Annotated Lucene(源码剖析中文版)

Annotated Lucene 作者：naven

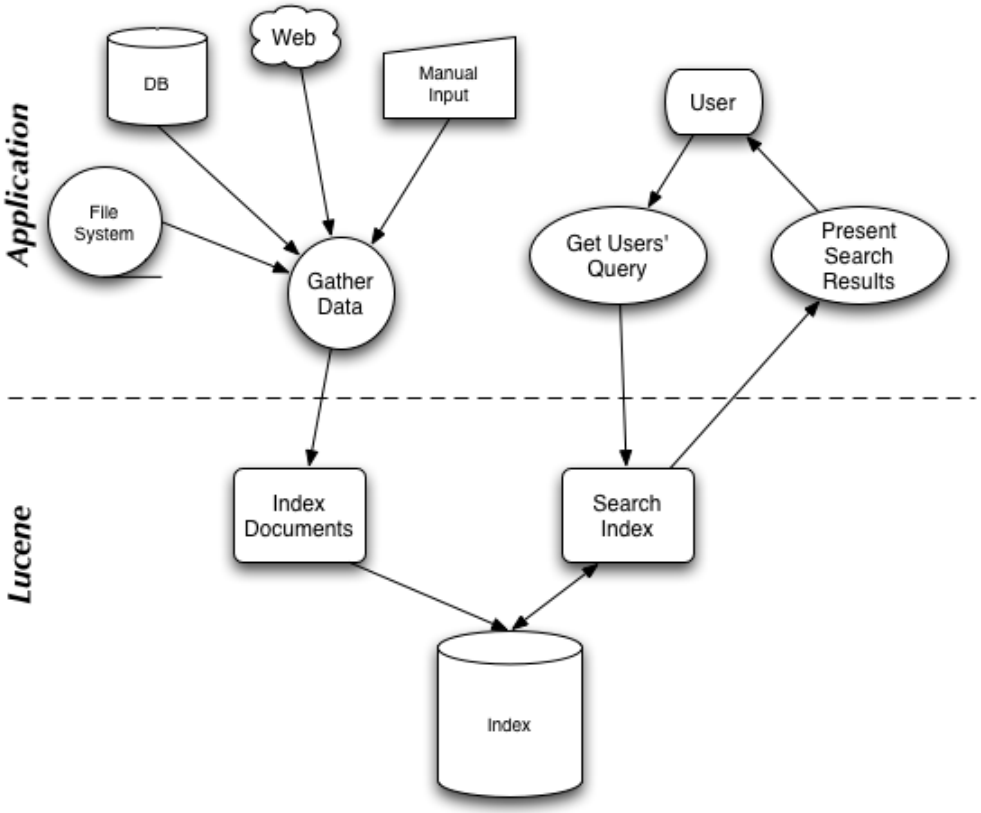
1 目录

Annotated Lucene(源码剖析中文版) ..... - 1 -
1 目录 ..... - 1 -
2 Lucene 是什么 ..... - 3 -
2.1.1 强大特性 ..... - 3 -
2.1.2 API 组成 ..... - 4 -
2.1.3 Hello World! ..... - 5 -
2.1.4 Lucene roadmap ..... - 6 -
3 索引文件结构 ..... - 7 -
3.1 索引数据术语和约定 ..... - 7 -
3.1.1 术语定义 ..... - 7 -
3.1.2 倒排索引（inverted indexing） ..... - 8 -
3.1.3 Fields 的种类 ..... - 8 -
3.1.4 片断（segments） ..... - 8 -
3.1.5 文档编号（document numbers） ..... - 9 -
3.1.6 索引结构概述 ..... - 9 -
3.1.7 索引文件中定义的数据类型 ..... - 10 -
3.2 索引文件结构 ..... - 10 -
3.2.1 索引文件概述 ..... - 10 -
3.2.2 每个 Index 包含的文件 ..... - 11 -
3.2.2.1 Segments 文件 ..... - 11 -
3.2.2.2 Lock 文件 ..... - 14 -
3.2.2.3 Deletable 文件 ..... - 14 -
3.2.2.4 Compound 文件（.cfs） ..... - 14 -
3.2.3 每个 Segment 包含的文件 ..... - 15 -
3.2.3.1 Field 信息（.fnm） ..... - 15 -
3.2.3.2 Field 数据（.fdx 和.fdt） ..... - 16 -
3.2.3.3 Term 字典（.tii 和.tis） ..... - 18 -
3.2.3.4 Term 频率数据（.frq） ..... - 21 -
3.2.3.5 Positions 位置信息数据（.prx） ..... - 23 -
3.2.3.6 Norms 调节因子文件（.nrm） ..... - 24 -
3.2.3.7 Term 向量文件 ..... - 25 -
3.2.3.8 删除的文档 （.del） ..... - 28 -
3.3 局限性（Limitations） ..... - 29 -
4 索引是如何创建的 ..... - 30 -
4.1 索引创建示例 ..... - 30 -

4.2 索引创建类 IndexWriter ..... - 30 -
4.2.1 org.apache.lucene.index.IndexWriter ..... - 31 -
4.2.2 org.apache.lucene.index.DocumentsWriter ..... - 33 -
4.3 索引创建过程 ..... - 34 -
4.3.1 DocFieldProcessorPerThread.processDocument() ..... - 36 -
4.3.2 DocInverterPerField.processFields() ..... - 37 -
4.3.3 TermsHashPerField.addToken() ..... - 38 -
4.3.4 FreqProxTermsWriterPerField.newTerm()/addTerm() ..... - 39 -
4.3.5 TermVectorsTermsWriterPerField.newTerm()/addTerm() ..... - 42 -
5 索引是如何存储的 ..... - 44 -
5.1 数据存储类 Directory ..... - 44 -
5.1.1 org.apache.lucene.store.Directory ..... - 44 -
5.1.2 org.apache.lucene.store.FSDirectory ..... - 44 -
5.1.3 org.apache.lucene.store.RAMDirectory ..... - 45 -
5.1.4 org.apache.lucene.store.IndexInput ..... - 46 -
5.1.5 org.apache.lucene.store.IndexOutput ..... - 47 -
6 文档内容是如何分析的 ..... - 49 -
6.1 文档分析类 Analyzer ..... - 49 -
6.1.1 org.apache.lucene.store.Analyzer ..... - 49 -
6.1.2 org.apache.lucene.store.StandardAnalyzer ..... - 49 -
7 如何给文档评分 ..... - 50 -
7.1 文档评分类 Similarity ..... - 50 -
7.1.1 org.apache.lucene.search.Similarity ..... - 50 -
7.2 Similarity 评分公式 ..... - 51 -

2 Lucene 是什么

Apache Lucene 是一个高性能（high-performance）的全能的全文检索（full-featured text search engine）的搜索引擎框架库，完全（entirely）使用 Java 开发。它是一种技术（technology），适合于（suitable for）几乎（nearly）任何一种需要全文检索（full-text search）的应用，特别是跨平台（cross-platform）的应用。



2.1.1 强大特性

Lucene 通过一些简单的接口（simple API）提供了强大的特征（powerful features）：

可扩展的高性能的索引能力（Scalable, High-Performance Indexing）

- ✓ 超过 20M/分钟的处理能力（Pentium M 1.5GHz）
- ✓ 很少的 RAM 内存需求，只需要 1MB heap
- ✓ 增量索引（incremental indexing）的速度与批量索引（batch indexing）的速度一样快
- ✓ 索引的大小粗略（roughly）为被索引的文本大小的 20-30%

强大的精确的高效率的检索算法（Powerful, Accurate and Efficient Search Algorithms）

- ✓ 分级检索（ranked searching）能力，最好的结果优先推出在前面
- ✓ 很多强大的 query 种类：phrase queries, wildcard queries, proximity queries, range queries 等

- ✓ 支持域检索（fielded searching），如标题、作者、正文等
- ✓ 支持日期范围检索（date-range searching）
- ✓ 可以按任意域排序（sorting by any field）
- ✓ 支持多个索引的检索（multiple-index searching）并合并结果集（merged results）
- ✓ 允许更新和检索（update and searching）并发进行（simultaneous）

跨平台解决方案（Cross-Platform Solution）

- ✓ 以 Open Source 方式提供并遵循 Apache License，允许你可以在即包括商业应用也包括 Open Source 程序中使用 Lucene
- ✓ 100%-pure Java（纯 Java 实现）
- ✓ 提供其他开发语言的实现版本并且它们的索引文件是兼容的

2.1.2 API 组成

Lucene API 被分成（divide into）如下几种包（package）

1. **org.apache.lucene.analysis**
   定义了一个抽象的 Analyser API，用于将 text 文本从一个 java.io.Reader 转换成一个 TokenStream，即包括一些 Tokens 的枚举容器（enumeration）。一个 TokenStream 的组成（compose）是通过在一个 Tokenizer 的输出的结果上再应用 TokenFilters 生成的。一些少量的 Analysers 实现已经提供，包括 StopAnalyzer 和基于语法（gramar-based）分析的 StandardAnalyzer。

2. **org.apache.lucene.document**
   提供一个简单的 Document 类，一个 document 只不过包括一系列的命名了（named）的 Fields（域），它们的内容可以是文本（strings）也可以是一个 java.io.Reader 的实例。

3. **org.apache.lucene.index**
   提供两个主要地饿类，一个是 IndexWriter 用于创建索引并添加文档（document），另一个是 IndexReader 用于访问索引中的数据。

4. **org.apache.lucene.search**
   提供数据结构（data structures）来呈现（represent）查询（queries）：TermQuery 用于单个的词（individual words），PhraseQuery 用于短语，BooleanQuery 用于通过 boolean 关系组合（combinations）在一起的 queries。而抽象的 Searcher 用于转变 queries 为命中的结果（hits）。IndexSearcher 实现了在一个单独（single）的 IndexReader 上检索。

5. **org.apache.lucene.queryParser**
   使用 JavaCC 实现一个 QueryParser。

6. **org.apache.lucene.store**
   定义了一个抽象的类用于存储呈现的数据(storing persistent data)，即 Directory（目录），一个收集器（collection）包含了一些命名了的文件（named files），它们通过一个 IndexOutput 来写入，以及一个 IndexInput 来读取。提供了两个实现，FSDirectory 使用一个文件系统目录来存储文件，而另一个

RAMDirectory 则实现了将文件当作驻留内存的数据结构（memory-resident data structures）。

7. **org.apache.lucene.util**
   包含了一小部分有用（handy）的数据结构，如 BitVector 和 PriorityQueue 等。

2.1.3 Hello World!

下面是一段简单的代码展示如何使用 Lucene 来进行索引和检索（使用 JUnit 来检查结果是否是我们预期的）：

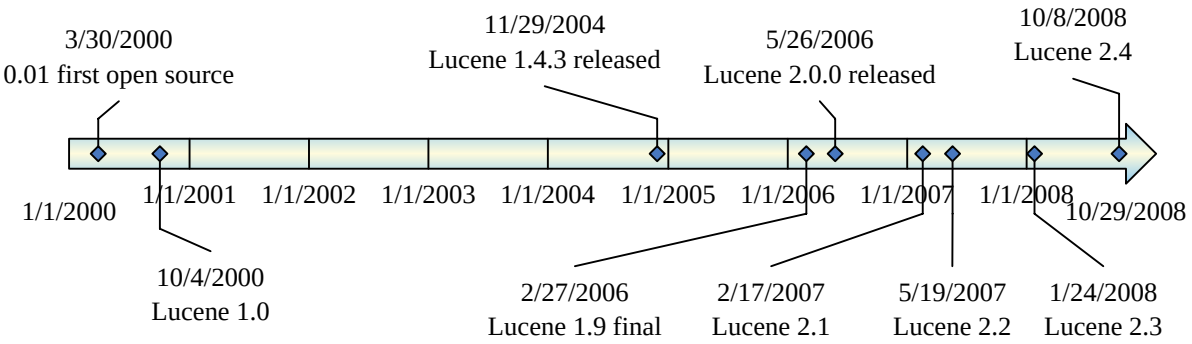
```
// Store the index in memory:
Directory directory = new RAMDirectory();
// To store an index on disk, use this instead:
//Directory directory = FSDirectory.getDirectory("/tmp/testindex");
IndexWriter iwriter = new IndexWriter(directory, analyzer, true);
iwriter.setMaxFieldLength(25000);
Document doc = new Document();
String text = "This is the text to be indexed.";
doc.add(new Field("fieldname", text, Field.Store.YES,
    Field.Index.TOKENIZED));
iwriter.addDocument(doc);
iwriter.optimize();
iwriter.close();

// Now search the index:
IndexSearcher isearcher = new IndexSearcher(directory);
// Parse a simple query that searches for "text":
QueryParser parser = new QueryParser("fieldname", analyzer);
Query query = parser.parse("text");
Hits hits = isearcher.search(query);
assertEquals(1, hits.length());
// Iterate through the results:
for (int i = 0; i < hits.length(); i++) {
  Document hitDoc = hits.doc(i);
  assertEquals("This is the text to be indexed.", hitDoc.get("fieldname"));
}
isearcher.close();
directory.close();
```

为了使用 Lucene，一个应用程序需要做如下几件事：

1. 通过添加一系列 Fields 来创建一批 Documents 对象。
2. 创建一个 IndexWriter 对象，并且调用它的 AddDocument()方法来添加进 Documents。
3. 调用 QueryParser.parse()处理一段文本（string）来建造一个查询（query）对象。
4. 创建一个 IndexReader 对象并将查询对象传入到它的 search()方法中。

2.1.4 Lucene roadmap



3 索引文件结构

为了使用 Lucene 来索引数据，首先你得把它转换成一个纯文本（plain-text）tokens 的数据流（stream），并通过它创建出 Document 对象，其包含的 Fields 成员容纳这些文本数据。一旦你准备好些 Document 对象，你就可以调用 IndexWriter 类的 addDocument(Document)方法来传递这些对象到 Lucene 并写入索引中。当你做这些的时候，Lucene 首先分析（analyzer）这些数据来使得它们更适合索引。详见《Lucene In Action》

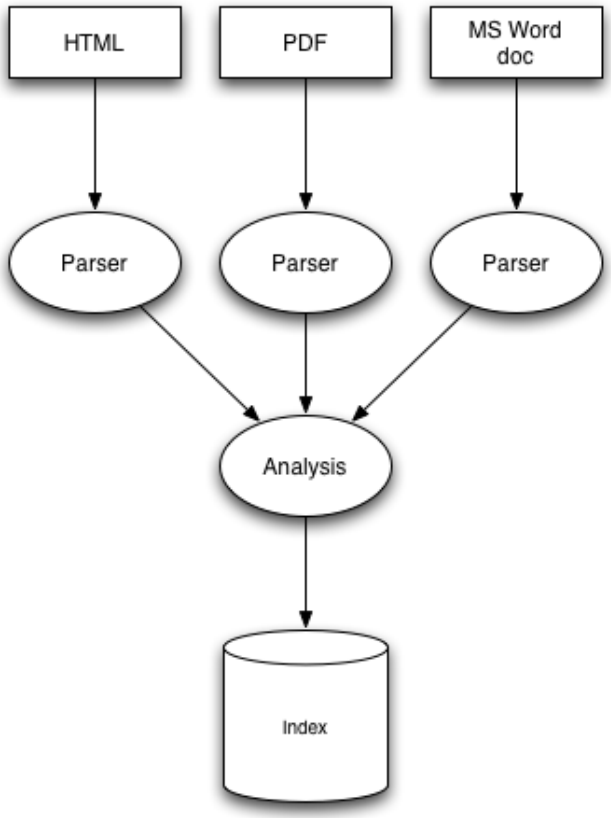


图 4-1 描述如何将各种格式文档建立索引及过程

下面先了解一下索引结构的一些术语。

3.1 索引数据术语和约定

3.1.1 术语定义

Lucene 中基本的概念（fundamental concepts）是 index、Document、Field 和 term。

- 一条索引（index）包含（contains）了一连串（a sequence of）文档（documents）。
- 一个文档（document）是由一连串 fields 组成。
- 一个 field 是由一连串命名了（a named sequence of）的 terms 组成。
- 一个 term 是一个 string（字符串）。

相同的字符串（same string）但是在两个不同的 fields 中被认为（considered）是不同的 term。因此（thus）term 被描述为（represent as）一对字符串（a pair of strings），第一个 string 取名（naming）为该 field 的名字，

第二个 string 取名为包含在该 field 中的文本（text within the field）。

3.1.2 倒排索引（inverted indexing）

索引（index）存储 terms 的统计数据（statistics about terms），为了使得基于 term 的检索（term-based search）效率更高（more efficient）。Lucene 的索引分成（fall into）被广为熟悉的（known as）索引种类（family of indexex）叫做倒排索引（inverted index）。这是因为它可以列举（list），对一个 term 来说，所有包含它的文档（documents that contain it）。这与自然关联规则（natural relationship）是相反，即由 documents 列举它所包含的 terms。

3.1.3 Fields 的种类

在 Lucene 中，fields 可以被存储（stored），在这种情况（in which case）下它们的文本被逐字地（literally）以一种非倒排的方式(in non-inverted manner)存储进 index 中。那些被倒排的 fields(that are inverted)称为(called)被索引（indexed）。一个 field 可以都被存储（stored）并且被索引（indexed）。

一个 field 的文本可以被分解为（be tokenized into）terms 以便被索引（indexed），或者 field 的文本可以被逐字地使用为（used literally as）一个 term 来被索引（be indexed）。大多数 fields 被分解（be tokenized），但是有时候对某种唯一性（certain identifier）的 field 来逐字地索引（be indexed literally）又是非常有用的，如 url。

3.1.4 片断（segments）

Lucene 的索引可以由多个复合的子索引(multiple sub-indexes)或者片断(segments)组成(be composed of)。每一个 segment 都是一个完全独立的索引（fully independent index），它能够被分离地进行检索（be searched seperately）。索引按如下方式进行演化（evolve）：

1. 为新添加的文档（newly added documents）创建新的片断（segments）。
2. 合并已存在的片断（merging existing segments）。

检索可以涉及（involve）多个复合（multiple）的 segments，并且/或者多个复合（multiple）的 indexes。每一个 index 潜在地（potentially）包含（composed of）一套（a set of）segments。

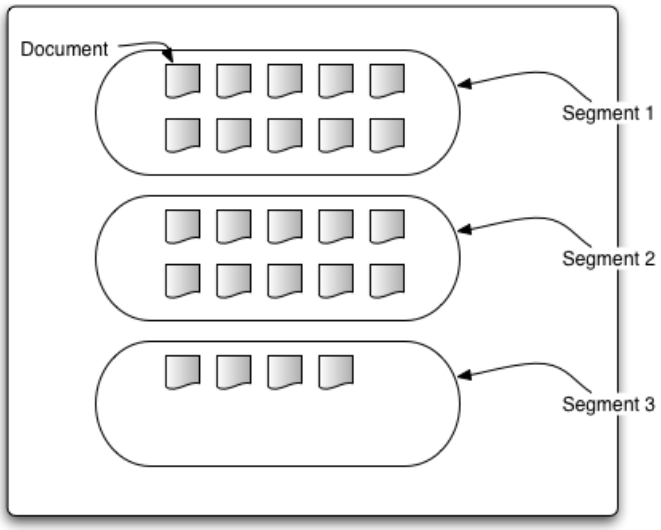


图 4-2 描述 segment 和文档的结构

3.1.5 文档编号（document numbers）

在内部（internally），Lucene 通过一个整数的（interger）文档编号（document number）来表示文档。第一篇被添加到索引中的文档编号为 0（be numbered zero），每一篇随后（subsequent）被添加的 document 获得一个比前一篇更大的数字（a number one greater than the previous）。

需要注意的是一篇文档的编号（document's number）可以更改，所以在 Lucene 之外（outside of）存储这些编号时需要特别小心（caution should be taken）。详细地说（in particular），编号在如下的情况（following situations）可以更改：

