Java实验一帮助文档

华中科技大学计算机学院
Java语言程序设计课程组

内容

- 搜索引擎的倒排索引数据结构
- 预定义的抽象类及接口API说明
- 本实验所涉及的JDK Java API说明

倒排索引（inverted index）

- 首先思考一个问题
- Google号称80亿网页，Baidu也有10亿网页，数量可谓巨大，但是当我们输入一个查询时，返回时间往往不到1秒，为什么这么快？奥秘在哪里？
- 奥秘在这些数量巨大的网页的组织方式
- 这些网页（或者说文档）是以倒排索引的方式组织的

倒排索引（inverted index）

- 索引单位及分词
- 在讨论倒排索引前，首先需要搞清楚文本文档是以什么为单位写入索引的？即索引单位是什么？
- 无论是中文文本文档还是其他语言（如英语）的文本文档，文本的内容都是由一个个的单词组成，在搜索引擎领域里，我们称这些单词为“term”。
- 将一个文本文档切分成terms的过程称为“分词”（Tokenization）
- 英文文档的分词比较简单：以空格+标点符号作为分隔符
- 例如，对于这样一篇英文文档

  In June, the dog likes to chase the cat in the barn.
- 可以将其切分成以下terms（不考虑重复的term）

in	June	the	dog	likes	to	chase	cat	barn

倒排索引（inverted index）

- 索引单位及分词
- 中文文档的分词则复杂的多，因为中文单词不是以空格作为分隔符
- 例如对于短语“中华人民共和国”，有多种分词的方法

中华	人民	共和国

中华人民	共和国

…

- 本实验只考虑英文文档

倒排索引（inverted index）

- 分词后的处理
- 即使是英文文档，分词后还需要后续的处理步骤，典型的步骤有
- 大小写转换（Case folding）
  - 将terms全部转成小写
- 基于正则表达式过滤不要的单词，例如过滤掉所有数字
- 过滤掉过短或过长的单词
- 过滤停用词（Stop word elimination）
  - 停用词是指指那些出现频率高但是无重要意义，通常不会作为查询词出现的词、消除停用词可以大大减少索引所占空间。
  - 英文停用词：a, an, and, are, as, at, be, by, for, from，...
  - 中文停用词：“的”、“地”、“得”、“都”、“是”，“我们”，...
  - 消除方法：定义一个停用词表，用查表的方法消除

倒排索引（inverted index）

- 英文停用词表定义示例：本实验采用这个停用词表

```
private String[] additinalStopWords = {"a", "about", "above", "above", "across", "after", "afterwards", "again", "against", "all",
"almost", "alone", "along", "already", "also", "although", "always", "am", "among", "amongst", "amongst", "amount",  "an",
"and", "another", "any", "anyhow", "anyone", anything",“ anyway", "anywhere", "are", "around", "as",  "at", "back",
"be", "became", "because", "become", "becomes", "becoming", "been", "before", "beforehand", "behind", "being", "below",
"beside", "besides", "between", "beyond", "bill", "both", "bottom", "but", "by", "call", "can", "cannot", "cant", "co", "con", "could",
"couldn't", "cry", "de", "describe", "detail", "do", "done", "down", "due", "during", "each", "e.g.", "eight", "either",
"eleven", "else", "elsewhere", "empty", "enough", "etc", "even", "ever", "every", "everyone", "everything", "everywhere",
"except", "few", "fifteen", "fifty", "fill", "find", "fire", "first", "five", "for", "former", "formerly", "forty", "found", "four", "from", "front",
"full", "further", "get", "give", "go", "had", "has", "hasn't", "have", "he", "hence", "her", "here", "hereafter",
"hereby", "herein", "hereupon", "hers", "herself", "him", "himself", "his", "how", "however", "hundred", "i", "i.e.", "if", "in", "inc",
"indeed", "interest", "into", "is", "it", "its", "itself", "keep", "last", "latter", "latterly", "least", "less", "ltd", "made", "many", "may",
"me", "meanwhile", "might", "mill", "mine", "more", "moreover", "most", "mostly", "move", "much", "must",
"my", "myself", "name", "namely", "neither", "never", "nevertheless", "next", "nine", "no", "nobody", "none", "no one", "nor",
"not", "nothing", "now", "nowhere", "of", "off", "often", "on", "once", "one", "only", "onto", "or", "other", "others", "otherwise",
"our", "ours", "ourselves", "out", "over", "own", "part", "per", "perhaps", "please", "put", "rather", "re", "same", "see", "seem",
"seemed", "seeming", "seems", "serious", "several", "she", "should", "show", "side", "since", "sincere", "six", "sixty",
"so", "some", "somehow", "someone", "something", "sometime", "sometimes", "somewhere", "still", "such", "system", "take",
"ten", "than", "that", "the", "their", "them", "themselves", "then", "thence", "there", "thereafter", "thereby", "therefore", "therein",
"thereupon", "these", "they", "thick", "thin", "third", "this", "those", "though", "three", "through", "throughout", "thru", "thus", "to",
"together", "too", "top", "toward", "towards", "twelve", "twenty", "two", "un", "under", "until", "up", "upon", "us", "very", "via",
"was", "we", "well", "were", "what", "whatever", "when", "whence", "whenever", "where", "where after", "whereas", "whereby",
"wherein", "whereupon", "wherever", "whether", "which", "while", "whither", "who", "whoever", "whole", "whom", "whose",
"why", "will", "with", "within", "without", "would", "yet", "you", "your", "yours", "yourself", "yourselves"};
```

倒排索引（inverted index）

- 字典（Dictionary）
- 通俗地讲，字典就是对文档集合里每个文档分词后得到的单词集合取并集
- 对于一个文档集合$C=\{d_1, d_2, \ldots, d_n\}$，$C$包含$n$个文档$d_1, d_2, \ldots, d_n$。
- 对文档d_i（$i=1, \ldots, n$）进行分词并进行分词后的处理， 得到的terms集合记为$Terms_{d_i}$
- 则文档集合C对应的字典D_C=每个文档分词得到的terms集合的并集，即

$$D_C = Terms_{d_1} \cup Terms_{d_2} \cup \ldots \cup Terms_{d_n}$$

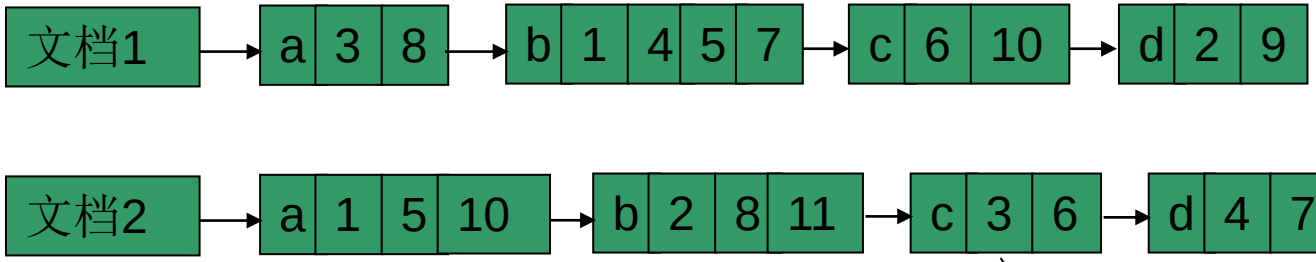
- 当文档集合的字典构建好后，就可以对分词后的每个文档构建倒排索引

倒排索引（inverted index）

- 什么是索引：为方便查找，描述原文件信息组织的文件
- 为了理解倒排索引的好处，首先看看前向索引（Forward Index）
- 为了方便描述，假设文档集合仅仅包含2个文档，分别为

