

Hadoop 集群（第 9 期）

——MapReduce 初级案例

1、数据去重

“**数据去重**”主要是为了掌握和利用**并行化思想**来对数据进行**有意义的筛选**。**统计大数据集上的数据种类个数、从网站日志中计算访问地**等这些看似庞杂的任务都会涉及数据去重。下面就进入这个实例的 MapReduce 程序设计。

1.1 实例描述

对数据文件中的数据进行去重。数据文件中的每行都是一个数据。

样例**输入**如下所示：

1) file1:

```
2012-3-1 a
2012-3-2 b
2012-3-3 c
2012-3-4 d
2012-3-5 a
2012-3-6 b
2012-3-7 c
2012-3-3 c
```

2) file2:

```
2012-3-1 b
2012-3-2 a
2012-3-3 b
2012-3-4 d
2012-3-5 a
2012-3-6 c
2012-3-7 d
2012-3-3 c
```

样例**输出**如下所示：

```
2012-3-1 a
2012-3-1 b
2012-3-2 a
```



```
2012-3-2 b
2012-3-3 b
2012-3-3 c
2012-3-4 d
2012-3-5 a
2012-3-6 b
2012-3-6 c
2012-3-7 c
2012-3-7 d
```

1.2 设计思路

数据去重的最终目标是让**原始数据**中出现**次数超过一次**的数据在**输出文件**中**只出现一次**。我们自然而然会想到将同一个数据的所有记录都交给**一台** reduce 机器，无论这个数据出现多少次，只要在最终结果中输出一次就可以了。具体就是 reduce 的**输入**应该以**数据**作为 **key**，而对 value-list 则**没有**要求。当 reduce 接收到一个<key, value-list>时就**直接**将 key 复制到输出的 key 中，并将 value 设置成**空值**。

在 MapReduce 流程中，map 的输出<key, value>经过 shuffle 过程聚集成<key, value-list>后会交给 reduce。所以从设计好的 reduce 输入可以反推出 map 的输出 key 应为数据，value 任意。继续反推，map 输出数据的 key 为数据，而在这个实例中每个数据代表输入文件中的一行内容，所以 map 阶段要完成的任务就是在采用 Hadoop 默认的作业输入方式之后，将 value 设置为 key，并直接输出（输出中的 value 任意）。map 中的结果经过 shuffle 过程之后交给 reduce。reduce 阶段不会管每个 key 有多少个 value，它直接将输入的 key 复制为输出的 key，并输出就可以了（输出中的 value 被设置成空了）。

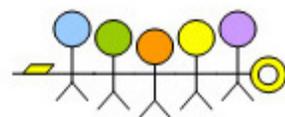
1.3 程序代码

程序代码如下所示：

```
package com.hebut.mr;

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.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.util.GenericOptionsParser;
```



```
public class Dedup {

    //map将输入中的value复制到输出数据的key上，并直接输出
    public static class Map extends Mapper<Object,Text,Text,Text>{
        private static Text line=new Text();//每行数据

        //实现map函数
        public void map(Object key,Text value,Context context)
            throws IOException,InterruptedException{
            line=value;
            context.write(line, new Text(""));
        }
    }

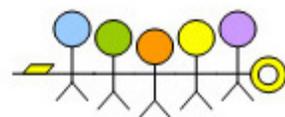
    //reduce将输入中的key复制到输出数据的key上，并直接输出
    public static class Reduce extends Reducer<Text,Text,Text,Text>{
        //实现reduce函数
        public void reduce(Text key,Iterable<Text> values,Context context)
            throws IOException,InterruptedException{
            context.write(key, new Text(""));
        }
    }

    public static void main(String[] args) throws Exception{
        Configuration conf = new Configuration();
        //这句话很关键
        conf.set("mapred.job.tracker", "192.168.1.2:9001");

        String[] ioArgs=new String[]{"dedup_in","dedup_out"};
        String[] otherArgs = new GenericOptionsParser(conf,
ioArgs).getRemainingArgs();
        if (otherArgs.length != 2) {
            System.err.println("Usage: Data Deduplication <in> <out>");
            System.exit(2);
        }

        Job job = new Job(conf, "Data Deduplication");
        job.setJarByClass(Dedup.class);

        //设置Map、Combine和Reduce处理类
        job.setMapperClass(Map.class);
```



```

job.setCombinerClass(Reduce.class);
job.setReducerClass(Reduce.class);

//设置输出类型
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(Text.class);

//设置输入和输出目录
FileInputFormat.addInputPath(job, new Path(otherArgs[0]));
FileOutputFormat.setOutputPath(job, new Path(otherArgs[1]));
System.exit(job.waitForCompletion(true) ? 0 : 1);
}
}

```

1.4 代码结果

1) 准备测试数据

通过 Eclipse 下面的“DFS Locations”在“/user/hadoop”目录下创建输入文件“dedup_in”文件夹（备注：“dedup_out”不需要创建。）如图 1.4-1 所示，已经成功创建。

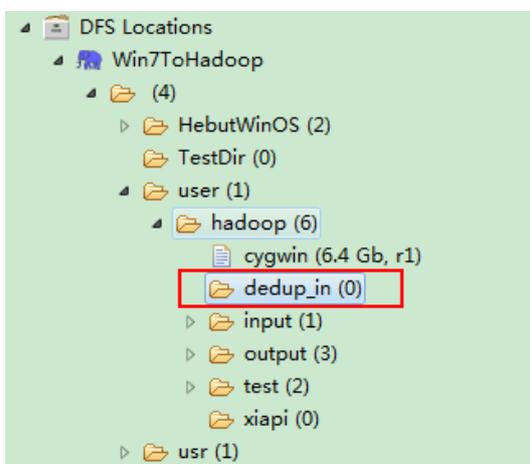


图 1.4-1 创建“dedup_in”

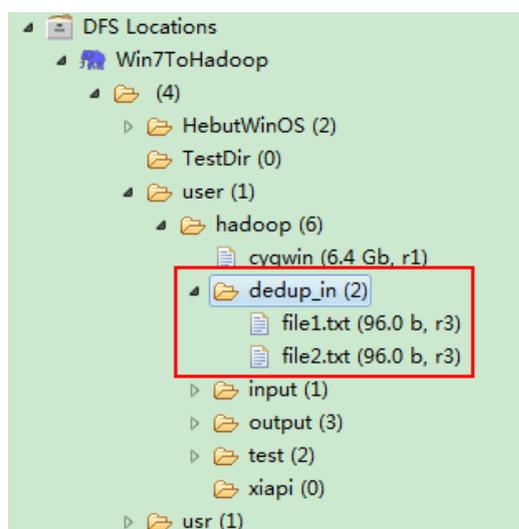


图 1.4.2 上传“file*.txt”

然后在本地建立两个 txt 文件，通过 Eclipse 上传到“/user/hadoop/dedup_in”文件夹中，两个 txt 文件的内容如“实例描述”那两个文件一样。如图 1.4-2 所示，成功上传之后。

从 SecureCRT 远处查看“Master.Hadoop”的也能证实我们上传的两个文件。

```

[hadoop@Master ~]$ hadoop fs -ls dedup_in
Found 2 items
-rw-r--r-- 3 hadoop supergroup          96 2012-03-09 23:45 /user/hadoop/dedup_in/file1.txt
-rw-r--r-- 3 hadoop supergroup          96 2012-03-09 23:45 /user/hadoop/dedup_in/file2.txt
[hadoop@Master ~]$

```

查看两个文件的内容如图 1.4-3 所示：



```
[hadoop@Master ~]$ hadoop fs -cat dedup_in/file1.txt
2012-3-1 a
2012-3-2 b
2012-3-3 c
2012-3-4 d
2012-3-5 a
2012-3-6 b
2012-3-7 c
2012-3-3 c
[hadoop@Master ~]$ hadoop fs -cat dedup_in/file2.txt
2012-3-1 b
2012-3-2 a
2012-3-3 b
2012-3-4 d
2012-3-5 a
2012-3-6 c
2012-3-7 d
2012-3-3 c
[hadoop@Master ~]$
```

图 1.4-3 文件 “file*.txt” 内容

2) 查看运行结果

这时我们**右击** Eclipse 的 “DFS Locations” 中 “/user/hadoop” 文件夹进行刷新，这时会发现多出一个 “dedup_out” 文件夹，且里面有 3 个文件，然后打开双击其 “part-r-00000” 文件，会在 Eclipse 中间把内容显示出来。如图 1.4-4 所示。

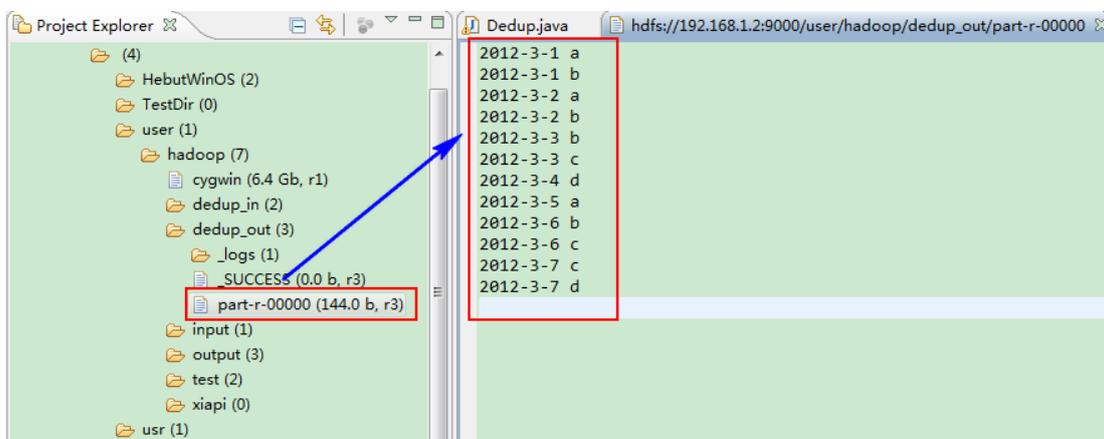


图 1.4-4 运行结果

此时，你可以对比一下和我们之前预期的结果是否一致。

2、数据排序

“**数据排序**” 是许多实际任务执行时要完成的第一项工作，比如**学生成绩评比**、**数据建立索引**等。这个实例和数据去重类似，都是**先对原始数据进行初步处理**，为**进一步**的数据操作**打好基础**。下面进入这个示例。



2.1 实例描述

对输入文件中数据进行排序。**输入文件**中的**每行内容**均为一个**数字**，**即一个数据**。要求在**输出**中每行有**两个间隔**的数字，其中，**第一个**代表原始数据在原始数据集中的**位次**，**第二个**代表**原始数据**。

