5. MAPREDUCE 实践篇

5.1 MAPREDUCE 示例编写及编程规范

5.1.1 编程规范

（1）**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)
（2）Mapper 的输入数据是 KV 对的形式（KV 的类型可自定义）
（3）Mapper 的输出数据是 KV 对的形式（KV 的类型可自定义）
（4）Mapper 中的业务逻辑写在 map()方法中
（5）map()方法（maptask 进程）对每一个<K,V>调用一次
（6）Reducer 的输入数据类型对应 Mapper 的输出数据类型，也是 KV
（7）Reducer 的业务逻辑写在 reduce()方法中
（8）Reducetask 进程对每一组相同 k 的<k,v>组调用一次 reduce()方法
（9）用户自定义的 Mapper 和 Reducer 都要继承各自的父类
（10）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

5.1.2 wordcount 示例编写

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)定义一个 mapper 类

```
//首先要定义四个泛型的类型
//keyin:    LongWritable       valuein: Text
//keyout: Text                    valueout:IntWritable


public class WordCountMapper extends Mapper<LongWritable, Text, Text, IntWritable>{
	//map 方法的生命周期：    框架每传一行数据就被调用一次
	//key :   这一行的起始点在文件中的偏移量
	//value:  这一行的内容
	@Override
	protected void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {
		//拿到一行数据转换为 string
		String line = value.toString();
		//将这一行切分出各个单词
		String[] words = line.split(" ");
		//遍历数组，输出<单词，1>
		for(String word:words){
```

```
                context.write(new Text(word), new IntWritable(1));
            }
        }
}
```

(2)定义一个 reducer 类

```
    //生命周期：框架每传递进来一个 kv 组，reduce 方法被调用一次
    @Override
    protected void reduce(Text key, Iterable<IntWritable> values, Context context) throws IOException, InterruptedException {
        //定义一个计数器
        int count = 0;
        //遍历这一组 kv 的所有 v，累加到 count 中
        for(IntWritable value:values){
            count += value.get();
        }
        context.write(key, new IntWritable(count));
    }
}
```

(3)定义一个主类，用来描述 job 并提交 job

```
public class WordCountRunner {
    //把业务逻辑相关的信息（哪个是 mapper，哪个是 reducer，要处理的数据在哪里，输出的结果放哪里……）描述成一个 job 对象
    //把这个描述好的 job 提交给集群去运行
    public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        Job wcjob = Job.getInstance(conf);
        //指定我这个 job 所在的 jar 包
//        wcjob.setJar("/home/hadoop/wordcount.jar");
        wcjob.setJarByClass(WordCountRunner.class);

        wcjob.setMapperClass(WordCountMapper.class);
        wcjob.setReducerClass(WordCountReducer.class);
        //设置我们的业务逻辑 Mapper 类的输出 key 和 value 的数据类型
        wcjob.setMapOutputKeyClass(Text.class);
        wcjob.setMapOutputValueClass(IntWritable.class);
        //设置我们的业务逻辑 Reducer 类的输出 key 和 value 的数据类型
        wcjob.setOutputKeyClass(Text.class);
        wcjob.setOutputValueClass(IntWritable.class);

        //指定要处理的数据所在的位置
        FileInputFormat.setInputPaths(wcjob, "hdfs://hdp-server01:9000/wordcount/data/big.txt");
        //指定处理完成之后的结果所保存的位置
```

```
        FileOutputFormat.setOutputPath(wcjob, new Path("hdfs://hdp-server01:9000/wordcount/output/"));


        //向 yarn 集群提交这个 job
        boolean res = wcjob.waitForCompletion(true);
        System.exit(res?0:1);
    }
```

5.2 MAPREDUCE 程序运行模式

5.2.1 本地运行模式

（1）mapreduce 程序是被提交给 LocalJobRunner 在本地以单进程的形式运行
（2）而处理的数据及输出结果可以在本地文件系统，也可以在 hdfs 上
（3）怎样实现本地运行？写一个程序，不要带集群的配置文件（本质是你的 mr 程序的 conf 中是否有 mapreduce.framework.name=local 以及 yarn.resourcemanager.hostname 参数）
（4）本地模式非常便于进行业务逻辑的 debug，只要在 eclipse 中打断点即可

如果在 windows 下想运行本地模式来测试程序逻辑，需要在 windows 中配置环境变量：
%HADOOP_HOME% = d:/hadoop-2.6.1
%PATH% = %HADOOP_HOME%\bin
并且要将 d:/hadoop-2.6.1 的 lib 和 bin 目录替换成 windows 平台编译的版本

5.2.2 集群运行模式

（1）将 mapreduce 程序提交给 yarn 集群 resourcemanager，分发到很多的节点上并发执行
（2）处理的数据和输出结果应该位于 hdfs 文件系统
（3）提交集群的实现步骤：

A、将程序打成 JAR 包，然后在集群的任意一个节点上用 hadoop 命令启动

```
$ hadoop jar wordcount.jar cn.zju.bigdata.mrsimple.WordCountDriver inputpath outputpath
```

B、直接在 linux 的 eclipse 中运行 main 方法

（项目中要带参数：mapreduce.framework.name=yarn 以及 yarn 的两个基本配置）

C、如果要在 windows 的 eclipse 中提交 job 给集群，则要修改 YarnRunner 类

5.3. MAPREDUCE 中的序列化

5.3.1 概述

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。。。。），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