  文档1：b d a b b c b a d c 文档2：a b c d a c d b d a b

  例子中为简化起见，用每个字母代表文档中出现的term
  文档1和2的字典为{a，b，c，d}
- 前向索引：将每篇文档表示成DocID及其文本内容组成的向量模式 。
- 前向索引是DocID->term的链表结构（从文档到term）



每个term后面的数字代表该term在文档中出现的位置。如term c在文档2中出现在位置3和6

倒排索引（inverted index）

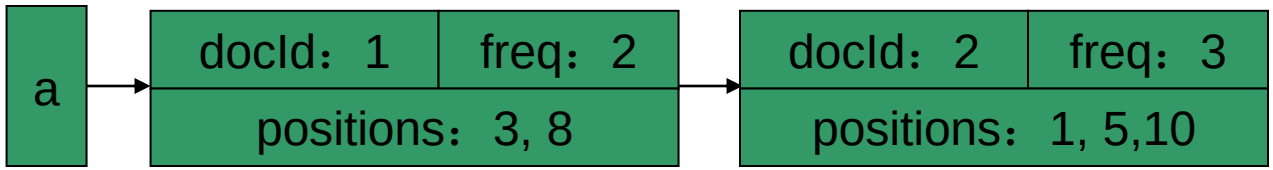
- 前向索引的问题
- 当用户输入一个查询词，如c时，需要一篇篇地扫描每个文档，看文档是否包含term c。
- 由于每个文档出现的term可以按字典序排序，因此可以利用二分查找法查找每个文档里是否出现term c，这样可以加快查找速度。
- 但是，由于前向索引必须一篇篇地扫描每个文档，当文档数量巨大时（如Google号称有80亿网页），前向索引的查找就非常的费时费力。
- 能不能有一种方法能够直接从查询词定位到文档？答案是当然有了，这就是倒排索引(inverted index)。
- 倒排索引思想：从term到文档
  - 每个文档都可以用一系列term来表示（通过分词）
  - 如果按term建立到文档的索引便可以根据term快速地检索到相关文档

倒排索引（inverted index）

- 倒排索引结构（本实验采用这种结构）
- 还是以文档1和2为例说明倒排索引的结构

  文档1：b d a b b c b a d c　文档2：a b c d a c d b d a b

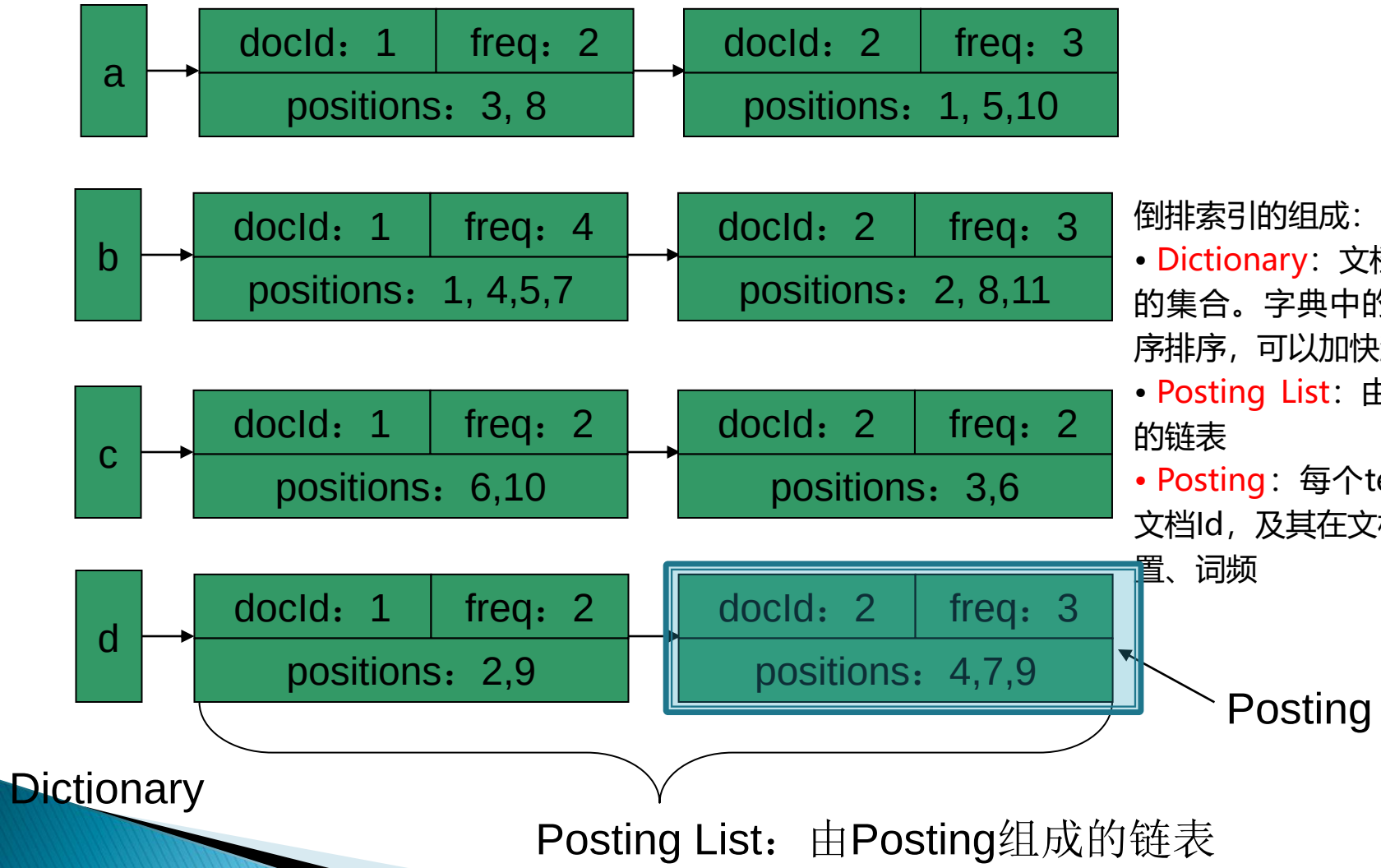
  文档1和2的字典为{a，b，c，d}



- 倒排索引是term->DocID的链表结构，和前向索引相反。因此叫倒排
- 上面的示例是term a的倒排索引，它描述了这样的信息
  - term a在文档1（docId：1）中出现，出现位置为3和8（positions：3,8）
  - term a在文档2中出现，出现位置为1,5,10
  - 还隐含了一个重要信息：term a在文档1中出现了2次（freq：2），在文档2中出现了3次（freq：3）
  - 一个term在某文档中出现的次数叫词频（term frequency，tf）。词频对于对搜索结果集合里的文档排序非常有用

倒排索引（inverted index）

- 因此，文档1和2完整的倒排索引为：



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

倒排索引（inverted index）

- 基于倒排索引的查询示例
- 还是假设用户输入查询词 c
- 对于前向索引，必须一篇篇扫描文档，看是否包含term c
  - 假设一共有10万篇文档，必须扫描10万次
  - 对于Google的80亿网页呢？无法想象
- 对于倒排索引，由于字典中的term是按字典序排好了序（假设字典里共包含N个term），那么利用二分查找法，我们最多需要$\log_2 N$次就可以定位到term c
  - 一旦在字典里找到term c，通过c的Postings List我们马上就知道有哪些文档包含了term c
  - 效率比前向索引高得多
  - 但你可能马上会有疑问：**那字典里包含的term个数N到底会多大？N要是非常大呢？Heaps定理可以回答你的问题。**