样例**输入**：

1) **file1:**

```
2
32
654
32
15
756
65223
```

2) **file2:**

```
5956
22
650
92
```

3) **file3:**

```
26
54
6
```

样例**输出**：

```
1 2
2 6
3 15
4 22
5 26
6 32
7 32
8 54
9 92
10 650
11 654
```



```
12 756
13 5956
14 65223
```

2.2 设计思路

这个实例**仅仅**要求对**输入数据进行排序**，熟悉 MapReduce 过程的读者会很快想到在 MapReduce 过程中就有排序，是否可以利用这个**默认**的排序，而不需要自己再实现具体的排序呢？答案是肯定的。

但是在使用之前**首先需要了解**它的**默认排序规则**。它是按照 **key** 值进行**排序**的，如果 key 为封装 int 的 **IntWritable** 类型，那么 MapReduce 按照**数字大小**对 key 排序，如果 key 为封装为 String 的 **Text** 类型，那么 MapReduce 按照**字典顺序**对字符串排序。

了解了这个细节，我们就知道应该使用封装 int 的 IntWritable 型数据结构了。也就是在 map 中将读入的数据转化成 IntWritable 型，然后作为 key 值输出（value 任意）。reduce 拿到 <key, value-list>之后，将输入的 key 作为 value 输出，并根据 **value-list** 中**元素的个数**决定输出的次数。输出的 key（即代码中的 linenum）是一个全局变量，它统计当前 key 的位次。需要注意的是这个程序中**没有配置**Combiner，也就是在 MapReduce 过程中不使用 Combiner。这主要是因为使用 map 和 reduce 就已经能够完成任务了。

2.3 程序代码

程序代码如下所示：

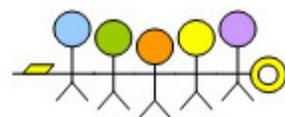
```
package com.hebut.mr;

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.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.util.GenericOptionsParser;

public class Sort {

    //map 将输入中的 value 化成 IntWritable 类型，作为输出的 key
    public static class Map extends
        Mapper<Object,Text,IntWritable,IntWritable>{
```



```
private static IntWritable data=new IntWritable();

//实现 map 函数
public void map(Object key,Text value,Context context)
    throws IOException,InterruptedException{
    String line=value.toString();
    data.set(Integer.parseInt(line));
    context.write(data, new IntWritable(1));
}

}

//reduce 将输入中的 key 复制到输出数据的 key 上,
//然后根据输入的 value-list 中元素的个数决定 key 的输出次数
//用全局linenum来代表key的位次
public static class Reduce extends
    Reducer<IntWritable,IntWritable,IntWritable,IntWritable>{

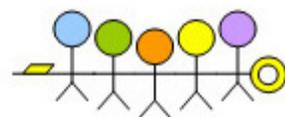
    private static IntWritable linenum = new IntWritable(1);

    //实现 reduce 函数
    public void reduce(IntWritable key,Iterable<IntWritable> values,
        Context context)
        throws IOException,InterruptedException{
        for(IntWritable val:values){
            context.write(linenum, key);
            linenum = new IntWritable(linenum.get()+1);
        }
    }

}

public static void main(String[] args) throws Exception{
    Configuration conf = new Configuration();
    //这句话很关键
    conf.set("mapred.job.tracker", "192.168.1.2:9001");

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



```

}

Job job = new Job(conf, "Data Sort");
job.setJarByClass(Sort.class);

//设置 Map 和 Reduce 处理类
job.setMapperClass(Map.class);
job.setReducerClass(Reduce.class);

//设置输出类型
job.setOutputKeyClass(IntWritable.class);
job.setOutputValueClass(IntWritable.class);

//设置输入和输出目录
FileInputFormat.addInputPath(job, new Path(otherArgs[0]));
FileOutputFormat.setOutputPath(job, new Path(otherArgs[1]));
System.exit(job.waitForCompletion(true) ? 0 : 1);
}
}

```

2.4 代码结果

1) 准备测试数据

通过 Eclipse 下面的“DFS Locations”在“/user/hadoop”目录下创建输入文件“sort_in”文件夹（备注：“sort_out”不需要创建。）如图 2.4-1 所示，已经成功创建。

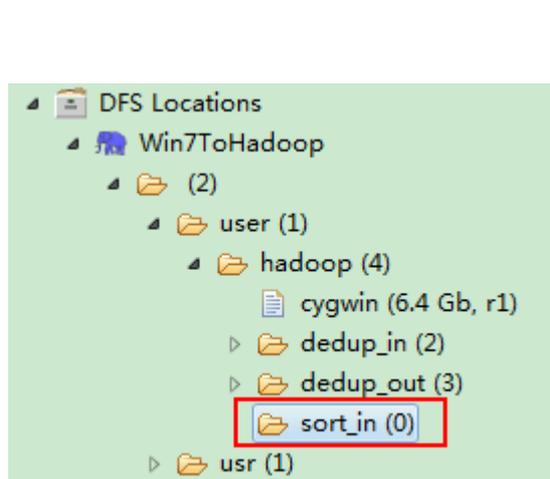


图 2.4-1 创建“sort_in”

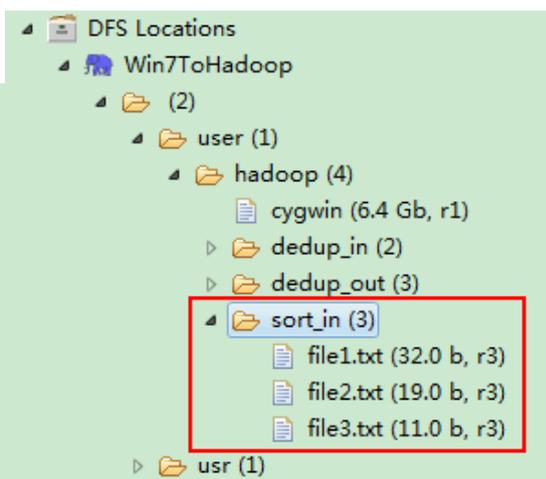


图 2.4.2 上传“file*.txt”

然后在本地建立三个 txt 文件，通过 Eclipse 上传到“/user/hadoop/sort_in”文件夹中，三个 txt 文件的内容如“实例描述”那三个文件一样。如图 2.4-2 所示，成功上传之后。

从 SecureCRT 远处查看“Master.Hadoop”的也能证实我们上传的三个文件。



```
[hadoop@Master ~]$ hadoop fs -ls sort_in
Found 3 items
-rw-r--r--  3 hadoop supergroup      32 2012-03-10 04:53 /user/hadoop/sort_in/file1.txt
-rw-r--r--  3 hadoop supergroup      19 2012-03-10 04:53 /user/hadoop/sort_in/file2.txt
-rw-r--r--  3 hadoop supergroup      11 2012-03-10 04:53 /user/hadoop/sort_in/file3.txt
[hadoop@Master ~]$
```

查看两个文件的内容如图 2.4-3 所示：

```
[hadoop@Master ~]$ hadoop fs -cat sort_in/file1.txt
2
32
654
32
15
756
65223
[hadoop@Master ~]$ hadoop fs -cat sort_in/file2.txt
5956
22
650
92
[hadoop@Master ~]$ hadoop fs -cat sort_in/file3.txt
26
54
6
[hadoop@Master ~]$
```

图 2.4-3 文件 “file*.txt” 内容

2) 查看运行结果

这时我们右击 Eclipse 的 “DFS Locations” 中 “/user/hadoop” 文件夹进行刷新，这时会发现多出一个 “sort_out” 文件夹，且里面有 3 个文件，然后打开双其 “part-r-00000” 文件，会在 Eclipse 中间把内容显示出来。如图 2.4-4 所示。

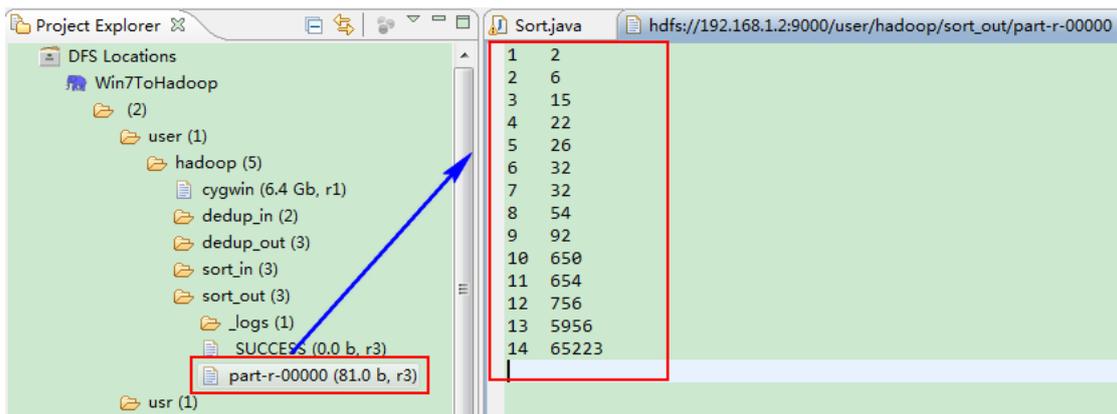


图 2.4-4 运行结果

3、平均成绩

“平均成绩” 主要目的还是在重温经典 “WordCount” 例子，可以说是在基础上的微变化版，该实例主要就是实现一个计算学生平均成绩的例子。



3.1 实例描述

对输入文件中数据进行计算学生平均成绩。输入文件中的每行内容均为一个学生的姓名和他相应的成绩，如果有多门学科，则每门学科为一个文件。要求在输出中每行有两个间隔的数据，其中，第一个代表学生的姓名，第二个代表其平均成绩。

样本输入：

1) math:

张三	88
李四	99
王五	66
赵六	77

2) china:

张三	78
李四	89
王五	96
赵六	67

3) english:

张三	80
李四	82
王五	84
赵六	86

样本输出：

张三	82
李四	90
王五	82
赵六	76

3.2 设计思路

计算学生平均成绩是一个仿“WordCount”例子，用来重温一下开发 MapReduce 程序的流程。程序包括两部分的内容：Map 部分和 Reduce 部分，分别实现了 map 和 reduce 的功能。

Map 处理的是一个纯文本文件，文件中存放的数据时每一行表示一个学生的姓名和他相应一科成绩。Mapper 处理的数据是由 InputFormat 分解过的数据集，其中 InputFormat 的作用是将数据集切割成小数据集 InputSplit，每一个 InputSplit 将由一个 Mapper 负责处理。此外，InputFormat 中还提供了一个 RecordReader 的实现，并将一个 InputSplit 解析成<key,value>



对提供了 map 函数。InputFormat 的默认值是 TextInputFormat，它针对文本文件，按行将文本切割成 InputSplit，并用 LineRecordReader 将 InputSplit 解析成<key,value>对，key 是行在文本中的位置，value 是文件中的一行。

Map 的结果会通过 partition 分发到 Reducer，Reducer 做完 Reduce 操作后，将通过以格式 OutputFormat 输出。

Mapper 最终处理的结果对<key,value>，会送到 Reducer 中进行合并，合并的时候，有相同 key 的键/值对则送到同一个 Reducer 上。Reducer 是所有用户定制 Reducer 类地基础，它的输入是 key 和这个 key 对应的所有 value 的一个迭代器，同时还有 Reducer 的上下文。Reduce 的结果由 Reducer.Context 的 write 方法输出到文件中。

3.3 程序代码

程序代码如下所示：

