

# Woogle

本项目利用 Hadoop MapReduce，构建了对 Wikipedia 语料库的倒排索引，并实现了一个简易的搜索引擎，可根据检索的关键词返回相应的索引信息，使用 Java 编写。

## 目录

- Woogle
  - 目录
  - 项目报告
    - 1. 任务说明与描述
    - 2. 参与人员任务分工说明
    - 3. 程序启动与操作说明
      - 3.1 开发
      - 3.2 启动
      - 3.3 执行结果
        - 3.3.1 索引格式
        - 3.3.2 搜索结果格式
    - 4. 程序文件 / 类功能说明
    - 5. 架构以及模块实现方法说明
      - 5.1 总览
      - 5.2 Driver
      - 5.3 Job 1 - token position
        - 5.3.1 Driver
        - 5.3.2 Mapper
        - 5.3.3 Reducer
      - 5.4 Job 2 - token count
        - 5.4.1 Driver
        - 5.4.2 Mapper
        - 5.4.3 Reducer
      - 5.5 Job 3 - inverted index
        - 5.5.1 Driver
        - 5.5.2 Mapper
        - 5.5.3 Reducer
      - 5.6 Woogle
        - 5.6.1 `searchAndPrint()`
        - 5.6.2 `getPartition()`
        - 5.6.3 `InverseDocumentFreq.parse()`
        - 5.6.4 `printInverseDocumentFreq()`
        - 5.6.5 `TermFreq.parse()`
        - 5.6.6 `printInvertedIndex()`
- 许可协议

# 项目报告

## 1. 任务说明与描述

### Hadoop 平台使用及 PJ 要求

在服务器上的 `/corpus/wiki` 目录下有 `0, 1, ..., 63.txt` 等 64 个文本文件，每个文件大小约为 300 MB，其内容格式为分行、无标点的英文文本，示例如下：

```
1 | lorem ipsum dolor sit amet consectetur adipisicing elit sed do eiusmod tempor incididunt ut labore et dolore m  
2 | ulla  
3 | eu fugiat nulla pariatur Excepteur sint occaecat cupidatat non proident sunt in culpa qui officia deserunt mol  
4 | ...
```

使用这些语料数据，计算文档中每个词的 TF-IDF（每个文件视为一个文档），要求实现以下功能：

1. 为每个出现的词构建索引，包括所属文档、出现次数、TF、IDF 信息；
2. 在上一步的基础上，包括此词在文档中出现的位置；
3. 支持关键词检索。实现程序，输入词后，程序输出这个词的索引。

## 2. 参与人员任务分工说明

- **陈泓宜 (18307130003)** - 复旦大学：独立完成全部功能，实现了对语料库倒排索引的构建，实现了基于索引的关键词搜索功能。

## 3. 程序启动与操作说明

### 3.1 开发

项目使用 IntelliJ IDEA 开发，相关构建、运行、打包配置已经写在了 `.idea` 目录下的配置文件里，直接在 IDE 里执行相应的任务即可：

- `index`：构建 package `xyz.hakula.index`，并执行其主类 `xyz.hakula.index.Driver` 的 `main()` 函数，传入参数 `input`, `output`, `temp`。这个包的功能是构建目录 `input` 下所有文件的倒排索引，索引结果保存在目录 `output` 下，执行过程中产生的临时数据放置在目录 `temp` 下。
- `woogle`：构建 package `xyz.hakula.woogle`，并执行其主类 `xyz.hakula.woogle.Woogle` 的 `main()` 函数，传入参数 `output`。这个包的功能是根据用户输入的关键词，在目录 `output` 下的索引里进行检索，最后输出这个关键词的倒排索引。
- `index_jar`：将上述 package `xyz.hakula.index` 打包为 JAR 包 `index.jar`，保存在目录 `jar` 下，之后同任务 `index` 一样执行
- `woogle_jar`：将上述 package `xyz.hakula.woogle` 打包为 JAR 包 `woogle.jar`，保存在目录 `jar` 下，之后同任务 `woogle` 一样执行

当然也可以通过 IDE 的构建选项只构建不执行，这里不作赘述。

原项目基于 Java SE 17 开发，为了兼容服务器的 Java SE 8 环境，在主分支 `master` 外维护了一个 `dev-jdk-1.8` 分支，提供了基于 Java 8 版本的实现，同时提供了配套的 IDE 配置文件。

## 3.2 启动

项目已经预先打包好了 `index.jar` 和 `woogle.jar` 文件，可以直接使用。

对于 `index.jar` 文件，如果希望在本机上使用，则执行以下命令（需提前配置好 Java 环境）：

```
java -jar index.jar <input_path> <output_path> <temp_path>
```

其中，`<input_path>`，`<output_path>`，`<temp_path>` 分别表示指定的输入路径（语料库位置）、输出路径（索引位置）和缓存路径（临时文件位置）。需要注意的是，如果 `<output_path>`、`<temp_path>/output_job1` 和 `<temp_path>/output_job2` 中的某些在程序运行前已经存在，则程序会跳过部分任务的执行（具体跳过了什么、为什么跳过将在之后展开阐述）。因此如果你需要重新执行全部任务，则需要显式地将这些文件夹手动删除。

如果希望在 Hadoop 集群上使用，则执行以下命令（需提前配置好 Hadoop 环境）：

```
hadoop jar index.jar <input_path> <output_path> <temp_path>
```

对于 `woogle.jar` 文件，类似地执行以下命令：

```
java -jar woogle.jar <index_path>
```

其中，`<index_path>` 表示指定的索引位置，通常也就是前面传入 `index.jar` 的 `<output_path>`。

## 3.3 执行结果

### 3.3.1 索引格式

执行 `index.jar` 后，我们将在输出路径 `<output_path>` 下得到我们的索引文件。其中 TF 保存在根目录下，IDF 保存在子目录 `inverse_document_freq` 下，之后会解释为什么选择分开保存。

TF 的格式如下（文件名：`part-r-00xxx`）：

```
1 | <token> <filename>:<token_count>:<tf>:<position_1>;...;<position_n>
```