倒排索引（inverted index）

- 倒排索引组成：由Dictionary和Postings List组成
- Dictionary
  - Heaps定理：具有n个term的文本集合，其包含的词汇量=O (n^{β}), β: 0.4~0.6
  - 原因：任何一种语言，其基本词汇的数量是有限的
  - 牛津英语词典：收录了30万个词汇。将所有英文网页构建倒排索引，其字典大小也就是几十万个term。几十万个词汇以当今计算机的处理能力来说，完全可以装入内存。
  - 在内存里将30万个词汇排好序，再利用二分查找算法找到一个term，速度会非常快
  - 因此倒排索引的字典是全部被装载到内存中，这样大大加快了字典的查找速度
- Postings List
  - 可能会非常大。例如出现“JAVA”这个term的网页可能要以百万计，因此“JAVA”对应的Postings List可能非常长
  - Postings List主要问题是空间问题，往往要采用压缩技术
  - Postings List往往是单独放在一个或多个文件里，而且保存在硬盘上。
- 因此一个实际的搜索引擎系统是不可能一次把所有PostingList加载到内存里的。本实验的文档集合很小，因此可以全部加载到内存里。因此叫基于内存的搜索引擎

倒排索引（inverted index)

- 倒排索引的例子
- 实际上倒排索引在很多书上都可以看到，下图为某英文教材最后列出的倒排索引

E-optimal experiment design, 387
eccentricity, 461
e_i (ith unit vector), 33
eigenvalue
    decomposition, 646
    generalized, 647
    interlacing theorem, 122
    maximum, 82, 203
    optimization, 203
    spread, 203
    sum of k largest, 118
electronic device sizing, 2
elementary symmetric functions, 122
elimination
    banded matrix, 675
    block, 546
    constraints, 132
    equality constraints, 523, 542
    variables, 672
ellipsoid, 29, 39, 635
    statistical, 351
Euclidean
    ball, 29
    distance
        matrix, 65
        problems, 405
    norm, 633
    projection via pseudo-inverse, 649
exact line search, 464
exchange rate, 184
expanded set, 61
experiment design, 384
    A-optimal, 387
    D-optimal, 387
    dual, 276
    E-optimal, 387
explanatory variables, 353
explicit constraint, 134
exponential, 71
    distribution, 105
    matrix, 110

倒排索引（inverted index)

- 倒排索引的例子
- 实际上倒排索引在很多书上都可以看到，下图为某英文教材最后列出的倒排索引

E-optimal experiment design, 387
eccentricity, 461
e_i (ith unit vector), 33
eigenvalue
    decomposition, 646
    generalized, 647
    interlacing theorem, 122
    maximum, 82, 203
    optimization, 203
    spread, 203
    sum of k largest, 118
electronic device sizing, 2
elementary symmetric functions, 122
elimination
    banded matrix, 675
    block, 546
    constraints, 132
    equality constraints, 523, 542
    variables, 672
ellipsoid, 29, 39, 635
    statistical, 351
Euclidean
    ball, 29
    distance
        matrix, 65
        problems, 405
    norm, 633
    projection via pseudo-inverse, 649
exact line search, 464
exchange rate, 184
expanded set, 61
experiment design, 384
    A-optimal, 387
    D-optimal, 387
    dual, 276
    E-optimal, 387
explanatory variables, 353
explicit constraint, 134
exponential, 71
    distribution, 105
    matrix, 110

内容

- 搜索引擎的倒排索引数据结构
- 预定义的抽象类及接口API说明
- 本实验所涉及的JDK Java API说明

预定义的抽象类及接口说明

- IDEA Java工程里定义了如下package, 这些包的作用如下图所示

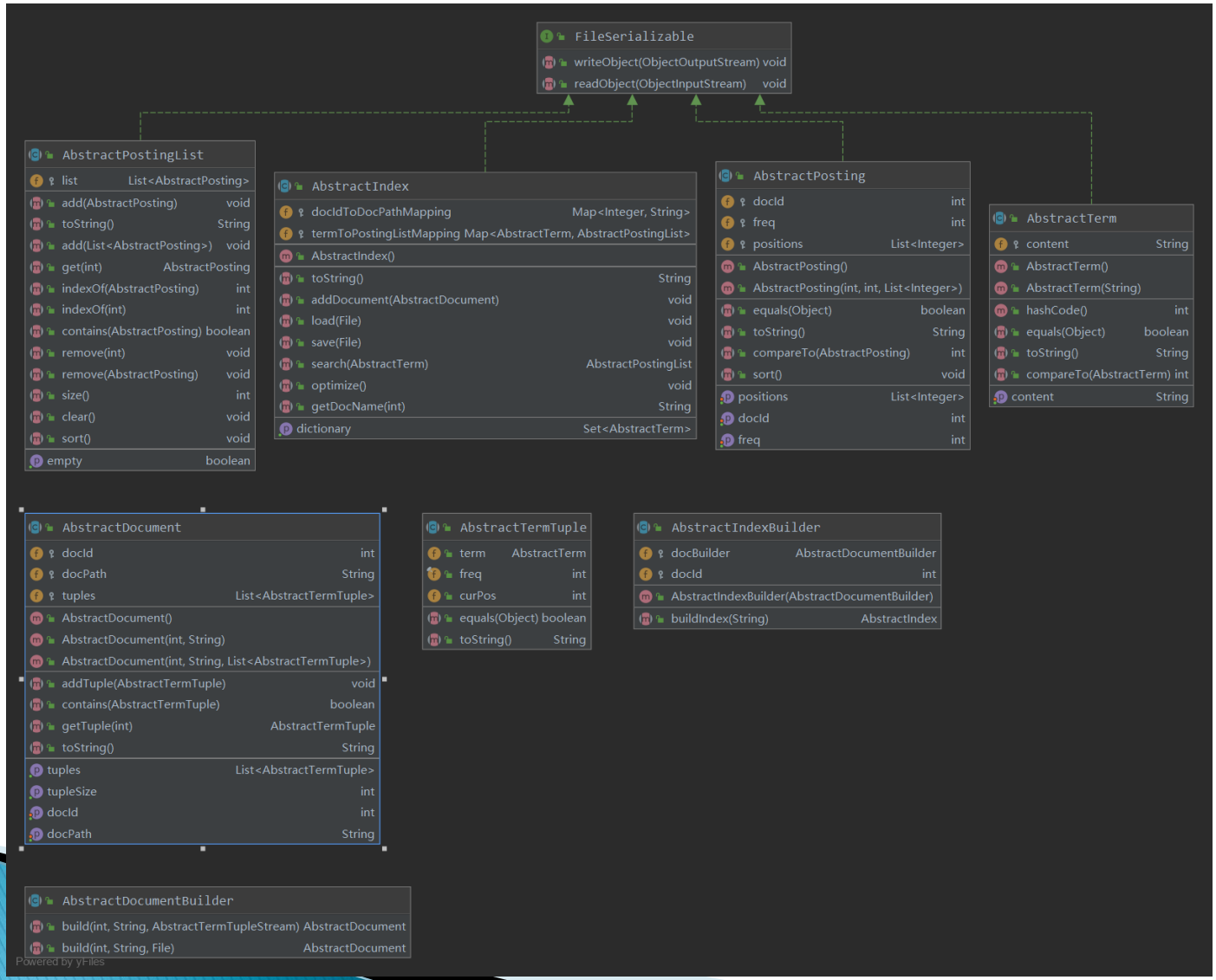
程序包

程序包	说明
hust.cs.javacourse.search.index	hust.cs.javacourse.search.index包里定义了和倒排索引数据结构相关的抽象类，以及和索引构建相关的抽象类和接口.
hust.cs.javacourse.search.parse	hust.cs.javacourse.search.parse包里定义了文档解析、分词，单词过滤有关的抽象类.学生需要实现这些抽象类的具体子类
hust.cs.javacourse.search.query	hust.cs.javacourse.search.query包里定义了和搜索有关的抽象类和接口.学生需要实现这些抽象类和接口的具体子类.
hust.cs.javacourse.search.run	最后的程序运行入口类放在hust.cs.javacourse.search.run里
hust.cs.javacourse.search.util	hust.cs.javacourse.search.util包里实现了一些工具类，学生可以参考和直接使用.具体包括： Config：索引构建和搜索的配置信息 StopWords：停用词表 StringSplitter：将字符串分割成一个个的单词 FileUtil：读写文本文件

