

**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 ： CS230x**

**学 号 ： U202311111**

**姓 名 ： losyi**

**指导教师 ： 辜希武**

**2025 年 5 月 1 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**

本项目要求实现一个基于内存的英文全文检索搜索引擎，通过分析题目要求，我们可以进一步细化和补充相关需求，为系统开发提供更明确的指导。

**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

内存索引构建是系统的核心功能。系统需要扫描指定目录下的所有txt文件，这一过程需要考虑文件访问权限不足、文件格式错误、目录下没有有效文件等异常情况的处理。在读取文件内容时，需要处理文件编码问题、超大文件可能导致的内存溢出问题，以及文件损坏情况下的容错机制。文本处理环节需要将内容分割成单词，这里需要明确分词规则（如何处理空格，空行，敏感词汇等），同时需要确保位置信息的准确记录，为后续的短语搜索提供支持。词语过滤部分需要实现数字、停用词和长度过滤，这要求系统维护一个完整的停用词列表，并能够灵活配置过滤规则，可以写在配置类中作为常量。

在倒排索引构建环节，系统应当高效地利用Java集合类实现Term到文档的映射关系。索引结构需要记录每个词在各文档中的出现频率和位置信息，同时考虑内存占用与检索效率的平衡。索引的序列化和反序列化功能要考虑序列化过程中可能出现的IO异常处理。控制台输出索引内容时，需要设计合理的格式化展示方式，实现各个类的toString方法时要考虑复用性。

**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

关于搜索功能，单关键词搜索需要对输入进行与索引构建相同的预处理（如大小写转换、过滤等），以确保查询词与索引词匹配。文档评分算法需要考虑词频，实现合理的相关性排序,本项目只实现了一个简单的打分和排序方法。结果展示时需要提供足够的信息，包括文档路径、内容预览、关键词出现位置和频率，以及相关性分数。

**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。

对于二关键词搜索，需要实现AND和OR操作的逻辑，这涉及集合运算的高效实现。尤其是AND操作，需要考虑如何快速找到同时包含两个词的文档

**功能4（未实现）：**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。

短语搜索功能作为进阶需求，本项目没有进行实现，不过我们也可以进行一定分析，为了让系统能够识别两个词相邻出现的情况。这需要在索引中精确记录每个词的位置，并在搜索时比较位置信息。需要处理的边界情况包括：短语跨段落或句子出现、文档中存在同一短语的多次出现、短语中包含停用词等。短语搜索结果的排序可能需要考虑短语频率、短语在文档中的位置等因素。

**其他需求：**

实现过程中必须严格遵循预定义的抽象类和接口，如已给出的AbstractDocument、AbstractTerm等，这些框架定义了系统的基本结构和行为。需要特别注意，不能修改这些预定义类的数据成员和方法签名，只能通过继承和实现来扩展功能。在异常处理方面，系统应当考虑到各种可能的错误情况，如IOExcption等，这些异常在Javadoc中已经给出。所有主要类和方法都需要添加规范的Javadoc注释，通过IDEA可以生产doc文档，便于代码维护和理解。

性能和资源管理是搜索引擎设计的关键。对于大规模文档集合，系统需要在内存占用和检索速度之间取得平衡。可能需要实现一些优化策略，如延迟加载、部分索引构建等。此外，系统的可扩展性也需要考虑，如何支持更多的查询过滤类型、更复杂的排序算法等将是系统后续升级的方向。当然，由于本项目作为实验课程而非完善的软件，测试文档也比较简单，并未在这些方面进行优化扩展，仅是实现了题目要求基础功能。

最后，测试和文档化是确保系统质量的重要环节，相关测试脚本在实验包中已经给出，不需要我们自己编写，只需要通过控制台输出记录测试结果。系统的设计文档和使用说明应当清晰描述系统架构、主要功能和使用方法，为用户提供必要的指导。

1. **系统设计**
2. **概要设计**

本搜索引擎系统采用层次化模块结构，通过抽象接口与实现类分离的设计思路，构建了灵活且可扩展的架构。系统主要分为索引层(index)、解析层(parse)、查询层(query)、运行层(run)和工具层(util)五个核心模块，各模块之间通过清晰的接口进行通信，形成了完整的全文检索功能体系。

索引模块是系统的基础，负责构建和管理倒排索引结构。该模块通过AbstractIndex、AbstractTerm、AbstractDocument等抽象类定义索引的基本行为和数据结构，而具体实现类如PostingList则实现了这些抽象类定义的功能。PostingList实现了对文档到词项的映射关系管理，包括添加、删除、排序等操作，并提供了序列化和反序列化功能，使索引可以持久化到文件并在需要时重新加载。索引结构使用了Java集合框架中的哈希表和列表的组合，确保了查询效率与内存使用的平衡。

解析模块处理文档内容的读取与处理，将原始文本转换为可索引的数据结构。该模块包含文档读取，分词，过滤等功能。负责从文件系统读取文档内容，进行分词、过滤和规范化处理。分词策略考虑了英文文本的特点，能够识别单词边界，并过滤掉数字、停用词和不符合长度要求的单词。模块使用了装饰者模式，方便添加多种过滤规则，实现了可扩展性。

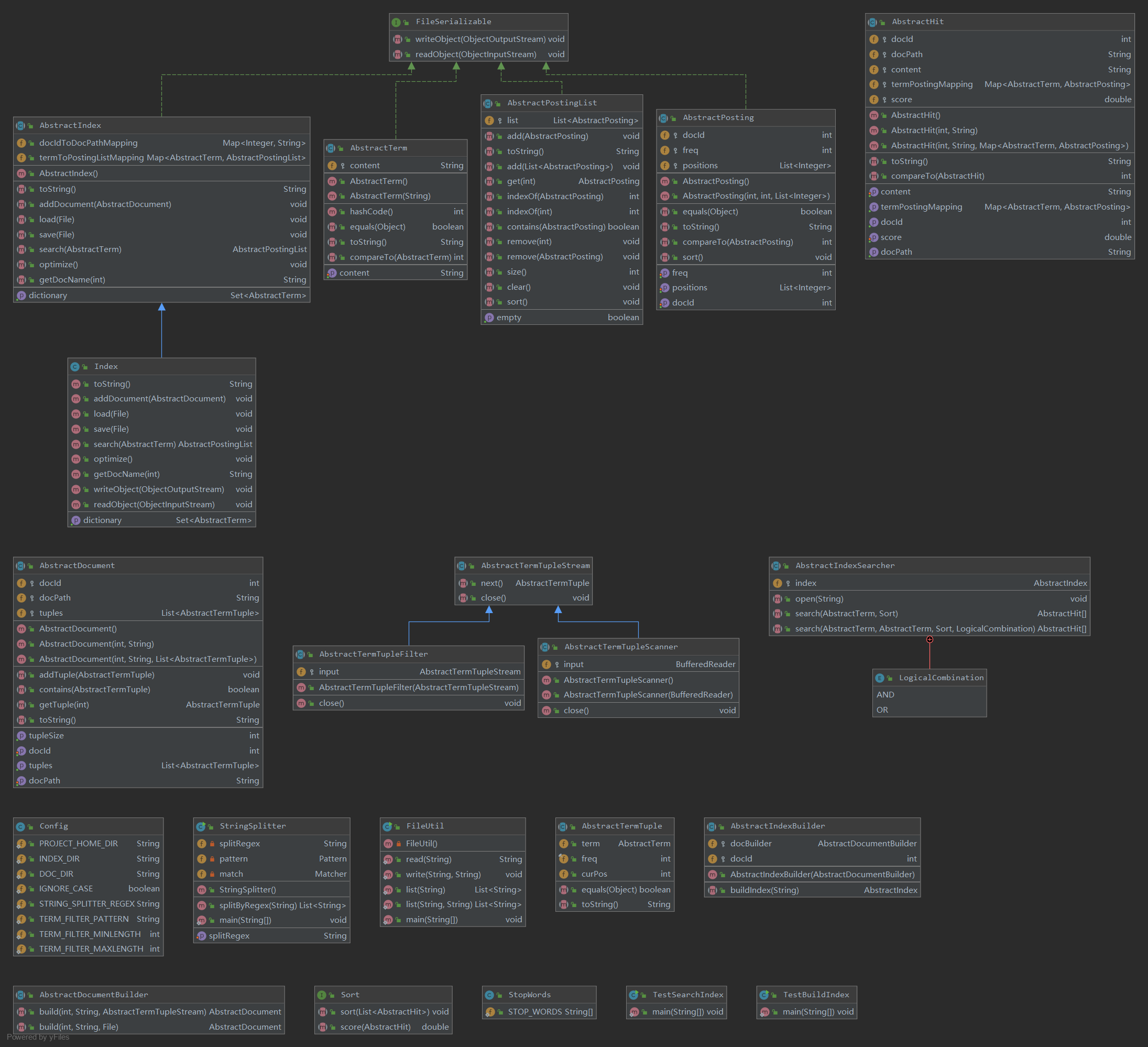
查询模块实现了搜索功能，是系统的核心交互部分。其中，IndexSearcher类实现了AbstractIndexSearcher抽象类，提供了单关键词和多关键词的搜索功能。单关键词搜索通过索引查找包含特定词项的文档列表，并封装为Hit对象返回。多关键词搜索则支持逻辑组合(AND/OR)，根据组合方式对两个检索词的结果进行集合运算。系统还实现了排序功能，通过Sort接口和SimpleSorter实现类，对检索结果根据相关性得分进行排序。SimpleSorter采用了简单的基于词频的评分策略，计算文档中关键词出现的总频率作为文档得分，并按得分降序排列结果。查询模块的设计体现了策略模式的应用，使得排序策略可以根据需求灵活更换。

运行模块是系统的入口，提供用户交互界面和程序流程控制。该模块整合了索引构建和检索功能，通过控制台接收用户输入的查询请求，调用查询模块执行搜索，并格式化展示结果。运行模块应该实现人机交互界面，为用户提供直观的操作方式，包括索引构建、单词查询、多词查询和短语搜索等功能选项。该模块还需要负责处理用户输入的异常情况，确保系统稳定运行。当然，限于课程设计要求和复杂度，本实验并没有实现交互界面，需要手动修改代码以进行测试。

工具模块提供了系统各部分共用的辅助功能，包括文件操作、字符串处理和配置管理等。这些工具类简化了其他模块的实现，提高了代码的复用性和可维护性。例如，文件工具类封装了文件读写操作，处理了可能的IO异常，字符串工具类提供了文本处理和格式转换功能，配置管理类则集中管理系统参数，便于全局调整。

各模块之间的调用关系形成了清晰的层次结构：运行模块位于顶层，负责协调整个系统的流程，查询模块依赖于索引模块，通过索引结构检索文档，索引模块依赖于解析模块，从解析的文档内容构建索引，工具模块则为所有其他模块提供支持。这种层次化的设计使得各模块职责明确，便于开发和维护。模块间通过预定义的接口进行通信，如查询模块通过AbstractIndex接口访问索引功能，解析模块通过AbstractDocument接口提供文档内容，这种接口化的设计增强了系统的可扩展性和模块的独立性。

总体而言，该搜索引擎系统通过模块化的设计和抽象接口的应用，实现了高内聚低耦合的架构，各功能模块各司其职，通过清晰的接口协同工作，共同构成了一个完整的全文检索系统。系统设计充分考虑了扩展性和可维护性，为未来功能的扩展和性能的优化提供了良好的基础。



## 详细设计

设计每个模块的实现算法（处理流程）、所需的局部数据结构。具体介绍每个模块/子程序的功能、入口参数、出口参数、流程（图）等。

### 模块1(index)：

package hust.cs.javacourse.search.index包里首先定义了与倒排索引结构相关的抽象类。

AbstractTerm：其具体子类实例为一个单词term。

AbstractPosting：其具体子类实例为倒排索引里的一个Posting，其中包含三个数据成员docId、freq、positions，分别代表单词出现的文档Id、出现频率、出现的位置列表，其中位置列表采用Java的集合类型List<Integer>存贮单词出现的多个位置

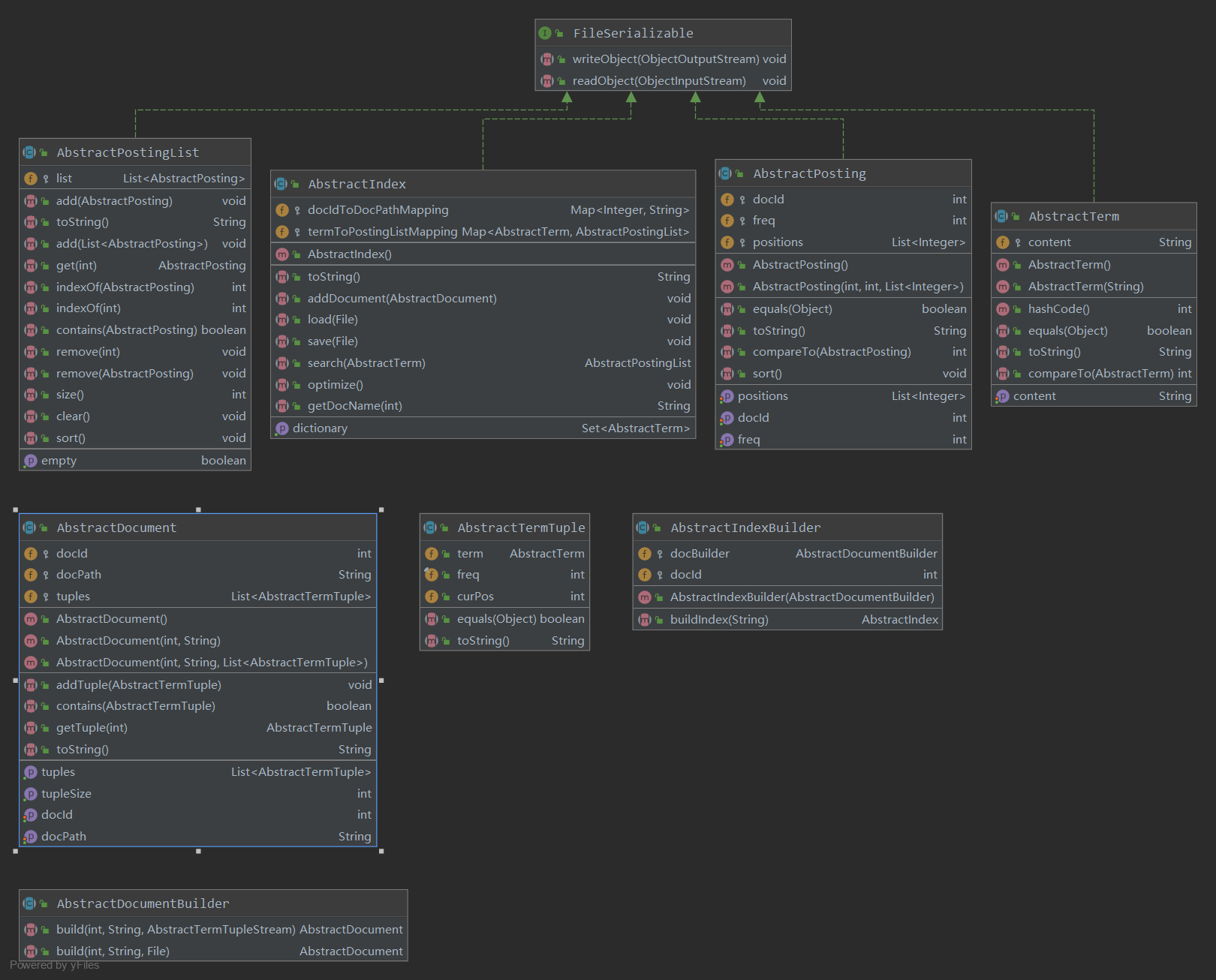
AbstractPostingList：其具体子类实例为倒排索引里一个单词对应的PostingList，其中包含一个List<AbstractPosting>类型的数据成员存放这个PostingList包含的多个Posting

AbstractIndex：其具体子类实例为内存中的整个倒排索引结构，其中包含了二个数据成员

docIdToDocPathMapping：类型为Map<Integer, String> ，保存了文档Id和文档绝对路径之间的映射关系（开始构建索引时我们只有每个文档的绝对路径，因此内部需要维护一个文档Id的计数器，每将一个文档加入到倒排索引，文档Id计数器加1）

termToPostingListMapping：类型为Map<AbstractTerm, AbstractPostingList> ，保存了每个单词与其对应的PostingList的映射关系。需要特别说明的是这里没有必要用专门的数据结构来存放字典内容，我们直接通过termToPostingListMapping.keySet( )方法就可以得到字典。

包里定义的抽象类和接口的UML模型见下图：



1. **Term类**

1）功能

Term 类是搜索引擎中对关键词的基本封装，用于表示文档中出现的基本单位（如词项、关键词等）。它实现了 AbstractTerm 抽象类，具备序列化、比较、相等判断等功能。

2）主要属性与局部数据结构

|  |  |  |
| --- | --- | --- |
| **变量名** | **类型** | **说明** |
| content | String | 词项内容（关键词） |

3）方法说明

|  |  |  |  |
| --- | --- | --- | --- |
| **方法签名** | **入口参数** | **出口参数** | **功能简述** |
| Term() | 无 | 构造空Term对象 | 构造一个默认的空词项实例 |
| Term(String content) | content: 词项内容 | 构造一个含内容的Term对象 | 根据给定字符串构造词项 |
| equals(Object obj) | obj: 要比较的对象 | boolean | 判断两个Term是否等价（内容相同） |
| toString() | 无 | String | 返回词项内容的字符串表示形式 |
| getContent() | 无 | String | 获取词项的内容 |
| setContent(String content) | content: 要设置的内容 | 无 | 设置词项内容 |
| compareTo(AbstractTerm o) | o: 待比较的Term对象 | int | 实现字典序比较，返回排序差值 |
| writeObject(ObjectOutputStream out) | out: 输出流对象 | 无 | 将词项对象写入二进制文件中（用于序列化） |
| readObject(ObjectInputStream in) | in: 输入流对象 | 无 | 从二进制文件读取词项数据（用于反序列化） |

1. **TermTuple类**

1）功能说明

TermTuple 类是词三元组，包含该词项对象（Term）、其当前位置（curPos）以及词频（freq）。这是构建倒排索引的基础数据结构之一，用于表示某个关键词在某个文档中的详细信息。

2）主要属性与局部数据结构

|  |  |  |
| --- | --- | --- |
| **变量名** | **类型** | **说明** |
| term | AbstractTerm | 当前词项对象 |
| curPos | int | 词项在文档中的当前位置 |
| freq | int | 当前词项的词频，默认为 1 |

3）方法说明

|  |  |  |  |
| --- | --- | --- | --- |
| **方法签名** | **入口参数** | **出口参数** | **功能简述** |
| TermTuple() | 无 | 构造空TermTuple | 默认构造函数，创建一个空三元组 |
| TermTuple(AbstractTerm term, int curPos) | term: 词项对象 curPos: 词项位置 | 构造对象 | 根据词项及位置创建三元组对象 |
| equals(Object obj) | obj: 待比较对象 | boolean | 比较两个三元组是否等价（term与curPos均相等） |
| toString() | 无 | String | 返回该三元组的字符串表示形式 |

1. **Posting类**

1）功能说明

Posting 类是倒排索引的基本单位，表示某个词项在某个文档中出现的信息。其核心包括：

词项所属文档的编号（docId），出现频率（freq），出现的位置列表（positions）。它支持排序、比较、序列化、反序列化等功能。

2）主要属性与局部数据结构

|  |  |  |
| --- | --- | --- |
| **属性名** | **类型** | **说明** |
| docId | int | 当前词项所在文档的 ID |
| freq | int | 词项在该文档中出现的次数 |
| positions | List<Integer> | 出现的具体位置（按词项顺序记录） |

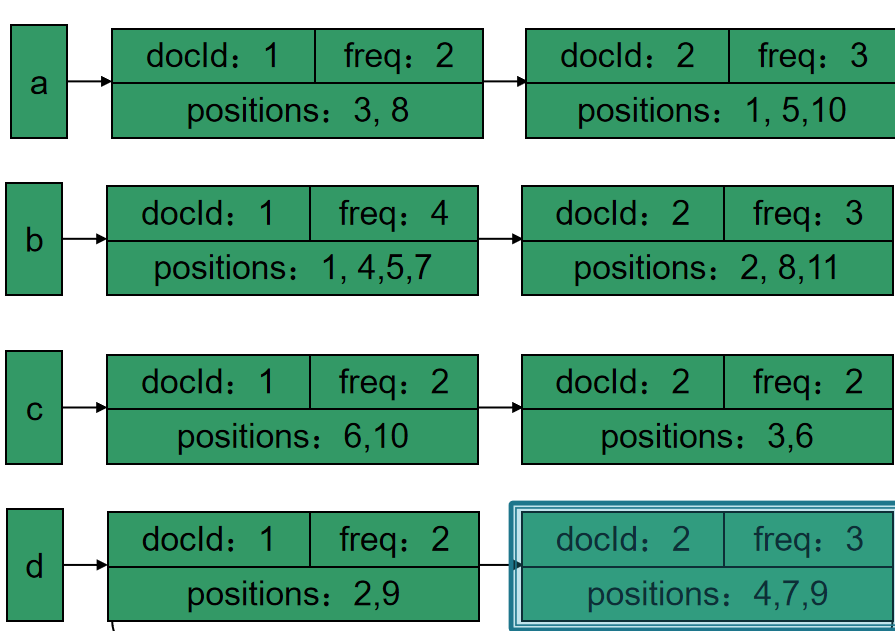
3）方法说明

|  |  |  |  |
| --- | --- | --- | --- |
| **方法签名** | **入口参数** | **出口参数** | **功能简述** |
| Posting() | 无 | 构造空对象 | 默认构造函数，初始化为空 |
| Posting(int docId, int freq, List<Integer> positions) | 传入文档id、词频和位置列表 | 构造对象 | 创建完整的Posting对象 |
| equals(Object obj) | 比较对象 | boolean | 判断两个Posting是否内容完全相等 |
| compareTo(AbstractPosting o) | 比较对象 | int | 比较两个Posting的docId，用于排序 |
| toString() | 无 | String | 返回JSON风格的字符串形式 |
| writeObject(ObjectOutputStream out) | 输出流 | 无 | 序列化Posting对象 |
| readObject(ObjectInputStream in) | 输入流 | 无 | 从二进制流中读取对象 |
| sort() | 无 | 无 | 对positions列表排序 |

