

**课 程 实 验 报 告**

**课程名称：Java语言程序设计**

**实验名称：基于内存的搜索引擎设计和实现**

**院 系 ：计算机科学与技术**

**专业班级 ：**

**学 号 ：**

**姓 名 ：**

**指导教师 ：**

**2022 年 4 月 30 日**

# 目 录

**1 需求分析** 4

1.1 题目要求 4

1.2 需求分析 5

**2 系统设计** 8

2.1 设计概要 8

2.1.1 项目架构设计 8

2.1.2 index模块 8

2.1.3 parse模块 11

2.1.4 query模块 14

2.2 详细设计 14

2.2.1 index模块 14

2.2.2 parse模块 18

2.2.3 query模块 20

**3** **软件开发** 24

3.1 软件开发环境说明 24

3.2 软件运行说明 24

**4 软件测试** 25

4.1 自动测试结果 25

4.2 数据集测试结果 26

4.2.1 功能性测试数据 26

4.2.2 真实测试数据 27

**5 特点与不足** 30

5.1 技术特点 30

5.2 不足 30

**6 过程和体会** 31

6.1 遇到的问题和解决方法 31

6.2 课程设计的体会 31

**7 源码和说明** 32

7.1 文件清单及功能说明 32

7.2 用户使用说明 32

7.3 源代码 32

# 1 需求分析

## 题目要求

实现一个基于内存的英文全文检索搜索引擎，需要完成以下功能：

1. 将指定目录下的一批.txt格式的文本文件扫描并在内存里建立倒排索引，这里面包含必须的子功能包括：
2. 读取文本文件的内容；
3. 将内容切分成一个个的单词；
4. 过滤掉其中一些不需要的单词,例如数字、停用词（the, is and这样的单词）、过短或过长的单词（例如长度小于3或长度大于20的单词）；
5. 利用Java的集合类在内存里建立过滤后剩下单词的倒排索引；
6. 内存里建立好的索引对象可以序列化到文件，同时可以从文件里反序列化成内存里的索引对象；
7. 可以在控制台输出索引的内容。
8. 基于构建好的索引，实现单个搜索关键词的全文检索，包含的子功能包括：
9. 根据搜索关键词得到命中的结果集合；
10. 可以计算每个命中的文档的得分，并根据文档得分对结果集排序；
11. 在控制台显示命中的文档的详细信息，如文档的路径、文档内容、命中的关键词信息（如在文档里出现次数）、文档得分；
12. 基于构建好的索引，实现二个搜索关键词的全文检索。包含的子功能包括：
13. 支持这二个关键词的与或查询。与关系必须返回同时包含这二个单词的文档集合，或关系返回包含这二个单词中的任何一个的文档集合；
14. 可以计算每个命中的文档的得分，并根据文档得分对结果集排序；
15. 在控制台显示命中的文档的详细信息，如文档的路径、文档内容、命中的关键词信息（如在文档里出现次数）、文档得分；
16. 基于构建好的索引，实现包含二个单词的短语检索，即这二个单词必须在作为短语文档里出现，它们的位置必须是相邻的，这个功能为进阶功能。
17. 除了以上功能上的要求外，其他要求包括：
18. 针对搜索引擎的倒排索引结构，已经定义好了创建索引和全文检索所需要的抽象类和接口。学生必须继承这些预定义的抽象类和和实现预定义接口来完成实验的功能，不能修改抽象类和接口里规定好的数据成员、抽象方法；也不能在预定义抽象类和接口里添加自己新的数据成员和方法。但是实现自己的子类和接口实现类则不作任何限定。
19. 自己实现的抽象类子类和接口实现类里的关键代码必须加上注释，其中每个类、每个类里的公有方法要加上Javadoc注释，并自动生成Java API文档作为实验报告附件提交。
20. 使用统一的测试文档集合、统一的搜索测试案例对代码进行功能测试，构建好的索引和基于统一的搜索测试案例的检索结果最后输出到文本文件里作为实验报告附件提交。
21. 本实验只需要基于控制台实现，实验报告里需要提供运行时控制台输出截屏。
22. 关于搜索引擎的倒排索引结构、相关的抽象类、接口定义、还有相关已经实现好的工具类会在单独的PPT文档里详细说明。同时也为学生提供了预定义抽象类和接口的Java API文档和UML模型图。

## 需求分析

程序的整体需求分析图如图1.2.1所示。

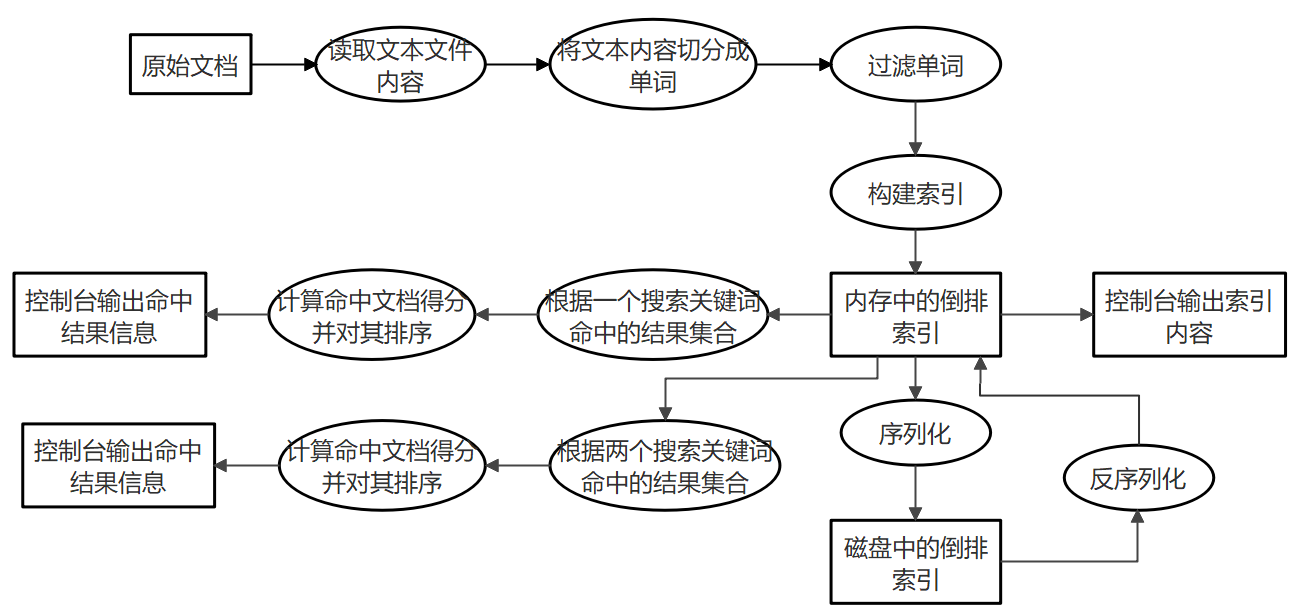


图1.2.1 系统需求分析图

1. 将指定目录下的一批.txt格式的文本文件扫描并在内存里建立倒排索引

实现需求1主要基于IndexBuilder类中的buildIndex方法，该方法的效果为遍历和解析一个目录下的文本文件，得到对应的Document对象，再一次加入到索引，并将索引保存到文件。该方法的接收参数：一个类型为String的参数表示文本文件的根路径。返回类型为AbstractIndex。具体实现如表1.2.1所示。

表1.2.1 需求1示意表

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 子需求 | 对应方法 | 接收参数 | 返回类型 | 异常捕获 |
| 读取文本文件的内容; 将内容切分成一个个的单词；过滤掉其中一些不需要的单词。 | DocumentBuilder类中的build方法 | int docId;  String docPath;  File file | AbstractDocument | FileNotFoundException |
| 在内存里建立过滤后剩下单词的倒排索引； | DocumentBuilder类中的build方法 | int docId;  String docPath;  AbstractTermTupleStream termTupleStream | AbstractDocument | 无 |
| 内存里建立好的索引对象可以序列化到文件 | Index类中的save方法 | File file | void | IOException |
| 从文件里反序列化成内存里的索引对象 | Index类中的load方法 | File file | void | IOException |
| 在控制台输出索引的内容 | Index类中的toString方法 | 无 | String | 无 |

1. 基于构建好的索引，实现单个搜索关键词的全文检索：

实现需求2主要基于IndexSearcher中的search方法，该方法的效果为基于现有的Index根据单个检索词进行搜索。该方法接收参数：第一个参数类型为AbstractTerm表示检索词，第二个参数类型为Sort，表示排序器。返回类型为AbstractHit[]。具体实现如表所示。

表1.2.2 需求2示意表

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 子需求 | 对应方法 | 接收参数 | 返回类型 | 异常捕获 |
| 根据搜索关键词得到命中的结果集合 | IndexSearcher中的search方法 | AbstractTerm queryTerm;  Sort sorter | AbstractHit[] | 无 |
| 计算每个命中的文档的得分，并根据文档得分对结果集排序 | SimpleSorter类中的sort方法 | List<AbstractHit> hits | void | 无 |
| 在控制台显示命中的文档的详细信息 | Hit类中的toString方法 | 无 | String | 无 |

1. 基于构建好的索引，实现二个搜索关键词的全文检索：

实现需求3主要基于IndexSearcher中的search方法，该方法的效果为基于现有的Index根据两个检索词进行搜索。该方法接收参数：第一个参数类型为AbstractTerm表示第一个检索词，第二个参数类型为AbstractTerm表示第一个检索词，第三个参数类型为Sort，表示排序器。返回类型为AbstractHit[]。具体实现如表所示。

表1.2.3 需求3示意表

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 子需求 | 对应方法 | 接收参数 | 返回类型 | 异常捕获 |
| 支持这二个关键词的与或查询 | IndexSearcher中的search方法 | AbstractTerm queryTerm1, AbstractTerm queryTerm2, Sort sorter, LogicalCombination combine | AbstractHit[] | 无 |
| 可以计算每个命中的文档的得分，并根据文档得分对结果集排序 | SimpleSorter类中的sort方法 | List<AbstractHit> hits | void | 无 |
| 在控制台显示命中的文档的详细信息 | Hit类中的toString方法 | 无 | String | 无 |

# 2 系统设计

## 设计概要

#### 2.1.1项目架构设计

项目的架构如图2.1.1所示。

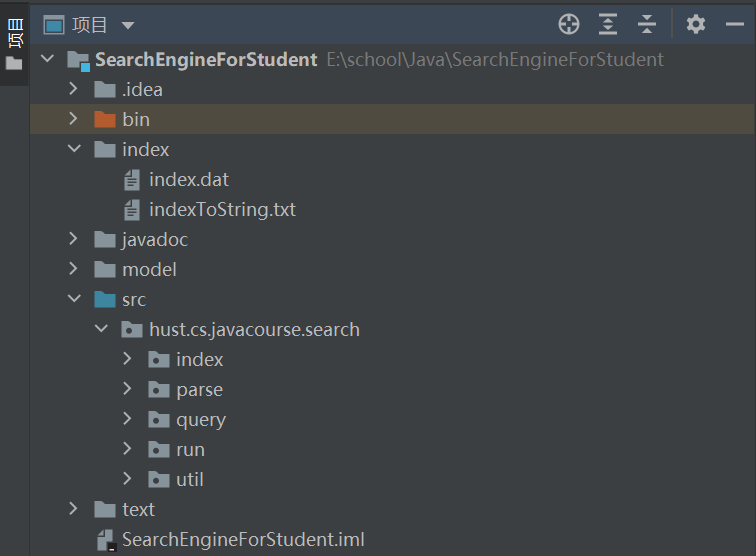


图2.1.1 项目架构图

#### index模块

该模块主要包括，倒排索引以及构建倒排索引过程中所需的各种中间结果的结构类、构建倒排索引及各种中间结果所需的各种工具类。

1. 定义倒排索引的结构及构成倒排索引的子结构
2. Term

表示文本文档里的一个单词

1. Posting

表示倒排索引的每一项，一个posting对象包括

* 1. 包含单词的文档编号docId
  2. 单词在文档里出现的次数freq

③ 单词在文档里出现的位置列表positions（由于单词在文档中可能会出现多次，因此positions应该是一个列表）

1. PostingList

表示一个Posting的列表。因为一个单词会出现在多个文档中，对应多个Posting，故每个Term需要维护一个对应的PostingList。

1. Index

倒排索引结构。记录一个docId和docPath的映射关docIdToDocPathMapping，还有一个Term和PostringList的映射关系termToPostingListMapping。

1. 定义一些在构建索引过程中所需要的中间结构
2. TermTuple

构建索引过程中，解析一个文档时，得到的一个单词的中间结果三元组。一个TermTuple对象包括：

* 1. Term
  2. freq：单词出现次数，因为解析的对象是一个个单词，所以始终为1。
  3. curPos：该单词在此文档中出现的位置。

1. Document

构建索引过程中，解析一个文档得到的结果。一个Document对象包括

1. docId：文档编号。
2. docPath：文档路径。
3. tuples：该文档包含的TermTuple列表。
4. 定义在构建索引过程中所需要的各种工具类
5. DocumentBulider

构建Document结构所需的工具类。含有两个重载的build方法，用来构建Document结构。

* 1. build方法1。接收参数docId，docPath以及termTupleStream(三元组流)，返回生成的Document结构。
  2. bulid方法2。接收参数docId，docPath以及文件指针，先通过parse模块中的各种工具类解析得到过滤后的三元组流，再调用bulid方法1返回生成的Document结构。程序结构如图2.1.2所示。

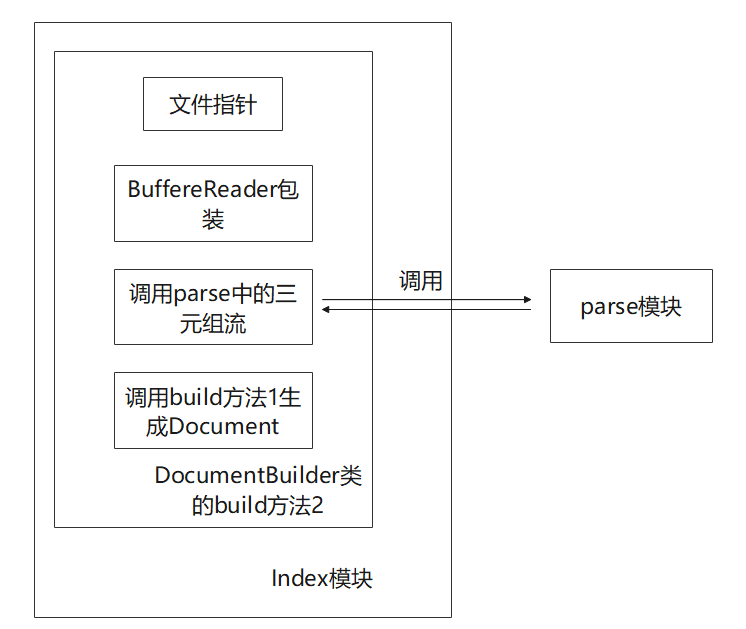


图2.1.2 build方法2

1. IndexBulider

构建Index结构所需的工具类。IndexBuilder类只有一个buildIndex方法。该方法接收一个String类型的参数，待解析的文本所存放的目录，返回一个index索引结构。该方法先对每个文档解析生成对用的Document结构，再依次对每个Document掉用Index类的addDocument，逐步生成Index。程序结构如图2.1.3所示。