1. 存储在每个 segment 中的编号仅仅是在所在的 segment 中是唯一的（unique），在它能够被使用在（be used in）一个更大的上下文（a larger context）中前必须被转变（converted）。标准的技术（standard technique）是给每一个 segment 分配（allocate）一个范围的值（a range of values），基于该 segment 所使用的编号的范围（the range of numbers）。为了将一篇文档的编号从一个 segment 转变为一个扩展的值（an external value），该片断的基础的文档编号（base document number）被添加（is added）。为了将一个扩展的值（external value）转变回一个 segment 的特定的值（specific value），该 segment 将该扩展的值所在的范围标识出来（be indentified），并且该 segment 的基础值（base value）将被减少（substracted）。例如，两个包含 5 篇文档的 segments 可能会被合并（combined），所以第一个 segment 有一个基础的值(base value)为 0，第二个 segment 则为 5。在第二个 segment 中的第 3 篇文档(document three from the second segment）将有一个扩展的值为 8。
2. 当文档被删除的时候，在编号序列中（in the numbering）将产生（created）间隔段（gaps）。这些最后（eventually）在索引通过合并演进时（index evolves through merging）将会被清除（removed）。当 segments 被合并后(merged)，已删除的文档将会被丢弃(dropped)，一个刚被合并的(freshly-merged) segment 因此在它的编号序列中（in its numbering）不再有间隔段（gaps）。

3.1.6 索引结构概述

每一个片断的索引（segment index）管理（maintains）如下的数据：

1. **Fields 名称：**这包含了（contains）在索引中使用的一系列 fields 的名称（the set of field names）。
2. **已存储的 field 的值：**它包含了，对每篇文档来说，一个属性-值数据对（attribute-value pairs）的清单（a list of），其中属性即为 field 的名字。这些被用来存储关于文档的备用信息(auxiliary information)，比如它的标题（title）、url、或者一个访问一个数据库（database）的唯一标识（identifier）。这套存储的 fields 就是那些在检索时对每一个命中的（hits）文档所返回的（returned）信息。这些是通过文档编号（document number）来做为 key 得到的。
3. **Term 字典（dictionary）：**一个包含（contains）所有 terms 的字典，被使用在所有文档中所有被索引的 fields 中。它还包含了该 term 所在的文档的数目(the number of documents which contains the term)，并且指向了（pointer to）term 的频率（frequency）和接近度（proximity）的数据（data）。
4. **Term 频率数据(frequency data)：**对字典中的每一个 term 来说，所有包含该 term(contains the term)的文档的编号（numbers of all documents），以及该 term 出现在该文档中的频率（frequency）。
5. **Term 接近度数据（proximity data）：**对字典中的每一个 term 来说，该 term 出现在（occur）每一篇文档中的位置（positions）。
6. **调整因子(normalization factors)：**对每一篇文档的每一个 field 来说，为一个存储的值(a value is stored)用来加入到（multiply into）命中该 field 的分数（score for hits on that field）中。
7. **Term 向量（vectors）：**对每一篇文档的每一个 field 来说，term 向量（有时候被称做文档向量）可以

被存储。一个 term 向量由 term 文本和 term 的频率（frequency）组成（consists of）。怎么添加 term 向量到你的索引中请参考 Field 类的构造方法（constructors）。

8. **删除的文档（deleted documents）：**一个可选的（optional）文件标示（indicating）哪一篇文档被删除。

关于这些项的详细信息在随后的章节（subsequent sections）中逐一介绍。

3.1.7 索引文件中定义的数据类型

数据类型	所占字节长度（字节）	说明
Byte	1	基本数据类型，其他数据类型以此为基础定义
UInt32	4	32 位无符号整数，高位优先
UInt64	8	64 位无符号整数，高位优先
VInt	不定，最少 1 字节	动态长度整数，每字节的最高位表明还剩多少字节，每字节的低七位表明整数的值，高位优先。可以认为值可以为无限大。其示例如下 值 字节 1 字节 2 字节 3 0 00000000 1 00000001 2 00000010 127 01111111 128 10000000 00000001 129 10000001 00000001 130 10000010 00000001 16383 10000000 10000000 00000001 16384 10000001 10000000 00000001 16385 10000010 10000000 00000001
Chars	不定，最少 1 字节	采用 UTF-8 编码[20]的 Unicode 字符序列
String	不定，最少 2 字节	由 VInt 和 Chars 组成的字符串类型，VInt 表示 Chars 的长度，Chars 则表示了 String 的值

3.2 索引文件结构

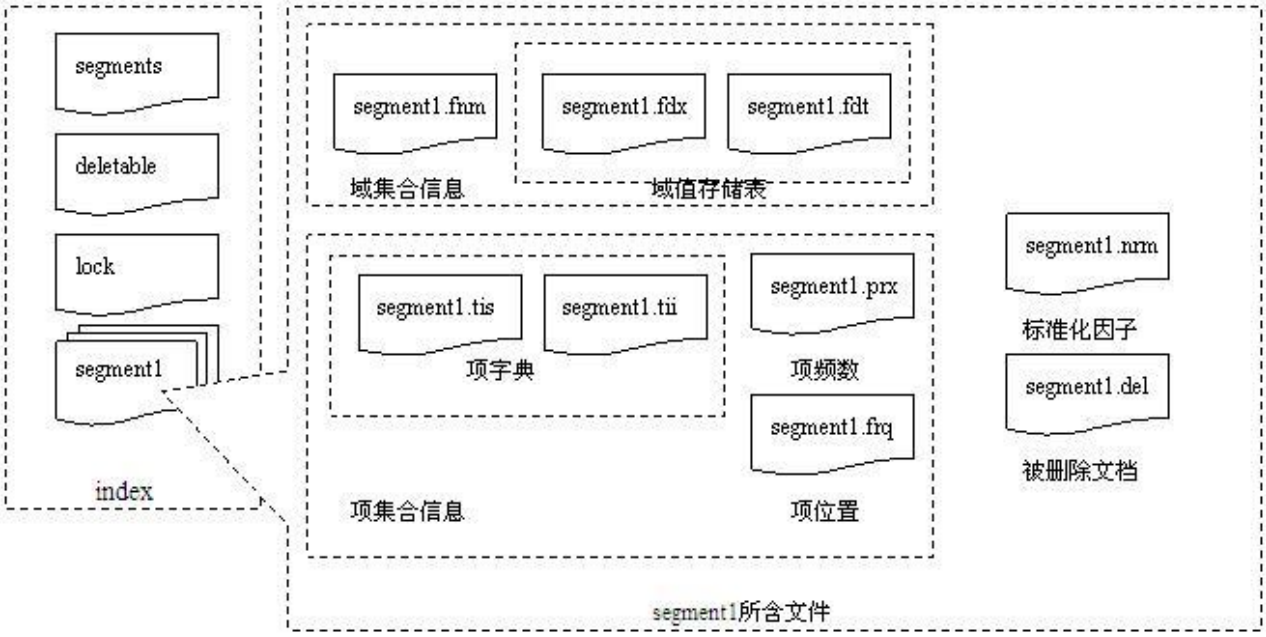
3.2.1 索引文件概述

Lucene 使用文件扩展名标识不同的索引文件，文件名标识不同版本或者代（generation）的索引片段（segment）。如.fnm 文件存储域 Fields 名称及其属性，.fdt 存储文档各项域数据，.fdx 存储文档在 fdt 中的偏移位置即其索引文件，.frq 存储文档中 term 位置数据，.tii 文件存储 term 字典，.tis 文件存储 term 频率数据，.prx 存储 term 接近度数据，.nrm 存储调节因子数据，另外 segments_X 文件存储当前最新索引片段的信息，其中 X 为其最新修改版本，segments.gen 存储当前版本即 X 值。

本节介绍的文件存在于每个索引中（exist one-per-index），下面的图描述了一个典型的 lucene 索引文件列表：

```
-rw-r--r-- 1 root root 10955227 07-22 01:40 _0.fdt
-rw-r--r-- 1 root root   320112 07-22 01:40 _0.fdx
-rw-r--r-- 1 root root      142 07-22 01:40 _0.fnm
-rw-r--r-- 1 root root 39675208 07-22 01:40 _0.frq
-rw-r--r-- 1 root root   360130 07-22 01:40 _0.nrm
-rw-r--r-- 1 root root 69944317 07-22 01:40 _0.prx
-rw-r--r-- 1 root root    33361 07-22 01:40 _0.tii
-rw-r--r-- 1 root root  2440334 07-22 01:40 _0.tis
-rw-r--r-- 1 root root       41 07-22 01:40 segments_3
-rw-r--r-- 1 root root       20 07-22 01:40 segments.gen
```

如果将它们的关系划成图则如下所示（该图来自网上，出处忘了）：



这些文件中存储数据的详细结构是怎样的呢，下面几个小节逐一介绍它们，熟悉它们的结构非常有助于优化 Lucene 的查询和索引效率和存储空间等。

3.2.2 每个 Index 包含的文件

下面几节介绍的文件存在于每个索引 index 中，并且只有一份。

3.2.2.1 Segments 文件

索引中活动（active）的 Segments 被存储在 segment info 文件中，*segments_N*，在索引中可能会包含一个或多个 *segments_N* 文件。然而，最大一代的那个文件（the one with largest generation）是活动的片断文件（这时更旧的 *segments_N* 文件依然存在（are present）是因为它们暂时（temporarily）还不能被删除，或者，一个 writer 正在处理提交请求（in the process of committing），或者一个用户定义的（custom）IndexDeletionPolicy 正被使用）。这个文件按照名称列举每一个片断（lists each segment by name），详细描述分离的标准（seperate norm）

和要删除的文件（deletion files），并且还包含了每一个片断的大小。

对 2.1 版本来说，还有一个文件 *segments.gen*。这个文件包含了该索引中当前生成的代（current generation）（*segments_N* 中的*_N*）。这个文件仅用于一个后退处理（fallback）以防止（in case）当前代（current generation）不能被准确地（accurately）通过单独地目录文件列举（by directory listing alone）来确定（determened）（由于某些 NFS 客户端因为基于时间的目录（time-based directory）的缓存终止（cache expiration）而引起）。这个文件简单地包含了一个 int32 的版本头（version header）（SegmentInfos.FORMAT_LOCKLESS=-2），遵照代的记录（followed by the generation recorded）规则，对 int64 来说会写两次（write twice）。

版本	包含的项	数目	类型	描述
2.1 之前版本	Format	1	Int32	在 Lucene1.4 中为-1，而在 Lucene 2.1 中为-3（SegmentsInfos.FORMAT_SINGLE_NORM_FILE）
	Version	1	Int64	统计在删除和添加文档时，索引被更改了多少次。
	NameCounter	1	Int32	用于为新的片断文件生成新的名字。
	SegCount	1	Int32	片断的数目
	SegName	SegCount	String	片断的名字，用于所有构成片断索引的文件的文件名前缀。
	SegSize	SegCount	Int32	包含在片断索引中的文档的数目。
2.1 及之后版本	Format	1	Int32	在 Lucene 2.1 和 Lucene 2.2 中为 -3（SegmentsInfos.FORMAT_SINGLE_NORM_FILE）
	Version	1	Int64	同上
	NameCounter	1	Int32	同上
	SegCount	1	Int32	同上
	SegName	SegCount	String	同上
	SegSize	SegCount	Int32	同上
	DelGen	SegCount	Int64	为分离的删除文件的代的数目（generation count of the separate deletes file），如果值为-1，表示没有分离的删除文件。如果值为 0，表示这是一个 2.1 版本之前的片断，这时你必须检查文件是否存在*_X.del* 这样的文件。任意大于 0 的值，表示有分离的删除文件，文件名为*_X_N.del*。
	HasSingleNormFile	SegCount	Int8	该值如果为 1，表示 Norm 域（field）被写为一个单一连接的文件（single joined file）中（扩展名为*.nrm*），如果值为 0，表示每一个 field 的 norms 被存储为分离的*.fN* 文件中，参考下面的“标准化因素（Normalization Factors）”
	NumField	SegCount	Int32	表示 NormGen 数组的大小，如果为-1 表示没有 NormGen 被存储。
	NormGen	SegCount * NumField	Int64	记录分离的标准文件（separate norm file）的代（generation），如果值为-1，表示没有 normGens 被存储，并且当片断文件是 2.1 之前版本生成的时，它们全部被假设为 0（assumed to be 0）。而当片断文件是 2.1 及更高版本生成的时，它们全部被假设为-1。这时这个代（generation）的意义与上面 DelGen 的意义一样。

	IsCompoundFile	SegCount	Int8	记录是否该片断文件被写为一个复合的文件，如果值为-1 表示它不是一个复合文件（compound file），如果为 1 则为一个复合文件。另外如果值为 0，表示我们需要检查文件系统是否存在*_X.cfs*。
2.3	Format	1	Int32	在 Lucene 2.3 中为 -4 (SegmentInfos.FORMAT_SHARED_DOC_STORE)
	Version	1	Int64	同上
	NameCounter	1	Int32	同上
	SegCount	1	Int32	同上
	SegName	SegCount	String	同上
	SegSize	SegCount	Int32	同上
	DelGen	SegCount	Int64	同上
	DocStoreOffset	1	Int32	如果值为-1 则该 segment 有自己的存储文档的 fields 数据和 term vectors 的文件，并且 DocStoreSegment, DocStoreIsCompoundFile 不会存储。在这种情况下，存储 fields 数据（*.fdt 和*.fdx 文件）以及 term vectors 数据（*.tvf 和*.tvd 和*.tvx 文件）的所有文件将存储在该 segment 下。另外，DocStoreSegment 将存储那些拥有共享的文档存储文件的 segment 。DocStoreIsCompoundFile 值为 1 如果 segment 存储为 compound 文件格式（如 .cfx 文件），并且 DocStoreOffset 值为那些共享文档存储文件中起始的文档编号，即该 segment 的文档开始的位置。在这种情况下，该 segment 不会存储自己的文档数据文件，而是与别的 segment 共享一个单一的数据文件集。
	[DocStoreSegment]	1	String	如上
	[DocStoreIsCompoundFile]	1	Int8	如上
	HasSingleNormFile	SegCount	Int8	同上
	NumField	SegCount	Int32	同上
	NormGen	SegCount * NumField	Int64	同上
	IsCompoundFile	SegCount	Int8	同上
2.4 及以上	Format	1	Int32	在 Lucene 2.4 中为 -7 (SegmentInfos.FORMAT_HAS_PROX)
	Version	1	Int64	同上
	NameCounter	1	Int32	同上
	SegCount	1	Int32	同上
	SegName	SegCount	String	同上
	SegSize	SegCount	Int32	同上
	DelGen	SegCount	Int64	同上
	DocStoreOffset	1	Int32	同上
	[DocStoreSegment]	1	String	同上

	[DocStoreIsCompoundFile]	1	Int8	同上
	HasSingleNormFile	SegCount	Int8	同上
	NumField	SegCount	Int32	同上
	NormGen	SegCount * NumField	Int64	同上
	IsCompoundFile	SegCount	Int8	同上
	DeletionCount	SegCount	Int32	记录该 segment 中删除的文档数目
	HasProx	SegCount	Int8	值为 1 表示该 segment 中至少一个 fields 的 omitTf 设置为 false，否则为 0
	Checksum	1	Int64	存储 segments_N 文件中直到 checksum 的所有字节的 CRC32 checksum 数据，用来校验打开的索引文件的完整性（integrity）。

3.2.2.2 Lock 文件

写锁（write lock）文件名为“*write.lock*”，它缺省存储在索引目录中。如果锁目录（lock directory）与索引目录不一致，写锁将被命名为“*XXXX-write.lock*”，其中“*XXXX*”是一个唯一的前缀（unique prefix），来源于（derived from）索引目录的全路径（full path）。当这个写锁出现时，一个 writer 当前正在修改索引（添加或者清除文档）。这个写锁确保在一个时刻只有一个 writer 修改索引。

需要注意的是在 2.1 版本之前（prior to），Lucene 还使用一个 commit lock，这个锁在 2.1 版本里被删除了。

3.2.2.3 Deletable 文件

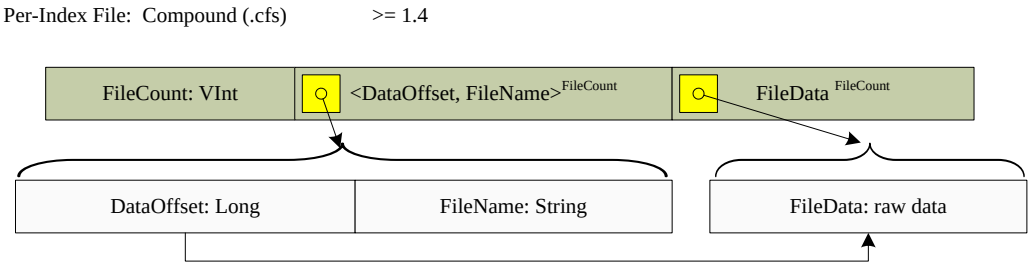
在 Lucene 2.1 版本之前，有一个“*deletable*”文件，包含了那些需要被删除文档的详细资料。在 2.1 版本后，一个 writer 会动态地（dynamically）计算哪些文件需要删除，因此，没有文件被写入文件系统。

3.2.2.4 Compound 文件（.cfs）

从 Lucene 1.4 版本开始，compound 文件格式成为缺省信息。这是一个简单的容器（container）来服务所有下一章节（next section）描述的文件（除了.del 文件），格式如下：

版本	包含的项	数目	类型	描述
1.4 之后版本	FileCount	1	VInt	
	DataOffset	FileCount	Long	
	FileName	FileCount	String	
	FileData	FileCount	raw	Raw 文件数据是上面命名的所有单个的文件数据（the individual named above）。

结构如下图所示：



从 Lucene 2.3 版本开始，文档存储文件（存储 fields 和 term vectors 数据）能够让多个 segment 共享一个单独的文件集，当 compound 文件启用时，这些共享的文件将被添加进一个单独的 compound 文件，格式同上，但是文件扩展名为.cfx。

3.2.3 每个 Segment 包含的文件

剩下的文件（remaining files）都是 per-segment（每个片断文件），因此（thus）都用后缀来定义（defined by suffix）。

3.2.3.1 Field 信息（.fnm）

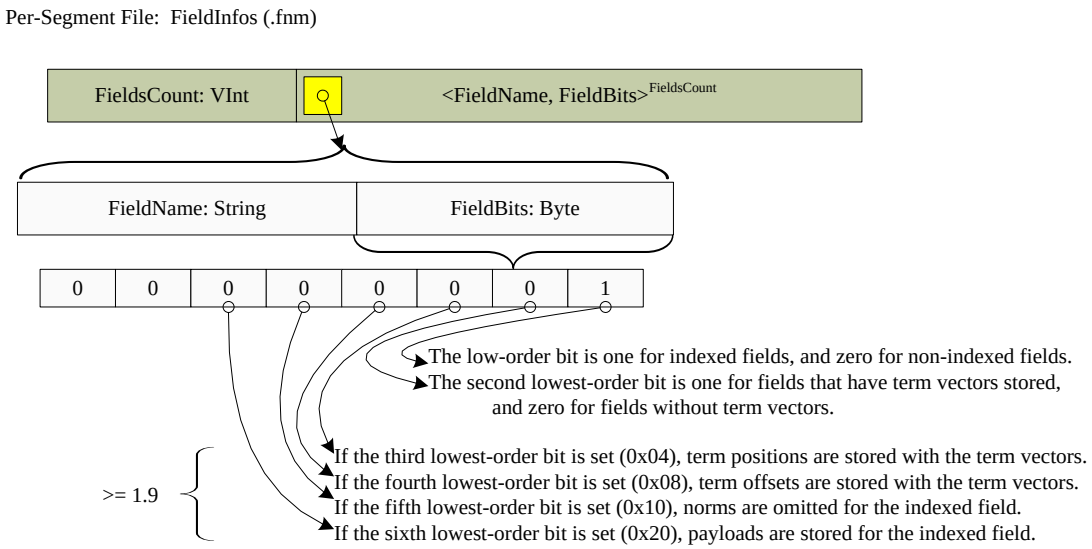
Field 的名字都存储在 Field 信息文件中，后缀是.fnm。

文件	包含的项	数目	类型	版本	描述
FieldsInfo (.fnm)	FieldsCount	1	VInt		
	FieldName	FieldsCount	String		
	FieldBits	FieldsCount	Byte		最低阶的 bit 位（low-order bit）值为 1 表示是被索引的 Fields，0 表示非索引的 Fields。
					第二个最低阶的 bit 位（second lowest-order bit）值为 1 表示该 Field 有 term 向量存储(term vectors stored)，0 表示该 field 没有 term 向量。
				>=1.9	如果第三个最低阶的 bit 位（third lowest-order bit）设置（0x04），term 的位置（term positions）将和 term 向量一起被存储（stored with term vectors）。
				>=1.9	如果第四个最低阶的 bit 位（fourth lowest-order bit）设置（0x08），term 的偏移（term offsets）将和 term 向量一起被存储（stored with term vectors）。
				>=1.9	如果第五个最低阶的 bit 位（fifth lowest-order bit）设置（0x10），norms 将对索引的 field 忽略掉（norms are omitted for the indexed field）。
				>=1.9	如果第六个最低阶的 bit 位（sixth lowest-order bit）设置（0x20），payloads 将为索引的 field 存储（payloads are stored for the indexed field）。

