Lucene 原理与代码分析

觉先 (forfuture1978)

博客:

http://blog.csdn.net/forfuture1978

http://www.cnblogs.com/forfuture1978/

http://forfuture1978.javaeye.com/

邮箱:

forfuture1978@gmail.com

目录

目录 2
第一篇：原理篇 9
第一章：全文检索的基本原理 10
一、总论 10
二、索引里面究竟存些什么 13
三、如何创建索引 14
第一步：一些要索引的原文档(Document)。 14
第二步：将原文档传给分次组件(Tokenizer)。 14
第三步：将得到的词元(Token)传给语言处理组件(Linguistic Processor)。 15
第四步：将得到的词(Term)传给索引组件(Indexer)。 16
1. 利用得到的词(Term)创建一个字典。 16
2. 对字典按字母顺序进行排序。 17
3. 合并相同的词(Term)成为文档倒排(Posting List)链表。 18
四、如何对索引进行搜索？ 20
第一步：用户输入查询语句。 21
第二步：对查询语句进行词法分析，语法分析，及语言处理。 21
1. 词法分析主要用来识别单词和关键字。 21
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。 21
3. 语言处理同索引过程中的语言处理几乎相同。 22
第三步：搜索索引，得到符合语法树的文档。 22
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。 23
1. 计算权重(Term weight)的过程。 24
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)。 25
第二章：Lucene 的总体架构 29

第二篇：代码分析篇.....34
第三章：Lucene 的索引文件格式.....35
一、基本概念.....35
二、基本类型.....38
三、基本规则.....39
1. 前缀后缀规则(Prefix+Suffix).....39
2. 差值规则(Delta).....40
3. 或然跟随规则(A, B?).....41
4. 跳跃表规则(Skip list).....42
四、具体格式.....44
4.1. 正向信息.....44
4.1.1. 段的元数据信息(segments_N).....44
4.1.2. 域(Field)的元数据信息(.fnm).....60
4.1.3. 域(Field)的数据信息(.fdt，.fdx).....66
4.1.3. 词向量(Term Vector)的数据信息(.tvx，.tvd，.tvf).....69
4.2. 反向信息.....72
4.2.1. 词典(tis)及词典索引(tii)信息.....72
4.2.2. 文档号及词频(frq)信息.....74
4.2.3. 词位置(prx)信息.....78
4.3. 其他信息.....79
4.3.1. 标准化因子文件(nrm).....79
4.3.2. 删除文档文件(del).....81
五、总体结构.....82
第四章：Lucene 索引过程分析.....84
一、索引过程体系结构.....84
二、详细索引过程.....86
1、创建 IndexWriter 对象.....86
2、创建文档 Document 对象，并加入域(Field).....100
3、将文档加入 IndexWriter.....103

4、将文档加入 DocumentsWriter.....103
4.1、得到当前线程对应的文档集处理对象(DocumentsWriterThreadState).....111
4.2、用得到的文档集处理对象(DocumentsWriterThreadState)处理文档.....113
4.3、用 DocumentsWriter.finishDocument 结束本次文档添加.....132
5、DocumentsWriter 对 CharBlockPool，ByteBlockPool，IntBlockPool 的缓存管理.....132
6、关闭 IndexWriter 对象.....146
6.1、得到要写入的段名.....147
6.2、将缓存的内容写入段.....148
6.3、生成新的段信息对象.....169
6.4、准备删除文档.....169
6.5、生成 cfs 段.....169
6.6、删除文档.....170
第五章：Lucene 段合并(merge)过程分析.....174
一、段合并过程总论.....174
1.1、合并策略对段的选择.....175
1.2、反向信息的合并.....182
二、段合并的详细过程.....191
2.1、将缓存写入新的段.....191
2.2、选择合并段，生成合并任务.....192
2.2.1、用合并策略选择合并段.....192
2.2.2、注册段合并任务.....198
2.3、段合并器进行段合并.....199
2.3.1、合并存储域.....201
2.3.2、合并标准化因子.....206
2.3.3、合并词向量.....207
2.3.4、合并词典和倒排表.....210
第六章：Lucene 打分公式的数学推导.....216
第七章：Lucene 搜索过程解析.....222
一、Lucene 搜索过程总论.....222

二、Lucene 搜索详细过程.....223
2.1、打开 IndexReader 指向索引文件夹.....223
2.1.1、找到最新的 segment_N 文件.....223
2.1.2、通过 segment_N 文件中保存的各个段的信息打开各个段.....225
2.1.3、得到的 IndexReader 对象如下.....228
2.2、打开 IndexSearcher.....236
2.3、QueryParser 解析查询语句生成查询对象.....237
2.4、搜索查询对象.....242
2.4.1、创建 Weight 对象树，计算 Term Weight.....243
2.4.2、创建 Scorer 及 SumScorer 对象树.....268
2.4.3、进行倒排表合并.....288
2.4.4、收集文档结果集合及计算打分.....318
2.4.5、Lucene 如何在搜索阶段读取索引信息.....324
第八章：Lucene 的查询语法，JavaCC 及 QueryParser.....330
一、Lucene 的查询语法.....330
二、JavaCC 介绍.....332
2.1、第一个实例——正整数相加.....334
2.2、扩展语法分析器.....339
2.3、第二个实例：计算器.....344
三、解析 QueryParser.jj.....357
3.1、声明 QueryParser 类.....357
3.2、声明词法分析器.....357
3.3、声明语法分析器.....360
第九章：Lucene 的查询对象.....376
1、BoostingQuery.....376
2、CustomScoreQuery.....381
3、MoreLikeThisQuery.....385
4、MultiTermQuery.....394
4.1、TermRangeQuery.....394

4.2、NumericRangeQuery.....397
5、SpanQuery.....399
5.1、SpanFirstQuery.....400
5.2、SpanNearQuery.....401
5.3、SpanNotQuery.....408
5.4、SpanOrQuery.....411
5.5、FieldMaskingSpanQuery.....411
5.6、PayloadTermQuery 及 PayloadNearQuery.....414
6、FilteredQuery.....416
6.1、TermsFilter.....416
6.2、BooleanFilter.....417
6.3、DuplicateFilter.....419
6.4、FieldCacheRangeFilter<T>及 FieldCacheTermsFilter.....425
6.5、MultiTermQueryWrapperFilter<Q>.....432
6.6、QueryWrapperFilter.....433
6.7、SpanFilter.....434
6.7.1、SpanQueryFilter.....434
6.7.2、CachingSpanFilter.....435
第十章：Lucene 的分词器 Analyzer.....437
1、抽象类 Analyzer.....437
2、TokenStream 抽象类.....438
3、几个具体的 TokenStream.....439
3.1、NumericTokenStream.....439
3.2、SingleTokenTokenStream.....441
4、Tokenizer 也是一种 TokenStream.....442
4.1、CharTokenizer.....442
4.2、ChineseTokenizer.....445
4.3、KeywordTokenizer.....446
4.4、CJKTokenizer.....447

4.5、SentenceTokenizer 451
5、TokenFilter 也是一种 TokenStream....454
5.1、ChineseFilter 454
5.2、LengthFilter....456
5.3、LowerCaseFilter 457
5.4、NumericPayloadTokenFilter....458
5.5、PorterStemFilter....458
5.6、ReverseStringFilter....460
5.7、SnowballFilter 462
5.8、TeeSinkTokenFilter 464
6、不同的 Analyzer 就是组合不同的 Tokenizer 和 TokenFilter 得到最后的 TokenStream....469
6.1、ChineseAnalyzer....469
6.2、CJKAnalyzer 469
6.3、PorterStemAnalyzer 470
6.4、SmartChineseAnalyzer 470
6.5、SnowballAnalyzer....470
7、Lucene 的标准分词器....471
7.1、StandardTokenizerImpl.jflex....471
7.2、StandardTokenizer....475
7.3、StandardFilter 476
7.4、StandardAnalyzer....477
8、不同的域使用不同的分词器....479
8.1、PerFieldAnalyzerWrapper 479
第三篇：问题篇....482
问题一：为什么能搜的到“中华 AND 共和国”却搜不到“中华共和国”?....483
问题二：stemming 和 lemmatization 的关系 487
问题三：影响 Lucene 对文档打分的四种方式 493
在索引阶段设置 Document Boost 和 Field Boost，存储在(.nrm)文件中。 493
在搜索语句中，设置 Query Boost....499

继承并实现自己的 Similarity.....501
继承并实现自己的 collector.....514
问题四：Lucene 中的 TooManyClause 异常.....517
问题五：Lucene 的事务性.....519
问题六：用 Lucene 构建实时的索引.....521
1、初始化阶段.....521
2、合并索引阶段.....522
3、重新打开硬盘索引的 IndexReader.....523
4、替代 IndexReader.....524
5、多个索引.....525

第一篇：原理篇

第一章：全文检索的基本原理

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？

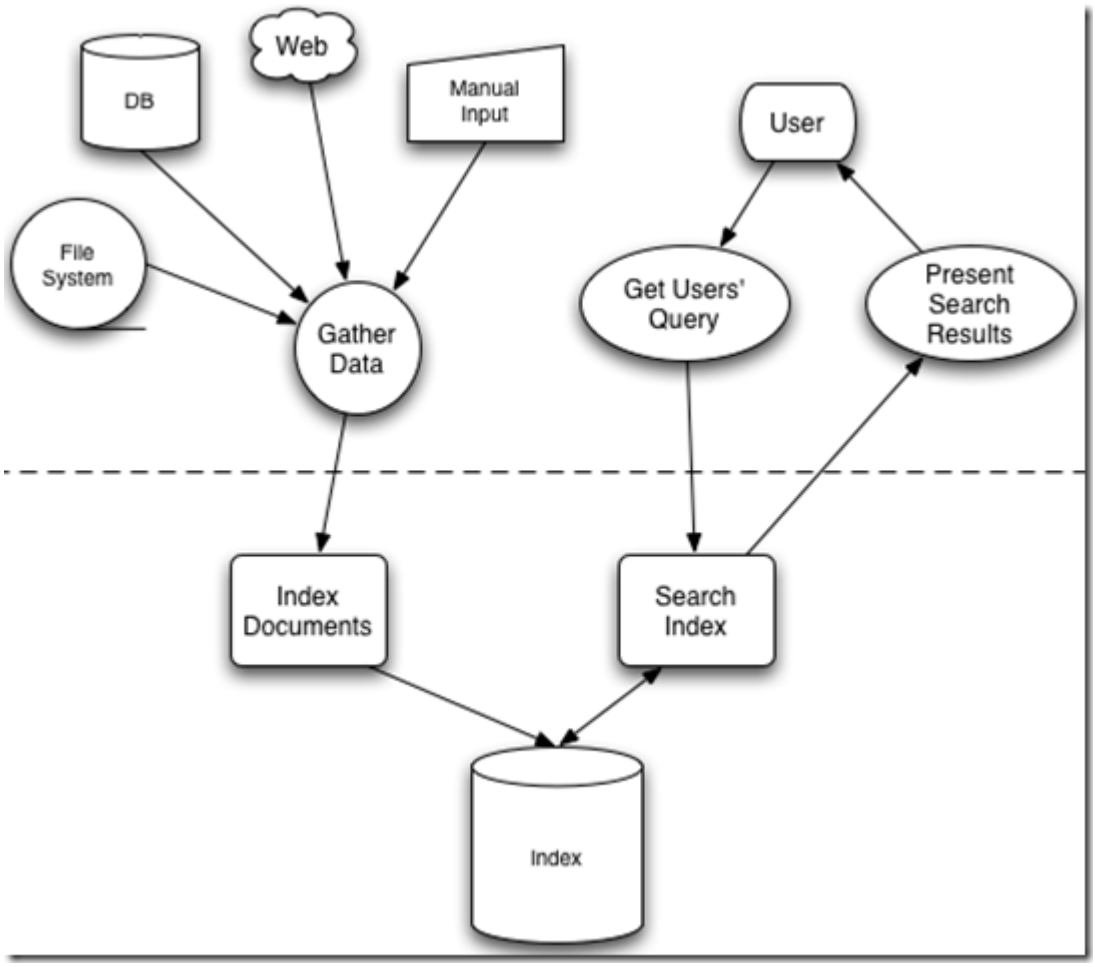
这种想法很天然，却构成了全文检索的基本思路，也即将非结构化数据中的一部分信息提取出来，重新组织，使其变得有一定结构，然后对此有一定结构的数据进行搜索，从而达到搜索相对较快的目的。

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：

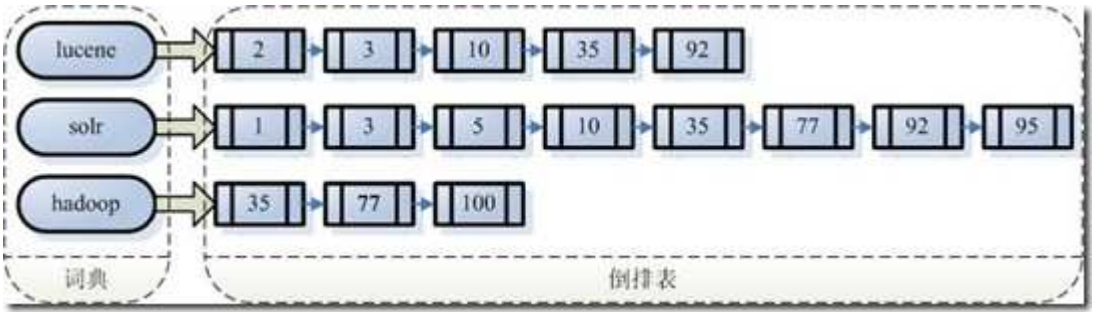
其实是由于我们想要搜索的信息和非结构化数据中所存储的信息不一致造成的。

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



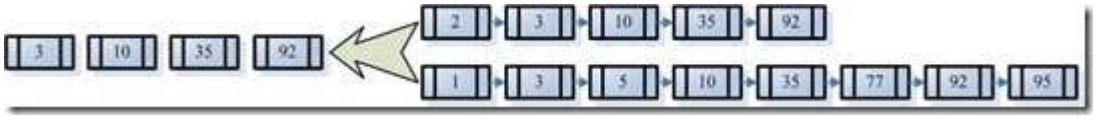
左边保存的是一系列字符串，称为**词典**。

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。



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

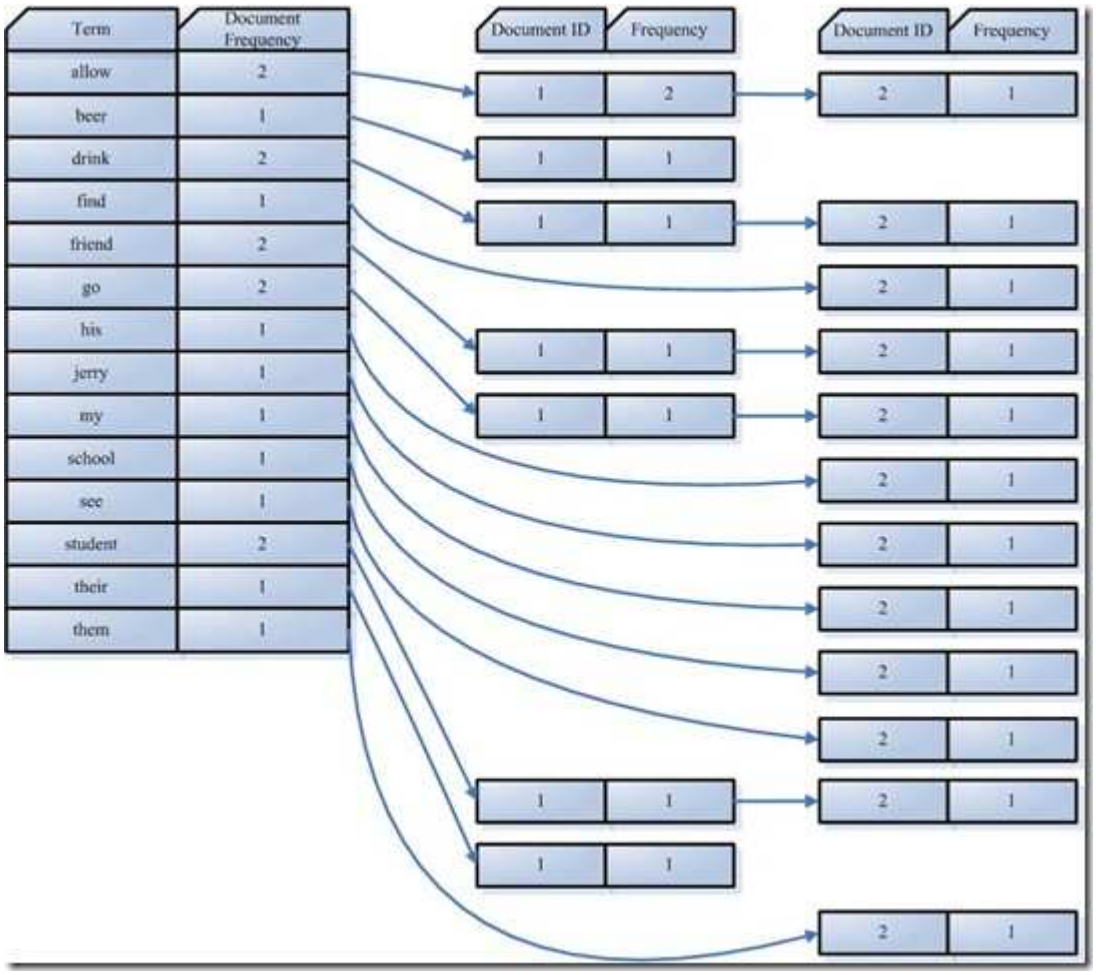
jerry	2
go	2
school	2
see	2
his	2
student	2
find	2
them	2
drink	2
allow	2

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。

四、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。

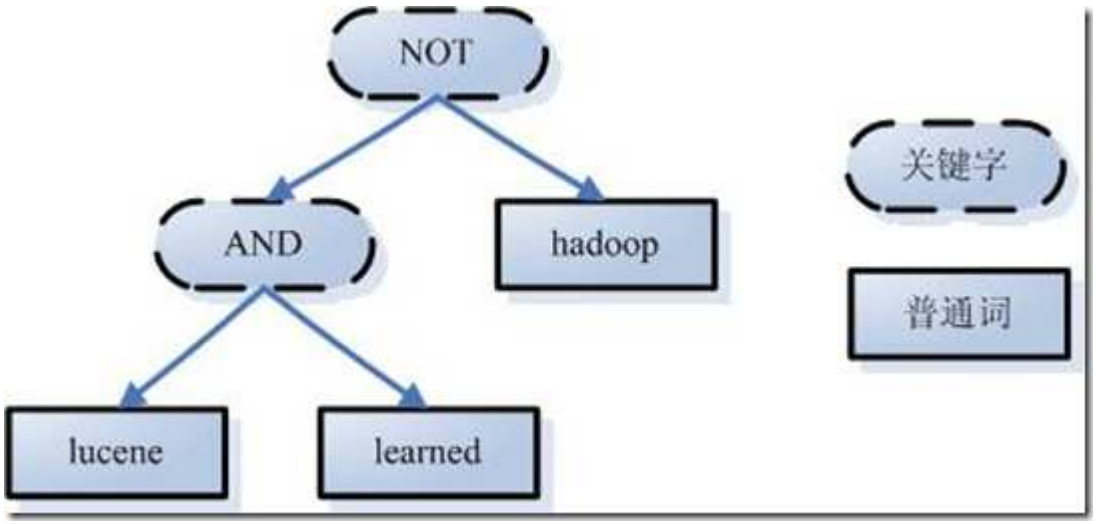
如上述例子中，经过词法分析，得到单词有 lucene，learned，hadoop, 关键字有 AND, NOT。

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。

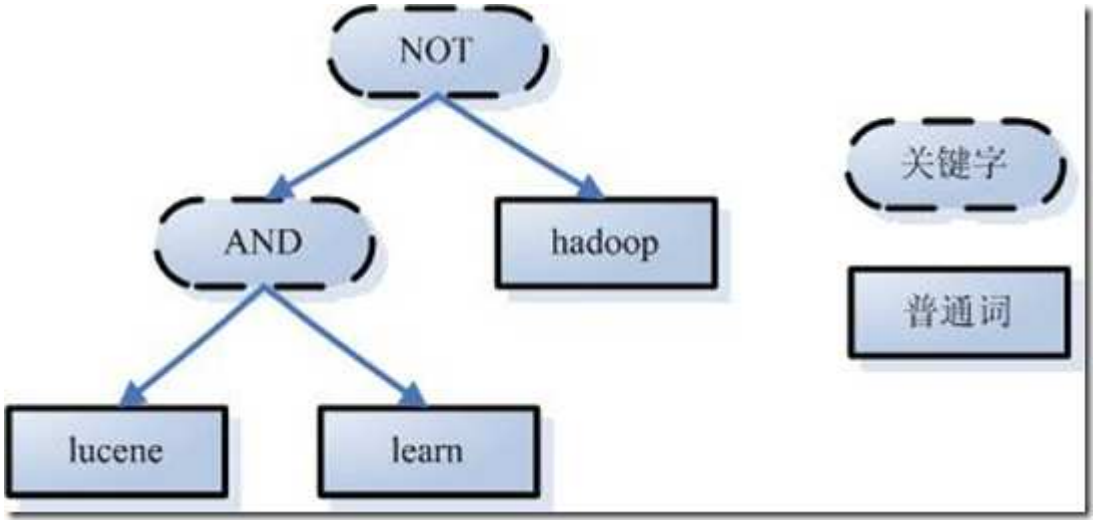
如上述例子，lucene AND learned NOT hadoop 形成的语法树如下：



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：

$$w_{t,d} = tf_{t,d} \times \log(n / df_t)$$

$w_{t,d}$ = the weight of the term t in document d
$tf_{t,d}$ = frequency of term t in document d
n = total number of documents
df_t = the number of documents that contain term t

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}

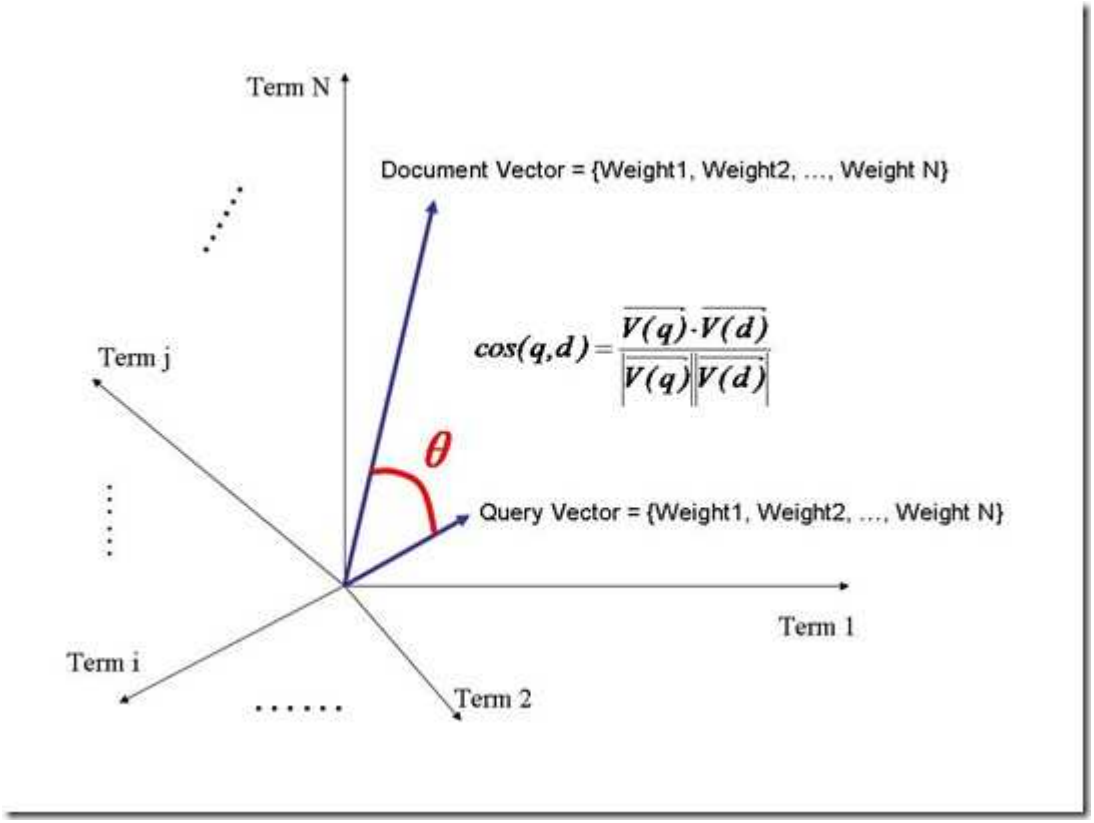
同样我们把查询语句看作一个简单的文档，也用向量来表示。

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：

$$score(q,d) = \frac{\vec{V}_q \cdot \vec{V}_d}{\left|\vec{V}_q\right|\left|\vec{V}_d\right|} = \frac{\sum_{i=1}^{n} w_{i,q} w_{i,d}}{\sqrt{\sum_{i=1}^{n} w_{i,q}^2}\sqrt{\sum_{i=1}^{n} w_{i,d}^2}}$$

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：

$$SC(Q,D_1) = \frac{(0.176)(0.176)}{\sqrt{0.477^2+0.477^2+0.176^2+0.176^2}\sqrt{0.176^2+0.477^2+0.176^2}} \approx 0.08$$

$$SC(Q,D_2) = \frac{(0.954)(0.477)+(0.176)^2}{\sqrt{0.176^2+0.477^2+0.954^2+0.176^2}\sqrt{0.176^2+0.477^2+0.176^2}} \approx 0.825$$

$$SC(Q,D_3) = \frac{(0.176)^2+(0.176)^2}{\sqrt{0.176^2+0.176^2+0.176^2+0.176^2}\sqrt{0.176^2+0.477^2+0.176^2}} \approx 0.327$$

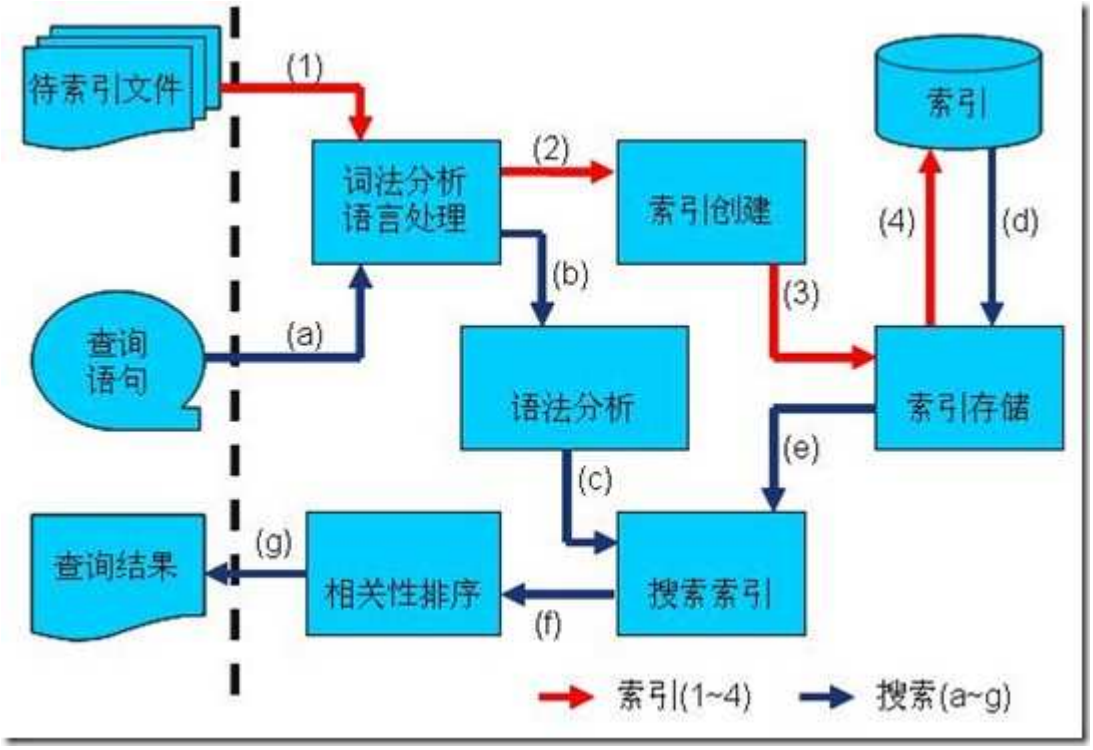
于是文档二相关性最高，先返回，其次是文档一，最后是文档三。

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。

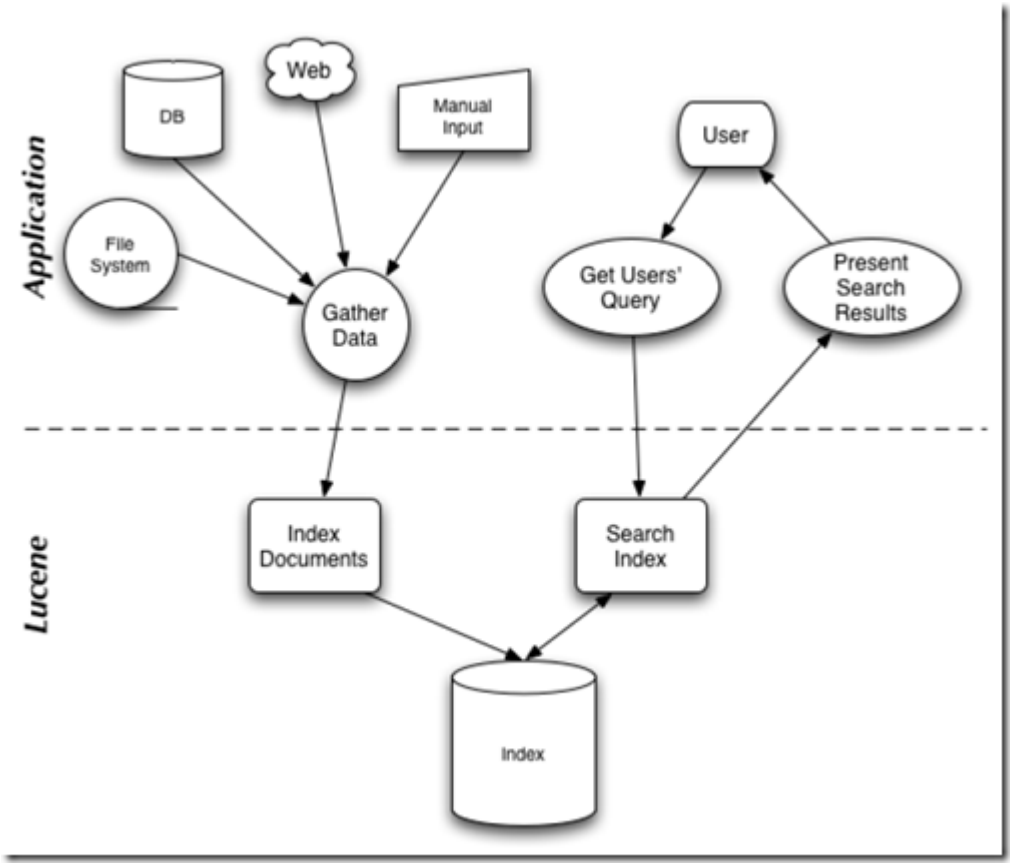
下面我们可以进入 Lucene 的世界了。

第二章：Lucene 的总体架构

Lucene 总的来说是：

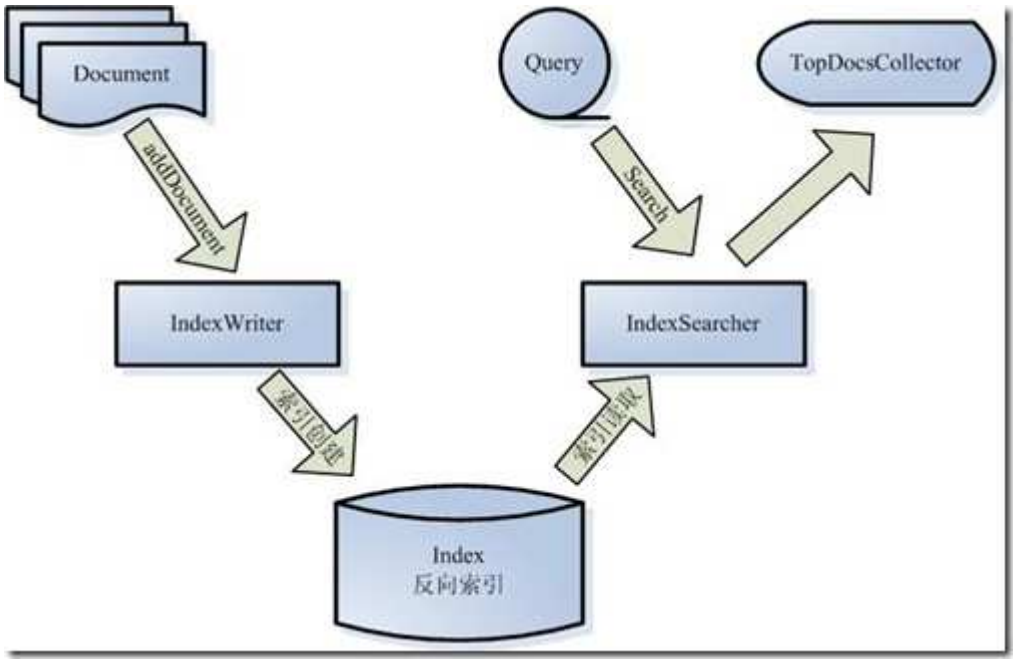
- 一个高效的，可扩展的，全文检索库。
- 全部用 Java 实现，无须配置。
- 仅支持纯文本文件的索引(Indexing)和搜索(Search)。
- 不负责由其他格式的文件抽取纯文本文件，或从网络中抓取文件的过程。

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。

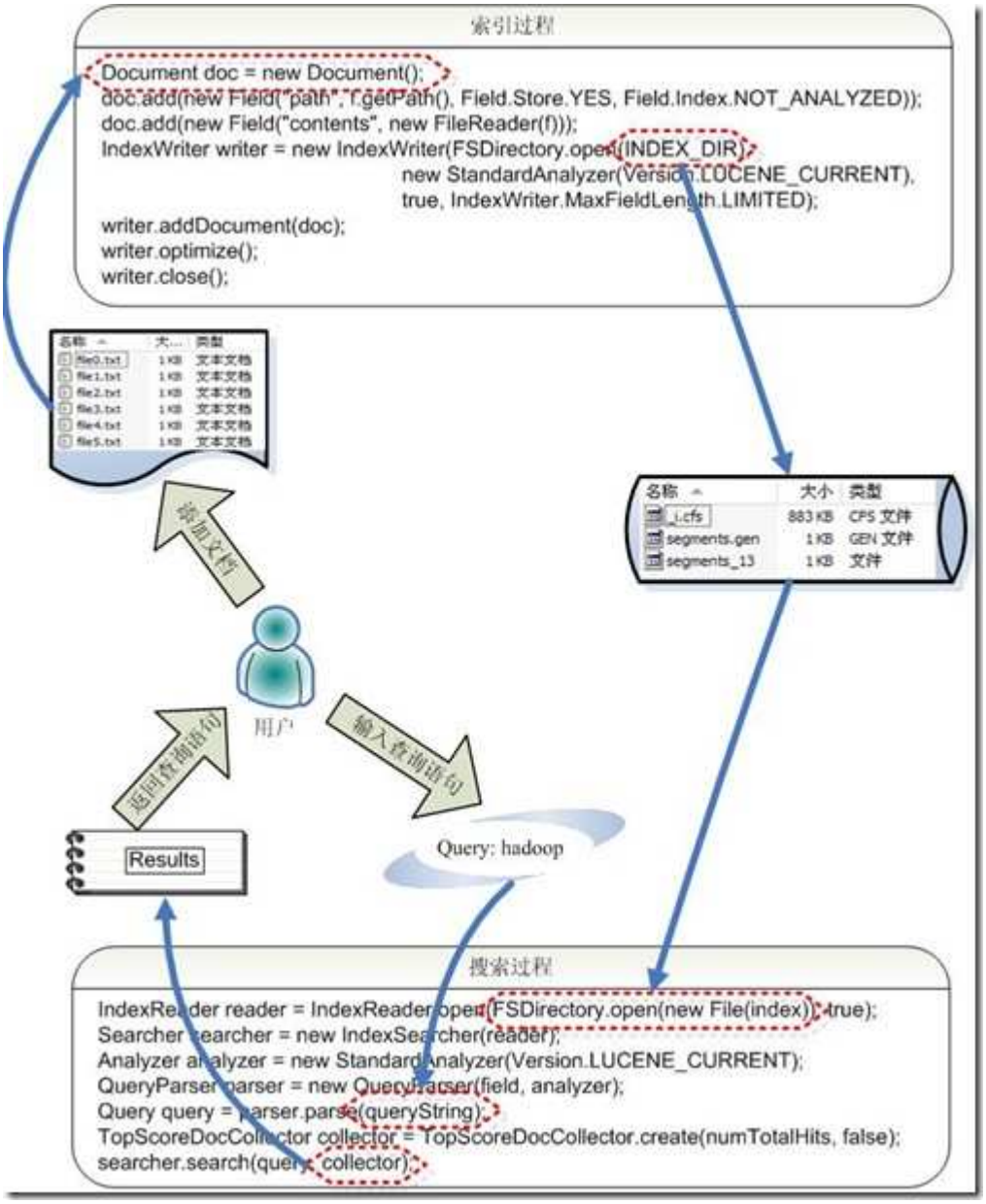
让我们更细一些看 Lucene 的各组件：



- **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。

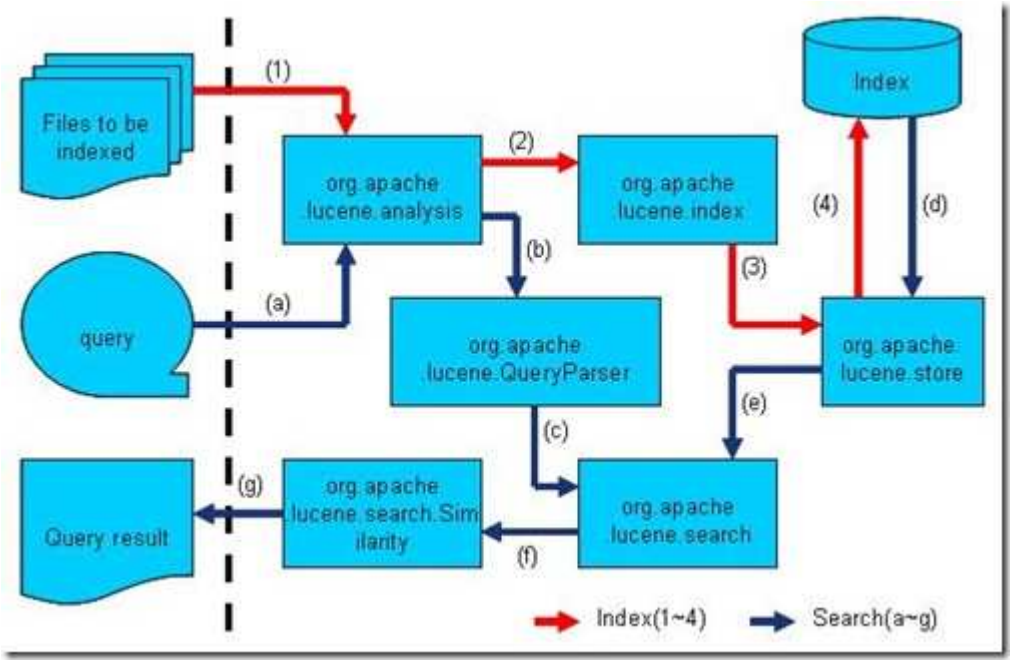
- 搜索过程如下：
  - IndexReader 将磁盘上的索引信息读入到内存，INDEX_DIR 就是索引文件存放的位置。
  - 创建 IndexSearcher 准备进行搜索。
  - 创建 Analyer 用来对查询语句进行词法分析和语言处理。
  - 创建 QueryParser 用来对查询语句进行语法分析。
  - QueryParser 调用 parser 进行语法分析，形成查询语法树，放到 Query 中。
  - IndexSearcher 调用 search 对查询语法树 Query 进行搜索，得到结果 TopScoreDocCollector。

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。

第二篇：代码分析篇

第三章：Lucene 的索引文件格式

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。

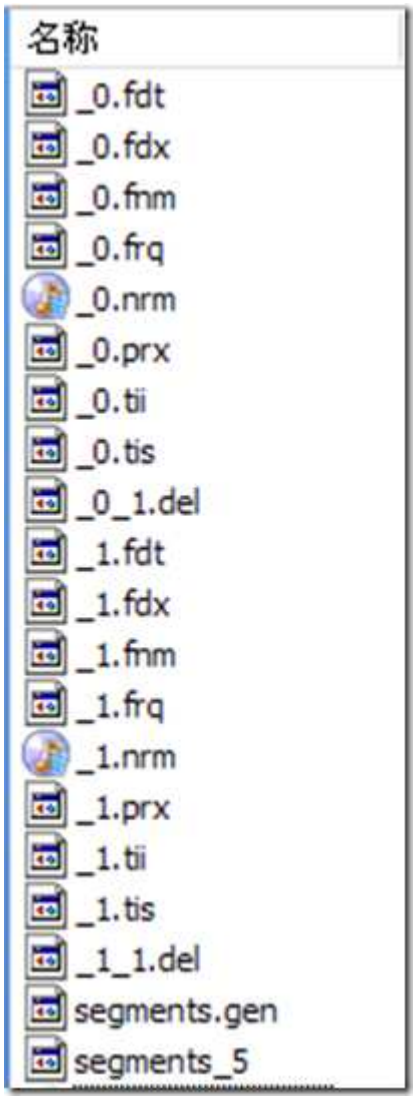
当我们真正进入到 Lucene 源代码之中的时候，我们会发现:

- 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。

Value	First byte	Second byte	Third byte
0	00000000		
1	00000001		
2	00000010		
...			
127	01111111		
128	10000000	00000001	
129	10000001	00000001	
130	10000010	00000001	
...			
16,383	11111111	01111111	
16,384	10000000	10000000	00000001
16,385	10000001	10000000	00000001
...			

- 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]

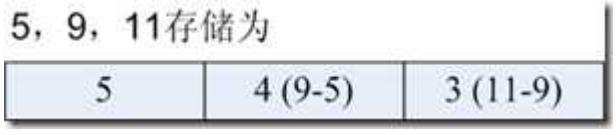
共需要 22 个 Byte。

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。

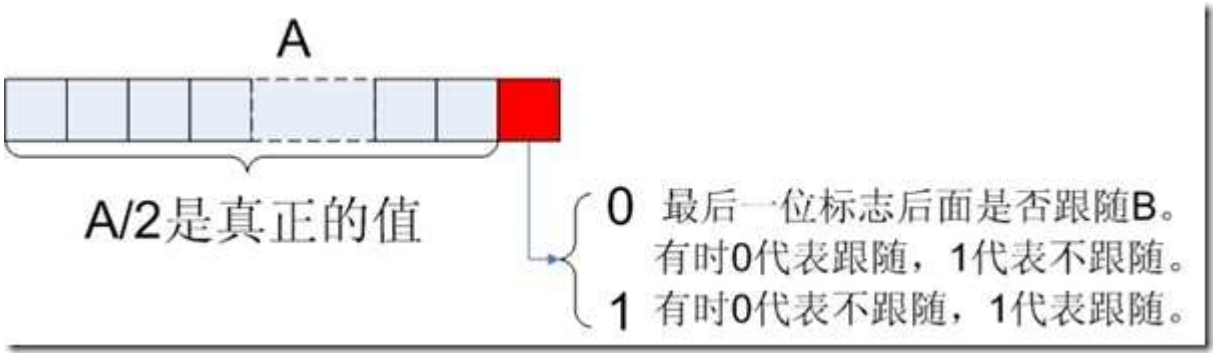
3. 或然跟随规则(A, B?)

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。

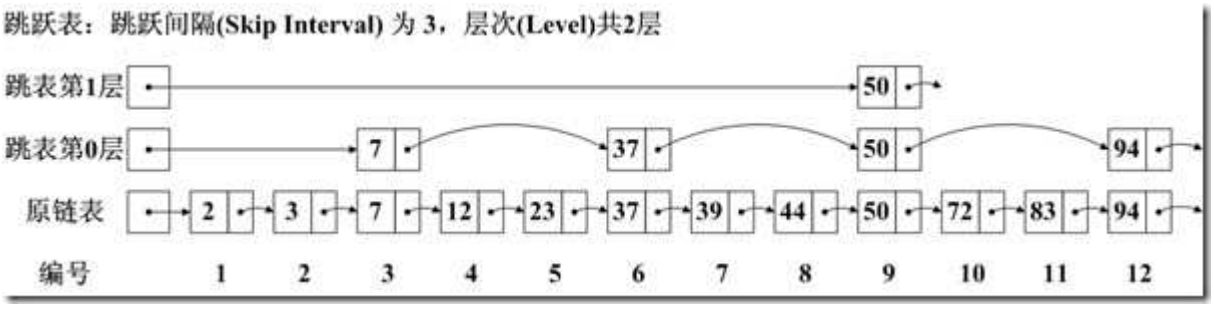
4. 跳跃表规则(Skip list)

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。

四、具体格式

上面曾经交代过，Lucene 保存了从 Index 到 Segment 到 Document 到 Field 一直到 Term 的正向信息，也包括了从 Term 到 Document 映射的反向信息，还有其他一些 Lucene 特有的信息。下面对这三种信息一一介绍。

4.1. 正向信息

Index –> Segments (segments.gen, segments_N) –> Field(fnm, fdx, fdt) –> Term (tvx, tvd, tvf)

上面的层次结构不是十分的准确，因为 segments.gen 和 segments_N 保存的是段(segment)的元数据信息(metadata)，其实是每个 Index 一个的，而段的真正的数据信息，是保存在域(Field)和词(Term)中的。

4.1.1. 段的元数据信息(segments_N)

一个索引(Index)可以同时存在多个 segments_N(至于如何存在多个 segments_N，在描述完详细信息之后会举例说明)，然而当我们要打开一个索引的时候，我们必须要选择一个来打开，那如何选择哪个 segments_N 呢？

Lucene 采取以下过程：

- 其一，在所有的 segments_N 中选择 N 最大的一个。基本逻辑参照 SegmentInfos.getCurrentSegmentGeneration(File[] files)，其基本思路就是在所有以 segments 开头，并且不是 segments.gen 的文件中，选择 N 最大的一个作为 genA。
- 其二，打开 segments.gen，其中保存了当前的 N 值。其格式如下，读出版本号(Version)，然后再读出两个 N，如果两者相等，则作为 genB。

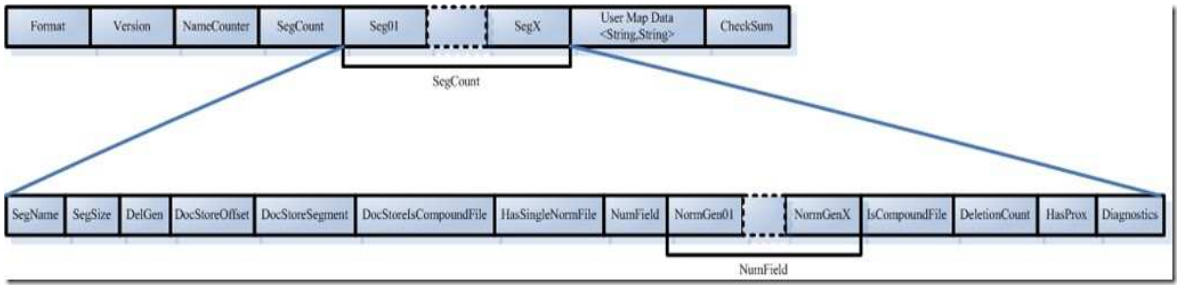

```
IndexInput genInput = directory.openInput(IndexFileNames.SEGMENTS_GEN);//"segments.gen"
int version = genInput.readInt();//读出版本号
if (version == FORMAT_LOCKLESS) {//如果版本号正确
```

```
    long gen0 = genInput.readLong();//读出第一个 N
    long gen1 = genInput.readLong();//读出第二个 N
    if (gen0 == gen1) {//如果两者相等则为 genB
        genB = gen0;
    }
}
```

- 其三，在上述得到的 genA 和 genB 中选择最大的那个作为当前的 N，方才打开 segments_N 文件。其基本逻辑如下：

```
if (genA > genB)
    gen = genA;
else
    gen = genB;
```

如下图是 segments_N 的具体格式：



- Format：
  - 索引文件格式的版本号。
  - 由于 Lucene 是在不断开发过程中的，因而不同版本的 Lucene，其索引文件格式也不尽相同，于是规定一个版本号。
  - Lucene 2.1 此值-3，Lucene 2.9 时，此值为-9。
  - 当用某个版本号的 IndexReader 读取另一个版本号生成的索引的时候，会因为此值不同而报错。

- Version：
  - 索引的版本号，记录了 IndexWriter 将修改提交到索引文件中的次数。
  - 其初始值大多数情况下从索引文件里面读出，仅仅在索引开始创建的时候，被赋予当前的时间，已取得一个唯一值。
  - 其值改变在

IndexWriter.commit->IndexWriter.startCommit->SegmentInfos.prepareCommit->SegmentInfos.write->writeLong(++version)

  - 其初始值之所最初取一个时间，是因为我们并不关心 IndexWriter 将修改提交到索引的具体次数，而更关心到底哪个是最新的。IndexReader 中常比较自己的 version 和索引文件中的 version 是否相同来判断此 IndexReader 被打开后，还有没有被 IndexWriter 更新。

```
//在 DirectoryReader 中有一下函数。
public boolean isCurrent() throws CorruptIndexException, IOException {
  return SegmentInfos.readCurrentVersion(directory) == segmentInfos.getVersion();
}
```

- NameCount
  - 是下一个新段(Segment)的段名。
  - 所有属于同一个段的索引文件都以段名作为文件名，一般为_0.xxx, _0.yyy, _1.xxx, _1.yyy ……
  - 新生成的段的段名一般为原有最大段名加一。
  - 如同的索引，NameCount 读出来是 2，说明新的段为_2.xxx, _2.yyy

- SegCount
  - 段(Segment)的个数。
  - 如上图，此值为 2。
- SegCount 个段的元数据信息：
  - SegName
    - 段名，所有属于同一个段的文件都有以段名作为文件名。
    - 如上图，第一个段的段名为"_0"，第二个段的段名为"_1"
  - SegSize
    - 此段中包含的文档数
    - 然而此文档数是包括已经删除，又没有 optimize 的文档的，因为在 optimize 之前，Lucene 的段中包含了所有被索引过的文档，而被删除的文档是保存在.del 文件中的，在搜索的过程中，是先从段中读到了被删除的文档，然后再用.del

中的标志，将这篇文档过滤掉。

- 如下的代码形成了上图的索引，可以看出索引了两篇文档形成了_0 段，然后又删除了其中一篇，形成了_0_1.del，又索引了两篇文档形成_1 段，然后又删除了其中一篇，形成_1_1.del。因而在两个段中，此值都是 2。

```
IndexWriter writer = new IndexWriter(FSDirectory.open(INDEX_DIR), new StandardAnalyzer(Version.LUCENE_CURRENT), true, IndexWriter.MaxFieldLength.LIMITED);
writer.setUseCompoundFile(false);
indexDocs(writer, docDir);//docDir 中只有两篇文档
//文档一为：Students should be allowed to go out with their friends, but not allowed to drink beer.
//文档二为：My friend Jerry went to school to see his students but found them drunk which is not allowed.
writer.commit();//提交两篇文档，形成_0 段。
writer.deleteDocuments(new Term("contents", "school"));//删除文档二
writer.commit();//提交删除，形成_0_1.del
indexDocs(writer, docDir);//再次索引两篇文档，Lucene 不能判别文档与文档的不同，因而算两篇新的文档。
writer.commit();//提交两篇文档，形成_1 段
writer.deleteDocuments(new Term("contents", "school"));//删除第二次添加的文档二
writer.close();//提交删除，形成_1_1.del
```

- DelGen
  - .del 文件的版本号
  - Lucene 中，在 optimize 之前，删除的文档是保存在.del 文件中的。
  - 在 Lucene 2.9 中，文档删除有以下几种方式：
    - ✓ IndexReader.deleteDocument(int docID)是用 IndexReader 按文档号删除。
    - ✓ IndexReader.deleteDocuments(Term term)是用 IndexReader 删除包含此词(Term)的文档。
    - ✓ IndexWriter.deleteDocuments(Term term)是用 IndexWriter 删除包含此词

(Term)的文档。

- ✓ IndexWriter.deleteDocuments(Term[] terms)是用 IndexWriter 删除包含这些词(Term)的文档。
- ✓ IndexWriter.deleteDocuments(Query query)是用 IndexWriter 删除能满足此查询(Query)的文档。
- ✓ IndexWriter.deleteDocuments(Query[] queries)是用 IndexWriter 删除能满足这些查询(Query)的文档。
- ✓ 原来的版本中 Lucene 的删除一直是由 IndexReader 来完成的，在 Lucene 2.9 中虽可以用 IndexWriter 来删除，但是其实真正的实现是在 IndexWriter 中，保存了 readerpool，当 IndexWriter 向索引文件提交删除的时候，仍然是从 readerpool 中得到相应的 IndexReader，并用 IndexReader 来进行删除的。下面的代码可以说明：

```
IndexWriter.applyDeletes()
-> DocumentsWriter.applyDeletes(SegmentInfos)
    -> reader.deleteDocument(doc);
```

- ◆ DelGen 是每当 IndexWriter 向索引文件中提交删除操作的时候，加 1，并生成新的.del 文件。

```
IndexWriter.commit()
-> IndexWriter.applyDeletes()
   -> IndexWriter$ReaderPool.release(SegmentReader)
      -> SegmentReader(IndexReader).commit()
         -> SegmentReader.doCommit(Map)
            -> SegmentInfo.advanceDelGen()
               -> if (delGen == NO) {
                    delGen = YES;
                  } else {
                    delGen++;
                  }
```

IndexWriter writer = new IndexWriter(FSDirectory.open(INDEX_DIR), new StandardAnalyzer(Version.LUCENE_CURRENT), true, IndexWriter.MaxFieldLength.LIMITED);

writer.setUseCompoundFile(false);

indexDocs(writer, docDir);//索引两篇文档，一篇包含"school"，另一篇包含"beer"

writer.commit();//提交两篇文档到索引文件，形成段(Segment) "_0"

writer.deleteDocuments(new Term("contents", "school"));//删除包含"school"的文档，其实是删除了两篇文档中的一篇。

writer.commit();//提交删除到索引文件，形成"_0_1.del"

writer.deleteDocuments(new Term("contents", "beer"));//删除包含"beer"的文档，其实是删除了两篇文档中的另一篇。

writer.commit();//提交删除到索引文件，形成"_0_2.del"

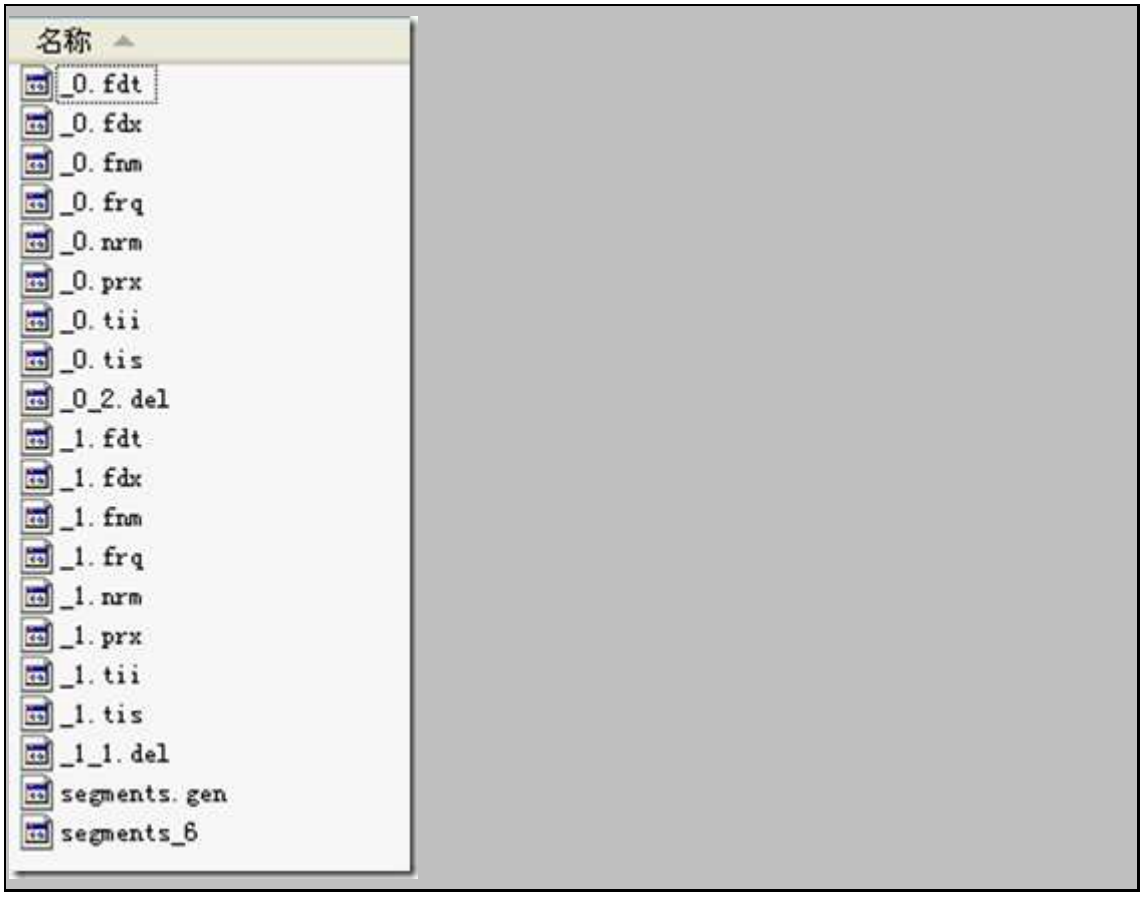
indexDocs(writer, docDir);//索引两篇文档，和上次的文档相同，但是 Lucene 无法区分，认为是另外两篇文档。

writer.commit();//提交两篇文档到索引文件，形成段"_1"

writer.deleteDocuments(new Term("contents", "beer"));//删除包含"beer"的文档，其中段"_0"已经无可删除，段"_1"被删除一篇。

writer.close();//提交删除到索引文件，形成"_1_1.del"

形成的索引文件如下：

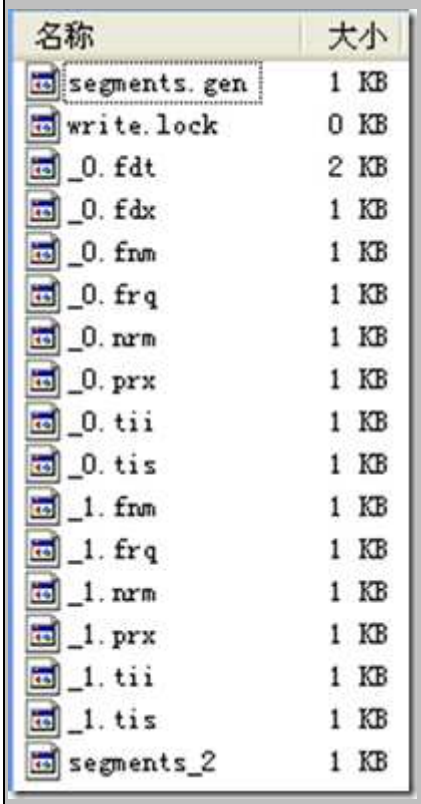


- DocStoreOffset
- DocStoreSegment
- DocStoreIsCompoundFile
  - 对于域(Stored Field)和词向量(Term Vector)的存储可以有不同的方式，即可以每个段(Segment)单独存储自己的域和词向量信息，也可以多个段共享域和词向量，把它们存储到一个段中去。
  - 如果 DocStoreOffset 为-1，则此段单独存储自己的域和词向量，从存储文件上来看，如果此段段名为 XXX，则此段有自己的 XXX.fdt，XXX.fdx，XXX.tvf，XXX.tvd，XXX.tvx 文件。DocStoreSegment 和 DocStoreIsCompoundFile 在此处不被保存。
  - 如果 DocStoreOffset 不为-1，则 DocStoreSegment 保存了共享的段的名字，比如为 YYY，DocStoreOffset 则为此段的域及词向量信息在共享段中的偏移量。则此段没有自己的 XXX.fdt，XXX.fdx，XXX.tvf，XXX.tvd，XXX.tvx 文件，而是将信息存放在共享段的 YYY.fdt，YYY.fdx，YYY.tvf，YYY.tvd，YYY.tvx 文件中。

- DocumentsWriter 中有两个成员变量：String segment 是当前索引信息存放的段，String docStoreSegment 是域和词向量信息存储的段。两者可以相同也可以不同，决定了域和词向量信息是存储在本段中，还是和其他的段共享。
- IndexWriter.flush(boolean triggerMerge, boolean flushDocStores, boolean flushDeletes)中第二个参数 flushDocStores 会影响到是否单独或是共享存储。其实最终影响的是 DocumentsWriter.closeDocStore()。每当 flushDocStores 为 false 时，closeDocStore 不被调用，说明下次添加到索引文件中的域和词向量信息是同此次共享一个段的。直到 flushDocStores 为 true 的时候，closeDocStore 被调用，从而下次添加到索引文件中的域和词向量信息将被保存在一个新的段中，不同此次共享一个段(在这里需要指出的是 Lucene 的一个很奇怪的实现，虽然下次域和词向量信息是被保存到新的段中，然而段名却是这次被确定了的，在 initSegmentName 中当 docStoreSegment == null 时，被置为当前的 segment，而非下一个新的 segment，docStoreSegment = segment，于是会出现如下面的例子的现象)。
- 好在共享域和词向量存储并不是经常被使用到，实现也或有缺陷，暂且解释到此。

```
IndexWriter writer = new IndexWriter(FSDirectory.open(INDEX_DIR), new StandardAnalyzer(Version.LUCENE_CURRENT), true, IndexWriter.MaxFieldLength.LIMITED);
writer.setUseCompoundFile(false);
indexDocs(writer, docDir);
writer.flush();
//flush 生成 segment "_0"，并且 flush 函数中，flushDocStores 设为 false，也即下个段将同本段共享域和词向量信息，这时 DocumentsWriter 中的 docStoreSegment= "_0"。
indexDocs(writer, docDir);
writer.commit();
//commit 生成 segment "_1"，由于上次 flushDocStores 设为 false，于是段"_1"的域以及词向量信息是保存在"_0"中的，在这个时刻，段"_1"并不生成自己的"_1.fdx"和"_1.fdt"。然而在 commit 函数中，flushDocStores 设为 true，也即下个段将单独使用新的段来存储域和词向量信息。然
```

而这时，DocumentsWriter 中的 docStoreSegment= "_1"，也即当段"_2"存储其域和词向量信息的时候，是存在"_1.fdx"和"_1.fdt"中的，而段"_1"的域和词向量信息却是存在"_0.fdt"和"_0.fdx"中的，这一点非常令人困惑。 如图 writer.commit 的时候，_1.fdt 和_1.fdx 并没有形成。



indexDocs(writer, docDir);

writer.flush();

//段"_2"形成，由于上次 flushDocStores 设为 true，其域和词向量信息是新创建一个段保存的，却是保存在_1.fdt 和_1.fdx 中的，这时候才产生了此二文件。

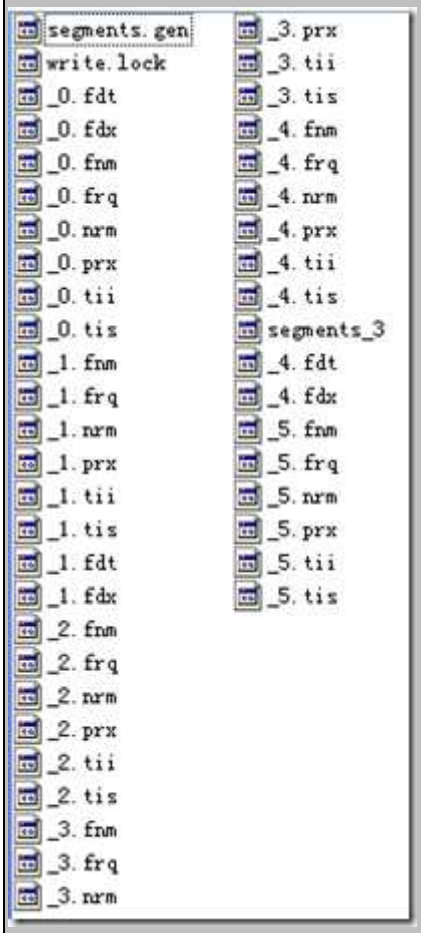
名称	大小
segments.gen	1 KB
write.lock	0 KB
_0.fdt	2 KB
_0.fdx	1 KB
_0.fnm	1 KB
_0.frq	1 KB
_0.nrm	1 KB
_0.prx	1 KB
_0.tii	1 KB
_0.tis	1 KB
_1.fnm	1 KB
_1.frq	1 KB
_1.nrm	1 KB
_1.prx	1 KB
_1.tii	1 KB
_1.tis	1 KB
segments_2	1 KB
_1.fdt	2 KB
_1.fdx	1 KB
_2.fnm	1 KB
_2.frq	1 KB
_2.nrm	1 KB
_2.prx	1 KB
_2.tii	1 KB
_2.tis	1 KB
_3.fnm	1 KB
_3.frq	1 KB
_3.nrm	1 KB
_3.prx	1 KB
_3.tii	1 KB
_3.tis	1 KB

```
indexDocs(writer, docDir);

writer.flush();

//段"_3"形成，由于上次 flushDocStores 设为 false，其域和词向量信息是共享一个段保存的，
也是是保存在_1.fdt 和_1.fdx 中的

indexDocs(writer, docDir);
```

```
writer.commit();
//段"_4"形成，由于上次 flushDocStores 设为 false，其域和词向量信息是共享一个段保存的，也是是保存在_1.fdt 和_1.fdx 中的。然而函数 commit 中 flushDocStores 设为 true，也意味着下一个段将新创建一个段保存域和词向量信息，此时 DocumentsWriter 中 docStoreSegment="_4"，也表明了虽然段"_4"的域和词向量信息保存在了段"_1"中，将来的域和词向量信息却要保存在段"_4"中。此时"_4.fdx"和"_4.fdt"尚未产生。
```


```
indexDocs(writer, docDir);

writer.flush();
```

//段"_5"形成，由于上次 flushDocStores 设为 true，其域和词向量信息是新创建一个段保存的，却是保存在_4.fdt 和_4.fdx 中的，这时候才产生了此二文件。

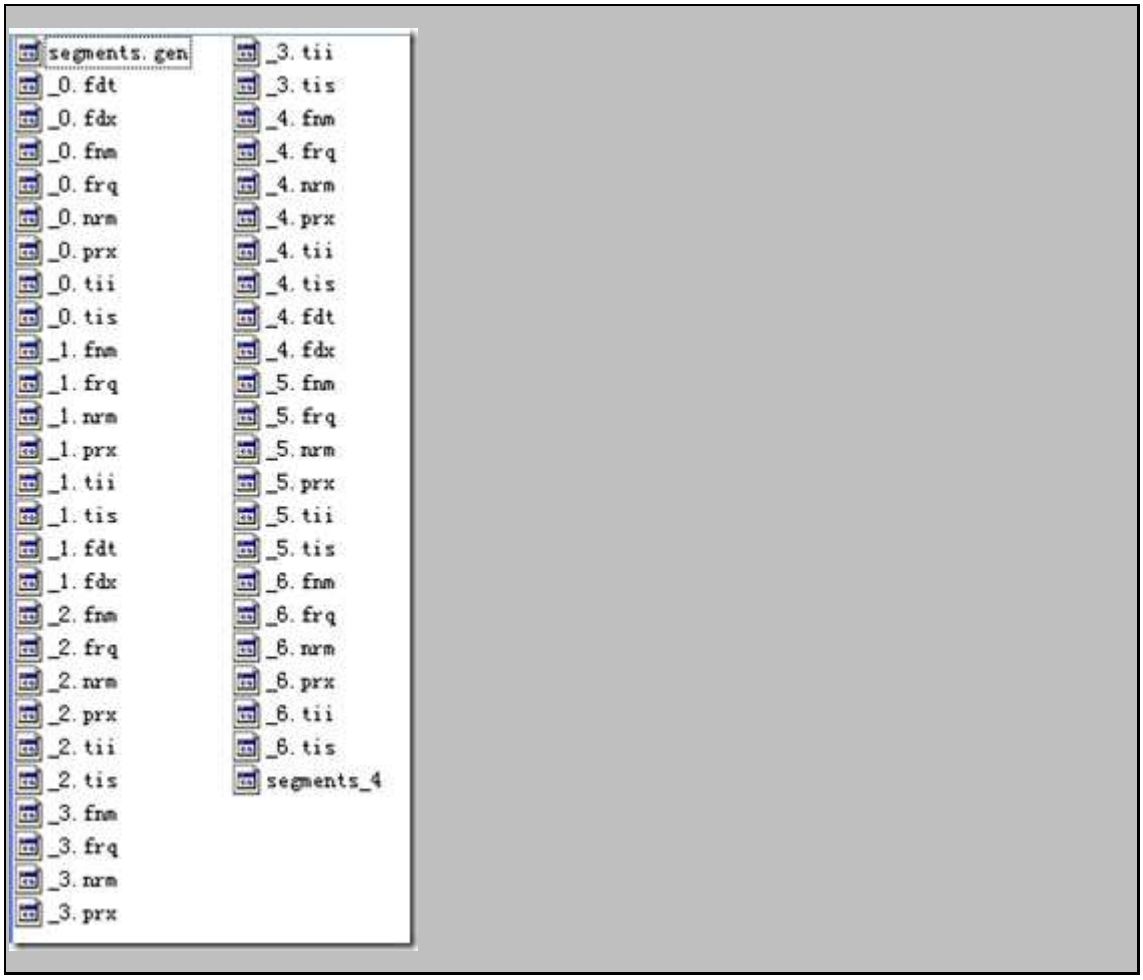


indexDocs(writer, docDir);

writer.commit();

writer.close();

//段"_6"形成，由于上次 flushDocStores 设为 false，其域和词向量信息是共享一个段保存的，也是是保存在_4.fdt 和_4.fdx 中的



- HasSingleNormFile
  - 在搜索的过程中，标准化因子(Normalization Factor)会影响文档最后的评分。
  - 不同的文档重要性不同，不同的域重要性也不同。因而每个文档的每个域都可以有自己的标准化因子。
  - 如果 HasSingleNormFile 为 1，则所有的标准化因子都是存在.nrm 文件中的。
  - 如果 HasSingleNormFile 不是 1，则每个域都有自己的标准化因子文件.fN
- NumField
  - 域的数量
- NormGen
  - 如果每个域有自己的标准化因子文件，则此数组描述了每个标准化因子文件的版本号，也即.fN 的 N。
- IsCompoundFile
  - 是否保存为复合文件，也即把同一个段中的文件按照一定格式，保存在一个文

件当中，这样可以减少每次打开文件的个数。

- 是否为复合文件，由接口 IndexWriter.setUseCompoundFile(boolean)设定。
- 非符合文件同符合文件的对比如下图：

- DeletionCount
  - 记录了此段中删除的文档的数目。
- HasProx
  - 如果至少有一个段 omitTf 为 false，也即词频(term freqency)需要被保存，则 HasProx 为 1，否则为 0。
- Diagnostics
  - 调试信息。

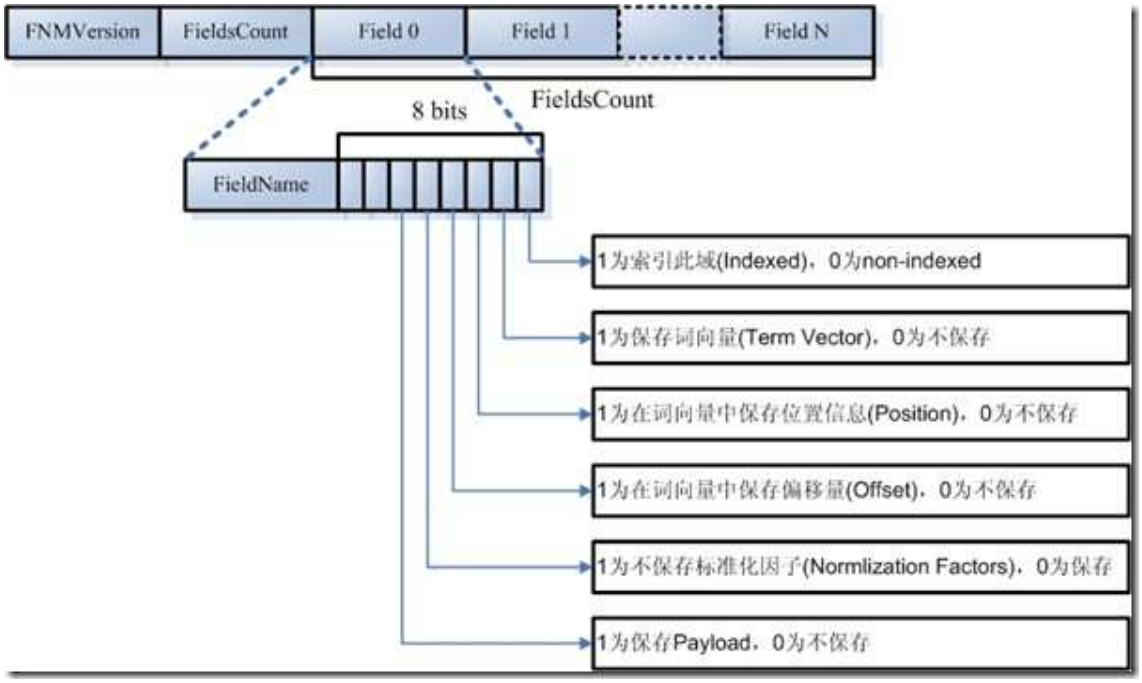
- User map data
  - 保存了用户从字符串到字符串的映射 Map<String,String>
- CheckSum
  - 此文件 segment_N 的校验和。

读取此文件格式参考 SegmentInfos.read(Directory directory, String segmentFileName):

- int format = input.readInt();
- version = input.readLong(); // read version
- counter = input.readInt(); // read counter
- for (int i = input.readInt(); i > 0; i--) // read segmentInfos
  - add(new SegmentInfo(directory, format, input));
    - name = input.readString();
    - docCount = input.readInt();
    - delGen = input.readLong();
    - docStoreOffset = input.readInt();
    - docStoreSegment = input.readString();
    - docStoreIsCompoundFile = (1 == input.readByte());
    - hasSingleNormFile = (1 == input.readByte());
    - int numNormGen = input.readInt();
    - normGen = new long[numNormGen];
    - for(int j=0;j<numNormGen;j++)
    - normGen[j] = input.readLong();
    - isCompoundFile = input.readByte();
    - delCount = input.readInt();
    - hasProx = input.readByte() == 1;
    - diagnostics = input.readStringStringMap();
- userData = input.readStringStringMap();
- final long checksumNow = input.getChecksum();
- final long checksumThen = input.readLong();

4.1.2. 域(Field)的元数据信息(.fnm)

一个段(Segment)包含多个域，每个域都有一些元数据信息，保存在.fnm 文件中，.fnm 文件的格式如下：



- FNMVersion
  - 是 fnm 文件的版本号，对于 Lucene 2.9 为-2
- FieldsCount
  - 域的数目
- 一个数组的域(Fields)
  - FieldName：域名，如"title"，"modified"，"content"等。
  - FieldBits:一系列标志位，表明对此域的索引方式
    - 最低位：1 表示此域被索引，0 则不被索引。所谓被索引，也即放到倒排表中去。
      - ✓ 仅仅被索引的域才能够被搜到。
      - ✓ Field.Index.NO 则表示不被索引。
      - ✓ Field.Index.ANALYZED 则表示不但被索引，而且被分词，比如索引"hello world"后，无论是搜"hello"，还是搜"world"都能够被搜到。
      - ✓ Field.Index.NOT_ANALYZED 表示虽然被索引，但是不分词，比如索引"hello world"后，仅当搜"hello world"时，能够搜到，搜"hello"和搜"world"都搜不到。

- ✓ 一个域出了能够被索引，还能够被存储，仅仅被存储的域是搜索不到的，但是能通过文档号查到，多用于不想被搜索到，但是在通过其它域能够搜索到的情况下，能够随着文档号返回给用户的域。
- ✓ Field.Store.Yes 则表示存储此域，Field.Store.NO 则表示不存储此域。

◆ 倒数第二位：1 表示保存词向量，0 为不保存词向量。

- ✓ Field.TermVector.YES 表示保存词向量。
- ✓ Field.TermVector.NO 表示不保存词向量。

◆ 倒数第三位：1 表示在词向量中保存位置信息。

- ✓ Field.TermVector.WITH_POSITIONS

◆ 倒数第四位：1 表示在词向量中保存偏移量信息。

- ✓ Field.TermVector.WITH_OFFSETS

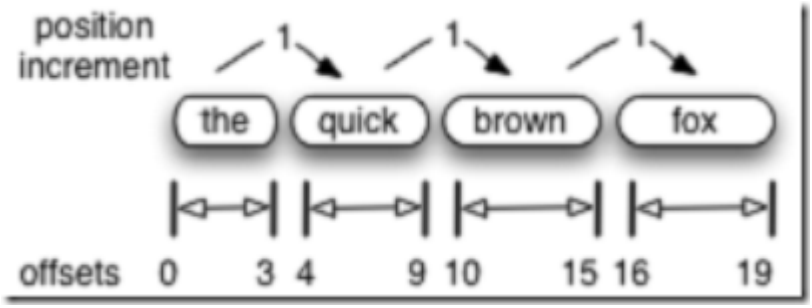
◆ 倒数第五位：1 表示不保存标准化因子

- ✓ Field.Index.ANALYZED_NO_NORMS
- ✓ Field.Index.NOT_ANALYZED_NO_NORMS

◆ 倒数第六位：是否保存 payload

要了解域的元数据信息，还要了解以下几点：

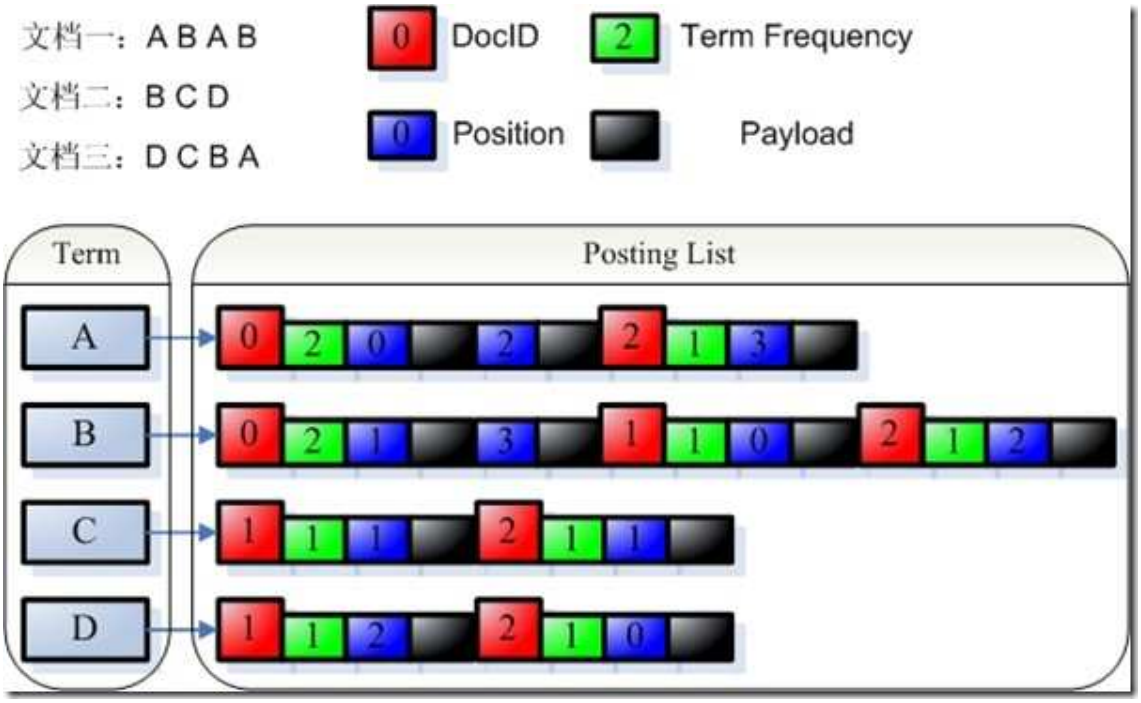
- 位置(Position)和偏移量(Offset)的区别
  - 位置是基于词 Term 的，偏移量是基于字母或汉字的。



- 索引域(Indexed)和存储域(Stored)的区别
  - 一个域为什么会被存储(store)而不被索引(Index)呢？在一个文档中的所有信息中，有这样一部分信息，可能不想被索引从而可以搜索到，但是当这个文档由于其他的信息被搜索到时，可以同其他信息一同返回。
  - 举个例子，读研究生时，您好不容易写了一篇论文交给您的导师，您的导师却要他

所第一作者而您做第二作者，然而您导师不想别人在论文系统中搜索您的名字时找到这篇论文，于是在论文系统中，把第二作者这个 Field 的 Indexed 设为 false，这样别人搜索您的名字，永远不知道您写过这篇论文，只有在别人搜索您导师的名字从而找到您的文章时，在一个角落表述着第二作者是您。

- payload 的使用
  - 我们知道，索引是以倒排表形式存储的，对于每一个词，都保存了包含这个词的一个链表，当然为了加快查询速度，此链表多用跳跃表进行存储。
  - Payload 信息就是存储在倒排表中的，同文档号一起存放，多用于存储与每篇文档相关的一些信息。当然这部分信息也可以存储域里(stored Field)，两者从功能上基本是一样的，然而当要存储的信息很多的时候，存放在倒排表里，利用跳跃表，有利于大大提高搜索速度。
  - Payload 的存储方式如下图：



  - Payload 主要有以下几种用法：
    - 存储每个文档都有的信息：比如有的时候，我们想给每个文档赋一个我们自己的文档号，而不是用 Lucene 自己的文档号。于是我们可以声明一个特殊的域(Field)"_ID"和特殊的词(Term)"_ID"，使得每篇文档都包含词"_ID"，于是在词

"_ID"的倒排表里面对于每篇文档又有一项，每一项都有一个 payload，于是我们可以在 payload 里面保存我们自己的文档号。每当我们得到一个 Lucene 的文档号的时候，就能从跳跃表中查找到我们自己的文档号。

```
//声明一个特殊的域和特殊的词
public static final String ID_PAYLOAD_FIELD = "_ID";
public static final String ID_PAYLOAD_TERM = "_ID";
public static final Term ID_TERM = new Term(ID_PAYLOAD_TERM, ID_PAYLOAD_FIELD);
//声明一个特殊的 TokenStream，它只生成一个词(Term)，就是那个特殊的词，在特殊的域里面。
static class SinglePayloadTokenStream extends TokenStream {
    private Token token;
    private boolean returnToken = false;
    SinglePayloadTokenStream(String idPayloadTerm) {
        char[] term = idPayloadTerm.toCharArray();
        token = new Token(term, 0, term.length, 0, term.length);
    }
    void setPayloadValue(byte[] value) {
        token.setPayload(new Payload(value));
        returnToken = true;
    }
    public Token next() throws IOException {
        if (returnToken) {
            returnToken = false;
            return token;
        } else {
            return null;
        }
    }
```

```
}
//对于每一篇文档，都让它包含这个特殊的词，在特殊的域里面
SinglePayloadTokenStream singlePayloadTokenStream = new SinglePayloadTokenStream(ID_PAYLOAD_TERM);
singlePayloadTokenStream.setPayloadValue(long2bytes(id));
doc.add(new Field(ID_PAYLOAD_FIELD, singlePayloadTokenStream));
//每当得到一个 Lucene 的文档号时，通过以下的方式得到 payload 里面的文档号
long id = 0;
TermPositions tp = reader.termPositions(ID_PAYLOAD_TERM);
boolean ret = tp.skipTo(docID);
tp.nextPosition();
int payloadlength = tp.getPayloadLength();
byte[] payloadBuffer = new byte[payloadlength];
tp.getPayload(payloadBuffer, 0);
id = bytes2long(payloadBuffer);
tp.close();
```

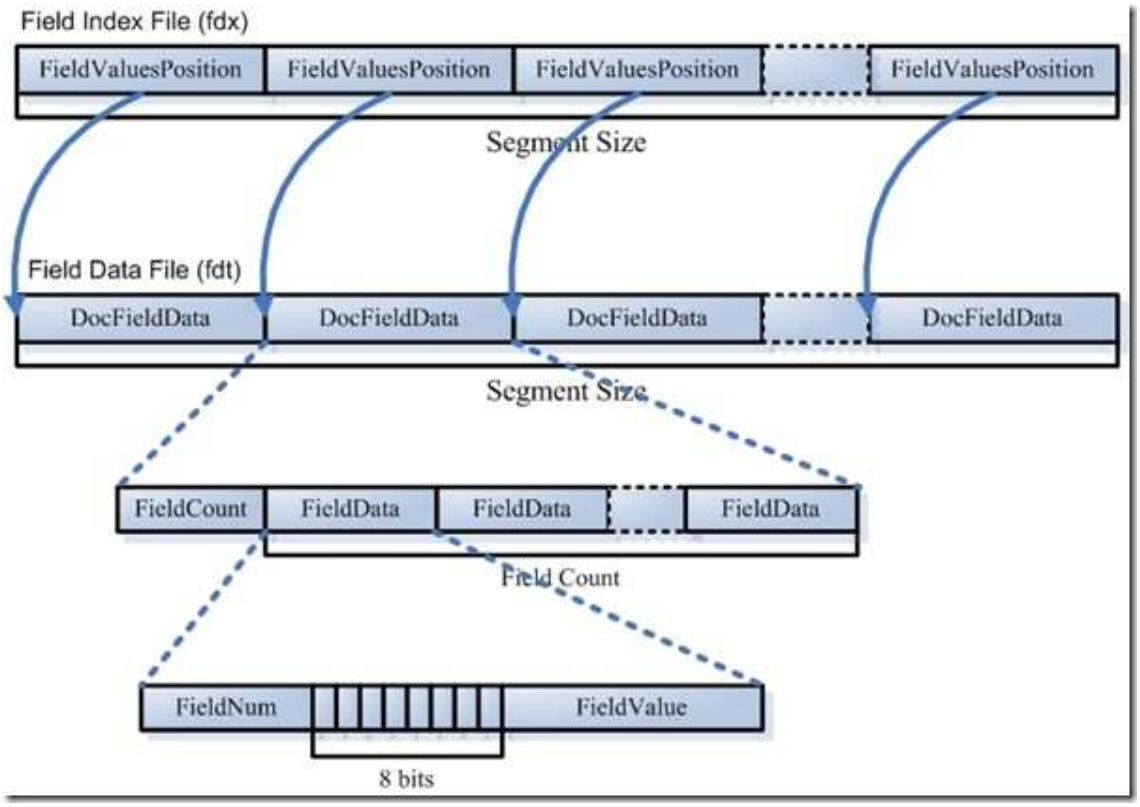
- 影响词的评分
  - ✓ 在 Similarity 抽象类中有函数 public float scorePayload(byte [] payload, int offset, int length) 可以根据 payload 的值影响评分。

读取域元数据信息的代码如下：

```
FieldInfos.read(IndexInput, String)
● int firstInt = input.readVInt();
● size = input.readVInt();
● for (int i = 0; i < size; i++)
    ■ String name = input.readString();
    ■ byte bits = input.readByte();
    ■ boolean isIndexed = (bits & IS_INDEXED) != 0;
    ■ boolean storeTermVector = (bits & STORE_TERMVECTOR) != 0;
```

- boolean storePositionsWithTermVector = (bits & STORE_POSITIONS_WITH_TERMVECTOR) != 0;
- boolean storeOffsetWithTermVector = (bits & STORE_OFFSET_WITH_TERMVECTOR) != 0;
- boolean omitNorms = (bits & OMIT_NORMS) != 0;
- boolean storePayloads = (bits & STORE_PAYLOADS) != 0;
- boolean omitTermFreqAndPositions = (bits & OMIT_TERM_FREQ_AND_POSITIONS) != 0;

4.1.3. 域(Field)的数据信息(.fdt，.fdx)



- 域数据文件(fdt):
  - 真正保存存储域(stored field)信息的是 fdt 文件
  - 在一个段(segment)中总共有 segment size 篇文档，所以 fdt 文件中共有 segment size 个项，每一项保存一篇文档的域的信息
  - 对于每一篇文档，一开始是一个 fieldcount，也即此文档包含的域的数目，接下来

是 fieldcount 个项，每一项保存一个域的信息。

- 对于每一个域，fieldnum 是域号，接着是一个 8 位的 byte，最低一位表示此域是否分词(tokenized)，倒数第二位表示此域是保存字符串数据还是二进制数据，倒数第三位表示此域是否被压缩，再接下来就是存储域的值，比如 new Field("title", "lucene in action", Field.Store.Yes, …)，则此处存放的就是"lucene in action"这个字符串。

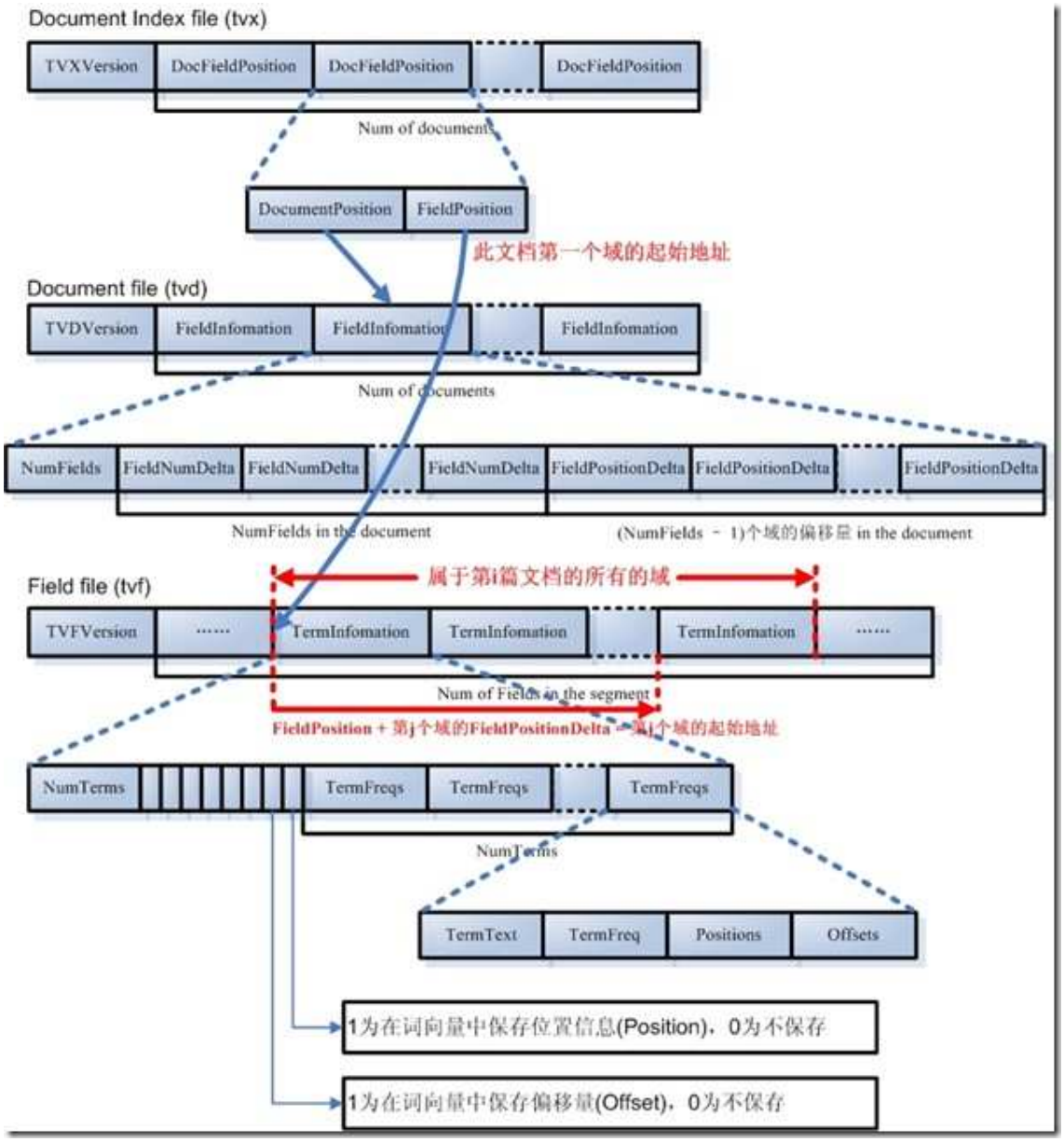
- 域索引文件(fdx)
  - 由域数据文件格式我们知道，每篇文档包含的域的个数，每个存储域的值都是不一样的，因而域数据文件中 segment size 篇文档，每篇文档占用的大小也是不一样的，那么如何在 fdt 中辨别每一篇文档的起始地址和终止地址呢，如何能够更快的找到第 n 篇文档的存储域的信息呢？就是要借助域索引文件。
  - 域索引文件也总共有 segment size 个项，每篇文档都有一个项，每一项都是一个 long，大小固定，每一项都是对应的文档在 fdt 文件中的起始地址的偏移量，这样如果我们想找到第 n 篇文档的存储域的信息，只要在 fdx 中找到第 n 项，然后按照取出的 long 作为偏移量，就可以在 fdt 文件中找到对应的存储域的信息。

读取域数据信息的代码如下：

```
Document FieldsReader.doc(int n, FieldSelector fieldSelector)
● long position = indexStream.readLong();//indexStream points to ".fdx"
● fieldsStream.seek(position);//fieldsStream points to "fdt"
● int numFields = fieldsStream.readVInt();
● for (int i = 0; i < numFields; i++)
    ■ int fieldNumber = fieldsStream.readVInt();
    ■ byte bits = fieldsStream.readByte();
    ■ boolean compressed = (bits & FieldsWriter.FIELD_IS_COMPRESSED) != 0;
    ■ boolean tokenize = (bits & FieldsWriter.FIELD_IS_TOKENIZED) != 0;
    ■ boolean binary = (bits & FieldsWriter.FIELD_IS_BINARY) != 0;
    ■ if (binary)
        ◆ int toRead = fieldsStream.readVInt();
```

- final byte[] b = new byte[toRead];
- fieldsStream.readBytes(b, 0, b.length);
- if (compressed)
  - int toRead = fieldsStream.readVInt();
  - final byte[] b = new byte[toRead];
  - fieldsStream.readBytes(b, 0, b.length);
  - uncompress(b),
- else
  - fieldsStream.readString()

4.1.3. 词向量(Term Vector)的数据信息(.tvx，.tvd，.tvf)



词向量信息是从索引(index)到文档(document)到域(field)到词(term)的正向信息，有了词向量信息，我们就可以得到一篇文档包含那些词的信息。

- 词向量索引文件(tvx)
  - 一个段(segment)包含 N 篇文档，此文件就有 N 项，每一项代表一篇文档。
  - 每一项包含两部分信息：第一部分是词向量文档文件(tvd)中此文档的偏移量，第二部分是词向量域文件(tvf)中此文档的第一个域的偏移量。

- 词向量文档文件(tvd)
  - 一个段(segment)包含 N 篇文档，此文件就有 N 项，每一项包含了此文档的所有的域的信息。
  - 每一项首先是此文档包含的域的个数 NumFields，然后是一个 NumFields 大小的数组，数组的每一项是域号。然后是一个(NumFields - 1)大小的数组，由前面我们知道，每篇文档的第一个域在 tvf 中的偏移量在 tvx 文件中保存，而其他(NumFields - 1)个域在 tvf 中的偏移量就是第一个域的偏移量加上这(NumFields - 1)个数组的每一项的值。
- 词向量域文件(tvf)
  - 此文件包含了此段中的所有的域，并不对文档做区分，到底第几个域到第几个域是属于那篇文档，是由 tvx 中的第一个域的偏移量以及 tvd 中的(NumFields - 1)个域的偏移量来决定的。
  - 对于每一个域，首先是此域包含的词的个数 NumTerms，然后是一个 8 位的 byte，最后一位是指定是否保存位置信息，倒数第二位是指定是否保存偏移量信息。然后是 NumTerms 个项的数组，每一项代表一个词(Term)，对于每一个词，由词的文本 TermText，词频 TermFreq(也即此词在此文档中出现的次数)，词的位置信息，词的偏移量信息。

读取词向量数据信息的代码如下：

TermVectorsReader.get(int docNum, String field, TermVectorMapper)

- int fieldNumber = fieldInfos.fieldNumber(field);//通过 field 名字得到 field 号
- seekTvx(docNum);//在 tvx 文件中按 docNum 文档号找到相应文档的项
- long tvdPosition = tvx.readLong();//找到 tvd 文件中相应文档的偏移量
- tvd.seek(tvdPosition);//在 tvd 文件中按偏移量找到相应文档的项
- int fieldCount = tvd.readVInt();//此文档包含的域的个数。
- for (int i = 0; i < fieldCount; i++) //按域号查找域
  - number = tvd.readVInt();
  - if (number == fieldNumber) found = i;

- position = tvx.readLong();//在 tvx 中读出此文档的第一个域在 tvf 中的偏移量
- for (int i = 1; i <= found; i++)
  - position += tvd.readVLong();//加上所要找的域在 tvf 中的偏移量
- tvf.seek(position);
- int numTerms = tvf.readVInt();
- byte bits = tvf.readByte();
- storePositions = (bits & STORE_POSITIONS_WITH_TERMVECTOR) != 0;
- storeOffsets = (bits & STORE_OFFSET_WITH_TERMVECTOR) != 0;
- for (int i = 0; i < numTerms; i++)
  - start = tvf.readVInt();
  - deltaLength = tvf.readVInt();
  - totalLength = start + deltaLength;
  - tvf.readBytes(byteBuffer, start, deltaLength);
  - term = new String(byteBuffer, 0, totalLength, "UTF-8");
  - if (storePositions)
    - positions = new int[freq];
    - int prevPosition = 0;
    - for (int j = 0; j < freq; j++)
      - positions[j] = prevPosition + tvf.readVInt();
      - prevPosition = positions[j];
  - if (storeOffsets)
    - offsets = new TermVectorOffsetInfo[freq];
    - int prevOffset = 0;
    - for (int j = 0; j < freq; j++)
    - int startOffset = prevOffset + tvf.readVInt();
    - int endOffset = startOffset + tvf.readVInt();
    - offsets[j] = new TermVectorOffsetInfo(startOffset, endOffset);
    - prevOffset = endOffset;

4.2. 反向信息

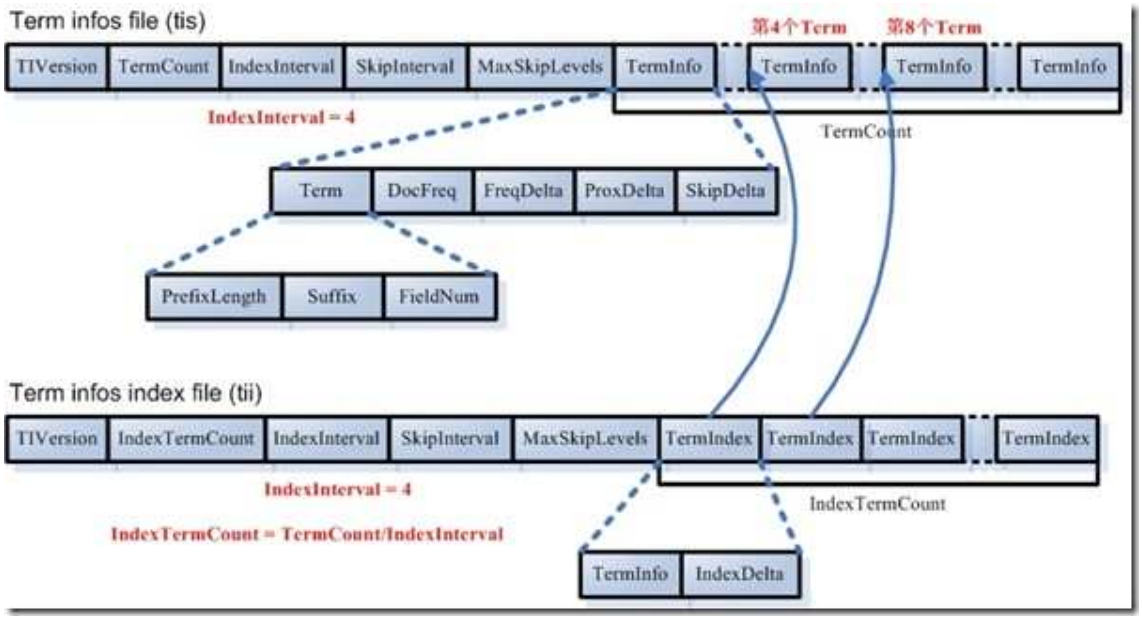
反向信息是索引文件的核心，也即反向索引。

反向索引包括两部分，左面是词典(Term Dictionary)，右面是倒排表(Posting List)。

在 Lucene 中，这两部分是分文件存储的，词典是存储在 tii，tis 中的，倒排表又包括两部分，一部分是文档号及词频，保存在 frq 中，一部分是词的位置信息，保存在 prx 中。

- Term Dictionary (tii, tis)
  - Frequencies (.frq)
  - Positions (.prx)

4.2.1. 词典(tis)及词典索引(tii)信息



在词典中，所有的词是按照字典顺序排序的。

- 词典文件(tis)
  - TermCount：词典中包含的总的词数
  - IndexInterval：为了加快对词的查找速度，也应用类似跳跃表的结构，假设 IndexInterval 为 4，则在词典索引(tii)文件中保存第 4 个，第 8 个，第 12 个词，这

样可以加快在词典文件中查找词的速度。

- SkipInterval：倒排表无论是文档号及词频，还是位置信息，都是以跳跃表的结构存在的，SkipInterval 是跳跃的步数。
- MaxSkipLevels：跳跃表是多层的，这个值指的是跳跃表的最大层数。
- TermCount 个项的数组，每一项代表一个词，对于每一个词，以前缀后缀规则存放词的文本信息(PrefixLength + Suffix)，词属于的域的域号(FieldNum)，有多少篇文档包含此词(DocFreq)，此词的倒排表在 frq，prx 中的偏移量(FreqDelta, ProxDelta)，此词的倒排表的跳跃表在 frq 中的偏移量(SkipDelta)，这里之所以用 Delta，是应用差值规则。

- 词典索引文件(tii)
  - 词典索引文件是为了加快对词典文件中词的查找速度，保存每隔 IndexInterval 个词。
  - 词典索引文件是会被全部加载到内存中去的。
  - IndexTermCount = TermCount / IndexInterval：词典索引文件中包含的词数。
  - IndexInterval 同词典文件中的 IndexInterval。
  - SkipInterval 同词典文件中的 SkipInterval。
  - MaxSkipLevels 同词典文件中的 MaxSkipLevels。
  - IndexTermCount 个项的数组，每一项代表一个词，每一项包括两部分，第一部分是词本身(TermInfo)，第二部分是在词典文件中的偏移量(IndexDelta)。假设 IndexInterval 为 4，此数组中保存第 4 个，第 8 个，第 12 个词。。。

读取词典及词典索引文件的代码如下：

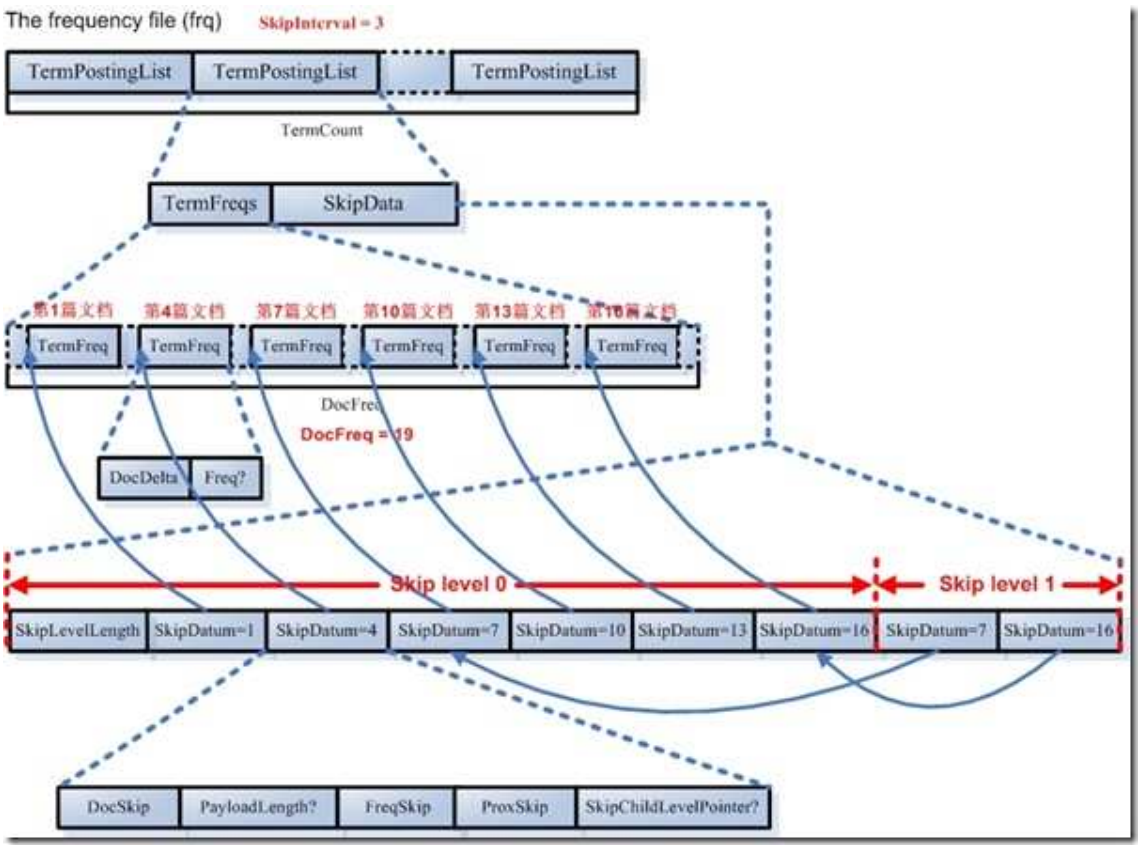
```
origEnum = new SegmentTermEnum(directory.openInput(segment + "." + IndexFileNames.TERMS_EXTENSION,readBufferSize), fieldInfos, false);//用于读取 tis 文件
int firstInt = input.readInt();
size = input.readLong();
indexInterval = input.readInt();
skipInterval = input.readInt();
```

- maxSkipLevels = input.readInt();

SegmentTermEnum indexEnum = new SegmentTermEnum(directory.openInput(segment + "." + IndexFileNames.TERMS_INDEX_EXTENSION, readBufferSize), fieldInfos, true);//用于读取 tii 文件

- indexTerms = new Term[indexSize];
- indexInfos = new TermInfo[indexSize];
- indexPointers = new long[indexSize];
- for (int i = 0; indexEnum.next(); i++)
  - indexTerms[i] = indexEnum.term();
  - indexInfos[i] = indexEnum.termInfo();
  - indexPointers[i] = indexEnum.indexPointer;

4.2.2. 文档号及词频(frq)信息



文档号及词频文件里面保存的是倒排表，是以跳跃表形式存在的。

- 此文件包含 TermCount 个项，每一个词都有一项，因为每一个词都有自己的倒排表。
- 对于每一个词的倒排表都包括两部分，一部分是倒排表本身，也即一个数组的文档号及词频，另一部分是跳跃表，为了更快的访问和定位倒排表中文档号及词频的位置。
- 对于文档号和词频的存储应用的是差值规则和或然跟随规则，Lucene 的文档本身有以下几句话，比较难以理解，在此解释一下：

For example, the TermFreqs for a term which occurs once in document seven and three times in document eleven, with omitTf false, would be the following sequence of VInts:

15, 8, 3

If omitTf were true it would be this sequence of VInts instead:

7,4

首先我们看 omitTf=false 的情况，也即我们在索引中会存储一个文档中 term 出现的次数。

例子中说了，表示在文档 7 中出现 1 次，并且又在文档 11 中出现 3 次的文档用以下序列表示：

15，8，3.