```
package com.hebut.mr;

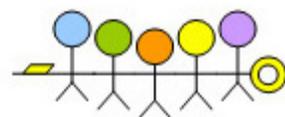
import java.io.IOException;
import java.util.Iterator;
import java.util.StringTokenizer;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;
import org.apache.hadoop.util.GenericOptionsParser;

public class Score {

    public static class Map extends
        Mapper<LongWritable, Text, Text, IntWritable> {

        // 实现 map 函数
        public void map(LongWritable key, Text value, Context context)
            throws IOException, InterruptedException {
            // 将输入的纯文本文件的数据转化成 String
            String line = value.toString();
```



```
// 将输入的数据首先按行进行分割
StringTokenizer tokenizerArticle = new StringTokenizer(line, "\n");

// 分别对每一行进行处理
while (tokenizerArticle.hasMoreElements()) {
    // 每行按空格划分
    StringTokenizer tokenizerLine = new StringTokenizer(
        tokenizerArticle.nextToken());

    String strName = tokenizerLine.nextToken();// 学生姓名部分
    String strScore = tokenizerLine.nextToken();// 成绩部分

    Text name = new Text(strName);
    int scoreInt = Integer.parseInt(strScore);
    // 输出姓名和成绩
    context.write(name, new IntWritable(scoreInt));
}
}

public static class Reduce extends
    Reducer<Text, IntWritable, Text, IntWritable> {
    // 实现 reduce 函数
    public void reduce(Text key, Iterable<IntWritable> values,
        Context context) throws IOException, InterruptedException {

        int sum = 0;
        int count = 0;

        Iterator<IntWritable> iterator = values.iterator();
        while (iterator.hasNext()) {
            sum += iterator.next().get();// 计算总分
            count++;// 统计总的科目数
        }

        int average = (int) sum / count;// 计算平均成绩
        context.write(key, new IntWritable(average));
    }
}

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



```
Configuration conf = new Configuration();
// 这句话很关键
conf.set("mapred.job.tracker", "192.168.1.2:9001");

String[] ioArgs = new String[] { "score_in", "score_out" };
String[] otherArgs = new GenericOptionsParser(conf, ioArgs)
    .getRemainingArgs();
if (otherArgs.length != 2) {
    System.err.println("Usage: Score Average <in> <out>");
    System.exit(2);
}

Job job = new Job(conf, "Score Average");
job.setJarByClass(Score.class);

// 设置 Map、Combine 和 Reduce 处理类
job.setMapperClass(Map.class);
job.setCombinerClass(Reduce.class);
job.setReducerClass(Reduce.class);

// 设置输出类型
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(IntWritable.class);

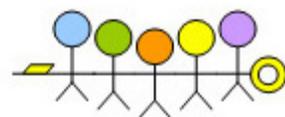
// 将输入的数据集分割成小数据块splites，提供一个RecordReader的实现
job.setInputFormatClass(TextInputFormat.class);
// 提供一个 RecordWriter 的实现，负责数据输出
job.setOutputFormatClass(TextOutputFormat.class);

// 设置输入和输出目录
FileInputFormat.addInputPath(job, new Path(otherArgs[0]));
FileOutputFormat.setOutputPath(job, new Path(otherArgs[1]));
System.exit(job.waitForCompletion(true) ? 0 : 1);
}
}
```

3.4 代码结果

1) 准备测试数据

通过 Eclipse 下面的“DFS Locations”在“/user/hadoop”目录下创建输入文件“score_in”文件夹（备注：“score_out”不需要创建。）如图 3.4-1 所示，已经成功创建。



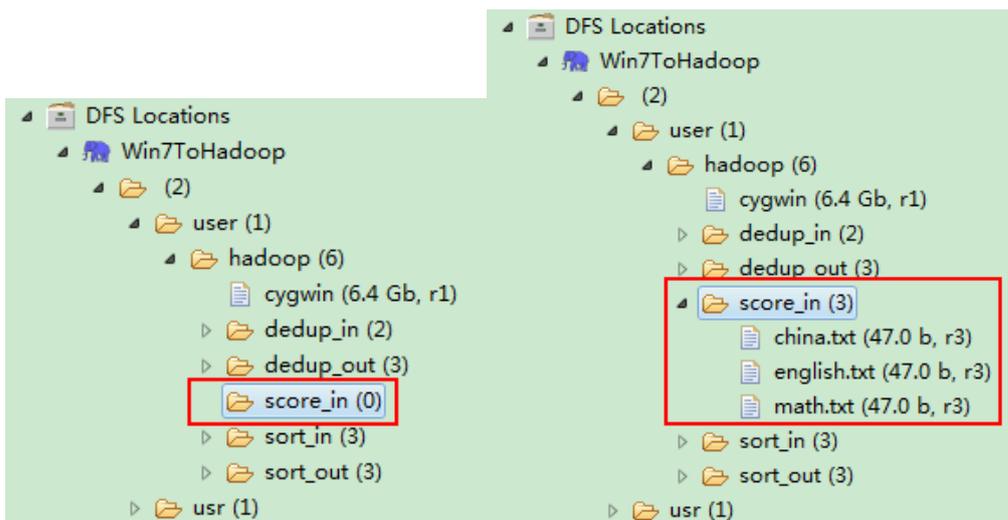


图 3.4-1 创建“score_in”

图 3.4.2 上传三门分数

然后在本地建立三个 txt 文件，通过 Eclipse 上传到“/user/hadoop/score_in”文件夹中，三个 txt 文件的内容如“实例描述”那三个文件一样。如图 3.4-2 所示，成功上传之后。

备注：文本文件的编码为“**UTF-8**”，默认为“**ANSI**”，可以另存为时选择，不然中文会出现乱码。

从 SecureCRT 远处查看“Master.Hadoop”的也能证实我们上传的三个文件。

```
[hadoop@Master ~]$ hadoop fs -ls score_in
Found 3 items
-rw-r--r--  3 hadoop supergroup          47 2012-03-10 18:25 /user/hadoop/score_in/china.txt
-rw-r--r--  3 hadoop supergroup          47 2012-03-10 18:22 /user/hadoop/score_in/english.txt
-rw-r--r--  3 hadoop supergroup          47 2012-03-10 18:25 /user/hadoop/score_in/math.txt
[hadoop@Master ~]$
```

查看三个文件的内容如图 3.4-3 所示：

```
[hadoop@Master ~]$ hadoop fs -cat score_in/math.txt
张三    88
李四    99
王五    66
赵六    77
[hadoop@Master ~]$ hadoop fs -cat score_in/china.txt
张三    78
李四    89
王五    96
赵六    67
[hadoop@Master ~]$ hadoop fs -cat score_in/english.txt
张三    80
李四    82
王五    84
赵六    86
[hadoop@Master ~]$
```

图 3.4.3 三门成绩的内容



2) 查看运行结果

这时我们**右击** Eclipse 的“DFS Locations”中“/user/hadoop”文件夹进行刷新，这时会发现多出一个“score_out”文件夹，且里面有 3 个文件，然后打开双击其“part-r-00000”文件，会在 Eclipse 中间把内容显示出来。如图 3.4-4 所示。

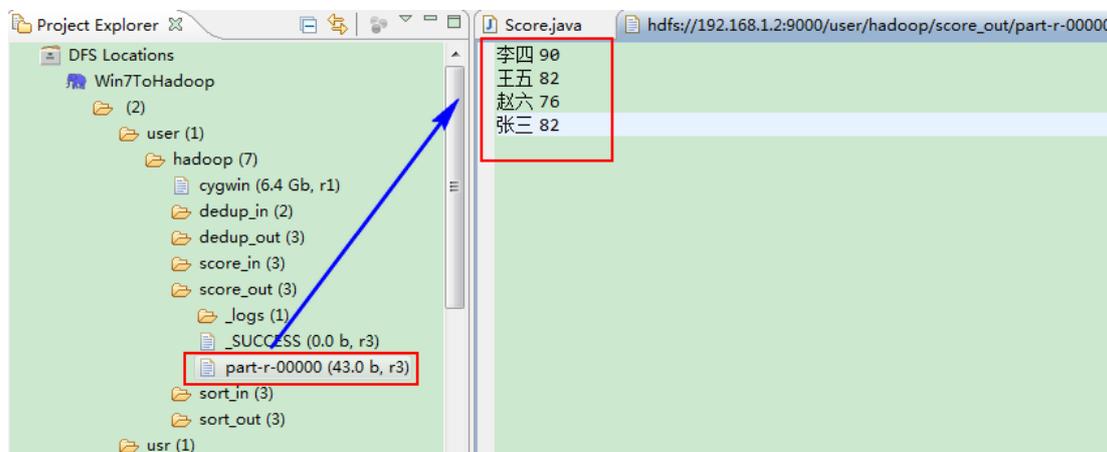


图 3.4-4 运行结果

4、单表关联

前面的实例都是在数据上进行一些简单的处理，为进一步的操作打基础。“**单表关联**”这个实例**要求从给出的数据中寻找所关心的数据**，它是对**原始数据**所包含信息的**挖掘**。下面进入这个实例。

4.1 实例描述

实例中给出 **child-parent**（孩子——父母）表，要求输出 **grandchild-grandparent**（孙子——爷奶）表。

样例**输入**如下所示。

file:

child	parent
Tom	Lucy
Tom	Jack
Jone	Lucy
Jone	Jack
Lucy	Mary
Lucy	Ben
Jack	Alice
Jack	Jesse
Terry	Alice
Terry	Jesse
Philip	Terry



