Lucene：基于Java的全文检索引擎简介

作者：[车东](http://www.chedong.com) 发表于：2002-08-06 18:08 最后更新于：2009-03-20 23:03  
[版权声明](http://creativecommons.org/licenses/by/3.0/deed.zh)：可以任意转载，转载时请务必以超链接形式标明文章[原始出处](http://www.chedong.com/tech/lucene.html)和作者信息及[本声明](http://www.chedong.com/blog/archives/001249.html)。  
<http://www.chedong.com/tech/lucene.html>

Lucene是一个基于Java的全文索引工具包。

1. [基于Java的全文索引引擎Lucene简介：关于作者和Lucene的历史](http://www.chedong.com/cgi-bin/mt/mt.cgi#intro)
2. [全文检索的实现：Luene全文索引和数据库索引的比较](http://www.chedong.com/cgi-bin/mt/mt.cgi#compare)
3. [中文切分词机制简介：基于词库和自动切分词算法的比较](http://www.chedong.com/cgi-bin/mt/mt.cgi#segment)
4. [具体的安装和使用简介：系统结构介绍和演示](http://www.chedong.com/cgi-bin/mt/mt.cgi#demo)
5. [Hacking Lucene：简化的查询分析器，删除的实现，定制的排序，应用接口的扩展](http://www.chedong.com/cgi-bin/mt/mt.cgi#hacking)
6. [从Lucene我们还可以学到什么](http://www.chedong.com/cgi-bin/mt/mt.cgi#learn)

另外，如果是在选择全文引擎，现在也许是试试[Sphinx](http://www.sphinxsearch.com/)的时候了：相比Lucene速度更快，[有中文分词的支持](http://www.coreseek.com)，而且内置了对简单的分布式检索的支持；

**基于Java的全文索引/检索引擎——Lucene**

Lucene不是一个完整的全文索引应用，而是是一个用Java写的全文索引引擎工具包，它可以方便的嵌入到各种应用中实现针对应用的全文索引/检索功能。

Lucene的作者：Lucene的贡献者[Doug Cutting](http://www.nutch.org/blog/cutting.html)是一位资深全文索引/检索专家，曾经是V-Twin搜索引擎(Apple的Copland操作系统的成就之一)的主要开发者，后在Excite担任高级系统架构设计师，目前从事于一些INTERNET底层架构的研究。他贡献出的Lucene的目标是为各种中小型应用程序加入全文检索功能。

Lucene的发展历程：早先发布在作者自己的[www.lucene.com](http://www.lucene.com)，后来发布在[SourceForge](http://sourceforge.net/projects/lucene/)，2001年年底成为APACHE基金会jakarta的一个子项目：<http://jakarta.apache.org/lucene/>

已经有很多Java项目都使用了Lucene作为其后台的全文索引引擎，比较著名的有：

* [Jive](http://www.jivesoftware.com)：WEB论坛系统；
* [Eyebrows](http://eyebrowse.tigris.org/)：邮件列表HTML归档/浏览/查询系统，本文的主要参考文档“[TheLucene search engine: Powerful, flexible, and free](http://www.javaworld.com/javaworld/jw-09-2000/jw-0915-lucene_p.html)”作者就是EyeBrows系统的主要开发者之一，而EyeBrows已经成为目前APACHE项目的主要邮件列表归档系统。
* [Cocoon](http://xml.apache.org/cocoon/index.html):基于XML的web发布框架，全文检索部分使用了Lucene
* [Eclipse](http://www.eclipse.org):基于Java的开放开发平台，帮助部分的全文索引使用了Lucene

对于中文用户来说，最关心的问题是其是否支持中文的全文检索。但通过后面对于Lucene的结构的介绍，你会了解到由于Lucene良好架构设计，对中文的支持只需对其语言词法分析接口进行扩展就能实现对中文检索的支持。

**全文检索的实现机制**

Lucene的API接口设计的比较通用，输入输出结构都很像数据库的表==>记录==>字段，所以很多传统的应用的文件、数据库等都可以比较方便的映射到Lucene的存储结构/接口中。总体上看：可以先把**Lucene当成一个支持全文索引的数据库系统**。