- 另外工程里还定义了下面三个空的包
  - hust.cs.javacourse.search.index.impl：对hust.cs.javacourse.search.index包里定义的抽象类和接口的具体实现建议放在这个包里。impl(implementation)
  - hust.cs.javacourse.search.parse.impl：对hust.cs.javacourse.search.parse包里定义的抽象类和接口的具体实现建议放在这个包里。
  - hust.cs.javacourse.search.query.impl：对hust.cs.javacourse.search.query包里定义的抽象类和接口的具体实现建议放在这个包里。

预定义的抽象类及接口说明

- **package hust.cs.javacourse.search.index**：包里定义的抽象类和接口的UML模型见下图：



预定义的抽象类及接口说明

- **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, 请参见PPT第12页
  - **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。

预定义的抽象类及接口说明

- **package hust.cs.javacourse.search.index**
  - 因此AbstractTerm、AbstractPosting、AbstractPostingList、AbstractIndex这四个抽象类就规定了倒排索引所需要的数据结构，
  - 它们之间的关系为：
    - **AbstractPostingList里包含了多个AbstractPosting**
    - **AbstractIndex包含了多个<AbstractTerm，AbstractPostingList>这样的key-value对，这些key-value对维护了单词到PostingList的链接关系**
    - 另外这四个类都要求实现FileSerializable接口，这样可以很方便的将构建好的索引序列化到文件，或者直接从文件里将构建好的索引反序列化出来。FileSerializable接口继承了java.io. Serializable接口。要求当序列化一个对象时，这个对象的所有数据成员都必须是可序列化的(即都实现java.io. Serializable接口），因此这四个类都必须实现FileSerializable接口。

预定义的抽象类及接口说明

- **package hust.cs.javacourse.search.index**包里还定义了构建索引时用到的中间数据结构，当解析一个文本文档时，会用到这些数据结构，包括：
  - AbstractTermTuple：其具体子类实例为和单词term相关的三元组，包括三个数据成员：
    - AbstractTerm term：代表当前解析得到的一个term
    - final int freq = 1：因为解析得到了一个term，该term出现了一次，因此其频率为1
    - int curPos：该term的位置（注意位置序号是以term为单位不是以字符为单位）
  - AbstractDocument：其具体子类实例为：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。

预定义的抽象类及接口说明

原始文档

由一个原始文本文件解析得到AbstractDocument子类对象

一个AbstractTermTuple子类对象就是一个term三元组

三元组：<“hello”，1，1>

三元组：<“java”，1，2>

三元组：<“hello”，1，3>

三元组：<“python”，1，4>

docId：文档Id

docPath：文档绝对路径

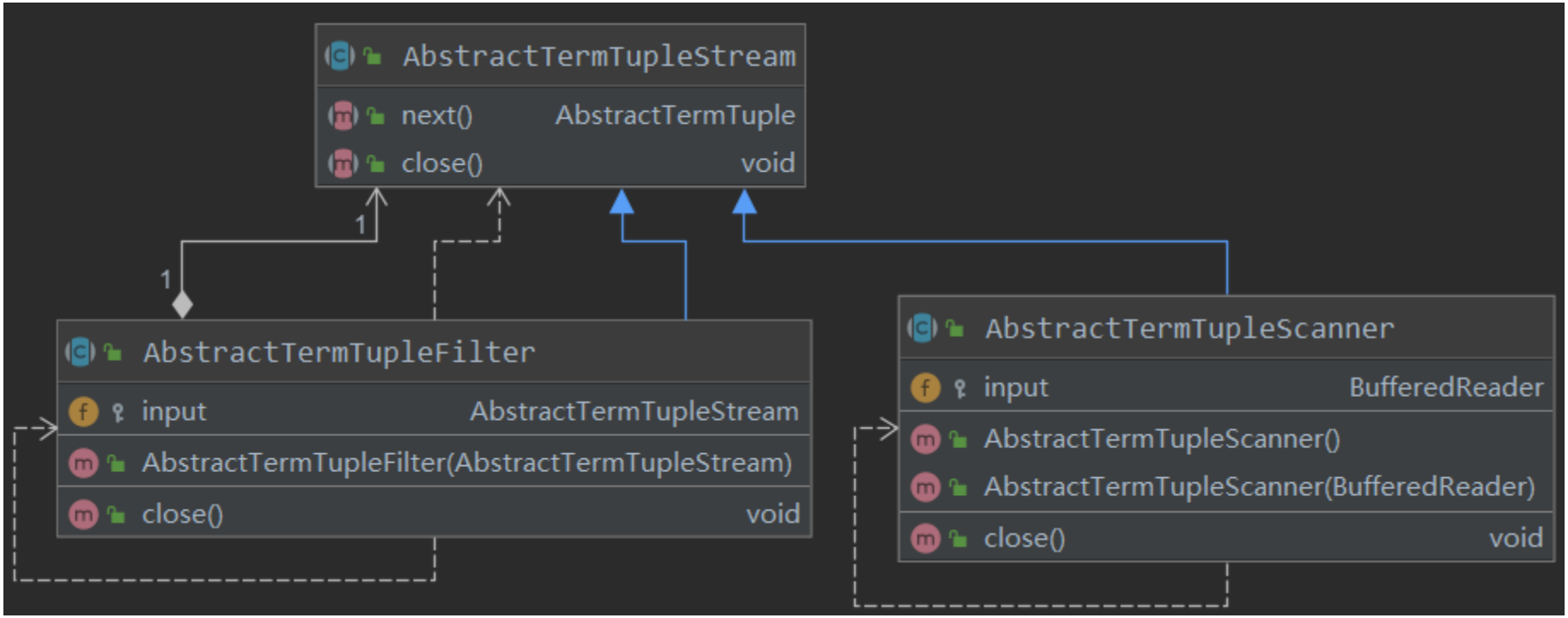
AbstractDocument子类对象，包含四个term三元组

预定义的抽象类及接口说明

- **package hust.cs.javacourse.search.index**包里还有二个抽象类规定了如何构建AbstractDocument子类对象和倒排索引，它们是：
  - AbstractDocumentBuilder：里面定义了二个抽象方法，这二个方法返回AbstractDocument子类对象，它们是重载的二个抽象函数，第三个参数分别为AbstractTermTupleStream 类型（代表一个被解析文档的三元组流对象）和File对象
    - public abstract AbstractDocument build(int docId, String docPath, AbstractTermTupleStream termTupleStream);
    - public abstract AbstractDocument build(int docId, String docPath, File file);
  - AbstractIndexBuilder：里面包含一个数据成员，定义了抽象方法，分别为：
    - AbstractDocumentBuilder docBuilder
    - public abstract AbstractIndex buildIndex(String rootDirectory);
    - 构建AbstractIndexBuilder子类对象时必须传入先构造好的AbstractDocumentBuilder子类对象