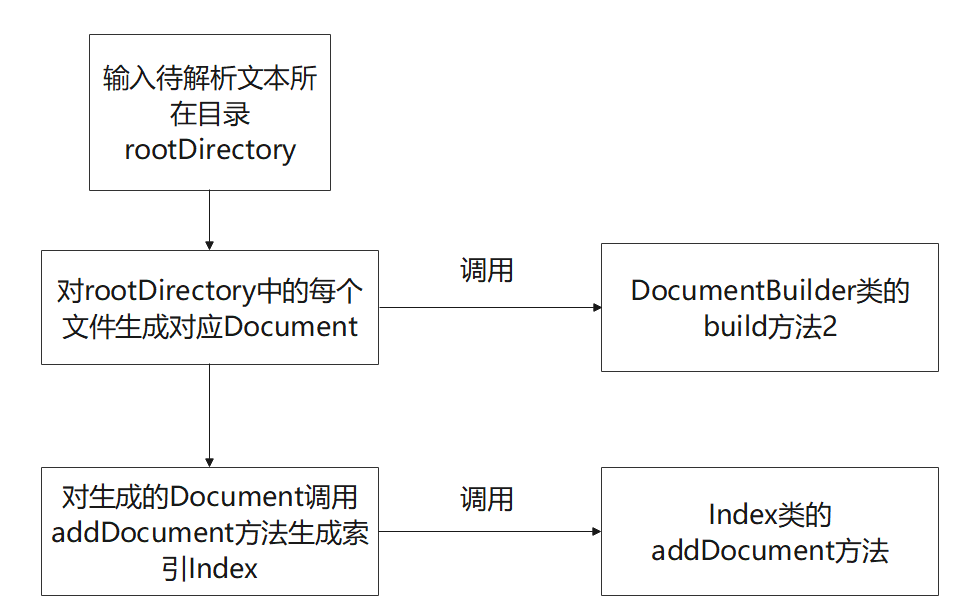


图2.1.3 buildIndex方法

1. 构建倒排索引文档

由原始文档构建倒排索引的总体流程图如图2.1.4所示。

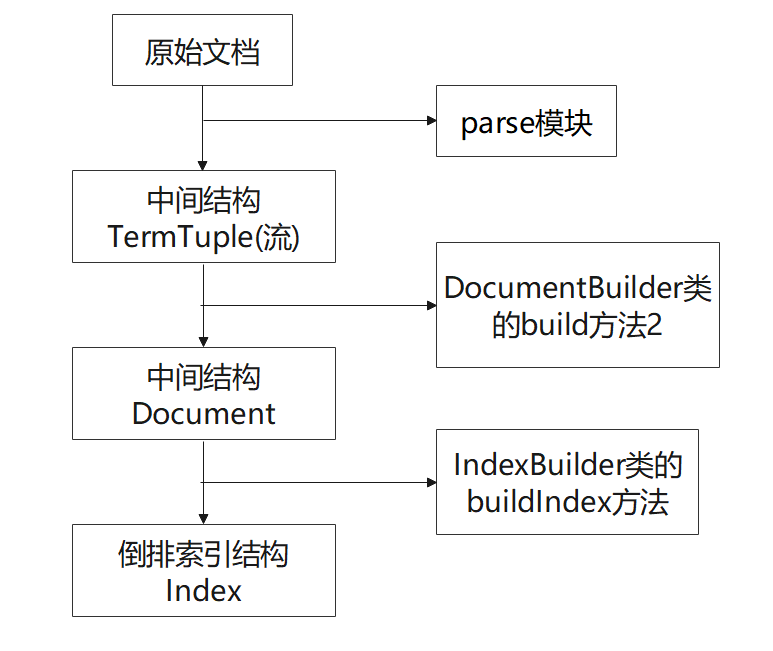


图2.1.4 倒排索引构建总体流程图

#### parse模块

1. 定义工具类

parse模块的所有工具类都是抽象类AbstractTermTupleStream的子类，采用了装饰者模式，每个AbstractTermTupleStream的子类都有next方法用于产生三元组流（或者过滤原三元组流产生新的三元组流），和close方法用于关闭三元组流。

1. TermTupleScanner类

该类具有两个内部成员。String类型的列表words，用于存放读取的每个单词，用于生成三元组流；pos用于记录目前输出到了三元组流的哪一个位置。

该类参数为BufferedReader类，读取BufferedReader所指向的原始文档，并解析产生三元组流。

该类的next方法实现此过程。每调用一次next方法返回一个三元组，循环调用next方法即可生成三元组流。

1. xxxxTermTupleFilter类

该类用于对生成的三元组流进行过滤，仍返回三元组流。

该类参数为AbstractTermTupleStream类，每次过滤可直接在上一次返回的三元组流上进行装饰。改写相应的next方法即可。

1. 由原始文档到三元组流

由原始文档到三元组流的过程采用装饰者模式，进行读取BufferedReader对象形成三元组流的TermTumpleScanner类和对三元组流进行过滤的多个类都是AbstractTermTumple类的子类。每装饰一次便对被装饰对象进行一种处理。经过多种处理后得到想要的三元组流。

首先将文件指针经过装饰后得到BufferedReader对象，用来每次读取文本文档的一行。（以上过程在index模块BuildDocument类的build方法中完成）。将得到的BufferedReader对象进行装饰，使用TermTumpleScanner对象中的next方法获取下一个三元组对象，形成三元组流。如图2.1.5所示。

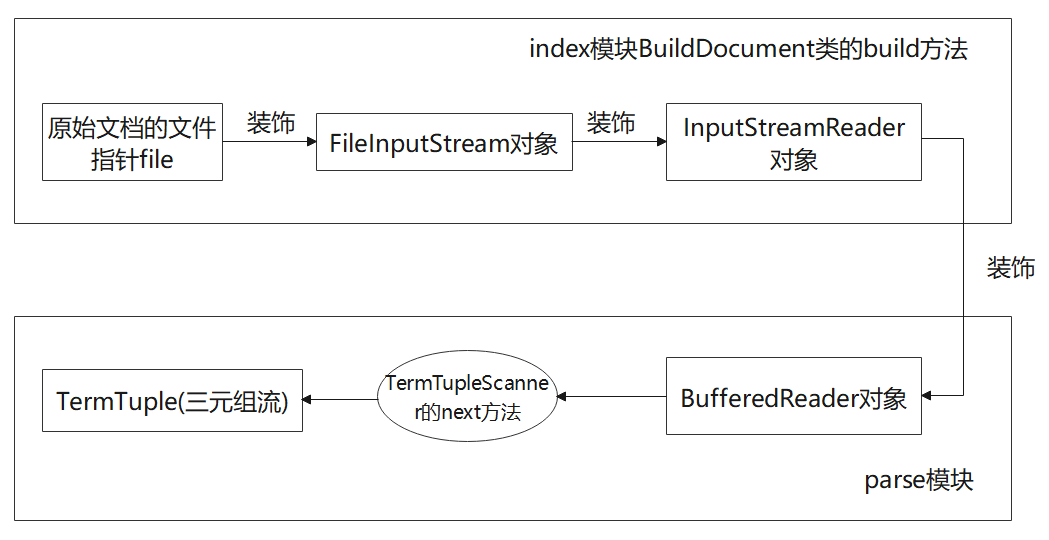


图2.1.5 由原始文档得到三元组流

1. 由三元组流到Document

由三元组流得到中间结构Document的过程，首先对三元组流进行多次过滤，依次使用LengthTermTupleFilter, PatternTermTupleFilter,

StopWordTermTupleFilter对三元组流进行过滤，过滤掉一些停用词，得到过滤后的三元组流。然后，组织形成三元组列表，添加上docId和docPath信息即可得到一个Document对象。如下图2.1.6所示。

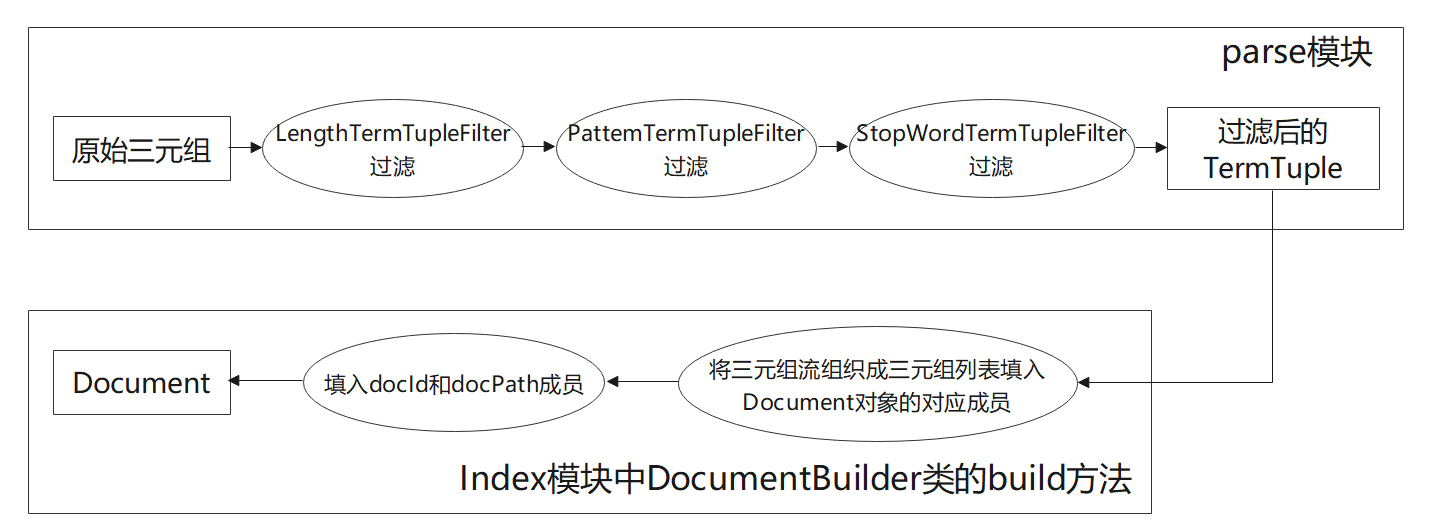


图2.1.6 由三元组流得到Document

1. parse模块总体结构及被调用情况

parse模块被DocumentBuilder类的build方法2调用，用于对原始文本文档进行解析，生成Index模块中定义的TermTuple流，并将得到的TermTuple流进行过滤操作，按照过滤规则过滤掉不需要的单词，最终得到的三元组流交由DocumentBuilder类的build方法1生成中间结构Document。程序流程如图2.1.7所示。

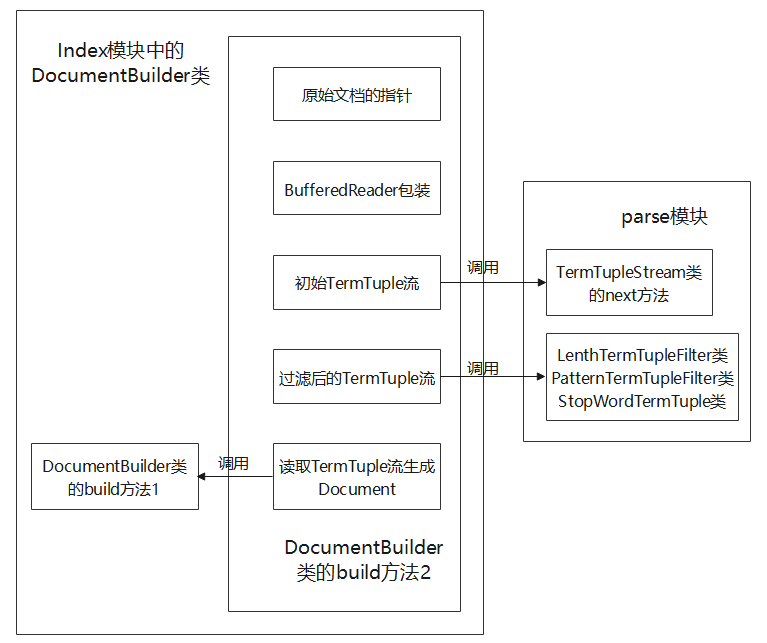


图2.1.7 parse模块总体结构及被调用情况图

#### query模块

1. 定义结构和工具类
2. Hit

Hit是根据所给检索词在已构建好的Index中检索得到的结果。一个Hit对象包括

1. docId：文档编号。
2. docPath：文档路径。
3. content：被检索的文档的文本内容。
4. termPostingMapping：命中的Term和对应的Posting键值对。
5. score：该命中文档的得分，通过单词出现次数计算。
6. SimpleSorter

实现了Sorter接口的类，用于计算命中结果的分数（score方法）以及对命中结果的数组按照分数进行排序（sort方法）。

1. IndexSearcher

用于检索的工具类。有两个重载的方法IndexSearcher，用于实现基于构建好的倒排索引的检索操作。IndexSearcher方法1进行单个检索词检索，IndexSearcher方法2根据两个检索词按照不同逻辑关系组合检索。每次检索前先将构建好的倒排索引Index读取到IndexSearcher的内部成员index，再调用searcher方法进行检索操作。

1. 检索

使用IndexSearcher类的search方法进行检索，每次检索前先将构建好的倒排索引Index读取到IndexSearcher的内部成员index，再根据检索词在Index中查找，其中会调用到Index模块中相关结构的各种方法，具体实现见详细设计部分。

### 详细设计

#### index模块

1. DocumentBuilder类
2. build方法1

入口参数：docId，docPath，三元组流对象TermTupleStream

出口参数：Document类对象

功能：根据三元组流对象的next方法获取一个个三元组，以构建生成索引所需的中间结构Document。流程图如图2.2.1所示。

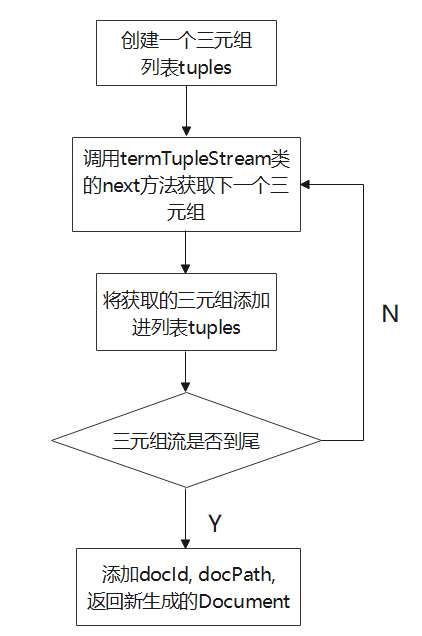


图2.2.1 build方法1

1. build方法2

入口参数：docId，docPath，文件指针file

出口参数：Document类对象

功能：读取文件指针file对应的文档内容，使用BufferedReader对其进行装饰后，将生成的BufferedReader对象使用TermTupleScanner类生成三元组流，并通过PatternTermTupleFilter、LengthTermTupleFilter、TermTupleScanner三层过滤后，在经过滤后的三元组流的基础上，调用build方法1构建生成索引所需的中间结构Document。流程图如图2.2.2所示。