Philip	Alma
Mark	Terry
Mark	Alma

家族树状关系谱：

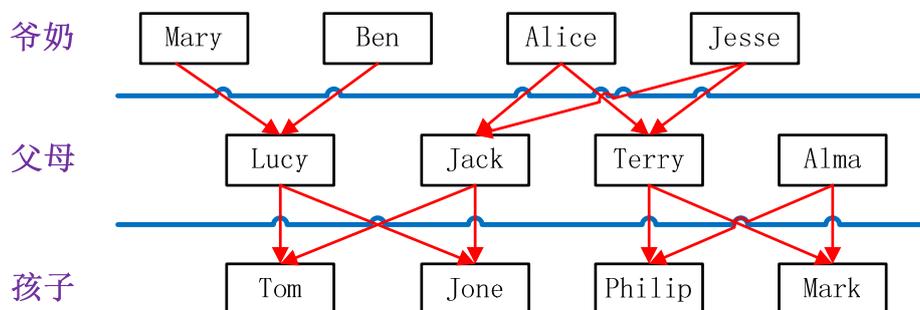


图 4.2-1 家族谱

样例输出如下所示。

file:

grandchild	grandparent
Tom	Alice
Tom	Jesse
Jone	Alice
Jone	Jesse
Tom	Mary
Tom	Ben
Jone	Mary
Jone	Ben
Philip	Alice
Philip	Jesse
Mark	Alma
Mark	Jesse

4.2 设计思路

分析这个实例，显然需要进行单表连接，连接的是左表的 parent 列和右表的 child 列，且左表和右表是同一个表。

连接结果中除去连接的两列就是所需要的结果——“grandchild--grandparent”表。要用 MapReduce 解决这个实例，首先应该考虑如何实现表的自连接；其次就是连接列的设置；最后是结果的整理。

考虑到 MapReduce 的 shuffle 过程会将相同的 key 会连接在一起，所以可以将 map 结果的 key 设置成待连接的列，然后列中相同的值就自然会连接在一起了。再与最开始的分析联系起来：

要连接的是左表的 parent 列和右表的 child 列，且左表和右表是同一个表，所以在 map



阶段将读入数据分割成 **child** 和 **parent** 之后，会将 **parent** 设置成 **key**，**child** 设置成 **value** 进行输出，并作为**左表**；再将**同一对 child** 和 **parent** 中的 **child** 设置成 **key**，**parent** 设置成 **value** 进行输出，作为**右表**。为了**区分**输出中的**左右表**，需要在输出的 **value** 中**再加上左右表的信息**，比如在 value 的 String 最开始处加上**字符 1** 表示**左表**，加上**字符 2** 表示**右表**。这样在 map 的结果中就形成了左表和右表，然后在 shuffle 过程中完成连接。reduce 接收到连接的结果，其中每个 key 的 value-list 就包含了“grandchild--grandparent”关系。取出每个 key 的 value-list 进行解析，将**左表**中的 **child** 放入一个**数组**，**右表**中的 **parent** 放入一个**数组**，然后对**两个数组求笛卡尔积**就是最后的结果了。

4.3 程序代码

程序代码如下所示。

```
package com.hebut.mr;

import java.io.IOException;
import java.util.*;

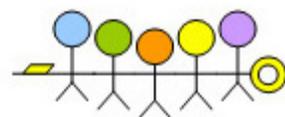
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.util.GenericOptionsParser;

public class STjoin {

    public static int time = 0;

    /*
     * map 将输出分割 child 和 parent，然后正序输出一次作为右表，
     * 反序输出一次作为左表，需要注意的是在输出的 value 中必须
     * 加上左右表的区别标识。
     */
    public static class Map extends Mapper<Object, Text, Text, Text> {

        // 实现 map 函数
        public void map(Object key, Text value, Context context)
            throws IOException, InterruptedException {
```



```
String childname = new String();// 孩子名称
String parentname = new String();// 父母名称
String relationtype = new String();// 左右表标识

// 输入的一行预处理文本
StringTokenizer itr=new StringTokenizer(value.toString());
String[] values=new String[2];
int i=0;
while(itr.hasMoreTokens()){
    values[i]=itr.nextToken();
    i++;
}

if (values[0].compareTo("child") != 0) {
    childname = values[0];
    parentname = values[1];

    // 输出左表
    relationtype = "1";
    context.write(new Text(values[1]), new Text(relationtype +
        "+"+ childname + "+" + parentname));

    // 输出右表
    relationtype = "2";
    context.write(new Text(values[0]), new Text(relationtype +
        "+"+ childname + "+" + parentname));
}
}

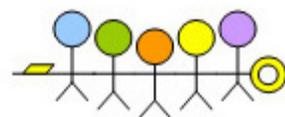
}

public static class Reduce extends Reducer<Text, Text, Text, Text> {

    // 实现 reduce 函数
    public void reduce(Text key, Iterable<Text> values, Context context)
        throws IOException, InterruptedException {

        // 输出表头
        if (0 == time) {
            context.write(new Text("grandchild"), new Text("grandparent"));
            time++;
        }

        int grandchildnum = 0;
```



```
String[] grandchild = new String[10];
int grandparentnum = 0;
String[] grandparent = new String[10];

Iterator ite = values.iterator();
while (ite.hasNext()) {
    String record = ite.next().toString();
    int len = record.length();
    int i = 2;
    if (0 == len) {
        continue;
    }

    // 取得左右表标识
    char relationtype = record.charAt(0);
    // 定义孩子和父母变量
    String childname = new String();
    String parentname = new String();

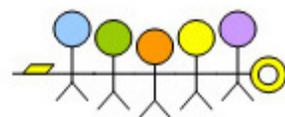
    // 获取 value-list 中 value 的 child
    while (record.charAt(i) != '+') {
        childname += record.charAt(i);
        i++;
    }

    i = i + 1;

    // 获取 value-list 中 value 的 parent
    while (i < len) {
        parentname += record.charAt(i);
        i++;
    }

    // 左表, 取出child放入grandchildren
    if ('1' == relationtype) {
        grandchild[grandchildnum] = childname;
        grandchildnum++;
    }

    // 右表, 取出 parent 放入 grandparent
    if ('2' == relationtype) {
        grandparent[grandparentnum] = parentname;
        grandparentnum++;
    }
}
```



```
    }

    // grandchild和grandparent数组求笛卡尔积
    if (0 != grandchildnum && 0 != grandparentnum) {
        for (int m = 0; m < grandchildnum; m++) {
            for (int n = 0; n < grandparentnum; n++) {
                // 输出结果
                context.write(new Text(grandchild[m]), new Text(
                    grandparent[n]));
            }
        }
    }
}

public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    // 这句话很关键
    conf.set("mapred.job.tracker", "192.168.1.2:9001");

    String[] ioArgs = new String[] { "STjoin_in", "STjoin_out" };
    String[] otherArgs = new GenericOptionsParser(conf, ioArgs)
        .getRemainingArgs();
    if (otherArgs.length != 2) {
        System.err.println("Usage: Single Table Join <in> <out>");
        System.exit(2);
    }

    Job job = new Job(conf, "Single Table Join");
    job.setJarByClass(STjoin.class);

    // 设置 Map 和 Reduce 处理类
    job.setMapperClass(Map.class);
    job.setReducerClass(Reduce.class);

    // 设置输出类型
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(Text.class);

    // 设置输入和输出目录
    FileInputFormat.addInputPath(job, new Path(otherArgs[0]));
    FileOutputFormat.setOutputPath(job, new Path(otherArgs[1]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
}
```



}

4.4 代码结果

1) 准备测试数据

通过 Eclipse 下面的“DFS Locations”在“/user/hadoop”目录下创建输入文件“STjoin_in”文件夹（备注：“STjoin_out”不需要创建。）如图 4.4-1 所示，已经成功创建。

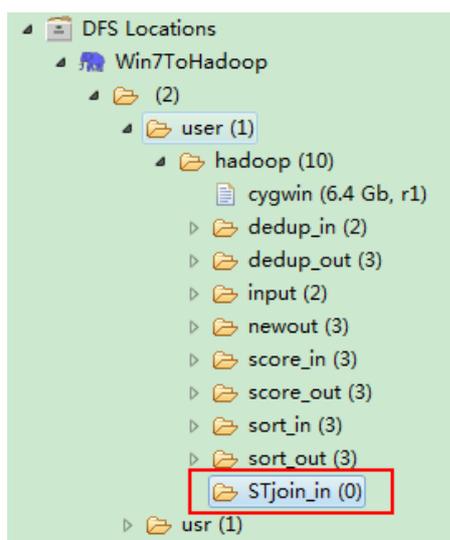


图 4.4-1 创建“STjoin_in”

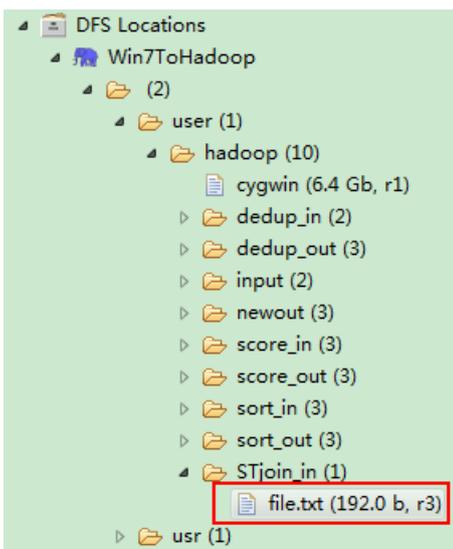


图 4.4.2 上传“child-parent”表

然后在本地建立一个 txt 文件，通过 Eclipse 上传到“/user/hadoop/STjoin_in”文件夹中，一个 txt 文件的内容如“实例描述”那个文件一样。如图 4.4-2 所示，成功上传之后。

从 SecureCRT 远处查看“Master.Hadoop”的也能证实我们上传的文件，显示其内容如图 4.4-3 所示：

```
[hadoop@Master ~]$ hadoop fs -cat STjoin_in/*
child      parent
Tom        Lucy
Tom        Jack
Jone       Lucy
Jone       Jack
Lucy       Mary
Lucy       Ben
Jack       Alice
Jack       Jesse
Terry     Alice
Terry     Jesse
Philip    Terry
Philip    Alma
Mark      Terry
Mark      Alma
[hadoop@Master ~]$
```