Fields 将使用它们在这个文件中的顺序来编号（fields are numbered by their order in this file）。因此 Field 为 0 表示为该文件中的第一个 field，1 表示下一个 field，一次类推。

需要注意的是，就像文档编号（document numbers）一样，field 编号（field numbers）与片断是相关的（are segment relative）。

结构如下图所示：



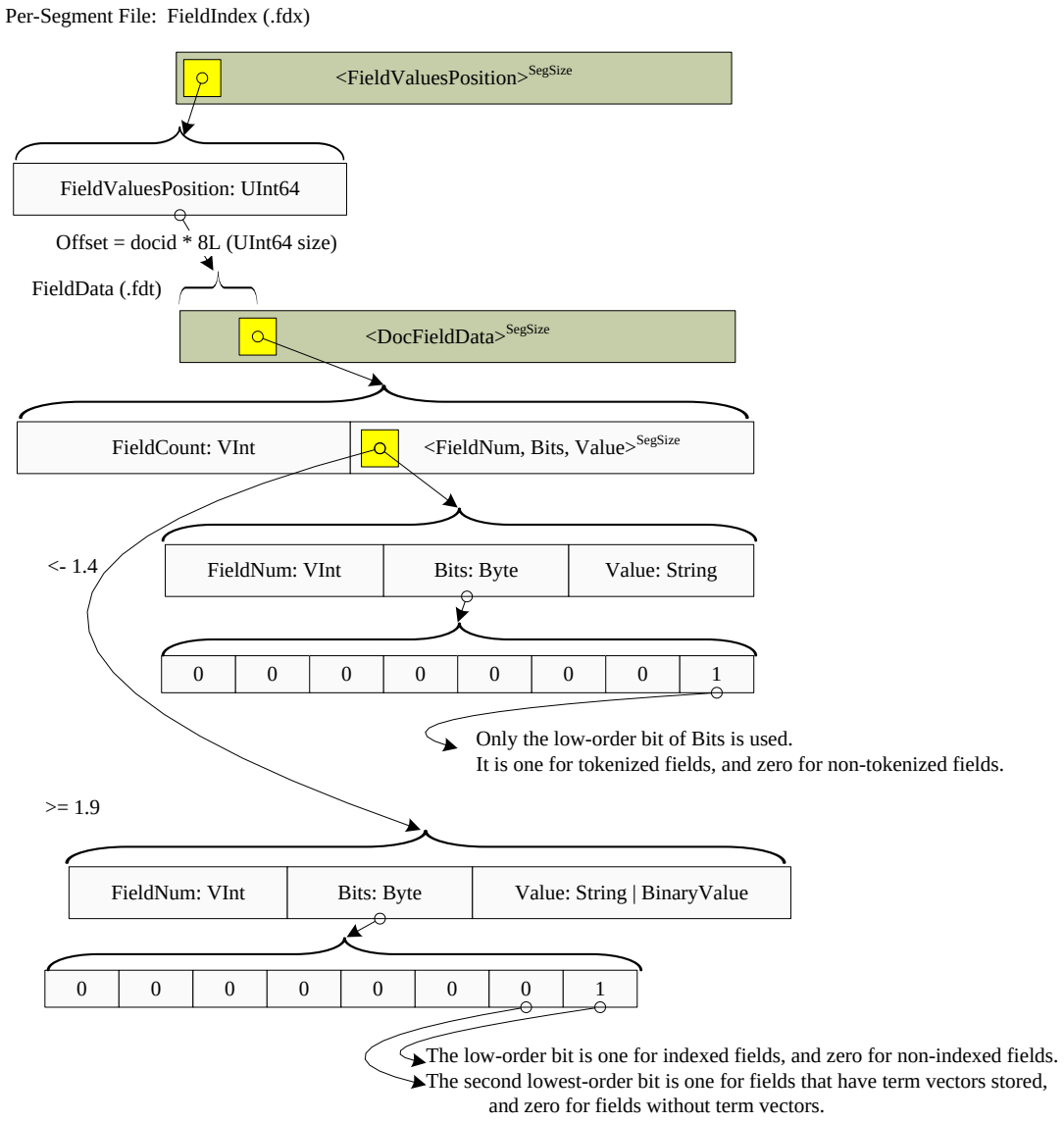
3.2.3.2 Field 数据（.fdx 和.fdt）

存储的 fields（stored fields）通过两个文件来呈现（represented by two files），即 field 索引文件（.fdx）和 field 数据文件（.fdt）。

文件	包含的项	父项	数目	类型	版本	描述
Fields Index (.fdx) 对每个文档来说，存储指向它的 fields 数据的指针（pointer）	FieldValuesPosition		SegSize	UInt64		用于找详细文档（a particular document）的所有 fields 的 field 数据文件中的位置（position），因为它包含的（contains）是固定长度的数据（fixed-length data），这个文件可以很容易地进行随机访问（randomly accessed）。
						文档 n 的 field 数据的位置是在该文件中 n*8 的位置中（UInt64 类型）。
Fields Data（.fdt）这个文件存储每个文档的 field 数据	DocFieldData		SegSize			
	FieldCount	DocFieldData	1	VInt		
	FieldNum	DocFieldData	FieldCount	VInt		
	Bits	DocFieldData	FieldCount	Byte	<=1.4	只有最低阶的 bit 位（low-order bits of Bits）被使用，值为 1 表示 tokenized field（分解过的 field），0 表示 non-tokenized field。

				Byte	>=1.9	最低阶的 bit 位表示 tokenized field
					>=1.9	第二个 bit（second bit）用于表示该 field 存储 binary 数据。
					>=1.9	第三个 bit（third bit）表示该 field 的压缩选项被开启（field with compression enabled），如果压缩选项开启，采用的压缩算法（algorithm）是 ZLIB
	Value	DocFieldData	FieldCount	String	<=1.4	
				String | BinaryValue	>=1.9	依赖于 Bits 的值
				BinaryValue	>=1.9	ValueSize,<Byte>^ValueSize
	ValueSize	Value	1	VInt	>=1.9	

结构如下图所示：



3.2.3.3 Term 字典（.tii 和.tis）

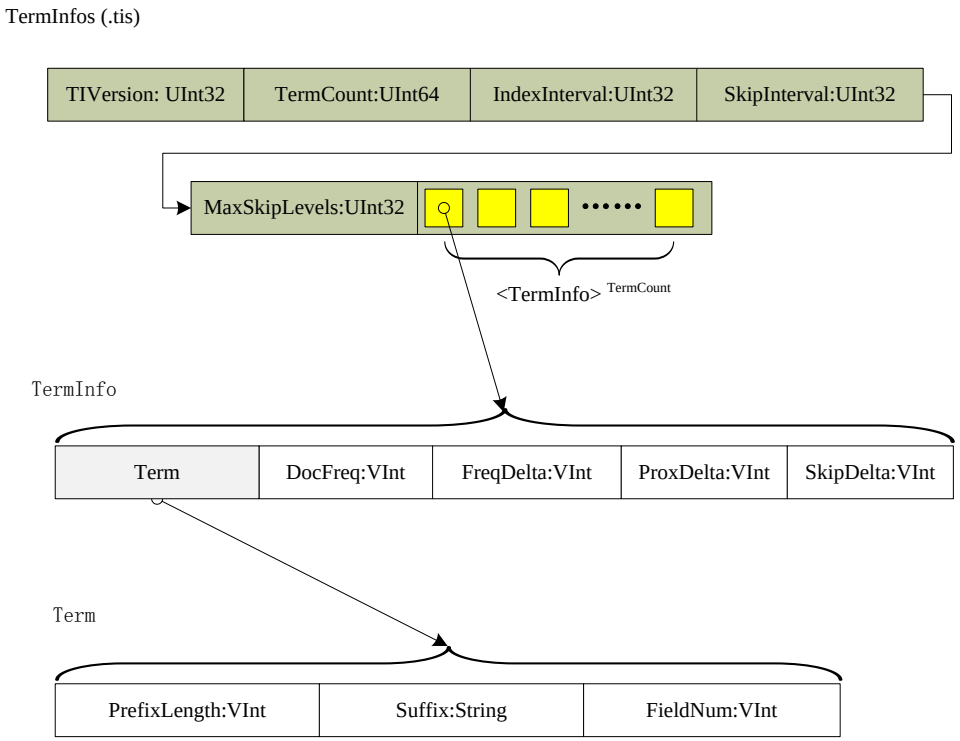
Term 字典使用如下两种文件存储，第一种是存储 term 信息（TermInfoFile）的文件，即.tis 文件，格式如下：

版本	包含的项	数目	类型	描述
全部版本	TIVersion	1	UInt32	记录该文件的版本，1.4 版本中为-2
	TermCount	1	UInt64	
	IndexInterval	1	UInt32	
	SkipInterval	1	UInt32	
	MaxSkipLevels	1	UInt32	
	TermInfos	1	TermInfo…	
	TermInfos->TermInfo	TermCount	TermInfo	
	TermInfo->Term	TermCount	Term	
	Term->PrefixLength	TermCount	VInt	Term 文本的前缀可以共享，该项的值表示根据前一个 term 的文本来初始化的字符串前缀长度，前一个 term 必须已经预设成后缀文本以便构成该 term 的文本。比如，如果前一个 term 为“bone”，而当前 term 为“boy”，则该 PrefixLength 值为 2，suffix 值为“y”
	Term->Suffix	TermCount	String	如上
	Term->FieldNum	TermCount	VInt	用来确定 term 的 field，它们存储在.fdt 文件中。
	TermInfo->DocFreq	TermCount	VInt	包含该 term 的文档数目
	TermInfo->FreqDelta	TermCount	VInt	用来确定包含在.frq 文件中该 term 的 TermFreqs 的位置。特别指出，它是该 term 的数据在文件中位置与前一个 term 的位置的差值，当为第一个 term 时，该值为 0
	TermInfo->ProxDelta	TermCount	VInt	用来确定包含在.prx 文件中该 term 的 TermPositions 的位置。特别指出，它是该 term 的数据在文件中的位置与前一个 term 的位置地差值，当为第一个 term 时，该值为 0。如果 fields 的 omitTF 设置为 true，该值也为 0，因为 prox 信息没有被存储。
	TermInfo->SkipDelta	TermCount	VInt	用来确定包含在.frq 文件中该 term 的 SkipData 的位置。特别指出，它是 TermFreqs 之后即 SkipData 开始的字节数目，换句话说，它是 TermFreq 的长度。

				SkipDelta 只有在 DocFreq 不比 SkipInteval 小的情况下才会存储。

TermInfoFile 文件按照 Term 来排序，排序方法首先按照 Term 的 field 名称（按照 UTF-16 字符编码）排序，然后按照 Term 的 Text 字符串（UTF-16 编码）排序。

结构如下图所示：

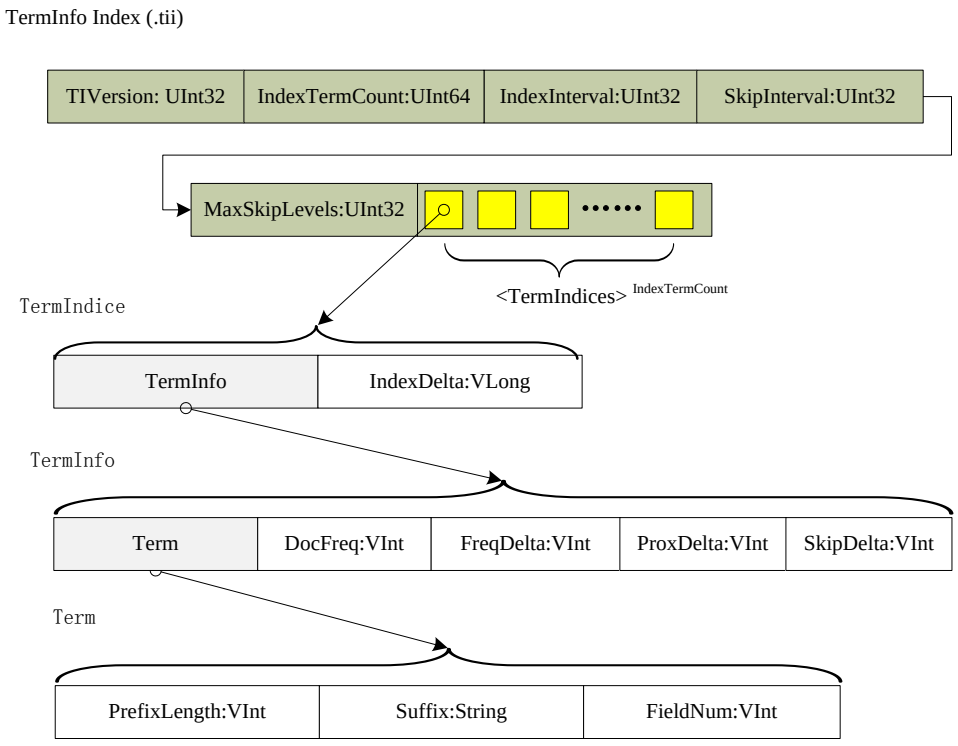


另一种是存储 term 信息的索引文件，即.tii 文件，该文件包含.tis 文件中每一个 IndexInterval 的值，与它在.tis 中的位置一起被存储，这被设计来完全地读进内存中（read entirely into memory），以便用来提供随机访问.tis 文件。该文件的结构与.tis 文件非常相似，只是添加了一项数据，即 IndexDelta。格式如下

版本	包含的项	数目	类型	描述
全部版本	TIVersion	1	UInt32	同 tis
	IndexTermCount	1	UInt64	同 tis
	IndexInterval	1	UInt32	同 tis
	SkipInterval	1	UInt32	是 TermDocs 存储在 skip 表中的分数（fraction），用来加速（accelerable）TermDocs.skipTo(int)的调用。在更小的索引中获得更大的结果值（larger values result），将获得更高的速度，但却更小开销？（fewer accelerable cases）。but fewer accelerable cases, while smaller values result in bigger

				indexes, less acceleration (in case of a small value for MaxSkipLevels) and more accelerable cases.
	MaxSkipLevels	1	UInt32	是.frq 文件中为每一个 term 存储的 skip levels 的最大数目，A low value results in smaller indexes but less acceleration, a larger value results in slighly larger indexes but greater acceleration.参见.frq 文件格式中关于 skip levels 的详细介绍。
	TermIndices	IndexTermCount	TermIndice	同 tis
	TermIndice->TermInfo	IndexTermCount	TermInfo	同 tis
	TermIndice->IndexDelta	IndexTermCount	VLong	用来确定该 Term 的 TermInfo 在.tis 文件中的位置，特别指出，它是该 term 的数据的位置与前一个 term 位置的差值。

结构如下图所示：

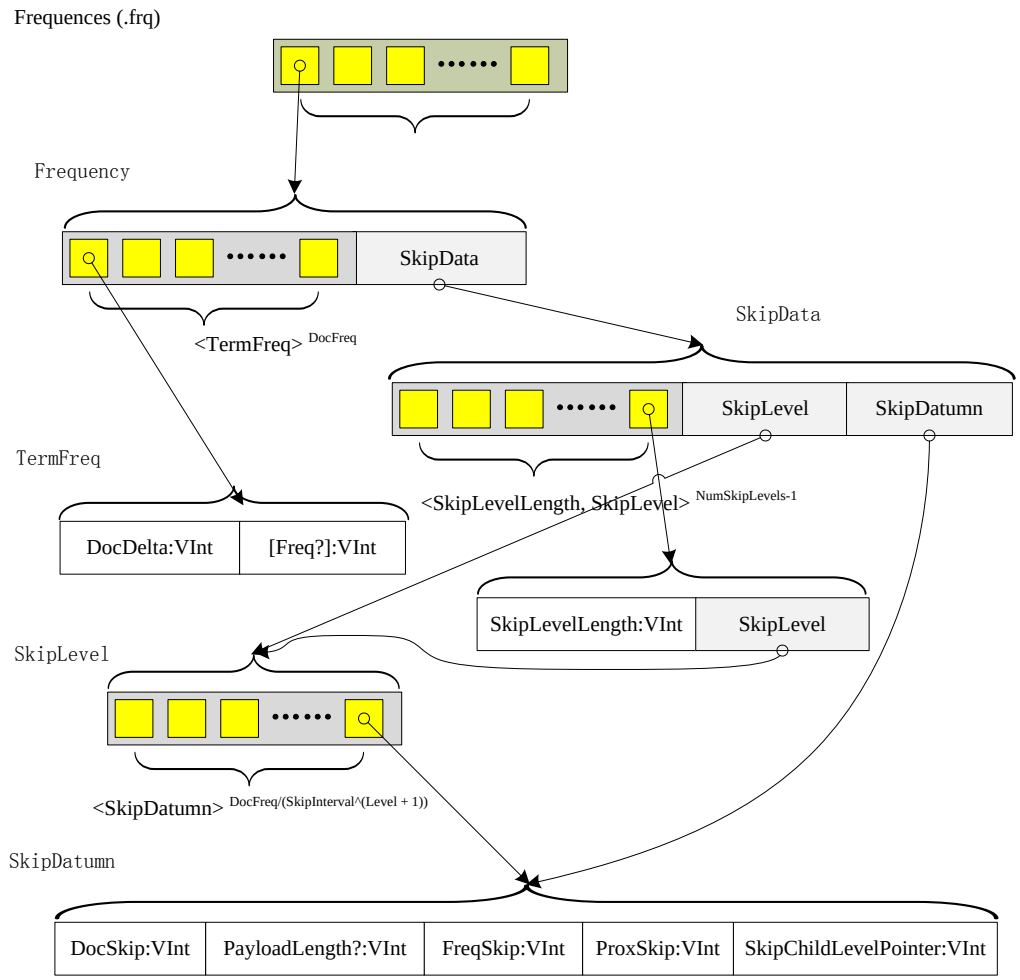


3.2.3.4 Term 频率数据（.frq）

Term 频率数据文件（.frq 文件）存储容纳了每一个 term 的文档列表，以及该 term 出现在该文档中的频率（出现次数 frequency，如果 omitTf 设置为 fals 时才存储）。

版本	包含的项	数目	类型	描述
全部版本	TermFreqs	TermCount	TermFreq	按照 term 顺序排序，term 是隐含的(?implicit)，来自.tis 文件。TermFreq 按文档编号递增的顺序排序。
	SkipData	TermCount	SkipData	
	TermFreq->DocDelta	TermCount	VInt	如果 omitTf 设置为 false，要同时检测文档编号和频率，特别指出，DocDelta/2 时该文档编号与上一个文档编号的差值（如果是第一个文档值为 0）。当 DocDelta 为单数时频率为 1，当 DocDelta 为偶数时频率为读取下一个 VInt 的值。如果 omitTf 设置为 true，DocDelta 为文档编号之间的差值（gap，不用乘以 2，multiplited），频率信息则不被存储。
	TermFreq->[Freq?]	TermCount	VInt	
	SkipData->SkipLevelLength	NumSkipLevels-1	VInt	
	SkipData->SkipLevel	TermCount	SkipDatums	
	SkipLevel->SkipDatum	DocFreq/(SkipInterval^(Level + 1))	SkipDatum	
	SkipData->SkipDatum	TermCount	SkipDatum	
	SkipDatum->DocSkip	1	VInt	
	SkipDatum->PayloadLength?	1	VInt	
	SkipDatum->FreqSkip	1	VInt	
	SkipDatum->ProxSkip	1	VInt	
	SkipDatum->SkipChildLevelPointer?	1	VLong	

结构如下图所示：



举例来说，当 omitTf 设置为 false 时，一个 term 的 TermFreqs 在文档 7 出现 1 次并且在文档 11 中出现 3 次，则为如下的 VInt 数字序列：