预定义的抽象类及接口说明

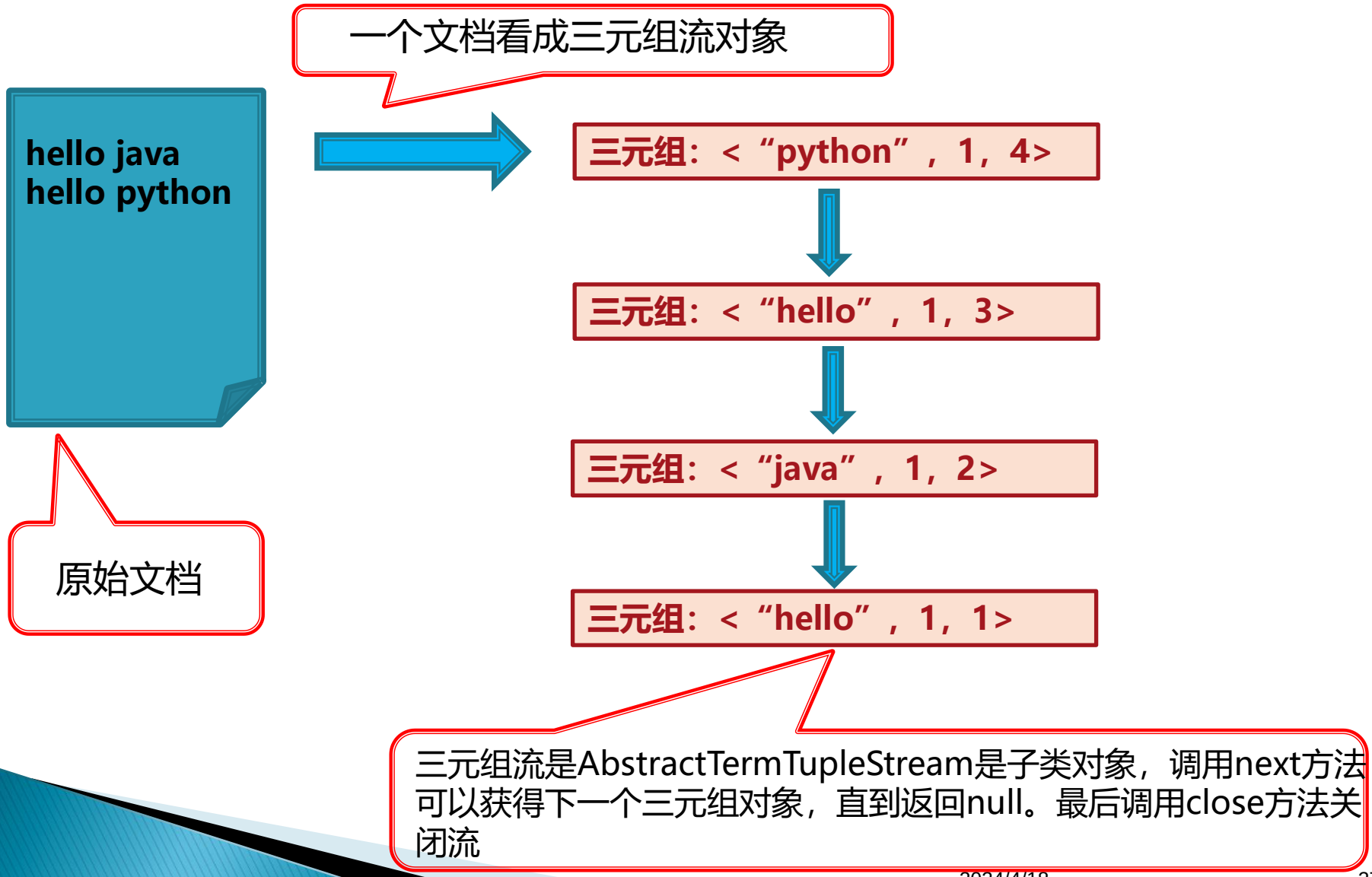
- **package hust.cs.javacourse.search.parser**：包里定义的抽象类和接口的UML模型见下图：



预定义的抽象类及接口说明

- **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，学生只需要实现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
- java.io包里包含了很多对文件系统访问的API，这些API都是基于流访问。因此这个包里的类也采用同样方法，同时也可以让了解设计模式中的“装饰者模式”。

预定义的抽象类及接口说明



预定义的抽象类及接口说明

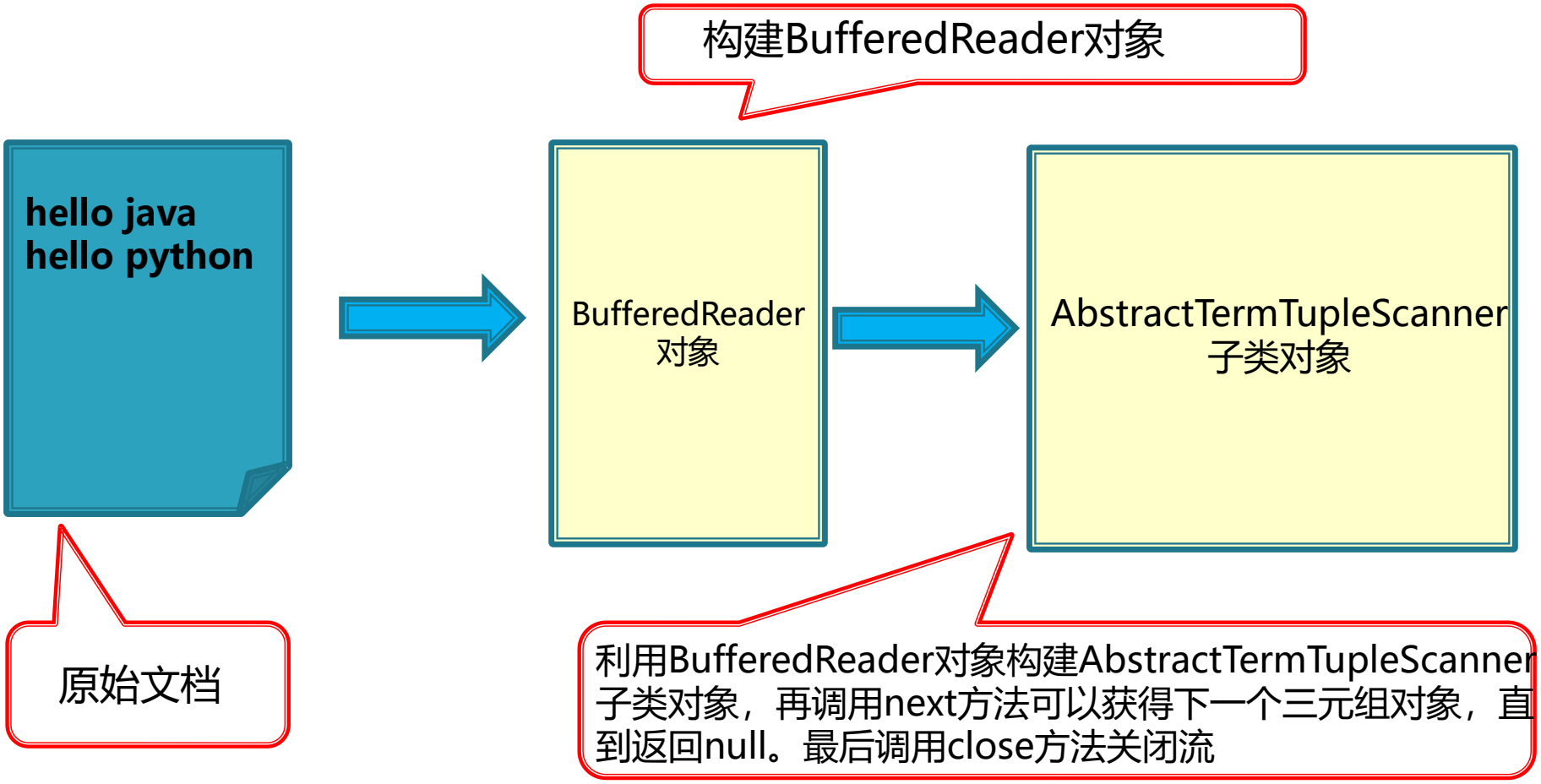
- **hust.cs.javacourse.search.parser. AbstractTermTupleScanner**
- 那么一个文件如何和AbstractTermTupleStream子类流对象关联起来呢？通过AbstractTermTupleScanner类，这个类定义了如下数据成员和构造函数：
  - protected BufferedReader input ;
  - public AbstractTermTupleScanner(BufferedReader input)
- 即通过java.io.BufferedReader关联到文件，BufferedReader的readLine方法可以让我们一次从文本文件里读取一行，当读取到文件末尾返回null。
- 将java.io.BufferedReader关联到文件通过下列语句：

