金庸的武侠世界

尤明辉，161220161

周敏苑，161220180

周宇航，161220182

周梓康，161220183

联系邮箱：744022996@qq.com

July 24, 2019

Abstract

” 飞雪连天射白鹿，笑书神侠倚碧鸳”

MapReduce 是一个并行程序设计模型与方法，它借助于函数式程序设计语言 Lisp 的设计思想，提供了一种简便的并行程序设计方法，用 Map 和 Reduce 两个函数编程实现基本的并行计算任务，提供了抽象的操作和并行编程接口，以简单方便地完成大规模数据的编程和计算处理。

在本次课程设计中，将通过一个综合数据分析案例：” 金庸的江湖——金庸武侠小说中的人物关系挖掘 “，来学习和掌握 MapReduce 程序设计。通过本课程设计的学习，可以体会如何使用 MapReduce 完成一个综合性的数据挖掘任务，包括全流程的数据预处理、数据分析、数据后处理等。

在使用 Hadoop 完成课程设计之外，同时使用 Spark 的伪分布模式完成了本次实验，体会到了 Scala 语言的简约之美。

Keywords：MapReduce，Spark，中文文本分析，金庸，武侠小说

Contents

1 任务描述

任务号	任务名	任务输入	任务输出	任务描述
1	数据预处理	金庸武侠小说全集	仅保留分段分词后的仅保留人名的金庸武侠小说全集	从原始的金庸小说文本中，抽取出不人物互劢相关的数据，而屏蔽掉不人物关系无关的文本内容，为后面的基于人物共现的分析做准备。
2	人名同现统计	仅保留分段分词后的金庸武侠小说全集	人名同现次数	完成基于单词同现算法的人物同现统计。如果两个人在原文的同一段落中出现，则认为两个人发生了一次同现关系。同现关系次数越多，则说明两人的关系越密切。
3	人物关系图构建与特征归一化	人物之间的共现次数	归一化权重后的人物邻接表	人物关系图中，人名是顶点，人物之间的共现关系是边。
4	基于人物关系邻接表的 PageRank 计算	归一化后权重的人物关系图	人物的 PageRank 值	通过计算 PageRank，我们可以定量的描述出金庸武侠小说中的主角们是哪些。
5	标签传播	归一化后权重的人物关系图	人物的标签信息	标签传播是一种半监督的图分析算法，他能为图上的顶点打标签，进行图顶点的聚类分析，从而在一张类似社交网络图中完成社区发现。
6	分析结果整理（选做）	PageRank 输出标签传播输出	升序人物 PageRank 值相同标签的人物	全局排序程序对人物的 PageRank 值排序，发现 PageRank 值最高的几个人物。编写程序将同一个标签的人物输出，以便查看标签传播结果。

1.1 任务分工

- 尤明辉：任务一、任务二的 Hadoop 以及 Spark 实现，实验报告整体框架编写补充
- 周敏苑：任务三、任务四的 Hadoop 以及 Spark 实现
- 周梓康：任务五的 Hadoop 以及 Spark 实现，对各个任务的 Spark 程序进行调试完善
- 周宇航：任务六的 Hadoop 以及 Spark 实现，绘制标签传播图例，本地测试运行程序
- 实验报告的各个部分实验细节，思考过程由承担相应任务的同学完成

1.2 任务一：数据预处理

本任务的主要工作是从原始的金庸小说文本中，抽取出不人物互劢相关的数据，而屏蔽掉不人物关系无关的文本内容，为后面的基于人物共现的分析做准备。

1.2.1 输入与输出

输入：金庸武侠小说全集、金庸武侠小说人名列表

输出：经过分段分词的且仅仅保留人名的金庸武侠小说全集

1.2.2 示例

输入：金庸 03 连城诀.txt 中的某一段内容

> **狄云**和**戚芳**一走到万家大宅之前，瞧见那高墙朱门、挂灯结彩的气派，心中都是暗自嘀咕。**戚芳**紧紧拉住了父亲的衣袖。**戚长发**正待向门公询问，忽见**卜垣**从门里出来，心中一喜，叫道：“卜贤侄，我来啦。”

输出：仅保留人名信息的分词结果

> **狄云 戚芳 戚芳 戚长发 卜垣**

由于后续步骤中统计人名同现时，在同一段落中重复出现的人名只记作一次，所以可以直接在任务一中执行去重操作，这时输出结果应为：

> **狄云 戚芳 戚长发 卜垣**

由于在一部小说中，可能会有很多段落中出现完全相同的人名，因此为了加速后续计算，我们对这类情况在任务一中进行统计归一。最终输出结果为以下格式，其中数字部分为这一组人名出现的次数：

> **狄云 戚芳 戚长发 卜垣**,1

1.3 任务二：人名同现统计

1.3.1 输入与输出

输入：经过分段分词的且仅仅保留人名的金庸武侠小说全集

输出：人名同现次数

1.3.2 示例

输入：由于在任务一中已经完成了同一段落中相同人名的去重，这时的输入处理起来更加方便。同时可以通过统计完成的数字省略大量计算步骤。

狄云 戚芳 戚长发 卜垣,1
戚芳 卜垣,1

输出:

< 狄云，戚芳 > 1
< 狄云，戚长发 > 1
< 狄云，卜垣 > 1
< 戚芳，狄云 > 1
< 戚芳，戚长发 > 1
< 戚芳，卜垣 > 2
< 戚长发，狄云 > 1
< 戚长发，戚芳 > 1
< 戚长发，卜垣 > 1
< 卜垣，狄云 > 1
< 卜垣，戚芳 > 2
< 卜垣，戚长发 > 1

1.4 任务三：特征处理：人物关系图构建与特征归一化

任务二中得到了两个人物共同出现的次数，但是使用边来就表示图不利于求的 pagerank 值，所以应该将边的表示转换为邻接表表示，且将共同出现次数归一化得到共同出现概率。两人之间的共现次数体现出两人关系的密切程度，反映到共现关系图上就是边的权重。边的权重越高则体现了两个人的关系越密切。

1.4.1 输入输出

输入：key：< 人名，人名 >，value：共现次数
输出：key：人名，value：[人名，共现次数（归一化）；人名，共现次数（归一化）...]

1.5 任务四：数据分析：人物关系图构建与特征归一化

通过任务三得到的归一化权值，计算每个人物的 pagerank 值从而定量的知道金庸武侠世界中的主角们。

1.5.1 输入输出

输入：key：人名，value：[人名，共现次数（归一化）；人名，共现次数（归一化）...]
输出：key: 人名，value: pagerank

1.6 任务五：数据分析：在人物关系图上的标签传播

标签传播（Label Propagation）是一种半监督的图分析算法，能为图上的顶点打标签，进行图顶点的聚类分析，从而在一张类似社交网络图中完成社区发现（Community Detection）。

1.6.1 输入输出

输入：key：人名，value：[人名，共现次数（归一化）；人名，共现次数（归一化）...]
输出：key: 人名，value: 标签

1.7 任务六：分析结果整理（选做）

任务 4 和任务 5 默认的输出内容是杂乱的，从中无法直接的得到分析结论，因此我们需要对上述分析的结果进行整理、从而很容易地从分析结果中发现一些有趣的结论。

由此我们有以下两个任务：

(1) 对于任务 5（PageRank）的输出，写一个全局排序的程序对人物的 PageRank 值进行排序，由此得到 PageRank 值最高的几个人物

(2) 对于任务 6（标签传播）的输出，写一个 MapReduce 程序，将属于同一个标签的人物输出到一起，以便查看标签传播的结果

1.7.1 输入与输出 (1)

输入：经过 PageRank 算法处理得到每个人物姓名及其 PageRank 值
输出：按 PageRank 值大小升序排列的人物姓名及 PageRank 值序列

1.7.2 示例 (1)

输入：

一灯大师 1.3705591
丁不三 1.4129652
丁不四 2.318331
丁典 1.6098274

输出：

一灯大师 1.3705591

丁不三 1.4129652

丁典 1.6098274

丁不四 2.318331

1.7.3 输入与输出 (2)

输入：经过标签传播算法处理得到每个人物姓名及其标签值

输出：按标签值进行聚集的人物姓名集合

1.7.4 示例 (2)

输入：

一灯大师 396

丁不三 819

丁不四 819

丁典 182

输出：

182 **丁典**

396 **一灯大师**

819 **丁不三，丁不四**

2 任务一：数据预处理

由于输入文件是金庸武侠小说全集，其内容包含大量与人物关系无关的信息，不便于直接进行数据分析。因此我们需要对小说进行预处理，排除与人物关系无关的文本内容，仅仅留下简洁直观的人名信息，用于进行后续任务。

由于输入数据集中小说内容并未进行分词，而使用遍历的方式直接筛选出人名信息是不可取的，我们使用 ansj_seg 外部库，进行自定义词库用于分词。

算法 1 构建自定义词库

输入: $name\_list$, $default\_diclibrary$
输出: $user\_diclibrary$

```
function INSERT_NAME(name_list, DicLibrary)
    for line in name_list do
        DicLibrary.insert(DicLibrary.DEFAULT, line)
    end for
    user_DicLibrary ← DicLibrary
    return user_DicLibrary
end function
```