15, 8, 3

如果 omitTf 设置为 true 时，则为如下数字序列：

7, 4

DocSkip 记录在 TermFreqs 中每隔 SkipInterval 个文档之前的文档编号。如果该 term 的域 fields 中被禁用 payloads 时，则 DocSkip 呈现在序列中（in the sequence）与上一个值之间的差值（difference）。如果 payloads 启用时，则 DocSkip/2 表示序列中与上一个值之间的差值。如果 payloads 启用并且 DocSkip 为奇数时，PayloadLength 将被存储并表示（indicating）在 TermPositions 中第 SkipInterval 个文档之前的最后一个 payload 的长度。FreqSkip 和 ProxSkip 分别(respectively)记录在 FreqFile 和 ProxFile 文件中每 SkipInterval 个记录(entry)的位置。文件的位置信息对序列中前一个 SkipDatumn 来说与 TermFreqs 和 Positions 的起始信息相关。

例如，如果 DocFreq=35 并且 SkipInterval=16，则在 TermFreqs 中有两个 SkipData 记录，容纳第 15 和第 31 个文档编号。第一个 FreqSkip 代表第 16 个 SkipDatumn 起始的 TermFreqs 数据开始之后的字节数目，第二个

FreqSkip 表示第 32 个 SkipDatumn 开始之后的字节数目。第一个 ProxSkip 代表第 16 个 SkipDatumn 起始的 Positions 数据开始之后的字节数目，第二个 ProxSkip 表示第 32 个 SkipDatumn 开始之后的字节数目。

在 Lucene 2.2 版本中介绍了 skip levels 的想法（notion），每一个 term 可以有多个 skip levels。一个 term 的 skip levels 的数目等于 NumSkipLevels = Min(MaxSkipLevels, floor(log(DocFreq/log(SkipInterval))))。对一个 skip level 来说 SkipData 记录的数目等于 DocFreq/(SkipInterval^(Level + 1))。然而（whereas）最低的（lowest）skip level 等于 Level = 0。

例如假设 SkipInterval = 4, MaxSkipLevels = 2, DocFreq = 35，则 skip level 0 有 8 个 SkipData 记录，在 TermFreqs 序列中包含第 3、7、11、15、19、23、27 和 31 个文档的编号。Skip level 1 则有 2 个 SkipData 记录，在 TermFreqs 中包含了第 15 和第 31 个文档的编号。

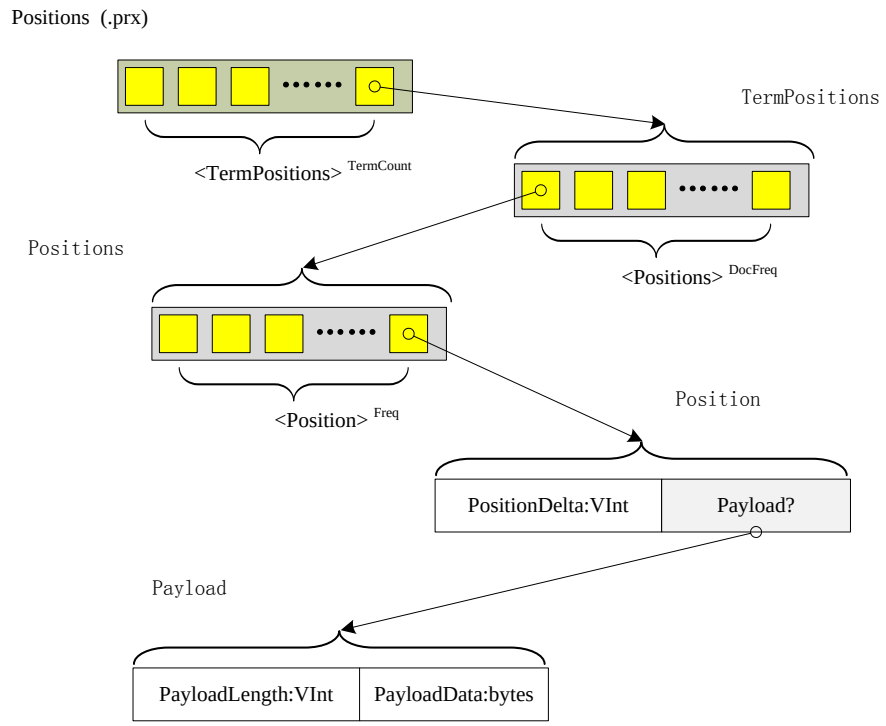
在所有 level>0 之上的 SkipData 记录中包含一个 SkipChildLevelPointer，指向（referencing）level-1 中相应）（corresponding）的 SkipData 记录。在这个例子中，level 1 中的记录 15 有一个指针指向 level 0 中的记录 15，level 1 中的记录 31 有一个指针指向 level 0 中的记录 31。

3.2.3.5 Positions 位置信息数据（.prx）

Positions 位置信息数据文件（.prx 文件）容纳了每一个 term 出现在所有文档中的位置的列表。注意如果在 fields 中的 omitTf 设置为 true 时将不会在此文件中存储任何信息，并且如果索引中所有 fields 中的 omitTf 都设置为 true，此.prx 文件将不会存在。

版本	包含的项	数目	类型	描述
全部版本	TermPositions	TermCount	TermPositions	按照 term 顺序排序，term 是隐含的（?implicit），来自.tis 文件。
	TermPositions->Positions	DocFreq	Positions	按文档编号递增的顺序排序。
	Positions->PositionDelta	Freq	VInt	如果 term 的 fields 中 payloads 被禁用，则取值为 term 出现在该文档中当前位置与前一个位置的差值（第一个位置取值 0）。如果 payloads 被启用，则取值为当前位置与上一个位置之间差值的 2 倍。如果 payloads 启用并且 PositionDelta 为单数，则 PayloadLength 被存储，表示当前位置的 payloads 的长度。
	Positions->Payload?	Freq	Payload	
	Payload->PayloadLength?	1	VInt	
	Payload->PayloadData	PayloadLength	byte	

结构如下图所示：



例如，如果一个 term 的 TermPositions 为一个文档中出现的第 4 个 term，并且为后来的文档（subsequent document）中出现的第 5 个和第 9 个 term，则将被存储为下面的 VInt 数据序列（payloads 禁用）：

4, 5, 4

PayloadData 是与 term 的当前位置相关联元数据（metadata），如果该位置的 PayloadLength 被存储，则它表示此 payload 的长度。如果 PayloadLength 没存储，则此 payload 与前一个位置的 payload 拥有相等的 PayloadLength。

3.2.3.6 Norms 调节因子文件（.nrm）

在 Lucene 2.1 版本之前，每一个索引都有一个 norm 文件给每一个文档都保存了一个字节。对每一个文档来说，那些.f[0-9]*包含了一个字节容纳一个被编码的分数，值为对 hits 结果集来说在那个 field 中被相乘得出的分数（multiplied into the score）。每一个分离的 norm 文件在适当的时候（when adequate）为复合的（compound）和非复合的 segment 片断创建，格式如下：

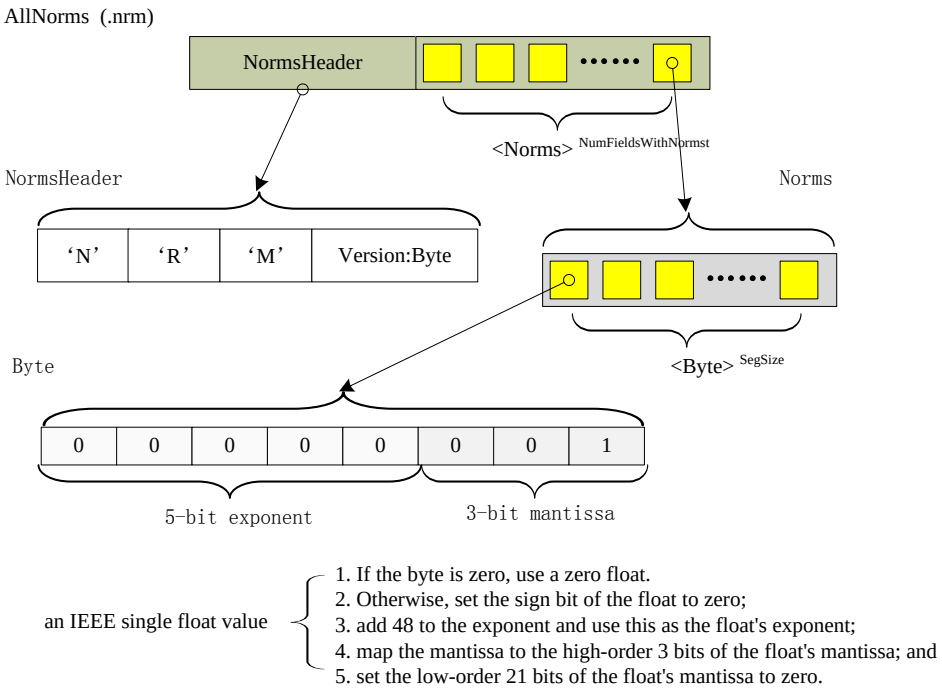
Norms (.f[0-9]*) --> <Byte> SegSize

在 Lucene 2.1 及以上版本，只有一个 norm 文件容纳了所有 norms 数据：

版本	包含的项	数目	类型	描述

2.1 及之后版本	NormsHeader	1	raw	'N','R','M',Version：4 个字节，最后字节表示该文件的格式版本，当前为-1
	Norms	NumFieldsWithNorms	Norms	
	Norms->Byte	SegSize	Byte	每一个字节编码了一个 float 指针数值，bits 0-2 容纳 3-bit 尾数（mantissa），bits 3-8 容纳 5-bit 指数（exponent），这些被转换成一个 IEEE 单独的 float 数值，如图所示
	NormsHeader->Version	1	Byte	

结构如下图所示：



一个分离的 norm 文件在一个存在的 segment 的 norm 数据被更改的时候被创建，当 field N 被修改时，一个分离的 norm 文件.sN 被创建，用来维护该 field 的 norm 数据。

3.2.3.7 Term 向量文件

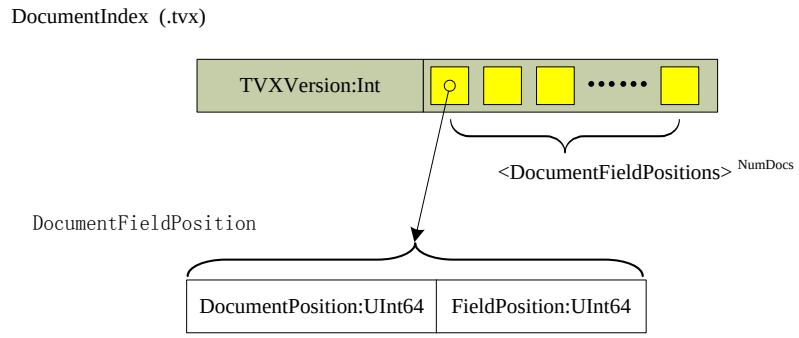
Term 向量（vector）的支持是 field 基本组成中对一个 field 来说的可选项，它包含如下 4 种文件：

1. 文档索引或.tvx 文件：对每个文档来说，它把偏移（offset）存储进文档数据（.tvd）文件和域 field 数据（.tvf）文件

版本	包含的项	数目	类型	描述

全部版本	TVXVersion	1	Int	在 Lucene 2.4 中为 3 (TermVectorsReader.FORMAT_VERSION2)
	DocumentPosition	NumDocs	UInt64	在.tvd 文件中的偏移
	FieldPosition	NumDocs	UInt64	在.tvf 文件中的偏移

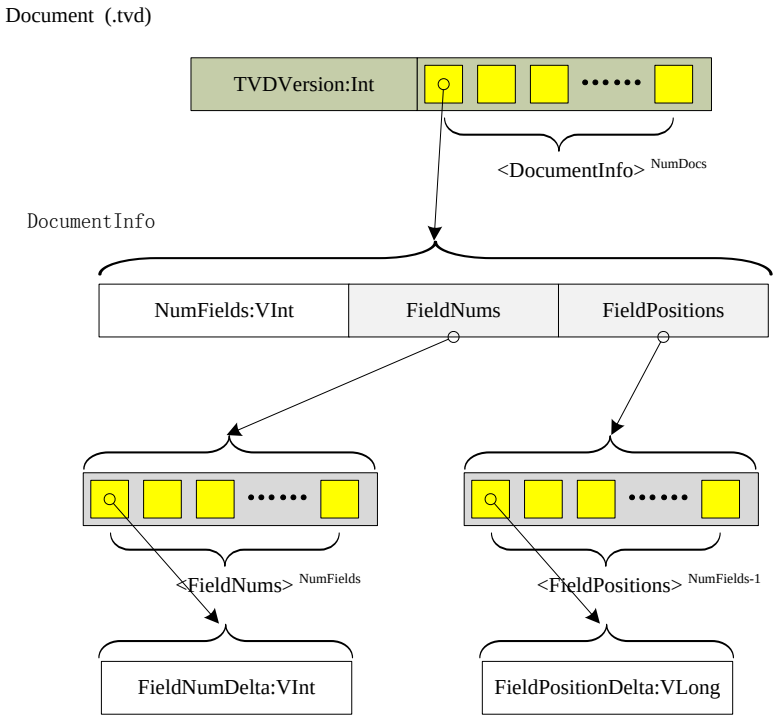
结构如下图所示：



2. 文档或.tvd 文件：对每个文档来说，它包含 fields 的数目，有 term 向量的 fields 的列表，还有指向 term 向量域文件(.tvf)中的域信息的指针列表。该文件用于映射(map out)出那些存储了 term 向量的 fields，以及这些 field 信息在.tvf 文件中的位置。

版本	包含的项	数目	类型	描述
全部版本	TVDVersion	1	Int	在 Lucene 2.4 中为 3 (TermVectorsReader.FORMAT_VERSION2)
	NumFields	NumDocs	VInt	
	FieldNums	NumDocs	FieldNums	
	FieldNums->FieldNumDelta	NumFields	VInt	
	FieldPositions	NumDocs	FieldPositions	
	FieldPositions->FieldPositionDelta	NumField-1	VLong	

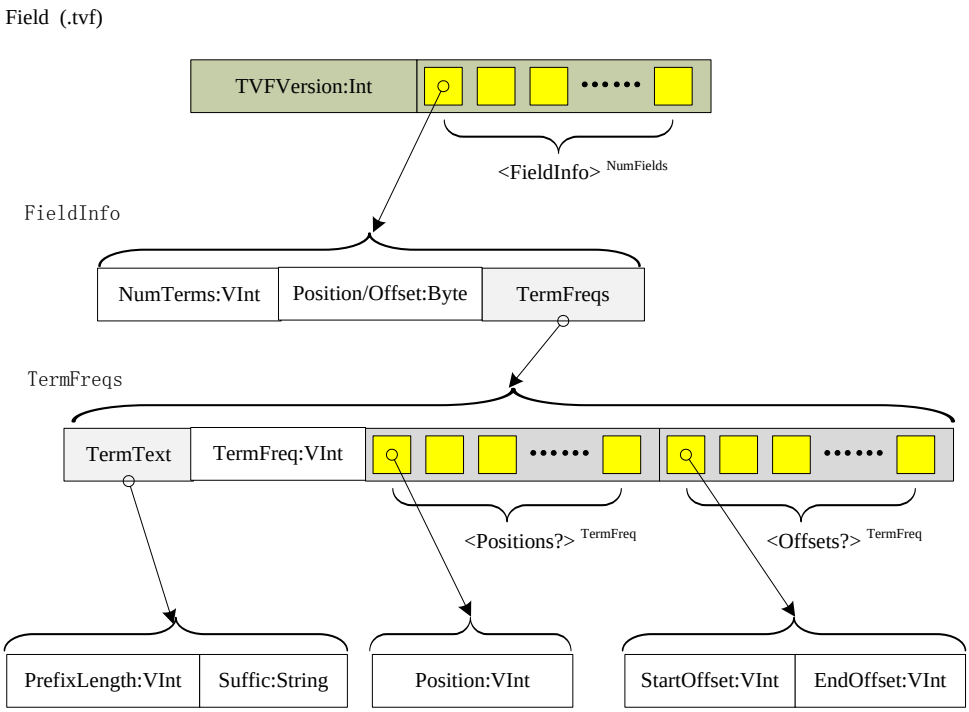
结构如下图所示：



3. 域 field 或.tvf 文件：对每个存储了 term 向量的 field 来说，该文件包含了一个 term 的列表，及它们的频率，还有可选的位置和偏移信息。

版本	包含的项	数目	类型	描述
全部版本	TVFVersion	1	Int	在 Lucene 2.4 中为 3 (TermVectorsReader.FORMAT_VERSION2)
	NumTerms	NumFields	VInt	
	Position/Offset	NumFields	Byte	
	TermFreqs	NumFields	TermFreqs	
	TermFreqs->TermText	NumTerms	TermText	
	TermText->PrefixLength	NumTerms	VInt	
	TermText->Suffix	NumTerms	String	
	TermFreqs->TermFreq	NumTerms	VInt	
	TermFreqs->Positions?	NumTerms	Positions	
	Positions->Position	TermFreq	VInt	
	TermFreqs->Offsets?	NumTerms	Offsets	
	Offsets->StartOffset	TermFreq	VInt	
	Offsets->EndOffset	TermFreq	VInt	

结构如下图所示：



备注：

- Positions/Offsets 字节存储的条件是当该 term 向量含有存储的位置或偏移信息时。
- Term Text prefixes 文本前缀是共享的，表示根据前一个 term 的文本来初始化的字符串前缀长度，前一个 term 必须已经预设成后缀文本以便构成该 term 的文本。比如，如果前一个 term 为“bone”，而当前 term 为“boy”，则该 PrefixLength 值为 2，suffix 值为“y”。
- Positions 存储为 Delta 编码的 VInts，意思是我们只能存储当前位置与最后位置的差值。
- Offsets 存储为 Delta 编码的 VInts，第一个 VInt 是 startOffset，第二个 VInt 是 endOffset。

3.2.3.8 删除的文档 （.del）

删除的文档（.del）文件是可选的，而且仅当一个 segment 存在有被删除的文档时才存在。即使对每一单个 segment，它也是维护复合 segment 的外部数据（exterior）。

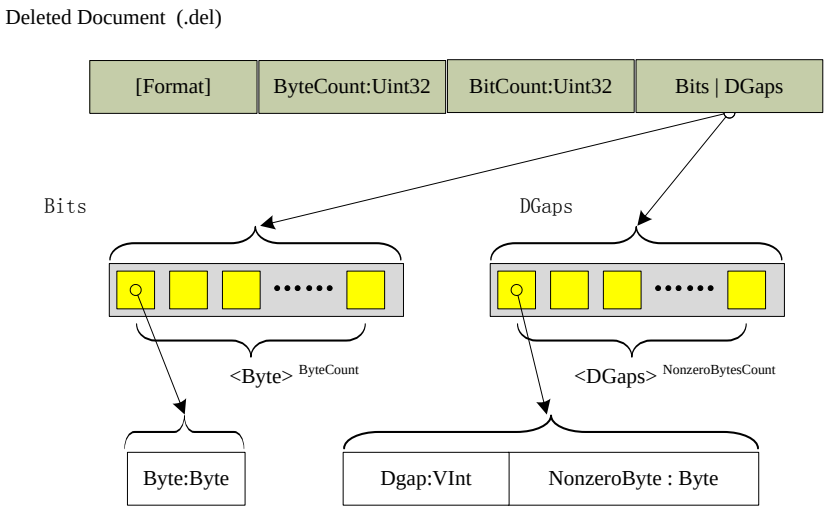
对 Lucene 2.1 及以前版本，它的格式为：Deletions (.del) --> ByteCount,BitCount,Bits

对 2.2 及以上版本，格式如下：

版本	包含的项	数目	类型	描述
2.2 之后版本	[Format]	1	UInt32	可选，-1 表示为 DGaps，非负数（negative）值表示为 Bit，并且此时不存储 Format
	ByteCount	1	UInt32	代表 Bit 里的字节数目，而且一般值为

				(SegSize/8)+1
	BitCount	1	UInt32	表示 Bit 里当前设置的字节数目
	Bit|DGaps	1		Bit 还是 DGaps 取决于 Format。Bits 中对每一个索引的文档均包含一个字节，当一个 bit 对应的一个文档编号被设置时，表示该文档被删除。Bit 从最低（least）到最重要（significant）的文档排序。所以 Bits 包含两个字节，0x00 和 0x02，则文档 9 被标记为删除。DGaps 表示松散（sparse）的 bit-vector 向量比 Bits 更有效率（efficiently）。DGaps 由索引中非 0 的 Bits 位生成，以及非 0 的字节数据本身。Bits 中非 0 字节数目（NonzeroBytesCoun）不会存储。
	Bit->Byte	ByteCount	Byte	
	DGaps->DGap	NonzeroBytesCount	VInt	
	DGaps-> NonzeroBytes	NonzeroBytesCount	Byte	