比较一下Lucene和数据库：

|  |  |
| --- | --- |
| Lucene | 数据库 |
| 索引数据源：doc(field1,field2...) doc(field1,field2...)  \ indexer /  \_\_\_\_\_\_\_\_\_\_\_\_\_  | Lucene Index|  --------------  / searcher \  结果输出：Hits(doc(field1,field2) doc(field1...)) | 索引数据源：record(field1,field2...) record(field1..)  \ SQL: insert/  \_\_\_\_\_\_\_\_\_\_\_\_\_  | DB Index |  -------------  / SQL: select \ 结果输出：results(record(field1,field2..) record(field1...)) |
| Document：一个需要进行索引的“单元” 一个Document由多个字段组成 | Record：记录，包含多个字段 |
| Field：字段 | Field：字段 |
| Hits：查询结果集，由匹配的Document组成 | RecordSet：查询结果集，由多个Record组成 |

**全文检索 ≠ like "%keyword%"**

通常比较厚的书籍后面常常附关键词索引表（比如：北京：12, 34页， 上海：3,77页……），它能够帮助读者比较快地找到相关内容的页码。而数据库索引能够大大提高查询的速度原理也是一样，想像一下通过书后面的索引查找的速度要比一页一页地翻内容高多少倍……而索引之所以效率高，另外一个原因是它是排好序的。**对于检索系统来说核心是一个排序问题**。

由于数据库索引不是为全文索引设计的，因此，**使用like "%keyword%"时，数据库索引是不起作用的**，在使用like查询时，搜索过程又变成类似于一页页翻书的遍历过程了，所以对于含有模糊查询的数据库服务来说，LIKE对性能的危害是极大的。如果是需要对多个关键词进行模糊匹配：like"%keyword1%" and like "%keyword2%" ...其效率也就可想而知了。

所以建立一个高效检索系统的关键是建立一个类似于科技索引一样的反向索引机制，将数据源（比如多篇文章）排序顺序存储的同时，有另外一个排好序的关键词列表，用于存储关键词==>文章映射关系，利用这样的映射关系索引：[关键词==>出现关键词的文章编号，出现次数（甚至包括位置：起始偏移量，结束偏移量），出现频率]，检索过程就是把**模糊查询变成多个可以利用索引的精确查询的逻辑组合的过程**。从而大大提高了多关键词查询的效率，所以，全文检索问题归结到最后是一个排序问题。

由此可以看出模糊查询相对数据库的精确查询是一个非常不确定的问题，这也是大部分数据库对全文检索支持有限的原因。Lucene最核心的特征是通过特殊的索引结构实现了传统数据库不擅长的全文索引机制，并提供了扩展接口，以方便针对不同应用的定制。

可以通过一下表格对比一下数据库的模糊查询：

|  |  |  |
| --- | --- | --- |
|  | Lucene全文索引引擎 | 数据库 |
| 索引 | 将数据源中的数据都通过全文索引一一建立反向索引 | 对于LIKE查询来说，数据传统的索引是根本用不上的。数据需要逐个便利记录进行GREP式的模糊匹配，比有索引的搜索速度要有多个数量级的下降。 |
| 匹配效果 | 通过词元(term)进行匹配，通过语言分析接口的实现，可以实现对中文等非英语的支持。 | 使用：like "%net%" 会把netherlands也匹配出来， 多个关键词的模糊匹配：使用like "%com%net%"：就不能匹配词序颠倒的xxx.net..xxx.com |
| 匹配度 | 有匹配度算法，将匹配程度（相似度）比较高的结果排在前面。 | 没有匹配程度的控制：比如有记录中net出现5词和出现1次的，结果是一样的。 |
| 结果输出 | 通过特别的算法，将最匹配度最高的头100条结果输出，结果集是缓冲式的小批量读取的。 | 返回所有的结果集，在匹配条目非常多的时候（比如上万条）需要大量的内存存放这些临时结果集。 |
| 可定制性 | 通过不同的语言分析接口实现，可以方便的定制出符合应用需要的索引规则（包括对中文的支持） | 没有接口或接口复杂，无法定制 |
| 结论 | 高负载的模糊查询应用，需要负责的模糊查询的规则，索引的资料量比较大 | 使用率低，模糊匹配规则简单或者需要模糊查询的资料量少 |