2.1 ansj_seg 自定义词库及分词

ansj_seg 分词的默认词库可以对汉语文本进行简单的，符合一般逻辑的分词，但是如果直接利用默认词库进行分词的话会带来以下后果：

- 人名分词不准确，例如可能会将“东方不败”划分为“东方”、“不败”两个汉语词语
- 无法为分词之后的人名打上正确的标签，难以区分哪些是人名，哪些是其他汉语词语

使用自定义词库可以解决以上所有问题。首先在分词时，ansj 会优先考虑用户自定义词库的分词标准，之后再考虑默认词库；再者在自定义词库时，我们可以为人名打上合适的，专有的标签，例如为所有人名打上“peoplename”标签。这样可以从分词结果中直接获得所需的，仅仅包含人名信息的部分。

算法 2 对文本进行分词并筛选

输入: 武侠小说, $user\_diclibrary$
输出: 分词结果

```
function INSERT_NAME(novel, DicLibrary)
    for line in novel do
        terms ← DicAnalysis.parse(line).getTerms()
        for i in terms do
            if i.label is 'peoplename' then
                words.add(i)
            end if
        end for
        res.add(words)
    end for
    return res
end function
```

由于同一段落中相同人名多次出现只应该算作一次，因此在实际代码中使用了集合来保证同一段落中人名分词结果的互异性。

2.2 MapReduce 实现

2.2.1 自定义词典的构建

由于 MapReduce 程序在集群的各个节点上分布式完成，因此需要保证每个节点的 Map 程序都可以获得相同的完整的用户自定义词典。由于人名列表信息存放在各个节点的 HDFS 系统中，我们只需要在每个节点上读取该节点 HDFS 系统中的 people_name_list.txt 即可。自定义词典的构建在各个 Map 程序的 setUp 步骤完成。

```
//java code
public static class PreSolveMapper
            extends Mapper<Object, Text, Text, IntWritable> {
    public void setup(Context context) {
        try {
            Path pt = new Path("hdfs:/data/task2/people_name_list.txt");
            FileSystem fs = FileSystem.get(new Configuration());
            BufferedReader br = new BufferedReader(new InputStreamReader(fs.open(pt)));
            String line;
            line = br.readLine();
            while (line != null) {
                DicLibrary.insert(DicLibrary.DEFAULT, line);
                line = br.readLine();
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public void map(Object key, Text value, Context context){...}
}
```

解决了在各个节点构建自定义词典的问题后，可以直接利用该词典对中文文本进行简单高效并且准确的分词，详细步骤在后续中说明。

2.2.2 Mapper 与 Reducer 的性能优化

任务一的 Mapper 与 Reducer 的任务相当简单：Mapper 负责分词，只需要将分词结果存放在 context 中；Reducer 负责将分词结果原封不动的直接输出到本地文件。

然而我们考虑这样的一个问题，当对任意几段文本的分词结果出现以下情况时：

张三 李四 王五
张三 李四 王五
张三 王五 李四

我们可以发现在后续分析同现关系时，这样的几段分词结果所展现的同现关系是完全相同的，然而我们却需要多次处理完全相同的分词结果。这种情况的存在无疑大大增加了后续数据处理的复杂度，我们应该在任务一消除这种情况。使这三种分词结果在任务一的输出文件中展示为：

张三 李四 王五,3

表示：张三、李四、王五三个人同时出现在同一段落的次数为 3 次。

将前两行数据归一化的方法很简单，只需要在 Reducer 中对出现的次数进行简单的累加计数即可。将第三行与前两行数据归一化则需要对每一行数据进行排序操作。这里我们省去后续的排序过程，直接在保存分词结果时使用哈希集合进行集合构建，最终的输出结果必然按照相同顺序排列。

```java
//java code
public void map(Object key, Text value, Context context)
                throws IOException, InterruptedException {
    String line = value.toString();
    List<Term> terms = DicAnalysis.parse(line).getTerms();
    HashSet<String> termSet = new HashSet<>();
    for (Term tmp : terms) {
        if (tmp.getNatureStr().equals("userDefine")) {
            termSet.add(tmp.getName());
        }
    }
    StringBuffer posting = new StringBuffer();
    for (String tmp : termSet) {
        posting.append(tmp);
        posting.append("␣");
    }
    if (posting.length() > 0) {
        posting.deleteCharAt(posting.length() - 1);
        String post = posting.toString();
        context.write(new Text(post), one);
    }
}

public void reduce(Text key, Iterable<IntWritable> values, Context context)
                throws IOException, InterruptedException {
    int sum = 0;
    for (IntWritable i : values) sum += i.get();
    result.set(sum);
    context.write(key, result);
}
```

经过这样简单的计数统计，我们对输出数据进行了最大程度上的精简压缩，有效的加快了后续运算速度。同时我们已经可以在任务一的输出结果中相当直观的观察出大致的人物同现次数，人为直观的判断金庸武侠

世界的主角。

2.3 结果展示

Figure 1: 任务一结果展示

3 任务二：人名同现统计

因为任务一中对输出结果进行了高度的精简压缩，任务二就变得相当直观而简单。我们所需要做的就是对任务一的每一行输出结果进行不同人名之间两两组合，将组合结果与该组合一共出现的次数进行统计求和并输出。由于在任务一中已经除去了每一段文本中分词结果的重复元素，实验二中我们可以放心大胆的世界两两组合人名。

3.1 人名组合与统计

算法 3 人名组合与统计

输入: 分词结果
输出: 人名组合对

```
function GETUNIT(< names, num >)
    for i in names do
        for j in names do
            if i is not j then
                res.add(< (i, j), num >)
            end if
        end for
    end for
    return res
end function
```

可以看出，经过人名组合后我们已经获得了可以进行加和操作的键值对，任务二的核心步骤已经完成，可见前期优化极大方便了后期处理。

3.2 MapReduce 实现

3.2.1 Mapper

在任务二中，Mapper 负责将每一行数据中的人名进行同现配对，即任意人名会和与其在同一段落出现的所有其他人名配对。以该配对为关键字，该配对出现的次数为键值记录到 context 中。思路相对简单易懂。需要注意的是，由于每行数据中同时记载了该数据出现的次数，需要合理划分字符串。

```
//java code
public void map(Object key, Text value, Context context)
        throws IOException, InterruptedException {
    String[] strarr = value.toString().split(",");
    one.set(Integer.parseInt(strarr[1]));
    StringTokenizer itr = new StringTokenizer(strarr[0]);
    while (itr.hasMoreTokens()) {
        String base = itr.nextToken();
        StringTokenizer itr2 = new StringTokenizer(strarr[0]);
        while (itr2.hasMoreTokens()) {
            String extra = itr2.nextToken();
            if (base.equals(extra)) continue;
            word.set("<" + base + "," + extra + ">");
            context.write(word, one);
        }
    }
}
```

3.2.2 Reducer

Reducer 与任务一职责相同，将键值累加求和并输出，容易理解。

```
//java code
public void reduce(Text key, Iterable<IntWritable> values, Context context)
        throws IOException, InterruptedException {
    int sum = 0;
    for (IntWritable val : values) {
        sum += val.get();
    }
    result.set(sum);
    context.write(key, result);
}
```

3.3 结果展示

Figure 2: 任务二结果展示

4 任务三：特征处理：人物关系图构建与特征归一化

4.1 邻接表的图的描述及特征归一化

任务二中得到了两个人物共同出现的次数，但是使用边来就表示图不利于求的 pagerank 值，所以应该将边的表示转换为邻接表表示，且将共同出现次数归一化得到共同出现概率。

任务涉及两个方面：

1. 邻接表的构建：

设收到输入为“<name1,name2>,num”“<name1,name2>,num”，则构建邻接表为：“name1 <name2, num>; <name3, num>”第一个节点是当前节点，后续为它的邻居节点及共现次数

2. 权值归一化：

对共现次数进行操作，将当前节点后的所有节点共现次数除以总次数；

4.2 伪代码实现

Mapper 操作将 key:<name1, name2>, value:name 转换为 key:name1, value:<name2, num> 那么通过 shuffle 阶段操作，进入 reduce 之前将相同的 node 传入 reducer，也就得到了这个图的邻接表，之后在 reduce 节点进行归一化操作，简单分为两步：第一步得到某一人名的邻接表中与它相邻的共现次数总和；第二步更新邻接表中的值，将共现次数除以总次数实现归一化。

4.2.1 Reducer

算法 4 权值归一化

输入: 人名组合对
输出: 权值归一化后的邻接表

```
1: function REDUCER(key, values, context)
2:     sumgets0
3:     for val in values do
4:         sum+ = val.num
5:         res.append(val)
6:     end for
7:     String[]normalization = out.split()
8:     for x in normalization do
9:         ot1.append(x.nane + ", " + x.val/sum)
10:    end for
11:    context.write(key, newText(out1.toString()))
12: end function
```