那这三个数字是怎么计算出来的呢？

首先，根据定义 TermFreq --> DocDelta[, Freq?]，一个 TermFreq 结构是由一个 DocDelta 后面或许跟着 Freq 组成，也即上面我们说的 A+B？结构。

DocDelta 自然是想存储包含此 Term 的文档的 ID 号了，Freq 是在此文档中出现的次数。

所以根据例子，应该存储的完整信息为[DocID = 7, Freq = 1] [DocID = 11， Freq = 3](见全文检索的基本原理章节)。

然而为了节省空间，Lucene 对编号此类的数据都是用差值来表示的，也即上面说的规则 2，Delta 规则，于是文档 ID 就不能按完整信息存了，就应该存放如下：

[DocIDDelta = 7, Freq = 1][DocIDDelta = 4 (11-7), Freq = 3]

然而 Lucene 对于 A+B?这种或然跟随的结果，有其特殊的存储方式，见规则 3，即 A+B?规则，如果 DocDelta 后面跟随的 Freq 为 1，则用 DocDelta 最后一位置 1 表示。

如果 DocDelta 后面跟随的 Freq 大于 1，则 DocDelta 得最后一位置 0，然后后面跟随真正的值，从而对于第一个 Term，由于 Freq 为 1，于是放在 DocDelta 的最后一位表示，DocIDDelta = 7 的二进制是 000 0111，必须要左移一位，且最后一位置一，000 1111 = 15，对于第二个 Term，由于 Freq 大于一，于是放在 DocDelta 的最后一位置零，DocIDDelta = 4 的二进制是 0000 0100，

必须要左移一位，且最后一位置零，0000 1000 = 8，然后后面跟随真正的 Freq = 3。

于是得到序列：[DocDleta = 15][DocDelta = 8, Freq = 3]，也即序列，15，8，3。

如果 omitTf=true，也即我们不在索引中存储一个文档中 Term 出现的次数，则只存 DocID 就可以了，因而不存在 A+B?规则的应用。

[DocID = 7][DocID = 11]，然后应用规则 2，Delta 规则，于是得到序列[DocDelta = 7][DocDelta = 4 (11 - 7)]，也即序列，7，4.

- 对于跳跃表的存储有以下几点需要解释一下：
  - 跳跃表可根据倒排表本身的长度(DocFreq)和跳跃的幅度(SkipInterval)而分不同的层次，层次数为 NumSkipLevels = Min(MaxSkipLevels, floor(log(DocFreq/log(SkipInterval)))).
  - 第 Level 层的节点数为 DocFreq/(SkipInterval^(Level + 1))，level 从零计数。
  - 除了最高层之外，其他层都有 SkipLevelLength 来表示此层的二进制长度(而非节点的个数)，方便读取某一层的跳跃表到缓存里面。
  - 低层在前，高层在后，当读完所有的低层后，剩下的就是最后一层，因而最后一层不需要 SkipLevelLength。这也是为什么 Lucene 文档中的格式描述为<SkipLevelLength, SkipLevel> $^{\text{NumSkipLevels-1}}$, SkipLevel，也即低 NumSKipLevels-1 层有 SkipLevelLength，最后一层只有 SkipLevel，没有 SkipLevelLength。
  - 除最低层以外，其他层都有 SkipChildLevelPointer 来指向下一层相应的节点。
  - 每一个跳跃节点包含以下信息：文档号，payload 的长度，文档号对应的倒排表中的节点在 frq 中的偏移量，文档号对应的倒排表中的节点在 prx 中的偏移量。
  - 虽然 Lucene 的文档中有以下的描述，然而实验的结果却不是完全准确的：

Example: SkipInterval = 4, MaxSkipLevels = 2, DocFreq = 35. Then skip level 0 has 8 SkipData entries, containing the 3rd, 7th, 11th, 15th, 19th, 23rd, 27th, and 31st document numbers in TermFreqs. Skip level 1 has 2 SkipData entries, containing the 15th and 31st document numbers in TermFreqs.

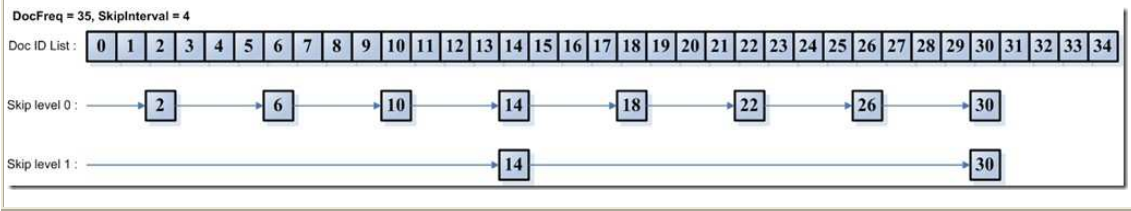
按照描述，当 SkipInterval 为 4，且有 35 篇文档的时候，Skip level = 0 应该包括第 3，第 7，第 11，第 15，第 19，第 23，第 27，第 31 篇文档，Skip level = 1 应该包括第 15，第 31 篇文档。然而真正的实现中，跳跃表节点的时候，却向前偏移了，偏移的原因在于下面的代码：

- FormatPostingsDocsWriter.addDoc(int docID, int termDocFreq)

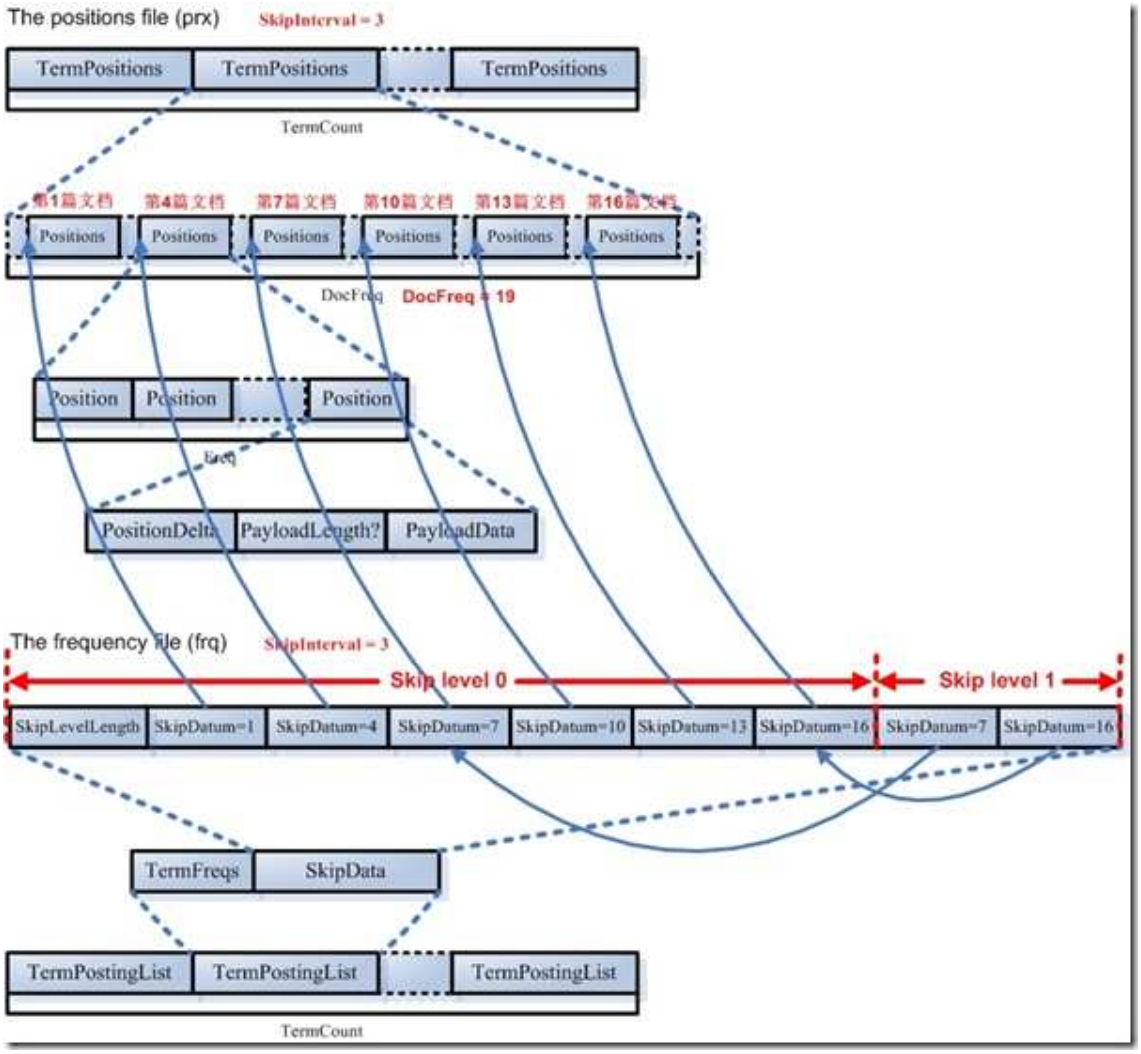
- final int delta = docID - lastDocID;
- if ((++df % skipInterval) == 0)
  - skipListWriter.setSkipData(lastDocID, storePayloads, posWriter.lastPayloadLength);
  - skipListWriter.bufferSkip(df);

从代码中，我们可以看出，当 SkipInterval 为 4 的时候，当 docID = 0 时，++df 为 1，1%4 不为 0，不是跳跃节点，当 docID = 3 时，++df=4，4%4 为 0，为跳跃节点，然而 skipData 里面保存的却是 lastDocID 为 2。

所以真正的倒排表和跳跃表中保存一下的信息：



4.2.3. 词位置(prx)信息



词位置信息也是倒排表，也是以跳跃表形式存在的。

- 此文件包含 TermCount 个项，每一个词都有一项，因为每一个词都有自己的词位置倒排表。
- 对于每一个词的都有一个 DocFreq 大小的数组，每项代表一篇文档，记录此文档中此词出现的位置。这个文档数组也是和 frq 文件中的跳跃表有关系的，从上面我们知道，在 frq 的跳跃表节点中有 ProxSkip，当 SkipInterval 为 3 的时候，frq 的跳跃表节点指向 prx 文件中的此数组中的第 1，第 4，第 7，第 10，第 13，第 16 篇文档。
- 对于每一篇文档，可能包含一个词多次，因而有一个 Freq 大小的数组，每一项代表此

词在此文档中出现一次，则有一个位置信息。

- 每一个位置信息包含：PositionDelta(采用差值规则)，还可以保存 payload，应用或然跟随规则。

4.3. 其他信息

4.3.1. 标准化因子文件(nrm)

为什么会有标准化因子呢？从第一章中的描述，我们知道，在搜索过程中，搜索出的文档要按与查询语句的相关性排序，相关性大的打分(score)高，从而排在前面。相关性打分(score)使用向量空间模型(Vector Space Model)，在计算相关性之前，要计算 Term Weight，也即某 Term 相对于某 Document 的重要性。在计算 Term Weight 时，主要有两个影响因素，一个是此 Term 在此文档中出现的次数，一个是此 Term 的普通程度。显然此 Term 在此文档中出现的次数越多，此 Term 在此文档中越重要。

这种 Term Weight 的计算方法是最普通的，然而存在以下几个问题：

- 不同的文档重要性不同。有的文档重要些，有的文档相对不重要，比如对于做软件的，在索引书籍的时候，我想让计算机方面的书更容易搜到，而文学方面的书籍搜索时排名靠后。
- 不同的域重要性不同。有的域重要一些，如关键字，如标题，有的域不重要一些，如附件等。同样一个词(Term)，出现在关键字中应该比出现在附件中打分要高。
- 根据词(Term)在文档中出现的绝对次数来决定此词对文档的重要性，有不合理的地方。比如长的文档词在文档中出现的次数相对较多，这样短的文档比较吃亏。比如一个词在一本砖头书中出现了 10 次，在另外一篇不足 100 字的文章中出现了 9 次，就说明砖头书应该排在前面码？不应该，显然此词在不足 100 字的文章中能出现 9 次，可见其对此文章的重要性。

由于以上原因，Lucene 在计算 Term Weight 时，都会乘上一个标准化因子(Normalization Factor)，来减少上面三个问题的影响。

标准化因子(Normalization Factor)是会影响随后打分(score)的计算的，Lucene 的打分计算一部分发生在索引过程中，一般是与查询语句无关的参数如标准化因子，大部分发生在搜索过程

中，会在搜索过程的代码分析中详述。

标准化因子(Normalization Factor)在索引过程总的计算如下：

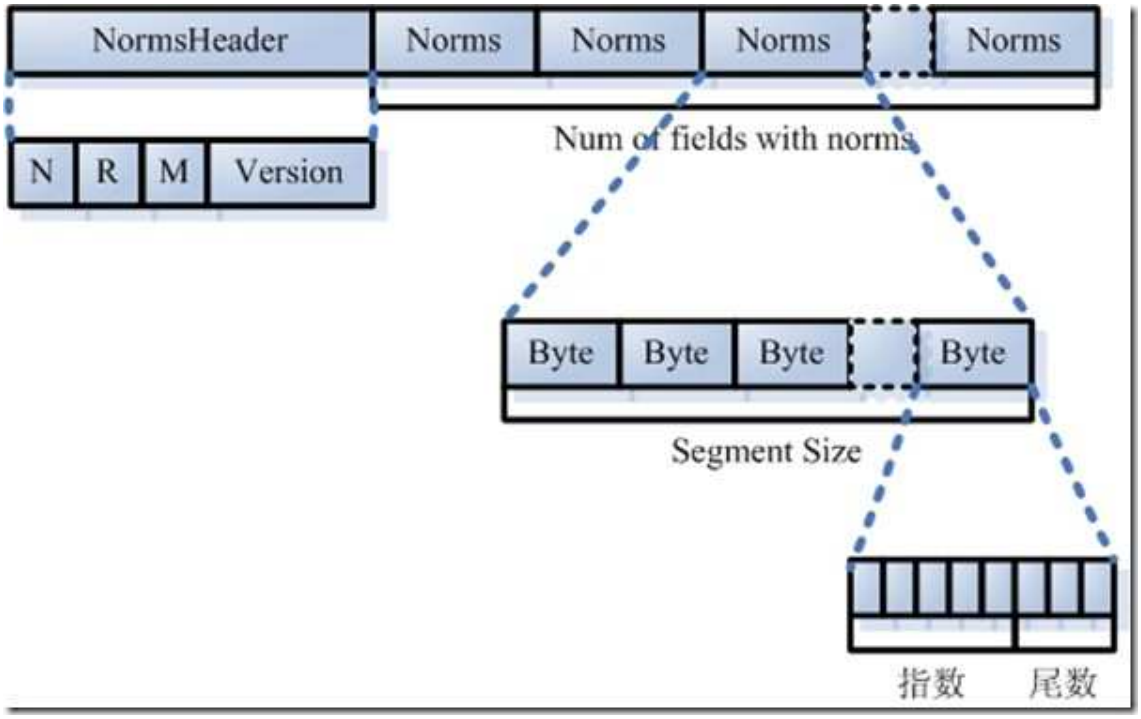
$$norm(t,d) = doc.getBoost() \cdot lengthNorm(field) \cdot \prod_{field\ f\ in\ d\ named\ as\ t} f.getBoost()$$

它包括三个参数：

- Document boost：此值越大，说明此文档越重要。
- Field boost：此域越大，说明此域越重要。
- lengthNorm(field) = (1.0 / Math.sqrt(numTerms))：一个域中包含的 Term 总数越多，也即文档越长，此值越小，文档越短，此值越大。

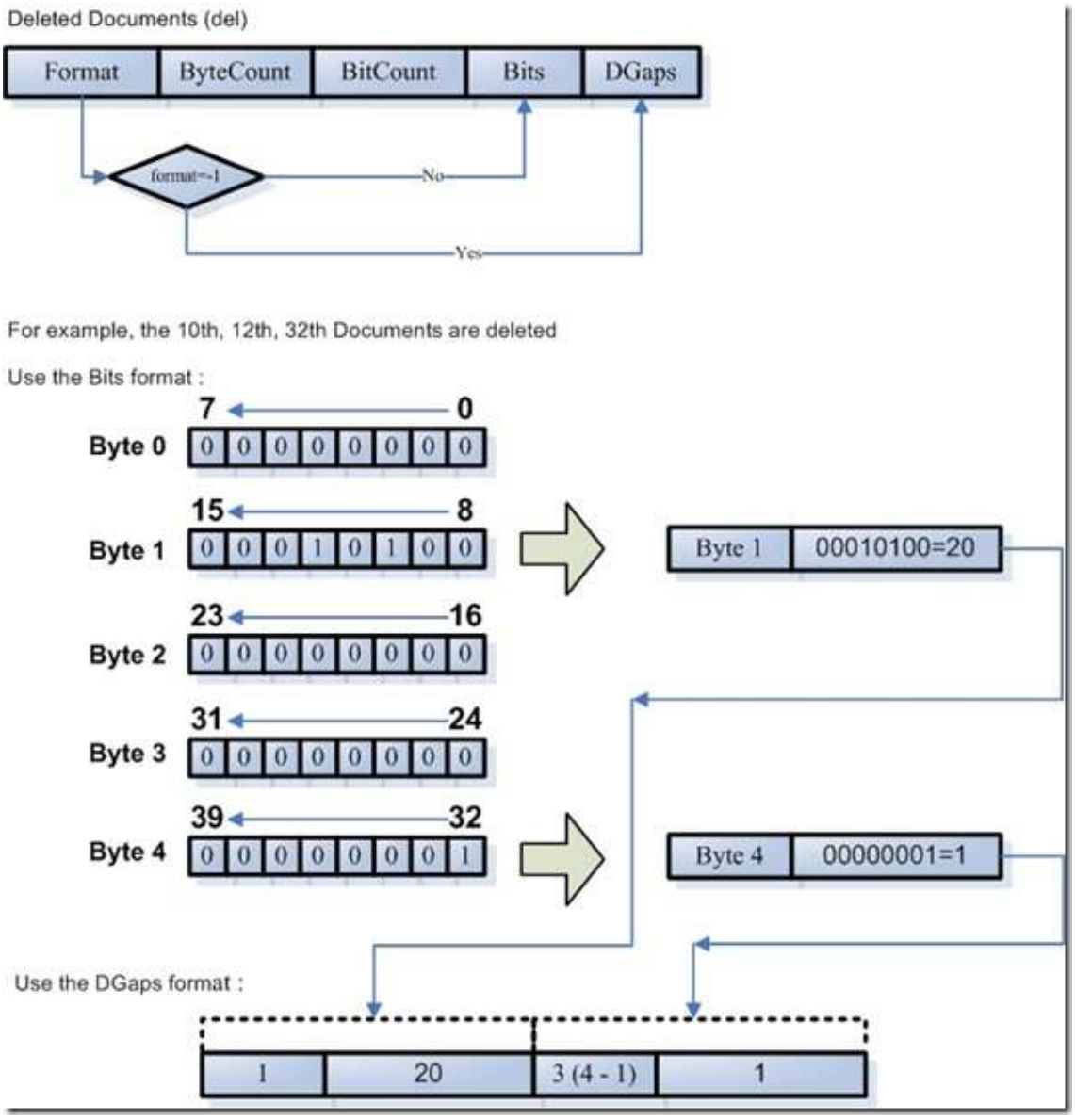
从上面的公式，我们知道，一个词(Term)出现在不同的文档或不同的域中，标准化因子不同。比如有两个文档，每个文档有两个域，如果不考虑文档长度，就有四种排列组合，在重要文档的重要域中，在重要文档的非重要域中，在非重要文档的重要域中，在非重要文档的非重要域中，四种组合，每种有不同的标准化因子。

于是在 Lucene 中，标准化因子共保存了(文档数目乘以域数目)个，格式如下：



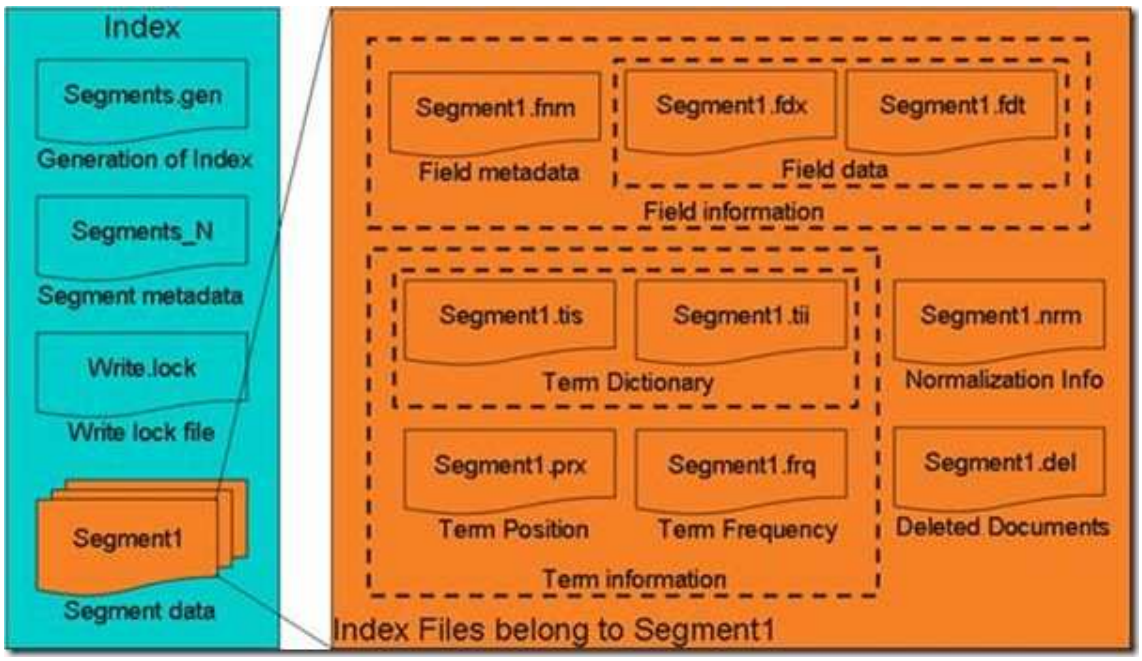
- 标准化因子文件(Normalization Factor File: nrm)：
  - NormsHeader：字符串“NRM”外加 Version，依 Lucene 的版本的不同而不同。
  - 接着是一个数组，大小为 NumFields，每个 Field 一项，每一项为一个 Norms。
  - Norms 也是一个数组，大小为 SegSize，即此段中文档的数量，每一项为一个 Byte，表示一个浮点数，其中 0~2 为尾数，3~8 为指数。

4.3.2. 删除文档文件(del)



- 被删除文档文件(Deleted Document File: .del)
  - Format：在此文件中，Bits 和 DGaps 只能保存其中之一，-1 表示保存 DGaps，非负值表示保存 Bits。
  - ByteCount：此段中有多少文档，就有多少个 bit 被保存，但是以 byte 形式计数，也即 Bits 的大小应该是 byte 的倍数。
  - BitCount：Bits 中有多少位被至 1，表示此文档已经被删除。
  - Bits：一个数组的 byte，大小为 ByteCount，应用时被认为是 byte*8 个 bit。
  - DGaps：如果删除的文档数量很小，则 Bits 大部分位为 0，很浪费空间。DGaps 采用以下的方式来保存稀疏数组：比如第十，十二，三十二个文档被删除，于是第十，十二，三十二位设为 1，DGaps 也是以 byte 为单位的，仅保存不为 0 的 byte，如第 1 个 byte，第 4 个 byte，第 1 个 byte 十进制为 20，第 4 个 byte 十进制为 1。于是保存成 DGaps，第 1 个 byte，位置 1 用不定长正整数保存，值为 20 用二进制保存，第 2 个 byte，位置 4 用不定长正整数保存，用差值为 3，值为 1 用二进制保存，二进制数据不用差值表示。

五、总体结构



- 图示为 Lucene 索引文件的整体结构：
  - 属于整个索引(Index)的 segment.gen，segment_N，其保存的是段(segment)的元数据信息，然后分多个 segment 保存数据信息，同一个 segment 有相同的前缀文件名。
  - 对于每一个段，包含域信息，词信息，以及其他信息(标准化因子，删除文档)
  - 域信息也包括域的元数据信息，在 fnm 中，域的数据信息，在 fdx，fdt 中。
  - 词信息是反向信息，包括词典(tis, tii)，文档号及词频倒排表(frq)，词位置倒排表(prx)。
  - 大家可以通过看源代码，相应的 Reader 和 Writer 来了解文件结构，将更为透彻。

第四章：Lucene 索引过程分析

对于 Lucene 的索引过程，除了将词(Term)写入倒排表并最终写入 Lucene 的索引文件外，还包括分词(Analyzer)和合并段(merge segments)的过程，本次不包括这两部分，将在以后的文章中进行分析。

Lucene 的索引过程，很多的博客，文章都有介绍，推荐大家上网搜一篇文章：《Annotated Lucene》，好像中文名称叫《Lucene 源码剖析》是很不错的。

想要真正了解 Lucene 索引文件过程，最好的办法是跟进代码调试，对着文章看代码，这样不但能够最详细准确的掌握索引过程(描述都是有偏差的，而代码是不会骗你的)，而且还能够学习 Lucene 的一些优秀的实现，能够在以后的工作中为我所用，毕竟 Lucene 是比较优秀的开源项目之一。

由于 Lucene 已经升级到 3.0.0 了，本索引过程为 Lucene 3.0.0 的索引过程。

一、索引过程体系结构

Lucene 3.0 的搜索要经历一个十分复杂的过程，各种信息分散在不同的对象中分析，处理，写入，为了支持多线程，每个线程都创建了一系列类似结构的对象集，为了提高效率，要复用一些对象集，这使得索引过程更加复杂。

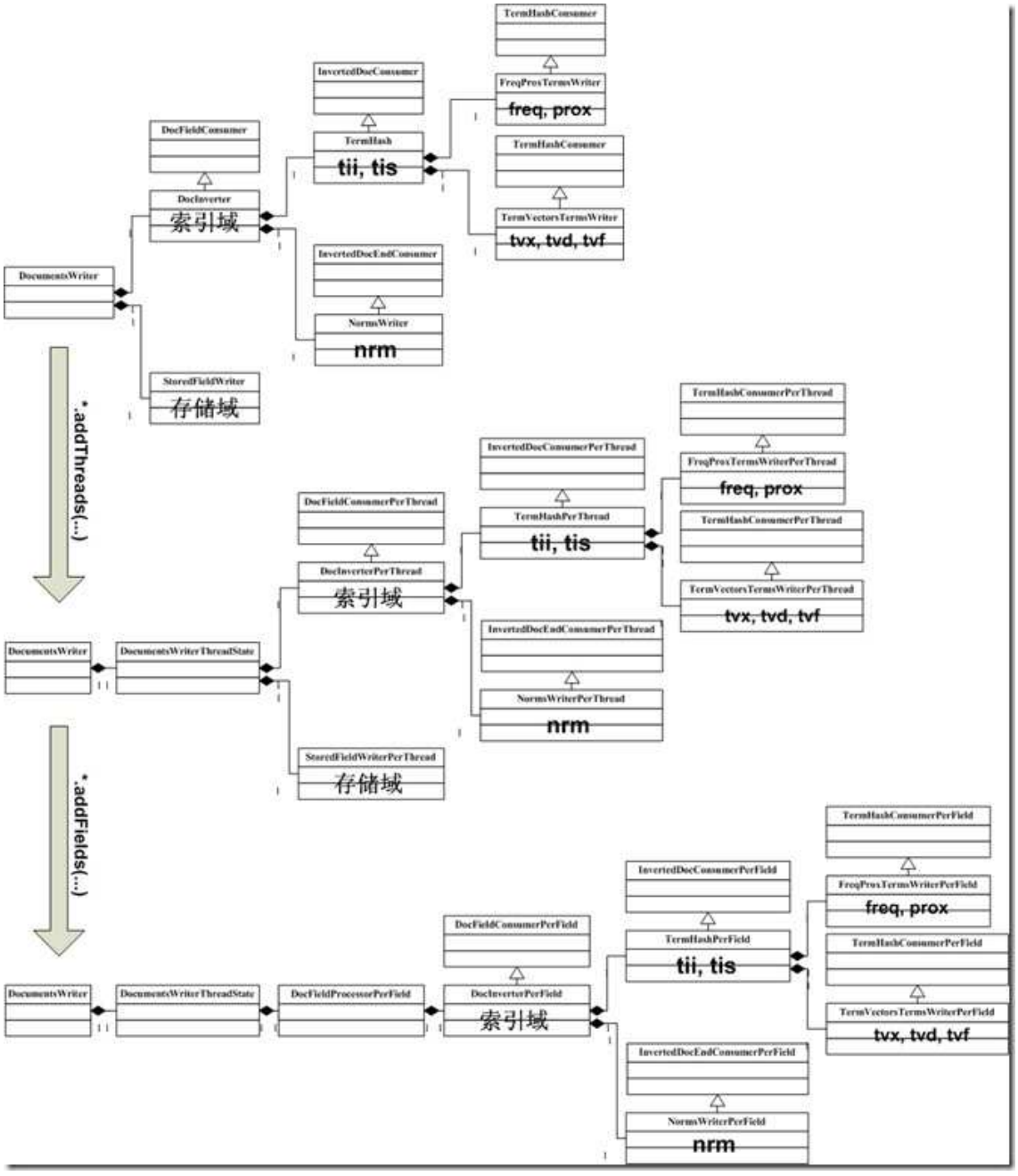
其实索引过程，就是经历下图中所示的索引链的过程，索引链中的每个节点，负责索引文档的不同部分的信息 ，当经历完所有的索引链的时候，文档就处理完毕了。最初的索引链，我们称之**基本索引链**。

为了支持多线程，使得多个线程能够并发处理文档，因而每个线程都要建立自己的索引链体系，使得每个线程能够独立工作，在基本索引链基础上建立起来的每个线程独立的索引链体系，我们称之**线程索引链**。线程索引链的每个节点是由基本索引链中的相应的节点调用函数 addThreads 创建的。

为了提高效率，考虑到对相同域的处理有相似的过程，应用的缓存也大致相当，因而不必每个线程在处理每一篇文档的时候都重新创建一系列对象，而是复用这些对象。所以对每个域也建立了自己的索引链体系，我们称之**域索引链**。域索引链的每个节点是由线程索引链中的相应的节点调用 addFields 创建的。

当完成对文档的处理后，各部分信息都要写到索引文件中，写入索引文件的过程是同步的，不是多线程的，也是沿着基本索引链将各部分信息依次写入索引文件的。

下面详细分析这一过程。



二、详细索引过程

1、创建 IndexWriter 对象

代码：

```
IndexWriter writer = new IndexWriter(FSDirectory.open(INDEX_DIR), new StandardAnalyzer(Version.LUCENE_CURRENT), true, IndexWriter.MaxFieldLength.LIMITED);
```

IndexWriter 对象主要包含以下几方面的信息：

- 用于索引文档
  - Directory directory; 指向索引文件夹
  - Analyzer analyzer; 分词器
  - Similarity similarity = Similarity.getDefault(); 影响打分的标准化因子(normalization factor)部分，对文档的打分分两个部分，一部分是索引阶段计算的，与查询语句无关，一部分是搜索阶段计算的，与查询语句相关。
  - SegmentInfos segmentInfos = new SegmentInfos(); 保存段信息，大家会发现，和 segments_N 中的信息几乎一一对应。
  - IndexFileDeleter deleter; 此对象不是用来删除文档的，而是用来管理索引文件的。
  - Lock writeLock; 每一个索引文件夹只能打开一个 IndexWriter，所以需要锁。
  - Set<SegmentInfo> segmentsToOptimize = new HashSet<SegmentInfo>(); 保存正在最优化(optimize)的段信息。当调用 optimize 的时候，当前所有的段信息加入此 Set，此后新生成的段并不参与此次最优化。
- 用于合并段，在合并段的文章中将详细描述
  - SegmentInfos localRollbackSegmentInfos;
  - HashSet<SegmentInfo> mergingSegments = new HashSet<SegmentInfo>();
  - MergePolicy mergePolicy = new LogByteSizeMergePolicy(this);
  - MergeScheduler mergeScheduler = new ConcurrentMergeScheduler();
  - LinkedList<MergePolicy.OneMerge> pendingMerges = new LinkedList<MergePolicy.OneMerge>();

- Set<MergePolicy.OneMerge> runningMerges = new HashSet<MergePolicy.OneMerge>();
- List<MergePolicy.OneMerge> mergeExceptions = new ArrayList<MergePolicy.OneMerge>();
- long mergeGen;

- 为保持索引完整性，一致性和事务性
  - SegmentInfos rollbackSegmentInfos; 当 IndexWriter 对索引进行了添加，删除文档操作后，可以调用 commit 将修改提交到文件中去，也可以调用 rollback 取消从上次 commit 到此时的修改。
  - SegmentInfos localRollbackSegmentInfos; 此段信息主要用于将其他的索引文件夹合并到此索引文件夹的时候，为防止合并到一半出错可回滚所保存的原来的段信息。
- 一些配置
  - long writeLockTimeout; 获得锁的时间超时。当超时的时候，说明此索引文件夹已经被另一个 IndexWriter 打开了。
  - int termIndexInterval; 同 tii 和 tis 文件中的 indexInterval。

有关 SegmentInfos 对象所保存的信息：

- 当索引文件夹如下的时候，SegmentInfos 对象如下表


```
segmentInfos    SegmentInfos   (id=37)
   capacityIncrement    0
   counter    3
   elementCount    3
   elementData    Object[10]   (id=68)
     [0]    SegmentInfo   (id=166)
```

```
    delCount    0
    delGen    -1
    diagnostics    HashMap<K,V>   (id=170)
    dir    SimpleFSDirectory   (id=171)
    docCount    2
    docStoreIsCompoundFile    false
    docStoreOffset    -1
    docStoreSegment    null
    files    ArrayList<E>   (id=173)
    hasProx    true
    hasSingleNormFile    true
    isCompoundFile    1
    name    "_0"
    normGen    null
    preLockless    false
    sizeInBytes    635
  [1]    SegmentInfo   (id=168)
    delCount    0
    delGen    -1
    diagnostics    HashMap<K,V>   (id=177)
    dir    SimpleFSDirectory   (id=171)
    docCount    2
    docStoreIsCompoundFile    false
    docStoreOffset    -1
    docStoreSegment    null
    files    ArrayList<E>   (id=178)
    hasProx    true
    hasSingleNormFile    true
```

```
      isCompoundFile    1
      name    "_1"
      normGen    null
      preLockless    false
      sizeInBytes    635
    [2]    SegmentInfo   (id=169)
      delCount    0
      delGen    -1
      diagnostics    HashMap<K,V>   (id=180)
      dir    SimpleFSDirectory   (id=171)
      docCount    2
      docStoreIsCompoundFile    false
      docStoreOffset    -1
      docStoreSegment    null
      files    ArrayList<E>   (id=214)
      hasProx    true
      hasSingleNormFile    true
      isCompoundFile    1
      name    "_2"
      normGen    null
      preLockless    false
      sizeInBytes    635
  generation    4
  lastGeneration    4
  modCount    3
  pendingSegnOutput    null
  userData    HashMap<K,V>   (id=146)
  version    1263044890832
```

有关 IndexFileDeleter：

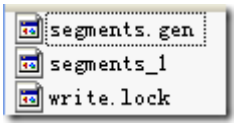
- 其不是用来删除文档的，而是用来管理索引文件的。
- 在对文档的添加，删除，对段的合并的处理过程中，会生成很多新的文件，并需要删除老的文件，因而需要管理。
- 然而要被删除的文件又可能在被用，因而要保存一个引用计数，仅仅当引用计数为零的时候，才执行删除。
- 下面这个例子能很好的说明 IndexFileDeleter 如何对文件引用计数并进行添加和删除的。

(1) 创建 IndexWriter 时

IndexWriter writer = new IndexWriter(FSDirectory.open(indexDir), new StandardAnalyzer(Version.LUCENE_CURRENT), true, IndexWriter.MaxFieldLength.LIMITED);

writer.setMergeFactor(3);

索引文件夹如下：



引用计数如下：

refCounts HashMap<K,V> (id=101)

size 1

table HashMap$Entry<K,V>[16] (id=105)

[8] HashMap$Entry<K,V> (id=110)

key "segments_1"

value IndexFileDeleter$RefCount (id=38)

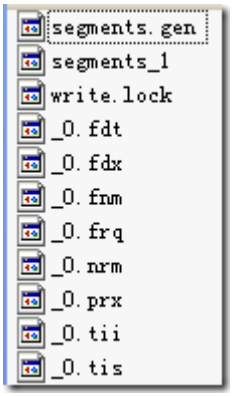
count 1

(2) 添加第一个段时

indexDocs(writer, docDir);

writer.commit();

首先生成的不是 compound 文件



因而引用计数如下：

refCounts HashMap<K,V> (id=101)

size 9

table HashMap$Entry<K,V>[16] (id=105)

[1] HashMap$Entry<K,V> (id=129)

key "_0.tis"

value IndexFileDeleter$RefCount (id=138)

count 1

[3] HashMap$Entry<K,V> (id=130)

key "_0.fnm"

value IndexFileDeleter$RefCount (id=141)

count 1

[4] HashMap$Entry<K,V> (id=134)

key "_0.tii"

value IndexFileDeleter$RefCount (id=142)

count 1

[8] HashMap$Entry<K,V> (id=135)

key "_0.frq"

value IndexFileDeleter$RefCount (id=143)

count 1

[10] HashMap$Entry<K,V> (id=136)

key "_0.fdx"

value IndexFileDeleter$RefCount (id=144)

count 1

[13] HashMap$Entry<K,V> (id=139)

key "_0.prx"

value IndexFileDeleter$RefCount (id=145)

count 1

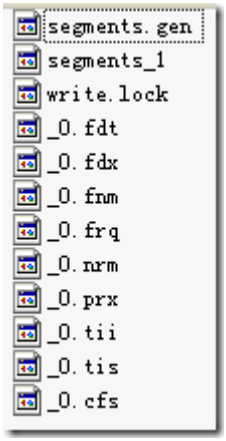
[14] HashMap$Entry<K,V> (id=140)

key "_0.fdt"

value IndexFileDeleter$RefCount (id=146)

count 1

然后会合并成 compound 文件，并加入引用计数



refCounts HashMap<K,V> (id=101)

size 10

table HashMap$Entry<K,V>[16] (id=105)

[1] HashMap$Entry<K,V> (id=129)

key "_0.tis"

value IndexFileDeleter$RefCount (id=138)

count 1

[2] HashMap$Entry<K,V> (id=154)

key "_0.cfs"

value IndexFileDeleter$RefCount (id=155)

count 1

[3] HashMap$Entry<K,V> (id=130)

key "_0.fnm"

value IndexFileDeleter$RefCount (id=141)

count 1

[4] HashMap$Entry<K,V> (id=134)

key "_0.tii"

value IndexFileDeleter$RefCount (id=142)

count 1

[8] HashMap$Entry<K,V> (id=135)

key "_0.frq"

value IndexFileDeleter$RefCount (id=143)

count 1

[10] HashMap$Entry<K,V> (id=136)

key "_0.fdx"

value IndexFileDeleter$RefCount (id=144)

count 1

[13] HashMap$Entry<K,V> (id=139)

key "_0.prx"

value IndexFileDeleter$RefCount (id=145)

count 1

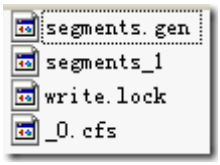
[14] HashMap$Entry<K,V> (id=140)

key "_0.fdt"

value IndexFileDeleter$RefCount (id=146)

count 1

然后会用 IndexFileDeleter.decRef()来删除[_0.nrm, _0.tis, _0.fnm, _0.tii, _0.frq, _0.fdx, _0.prx, _0.fdt]文件



refCounts HashMap<K,V> (id=101)

size 2

table HashMap$Entry<K,V>[16] (id=105)

[2] HashMap$Entry<K,V> (id=154)

key "_0.cfs"

value IndexFileDeleter$RefCount (id=155)

count 1

[8] HashMap$Entry<K,V> (id=110)

key "segments_1"

value IndexFileDeleter$RefCount (id=38)

count 1

然后为建立新的 segments_2

refCounts HashMap<K,V> (id=77)

size 3

table HashMap$Entry<K,V>[16] (id=84)

[2] HashMap$Entry<K,V> (id=87)

key "_0.cfs"

value IndexFileDeleter$RefCount (id=91)

count 3

[8] HashMap$Entry<K,V> (id=89)

key "segments_1"

value IndexFileDeleter$RefCount (id=62)

count 0

[9] HashMap$Entry<K,V> (id=90)

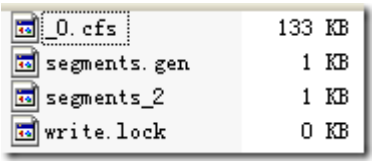
key "segments_2"

next null

value IndexFileDeleter$RefCount (id=93)

count 1

然后 IndexFileDeleter.decRef() 删除 segments_1 文件



refCounts HashMap<K,V> (id=77)

size 2

table HashMap$Entry<K,V>[16] (id=84)

[2] HashMap$Entry<K,V> (id=87)

key "_0.cfs"

value IndexFileDeleter$RefCount (id=91)

count 2

[9] HashMap$Entry<K,V> (id=90)

key "segments_2"

value IndexFileDeleter$RefCount (id=93)

count 1

(3) 添加第二个段

indexDocs(writer, docDir);

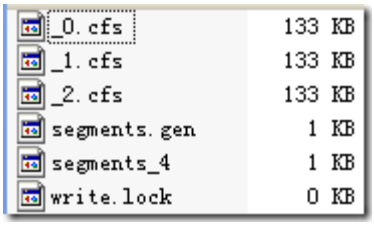
writer.commit();

(4) 添加第三个段，由于 MergeFactor 为 3，则会进行一次段合并。

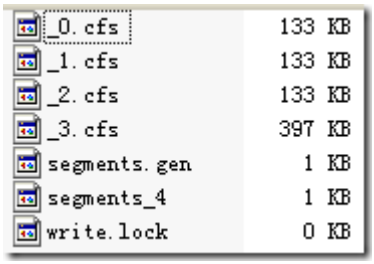
indexDocs(writer, docDir);

writer.commit();

首先和其他的段一样，生成_2.cfs 以及 segments_4



同时创建了一个线程来进行背后进行段合并(ConcurrentMergeScheduler$MergeThread.run())



这时候的引用计数如下

refCounts HashMap<K,V> (id=84)

size 5

table HashMap$Entry<K,V>[16] (id=98)

[2] HashMap$Entry<K,V> (id=112)

key "_0.cfs"

value IndexFileDeleter$RefCount (id=117)

count 1

[4] HashMap$Entry<K,V> (id=113)

key "_3.cfs"

value IndexFileDeleter$RefCount (id=118)

count 1

[12] HashMap$Entry<K,V> (id=114)

key "_1.cfs"

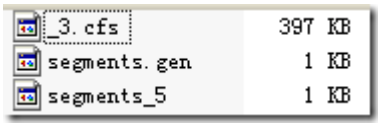
value IndexFileDeleter$RefCount (id=119)

```
        count    1
    [13]    HashMap$Entry<K,V>   (id=115)
      key    "_2.cfs"
      value    IndexFileDeleter$RefCount   (id=120)
        count    1
    [15]    HashMap$Entry<K,V>   (id=116)
      key    "segments_4"
      value    IndexFileDeleter$RefCount   (id=121)
        count    1
```

(5) 关闭 writer

writer.close();

通过 IndexFileDeleter.decRef()删除被合并的段



有关 SimpleFSLock 进行 JVM 之间的同步：

- 有时候，我们写 java 程序的时候，也需要不同的 JVM 之间进行同步，来保护一个整个系统中唯一的资源。
- 如果唯一的资源仅仅在一个进程中，则可以使用线程同步的机制
- 然而如果唯一的资源要被多个进程进行访问，则需要进程间同步的机制，无论是 Windows 和 Linux 在操作系统层面都有很多的进程间同步的机制。
- 但进程间的同步却不是 Java 的特长，Lucene 的 SimpleFSLock 给我们提供了一种方式。

Lock 的抽象类

```
public abstract class Lock {
  public static long LOCK_POLL_INTERVAL = 1000;
  public static final long LOCK_OBTAIN_WAIT_FOREVER = -1;
  public abstract boolean obtain() throws IOException;
  public boolean obtain(long lockWaitTimeout) throws LockObtainFailedException, IOException {
    boolean locked = obtain();
```

```
    if (lockWaitTimeout < 0 && lockWaitTimeout != LOCK_OBTAIN_WAIT_FOREVER)
      throw new IllegalArgumentException("...");
    long maxSleepCount = lockWaitTimeout / LOCK_POLL_INTERVAL;
    long sleepCount = 0;
    while (!locked) {
      if (lockWaitTimeout != LOCK_OBTAIN_WAIT_FOREVER && sleepCount++ >= maxSleepCount) {
        throw new LockObtainFailedException("Lock obtain timed out.");
      }
      try {
        Thread.sleep(LOCK_POLL_INTERVAL);
      } catch (InterruptedException ie) {
        throw new ThreadInterruptedException(ie);
      }
      locked = obtain();
    }
    return locked;
  }
  public abstract void release() throws IOException;
  public abstract boolean isLocked() throws IOException;
}
```

LockFactory 的抽象类

```
public abstract class LockFactory {
  public abstract Lock makeLock(String lockName);
  abstract public void clearLock(String lockName) throws IOException;
}
```

SimpleFSLock 的实现类

```
class SimpleFSLock extends Lock {
  File lockFile;
```

```
  File lockDir;
  public SimpleFSLock(File lockDir, String lockFileName) {
    this.lockDir = lockDir;
    lockFile = new File(lockDir, lockFileName);
  }
  @Override
  public boolean obtain() throws IOException {
    if (!lockDir.exists()) {
      if (!lockDir.mkdirs())
        throw new IOException("Cannot create directory: " + lockDir.getAbsolutePath());
    } else if (!lockDir.isDirectory()) {
        throw new IOException("Found regular file where directory expected: " +
lockDir.getAbsolutePath());
    }
    return lockFile.createNewFile();
  }
  @Override
  public void release() throws LockReleaseFailedException {
    if (lockFile.exists() && !lockFile.delete())
      throw new LockReleaseFailedException("failed to delete " + lockFile);
  }
  @Override
  public boolean isLocked() {
    return lockFile.exists();
  }
}
```

SimpleFSLockFactory 的实现类

```
public class SimpleFSLockFactory extends FSLockFactory {
```

```
  public SimpleFSLockFactory(String lockDirName) throws IOException {
    setLockDir(new File(lockDirName));
  }
  @Override
  public Lock makeLock(String lockName) {
    if (lockPrefix != null) {
      lockName = lockPrefix + "-" + lockName;
    }
    return new SimpleFSLock(lockDir, lockName);
  }
  @Override
  public void clearLock(String lockName) throws IOException {
    if (lockDir.exists()) {
      if (lockPrefix != null) {
        lockName = lockPrefix + "-" + lockName;
      }
      File lockFile = new File(lockDir, lockName);
      if (lockFile.exists() && !lockFile.delete()) {
        throw new IOException("Cannot delete " + lockFile);
      }
    }
  }
};
```

2、创建文档 Document 对象，并加入域(Field)

代码：

```
Document doc = new Document();
doc.add(new Field("path", f.getPath(), Field.Store.YES, Field.Index.NOT_ANALYZED));
doc.add(new Field("modified",DateTools.timeToString(f.lastModified(),
DateTools.Resolution.MINUTE), Field.Store.YES, Field.Index.NOT_ANALYZED));
doc.add(new Field("contents", new FileReader(f)));
```

Document 对象主要包括以下部分：

- 此文档的 boost，默认为 1，大于一说明比一般的文档更加重要，小于一说明更不重要。
- 一个 ArrayList 保存此文档所有的域
- 每一个域包括域名，域值，和一些标志位，和 fnm，fdx，fdt 中的描述相对应。

```
doc    Document   (id=42)
   boost    1.0
   fields    ArrayList<E>   (id=44)
      elementData    Object[10]   (id=46)
         [0]    Field   (id=48)
            binaryLength    0
            binaryOffset    0
            boost    1.0
            fieldsData    "exampledocs\\file01.txt"
            isBinary    false
            isIndexed    true
            isStored    true
            isTokenized    false
            lazy    false
            name    "path"
            omitNorms    false
            omitTermFreqAndPositions    false
            storeOffsetWithTermVector    false
            storePositionWithTermVector    false
```

```
    storeTermVector    false
    tokenStream    null
  [1]    Field   (id=50)
    binaryLength    0
    binaryOffset    0
    boost    1.0
    fieldsData    "200910240957"
    isBinary    false
    isIndexed    true
    isStored    true
    isTokenized    false
    lazy    false
    name    "modified"
    omitNorms    false
    omitTermFreqAndPositions    false
    storeOffsetWithTermVector    false
    storePositionWithTermVector    false
    storeTermVector    false
    tokenStream    null
  [2]    Field   (id=52)
    binaryLength    0
    binaryOffset    0
    boost    1.0
    fieldsData    FileReader   (id=58)
    isBinary    false
    isIndexed    true
    isStored    false
    isTokenized    true
```

```
        lazy    false
        name    "contents"
        omitNorms    false
        omitTermFreqAndPositions    false
        storeOffsetWithTermVector    false
        storePositionWithTermVector    false
        storeTermVector    false
        tokenStream    null
    modCount    3
    size    3
```

3、将文档加入 IndexWriter

代码：

```
writer.addDocument(doc);
-->IndexWriter.addDocument(Document doc, Analyzer analyzer)
    -->doFlush = docWriter.addDocument(doc, analyzer);
        --> DocumentsWriter.updateDocument(Document, Analyzer, Term)
注：--> 代表一级函数调用
```

IndexWriter 继而调用 DocumentsWriter.addDocument，其又调用 DocumentsWriter.updateDocument。

4、将文档加入 DocumentsWriter

代码：

```
DocumentsWriter.updateDocument(Document doc, Analyzer analyzer, Term delTerm)
-->(1) DocumentsWriterThreadState state = getThreadState(doc, delTerm);
```

```
-->(2) DocWriter perDoc = state.consumer.processDocument();
-->(3) finishDocument(state, perDoc);
```

DocumentsWriter 对象主要包含以下几部分：

- 用于写索引文件
  - IndexWriter writer;
  - Directory directory;
  - Similarity similarity：分词器
  - String segment：当前的段名，每当 flush 的时候，将索引写入以此为名称的段。

```
IndexWriter.doFlushInternal()
--> String segment = docWriter.getSegment();//return segment
--> newSegment = new SegmentInfo(segment,……);
--> docWriter.createCompoundFile(segment);//根据 segment 创建 cfs 文件。
```

  - String docStoreSegment：存储域所要写入的目标段。(在索引文件格式一文中已经详细描述)
  - int docStoreOffset：存储域在目标段中的偏移量。
  - int nextDocID：下一篇添加到此索引的文档 ID 号，对于同一个索引文件夹，此变量唯一，且同步访问。
  - DocConsumer consumer; 这是整个索引过程的核心，是 IndexChain 整个索引链的源头。

基本索引链：

对于一篇文档的索引过程，不是由一个对象来完成的，而是用对象组合的方式形成的一个处理链，链上的每个对象仅仅处理索引过程的一部分，称为索引链，由于后面还有其他的索引链，所以此处的索引链我称为基本索引链。

DocConsumer consumer 类型为 DocFieldProcessor，是整个索引链的源头，包含如下部分：

- 对索引域的处理
  - DocFieldConsumer consumer 类型为 DocInverter，包含如下部分
    - InvertedDocConsumer consumer 类型为 TermsHash，包含如下部分
      - TermsHashConsumer consumer 类型为 FreqProxTermsWriter，负责写 freq,

prox 信息

- TermsHash nextTermsHash
  - TermsHashConsumer consumer 类型为 TermVectorsTermsWriter，负责写 tvx, tvd, tvf 信息
  - InvertedDocEndConsumer endConsumer 类型为 NormsWriter，负责写 nrm 信息

- 对存储域的处理
  - FieldInfos fieldInfos = new FieldInfos();
  - StoredFieldsWriter fieldsWriter 负责写 fnm, fdt, fdx 信息

- 删除文档
  - BufferedDeletes deletesInRAM = new BufferedDeletes();
  - BufferedDeletes deletesFlushed = new BufferedDeletes();

类 BufferedDeletes 包含了一下的成员变量：

- HashMap<Term,Num> terms = new HashMap<Term,Num>();删除的词(Term)
- HashMap<Query,Integer> queries = new HashMap<Query,Integer>();删除的查询(Query)
- List<Integer> docIDs = new ArrayList<Integer>();删除的文档 ID
- long bytesUsed：用于判断是否应该对删除的文档写入索引文件。

由此可见，文档的删除主要有三种方式：

- IndexWriter.deleteDocuments(Term term)：所有包含此词的文档都会被删除。
- IndexWriter.deleteDocuments(Query query)：所有能满足此查询的文档都会被删除。
- IndexReader.deleteDocument(int docNum)：删除此文档 ID

删除文档既可以用 reader 进行删除，也可以用 writer 进行删除，不同的是，reader 进行删除后，此 reader 马上能够生效，而用 writer 删除后，会被缓存在 deletesInRAM 及 deletesFlushed 中，只有写入到索引文件中，当 reader 再次打开的时候，才能够看到。

那 deletesInRAM 和 deletesFlushed 各有什么用处呢？

此版本的 Lucene 对文档的删除是支持多线程的，当用 IndexWriter 删除文档的时候，都是缓存在 deletesInRAM 中的，直到 flush，才将删除的文档写入到索引文件中去，我们知道 flush 是需要一段时间的，那么在 flush 的过程中，另一个线程又有文档删除怎么办呢？

一般过程是这个样子的，当 flush 的时候，首先在同步(synchornized)的方法 pushDeletes 中，

将 deletesInRAM 全部加到 deletesFlushed 中，然后将 deletesInRAM 清空，退出同步方法，于是 flush 的线程程就向索引文件写 deletesFlushed 中的删除文档的过程，而与此同时其他线程新删除的文档则添加到新的 deletesInRAM 中去，直到下次 flush 才写入索引文件。

- 缓存管理
  - 为了提高索引的速度，Lucene 对很多的数据进行了缓存，使一起写入磁盘，然而缓存需要进行管理，何时分配，何时回收，何时写入磁盘都需要考虑。
  - ArrayList<char[]> freeCharBlocks = new ArrayList<char[]>();将用于缓存词(Term)信息的空闲块
  - ArrayList<byte[]> freeByteBlocks = new ArrayList<byte[]>();将用于缓存文档号(doc id)及词频(freq)，位置(prox)信息的空闲块。
  - ArrayList<int[]> freeIntBlocks = new ArrayList<int[]>();将存储某词的词频(freq)和位置(prox)分别在 byteBlocks 中的偏移量
  - boolean bufferIsFull;用来判断缓存是否满了，如果满了，则应该写入磁盘
  - long numBytesAlloc;分配的内存数量
  - long numBytesUsed;使用的内存数量
  - long freeTrigger;应该开始回收内存时的内存用量。
  - long freeLevel;回收内存应该回收到的内存用量。
  - long ramBufferSize;用户设定的内存用量。

缓存用量之间的关系如下：

```
DocumentsWriter.setRAMBufferSizeMB(double mb){
   ramBufferSize = (long) (mb*1024*1024);//用户设定的内存用量，当使用内存大于此时，开始写入磁盘
   waitQueuePauseBytes = (long) (ramBufferSize*0.1);
   waitQueueResumeBytes = (long) (ramBufferSize*0.05);
   freeTrigger = (long) (1.05 * ramBufferSize);//当分配的内存到达 105%的时候开始释放 freeBlocks 中的内存
   freeLevel = (long) (0.95 * ramBufferSize);//一直释放到 95%
}
```

```
DocumentsWriter.balanceRAM(){
   if (numBytesAlloc+deletesRAMUsed > freeTrigger) {
   //当分配的内存加删除文档所占用的内存大于 105%的时候，开始释放内存
      while(numBytesAlloc+deletesRAMUsed > freeLevel) {
      //一直进行释放，直到 95%
           //释放 free blocks
           byteBlockAllocator.freeByteBlocks.remove(byteBlockAllocator.freeByteBlocks.size()-1);
           numBytesAlloc -= BYTE_BLOCK_SIZE;
           freeCharBlocks.remove(freeCharBlocks.size()-1);
           numBytesAlloc -= CHAR_BLOCK_SIZE * CHAR_NUM_BYTE;
           freeIntBlocks.remove(freeIntBlocks.size()-1);
           numBytesAlloc -= INT_BLOCK_SIZE * INT_NUM_BYTE;
       }
    } else {
       if (numBytesUsed+deletesRAMUsed > ramBufferSize){
       //当使用的内存加删除文档占有的内存大于用户指定的内存时，可以写入磁盘
            bufferIsFull = true;
       }
    }
}
```

当判断是否应该写入磁盘时:

- 如果使用的内存大于用户指定内存时，bufferIsFull = true
- 当使用的内存加删除文档所占的内存加正在写入的删除文档所占的内存大于用户指定内存时 deletesInRAM.bytesUsed + deletesFlushed.bytesUsed + numBytesUsed) >= ramBufferSize
- 当删除的文档数目大于 maxBufferedDeleteTerms 时

```
DocumentsWriter.timeToFlushDeletes(){
   return (bufferIsFull || deletesFull()) && setFlushPending();
```

```
}
DocumentsWriter.deletesFull(){
    return (ramBufferSize != IndexWriter.DISABLE_AUTO_FLUSH &&
      (deletesInRAM.bytesUsed + deletesFlushed.bytesUsed + numBytesUsed) >= ramBufferSize) ||
      (maxBufferedDeleteTerms != IndexWriter.DISABLE_AUTO_FLUSH &&
      ((deletesInRAM.size() + deletesFlushed.size()) >= maxBufferedDeleteTerms));
}
```

- 多线程并发索引
  - 为了支持多线程并发索引，对每一个线程都有一个 DocumentsWriterThreadState，其为每一个线程根据 DocConsumer consumer 的索引链来创建每个线程的索引链(XXXPerThread)，来进行对文档的并发处理。
  - DocumentsWriterThreadState[] threadStates = new DocumentsWriterThreadState[0];
  - HashMap<Thread,DocumentsWriterThreadState> threadBindings = new HashMap<Thread,DocumentsWriterThreadState>();
  - 虽然对文档的处理过程可以并行，但是将文档写入索引文件却必须串行进行，串行写入的代码在 DocumentsWriter.finishDocument 中
  - WaitQueue waitQueue = new WaitQueue()
  - long waitQueuePauseBytes
  - long waitQueueResumeBytes

在 Lucene 中，文档是按添加的顺序编号的，DocumentsWriter 中的 nextDocID 就是记录下一个添加的文档 id。 当 Lucene 支持多线程的时候，就必须要有一个 synchornized 方法来付给文档 id 并且将 nextDocID 加一，这些是在 DocumentsWriter.getThreadState 这个函数里面做的。虽然给文档付 ID 没有问题了。但是由 Lucene 索引文件格式我们知道，文档是要按照 ID 的顺序从小到大写到索引文件中去的，然而不同的文档处理速度不同，当一个先来的线程一处理一篇需要很长时间的大文档时，另一个后来的线程二可能已经处理了很多小的文档了，但是这些后来小文档的 ID 号都大于第一个线程所处理的大文档，因而不能马上写到索引文件中去，而是放到 waitQueue 中，仅仅当大文档处理完了之后才写入索引文件。

waitQueue 中有一个变量 nextWriteDocID 表示下一个可以写入文件的 ID，当付给大文档 ID=4

时，则 nextWriteDocID 也设为 4，虽然后来的小文档 5，6，7，8 等都已处理结束，但是如下代码，

```
WaitQueue.add(){
    if (doc.docID == nextWriteDocID){
      ……
    } else {
       waiting[loc] = doc;
       waitingBytes += doc.sizeInBytes();
   }
   doPause()
}
```

则把 5, 6, 7, 8 放入 waiting 队列，并且记录当前等待的文档所占用的内存大小 waitingBytes。

当大文档 4 处理完毕后，不但写入文档 4，把原来等待的文档 5, 6, 7, 8 也一起写入。

```
WaitQueue.add(){
    if (doc.docID == nextWriteDocID) {
      writeDocument(doc);
      while(true) {
         doc = waiting[nextWriteLoc];
         writeDocument(doc);
      }
   } else {
      ……
   }
   doPause()
}
```

但是这存在一个问题：当大文档很大很大，处理的很慢很慢的时候，后来的线程二可能已经处理了很多的小文档了，这些文档都是在 waitQueue 中，则占有了越来越多的内存，长此以往，有内存不够的危险。

因而在 finishDocuments 里面，在 WaitQueue.add 最后调用了 doPause()函数

```
DocumentsWriter.finishDocument(){
    doPause = waitQueue.add(docWriter);
    if (doPause)
       waitForWaitQueue();
    notifyAll();
}
WaitQueue.doPause() {
    return waitingBytes > waitQueuePauseBytes;
}
```

当 waitingBytes 足够大的时候(为用户指定的内存使用量的 10%)，doPause 返回 true，于是后来的线程二会进入 wait 状态，不再处理另外的文档，而是等待线程一处理大文档结束。

当线程一处理大文档结束的时候，调用 notifyAll 唤醒等待他的线程。

```
DocumentsWriter.waitForWaitQueue() {
  do {
    try {
     wait();
    } catch (InterruptedException ie) {
     throw new ThreadInterruptedException(ie);
    }
  } while (!waitQueue.doResume());
}
WaitQueue.doResume() {
     return waitingBytes <= waitQueueResumeBytes;
}
```

当 waitingBytes 足够小的时候，doResume 返回 true，则线程二不用再 wait 了，可以继续处理另外的文档。

- 一些标志位

- int maxFieldLength：一篇文档中，一个域内可索引的最大的词(Term)数。
- int maxBufferedDeleteTerms：可缓存的最大的删除词(Term)数。当大于这个数的时候，就要写到文件中了。

此过程又包含如下三个子过程：

4.1、得到当前线程对应的文档集处理对象(DocumentsWriterThreadState)

代码为：

```
DocumentsWriterThreadState state = getThreadState(doc, delTerm);
```

在 Lucene 中，对于同一个索引文件夹，只能够有一个 IndexWriter 打开它，在打开后，在文件夹中，生成文件 write.lock，当其他 IndexWriter 再试图打开此索引文件夹的时候，则会报 org.apache.lucene.store.LockObtainFailedException 错误。

这样就出现了这样一个问题，在同一个进程中，对同一个索引文件夹，只能有一个 IndexWriter 打开它，因而如果想多线程向此索引文件夹中添加文档，则必须共享一个 IndexWriter，而且在以往的实现中，addDocument 函数是同步的(synchronized)，也即多线程的索引并不能起到提高性能的效果。

于是为了支持多线程索引，不使 IndexWriter 成为瓶颈，对于每一个线程都有一个相应的文档集处理对象(DocumentsWriterThreadState)，这样对文档的索引过程可以多线程并行进行，从而增加索引的速度。

getThreadState 函数是同步的(synchronized)，DocumentsWriter 有一个成员变量 threadBindings，它是一个 HashMap，键为线程对象(Thread.currentThread())，值为此线程对应的 DocumentsWriterThreadState 对象。

DocumentsWriterThreadState DocumentsWriter.getThreadState(Document doc, Term delTerm)包含如下几个过程：

- 根据当前线程对象，从 HashMap 中查找相应的 DocumentsWriterThreadState 对象，如果没找到，则生成一个新对象，并添加到 HashMap 中

```
DocumentsWriterThreadState state = (DocumentsWriterThreadState)
```

```
threadBindings.get(Thread.currentThread());
if (state == null) {
    ……
    state = new DocumentsWriterThreadState(this);
    ……
    threadBindings.put(Thread.currentThread(), state);
}
```

- 如果此线程对象正在用于处理上一篇文档，则等待，直到此线程的上一篇文档处理完。

```
DocumentsWriter.getThreadState() {
    waitReady(state);
    state.isIdle = false;
}

waitReady(state) {
    while (!state.isIdle) {wait();}
}
```

显然如果 state.isIdle 为 false，则此线程等待。

在一篇文档处理之前，state.isIdle = false 会被设定，而在一篇文档处理完毕之后，DocumentsWriter.finishDocument(DocumentsWriterThreadState perThread, DocWriter docWriter)中，会首先设定 perThread.isIdle = true; 然后 notifyAll()来唤醒等待此文档完成的线程，从而处理下一篇文档。

- 如果 IndexWriter 刚刚 commit 过，则新添加的文档要加入到新的段中(segment)，则首先要生成新的段名。

```
initSegmentName(false);
--> if (segment == null) segment = writer.newSegmentName();
```

- 将此线程的文档处理对象设为忙碌：state.isIdle = false;

4.2、用得到的文档集处理对象(DocumentsWriterThreadState)处理文档

代码为：

```
DocWriter perDoc = state.consumer.processDocument();
```

每一个文档集处理对象 DocumentsWriterThreadState 都有一个文档及域处理对象 DocFieldProcessorPerThread，它的成员函数 processDocument()被调用来对文档及域进行处理。

线程索引链(XXXPerThread):

由于要多线程进行索引，因而每个线程都要有自己的索引链，称为线程索引链。

线程索引链同**基本索引链**有相似的树形结构，由基本索引链中每个层次的对象调用 addThreads 进行创建的，负责每个线程的对文档的处理。

DocFieldProcessorPerThread 是线程索引链的源头，由 DocFieldProcessor.addThreads(…)创建 DocFieldProcessorPerThread 对象结构如下：

- 对索引域进行处理
  - DocFieldConsumerPerThread consumer 类型为 DocInverterPerThread，由 DocInverter.addThreads 创建
    - InvertedDocConsumerPerThread consumer 类型为 TermsHashPerThread，由 TermsHash.addThreads 创建
      - TermsHashConsumerPerThread consumer 类型为 FreqProxTermsWriterPerThread，由 FreqProxTermsWriter.addThreads 创建，负责每个线程的 freq，prox 信息处理
      - TermsHashPerThread nextPerThread
        - TermsHashConsumerPerThread consumer 类型 TermVectorsTermsWriterPerThread，由 TermVectorsTermsWriter 创建，负责每个线程的 tvx，tvd，tvf 信息处理
    - InvertedDocEndConsumerPerThread endConsumer 类型为 NormsWriterPerThread，由 NormsWriter.addThreads 创建，负责 nrm 信息的处理

- 对存储域进行处理
  - StoredFieldsWriterPerThread fieldsWriter 由 StoredFieldsWriter.addThreads 创建，负责 fnm，fdx，fdt 的处理。
  - FieldInfos fieldInfos;

DocumentsWriter.DocWriter DocFieldProcessorPerThread.processDocument()包含以下几个过程：

4.2.1、开始处理当前文档

```
consumer(DocInverterPerThread).startDocument();
fieldsWriter(StoredFieldsWriterPerThread).startDocument();
```

在此版的 Lucene 中，几乎所有的 XXXPerThread 的类，都有 startDocument 和 finishDocument 两个函数，因为对同一个线程，这些对象都是复用的，而非对每一篇新来的文档都创建一套，这样也提高了效率，也牵扯到数据的清理问题。一般在 startDocument 函数中，清理处理上篇文档遗留的数据，在 finishDocument 中，收集本次处理的结果数据，并返回，一直返回到 DocumentsWriter.updateDocument(Document, Analyzer, Term) 然后根据条件判断是否将数据刷新到硬盘上。

4.2.2、逐个处理文档的每一个域

由于一个线程可以连续处理多个文档，而在普通的应用中，几乎每篇文档的域都是大致相同的，为每篇文档的每个域都创建一个处理对象非常低效，因而考虑到复用域处理对象 DocFieldProcessorPerField，对于每一个域都有一个此对象。

那当来到一个新的域的时候，如何更快的找到此域的处理对象呢？Lucene 创建了一个 DocFieldProcessorPerField[] fieldHash 哈希表来方便更快查找域对应的处理对象。

当处理各个域的时候，按什么顺序呢？其实是按照域名的字典顺序。因而 Lucene 创建了 DocFieldProcessorPerField[] fields 的数组来方便按顺序处理域。

因而一个域的处理对象被放在了两个地方。

对于域的处理过程如下：

4.2.2.1、首先：对于每一个域，按照域名，在 fieldHash 中查找域处理对象 DocFieldProcessorPerField

代码如下：

```
final int hashPos = fieldName.hashCode() & hashMask;//计算哈希值
DocFieldProcessorPerField fp = fieldHash[hashPos];//找到哈希表中对应的位置
while(fp != null && !fp.fieldInfo.name.equals(fieldName)) fp = fp.next;//链式哈希表
```

如果能够找到，则更新 DocFieldProcessorPerField 中的域信息 fp.fieldInfo.update(field.isIndexed()…)

如果没有找到，则添加域到 DocFieldProcessorPerThread.fieldInfos 中，并创建新的 DocFieldProcessorPerField，且将其加入哈希表。代码如下：

```
fp = new DocFieldProcessorPerField(this, fi);
fp.next = fieldHash[hashPos];
fieldHash[hashPos] = fp;
```

如果是一个新的 field，则将其加入 fields 数组 fields[fieldCount++] = fp;

并且如果是存储域的话，用 StoredFieldsWriterPerThread 将其写到索引中：

```
if (field.isStored()) {
  fieldsWriter.addField(field, fp.fieldInfo);
}
```

处理存储域的过程如下：

```
StoredFieldsWriterPerThread.addField(Fieldable field, FieldInfo fieldInfo)
--> localFieldsWriter.writeField(fieldInfo, field);
```

FieldsWriter.writeField(FieldInfo fi, Fieldable field)代码如下：

```
请参照 fdt 文件的格式，则一目了然：
fieldsStream.writeVInt(fi.number);//文档号
byte bits = 0;
if (field.isTokenized())
```

```
    bits |= FieldsWriter.FIELD_IS_TOKENIZED;
if (field.isBinary())
    bits |= FieldsWriter.FIELD_IS_BINARY;
if (field.isCompressed())
    bits |= FieldsWriter.FIELD_IS_COMPRESSED;
fieldsStream.writeByte(bits); //域的属性位
if (field.isCompressed()) {//对于压缩域
    // compression is enabled for the current field
    final byte[] data;
    final int len;
    final int offset;
    // check if it is a binary field
    if (field.isBinary()) {
        data = CompressionTools.compress(field.getBinaryValue(), field.getBinaryOffset(),
field.getBinaryLength());
    } else {
       byte x[] = field.stringValue().getBytes("UTF-8");
       data = CompressionTools.compress(x, 0, x.length);
    }
    len = data.length;
    offset = 0;
    fieldsStream.writeVInt(len);//写长度
    fieldsStream.writeBytes(data, offset, len);//写二进制内容
} else {//对于非压缩域
    // compression is disabled for the current field
    if (field.isBinary()) {//如果是二进制域
       final byte[] data;
       final int len;
```

```
        final int offset;
        data = field.getBinaryValue();
        len = field.getBinaryLength();
        offset = field.getBinaryOffset();
        fieldsStream.writeVInt(len);//写长度
        fieldsStream.writeBytes(data, offset, len);//写二进制内容
      } else {
        fieldsStream.writeString(field.stringValue());//写字符内容
      }
    }
```

4.2.2.2、然后：对 fields 数组进行排序，是域按照名称排序。

quickSort(fields, 0, fieldCount-1);

4.2.2.3、最后：按照排序号的顺序，对域逐个处理，此处处理的仅仅是索引域，代码如下：

```
for(int i=0;i<fieldCount;i++)
      fields[i].consumer.processFields(fields[i].fields, fields[i].fieldCount);
```

域处理对象(DocFieldProcessorPerField)结构如下：

域索引链：

每个域也有自己的索引链，称为域索引链，每个域的索引链也有同**线程索引链**有相似的树形结构，由线程索引链中每个层次的每个层次的对象调用 addField 进行创建，负责对此域的处理。和基本索引链及线程索引链不同的是，域索引链仅仅负责处理索引域，而不负责存储域的处理。

DocFieldProcessorPerField 是域索引链的源头，对象结构如下：

DocFieldConsumerPerField consumer 类型为 DocInverterPerField，由 DocInverterPerThread.addField 创建

InvertedDocConsumerPerField consumer 类型为 TermsHashPerField，由 TermsHashPerThread.addField 创建

TermsHashConsumerPerField consumer 类型为 FreqProxTermsWriterPerField，由 FreqProxTermsWriterPerThread.addField 创建，负责 freq, prox 信息的处理

TermsHashPerField nextPerField

TermsHashConsumerPerField consumer 类型为 TermVectorsTermsWriterPerField，由 TermVectorsTermsWriterPerThread.addField 创建，负责 tvx, tvd, tvf 信息的处理

InvertedDocEndConsumerPerField endConsumer 类型为 NormsWriterPerField，由 NormsWriterPerThread.addField 创建，负责 nrm 信息的处理。

处理索引域的过程如下：

DocInverterPerField.processFields(Fieldable[], int) 过程如下：

- 判断是否要形成倒排表，代码如下：

```
boolean doInvert = consumer.start(fields, count);
--> TermsHashPerField.start(Fieldable[], int)
    --> for(int i=0;i<count;i++)
          if (fields[i].isIndexed())
              return true;
         return false;
```

读到这里，大家可能会发生困惑，既然 XXXPerField 是对于每一个域有一个处理对象的，那为什么参数传进来的是 Fieldable[]数组，并且还有域的数目 count 呢？

其实这不经常用到，但必须得提一下，由上面的 fieldHash 的实现我们可以看到，是根据域名进行哈希的，所以准确的讲，XXXPerField 并非对于每一个域有一个处理对象，而是对每一组相同名字的域有相同的处理对象。

对于同一篇文档，相同名称的域可以添加多个，代码如下：

```
doc.add(new Field("contents", "the content of the file.", Field.Store.NO, Field.Index.NOT_ANALYZED));
doc.add(new Field("contents", new FileReader(f)));
```

则传进来的名为"contents"的域如下：

```
fields    Fieldable[2]   (id=52)
   [0]    Field   (id=56)
      binaryLength    0
      binaryOffset    0
      boost    1.0
      fieldsData    "the content of the file."
      isBinary    false
      isCompressed    false
      isIndexed    true
      isStored    false
      isTokenized    false
      lazy    false
      name    "contents"
      omitNorms    false
      omitTermFreqAndPositions    false
      storeOffsetWithTermVector    false
      storePositionWithTermVector    false
      storeTermVector    false
      tokenStream    null
   [1]    Field   (id=58)
      binaryLength    0
      binaryOffset    0
      boost    1.0
      fieldsData    FileReader   (id=131)
      isBinary    false
      isCompressed    false
      isIndexed    true
      isStored    false
```

```
    isTokenized    true
    lazy    false
    name    "contents"
    omitNorms    false
    omitTermFreqAndPositions    false
    storeOffsetWithTermVector    false
    storePositionWithTermVector    false
    storeTermVector    false
    tokenStream    null
```

- 对传进来的同名域逐一处理，代码如下

```
for(int i=0;i<count;i++){
   final Fieldable field = fields[i];
   if (field.isIndexed() && doInvert) {
     //仅仅对索引域进行处理
     if (!field.isTokenized()) {
       //如果此域不分词，见(1)对不分词的域的处理
     } else {
       //如果此域分词，见(2)对分词的域的处理
     }
   }
}
```

(1) 对不分词的域的处理

(1-1) 得到域的内容，并构建单个 Token 形成的 SingleTokenAttributeSource。因为不进行分词，因而整个域的内容算做一个 Token.

```
String stringValue = field.stringValue(); //stringValue    "200910240957"
final int valueLength = stringValue.length();
perThread.singleToken.reinit(stringValue, 0, valueLength);
```

对于此域唯一的一个 Token 有以下的属性：

- Term：文字信息。在处理过程中，此值将保存在 TermAttribute 的实现类实例化的对象 TermAttributeImp 里面。
- Offset：偏移量信息，是按字或字母的起始偏移量和终止偏移量，表明此 Token 在文章中的位置，多用于加亮。在处理过程中，此值将保存在 OffsetAttribute 的实现类实例化的对象 OffsetAttributeImp 里面。

在 SingleTokenAttributeSource 里面，有一个 HashMap 来保存可能用于保存属性的类名(Key，准确的讲是接口)以及保存属性信息的对象(Value)：

```
singleToken    DocInverterPerThread$SingleTokenAttributeSource  (id=150)
   attributeImpls    LinkedHashMap<K,V>  (id=945)
   attributes    LinkedHashMap<K,V>  (id=946)
     size    2
     table    HashMap$Entry<K,V>[16]  (id=988)
       [0]    LinkedHashMap$Entry<K,V>  (id=991)
         key    Class<T> (org.apache.lucene.analysis.tokenattributes.TermAttribute) (id=755)
         value    TermAttributeImpl  (id=949)
           termBuffer    char[19]  (id=954)    //[2, 0, 0, 9, 1, 0, 2, 4, 0, 9, 5, 7]
             termLength    12
       [7]    LinkedHashMap$Entry<K,V>  (id=993)
           key       Class<T>      (org.apache.lucene.analysis.tokenattributes.OffsetAttribute)
(id=274)
         value    OffsetAttributeImpl  (id=948)
           endOffset    12
             startOffset    0
   factory    AttributeSource$AttributeFactory$DefaultAttributeFactory  (id=947)
   offsetAttribute    OffsetAttributeImpl  (id=948)
   termAttribute    TermAttributeImpl  (id=949)
```

(1-2) 得到 Token 的各种属性信息，为索引做准备。

consumer.start(field)做的主要事情就是根据各种属性的类型来构造保存属性的对象

(HashMap 中有则取出，无则构造)，为索引做准备。

```
consumer(TermsHashPerField).start(…)
--> termAtt = fieldState.attributeSource.addAttribute(TermAttribute.class); 得到的就是上述
HashMap 中的 TermAttributeImpl
--> consumer(FreqProxTermsWriterPerField).start(f);
    --> if (fieldState.attributeSource.hasAttribute(PayloadAttribute.class)) {
          payloadAttribute = fieldState.attributeSource.getAttribute(PayloadAttribute.class);
          存储 payload 信息则得到 payload 的属}
--> nextPerField(TermsHashPerField).start(f);
    --> termAtt = fieldState.attributeSource.addAttribute(TermAttribute.class);得到的还是上述
HashMap 中的 TermAttributeImpl
    --> consumer(TermVectorsTermsWriterPerField).start(f);
        --> if (doVectorOffsets) {
              offsetAttribute = fieldState.attributeSource.addAttribute(OffsetAttribute.class);
              如果存储词向量则得到的是上述 HashMap 中的 OffsetAttributeImp }
```

(1-3) 将 Token 加入倒排表

consumer(TermsHashPerField).add();

加入倒排表的过程，无论对于分词的域和不分词的域，过程是一样的，因而放到对分词的域的解析中一起说明。

(2) 对分词的域的处理

(2-1) 构建域的 TokenStream

```
final TokenStream streamValue = field.tokenStreamValue();
//用户可以在添加域的时候，应用构造函数 public Field(String name, TokenStream tokenStream)
直接传进一个 TokenStream 过来，这样就不用另外构建一个 TokenStream 了。
if (streamValue != null)
  stream = streamValue;
else {
  ……
```

```
  stream = docState.analyzer.reusableTokenStream(fieldInfo.name, reader);
}
```

此时 TokenStream 的各项属性值还都是空的，等待一个一个被分词后得到，此时的 TokenStream 对象如下：

```
stream    StopFilter  (id=112)
   attributeImpls    LinkedHashMap<K,V>  (id=121)
   attributes    LinkedHashMap<K,V>  (id=122)
     size    4
     table    HashMap$Entry<K,V>[16]  (id=146)
       [2]    LinkedHashMap$Entry<K,V>  (id=148)
         key    Class<T> (org.apache.lucene.analysis.tokenattributes.TypeAttribute) (id=154)
         value    TypeAttributeImpl  (id=157)
           type    "word"
       [8]    LinkedHashMap$Entry<K,V>  (id=150)
         after    LinkedHashMap$Entry<K,V>  (id=156)
             key      Class<T>    (org.apache.lucene.analysis.tokenattributes.OffsetAttribute)
(id=163)
           value    OffsetAttributeImpl  (id=164)
             endOffset    0
                 startOffset    0
         key    Class<T> (org.apache.lucene.analysis.tokenattributes.TermAttribute) (id=142)
         value    TermAttributeImpl  (id=133)
           termBuffer    char[17]  (id=173)
               termLength    0
       [10]    LinkedHashMap$Entry<K,V>  (id=151)
                                          key                                    Class<T>
(org.apache.lucene.analysis.tokenattributes.PositionIncrementAttribute) (id=136)
         value    PositionIncrementAttributeImpl  (id=129)
```

```
        positionIncrement    1
  currentState    AttributeSource$State  (id=123)
  enablePositionIncrements    true
  factory    AttributeSource$AttributeFactory$DefaultAttributeFactory  (id=125)
  input    LowerCaseFilter  (id=127)
    input    StandardFilter  (id=213)
      input    StandardTokenizer  (id=218)
        input    FileReader  (id=93)    //从文件中读出来的文本，将经过分词器分词，并一
层层的 Filter 的处理，得到一个个 Token
  stopWords    CharArraySet$UnmodifiableCharArraySet  (id=131)
  termAtt    TermAttributeImpl  (id=133)
```

(2-2) 得到第一个 Token，并初始化此 Token 的各项属性信息，并为索引做准备(start)。

boolean hasMoreTokens = stream.incrementToken();//得到第一个 Token

OffsetAttribute offsetAttribute = fieldState.attributeSource.addAttribute(OffsetAttribute.class);//得到偏移量属性

```
offsetAttribute    OffsetAttributeImpl  (id=164)
  endOffset    8
  startOffset    0
```

PositionIncrementAttribute posIncrAttribute = fieldState.attributeSource.addAttribute(PositionIncrementAttribute.class);//得到位置属性

```
posIncrAttribute    PositionIncrementAttributeImpl  (id=129)
  positionIncrement    1
```

consumer.start(field);// 其中得到了 TermAttribute 属性，如果存储 payload 则得到 PayloadAttribute 属性，如果存储词向量则得到 OffsetAttribute 属性。

(2-3) 进行循环，不断的取下一个 Token，并添加到倒排表

```
for(;;) {
```

```
    if (!hasMoreTokens) break;

    ……
    consumer.add();

    ……
    hasMoreTokens = stream.incrementToken();
}
```