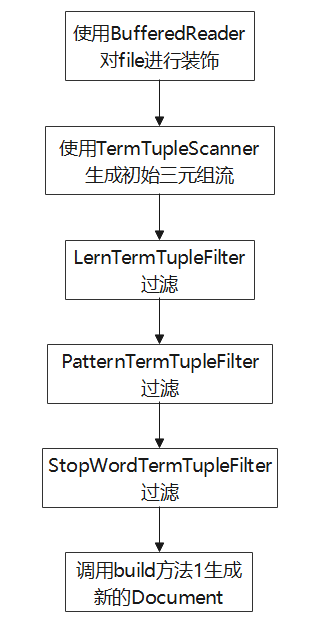


图2.2.2 build方法2

1. Index类
2. addDocument方法

入口参数：Document类对象

出口参数：无

功能：根据一个Document对象的内容构建索引。如图2.2.3所示。

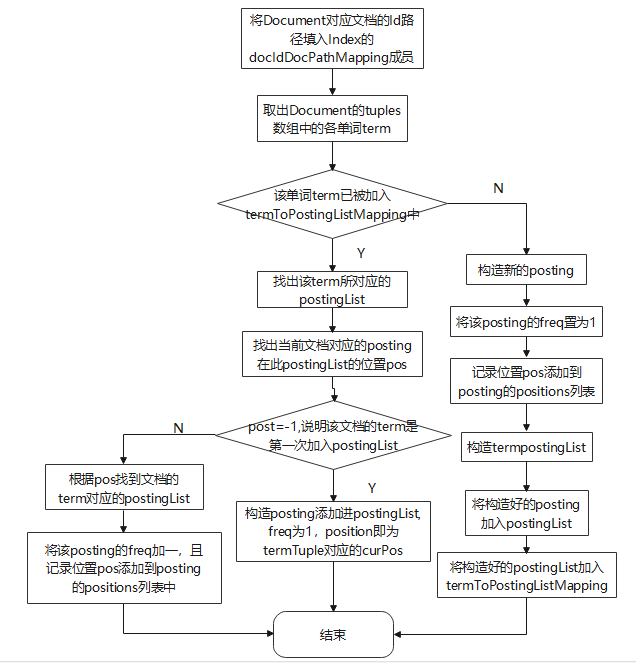


图2.2.3 addDocument方法

1. IndexBuilder类
2. buildIndex方法

入口参数：待解析文档的目录rootDirectory

出口参数：倒排索引index

功能：根据所给目录，解析该目录下所有文档，构建起倒排索引结构Index。如图2.2.4所示。

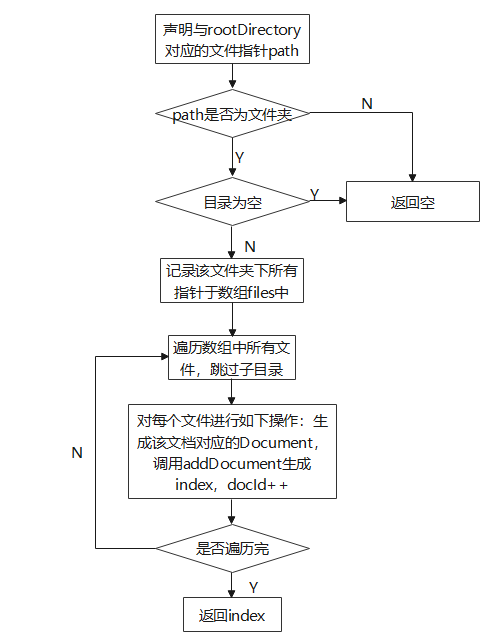


图2.2.4 buildIndex方法

#### parse模块

1. TermTupleScanner类
2. next方法

入口参数：无

出口参数：三元组对象TermTuple

功能：根据TermTumpleScanner类的构造函数获取的BufferedReader参数，对该文档进行逐行读取，并拆分单词，生成一个个三元组。流程图如图2.5所示。每次next方法被调用时，如果words列表为空的话就从原始文档中读取一行进行分词后放入words列表，然后将words列表首元素构造成一个TermTuple返回，并删除words列表中的该单词。如图2.2.5所示。

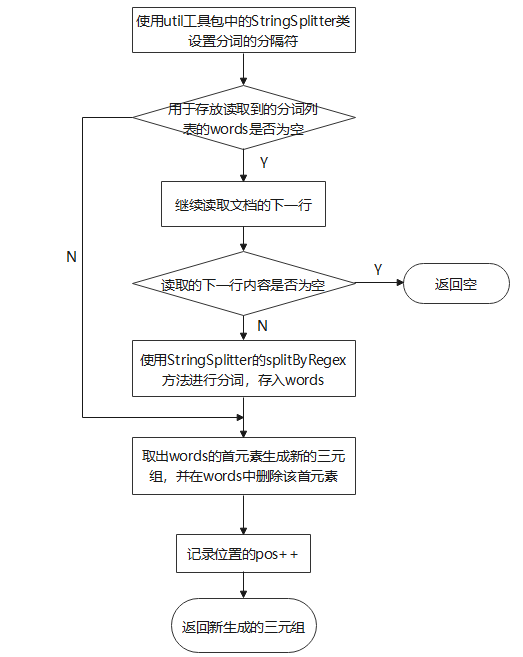


图2.2.5 next方法

1. StopWordTermTupleFilter,PatternTermTupleFilter,LengthTermTupleFilter类
2. next方法

入口参数：无

出口参数：三元组对象TermTuple

功能：根据类的构造函数获取的参数AbstractTermTupleStream，对得到的三元组流中的一个个三元组按照相应的过滤规则进行过滤。StopWordTermTupleFilter对应的规则是去掉一些停用词（英文中非常常用的一些词语），PatternTermTupleFilter对应的规则是只选取由26个大小写字母构成的字符串（即英文单词），LengthTermTupleFilter对应的规则是只选择长度在3-20之间的单词。

#### query模块

1. SimpleSorter接口
2. score方法

入口参数：命中结果Hit

出口参数：该Hit的分数score

功能：计算传入Hit的分数score，以便用于给命中结果排序。

1. IndexSearcher类
2. search方法1

入口参数：一个检索单词queryTerm，用于确定计算分数方式和排序方式的接口Sort

出口参数：检索的结果数组Hits

功能：根据检索词queryTerm在文档中检索得到命中结果数组Hits。流程图如图2.2.6所示。

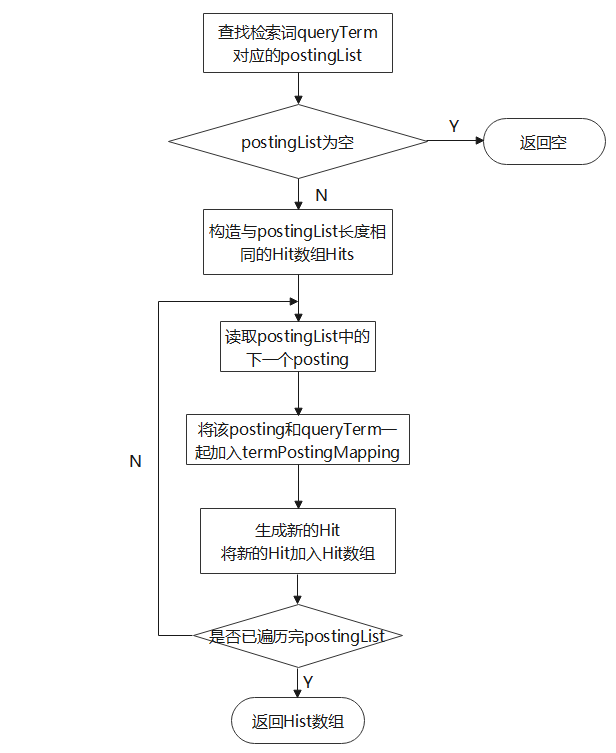


图2.2.6 search方法1

1. search方法2

入口参数：两个检索单词queryTerm1、queryTerm2，用于确定计算分数方式和排序方式的接口Sort，两个检索词的组合方式combine

出口参数：检索的结果数组Hits

功能：根据两个检索词以及它们不同的组合方式在文档中检索得到命中结果数组Hits。流程图如图2.2.7、图2.2.8、图2.2.9所示。

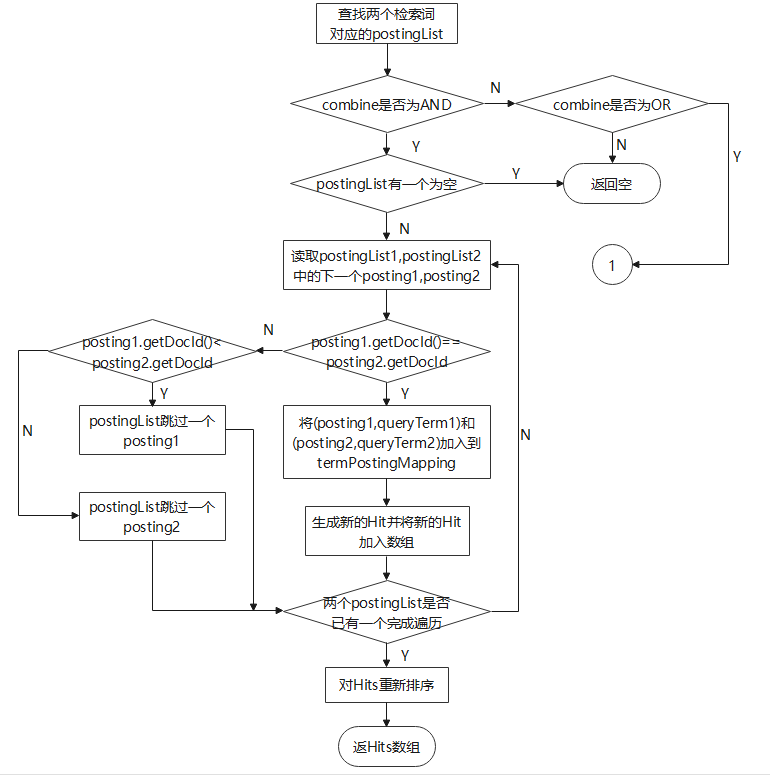


图2.2.7 search方法2

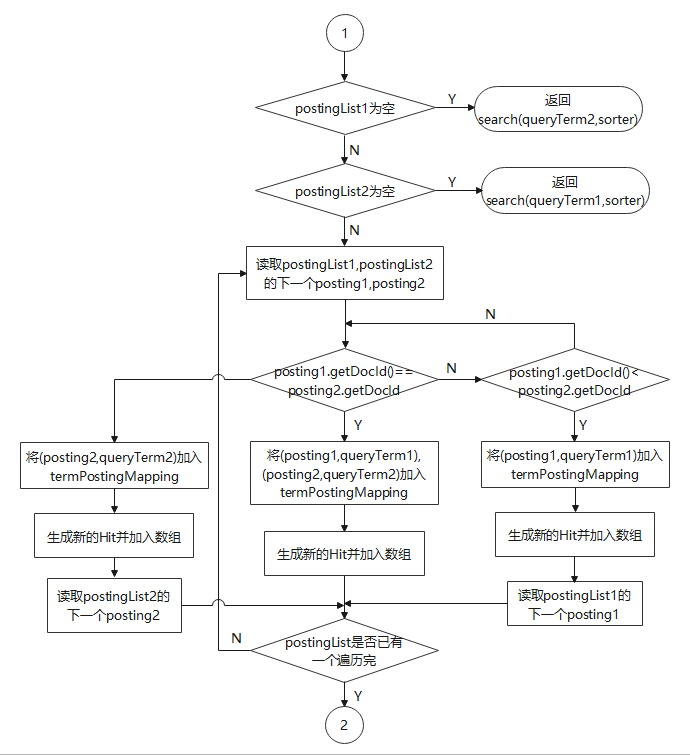


图2.2.8 search方法2

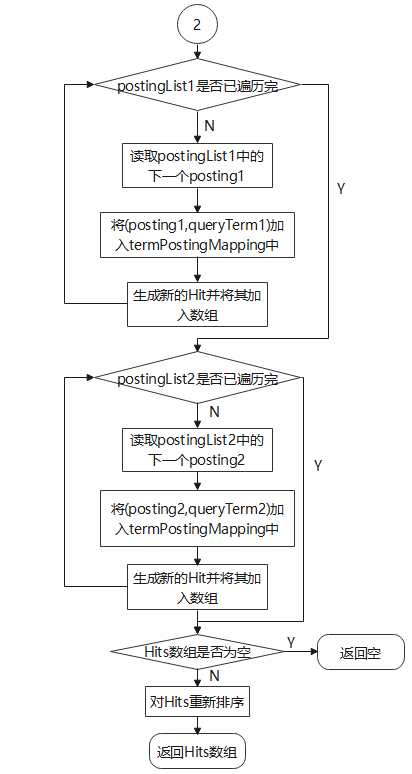


图2.2.8 search方法2

# 3 软件开发

## 软件开发

#### 3.1.1软件开发环境说明

处理器：Intel(R) Core(TM) i5-1035G1 CPU @ 1.00GHz，1190 Mhz，4 个内核

内存：8.0 GB

操作系统：Microsoft Windows 10 专业版19044

集成开发环境：IntelliJ IDEA 2020. 1(Ultimate Edition)