4.3 结果展示

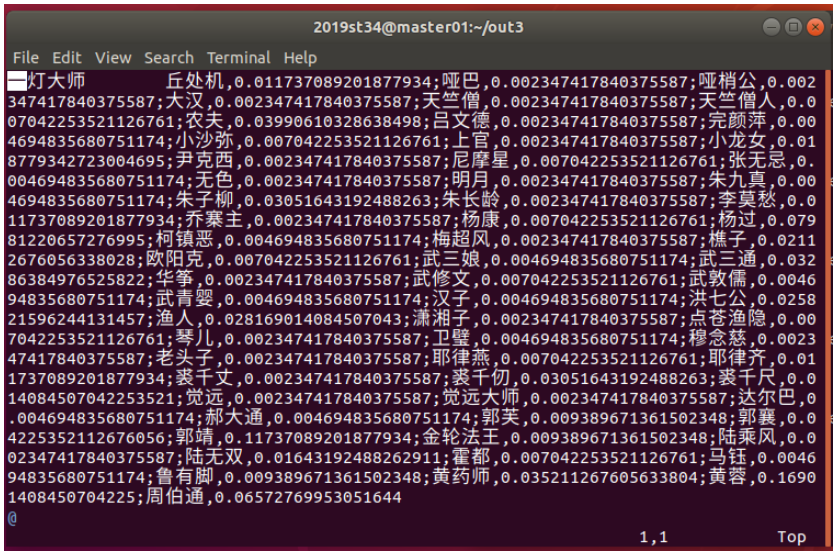

Figure 3: 任务三输出人物关系邻接表

5 任务四：基于人物关系图的 PageRank 计算

5.1 主要流程

在主函数中调用 PageRankIter 和 PageRankViewer 函数，其中，PageRankIter 是执行 PageRank 计算的主体，需要进行多次迭代，PageRankViewer 负责执行 PageRank 结果的键值对输出。具体的，Iter 主要

分为两部 map 和 reduce：Map 函数会读取每一个人物对应的 pagerank 值以及人物连接关系，对于关系列表，map 在收到后不做处理直接发送给 reduce 节点，而对于 pagerank 值，乘以邻接表中每一个人物的共现概率后与该人物组成键值对发送到 reduce 端。Reduce 函数接受每一个人名对应的键值对后计算更行后的 pagerank 值，记录其关系列表，然后输出到文件中去以作为下一次迭代 map 的输入。对于经典的 PageRank 算法，计算公式为：

$$PR(p_i) = \frac{1-d}{N} + d \sum_{p_j \in M(p_i)} \frac{PR(p_j)}{L(p_j)}$$

其中 M(pi) 表示所有连接到 pi 的点的集合。但是考虑这次实验具有特殊性，归一化后得到的共现次数实际上就是贡献概率，所以用 weight(pi,pj) 代替 1/L(pi)，

$$PR(p_i) = \frac{1-d}{N} + d \sum_{p_j \in M(p_i)} PR(p_j) * weight(p_i, p_j)$$

5.2 伪代码实现

5.2.1 PageRankIter

算法 5 PageRankIterMapper

输入: key : 人名，$value$ $pr\_init, link\_list$
输出: key : 人名，$value$: $cur\_rank, link\_list$

function MAP($key, values, context$)
  $line = values[1]$
  for $tuple$ in $line$ **do**
    $context.write(tuple[0], tuple[1] * curPR)$
  end for
  $context.write(key, values[1])$
end function

算法 6 PageRankIterReduce

输入: key 人名,$values$: $pagerank\_list|link\_list$
输出: key : 人名,$values$: $cur\_pagerank|link\_list$

function REDUCE($key, values, context$)
  for pr in $pagerank\_list$ **do**
    $pagerank+ = pr$
  end for
  $pagerank = 0.15 + 0.85 * pagerank$
  $context.write(key, pagerank|link\_list)$
end function

算法 7 PageRankViewerMapper

输入: key 人名,$values$: $cur\_pagerank|link\_list$
输出: key : 人名,$pagerank$

```
1: function MAP(key, values, context)
2:     context.write(key, pagerank)
3: end function
```

5.3 设计细节

1. Iter 的 map 算法需要传递两种 value，一种是 double 类型，一种是 list，将他们全部转为 string，提升了通信效率，在 reduce 函数中再进行解码。经过多次运行统计，每一次 PageRankIter 在集群上运行时间为 20s 到 30s

2. pagerank 值计算需要迭代计算，每次将前一次迭代的结果保存在文件中以便于后一次迭代的计算，另外，实现迭代的过程在 main 函数中，每次迭代就创建一个新的 job，因为保证了前一次迭代 reduce 输出的结果格式与后一次 map 的输入格式是一样的，所以可以保证迭代的正确执行

5.4 实验结果展示

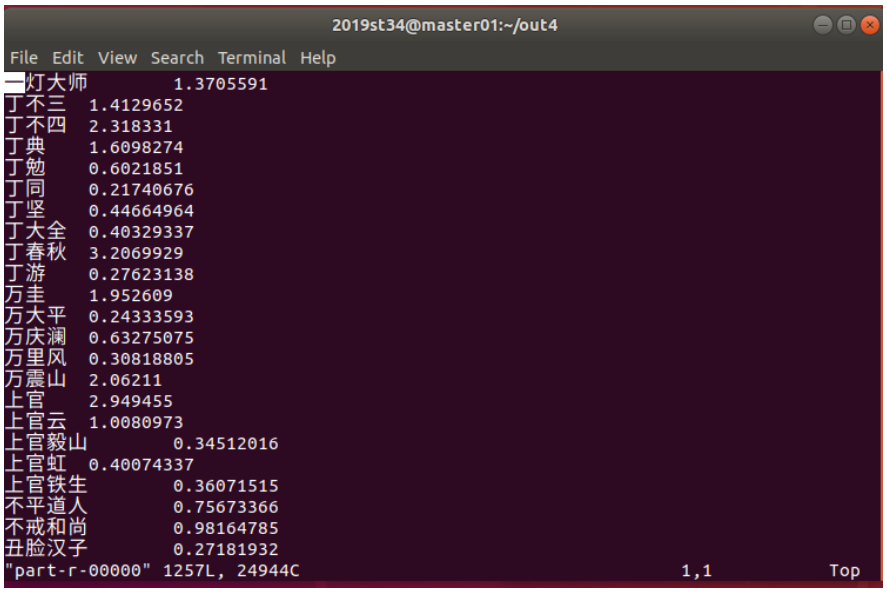

Figure 4: 人物 pagerank 值

6 任务五：在人物关系图上的标签传播

6.1 标签传播

标签传播 (Label Propagation) 是一种半监督的图分析算法，能为图上的顶点打标签，进行图顶点的聚

类分析，从而在一张类似社交网络图中完成**社区发现（Community Detection）**。社区是指网络中节点的集合，这些节点内部连接较为紧密而外部连接较为稀疏。根据各社区节点集合彼此是否有交集，社区分为非重叠型 (disjoint) 社区和重叠型 (overlapping) 社区。

6.1.1 标签传播的基本思想

标签传播算法就是一种不重叠社区发现的经典算法。**其基本思想是：每个结点的标签应该和其大多数邻居的标签相同**。因此，将一个节点的邻居节点的标签中数量最多的标签作为该节点自身的标签（bagging 思想）。给每个节点添加标签（label）以代表它所属的社区，并通过标签的“传播”形成同一个“社区”内部拥有同一个“标签”。

标签传播算法和 Kmeans 的本质非常类似，在标签传播的每轮迭代中，节点被归属于哪个社区，取决于其邻居中累加权重最大的 label（取数量最多的节点列表对应的 label 是 weight=1 时的一种特例），而 Kmeans 则是计算和当前节点“最近”的社区，将该节点归入哪个社区。

6.1.2 标签传播算法的特点

标签传播算法最大的优点是其算法过程比较简单，算法速度非常快。算法利用网络的结构指导标签的传播过程，在这个过程中无需优化任何函数。在算法开始前我们不必要知道社区的个数，随着算法的迭代，在最终的过程中，算法将自己决定社区的个数。

另一方面，划分结果不稳定、随机性强是这个算法致命的缺点。具体体现在：

- 随机更新顺序。节点标签更新顺序随机，但是很明显，越重要的节点越早更新会加速收敛过程；
- 随机选择。如果一个节点的出现次数最大的邻居标签不止一个时，随机选择一个标签作为自己标签。这种随机性可能会带来一个雪崩效应，即刚开始一个小小的聚类错误会不断被放大。

6.1.3 标签传播算法过程

标签传播算法的过程如下：

1) 先给每个节点分配对应标签，每个节点的标签为其自身，即节点 i 对应标签 i；
2) 遍历节点，找到对应节点的邻居，获取此节点邻居的标签，找到出现次数最大的标签。若出现次数最多标签不止一个，则随机选择一个标签替换成此节点的标签；
3) 重复执行步骤 2)，直到节点标签变化收敛或者达到设定的迭代次数。

算法的伪代码如下：

算法 8 标签传播算法

输入: 图的邻接表 $adjList$, 最大迭代次数 $maxIter$
输出: 带有分类标签的节点 $vertices$

```
1: function LabelPropagation(adjList, maxIter)
2:     for vertex in adjList do
3:         vertex.label ← vertex.id
4:     end for
5:     repeat
6:         for vertex in adjList do
7:             统计 vertex.neighbors 中标签数量最多的标签 label
8:             vertex.label ← label
9:         end for
10:    until convergence or reach maxIter
11:    return adjList.vertices
12: end function
```

6.1.4 标签传播算法的改进

在原始版本的标签传播算法中，当一个节点的出现次数最大的邻居的标签不止一个时，随机选择其中一个标签作为新的标签。为了缓解这种随机性对算法造成的影响，可以考虑将边权作为参考因素。在本次实验中，边权的含义是归一化后的人物同现概率，我们可以认为边权越大，两个节点属于同一个社区的概率越大，因此我们可以将边权作为标签从边的源点传播到边的终点的概率。