IDF 的格式如下（文件名：`<token>`）：

```
1 | <file_count> <idf>
```

其中：

- `<token>`：表示一个短语  $t$
- `<filename>`：表示出现这个短语  $t$  的文档  $d$  的文件名
- `<token_count>`：表示这个短语  $t$  在文档  $d$  中出现的次数  $c_{t,d}$
- `<tf>`：表示这个短语  $t$  在文档  $d$  中的词频 TF (Term Frequency)，使用科学计数法表示，这里我们采用的算法是

$$\text{TF}(t, d) = \frac{c_{t,d}}{C_d}$$

其中  $C_d$  表示文档  $d$  中的短语总数

- `<position_i>`：表示这个短语  $t$  在文档  $d$  中出现的位置  $p_{t,d,i}$ ，这里我们取的是该短语首字符关于文档起始位置的字节偏移量
- `file_count`：表示出现这个短语  $t$  的文档总数  $n_t$
- `<idf>`：表示这个短语  $t$  在所有文档  $D$  中的逆向文件频率 IDF (Inverse Document Frequency)，这里我们采用的算法是

$$\text{IDF}(t) = \log_2 \frac{N}{n_t}$$

其中  $N$  表示语料库中文档的总数  $|D|$

- `<token>` 和其余部分之间以 `Tab` 分隔
- `<filename>`, `<token_count>`, `<tf>`, `<position>` 之间以 `:` 分隔
- `<position>` 之间以 `;` 分隔

以一个小样例的索引结果为例：

TF (文件名: `part-r-00121`) :

```
1 | lease    04.txt:1:4.710316e-04:9845
2 | mobilise  02.txt:1:9.689922e-04:5182
3 | past     01.txt:1:1.937984e-03:472
4 | their    01.txt:4:7.751938e-03:285;611;1616;2684
5 | their    03.txt:2:2.844950e-03:2867;3033
6 | their    04.txt:6:2.826189e-03:1109;6786;7437;10379;10416;12914
7 | their    02.txt:3:2.906977e-03:5191;5326;5523
```

IDF (文件名: `lease`) :

```
1 | 1 2.0
```

注意到相同 `<token>` 在不同文件里的索引也被分成了不同的行，之后也会解释原因。

索引过程中产生的日志文件会保存在 `logs/app.log` 文件里（文件名随日期滚动）。

### 3.3.2 搜索结果格式

执行 `woogle.jar` 后，程序会提示用户输入一个关键词：

```
Please input a keyword:  
>
```

输入关键词并回车后，程序将输出这个关键词的搜索结果，其格式如下：

```
<token>: IDF = <idf> | found in <file_count> files:  
<filename>: TF = <tf> (<token_count> times) | TF-IDF = <tfidf> | positions: <position> <position> ...  
...
```

其中：

- <tfidf>：表示这个短语  $t$  在文档  $d$  中的 TF-IDF，使用科学计数法表示，这里我们采用的算法是

$$\text{TFIDF}(t, d) = \text{TF}(t, d) \cdot \text{IDF}(t)$$

通常，这个值可以作为这个文档在搜索结果中的权重。

例如：

```
> back
back: IDF = 1.000000 | found in 2 files:
02.txt: TF = 9.689922e-04 (1 times) | TF-IDF = 9.689922e-04 | positions: 3836
03.txt: TF = 2.844950e-03 (2 times) | TF-IDF = 2.844950e-03 | positions: 518 1398
```

如果没有找到，程序则会输出：

```
> aaaa
aaaa: not found
```

## 4. 程序文件 / 类功能说明

这里重点讲项目的核心代码部分，一些诸如 `log4j.properties` 之类的配置文件就略过了。

- `src/main/java/`：项目源代码
  - `xyz/hakula/index/`：package `xyz.hakula.index`，倒排索引构建功能的实现
    - `io/`：一些自定义 `Writable` 类型的定义，令 `MapReduce` 的 `key` 和 `value` 可以使用自定义类型。在使接口和实现更清晰可读、易于维护的同时，也节省了每次 `join` 成 `String` 再 `split` 回来的性能开销。因为比较 trivial，这里就不细讲了，可以直接看源代码，写得很清楚。
    - `Driver.java`：索引程序的主类，配置了所有的 `Job`，然后依次执行
    - `TokenPosition.java`：第 1 个 `Job`，读取目录 `<input_path>` 里的文件，提取所有短语在各文件中出现的位置，保存在路径 `<temp_path>/output_job1` 下
    - `TokenCount.java`：第 2 个 `Job`，读取目录 `<temp_path>/output_job1` 里的文件，统计所有短语在各文件中出现的次数，保存在路径 `<temp_path>/output_job2` 下；同时统计各文件的短语总数，保存在路径 `<temp_path>/file_token_count` 下对应的文件名（`<filename>`）里
    - `InvertedIndex.java`：第 3 个 `Job`，从路径 `<temp_path>/file_token_count` 里按需读取各文件的短语总数到内存中；然后读取目录 `<temp_path>/output_job2` 里的文件，计算所有短语在各文件中的 `TF`，保存在路径 `<output_path>` 下；同时计算所有短语的 `IDF`，保存在路径 `<output_path>/inverse_document_freq` 下对应的文件名（`<token>`）里
  - `xyz/hakula/woogle/`：package `xyz.hakula.woogle`，倒排索引检索功能的实现
    - `model/`：一些自定义类型的定义，类似于 package `xyz.hakula.index` 下 `io/` 里的类，此外也提供了一些格式化输出索引的方法
    - `Woogle.java`：检索程序的主类，从终端读取用户输入，定位到对应的索引文件进行查询，然后利用 `model/` 里提供的方法格式化输出到终端

## 5. 架构以及模块实现方法说明

### 5.1 总览

项目的整体架构分为 3 个 `MapReduce Job`。一般来说，关注程序的输入和输出是一个理清脉络的好方法。

开始时，输入数据的格式如下：

```
1 | <token> <token>
```

这里每个 `<token>` 就代表了一个短语。

首先这些数据经过 Job 1 - token position 的 Mapper，输出所有短语在各文件中出现的位置，格式如下：

```
1 | <token>@<filename> <position>
```