5.3.2 Jdk 序列化和 MR 序列化之间的比较

简单代码验证两种序列化机制的差别：

```
public class TestSeri {
    public static void main(String[] args) throws Exception {
        //定义两个 ByteArrayOutputStream，用来接收不同序列化机制的序列化结果
        ByteArrayOutputStream ba = new ByteArrayOutputStream();
        ByteArrayOutputStream ba2 = new ByteArrayOutputStream();

        //定义两个 DataOutputStream，用于将普通对象进行 jdk 标准序列化
        DataOutputStream dout = new DataOutputStream(ba);
        DataOutputStream dout2 = new DataOutputStream(ba2);
        ObjectOutputStream obout = new ObjectOutputStream(dout2);
        //定义两个 bean，作为序列化的源对象
        ItemBeanSer itemBeanSer = new ItemBeanSer(1000L, 89.9f);
        ItemBean itemBean = new ItemBean(1000L, 89.9f);

        //用于比较 String 类型和 Text 类型的序列化差别
        Text atext = new Text("a");
        // atext.write(dout);
        itemBean.write(dout);

        byte[] byteArray = ba.toByteArray();

        //比较序列化结果
        System.out.println(byteArray.length);
        for (byte b : byteArray) {

            System.out.print(b);
            System.out.print(":");
        }

        System.out.println("----------------------");
```

```
        String astr = "a";
        // dout2.writeUTF(astr);
        obout.writeObject(itemBeanSer);

        byte[] byteArray2 = ba2.toByteArray();
        System.out.println(byteArray2.length);
        for (byte b : byteArray2) {
            System.out.print(b);
            System.out.print(":");
        }
    }
}
```

5.3.3 自定义对象实现 MR 中的序列化接口

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,此时，自定义的 bean 实现的接口应该是：

public class FlowBean implements WritableComparable<FlowBean>

需要自己实现的方法是：

```
    /**
     * 反序列化的方法，反序列化时，从流中读取到的各个字段的顺序应该与序列化时写出去的顺序保持一致
     */
    @Override
    public void readFields(DataInput in) throws IOException {

        upflow = in.readLong();
        dflow = in.readLong();
        sumflow = in.readLong();


    }

    /**
     * 序列化的方法
     */
    @Override
    public void write(DataOutput out) throws IOException {
```

```
        out.writeLong(upflow);
        out.writeLong(dflow);
        //可以考虑不序列化总流量，因为总流量是可以通过上行流量和下行流量计算出来的
        out.writeLong(sumflow);

    }

    @Override
    public int compareTo(FlowBean o) {

        //实现按照 sumflow 的大小倒序排序
        return sumflow>o.getSumflow()?-1:1;
    }
```

6. MAPREDUCE 实践篇（2）

6.1. Mapreduce 中的排序初步

6.1.1 需求

对日志数据中的上下行流量信息汇总，并输出按照总流量倒序排序的结果
数据如下：

1363157985066	13726230503	00-FD-07-A4-72-B8:CMCC	120.196.100.82	24	27	2481	24681	200
1363157995052	13826544101	5C-0E-8B-C7-F1-E0:CMCC	120.197.40.4	4	0	264	0	200
1363157991076	13926435656	20-10-7A-28-CC-0A:CMCC	120.196.100.99	2	4	132	1512	200
1363154400022	13926251106	5C-0E-8B-8B-B1-50:CMCC	120.197.40.4	4	0	240	0	200

6.1.2 分析

基本思路：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

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

6.1.3 实现

1、 自定义的 bean

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

    long upflow;
    long downflow;
    long sumflow;

    //如果空参构造函数被覆盖，一定要显示定义一下，否则在反序列时会抛异常
    public FlowBean(){}

    public FlowBean(long upflow, long downflow) {
        super();
        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，将对象的字段信息写入输出流
    @Override
```

```
    public void write(DataOutput out) throws IOException {

        out.writeLong(upflow);
        out.writeLong(downflow);
        out.writeLong(sumflow);

    }

    //反序列化，从输入流中读取各个字段信息
    @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) {
        //自定义倒序比较规则
        return sumflow > o.getSumflow() ? -1:1;
    }
}
```

2、 mapper 和 reducer

```
public class FlowCount {

    static class FlowCountMapper extends Mapper<LongWritable, Text, FlowBean,Text > {

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

            String line = value.toString();
            String[] fields = line.split("\t");
            try {
                String phonenbr = fields[0];
```

```
                long upflow = Long.parseLong(fields[1]);
                long dflow = Long.parseLong(fields[2]);

                FlowBean flowBean = new FlowBean(upflow, dflow);

                context.write(flowBean,new Text(phonenbr));
            } catch (Exception e) {

                e.printStackTrace();
            }

        }

    }

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

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

            Text phoneNbr = phonenbr.iterator().next();

            context.write(phoneNbr, bean);

        }

    }

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

        Configuration conf = new Configuration();

        Job job = Job.getInstance(conf);

        job.setJarByClass(FlowCount.class);

        job.setMapperClass(FlowCountMapper.class);
        job.setReducerClass(FlowCountReducer.class);

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

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

        // job.setInputFormatClass(TextInputFormat.class);

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

        job.waitForCompletion(true);

    }

}
```

6.2. Mapreduce 中的分区 Partitioner

6.2.1 需求

根据归属地输出流量统计数据结果到不同文件，以便于在查询统计结果时可以定位到省级范围进行

6.2.2 分析

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
所以：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
然后在 job 对象中，设置自定义 partitioner： job.setPartitionerClass(CustomPartitioner.class)

6.2.3 实现

```
/**
 * 定义自己的从 map 到 reduce 之间的数据（分组）分发规则 按照手机号所属的省份来分发（分组）
ProvincePartitioner
 * 默认的分组组件是 HashPartitioner
 *
 * @author
```

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

	static HashMap<String, Integer> provinceMap = new HashMap<String, Integer>();

	static {

		provinceMap.put("135", 0);
		provinceMap.put("136", 1);
		provinceMap.put("137", 2);
		provinceMap.put("138", 3);
		provinceMap.put("139", 4);

	}

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

		Integer code = provinceMap.get(key.toString().substring(0, 3));

		return code == null ? 5 : code;
	}

}
```