这种情况下，在一次迭代过程中，每个节点需要更新的参数将不再是标签，而是标签的概率分布。具体而言，节点将所有邻居节点的标签分布加权后作为自己新的标签分布；算法结束时，取标签概率分布中概率最大的标签作为节点最终的标签。算法伪代码如下：

算法 9 带边权的标签传播算法

输入: 图的邻接表 $adjList$, 最大迭代次数 $maxIter$
输出: 带有分类标签的节点 $vertices$

```
1: function WeightedLabelPropagation(adjList, maxIter)
2:     for vertex in adjList do
3:         vertex.distribution ← 0
4:     end for
5:     repeat
6:         for vertex in adjList do
7:             for neighbor in vertex.neighbors do
8:                 vertex.distribution += neighbor.weight * neighbor.distribution
9:             end for
10:            normalize(vertex.distribution)
11:        end for
12:    until convergence or reach maxIter
13:    for vertex in adjList do
14:        vertex.label ← vertex.distribution.indexOfMaxValue
15:    end for
16:    return adjList.vertices
17: end function
```

6.1.5 参数更新方式

标签传播算法的标签或标签分布参数的更新可分为同步更新与异步更新两种方式。

同步更新是指对于节点 x，在第 t 代时，根据其所有邻居节点在第 t−1 代时的社区标签对其标签进行更新，即：

$$C_x(t) = f(C_{x_1}(t-1), C_{x_2}(t-1), C_{x_3}(t-1), ..., C_{x_k}(t-1)) \tag{6.1}$$

其中 $C_x(t)$ 表示节点 x 在第 t 次迭代时的社区标签，函数 f 的作用是取邻居节点中社区标签最多的。

然而，同步更新的方式存在一个问题，当遇到二分图等网络结构时会出现标签震荡，如图所示：

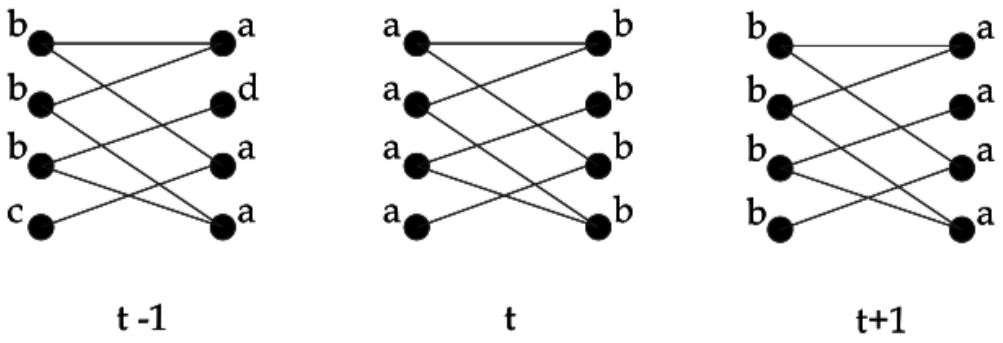

Figure 5: 同步更新在二分图上引起的震荡

在上图中，两边的标签会在社区标签 a 和社区标签 b 不停地震荡。

对于异步更新方式，其更新公式为：

$$C_x(t) = f(C_{x_{i1}}(t), ..., C_{x_{im}}(t), C_{x_{i(m+1)}}(t-1), ..., C_{x_{ik}}(t-1)) \tag{6.2}$$

其中，邻居节点 $x_{i1}, ..., x_{im}$ 的社区标签在第 t 代已经更新过，使用其最新的标签参数；而邻居节点 $x_{i(m+1)}, ..., x_{ik}$ 在第 t 代时尚未更新，则对于这些邻居节点仍使用其在第 t-1 代时的标签参数。

6.1.6 算法收敛条件

理想情况下，若图中节点的社区标签不再改变，此时迭代过程便可以停止。但是这样的收敛条件存在一个问题：若对于某个节点，其邻居节点中存在两个或者多个最大的社区标签，则其更新时所属的社区是随机选取的。这样，按照上述收敛条件，算法会一直执行下去。

可以对上述的迭代终止条件修改为：对于每一个节点，在其所有的邻居节点所属的社区中，其所属的社区标签是最大的。

6.2 Hadoop 实现

MapReduce 编程模型适用于标签传播算法的理论依据是：**节点标签的更新只与其邻居的标签有关，**因此满足数据的局部相关性。基于这个基本事实，我们可以对模型进行分片，将图切割为多个部分交给多个计算节点进行并行处理，而邻接表的数据结构恰好十分便利于标签传播任务下的图分割。

6.2.1 设计思路

受大规模分布式深度学习中的**参数服务器（Parameter Server）**架构启发，针对 Hadoop 框架，也可以设计一种类似参数服务器的参数同步架构，只不过在这里参数服务器的载体是 HDFS 文件系统。对于标签传播算法，设计思路如下：

- 所有图节点的标签分布、收敛状态等全局参数存储在“参数服务器”（即 HDFS 上的一个或多个文件）中；
- 采用模型并行的分布式策略，每个计算节点负责图的部分节点的计算；
- 在一次迭代中，map 计算节点从“参数服务器”上拉取**（pull）**计算所需的参数，包括图节点的标签分布参数等；
- 在 map 计算节点完成计算后，将新的标签分布参数以及收敛状态参数发送**（push）**至“参数服务器”；
- “参数服务器”可采用同步更新（即等待所有 map 节点发来新的参数后才能更新全局参数）或异步更新（即无需等待所有 map 节点发来新的参数便可更新全局参数）两种参数更新方式。

参数服务器架构如图所示：

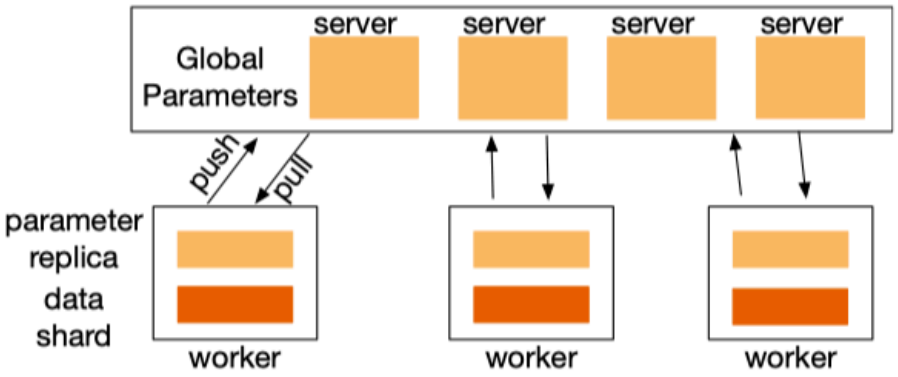

Figure 6: 参数服务器架构

6.2.2 标签传播的 MapReduce 算法

同步更新算法利用了 Hadoop 框架在 map 节点与 reduce 节点之间的**同步屏障**：当所有 map 节点均完成计算后，reduce 节点才开始执行。利用这一点，同步更新算法不在 map 节点而是在 reduce 节点中将参数 push 到“参数服务器”，这样可以保证每个 map 节点从“参数服务器”拉取的参数都是上一轮迭代的参数。

而异步更新算法与同步更新算法的显著区别在于，异步更新算法在 map 阶段便可将新的参数传输至“参数服务器”中，因此一个 map 节点无需等待所有其他 map 节点的执行，只要其完成了自己的计算任务就可以立即执行数据传输任务。这在一定程度上缓解了因多个 reduce 节点同时向参数服务器传输新参数所造成的网络瓶颈。此外，异步更新算法还可以有效避免标签传播算法的标签震荡问题。

同步更新的带边权标签传播算法的 MapReduce 伪代码如下图所示：

算法 10 同步更新的带边权标签传播 MapReduce 算法

```
1: function MAP(key : vertex, value : neighbors)
2:     vertex.distribution ← 0
3:     for neighbor in neighbors do
4:         neighborDistribution ← pull(neighbor.distribution)
5:         vertex.distribution += neighbor.weight * neighborDistribution
6:     end for
7:     normalize(vertex.distribution)
8:     emit(vertex, vertex.distribution)
9: end function
10: function REDUCE(key : vertex, values : [distribution])
11:     distribution ← values[0]
12:     push(key, distribution)
13:     emit(vertex, distribution.indexOfMaxValue)
14: end function
```

异步更新的带边权标签传播算法的 MapReduce 伪代码如下图所示：

算法 11 异步更新的带边权标签传播 MapReduce 算法

```
1: function MAP(key : vertex, value : neighbors)
2:     vertex.distribution ← 0
3:     for neighbor in neighbors do
4:         neighborDistribution ← pull(neighbor.distribution)
5:         vertex.distribution += neighbor.weight * neighborDistribution
6:     end for
7:     normalize(vertex.distribution)
8:     push(key, distribution)
9:     emit(vertex, vertex.distribution)
10: end function
11: function REDUCE(key : vertex, values : [distribution])
12:     distribution ← values[0]
13:     emit(vertex, distribution.indexOfMaxValue)
14: end function
```

6.2.3 性能优化

事实上，Hadoop 对需要进行多次迭代的机器学习算法不是特别友好，其性能瓶颈主要体现在磁盘 I/O 和网络 I/O 两个方面。原因在于，在 Hadoop 中迭代的控制是在 job 间的，因为 Hadoop 的一个 job 只能是一个 map-reduce 对，而迭代算法的计算需要多次 map-reduce，因此需要启动多个 job；由于 job 无法复用，每一个 job 完成之后需要把这一次迭代的结果写进 hdfs 文件中，下一次迭代启动新的 job 时又要从 hdfs 中读取上一次迭代的输出结果，造成了很多本来不必要的磁盘 I/O。另一方面，采用类似参数服务器这样的参数同步架构时天然面临着在参数服务器端的网络瓶颈，因为当多个计算节点同时向参数服务器发送或拉取新的参数时，参数服务器的通信压力巨大，集群中有限的带宽无法满足这样的网络 I/O 需求。