(2-4) 添加 Token 到倒排表的过程 consumer(TermsHashPerField).add()

TermsHashPerField 对象主要包括以下部分：

- CharBlockPool charPool; 用于存储 Token 的文本信息，如果不足时，从 DocumentsWriter 中的 freeCharBlocks 分配
- ByteBlockPool bytePool;用于存储 freq, prox 信息，如果不足时，从 DocumentsWriter 中的 freeByteBlocks 分配
- IntBlockPool intPool; 用于存储分别指向每个 Token 在 bytePool 中 freq 和 prox 信息的偏移量。如果不足时，从 DocumentsWriter 的 freeIntBlocks 分配
- TermsHashConsumerPerField consumer 类型为 FreqProxTermsWriterPerField，用于写 freq, prox 信息到缓存中。
- RawPostingList[] postingsHash = new RawPostingList[postingsHashSize];存储倒排表，每一个 Term 都有一个 RawPostingList (PostingList)，其中包含了 int textStart，也即文本在 charPool 中的偏移量，int byteStart，即此 Term 的 freq 和 prox 信息在 bytePool 中的起始偏移量，int intStart，即此 term 的在 intPool 中的起始偏移量。

形成倒排表的过程如下：

```
//得到 token 的文本及文本长度

final char[] tokenText = termAtt.termBuffer();//[s, t, u, d, e, n, t, s]

final int tokenTextLen = termAtt.termLength();//tokenTextLen 8
```

```
//按照 token 的文本计算哈希值，以便在 postingsHash 中找到此 token 对应的倒排表

int downto = tokenTextLen;
int code = 0;
while (downto > 0) {
  char ch = tokenText[—downto];
  code = (code*31) + ch;
}

int hashPos = code & postingsHashMask;

//在倒排表哈希表中查找此 Token，如果找到相应的位置，但是不是此 Token，说明此位置存在哈希冲突，采取重新哈希 rehash 的方法。

p = postingsHash[hashPos];

if (p != null && !postingEquals(tokenText, tokenTextLen)) {
  final int inc = ((code>>8)+code)|1;
  do {
    code += inc;
    hashPos = code & postingsHashMask;
    p = postingsHash[hashPos];
  } while (p != null && !postingEquals(tokenText, tokenTextLen));
}

//如果此 Token 之前从未出现过

if (p == null) {

    if (textLen1 + charPool.charUpto > DocumentsWriter.CHAR_BLOCK_SIZE) {
```

```
      //当 charPool 不足的时候，在 freeCharBlocks 中分配新的 buffer

      charPool.nextBuffer();

    }

    //从空闲的倒排表中分配新的倒排表

    p = perThread.freePostings[--perThread.freePostingsCount];

    //将文本复制到 charPool 中

    final char[] text = charPool.buffer;
    final int textUpto = charPool.charUpto;
    p.textStart = textUpto + charPool.charOffset;
    charPool.charUpto += textLen1;
    System.arraycopy(tokenText, 0, text, textUpto, tokenTextLen);
    text[textUpto+tokenTextLen] = 0xffff;

    //将倒排表放入哈希表中

    postingsHash[hashPos] = p;
    numPostings++;

    if (numPostingInt + intPool.intUpto > DocumentsWriter.INT_BLOCK_SIZE)
intPool.nextBuffer();

    //当 intPool 不足的时候，在 freeIntBlocks 中分配新的 buffer。

    if (DocumentsWriter.BYTE_BLOCK_SIZE - bytePool.byteUpto <
numPostingInt*ByteBlockPool.FIRST_LEVEL_SIZE)
```

```
        bytePool.nextBuffer();

    //当 bytePool 不足的时候，在 freeByteBlocks 中分配新的 buffer。

    //此处 streamCount 为 2，表明在 intPool 中，每两项表示一个词，一个是指向 bytePool 中 freq
信息偏移量的，一个是指向 bytePool 中 prox 信息偏移量的。

    intUptos = intPool.buffer;
    intUptoStart = intPool.intUpto;
    intPool.intUpto += streamCount;

    p.intStart = intUptoStart + intPool.intOffset;

    //在 bytePool 中分配两个空间，一个放 freq 信息，一个放 prox 信息的。
    for(int i=0;i<streamCount;i++) {

        final int upto = bytePool.newSlice(ByteBlockPool.FIRST_LEVEL_SIZE);
        intUptos[intUptoStart+i] = upto + bytePool.byteOffset;
    }
    p.byteStart = intUptos[intUptoStart];

    //当 Term 原来没有出现过的时候，调用 newTerm

    consumer(FreqProxTermsWriterPerField).newTerm(p);

}

//如果此 Token 之前曾经出现过，则调用 addTerm。

else {

    intUptos = intPool.buffers[p.intStart >> DocumentsWriter.INT_BLOCK_SHIFT];
    intUptoStart = p.intStart & DocumentsWriter.INT_BLOCK_MASK;
```

```
    consumer(FreqProxTermsWriterPerField).addTerm(p);

}
```

(2-5) 添加新 Term 的过程，consumer(FreqProxTermsWriterPerField).newTerm

```
final void newTerm(RawPostingList p0) {
  FreqProxTermsWriter.PostingList p = (FreqProxTermsWriter.PostingList) p0;
  p.lastDocID = docState.docID; //当一个新的 term 出现的时候，包含此 Term 的就只有本篇文档，记录其 ID
  p.lastDocCode = docState.docID << 1; //docCode 是文档 ID 左移一位，为什么左移，请参照索引文件格式(1)中的或然跟随规则。
  p.docFreq = 1; //docFreq 这里用词可能容易引起误会，docFreq 这里指的是此文档所包含的此 Term 的次数，并非包含此 Term 的文档的个数。
  writeProx(p, fieldState.position); //写入 prox 信息到 bytePool 中，此时 freq 信息还不能写入，因为当前的文档还没有处理完，尚不知道此文档包含此 Term 的总数。
}

writeProx(FreqProxTermsWriter.PostingList p, int proxCode) {

  termsHashPerField.writeVInt(1, proxCode<<1);//第一个参数所谓 1，也就是写入此文档在 intPool 中的第 1 项——prox 信息。为什么左移一位呢？是因为后面可能跟着 payload 信息，参照索引文件格式(1)中或然跟随规则。
  p.lastPosition = fieldState.position;//总是要记录 lastDocID, lastPostion，是因为要计算差值，参照索引文件格式(1)中的差值规则。

}
```

(2-6) 添加已有 Term 的过程

```
final void addTerm(RawPostingList p0) {

  FreqProxTermsWriter.PostingList p = (FreqProxTermsWriter.PostingList) p0;
```

```
  if (docState.docID != p.lastDocID) {

    //当文档 ID 变了的时候，说明上一篇文档已经处理完毕，可以写入 freq 信息了。

    //第一个参数所谓 0，也就是写入上一篇文档在 intPool 中的第 0 项——freq 信息。至于信息为何这样写，参照索引文件格式(1)中的或然跟随规则，及 tis 文件格式。

    if (1 == p.docFreq)
      termsHashPerField.writeVInt(0, p.lastDocCode|1);
    else {
      termsHashPerField.writeVInt(0, p.lastDocCode);
      termsHashPerField.writeVInt(0, p.docFreq);
    }
    p.docFreq = 1;//对于新的文档，freq 还是为 1.
    p.lastDocCode = (docState.docID - p.lastDocID) << 1;//文档号存储差值
    p.lastDocID = docState.docID;
    writeProx(p, fieldState.position);
   } else {

    //当文档 ID 不变的时候，说明此文档中这个词又出现了一次，从而 freq 加一，写入再次出现的位置信息，用差值。
    p.docFreq++;
    writeProx(p, fieldState.position-p.lastPosition);
  }
}
```

(2-7) 结束处理当前域

```
consumer(TermsHashPerField).finish();

--> FreqProxTermsWriterPerField.finish()
```

```
--> TermVectorsTermsWriterPerField.finish()

endConsumer(NormsWriterPerField).finish();

--> norms[upto] = Similarity.encodeNorm(norm);//计算标准化因子的值。

--> docIDs[upto] = docState.docID;
```

4.2.3、结束处理当前文档

final DocumentsWriter.DocWriter one =

fieldsWriter(StoredFieldsWriterPerThread).finishDocument();

存储域返回结果：一个写成了二进制的存储域缓存。

```
one    StoredFieldsWriter$PerDoc  (id=322)
   docID    0
   fdt    RAMOutputStream  (id=325)
      bufferLength    1024
      bufferPosition    40
      bufferStart    0
      copyBuffer    null
      currentBuffer    byte[1024]  (id=332)
      currentBufferIndex    0
      file    RAMFile  (id=333)
      utf8Result    UnicodeUtil$UTF8Result  (id=335)
   next    null
   numStoredFields    2
   this$0    StoredFieldsWriter  (id=327)
```

final DocumentsWriter.DocWriter two = consumer(DocInverterPerThread).finishDocument();

--> NormsWriterPerThread.finishDocument()

--> TermsHashPerThread.finishDocument()

索引域的返回结果为 null

4.3、用 DocumentsWriter.finishDocument 结束本次文档添加

代码：

```
DocumentsWriter.updateDocument(Document, Analyzer, Term)
--> DocumentsWriter.finishDocument(DocumentsWriterThreadState,
DocumentsWriter$DocWriter)
    --> doPause = waitQueue.add(docWriter);//有关 waitQueue，在 DocumentsWriter 的缓存管理中已作解释
        --> DocumentsWriter$WaitQueue.writeDocument(DocumentsWriter$DocWriter)
            --> StoredFieldsWriter$PerDoc.finish()
                --> fieldsWriter.flushDocument(perDoc.numStoredFields, perDoc.fdt);将存储域信息真正写入文件。
```

5、DocumentsWriter 对 CharBlockPool，ByteBlockPool，IntBlockPool 的缓存管理

- 在索引的过程中，DocumentsWriter 将词信息(term)存储在 CharBlockPool 中，将文档号(doc ID)，词频(freq)和位置(prox)信息存储在 ByteBlockPool 中。
- 在 ByteBlockPool 中，缓存是分块(slice)分配的，块(slice)是分层次的，层次越高，此层的块越大，每一层的块大小事相同的。
  - nextLevelArray 表示的是当前层的下一层是第几层，可见第 9 层的下一层还是第 9 层，也就是说最高有 9 层。
  - levelSizeArray 表示每一层的块大小，第一层是 5 个 byte，第二层是 14 个 byte 以此类推。

```
ByteBlockPool 类中有以下静态变量：

final static int[] nextLevelArray = {1, 2, 3, 4, 5, 6, 7, 8, 9, 9};
```

```
final static int[] levelSizeArray = {5, 14, 20, 30, 40, 40, 80, 80, 120, 200};
```

- 在 ByteBlockPool 中分配一个块的代码如下：

```
//此函数仅仅在 upto 已经是当前块的结尾的时候方才调用来分配新块。
public int allocSlice(final byte[] slice, final int upto) {
  //可根据块的结束符来得到块所在的层次。从而我们可以推断，每个层次的块都有不同的结束符，第 1 层为 16，第 2 层位 17，第 3 层 18，依次类推。
  final int level = slice[upto] & 15;
  //从数组总得到下一个层次及下一层块的大小。
  final int newLevel = nextLevelArray[level];
  final int newSize = levelSizeArray[newLevel];
  // 如果当前缓存总量不够大，则从 DocumentsWriter 的 freeByteBlocks 中分配。
  if (byteUpto > DocumentsWriter.BYTE_BLOCK_SIZE-newSize)
    nextBuffer();
  final int newUpto = byteUpto;
  final int offset = newUpto + byteOffset;
  byteUpto += newSize;
  //当分配了新的块的时候，需要有一个指针从本块指向下一个块，使得读取此信息的时候，能够在此块读取结束后，到下一个块继续读取。
  //这个指针需要 4 个 byte，在本块中，除了结束符所占用的一个 byte 之外，之前的三个 byte 的数据都应该移到新的块中，从而四个 byte 连起来形成一个指针。
  buffer[newUpto] = slice[upto-3];
  buffer[newUpto+1] = slice[upto-2];
  buffer[newUpto+2] = slice[upto-1];
  // 将偏移量(也即指针)写入到连同结束符在内的四个 byte
  slice[upto-3] = (byte) (offset >>> 24);
  slice[upto-2] = (byte) (offset >>> 16);
  slice[upto-1] = (byte) (offset >>> 8);
  slice[upto] = (byte) offset;
```

```
  // 在新的块的结尾写入新的结束符，结束符和层次的关系就是(endbyte = 16 | level)
  buffer[byteUpto-1] = (byte) (16|newLevel);
  return newUpto+3;
}
```

- 在 ByteBlockPool 中，文档号和词频(freq)信息是应用或然跟随原则写到一个块中去的，而位置信息(prox)是写入到另一个块中去的，对于同一个词，这两块的偏移量保存在 IntBlockPool 中。因而在 IntBlockPool 中，每一个词都有两个 int，第 0 个表示 docid + freq 在 ByteBlockPool 中的偏移量，第 1 个表示 prox 在 ByteBlockPool 中的偏移量。
- 在写入 docid + freq 信息的时候，调用 termsHashPerField.writeVInt(0, p.lastDocCode)，第一个参数表示向此词的第 0 个偏移量写入；在写入 prox 信息的时候，调用 termsHashPerField.writeVInt(1, (proxCode<<1)|1)，第一个参数表示向此词的第 1 个偏移量写入。
- CharBlockPool 是按照出现的先后顺序保存词(term)
- 在 TermsHashPerField 中，有一个成员变量 RawPostingList[] postingsHash，为每一个 term 分配了一个 RawPostingList，将上述三个缓存关联起来。

```
abstract class RawPostingList {
    final static int BYTES_SIZE = DocumentsWriter.OBJECT_HEADER_BYTES +
3*DocumentsWriter.INT_NUM_BYTE;
  int textStart; //此词在 CharBlockPool 中的偏移量，由此可以知道是哪个词。
  int intStart; //此词在 IntBlockPool 中的偏移量，在指向的位置有两个 int，一个是 docid + freq
信息的偏移量，一个是 prox 信息的偏移量。
  int byteStart; //此词在 ByteBlockPool 中的起始偏移量
}
static final class PostingList extends RawPostingList {
  int docFreq;                    // 此词在此文档中出现的次数
  int lastDocID;                  // 上次处理完的包含此词的文档号。
  int lastDocCode;                  // 文档号和词频按照或然跟随原则形成的编码
  int lastPosition;                // 上次处理完的此词的位置
```

}

这里需要说明的是，在 IntBlockPool 中保存了两个在 ByteBlockPool 中的偏移量，而在 RawPostingList 的 byteStart 又保存了在 ByteBlockPool 中的偏移量，这两者有什么区别呢？

在 IntBlockPool 中保存的分别指向 docid+freq 及 prox 信息在 ByteBlockPool 中的偏移量是主要用来写入信息的，它记录的偏移量是下一个要写入的 docid+freq 或者 prox 在 ByteBlockPool 中的位置，随着信息的不断写入，IntBlockPool 中的两个偏移量是不断改变的，始终指向下一个可以写入的位置。

RawPostingList 中 byteStart 主要是用来读取 docid 及 prox 信息的，当索引过程基本结束，所有的信息都写入在缓存中了，那么如何找到此词对应的文档号偏移量及位置信息，然后写到索引文件中去呢？自然是通过 RawPostingList 找到 byteStart，然后根据 byteStart 在 ByteBlockPool 中找到 docid+freq 及 prox 信息的起始位置，从起始位置开始的两个大小为 5 的块，第一个就是 docid+freq 信息的源头，第二个就是 prox 信息的源头，如果源头的块中包含了所有的信息，读出来就可以了，如果源头的块中有指针，则沿着指针寻找到下一个块，从而可以找到所有的信息。

- 下面举一个实例来表明如果进行缓存管理的：

此例子中，准备添加三个文件：

file01: common common common common common term

file02: common common common common common term term

file03: term term term common common common common common

file04: term

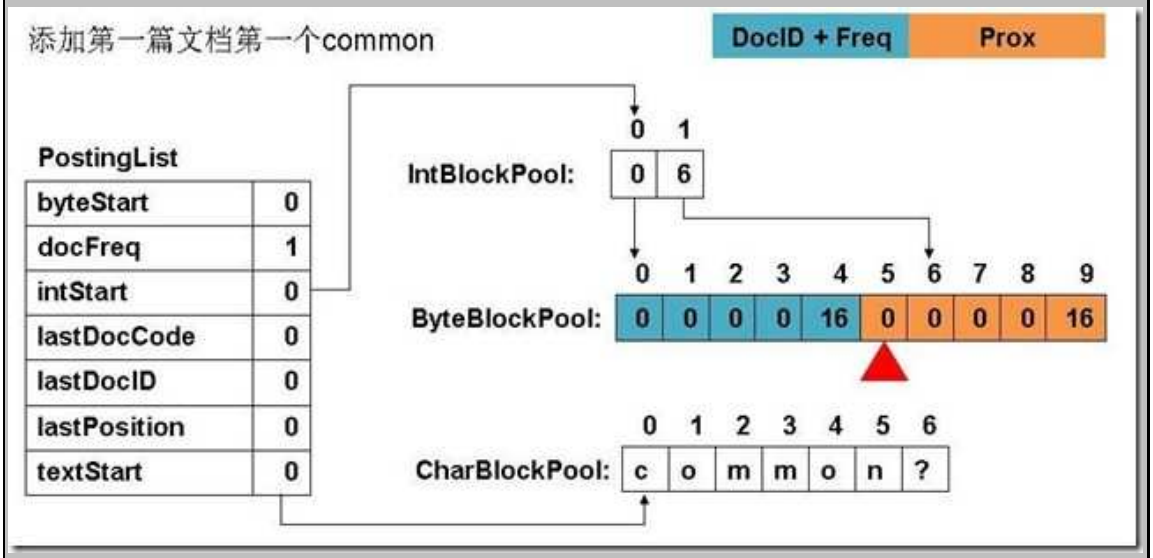
(1) 添加第一篇文档第一个 common

在 CharBlockPool 中分配 6 个 char 来存放"common"字符串

在 ByteBlockPool 中分配两个块，每个块大小为 5，以 16 结束，第一个块用来存放 docid+freq 信息，第二个块用来存放 prox 信息。此时 docid+freq 信息没有写入，docid+freq 信息总是在下一篇文档的处理过程出现了"common"的时候方才写入，因为当一篇文档没有处理完毕的时候，freq 也即词频是无法知道的。而 prox 信息存放 0，是因为第一个 common 的位置为 0，但是此 0 应该左移一位，最后一位置 0 表示没有 payload 存储，因而 0<<1 + 0 = 0。

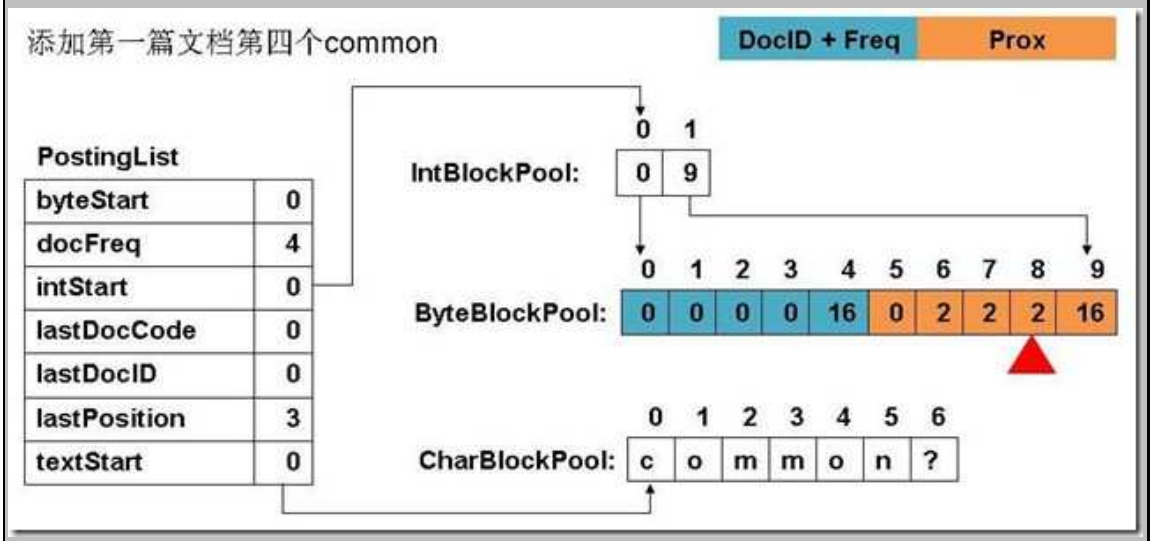
在 IntBlockPool 中分配两个 int，一个指向第 0 个位置，是因为当前没有 docid+freq 信息写入，

第二个指向第 6 个位置，是因为第 5 个位置写入了 prox 信息。所以 IntBlockPool 中存放的是下一个要写入的位置。



(2) 添加第四个 common

在 ByteBlockPool 中，prox 信息已经存放了 4 个，第一个 0 是代表第一个位置为 0，后面不跟随 payload。第二个 2 表示，位置增量(差值原则)为 1，后面不跟随 payload(或然跟随原则)，1<<1 + 0 =2。第三个第四个同第二个。



(3) 添加第五个 common

ByteBlockPool 中，存放 prox 信息的块已经全部填满，必须重新分配新的块。

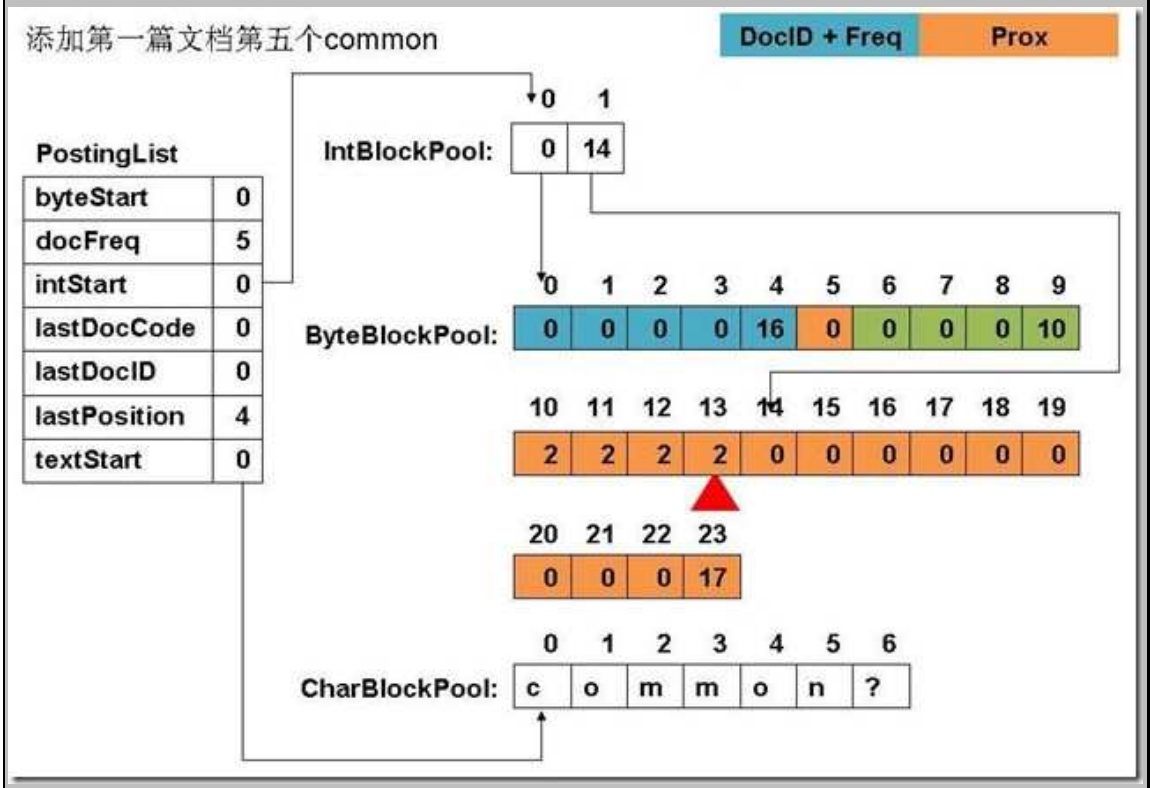
新的块层次为 2，大小为 14，在缓存的最后追加分配。

原块中连同结束位在内的四个 byte 作为指针(绿色部分)，指向新的块的其实地址，在此为 10.

被指针占用的结束位之前的三位移到新的块中，也即 6, 7, 8 移到 10, 11, 12 处，13 处是第五个 common 的 prox 信息。

指针的值并不都是四个 byte 的最后一位，当缓存很大的时候，指针的值也会很大。比如指针出现[0, 0, 0, -56]，最后一位为负，并不表示指向的负位置，而是最后一个 byte 的第一位为 1，显示为有符号数为负，-56 的二进制是 11001000，和前三个 byte 拼称 int，大小为 200 也即指向第 200 个位置。比如指针出现[0, 0, 1, 2]，其转换为二进制的 int 为 100000010，大小为 258，也即指向第 258 个位置。比如指针出现

[0, 0, 1, -98]，转换为二进制的 int 为 110011110，大小为 414，也即指向第 414 个位置。

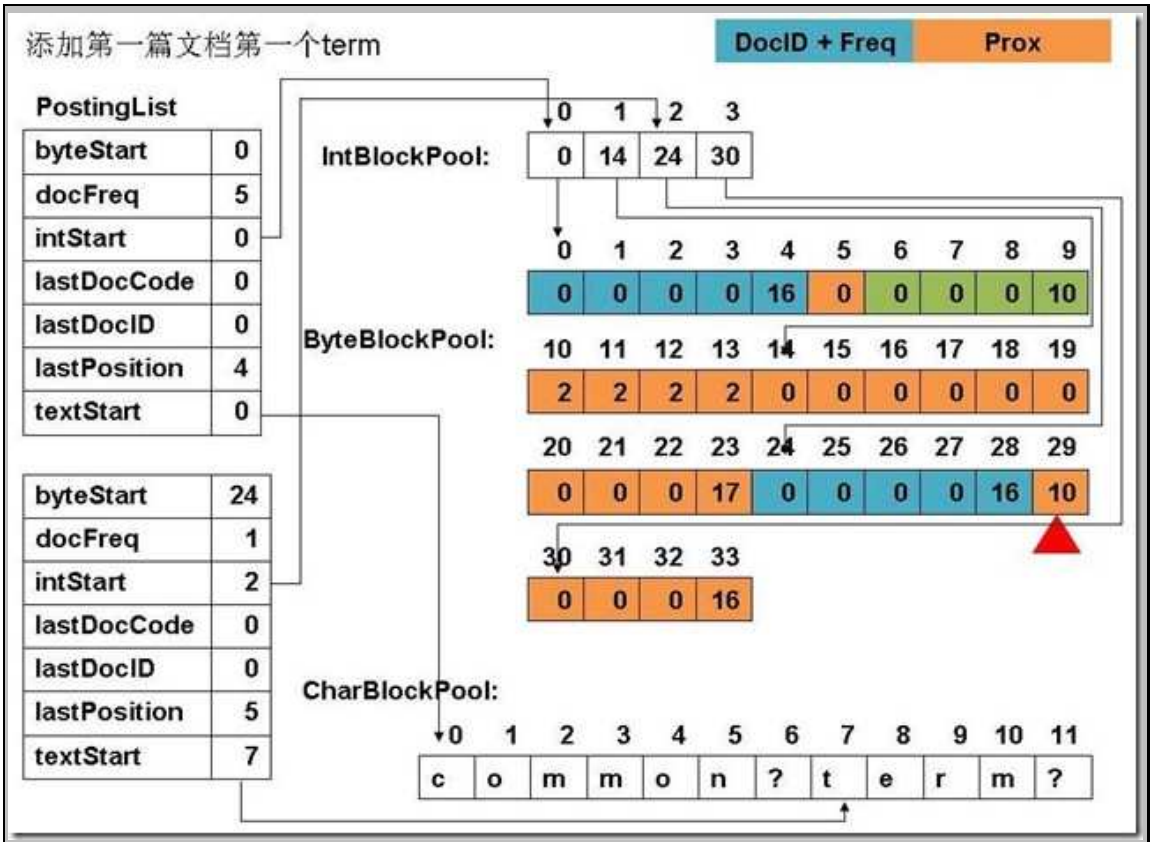


(4) 添加第一篇文档，第一个 term

CharBlockPool 中分配了 5 个 char 来存放"term"

ByteBlockPool 中分配了两个块来分别存放 docid+freq 信息和 prox 信息。第一个块没有信息写入，第二个块写入了"term"的位置信息，即出现在第 5 个位置，并且后面没有 payload，5<<1 + 0 = 10。

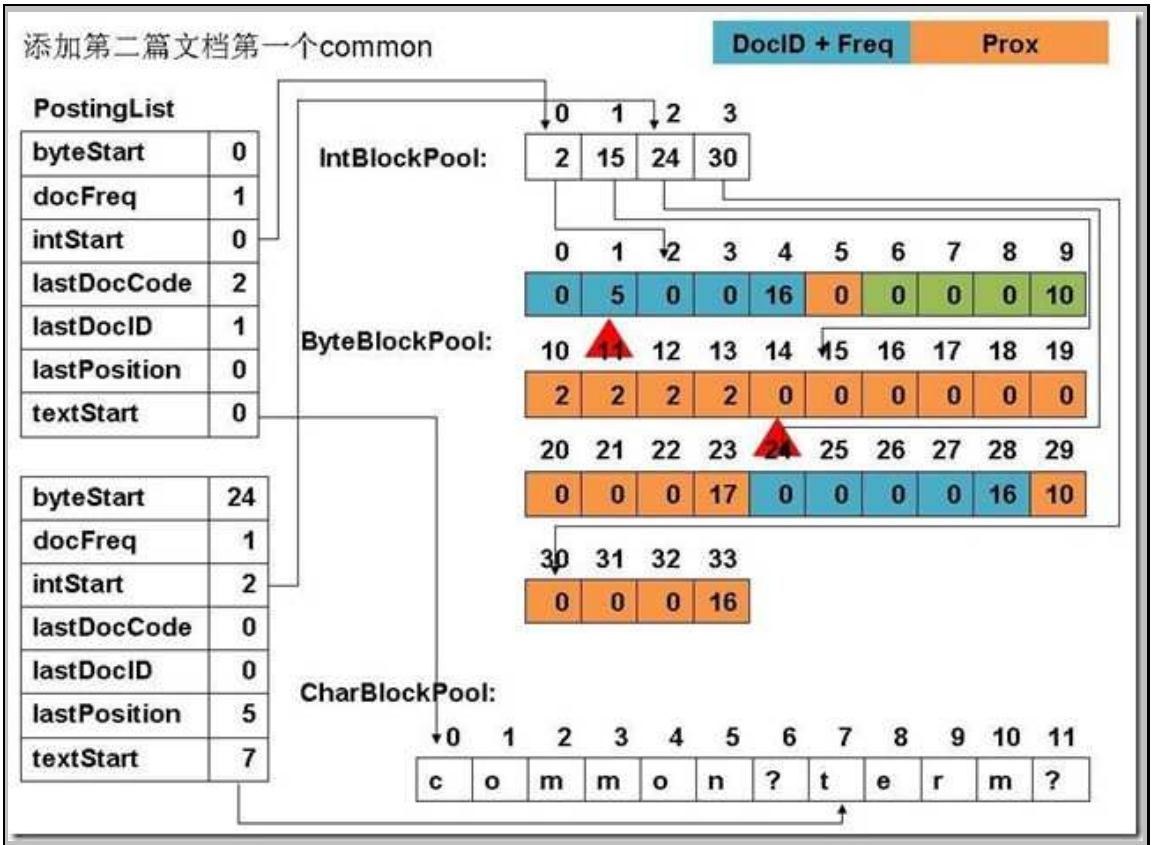
IntBlockPool 中分配了两个 int 来指向"term"的两个块中下一个写入的位置。



(5) 添加第二篇文档第一个 common

第一篇文档的 common 的 docid+freq 信息写入。在第一篇文档中，"common"出现了 5 次，文档号为 0，按照或然跟随原则，存放的信息为[docid<<1 + 0, 5] = [0, 5]，docid 左移一位，最后一位为 0 表示 freq 大于 1。

第二篇文档第一个 common 的位置信息也写入了，位置为 0，0<<1 + 0 = 0。

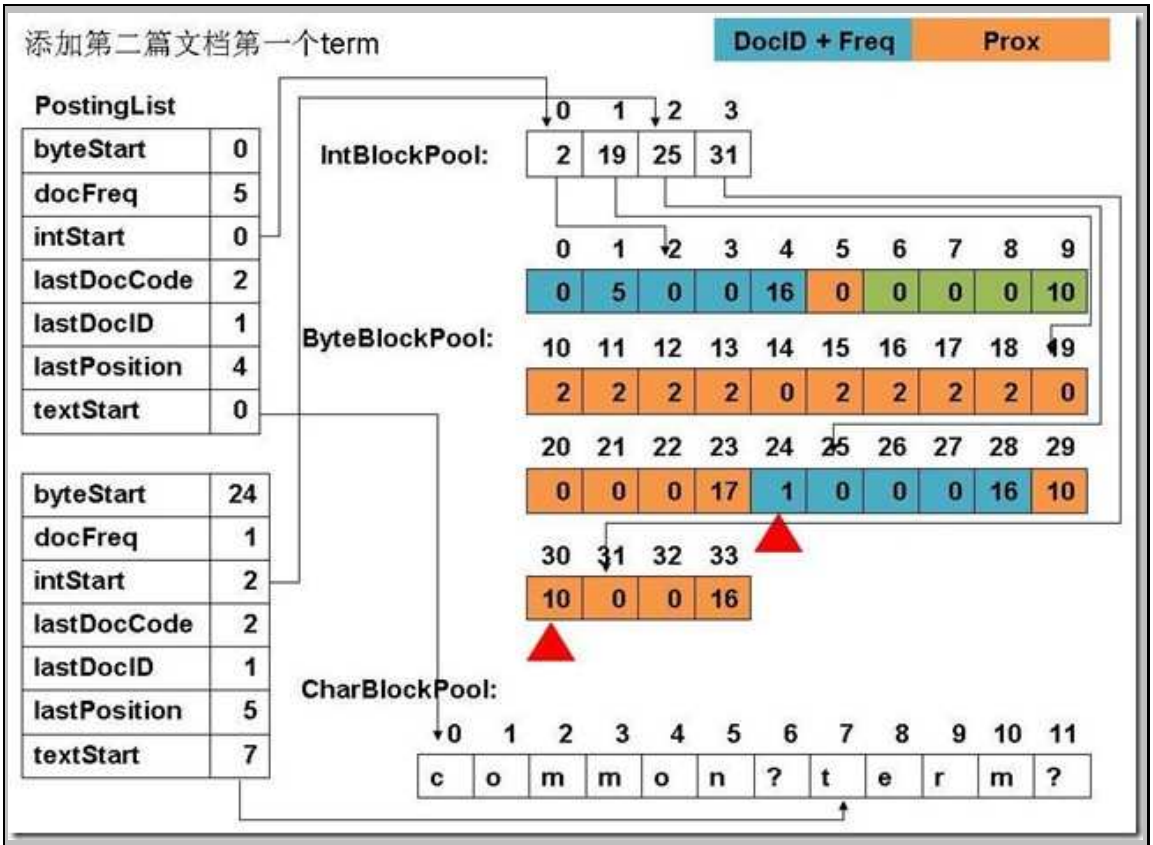


(6) 添加第二篇文档第一个 term

第一篇文档中的 term 的 docid+freq 信息写入，在第一篇文档中，"term"出现了 1 次，文档号为 0，所以存储信息为[docid<<1 + 1] = [1]，文档号左移一位，最后一位为 1 表示 freq 为 1。

第二篇文档第一个 term 的 prox 信息也写入了，在第 5 个位置，5<<1 + 0 = 10。

第二篇文档中的 5 个 common 的 prox 信息也写入了，分别为从 14 到 18 的[0, 2, 2, 2, 2]

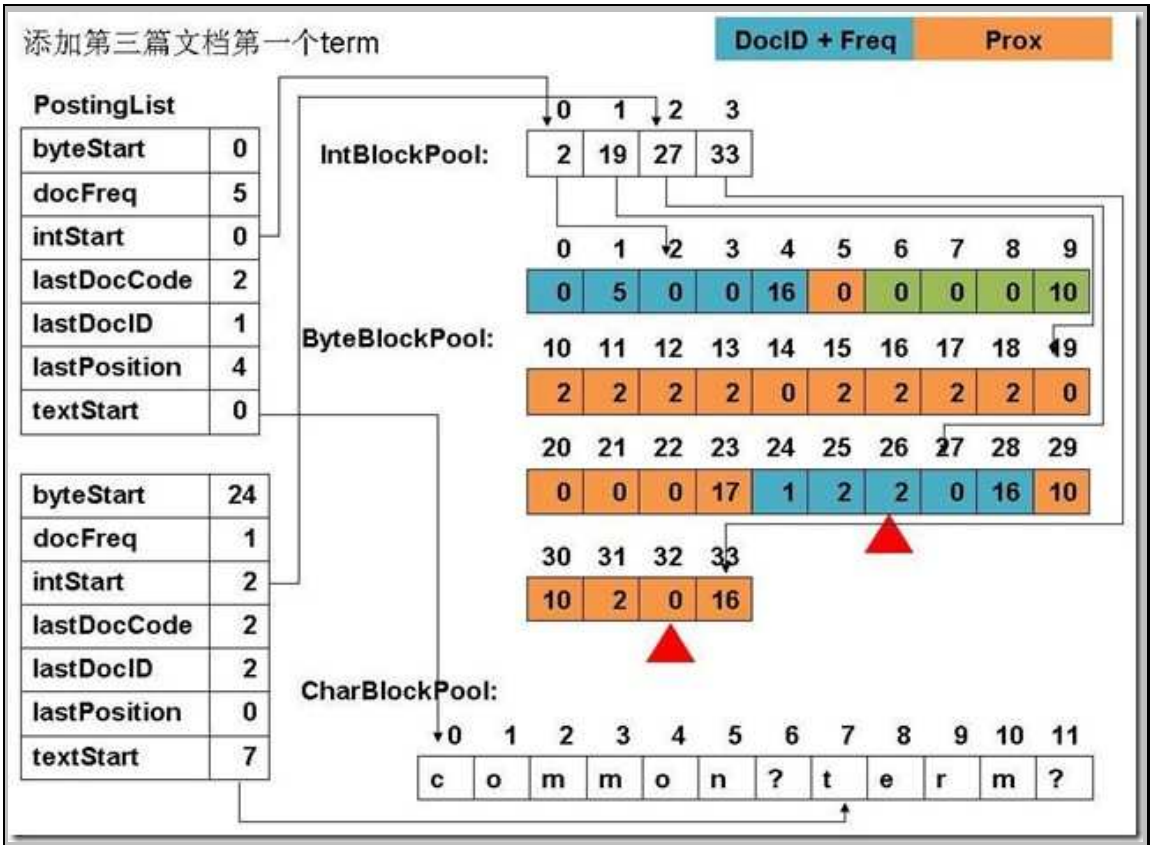


(7) 添加第三篇文档的第一个 term

第二篇文档的 term 的 docid+freq 信息写入，在第二篇文档中，文档号为 1，"term"出现了 2 次，所以存储为[docid<<1 + 0, freq] = [2, 2]，存储在 25, 26 两个位置。

第二篇文档中两个 term 的位置信息也写入了，为 30, 31 的[10, 2]，也即出现在第 5 个，第 6 个位置，后面不跟随 payload。

第三篇文档的第一个 term 的位置信息也写入了，在第 0 个位置，不跟 payload，为 32 保存的[0]



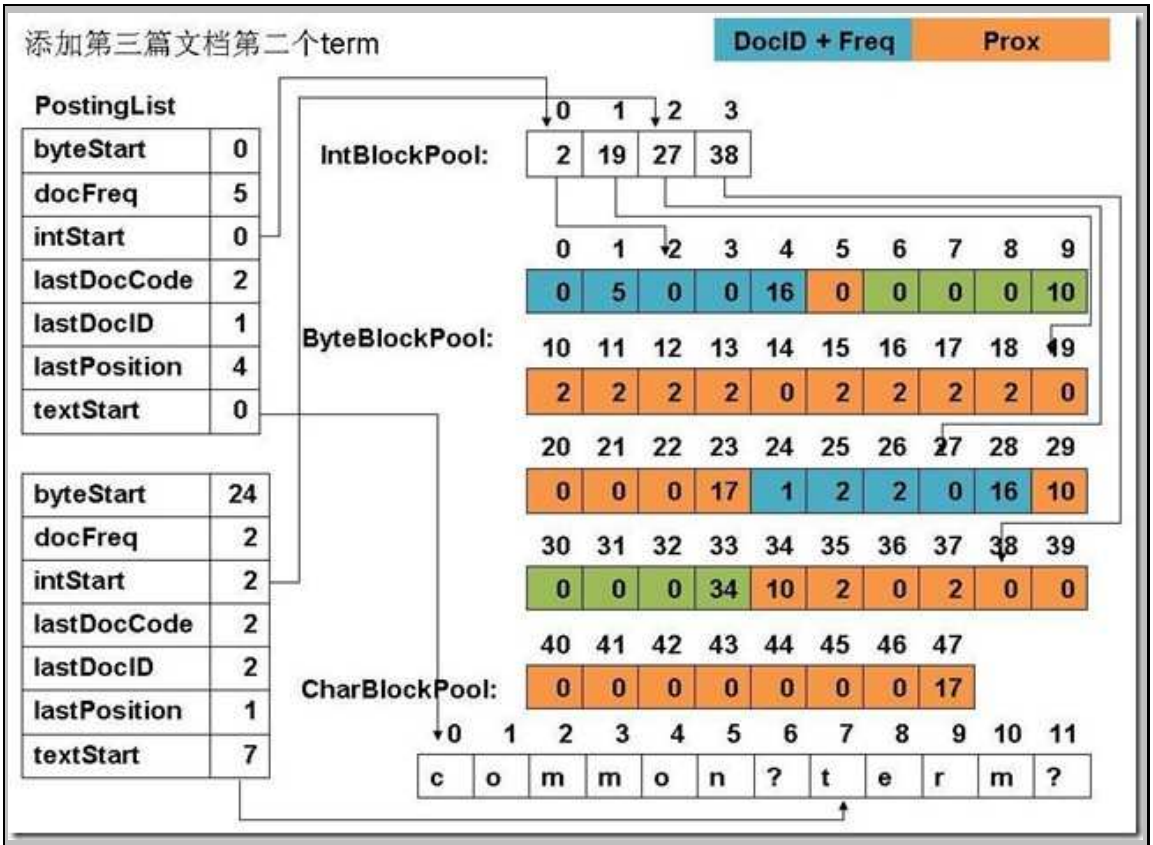
(8) 添加第三篇文档第二个 term

term 的位置信息已经填满了，必须分配新的块，层次为 2，大小为 14，结束符为 17，也即图中 34 到 47 的位置。

30 到 33 的四个 byte 组成一个 int 指针，指向第 34 个位置

原来 30 到 32 的三个 prox 信息移到 34 到 36 的位置。

在 37 处保存第三篇文档第二个 term 的位置信息，位置为 1，不跟随 payload，1<<1 + 0 = 2。

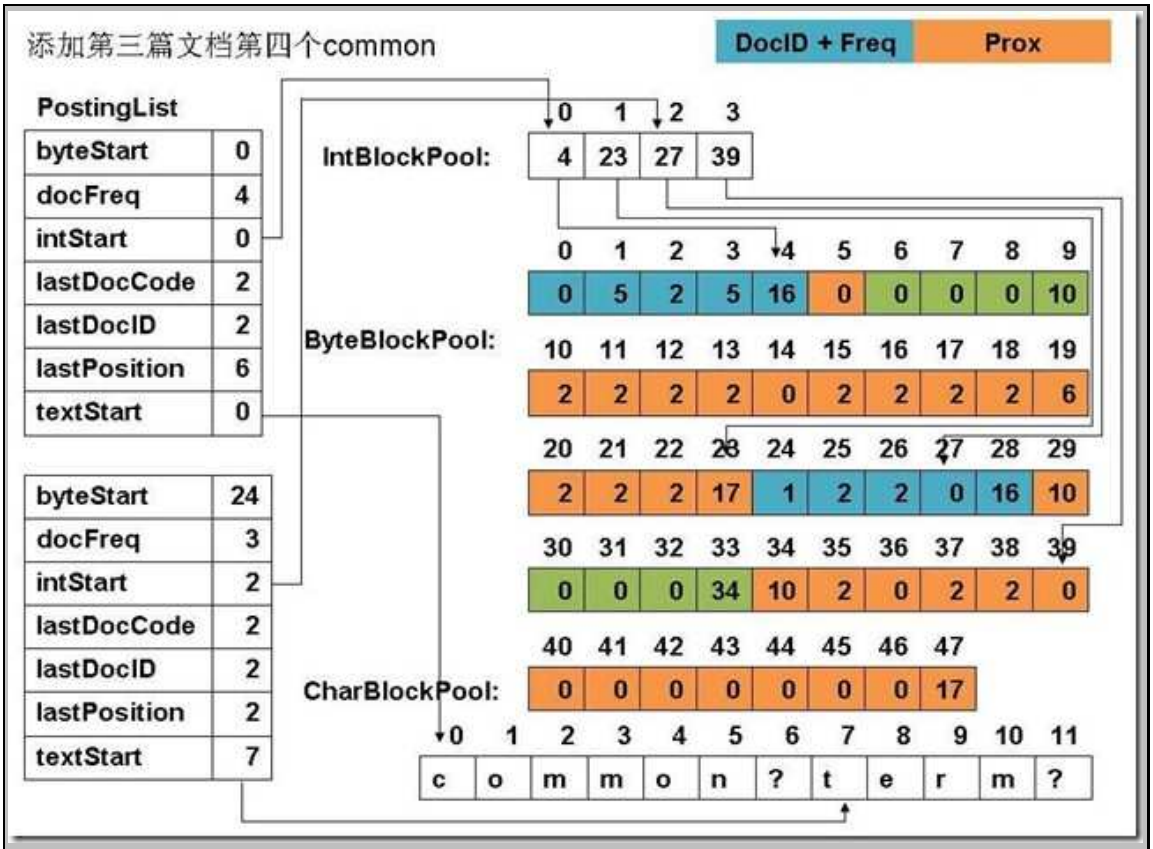


(9) 添加第三篇文档第四个 common

第二篇文档中"common"的 docid+freq 信息写入，文档号为 1，出现了 5 次，存储为[docid << 1 + 0, freq]，docid 取差值为 1，因而存储为 [2, 5]，即 2，3 的位置。

第三篇文档中前四个 common 的位置信息写入，即从 19 到 22 的[6, 2, 2, 2]，即出现在第 3 个，第 4 个，第 5 个，第 6 个位置。

第三篇文档中的第三个"term"的位置信息也写入，为 38 处的[2]。



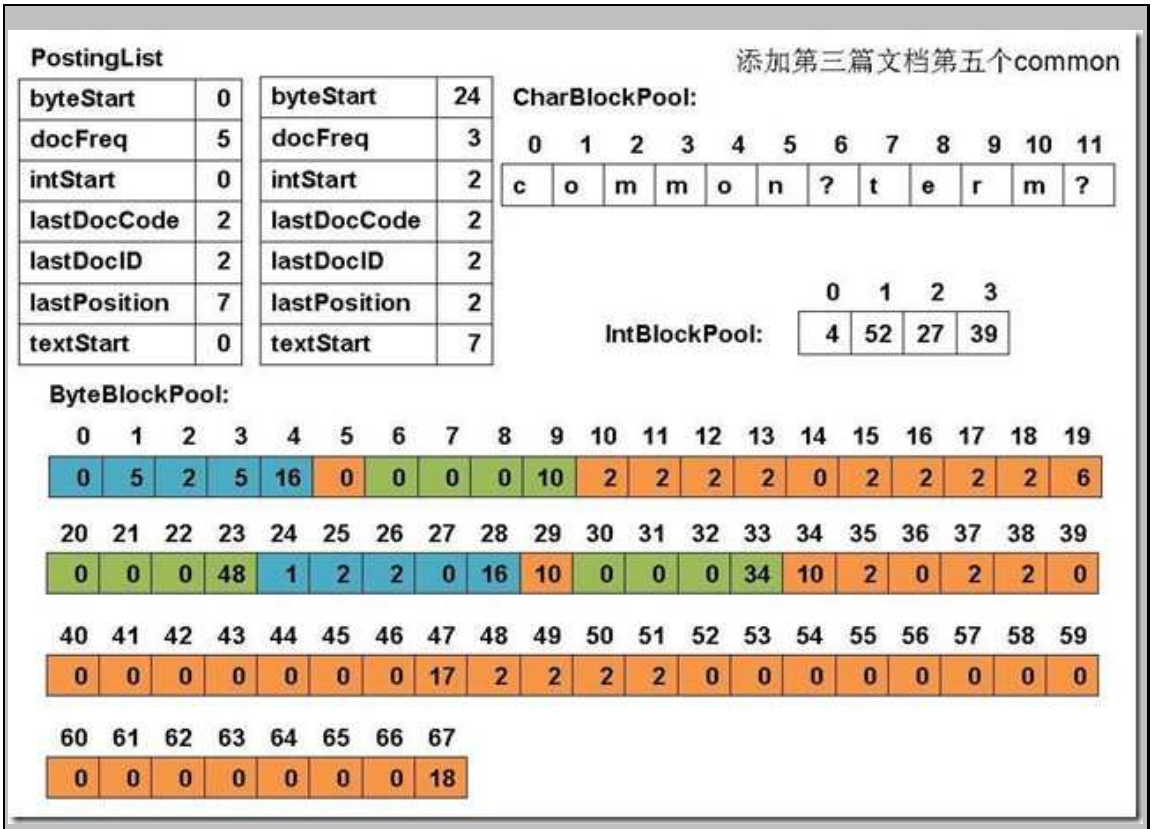
(10) 添加第三篇文档的第五个 common

虽然 common 已经分配了层次为 2，大小为 14 的第二个块(从 10 到 23)，不过还是用完了，需要在缓存的最后分配新的块，层次为 3，大小为 20，结束符为 18，也即从 48 到 67 的位置。

从 20 到 23 的四个 byte 组成一个 int 指针指向新分配的块。

原来 20 到 22 的数据移到 48 至 50 的位置。

第三篇文档的第五个 common 的位置信息写入，为第 51 个位置的[2]，也即紧跟上个 common，后面没有 payload 信息。



(11) 添加第四篇文档的第一个 term

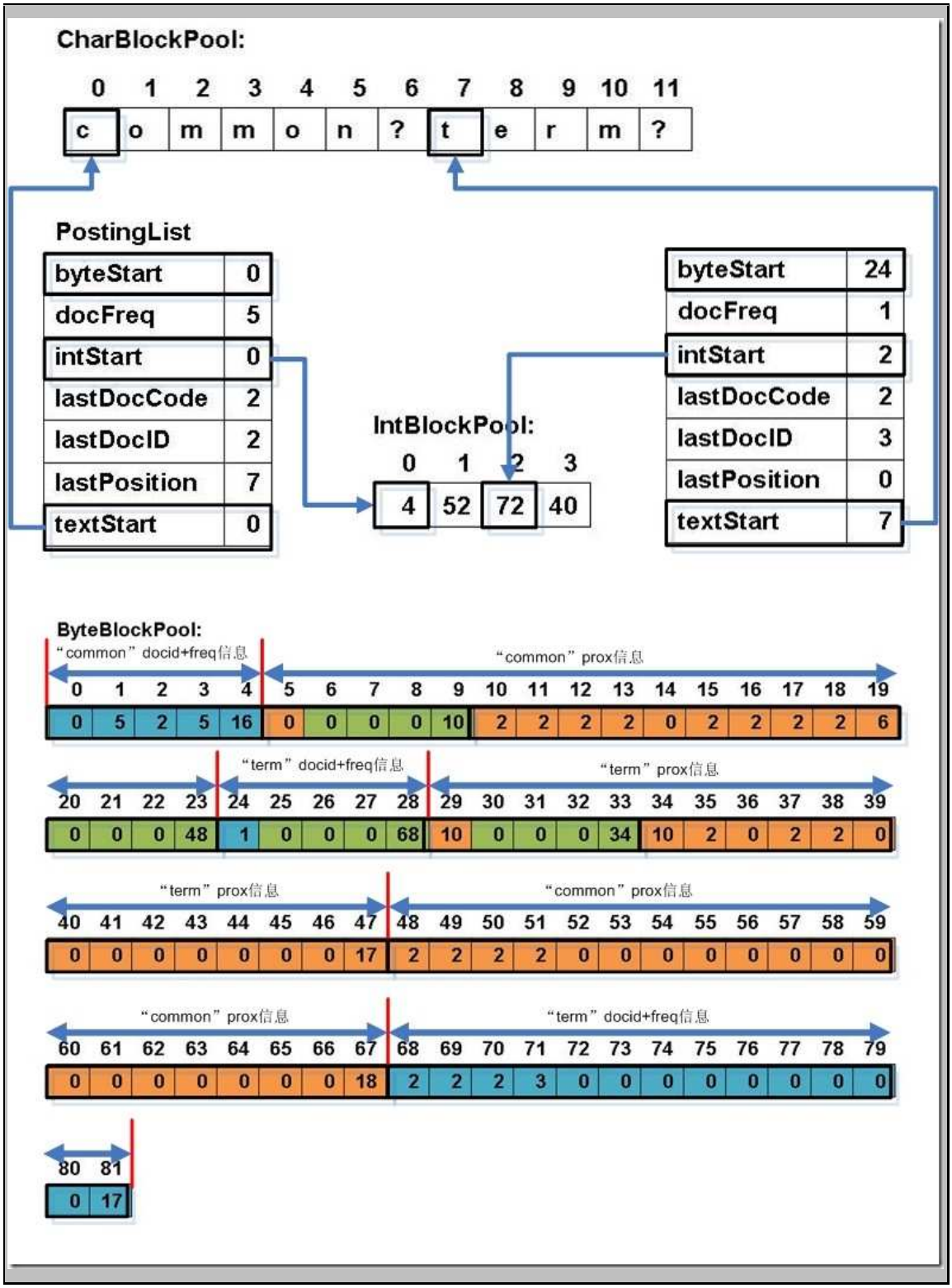
写入第三篇文档的 term 的 docid+freq 信息，文档号为 2，出现了三次，存储为[docid<<1+0, freq]，docid 取差值为 1，因而存储为[2, 3]。

然而存储 term 的 docid+freq 信息的块已经满了，需要在缓存的最后追加新的块，层次为 2，大小为 14，结束符为 17，即从 68 到 81 的位置。

从 25 到 28 的四个 byte 组成一个 int 指针指向新分配的块。

原来 25 到 26 的信息移到 68, 69 处，在 70, 71 处写入第三篇文档的 docid+freq 信息[2, 3]

(12) 最终 PostingList， CharBlockPool， IntBlockPool，ByteBlockPool 的关系如下图：



6、关闭 IndexWriter 对象

代码：

```
writer.close();
--> IndexWriter.closeInternal(boolean)
    --> (1) 将索引信息由内存写入磁盘: flush(waitForMerges, true, true);
    --> (2) 进行段合并: mergeScheduler.merge(this);
```

对段的合并将在后面的章节进行讨论，此处仅仅讨论将索引信息由写入磁盘的过程。

代码：

```
IndexWriter.flush(boolean triggerMerge, boolean flushDocStores, boolean flushDeletes)
--> IndexWriter.doFlush(boolean flushDocStores, boolean flushDeletes)
    --> IndexWriter.doFlushInternal(boolean flushDocStores, boolean flushDeletes)
```

将索引写入磁盘包括以下几个过程：

- 得到要写入的段名：String segment = docWriter.getSegment();
- DocumentsWriter 将缓存的信息写入段：docWriter.flush(flushDocStores);
- 生成新的段信息对象：newSegment = new SegmentInfo(segment, flushedDocCount, directory, false, true, docStoreOffset, docStoreSegment, docStoreIsCompoundFile, docWriter.hasProx());
- 准备删除文档：docWriter.pushDeletes();
- 生成 cfs 段：docWriter.createCompoundFile(segment);
- 删除文档：applyDeletes();

6.1、得到要写入的段名

代码：

```
SegmentInfo newSegment = null;
final int numDocs = docWriter.getNumDocsInRAM();//文档总数
String docStoreSegment = docWriter.getDocStoreSegment();//存储域和词向量所要要写入的段名，"_0"
int docStoreOffset = docWriter.getDocStoreOffset();//存储域和词向量要写入的段中的偏移量
String segment = docWriter.getSegment();//段名，"_0"
```

在 Lucene 的索引文件结构一章做过详细介绍，存储域和词向量可以和索引域存储在不同的段中。

6.2、将缓存的内容写入段

代码：

```
flushedDocCount = docWriter.flush(flushDocStores);
```

此过程又包含以下两个阶段；

- 按照基本索引链关闭存储域和词向量信息
- 按照基本索引链的结构将索引结果写入段

6.2.1、按照基本索引链关闭存储域和词向量信息

代码为：

```
closeDocStore();

flushState.numDocsInStore = 0;
```

其主要是根据基本索引链结构，关闭存储域和词向量信息：

- consumer(**DocFieldProcessor**).closeDocStore(flushState);
  - consumer(**DocInverter**).closeDocStore(state);
    - consumer(**TermsHash**).closeDocStore(state);
      - consumer(**FreqProxTermsWriter**).closeDocStore(state);
      - if (nextTermsHash != null) nextTermsHash.closeDocStore(state);
        - consumer(**TermVectorsTermsWriter**).closeDocStore(state);
    - endConsumer(**NormsWriter**).closeDocStore(state);
  - fieldsWriter(**StoredFieldsWriter**).closeDocStore(state);

其中有实质意义的是以下两个 closeDocStore:

- 词向量的关闭：TermVectorsTermsWriter.closeDocStore(SegmentWriteState)

```
void closeDocStore(final SegmentWriteState state) throws IOException {

            if (tvx != null) {

       //为不保存词向量的文档在 tvd 文件中写入零。即便不保存词向量，在 tvx, tvd 中也保
```

```
留一个位置
        fill(state.numDocsInStore - docWriter.getDocStoreOffset());
        //关闭 tvx, tvf, tvd 文件的写入流
        tvx.close();
        tvf.close();
        tvd.close();
        tvx = null;
        //记录写入的文件名，为以后生成 cfs 文件的时候，将这些写入的文件生成一个统一的
cfs 文件。
        state.flushedFiles.add(state.docStoreSegmentName + "." +
IndexFileNames.VECTORS_INDEX_EXTENSION);
        state.flushedFiles.add(state.docStoreSegmentName + "." +
IndexFileNames.VECTORS_FIELDS_EXTENSION);
        state.flushedFiles.add(state.docStoreSegmentName + "." +
IndexFileNames.VECTORS_DOCUMENTS_EXTENSION);
        //从 DocumentsWriter 的成员变量 openFiles 中删除，未来可能被 IndexFileDeleter 删除
        docWriter.removeOpenFile(state.docStoreSegmentName + "." +
IndexFileNames.VECTORS_INDEX_EXTENSION);
        docWriter.removeOpenFile(state.docStoreSegmentName + "." +
IndexFileNames.VECTORS_FIELDS_EXTENSION);
        docWriter.removeOpenFile(state.docStoreSegmentName + "." +
IndexFileNames.VECTORS_DOCUMENTS_EXTENSION);
        lastDocID = 0;
      }
}
```

- 存储域的关闭：StoredFieldsWriter.closeDocStore(SegmentWriteState)

```
public void closeDocStore(SegmentWriteState state) throws IOException {
   //关闭 fdx, fdt 写入流
```

```
    fieldsWriter.close();
    --> fieldsStream.close();
    --> indexStream.close();
    fieldsWriter = null;
    lastDocID = 0;
    //记录写入的文件名
    state.flushedFiles.add(state.docStoreSegmentName + "." +
IndexFileNames.FIELDS_EXTENSION);
    state.flushedFiles.add(state.docStoreSegmentName + "." +
IndexFileNames.FIELDS_INDEX_EXTENSION);
    state.docWriter.removeOpenFile(state.docStoreSegmentName + "." +
IndexFileNames.FIELDS_EXTENSION);
    state.docWriter.removeOpenFile(state.docStoreSegmentName + "." +
IndexFileNames.FIELDS_INDEX_EXTENSION);
}
```

6.2.2、按照基本索引链的结构将索引结果写入段

代码为：

```
consumer(DocFieldProcessor).flush(threads, flushState);
    //回收 fieldHash，以便用于下一轮的索引，为提高效率，索引链中的对象是被复用的。
    Map<DocFieldConsumerPerThread, Collection<DocFieldConsumerPerField>>
childThreadsAndFields = new HashMap<DocFieldConsumerPerThread,
Collection<DocFieldConsumerPerField>>();
    for ( DocConsumerPerThread thread : threads) {
        DocFieldProcessorPerThread perThread = (DocFieldProcessorPerThread) thread;
        childThreadsAndFields.put(perThread.consumer, perThread.fields());
        perThread.trimFields(state);
```

```
}
//写入存储域
--> fieldsWriter(StoredFieldsWriter).flush(state);
//写入索引域
--> consumer(DocInverter).flush(childThreadsAndFields, state);
//写入域元数据信息，并记录写入的文件名，以便以后生成 cfs 文件
--> final String fileName = state.segmentFileName(IndexFileNames.FIELD_INFOS_EXTENSION);
--> fieldInfos.write(state.directory, fileName);
--> state.flushedFiles.add(fileName);
```

此过程也是按照基本索引链来的：

- consumer(**DocFieldProcessor**).flush(…);
  - consumer(**DocInverter**).flush(…);
    - consumer(**TermsHash**).flush(…);
      - consumer(**FreqProxTermsWriter**).flush(…);
      - if (nextTermsHash != null) nextTermsHash.flush(…);
        - consumer(**TermVectorsTermsWriter**).flush(…);
    - endConsumer(**NormsWriter**).flush(…);
  - fieldsWriter(**StoredFieldsWriter**).flush(…);

6.2.2.1、写入存储域

代码为：

```
StoredFieldsWriter.flush(SegmentWriteState state) {
  if (state.numDocsInStore > 0) {
    initFieldsWriter();
    fill(state.numDocsInStore - docWriter.getDocStoreOffset());
  }
  if (fieldsWriter != null)
    fieldsWriter.flush();
}
```

从代码中可以看出，是写入 fdx, fdt 两个文件，但是在上述的 closeDocStore 已经写入了，并

且把 state.numDocsInStore 置零，fieldsWriter 设为 null，在这里其实什么也不做。

6.2.2.2、写入索引域

代码为：

```
DocInverter.flush(Map<DocFieldConsumerPerThread,Collection<DocFieldConsumerPerField>>,
SegmentWriteState)
   //写入倒排表及词向量信息
   --> consumer(TermsHash).flush(childThreadsAndFields, state);
   //写入标准化因子
   --> endConsumer(NormsWriter).flush(endChildThreadsAndFields, state);
```

6.2.2.2.1、写入倒排表及词向量信息

代码为：

```
TermsHash.flush(Map<InvertedDocConsumerPerThread,Collection<InvertedDocConsumerPerField>>,
SegmentWriteState)
   //写入倒排表信息
   --> consumer(FreqProxTermsWriter).flush(childThreadsAndFields, state);
  //回收 RawPostingList
   --> shrinkFreePostings(threadsAndFields, state);
   //写入词向量信息
   --> if (nextTermsHash != null) nextTermsHash.flush(nextThreadsAndFields, state);
       --> consumer(TermVectorsTermsWriter).flush(childThreadsAndFields, state);
```

6.2.2.2.1.1、写入倒排表信息

代码为：

```
FreqProxTermsWriter.flush(Map<TermsHashConsumerPerThread,
                          Collection<TermsHashConsumerPerField>>, SegmentWriteState)
   (a) 所有域按名称排序，使得同名域能够一起处理
   Collections.sort(allFields);
```

```
final int numAllFields = allFields.size();
(b) 生成倒排表的写对象
final FormatPostingsFieldsConsumer consumer = new FormatPostingsFieldsWriter(state, fieldInfos);
int start = 0;
(c) 对于每一个域
while(start < numAllFields) {
  (c-1) 找出所有的同名域
  final FieldInfo fieldInfo = allFields.get(start).fieldInfo;
  final String fieldName = fieldInfo.name;
  int end = start+1;
  while(end < numAllFields && allFields.get(end).fieldInfo.name.equals(fieldName))
    end++;
  FreqProxTermsWriterPerField[] fields = new FreqProxTermsWriterPerField[end-start];
  for(int i=start;i<end;i++) {
    fields[i-start] = allFields.get(i);
    fieldInfo.storePayloads |= fields[i-start].hasPayloads;
  }
  (c-2) 将同名域的倒排表添加到文件
  appendPostings(fields, consumer);
  (c-3) 释放空间
  for(int i=0;i<fields.length;i++) {
    TermsHashPerField perField = fields[i].termsHashPerField;
    int numPostings = perField.numPostings;
    perField.reset();
    perField.shrinkHash(numPostings);
    fields[i].reset();
  }
```

```
    start = end;
  }
  (d) 关闭倒排表的写对象
  consumer.finish();
```

(b) 生成倒排表的写对象

代码为：

```
public FormatPostingsFieldsWriter(SegmentWriteState state, FieldInfos fieldInfos) throws IOException {
  dir = state.directory;
  segment = state.segmentName;
  totalNumDocs = state.numDocs;
  this.fieldInfos = fieldInfos;
  //用于写 tii,tis
  termsOut = new TermInfosWriter(dir, segment, fieldInfos, state.termIndexInterval);
  //用于写 freq, prox 的跳表
  skipListWriter = new DefaultSkipListWriter(termsOut.skipInterval, termsOut.maxSkipLevels, totalNumDocs, null, null);
  //记录写入的文件名，
  state.flushedFiles.add(state.segmentFileName(IndexFileNames.TERMS_EXTENSION));
  state.flushedFiles.add(state.segmentFileName(IndexFileNames.TERMS_INDEX_EXTENSION));
  //用以上两个写对象，按照一定的格式写入段
  termsWriter = new FormatPostingsTermsWriter(state, this);
}
```

对象结构如下：

```
consumer  FormatPostingsFieldsWriter  (id=119)  //用于处理一个域
  dir  SimpleFSDirectory  (id=126)  //目标索引文件夹
  totalNumDocs  8  //文档总数
  fieldInfos  FieldInfos  (id=70)  //域元数据信息
```

segment "_0" //段名

skipListWriter DefaultSkipListWriter (id=133) //freq, prox 中跳表的写对象

termsOut TermInfosWriter (id=125) //tii, tis 文件的写对象

termsWriter **FormatPostingsTermsWriter** (id=135) //用于添加词(Term)

currentTerm null

currentTermStart 0

fieldInfo null

freqStart 0

proxStart 0

termBuffer null

termsOut TermInfosWriter (id=125)

docsWriter **FormatPostingsDocsWriter** (id=139) //用于写入此词的 docid, freq 信息

df 0

fieldInfo null

freqStart 0

lastDocID 0

omitTermFreqAndPositions false

out SimpleFSDirectory$SimpleFSIndexOutput (id=144)

skipInterval 16

skipListWriter DefaultSkipListWriter (id=133)

storePayloads false

termInfo TermInfo (id=151)

totalNumDocs 8

posWriter **FormatPostingsPositionsWriter** (id=146) //用于写入此词在此文档中的位置信息

lastPayloadLength -1

lastPosition 0

```
omitTermFreqAndPositions    false
out    SimpleFSDirectory$SimpleFSIndexOutput   (id=157)
parent    FormatPostingsDocsWriter   (id=139)
storePayloads    false
```

- FormatPostingsFieldsWriter.addField(FieldInfo field)用于添加索引域信息，其返回FormatPostingsTermsConsumer用于添加词信息
- FormatPostingsTermsConsumer.addTerm(char[] text, int start)用于添加词信息，其返回FormatPostingsDocsConsumer用于添加freq信息
- FormatPostingsDocsConsumer.addDoc(int docID, int termDocFreq)用于添加freq信息，其返回FormatPostingsPositionsConsumer用于添加prox信息
- FormatPostingsPositionsConsumer.addPosition(int position, byte[] payload, int payloadOffset, int payloadLength)用于添加prox信息

(c-2) 将同名域的倒排表添加到文件

代码为：

```
FreqProxTermsWriter.appendPostings(FreqProxTermsWriterPerField[],
FormatPostingsFieldsConsumer) {
    int numFields = fields.length;
    final FreqProxFieldMergeState[] mergeStates = new FreqProxFieldMergeState[numFields];
    for(int i=0;i<numFields;i++) {
      FreqProxFieldMergeState fms = mergeStates[i] = new FreqProxFieldMergeState(fields[i]);
      boolean result = fms.nextTerm(); //对所有的域，取第一个词(Term)
    }
    (1) 添加此域，虽然有多个域，但是由于是同名域，只取第一个域的信息即可。返回的是用于添加此域中的词的对象。
    final FormatPostingsTermsConsumer termsConsumer = consumer.addField(fields[0].fieldInfo);
    FreqProxFieldMergeState[] termStates = new FreqProxFieldMergeState[numFields];
    final boolean currentFieldOmitTermFreqAndPositions =
fields[0].fieldInfo.omitTermFreqAndPositions;
```

（2）此 while 循环是遍历每一个尚有未处理的词的域，依次按照词典顺序处理这些域所包含的词。当一个域中的所有的词都被处理过后，则 numFields 减一，并从 mergeStates 数组中移除此域。直到所有的域的所有的词都处理完毕，方才退出此循环。

```
while(numFields > 0) {
```

（2-1）找出所有域中按字典顺序的下一个词。可能多个同名域中，都包含同一个 term，因而要遍历所有的 numFields，得到所有的域里的下一个词，numToMerge 即为有多少个域包含此词。

```
termStates[0] = mergeStates[0];
int numToMerge = 1;
for(int i=1;i<numFields;i++) {
 final char[] text = mergeStates[i].text;
 final int textOffset = mergeStates[i].textOffset;
 final int cmp = compareText(text, textOffset, termStates[0].text, termStates[0].textOffset);
 if (cmp < 0) {
  termStates[0] = mergeStates[i];
  numToMerge = 1;
 } else if (cmp == 0)
  termStates[numToMerge++] = mergeStates[i];
}
```

（2-2）添加此词，返回 FormatPostingsDocsConsumer 用于添加文档号(doc ID)及词频信息(freq)

```
final FormatPostingsDocsConsumer docConsumer =
termsConsumer.addTerm(termStates[0].text, termStates[0].textOffset);
```

（2-3）由于共 numToMerge 个域都包含此词，每个词都有一个链表的文档号表示包含这些词的文档。此循环遍历所有的包含此词的域，依次按照从小到大的循序添加包含此词的文档号及词频信息。当一个域中对此词的所有文档号都处理过了，则 numToMerge 减一，并从 termStates 数组中移除此域。当所有包含此词的域的所有文档号都处理过了，则结束此循环。

```
while(numToMerge > 0) {
```

（2-3-1）找出最小的文档号

```
      FreqProxFieldMergeState minState = termStates[0];
      for(int i=1;i<numToMerge;i++)
        if (termStates[i].docID < minState.docID)
          minState = termStates[i];
      final int termDocFreq = minState.termFreq;
      (2-3-2) 添加文档号及词频信息，并形成跳表，返回
FormatPostingsPositionsConsumer 用于添加位置(prox)信息
      final FormatPostingsPositionsConsumer posConsumer =
docConsumer.addDoc(minState.docID, termDocFreq);
      //ByteSliceReader 是用于读取 bytepool 中的 prox 信息的。
      final ByteSliceReader prox = minState.prox;
      if (!currentFieldOmitTermFreqAndPositions) {
        int position = 0;
        (2-3-3) 此循环对包含此词的文档，添加位置信息
        for(int j=0;j<termDocFreq;j++) {
          final int code = prox.readVInt();
          position += code >> 1;
          final int payloadLength;
          //  如果此位置有 payload 信息，则从 bytepool 中读出，否则设为零。
          if ((code & 1) != 0) {
            payloadLength = prox.readVInt();
            if (payloadBuffer == null || payloadBuffer.length < payloadLength)
              payloadBuffer = new byte[payloadLength];
            prox.readBytes(payloadBuffer, 0, payloadLength);
          } else
            payloadLength = 0;
          //添加位置(prox)信息
          posConsumer.addPosition(position, payloadBuffer, 0, payloadLength);
```

```
      }
      posConsumer.finish();
    }
```

(2-3-4) 判断退出条件，上次选中的域取得下一个文档号，如果没有，则说明此域包含此词的文档已经处理完毕，则从 termStates 中删除此域，并将 numToMerge 减一。然后此域取得下一个词，当循环到(2)的时候，表明此域已经开始处理下一个词。如果没有下一个词，说明此域中的所有的词都处理完毕，则从 mergeStates 中删除此域，并将 numFields 减一，当 numFields 为 0 的时候，循环(2)也就结束了。

```
    if (!minState.nextDoc()) {//获得下一个 docid
      //如果此域包含此词的文档已经没有下一篇 docid，则从数组 termStates 中移除，
numToMerge 减一。
      int upto = 0;
      for(int i=0;i<numToMerge;i++)
        if (termStates[i] != minState)
          termStates[upto++] = termStates[i];
      numToMerge--;
      //此域则取下一个词(term)，在循环(2)处来参与下一个词的合并
      if (!minState.nextTerm()) {
        //如果此域没有下一个词了，则此域从数组 mergeStates 中移除，numFields 减一。
        upto = 0;
        for(int i=0;i<numFields;i++)
          if (mergeStates[i] != minState)
            mergeStates[upto++] = mergeStates[i];
        numFields--;
      }
    }
  }
```

(2-4) 经过上面的过程，docid 和 freq 信息虽已经写入段文件，而跳表信息并没有写到文件中，

```
而是写入 skip buffer 里面了，此处真正写入文件。并且词典(tii, tis)也应该写入文件。
    docConsumer(FormatPostingsDocsWriter).finish();
  }
  termsConsumer.finish();
 }
```

(2-3-4)获得下一篇文档号代码如下：

```
public boolean nextDoc() {//如何获取下一个 docid
 if (freq.eof()) {//如果 bytepool 中的 freq 信息已经读完
  if (p.lastDocCode != -1) {//由上述缓存管理，PostingList 里面还存着最后一篇文档的文档号及词频信息，则将最后一篇文档返回
    docID = p.lastDocID;
    if (!field.omitTermFreqAndPositions)
     termFreq = p.docFreq;
    p.lastDocCode = -1;
    return true;
  } else
    return false;//没有下一篇文档
 }
 final int code = freq.readVInt();//如果 bytepool 中的 freq 信息尚未读完
 if (field.omitTermFreqAndPositions)
  docID += code;
 else {
  //读出文档号及词频信息。
  docID += code >>> 1;
  if ((code & 1) != 0)
   termFreq = 1;
  else
   termFreq = freq.readVInt();
```

```
  }
  return true;
}
```

（2-3-2）添加文档号及词频信息代码如下：

```
FormatPostingsPositionsConsumer FormatPostingsDocsWriter.addDoc(int docID, int termDocFreq)
{
    final int delta = docID - lastDocID;
    //当文档数量达到 skipInterval 倍数的时候，添加跳表项。
    if ((++df % skipInterval) == 0) {
      skipListWriter.setSkipData(lastDocID, storePayloads, posWriter.lastPayloadLength);
      skipListWriter.bufferSkip(df);
    }
    lastDocID = docID;
    if (omitTermFreqAndPositions)
      out.writeVInt(delta);
    else if (1 == termDocFreq)
      out.writeVInt((delta<<1) | 1);
    else {
      //写入文档号及词频信息。
      out.writeVInt(delta<<1);
      out.writeVInt(termDocFreq);
    }
    return posWriter;
}
```

（2-3-3）添加位置信息：

```
FormatPostingsPositionsWriter.addPosition(int position, byte[] payload, int payloadOffset, int
payloadLength) {
    final int delta = position - lastPosition;
```

```
    lastPosition = position;
    if (storePayloads) {
      //保存位置及 payload 信息
      if (payloadLength != lastPayloadLength) {
        lastPayloadLength = payloadLength;
        out.writeVInt((delta<<1)|1);
        out.writeVInt(payloadLength);
      } else
        out.writeVInt(delta << 1);
        if (payloadLength > 0)
          out.writeBytes(payload, payloadLength);
    } else
      out.writeVInt(delta);
}
```

(2-4) 将跳表和词典(tii, tis)写入文件

```
FormatPostingsDocsWriter.finish() {
    //将跳表缓存写入文件
    long skipPointer = skipListWriter.writeSkip(out);
    if (df > 0) {
      //将词典(terminfo)写入 tii,tis 文件
     parent.termsOut(TermInfosWriter).add(fieldInfo.number, utf8.result, utf8.length, termInfo);
    }
  }
```

将跳表缓存写入文件：

```
DefaultSkipListWriter(MultiLevelSkipListWriter).writeSkip(IndexOutput)   {
    long skipPointer = output.getFilePointer();
    if (skipBuffer == null || skipBuffer.length == 0) return skipPointer;
    //正如我们在索引文件格式中分析的那样， 高层在前，低层在后，除最低层外，其他的层
```

```
都有长度保存。
    for (int level = numberOfSkipLevels - 1; level > 0; level--) {
      long length = skipBuffer[level].getFilePointer();
      if (length > 0) {
        output.writeVLong(length);
        skipBuffer[level].writeTo(output);
      }
    }
    //写入最低层
    skipBuffer[0].writeTo(output);
    return skipPointer;
  }
```

将词典(terminfo)写入 tii,tis 文件：

- tii 文件是 tis 文件的类似跳表的东西，是在 tis 文件中每隔 indexInterval 个词提取出一个词放在 tii 文件中，以便很快的查找到词。
- 因而 TermInfosWriter 类型中有一个成员变量 other 也是 TermInfosWriter 类型的，还有一个成员变量 isIndex 来表示此对象是用来写 tii 文件的还是用来写 tis 文件的。
- 如果一个 TermInfosWriter 对象的 isIndex=false 则，它是用来写 tis 文件的，它的 other 指向的是用来写 tii 文件的 TermInfosWriter 对象
- 如果一个 TermInfosWriter 对象的 isIndex=true 则，它是用来写 tii 文件的，它的 other 指向的是用来写 tis 文件的 TermInfosWriter 对象

```
TermInfosWriter.add (int fieldNumber, byte[] termBytes, int termBytesLength, TermInfo ti) {
    //如果词的总数是 indexInterval 的倍数，则应该写入 tii 文件
    if (!isIndex && size % indexInterval == 0)
      other.add(lastFieldNumber, lastTermBytes, lastTermBytesLength, lastTi);
    //将词写入 tis 文件
    writeTerm(fieldNumber, termBytes, termBytesLength);
    output.writeVInt(ti.docFreq);                       // write doc freq
```

```
    output.writeVLong(ti.freqPointer - lastTi.freqPointer); // write pointers
    output.writeVLong(ti.proxPointer - lastTi.proxPointer);
    if (ti.docFreq >= skipInterval) {
      output.writeVInt(ti.skipOffset);
    }
    if (isIndex) {
      output.writeVLong(other.output.getFilePointer() - lastIndexPointer);
      lastIndexPointer = other.output.getFilePointer(); // write pointer
    }
    lastFieldNumber = fieldNumber;
    lastTi.set(ti);
    size++;
  }
```

6.2.2.2.1.2、写入词向量信息

代码为：

```
TermVectorsTermsWriter.flush
(Map<TermsHashConsumerPerThread,Collection<TermsHashConsumerPerField>>
                                    threadsAndFields, final SegmentWriteState state) {
    if (tvx != null) {
      if (state.numDocsInStore > 0)
        fill(state.numDocsInStore - docWriter.getDocStoreOffset());
      tvx.flush();
      tvd.flush();
      tvf.flush();
    }
    for (Map.Entry<TermsHashConsumerPerThread,Collection<TermsHashConsumerPerField>> entry :
                                                                        threadsAndFields.entrySet())
{
```

```
    for (final TermsHashConsumerPerField field : entry.getValue() ) {
      TermVectorsTermsWriterPerField perField = (TermVectorsTermsWriterPerField) field;
      perField.termsHashPerField.reset();
      perField.shrinkHash();
    }
    TermVectorsTermsWriterPerThread perThread = (TermVectorsTermsWriterPerThread)
entry.getKey();
    perThread.termsHashPerThread.reset(true);
   }
  }
```

从代码中可以看出，是写入 tvx, tvd, tvf 三个文件，但是在上述的 closeDocStore 已经写入了，并且把 tvx 设为 null，在这里其实什么也不做，仅仅是清空 postingsHash，以便进行下一轮索引时重用此对象。

6.2.2.2.2、写入标准化因子

代码为：

```
NormsWriter.flush(Map<InvertedDocEndConsumerPerThread,Collection<InvertedDocEndConsumerPerField>>
threadsAndFields, SegmentWriteState state) {
   final Map<FieldInfo,List<NormsWriterPerField>> byField = new
HashMap<FieldInfo,List<NormsWriterPerField>>();
   for (final Map.Entry<InvertedDocEndConsumerPerThread,Collection<InvertedDocEndConsumerPerField>>
entry : threadsAndFields.entrySet()) {
    //遍历所有的域，将同名域对应的 NormsWriterPerField 放到同一个链表中。
     final Collection<InvertedDocEndConsumerPerField> fields = entry.getValue();
     final Iterator<InvertedDocEndConsumerPerField> fieldsIt = fields.iterator();
     while (fieldsIt.hasNext()) {
       final NormsWriterPerField perField = (NormsWriterPerField) fieldsIt.next();
       List<NormsWriterPerField> l = byField.get(perField.fieldInfo);
       if (l == null) {
```

```
      l = new ArrayList<NormsWriterPerField>();
      byField.put(perField.fieldInfo, l);
    }
    l.add(perField);
  }
  //记录写入的文件名，方便以后生成 cfs 文件。
  final String normsFileName = state.segmentName + "." + IndexFileNames.NORMS_EXTENSION;
  state.flushedFiles.add(normsFileName);
  IndexOutput normsOut = state.directory.createOutput(normsFileName);
  try {
   //写入 nrm 文件头
   normsOut.writeBytes(SegmentMerger.NORMS_HEADER, 0, SegmentMerger.NORMS_HEADER.length);
   final int numField = fieldInfos.size();
   int normCount = 0;
   //对每一个域进行处理
   for(int fieldNumber=0;fieldNumber<numField;fieldNumber++) {
    final FieldInfo fieldInfo = fieldInfos.fieldInfo(fieldNumber);
    //得到同名域的链表
    List<NormsWriterPerField> toMerge = byField.get(fieldInfo);
    int upto = 0;
    if (toMerge != null) {
     final int numFields = toMerge.size();
     normCount++;
     final NormsWriterPerField[] fields = new NormsWriterPerField[numFields];
     int[] uptos = new int[numFields];
     for(int j=0;j<numFields;j++)
      fields[j] = toMerge.get(j);
     int numLeft = numFields;
```

```
//处理同名的多个域
while(numLeft > 0) {
 //得到所有的同名域中最小的文档号
 int minLoc = 0;
 int minDocID = fields[0].docIDs[uptos[0]];
 for(int j=1;j<numLeft;j++) {
  final int docID = fields[j].docIDs[uptos[j]];
  if (docID < minDocID) {
   minDocID = docID;
   minLoc = j;
  }
 }
 // 在 nrm 文件中，每一个文件都有一个位置，没有设定的，放入默认值
 for(;upto<minDocID;upto++)
  normsOut.writeByte(defaultNorm);
 //写入当前的 nrm 值
 normsOut.writeByte(fields[minLoc].norms[uptos[minLoc]]);
 (uptos[minLoc])++;
 upto++;
 //如果当前域的文档已经处理完毕，则 numLeft 减一，归零时推出循环。
 if (uptos[minLoc] == fields[minLoc].upto) {
  fields[minLoc].reset();
  if (minLoc != numLeft-1) {
   fields[minLoc] = fields[numLeft-1];
   uptos[minLoc] = uptos[numLeft-1];
  }
  numLeft--;
 }
```

```
      }
      // 对所有的未设定 nrm 值的文档写入默认值。
      for(;upto<state.numDocs;upto++)
        normsOut.writeByte(defaultNorm);
      } else if (fieldInfo.isIndexed && !fieldInfo.omitNorms) {
        normCount++;
        // Fill entire field with default norm:
        for(;upto<state.numDocs;upto++)
          normsOut.writeByte(defaultNorm);
      }
    }
  } finally {
    normsOut.close();
  }
}
```

6.2.2.3、写入域元数据

代码为：

```
FieldInfos.write(IndexOutput) {
    output.writeVInt(CURRENT_FORMAT);
    output.writeVInt(size());
    for (int i = 0; i < size(); i++) {
      FieldInfo fi = fieldInfo(i);
      byte bits = 0x0;
      if (fi.isIndexed) bits |= IS_INDEXED;
      if (fi.storeTermVector) bits |= STORE_TERMVECTOR;
      if (fi.storePositionWithTermVector) bits |= STORE_POSITIONS_WITH_TERMVECTOR;
      if (fi.storeOffsetWithTermVector) bits |= STORE_OFFSET_WITH_TERMVECTOR;
      if (fi.omitNorms) bits |= OMIT_NORMS;
```

```
      if (fi.storePayloads) bits |= STORE_PAYLOADS;
      if (fi.omitTermFreqAndPositions) bits |= OMIT_TERM_FREQ_AND_POSITIONS;
      output.writeString(fi.name);
      output.writeByte(bits);
    }
}
```

此处基本就是按照 fnm 文件的格式写入的。

6.3、生成新的段信息对象

代码：

```
newSegment = new SegmentInfo(segment, flushedDocCount, directory, false, true, docStoreOffset,
docStoreSegment, docStoreIsCompoundFile, docWriter.hasProx());
segmentInfos.add(newSegment);
```

6.4、准备删除文档

代码：

```
docWriter.pushDeletes();
    --> deletesFlushed.update(deletesInRAM);
```

此处将 deletesInRAM 全部加到 deletesFlushed 中，并把 deletesInRAM 清空。原因上面已经阐明。

6.5、生成 cfs 段

代码：

```
docWriter.createCompoundFile(segment);
newSegment.setUseCompoundFile(true);
```

代码为：

```
DocumentsWriter.createCompoundFile(String segment) {
    CompoundFileWriter cfsWriter = new CompoundFileWriter(directory, segment + "." +
IndexFileNames.COMPOUND_FILE_EXTENSION);
   //将上述中记录的文档名全部加入 cfs 段的写对象。
   for (final String flushedFile : flushState.flushedFiles)
     cfsWriter.addFile(flushedFile);
   cfsWriter.close();
  }
```

6.6、删除文档

代码：

```
applyDeletes();
```

代码为：

```
boolean applyDeletes(SegmentInfos infos) {
  if (!hasDeletes())
    return false;
  final int infosEnd = infos.size();
  int docStart = 0;
  boolean any = false;
  for (int i = 0; i < infosEnd; i++) {
    assert infos.info(i).dir == directory;
    SegmentReader reader = writer.readerPool.get(infos.info(i), false);
    try {
      any |= applyDeletes(reader, docStart);
      docStart += reader.maxDoc();
    } finally {
```

```
    writer.readerPool.release(reader);
   }
  }
  deletesFlushed.clear();
  return any;
}
```

- Lucene 删除文档可以用 reader，也可以用 writer，但是归根结底还是用 reader 来删除的。
- reader 的删除有以下三种方式：
  - 按照词删除，删除所有包含此词的文档。
  - 按照文档号删除。
  - 按照查询对象删除，删除所有满足此查询的文档。
- 但是这三种方式归根结底还是按照文档号删除，也就是写.del 文件的过程。

```
private final synchronized boolean applyDeletes(IndexReader reader, int docIDStart)
  throws CorruptIndexException, IOException {
  final int docEnd = docIDStart + reader.maxDoc();
  boolean any = false;
  //按照词删除，删除所有包含此词的文档。
  TermDocs docs = reader.termDocs();
  try {
   for (Entry<Term, BufferedDeletes.Num> entry: deletesFlushed.terms.entrySet()) {
    Term term = entry.getKey();
    docs.seek(term);
    int limit = entry.getValue().getNum();
    while (docs.next()) {
     int docID = docs.doc();
     if (docIDStart+docID >= limit)
      break;
     reader.deleteDocument(docID);
     any = true;
```

```
    }
   }
  } finally {
   docs.close();
  }
  //按照文档号删除。
  for (Integer docIdInt : deletesFlushed.docIDs) {
   int docID = docIdInt.intValue();
   if (docID >= docIDStart && docID < docEnd) {
    reader.deleteDocument(docID-docIDStart);
    any = true;
   }
  }
  //按照查询对象删除，删除所有满足此查询的文档。
  IndexSearcher searcher = new IndexSearcher(reader);
  for (Entry<Query, Integer> entry : deletesFlushed.queries.entrySet()) {
   Query query = entry.getKey();
   int limit = entry.getValue().intValue();
   Weight weight = query.weight(searcher);
   Scorer scorer = weight.scorer(reader, true, false);
   if (scorer != null) {
    while(true)   {
     int doc = scorer.nextDoc();
     if (((long) docIDStart) + doc >= limit)
      break;
     reader.deleteDocument(doc);
     any = true;
    }
```

```
    }
  }
  searcher.close();
  return any;
}
```

第五章：Lucene 段合并(merge)过程分析

一、段合并过程总论

IndexWriter 中与段合并有关的成员变量有：

- HashSet<SegmentInfo> mergingSegments = new HashSet<SegmentInfo>(); //保存正在合并的段，以防止合并期间再次选中被合并。
- MergePolicy mergePolicy = new LogByteSizeMergePolicy(this);//合并策略，也即选取哪些段来进行合并。
- MergeScheduler mergeScheduler = new ConcurrentMergeScheduler();//段合并器，背后有一个线程负责合并。
- LinkedList<MergePolicy.OneMerge> pendingMerges = new LinkedList<MergePolicy.OneMerge>();//等待被合并的任务
- Set<MergePolicy.OneMerge> runningMerges = new HashSet<MergePolicy.OneMerge>();//正在被合并的任务

和段合并有关的一些参数有：

- mergeFactor：当大小几乎相当的段的数量达到此值的时候，开始合并。
- minMergeSize：所有大小小于此值的段，都被认为是大小几乎相当，一同参与合并。
- maxMergeSize：当一个段的大小大于此值的时候，就不再参与合并。
- maxMergeDocs：当一个段包含的文档数大于此值的时候，就不再参与合并。

段合并一般发生在添加完一篇文档的时候，当一篇文档添加完后，发现内存已经达到用户设定的 ramBufferSize，则写入文件系统，形成一个新的段。新段的加入可能造成差不多大小的段的个数达到 mergeFactor，从而开始了合并的过程。

合并过程最重要的是两部分：

- 一个是选择哪些段应该参与合并，这一步由 MergePolicy 来决定。
- 一个是将选择出的段合并成新段的过程，这一步由 MergeScheduler 来执行。段的合并也主要包括：
  - 对正向信息的合并，如存储域，词向量，标准化因子等。
  - 对反向信息的合并，如词典，倒排表。

在总论中，我们重点描述合并策略对段的选择以及反向信息的合并。

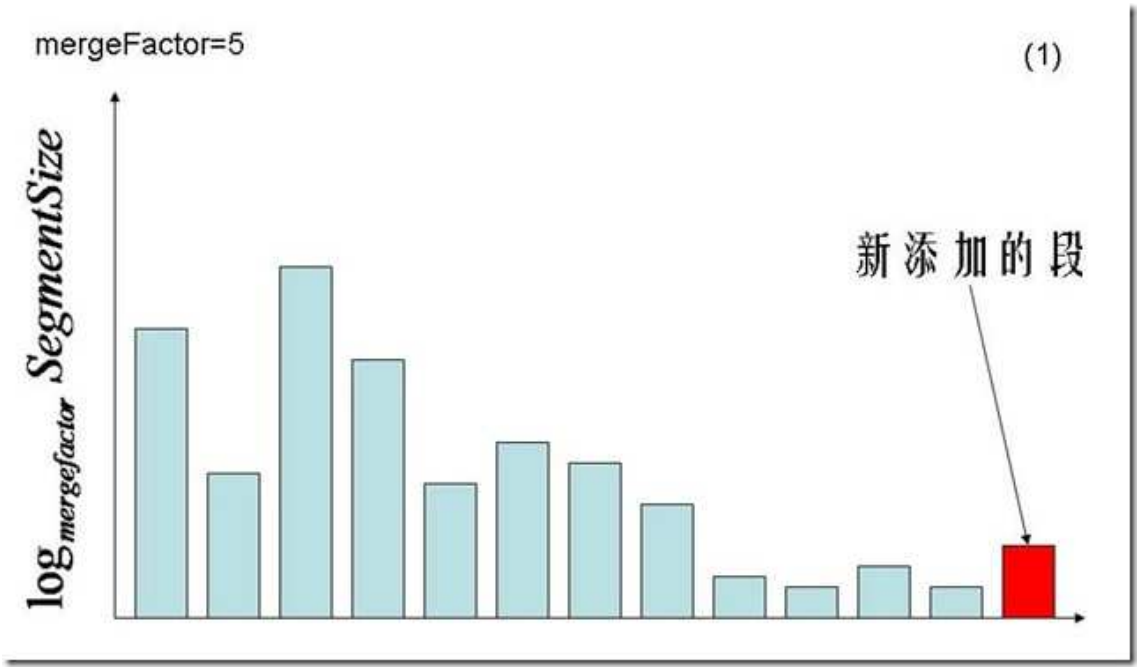
1.1、合并策略对段的选择

在 LogMergePolicy 中，选择可以合并的段的基本逻辑是这样的：

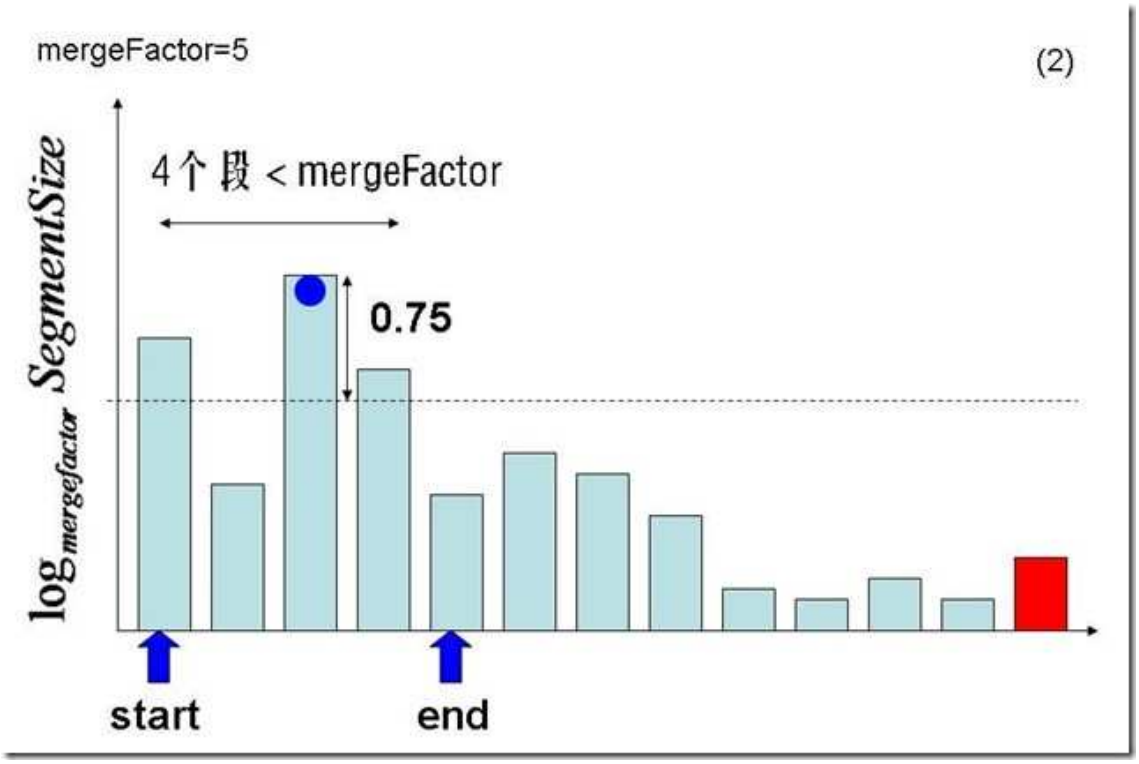
- 选择的可以合并的段都是在硬盘上的，不再存在内存中的段，也不是像早期的版本一样每添加一个 Document 就生成一个段，然后进行内存中的段合并，然后再合并到硬盘中。
- 由于从内存中 flush 到硬盘上是按照设置的内存大小来 DocumentsWriter.ramBufferSize 触发的，所以每个刚 flush 到硬盘上的段大小差不多，当然不排除中途改变内存设置，接下来的算法可以解决这个问题。
- 合并的过程是尽量按照合并几乎相同大小的段这一原则，只有大小相当的 mergeFacetor 个段出现的时候，才合并成一个新的段。
- 在硬盘上的段基本应该是大段在前，小段在后，因为大段总是由小段合并而成的，当小段凑够 mergeFactor 个的时候，就合并成一个大段，小段就被删除了，然后新来的一定是新的小段。
- 比如 mergeFactor=3，开始来的段大小为 10M，当凑够 3 个 10M 的时候，0.cfs, 1.cfs, 2.cfs 则合并成一个新的段 3.cfs，大小为 30M，然后再来 4.cfs, 5.cfs, 6.cfs，合并成 7.cfs，大小为 30M，然后再来 8.cfs, 9.cfs, a.cfs 合并成 b.cfs, 大小为 30M，这时候又凑够了 3 个 30M 的，合并成 90M 的 c.cfs，然后又来 d.cfs, e.cfs, f.cfs 合并成 10.cfs，大小为 30M，然后 11.cfs 大小为 10M，这时候硬盘上的段为：c.cfs(90M) 10.cfs(30M),11.cfs(10M)。