**全文检索和数据库应用最大的不同在于：让最相关的头100条结果满足98%以上用户的需求**  
Lucene的创新之处：

大部分的搜索（数据库）引擎都是用B树结构来维护索引，索引的更新会导致大量的IO操作，Lucene在实现中，对此稍微有所改进：不是维护一个索引文件，而是在扩展索引的时候不断创建新的索引文件，然后定期的把这些新的小索引文件合并到原先的大索引中（针对不同的更新策略，批次的大小可以调整），这样在不影响检索的效率的前提下，提高了索引的效率。

Lucene和其他一些全文检索系统/应用的比较：

|  |  |  |
| --- | --- | --- |
|  | Lucene | 其他开源全文检索系统 |
| 增量索引和批量索引 | 可以进行增量的索引(Append)，可以对于大量数据进行批量索引，并且接口设计用于优化批量索引和小批量的增量索引。 | 很多系统只支持批量的索引，有时数据源有一点增加也需要重建索引。 |
| 数据源 | Lucene没有定义具体的数据源，而是一个文档的结构，因此可以非常灵活的适应各种应用（只要前端有合适的转换器把数据源转换成相应结构）， | 很多系统只针对网页，缺乏其他格式文档的灵活性。 |
| 索引内容抓取 | Lucene的文档是由多个字段组成的，甚至可以控制那些字段需要进行索引，那些字段不需要索引，近一步索引的字段也分为需要分词和不需要分词的类型：    需要进行分词的索引，比如：标题，文章内容字段    不需要进行分词的索引，比如：作者/日期字段 | 缺乏通用性，往往将文档整个索引了 |
| 语言分析 | 通过语言分析器的不同扩展实现： 可以过滤掉不需要的词：an the of 等， 西文语法分析：将jumps jumped jumper都归结成jump进行索引/检索 非英文支持：对亚洲语言，阿拉伯语言的索引支持 | 缺乏通用接口实现 |
| 查询分析 | 通过查询分析接口的实现，可以定制自己的查询语法规则： 比如： 多个关键词之间的 + - and or关系等 |  |
| 并发访问 | 能够支持多用户的使用 |  |

**关于亚洲语言的的切分词问题(Word Segment)**

对于中文来说，全文索引首先还要解决一个语言分析的问题，对于英文来说，语句中单词之间是天然通过空格分开的，但亚洲语言的中日韩文语句中的字是一个字挨一个，所有，首先要把语句中按“词”进行索引的话，这个词如何切分出来就是一个很大的问题。

首先，肯定不能用单个字符作(si-gram)为索引单元，否则查“上海”时，不能让含有“海上”也匹配。

但一句话：“北京天安门”，计算机如何按照中文的语言习惯进行切分呢？  
“北京 天安门” 还是“北 京 天安门”？让计算机能够按照语言习惯进行切分，往往需要机器有一个比较丰富的词库才能够比较准确的识别出语句中的单词。

另外一个解决的办法是采用自动切分算法：将单词按照2元语法(bigram)方式切分出来，比如：  
"北京天安门" ==> "北京 京天 天安 安门"。

这样，在查询的时候，无论是查询"北京" 还是查询"天安门"，将查询词组按同样的规则进行切分："北京"，"天安安门"，多个关键词之间按与"and"的关系组合，同样能够正确地映射到相应的索引中。这种方式对于其他亚洲语言：韩文，日文都是通用的。

基于自动切分的最大优点是没有词表维护成本，实现简单，缺点是索引效率低，但对于中小型应用来说，基于2元语法的切分还是够用的。基于2元切分后的索引一般大小和源文件差不多，而对于英文，索引文件一般只有原文件的30%-40%不同，

|  |  |  |
| --- | --- | --- |
|  | 自动切分 | 词表切分 |
| 实现 | 实现非常简单 | 实现复杂 |
| 查询 | 增加了查询分析的复杂程度， | 适于实现比较复杂的查询语法规则 |
| 存储效率 | 索引冗余大，索引几乎和原文一样大 | 索引效率高，为原文大小的30％左右 |
| 维护成本 | 无词表维护成本 | 词表维护成本非常高：中日韩等语言需要分别维护。 还需要包括词频统计等内容 |
| 适用领域 | 嵌入式系统：运行环境资源有限 分布式系统：无词表同步问题 多语言环境：无词表维护成本 | 对查询和存储效率要求高的专业搜索引擎 |