6.3. mapreduce 数据压缩

6.3.1 概述

这是 **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）

1、 Mapreduce 支持将 map 输出的结果或者 reduce 输出的结果进行压缩，以减少网络 IO 或最终输出数据的体积

2、 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

3、 基本原则：

运算密集型的 job，少用压缩

IO 密集型的 job，多用压缩

6.3.2 MR 支持的压缩编码

Compression format	Tool	Algorithm	Filename extension	Splittable?
DEFLATE[a]	N/A	DEFLATE	*.deflate*	No
gzip	*gzip*	DEFLATE	*.gz*	No
bzip2	*bzip2*	bzip2	*.bz2*	Yes
LZO	*lzop*	LZO	*.lzo*	No[b]
LZ4	N/A	LZ4	*.lz4*	No
Snappy	N/A	Snappy	*.snappy*	No

6.3.3 Reducer 输出压缩

在配置参数或在代码中都可以设置 reduce 的输出压缩

1、在配置参数中设置

mapreduce.output.fileoutputformat.compress=false

mapreduce.output.fileoutputformat.compress.codec=org.apache.hadoop.io.compress.DefaultCodec

mapreduce.output.fileoutputformat.compress.type=RECORD

2、在代码中设置

```
Job job = Job.getInstance(conf);
FileOutputFormat.setCompressOutput(job, true);
FileOutputFormat.setOutputCompressorClass(job, (Class<? extends CompressionCodec>) Class.forName(""));
```

6.3.4 Mapper 输出压缩

在配置参数或在代码中都可以设置 reduce 的输出压缩

1、在配置参数中设置

mapreduce.map.output.compress=false

mapreduce.map.output.compress.codec=org.apache.hadoop.io.compress.DefaultCodec

2、在代码中设置：

```
conf.setBoolean(Job.MAP_OUTPUT_COMPRESS, true);
conf.setClass(Job.MAP_OUTPUT_COMPRESS_CODEC, GzipCodec.class, CompressionCodec.class);
```

6.3.5 压缩文件的读取

Hadoop 自带的 InputFormat 类内置支持压缩文件的读取，比如 TextInputformat 类，在其 initialize 方法中：

```
public void initialize(InputSplit genericSplit,
                       TaskAttemptContext context) throws IOException {
  FileSplit split = (FileSplit) genericSplit;
  Configuration job = context.getConfiguration();
  this.maxLineLength = job.getInt(MAX_LINE_LENGTH, Integer.MAX_VALUE);
  start = split.getStart();
  end = start + split.getLength();
  final Path file = split.getPath();

  // open the file and seek to the start of the split
  final FileSystem fs = file.getFileSystem(job);
  fileIn = fs.open(file);
  //根据文件后缀名创建相应压缩编码的 codec
  CompressionCodec codec = new CompressionCodecFactory(job).getCodec(file);
  if (null!=codec) {
    isCompressedInput = true;
    decompressor = CodecPool.getDecompressor(codec);
     //判断是否属于可切片压缩编码类型
    if (codec instanceof SplittableCompressionCodec) {
      final SplitCompressionInputStream cIn =
        ((SplittableCompressionCodec)codec).createInputStream(
          fileIn, decompressor, start, end,
          SplittableCompressionCodec.READ_MODE.BYBLOCK);
       //如果是可切片压缩编码，则创建一个 CompressedSplitLineReader 读取压缩数据
      in = new CompressedSplitLineReader(cIn, job,
          this.recordDelimiterBytes);
      start = cIn.getAdjustedStart();
      end = cIn.getAdjustedEnd();
      filePosition = cIn;
    } else {
       //如果是不可切片压缩编码，则创建一个 SplitLineReader 读取压缩数据，并将文件输入流转换成解压数据流传递给普通 SplitLineReader 读取
      in = new SplitLineReader(codec.createInputStream(fileIn,
          decompressor), job, this.recordDelimiterBytes);
      filePosition = fileIn;
    }
  } else {
    fileIn.seek(start);
     //如果不是压缩文件，则创建普通 SplitLineReader 读取数据
```

```
        in = new SplitLineReader(fileIn, job, this.recordDelimiterBytes);
        filePosition = fileIn;
    }
```

6.4. 更多 MapReduce 编程案例

6.4.1 reduce 端 join 算法实现

1、需求：

订单数据表 t_order：

Id	date	pid	amount
1001	20150710	P0001	2
1002	20150710	P0001	3
1002	20150710	P0002	3

商品信息表 t_product

Id	name	category_id	price
P0001	小米 5	C01	2
P0002	锤子 T1	C01	3

假如数据量巨大，两表的数据是以文件的形式存储在 HDFS 中，需要用 mapreduce 程序来实现一下 SQL 查询运算：

```
select    a.id,a.date,b.name,b.category_id,b.price from t_order a join t_product b on a.pid = b.id
```

2、实现机制：

通过将关联的条件作为 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

```
public class OrderJoin {

    static class OrderJoinMapper extends Mapper<LongWritable, Text, Text, OrderJoinBean> {

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

            // 拿到一行数据，并且要分辨出这行数据所属的文件
            String line = value.toString();

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

            // 拿到 itemid
            String itemid = fields[0];

            // 获取到这一行所在的文件名（通过 inpusplit）
```

```
                String name = "你拿到的文件名";

                // 根据文件名，切分出各字段（如果是 a，切分出两个字段，如果是 b，切分出 3 个字段）

                OrderJoinBean bean = new OrderJoinBean();
                bean.set(null, null, null, null, null);
                context.write(new Text(itemid), bean);

            }

        }

        static class OrderJoinReducer extends Reducer<Text, OrderJoinBean, OrderJoinBean, NullWritable> {

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

                 //拿到的 key 是某一个 itemid,比如 1000
                //拿到的 beans 是来自于两类文件的 bean
                //   {1000,amount} {1000,amount} {1000,amount}     ---     {1000,price,name}

                //将来自于 b 文件的 bean 里面的字段，跟来自于 a 的所有 bean 进行字段拼接并输出
            }
        }
}
```