JDK版本：jdk-13.0.2

#### 3.1.2软件运行说明

通过IDEA中的构建功能可以生成可执行jar包。具体操作如下：

文件→项目结构→工件→左上角+号→点击JAR→来自具有依赖项的模块→选择主类→确定；

构建→构建工件→SearchEngineForStudent.jar→构建

最后通过在相应目录下进入cmd通过java -jar jar包名 即可执行程序。

# 4 软件测试

## 4.1 自动测试结果

test.bat脚本自动测试的运行结果如图4.1.1所示。

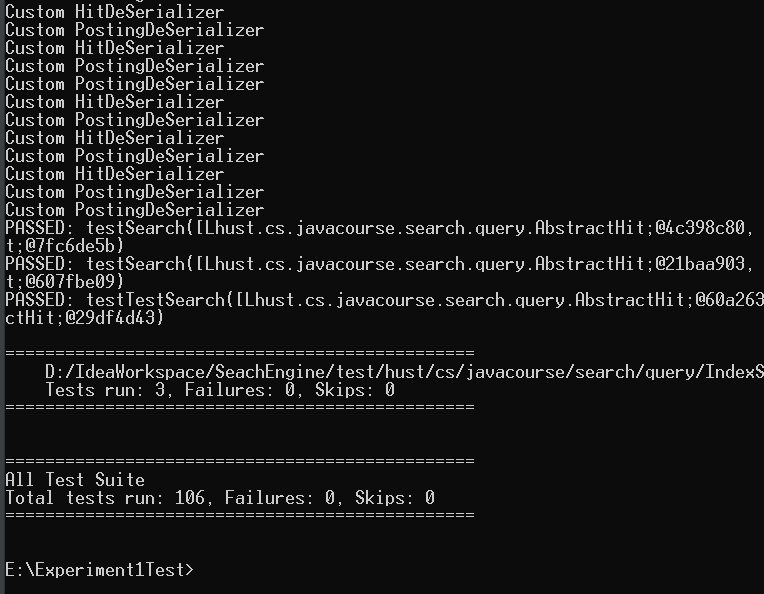


图4.1.1 test.bat自动测试结果

进入到子目录 test-output，打开 index.html，可以看到一共有 106 个测试用例，全部通过，如图4.1.2所示。

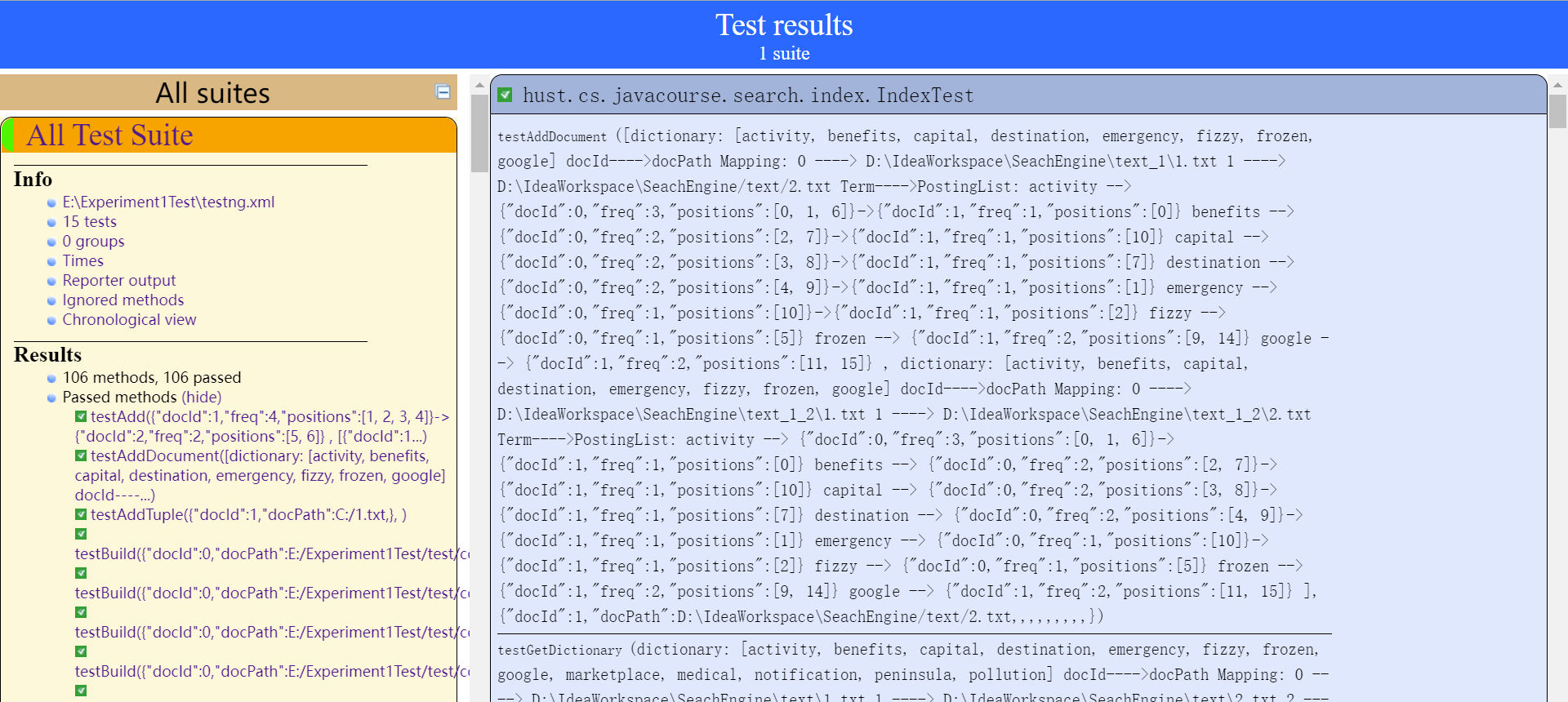


图4.1.2 index.html测试结果

## 数据集测试

#### 4.2.1功能性测试数据集的测试结果

1. 如果不采用任何过滤器，则用功能性测试数据集构建索引的结果示意如图4.2.1所示。

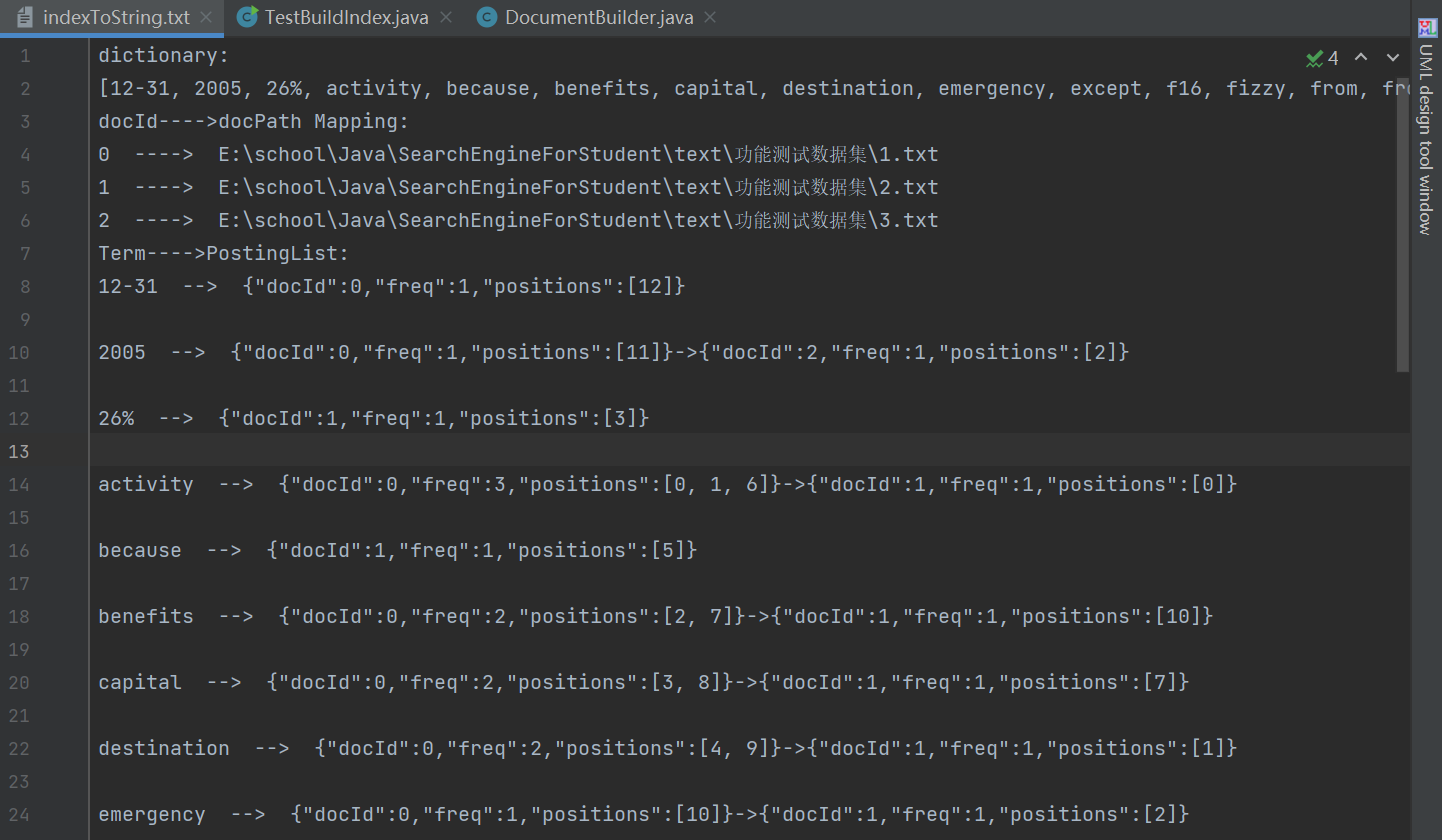


图4.2.1 用功能性测试数据集构建索引结果(不带任何过滤器)

1. 如果加上停用词过滤器、正则表达式过滤器、单词长度过滤器，则将停用词、包含非字母字符的单词、长度小于 3 或大于 20 的单词全部过滤掉后，构建索引的结果示意如图4.2.2所示。

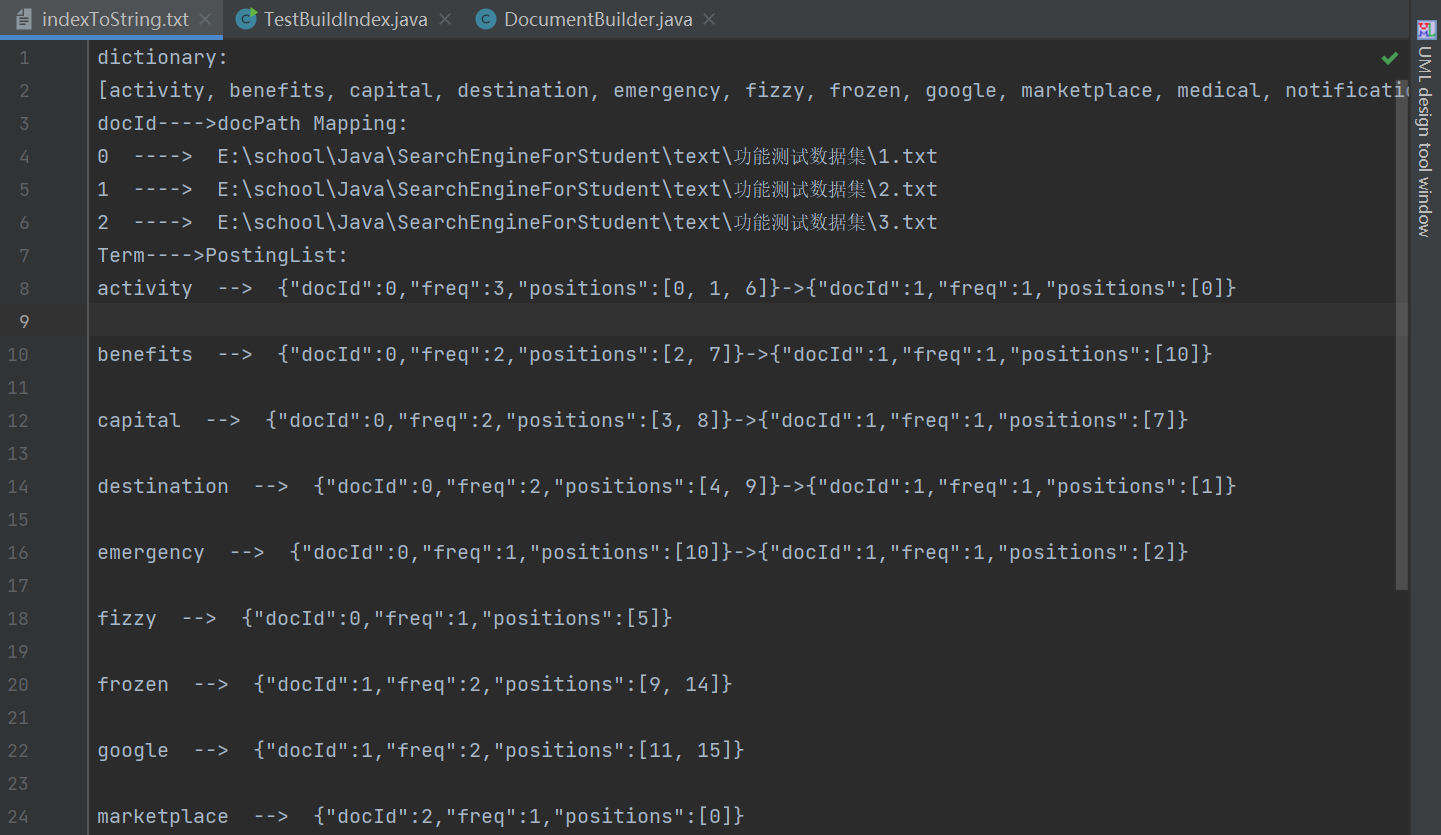


图4.2.2 用功能性测试数据集构建索引结果(带三种过滤器)

#### 4.2.2真实测试数据集的测试结果

1. 采用真实测试数据集，不加任何过滤器，构建索引的结果示意如图4.2.3所示。

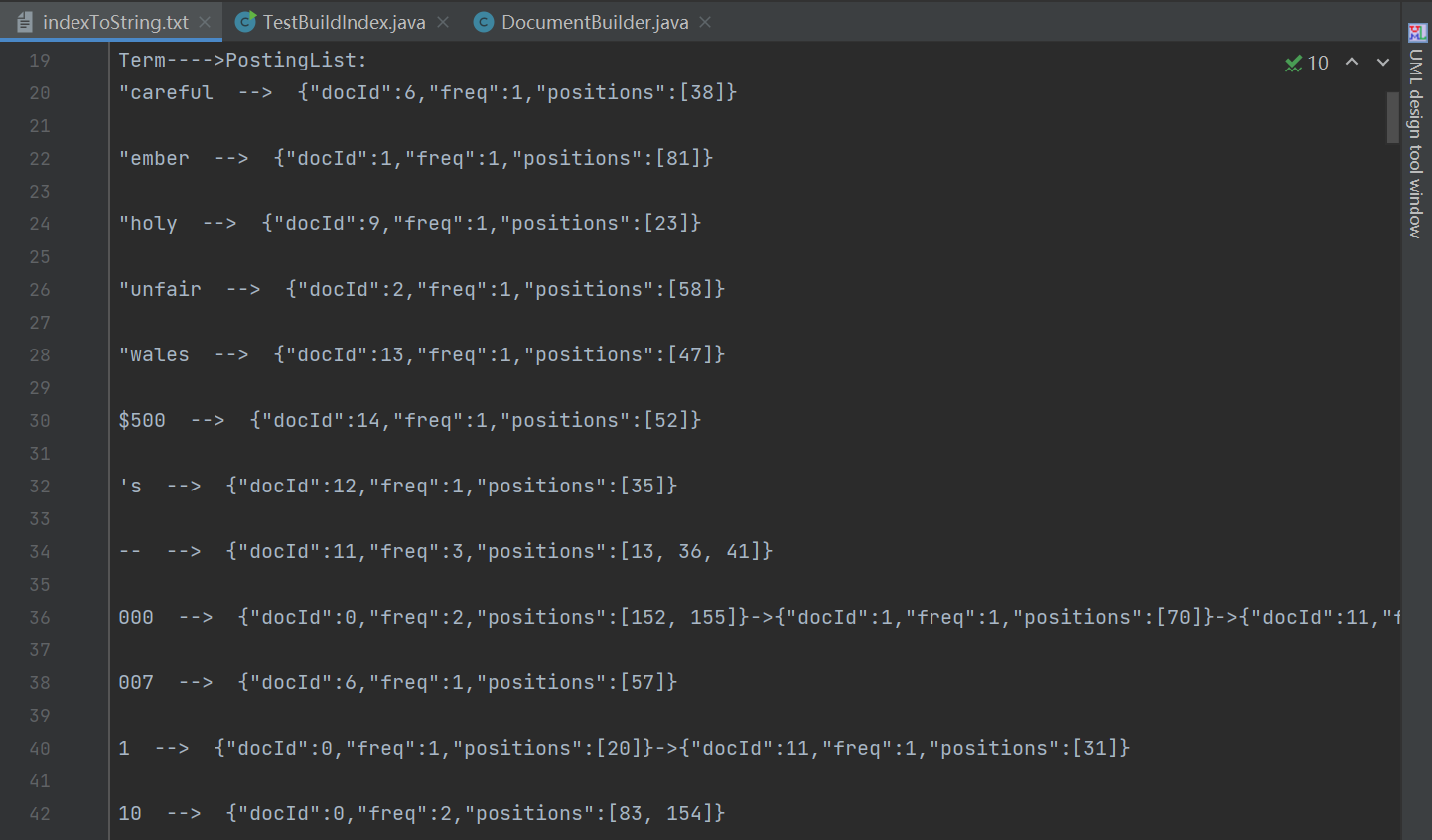


图4.2.3 用真实测试数据集构建索引结果(不带任何过滤器)

1. 采用真实测试数据集，加上三个过滤器，构建索引的结果示意如图4.2.4所示。

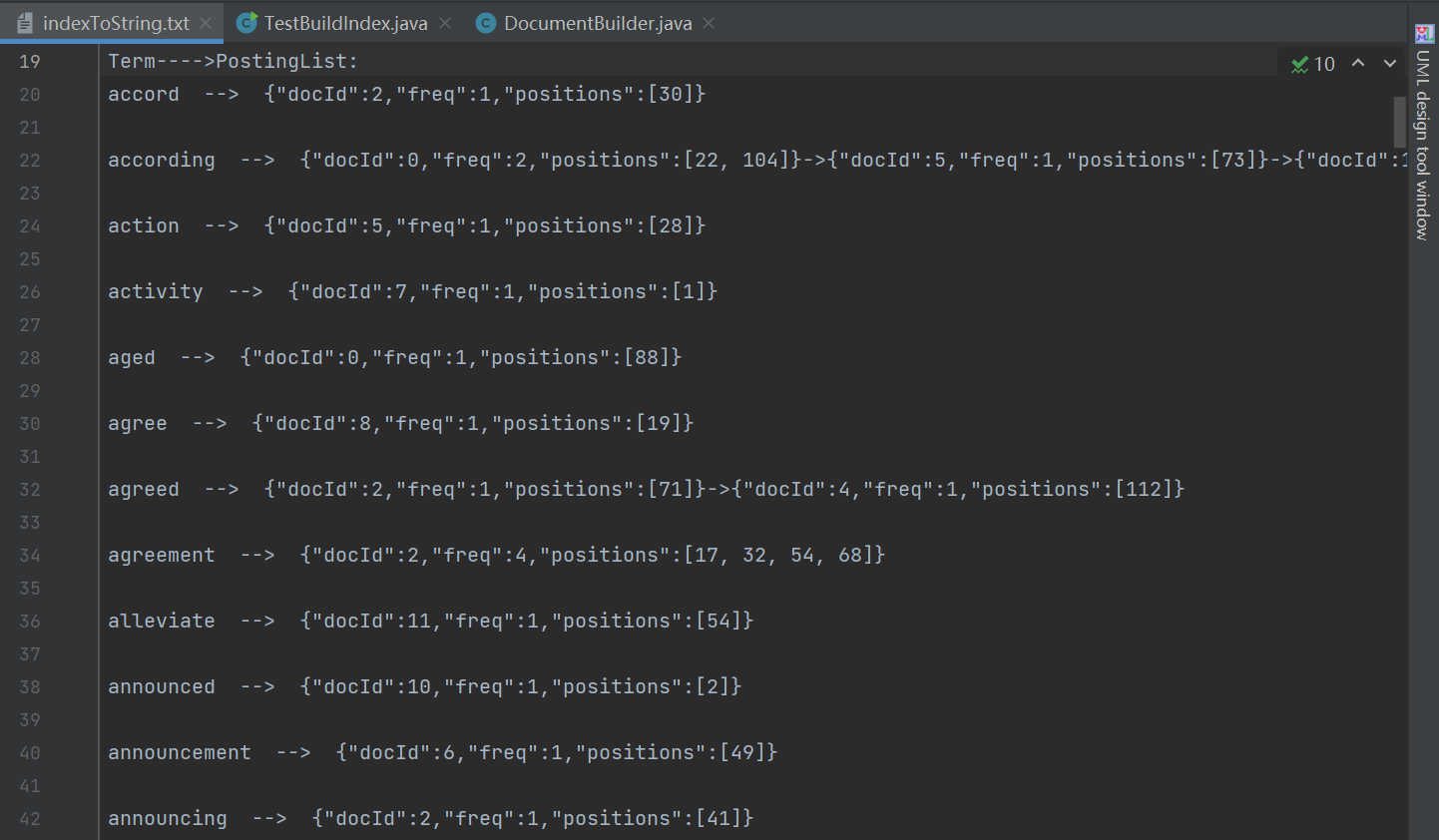


图4.2.4 用真实测试数据集构建索引结果(带三种过滤器)

