

**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 ：**

**2025 年 4 月 29 日**

1. **需求分析**
2. **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：

**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：

（1）读取文本文件的内容；

（2）将内容切分成一个个的单词；

（3）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）；

（4）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；

（5）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；

（6）可以在控制台输出索引的内容。

**功能2：**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：

（1）根据搜索关键词得到命中的结果集合；

（2）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；

（3）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；

**功能3：**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：

（1）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；

（2）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；

（3）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；

**功能4：**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**。

除了以上功能上的要求外，其他要求包括：

（1）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。

（2）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。

（3）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。

（4）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。

**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。**

1. **需求分析**

自行对题目要求进行细化、补充， 例如发生异常的条件。

1. **项目整体需求**

表格 1 项目整体需求

|  |  |
| --- | --- |
| **功能** | **要求** |
| 文档索引系统 | 能够处理文本文档，提取关键词 |
| 构建倒排索引，记录词项与文档的对应关系 |
| 支持索引的持久化存储与加载 |
| 文本解析功能 | 支持文本分词 |
| 过滤停用词 |
| 基于长度和模式的词项过滤 |
| 处理标点符号和特殊字符 |
| 查询系统 | 支持单词查询 |
| 支持多词逻辑组合查询(AND, OR) |
| 支持相邻词查询(ADJACENT) |
| 结果排序与评分 |
| 用户接口 | 提供测试用例展示各种查询功能 |

1. **功能模块分析**

表格 2 文本解析模块

|  |  |
| --- | --- |
| 分词器 | 将文本分割为单词序列 |
| 过滤链 | 通过多级过滤器处理分词结果 |
| 停用词过滤器：移除常见无意义词 |
| 长度过滤器：基于词长进行筛选 |
| 模式过滤器：基于正则表达式进行筛选 |

表格 3 文档处理模块

|  |  |
| --- | --- |
| 文档表示 | 包含文档ID、路径等元数据 |
| 文档构建 | 从原始文本构建结构化文档对象 |
| 词项处理 | 记录词项在文档中的位置信息 |

表格 4 索引模块

|  |  |
| --- | --- |
| 索引结构 | 使用倒排索引存储词项到文档的映射 |
| 索引构建 | 从文档集合中提取词项并构建索引 |
| 索引操作 | 支持添加、删除、查询和序列化操作 |

表格 5 查询模块

|  |  |
| --- | --- |
| 查询执行 | 根据用户输入在索引中检索匹配文档 |
| 逻辑运算 | 支持AND、OR、ADJACENT（**进阶功能**）等逻辑操作 |
| 结果处理 | 记录匹配文档及词项出现频率、位置等信息 |
| 结果排序 | 根据相关性对查询结果进行排序 |

1. **系统设计**
2. **概要设计**

根据以上模块分析，可以画出如下ER图：

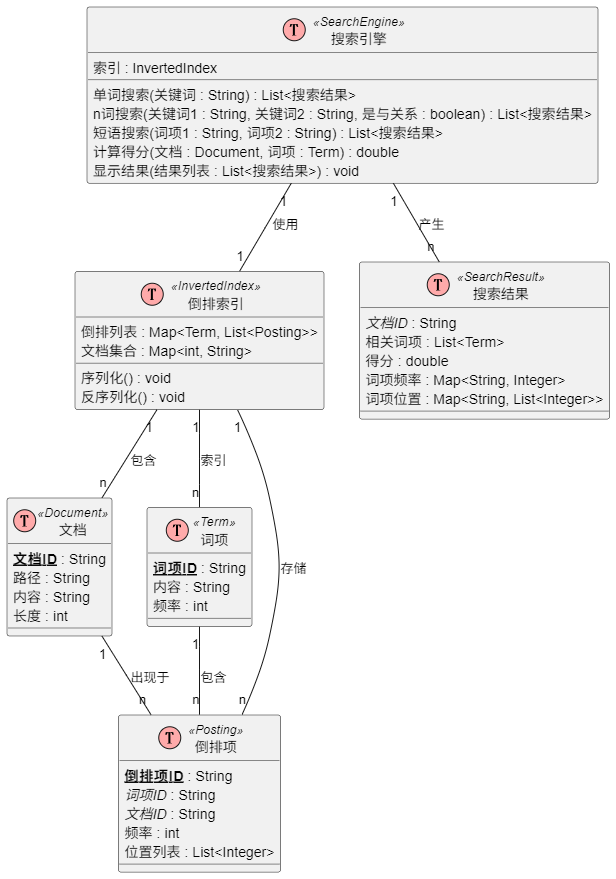


图 1 ER图分析

1. 文档 (Document)：被索引和搜索的基本对象，代表待搜索的文本文件，包含文档ID、路径、内容和长度等属性。
2. 词项 (Term)：索引的基本单位，代表经过分词和过滤后的单词，包含词项ID、值和文档频率(出现该词的文档数量)。
3. 倒排项 (Posting)：连接词项和文档的关系实体，记录词项在各个文档中的频率和位置信息，支持词项到文档的快速查找。
4. 倒排索引 (InvertedIndex)：系统的核心数据结构，管理词项映射、倒排列表和文档集合，提供序列化/反序列化功能。
5. 搜索结果 (SearchResult)：表示搜索返回的结果，包含文档ID、相关词项、得分等信息，记录词项在文档中的频率和位置。
6. 搜索引擎 (SearchEngine)：系统的功能实现层，使用倒排索引提供各种搜索功能，包含单词搜索、多词搜索和短语搜索等方法。

经过进一步分析和细化，可以得出如下UML模型，其中：

1. Term：表示索引和查询中的单个词项（单词），是索引的基本单位，也是搜索的基本单位。
2. TermTuple：表示文档解析过程中产生的词项元组，作为文档解析和索引构建过程中的中间数据结构，封装了词项、位置和频率信息，是从文本流向索引结构转换的桥梁。
3. Document：表示被索引的文本文档，存储原始文本内容，是索引指向的目标对象。
4. Posting：倒排索引中的倒排项，存储词项对应的文档及相关信息（出现频率、位置等）。
5. PostingList：某个词项的倒排列表，存储包含该词项的所有文档的倒排项集合。
6. Index：倒排索引，维护词项到文档的映射关系，是搜索引擎的核心数据结构。
7. Hit：表示搜索查询结果中的一个命中文档及其详细信息，封装搜索结果的详细信息，提供命中文档与查询词项的关联数据，存储用于结果排序的得分。

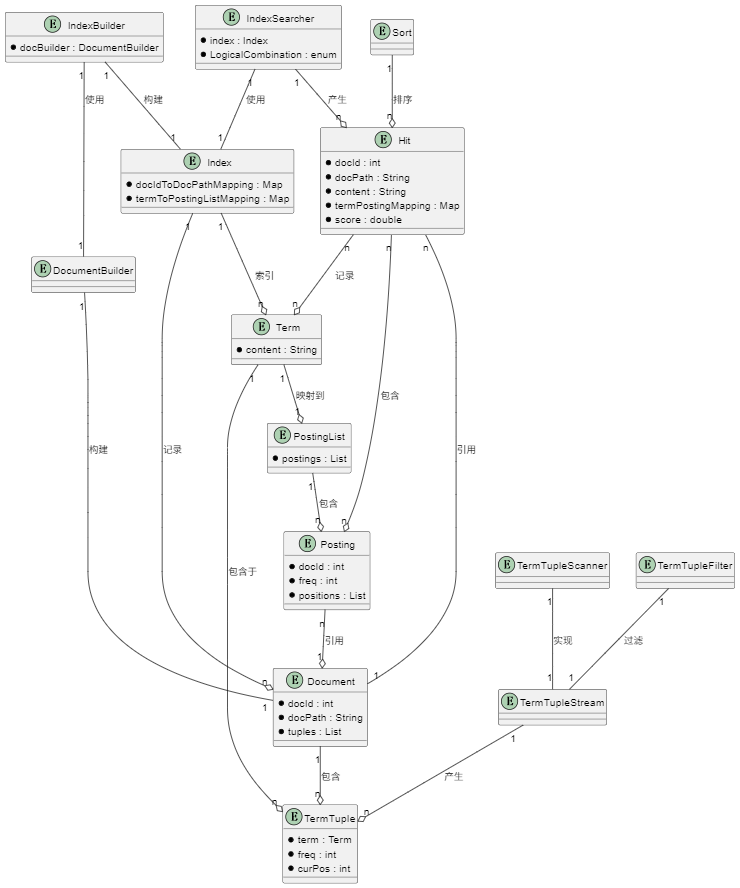


图 2 UML模型图

程序工作流程如下：

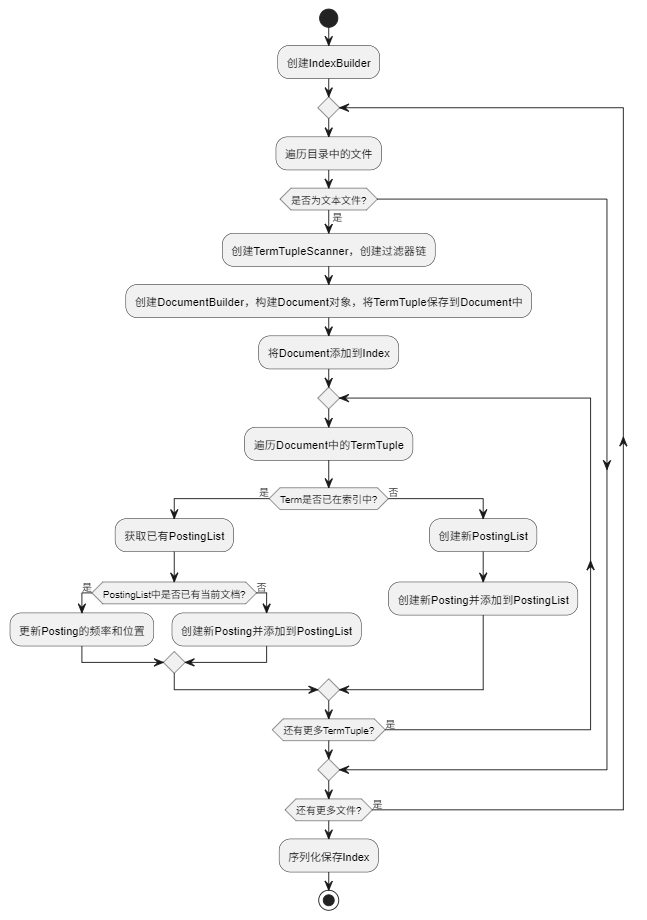


图 3 索引创建流程

**索引创建流程：**

1. 程序启动后，首先读取配置的文档目录路径，创建IndexBuilder对象准备构建索引。
2. 文档遍历：系统遍历指定目录下的所有文本文件，对每个文件进行处理。
3. 文档处理：为每个文本文件创建TermTupleScanner，将文本内容读入并分词，构建过滤器链，包括StopWordTermTupleFilter（停用词过滤）、PatternTermTupleFilter（模式过滤）和LengthTermTupleFilter（长度过滤），过滤器链依次处理分词结果，去除不需要的词项，使用DocumentBuilder将过滤后的词项流构建成Document对象。
4. 索引更新：将构建好的Document添加到Index中；遍历Document中的每个TermTuple（词项和位置信息），检查Term是否已存在于索引中。如果存在，获取对应的PostingList，更新对应Posting的词频和位置信息；如果不存在，创建新的Posting和PostingList，将新建或更新的PostingList添加到索引中
5. 重复步骤2-4，直到处理完所有文档。
6. 索引保存：将构建好的索引序列化保存到指定的文件中，以便后续使用。

**单词搜索流程：**

1. 程序启动后，从指定路径加载预先构建好的索引文件，创建IndexSearcher对象。
2. 查询准备：根据用户输入创建查询Term对象。
3. 执行搜索：在索引中查找匹配的Term。如果找到，获取对应的PostingList，如果未找到，返回空结果。
4. 结果处理：遍历获取到的PostingList中的每个Posting，从Posting中获取文档ID、词频和位置信息，为每个匹配的文档创建Hit对象，并根据词频计算文档得分，最后将Hit对象添加到结果列表中。
5. 结果排序：根据文档得分对结果列表进行排序，得分越高的排在前面。
6. 结果展示：输出显示排序后的搜索结果，包括匹配文档的路径、内容、得分和匹配词项信息。

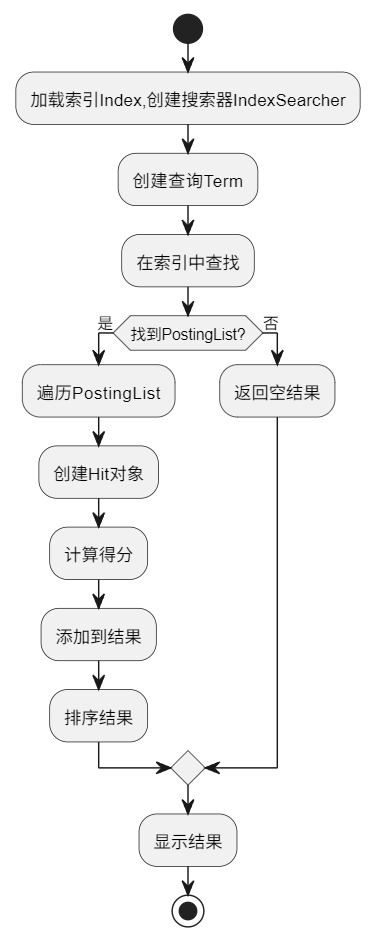


图 4 单词搜索流程图

1. **详细设计**
2. **索引相关模块 (hust.cs.javacourse.search.index.impl)**

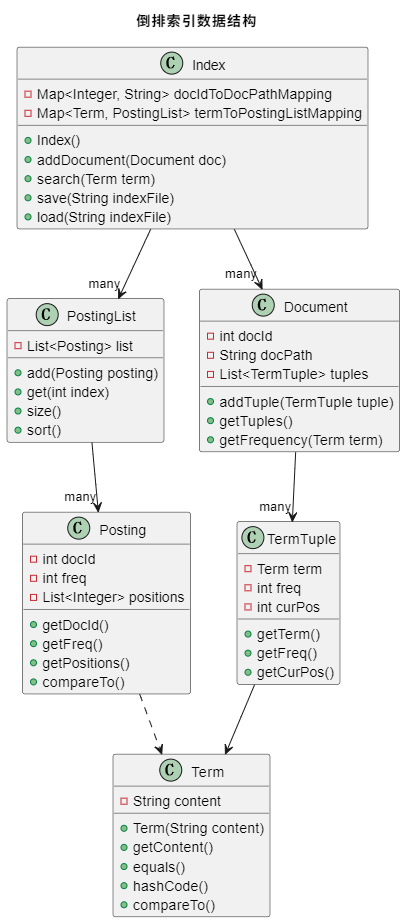


图 5 索引模块数据结构

1. **Term类**

表示文档中的一个词项，实现可比较接口用于字典排序，支持序列化/反序列化以实现索引持久化，包含词项内容及相关操作。

表格 6 Term类成员

|  |  |  |
| --- | --- | --- |
| 数据成员 | 类型 | 说明 |
| content | String | 表示单词内容 |

表格 7 Term类方法

|  |  |  |  |
| --- | --- | --- | --- |
| 方法 | 参数 | 返回类型 | 功能描述 |
| Term() | 无 | - | 无参构造函数 |
| Term(String content) | content: 单词内容 | - | 有参构造函数 |
| equals(Object obj) | obj: 比较对象 | boolean | 判断两个Term内容是否相同 |
| toString() | 无 | String | 返回Term的字符串表示 |
| getContent() | 无 | String | 获取Term内容 |
| setContent(String content) | content: 单词内容 | void | 设置Term内容 |
| compareTo(AbstractTerm o) | o: 比较对象 | int | 比较两个Term字典序 |
| writeObject(ObjectOutputStream out) | out: 输出流 | void | 序列化Term对象 |
| readObject(ObjectInputStream in) | in: 输入流 | void | 反序列化Term对象 |

Term类是索引中的基本单元，表示一个词项。创建Term对象时，会对输入的content字符串进行处理：转换为小写形式，去除两端空白字符。Term类实现了hashCode和equals方法，基于content字段进行判断，这使得Term可以作为HashMap的键。Term还实现了Comparable接口的compareTo方法，允许按字典序比较Term对象。此外，Term实现了序列化接口，支持通过I/O流进行传输和存储。这些特性使Term成为构建倒排索引的理想基础组件。

1. **TermTuple类**

表示带有位置信息的词项出现，存储词项、频率（单个出现的频率始终为1）和位置，用于文档解析和索引构建过程。

表格 8 TermTuple 类成员

|  |  |  |
| --- | --- | --- |
| 数据成员 | 类型 | 说明 |
| term | AbstractTerm | 单词 |
| freq | int | 出现频率(固定为1) |
| curPos | int | 单词在文档中的位置 |

表格 9 TermTuple 类方法

|  |  |  |  |
| --- | --- | --- | --- |
| 方法 | 参数 | 返回类型 | 功能描述 |
| TermTuple() | 无 | - | 无参构造函数 |
| TermTuple(String word, int curPos) | word: 单词, curPos: 位置 | - | 有参构造函数 |
| equals(Object obj) | obj: 比较对象 | boolean | 判断两个TermTuple是否相同 |
| toString() | 无 | String | 返回TermTuple的字符串表示 |