缺点：这种方式中，join 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 join 实现方式

6.4.2 map 端 join 算法实现

1、原理阐述

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 join 并输出最终结果，可以大大提高 join 操作的并发度，加快处理速度

2、实现示例

--先在 mapper 类中预先定义好小表，进行 join

--引入实际场景中的解决方案：一次加载数据库或者用 distributedcache

```
public class TestDistributedCache {
    static class TestDistributedCacheMapper extends Mapper<LongWritable, Text, Text, Text>{
        FileReader in = null;
        BufferedReader reader = null;
        HashMap<String,String> b_tab = new HashMap<String, String>();
        String localpath =null;
        String uirpath = null;

        //是在 map 任务初始化的时候调用一次
        @Override
        protected void setup(Context context) throws IOException, InterruptedException {
            //通过这几句代码可以获取到 cache file 的本地绝对路径，测试验证用
            Path[] files = context.getLocalCacheFiles();
            localpath = files[0].toString();
            URI[] cacheFiles = context.getCacheFiles();


            //缓存文件的用法——直接用本地 IO 来读取
            //这里读的数据是 map task 所在机器本地工作目录中的一个小文件
            in = new FileReader("b.txt");
            reader =new BufferedReader(in);
            String line =null;
            while(null!=(line=reader.readLine())){

                String[] fields = line.split(",");
                b_tab.put(fields[0],fields[1]);

            }
            IOUtils.closeStream(reader);
            IOUtils.closeStream(in);

        }

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

            //这里读的是这个 map task 所负责的那一个切片数据（在 hdfs 上）
             String[] fields = value.toString().split("\t");
```

```
                String a_itemid = fields[0];
                String a_amount = fields[1];

                String b_name = b_tab.get(a_itemid);

                // 输出结果   1001    98.9 banan
                context.write(new Text(a_itemid), new Text(a_amount + "\t" + ":" + localpath + "\t" +b_name ));

        }


    }


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

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

        job.setJarByClass(TestDistributedCache.class);

        job.setMapperClass(TestDistributedCacheMapper.class);

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

        //这里是我们正常的需要处理的数据所在路径
        FileInputFormat.setInputPaths(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        //不需要 reducer
        job.setNumReduceTasks(0);
        //分发一个文件到 task 进程的工作目录
        job.addCacheFile(new URI("hdfs://hadoop-server01:9000/cachefile/b.txt"));

        //分发一个归档文件到 task 进程的工作目录
//        job.addArchiveToClassPath(archive);

        //分发 jar 包到 task 节点的 classpath 下
//        job.addFileToClassPath(jarfile);

        job.waitForCompletion(true);
    }
```

```
}
```

6.4.3 web 日志预处理

1、需求：
对 web 访问日志中的各字段识别切分
去除日志中不合法的记录
根据 KPI 统计需求，生成各类访问请求过滤数据

2、实现代码：
a) 定义一个 bean，用来记录日志数据中的各数据字段

```
public class WebLogBean {

    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();
        }
}
```

b)定义一个 parser 用来解析过滤 web 访问日志原始记录

```
public class WebLogParser {
        public static WebLogBean parser(String line) {
            WebLogBean webLogBean = new WebLogBean();
            String[] arr = line.split(" ");
            if (arr.length > 11) {
              webLogBean.setRemote_addr(arr[0]);
              webLogBean.setRemote_user(arr[1]);
              webLogBean.setTime_local(arr[3].substring(1));
              webLogBean.setRequest(arr[6]);
              webLogBean.setStatus(arr[8]);
              webLogBean.setBody_bytes_sent(arr[9]);
              webLogBean.setHttp_referer(arr[10]);

                  if (arr.length > 12) {
                     webLogBean.setHttp_user_agent(arr[11] + " " + arr[12]);
```

```
                } else {
                    webLogBean.setHttp_user_agent(arr[11]);
                }
                if (Integer.parseInt(webLogBean.getStatus()) >= 400) {// 大于 400，HTTP 错误
                    webLogBean.setValid(false);
                }
            } else {
              webLogBean.setValid(false);
            }
            return webLogBean;
        }

        public static String parserTime(String time) {

         time.replace("/", "-");
         return time;

        }
}
```

c) mapreduce 程序

```
public class WeblogPreProcess {

        static class WeblogPreProcessMapper extends Mapper<LongWritable, Text, Text, NullWritable> {
                Text k = new Text();
                NullWritable v = NullWritable.get();

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

                        String line = value.toString();
                        WebLogBean webLogBean = WebLogParser.parser(line);
                        if (!webLogBean.isValid())
                                return;
                        k.set(webLogBean.toString());
                        context.write(k, v);

                }

        }

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

```
        Configuration conf = new Configuration();
        Job job = Job.getInstance(conf);

        job.setJarByClass(WeblogPreProcess.class);

        job.setMapperClass(WeblogPreProcessMapper.class);

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

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

        job.waitForCompletion(true);

    }
}
```

流量统计相关需求

1、对流量日志中的用户统计总上、下行流量
技术点： 自定义 javaBean 用来在 mapreduce 中充当 value
注意： javaBean 要实现 Writable 接口，实现两个方法