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

其中filePath是文件的绝对路径。注意BufferedReader 只能读取文本文件。

- 因此我们只要先构造好BufferedReader对象，再用该对象为参数去构造AbstractTermTupleScanner子类对象，再调用自己实现好的next方法就可以获取一个个三元组

预定义的抽象类及接口说明



预定义的抽象类及接口说明

- **hust.cs.javacourse.search.parser. AbstractTermTupleFilter**
- 文本文档里停用词、非英文单词如数字、过长或过短的单词对应的三元组不是我们想要的。那么如何去掉通过AbstractTermTupleScanner子类对象获得的三元组呢？通过AbstractTermTupleFilter子类对象。同时采用设计模式“装饰者模式” 来设计代码
  - java.io包里大量使用了“装饰者模式” ，例如第28页PPT里创建BufferedReader的代码就是装饰者设计模式：首先构建File对象，外面再包装（装饰）一层FileInputStream，外面再包装一层InputStreamReader，最后包装一层BufferedReader。
- AbstractTermTupleFilter包含有：
  - 数据成员：AbstractTermTupleStream input
  - 构造函数： AbstractTermTupleFilter(AbstractTermTupleStream input)
- 注意到数据成员类型和构造函数参数类型都是AbstractTermTupleStream
- AbstractTermTupleFilter也是AbstractTermTupleStream类型
- AbstractTermTupleScanner也是AbstractTermTupleStream类型
- 这是装饰者模式最关键之处：被装饰的对象和装饰以后得到的新对象都有共同的抽象祖先类，同时都会覆盖实现自己的next方法

预定义的抽象类及接口说明

- **hust.cs.javacourse.search.parser. AbstractTermTupleFilter**
- AbstractTermTupleFilter的构造函数参数指向被装饰的对象，数据成员input也就指向了被装饰的对象
- 构造函数产生的新对象和被装饰的对象都是AbstractTermTupleStream类型对象
- 假设已经实现了AbstractTermTupleScanner的一个具体子类SimpleScanner，这个类必须实现了next方法，获取三元组流中的下一个三元组
- 假设也已经实现AbstractTermTupleFilter的一个具体子类SimpleFilter，这个类的next方法里应该首先调用input.next()方法获取被装饰对象的next方法返回的三元组;然后根据SimpleFilter的过滤规则判断这个三元组是否需要过滤：如果不需要过滤则从next直接返回这个三元组；如果需要过滤，则这个三元组跳过，继续调用input.next()方法获取被装饰对象的next方法返回的下一个三元组
- 注意在装饰者的next方法里调用input.next()方法时，根据多态特性， input.next()一定会调用input所引用的被装饰对象的next方法
- 把SimpleScanner和SimpleFilter组合在一起使用的代码应该为：

预定义的抽象类及接口说明

```
BufferedReader  reader = new BufferedReader(
    new InputStreamReader(new FileInputStream(new File(filePath))));
AbstractTermTupleStream scanner = new SimpleScanner(readrer);
AbstractTermTupleStream filter = new SimpleFilter(scanner);
```

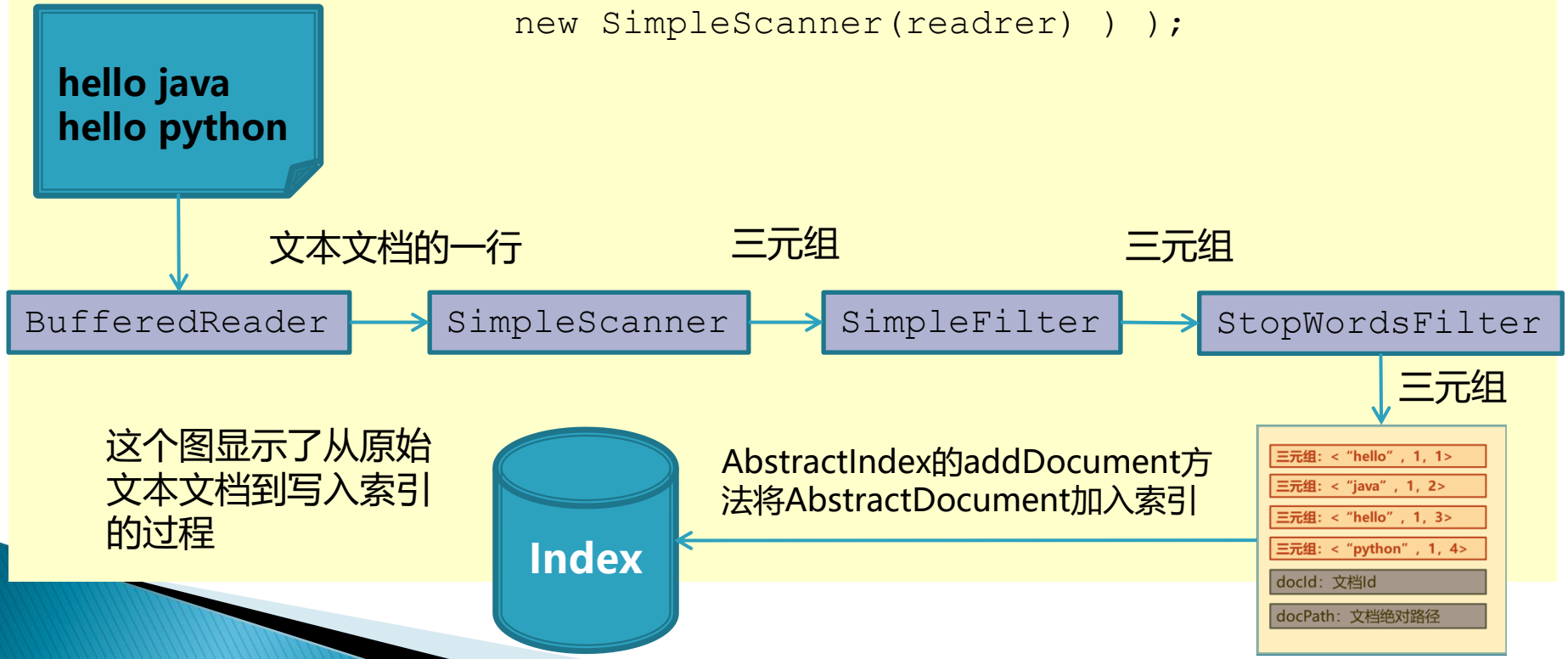
装饰者，filter里的input指向了scanner

被装饰者

```
//再调用filter.next()方法获取三元组，这时凡满足SimpleFiler过滤规则的三元组都会被去掉
//如果我们又实现了另外一个过滤器StopWordsFilter，那么可以把多个对象层层包装
BufferedReader  reader = new BufferedReader(
    new InputStreamReader(new FileInputStream(new File(filePath))));
AbstractTermTupleStream ts =
            new StopWordsFilter(new SimpleFilter (
                    new SimpleScanner(readrer) ) );
```