其中，`Tab` 的左右侧分别是 key 和 value。这里每行的 `<position>` 只有 1 个。

然后这些数据经过 Job 1 的 Reducer，将同文件下相同短语（也就是 key 相同）的位置聚合了起来，输出所有短语在各文件中出现的位置数组，格式如下：

```
1 | <token>@<filename> <position>;<position>;<position>
```

或者表示成：

```
1 | <token>@<filename> [<position>]
```

至此 Job 1 结束，所有结果保存在目录 `<temp_path>/output_job1` 下的文件里。为了节省 Job 间原始数据和 `String` 之间互相转换的开销，这里我们直接顺序输出二进制格式的数据（`SequenceFileOutputFormat`），因此直接打开文件是无法阅读的。

接下来这些数据经过 Job 2 - token count 的 Mapper，将 key 里的 `<token>` 字段移到 value 里，以便后续可以对 `<filename>` 聚合处理，格式如下：

```
1 | <filename> <token>:[<position>]
```

然后这些数据经过 Job 2 的 Reducer，对于每个文件，统计各短语在其中出现的次数，并交换 `<token>` 和 `<filename>` 字段的位置，为 Job 3 做准备，格式如下：

```
1 | <token> <filename>:<token_count>:@:[<position>]
```

这里这个 `@` 是 TF 的占位符，目前还无法计算（之后会讲为什么），因此先留空。

与此同时，我们对文件里所有短语的出现次数求和，从而得到文件的短语总数，格式如下：

```
1 | <total_count>
```

至此 Job 2 结束，所有结果保存在目录 `<temp_path>/output_job2` 下的文件里，每个文件的短语总数保存在文件 `<temp_path>/file_token_count/<filename>` 里。

接下来这些数据经过 Job 3 - inverted index 的 Mapper，根据文件 `<temp_path>/file_token_count/<filename>` 里保存的文件短语总数，计算得到 TF，替换掉原来的占位符，格式如下：

```
1 | <token> <filename>:<token_count>:<tf>:[<position>]
```

最后这些数据经过 Job 3 的 Reducer，对于每个短语，聚合在这里的 TF 条目总数就是出现了这个短语的文件总数。然后根据预先在外侧（`xyz.hakula.index.Driver`）统计的文件总数，计算得到每个短语的 IDF。

TF 直接原样输出，格式如下：

```
1 | <token> <filename>:<token_count>:<tf>:[<position>]
```

IDF 的格式如下：

```
1 | <file_count> <idf>
```

至此 Job 3 结束，索引结果的 TF 部分保存在目录 `<output_path>` 下的文件里，IDF 部分保存在文件 `<output_path>/inverse_document_freq/<token>` 里。

下面我们来看看具体的实现。

## 5.2 Driver

首先是索引程序的主类 `Driver`，也就是整个程序的入口。以下是基于 Java SE 17 的实现：

```

1 // src/main/java/xyz/hakula/index/Driver.java
2
3 public class Driver extends Configured implements Tool {
4     public static final int NUM_REDUCE_TASKS = 128;
5
6     public static void main(String[] args) throws Exception {
7         var conf = new Configuration();
8         System.exit(ToolRunner.run(conf, new Driver(), args));
9     }
10
11    public int run(String[] args) throws Exception {
12        var inputPath = new Path(args[0]);
13        var outputPath = new Path(args[1]);
14        var tempPath = new Path(args[2]);
15        var tempPath1 = new Path(tempPath, "output_job1");
16        var tempPath2 = new Path(tempPath, "output_job2");
17        var fileTokenCountPath = new Path(tempPath, "file_token_count");
18
19        var conf = getConf();
20        try (var fs = FileSystem.get(conf)) {
21            var totalFileCount = fs.getContentSummary(inputPath).getFileCount();
22            if (totalFileCount == 0) return 0;
23            conf.setLong("totalFileCount", totalFileCount);
24            conf.set("fileTokenCountPath", fileTokenCountPath.toString());
25
26            if (!fs.exists(tempPath1) && !runJob1(inputPath, tempPath1)) System.exit(1);
27            if (!fs.exists(tempPath2) && !runJob2(tempPath1, tempPath2)) System.exit(1);
28            if (!fs.exists(outputPath) && !runJob3(tempPath2, outputPath)) System.exit(1);
29        }
30        return 0;
31    }
32}

```

先看主体部分，开始时先从命令行参数里读取 `<input_path>` , `<output_path>` , `<temp_path>` , 然后确定几个 Job 的输出路径 `<temp_path_1>` , `<temp_path_2>` , `<file_token_count_path>` 。

接下来先利用 `fs.getContentSummary(inputPath).getFileCount()` 直接得到输入路径里的文件总数，为之后计算 IDF 做准备。为什么这样实现呢？因为这样比较简单，通过 MapReduce 会比较麻烦。

然后将这个文件总数 `totalFileCount` 和未来保存各文件短语总数的路径 `fileTokenCountPath` 写入配置 `conf` ，以供接下来的 MapReduce Job 使用。

最后就是依次执行 3 个 Job 了，如果失败就退出。这里对路径是否存在做了一个判断，目的有两个：首先，如果输出路径已经存在的话，Hadoop 会抛异常。这是因为 Hadoop 在设计上是希望使用者一次写入、多次读取的，因此如果需要重新写入的话，需要显式地手动删除这个文件夹。其次，如果每次重启都直接删除所有文件夹的话，有点浪费，因为很多时候我们可能只希望重做其中一两个 Job（比如错误恢复的情形）。对于大数据集来说，保留一部分中间结果，重做时可以节省大量的时间。