```
//序列化，将对象的字段信息写入输出流
@Override
public void write(DataOutput out) throws IOException {

	out.writeLong(upflow);
	out.writeLong(downflow);
	out.writeLong(sumflow);

}

//反序列化，从输入流中读取各个字段信息
@Override
public void readFields(DataInput in) throws IOException {
	upflow = in.readLong();
	downflow = in.readLong();
	sumflow = in.readLong();

}
```

2、统计流量且按照流量大小倒序排序
技术点：这种需求，用一个 mapreduce -job 不好实现，需要两个 mapreduce -job
第一个 job 负责流量统计，跟上题相同
第二个 job 读入第一个 job 的输出，然后做排序
要将 flowBean 作为 map 的 key 输出，这样 mapreduce 就会自动排序
此时，flowBean 要实现接口 WritableComparable
要实现其中的 compareTo()方法，方法中，我们可以定义倒序比较的逻辑

3、统计流量且按照手机号的归属地，将结果数据输出到不同的省份文件中
技术点：自定义 Partitioner

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

	String prefix = key.toString().substring(0,3);
	Integer partNum = pmap.get(prefix);

	return (partNum==null?4:partNum);
```

```
    }
```

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);
```

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

社交粉丝数据分析

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

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

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？
解题思路：

```
第一步
map
读一行     A:B,C,D,F,E,O
输出       <B,A><C,A><D,A><F,A><E,A><O,A>
在读一行     B:A,C,E,K
输出      <A,B><C,B><E,B><K,B>


REDUCE
```

拿到的数据比如<C,A><C,B><C,E><C,F><C,G>.....
输出：
<A-B,C>
<A-E,C>
<A-F,C>
<A-G,C>
<B-E,C>
<B-F,C>.....

第二步
map
读入一行<A-B,C>
直接输出<A-B,C>

reduce
读入数据　<A-B,C><A-B,F><A-B,G>.....
输出：　A-B　C,F,G,.....

扩展：求互粉的人!!!!

倒排索引建立

需求：有大量的文本（文档、网页），需要建立搜索索引

1. 自定义 inputFormat

1.1 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

1.2 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 combineInputFormat 提高效率

1.3 实现

本节实现的是上述第二种方式
程序的核心机制：

自定义一个 InputFormat
改写 RecordReader，实现一次读取一个完整文件封装为 KV
在输出时使用 SequenceFileOutPutFormat 输出合并文件

代码如下：
自定义 InputFromat

```
public class WholeFileInputFormat extends
        FileInputFormat<NullWritable, BytesWritable> {
    //设置每个小文件不可分片,保证一个小文件生成一个 key-value 键值对
    @Override
    protected boolean isSplitable(JobContext context, Path file) {
        return false;
    }

    @Override
    public RecordReader<NullWritable, BytesWritable> createRecordReader(
            InputSplit split, TaskAttemptContext context) throws IOException,
            InterruptedException {
        WholeFileRecordReader reader = new WholeFileRecordReader();
        reader.initialize(split, context);
```

```
            return reader;
        }
}
```

自定义 RecordReader

```
class WholeFileRecordReader extends RecordReader<NullWritable, BytesWritable> {
        private FileSplit fileSplit;
        private Configuration conf;
        private BytesWritable value = new BytesWritable();
        private boolean processed = false;

        @Override
        public void initialize(InputSplit split, TaskAttemptContext context)
                        throws IOException, InterruptedException {
                this.fileSplit = (FileSplit) split;
                this.conf = context.getConfiguration();
        }

        @Override
        public boolean nextKeyValue() throws IOException, InterruptedException {
                if (!processed) {
                        byte[] contents = new byte[(int) fileSplit.getLength()];
                        Path file = fileSplit.getPath();
                        FileSystem fs = file.getFileSystem(conf);
                        FSDataInputStream in = null;
                        try {
                                in = fs.open(file);
                                IOUtils.readFully(in, contents, 0, contents.length);
                                value.set(contents, 0, contents.length);
                        } finally {
                                IOUtils.closeStream(in);
                        }
                        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 {
        return processed ? 1.0f : 0.0f;
    }

    @Override
    public void close() throws IOException {
        // do nothing
    }
}
```

定义 mapreduce 处理流程

```
public class SmallFilesToSequenceFileConverter extends Configured implements
        Tool {
    static class SequenceFileMapper extends
            Mapper<NullWritable, BytesWritable, Text, BytesWritable> {
        private Text filenameKey;

        @Override
        protected void setup(Context context) throws IOException,
                InterruptedException {
            InputSplit split = context.getInputSplit();
            Path path = ((FileSplit) split).getPath();
            filenameKey = new Text(path.toString());
        }

        @Override
        protected void map(NullWritable key, BytesWritable value,
                Context context) throws IOException, InterruptedException {
            context.write(filenameKey, value);
        }
    }

    @Override
    public int run(String[] args) throws Exception {
        Configuration conf = new Configuration();
        System.setProperty("HADOOP_USER_NAME", "hdfs");
```

```
        String[] otherArgs = new GenericOptionsParser(conf, args)
                .getRemainingArgs();
        if (otherArgs.length != 2) {
            System.err.println("Usage: combinefiles <in> <out>");
            System.exit(2);
        }

        Job job = Job.getInstance(conf,"combine small files to sequencefile");
//      job.setInputFormatClass(WholeFileInputFormat.class);
        job.setOutputFormatClass(SequenceFileOutputFormat.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(BytesWritable.class);
        job.setMapperClass(SequenceFileMapper.class);
        return job.waitForCompletion(true) ? 0 : 1;
    }

    public static void main(String[] args) throws Exception {
        int exitCode = ToolRunner.run(new SmallFilesToSequenceFileConverter(),
                args);
        System.exit(exitCode);