所以 LogMergePolicy 对合并段的选择过程如下：

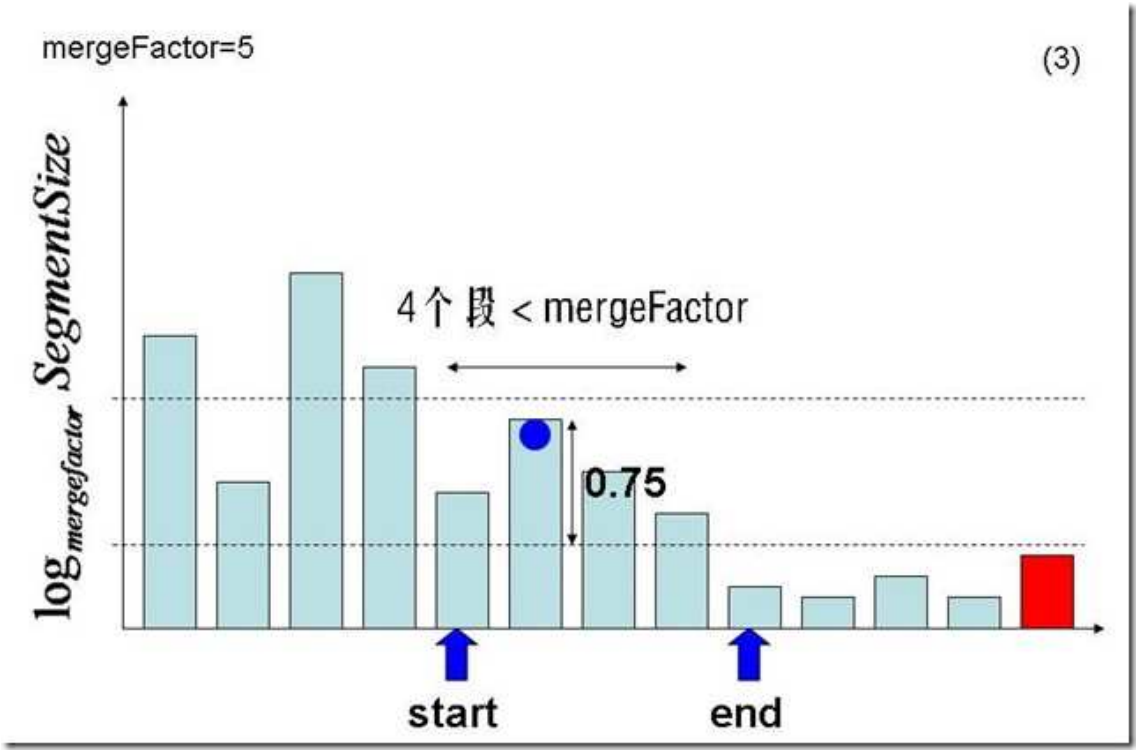
- 将所有的段按照生成的顺序，将段的大小以 mergeFactor 为底取对数，放入数组中，作为选择的标准。



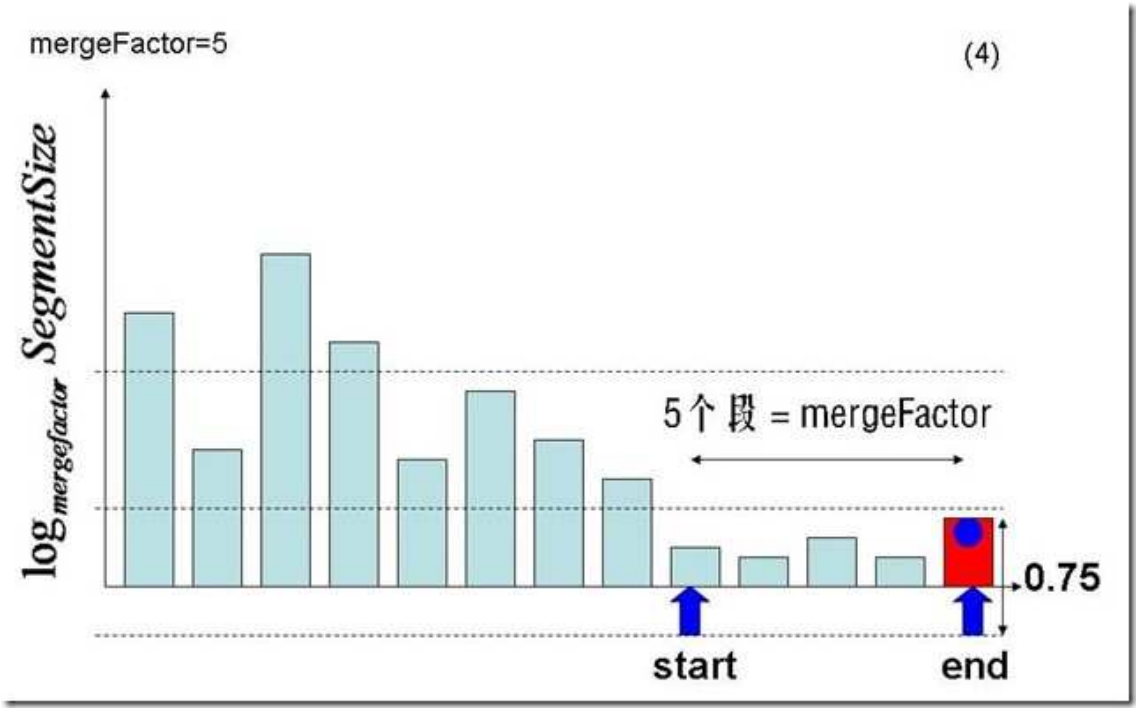
- 从头开始，选择一个值最大的段，然后将此段的值减去 0.75(LEVEL_LOG_SPAN) ，之间的段被认为是大小差不多的段，属于同一阶梯，此处称为第一阶梯。
- 然后从后向前寻找第一个属于第一阶梯的段，从 start 到此段之间的段都被认为是属于这一阶梯的。也包括之间生成较早但大小较小的段，因为考虑到以下几点：
  - 防止较早生成的段由于人工 flush 或者人工调整 ramBufferSize，因而很小，却破坏了基本从大到小的规则。
  - 如果运行较长时间后，致使段的大小参差不齐，很难合并相同大小的段。
  - 也防止一个段由于较小，而不断的都有大的段生成从而始终不能参与合并。
- 第一阶梯总共 4 个段，小于 mergeFactor 因而不合并，接着 start=end 从而选择下一阶梯。



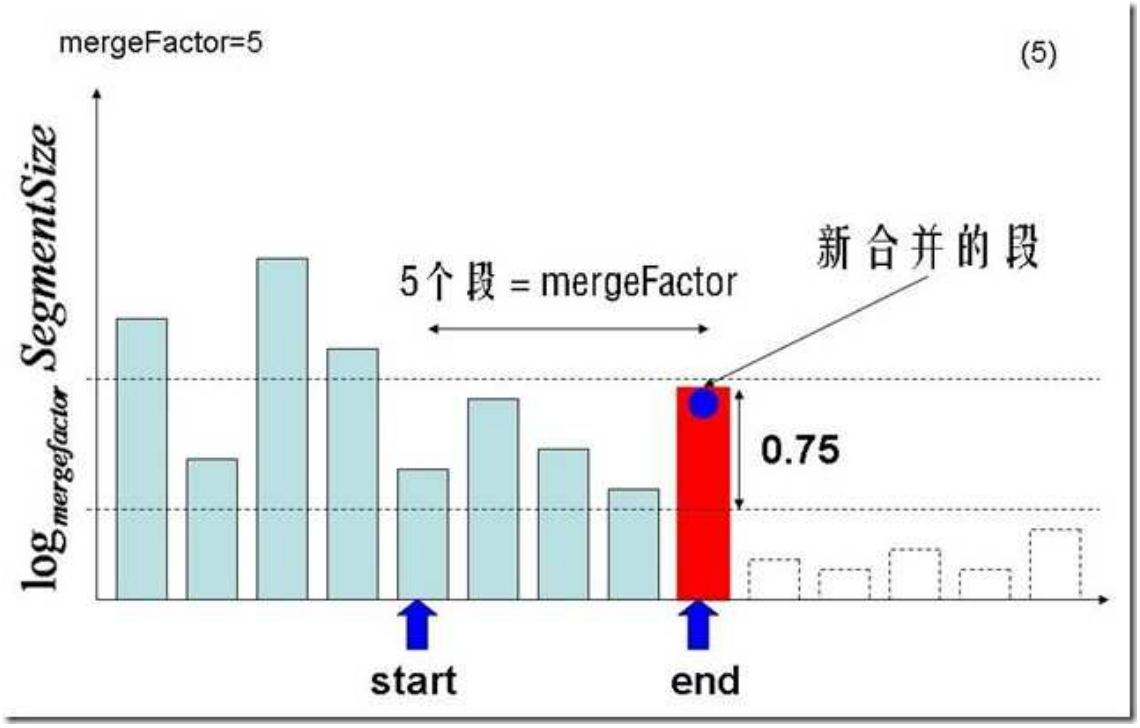
- 从 start 开始，选择一个值最大的段，然后将此段的值减去 0.75(LEVEL_LOG_SPAN) ，之间的段被认为属于同一阶梯，此处称为第二阶梯。
- 然后从后向前寻找第一个属于第二阶梯的段，从 start 到此段之间的段都被认为是属于这一阶梯的。
- 第二阶梯总共 4 个段，小于 mergeFactor 因而不合并，接着 start=end 从而选择下一阶梯。



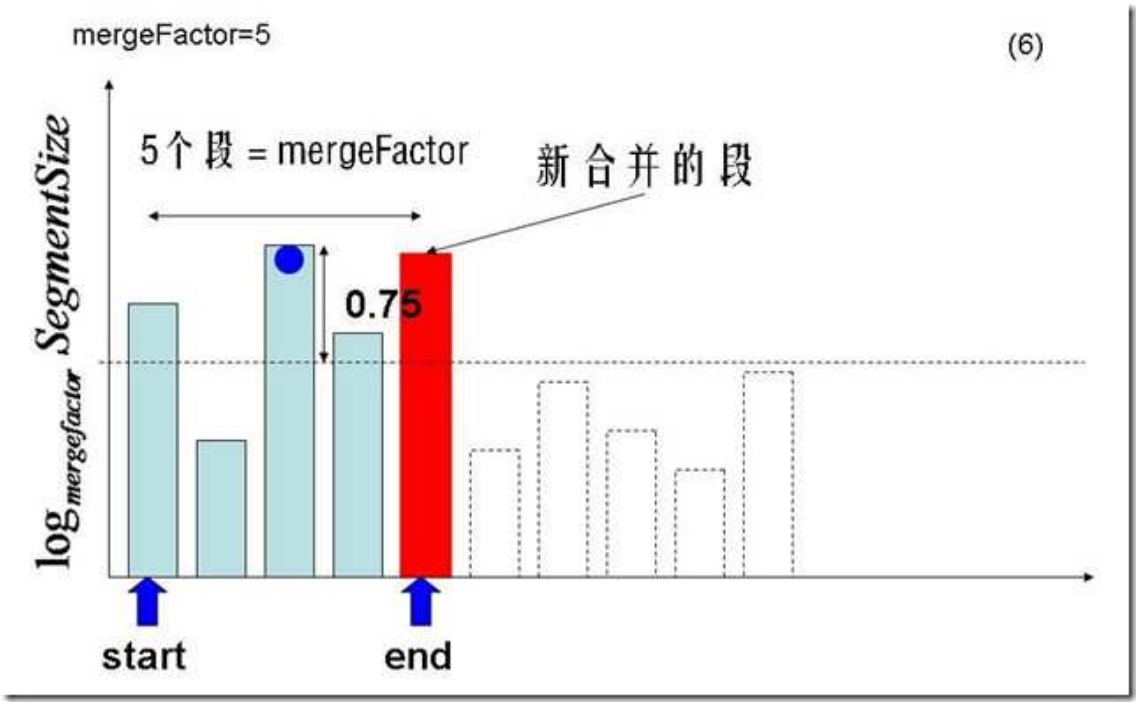
- 从 start 开始，选择一个值最大的段，然后将此段的值减去 0.75(LEVEL_LOG_SPAN) ，之间的段被认为属于同一阶梯，此处称为第三阶梯。
- 由于最大的段减去 0.75 后为负的，因而从 start 到此段之间的段都被认为是属于这一阶梯的。
- 第三阶梯总共 5 个段，等于 mergeFactor，因而进行合并。



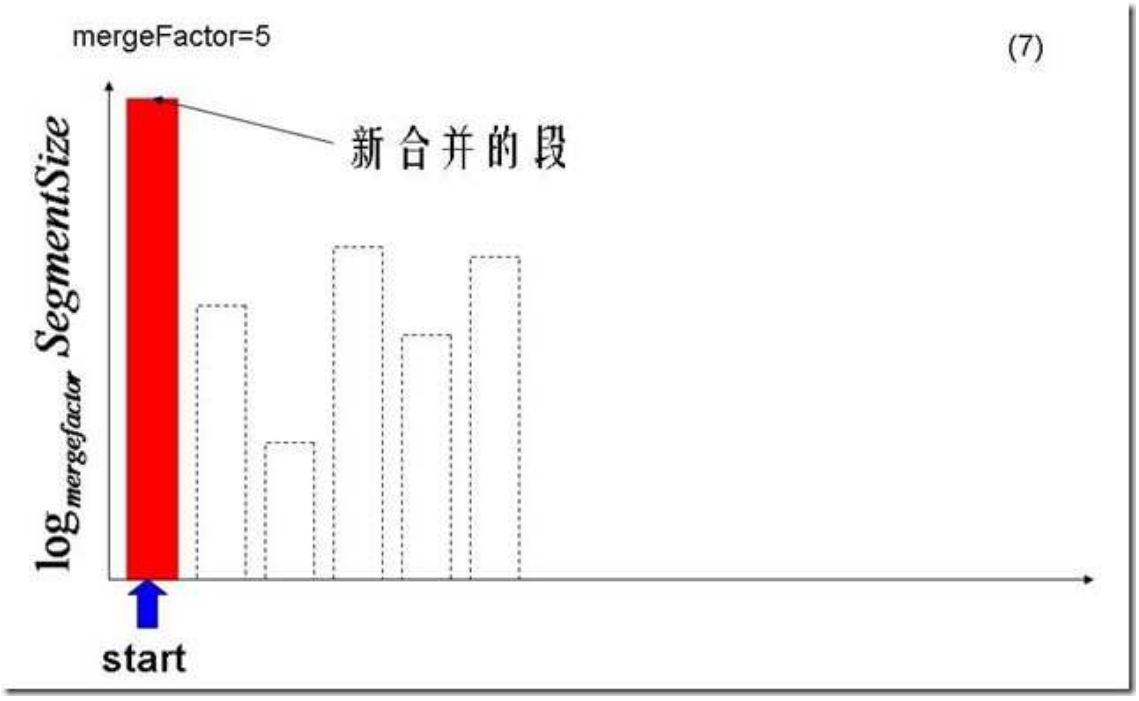
- 第三阶梯的五个段合并成一个较大的段。
- 然后从头开始，依然先考察第一阶梯，仍然是 4 个段，不合并。
- 然后是第二阶梯，因为有了新生成的段，并且大小足够属于第二阶梯，从而第二阶梯有 5 个段，可以合并。



- 第二阶段的五个段合并成一个较大的段。
- 然后从头开始，考察第一阶梯，因为有了新生成的段，并且大小足够属于第一阶梯，从而第一阶梯有 5 个段，可以合并。



- 第一阶梯的五个段合并成一个大的段。



1.2、反向信息的合并

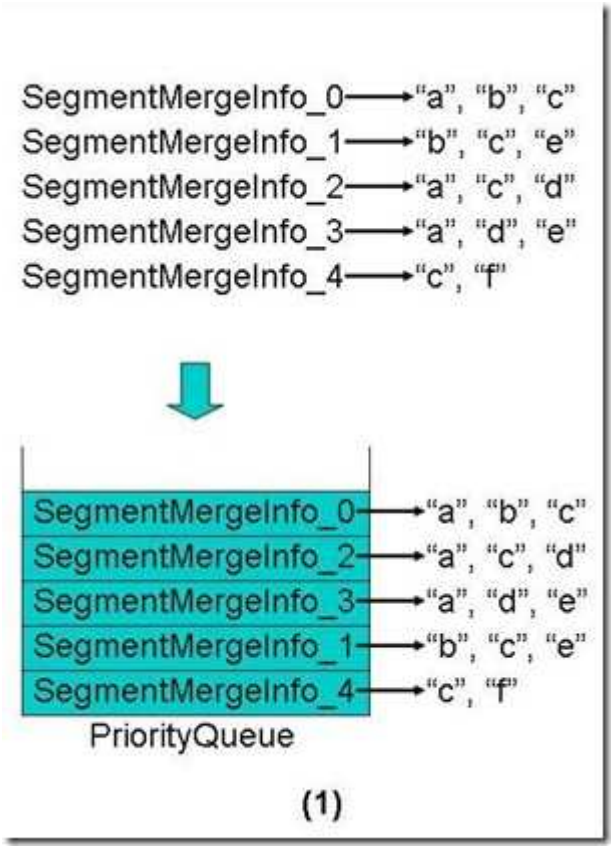
反向信息的合并包括两部分：

- 对字典的合并，词典中的 Term 是按照字典顺序排序的，需要对词典中的 Term 进行重新排序
- 对于相同的 Term，对包含此 Term 的文档号列表进行合并，需要对文档号重新编号。

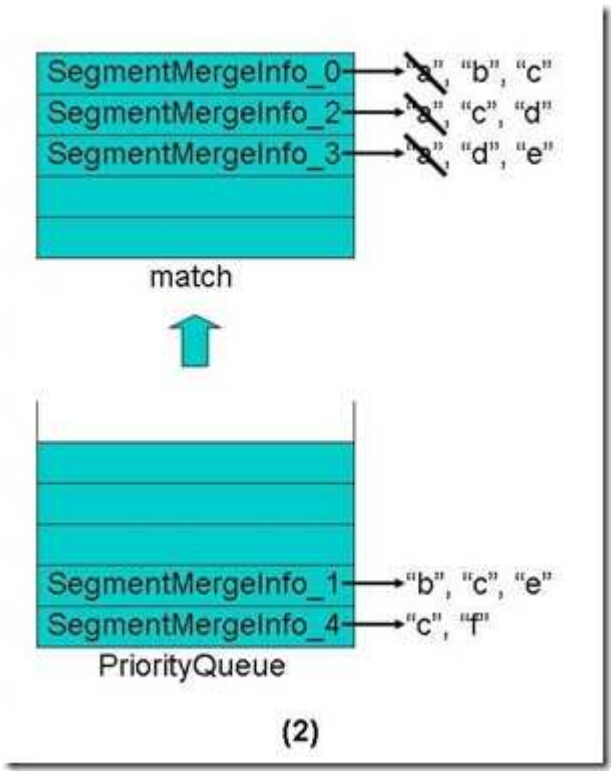
对词典的合并需要找出两个段中相同的词，Lucene 是通过一个称为 match 的 SegmentMergeInfo 类型的数组以及称为 queue 的 SegmentMergeQueue 实现的，SegmentMergeQueue 是继承于 PriorityQueue<SegmentMergeInfo>，是一个优先级队列，是按照字典顺序排序的。SegmentMergeInfo 保存要合并的段的词典及倒排表信息，在 SegmentMergeQueue 中用来排序的 key 是它代表的段中的第一个 Term。

我们来举一个例子来说明合并词典的过程，以便后面解析代码的时候能够很好的理解：

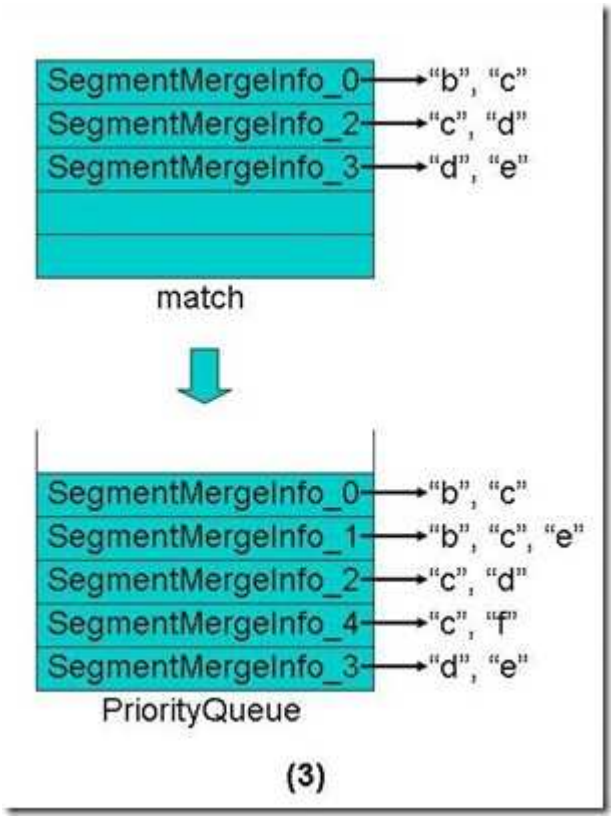
- 假设要合并五个段，每个段包含的 Term 也是按照字典顺序排序的，如下图所示。
- 首先把五个段全部放入优先级队列中，段在其中也是按照第一个 Term 的字典顺序排序的，如下图。



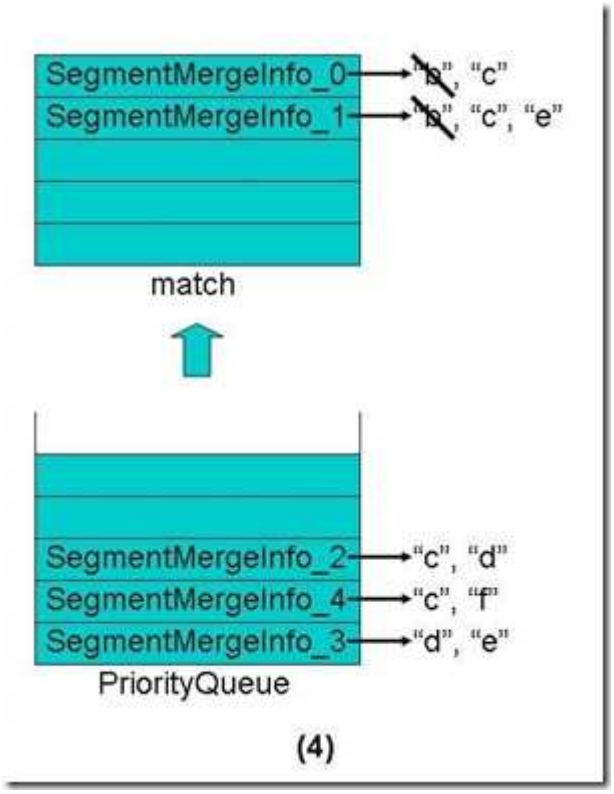
- 从优先级队列中弹出第一个 Term("a")相同的段到 match 数组中，如下图。
- 合并这些段的第一个 Term("a")的倒排表，并把此 Term 和它的倒排表一同加入新生成的段中。
- 对于 match 数组中的每个段取下一个 Term



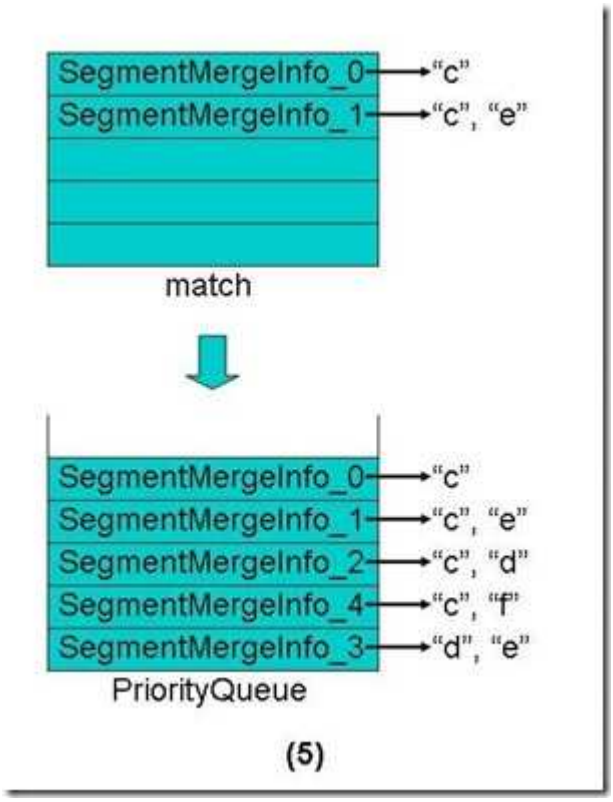
- 将 match 数组中还有 Term 的段重新放入优先级队列中，这些段也是按照第一个 Term 的字典顺序排序。



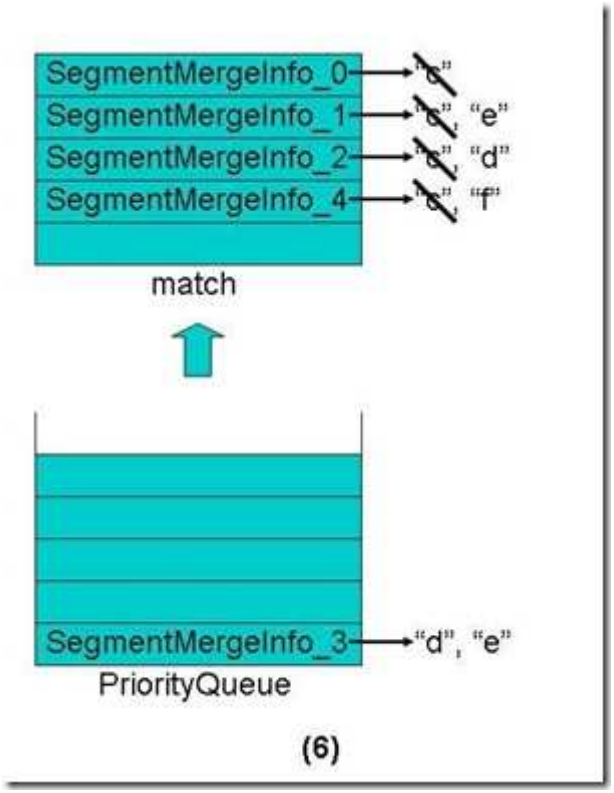
- 从优先级队列中弹出第一个 Term("b")相同的段到 match 数组中。
- 合并这些段的第一个 Term("b")的倒排表，并把此 Term 和它的倒排表一同加入新生成的段中。
- 对于 match 数组中的每个段取下一个 Term



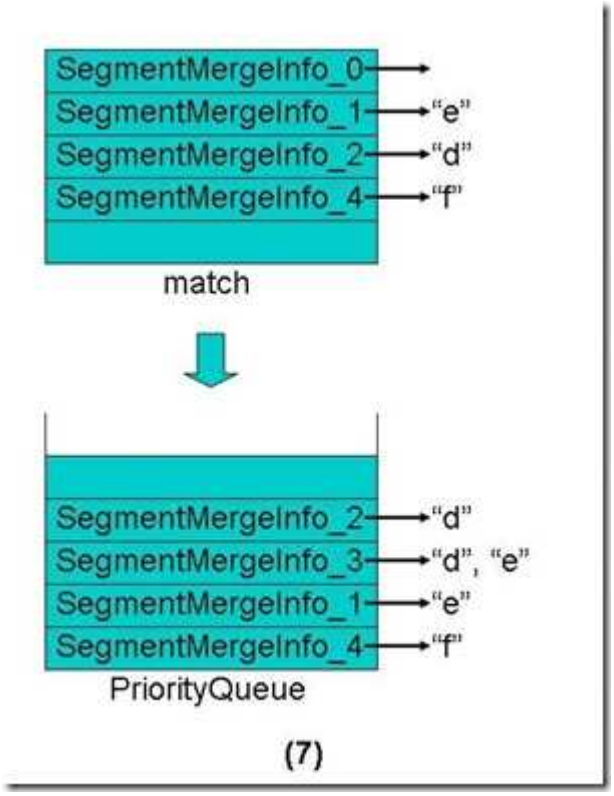
- 将 match 数组中还有 Term 的段重新放入优先级队列中，这些段也是按照第一个 Term 的字典顺序排序。



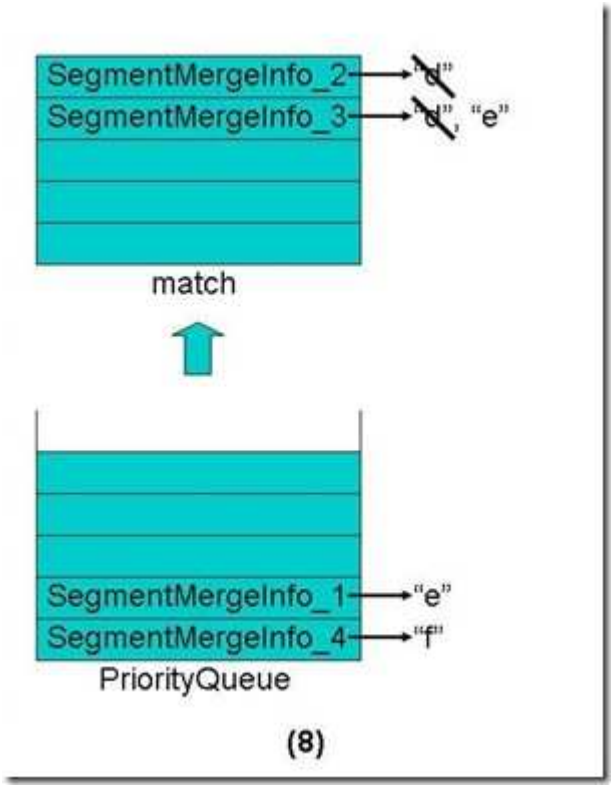
- 从优先级队列中弹出第一个 Term("c")相同的段到 match 数组中。
- 合并这些段的第一个 Term("c")的倒排表，并把此 Term 和它的倒排表一同加入新生成的段中。
- 对于 match 数组中的每个段取下一个 Term



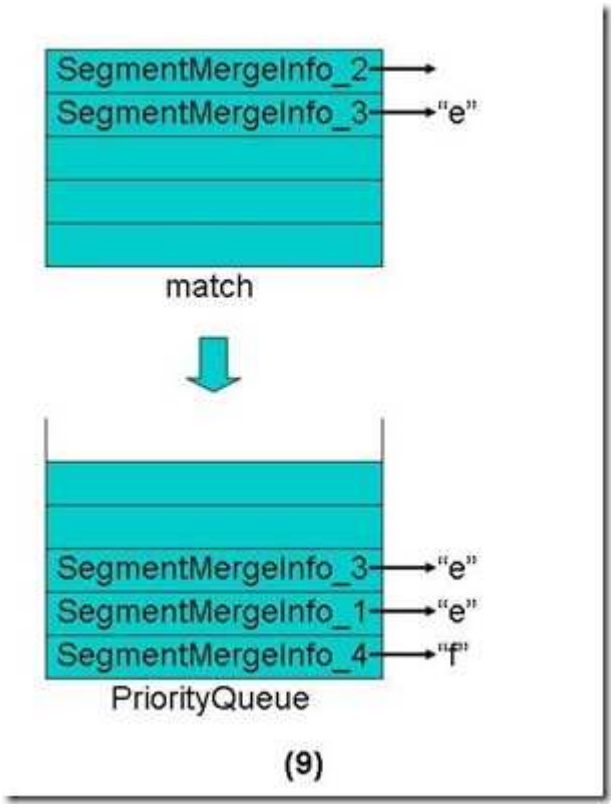
- 将 match 数组中还有 Term 的段重新放入优先级队列中，这些段也是按照第一个 Term 的字典顺序排序。



- 从优先级队列中弹出第一个 Term("d")相同的段到 match 数组中。
- 合并这些段的第一个 Term("d")的倒排表，并把此 Term 和它的倒排表一同加入新生成的段中。
- 对于 match 数组中的每个段取下一个 Term

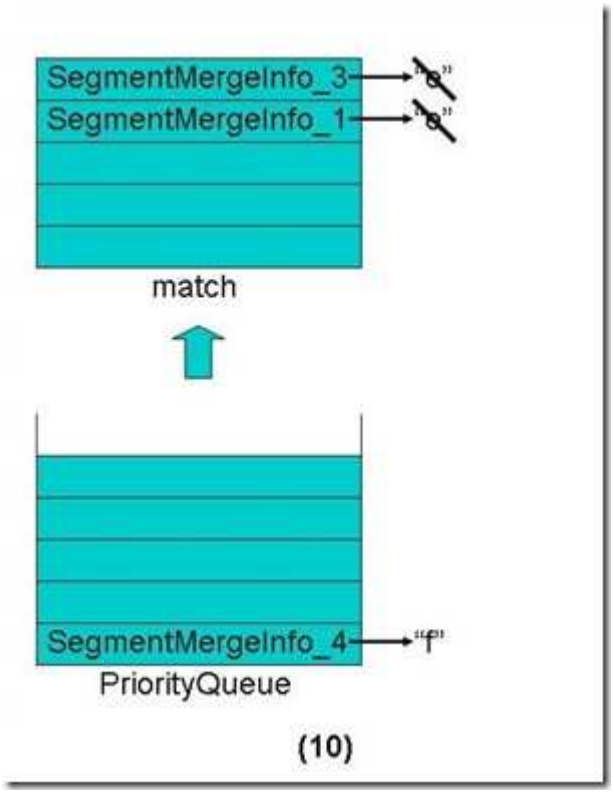


- 将 match 数组中还有 Term 的段重新放入优先级队列中，这些段也是按照第一个 Term 的字典顺序排序。

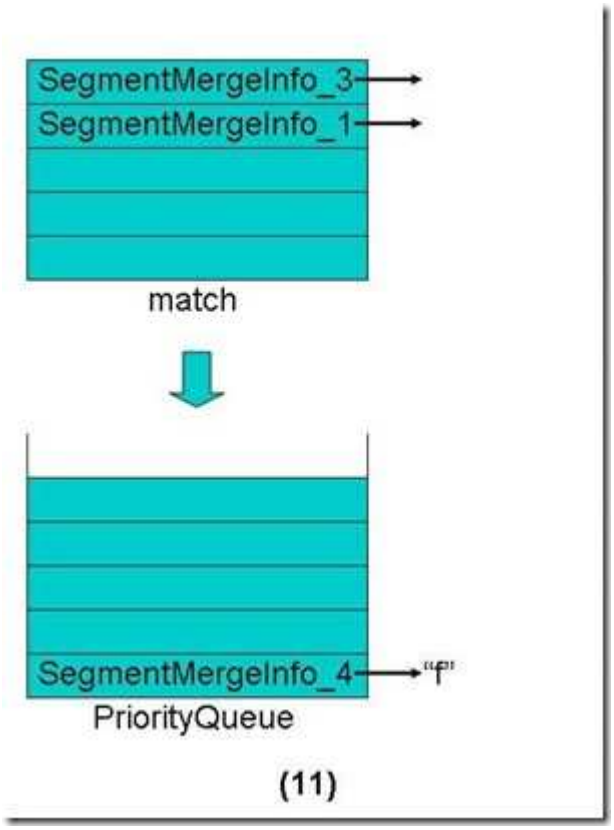


- 从优先级队列中弹出第一个 Term("e")相同的段到 match 数组中。

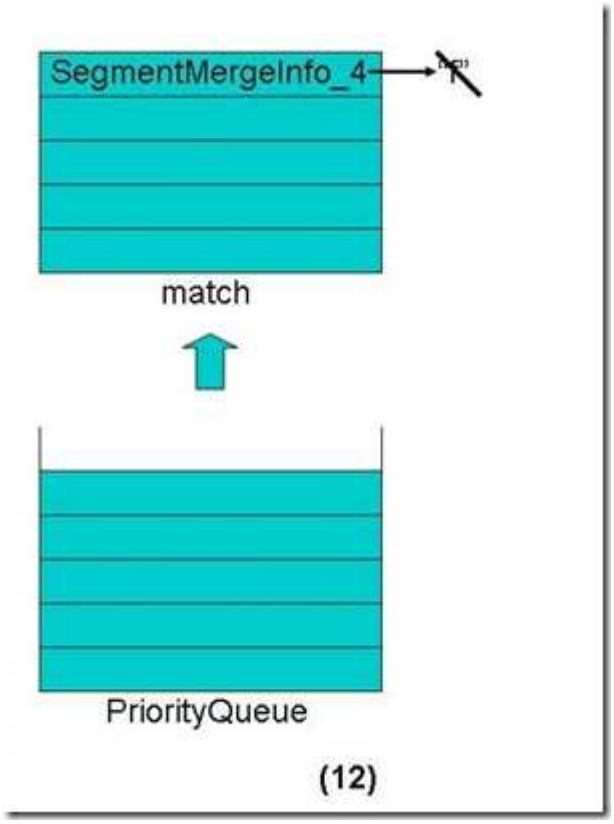
- 合并这些段的第一个 Term("e")的倒排表，并把此 Term 和它的倒排表一同加入新生成的段中。
- 对于 match 数组中的每个段取下一个 Term



- 将 match 数组中还有 Term 的段重新放入优先级队列中，这些段也是按照第一个 Term 的字典顺序排序。



- 从优先级队列中弹出第一个 Term("f")相同的段到 match 数组中。
- 合并这些段的第一个 Term("f")的倒排表，并把此 Term 和它的倒排表一同加入新生成的段中。
- 对于 match 数组中的每个段取下一个 Term



- 合并完毕。

二、段合并的详细过程

2.1、将缓存写入新的段

IndexWriter 在添加文档的时候调用函数 addDocument(Document doc, Analyzer analyzer)，包含如下步骤：

- doFlush = docWriter.addDocument(doc, analyzer);//DocumentsWriter 添加文档，最后返回是否进行向硬盘写入
  - return state.doFlushAfter || timeToFlushDeletes();//这取决于 timeToFlushDeletes

timeToFlushDeletes 返回 return (bufferIsFull || deletesFull()) && setFlushPending()，而在 Lucene 索引过程分析(2)的 DocumentsWriter 的缓存管理部分提到，当

numBytesUsed+deletesRAMUsed > ramBufferSize 的时候 bufferIsFull 设为 true，也即当使用的内存大于 ramBufferSize 的时候，则由内存向硬盘写入。ramBufferSize 可以用 IndexWriter.setRAMBufferSizeMB(double mb)设定。

- if (doFlush) flush(true, false, false);//如果内存中缓存满了，则写入硬盘
  - if (doFlush(flushDocStores, flushDeletes) && triggerMerge) maybeMerge();//doFlush 将缓存写入硬盘，此过程在 Lucene 索引过程分析(4)中关闭 IndexWriter 一节已经描述。

当缓存写入硬盘，形成了新的段后，就有可能触发一次段合并，所以调用 maybeMerge()

```
IndexWriter.maybeMerge()
--> maybeMerge(false);
    --> maybeMerge(1, optimize);
        --> updatePendingMerges(maxNumSegmentsOptimize, optimize);
        --> mergeScheduler.merge(this);
```

IndexWriter.updatePendingMerges(int maxNumSegmentsOptimize, boolean optimize)主要负责找到可以合并的段，并生产段合并任务对象，并向段合并器注册这个任务。

ConcurrentMergeScheduler.merge(IndexWriter)主要负责进行段的合并。

2.2、选择合并段，生成合并任务

IndexWriter.updatePendingMerges(int maxNumSegmentsOptimize, boolean optimize)主要包括两部分：

- 选择能够合并段：MergePolicy.MergeSpecification spec = mergePolicy.findMerges(segmentInfos);
- 向段合并器注册合并任务，将任务加到 pendingMerges 中：
  - for(int i=0;i<spec.merges.size();i++)
    - registerMerge(spec.merges.get(i));

2.2.1、用合并策略选择合并段

默认的段合并策略是 LogByteSizeMergePolicy，其选择合并段由 LogMergePolicy.findMerges(SegmentInfos infos) 完成，包含以下过程：

(1) 生成 **levels** 数组，每个段一项。然后根据每个段的大小，计算每个项的值，**levels[i]**和段的大小的关系为 **Math.log(size)/Math.log(mergeFactor)**，代码如下：

```
final int numSegments = infos.size();
float[] levels = new float[numSegments];
final float norm = (float) Math.log(mergeFactor);
for(int i=0;i<numSegments;i++) {
  final SegmentInfo info = infos.info(i);
  long size = size(info);
  levels[i] = (float) Math.log(size)/norm;
}
```

(2) 由于段基本是按照由大到小排列的，而且合并段应该大小差不多的段中进行。我们把大小差不多的段称为属于同一阶梯，因而此处从第一个段开始找属于相同阶梯的段，如果属于此阶梯的段数量达到 **mergeFactor** 个，则生成合并任务，否则继续向后寻找下一阶梯。

```
//计算最低阶梯值，所有小于此值的都属于最低阶梯
final float levelFloor = (float) (Math.log(minMergeSize)/norm);
MergeSpecification spec = null;
int start = 0;
while(start < numSegments) {
 //找到 levels 数组的最大值，也即当前阶梯中的峰值
 float maxLevel = levels[start];
 for(int i=1+start;i<numSegments;i++) {
  final float level = levels[i];
  if (level > maxLevel)
    maxLevel = level;
 }
 //计算出此阶梯的谷值，也即最大值减去 0.75，之间的都属于此阶梯。如果峰值小于最低阶梯值，则所有此阶梯的段都属于最低阶梯。如果峰值大于最低阶梯值，谷值小于最低阶梯值，则设置谷值为最低阶梯值，以保证所有小于最低阶梯值的段都属于最低阶梯。
```

```
    float levelBottom;
    if (maxLevel < levelFloor)
      levelBottom = -1.0F;
    else {
      levelBottom = (float) (maxLevel - LEVEL_LOG_SPAN);
      if (levelBottom < levelFloor && maxLevel >= levelFloor)
        levelBottom = levelFloor;
    }
    float levelBottom = (float) (maxLevel - LEVEL_LOG_SPAN);
```

//从最后一个段向左找，当然段越来越大，找到第一个大于此阶梯的谷值的段，从 start 的段开始，一直到 upto 这个段，都属于此阶梯了。尽管 upto 左面也有的段由于内存设置原因，虽形成较早，但是没有足够大，也作为可合并的一员考虑在内了，将被并入一个大的段，从而保证了基本上左大右小的关系。从 upto 这个段向右都是比此阶梯小的多的段，应该属于下一阶梯。

```
    int upto = numSegments-1;
    while(upto >= start) {
      if (levels[upto] >= levelBottom) {
        break;
      }
      upto--;
    }
```

//从 start 段开始，数 mergeFactor 个段，如果不超过 upto 段，说明此阶梯已经足够 mergeFactor 个了，可以合并了。当然如果此阶梯包含太多要合并的段，也是每 mergeFactor 个段进行一次合并，然后再依次数 mergeFactor 段进行合并，直到此阶梯的段合并完毕。

```
    int end = start + mergeFactor;
    while(end <= 1+upto) {
      boolean anyTooLarge = false;
      for(int i=start;i<end;i++) {
        final SegmentInfo info = infos.info(i);
```

```
    //如果一个段的大小超过 maxMergeSize 或者一个段包含的文档数量超过 maxMergeDocs 则不再合并。
      anyTooLarge |= (size(info) >= maxMergeSize || sizeDocs(info) >= maxMergeDocs);
    }
    if (!anyTooLarge) {
      if (spec == null)
        spec = new MergeSpecification();
      //如果确认要合并，则从 start 到 end 生成一个段合并任务 OneMerge.
      spec.add(new OneMerge(infos.range(start, end), useCompoundFile));
    }
   //刚刚合并的是从 start 到 end 共 mergeFactor 和段，此阶梯还有更多的段，则再依次数 mergeFactor 个段。
    start = end;
    end = start + mergeFactor;
   }
   //从 start 到 upto 是此阶梯的所有的段，已经选择完毕，下面选择更小的下一个阶梯的段
   start = 1+upto;
  }
```

选择的结果保存在 MergeSpecification 中，结构如下：

```
spec    MergePolicy$MergeSpecification   (id=25)
   merges    ArrayList<E>   (id=28)
     elementData    Object[10]   (id=39)
       [0]    MergePolicy$OneMerge   (id=42)
         aborted    false
         error    null
         increfDone    false
         info    null
         isExternal    false
```

```
maxNumSegmentsOptimize  0
mergeDocStores  false
mergeGen  0
optimize  false
readers  null
readersClone  null
registerDone  false
segments  SegmentInfos  (id=50)
  capacityIncrement  0
  counter  0
  elementCount  3
  elementData  Object[10]  (id=54)
    [0]  SegmentInfo  (id=62)
      delCount  0
      delGen  -1
      diagnostics  HashMap<K,V>  (id=67)
      dir  SimpleFSDirectory  (id=69)
      docCount  1062
      docStoreIsCompoundFile  false
      docStoreOffset  0
      docStoreSegment  "_0"
      files  ArrayList<E>  (id=73)
      hasProx  true
      hasSingleNormFile  true
      isCompoundFile  1
      name  "_0"
      normGen  null
      preLockless  false
```

```
    sizeInBytes    15336467
  [1]    SegmentInfo   (id=64)
    delCount    0
    delGen    -1
    diagnostics    HashMap<K,V>   (id=79)
    dir    SimpleFSDirectory   (id=69)
    docCount    1068
    docStoreIsCompoundFile    false
    docStoreOffset    1062
    docStoreSegment    "_0"
    files    ArrayList<E>   (id=80)
    hasProx    true
    hasSingleNormFile    true
    isCompoundFile    1
    name    "_1"
    normGen    null
    preLockless    false
    sizeInBytes    15420953
  [2]    SegmentInfo   (id=65)
    delCount    0
    delGen    -1
    diagnostics    HashMap<K,V>   (id=86)
    dir    SimpleFSDirectory   (id=69)
    docCount    1068
    docStoreIsCompoundFile    false
    docStoreOffset    2130
    docStoreSegment    "_0"
    files    ArrayList<E>   (id=88)
```

```
                hasProx    true
                hasSingleNormFile    true
                isCompoundFile    1
                name    "_2"
                normGen    null
                preLockless    false
                sizeInBytes    15420953
            generation    0
            lastGeneration    0
            modCount    1
            pendingSegnOutput    null
            userData    Collections$EmptyMap   (id=57)
            version    1267460515437
          useCompoundFile    true
      modCount    1
      size    1
```

2.2.2、注册段合并任务

注册段合并任务由 IndexWriter.registerMerge(MergePolicy.OneMerge merge)完成：

（1）如果选择出的段正在被合并，或者不存在，则退出。

```
final int count = merge.segments.size();
boolean isExternal = false;
for(int i=0;i<count;i++) {
  final SegmentInfo info = merge.segments.info(i);
  if (mergingSegments.contains(info))
    return false;
  if (segmentInfos.indexOf(info) == -1)
```

```
    return false;
  if (info.dir != directory)
    isExternal = true;
}
```

(2) 将合并任务加入 pendingMerges：pendingMerges.add(merge);

(3) 将要合并的段放入 mergingSegments 以防正在合并又被选为合并段。

```
for(int i=0;i<count;i++)
  mergingSegments.add(merge.segments.info(i));
```

2.3、段合并器进行段合并

段合并器默认为 ConcurrentMergeScheduler，段的合并工作由 ConcurrentMergeScheduler.merge(IndexWriter) 完成，它包含 while(true)的循环，在循环中不断做以下事情：

- 得到下一个合并任务：MergePolicy.OneMerge merge = writer.getNextMerge();
- 初始化合并任务：writer.mergeInit(merge);
  - 将删除文档写入硬盘：applyDeletes();
  - 是否合并存储域：mergeDocStores = false。按照 Lucene 的索引文件格式(2)中段的元数据信息(segments_N)中提到的，IndexWriter.flush(boolean triggerMerge, boolean flushDocStores, boolean flushDeletes)中第二个参数 flushDocStores 会影响到是否单独或是共享存储。其实最终影响的是 DocumentsWriter.closeDocStore()。每当 flushDocStores 为 false 时，closeDocStore 不被调用，说明下次添加到索引文件中的域和词向量信息是同此次共享一个段的。直到 flushDocStores 为 true 的时候，closeDocStore 被调用，从而下次添加到索引文件中的域和词向量信息将被保存在一个新的段中，不同此次共享一个段。如 2.1 节中说的那样，在 addDocument 中，如果内存中缓存满了，则写入硬盘，调用的是 flush(true, false, false)，也即所有的存储域都存储在共享的域中(_0.fdt)，因而不需要合并存储域。
  - 生成新的段：merge.info = new SegmentInfo(newSegmentName(),…)
  - 将新的段加入 mergingSegments
- 如果已经有足够多的段合并线程，则等待 while (mergeThreadCount() >= maxThreadCount) wait();
- 生成新的段合并线程：
  - merger = getMergeThread(writer, merge);
  - mergeThreads.add(merger);
- 启动段合并线程：merger.start();

段合并线程的类型为 MergeThread，MergeThread.run()包含 while(truy)循环，在循环中做以下事情：

- 合并当前的任务：doMerge(merge);
- 得到下一个段合并任务：merge = writer.getNextMerge();

ConcurrentMergeScheduler.doMerge(OneMerge) 最终调用 IndexWriter.merge(OneMerge) ，主要做以下事情：

- 初始化合并任务：mergeInit(merge);
- 进行合并：mergeMiddle(merge);
- 完成合并任务：mergeFinish(merge);
  - 从 mergingSegments 中移除被合并的段和合并新生成的段：
    - for(int i=0;i<end;i++)
      mergingSegments.remove(sourceSegments.info(i));
    - mergingSegments.remove(merge.info);
  - 从 runningMerges 中移除此合并任务：runningMerges.remove(merge);

IndexWriter.mergeMiddle(OneMerge)主要做以下几件事情：

- 生成用于合并段的对象 SegmentMerger merger = new SegmentMerger(this, mergedName, merge);
- 打开 Reader 指向要合并的段：

```
merge.readers = new SegmentReader[numSegments];
merge.readersClone = new SegmentReader[numSegments];
for (int i = 0; i < numSegments; i++) {
    final SegmentInfo info = sourceSegments.info(i);
   // Hold onto the "live" reader; we will use this to
   // commit merged deletes
   SegmentReader reader = merge.readers[i] = readerPool.get(info,
merge.mergeDocStores,MERGE_READ_BUFFER_SIZE,-1);
    // We clone the segment readers because other
    // deletes may come in while we're merging so we
    // need readers that will not change
```

```
    SegmentReader clone = merge.readersClone[i] = (SegmentReader) reader.clone(true);
    merger.add(clone);
}
```

- 进行段合并：mergedDocCount = merge.info.docCount = merger.merge(merge.mergeDocStores);
- 合并生成的段生成为 cfs：merger.createCompoundFile(compoundFileName);

SegmentMerger.merge(boolean) 包含以下几部分：

- 合并域：mergeFields()
- 合并词典和倒排表：mergeTerms();
- 合并标准化因子：mergeNorms();
- 合并词向量：mergeVectors();

下面依次分析者几部分。

2.3.1、合并存储域

合并存储域主要包含两部分：一部分是合并 fnm 信息，也即域元数据信息，一部分是合并 fdt,fdx 信息，也即域数据信息。

(1) 合并 fnm 信息

- 首先生成新的域元数据信息：fieldInfos = new FieldInfos();
- 依次用 reader 读取每个合并段的域元数据信息，加入上述对象

```
for (IndexReader reader : readers) {
    SegmentReader segmentReader = (SegmentReader) reader;
    FieldInfos readerFieldInfos = segmentReader.fieldInfos();
    int numReaderFieldInfos = readerFieldInfos.size();
    for (int j = 0; j < numReaderFieldInfos; j++) {
      FieldInfo fi = readerFieldInfos.fieldInfo(j);
    //在通常情况下，所有的段中的文档都包含相同的域，比如添加文档的时候，每篇文档都包含"title"，"description"，"author"，"time"等，不会为某一篇文档添加或减少与其他文档不同的域。但也不排除特殊情况下有特殊的文档有特殊的域。因而此处的 add 是无则添加，有则更新。
```

```
    fieldInfos.add(fi.name, fi.isIndexed, fi.storeTermVector,
      fi.storePositionWithTermVector, fi.storeOffsetWithTermVector,
      !reader.hasNorms(fi.name), fi.storePayloads,
      fi.omitTermFreqAndPositions);
   }
}
```

- 将域元数据信息 fnm 写入文件：fieldInfos.write(directory, segment + ".fnm");

(2) 合并段数据信息 fdt, fdx

在合并段的数据信息的时候，有两种情况：

- 情况一：通常情况，要合并的段和新生成段包含的域的名称，顺序都是一样的，这样就可以把要合并的段的 fdt 信息直接拷贝到新生成段的最后，以提高合并效率。
- 情况二：要合并的段包含特殊的文档，其包含的域多于或者少于新生成段的域，这样就不能够直接拷贝，而是一篇文档一篇文档的添加。这样合并效率大大降低，因而不鼓励添加文档的时候，不同的文档使用不同的域。

具体过程如下：

- 首先检查要合并的各个段，其包含域的名称，顺序是否同新生成段的一致，也即是否属于第一种情况：setMatchingSegmentReaders();

```
private void setMatchingSegmentReaders() {
  int numReaders = readers.size();
  matchingSegmentReaders = new SegmentReader[numReaders];
  //遍历所有的要合并的段
  for (int i = 0; i < numReaders; i++) {
    IndexReader reader = readers.get(i);
    if (reader instanceof SegmentReader) {
      SegmentReader segmentReader = (SegmentReader) reader;
      boolean same = true;
      FieldInfos segmentFieldInfos = segmentReader.fieldInfos();
      int numFieldInfos = segmentFieldInfos.size();
```

```
    //依次比较要合并的段和新生成的段的段名，顺序是否一致。
    for (int j = 0; same && j < numFieldInfos; j++) {
      same = fieldInfos.fieldName(j).equals(segmentFieldInfos.fieldName(j));
    }
    //最后生成 matchingSegmentReaders 数组，如果此数组的第 i 项不是 null，则说明第 i 个段同新生成的段名称，顺序完全一致，可以采取情况一得方式。如果此数组的第 i 项是 null，则说明第 i 个段包含特殊的域，则采取情况二的方式。
    if (same) {
      matchingSegmentReaders[i] = segmentReader;
    }
   }
  }
}
```

- 生成存储域的写对象：FieldsWriter fieldsWriter = new FieldsWriter(directory, segment, fieldInfos);
- 依次遍历所有的要合并的段，按照上述两种情况，使用不同策略进行合并

```
int idx = 0;
for (IndexReader reader : readers) {
  final SegmentReader matchingSegmentReader = matchingSegmentReaders[idx++];
  FieldsReader matchingFieldsReader = null;
  // 如果 matchingSegmentReader!=null，表示此段属于情况一，得到 matchingFieldsReader
  if (matchingSegmentReader != null) {
    final FieldsReader fieldsReader = matchingSegmentReader.getFieldsReader();
    if (fieldsReader != null && fieldsReader.canReadRawDocs()) {
      matchingFieldsReader = fieldsReader;
    }
  }
```

```
  //根据此段是否包含删除的文档采取不同的策略
  if (reader.hasDeletions()) {
    docCount += copyFieldsWithDeletions(fieldsWriter, reader, matchingFieldsReader);
  } else {
    docCount += copyFieldsNoDeletions(fieldsWriter,reader, matchingFieldsReader);
  }
}
```

- 合并包含删除文档的段

```
private int copyFieldsWithDeletions(final FieldsWriter fieldsWriter, final IndexReader reader,
                                    final FieldsReader matchingFieldsReader)
  throws IOException, MergeAbortedException, CorruptIndexException {
  int docCount = 0;
  final int maxDoc = reader.maxDoc();
  //matchingFieldsReader!=null，说明此段属于情况一， 则可以直接拷贝。
  if (matchingFieldsReader != null) {
    for (int j = 0; j < maxDoc;) {
      if (reader.isDeleted(j)) {
        // 如果文档被删除，则跳过此文档。
        ++j;
        continue;
      }
      int start = j, numDocs = 0;
      do {
        j++;
        numDocs++;
        if (j >= maxDoc) break;
        if (reader.isDeleted(j)) {
          j++;
```

```
        break;
       }
     } while(numDocs < MAX_RAW_MERGE_DOCS);
     //从要合并的段中从第 start 篇文档开始，依次读取 numDocs 篇文档的文档长度到 rawDocLengths 中。
     IndexInput stream = matchingFieldsReader.rawDocs(rawDocLengths, start, numDocs);
     //用 fieldsStream.copyBytes(...)直接将 fdt 信息从要合并的段拷贝到新生成的段，然后将上面读出的 rawDocLengths 转换成为每篇文档在 fdt 中的偏移量，写入 fdx 文件。
     fieldsWriter.addRawDocuments(stream, rawDocLengths, numDocs);
     docCount += numDocs;
     checkAbort.work(300 * numDocs);
    }
   } else {
    //matchingFieldsReader==null，说明此段属于情况二，必须每篇文档依次添加。
    for (int j = 0; j < maxDoc; j++) {
     if (reader.isDeleted(j)) {
      //  如果文档被删除，则跳过此文档。
       continue;
     }
     //同 addDocument 的过程中一样，重新将文档添加一遍。
     Document doc = reader.document(j);
     fieldsWriter.addDocument(doc);
     docCount++;
     checkAbort.work(300);
    }
   }
   return docCount;
}
```

- 合并不包含删除文档的段：除了跳过删除的文档的部分，同上述过程一样。
- 关闭存储域的写对象：fieldsWriter.close();

2.3.2、合并标准化因子

合并标准化因子的过程比较简单，基本就是对每一个域，用指向合并段的 reader 读出标准化因子，然后再写入新生成的段。

```
private void mergeNorms() throws IOException {
  byte[] normBuffer = null;
  IndexOutput output = null;
  try {
   int numFieldInfos = fieldInfos.size();
  //对于每一个域
   for (int i = 0; i < numFieldInfos; i++) {
    FieldInfo fi = fieldInfos.fieldInfo(i);
    if (fi.isIndexed && !fi.omitNorms) {
     if (output == null) {
        //指向新生成的段的 nrm 文件的写入流
      output = directory.createOutput(segment + "." + IndexFileNames.NORMS_EXTENSION);
        //写 nrm 文件头
      output.writeBytes(NORMS_HEADER,NORMS_HEADER.length);
     }
      //对于每一个合并段的 reader
     for ( IndexReader reader : readers) {
      int maxDoc = reader.maxDoc();
      if (normBuffer == null || normBuffer.length < maxDoc) {
       // the buffer is too small for the current segment
       normBuffer = new byte[maxDoc];
      }
```

```
          //读出此段的 nrm 信息。
        reader.norms(fi.name, normBuffer, 0);
        if (!reader.hasDeletions()) {
            //如果没有文档被删除则写入新生成的段。
          output.writeBytes(normBuffer, maxDoc);
        } else {
            //如果有文档删除则跳过删除的文档写入新生成的段。
          for (int k = 0; k < maxDoc; k++) {
            if (!reader.isDeleted(k)) {
              output.writeByte(normBuffer[k]);
            }
          }
        }
        checkAbort.work(maxDoc);
      }
    }
   }
  } finally {
   if (output != null) {
     output.close();
   }
  }
}
```

2.3.3、合并词向量

合并词向量的过程同合并存储域的过程非常相似，也包括两种情况：

- 情况一：通常情况，要合并的段和新生成段包含的域的名称，顺序都是一样的，这样就可以把要合并的段的词向量信息直接拷贝到新生成段的最后，以提高合并效率。

- 情况二：要合并的段包含特殊的文档，其包含的域多于或者少于新生成段的域，这样就不能够直接拷贝，而是一篇文档一篇文档的添加。这样合并效率大大降低，因而不鼓励添加文档的时候，不同的文档使用不同的域。

具体过程如下：

- 生成词向量的写对象：TermVectorsWriter termVectorsWriter = new TermVectorsWriter(directory, segment, fieldInfos);
- 依次遍历所有的要合并的段，按照上述两种情况，使用不同策略进行合并

```
int idx = 0;
for (final IndexReader reader : readers) {
  final SegmentReader matchingSegmentReader = matchingSegmentReaders[idx++];
  TermVectorsReader matchingVectorsReader = null;
   //如果matchingSegmentReader!=null，表示此段属于情况一，得到matchingFieldsReader
  if (matchingSegmentReader != null) {
    TermVectorsReader vectorsReader = matchingSegmentReader.getTermVectorsReaderOrig();
    if (vectorsReader != null && vectorsReader.canReadRawDocs()) {
      matchingVectorsReader = vectorsReader;
    }
  }
  //根据此段是否包含删除的文档采取不同的策略
  if (reader.hasDeletions()) {
    copyVectorsWithDeletions(termVectorsWriter, matchingVectorsReader, reader);
  } else {
    copyVectorsNoDeletions(termVectorsWriter, matchingVectorsReader, reader);
  }
}
```

- 合并包含删除文档的段

```
private void copyVectorsWithDeletions(final TermVectorsWriter termVectorsWriter, final
```

```
TermVectorsReader matchingVectorsReader, final IndexReader reader)
  throws IOException, MergeAbortedException {
  final int maxDoc = reader.maxDoc();
  //matchingFieldsReader!=null，说明此段属于情况一， 则可以直接拷贝。
  if (matchingVectorsReader != null) {
   for (int docNum = 0; docNum < maxDoc;) {
    if (reader.isDeleted(docNum)) {
      // 如果文档被删除，则跳过此文档。
     ++docNum;
     continue;
    }
    int start = docNum, numDocs = 0;
    do {
     docNum++;
     numDocs++;
     if (docNum >= maxDoc) break;
     if (reader.isDeleted(docNum)) {
      docNum++;
      break;
     }
    } while(numDocs < MAX_RAW_MERGE_DOCS);
    //从要合并的段中从第 start 篇文档开始，依次读取 numDocs 篇文档的 tvd 到 rawDocLengths 中，tvf 到 rawDocLengths2。
    matchingVectorsReader.rawDocs(rawDocLengths, rawDocLengths2, start, numDocs);
    //用 tvd.copyBytes(...)直接将 tvd 信息从要合并的段拷贝到新生成的段，然后将上面读出的 rawDocLengths 转换成为每篇文档在 tvd 文件中的偏移量，写入 tvx 文件。用 tvf.copyBytes(...)直接将 tvf 信息从要合并的段拷贝到新生成的段，然后将上面读出的 rawDocLengths2 转换成为每篇文档在 tvf 文件中的偏移量，写入 tvx 文件。
```

```
            termVectorsWriter.addRawDocuments(matchingVectorsReader, rawDocLengths,
rawDocLengths2, numDocs);
      checkAbort.work(300 * numDocs);
    }
  } else {
    //matchingFieldsReader==null，说明此段属于情况二，必须每篇文档依次添加。
    for (int docNum = 0; docNum < maxDoc; docNum++) {
      if (reader.isDeleted(docNum)) {
        //  如果文档被删除，则跳过此文档。
        continue;
      }
      //同 addDocument 的过程中一样，重新将文档添加一遍。
      TermFreqVector[] vectors = reader.getTermFreqVectors(docNum);
      termVectorsWriter.addAllDocVectors(vectors);
      checkAbort.work(300);
    }
  }
}
```

- 合并不包含删除文档的段：除了跳过删除的文档的部分，同上述过程一样。
- 关闭词向量的写对象：termVectorsWriter.close();

2.3.4、合并词典和倒排表

以上都是合并正向信息，相对过程比较清晰。而合并词典和倒排表就不这么简单了，因为在词典中，Lucene 要求按照字典顺序排序，在倒排表中，文档号要按照从小到大顺序排序排序，在每个段中，文档号都是从零开始编号的。

所以反向信息的合并包括两部分：

- 对字典的合并，需要对词典中的 Term 进行重新排序
- 对于相同的 Term，对包含此 Term 的文档号列表进行合并，需要对文档号重新编号。

后者相对简单，假设如果第一个段的编号是 0~N，第二个段的编号是 0~M，当两个段合并成一个段的时候，第一个段的编号依然是 0~N，第二个段的编号变成 N~N+M 就可以了，也即增加一个偏移量(前一个段的文档个数)。

对词典的合并需要找出两个段中相同的词，Lucene 是通过一个称为 match 的 SegmentMergeInfo 类型的数组以及称为 queue 的 SegmentMergeQueue 实现的，SegmentMergeQueue 是继承于 PriorityQueue<SegmentMergeInfo>，是一个优先级队列，是按照字典顺序排序的。SegmentMergeInfo 保存要合并的段的词典及倒排表信息，在 SegmentMergeQueue 中用来排序的 key 是它代表的段中的第一个 Term。

在总论部分，举了一个例子表明词典和倒排表合并的过程。

下面让我们深入代码看一看具体的实现：

(1) 生成优先级队列，并将所有的段都加入优先级队列。

```
//在 Lucene 索引过程分析(4)中提到过，FormatPostingsFieldsConsumer 是用来写入倒排表信息的。
//FormatPostingsFieldsWriter.addField(FieldInfo field)用于添加索引域信息，其返回 FormatPostingsTermsConsumer 用于添加词信息。
//FormatPostingsTermsConsumer.addTerm(char[] text, int start)用于添加词信息，其返回 FormatPostingsDocsConsumer 用于添加 freq 信息
//FormatPostingsDocsConsumer.addDoc(int docID, int termDocFreq)用于添加 freq 信息，其返回 FormatPostingsPositionsConsumer 用于添加 prox 信息
//FormatPostingsPositionsConsumer.addPosition(int position, byte[] payload, int payloadOffset, int payloadLength)用于添加 prox 信息
FormatPostingsFieldsConsumer consumer = new FormatPostingsFieldsWriter(state, fieldInfos);
//优先级队列
queue = new SegmentMergeQueue(readers.size());
//对于每一个段
final int readerCount = readers.size();
for (int i = 0; i < readerCount; i++) {
    IndexReader reader = readers.get(i);
```

```
    TermEnum termEnum = reader.terms();
    //生成 SegmentMergeInfo 对象，termEnum 就是此段的词典及倒排表。
    SegmentMergeInfo smi = new SegmentMergeInfo(base, termEnum, reader);
    //base 就是下一个段的文档号偏移量，等于此段的文档数目。
    base += reader.numDocs();
    if (smi.next()) //得到段的第一个 Term
        queue.add(smi); //将此段放入优先级队列。
    else
        smi.close();
}
```

（2）生成 match 数组

```
SegmentMergeInfo[] match = new SegmentMergeInfo[readers.size()];
```

（3）合并词典

```
//如果队列不为空，则合并尚未结束
while (queue.size() > 0) {
  int matchSize = 0;
  //取出优先级队列的第一个段，放到 match 数组中
  match[matchSize++] = queue.pop();
  Term term = match[0].term;
  SegmentMergeInfo top = queue.top();
  //如果优先级队列的最顶端和已经弹出的 match 中的段的第一个 Term 相同，则全部弹出。
  while (top != null && term.compareTo(top.term) == 0) {
    match[matchSize++] =   queue.pop();
    top =   queue.top();
  }
  if (currentField != term.field) {
    currentField = term.field;
    if (termsConsumer != null)
```

```
    termsConsumer.finish();
   final FieldInfo fieldInfo = fieldInfos.fieldInfo(currentField);
   //FormatPostingsFieldsWriter.addField(FieldInfo field)用于添加索引域信息，其返回 FormatPostingsTermsConsumer 用于添加词信息。
   termsConsumer = consumer.addField(fieldInfo);
   omitTermFreqAndPositions = fieldInfo.omitTermFreqAndPositions;
  }
  //合并 match 数组中的所有的段的第一个 Term 的倒排表信息，并写入新生成的段。
  int df = appendPostings(termsConsumer, match, matchSize);
  checkAbort.work(df/3.0);
  while (matchSize > 0) {
   SegmentMergeInfo smi = match[—matchSize];
   //如果 match 中的段还有下一个 Term，则放回优先级队列，进行下一轮的循环。
   if (smi.next())
    queue.add(smi);
   else
    smi.close();
  }
}
```

(4) 合并倒排表

```
private final int appendPostings(final FormatPostingsTermsConsumer termsConsumer, SegmentMergeInfo[] smis, int n)
    throws CorruptIndexException, IOException {
//FormatPostingsTermsConsumer.addTerm(char[] text, int start)用于添加词信息，其返回 FormatPostingsDocsConsumer 用于添加 freq 信息
  //将 match 数组中段的第一个 Term 添加到新生成的段中。
  final FormatPostingsDocsConsumer docConsumer = termsConsumer.addTerm(smis[0].term.text);
  int df = 0;
```

```
for (int i = 0; i < n; i++) {
  SegmentMergeInfo smi = smis[i];
  //得到要合并的段的位置信息(prox)
  TermPositions postings = smi.getPositions();
  //此段的文档号偏移量
  int base = smi.base;
  //在要合并的段中找到 Term 的倒排表位置。
  postings.seek(smi.termEnum);
  //不断得到下一篇文档号
  while (postings.next()) {
    df++;
    int doc = postings.doc();
     //文档号都要加上偏移量
    doc += base;
    //得到词频信息(frq)
    final int freq = postings.freq();
    //FormatPostingsDocsConsumer.addDoc(int docID, int termDocFreq)用于添加 freq 信息，其返回 FormatPostingsPositionsConsumer 用于添加 prox 信息
    final FormatPostingsPositionsConsumer posConsumer = docConsumer.addDoc(doc, freq);
     //如果位置信息需要保存
    if (!omitTermFreqAndPositions) {
      for (int j = 0; j < freq; j++) {
         //得到位置信息(prox)以及 payload 信息
        final int position = postings.nextPosition();
        final int payloadLength = postings.getPayloadLength();
        if (payloadLength > 0) {
          if (payloadBuffer == null || payloadBuffer.length < payloadLength)
            payloadBuffer = new byte[payloadLength];
```

```
        postings.getPayload(payloadBuffer, 0);
       }
        //FormatPostingsPositionsConsumer.addPosition(int position, byte[]
payload, int payloadOffset, int payloadLength)用于添加 prox 信息
        posConsumer.addPosition(position, payloadBuffer, 0, payloadLength);
      }
      posConsumer.finish();
    }
   }
  }
  docConsumer.finish();
  return df;
}
```

第六章：Lucene 打分公式的数学推导

在进行 Lucene 的搜索过程解析之前，有必要单独的一张把 Lucene score 公式的推导，各部分的意义阐述一下。因为 Lucene 的搜索过程，很重要的一个步骤就是逐步的计算各部分的分数。

Lucene 的打分公式非常复杂，如下：

$$score(q,d) = coord(q,d) \times queryNorm(q) \times \sum_{t in q} (tf(t in d) \times idf(t)^2 \times t.getBoost() \times norm(t,d))$$

在推导之前，先逐个介绍每部分的意义：

- t：Term，这里的 Term 是指包含域信息的 Term，也即 title:hello 和 content:hello 是不同的 Term
- coord(q,d)：一次搜索可能包含多个搜索词，而一篇文档中也可能包含多个搜索词，此项表示，当一篇文档中包含的搜索词越多，则此文档则打分越高。
- queryNorm(q)：计算每个查询条目的方差和，此值并不影响排序，而仅仅使得不同的 query 之间的分数可以比较。其公式如下：

$$queryNorm(q) = \frac{1}{\sqrt{q.getBoost()^2 \times \sum_{t in q} (idf(t) \times t.getBoost())^2}}$$

- tf(t in d)：Term t 在文档 d 中出现的词频
- idf(t)：Term t 在几篇文档中出现过
- norm(t, d)：标准化因子，它包括三个参数：
  - Document boost：此值越大，说明此文档越重要。
  - Field boost：此域越大，说明此域越重要。
  - lengthNorm(field) = (1.0 / Math.sqrt(numTerms))：一个域中包含的 Term 总数越多，也即文档越长，此值越小，文档越短，此值越大。

$$norm(t,d) = d.getBoost() \times lengthNorm(field) \times \prod_{field f in d} f.getBoost()$$

$$lengthNorm(f) = \frac{1}{\sqrt{num\ of\ terms\ in\ field\ f}}$$

- 各类 Boost 值
  - t.getBoost()：查询语句中每个词的权重，可以在查询中设定某个词更加重要，common^4 hello
  - d.getBoost()：文档权重，在索引阶段写入 nrm 文件，表明某些文档比其他文档更重要。
  - f.getBoost()：域的权重，在索引阶段写入 nrm 文件，表明某些域比其他的域更重要。

以上在 Lucene 的文档中已经详细提到，并在很多文章中也被阐述过，如何调整上面的各部分，以影响文档的打分，请参考有关 Lucene 的问题(4):影响 Lucene 对文档打分的四种方式一文。

然而上面各部分为什么要这样计算在一起呢？这么复杂的公式是怎么得出来的呢？下面我们来推导。

首先，将以上各部分代入 score(q, d)公式，将得到一个非常复杂的公式，让我们忽略所有的 boost，因为这些属于人为的调整，也省略 coord，这和公式所要表达的原理无关。得到下面的公式：

$$score(q,d) = \frac{1}{\sqrt{\sum_{t in q} idf(t)^2}} \times \sum_{t in q} tf(t \text{ in } d) \times idf(t)^2 \times \frac{1}{\sqrt{num\ of\ terms\ in\ field\ f}}$$

去掉boost信息的lengthNorm部分　　去掉boost信息的norm部分

然后，有 Lucene 学习总结之一：全文检索的基本原理中的描述我们知道，Lucene 的打分机制是采用向量空间模型的：

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}

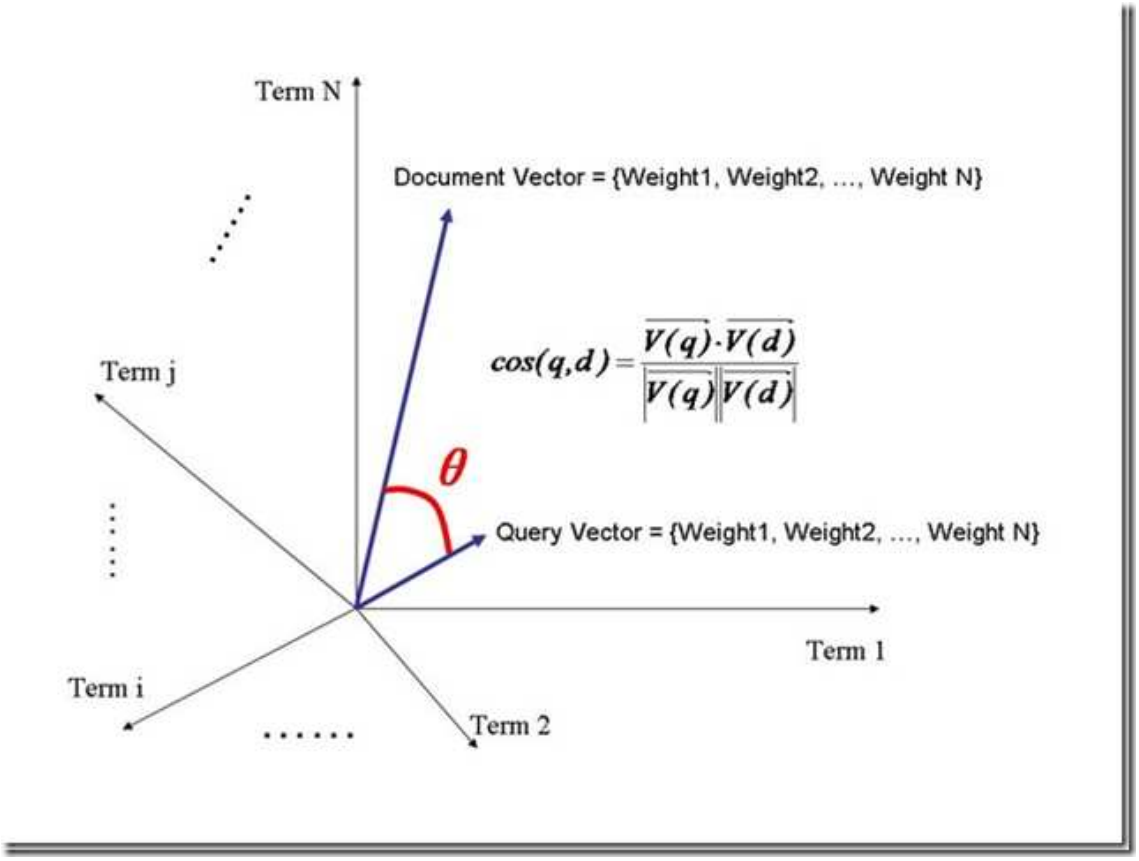
Document Vector = {weight1, weight2, …… ,weight N}

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。



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。

余弦公式如下：

$$score(q,d) = \cos(\theta) = \frac{\overrightarrow{V_q} \bullet \overrightarrow{V_d}}{\left|\overrightarrow{V_q}\right| \times \left|\overrightarrow{V_d}\right|}$$

下面我们假设：

查询向量为 Vq = <w(t1, q), w(t2, q), ……, w(tn, q)>

文档向量为 Vd = <w(t1, d), w(t2, d), ……, w(tn, d)>

向量空间维数为 n，是查询语句和文档的并集的长度，当某个 Term 不在查询语句中出现的时候，w(t, q)为零，当某个 Term 不在文档中出现的时候，w(t, d)为零。

w 代表 weight，计算公式一般为 tf*idf。

我们首先计算余弦公式的分子部分，也即两个向量的点积：

Vq*Vd = w(t1, q)*w(t1, d) + w(t2, q)*w(t2, d) + …… + w(tn ,q)*w(tn, d)

把 w 的公式代入，则为

Vq*Vd = tf(t1, q)*idf(t1, q)*tf(t1, d)*idf(t1, d) + tf(t2, q)*idf(t2, q)*tf(t2, d)*idf(t2, d) + …… + tf(tn ,q)*idf(tn, q)*tf(tn, d)*idf(tn, d)

在这里有三点需要指出：

- 由于是点积，则此处的 t1, t2, ……, tn 只有查询语句和文档的并集有非零值，只在查询语句出现的或只在文档中出现的 Term 的项的值为零。
- 在查询的时候，很少有人会在查询语句中输入同样的词，因而可以假设 tf(t, q)都为 1
- idf 是指 Term 在多少篇文档中出现过，其中也包括查询语句这篇小文档，因而 idf(t, q)和 idf(t, d)其实是一样的，是索引中的文档总数加一，当索引中的文档总数足够大的时候，查询语句这篇小文档可以忽略，因而可以假设 idf(t, q) = idf(t, d) = idf(t)

基于上述三点，点积公式为：

Vq*Vd = tf(t1, d) * idf(t1) * idf(t1) + tf(t2, d) * idf(t2) * idf(t2) + …… + tf(tn, d) * idf(tn) * idf(tn)

所以余弦公式变为：

$$score(q,d) = \cos(\theta) = \frac{\overrightarrow{V_q} \bullet \overrightarrow{V_d}}{\left|\overrightarrow{V_q}\right| \times \left|\overrightarrow{V_d}\right|} = \frac{1}{\left|\overrightarrow{V_q}\right|} \times \sum_{t in q} (tf(t,d) \times idf(t)^2 \times \frac{1}{\left|\overrightarrow{V_d}\right|})$$

下面要推导的就是查询语句的长度了。

由上面的讨论，查询语句中 tf 都为 1，idf 都忽略查询语句这篇小文档，得到如下公式

$$\left|\overrightarrow{V_q}\right| = \sqrt{w(t_1,q)^2 + w(t_2,q)^2 + ... + w(t_n,q)^2}$$
$$= \sqrt{\sum_{t in q} w(t,q)^2} = \sqrt{\sum_{t in q} (tf(t,q) \times idf(t,q))^2}$$
$$= \sqrt{\sum_{t in q} idf(t)^2}$$

所以余弦公式变为：

$$score(q,d) = \cos(\theta) = \frac{\overrightarrow{V_q} \bullet \overrightarrow{V_d}}{\left|\overrightarrow{V_q}\right| \times \left|\overrightarrow{V_d}\right|} = \frac{1}{\sqrt{\sum_{tinq} idf(t)^2}} \times \sum_{tinq} (tf(t,d) \times idf(t)^2 \times \frac{1}{\left|\overrightarrow{V_d}\right|})$$

下面推导的就是文档的长度了，本来文档长度的公式应该如下：

$$\left|\overrightarrow{V_d}\right| = \sqrt{w(t_1,d)^2 + w(t_2,d)^2 + \ldots + w(t_n,d)^2} = \sqrt{\sum_{tind} w(t,d)^2}$$

这里需要讨论的是，为什么在打分过程中，需要除以文档的长度呢？

因为在索引中，不同的文档长度不一样，很显然，对于任意一个 term，在长的文档中的 tf 要大的多，因而分数也越高，这样对小的文档不公平，举一个极端的例子，在一篇 1000 万个词的鸿篇巨著中，"lucene"这个词出现了 11 次，而在一篇 12 个词的短小文档中，"lucene"这个词出现了 10 次，如果不考虑长度在内，当然鸿篇巨著应该分数更高，然而显然这篇小文档才是真正关注"lucene"的。

然而如果按照标准的余弦计算公式，完全消除文档长度的影响，则又对长文档不公平(毕竟它是包含了更多的信息)，偏向于首先返回短小的文档的，这样在实际应用中使得搜索结果很难看。

所以在 Lucene 中，Similarity 的 lengthNorm 接口是开放出来，用户可以根据自己应用的需要，改写 lengthNorm 的计算公式。比如我想做一个经济学论文的搜索系统，经过一定时间的调研，发现大多数的经济学论文的长度在 8000 到 10000 词，因而 lengthNorm 的公式应该是一个倒抛物线型的，8000 到 10000 词的论文分数最高，更短或更长的分数都应该偏低，方能够返回给用户最好的数据。

在默认状况下，Lucene 采用 DefaultSimilarity，认为在计算文档的向量长度的时候，每个 Term 的权重就不再考虑在内了，而是全部为一。

而从 Term 的定义我们可以知道，Term 是包含域信息的，也即 title:hello 和 content:hello 是不同的 Term，也即一个 Term 只可能在文档中的一个域中出现。

所以文档长度的公式为：

$$\left|\overrightarrow{V_d}\right| = \sqrt{\sum_{t in d} w(t,d)^2} = \sqrt{\sum_{t in f} 1^2} = \sqrt{num\ of\ terms\ in\ field\ f}$$

代入余弦公式：

$$score(q,d) = \cos(\theta) = \frac{\overrightarrow{V_q} \bullet \overrightarrow{V_d}}{\left|\overrightarrow{V_q}\right| \times \left|\overrightarrow{V_d}\right|} = \frac{1}{\sqrt{\sum_{t in q} idf(t)^2}} \times \sum_{t in q} (tf(t,d) \times idf(t)^2 \times \frac{1}{\sqrt{num\ of\ terms\ in\ field\ f}})$$

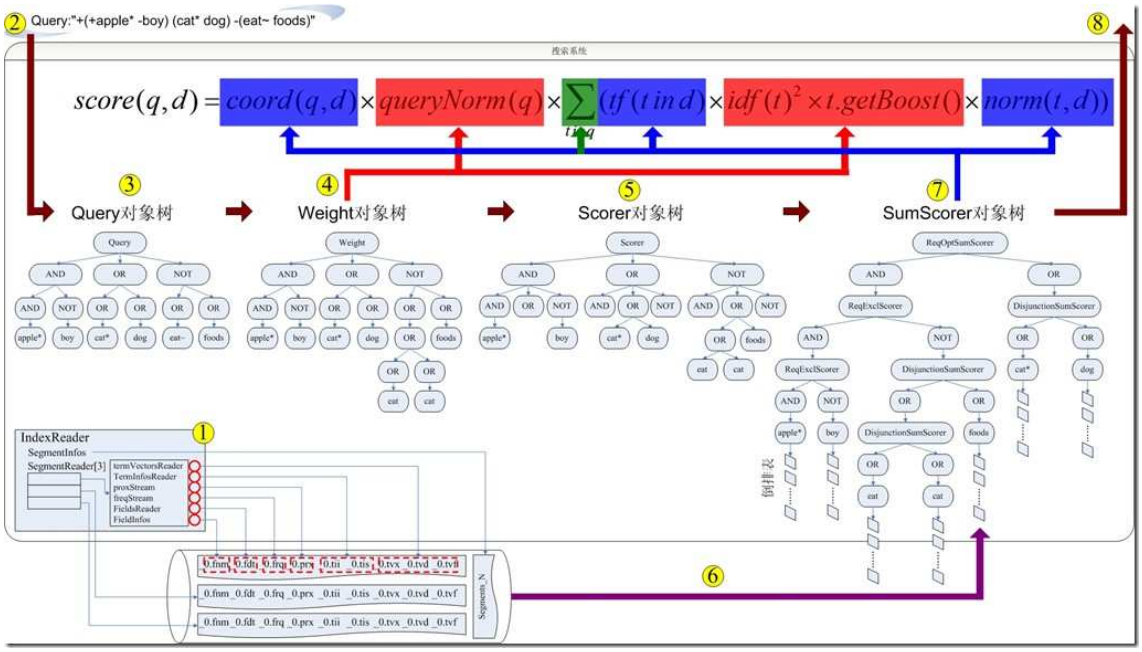
再加上各种 boost 和 coord，则可得出 Lucene 的打分计算公式。

第七章：Lucene 搜索过程解析

一、Lucene 搜索过程总论

搜索的过程总的来说就是将词典及倒排表信息从索引中读出来，根据用户输入的查询语句合并倒排表，得到结果文档集并对文档进行打分的过程。

其可用如下图示：



总共包括以下几个过程：

1. IndexReader 打开索引文件，读取并打开指向索引文件的流。
2. 用户输入查询语句
3. 将查询语句转换为查询对象 Query 对象树
4. 构造 Weight 对象树，用于计算词的权重 Term Weight，也即计算打分公式中与仅与搜索语句相关与文档无关的部分(红色部分)。
5. 构造 Scorer 对象树，用于计算打分(TermScorer.score())。
6. 在构造 Scorer 对象树的过程中，其叶子节点的 TermScorer 会将词典和倒排表从索引中读出来。

7. 构造 SumScorer 对象树，其是为了方便合并倒排表对 Scorer 对象树的从新组织，它的叶子节点仍为 TermScorer，包含词典和倒排表。此步将倒排表合并后得到结果文档集，并对结果文档计算打分公式中的蓝色部分。打分公式中的求和符合，并非简单的相加，而是根据子查询倒排表的合并方式(与或非)来对子查询的打分求和，计算出父查询的打分。
8. 将收集的结果集合及打分返回给用户。

二、Lucene 搜索详细过程

为了解析 Lucene 对索引文件搜索的过程，预先写入索引了如下几个文件：

file01.txt: apple apples cat dog

file02.txt: apple boy cat category

file03.txt: apply dog eat etc

file04.txt: apply cat foods

2.1、打开 IndexReader 指向索引文件夹

代码为：

IndexReader reader = IndexReader.open(FSDirectory.open(indexDir));

其实是调用了 DirectoryReader.open(Directory, IndexDeletionPolicy, IndexCommit, boolean, int)函数，其主要作用是生成一个 SegmentInfos.FindSegmentsFile 对象，并用它来找到此索引文件中所有的段，并打开这些段。

SegmentInfos.FindSegmentsFile.run(IndexCommit commit)主要做以下事情：

2.1.1、找到最新的 segment_N 文件

- 由于 segment_N 是整个索引中总的元数据，因而正确的选择 segment_N 更加重要。
- 然而有时候为了使得索引能够保存在另外的存储系统上，有时候需要用 NFS mount 一个远程的磁盘来存放索引，然而 NFS 为了提高性能，在本地有 Cache，因而有可能使得此次打开的索引不是另外的 writer 写入的最新信息，所以在此处用了双保险。

- 一方面，列出所有的 segment_N，并取出其中的最大的 N，设为 genA

```
String[] files = directory.listAll();
long genA = getCurrentSegmentGeneration(files);
```

```
long getCurrentSegmentGeneration(String[] files) {
    long max = -1;
    for (int i = 0; i < files.length; i++) {
      String file = files[i];
      if (file.startsWith(IndexFileNames.SEGMENTS) //"segments_N"
          && !file.equals(IndexFileNames.SEGMENTS_GEN)) { //"segments.gen"
        long gen = generationFromSegmentsFileName(file);
        if (gen > max) {
          max = gen;
        }
      }
    }
    return max;
  }
```

- 另一方面，打开 segment.gen 文件，从中读出 N，设为 genB

```
IndexInput genInput = directory.openInput(IndexFileNames.SEGMENTS_GEN);
int version = genInput.readInt();
long gen0 = genInput.readLong();
long gen1 = genInput.readLong();
if (gen0 == gen1) {
    genB = gen0;
}
```

- 在 genA 和 genB 中去较大者，为 gen，并用此 gen 构造要打开的 segments_N 的文件名

```
if (genA > genB)
    gen = genA;
```

```
else
    gen = genB;
String segmentFileName = IndexFileNames.fileNameFromGeneration(IndexFileNames.SEGMENTS,
"", gen); //segmentFileName    "segments_4"
```

2.1.2、通过 segment_N 文件中保存的各个段的信息打开各个段

- 从 segment_N 中读出段的元数据信息，生成 SegmentInfos

```
SegmentInfos infos = new SegmentInfos();
infos.read(directory, segmentFileName);
```

SegmentInfos.read(Directory, String) 代码如下：

```
int format = input.readInt();
version = input.readLong();
counter = input.readInt();
for (int i = input.readInt(); i > 0; i—) {
  //读出每一个段，并构造 SegmentInfo 对象
  add(new SegmentInfo(directory, format, input));
}
```

SegmentInfo(Directory dir, int format, IndexInput input)构造函数如下：

```
name = input.readString();
docCount = input.readInt();
delGen = input.readLong();
docStoreOffset = input.readInt();
if (docStoreOffset != -1) {
  docStoreSegment = input.readString();
  docStoreIsCompoundFile = (1 == input.readByte());
```

```
} else {
   docStoreSegment = name;
   docStoreIsCompoundFile = false;
}
hasSingleNormFile = (1 == input.readByte());
int numNormGen = input.readInt();
normGen = new long[numNormGen];
for(int j=0;j<numNormGen;j++) {
   normGen[j] = input.readLong();
}
isCompoundFile = input.readByte();
delCount = input.readInt();
hasProx = input.readByte() == 1;
```

其实不用多介绍，看过 Lucene 学习总结之三：Lucene 的索引文件格式（2)一章，就很容易明白。

- 根据生成的 SegmentInfos 打开各个段，并生成 ReadOnlyDirectoryReader

```
SegmentReader[] readers = new SegmentReader[sis.size()];
for (int i = sis.size()-1; i >= 0; i—) {
   //打开每一个段
   readers[i] = SegmentReader.get(readOnly, sis.info(i), termInfosIndexDivisor);
}
```

SegmentReader.get(boolean, Directory, SegmentInfo, int, boolean, int) 代码如下：

```
instance.core = new CoreReaders(dir, si, readBufferSize, termInfosIndexDivisor);
instance.core.openDocStores(si); //生成用于读取存储域和词向量的对象。
instance.loadDeletedDocs(); //读取被删除文档(.del)文件
instance.openNorms(instance.core.cfsDir, readBufferSize); //读取标准化因子(.nrm)
```

CoreReaders(Directory dir, SegmentInfo si, int readBufferSize, int termsIndexDivisor)构造函数代码如下：

```
cfsReader = new CompoundFileReader(dir, segment + "." + IndexFileNames.COMPOUND_FILE_EXTENSION, readBufferSize); //读取 cfs 的 reader
fieldInfos = new FieldInfos(cfsDir, segment + "." + IndexFileNames.FIELD_INFOS_EXTENSION); //读取段元数据信息(.fnm)
TermInfosReader reader = new TermInfosReader(cfsDir, segment, fieldInfos, readBufferSize, termsIndexDivisor); //用于读取词典信息(.tii .tis)
freqStream = cfsDir.openInput(segment + "." + IndexFileNames.FREQ_EXTENSION, readBufferSize); //用于读取 freq
proxStream = cfsDir.openInput(segment + "." + IndexFileNames.PROX_EXTENSION, readBufferSize); //用于读取 prox
```

FieldInfos(Directory d, String name)构造函数如下：

```
IndexInput input = d.openInput(name);
int firstInt = input.readVInt();
size = input.readVInt();
for (int i = 0; i < size; i++) {
  //读取域名
  String name = StringHelper.intern(input.readString());
  //读取域的各种标志位
  byte bits = input.readByte();
  boolean isIndexed = (bits & IS_INDEXED) != 0;
  boolean storeTermVector = (bits & STORE_TERMVECTOR) != 0;
  boolean storePositionsWithTermVector = (bits & STORE_POSITIONS_WITH_TERMVECTOR) != 0;
  boolean storeOffsetWithTermVector = (bits & STORE_OFFSET_WITH_TERMVECTOR) != 0;
  boolean omitNorms = (bits & OMIT_NORMS) != 0;
  boolean storePayloads = (bits & STORE_PAYLOADS) != 0;
  boolean omitTermFreqAndPositions = (bits & OMIT_TERM_FREQ_AND_POSITIONS) != 0;
  //将读出的域生成 FieldInfo 对象，加入 fieldinfos 进行管理
      addInternal(name, isIndexed, storeTermVector, storePositionsWithTermVector,
```

storeOffsetWithTermVector, omitNorms, storePayloads, omitTermFreqAndPositions);