1. **Posting类**

表示词项倒排列表中的一个条目，包含文档ID、词项在文档中的频率和位置列表，实现可比较接口用于高效合并操作，支持序列化/反序列化。

表格 10 Posting类成员

|  |  |  |
| --- | --- | --- |
| 数据成员 | 类型 | 说明 |
| docId | int | 文档ID |
| freq | int | 词频 |
| positions | List<Integer> | 单词出现位置列表 |

表格 11 Posting类方法

|  |  |  |  |
| --- | --- | --- | --- |
| 方法 | 参数 | 返回类型 | 功能描述 |
| Posting() | 无 | - | 无参构造函数 |
| Posting(int docId, int freq, List<Integer> positions) | docId: 文档ID, freq: 频率, positions: 位置列表 | - | 有参构造函数 |
| equals(Object obj) | obj: 比较对象 | boolean | 判断两个Posting是否相同 |
| toString() | 无 | String | 返回Posting的字符串表示 |
| getDocId() | 无 | int | 获取文档ID |
| setDocId(int docId) | docId: 文档ID | void | 设置文档ID |
| getFreq() | 无 | int | 获取频率 |
| setFreq(int freq) | freq: 频率 | void | 设置频率 |
| getPositions() | 无 | List<Integer> | 获取位置列表 |
| setPositions(List<Integer> positions) | positions: 位置列表 | void | 设置位置列表 |
| compareTo(AbstractPosting o) | o: 比较对象 | int | 比较两个Posting(按docId) |
| sort() | 无 | void | 对位置列表排序 |
| writeObject(ObjectOutputStream out) | out: 输出流 | void | 序列化Posting对象 |
| readObject(ObjectInputStream in) | in: 输入流 | void | 反序列化Posting对象 |

1. **PostingList类**

管理特定词项的所有倒排记录，提供添加、排序和搜索倒排记录的方法，实现序列化/反序列化用于索引持久化。

表格 12 PostingList类成员

|  |  |  |
| --- | --- | --- |
| 数据成员 | 类型 | 说明 |
| list | List<AbstractPosting> | Posting列表 |

表格 13 PostingList类方法

|  |  |  |  |
| --- | --- | --- | --- |
| 方法 | 参数 | 返回类型 | 功能描述 |
| add(AbstractPosting posting) | posting: 添加的Posting | void | 添加Posting |
| toString() | 无 | String | 返回PostingList的字符串表示 |
| add(List<AbstractPosting> postings) | postings: Posting列表 | void | 添加多个Posting |
| get(int index) | index: 索引 | AbstractPosting | 获取指定位置的Posting |
| indexOf(AbstractPosting posting) | posting: 查找的Posting | int | 查找Posting的索引 |
| indexOf(int docId) | docId: 文档ID | int | 查找包含指定文档ID的Posting索引 |
| contains(AbstractPosting posting) | posting: 查找的Posting | boolean | 判断是否包含指定Posting |
| remove(int index) | index: 索引 | void | 删除指定位置的Posting |
| remove(AbstractPosting posting) | posting: 要删除的Posting | void | 删除指定Posting |
| size() | 无 | int | 获取PostingList大小 |
| clear() | 无 | void | 清空PostingList |
| isEmpty() | 无 | boolean | 判断PostingList是否为空 |
| sort() | 无 | void | 对PostingList排序 |
| writeObject(ObjectOutputStream out) | out: 输出流 | void | 序列化PostingList对象 |
| readObject(ObjectInputStream in) | in: 输入流 | void | 反序列化PostingList对象 |

PostingList.sort方法用于优化倒排列表的结构。首先，它对Posting列表按照文档ID进行排序，使用Posting类定义的compareTo方法（通常是按docId升序排列）。然后，对每个Posting内部的位置列表也进行排序，确保位置值从小到大排列。这种排序操作有助于提高后续搜索操作的效率，特别是在执行交集（AND）、并集（OR）或相邻（ADJACENT）等复杂操作时。

1. **Document类**

表示搜索系统中的一个文档，包含文档ID、路径和所有词项元组，管理词项元组的添加和检索。

表格 14 Document类成员

|  |  |  |
| --- | --- | --- |
| 数据成员 | 类型 | 说明 |
| docId | int | 文档ID |
| docPath | String | 文档绝对路径 |
| tuples | List<AbstractTermTuple> | 文档包含的三元组列表 |

表格 15 Document类方法

|  |  |  |  |
| --- | --- | --- | --- |
| 方法 | 参数 | 返回类型 | 功能描述 |
| Document() | 无 | - | 无参构造函数 |
| Document(int docId, String docPath) | docId: 文档ID, docPath: 文档路径 | - | 有参构造函数 |
| Document(int docId, String docPath, List<AbstractTermTuple> tuples) | docId: 文档ID, docPath: 文档路径, tuples: 三元组列表 | - | 有参构造函数 |
| getDocId() | 无 | int | 获取文档ID |
| setDocId(int docId) | docId: 文档ID | void | 设置文档ID |
| getDocPath() | 无 | String | 获取文档路径 |
| setDocPath(String docPath) | docPath: 文档路径 | void | 设置文档路径 |
| getTuples() | 无 | List<AbstractTermTuple> | 获取三元组列表 |
| addTuple(AbstractTermTuple tuple) | tuple: 三元组 | void | 添加三元组 |
| contains(AbstractTermTuple tuple) | tuple: 三元组 | boolean | 判断是否包含指定三元组 |
| getTuple(int index) | index: 索引 | AbstractTermTuple | 获取指定位置的三元组 |
| getTupleSize() | 无 | int | 获取三元组数量 |
| toString() | 无 | String | 返回Document的字符串表示 |

1. **DocumentBuilder类**

从文件或词项元组流创建Document对象，使用装饰者模式的过滤器管道处理文档内容，连接解析模块和索引模块。

表格 16 DocumentBuilder类方法

|  |  |  |  |
| --- | --- | --- | --- |
| 方法 | 参数 | 返回类型 | 功能描述 |
| build(int docId, String docPath, AbstractTermTupleStream termTupleStream) | docId: 文档ID, docPath: 文档路径, termTupleStream: 三元组流 | AbstractDocument | 从三元组流构建文档 |
| build(int docId, String docPath, File file) | docId: 文档ID, docPath: 文档路径, file: 文件 | AbstractDocument | 从文件构建文档 |

1. **Index类**

实现核心的倒排索引数据结构，维护从文档ID到路径以及从词项到倒排列表的映射，提供添加文档、保存/加载索引到磁盘的方法，支持词项搜索操作，返回查询词项的倒排列表，包含优化方法用于排序倒排列表和位置信息。

表格 17 Index类成员

|  |  |  |
| --- | --- | --- |
| 数据成员 | 类型 | 说明 |
| docIdToDocPathMapping | Map<Integer, String> | 文档ID到路径的映射 |
| termToPostingListMapping | Map<AbstractTerm, AbstractPostingList> | 词项到倒排列表的映射 |

表格 18 Index类方法

|  |  |  |  |
| --- | --- | --- | --- |
| 方法 | 参数 | 返回类型 | 功能描述 |
| toString() | 无 | String | 返回索引的字符串表示 |
| addDocument(AbstractDocument document) | document: 文档 | void | 添加文档到索引 |
| load(File file) | file: 文件 | void | 从文件加载索引 |
| save(File file) | file: 文件 | void | 将索引保存到文件 |
| search(AbstractTerm term) | term: 查询词项 | AbstractPostingList | 查找指定词项的倒排列表 |
| getDictionary() | 无 | Set<AbstractTerm> | 获取索引词典 |
| optimize() | 无 | void | 优化索引 |
| getDocName(int docId) | docId: 文档ID | String | 获取指定ID的文档路径 |
| writeObject(ObjectOutputStream out) | out: 输出流 | void | 序列化索引 |
| readObject(ObjectInputStream in) | in: 输入流 | void | 反序列化索引 |

当一个Document需要被添加到索引时，Index.addDocument方法首先会将文档ID与文档路径的映射关系存储到docIdToDocPathMapping中。然后，它会遍历Document中的所有TermTuple（词项元组），对每个TermTuple执行以下操作：获取当前的Term（词项），检查索引中是否已存在该Term。如果已存在，则获取对应的PostingList；否则，创建一个新的PostingList并添加到索引的termToPostingListMapping中。接着，检查当前PostingList中是否已有当前文档的Posting记录。如果已有，则更新现有Posting，增加词频，并添加新的位置信息；如果没有，则创建一个新的Posting，设置文档ID、词频（初始为1）和位置信息，然后将这个新Posting添加到PostingList中。整个过程实现了文档内容到倒排索引结构的转换，确保每个词项都能通过索引找到它出现的所有文档及位置。

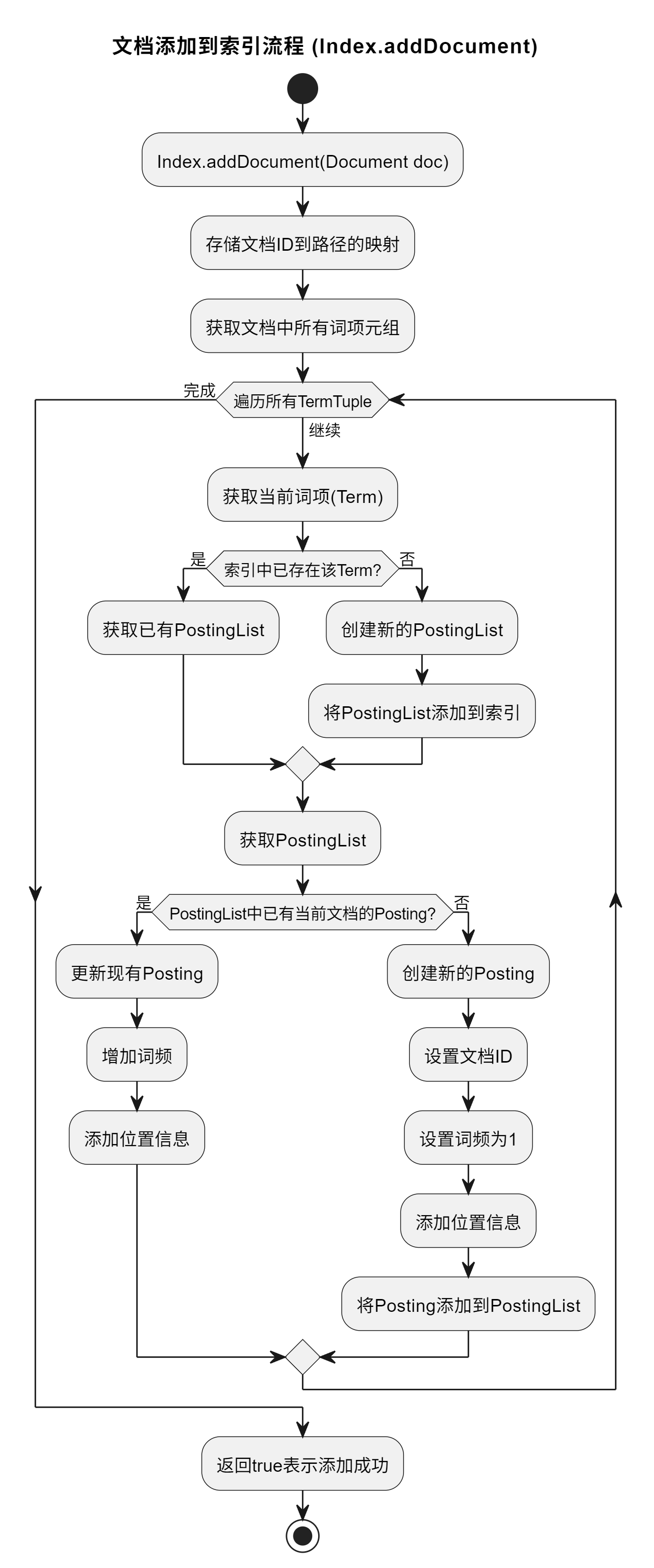


图 6 addDocument算法流程

.search方法实现了对单个词项的搜索功能。首先在索引的termToPostingListMapping中查找参数Term。如果索引中存在该Term，则获取对应的PostingList并返回其副本（防止原始数据被修改）；如果不存在，则返回一个空的PostingList。这个操作是所有搜索功能的基础，后续的复杂查询（如AND、OR、ADJACENT）都建立在单词搜索的基础上。

Index.save方法负责将索引内容写入文件：首先创建输出流，写入docIdToDocPathMapping的大小，然后遍历所有文档ID和路径对，将它们写入文件。接着，写入termToPostingListMapping的大小，遍历所有Term和对应的PostingList，依次序列化并写入文件。最后关闭输出流完成保存。load方法则是相反的过程：创建输入流，读取docIdToDocPathMapping的大小，然后读取相应数量的文档ID和路径对，重建docIdToDocPathMapping。接着读取termToPostingListMapping的大小，依次反序列化Term和PostingList对象，重建termToPostingListMapping。这两个方法使得索引可以在系统关闭后仍然保持，避免重复构建索引的开销。

1. **IndexBuilder类**

从文档目录构建倒排索引，协调文档构建和添加到索引的过程，管理文档ID分配过程。

IndexBuilder负责从文档集合构建倒排索引的过程。首先，它会创建一个空的Index实例作为索引容器。然后遍历指定目录下的所有文本文件，对每个文件执行以下操作：获取文件路径，为文档分配唯一的文档ID，使用DocumentBuilder将文件内容解析为Document对象，最后将该Document添加到索引中。在所有文档处理完毕后，会对索引进行优化操作，主要是对各个PostingList进行排序，以便后续的检索操作更高效。这个过程将一组原始文档转换为结构化的倒排索引，为后续的搜索查询提供基础。

表格 19 IndexBuilder类成员

|  |  |  |
| --- | --- | --- |
| 数据成员 | 类型 | 说明 |
| docBuilder | AbstractDocumentBuilder | 文档构建器 |

表格 20 IndexBuilder类方法

|  |  |  |  |
| --- | --- | --- | --- |
| 方法 | 参数 | 返回类型 | 功能描述 |
| IndexBuilder(AbstractDocumentBuilder docBuilder) | docBuilder: 文档构建器 | - | 构造函数 |
| buildIndex(String rootDirectory) | rootDirectory: 根目录 | AbstractIndex | 构建索引 |

1. **解析相关模块 (hust.cs.javacourse.search.parse.impl)**

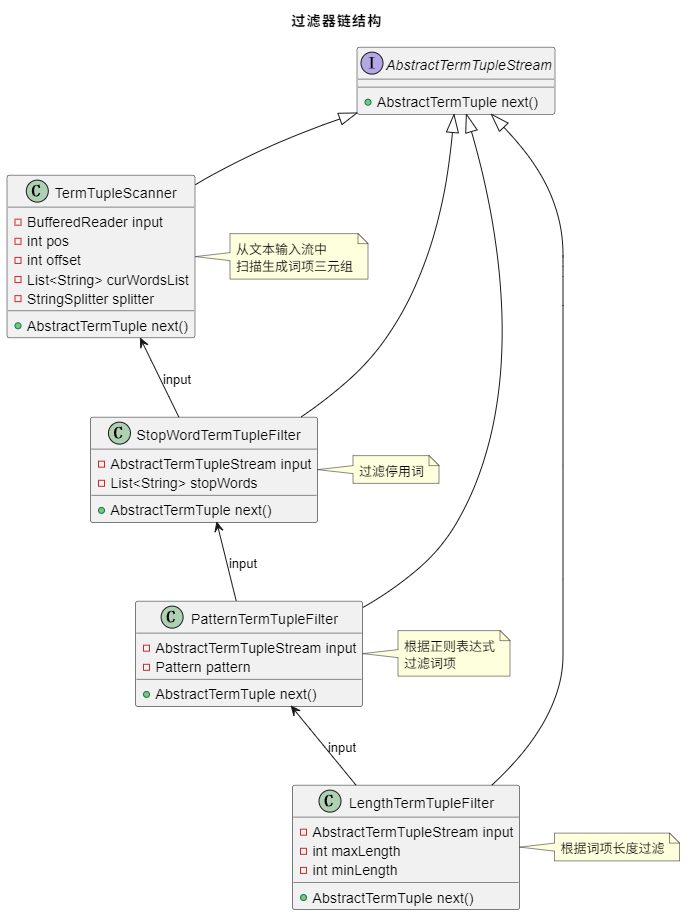


图 7 解析模块数据结构

1. **TermTupleScanner类**

读取文本内容并生成词项元组，按行处理输入，分割成单词，跟踪单词在文档中的位置。

表格 21 TermTupleScanner类成员

|  |  |  |
| --- | --- | --- |
| 数据成员 | 类型 | 说明 |
| pos | int | 当前行内位置 |
| offset | int | 之前行的词数 |
| curWordsList | List<String> | 当前行单词列表 |
| splitter | StringSplitter | 字符串分割器 |

表格 22 TermTupleScanner类方法

|  |  |  |  |
| --- | --- | --- | --- |
| 方法 | 参数 | 返回类型 | 功能描述 |
| TermTupleScanner (BufferedReader input) | input: 输入流 | - | 构造函数 |
| next() | 无 | AbstractTermTuple | 获取下一个三元组 |

1. **StopWordTermTupleFilter类**

过滤掉常用词（如"the"、"is"、"and"）。

表格 23 StopWordTermTupleFilter类成员

|  |  |  |
| --- | --- | --- |
| 数据成员 | 类型 | 说明 |
| stopWords | List<String> | 停用词列表 |