接下来讲讲这 3 个 MapReduce Job。

## 5.3 Job 1 - token position

### 5.3.1 Driver

在 `Driver` 里，我们需要先对这个 Job 进行一些设定。

```
1 // src/main/java/xyz/hakula/index/Driver.java
2
3 public class Driver extends Configured implements Tool {
4     private boolean runJob1(Path inputPath, Path outputPath)
5         throws IOException, InterruptedException, ClassNotFoundException {
6         var job1 = Job.getInstance(getConf(), "token position");
7         job1.setJarByClass(TokenPosition.class);
8
9         job1.setMapperClass(TokenPosition.Map.class);
10        job1.setMapOutputKeyClass(TokenFromFileWritable.class);
11        job1.setMapOutputValueClass(LongWritable.class);
12
13        job1.setReducerClass(TokenPosition.Reduce.class);
14        job1.setNumReduceTasks(NUM_REDUCE_TASKS);
15        job1.setOutputKeyClass(TokenFromFileWritable.class);
16        job1.setOutputValueClass(LongArrayWritable.class);
17        job1.setOutputFormatClass(SequenceFileOutputFormat.class);
18
19        FileInputFormat.addInputPath(job1, inputPath);
20        FileOutputFormat.setOutputPath(job1, outputPath);
21
22        return job1.waitForCompletion(true);
23    }
24 }
```

主要做的事情是：

- 设定 Mapper 的类为 `TokenPosition.Map`，输出的 key 和 value 的类型分别为 `TokenFromFileWritable` 和 `LongWritable`
- 设定 Reducer 的类为 `TokenPosition.Reduce`，任务数量为 `128`，输出的 key 和 value 的类型分别为 `TokenFromFileWritable` 和 `LongArrayWritable`，输出到文件的格式为顺序输出二进制文件
- 设定输入路径为 `inputPath`，输出路径为 `outputPath`

最后等待 Job 1 完成。

那 Job 1 的核心类可想而知，就是 `TokenPosition` 了。我们来看看相关的实现。

### 5.3.2 Mapper

```

1 // src/main/java/xyz/hakula/index/TokenPosition.java
2
3 public class TokenPosition {
4     public static class Map extends Mapper<LongWritable, Text, TokenFromFileWritable, LongWritable> {
5         private final TokenFromFileWritable key = new TokenFromFileWritable();
6         private final LongWritable offset = new LongWritable();
7
8         // Yield the byte offset of a token in each file.
9         @Override
10        public void map(LongWritable key, Text value, Context context)
11            throws IOException, InterruptedException {
12            var filename = ((FileSplit) context.getInputSplit()).getPath().getName();
13            var offset = key.get(); // byte offset
14
15            var it = new StringTokenizer(value.toString(), " \t\r\f");
16            while (it.hasMoreTokens()) {
17                var token = it.nextToken().toLowerCase(Locale.ROOT);
18                this.key.set(token, filename);
19                this.offset.set(offset);
20                context.write(this.key, this.offset);
21
22                // Suppose all words are separated with a single whitespace character.
23                offset += token.getBytes(StandardCharsets.UTF_8).length + 1;
24            }
25        }
26    }
27 }

```

首先我们需要知道，最开始第一个 Mapper 直接从原始文件读取时，是按行读取的。也就是说，每个输入的 value 是源文件的一个行，而不是整个文件的内容。但接下来悲剧来了，key 是什么？很多教程里这个 key 的类型写的是 object，因为他们并没有用到这个 key，实际上 key 的类型应该是 LongWritable。有些教程说 key 的含义是 value 在文件中的行号，这也是错的，实际上 key 代表的是 value 关于文档起始位置的**字节偏移量**，而且不是**列偏移量**。

为什么说这是个悲剧呢？因为这意味着想得到当前 value 实际在文件中的行号将变得异常困难，这是 Hadoop 的第一个坑。经过几天的研究，我目前了解到的有以下方案 [1]：

1. 写个脚本程序对输入文件进行预处理，给每行的开头加一个行号
2. 重新实现一个 InputFormat，在最开始读入文件时，将 key 设定成行号

这两个方案我都不太满意，因此在进行了诸多尝试之后，我最后决定不做这个事了。因此目前用来表示短语在文档中位置的 <position> 的值，就是这个短语首字符关于文档起始位置的**字节偏移量**。其实这样在检索时也方便快速定位。

那么如何得到每个短语的字节偏移量 <position> 呢？考虑到语料库里所有短语都是用单个空格分隔的，那就好办多了。我们直接利用 StringTokenizer 将这行的内容分隔成一个个短语，然后每个短语的 <position> 就是前一个短语的 <position> 加上其所占字节数加 1，第一个短语的 <position> 就是行首关于文档起始位置的字节偏移量，也就是 key 的值。

最后我们设置输出的 key 为 <token>@<filename>，以便 Reducer 进一步聚合每个短语在各文件里出现的所有位置。输出的 value 就是短语出现的位置，也就是前面讲的字节偏移量。

### 5.3.3 Reducer

```
1 // src/main/java/xyz/hakula/index/TokenPosition.java
2
3 public class TokenPosition {
4     public static class Reduce extends
5         Reducer<TokenFromFileWritable, LongWritable, TokenFromFileWritable, LongArrayWritable> {
6         private final LongArrayWritable offsets = new LongArrayWritable();
7
8         // Yield all occurrences of a token in each file.
9         @Override
10        public void reduce(TokenFromFileWritable key, Iterable<LongWritable> values, Context context)
11            throws IOException, InterruptedException {
12            var offsets = new ArrayList<LongWritable>();
13            for (var value : values) {
14                offsets.add(WritableUtils.clone(value, context.getConfiguration()));
15            }
16            offsets.sort(LongWritable::compareTo);
17            this.offsets.set(offsets.toArray(LongWritable[]::new));
18            context.write(key, this.offsets);
19        }
20    }
21 }
```

Reducer 的逻辑就比较简单了，就是将每个短语在各文件里出现的所有位置排个序，然后聚合成一个数组，最后输出。这里使用 `SequenceFileOutputFormat` 的好处就体现出来了，我们可以直接输出自定义 Writable 类型，而不用一定要转化成 Text。

需要注意的是，MapReduce 在遍历一个 Iterable 时，为了节省内存开销，会**复用同一个 value 对象**，这是 Hadoop 的第二个坑。那我们知道 Java 底层全都是传的 reference，所以如果你直接将 value 传入数组的话，最后数组里所有元素的值就都会是同一个值（也就是最后一个元素）。因此这里传入数组的时候，一定要使用 `WritableUtils.clone()` 方法复制一个对象。

