[Lucene学习总结之一：全文检索的基本原理](http://blog.csdn.net/forfuture1978/article/details/4711308)

**一、总论**

根据<http://lucene.apache.org/java/docs/index.html> 定义：

**Lucene** **是一个高效的，基于Java** **的全文检索库。**

所以在了解Lucene之前要费一番工夫了解一下全文检索。

那么什么叫做全文检索呢？这要从我们生活中的数据说起。

我们生活中的数据总体分为两种：**结构化数据** 和**非结构化数据** 。

* **结构化数据：** 指具有固定格式或有限长度的数据，如数据库，元数据等。
* **非结构化数据：** 指不定长或无固定格式的数据，如邮件，word文档等。

当然有的地方还会提到第三种，半结构化数据，如XML，HTML等，当根据需要可按结构化数据来处理，也可抽取出纯文本按非结构化数据来处理。

**非结构化数据又一种叫法叫全文数据。**

按照数据的分类，搜索也分为两种：

* **对结构化数据的搜索** ：如对数据库的搜索，用SQL语句。再如对元数据的搜索，如利用windows搜索对文件名，类型，修改时间进行搜索等。
* **对非结构化数据的搜索** ：如利用windows的搜索也可以搜索文件内容，Linux下的grep命令，再如用Google和百度可以搜索大量内容数据。

对非结构化数据也即对全文数据的搜索主要有两种方法：

一种是**顺序扫描法** **(Serial Scanning)：** 所谓顺序扫描，比如要找内容包含某一个字符串的文件，就是一个文档一个文档的看，对于每一个文档，从头看到尾，如果此文档包含此字符串，则此文档为我们要找的文件，接着看下一个文件，直到扫描完所有的文件。如利用windows的搜索也可以搜索文件内容，只是相当的慢。如果你有一个80G硬盘，如果想在上面找到一个内容包含某字符串的文件，不花他几个小时，怕是做不到。Linux下的grep命令也是这一种方式。大家可能觉得这种方法比较原始，但对于小数据量的文件，这种方法还是最直接，最方便的。但是对于大量的文件，这种方法就很慢了。

有人可能会说，对非结构化数据顺序扫描很慢，对结构化数据的搜索却相对较快（由于结构化数据有一定的结构可以采取一定的搜索算法加快速度），那么把我们的非结构化数据想办法弄得有一定结构不就行了吗？

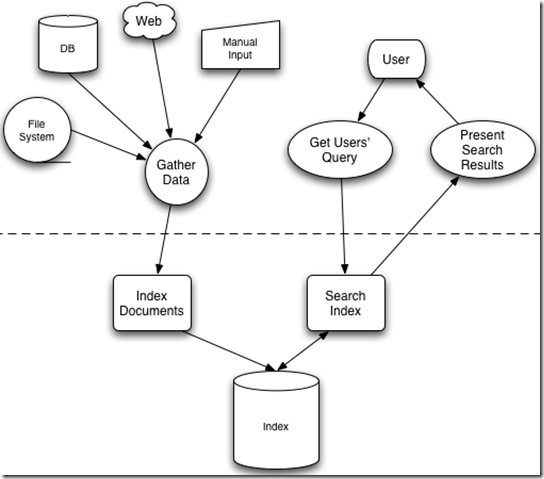
这种想法很天然，却构成了全文检索的基本思路，也即将非结构化数据中的一部分信息提取出来，重新组织，使其变得有一定结构，然后对此有一定结构的数据进行搜索，从而达到搜索相对较快的目的。

这部分从非结构化数据中提取出的然后重新组织的信息，我们称之**索引** 。

这种说法比较抽象，举几个例子就很容易明白，比如字典，字典的拼音表和部首检字表就相当于字典的索引，对每一个字的解释是非结构化的，如果字典没有音节表和部首检字表，在茫茫辞海中找一个字只能顺序扫描。然而字的某些信息可以提取出来进行结构化处理，比如读音，就比较结构化，分声母和韵母，分别只有几种可以一一列举，于是将读音拿出来按一定的顺序排列，每一项读音都指向此字的详细解释的页数。我们搜索时按结构化的拼音搜到读音，然后按其指向的页数，便可找到我们的非结构化数据——也即对字的解释。

**这种先建立索引，再对索引进行搜索的过程就叫全文检索(Full-text Search)** **。**

下面这幅图来自《Lucene in action》，但却不仅仅描述了Lucene的检索过程，而是描述了全文检索的一般过程。



全文检索大体分两个过程，**索引创建** **(Indexing)** 和**搜索索引** **(Search)** 。

* 索引创建：将现实世界中所有的结构化和非结构化数据提取信息，创建索引的过程。
* 搜索索引：就是得到用户的查询请求，搜索创建的索引，然后返回结果的过程。

于是全文检索就存在三个重要问题：

**1. 索引里面究竟存些什么？(Index)**

**2. 如何创建索引？(Indexing)**

**3. 如何对索引进行搜索？(Search)**

下面我们顺序对每个个问题进行研究。

**二、索引里面究竟存些什么**

索引里面究竟需要存些什么呢？

首先我们来看为什么顺序扫描的速度慢：

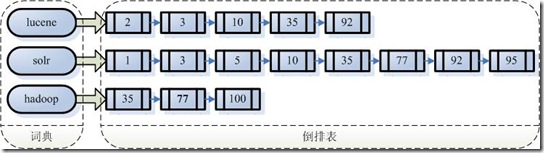
其实是由于我们想要搜索的信息和非结构化数据中所存储的信息不一致造成的。

非结构化数据中所存储的信息是每个文件包含哪些字符串，也即已知文件，欲求字符串相对容易，也即是从文件到字符串的映射。而我们想搜索的信息是哪些文件包含此字符串，也即已知字符串，欲求文件，也即从字符串到文件的映射。两者恰恰相反。于是如果索引总能够保存从字符串到文件的映射，则会大大提高搜索速度。