图 4.4-3 表“child-parent”内容



2) 运行详解

(1) Map 处理:

map 函数输出结果如下所示。

child	parent	→→	忽略此行
Tom	Lucy	→→	<Lucy, 1+Tom+Lucy> <Tom, 2+Tom+Lucy >
Tom	Jack	→→	<Jack, 1+Tom+Jack> <Tom, 2+Tom+Jack>
Jone	Lucy	→→	<Lucy, 1+Jone+Lucy> <Jone, 2+Jone+Lucy>
Jone	Jack	→→	<Jack, 1+Jone+Jack> <Jone, 2+Jone+Jack>
Lucy	Mary	→→	<Mary, 1+Lucy+Mary> <Lucy, 2+Lucy+Mary>
Lucy	Ben	→→	<Ben, 1+Lucy+Ben> <Lucy, 2+Lucy+Ben>
Jack	Alice	→→	<Alice, 1+Jack+Alice> <Jack, 2+Jack+Alice>
Jack	Jesse	→→	<Jesse, 1+Jack+Jesse> <Jack, 2+Jack+Jesse>
Terry	Alice	→→	<Alice, 1+Terry+Alice> <Terry, 2+Terry+Alice>
Terry	Jesse	→→	<Jesse, 1+Terry+Jesse> <Terry, 2+Terry+Jesse>
Philip	Terry	→→	<Terry, 1+Philip+Terry> <Philip, 2+Philip+Terry>
Philip	Alma	→→	<Alma, 1+Philip+Alma> <Philip, 2+Philip+Alma>
Mark	Terry	→→	<Terry, 1+Mark+Terry> <Mark, 2+Mark+Terry>
Mark	Alma	→→	<Alma, 1+Mark+Alma> <Mark, 2+Mark+Alma>

(2) Shuffle 处理

在 shuffle 过程中完成连接。

map 函数输出	排序结果	shuffle 连接
<Lucy, 1+Tom+Lucy>	<Alice, 1+Jack+Alice>	<Alice, 1+Jack+Alice,
<Tom, 2+Tom+Lucy>	<Alice, 1+Terry+Alice>	1+Terry+Alice ,
<Jack, 1+Tom+Jack>	<Alma, 1+Philip+Alma>	1+Philip+Alma,
<Tom, 2+Tom+Jack>	<Alma, 1+Mark+Alma>	1+Mark+Alma >
<Lucy, 1+Jone+Lucy>	<Ben, 1+Lucy+Ben>	<Ben, 1+Lucy+Ben>
<Jone, 2+Jone+Lucy>	<Jack, 1+Tom+Jack>	<Jack, 1+Tom+Jack,



<Jack, 1+Jone+Jack>	<Jack, 1+Jone+Jack>	1+Jone+Jack,
<Jone, 2+Jone+Jack>	<Jack, 2+Jack+Alice>	2+Jack+Alice,
<Mary, 1+Lucy+Mary>	<Jack, 2+Jack+Jesse>	2+Jack+Jesse >
<Lucy, 2+Lucy+Mary>	<Jesse, 1+Jack+Jesse>	<Jesse, 1+Jack+Jesse,
<Ben, 1+Lucy+Ben>	<Jesse, 1+Terry+Jesse>	1+Terry+Jesse >
<Lucy, 2+Lucy+Ben>	<Jone, 2+Jone+Lucy>	<Jone, 2+Jone+Lucy,
<Alice, 1+Jack+Alice>	<Jone, 2+Jone+Jack>	2+Jone+Jack>
<Jack, 2+Jack+Alice>	<Lucy, 1+Tom+Lucy>	<Lucy, 1+Tom+Lucy,
<Jesse, 1+Jack+Jesse>	<Lucy, 1+Jone+Lucy>	1+Jone+Lucy,
<Jack, 2+Jack+Jesse>	<Lucy, 2+Lucy+Mary>	2+Lucy+Mary,
<Alice, 1+Terry+Alice>	<Lucy, 2+Lucy+Ben>	2+Lucy+Ben>
<Terry, 2+Terry+Alice>	<Mary, 1+Lucy+Mary>	<Mary, 1+Lucy+Mary,
<Jesse, 1+Terry+Jesse>	<Mark, 2+Mark+Terry>	2+Mark+Terry,
<Terry, 2+Terry+Jesse>	<Mark, 2+Mark+Alma>	2+Mark+Alma>
<Terry, 1+Philip+Terry>	<Philip, 2+Philip+Terry>	<Philip, 2+Philip+Terry,
<Philip, 2+Philip+Terry>	<Philip, 2+Philip+Alma>	2+Philip+Alma>
<Alma, 1+Philip+Alma>	<Terry, 2+Terry+Alice>	<Terry, 2+Terry+Alice,
<Philip, 2+Philip+Alma>	<Terry, 2+Terry+Jesse>	2+Terry+Jesse,
<Terry, 1+Mark+Terry>	<Terry, 1+Philip+Terry>	1+Philip+Terry,
<Mark, 2+Mark+Terry>	<Terry, 1+Mark+Terry>	1+Mark+Terry>
<Alma, 1+Mark+Alma>	<Tom, 2+Tom+Lucy>	<Tom, 2+Tom+Lucy,
<Mark, 2+Mark+Alma>	<Tom, 2+Tom+Jack>	2+Tom+Jack>

(3) Reduce 处理

首先由语句“**0 != grandchildnum && 0 != grandparentnum**”得知，只要在“value-list”中没有左表或者右表，则不会做处理，可以根据这条规则去除无效的 shuffle 连接。

无效的 shuffle 连接	有效的 shuffle 连接
<Alice, 1+Jack+Alice, 1+Terry+Alice , 1+Philip+Alma, 1+Mark+Alma >	<Jack, 1+Tom+Jack, 1+Jone+Jack, 2+Jack+Alice, 2+Jack+Jesse >
<Ben, 1+Lucy+Ben>	<Lucy, 1+Tom+Lucy, 1+Jone+Lucy, 2+Lucy+Mary, 2+Lucy+Ben>
<Jesse, 1+Jack+Jesse, 1+Terry+Jesse >	<Terry, 2+Terry+Alice, 2+Terry+Jesse, 1+Philip+Terry, 1+Mark+Terry>
<Jone, 2+Jone+Lucy, 2+Jone+Jack>	
<Mary, 1+Lucy+Mary, 2+Mark+Terry, 2+Mark+Alma>	
<Philip, 2+Philip+Terry, 2+Philip+Alma>	
<Tom, 2+Tom+Lucy, 2+Tom+Jack>	



然后根据下面语句进一步对有效的 shuffle 连接做处理。

```
// 左表，取出 child 放入 grandchildren
if ('1' == relationtype) {
    grandchild[grandchildnum] = childname;
    grandchildnum++;
}

// 右表，取出 parent 放入 grandparent
if ('2' == relationtype) {
    grandparent[grandparentnum] = parentname;
    grandparentnum++;
}
```

针对一条数据进行分析：

```
<Jack, 1+Tom+Jack,
    1+Jone+Jack,
    2+Jack+Alice,
    2+Jack+Jesse >
```

分析结果：左表用“字符 1”表示，右表用“字符 2”表示，上面的<key, value-list>中的“key”表示左表与右表的连接键。而“value-list”表示以“key”连接的左表与右表的相关数据。

根据上面针对左表与右表不同的处理规则，取得两个数组的数据如下所示：

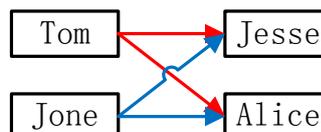
grandchild	Tom、Jone (grandchild[grandchildnum] = childname;)
grandparent	Alice、Jesse (grandparent[grandparentnum] = parentname;)

然后根据下面语句进行处理。

```
for (int m = 0; m < grandchildnum; m++) {
    for (int n = 0; n < grandparentnum; n++) {
        context.write(new Text(grandchild[m]), new Text(grandparent[n]));
    }
}
```

处理结果如下面所示：

Tom	Jesse
Tom	Alice
Jone	Jesse
Jone	Alice



其他的**有效 shuffle 连接**处理都是如此。

3) 查看运行结果

这时我们**右击** Eclipse 的“DFS Locations”中“/user/hadoop”文件夹进行刷新，这时会发现多出一个“STjoin_out”文件夹，且里面有 3 个文件，然后打开双其“part-r-00000”文件，会在 Eclipse 中间把内容显示出来。如图 4.4-4 所示。

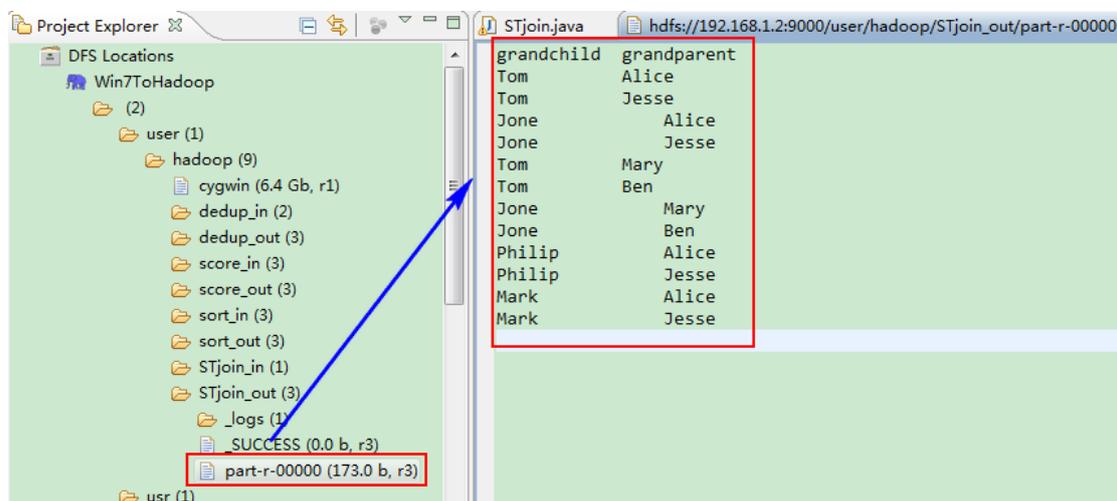


图 4.4-4 运行结果

5、多表关联

多表关联和**单表**关联类似，它也是通过对原始数据进行一定的处理，从其中挖掘出关心的信息。下面进入这个实例。

5.1 实例描述

输入是两个文件，一个代表**工厂表**，包含**工厂名列**和**地址编号列**；另一个代表**地址表**，包含**地址名列**和**地址编号列**。要求从**输入数据**中找出**工厂名**和**地址名**的**对应关系**，输出“**工厂名——地址名**”表。

样例**输入**如下所示。

1) factory:

factoryname	addressed
Beijing Red Star	1
Shenzhen Thunder	3
Guangzhou Honda	2
Beijing Rising	1
Guangzhou Development Bank	2
Tencent	3
Back of Beijing	1

2) address:



addressID	addressname
1	Beijing
2	Guangzhou
3	Shenzhen
4	Xian

样例输出如下所示。

factoryname	addressname
Back of Beijing	Beijing
Beijing Red Star	Beijing
Beijing Rising	Beijing
Guangzhou Development Bank	Guangzhou
Guangzhou Honda	Guangzhou
Shenzhen Thunder	Shenzhen
Tencent	Shenzhen