结构如下图所示：



举例来说，如果有 8000 bits，并且只有 bits 10, 12, 32 被设置，DGaps 将会存储如下数据：

(VInt) 1 , (byte) 20 , (VInt) 3 , (Byte) 1

3.3 局限性（Limitations）

有几个地方这些文件格式会让 terms 和文档的最大数目受限于 32-bit 的大小，大约最大 40 亿。这在今天不是一个问题，长远来看（in the long term）可能会成为个问题。因此它们应该替换为 UInt64 类型或者更好的类

型，如 VInt 则没有大小限制。

4 索引是如何创建的

为了使用 Lucene 来索引数据，首先你得把它转换成一个纯文本（plain-text）tokens 的数据流（stream），并通过它创建出 Document 对象，其包含的 Fields 成员容纳这些文本数据。一旦你准备好些 Document 对象，你就可以调用 IndexWriter 类的 addDocument(Document)方法来传递这些对象到 Lucene 并写入索引中。当你做这些的时候，Lucene 首先分析（analyzer）这些数据来使得它们更适合索引。详见《Lucene In Action》

4.1 索引创建示例

下面的代码示例如何给一个文件建立索引。

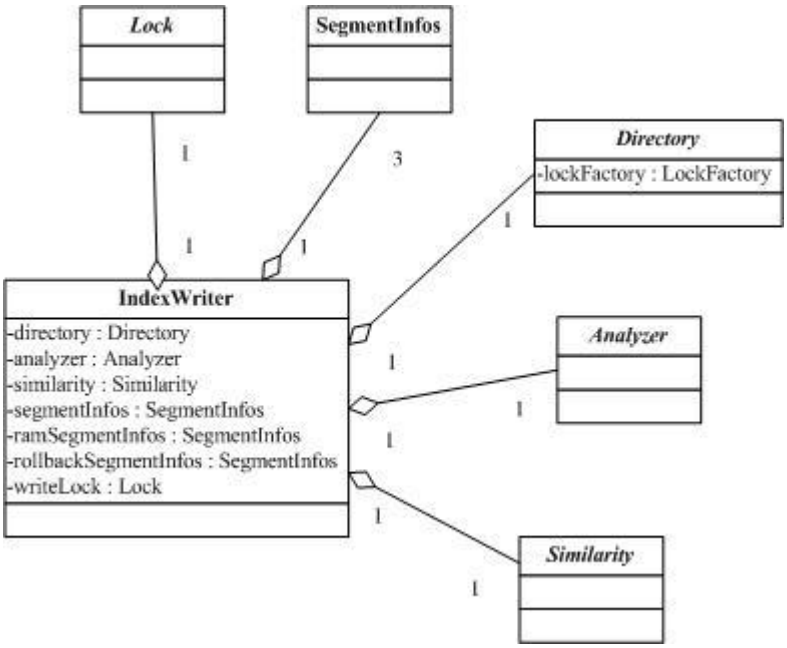
```
    // Store the index on disk
    Directory directory = FSDirectory.getDirectory("/tmp/testindex");
    // Use standard analyzer
    Analyzer analyzer = new StandardAnalyzer();
    // Create IndexWriter object
    IndexWriter iwriter = new IndexWriter(directory, analyzer, true);
    iwriter.setMaxFieldLength(25000);
    // make a new, empty document
    Document doc = new Document();
    File f = new File("/tmp/test.txt");
    // Add the path of the file as a field named "path".  Use a field that is
    // indexed (i.e. searchable), but don't tokenize the field into words.
    doc.add(new Field("path", f.getPath(), Field.Store.YES,
Field.Index.UN_TOKENIZED));
    String text = "This is the text to be indexed.";
    doc.add(new Field("fieldname", text, Field.Store.YES,
Field.Index.TOKENIZED));
    // Add the last modified date of the file a field named "modified".  Use
    // a field that is indexed (i.e. searchable), but don't tokenize the field
    // into words.
    doc.add(new Field("modified",
        DateTools.timeToString(f.lastModified(), DateTools.Resolution.MINUTE),
        Field.Store.YES, Field.Index.UN_TOKENIZED));
    // Add the contents of the file to a field named "contents".  Specify a Reader,
    // so that the text of the file is tokenized and indexed, but not stored.
    // Note that FileReader expects the file to be in the system's default encoding.
    // If that's not the case searching for special characters will fail.
    doc.add(new Field("contents", new FileReader(f)));
    iwriter.addDocument(doc);
    iwriter.optimize();
    iwriter.close();
```

4.2 索引创建类 IndexWriter

一个 IndexWriter 对象创建并且维护(maintains) 一条索引并生成 segment，使用 DocumentsWriter 类来建立多个文档的索引数据，SegmentMerger 类负责合并多个 segment。

4.2.1 org.apache.lucene.index.IndexWriter

IndexWriter 通过指定存放的目录（Directory）以及文档分析器（Analyzer）来构建，direcotry 代表索引存储（resides）在哪里；analyzer 表示如何来分析文档的内容；similarity 用来规格化（normalize）文档，给文档算分（scoring）；IndexWriter 类里还有一些 SegmentInfos 对象用于存储索引片段信息，以及发生故障回滚等。以下是它们的类图：



它的构造函数(constructor)的 create 参数(argument)确定(determines)是否一条新的索引将被创建，或者是否一条已经存在的索引将被打开。需要注意的是你可以使用 create=true 参数打开一条索引，即使有其他 readers 也在在使用这条索引。旧的 readers 将继续检索它们已经打开的”point in time”快照（snapshot），并不能看见那些新已创建的索引，直到它们再次打开（re-open）。另外还有一个没有 create 参数的构造函数，如果提供的目录（provided path）中没有已经存在的索引，它将创建它，否则将打开此存在的索引。

另一方面（in either case），添加文档使用 addDocument()方法，删除文档使用 deleteDocuments(Term)或者 deleteDocuments(Query)方法，而且一篇文档可以使用 updateDocument()方法来更新（仅仅是先执行 delete 在执行 add 操作而已）。当完成了添加、删除、更新文档，应该需要调用 close 方法。

这些修改会缓存在内存中（buffered in memory），并且定期地（periodically）刷新到（flush）Directory 中（在上述方法的调用期间）。一次 flush 操作会在如下时候触发（triggered）：当从上一次 flush 操作后有足够多缓存的 delete 操作（参见 setMaxBufferedDeleteTerms(int)），或者足够多已添加的文档（参见 setMaxBufferedDocs(int)），无论哪个更快些（whichever is sooner）。对被添加的文档来说，一次 flush 会在如下任何一种情况下触发，文档的 RAM 缓存使用率（setRAMBufferSizeMB）或者已添加的文档数目，缺省的 RAM 最高使用率是 16M，为得到索引的最高效率，你需要使用更大的 RAM 缓存大小。需要注意的是，flush 处理仅仅是将 IndexWriter 中内部缓存的状态(internal buffered state)移动进索引里去，但是这些改变不会让 IndexReader 见到，直到 commit()和 close()中的任何一个方法被调用时。一次 flush 可能触发一个或更多的片断合并（segment

merges），这时会启动一个后台的线程来处理，所以不会中断 addDocument 的调用，请参考 MergeScheduler。

构造函数中的可选参数（optional argument）autoCommit 控制（controls）修改对 IndexReader 实体（instance）读取相同索引的能见度（visibility）。当设置为 false 时，修改操作将不可见（visible）直到 close()方法被调用后。需要注意的是修改将依然被 flush 进 Directory，就像新文件一样（as new files），但是却不会被提交（commit）（没有新的引用那些新文件的 segments_N 文件会被写入（written referencing the new files））直道 close()方法被调用。如果在调用 close()之前发生了某种严重错误（something goes terribly wrong）（例如 JVM 崩溃了），于是索引将反映（reflect）没有任何修改发生过（none of changes made）（它将保留它开始的状态（remain in its starting state））。你还可以调用 rollback()，这样可以关闭那些没有提交任何修改操作的 writers，并且清除所有那些已经 flush 但是现在不被引用的（unreferenced）索引文件。这个模式（mode）对防止（prevent）readers 在一个错误的时间重新刷新（refresh）非常有用（例如在你完成所有 delete 操作后，但是在你完成添加操作前的时候）。它还能被用来实现简单的 single-writer 的事务语义（transactional semantics）（"all or none"）。你还可以执行两条语句（two-phase）的 commit，通过调用 prepareCommit()方法，之后再调用 commit()方法。这在 Lucene 与外部资源（例如数据库）交互的时候是很需要的，而且必须执行 commit 或 rollback 该事务。

当 autoCommit 设为 true 的时候，该 writer 会周期性地提交它自己的数据。**已过时**：注意在 3.0 版本中，IndexWriter 将不会接收 autoCommit=true，它会硬设置（hardwired）为 false。你可以自己在需要的时候经常调用 commit()方法。这不保证什么时候一个确定的 commit 会处理。它被曾经用来在每次 flush 的时候处理，但是现在会在每次完成 merge 操作后处理，如 2.4 版本中即如此。如果你想强行执行 commit，请调用 commit 方法或者 close 这个 writer。一旦一个 commit 完成后，新打开的 IndexReader 实例将会看到索引中该 commit 更改的数据。当以这种模式运行时，当优化（optimize）或者片断合并（segment merges）正在进行（take place）的时候需要小心地重新刷新（refresh）你的 readers，因为这两个操作会绑定（tie up）可观的（substantial）磁盘空间。

不管（Regardless）autoCommit 参数如何，一个 IndexReader 或者 IndexSearcher 只会看到索引在它打开的当时的状态。任何在索引被打开之后提交到索引中的 commit 信息，在它被重新打开之前都不会见到。

当一条索引暂时（for a while）将不会有更多的文档被添加，并且期望（desired）得到最理想（optimal）的检索性能（performance），于是 optimize()方法应该在索引被关闭之前被调用。

打开 IndexWriter 会为使用的 Directory 创建一个 lock 文件。尝试对相同的 Directory 打开另一个 IndexWriter 将会导致（lead to）一个 LockObtainFailedException 异常。如果一个建立在相同的 Directory 的 IndexReader 对象被用来从这条索引中删除文档的时候，这个异常也会被抛出。

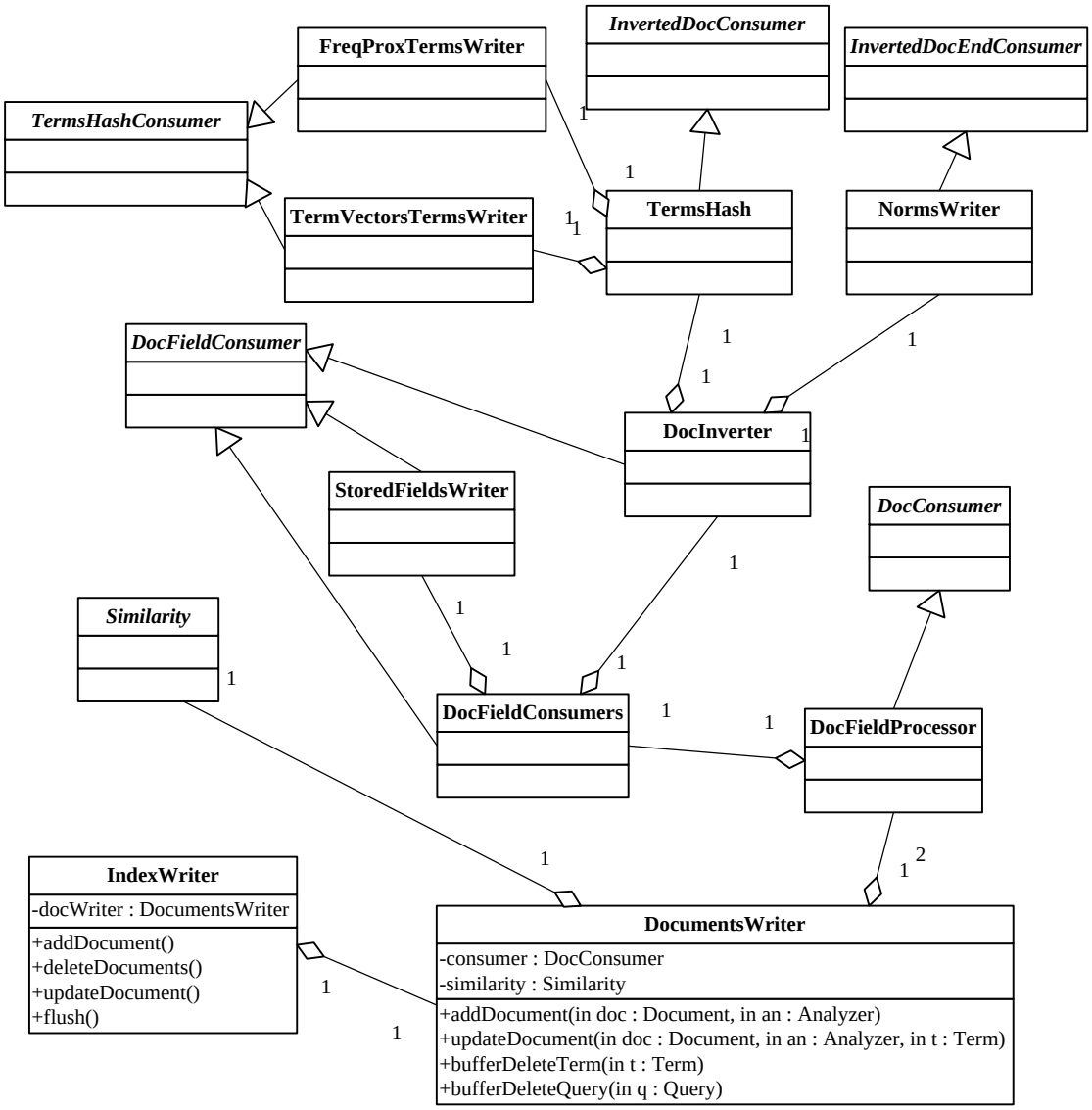
专家（Expert）：IndexWriter 允许指定（specify）一个可选的（optional）IndexDeletionPolicy 实现。你可以通过这个控制什么时候优先的提交（prior commit）从索引中被删除。缺省的策略（policy）是 KeepOnlyLastCommitDeletionPolicy 类，在一个新的提交完成的时候它会马上所有的优先提交（prior commit）（这匹配 2.2 版本之前的行为）。创建你自己的策略能够允许你明确地（explicitly）保留以前的”point in time”提交（commit）在索引中存在（alive）一段时间。为了让 readers 刷新到新的提交，在它们之下没有被删除的旧的提交（without having the old commit deleted out from under them）。这对那些不支持“在最后关闭时才删除”语义（”delete on last close” semantics）的文件系统（filesystem）如 NFS，而这是 Lucene 的“point in time”检索通常所依赖的（normally rely on）。

专家（Expert）：IndexWriter 允许你分别修改 MergePolicy 和 MergeScheduler。MergePolicy 会在该索引中的

segment 有更改的任何时候被调用。它的角色是选择哪一个 merge 来做，如果有（if any）则传回一个 MergePolicy.MergeSpecificatio 来描述这些 merges。它还会选择 merges 来为 optimize()做处理，缺省是 LogByteSizeMergePolicy。然后 MergeScheduler 会通过传递这些 merges 来被调用，并且它决定什么时候和怎么样来执行这些 merges 处理，缺省是 ConcurrentMergeScheduler。

4.2.2 org.apache.lucene.index.DocumentsWriter

DocumentsWriter 是由 IndexWriter 调用来负责处理多个文档的类，它通过与 Directory 类及 Analyzer 类、Scorer 类等将文档内容提取出来，并分解成一组 term 列表再生成一个单一的 segment 所需要的数据文件，如 term 频率、term 位置、term 向量等索引文件，以便 SegmentMerger 将它合并到统一的 segment 中去。以下是它的类图：



该类可接收多个添加的文档，并且直接写成一个单独的 segment 文件。这比为每一个文档创建一个 segment（使用 DocumentWriter）以及对那些 segments 执行合作处理更有效率。

每一个添加的文档都被传递给 DocConsumer 类，它处理该文档并且与索引链表中（indexing chain）其它的 consumers 相互发生作用（interacts with）。确定的 consumers，就像 StoredFieldWriter 和 TermVectorsTermsWriter，提取一个文档的摘要（digest），并且马上把字节写入“文档存储”文件（比如它们不为每一个文档消耗（consume）内存 RAM，除了当它们正在处理文档的时候）。

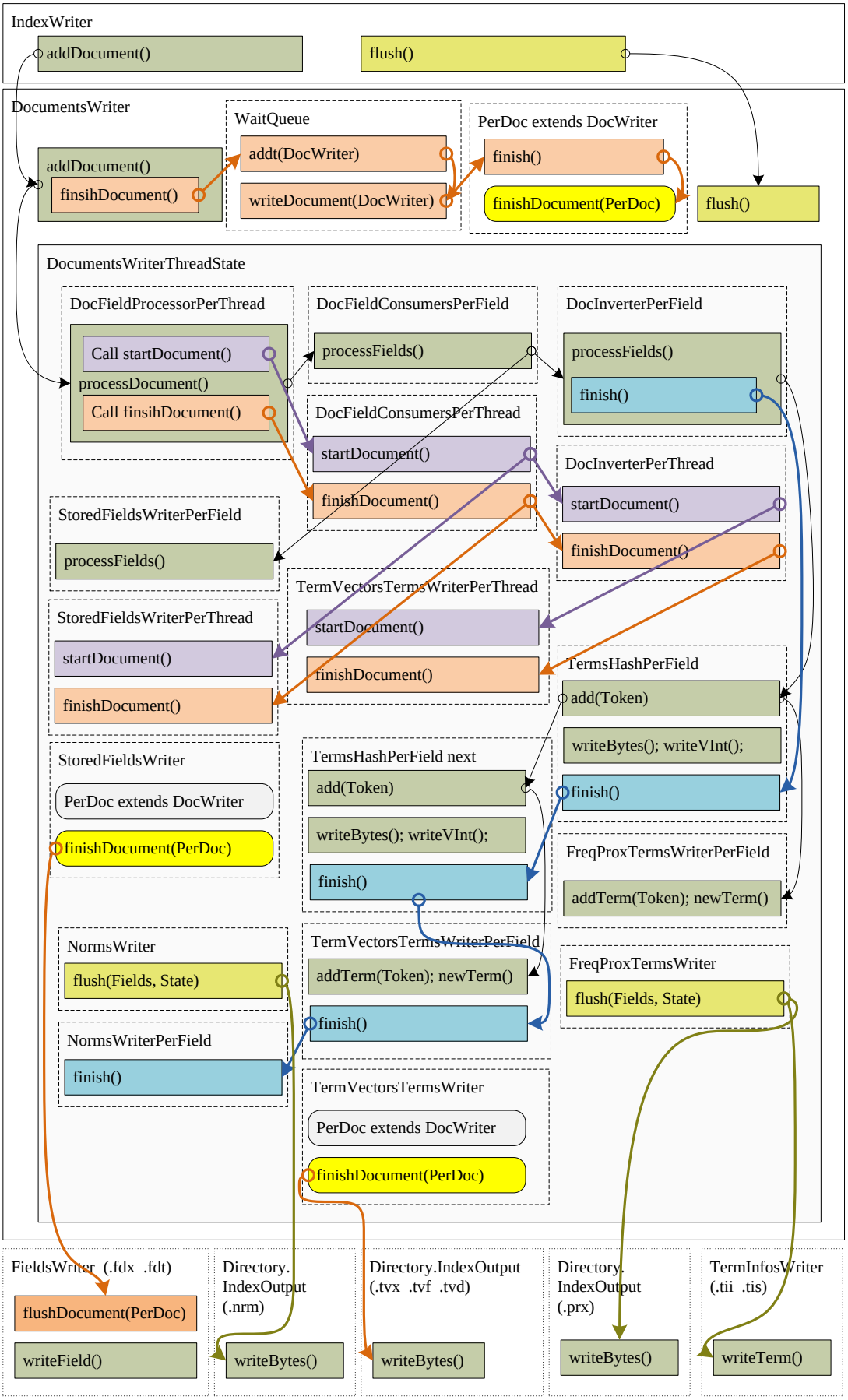
其它的 consumers，比如 FreqProxTermsWriter 和 NormsWriter，会缓存字节在内存中，只有当一个新的 segment 制造出的时候才会 flush 到磁盘中。