由于从字符串到文件的映射是文件到字符串映射的反向过程，于是保存这种信息的索引称为**反向索引** 。

反向索引的所保存的信息一般如下：

假设我的文档集合里面有100篇文档，为了方便表示，我们为文档编号从1到100，得到下面的结构



左边保存的是一系列字符串，称为**词典** 。

每个字符串都指向包含此字符串的文档(Document)链表，此文档链表称为**倒排表** (Posting List)。

有了索引，便使保存的信息和要搜索的信息一致，可以大大加快搜索的速度。

比如说，我们要寻找既包含字符串“lucene”又包含字符串“solr”的文档，我们只需要以下几步：

1. 取出包含字符串“lucene”的文档链表。

2. 取出包含字符串“solr”的文档链表。

3. 通过合并链表，找出既包含“lucene”又包含“solr”的文件。

[图]倒排表合并过程

看到这个地方，有人可能会说，全文检索的确加快了搜索的速度，但是多了索引的过程，两者加起来不一定比顺序扫描快多少。的确，加上索引的过程，全文检索不一定比顺序扫描快，尤其是在数据量小的时候更是如此。而对一个很大量的数据创建索引也是一个很慢的过程。

然而两者还是有区别的，顺序扫描是每次都要扫描，而创建索引的过程仅仅需要一次，以后便是一劳永逸的了，每次搜索，创建索引的过程不必经过，仅仅搜索创建好的索引就可以了。

**这也是全文搜索相对于顺序扫描的优势之一：一次索引，多次使用。**

**三、如何创建索引**

全文检索的索引创建过程一般有以下几步：

**第一步：一些要索引的原文档(Document)。**

为了方便说明索引创建过程，这里特意用两个文件为例：

文件一：Students should be allowed to go out with their friends, but not allowed to drink beer.

文件二：My friend Jerry went to school to see his students but found them drunk which is not allowed.

**第二步：将原文档传给分次组件(Tokenizer)。**

**分词组件(Tokenizer)会做以下几件事情(** **此过程称为Tokenize)** **：**

**1. 将文档分成一个一个单独的单词。**

**2. 去除标点符号。**

**3. 去除停词(Stop word)** **。**

所谓停词(Stop word)就是一种语言中最普通的一些单词，由于没有特别的意义，因而大多数情况下不能成为搜索的关键词，因而创建索引时，这种词会被去掉而减少索引的大小。

英语中挺词(Stop word)如：“the”,“a”，“this”等。

对于每一种语言的分词组件(Tokenizer)，都有一个停词(stop word)集合。

**经过分词(Tokenizer)** **后得到的结果称为词元(Token)** **。**

在我们的例子中，便得到以下词元(Token)：

“Students”，“allowed”，“go”，“their”，“friends”，“allowed”，“drink”，“beer”，“My”，“friend”，“Jerry”，“went”，“school”，“see”，“his”，“students”，“found”，“them”，“drunk”，“allowed”。

**第三步：将得到的词元(Token)传给语言处理组件(Linguistic Processor)。**

语言处理组件(linguistic processor)主要是对得到的词元(Token)做一些同语言相关的处理。

**对于英语，语言处理组件(Linguistic Processor)** **一般做以下几点：**

**1. 变为小写(Lowercase)** **。**

**2. 将单词缩减为词根形式，如“cars** **”到“car** **”等。这种操作称为：stemming** **。**

**3. 将单词转变为词根形式，如“drove** **”到“drive** **”等。这种操作称为：lemmatization** **。**

**Stemming 和 lemmatization的异同：**

* 相同之处：Stemming和lemmatization都要使词汇成为词根形式。
* 两者的方式不同：
  + Stemming采用的是“缩减”的方式：“cars”到“car”，“driving”到“drive”。
  + Lemmatization采用的是“转变”的方式：“drove”到“drove”，“driving”到“drive”。
* 两者的算法不同：
  + Stemming主要是采取某种固定的算法来做这种缩减，如去除“s”，去除“ing”加“e”，将“ational”变为“ate”，将“tional”变为“tion”。
  + Lemmatization主要是采用保存某种字典的方式做这种转变。比如字典中有“driving”到“drive”，“drove”到“drive”，“am, is, are”到“be”的映射，做转变时，只要查字典就可以了。
* Stemming和lemmatization不是互斥关系，是有交集的，有的词利用这两种方式都能达到相同的转换。

**语言处理组件(linguistic processor)的结果称为词(Term)** **。**

在我们的例子中，经过语言处理，得到的词(Term)如下：

“student”，“allow”，“go”，“their”，“friend”，“allow”，“drink”，“beer”，“my”，“friend”，“jerry”，“go”，“school”，“see”，“his”，“student”，“find”，“them”，“drink”，“allow”。

也正是因为有语言处理的步骤，才能使搜索drove，而drive也能被搜索出来。

**第四步：将得到的词(Term)传给索引组件(Indexer)。**

**索引** **组件(Indexer)主要做以下几件事情：**

**1. 利用得到的词(Term)创建一个字典。**

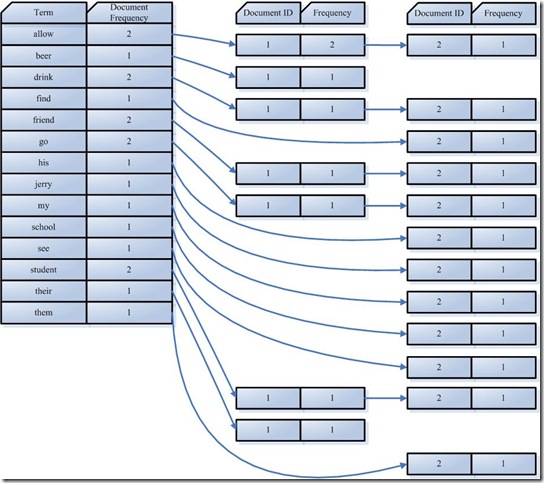
在我们的例子中字典如下：

|  |  |
| --- | --- |
| Term | Document ID |
| student | 1 |
| allow | 1 |
| go | 1 |
| their | 1 |
| friend | 1 |
| allow | 1 |
| drink | 1 |
| beer | 1 |
| my | 2 |
| friend | 2 |
| jerry | 2 |
| go | 2 |
| school | 2 |
| see | 2 |
| his | 2 |
| student | 2 |
| find | 2 |
| them | 2 |
| drink | 2 |
| allow | 2 |

