**导入一个json格式的日志数据**

导入时需要有相应的索引名、类型名以及id，如果没有索引及类型es会自动创建新的索引及类型，id会自动生成。

返回结果有相应的信息（索引名、类型名、id以及版本），索引创建的过程信息（total：文档被创建的时候，在多少分片中进行了操作，包括主分片和副本分片；successful：成功建立索引分片的数量，当创建成功后，成功创建索引分片的数量最少是1；failed：失败建立索引分片的数量。）

## 自动创建索引

当创建文档的时候，如果索引不存在，则会自动创建该索引。自动创建的索引会自动映射每个字段的类型。自动创建字段类型是非常灵活的，新的字段类型将会自动匹配字段对象的类型。比如字符串类型，日期类型。自动创建索引可以通过配置文件设置action.auto\_create\_index为false在所有节点的配置文件中禁用。自动映射的字段类型可以通过配置文件设置index.mapper.dynamic为false禁用。自动创建索引可以通过模板设置索引名称，例如：可以设置 action.auto\_create\_index为 +aaa\*,-bbb\*,+ccc\*,-\* (+表示准许，-表示禁止)。

## 版本号

每个文档都有一个版本号，版本号的具体值放在在创建索引的返回值中("\_version":)。通过版本号参数可以达到乐观并发控制的效果。当在操作文档的过程中指定版本号，如果和版本号不一致的时候操作会被拒绝。

 默认情况下对文档的操作版本号从1开始递增，包括修改文档和删除文档。当然版本号还可以从外部获取，比如从数据库中获取，要启用此功能，version\_type应设置为external，这个值必须是一个大于0小于 9.2e+18的数字。当使用外部版本号来代替自动生成的版本号时，在操作文档的时候，系统通过对比参数中的版本号是否大于文档中的版本号来做判断，当参数中的版本号大于系统中的版本号，则执行此操作，并更新版本号。反之则拒绝操作(包括小于或者等于)。

    版本号同时产生了一个比较实用的功能，只要版本号从源数据库中使用，在异步索引操作的时候就不需要对源数据库的变化执行严格排序。任何操作都只会对最新的版本号起作用，不管这个版本号是内部的还是从外部获取的。

## 自动创建ID

当创建文档的时候，如果不指定ID，系统会自动创建ID，自动生成的id是一个不会重复的随机数。

## 分片选择

当你索引一个文档，它被存储在单独一个主分片上。Elasticsearch是如何知道文档属于哪个分片的呢？当你创建一个新文档，它是如何知道是应该存储在分片1还是分片2上的呢？

进程不能是随机的，因为我们将来要检索文档。事实上，它根据一个简单的算法决定：

shard = hash(routing) % number\_of\_primary\_shards

routing值是一个任意字符串，它默认是\_id但也可以自定义。这个routing字符串通过哈希函数生成一个数字，然后除以主切片的数量得到一个**余数(remainder)**，余数的范围永远是0到number\_of\_primary\_shards - 1，这个数字就是特定文档所在的分片。

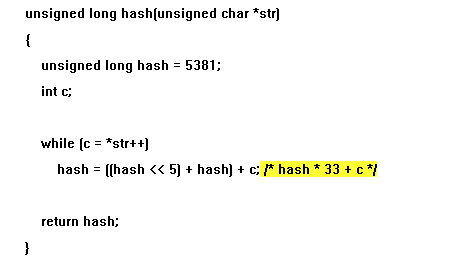
这也解释了为什么主分片的数量只能在创建索引时定义且不能修改：如果主分片的数量在未来改变了，所有先前的路由值就失效了，文档也就永远找不到了