5.2 设计思路

多表关联和单表关联相似，都类似于数据库中的自然连接。相比单表关联，多表关联的左右表和连接列更加清楚。所以可以采用和单表关联的相同的处理方式，map 识别出输入的行属于哪个表之后，对其进行分割，将连接的列值保存在 key 中，另一列和左右表标识保存在 value 中，然后输出。reduce 拿到连接结果之后，解析 value 内容，根据标志将左右表内容分开存放，然后求笛卡尔积，最后直接输出。

这个实例的具体分析参考单表关联实例。下面给出代码。

5.3 程序代码

程序代码如下所示：

```
package com.hebut.mr;

import java.io.IOException;
import java.util.*;

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.Mapper;
```



```
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.util.GenericOptionsParser;

public class MTjoin {

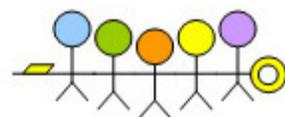
    public static int time = 0;

    /*
     * 在 map 中先区分输入行属于左表还是右表，然后对两列值进行分割，
     * 保存连接列在 key 值，剩余列和左右表标志在 value 中，最后输出
     */
    public static class Map extends Mapper<Object, Text, Text, Text> {

        // 实现 map 函数
        public void map(Object key, Text value, Context context)
            throws IOException, InterruptedException {
            String line = value.toString();// 每行文件
            String relationtype = new String();// 左右表标识

            // 输入文件首行，不处理
            if (line.contains("factoryname") == true
                || line.contains("addressed") == true) {
                return;
            }

            // 输入的一行预处理文本
            StringTokenizer itr = new StringTokenizer(line);
            String mapkey = new String();
            String mapvalue = new String();
            int i = 0;
            while (itr.hasMoreTokens()) {
                // 先读取一个单词
                String token = itr.nextToken();
                // 判断该地址 ID 就把存到“values[0]”
                if (token.charAt(0) >= '0' && token.charAt(0) <= '9') {
                    mapkey = token;
                    if (i > 0) {
                        relationtype = "1";
                    } else {
                        relationtype = "2";
                    }
                }
                continue;
            }
        }
    }
}
```



```
    }

    // 存工厂名
    mapvalue += token + " ";
    i++;
}

// 输出左右表
context.write(new Text(mapkey), new Text(relationtype + "+"
    + mapvalue));
}
}

/*
 * reduce 解析 map 输出，将 value 中数据按照左右表分别保存，
 * 然后求出笛卡尔积，并输出。
 */
public static class Reduce extends Reducer<Text, Text, Text, Text> {

    // 实现 reduce 函数
    public void reduce(Text key, Iterable<Text> values, Context context)
        throws IOException, InterruptedException {

        // 输出表头
        if (0 == time) {
            context.write(new Text("factoryname"), new Text("addressname"));
            time++;
        }

        int factorynum = 0;
        String[] factory = new String[10];
        int addressnum = 0;
        String[] address = new String[10];

        Iterator ite = values.iterator();
        while (ite.hasNext()) {
            String record = ite.next().toString();
            int len = record.length();
            int i = 2;
            if (0 == len) {
                continue;
            }

            // 取得左右表标识
```



```
        char relationtype = record.charAt(0);

        // 左表
        if ('1' == relationtype) {
            factory[factorynum] = record.substring(i);
            factorynum++;
        }

        // 右表
        if ('2' == relationtype) {
            address[addressnum] = record.substring(i);
            addressnum++;
        }
    }

    // 求笛卡尔积
    if (0 != factorynum && 0 != addressnum) {
        for (int m = 0; m < factorynum; m++) {
            for (int n = 0; n < addressnum; n++) {
                // 输出结果
                context.write(new Text(factory[m]),
                    new Text(address[n]));
            }
        }
    }
}

public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    // 这句话很关键
    conf.set("mapred.job.tracker", "192.168.1.2:9001");

    String[] ioArgs = new String[] { "MTjoin_in", "MTjoin_out" };
    String[] otherArgs = new GenericOptionsParser(conf, ioArgs)
        .getRemainingArgs();
    if (otherArgs.length != 2) {
        System.err.println("Usage: Multiple Table Join <in> <out>");
        System.exit(2);
    }

    Job job = new Job(conf, "Multiple Table Join");
```



```

job.setJarByClass(MTjoin.class);

// 设置 Map 和 Reduce 处理类
job.setMapperClass(Map.class);
job.setReducerClass(Reduce.class);

// 设置输出类型
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(Text.class);

// 设置输入和输出目录
FileInputFormat.addInputPath(job, new Path(otherArgs[0]));
FileOutputFormat.setOutputPath(job, new Path(otherArgs[1]));
System.exit(job.waitForCompletion(true) ? 0 : 1);
}
}

```

5.4 代码结果

1) 准备测试数据

通过 Eclipse 下面的“DFS Locations”在“/user/hadoop”目录下创建输入文件“MTjoin_in”文件夹（备注：“MTjoin_out”不需要创建。）如图 5.4-1 所示，已经成功创建。

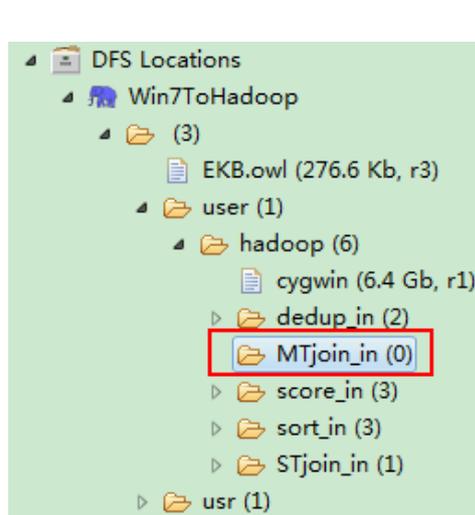


图 5.4-1 创建“MTjoin_in”

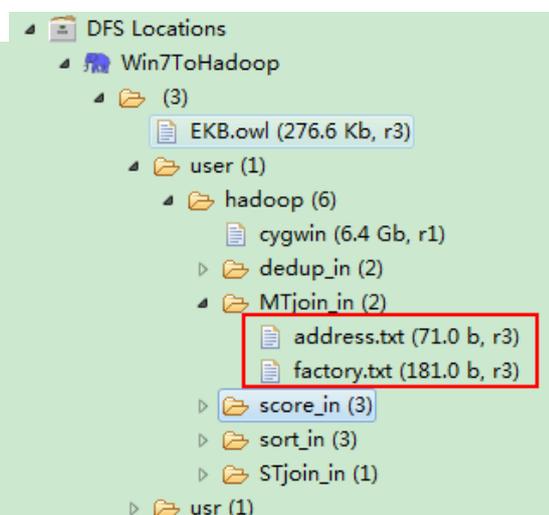


图 5.4.2 上传两个数据表

然后在本地建立两个 txt 文件，通过 Eclipse 上传到“/user/hadoop/MTjoin_in”文件夹中，两个 txt 文件的内容如“实例描述”那两个文件一样。如图 5.4-2 所示，成功上传之后。

从 SecureCRT 远处查看“Master.Hadoop”的也能证实我们上传的两个文件。



```
[hadoop@Master ~]$ hadoop fs -cat MTjoin_in/factory.txt
factoryname                addressed
Beijing Red Star           1
Shenzhen Thunder           3
Guangzhou Honda            2
Beijing Rising             1
Guangzhou Development Bank 2
Tencent                    3
Back of Beijing           1
[hadoop@Master ~]$ hadoop fs -cat MTjoin_in/address.txt
addressID  addressname
1          Beijing
2          Guangzhou
3          Shenzhen
4          Xian
[hadoop@Master ~]$
```

图 5.4.3 两个数据表的内容

2) 查看运行结果

这时我们右击 Eclipse 的“DFS Locations”中“/user/hadoop”文件夹进行刷新，这时会发现多出一个“MTjoin_out”文件夹，且里面有 3 个文件，然后打开双其“part-r-00000”文件，会在 Eclipse 中间把内容显示出来。如图 5.4-4 所示。

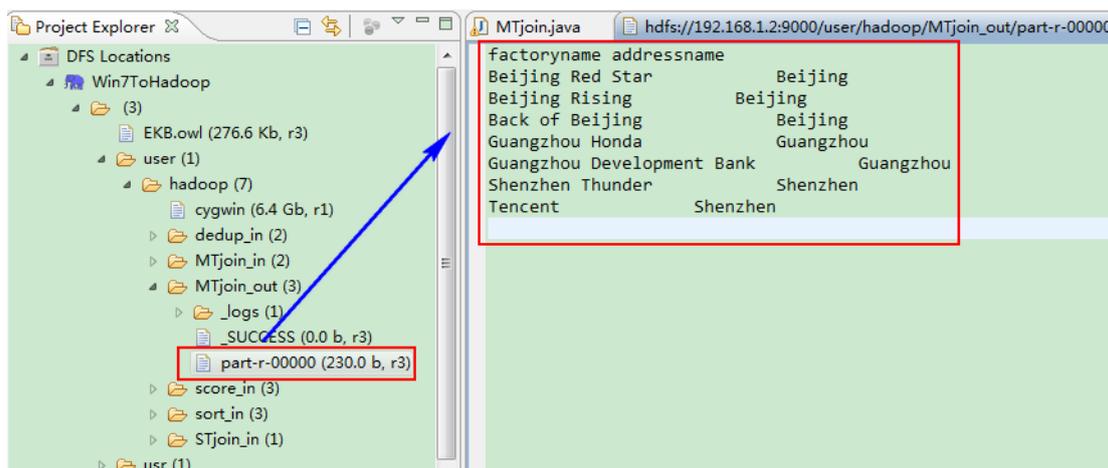


图 5.4-4 运行结果

6、倒排索引

“倒排索引”是文档检索系统中最常用的数据结构，被广泛地应用于全文搜索引擎。它主要是用来存储某个单词（或词组）在一个文档或一组文档中的存储位置的映射，即提供了一种根据内容来查找文档的方式。由于不是根据文档来确定文档所包含的内容，而是进行相反的操作，因而称为倒排索引（Inverted Index）。

6.1 实例描述

通常情况下，倒排索引由一个单词（或词组）以及相关的文档列表组成，文档列表中的文档或者是标识文档的 ID 号，或者是指文档所在位置的 URL，如图 6.1-1 所示。