**2. 对字典按字母顺序进行排序。**

|  |  |
| --- | --- |
| Term | Document ID |
| allow | 1 |
| allow | 1 |
| allow | 2 |
| beer | 1 |
| drink | 1 |
| drink | 2 |
| find | 2 |
| friend | 1 |
| friend | 2 |
| go | 1 |
| go | 2 |
| his | 2 |
| jerry | 2 |
| my | 2 |
| school | 2 |
| see | 2 |
| student | 1 |
| student | 2 |
| their | 1 |
| them | 2 |

**3. 合并相同的词(Term)** **成为文档倒排(Posting List)** **链表。**



在此表中，有几个定义：

* Document Frequency 即文档频次，表示总共有多少文件包含此词(Term)。
* Frequency 即词频率，表示此文件中包含了几个此词(Term)。

所以对词(Term) “allow”来讲，总共有两篇文档包含此词(Term)，从而词(Term)后面的文档链表总共有两项，第一项表示包含“allow”的第一篇文档，即1号文档，此文档中，“allow”出现了2次，第二项表示包含“allow”的第二个文档，是2号文档，此文档中，“allow”出现了1次。

到此为止，索引已经创建好了，我们可以通过它很快的找到我们想要的文档。

而且在此过程中，我们惊喜地发现，搜索“drive”，“driving”，“drove”，“driven”也能够被搜到。因为在我们的索引中，“driving”，“drove”，“driven”都会经过语言处理而变成“drive”，在搜索时，如果您输入“driving”，输入的查询语句同样经过我们这里的一到三步，从而变为查询“drive”，从而可以搜索到想要的文档。

**三、如何对索引进行搜索？**

到这里似乎我们可以宣布“我们找到想要的文档了”。

然而事情并没有结束，找到了仅仅是全文检索的一个方面。不是吗？如果仅仅只有一个或十个文档包含我们查询的字符串，我们的确找到了。然而如果结果有一千个，甚至成千上万个呢？那个又是您最想要的文件呢？

打开Google吧，比如说您想在微软找份工作，于是您输入“Microsoft job”，您却发现总共有22600000个结果返回。好大的数字呀，突然发现找不到是一个问题，找到的太多也是一个问题。在如此多的结果中，如何将最相关的放在最前面呢？



当然Google做的很不错，您一下就找到了jobs at Microsoft。想象一下，如果前几个全部是“Microsoft does a good job at software industry…”将是多么可怕的事情呀。

如何像Google一样，在成千上万的搜索结果中，找到和查询语句最相关的呢？

如何判断搜索出的文档和查询语句的相关性呢？

这要回到我们第三个问题：如何对索引进行搜索？

搜索主要分为以下几步：

**第一步：用户输入查询语句。**

查询语句同我们普通的语言一样，也是有一定语法的。

不同的查询语句有不同的语法，如SQL语句就有一定的语法。

查询语句的语法根据全文检索系统的实现而不同。最基本的有比如：AND, OR, NOT等。

举个例子，用户输入语句：lucene AND learned NOT hadoop。

说明用户想找一个包含lucene和learned然而不包括hadoop的文档。

**第二步：对查询语句进行词法分析，语法分析，及语言处理。**

由于查询语句有语法，因而也要进行语法分析，语法分析及语言处理。

**1. 词法分析主要用来识别单词和关键字。**

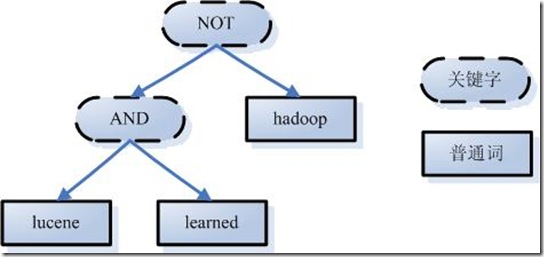
如上述例子中，经过词法分析，得到单词有lucene，learned，hadoop, 关键字有AND, NOT。

如果在词法分析中发现不合法的关键字，则会出现错误。如lucene AMD learned，其中由于AND拼错，导致AMD作为一个普通的单词参与查询。

**2. 语法分析主要是根据查询语句的语法规则来形成一棵语法树。**

如果发现查询语句不满足语法规则，则会报错。如lucene NOT AND learned，则会出错。

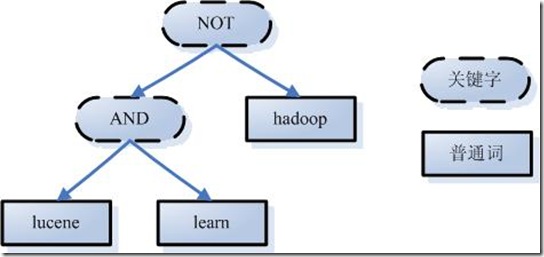
如上述例子，lucene AND learned NOT hadoop形成的语法树如下：



**3. 语言处理同索引过程中的语言处理几乎相同。**

如learned变成learn等。

经过第二步，我们得到一棵经过语言处理的语法树。



**第三步：搜索索引，得到符合语法树的文档。**

此步骤有分几小步：

1. 首先，在反向索引表中，分别找出包含lucene，learn，hadoop的文档链表。
2. 其次，对包含lucene，learn的链表进行合并操作，得到既包含lucene又包含learn的文档链表。
3. 然后，将此链表与hadoop的文档链表进行差操作，去除包含hadoop的文档，从而得到既包含lucene又包含learn而且不包含hadoop的文档链表。
4. 此文档链表就是我们要找的文档。

**第四步：根据得到的文档和查询语句的相关性，对结果进行排序。**

虽然在上一步，我们得到了想要的文档，然而对于查询结果应该按照与查询语句的相关性进行排序，越相关者越靠前。

如何计算文档和查询语句的相关性呢？

不如我们把查询语句看作一片短小的文档，对文档与文档之间的相关性(relevance)进行打分(scoring)，分数高的相关性好，就应该排在前面。

那么又怎么对文档之间的关系进行打分呢？

**这可不是一件容易的事情，首先我们看一看判断人之间的关系吧。**

**首先** 看一个人，往往有很多**要素** ，如性格，信仰，爱好，衣着，高矮，胖瘦等等。

**其次** 对于人与人之间的关系，**不同的要素重要性不同** ，性格，信仰，爱好可能重要些，衣着，高矮，胖瘦可能就不那么重要了，所以具有相同或相似性格，信仰，爱好的人比较容易成为好的朋友，然而衣着，高矮，胖瘦不同的人，也可以成为好的朋友。