下面主要从磁盘 I/O 以及网络 I/O 两个方面对基于 Hadoop 的标签传播进行优化：

- 磁盘 I/O：

  在 hadoop 标签传播任务下，hdfs 的一个或多个文件充当了参数服务器的角色，每次 push 和 pull 参数时存储全局参数的节点上都会发生大量磁盘 I/O，因此我们可以考虑提高磁盘 I/O 的效率。注意到当一个 map 节点完成计算后，大部分参数都不需要更新，只有该 map 节点所涉及的图节点的标签分布参数需要更新；更进一步说，如果某个图节点的标签分布参数没有发生变化或变化很小，那么这样的参数也可以不进行更新。

  如何才能做到更新部分参数？由于 hdfs 文件不支持随机位置写入，我们之前采取的做法是每次更新参数时将文件中的所有参数进行覆写。一个改进的方案是，不对整个参数文件进行覆写，而是直接在原有文件的末尾追加新的参数，这样可以节省很多写文件的时间。在该优化方案下，一个需要解决的问题是参数文件中的参数存在多个不同迭代次数下的中间值。为了保持数据的一致性，我们可以给参数打上时间戳（即参数所处的迭代次数）。

- 网络 I/O：可以从减少通信量以及减少网络通信次数的角度进行优化。

  1）减少通信量：我们可以只向参数服务器发送需要更新的参数，那些标签分布参数变化较小的图节点可以不与参数服务器进行参数传输，这样可以极大地减少参数传输的数量，缓解通信的压力。

  2）减少网络通信次数：在异步更新的标签传播 MapReduce 算法中，标签分布参数在 map 阶段便 push 到参数服务器上，这样会导致一个计算节点向参数服务器发送新参数的次数与该计算节点中参数需要更新的图节点的个数相同。发生多次参数通信也意味着多次磁盘 I/O，因为参数服务器端需要分多次写入新的参数。为了提高效率，我们可以将一个 map 节点上需要传输至参数服务器的参数一次性进行传输。具体而言，我们将 push 参数的操作转移至 Combiner 中，Combiner 执行时，相应 map 节点上的所有计算任务已经完成，因此我们可以在 Combiner 中将该节点上所有需要更新的参数一次性传输至参数服务器中。

7 任务六：分析结果整理（选做）

7.1 全局排序

对于经过 PageRank 算法处理得到的人物 PageRank 值，我们要求按照 PageRank 值进行全局排序。

我们知道 MapReduce 程序中保证同一个 Reducer 内的 Key 是有序的，但并不保证全局有序，所以我们需要进行一些额外处理实现全局排序。

而对于输入文件中按照人物 +PageRank 的格式，我们得到的 key 为人物姓名，value 为 PageRank 值，所以我们还需要将 key 和 value 进行调换。

7.1.1 算法

(1) 算法思想：使用一个 Reducer 进行排序。

(2) 算法描述：MapReduce 保证同一个 Reducer 内的 Key 是有序的，却不保证全局有序。但如果我们将所有的数据全部发送到一个 Reducer，那么就可以实现结果全局有序。

(3) 算法特点：以上算法采用所有的数据都发送到一个 Reducer 进行排序，思路十分简洁有效，但同时有很大局限性：这样不能充分利用集群的计算资源，而且在数据量很大的情况下，很有可能会出现 OOM 问题

算法伪代码如下：

算法 12 全局排序

输入: 人物 Pagerank 值对
输出: 升序排列人物 PageRank 值对

```
1: function TotalSortMap(< name, pagerank >)
2:     key = pagerank
3:     value = name
4:     emit(pagerank, name)
5: end function
6: Reducer.num = 1
7: function TotalSortReduce(< pagerank, name >)
8:     emit(name, pagerank)
9: end function
```

7.1.2 MapReduce 实现

Mapper 在此任务中，Mapper 负责读取输入文件中每一行数据，得到的初始键值对中，人名为关键字，PageRank 值为键值。

但是这样将键值对发送至 Reducer 中，并不能按 PageRank 值排序，因此我们调换键值对关键字和键值顺序，记录到 context 中，发射给 Reducer。

```java
//java code
public static class TotalSortMapper extends
            Mapper<Text, Text, FloatWritable,  Text> {
        @Override
        protected void map(Text key, Text value,
                           Context context) throws IOException, InterruptedException {
            FloatWritable pagerank = new FloatWritable(Float.parseFloat(value.toString()));
            context.write(pagerank, key);
        }
    }
```

Reducer 将 Reducer 数目置为 1，即可在一个 Reducer 中实现所有键值对的全局排序，再次调换键值对关键字和键值顺序，按照原来的格式输出。

```java
//java code
public static class TotalSortReducer extends
            Reducer<FloatWritable, Text, Text, FloatWritable> {
        @Override
        protected void reduce(FloatWritable key, Iterable<Text> values,
                              Context context) throws IOException, InterruptedException {
            for (Text value : values)
                context.write(value, key);
        }
    }
```

7.1.3 算法优化

优化思想 MapReduce 默认的分区函数是 HashPartitioner，其实现的原理是计算 map 输出 key 的 hashCode，然后对 Reducer 个数求模，这样只要求模结果一样的 Key 都会发送到同一个 Reducer。
如果我们能够实现一个分区函数，使得所有 pagerank 小于一定值的数据都发送到 Reduce 0；所有 pagerank 值在一定范围内的数据都发送到 Reduce 1；其余的 pagerank 值都发送到 Reduce 2；
这就实现了 Reduce 0 的数据 pagerank 值一定全部小于 Reduce 1，且 Reduce 1 的数据 pagerank 值全部小于 Reduce 2，再加上同一个 Reduce 里面的数据局部有序，这样就实现了数据的全局有序。

伪代码 算法伪代码如下：

mapreduce 实现 由于优化后排序实现除了自定义的 TotalSortPartitioner 之外，其余的和第一种实现一样，所以仅给出此部分代码。

算法 13 全局排序

输入: 人物 Pagerank 值对
输出: 升序排列人物 PageRank 值对

function TotalSortMap($< name, pagerank >$)
    $key = pagerank$
    $value = name$
    $emit(pagerank, name)$
end function
function TotalSortPartition($< pagerank, name >$)
    for i in $pagerank$ **do**
        if $i < 10$ **then return** 0
        end if
        if $10 <= i < 20$ **then return** 1
        end if
        if $i >= 20$ **then return** 2
        end if
    end for
end function
$Reducer.num = 3$
function TotalSortReduce($< pagerank, name >$)
    $emit(name, pagerank)$
end function

```java
//java code
public static class TotalSortPartitioner extends Partitioner<FloatWritable, Text> {
    @Override
    public int getPartition(FloatWritable key, Text value, int numPartitions) {
        float number = Float.parseFloat(key.toString());
        if (number < 10) {
            return 0;
        } else if (number < 20) {
            return 1;
        } else {
            return 2;
        }
    }
}
```

7.1.4 结果展示

7.2 标签聚集

对于给定的人物及其标签序列，我们需要将相同标签的人物进行聚集。

同样由于格式的设置，我们要根据标签进行聚集，必须在 map 操作中进行键值对中 key 和 value 的调换，随后将键值对发射至 Reducer 中。

由于相同 key 的键值对会经过 Partition 到达同一个 Reducer，所以我们在 Reducer 中将相同 key 的键值对中 value 集合输出出来，即可完成任务。

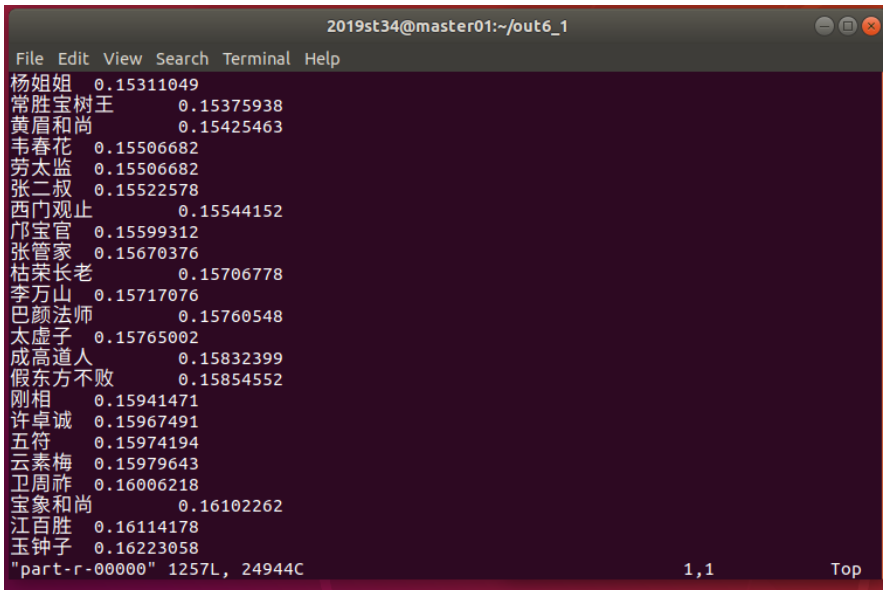

Figure 7: 任务六全局排序结果展示

7.2.1 算法

此算法思想简单，无需优化，算法伪代码如下：

算法 14 标签聚集

输入: 人名及标签对
输出: 标签及对应人名集合

```
function ClusterMap(< name, label >)
    key = label
    value = name
    emit(label, name)
end function
function ClusterReduce(< label, names >)
    emit(label, names)
end function
```