目前比较大的搜索引擎的语言分析算法一般是基于以上2个机制的结合。关于中文的语言分析算法，大家可以在Google查关键词"wordsegment search"能找到更多相关的资料。

**安装和使用**

下载：<http://jakarta.apache.org/lucene/>

注意：Lucene中的一些比较复杂的词法分析是用JavaCC生成的（JavaCC：JavaCompilerCompiler，纯Java的词法分析生成器），所以如果从源代码编译或需要修改其中的QueryParser、定制自己的词法分析器，还需要从<https://javacc.dev.java.net/>下载javacc。

lucene的组成结构：对于外部应用来说索引模块(index)和检索模块(search)是主要的外部应用入口

|  |  |
| --- | --- |
| org.apache.Lucene.search/ | 搜索入口 |
| org.apache.Lucene.index/ | 索引入口 |
| org.apache.Lucene.analysis/ | 语言分析器 |
| org.apache.Lucene.queryParser/ | 查询分析器 |
| org.apache.Lucene.document/ | 存储结构 |
| org.apache.Lucene.store/ | 底层IO/存储结构 |
| org.apache.Lucene.util/ | 一些公用的数据结构 |

简单的例子演示一下Lucene的使用方法：

索引过程：从命令行读取文件名（多个），将文件分路径(path字段)和内容(body字段)2个字段进行存储，并对内容进行全文索引：索引的单位是Document对象，每个Document对象包含多个字段Field对象，针对不同的字段属性和数据输出的需求，对字段还可以选择不同的索引/存储字段规则，列表如下：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **方法** | **切词** | **索引** | **存储** | **用途** |
| Field.Text(String name, String value) | Yes | Yes | Yes | 切分词索引并存储，比如：标题，内容字段 |
| Field.Text(String name, Reader value) | Yes | Yes | No | 切分词索引不存储，比如：META信息， 不用于返回显示，但需要进行检索内容 |
| Field.Keyword(String name, String value) | No | Yes | Yes | 不切分索引并存储，比如：日期字段 |
| Field.UnIndexed(String name, String value) | No | No | Yes | 不索引，只存储，比如：文件路径 |
| Field.UnStored(String name, String value) | Yes | Yes | No | 只全文索引，不存储 |

public class IndexFiles {   
 //使用方法：: IndexFiles [索引输出目录] [索引的文件列表] ...   
 public static void main(String[] args) throws Exception {  
 String indexPath = args[0];  
 IndexWriter writer;  
 //用指定的语言分析器构造一个新的写索引器（第3个参数表示是否为追加索引）  
 writer = new IndexWriter(indexPath, new SimpleAnalyzer(), false);  
  
 for (int i=1; i<args.length; i++) {  
 System.out.println("Indexing file " + args[i]);  
 InputStream is = new FileInputStream(args[i]);  
  
 //构造包含2个字段Field的Document对象  
 //一个是路径path字段，不索引，只存储  
 //一个是内容body字段，进行全文索引，并存储  
 Document doc = new Document();  
 doc.add(Field.UnIndexed("path", args[i]));  
 doc.add(Field.Text("body", (Reader) new InputStreamReader(is)));  
 //将文档写入索引  
 writer.addDocument(doc);  
 is.close();  
 };  
 //关闭写索引器  
 writer.close();  
 }  
}

索引过程中可以看到：

* 语言分析器提供了抽象的接口，因此语言分析(Analyser)是可以定制的，虽然lucene缺省提供了2个比较通用的分析器SimpleAnalyser和StandardAnalyser，这2个分析器缺省都不支持中文，所以要加入对中文语言的切分规则，需要修改这2个分析器。
* Lucene并没有规定数据源的格式，而只提供了一个通用的结构（Document对象）来接受索引的输入，因此输入的数据源可以是：数据库，WORD文档，PDF文档，HTML文档……只要能够设计相应的解析转换器将数据源构造成成Docuement对象即可进行索引。
* 对于大批量的数据索引，还可以通过调整IndexerWrite的文件合并频率属性（mergeFactor）来提高批量索引的效率。