1. **PostingList类**

1）功能说明

PostingList 是表示某个词项在所有文档中出现的集合。每个词项对应一个 PostingList，其中包含若干 Posting 对象，记录了这个词项在各文档中的出现情况。该类支持增删、查找、排序、序列化等操作。通过PostingList，我们便可以建立文档的倒排索引，如下图所示：



倒排索引的组成：

Dictionary：文档集所有term的集合。字典中的term按字典序排序，可以加快查找速度

Posting List：由Posting组成的链表

Posting：每个term所出现的文档Id，及其在文档中出现的位置、词频

2）主要属性与局部数据结构

|  |  |  |
| --- | --- | --- |
| **属性名** | **类型** | **说明** |
| list | List<AbstractPosting> | 存储所有Posting对象的列表 |

3）方法说明

|  |  |  |  |
| --- | --- | --- | --- |
| **方法签名** | **参数** | **返回值** | **说明** |
| add(AbstractPosting posting) | posting：待添加项 | void | 向列表添加一个Posting，不允许重复 |
| add(List<AbstractPosting> postings) | 多个Posting | void | 批量添加，内部调用add(posting) |
| get(int index) | 指定下标 | 对应Posting | 获取某个Posting |
| indexOf(AbstractPosting posting) | Posting对象 | int | 返回指定Posting的下标 |
| indexOf(int docId) | 文档ID | int | 查找具有指定docId的Posting下标 |
| contains(AbstractPosting posting) | Posting对象 | boolean | 判断是否存在 |
| remove(int index) | 下标 | void | 按索引移除Posting |
| remove(AbstractPosting posting) | Posting对象 | void | 删除对应的Posting |
| size() | 无 | 列表长度 | 返回Posting的数量 |
| isEmpty() | 无 | boolean | 是否为空 |
| clear() | 无 | void | 清空列表 |
| sort() | 无 | void | 按 docId 升序排序 |
| writeObject(ObjectOutputStream out) | 输出流 | void | 写入文件 |
| readObject(ObjectInputStream in) | 输入流 | void | 从文件中读取 |

1. **Document类**

1）类功能说明

Document 类是解析完一个文档后文档在内存中的表示。因为当解析完文档后，文档需要一种中间类型的数据结构表示，以方便后面倒排索引的建立。它包括三个数据成员：

int docId：文档Id

String docPath：文档绝对路径

List<AbstractTermTuple> tuples：文档解析完后得到的所有term的三元组

可以看到，当解析完文档，得到AbstractDocument子类对象，我们有了方便后面构建倒排索引的一种中间数据结构。得到AbstractDocument子类对象后调用AbstractIndex子类对象的addDocument方法就可以将该文档加入倒排索引。另外由于他们是中间的数据结构，因此不需要序列化。

2）主要属性与局部数据结构

|  |  |  |
| --- | --- | --- |
| **属性名** | **类型** | **说明** |
| docId | int | 文档的唯一编号 |
| docPath | String | 文档的绝对路径 |
| tuples | List<AbstractTermTuple> | 存储该文档的所有词项三元组 |

3）方法说明

|  |  |  |  |
| --- | --- | --- | --- |
| **方法签名** | **参数** | **返回值** | **功能描述** |
| Document() | 无 | 构造函数 | 空文档构造器 |
| Document(int docId, String docPath) | 文档ID和路径 | 构造函数 | 构造一个不带三元组的新文档 |
| Document(int docId, String docPath, List<AbstractTermTuple> tuples) | 包含三元组列表 | 构造函数 | 初始化完整文档信息 |
| int getDocId() | 无 | int | 获取文档ID |
| void setDocId(int docId) | 文档ID | 无 | 设置文档ID |
| String getDocPath() | 无 | String | 获取文档路径 |
| void setDocPath(String docPath) | 路径字符串 | 无 | 设置文档路径 |
| List<AbstractTermTuple> getTuples() | 无 | List | 获取三元组列表 |
| void addTuple(AbstractTermTuple tuple) | 一个三元组 | 无 | 添加一个三元组（不重复） |
| boolean contains(AbstractTermTuple tuple) | 一个三元组 | boolean | 判断三元组是否存在 |
| AbstractTermTuple getTuple(int index) | 下标 | 三元组 | 获取指定下标处的三元组 |
| int getTupleSize() | 无 | int | 获取三元组数量 |
| String toString() | 无 | String | 返回完整字符串表示（包括文档ID、路径、三元组信息） |

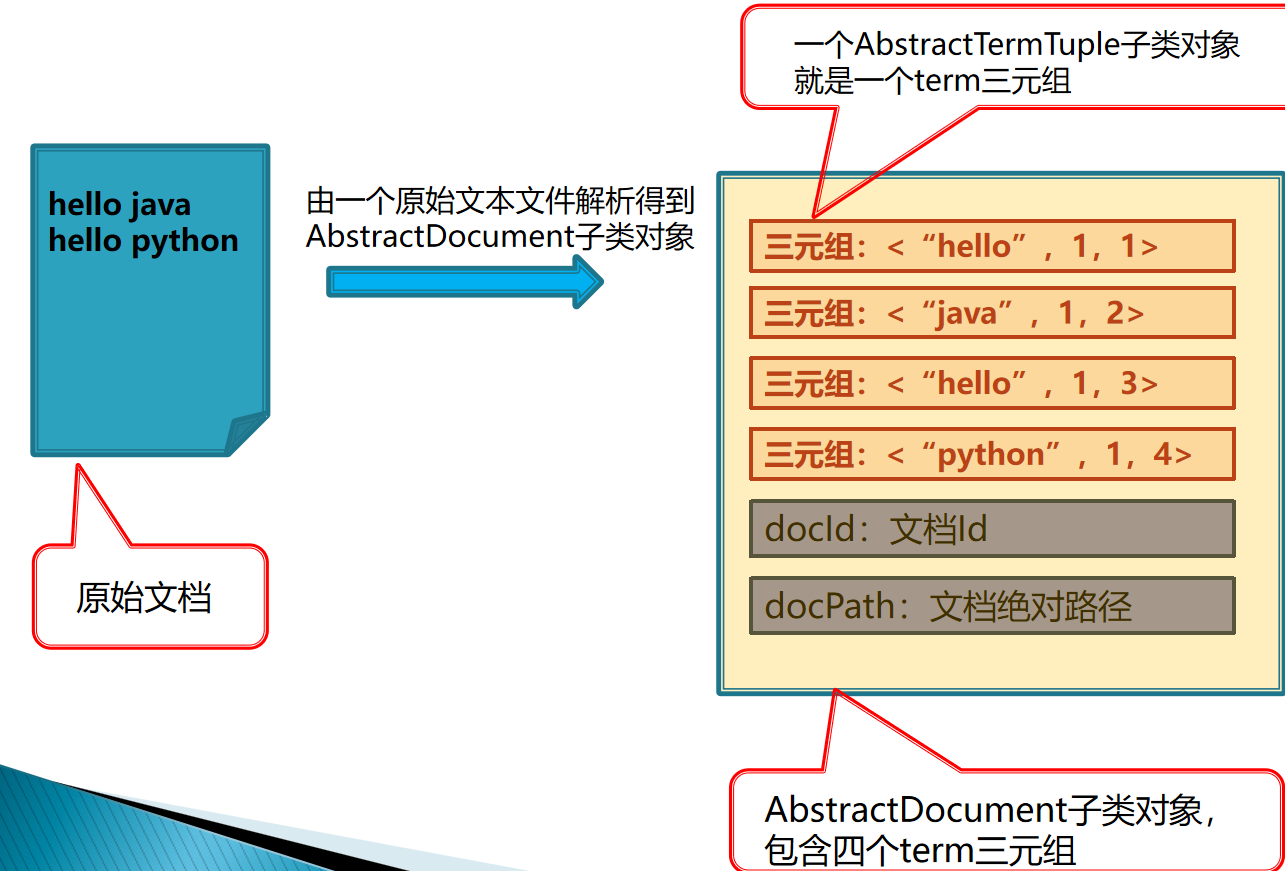
4）处理流程简述

Document类提供三种构造方式以供选择：空构造、路径+ID构造以及路径+ID+三元组构造。通过提供的getter和setter方法，用户可以方便地访问和修改文档编号与路径，同时，系统也支持对三元组的访问和查询操作。在添加三元组时，可以使用addTuple()方法，该方法在内部会调用contains()方法以确保添加的三元组不会重复，contains()方法则是通过List的contains()方法来进行查找的，这一查找过程依赖于三元组自身的equals()方法来判断两个三元组是否相等，此外，系统还支持通过索引访问三元组以及统计三元组的数量，为用户提供了更多的操作便利。在输出方面，系统提供了格式化输出功能，包括文档ID、路径以及所有三元组的内容，这样的输出方式便于用户进行调试和展示。

1. **DocumentBuilder类**

1）类功能说明

DocumentBuilder 是 AbstractDocumentBuilder 的具体实现类，负责将文本文件解析成由多个三元组（TermTuple）组成的 Document 对象（如下图所示）它利用 TermTupleStream（在后续模块中介绍） 处理文本，并在必要时使用多层过滤器（如停用词过滤器、正则过滤器、长度过滤器）来清洗和提取有效词项。



2）主要属性与局部数据结构

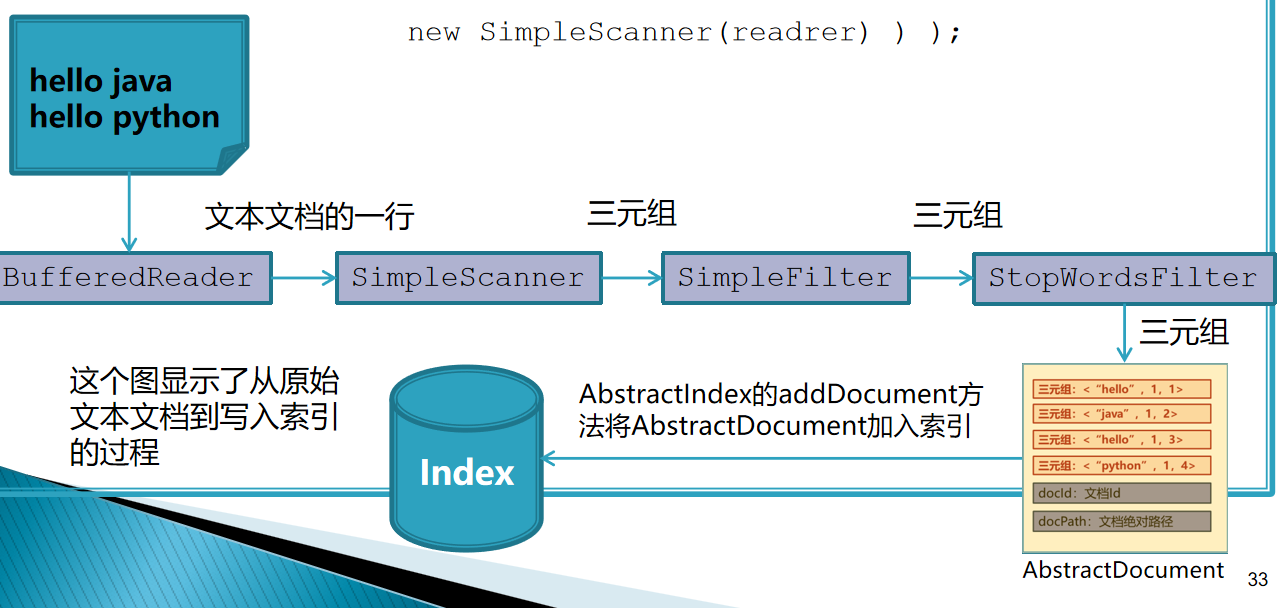
本类没有成员变量，仅通过方法构建 Document 对象。

3）方法说明

|  |  |  |  |
| --- | --- | --- | --- |
| **方法签名** | **参数** | **返回值** | **功能描述** |
| DocumentBuilder() | 无 | 构造器 | 默认构造函数 |
| AbstractDocument build(int docId, String docPath, AbstractTermTupleStream termTupleStream) | 文档ID、路径、三元组流 | 文档对象 | 从一个三元组流构建文档对象 |
| AbstractDocument build(int docId, String docPath, File file) | 文档ID、路径、源文件 | 文档对象 | 利用输入文件构造 TermTupleStream 并调用上一个方法构造文档对象 |

4）处理流程简述

1. build(docId, docPath, AbstractTermTupleStream)：创建一个新的 Document 实例，不断调用 termTupleStream.next() 获取下一个 TermTuple，将其添加到文档中，最后返回构建好的 Document。
2. build(docId, docPath, File file)：利用 TermTupleScanner 对文本文件进行分词，通过三层过滤器包裹（StopWord → Pattern → Length）过滤无效词项，最终构造 TermTupleStream，调用上一个 build 方法生成最终文档。这里利用了装饰者模式，通过多个过滤器将对象层层包装，处理流程如下图所示：



1. **Index类**

1）类功能说明

Index 是搜索引擎索引模块的核心实现类，继承自抽象类 AbstractIndex，用于建立和管理倒排索引结构。它支持将文档对象转化为索引、将索引持久化到磁盘、从磁盘读取索引、搜索指定词项的文档列表，并对索引数据进行排序优化。

2）主要属性与局部数据结构

Map<Integer, String> docIdToDocPathMapping

存储文档编号与文件路径之间的映射。

Map<AbstractTerm, AbstractPostingList> termToPostingListMapping

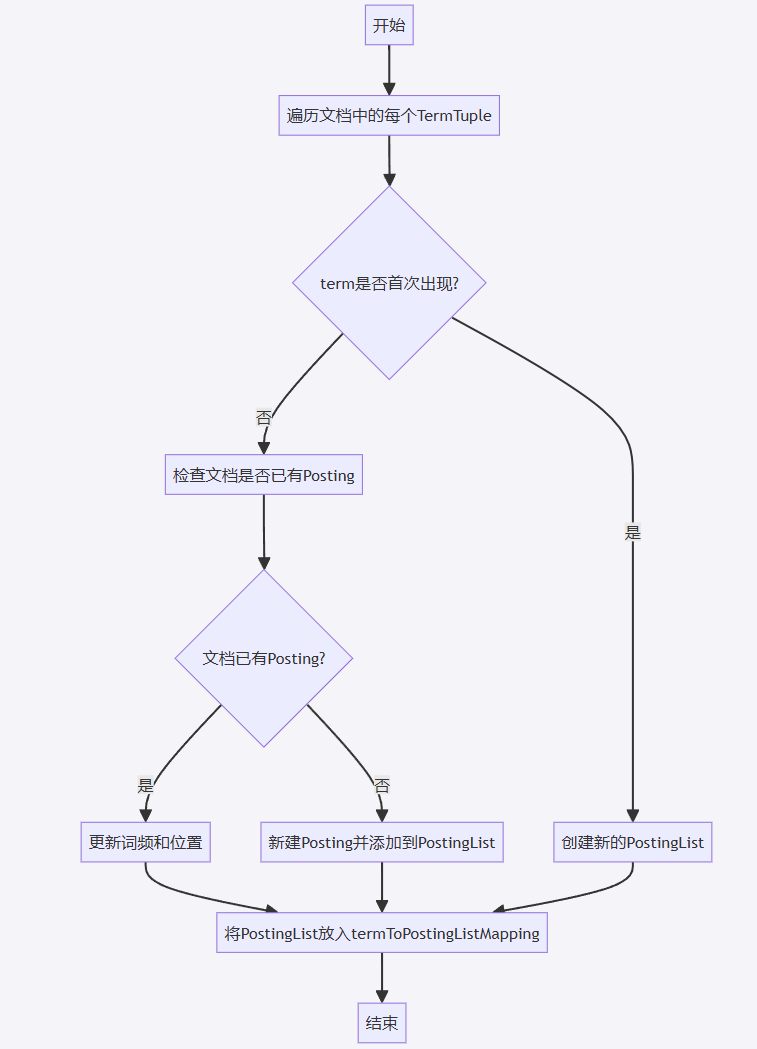
存储每个词项对应的倒排记录列表（PostingList）。

3）方法说明

|  |  |
| --- | --- |
| **方法签名** | **功能说明** |
| String toString() | 返回索引的字符串形式，便于调试和打印。 |
| void addDocument(AbstractDocument document) | 将文档中的每个三元组加入倒排索引。 |
| void save(File file) | 将索引序列化保存到指定文件。 |
| void load(File file) | 从文件中加载索引结构。 |
| AbstractPostingList search(AbstractTerm term) | 根据词项返回其对应的 PostingList。 |
| Set<AbstractTerm> getDictionary() | 返回索引中的所有词项集合（即词典）。 |
| void optimize() | 对每个 PostingList 按 docId 排序，并对每个 posting 的词位位置排序。 |
| String getDocName(int docId) | 根据文档编号返回对应路径。 |
| void writeObject(ObjectOutputStream out) | 将当前索引对象写入输出流（序列化）。 |
| void readObject(ObjectInputStream in) | 从输入流中读取索引对象（反序列化）。 |

4）处理流程简述（addDocument 方法）

遍历文档中的每个 TermTuple（词项 + 位置），若该词项 term 首次出现，创建新的 PostingList，对于该文档是否已有 Posting：若有，则更新词频、位置，若无，则新建一个 Posting，添加到 PostingList，将更新后的 PostingList 放入 termToPostingListMapping 中。



1. **IndexBuilder类**

1）类功能说明

IndexBuilder 类是抽象类 AbstractIndexBuilder 的具体子类，用于构建指定目录下所有 .txt 文本文件的倒排索引。该类的主要功能是利用 DocumentBuilder 将文件解析为文档对象，再通过 Index 类的方法将其添加进索引中，最终形成一个完整的倒排索引结构，供后续检索模块使用。

2）主要属性与局部数据结构

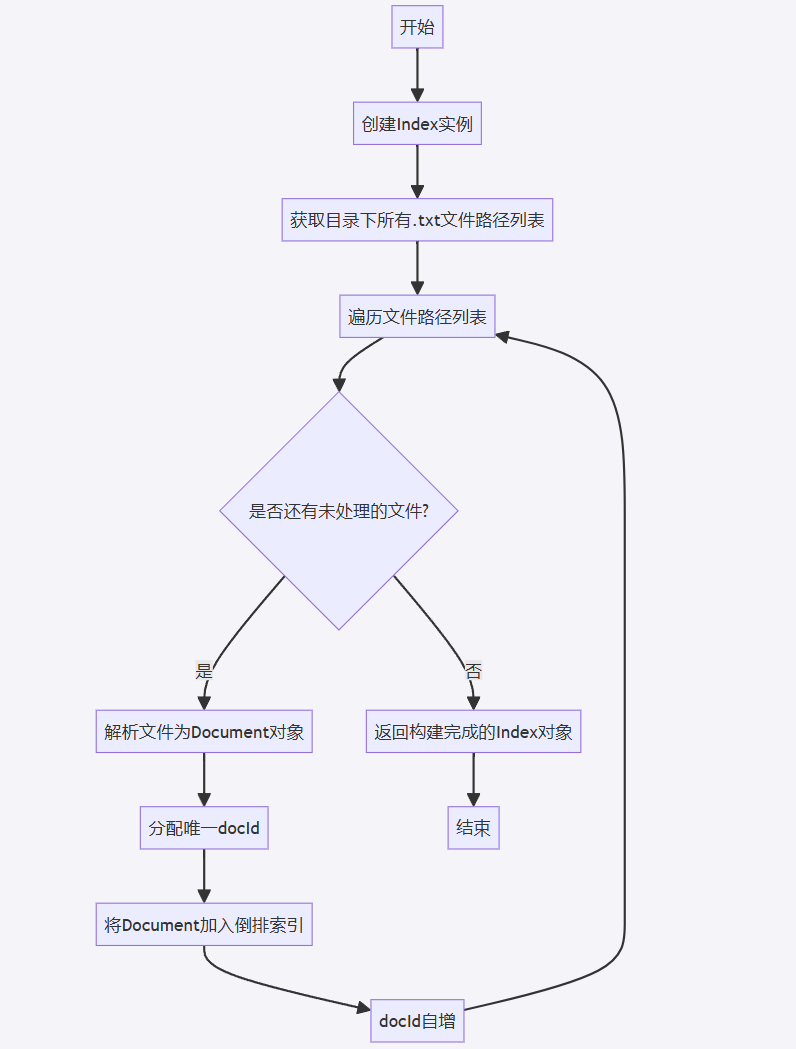
|  |  |  |
| --- | --- | --- |
| **名称** | **类型** | **说明** |
| docBuilder | AbstractDocumentBuilder | 来自父类的成员，用于构建文档对象 |
| docId | int | 当前处理的文档编号，从0开始递增 |
| docPaths | List<String>（局部变量） | 指定目录下所有.txt文本文件的完整路径列表 |
| index | AbstractIndex（局部变量） | 实际创建的倒排索引结构，返回给调用者 |