因而判断人与人之间的关系，**首先要找出哪些要素对人与人之间的关系最重要** ，比如性格，信仰，爱好。**其次要判断两个人的这些要素之间的关系** ，比如一个人性格开朗，另一个人性格外向，一个人信仰佛教，另一个信仰上帝，一个人爱好打篮球，另一个爱好踢足球。我们发现，两个人在性格方面都很积极，信仰方面都很善良，爱好方面都爱运动，因而两个人关系应该会很好。

**我们再来看看公司之间的关系吧。**

**首先** 看一个公司，有很多人组成，如总经理，经理，首席技术官，普通员工，保安，门卫等。

**其次对于公司与公司之间的关系，不同的人重要性不同** ，总经理，经理，首席技术官可能更重要一些，普通员工，保安，门卫可能较不重要一点。所以如果两个公司总经理，经理，首席技术官之间关系比较好，两个公司容易有比较好的关系。然而一位普通员工就算与另一家公司的一位普通员工有血海深仇，怕也难影响两个公司之间的关系。

因而判断公司与公司之间的关系，**首先要找出哪些人对公司与公司之间的关系最重要** ，比如总经理，经理，首席技术官。**其次要判断这些人之间的关系** ，不如两家公司的总经理曾经是同学，经理是老乡，首席技术官曾是创业伙伴。我们发现，两家公司无论总经理，经理，首席技术官，关系都很好，因而两家公司关系应该会很好。

分析了两种关系，下面看一下**如何判断文档之间的关系** 了。

**首先，一个文档有很多词(Term)组成** ，如search, lucene, full-text, this, a, what等。

**其次对于文档之间的关系，不同的Term重要性不同** ，比如对于本篇文档，search, Lucene, full-text就相对重要一些，this, a , what可能相对不重要一些。所以如果两篇文档都包含search, Lucene，fulltext，这两篇文档的相关性好一些，然而就算一篇文档包含this, a, what，另一篇文档不包含this, a, what，也不能影响两篇文档的相关性。

因而判断文档之间的关系，首先找出哪些词(Term)对文档之间的关系最重要，如search, Lucene, fulltext。然后判断这些词(Term)之间的关系。

**找出词(Term)** **对文档的重要性的过程称为计算词的权重(Term weight)** **的过程。**

计算词的权重(term weight)有两个参数，第一个是词(Term)，第二个是文档(Document)。

词的权重(Term weight)表示此词(Term)在此文档中的重要程度，越重要的词(Term)有越大的权重(Term weight)，因而在计算文档之间的相关性中将发挥更大的作用。

**判断词(Term)** **之间的关系从而得到文档相关性的过程应用一种叫做向量空间模型的算法(Vector Space Model)** **。**

下面仔细分析一下这两个过程：

**1. 计算权重(Term weight)的过程。**

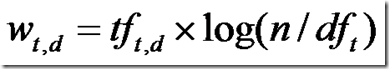
影响一个词(Term)在一篇文档中的重要性主要有两个因素：

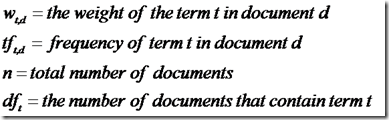
* Term Frequency (tf)：即此Term在此文档中出现了多少次。tf 越大说明越重要。
* Document Frequency (df)：即有多少文档包含次Term。df 越大说明越不重要。

容易理解吗？词(Term)在文档中出现的次数越多，说明此词(Term)对该文档越重要，如“搜索”这个词，在本文档中出现的次数很多，说明本文档主要就是讲这方面的事的。然而在一篇英语文档中，this出现的次数更多，就说明越重要吗？不是的，这是由第二个因素进行调整，第二个因素说明，有越多的文档包含此词(Term), 说明此词(Term)太普通，不足以区分这些文档，因而重要性越低。

这也如我们程序员所学的技术，对于程序员本身来说，这项技术掌握越深越好（掌握越深说明花时间看的越多，tf越大），找工作时越有竞争力。然而对于所有程序员来说，这项技术懂得的人越少越好（懂得的人少df小），找工作越有竞争力。人的价值在于不可替代性就是这个道理。

道理明白了，我们来看看公式：





这仅仅只term weight计算公式的简单典型实现。实现全文检索系统的人会有自己的实现，Lucene就与此稍有不同。

**2. 判断Term之间的关系从而得到文档相关性的过程，也即向量空间模型的算法(VSM)。**

我们把文档看作一系列词(Term)，每一个词(Term)都有一个权重(Term weight)，不同的词(Term)根据自己在文档中的权重来影响文档相关性的打分计算。

于是我们把所有此文档中词(term)的权重(term weight) 看作一个向量。

Document = {term1, term2, …… ,term N}

Document Vector = {weight1, weight2, …… ,weight N}

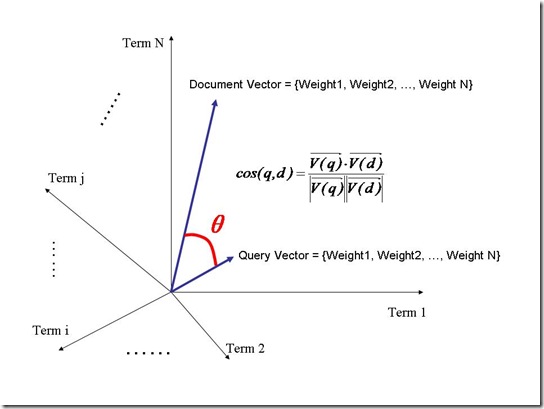
同样我们把查询语句看作一个简单的文档，也用向量来表示。

Query = {term1, term 2, …… , term N}

Query Vector = {weight1, weight2, …… , weight N}

我们把所有搜索出的文档向量及查询向量放到一个N维空间中，每个词(term)是一维。

如图：



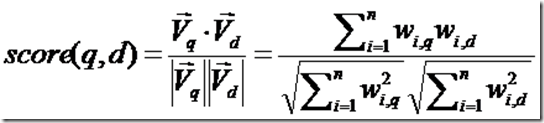
我们认为两个向量之间的夹角越小，相关性越大。

所以我们计算夹角的余弦值作为相关性的打分，夹角越小，余弦值越大，打分越高，相关性越大。

有人可能会问，查询语句一般是很短的，包含的词(Term)是很少的，因而查询向量的维数很小，而文档很长，包含词(Term)很多，文档向量维数很大。你的图中两者维数怎么都是N呢？