1. 基于图 4 所示的索引构建结果， 采用 coronavirus 作为检索词的检索结果如图4.2.5所示。

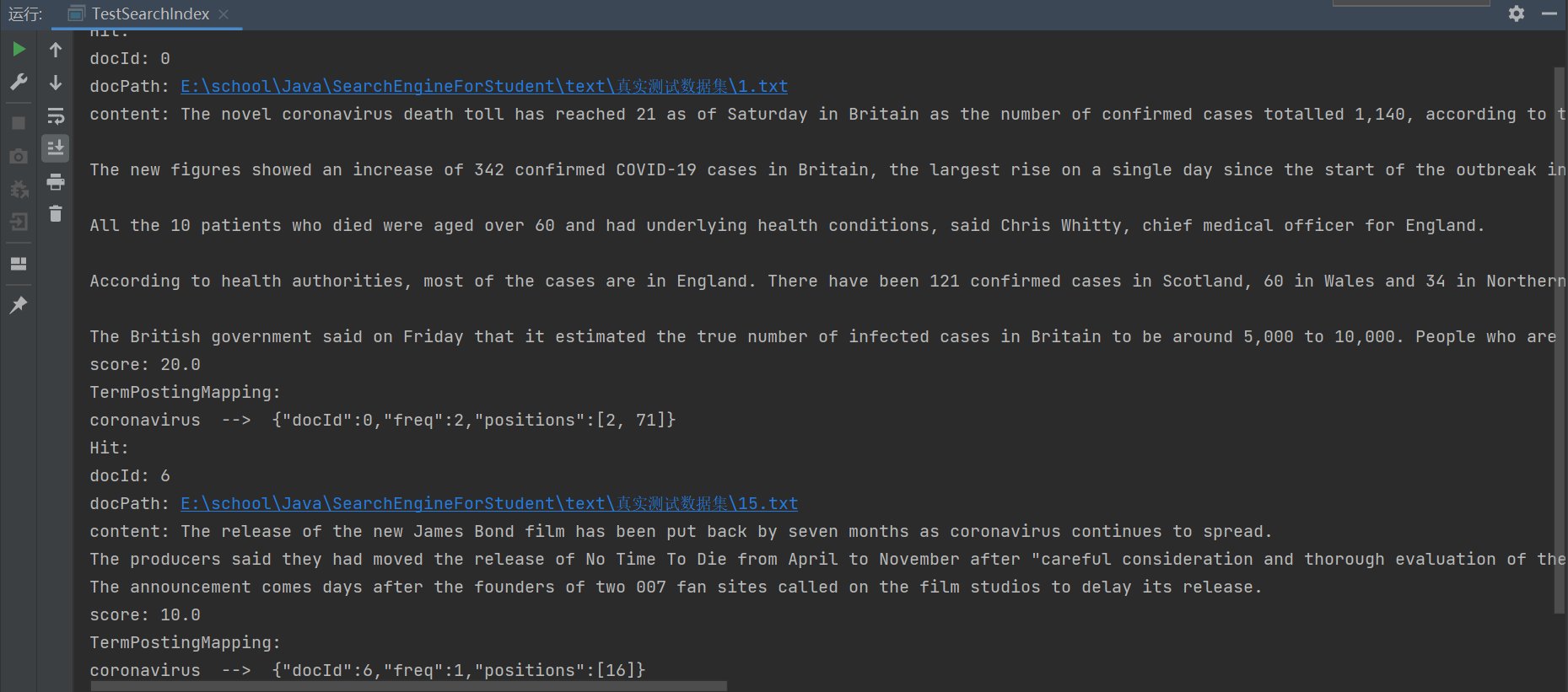


图4.2.5 采用 coronavirus 作为检索词的检索结果

由图可以看到，采用 coronavirus 作为检索词命中了二个结果： 第一个结果的得分为 20，因此排在前面；第二个结果的得分为 10，因此排在第二位。 同时检索结果也显示了命中的文档的原文内容。

1. 基于图 4 所示的索引构建结果， 采用 coronavirus 和wales作为检索词的检索结果如图4.2.6 所示。

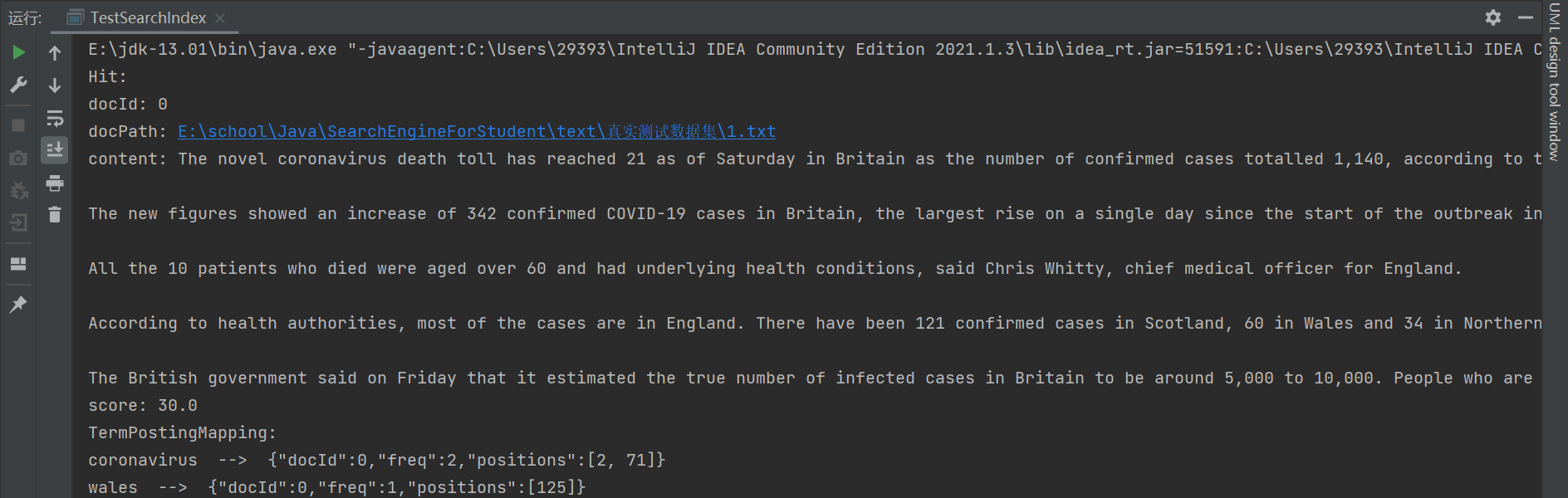


图4.2.6 采用coronavirus和wales作为检索词的检索结果

由图可以看出，对于两个检索词，程序也可以正确运行。

# 5 特点与不足

## 5.1 技术特点

本次实验实现了所有需求，可以实现一个词的检索，也可以实现两个词的检索；并可以通过序列化、反序列化去读取文件。

通过实验，了解了实际项目编写中一些常见的设计模式的思想，并将其运用到本实验当中，如装饰者模式等；在对map等数据类型进行遍历时，尝试使用了lambda表达式，即forEach进行遍历，在使用lambda表达式的时候，好处在于不用写key和value的类型，编译器会自动帮我们断言。

## 5.2 不足与建议

所花时间过长，java的一些api函数还不够熟悉，编程过程中多次查阅资料，以后还需多加进行实践练习。

在实验过程中有些算法的代码不够精简，没有考虑到算法的时间空间复杂度。而且在实现具体类的过程中，只是一味地跟着老师预先设置好的抽象类进行实现，没有很多自己的思考。

个人的建议是希望可以灵活变动一部分设计思路，留给学生更多自由发挥的空间，但是这样就很难实现自由测试，可能会加大实验检测的时间成本。

# 6 过程和体会

## 6.1 遇到的问题和解决方法

在实验过程中，由于对List的equals方法不熟悉，在编写Posting类的equals方法时，在判断positions列表是否相等时，直接调用了ArrayList的equals方法。这一问题在后来自动测试程序的检测中被检出。通过查阅资料知，ArrayList的equals方法除了比较内容是否相等外，还需要顺序也一致才会返回true。故，修改成，比较positions的size相等，且positions.containsAll(p.positions)为真时相等，便可以成功通过自动测试。

在实现termTupleScanner以及termTupleFilter的时候也遇到了困难。由于没有了解过对数据流的处理，对流没有一种具体的概念，也没有了解过装饰者模式的设计模式，看到PPT上的设计需求时有很大的阅读困难。经过仔细研读ppt以及到网上查阅相关资料，终于理清了装饰者模式的设计思想，最终完成了相关模块的编写。

在进行序列化以及反序列化操作时，一开始对这些完全没有概念，通过老师的帮助文档，让我对序列化反序列化操作有了初步的理解并完成了相关模块的实现。

此外，在运行test.bat脚本时，由于jdk版本、文件路径等问题，自动测试一直无法通过。在反复查阅资料后，终于正确配置了jdk版本和环境。此次实验也让我对Java环境的配置更加熟练。

## 6.2 课程设计的体会

此次实验是第一次独立完成较大的java综合性项目，首先对这个题目的理解有一定难度，在反反复复地多次阅读文档后，对所实现的东西逐渐清晰。刚开始对某些设计模式不太理解，后来老师发了相关设计模式的文档后，经过阅读对设计模式的理解更加透彻。编程过程中对某些Java API不够熟悉，多次进行了资料查阅。还因此犯了一些小错误。

此次实验让我深刻体验到了什么是所谓的面向对象编程，其主要思想就是把一个庞大的系统抽离成一个个具体的模块，每个模块中有自己的属性以及方法，最后再将这些模块整合到这个系统中去。不同于之前的C语言实验或者是数据结构实验，此次实验培养了学生们写真实项目代码的能力，虽然离代码真正落地上线还有很长一段距离，但是通过这次试验让学生们了解了许多知识如常见的设计模式，序列化以及反序列化操作。学生通过完善每一个类最终完成一个项目，会有逐步闯关的条理感和成就感。

# 7 源码和说明

## 7.1 文件清单及功能说明

1. 实验报告文档CS2011\_U202011675\_徐锦慧

2. 自动测试程序压缩包Experiment1Test.zip

3. 整个项目的工程文件压缩包SearchEngineForStudent.zip

下图是工程SearchEngineForStudent的文件清单。其中src下包含了本次实验的源代码，bin为生成的class字节码，index文件夹下包含了倒排索引的index.dat二进制文件和indexToString.txt文本文件，text下包含了测试集。

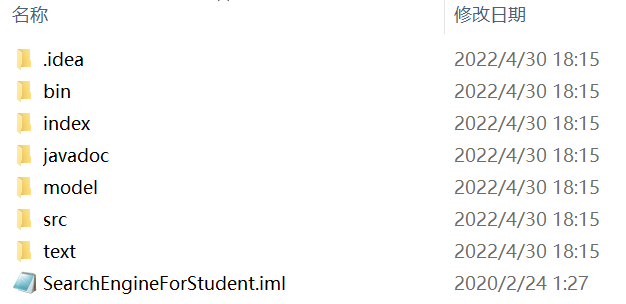


图7.1.1 SearchEngineForStudent文件清单

## 7.2 用户使用说明

解压项目工程文件压缩包，通过IDEA打开本工程文件，通过运行run包下的application的main函数即可运行本函数。

解压自动测试程序压缩包在命令行中运行test.bat即可观察测试结果。

## 源代码

1. **Document.java**

package hust.cs.javacourse.search.index.impl;

import hust.cs.javacourse.search.index.AbstractDocument;

import hust.cs.javacourse.search.index.AbstractTermTuple;

import java.util.List;

/\*\*

\* AbstractDocument的具体实现类.

\*/

public class Document extends AbstractDocument {

/\*\*

\* 缺省构造函数

\*/

public Document() {

super();

}

/\*\*

\* 构造函数

\*

\* @param docId ：文档id

\* @param docPath ：文档绝对路径

\*/

public Document(int docId, String docPath) {

super(docId, docPath);

}

public Document(int docId, String docPath, List<AbstractTermTuple> tuples) {

super(docId, docPath, tuples);

}

/\*\*

\* 获得文档id

\*

\* @return ：文档id

\*/

@Override

public int getDocId() {

return this.docId;

}

/\*\*

\* 设置文档id

\*

\* @param docId ：文档id

\*/

@Override

public void setDocId(int docId) {

this.docId = docId;

}

/\*\*

\* 获得文档绝对路径

\*

\* @return ：文档绝对路径

\*/

@Override

public String getDocPath() {

return this.docPath;

}

/\*\*

\* 设置文档绝对路径

\*

\* @param docPath ：文档绝对路径

\*/

@Override

public void setDocPath(String docPath) {

this.docPath = docPath;

}

/\*\*

\* 获得文档包含的三元组列表

\*

\* @return ：文档包含的三元组列表

\*/

@Override

public List<AbstractTermTuple> getTuples() {

return this.tuples;

}

/\*\*

\* 向文档对象里添加三元组, 要求不能有内容重复的三元组

\*

\* @param tuple ：要添加的三元组

\*/

@Override

public void addTuple(AbstractTermTuple tuple) {

if(!this.tuples.contains(tuple)){

this.tuples.add(tuple);

}

}

/\*\*

\* 判断是否包含指定的三元组

\*

\* @param tuple ： 指定的三元组

\* @return ： 如果包含指定的三元组，返回true;否则返回false

\*/

@Override

public boolean contains(AbstractTermTuple tuple) {

return this.tuples.contains(tuple);

}

/\*\*

\* 获得指定下标位置的三元组

\*

\* @param index ：指定下标位置

\* @return：三元组

\*/

@Override

public AbstractTermTuple getTuple(int index) {

if(index >= 0 && index < this.tuples.size())

return this.tuples.get(index);

else

return null;

}

/\*\*

\* 返回文档对象包含的三元组的个数

\*

\* @return ：文档对象包含的三元组的个数

\*/

@Override

public int getTupleSize() {

return this.tuples.size();

}

/\*\*

\* 获得Document的字符串表示

\*

\* @return ： Document的字符串表示

\*/

@Override

public String toString() {

StringBuilder str = new StringBuilder();

str.append("{\"docId\":" + this.docId +

",\"docPath\":" + this.docPath);

for(AbstractTermTuple tuple : this.tuples){

str.append(",").append(tuple.toString());

}

str.append("}");

return str.toString();

}

}

1. **DocumentBuilder.java**

package hust.cs.javacourse.search.index.impl;

import hust.cs.javacourse.search.index.AbstractDocument;

import hust.cs.javacourse.search.index.AbstractDocumentBuilder;

import hust.cs.javacourse.search.index.AbstractTermTuple;

import hust.cs.javacourse.search.parse.AbstractTermTupleStream;

import hust.cs.javacourse.search.parse.impl.LengthTermTupleFilter;

import hust.cs.javacourse.search.parse.impl.PatternTermTupleFilter;

import hust.cs.javacourse.search.parse.impl.StopWordTermTupleFilter;

import hust.cs.javacourse.search.parse.impl.TermTupleScanner;

import java.io.\*;

import java.util.ArrayList;

import java.util.List;

/\*\*

\* AbstractDocumentBuilder的具体实现类

\*/

public class DocumentBuilder extends AbstractDocumentBuilder {

/\*\*

\* <pre>

\* 由解析文本文档得到的TermTupleStream,构造Document对象.