表格 24 StopWordTermTupleFilter类方法

|  |  |  |  |
| --- | --- | --- | --- |
| 方法 | 参数 | 返回类型 | 功能描述 |
| StopWordTermTupleFilter (AbstractTermTupleStream input) | input: 输入流 | - | 构造函数(默认停用词) |
| StopWordTermTupleFilter(AbstractTermTupleStream input, List<String> stopWords) | input: 输入流, stopWords: 停用词列表 | - | 构造函数(自定义停用词) |
| next() | 无 | AbstractTermTuple | 获取下一个非停用词三元组 |

1. **PatternTermTupleFilter类**

基于正则表达式模式过滤词项，只保留匹配指定模式的词项（如仅字母字符）。

表格 25 类成员

|  |  |  |
| --- | --- | --- |
| 数据成员 | 类型 | 说明 |
| pattern | Pattern | 单词匹配模式 |

表格 26 类方法

|  |  |  |  |
| --- | --- | --- | --- |
| 方法 | 参数 | 返回类型 | 功能描述 |
| PatternTermTupleFilter(AbstractTermTupleStream input) | input: 输入流 | - | 构造函数 |
| next() | 无 | AbstractTermTuple | 获取下一个匹配模式的三元组 |

1. **LengthTermTupleFilter类**

基于词项长度进行过滤，移除过短（<3字符）或过长（>20字符）的词项。

表格 27 类成员

|  |  |  |
| --- | --- | --- |
| 数据成员 | 类型 | 说明 |
| maxLength | int | 最大长度 |
| minLength | int | 最小长度 |

表格 28 类方法

|  |  |  |  |
| --- | --- | --- | --- |
| 方法 | 参数 | 返回类型 | 功能描述 |
| LengthTermTupleFilter(AbstractTermTupleStream input) | input: 输入流 | - | 构造函数 |
| next() | 无 | AbstractTermTuple | 获取下一个符合长度要求的三元组 |

1. **查询相关模块 (hust.cs.javacourse.search.query.impl)**

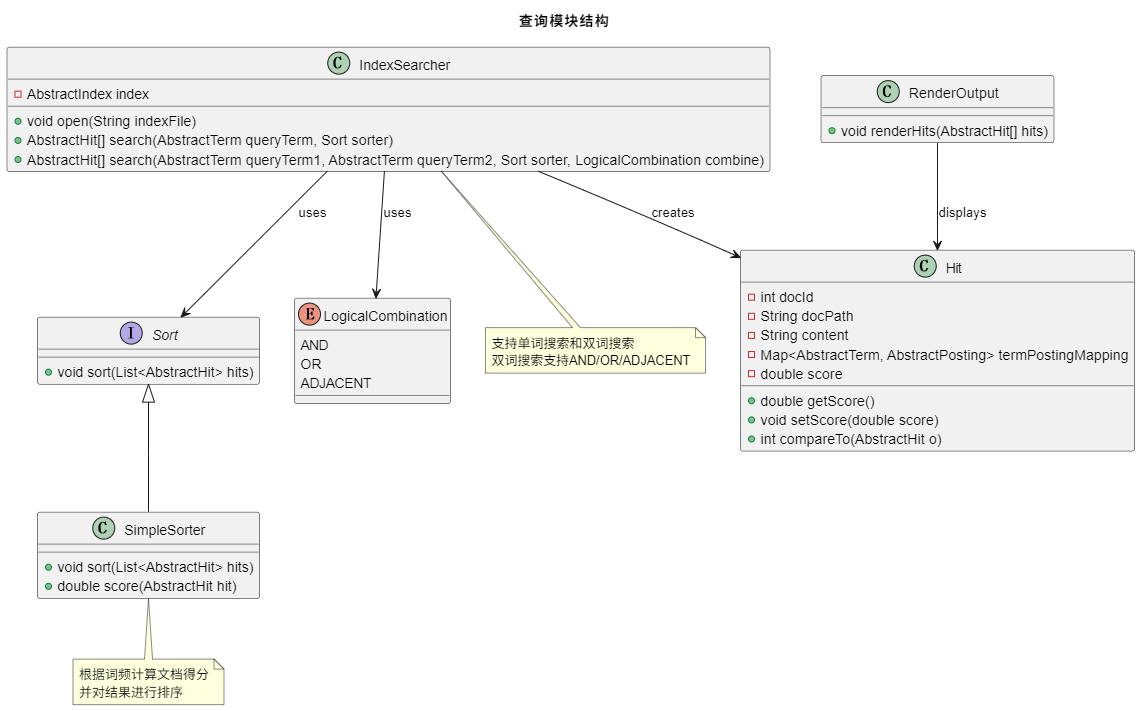


图 8 查询模块数据结构

1. **Hit类**

表示搜索结果（匹配查询的文档），包含文档ID、路径、内容、匹配词项和相关性分数，实现可比较接口用于结果排序。

表格 29 类成员

|  |  |  |
| --- | --- | --- |
| 数据成员 | 类型 | 说明 |
| docId | int | 文档ID |
| docPath | String | 文档绝对路径 |
| content | String | 文档内容 |
| termPostingMapping | Map<AbstractTerm, AbstractPosting> | 词项到Posting的映射 |
| score | double | 文档得分 |

表格 30 类方法

|  |  |  |  |
| --- | --- | --- | --- |
| 方法 | 参数 | 返回类型 | 功能描述 |
| Hit() | 无 | - | 无参构造函数 |
| Hit(int docId, String docPath) | docId: 文档ID, docPath: 文档路径 | - | 有参构造函数 |
| Hit(int docId, String docPath, Map<AbstractTerm, AbstractPosting> termPostingMapping) | docId: 文档ID, docPath: 文档路径, termPostingMapping: 词项映射 | - | 有参构造函数 |
| getDocId() | 无 | int | 获取文档ID |
| getDocPath() | 无 | String | 获取文档路径 |
| getContent() | 无 | String | 获取文档内容 |
| setContent(String content) | content: 文档内容 | void | 设置文档内容 |
| getScore() | 无 | double | 获取文档得分 |
| setScore(double score) | score: 得分 | void | 设置文档得分 |
| getTermPostingMapping() | 无 | Map<AbstractTerm, AbstractPosting> | 获取词项映射 |
| toString() | 无 | String | 返回Hit的字符串表示 |
| compareTo(AbstractHit o) | o: 比较对象 | int | 比较两个Hit的得分 |

1. **IndexSearcher 类**

使用倒排索引实现搜索功能，支持单词项搜索，支持不同逻辑组合的双词项搜索：AND/OR/ADJACENT。

表格 31 类成员

|  |  |  |
| --- | --- | --- |
| 数据成员 | 类型 | 说明 |
| index | AbstractIndex | 搜索使用的索引 |

表格 32 类方法

|  |  |  |  |
| --- | --- | --- | --- |
| 方法 | 参数 | 返回类型 | 功能描述 |
| open(String indexFile) | indexFile: 索引文件 | void | 打开索引文件 |
| search(AbstractTerm queryTerm, Sort sorter) | queryTerm: 查询词项, sorter: 排序器 | AbstractHit[] | 单词搜索 |
| search(AbstractTerm queryTerm1, AbstractTerm queryTerm2, Sort sorter, LogicalCombination combine) | queryTerm1: 查询词项1, queryTerm2: 查询词项2, sorter: 排序器, combine: 逻辑组合方式 | AbstractHit[] | 双词搜索 |

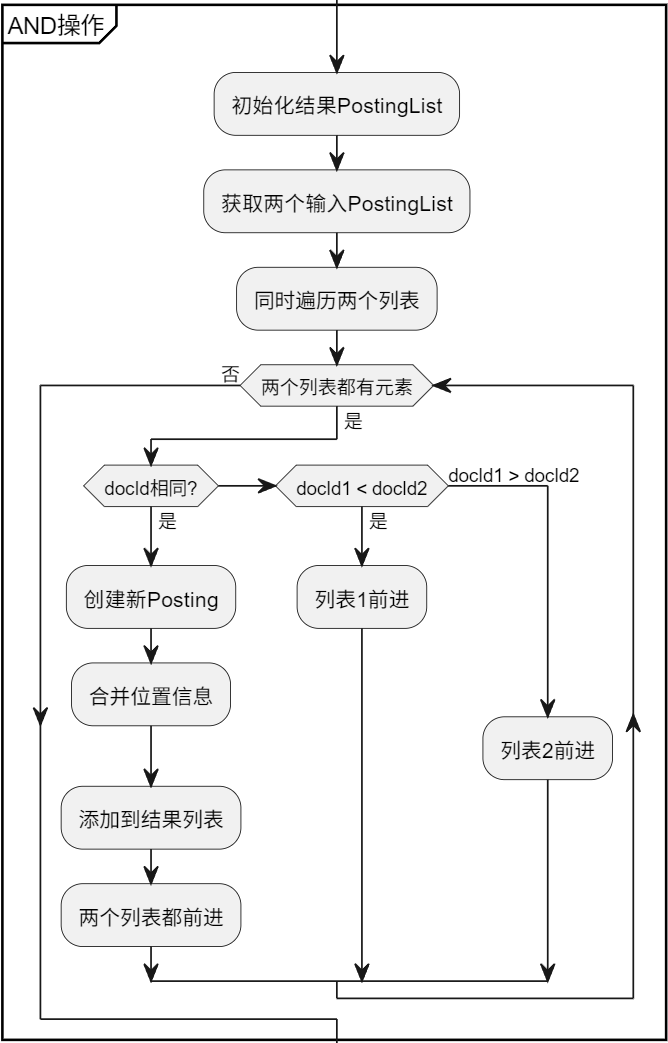


图 9 AND逻辑操作算法流程

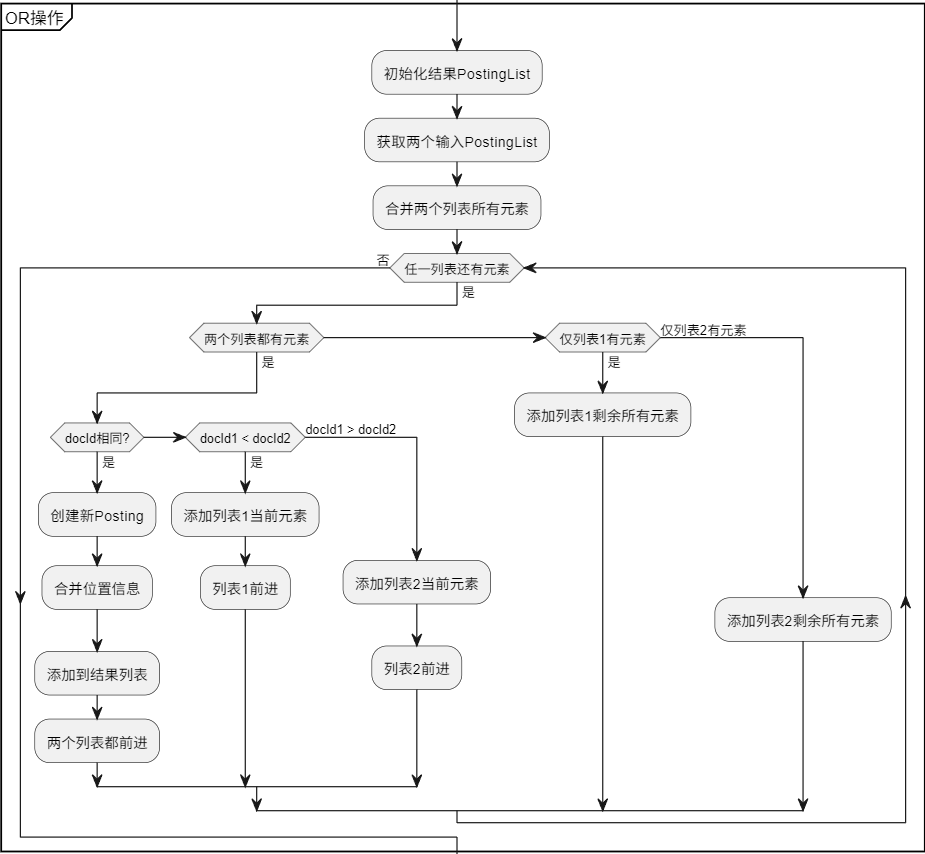


图 10 OR逻辑操作算法流程

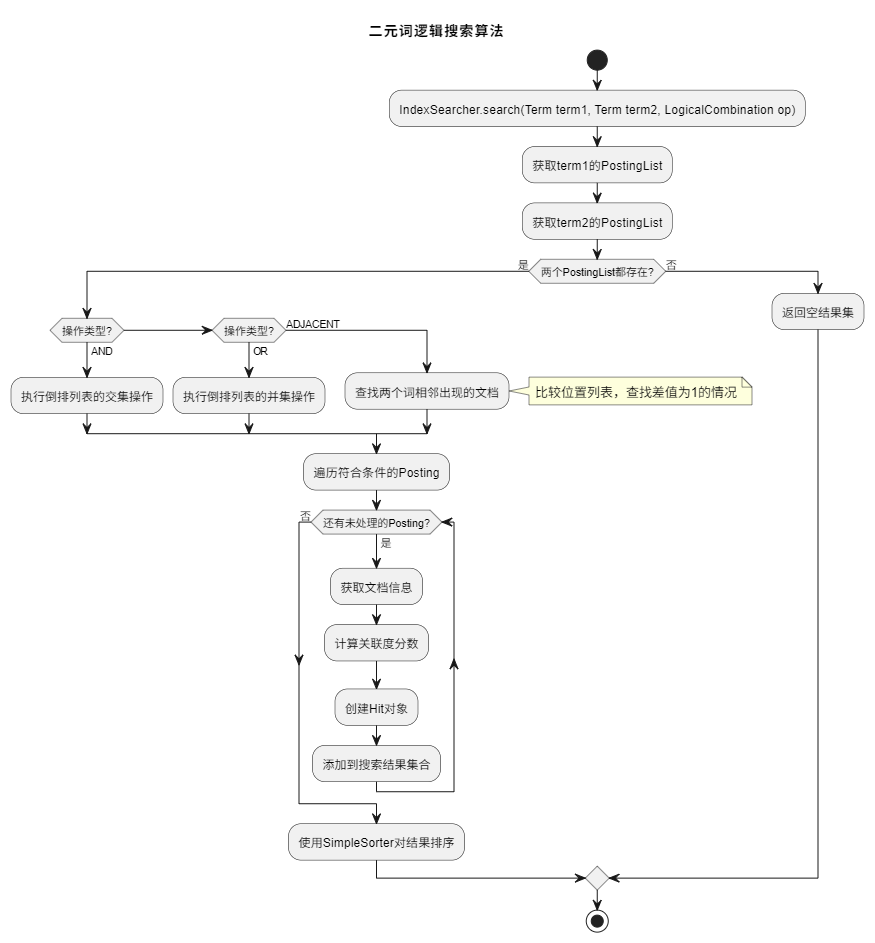
**进阶功能**短语搜索与OR操作类似但不同，ADJACENT只返回包含指定短语的文档，而不是包含任一词项的所有文档。ADJACENT操作算法首先获取两个Term的PostingList，如果任一PostingList为空，则直接返回空结果。然后同时遍历两个PostingList，当发现相同docId的Posting时，获取两个Term在该文档中的位置列表pos1和pos2。接着通过双重循环比较位置：对于pos1中的每个位置p1，检查pos2中是否存在位置p2使得p2等于p1+1（即term2紧跟在term1之后）。如果找到这样的位置对，则表示这两个词项在文档中相邻出现，需要创建一个新的Posting，设置文档ID、词频（相邻出现的次数）和相应的位置信息，并将其添加到结果PostingList中。

图 11 双关键字搜索流程图

1. **SimpleSorter类**

实现搜索结果的排序逻辑，基于词项频率计算文档分数，按分数降序排序命中结果。

表格 33 类方法

|  |  |  |  |
| --- | --- | --- | --- |
| 方法 | 参数 | 返回类型 | 功能描述 |
| sort(List<AbstractHit> hits) | hits: 命中结果列表 | void | 对结果排序 |
| score(AbstractHit hit) | hit: 命中对象 | double | 计算命中文档得分 |

1. **工具类模块 (hust.cs.javacourse.search.util)**
2. **Config类**

表格 34 类成员

|  |  |  |
| --- | --- | --- |
| 数据成员 | 类型 | 说明 |
| PROJECT\_HOME\_DIR | String | 项目主目录 |
| INDEX\_DIR | String | 索引文件目录 |
| DOC\_DIR | String | 文档目录 |
| IGNORE\_CASE | boolean | 是否忽略大小写 |
| STRING\_SPLITTER\_REGEX | String | 分词正则表达式 |
| TERM\_FILTER\_PATTERN | String | 词项过滤正则表达式 |
| TERM\_FILTER\_MINLENGTH | int | 词项最小长度 |
| TERM\_FILTER\_MAXLENGTH | int | 词项最大长度 |

1. **FileUtil类**

表格 35 类方法

|  |  |  |  |
| --- | --- | --- | --- |
| 方法 | 参数 | 返回类型 | 功能描述 |
| read(String filePath) | filePath: 文件路径 | String | 读取文本文件内容 |
| write(String content, String filePath) | content: 内容, filePath: 文件路径 | void | 写入内容到文件 |
| list(String dirPath) | dirPath: 目录路径 | List<String> | 列出目录下所有文件 |
| list(String dirPath, String suffix) | dirPath: 目录路径, suffix: 后缀 | List<String> | 列出目录下指定后缀的文件 |

1. **StringSplitter类**

表格 36 类成员

|  |  |  |
| --- | --- | --- |
| 数据成员 | 类型 | 说明 |
| splitRegex | String | 分词正则表达式 |
| pattern | Pattern | 正则表达式模式 |
| match | Matcher | 匹配器 |