在这里，既然要放到相同的向量空间，自然维数是相同的，不同时，取二者的并集，如果不含某个词(Term)时，则权重(Term Weight)为0。

相关性打分公式如下：



举个例子，查询语句有11个Term，共有三篇文档搜索出来。其中各自的权重(Term weight)，如下表格。

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | t1 | t2 | t3 | t4 | t5 | t6 | t7 | t8 | t9 | t10 | t11 |
| D1 | 0 | 0 | .477 | 0 | .477 | .176 | 0 | 0 | 0 | .176 | 0 |
| D2 | 0 | .176 | 0 | .477 | 0 | 0 | 0 | 0 | .954 | 0 | .176 |
| D3 | 0 | .176 | 0 | 0 | 0 | .176 | 0 | 0 | 0 | .176 | .176 |
| Q | 0 | 0 | 0 | 0 | 0 | .176 | 0 | 0 | .477 | 0 | .176 |

于是计算，三篇文档同查询语句的相关性打分分别为：

[图]文档一的打分计算

[图]文档二的打分计算

[图]文档三的打分计算

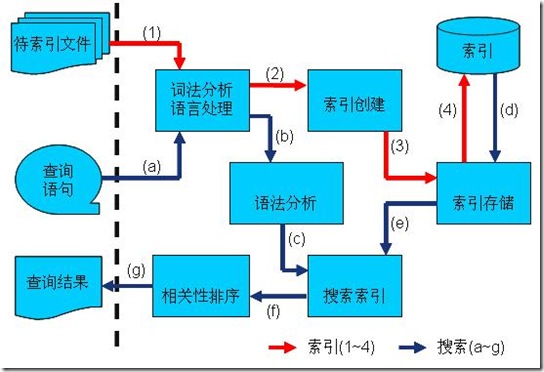
于是文档二相关性最高，先返回，其次是文档一，最后是文档三。

到此为止，我们可以找到我们最想要的文档了。

说了这么多，其实还没有进入到Lucene，而仅仅是信息检索技术(Information retrieval)中的基本理论，然而当我们看过Lucene后我们会发现，Lucene是对这种基本理论的一种基本的的实践。所以在以后分析Lucene的文章中，会常常看到以上理论在Lucene中的应用。

在进入Lucene之前，对上述索引创建和搜索过程所一个总结，如图：

此图参照<http://www.lucene.com.cn/about.htm> 中文章《开放源代码的全文检索引擎Lucene》



**1. 索引过程：**

**1) 有一系列被索引文件**

**2) 被索引文件经过语法分析和语言处理形成一系列词(Term)** **。**

**3) 经过索引创建形成词典和反向索引表。**

**4) 通过索引存储将索引写入硬盘。**

**2. 搜索过程：**

**a) 用户输入查询语句。**

**b) 对查询语句经过语法分析和语言分析得到一系列词(Term)** **。**

**c) 通过语法分析得到一个查询树。**

**d) 通过索引存储将索引读入到内存。**

**e) 利用查询树搜索索引，从而得到每个词(Term)** **的文档链表，对文档链表进行交，差，并得到结果文档。**

**f) 将搜索到的结果文档对查询的相关性进行排序。**

**g) 返回查询结果给用户。**

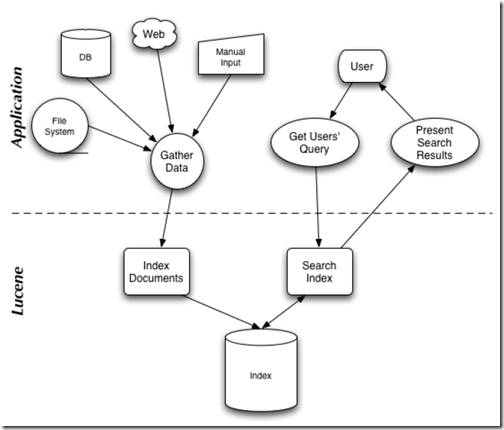
下面我们可以进入Lucene的世界了。

[Lucene学习总结之二：Lucene的总体架构](http://blog.csdn.net/forfuture1978/article/details/4745802)

Lucene总的来说是：

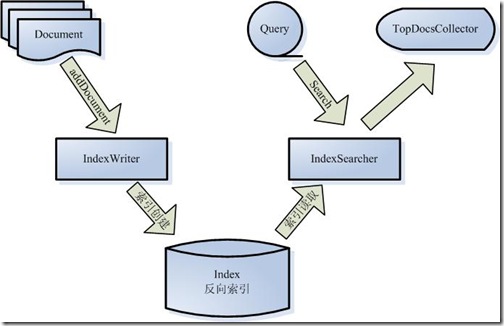
* 一个高效的，可扩展的，全文检索库。
* 全部用Java实现，无须配置。
* 仅支持纯文本文件的索引(Indexing)和搜索(Search)。
* 不负责由其他格式的文件抽取纯文本文件，或从网络中抓取文件的过程。