一旦使用完我们分配的 RAM 缓存，或者已添加的文档数目足够多的时候（这时候是根据添加的文档数目而不是 RAM 的使用率来确定是否 flush），我们将创建一个真实的 segment，并将它写入 Directory 中去。

4.3 索引创建过程

文档的索引过程是通过 DocumentsWriter 的内部数据处理链完成的，DocumentsWriter 可以实现同时添加多个文档并将它们写入一个临时的 segment 中，完成后再由 IndexWriter 和 SegmentMerger 合并到统一的 segment 中去。DocumentsWriter 支持多线程处理，即多个线程同时添加文档，它会为每个请求分配一个 DocumentsWriterThreadState 对象来监控此处理过程。处理时通过 DocumentsWriter 初始化时建立的 DocFieldProcessor 管理的索引处理链来完成的，依次处理为 DocFieldConsumers、DocInverter、TermsHash、FreqProxTermsWriter、TermVectorsTermsWriter、NormsWriter 以及 StoredFieldsWriter 等。

索引创建处理过程及类的主线请求链表如下图所示：


IndexWriter
addDocument()
flush()
DocumentsWriter
addDocument()
finsihDocument()
WaitQueue
addt(DocWriter)
writeDocument(DocWriter)
PerDoc extends DocWriter
finish()
finishDocument(PerDoc)
flush()
DocumentsWriterThreadState
DocFieldProcessorPerThread
Call startDocument()
processDocument()
Call finsihDocument()
DocFieldConsumersPerField
processFields()
DocInverterPerField
processFields()
finish()
DocFieldConsumersPerThread
startDocument()
finishDocument()
DocInverterPerThread
startDocument()
finishDocument()
StoredFieldsWriterPerField
processFields()
TermVectorsTermsWriterPerThread
startDocument()
finishDocument()
StoredFieldsWriterPerThread
startDocument()
finishDocument()
TermsHashPerField
add(Token)
writeBytes(); writeVInt();
finish()
StoredFieldsWriter
PerDoc extends DocWriter
finishDocument(PerDoc)
TermsHashPerField next
add(Token)
writeBytes(); writeVInt();
finish()
FreqProxTermsWriterPerField
addTerm(Token); newTerm()
NormsWriter
flush(Fields, State)
TermVectorsTermsWriterPerField
addTerm(Token); newTerm()
finish()
FreqProxTermsWriter
flush(Fields, State)
NormsWriterPerField
finish()
TermVectorsTermsWriter
PerDoc extends DocWriter
finishDocument(PerDoc)
FieldsWriter (.fdx .fdt)
flushDocument(PerDoc)
writeField()
Directory.
IndexOutput
(.nrm)
writeBytes()
Directory.IndexOutput
(.tvx .tvf .tvd)
writeBytes()
Directory.
IndexOutput
(.prx)
writeBytes()
TermInfosWriter
(.tii .tis)
writeTerm()

下面介绍主要步骤的处理过程

4.3.1 DocFieldProcessorPerThread.processDocument()

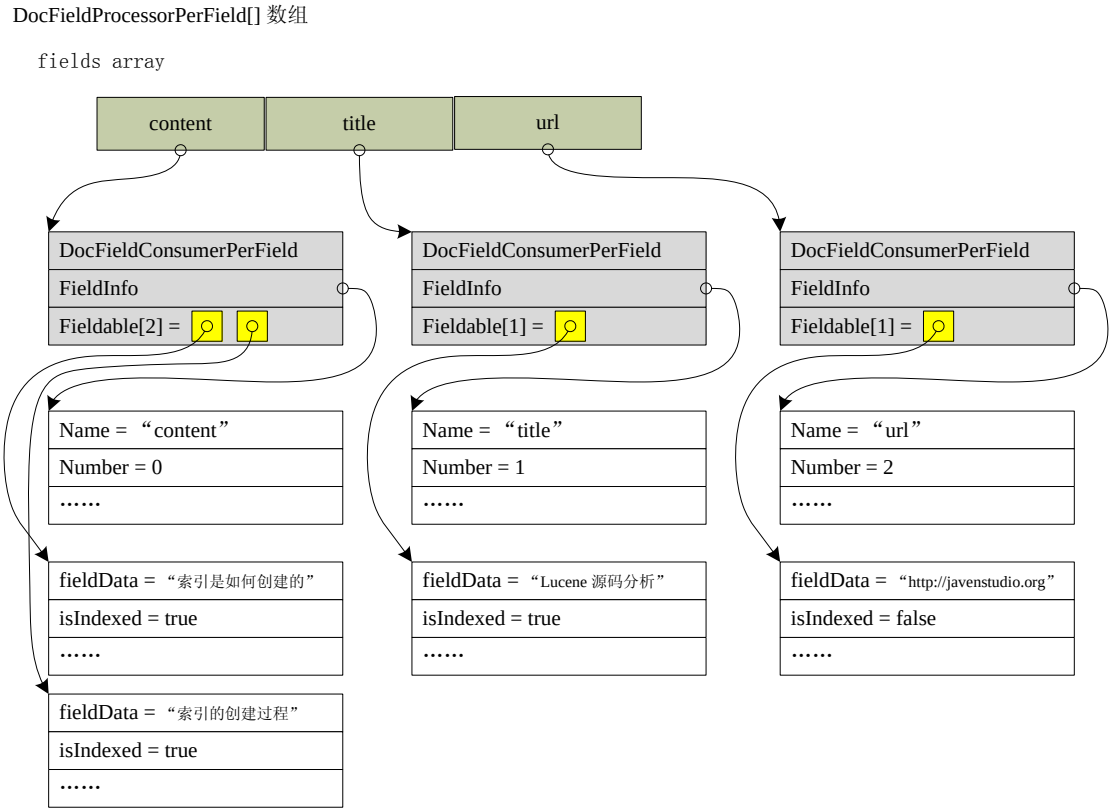
该方法是处理一个文档的调度函数，负责整理文档的各个 fields 数据，并创建相应的 DocFieldProcessorPerField 对象来依次处理每一个 field。该方法首先调用索引链表的 startDocument()来初始化各项数据，然后依次遍历每一个 fields，将它们建立一个以 field 名字计算的 hash 值为 key 的 hash 表，值为 DocFieldProcessorPerField 类型。如果 hash 表中已存在该 field，则更新该 FieldInfo（调用 FieldInfo.update()方法），如果不存在则创建一个新的 DocFieldProcessorPerField 来加入 hash 表中。注意，该 hash 表会存储包括当前添加文档的所有文档的 fields 信息，并根据 FieldInfo.update()来合并相同 field 名字的域设置信息。

建立 hash 表的同时，生成针对该文档的 fields[]数组（只包含该文档的 fields，但会共用相同的 fields 数组，通过 lastGen 来控制当前文档），如果 field 名字相同，则将 Field 添加到 DocFieldProcessorPerField 中的 fields 数组中。建立完 fields 后再将此 fields 数组按 field 名字排序，使得写入的 vectors 等数据也按此顺序排序。之后开始正式的文档处理，通过遍历 fields 数组依次调用 DocFieldProcessorPerField 的 processFields()方法进行（下小节继续讲解），完成后调用 finishDocument()完成后序工作，如写入 FieldInfos 等。

下面举例说明此过程，假设要添加如下一个文档：

文档域	内容	是否索引	是否存储	是否分析
title	Lucene 源码分析	true	true	true
url	http://javenstudio.org	false	true	false
content	索引是如何创建的	true	true	true
content	索引的创建过程	true	true	true

下图描述处理后 fields 数组的数据结构



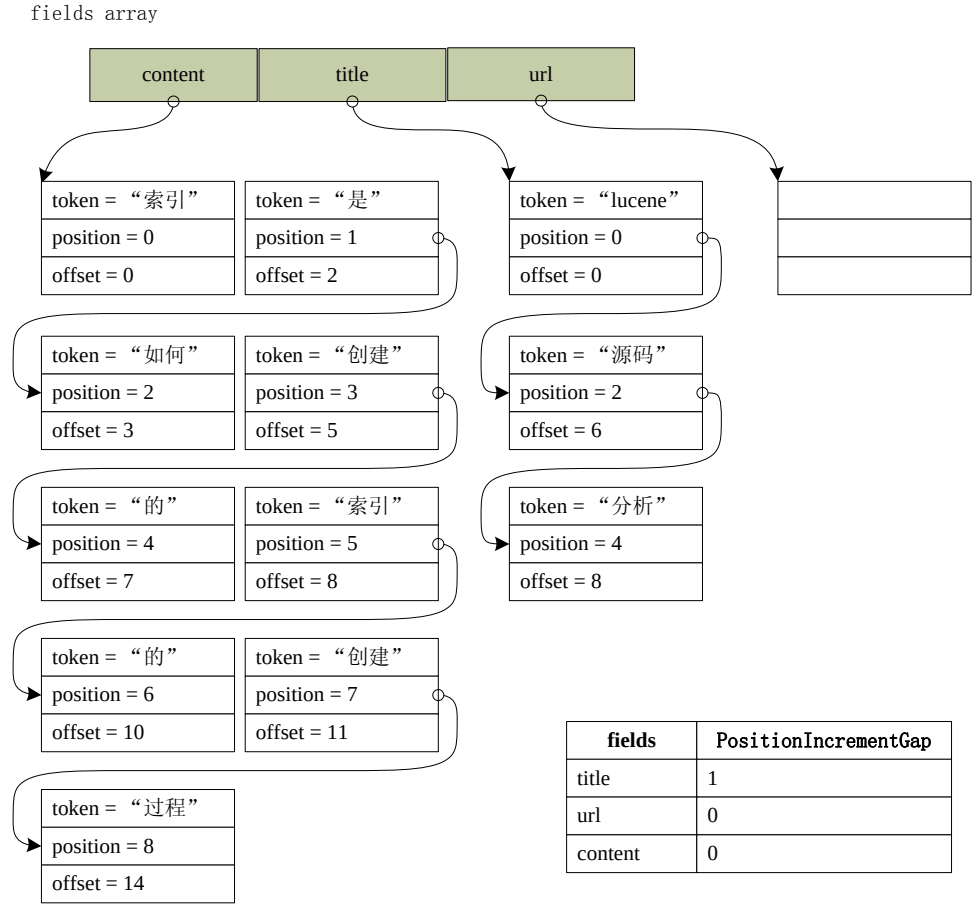
4.3.2 DocInverterPerField.processFields()

该方法负责进行文档 field 数据的倒排表（inverter）建立的处理工作。处理文档的有同一个名字的所有 field 数据，即上图的 Fieldable 数组，它负责将各个数据分解成（Tokenizer）一个个 term 并存储它们在文档中的位置及出现的频率，即 Term 向量和 Term 频率等数据。此方法负责文档的文本的分解工作（调用 Analyzer），将 term 传递到别的 consumers 进行具体的数据处理，即 InvertedDocConsumerPerField 和 InvertedDocEndConsumerPerField，在这里是 TermsHash 和 NormsWriter。

该方法通过一个 DocInverter.FieldInvertState 对象来存储和统计文档的当前 field 的所有 term 出现的位置 position 和 offset 以及频率 frequency 等信息，同时计算该 field 的 boost 分值，为所有相同名字的 fields 的 boost 与文档的 boost 的乘积。通过循环调用各个 consumer 的 start()方法检查该 field 是否需要进行 invert 处理，比如某些 field 是不需要索引的，或者 term 向量不用存储等。如果一个 field 需要索引但不需要分词（tokenize），则将该 field 整个文本数据当做一个 term 存储，term 长度也是该文本的长度，即 offset 按此长度累加，position 则累计加 1。如果需要索引并且分词，则调用 analyzer 指定的 tokenizer 进行分词处理，分解出一个个 token 来创建 term 添加到 TermsHash 和 NormsWriter。注意，该 term 的 position 会添加 token 的 positionIncrement 设置值（即用户指定的某些 token 的位置）。Offset 也按 token 的 offset 值计算，这些数值由分词模块计算，可以通过分词模块调优这些数值，比如索引带有歧义的词，或者增加索引同义词等。

分词以及 term 添加完成后，即调用 TermsHash 和 NormsWriter 的 finish()方法，统计和存储这些数据，下节详细讲解。下图描述方法处理中 field 的各项数据的结构：

processFields 处理



fields	PositionIncrementGap
title	1
url	0
content	0

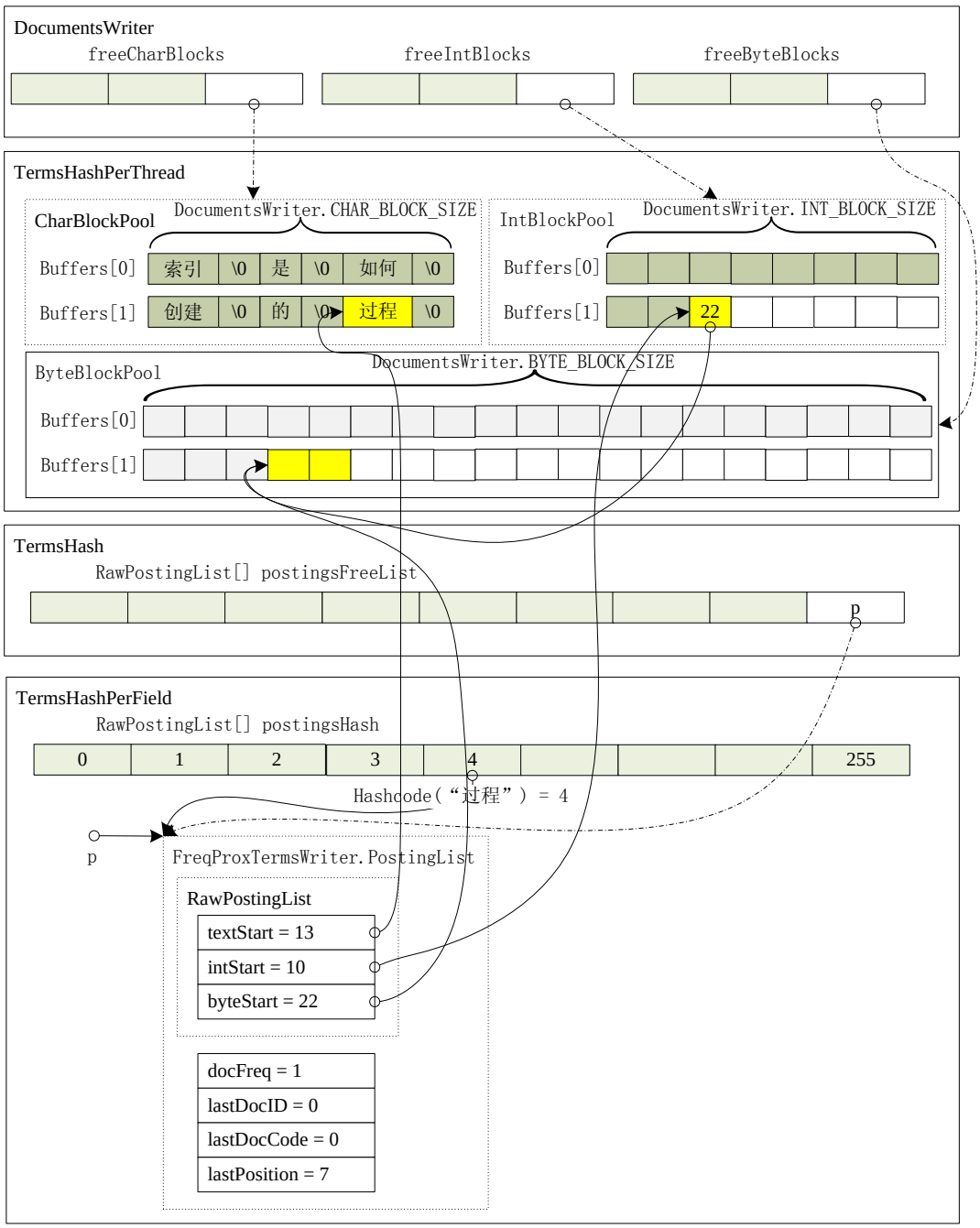
4.3.3 TermsHashPerField.addToken()

该方法负责进行将上面分解出的 token 字符串添加到 PostingList 表中，添加位置等信息的处理会调用接下来的 consumer 的方法，如 FreqProxTermsWriterPerField 或 TermVectorsTermsWriterPerField。此方法涉及到访问和管理三个内存中的数据池，即 CharBlockPool、IntBlockPool 和 ByteBlockPool。这三个池均采用分片（slice 或 block）来管理，每一片有固定的长度，定义在 DocumentsWriter 中，并且也由 DocumentsWriter 来分配新的块或者回收不用的块，以达到节省内存和提高效率的作用。

TermsHashPerField 会维护一个 postingHash 的散列表,用来存储和管理每一个添加的 token 及其位置信息，即 RawPostingsList 类。添加的时候先会计算 termText 的 hashCode 值，同时处理掉非法 Unicode 字符，然后在 postingsHash 表中查找该 TermText 的 RawPostingsList，找不到则添加一个新的，否则就追加新的位置信息。TermText 文本会写入到 CharBlockPool 中去，同时 RawPostingsList 中记录其位置即 textStart 值。ByteBlockPool 中写入各个 postiong 和 freq 数据（freq 在 fieldInfo 的 omitTf 设置为 false 时记录），这个处理在后面的类方法中进行（FreqProxTermsWriterPerField 和 TermVectorsTermsWriterPerField），RawPostingsList.byteStart 记录起始偏移。IntBlockPool 中记录这些数据在 ByteBlockPool 中位置偏移，RawPostingsList.intStart 记录起始偏移。

该方法处理逻辑和内存中重要的数据对象的关系图如下（其中假设 consumer 为 FreqProxTermsWriter）

TermsHashPerField.addToken(Token) 处理



4.3.4 FreqProxTermsWriterPerField.newTerm()/addTerm()

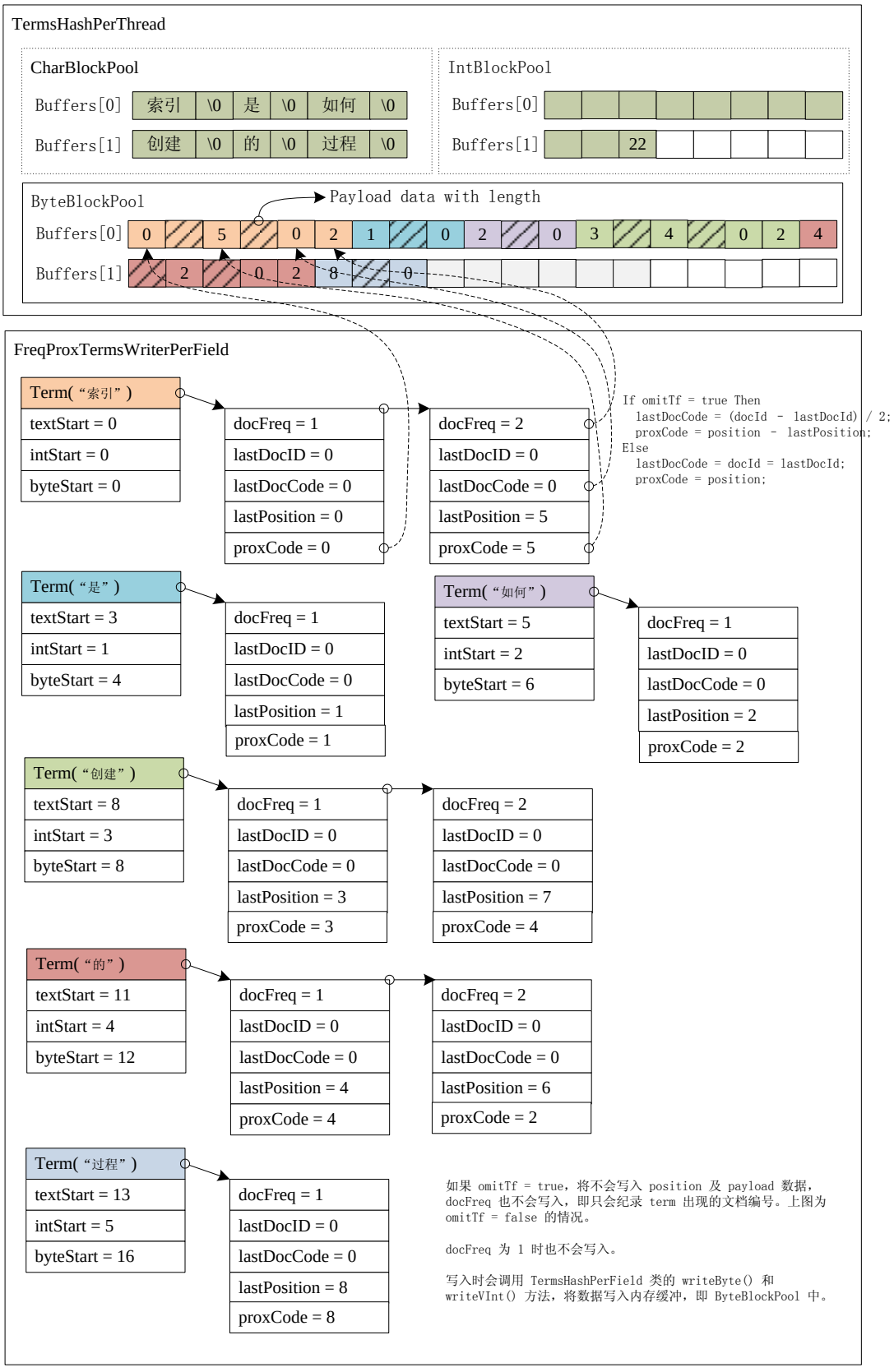
该方法负责将 term 在文档中出现的位置（position）和频率（frequency），以及 term 自带的 payloads 以及 term 所在的文档编号等信息写入内存中的缓冲区，即 TermsHashPerThread 中的 ByteBlockPool 对象。调用