## 5.4 Job 2 - token count

### 5.4.1 Driver

```

1 // src/main/java/xyz/hakula/index/Driver.java
2
3 public class Driver extends Configured implements Tool {
4     private boolean runJob2(Path inputPath, Path outputPath)
5         throws IOException, InterruptedException, ClassNotFoundException {
6         var conf = getConf();
7         var job2 = Job.getInstance(conf, "token count");
8         job2.setJarByClass(TokenCount.class);
9
10        job2.setInputFormatClass(SequenceFileInputFormat.class);
11        job2.setMapperClass(TokenCount.Map.class);
12        job2.setMapOutputKeyClass(Text.class);
13        job2.setMapOutputValueClass(TokenPositionsWritable.class);
14
15        var totalFileCount = conf.getLong("totalFileCount", 1);
16        job2.setReducerClass(TokenCount.Reduce.class);
17        job2.setNumReduceTasks((int) totalFileCount);
18        job2.setOutputKeyClass(Text.class);
19        job2.setOutputValueClass(TermFreqWritable.class);
20        job2.setOutputFormatClass(SequenceFileOutputFormat.class);
21
22        FileInputFormat.addInputPath(job2, inputPath);
23        FileOutputFormat.setOutputPath(job2, outputPath);
24
25        return job2.waitForCompletion(true);
26    }
27}

```

和 Job 1 基本没什么区别，不再赘述。

这里将 Reducer 的任务数量设置为了文件总数，是因为这一步是在对 `<filename>` 进行聚合。这里最好是设置一个 Partitioner，让每个 `<filename>` 可以和 Reducer 一一对应，不过因为对效率影响不大，这里就不写了。

#### 5.4.2 Mapper

```

1 // src/main/java/xyz/hakula/index/TokenCount.java
2
3 public class TokenCount {
4     public static class Map
5         extends Mapper<TokenFromFileWritable, LongArrayWritable, Text, TokenPositionsWritable> {
6             private final Text key = new Text();
7             private final TokenPositionsWritable value = new TokenPositionsWritable();
8
9             // (<token>@<filename>, [<offset>]) -> (<filename>, (<token>, [<offset>]))
10            @Override
11            public void map(TokenFromFileWritable key, LongArrayWritable value, Context context)
12                throws IOException, InterruptedException {
13                this.key.set(key.getFilename());
14                this.value.set(key.getToken(), (Writable[]) value.toArray());
15                context.write(this.key, this.value);
16            }
17        }
18    }

```

Job 2 的 Mapper 就是字段改个位置，这个前面讲过了。接下来 Reducer 就可以对 `<filename>` 进行聚合。

### 5.4.3 Reducer

```
1 // src/main/java/xyz/hakula/index/TokenCount.java
2
3 public class TokenCount {
4     public static class Reduce extends Reducer<Text, TokenPositionsWritable, Text, TermFreqWritable> {
5         private final Text key = new Text();
6         private final TermFreqWritable value = new TermFreqWritable();
7
8         // Yield the token count of each token in each file,
9         // and calculate the total token count of each file.
10        // (<filename>, (<token>, [<offset>]))
11        // → (<token>, (<filename>, <token_count>, 0, [<positions>]))
12        @Override
13        public void reduce(Text key, Iterable<TokenPositionsWritable> values, Context context)
14            throws IOException, InterruptedException {
15            var filename = key.toString();
16            long totalTokenCount = 0;
17            for (var value : values) {
18                var positions = value.getPositions();
19                var tokenCount = positions.length;
20                this.key.set(value.getToken());
21                // The Term Frequency (TF) will be calculated in next job, and hence left blank here.
22                this.value.set(filename, tokenCount, 0, positions);
23                context.write(this.key, this.value);
24                totalTokenCount += tokenCount;
25            }
26            writeToFile(context, key.toString(), totalTokenCount);
27        }
28
29        private void writeToFile(Context context, String key, long totalTokenCount) throws IOException {
30            var conf = context.getConfiguration();
31            var fs = FileSystem.get(conf);
32            var fileTokenCountPath = conf.get("fileTokenCountPath");
33            var outputPath = new Path(fileTokenCountPath, key);
34            try (var writer = new BufferedWriter(new OutputStreamWriter(fs.create(outputPath, true)))) {
35                writer.write(totalTokenCount + "\n");
36            }
37        }
38    }
39 }
```

Reducer 比较复杂，是整个项目最大的难点。困难的不是实现本身，而是选择这个解决方案的思考过程。

我们的目标是得到 TF，现在我们有所有短语在各文件里出现的所有位置，因此我们就有所有短语在各文件里的出现次数。我们知道

$$\text{TF}(t, d) = \frac{c_{t,d}}{\sum_{t' \in d} c_{t',d}}$$

所以我们现在只需要各文件的短语总数，也就是将每个短语的出现次数求和。

听起来是不是很简单？我们已经有所有短语的出现次数了，直接遍历一下加起来不就行了？然而问题来了，现在我们不仅需要累加，而且还需要计算每个短语在各文件里的 TF。但问题是得先遍历一次，得到文件的短语总数，然后才能算出短语在这个文件里的 TF。

这有什么难的，那就先遍历一次求和，然后再遍历一次分别计算出 TF 的值不就好了？麻烦来了，MapReduce 为了节省内存开销，Iterable 只能遍历一次，阅后即焚，这是 Hadoop 的第三个坑。

怎么解决呢？很简单，我直接开个数组把这些 value 都存下来不就好了。悲剧来了，`java.lang.OutOfMemoryError`！数据集太大，爆内存了。

看来遍历的时候不能将数据保存在内存里，必须直接写入文件。这下没办法在 Job 2 直接得到 TF 了，只能先用 0 占个位，我们到 Job 3 再算。问题又来了，那这些文件短语总数存在哪里呢？

一个很直观的想法是，那我在内存里建一个 HashMap，执行 Job 2 的过程中保存在里面，执行 Job 3 时再读取不就好了。至于这个 HashMap 放哪里，无所谓，反正基本约等于全局变量（准确来说是 static 变量），放 Driver 类里和放 InvertedIndex 类里都一样。