}

CoreReaders.openDocStores(SegmentInfo)主要代码如下：

fieldsReaderOrig = new FieldsReader(storeDir, storesSegment, fieldInfos, readBufferSize, si.getDocStoreOffset(), si.docCount); **//用于读取存储域(.fdx, .fdt)**

termVectorsReaderOrig = new TermVectorsReader(storeDir, storesSegment, fieldInfos, readBufferSize, si.getDocStoreOffset(), si.docCount); **//用于读取词向量(.tvx, .tvd, .tvf)**

- 初始化生成的 ReadOnlyDirectoryReader，对打开的多个 SegmentReader 中的文档编号

在 Lucene 中，每个段中的文档编号都是从 0 开始的，而一个索引有多个段，需要重新进行编号，于是维护数组 start[]，来保存每个段的文档号的偏移量，从而第 i 个段的文档号是从 start[i] 至 start[i]+Num

```
private void initialize(SegmentReader[] subReaders) {
  this.subReaders = subReaders;
  starts = new int[subReaders.length + 1];
  for (int i = 0; i < subReaders.length; i++) {
    starts[i] = maxDoc;
    maxDoc += subReaders[i].maxDoc();
    if (subReaders[i].hasDeletions())
      hasDeletions = true;
  }
  starts[subReaders.length] = maxDoc;
}
```

2.1.3、得到的 IndexReader 对象如下

```
reader    ReadOnlyDirectoryReader  (id=466)
   closed    false
   deletionPolicy    null
```

```
//索引文件夹
directory    SimpleFSDirectory   (id=31)
  checked    false
  chunkSize    104857600
  directory    File   (id=487)
    path    "D:\\lucene-3.0.0\\TestSearch\\index"
    prefixLength    3
  isOpen    true
  lockFactory    NativeFSLockFactory   (id=488)
hasChanges    false
hasDeletions    false
maxDoc    12
normsCache    HashMap<K,V>   (id=483)
numDocs    -1
readOnly    true
refCount    1
rollbackHasChanges    false
rollbackSegmentInfos    null
//段元数据信息
segmentInfos    SegmentInfos   (id=457)
  elementCount    3
  elementData    Object[10]   (id=532)
    [0]    SegmentInfo   (id=464)
      delCount    0
      delGen    -1
      diagnostics    HashMap<K,V>   (id=537)
      dir    SimpleFSDirectory   (id=31)
      docCount    4
```

docStoreIsCompoundFile false

docStoreOffset -1

docStoreSegment "_0"

files null

hasProx true

hasSingleNormFile true

isCompoundFile 1

name "_0"

normGen null

preLockless false

sizeInBytes -1

[1] SegmentInfo (id=517)

delCount 0

delGen -1

diagnostics HashMap<K,V> (id=542)

dir SimpleFSDirectory (id=31)

docCount 4

docStoreIsCompoundFile false

docStoreOffset -1

docStoreSegment "_1"

files null

hasProx true

hasSingleNormFile true

isCompoundFile 1

name "_1"

normGen null

preLockless false

sizeInBytes -1

[2] SegmentInfo (id=470)

delCount 0

delGen -1

diagnostics HashMap<K,V> (id=547)

dir SimpleFSDirectory (id=31)

docCount 4

docStoreIsCompoundFile false

docStoreOffset -1

docStoreSegment "_2"

files null

hasProx true

hasSingleNormFile true

isCompoundFile 1

name "_2"

normGen null

preLockless false

sizeInBytes -1

generation 4

lastGeneration 4

modCount 4

pendingSegnOutput null

userData HashMap<K,V> (id=533)

version 1268193441675

segmentInfosStart null

stale false

starts int[4] (id=484)

//每个段的 Reader

subReaders SegmentReader[3] (id=467)

[0] ReadOnlySegmentReader (id=492)

closed false

core SegmentReader$CoreReaders (id=495)

cfsDir CompoundFileReader (id=552)

cfsReader CompoundFileReader (id=552)

dir SimpleFSDirectory (id=31)

fieldInfos FieldInfos (id=553)

fieldsReaderOrig FieldsReader (id=554)

freqStream CompoundFileReader$CSIndexInput (id=555)

proxStream CompoundFileReader$CSIndexInput (id=556)

readBufferSize 1024

ref SegmentReader$Ref (id=557)

segment "_0"

storeCFSReader null

termsIndexDivisor 1

termVectorsReaderOrig null

tis TermInfosReader (id=558)

tisNoIndex null

deletedDocs null

deletedDocsDirty false

deletedDocsRef null

fieldsReaderLocal SegmentReader$FieldsReaderLocal (id=496)

hasChanges false

norms HashMap<K,V> (id=500)

normsDirty false

pendingDeleteCount 0

readBufferSize 1024

readOnly true

refCount 1

rollbackDeletedDocsDirty false

rollbackHasChanges false

rollbackNormsDirty false

rollbackPendingDeleteCount 0

si SegmentInfo (id=464)

singleNormRef SegmentReader$Ref (id=504)

singleNormStream CompoundFileReader$CSIndexInput (id=506)

termVectorsLocal CloseableThreadLocal<T> (id=508)

[1] ReadOnlySegmentReader (id=493)

closed false

core SegmentReader$CoreReaders (id=511)

cfsDir CompoundFileReader (id=561)

cfsReader CompoundFileReader (id=561)

dir SimpleFSDirectory (id=31)

fieldInfos FieldInfos (id=562)

fieldsReaderOrig FieldsReader (id=563)

freqStream CompoundFileReader$CSIndexInput (id=564)

proxStream CompoundFileReader$CSIndexInput (id=565)

readBufferSize 1024

ref SegmentReader$Ref (id=566)

segment "_1"

storeCFSReader null

termsIndexDivisor 1

termVectorsReaderOrig null

tis TermInfosReader (id=567)

tisNoIndex null

deletedDocs null

deletedDocsDirty false

deletedDocsRef null

fieldsReaderLocal SegmentReader$FieldsReaderLocal (id=512)

hasChanges false

norms HashMap<K,V> (id=514)

normsDirty false

pendingDeleteCount 0

readBufferSize 1024

readOnly true

refCount 1

rollbackDeletedDocsDirty false

rollbackHasChanges false

rollbackNormsDirty false

rollbackPendingDeleteCount 0

si SegmentInfo (id=517)

singleNormRef SegmentReader$Ref (id=519)

singleNormStream CompoundFileReader$CSIndexInput (id=520)

termVectorsLocal CloseableThreadLocal<T> (id=521)

[2] ReadOnlySegmentReader (id=471)

closed false

core SegmentReader$CoreReaders (id=475)

cfsDir CompoundFileReader (id=476)

cfsReader CompoundFileReader (id=476)

dir SimpleFSDirectory (id=31)

fieldInfos FieldInfos (id=480)

fieldsReaderOrig FieldsReader (id=570)

freqStream CompoundFileReader$CSIndexInput (id=571)

proxStream CompoundFileReader$CSIndexInput (id=572)

readBufferSize 1024

ref SegmentReader$Ref (id=573)

segment "_2"

storeCFSReader null

termsIndexDivisor 1

termVectorsReaderOrig null

tis TermInfosReader (id=574)

tisNoIndex null

deletedDocs null

deletedDocsDirty false

deletedDocsRef null

fieldsReaderLocal SegmentReader$FieldsReaderLocal (id=524)

hasChanges false

norms HashMap<K,V> (id=525)

normsDirty false

pendingDeleteCount 0

readBufferSize 1024

readOnly true

refCount 1

rollbackDeletedDocsDirty false

rollbackHasChanges false

rollbackNormsDirty false

rollbackPendingDeleteCount 0

si SegmentInfo (id=470)

singleNormRef SegmentReader$Ref (id=527)

singleNormStream CompoundFileReader$CSIndexInput (id=528)

termVectorsLocal CloseableThreadLocal<T> (id=530)

synced HashSet<E> (id=485)

```
termInfosIndexDivisor    1
writeLock    null
writer    null
```

从上面的过程来看，IndexReader 有以下几个特性：

- 段元数据信息已经被读入到内存中，因而索引文件夹中因为新添加文档而新增加的段对已经打开的 reader 是不可见的。
- .del 文件已经读入内存，因而其他的 reader 或者 writer 删除的文档对打开的 reader 也是不可见的。
- 打开的 reader 已经有 inputstream 指向 cfs 文件，从段合并的过程我们知道，一个段文件从生成起就不会改变，新添加的文档都在新的段中，删除的文档都在.del 中，段之间的合并是生成新的段，而不会改变旧的段，只不过在段的合并过程中，会将旧的段文件删除，这没有问题，因为从操作系统的角度来讲，一旦一个文件被打开一个 inputstream 也即打开了一个文件描述符，在内核中，此文件会保持 reference count，只要 reader 还没有关闭，文件描述符还在，文件是不会被删除的，仅仅 reference count 减一。
- 以上三点保证了 IndexReader 的 snapshot 的性质，也即一个 IndexReader 打开一个索引，就好像对此索引照了一张像，无论背后索引如何改变，此 IndexReader 在被重新打开之前，看到的信息总是相同的。
- 严格的来讲，Lucene 的文档号仅仅对打开的某个 reader 有效，当索引发生了变化，再打开另外一个 reader 的时候，前面 reader 的文档 0 就不一定是后面 reader 的文档 0 了，因而我们进行查询的时候，从结果中得到文档号的时候，一定要在 reader 关闭之前应用，从存储域中得到真正能够唯一标识你的业务逻辑中的文档的信息，如 url，md5 等等，一旦 reader 关闭了，则文档号已经无意义，如果用其他的 reader 查询这些文档号，得到的可能是不期望的文档。

2.2、打开 IndexSearcher

代码为：

```
IndexSearcher searcher = new IndexSearcher(reader);
```

其过程非常简单：

```
private IndexSearcher(IndexReader r, boolean closeReader) {
  reader = r;
  //当关闭 searcher 的时候，是否关闭其 reader
  this.closeReader = closeReader;
  //对文档号进行编号
  List<IndexReader> subReadersList = new ArrayList<IndexReader>();
  gatherSubReaders(subReadersList, reader);
  subReaders = subReadersList.toArray(new IndexReader[subReadersList.size()]);
  docStarts = new int[subReaders.length];
  int maxDoc = 0;
  for (int i = 0; i < subReaders.length; i++) {
    docStarts[i] = maxDoc;
    maxDoc += subReaders[i].maxDoc();
  }
}
```

IndexSearcher 表面上看起来好像仅仅是 reader 的一个封装，它的很多函数都是直接调用 reader 的相应函数，如：int docFreq(Term term)，Document doc(int i)，int maxDoc()。然而它提供了两个非常重要的函数：

- void setSimilarity(Similarity similarity)，用户可以实现自己的 Similarity 对象，从而影响搜索过程的打分，详见有关 Lucene 的问题(4):影响 Lucene 对文档打分的四种方式
- 一系列 search 函数，是搜索过程的关键，主要负责打分的计算和倒排表的合并。

因而在某些应用之中，只想得到某个词的倒排表的时候，最好不要用 IndexSearcher，而直接用 IndexReader.termDocs(Term term)，则省去了打分的计算。

2.3、QueryParser 解析查询语句生成查询对象

代码为：

```
QueryParser parser = new QueryParser(Version.LUCENE_CURRENT, "contents", new
```

```
StandardAnalyzer(Version.LUCENE_CURRENT));
Query query = parser.parse("+(+apple* -boy) (cat* dog) -(eat~ foods)");
```

此过程相对复杂，涉及 JavaCC，QueryParser，分词器，查询语法等，本章不会详细论述，会在后面的章节中一一说明。

此处唯一要说明的是，根据查询语句生成的是一个 Query 树，这棵树很重要，并且会生成其他的树，一直贯穿整个索引过程。

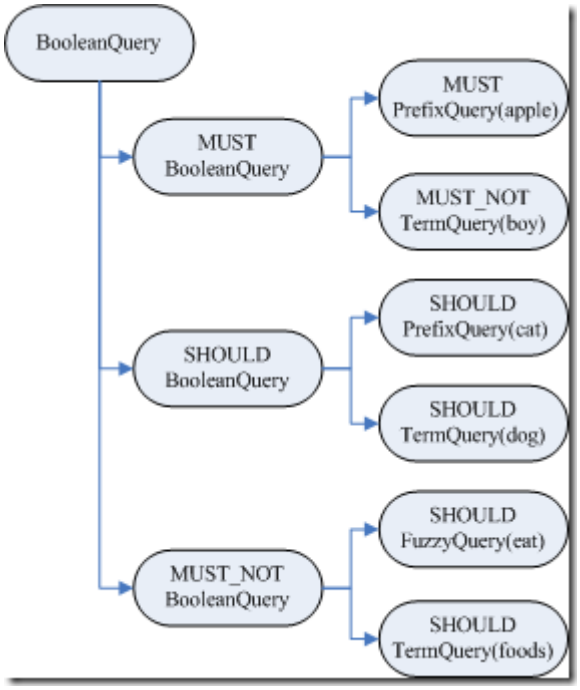
```
query    BooleanQuery  (id=96)
  |  boost    1.0
  |  clauses    ArrayList<E>  (id=98)
  |     elementData    Object[10]  (id=100)
  |------[0]    BooleanClause  (id=102)
  |        |   occur    BooleanClause$Occur$1  (id=106)
  |        |      name    "MUST" //AND
  |        |      ordinal    0
  |        |---query    BooleanQuery  (id=108)
  |            |   boost    1.0
  |            |   clauses    ArrayList<E>  (id=112)
  |            |     elementData    Object[10]  (id=113)
  |            |------[0]    BooleanClause  (id=114)
  |            |        |   occur    BooleanClause$Occur$1  (id=106)
  |            |        |    name    "MUST"   //AND
  |            |        |    ordinal    0
  |            |        |--query    PrefixQuery  (id=116)
  |            |            boost    1.0
  |            |            numberOfTerms    0
  |            |            prefix    Term  (id=117)
  |            |                field    "contents"
  |            |                text    "apple"
```

```
|          |          rewriteMethod    MultiTermQuery$1  (id=119)
|          |            docCountPercent    0.1
|          |            termCountCutoff    350
|          |------[1]    BooleanClause  (id=115)
|               |  occur    BooleanClause$Occur$3  (id=123)
|               |    name    "MUST_NOT"  //NOT
|               |    ordinal    2
|               |--query    TermQuery  (id=125)
|                   boost    1.0
|                          term    Term  (id=127)
|                              field    "contents"
|                              text    "boy"
|            size    2
|          disableCoord    false
|          minNrShouldMatch    0
|------[1]    BooleanClause  (id=104)
|      |  occur    BooleanClause$Occur$2  (id=129)
|      |    name    "SHOULD"  //OR
|      |    ordinal    1
|      |---query    BooleanQuery  (id=131)
|          |  boost    1.0
|          |  clauses    ArrayList<E>  (id=133)
|          |    elementData    Object[10]  (id=134)
|          |------[0]    BooleanClause  (id=135)
|          |      |  occur    BooleanClause$Occur$2  (id=129)
|          |      |    name    "SHOULD"  //OR
|          |      |    ordinal    1
|          |      |--query    PrefixQuery  (id=137)
```

```
|          |          boost    1.0
|          |          numberOfTerms    0
|          |              prefix    Term  (id=138)
|          |                 field    "contents"
|          |                 text    "cat"
|          |          rewriteMethod    MultiTermQuery$1  (id=119)
|          |            docCountPercent    0.1
|          |            termCountCutoff    350
|          |------[1]    BooleanClause  (id=136)
|               | occur    BooleanClause$Occur$2  (id=129)
|               |    name    "SHOULD"  //OR
|               |    ordinal    1
|               |--query    TermQuery  (id=140)
|                  boost    1.0
|                           term    Term  (id=141)
|                              field    "contents"
|                              text    "dog"
|            size    2
|          disableCoord    false
|          minNrShouldMatch    0
|------[2]    BooleanClause  (id=105)
      | occur    BooleanClause$Occur$3  (id=123)
      |    name    "MUST_NOT"  //NOT
      |    ordinal    2
      |---query    BooleanQuery  (id=143)
          | boost    1.0
          | clauses    ArrayList<E>  (id=146)
          |   elementData    Object[10]  (id=147)
```

```
        |------[0]    BooleanClause  (id=148)
        |       |    occur    BooleanClause$Occur$2  (id=129)
        |       |     name    "SHOULD"   //OR
        |       |     ordinal    1
        |       |--query    FuzzyQuery  (id=150)
        |          boost    1.0
        |          minimumSimilarity    0.5
        |          numberOfTerms    0
        |          prefixLength    0
                                    |                                rewriteMethod
MultiTermQuery$ScoringBooleanQueryRewrite  (id=152)
        |             term    Term  (id=153)
        |               field    "contents"
        |               text    "eat"
        |          termLongEnough    true
        |------[1]    BooleanClause  (id=149)
              |    occur    BooleanClause$Occur$2  (id=129)
              |     name    "SHOULD"   //OR
              |     ordinal    1
              |--query    TermQuery  (id=155)
                 boost    1.0
                          term    Term  (id=156)
                             field    "contents"
                             text    "foods"
          size    2
        disableCoord    false
        minNrShouldMatch    0
    size    3
```

```
disableCoord    false
minNrShouldMatch    0
```



对于 Query 对象有以下说明：

- BooleanQuery 即所有的子语句按照布尔关系合并
  - +也即 MUST 表示必须满足的语句
  - SHOULD 表示可以满足的，minNrShouldMatch 表示在 SHOULD 中必须满足的最小语句个数，默认是 0，也即既然是 SHOULD，也即或的关系，可以一个也不满足(当然没有 MUST 的时候除外)。
  - -也即 MUST_NOT 表示必须不能满足的语句
- 树的叶子节点中：
  - 最基本的是 TermQuery，也即表示一个词
  - 当然也可以是 PrefixQuery 和 FuzzyQuery，这些查询语句由于特殊的语法，可能对应的不是一个词，而是多个词，因而他们都有 rewriteMethod 对象指向 MultiTermQuery 的 Inner Class，表示对应多个词，在查询过程中会得到特殊处理。

2.4、搜索查询对象

代码为：

```
TopDocs docs = searcher.search(query, 50);
```

其最终调用 search(createWeight(query), filter, n);

索引过程包含以下子过程：

- 创建 weight 树，计算 term weight
- 创建 scorer 及 SumScorer 树，为合并倒排表做准备
- 用 SumScorer 进行倒排表合并
- 收集文档结果集合及计算打分

2.4.1、创建 Weight 对象树，计算 Term Weight

IndexSearcher(Searcher).createWeight(Query) 代码如下：

```
protected Weight createWeight(Query query) throws IOException {
  return query.weight(this);
}
```

BooleanQuery(Query).weight(Searcher) 代码为：

```
public Weight weight(Searcher searcher) throws IOException {
  //重写 Query 对象树
  Query query = searcher.rewrite(this);
  //创建 Weight 对象树
  Weight weight = query.createWeight(searcher);
  //计算 Term Weight 分数
  float sum = weight.sumOfSquaredWeights();
  float norm = getSimilarity(searcher).queryNorm(sum);
  weight.normalize(norm);
  return weight;
}
```

此过程又包含以下过程：

- 重写 Query 对象树
- 创建 Weight 对象树
- 计算 Term Weight 分数

2.4.1.1、重写 Query 对象树

从 BooleanQuery 的 rewrite 函数我们可以看出，重写过程也是一个递归的过程，一直到 Query 对象树的叶子节点。

```
BooleanQuery.rewrite(IndexReader) 代码如下：
BooleanQuery clone = null;
for (int i = 0 ; i < clauses.size(); i++) {
  BooleanClause c = clauses.get(i);
  //对每一个子语句的 Query 对象进行重写
  Query query = c.getQuery().rewrite(reader);
  if (query != c.getQuery()) {
    if (clone == null)
      clone = (BooleanQuery)this.clone();
    //重写后的 Query 对象加入复制的新 Query 对象树
    clone.clauses.set(i, new BooleanClause(query, c.getOccur()));
  }
}
if (clone != null) {
  return clone; //如果有子语句被重写，则返回复制的新 Query 对象树。
} else
  return this; //否则将老的 Query 对象树返回。
```

让我们把目光聚集到叶子节点上，叶子节点基本是两种，或是 TermQuery，或是 MultiTermQuery，从 Lucene 的源码可以看出 TermQuery 的 rewrite 函数就是返回对象本身，也即真正需要重写的是 MultiTermQuery，也即一个 Query 代表多个 Term 参与查询，如本例子中的 PrefixQuery 及 FuzzyQuery。

对此类的 Query，Lucene 不能够直接进行查询，必须进行重写处理：

- 首先，要从索引文件的词典中，把多个 Term 都找出来，比如"appl*"，我们在索引文件的词典中可以找到如下 Term："apple"，"apples"，"apply"，这些 Term 都要参与查询过程，而非原来的"appl*"参与查询过程，因为词典中根本就没有"appl*"。

- 然后，将取出的多个 Term 重新组织成新的 Query 对象进行查询，基本有两种方式：
  - 方式一：将多个 Term 看成一个 Term，将包含它们的文档号取出来放在一起 (DocId Set)，作为一个统一的倒排表来参与倒排表的合并。
  - 方式二：将多个 Term 组成一个 BooleanQuery，它们之间是 OR 的关系。

从上面的 Query 对象树中，我们可以看到，MultiTermQuery 都有一个 RewriteMethod 成员变量，就是用来重写 Query 对象的，有以下几种：

- ConstantScoreFilterRewrite 采取的是方式一，其 rewrite 函数实现如下：

```
public Query rewrite(IndexReader reader, MultiTermQuery query) {
  Query result = new ConstantScoreQuery(new
MultiTermQueryWrapperFilter<MultiTermQuery>(query));
  result.setBoost(query.getBoost());
  return result;
}
```

MultiTermQueryWrapperFilter 中的 getDocIdSet 函数实现如下：

```
public DocIdSet getDocIdSet(IndexReader reader) throws IOException {
  //得到 MultiTermQuery 的 Term 枚举器
  final TermEnum enumerator = query.getEnum(reader);
  try {
    if (enumerator.term() == null)
      return DocIdSet.EMPTY_DOCIDSET;
    //创建包含多个 Term 的文档号集合
    final OpenBitSet bitSet = new OpenBitSet(reader.maxDoc());
    final int[] docs = new int[32];
    final int[] freqs = new int[32];
    TermDocs termDocs = reader.termDocs();
    try {
      int termCount = 0;
    //一个循环，取出对应 MultiTermQuery 的所有的 Term，取出他们的文档号，加
入集合
```

```
      do {
        Term term = enumerator.term();
        if (term == null)
          break;
        termCount++;
        termDocs.seek(term);
        while (true) {
          final int count = termDocs.read(docs, freqs);
          if (count != 0) {
            for(int i=0;i<count;i++) {
              bitSet.set(docs[i]);
            }
          } else {
            break;
          }
        }
      } while (enumerator.next());
      query.incTotalNumberOfTerms(termCount);
    } finally {
      termDocs.close();
    }
    return bitSet;
  } finally {
    enumerator.close();
  }
}
```

- ScoringBooleanQueryRewrite 及其子类 ConstantScoreBooleanQueryRewrite 采取方式二，其 rewrite 函数代码如下：

```
public Query rewrite(IndexReader reader, MultiTermQuery query) throws IOException {
  //得到 MultiTermQuery 的 Term 枚举器
  FilteredTermEnum enumerator = query.getEnum(reader);
  BooleanQuery result = new BooleanQuery(true);
  int count = 0;
  try {
    //一个循环，取出对应 MultiTermQuery 的所有的 Term，加入 BooleanQuery
    do {
      Term t = enumerator.term();
      if (t != null) {
        TermQuery tq = new TermQuery(t);
        tq.setBoost(query.getBoost() * enumerator.difference());
        result.add(tq, BooleanClause.Occur.SHOULD);
        count++;
      }
    } while (enumerator.next());
  } finally {
    enumerator.close();
  }
  query.incTotalNumberOfTerms(count);
  return result;
}
```

- 以上两种方式各有优劣：
  - 方式一使得 MultiTermQuery 对应的所有的 Term 看成一个 Term，组成一个 docid set，作为统一的倒排表参与倒排表的合并，这样无论这样的 Term 在索引中有多少，都只会有一个倒排表参与合并，不会产生 TooManyClauses 异常，也使得性能得到提高。但是多个 Term 之间的 tf, idf 等差别将被忽略，所以采用方式二的 RewriteMethod 为 ConstantScoreXXX，也即除了用户指定的 Query boost，其他的打分计算全部忽略。
  - 方式二使得整个 Query 对象树被展开，叶子节点都为 TermQuery，MultiTermQuery 中的多个 Term 可根据在索引中的 tf, idf 等参与打分计算，然

而我们事先并不知道索引中和 MultiTermQuery 相对应的 Term 到底有多少个，因而会出现 TooManyClauses 异常，也即一个 BooleanQuery 中的子查询太多。这样会造成要合并的倒排表非常多，从而影响性能。

- Lucene 认为对于 MultiTermQuery 这种查询，打分计算忽略是很合理的，因为当用户输入"appl*"的时候，他并不知道索引中有什么与此相关，也并不偏爱其中之一，因而计算这些词之间的差别对用户来讲是没有意义的。从而 Lucene 对方式二也提供了 ConstantScoreXXX，来提高搜索过程的性能，从后面的例子来看，会影响文档打分，在实际的系统应用中，还是存在问题的。
- 为了兼顾上述两种方式，Lucene 提供了 ConstantScoreAutoRewrite，来根据不同的情况，选择不同的方式。

ConstantScoreAutoRewrite.rewrite 代码如下：

```
public Query rewrite(IndexReader reader, MultiTermQuery query) throws IOException {
  final Collection<Term> pendingTerms = new ArrayList<Term>();
  //计算文档数目限制，docCountPercent 默认为 0.1，也即索引文档总数的 0.1%
  final int docCountCutoff = (int) ((docCountPercent / 100.) * reader.maxDoc());
  //计算 Term 数目限制，默认为 350
  final int termCountLimit = Math.min(BooleanQuery.getMaxClauseCount(), termCountCutoff);
  int docVisitCount = 0;
  FilteredTermEnum enumerator = query.getEnum(reader);
  try {
   //一个循环，取出与 MultiTermQuery 相关的所有的 Term。
   while(true) {
    Term t = enumerator.term();
    if (t != null) {
     pendingTerms.add(t);
     docVisitCount += reader.docFreq(t);
    }
   //如果 Term 数目超限，或者文档数目超限，则可能非常影响倒排表合并的性能，因而选用方式一，也即 ConstantScoreFilterRewrite 的方式
    if (pendingTerms.size() >= termCountLimit || docVisitCount >= docCountCutoff) {
                Query result = new ConstantScoreQuery(new
```

```
MultiTermQueryWrapperFilter<MultiTermQuery>(query));
        result.setBoost(query.getBoost());
        return result;
      } else   if (!enumerator.next()) {
      //如果 Term 数目不太多，而且文档数目也不太多，不会影响倒排表合并的性能，因而选用方式二，也即 ConstantScoreBooleanQueryRewrite 的方式。
        BooleanQuery bq = new BooleanQuery(true);
        for (final Term term: pendingTerms) {
          TermQuery tq = new TermQuery(term);
          bq.add(tq, BooleanClause.Occur.SHOULD);
        }
        Query result = new ConstantScoreQuery(new QueryWrapperFilter(bq));
        result.setBoost(query.getBoost());
        query.incTotalNumberOfTerms(pendingTerms.size());
        return result;
      }
    }
  } finally {
    enumerator.close();
  }
}
```

从上面的叙述中，我们知道，在重写 Query 对象树的时候，从 MultiTermQuery 得到的 TermEnum 很重要，能够得到对应 MultiTermQuery 的所有的 Term，这是怎么做的的呢？MultiTermQuery 的 getEnum 返回的是 FilteredTermEnum，它有两个成员变量，其中 TermEnum actualEnum 是用来枚举索引中所有的 Term 的，而 Term currentTerm 指向的是当前满足条件的 Term，FilteredTermEnum 的 next()函数如下：

```
public boolean next() throws IOException {
    if (actualEnum == null) return false;
```

```
    currentTerm = null;
    //不断得到下一个索引中的 Term
    while (currentTerm == null) {
      if (endEnum()) return false;
      if (actualEnum.next()) {
        Term term = actualEnum.term();
        //如果当前索引中的 Term 满足条件，则赋值为当前的 Term
        if (termCompare(term)) {
          currentTerm = term;
          return true;
        }
      }
      else return false;
    }
    currentTerm = null;
    return false;
}
```

不同的 MultiTermQuery 的 termCompare 不同：

对于 PrefixQuery 的 getEnum(IndexReader reader)得到的是 PrefixTermEnum，其 termCompare 实现如下：

```
protected boolean termCompare(Term term) {
  //只要前缀相同，就满足条件
                if (term.field() == prefix.field() &&
term.text().startsWith(prefix.text())){
    return true;
  }
  endEnum = true;
  return false;
```

```
}
```

对于 FuzzyQuery 的 getEnum 得到的是 FuzzyTermEnum，其 termCompare 实现如下：

```
protected final boolean termCompare(Term term) {
  //对于 FuzzyQuery，其 prefix 设为空""，也即这一条件一定满足，只要计算的是similarity
  if (field == term.field() && term.text().startsWith(prefix)) {
    final String target = term.text().substring(prefix.length());
    this.similarity = similarity(target);
    return (similarity > minimumSimilarity);
  }
  endEnum = true;
  return false;
}
//计算 Levenshtein distance 也即 edit distance，对于两个字符串，从一个转换成为另一个所需要的最少基本操作(添加，删除，替换)数。
private synchronized final float similarity(final String target) {
    final int m = target.length();
    final int n = text.length();
    // init matrix d
    for (int i = 0; i<=n; ++i) {
      p[i] = i;
    }
    // start computing edit distance
    for (int j = 1; j<=m; ++j) { // iterates through target
      int bestPossibleEditDistance = m;
      final char t_j = target.charAt(j-1); // jth character of t
      d[0] = j;
      for (int i=1; i<=n; ++i) { // iterates through text
```

```
      // minimum of cell to the left+1, to the top+1, diagonally left and up +(0|1)
      if (t_j != text.charAt(i-1)) {
        d[i] = Math.min(Math.min(d[i-1], p[i]),   p[i-1]) + 1;
      } else {
        d[i] = Math.min(Math.min(d[i-1]+1, p[i]+1),   p[i-1]);
      }
      bestPossibleEditDistance = Math.min(bestPossibleEditDistance, d[i]);
    }
    // copy current distance counts to 'previous row' distance counts: swap p and d
    int _d[] = p;
    p = d;
    d = _d;
  }
  return 1.0f - ((float)p[n] / (float) (Math.min(n, m)));
 }
```

有关 edit distance 的算法详见 http://www.merriampark.com/ld.htm

计算两个字符串 s 和 t 的 edit distance 算法如下：

Step 1:

Set n to be the length of s.

Set m to be the length of t.

If n = 0, return m and exit.

If m = 0, return n and exit.

Construct a matrix containing 0..m rows and 0..n columns.

Step 2:

Initialize the first row to 0..n.

Initialize the first column to 0..m.

Step 3:

Examine each character of s (i from 1 to n).

Step 4:

Examine each character of t (j from 1 to m).

Step 5:

If s[i] equals t[j], the cost is 0.

If s[i] doesn't equal t[j], the cost is 1.

Step 6:

Set cell d[i,j] of the matrix equal to the minimum of:

a. The cell immediately above plus 1: d[i-1,j] + 1.

b. The cell immediately to the left plus 1: d[i,j-1] + 1.

c. The cell diagonally above and to the left plus the cost: d[i-1,j-1] + cost.

Step 7:

After the iteration steps (3, 4, 5, 6) are complete, the distance is found in cell d[n,m].

举例说明其过程如下：

比较的两个字符串为：“GUMBO”和 "GAMBOL".

比较的两个字符串为："GUMBO" 和 "GAMBOL".

Steps 1 and 2

		G	U	M	B	O
	0	1	2	3	4	5
G	1					
A	2					
M	3					
B	4					
O	5					
L	6					

Steps 3 to 6 When i = 1

		G	U	M	B	O
	0	1	2	3	4	5
G	1	0				
A	2	1				
M	3	2				
B	4	3				
O	5	4				
L	6	5				

Steps 3 to 6 When i = 2

		G	U	M	B	O
	0	1	2	3	4	5
G	1	0	1			
A	2	1	1			
M	3	2	2			
B	4	3	3			
O	5	4	4			
L	6	5	5			

Steps 3 to 6 When i = 3

		G	U	M	B	O
	0	1	2	3	4	5
G	1	0	1	2		
A	2	1	1	2		
M	3	2	2	1		
B	4	3	3	2		
O	5	4	4	3		
L	6	5	5	4		

Steps 3 to 6 When i = 4

		G	U	M	B	O
	0	1	2	3	4	5
G	1	0	1	2	3	
A	2	1	1	2	3	
M	3	2	2	1	2	
B	4	3	3	2	1	
O	5	4	4	3	2	
L	6	5	5	4	3	

Steps 3 to 6 When i = 5

		G	U	M	B	O
	0	1	2	3	4	5
G	1	0	1	2	3	4
A	2	1	1	2	3	4
M	3	2	2	1	2	3
B	4	3	3	2	1	2
O	5	4	4	3	2	1
L	6	5	5	4	3	2

下面做一个试验，来说明 ConstantScoreXXX 对评分的影响：

在索引中，添加了以下四篇文档：

file01.txt : apple other other other other

file02.txt : apple apple other other other

file03.txt : apple apple apple other other

file04.txt : apple apple apple other other

搜索"apple"结果如下：

docid : 3 score : 0.67974937

docid : 2 score : 0.58868027

docid : 1 score : 0.4806554

docid : 0 score : 0.33987468

文档按照包含"apple"的多少排序。

而搜索"apple*"结果如下：

docid : 0 score : 1.0

docid : 1 score : 1.0

docid : 2 score : 1.0

docid : 3 score : 1.0

也即 Lucene 放弃了对 score 的计算。

经过 rewrite，得到的新 Query 对象树如下：

```
query    BooleanQuery  (id=89)
  |  boost    1.0
  |  clauses    ArrayList<E>  (id=90)
  |    elementData    Object[3]  (id=97)
  |------[0]    BooleanClause  (id=99)
  |       |   occur    BooleanClause$Occur$1  (id=103)
  |       |     name    "MUST"
  |       |     ordinal    0
  |       |---query    BooleanQuery  (id=105)
  |           |  boost    1.0
  |           |  clauses    ArrayList<E>  (id=115)
  |           |    elementData    Object[2]  (id=120)
  |           |     //"apple*"被用方式一重写为 ConstantScoreQuery
  |           |---[0]    BooleanClause  (id=121)
  |           |     |    occur    BooleanClause$Occur$1  (id=103)
  |           |     |      name    "MUST"
  |           |     |      ordinal    0
  |           |     |---query    ConstantScoreQuery  (id=123)
  |           |           boost    1.0
  |           |           filter    MultiTermQueryWrapperFilter<Q>  (id=125)
```

```
|           |           query    PrefixQuery  (id=48)
|           |             boost    1.0
|           |             numberOfTerms    0
|           |                   prefix    Term  (id=127)
|           |                     field    "contents"
|           |                     text    "apple"
|           |             rewriteMethod    MultiTermQuery$1  (id=50)
|           |---[1]    BooleanClause  (id=122)
|              |    occur    BooleanClause$Occur$3  (id=111)
|              |      name    "MUST_NOT"
|              |      ordinal    2
|              |---query    TermQuery  (id=124)
|                   boost    1.0
|                   term    Term  (id=130)
|                             field    "contents"
|                             text    "boy"
|            modCount    0
|            size    2
|          disableCoord    false
|          minNrShouldMatch    0
|------[1]    BooleanClause  (id=101)
|       |   occur    BooleanClause$Occur$2  (id=108)
|       |     name    "SHOULD"
|       |     ordinal    1
|       |---query    BooleanQuery  (id=110)
|           |  boost    1.0
|           |  clauses    ArrayList<E>  (id=117)
|           |    elementData    Object[2]  (id=132)
```

```
|          |      //"cat*"被用方式一重写为 ConstantScoreQuery
|          |------[0]   BooleanClause  (id=133)
|          |       |  occur    BooleanClause$Occur$2  (id=108)
|          |       |    name   "SHOULD"
|          |       |    ordinal   1
|          |       |---query   ConstantScoreQuery  (id=135)
|          |           boost   1.0
|          |           filter   MultiTermQueryWrapperFilter<Q>  (id=137)
|          |            query    PrefixQuery  (id=63)
|          |              boost   1.0
|          |              numberOfTerms   0
|          |              prefix   Term  (id=138)
|          |                       field   "contents"
|          |                       text   "cat"
|          |              rewriteMethod   MultiTermQuery$1  (id=50)
|          |------[1]   BooleanClause  (id=134)
|               |  occur    BooleanClause$Occur$2  (id=108)
|               |    name   "SHOULD"
|               |    ordinal   1
|               |---query   TermQuery  (id=136)
|                   boost   1.0
|                   term   Term  (id=140)
|                          field   "contents"
|                          text   "dog"
|          modCount   0
|          size   2
|          disableCoord   false
|          minNrShouldMatch   0
```

|------**[2] BooleanClause** (id=102)

| occur BooleanClause$Occur$3 (id=111)

| **name "MUST_NOT"**

| ordinal 2

|---**query BooleanQuery** (id=113)

| boost 1.0

| clauses ArrayList<E> (id=119)

| elementData Object[2] (id=142)

|------**[0] BooleanClause** (id=143)

| | occur BooleanClause$Occur$2 (id=108)

| | **name** "**SHOULD**"

| | ordinal 1

| | **//"eat~"作为 FuzzyQuery，被重写成 BooleanQuery，索引中满足条件的 Term 有"eat"和"cat"。FuzzyQuery 不用上述的任何一种 RewriteMethod，而是用方式二自己实现了 rewrite 函数，是将同"eat"的 edit distance 最近的最多 maxClauseCount(默认 1024)个 Term 组成 BooleanQuery。**

| |---**query BooleanQuery** (id=145)

| | boost 1.0

| | clauses ArrayList<E> (id=146)

| | elementData Object[10] (id=147)

| |------**[0] BooleanClause** (id=148)

| | | occur BooleanClause$Occur$2 (id=108)

| | | **name "SHOULD"**

| | | ordinal 1

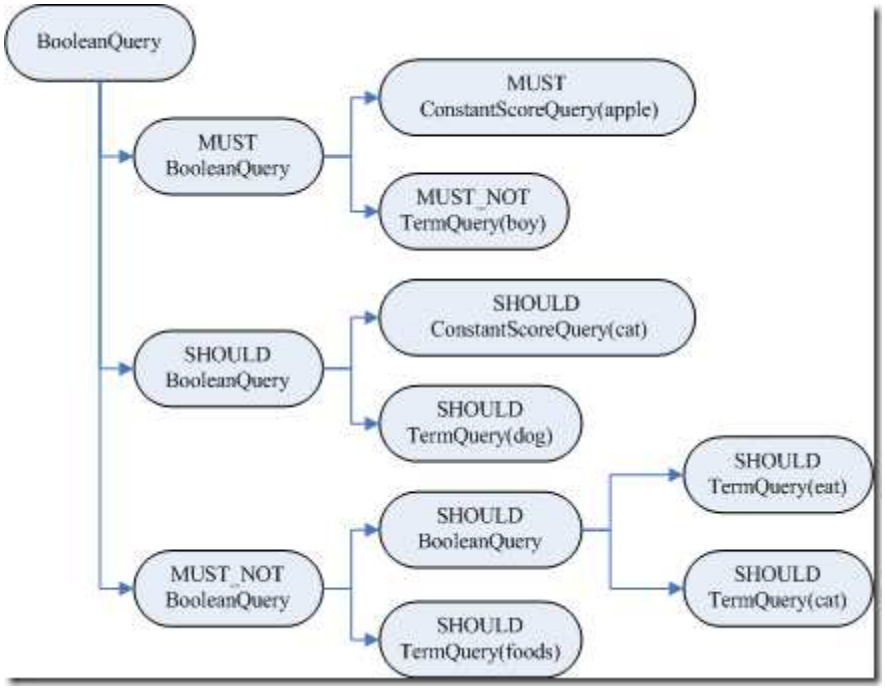
| | |---**query TermQuery** (id=150)

| | boost 1.0

| | **term Term (id=152)**

```
|           |                  field    "contents"
|           |                  text    "eat"
|           |------[1]    BooleanClause  (id=149)
|                 |   occur    BooleanClause$Occur$2  (id=108)
|                 |    name    "SHOULD"
|                 |    ordinal    1
|                 |---query    TermQuery  (id=151)
|                       boost    0.33333325
|                       term    Term  (id=153)
|                                  field    "contents"
|                                  text    "cat"
|           modCount    2
|           size    2
|         disableCoord    true
|         minNrShouldMatch    0
|------[1]    BooleanClause  (id=144)
       |   occur    BooleanClause$Occur$2  (id=108)
       |    name    "SHOULD"
       |    ordinal    1
       |---query    TermQuery  (id=154)
            boost    1.0
            term    Term  (id=155)
                        field    "contents"
                        text    "foods"
  modCount    0
  size    2
disableCoord    false
minNrShouldMatch    0
```

```
    modCount    0
    size    3
  disableCoord    false
  minNrShouldMatch    0
```



2.4.1.2、创建 Weight 对象树

BooleanQuery.createWeight(Searcher) 最终返回 return new BooleanWeight(searcher)，BooleanWeight 构造函数的具体实现如下：

```
public BooleanWeight(Searcher searcher) {
  this.similarity = getSimilarity(searcher);
  weights = new ArrayList<Weight>(clauses.size());
  //也是一个递归的过程，沿着新的 Query 对象树一直到叶子节点
  for (int i = 0 ; i < clauses.size(); i++) {
    weights.add(clauses.get(i).getQuery().createWeight(searcher));
  }
}
```

对于 TermQuery 的叶子节点，其 TermQuery.createWeight(Searcher) 返回 return new TermWeight(searcher)对象，TermWeight 构造函数如下：

```
public TermWeight(Searcher searcher) {
  this.similarity = getSimilarity(searcher);
  //此处计算了 idf
  idfExp = similarity.idfExplain(term, searcher);
  idf = idfExp.getIdf();
}
```

//idf 的计算完全符合文档中的公式：

$$idf(t) = 1 + \log(\frac{numDocs}{docFreq+1})$$

```
public IDFExplanation idfExplain(final Term term, final Searcher searcher) {
  final int df = searcher.docFreq(term);
  final int max = searcher.maxDoc();
  final float idf = idf(df, max);
  return new IDFExplanation() {
    public float getIdf() {
      return idf;
    }};
}
```

```
public float idf(int docFreq, int numDocs) {
  return (float)(Math.log(numDocs/(double)(docFreq+1)) + 1.0);
}
```

而 ConstantScoreQuery.createWeight(Searcher) 除了创建 ConstantScoreQuery.ConstantWeight(searcher)对象外，没有计算 idf。

由此创建的 Weight 对象树如下：

weight **BooleanQuery$BooleanWeight** (id=169)

| similarity DefaultSimilarity (id=177)

```
|   this$0    BooleanQuery   (id=89)
|   weights    ArrayList<E>   (id=188)
|     elementData    Object[3]   (id=190)
|------[0]    BooleanQuery$BooleanWeight   (id=171)
|       |   similarity    DefaultSimilarity   (id=177)
|       |   this$0    BooleanQuery   (id=105)
|       |   weights    ArrayList<E>   (id=193)
|       |     elementData    Object[2]   (id=199)
|       |------[0]    ConstantScoreQuery$ConstantWeight   (id=183)
|       |           queryNorm    0.0
|       |           queryWeight    0.0
|       |           similarity    DefaultSimilarity   (id=177)
|       |           //ConstantScore(contents:apple*)
|       |           this$0    ConstantScoreQuery   (id=123)
|       |------[1]    TermQuery$TermWeight   (id=175)
|                 idf    2.0986123
|                 idfExp    Similarity$1   (id=241)
|                 queryNorm    0.0
|                 queryWeight    0.0
|                 similarity    DefaultSimilarity   (id=177)
|                 //contents:boy
|                 this$0    TermQuery   (id=124)
|                 value    0.0
|             modCount    2
|             size    2
|------[1]    BooleanQuery$BooleanWeight   (id=179)
|       |   similarity    DefaultSimilarity   (id=177)
|       |   this$0    BooleanQuery   (id=110)
```

```
|       |   weights    ArrayList<E>  (id=195)
|       |     elementData    Object[2]  (id=204)
|       |------[0]    ConstantScoreQuery$ConstantWeight  (id=206)
|       |           queryNorm    0.0
|       |           queryWeight    0.0
|       |           similarity    DefaultSimilarity  (id=177)
|       |           //ConstantScore(contents:cat*)
|       |           this$0    ConstantScoreQuery  (id=135)
|       |------[1]    TermQuery$TermWeight  (id=207)
|                 idf    1.5389965
|                 idfExp    Similarity$1  (id=210)
|                 queryNorm    0.0
|                 queryWeight    0.0
|                 similarity    DefaultSimilarity  (id=177)
|                 //contents:dog
|                 this$0    TermQuery  (id=136)
|                 value    0.0
|           modCount    2
|           size    2
|------[2]    BooleanQuery$BooleanWeight  (id=182)
       |  similarity    DefaultSimilarity  (id=177)
       |  this$0    BooleanQuery  (id=113)
       |  weights    ArrayList<E>  (id=197)
       |    elementData    Object[2]  (id=216)
       |------[0]    BooleanQuery$BooleanWeight  (id=181)
       |       |    similarity    BooleanQuery$1  (id=220)
       |       |    this$0    BooleanQuery  (id=145)
       |       |    weights    ArrayList<E>  (id=221)
```

| | elementData Object[2] (id=224)

| |------**[0] TermQuery$TermWeight** (id=226)

| | idf 2.0986123

| | idfExp Similarity$1 (id=229)

| | queryNorm 0.0

| | queryWeight 0.0

| | similarity DefaultSimilarity (id=177)

| | **//contents:eat**

| | **this$0 TermQuery** (id=150)

| | value 0.0

| |------**[1] TermQuery$TermWeight** (id=227)

| idf 1.1823215

| idfExp Similarity$1 (id=231)

| queryNorm 0.0

| queryWeight 0.0

| similarity DefaultSimilarity (id=177)

| **//contents:cat^0.33333325**

| **this$0 TermQuery** (id=151)

| value 0.0

| modCount 2

| size 2

|------**[1] TermQuery$TermWeight** (id=218)

idf 2.0986123

idfExp Similarity$1 (id=233)

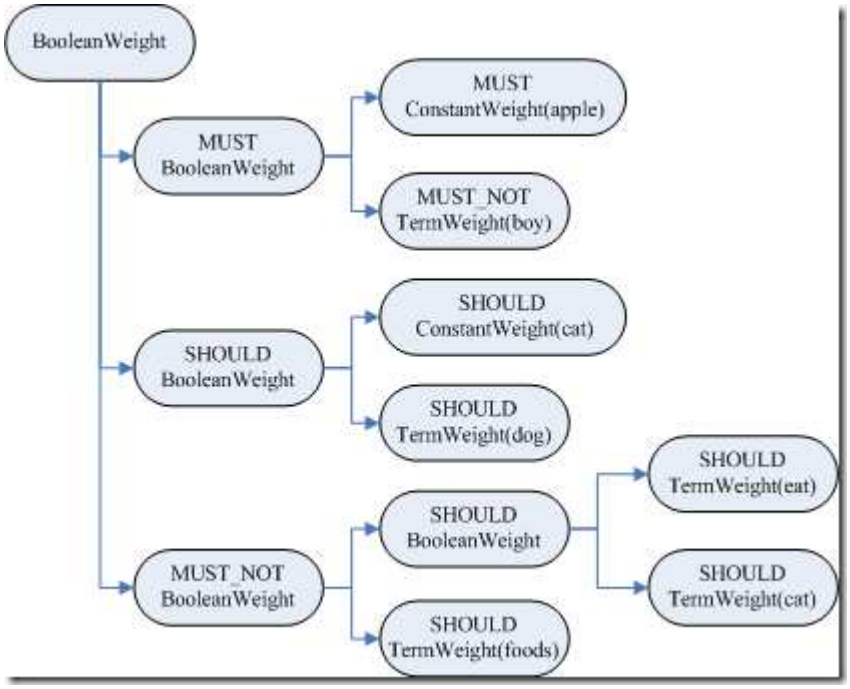
queryNorm 0.0

queryWeight 0.0

similarity DefaultSimilarity (id=177)

//contents:foods

```
            this$0    TermQuery  (id=154)
            value    0.0
        modCount    2
        size    2
    modCount    3
    size    3
```



2.4.1.3、计算 Term Weight 分数

(1) 首先计算 sumOfSquaredWeights

按照公式：

$$sumOfSquaredWeights = q.getBoost()^2 \times \sum_{t in q} (idf(t) \times t.getBoost())^2$$

代码如下：

```
float sum = weight.sumOfSquaredWeights();
```

```
//可以看出，也是一个递归的过程
public float sumOfSquaredWeights() throws IOException {
  float sum = 0.0f;
  for (int i = 0 ; i < weights.size(); i++) {
    float s = weights.get(i).sumOfSquaredWeights();
    if (!clauses.get(i).isProhibited())
      sum += s;
  }
  sum *= getBoost() * getBoost();  //乘以 query boost
  return sum ;
}
```

对于叶子节点 TermWeight 来讲，其 TermQuery$TermWeight.sumOfSquaredWeights()实现如下：

```
public float sumOfSquaredWeights() {
  //计算一部分打分，idf*t.getBoost()，将来还会用到。
  queryWeight = idf * getBoost();
  //计算(idf*t.getBoost())^2
  return queryWeight * queryWeight;
}
```

对于叶子节点 ConstantWeight 来讲，其

ConstantScoreQuery$ConstantWeight.sumOfSquaredWeights() 如下：

```
public float sumOfSquaredWeights() {
  //除了用户指定的 boost 以外，其他都不计算在打分内
  queryWeight = getBoost();
  return queryWeight * queryWeight;
}
```

(2) 计算 queryNorm

其公式如下：

$$queryNorm(q) = \frac{1}{\sqrt{sumOfSquaredWeights}}$$

其代码如下：

```
public float queryNorm(float sumOfSquaredWeights) {
  return (float)(1.0 / Math.sqrt(sumOfSquaredWeights));
}
```

(3) 将 queryNorm 算入打分

代码为：

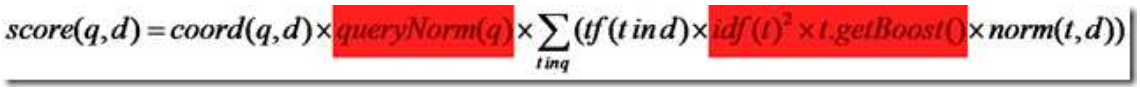
weight.normalize(norm);

```
//又是一个递归的过程
public void normalize(float norm) {
  norm *= getBoost();
  for (Weight w : weights) {
    w.normalize(norm);
  }
}
```

其叶子节点 TermWeight 来讲，其 TermQuery$TermWeight.normalize(float) 代码如下：

```
public void normalize(float queryNorm) {
  this.queryNorm = queryNorm;
  //原来 queryWeight 为 idf*t.getBoost()，现在为
queryNorm*idf*t.getBoost()。
  queryWeight *= queryNorm;
  //打分到此计算了 queryNorm*idf*t.getBoost()*idf =
queryNorm*idf^2*t.getBoost()部分。
  value = queryWeight * idf;
}
```

我们知道，Lucene 的打分公式整体如下，到此计算了图中，红色的部分：



2.4.2、创建 Scorer 及 SumScorer 对象树

当创建完 Weight 对象树的时候，调用 IndexSearcher.search(Weight, Filter, int)，代码如下：

```
//(a)创建文档号收集器
TopScoreDocCollector collector =
TopScoreDocCollector.create(nDocs, !weight.scoresDocsOutOfOrder());
search(weight, filter, collector);
//(b)返回搜索结果
return collector.topDocs();
```

```
public void search(Weight weight, Filter filter, Collector collector)
    throws IOException {
  if (filter == null) {
    for (int i = 0; i < subReaders.length; i++) {
      collector.setNextReader(subReaders[i], docStarts[i]);
      //(c)创建 Scorer 对象树，以及 SumScorer 树用来合并倒排表
      Scorer scorer = weight.scorer(subReaders[i], !collector.acceptsDocsOutOfOrder(), true);
      if (scorer != null) {
        //(d)合并倒排表，(e)收集文档号
        scorer.score(collector);
      }
    }
  } else {
    for (int i = 0; i < subReaders.length; i++) {
      collector.setNextReader(subReaders[i], docStarts[i]);
      searchWithFilter(subReaders[i], weight, filter, collector);
```

```
   }
  }
}
```

在本节中，重点分析**(c)创建 Scorer 对象树，以及 SumScorer 树用来合并倒排表，**在 2.4.3 节中，分析 **(d)合并倒排表，**在 2.4.4 节中，分析文档结果收集器的创建**(a)**，结果文档的收集**(e)**，以及文档的返回**(b)**。

BooleanQuery$BooleanWeight.scorer(IndexReader, boolean, boolean) 代码如下：

```
public Scorer scorer(IndexReader reader, boolean scoreDocsInOrder, boolean topScorer){
  //存放对应于 MUST 语句的 Scorer
  List<Scorer> required = new ArrayList<Scorer>();
  //存放对应于 MUST_NOT 语句的 Scorer
  List<Scorer> prohibited = new ArrayList<Scorer>();
  //存放对应于 SHOULD 语句的 Scorer
  List<Scorer> optional = new ArrayList<Scorer>();
  //遍历每一个子语句，生成子 Scorer 对象，并加入相应的集合，这是一个递归的过程。
  Iterator<BooleanClause> cIter = clauses.iterator();
  for (Weight w  : weights) {
    BooleanClause c =  cIter.next();
    Scorer subScorer = w.scorer(reader, true, false);
    if (subScorer == null) {
      if (c.isRequired()) {
        return null;
      }
    } else if (c.isRequired()) {
      required.add(subScorer);
    } else if (c.isProhibited()) {
      prohibited.add(subScorer);
    } else {
```

```
    optional.add(subScorer);
   }
  }
  //此处在有关 BooleanScorer 及 scoreDocsInOrder 一节会详细描述
  if (!scoreDocsInOrder && topScorer && required.size() == 0 && prohibited.size() < 32) {
    return new BooleanScorer(similarity, minNrShouldMatch, optional, prohibited);
  }
  //生成 Scorer 对象树，同时生成 SumScorer 对象树
  return new BooleanScorer2(similarity, minNrShouldMatch, required, prohibited, optional);
}
```

对其叶子节点 TermWeight 来说，TermQuery$TermWeight.scorer(IndexReader, boolean, boolean)

代码如下：

```
public Scorer scorer(IndexReader reader, boolean scoreDocsInOrder, boolean topScorer) throws
IOException {
  //此 Term 的倒排表
  TermDocs termDocs = reader.termDocs(term);
  if (termDocs == null)
    return null;
  return new TermScorer(this, termDocs, similarity, reader.norms(term.field()));
}
```

```
TermScorer(Weight weight, TermDocs td, Similarity similarity, byte[] norms) {
  super(similarity);
  this.weight = weight;
  this.termDocs = td;
  //得到标准化因子
  this.norms = norms;
  //得到原来计算得的打分：queryNorm*idf^2*t.getBoost()
  this.weightValue = weight.getValue();
```

```
  for (int i = 0; i < SCORE_CACHE_SIZE; i++)
    scoreCache[i] = getSimilarity().tf(i) * weightValue;
}
```

对其叶子节点 ConstantWeight 来说，ConstantScoreQuery$ConstantWeight.scorer(IndexReader, boolean, boolean) 代码如下：

```
public ConstantScorer(Similarity similarity, IndexReader reader, Weight w) {
  super(similarity);
  theScore = w.getValue();
  //得到所有的文档号，形成统一的倒排表，参与倒排表合并。
  DocIdSet docIdSet = filter.getDocIdSet(reader);
  DocIdSetIterator docIdSetIterator = docIdSet.iterator();
}
```

对于 BooleanWeight，最后要产生的是 BooleanScorer2，其构造函数代码如下：

```
public BooleanScorer2(Similarity similarity, int minNrShouldMatch,
    List<Scorer> required, List<Scorer> prohibited, List<Scorer> optional) {
  super(similarity);
  //为了计算打分公式中的 coord 项做统计
  coordinator = new Coordinator();
  this.minNrShouldMatch = minNrShouldMatch;
  //SHOULD 的部分
  optionalScorers = optional;
  coordinator.maxCoord += optional.size();
  //MUST 的部分
  requiredScorers = required;
  coordinator.maxCoord += required.size();
  //MUST_NOT 的部分
  prohibitedScorers = prohibited;
  //事先计算好各种情况的 coord 值
```

```
  coordinator.init();
  //创建 SumScorer 为倒排表合并做准备
  countingSumScorer = makeCountingSumScorer();
}
```

```
Coordinator.init() {
  coordFactors = new float[maxCoord + 1];
  Similarity sim = getSimilarity();
  for (int i = 0; i <= maxCoord; i++) {
   //计算总的子语句的个数和一个文档满足的子语句的个数之间的关系，自然是一篇文档满足的子语句个个数越多，打分越高。
    coordFactors[i] = sim.coord(i, maxCoord);
  }
}
```

在生成 Scorer 对象树之外，还会生成 SumScorer 对象树，来表示各个语句之间的关系，为合并倒排表做准备。

在解析 BooleanScorer2.makeCountingSumScorer() 之前，我们先来看不同的语句之间都存在什么样的关系，又将如何影响倒排表合并呢？

语句主要分三类：MUST，SHOULD，MUST_NOT

语句之间的组合主要有以下几种情况：

- 多个 MUST，如"(+apple +boy +dog)"，则会生成 ConjunctionScorer(Conjunction 交集)，也即倒排表取交集
- MUST 和 SHOULD，如"(+apple boy)"，则会生成 ReqOptSumScorer(required optional)，也即 MUST 的倒排表返回，如果文档包括 SHOULD 的部分，则增加打分。
- MUST 和 MUST_NOT，如"(+apple –boy)"，则会生成 ReqExclScorer(required exclusive)，也即返回 MUST 的倒排表，但扣除 MUST_NOT 的倒排表中的文档。
- 多个 SHOULD，如"(apple boy dog)"，则会生成 DisjunctionSumScorer(Disjunction 并集)，也即倒排表去并集
- SHOULD 和 MUST_NOT，如"(apple –boy)"，则 SHOULD 被认为成 MUST，会生成

ReqExclScorer

- MUST，SHOULD，MUST_NOT 同时出现，则 MUST 首先和 MUST_NOT 组合成 ReqExclScorer，SHOULD 单独成为 SingleMatchScorer，然后两者组合成 ReqOptSumScorer。

下面分析生成 SumScorer 的过程：

BooleanScorer2.makeCountingSumScorer() 分两种情况：

- 当有 MUST 的语句的时候，则调用 makeCountingSumScorerSomeReq()
- 当没有 MUST 的语句的时候，则调用 makeCountingSumScorerNoReq()

首先来看 makeCountingSumScorerSomeReq 代码如下：

```
private Scorer makeCountingSumScorerSomeReq() {
  if (optionalScorers.size() == minNrShouldMatch) {
   //如果 optional 的语句个数恰好等于最少需满足的 optional 的个数，则所有的 optional 都变成 required。于是首先所有的 optional 生成 ConjunctionScorer(交集)，然后再通过 addProhibitedScorers 将 prohibited 加入，生成 ReqExclScorer(required exclusive)
    ArrayList<Scorer> allReq = new ArrayList<Scorer>(requiredScorers);
    allReq.addAll(optionalScorers);
    return addProhibitedScorers(countingConjunctionSumScorer(allReq));
  } else {
   //首先所有的 required 的语句生成 ConjunctionScorer(交集)
    Scorer requiredCountingSumScorer =
        requiredScorers.size() == 1
        ? new SingleMatchScorer(requiredScorers.get(0))
        : countingConjunctionSumScorer(requiredScorers);
    if (minNrShouldMatch > 0) {
   //如果最少需满足的 optional 的个数有一定的限制，则意味着 optional 中有一部分要相当于 required，会影响倒排表的合并。因而 required 生成的 ConjunctionScorer(交集)和 optional 生成的 DisjunctionSumScorer(并集)共同组合成一个 ConjunctionScorer(交集)，然后再加入 prohibited，生成
```

ReqExclScorer

```
    return addProhibitedScorers(
            dualConjunctionSumScorer(
                requiredCountingSumScorer,
                countingDisjunctionSumScorer(
                    optionalScorers,
                    minNrShouldMatch)));
  } else { // minNrShouldMatch == 0
```

//如果最少需满足的 optional 的个数没有一定的限制，则 optional 并不影响倒排表的合并，仅仅在文档包含 optional 部分的时候增加打分。所以 required 和 prohibited 首先生成 ReqExclScorer，然后再加入 optional，生成 ReqOptSumScorer(required optional)

```
    return new ReqOptSumScorer(
            addProhibitedScorers(requiredCountingSumScorer),
            optionalScorers.size() == 1
             ? new SingleMatchScorer(optionalScorers.get(0))
             : countingDisjunctionSumScorer(optionalScorers, 1));
  }
 }
}
```

然后我们来看 makeCountingSumScorerNoReq 代码如下：

```
private Scorer makeCountingSumScorerNoReq() {
  // minNrShouldMatch optional scorers are required, but at least 1
  int nrOptRequired = (minNrShouldMatch < 1) ? 1 : minNrShouldMatch;
  Scorer requiredCountingSumScorer;
  if (optionalScorers.size() > nrOptRequired)
```

//如果 optional 的语句个数多于最少需满足的 optional 的个数，则 optional 中一部分相当 required，影响倒排表的合并，所以生成 DisjunctionSumScorer

```
    requiredCountingSumScorer = countingDisjunctionSumScorer(optionalScorers, nrOptRequired);
  else if (optionalScorers.size() == 1)
    //如果 optional 的语句只有一个，则返回 SingleMatchScorer，不存在倒排表合并的问题。
    requiredCountingSumScorer = new SingleMatchScorer(optionalScorers.get(0));
  else
    //如果 optional 的语句个数少于等于最少需满足的 optional 的个数，则所有的 optional 都算 required，所以生成 ConjunctionScorer
    requiredCountingSumScorer = countingConjunctionSumScorer(optionalScorers);
  //将 prohibited 加入，生成 ReqExclScorer
  return addProhibitedScorers(requiredCountingSumScorer);
}
```

经过此步骤，生成的 Scorer 对象树如下：

```
scorer    BooleanScorer2  (id=50)
  |   coordinator    BooleanScorer2$Coordinator   (id=53)
  |   countingSumScorer    ReqOptSumScorer   (id=54)
  |   minNrShouldMatch    0
  |---optionalScorers    ArrayList<E>   (id=55)
  |     |   elementData    Object[10]   (id=69)
  |     |---[0]    BooleanScorer2   (id=73)
  |         |   coordinator    BooleanScorer2$Coordinator   (id=74)
  |         |   countingSumScorer    BooleanScorer2$1   (id=75)
  |         |   minNrShouldMatch    0
  |         |---optionalScorers    ArrayList<E>   (id=76)
  |         |     |   elementData    Object[10]   (id=83)
  |         |     |---[0]    ConstantScoreQuery$ConstantScorer   (id=86)
  |         |     |     docIdSetIterator    OpenBitSetIterator   (id=88)
  |         |     |     similarity    DefaultSimilarity   (id=64)
```

| | | theScore 0.47844642

| | | **//ConstantScore(contents:cat*)**

| | | **this$0 ConstantScoreQuery** (id=90)

| | |---**[1] TermScorer** (id=87)

| | doc -1

| | doc 0

| | docs int[32] (id=93)

| | freqs int[32] (id=95)

| | norms byte[4] (id=96)

| | pointer 0

| | pointerMax 2

| | scoreCache float[32] (id=98)

| | similarity DefaultSimilarity (id=64)

| | termDocs SegmentTermDocs (id=103)

| | **//weight(contents:dog)**

| | **weight TermQuery$TermWeight** (id=106)

| | weightValue 1.1332052

| | modCount 2

| | size 2

| |---**prohibitedScorers** ArrayList<E> (id=77)

| | elementData Object[10] (id=84)

| | size 0

| |---**requiredScorers** ArrayList<E> (id=78)

| elementData Object[10] (id=85)

| size 0

| similarity DefaultSimilarity (id=64)

| size 1

|---**prohibitedScorers** ArrayList<E> (id=60)

```
|   | elementData   Object[10]  (id=71)
|   |---[0]   BooleanScorer2  (id=81)
|       | coordinator   BooleanScorer2$Coordinator  (id=114)
|       | countingSumScorer   BooleanScorer2$1  (id=115)
|       | minNrShouldMatch   0
|       |---optionalScorers   ArrayList<E>  (id=116)
|       |   | elementData   Object[10]  (id=119)
|       |   |---[0]   BooleanScorer2  (id=122)
|       |   |   | coordinator   BooleanScorer2$Coordinator  (id=124)
|       |   |   | countingSumScorer   BooleanScorer2$1  (id=125)
|       |   |   | minNrShouldMatch   0
|       |   |   |---optionalScorers   ArrayList<E>  (id=126)
|       |   |   |   | elementData   Object[10]  (id=138)
|       |   |   |   |---[0]   TermScorer  (id=156)
|       |   |   |   |   docs   int[32]  (id=162)
|       |   |   |   |   freqs   int[32]  (id=163)
|       |   |   |   |   norms   byte[4]  (id=96)
|       |   |   |   |   pointer   0
|       |   |   |   |   pointerMax   1
|       |   |   |   |   scoreCache   float[32]  (id=164)
|       |   |   |   |   similarity   DefaultSimilarity  (id=64)
|       |   |   |   |   termDocs   SegmentTermDocs  (id=165)
|       |   |   |   |   //weight(contents:eat)
|       |   |   |   |   weight   TermQuery$TermWeight  (id=166)
|       |   |   |   |   weightValue   2.107161
|       |   |   |   |---[1]   TermScorer  (id=157)
|       |   |   |       doc   -1
|       |   |   |       doc   1
```

| | | | docs int[32] (id=171)

| | | | freqs int[32] (id=172)

| | | | norms byte[4] (id=96)

| | | | pointer 1

| | | | pointerMax 3

| | | | scoreCache float[32] (id=173)

| | | | similarity DefaultSimilarity (id=64)

| | | | termDocs SegmentTermDocs (id=180)

| | | | **//weight(contents:cat^0.33333325)**

| | | | **weight TermQuery$TermWeight** (id=181)

| | | | weightValue 0.22293752

| | | | size 2

| | | |---**prohibitedScorers** ArrayList<E> (id=127)

| | | | elementData Object[10] (id=140)

| | | | modCount 0

| | | | size 0

| | | |---**requiredScorers** ArrayList<E> (id=128)

| | | elementData Object[10] (id=142)

| | | modCount 0

| | | size 0

| | | similarity BooleanQuery$1 (id=129)

| | |---**[1] TermScorer** (id=123)

| | doc -1

| | doc 3

| | docs int[32] (id=131)

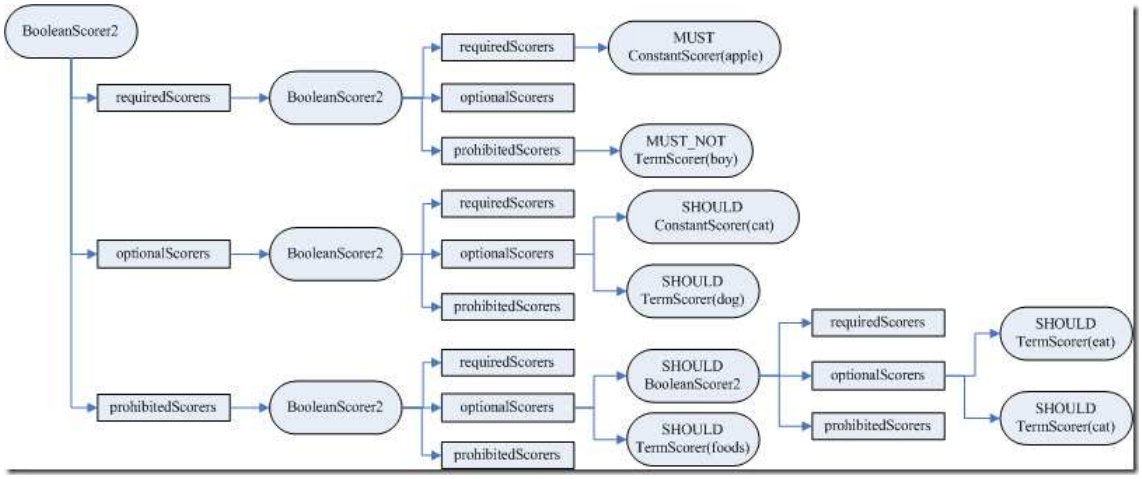
| | freqs int[32] (id=132)

| | norms byte[4] (id=96)

| | pointer 0

```
|          |          pointerMax    1
|          |          scoreCache    float[32]   (id=133)
|          |          similarity    DefaultSimilarity   (id=64)
|          |          termDocs    SegmentTermDocs   (id=134)
|          |          //weight(contents:foods)
|          |          weight    TermQuery$TermWeight   (id=135)
|          |          weightValue    2.107161
|          |       size    2
|          |---prohibitedScorers    ArrayList<E>   (id=117)
|          |      elementData    Object[10]   (id=120)
|          |      size    0
|          |---requiredScorers    ArrayList<E>   (id=118)
|               elementData    Object[10]   (id=121)
|               size    0
|         similarity    DefaultSimilarity   (id=64)
|    size    1
|---requiredScorers    ArrayList<E>   (id=63)
     |  elementData    Object[10]   (id=72)
     |---[0]    BooleanScorer2   (id=82)
          |    coordinator    BooleanScorer2$Coordinator   (id=183)
          |    countingSumScorer    ReqExclScorer   (id=184)
          |    minNrShouldMatch    0
          |---optionalScorers    ArrayList<E>   (id=185)
          |      elementData    Object[10]   (id=189)
          |      size    0
          |---prohibitedScorers    ArrayList<E>   (id=186)
          |      |  elementData    Object[10]   (id=191)
          |      |---[0]    TermScorer   (id=195)
```

```
|           docs    int[32]   (id=197)
|           freqs    int[32]   (id=198)
|           norms    byte[4]   (id=96)
|           pointer    0
|           pointerMax    0
|           scoreCache    float[32]   (id=199)
|           similarity    DefaultSimilarity   (id=64)
|           termDocs    SegmentTermDocs   (id=200)
|           //weight(contents:boy)
|           weight    TermQuery$TermWeight   (id=201)
|           weightValue    2.107161
|       size    1
|---requiredScorers    ArrayList<E>   (id=187)
    |   elementData    Object[10]   (id=193)
    |---[0]    ConstantScoreQuery$ConstantScorer   (id=203)
        docIdSetIterator    OpenBitSetIterator   (id=206)
        similarity    DefaultSimilarity   (id=64)
        theScore    0.47844642
        //ConstantScore(contents:apple*)
        this$0    ConstantScoreQuery   (id=207)
  size    1
similarity    DefaultSimilarity   (id=64)
size    1
similarity    DefaultSimilarity   (id=64)
```



生成的 SumScorer 对象树如下：

```
scorer    BooleanScorer2  (id=50)
  |    coordinator    BooleanScorer2$Coordinator  (id=53)
  |---countingSumScorer    ReqOptSumScorer  (id=54)
        |---optScorer    BooleanScorer2$SingleMatchScorer  (id=79)
        |      |    lastDocScore    NaN
        |      |    lastScoredDoc    -1
        |      |---scorer    BooleanScorer2  (id=73)
        |             |    coordinator    BooleanScorer2$Coordinator  (id=74)
        |             |---countingSumScorer    BooleanScorer2$1(DisjunctionSumScorer) (id=75)
        |                    |    currentDoc    -1
        |                    |    currentScore    NaN
        |                    |    doc    -1
        |                    |    lastDocScore    NaN
        |                    |    lastScoredDoc    -1
        |                    |    minimumNrMatchers    1
        |                    |    nrMatchers    -1
        |                    |    nrScorers    2
        |                    |    scorerDocQueue    ScorerDocQueue  (id=243)
```

```
|                |  similarity   null
|                |---subScorers   ArrayList<E>  (id=76)
|                     |  elementData   Object[10]  (id=83)
|                     |---[0]   ConstantScoreQuery$ConstantScorer  (id=86)
|                     |     doc   -1
|                     |     doc   -1
|                     |     docIdSetIterator   OpenBitSetIterator  (id=88)
|                     |     similarity   DefaultSimilarity  (id=64)
|                     |     theScore   0.47844642
|                     |     //ConstantScore(contents:cat*)
|                     |     this$0   ConstantScoreQuery  (id=90)
|                     |---[1]   TermScorer  (id=87)
|                           doc   -1
|                           doc   0
|                           docs   int[32]  (id=93)
|                           freqs   int[32]  (id=95)
|                           norms   byte[4]  (id=96)
|                           pointer   0
|                           pointerMax   2
|                           scoreCache   float[32]  (id=98)
|                           similarity   DefaultSimilarity  (id=64)
|                           termDocs   SegmentTermDocs  (id=103)
|                           //weight(contents:dog)
|                           weight   TermQuery$TermWeight  (id=106)
|                           weightValue   1.1332052
|          size   2
|        this$0   BooleanScorer2  (id=73)
|      minNrShouldMatch   0
```

```
|       optionalScorers    ArrayList<E>  (id=76)
|       prohibitedScorers    ArrayList<E>  (id=77)
|       requiredScorers    ArrayList<E>  (id=78)
|       similarity    DefaultSimilarity  (id=64)
|   similarity    DefaultSimilarity  (id=64)
|   this$0    BooleanScorer2  (id=50)
|---reqScorer    ReqExclScorer  (id=80)
    |---exclDisi    BooleanScorer2  (id=81)
    |       |   coordinator    BooleanScorer2$Coordinator  (id=114)
    |       |---countingSumScorer    BooleanScorer2$1(DisjunctionSumScorer) (id=11
    |           |   currentDoc    -1
    |           |   currentScore    NaN
    |           |   doc    -1
    |           |   lastDocScore    NaN
    |           |   lastScoredDoc    -1
    |           |   minimumNrMatchers    1
    |           |   nrMatchers    -1
    |           |   nrScorers    2
    |           |   scorerDocQueue    ScorerDocQueue  (id=260)
    |           |   similarity    null
    |           |---subScorers    ArrayList<E>  (id=116)
    |               |   elementData    Object[10]  (id=119)
    |               |---[0]    BooleanScorer2  (id=122)
    |               |       |   coordinator    BooleanScorer2$Coordinator  (id=124)
    |               |       |---countingSumScorer    BooleanScorer2$1(DisjunctionSum
(id=125)
    |               |           |   currentDoc    0
    |               |           |   currentScore    0.11146876
```

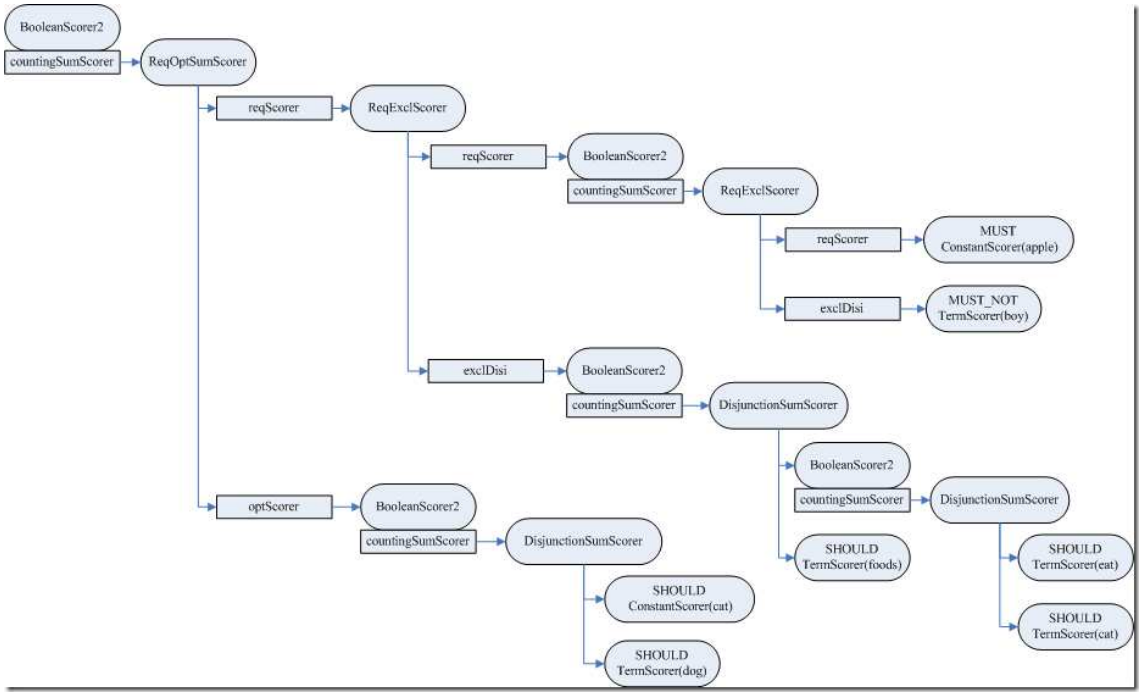
```
|               |           |   doc    -1
|               |           |   lastDocScore    NaN
|               |           |   lastScoredDoc    -1
|               |           |   minimumNrMatchers    1
|               |           |   nrMatchers    1
|               |           |   nrScorers    2
|               |           |   scorerDocQueue    ScorerDocQueue  (id=270)
|               |           |   similarity    null
|               |           |---subScorers    ArrayList<E>  (id=126)
|               |           |       |   elementData    Object[10]  (id=138)
|               |           |       |---[0]    TermScorer  (id=156)
|               |           |       |       doc    -1
|               |           |       |       doc    2
|               |           |       |       docs    int[32]  (id=162)
|               |           |       |       freqs    int[32]  (id=163)
|               |           |       |       norms    byte[4]  (id=96)
|               |           |       |       pointer    0
|               |           |       |       pointerMax    1
|               |           |       |       scoreCache    float[32]  (id=164)
|               |           |       |       similarity    DefaultSimilarity  (id=64)
|               |           |       |       termDocs    SegmentTermDocs  (id=165)
|               |           |       |       //weight(contents:eat)
|               |           |       |       weight    TermQuery$TermWeight  (id=166)
|               |           |       |       weightValue    2.107161
|               |           |       |---[1]    TermScorer  (id=157)
|               |           |               doc    -1
|               |           |               doc    1
|               |           |               docs    int[32]  (id=171)
```

```
|                |                    freqs    int[32]  (id=172)
|                |                    norms    byte[4]  (id=96)
|                |                    pointer    1
|                |                    pointerMax    3
|                |                    scoreCache    float[32]  (id=173)
|                |                    similarity    DefaultSimilarity  (id=64)
|                |                    termDocs    SegmentTermDocs  (id=180)
|                |                    //weight(contents:cat^0.33333325)
|                |                    weight    TermQuery$TermWeight  (id=181)
|                |                    weightValue    0.22293752
|                |                size    2
|                |            this$0    BooleanScorer2  (id=122)
|                |        doc    -1
|                |        doc    0
|                |        minNrShouldMatch    0
|                |        optionalScorers    ArrayList<E>  (id=126)
|                |        prohibitedScorers    ArrayList<E>  (id=127)
|                |        requiredScorers    ArrayList<E>  (id=128)
|                |        similarity    BooleanQuery$1  (id=129)
|                |---[1]    TermScorer  (id=123)
|                    doc    -1
|                    doc    3
|                    docs    int[32]  (id=131)
|                    freqs    int[32]  (id=132)
|                    norms    byte[4]  (id=96)
|                    pointer    0
|                    pointerMax    1
|                    scoreCache    float[32]  (id=133)
```

```
|                    similarity    DefaultSimilarity  (id=64)
|                    termDocs    SegmentTermDocs  (id=134)
|                    //weight(contents:foods)
|                    weight    TermQuery$TermWeight  (id=135)
|                    weightValue    2.107161
|                size    2
|            this$0    BooleanScorer2  (id=81)
|        doc    -1
|        doc    -1
|        minNrShouldMatch    0
|        optionalScorers    ArrayList<E>  (id=116)
|        prohibitedScorers    ArrayList<E>  (id=117)
|        requiredScorers    ArrayList<E>  (id=118)
|        similarity    DefaultSimilarity  (id=64)
|---reqScorer    BooleanScorer2$SingleMatchScorer  (id=237)
    |    doc    -1
    |    lastDocScore    NaN
    |    lastScoredDoc    -1
    |---scorer    BooleanScorer2  (id=82)
        |    coordinator    BooleanScorer2$Coordinator  (id=183)
        |---countingSumScorer    ReqExclScorer  (id=184)
            |---exclDisi    TermScorer  (id=195)
            |    doc    -1
            |    doc    -1
            |    docs    int[32]  (id=197)
            |    freqs    int[32]  (id=198)
            |    norms    byte[4]  (id=96)
            |    pointer    0
```

```
|       pointerMax    0
|       scoreCache    float[32]   (id=199)
|       similarity    DefaultSimilarity   (id=64)
|       termDocs    SegmentTermDocs   (id=200)
|       //weight(contents:boy)
|       weight    TermQuery$TermWeight   (id=201)
|       weightValue    2.107161
|---reqScorer    BooleanScorer2$2(ConjunctionScorer)   (id=281)
      |    coord    1.0
      |    doc    -1
      |    lastDoc    -1
      |    lastDocScore    NaN
      |    lastScoredDoc    -1
      |---scorers    Scorer[1]   (id=283)
          |---[0]    ConstantScoreQuery$ConstantScorer   (id=203)
                doc    -1
                doc    -1
                docIdSetIterator    OpenBitSetIterator   (id=206)
                similarity    DefaultSimilarity   (id=64)
                theScore    0.47844642
                //ConstantScore(contents:apple*)
                this$0    ConstantScoreQuery   (id=207)
          similarity    DefaultSimilarity   (id=64)
          this$0    BooleanScorer2   (id=82)
          val$requiredNrMatchers    1
      similarity    null
minNrShouldMatch    0
optionalScorers    ArrayList<E>   (id=185)
```

```
                    prohibitedScorers    ArrayList<E>  (id=186)
                    requiredScorers    ArrayList<E>  (id=187)
                    similarity    DefaultSimilarity  (id=64)
                similarity    DefaultSimilarity  (id=64)
                this$0    BooleanScorer2  (id=50)
            similarity    null
        similarity    null
    minNrShouldMatch    0
    optionalScorers    ArrayList<E>  (id=55)
    prohibitedScorers    ArrayList<E>  (id=60)
    requiredScorers    ArrayList<E>  (id=63)
    similarity    DefaultSimilarity  (id=64)
```



2.4.3、进行倒排表合并

在得到了 Scorer 对象树以及 SumScorer 对象树后，便是倒排表的合并以及打分计算的过程。合并倒排表在此节中进行分析，而 Scorer 对象树来进行打分的计算则在下一节分析。

BooleanScorer2.score(Collector) 代码如下：

```
public void score(Collector collector) throws IOException {
  collector.setScorer(this);
  while ((doc = countingSumScorer.nextDoc()) != NO_MORE_DOCS) {
    collector.collect(doc);
  }
}
```

从代码我们可以看出，此过程就是不断的取下一篇文档号，然后加入文档结果集。

取下一篇文档的过程，就是合并倒排表的过程，也就是对多个查询条件进行综合考虑后的下一篇文档的编号。

由于 SumScorer 是一棵树，因而合并倒排表也是按照树的结构进行的，先合并子树，然后子树与子树再进行合并，直到根。

按照上一节的分析，倒排表的合并主要用了以下几个 SumScorer:

- 交集 ConjunctionScorer
- 并集 DisjunctionSumScorer
- 差集 ReqExclScorer
- ReqOptSumScorer

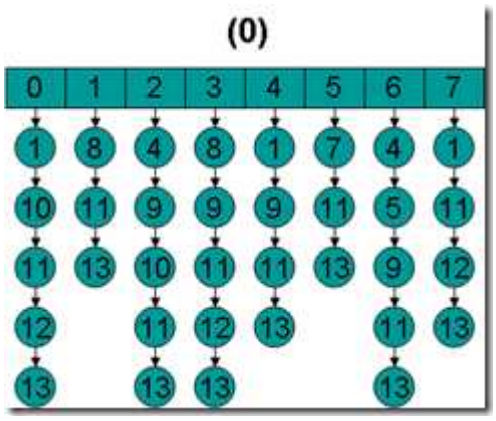
下面我们一一分析：

2.4.3.1、交集 ConjunctionScorer(+A +B)

ConjunctionScorer 中有成员变量 Scorer[] scorers，是一个 Scorer 的数组，每一项代表一个倒排表，ConjunctionScorer 就是对这些倒排表取交集，然后将交集中的文档号在 nextDoc()函数中依次返回。

为了描述清楚此过程，下面举一个具体的例子来解释倒排表合并的过程：

（1）倒排表最初如下：



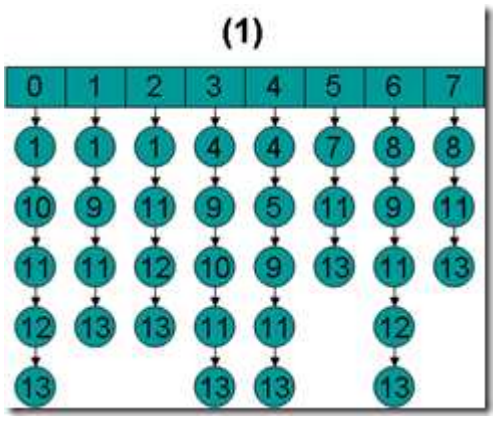
(2) 在 ConjunctionScorer 的构造函数中，首先调用每个 Scorer 的 nextDoc()函数，使得每个 Scorer 得到自己的第一篇文档号。

```
for (int i = 0; i < scorers.length; i++) {
  if (scorers[i].nextDoc() == NO_MORE_DOCS) {
   //由于是取交集，因而任何一个倒排表没有文档，交集就为空。
    lastDoc = NO_MORE_DOCS;
    return;
  }
}
```

(3) 在 ConjunctionScorer 的构造函数中，将 Scorer 按照第一篇的文档号从小到大进行排列。

```
Arrays.sort(scorers, new Comparator<Scorer>() {
  public int compare(Scorer o1, Scorer o2) {
    return o1.docID() - o2.docID();
  }
});
```

倒排表如下：



(4) 在 ConjunctionScorer 的构造函数中，第一次调用 doNext()函数。

```
if (doNext() == NO_MORE_DOCS) {
  lastDoc = NO_MORE_DOCS;
  return;
}
```

```
private int doNext() throws IOException {
  int first = 0;
  int doc = scorers[scorers.length - 1].docID();
  Scorer firstScorer;
  while ((firstScorer = scorers[first]).docID() < doc) {
    doc = firstScorer.advance(doc);
    first = first == scorers.length - 1 ? 0 : first + 1;
  }
  return doc;
}
```

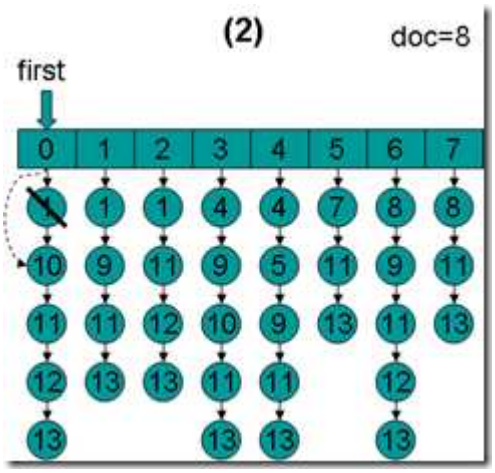
姑且我们称拥有最小文档号的倒排表称为 first，其实从 doNext()函数中的 first = first == scorers.length - 1 ? 0 : first + 1;我们可以看出，在处理过程中，Scorer 数组被看成一个循环数组(Ring)。

而此时 scorer[scorers.length - 1]拥有最大的文档号，doNext()中的循环，将所有的小于当前数组中最大文档号的文档全部用 firstScorer.advance(doc)(其跳到大于或等于 doc 的文档)函数跳过，因为既然它们小于最大的文档号，而 ConjunctionScorer 又是取交集，它们当然不会在交

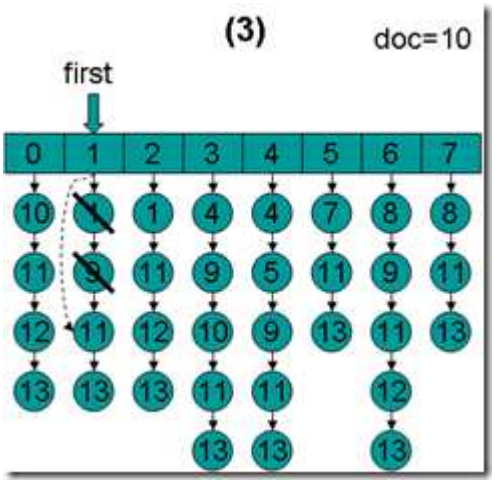
集中。

此过程如下：

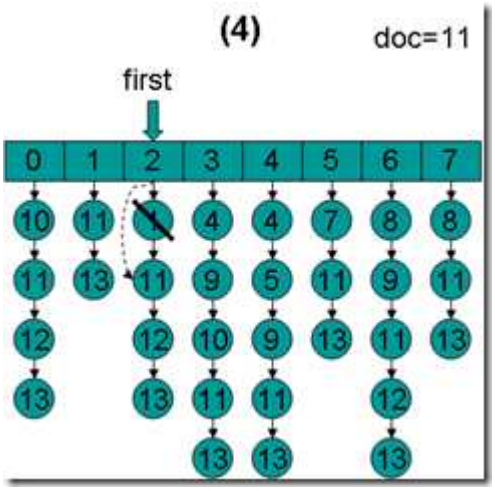
- doc = 8，first 指向第 0 项，advance 到大于 8 的第一篇文档，也即文档 10，然后设 doc = 10，first 指向第 1 项。



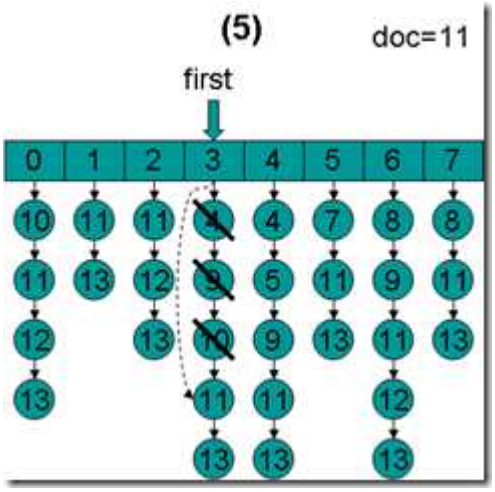
- doc = 10，first 指向第 1 项，advance 到文档 11，然后设 doc = 11，first 指向第 2 项。



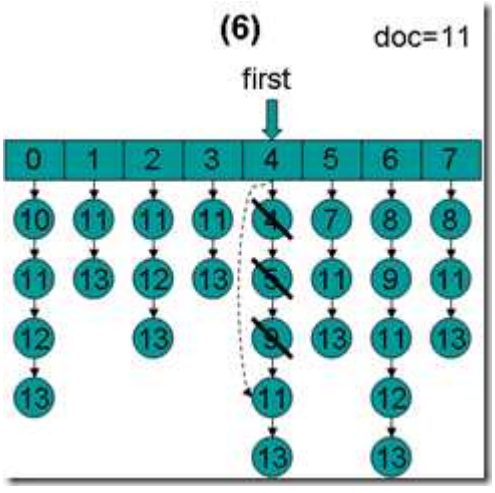
- doc = 11，first 指向第 2 项，advance 到文档 11，然后设 doc = 11，first 指向第 3 项。



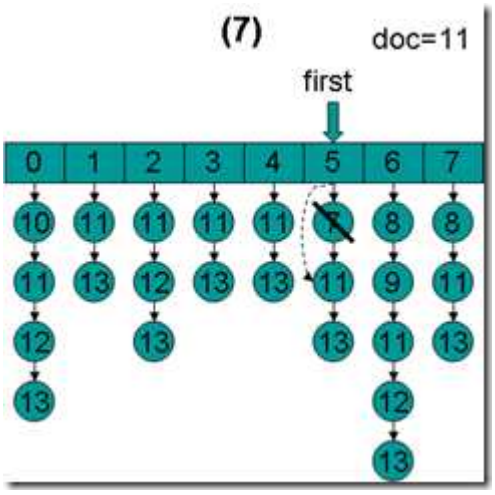
- doc = 11，first 指向第 3 项，advance 到文档 11，然后设 doc = 11，first 指向第 4 项。



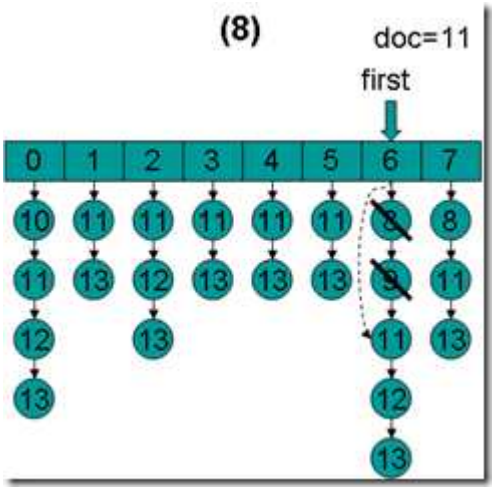
- doc = 11，first 指向第 4 项，advance 到文档 11，然后设 doc = 11，first 指向第 5 项。



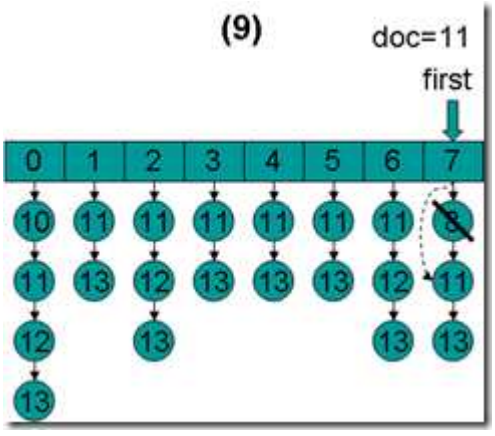
- doc = 11，first 指向第 5 项，advance 到文档 11，然后设 doc = 11，first 指向第 6 项。



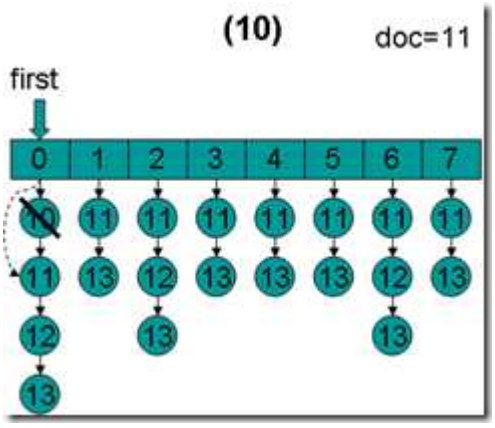
- doc = 11，first 指向第 6 项，advance 到文档 11，然后设 doc = 11，first 指向第 7 项。