7.2.2 MapReduce 实现

Mapper 在此任务中，Mapper 负责读取输入文件中每一行数据，得到的初始键值对中，人名为关键字，标签值为键值。

但是这样将键值对发送至 Reducer 中，并不能将相同标签值人物聚集在一起，因此我们调换键值对关键字和键值顺序，记录到 context 中，发射给 Reducer。

```java
//java code
public static class ClusterMapper extends
            Mapper<Text, Text, IntWritable, Text> {
        @Override
        protected void map(Text key, Text value,
                           Context context) throws IOException, InterruptedException {
            IntWritable label = new IntWritable(Integer.parseInt(value.toString()));
            context.write(label, key);
        }
    }
```

Reducer 经过 Partition 操作后，相同 key 值（标签值）的键值对被转化为一个 key 对应一个 Iterable<Text> 类型的 value 集合，也正是我们需要的人名集合。当然我们为了规范输出格式，将其组织为用逗号隔开人名的文本 Text，最后输出键值对。

```java
//java code
public static class ClusterReducer extends
            Reducer<IntWritable, Text, IntWritable, Text> {
        @Override
        protected void reduce(IntWritable key, Iterable<Text> values,
                              Context context) throws IOException, InterruptedException {
            StringBuilder allkey = new StringBuilder();
            for (Text value : values) {
                allkey.append(value.toString());
                allkey.append(",");
            }
            allkey.setLength(allkey.length() - 1);
            context.write(key, new Text(allkey.toString()));
        }
    }
```

7.2.3 结果展示

8 补充任务：生成人物关系图

8.1 任务描述

根据任务 5 的标签传播和任务 2 的人物同现统计结果作为输入，构建人物关系图。而通过人物关系图的构建,我们可以检验任务 3 的图构建、任务 4 的 PageRank 值计算和任务 5 的标签传播算法结果是否正确。

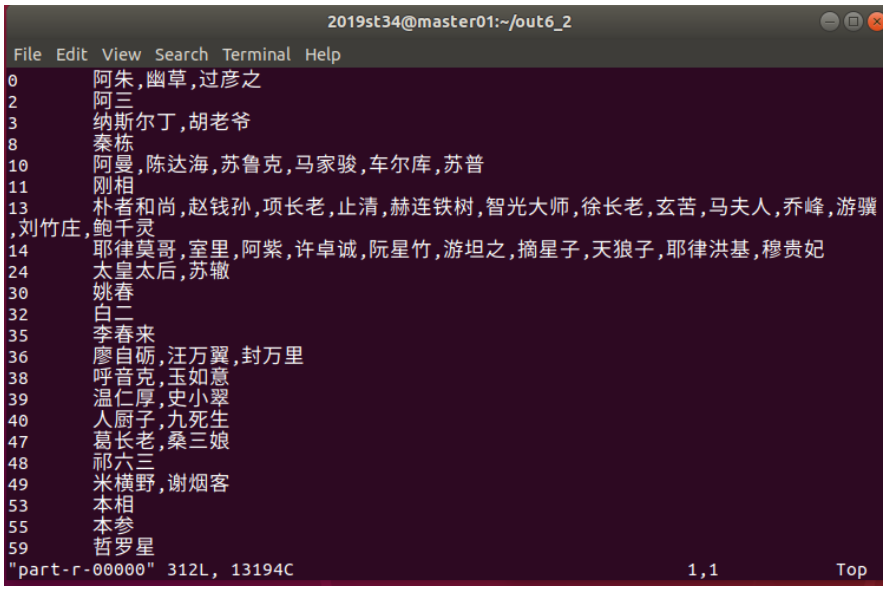

Figure 8: 任务六标签聚集结果展示

8.2 构图思路

有以下原则：人物名字的大小由人物顶点的度数确定，人物标签的颜色根据标签传播算法的分析结果确定。在此基础上，我们选用 Gephi 工具作图。

8.3 构图过程

8.3.1 数据预处理

在一开始运行 Gephi 时，遇到了导入数据格式的问题，首先要求导入电子表格为 csv 格式，同时节点和边表格的格式均有要求。因此我们通过编写 Python 脚本 (见附件 gephi.py) 实现了导入数据的转化，最终得到了如下格式的数据：

节点表格：包含 ID（自动生成）和 Label（即姓名）、Class（即标签值）

边表格：包含 Source（有向图源节点 ID）和 Target（有向图目标节点 ID），此处将共现次数转化为多条边

Table 1: 节点表格

ID	Label	Class
0	阿凡提妻	4
1	巴郎星	25
2	白阿绣	29
3	熊元献	44

Table 2: 边表格

Source	Target
1179	320
1179	1281
1179	1281
1179	1281

8.3.2 图形生成

通过导入数据后，进入概览页面，在外观栏中设置节点颜色为 Class 属性，用 13 种颜色表示对应标签（占比大于 0.0095），大小根据节点度数进行调节。

随后在布局栏选择 Force Atlas 算法，根据节点连接情况产生斥力和引力，重新生成布局，如图 9 所示。

Figure 9: 生成布局

随后在预览页面，可以看到最终的布局如图 10 所示：明显看出最外围为孤立点，中间为各类标签节点聚集图像。

最后给各个节点加上标签，显示如图 11 所示。

仔细查看可以知道标签相同节点多为同一本书中人物，而每种颜色中最大的节点多为该书中主角，如韦小宝、张无忌、郭靖、袁承志、狄云等，此也与任务 4、5 结果相符。

如图中韦小宝、袁承志为例，如图 12 所示：

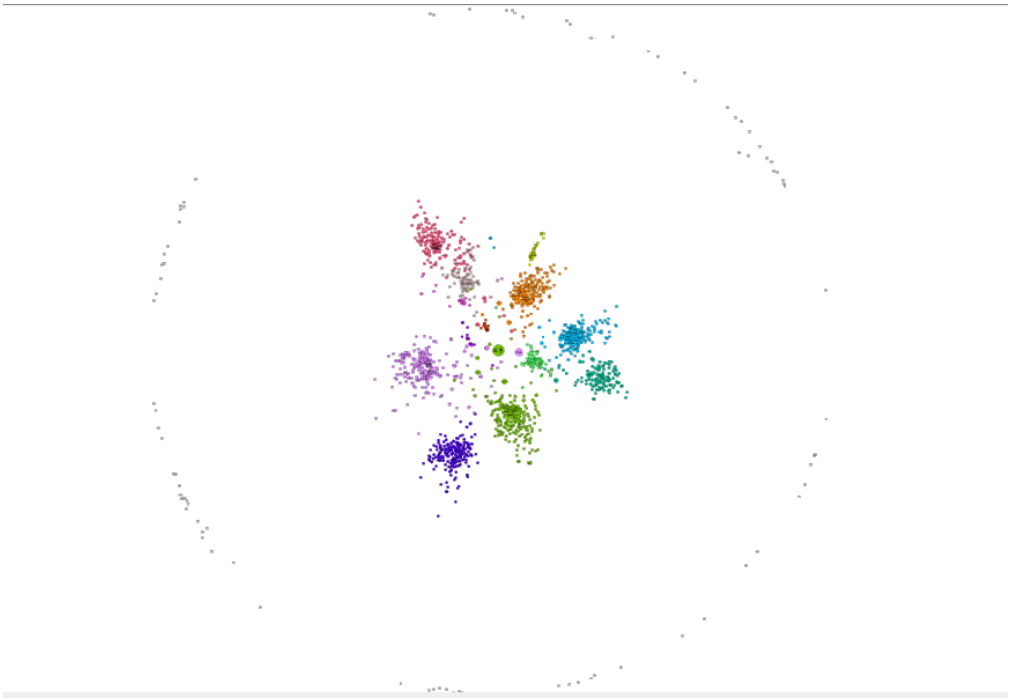

Figure 10: 最终布局

9 Spark 实现人名同现统计

利用 Spark 实现人名同现统计的整体思路与 Hadoop 一致：进行两两人名配对。将人名对与数据出现次数合成键值对后利用 Reduce 完成加和计数。

与 Hadoop 不同的是，Spark 的高级 API 剥离了对集群本身的关注，我们所需要构思的只有应用所要做的计算本身。因此代码更加简洁明了。同时在使用 Spark 中的 flatMap 对数据进行扁平化操作时，我们需要将所有数据整合为一个序列化的数据而不是单个数据项，这一特性与 Hadoop 有所不同。

在考虑了 Spark 的以上特性后，我们可以很容易的编写 Scala 代码，Scala 语言相对于 Java 更加精简，我们只需要专注于算法实现，可以忽略集群或者各个节点的考量。

```
1  //Scala code
2  val lines = input.flatMap(line => {
3      var res = Seq[(String, Int)]()
4      val a = line.split(",")(0)
5      val b = line.split(",")(1).toInt
6      val c = a.split("␣")
7      for (i <- 0 until c.length - 1) {
8          for (j <- 0 until c.length - 1) {
9              if (i != j) {
10                 res = res :+ ("<" + c(i) + "," + c(j) + ">", b)
11             }
12         }
13     }
```