我一开始也是这么实现的，而且在本地跑得很正常，一点问题都没有。结果一上 Hadoop 集群傻眼了，`java.lang.NullPointerException`！写进 HashMap 的键值对，Job 3 读不到。

怎么一回事呢？原来，Hadoop 集群上每个任务都单开了一个 JVM [2]，对于其他语言的实现就是单开了一个进程，这是 Hadoop 的第四个坑。所以你在这个 JVM 里写进内存的数据，其他 JVM 当然读不到了。其实仔细想想，显然是这么个道理，毕竟分布式系统，怎么可能所有任务都跑在同一个进程上。但第一次接触分布式系统的话，难免容易用单机的思路想问题，然后就上当了。

那怎么办呢？一种思路，也就是我目前的实现，是在执行 Job 2 的过程中，直接将文件短语总数写进文件里，之后执行 Job 3 前再读取到内存中。需要注意的是，不可以先写进内存，最后统一写进文件里，因为这同样会遇到前面提到的 JVM 分离的问题，不同任务的内存都是分开的。此外，为了避免并发写的问题，这里我将不同文件的短语总数都写到了不同的文件里（以 `<filename>` 命名）。这里还需要注意的是，学校服务器的 HDFS 似乎不支持 append 写，因此你也做不到把他们都写进一个文件里。

这个问题的解决方案想了我至少两天，可以说是本项目最大的难点了，期间真的是踩了不少坑。

别的就没什么好说的了，代码很直观。这里我们输出的 key 又变回了 `<token>`，接下来我们将对 `<token>` 进行聚合。

## 5.5 Job 3 - inverted index

### 5.5.1 Driver

```

1 // src/main/java/xyz/hakula/index/Driver.java
2
3 public class Driver extends Configured implements Tool {
4     private boolean runJob3(Path inputPath, Path outputPath)
5         throws IOException, InterruptedException, ClassNotFoundException {
6         var job3 = Job.getInstance(getConf(), "inverted index");
7         job3.setJarByClass(InvertedIndex.class);
8
9         job3.setInputFormatClass(SequenceFileInputFormat.class);
10        job3.setMapperClass(InvertedIndex.Map.class);
11        job3.setMapOutputKeyClass(Text.class);
12        job3.setMapOutputValueClass(TermFreqWritable.class);
13
14        job3.setReducerClass(InvertedIndex.Reduce.class);
15        job3.setNumReduceTasks(NUM_REDUCE_TASKS);
16        job3.setOutputKeyClass(Text.class);
17        job3.setOutputValueClass(TermFreqWritable.class);
18
19        FileInputFormat.addInputPath(job3, inputPath);
20        FileOutputFormat.setOutputPath(job3, outputPath);
21
22        return job3.waitForCompletion(true);
23    }
24}

```

和 Job 1 基本没什么区别，不再赘述。

这里不再需要设置 `setOutputFormatClass` 了，我们直接以文本格式输出。

### 5.5.2 Mapper

```

1 // src/main/java/xyz/hakula/index/InvertedIndex.java
2
3 public class InvertedIndex {
4     public static class Map extends Mapper<Text, TermFreqWritable, Text, TermFreqWritable> {
5         // Yield the Term Frequency (TF) of each token in each file.
6         @Override
7         public void map(Text key, TermFreqWritable value, Context context)
8             throws IOException, InterruptedException {
9             var fileTokenCount = readFromFile(context, value.getFilename());
10            value.setTermFreq((double) value getTokenCount() / fileTokenCount);
11            context.write(key, value);
12        }
13
14        private long readFromFile(Context context, String key) throws IOException {
15            var conf = context.getConfiguration();
16            var fs = FileSystem.get(conf);
17            var fileTokenCountPath = conf.get("fileTokenCountPath");
18            var inputPath = new Path(fileTokenCountPath, key);
19            try (var reader = new BufferedReader(new InputStreamReader(fs.open(inputPath)))) {
20                return Long.parseLong(reader.readLine());
21            }
22        }
23    }
24}

```

既然在 Job 2 的 Reducer 里不能得到 TF，那我们就在 Job 3 的 Mapper 里得到。当 Job 3 的 Mapper 需要一个文件的短语总数时，就从 Job 2 输出的中间文件里读取。顺便一提，MapReduce 的 Job 之间是顺序执行的，但同一个 Job 的 Mapper 和 Reducer 是并行的，因此我们也不能让 Mapper 或 Combiner 计算文件短语总数，然后在 Reducer 里读取。

将每个短语的出现次数除以文件的短语总数，我们就得到了短语的 TF，这下可以替换掉原来的占位符了。

### 5.5.3 Reducer

```
1 // src/main/java/xyz/hakula/index/InvertedIndex.java
2
3 public class InvertedIndex {
4     public static class Reduce extends Reducer<Text, TermFreqWritable, Text, TermFreqWritable> {
5         // Yield the Inverse Document Frequency (IDF) of each token.
6         @Override
7         public void reduce(Text key, Iterable<TermFreqWritable> values, Context context)
8             throws IOException, InterruptedException {
9             long fileCount = 0;
10            for (var value : values) {
11                context.write(key, value);
12                ++fileCount;
13            }
14            writeToFile(context, key.toString(), fileCount);
15        }
16
17        private void writeToFile(Context context, String key, long fileCount)
18            throws IOException {
19            var conf = context.getConfiguration();
20            var fs = FileSystem.get(conf);
21            var inverseDocumentFreqPath = conf.get("inverseDocumentFreqPath");
22            var outputPath = new Path(inverseDocumentFreqPath, parseFilename(key));
23
24            var totalFileCount = conf.getLong("totalFileCount", 1);
25            var inverseDocumentFreq = Math.log((double) totalFileCount / fileCount) / Math.log(2);
26            try (var writer = new BufferedWriter(new OutputStreamWriter(fs.create(outputPath, true)))) {
27                writer.write(fileCount + " " + inverseDocumentFreq + "\n");
28            }
29        }
30    }
31
32    public static String parseFilename(String filename) throws UnsupportedEncodingException {
33        // Use URL encoding to avoid invalid characters in filename.
34        filename = URLEncoder.encode(
35            filename.toLowerCase(Locale.ROOT),
36            StandardCharsets.UTF_8.name()
37        );
38        // Create buckets to limit the number of files in a single directory.
39        return filename.charAt(0) + "/" + filename;
40    }
41 }
```