检索过程和结果显示：

搜索结果返回的是Hits对象，可以通过它再访问Document==>Field中的内容。

假设根据body字段进行全文检索，可以将查询结果的path字段和相应查询的匹配度(score)打印出来，

public class Search {   
 public static void main(String[] args) throws Exception {  
 String indexPath = args[0], queryString = args[1];  
 //指向索引目录的搜索器  
 Searcher searcher = new IndexSearcher(indexPath);  
 //查询解析器：使用和索引同样的语言分析器  
 Query query = QueryParser.parse(queryString, "body",   
 new SimpleAnalyzer());  
 //搜索结果使用Hits存储  
 Hits hits = searcher.search(query);  
 //通过hits可以访问到相应字段的数据和查询的匹配度  
 for (int i=0; i<hits.length(); i++) {  
 System.out.println(hits.doc(i).get("path") + "; Score: " +   
 hits.score(i));  
 };  
 }  
}

在整个检索过程中，语言分析器，查询分析器，甚至搜索器（Searcher）都是提供了抽象的接口，可以根据需要进行定制。

**Hacking Lucene**

**简化的查询分析器**

个人感觉lucene成为JAKARTA项目后，画在了太多的时间用于调试日趋复杂QueryParser，而其中大部分是大多数用户并不很熟悉的，目前LUCENE支持的语法：

Query ::= ( Clause )\*  
Clause ::= ["+", "-"] [<TERM> ":"] ( <TERM> | "(" Query ")")

中间的逻辑包括：and or + - &&||等符号，而且还有"短语查询"和针对西文的前缀/模糊查询等，个人感觉对于一般应用来说，这些功能有一些华而不实，其实能够实现目前类似于Google的查询语句分析功能其实对于大多数用户来说已经够了。所以，Lucene早期版本的QueryParser仍是比较好的选择。

**添加修改删除指定记录（Document）**

Lucene提供了索引的扩展机制，因此索引的动态扩展应该是没有问题的，而指定记录的修改也似乎只能通过记录的删除，然后重新加入实现。如何删除指定的记录呢？删除的方法也很简单，只是需要在索引时根据数据源中的记录ID专门另建索引，然后利用IndexReader.delete(Termterm)方法通过这个记录ID删除相应的Document。

**根据某个字段值的排序功能**

lucene缺省是按照自己的相关度算法（score）进行结果排序的，但能够根据其他字段进行结果排序是一个在LUCENE的开发邮件列表中经常提到的问题，很多原先基于数据库应用都需要除了基于匹配度（score）以外的排序功能。而从全文检索的原理我们可以了解到，任何不基于索引的搜索过程效率都会导致效率非常的低，如果基于其他字段的排序需要在搜索过程中访问存储字段，速度回大大降低，因此非常是不可取的。

但这里也有一个折中的解决方法：在搜索过程中能够影响排序结果的只有索引中已经存储的docID和score这2个参数，所以，基于score以外的排序，其实可以通过将数据源预先排好序，然后根据docID进行排序来实现。这样就避免了在LUCENE搜索结果外对结果再次进行排序和在搜索过程中访问不在索引中的某个字段值。

这里需要修改的是IndexSearcher中的HitCollector过程：

...  
　scorer.score(new HitCollector() {  
 private float minScore = 0.0f;  
 public final void collect(int doc, float score) {  
 if (score > 0.0f && // ignore zeroed buckets  
 (bits==null || bits.get(doc))) { // skip docs not in bits  
 totalHits[0]++;  
 if (score >= minScore) {  
 /\* 原先：Lucene将docID和相应的匹配度score例入结果命中列表中：  
 \* hq.put(new ScoreDoc(doc, score)); // update hit queue  
 \* 如果用doc 或 1/doc 代替 score，就实现了根据docID顺排或逆排  
 \* 假设数据源索引时已经按照某个字段排好了序，而结果根据docID排序也就实现了  
 \* 针对某个字段的排序，甚至可以实现更复杂的score和docID的拟合。  
 \*/  
 hq.put(new ScoreDoc(doc, (float) 1/doc ));   
 if (hq.size() > nDocs) { // if hit queue overfull  
 hq.pop(); // remove lowest in hit queue  
 minScore = ((ScoreDoc)hq.top()).score; // reset minScore  
 }  
 }  
 }  
 }  
 }, reader.maxDoc());