\* @param docId : 文档id

\* @param docPath : 文档绝对路径

\* @param termTupleStream : 文档对应的TermTupleStream

\* @return ：Document对象

\* </pre>

\*/

@Override

public AbstractDocument build(int docId, String docPath, AbstractTermTupleStream termTupleStream) {

List<AbstractTermTuple> tuples = new ArrayList<>();

AbstractTermTuple temp = termTupleStream.next();

while (temp!=null){

tuples.add(temp);

temp = termTupleStream.next();

}

return new Document(docId,docPath,tuples);

}

/\*\*

\* <pre>

\* 由给定的File,构造Document对象.

\* 该方法利用输入参数file构造出AbstractTermTupleStream子类对象后,内部调用

\* AbstractDocument build(int docId, String docPath, AbstractTermTupleStream termTupleStream)

\* @param docId : 文档id

\* @param docPath : 文档绝对路径

\* @param file : 文档对应File对象

\* @return : Document对象

\* </pre>

\*/

@Override

public AbstractDocument build(int docId, String docPath, File file) {

AbstractDocument newDocument = null;

AbstractTermTupleStream tupleStream = null;

try {

BufferedReader reader = new BufferedReader(new InputStreamReader(new FileInputStream(file)));

tupleStream = new StopWordTermTupleFilter(

new PatternTermTupleFilter(

new LengthTermTupleFilter(

new TermTupleScanner(reader))));

//不过滤

//tupleStream = new TermTupleScanner(reader);

newDocument = build(docId,docPath,tupleStream);

} catch (FileNotFoundException e) {

e.printStackTrace();

}

finally {

tupleStream.close();

}

return newDocument;

}

}

1. **Index.java**

package hust.cs.javacourse.search.index.impl;

import hust.cs.javacourse.search.index.\*;

import java.io.\*;

import java.util.ArrayList;

import java.util.List;

import java.util.Map;

import java.util.Set;

/\*\*

\* AbstractIndex的具体实现类

\*/

public class Index extends AbstractIndex {

/\*\*

\* 返回索引的字符串表示

\*

\* @return 索引的字符串表示

\*/

@Override

public String toString() {

StringBuilder str = new StringBuilder();

//显示字典

str.append("dictionary: \n");

str.append(this.getDictionary().toString());

str.append("\n");

//显示文档id和文档路径的映射关系

str.append("docId---->docPath Mapping: " + "\n");

for (Map.Entry<Integer, String> entry : docIdToDocPathMapping.entrySet()) {

str.append(entry.getKey()).append(" ----> ").append(entry.getValue()).append("\n");

}

//显示倒排索引结构

str.append("Term---->PostingList: ");

for (Map.Entry<AbstractTerm, AbstractPostingList> entry : termToPostingListMapping.entrySet()) {

str.append("\n").append(entry.getKey().toString()).append(" --> ");

str.append(entry.getValue().toString());

}

return str.toString();

}

/\*\*

\* 添加文档到索引，更新索引内部的HashMap

\*

\* @param document ：文档的AbstractDocument子类型表示

\*/

@Override

public void addDocument(AbstractDocument document) {

this.docIdToDocPathMapping.put(document.getDocId(),document.getDocPath());

AbstractTermTuple termTuple;

AbstractTerm term;

AbstractPostingList postingList;

AbstractPosting posting;

for(int i = 0; i < document.getTupleSize(); i++){

termTuple = document.getTuple(i);

term = termTuple.term;

//如果当前term已被加入Map

if(this.termToPostingListMapping.containsKey(term)){

postingList = this.termToPostingListMapping.get(term);

int pos = postingList.indexOf(document.getDocId());

//当前单词是第一次加入PostingList

if(pos == -1){

List<Integer> positions = new ArrayList<>();

positions.add(termTuple.curPos);

posting = new Posting(document.getDocId(),1,positions);

postingList.add(posting);

}

else{

posting = postingList.get(pos);

posting.setFreq(posting.getFreq()+1);

posting.getPositions().add(termTuple.curPos);

}

}

//当前term未被加入map

else{

List<Integer> positions = new ArrayList<>();

positions.add(termTuple.curPos);

posting = new Posting(document.getDocId(),1,positions);

postingList = new PostingList();

postingList.add(posting);

this.termToPostingListMapping.put(term,postingList);

}

}

}

/\*\*

\* <pre>

\* 从索引文件里加载已经构建好的索引.内部调用FileSerializable接口方法readObject即可

\* @param file ：索引文件

\* </pre>

\*/

@Override

public void load(File file) {

try {

FileInputStream inStream = new FileInputStream(file);

ObjectInputStream objectInputStream = new ObjectInputStream(inStream);

this.readObject(objectInputStream);

inStream.close();

} catch (IOException e) {

e.printStackTrace();

}

}

/\*\*

\* <pre>

\* 将在内存里构建好的索引写入到文件. 内部调用FileSerializable接口方法writeObject即可

\* @param file ：写入的目标索引文件

\* </pre>

\*/

@Override

public void save(File file) {

try {

FileOutputStream outStream = new FileOutputStream(file);

ObjectOutputStream objectOutputStream = new ObjectOutputStream(outStream);

this.writeObject(objectOutputStream);

outStream.close();

} catch (IOException e) {

e.printStackTrace();

}

}

/\*\*

\* 返回指定单词的PostingList

\*

\* @param term : 指定的单词

\* @return ：指定单词的PostingList;如果索引字典没有该单词，则返回null

\*/

@Override

public AbstractPostingList search(AbstractTerm term) {

return this.termToPostingListMapping.get(term);

}

/\*\*

\* 返回索引的字典.字典为索引里所有单词的并集

\*

\* @return ：索引中Term列表

\*/

@Override

public Set<AbstractTerm> getDictionary() {

return this.termToPostingListMapping.keySet();

}

/\*\*

\* <pre>

\* 对索引进行优化，包括：

\* 对索引里每个单词的PostingList按docId从小到大排序

\* 同时对每个Posting里的positions从小到大排序

\* 在内存中把索引构建完后执行该方法

\* </pre>

\*/

@Override

public void optimize() {

for (Map.Entry<AbstractTerm, AbstractPostingList> entry : termToPostingListMapping.entrySet()){

AbstractPostingList postingList = entry.getValue();

postingList.sort();

for( int i = 0; i < postingList.size(); i++){

postingList.get(i).sort();

}

}

}

/\*\*

\* 根据docId获得对应文档的完全路径名

\*

\* @param docId ：文档id

\* @return : 对应文档的完全路径名

\*/

@Override

public String getDocName(int docId) {

return this.docIdToDocPathMapping.get(docId);

}

/\*\*

\* 写到二进制文件

\*

\* @param out :输出流对象

\*/

@Override

public void writeObject(ObjectOutputStream out) {

try {

out.writeObject(this);

} catch (IOException e) {

e.printStackTrace();

}

}

/\*\*

\* 从二进制文件读

\*

\* @param in ：输入流对象

\*/

@Override

public void readObject(ObjectInputStream in) {

try {

Index index = ((Index)in.readObject());

this.termToPostingListMapping = index.termToPostingListMapping;

this.docIdToDocPathMapping = index.docIdToDocPathMapping;

} catch (IOException | ClassNotFoundException e) {

e.printStackTrace();

}

}

}

1. **IndexBuilder.java**

package hust.cs.javacourse.search.index.impl;

import hust.cs.javacourse.search.index.AbstractDocument;

import hust.cs.javacourse.search.index.AbstractDocumentBuilder;

import hust.cs.javacourse.search.index.AbstractIndex;

import hust.cs.javacourse.search.index.AbstractIndexBuilder;

import java.io.File;

/\*\*

\* AbstractIndexBuilder的具体实现类.

\*/

public class IndexBuilder extends AbstractIndexBuilder {

public IndexBuilder(AbstractDocumentBuilder docBuilder) {

super(docBuilder);

}

/\*\*

\* <pre>

\* 构建指定目录下的所有文本文件的倒排索引.

\* 需要遍历和解析目录下的每个文本文件, 得到对应的Document对象，再依次加入到索引，并将索引保存到文件.

\* @param rootDirectory ：指定目录

\* @return ：构建好的索引

\* </pre>

\*/

@Override

public AbstractIndex buildIndex(String rootDirectory) {

File path = new File(rootDirectory);

Index index = new Index();

if (path.isDirectory()){//是否为文件夹

File[] files = path.listFiles();

if (files != null){//目录下有文件

for (File file : files){

if (file.isFile()){//若有子目录跳过子目录

AbstractDocument document = this.docBuilder.build(this.docId,file.getPath(),file);

index.addDocument(document);

this.docId++;

}

}

return index;

}

}

return null;

}

}

1. **Posting.java**

package hust.cs.javacourse.search.index.impl;

import hust.cs.javacourse.search.index.AbstractPosting;

import java.io.IOException;

import java.io.ObjectInputStream;

import java.io.ObjectOutputStream;

import java.util.Collections;

import java.util.List;

/\*\*

\* AbstractPosting的具体实现类.

\*/

public class Posting extends AbstractPosting {

/\*\*

\* 缺省构造函数

\*/

public Posting() {

super();

}

/\*\*

\* 构造函数

\* @param docId ：包含单词的文档id

\* @param freq ：单词在文档里出现的次数

\* @param positions ：单词在文档里出现的位置

\*/

public Posting(int docId, int freq, List<Integer> positions) {

super(docId, freq, positions);

}

/\*\*

\* 判断二个Posting内容是否相同

\* @param obj ：要比较的另外一个Posting

\* @return 如果内容相等返回true，否则返回false

\*/

@Override

public boolean equals(Object obj) {

if (obj instanceof Posting){

Posting p = (Posting)obj;

return this.docId == p.docId &&

this.freq == p.freq &&

this.positions.size() == p.positions.size() &&

this.positions.containsAll(p.positions);

}

return false;

}

/\*\*

\* 返回Posting的字符串表示

\*

\* @return 字符串

\*/

@Override

public String toString() {

return "{\"docId\":" + this.docId +

",\"freq\":" + this.freq +

",\"positions\":" + this.positions.toString() + "}";

}

/\*\*

\* 返回包含单词的文档id

\*

\* @return ：文档id

\*/

@Override

public int getDocId() {

return this.docId;

}

/\*\*

\* 设置包含单词的文档id

\*

\* @param docId ：包含单词的文档id

\*/

@Override

public void setDocId(int docId) {

this.docId = docId;

}

/\*\*

\* 返回单词在文档里出现的次数

\*

\* @return ：出现次数

\*/

@Override

public int getFreq() {

return this.freq;

}

/\*\*

\* 设置单词在文档里出现的次数

\*

\* @param freq :单词在文档里出现的次数

\*/

@Override

public void setFreq(int freq) {

this.freq = freq;

}

/\*\*

\* 返回单词在文档里出现的位置列表

\*

\* @return ：位置列表

\*/

@Override

public List<Integer> getPositions() {

return this.positions;

}

/\*\*

\* 设置单词在文档里出现的位置列表

\*

\* @param positions ：单词在文档里出现的位置列表

\*/

@Override

public void setPositions(List<Integer> positions) {

this.positions = positions;

}

/\*\*

\* 比较二个Posting对象的大小（根据docId）

\*

\* @param o ： 另一个Posting对象

\* @return ：二个Posting对象的docId的差值

\*/

@Override

public int compareTo(AbstractPosting o) {

if(o instanceof Posting){

Posting p = (Posting)o;

return this.docId - p.docId;

}

throw new RuntimeException("比较类型不匹配");

}

/\*\*

\* 对内部positions从小到大排序

\*/

@Override

public void sort() {

Collections.sort(this.positions);

}

/\*\*

\* 写到二进制文件

\*

\* @param out :输出流对象

\*/

@Override

public void writeObject(ObjectOutputStream out) {

try {

out.writeObject(this);

} catch (IOException e) {

e.printStackTrace();

}

}

/\*\*

\* 从二进制文件读

\*

\* @param in ：输入流对象

\*/

@Override

public void readObject(ObjectInputStream in) {

try {

Posting p = (Posting)in.readObject();

this.docId = p.docId;

this.freq = p.freq;

this.positions = p.positions;

} catch (IOException | ClassNotFoundException e) {

e.printStackTrace();

}

}

}

1. **PostingList.java**

package hust.cs.javacourse.search.index.impl;

import hust.cs.javacourse.search.index.AbstractPosting;

import hust.cs.javacourse.search.index.AbstractPostingList;

import java.io.IOException;

import java.io.ObjectInputStream;

import java.io.ObjectOutputStream;

import java.util.Collections;

import java.util.List;

/\*\*

\* AbstractPostingList的具体实现类.

\*/

public class PostingList extends AbstractPostingList {

/\*\*

\* 添加Posting,要求不能有内容重复的posting

\*

\* @param posting ：Posting对象

\*/

@Override

public void add(AbstractPosting posting) {

if (!this.list.contains(posting)){

this.list.add(posting);

}

}

/\*\*

\* 获得PosingList的字符串表示

\*

\* @return ： PosingList的字符串表示

\*/

@Override

public String toString() {

StringBuilder str = new StringBuilder();

if (this.list == null)

return null;

boolean flag = false;

for (AbstractPosting p : this.list){

if (flag){

str.append("->");

}

flag = true;

str.append(p.toString());

}

str.append("\n");

return str.toString();

}

/\*\*

\* 添加Posting列表,,要求不能有内容重复的posting

\*

\* @param postings ：Posting列表

\*/

@Override

public void add(List<AbstractPosting> postings) {

AbstractPosting temp;

for (AbstractPosting p : postings){

temp = p;

if (!this.list.contains(temp)){

this.list.add(temp);

}

}

}

/\*\*

\* 返回指定下标位置的Posting

\*

\* @param index ：下标

\* @return： 指定下标位置的Posting

\*/

@Override

public AbstractPosting get(int index) {

if(index >= 0 && index < this.list.size())

return this.list.get(index);

else

return null;

}

/\*\*

\* 返回指定Posting对象的下标

\*

\* @param posting ：指定的Posting对象

\* @return ：如果找到返回对应下标；否则返回-1

\*/

@Override

public int indexOf(AbstractPosting posting) {

for (int i=0;i<this.list.size();i++){

if (this.list.get(i).equals(posting))

return i;

}

return -1;

}

/\*\*

\* 返回指定文档id的Posting对象的下标

\*

\* @param docId ：文档id

\* @return ：如果找到返回对应下标；否则返回-1

\*/

@Override

public int indexOf(int docId) {

for (int i=0;i<this.list.size();i++){

if (this.list.get(i).getDocId() == docId)

return i;

}

return -1;

}

/\*\*

\* 是否包含指定Posting对象

\*

\* @param posting ： 指定的Posting对象

\* @return : 如果包含返回true，否则返回false

\*/

@Override

public boolean contains(AbstractPosting posting) {

return this.list.contains(posting);

}

/\*\*

\* 删除指定下标的Posting对象

\*

\* @param index ：指定的下标

\*/

@Override

public void remove(int index) {

if(index >= 0 && index < this.list.size())

this.list.remove(index);

}

/\*\*

\* 删除指定的Posting对象

\*

\* @param posting ：定的Posting对象

\*/

@Override

public void remove(AbstractPosting posting) {

this.list.remove(posting);

}

/\*\*

\* 返回PostingList的大小，即包含的Posting的个数

\*

\* @return ：PostingList的大小

\*/

@Override

public int size() {

return this.list.size();

}

/\*\*

\* 清除PostingList

\*/

@Override

public void clear() {

this.list.clear();

}

/\*\*

\* PostingList是否为空

\*

\* @return 为空返回true;否则返回false

\*/

@Override

public boolean isEmpty() {

return this.list.isEmpty();

}

/\*\*

\* 根据文档id的大小对PostingList进行从小到大的排序

\*/

@Override

public void sort() {

Collections.sort(this.list);

}

/\*\*

\* 写到二进制文件

\*

\* @param out :输出流对象

\*/

@Override

public void writeObject(ObjectOutputStream out) {

try {

out.writeObject(this);

} catch (IOException e) {

e.printStackTrace();

}

}

/\*\*

\* 从二进制文件读

\*

\* @param in ：输入流对象

\*/

@Override

public void readObject(ObjectInputStream in) {

try {

this.list = ((PostingList)in.readObject()).list;

} catch (IOException | ClassNotFoundException e) {

e.printStackTrace();

}

}

}

1. **Term.java**

package hust.cs.javacourse.search.index.impl;

import hust.cs.javacourse.search.index.AbstractTerm;

import java.io.ObjectInputStream;

import java.io.ObjectOutputStream;

import java.io.IOException;

/\*\*

\* AbstractTerm的具体实现类.