在Lucene in action中，Lucene 的构架和过程如下图，



**说明Lucene** **是有索引和搜索的两个过程，包含索引创建，索引，搜索三个要点。**

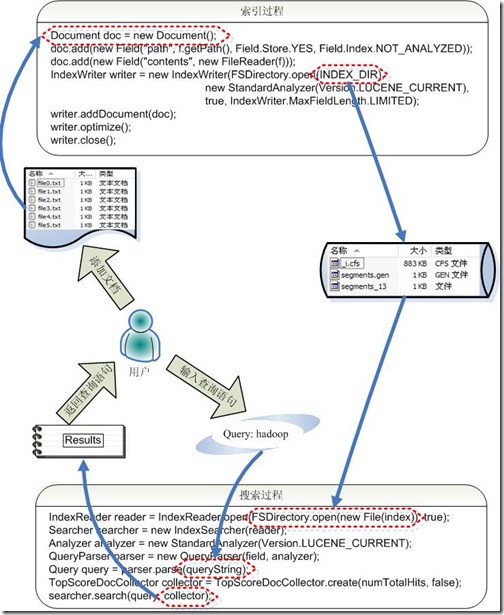
让我们更细一些看Lucene的各组件：



* **被索引的文档用Document对象** **表示。**
* **IndexWriter** **通过函数addDocument** **将文档添加到索引中，实现创建索引的过程。**
* **Lucene** **的索引是应用反向索引。**
* **当用户有请求时，Query** **代表用户的查询语句。**
* **IndexSearcher** **通过函数search** **搜索Lucene Index** **。**
* **IndexSearcher** **计算term weight** **和score** **并且将结果返回给用户。**
* **返回给用户的文档集合用TopDocsCollector** **表示。**

那么如何应用这些组件呢？

让我们再详细到对Lucene API 的调用实现索引和搜索过程。



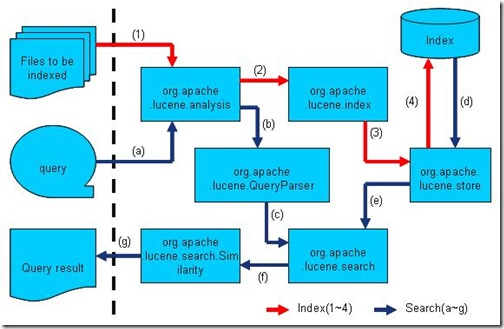
* **索引过程如下：**
  + **创建一个IndexWriter** **用来写索引文件，它有几个参数，INDEX\_DIR** **就是索引文件所存放的位置，Analyzer** **便是用来对文档进行词法分析和语言处理的。**
  + **创建一个Document** **代表我们要索引的文档。**
  + **将不同的Field** **加入到文档中。我们知道，一篇文档有多种信息，如题目，作者，修改时间，内容等。不同类型的信息用不同的Field** **来表示，在本例子中，一共有两类信息进行了索引，一个是文件路径，一个是文件内容。其中FileReader** **的SRC\_FILE** **就表示要索引的源文件。**
  + **IndexWriter** **调用函数addDocument** **将索引写到索引文件夹中。**
* **搜索过程如下：**
  + **IndexReader** **将磁盘上的索引信息读入到内存，INDEX\_DIR** **就是索引文件存放的位置。**
  + **创建IndexSearcher** **准备进行搜索。**
  + **创建Analyer** **用来对查询语句进行词法分析和语言处理。**
  + **创建QueryParser** **用来对查询语句进行语法分析。**
  + **QueryParser** **调用parser** **进行语法分析，形成查询语法树，放到Query** **中。**
  + **IndexSearcher** **调用search** **对查询语法树Query** **进行搜索，得到结果TopScoreDocCollector** **。**

以上便是Lucene API函数的简单调用。

然而当进入Lucene的源代码后，发现Lucene有很多包，关系错综复杂。

然而通过下图，我们不难发现，Lucene的各源码模块，都是对普通索引和搜索过程的一种实现。

此图是上一节介绍的全文检索的流程对应的Lucene实现的包结构。(参照<http://www.lucene.com.cn/about.htm> 中文章《开放源代码的全文检索引擎Lucene》)



* **Lucene** **的analysis** **模块主要负责词法分析及语言处理而形成Term** **。**
* **Lucene** **的index** **模块主要负责索引的创建，里面有IndexWriter** **。**
* **Lucene** **的store** **模块主要负责索引的读写。**
* **Lucene** **的QueryParser** **主要负责语法分析。**
* **Lucene** **的search** **模块主要负责对索引的搜索。**
* **Lucene** **的similarity** **模块主要负责对相关性打分的实现。**

了解了Lucene的整个结构，我们便可以开始Lucene的源码之旅了。

[Lucene学习总结之三：Lucene的索引文件格式(1)](http://blog.csdn.net/forfuture1978/article/details/4981893)

Lucene的索引里面存了些什么，如何存放的，也即Lucene的索引文件格式，是读懂Lucene源代码的一把钥匙。

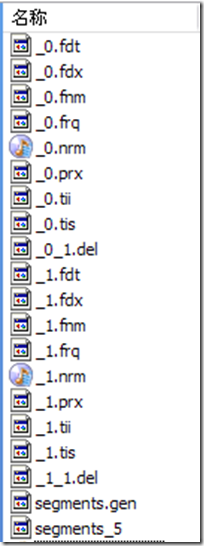
当我们真正进入到Lucene源代码之中的时候，我们会发现:

* Lucene的索引过程，就是按照全文检索的基本过程，将倒排表写成此文件格式的过程。
* Lucene的搜索过程，就是按照此文件格式将索引进去的信息读出来，然后计算每篇文档打分(score)的过程。

本文详细解读了Apache Lucene - Index File Formats(<http://lucene.apache.org/java/2_9_0/fileformats.html> ) 这篇文章。

**一、基本概念**

下图就是Lucene生成的索引的一个实例：



Lucene的索引结构是有层次结构的，主要分以下几个层次：

* 索引(Index)：
  + 在Lucene中一个索引是放在一个文件夹中的。
  + 如上图，同一文件夹中的所有的文件构成一个Lucene索引。
* 段(Segment)：
  + 一个索引可以包含多个段，段与段之间是独立的，添加新文档可以生成新的段，不同的段可以合并。
  + 如上图，具有相同前缀文件的属同一个段，图中共两个段 "\_0" 和 "\_1"。
  + segments.gen和segments\_5是段的元数据文件，也即它们保存了段的属性信息。
* 文档(Document)：
  + 文档是我们建索引的基本单位，不同的文档是保存在不同的段中的，一个段可以包含多篇文档。
  + 新添加的文档是单独保存在一个新生成的段中，随着段的合并，不同的文档合并到同一个段中。
* 域(Field)：
  + 一篇文档包含不同类型的信息，可以分开索引，比如标题，时间，正文，作者等，都可以保存在不同的域里。
  + 不同域的索引方式可以不同，在真正解析域的存储的时候，我们会详细解读。
* 词(Term)：
  + 词是索引的最小单位，是经过词法分析和语言处理后的字符串。