表格 37 类方法

|  |  |  |  |
| --- | --- | --- | --- |
| 方法 | 参数 | 返回类型 | 功能描述 |
| setSplitRegex(String regex) | regex: 正则表达式 | void | 设置分词正则表达式 |
| splitByRegex(String input) | input: 输入字符串 | List<String> | 分割字符串为单词列表 |

1. **StopWords 类**

表格 38 类成员

|  |  |  |
| --- | --- | --- |
| 数据成员 | 类型 | 说明 |
| STOP\_WORDS | String[] | 停用词数组 |

1. **软件开发**

表格 39 开发环境

|  |  |
| --- | --- |
| 处理器 | 13th Gen Intel(R) Core(TM) i5-13400F 2.50 GHz |
| 版本 | Windows 11 专业版 |
| 版本号 | 24H2 |
| 操作系统版本 | 26100.4061 |
| IDE | IntelliJ IDEA 2025.1 (Community Edition) |
| JDK | Microsoft OpenJDK 17.0.5 |

本项目在Windows 11下使用集成开发环境IntelliJ IDEA 2025.1 (Community Edition)开发完成，实验中使用的JDK版本为Microsoft OpenJDK 17.0.5。程序的运行与调试过程均通过IDE提供的功能完成。

1. **软件测试**

对照题目要求，构造测试例，给出程序界面截图，举证题目要求的功能（以及自行补充的功能）已实现。

分析测试效果。

注意：已实现但未在报告中主动举证的功能可能被当作没有实现。

1. **自动测试**

测试结果如下：

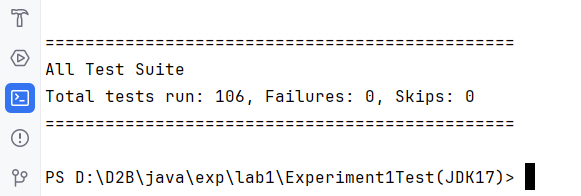


图 12 自动测试结果

成功通过106个自动测试，表明程序各模块基本功能良好。

1. **索引构建测试**

功能：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，索引对象可序列化到文件，文件也可反序列化成索引对象。

1. **构建未经过滤的索引**

构建文档时不对三元组流采取过滤策略，即不用Filter类对termTupleStream进行装饰，得到完整的未经过滤的索引。将其打印在控制台，如图所示：

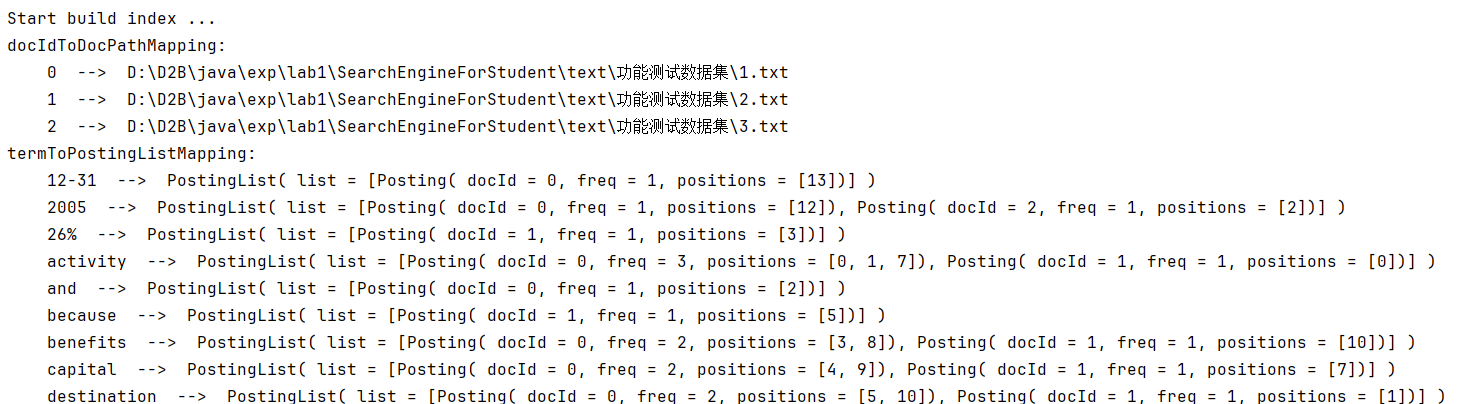


图 13 测试1输出

1. **构建经过滤的索引**

测试2同样使用功能测试数据集，但在构建文档时对三元组流采取过滤策略，使用三个Filter类（StopWordFilter、NumericFilter和LengthFilter）同时对termTupleStream进行装饰，得到经过过滤的索引。结果如下：

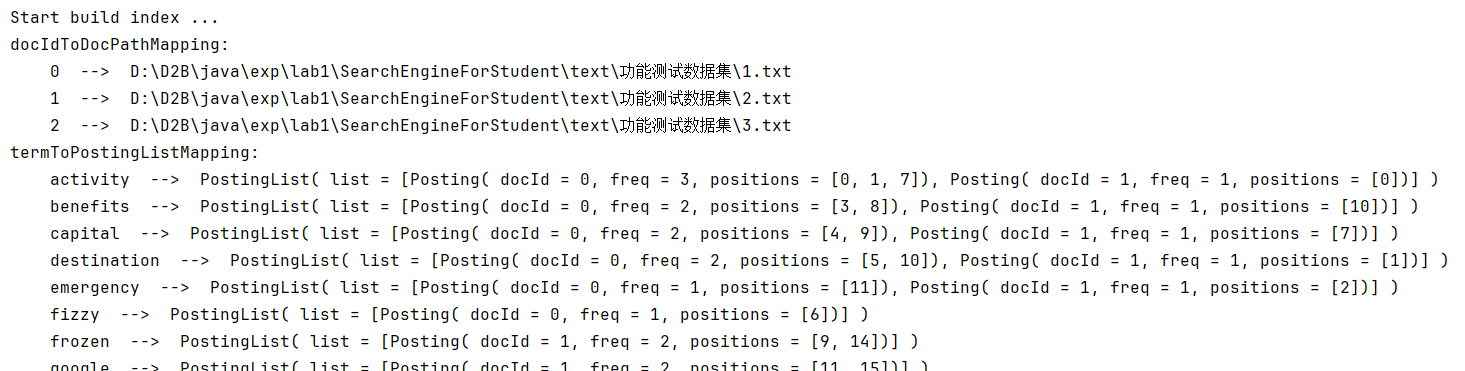


图 14 测试2输出

由测试结果可以看到，经过过滤的索引中不符合要求的单词已被过滤，例如测试1中的"12-31"、"2005"、"is"、"a"等停用词或包含数字的词已不再显示。控制台中只打印了符合过滤要求的单词，验证了构建经过过滤的索引功能正常工作。

1. **索引对象序列化和反序列化**

测试3使用功能测试数据集，验证索引对象的序列化与反序列化功能。从测试1和测试2的结果可以看到，已成功将索引保存至二进制文件index.dat中，使用工具可查看文件数据。



图 15 序列化对象二进制数据

接下来对文件反序列化成索引对象进行测试，结果如下：

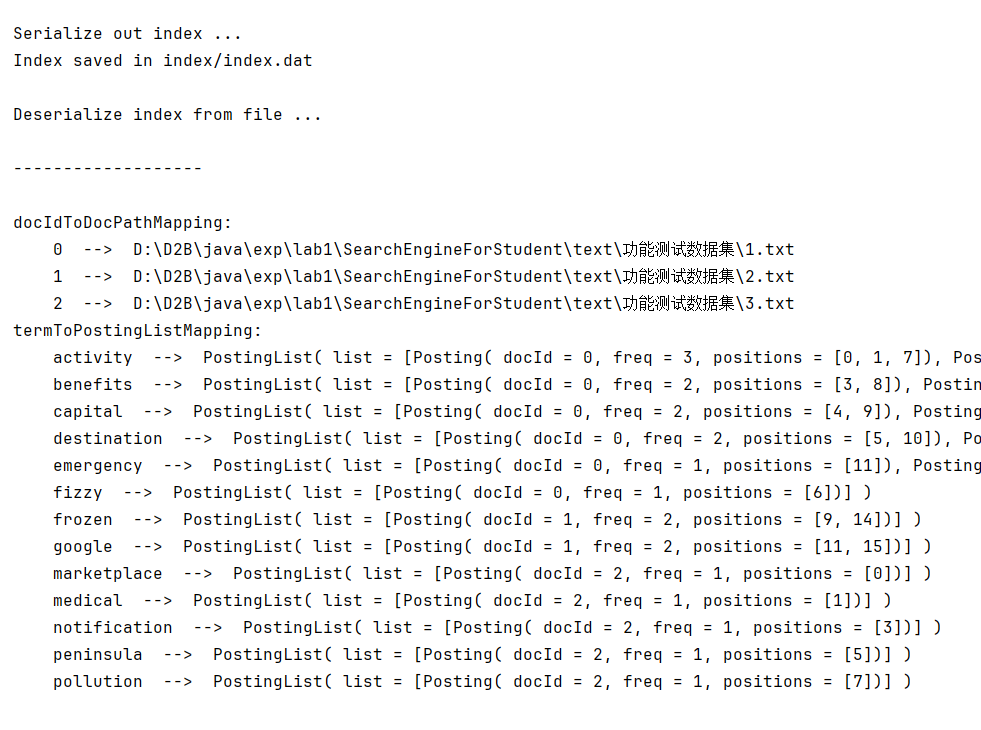


图 16 反序列化输出

由测试结果可以看到，文件反序列化成的索引对象被打印在控制台上，其结果与测试2中由文本文件构建出的经过过滤的索引完全一致。这证明了索引构建时序列化与反序列化的功能正常工作，能够正确地保存和恢复索引数据。

1. **单词检索测试**

功能：基于构建好的索引，实现单个搜索关键词的全文检索。

1. **查询在索引中存在的单个单词**

使用功能测试数据集，查询在索引中存在的单个单词"activity"，并将所有查询结果打印在控制台。测试结果如下：

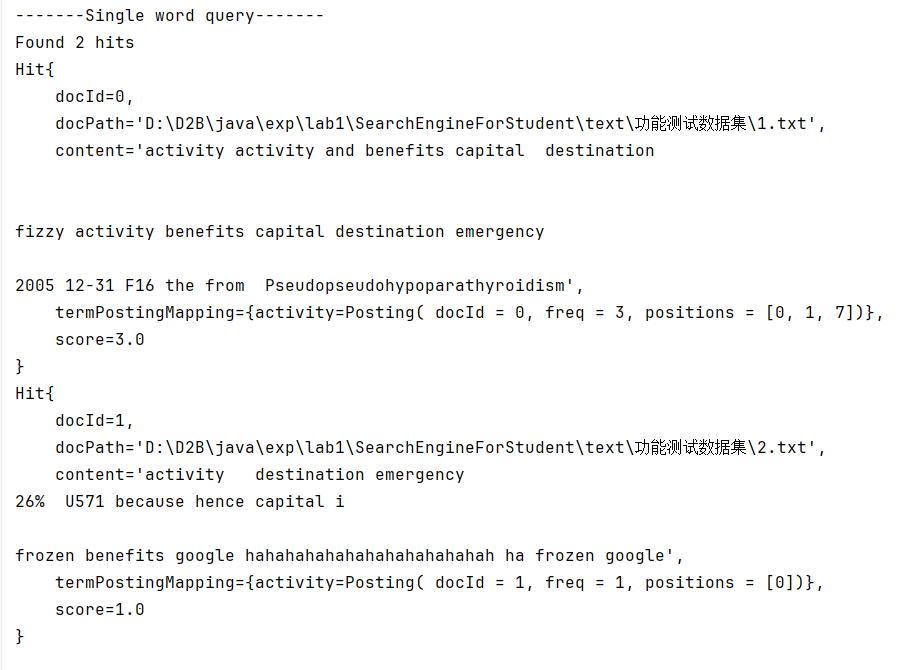


图 17 测试1输出

1. **查询在索引中不存在/被过滤的单个单词**

使用功能测试数据集，但查询了在索引中**被过滤**的单个单词"luxembourg"。测试结果如下：

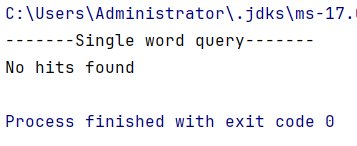


图 18 测试2输出

1. **双单词查询**

功能：基于构建好的索引，实现二个搜索关键词的与查询，返回同时包含这二个单词的文档集合。

1. **查询两个相同的单词**

使用真实测试数据集，查询两个相同的单词"wales"，以检验分数计算和查询结果是否正常。测试结果如下：

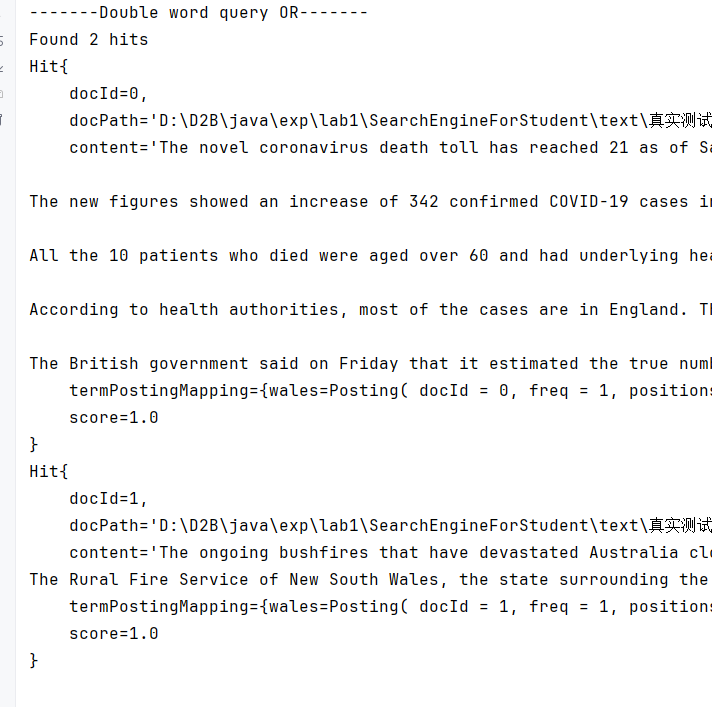


图 19 测试输出

从测试结果可以看到，在查询两个相同的单词时，系统正确地返回了包含该单词的文档。虽然查询了两次相同的单词，但没有因为查询了两次相同的单词而导致分数异常。这说明程序的检索算法设计严谨，能够很好地应对这种特殊情况下的查询需求。

1. **查询在索引中不存在/被过滤的单词**

使用真实测试数据集，使用AND模式查询单词"wales"和"luxembourg"，以检验分数计算和查询结果是否正常。测试结果如下：

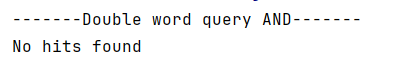


图 20 测试输出

测试结果正确。

1. **两个单词的或查询**

使用真实测试数据集，查询不在同一文档中的两个单词"wales"和"television"，并将所有查询结果打印在控制台。测试结果如下：

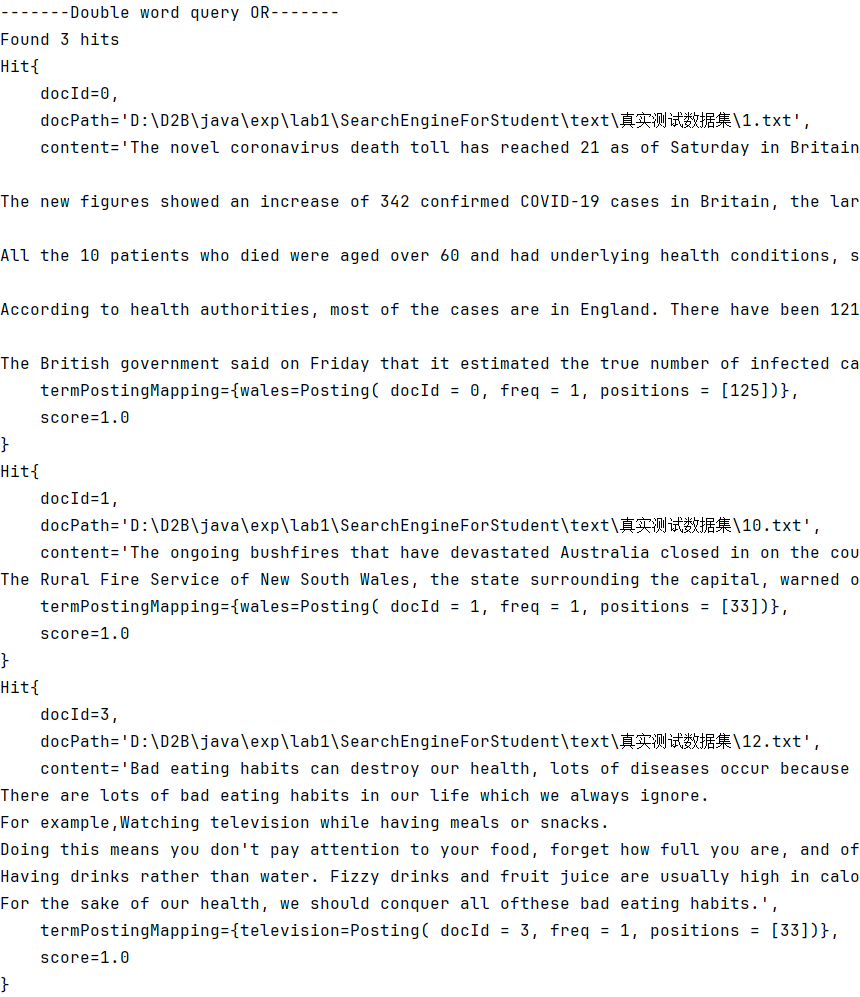


图 21 测试输出

从测试结果可以看到，对于不在同一文档中的两个单词，或查询成功地返回了包含任一单词的所有文档,并按照相关度从高到低排序。验证了系统能够正确区分并实现不同类型的查询功能。