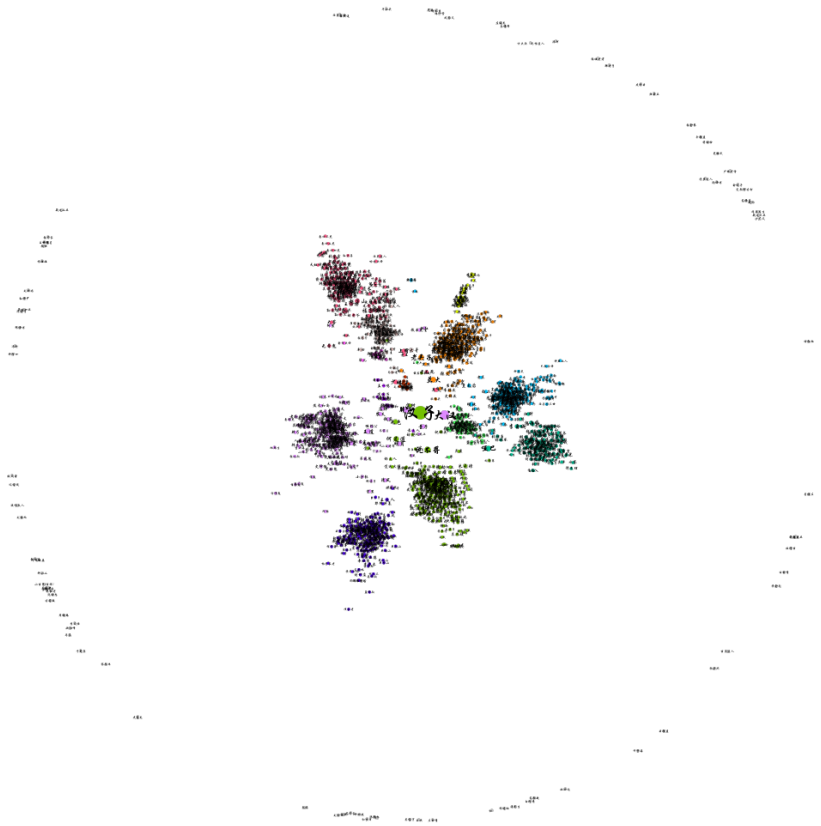

Figure 11: 结果图像

```
    res
}).reduceByKey { case (x, y) => x + y }
lines.saveAsTextFile("output2")
```

实验结果与 Hadoop 集群运行结果一致，截图略。

10 Spark 实现人物关系图构建与特征归一化

10.1 主要流程

考虑到实现归一化的第一步就是将各个人物邻接表中的权值加起来，所以我们将建立一个 RDD 用来存储（人物，权值和）的键值对第二步就是将邻接表中的权值除以权值和得到归一化结果

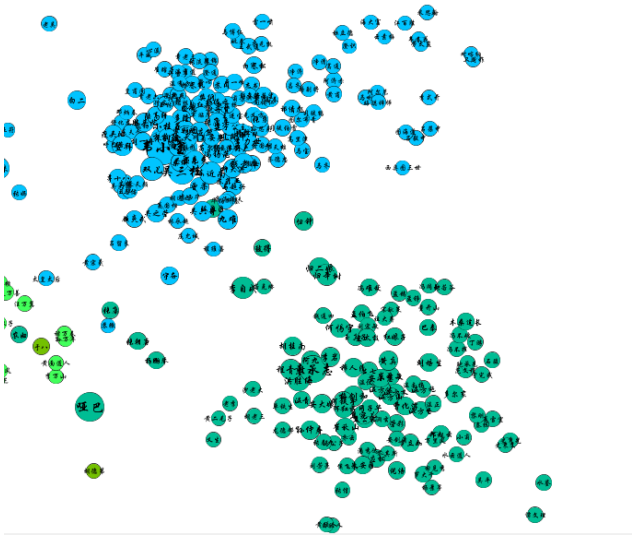

Figure 12: 主角图像

10.2 实现细节

正真实现远比直观上的困难，因为之前并未接触过 scala 语言，所以调试了很久也查了很多资料，最后才得出结果，这里简单阐述下实现细节：

1. 因为从实验二传入的格式是类似（< 狄云，戚芳 >,1）这个格式的，所以第一步就设置正则表达式对其进行切分，在切分之后，要获得所需内容的正确下标，这是至关重要的，实验一开始因为没搞清楚下标造成了许多想不通的 bug

2. 第一步需要将权值加起来，也就是把 value 加起来，在 scala 语言中只需要对 RDD 进行一句 reduceByKeycase(x,y) => x + y 操作就可以得到结果

3. 在第二步中需要用到第一步的 RDD 内容，而直接在一个 RDD 中调用另一个 RDD 是不符合规范的，编译器会报错，需要使用 broadcast 函数即将一个 RDD 的内容进行广播使得其他 RDD 可以知道它的内容从而使用之。

10.3 代码展示

```
1   val inputFile = "output2/part-00000"
2   val input = sc.textFile(inputFile)
3   val sum = input.flatMap(line => {
4     var res = Seq[(String, Int)]()
5     val name = line.split("(\\(<)|,|(>,)|\\)")(1)
6     val tot = line.split("(\\(<)|,|(>,)|\\)")(3).toInt
7     res = res :+ (name, tot)
8     res
9   }).sortByKey(true).reduceByKey {
10    (x, y) => x + y
11  }                    //the first step
12  val sumBcast = sc.broadcast(sum.collectAsMap())
```

```
13
14      val nameFile = "people_name_list.txt"
15      val nameId = sc.textFile(nameFile).zipWithIndex()
16      val nameIdMap = nameId.collectAsMap()
17
18      val vertices = nameId.map(x => (x._2.toLong, x._1))
19      val edges = input.flatMap(line => {
20        var res = Seq[(String, (String, String))]()
21        val srcAttr = line.split("(\\(<)|,|(>,)|\\)")(1)
22        val dstAttr = line.split("(\\(<)|,|(>,)|\\)")(2)
23        val edgeAttr = line.split("(\\(<)|,|(>,)|\\)")(3)
24        res = res :+ (srcAttr, (dstAttr, edgeAttr))
25        res
26      }).sortByKey(true).map { edge =>
27        val neighbor = edge._2
28        Edge(nameIdMap(edge._1), nameIdMap(neighbor._1), neighbor._2.toFloat /
29          sumBcast.value(edge._1))
30      }                    //the second step
```

10.4 实验结果展示

```
(0,阿碧)
(1,阿尔尼)
(2,阿二)
(3,阿凡提)
(4,阿凡提妻)
(5,阿根)
(6,阿洪)
(7,阿济赤)
(8,阿九)
(9,阿里)
(10,阿曼)
(11,阿三)
(12,阿胜)
(13,阿朱)
(14,阿紫)
```

Figure 13: 人物关系图的顶点

```
Edge(1170,1067,0.0071770335)
Edge(1170,1354,0.028708134)
Edge(1170,749,0.011961723)
Edge(1170,1221,0.02631579)
Edge(1170,1146,0.078947365)
Edge(1170,1166,0.0071770335)
Edge(1170,961,0.004784689)
Edge(1170,1088,0.0023923444)
Edge(1170,399,0.033492822)
```

Figure 14: 人物关系图的边

11 Spark 实现基于人物关系图的 PageRank 计算

11.1 实验过程

算法从将 rankRDD 的每个元素的值初始化为 1.0 开始，然后在每次迭代中不断更新 rank 变量。在 Spark 中编写 PageRank 的主体相当简单：首先对当前的 rankRDD 计算出每个节点对目标节点的 pagerank 贡献值，并将这个贡献值加入目标 RDD 的节点中，再将原 RDD 图与更新节点的图 RDD 进行一次 joinVertices 操作，把该节点的 pagerank 值设置为 0.15 + 0.85 * contributionsReceived。

为了实现简单，也做了许多优化工作：

1. 将原来的人物关系构建成了图的形式，在 spark 中用图 RDD 的形式来刻画：顶点为人物，边权是人物共现的概率；

2. 在每次迭代开始前使用 rankGraph.cache 方法，将它保留在内存中以供每次迭代使用

3. 由于每条边是共现概率所以计算每个节点对目标节点的 pagerank 共现度的时候只需要乘以边权即可，避免了经典 pagerank 算法中求出度的麻烦；

4. 主要调用的函数：

函数名	参数	作用
aggregateMessages	(EdgeContext[VD, ED, A]) Unit, mergeMsg: (A, A) A TripletFields = TripletFields.All	聚合邻边以及相邻点的信息
outerJoinVertices	(other: RDD[(VertexId, U)]) (VertexId, VD, Option[U]) VD2)	将一个图中的节点并入另一张图中，顶点属性由 func 确定