Lucene的索引结构中，即保存了正向信息，也保存了反向信息。

所谓正向信息：

* 按层次保存了从索引，一直到词的包含关系：索引(Index) –> 段(segment) –> 文档(Document) –> 域(Field) –> 词(Term)
* 也即此索引包含了那些段，每个段包含了那些文档，每个文档包含了那些域，每个域包含了那些词。
* 既然是层次结构，则每个层次都保存了本层次的信息以及下一层次的元信息，也即属性信息，比如一本介绍中国地理的书，应该首先介绍中国地理的概况，以及中国包含多少个省，每个省介绍本省的基本概况及包含多少个市，每个市介绍本市的基本概况及包含多少个县，每个县具体介绍每个县的具体情况。
* 如上图，包含正向信息的文件有：
  + segments\_N保存了此索引包含多少个段，每个段包含多少篇文档。
  + XXX.fnm保存了此段包含了多少个域，每个域的名称及索引方式。
  + XXX.fdx，XXX.fdt保存了此段包含的所有文档，每篇文档包含了多少域，每个域保存了那些信息。
  + XXX.tvx，XXX.tvd，XXX.tvf保存了此段包含多少文档，每篇文档包含了多少域，每个域包含了多少词，每个词的字符串，位置等信息。

所谓反向信息：

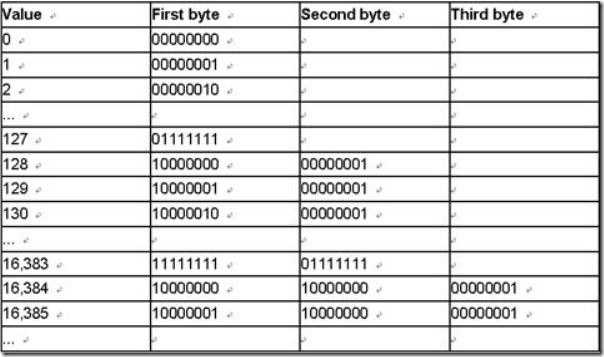
* 保存了词典到倒排表的映射：词(Term) –> 文档(Document)
* 如上图，包含反向信息的文件有：
  + XXX.tis，XXX.tii保存了词典(Term Dictionary)，也即此段包含的所有的词按字典顺序的排序。
  + XXX.frq保存了倒排表，也即包含每个词的文档ID列表。
  + XXX.prx保存了倒排表中每个词在包含此词的文档中的位置。

在了解Lucene索引的详细结构之前，先看看Lucene索引中的基本数据类型。

**二、基本类型**

Lucene索引文件中，用一下基本类型来保存信息：

* Byte：是最基本的类型，长8位(bit)。
* UInt32：由4个Byte组成。
* UInt64：由8个Byte组成。
* VInt：
  + 变长的整数类型，它可能包含多个Byte，对于每个Byte的8位，其中后7位表示数值，最高1位表示是否还有另一个Byte，0表示没有，1表示有。
  + 越前面的Byte表示数值的低位，越后面的Byte表示数值的高位。
  + 例如130化为二进制为 1000, 0010，总共需要8位，一个Byte表示不了，因而需要两个Byte来表示，第一个Byte表示后7位，并且在最高位置1来表示后面还有一个Byte，所以为(1) 0000010，第二个Byte表示第8位，并且最高位置0来表示后面没有其他的Byte了，所以为(0) 0000001。



* Chars：是UTF-8编码的一系列Byte。
* String：一个字符串首先是一个VInt来表示此字符串包含的字符的个数，接着便是UTF-8编码的字符序列Chars。

**三、基本规则**

Lucene为了使的信息的存储占用的空间更小，访问速度更快，采取了一些特殊的技巧，然而在看Lucene文件格式的时候，这些技巧却容易使我们感到困惑，所以有必要把这些特殊的技巧规则提取出来介绍一下。

在下不才，胡乱给这些规则起了一些名字，是为了方便后面应用这些规则的时候能够简单，不妥之处请大家谅解。

**1. 前缀后缀规则(Prefix+Suffix)**

Lucene在反向索引中，要保存词典(Term Dictionary)的信息，所有的词(Term)在词典中是按照字典顺序进行排列的，然而词典中包含了文档中的几乎所有的词，并且有的词还是非常的长的，这样索引文件会非常的大，所谓前缀后缀规则，即当某个词和前一个词有共同的前缀的时候，后面的词仅仅保存前缀在词中的偏移(offset)，以及除前缀以外的字符串(称为后缀)。



比如要存储如下词:term，termagancy，termagant，terminal，

如果按照正常方式来存储，需要的空间如下：

[VInt = 4] [t][e][r][m]，[VInt = 10][t][e][r][m][a][g][a][n][c][y]，[VInt = 9][t][e][r][m][a][g][a][n][t]，[VInt = 8][t][e][r][m][i][n][a][l]

共需要35个Byte.

如果应用前缀后缀规则，需要的空间如下：

[VInt = 4] [t][e][r][m]，[VInt = 4 (offset)][VInt = 6][a][g][a][n][c][y]，[VInt = 8 (offset)][VInt = 1][t]，[VInt = 4(offset)][VInt = 4][i][n][a][l]

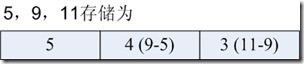
共需要22个Byte。

大大缩小了存储空间，尤其是在按字典顺序排序的情况下，前缀的重合率大大提高。

**2. 差值规则(Delta)**

在Lucene的反向索引中，需要保存很多整型数字的信息，比如文档ID号，比如词(Term)在文档中的位置等等。

由上面介绍，我们知道，整型数字是以VInt的格式存储的。随着数值的增大，每个数字占用的Byte的个数也逐渐的增多。所谓差值规则(Delta)就是先后保存两个整数的时候，后面的整数仅仅保存和前面整数的差即可。