1. **进阶功能：双单词短语查询**
2. **查询位置相邻的2个单词**

使用真实测试数据集，查询在同一文档中位置相邻的两个单词"watching television"，并将所有查询结果打印在控制台。测试结果如下：

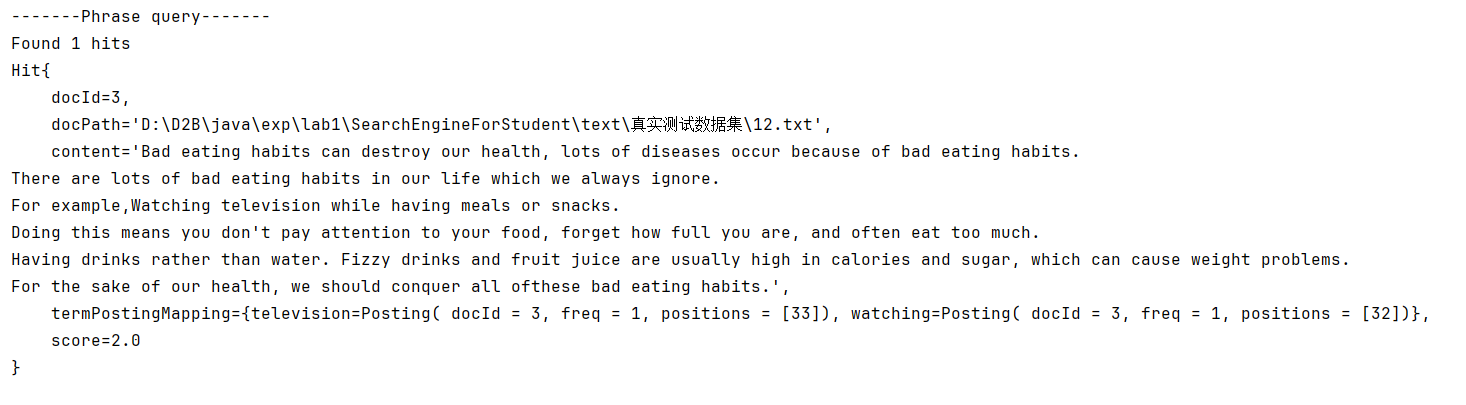


图 22 测试输出

从测试结果可以看到，系统成功找到了包含短语"watching television"的文档，即两个单词"watching"和"television"在文档中相邻出现的情况。

1. **查询位置不相邻的2个单词**

使用真实测试数据集，查询在同一文档中位置相邻的两个单词"having television"，并将所有查询结果打印在控制台。测试结果如下：

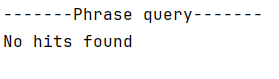


图 23 测试输出

测试结果可以看到，虽然"having"和"television"这两个单词在同一文档中出现，但由于它们在文档中的位置不相邻，不构成短语，因此系统正确地提示未找到。说明短语查询算法的正确，只有当两个单词在文档中紧密相邻且顺序一致时，才会被识别为符合短语查询条件。

1. **特点与不足**
2. **技术特点**
3. **优秀的面向对象设计**

本项目以面向对象的设计理念，通过抽象类和接口清晰地分离了设计与实现。这种架构使得各个组件之间松耦合，同时保持了高内聚性。项目中的AbstractIndex、AbstractDocument等抽象类提供了统一的接口规范，而具体实现类则负责实际功能实现，体现了"面向接口编程"的思想。这种设计方式不仅提高了代码的可维护性，也为后续功能扩展提供了良好的基础架构。

1. **流水线式文本处理**

本项目实现了一种流水线式的文本处理机制，采用装饰者模式构建了灵活的过滤器链。通过AbstractTermTupleStream作为基础接口，结合TermTupleScanner和各种过滤器（如StopWordTermTupleFilter、LengthTermTupleFilter等），形成了一个可配置的文本处理流水线。这种设计允许用户根据需求动态组合不同的过滤器，增强了系统的灵活性和可扩展性，同时也使得文本处理逻辑清晰直观。

1. **高效的索引结构**

项目采用了倒排索引作为核心数据结构，这也是现代搜索引擎的基础架构。每个词项映射到一个包含文档ID、频率和位置信息的发布列表，支持快速查询和复杂的搜索逻辑。特别是通过保存词项的位置信息，系统能够支持短语查询等高级功能。同时，项目还实现了索引的序列化和反序列化功能，使索引可以持久化存储，避免了每次使用时重建索引的开销。

1. **可扩展的查询逻辑**

查询系统设计得非常灵活，支持单词查询、逻辑组合查询（AND、OR）以及相邻词查询（ADJACENT）。AbstractIndexSearcher提供了统一的查询接口，而不同的查询逻辑被封装在相应的方法中。系统采用Hit对象表示查询结果，并通过Sorter接口实现了可定制的结果排序策略，使得添加新的查询类型或优化现有查询算法变得简单直观。

1. **多种设计模式的巧妙结合**

本搜索引擎项目在系统设计上运用了丰富的设计模式，包括装饰者模式、迭代器模式、策略模式等，显著地提高了程序的可读性与健壮性，为程序复杂模块的扩展提供了便捷。如在文本处理过程中采用装饰者模式，通过AbstractTermTupleStream抽象类构建了灵活的过滤器链，使不同的过滤器可以按需组合，从而实现了可配置的文本处理流水线；查询结果排序时，采用策略模式通过AbstractHit和Sorter接口实现了可替换的排序算法，使系统能够根据不同需求灵活切换排序策略，而无需修改核心代码；在创建文档和索引对象时，运用工厂模式通过DocumentBuilder和IndexBuilder类集中管理对象的创建过程，既简化了客户端代码，又提高了对象创建的一致性和可维护性。

1. **不足和改进的建议**
2. **性能优化空间**

当前的实现在处理大规模数据集时可能存在效率问题。索引结构未进行压缩，可能导致较大的内存占用；对于复杂查询，尤其是包含多个高频词的查询，处理效率可能不够理想。可以引入索引压缩技术减少内存占用，实现增量索引更新机制，添加查询结果缓存以减少重复计算，并考虑使用跳表等高效数据结构优化查询性能。

1. **文本处理局限**

现有系统主要针对英文文本设计，对中文等其他语言的支持有限。文本处理过程中没有实现词形还原（如词干提取、词形变化处理），停用词处理也相对简单，缺乏针对特定语言的优化。建议添加中文分词能力扩大应用范围，实现词干提取和词形还原提高匹配效果，并支持多语言停用词列表

1. **过程和体会**
2. **遇到的主要问题和解决方法**
3. **对Filter的理解与实现问题**

在实现Filter功能时，我最初误以为将三种过滤功能写在一个类中即可，导致测试时找不到对应的类。仔细阅读实验要求后，发现需要分别实现三种独立的Filter类：LengthTermTupleFilter、PatternTermTupleFilter和StopWordTermTupleFilter。这种设计符合装饰者模式的思想，每个过滤器独立负责一种过滤逻辑，可以灵活组合。重新按照要求实现后，问题得到解决，也加深了对装饰者模式的理解。

1. **流处理与序列化实现困难**

对"流"的概念理解不足，对FileInputStream和InputStreamReader等流相关的API不熟悉，导致在实现Index、Posting等类的序列化接口时遇到困难。在编写writeObject和readObject方法时，对需要序列化的对象和可能抛出的异常缺乏概念。通过查阅文档和网上资料，深入学习了Java序列化机制，理解了对象持久化的原理，并成功实现了各类的序列化与反序列化功能，为索引的存储和加载奠定了基础。

1. **文件路径与编码问题**

在测试阶段发现跳过了很多测试，经排查发现是因为测试包放在了中文目录下导致路径解析问题。此外，在不同操作系统下文件路径分隔符不同也曾造成困扰。通过修改目录名为英文，并使用File.separator来确保路径跨平台兼容，解决了这一问题。

1. **课程设计的体会**

本次实验让我深刻体会到了面向对象设计的强大。通过抽象类和接口的设计，实现了系统各部分的解耦和重用。不同于以往编写的高度冗余、低健壮性的代码，这次实验中各模块间的关系清晰，功能划分明确，扩展性强。特别是通过继承和多态机制，能够在不修改原有代码的情况下增加新功能，这正是"开闭原则"的体现。

实验中运用了多种设计模式，如迭代器模式、装饰者模式和策略模式等。特别是在实现词项过滤器时，装饰者模式的应用使得过滤器可以灵活组合，大大提高了代码的可扩展性。通过这些设计模式的实践，我认识到好的设计能够大幅减少代码复杂度，提高系统的可维护性和可读性。这些模式不仅是理论知识，更是解决实际问题的有效工具。

随着项目结构越来越复杂，模块化设计的优势愈发明显。本实验将搜索引擎分为索引、解析、查询等多个模块，每个模块负责特定功能，可以独立开发和测试。这种模块化结构不仅提高了开发效率，也使得系统更容易维护和扩展。当需要修改某一功能时，只需关注相应模块，而不会影响其他部分。

通过实验，我将课堂上学到的Java理论知识应用到实际项目中，深刻理解了倒排索引的原理和实现方法。在实践过程中遇到的问题和解决的过程也加深了对理论知识的理解。特别是对于序列化、集合框架、IO流等知识点，通过亲手实现才真正掌握了其使用方法和适用场景。

本次实验不仅是对Java知识的综合应用，更是对软件工程思想的深入理解。它让我认识到编程不仅是写代码，更是一种设计艺术，需要清晰的思路和合理的结构。看着系统各部分功能逐步实现，最终相互配合运行，给我带来了极大的成就感。从最初的困惑到最终的完成，这个过程虽然充满挑战，但也是一次宝贵的学习经历。通过这次实验，不仅提升了我的Java编程能力，更重要的是培养了解决复杂问题的思维方式和信心。

最后，我要衷心感谢授课老师和课程组全体成员对本次实验的精心设计和悉心指导。整个实验框架结构清晰，文档详实，为我们提供了绝佳的学习机会。从抽象类的设计到测试用例的编写，都体现了教学团队的专业素养和对学生学习效果的重视。

特别感谢老师在实验过程中的耐心解答和指导。每当遇到技术难题时，老师总能给予及时、有效的帮助，不仅解决了当前问题，更引导我们思考问题的本质，培养独立分析和解决问题的能力。课程组提供的辅助材料和实验环境也大大降低了我们的学习门槛，使我们能够专注于核心概念的理解和实现。

感谢这次宝贵的学习机会，这是我计算机专业学习道路上的一次重要实践体验。

1. **源码和说明**
2. **文件清单及其功能说明**

提交的文件目录如图所示。

SearchEngineForStudent是实验的源码工程，其中bin目录下包括.class文件，javadoc目录下包含javadoc文件，src目录下包括,java文件，text目录下包括测试集文件。

表格 40 SearchEngineForStudent工程目录

|  |  |  |
| --- | --- | --- |
| 目录 | | 内容 |
| index | | 索引生成的序列化文件 |
| javadoc | | Javadoc文档 |
| src | index  parse  query  run  util | 倒排索引结构相关  文档解析、三元组的读取和过滤  单词的查找和命中单词的排序和信息保存  测试程序  工具类 |
| text | | 待搜索的文件 |

Experiment1Test(JDK17)是自动测试文件，其中所有的.class文件已经放入betest目录下。

1. **用户使用说明书**
2. **系统要求**

Java运行环境：JRE 8或更高版本。

操作系统：Windows/Linux/MacOS。

1. **安装步骤**
2. 确保系统已安装Java17版本：

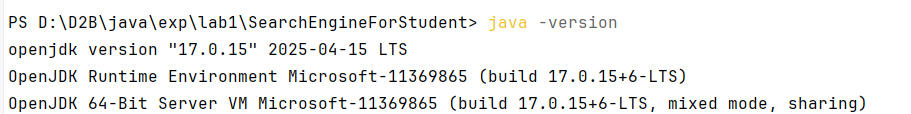


图 24 使用指令查看JDK版本

1. 下载并解压搜索引擎项目压缩包到任意目录
2. 将项目添加到IDE（如IntelliJ IDEA）中：IntelliJ IDEA: File -> Open -> 选择项目目录，确认项目结构正确，所有依赖都已正确加载。
3. **构建索引**

打开TestBuildIndex.java文件。根据需要修改待索引的文档目录路径和索引输出路径：

1. String rootDir = "text"; // 文档目录

2. String indexFile = "index/index.dat"; // 索引保存路径

运行TestBuildIndex类的main方法。程序会读取指定目录下的所有文本文件，构建索引，并序列化保存到指定路径

1. **单词搜索**

打开TestSearchIndexSingleWord.java，修改搜索关键词和索引文件路径：

1. String indexFile = "index/index.dat";

2. String queryWord = "computer"; // 修改为您要搜索的单词

运行该类的main方法执行搜索。

1. **AND逻辑搜索**

打开TestSearchIndexTwoWordAND.java，修改搜索关键词和索引文件路径：

1. String indexFile = "index/index.dat";

2. String queryWord1 = "computer";

3. String queryWord2 = "science"; // 修改为您要搜索的两个关键词

运行该类的main方法执行搜索。

1. **OR逻辑搜索**

打开TestSearchIndexTwoWordOR.java，类似上面，修改搜索关键词和索引文件路径，运行该类的main方法执行搜索

1. **相邻词搜索**

打开TestSearchIndexTwoWordADJACENT.java，类似上面，修改搜索关键词和索引文件路径，运行该类的main方法执行搜索。

1. **搜索结果说明**

搜索执行后，控制台会显示以下信息：

匹配的文档数量、每个匹配文档的详细信息（文档ID和路径、关键词在文档中的出现频率、关键词在文档中的位置、文档的相关性得分）

1. **自定义配置**

如需调整系统参数（如停用词列表、文本过滤规则等），可修改Config.java文件中的相关常量。

1. **源代码**

打印源码清单。源码关键位置要有注释。

package hust.cs.javacourse.search.index;

import java.util.ArrayList;

import java.util.List;

/\*\*

\*<pre>

\* AbstractDocument是文档对象的抽象父类.

\* 文档对象是解析一个文本文件得到结果，文档对象里面包含：

\* 文档id.

\* 文档的绝对路径.

\* 文档包含的三元组对象列表，一个三元组对象是抽象类AbstractTermTuple的子类实例

\*</pre>

\*/

public abstract class AbstractDocument {

/\*\*

\* 文档id

\*/

protected int docId;

/\*\*

\* 文档绝对路径

\*/

protected String docPath;

/\*\*

\* 文档包含的三元组列表

\*/

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

/\*\*

\* 缺省构造函数

\*/

public AbstractDocument(){

}

/\*\*

\* 构造函数

\* @param docId：文档id

\* @param docPath：文档绝对路径

\*/

public AbstractDocument(int docId, String docPath){

this.docId = docId;

this.docPath = docPath;

}

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

this.docId = docId;

this.docPath = docPath;

this.tuples = tuples;

}

/\*\*

\* 获得文档id

\* @return ：文档id

\*/

public abstract int getDocId();

/\*\*

\* 设置文档id

\* @param docId：文档id

\*/

public abstract void setDocId(int docId);

/\*\*

\* 获得文档绝对路径

\* @return ：文档绝对路径

\*/

public abstract String getDocPath();

/\*\*

\* 设置文档绝对路径

\* @param docPath：文档绝对路径

\*/

public abstract void setDocPath(String docPath);

/\*\*

\* 获得文档包含的三元组列表

\* @return ：文档包含的三元组列表

\*/

public abstract List<AbstractTermTuple> getTuples();

/\*\*

\* 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

\*/

public abstract void addTuple(AbstractTermTuple 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

\*/

public abstract boolean contains(AbstractTermTuple 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

\*/

public abstract AbstractTermTuple getTuple(int index);

/\*\*

\* 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

\*/

public abstract int getTupleSize();

/\*\*

\* 获得Document的字符串表示

\* @return ： Document的字符串表示

\*/

@Override

public abstract String toString();

}

package hust.cs.javacourse.search.index;

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

import java.io.File;

/\*\*

\* <pre>

\* AbstractDocumentBuilder是Document构造器的抽象父类.

\* Document构造器的功能应该是由解析文本文档得到的TermTupleStream，产生Document对象.

\* </pre>

\*/

public abstract class 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>

\*/

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

// /\*\*

// \* <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,可以调用AbstractDocument build(AbstractTermTupleStream termTupleStream)

// \* @param file: 给定的File对象

// \* @return ：Document对象

// \* </pre>

// \*/

/\*\*

\* <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>

\*/

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

}

package hust.cs.javacourse.search.index;

import java.util.Map;

import java.io.File;

import java.util.Set;

import java.util.TreeMap;

/\*\*

\* <pre>

\* AbstractIndex是内存中的倒排索引对象的抽象父类.

\* 一个倒排索引对象包含了一个文档集合的倒排索引.

\* 内存中的倒排索引结构为HashMap，key为Term对象，value为对应的PostingList对象.