最后的 Reducer 主要就是计算一下每个短语的 IDF。通过这次聚合，我们可以得到出现短语 key 的所有文档的 TF 条目，遍历一次就可以得到出现这个短语的文档总数了。然后我们从配置 `conf` 里读取文档总数 `totalFileCount`，除一下取个对数就得到了 IDF。

在本来的实现中，我将所有的 TF 条目聚合成了一个数组，然后短语的 IDF 也是和这个数组放在了一起。然而不幸的是，出现了和 Job 2 的 Reducer 一样的问题，爆内存了！数据集实在太大，即使只是 1 个短语的索引结果都放不进内存。我推测估计是像 `a`, `the` 这种常见短语，出现次数实在太多，所以就爆了。

于是我做了下调整，将同一短语在不同文件的索引结果分成了不同的条目，在遍历时就直接写进文件，不用再建一个巨大的数组了。聚合这些索引的逻辑交给检索程序完成。这样 TF 条目爆内存的问题是解决了，那 IDF 怎么办呢？和 Job 2 那时一样，IDF 显然是没办法写进索引里了，因为 IDF 要等 TF 条目全遍历完才能得到。因此同理，我们将 IDF 也写进另外的文件中。

本来到这里就结束了，本地也都跑得很正常。结果一上服务器，又出问题了。写入文件时出现了 `java.io.FileNotFoundException / org.apache.hadoop.ipc.RemoteException`。这就奇怪了，我是在写入文件，而且参数设定了 `overwrite` 为 `true`，怎么还会提示文件不存在呢？

调查了一段时间后，推测可能有两个原因：

1. HDFS 的目录有文件数量限制。在默认设置中，HDFS 单个目录里的最大文件数量是 1048576 [3]。而在报错时，检查发现目录 `<output_path>/inverse_document_freq` 下已有超过 600000 个文件，因此确实存在超过最大文件数量的风险。
2. 出现了写入冲突。虽然，首先写入冲突就不该发生，因为在最后的 Reducer 里，每个 key 一定是不一样的，所以不同 Reducer 写入的文件名必然不同。不过，确实还是存在命名冲突的可能。因为我为了泛用性和安全性考虑，即使项目要求里声称，文档的内容格式为分行、无标点的英文文本，我还是假设其中可能存在一些特殊字符。因此我在设置文件名时，做了一步 `replaceAll("\\W+", "_")` 处理，将所有非英文数字的字符都替换成 `_`，以确保文件名一定合法。那么假如文档的短语确实不是全英文的，而存在一些诸如 `foobar,` 和 `foobar.` 这样的短语，那就会导致冲突，从而出现并发写的问题。结果我检查目录 `<output_path>/inverse_document_freq` 里的文件发现，竟然真的有包含 `_` 的文件名。而且检查日志时也发现，报错的全部都是包含 `_` 的短语。所以推测比较可能是这个问题。

对于第一个问题，我的解决方案是：加个桶。也就是保存文件 `<token>` 时，加一级目录，保存到以 `<token>` 首字母命名的桶里。例如短语 `foobar` 的 IDF 就是保存到文件 `<output_path>/inverse_document_freq/f/foobar` 里。这样至少可以让单目录下的文件总数除以 26 了，如果还不够的话，我们也可以把桶名提高到 2 ~ 3 位字符，就是逻辑会稍微复杂一点点。

对于第二个问题，我的解决方案是：采用 URL 编码。既然简单替换成 `_` 不行，那就直接编码成 URL 格式嘛（利用 `URLEncoder.encode()`）。这下总归没问题了，而且也不会出现命名冲突。

至此，我们就成功实现了一个索引程序。

## 5.6 Woogle

下面简单讲讲检索程序的主类 `Woogle`，以下是基于 Java SE 17 的实现：

```
1 // src/main/java/xyz/hakula/woogle/Woogle.java
2
3 public class Woogle extends Configured implements Tool {
4     private static final Logger log = Logger.getLogger(Woogle.class.getName());
5
6     public static void main(String[] args) throws Exception {
7         var conf = new Configuration();
8         System.exit(ToolRunner.run(conf, new Woogle(), args));
9     }
10
11    public int run(String[] args) throws Exception {
12        var key = "";
13        try (var scanner = new Scanner(System.in)) {
14            System.out.print("Please input a keyword:\n> ");
15            key = scanner.nextLine().trim().toLowerCase(Locale.ROOT);
16        }
17        if (!key.isBlank()) {
18            var indexPath = new Path(args[0]);
19            searchAndPrint(key, indexPath);
20        }
21        return 0;
22    }
23 }
```

先讲大体框架，流程上很简单，就是：

1. 提示用户输入一个关键词
2. 在索引文件夹里搜索并输出结果

接下来讲一下具体实现。

#### 5.6.1 `searchAndPrint()`

```

1 // src/main/java/xyz/hakula/woogle/Woogle.java
2
3 public class Woogle extends Configured implements Tool {
4     protected void searchAndPrint(String key, Path indexPath) throws IOException {
5         var conf = getConf();
6         var fs = FileSystem.get(conf);
7
8         InverseDocumentFreq idf;
9         var inverseDocumentFreqPath = new Path(indexPath, "inverse_document_freq");
10        var filePath = new Path(inverseDocumentFreqPath, InvertedIndex.parseFilename(key));
11        try (var reader = new BufferedReader(new InputStreamReader(fs.open(filePath)))) {
12            var line = reader.readLine();
13            idf = InverseDocumentFreq.parse(line);
14            printInverseDocumentFreq(key, idf);
15        } catch (FileNotFoundException e) {
16            System.out.println(key + ": not found");
17            return;
18        }
19
20        TermFreq tf;
21        var partition = getPartition(key);
22        var termFreqsPath = new Path(indexPath, String.format("part-r-%05d", partition));
23        try (var reader = new BufferedReader(new InputStreamReader(fs.open(termFreqsPath)))) {
24            var line = "";
25            while ((line = reader.readLine()) != null) {
26                var lineSplit = line.split("\t");
27                var token = lineSplit[0];
28                if (Objects.equals(key, token)) {
29                    try {
30                        tf = TermFreq.parse(lineSplit[1]);
31                        printInvertedIndex(tf, idf);
32                    } catch (Exception e) {
33                        log.warn(token + ": invalid index entry, error: " + e);
34                    }
35                }
36            }
37        } catch (FileNotFoundException e) {
38            log.error(key + ": index not exists");
39        }
40    }
41 }