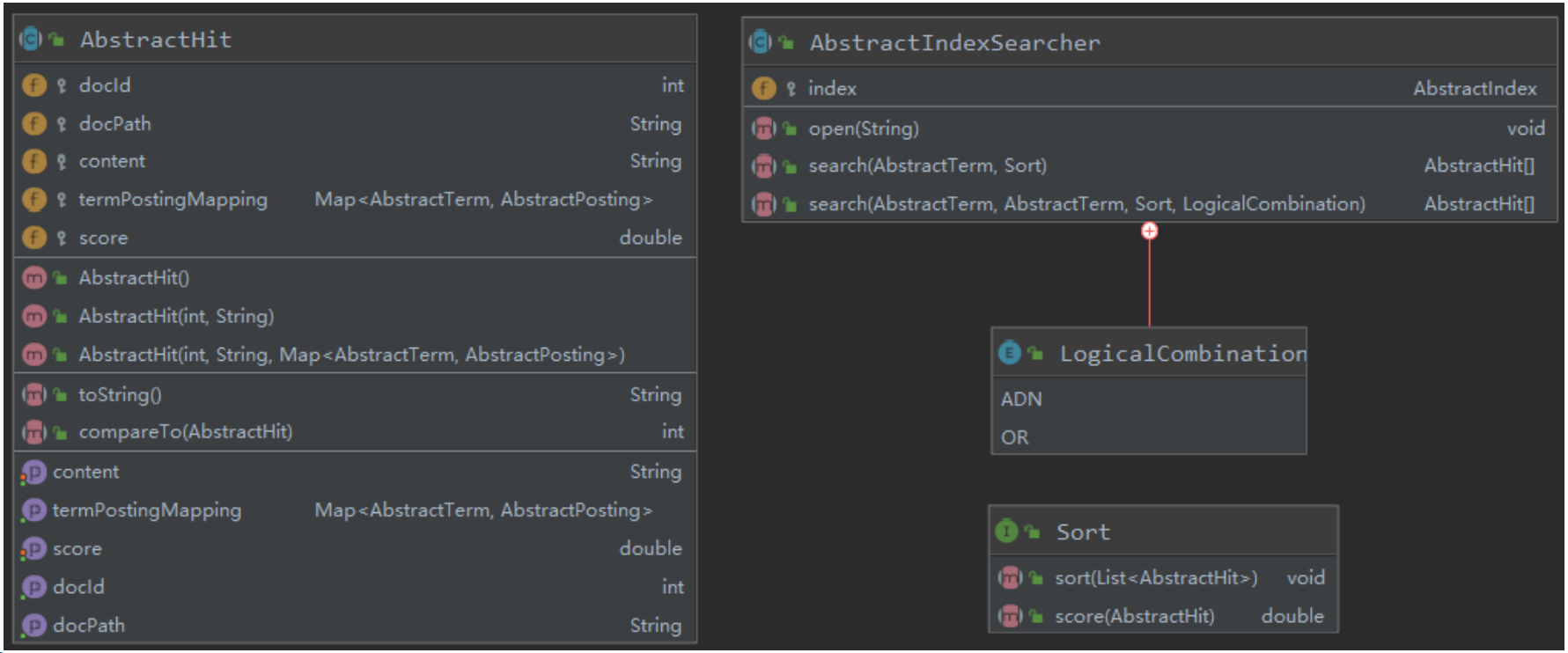
预定义的抽象类及接口说明

```
//如果我们又实现了另外一个过滤器StopWordsFilter，那么可以把多个对象层层包装
BufferedReader  reader = new BufferedReader(
    new InputStreamReader(new FileInputStream(new File(filePath))));
AbstractTermTupleStream ts =
              new StopWordsFilter(new SimpleFilter (
                    new SimpleScanner(readrer) ) );
```



预定义的抽象类及接口说明

- **package hust.cs.javacourse.search.query**：包里定义了如下抽象类和接口：
  - AbstractHit
  - AbstractIndexSearcher
  - Sort接口



预定义的抽象类及接口说明

- **package hust.cs.javacourse.search.query. AbstractHit**
- 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。类里定义的数据成员有：
  - int docId：文档Id
  - String docPath：文档绝对路径
  - String content: 文档内容
  - double score = 1.0：命中结果得分，默认值为1.0，得分通过Sort接口计算
  - Map<AbstractTerm, AbstractPosting> termPostingMapping：命中的单词和对应的Posting键值对，对计算文档得分有用，对于一个查询命中结果（一个文档），一个term对应的是Posting而不是PostingList。
  - 例如我们可以把Posting里单词的freq作为计算文档得分的重要依据，频率越高越应该排前面。(当我们用Java作为检索词时，出现了10次Java的文档应该排在只出现1次Java的文档的前面）。另外termPostingMapping可以支持文档显示时命中词的Highlight显示（本实验不要求）

预定义的抽象类及接口说明

- **package hust.cs.javacourse.search.query. AbstractIndexSearcher**：进行全文检索的抽象类，定义了下面的数据成员和抽象方法：
  - protected AbstractIndex index = new Index()：注意这里的Index是AbstractIndex的具体子类，因为AbstractIndexSearcher内部应该包含一个index对象，比较好的方式是当AbstractIndexSearcher子类对象被实例化时，其内部的index也应该被实例化（但索引内容为空），因此在hust.cs.javacourse.search.index.impl里定义了具体类Index，但是这个类所有的需要覆盖的方法体为空，留给学生去实现。
  - public static enum LogicalCombination：一个嵌套的静态枚举类型，定义了多个检索词之间的与或关系
  - public abstract void open(String indexFile)：打开指定的索引文件。open方法里应该调用内部包含的index对象的load方法，把索引文件反序列化到index对象里。这个索引文件应该是索引构建好以后序列化好的目标文件。
  - public abstract AbstractHit[] search(AbstractTerm queryTerm, Sort sorter)：单个检索词的检索，方法的第二个参数应该传入实现了Sort接口的具体类的实例，对命中结果进行排序。方法返回AbstractHit子类实例数组（已经排好序）。
  - public abstract AbstractHit[] search(AbstractTerm queryTerm1, AbstractTerm queryTerm2, Sort sorter, LogicalCombination combine):二个检索词的检索