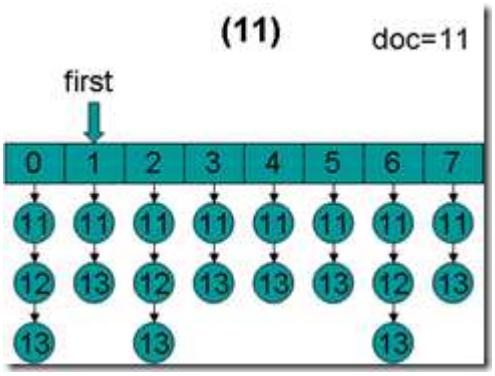
- doc = 11，first 指向第 7 项，advance 到文档 11，然后设 doc = 11，first 指向第 0 项。



- doc = 11，first 指向第 0 项，advance 到文档 11，然后设 doc = 11，first 指向第 1 项。



- doc = 11，first 指向第 1 项。因为 11 < 11 为 false，因而结束循环，返回 doc = 11。这时候我们会发现，在循环退出的时候，所有的倒排表的第一篇文档都是 11。



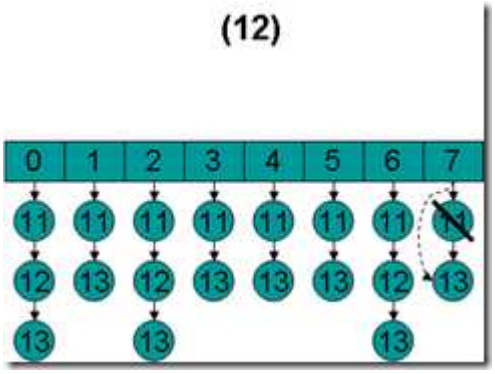
(5) 当 BooleanScorer2.score(Collector)中第一次调用 ConjunctionScorer.nextDoc()的时候，lastDoc 为-1，根据 nextDoc 函数的实现，返回 lastDoc = scorers[scorers.length - 1].docID()也即返回 11，lastDoc 也设为 11。

```
public int nextDoc() throws IOException {
  if (lastDoc == NO_MORE_DOCS) {
    return lastDoc;
  } else if (lastDoc == -1) {
    return lastDoc = scorers[scorers.length - 1].docID();
  }
  scorers[(scorers.length - 1)].nextDoc();
  return lastDoc = doNext();
}
```

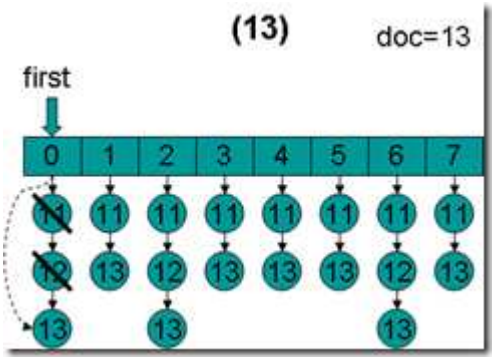
(6) 在 BooleanScorer2.score(Collector)中，调用 nextDoc()后，collector.collect(doc)来收集文档号(收集过程下节分析)，在收集文档的过程中，ConjunctionScorer.docID()会被调用，返回 lastDoc，也即当前的文档号为 11。

(7) 当 BooleanScorer2.score(Collector)第二次调用 ConjunctionScorer.nextDoc()时：

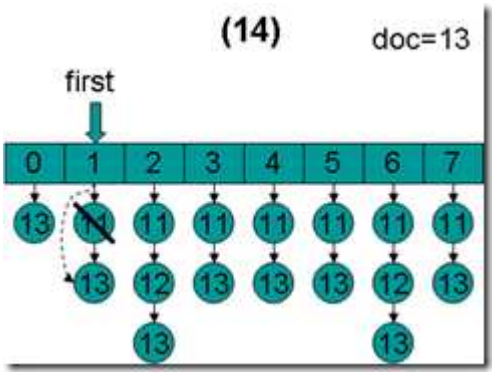
- 根据 nextDoc 函数的实现，首先调用 scorers[(scorers.length - 1)].nextDoc()，取最后一项的下一篇文档 13。



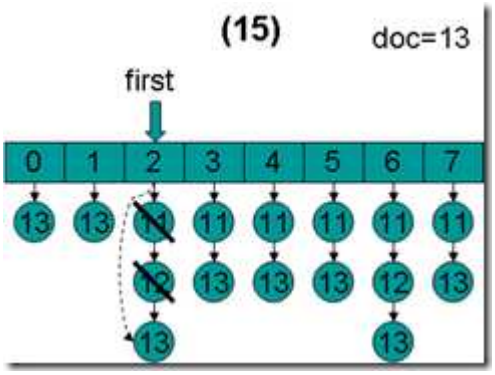
- 然后调用 lastDoc = doNext()，设 doc = 13，first = 0，进入循环。
- doc = 13，first 指向第 0 项，advance 到文档 13，然后设 doc = 13，first 指向第 1 项。



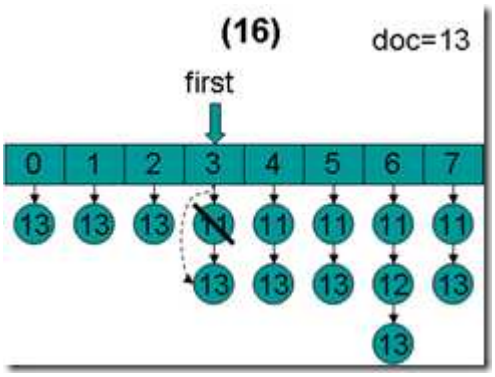
- doc = 13，first 指向第 1 项，advance 到文档 13，然后设 doc = 13，first 指向第 2 项。



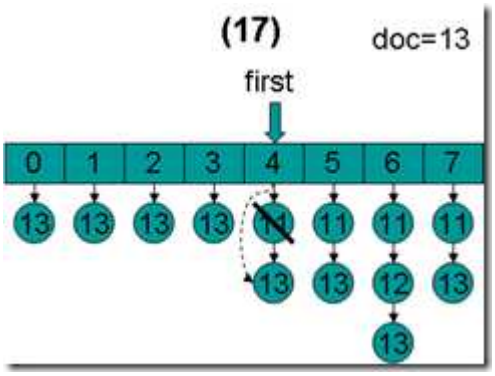
- doc = 13，first 指向第 2 项，advance 到文档 13，然后设 doc = 13，first 指向第 3 项。



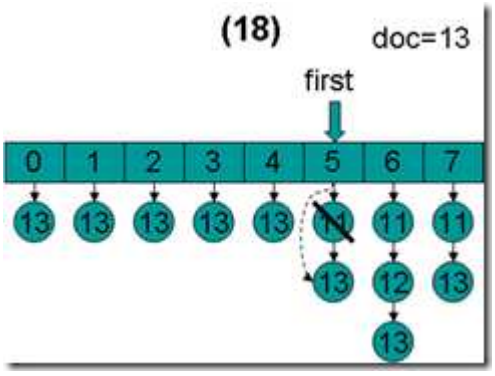
- doc = 13，first 指向第 3 项，advance 到文档 13，然后设 doc = 13，first 指向第 4 项。



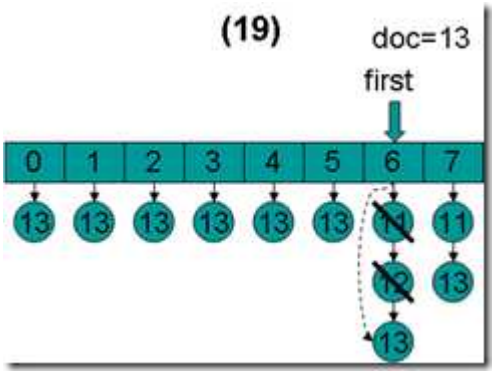
- doc = 13，first 指向第 4 项，advance 到文档 13，然后设 doc = 13，first 指向第 5 项。



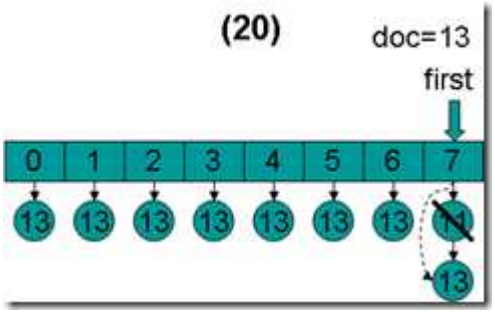
- doc = 13，first 指向第 5 项，advance 到文档 13，然后设 doc = 13，first 指向第 6 项。



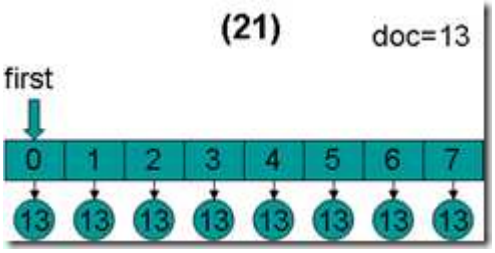
- doc = 13，first 指向第 6 项，advance 到文档 13，然后设 doc = 13，first 指向第 7 项。



- doc = 13，first 指向第 7 项，advance 到文档 13，然后设 doc = 13，first 指向第 0 项。



- doc = 13，first 指向第 0 项。因为 13 < 13 为 false，因而结束循环，返回 doc = 13。在循环退出的时候，所有的倒排表的第一篇文档都是 13。



(8) lastDoc 设为 13，在收集文档的过程中，ConjunctionScorer.docID()会被调用，返回 lastDoc，也即当前的文档号为 13。

(9) 当再次调用 nextDoc()的时候，返回 NO_MORE_DOCS，倒排表合并结束。

2.4.3.2、并集 DisjunctionSumScorer(A OR B)

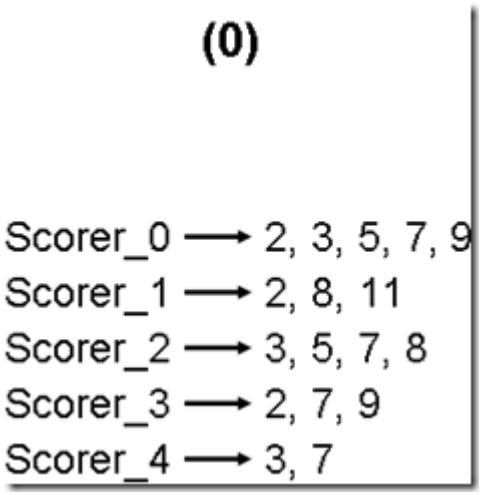
DisjunctionSumScorer 中有成员变量 List<Scorer> subScorers，是一个 Scorer 的链表，每一项代表一个倒排表，DisjunctionSumScorer 就是对这些倒排表取并集，然后将并集中的文档号在 nextDoc()函数中依次返回。

DisjunctionSumScorer 还有一个成员变量 minimumNrMatchers，表示最少需满足的子条件的个数，也即 subScorer 中，必须有至少 minimumNrMatchers 个 Scorer 都包含某个文档号，此

文档号才能够返回。

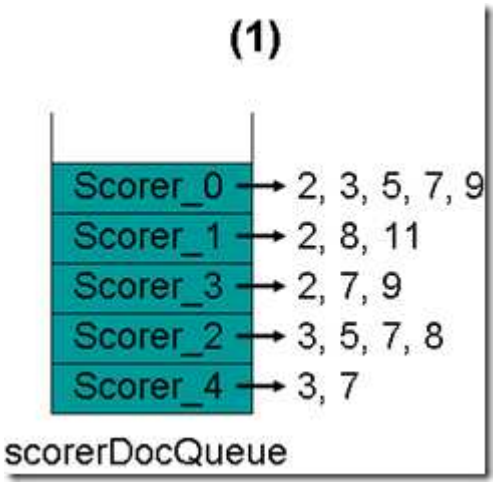
为了描述清楚此过程，下面举一个具体的例子来解释倒排表合并的过程：

(1) 假设 minimumNrMatchers = 4，倒排表最初如下：



(2) 在 DisjunctionSumScorer 的构造函数中，将倒排表放入一个优先级队列 scorerDocQueue 中(scorerDocQueue 的实现是一个最小堆)，队列中的 Scorer 按照第一篇文档的大小排序。

```
private void initScorerDocQueue() throws IOException {
  scorerDocQueue = new ScorerDocQueue(nrScorers);
  for (Scorer se : subScorers) {
   if (se.nextDoc() != NO_MORE_DOCS) { //此处的nextDoc使得每个Scorer得到第一篇文档号。
     scorerDocQueue.insert(se);
   }
  }
}
```



(3) 当 **BooleanScorer2.score(Collector)**中第一次调用 **nextDoc()**的时候，**advanceAfterCurrent** 被调用。

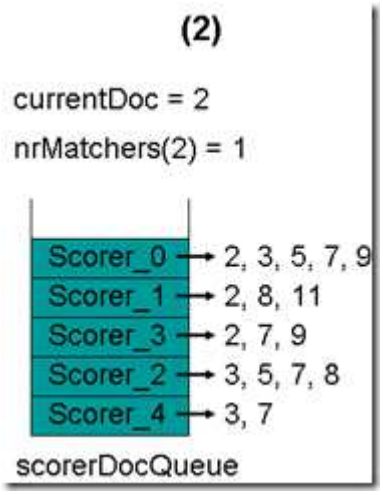
```
public int nextDoc() throws IOException {
  if (scorerDocQueue.size() < minimumNrMatchers || !advanceAfterCurrent()) {
    currentDoc = NO_MORE_DOCS;
  }
  return currentDoc;
}
```

```
protected boolean advanceAfterCurrent() throws IOException {
  do {
    currentDoc = scorerDocQueue.topDoc(); //当前的文档号为最顶层
    currentScore = scorerDocQueue.topScore(); //当前文档的打分
    nrMatchers = 1; //当前文档满足的子条件的个数，也即包含当前文档号的 Scorer 的个数
    do {
      //所谓 topNextAndAdjustElsePop 是指，最顶层(top)的 Scorer 取下一篇文档(Next)，如果能够取到，则最小堆的堆顶可能不再是最小值了，需要调整(Adjust，其实是 downHeap())，如果不能够取到，则最顶层的 Scorer 已经为空，则弹出队列(Pop)。
      if (!scorerDocQueue.topNextAndAdjustElsePop()) {
        if (scorerDocQueue.size() == 0) {
          break; // nothing more to advance, check for last match.
```

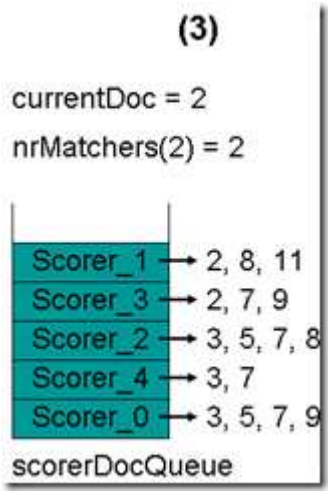
```
      }
    }
    //当最顶层的 Scorer 取到下一篇文档，并且调整完毕后，再取出此时最上层的 Scorer 的第一篇文档，如果不是 currentDoc，说明 currentDoc 此文档号已经统计完毕 nrMatchers，则退出内层循环。
    if (scorerDocQueue.topDoc() != currentDoc) {
      break; // All remaining subscorers are after currentDoc.
    }
    //否则 nrMatchers 加一，也即又多了一个 Scorer 也包含此文档号。
    currentScore += scorerDocQueue.topScore();
    nrMatchers++;
   } while (true);
   //如果统计出的 nrMatchers 大于最少需满足的子条件的个数，则此 currentDoc 就是满足条件的文档，则返回 true，在收集文档的过程中，DisjunctionSumScorer.docID()会被调用，返回 currentDoc。
   if (nrMatchers >= minimumNrMatchers) {
    return true;
   } else if (scorerDocQueue.size() < minimumNrMatchers) {
    return false;
   }
  } while (true);
}
```

advanceAfterCurrent 具体过程如下：

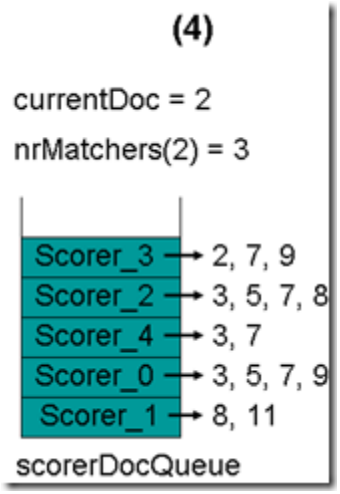
- 最初，currentDoc=2，文档 2 的 nrMatchers=1



- 最顶层的 Scorer 0 取得下一篇文档，为文档 3，重新调整最小堆后如下图。此时 currentDoc 等于最顶层 Scorer 1 的第一篇文档号，都为 2，文档 2 的 nrMatchers 为 2。

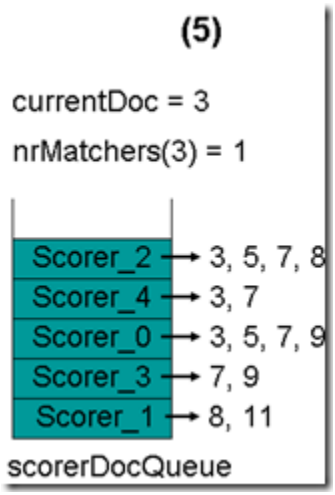


- 最顶层的 Scorer 1 取得下一篇文档，为文档 8，重新调整最小堆后如下图。此时 currentDoc 等于最顶层 Scorer 3 的第一篇文档号，都为 2，文档 2 的 nrMatchers 为 3。

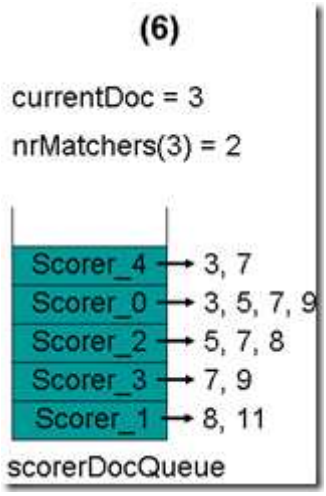


- 最顶层的 Scorer 3 取得下一篇文档，为文档 7，重新调整最小堆后如下图。此时 currentDoc

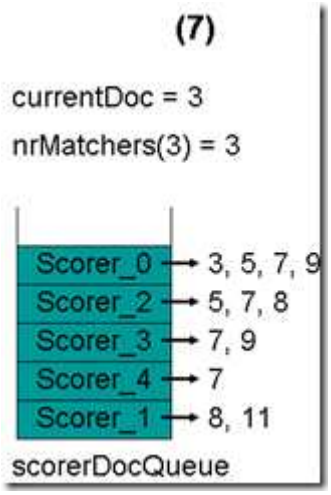
还为 2，不等于最顶层 Scorer 2 的第一篇文档 3，于是退出内循环。此时检查，发现文档 2 的 nrMatchers 为 3，小于 minimumNrMatchers，不满足条件。于是 currentDoc 设为最顶层 Scorer 2 的第一篇文档 3，nrMatchers 设为 1，重新进入下一轮循环。



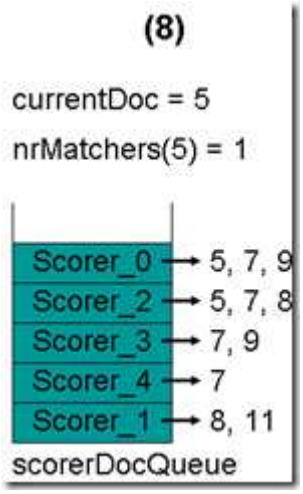
- 最顶层的 Scorer 2 取得下一篇文档，为文档 5，重新调整最小堆后如下图。此时 currentDoc 等于最顶层 Scorer 4 的第一篇文档号，都为 3，文档 3 的 nrMatchers 为 2。



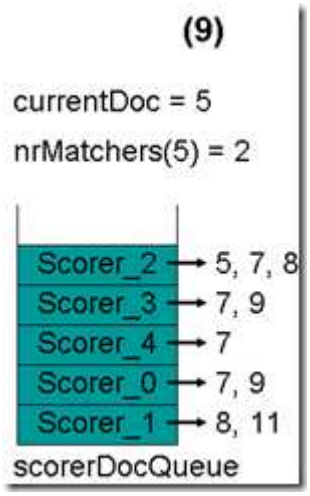
- 最顶层的 Scorer 4 取得下一篇文档，为文档 7，重新调整最小堆后如下图。此时 currentDoc 等于最顶层 Scorer 0 的第一篇文档号，都为 3，文档 3 的 nrMatchers 为 3。



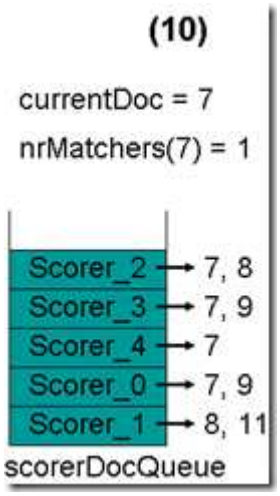
- 最顶层的 Scorer 0 取得下一篇文档，为文档 5，重新调整最小堆后如下图。此时 currentDoc 还为 3，不等于最顶层 Scorer 0 的第一篇文档 5，于是退出内循环。此时检查，发现文档 3 的 nrMatchers 为 3，小于 minimumNrMatchers，不满足条件。于是 currentDoc 设为最顶层 Scorer 0 的第一篇文档 5，nrMatchers 设为 1，重新进入下一轮循环。



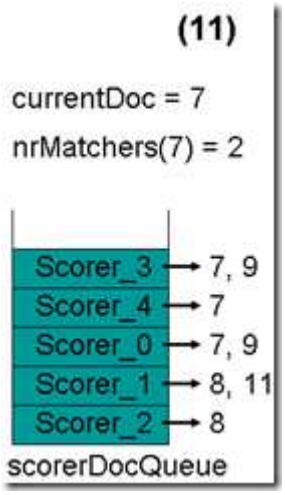
- 最顶层的 Scorer 0 取得下一篇文档，为文档 7，重新调整最小堆后如下图。此时 currentDoc 等于最顶层 Scorer 2 的第一篇文档号，都为 5，文档 5 的 nrMatchers 为 2。



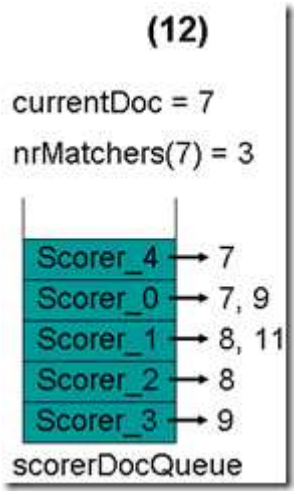
- 最顶层的 Scorer 2 取得下一篇文档，为文档 7，重新调整最小堆后如下图。此时 currentDoc 还为 5，不等于最顶层 Scorer 2 的第一篇文档 7，于是退出内循环。此时检查，发现文档 5 的 nrMatchers 为 2，小于 minimumNrMatchers，不满足条件。于是 currentDoc 设为最顶层 Scorer 2 的第一篇文档 7，nrMatchers 设为 1，重新进入下一轮循环。



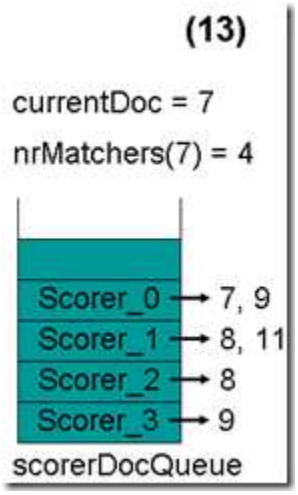
- 最顶层的 Scorer 2 取得下一篇文档，为文档 8，重新调整最小堆后如下图。此时 currentDoc 等于最顶层 Scorer 3 的第一篇文档号，都为 7，文档 7 的 nrMatchers 为 2。



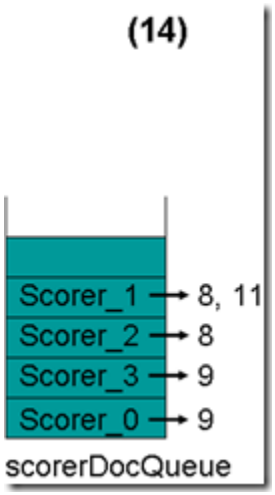
- 最顶层的 Scorer 3 取得下一篇文档，为文档 9，重新调整最小堆后如下图。此时 currentDoc 等于最顶层 Scorer 4 的第一篇文档号，都为 7，文档 7 的 nrMatchers 为 3。



- 最顶层的 Scorer 4 取得下一篇文档，结果为空，Scorer 4 所有的文档遍历完毕，弹出队列，重新调整最小堆后如下图。此时 currentDoc 等于最顶层 Scorer 0 的第一篇文档号，都为 7，文档 7 的 nrMatchers 为 4。



- 最顶层的 Scorer 0 取得下一篇文档，为文档 9，重新调整最小堆后如下图。此时 currentDoc 还为 7，不等于最顶层 Scorer 1 的第一篇文档 8，于是退出内循环。此时检查，发现文档 7 的 nrMatchers 为 4，大于等于 minimumNrMatchers，满足条件，返回 true，退出外循环。



(4) currentDoc 设为 7，在收集文档的过程中，DisjunctionSumScorer.docID() 会被调用，返回 currentDoc，也即当前的文档号为 7。

(5) 当再次调用 nextDoc()的时候，文档 8, 9, 11 都不满足要求，最后返回 NO_MORE_DOCS，倒排表合并结束。

2.4.3.3、差集 ReqExclScorer(+A -B)

ReqExclScorer 有成员变量 Scorer reqScorer 表示必须满足的部分(required)，成员变量 DocIdSetIterator exclDisi 表示必须不能满足的部分，ReqExclScorer 就是返回 reqScorer 和

exclDisi 的倒排表的差集，也即在 reqScorer 的倒排表中排除 exclDisi 中的文档号。

当 nextDoc()调用的时候，首先取得 reqScorer 的第一个文档号，然后 toNonExcluded()函数则判断此文档号是否被 exclDisi 排除掉，如果没有，则返回此文档号，如果排除掉，则取下一个文档号，看是否被排除掉，依次类推，直到找到一个文档号，或者返回 NO_MORE_DOCS。

```
public int nextDoc() throws IOException {
  if (reqScorer == null) {
    return doc;
  }
  doc = reqScorer.nextDoc();
  if (doc == NO_MORE_DOCS) {
    reqScorer = null;
    return doc;
  }
  if (exclDisi == null) {
    return doc;
  }
  return doc = toNonExcluded();
}
```

```
private int toNonExcluded() throws IOException {
  //取得被排除的文档号
  int exclDoc = exclDisi.docID();
  //取得当前 required 文档号
  int reqDoc = reqScorer.docID();
  do {
   //如果 required 文档号小于被排除的文档号，由于倒排表是按照从小到大的顺序排列的，因而此 required 文档号不会被排除，返回。
    if (reqDoc < exclDoc) {
      return reqDoc;
```

```
    } else if (reqDoc > exclDoc) {
   //如果 required 文档号大于被排除的文档号，则此 required 文档号有可能被排除。于是 exclDisi 移动到大于或者等于 required 文档号的文档。
     exclDoc = exclDisi.advance(reqDoc);
    //如果被排除的倒排表遍历结束，则 required 文档号不会被排除，返回。
     if (exclDoc == NO_MORE_DOCS) {
      exclDisi = null;
      return reqDoc;
     }
   //如果 exclDisi 移动后，大于 required 文档号，则 required 文档号不会被排除，返回。
     if (exclDoc > reqDoc) {
      return reqDoc; // not excluded
     }
    }
   //如果 required 文档号等于被排除的文档号，则被排除，取下一个 required 文档号。
  } while ((reqDoc = reqScorer.nextDoc()) != NO_MORE_DOCS);
  reqScorer = null;
  return NO_MORE_DOCS;
}
```

2.4.3.4、ReqOptSumScorer(+A B)

ReqOptSumScorer 包含两个成员变量，Scorer reqScorer 代表必须(required)满足的文档倒排表，Scorer optScorer 代表可以(optional)满足的文档倒排表。

如代码显示，在 nextDoc()中，返回的就是 required 的文档倒排表，只不过在计算 score 的时候打分更高。

```
public int nextDoc() throws IOException {
```

```
  return reqScorer.nextDoc();
}
```

2.4.3.5、有关 BooleanScorer 及 scoresDocsOutOfOrder

在 BooleanWeight.scorer 生成 Scorer 树的时候，除了生成上述的 BooleanScorer2 外， 还会生成 BooleanScorer，是在以下的条件下：

- !scoreDocsInOrder ： 根 据 2.4.2 节 的 步 骤 (c) ， scoreDocsInOrder = !collector.acceptsDocsOutOfOrder() ， 此 值 是 在 search 中 调 用 TopScoreDocCollector.create(nDocs, !weight.scoresDocsOutOfOrder()) 的 时 候 设 定 的 ， scoreDocsInOrder = !weight.scoresDocsOutOfOrder()，其代码如下：

```
public boolean scoresDocsOutOfOrder() {
  int numProhibited = 0;
  for (BooleanClause c : clauses) {
    if (c.isRequired()) {
      return false;
    } else if (c.isProhibited()) {
      ++numProhibited;
    }
  }
  if (numProhibited > 32) {
    return false;
  }
  return true;
}
```

- topScorer：根据 2.4.2 节的步骤(c)，此值为 true。
- required.size() == 0，没有必须满足的子语句。
- prohibited.size() < 32，不需不能满足的子语句小于 32。

从上面可以看出，最后两个条件和 scoresDocsOutOfOrder 函数中的逻辑是一致的。

下面我们看看 BooleanScorer 如何合并倒排表的：

```
public int nextDoc() throws IOException {
  boolean more;
  do {
   //bucketTable 等于是存放合并后的倒排表的文档队列
   while (bucketTable.first != null) {
    //从队列中取出第一篇文档，返回
    current = bucketTable.first;
    bucketTable.first = current.next;
    if ((current.bits & prohibitedMask) == 0 &&
       (current.bits & requiredMask) == requiredMask &&
       current.coord >= minNrShouldMatch) {
     return doc = current.doc;
    }
   }
   //如果队列为空，则填充队列。
   more = false;
   end += BucketTable.SIZE;
   //按照 Scorer 的顺序，依次用 Scorer 中的倒排表填充队列，填满为止。
   for (SubScorer sub = scorers; sub != null; sub = sub.next) {
    Scorer scorer = sub.scorer;
    sub.collector.setScorer(scorer);
    int doc = scorer.docID();
    while (doc < end) {
     sub.collector.collect(doc);
     doc = scorer.nextDoc();
    }
```

```
      more |= (doc != NO_MORE_DOCS);
    }
  } while (bucketTable.first != null || more);
  return doc = NO_MORE_DOCS;
}
```

```
public final void collect(final int doc) throws IOException {
  final BucketTable table = bucketTable;
  final int i = doc & BucketTable.MASK;
  Bucket bucket = table.buckets[i];
  if (bucket == null)
    table.buckets[i] = bucket = new Bucket();
  if (bucket.doc != doc) {
    bucket.doc = doc;
    bucket.score = scorer.score();
    bucket.bits = mask;
    bucket.coord = 1;
    bucket.next = table.first;
    table.first = bucket;
  } else {
    bucket.score += scorer.score();
    bucket.bits |= mask;
    bucket.coord++;
  }
}
```

从上面的实现我们可以看出，BooleanScorer 合并倒排表的时候，并不是按照文档号从小到大的顺序排列的。

从原理上我们可以理解，在 AND 的查询条件下，倒排表的合并按照算法需要按照文档号从小到大的顺序排列。然而在没有 AND 的查询条件下，如果都是 OR，则文档号是否按照顺序

返回就不重要了，因而 scoreDocsInOrder 就是 false。

因而上面的 DisjunctionSumScorer，其实"apple boy dog"是不能产生 DisjunctionSumScorer 的，而仅有在有 AND 的查询条件下，才产生 DisjunctionSumScorer。

我们做实验如下：

对于查询语句"apple boy dog"，生成的 Scorer 如下：

```
scorer    BooleanScorer (id=34)
  bucketTable    BooleanScorer$BucketTable  (id=39)
  coordFactors    float[4]  (id=41)
  current    null
  doc    -1
  doc    -1
  end    0
  maxCoord    4
  minNrShouldMatch    0
  nextMask    1
  prohibitedMask    0
  requiredMask    0
  scorers    BooleanScorer$SubScorer  (id=43)
    collector    BooleanScorer$BooleanScorerCollector  (id=49)
    next    BooleanScorer$SubScorer  (id=51)
      collector    BooleanScorer$BooleanScorerCollector  (id=68)
      next    BooleanScorer$SubScorer  (id=69)
        collector    BooleanScorer$BooleanScorerCollector  (id=76)
        next    null
        prohibited    false
        required    false
        scorer    TermScorer  (id=77)
          doc    -1
```

doc 0

docs int[32] (id=79)

freqs int[32] (id=80)

norms byte[4] (id=58)

pointer 0

pointerMax 2

scoreCache float[32] (id=81)

similarity DefaultSimilarity (id=45)

termDocs SegmentTermDocs (id=82)

weight TermQuery$TermWeight (id=84) **//weight(contents:apple)**

weightValue 0.828608

prohibited false

required false

scorer TermScorer (id=70)

doc -1

doc 1

docs int[32] (id=72)

freqs int[32] (id=73)

norms byte[4] (id=58)

pointer 0

pointerMax 1

scoreCache float[32] (id=74)

similarity DefaultSimilarity (id=45)

termDocs SegmentTermDocs (id=86)

weight TermQuery$TermWeight (id=87) **//weight(contents:boy)**

weightValue 1.5407716

```
    prohibited    false
    required    false
    scorer    TermScorer  (id=52)
      doc    -1
      doc    0
      docs    int[32]  (id=54)
      freqs    int[32]  (id=56)
      norms    byte[4]  (id=58)
      pointer    0
      pointerMax    3
      scoreCache    float[32]  (id=61)
      similarity    DefaultSimilarity  (id=45)
      termDocs    SegmentTermDocs  (id=62)
      weight    TermQuery$TermWeight  (id=66)  //weight(contents:cat)
      weightValue    0.48904076
  similarity    DefaultSimilarity  (id=45)
```

对于查询语句"+hello (apple boy dog)"，生成的 Scorer 对象如下：

```
scorer    BooleanScorer2  (id=40)
  coordinator    BooleanScorer2$Coordinator  (id=42)
  countingSumScorer    ReqOptSumScorer  (id=43)
  minNrShouldMatch    0
  optionalScorers    ArrayList<E>  (id=44)
    elementData    Object[10]  (id=62)
      [0]    BooleanScorer2  (id=84)
        coordinator    BooleanScorer2$Coordinator  (id=87)
        countingSumScorer    BooleanScorer2$1  (id=88)
        minNrShouldMatch    0
        optionalScorers    ArrayList<E>  (id=89)
```

elementData Object[10] (id=95)

[0] TermScorer (id=97)

docs int[32] (id=101)

freqs int[32] (id=102)

norms byte[4] (id=71)

pointer 0

pointerMax 2

scoreCache float[32] (id=103)

similarity DefaultSimilarity (id=48)

termDocs SegmentTermDocs (id=104)

//weight(contents:apple)

weight TermQuery$TermWeight (id=105)

weightValue 0.525491

[1] TermScorer (id=98)

docs int[32] (id=107)

freqs int[32] (id=108)

norms byte[4] (id=71)

pointer 0

pointerMax 1

scoreCache float[32] (id=110)

similarity DefaultSimilarity (id=48)

termDocs SegmentTermDocs (id=111)

//weight(contents:boy)

weight TermQuery$TermWeight (id=112)

weightValue 0.9771348

[2] TermScorer (id=99)

docs int[32] (id=114)

freqs int[32] (id=118)

norms byte[4] (id=71)

pointer 0

pointerMax 3

scoreCache float[32] (id=119)

similarity DefaultSimilarity (id=48)

termDocs SegmentTermDocs (id=120)

//weight(contents:cat)

weight TermQuery$TermWeight (id=121)

weightValue 0.3101425

size 3

prohibitedScorers ArrayList<E> (id=90)

requiredScorers ArrayList<E> (id=91)

similarity DefaultSimilarity (id=48)

size 1

prohibitedScorers ArrayList<E> (id=46)

requiredScorers ArrayList<E> (id=47)

elementData Object[10] (id=59)

[0] TermScorer (id=66)

docs int[32] (id=68)

freqs int[32] (id=70)

norms byte[4] (id=71)

pointer 0

pointerMax 0

scoreCache float[32] (id=73)

similarity DefaultSimilarity (id=48)

termDocs SegmentTermDocs (id=76)

weight TermQuery$TermWeight (id=78) **//weight(contents:hello)**

weightValue 2.6944637

```
    size    1
  similarity    DefaultSimilarity   (id=48)
```

2.4.4、收集文档结果集合及计算打分

在函数 IndexSearcher.search(Weight, Filter, int) 中，有如下代码：

TopScoreDocCollector collector =

TopScoreDocCollector.create(nDocs, !weight.scoresDocsOutOfOrder());

search(weight, filter, collector);

return collector.topDocs();

2.4.4.1、创建结果文档收集器

TopScoreDocCollector collector =

TopScoreDocCollector.create(nDocs, !weight.scoresDocsOutOfOrder());

```
public static TopScoreDocCollector create(int numHits, boolean docsScoredInOrder) {
  if (docsScoredInOrder) {
    return new InOrderTopScoreDocCollector(numHits);
  } else {
    return new OutOfOrderTopScoreDocCollector(numHits);
  }
}
```

其根据是否按照文档号从小到大返回文档而创建 InOrderTopScoreDocCollector 或者 OutOfOrderTopScoreDocCollector，两者的不同在于收集文档的方式不同。

2.4.4.2、收集文档号

当创建完毕 Scorer 对象树和 SumScorer 对象树后，IndexSearcher.search(Weight, Filter, Collector) 有以下调用：

scorer.score(collector) ，如下代码所示，其不断的得到合并的倒排表后的文档号，并收集它

们。

```
public void score(Collector collector) throws IOException {
  collector.setScorer(this);
  while ((doc = countingSumScorer.nextDoc()) != NO_MORE_DOCS) {
    collector.collect(doc);
  }
}
```

InOrderTopScoreDocCollector 的 collect 函数如下：

```
public void collect(int doc) throws IOException {
  float score = scorer.score();
  totalHits++;
  if (score <= pqTop.score) {
    return;
  }
  pqTop.doc = doc + docBase;
  pqTop.score = score;
  pqTop = pq.updateTop();
}
```

OutOfOrderTopScoreDocCollector 的 collect 函数如下：

```
public void collect(int doc) throws IOException {
  float score = scorer.score();
  totalHits++;
  doc += docBase;
  if (score < pqTop.score || (score == pqTop.score && doc > pqTop.doc)) {
    return;
  }
  pqTop.doc = doc;
```

```
  pqTop.score = score;
  pqTop = pq.updateTop();
}
```

从上面的代码可以看出，collector 的作用就是首先计算文档的打分，然后根据打分，将文档放入优先级队列(最小堆)中，最后在优先级队列中取前 N 篇文档。

然而存在一个问题，如果要取 10 篇文档，而第 8,9,10,11,12 篇文档的打分都相同，则抛弃那些呢？Lucene 的策略是，在文档打分相同的情况下，文档号小的优先。

也即 8,9,10 被保留，11,12 被抛弃。

由上面的叙述可知，创建 collector 的时候，根据文档是否将按照文档号从小到大的顺序返回而创建 InOrderTopScoreDocCollector 或者 OutOfOrderTopScoreDocCollector。

对于 InOrderTopScoreDocCollector，由于文档是按照顺序返回的，后来的文档号肯定大于前面的文档号，因而当 score <= pqTop.score 的时候，直接抛弃。

对于 OutOfOrderTopScoreDocCollector，由于文档不是按顺序返回的，因而当 score<pqTop.score，自然直接抛弃，当 score==pqTop.score 的时候，则要比较后来的文档和前面的文档的大小，如果大于，则抛弃，如果小于则入队列。

2.4.4.3、打分计算

BooleanScorer2 的打分函数如下：

- 将子语句的打分乘以 **coord**

```
public float score() throws IOException {
  coordinator.nrMatchers = 0;
  float sum = countingSumScorer.score();
  return sum * coordinator.coordFactors[coordinator.nrMatchers];
}
```

ConjunctionScorer 的打分函数如下：

- 将取交集的子语句的打分相加，然后乘以 **coord**

```
public float score() throws IOException {
```

```
  float sum = 0.0f;
  for (int i = 0; i < scorers.length; i++) {
    sum += scorers[i].score();
  }
  return sum * coord;
}
```

DisjunctionSumScorer 的打分函数如下：

```
public float score() throws IOException { return currentScore; }
```

currentScore 计算如下：

```
currentScore += scorerDocQueue.topScore();
```

以上计算是在 DisjunctionSumScorer 的倒排表合并算法中进行的，其是取堆顶的打分函数。

```
public final float topScore() throws IOException {
    return topHSD.scorer.score();
}
```

ReqExclScorer 的打分函数如下：

- 仅仅取 required 语句的打分

```
public float score() throws IOException {
  return reqScorer.score();
}
```

ReqOptSumScorer 的打分函数如下：

- 上面曾经指出，ReqOptSumScorer 的 nextDoc()函数仅仅返回 required 语句的文档号。
- 而 optional 的部分仅仅在打分的时候有所体现，从下面的实现可以看出 optional 的语句的分数加到 required 语句的分数上，也即文档还是 required 语句包含的文档，只不过是当此文档能够满足 optional 的语句的时候，打分得到增加。

```
public float score() throws IOException {
  int curDoc = reqScorer.docID();
```

```
  float reqScore = reqScorer.score();
  if (optScorer == null) {
    return reqScore;
  }
  int optScorerDoc = optScorer.docID();
  if (optScorerDoc < curDoc && (optScorerDoc = optScorer.advance(curDoc)) == NO_MORE_DOCS)
{
    optScorer = null;
    return reqScore;
  }
  return optScorerDoc == curDoc ? reqScore + optScorer.score() : reqScore;
}
```

TermScorer 的打分函数如下：

- 整个 Scorer 及 SumScorer 对象树的打分计算，最终都会源自叶子节点 TermScorer 上。
- 从 TermScorer 的计算可以看出，它计算出 tf * norm * weightValue = **tf * norm * queryNorm * idf^2 * t.getBoost()**

```
public float score() {
  int f = freqs[pointer];
  float raw = f < SCORE_CACHE_SIZE ? scoreCache[f] : getSimilarity().tf(f)*weightValue;
  return norms == null ? raw : raw * SIM_NORM_DECODER[norms[doc] & 0xFF];
}
```

Lucene 的打分公式整体如下，2.4.1 计算了图中的红色的部分，此步计算了蓝色的部分：

$$score(q,d) = coord(q,d) \times queryNorm(q) \times \sum_{t \text{ in } q} (tf(t \text{ in } d) \times idf(t)^2 \times t.getBoost() \times norm(t,d))$$

打分计算到此结束。

2.4.4.4、返回打分最高的 N 篇文档

IndexSearcher.search(Weight, Filter, int)中，在收集完文档后，调用 collector.topDocs()返回打分最高的 N 篇文档：

```
public final TopDocs topDocs() {
  return topDocs(0, totalHits < pq.size() ? totalHits : pq.size());
}
```

```
public final TopDocs topDocs(int start, int howMany) {
  int size = totalHits < pq.size() ? totalHits : pq.size();
  howMany = Math.min(size - start, howMany);
  ScoreDoc[] results = new ScoreDoc[howMany];
  //由于 pq 是最小堆，因而要首先弹出最小的文档。比如 qp 中总共有 50 篇文档，想取第 5 到 10 篇文档，则应该先弹出打分最小的 40 篇文档。
  for (int i = pq.size() - start - howMany; i > 0; i--) { pq.pop(); }
  populateResults(results, howMany);
  return newTopDocs(results, start);
}
```

```
protected void populateResults(ScoreDoc[] results, int howMany) {
  //然后再从 pq 弹出第 5 到 10 篇文档，并按照打分从大到小的顺序放入 results 中。
  for (int i = howMany - 1; i >= 0; i--) {
    results[i] = pq.pop();
  }
}
```

```
protected TopDocs newTopDocs(ScoreDoc[] results, int start) {
  return results == null ? EMPTY_TOPDOCS : new TopDocs(totalHits, results);
}
```

2.4.5、Lucene 如何在搜索阶段读取索引信息

以上叙述的是搜索过程中如何进行倒排表合并以及计算打分。然而索引信息是从索引文件中读出来的，下面分析如何读取这些信息。

其实读取的信息无非是两种信息，一个是词典信息，一个是倒排表信息。

词典信息的读取是在 Scorer 对象树生成的时候进行的，真正读取这些信息的是叶子节点 TermScorer。

倒排表信息的读取时在合并倒排表的时候进行的，真正读取这些信息的也是叶子节点 TermScorer.nextDoc()。

2.4.5.1、读取词典信息

此步是在 TermWeight.scorer(IndexReader, boolean, boolean) 中进行的，其代码如下：

```
public Scorer scorer(IndexReader reader, boolean scoreDocsInOrder, boolean topScorer) {
  TermDocs termDocs = reader.termDocs(term);
  if (termDocs == null)
    return null;
  return new TermScorer(this, termDocs, similarity, reader.norms(term.field()));
}
```

ReadOnlySegmentReader.termDocs(Term)是找到 Term 并生成用来读倒排表的 TermDocs 对象：

```
public TermDocs termDocs(Term term) throws IOException {
  ensureOpen();
  TermDocs termDocs = termDocs();
  termDocs.seek(term);
  return termDocs;
}
```

termDocs()函数首先生成 SegmentTermDocs 对象，用于读取倒排表：

```
protected SegmentTermDocs(SegmentReader parent) {
```

```
  this.parent = parent;
  this.freqStream = (IndexInput) parent.core.freqStream.clone();//用于读取 freq
  synchronized (parent) {
    this.deletedDocs = parent.deletedDocs;
  }
  this.skipInterval = parent.core.getTermsReader().getSkipInterval();
  this.maxSkipLevels = parent.core.getTermsReader().getMaxSkipLevels();
}
```

SegmentTermDocs.seek(Term)是读取词典中的 Term，并将 freqStream 指向此 Term 对应的倒排表：

```
public void seek(Term term) throws IOException {
  TermInfo ti = parent.core.getTermsReader().get(term);
  seek(ti, term);
}
```

TermInfosReader.get(Term, boolean)主要是读取词典中的 Term 得到 TermInfo，代码如下：

```
  private TermInfo get(Term term, boolean useCache) {
    if (size == 0) return null;
    ensureIndexIsRead();
    TermInfo ti;
    ThreadResources resources = getThreadResources();
    SegmentTermEnum enumerator = resources.termEnum;
    seekEnum(enumerator, getIndexOffset(term));
    enumerator.scanTo(term);
    if (enumerator.term() != null && term.compareTo(enumerator.term()) == 0) {
      ti = enumerator.termInfo();
    } else {
      ti = null;
    }
```

```
    return ti;
  }
```

在 IndexReader 打开一个索引文件夹的时候，会从 tii 文件中读出的 Term index 到 indexPointers 数组中， TermInfosReader.seekEnum(SegmentTermEnum enumerator, int indexOffset)负责在 indexPointers 数组中找 Term 对应的 tis 文件中所在的跳表区域的位置。

```
private final void seekEnum(SegmentTermEnum enumerator, int indexOffset) throws IOException {
  enumerator.seek(indexPointers[indexOffset],
            (indexOffset * totalIndexInterval) - 1,
            indexTerms[indexOffset], indexInfos[indexOffset]);
}
```

```
final void SegmentTermEnum.seek(long pointer, int p, Term t, TermInfo ti) {
  input.seek(pointer);
  position = p;
  termBuffer.set(t);
  prevBuffer.reset();
  termInfo.set(ti);
}
```

SegmentTermEnum.scanTo(Term)在跳表区域中，一个一个往下找，直到找到 Term：

```
final int scanTo(Term term) throws IOException {
  scanBuffer.set(term);
  int count = 0;
  //不断取得下一个 term 到 termBuffer 中，目标 term 放入 scanBuffer 中，当两者相等的时候，目标 Term 找到。
  while (scanBuffer.compareTo(termBuffer) > 0 && next()) {
    count++;
  }
  return count;
}
```

```
public final boolean next() throws IOException {
  if (position++ >= size - 1) {
    prevBuffer.set(termBuffer);
    termBuffer.reset();
    return false;
  }
  prevBuffer.set(termBuffer);
  //读取 Term 的字符串
  termBuffer.read(input, fieldInfos);
  //读取 docFreq，也即多少文档包含此 Term
  termInfo.docFreq = input.readVInt();
  //读取偏移量
  termInfo.freqPointer += input.readVLong();
  termInfo.proxPointer += input.readVLong();
  if (termInfo.docFreq >= skipInterval)
     termInfo.skipOffset = input.readVInt();
  indexPointer += input.readVLong();
  return true;
}
```

TermBuffer.read(IndexInput, FieldInfos) 代码如下：

```
  public final void read(IndexInput input, FieldInfos fieldInfos) {
    this.term = null;
    int start = input.readVInt();
    int length = input.readVInt();
    int totalLength = start + length;
    text.setLength(totalLength);
    input.readChars(text.result, start, length);
    this.field = fieldInfos.fieldName(input.readVInt());
```

```
  }
```

SegmentTermDocs.seek(TermInfo ti, Term term)根据 TermInfo，将 freqStream 指向此 Term 对应的倒排表位置：

```
void seek(TermInfo ti, Term term) {
  count = 0;
  FieldInfo fi = parent.core.fieldInfos.fieldInfo(term.field);
  df = ti.docFreq;
  doc = 0;
  freqBasePointer = ti.freqPointer;
  proxBasePointer = ti.proxPointer;
  skipPointer = freqBasePointer + ti.skipOffset;
  freqStream.seek(freqBasePointer);
  haveSkipped = false;
}
```

2.4.5.2、读取倒排表信息

当读出 Term 的信息得到 TermInfo 后，并且 freqStream 指向此 Term 的倒排表位置的时候，下面就是在 TermScorer.nextDoc()函数中读取倒排表信息：

```
public int nextDoc() throws IOException {
  pointer++;
  if (pointer >= pointerMax) {
    pointerMax = termDocs.read(docs, freqs);
    if (pointerMax != 0) {
      pointer = 0;
    } else {
      termDocs.close();
      return doc = NO_MORE_DOCS;
```

```
    }
   }
   doc = docs[pointer];
   return doc;
}
```

SegmentTermDocs.read(int[], int[]) 代码如下：

```
public int read(final int[] docs, final int[] freqs) {
   final int length = docs.length;
   int i = 0;
   while (i < length && count < df) {
     //读取 docid
     final int docCode = freqStream.readVInt();
     doc += docCode >>> 1;
     if ((docCode & 1) != 0)
       freq = 1;
     else
       freq = freqStream.readVInt();     //读取 freq
     count++;
     if (deletedDocs == null || !deletedDocs.get(doc)) {
       docs[i] = doc;
       freqs[i] = freq;
       ++i;
     }
     return i;
   }
}
```

第八章：Lucene 的查询语法，JavaCC 及 QueryParser

一、Lucene 的查询语法

Lucene 所支持的查询语法可见

http://lucene.apache.org/java/3_0_1/queryparsersyntax.html

(1) 语法关键字

+ - && || ! ( ) { } [ ] ^ " ~ * ? : \

如果所要查询的查询词中本身包含关键字，则需要用\进行转义

(2) 查询词(Term)

Lucene 支持两种查询词，一种是单一查询词，如"hello"，一种是词组(phrase)，如"hello world"。

(3) 查询域(Field)

在查询语句中，可以指定从哪个域中寻找查询词，如果不指定，则从默认域中查找。

查询域和查询词之间用:分隔，如 title:"Do it right"。

:仅对紧跟其后的查询词起作用，如果 title:Do it right，则仅表示在 title 中查询 Do，而 it right 要在默认域中查询。

(4) 通配符查询(Wildcard)

支持两种通配符：?表示一个字符，*表示多个字符。

通配符可以出现在查询词的中间或者末尾，如 te?t，test*，te*t，但决不能出现在开始，如*test，?test。

(5) 模糊查询(Fuzzy)

模糊查询的算法是基于 Levenshtein Distance，也即当两个词的差别小于某个比例的时候，就算匹配，如 roam~0.8，即表示差别小于 0.2，相似度大于 0.8 才算匹配。

(6) 临近查询(Proximity)

在词组后面跟随~10，表示词组中的多个词之间的距离之和不超过 10，则满足查询。

所谓词之间的距离，即查询词组中词为满足和目标词组相同的最小移动次数。

如索引中有词组"apple boy cat"。

如果查询词为"apple boy cat"~0，则匹配。

如果查询词为"boy apple cat"~2，距离设为 2 方能匹配，设为 1 则不能匹配。

(0)	boy	apple	cat
(1)		boy apple	cat
(2)	apple	boy	cat

如果查询词为"cat boy apple"~4，距离设为 4 方能匹配。

(0)	cat	boy	apple
(1)		cat boy	apple
(2)		boy	cat apple
(3)		boy apple	cat
(4)	apple	boy	cat

(7) 区间查询(Range)

区间查询包含两种，一种是包含边界，用[A TO B]指定，一种是不包含边界，用{A TO B}指定。

如 date:[20020101 TO 20030101]，当然区间查询不仅仅用于时间，如 title:{Aida TO Carmen}

(8) 增加一个查询词的权重(Boost)

可以在查询词后面加^N 来设定此查询词的权重，默认是 1，如果 N 大于 1，则说明此查询词更重要，如果 N 小于 1，则说明此查询词更不重要。

如 jakarta^4 apache，"jakarta apache"^4 "Apache Lucene"

(9) 布尔操作符

布尔操作符包括连接符，如 AND，OR，和修饰符，如 NOT，+，-。

默认状态下，空格被认为是 OR 的关系，

QueryParser.setDefaultOperator(Operator.AND)设置为空格为 AND。

+表示一个查询语句是必须满足的(required)，NOT 和-表示一个查询语句是不能满足的(prohibited)。

(10) 组合

可以用括号，将查询语句进行组合，从而设定优先级。

如(jakarta OR apache) AND website

Lucene 的查询语法是由 QueryParser 来进行解析，从而生成查询对象的。

通过编译原理我们知道，解析一个语法表达式，需要经过词法分析和语法分析的过程，也即需要词法分析器和语法分析器。

QueryParser 是通过 JavaCC 来生成词法分析器和语法分析器的。

二、JavaCC 介绍

JavaCC 是一个词法分析器和语法分析器的生成器。

所谓词法分析器就是将一系列字符分成一个个的 Token，并标记 Token 的分类。

例如，对于下面的 C 语言程序：

```
int main() {
    return 0 ;
}
```

将被分成以下的 Token:

```
“int”, “ ”, “main”, “(”, “)”,
“”,“{”, “\n”, “\t”, “return”
```

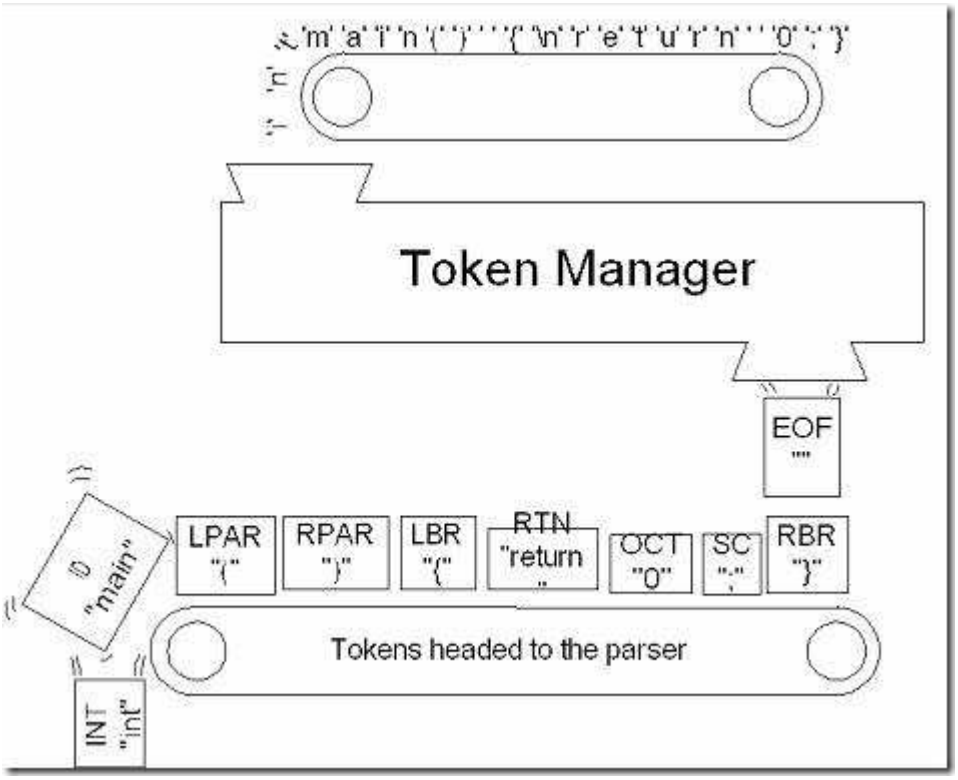
“”,“0”,“”,“;”,“\n”,
“}”, “\n”, “”

标记了 Token 的类型后如下：

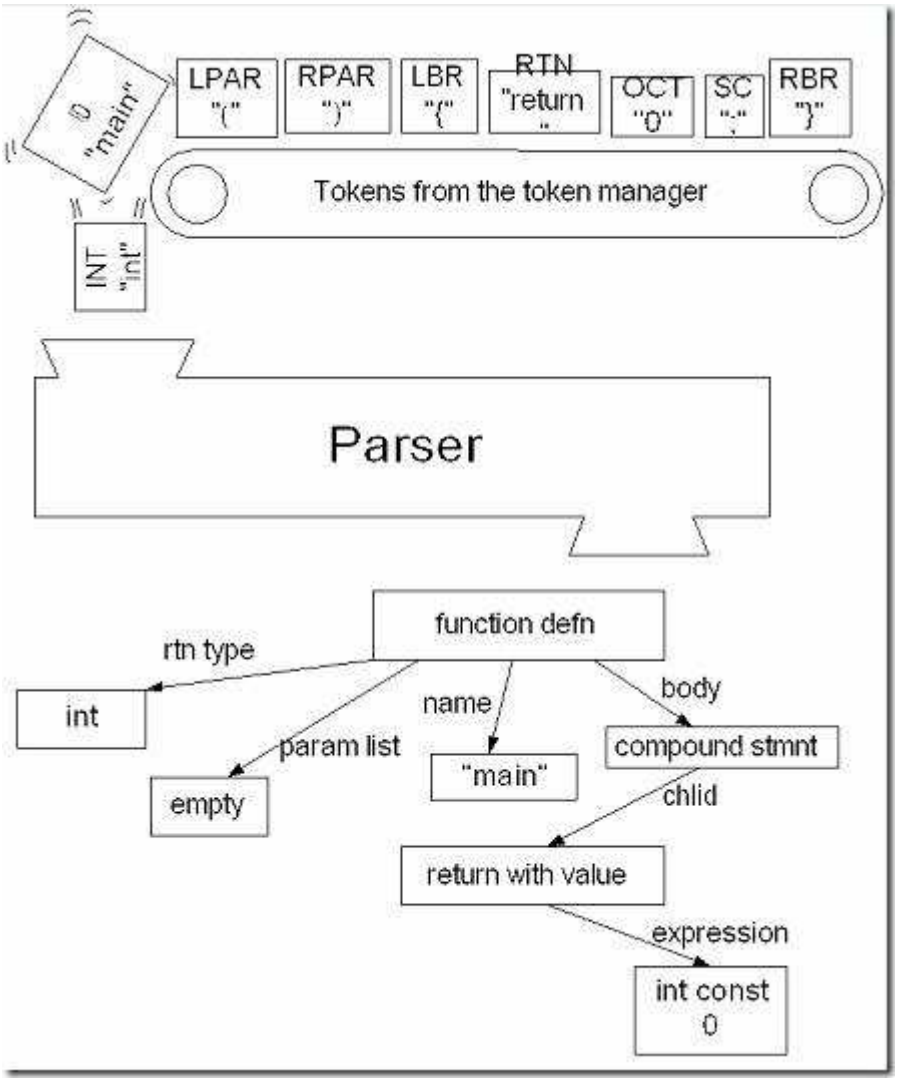
KWINT, SPACE, ID, OPAR, CPAR,
SPACE, OBRACE, SPACE, SPACE, KWRETURN,
SPACE, OCTALCONST, SPACE, SEMICOLON, SPACE,
CBRACE, SPACE, EOF

EOF 表示文件的结束。

词法分析器工作过程如图所示：



此一系列 Token 将被传给语法分析器(当然并不是所有的 Token 都会传给语法分析器，本例中 SPACE 就例外)，从而形成一棵语法分析树来表示程序的结构。



JavaCC 本身既不是一个词法分析器，也不是一个语法分析器，而是根据指定的规则生成两者的生成器。

2.1、第一个实例——正整数相加

下面我们来看第一个例子，即能够解析正整数相加的表达式，例如 99+42+0+15。

（1）生成一个 adder.jj 文件

此文件中写入的即生成词法分析器和语法分析器的规则。

（2）设定选项，并声明类

```
/* adder.jj Adding up numbers */
options {
```

```
  STATIC = false ;
}
PARSER_BEGIN(Adder)
class Adder {
  static void main( String[] args ) throws ParseException, TokenMgrError {
    Adder parser = new Adder( System.in ) ;
    parser.Start() ;
  }
}
PARSER_END(Adder)
```

STATIC 选项默认是 true，设为 false，使得生成的函数不是 static 的。

PARSER_BEGIN 和 PARSER_END 之间的 java 代码部分，此部分不需要通过 JavaCC 根据规则生成 java 代码，而是直接拷贝到生成的 java 代码中的。

(3) 声明一个词法分析器

```
SKIP : { " " }
SKIP : { "\n" | "\r" | "\r\n" }
TOKEN : { < PLUS : "+" > }
TOKEN : { < NUMBER : (["0"-"9"])+ > }
```

第一二行表示空格和回车换行是不会传给语法分析器的。

第三行声明了一个 Token，名称为 PLUS，符号为“+”。

第四行声明了一个 Token，名称为 NUMBER，符号位一个或多个 0-9 的数的组合。

如果词法分析器分析的表达式如下：

- “123 + 456\n”，则分析为 NUMBER, PLUS, NUMBER, EOF
- “123 - 456\n”，则报 TokenMgrError，因为“-”不是一个有效的 Token.
- “123 ++ 456\n”，则分析为 NUMBER, PLUS, PLUS, NUMBER, EOF，词法分析正确，后面的语法分析将会错误。

(4) 声明一个语法分析器

```
void Start() :
```

```
{}
{
  <NUMBER>
  (
   <PLUS>
   <NUMBER>
  )*
  <EOF>
}
```

语法分析器使用 BNF 表达式。

上述声明将生成 start 函数，称为 Adder 类的一个成员函数

语法分析器要求输入的语句必须以 NUMBER 开始，以 EOF 结尾，中间是零到多个 PLUS 和 NUMBER 的组合。

(5) 用 javacc 编译 adder.jj 来生成语法分析器和词法分析器

最后生成的 adder.jj 如下：

```
options
{
  static = false;
}
PARSER_BEGIN(Adder)
package org.apache.javacc;
public class Adder
{
  public static void main(String args []) throws ParseException
  {
    Adder parser = new Adder(System.in);
    parser.start();
  }
```

```
}
PARSER_END(Adder)
SKIP :
{
  " "
| "\r"
| "\t"
| "\n"
}
TOKEN : /* OPERATORS */
{
  < PLUS : "+" >
}
TOKEN :
{
  < NUMBER : ([ "0"-"9" ])+ >
}
void start() :
{}
{
  <NUMBER>
  (
    <PLUS>
    <NUMBER>
  )*
}
```

用 JavaCC 编译 adder.jj 生成如下文件：

- Adder.java：语法分析器。其中的 main 函数是完全从 adder.jj 中拷贝的，而 start 函数是

被 javacc 由 adder.jj 描述的规则生成的。

- AdderConstants.java：一些常量，如 PLUS, NUMBER, EOF 等。
- AdderTokenManager.java：词法分析器。
- ParseException.java：用于在语法分析错误的时候抛出。
- SimpleCharStream.java：用于将一系列字符串传入词法分析器。
- Token.java：代表词法分析后的一个个 Token。Token 对象有一个整型域 kind，来表示此 Token 的类型(PLUS, NUMBER, EOF)，有一个 String 类型的域 image，来表示此 Token 的值。
- TokenMgrError.java：用于在词法分析错误的时候抛出。

下面我们对 adder.jj 生成的 start 函数进行分析：

```
final public void start() throws ParseException {
  //从词法分析器取得下一个 Token，而且要求必须是 NUMBER 类型，否则抛出异常。
  //此步要求表达式第一个出现的字符必须是 NUMBER。
  jj_consume_token(NUMBER);
  label_1:
  while (true) {
   //jj_ntk()是取得下一个 Token 的类型，如果是 PLUS，则继续进行，如果是 EOF 则退出循环。
    switch ((jj_ntk==-1)?jj_ntk():jj_ntk) {
    case PLUS:
     ;
     break;
    default:
     jj_la1[0] = jj_gen;
     break label_1;
    }
   //要求下一个 PLUS 字符，再下一个是一个 NUMBER，如此下去。
    jj_consume_token(PLUS);
    jj_consume_token(NUMBER);
```

```
  }
}
```

(6) 运行 Adder.java

如果输入“123+456”则不报任何错误。

如果输入“123++456”则报如下异常：

```
Exception in thread "main" org.apache.javacc.ParseException: Encountered " "+" "+ "" at line 1, column 5.
Was expecting:
    <NUMBER> ...
    at org.apache.javacc.Adder.generateParseException(Adder.java:185)
    at org.apache.javacc.Adder.jj_consume_token(Adder.java:123)
    at org.apache.javacc.Adder.start(Adder.java:24)
    at org.apache.javacc.Adder.main(Adder.java:8)
```

如果输入“123-456”则报如下异常：

```
Exception in thread "main" org.apache.javacc.TokenMgrError: Lexical error at line 1, column 4.  Encountered: "-" (45), after : ""
    at org.apache.javacc.AdderTokenManager.getNextToken(AdderTokenManager.java:262)
    at org.apache.javacc.Adder.jj_ntk(Adder.java:148)
    at org.apache.javacc.Adder.start(Adder.java:15)
    at org.apache.javacc.Adder.main(Adder.java:8)
```

2.2、扩展语法分析器

在上面的例子中的 start 函数中，我们仅仅通过语法分析器来判断输入的语句是否正确。

我们可以扩展 BNF 表达式，加入 Java 代码，使得经过语法分析后，得到我们想要的结果或者对象。

我们将 start 函数改写为：

```
int start() throws NumberFormatException :
```

```
{
  //start 函数中有三个变量
  Token t ;
  int i ;
  int value ;
}
{
  //首先要求表达式的第一个一定是一个 NUMBER，并把其值付给 t
  t= <NUMBER>
  //将 t 的值取出来，解析为整型，放入变量 i 中
  { i = Integer.parseInt( t.image ) ; }
  //最后的结果 value 设为 i
  { value = i ; }
  //紧接着应该是零个或者多个 PLUS 和 NUMBER 的组合
  (
   <PLUS>
   //每出现一个 NUMBER，都将其付给 t，并将 t 的值解析为整型，付给 i
   t= <NUMBER>
   { i = Integer.parseInt( t.image ) ; }
   //将 i 加到 value 上
   { value += i ; }
  )*
  //最后的 value 就是表达式的和
  { return value ; }
}
```

生成的 start 函数如下：

```
final public int start() throws ParseException, NumberFormatException {
  Token t;
```

```
  int i;
  int value;
  t = jj_consume_token(NUMBER);
  i = Integer.parseInt(t.image);
  value = i;
  label_1: while (true) {
    switch ((jj_ntk == -1) ? jj_ntk() : jj_ntk) {
    case PLUS:
      ;
      break;
    default:
      jj_la1[0] = jj_gen;
      break label_1;
    }
    jj_consume_token(PLUS);
    t = jj_consume_token(NUMBER);
    i = Integer.parseInt(t.image);
    value += i;
  }
  {
    if (true)
      return value;
  }
  throw new Error("Missing return statement in function");
}
```

从上面的例子，我们发现，把一个 NUMBER 取出，并解析为整型这一步是可以共用的，所以可以抽象为一个函数：

```
int start() throws NumberFormatException :
```

```
{
  int i;
  int value ;
}
{
  value = getNextNumberValue()
  (
    <PLUS>
    i = getNextNumberValue()
    { value += i ; }
  )*
  { return value ; }
}
int getNextNumberValue() throws NumberFormatException :
{
  Token t ;
}
{
  t=<NUMBER>
  { return Integer.parseInt( t.image ) ; }
}
```

生成的函数如下：

```
final public int start() throws ParseException, NumberFormatException {
  int i;
  int value;
  value = getNextNumberValue();
  label_1: while (true) {
    switch ((jj_ntk == -1) ? jj_ntk() : jj_ntk) {
```

```
    case PLUS:
      ;
      break;
    default:
      jj_la1[0] = jj_gen;
      break label_1;
    }
    jj_consume_token(PLUS);
    i = getNextNumberValue();
    value += i;
  }
  {
    if (true)
      return value;
  }
  throw new Error("Missing return statement in function");
}
final public int getNextNumberValue() throws ParseException, NumberFormatException {
  Token t;
  t = jj_consume_token(NUMBER);
  {
    if (true)
      return Integer.parseInt(t.image);
  }
  throw new Error("Missing return statement in function");
}
```

2.3、第二个实例：计算器

（1）生成一个 calculator.jj 文件

用于写入生成计算器词法分析器和语法分析器的规则。

（2）设定选项，并声明类

```
options {
STATIC = false ;
}
PARSER_BEGIN(Calculator)
  import java.io.PrintStream ;
  class Calculator {
    static void main( String[] args ) throws ParseException, TokenMgrError, NumberFormatException
{
      Calculator parser = new Calculator( System.in ) ;
      parser.Start( System.out ) ;
    }
    double previousValue = 0.0 ;
  }
PARSER_END(Calculator)
```

previousValue 用来记录上一次计算的结果。

（3）声明一个词法分析器

```
SKIP : { " " }
TOKEN : { < EOL:"\n" | "\r" | "\r\n" > }
TOKEN : { < PLUS : "+" > }
```

我们想要支持小数，则有四种情况：没有小数，小数点在中间，小数点在前面，小数点在后面。则语法规则如下：

```
TOKEN { < NUMBER : (["0"-"9"])+ | (["0"-"9"])+ "." (["0"-"9"])+ | (["0"-"9"])+ "." | "." (["0"-"9"])+ > }
```

由于同一个表达式["0"-"9"]使用了多次，因而我们可以定义变量，如下：

```
TOKEN : { < NUMBER : <DIGITS> | <DIGITS> "." <DIGITS> | <DIGITS> "." | "." <DIGITS>> }

TOKEN : { < #DIGITS : (["0"-"9"])+ > }
```

(4) 声明一个语法分析器

我们想做的计算器包含多行，每行都是一个四则运算表达式，语法规则如下：

```
Start -> (Expression EOL)* EOF
```

```
void Start(PrintStream printStream) throws NumberFormatException :

{}

{

  (

    previousValue = Expression()

    <EOL>

    { printStream.println( previousValue ) ; }

  )*

  <EOF>

}
```

每一行的四则运算表达式如果只包含加法，则语法规则如下：

```
Expression -> Primary (PLUS Primary)*
```

```
double Expression() throws NumberFormatException :

{

  double i ;

  double value ;

}

{

  value = Primary()

  (

    <PLUS>

    i= Primary()
```

```
   { value += i ; }
  )*
  { return value ; }
}
```

其中 Primary()得到一个数的值：

```
double Primary() throws NumberFormatException :
{
  Token t ;
}
{
  t= <NUMBER>
  { return Double.parseDouble( t.image ) ; }
}
```

(5) 扩展词法分析器和语法分析器

如果我们想支持减法，则需要在词法分析器中添加：

```
TOKEN : { < MINUS : "-" > }
```

语法分析器应该变为：

```
Expression -> Primary (PLUS Primary | MINUS Primary)*
```

```
double Expression() throws NumberFormatException :
{
  double i ;
  double value ;
}
{
  value = Primary()
  (
    <PLUS>
```

```
    i = Primary()
    { value += i ; }
    |
    <MINUS>
    i = Primary()
    { value -= i ; }
  )*
  { return value ; }
}
```

如果我们想添加乘法和除法，则在词法分析器中应该加入：

```
TOKEN : { < TIMES : "*" > }
TOKEN : { < DIVIDE : "/" > }
```

对于加减乘除混合运算，则应该考虑优先级，乘除的优先级高于加减，应该先做乘除，再做加减：

```
Expression -> Term (PLUSTerm | MINUSTerm)*
Term -> Primary (TIMES Primary | DIVIDE Primary)*
```

```
double Expression() throws NumberFormatException :
{
  double i ;
  double value ;
}
{
  value = Term()
  (
    <PLUS>
    i= Term()
    { value += i ; }
    |
```

```
    <MINUS>
    i= Term()
    { value -= i ; }
   )*
   { return value ; }
}
```

```
double Term() throws NumberFormatException :
{
   double i ;
   double value ;
}
{
   value = Primary()
   (
    <TIMES>
    i = Primary()
    { value *= i ; }
    |
    <DIVIDE>
    i = Primary()
    { value /= i ; }
   )*
   { return value ; }
}
```

下面我们要开始支持括号，负号，以及取得上一行四则运算表达式的值。

对于词法分析器，我们添加如下 Token：

```
TOKEN : { < OPEN PAR : "(" > }
TOKEN : { < CLOSE PAR : ")" > }
```

```
TOKEN : { < PREVIOUS : "$" > }
```

对于语法分析器，对于最基本的表达式，有四种情况：

其可以是一个 NUMBER，也可以是上一行四则运算表达式的值 PREVIOUS，也可以是被括号括起来的一个子语法表达式，也可以是取负的一个基本语法表达式。

```
Primary –> NUMBER | PREVIOUS | OPEN_PAR Expression CLOSE_PAR | MINUS Primary
```

```
double Primary() throws NumberFormatException :
{
  Token t ;
  double d ;
}
{
  t=<NUMBER>
  { return Double.parseDouble( t.image ) ; }
  |
  <PREVIOUS>
  { return previousValue ; }
  |
  <OPEN PAR> d=Expression() <CLOSE PAR>
  { return d ; }
  |
  <MINUS> d=Primary()
  { return -d ; }
}
```

(6) 用 javacc 编译 calculator.jj 来生成语法分析器和词法分析器

最后生成的 calculator.jj 如下：

```
options
{
  static = false;
```

```
}
PARSER_BEGIN(Calculator)
package org.apache.javacc.calculater;
  import java.io.PrintStream ;
  class Calculator {
    static void main( String[] args ) throws ParseException, TokenMgrError, NumberFormatException
{
      Calculator parser = new Calculator( System.in ) ;
      parser.start( System.out ) ;
    }
    double previousValue = 0.0 ;
  }
PARSER_END(Calculator)
SKIP : { " " }
TOKEN : { < EOL: "\n" | "\r" | "\r\n" > }
TOKEN : { < PLUS : "+" > }
TOKEN : { < MINUS : "-" > }
TOKEN : { < TIMES : "*" > }
TOKEN : { < DIVIDE : "/" > }
TOKEN : { < NUMBER : <DIGITS> | <DIGITS> "." <DIGITS> | <DIGITS> "." | "." <DIGITS>> }
TOKEN : { < #DIGITS : (["0"-"9"])+ > }
TOKEN : { < OPEN_PAR : "(" > }
TOKEN : { < CLOSE_PAR : ")" > }
TOKEN : { < PREVIOUS : "$" > }
void start(PrintStream printStream) throws NumberFormatException :
{}
{
  (
```

```
    previousValue = Expression()
    { printStream.println( previousValue ) ; }
  )*
}
double Expression() throws NumberFormatException :
{
  double i ;
  double value ;
}
{
  value = Term()
  (
    <PLUS>
    i= Term()
    { value += i ; }
    |
    <MINUS>
    i= Term()
    { value -= i ; }
  )*
  { return value ; }
}
double Term() throws NumberFormatException :
{
  double i ;
  double value ;
}
{
```

```
  value = Primary()
  (
   <TIMES>
   i = Primary()
   { value *= i ; }
   |
   <DIVIDE>
   i = Primary()
   { value /= i ; }
  )*
  { return value ; }
}
double Primary() throws NumberFormatException :
{
  Token t ;
  double d ;
}
{
  t=<NUMBER>
  { return Double.parseDouble( t.image ) ; }
  |
  <PREVIOUS>
  { return previousValue ; }
  |
  <OPEN_PAR> d=Expression() <CLOSE_PAR>
  { return d ; }
  |
  <MINUS> d=Primary()
```

```
  { return -d ; }
}
```

生成的 start 函数如下：

```
final public void start(PrintStream printStream) throws ParseException, NumberFormatException {
  label_1:
  while (true) {
    switch ((jj_ntk==-1)?jj_ntk():jj_ntk) {
    case MINUS:
    case NUMBER:
    case OPEN_PAR:
    case PREVIOUS:
      ;
      break;
    default:
      jj_la1[0] = jj_gen;
      break label_1;
    }
    previousValue = Expression();
    printStream.println( previousValue ) ;
  }
}
final public double Expression() throws ParseException, NumberFormatException {
  double i ;
  double value ;
  value = Term();
  label_2:
  while (true) {
    switch ((jj_ntk==-1)?jj_ntk():jj_ntk) {
```

```
    case PLUS:
    case MINUS:
      ;
      break;
    default:
      jj_la1[1] = jj_gen;
      break label_2;
    }
    switch ((jj_ntk==-1)?jj_ntk():jj_ntk) {
    case PLUS:
      jj_consume_token(PLUS);
      i = Term();
      value += i ;
      break;
    case MINUS:
      jj_consume_token(MINUS);
      i = Term();
      value -= i ;
      break;
    default:
      jj_la1[2] = jj_gen;
      jj_consume_token(-1);
      throw new ParseException();
    }
  }
  {if (true) return value ;}
  throw new Error("Missing return statement in function");
}
```

```
final public double Term() throws ParseException, NumberFormatException {
  double i ;
  double value ;
  value = Primary();
  label_3:
  while (true) {
   switch ((jj_ntk==-1)?jj_ntk():jj_ntk) {
   case TIMES:
   case DIVIDE:
    ;
    break;
   default:
    jj_la1[3] = jj_gen;
    break label_3;
   }
   switch ((jj_ntk==-1)?jj_ntk():jj_ntk) {
   case TIMES:
    jj_consume_token(TIMES);
    i = Primary();
    value *= i ;
    break;
   case DIVIDE:
    jj_consume_token(DIVIDE);
    i = Primary();
    value /= i ;
    break;
   default:
    jj_la1[4] = jj_gen;
```

```
      jj_consume_token(-1);
      throw new ParseException();
    }
  }
  {if (true) return value ;}
  throw new Error("Missing return statement in function");
}
final public double Primary() throws ParseException, NumberFormatException {
  Token t ;
  double d ;
  switch ((jj_ntk==-1)?jj_ntk():jj_ntk) {
  case NUMBER:
    t = jj_consume_token(NUMBER);
    {if (true) return Double.parseDouble( t.image ) ;}
    break;
  case PREVIOUS:
    jj_consume_token(PREVIOUS);
    {if (true) return previousValue ;}
    break;
  case OPEN_PAR:
    jj_consume_token(OPEN_PAR);
    d = Expression();
    jj_consume_token(CLOSE_PAR);
    {if (true) return d ;}
    break;
  case MINUS:
    jj_consume_token(MINUS);
    d = Primary();
```

```
    {if (true) return -d ;}
    break;
  default:
    jj_la1[5] = jj_gen;
    jj_consume_token(-1);
    throw new ParseException();
  }
  throw new Error("Missing return statement in function");
}
```

三、解析 QueryParser.jj

3.1、声明 QueryParser 类

在 QueryParser.jj 文件中，PARSER_BEGIN(QueryParser)和 PARSER_END(QueryParser)之间，定义了 QueryParser 类。

其中最重要的一个函数是 public Query parse(String query)函数，也即我们解析 Lucene 查询语法的时候调用的函数。

这是一个纯 Java 代码定义的函数，会直接拷贝到 QueryParser.java 文件中。

parse 函数中，最重要的一行代码是调用 Query res = TopLevelQuery(field)，而 TopLevelQuery 函数是 QueryParser.jj 中定义的语法分析器被 JavaCC 编译后会生成的函数。

3.2、声明词法分析器

在解析词法分析器之前，首先介绍一下 JavaCC 的词法状态的概念(lexical state)。

有可能存在如下的情况，在不同的情况下，要求的词法词法规则不同，比如我们要解析一个 java 文件(即满足 java 语法的表达式)，在默认的状态 DEFAULT 下，是要求解析的对象(即表达式)满足 java 语言的词法规则，然而当出现"/**"的时候，其后面的表达式则不需要满足 java

语言的语法规则，而是应该满足 java 注释的语法规则(要识别@param 变量等)，于是我们做如下定义：

```
//默认处于 DEFAULT 状态，当遇到/**的时候，转换为 IN_JAVADOC_COMMENT 状态
<DEFAULT> TOKEN : {<STARTDOC : "/**" > : IN_JAVADOC_COMMENT }
//在 IN_JAVADOC_COMMENT 状态下，需要识别@param 变量
<IN_JAVADOC_COMMENT> TOKEN : {<PARAM : "@param" >}
//在 IN_JAVADOC_COMMENT 状态下，遇到*/的时候，装换为 DEFAULT 状态
<IN_JAVADOC_COMMENT> TOKEN : {<ENDDOC: "*/">: DEFAULT }
```

<*> 表示应用于任何状态。

(1) 应用于所有状态的变量

```
<*> TOKEN : {
  <#_NUM_CHAR:   ["0"-"9"] > //数字
| <#_ESCAPED_CHAR: "\\" ~[] > //"\"后的任何一个字符都是被转义的
| <#_TERM_START_CHAR: ( ~[ " ", "\t", "\n", "\r", "\u3000", "+", "-", "!", "(", ")", ":", "^", "[", "]",
"\"", "{", "}", "~", "*", "?", "\\" ] | <_ESCAPED_CHAR> ) > //表达式中任何一个 term，都不能以[]括起来的列表中的 lucene 查询语法关键字开头，当然被转义的除外。
| <#_TERM_CHAR: ( <_TERM_START_CHAR> | <_ESCAPED_CHAR> | "-" | "+" ) > //表达式中的 term 非起始字符，可以包含任何非语法关键字字符，转义过的字符，也可以包含+，-(但包含+,-的符合词法，不合语法)。
| <#_WHITESPACE: ( " " | "\t" | "\n" | "\r" | "\u3000") > //被认为是空格的字符
| <#_QUOTED_CHAR: ( ~[ "\"", "\\" ] | <_ESCAPED_CHAR> ) > //被引号括起来的字符不应再包括"和\，当然转义过的除外。
}
```

(2) 默认状态的 Token

```
<DEFAULT> TOKEN : {
  <AND:      ("AND" | "&&") >
```

| <OR: ("OR" | "||") >

| <NOT: ("NOT" | "!") >

| <PLUS: "+" >

| <MINUS: "-" >

| <LPAREN: "(" >

| <RPAREN: ")" >

| <COLON: ":" >

| <STAR: "*" >

| <CARAT: "^" > : Boost **//当遇到^的时候，后面跟随的是 boost 表达式，进入 Boost 状态**

| <QUOTED: "\"" (<_QUOTED_CHAR>)* "\"">

| <TERM: <_TERM_START_CHAR> (<_TERM_CHAR>)* >

| <FUZZY_SLOP: "~" ((<_NUM_CHAR>)+ ("." (<_NUM_CHAR>)+)?)? > **//Fuzzy 查询，~后面跟小数。**

| <PREFIXTERM: ("*") | (<_TERM_START_CHAR> (<_TERM_CHAR>)* "*") > **//使用*进行 Prefix 查询，可以尽包含*，或者末尾包含*，然而只包含*符合词法，不合语法。**

| <WILDTERM: (<_TERM_START_CHAR> | ["*", "?"]) (<_TERM_CHAR> | (["*", "?"]))* > **//使用*和?进行 wildcard 查询**

| <RANGEIN_START: "[" > : RangeIn **//遇到[]的时候，是包含边界的 Range 查询**

| <RANGEEX_START: "{" > : RangeEx **//遇到{}的时候，是不包含边界的 Range 查询**

}

<Boost> TOKEN : {

<NUMBER: (<_NUM_CHAR>)+ ("." (<_NUM_CHAR>)+)? > : DEFAULT **//boost 是一个小数**

}

//包含边界的 Range 查询是[A TO B]的形式。

<RangeIn> TOKEN : {

<RANGEIN_TO: "TO">

| <RANGEIN_END: "]"> : DEFAULT

```
| <RANGEIN_QUOTED: "\"" (~["\""] | "\\\"")+ "\"">
| <RANGEIN_GOOP: (~[ " ", "]" ])+ >
}
```

```
//不包含边界的 Range 查询是{A TO B}的形式
<RangeEx> TOKEN : {
<RANGEEX_TO: "TO">
| <RANGEEX_END: "}"> : DEFAULT
| <RANGEEX_QUOTED: "\"" (~["\""] | "\\\"")+ "\"">
| <RANGEEX_GOOP: (~[ " ", "}" ])+ >
}
```

3.3、声明语法分析器

Lucene 的语法规则如下：

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

(1) 从 Query 到 Clause

一个 Query 查询语句，是由多个 clause 组成的，每个 clause 有修饰符 Modifier，或为+, 或为-，clause 之间的有连接符，或为 AND，或为 OR，或为 NOT。

在 Lucene 的语法解析中 NOT 被算作 Modifier，和-起相同作用。

此过程表达式如下：

```
Query TopLevelQuery(String field) :
{
    Query q;
}
{
    q=Query(field) <EOF>
```

```
    {
      return q;
    }
}
```

```
Query Query(String field) :
{
  List<BooleanClause> clauses = new ArrayList<BooleanClause>();
  Query q, firstQuery=null;
  int conj, mods;
}
{
  //查询语句开头是一个 Modifier，可以为空
  //Modifier 后面便是子语句 clause，可以生成子查询语句 q
  mods=Modifiers() q=Clause(field)
  {
   //如果第一个语句的 Modifier 是空，则将子查询 q 付给 firstQuery，从后面我们可以看到，当只有一个查询语句的时候，如果其 Modifier 为空，则不返回 BooleanQuery，而是返回子查询对象 firstQuery。从这里我们可以看出，如果查询语句为"A"，则生成 TermQuery，其 term 为"A"，如果查询语句为"+A"，则生成 BooleanQuery，其子查询只有一个，就是 TermQuery，其 term 为"A"。
    addClause(clauses, CONJ_NONE, mods, q);
    if (mods == MOD_NONE)
      firstQuery=q;
  }
  (
   //除了第一个语句外，其他的前面可以有连接符，或为 AND，或为 OR。
    //如果在第一个语句之前出现连接符，则报错，如"OR a"，会报 Encountered " <OR> "OR "" at line 1, column 0.
```

```
  //除了连接符，也会有 Modifier，后面是子语句 clause，生成子查询 q，并加入 BooleanQuery 中。
   conj=Conjunction() mods=Modifiers() q=Clause(field)
   { addClause(clauses, conj, mods, q); }
  )*
  {
  //如果只有一个查询语句，且其 modifier 为空，则返回 firstQuery，否则由所有的子语句 clause，生成 BooleanQuery。
   if (clauses.size() == 1 && firstQuery != null)
    return firstQuery;
   else {
    return getBooleanQuery(clauses);
   }
  }
}
```

```
int Modifiers() : {
 //默认 modifier 为空，如果遇到+，就是 required，如果遇到-或者 NOT，就是 prohibited。
  int ret = MOD_NONE;
}
{
  [
    <PLUS> { ret = MOD_REQ; }
    | <MINUS> { ret = MOD_NOT; }
    | <NOT> { ret = MOD_NOT; }
  ]
  { return ret; }
}
```

```
//连接符
int Conjunction() : {
  int ret = CONJ_NONE;
}
{
  [
    <AND> { ret = CONJ_AND; }
    | <OR>  { ret = CONJ_OR; }
  ]
  { return ret; }
}
```

(2) 一个子语句 clause

由上面的分析我们可以知道，JavaCC 使用的是编译原理里面的自上而下分析法，基本采用的是 LL(1)的方法：

- 第一个 L ：从左到右扫描输入串
- 第二个 L ：生成的是最左推导
- (1)：向前看一个输入符号（lookahead)

JavaCC 还提供 LOOKAHEAD(n)，也即当仅读入下一个符号时，不足以判断接下来的如何解析，会出现 Choice Conflict，则需要多读入几个符号，来进一步判断。

```
Query Clause(String field) : {
  Query q;
  Token fieldToken=null, boost=null;
}
{
  //此处之所以向前看两个符号，就是当看到<TERM>的时候，不知道它是一个 field，还是一个 term，当<TERM><COLON>在一起的时候，说明<TERM>代表一个 field，否则代表一个 term
```

```
  [
    LOOKAHEAD(2)
    (
    fieldToken=<TERM> <COLON> {field=discardEscapeChar(fieldToken.image);}
    | <STAR> <COLON> {field="*";}
    )
  ]
  (
  //或者是一个 term，则由此 term 生成一个查询对象
    //或者是一个由括号括起来的子查询
    //()?表示可能存在一个 boost，格式为^加一个数字
   q=Term(field)
   | <LPAREN> q=Query(field) <RPAREN> (<CARAT> boost=<NUMBER>)?
  )
  {
    //如果存在 boost，则设定查询对象的 boost
    if (boost != null) {
      float f = (float)1.0;
      try {
        f = Float.valueOf(boost.image).floatValue();
        q.setBoost(f);
      } catch (Exception ignored) { }
    }
    return q;
  }
}
```

```
Query Term(String field) : {
  Token term, boost=null, fuzzySlop=null, goop1, goop2;
```

```
  boolean prefix = false;
  boolean wildcard = false;
  boolean fuzzy = false;
  Query q;
}
{
  (
    (
    //如果 term 仅结尾包含*则是 prefix 查询。
      //如果以*开头，或者中间包含*，或者结尾包含*(如果仅结尾包含，则 prefix 优先)
则为 wildcard 查询。
     term=<TERM>
     | term=<STAR> { wildcard=true; }
     | term=<PREFIXTERM> { prefix=true; }
     | term=<WILDTERM> { wildcard=true; }
     | term=<NUMBER>
    )
   //如果 term 后面是~，则是 fuzzy 查询
    [ fuzzySlop=<FUZZY_SLOP> { fuzzy=true; } ]
    [ <CARAT> boost=<NUMBER> [ fuzzySlop=<FUZZY_SLOP> { fuzzy=true; } ] ]
    {
     //如果是 wildcard 查询，则调用 getWildcardQuery，
       //    *:*得到 MatchAllDocsQuery，将返回所有的文档
       //    目前不支持最前面带通配符的查询(虽然词法分析和语法分析都能通过)，否则
报 ParseException
       //    最后生成 WildcardQuery
       //如果是 prefix 查询，则调用 getPrefixQuery，生成 PrefixQuery
       //如果是 fuzzy 查询，则调用 getFuzzyQuery,生成 FuzzyQuery
```

```
      //如果是普通查询，则调用 getFieldQuery
      String termImage=discardEscapeChar(term.image);
      if (wildcard) {
       q = getWildcardQuery(field, termImage);
      } else if (prefix) {
              q = getPrefixQuery(field, discardEscapeChar(term.image.substring(0,
term.image.length()-1)));
      } else if (fuzzy) {
       float fms = fuzzyMinSim;
       try {
        fms = Float.valueOf(fuzzySlop.image.substring(1)).floatValue();
       } catch (Exception ignored) { }
       if(fms < 0.0f || fms > 1.0f){
        throw new ParseException("Minimum similarity for a FuzzyQuery has to be between 0.0f
and 1.0f !");
       }
       q = getFuzzyQuery(field, termImage,fms);
      } else {
       q = getFieldQuery(field, termImage);
      }
     }
   //包含边界的 range 查询，取得[goop1 TO goop2]，调用 getRangeQuery，
生成 TermRangeQuery
    | ( <RANGEIN_START> ( goop1=<RANGEIN_GOOP>|goop1=<RANGEIN_QUOTED> )
      [ <RANGEIN_TO> ] ( goop2=<RANGEIN_GOOP>|goop2=<RANGEIN_QUOTED> )
      <RANGEIN_END> )
     [ <CARAT> boost=<NUMBER> ]
      {
```

```
        if (goop1.kind == RANGEIN_QUOTED) {
          goop1.image = goop1.image.substring(1, goop1.image.length()-1);
        }
        if (goop2.kind == RANGEIN_QUOTED) {
          goop2.image = goop2.image.substring(1, goop2.image.length()-1);
        }
                q          =          getRangeQuery(field,          discardEscapeChar(goop1.image),
discardEscapeChar(goop2.image), true);
      }
```

//不包含边界的 range 查询，取得{goop1 TO goop2}，调用 getRangeQuery，生成 TermRangeQuery

```
    | ( <RANGEEX_START> ( goop1=<RANGEEX_GOOP>|goop1=<RANGEEX_QUOTED> )
       [ <RANGEEX_TO> ] ( goop2=<RANGEEX_GOOP>|goop2=<RANGEEX_QUOTED> )
       <RANGEEX_END> )
      [ <CARAT> boost=<NUMBER> ]
      {
        if (goop1.kind == RANGEEX_QUOTED) {
          goop1.image = goop1.image.substring(1, goop1.image.length()-1);
        }
        if (goop2.kind == RANGEEX_QUOTED) {
          goop2.image = goop2.image.substring(1, goop2.image.length()-1);
        }
                q          =          getRangeQuery(field,          discardEscapeChar(goop1.image),
discardEscapeChar(goop2.image), false);
      }
```

//被""括起来的 term，得到 phrase 查询，调用 getFieldQuery

```
    | term=<QUOTED>
      [ fuzzySlop=<FUZZY_SLOP> ]
```

```
    [ <CARAT> boost=<NUMBER> ]
    {
      int s = phraseSlop;
      if (fuzzySlop != null) {
        try {
          s = Float.valueOf(fuzzySlop.image.substring(1)).intValue();
        }
        catch (Exception ignored) { }
      }
      q = getFieldQuery(field, discardEscapeChar(term.image.substring(1, term.image.length()-1)),
s);
    }
  )
  {
   if (boost != null) {
    float f = (float) 1.0;
    try {
     f = Float.valueOf(boost.image).floatValue();
    }
    catch (Exception ignored) {
    }
    // avoid boosting null queries, such as those caused by stop words
    if (q != null) {
     q.setBoost(f);
    }
   }
   return q;
  }
```

```
}
```