\* 另外在AbstractIndex里还定义了从docId和docPath之间的映射关系.

\* 必须实现下面接口:

\* FileSerializable：可序列化到文件或从文件反序列化.

\* </pre>

\*/

public abstract class AbstractIndex implements FileSerializable{

/\*\*

\* 内存中的docId和docPath的映射关系, key为docId，value为对应的docPath.

\* TreeMap可以对键值排序

\*/

public Map<Integer, String> docIdToDocPathMapping = new TreeMap<>();

/\*\*

\* 内存中的倒排索引结构为HashMap，key为Term对象，value为对应的PostingList对象.

\*/

public Map<AbstractTerm, AbstractPostingList> termToPostingListMapping = new TreeMap<AbstractTerm, AbstractPostingList>();

/\*\*

\* 缺省构造函数,构建空的索引

\*/

public AbstractIndex(){

}

/\*\*

\* 返回索引的字符串表示

\* @return 索引的字符串表示

\*/

@Override

public abstract String 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

\*/

public abstract void addDocument(AbstractDocument document);

/\*\*

\* <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>

\*/

public abstract void load(File file);

/\*\*

\* <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>

\*/

public abstract void save(File file);

/\*\*

\* 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

\*/

public abstract AbstractPostingList search(AbstractTerm 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

\*/

public abstract Set<AbstractTerm> getDictionary();

/\*\*

\* <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>

\*/

public abstract void optimize();

/\*\*

\* 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

\*/

public abstract String getDocName(int docId);

}

package hust.cs.javacourse.search.index;

/\*\*

\* <pre>

\* AbstractIndexBuilder是索引构造器的抽象父类

\* 需要实例化一个具体子类对象完成索引构造的工作

\* </pre>

\*/

public abstract class AbstractIndexBuilder {

/\*\*

\* 构建索引必须解析文档构建Document对象，因此包含AbstractDocumentBuilder的子类对象

\*/

protected AbstractDocumentBuilder docBuilder;

/\*\*

\* docId计数器，每当解析一个文档并写入索引，计数器应该+1

\*/

protected int docId = 0;

public AbstractIndexBuilder(AbstractDocumentBuilder docBuilder){

this.docBuilder = 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>

\*/

public abstract AbstractIndex buildIndex(String rootDirectory);

}

package hust.cs.javacourse.search.index;

import java.util.ArrayList;

import java.util.List;

/\*\*

\* <pre>

\* AbstractPosting是Posting对象的抽象父类.

\* Posting对象代表倒排索引里每一项， 一个Posting对象包括:

\* 包含单词的文档id.

\* 单词在文档里出现的次数.

\* 单词在文档里出现的位置列表（位置下标不是以字符为编号，而是以单词为单位进行编号.

\* 必须实现下面二个接口:

\* Comparable：可比较大小（按照docId大小排序）,

\* 当检索词为二个单词时，需要求这二个单词对应的PostingList的交集,

\* 如果每个PostingList按docId从小到大排序，可以提高求交集的效率.

\* FileSerializable：可序列化到文件或从文件反序列化

\* </pre>

\*/

public abstract class AbstractPosting implements Comparable<AbstractPosting>, FileSerializable {

/\*\*

\* 包含单词的文档id

\*/

protected int docId;

/\*\*

\* 单词在文档里出现的次数

\*/

protected int freq;

/\*\*

\* 单词在文档里出现的位置列表（以单词为单位进行编号，如第1个单词，第2个单词，...), 单词可能在文档里出现多次，因此需要一个List来保存

\*/

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

/\*\*

\* 缺省构造函数

\*/

public AbstractPosting(){

}

/\*\*

\* 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 AbstractPosting(int docId, int freq, List<Integer> positions){

this.docId = docId;

this.freq = freq;

this.positions = 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 abstract boolean equals(Object obj) ;

/\*\*

\* 返回Posting的字符串表示

\* @return 字符串

\*/

@Override

public abstract String toString() ;

/\*\*

\* 返回包含单词的文档id

\* @return ：文档id

\*/

public abstract int getDocId();

/\*\*

\* 设置包含单词的文档id

\* @param docId：包含单词的文档id

\*/

public abstract void setDocId(int docId);

/\*\*

\* 返回单词在文档里出现的次数

\* @return ：出现次数

\*/

public abstract int getFreq();

/\*\*

\* 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

\*/

public abstract void setFreq(int freq);

/\*\*

\* 返回单词在文档里出现的位置列表

\* @return ：位置列表

\*/

public abstract List<Integer> getPositions();

/\*\*

\* 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

\*/

public abstract void setPositions(List<Integer> 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 abstract int compareTo(AbstractPosting o);

/\*\*

\* 对内部positions从小到大排序

\*/

public abstract void sort();

}

package hust.cs.javacourse.search.index;

import java.util.ArrayList;

import java.util.List;

/\*\*

\* <pre>

\* AbstractPostingList是所有PostingList对象的抽象父类.

\* PostingList对象包含了一个单词的Posting列表.

\* 必须实现下面接口:

\* FileSerializable：可序列化到文件或从文件反序列化.

\* </pre>

\*/

public abstract class AbstractPostingList implements FileSerializable{

/\*\*

\* Posting列表，Posting必须是AbstractPosting子类型对象

\*/

protected List<AbstractPosting> list = new ArrayList<AbstractPosting>();

/\*\*

\* 添加Posting,要求不能有内容重复的posting

\* @param posting：Posting对象

\*/

public abstract void add(AbstractPosting posting);

/\*\*

\* 获得PosingList的字符串表示

\* @return ： PosingList的字符串表示

\*/

@Override

public abstract String toString();

/\*\*

\* 添加Posting列表,,要求不能有内容重复的posting

\* @param postings：Posting列表

\*/

public abstract void add(List<AbstractPosting> postings);

/\*\*

\* 返回指定下标位置的Posting

\* @param index ：下标

\* @return： 指定下标位置的Posting

\*/

public abstract AbstractPosting get(int index);

/\*\*

\* 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

\*/

public abstract int indexOf(AbstractPosting posting);

/\*\*

\* 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

\*/

public abstract int indexOf(int docId);

/\*\*

\* 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

\*/

public abstract boolean contains(AbstractPosting posting);

/\*\*

\* 删除指定下标的Posting对象

\* @param index：指定的下标

\*/

public abstract void remove(int index);

/\*\*

\* 删除指定的Posting对象

\* @param posting ：定的Posting对象

\*/

public abstract void remove(AbstractPosting posting);

/\*\*

\* 返回PostingList的大小，即包含的Posting的个数

\* @return ：PostingList的大小

\*/

public abstract int size();

/\*\*

\* 清除PostingList

\*/

public abstract void clear();

/\*\*

\* PostingList是否为空

\* @return 为空返回true;否则返回false

\*/

public abstract boolean isEmpty();

/\*\*

\* 根据文档id的大小对PostingList进行从小到大的排序

\*/

public abstract void sort();

}

package hust.cs.javacourse.search.index;

/\*\*

\* <pre>

\* AbstractTerm是Term对象的抽象父类.

\* Term对象表示文本文档里的一个单词.

\* 必须实现下面二个接口:

\* Comparable：可比较大小（字典序）,为了加速检索过程，字典需要将单词进行排序.

\* FileSerializable：可序列化到文件或从文件反序列化.

\* </pre>

\*/

public abstract class AbstractTerm implements Comparable<AbstractTerm>, FileSerializable{

/\*\*

\* Term内容

\*/

protected String content;

/\*\*

\* 缺省构造函数

\*/

public AbstractTerm(){

}

/\*\*

\* 因为要作为HashMap里面的key，因此必须要覆盖hashCode方法

\* 返回对象的HashCode

\* @return ：对象的HashCode

\*/

@Override

public int hashCode() {

return this.content.hashCode();

}

/\*\*

\* 构造函数

\* @param content ：Term内容

\*/

public AbstractTerm(String content){

this.content = 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 abstract boolean equals(Object obj) ;

/\*\*

\* 返回Term的字符串表示

\* @return 字符串

\*/

@Override

public abstract String toString() ;

/\*\*

\* 返回Term内容

\* @return Term内容

\*/

public abstract String getContent();

/\*\*

\* 设置Term内容

\* @param content：Term的内容

\*/

public abstract void setContent(String 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 abstract int compareTo(AbstractTerm o) ;

}

package hust.cs.javacourse.search.index;

import java.util.ArrayList;

import java.util.List;

/\*\*

\* <pre>

\* AbstractTermTuple是所有TermTuple对象的抽象父类.

\* 一个TermTuple对象为三元组(单词，出现频率，出现的当前位置).

\* 当解析一个文档时，每解析到一个单词，应该产生一个三元组，其中freq始终为1(因为单词出现了一次).

\* </pre>

\*

\*

\*/

public abstract class AbstractTermTuple {

/\*\*

\* 单词

\*/

public AbstractTerm term;

/\*\*

\* 出现次数，始终为1

\*/

public final int freq = 1;

/\*\*

\* 单词出现的当前位置

\*/

public int curPos ;

/\*\*

\* 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 abstract boolean equals(Object obj);

/\*\*

\* 获得三元组的字符串表示

\* @return ： 三元组的字符串表示

\*/

@Override

public abstract String toString();

}

package hust.cs.javacourse.search.index;

import java.io.BufferedReader;

import java.io.ObjectInputStream;

import java.io.ObjectOutputStream;

import java.io.OutputStream;

/\*\*

\* 定义文件序列化接口

\*/

public interface FileSerializable extends java.io.Serializable{

/\*\*

\* 写到二进制文件

\* @param out :输出流对象

\*/

public abstract void writeObject(ObjectOutputStream out);

/\*\*

\* 从二进制文件读

\* @param in ：输入流对象

\*/

public abstract void readObject(ObjectInputStream in);

}

/\*\*

\* hust.cs.javacourse.search.index包里定义了和倒排索引数据结构相关的抽象类，以及和索引构建相关的抽象类和接口.

\* 学生需要实现这些抽象类和接口的具体子类

\*/

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

/\*\*

\*<pre>

\* AbstractDocument的具体实现类.

\* 文档对象是解析一个文本文件得到结果，文档对象里面包含：

\* 文档id.

\* 文档的绝对路径.

\* 文档包含的三元组对象列表，一个三元组对象是抽象类AbstractTermTuple的子类实例

\*</pre>

\*/

public class Document extends AbstractDocument {

/\*\*

\* 无参构造函数

\*/

public Document() {

super();

}

/\*\*

\* 构造函数

\* 三元组列表初始化为空

\*

\* @param docId 文档id

\* @param docPath 文档的绝对路径

\*/

public Document(int docId, String docPath) {

super(docId, docPath);

}

/\*\*

\* 构造函数

\*

\* @param docId 文档id

\* @param docPath 文档的绝对路径

\* @param tuples 文档包含的三元组对象列表，一个三元组对象是抽象类AbstractTermTuple的子类实例

\*/

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 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 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 tuples;

}

/\*\*

\* 设置文档包含的三元组列表

\* @param tuples ：文档包含的三元组列表

\*/

@Override

public void addTuple(AbstractTermTuple tuple) {

if (!tuples.contains(tuple)) {

tuples.add(tuple);

}

}

/\*\*

\* 判断文档是否包含指定的三元组

\* @param tuple ：指定的三元组

\* @return ：如果包含返回true，否则返回false

\*/

@Override

public boolean contains(AbstractTermTuple tuple) {

return tuples.contains(tuple);

}

/\*\*

\* 返回指定下标位置的三元组

\* @param index ：下标

\* @return ：指定下标位置的三元组

\*/

@Override

public AbstractTermTuple getTuple(int index) {

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

return tuples.get(index);

}else {

throw new IndexOutOfBoundsException();

}

}

/\*\*

\* 返回文档包含的三元组的个数

\* @return ：文档包含的三元组的个数

\*/

@Override

public int getTupleSize() {

return tuples.size();

}

/\*\*

\* 返回文档对象的字符串表示

\* @return 字符串

\*/

@Override

public String toString() {

return "Document{ docId=" + docId + ", docPath=" + docPath + ", termTuples=" + tuples + '}';

}

}

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.AbstractTermTupleFilter;

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

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

import java.io.\*;

import java.util.Arrays;

import java.util.List;

/\*\*

\* <pre>

\* DocumentBuilder是Document构造器的具体实现父类.

\* Document构造器的功能应该是由解析文本文档得到的TermTupleStream，产生Document对象.

\* </pre>

\*/

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) {

Document doc = new Document(docId, docPath);

AbstractTermTuple tuple;

while (true){

tuple = termTupleStream.next();

if (tuple == null){

break;

}

doc.addTuple(tuple);

}

return doc;

}

/\*\*

\* <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) {

Document doc = new Document(docId, docPath);

BufferedReader br;

try {

br = new BufferedReader(new FileReader(file));

} catch (FileNotFoundException e) {

throw new RuntimeException(e);

}

TermTupleScanner termTupleScanner = new TermTupleScanner(br);

AbstractTermTupleFilter filter = new StopWordTermTupleFilter(new PatternTermTupleFilter(new LengthTermTupleFilter(termTupleScanner)));

AbstractTermTuple termTuple;

while (true) {

termTuple = filter.next();

if (termTuple == null) break;

doc.addTuple(termTuple);

}

return doc;

}

public static void main(String[] args) {

String filePath = Config.DOC\_DIR + "功能测试数据集/2.txt";

File f = new File(filePath);

AbstractDocument doc = new DocumentBuilder().build(1, filePath, f);

System.out.println(doc);

BufferedReader br;

try {

br = new BufferedReader(new FileReader(Config.DOC\_DIR + "功能测试数据集/2.txt"));

} catch (FileNotFoundException e) {

throw new RuntimeException(e);

}

TermTupleScanner termTupleScanner = new TermTupleScanner(br);

List<String> stopWords = Arrays.asList(StopWords.STOP\_WORDS);

StopWordTermTupleFilter stopWordTermTupleFilter = new StopWordTermTupleFilter(new PatternTermTupleFilter(new LengthTermTupleFilter(termTupleScanner)));

AbstractDocument doc2 = new DocumentBuilder().build(2, filePath, stopWordTermTupleFilter);

System.out.println(doc2);

}

}

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

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

import java.io.\*;

import java.util.\*;

/\*\*

\* 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() {

StringBuffer sb = new StringBuffer();

sb.append("docIdToDocPathMapping:\n");

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

sb.append("\t"+entry.getKey() + " --> " + entry.getValue() + "\n");

}

sb.append("termToPostingListMapping:\n");

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

sb.append("\t"+entry.getKey() + " --> " + entry.getValue() + "\n");

}

return sb.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) {

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

List<AbstractTermTuple> tuples = document.getTuples();

// tuples.sort((a,b)->{return a.term.getContent().compareTo(b.term.getContent());});

// 汇总位置

HashMap<AbstractTerm, List<Integer>> termToPosArrMap = new HashMap<>();

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

for (AbstractTermTuple tuple : tuples) {

posArr = termToPosArrMap.get(tuple.term);

if (posArr == null) {

posArr = new ArrayList<>();

}

posArr.add(tuple.curPos);

termToPosArrMap.put(tuple.term, posArr);

}

// 添加

termToPosArrMap.forEach((term, posArrList) -> {

AbstractPostingList postingList = termToPostingListMapping.get(term);

if (postingList == null) {

postingList = new PostingList();

termToPostingListMapping.put(term, postingList);

}

postingList.add(new Posting(document.getDocId(), posArrList.size(), posArrList));

});

}

/\*\*

\* <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) {

ObjectInputStream ois = null;

try {

ois = new ObjectInputStream(new FileInputStream(file));

} catch (IOException e) {

e.printStackTrace();

}

this.readObject(ois);

}

/\*\*

\* <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) {

ObjectOutputStream oos = null;

try {

oos = new ObjectOutputStream (new FileOutputStream(file));

} catch (IOException e) {

e.printStackTrace();

}

this.writeObject(oos);

}

/\*\*

\* 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) {

AbstractPostingList postingList = termToPostingListMapping.get(term);

if (postingList == null) {

return new PostingList();

}

return postingList;

}

/\*\*

\* 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 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() {

termToPostingListMapping.forEach((k, v) -> {

v.sort();

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

v.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) {

if (docIdToDocPathMapping.containsKey(docId)) {

return docIdToDocPathMapping.get(docId);

} else {

return null;

}

}

/\*\*

\* 写到二进制文件

\*

\* @param out :输出流对象

\*/

@Override

public void writeObject(ObjectOutputStream out) {

try {

out.writeObject(this.docIdToDocPathMapping);

out.writeObject(this.termToPostingListMapping);

} 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.docIdToDocPathMapping = (Map<Integer, String>) in.readObject();

this.termToPostingListMapping = (Map<AbstractTerm, AbstractPostingList>) in.readObject();

} catch (IOException e) {

e.printStackTrace();

} catch (ClassNotFoundException e) {

e.printStackTrace();

}

}

}

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

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.util.FileUtil;

import java.io.File;

import java.util.List;

/\*\*

\* <pre>

\* IndexBuilder是索引构造器的具体实现父类

\* 需要实例化一个具体子类对象完成索引构造的工作

\* </pre>

\*/

public class IndexBuilder extends AbstractIndexBuilder {

/\*\*

\* 构造函数

\*

\* @param docBuilder AbstractDocumentBuilder的具体子类对象

\*/

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) {

AbstractIndex index = new Index();

List<String> filePaths = FileUtil.list(rootDirectory, "txt");

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

File file = new File(filePaths.get(i));

index.addDocument(this.docBuilder.build(i, filePaths.get(i), file));

}

return index;

}

}

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.ArrayList;

import java.util.Comparator;

import java.util.List;

/\*\*

\* <pre>

\* AbstractPosting抽象类的具体实现类.