3）方法说明

|  |  |
| --- | --- |
| **方法名** | **说明** |
| IndexBuilder(AbstractDocumentBuilder docBuilder) | 构造方法，传入一个文档构建器，用于后续生成 Document 对象。 |
| AbstractIndex buildIndex(String rootDirectory) | 构建 rootDirectory 目录下所有 .txt 文件的倒排索引，返回 Index 对象。 |

4）处理流程简述

当调用 buildIndex(String rootDirectory) 方法时，执行过程如下：创建一个新的 Index 实例，用于存储构建的倒排索引。使用 FileUtil.list(rootDirectory, ".txt") 方法获取该目录下所有 .txt 文件的完整路径列表。遍历上述文件路径列表，对于每一个 .txt 文件：使用传入的 docBuilder 将该文件解析为一个 Document 对象，分配唯一的 docId，调用 index.addDocument(document) 将其加入倒排索引结构，docId 自增，为下一个文档准备。返回最终构建完成的 Index 对象，供后续搜索模块使用。



### **模块2(parse)：**

package hust.cs.javacourse.search.parser：包里定义了三个抽象类：

AbstractTermTupleStream

AbstractTermTupleScanner

AbstractTermTupleFilter

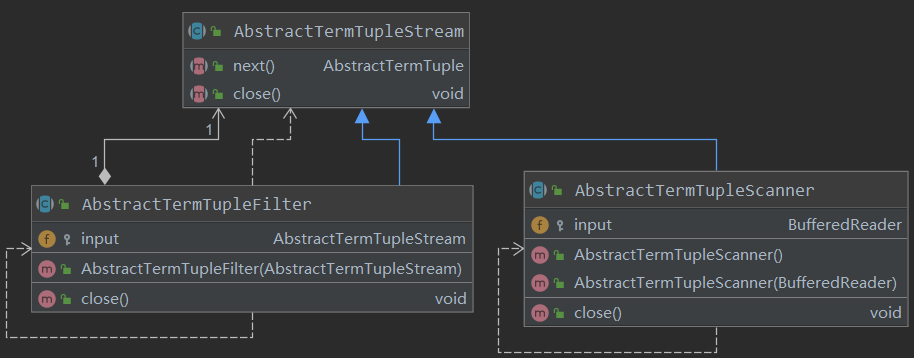
AbstractTermTupleStream是另外二个抽象类的父类，我们只需要实现 AbstractTermTupleScanner和AbstractTermTupleStream的具体子类。必须要强调的是它们具体子类对象类型都是AbstractTermTupleStream。

AbstractTermTupleStream类是对解析文档过程中产生的一个个单词的三元组(AbstractTermTuple子类对象）的一个抽象，即把一个文档看成三元组流，其中规定了二个基于流的抽象方法：

public abstract AbstractTermTuple next()：从流中获得下一个三元组

public abstract void close()：关闭流

包里定义的抽象类和接口的UML模型见下图：



1. **TermTupleScanner类**

1）类功能说明

TermTupleScanner 是 AbstractTermTupleScanner 的具体子类，其功能是将输入流中的文本数据逐行读取、按指定正则表达式切分为单词，并封装为三元组（TermTuple）对象。该类是整个解析模块的基础组件，用于实现文本到三元组流（TermTuple流）的转换。

2）主要属性与局部数据结构

|  |  |  |
| --- | --- | --- |
| **名称** | **类型** | **说明** |
| buffer | Deque<String> | 缓冲区，保存当前行中尚未处理的单词 |
| curPos | int | 当前处理单词在整个文本流中的位置索引（从0开始递增） |
| input | BufferedReader | 父类字段，读取文本内容的输入流 |

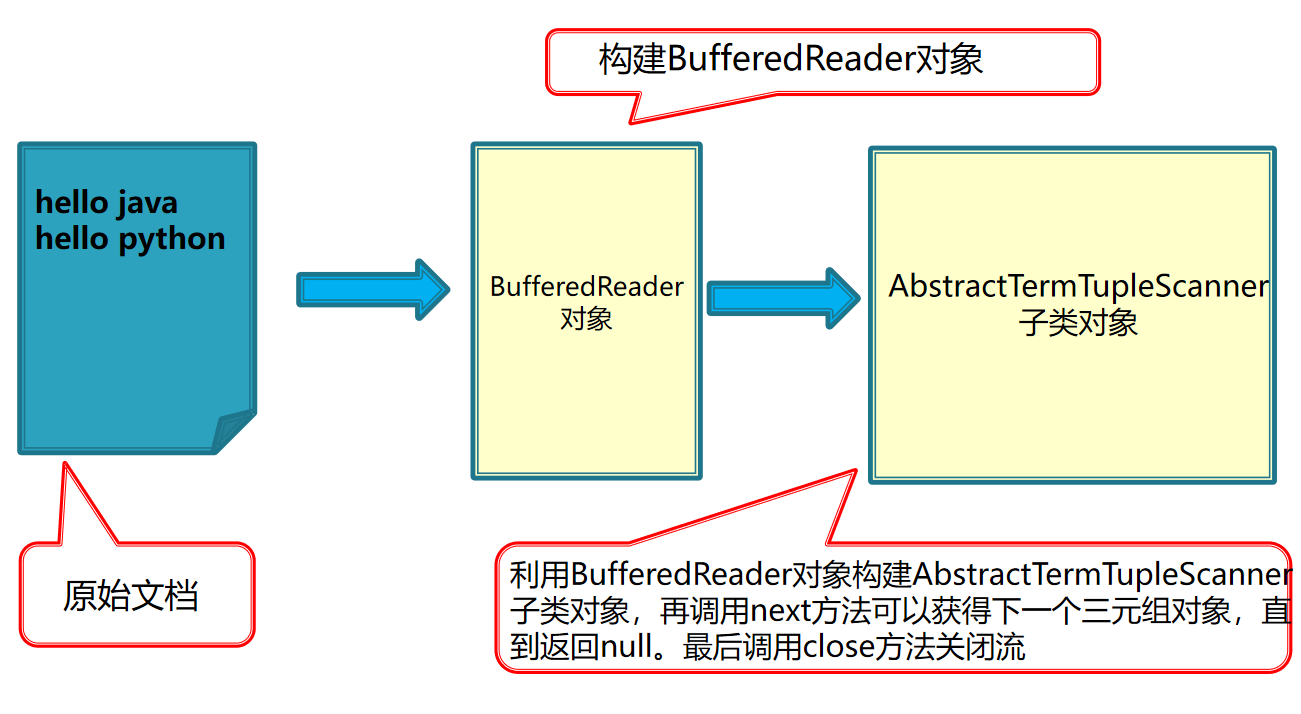
3）方法说明

|  |  |
| --- | --- |
| **方法名** | **说明** |
| TermTupleScanner() | 无参构造函数，供框架调用时使用 |
| TermTupleScanner(BufferedReader input) | 关联指定输入流的构造方法，用于处理具体文本内容 |
| AbstractTermTuple next() | 获取下一个三元组对象，若文件已结束或读取异常，则返回 null |

4）处理流程简述

当用户调用 next() 方法以获取下一个三元组时，执行流程如下：

若当前缓冲区 buffer 为空，说明需要从输入流中读取新的一行文本,使用 BufferedReader.readLine() 读取一行，如果是空行或为空，跳过该行，使用 Config.STRING\_SPLITTER\_REGEX 分割该行得到词项列表，若配置中启用了忽略大小写选项（Config.IGNORE\_CASE），则将所有单词转换为小写,将分割后的词项加入 buffer 中备用。从 buffer 中取出第一个词项，使用该词项构造一个 TermTuple 对象，词项位置由 curPos 指定,将 curPos 自增 1，为下一次构造准备,返回构造好的 TermTuple 对象,若输入流读完，则返回 null。



1. **LengthTermTupleFilter类**

1）类功能说明

LengthTermTupleFilter 是 AbstractTermTupleFilter 的子类，用于过滤掉词项长度不在指定范围内的三元组。其工作原理是检查每个 TermTuple 中的词项（term）的长度，若长度不符合配置文件中指定的最小与最大值要求，则丢弃该三元组，继续处理下一个。

2）主要属性与局部数据结构

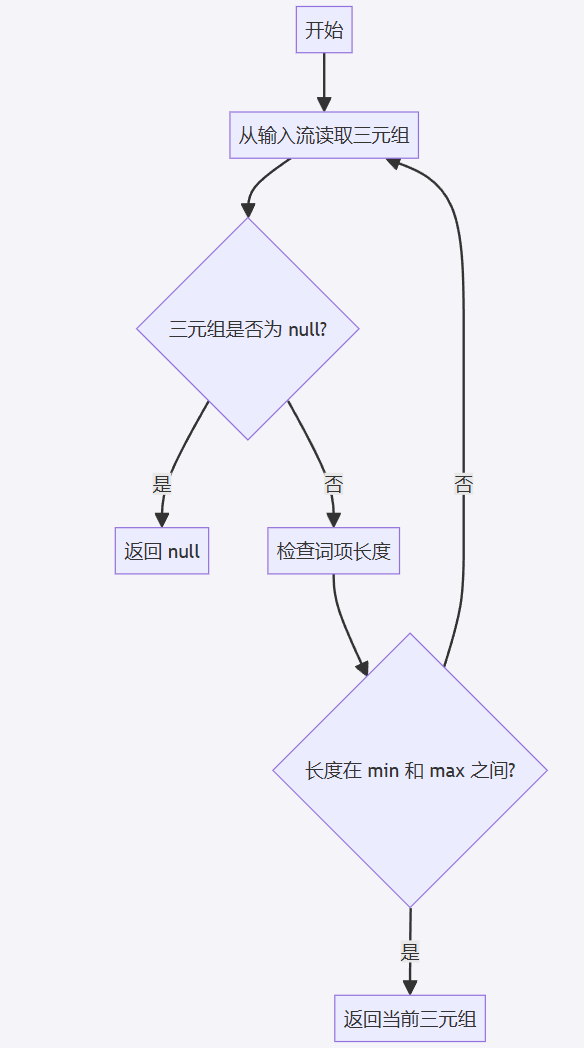
|  |  |  |
| --- | --- | --- |
| **名称** | **类型** | **说明** |
| input | AbstractTermTupleStream | 父类字段，表示过滤器接收的三元组流（通常为TermTupleScanner或其他Filter） |
| tuple | AbstractTermTuple | 当前从输入流中读取的三元组，用于判断其词项长度是否合法 |

3）方法说明

|  |  |
| --- | --- |
| **方法名** | **说明** |
| LengthTermTupleFilter(input) | 构造函数，设置输入流 |
| AbstractTermTuple next() | 获取下一个长度合法的三元组，不合法则跳过，直到找到或读完 |

4）处理流程简述

从输入流 input 中获取一个三元组 tuple，如果该三元组的 term.content.length() 落在 [TERM\_FILTER\_MINLENGTH, TERM\_FILTER\_MAXLENGTH] 范围内，返回该三元组，否则继续循环读取下一个三元组若输入流读完（返回 null），则 next() 返回 null。通过这一过滤器，可以有效去除过短或过长的噪音词，提升后续索引质量和搜索精度。



1. **StopWordTermTupleFilter类**

1）类功能说明

StopWordTermTupleFilter 是 AbstractTermTupleFilter 的具体子类，用于过滤掉所有出现在停用词表 STOP\_WORDS 中的词项，通过屏蔽掉常见但无意义的词（如 “the”, “and”, “is” 等），该类有助于提升索引结构的质量与搜索结果的相关性。

2）主要属性与局部数据结构

|  |  |  |
| --- | --- | --- |
| **名称** | **类型** | **说明** |
| input | AbstractTermTupleStream | 父类字段，表示输入的三元组流 |
| tuple | AbstractTermTuple | 当前读取的三元组 |
| STOP\_WORDS | String[] | 停用词数组，定义于 hust.cs.javacourse.search.util.StopWords 类中 |

3）方法说明

|  |  |
| --- | --- |
| **方法名** | **说明** |
| StopWordTermTupleFilter(input) | 构造函数，设置输入流 |
| AbstractTermTuple next() | 获取下一个非停用词三元组，跳过所有属于停用词表中的三元组 |

4）处理流程简述

从输入流 input 中不断获取下一个三元组 tuple，判断 tuple.term.getContent() 是否在 STOP\_WORDS 停用词表中，若不在停用词表中，则返回该三元组，若在停用词表中，则跳过继续读取，若整个输入流读取完毕仍未找到合法词项，则返回 null，该过滤器可与其他过滤器（如 LengthTermTupleFilter、PatternTermTupleFilter）串联使用，实现复合过滤逻辑，流程与前者类似，这里不再给出流程图。

1. **PatternTermTupleFilterl类**

1）类功能说明

PatternTermTupleFilter 是 AbstractTermTupleFilter 的具体子类，它基于配置文件中提供的正则表达式规则 Config.TERM\_FILTER\_PATTERN，对三元组中的词项进行模式匹配过滤。仅当词项内容符合正则表达式（如仅由英文字母组成）时，才保留该三元组。

2）主要属性与局部数据结构

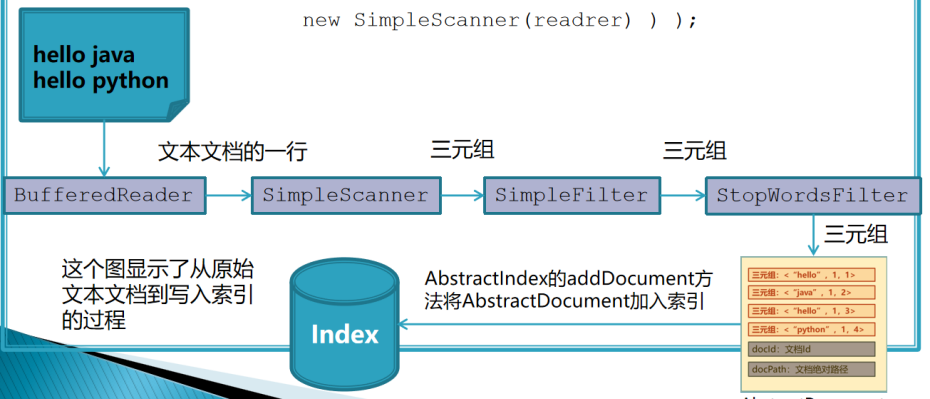
|  |  |  |
| --- | --- | --- |
| **名称** | **类型** | **说明** |
| input | AbstractTermTupleStream | 父类字段，表示过滤器的输入流 |
| tuple | AbstractTermTuple | 当前读取的三元组 |
| TERM\_FILTER\_PATTERN | String | 正则表达式配置，如 "[a-zA-Z]+" 表示只保留纯英文词项 |

3）方法说明

|  |  |
| --- | --- |
| **方法名** | **说明** |
| PatternTermTupleFilter(input) | 构造函数，设置输入流 |
| AbstractTermTuple next() | 获取下一个匹配正则表达式的三元组，跳过不符合格式的词项 |

4）处理流程简述

从输入流 input 中逐个读取三元组 tuple，使用正则表达式 Config.TERM\_FILTER\_PATTERN 对三元组中的词项 tuple.term.getContent() 进行匹配，若匹配成功，则返回该三元组；若不匹配（例如包含数字、标点或其他非法字符），则跳过，若全部输入已处理完且无符合条件的词项，返回 null。此类通常用于预过滤非法词项，以提高索引质量，可配合停用词和长度过滤器共同组成过滤流水线。如下图所示：



### **模块3(query)：**

1. **Hit类**
2. 类功能说明

AbstractHit是搜索命中结果的抽象类。命中的文档必须要有特定的数据结构来支持命中结果的显示和排序，因此不能简单地用原始文本文档的内容作为命中结果。

Hit 类是 AbstractHit 的实现类，表示一次搜索命中的结果项（即与查询词匹配的一个文档），它封装了文档编号、路径、内容、得分信息以及匹配到的 Term 和对应的 Posting 映射关系，该类实现了 Comparable 接口，支持按得分排序。

2）主要属性说明（继承自 AbstractHit）

|  |  |  |
| --- | --- | --- |
| **成员名** | **类型** | **说明** |
| docId | int | 文档编号，唯一标识某个文档 |
| docPath | String | 文档的绝对路径 |
| content | String | 文档内容（可选，用于展示） |
| termPostingMapping | Map<AbstractTerm, AbstractPosting> | 命中的 Term 与对应 Posting 的映射 |
| score | double | 当前命中的文档得分，用于排序 |

3）方法说明

|  |  |
| --- | --- |
| **方法名** | **说明** |
| Hit() | 默认构造方法 |
| Hit(int, String) | 创建包含文档编号和路径的 Hit 对象 |
| Hit(int, String, Map) | 创建包含文档编号、路径和匹配词项映射的 Hit 对象 |
| getDocId() / getDocPath() | 获取文档编号 / 路径 |
| getContent() / setContent(String) | 获取 / 设置文档内容 |
| getScore() / setScore(double) | 获取 / 设置文档得分 |
| getTermPostingMapping() | 获取当前命中的 Term → Posting 映射 |
| compareTo(AbstractHit) | 按照 score 字段进行比较，用于排序 |
| equals(Object) | 判断是否为同一个 Hit（包括得分和 Term 映射完全一致） |
| toString() | 返回完整的 Hit 信息字符串，适合调试或结果展示 |

4）处理流程简述

当查询系统从索引中找出满足条件的文档集合后，程序会将每个结果封装成一个 Hit 实例。

首先创建 Hit 对象并初始化 docId 与 docPath，设置命中词项映射关系 termPostingMapping，在排序前，根据词项频率或位置计算得分 score 并设置，所有 Hit 对象将进入排序器中，根据得分由高到低排序，展示或返回前，可以设置 content（文档预览内容）。

1. **SimpleSorter类**

1）类功能说明

package hust.cs.javacourse.search.query. Sor是对命中结果计算得分和排序的接口SimpleSorter 是 Sort 接口的一个实现类，负责对搜索命中结果 Hit 列表按文档得分进行排序。得分的计算方式是命中词项在文档中出现的总频率，然后按得分从高到低排序（即降序排序）。

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。又例如，如果把文档的得分值设置成等于文档id，就是实现了按文档id排序的简单策略。

文档的得分值计算出来后要设置到AbstractHit子类对象里。若集合里每个元素都实现了Comparable接口，那么只需要调用方法就可实现从小到大排序。这也是为什么很多抽象类都规定子类必须实现Comparable接口的原因。

2）成员方法说明

|  |  |  |
| --- | --- | --- |
| **方法名** | **返回值类型** | **说明** |
| sort(List<AbstractHit>) | void | 根据 score 对 Hit 列表降序排序。排序前先调用 score() 方法计算每个文档得分。 |
| score(AbstractHit) | double | 计算一个 Hit 对象的得分：所有命中词项在该文档中出现次数的总和，并设置 score。 |

3）得分计算逻辑

score(hit) 方法会遍历 hit.getTermPostingMapping() 中所有词项对应的 Posting，从每个 Posting中获取词项在该文档中的出现频次 ap.getFreq()，对所有频次求和作为该 Hit 的得分；

调用 hit.setScore(score) 更新该命中文档的得分。

4）排序逻辑

在 sort() 方法中，先对列表中每个 Hit 进行 score() 计算；然后使用 hits.sort(...) 按得分从高到低（降序）排序。

1. **indexSearcher类**

1）类功能说明

package hust.cs.javacourse.search.query. AbstractIndexSearcher是进行全文检索的抽象类，IndexSearcher 是 AbstractIndexSearcher 的具体实现类，负责从本地索引文件中加载倒排索引，并基于关键词执行搜索操作。支持单个关键词搜索和两个关键词的搜索，并支持 AND / OR 逻辑组合。使用 Sort 接口进行结果排序。

2）主要成员方法说明

|  |  |  |
| --- | --- | --- |
| **方法名** | **返回值类型** | **功能描述** |
| void open(String indexFile) | void | 加载本地序列化索引文件到 index 成员变量中 |
| AbstractHit[] search(AbstractTerm queryTerm, Sort sorter) | AbstractHit[] | 执行单关键词搜索，构造并返回命中结果列表 |
| AbstractHit[] search(AbstractTerm t1, AbstractTerm t2, Sort s, LogicalCombination c) | AbstractHit[] | 执行双关键词搜索，支持 AND / OR 查询 |
| private void processAndLogic(...) | void | 内部辅助方法，处理 AND 逻辑命中文档 |
| private void processOrLogic(...) | void | 内部辅助方法，处理 OR 逻辑命中文档 |
| private AbstractHit createHit(...) | AbstractHit | 构造单关键词命中结果 Hit 对象 |