**更通用的输入输出接口**

虽然lucene没有定义一个确定的输入文档格式，但越来越多的人想到使用一个标准的中间格式作为Lucene的数据导入接口，然后其他数据，比如PDF只需要通过解析器转换成标准的中间格式就可以进行数据索引了。这个中间格式主要以XML为主，类似实现已经不下4，5个：

数据源: WORD PDF HTML DB other  
 \ | | | /  
 XML中间格式  
 |  
 Lucene INDEX

目前还没有针对MSWord文档的解析器，因为Word文档和基于ASCII的RTF文档不同，需要使用COM对象机制解析。这个是我在Google上查的相关资料：<http://www.intrinsyc.com/products/enterprise_applications.asp>  
另外一个办法就是把Word文档转换成text：<http://www.winfield.demon.nl/index.html>

**索引过程优化**

索引一般分2种情况，一种是小批量的索引扩展，一种是大批量的索引重建。在索引过程中，并不是每次新的DOC加入进去索引都重新进行一次索引文件的写入操作（文件I/O是一件非常消耗资源的事情）。

Lucene先在内存中进行索引操作，并根据一定的批量进行文件的写入。这个批次的间隔越大，文件的写入次数越少，但占用内存会很多。反之占用内存少，但文件IO操作频繁，索引速度会很慢。在IndexWriter中有一个MERGE\_FACTOR参数可以帮助你在构造索引器后根据应用环境的情况充分利用内存减少文件的操作。根据我的使用经验：缺省Indexer是每20条记录索引后写入一次，每将MERGE\_FACTOR增加50倍，索引速度可以提高1倍左右。

**搜索过程优化**

**lucene支持内存索引：这样的搜索比基于文件的I/O有数量级的速度提升。**[**http://www.onjava.com/lpt/a/3273**](http://www.onjava.com/lpt/a/3273) **而尽可能减少IndexSearcher的创建和对搜索结果的前台的缓存也是必要的。**

**Lucene面向全文检索的优化在于首次索引检索后，并不把所有的记录（Document）具体内容读取出来，而起只将所有结果中匹配度最高的头100条结果（TopDocs）的ID放到结果集缓存中并返回，这里可以比较一下数据库检索：如果是一个10,000条的数据库检索结果集，数据库是一定要把所有记录内容都取得以后再开始返回给应用结果集的。所以即使检索匹配总数很多，Lucene的结果集占用的内存空间也不会很多。对于一般的模糊检索应用是用不到这么多的结果的，头100条已经可以满足90%以上的检索需求。**

**如果首批缓存结果数用完后还要读取更后面的结果时Searcher会再次检索并生成一个上次的搜索缓存数大1倍的缓存，并再重新向后抓取。所以如果构造一个Searcher去查1－120条结果，Searcher其实是进行了2次搜索过程：头100条取完后，缓存结果用完，Searcher重新检索再构造一个200条的结果缓存，依此类推，400条缓存，800条缓存。由于每次Searcher对象消失后，这些缓存也访问那不到了，你有可能想将结果记录缓存下来，缓存数尽量保证在100以下以充分利用首次的结果缓存，不让Lucene浪费多次检索，而且可以分级进行结果缓存。**

**Lucene的另外一个特点是在收集结果的过程中将匹配度低的结果自动过滤掉了。这也是和数据库应用需要将搜索的结果全部返回不同之处。**

[**我的一些尝试**](http://sourceforge.net/projects/weblucene/)**：**

* **支持中文的Tokenizer：这里有2个版本，一个是通过JavaCC生成的，对CJK部分按一个字符一个TOKEN索引，另外一个是从SimpleTokenizer改写的，对英文支持数字和字母TOKEN，对中文按迭代索引。**
* **基于XML数据源的索引器：XMLIndexer，因此所有数据源只要能够按照DTD转换成指定的XML，就可以用XMLIndxer进行索引了。**
* **根据某个字段排序：按记录索引顺序排序结果的搜索器：IndexOrderSearcher，因此如果需要让搜索结果根据某个字段排序，可以让数据源先按某个字段排好序（比如：PriceField），这样索引后，然后在利用这个按记录的ID顺序检索的搜索器，结果就是相当于是那个字段排序的结果了。**