比如要存储如下整数：16386，16387，16388，16389

如果按照正常方式来存储，需要的空间如下：

[(1) 000, 0010][(1) 000, 0000][(0) 000, 0001]，[(1) 000, 0011][(1) 000, 0000][(0) 000, 0001]，[(1) 000, 0100][(1) 000, 0000][(0) 000, 0001]，[(1) 000, 0101][(1) 000, 0000][(0) 000, 0001]

供需12个Byte。

如果应用差值规则来存储，需要的空间如下：

[(1) 000, 0010][(1) 000, 0000][(0) 000, 0001]，[(0) 000, 0001]，[(0) 000, 0001]，[(0) 000, 0001]

共需6个Byte。

大大缩小了存储空间，而且无论是文档ID，还是词在文档中的位置，都是按从小到大的顺序，逐渐增大的。

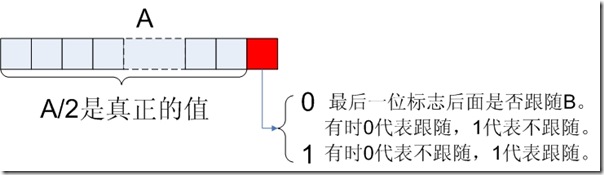
**3. 或然跟随规则(A, B?)**

Lucene的索引结构中存在这样的情况，某个值A后面可能存在某个值B，也可能不存在，需要一个标志来表示后面是否跟随着B。

一般的情况下，在A后面放置一个Byte，为0则后面不存在B，为1则后面存在B，或者0则后面存在B，1则后面不存在B。

但这样要浪费一个Byte的空间，其实一个Bit就可以了。

在Lucene中，采取以下的方式：A的值左移一位，空出最后一位，作为标志位，来表示后面是否跟随B，所以在这种情况下，A/2是真正的A原来的值。



如果去读Apache Lucene - Index File Formats这篇文章，会发现很多符合这种规则的：

* .frq文件中的DocDelta[, Freq?]，DocSkip,PayloadLength?
* .prx文件中的PositionDelta,Payload? (但不完全是，如下表分析)

当然还有一些带?的但不属于此规则的：

* .frq文件中的SkipChildLevelPointer?，是多层跳跃表中，指向下一层表的指针，当然如果是最后一层，此值就不存在，也不需要标志。
* .tvf文件中的Positions?, Offsets?。
  + 在此类情况下，带?的值是否存在，并不取决于前面的值的最后一位。
  + 而是取决于Lucene的某项配置，当然这些配置也是保存在Lucene索引文件中的。
  + 如Position和Offset是否存储，取决于.fnm文件中对于每个域的配置(TermVector.WITH\_POSITIONS和TermVector.WITH\_OFFSETS)

为什么会存在以上两种情况，其实是可以理解的：

* 对于符合或然跟随规则的，是因为对于每一个A，B是否存在都不相同，当这种情况大量存在的时候，从一个Byte到一个Bit如此8倍的空间节约还是很值得的。
* 对于不符合或然跟随规则的，是因为某个值的是否存在的配置对于整个域(Field)甚至整个索引都是有效的，而非每次的情况都不相同，因而可以统一存放一个标志。

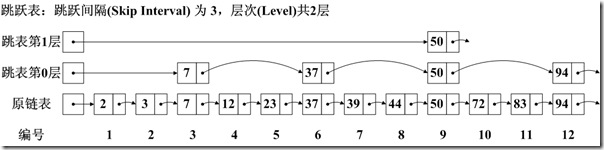
|  |
| --- |
| 文章中对如下格式的描述令人困惑：  Positions --> <PositionDelta,Payload?> Freq  Payload --> <PayloadLength?,PayloadData>  PositionDelta和Payload是否适用或然跟随规则呢？如何标识PayloadLength是否存在呢？  其实PositionDelta和Payload并不符合或然跟随规则，Payload是否存在，是由.fnm文件中对于每个域的配置中有关Payload的配置决定的(FieldOption.STORES\_PAYLOADS) 。  当Payload不存在时，PayloadDelta本身不遵从或然跟随原则。  当Payload存在时，格式应该变成如下：Positions --> <PositionDelta,PayloadLength?,PayloadData> Freq  从而PositionDelta和PayloadLength一起适用或然跟随规则。 |

**4. 跳跃表规则(Skip list)**

为了提高查找的性能，Lucene在很多地方采取的跳跃表的数据结构。

跳跃表(Skip List)是如图的一种数据结构，有以下几个基本特征：

* 元素是按顺序排列的，在Lucene中，或是按字典顺序排列，或是按从小到大顺序排列。
* 跳跃是有间隔的(Interval)，也即每次跳跃的元素数，间隔是事先配置好的，如图跳跃表的间隔为3。
* 跳跃表是由层次的(level)，每一层的每隔指定间隔的元素构成上一层，如图跳跃表共有2层。



需要注意一点的是，在很多数据结构或算法书中都会有跳跃表的描述，原理都是大致相同的，但是定义稍有差别：

* 对间隔(Interval)的定义： 如图中，有的认为间隔为2，即两个上层元素之间的元素数，不包括两个上层元素；有的认为是3，即两个上层元素之间的差，包括后面上层元素，不包括前面的上层元素；有的认为是4，即除两个上层元素之间的元素外，既包括前面，也包括后面的上层元素。Lucene是采取的第二种定义。
* 对层次(Level)的定义：如图中，有的认为应该包括原链表层，并从1开始计数，则总层次为3，为1，2，3层；有的认为应该包括原链表层，并从0计数，为0，1，2层；有的认为不应该包括原链表层，且从1开始计数，则为1，2层；有的认为不应该包括链表层，且从0开始计数，则为0，1层。Lucene采取的是最后一种定义。

跳跃表比顺序查找，大大提高了查找速度，如查找元素72，原来要访问2，3，7，12，23，37，39，44，50，72总共10个元素，应用跳跃表后，只要首先访问第1层的50，发现72大于50，而第1层无下一个节点，然后访问第2层的94，发现94大于72，然后访问原链表的72，找到元素，共需要访问3个元素即可。

然而Lucene在具体实现上，与理论又有所不同，在具体的格式中，会详细说明。