3）搜索与排序流程说明

【单关键词搜索流程】

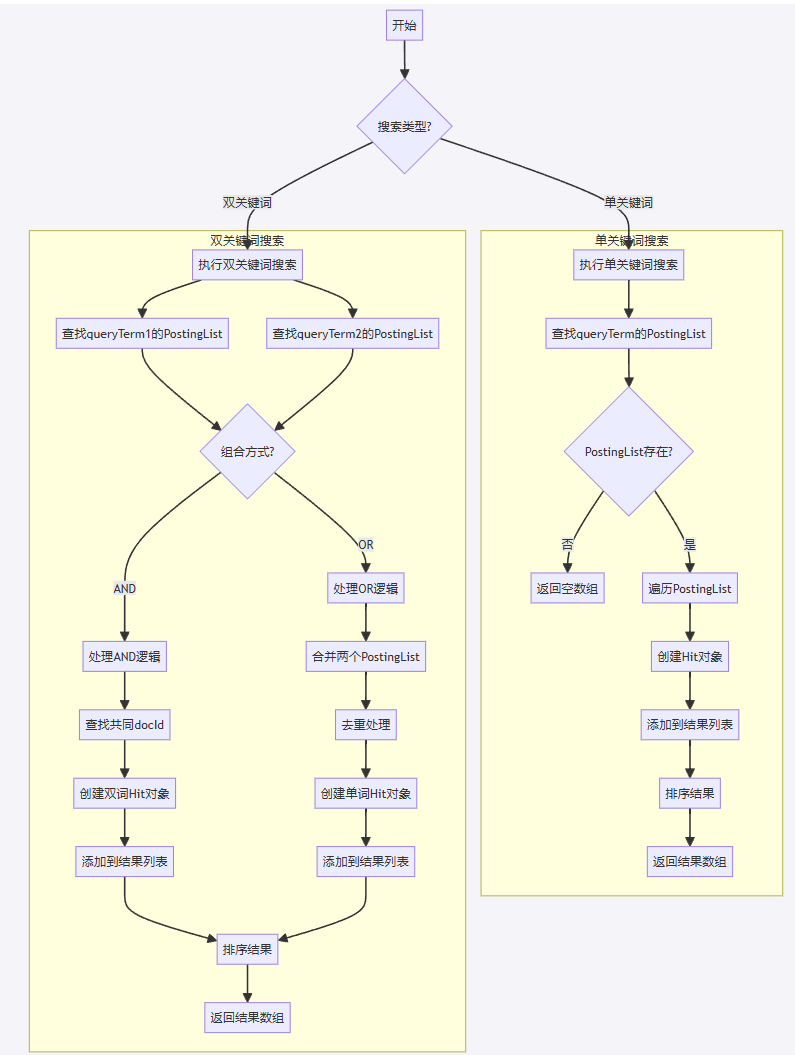
首先在索引中查找关键词对应的倒排列表，通过调用 index.search(queryTerm)，返回该关键词对应的 PostingList（一个包含多个文档位置和词频的列表），若返回为 null，说明该关键词未出现于任何文档中，直接返回空数组。接着遍历 PostingList，构建 Hit 对象，对每一个 Posting（表示关键词在某文档中的出现情况）执行，获取 docId。调用index.getDocName(docId) 获取文档路径。构造一个新的 Hit 对象，包含文档 ID、路径和 Map<AbstractTerm, AbstractPosting> 形式的关键词映射，将该 Hit 加入结果列表。然后使用排序器对命中结果排序，将命中结果传入 Sort.sort(List<AbstractHit> hits) 方法，调用 SimpleSorter.score(hit) 计算每个 Hit 的得分（默认为所有匹配词频之和），然后使用 Comparator 对命中结果按得分降序排序。最后将 List<AbstractHit> 转换为 AbstractHit[] 数组作为返回结果。

【双关键词搜索流程】

分别查找两个关键词的倒排列表，使用 index.search(queryTerm1) 和 index.search(queryTerm2) 分别获取两个关键词的 PostingList。根据逻辑组合方式选择搜索策略

若为 AND 组合：遍历 postingList1 中的每个 Posting，获取对应的 docId；调用 postingList2.indexOf(docId) 判断该 docId 是否同时出现在 postingList2 中；若存在匹配，说明两个词都出现在该文档中，将两个词对应的 Posting 加入 Map<AbstractTerm, AbstractPosting> 中；使用 docId 和文档路径创建一个包含两个关键词信息的 Hit，加入 hits 列表，并记录 docId 进入 processedDocIds 避免重复处理。

若为 OR 组合：在 postingList1 和 postingList2 中分别遍历每个 Posting对于未在 processedDocIds 中出现的 docId，构造仅包含该词的 Hit 并加入 hits。同时将该 docId 加入 processedDocIds 避免重复。排序命中结果。与单关键词相同，调用 sorter.sort(hits) 对结果按照得分进行降序排序，最后将排序后的命中结果转换为 AbstractHit[] 并返回。



1. **软件开发**

操作系统：Windows 11 23H2 (家庭版)

集成开发环境：IntelliJ IDEA 2024.3.4 x64（Ultimate Edition专业版）

JDK版本：JDK17.0.1

通过IDE原生工具进行构建项目，调试程序。

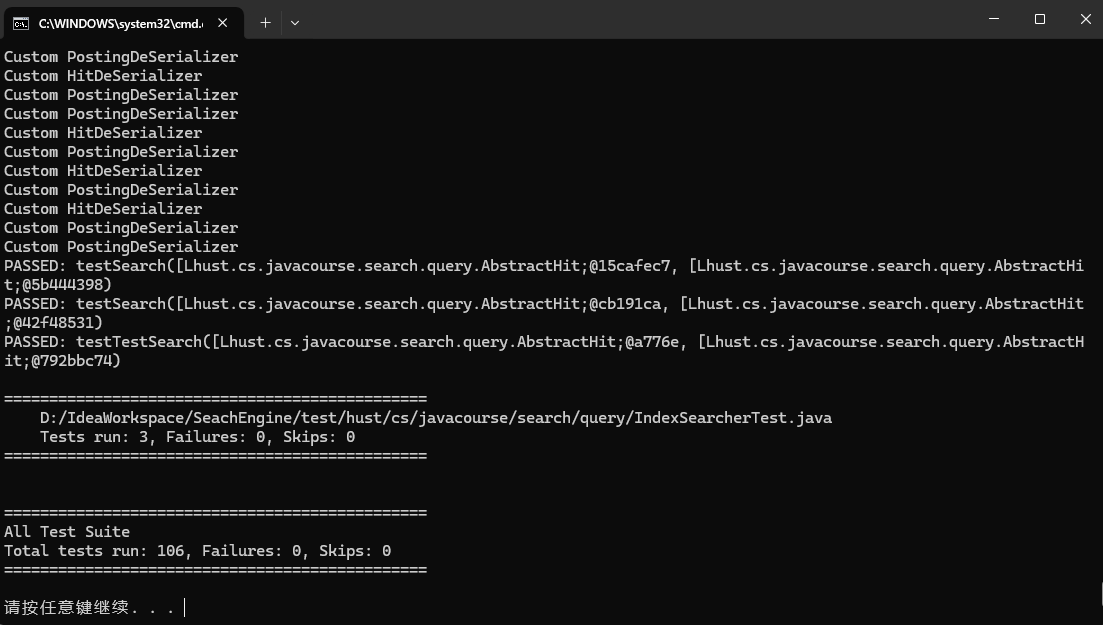
1. **软件测试**

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**



1. **功能测试**

**（1）索引构建**

采用实验提供的功能测试数据集，验证索引构建模块是否能够，正确读取文档目录，解析文档内容生成索引，将索引保存到文件并能正确加载，输出的索引结构内容与预期一致。测试代码如下：

    public static void main(String[] args){

        AbstractDocumentBuilder documentBuilder = new DocumentBuilder();

        AbstractIndexBuilder indexBuilder = new IndexBuilder(documentBuilder);

        String rootDir = Config.DOC\_DIR;

        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*

*//测试保存到文件*

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

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

*//测试从文件读取*

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

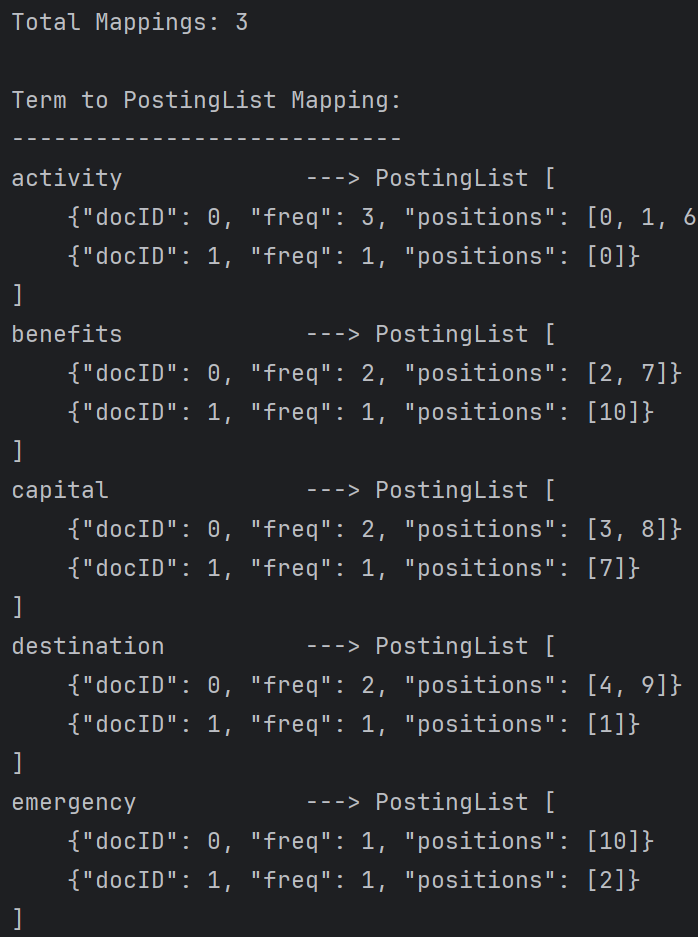
        index2.load(new File(indexFile)); *//从文件加载对象的内容*

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

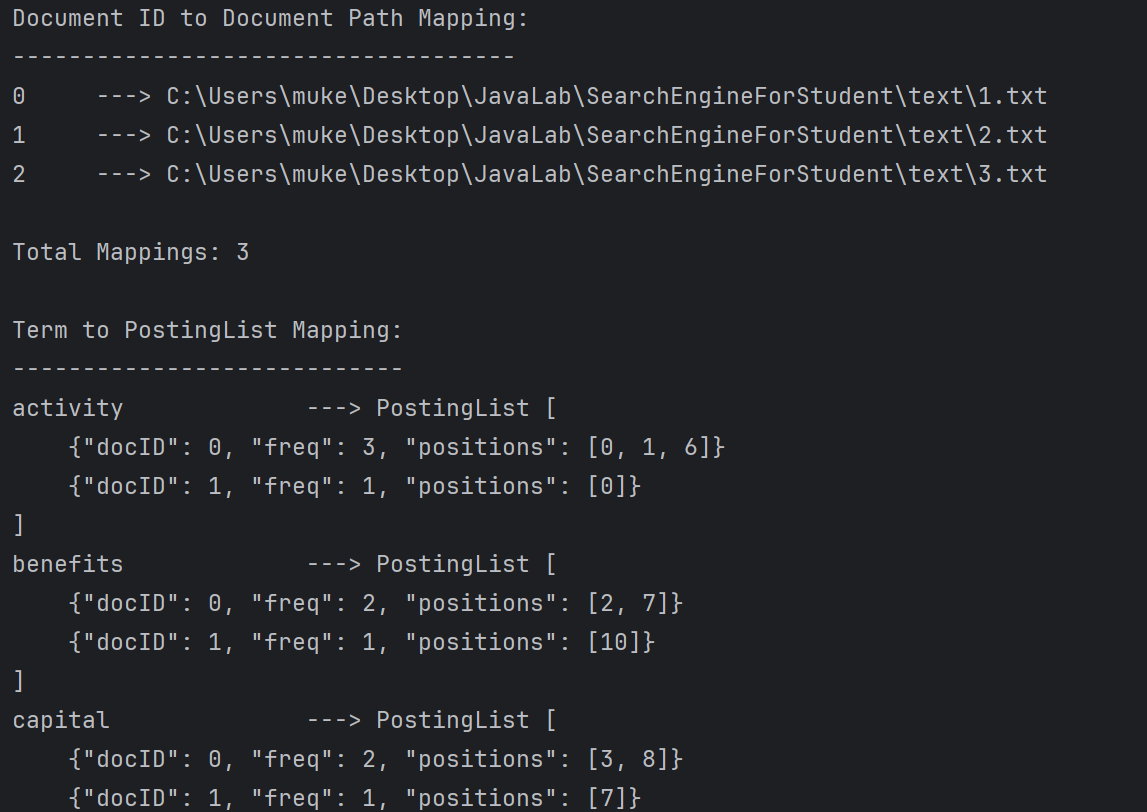
        System.out.println(index2); *//控制台打印 index2 的内容*

}

输出如下：



对照原txt文件，可见程序成功构建了索引并保存，读取索引文件后输出的结果与原来相同，说明索引构建功能和序列化/反序列化功能正常。



进一步观察原txt文件，可见2005 12-31 F16等数字和i等单字符没有参与索引构建，说明我们的过滤功能成功发挥了作用，因为在配置文件中，我们有如下过滤规则：

*/*

     \* <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;

**（2）单词查询**

已构建的索引文件 index.dat 来自以下 3 个文档，路径与内容如下所示：

1.txt：

activity activity benefits capital destination

fizzy activity benefits capital destination emergency

2005 12-31 F16 the from Pseudopseudohypoparathyroidism

2.txt：

activity destination emergency

26% U571 because hence capital i

frozen benefits google hahahahahahaha ha frozen google

3.txt：

marketplace medical 2005 notification except peninsula Pseudopseudohypoparathyroidism pollution

其中

activity 出现在 docID 0 和 1；

because 只出现在 docID 1；

frozen 和 medical 分别出现在 docID 1 和 2。

由此撰写测试代码：

public static void main(String[] args) {

        Sort simpleSorter = new SimpleSorter();

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

        AbstractIndexSearcher searcher = new IndexSearcher();

        searcher.open(indexFile);

        System.out.println("===== [1] 单词查询：Term(\"activity\") =====");

        AbstractHit[] hits1 = searcher.search(new Term("activity"), simpleSorter);

        for (AbstractHit hit : hits1) {

            System.out.println(hit);

        }

        System.out.println("\n===== [2] 多词查询（AND）：\"activity\" AND \"because\" =====");

        AbstractHit[] hits2 = searcher.search(new Term("activity"), new Term("because"), simpleSorter, AbstractIndexSearcher.LogicalCombination.AND);

        for (AbstractHit hit : hits2) {

            System.out.println(hit);

        }

        System.out.println("\n===== [3] 多词查询（OR）：\"frozen\" OR \"medical\" =====");

        AbstractHit[] hits3 = searcher.search(new Term("frozen"), new Term("medical"), simpleSorter, AbstractIndexSearcher.LogicalCombination.OR);

        for (AbstractHit hit : hits3) {

            System.out.println(hit);

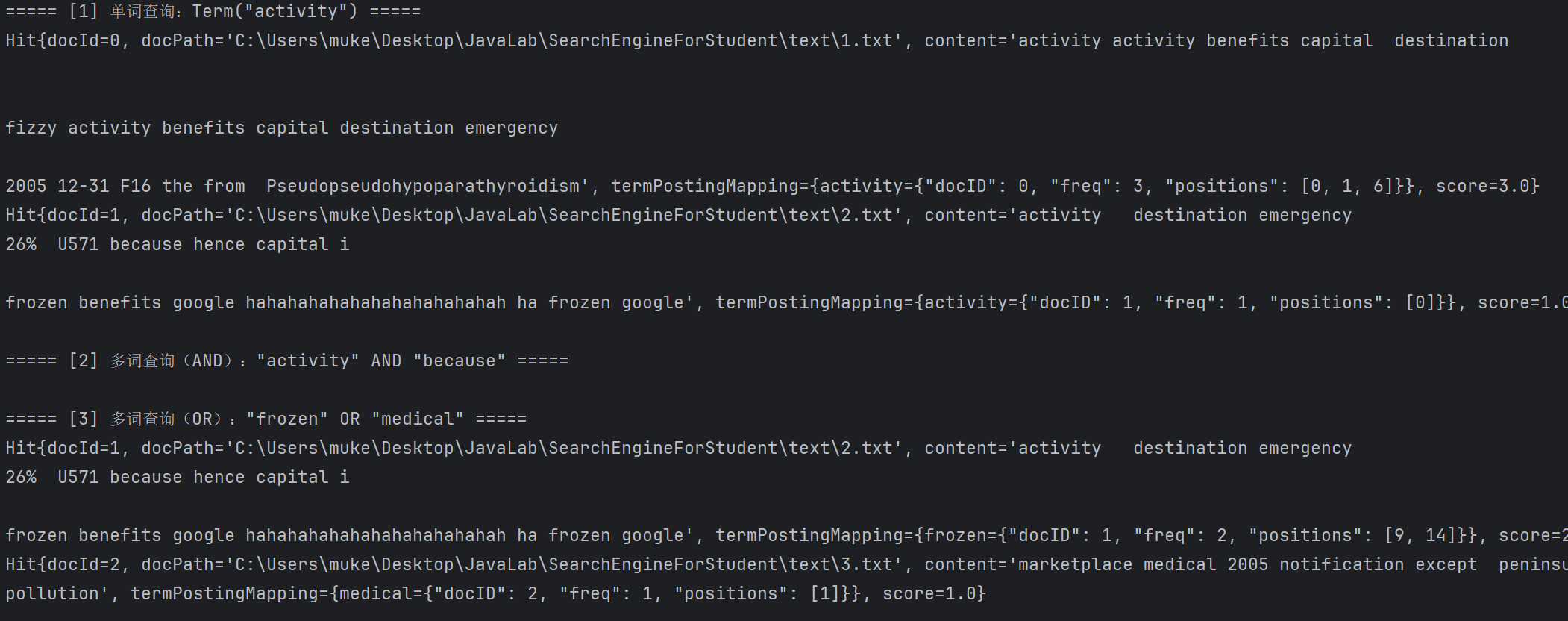
        }

    }

**测试用例和预期输出表**

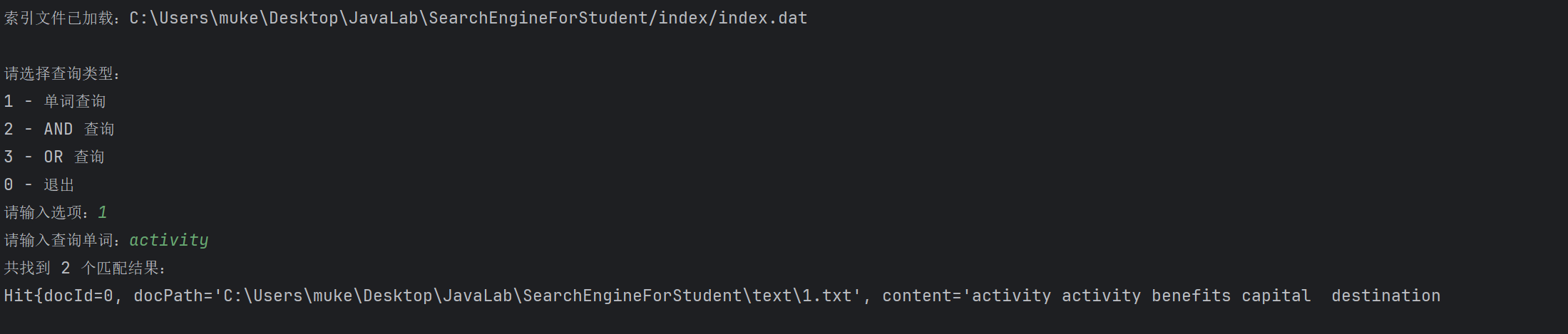
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **用例编号** | **检索条件** | **逻辑组合** | **预期文档ID** | **说明** |
| TC01 | "activity" | 单词 | 0, 1 | 两文档中都出现 |
| TC02 | "activity" AND "because" | AND | 1 | 只有 docID 1 同时包含 |
| TC03 | "frozen" OR "medical" | OR | 1, 2 | 分别出现在两文档中 |

实际输出为：



**（3）交互测试查询**

TestSearchIndexInteractive类供用户使用，可自行选择查询类型和查询单词（需要先构建索引）,程序会自行读取索引文件进行查询。



1. **特点与不足**
2. **技术特点**

本项目构建了一个结构清晰、功能完善的基于内存的搜索引擎系统，体现出良好的软件工程设计思想与信息检索技术基础。在实现过程中，巧妙融合了高效的数据结构、面向对象编程范式及设计模式，为系统打下了坚实的技术基础，具有如下突出特点：

**（1）高效的倒排索引结构**

项目核心采用倒排索引（Inverted Index）作为存储和查询的数据结构，实现了从词项（Term）到文档（Document）的一对多映射。这种结构相比前向索引更具检索效率优势，尤其在大规模文档场景下更能体现其性能优势。每个词项关联的 Postings List 中不仅包含出现的文档编号，还记录了词频和词位，为后续查询优化提供了丰富的结构信息。由于索引中的词项是有序的，因此支持基于二分查找的快速定位，理论上查询复杂度可降低为 O(log N)，具备处理上亿级别文档的潜力。