\*/

public class Term extends AbstractTerm {

/\*\*

\* 缺省构造函数

\*/

public Term() {

super();

}

/\*\*

\* 构造函数

\* @param content ：Term内容

\*/

public Term(String content) {

super(content);

}

/\*\*

\* 判断二个Term内容是否相同

\* @param obj ：要比较的另外一个Term

\* @return 如果内容相等返回true，否则返回false

\*/

@Override

public boolean equals(Object obj) {

if (obj instanceof Term) {

return content.equals(((Term) obj).content);

}

return false;

}

/\*\*

\* 返回Term的字符串表示

\* @return 字符串

\*/

@Override

public String toString() {

return this.content;

}

/\*\*

\* 返回Term内容

\* @return Term内容

\*/

@Override

public String getContent() {

return this.content;

}

/\*\*

\* 设置Term内容

\* @param content：Term的内容

\*/

@Override

public void setContent(String content) {

this.content = content;

}

/\*\*

\* 比较二个Term大小（按字典序）

\* @param o： 要比较的Term对象

\* @return ： 返回二个Term对象的字典序差值

\*/

@Override

public int compareTo(AbstractTerm o) {

if(o instanceof Term){

Term t = (Term)o;

return this.content.compareTo(t.content);

}

throw new RuntimeException("compareTo的类型不匹配");

}

/\*\*

\* 写到二进制文件

\*

\* @param out :输出流对象

\*/

@Override

public void writeObject(ObjectOutputStream out) {

try {

out.writeObject(this);

} catch (IOException e) {

e.printStackTrace();

}

}

/\*\*

\* 从二进制文件读

\*

\* @param in ：输入流对象

\*/

@Override

public void readObject(ObjectInputStream in) {

try {

this.content = ((Term)in.readObject()).content;

} catch (IOException | ClassNotFoundException e) {

e.printStackTrace();

}

}

}

1. **TermTuple.java**

package hust.cs.javacourse.search.index.impl;

import hust.cs.javacourse.search.index.AbstractTermTuple;

/\*\*

\* AbstractTermTuple的具体实现类.

\*/

public class TermTuple extends AbstractTermTuple {

/\*\*

\* 判断二个三元组内容是否相同

\*

\* @param obj ：要比较的另外一个三元组

\* @return 如果内容相等（三个属性内容都相等）返回true，否则返回false

\*/

@Override

public boolean equals(Object obj) {

if(obj instanceof TermTuple){

TermTuple t = (TermTuple)obj;

return (this.curPos == t.curPos) &&

(this.freq == t.freq) &&

(this.term.equals(t.term));

}

return false;

}

/\*\*

\* 获得三元组的字符串表示

\*

\* @return ： 三元组的字符串表示

\*/

@Override

public String toString() {

return "<term:\"" + this.term + "\",freq:" + this.freq + ",curPos:" + this.curPos + ">";

}

}

1. **LengthTermTupleFilter.java**

package hust.cs.javacourse.search.parse.impl;

import hust.cs.javacourse.search.index.AbstractTermTuple;

import hust.cs.javacourse.search.parse.AbstractTermTupleFilter;

import hust.cs.javacourse.search.parse.AbstractTermTupleStream;

import hust.cs.javacourse.search.util.Config;

/\*\*

\* LengthTermTupleFilter是抽象类AbstractTermTupleFilter的子类.

\* 将长度小于3或长度大于20的单词过滤掉

\*/

public class LengthTermTupleFilter extends AbstractTermTupleFilter {

/\*\*

\* 构造函数

\*

\* @param input ：Filter的输入，类型为AbstractTermTupleStream

\*/

public LengthTermTupleFilter(AbstractTermTupleStream input) {

super(input);

}

/\*\*

\* 获得下一个三元组

\*

\* @return: 下一个三元组；如果到了流的末尾，返回null

\*/

@Override

public AbstractTermTuple next() {

AbstractTermTuple temp = input.next();

int len;

while (temp!=null){

len = temp.term.getContent().length();

if(len <= Config.TERM\_FILTER\_MAXLENGTH && len >= Config.TERM\_FILTER\_MINLENGTH){

return temp;

}

else{

temp = input.next();

}

}

return null;

}

}

1. **PatternTermTupleFilter.java**

package hust.cs.javacourse.search.parse.impl;

import hust.cs.javacourse.search.index.AbstractTermTuple;

import hust.cs.javacourse.search.parse.AbstractTermTupleFilter;

import hust.cs.javacourse.search.parse.AbstractTermTupleStream;

import hust.cs.javacourse.search.util.Config;

/\*\*

\* PatternTermTupleFilter是抽象类AbstractTermTupleFilter的子类.

\* 选择英文字母组成的Term对应的三元组

\*/

public class PatternTermTupleFilter extends AbstractTermTupleFilter {

/\*\*

\* 构造函数

\*

\* @param input ：Filter的输入，类型为AbstractTermTupleStream

\*/

public PatternTermTupleFilter(AbstractTermTupleStream input) {

super(input);

}

/\*\*

\* 获得下一个三元组

\*

\* @return: 下一个三元组；如果到了流的末尾，返回null

\*/

@Override

public AbstractTermTuple next() {

AbstractTermTuple temp = input.next();

while (temp!=null){

if(temp.term.getContent().matches(Config.TERM\_FILTER\_PATTERN)){

return temp;

}

else{

temp = input.next();

}

}

return null;

}

}

1. **StopWordTermTupleFilter.java**

package hust.cs.javacourse.search.parse.impl;

import hust.cs.javacourse.search.index.AbstractTermTuple;

import hust.cs.javacourse.search.parse.AbstractTermTupleFilter;

import hust.cs.javacourse.search.parse.AbstractTermTupleStream;

import hust.cs.javacourse.search.util.StopWords;

import java.util.Arrays;

/\*\*

\* StopWordTermTupleFilter是抽象类AbstractTermTupleFilter的子类.

\* 用于过滤含StopWords类中定义的停用词的三元组

\*/

public class StopWordTermTupleFilter extends AbstractTermTupleFilter {

/\*\*

\* 构造函数

\*

\* @param input ：Filter的输入，类型为AbstractTermTupleStream

\*/

public StopWordTermTupleFilter(AbstractTermTupleStream input) {

super(input);

}

/\*\*

\* 获得下一个三元组

\*

\* @return: 下一个三元组；如果到了流的末尾，返回null

\*/

@Override

public AbstractTermTuple next() {

AbstractTermTuple temp = input.next();

while (temp!=null){

if(!Arrays.asList(StopWords.STOP\_WORDS).contains(temp.term.getContent())){

return temp;

}

else{

temp = input.next();

}

}

return null;

}

}

1. **TermTupleScanner.java**

package hust.cs.javacourse.search.parse.impl;

import hust.cs.javacourse.search.index.AbstractTermTuple;

import hust.cs.javacourse.search.index.impl.Term;

import hust.cs.javacourse.search.index.impl.TermTuple;

import hust.cs.javacourse.search.parse.AbstractTermTupleScanner;

import hust.cs.javacourse.search.util.Config;

import hust.cs.javacourse.search.util.StringSplitter;

import java.io.BufferedReader;

import java.io.IOException;

import java.util.ArrayList;

import java.util.List;

/\*\*

\* TermTupleScanner是抽象类AbstractTermTupleScanner的子类.

\* 读取所输入文档的每一行，并返回解析该行内容后得到的Term三元组

\*/

public class TermTupleScanner extends AbstractTermTupleScanner {

private List<String> words = new ArrayList<>();

private int pos = 0;

/\*\*

\* 缺省构造函数

\*/

public TermTupleScanner() {

super();

}

/\*\*

\* 构造函数

\*

\* @param input ：指定输入流对象，应该关联到一个文本文件

\*/

public TermTupleScanner(BufferedReader input) {

super(input);

}

/\*\*

\* 获得下一个三元组

\*

\* @return: 下一个三元组；如果到了流的末尾，返回null

\*/

@Override

public AbstractTermTuple next() {

AbstractTermTuple newTermTuple = new TermTuple();

StringSplitter splitter = new StringSplitter();

splitter.setSplitRegex(Config.STRING\_SPLITTER\_REGEX);

while (this.words.isEmpty()){

String str = null;

try {

str = input.readLine();

} catch (IOException e) {

e.printStackTrace();

}

if(str==null)

return null;

if(Config.IGNORE\_CASE)

this.words = splitter.splitByRegex(str.toLowerCase());

else

this.words = splitter.splitByRegex(str);

}

newTermTuple.term = new Term(this.words.get(0));

this.words.remove(0);

newTermTuple.curPos = pos;

pos++;

return newTermTuple;

}

}

1. **Hit.java**

package hust.cs.javacourse.search.query.impl;

import hust.cs.javacourse.search.index.AbstractPosting;

import hust.cs.javacourse.search.index.AbstractTerm;

import hust.cs.javacourse.search.query.AbstractHit;

import java.util.Map;

/\*\*

\* AbstractHit的具体实现类

\*/

public class Hit extends AbstractHit {

/\*\*

\* 默认构造函数

\*/

public Hit() {

super();

}

/\*\*

\* 构造函数

\*

\* @param docId : 文档id

\* @param docPath : 文档绝对路径

\*/

public Hit(int docId, String docPath) {

super(docId, docPath);

}

/\*\*

\* 构造函数

\*

\* @param docId ：文档id

\* @param docPath ：文档绝对路径

\* @param termPostingMapping ：命中的三元组列表

\*/

public Hit(int docId, String docPath, Map<AbstractTerm, AbstractPosting> termPostingMapping) {

super(docId, docPath, termPostingMapping);

}

/\*\*

\* 获得文档id

\*

\* @return ： 文档id

\*/

@Override

public int getDocId() {

return this.docId;

}

/\*\*

\* 获得文档绝对路径

\*

\* @return ：文档绝对路径

\*/

@Override

public String getDocPath() {

return this.docPath;

}

/\*\*

\* 获得文档内容

\*

\* @return ： 文档内容

\*/

@Override

public String getContent() {

return this.content;

}

/\*\*

\* 设置文档内容

\*

\* @param content ：文档内容

\*/

@Override

public void setContent(String content) {

this.content = ((content == null) ? "" : content);

}

/\*\*

\* 获得文档得分

\*

\* @return ： 文档得分

\*/

@Override

public double getScore() {

return this.score;

}

/\*\*

\* 设置文档得分

\*

\* @param score ：文档得分

\*/

@Override

public void setScore(double score) {

this.score = score;

}

/\*\*

\* 获得命中的单词和对应的Posting键值对

\*

\* @return ：命中的单词和对应的Posting键值对

\*/

@Override

public Map<AbstractTerm, AbstractPosting> getTermPostingMapping() {

return this.termPostingMapping;

}

/\*\*

\* 获得命中结果的字符串表示, 用于显示搜索结果.

\*

\* @return : 命中结果的字符串表示

\*/

@Override

public String toString() {

StringBuilder str = new StringBuilder();

str.append("Hit: \n");

str.append("docId: ").append(this.docId).append("\n");

str.append("docPath: ").append(this.docPath).append("\n");

str.append("content: ").append(this.content).append("\n");

str.append("score: ").append(-this.score).append("\n");

str.append("TermPostingMapping: ");

for (Map.Entry<AbstractTerm, AbstractPosting> entry : termPostingMapping.entrySet()) {

str.append("\n").append(entry.getKey()).append(" --> ").append(entry.getValue());

}

return str.toString();

}

/\*\*

\* 比较二个命中结果的大小，根据score比较

\*

\* @param o ：要比较的名字结果

\* @return ：二个命中结果得分的差值

\*/

@Override

public int compareTo(AbstractHit o) {

return (int)(this.score-o.getScore());

}

}

1. **IndexSearcher.java**

package hust.cs.javacourse.search.query.impl;

import hust.cs.javacourse.search.index.AbstractPosting;

import hust.cs.javacourse.search.index.AbstractPostingList;

import hust.cs.javacourse.search.index.AbstractTerm;

import hust.cs.javacourse.search.query.AbstractHit;

import hust.cs.javacourse.search.query.AbstractIndexSearcher;

import hust.cs.javacourse.search.query.Sort;

import java.io.File;

import java.util.\*;

/\*\*

\* AbstractIndexSearcher的具体实现类

\*/

public class IndexSearcher extends AbstractIndexSearcher {

/\*\*

\* 从指定索引文件打开索引，加载到index对象里. 一定要先打开索引，才能执行search方法

\*

\* @param indexFile ：指定索引文件

\*/

@Override

public void open(String indexFile) {

this.index.load(new File(indexFile));

}

/\*\*

\* 根据单个检索词进行搜索

\*

\* @param queryTerm ：检索词

\* @param sorter ：排序器

\* @return ：命中结果数组

\*/

@Override

public AbstractHit[] search(AbstractTerm queryTerm, Sort sorter) {

AbstractPostingList postingList = this.index.search(queryTerm);

if(postingList == null)

return null;

AbstractHit[] Hits = new AbstractHit[postingList.size()];

AbstractHit Hit;

AbstractPosting Posting;

for(int i = 0; i < Hits.length; i++){

Posting = postingList.get(i);

Map<AbstractTerm, AbstractPosting> termPostingMapping = new TreeMap<>(); //应该在循环体内实例化

termPostingMapping.put(queryTerm,Posting);

Hit = new Hit(Posting.getDocId(),this.index.getDocName(Posting.getDocId()),termPostingMapping);

sorter.score(Hit);

Hits[i] = Hit;

}

sorter.sort(Arrays.asList(Hits));

return Hits;

}

/\*\*

\* 根据二个检索词进行搜索

\*

\* @param queryTerm1 ：第1个检索词

\* @param queryTerm2 ：第2个检索词

\* @param sorter ： 排序器

\* @param combine ： 多个检索词的逻辑组合方式

\* @return ：命中结果数组

\*/

@Override

public AbstractHit[] search(AbstractTerm queryTerm1, AbstractTerm queryTerm2, Sort sorter, LogicalCombination combine) {

AbstractPostingList postingList1 = this.index.search(queryTerm1);

AbstractPostingList postingList2 = this.index.search(queryTerm2);

List<AbstractHit> hitList = new ArrayList<>();

if(combine == LogicalCombination.AND){

if(postingList1 == null || postingList2 == null){

return null;

}

int i = 0, j = 0;

AbstractPosting posting1,posting2;

while(i < postingList1.size() && j < postingList2.size()){

posting1 = postingList1.get(i);

posting2 = postingList2.get(j);

if(posting1.getDocId() == posting2.getDocId()){

int docId = posting1.getDocId();

Map<AbstractTerm, AbstractPosting> termPostingMapping = new TreeMap<>();

termPostingMapping.put(queryTerm1,posting1);

termPostingMapping.put(queryTerm2,posting2);