\* Posting对象代表倒排索引里每一项， 一个Posting对象包括:

\* 包含单词的文档id.

\* 单词在文档里出现的次数.

\* 单词在文档里出现的位置列表（位置下标不是以字符为编号，而是以单词为单位进行编号.

\* 实现了下面二个接口:

\* Comparable：可比较大小（按照docId大小排序）,

\* 当检索词为二个单词时，需要求这二个单词对应的PostingList的交集,

\* 如果每个PostingList按docId从小到大排序，可以提高求交集的效率.

\* FileSerializable：可序列化到文件或从文件反序列化

\* </pre>

\*/

public class Posting extends AbstractPosting {

/\*\*

\* 无参构造函数

\* docId为-1，freq为0，positions为空列表。

\*/

public Posting() {

this.docId = -1;

this.freq = 0;

this.positions = new ArrayList<>();

}

/\*\*

\* 构造函数

\*

\* @param docId 文档ID，类型为int

\* @param freq 频率，类型为int

\* @param positions 出现位置列表，类型为List<Integer>

\*/

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

this.docId = docId;

this.freq = freq;

this.positions = 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(this == obj) return true;

if(obj.getClass() != this.getClass()) return false;

Posting posting = (Posting) obj;

return this.docId == posting.docId && this.freq == posting.freq && this.positions.size()==posting.positions.size() && this.positions.containsAll(posting.positions) ;

}

/\*\*

\* 返回Posting的字符串表示

\* @return 字符串

\*/

@Override

public String toString() {

return "Posting( docId = " + docId + ", freq = " + freq + ", positions = " + positions + ")";

}

/\*\*

\* 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) {

return this.docId - o.getDocId();

}

/\*\*

\* 对内部positions从小到大排序

\*/

@Override

public void sort() {

this.positions.sort(Comparator.comparingInt(Integer::intValue));

}

/\*\*

\*将 Posting对象的docId, freq和positions字段序列化到流中

\* @param out :输出流对象

\*/

@Override

public void writeObject(ObjectOutputStream out) {

try {

out.writeObject(this.docId);

out.writeObject(this.freq);

out.writeObject(this.positions);

} catch (IOException e) {

e.printStackTrace();

}

}

/\*\*

\* 从流中读取docId, freq和positions字段

\* @param in :输入流对象

\*/

@Override

public void readObject(ObjectInputStream in) {

try {

this.docId = (int)in.readObject();

this.freq = (int)in.readObject();

this.positions = (List<Integer>)in.readObject();

} catch (IOException e) {

e.printStackTrace();

} catch (ClassNotFoundException e) {

e.printStackTrace();

}

}

}

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.Comparator;

import java.util.List;

/\*\*

\* <pre>

\* AbstractPostingList的具体实现类.

\* PostingList对象包含了一个单词的Posting列表.

\* 实现了下面接口:

\* FileSerializable：可序列化到文件或从文件反序列化.

\* </pre>

\*

\*/

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.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() {

return "PostingList( list = " + list + " )";

}

/\*\*

\* 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) {

for (AbstractPosting posting : postings) {

this.add(posting);

}

}

/\*\*

\* 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) {

return this.list.get(index);

}

/\*\*

\* 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) {

return this.list.indexOf(posting);

}

/\*\*

\* 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) {

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() {

this.list.sort(new Comparator<AbstractPosting>() {

@Override

public int compare(AbstractPosting o1, AbstractPosting o2) {

return o1.compareTo(o2);

}

});

}

/\*\*

\* 根据文档id的大小对PostingList进行从小到大的排序

\* @param comparator ：比较器

\*/

@Override

public void writeObject(ObjectOutputStream out) {

try {

out.writeObject(this.list);

} catch (IOException e) {

e.printStackTrace();

}

}

/\*\*

\* 反序列化，从流中读取list字段

\* @param in ：输入流对象

\*/

@Override

public void readObject(ObjectInputStream in) {

try {

this.list = (List<AbstractPosting>) in.readObject();

} catch (IOException e) {

e.printStackTrace();

} catch (ClassNotFoundException e) {

e.printStackTrace();

}

}

}

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

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

import java.io.\*;

import java.util.Objects;

/\*\*

\* <pre>

\* AbstractTerm类的具体实现类.

\* Term对象表示文本文档里的一个单词.

\* 实现了下面二个接口:

\* Comparable：可比较大小（字典序）,为了加速检索过程，字典需要将单词进行排序.

\* FileSerializable：可序列化到文件或从文件反序列化.

\* </pre>

\*/

public class Term extends AbstractTerm {

/\*\*

\* 无参构造函数，后续可通过setter方法指定内容

\*/

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 (this == obj) return true;

if (!(obj instanceof Term term)) return false;

return Objects.equals(this.content, term.content);

}

/\*\*

\* 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) {

return this.content.compareTo(o.getContent());

}

/\*\*

\* 序列化Term对象到输出流

\* @param out ：输出流

\*/

@Override

public void writeObject(ObjectOutputStream out) {

try {

out.writeObject(this.content);

} catch (IOException e) {

e.printStackTrace();

}

}

/\*\*

\* 反序列化Term对象

\* @param in ：输入流

\*/

@Override

public void readObject(ObjectInputStream in) {

try {

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

} catch (IOException | ClassNotFoundException e) {

e.printStackTrace();

}

}

}

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

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

import java.util.Objects;

/\*\*

\* <pre>

\* AbstractTermTuple的具体子类.

\* 一个TermTuple对象为三元组(单词，出现频率，出现的当前位置).

\* 当解析一个文档时，每解析到一个单词，应该产生一个三元组，其中freq始终为1(因为单词出现了一次).

\* </pre>

\*/

public class TermTuple extends AbstractTermTuple {

/\*\*

\* 无参构造函数

\*/

public TermTuple(){

super();

}

/\*\*

\* 构造函数

\*

\* @param word 单词

\* @param curPos 位置

\*/

public TermTuple(String word, int curPos) {

this.term = new Term(word);

this.curPos = curPos;

}

/\*\*

\* 判断二个TermTuple内容是否相同

\* @param obj ：要比较的另外一个TermTuple

\* @return 如果内容相等返回true，否则返回false

\*/

@Override

public boolean equals(Object obj) {

if (this == obj) return true;

if (obj.getClass() != TermTuple.class) return false;

TermTuple termTuple = (TermTuple) obj;

return Objects.equals(this.term, termTuple.term) && this.curPos == termTuple.curPos;

}

/\*\*

\* 返回TermTuple的字符串表示

\* @return 字符串

\*/

@Override

public String toString() {

return "TermTuple( term = " + term + ", curPos = " + curPos + ")";

}

}

/\*\*

\* 对hust.cs.javacourse.search.index包里定义的抽象类和接口的具体实现放在这个包里。impl(implementation)

\*/

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

/\*\*

\* <pre>

\* 抽象类AbstractTermTupleFilter类型是AbstractTermTupleStream的子类,里面包含另一个

\* AbstractTermTupleStream对象作为输入，并对输入的AbstractTermTupleStream进行过滤,

\* 例如过滤掉所有停用词（the，is are...)对应的三元组

\*

\* 其具体子类需要重新实现next方法以过滤掉不需要的单词对应的三元组.同时可以实现多个不同的过滤器

\* 完成不同的过滤功能，多个过滤器可以形成过滤管道.

\* </pre>

\*/

public abstract class AbstractTermTupleFilter extends AbstractTermTupleStream{

/\*\*

\* Filter的输入，类型为AbstractTermTupleStream

\*/

protected AbstractTermTupleStream input;

/\*\*

\* 构造函数

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

\*/

public AbstractTermTupleFilter(AbstractTermTupleStream input){

this.input = input;

}

/\*\*

\* 实现父类AbstractTermTupleStream的close方法，关闭流

\*/

@Override

public void close(){

input.close();

}

}

package hust.cs.javacourse.search.parse;

import java.io.BufferedReader;

import java.io.IOException;

/\*\*

\* <pre>

\* AbstractTermTupleScanner是AbstractTermTupleStream的抽象子类，即一个具体的TermTupleScanner对象就是

\* 一个AbstractTermTupleStream流对象，它利用java.io.BufferedReader去读取文本文件得到一个个三元组TermTuple.

\*

\* 其具体子类需要重新实现next方法获得文本文件里的三元组

\* </pre>

\*/

public abstract class AbstractTermTupleScanner extends AbstractTermTupleStream {

/\*\*

\* input作为输入流对象，读取文本文件得到一个个三元组TermTuple

\*/

protected BufferedReader input = null;

/\*\*

\* 缺省构造函数

\*/

public AbstractTermTupleScanner() {

}

/\*\*

\* 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 AbstractTermTupleScanner(BufferedReader input) {

this.input = input;

}

/\*\*

\* 实现父类AbstractTermTupleStream的close方法，关闭流

\*/

@Override

public void close() {

try {

input.close();

} catch (IOException e) {

e.printStackTrace();

}

}

}

package hust.cs.javacourse.search.parse;

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

/\*\*

\* <pre>

\* AbstractTermTupleStream是各种TermFreqPosTupleStream对象的抽象父类

\* TermFreqPosTupleStream是三元组TermTuple流对象，包含了解析文本文件得到的三元组序列

\* </pre>

\*/

public abstract class AbstractTermTupleStream {

/\*\*

\* 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

\*/

public abstract AbstractTermTuple next() ;

/\*\*

\* 关闭流

\*/

public abstract void close();

}

/\*\*

\* hust.cs.javacourse.search.parse包里定义了文档解析、分词，单词过滤有关的抽象类.学生需要实现这些抽象类的具体子类

\*/

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

/\*\*

\* <pre>

\* 过滤掉不满足长度范围的三元组

\*

\* next方法可以过滤掉不需要的单词对应的三元组.同时可以实现多个不同的过滤器

\* 完成不同的过滤功能，多个过滤器可以形成过滤管道.

\* </pre>

\*/

public class LengthTermTupleFilter extends AbstractTermTupleFilter {

/\*\*

\* 最大长度

\*/

private int maxLength;

/\*\*

\* 最小长度

\*/

private int minLength;

/\*\*

\* 构造函数，使用配置文件中默认的最大长度和最小长度

\*

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

\*/

public LengthTermTupleFilter(AbstractTermTupleStream input) {

super(input);

this.minLength = Config.TERM\_FILTER\_MINLENGTH;

this.maxLength = Config.TERM\_FILTER\_MAXLENGTH;

}

/\*\*

\* 获得满足长度范围的下一个三元组

\* @return: 下一个三元组；如果到了流的末尾，返回null

\*/

@Override

public AbstractTermTuple next() {

AbstractTermTuple termTuple = input.next();

while (termTuple != null && (termTuple.term.getContent().length() > maxLength || termTuple.term.getContent().length() < minLength)) {

termTuple = input.next();

}

return termTuple;

}

}

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;

import java.util.regex.Pattern;

/\*\*

\* <pre>

\* 过滤掉不满足正则表达式的三元组

\*

\* next方法可以过滤掉不需要的单词对应的三元组.同时可以实现多个不同的过滤器

\* 完成不同的过滤功能，多个过滤器可以形成过滤管道.

\* </pre>

\*/

public class PatternTermTupleFilter extends AbstractTermTupleFilter {

/\*\*

\* 单词过滤的正则表达式.

\*/

private Pattern pattern;

/\*\*

\* 构造函数

\*

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

\*/

public PatternTermTupleFilter(AbstractTermTupleStream input) {

super(input);

this.pattern = Pattern.compile(Config.TERM\_FILTER\_PATTERN);

}

/\*\*

\* 获得满足正则表达式的下一个三元组

\* @return: 下一个三元组；如果到了流的末尾，返回null

\*/

@Override

public AbstractTermTuple next() {

AbstractTermTuple termTuple = input.next();

while (termTuple!=null && this.pattern != null && !(this.pattern.matcher(termTuple.term.getContent()).matches())){

termTuple = input.next();

}

return termTuple;

}

}

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

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

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

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

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

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

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

import java.io.BufferedReader;

import java.io.FileInputStream;

import java.io.FileNotFoundException;

import java.io.FileReader;

import java.util.ArrayList;

import java.util.Arrays;

import java.util.List;

/\*\*

\* <pre>

\* 过滤掉所有停用词（the，is are...)对应的三元组

\*

\* next方法可以过滤掉不需要的单词对应的三元组.同时可以实现多个不同的过滤器

\* 完成不同的过滤功能，多个过滤器可以形成过滤管道.

\* </pre>

\*/

public class StopWordTermTupleFilter extends AbstractTermTupleFilter {

/\*\*

\* 停用词列表

\*/

private List<String> stopWords;

/\*\*

\* 构造函数，使用配置中的默认停用词表

\*

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

\*/

public StopWordTermTupleFilter(AbstractTermTupleStream input){

super(input);

this.stopWords = Arrays.asList(StopWords.STOP\_WORDS);

}

/\*\*

\* 构造函数，使用自定义的停用词表

\*

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

\* @param stopWords ：Filter的输入，类型为List<String>

\*/

public StopWordTermTupleFilter(AbstractTermTupleStream input, List<String> stopWords) {

super(input);

this.stopWords = stopWords;

}

/\*\*

\* 获得不在停用词列表内的下一个三元组

\* @return: 下一个三元组；如果到了流的末尾，返回null

\*/

@Override

public AbstractTermTuple next() {

AbstractTermTuple termTuple = input.next();

while (termTuple!=null && this.stopWords != null && this.stopWords.contains(termTuple.term.getContent())){

termTuple = input.next();

}

return termTuple;

}

/\*\*

\* 模块测试

\*

\* @param args

\*/

public static void main(String[] args) {

BufferedReader br;

try {

br = new BufferedReader(new FileReader(Config.DOC\_DIR + "功能测试数据集/1.txt"));

} catch (FileNotFoundException e) {

throw new RuntimeException(e);

}

TermTupleScanner termTupleScanner = new TermTupleScanner(br);

List<String> stopWords = Arrays.asList(StopWords.STOP\_WORDS);

StopWordTermTupleFilter stopWordTermTupleFilter = new StopWordTermTupleFilter(termTupleScanner, stopWords);

while (true){

AbstractTermTuple termTuple = stopWordTermTupleFilter.next();

if (termTuple == null) break;

System.out.println(termTuple);

}

}

}

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

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

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.FileNotFoundException;

import java.io.FileReader;

import java.io.IOException;

import java.util.ArrayList;

import java.util.List;

/\*\*

\* <pre>

\* TermTupleScanner是AbstractTermTupleStream的具体实现，它利用java.io.BufferedReader去读取文本文件得到一个个三元组TermTuple.

\* </pre>

\*/

public class TermTupleScanner extends AbstractTermTupleScanner {

private int pos = -1; // 词在当前行的位置

private int offset = 0; // 当前行之前的词数，即真实位置 = offset + pos

private List<String> curWordsList;

private final StringSplitter splitter = new StringSplitter();

// 实例初始化块，设置分词正则表达式

{

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

}

/\*\*

\* 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() {

if (input == null) return null;

pos++;

while (curWordsList == null || curWordsList.isEmpty() || pos >= curWordsList.size()) {

offset += (curWordsList != null) ? curWordsList.size() : 0;

String newLine = null;

try {

newLine = input.readLine();

} catch (IOException e) {

throw new RuntimeException(e);

}

if (newLine == null) {

return null;

}

curWordsList = splitter.splitByRegex(newLine);

pos = 0;

}

return new TermTuple(curWordsList.get(pos).toLowerCase(), offset + pos);

}

/\*\*

\* 模块测试

\*

\* @param args

\*/

public static void main(String[] args) {

BufferedReader br;

try {

br = new BufferedReader(new FileReader(Config.DOC\_DIR + "功能测试数据集/1.txt"));

} catch (FileNotFoundException e) {

throw new RuntimeException(e);

}

TermTupleScanner termTupleScanner = new TermTupleScanner(br);

while (true) {

AbstractTermTuple termTuple = termTupleScanner.next();

if (termTuple == null) break;

System.out.println(termTuple);

}

}

}

/\*\*

\* 对hust.cs.javacourse.search.parse包里定义的抽象类和接口的具体实现放在这个包里。impl(implementation)

\*/

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

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

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

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

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

import java.util.List;

import java.util.Map;

import java.util.TreeMap;

/\*\*

\* <pre>

\* AbstractHit是一个搜索命中结果的抽象类. 该类子类要实现Comparable接口.

\* 实现该接口是因为需要必须比较大小，用于命中结果的排序.