TermsHashPerThread.writeByte()时同时会更新 IntBlockPool 中的值，指向对应的 ByteBlockPool 位置。

当 field 的 omitTf 为 true 时，只会记录 term 出现的文档的编号，记录时写入 docId 与上一个文档 docId 的差值，第一个为 docId 本身。如果 omitTf 为 false，则 term 出现的 position 以及 payloads（如果有的话），position 会记录当前位置与上一个位置的差值的一半，第一个值为本身的一半；并且在最后（term 出现在另一个文档时）写入文档编号（方法同 omitTf=true），以及 term 出现在该文档的频率 docFreq，不过如果 docFreq=1 就不会写入 docFreq，并且 docCode 写入与 1 的或运算值。

处理后的数据结构如下图所示，其中假设 docId=0，并且只画了处理 content 的 field 的在 omitTf 设置为 false 的数据，另外图中 ByteBlockPool 记录的是原始 int 数字，实际会写成 VInt 的字节序列，所以某些 offset 值只是示例。请参看图中的说明。

FreqProxTermsWriterPerField.newTerm()/addTerm() 处理



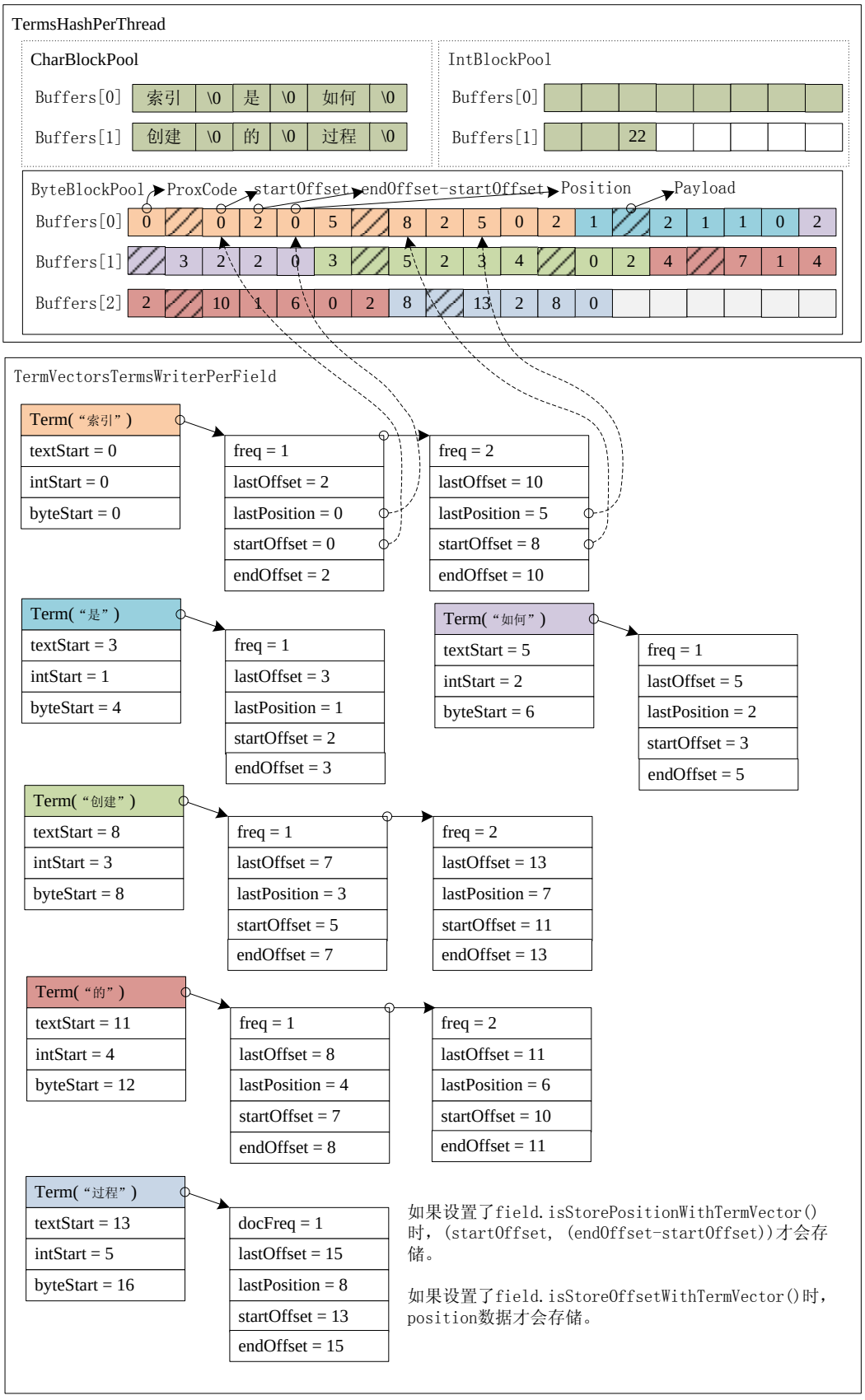
4.3.5 TermVectorsTermsWriterPerField.newTerm()/addTerm()

该方法负责将 term 在文档中出现的位置（position）和偏移（offset）向量（startOffset, endOffset）等信息写入内存中的缓冲区，即 TermsHashPerThread 中的 ByteBlockPool 对象。需要注意的是只有当 field.isStorePositionWithTermVector() 或 field.isStoreOffsetWithTermVector() 为 true 时，相应的信息(startOffset,endOffset)或 position 才会写入。

写入的这两项数据紧跟着上一节写入的 ProxCode 和 Payload 数据之后。写入 position 比较简单，即把该位置按 VInt 类型写入即可。偏移向量的写入是记录开始的偏移即 startOffset，以及结束的偏移与开始偏移的差值即 endOffset-startOffset，同样以 VInt 类型写入 ByteBlockPool。另外这些数据只是写入到内存缓冲中，在 finish()调用时才会把它们真正写入.tvx 和.tvf 和.tvd 向量文件中，下一节再详细介绍。

处理后的数据结构如下图所示，另外写入数据到 ByteBlockPool 的方法与上一节中 FreqProx 写入的处理一致，同样需要调用 TermsHashPerThread.writeByte()方法。请参看图中的说明。

TermVectorsTermsWriterPerField.newTerm()/addTerm() 处理



5 索引是如何存储的

5.1 数据存储类 Directory

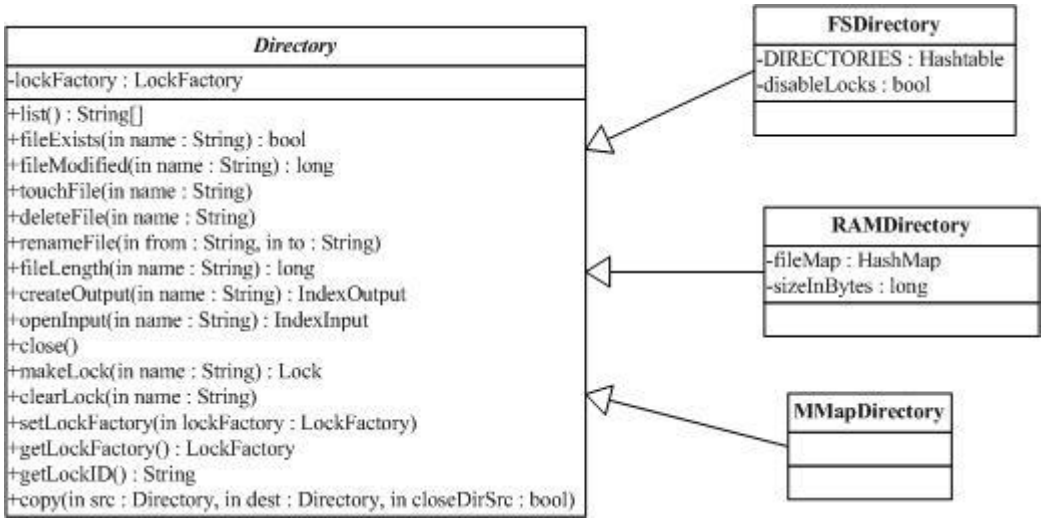
Directory 及相关类负责文档索引的存储。

5.1.1 org.apache.lucene.store.Directory

一个 Directory 对象是一系列统一的文件列表（a flat list of files）。文件可以在它们被创建的时候一次写入，一旦文件被创建，它再次打开后只能用于读取（read）或者删除（delete）操作。并且同时在读取和写入的时候允许随机访问（random access）。

在这里并不直接使用 Java I/O API，但是更确切地说，所有 I/O 操作都是通过这个 API 处理的。这使得读写操作方式更统一起来，如基于内存的索引（RAM-based indices）的实现（即 RAMDirectory）、通过 JDBC 存储在数据库中的索引、将一个索引存储为一个文件的实现（即 FSDirectory）。

Directory 的锁机制是一个 LockFactory 的实例实现的，可以通过调用 Directory 实例的 setLockFactory()方法来更改。

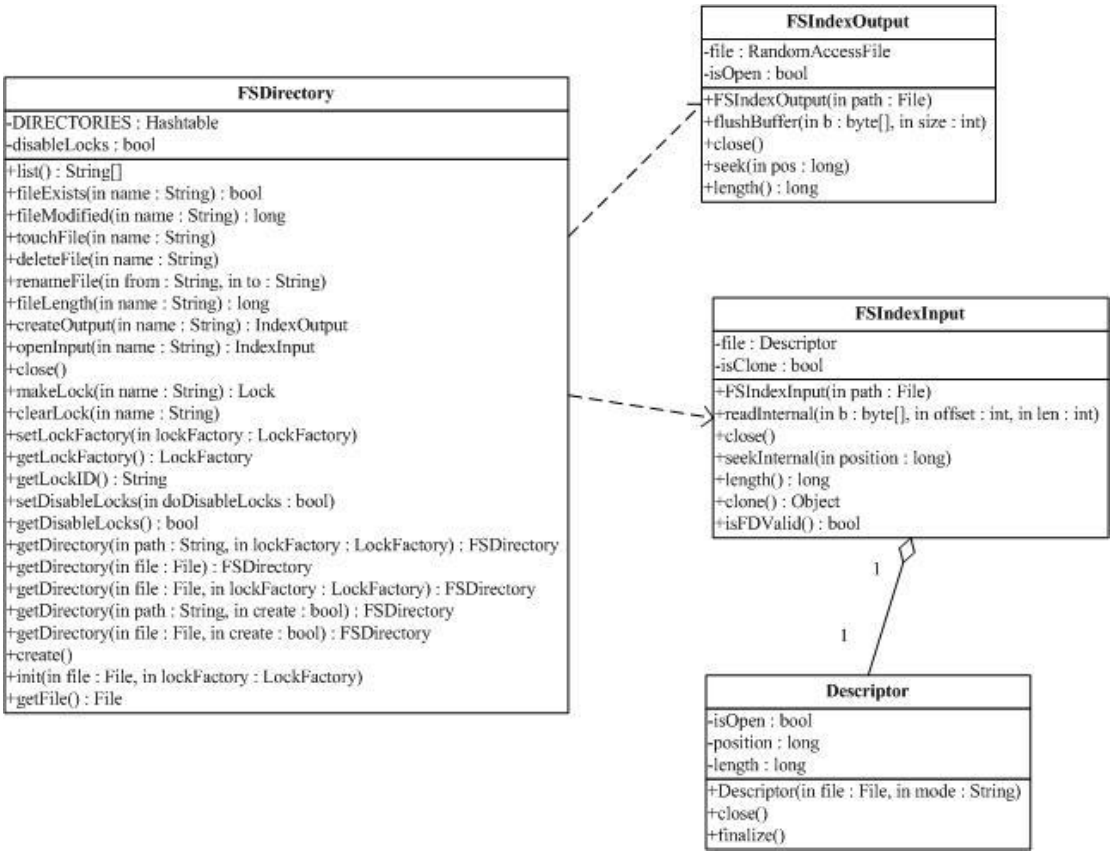


5.1.2 org.apache.lucene.store.FSDirectory

FSDirectory 类直接实现 Directory 抽象类为一个包含文件的目录。目录锁的实现使用缺省的

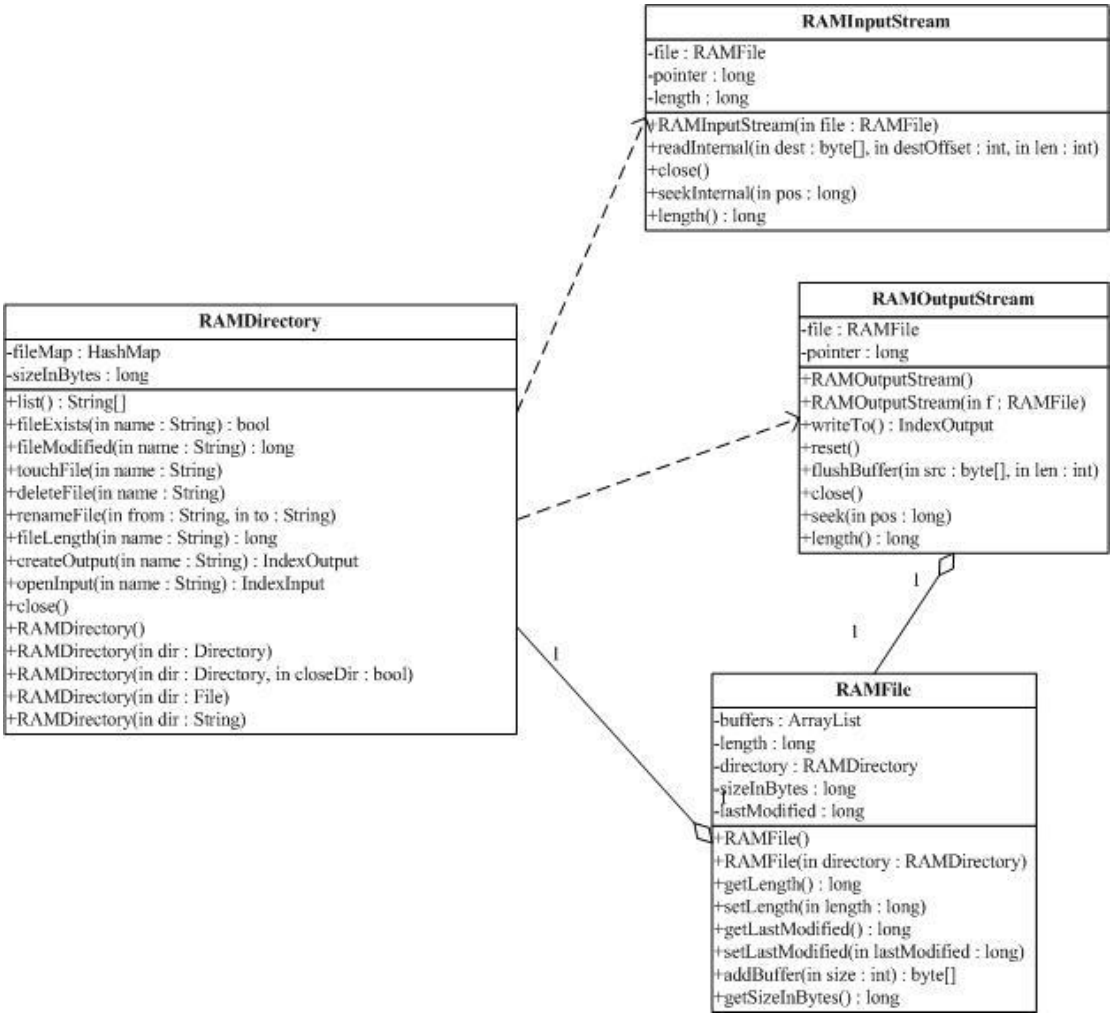
SimpleFSLockFactory，但是可以通过两种方式修改，即给 getLockFactory()传入一个 LockFactory 实例，或者通过调用 setLockFactory()方法明确制定 LockFactory 类。

目录将被缓存（cache）起来，对一个指定的符合规定的路径（canonical path）来说，同样的 FSDirectory 实例通常通过 getDirectory()方法返回。这使得同步机制（synchronization）能对目录起作用。



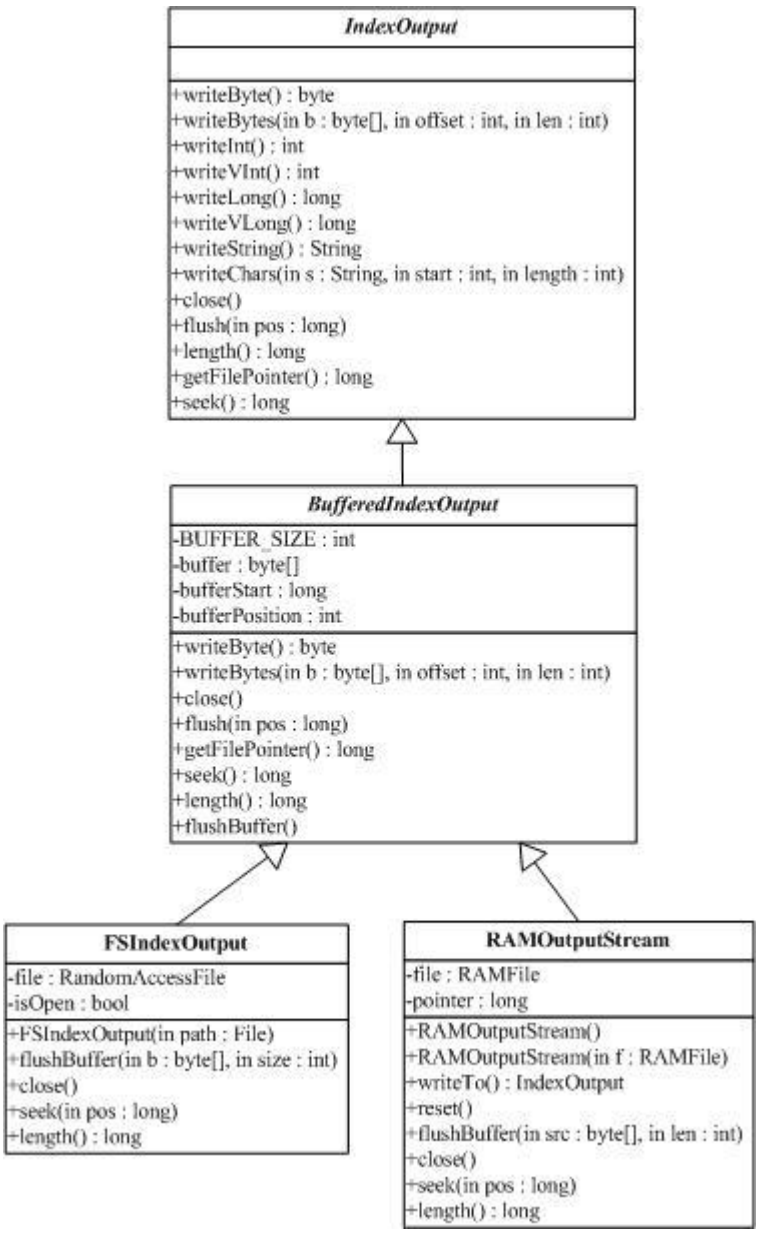
5.1.3 org.apache.lucene.store.RAMDirectory