11.2 代码展示

```
1   def PageRank[VD: ClassTag, ED:ClassTag](graph: Graph[VD,ED]):Graph[Double, Double] ={
2   var rankGraph: Graph[Double, Double] = graph
3     .mapTriplets( e => e.attr.toString.toFloat, TripletFields.Src )
4     .mapVertices { (id, attr) => 1.0      // Initialize the PageRank graph with 1.0
5   }
6
```

```
    var iteration = 0
    var prevRankGraph: Graph[Double, Double] = null
    while (iteration < numIter) {
      rankGraph.cache()

      val rankUpdates = rankGraph.aggregateMessages[Double](
        ctx => ctx.sendToDst(ctx.srcAttr * ctx.attr), _ + _, TripletFields.Src)
        // Compute the outgoing rank contributions of each vertex, perform local preaggregation, and
        // do the final aggregation at the receiving vertices.
      prevRankGraph = rankGraph
      rankGraph = rankGraph.outerJoinVertices(rankUpdates) {
        (id, oldRank, msgSumOpt) => 0.15 + 0.85 * msgSumOpt.getOrElse(0.0)
        // Apply the final rank updates to get the new ranks
      }.cache()
      rankGraph.edges.foreachPartition(x => {})
      iteration += 1
    }
    return  rankGraph
  }
```

11.3 实验结果展示

```
(赵敏,7.215189569192543)
(岳不群,7.339756021133582)
(石破天,7.446039828270495)
(吴三桂,7.858545405203923)
(陈家洛,9.341634623256047)
(汉子,13.099540211897303)
(胡斐,13.305463031280958)
(段誉,13.717216884358804)
(黄蓉,13.778246052518343)
(袁承志,13.82823365560552)
(杨过,14.133022859754448)
(郭靖,16.084685745047857)
(张无忌,19.878320439519772)
(令狐冲,20.439427484104787)
(韦小宝,34.49267615888099)
```

Figure 15: 人物 pagerank 值

可见通过 Spark 得到的各个人物的 pagerank 值与通过 hadoop 得到的各个人物的 pagerank 值是一样的，从另一个方面印证了两种方法的等价性；但是通过实验，可以明显感受到 spark 的运行速度是 hadoop 无法比拟的；

12 Spark 实现标签传播

Spark 框架中的 GraphX 组件可以用于图和并行图的计算，通过引入 Resilient Distributed Property Graph 来扩展 Spark RDD。为了支持图计算，GraphX 提供了一组基本的功能操作 API。

12.1 Pregel API 简介

Spark 框架下的标签传播算法可以借助 GraphX 的 Pregel API 实现。Pregel API 是一个约束到图拓扑的**批量同步（bulk-synchronous）**并行消息抽象。它执行一系列超级步骤（super steps），在这些步骤中，顶点从之前的超级步骤中接收进入消息（inbound message）的总和，为顶点属性计算一个新的值，然后在以后的超级步骤中发送消息到邻居节点。批量同步并行消息抽象如图所示：

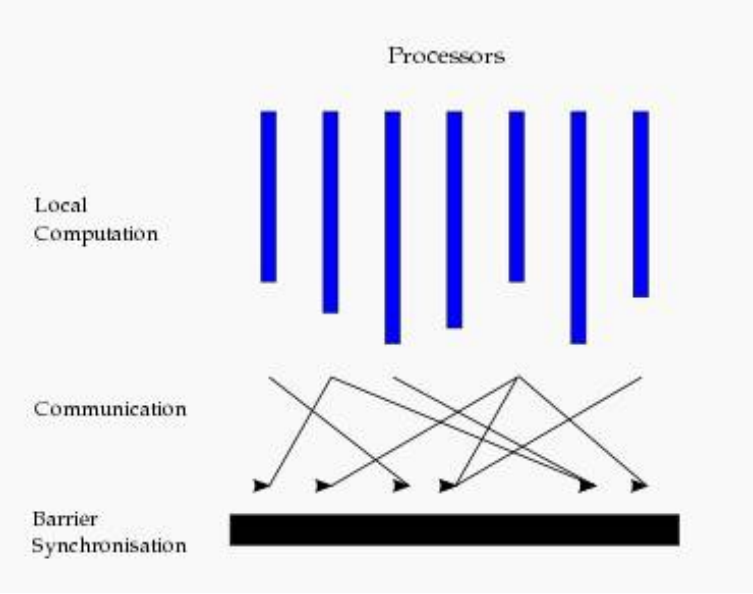

Figure 16: 批量同步并行消息抽象

12.2 标签传播的 Bulk Synchronous 算法

同步更新的标签传播算法可以完美地嵌入 Pregel 的批量同步并行消息抽象中。算法的过程如下：

1) 输入图数据，每个节点的标签初始化为其自身，所有节点置为活跃状态；

2) 每个节点向其所有邻居节点发送自己的标签信息 (label, 1)；

3) 每个节点对收到的所有消息进行合并，并对键值相同即标签相同的值进行累加，这样就可以统计出邻居标签的数目；

4) 每个节点将自己的标签更新为数量最多的标签；

5) 重复步骤 2)，直至达到最大迭代次数或所有节点的状态均为不活跃。

同步更新的标签传播 Bulk Synchronous 算法的伪代码如下：

算法 15 同步更新的标签传播 Bulk Synchronous 算法

```
1: function sendMsg(edge)
2:     send(edge.src, (edge.dst.label, 1))
3:     send(edge.dst, (edge.src.label, 1))
4: end function
5: function mergeMsg(labelCount1 : [(label, cnt)], labelCount2 : [(label, count)])
6:     msg ← ∅
7:     for label in labelCount1.keySet + labelCount2.keySet do
8:         count1 = labelCount1.get(label)
9:         count2 = labelCount2.get(label)
10:        msg+ = (label, count1 + count2)
11:    end for
12:    return msg
13: end function
14: function update(vertex, msg : [(label, cnt)])
15:    vertex.label = msg.maxByValue().key()
16: end function
```

12.3 性能优化

在发送消息阶段，对于图上的每条边而言，都发送了两次消息：一次是将源点的标签发送至终点，一次是将终点的标签发送至源点。回想任务三中得到的邻接表，事实上两个节点之间或不存在边，或存在两条权值不同、方向相反的有向边。这样一来，每个节点会收到来自同一个邻居的两份相同的消息。虽然这不影响算法的运行结果，但无疑使得通信量翻倍，从而影响算法的性能。因此在发送消息函数中，只将源点的标签发送至终点或只将终点的标签发送至源点即可。

12.4 结果展示

同步更新的标签传播算法的结果如图所示：

13 Spark 实现全局排序

利用 Spark 实现全局排序的整体思路与 Hadoop 大体一致：将人名与 PageRank 合成键值对后，在 Reduce 中对 PageRank 值进行排序。同样的，Spark 的高级 API 让我们的运算更加简便。

首先通过 SparkContext 读入 textFile，然后通过 map 操作将每一行数据切分成 $(name, pr)$ 键值对，从而生成 $(String, Float)$ 类型的 RDD 中。

而十分幸运的是，对于排序，Spark 提供了十分简便的 API，包括 sortBykey 对关键字排序，此外还有指定排序对象的 API 即 sortBy，所以此处可以不用调换键值对的 key 和 value，可以直接指定键值对中 pr 值作为排序对象。得到结果最后调用用 saveAsTextFile 存储。

查继左 84
察尔珠 85
陈珂 105
陈老丐 106
钟兆文 119
褚轰 119
刘鹤真 119
蓝秦 119
王师兄 119
汪铁鹗 119
蛇皮张 119
胖商人 119
南兰 119

Figure 17: 任务五结果展示

在考虑以上算法后，我们编写 Scala 代码如下。

```scala
//Scala code
    val conf = new SparkConf()
    conf.setMaster("local").setAppName("sort")
    val sc = new SparkContext(conf)

    //读取数据（人物，pagerank值）
    val numbers: RDD[(String, Float)] = sc.textFile("out4").map { line =>
      val name = line.split("\t")(0)
      val pr = line.split("\t")(1).toFloat
      (name, pr)
    }
    //将rdd中的数据转为key-value的形式，使用sortBy进行排序
    val result = numbers.sortBy(s => s._2)
    result.saveAsTextFile("out6_1")
    sc.stop()
```

14 Spark 实现标签聚集

有了使用 Spark 实现全局排序的经验后，那么实现标签聚集十分简单。将人名与标签值合成键值对后，在 Reduce 中直接将相同标签值聚集在一起。而对此 Spark 有提供相应的 API。

首先通过 SparkContext 读入 textFile，然后通过 map 操作将每一行数据切分成 $(name, label)$ 键值对，从而生成 $(String, Int)$ 类型的 RDD 中。

而我们需要将相同 label 的 name 聚集在一起，要用到 Spark 提供的 groupByKey 函数，可以实现相同 key 的聚集。

但是由于最初的 key 为 name，所以我们需要通过 map 操作将原有 $(String, Int)$ 类型的 RDD 的 key 和 value 对换得到新的 $(Int, String)$ 类型的 RDD，随后对其调用 groupByKey，最终得到 $(Int, Iterable[String])$ 类型的 RDD 即为最终结果。

最后调用用 saveAsTextFile 存储。

在考虑以上算法后，我们编写 Scala 代码如下。

```
//Scala code
    val conf = new SparkConf()
    conf.setMaster("local").setAppName("cluster")
    val sc = new SparkContext(conf)

    //读取数据（人物，标签值）
    val numbers:RDD[(String, Int)] = sc.textFile("out5").map { line =>
    val name = line.split("\t")(0)
    val label = line.split("\t")(1).toInt
    (name, label)
    }
    val result:RDD[(Int, String)]  = numbers.map(x=>(x._2, x._1))
    val all:RDD[(Int, Iterable[String])] = result.groupByKey()
    all.saveAsTextFile("out6_2")
    sc.stop()
```