```

这个是检索程序的核心代码。总体分两步走：

- 根据关键词 `<key>`，查找 IDF 文件，读取文件的命名方式同 Job 3 的 Reducer。如果能查到，说明关键词在索引里存在，解析出 IDF 并继续接下来的步骤，否则输出 `<key>: not found`。
- 根据关键词 `<key>`，先利用函数 `getPartition()` 定位到相应的 TF 文件（将在下一个章节讲解），然后逐行遍历查询，查询到与 `<key>` 相同的 `<token>` 后，解析出 TF 并输出。

为什么要遍历查询呢？主要还是因为并行的 MapReduce 程序不保序，不一定可以进行二分查找。事实上如果觉得慢的话，完全可以把最后一个 Reducer 的任务数量设置得大一点，因为最后索引的分段数量就等于这个 Reducer 的任务数量，总可以设置到一个足够大的值，使得线性复杂度的耗时可以接受。

## 5.6.2 `getPartition()`

```
1 // src/main/java/xyz/hakula/woogle/Woogle.java
2
3 public class Woogle extends Configured implements Tool {
4     protected int getPartition(String key) {
5         var textKey = new Text(key);
6         return (textKey.hashCode() & Integer.MAX_VALUE) % Driver.NUM_REDUCE_TASKS;
7     }
8 }
```

函数 `getPartition()` 的实现很简单，其实就是沿用了 Job 2 的默认 Partitioner 的分配方法，也就是直接对 `<key>` 哈希，然后 mod 一下 Reducer 的任务数量。这样就可以定位到当时 `reduce()` 这个 `<key>` 的 Reducer，从而定位到相应的索引片段。

### 5.6.3 InverseDocumentFreq.parse()

```
1 // src/main/java/xyz/hakula/woogle/model/InverseDocumentFreq.java
2
3 public record InverseDocumentFreq(long fileCount, double inverseDocumentFreq) {
4     private static final String DELIM = " ";
5
6     public static InverseDocumentFreq parse(String entry) {
7         var entrySplit = entry.split(Pattern.quote(DELIM));
8         var fileCount = Long.parseLong(entrySplit[0]);
9         var inverseDocumentFreq = Double.parseDouble(entrySplit[1]);
10        return new InverseDocumentFreq(fileCount, inverseDocumentFreq);
11    }
12}
```

函数 `InverseDocumentFreq.parse()` 就是解析一下 IDF 文件的内容，简单 `split()` 一下就行。

### 5.6.4 printInverseDocumentFreq()

```
1 // src/main/java/xyz/hakula/woogle/Woogle.java
2
3 public class Woogle extends Configured implements Tool {
4     private void printInverseDocumentFreq(String token, InverseDocumentFreq idf) {
5         System.out.printf(
6             "%s: IDF = %6f | found in %d files:\n",
7             token,
8             idf.inverseDocumentFreq(),
9             idf.fileCount()
10            );
11    }
12 }
```

对 IDF 进行普通的格式化输出。

### 5.6.5 TermFreq.parse()

```

1 // src/main/java/xyz/hakula/woogle/model/TermFreq.java
2
3 public record TermFreq(String filename, long tokenCount, double termFreq, long[] positions) {
4     private static final String DELIM = ":";  

5     private static final String POS_ARRAY_DELIM = ";";
6
7     public static TermFreq parse(String entry) {
8         var entrySplit = entry.split(Pattern.quote(DELIM));
9         var filename = entrySplit[0];
10        var tokenCount = Long.parseLong(entrySplit[1]);
11        var termFreq = Double.parseDouble(entrySplit[2]);
12        var positionsSplit = entrySplit[3].split(Pattern.quote(POS_ARRAY_DELIM));
13        var positions = Arrays.stream(positionsSplit).mapToLong(Long::parseLong).toArray();
14        return new TermFreq(filename, tokenCount, termFreq, positions);
15    }
16}

```

函数 `TermFreq.parse()` 就是解析一下 TF 条目的内容，同样简单 `split()` 一下就行。`positions` 的解析用了个比较函数式的写法。其实 MapReduce 本身就很函数式，只可惜 Java 不太函数式，写起来就不怎么优雅。

### 5.6.6 `printInvertedIndex()`

```

1 // src/main/java/xyz/hakula/woogle/Woogle.java
2
3 public class Woogle extends Configured implements Tool {
4     private void printInvertedIndex(TermFreq tf, InverseDocumentFreq idf) {
5         System.out.printf(
6             " %s: TF = %6e (%d times) | TF-IDF = %6e | positions:",
7             tf.filename(),
8             tf.termFreq(),
9             tf.tokenCount(),
10            tf.termFreq() * idf.inverseDocumentFreq()
11        );
12        for (var position : tf.positions()) {
13            System.out.print(" ");
14            System.out.print(position);
15        }
16        System.out.println();
17    }
18}

```

对 TF 进行普通的格式化输出。

至此，我们就成功实现了一个检索程序。

当然有了索引文件，想写个网页界面啦，或者对结果按 TF-IDF 的大小排个序啦都很容易。主要还是期末季太忙了，实在没时间，不然都好做。

## 许可协议

本项目遵循 MIT 许可协议，详情参见 [LICENSE](#) 文件。

1. [java - Get unique line number from a input file in MapReduce mapper - Stack Overflow ↵](#)
2. [java - Static variable value is not changing in mapper function - Stack Overflow ↵](#)
3. [Apache Hadoop 3.3.1 - Hadoop - Configuration - hdfs-default.xml ↵](#)