\* </pre>

\*/

public abstract class AbstractHit implements Comparable<AbstractHit>{

/\*\*

\* 文档id

\*/

protected int docId;

/\*\*

\* 文档绝对路径

\*/

protected String docPath;

/\*\*

\* 文档原文内容，显示搜索结果时有用

\*/

protected String content;

/\*\*

\* 命中的单词和对应的Posting键值对，对计算文档得分有用，对于一个查询命中结果，一个term对应的是Posting而不是PostingList

\*/

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

/\*\*

\* 该命中文档的得分，文档的得分通过Sort接口计算.每个文档得分默认值为1.0

\*/

protected double score = 1.0;

/\*\*

\* 默认构造函数

\*/

public AbstractHit(){

}

/\*\*

\* 构造函数

\* @param docId : 文档id

\* @param docPath : 文档绝对路径

\*/

public AbstractHit(int docId, String docPath){

this.docId = docId;

this.docPath = docPath;

this.content = FileUtil.read(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 AbstractHit(int docId, String docPath, Map<AbstractTerm, AbstractPosting> termPostingMapping){

this.docId = docId;

this.docPath = docPath;

this.termPostingMapping.putAll(termPostingMapping);

this.content = FileUtil.read(docPath);

}

/\*\*

\* 获得文档id

\* @return ： 文档id

\*/

public abstract int getDocId();

/\*\*

\* 获得文档绝对路径

\* @return ：文档绝对路径

\*/

public abstract String getDocPath();

/\*\*

\* 获得文档内容

\* @return ： 文档内容

\*/

public abstract String getContent();

/\*\*

\* 设置文档内容

\* @param content ：文档内容

\*/

public abstract void setContent(String content);

/\*\*

\* 获得文档得分

\* @return ： 文档得分

\*/

public abstract double getScore();

/\*\*

\* 设置文档得分

\* @param score ：文档得分

\*/

public abstract void setScore(double 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

\*/

public abstract Map<AbstractTerm, AbstractPosting> getTermPostingMapping();

/\*\*

\* 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 abstract String 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 abstract int compareTo(AbstractHit o);

}

package hust.cs.javacourse.search.query;

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

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

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

/\*\*

\* <pre>

\* AbstractIndexSearcher是检索具体实现的抽象类

\* </pre>

\*/

public abstract class AbstractIndexSearcher {

/\*\*

\* 内存中的索引，子类对象被初始化时为空

\*/

//Index是AbstractIndex的具体实现类，在hust.cs.javacourse.search.index.impl里定义，但没有实现任何覆盖方法，需要学生实现

//如果学生AbstractIndex的具体实现类类名不是Index，这里需要相应修改

protected AbstractIndex index = new Index();

/\*\*

\* 多个检索词的逻辑组合

\*/

public static enum LogicalCombination{

/\*\*

\* 与,即多个检索词必须都在命中文档里出现

\*/

AND,

/\*\*

\* 或, 即任意一个检索词在命中文档里出现

\*/

OR,

/\*\*

\* 短语，即两个检索词前后相邻

\*/

ADJACENT

}

/\*\*

\* 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

\*/

public abstract void open(String 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

\*/

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

/\*\*

\*

\* 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

\*/

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

}

package hust.cs.javacourse.search.query;

import java.util.List;

/\*\*

\* Sort定义了对搜索结果排序的接口

\*/

public interface Sort {

/\*\*

\* 对命中结果集合根据文档得分排序

\* @param hits ：命中结果集合

\*/

public abstract void sort(List<AbstractHit> 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>

\*/

public abstract double score(AbstractHit hit);

}

/\*\*

\* hust.cs.javacourse.search.query包里定义了和搜索有关的抽象类和接口.学生需要实现这些抽象类和接口的具体子类.

\*/

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

/\*\*

\* <pre>

\* AbstractHit的具体子类. 实现Comparable接口.

\* </pre>

\*/

public class Hit extends AbstractHit {

/\*\*

\* 默认构造函数

\*/

public Hit() {

}

/\*\*

\* 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 : 命中的单词和对应的Posting键值对，对计算文档得分有用，对于一个查询命中结果，一个term对应的是Posting而不是PostingList

\*/

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;

}

/\*\*

\* 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() {

return "Hit{" +

"\n\tdocId=" + docId +

", \n\tdocPath='" + docPath + '\'' +

", \n\tcontent='" + content+ '\'' + // .replace("\n", "\\n")

", \n\ttermPostingMapping=" + termPostingMapping +

", \n\tscore=" + score +

"\n}";

}

/\*\*

\* 比较两个Hit对象的得分

\* @param o : 另一个Hit对象

\* @return : 返回值大于0表示当前对象得分高，返回值小于0表示当前对象得分低，返回值等于0表示相等

\*/

@Override

public int compareTo(AbstractHit o) {

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

}

}

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.io.FileNotFoundException;

import java.util.\*;

/\*\*

\* <pre>

\* IndexSearcher类实现了AbstractIndexSearcher类的抽象方法.

\* </pre>

\*/

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) {

File file = new File(indexFile);

if (!file.exists()) {

System.out.println("Index file does not exist");

}

this.index.load(file);

}

/\*\*

\* 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 = index.search(queryTerm);

if (postingList == null) {

return null;

}

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

AbstractPosting posting;

int length = postingList.size();

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

posting = postingList.get(i);

HashMap<AbstractTerm, AbstractPosting> newMap = new HashMap<>();

newMap.put(queryTerm, posting);

AbstractHit newHit = new Hit(posting.getDocId(), index.getDocName(posting.getDocId()), newMap);

newHit.setScore(sorter.score(newHit));

hits.add(newHit);

}

sorter.sort(hits);

return hits.toArray(new AbstractHit[hits.size()]);

}

/\*\*

\* 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) {

// queryTerm2 = new Term(queryTerm2.toString().toLowerCase());

// queryTerm1 = new Term(queryTerm1.toString().toLowerCase());

AbstractPostingList postingList1 = index.search(queryTerm1);

AbstractPostingList postingList2 = index.search(queryTerm2);

int length1 = postingList1.size();

int length2 = postingList2.size();

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

AbstractPosting posting;

AbstractPosting posting1;

AbstractPosting posting2;

if (combine==LogicalCombination.OR){

if (postingList1 == null && postingList2 == null) return new AbstractHit[0];

int i=0,j=0;

while (i < length1 && j < length2) {

posting1 = postingList1.get(i);

posting2 = postingList2.get(j);

Map<AbstractTerm,AbstractPosting> newMap=new HashMap<>();

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

posting = posting1;

newMap.put(queryTerm1, posting1);

i++;

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

posting = posting2;

newMap.put(queryTerm2, posting2);

j++;

}else {

posting = posting1;

newMap.put(queryTerm1, posting1);

newMap.put(queryTerm2, posting2);

i++;

j++;

}

AbstractHit newHit = new Hit(posting.getDocId(), index.getDocName(posting.getDocId()), newMap);

newHit.setScore(sorter.score(newHit));

hits.add(newHit);

}

while (i < length1 ) {

posting1 = postingList1.get(i);

Map<AbstractTerm,AbstractPosting> newMap=new HashMap<>();

newMap.put(queryTerm1, posting1);

AbstractHit newHit = new Hit(posting1.getDocId(), index.getDocName(posting1.getDocId()), newMap);

newHit.setScore(sorter.score(newHit));

hits.add(newHit);

i++;

}

while (j < length2 ) {

posting2 = postingList2.get(j);

Map<AbstractTerm,AbstractPosting> newMap=new HashMap<>();

newMap.put(queryTerm2, posting2);

AbstractHit newHit = new Hit(posting2.getDocId(), index.getDocName(posting2.getDocId()), newMap);

newHit.setScore(sorter.score(newHit));

hits.add(newHit);

j++;

}

} else if (combine==LogicalCombination.AND) {

if (postingList1 == null || postingList2 == null) return new AbstractHit[0];

int i=0,j=0;

while (i < length1 && j < length2) {

posting1 = postingList1.get(i);

posting2 = postingList2.get(j);

Map<AbstractTerm,AbstractPosting> newMap=new HashMap<>();

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

i++;

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

j++;

}else {

newMap.put(queryTerm1, posting1);

newMap.put(queryTerm2, posting2);

AbstractHit newHit = new Hit(posting1.getDocId(), index.getDocName(posting1.getDocId()), newMap);

newHit.setScore(sorter.score(newHit));

hits.add(newHit);

i++;

j++;

}

}

}else if (combine==LogicalCombination.ADJACENT){

if (postingList1 == null || postingList2 == null) return new AbstractHit[0];

int i=0,j=0;

while (i < length1 && j < length2) {

posting1 = postingList1.get(i);

posting2 = postingList2.get(j);

Map<AbstractTerm,AbstractPosting> newMap=new HashMap<>();

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

i++;

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

j++;

}else {

List<Integer> positions1 = posting1.getPositions();

List<Integer> positions2 = posting2.getPositions();

int p1=0, p2=0, pos1, pos2;

while (p1 < positions1.size() && p2 < positions1.size()) {

pos1 = positions1.get(p1);

pos2 = positions2.get(p2);

if (pos1+1 < pos2) {

p1++;

}

else if (pos1+1>pos2) {

p2++;

}

else {

newMap.put(queryTerm1, posting1);

newMap.put(queryTerm2, posting2);

AbstractHit newHit = new Hit(posting1.getDocId(), index.getDocName(posting1.getDocId()), newMap);

newHit.setScore(sorter.score(newHit));

hits.add(newHit);

i++;

j++;

break;

}

}

}

}

}

sorter.sort(hits);

return hits.toArray(new AbstractHit[0]);

}

}

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.List;

import java.util.Map;

/\*\*

\* <pre>

\* SimpleSorter类实现了Sort接口的抽象方法.

\* 实现了简单的降序排列。

\* </pre>

\*/

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) {

hits.sort((a,b)->{

return (int)(-score(a)+score(b));

});

}

/\*\*

\* <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 score = 0;

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

score +=entry.getValue().getFreq();

}

return score;

}

}

/\*\*

\* 对hust.cs.javacourse.search.query包里定义的抽象类和接口的具体实现放在这个包里。impl(implementation)

\*/

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

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

public class RenderOutput {

public static void renderHits(AbstractHit[] hits){

if (hits== null || hits.length == 0){

System.out.println("No hits found");

return;

}

System.out.println("Found " + hits.length + " hits");

for(AbstractHit hit : hits){

System.out.println(hit);

}

}

}

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) {

/\*\*\*\*\*\* PARAMETER \*\*\*\*\*\*\*\*\*/

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

String indexFile = "index/index.dat"; // 索引保存路径

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

AbstractDocumentBuilder documentBuilder = new DocumentBuilder();

AbstractIndexBuilder indexBuilder = new IndexBuilder(documentBuilder);

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

AbstractIndex index = indexBuilder.buildIndex(rootDir);

index.optimize();

System.out.println(index); //控制台打印index的内容

System.out.println("Finish build index ...\n\n");

//序列化

System.out.println("Serialize out index ...");

System.out.println("Index saved in " + indexFile);

index.save(new File(indexFile)); //索引保存到文件

//测试从文件读取

System.out.println("\nDeserialize index from file ...");

AbstractIndex index2 = new Index(); //创建一个空的index

index2.load(new File(indexFile)); //从文件加载对象的内容

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

System.out.println(index2); //控制台打印index2的内容

}

}

package hust.cs.javacourse.search.run;

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

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

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;

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

import javax.swing.plaf.nimbus.AbstractRegionPainter;

/\*\*

\* 测试搜索: 单关键字

\*/

public class TestSearchIndexSingleWord {

/\*\*

\* 搜索程序入口

\* @param args ：命令行参数

\*/

public static void main(String[] args){

/\*\*\*\*\*\* PARAMETER \*\*\*\*\*\*\*\*\*/

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

String queryWord = "luxembourg"; // 修改为您要搜索的单词

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Sort simpleSorter = new SimpleSorter();

AbstractIndexSearcher searcher = new IndexSearcher();

searcher.open(indexFile);

System.out.println("-------Single word query-------");

AbstractHit[] hits = searcher.search(new Term(queryWord), simpleSorter);

RenderOutput.renderHits(hits);

}

}

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.AbstractIndexSearcher.LogicalCombination;

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：双关键字相连

\*/

public class TestSearchIndexTwoWordADJACENT {

/\*\*

\* 搜索程序入口

\* @param args ：命令行参数

\*/

public static void main(String[] args){

/\*\*\*\*\*\* PARAMETER \*\*\*\*\*\*\*\*\*/

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

String queryWord1 = "watch"; // 修改为您要搜索的单词

String queryWord2 = "television"; // 修改为您要搜索的单词

LogicalCombination logicalCombination = AbstractIndexSearcher.LogicalCombination.ADJACENT; // 逻辑组合方式

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Sort simpleSorter = new SimpleSorter();

AbstractIndexSearcher searcher = new IndexSearcher();

searcher.open(indexFile);

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

AbstractHit[] hits = searcher.search(new Term(queryWord1), new Term(queryWord2),simpleSorter, logicalCombination);

RenderOutput.renderHits(hits);

}

}

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.AbstractIndexSearcher.LogicalCombination;

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：双关键字且

\*/

public class TestSearchIndexTwoWordAND {

/\*\*

\* 搜索程序入口

\* @param args ：命令行参数

\*/

public static void main(String[] args){

/\*\*\*\*\*\* PARAMETER \*\*\*\*\*\*\*\*\*/

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

String queryWord1 = "wales"; // 修改为您要搜索的单词

String queryWord2 = "luxembourg"; // 修改为您要搜索的单词

LogicalCombination logicalCombination = AbstractIndexSearcher.LogicalCombination.AND;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Sort simpleSorter = new SimpleSorter();

AbstractIndexSearcher searcher = new IndexSearcher();

searcher.open(indexFile);

System.out.println("-------Double word query AND-------");

AbstractHit[] hits = searcher.search(new Term(queryWord1), new Term(queryWord2), simpleSorter, logicalCombination);

RenderOutput.renderHits(hits);

}

}

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.AbstractIndexSearcher.LogicalCombination;

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：双关键字或

\*/

public class TestSearchIndexTwoWordOR {

/\*\*

\* 搜索程序入口

\* @param args ：命令行参数

\*/

public static void main(String[] args){

/\*\*\*\*\*\* PARAMETER \*\*\*\*\*\*\*\*\*/

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

String queryWord1 = "watching"; // 修改为您要搜索的单词

String queryWord2 = "television"; // 修改为您要搜索的单词

LogicalCombination logicalCombination = AbstractIndexSearcher.LogicalCombination.OR; //

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Sort simpleSorter = new SimpleSorter();

AbstractIndexSearcher searcher = new IndexSearcher();

searcher.open(indexFile);

System.out.println("-------Double word query OR-------");

AbstractHit[] hits = searcher.search(new Term(queryWord1), new Term(queryWord2),simpleSorter, logicalCombination);

RenderOutput.renderHits(hits);

}

}

/\*\*

\* 最后的程序运行入口类放在hust.cs.javacourse.search.run里

\*/

package hust.cs.javacourse.search.run;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;

}

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);

}

}

}

package hust.cs.javacourse.search.util;

/\*\*

\* 停用词表类

\*/

public class StopWords {

/\*\*

\* 停用词字符串数组

\*/

public static String[] STOP\_WORDS =

{"a", "about", "above", "above", "across", "after", "afterwards",

"again", "against", "all", "almost", "alone", "along", "already", "also","although",

"always","am","among", "amongst", "amoungst", "amount", "an", "and", "another", "any",

"anyhow","anyone","anything","anyway", "anywhere", "are", "around", "as", "at", "back",

"be","became", "because","become","becomes", "becoming", "been", "before", "beforehand",

"behind", "being", "below", "beside", "besides", "between", "beyond", "bill", "both", "bottom",

"but", "by", "call", "can", "cannot", "cant", "co", "con", "could", "couldnt", "cry", "de",

"describe", "detail", "do", "done", "down", "due", "during", "each", "eg", "eight", "either",

"eleven","else", "elsewhere", "empty", "enough", "etc", "even", "ever", "every", "everyone",

"everything", "everywhere", "except", "few", "fifteen", "fify", "fill", "find", "fire", "first",

"five", "for", "former", "formerly", "forty", "found", "four", "from", "front", "full", "further",

"get", "give", "go", "had", "has", "hasnt", "have", "he", "hence", "her", "here", "hereafter",

"hereby", "herein", "hereupon", "hers", "herself", "him", "himself", "his", "how", "however",

"hundred", "i","ie", "if", "in", "inc", "indeed", "interest", "into", "is", "it", "its", "itself",

"keep", "last", "latter", "latterly", "least", "less", "ltd", "made", "many", "may", "me",

"meanwhile", "might", "mill", "mine", "more", "moreover", "most", "mostly", "move", "much", "must",

"my", "myself", "name", "namely", "neither", "never", "nevertheless", "next", "nine", "no", "nobody",

"none", "noone", "nor", "not", "nothing", "now", "nowhere", "of", "off", "often", "on", "once", "one",

"only", "onto", "or", "other", "others", "otherwise", "our", "ours", "ourselves", "out", "over", "own",

"part", "per", "perhaps", "please", "put", "rather", "re", "same", "see", "seem", "seemed", "seeming",

"seems", "serious", "several", "she", "should", "show", "side", "since", "sincere", "six", "sixty",

"so", "some", "somehow", "someone", "something", "sometime", "sometimes", "somewhere", "still", "such",

"system", "take", "ten", "than", "that", "the", "their", "them", "themselves", "then", "thence", "there",

"thereafter", "thereby", "therefore", "therein", "thereupon", "these", "they", "thickv", "thin", "third",

"this", "those", "though", "three", "through", "throughout", "thru", "thus", "to", "together", "too", "top",

"toward", "towards", "twelve", "twenty", "two", "un", "under", "until", "up", "upon", "us", "very",

"via", "was", "we", "well", "were", "what", "whatever", "when", "whence", "whenever", "where", "whereafter",

"whereas", "whereby", "wherein", "whereupon", "wherever", "whether", "which", "while", "whither", "who",

"whoever", "whole", "whom", "whose", "why", "will", "with", "within", "without", "would", "yet", "you",

"your", "yours", "yourself", "yourselves","s","t"};

}

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";

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) + " ");

}

}

}