**这里用的是djb2哈希函数**

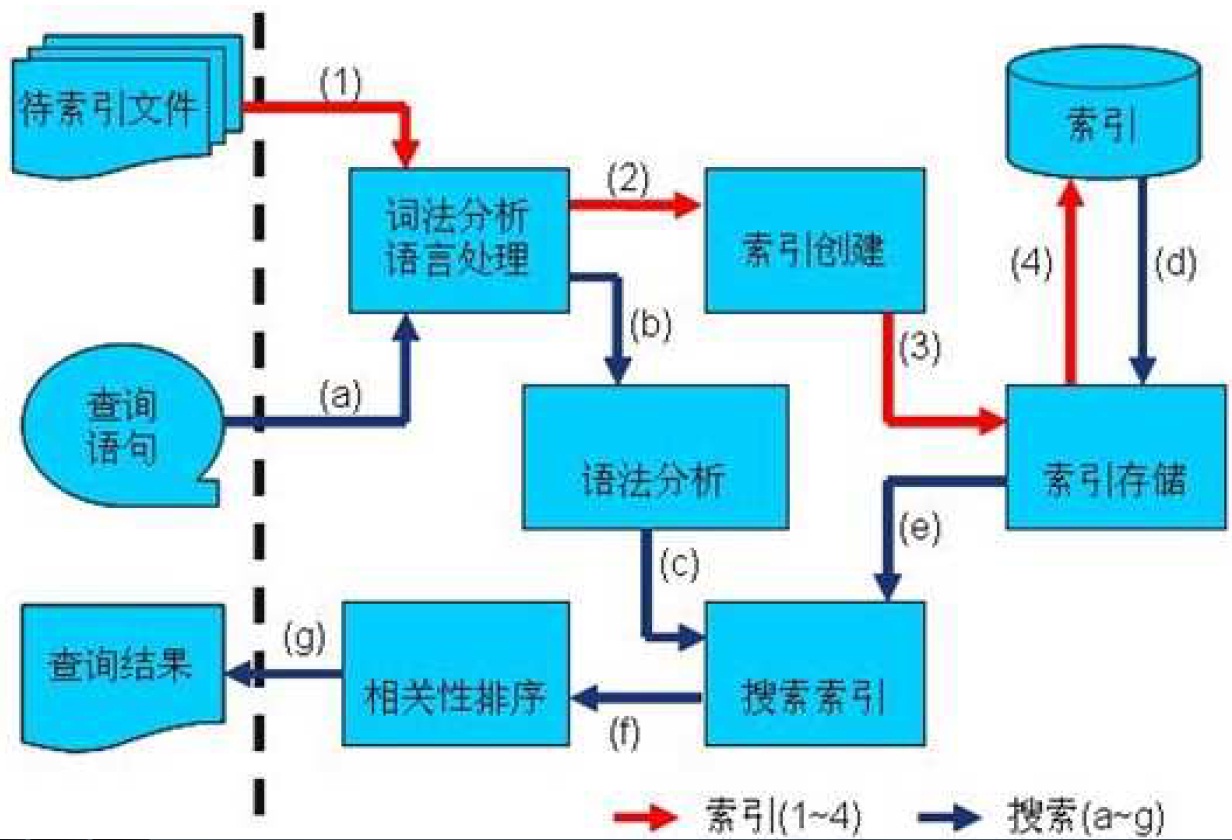
 默认情况下，分片的选择是通过ID的散列值进行控制，分片的选择。这个只可以通过router 参数进行手动的控制。可以在每个操作的基础上直接通过哈希函数的值来指定分片的选择。例如：

请求：POST http://localhost:9200/secilog/log/?routing=secisland&pretty

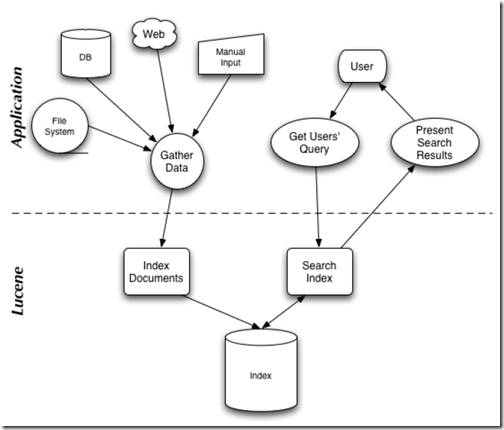
在上面的例子中，分片的选择是通过指定routing=secisland参数的哈希值来确定的。



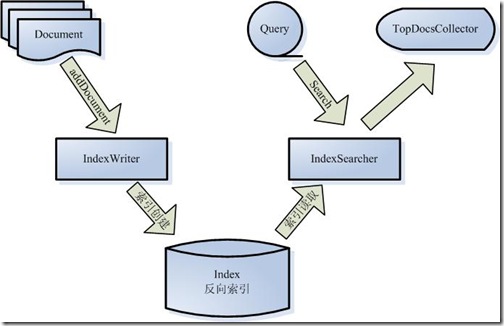
## 分片存储

* 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。
* 

### Lucene架构

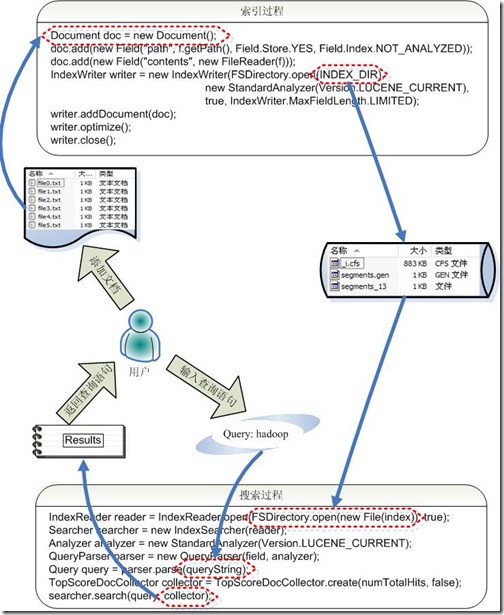


组件：



* 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。

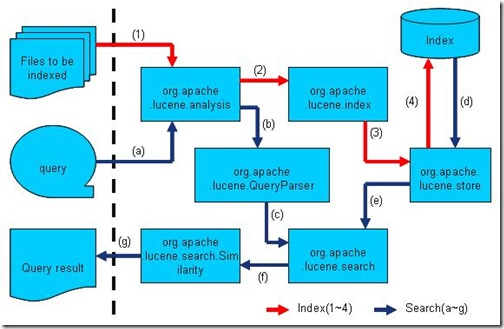
**Lucene API的调用实现索引和搜索过程：**



* **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。

**Lucene源码模块：**



* 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。

### Lucene索引结构

**索引vs分片**

**为了避免混淆，需要说明，Lucene索引是Elasticsearch中的分片，Elasticsearch中的索引是分片的集合。当Elasticsearch搜索索引时，它发送查询请求给该索引下的所有分片，然后过滤这些结果，聚合成全局的结果。**