**（2）分层抽象的类与接口设计**

系统采用分层式架构设计，各模块职责分明。索引结构、文档解析、查询处理三大功能通过多个抽象类进行模块划分，如 AbstractIndex、AbstractTermTupleScanner、AbstractIndexSearcher 等，每个类或接口只承担单一责任，体现出良好的高内聚、低耦合设计理念。这种架构模式大大提高了系统的可维护性与可扩展性。只需继承抽象类并实现其具体子类，即可快速定制搜索引擎的行为。

**（3）多种设计模式应用**

项目中合理使用了多种经典设计模式提升了代码的灵活性与复用性。例如：

装饰器模式（Decorator Pattern）在文档解析部分得到灵活运用，AbstractTermTupleFilter 对原始的 AbstractTermTupleScanner 进行链式封装，使得词项过滤逻辑（如去除非法字符、去重等）可组合扩展，便于构建灵活的解析流水线。

策略模式（Strategy Pattern）用于查询结果排序功能，Sort 接口定义了统一的排序策略，通过实现不同子类如 SimpleSorter，可以轻松切换排序算法，增强了系统可配置性。

**（4）模块独立、可扩展性强**

项目结构中，索引构建、索引保存、加载、查询执行与结果排序均通过独立模块实现，彼此通过接口解耦。这种设计不仅便于模块复用，也为今后引入复杂功能（如增量索引、多字段支持、查询缓存等）打下基础。

**（5）测试流程完善、输出直观**

通过 TestSearchIndex 类对各模块功能进行测试，测试输出中详细展示了每个查询词的命中文档、出现位置、出现频率和得分，便于用户理解检索原理与运行结果。此外，项目提供了查询示例如单词查询、AND查询和OR查询，覆盖了基本用例，为后续功能拓展和展示提供了良好基础。

1. **不足和改进的建议**

**（1）缺乏用户界面和包装**

目前系统仅支持手动运行测试类进行使用，没有包装成exe文件，缺乏图形用户界面或 Web 页面，不利于用户体验与演示使用。且相关配置均写在config文件中，不能在使用时动态调整，需要用户有一定的计算机基础，尽管实验没有做这方面的要求，但可以提供一些帮助文档和建议供学有余力的同学将继续深入。

**（2）功能较为单一**

目前系统虽然完成了需求，但功能较少，例如排序接口只实现了一个简单排序，查询功能仅支持布尔查询与词频排序，缺乏短语查询、前缀匹配、模糊搜索、通配符等高级查询能力，无法满足实际搜索应用中用户的多样化需求。当然，更高级的算法不是本实验的重点，本实验注重于系统结构和设计模式，但对Java的各种特性和API却涉及不够广泛，以后实验可以考虑在这方面进行补充。

**（3）自动测试不够直观**

实验提供的自动测试包测试用例齐全，但输出不够直观，不便于同学Debug和分步完成实验，可以考虑将实验迁移到头歌平台，增加“闯关”的体验。

1. **过程和体会**
2. **遇到的主要问题和解决方法**

**（1）序列化和反序列化理解不清**

实验遇到的第一个难关便是writeObject和readObject函数，由于对其概念和相关API过于陌生，起初无从下手。查询相关资料和实验文档后，我了解到序列化是指把Java对象转换为字节序列（binary stream）的过程，从而能将对象保存到磁盘、通过网络传输。Java 中序列化的核心机制是实现 Serializable 接口，而本实验定义了接口 FileSerializable，Index 类中用到了 FileSerializable， 写入阶段（save() 方法）：用 ObjectOutputStream 将两个重要字段写入文件中；调用的是自己实现的 writeObject() 方法，不是 Java 自动的那个。读取阶段（load() 方法）：从 ObjectInputStream 反序列化读取两个字段；通过 readObject() 恢复完整的 Index 对象状态。

**（2）错误理解排序接口的要求**

在实现Sort接口时我曾遇到多个Bug，经过反复尝试后发现，score得分应该是命中单词的总频率，而排序则应该降序，由于sort函数默认升序，我应该手动传入一个Lambda表达式，通过Double.compare()函数,实现降序。

**（3）自动测试定位Bug时遇到麻烦**

由于自动测试的输出过于混乱，起初我未能明白测试用例是什么，错误发生在哪里，后来发现文件夹中会生成一个测试报告，通过浏览器打开，我可以大致定位到错误发生的地点，从而进行修改

**（4）对API不够熟练**

实验中用到了很多Java的集合框架和相关内置函数，例如List，Map，String，通过使用这些API，可以很轻松地实现数据存储，映射，字符串分割等功能，通过向AI询问需求和查询Java文档，我对各种类的使用更加熟练。

1. **课程设计的体会**

在专业知识方面，我深入理解了倒排索引这一关键数据结构的原理与优势。它通过将文档中的词项映射到包含该词项的文档列表，实现了词项到文档的快速定位，是现代搜索引擎中不可或缺的组成部分。通过本次实验，我不仅掌握了索引中 PostingList 的基本结构，还了解了其如何支持频次统计、词位位置记录等功能，为高级检索功能如短语搜索、临近查询等打下了基础。同时，我认识到在真实系统中，倒排索引常配合压缩技术和跳表等加速机制使用，这对提高存储效率和检索性能具有重要意义。

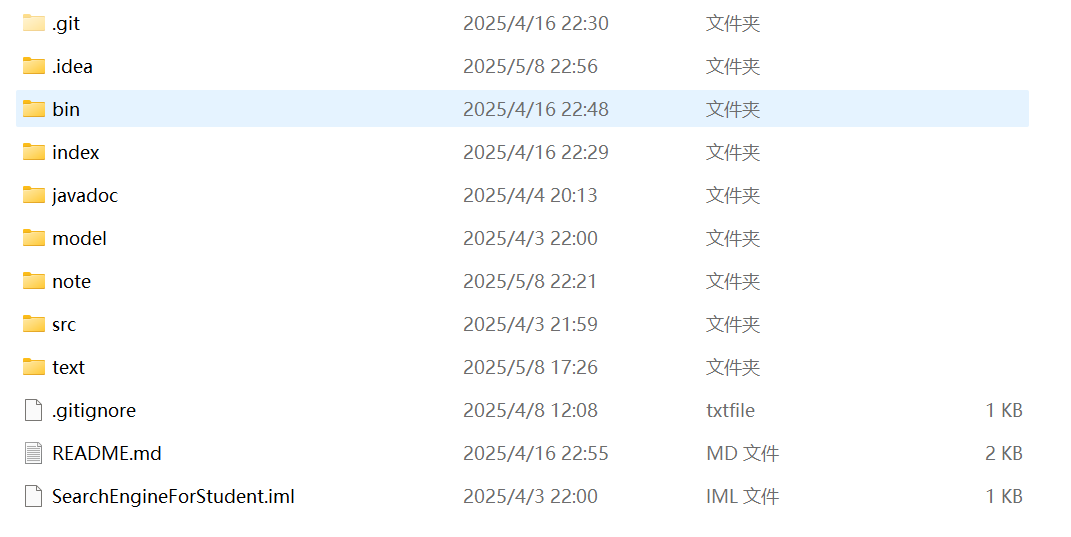
在编程技能方面，我全面提升了对 Java 语言核心特性的运用能力。集合类方面，我学习并灵活使用了 HashMap、TreeMap、ArrayList 等容器，尤其是 TreeMap 在倒排索引中对关键词进行自动排序，简化了索引字典的构建流程。在文件处理方面，我掌握了 BufferedReader 和 PrintWriter 的用法，能够高效读取文本文档并将索引结果持久化存储。更重要的是，我学会了如何使用 Java 的对象序列化与反序列化功能将内存结构写入磁盘文件，并在查询时快速恢复使用，显著提升了系统的可扩展性和持久性。

在工程设计方面，我深刻体会到良好架构对程序开发的重要性。本项目充分体现了“高内聚、低耦合”的面向对象设计理念。系统采用了多层抽象的架构，将功能逻辑拆分为多个层级：文档解析使用流式处理接口，三元组过滤采用装饰器模式，查询排序使用策略模式，不同模块职责明确，便于开发与调试。通过实现 AbstractTermTupleStream、AbstractIndex、AbstractIndexSearcher 等抽象类的具体子类，我加深了对接口继承、模块解耦和系统扩展性的理解。特别是在添加新功能（如多词查询、布尔逻辑）时，我能在不影响原有模块的前提下灵活扩展系统，这种开发方式大大提升了开发效率和系统的可维护性。

与上学期的C++实验相比，两者有很多相似之处，注重于工程能力的培养，C++实验逐步引导学生改进代码，而Java实验则提供了一个完善的框架，包含各种抽象类和JavaDoc文档，两者各有优点，或许可以进行一定的结合。

本次课程设计还锻炼了我自主学习和独立解决问题的能力。项目中有许多功能需查阅 Java 官方文档或查找开源实现做参考，过程中我学会了如何从具体问题出发，拆解成小任务并逐步解决。这种经验让我认识到编程不仅是对语法的掌握，更是对逻辑、结构、效率的系统把控。最终，我成功完成了索引构建、数据持久化、词项检索、布尔查询、结果排序等功能模块，并通过大量测试验证了系统的正确性和稳定性。

1. **源码和说明**
2. **文件清单及其功能说明**

本项目命名为 SearchEngineForStudent，目录结构清晰，主要包含以下文件和文件夹，各自功能如下表所示：

|  |  |  |
| --- | --- | --- |
| **文件/文件夹** | **类型** | **功能说明** |
| .idea/ | 配置目录 | IntelliJ IDEA 的项目配置目录，包含项目设置文件，不影响程序功能。 |
| bin/ | 输出目录 | Java 编译后的 .class 文件所在目录，含项目运行时所需的二进制文件。 |
| index/ | 数据目录 | 存储构建后的倒排索引数据，主要文件为 index.dat，在查询阶段用于加载索引。 |
| javadoc/ | 文档目录 | 存放通过 javadoc 工具生成的项目文档，便于开发者理解类与接口的使用。 |
| model/ | 模型结构图 | 包含类图（如 model.png、model.uml、model-带类之间关系.png），展示了系统各模块间的关系与继承结构，辅助理解系统架构。 |
| note/ | 学习笔记 | 用于记录学习笔记 |
| src/ | 源码目录 | 核心源码所在路径，其中包含完整的 Java 包结构 hust.cs.javacourse.search。 |
| src/hust/cs/javacourse/search/index/ | 索引模块 | 实现倒排索引的数据结构、索引构建与持久化等功能。 |
| src/hust/cs/javacourse/search/parse/ | 解析模块 | 负责文档的读取与词项三元组的提取。 |
| src/hust/cs/javacourse/search/query/ | 查询模块 | 实现布尔查询逻辑和排序机制。 |
| src/hust/cs/javacourse/search/run/ | 执行模块 | 包含主程序入口类，用于测试和运行整个搜索系统。 |
| src/hust/cs/javacourse/search/util/ | 工具模块 | 包含常用工具类，如序列化工具、配置读取等。 |
| text/ | 测试文档目录 | 存放样例文本文档 1.txt、2.txt、3.txt，用于索引构建与查询测试。 |
| .gitignore | Git配置 | 指定在版本控制中需要忽略的文件，如 bin/、临时文件等。 |
| README.md | 项目说明文档 | 简要介绍项目背景、功能、使用方法等信息，便于用户快速上手。 |
| SearchEngineForStudent.iml | IDEA模块文件 | IntelliJ IDEA 的模块配置文件。 |

1. **用户使用说明书**

**（1）Java开发环境**

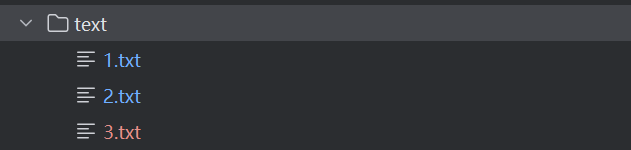
确保你的电脑安装了IntelliJ IDEA 或 Eclipse 等 Java IDE，建议使用Java17以上版本的JDE。

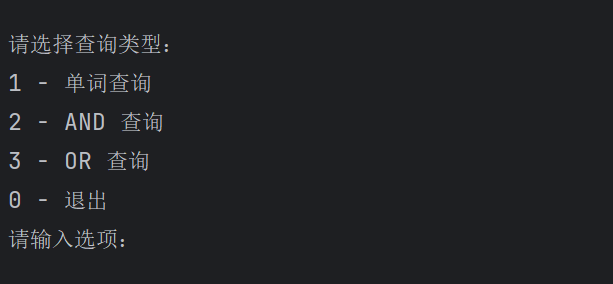
建议使用Windos10或更高版本的操作系统运行程序。

**（2）解压项目**

解压项目后，将项目文件夹（如 SearchEngineForStudent/）复制到本地任意目录。打开 IntelliJ IDEA，选择 Open，导入整个项目目录。确保 JDK 配置正确，IDE 会自动识别项目结构。使用 Build -> Build Project 进行编译，生成 .class 文件。

**（3）开始使用**

将你需要构建索引的文本文档(.txt格式)全部复制到项目的./text目录下，如下图所示：

接着运行src/hust/cs/javacourse/search/run/TestBuildIndex.java类，构建索引，输出文件保存至 index/index.dat，最后，运行src/hust/cs/javacourse/search/run/TestSearchIndexInteractive.java类，开始使用，根据查询需求进行输入。