AbstractHit hit = new Hit(docId,index.getDocName(docId),termPostingMapping);

sorter.score(hit);

hitList.add(hit);

i++;

j++;

} else if(posting1.getDocId() < posting2.getDocId()){

i++;

} else{

j++;

}

}

if(hitList.isEmpty())

return null;

sorter.sort(hitList);

AbstractHit[] Hits = new AbstractHit[hitList.size()];

hitList.toArray(Hits);

return Hits;

}

else if(combine == LogicalCombination.OR){

if(postingList1 == null){

return search(queryTerm2,sorter);

}

else if(postingList2 == null){

return search(queryTerm1,sorter);

}

else{

int i = 0, j = 0, docId;

AbstractPosting posting1,posting2;

while(i < postingList1.size() && j < postingList2.size()) {

posting1 = postingList1.get(i);

posting2 = postingList2.get(j);

//对于两个关键词都同时命中的情况单独处理

if (posting1.getDocId() == posting2.getDocId()) {

docId = posting1.getDocId();

Map<AbstractTerm, AbstractPosting> termPostingMapping = new TreeMap<>();

termPostingMapping.put(queryTerm1, posting1);

termPostingMapping.put(queryTerm2, posting2);

AbstractHit hit = new Hit(docId, index.getDocName(docId), termPostingMapping);

sorter.score(hit);

hitList.add(hit);

i++;

j++;

}else if(posting1.getDocId() < posting2.getDocId()){

docId = posting1.getDocId();

Map<AbstractTerm, AbstractPosting> termPostingMapping = new TreeMap<>();

termPostingMapping.put(queryTerm1, posting1);

AbstractHit hit = new Hit(docId, index.getDocName(docId), termPostingMapping);

sorter.score(hit);

hitList.add(hit);

i++;

} else{

docId = posting2.getDocId();

Map<AbstractTerm, AbstractPosting> termPostingMapping = new TreeMap<>();

termPostingMapping.put(queryTerm2, posting2);

AbstractHit hit = new Hit(docId, index.getDocName(docId), termPostingMapping);

sorter.score(hit);

hitList.add(hit);

j++;

}

}

while( i < postingList1.size()){

posting1 = postingList1.get(i);

docId = posting1.getDocId();

Map<AbstractTerm, AbstractPosting> termPostingMapping = new TreeMap<>();

termPostingMapping.put(queryTerm1, posting1);

AbstractHit hit = new Hit(docId, index.getDocName(docId), termPostingMapping);

sorter.score(hit);

hitList.add(hit);

i++;

}

while( j < postingList2.size()){

posting2 = postingList2.get(j);

docId = posting2.getDocId();

Map<AbstractTerm, AbstractPosting> termPostingMapping = new TreeMap<>();

termPostingMapping.put(queryTerm2, posting2);

AbstractHit hit = new Hit(docId, index.getDocName(docId), termPostingMapping);

sorter.score(hit);

hitList.add(hit);

j++;

}

if(hitList.isEmpty())

return null;

sorter.sort(hitList);

AbstractHit[] Hits = new AbstractHit[hitList.size()];

hitList.toArray(Hits);

return Hits;

}

}

return null;

}

}

1. **SimpleSorter.java**

package hust.cs.javacourse.search.query.impl;

import hust.cs.javacourse.search.index.AbstractPosting;

import hust.cs.javacourse.search.index.AbstractTerm;

import hust.cs.javacourse.search.query.AbstractHit;

import hust.cs.javacourse.search.query.Sort;

import java.util.Collections;

import java.util.List;

/\*\*

\* Sort接口的具体实现类

\*/

public class SimpleSorter implements Sort {

/\*\*

\* 对命中结果集合根据文档得分排序

\* @param hits ：命中结果集合

\*/

@Override

public void sort(List<AbstractHit> hits) {

Collections.sort(hits);

}

/\*\*

\* <pre>

\* 计算命中文档的得分, 作为命中结果排序的依据.

\* 计算文档的得分可以采取不同的策略, 因此这里的设计模式采用了策略模式，没有把这个方法放到AbstractHit及其子类里.

\* 而是放到接口Sort里，当我们需要不同的排序策略，只需要重新实现Sort的子类即可。即排序策略与被排序的对象(AbstractHit及其子类)应该分开。

\* 比如如果不排序，只需实现一个最简单的Sort接口实现类，比如叫NullSort类，在这个类里把所有文档的得分设置成一样的值。

\* 文档的得分值计算出来后要设置到AbstractHit子类对象里.

\* @param hit ：命中文档

\* @return ：命中文档的得分

\* </pre>

\*/

@Override

public double score(AbstractHit hit) {

double s = 0.0;

for(AbstractTerm term : hit.getTermPostingMapping().keySet()){

AbstractPosting posting = hit.getTermPostingMapping().get(term);

s += posting.getFreq()\*10.0;

}

hit.setScore(-s);

return -s;

}

}

1. **TestBuildIndex.java**

package hust.cs.javacourse.search.run;

import hust.cs.javacourse.search.index.AbstractDocumentBuilder;

import hust.cs.javacourse.search.index.AbstractIndex;

import hust.cs.javacourse.search.index.AbstractIndexBuilder;

import hust.cs.javacourse.search.index.impl.DocumentBuilder;

import hust.cs.javacourse.search.index.impl.Index;

import hust.cs.javacourse.search.index.impl.IndexBuilder;

import hust.cs.javacourse.search.util.Config;

import hust.cs.javacourse.search.util.FileUtil;

import java.io.File;

/\*\*

\* 测试索引构建

\*/

public class TestBuildIndex {

/\*\*

\* 索引构建程序入口

\* @param args : 命令行参数

\*/

public static void main(String[] args){

AbstractDocumentBuilder documentBuilder = new DocumentBuilder();

AbstractIndexBuilder indexBuilder = new IndexBuilder(documentBuilder);

String rootDir = Config.DOC\_DIR + "真实测试数据集/";

System.out.println("Start building index.");

AbstractIndex index = indexBuilder.buildIndex(rootDir);

System.out.println(index);

String indexFile = Config.INDEX\_DIR + "index.dat";

index.save(new File(indexFile));

AbstractIndex index2 = new Index();

index2.load(new File(indexFile));

System.out.println("\n---------------------------------------------\n");

System.out.println(index2);

//写入文本文件

String txtFile = Config.INDEX\_DIR + "indexToString.txt";

FileUtil.write(index2.toString(),txtFile);

}

}

1. **TestSearchIndex.java**

package hust.cs.javacourse.search.run;

import hust.cs.javacourse.search.index.impl.Term;

import hust.cs.javacourse.search.query.AbstractHit;

import hust.cs.javacourse.search.query.AbstractIndexSearcher;

import hust.cs.javacourse.search.query.Sort;

import hust.cs.javacourse.search.query.impl.IndexSearcher;

import hust.cs.javacourse.search.query.impl.SimpleSorter;

import hust.cs.javacourse.search.util.Config;

/\*\*

\* 测试搜索

\*/

public class TestSearchIndex {

/\*\*

\* 搜索程序入口

\* @param args ：命令行参数

\*/

public static void main(String[] args){

Sort simpleSorter = new SimpleSorter();

String indexFile = Config.INDEX\_DIR + "index.dat";

AbstractIndexSearcher searcher = new IndexSearcher();

searcher.open(indexFile);

AbstractHit[] hits = searcher.search(new Term("coronavirus"),

new Term("wales"),

simpleSorter,

AbstractIndexSearcher.LogicalCombination.AND);

//AbstractHit[] hits = searcher.search(new Term("coronavirus"), simpleSorter);

if(hits == null) {

System.out.println("Not Found");

System.exit(1);

}

for(AbstractHit hit : hits){

System.out.println(hit);

}

}

}

1. **Config.java**

package hust.cs.javacourse.search.util;

/\*\*

\* <pre>

\*

\* 保存搜索引擎的配置信息，例如：

\* 索引文件所在目录

\* 要建立索引的文本文件所在目录

\* 构建索引时是否忽略单词大小写

\* 分词所需要的正则表达式

\* 基于正则表达式的三元组过滤器所需的正则表达式

\* 基于单词长度的三元组过滤器所需的最小单词长度和最大单词长度

\* ...

\* </pre>

\*

\*/

public class Config {

/\*\*

\* Java工程HOME目录, System.getProperty("user.dir")返回当前JAVA工程目录

\*/

public static String PROJECT\_HOME\_DIR= System.getProperty("user.dir");

/\*\*

\* <pre>

\* 索引文件的目录,以相对路径指定索引文件目录,将索引文件保存在当前工程目录下的index子目录中.

\* 这样做的好处：索引文件目录是相对路径,无论你把整个工程放在什么位置，程序都可以正常运行.

\* </pre>

\*/

public static String INDEX\_DIR = PROJECT\_HOME\_DIR + "/index/";

/\*\*

\* <pre>

\* 文本文件的目录,以相对路径指定文本文件目录 ,将文本文件保存在当前工程目录下的text子目录中

\* 这样做的好处：文本文件目录是相对路径,无论你把整个工程放在什么位置，程序都可以正常运行.

\* </pre>

\*

\*/

public static String DOC\_DIR = PROJECT\_HOME\_DIR + "/text/";

/\*\*

\* 构建索引和检索时是否忽略单词大小写

\*/

public static boolean IGNORE\_CASE = true;

/\*\*

\* <pre>

\* 将字符串切分成单词时所需的正则表达式.

\* 例如根据中英文的逗号,分号,句号，问号，冒号,感叹号，中文顿号，空白分割符进行切分

\* </pre>

\*

\*/

public static String STRING\_SPLITTER\_REGEX = "[,|，|;|；|.|。|?|？|:|：|!|！|、|\\s]+";

/\*\*

\* <pre>

\* 单词过滤的正则表达式.

\* 例如正则表达式指定只保留由字母组成的term,其他的term全部过滤掉,不写入倒排索引

\* </pre>

\*/

public static String TERM\_FILTER\_PATTERN = "[a-zA-z]+";

/\*\*

\* <pre>

\* 基于单词的最小长度过滤单词.

\* 例如指定最短单词长度为3，长度小于3的单词过滤掉，不写入倒排索引

\* </pre>

\*/

public static int TERM\_FILTER\_MINLENGTH = 3;

/\*\*

\* <pre>

\* 基于单词的最小长度过滤单词.

\* 例如指定最长单词长度为20，长度大于20的单词过滤掉，不写入倒排索引

\* </pre>

\*/

public static int TERM\_FILTER\_MAXLENGTH = 20;

}

1. **FileUtil**

package hust.cs.javacourse.search.util;

import java.io.\*;

import java.util.ArrayList;

import java.util.List;

/\*\*

\* 文件操作的工具类

\*/

public class FileUtil {

private FileUtil(){}

/\*\*

\* 一次读取指定文本文件的所有内容

\* @param filePath ：指定文本文件的绝对路径

\* @return ： 文本文件的内容

\*/

public static String read(String filePath){

String s = null;

BufferedReader reader = null;

try{

StringBuffer buf = new StringBuffer();

reader = new BufferedReader(new InputStreamReader(new FileInputStream(new File(filePath))));

while( (s = reader.readLine()) != null){

buf.append(s).append("\n"); //reader.readLine())返回的字符串会去掉换行符，因此这里要加上

}

s = buf.toString().trim(); //去掉最后一个多的换行符

} catch (FileNotFoundException e) {

e.printStackTrace();

} catch (IOException e) {

e.printStackTrace();

}

finally {

if(reader != null) {

try {

reader.close();

} catch (IOException e) {

e.printStackTrace();

}

}

}

return s;

}

/\*\*

\* 将字符串写入到指定的文本文件

\* @param content ： 写入的内容

\* @param filePath ： 指定的文本文件路径

\*/

public static void write(String content, String filePath){

PrintWriter writer = null;

try{

writer = new PrintWriter(new BufferedWriter(new OutputStreamWriter(new FileOutputStream(new File(filePath)))));

writer.print(content);

} catch (FileNotFoundException e) {

e.printStackTrace();

}

finally {

if(writer != null){

writer.close();

}

}

}

/\*\*

\* 列出指定目录下所有文件的绝对路径(不递归)

\* @param dirPath : 指定目录

\* @return ： 指定目录下所有文件的绝对路径的列表

\*/

public static List<String> list(String dirPath){

List<String> filePaths = new ArrayList<String>();

try{

File dir = new File(dirPath);

if(dir.isDirectory()){

File[] files = dir.listFiles();

for(File f: files){

if(f.isFile()) {

filePaths.add(f.getCanonicalPath()); //File类的getCanonicalPath方法返回绝对路径

}

}

}

}

catch (IOException e) {

e.printStackTrace();

}

return filePaths;

}

/\*\*

\* 列出指定目录下的匹配指定后缀名的所有文件的绝对路径(不递归)

\* @param dirPath : 指定目录

\* @param suffix ：指定后缀名, 如 .txt

\* @return : 所有匹配指定后缀名的文件绝对路径列表

\*/

public static List<String> list(String dirPath, String suffix){

List<String> filePaths = new ArrayList<String>();

try{

File dir = new File(dirPath);

if(dir.isDirectory()){

File[] files = dir.listFiles();

for(File f: files){

if(f.isFile()) {

String path = f.getCanonicalPath(); //File类的getCanonicalPath方法返回绝对路径

if(path.toLowerCase().endsWith(suffix.toLowerCase())) {

filePaths.add(path);

}

}

}

}

}

catch (IOException e) {

e.printStackTrace();

}

return filePaths;

}

public static void main(String[] args){

//从文件读

String fileName = Config.DOC\_DIR + "1.txt";

String s = FileUtil.read(fileName);

System.out.println(s);

//写入到另一个文件

String fileName2 = Config.DOC\_DIR + "2.txt";

FileUtil.write(s,fileName2);

//列出指定目录下所有文件

String dirName = Config.PROJECT\_HOME\_DIR + "/javadoc";

List<String> fileNames = FileUtil.list(dirName);

for(String p: fileNames){

System.out.println(p);

}

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

//列出指定目录下所有.html文件

String dirName1 = Config.PROJECT\_HOME\_DIR + "/javadoc";

List<String> fileNames1 = FileUtil.list(dirName,".html");

for(String p: fileNames1){

System.out.println(p);

}

}

}

1. **StringSplitter.java**

package hust.cs.javacourse.search.util;

import java.util.ArrayList;

import java.util.List;

import java.util.regex.Matcher;

import java.util.regex.Pattern;

/\*\*

\* 字符串分割类，根据标点符号和空白符将字符串分成一个个单词

\*/

public class StringSplitter {

public StringSplitter(){}

private String splitRegex = null;

private Pattern pattern = null;

private Matcher match = null;

/\*\*

\* 设置分词用的正则表达式

\* @param regex：分词用的正则表达式

\*/

public void setSplitRegex(String regex){

splitRegex = regex;

pattern = Pattern.compile(splitRegex);

}

/\*\*

\* 将字符串分割成单词列表

\* @param input： 输入字符串

\* @return ： 分词得到的单词列表

\*/

public List<String> splitByRegex(String input){

List<String> list = new ArrayList<String>();

match = pattern.matcher(input);

String part = null;

int lastEnd = 0;

while(match.find()){

int start = match.start(0);

int end = match.end(0);

part = input.substring(lastEnd, match.start(0));

lastEnd = match.end(0);

if( (part != null && part.equals("")) || part == null) {

continue;

}

list.add(part);

}

//取得最后一部分

if(lastEnd < input.length()){

part = input.substring(lastEnd);

if(part != null && !part.equals(""))

list.add(part);

}

return list;

}

public static void main(String[] args){

StringSplitter splitter = new StringSplitter();

String regex1TestInput = "key1,， ,Key2;Key3，:Key4;；;Key5；？?Key6，!Key7；Key8 key9\nkey10.。？key11";

// String regex1TestInput = "\n\naaa bbb ccc\n\nddd is F17";

splitter.setSplitRegex(Config.STRING\_SPLITTER\_REGEX);

List<String> parts = splitter.splitByRegex(regex1TestInput);

for(int i = 0; i < parts.size(); i++){

System.out.print(parts.get(i) + " ");

}

}

}