    }
}
```

2. 自定义 outputFormat

2.1 需求

现有一些原始日志需要做增强解析处理，流程：

1、 从原始日志文件中读取数据

2、 根据日志中的一个 URL 字段到外部知识库中获取信息增强到原始日志

3、 如果成功增强，则输出到增强结果目录；如果增强失败，则抽取原始数据中 URL 字段输出到待爬清单目录

2.2 分析

程序的关键点是要在一个 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

2.3 实现

实现要点：

1、 在 mapreduce 中访问外部资源

2、 自定义 outputformat，改写其中的 recordwriter，改写具体输出数据的方法 write()

代码实现如下：

数据库获取数据的工具

```
public class DBLoader {

    public static void dbLoader(HashMap<String, String> ruleMap) {
        Connection conn = null;
        Statement st = null;
        ResultSet res = null;

        try {
            Class.forName("com.mysql.jdbc.Driver");
            conn =
DriverManager.getConnection("jdbc:mysql://hdp-node01:3306/urlknowledge", "root", "root");
            st = conn.createStatement();
            res = st.executeQuery("select url,content from urlcontent");
            while (res.next()) {
                ruleMap.put(res.getString(1), res.getString(2));
            }
        } catch (Exception e) {
            e.printStackTrace();

        } finally {
            try{
                if(res!=null){
                    res.close();
                }
                if(st!=null){
                    st.close();
                }
                if(conn!=null){
                    conn.close();
```

```
                }

            }catch(Exception e){
                e.printStackTrace();
            }
        }
    }


    public static void main(String[] args) {
        DBLoader db = new DBLoader();
        HashMap<String, String> map = new HashMap<String,String>();
        db.dbLoader(map);
        System.out.println(map.size());
    }
}
```

自定义一个 outputformat

```
public class LogEnhancerOutputFormat extends FileOutputFormat<Text, NullWritable>{



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



        FileSystem fs = FileSystem.get(context.getConfiguration());
        Path enhancePath = new Path("hdfs://hdp-node01:9000/flow/enhancelog/enhanced.log");
        Path toCrawlPath = new Path("hdfs://hdp-node01:9000/flow/tocrawl/tocrawl.log");

        FSDataOutputStream enhanceOut = fs.create(enhancePath);
        FSDataOutputStream toCrawlOut = fs.create(toCrawlPath);



        return new MyRecordWriter(enhanceOut,toCrawlOut);
    }



    static class MyRecordWriter extends RecordWriter<Text, NullWritable>{

        FSDataOutputStream enhanceOut = null;
```

```
        FSDataOutputStream toCrawlOut = null;

        public MyRecordWriter(FSDataOutputStream enhanceOut, FSDataOutputStream
toCrawlOut) {
            this.enhanceOut = enhanceOut;
            this.toCrawlOut = toCrawlOut;
        }

        @Override
        public void write(Text key, NullWritable value) throws IOException,
InterruptedException {

            //有了数据，你来负责写到目的地 —— hdfs
            //判断，进来内容如果是带 tocrawl 的，就往待爬清单输出流中写 toCrawlOut
            if(key.toString().contains("tocrawl")){
                toCrawlOut.write(key.toString().getBytes());
            }else{
                enhanceOut.write(key.toString().getBytes());
            }

        }

        @Override
        public void close(TaskAttemptContext context) throws IOException,
InterruptedException {

            if(toCrawlOut!=null){
                toCrawlOut.close();
            }
            if(enhanceOut!=null){
                enhanceOut.close();
            }

        }


    }
}
```

开发 mapreduce 处理流程

```
/**
 * 这个程序是对每个小时不断产生的用户上网记录日志进行增强(将日志中的 url 所指向的
网页内容分析结果信息追加到每一行原始日志后面)
 *
```

```
 * @author
 *
 */
public class LogEnhancer {

    static class LogEnhancerMapper extends Mapper<LongWritable, Text, Text, NullWritable> {

        HashMap<String, String> knowledgeMap = new HashMap<String, String>();

        /**
         * maptask 在初始化时会先调用 setup 方法一次 利用这个机制，将外部的知识库加载到 maptask 执行的机器内存中
         */
        @Override
        protected void setup(org.apache.hadoop.mapreduce.Mapper.Context context) throws IOException, InterruptedException {

            DBLoader.dbLoader(knowledgeMap);

        }

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

            String line = value.toString();

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

            try {
                String url = fields[26];

                // 对这一行日志中的 url 去知识库中查找内容分析信息
                String content = knowledgeMap.get(url);

                // 根据内容信息匹配的结果，来构造两种输出结果
                String result = "";
                if (null == content) {
                    // 输往待爬清单的内容
                    result = url + "\t" + "tocrawl\n";
                } else {
                    // 输往增强日志的内容
                    result = line + "\t" + content + "\n";
                }
```

```
                    context.write(new Text(result), NullWritable.get());
                } catch (Exception e) {

                }
            }

    }

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

        Configuration conf = new Configuration();

        Job job = Job.getInstance(conf);

        job.setJarByClass(LogEnhancer.class);

        job.setMapperClass(LogEnhancerMapper.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(LogEnhancerOutputFormat.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]));

        job.waitForCompletion(true);
        System.exit(0);

    }

}
```

3. 自定义 GroupingComparator

3.1 需求

有如下订单数据

订单 id	商品 id	成交金额
Order_0000001	Pdt_01	222.8
Order_0000001	Pdt_05	25.8
Order_0000002	Pdt_03	522.8
Order_0000002	Pdt_04	122.4
Order_0000003	Pdt_01	222.8

现在需要求出每一个订单中成交金额最大的一笔交易

3.2 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

3.3 实现

自定义 groupingcomparator