此处需要详细解析的是 getFieldQuery：

```
protected Query getFieldQuery(String field, String queryText)   throws ParseException {
  //需要用 analyzer 对文本进行分词
  TokenStream source;
  try {
    source = analyzer.reusableTokenStream(field, new StringReader(queryText));
    source.reset();
  } catch (IOException e) {
    source = analyzer.tokenStream(field, new StringReader(queryText));
  }
  CachingTokenFilter buffer = new CachingTokenFilter(source);
  TermAttribute termAtt = null;
  PositionIncrementAttribute posIncrAtt = null;
  int numTokens = 0;
  boolean success = false;
  try {
    buffer.reset();
    success = true;
  } catch (IOException e) {
  }
  //得到 TermAttribute 和 PositionIncrementAttribute，此两项将决定到底产生什么样的 Query 对象
  if (success) {
    if (buffer.hasAttribute(TermAttribute.class)) {
      termAtt = buffer.getAttribute(TermAttribute.class);
    }
```

```
    if (buffer.hasAttribute(PositionIncrementAttribute.class)) {
      posIncrAtt = buffer.getAttribute(PositionIncrementAttribute.class);
    }
  }
  int positionCount = 0;
  boolean severalTokensAtSamePosition = false;
  boolean hasMoreTokens = false;
  if (termAtt != null) {
    try {
      //遍历分词后的所有 Token，统计 Tokens 的个数 numTokens，以及 positionIncrement 的总数，即 positionCount。
      // 当有一次 positionIncrement 为 0 的时候，severalTokensAtSamePosition 设为 true，表示有多个 Token 处在同一个位置。
      hasMoreTokens = buffer.incrementToken();
      while (hasMoreTokens) {
        numTokens++;
        int positionIncrement = (posIncrAtt != null) ? posIncrAtt.getPositionIncrement() : 1;
        if (positionIncrement != 0) {
          positionCount += positionIncrement;
        } else {
          severalTokensAtSamePosition = true;
        }
        hasMoreTokens = buffer.incrementToken();
      }
    } catch (IOException e) {
    }
  }
  try {
```

```
  //重设 buffer，以便生成 phrase 查询的时候，term 和 position 可以重新遍历。
   buffer.reset();
   source.close();
  }
  catch (IOException e) {
  }
  if (numTokens == 0)
   return null;
  else if (numTokens == 1) {
  //如果分词后只有一个 Token，则生成 TermQuery
   String term = null;
   try {
    boolean hasNext = buffer.incrementToken();
    term = termAtt.term();
   } catch (IOException e) {
   }
   return newTermQuery(new Term(field, term));
  } else {
  //如果分词后不只有一个 Token
   if (severalTokensAtSamePosition) {
  //如果有多个 Token 处于同一个位置
    if (positionCount == 1) {
     //并且处于同一位置的 Token 还全部处于第一个位置，则生成 BooleanQuery，
处于同一位置的 Token 之间是 OR 的关系
      BooleanQuery q = newBooleanQuery(true);
      for (int i = 0; i < numTokens; i++) {
       String term = null;
       try {
```

```
        boolean hasNext = buffer.incrementToken();
        term = termAtt.term();
      } catch (IOException e) {
      }
      Query currentQuery = newTermQuery(new Term(field, term));
      q.add(currentQuery, BooleanClause.Occur.SHOULD);
    }
    return q;
  }
  else {
```

//如果有多个 Token 处于同一位置，但不是第一个位置，则生成 MultiPhraseQuery。

//所谓 MultiPhraseQuery 即其可以包含多个 phrase，其又一个 ArrayList<Term[]> termArrays，每一项都是一个 Term 的数组，属于同一个数组的 Term 表示在同一个位置。它有函数 void add(Term[] terms)一次添加一个数组的 Term。比如我们要搜索"microsoft app*"，其表示多个 phrase，"microsoft apple"，"microsoft application"都算。此时用 QueryParser.parse("\"microsoft app*\"")从而生成 PhraseQuery 是搜不出 microsoft apple 和 microsoft application 的，仅能搜出 microsoft app。所以必须生成 MultiPhraseQuery，首先用 add(new Term[]{new Term("field", "microsoft")})将 microsoft 作为一个 Term 数组添加进去，然后用 add(new Term[]{new Term("field", "app"), new Term("field", "apple"), new Term("field", "application")})作为一个 Term 数组添加进去(算作同一个位置的)，则三者都能搜的出来。

```
    MultiPhraseQuery mpq = newMultiPhraseQuery();
    mpq.setSlop(phraseSlop);
    List<Term> multiTerms = new ArrayList<Term>();
    int position = -1;
```

```
      for (int i = 0; i < numTokens; i++) {
        String term = null;
        int positionIncrement = 1;
        try {
          boolean hasNext = buffer.incrementToken();
          assert hasNext == true;
          term = termAtt.term();
          if (posIncrAtt != null) {
            positionIncrement = posIncrAtt.getPositionIncrement();
          }
        } catch (IOException e) {
        }
        if (positionIncrement > 0 && multiTerms.size() > 0) {
          //如果 positionIncrement 大于零，说明此 Term 和前一个 Term 已经不是同一个位置了，所以原来收集在 multiTerms 中的 Term 都算作同一个位置，添加到 MultiPhraseQuery 中作为一项。并清除 multiTerms，以便重新收集相同位置的 Term。
          if (enablePositionIncrements) {
            mpq.add(multiTerms.toArray(new Term[0]),position);
          } else {
            mpq.add(multiTerms.toArray(new Term[0]));
          }
          multiTerms.clear();
        }
        //将此 Term 收集到 multiTerms 中。
        position += positionIncrement;
        multiTerms.add(new Term(field, term));
      }
```

```
    //当遍历完所有的Token，同处于最后一个位置的Term已经收集到multiTerms中了，把他们加到MultiPhraseQuery中作为一项。
      if (enablePositionIncrements) {
        mpq.add(multiTerms.toArray(new Term[0]),position);
      } else {
        mpq.add(multiTerms.toArray(new Term[0]));
      }
      return mpq;
     }
    }
    else {
    //如果不存在多个Token处于同一个位置的情况，则直接生成PhraseQuery
     PhraseQuery pq = newPhraseQuery();
     pq.setSlop(phraseSlop);
     int position = -1;
     for (int i = 0; i < numTokens; i++) {
      String term = null;
      int positionIncrement = 1;
      try {
        boolean hasNext = buffer.incrementToken();
        assert hasNext == true;
        term = termAtt.term();
        if (posIncrAtt != null) {
          positionIncrement = posIncrAtt.getPositionIncrement();
        }
      } catch (IOException e) {
      }
      if (enablePositionIncrements) {
```

```
        position += positionIncrement;
        pq.add(new Term(field, term),position);
       } else {
        pq.add(new Term(field, term));
       }
     }
     return pq;
   }
  }
}
```

第九章：Lucene 的查询对象

Lucene 除了支持查询语法以外，还可以自己构造查询对象进行搜索。

从上一节的 Lucene 的语法一章可以知道，能与查询语句对应的查询对象有：BooleanQuery，FuzzyQuery， MatchAllDocsQuery， MultiTermQuery， MultiPhraseQuery， PhraseQuery，PrefixQuery，TermRangeQuery，TermQuery，WildcardQuery。

Lucene 还支持一些查询对象并没有查询语句与之对应，但是能够实现相对高级的功能，本节主要讨论这些高级的查询对象。

它们中间最主要的一些层次结构如下，我们将一一解析。

Query

- BoostingQuery
- CustomScoreQuery
- MoreLikeThisQuery
- MultiTermQuery
  - NumericRangeQuery<T>
  - TermRangeQuery
- SpanQuery
  - FieldMaskingSpanQuery
  - SpanFirstQuery
  - SpanNearQuery
    - PayloadNearQuery
  - SpanNotQuery
  - SpanOrQuery
  - SpanRegexQuery
  - SpanTermQuery
    - PayloadTermQuery
- FilteredQuery

1、BoostingQuery

BoostingQuery 包含三个成员变量：

- Query match：这是结果集必须满足的查询对象

- Query context：此查询对象不对结果集产生任何影响，仅在当文档包含 context 查询的时候，将文档打分乘上 boost
- float boost

在 BoostingQuery 构造函数中：

```
public BoostingQuery(Query match, Query context, float boost) {
    this.match = match;
    this.context = (Query)context.clone();
    this.boost = boost;
    this.context.setBoost(0.0f);
}
```

在 BoostingQuery 的 rewrite 函数如下：

```
public Query rewrite(IndexReader reader) throws IOException {
  BooleanQuery result = new BooleanQuery() {
   @Override
   public Similarity getSimilarity(Searcher searcher) {
    return new DefaultSimilarity() {
     @Override
     public float coord(int overlap, int max) {
      switch (overlap) {
      case 1:
       return 1.0f;
      case 2:
       return boost;
      default:
       return 0.0f;
      }
     }
    };
```

```
    }
  };
  result.add(match, BooleanClause.Occur.MUST);
  result.add(context, BooleanClause.Occur.SHOULD);
  return result;
}
```

由上面实现可知，BoostingQuery 最终生成一个 BooleanQuery，第一项是 match 查询，是 MUST，即 required，第二项是 context 查询，是 SHOULD，即 optional

然而由查询过程分析可得，即便是 optional 的查询，也会影响整个打分。

所以在 BoostingQuery 的构造函数中，设定 context 查询的 boost 为零，则无论文档是否包含 context 查询，都不会影响最后的打分。

在 rewrite 函数中，重载了 DefaultSimilarity 的 coord 函数，当仅包含 match 查询的时候，其返回 1，当既包含 match 查询，又包含 context 查询的时候，返回 boost，也即会在最后的打分中乘上 boost 的值。

下面我们做实验如下：

```
索引如下文件：
file01: apple other other other boy
file02: apple apple other other other
file03: apple apple apple other other
file04: apple apple apple apple other
对于如下查询(1)：
TermQuery must = new TermQuery(new Term("contents","apple"));
TermQuery context = new TermQuery(new Term("contents","boy"));
BoostingQuery query = new BoostingQuery(must, context, 1f);
或者如下查询(2)：
TermQuery query = new TermQuery(new Term("contents","apple"));
两者的结果是一样的，如下：
docid : 3 score : 0.67974937
```

docid : 2 score : 0.58868027

docid : 1 score : 0.4806554

docid : 0 score : 0.33987468

自然是包含 apple 越多的文档打分越高。

然而他们的打分计算过程却不同，用 explain 得到查询(1)打分细节如下：

docid : 0 score : 0.33987468

0.33987468 = (MATCH) fieldWeight(contents:apple in 0), product of:

1.0 = tf(termFreq(contents:apple)=1)

0.7768564 = idf(docFreq=4, maxDocs=4)

0.4375 = fieldNorm(field=contents, doc=0)

explain 得到的查询(2)的打分细节如下：

docid : 0 score : 0.33987468

0.33987468 = (MATCH) sum of:

0.33987468 = (MATCH) fieldWeight(contents:apple in 0), product of:

1.0 = tf(termFreq(contents:apple)=1)

0.7768564 = idf(docFreq=4, maxDocs=4)

0.4375 = fieldNorm(field=contents, doc=0)

0.0 = (MATCH) weight(contents:boy^0.0 in 0), product of:

0.0 = queryWeight(contents:boy^0.0), product of:

0.0 = boost

1.6931472 = idf(docFreq=1, maxDocs=4)

1.2872392 = queryNorm

0.74075186 = (MATCH) fieldWeight(contents:boy in 0), product of:

1.0 = tf(termFreq(contents:boy)=1)

1.6931472 = idf(docFreq=1, maxDocs=4)

0.4375 = fieldNorm(field=contents, doc=0)

可以知道，查询(2)中，boy 的部分是计算了的，但是由于 boost 为 0 被忽略了。

让我们改变 boost，将包含 boy 的文档打分乘以 10：

```
TermQuery must = new TermQuery(new Term("contents","apple"));
TermQuery context = new TermQuery(new Term("contents","boy"));
BoostingQuery query = new BoostingQuery(must, context, 10f);
结果如下：
docid : 0 score : 3.398747
docid : 3 score : 0.67974937
docid : 2 score : 0.58868027
docid : 1 score : 0.4806554
explain 得到的打分细节如下：
docid : 0 score : 3.398747
3.398747 = (MATCH) product of:
  0.33987468 = (MATCH) sum of:
    0.33987468 = (MATCH) fieldWeight(contents:apple in 0), product of:
      1.0 = tf(termFreq(contents:apple)=1)
      0.7768564 = idf(docFreq=4, maxDocs=4)
      0.4375 = fieldNorm(field=contents, doc=0)
    0.0 = (MATCH) weight(contents:boy^0.0 in 0), product of:
      0.0 = queryWeight(contents:boy^0.0), product of:
        0.0 = boost
        1.6931472 = idf(docFreq=1, maxDocs=4)
        1.2872392 = queryNorm
      0.74075186 = (MATCH) fieldWeight(contents:boy in 0), product of:
        1.0 = tf(termFreq(contents:boy)=1)
        1.6931472 = idf(docFreq=1, maxDocs=4)
        0.4375 = fieldNorm(field=contents, doc=0)
  10.0 = coord(2/2)
```

2、CustomScoreQuery

CustomScoreQuery 主要包含以下成员变量：

- Query subQuery：子查询
- ValueSourceQuery[] valSrcQueries：其他信息源

ValueSourceQuery 主要包含 ValueSource valSrc 成员变量，其代表一个信息源。

ValueSourceQuery 会在查询过程中生成 ValueSourceWeight 并最终生成 ValueSourceScorer，

ValueSourceScorer 在 score 函数如下：

```
public float score() throws IOException {
  return qWeight * vals.floatVal(termDocs.doc());
}
```

其中 vals = valSrc.getValues(reader)类型为 DocValues，也即可以根据文档号得到值。

也即 CustomScoreQuery 会根据子查询和其他的信息源来共同决定最后的打分，而且公式可以自己实现，以下是默认实现：

```
public float customScore(int doc, float subQueryScore, float valSrcScores[]) {
  if (valSrcScores.length == 1) {
    return customScore(doc, subQueryScore, valSrcScores[0]);
  }
  if (valSrcScores.length == 0) {
    return customScore(doc, subQueryScore, 1);
  }
  float score = subQueryScore;
  for(int i = 0; i < valSrcScores.length; i++) {
    score *= valSrcScores[i];
  }
  return score;
}
```

一般是什么样的信息源会对文档的打分有影响的？

比如说文章的作者，可能被保存在 Field 当中，我们可以认为名人的文章应该打分更高，所以可以根据此 Field 的值来影响文档的打分。

然而我们知道，如果对每一个文档号都用 reader 读取域的值会影响速度，所以 Lucene 引入了 FieldCache 来进行缓存，而 FieldCache 并非在存储域中读取，而是在索引域中读取，从而不必构造 Document 对象，然而要求此索引域是不分词的，有且只有一个 Token。

所以有 FieldCacheSource 继承于 ValueSource，而大多数的信息源都继承于 FieldCacheSource，其最重要的一个函数即：

```
public final DocValues getValues(IndexReader reader) throws IOException {
  return getCachedFieldValues(FieldCache.DEFAULT, field, reader);
}
```

我们举 ByteFieldSource 为例，其 getCachedFieldValues 函数如下：

```
public DocValues getCachedFieldValues (FieldCache cache, String field, IndexReader reader) throws IOException {
  final byte[] arr = cache.getBytes(reader, field, parser);
  return new DocValues() {
    @Override
    public float floatVal(int doc) {
     return (float) arr[doc];
    }
    @Override
    public   int intVal(int doc) {
     return arr[doc];
    }
    @Override
    public String toString(int doc) {
     return   description() + '=' + intVal(doc);
    }
    @Override
```

```
    Object getInnerArray() {
      return arr;
    }
  };
}
```

其最终可以用 DocValues 根据文档号得到一个 float 值，并影响打分。

还用作者的例子，假设我们给每一个作者一个 float 的评级分数，保存在索引域中，用 CustomScoreQuery 可以将此评级融入到打分中去。

FieldScoreQuery 即是 ValueSourceQuery 的一个实现。

举例如下：

索引如下文件：

file01: apple other other other boy

file02: apple apple other other other

file03: apple apple apple other other

file04: apple apple apple apple other

在索引过程中，对 file01 的"scorefield"域中索引"10"，而其他的文件"scorefield"域中索引"1"，代码如下：

```
Document doc = new Document();
doc.add(new Field("contents", new FileReader(file)));
if(file.getName().contains("01")){
  doc.add(new Field("scorefield", "10", Field.Store.NO, Field.Index.NOT_ANALYZED));
} else {
  doc.add(new Field("scorefield", "1", Field.Store.NO, Field.Index.NOT_ANALYZED));
}
writer.addDocument(doc);
```

对于建好的索引，如果进行如下查询 TermQuery query = new TermQuery(new Term("contents", "apple"));

则得到如下结果：

docid : 3 score : 0.67974937

docid : 2 score : 0.58868027

docid : 1 score : 0.4806554

docid : 0 score : 0.33987468

自然是包含"apple"多的文档打分较高。

然而如果使用 CustomScoreQuery 进行查询：

```
TermQuery subquery = new TermQuery(new Term("contents","apple"));
FieldScoreQuery scorefield = new FieldScoreQuery("scorefield", FieldScoreQuery.Type.BYTE);
CustomScoreQuery query = new CustomScoreQuery(subquery, scorefield);
```

则得到如下结果：

docid : 0 score : 1.6466033

docid : 3 score : 0.32932067

docid : 2 score : 0.28520006

docid : 1 score : 0.23286487

显然文档 0 因为设置了数据源评分为 10 而跃居首位。

如果进行 explain，我们可以看到，对于普通的查询，文档 0 的打分细节如下：

docid : 0 score : 0.33987468

0.33987468 = (MATCH) fieldWeight(contents:apple in 0), product of:

1.0 = tf(termFreq(contents:apple)=1)

0.7768564 = idf(docFreq=4, maxDocs=4)

0.4375 = fieldNorm(field=contents, doc=0)

如果对于 CustomScoreQuery，文档 0 的打分细节如下：

docid : 0 score : 1.6466033

1.6466033 = (MATCH) custom(contents:apple, byte(scorefield)), product of:

1.6466033 = custom score: product of:

0.20850874 = (MATCH) weight(contents:apple in 0), product of:

0.6134871 = queryWeight(contents:apple), product of:

0.7768564 = idf(docFreq=4, maxDocs=4)

```
    0.7897047 = queryNorm
  0.33987468 = (MATCH) fieldWeight(contents:apple in 0), product of:
    1.0 = tf(termFreq(contents:apple)=1)
    0.7768564 = idf(docFreq=4, maxDocs=4)
    0.4375 = fieldNorm(field=contents, doc=0)
 7.897047 = (MATCH) byte(scorefield), product of:
  10.0 = byte(scorefield)=10
  1.0 = boost
  0.7897047 = queryNorm
1.0 = queryBoost
```

3、MoreLikeThisQuery

在分析 MoreLikeThisQuery 之前，首先介绍一下 MoreLikeThis。

在实现搜索应用的时候，时常会遇到"更多相似文章"，"更多相关问题"之类的需求，也即根据当前文档的文本内容，在索引库中查询相类似的文章。

我们可以使用 MoreLikeThis 实现此功能：

```
IndexReader reader = IndexReader.open(……);
IndexSearcher searcher = new IndexSearcher(reader);
MoreLikeThis mlt = new MoreLikeThis(reader);
Reader target = ... //此是一个 io reader，指向当前文档的文本内容。
Query query = mlt.like( target); //根据当前的文本内容，生成查询对象。
Hits hits = searcher.search(query); //查询得到相似文档的结果。
```

MoreLikeThis 的 Query like(Reader r)函数如下：

```
public Query like(Reader r) throws IOException {
    return createQuery(retrieveTerms(r)); //其首先从当前文档的文本内容中抽取 term，然后利用这些 term 构建一个查询对象。
}
```

```
public PriorityQueue <Object[]> retrieveTerms(Reader r) throws IOException {
      Map<String,Int> words = new HashMap<String,Int>();
      //根据不同的域中抽取 term，到底根据哪些域抽取，可用函数 void setFieldNames(String[] fieldNames)设定。
      for (int i = 0; i < fieldNames.length; i++) {
         String fieldName = fieldNames[i];
         addTermFrequencies(r, words, fieldName);
      }
      //将抽取的 term 放入优先级队列中
      return createQueue(words);
}
```

```
private void addTermFrequencies(Reader r, Map<String,Int> termFreqMap, String fieldName)
throws IOException
{
     //首先对当前的文本进行分词，分词器可以由 void setAnalyzer(Analyzer analyzer)设定。
     TokenStream ts = analyzer.tokenStream(fieldName, r);
      int tokenCount=0;
      TermAttribute termAtt = ts.addAttribute(TermAttribute.class);
     //遍历分好的每一个词
      while (ts.incrementToken()) {
         String word = termAtt.term();
         tokenCount++;
         //如果分词后的 term 的数量超过某个设定的值，则停止，可由 void setMaxNumTokensParsed(int i)设定。
         if(tokenCount>maxNumTokensParsed)
         {
            break;
```

```
        }
        //如果此词小于最小长度，或者大于最大长度，或者属于停词，则属于干扰词。
            //最小长度由 void setMinWordLen(int minWordLen)设定。
            //最大长度由 void setMaxWordLen(int maxWordLen)设定。
            //停词表由 void setStopWords(Set<?> stopWords)设定。
        if(isNoiseWord(word)){
          continue;
        }
        //  统计词频 tf
        Int cnt = termFreqMap.get(word);
        if (cnt == null) {
          termFreqMap.put(word, new Int());
        }
        else {
          cnt.x++;
        }
      }
}
```

```
private PriorityQueue createQueue(Map<String,Int> words) throws IOException {
   //根据统计的 term 及词频构造优先级队列。
   int numDocs = ir.numDocs();
   FreqQ res = new FreqQ(words.size()); //  优先级队列，将按 tf*idf 排序
   Iterator<String> it = words.keySet().iterator();
   //遍历每一个词
   while (it.hasNext()) {
     String word = it.next();
     int tf = words.get(word).x;
     //如果词频小于最小词频，则忽略此词，最小词频可由 void setMinTermFreq(int
```

minTermFreq)设定。

```
if (minTermFreq > 0 && tf < minTermFreq) {
   continue;
}
```

//遍历所有域，得到包含当前词，并且拥有最大的 **doc frequency** 的域

```
String topField = fieldNames[0];
int docFreq = 0;
for (int i = 0; i < fieldNames.length; i++) {
   int freq = ir.docFreq(new Term(fieldNames[i], word));
   topField = (freq > docFreq) ? fieldNames[i] : topField;
   docFreq = (freq > docFreq) ? freq : docFreq;
}
```

//如果文档频率小于最小文档频率，则忽略此词。最小文档频率可由 **void setMinDocFreq(int minDocFreq)**设定。

```
if (minDocFreq > 0 && docFreq < minDocFreq) {
   continue;
}
```

//如果文档频率大于最大文档频率，则忽略此词。最大文档频率可由 **void setMaxDocFreq(int maxFreq)**设定。

```
if (docFreq > maxDocFreq) {
   continue;
}
if (docFreq == 0) {
   continue;
}
```

//计算打分 **tf*idf**

```
float idf = similarity.idf(docFreq, numDocs);
float score = tf * idf;
```

```
      //将 object 的数组放入优先级队列，只有前三项有用，按照第三项 score 排序。
      res.insertWithOverflow(new Object[]{word,              // 词
                        topField,          // 域
                        Float.valueOf(score),     // 打分
                        Float.valueOf(idf),       // idf
                        Integer.valueOf(docFreq),  // 文档频率
                        Integer.valueOf(tf) //词频
      });
    }
    return res;
}
```

```
private Query createQuery(PriorityQueue q) {
    //最后生成的是一个布尔查询
    BooleanQuery query = new BooleanQuery();
    Object cur;
    int qterms = 0;
    float bestScore = 0;
    //不断从队列中优先取出打分最高的词
    while (((cur = q.pop()) != null)) {
      Object[] ar = (Object[]) cur;
      TermQuery tq = new TermQuery(new Term((String) ar[1], (String) ar[0]));
      if (boost) {
        if (qterms == 0) {
          //第一个词的打分最高，作为 bestScore
          bestScore = ((Float) ar[2]).floatValue();
        }
        float myScore = ((Float) ar[2]).floatValue();
        //其他的词的打分除以最高打分，乘以 boostFactor，得到相应的词所生成的查询
```

```
的 boost，从而在当前文本文档中打分越高的词在查询语句中也有更高的 boost，起重要的作用。
          tq.setBoost(boostFactor * myScore / bestScore);
        }
        try {
          query.add(tq, BooleanClause.Occur.SHOULD);
        }
        catch (BooleanQuery.TooManyClauses ignore) {
          break;
        }
        qterms++;
        //如果超过了设定的最大的查询词的数目，则停止，最大查询词的数目可由 void setMaxQueryTerms(int maxQueryTerms)设定。
        if (maxQueryTerms > 0 && qterms >= maxQueryTerms) {
          break;
        }
      }
      return query;
}
```

MoreLikeThisQuery 只是 MoreLikeThis 的封装，其包含了 MoreLikeThis 所需要的参数，并在 rewrite 的时候，由 MoreLikeThis.like 生成查询对象。

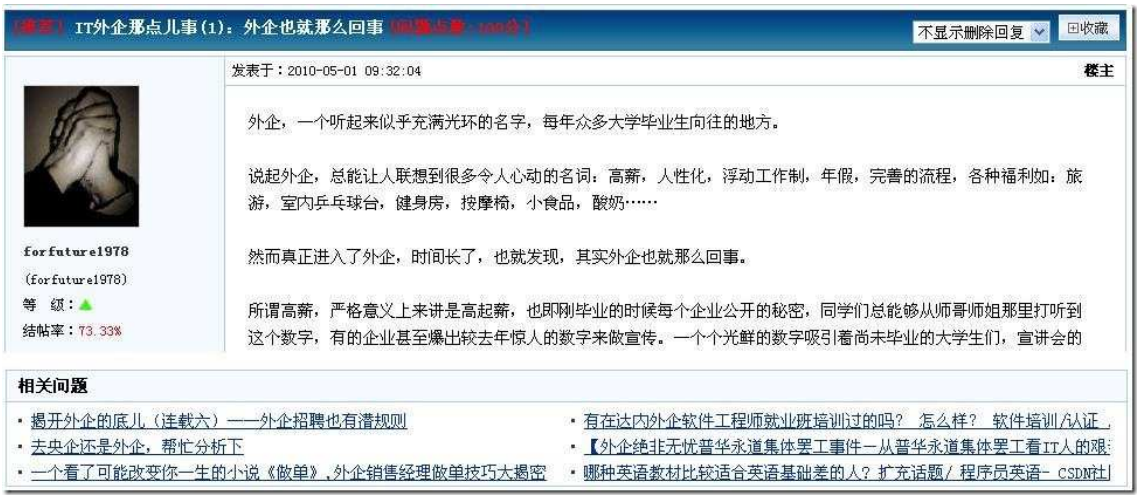
- String likeText;当前文档的文本
- String[] moreLikeFields;根据哪个域来抽取查询词
- Analyzer analyzer;分词器
- float percentTermsToMatch=0.3f;最后生成的 BooleanQuery 之间都是 SHOULD 的关系，其中至少有多少比例必须得到满足
- int minTermFrequency=1;最少的词频
- int maxQueryTerms=5;最多的查询词数目

- Set<?> stopWords=null;停词表
- int minDocFreq=-1;最小的文档频率

```
public Query rewrite(IndexReader reader) throws IOException
{
    MoreLikeThis mlt=new MoreLikeThis(reader);
    mlt.setFieldNames(moreLikeFields);
    mlt.setAnalyzer(analyzer);
    mlt.setMinTermFreq(minTermFrequency);
    if(minDocFreq>=0)
    {
       mlt.setMinDocFreq(minDocFreq);
    }
    mlt.setMaxQueryTerms(maxQueryTerms);
    mlt.setStopWords(stopWords);
                BooleanQuery bq= (BooleanQuery) mlt.like(new
ByteArrayInputStream(likeText.getBytes()));
    BooleanClause[] clauses = bq.getClauses();
    bq.setMinimumNumberShouldMatch((int)(clauses.length*percentTermsToMatch));
    return bq;
}
```

举例，对于

http://topic.csdn.net/u/20100501/09/64e41f24-e69a-40e3-9058-17487e4f311b.html?1469 中的帖子



我们姑且将相关问题中的帖子以及其他共 20 篇文档索引。

```
File indexDir = new File("TestMoreLikeThisQuery/index");
IndexReader reader = IndexReader.open(indexDir);
IndexSearcher searcher = new IndexSearcher(reader);
//将《IT 外企那点儿事》作为 likeText，从文件读入。
StringBuffer contentBuffer = new StringBuffer();
    BufferedReader input = new BufferedReader(new InputStreamReader(new FileInputStream("TestMoreLikeThisQuery/IT 外企那点儿事.txt"), "utf-8"));
String line = null;
while((line = input.readLine()) != null){
 contentBuffer.append(line);
}
String content = contentBuffer.toString();
//分词用中科院分词
MoreLikeThisQuery query = new MoreLikeThisQuery(content, new String[]{"contents"}, new MyAnalyzer(new ChineseAnalyzer()));
//将 80%都包括的词作为停词，在实际应用中，可以有其他的停词策略。
query.setStopWords(getStopWords(reader));
//至少包含 5 个的词才认为是重要的
```

```
    query.setMinTermFrequency(5);
    //只取其中之一
    query.setMaxQueryTerms(1);
    TopDocs docs = searcher.search(query, 50);
    for (ScoreDoc doc : docs.scoreDocs) {
      Document ldoc = reader.document(doc.doc);
      String title = ldoc.get("title");
      System.out.println(title);
    }
```

```
static Set<String> getStopWords(IndexReader reader) throws IOException{
  HashSet<String> stop = new HashSet<String>();
  int numOfDocs = reader.numDocs();
  int stopThreshhold = (int) (numOfDocs*0.7f);
  TermEnum te = reader.terms();
  while(te.next()){
    String text = te.term().text();
    if(te.docFreq() >= stopThreshhold){
      stop.add(text);
    }
  }
  return stop;
}
```

结果为：

揭开外企的底儿（连载六）——外企招聘也有潜规则.txt

去央企还是外企，帮忙分析下.txt

哪种英语教材比较适合英语基础差的人.txt

有在达内外企软件工程师就业班培训过的吗.txt

两个月的“骑驴找马”，面试无数家公司的深圳体验.txt

```
一个看了可能改变你一生的小说《做单》,外企销售经理做单技巧大揭密.txt
HR 的至高机密：20 个公司绝对不会告诉你的潜规则.txt
```

4、MultiTermQuery

此类查询包含一到多个 Term 的查询，主要包括 FuzzyQuery，PrefixQuery，WildcardQuery，NumericRangeQuery<T>，TermRangeQuery。

本章主要讨论后两者。

4.1、TermRangeQuery

在较早版本的 Lucene，对一定范围内的查询所对应的查询对象是 RangeQuery，然而其仅支持字符串形式的范围查询，因为 Lucene 3.0 提供了数字形式的范围查询 NumericRangeQuery，所以原来的 RangeQuery 变为 TermRangeQuery。

其包含的成员变量如下：

- String lowerTerm; 左边界字符串
- String upperTerm; 右边界字符串
- boolean includeLower; 是否包括左边界
- boolean includeUpper; 是否包含右边界
- String field; 域
- Collator collator; 其允许用户实现其函数 int compare(String source, String target)来决定怎么样算是大于，怎么样算是小于

其提供函数 FilteredTermEnum getEnum(IndexReader reader)用于得到属于此范围的所有 Term：

```
protected FilteredTermEnum getEnum(IndexReader reader) throws IOException {
   return new TermRangeTermEnum(reader, field, lowerTerm, upperTerm, includeLower, includeUpper, collator);
}
```

FilteredTermEnum 不断取下一个 Term 的 next 函数如下：

```
public boolean next() throws IOException {
    if (actualEnum == null) return false;
    currentTerm = null;
    while (currentTerm == null) {
       if (endEnum()) return false;
       if (actualEnum.next()) {
          Term term = actualEnum.term();
          if (termCompare(term)) {
             currentTerm = term;
             return true;
          }
       }
       else return false;
    }
    currentTerm = null;
    return false;
}
```

其中调用 termCompare 来判断此 Term 是否在范围之内，TermRangeTermEnum 的 termCompare 如下：

```
protected boolean termCompare(Term term) {
  if (collator == null) {
    //如果用户没有设定 collator，则使用字符串比较。
    boolean checkLower = false;
    if (!includeLower)
      checkLower = true;
    if (term != null && term.field() == field) {
      if (!checkLower || null==lowerTermText || term.text().compareTo(lowerTermText) > 0) {
        checkLower = false;
```

```
    if (upperTermText != null) {
      int compare = upperTermText.compareTo(term.text());
      if ((compare < 0) ||
          (!includeUpper && compare==0)) {
        endEnum = true;
        return false;
      }
    }
    return true;
  }
 } else {
  endEnum = true;
  return false;
 }
 return false;
} else {
//如果用户设定了 collator，则使用 collator 来比较字符串。
 if (term != null && term.field() == field) {
  if ((lowerTermText == null
      || (includeLower
          ? collator.compare(term.text(), lowerTermText) >= 0
          : collator.compare(term.text(), lowerTermText) > 0))
     && (upperTermText == null
         || (includeUpper
             ? collator.compare(term.text(), upperTermText) <= 0
             : collator.compare(term.text(), upperTermText) < 0))) {
    return true;
  }
```

```
      return false;
    }
    endEnum = true;
    return false;
  }
}
```

由前面分析的 MultiTermQuery 的 rewrite 可以知道，TermRangeQuery 可能生成 BooleanQuery，然而当此范围过大，或者范围内的 Term 过多的时候，可能出现 TooManyClause 异常。

另一种方式可以用 TermRangeFilter，并不变成查询对象，而是对查询结果进行过滤，在 Filter 一节详细介绍。

4.2、NumericRangeQuery

从 Lucene 2.9 开始，提供对数字范围的支持，然而欲使用此查询，必须使用 NumericField 添加域：

```
document.add(new NumericField(name).setIntValue(value));
```

或者使用 NumericTokenStream 添加域：

```
Field field = new Field(name, new NumericTokenStream(precisionStep).setIntValue(value));
field.setOmitNorms(true);
field.setOmitTermFreqAndPositions(true);
document.add(field);
```

NumericRangeQuery 可因不同的类型用如下方法生成：

- newDoubleRange(String, Double, Double, boolean, boolean)
- newFloatRange(String, Float, Float, boolean, boolean)
- newIntRange(String, Integer, Integer, boolean, boolean)
- newLongRange(String, Long, Long, boolean, boolean)

```
public static NumericRangeQuery<Integer> newIntRange(final String field, Integer min, Integer max, final boolean minInclusive, final boolean maxInclusive) {
```

```
  return new NumericRangeQuery<Integer>(field, NumericUtils.PRECISION_STEP_DEFAULT, 32,
min, max, minInclusive, maxInclusive);
}
```

其提供函数 FilteredTermEnum getEnum(IndexReader reader)用于得到属于此范围的所有 Term：

```
protected FilteredTermEnum getEnum(final IndexReader reader) throws IOException {
  return new NumericRangeTermEnum(reader);
}
```

NumericRangeTermEnum 的 termCompare 如下：

```
protected boolean termCompare(Term term) {
  return (term.field() == field && term.text().compareTo(currentUpperBound) <= 0);
}
```

另一种方式可以使用 NumericRangeFilter，下面会详细论述。

举例，我们索引 id 从 0 到 9 的十篇文档到索引中：

```
Document doc = new Document();
doc.add(new Field("contents", new FileReader(file)));
String name = file.getName();
Integer id = Integer.parseInt(name);
doc.add(new NumericField("id").setIntValue(id));
writer.addDocument(doc);
```

搜索的时候，生成 NumericRangeQuery:

```
File indexDir = new File("TestNumericRangeQuery/index");
IndexReader reader = IndexReader.open(FSDirectory.open(indexDir));
IndexSearcher searcher = new IndexSearcher(reader);
NumericRangeQuery<Integer> query = NumericRangeQuery.newIntRange("id", 3, 6, true, false);
TopDocs docs = searcher.search(query, 50);
for (ScoreDoc doc : docs.scoreDocs) {
```

```
  System.out.println("docid : " + doc.doc + " score : " + doc.score);
}
```

结果如下：

```
docid : 3 score : 1.0
docid : 4 score : 1.0
docid : 5 score : 1.0
```

5、SpanQuery

所谓 SpanQuery 也即在查询过程中需要考虑进 Term 的位置信息的查询对象。

SpanQuery 中最基本的是 SpanTermQuery，其只包含一个 Term，与 TermQuery 所不同的是，其提供一个函数来得到位置信息：

```
public Spans getSpans(final IndexReader reader) throws IOException {
  return new TermSpans(reader.termPositions(term), term);
}
```

Spans 有以下方法：

- next() 得到下一篇文档号，不同的 SpanQuery 此方法实现不同
- skipTo(int) 跳到指定的文档
- doc() 得到当前的文档号
- start() 得到起始位置，不同的 SpanQuery 此方法实现不同
- end() 得到结束位置，不同的 SpanQuery 此方法实现不同
- isPayloadAvailable() 是否有 payload
- getPayload() 得到 payload

SpanScorer 的 nextDoc 函数如下：

```
public int nextDoc() throws IOException {
  if (!setFreqCurrentDoc()) {
```

```
    doc = NO_MORE_DOCS;
   }
   return doc;
}
```

```
protected boolean setFreqCurrentDoc() throws IOException {
   if (!more) {
     return false;
   }
   doc = spans.doc();
   freq = 0.0f;
   do {
     //根据结束位置和起始位置来计算 freq 从而影响打分
     int matchLength = spans.end() - spans.start();
     freq += getSimilarity().sloppyFreq(matchLength);
     more = spans.next();
   } while (more && (doc == spans.doc()));
   return true;
}
```

5.1、SpanFirstQuery

SpanFirstQuery 仅取在开头部分包含查询词的文档，其包含如下成员变量：

- SpanQuery match；需要满足的查询
- int end；如何定义开头

其 getSpans 函数如下：

```
public Spans getSpans(final IndexReader reader) throws IOException {
   return new Spans() {
```

```
    private Spans spans = match.getSpans(reader);
    @Override
    public boolean next() throws IOException {
     while (spans.next()) {
      //仅查询词的位置在设定的 end 之前的文档才返回。
      if (end() <= end)
       return true;
     }
     return false;
    }
    @Override
    public boolean skipTo(int target) throws IOException {
     if (!spans.skipTo(target))
      return false;
     return spans.end() <= end || next();
    }
    @Override
    public int doc() { return spans.doc(); }
    @Override
    public int start() { return spans.start(); }
    @Override
    public int end() { return spans.end(); }
   };
}
```

5.2、SpanNearQuery

SpanNearQuery 包含以下成员变量：

- List<SpanQuery> clauses; 一个列表的子 SpanQuery

- int slop；设定这些字 SpanQuery 之间的距离的最大值，大于此值则文档不返回。
- boolean inOrder；是否按顺序计算子 SpanQuery 之间的距离
- String field；域
- boolean collectPayloads；是否收集 payload

其 getSpans 函数如下：

```
public Spans getSpans(final IndexReader reader) throws IOException {
  if (clauses.size() == 0)
    return new SpanOrQuery(getClauses()).getSpans(reader);
  if (clauses.size() == 1)
    return clauses.get(0).getSpans(reader);
  return inOrder
        ? (Spans) new NearSpansOrdered(this, reader, collectPayloads)
        : (Spans) new NearSpansUnordered(this, reader);
}
```

是否 inorder，举例如下：

假设索引了文档 "apple boy cat"，如果将 SpanNearQuery 的 clauses 依次设为 "apple","cat","boy"，如果 inorder=true，则文档不会被搜索出来，即便 slop 设为很大，如果 inorder=false，则文档会被搜出来，而且 slop 设为 0 就能被搜出来。

因为在 NearSpansOrdered 的 next 函数如下：

```
public boolean next() throws IOException {
  if (firstTime) {
    firstTime = false;
    for (int i = 0; i < subSpans.length; i++) {
     //每个子 SpanQuery 都取第一篇文档
      if (! subSpans[i].next()) {
        more = false;
        return false;
      }
```

```
    }
    more = true;
  }
  if(collectPayloads) {
    matchPayload.clear();
  }
  return advanceAfterOrdered();
}
```

```
private boolean advanceAfterOrdered() throws IOException {
  //如果各子 SpanQuery 指向同一文档
  while (more && (inSameDoc || toSameDoc())) {
    //stretchToOrder 要保证各子 SpanQuery 一定是按照顺序排列的
    //shrinkToAfterShortestMatch 保证各子 SpanQuery 之间的距离不大于 slop
    if (stretchToOrder() && shrinkToAfterShortestMatch()) {
      return true;
    }
  }
  return false;
}
```

```
private boolean stretchToOrder() throws IOException {
  matchDoc = subSpans[0].doc();
  for (int i = 1; inSameDoc && (i < subSpans.length); i++) {
    //docSpansOrdered 要保证第 i-1 个子 SpanQuery 的 start 和 end 都应在第 i 个之前，否则取下一篇文档。
    while (! docSpansOrdered(subSpans[i-1], subSpans[i])) {
      if (! subSpans[i].next()) {
        inSameDoc = false;
        more = false;
```

```
      break;
     } else if (matchDoc != subSpans[i].doc()) {
      inSameDoc = false;
      break;
     }
    }
  }
  return inSameDoc;
}
```

```
static final boolean docSpansOrdered(Spans spans1, Spans spans2) {
  assert spans1.doc() == spans2.doc() : "doc1 " + spans1.doc() + " != doc2 " + spans2.doc();
  int start1 = spans1.start();
  int start2 = spans2.start();
  return (start1 == start2) ? (spans1.end() < spans2.end()) : (start1 < start2);
}
```

```
private boolean shrinkToAfterShortestMatch() throws IOException {
  //从最后一个子 SpanQuery 开始
  matchStart = subSpans[subSpans.length - 1].start();
  matchEnd = subSpans[subSpans.length - 1].end();
  int matchSlop = 0;
  int lastStart = matchStart;
  int lastEnd = matchEnd;
  for (int i = subSpans.length - 2; i >= 0; i—) {
   //不断的取前一个子 SpanQuery
   Spans prevSpans = subSpans[i];
   int prevStart = prevSpans.start();
   int prevEnd = prevSpans.end();
   while (true) {
```

```
    if (! prevSpans.next()) {
      inSameDoc = false;
      more = false;
      break;
    } else if (matchDoc != prevSpans.doc()) {
      inSameDoc = false;
      break;
    } else {
      int ppStart = prevSpans.start();
      int ppEnd = prevSpans.end();
      if (! docSpansOrdered(ppStart, ppEnd, lastStart, lastEnd)) {
        break;
      } else {
        prevStart = ppStart;
        prevEnd = ppEnd;
      }
     }
   }
   assert prevStart <= matchStart;
   if (matchStart > prevEnd) {
    //总是从下一个的开始位置，减去前一个的结束位置，所以上面的例子中，如果将SpanNearQuery 的 clauses 依次设为"apple","boy","cat"，inorder=true, slop=0，是能够搜索的出的。
     matchSlop += (matchStart - prevEnd);
   }
   matchStart = prevStart;
   lastStart = prevStart;
   lastEnd = prevEnd;
```

```
  }
  boolean match = matchSlop <= allowedSlop;
  return match;
}
```

NearSpansUnordered 的 next 函数如下：

```
public boolean next() throws IOException {
  if (firstTime) {
   //将一个 Spans 生成一个 SpansCell，既放入链表中，也放入优先级队列中，在队列中按照第一篇文档号由小到大排列，若文档号相同，则按照位置顺序排列。
    initList(true);
    listToQueue();
    firstTime = false;
  } else if (more) {
    if (min().next()) { //最上面的取下一篇文档，并调整队列。
      queue.updateTop();
    } else {
      more = false;
    }
  }
  while (more) {
    boolean queueStale = false;
    if (min().doc() != max.doc()) { //如果队列中最小的文档号和最大的文档号不相同，将队列生成链表。
      queueToList();
      queueStale = true;
    }
   //应该不断的 skip 每个子 SpanQuery 直到最小的文档号和最大的文档号相同，不同的是在文档中的位置。
```

```
    while (more && first.doc() < last.doc()) {
      more = first.skipTo(last.doc());
      firstToLast();
      queueStale = true;
    }
    if (!more) return false;
    //调整完毕后，将链表写回队列。
    if (queueStale) {
      listToQueue();
      queueStale = false;
    }
    //判断是否匹配
    if (atMatch()) {
      return true;
    }
    more = min().next();
    if (more) {
      queue.updateTop();
    }
  }
  return false;
}
```

```
private boolean atMatch() {

  //匹配有两个条件，一个是最小和最大的文档号相同，一个是最大的结束位置减去最小的开始位置再减去最大和最小的自身的长度之和小于等于 slop。

  //在上面的例子中，如果将 SpanNearQuery 的 clauses 依次设为"cat","apple"，inorder=false，则 slop 设为 1 可以搜索的出来。因为"cat".end = 3,
```

```
"apple".start=0, totalLength = ("cat".end – "cat".start) + ("apple".end – "apple.start") = 2，所以 slop=1 即可。

  return (min().doc() == max.doc())

     && ((max.end() - min().start() - totalLength) <= slop);

}
```

5.3、SpanNotQuery

SpanNotQuery 包含如下两个成员变量：

- SpanQuery include; 必须满足的 SpanQuery
- SpanQuery exclude; 必须不能满足的 SpanQuery

其 next 函数从 include 中取出文档号，如果 exclude 也包括此文档号，则过滤掉。

其 getSpans 函数如下：

```
public Spans getSpans(final IndexReader reader) throws IOException {

  return new Spans() {

     private Spans includeSpans = include.getSpans(reader);

     private boolean moreInclude = true;

     private Spans excludeSpans = exclude.getSpans(reader);

     private boolean moreExclude = excludeSpans.next();

     @Override
```

```
    public boolean next() throws IOException {
      //得到下一个 include 的文档号
      if (moreInclude)
        moreInclude = includeSpans.next();
      //此循环查看此文档号是否被 exclude，如果是则取下一个 include 的文档号。
      while (moreInclude && moreExclude) {
        //将 exclude 跳到 include 文档号
        if (includeSpans.doc() > excludeSpans.doc())
          moreExclude = excludeSpans.skipTo(includeSpans.doc());
        //当 include 和 exclude 文档号相同的时候，不断取得下一个 exclude，如果 exclude 的 end
的 start，则说明当前文档号应该被 exclude。
        while (moreExclude
            && includeSpans.doc() == excludeSpans.doc()
            && excludeSpans.end() <= includeSpans.start()) {
          moreExclude = excludeSpans.next();
        }
        //如果是因为没有 exclude 了，或者文档号不相同，或者 include 的 end 小于 exclude 的 star
不应该被 exclude。
```

```
        if (!moreExclude
            || includeSpans.doc() != excludeSpans.doc()
            || includeSpans.end() <= excludeSpans.start())
          break;
        //否则此文档应该被 exclude，include 取下一篇文档号。
        moreInclude = includeSpans.next();
      }
      return moreInclude;
    }
    @Override
    public int doc() { return includeSpans.doc(); }
    @Override
    public int start() { return includeSpans.start(); }
    @Override
    public int end() { return includeSpans.end(); }
  };
}
```

5.4、SpanOrQuery

SpanOrQuery 包含一个列表的子 SpanQuery，并对它们取 OR 的关系，用于满足"apple 和 boy 临近或者 cat 和 dog 临近的文档"此类的查询。

其 OR 的合并算法同 BooleanQuery 的 OR 关系的算法 DisjunctionSumScorer 类似。

```
public boolean next() throws IOException {
  if (queue == null) {
    return initSpanQueue(-1);
  }
  if (queue.size() == 0) {
    return false;
  }
  //在优先级队列顶部取下一篇文档或者下一位置，并重新排列队列
  if (top().next()) {
    queue.updateTop();
    return true;
  }
  //如果最顶部的 SpanQuery 没有下一篇文档或者下一位置，则弹出
  queue.pop();
  return queue.size() != 0;
}
```

5.5、FieldMaskingSpanQuery

在 SpanNearQuery 中，需要进行位置比较，相互比较位置的 Term 必须要在同一个域中，否则报异常 IllegalArgumentException("Clauses must have same field.").

然而有时候我们需要对不同的域中的位置进行比较，例如：

文档一：

```
teacherid: 1
studentfirstname: james
studentsurname: jones
```

我们建索引如下：

```
Document doc = new Document();
doc.add(new Field("teacherid", "1", Field.Store.YES, Field.Index.NOT_ANALYZED));
doc.add(new Field("studentfirstname", "james", Field.Store.YES, Field.Index.NOT_ANALYZED));
doc.add(new Field("studentsurname", "jones", Field.Store.YES, Field.Index.NOT_ANALYZED));
writer.addDocument(doc);
```

文档二：

```
teacherid: 2
studenfirstname: james
studentsurname: smith
studentfirstname: sally
studentsurname: jones
```

我们建索引如下：

```
doc = new Document();
doc.add(new Field("teacherid", "2", Field.Store.YES, Field.Index.NOT_ANALYZED));
doc.add(new Field("studentfirstname", "james", Field.Store.YES, Field.Index.NOT_ANALYZED));
doc.add(new Field("studentsurname", "smith", Field.Store.YES, Field.Index.NOT_ANALYZED));
doc.add(new Field("studentfirstname", "sally", Field.Store.YES, Field.Index.NOT_ANALYZED));
doc.add(new Field("studentsurname", "jones", Field.Store.YES, Field.Index.NOT_ANALYZED));
writer.addDocument(doc);
```

现在我们想找 firstname 是 james，surname 是 jones 的学生的老师，显然如果搜索 "studenfirstname: james AND studentsurname: jones"，显然上面两个老师都能够搜索出来，可以辨别 james 和 jones 属于同一学生的一种方法是位置信息，也即当 james 和 jones 处于两个域的同一位置的时候，其属于同一个学生。

这时我们如果声明两个 SpanTermQuery：

```
SpanQuery q1  = new SpanTermQuery(new Term("studentfirstname", "james"));
SpanQuery q2  = new SpanTermQuery(new Term("studentsurname", "jones"));
```

然后构建 SpanNearQuery，子 SpanQuery 为上述 q1, q2，因为在同一位置 inorder=false，slop 设为-1，因为

"jones".end – "james".start – totallength = 1 – 0 – 2 = -1，这样就能够搜的出来。

然而在构建 SpanNearQuery 的时候，其构造函数如下：

```
public SpanNearQuery(SpanQuery[] clauses, int slop, boolean inOrder, boolean collectPayloads) {
  this.clauses = new ArrayList<SpanQuery>(clauses.length);
  for (int i = 0; i < clauses.length; i++) {
    SpanQuery clause = clauses[i];
    if (i == 0) {
      field = clause.getField();
    } else if (!clause.getField().equals(field)) { //要求所有的子 SpanQuery 都属于同一个域
      throw new IllegalArgumentException("Clauses must have same field.");
    }
    this.clauses.add(clause);
  }
  this.collectPayloads = collectPayloads;
  this.slop = slop;
  this.inOrder = inOrder;
}
```

所以我们引入 FieldMaskingSpanQuery，SpanQuery q2m = new FieldMaskingSpanQuery(q2, "studentfirstname");

FieldMaskingSpanQuery.getField()得到的是你指定的假的域信息"studentfirstname"，从而通过了审核，就可以计算位置信息了。

我们的查询过程如下：

```
File indexDir = new File("TestFieldMaskingSpanQuery/index");
IndexReader reader = IndexReader.open(FSDirectory.open(indexDir));
```

```
IndexSearcher searcher = new IndexSearcher(reader);
SpanQuery q1  = new SpanTermQuery(new Term("studentfirstname", "james"));
SpanQuery q2  = new SpanTermQuery(new Term("studentsurname", "jones"));
SpanQuery q2m = new FieldMaskingSpanQuery(q2, "studentfirstname");
Query query = new SpanNearQuery(new SpanQuery[]{q1, q2m}, -1, false);
TopDocs docs = searcher.search(query, 50);
for (ScoreDoc doc : docs.scoreDocs) {
  System.out.println("docid : " + doc.doc + " score : " + doc.score);
}
```

5.6、PayloadTermQuery 及 PayloadNearQuery

带 Payload 前缀的查询对象不会因为 payload 的存在而使得结果集发生改变，而仅仅改变其评分。

欲使用 Payload 系列的查询语句:

- 首先在索引阶段，要将 payload 存入到索引中去：PayloadAttribute..setPayload(new Payload(byte[] b));
- 其次是实现自己的 Similarity，并实现其接口 float scorePayload(int docId, String fieldName, int start, int end, byte [] payload, int offset, int length)，可以指定如何根据读出的二进制 payload 计算 payload 的打分。
- 最后在构建 PayloadTermQuery 及 PayloadNearQuery 的时候传入 PayloadFunction function

PayloadFunction 需要实现两个接口：

- float currentScore(int docId, String field, int start, int end, int numPayloadsSeen, float currentScore, float currentPayloadScore)是在上一步用 Similarity 根据二进制 payload 计算出 payload 打分后，此打分作为 currentPayloadScore 传入，此次计算前的原分数作为 currentScore 传入，此处可以指定 payload 如何影响原来的打分。
- float docScore(int docId, String field, int numPayloadsSeen, float payloadScore)当所有的 payload 都被计算完毕后，如何调整最终的打分。

PayloadFunction 有三种实现：

- AveragePayloadFunction，其在 currentScore 函数中，总是将 payload 的打分加到原分数中，currentPayloadScore + currentScore，然后在所有的 payload 都计算完毕后，在 docScore 函数中，对这些打分取平均值，return numPayloadsSeen > 0 ? (payloadScore / numPayloadsSeen) : 1
- MaxPayloadFunction，其在 currentScore 函数中，总是取两者的最大值 Math.max(currentPayloadScore, currentScore)，最后在 docScore 函数中将最大值返回，return numPayloadsSeen > 0 ? payloadScore : 1
- MinPayloadFunction，其在 currentScore 函数中，总是取两者的最小值 Math.min(currentPayloadScore, currentScore)，最后在 docScore 函数中将最小值返回，return numPayloadsSeen > 0 ? payloadScore : 1

对于 PayloadTermQuery 来讲，在其生成的 PayloadTermSpanScorer 中：

- 首先计算出 payloadScore

```
payloadScore = function.currentScore(doc, term.field(), spans.start(), spans.end(), payloadsSeen, payloadScore, similarity.scorePayload(doc, term.field(), spans.start(), spans.end(), payload, 0, positions.getPayloadLength()));
```

- 然后在 score 函数中调用 getSpanScore() * getPayloadScore()

```
protected float getPayloadScore() {
  return function.docScore(doc, term.field(), payloadsSeen, payloadScore);
}
```

对于 PayloadNearQuery 来讲，在其生成的 PayloadNearSpanScorer 中：

- 首先计算出 payloadScore

```
payloadScore = function.currentScore(doc, fieldName, start, end, payloadsSeen, payloadScore, similarity.scorePayload(doc, fieldName, spans.start(), spans.end(), thePayload, 0, thePayload.length) );
```

- 然后在 score 函数中

```
public float score() throws IOException {
```

```
  return super.score() * function.docScore(doc, fieldName, payloadsSeen, payloadScore);
}
```

6、FilteredQuery

FilteredQuery 包含两个成员变量：

- Query query：查询对象
- Filter filter：其有一个函数 DocIdSet getDocIdSet(IndexReader reader) 得到一个文档号集合，结果文档必须出自此文档集合，注此处的过滤器所包含的文档号并不是要过滤掉的文档号，而是过滤后需要的文档号。

FilterQuery 所得到的结果集同两者取 AND 查询相同，只不过打分的时候，FilterQuery 只考虑 query 的部分，不考虑 filter 的部分。

Filter 包含很多种如下：

6.1、TermsFilter

其包含一个成员变量 Set<Term> terms=new TreeSet<Term>()，所有包含 terms 集合中任一 term 的文档全部属于文档号集合。

其 getDocIdSet 函数如下：

```
public DocIdSet getDocIdSet(IndexReader reader) throws IOException
{
    //生成一个 bitset，大小为索引中文档总数
    OpenBitSet result=new OpenBitSet(reader.maxDoc());
    TermDocs td = reader.termDocs();
    try
    {
      //遍历每个 term 的文档列表，将文档号都在 bitset 中置一，从而 bitset 包含了
```

```
所有的文档号。
        for (Iterator<Term> iter = terms.iterator(); iter.hasNext();)
        {
           Term term = iter.next();
           td.seek(term);
           while (td.next())
           {
              result.set(td.doc());
           }
        }
     }
     finally
     {
        td.close();
     }
     return result;
  }
```

6.2、BooleanFilter

其像 BooleanQuery 相似，包含 should 的 filter，must 的 filter，not 的 filter，在 getDocIdSet 的时候，先将所有满足 should 的文档号集合之间取 OR 的关系，然后同 not 的文档号集合取 NOT 的关系，最后同 must 的文档号集合取 AND 的关系，得到最后的文档集合。

其 getDocIdSet 函数如下：

```
public DocIdSet getDocIdSet(IndexReader reader) throws IOException
{
  OpenBitSetDISI res = null;
```

```
if (shouldFilters != null) {
  for (int i = 0; i < shouldFilters.size(); i++) {
    if (res == null) {
      res = new OpenBitSetDISI(getDISI(shouldFilters, i, reader), reader.maxDoc());
    } else {
      //将 should 的 filter 的文档号全部取 OR 至 bitset 中
      DocIdSet dis = shouldFilters.get(i).getDocIdSet(reader);
      if(dis instanceof OpenBitSet) {
        res.or((OpenBitSet) dis);
      } else {
        res.inPlaceOr(getDISI(shouldFilters, i, reader));
      }
    }
  }
}
if (notFilters!=null) {
  for (int i = 0; i < notFilters.size(); i++) {
    if (res == null) {
      res = new OpenBitSetDISI(getDISI(notFilters, i, reader), reader.maxDoc());
      res.flip(0, reader.maxDoc());
    } else {
      //将 not 的 filter 的文档号全部取 NOT 至 bitset 中
      DocIdSet dis = notFilters.get(i).getDocIdSet(reader);
      if(dis instanceof OpenBitSet) {
        res.andNot((OpenBitSet) dis);
      } else {
        res.inPlaceNot(getDISI(notFilters, i, reader));
      }
```

```
    }
   }
  }
  if (mustFilters!=null) {
   for (int i = 0; i < mustFilters.size(); i++) {
    if (res == null) {
     res = new OpenBitSetDISI(getDISI(mustFilters, i, reader), reader.maxDoc());
    } else {
     //将 must 的 filter 的文档号全部取 AND 至 bitset 中
     DocIdSet dis = mustFilters.get(i).getDocIdSet(reader);
     if(dis instanceof OpenBitSet) {
      res.and((OpenBitSet) dis);
     } else {
      res.inPlaceAnd(getDISI(mustFilters, i, reader));
     }
    }
   }
  }
  if (res !=null)
   return finalResult(res, reader.maxDoc());
  return DocIdSet.EMPTY_DOCIDSET;
}
```

6.3、DuplicateFilter

DuplicateFilter 实现了如下的功能：

比如说我们有这样一批文档，每篇文档都分成多页，每篇文档都有一个 id，然而每一页是按照单独的 Document 进行索引的，于是进行搜索的时候，当一篇文档的两页都包含关键词的时候，此文档 id 在结果集中出现两次，这是我们不想看到的，DuplicateFilter 就是指定一个

域如 id，在此域相同的文档仅取其中一篇。

DuplicateFilter 包含以下成员变量：

- String fieldName：域的名称
- int keepMode：KM_USE_FIRST_OCCURRENCE 表示重复的文档取第一篇，KM_USE_LAST_OCCURRENCE 表示重复的文档取最后一篇。
- int processingMode：
  - PM_FULL_VALIDATION 是首先将 bitset 中所有文档都设为 false，当出现同组重复文章的第一篇的时候，将其设为 1
  - PM_FAST_INVALIDATION 是首先将 bitset 中所有文档都设为 true，除了同组重复文章的第一篇，其他的的全部设为 0
  - 两者在所有的文档都包含指定域的情况下，功能一样，只不过后者不用处理 docFreq=1 的文档，速度加快。
  - 然而当有的文档不包含指定域的时候，后者由于都设为 true，则没有机会将其清零，因而会被允许返回，当然工程中应避免这种情况。

其 getDocIdSet 函数如下：

```
public DocIdSet getDocIdSet(IndexReader reader) throws IOException
 {
   if(processingMode==PM_FAST_INVALIDATION)
   {
      return fastBits(reader);
   }
   else
   {
      return correctBits(reader);
   }
 }
```

```
private OpenBitSet correctBits(IndexReader reader) throws IOException
 {
   OpenBitSet bits=new OpenBitSet(reader.maxDoc());
   Term startTerm=new Term(fieldName);
   TermEnum te = reader.terms(startTerm);
```

```
if(te!=null)
{
   Term currTerm=te.term();
  //如果属于指定的域
   while((currTerm!=null)&&(currTerm.field()==startTerm.field()))
   {
      int lastDoc=-1;
      //则取出包含此 term 的所有的文档
      TermDocs td = reader.termDocs(currTerm);
      if(td.next())
      {
         if(keepMode==KM_USE_FIRST_OCCURRENCE)
         {
            //第一篇设为 true
            bits.set(td.doc());
         }
         else
         {
            do
            {
               lastDoc=td.doc();
            }while(td.next());
            bits.set(lastDoc); //最后一篇设为 true
         }
      }
      if(!te.next())
      {
         break;
```

```
            }
            currTerm=te.term();
        }
    }
    return bits;
}
```

```
private OpenBitSet fastBits(IndexReader reader) throws IOException
{
    OpenBitSet bits=new OpenBitSet(reader.maxDoc());
    bits.set(0,reader.maxDoc());  //全部设为 true
    Term startTerm=new Term(fieldName);
    TermEnum te = reader.terms(startTerm);
    if(te!=null)
    {
        Term currTerm=te.term();
        //如果属于指定的域
        while((currTerm!=null)&&(currTerm.field()==startTerm.field()))
        {
            if(te.docFreq()>1)
            {
                int lastDoc=-1;
                //取出所有的文档
                TermDocs td = reader.termDocs(currTerm);
                td.next();
                if(keepMode==KM_USE_FIRST_OCCURRENCE)
                {
                    //除了第一篇不清零
                    td.next();
```

```
            }
            do
            {
               lastDoc=td.doc();
               bits.clear(lastDoc); //其他全部清零
            }while(td.next());
            if(keepMode==KM_USE_LAST_OCCURRENCE)
            {
               bits.set(lastDoc); //最后一篇设为 true
            }
          }
          if(!te.next())
          {
            break;
          }
          currTerm=te.term();
        }
      }
      return bits;
    }
```

举例，我们索引如下的文件：

```
File indexDir = new File("TestDuplicateFilter/index");
IndexWriter writer = new IndexWriter(FSDirectory.open(indexDir), new StandardAnalyzer(Version.LUCENE_CURRENT), true, IndexWriter.MaxFieldLength.LIMITED);
Document doc = new Document();
doc.add(new Field("id", "1", Field.Store.YES, Field.Index.NOT_ANALYZED));
doc.add(new Field("contents", "page 1: hello world", Field.Store.YES, Field.Index.ANALYZED));
writer.addDocument(doc);
```

```
doc = new Document();
doc.add(new Field("id", "1", Field.Store.YES, Field.Index.NOT_ANALYZED));
doc.add(new Field("contents", "page 2: hello world", Field.Store.YES, Field.Index.ANALYZED));
writer.addDocument(doc);
doc = new Document();
doc.add(new Field("id", "1", Field.Store.YES, Field.Index.NOT_ANALYZED));
doc.add(new Field("contents", "page 3: hello world", Field.Store.YES, Field.Index.ANALYZED));
writer.addDocument(doc);
doc = new Document();
doc.add(new Field("id", "2", Field.Store.YES, Field.Index.NOT_ANALYZED));
doc.add(new Field("contents", "page 1: hello world", Field.Store.YES, Field.Index.ANALYZED));
writer.addDocument(doc);
doc = new Document();
doc.add(new Field("id", "2", Field.Store.YES, Field.Index.NOT_ANALYZED));
doc.add(new Field("contents", "page 2: hello world", Field.Store.YES, Field.Index.ANALYZED));
writer.addDocument(doc);
writer.close();
```

如果搜索 TermQuery tq = new TermQuery(new Term("contents","hello"))，则结果为：

```
id : 1
id : 1
id : 1
id : 2
id : 2
```

如果按如下进行搜索：

```
File indexDir = new File("TestDuplicateFilter/index");
IndexReader reader = IndexReader.open(FSDirectory.open(indexDir));
IndexSearcher searcher = new IndexSearcher(reader);
TermQuery tq = new TermQuery(new Term("contents","hello"));
```

```
DuplicateFilter filter = new DuplicateFilter("id");
FilteredQuery query = new FilteredQuery(tq, filter);
TopDocs docs = searcher.search(query, 50);
for (ScoreDoc doc : docs.scoreDocs) {
  Document ldoc = reader.document(doc.doc);
  String id = ldoc.get("id");
  System.out.println("id : " + id);
}
```

则结果为：

```
id : 1
id : 2
```

6.4、FieldCacheRangeFilter<T>及 FieldCacheTermsFilter

在介绍与 FieldCache 相关的 Filter 之前，先介绍 FieldCache。

FieldCache 缓存的是不是存储域的内容，而是索引域中 term 的内容，索引中的 term 是 String 的类型，然而可以将其他的类型作为 String 类型索引进去，例如"1"，"2.3"等，然后搜索的时候将这些信息取出来。

FieldCache 支持如下类型：

- byte[] getBytes (IndexReader reader, String field, ByteParser parser)
- double[] getDoubles(IndexReader reader, String field, DoubleParser parser)
- float[] getFloats (IndexReader reader, String field, FloatParser parser)
- int[] getInts (IndexReader reader, String field, IntParser parser)
- long[] getLongs(IndexReader reader, String field, LongParser parser)
- short[] getShorts (IndexReader reader, String field, ShortParser parser)
- String[] getStrings (IndexReader reader, String field)
- StringIndex getStringIndex (IndexReader reader, String field)

其中 StringIndex 包含两个成员：

- String[] lookup：按照字典顺序排列的所有 term。
- int[] order：其中位置表示文档号，order[i]第 i 篇文档包含的 term 在 lookup 中的位置。

FieldCache 默认的实现 FieldCacheImpl，其中包含成员变量 Map<Class<?>,Cache> caches 保存从类型到 Cache 的映射。

```
private synchronized void init() {
  caches = new HashMap<Class<?>,Cache>(7);
  caches.put(Byte.TYPE, new ByteCache(this));
  caches.put(Short.TYPE, new ShortCache(this));
  caches.put(Integer.TYPE, new IntCache(this));
  caches.put(Float.TYPE, new FloatCache(this));
  caches.put(Long.TYPE, new LongCache(this));
  caches.put(Double.TYPE, new DoubleCache(this));
  caches.put(String.class, new StringCache(this));
  caches.put(StringIndex.class, new StringIndexCache(this));
}
```

其实现接口 getInts 如下，即先得到 Integer 类型所对应的 IntCache 然后，再从其中根据 reader 和由 field 和 parser 组成的 Entry 得到整型值。

```
public int[] getInts(IndexReader reader, String field, IntParser parser) throws IOException {
  return (int[]) caches.get(Integer.TYPE).get(reader, new Entry(field, parser));
}
```

各类缓存的父类 Cache 包含成员变量 Map<Object, Map<Entry, Object>> readerCache，其中 key 是 IndexReader，value 是一个 Map，此 Map 的 key 是 Entry，也即是 field，value 是缓存的 int[]的值。(也即在这个 reader 的这个 field 中有一个数组的 int，每一项代表一篇文档)。Cache 的 get 函数如下：

```
public Object get(IndexReader reader, Entry key) throws IOException {
  Map<Entry,Object> innerCache;
  Object value;
   final Object readerKey = reader.getFieldCacheKey(); // 此函数返回 this，也即
```

IndexReader 本身

```
 synchronized (readerCache) {
  innerCache = readerCache.get(readerKey); //通过 IndexReader 得到 Map
  if (innerCache == null) { //如果没有则新建一个 Map
   innerCache = new HashMap<Entry,Object>();
   readerCache.put(readerKey, innerCache);
   value = null;
  } else {
   value = innerCache.get(key); //此 Map 的 key 是 Entry，value 即是缓存的值
  }
  //如果缓存不命中，则创建此值
  if (value == null) {
   value = new CreationPlaceholder();
   innerCache.put(key, value);
  }
 }
 if (value instanceof CreationPlaceholder) {
  synchronized (value) {
   CreationPlaceholder progress = (CreationPlaceholder) value;
   if (progress.value == null) {
    progress.value = createValue(reader, key); //调用此函数创建缓存值
    synchronized (readerCache) {
     innerCache.put(key, progress.value);
     }
    }
   }
   return progress.value;
 }
```

```
  return value;
}
```

Cache 的 createValue 函数根据类型的不同而不同，我们仅分析 IntCache 和 StringIndexCache 的实现.

IntCache 的 createValue 函数如下：

```
  protected Object createValue(IndexReader reader, Entry entryKey) throws IOException {
    Entry entry = entryKey;
    String field = entry.field;
    IntParser parser = (IntParser) entry.custom;
    int[] retArray = null;
    TermDocs termDocs = reader.termDocs();
    TermEnum termEnum = reader.terms (new Term (field));
    try {
      //依次将域中所有的 term 都取出来，用 IntParser 进行解析，缓存 retArray[]位置即文档号，retArray[i]即第 i 篇文档所包含的 int 值.
      do {
        Term term = termEnum.term();
        if (term==null || term.field() != field) break;
        int termval = parser.parseInt(term.text());
        if (retArray == null)
          retArray = new int[reader.maxDoc()];
        termDocs.seek (termEnum);
        while (termDocs.next()) {
          retArray[termDocs.doc()] = termval;
        }
      } while (termEnum.next());
    } catch (StopFillCacheException stop) {
    } finally {
```

```
    termDocs.close();
    termEnum.close();
   }
   if (retArray == null)
    retArray = new int[reader.maxDoc()];
   return retArray;
  }
};
```

StringIndexCache 的 createValue 函数如下：

```
protected Object createValue(IndexReader reader, Entry entryKey) throws IOException {
  String field = StringHelper.intern(entryKey.field);
  final int[] retArray = new int[reader.maxDoc()];
  String[] mterms = new String[reader.maxDoc()+1];
  TermDocs termDocs = reader.termDocs();
  TermEnum termEnum = reader.terms (new Term (field));
  int t = 0;
  mterms[t++] = null;
  try {
   do {
    Term term = termEnum.term();
    if (term==null || term.field() != field) break;
    mterms[t] = term.text(); //mterms[i]保存的是按照字典顺序第 i 个 term 所对应的字符串。
    termDocs.seek (termEnum);
    while (termDocs.next()) {
     retArray[termDocs.doc()] = t; //retArray[i]保存的是第 i 篇文档所包含的字符串在 mterms 中的位置。
    }
```

```
      t++;
    } while (termEnum.next());
  } finally {
    termDocs.close();
    termEnum.close();
  }
  if (t == 0) {
    mterms = new String[1];
  } else if (t < mterms.length) {
    String[] terms = new String[t];
    System.arraycopy (mterms, 0, terms, 0, t);
    mterms = terms;
  }
  StringIndex value = new StringIndex (retArray, mterms);
  return value;
}
```

FieldCacheRangeFilter 的可以是各种类型的 Range，其中 Int 类型用下面的函数生成：

```
public static FieldCacheRangeFilter<Integer> newIntRange(String field, FieldCache.IntParser parser,
Integer lowerVal, Integer upperVal, boolean includeLower, boolean includeUpper) {
  return new FieldCacheRangeFilter<Integer>(field, parser, lowerVal, upperVal, includeLower,
includeUpper) {
    @Override
    public DocIdSet getDocIdSet(IndexReader reader) throws IOException {
      final int inclusiveLowerPoint, inclusiveUpperPoint;
      //计算左边界
      if (lowerVal != null) {
        int i = lowerVal.intValue();
        if (!includeLower && i == Integer.MAX_VALUE)
```

```
        return DocIdSet.EMPTY_DOCIDSET;
      inclusiveLowerPoint = includeLower ? i : (i + 1);
    } else {
      inclusiveLowerPoint = Integer.MIN_VALUE;
    }
    //计算右边界
    if (upperVal != null) {
      int i = upperVal.intValue();
      if (!includeUpper && i == Integer.MIN_VALUE)
        return DocIdSet.EMPTY_DOCIDSET;
      inclusiveUpperPoint = includeUpper ? i : (i - 1);
    } else {
      inclusiveUpperPoint = Integer.MAX_VALUE;
    }
    if (inclusiveLowerPoint > inclusiveUpperPoint)
      return DocIdSet.EMPTY_DOCIDSET;
    //从 cache 中取出 values，values[i]表示第 i 篇文档在此域中的值
    final int[] values = FieldCache.DEFAULT.getInts(reader, field, (FieldCache.IntParser) parser);
    return new FieldCacheDocIdSet(reader, (inclusiveLowerPoint <= 0 && inclusiveUpperPoint >=
0)) {
      @Override
      boolean matchDoc(int doc) {
        //仅在文档 i 所对应的值在区间内的时候才返回。
        return values[doc] >= inclusiveLowerPoint && values[doc] <= inclusiveUpperPoint;
      }
    };
  }
};
```

```
}
```

FieldCacheRangeFilter 同 NumericRangeFilter 或者 TermRangeFilter 功能类似，只不过后两者取得 docid 的 bitset 都是从索引中取出，而前者是缓存了的，加快了速度。

同样 FieldCacheTermsFilter 同 TermFilter 功能类似，也是前者进行了缓存，加快了速度。

6.5、MultiTermQueryWrapperFilter<Q>

MultiTermQueryWrapperFilter 包含成员变量 Q query，其 getDocIdSet 得到满足此 query 的文档号 bitset。

```
public DocIdSet getDocIdSet(IndexReader reader) throws IOException {
  final TermEnum enumerator = query.getEnum(reader);
  try {
    if (enumerator.term() == null)
      return DocIdSet.EMPTY_DOCIDSET;
    final OpenBitSet bitSet = new OpenBitSet(reader.maxDoc());
    final int[] docs = new int[32];
    final int[] freqs = new int[32];
    TermDocs termDocs = reader.termDocs();
    try {
      int termCount = 0;
      //遍历满足 query 的所有 term
      do {
        Term term = enumerator.term();
        if (term == null)
          break;
        termCount++;
        termDocs.seek(term);
        while (true) {
```

```
      //得到每个 term 的文档号列表，放入 bitset
      final int count = termDocs.read(docs, freqs);
      if (count != 0) {
        for(int i=0;i<count;i++) {
          bitSet.set(docs[i]);
        }
      } else {
        break;
      }
     }
    } while (enumerator.next());
    query.incTotalNumberOfTerms(termCount);
   } finally {
    termDocs.close();
   }
   return bitSet;
  } finally {
   enumerator.close();
  }
}
```

MultiTermQueryWrapperFilter 有三个重要的子类：

- NumericRangeFilter<T>：以 NumericRangeQuery 作为 query
- PrefixFilter：以 PrefixQuery 作为 query
- TermRangeFilter：以 TermRangeQuery 作为 query

6.6、QueryWrapperFilter

其包含一个查询对象，getDocIdSet 会获得所有满足此查询的文档号：

```
public DocIdSet getDocIdSet(final IndexReader reader) throws IOException {
  final Weight weight = query.weight(new IndexSearcher(reader));
  return new DocIdSet() {
    public DocIdSetIterator iterator() throws IOException {
      return weight.scorer(reader, true, false); //Scorer 的 next 即返回一个个文档号。
    }
  };
}
```

6.7、SpanFilter

6.7.1、SpanQueryFilter

其包含一个 SpanQuery query，作为过滤器，其除了通过 getDocIdSet 得到文档号之外，bitSpans 函数得到的 SpanFilterResult 还包含位置信息，可以用于在 FilterQuery 中起过滤作用。

```
public DocIdSet getDocIdSet(IndexReader reader) throws IOException {
  SpanFilterResult result = bitSpans(reader);
  return result.getDocIdSet();
}
```

```
public SpanFilterResult bitSpans(IndexReader reader) throws IOException {
  final OpenBitSet bits = new OpenBitSet(reader.maxDoc());
  Spans spans = query.getSpans(reader);
  List<SpanFilterResult.PositionInfo> tmp = new ArrayList<SpanFilterResult.PositionInfo>(20);
  int currentDoc = -1;
  SpanFilterResult.PositionInfo currentInfo = null;
  while (spans.next())
  {
```

```
    //将 docid 放入 bitset
    int doc = spans.doc();
    bits.set(doc);
    if (currentDoc != doc)
    {
      currentInfo = new SpanFilterResult.PositionInfo(doc);
      tmp.add(currentInfo);
      currentDoc = doc;
    }
    //将 start 和 end 信息放入 PositionInfo
    currentInfo.addPosition(spans.start(), spans.end());
  }
  return new SpanFilterResult(bits, tmp);
}
```

6.7.2、CachingSpanFilter

由 Filter 的接口 DocIdSet getDocIdSet(IndexReader reader)得知，一个 docid 的 bitset 是同一个 reader 相对应的。

有前面对 docid 的描述可知，其仅对一个打开的 reader 有意义。

CachingSpanFilter 有一个成员变量 Map<IndexReader,SpanFilterResult> cache 保存从 reader 到 SpanFilterResult 的映射，另一个成员变量 SpanFilter filter 用于缓存不命中的时候得到 SpanFilterResult。

其 getDocIdSet 如下：

```
public DocIdSet getDocIdSet(IndexReader reader) throws IOException {
  SpanFilterResult result = getCachedResult(reader);
  return result != null ? result.getDocIdSet() : null;
```

```
}
private SpanFilterResult getCachedResult(IndexReader reader) throws IOException {
  lock.lock();
  try {
    if (cache == null) {
      cache = new WeakHashMap<IndexReader,SpanFilterResult>();
    }
    //如果缓存命中，则返回缓存中的结果。
    final SpanFilterResult cached = cache.get(reader);
    if (cached != null) return cached;
  } finally {
    lock.unlock();
  }
  //如果缓存不命中，则用 SpanFilter 直接从 reader 中得到结果。
  final SpanFilterResult result = filter.bitSpans(reader);
  lock.lock();
  try {
    //将新得到的结果放入缓存
    cache.put(reader, result);
  } finally {
    lock.unlock();
  }
  return result;

}
```

第十章：Lucene 的分词器 Analyzer

1、抽象类 Analyzer

其主要包含两个接口，用于生成 TokenStream：

- TokenStream tokenStream(String fieldName, Reader reader);
- TokenStream reusableTokenStream(String fieldName, Reader reader) ;

所谓 TokenStream，后面我们会讲到，是一个由分词后的 Token 结果组成的流，能够不断的得到下一个分成的 Token。

为了提高性能，使得在同一个线程中无需再生成新的 TokenStream 对象，老的可以被重用，所以有 reusableTokenStream 一说。

所以 Analyzer 中有 CloseableThreadLocal< Object > tokenStreams = new CloseableThreadLocal< Object >(); 成员变量，保存当前线程原来创建过的 TokenStream，可用函数 setPreviousTokenStream 设定，用函数 getPreviousTokenStream 得到。

在 reusableTokenStream 函数中，往往用 getPreviousTokenStream 得到老的 TokenStream 对象，然后将 TokenStream 对象 reset 以下，从而可以从新开始得到 Token 流。

让我们看一下最简单的一个 Analyzer:

```
public final class SimpleAnalyzer extends Analyzer {
  @Override
  public TokenStream tokenStream(String fieldName, Reader reader) {
   //返回的是将字符串最小化，并且按照空格分隔的 Token
   return new LowerCaseTokenizer(reader);
  }
  @Override
  public TokenStream reusableTokenStream(String fieldName, Reader reader) throws IOException {
   //得到上一次使用的 TokenStream，如果没有则生成新的，并且用 setPreviousTokenStream 放入成员变量，使得下一个可用。
```

```
    Tokenizer tokenizer = (Tokenizer) getPreviousTokenStream();
    if (tokenizer == null) {
      tokenizer = new LowerCaseTokenizer(reader);
      setPreviousTokenStream(tokenizer);
    } else
      //如果上一次生成过 TokenStream，则 reset。
      tokenizer.reset(reader);
    return tokenizer;
  }
}
```

2、TokenStream 抽象类

TokenStream 主要包含以下几个方法：

- boolean incrementToken()用于得到下一个 Token。
- public void reset() 使得此 TokenStrean 可以重新开始返回各个分词。

和原来的 TokenStream 返回一个 Token 对象不同，Lucene 3.0 的 TokenStream 已经不返回 Token 对象了，那么如何保存下一个 Token 的信息呢。

在 Lucene 3.0 中，TokenStream 是继承于 AttributeSource，其包含 Map，保存从 class 到对象的映射，从而可以保存不同类型的对象的值。

在 TokenStream 中，经常用到的对象是 TermAttributeImpl，用来保存 Token 字符串；PositionIncrementAttributeImpl 用来保存位置信息；OffsetAttributeImpl 用来保存偏移量信息。所以当生成 TokenStream 的时候，往往调用 AttributeImpl tokenAtt = (AttributeImpl) addAttribute(TermAttribute.class)将 TermAttributeImpl 添加到 Map 中，并保存一个成员变量。在 incrementToken()中，将下一个 Token 的信息写入当前的 tokenAtt，然后使用 TermAttributeImpl.term()得到 Token 的字符串。

3、几个具体的 TokenStream

在索引的时候，添加域的时候，可以指定 Analyzer，使其生成 TokenStream，也可以直接指定 TokenStream：

public Field(String name, TokenStream tokenStream);

下面介绍两个单独使用的 TokenStream

3.1、NumericTokenStream

上一节介绍 NumericRangeQuery 的时候，在生成 NumericField 的时候，其会使用 NumericTokenStream，其 incrementToken 如下：

```
public boolean incrementToken() {
  if (valSize == 0)
    throw new IllegalStateException("call set???Value() before usage");
  if (shift >= valSize)
    return false;
  clearAttributes();
  //虽然 NumericTokenStream 欲保存数字，然而 Lucene 的 Token 只能保存字符串，因而要将数字编码为字符串，然后存入索引。
  final char[] buffer;
  switch (valSize) {
    //首先分配 TermBuffer，然后将数字编码为字符串
    case 64:
      buffer = termAtt.resizeTermBuffer(NumericUtils.BUF_SIZE_LONG);
      termAtt.setTermLength(NumericUtils.longToPrefixCoded(value, shift, buffer));
      break;
    case 32:
      buffer = termAtt.resizeTermBuffer(NumericUtils.BUF_SIZE_INT);
```

```
    termAtt.setTermLength(NumericUtils.intToPrefixCoded((int) value, shift, buffer));
    break;
   default:
    throw new IllegalArgumentException("valSize must be 32 or 64");
  }
  typeAtt.setType((shift == 0) ? TOKEN_TYPE_FULL_PREC : TOKEN_TYPE_LOWER_PREC);
  posIncrAtt.setPositionIncrement((shift == 0) ? 1 : 0);
  shift += precisionStep;
  return true;
}
```

```
public static int intToPrefixCoded(final int val, final int shift, final char[] buffer) {
  if (shift>31 || shift<0)
   throw new IllegalArgumentException("Illegal shift value, must be 0..31");
  int nChars = (31-shift)/7 + 1, len = nChars+1;
  buffer[0] = (char)(SHIFT_START_INT + shift);
  int sortableBits = val ^ 0x80000000;
  sortableBits >>>= shift;
  while (nChars>=1) {
   //int 按照每七位组成一个 utf-8 的编码，并且字符串大小比较的顺序同 int 大小比较的顺序完全相同。
    buffer[nChars--] = (char)(sortableBits & 0x7f);
    sortableBits >>>= 7;
  }
  return len;
}
```

3.2、SingleTokenTokenStream

SingleTokenTokenStream 顾名思义就是此 TokenStream 仅仅包含一个 Token，多用于保存一篇文档仅有一个的信息，如 id，如 time 等，这些信息往往被保存在一个特殊的 Token(如 ID:ID, TIME:TIME)的倒排表的 payload 中的，这样可以使用跳表来增加访问速度。

所以 SingleTokenTokenStream 返回的 Token 则不是 id 或者 time 本身，而是特殊的 Token，"ID:ID", "TIME:TIME"，而是将 id 的值或者 time 的值放入 payload 中。

```
//索引的时候
int id = 0; //用户自己的文档号
String tokenstring = "ID";
byte[] value = idToBytes(); //将 id 装换为 byte 数组
Token token = new Token(tokenstring, 0, tokenstring.length);
token.setPayload(new Payload(value));
SingleTokenTokenStream tokenstream = new SingleTokenTokenStream(token);
Document doc = new Document();
doc.add(new Field("ID", tokenstream));
……
//当得到 Lucene 的文档号 docid，并不想构造 Document 对象就得到用户的文档号时
TermPositions tp = reader.termPositions("ID:ID");
boolean ret = tp.skipTo(docid);
tp.nextPosition();
int payloadlength = tp.getPayloadLength();
byte[] payloadBuffer = new byte[payloadlength];
tp.getPayload(payloadBuffer, 0);
int id = bytesToID(); //将 payloadBuffer 转换为用户 id
```

4、Tokenizer 也是一种 TokenStream

```
public abstract class Tokenizer extends TokenStream {
  protected Reader input;
  protected Tokenizer(Reader input) {
    this.input = CharReader.get(input);
  }
  public void reset(Reader input) throws IOException {
    this.input = input;
  }
}
```

以下重要的 Tokenizer 如下，我们将一一解析：

- CharTokenizer
  - LetterTokenizer
    - LowerCaseTokenizer
  - WhitespaceTokenizer
- ChineseTokenizer
- CJKTokenizer
- EdgeNGramTokenizer
- KeywordTokenizer
- NGramTokenizer
- SentenceTokenizer
- StandardTokenizer

4.1、CharTokenizer

CharTokenizer 是一个抽象类，用于对字符串进行分词。

在构造函数中，生成了 TermAttribute 和 OffsetAttribute 两个属性，说明分词后除了返回分词后的字符外，还要返回 offset。

```
offsetAtt = addAttribute(OffsetAttribute.class);
termAtt = addAttribute(TermAttribute.class);
```

其 incrementToken 函数如下：

```
public final boolean incrementToken() throws IOException {
  clearAttributes();
  int length = 0;
  int start = bufferIndex;
  char[] buffer = termAtt.termBuffer();
  while (true) {
   //不断读取 reader 中的字符到 buffer 中
   if (bufferIndex >= dataLen) {
    offset += dataLen;
    dataLen = input.read(ioBuffer);
    if (dataLen == -1) {
     dataLen = 0;
     if (length > 0)
      break;
     else
      return false;
    }
    bufferIndex = 0;
   }
   //然后逐一遍历 buffer 中的字符
   final char c = ioBuffer[bufferIndex++];
   //如果是一个 token 字符，则 normalize 后接着取下一个字符，否则当前 token 结束。
   if (isTokenChar(c)) {
    if (length == 0)
     start = offset + bufferIndex - 1;
    else if (length == buffer.length)
```

```
      buffer = termAtt.resizeTermBuffer(1+length);
    buffer[length++] = normalize(c);
    if (length == MAX_WORD_LEN)
      break;
   } else if (length > 0)
    break;
  }
  termAtt.setTermLength(length);
  offsetAtt.setOffset(correctOffset(start), correctOffset(start+length));
  return true;
}
```

CharTokenizer 是一个抽象类，其 isTokenChar 函数和 normalize 函数由子类实现。

其子类 WhitespaceTokenizer 实现了 isTokenChar 函数：

```
//当遇到空格的时候，当前 token 结束
protected boolean isTokenChar(char c) {
  return !Character.isWhitespace(c);
}
```

其子类 LetterTokenizer 如下实现 isTokenChar 函数：

```
protected boolean isTokenChar(char c) {
  return Character.isLetter(c);
}
```

LetterTokenizer 的子类 LowerCaseTokenizer 实现了 normalize 函数，将字符串转换为小写：

```
protected char normalize(char c) {
  return Character.toLowerCase(c);
}
```

4.2、ChineseTokenizer

其在初始化的时候，添加 TermAttribute 和 OffsetAttribute。

其 incrementToken 实现如下：

```
public boolean incrementToken() throws IOException {
    clearAttributes();
    length = 0;
    start = offset;
    while (true) {
       final char c;
       offset++;
       if (bufferIndex >= dataLen) {
          dataLen = input.read(ioBuffer);
          bufferIndex = 0;
       }
       if (dataLen == -1) return flush();
       else
          c = ioBuffer[bufferIndex++];
       switch(Character.getType(c)) {
       //如果是英文下小写字母或数字的时候，则属于同一个 Token，push 到 buffer 中
       case Character.DECIMAL_DIGIT_NUMBER:
       case Character.LOWERCASE_LETTER:
       case Character.UPPERCASE_LETTER:
          push(c);
          if (length == MAX_WORD_LEN) return flush();
          break;
       //中文属于 OTHER_LETTER，当出现中文字符的时候，则上一个 Token 结束，并
将当前字符 push 到 buffer 中
```

```
      case Character.OTHER_LETTER:
        if (length>0) {
          bufferIndex--;
          offset--;
          return flush();
        }
        push(c);
        return flush();
      default:
        if (length>0) return flush();
        break;
      }
    }
}
```

4.3、KeywordTokenizer

KeywordTokenizer 是将整个字符作为一个 Token 返回的。

其 incrementToken 函数如下：

```
public final boolean incrementToken() throws IOException {
  if (!done) {
    clearAttributes();
    done = true;
    int upto = 0;
    char[] buffer = termAtt.termBuffer();
    //将字符串全部读入 buffer，然后返回。
    while (true) {
```

```
      final int length = input.read(buffer, upto, buffer.length-upto);
      if (length == -1) break;
      upto += length;
      if (upto == buffer.length)
        buffer = termAtt.resizeTermBuffer(1+buffer.length);
    }
    termAtt.setTermLength(upto);
    finalOffset = correctOffset(upto);
    offsetAtt.setOffset(correctOffset(0), finalOffset);
    return true;
  }
  return false;
}
```

4.4、CJKTokenizer

其 incrementToken 函数如下：

```
public boolean incrementToken() throws IOException {
    clearAttributes();
    while(true) {
      int length = 0;
      int start = offset;
      while (true) {
        //得到当前的字符，及其所属的 Unicode 块
        char c;
        Character.UnicodeBlock ub;
        offset++;
```

```
    if (bufferIndex >= dataLen) {
      dataLen = input.read(ioBuffer);
      bufferIndex = 0;
    }
    if (dataLen == -1) {
      if (length > 0) {
        if (preIsTokened == true) {
          length = 0;
          preIsTokened = false;
        }
        break;
      } else {
        return false;
      }
    } else {
      c = ioBuffer[bufferIndex++];
      ub = Character.UnicodeBlock.of(c);
    }
    //如果当前字符输入 ASCII 码
        if ((ub == Character.UnicodeBlock.BASIC_LATIN) || (ub ==
Character.UnicodeBlock.HALFWIDTH_AND_FULLWIDTH_FORMS)) {
      if (ub == Character.UnicodeBlock.HALFWIDTH_AND_FULLWIDTH_FORMS) {
        int i = (int) c;
        if (i >= 65281 && i <= 65374) {
          //将半型及全型形式 Unicode 转变为普通的 ASCII 码
          i = i - 65248;
          c = (char) i;
        }
```

```
    }
    //如果当前字符是字符或者"_" "+" "#"
    if (Character.isLetterOrDigit(c) || ((c == '_') || (c == '+') || (c == '#'))) {
        if (length == 0) {
            start = offset - 1;
        } else if (tokenType == DOUBLE_TOKEN_TYPE) {
            offset--;
            bufferIndex--;
            if (preIsTokened == true) {
                length = 0;
                preIsTokened = false;
                break;
            } else {
                break;
            }
        }
        //将当前字符放入 buffer
        buffer[length++] = Character.toLowerCase(c);
        tokenType = SINGLE_TOKEN_TYPE;
        if (length == MAX_WORD_LEN) {
            break;
        }
    } else if (length > 0) {
        if (preIsTokened == true) {
            length = 0;
            preIsTokened = false;
        } else {
            break;
```

```
            }
          }
        } else {
          //如果非 ASCII 字符
          if (Character.isLetter(c)) {
            if (length == 0) {
              start = offset - 1;
              buffer[length++] = c;
              tokenType = DOUBLE_TOKEN_TYPE;
            } else {
              if (tokenType == SINGLE_TOKEN_TYPE) {
                offset--;
                bufferIndex--;
                break;
              } else {
                //非 ASCII 码字符，两个字符作为一个 Token
                    //(如"中华人民共和国"分词为"中华"，"华人"，"人民"，"民共"，"
共和"，"和国")
                buffer[length++] = c;
                tokenType = DOUBLE_TOKEN_TYPE;
                if (length == 2) {
                  offset--;
                  bufferIndex--;
                  preIsTokened = true;
                  break;
                }
              }
            }
```

```
        } else if (length > 0) {
          if (preIsTokened == true) {
            length = 0;
            preIsTokened = false;
          } else {
            break;
          }
        }
      }
    }
    if (length > 0) {
      termAtt.setTermBuffer(buffer, 0, length);
      offsetAtt.setOffset(correctOffset(start), correctOffset(start+length));
      typeAtt.setType(TOKEN_TYPE_NAMES[tokenType]);
      return true;
    } else if (dataLen == -1) {
      return false;
    }
  }
}
```

4.5、SentenceTokenizer

其是按照如下的标点来拆分句子："。,！？；,!?;"

让我们来看下面的例子：

```
String s = "据纽约时报周三报道称，苹果已经超过微软成为美国最有价值的 科技公司。这是一个不容忽视的转折点。";
StringReader sr = new StringReader(s);
```

```
SentenceTokenizer tokenizer = new SentenceTokenizer(sr);
boolean hasnext = tokenizer.incrementToken();
while(hasnext){
  TermAttribute ta = tokenizer.getAttribute(TermAttribute.class);
  System.out.println(ta.term());
  hasnext = tokenizer.incrementToken();
}
```

结果为：

```
据纽约时报周三报道称，
苹果已经超过微软成为美国最有价值的
科技公司。
这是一个不容忽视的转折点。
```

其 incrementToken 函数如下：

```
public boolean incrementToken() throws IOException {
  clearAttributes();
  buffer.setLength(0);
  int ci;
  char ch, pch;
  boolean atBegin = true;
  tokenStart = tokenEnd;
  ci = input.read();
  ch = (char) ci;
  while (true) {
   if (ci == -1) {
     break;
   } else if (PUNCTION.indexOf(ch) != -1) {
     //出现标点符号，当前句子结束，返回当前 Token
     buffer.append(ch);
```

```
    tokenEnd++;
    break;
   } else if (atBegin && Utility.SPACES.indexOf(ch) != -1) {
    tokenStart++;
    tokenEnd++;
    ci = input.read();
    ch = (char) ci;
   } else {
    buffer.append(ch);
    atBegin = false;
    tokenEnd++;
    pch = ch;
    ci = input.read();
    ch = (char) ci;
    //当连续出现两个空格，或者\r\n 的时候，则当前句子结束，返回当前 Token
    if (Utility.SPACES.indexOf(ch) != -1
      && Utility.SPACES.indexOf(pch) != -1) {
     tokenEnd++;
     break;
    }
   }
  }
  if (buffer.length() == 0)
   return false;
  else {
   termAtt.setTermBuffer(buffer.toString());
   offsetAtt.setOffset(correctOffset(tokenStart), correctOffset(tokenEnd));
   typeAtt.setType("sentence");
```

```
    return true;
  }
}
```