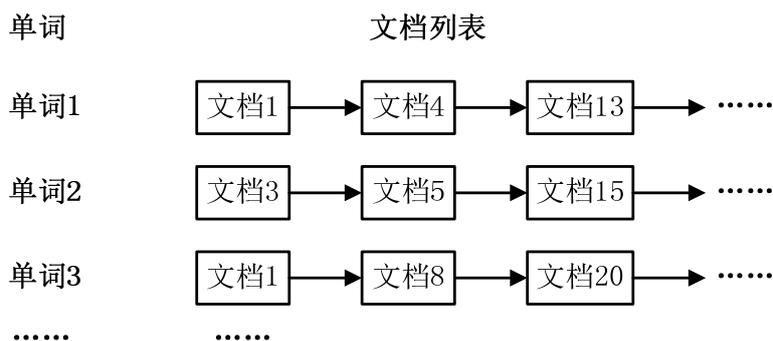


图 6.1-1 倒排索引结构

从图 6.1-1 可以看出，单词 1 出现在{文档 1，文档 4，文档 13，……}中，单词 2 出现在{文档 3，文档 5，文档 15，……}中，而单词 3 出现在{文档 1，文档 8，文档 20，……}中。在**实际应用中**，**还需要给每个文档添加一个权值**，用来**指出**每个文档与搜索内容的**相关度**，如图 6.1-2 所示。

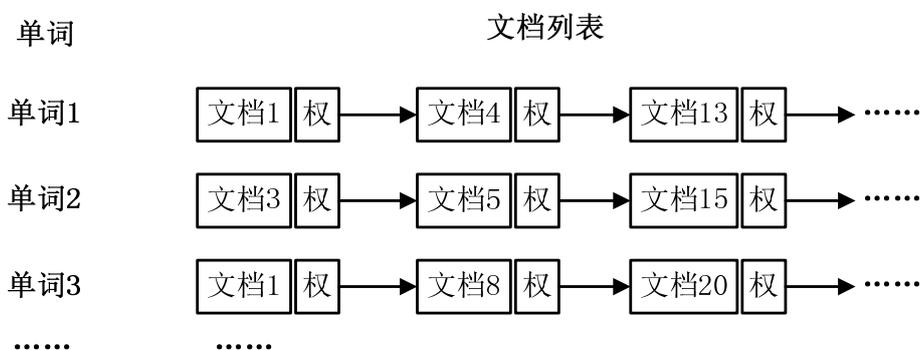


图 6.1-2 添加权重的倒排索引

最常用的是使用**词频**作为**权重**，即记录单词在文档中出现的次数。以英文为例，如图 6.1-3 所示，索引文件中的“MapReduce”一行表示：“MapReduce”这个单词在文本 T0 中出现过 1 次，T1 中出现过 1 次，T2 中出现过 2 次。当搜索条件为“MapReduce”、“is”、“Simple”时，对应的集合为： $\{T0, T1, T2\} \cap \{T0, T1\} \cap \{T0, T1\} = \{T0, T1\}$ ，即文档 T0 和 T1 包含了所要索引的单词，而且只有 T0 是连续的。

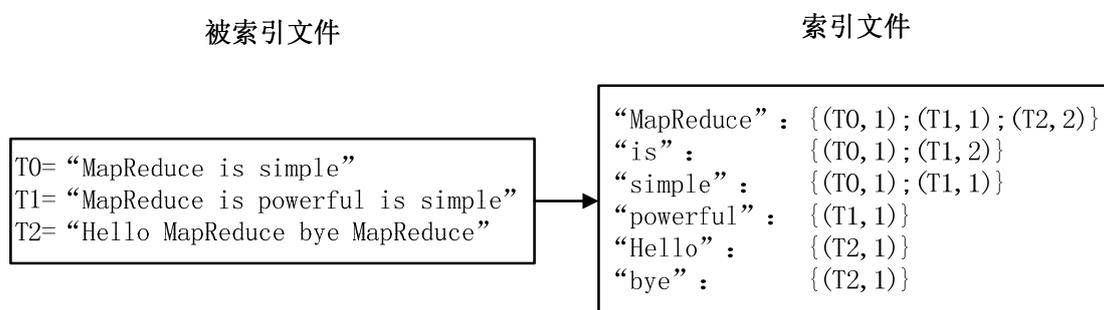


图 6.1-3 倒排索引示例

更复杂的权重还可能要记录单词在多少个文档中出现过，以实现 TF-IDF (Term Frequency-Inverse Document Frequency) 算法，或者考虑单词在文档中的位置信息（单词是否出现在标题中，反映了单词在文档中的重要性）等。

样例**输入**如下所示。



1) file1:

MapReduce is simple

2) file2:

MapReduce is powerful is simple

3) file3:

Hello MapReduce bye MapReduce

样例输出如下所示。

```
MapReduce file1.txt:1;file2.txt:1;file3.txt:2;
is file1.txt:1;file2.txt:2;
simple file1.txt:1;file2.txt:1;
powerful file2.txt:1;
Hello file3.txt:1;
bye file3.txt:1;
```

6.2 设计思路

实现“倒排索引”只要关注的信息为：单词、文档 URL 及词频，如图 3-11 所示。但是在实现过程中，索引文件的格式与图 6.1-3 会略有所不同，以避免重写 OutPutFormat 类。下面根据 MapReduce 的处理过程给出倒排索引的设计思路。

1) Map 过程

首先使用默认的 TextInputFormat 类对输入文件进行处理，得到文本中每行的偏移量及其内容。显然，Map 过程首先必须分析输入的<key,value>对，得到倒排索引中需要的三个信息：单词、文档 URL 和词频，如图 6.2-1 所示。

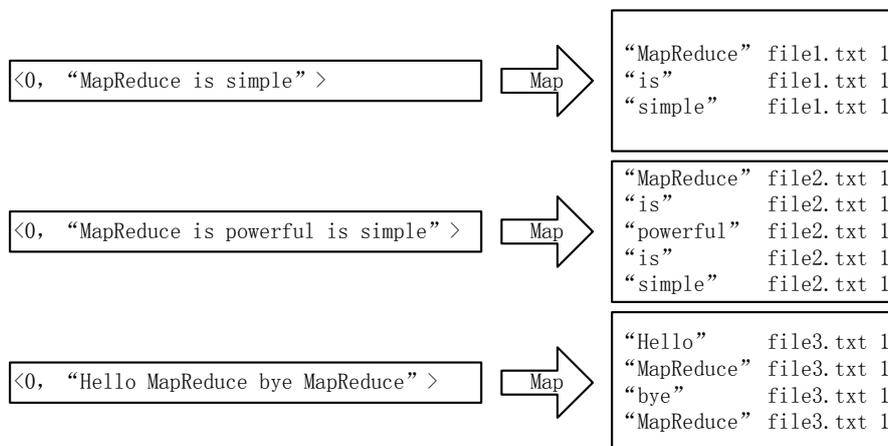


图 6.2-1 Map 过程输入/输出



这里存在两个问题：第一，<key,value>对只能有两个值，在不使用 Hadoop 自定义数据类型的前提下，需要根据情况将其中两个值合并成一个值，作为 key 或 value 值；第二，通过一个 Reduce 过程无法同时完成词频统计和生成文档列表，所以必须增加一个 Combine 过程完成词频统计。

这里讲单词和 URL 组成 key 值（如“MapReduce: file1.txt”），将词频作为 value，这样做的好处是可以利用 MapReduce 框架自带的 Map 端排序，将同一文档的相同单词的词频组成列表，传递给 Combine 过程，实现类似于 WordCount 的功能。

2) Combine 过程

经过 map 方法处理后，Combine 过程将 key 值相同的 value 值累加，得到一个单词在文档中的词频，如图 6.2-2 所示。如果直接将图 6.2-2 所示的输出作为 Reduce 过程的输入，在 Shuffle 过程时将面临一个问题：所有具有相同单词的记录（由单词、URL 和词频组成）应该交由同一个 Reducer 处理，但当前的 key 值无法保证这一点，所以必须修改 key 值和 value 值。这次将单词作为 key 值，URL 和词频组成 value 值（如“file1.txt: 1”）。这样做的好处是可以利用 MapReduce 框架默认的 HashPartitioner 类完成 Shuffle 过程，将相同单词的所有记录发送给同一个 Reducer 进行处理。

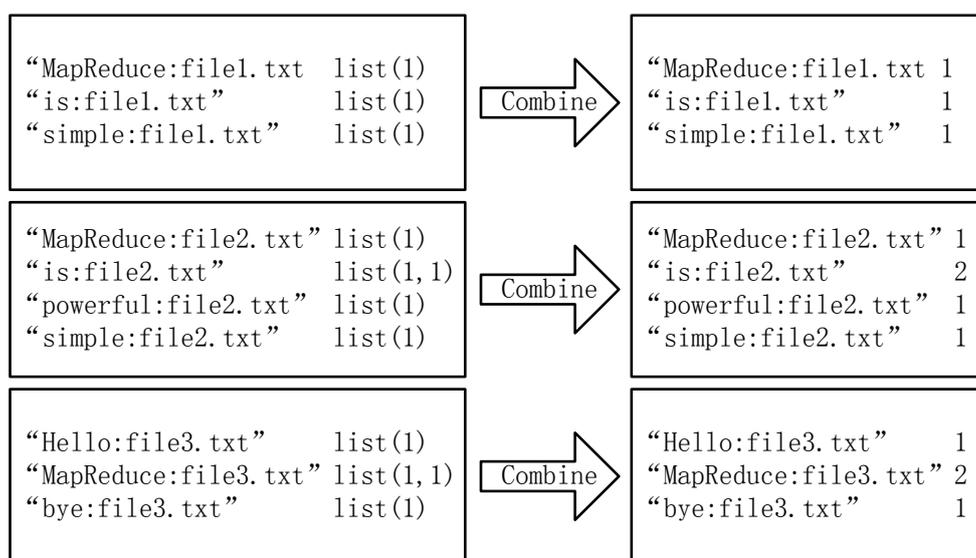


图 6.2-2 Combine 过程输入/输出

3) Reduce 过程

经过上述两个过程后，Reduce 过程只需将相同 key 值的 value 值组合成倒排索引文件所需的格式即可，剩下的事情就可以直接交给 MapReduce 框架进行了。如图 6.2-3 所示。索引文件的内容除分隔符外与图 6.1-3 解释相同。

4) 需要解决的问题

本实例设计的倒排索引在文件数目上没有限制，但是单词文件不宜过大（具体值与默认 HDFS 块大小及相关配置有关），要保证每个文件对应一个 split。否则，由于 Reduce 过程没有进一步统计词频，最终结果可能会出现词频未统计完全的单词。可以通过重写 InputFormat 类将每个文件为一个 split，避免上述情况。或者执行两次 MapReduce，第一次 MapReduce 用于统计词频，第二次 MapReduce 用于生成倒排索引。除此之外，还可以利用复合键值对等实现包含更多信息的倒排索引。