```
/**
 * 用于控制 shuffle 过程中 reduce 端对 kv 对的聚合逻辑
 * @author
 *
 */
public class ItemidGroupingComparator extends WritableComparator {

    protected ItemidGroupingComparator() {

        super(OrderBean.class, true);
    }
```

```
	@Override
	public int compare(WritableComparable a, WritableComparable b) {
		OrderBean abean = (OrderBean) a;
		OrderBean bbean = (OrderBean) b;

		//将 item_id 相同的 bean 都视为相同，从而聚合为一组
		return abean.getItemid().compareTo(bbean.getItemid());
	}
}
```

定义订单信息 bean

```
/**
 * 订单信息 bean，实现 hadoop 的序列化机制
 * @author
 *
 */
public class OrderBean implements WritableComparable<OrderBean>{
	private Text itemid;
	private DoubleWritable amount;

	public OrderBean() {
	}
	public OrderBean(Text itemid, DoubleWritable amount) {
		set(itemid, amount);
	}

	public void set(Text itemid, DoubleWritable amount) {

		this.itemid = itemid;
		this.amount = amount;

	}

	public Text getItemid() {
		return itemid;
	}

	public DoubleWritable getAmount() {
		return amount;
	}

	@Override
	public int compareTo(OrderBean o) {
```

```
            int cmp = this.itemid.compareTo(o.getItemid());
            if (cmp == 0) {

                cmp = -this.amount.compareTo(o.getAmount());
            }
            return cmp;
        }

        @Override
        public void write(DataOutput out) throws IOException {
            out.writeUTF(itemid.toString());
            out.writeDouble(amount.get());

        }

        @Override
        public void readFields(DataInput in) throws IOException {
            String readUTF = in.readUTF();
            double readDouble = in.readDouble();

            this.itemid = new Text(readUTF);
            this.amount= new DoubleWritable(readDouble);
        }



        @Override
        public String toString() {
            return itemid.toString() + "\t" + amount.get();
        }
}
```

编写 mapreduce 处理流程

```
/**
 * 利用 secondarysort 机制输出每种 item 订单金额最大的记录
 * @author
 *
 */
public class SecondarySort {

    static class SecondarySortMapper extends Mapper<LongWritable, Text, OrderBean, NullWritable>{

        OrderBean bean = new OrderBean();
```

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

            String line = value.toString();
            String[] fields = StringUtils.split(line, "\t");

            bean.set(new Text(fields[0]), new DoubleWritable(Double.parseDouble(fields[1])));

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

        }

    }

    static class SecondarySortReducer extends Reducer<OrderBean, NullWritable, OrderBean, NullWritable>{


        //在设置了 groupingcomparator 以后，这里收到的 kv 数据 就是： <1001 87.6>,null <1001 76.5>,null ....
        //此时，reduce 方法中的参数 key 就是上述 kv 组中的第一个 kv 的 key：<1001 87.6>
        //要输出同一个 item 的所有订单中最大金额的那一个，就只要输出这个 key
        @Override
        protected void reduce(OrderBean key, Iterable<NullWritable> values, Context context) throws IOException, InterruptedException {
            context.write(key, NullWritable.get());
        }
    }


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

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

        job.setJarByClass(SecondarySort.class);

        job.setMapperClass(SecondarySortMapper.class);
        job.setReducerClass(SecondarySortReducer.class);


        job.setOutputKeyClass(OrderBean.class);
```

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

        FileInputFormat.setInputPaths(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));
        //指定 shuffle 所使用的 GroupingComparator 类
        job.setGroupingComparatorClass(ItemidGroupingComparator.class);
        //指定 shuffle 所使用的 partitioner 类
        job.setPartitionerClass(ItemIdPartitioner.class);

        job.setNumReduceTasks(3);

        job.waitForCompletion(true);

    }

}
```

4. Mapreduce 中的 DistributedCache 应用

4.1 Map 端 join 案例

4.1.1 需求

实现两个“表”的 join 操作，其中一个表数据量小，一个表很大，这种场景在实际中非常常见，比如“订单日志” join “产品信息”

4.1.2 分析

--原理阐述
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 join 并输出最终结果
可以大大提高 join 操作的并发度，加快处理速度

--示例：先在 mapper 类中预先定义好小表，进行 join
--并用 distributedcache 机制将小表的数据分发到每一个 maptask 执行节点，从而每一个

maptask 节点可以从本地加载到小表的数据，进而在本地即可实现 join

4.1.3 实现

```
public class TestDistributedCache {
    static class TestDistributedCacheMapper extends Mapper<LongWritable, Text, Text, Text>{
        FileReader in = null;
        BufferedReader reader = null;
        HashMap<String,String> b_tab = new HashMap<String, String>();
        String localpath =null;
        String uirpath = null;

        //是在 map 任务初始化的时候调用一次
        @Override
        protected void setup(Context context) throws IOException, InterruptedException {
            //通过这几句代码可以获取到 cache file 的本地绝对路径，测试验证用
            Path[] files = context.getLocalCacheFiles();
            localpath = files[0].toString();
            URI[] cacheFiles = context.getCacheFiles();


            //缓存文件的用法——直接用本地 IO 来读取
            //这里读的数据是 map task 所在机器本地工作目录中的一个小文件
            in = new FileReader("b.txt");
            reader =new BufferedReader(in);
            String line =null;
            while(null!=(line=reader.readLine())){

                String[] fields = line.split(",");
                b_tab.put(fields[0],fields[1]);

            }
            IOUtils.closeStream(reader);
            IOUtils.closeStream(in);

        }

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

            //这里读的是这个 map task 所负责的那一个切片数据（在 hdfs 上）
             String[] fields = value.toString().split("\t");
```

```
                String a_itemid = fields[0];
                String a_amount = fields[1];

                String b_name = b_tab.get(a_itemid);