**从Lucene学到更多**

**Luene的确是一个面对对象设计的典范**

* **所有的问题都通过一个额外抽象层来方便以后的扩展和重用：你可以通过重新实现来达到自己的目的，而对其他模块而不需要；**
* **简单的应用入口Searcher, Indexer，并调用底层一系列组件协同的完成搜索任务；**
* **所有的对象的任务都非常专一：比如搜索过程：QueryParser分析将查询语句转换成一系列的精确查询的组合(Query),通过底层的索引读取结构IndexReader进行索引的读取，并用相应的打分器给搜索结果进行打分/排序等。所有的功能模块原子化程度非常高，因此可以通过重新实现而不需要修改其他模块。**
* **除了灵活的应用接口设计，Lucene还提供了一些适合大多数应用的语言分析器实现（SimpleAnalyser,StandardAnalyser），这也是新用户能够很快上手的重要原因之一。**

**这些优点都是非常值得在以后的开发中学习借鉴的。作为一个通用工具包，Lunece的确给予了需要将全文检索功能嵌入到应用中的开发者很多的便利。**

**此外，通过对Lucene的学习和使用，我也更深刻地理解了为什么很多数据库优化设计中要求，比如：**

* **尽可能对字段进行索引来提高查询速度，但过多的索引会对数据库表的更新操作变慢，而对结果过多的排序条件，实际上往往也是性能的杀手之一。**
* **很多商业数据库对大批量的数据插入操作会提供一些优化参数，这个作用和索引器的merge\_factor的作用是类似的，**
* **20%/80%原则：查的结果多并不等于质量好，尤其对于返回结果集很大，如何优化这头几十条结果的质量往往才是最重要的。**
* **尽可能让应用从数据库中获得比较小的结果集，因为即使对于大型数据库，对结果集的随机访问也是一个非常消耗资源的操作。**

**参考资料：**

**Apache: Lucene Project**[**http://jakarta.apache.org/lucene/**](http://jakarta.apache.org/lucene/)**Lucene开发/用户邮件列表归档**[**Lucene-dev@jakarta.apache.org**](http://www.mail-archive.com/lucene-dev@jakarta.apache.org/)[**Lucene-user@jakarta.apache.org**](http://www.mail-archive.com/lucene-user@jakarta.apache.org/)

**The Lucene search engine: Powerful, flexible, and free**[**http://www.javaworld.com/javaworld/jw-09-2000/jw-0915-Lucene\_p.html**](http://www.javaworld.com/javaworld/jw-09-2000/jw-0915-lucene_p.html)

**Lucene Tutorial**[**http://www.darksleep.com/puff/lucene/lucene.html**](http://www.darksleep.com/puff/lucene/lucene.html)

**Notes on distributed searching with Lucene**[**http://home.clara.net/markharwood/lucene/**](http://home.clara.net/markharwood/lucene/)

**中文语言的切分词**[**http://www.google.com/search?sourceid=navclient&hl=zh-CN&q=chinese+word+segment**](http://www.google.com/search?sourceid=navclient&hl=zh-CN&q=chinese+word+segment)

**搜索引擎工具介绍** [**http://searchtools.com/**](http://searchtools.com/)

**Lucene作者Cutting的几篇论文和专利**[**http://lucene.sourceforge.net/publications.html**](http://lucene.sourceforge.net/publications.html)

**Lucene的.NET实现：dotLucene**[**http://sourceforge.net/projects/dotlucene/**](http://sourceforge.net/projects/dotlucene/)

**Lucene作者Cutting的另外一个项目：基于Java的搜索引擎Nutch**[**http://www.nutch.org/**](http://www.nutch.org/)[**http://sourceforge.net/projects/nutch/**](http://sourceforge.net/projects/nutch/)

**关于基于词表和N-Gram的切分词比较**[**http://china.nikkeibp.co.jp/cgi-bin/china/news/int/int200302100112.html**](http://china.nikkeibp.co.jp/cgi-bin/china/news/int/int200302100112.html) **2005-01-08** [**Cutting在Pisa大学做的关于Lucene的讲座：非常详细的Lucene架构解说**](http://lucene.sourceforge.net/talks/pisa/)

**特别感谢：**[**前网易CTO许良杰(Jack Xu)**](http://www.google.com/search?q=%22Jack+Xu%22+Excite)**给我的指导：是您将我带入了搜索引擎这个行业。**