预定义的抽象类及接口说明

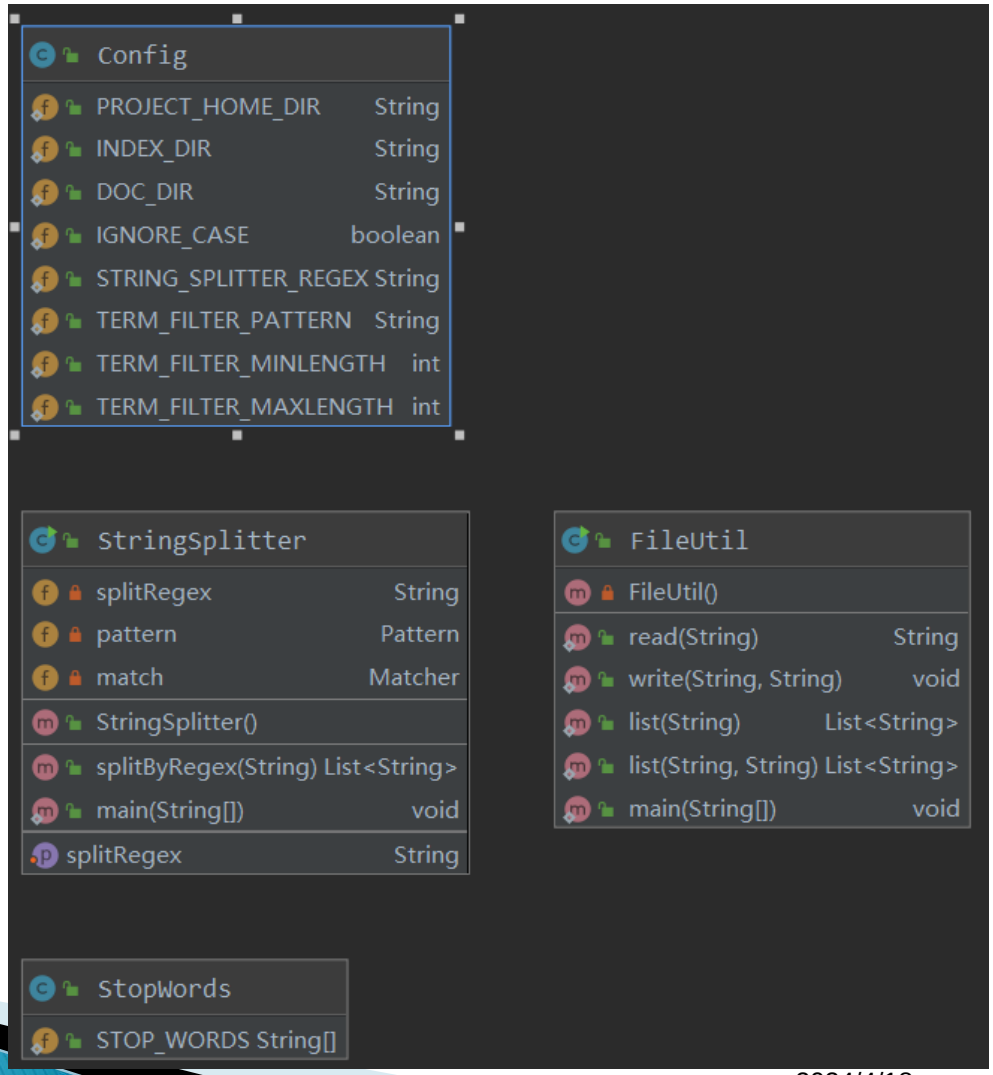
- **package hust.cs.javacourse.search.query. Sort**：对命中结果计算得分和排序的接口，包含以下接口方法：
  - public abstract void sort(List<AbstractHit> hits)：对命中结果集合排序，排序依据是每个文档的得分
  - public abstract double score(AbstractHit hit)：计算一个命中结果的得分
- 计算文档的得分可以采取不同的策略, 因此这里的设计模式采用了策略模式，没有把这个方法放到AbstractHit及其子类里。而是放到接口Sort里
- 当我们需要不同的排序策略，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.
- Tips：如果集合里每个元素都实现了Comparable接口，那么只需要调用

  Collections.sort(hits)

  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。但我们需要的是按得分从大到小排序怎么办？把文档得分都取 负值即可。

预定义的抽象类及接口说明

- **package hust.cs.javacourse.search.util**：包里定义了工具类，学生可以直接使用，建议学生看看具体实现也熟悉Java JDK API



预定义的抽象类及接口说明

- **package hust.cs.javacourse.search.util**：包里定义的工具类有：
  - Config：定义了搜索引擎的配置信息
  - StopWords：定义了停用词表
  - StringSplitter：基于正则表达式的分词器，根据标点符号和空白符将英文字符串分成terms
  - FileUtil：提供了文本文件的读、写、遍历目录方法
- 这些工具类学生可以直接使用。建议对Java JDK不熟悉的同学应该仔细阅读代码以熟悉API
- 这些类不再这里描述，各位同学直接看代码

预定义的抽象类及接口说明

- 这一部分最后需要说明的是：
- 所有预定义的抽象类和接口已经生成好Javadoc API文档
- 同时所有预定义的抽象类和接口已经生成好对应的UML模型，UML模型以图片和.uml文件二种形式提供给各位同学。
- Javadoc API文档和UML模型都是通过IDEA自动生成的，因此.uml文件可以在IDEA里打开并浏览
- Javadoc API文档位于工程目录的子目录javadoc下，各位同学点击该目录下的index.html即可浏览API文档
- UML模型文件位于工程目录的子目录model下
- 在工程目录的子目录text下有二个简单的txt文档，可以测试程序的功能。最终的测试文档会在后面发布
- 构建好的索引放在工程目录的index子目录
- 创建好的工程也会发布给各位同学

内容

- 搜索引擎的倒排索引数据结构
- 预定义的抽象类及接口API说明
- 本实验所涉及的JDK Java API说明

本实验所涉及的JDK Java API说明

- 首先本实验里面大量使用了Java JDK里面提供的集合类，例如
  - List接口：具体子类有ArrayList
  - Map接口：具体子类有TreeMap（可以自动对key排序，而HashMap不能对键值排序）
- 因此必须熟悉这些集合类的使用，否则无法顺利完成本实验
  - 对这些集合类的操作包括：遍历、添加、排序，得到集合的大小、判断集合里是否已存在指定元素
  - 具体使用请参考教材的进阶篇第20章和第21章
- 对应Map类型集合的遍历，推荐使用下面方法：

本实验所涉及的JDK Java API说明

```
Map<Integer, String> map = new TreeMap<>();
//先添加几个元素
map.put(1, "aaa");
map.put(2, "bbb");
map.put(3, "ccc");
map.put(4, "ddd");

//再遍历
StringBuffer buf = new StringBuffer();
//通过迭代器遍历。Java的集合类都实现了迭代器接口
Iterator<Map.Entry<Integer,String>> it = map.entrySet().iterator();
while (it.hasNext()){
    Map.Entry<Integer,String> entry = it.next();
    buf.append(entry.getKey() + "   ---->   " +
                    entry.getValue()).append("\n");
}
System.out.println(buf.toString());
```

```
D:\jdk1.8.0_231_64bit\bin\java.exe ...
1   ---->   aaa
2   ---->   bbb
3   ---->   ccc
4   ---->   ddd
```

本实验所涉及的JDK Java API说明

- 第二要熟悉对文件，特别是文本文件的操作
- 如果读取文本文件，推荐使用BufferedReader
- 如果写文本文件，推荐使用PrintWriter，当创建好PrintWriter对象后，调用其println和print方法可以将字符串一行行的写入到文本文件，使用方法与System.out.println, System.out.print完全一样
- 具体使用方法，请见hust.cs.javacourse.search.util.FileUtils类的read方法和write方法

本实验所涉及的JDK Java API说明

- 第三是熟悉对象的序列化和反序列化。可以参考教材基础篇17.6节。
- 但必须要注意的是本实验要求是作为类的方法来实现下面二个方法：
  - public abstract void writeObject(ObjectOutputStream out);
  - public abstract void readObject(ObjectInputStream in);
- 另外要注意的是对象序列化文件后是二进制文件不是文本文件。因此如果要完成本实验要求的将内存中的索引写到文本文件，不能采用上面二个方法。
- 推荐的做法是：在AbstractIndex的子类覆盖toString方法，将内存的内容转换成格式良好的字符串，再用FileUtils类的write方法写到文本文件里。
- 那么AbstractIndex子类里的数据成员怎么转换成格式良好的字符串？这里就看出覆盖toString方法的重要性：如果实现好AbstractTerm子类、AbstractPosting子类、AbstractPostingList子类的toString方法，那么实现AbstractIndex子类的toString就很简单：嵌套调用就行了。