                // 输出结果   1001    98.9 banan
                context.write(new Text(a_itemid), new Text(a_amount + "\t" + ":" + localpath +
"\t" +b_name ));

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

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

            job.setJarByClass(TestDistributedCache.class);

            job.setMapperClass(TestDistributedCacheMapper.class);

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

            //这里是我们正常的需要处理的数据所在路径
            FileInputFormat.setInputPaths(job, new Path(args[0]));
            FileOutputFormat.setOutputPath(job, new Path(args[1]));

            //不需要 reducer
            job.setNumReduceTasks(0);
            //分发一个文件到 task 进程的工作目录
            job.addCacheFile(new URI("hdfs://hadoop-server01:9000/cachefile/b.txt"));

            //分发一个归档文件到 task 进程的工作目录
//          job.addArchiveToClassPath(archive);

            //分发 jar 包到 task 节点的 classpath 下
//          job.addFileToClassPath(jarfile);

            job.waitForCompletion(true);
        }
}
```

5. Mapreduce 的其他补充

5.1 计数器应用

在实际生产代码中，常常需要将数据处理过程中遇到的不合规数据行进行全局计数，类似这种需求可以借助 mapreduce 框架中提供的全局计数器来实现
示例代码如下：

```
public class MultiOutputs {
    //通过枚举形式定义自定义计数器
    enum MyCounter{MALFORORMED,NORMAL}

    static class CommaMapper extends Mapper<LongWritable, Text, Text, LongWritable> {

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

            String[] words = value.toString().split(",");

            for (String word : words) {
                context.write(new Text(word), new LongWritable(1));
            }
            //对枚举定义的自定义计数器加 1
            context.getCounter(MyCounter.MALFORORMED).increment(1);
            //通过动态设置自定义计数器加 1
            context.getCounter("counterGroupa", "countera").increment(1);
        }

    }
```

5.2 多 job 串联

一个稍复杂点的处理逻辑往往需要多个 mapreduce 程序串联处理，多 job 的串联可以借助 mapreduce 框架的 JobControl 实现

示例代码：

```
ControlledJob cJob1 = new ControlledJob(job1.getConfiguration());
   ControlledJob cJob2 = new ControlledJob(job2.getConfiguration());
   ControlledJob cJob3 = new ControlledJob(job3.getConfiguration());

   // 设置作业依赖关系
   cJob2.addDependingJob(cJob1);
   cJob3.addDependingJob(cJob2);

   JobControl jobControl = new JobControl("RecommendationJob");
   jobControl.addJob(cJob1);
   jobControl.addJob(cJob2);
   jobControl.addJob(cJob3);

   cJob1.setJob(job1);
   cJob2.setJob(job2);
   cJob3.setJob(job3);

   // 新建一个线程来运行已加入 JobControl 中的作业，开始进程并等待结束
   Thread jobControlThread = new Thread(jobControl);
   jobControlThread.start();
   while (!jobControl.allFinished()) {
         Thread.sleep(500);
   }
   jobControl.stop();

   return 0;
```

5.3 Configuration 对象高级应用

6. mapreduce 参数优化

MapReduce 重要配置参数

11.1 资源相关参数

(1) 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。
(2) 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。
(3) mapreduce.map.java.opts: Map Task 的 JVM 参数，你可以在此配置默认的 java heap size 等参数, e.g.
“-Xmx1024m -verbose:gc -Xloggc:/tmp/@taskid@.gc” （@taskid@会被 Hadoop 框架自动换为相应的 taskid）, 默认值: “”
(4) mapreduce.reduce.java.opts: Reduce Task 的 JVM 参数，你可以在此配置默认的 java heap size 等参数, e.g.
“-Xmx1024m -verbose:gc -Xloggc:/tmp/@taskid@.gc”, 默认值: “”
(5) mapreduce.map.cpu.vcores: 每个 Map task 可使用的最多 cpu core 数目, 默认值: 1
(6) mapreduce.map.cpu.vcores: 每个 Reduce task 可使用的最多 cpu core 数目, 默认值: 1

11.2 容错相关参数

(1) 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。
(2) 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。
(3) mapreduce.map.failures.maxpercent: 当失败的 Map Task 失败比例超过该值为，整个作业则失败，默认值为 0. 如果你的应用程序允许丢弃部分输入数据，则该该值设为一个大于 0 的值，比如 5，表示如果有低于 5%的 Map Task 失败（如果一个 Map Task 重试次数超过 mapreduce.map.maxattempts，则认为这个 Map Task 失败，其对应的输入数据将不会产生任何结果），整个作业扔认为成功。
(4) mapreduce.reduce.failures.maxpercent: 当失败的 Reduce Task 失败比例超过该值为，整个作业则失败，默认值为 0.
(5) 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 300000。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.”。

11.3 本地运行 mapreduce 作业

设置以下几个参数:
mapreduce.framework.name=local
mapreduce.jobtracker.address=local
fs.defaultFS=local

11.4 效率和稳定性相关参数

(1) mapreduce.map.speculative: 是否为 Map Task 打开推测执行机制，默认为 false
(2) mapreduce.reduce.speculative: 是否为 Reduce Task 打开推测执行机制，默认为 false
(3) mapreduce.job.user.classpath.first & mapreduce.task.classpath.user.precedence：当同一个 class 同时出现在用户 jar 包和 hadoop jar 中时，优先使用哪个 jar 包中的 class，默认为 false，表示优先使用 hadoop jar 中的 class。
(4) mapreduce.input.fileinputformat.split.minsize: 每个 Map Task 处理的数据量（仅针对基于文件的 Inputformat 有效，比如 TextInputFormat，SequenceFileInputFormat），默认为一个 block 大小，即 134217728。