RAMDirectory 类是一个驻留内存的（memory-resident）Directory 抽象类的实现。目录锁的实现使用缺省的 SingleInstanceLockFactory，但是可以通过 setLockFactory()方法修改。



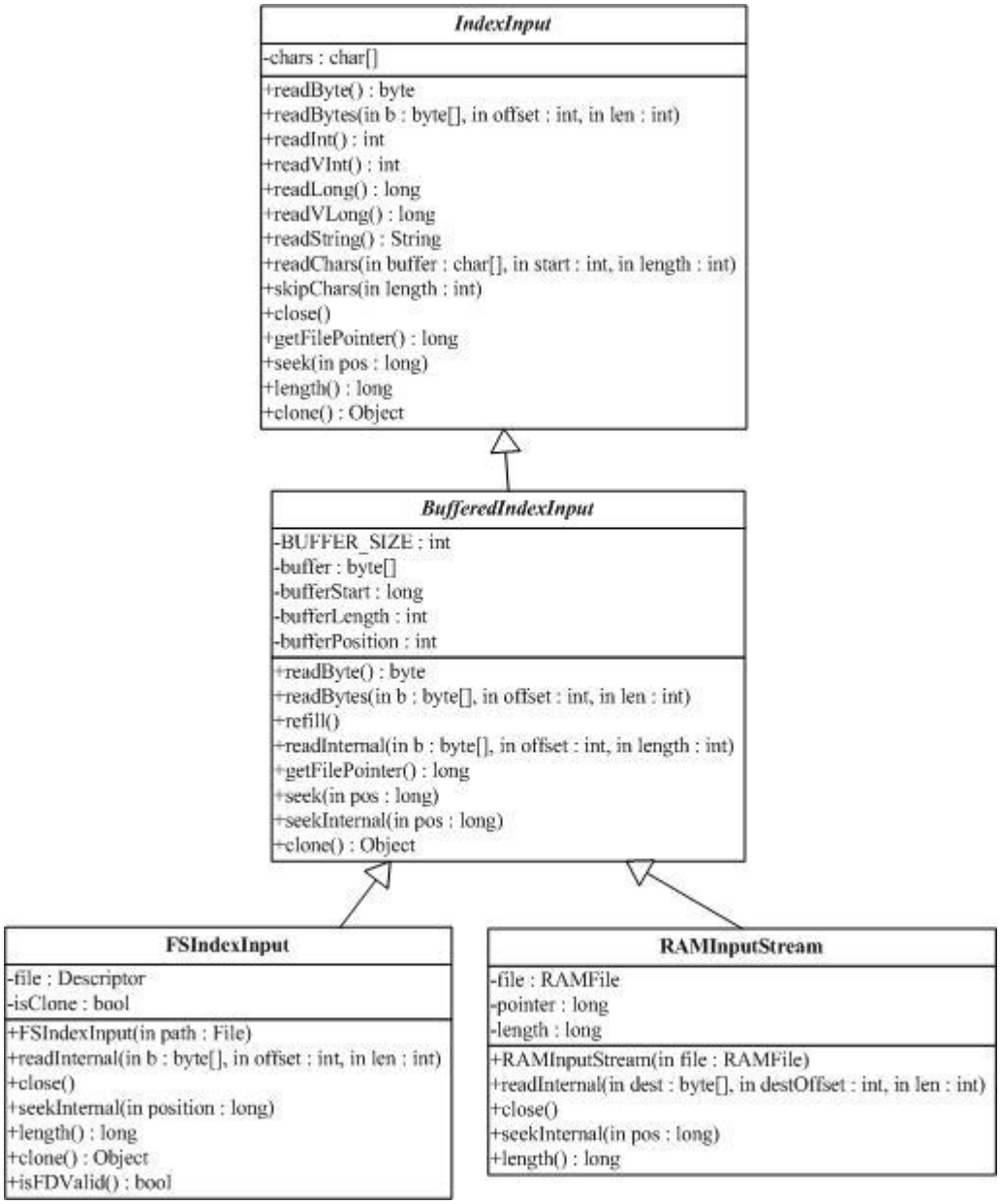
5.1.4 org.apache.lucene.store.IndexInput

IndexInput 类是一个为了从一个目录（Directory）中读取文件的抽象基类，是一个随机访问（random-access）的输入流（input stream），用于所有 Lucene 读取 Index 的操作。BufferedIndexInput 是一个实现了带缓冲的 IndexInput 的基础实现。



5.1.5 org.apache.lucene.store.IndexOutput

IndexOutput 类是一个为了写入文件到一个目录(Directory)中的抽象基类，是一个随机访问(random-access)的输出流（output stream），用于所有 Lucene 写入 Index 的操作。BufferedIndexOutput 是一个实现了带缓冲的 IndexOutput 的基础实现。RAMOuputStream 是一个内存驻留（memory-resident）的 IndexOutput 的实现类。


IndexInput
-chars : char[]
+readByte() : byte
+readBytes(in b : byte[], in offset : int, in len : int)
+readInt() : int
+readVInt() : int
+readLong() : long
+readVLong() : long
+readString() : String
+readChars(in buffer : char[], in start : int, in length : int)
+skipChars(in length : int)
+close()
+getFilePointer() : long
+seek(in pos : long)
+length() : long
+clone() : Object
BufferedIndexInput
-BUFFER_SIZE : int
-buffer : byte[]
-bufferStart : long
-bufferLength : int
-bufferPosition : int
+readByte() : byte
+readBytes(in b : byte[], in offset : int, in len : int)
+refill()
+readInternal(in b : byte[], in offset : int, in length : int)
+getFilePointer() : long
+seek(in pos : long)
+seekInternal(in pos : long)
+clone() : Object
FSIndexInput
-file : Descriptor
-isClone : bool
+FSIndexInput(in path : File)
+readInternal(in b : byte[], in offset : int, in len : int)
+close()
+seekInternal(in position : long)
+length() : long
+clone() : Object
+isFDValid() : bool
RAMInputStream
-file : RAMFile
-pointer : long
-length : long
+RAMInputStream(in file : RAMFile)
+readInternal(in dest : byte[], in destOffset : int, in len : int)
+close()
+seekInternal(in pos : long)
+length() : long

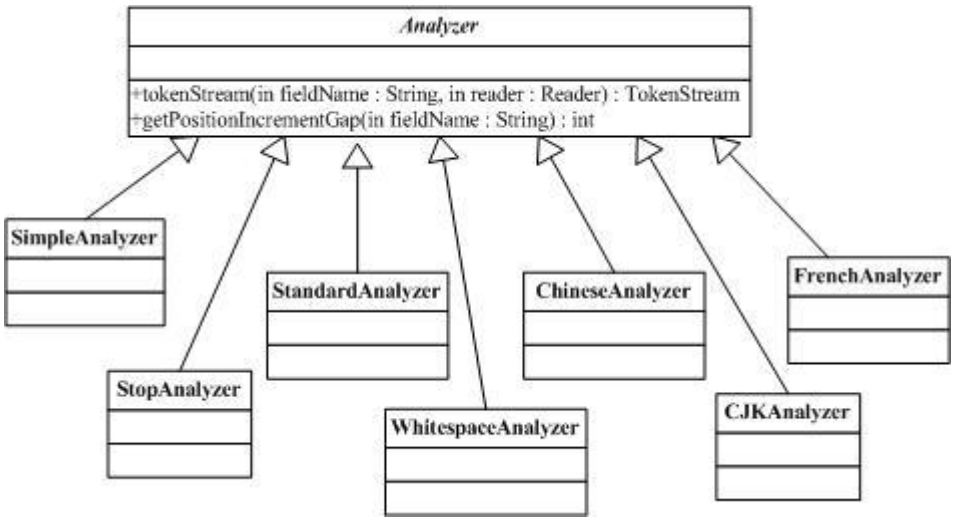
6 文档内容是如何分析的

Analyzer 类负责分析文档结构并提取内容。

6.1 文档分析类 Analyzer

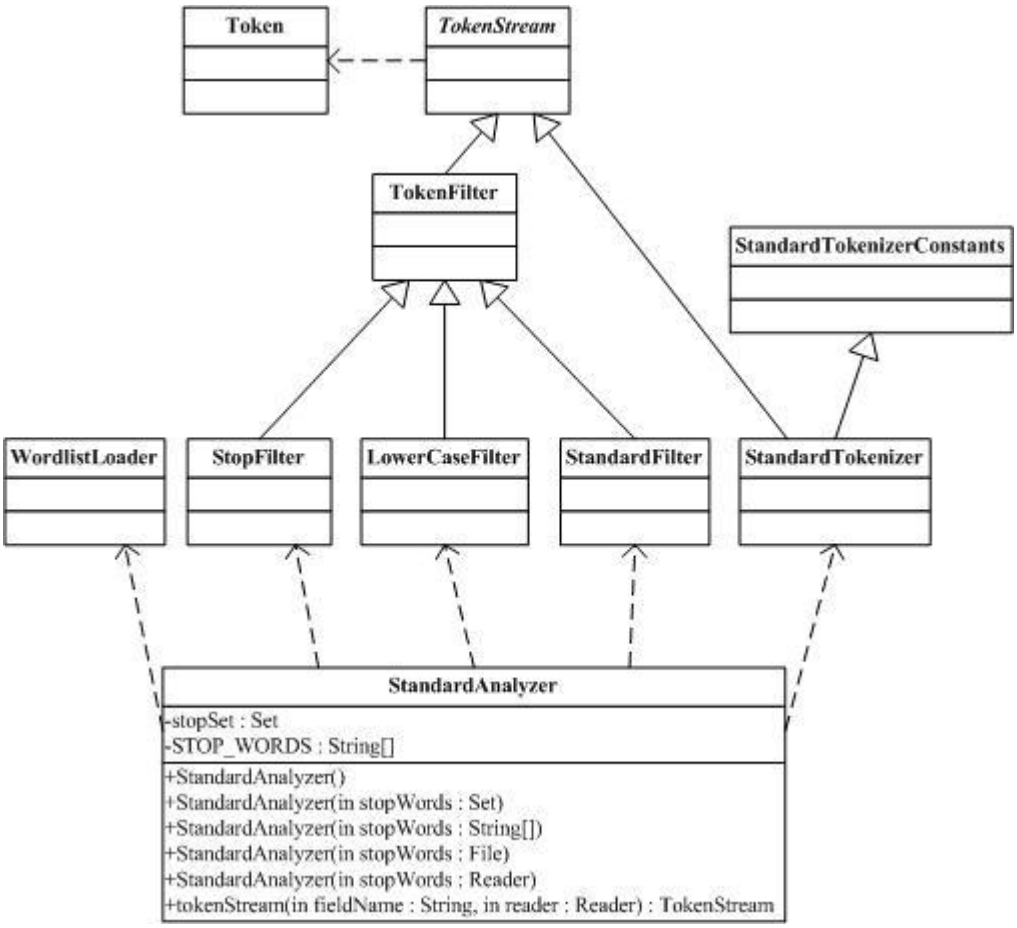
6.1.1 org.apache.lucene.store.Analyzer

Analyzer 类构建用于分析文本的 TokenStream 对象，因此(thus)它表示(represent)用于从文本中分解(extract)出组成索引的 terms 的一个规则器（policy）。典型的（typical）实现首先创建一个 Tokenizer，它将那些从 Reader 对象中读取字符流（stream of characters）打碎为（break into）原始的 Tokens（raw Tokens）。然后一个或更多的 TokenFilters 可以应用在这个 Tokenizer 的输出上。警告：你必须在你的子类（subclass）中覆写（override）定义在这个类中的其中一个方法，否则的话 Analyzer 将会进入一个无限循环（infinite loop）中。



6.1.2 org.apache.lucene.store.StandardAnalyzer

StandardAnalyzer 类是使用一个 English 的 stop words 列表来进行 tokenize 分解出文本中 word，使用 StandardTokenizer 类分解词，再加上 StandardFilter 以及 LowerCaseFilter 以及 StopFilter 这些过滤器进行处理的这样一个 Analyzer 类的实现。



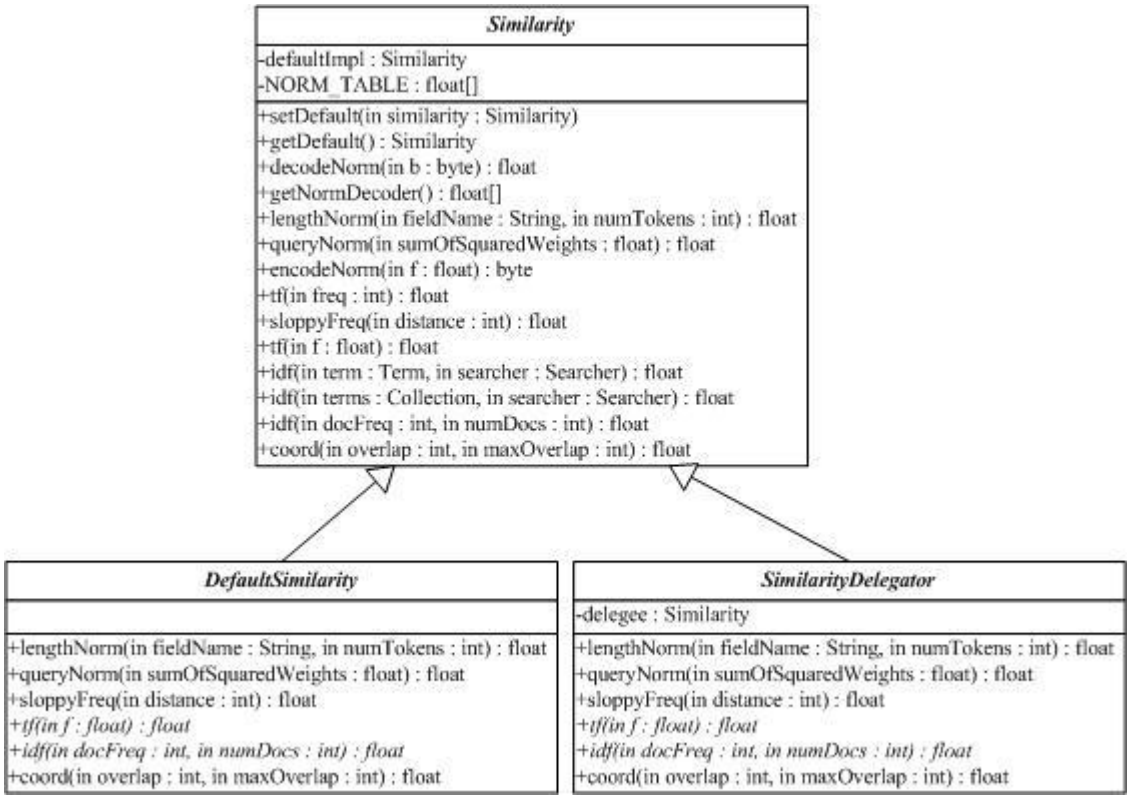
7 如何给文档评分

Similarity 类负责给文档评分。

7.1 文档评分类 Similarity

7.1.1 org.apache.lucene.search.Similarity

Similarity 类实现算分（scoring）的 API，它的子类实现了检索算分的算法。DefaultSimilarity 类是缺省的算分的实现，SimilarityDelegator 类是用于委托算分（delegating scoring）的实现，在 Query.getSimilarity(Searcher)}的实现里起作用，以便覆写（override）一个 Searcher 中 Similarity 实现类的仅有的确定方法（certain methods）。



7.2 Similarity 评分公式

查询 q 相对于文档 d 的分数与在文档和查询向量（query vectors）之间的余弦距离（cosing-distance）或者点乘积（dot-product）有关系（correlates to），文档和查询向量存于一个信息检索（Information Retrieval）的向量空间模型（Vector Space Model (VSM)）之中。一篇文档的向量与查询向量越接近（closer to），它的得分也越高（scored higher），这个分数按如下公式计算：

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

其中：

1. **tf(t in d)** 与 term 的出现次数（*frequency*）有关系（correlate to），定义为（defined as）term **t** 在当前算分（currently scored）的文档 **d** 中出现（appear in）的次数（number of times）。对一个给定（gived）的 term，那些出现此 term 的次数越多（more occurences）的文档将获得越高的分数（higher score）。缺省的 *tf(t in d)*算法实现在 DefaultSimilarity 类中，公式如下：

$$tf(t\_in\_d) = frequency^{\frac{1}{2}}$$

2. **idf(t)** 代表（stand for）反转文档频率（Inverse Document Frequency）。这个分数与反转（inverse of）的 *docFreq*（出现过 term **t** 的文档数目）有关系。这个分数的意义是越不常出现（rarer）的 term 将为最后的总分贡献（contribution）更多的分数。缺省 idf*f(t in d)*算法实现在 DefaultSimilarity 类中，公式

如下：

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

3. **coord(q,d)** 是一个评分因子，基于（based on）有多少个查询 terms 在特定的文档（specified document）中被找到。通常（typically），一篇包含了越多的查询 terms 的文档将比另一篇包含更少查询 terms 的文档获得更高的分数。这是一个搜索时的因子（search time factor）是在搜索的时候起作用（in effect at search time），它在 Similarity 对象的 *coord(q,d)*函数中计算。

4. **queryNorm(q)** 是一个修正因子（normalizing factor），用来使不同查询间的分数更可比较（comparable）。这个因子不影响文档的排名（ranking）（因为搜索排好序的文档（ranked document）会增加（multiplied）相同的因数（same factor）），更确切地说只是（but rather just）为了尝试（attempt to）使得不同查询条件（甚至不同索引（different indexes））之间更可比较性。这是一个搜索时的因子是在搜索的时候起作用，由 Similarity 对象计算。缺省 queryNorm*(q)*算法实现在 DefaultSimilarity 类中，公式如下：

5.

$$queryNorm(q) = queryNorm(sumOfSquaredWeights) = \frac{1}{sumOfSquaredWeights^{1/2}}$$

sumOfSquaredWeights（查询的 terms）是由查询 Weight 对象计算的，例如一个布尔（boolean）条件查询的计算公式为：

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

6. **t.getBoost()** 是一个搜索时（search time）的代表查询 **q** 中的 term **t** 的 boost 数值，具体指定在（as specified in）查询的文本中（参见查询语法），或者由应用程序调用 setBoost()来指定。需要注意的是实际上（really）没有一个直接（direct）的 API 来访问（accessing）一个多个 term 的查询（multi term query）中的一个 term 的 boost 值，更确切地说（but rather），多个 terms（multi terms）在一个查询里的表示形式（represent as）是多个 TermQuery 对象，所以查询里的一个 term 的 boost 值的访问是通过调用子查询（sub-query）的 getBoost()方法实现的。

7. **norm(t,d)** 是提炼取得（encapsulate）一小部分 boost 值（在索引时间）和长度因子（length factor）：
   - **document boost** – 在添加文档到索引之前通过调用 doc.setBoost()来设置。
   - **Field boost** – 在添加 Field 到文档之前通过调用 field.setBoost()来设置。
   - **lengthNorm(field)** – 在文档添加到索引的时候，根据（in accordance with）文档中该 field 的 tokens 数目计算得出，所以更短（shorter）的 field 会贡献更多的分数。lengthNorm 是在索引的时候起作用，由 Similarity 类计算得出。

当一篇文档被添加到索引的时候，所有上面计算出的因子将相乘起来（multiplied）。如果文档拥有多个相同名字的 fields（multiple fields with same name），所有这些 fields 的 boost 值也会被一起相乘起来（multiplied together）：

$$norm(t,d) = doc.getBoost() \times lengthNorm(field) \times \prod_{field\_f\_in\_d\_named\_as\_t} f.getBoost()$$

然而 *norm* 数值的结果在被存储（stored）之前被编码成（encoded as）一个单独的字节(single byte)。在检索的时候，这个 *norm* 字节值从索引目录（index directory）中读取出来，并解码回（decoded back）一个 *norm* 浮点数值（float value）。这个编/解码（encoding/decoding）行为，会缩减（reduce）索引的大小（index size），这得自于（come with）精度损耗的代价（price of precision loss）- 它不保证 decode(encode(x))=x，举例来说 decode(encode(0.89))=0.75。还有需要注意的是，检索的时候再修改评分（scoring）的这个 norm 部分已近太迟了，例如，为检索使用不同的 Similarity。

Annotated Lucene 作者：naven 日期：2008-10-20