1. **源代码**
2. **src/hust/cs/javacourse/search/index/impl**
3. **Term类**
4. package hust.cs.javacourse.search.index.impl;
5. import hust.cs.javacourse.search.index.AbstractTerm;
6. import java.io.IOException;
7. import java.io.ObjectInputStream;
8. import java.io.ObjectOutputStream;
9. import java.util.Objects;
10. public class Term extends AbstractTerm {
11. */\*\**
12. \* 缺省构造函数
13. \*/
14. public Term() {
15. super();
16. }
17. */\*\**
18. \* 构造函数
19. \*
20. \* @param content ：Term内容
21. \*/
22. public Term(String content) {
23. super(content);
24. }
25. */\*\**
26. \* 判断二个Term内容是否相同
27. \*
28. \* @param obj ：要比较的另外一个Term
29. \* @return 如果内容相等返回true，否则返回false
30. \*/
31. @Override
32. public boolean equals(Object obj) {
33. if (this == obj) return true;
34. if (obj == null || getClass() != obj.getClass()) return false;
35. Term term = (Term) obj;
36. return Objects.equals(content, term.content);
37. }
38. */\*\**
39. \* 返回Term的字符串表示
40. \*
41. \* @return 字符串
42. \*/
43. @Override
44. public String toString() {
45. return this.content;
46. }
47. */\*\**
48. \* 返回Term内容
49. \*
50. \* @return Term内容
51. \*/
52. @Override
53. public String getContent() {
54. return this.content;
55. }
56. */\*\**
57. \* 设置Term内容
58. \*
59. \* @param content：Term的内容
60. \*/
61. @Override
62. public void setContent(String content) {
63. this.content = content;
64. }
65. */\*\**
66. \* 比较二个Term大小（按字典序）
67. \*
68. \* @param o： 要比较的Term对象
69. \* @return ： 返回二个Term对象的字典序差值
70. \*/
71. @Override
72. public int compareTo(AbstractTerm o) {
73. return this.content.compareTo(o.getContent());
74. }
75. */\*\**
76. \* 写到二进制文件
77. \*
78. \* @param out :输出流对象
79. \*/
80. @Override
81. public void writeObject(ObjectOutputStream out) throws IOException {
82. if (content == null) {
83. out.writeInt(-1); *// 使用 -1 表示 content 为 null*
84. } else {
85. out.writeUTF(content); *// 直接写入字符串*
86. }
87. }
88. */\*\**
89. \* 从二进制文件读
90. \*
91. \* @param in ：输入流对象
92. \*/
93. @Override
94. public void readObject(ObjectInputStream in) throws IOException {
95. this.content = in.readUTF(); *// 直接读取字符串*
96. }
97. }
98. **TermTuple类**
99. package hust.cs.javacourse.search.index.impl;
100. import hust.cs.javacourse.search.index.AbstractTerm;
101. import hust.cs.javacourse.search.index.AbstractTermTuple;
102. public class TermTuple extends AbstractTermTuple {
103. */\*\**
104. \* 缺省构造函数
105. \*/
106. public TermTuple() {
107. super();
108. }
109. */\*\**
110. \* 构造函数
111. \* @param term ：单词
112. \* @param curPos ：单词出现的当前位置
113. \*/
114. public TermTuple(AbstractTerm term, int curPos) {
115. this.term = term;
116. this.curPos = curPos;
117. }
118. */\*\**
119. \* 判断二个三元组内容是否相同
120. \* @param obj ：要比较的另外一个三元组
121. \* @return 如果内容相等（三个属性内容都相等）返回true，否则返回false
122. \*/
123. @Override
124. public boolean equals(Object obj) {
125. if (this == obj) return true; *// 如果是同一个对象，直接返回 true*
126. if (obj == null || getClass() != obj.getClass()) return false; *// 如果对象为 null 或类型不同，返回 false*
127. TermTuple termTuple = (TermTuple) obj; *// 强制转换为 TermTuple 类型*
128. *// 比较 freq*
129. *// 比较 curPos*
130. return this.curPos == termTuple.curPos && (this.term != null ? this.term.equals(termTuple.term) : termTuple.term == null); *// 比较 term*
131. }
132. */\*\**
133. \* 获得三元组的字符串表示
134. \* @return ： 三元组的字符串表示
135. \*/
136. @Override
137. public String toString() {
138. return "TermTuple{" +
139. "term=" + term.toString() +
140. ", freq=" + freq +
141. ", curPos=" + curPos +
142. '}';
143. }
144. }
145. **Posting类**
146. package hust.cs.javacourse.search.index.impl;
147. import hust.cs.javacourse.search.index.AbstractPosting;
148. import java.io.IOException;
149. import java.io.ObjectInputStream;
150. import java.io.ObjectOutputStream;
151. import java.util.ArrayList;
152. import java.util.Collections;
153. import java.util.List;
154. import java.util.StringJoiner;
155. public class Posting extends AbstractPosting {
156. */\*\**
157. \* 缺省构造函数
158. \*/
159. public Posting() {
160. super();
161. }
162. */\*\**
163. \* 构造函数
164. \* @param docId ：包含单词的文档id
165. \* @param freq  ：单词在文档里出现的次数
166. \* @param positions   ：单词在文档里出现的位置
167. \*/
168. public Posting(int docId, int freq, List<Integer> positions) {
169. super(docId, freq, positions);
170. }
171. */\*\**
172. \* 判断二个Posting内容是否相同
173. \* @param obj ：要比较的另外一个Posting
174. \* @return 如果内容相等返回true，否则返回false
175. \*/
176. @Override
177. public boolean equals(Object obj) {
178. if (this == obj) return true; *// 如果是同一个对象，直接返回 true*
179. if (obj == null || getClass() != obj.getClass()) return false; *// 如果对象为 null 或类型不同，返回 false*
180. Posting posting = (Posting) obj;
181. *// 比较 docId 和 freq*
182. if (this.docId != posting.docId || this.freq != posting.freq) return false;
183. *// 如果 positions 列表大小不同，直接返回 false*
184. if (this.positions.size() != posting.positions.size()) return false;
185. *// 确保 positions 列表已经排序*
186. this.sort();
187. posting.sort();
188. *// 遍历比较 positions 列表的每个元素*
189. for (int i = 0; i < this.positions.size(); i++) {
190. if (!this.positions.get(i).equals(posting.positions.get(i))) {
191. return false;
192. }
193. }
194. return true;
195. }
196. */\*\**
197. \* 返回Posting的字符串表示
198. \* @return 字符串
199. \*/
200. @Override
201. public String toString() {
202. StringJoiner sj = new StringJoiner(", ", "{", "}");
203. sj.add("\"docID\": " + this.docId);
204. sj.add("\"freq\": " + this.freq);
205. StringJoiner positionsJoiner = new StringJoiner(", ", "\"positions\": [", "]");
206. for (int pos : this.positions) {
207. positionsJoiner.add(String.valueOf(pos));
208. }
209. sj.add(positionsJoiner.toString());
210. return sj.toString();
211. }
212. */\*\**
213. \* 返回包含单词的文档id
214. \* @return ：文档id
215. \*/
216. @Override
217. public int getDocId() {
218. return docId;
219. }
220. */\*\**
221. \* 设置包含单词的文档id
222. \* @param docId：包含单词的文档id
223. \*/
224. @Override
225. public void setDocId(int docId) {
226. this.docId = docId;
227. }
228. */\*\**
229. \* 返回单词在文档里出现的次数
230. \* @return ：出现次数
231. \*/
232. @Override
233. public int getFreq() {
234. return freq;
235. }
236. */\*\**
237. \* 设置单词在文档里出现的次数
238. \* @param freq:单词在文档里出现的次数
239. \*/
240. @Override
241. public void setFreq(int freq) {
242. this.freq = freq;
243. }
244. */\*\**
245. \* 返回单词在文档里出现的位置列表
246. \* @return ：位置列表
247. \*/
248. @Override
249. public List<Integer> getPositions() {
250. return positions;
251. }
252. */\*\**
253. \* 设置单词在文档里出现的位置列表
254. \* @param positions：单词在文档里出现的位置列表
255. \*/
256. @Override
257. public void setPositions(List<Integer> positions) {
258. this.positions = positions;
259. }
260. */\*\**
261. \* 比较二个Posting对象的大小（根据docId）
262. \* @param o： 另一个Posting对象
263. \* @return ：二个Posting对象的docId的差值
264. \*/
265. @Override
266. public int compareTo(AbstractPosting o) {
267. return this.docId - o.getDocId();
268. }
269. */\*\**
270. \* 对内部positions从小到大排序
271. \*/
272. @Override
273. public void sort() {
274. Collections.sort(positions);
275. }
276. */\*\**
277. \* 将Posting对象序列化到二进制文件中
278. \* @param out ：输出流
279. \* @throws IOException ：如果发生I/O错误
280. \*/
281. @Override
282. public void writeObject(ObjectOutputStream out) throws IOException {
283. out.writeInt(docId);
284. out.writeInt(freq);
285. out.writeObject(positions);
286. }
287. */\*\**
288. \* 从二进制文件中反序列化Posting对象
289. \* @param in ：输入流
290. \* @throws IOException ：如果发生I/O错误
291. \* @throws ClassNotFoundException ：如果类未找到
292. \*/
293. @Override
294. public void readObject(ObjectInputStream in) throws IOException {
295. this.docId = in.readInt();
296. this.freq = in.readInt();
297. try{
298. this.positions = (List<Integer>) in.readObject();
299. }catch(ClassNotFoundException e){}
300. }
301. }
302. **PostingList类**
303. package hust.cs.javacourse.search.index.impl;
304. import hust.cs.javacourse.search.index.AbstractPosting;
305. import hust.cs.javacourse.search.index.AbstractPostingList;
306. import java.io.IOException;
307. import java.io.ObjectInputStream;
308. import java.io.ObjectOutputStream;
309. import java.util.Comparator;
310. import java.util.List;
311. public class PostingList extends AbstractPostingList {
312. */\*\**
313. \* 添加Posting,要求不能有内容重复的posting
314. \* @param posting：Posting对象
315. \*/
316. @Override
317. public void add(AbstractPosting posting) {
318. if (!this.list.contains(posting)) {
319. this.list.add(posting);
320. }
321. }
322. */\*\**
323. \* 获得PosingList的字符串表示
324. \* @return ： PosingList的字符串表示
325. \*/
326. @Override
327. public String toString() {
328. StringBuilder sb = new StringBuilder("PostingList [\n");
329. for (AbstractPosting posting : this.list) {
330. sb.append("    ").append(posting.toString()).append("\n");
331. }
332. sb.append("]");
333. return sb.toString();
334. }
335. */\*\**
336. \* 添加Posting列表,,要求不能有内容重复的posting
337. \* @param postings：Posting列表
338. \*/
339. @Override
340. public void add(List<AbstractPosting> postings) {
341. for (AbstractPosting posting : postings) {
342. this.add(posting);
343. }
344. }
345. */\*\**
346. \* 返回指定下标位置的Posting
347. \* @param index ：下标
348. \* @return 指定下标位置的Posting
349. \*/
350. @Override
351. public AbstractPosting get(int index) {
352. return this.list.get(index);
353. }
354. */\*\**
355. \* 返回指定Posting对象的下标
356. \* @param posting：指定的Posting对象
357. \* @return ：如果找到返回对应下标；否则返回-1
358. \*/
359. @Override
360. public int indexOf(AbstractPosting posting) {
361. return this.list.indexOf(posting);
362. }
363. */\*\**
364. \* 返回指定文档id的Posting对象的下标
365. \* @param docId ：文档id
366. \* @return ：如果找到返回对应下标；否则返回-1
367. \*/
368. @Override
369. public int indexOf(int docId) {
370. for (int i = 0; i < this.list.size(); i++) {
371. if (this.list.get(i).getDocId() == docId) {
372. return i;
373. }
374. }
375. return -1;
376. }
377. */\*\**
378. \* 是否包含指定Posting对象
379. \* @param posting： 指定的Posting对象
380. \* @return : 如果包含返回true，否则返回false
381. \*/
382. @Override
383. public boolean contains(AbstractPosting posting) {
384. return this.list.contains(posting);
385. }
386. */\*\**
387. \* 删除指定下标的Posting对象
388. \* @param index：指定的下标
389. \*/
390. @Override
391. public void remove(int index) {
392. this.list.remove(index);
393. }
394. */\*\**
395. \* 删除指定的Posting对象
396. \* @param posting ：定的Posting对象
397. \*/
398. @Override
399. public void remove(AbstractPosting posting) {
400. this.list.remove(posting);
401. }
402. */\*\**
403. \* 返回PostingList的大小，即包含的Posting的个数
404. \* @return ：PostingList的大小
405. \*/
406. @Override
407. public int size() {
408. return this.list.size();
409. }
410. */\*\**
411. \* 清除PostingList
412. \*/
413. @Override
414. public void clear() {
415. this.list.clear();
416. }
417. */\*\**
418. \* PostingList是否为空
419. \* @return 为空返回true;否则返回false
420. \*/
421. @Override
422. public boolean isEmpty() {
423. return this.list.isEmpty();
424. }
425. */\*\**
426. \* 根据文档id的大小对PostingList进行从小到大的排序
427. \*/
428. @Override
429. public void sort() {
430. this.list.sort(new Comparator<AbstractPosting>() {
431. @Override
432. public int compare(AbstractPosting o1, AbstractPosting o2) {
433. return o1.getDocId() - o2.getDocId();
434. }
435. });
436. }
437. */\*\**
438. \* 将PostingList序列化到二进制文件中
439. \* @param out ：输出流
440. \* @throws IOException ：如果发生I/O错误
441. \*/
442. @Override
443. public void writeObject(ObjectOutputStream out) throws IOException {
444. out.writeInt(this.list.size());
445. for (AbstractPosting posting : this.list) {
446. posting.writeObject(out);
447. }
448. }
449. */\*\**
450. \* 从二进制文件中反序列化PostingList
451. \* @param in ：输入流
452. \* @throws IOException ：如果发生I/O错误
453. \*/
454. @Override
455. public void readObject(ObjectInputStream in) throws IOException {
456. int size = in.readInt();
457. this.list.clear();
458. for (int i = 0; i < size; i++) {
459. Posting posting = new Posting();
460. posting.readObject(in);
461. this.list.add(posting);
462. }
463. }
464. }
465. **Document类**
466. package hust.cs.javacourse.search.index.impl;
467. import hust.cs.javacourse.search.index.AbstractDocument;
468. import hust.cs.javacourse.search.index.AbstractTermTuple;
469. import java.util.ArrayList;
470. import java.util.List;
471. public class Document extends AbstractDocument {
472. */\*\**
473. \* 缺省构造函数
474. \*/
475. public Document() {
476. super();
477. }
478. */\*\**
479. \* 构造函数
480. \* @param docId：文档id
481. \* @param docPath：文档绝对路径
482. \*/
483. public Document(int docId, String docPath) {
484. super(docId, docPath);
485. }
486. */\*\**
487. \* 构造函数
488. \* @param docId：文档id
489. \* @param docPath：文档绝对路径
490. \* @param tuples：文档包含的三元组列表
491. \*/
492. public Document(int docId, String docPath, List<AbstractTermTuple> tuples) {
493. super(docId, docPath, tuples);
494. }
495. */\*\**
496. \* 获得文档id
497. \* @return ：文档id
498. \*/
499. @Override
500. public int getDocId() {
501. return this.docId;
502. }
503. */\*\**
504. \* 设置文档id
505. \* @param docId：文档id
506. \*/
507. @Override
508. public void setDocId(int docId) {
509. this.docId = docId;
510. }
511. */\*\**
512. \* 获得文档绝对路径
513. \* @return ：文档绝对路径
514. \*/
515. @Override
516. public String getDocPath() {
517. return this.docPath;
518. }
519. */\*\**
520. \* 设置文档绝对路径
521. \* @param docPath：文档绝对路径
522. \*/
523. @Override
524. public void setDocPath(String docPath) {
525. this.docPath = docPath;
526. }
527. */\*\**
528. \* 获得文档包含的三元组列表
529. \* @return ：文档包含的三元组列表
530. \*/
531. @Override
532. public List<AbstractTermTuple> getTuples() {
533. return this.tuples;
534. }
535. */\*\**
536. \* 向文档对象里添加三元组, 要求不能有内容重复的三元组
537. \* @param tuple ：要添加的三元组
538. \*/
539. @Override
540. public void addTuple(AbstractTermTuple tuple) {
541. if (!this.contains(tuple)) {
542. this.tuples.add(tuple);
543. }
544. }
545. */\*\**
546. \* 判断是否包含指定的三元组
547. \* @param tuple ： 指定的三元组
548. \* @return ： 如果包含指定的三元组，返回true;否则返回false
549. \*/
550. @Override
551. public boolean contains(AbstractTermTuple tuple) {
552. return this.tuples.contains(tuple);
553. }
554. */\*\**
555. \* 获得指定下标位置的三元组
556. \* @param index：指定下标位置
557. \* @return：三元组
558. \*/
559. @Override
560. public AbstractTermTuple getTuple(int index) {
561. return this.tuples.get(index);
562. }
563. */\*\**
564. \* 返回文档对象包含的三元组的个数
565. \* @return ：文档对象包含的三元组的个数
566. \*/
567. @Override
568. public int getTupleSize() {
569. return this.tuples.size();
570. }
571. */\*\**
572. \* 获得Document的字符串表示
573. \* @return ： Document的字符串表示
574. \*/
575. @Override
576. public String toString() {
577. StringBuilder sb = new StringBuilder("Document {\n");
578. sb.append("    docId = ").append(this.docId).append(",\n");
579. sb.append("    docPath = ").append(this.docPath).append(",\n");
580. sb.append("    tuples = [\n");
581. for (AbstractTermTuple tuple : this.tuples) {
582. sb.append("        ").append(tuple.toString()).append(",\n");
583. }
584. if (!this.tuples.isEmpty()) {
585. sb.deleteCharAt(sb.length() - 2); *// 删除最后一个三元组后的逗号和换行符*
586. }
587. sb.append("    ]\n");
588. sb.append("}");
589. return sb.toString();
590. }
591. }
592. **DocunmentBulider类**
593. package hust.cs.javacourse.search.index.impl;
594. import hust.cs.javacourse.search.index.AbstractDocument;
595. import hust.cs.javacourse.search.index.AbstractDocumentBuilder;
596. import hust.cs.javacourse.search.index.AbstractTermTuple;
597. import hust.cs.javacourse.search.parse.AbstractTermTupleStream;
598. import hust.cs.javacourse.search.parse.impl.LengthTermTupleFilter;
599. import hust.cs.javacourse.search.parse.impl.PatternTermTupleFilter;
600. import hust.cs.javacourse.search.parse.impl.StopWordTermTupleFilter;
601. import hust.cs.javacourse.search.parse.impl.TermTupleScanner;
602. import java.io.\*;
603. */\*\**
604. \* AbstractDocumentBuilder的具体实现类
605. \*/
606. public class DocumentBuilder extends AbstractDocumentBuilder {
607. */\*\**
608. \* 缺省构造函数
609. \*/
610. public DocumentBuilder() {
611. }
612. */\*\**
613. \* 由解析文本文档得到的TermTupleStream,构造Document对象.
614. \* @param docId             : 文档id
615. \* @param docPath           : 文档绝对路径
616. \* @param termTupleStream   : 文档对应的TermTupleStream
617. \* @return ：Document对象
618. \*/
619. @Override
620. public AbstractDocument build(int docId, String docPath, AbstractTermTupleStream termTupleStream) {
621. AbstractDocument document = new Document(docId, docPath);
622. AbstractTermTuple tuple;
623. while ((tuple = termTupleStream.next()) != null) {
624. document.getTuples().add(tuple); *// 将三元组添加到文档中*
625. }
626. return document;
627. }
628. */\*\**
629. \* 由给定的File,构造Document对象.
630. \* 该方法利用输入参数file构造出AbstractTermTupleStream子类对象后,内部调用
631. \*      AbstractDocument build(int docId, String docPath, AbstractTermTupleStream termTupleStream)
632. \* @param docId     : 文档id
633. \* @param docPath   : 文档绝对路径
634. \* @param file      : 文档对应File对象
635. \* @return          : Document对象
636. \*/
637. @Override
638. public AbstractDocument build(int docId, String docPath, File file) {
639. AbstractTermTupleStream ts;
640. try {
641. BufferedReader reader = new BufferedReader(new InputStreamReader(new FileInputStream(file)));
642. *// 创建 TermTupleScanner*
643. *// 添加过滤器*
644. ts = new LengthTermTupleFilter(new PatternTermTupleFilter
645. (new StopWordTermTupleFilter
646. (new TermTupleScanner(reader))));
647. } catch (FileNotFoundException e) {
648. throw new RuntimeException(e); *// 如果文件未找到，抛出运行时异常*
649. }
650. return this.build(docId, docPath, ts);
651. }
652. }
653. **Index类**
654. package hust.cs.javacourse.search.index.impl;
655. import hust.cs.javacourse.search.index.\*;
656. import java.io.\*;
657. import java.util.\*;
658. import java.util.logging.Level;
659. import java.util.logging.Logger;
660. public class Index extends AbstractIndex {
661. */\*\**
662. \* 返回索引的字符串表示
663. \* @return 索引的字符串表示
664. \*/
665. @Override
666. public String toString() {
667. StringBuilder sb = new StringBuilder();
668. *// 输出 docId 到 docPath 的映射*
669. sb.append("Document ID to Document Path Mapping:\n");
670. sb.append("------------------------------------\n");
671. for (int docId : this.docIdToDocPathMapping.keySet()) {
672. sb.append(String.format("%-5d ---> %s\n", docId, this.docIdToDocPathMapping.get(docId)));
673. }
674. sb.append("\nTotal Mappings: ").append(this.docIdToDocPathMapping.size()).append("\n\n");
675. *// 输出 term 到 postingList 的映射*
676. sb.append("Term to PostingList Mapping:\n");
677. sb.append("----------------------------\n");
678. for (AbstractTerm term : this.termToPostingListMapping.keySet()) {
679. sb.append(String.format("%-20s ---> %s\n", term.toString(), this.termToPostingListMapping.get(term).toString()));
680. }
681. sb.append("\nTotal Mappings: ").append(this.termToPostingListMapping.size()).append("\n");
682. return sb.toString();
683. }
684. */\*\**
685. \* 将文档对象加入索引结构
686. \* @param document ：一个文档对象
687. \*/
688. @Override
689. public void addDocument(AbstractDocument document){
690. int docId = document.getDocId();         *// 获取文档ID*
691. String docPath = document.getDocPath();  *// 获取文档路径*
692. docIdToDocPathMapping.put(docId, docPath);  *// 建立ID与路径的映射*
693. for (AbstractTermTuple tuple : document.getTuples()) {
694. AbstractTerm term = tuple.term;  *// 获取词项*
695. AbstractPostingList postingList = termToPostingListMapping.get(term);  *// 获取该词项的PostingList*
696. if (postingList == null) {
697. postingList = new PostingList(); *// 如果没有，则创建一个新的*
698. }
699. int docIndex = postingList.indexOf(docId); *// 查看该PostingList中是否已有此文档*
700. AbstractPosting posting;
701. if (docIndex != -1) {
702. posting = postingList.get(docIndex);  *// 已有则取出*
703. } else {
704. posting = new Posting(docId, 0, new ArrayList<>());  *// 否则新建一个*
705. postingList.add(posting);  *// 添加新的Posting*
706. }
707. posting.setFreq(posting.getFreq() + 1);  *// 更新词频*
708. posting.getPositions().add(tuple.curPos); *// 添加词出现的位置*