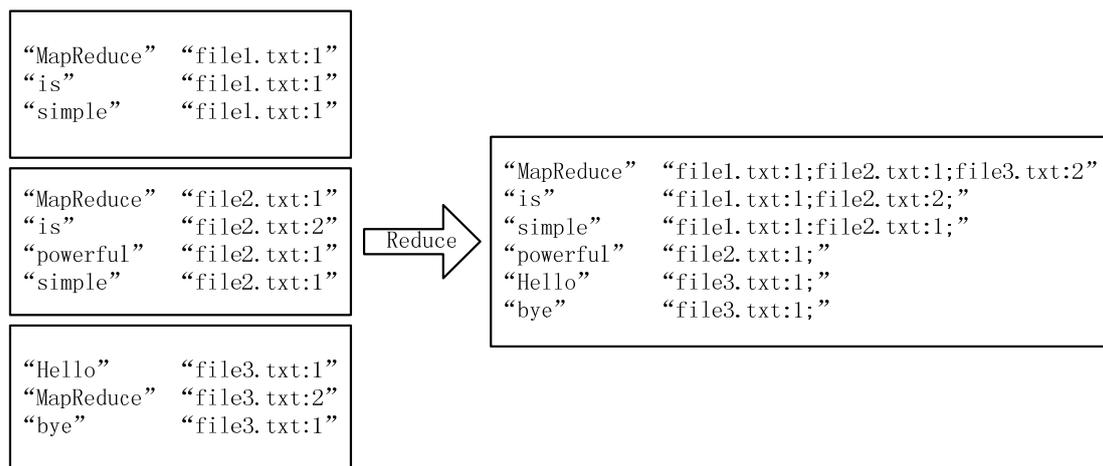


图 6.2-3 Reduce 过程输入/输出

6.3 程序代码

程序代码如下所示：

```
package com.hebut.mr;

import java.io.IOException;
import java.util.StringTokenizer;

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.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.input.FileSplit;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.util.GenericOptionsParser;

public class InvertedIndex {

    public static class Map extends Mapper<Object, Text, Text, Text> {

        private Text keyInfo = new Text(); // 存储单词和 URL 组合
        private Text valueInfo = new Text(); // 存储词频
        private FileSplit split; // 存储 Split 对象
```



```
// 实现 map 函数
public void map(Object key, Text value, Context context)
    throws IOException, InterruptedException {

    // 获得<key,value>对所属的 FileSplit 对象
    split = (FileSplit) context.getInputSplit();

    StringTokenizer itr = new StringTokenizer(value.toString());

    while (itr.hasMoreTokens()) {
        // key值由单词和URL组成，如“MapReduce: file1.txt”

        // 获取文件的完整路径
        // keyInfo.set(itr.nextToken()+":"+split.getPath().toString());

        // 这里为了好看，只获取文件的名称。
        int splitIndex = split.getPath().toString().indexOf("file");
        keyInfo.set(itr.nextToken() + ":"
            + split.getPath().toString().substring(splitIndex));
        // 词频初始化为 1
        valueInfo.set("1");

        context.write(keyInfo, valueInfo);
    }
}

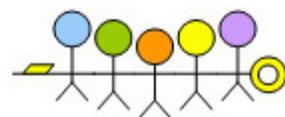
public static class Combine extends Reducer<Text, Text, Text, Text> {

    private Text info = new Text();

    // 实现 reduce 函数
    public void reduce(Text key, Iterable<Text> values, Context context)
        throws IOException, InterruptedException {

        // 统计词频
        int sum = 0;
        for (Text value : values) {
            sum += Integer.parseInt(value.toString());
        }

        int splitIndex = key.toString().indexOf(":");
        // 重新设置 value 值由 URL 和词频组成
```



```
        info.set(key.toString().substring(splitIndex + 1) + ":" + sum);
        // 重新设置 key 值为单词
        key.set(key.toString().substring(0, splitIndex));

        context.write(key, info);
    }
}

public static class Reduce extends Reducer<Text, Text, Text, Text> {

    private Text result = new Text();

    // 实现 reduce 函数
    public void reduce(Text key, Iterable<Text> values, Context context)
        throws IOException, InterruptedException {

        // 生成文档列表
        String fileList = new String();
        for (Text value : values) {
            fileList += value.toString() + ";";
        }

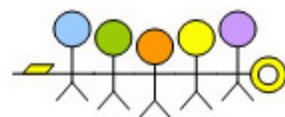
        result.set(fileList);

        context.write(key, result);
    }
}

public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    // 这句话很关键
    conf.set("mapred.job.tracker", "192.168.1.2:9001");

    String[] ioArgs = new String[] { "index_in", "index_out" };
    String[] otherArgs = new GenericOptionsParser(conf, ioArgs)
        .getRemainingArgs();
    if (otherArgs.length != 2) {
        System.err.println("Usage: Inverted Index <in> <out>");
        System.exit(2);
    }

    Job job = new Job(conf, "Inverted Index");
```



```
job.setJarByClass(InvertedIndex.class);

// 设置 Map、Combine 和 Reduce 处理类
job.setMapperClass(Map.class);
job.setCombinerClass(Combine.class);
job.setReducerClass(Reduce.class);

// 设置 Map 输出类型
job.setMapOutputKeyClass(Text.class);
job.setMapOutputValueClass(Text.class);

// 设置 Reduce 输出类型
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(Text.class);

// 设置输入和输出目录
FileInputFormat.addInputPath(job, new Path(otherArgs[0]));
FileOutputFormat.setOutputPath(job, new Path(otherArgs[1]));
System.exit(job.waitForCompletion(true) ? 0 : 1);
}
}
```

6.4 代码结果

1) 准备测试数据

通过 Eclipse 下面的“DFS Locations”在“/user/hadoop”目录下创建输入文件“index_in”文件夹（备注：“index_out”不需要创建。）如图 6.4-1 所示，已经成功创建。

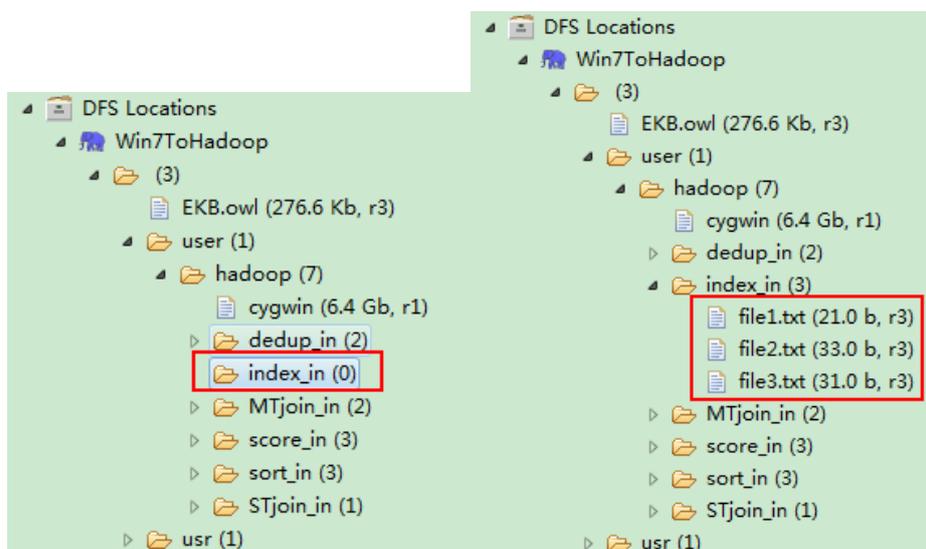


图 6.4-1 创建“index_in”

图 6.4.2 上传“file*.txt”



然后在本地建立三个 txt 文件，通过 Eclipse 上传到 “/user/hadoop/index_in” 文件夹中，三个 txt 文件的内容如 “实例描述” 那三个文件一样。如图 6.4-2 所示，成功上传之后。

从 SecureCRT 远处查看 “Master.Hadoop” 的也能证实我们上传的三个文件。

```
[hadoop@Master ~]$ hadoop fs -ls index_in
Found 3 items
-rw-r--r--  3 hadoop supergroup      21 2012-03-12 00:56 /user/hadoop/index_in/file1.txt
-rw-r--r--  3 hadoop supergroup      33 2012-03-12 00:56 /user/hadoop/index_in/file2.txt
-rw-r--r--  3 hadoop supergroup      31 2012-03-12 00:56 /user/hadoop/index_in/file3.txt
[hadoop@Master ~]$ hadoop fs -cat index_in/file1.txt
MapReduce is simple
[hadoop@Master ~]$ hadoop fs -cat index_in/file2.txt
MapReduce is powerful is simple
[hadoop@Master ~]$ hadoop fs -cat index_in/file3.txt
Hello MapReduce bye MapReduce
[hadoop@Master ~]$
```

图 6.4.3 三个 “file*.txt” 的内容

2) 查看运行结果

这时我们右击 Eclipse 的 “DFS Locations” 中 “/user/hadoop” 文件夹进行刷新，这时会发现多出一个 “index_out” 文件夹，且里面有 3 个文件，然后打开双其 “part-r-00000” 文件，会在 Eclipse 中间把内容显示出来。如图 6.4-4 所示。

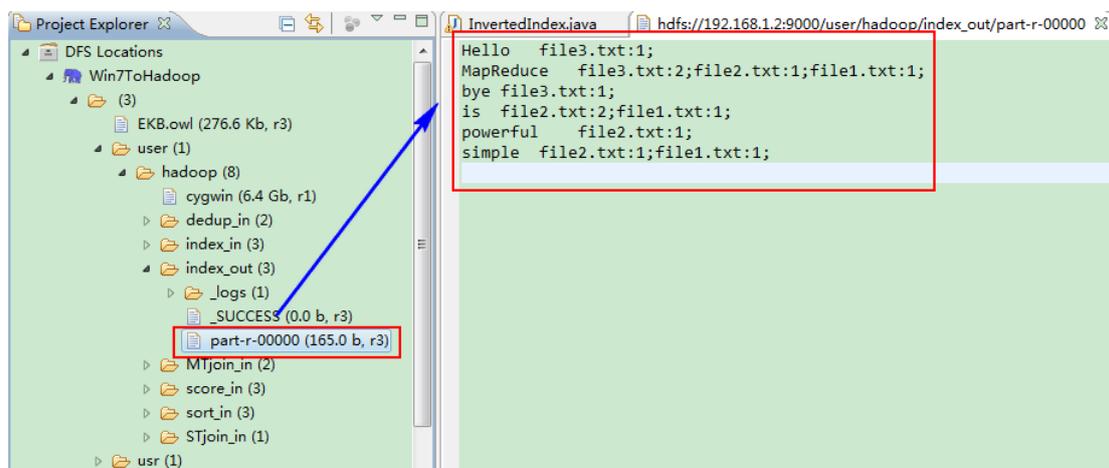


图 6.4-4 运行结果

