion(true))
             return;
    }

}
```

二次排序

待处理文件：

2 tom 345

1 rose 235

1 tom 234

2 jim 572

3 rose 123

1 jim 321

2 tom 573

3 jim 876

3 tom 648

需求：

对每个人每个月的销售业绩排序，先按月份做升序排序，再按利润做降序排序

最后的结果：

Profit [month=1, name=jim, profit=321]

Profit [month=1, name=rose, profit=235]

Profit [month=1, name=tom, profit=234]

Profit [month=2, name=tom, profit=573]

Profit [month=2, name=jim, profit=572]

Profit [month=2, name=tom, profit=345]

Profit [month=3, name=jim, profit=876]

Profit [month=3, name=tom, profit=648]

Profit [month=3, name=rose, profit=123]

ProfitMapper:

```
public class ProfitMapper extends Mapper<LongWritable,Text,Profit,NullWritable>{

    @Override
    protected void map(LongWritable key, Text value, Mapper<LongWritable, Text, Profit,
    NullWritable>.Context context)
            throws IOException, InterruptedException {
        String line=value.toString();
```

```
            String[] data=line.split(" ");
            int month=Integer.parseInt(data[0]);
            String name=data[1];
            int profit=Integer.parseInt(data[2]);
            Profit p=new Profit();
            p.setMonth(month);
            p.setName(name);
            p.setProfit(profit);
            context.write(p, NullWritable.get());
        }
}
```

Profit：

```
public class Profit implements WritableComparable<Profit>{

    private int month;
    private String name;
    private int profit;

    @Override
    public void write(DataOutput out) throws IOException {
        out.writeInt(month);
        out.writeUTF(name);
        out.writeInt(profit);

    }

    @Override
    public void readFields(DataInput in) throws IOException {
        this.month=in.readInt();
        this.name=in.readUTF();
        this.profit=in.readInt();

    }

    @Override
    public int compareTo(Profit o) {
        int result=this.month-o.month;
        if(result!=0){
```

```
            return result;
        }else{
            return o.profit-this.profit;
        }


}

public int getMonth() {
        return month;
}

public void setMonth(int month) {
        this.month = month;
}

public String getName() {
        return name;
}

public void setName(String name) {
        this.name = name;
}

public int getProfit() {
        return profit;
}

public void setProfit(int profit) {
        this.profit = profit;
}

@Override
public String toString() {
        return "Profit [month=" + month + ", name=" + name + ", profit=" + profit + "]";
}
```

```
}
```

ProfitDriver：

```
public class ProfitDriver {

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


        job.setJarByClass(ProfitDriver.class);

        job.setMapperClass(ProfitMapper.class);



        job.setMapOutputKeyClass(Profit.class);

        job.setMapOutputValueClass(NullWritable.class);




        FileInputFormat.setInputPaths(job,new Path("hdfs://192.168.234.241:9000/ssort"));

        FileOutputFormat.setOutputPath(job, new
        Path("hdfs://192.168.234.241:9000/ssort/result"));


        job.waitForCompletion(true);
    }

}
```

课后作业—Join

2018年7月16日 20:09

订单表数据

1001 20170710 4 2

1002 20170710 3 100

1003 20170710 2 40

1004 20170711 2 23

1005 20170823 4 55

1006 20170824 3 20

1007 20170825 2 3

1008 20170826 4 23

1009 20170912 2 10

1010 20170913 2 2

1011 20170914 3 14

1012 20170915 2 18

商品表数据

1 chuizi 3999

2 Huawei 3999

3 Xiaomi 2999

4 Apple 5999

最后输出的结果：

20170710 Item [订单id=1003, 订单日期=20170710, 物品id=2, 出货量=40, 品牌=Huawei, 商品单价=3999.0]

20170710 Item [订单id=1002, 订单日期=20170710, 物品id=3, 出货量=100, 品牌=Xiaomi, 商品单价=2999.0]

20170710 Item [订单id=1001, 订单日期=20170710, 物品id=4, 出货量=2, 品牌=Apple, 商品单价=5999.0]

20170711 Item [订单id=1004, 订单日期=20170711, 物品id=2, 出货量=23, 品牌=Huawei, 商品单价=3999.0]

20170823 Item [订单id=1005, 订单日期=20170823, 物品id=4, 出货量=55, 品牌=Apple, 商品单价=5999.0]

20170824 Item [订单id=1006, 订单日期=20170824, 物品id=3, 出货量=20, 品牌=Xiaomi, 商品单价=2999.0]

20170825 Item [订单id=1007, 订单日期=20170825, 物品id=2, 出货量=3, 品牌=Huawei, 商品单价=3999.0]

20170826 Item [订单id=1008, 订单日期=20170826, 物品id=4, 出货量=23, 品牌=Apple, 商品单价=5999.0]

课后作业—倒排索引

2018年8月2日 10:00

a.txt

hello hadoop

hello world

hello hadoop

b.txt

hello spark

hello hive

hello hadoop

c.txt

hello hadoop

1803 hadoop

hadoop hive

最后形成的倒排索引表：

1803 c.txt 1

hadoop c.txt 2 b.txt 1 a.txt 2

hello c.txt 1 b.txt 3 a.txt 3

hive c.txt 1 b.txt 1

spark b.txt 1

world a.txt 1