709. termToPostingListMapping.put(term, postingList); *// 更新 term 到 PostingList 的映射*
710. }
711. }
712. */\*\**
713. \* 从文件中加载索引结构
714. \* @param file ：已存储索引的文件
715. \*/
716. @Override
717. public void load(File file){
718. try {
719. readObject(new ObjectInputStream(new FileInputStream(file)));
720. } catch (Exception e){
721. *// 使用日志框架记录异常*
722. Logger.getLogger(Index.class.getName()).log(Level.SEVERE, "Failed to load index from file", e);
723. }
724. }
725. */\*\**
726. \* 将索引保存到文件中
727. \* @param file ：目标文件
728. \*/
729. @Override
730. public void save(File file){
731. try {
732. writeObject(new ObjectOutputStream(new FileOutputStream(file)));
733. } catch (Exception e){
734. *// 使用日志框架记录异常*
735. Logger.getLogger(Index.class.getName()).log(Level.SEVERE, "Failed to save index to file", e);
736. }
737. }
738. */\*\**
739. \* 搜索指定词项对应的PostingList
740. \* @param term : 查询的词项
741. \* @return ：返回对应的PostingList（可为null）
742. \*/
743. @Override
744. public AbstractPostingList search(AbstractTerm term){
745. return termToPostingListMapping.get(term);
746. }
747. */\*\**
748. \* 获取索引中所有词项的集合（即字典）
749. \* @return ：Set<AbstractTerm>
750. \*/
751. @Override
752. public Set<AbstractTerm> getDictionary(){
753. return termToPostingListMapping.keySet();
754. }
755. */\*\**
756. \* 优化索引：对PostingList和词位位置进行排序
757. \*/
758. @Override
759. public void optimize(){
760. for (AbstractTerm term : getDictionary()) {
761. AbstractPostingList postingList = termToPostingListMapping.get(term);
762. postingList.sort(); *// 对postingList按docId升序*
763. for (int i = 0; i < postingList.size(); i++) {
764. postingList.get(i).sort(); *// 对每个posting的positions排序*
765. }
766. }
767. }
768. */\*\**
769. \* 根据docId获取文档路径
770. \* @param docId ：文档ID
771. \* @return ：完整路径字符串
772. \*/
773. @Override
774. public String getDocName(int docId){
775. return docIdToDocPathMapping.get(docId);
776. }
777. */\*\**
778. \* 将索引对象写入到输出流中（序列化）
779. \* @param out : ObjectOutputStream输出流
780. \*/
781. @Override
782. public void writeObject(ObjectOutputStream out) {
783. try {
784. *// 1. 将文档ID与路径的映射写出*
785. out.writeObject(docIdToDocPathMapping);
786. *// 2. 将词项与其PostingList的映射写出*
787. out.writeObject(termToPostingListMapping);
788. } catch (Exception e) {
789. *// 使用日志框架记录异常*
790. Logger.getLogger(Index.class.getName()).log(Level.SEVERE, "Failed to write object to output stream", e);
791. }
792. }
793. */\*\**
794. \* 从输入流中读取索引对象（反序列化）
795. \* @param in ：ObjectInputStream输入流
796. \*/
797. @Override
798. @SuppressWarnings("unchecked")
799. public void readObject(ObjectInputStream in) {
800. try {
801. *// 1. 从输入流中读取文档ID到路径的映射*
802. Object docIdToDocPathMappingObj = in.readObject();
803. if (docIdToDocPathMappingObj instanceof Map) {
804. docIdToDocPathMapping = (Map<Integer, String>) docIdToDocPathMappingObj;
805. } else {
806. throw new ClassCastException("Expected Map<Integer, String>, but got " + docIdToDocPathMappingObj.getClass());
807. }
808. *// 2. 读取词项到PostingList的映射*
809. Object termToPostingListMappingObj = in.readObject();
810. if (termToPostingListMappingObj instanceof Map) {
811. termToPostingListMapping = (Map<AbstractTerm, AbstractPostingList>) termToPostingListMappingObj;
812. } else {
813. throw new ClassCastException("Expected Map<AbstractTerm, AbstractPostingList>, but got " + termToPostingListMappingObj.getClass());
814. }
815. } catch (Exception e) {
816. *// 使用日志框架记录异常*
817. Logger.getLogger(Index.class.getName()).log(Level.SEVERE, "Failed to read object from input stream", e);
818. }
819. }
820. }
821. **IndexBuilder类**
822. package hust.cs.javacourse.search.index.impl;
823. import hust.cs.javacourse.search.index.AbstractDocument;
824. import hust.cs.javacourse.search.index.AbstractDocumentBuilder;
825. import hust.cs.javacourse.search.index.AbstractIndex;
826. import hust.cs.javacourse.search.index.AbstractIndexBuilder;
827. import hust.cs.javacourse.search.util.FileUtil;
828. import java.io.File;
829. import java.util.List;
830. */\*\**
831. \* <pre>
832. \* IndexBuilder是AbstractIndexBuilder的具体子类，用于构建指定目录下的所有文本文件的倒排索引。
833. \* 该类通过遍历指定目录下的所有文本文件，解析每个文件并构建Document对象，然后将这些Document对象加入到索引中。
834. \* </pre>
835. \*/
836. public class IndexBuilder extends AbstractIndexBuilder {
837. */\*\**
838. \* 构造函数，初始化父类的AbstractDocumentBuilder对象
839. \*
840. \* @param docBuilder ：用于构建Document对象的AbstractDocumentBuilder子类对象
841. \*/
842. public IndexBuilder(AbstractDocumentBuilder docBuilder) {
843. super(docBuilder);
844. }
845. */\*\**
846. \* <pre>
847. \* 构建指定目录下的所有文本文件的倒排索引.
848. \* 该方法首先创建一个新的Index对象，然后遍历指定目录下的所有文本文件，解析每个文件并构建Document对象，
849. \* 最后将这些Document对象加入到索引中，并返回构建好的索引。
850. \* @param rootDirectory ：指定目录
851. \* @return ：构建好的索引
852. \* </pre>
853. \*/
854. @Override
855. public AbstractIndex buildIndex(String rootDirectory) {
856. *// 创建一个新的Index对象，用于存储构建好的索引*
857. AbstractIndex index = new Index();
858. *// 使用FileUtil工具类获取指定目录下所有以.txt结尾的文本文件路径*
859. List<String> docPaths = FileUtil.list(rootDirectory, ".txt");
860. *// 遍历每个文件路径*
861. for (String docPath : docPaths) {
862. *// 将文件路径转换为File对象*
863. File file = new File(docPath);
864. *// 使用docBuilder构建Document对象，传入当前的docId、文件路径和File对象*
865. AbstractDocument document = docBuilder.build(docId, docPath, file);
866. *// 将构建好的Document对象加入到索引中*
867. index.addDocument(document);
868. *// docId计数器*
869. docId++;
870. }
871. *// 返回构建好的索引*
872. return index;
873. }
874. }
875. **src/hust/cs/javacourse/search/parse/impl**
876. **TermTupleScanner类**
877. package hust.cs.javacourse.search.parse.impl;
878. import hust.cs.javacourse.search.index.AbstractTermTuple;
879. import hust.cs.javacourse.search.index.impl.Term;
880. import hust.cs.javacourse.search.index.impl.TermTuple;
881. import hust.cs.javacourse.search.parse.AbstractTermTupleScanner;
882. import hust.cs.javacourse.search.util.Config;
883. import java.io.BufferedReader;
884. import java.io.IOException;
885. import java.util.ArrayDeque;
886. import java.util.Arrays;
887. import java.util.Deque;
888. public class TermTupleScanner extends AbstractTermTupleScanner {
889. private final Deque<String> buffer = new ArrayDeque<>(); *// 缓冲区，用于存储分割后的单词*
890. private int curPos = 0; *// 当前位置计数器*
891. */\*\**
892. \* 缺省构造函数
893. \*/
894. public TermTupleScanner() {
895. super();
896. }
897. */\*\**
898. \* 构造函数
899. \* @param input：指定输入流对象，应该关联到一个文本文件
900. \*/
901. public TermTupleScanner(BufferedReader input) {
902. super(input);
903. }
904. */\*\**
905. \* 获得下一个三元组
906. \* @return: 下一个三元组；如果到了流的末尾，返回null
907. \*/
908. @Override
909. public AbstractTermTuple next() {
910. *// 如果缓冲区为空，读取新行并分割*
911. while (buffer.isEmpty()) {
912. String line;
913. try {
914. line = input.readLine(); *// 读取一行*
915. } catch (IOException e) {
916. e.printStackTrace();
917. return null;
918. }
919. if (line == null) {
920. return null; *// 如果流结束，返回 null*
921. }
922. if (line.isBlank()) {
923. continue; *// 如果行是空行或空白行，跳过*
924. }
925. *// 将行按正则表达式分割，并根据配置决定是否转换为小写*
926. String[] parts = line.trim().split(Config.STRING\_SPLITTER\_REGEX);
927. *//trim()去掉首尾空格*
928. *//split()方法根据正则表达式分割字符串*
929. if (Config.IGNORE\_CASE) {
930. for (String part : parts) {
931. buffer.add(part.toLowerCase()); *// 忽略大小写，转换为小写*
932. }
933. } else {
934. buffer.addAll(Arrays.asList(parts)); *// 不忽略大小写，直接添加*
935. }
936. }
937. *// 从缓冲区取出第一个单词，生成三元组*
938. String termContent = buffer.removeFirst();
939. *//removeFirst()方法从缓冲区中删除并返回第一个元素*
940. return new TermTuple(new Term(termContent), curPos++);
941. }
942. }
943. **LengthTermTupleFilter类**
944. package hust.cs.javacourse.search.parse.impl;
945. import hust.cs.javacourse.search.index.AbstractTermTuple;
946. import hust.cs.javacourse.search.parse.AbstractTermTupleFilter;
947. import hust.cs.javacourse.search.parse.AbstractTermTupleStream;
948. import hust.cs.javacourse.search.util.Config;
949. */\*\**
950. \* 基于单词长度的三元组过滤器
951. \*/
952. public class LengthTermTupleFilter extends AbstractTermTupleFilter {
953. */\*\**
954. \* 构造函数
955. \* @param input：Filter的输入，类型为AbstractTermTupleStream
956. \*/
957. public LengthTermTupleFilter(AbstractTermTupleStream input) {
958. super(input);
959. }
960. */\*\**
961. \* 获得下一个三元组
962. \* @return: 下一个三元组；如果到了流的末尾，返回null
963. \*/
964. @Override
965. public AbstractTermTuple next() {
966. AbstractTermTuple tuple;
967. while ((tuple = input.next()) != null) {
968. int length = tuple.term.getContent().length();
969. if (length >= Config.TERM\_FILTER\_MINLENGTH && length <= Config.TERM\_FILTER\_MAXLENGTH) {
970. return tuple; *// 如果单词长度在范围内，返回三元组*
971. }
972. }
973. return null; *// 如果流结束，返回 null*
974. }
975. }
976. **StopWordTermTupleFilter类**
977. package hust.cs.javacourse.search.parse.impl;
978. import hust.cs.javacourse.search.index.AbstractTermTuple;
979. import hust.cs.javacourse.search.parse.AbstractTermTupleFilter;
980. import hust.cs.javacourse.search.parse.AbstractTermTupleStream;
981. import java.util.Arrays;
982. import static hust.cs.javacourse.search.util.StopWords.STOP\_WORDS;
983. public class StopWordTermTupleFilter extends AbstractTermTupleFilter {
984. */\*\**
985. \* 构造函数
986. \* @param input：Filter的输入，类型为AbstractTermTupleStream
987. \*/
988. public StopWordTermTupleFilter(AbstractTermTupleStream input) {
989. super(input);
990. }
991. */\*\**
992. \* 获得下一个三元组
993. \* @return: 下一个三元组；如果到了流的末尾，返回null
994. \*/
995. @Override
996. public AbstractTermTuple next() {
997. AbstractTermTuple tuple;
998. while ((tuple = input.next()) != null) {
999. if (!Arrays.asList(STOP\_WORDS).contains(tuple.term.getContent())) {
1000. return tuple; *// 如果不是停用词，返回三元组*
1001. }
1002. }
1003. return null; *// 如果流结束，返回 null*
1004. }
1005. }
1006. **PatternTermTupleFilter类**
1007. package hust.cs.javacourse.search.parse.impl;
1008. import hust.cs.javacourse.search.index.AbstractTermTuple;
1009. import hust.cs.javacourse.search.parse.AbstractTermTupleFilter;
1010. import hust.cs.javacourse.search.parse.AbstractTermTupleStream;
1011. import hust.cs.javacourse.search.util.Config;
1012. */\*\**
1013. \* 基于正则表达式的三元组过滤器
1014. \*/
1015. public class PatternTermTupleFilter extends AbstractTermTupleFilter {
1016. */\*\**
1017. \* 构造函数
1018. \* @param input：Filter的输入，类型为AbstractTermTupleStream
1019. \*/
1020. public PatternTermTupleFilter(AbstractTermTupleStream input) {
1021. super(input);
1022. }
1023. */\*\**
1024. \* 获得下一个三元组
1025. \* @return: 下一个三元组；如果到了流的末尾，返回null
1026. \*/
1027. @Override
1028. public AbstractTermTuple next() {
1029. AbstractTermTuple tuple;
1030. while ((tuple = input.next()) != null) {
1031. if (tuple.term.getContent().matches(Config.TERM\_FILTER\_PATTERN)) {
1032. return tuple; *// 如果单词符合正则表达式，返回三元组*
1033. }
1034. }
1035. return null; *// 如果流结束，返回 null*
1036. }
1037. }
1038. **src/hust/cs/javacourse/search/query/impl**
1039. **Hit类**
1040. package hust.cs.javacourse.search.query.impl;
1041. import hust.cs.javacourse.search.index.AbstractPosting;
1042. import hust.cs.javacourse.search.index.AbstractTerm;
1043. import hust.cs.javacourse.search.query.AbstractHit;
1044. import hust.cs.javacourse.search.util.FileUtil;
1045. import java.util.Map;
1046. import java.util.TreeMap;
1047. */\*\**
1048. \* <pre>
1049. \* Hit是AbstractHit的具体子类，表示一个搜索命中结果。
1050. \* 该类实现了Comparable接口，用于根据文档得分对命中结果进行排序。
1051. \* </pre>
1052. \*/
1053. public class Hit extends AbstractHit {
1054. */\*\**
1055. \* 默认构造函数
1056. \*/
1057. public Hit() {
1058. super();
1059. }
1060. */\*\**
1061. \* 构造函数
1062. \*
1063. \* @param docId   : 文档id
1064. \* @param docPath : 文档绝对路径
1065. \*/
1066. public Hit(int docId, String docPath) {
1067. super(docId, docPath);
1068. }
1069. */\*\**
1070. \* 构造函数
1071. \*
1072. \* @param docId              ：文档id
1073. \* @param docPath            ：文档绝对路径
1074. \* @param termPostingMapping ：命中的单词和对应的Posting键值对
1075. \*/
1076. public Hit(int docId, String docPath, Map<AbstractTerm, AbstractPosting> termPostingMapping) {
1077. super(docId, docPath, termPostingMapping);
1078. }
1079. */\*\**
1080. \* 获得文档id
1081. \*
1082. \* @return ：文档id
1083. \*/
1084. @Override
1085. public int getDocId() {
1086. return this.docId;
1087. }
1088. */\*\**
1089. \* 获得文档绝对路径
1090. \*
1091. \* @return ：文档绝对路径
1092. \*/
1093. @Override
1094. public String getDocPath() {
1095. return this.docPath;
1096. }
1097. */\*\**
1098. \* 获得文档内容
1099. \*
1100. \* @return ：文档内容
1101. \*/
1102. @Override
1103. public String getContent() {
1104. return this.content;
1105. }
1106. */\*\**
1107. \* 设置文档内容
1108. \*
1109. \* @param content ：文档内容
1110. \*/
1111. @Override
1112. public void setContent(String content) {
1113. this.content = content;
1114. }
1115. */\*\**
1116. \* 获得文档得分
1117. \*
1118. \* @return ：文档得分
1119. \*/
1120. @Override
1121. public double getScore() {
1122. return this.score;
1123. }
1124. */\*\**
1125. \* 设置文档得分
1126. \*
1127. \* @param score ：文档得分
1128. \*/
1129. @Override
1130. public void setScore(double score) {
1131. this.score = score;
1132. }
1133. */\*\**
1134. \* 获得命中的单词和对应的Posting键值对
1135. \*
1136. \* @return ：命中的单词和对应的Posting键值对
1137. \*/
1138. @Override
1139. public Map<AbstractTerm, AbstractPosting> getTermPostingMapping() {
1140. return this.termPostingMapping;
1141. }
1142. */\*\**
1143. \* 获得命中结果的字符串表示, 用于显示搜索结果.
1144. \*
1145. \* @return : 命中结果的字符串表示
1146. \*/
1147. @Override
1148. public String toString() {
1149. return "Hit{" +
1150. "docId=" + docId +
1151. ", docPath='" + docPath + '\'' +
1152. ", content='" + content + '\'' +
1153. ", termPostingMapping=" + termPostingMapping +
1154. ", score=" + score +
1155. '}';
1156. }
1157. */\*\**
1158. \* 比较二个命中结果的大小，根据score比较
1159. \*
1160. \* @param o ：要比较的命中结果
1161. \* @return ：二个命中结果得分的差值
1162. \*/
1163. @Override
1164. public int compareTo(AbstractHit o) {
1165. return (int) (this.score - o.getScore());
1166. }
1167. */\*\**
1168. \* 判断两个命中结果是否相等
1169. \*
1170. \* @param obj ：要比较的对象
1171. \* @return ：如果相等返回true;否则返回false
1172. \*/
1173. @Override
1174. public boolean equals(Object obj) {
1175. if (!(obj instanceof Hit other)) return false;
1176. boolean mappingEqual = (this.termPostingMapping == null && other.termPostingMapping == null)
1177. || (this.termPostingMapping != null && this.termPostingMapping.equals(other.termPostingMapping));
1178. return this.docId == other.docId &&
1179. Double.compare(this.score, other.score) == 0 &&
1180. mappingEqual;
1181. }
1182. }
1183. **SimpleSort类**
1184. package hust.cs.javacourse.search.query.impl;
1185. import hust.cs.javacourse.search.index.AbstractPosting;
1186. import hust.cs.javacourse.search.query.AbstractHit;
1187. import hust.cs.javacourse.search.query.Sort;
1188. import java.util.Comparator;
1189. import java.util.List;
1190. */\*\**
1191. \* <pre>
1192. \* SimpleSort是Sort接口的简单实现类，用于根据文档得分对命中结果进行排序。
1193. \* 该类提供了一个简单的得分计算方法，并根据得分对命中结果进行升序排序。
1194. \* </pre>
1195. \*/
1196. public class SimpleSorter implements Sort {
1197. */\*\**
1198. \* 对命中结果集合根据文档得分进行排序。
1199. \*
1200. \* @param hits ：命中结果集合
1201. \*/
1202. @Override
1203. public void sort(List<AbstractHit> hits) {
1204. *// 首先计算每个命中结果的得分*
1205. for (AbstractHit hit : hits) {
1206. *// 计算得分*
1207. score(hit);
1208. }
1209. *// 根据得分对命中结果进行降序排序*
1210. hits.sort((h1, h2) -> Double.compare(h2.getScore(), h1.getScore())); *// ✅ 降序*
1211. }
1212. */\*\**
1213. \* 计算命中文档的得分。
1214. \* 这里采用简单的得分计算方法：根据命中的单词数量计算得分。
1215. \*
1216. \* @param hit ：命中文档
1217. \* @return ：命中文档的得分
1218. \*/
1219. @Override
1220. public double score(AbstractHit hit) {
1221. *// 得分计算方法：命中的单词次数*
1222. double score = 0;
1223. for(AbstractPosting ap : hit.getTermPostingMapping().values())
1224. score += ap.getFreq();
1225. hit.setScore(score);
1226. return score;
1227. }
1228. }
1229. **IndexSearcher类**
1230. package hust.cs.javacourse.search.query.impl;
1231. import hust.cs.javacourse.search.index.AbstractIndex;
1232. import hust.cs.javacourse.search.index.AbstractPosting;
1233. import hust.cs.javacourse.search.index.AbstractPostingList;
1234. import hust.cs.javacourse.search.index.AbstractTerm;
1235. import hust.cs.javacourse.search.index.impl.PostingList;
1236. import hust.cs.javacourse.search.query.AbstractHit;
1237. import hust.cs.javacourse.search.query.AbstractIndexSearcher;
1238. import hust.cs.javacourse.search.query.Sort;
1239. import java.io.File;
1240. import java.util.ArrayList;
1241. import java.util.HashMap;
1242. import java.util.List;
1243. import java.util.Map;
1244. */\*\**
1245. \* <pre>
1246. \* IndexSearcher是AbstractIndexSearcher的具体子类，用于从索引文件中加载索引并执行搜索操作。
1247. \* 该类实现了单词搜索和双词搜索（支持AND/OR逻辑组合）的功能。
1248. \* </pre>
1249. \*/
1250. public class IndexSearcher extends AbstractIndexSearcher {
1251. */\*\**
1252. \* 从指定索引文件打开索引，加载到index对象里。
1253. \*
1254. \* @param indexFile 指定索引文件的路径
1255. \*/
1256. @Override
1257. public void open(String indexFile) {
1258. *// 使用index对象的load方法从指定文件加载索引*
1259. index.load(new File(indexFile));
1260. }
1261. */\*\**
1262. \* 根据单个检索词进行搜索。
1263. \* 该方法会查找包含指定检索词的所有文档，并返回搜索结果。
1264. \*
1265. \* @param queryTerm 检索词
1266. \* @param sorter 排序器，用于对搜索结果进行排序
1267. \* @return 命中结果数组，如果没有找到匹配的文档则返回空数组
1268. \*/
1269. @Override
1270. public AbstractHit[] search(AbstractTerm queryTerm, Sort sorter) {
1271. *// 在索引中搜索包含检索词的PostingList*
1272. AbstractPostingList postingList = index.search(queryTerm);
1273. List<AbstractHit> hits = new ArrayList<>();
1274. if (postingList != null) {
1275. *// 如果找到了包含检索词的PostingList*
1276. for (int i = 0; i < postingList.size(); i++) {
1277. *// 获取每个Posting*
1278. AbstractPosting posting = postingList.get(i);
1279. *// 创建Hit对象并添加到命中结果列表*
1280. AbstractHit hit = createHit(queryTerm, posting);
1281. hits.add(hit);
1282. }
1283. *// 对命中结果进行排序*
1284. sorter.sort(hits);
1285. }
1286. *// 将List转换为数组并返回*
1287. return hits.toArray(new AbstractHit[0]);
1288. }
1289. */\*\**
1290. \* 根据两个检索词进行搜索，支持AND/OR逻辑组合。
1291. \* AND逻辑：返回同时包含两个检索词的文档
1292. \* OR逻辑：返回包含任一检索词的文档
1293. \*
1294. \* @param queryTerm1 第1个检索词
1295. \* @param queryTerm2 第2个检索词
1296. \* @param sorter 排序器，用于对搜索结果进行排序
1297. \* @param combine 多个检索词的逻辑组合方式（AND/OR）
1298. \* @return 命中结果数组，如果没有找到匹配的文档则返回空数组
1299. \*/
1300. @Override
1301. public AbstractHit[] search(AbstractTerm queryTerm1, AbstractTerm queryTerm2, Sort sorter, LogicalCombination combine) {
1302. *// 在索引中搜索包含检索词的PostingList*
1303. AbstractPostingList postingList1 = index.search(queryTerm1);
1304. AbstractPostingList postingList2 = index.search(queryTerm2);
1305. List<AbstractHit> hits = new ArrayList<>();
1306. *// 用于存储已处理的文档ID，避免重复处理*
1307. List<Integer> processedDocIds = new ArrayList<>();
1308. *// 处理AND逻辑：查找同时包含两个检索词的文档*
1309. if (postingList1 != null && postingList2 != null) {
1310. processAndLogic(queryTerm1, queryTerm2, postingList1, postingList2, hits, processedDocIds);
1311. }
1312. *// 处理OR逻辑：添加只包含其中一个检索词的文档*
1313. if (combine == LogicalCombination.OR) {
1314. if (postingList1 != null) {
1315. processOrLogic(queryTerm1, postingList1, hits, processedDocIds);
1316. }
1317. if (postingList2 != null) {
1318. processOrLogic(queryTerm2, postingList2, hits, processedDocIds);
1319. }
1320. }
1321. *// 对命中结果进行排序*
1322. sorter.sort(hits);
1323. *// 将List转换为数组并返回*
1324. return hits.toArray(new AbstractHit[0]);
1325. }
1326. */\*\**
1327. \* 处理AND逻辑：查找同时包含两个检索词的文档
1328. \*
1329. \* @param queryTerm1 第1个检索词
1330. \* @param queryTerm2 第2个检索词
1331. \* @param postingList1 第1个检索词的PostingList
1332. \* @param postingList2 第2个检索词的PostingList
1333. \* @param hits 命中结果列表
1334. \* @param processedDocIds 已处理的文档ID列表
1335. \*/
1336. private void processAndLogic(AbstractTerm queryTerm1, AbstractTerm queryTerm2,
1337. AbstractPostingList postingList1, AbstractPostingList postingList2,
1338. List<AbstractHit> hits, List<Integer> processedDocIds) {
1339. for (int i = 0; i < postingList1.size(); i++) {