Lucene的索引结构是有层次结构的，主要分为以下几个层次：

* 索引(Index)：（es中的分片）
  + 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，既包含正向信息，也保存了反向信息。

**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。

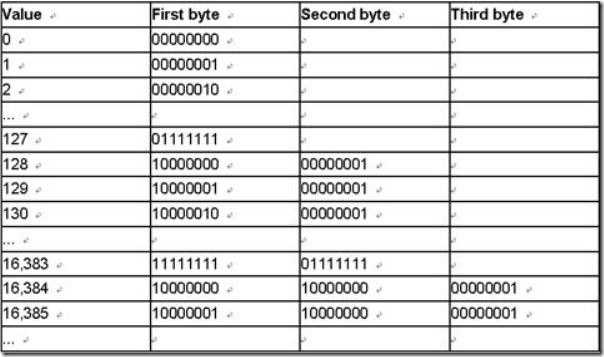
**反向信息：**

* 保存了词典到倒排表的映射：词(Term) –> 文档(Document)
* 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。

### 基本类型

Lucene索引文件中，用一下基本类型来保存信息：

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



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

### 基本规则

#### 前缀后缀规则（prefix+suffix）

#### 差值规则（Delta）

#### 或然跟随规则（A,B?）

#### 跳跃表规则

### 具体格式

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。

#### 正向信息

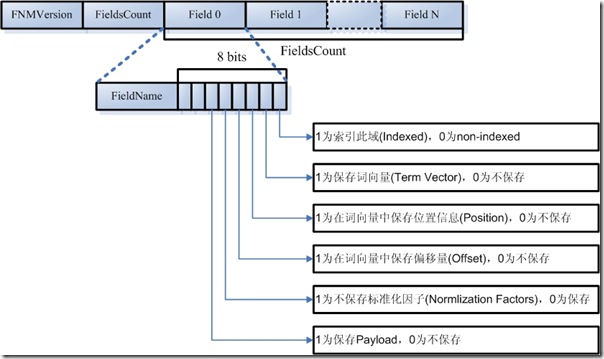
Index –> Segments (segments.gen, segments\_N) –> Field(fnm, fdx, fdt) –> Term (tvx, tvd, tvf)

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。

##### 段的元数据信息（segment\_N）与segment\_gen的内容差别？？

##### 域（field）的元数据信息（.fnm）以及数据信息（.fdt,.fdx）

.fnm文件的格式如下：



**.fdt(域数据文件)：**

* 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。

##### 词向量（term vector）的数据信息（.tvx,.tvd,.tvf）

**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。

**内容在：<http://www.cnblogs.com/forfuture1978/archive/2009/12/14/1623599.html>中**

#### 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，左边是词典（term dictionary），右边是倒排表（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。

**词典（tis）及词典索引（tii）信息**

* 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。。。

**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

**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。

**在：<http://www.cnblogs.com/forfuture1978/archive/2010/02/02/1661436.html>中**

#### 其它信息

**标准化因子文件（nrm）**

标准化因子文件(Normalization Factor File: nrm)：

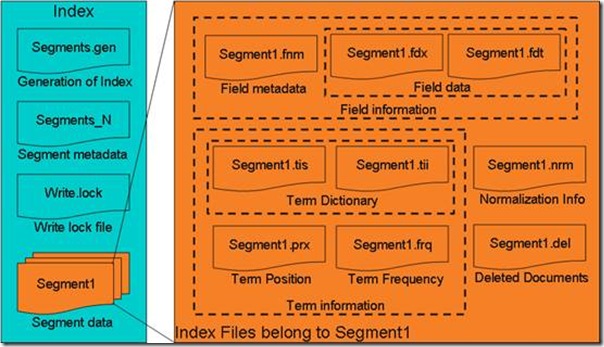
* NormsHeader：字符串“NRM”外加Version，依Lucene的版本的不同而不同。
* 接着是一个数组，大小为NumFields，每个Field一项，每一项为一个Norms。
* Norms也是一个数组，大小为SegSize，即此段中文档的数量，每一项为一个Byte，表示一个浮点数，其中0~2为尾数，3~8为指数。

**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。

### 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)。

<http://www.infoq.com/cn/articles/analysis-of-elasticsearch-cluster-part01>