5、TokenFilter 也是一种 TokenStream

来对 Tokenizer 后的 Token 作过滤，其使用的是装饰者模式。

```
public abstract class TokenFilter extends TokenStream {
  protected final TokenStream input;
  protected TokenFilter(TokenStream input) {
    super(input);
    this.input = input;
  }
}
```

5.1、ChineseFilter

其 incrementToken 函数如下：

```
public boolean incrementToken() throws IOException {
    while (input.incrementToken()) {
        char text[] = termAtt.termBuffer();
        int termLength = termAtt.termLength();
      //如果不被停词表过滤掉
        if (!stopTable.contains(text, 0, termLength)) {
            switch (Character.getType(text[0])) {
          //如果是英文且长度超过一，则算一个 Token，否则不算一个 Token
```

```
        case Character.LOWERCASE_LETTER:
        case Character.UPPERCASE_LETTER:
          if (termLength>1) {
            return true;
          }
          break;
        //如果是中文则算一个 Token
        case Character.OTHER_LETTER:
          return true;
        }
      }
    }
    return false;
}
```

举例：

```
String s = "Javaeye: IT 外企那点儿事。1.外企也就那么会儿事。";
StringReader sr = new StringReader(s);
ChineseTokenizer ct = new ChineseTokenizer(sr);
ChineseFilter filter = new ChineseFilter(ct);
boolean hasnext = filter.incrementToken();
while(hasnext){
  TermAttribute ta = filter.getAttribute(TermAttribute.class);
  System.out.println(ta.term());
  hasnext = filter.incrementToken();
}
```

结果为：

```
javaeye
外
```

```
企
那
点
儿
事
外
企
也
就
那
么
会
儿
事
```

5.2、LengthFilter

其 incrementToken 函数如下：

```
public final boolean incrementToken() throws IOException {
  while (input.incrementToken()) {
    int len = termAtt.termLength();
    //当当前字符串的长度在指定范围内的时候则返回。
    if (len >= min && len <= max) {
       return true;
    }
  }
  return false;
```

```
}
```

举例如下：

```
String s = "a it has this there string english analyzer";
StringReader sr = new StringReader(s);
WhitespaceTokenizer wt = new WhitespaceTokenizer(sr);
LengthFilter filter = new LengthFilter(wt, 4, 7);
boolean hasnext = filter.incrementToken();
while(hasnext){
  TermAttribute ta = filter.getAttribute(TermAttribute.class);
  System.out.println(ta.term());
  hasnext = filter.incrementToken();
}
```

结果如下：

```
this
there
string
english
```

5.3、LowerCaseFilter

其 incrementToken 函数如下：

```
public final boolean incrementToken() throws IOException {
  if (input.incrementToken()) {
    final char[] buffer = termAtt.termBuffer();
    final int length = termAtt.termLength();
    for(int i=0;i<length;i++)
     //转小写
```

```
      buffer[i] = Character.toLowerCase(buffer[i]);
    return true;
  } else
    return false;
}
```

5.4、NumericPayloadTokenFilter

```
public final boolean incrementToken() throws IOException {
  if (input.incrementToken()) {
    if (typeAtt.type().equals(typeMatch))
      //设置 payload
      payloadAtt.setPayload(thePayload);
    return true;
  } else {
    return false;
  }
}
```

5.5、PorterStemFilter

其成员变量 PorterStemmer stemmer，其实现著名的 stemming 算法是 The Porter Stemming Algorithm，其主页为 http://tartarus.org/~martin/PorterStemmer/，也可查看其论文 http://tartarus.org/~martin/PorterStemmer/def.txt。

通过以下网页可以进行简单的测试：**Porter's Stemming Algorithm Online**[http://facweb.cs.depaul.edu/mobasher/classes/csc575/porter.html]

cars –> car

driving –> drive

tokenization –> token

其 incrementToken 函数如下：

```
public final boolean incrementToken() throws IOException {
  if (!input.incrementToken())
    return false;
  if (stemmer.stem(termAtt.termBuffer(), 0, termAtt.termLength()))
    termAtt.setTermBuffer(stemmer.getResultBuffer(), 0, stemmer.getResultLength());
  return true;
}
```

举例：

```
String s = "Tokenization is the process of breaking a stream of text up into meaningful elements
called tokens.";
StringReader sr = new StringReader(s);
LowerCaseTokenizer lt = new LowerCaseTokenizer(sr);
PorterStemFilter filter = new PorterStemFilter(lt);
boolean hasnext = filter.incrementToken();
while(hasnext){
  TermAttribute ta = filter.getAttribute(TermAttribute.class);
  System.out.println(ta.term());
  hasnext = filter.incrementToken();
}
```

结果为：

```
token
is
the
process
of
```

```
break
a
stream
of
text
up
into
meaning
element
call
token
```

5.6、ReverseStringFilter

```
public boolean incrementToken() throws IOException {
  if (input.incrementToken()) {
    int len = termAtt.termLength();
    if (marker != NOMARKER) {
      len++;
      termAtt.resizeTermBuffer(len);
      termAtt.termBuffer()[len - 1] = marker;
    }
    //将 token 反转
    reverse( termAtt.termBuffer(), len );
    termAtt.setTermLength(len);
    return true;
  } else {
```

```
    return false;
  }
}
```

```
public static void reverse( char[] buffer, int start, int len ){
  if( len <= 1 ) return;
  int num = len>>1;
  for( int i = start; i < ( start + num ); i++ ){
    char c = buffer[i];
    buffer[i] = buffer[start * 2 + len - i - 1];
    buffer[start * 2 + len - i - 1] = c;
  }
}
```

举例：

```
String s = "Tokenization is the process of breaking a stream of text up into meaningful elements called tokens.";
StringReader sr = new StringReader(s);
LowerCaseTokenizer lt = new LowerCaseTokenizer(sr);
ReverseStringFilter filter = new ReverseStringFilter(lt);
boolean hasnext = filter.incrementToken();
while(hasnext){
  TermAttribute ta = filter.getAttribute(TermAttribute.class);
  System.out.println(ta.term());
  hasnext = filter.incrementToken();
}
```

结果为：

```
noitazinekot
si
eht
```

```
ssecorp
fo
gnikaerb
a
maerts
fo
txet
pu
otni
lufgninaem
stnemele
dellac
snekot
```

5.7、SnowballFilter

其包含成员变量 SnowballProgram stemmer，其是一个抽象类，其子类有 EnglishStemmer 和 PorterStemmer 等。

```
public final boolean incrementToken() throws IOException {
  if (input.incrementToken()) {
    String originalTerm = termAtt.term();
    stemmer.setCurrent(originalTerm);
    stemmer.stem();
    String finalTerm = stemmer.getCurrent();
    if (!originalTerm.equals(finalTerm))
      termAtt.setTermBuffer(finalTerm);
    return true;
```

```
  } else {
    return false;
  }
}
```

举例：

```
String s = "Tokenization is the process of breaking a stream of text up into meaningful elements called tokens.";
StringReader sr = new StringReader(s);
LowerCaseTokenizer lt = new LowerCaseTokenizer(sr);
SnowballFilter filter = new SnowballFilter(lt, new EnglishStemmer());
boolean hasnext = filter.incrementToken();
while(hasnext){
  TermAttribute ta = filter.getAttribute(TermAttribute.class);
  System.out.println(ta.term());
  hasnext = filter.incrementToken();
}
```

结果如下：

```
token
is
the
process
of
break
a
stream
of
text
up
```

```
into
meaning
element
call
token
```

5.8、TeeSinkTokenFilter

TeeSinkTokenFilter 可以使得已经分好词的 Token 全部或者部分的被保存下来，用于生成另一个 TokenStream 可以保存在其他的域中。

我们可用如下的语句生成一个 TeeSinkTokenFilter：

```
TeeSinkTokenFilter source = new TeeSinkTokenFilter(new WhitespaceTokenizer(reader));
```

然后使用函数 newSinkTokenStream()或者 newSinkTokenStream(SinkFilter filter)生成一个 SinkTokenStream：

```
TeeSinkTokenFilter.SinkTokenStream sink = source.newSinkTokenStream();
```

其中在 newSinkTokenStream(SinkFilter filter)函数中，将新生成的 SinkTokenStream 保存在 TeeSinkTokenFilter 的成员变量 sinks 中。

在 TeeSinkTokenFilter 的 incrementToken 函数中：

```
public boolean incrementToken() throws IOException {
  if (input.incrementToken()) {
   //对于每一个 Token，依次遍历成员变量 sinks
   AttributeSource.State state = null;
   for (WeakReference<SinkTokenStream> ref : sinks) {
    //对于每一个 SinkTokenStream，首先调用函数 accept 看是否接受，如果接受则将此 Token 也加入此 SinkTokenStream。
     final SinkTokenStream sink = ref.get();
     if (sink != null) {
```

```
      if (sink.accept(this)) {
        if (state == null) {
          state = this.captureState();
        }
        sink.addState(state);
      }
    }
   }
   return true;
  }
  return false;
}
```

SinkTokenStream.accept 调用 SinkFilter.accept，对于默认的 ACCEPT_ALL_FILTER 则接受所有的 Token：

```
private static final SinkFilter ACCEPT_ALL_FILTER = new SinkFilter() {
  @Override
  public boolean accept(AttributeSource source) {
   return true;
  }
};
```

这样 SinkTokenStream 就能够保存下所有 WhitespaceTokenizer 分好的 Token。

当我们使用比较复杂的分成系统的时候，分词一篇文章往往需要耗费比较长的时间，当分好的词需要再次使用的时候，再分一次词实在太浪费了，于是可以用上述的例子，将分好的词保存在一个 TokenStream 里面就可以了。

如下面的例子：

```
String s = "this is a book";
StringReader reader = new StringReader(s);
TeeSinkTokenFilter source = new TeeSinkTokenFilter(new WhitespaceTokenizer(reader));
```

```
TeeSinkTokenFilter.SinkTokenStream sink = source.newSinkTokenStream();
boolean hasnext = source.incrementToken();
while(hasnext){
  TermAttribute ta = source.getAttribute(TermAttribute.class);
  System.out.println(ta.term());
  hasnext = source.incrementToken();
}
System.out.println("-----------------------------------------");
hasnext = sink.incrementToken();
while(hasnext){
  TermAttribute ta = sink.getAttribute(TermAttribute.class);
  System.out.println(ta.term());
  hasnext = sink.incrementToken();
}
```

结果为：

```
this
is
a
book
-----------------------------------------
this
is
a
book
```

当然有时候我们想在分好词的一系列 Token 中，抽取我们想要的一些实体，保存下来。

如下面的例子：

```
  String s = "Japan will always balance its national interests between China and America.";
  StringReader reader = new StringReader(s);
```

```
TeeSinkTokenFilter source = new TeeSinkTokenFilter(new LowerCaseTokenizer(reader));
//一个集合，保存所有的国家名称
final HashSet<String> countryset = new HashSet<String>();
countryset.add("japan");
countryset.add("china");
countryset.add("america");
countryset.add("korea");
SinkFilter countryfilter = new SinkFilter() {
 @Override
 public boolean accept(AttributeSource source) {
  TermAttribute ta = source.getAttribute(TermAttribute.class);
  //如果在国家名称列表中，则保留
  if(countryset.contains(ta.term())){
   return true;
  }
  return false;
 }
};
TeeSinkTokenFilter.SinkTokenStream sink = source.newSinkTokenStream(countryfilter);
//由 LowerCaseTokenizer 对语句进行分词，并把其中的国家名称保存在 SinkTokenStream 中
boolean hasnext = source.incrementToken();
while(hasnext){
 TermAttribute ta = source.getAttribute(TermAttribute.class);
 System.out.println(ta.term());
 hasnext = source.incrementToken();
}
System.out.println("-----------------------------------------");
```

```
  hasnext = sink.incrementToken();
  while(hasnext){
    TermAttribute ta = sink.getAttribute(TermAttribute.class);
    System.out.println(ta.term());
    hasnext = sink.incrementToken();
  }
}
```

结果为：

```
japan
will
always
balance
its
national
interests
between
china
and
america
-------------------------------------------
japan
china
america
```

6、不同的 Analyzer 就是组合不同的 Tokenizer 和 TokenFilter 得到最后的 TokenStream

6.1、ChineseAnalyzer

```
public final TokenStream tokenStream(String fieldName, Reader reader) {
   //按字分词，并过滤停词，标点，英文
    TokenStream result = new ChineseTokenizer(reader);
    result = new ChineseFilter(result);
    return result;
}
```

举例："This year, president Hu 科学发展观" 被分词为 "year","president","hu","科","学","发","展","观"

6.2、CJKAnalyzer

```
public final TokenStream tokenStream(String fieldName, Reader reader) {
   //每两个字组成一个词，并去除停词
     return  new  StopFilter(StopFilter.getEnablePositionIncrementsVersionDefault(matchVersion),
new CJKTokenizer(reader), stopTable);
}
```

举例："This year, president Hu 科学发展观" 被分词为"year","president","hu","科学","学发","发展","展观"。

6.3、PorterStemAnalyzer

```
public TokenStream tokenStream(String fieldName, Reader reader) {
    //将转为小写的 token，利用 porter 算法进行 stemming
    return new PorterStemFilter(new LowerCaseTokenizer(reader));
}
```

6.4、SmartChineseAnalyzer

```
public TokenStream tokenStream(String fieldName, Reader reader) {
    //先分句子
    TokenStream result = new SentenceTokenizer(reader);
    //句子中分词组
    result = new WordTokenFilter(result);
    //用 porter 算法进行 stemming
    result = new PorterStemFilter(result);
    //去停词
    if (!stopWords.isEmpty()) {
      result = new StopFilter(StopFilter.getEnablePositionIncrementsVersionDefault(matchVersion),
result, stopWords, false);
    }
    return result;
}
```

6.5、SnowballAnalyzer

```
public TokenStream tokenStream(String fieldName, Reader reader) {
```

```
    //使用标准的分词器
    TokenStream result = new StandardTokenizer(matchVersion, reader);
    //标准的过滤器
    result = new StandardFilter(result);
    //转换为小写
    result = new LowerCaseFilter(result);
    //去停词
    if (stopSet != null)
      result = new StopFilter(StopFilter.getEnablePositionIncrementsVersionDefault(matchVersion),
result, stopSet);
    //根据设定的 stemmer 进行 stemming
    result = new SnowballFilter(result, name);
    return result;
}
```

7、Lucene 的标准分词器

7.1、StandardTokenizerImpl.jflex

和 QueryParser 类似，标准分词器也需要词法分析，在原来的版本中，也是用 javacc，当前的版本中，使用的是 jflex。

jflex 也是一个词法及语法分析器的生成器，它主要包括三部分，由%%分隔：

- 用户代码部分：多为 package 或者 import
- 选项及词法声明
- 语法规则声明

用于生成标准分词器的 flex 文件尾 StandardTokenizerImpl.jflex，如下:

```
import org.apache.lucene.analysis.Token;
import org.apache.lucene.analysis.tokenattributes.TermAttribute;
%% //以上是用户代码部分，以下是选项及词法声明
%class StandardTokenizerImpl //类名
%unicode
%integer //下面函数的返回值
%function getNextToken //进行词法及语法分析的函数
%pack
%char
%{ //此之间的代码之间拷贝到生成的 java 文件中
public static final int ALPHANUM          = StandardTokenizer.ALPHANUM;
public static final int APOSTROPHE        = StandardTokenizer.APOSTROPHE;
public static final int ACRONYM           = StandardTokenizer.ACRONYM;
public static final int COMPANY           = StandardTokenizer.COMPANY;
public static final int EMAIL             = StandardTokenizer.EMAIL;
public static final int HOST              = StandardTokenizer.HOST;
public static final int NUM               = StandardTokenizer.NUM;
public static final int CJ                = StandardTokenizer.CJ;
public static final int ACRONYM_DEP       = StandardTokenizer.ACRONYM_DEP;
public static final String [] TOKEN_TYPES = StandardTokenizer.TOKEN_TYPES;
public final int yychar()
{
    return yychar;
}
final void getText(Token t) {
  t.setTermBuffer(zzBuffer, zzStartRead, zzMarkedPos-zzStartRead);
}
final void getText(TermAttribute t) {
```

```
  t.setTermBuffer(zzBuffer, zzStartRead, zzMarkedPos-zzStartRead);
}
%}
THAI       = [\u0E00-\u0E59]
//一系列字母和数字的组合
ALPHANUM    = ({LETTER}|{THAI}|[:digit:])+
//省略符号，如 you're
APOSTROPHE =   {ALPHA} ("'" {ALPHA})+
//缩写，如 U.S.A.
ACRONYM     =   {LETTER} "." ({LETTER} ".")+
ACRONYM_DEP     = {ALPHANUM} "." ({ALPHANUM} ".")+
//  公司名称如 AT&T，Excite@Home.
COMPANY     =   {ALPHA} ("&"|"@") {ALPHA}
//  邮箱地址
EMAIL =   {ALPHANUM} (("."|"-"|"_") {ALPHANUM})* "@" {ALPHANUM} (("."|"-") {ALPHANUM})+
//  主机名
HOST   =   {ALPHANUM} (("."){ALPHANUM})+
NUM   = ({ALPHANUM} {P} {HAS_DIGIT}
        | {HAS_DIGIT} {P} {ALPHANUM}
        | {ALPHANUM} ({P} {HAS_DIGIT} {P} {ALPHANUM})+
        | {HAS_DIGIT} ({P} {ALPHANUM} {P} {HAS_DIGIT})+
        | {ALPHANUM} {P} {HAS_DIGIT} ({P} {ALPHANUM} {P} {HAS_DIGIT})+
        | {HAS_DIGIT} {P} {ALPHANUM} ({P} {HAS_DIGIT} {P} {ALPHANUM})+)
//标点
P   = ("_"|"-"|"/"|"."|",")
//至少包含一个数字的字符串
HAS_DIGIT   = ({LETTER}|[:digit:])* [:digit:] ({LETTER}|[:digit:])*
ALPHA   = ({LETTER})+
```

```
//所谓字符，即出去所有的非字符的 ASCII 及中日文。
LETTER = !(![:letter:]|{CJ})
//中文或者日文
CJ
[\u3100-\u312f\u3040-\u309F\u30A0-\u30FF\u31F0-\u31FF\u3300-\u337f\u3400-\u4dbf\u4e00-\u9fff\uf900-\ufaff\u
//空格
WHITESPACE = \r\n | [ \r\n\t\f]
%% //以下是语法规则部分，由于是分词器，因而不需要进行语法分析，则全部原样返回
{ALPHANUM}                                   { return ALPHANUM; }
{APOSTROPHE}                                 { return APOSTROPHE; }
{ACRONYM}                                    { return ACRONYM; }
{COMPANY}                                    { return COMPANY; }
{EMAIL}                                      { return EMAIL; }
{HOST}                                       { return HOST; }
{NUM}                                        { return NUM; }
{CJ}                                         { return CJ; }
{ACRONYM_DEP}                                { return ACRONYM_DEP; }
```

下面我们看下面的例子，来说明 StandardTokenizerImpl 的功能：

```
String s = "I'm Juexian, my email is forfuture1978@gmail.com. My ip address is 192.168.0.1, AT&T
and I.B.M are all great companies.";
StringReader reader = new StringReader(s);
StandardTokenizerImpl impl = new StandardTokenizerImpl(reader);
while(impl.getNextToken() != StandardTokenizerImpl.YYEOF){
    TermAttributeImpl ta = new TermAttributeImpl();
    impl.getText(ta);
    System.out.println(ta.term());
}
```

```
结果为：
I'm
Juexian
my
email
is
forfuture1978@gmail.com
My
ip
address
is
192.168.0.1
AT&T
and
I.B.M
are
all
great
companies
```

7.2、StandardTokenizer

其有一个成员变量 StandardTokenizerImpl scanner;

其 incrementToken 函数如下：

```
public final boolean incrementToken() throws IOException {
  clearAttributes();
  int posIncr = 1;
```

```
  while(true) {
    //用词法分析器得到下一个 Token 以及 Token 的类型
    int tokenType = scanner.getNextToken();
    if (tokenType == StandardTokenizerImpl.YYEOF) {
      return false;
    }
    if (scanner.yylength() <= maxTokenLength) {
      posIncrAtt.setPositionIncrement(posIncr);
      //得到 Token 文本
      scanner.getText(termAtt);
      final int start = scanner.yychar();
      offsetAtt.setOffset(correctOffset(start), correctOffset(start+termAtt.termLength()));
      //设置类型
      typeAtt.setType(StandardTokenizerImpl.TOKEN_TYPES[tokenType]);
      return true;
    } else
      posIncr++;
  }
}
```

7.3、StandardFilter

其 incrementToken 函数如下：

```
public final boolean incrementToken() throws java.io.IOException {
  if (!input.incrementToken()) {
    return false;
  }
```

```
  char[] buffer = termAtt.termBuffer();
  final int bufferLength = termAtt.termLength();
  final String type = typeAtt.type();
  //如果是省略符号，如 He's，则去掉's
  if (type == APOSTROPHE_TYPE && bufferLength >= 2 &&
      buffer[bufferLength-2] == '\'' && (buffer[bufferLength-1] == 's' || buffer[bufferLength-1] ==
'S')) {
    termAtt.setTermLength(bufferLength - 2);
  } else if (type == ACRONYM_TYPE) {
    //如果是缩略语 I.B.M.，则去掉.
    int upto = 0;
    for(int i=0;i<bufferLength;i++) {
      char c = buffer[i];
      if (c != '.')
        buffer[upto++] = c;
    }
    termAtt.setTermLength(upto);
  }
  return true;
}
```

7.4、StandardAnalyzer

```
public TokenStream tokenStream(String fieldName, Reader reader) {
    //用词法分析器分词
    StandardTokenizer tokenStream = new StandardTokenizer(matchVersion, reader);
    tokenStream.setMaxTokenLength(maxTokenLength);
```

```
    //用标准过滤器过滤
    TokenStream result = new StandardFilter(tokenStream);
    //转换为小写
    result = new LowerCaseFilter(result);
    //去停词
    result = new StopFilter(enableStopPositionIncrements, result, stopSet);
    return result;
}
```

举例如下：

```
String s = "He's Juexian, His email is forfuture1978@gmail.com. He's an ip address 192.168.0.1,
AT&T and I.B.M. are all great companies.";
StringReader reader = new StringReader(s);
StandardAnalyzer analyzer = new StandardAnalyzer(Version.LUCENE_CURRENT);
TokenStream ts = analyzer.tokenStream("field", reader);
boolean hasnext = ts.incrementToken();
while(hasnext){
  TermAttribute ta = ts.getAttribute(TermAttribute.class);
  System.out.println(ta.term());
  hasnext = ts.incrementToken();
}
```

结果为：

```
he
juexian
his
email
forfuture1978@gmail.com
he
ip
```

```
address
192.168.0.1
at&t
ibm
all
great
companies
```

8、不同的域使用不同的分词器

8.1、PerFieldAnalyzerWrapper

有时候，我们想不同的域使用不同的分词器，则可以用 PerFieldAnalyzerWrapper 进行封装。

其有两个成员函数：

- Analyzer defaultAnalyzer：即当域没有指定分词器的时候使用此分词器
- Map<String,Analyzer> analyzerMap = new HashMap<String,Analyzer>()：一个从域名到分词器的映射，将根据域名使用相应的分词器。

其 TokenStream 函数如下：

```
public TokenStream tokenStream(String fieldName, Reader reader) {
  Analyzer analyzer = analyzerMap.get(fieldName);
  if (analyzer == null) {
    analyzer = defaultAnalyzer;
  }
  return analyzer.tokenStream(fieldName, reader);
}
```

举例说明：

```
String s = "Hello World";
```

```
PerFieldAnalyzerWrapper analyzer = new PerFieldAnalyzerWrapper(new SimpleAnalyzer());
analyzer.addAnalyzer("f1", new KeywordAnalyzer());
analyzer.addAnalyzer("f2", new WhitespaceAnalyzer());
TokenStream ts = analyzer.reusableTokenStream("f1", new StringReader(s));
boolean hasnext = ts.incrementToken();
while(hasnext){
   TermAttribute ta = ts.getAttribute(TermAttribute.class);
   System.out.println(ta.term());
   hasnext = ts.incrementToken();
}
System.out.println("----------------------------------------");
ts = analyzer.reusableTokenStream("f2", new StringReader(s));
hasnext = ts.incrementToken();
while(hasnext){
   TermAttribute ta = ts.getAttribute(TermAttribute.class);
   System.out.println(ta.term());
   hasnext = ts.incrementToken();
}
System.out.println("----------------------------------------");
ts = analyzer.reusableTokenStream("none", new StringReader(s));
hasnext = ts.incrementToken();
while(hasnext){
   TermAttribute ta = ts.getAttribute(TermAttribute.class);
   System.out.println(ta.term());
   hasnext = ts.incrementToken();
}
```

结果为：

```
Hello World
```

```
-------------------------------------------

Hello

World

-------------------------------------------

hello

world
```

第三篇：问题篇

问题一：为什么能搜的到“中华 AND 共和国”却搜不到“中华共和国”？

使用中科院的中文分词对“中华人民共和国” 进行索引，它被分词为"中华", "人民", "共和国"，用“人民共和国”进行搜索，可以搜到，而搜索"中华共和国"却搜索不到，用“中华 AND 共和国”却可以搜出来，为什么？

下载 http://ictclas.org/Download.html 中科院的词做了简单的分析，如果索引的时候“中华人民共和国”被分成了“中华”“人民”“共和国”，而搜索的时候，搜“中华共和国”，则被分为了“中华 共和国”，然而构建 Query Parser 构建 Query Object 的时候，却将它构建成了 PhraseQuery—— contents:"中华 共和国" ，而非 BooleanQuery——contents:中华 contents:共和国，根据 PhraseQuery 的解释，它有一个参数 slop 来表示两个词之间的距离，默认为 0，也即只有在文档不但包含“中华”而且包含“共和国”并且二者相邻的时候才能返回。这就是为什么“人民共和国”可以搜出来(它构建的是 PhraseQuery，但是相邻)，“中华 AND 共和国”能搜索出来(它构建的是 BooleanQuery)，而“中华共和国”搜不出来的原因(它构建的是 PhraseQuery，但不相邻)。

尝试解析 Query query = parser.parse("\"中华共和国\"~1")

或者用 API 设置 Slop 为 1，就能搜索出结果了。

```
Query query = parser.parse("中华共和国");
PhraseQuery pquery = (PhraseQuery)query;
pquery.setSlop(1);
```

例如：

```
Analyzer ca = new ChineseAnalyzer();
QueryParser parser = new QueryParser(field, ca);
Query query1 = parser.parse("人民共和国");
System.out.println("Searching for: " + query1.toString(field));
```

查询对象为：

```
query1    PhraseQuery   (id=39)
   boost    1.0
   field    "contents"
   maxPosition    1
   positions    ArrayList   (id=45)
   slop    0
   terms    ArrayList   (id=49)
      elementData    Object[4]   (id=74)
         [0]    Term   (id=76)
            field    "contents"
            text    "人民"
         [1]    Term   (id=77)
            field    "contents"
            text    "共和国"
```

相当于查询语句：

```
Searching for: "人民 共和国"
```

```
Query query2 = parser.parse("中华 AND 共和国");
System.out.println("Searching for: " + query2.toString(field));
```

查询对象为：

```
query2    BooleanQuery   (id=43)
   boost    1.0
   clauses    ArrayList   (id=56)
      elementData    Object[10]   (id=57)
         [0]    BooleanClause   (id=59)
            occur    BooleanClause$Occur   (id=62)
               name    "MUST"
            query    TermQuery   (id=65)
               boost    1.0
```

```
            term    Term  (id=70)
              field    "contents"
              text    "中华"
        [1]    BooleanClause  (id=61)
          occur    BooleanClause$Occur  (id=62)
            name    "MUST"
          query    TermQuery  (id=64)
            boost    1.0
            term    Term  (id=68)
              field    "contents"
              text    "共和国"
```

相当于查询语句：

```
Searching for: +中华 +共和国
```

```
Query query3 = parser.parse("\"中华共和国\"~1");
System.out.println("Searching for: " + query3.toString(field));
```

查询对象为：

```
query3    PhraseQuery  (id=54)
   boost    1.0
   field    "contents"
   maxPosition    1
   positions    ArrayList  (id=93)
   slop    1
   terms    ArrayList  (id=94)
     elementData    Object[4]  (id=96)
       [0]    Term  (id=97)
         field    "contents"
         text    "中华"
       [1]    Term  (id=98)
```

```
            field     "contents"
            text     "共和国"
```

相当于查询语句：

```
Searching for: "中华 共和国"~1
```

```
Query query4 = parser.parse("中华共和国");
PhraseQuery pquery = (PhraseQuery)query4;
pquery.setSlop(1);
System.out.println("Searching for: " + query4.toString(field));
```

查询对象为：

```
query4     PhraseQuery   (id=55)
    boost     1.0
    field     "contents"
    maxPosition     1
    positions     ArrayList   (id=102)
    slop     1
    terms     ArrayList   (id=103)
       elementData     Object[4]   (id=105)
          [0]     Term   (id=107)
             field     "contents"
             text     "中华"
          [1]     Term   (id=108)
             field     "contents"
             text     "共和国"
```

相当于查询语句：

```
Searching for: "中华 共和国"~1
```

问题二：stemming 和 lemmatization 的关系

StandardAnalyzer 并不能进行 stemming 和 lemmatization，因而不能够区分单复数和词型。

文章中讲述的是全文检索的基本原理，理解了他，有利于更好的理解 Lucene，但不代表 Lucene 是完全按照此基本流程进行的。

（1）有关 stemming

作为 stemming，一个著名的算法是 The Porter Stemming Algorithm，其主页为 http://tartarus.org/~martin/PorterStemmer/，也可查看其论文 http://tartarus.org/~martin/PorterStemmer/def.txt。

通过以下网页可以进行简单的测试：**Porter's Stemming Algorithm Online**[http://facweb.cs.depaul.edu/mobasher/classes/csc575/porter.html]

cars –> car

driving –> drive

tokenization –> token

然而

drove –> drove

可见 stemming 是通过规则缩减为词根的，而不能识别词型的变化。

在最新的 Lucene 3.0 中，已经有了 PorterStemFilter 这个类来实现上述算法，只可惜没有 Analyzer 向匹配，不过不要紧，我们可以简单实现：

```
public class PorterStemAnalyzer extends Analyzer
{
    @Override
    public TokenStream tokenStream(String fieldName, Reader reader) {
     return new PorterStemFilter(new LowerCaseTokenizer(reader));
    }
}
```

把此分词器用在你的程序中，就能够识别单复数和规则的词型变化了。

```
public void createIndex() throws IOException {
  Directory d = new SimpleFSDirectory(new File("d:/falconTest/lucene3/norms"));
    IndexWriter writer = new IndexWriter(d, new PorterStemAnalyzer(), true,
IndexWriter.MaxFieldLength.UNLIMITED);
  Field field = new Field("desc", "", Field.Store.YES, Field.Index.ANALYZED);
  Document doc = new Document();
  field.setValue("Hello students was driving cars professionally");
  doc.add(field);
  writer.addDocument(doc);
  writer.optimize();
  writer.close();
}
public void search() throws IOException {
  Directory d = new SimpleFSDirectory(new File("d:/falconTest/lucene3/norms"));
  IndexReader reader = IndexReader.open(d);
  IndexSearcher searcher = new IndexSearcher(reader);
  TopDocs docs = searcher.search(new TermQuery(new Term("desc", "car")), 10);
  System.out.println(docs.totalHits);
  docs = searcher.search(new TermQuery(new Term("desc", "drive")), 10);
  System.out.println(docs.totalHits);
  docs = searcher.search(new TermQuery(new Term("desc", "profession")), 10);
  System.out.println(docs.totalHits);
}
```

(2) 有关 lemmatization

至于 lemmatization，一般是有字典的，方能够由"drove"对应到"drive".

在网上搜了一下，找到 European languages lemmatizer[http://lemmatizer.org/]，只不过是在 linux 下面 C++开发的，有兴趣可以试验一下。

首先按照网站的说明下载，编译，安装：

libMAFSA is the core of the lemmatizer. All other libraries depend on it. Download the last version from the following page, unpack it and compile:

```
# tar xzf libMAFSA-0.2.tar.gz
# cd libMAFSA-0.2/
# cmake .
# make
# sudo make install
```

After this you should install libturglem. You can download it at the same place.

```
# tar xzf libturglem-0.2.tar.gz
# cd libturglem-0.2
# cmake .
# make
# sudo make install
```

Next you should install english dictionaries with some additional features to work with.

```
# tar xzf turglem-english-0.2.tar.gz
# cd turglem-english-0.2
# cmake .
# make
# sudo make install
```

安装完毕后：

- /usr/local/include/turglem 是头文件，用于编译自己编写的代码
- /usr/local/share/turglem/english 是字典文件，其中 lemmas.xml 中我们可以看到"drove"和"drive"的对应，"was"和"be"的对应。
- /usr/local/lib 中的 libMAFSA.a libturglem.a libturglem-english.a libtxml.a 是用于生成应用程序的静态库

在 turglem-english-0.2 目录下有例子测试程序 test_utf8.cpp

```
int main(int argc, char **argv)
{
```

```
char in_s_buf[1024];
char *nl_ptr;
tl::lemmatizer lem;
if(argc != 4)
{
	printf("Usage: %s words.dic predict.dic flexias.bin\n", argv[0]);
	return -1;
}
lem.load_lemmatizer(argv[1], argv[3], argv[2]);
while (!feof(stdin))
{
	fgets(in_s_buf, 1024, stdin);
	nl_ptr = strchr(in_s_buf, '\n');
	if (nl_ptr) *nl_ptr = 0;
	nl_ptr = strchr(in_s_buf, '\r');
	if (nl_ptr) *nl_ptr = 0;
	if (in_s_buf[0])
	{
		printf("processing %s\n", in_s_buf);
		tl::lem_result pars;
		size_t pcnt = lem.lemmatize(in_s_buf, pars);
		printf("%d\n", pcnt);
		for (size_t i = 0; i < pcnt; i++)
		{
			std::string s;
			u_int32_t src_form = lem.get_src_form(pars, i);
			s = lem.get_text(pars, i, 0);
			printf("PARADIGM %d: normal form '%s'\n", (unsigned int)i, s.c_str());
```

```
                    printf("\tpart of speech:%d\n", lem.get_part_of_speech(pars, (unsigned int)i,
src_form));
                }
            }
        }
        return 0;
}
```

编译此文件，并且链接静态库：注意链接顺序，否则可能出错。

```
g++ -g -o output test_utf8.cpp -L/usr/local/lib/ -lturglem-english -lturglem -lMAFSA –ltxml
```

运行编译好的程序：

```
./output /usr/local/share/turglem/english/dict_english.auto
/usr/local/share/turglem/english/prediction_english.auto
/usr/local/share/turglem/english/paradigms_english.bin
```

做测试，虽然对其机制尚不甚了解，但是可以看到 lemmatization 的作用：

```
drove
processing drove
3
PARADIGM 0: normal form 'DROVE'
        part of speech:0
PARADIGM 1: normal form 'DROVE'
        part of speech:2
PARADIGM 2: normal form 'DRIVE'
        part of speech:2
was
processing was
3
PARADIGM 0: normal form 'BE'
```

```
      part of speech:3
PARADIGM 1: normal form 'BE'
      part of speech:3
PARADIGM 2: normal form 'BE'
      part of speech:3
```

问题三：影响 Lucene 对文档打分的四种方式

在索引阶段设置 Document Boost 和 Field Boost，存储在(.nrm)文件中。

如果希望某些文档和某些域比其他的域更重要，如果此文档和此域包含所要查询的词则应该得分较高，则可以在索引阶段设定文档的 boost 和域的 boost 值。

这些值是在索引阶段就写入索引文件的，存储在标准化因子(.nrm)文件中，一旦设定，除非删除此文档，否则无法改变。

如果不进行设定，则 Document Boost 和 Field Boost 默认为 1。

Document Boost 及 FieldBoost 的设定方式如下：

```
Document doc = new Document();
Field f = new Field("contents", "hello world", Field.Store.NO, Field.Index.ANALYZED);
f.setBoost(100);
doc.add(f);
doc.setBoost(100);
```

两者是如何影响 Lucene 的文档打分的呢？

让我们首先来看一下 Lucene 的文档打分的公式：

$$score(q,d) = coord(q,d) \cdot queryNorm(q) \cdot \sum_{t \text{ in } q} (tf(t \text{ in } d) \cdot idf(t)^2 \cdot t.getBoost() \cdot \mathbf{norm(t,d)})$$

Document Boost 和 Field Boost 影响的是 norm(t, d)，其公式如下：

$$norm(t,d) = \mathbf{doc.getBoost()} \cdot lengthNorm(field) \cdot \prod_{\text{field } \boldsymbol{f} \text{ in } d \text{ named as } \boldsymbol{t}} \mathbf{f.getBoost()}$$

它包括三个参数：

- Document boost：此值越大，说明此文档越重要。

- Field boost：此域越大，说明此域越重要。
- lengthNorm(field) = (1.0 / Math.sqrt(numTerms))：一个域中包含的 Term 总数越多，也即文档越长，此值越小，文档越短，此值越大。

其中第三个参数可以在自己的 Similarity 中影响打分，下面会论述。

当然，也可以在添加 Field 的时候，设置 Field.Index.ANALYZED_NO_NORMS 或 Field.Index.NOT_ANALYZED_NO_NORMS，完全不用 norm，来节约空间。

根据 Lucene 的注释，No norms means that index-time field and document boosting and field length normalization are disabled. The benefit is less memory usage as norms take up one byte of RAM per indexed field for every document in the index, during searching. Note that once you index a given field with norms enabled, disabling norms will have no effect. 没有 norms 意味着索引阶段禁用了文档 boost 和域的 boost 及长度标准化。好处在于节省内存，不用在搜索阶段为索引中的每篇文档的每个域都占用一个字节来保存 norms 信息了。但是对 norms 信息的禁用是必须全部域都禁用的，一旦有一个域不禁用，则其他禁用的域也会存放默认的 norms 值。因为为了加快 norms 的搜索速度，Lucene 是根据文档号乘以每篇文档的 norms 信息所占用的大小来计算偏移量的，中间少一篇文档，偏移量将无法计算。也即 norms 信息要么都保存，要么都不保存。

下面几个试验可以验证 norms 信息的作用：

试验一：Document Boost 的作用

```
public void testNormsDocBoost() throws Exception {
  File indexDir = new File("testNormsDocBoost");
      IndexWriter writer = new IndexWriter(FSDirectory.open(indexDir), new
StandardAnalyzer(Version.LUCENE_CURRENT), true, IndexWriter.MaxFieldLength.LIMITED);
  writer.setUseCompoundFile(false);
  Document doc1 = new Document();
  Field f1 = new Field("contents", "common hello hello", Field.Store.NO, Field.Index.ANALYZED);
  doc1.add(f1);
  doc1.setBoost(100);
  writer.addDocument(doc1);
```

```
  Document doc2 = new Document();
    Field f2 = new Field("contents", "common common hello", Field.Store.NO,
Field.Index.ANALYZED_NO_NORMS);
  doc2.add(f2);
  writer.addDocument(doc2);
  Document doc3 = new Document();
    Field f3 = new Field("contents", "common common common", Field.Store.NO,
Field.Index.ANALYZED_NO_NORMS);
  doc3.add(f3);
  writer.addDocument(doc3);
  writer.close();
  IndexReader reader = IndexReader.open(FSDirectory.open(indexDir));
  IndexSearcher searcher = new IndexSearcher(reader);
  TopDocs docs = searcher.search(new TermQuery(new Term("contents", "common")), 10);
  for (ScoreDoc doc : docs.scoreDocs) {
    System.out.println("docid : " + doc.doc + " score : " + doc.score);
  }
}
```

如果第一篇文档的域 f1 也为 Field.Index.ANALYZED_NO_NORMS 的时候，搜索排名如下：

```
docid : 2 score : 1.2337708
docid : 1 score : 1.0073696
docid : 0 score : 0.71231794
```

如果第一篇文档的域 f1 设为 Field.Index.ANALYZED，则搜索排名如下：

```
docid : 0 score : 39.889805
docid : 2 score : 0.6168854
docid : 1 score : 0.5036848
```

试验二：Field Boost 的作用

如果我们觉得 title 要比 contents 要重要，可以做一下设定。

```
public void testNormsFieldBoost() throws Exception {
  File indexDir = new File("testNormsFieldBoost");
      IndexWriter writer = new IndexWriter(FSDirectory.open(indexDir), new
StandardAnalyzer(Version.LUCENE_CURRENT), true, IndexWriter.MaxFieldLength.LIMITED);
  writer.setUseCompoundFile(false);
  Document doc1 = new Document();
  Field f1 = new Field("title", "common hello hello", Field.Store.NO, Field.Index.ANALYZED);
  f1.setBoost(100);
  doc1.add(f1);
  writer.addDocument(doc1);
  Document doc2 = new Document();
    Field f2 = new Field("contents", "common common hello", Field.Store.NO,
Field.Index.ANALYZED_NO_NORMS);
  doc2.add(f2);
  writer.addDocument(doc2);
  writer.close();
  IndexReader reader = IndexReader.open(FSDirectory.open(indexDir));
  IndexSearcher searcher = new IndexSearcher(reader);
   QueryParser parser = new QueryParser(Version.LUCENE_CURRENT, "contents", new
StandardAnalyzer(Version.LUCENE_CURRENT));
  Query query = parser.parse("title:common contents:common");
  TopDocs docs = searcher.search(query, 10);
  for (ScoreDoc doc : docs.scoreDocs) {
   System.out.println("docid : " + doc.doc + " score : " + doc.score);
  }
}
```

如果第一篇文档的域 f1 也为 Field.Index.ANALYZED_NO_NORMS 的时候，搜索排名如下：

```
docid : 1 score : 0.49999997
```

```
docid : 0 score : 0.35355338
```

如果第一篇文档的域 f1 设为 Field.Index.ANALYZED，则搜索排名如下：

```
docid : 0 score : 19.79899
docid : 1 score : 0.49999997
```

试验三：norms 中文档长度对打分的影响

```
public void testNormsLength() throws Exception {
  File indexDir = new File("testNormsLength");
      IndexWriter writer = new IndexWriter(FSDirectory.open(indexDir), new
StandardAnalyzer(Version.LUCENE_CURRENT), true, IndexWriter.MaxFieldLength.LIMITED);
  writer.setUseCompoundFile(false);
  Document doc1 = new Document();
    Field f1 = new Field("contents", "common hello hello", Field.Store.NO,
Field.Index.ANALYZED_NO_NORMS);
  doc1.add(f1);
  writer.addDocument(doc1);
  Document doc2 = new Document();
  Field f2 = new Field("contents", "common common hello hello hello hello", Field.Store.NO,
Field.Index.ANALYZED_NO_NORMS);
  doc2.add(f2);
  writer.addDocument(doc2);
  writer.close();
  IndexReader reader = IndexReader.open(FSDirectory.open(indexDir));
  IndexSearcher searcher = new IndexSearcher(reader);
   QueryParser parser = new QueryParser(Version.LUCENE_CURRENT, "contents", new
StandardAnalyzer(Version.LUCENE_CURRENT));
  Query query = parser.parse("title:common contents:common");
  TopDocs docs = searcher.search(query, 10);
  for (ScoreDoc doc : docs.scoreDocs) {
```

```
    System.out.println("docid : " + doc.doc + " score : " + doc.score);
  }
}
```

当 norms 被禁用的时候，包含两个 common 的第二篇文档打分较高：

```
docid : 1 score : 0.13928263
docid : 0 score : 0.09848769
```

当 norms 起作用的时候，虽然包含两个 common 的第二篇文档，由于长度较长，因而打分较低：

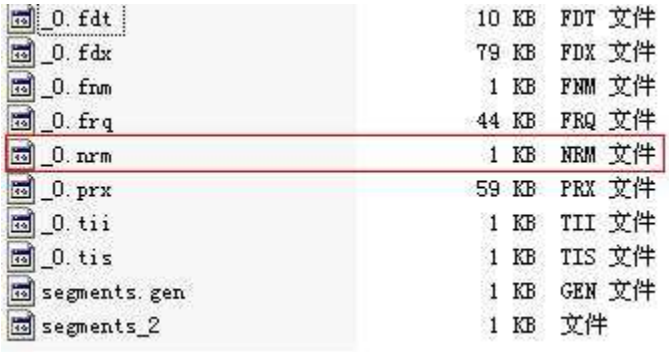
```
docid : 0 score : 0.09848769
docid : 1 score : 0.052230984
```

试验四：norms 信息要么都保存，要么都不保存的特性

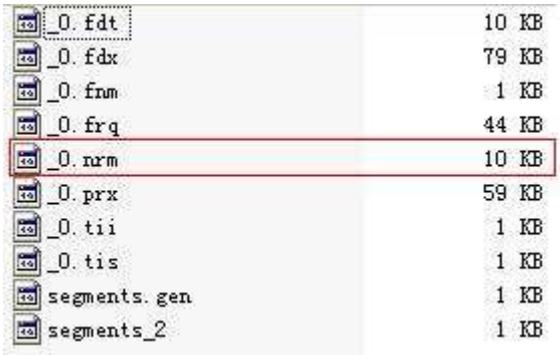
```
public void testOmitNorms() throws Exception {
  File indexDir = new File("testOmitNorms");
      IndexWriter writer = new IndexWriter(FSDirectory.open(indexDir), new StandardAnalyzer(Version.LUCENE_CURRENT), true, IndexWriter.MaxFieldLength.LIMITED);
  writer.setUseCompoundFile(false);
  Document doc1 = new Document();
  Field f1 = new Field("title", "common hello hello", Field.Store.NO, Field.Index.ANALYZED);
  doc1.add(f1);
  writer.addDocument(doc1);
  for (int i = 0; i < 10000; i++) {
    Document doc2 = new Document();
    Field f2 = new Field("contents", "common common hello hello hello hello", Field.Store.NO, Field.Index.ANALYZED_NO_NORMS);
    doc2.add(f2);
    writer.addDocument(doc2);
  }
```

```
  writer.close();
}
```

当我们添加 10001 篇文档，所有的文档都设为 Field.Index.ANALYZED_NO_NORMS 的时候，我们看索引文件，发现.nrm 文件只有 1K，也即其中除了保持一定的格式信息，并无其他数据。



当我们把第一篇文档设为 Field.Index.ANALYZED，而其他 10000 篇文档都设为 Field.Index.ANALYZED_NO_NORMS 的时候，发现.nrm 文件又 10K，也即所有的文档都存储了 norms 信息，而非只有第一篇文档。



在搜索语句中，设置 Query Boost.

在搜索中，我们可以指定，某些词对我们来说更重要，我们可以设置这个词的 boost：

```
common^4 hello
```

使得包含 common 的文档比包含 hello 的文档获得更高的分数。

由于在 Lucene 中，一个 Term 定义为 Field:Term，则也可以影响不同域的打分：

```
title:common^4 content:common
```

使得 title 中包含 common 的文档比 content 中包含 common 的文档获得更高的分数。

实例：

```
public void testQueryBoost() throws Exception {
  File indexDir = new File("TestQueryBoost");
      IndexWriter writer = new IndexWriter(FSDirectory.open(indexDir), new
StandardAnalyzer(Version.LUCENE_CURRENT), true, IndexWriter.MaxFieldLength.LIMITED);
  Document doc1 = new Document();
  Field f1 = new Field("contents", "common1 hello hello", Field.Store.NO, Field.Index.ANALYZED);
  doc1.add(f1);
  writer.addDocument(doc1);
  Document doc2 = new Document();
   Field f2 = new Field("contents", "common2 common2 hello", Field.Store.NO,
Field.Index.ANALYZED);
  doc2.add(f2);
  writer.addDocument(doc2);
  writer.close();
  IndexReader reader = IndexReader.open(FSDirectory.open(indexDir));
  IndexSearcher searcher = new IndexSearcher(reader);
   QueryParser parser = new QueryParser(Version.LUCENE_CURRENT, "contents", new
StandardAnalyzer(Version.LUCENE_CURRENT));
  Query query = parser.parse("common1 common2");
  TopDocs docs = searcher.search(query, 10);
  for (ScoreDoc doc : docs.scoreDocs) {
   System.out.println("docid : " + doc.doc + " score : " + doc.score);
  }
}
```

根据 tf/idf，包含两个 common2 的第二篇文档打分较高：

```
docid : 1 score : 0.24999999

docid : 0 score : 0.17677669
```

如果我们输入的查询语句为："common1^100 common2"，则第一篇文档打分较高：

```
docid : 0 score : 0.2499875

docid : 1 score : 0.0035353568
```

那 Query Boost 是如何影响文档打分的呢？

根据 Lucene 的打分计算公式：

$$score(q,d) = coord(q,d) \cdot queryNorm(q) \cdot \sum_{t \text{ in } q} (tf(t \text{ in } d) \cdot idf(t)^2 \cdot \mathbf{t.getBoost()} \cdot norm(t,d))$$

注：在 queryNorm 的部分，也有 q.getBoost()的部分，但是对 query 向量的归一化(见向量空间模型与 Lucene 的打分机制[http://forfuture1978.javaeye.com/blog/588721])。

继承并实现自己的 Similarity

Similariy 是计算 Lucene 打分的最主要的类，实现其中的很多借口可以干预打分的过程。

(1) float computeNorm(String field, FieldInvertState state)

(2) float lengthNorm(String fieldName, int numTokens)

(3) float queryNorm(float sumOfSquaredWeights)

(4) float tf(float freq)

(5) float idf(int docFreq, int numDocs)

(6) float coord(int overlap, int maxOverlap)

(7) float scorePayload(int docId, String fieldName, int start, int end, byte [] payload, int offset, int length)

它们分别影响 Lucene 打分计算的如下部分：

$$score(q,d) = (6)coord(q,d) \cdot (3)queryNorm(q) \cdot \sum_{t \text{ in } q} ((4)tf(t \text{ in } d) \cdot (5)idf(t)^2 \cdot t.getBoost() \cdot (1)norm(t,d))$$

$$norm(t,d) = doc.getBoost() \cdot (2)lengthNorm(field) \cdot \prod_{\text{field } f \text{ in } d \text{ named as } t} f.getBoost()$$

下面逐个进行解释：

(1) float computeNorm(String field, FieldInvertState state)

影响标准化因子的计算，如上述，他主要包含了三部分：文档 boost，域 boost，以及文档长度归一化。此函数一般按照上面 norm(t, d)的公式进行计算。

(2) float lengthNorm(String fieldName, int numTokens)

主要计算文档长度的归一化，默认是 1.0 / Math.sqrt(numTerms)。

因为在索引中，不同的文档长度不一样，很显然，对于任意一个 term，在长的文档中的 tf 要大的多，因而分数也越高，这样对小的文档不公平，举一个极端的例子，在一篇 1000 万个词的鸿篇巨著中，"lucene"这个词出现了 11 次，而在一篇 12 个词的短小文档中，"lucene"这个词出现了 10 次，如果不考虑长度在内，当然鸿篇巨著应该分数更高，然而显然这篇小文档才是真正关注"lucene"的。

因而在此处是要除以文档的长度，从而减少因文档长度带来的打分不公。

然而现在这个公式是偏向于首先返回短小的文档的，这样在实际应用中使得搜索结果也很难看。

于是在实践中，要根据项目的需要，根据搜索的领域，改写 lengthNorm 的计算公式。比如我想做一个经济学论文的搜索系统，经过一定时间的调研，发现大多数的经济学论文的长度在 8000 到 10000 词，因而 lengthNorm 的公式应该是一个倒抛物线型的，8000 到 10000 词的论文分数最高，更短或更长的分数都应该偏低，方能够返回给用户最好的数据。

(3) float queryNorm(float sumOfSquaredWeights)

这是按照向量空间模型，对 query 向量的归一化。此值并不影响排序，而仅仅使得不同的 query 之间的分数可以比较。

(4) float tf(float freq)

freq 是指在一篇文档中包含的某个词的数目。tf 是根据此数目给出的分数，默认为 Math.sqrt(freq)。也即此项并不是随着包含的数目的增多而线性增加的。

(5) float idf(int docFreq, int numDocs)

idf 是根据包含某个词的文档数以及总文档数计算出的分数，默认为 (Math.log(numDocs/(double)(docFreq+1)) + 1.0)。

由于此项计算涉及到总文档数和包含此词的文档数，因而需要全局的文档数信息，这给跨索引搜索造成麻烦。

从下面的例子我们可以看出，用 MultiSearcher 来一起搜索两个索引和分别用 IndexSearcher 来搜索两个索引所得出的分数是有很大差异的。

究其原因是 MultiSearcher 的 docFreq(Term term)函数计算了包含两个索引中包含此词的总文档数，而 IndexSearcher 仅仅计算了每个索引中包含此词的文档数。当两个索引包含的文档总数是有很大不同的时候，分数是无法比较的。

```
public void testMultiIndex() throws Exception{
  MultiIndexSimilarity sim = new MultiIndexSimilarity();
  File indexDir01 = new File("TestMultiIndex/TestMultiIndex01");
  File indexDir02 = new File("TestMultiIndex/TestMultiIndex02");
  IndexReader reader01 = IndexReader.open(FSDirectory.open(indexDir01));
  IndexReader reader02 = IndexReader.open(FSDirectory.open(indexDir02));
  IndexSearcher searcher01 = new IndexSearcher(reader01);
  searcher01.setSimilarity(sim);
  IndexSearcher searcher02 = new IndexSearcher(reader02);
  searcher02.setSimilarity(sim);
  MultiSearcher multiseacher = new MultiSearcher(searcher01, searcher02);
  multiseacher.setSimilarity(sim);
   QueryParser parser = new QueryParser(Version.LUCENE_CURRENT, "contents", new
StandardAnalyzer(Version.LUCENE_CURRENT));
  Query query = parser.parse("common");
  TopDocs docs = searcher01.search(query, 10);
  System.out.println("----------------------------------------------");
  for (ScoreDoc doc : docs.scoreDocs) {
   System.out.println("docid : " + doc.doc + " score : " + doc.score);
  }
  System.out.println("----------------------------------------------");
  docs = searcher02.search(query, 10);
  for (ScoreDoc doc : docs.scoreDocs) {
```

```
    System.out.println("docid : " + doc.doc + " score : " + doc.score);
  }
  System.out.println("----------------------------------------------");
  docs = multiseacher.search(query, 20);
  for (ScoreDoc doc : docs.scoreDocs) {
    System.out.println("docid : " + doc.doc + " score : " + doc.score);
  }
```

结果为：

```
------------------------------
docid : 0 score : 0.49317428
docid : 1 score : 0.49317428
docid : 2 score : 0.49317428
docid : 3 score : 0.49317428
docid : 4 score : 0.49317428
docid : 5 score : 0.49317428
docid : 6 score : 0.49317428
docid : 7 score : 0.49317428
------------------------------
docid : 0 score : 0.45709616
docid : 1 score : 0.45709616
docid : 2 score : 0.45709616
docid : 3 score : 0.45709616
docid : 4 score : 0.45709616
------------------------------
docid : 0 score : 0.5175894
docid : 1 score : 0.5175894
docid : 2 score : 0.5175894
docid : 3 score : 0.5175894
```

```
docid : 4 score : 0.5175894
docid : 5 score : 0.5175894
docid : 6 score : 0.5175894
docid : 7 score : 0.5175894
docid : 8 score : 0.5175894
docid : 9 score : 0.5175894
docid : 10 score : 0.5175894
docid : 11 score : 0.5175894
docid : 12 score : 0.5175894
```

如果几个索引都是在一台机器上，则用 MultiSearcher 或者 MultiReader 就解决问题了，然而有时候索引是分布在多台机器上的，虽然 Lucene 也提供了 RMI，或用 NFS 保存索引的方法，然而效率和并行性一直是一个问题。

一个可以尝试的办法是在 Similarity 中，idf 返回 1，然后多个机器上的索引并行搜索，在汇总结果的机器上，再融入 idf 的计算。

如下面的例子可以看出，当 idf 返回 1 的时候，打分可以比较了：

```
class MultiIndexSimilarity extends Similarity {
  @Override
  public float idf(int docFreq, int numDocs) {
   return 1.0f;
  }
```

```
----------------------------
docid : 0 score : 0.559017
docid : 1 score : 0.559017
docid : 2 score : 0.559017
docid : 3 score : 0.559017
docid : 4 score : 0.559017
docid : 5 score : 0.559017
docid : 6 score : 0.559017
```

```
docid : 7 score : 0.559017
----------------------------
docid : 0 score : 0.559017
docid : 1 score : 0.559017
docid : 2 score : 0.559017
docid : 3 score : 0.559017
docid : 4 score : 0.559017
----------------------------
docid : 0 score : 0.559017
docid : 1 score : 0.559017
docid : 2 score : 0.559017
docid : 3 score : 0.559017
docid : 4 score : 0.559017
docid : 5 score : 0.559017
docid : 6 score : 0.559017
docid : 7 score : 0.559017
docid : 8 score : 0.559017
docid : 9 score : 0.559017
docid : 10 score : 0.559017
docid : 11 score : 0.559017
docid : 12 score : 0.559017
```

(6) float coord(int overlap, int maxOverlap)

一次搜索可能包含多个搜索词，而一篇文档中也可能包含多个搜索词，此项表示，当一篇文档中包含的搜索词越多，则此文档则打分越高。

```
public void TestCoord() throws Exception {
  MySimilarity sim = new MySimilarity();
  File indexDir = new File("TestCoord");
      IndexWriter writer = new IndexWriter(FSDirectory.open(indexDir), new
```

```
StandardAnalyzer(Version.LUCENE_CURRENT), true, IndexWriter.MaxFieldLength.LIMITED);
  Document doc1 = new Document();
  Field f1 = new Field("contents", "common hello world", Field.Store.NO, Field.Index.ANALYZED);
  doc1.add(f1);
  writer.addDocument(doc1);
  Document doc2 = new Document();
   Field f2 = new Field("contents", "common common common", Field.Store.NO,
Field.Index.ANALYZED);
  doc2.add(f2);
  writer.addDocument(doc2);
  for(int i = 0; i < 10; i++){
   Document doc3 = new Document();
   Field f3 = new Field("contents", "world", Field.Store.NO, Field.Index.ANALYZED);
   doc3.add(f3);
   writer.addDocument(doc3);
  }
  writer.close();
  IndexReader reader = IndexReader.open(FSDirectory.open(indexDir));
  IndexSearcher searcher = new IndexSearcher(reader);
  searcher.setSimilarity(sim);
   QueryParser parser = new QueryParser(Version.LUCENE_CURRENT, "contents", new
StandardAnalyzer(Version.LUCENE_CURRENT));
  Query query = parser.parse("common world");
  TopDocs docs = searcher.search(query, 2);
  for (ScoreDoc doc : docs.scoreDocs) {
   System.out.println("docid : " + doc.doc + " score : " + doc.score);
  }
}
```

```
class MySimilarity extends Similarity {
  @Override
  public float coord(int overlap, int maxOverlap) {
   return 1;
  }
}
```

如上面的实例，当 coord 返回 1，不起作用的时候，文档一虽然包含了两个搜索词 common 和 world，但由于 world 的所在的文档数太多，而文档二包含 common 的次数比较多，因而文档二分数较高：

```
docid : 1 score : 1.9059997
docid : 0 score : 1.2936771
```

而当 coord 起作用的时候，文档一由于包含了两个搜索词而分数较高：

```
class MySimilarity extends Similarity {
  @Override
  public float coord(int overlap, int maxOverlap) {
   return overlap / (float)maxOverlap;
  }
}
```

```
docid : 0 score : 1.2936771
docid : 1 score : 0.95299983
```

(7) float scorePayload(int docId, String fieldName, int start, int end, byte [] payload, int offset, int length)

由于 Lucene 引入了 payload，因而可以存储一些自己的信息，用户可以根据自己存储的信息，来影响 Lucene 的打分。

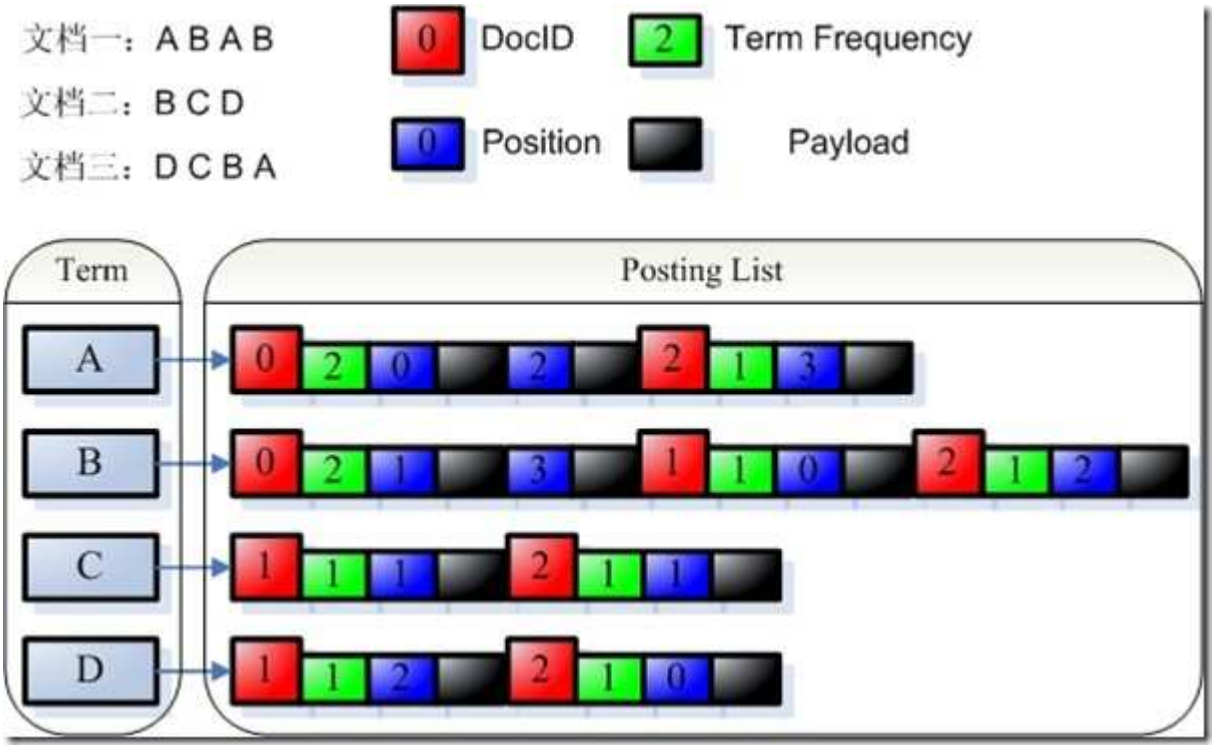
payload 的定义

我们知道，索引是以倒排表形式存储的，对于每一个词，都保存了包含这个词的一个链表，当然为了加快查询速度，此链表多用跳跃表进行存储。

Payload 信息就是存储在倒排表中的，同文档号一起存放，多用于存储与每篇文档相关的一

些信息。当然这部分信息也可以存储域里(stored Field)，两者从功能上基本是一样的，然而当要存储的信息很多的时候，存放在倒排表里，利用跳跃表，有利于大大提高搜索速度。

Payload 的存储方式如下图：



由 payload 的定义，我们可以看出，payload 可以存储一些不但与文档相关，而且与查询词也相关的信息。比如某篇文档的某个词有特殊性，则可以在这个词的这个文档的 position 信息后存储 payload 信息，使得当搜索这个词的时候，这篇文档获得较高的分数。

要利用 payload 来影响查询需要做到以下几点，下面举例用标记的词在 payload 中存储 1，否则存储 0：

首先要实现自己的 Analyzer 从而在 Token 中放入 payload 信息：

```
class BoldAnalyzer extends Analyzer {
  @Override
  public TokenStream tokenStream(String fieldName, Reader reader) {
   TokenStream result = new WhitespaceTokenizer(reader);
   result = new BoldFilter(result);
   return result;
```

```
  }
}
class BoldFilter extends TokenFilter {
  public static int IS_NOT_BOLD = 0;
  public static int IS_BOLD = 1;
  private TermAttribute termAtt;
  private PayloadAttribute payloadAtt;
  protected BoldFilter(TokenStream input) {
    super(input);
    termAtt = addAttribute(TermAttribute.class);
    payloadAtt = addAttribute(PayloadAttribute.class);
  }
  @Override
  public boolean incrementToken() throws IOException {
    if (input.incrementToken()) {
      final char[] buffer = termAtt.termBuffer();
      final int length = termAtt.termLength();
      String tokenstring = new String(buffer, 0, length);
      if (tokenstring.startsWith("") && tokenstring.endsWith("")) {
        tokenstring = tokenstring.replace("", "");
          tokenstring = tokenstring.replace("", "");
        termAtt.setTermBuffer(tokenstring);
        payloadAtt.setPayload(new Payload(int2bytes(IS_BOLD)));
      } else {
        payloadAtt.setPayload(new Payload(int2bytes(IS_NOT_BOLD)));
      }
      return true;
    } else
```

```
    return false;
  }
  public static int bytes2int(byte[] b) {
    int mask = 0xff;
    int temp = 0;
    int res = 0;
    for (int i = 0; i < 4; i++) {
      res <<= 8;
      temp = b[i] & mask;
      res |= temp;
    }
    return res;
  }
  public static byte[] int2bytes(int num) {
    byte[] b = new byte[4];
    for (int i = 0; i < 4; i++) {
      b[i] = (byte) (num >>> (24 - i * 8));
    }
    return b;
  }
}
```

然后，实现自己的 Similarity，从 payload 中读出信息，根据信息来打分。

```
class PayloadSimilarity extends DefaultSimilarity {
  @Override
  public float scorePayload(int docId, String fieldName, int start, int end, byte[] payload, int offset,
int length) {
    int isbold = BoldFilter.bytes2int(payload);
    if(isbold == BoldFilter.IS_BOLD){
```

```
      System.out.println("It is a bold char.");
    } else {
      System.out.println("It is not a bold char.");
    }
    return 1;
  }
}
```

最后，查询的时候，一定要用 PayloadXXXQuery(在此用 PayloadTermQuery，在 Lucene 2.4.1 中，用 BoostingTermQuery)，否则 scorePayload 不起作用。

```
public void testPayloadScore() throws Exception {
  PayloadSimilarity sim = new PayloadSimilarity();
  File indexDir = new File("TestPayloadScore");
  IndexWriter writer = new IndexWriter(FSDirectory.open(indexDir), new BoldAnalyzer(), true,
IndexWriter.MaxFieldLength.LIMITED);
  Document doc1 = new Document();
  Field f1 = new Field("contents", "common hello world", Field.Store.NO, Field.Index.ANALYZED);
  doc1.add(f1);
  writer.addDocument(doc1);
  Document doc2 = new Document();
  Field f2 = new Field("contents", "common hello world", Field.Store.NO, Field.Index.ANALYZED);
  doc2.add(f2);
  writer.addDocument(doc2);
  writer.close();
  IndexReader reader = IndexReader.open(FSDirectory.open(indexDir));
  IndexSearcher searcher = new IndexSearcher(reader);
  searcher.setSimilarity(sim);
  PayloadTermQuery query = new PayloadTermQuery(new Term("contents",
"hello"), new MaxPayloadFunction());
```

```
  TopDocs docs = searcher.search(query, 10);
  for (ScoreDoc doc : docs.scoreDocs) {
    System.out.println("docid : " + doc.doc + " score : " + doc.score);
  }
}
```

如果 scorePayload 函数始终是返回 1，则结果如下，不起作用。

```
It is not a bold char.
It is a bold char.
docid : 0 score : 0.2101998
docid : 1 score : 0.2101998
```

如果 scorePayload 函数如下：

```
class PayloadSimilarity extends DefaultSimilarity {
  @Override
  public float scorePayload(int docId, String fieldName, int start, int end, byte[] payload, int offset,
int length) {
    int isbold = BoldFilter.bytes2int(payload);
    if(isbold == BoldFilter.IS_BOLD){
      System.out.println("It is a bold char.");
      return 10;
    } else {
      System.out.println("It is not a bold char.");
      return 1;
    }
  }
}
```

则结果如下，同样是包含 hello，包含加粗的文档获得较高分：

```
It is not a bold char.
```

```
It is a bold char.
docid : 1 score : 2.101998
docid : 0 score : 0.2101998
```

继承并实现自己的 collector

以上各种方法，已经把 Lucene score 计算公式的所有变量都涉及了，如果这还不能满足您的要求，还可以继承实现自己的 collector。

在 Lucene 2.4 中，HitCollector 有个函数 public abstract void collect(int doc, float score)，用来收集搜索的结果。

其中 TopDocCollector 的实现如下：

```
public void collect(int doc, float score) {
  if (score > 0.0f) {
    totalHits++;
    if (reusableSD == null) {
      reusableSD = new ScoreDoc(doc, score);
    } else if (score >= reusableSD.score) {
      reusableSD.doc = doc;
      reusableSD.score = score;
    } else {
      return;
    }
    reusableSD = (ScoreDoc) hq.insertWithOverflow(reusableSD);
  }
}
```

此函数将 docid 和 score 插入一个 PriorityQueue 中，使得得分最高的文档先返回。

我们可以继承 HitCollector，并在此函数中对 score 进行修改，然后再插入 PriorityQueue，或者插入自己的数据结构。

比如我们在另外的地方存储 docid 和文档创建时间的对应，我们希望当文档时间是一天之内

的分数最高，一周之内的分数其次，一个月之外的分数很低。

我们可以这样修改：

```
public static long milisecondsOneDay = 24L * 3600L * 1000L;
public static long millisecondsOneWeek = 7L * 24L * 3600L * 1000L;
public static long millisecondsOneMonth = 30L * 24L * 3600L * 1000L;
public void collect(int doc, float score) {
  if (score > 0.0f) {
    long time = getTimeByDocId(doc);
    if(time < milisecondsOneDay) {
       score = score * 1.0;
    } else if (time < millisecondsOneWeek){
       score = score * 0.8;
    } else if (time < millisecondsOneMonth) {
       score = score * 0.3;
    } else {
       score = score * 0.1;
    }
    totalHits++;
    if (reusableSD == null) {
      reusableSD = new ScoreDoc(doc, score);
    } else if (score >= reusableSD.score) {
      reusableSD.doc = doc;
      reusableSD.score = score;
    } else {
      return;
    }
    reusableSD = (ScoreDoc) hq.insertWithOverflow(reusableSD);
  }
```

```
}
```

在 Lucene 3.0 中，Collector 接口为 void collect(int doc)，TopScoreDocCollector 实现如下：

```
public void collect(int doc) throws IOException {
  float score = scorer.score();
  totalHits++;
  if (score <= pqTop.score) {
    return;
  }
  pqTop.doc = doc + docBase;
  pqTop.score = score;
  pqTop = pq.updateTop();
}
```

同样可以用上面的方式影响其打分。

问题四：Lucene 中的 TooManyClause 异常

使用 Lucene 检索过程中如果用到 RangeQuery,PrefixQuery,WildcardQuery,FuzzyQuery 等 Query 的时候，可能会产生 TooManyClauses 异常。为什么呢？

以 RangeQuery 为例，如果日期范围为 19990101 到 20091231，在索引文件中有 19990102，19990103 等等这些日期词组，那么 RangeQuery 会被扩展成“19990102 OR 19990103”，成了 2 个子句。可以想象，如果索引文件里面在这个时间段内的日期有很多，那么就会产生很多子句。

PrefixQuery 等也是同样的道理，如查询词为“法*”，如果索引文件中有“法律”、“法场”、“法医”、“法典”等等，这个 Query 就会被扩展成“法律 OR 法场 OR 法医 OR 法典”，或许会更多更多。

而为了节省内存，Lucene 默认将子句数目限制为 1024，如果超出限制，就会抛出 TooManyClauses 异常。

怎么解决这个问题呢，Lucene 提供了三种方法：

(1) 使用 filter 替代 Query，当然这是以牺牲查询速度为代价的，不过可以通过缓存的方式缓解这个问题。仍然以前面 RangeQuery 为例，可以使用 RangeFilter 来替代 RangeQuery，如下：

之前的代码：

```
BooleanQuery simpleQuery = new BooleanQuery();
Term dateLower = new Term("publishDate", startYear + "0101");
Term dateUpper = new Term("publishDate", endYear + "1231");
RangeQuery dateQuery = new RangeQuery(dateLower, dateUpper, true);
simpleQuery.add(dateQuery, Occur.MUST);
```

之后的代码：

```
BooleanQuery simpleQuery = new BooleanQuery();
RangeFilter dateFilter = new RangeFilter("publishDate", startYear + "0101", endYear + "1231", true, true);
FilteredQuery filteredQuery = new FilteredQuery(simpleQuery, dateFilter);
```

(2) 通过 BooleanQuery.setMaxClauseCount(10240)来限制数目。这样会加大内存的消耗。使用 BooleanQuery.setMaxClauseCount(Integer.MAX_VALUE)，可以完全去掉这个限制。

(3) 对于范围查询，可以尽可能的降低精度，比如如果查询不需要精确到月份与日期，只需要精确到年，据说可以使用 DateTools 这个类可以很简单解决时间转化问题。

问题五：Lucene 的事务性

所谓事务性，本多指数据库的属性，包括 ACID 四个基本要素：原子性(Atomicity）、一致性（Consistency）、隔离性（Isolation）、持久性（Durability）。

我们这里主要讨论隔离性，Lucene 的 IndexReader 和 IndexWriter 具有隔离性。

- 当 IndexReader.open 打开一个索引的时候，相对于给当前索引进行了一次 snapshot，此后的任何修改都不会被看到。
- 仅当 IndexReader.open 打开一个索引后，才有可能看到从上次打开后对索引的修改。
- 当 IndexWriter 没有调用 Commit 的时候，其修改的内容是不能够被看到的，哪怕 IndexReader 被重新打开。
- 欲使最新的修改被看到，一方面 IndexWriter 需要 commit，一方面 IndexReader 重新打开。

下面我们举几个例子来说明上述隔离性：

```
(1) 首先做准备，索引十篇文档
File indexDir = new File("TestIsolation/index");
IndexWriter writer = new IndexWriter(FSDirectory.open(indexDir), new
StandardAnalyzer(Version.LUCENE_CURRENT), true, IndexWriter.MaxFieldLength.LIMITED);
for(int i =0; i < 10; i++){
  indexDocs(writer);
}
writer.close();
(2) 然后再索引十篇文档，并不 commit
writer = new IndexWriter(FSDirectory.open(indexDir), new
StandardAnalyzer(Version.LUCENE_CURRENT), IndexWriter.MaxFieldLength.LIMITED);
for(int i =0; i < 10; i++){
  indexDocs(writer);
}
```

(3) 打开一个 IndexReader，但是由于 IndexWriter 没有 commit，所以仍然仅看到十篇文档。

```
IndexReader reader = IndexReader.open(FSDirectory.open(indexDir));
IndexSearcher searcher = new IndexSearcher(reader);
TopDocs docs = searcher.search(new TermQuery(new Term("contents","hello")), 50);
System.out.println(docs.totalHits);
```

(4) IndexWriter 进行提交 commit

```
writer.commit();
```

(5) 不重新打开 IndexReader，进行搜索，仍然仅看到十篇文档。

```
docs = searcher.search(new TermQuery(new Term("contents","hello")), 50);
System.out.println(docs.totalHits);
```

(6) IndexReader 重新打开，则可以看到二十篇文档。

```
reader = IndexReader.open(FSDirectory.open(indexDir));
searcher = new IndexSearcher(reader);
docs = searcher.search(new TermQuery(new Term("contents","hello")), 50);
System.out.println(docs.totalHits);
```

问题六：用 Lucene 构建实时的索引

由于前一章所述的 Lucene 的事务性，使得 Lucene 可以增量的添加一个段，我们知道，倒排索引是有一定的格式的，而这个格式一旦写入是非常难以改变的，那么如何能够增量建索引呢？Lucene 使用段这个概念解决了这个问题，对于每个已经生成的段，其倒排索引结构不会再改变，而增量添加的文档添加到新的段中，段之间在一定的时刻进行合并，从而形成新的倒排索引结构。

然而也正因为 Lucene 的事务性，使得 Lucene 的索引不够实时，如果想 Lucene 实时，则必须新添加的文档后 IndexWriter 需要 commit，在搜索的时候 IndexReader 需要重新的打开，然而当索引在硬盘上的时候，尤其是索引非常大的时候，IndexWriter 的 commit 操作和 IndexReader 的 open 操作都是非常慢的，根本达不到实时性的需要。

好在 Lucene 提供了 RAMDirectory，也即内存中的索引，能够很快的 commit 和 open，然而又存在如果索引很大，内存中不能够放下的问题。

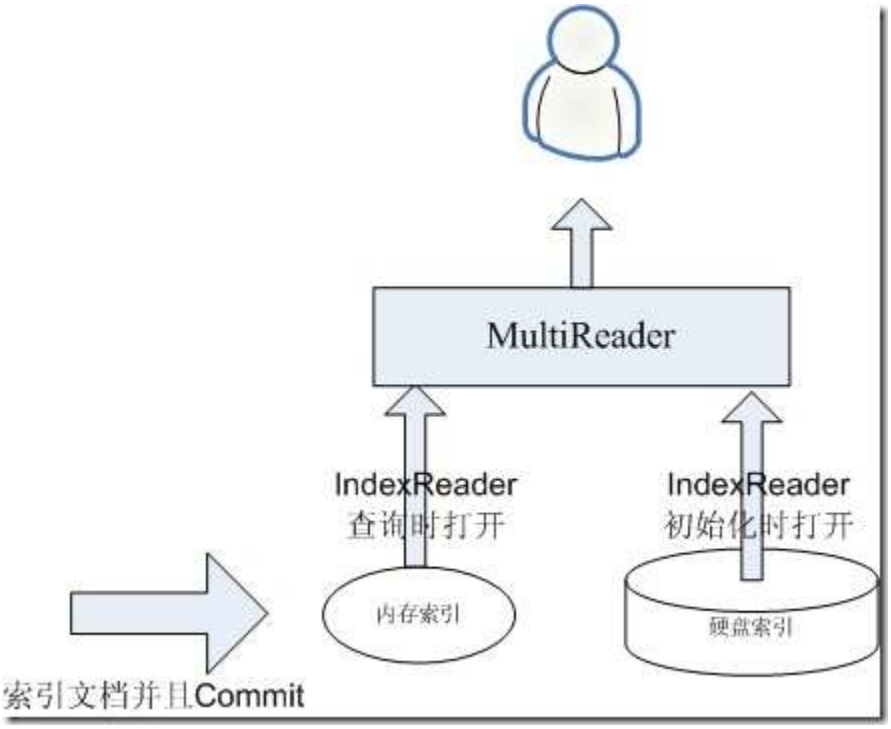
所以要构建实时的索引，就需要内存中的索引 RAMDirectory 和硬盘上的索引 FSDirectory 相互配合来解决问题。

1、初始化阶段

首先假设我们硬盘上已经有一个索引 FileSystemIndex，由于 IndexReader 打开此索引非常的慢，因而其是需要事先打开的，并且不会时常的重新打开。

我们在内存中有一个索引 MemoryIndex，新来的文档全部索引到内存索引中，并且是索引完 IndexWriter 就 commit，IndexReader 就重新打开，这两个操作时非常快的。

如下图，则此时新索引的文档全部能被用户看到，达到实时的目的。



2、合并索引阶段

然而经过一段时间，内存中的索引会比较大了，如果不合并到硬盘上，则可能造成内存不够用，则需要进行合并的过程。

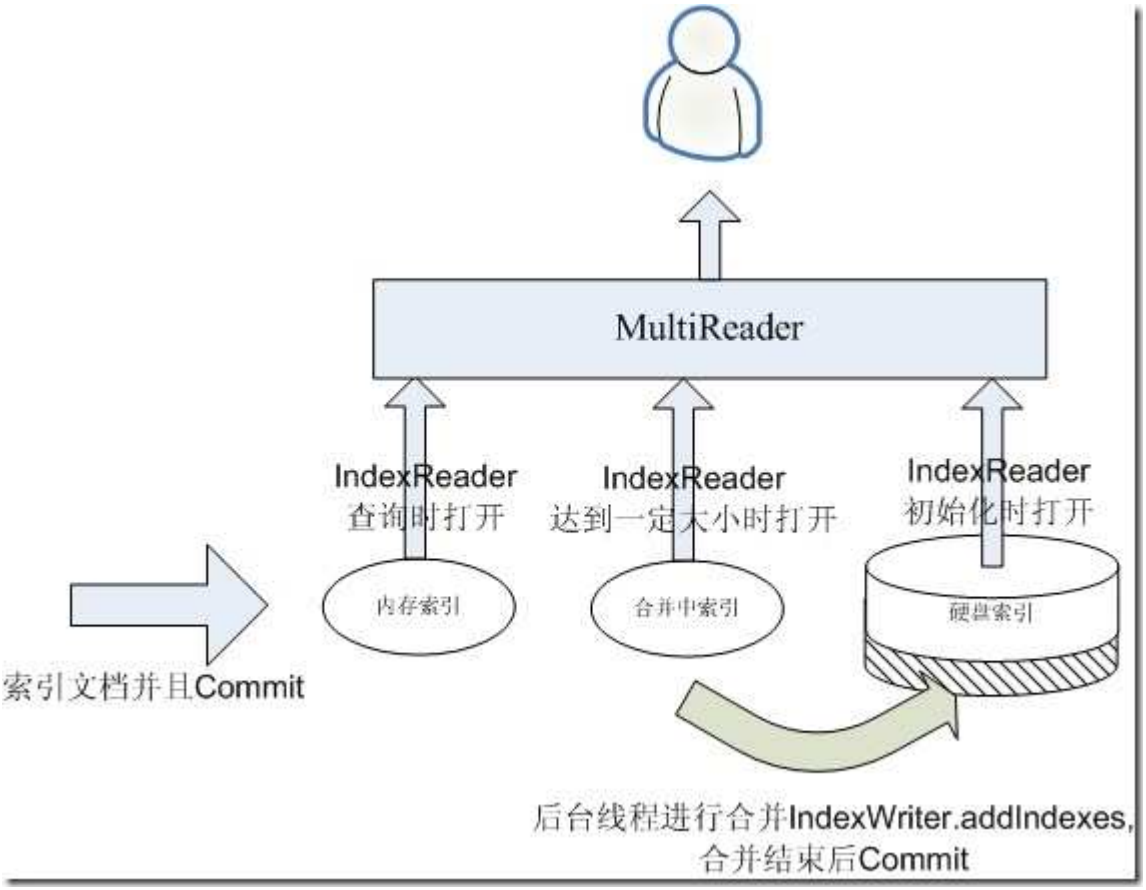
当然在合并的过程中，我们依然想让我们的搜索是实时的，这是就需要一个过渡的索引，我们称为 MergingIndex。

一旦内存索引达到一定的程度，则我们重新建立一个空的内存索引，用于合并阶段索引新的文档，然后将原来的内存索引称为合并中索引，并启动一个后台线程进行合并的操作。

在合并的过程中，如果有查询过来，则需要三个 IndexReader，一个是内存索引的 IndexReader 打开，这个过程是很快的，一个是合并中索引的 IndexReader 打开，这个过程也是很快的，一个是已经打开的硬盘索引的 IndexReader，无需重新打开。这三个 IndexReader 可以覆盖所有的文档，唯一有可能重复的是，硬盘索引中已经有一些从合并中索引合并过去的文档了，然而不用担心，根据 Lucene 的事务性，在硬盘索引的 IndexReader 没有重新打开的情况下，背后的合并操作它是看不到的，因而这三个 IndexReader 所看到的文档应该是既不少也不多。

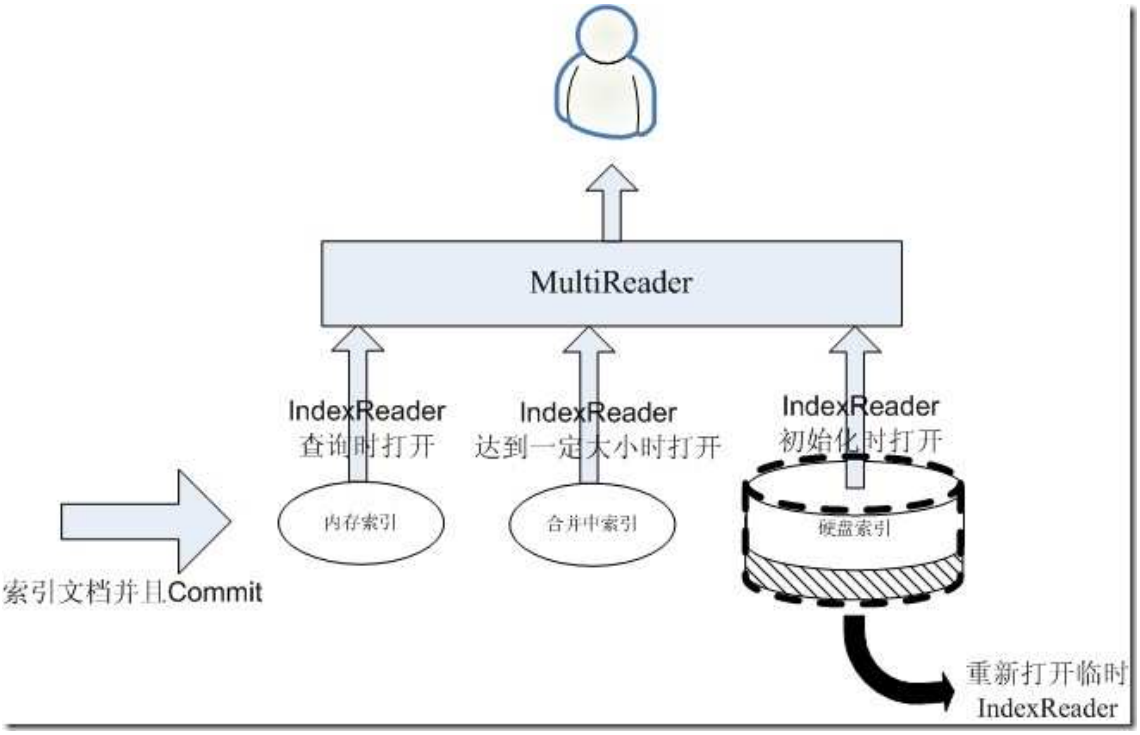
合并使用 IndexWriter(硬盘索引).addIndexes(IndexReader(合并中索引))，合并结束后 Commit。

如下图：



3、重新打开硬盘索引的 IndexReader

当合并结束后，是应该重新打开硬盘索引的时候了，然而这是一个可能比较慢的过程，在此过程中，我们仍然想保持实时性，因而在此过程中，合并中的索引不能丢弃，硬盘索引的 IndexReader 也不要动，而是为硬盘索引打开一个临时的 IndexReader，在打开的过程中，如果有搜索进来，返回的仍然是上述的三个 IndexReader，仍能够不多不少的看到所有的文档，而将要打开的临时的 IndexReader 将能看到合并中索引和原来的硬盘索引所有的文档，此 IndexReader 并不返回给客户。如下图：



4、替代 IndexReader

当临时的 IndexReader 被打开的时候，其看到的是合并中索引的 IndexReader 和硬盘索引原来的 IndexReader 之和，下面要做的是:

(1) 关闭合并中索引的 IndexReader

(2) 抛弃合并中索引

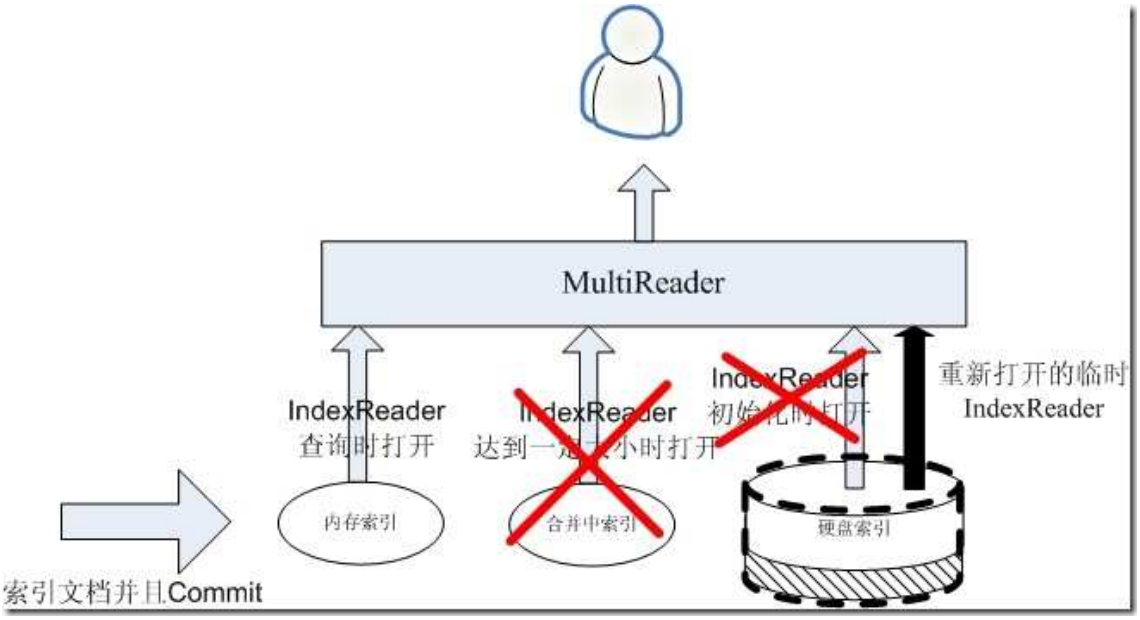
(3) 用临时的 IndexReader 替换硬盘索引原来的 IndexReader

(4) 关闭硬盘索引原来的 IndexReader。

上面说的这几个操作必须是原子性的，如果做了(2)但没有做(3)，如果来一个搜索，则将少看到一部分数据，如果做了(3)没有做(2)则，多看到一部分数据。

所以在进行上述四步操作的时候，需要加一个锁，如果这个时候有搜索进来的时候，或者在完全没有做的时候得到所有的 IndexReader，或者在完全做好的时候得到所有的 IndexReader，这时此搜索可能被 block，但是没有关系，这四步是非常快的，丝毫不影响替代性。

如下图：

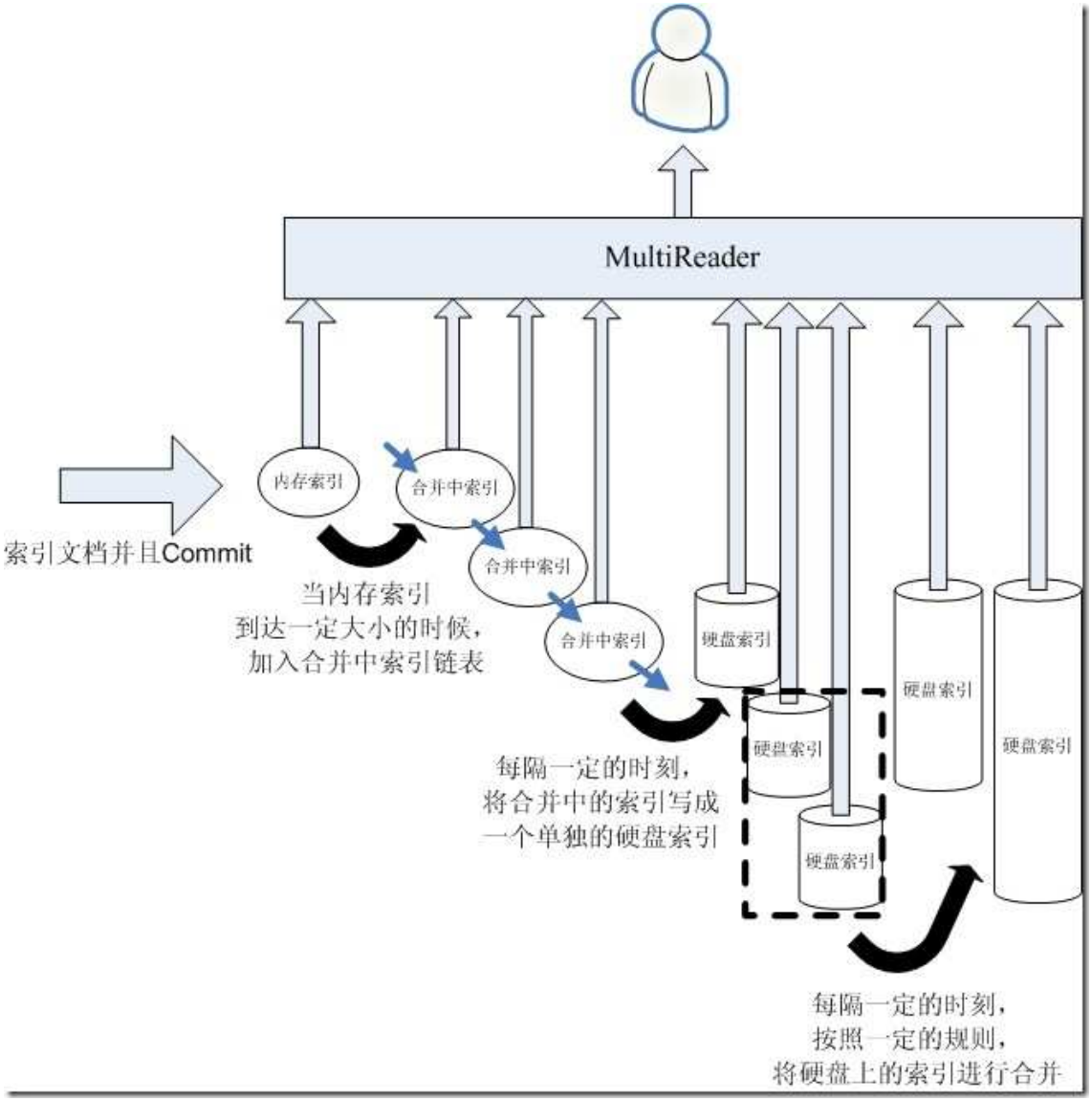


经过这几个过程，又达到了第一步的状态，则进行下一个合并的过程。

5、多个索引

有一点需要注意的是，在上述的合并过程中，新添加的文档是始终添加到内存索引中的，如果存在如下的情况，索引速度实在太快，在合并过程没有完成的时候，内存索引又满了，或者硬盘上的索引实在太大，合并和重新打开要花费太长的时间，使得内存索引以及满的情况下，还没有合并完成。

为了处理这种情况，我们可以拥有多个合并中的索引，多个硬盘上的索引，如下图：



- 新添加的文档永远是进入内存索引
- 当内存索引到达一定的大小的时候，将其加入合并中索引链表
- 有一个后台线程，每隔一定的时刻，将合并中索引写入一个新的硬盘索引中取。这样可以避免由于硬盘索引过大而合并较慢的情况。硬盘索引的 IndexReader 也是写完并重新打开后才替换合并中索引的 IndexReader，新的硬盘索引也可保证打开的过程不会花费太长时间。
- 这样会造成硬盘索引很多，所以，每隔一定的时刻，将硬盘索引合并成一个大的索引。也是合并完成后方才替换 IndexReader

大家可能会发现，此合并的过程和 Lucene 的段的合并很相似。然而 Lucene 的一个函数

IndexReader.reopen 一直是没有实现的，也即我们不能选择哪个段是在内存中的，可以被打开，哪些是硬盘中的，需要在后台打开然后进行替换，而 IndexReader.open 是会打开所有的内存中的和硬盘上的索引，因而会很慢，从而降低了实时性。