1340. AbstractPosting posting1 = postingList1.get(i);
1341. int docId = posting1.getDocId();
1342. *// 在第2个PostingList中查找相同的文档ID*
1343. int docIndex = postingList2.indexOf(docId);
1344. if (docIndex != -1) {
1345. *// 如果找到了相同的文档ID，说明两个检索词都出现在该文档中*
1346. AbstractPosting posting2 = postingList2.get(docIndex);
1347. *// 创建包含两个检索词的Hit对象*
1348. Map<AbstractTerm, AbstractPosting> termPostingMapping = new HashMap<>();
1349. termPostingMapping.put(queryTerm1, posting1);
1350. termPostingMapping.put(queryTerm2, posting2);
1351. String docPath = index.getDocName(docId);
1352. AbstractHit hit = new Hit(docId, docPath, termPostingMapping);
1353. hits.add(hit);
1354. *// 记录已处理的文档ID*
1355. processedDocIds.add(docId);
1356. }
1357. }
1358. }
1359. */\*\**
1360. \* 处理OR逻辑：添加只包含其中一个检索词的文档
1361. \*
1362. \* @param queryTerm 检索词
1363. \* @param postingList 检索词的PostingList
1364. \* @param hits 命中结果列表
1365. \* @param processedDocIds 已处理的文档ID列表
1366. \*/
1367. private void processOrLogic(AbstractTerm queryTerm, AbstractPostingList postingList,
1368. List<AbstractHit> hits, List<Integer> processedDocIds) {
1369. for (int i = 0; i < postingList.size(); i++) {
1370. AbstractPosting posting = postingList.get(i);
1371. int docId = posting.getDocId();
1372. *// 只处理尚未处理过的文档*
1373. if (!processedDocIds.contains(docId)) {
1374. *// 创建只包含一个检索词的Hit对象*
1375. AbstractHit hit = createHit(queryTerm, posting);
1376. hits.add(hit);
1377. *// 记录已处理的文档ID*
1378. processedDocIds.add(docId);
1379. }
1380. }
1381. }
1382. */\*\**
1383. \* 创建Hit对象
1384. \*
1385. \* @param queryTerm 检索词
1386. \* @param posting 检索词对应的Posting
1387. \* @return 创建的Hit对象
1388. \*/
1389. private AbstractHit createHit(AbstractTerm queryTerm, AbstractPosting posting) {
1390. *// 创建Term和Posting的映射*
1391. Map<AbstractTerm, AbstractPosting> termPostingMapping = new HashMap<>();
1392. termPostingMapping.put(queryTerm, posting);
1393. *// 获取文档ID和文档路径*
1394. int docId = posting.getDocId();
1395. String docPath = index.getDocName(docId);
1396. *// 创建并返回Hit对象*
1397. return new Hit(docId, docPath, termPostingMapping);
1398. }
1399. }
1400. **src/hust/cs/javacourse/search/run**
1401. **TestBuildIndex类**
1402. package hust.cs.javacourse.search.run;
1403. import hust.cs.javacourse.search.index.AbstractDocumentBuilder;
1404. import hust.cs.javacourse.search.index.AbstractIndex;
1405. import hust.cs.javacourse.search.index.AbstractIndexBuilder;
1406. import hust.cs.javacourse.search.index.impl.DocumentBuilder;
1407. import hust.cs.javacourse.search.index.impl.Index;
1408. import hust.cs.javacourse.search.index.impl.IndexBuilder;
1409. import hust.cs.javacourse.search.util.Config;
1410. import java.io.File;
1411. */\*\**
1412. \* 测试索引构建
1413. \*/
1414. public class TestBuildIndex {
1415. */\*\**
1416. \*  索引构建程序入口
1417. \* @param args : 命令行参数
1418. \*/
1419. public static void main(String[] args){
1420. AbstractDocumentBuilder documentBuilder = new DocumentBuilder();
1421. AbstractIndexBuilder indexBuilder = new IndexBuilder(documentBuilder);
1422. String rootDir = Config.DOC\_DIR;
1423. System.out.println("Start build index ...");
1424. AbstractIndex index = indexBuilder.buildIndex(rootDir);
1425. index.optimize();
1426. System.out.println(index); *//控制台打印 index 的内容*
1427. *//测试保存到文件*
1428. String indexFile = Config.INDEX\_DIR + "index.dat";
1429. index.save(new File(indexFile)); *//索引保存到文件*
1430. *//测试从文件读取*
1431. AbstractIndex index2 = new Index(); *//创建一个空的 index*
1432. index2.load(new File(indexFile)); *//从文件加载对象的内容*
1433. System.out.println("\n-------------------\n");
1434. System.out.println(index2); *//控制台打印 index2 的内容*
1435. }
1436. }
1437. **TestSearchIndex类**
1438. package hust.cs.javacourse.search.run;
1439. import hust.cs.javacourse.search.index.impl.Term;
1440. import hust.cs.javacourse.search.parse.AbstractTermTupleStream;
1441. import hust.cs.javacourse.search.query.AbstractHit;
1442. import hust.cs.javacourse.search.query.AbstractIndexSearcher;
1443. import hust.cs.javacourse.search.query.Sort;
1444. import hust.cs.javacourse.search.query.impl.IndexSearcher;
1445. import hust.cs.javacourse.search.query.impl.SimpleSorter;
1446. import hust.cs.javacourse.search.util.Config;
1447. import javax.swing.plaf.nimbus.AbstractRegionPainter;
1448. */\*\**
1449. \* 测试搜索
1450. \*/
1451. public class TestSearchIndex {
1452. */\*\**
1453. \* 搜索程序入口
1454. \*
1455. \* @param args ：命令行参数
1456. \*/
1457. public static void main(String[] args) {
1458. Sort simpleSorter = new SimpleSorter();
1459. String indexFile = Config.INDEX\_DIR + "index.dat";
1460. AbstractIndexSearcher searcher = new IndexSearcher();
1461. searcher.open(indexFile);
1462. System.out.println("===== [1] 单词查询：Term(\"activity\") =====");
1463. AbstractHit[] hits1 = searcher.search(new Term("activity"), simpleSorter);
1464. for (AbstractHit hit : hits1) {
1465. System.out.println(hit);
1466. }
1467. System.out.println("\n===== [2] 多词查询（AND）：\"activity\" AND \"because\" =====");
1468. AbstractHit[] hits2 = searcher.search(new Term("activity"), new Term("because"), simpleSorter, AbstractIndexSearcher.LogicalCombination.AND);
1469. for (AbstractHit hit : hits2) {
1470. System.out.println(hit);
1471. }
1472. System.out.println("\n===== [3] 多词查询（OR）：\"frozen\" OR \"medical\" =====");
1473. AbstractHit[] hits3 = searcher.search(new Term("frozen"), new Term("medical"), simpleSorter, AbstractIndexSearcher.LogicalCombination.OR);
1474. for (AbstractHit hit : hits3) {
1475. System.out.println(hit);
1476. }
1477. }
1478. }
1479. **TestSearchIndexInteractive类**
1480. package hust.cs.javacourse.search.run;
1481. import hust.cs.javacourse.search.index.impl.Term;
1482. import hust.cs.javacourse.search.parse.AbstractTermTupleStream;
1483. import hust.cs.javacourse.search.query.AbstractHit;
1484. import hust.cs.javacourse.search.query.AbstractIndexSearcher;
1485. import hust.cs.javacourse.search.query.Sort;
1486. import hust.cs.javacourse.search.query.impl.IndexSearcher;
1487. import hust.cs.javacourse.search.query.impl.SimpleSorter;
1488. import hust.cs.javacourse.search.util.Config;
1489. import javax.swing.plaf.nimbus.AbstractRegionPainter;
1490. import java.util.Scanner;
1491. */\*\**
1492. \* 交互式搜索测试程序
1493. \*/
1494. public class TestSearchIndexInteractive {
1495. public static void main(String[] args) {
1496. Sort sorter = new SimpleSorter();
1497. String indexFile = Config.INDEX\_DIR + "index.dat";
1498. AbstractIndexSearcher searcher = new IndexSearcher();
1499. searcher.open(indexFile);
1500. Scanner scanner = new Scanner(System.in);
1501. System.out.println("====== 欢迎使用简易搜索引擎 ======");
1502. System.out.println("索引文件已加载：" + indexFile);
1503. while (true) {
1504. System.out.println("\n请选择查询类型：");
1505. System.out.println("1 - 单词查询");
1506. System.out.println("2 - AND 查询");
1507. System.out.println("3 - OR 查询");
1508. System.out.println("0 - 退出");
1509. System.out.print("请输入选项：");
1510. String choice = scanner.nextLine().trim();
1511. if (choice.equals("0")) {
1512. System.out.println("感谢使用，再见！");
1513. break;
1514. }
1515. switch (choice) {
1516. case "1":
1517. System.out.print("请输入查询单词：");
1518. String word = scanner.nextLine().trim().toLowerCase();
1519. AbstractHit[] hits1 = searcher.search(new Term(word), sorter);
1520. printHits(hits1);
1521. break;
1522. case "2":
1523. System.out.print("请输入第一个单词：");
1524. String andWord1 = scanner.nextLine().trim().toLowerCase();
1525. System.out.print("请输入第二个单词：");
1526. String andWord2 = scanner.nextLine().trim().toLowerCase();
1527. AbstractHit[] hits2 = searcher.search(new Term(andWord1), new Term(andWord2), sorter,
1528. AbstractIndexSearcher.LogicalCombination.AND);
1529. printHits(hits2);
1530. break;
1531. case "3":
1532. System.out.print("请输入第一个单词：");
1533. String orWord1 = scanner.nextLine().trim().toLowerCase();
1534. System.out.print("请输入第二个单词：");
1535. String orWord2 = scanner.nextLine().trim().toLowerCase();
1536. AbstractHit[] hits3 = searcher.search(new Term(orWord1), new Term(orWord2), sorter,
1537. AbstractIndexSearcher.LogicalCombination.OR);
1538. printHits(hits3);
1539. break;
1540. default:
1541. System.out.println("无效选项，请重新输入。");
1542. break;
1543. }
1544. }
1545. scanner.close();
1546. }
1547. private static void printHits(AbstractHit[] hits) {
1548. if (hits == null || hits.length == 0) {
1549. System.out.println("未找到匹配结果。");
1550. return;
1551. }
1552. System.out.println("共找到 " + hits.length + " 个匹配结果：");
1553. for (AbstractHit hit : hits) {
1554. System.out.println(hit);
1555. }
1556. }
1557. }
1558. **src/hust/cs/javacourse/search/util**
1559. **config类**
1560. package hust.cs.javacourse.search.util;
1561. */\*\**
1562. \* <pre>
1563. \*
1564. \* 保存搜索引擎的配置信息，例如：
1565. \*      索引文件所在目录
1566. \*      要建立索引的文本文件所在目录
1567. \*      构建索引时是否忽略单词大小写
1568. \*      分词所需要的正则表达式
1569. \*      基于正则表达式的三元组过滤器所需的正则表达式
1570. \*      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
1571. \*      ...
1572. \* </pre>
1573. \*
1574. \*/
1575. public class Config {
1576. */\*\**
1577. \* Java工程HOME目录, System.getProperty("user.dir")返回当前JAVA工程目录
1578. \*/
1579. public static String PROJECT\_HOME\_DIR= System.getProperty("user.dir");
1580. */\*\**
1581. \* <pre>
1582. \* 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.
1583. \* 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.
1584. \* </pre>
1585. \*/
1586. public static String INDEX\_DIR = PROJECT\_HOME\_DIR + "/index/";
1587. */\*\**
1588. \* <pre>
1589. \*  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
1590. \*  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.
1591. \* </pre>
1592. \*
1593. \*/
1594. public static String DOC\_DIR  = PROJECT\_HOME\_DIR + "/text/";
1595. */\*\**
1596. \* 构建索引和检索时是否忽略单词大小写
1597. \*/
1598. public static boolean IGNORE\_CASE = true;
1599. */\*\**
1600. \* <pre>
1601. \* 将字符串切分成单词时所需的正则表达式.
1602. \* 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
1603. \* </pre>
1604. \*
1605. \*/
1606. public static String STRING\_SPLITTER\_REGEX = "[,|，|;|；|.|。|?|？|:|：|!|！|、|\\s]+";
1607. */\*\**
1608. \* <pre>
1609. \* 单词过滤的正则表达式.
1610. \* 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
1611. \* </pre>
1612. \*/
1613. public static String TERM\_FILTER\_PATTERN = "[a-zA-Z]+";
1614. */\*\**
1615. \* <pre>
1616. \* 基于单词的最小长度过滤单词.
1617. \* 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
1618. \* </pre>
1619. \*/
1620. public static int TERM\_FILTER\_MINLENGTH = 3;
1621. */\*\**
1622. \* <pre>
1623. \* 基于单词的最小长度过滤单词.
1624. \* 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
1625. \* </pre>
1626. \*/
1627. public static int TERM\_FILTER\_MAXLENGTH = 20;
1628. }
1629. **FileUtil类**
1630. package hust.cs.javacourse.search.util;
1631. import java.io.\*;
1632. import java.util.ArrayList;
1633. import java.util.List;
1634. */\*\**
1635. \* 文件操作的工具类
1636. \*/
1637. public class FileUtil {
1638. private FileUtil(){}
1639. */\*\**
1640. \* 一次读取指定文本文件的所有内容
1641. \* @param filePath ：指定文本文件的绝对路径
1642. \* @return  ： 文本文件的内容
1643. \*/
1644. public static String read(String filePath){
1645. String s = null;
1646. BufferedReader reader = null;
1647. try{
1648. StringBuffer buf = new StringBuffer();
1649. reader = new BufferedReader(new InputStreamReader(new FileInputStream(new File(filePath))));
1650. while( (s = reader.readLine()) != null){
1651. 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*
1652. }
1653. s = buf.toString().trim(); *//去掉最后一个多的换行符*
1654. } catch (FileNotFoundException e) {
1655. e.printStackTrace();
1656. } catch (IOException e) {
1657. e.printStackTrace();
1658. }
1659. finally {
1660. if(reader != null) {
1661. try {
1662. reader.close();
1663. } catch (IOException e) {
1664. e.printStackTrace();
1665. }
1666. }
1667. }
1668. return s;
1669. }
1670. */\*\**
1671. \* 将字符串写入到指定的文本文件
1672. \* @param content  ：  写入的内容
1673. \* @param filePath ： 指定的文本文件路径
1674. \*/
1675. public static void write(String content, String filePath){
1676. PrintWriter writer = null;
1677. try{
1678. writer = new PrintWriter(new BufferedWriter(new OutputStreamWriter(new FileOutputStream(new File(filePath)))));
1679. writer.print(content);
1680. } catch (FileNotFoundException e) {
1681. e.printStackTrace();
1682. }
1683. finally {
1684. if(writer != null){
1685. writer.close();
1686. }
1687. }
1688. }
1689. */\*\**
1690. \* 列出指定目录下所有文件的绝对路径(不递归)
1691. \* @param dirPath : 指定目录
1692. \* @return ： 指定目录下所有文件的绝对路径的列表
1693. \*/
1694. public static List<String> list(String dirPath){
1695. List<String> filePaths = new ArrayList<String>();
1696. try{
1697. File dir = new File(dirPath);
1698. if(dir.isDirectory()){
1699. File[] files = dir.listFiles();
1700. for(File f: files){
1701. if(f.isFile()) {
1702. filePaths.add(f.getCanonicalPath()); *//File类的getCanonicalPath方法返回绝对路径*
1703. }
1704. }
1705. }
1706. }
1707. catch (IOException e) {
1708. e.printStackTrace();
1709. }
1710. return filePaths;
1711. }
1712. */\*\**
1713. \* 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)
1714. \* @param dirPath : 指定目录
1715. \* @param suffix  ：指定后缀名, 如 .txt
1716. \* @return : 所有匹配指定后缀名的文件绝对路径列表
1717. \*/
1718. public static List<String> list(String dirPath, String suffix){
1719. List<String> filePaths = new ArrayList<String>();
1720. try{
1721. File dir = new File(dirPath);
1722. if(dir.isDirectory()){
1723. File[] files = dir.listFiles();
1724. for(File f: files){
1725. if(f.isFile()) {
1726. String path = f.getCanonicalPath(); *//File类的getCanonicalPath方法返回绝对路径*
1727. if(path.toLowerCase().endsWith(suffix.toLowerCase())) {
1728. filePaths.add(path);
1729. }
1730. }
1731. }
1732. }
1733. }
1734. catch (IOException e) {
1735. e.printStackTrace();
1736. }
1737. return filePaths;
1738. }
1739. public static void main(String[] args){
1740. *//从文件读*
1741. String fileName = Config.DOC\_DIR + "1.txt";
1742. String s = FileUtil.read(fileName);
1743. System.out.println(s);
1744. *//写入到另一个文件*
1745. String fileName2 = Config.DOC\_DIR + "2.txt";
1746. FileUtil.write(s,fileName2);
1747. *//列出指定目录下所有文件*
1748. String dirName = Config.PROJECT\_HOME\_DIR + "/javadoc";
1749. List<String> fileNames = FileUtil.list(dirName);
1750. for(String p: fileNames){
1751. System.out.println(p);
1752. }
1753. System.out.println("----------------");
1754. *//列出指定目录下所有.html文件*
1755. String dirName1 = Config.PROJECT\_HOME\_DIR + "/javadoc";
1756. List<String> fileNames1 = FileUtil.list(dirName,".html");
1757. for(String p: fileNames1){
1758. System.out.println(p);
1759. }
1760. }
1761. }
1762. **StopWords类**
1763. package hust.cs.javacourse.search.util;
1764. /\*\*
1765. \* 停用词表类
1766. \*/
1767. public class StopWords {
1768. /\*\*
1769. \* 停用词字符串数组
1770. \*/
1771. public static String[] STOP\_WORDS =
1772. {"a", "about", "above", "above", "across", "after", "afterwards",
1773. "again", "against", "all", "almost", "alone", "along", "already", "also","although",
1774. "always","am","among", "amongst", "amoungst", "amount",  "an", "and", "another", "any",
1775. "anyhow","anyone","anything","anyway", "anywhere", "are", "around", "as",  "at", "back",
1776. "be","became", "because","become","becomes", "becoming", "been", "before", "beforehand",
1777. "behind", "being", "below", "beside", "besides", "between", "beyond", "bill", "both", "bottom",
1778. "but", "by", "call", "can", "cannot", "cant", "co", "con", "could", "couldnt", "cry", "de",
1779. "describe", "detail", "do", "done", "down", "due", "during", "each", "eg", "eight", "either",
1780. "eleven","else", "elsewhere", "empty", "enough", "etc", "even", "ever", "every", "everyone",
1781. "everything", "everywhere", "except", "few", "fifteen", "fify", "fill", "find", "fire", "first",
1782. "five", "for", "former", "formerly", "forty", "found", "four", "from", "front", "full", "further",
1783. "get", "give", "go", "had", "has", "hasnt", "have", "he", "hence", "her", "here", "hereafter",
1784. "hereby", "herein", "hereupon", "hers", "herself", "him", "himself", "his", "how", "however",
1785. "hundred", "i","ie", "if", "in", "inc", "indeed", "interest", "into", "is", "it", "its", "itself",
1786. "keep", "last", "latter", "latterly", "least", "less", "ltd", "made", "many", "may", "me",
1787. "meanwhile", "might", "mill", "mine", "more", "moreover", "most", "mostly", "move", "much", "must",
1788. "my", "myself", "name", "namely", "neither", "never", "nevertheless", "next", "nine", "no", "nobody",
1789. "none", "noone", "nor", "not", "nothing", "now", "nowhere", "of", "off", "often", "on", "once", "one",
1790. "only", "onto", "or", "other", "others", "otherwise", "our", "ours", "ourselves", "out", "over", "own",
1791. "part", "per", "perhaps", "please", "put", "rather", "re", "same", "see", "seem", "seemed", "seeming",
1792. "seems", "serious", "several", "she", "should", "show", "side", "since", "sincere", "six", "sixty",
1793. "so", "some", "somehow", "someone", "something", "sometime", "sometimes", "somewhere", "still", "such",
1794. "system", "take", "ten", "than", "that", "the", "their", "them", "themselves", "then", "thence", "there",
1795. "thereafter", "thereby", "therefore", "therein", "thereupon", "these", "they", "thickv", "thin", "third",
1796. "this", "those", "though", "three", "through", "throughout", "thru", "thus", "to", "together", "too", "top",
1797. "toward", "towards", "twelve", "twenty", "two", "un", "under", "until", "up", "upon", "us", "very",
1798. "via", "was", "we", "well", "were", "what", "whatever", "when", "whence", "whenever", "where", "whereafter",
1799. "whereas", "whereby", "wherein", "whereupon", "wherever", "whether", "which", "while", "whither", "who",
1800. "whoever", "whole", "whom", "whose", "why", "will", "with", "within", "without", "would", "yet", "you",
1801. "your", "yours", "yourself", "yourselves","s","t"};
1802. }
1803. **StringSplitter类**
1804. package hust.cs.javacourse.search.util;
1805. import java.util.ArrayList;
1806. import java.util.List;
1807. import java.util.regex.Matcher;
1808. import java.util.regex.Pattern;
1809. */\*\**
1810. \* 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
1811. \*/
1812. public class StringSplitter {
1813. public StringSplitter(){}
1814. private  String splitRegex = null;
1815. private  Pattern pattern = null;
1816. private  Matcher match = null;
1817. */\*\**
1818. \* 设置分词用的正则表达式
1819. \* @param regex：分词用的正则表达式
1820. \*/
1821. public  void setSplitRegex(String regex){
1822. splitRegex = regex;
1823. pattern = Pattern.compile(splitRegex);
1824. }
1825. */\*\**
1826. \* 将字符串分割成单词列表
1827. \* @param input： 输入字符串
1828. \* @return ： 分词得到的单词列表
1829. \*/
1830. public  List<String> splitByRegex(String input){
1831. List<String> list = new ArrayList<String>();
1832. match = pattern.matcher(input);
1833. String part = null;
1834. int lastEnd = 0;
1835. while(match.find()){
1836. int start = match.start(0);
1837. int end = match.end(0);
1838. part = input.substring(lastEnd, match.start(0));
1839. lastEnd = match.end(0);
1840. if( (part != null && part.equals("")) || part == null) {
1841. continue;
1842. }
1843. list.add(part);
1844. }
1845. *//取得最后一部分*
1846. if(lastEnd < input.length()){
1847. part = input.substring(lastEnd);
1848. if(part != null && !part.equals(""))
1849. list.add(part);
1850. }
1851. return list;
1852. }
1853. public static void main(String[] args){
1854. StringSplitter splitter = new StringSplitter();
1855. String regex1TestInput = "key1,， ,Key2;Key3，:Key4;；;Key5；？?Key6，!Key7；Key8   key9\nkey10.。？key11";
1856. splitter.setSplitRegex(Config.STRING\_SPLITTER\_REGEX);
1857. List<String> parts = splitter.splitByRegex(regex1TestInput);
1858. for(int i = 0; i < parts.size(); i++){
1859. System.out.print(parts.get(i) + " ");
1860. }
1861. }
1862. }