全文检索

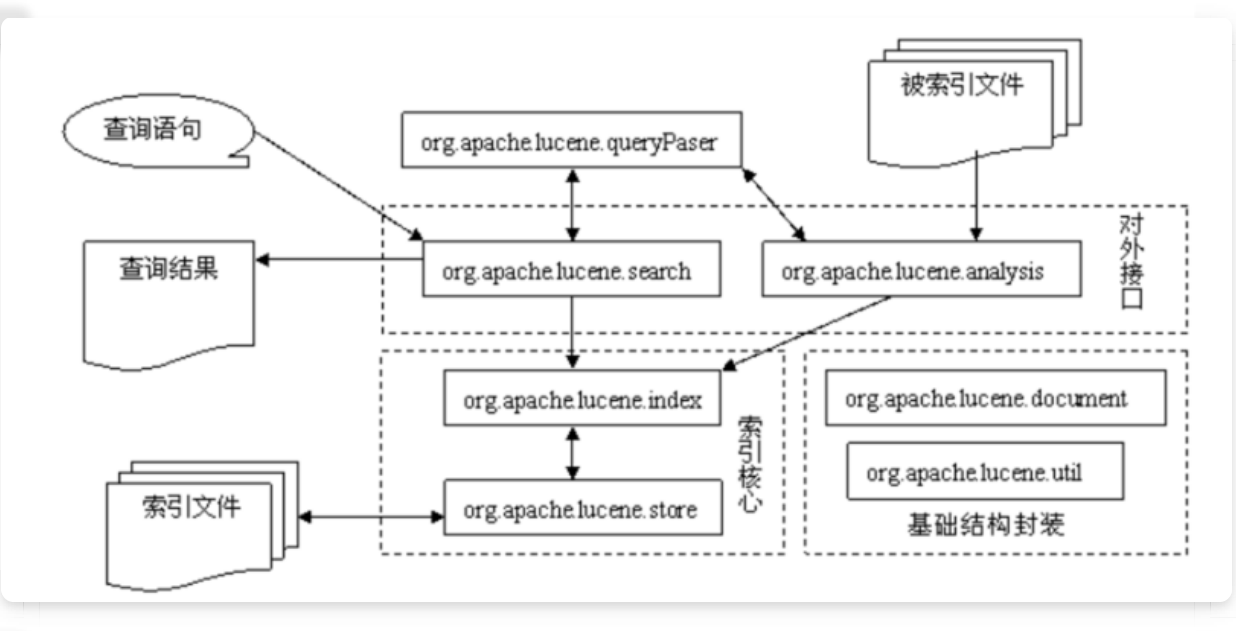
lucene 官方定义：lucene是个高效的全文检索引擎

什么是全文检索： 非结构化数据又叫全文数据
对于全文数据的搜索：

1. 顺序扫描法(grep)
2. 一部分信息提取重新组织，变成结构化数据（索引）
   这种先建立索引,再对索引进行搜索的过程就叫全文检索

所以全文检索大体分为两部分：建立索引，搜索索引

lucene核心结构：



包名	功能
org.apache.lucene.analysis	语言分析器，主要用于的切词，支持中文主要是扩展此类
org.apache.lucene.document	索引存储时的文档结构管理，类似于关系型数据库的表结构
org.apache.lucene.index	索引管理，包括索引建立、删除等
org.apache.lucene.queryParser	查询分析器，实现查询关键词间的运算，如与、或、非等
org.apache.lucene.search	检索管理，根据查询条件，检索得到结果
org.apache.lucene.store	数据存储管理，主要包括一些底层的I/O操作
org.apache.lucene.util	一些公用类

建立索引

document -> Tokenizer ->TokenFilter -> Indexer

原始文档->分词组件->处理组件->索引组件

处理组件：
1.使用 LowerCaseFilter or LowerCaseTokenizer 把token变为小写
2.使用 PorterStemFilter 将单词变为词根形式（porter词干算法）

索引组件：
1.term 建立字典
2.字典按照字母顺序排序
3.合并相同的term成为倒排

正向信息：
索引(Index) –> 段(segment) –> 文档(Document) –> 域(Field) –> 词(Term)

反向信息：
词典到倒排表的映射: 词(Term) –> 文档(Document)

lucene 索引文件

后缀名	文件名	描述
.fdt	field data	存储所有文档的字段信息
.fdx	field index	存储域数据的指针
.fnm	fields	存储域文件的信息
.doc	frequencies	包含每个term以及频率
.pos	positions	存储一个term在索引中的位置
.tim	term dictionary	term字典，存储term信息
.tip	term index	term字典的索引文件
.dvd	docValues data	
.dvm	docValues metadata	
.nvd	norms data	
.nvm	norms metadata	
.si	segment info	存储每个段文件的元数据信息
.cfe	Compound Entry Table	The "virtual" compound file's entry table holding all entries in the corresponding .cfs file.
.cfs	Compound	An optional "virtual" file consisting of all the other index files for systems that frequently run out of file handles.
segment_N	segments files	段文件，存储提交点的信息
write.lock	lock file	文件锁，保证任何时刻只有一个线程可以写入索引

.tim .tip .doc .pos

1. 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
   1.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。
   2.越前面的 Byte 表示数值的低位,越后面的 Byte 表示数值的高位。
   3.例如 16383 在最高位置 1 来表示后面还有 一个Byte,所以为(1) 1111111,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) 1111111。

```
public final void writeVInt(int i) throws IOException {
  while ((i & ~0x7F) != 0) {
    writeByte((byte)((i & 0x7F) | 0x80));
    i >>>= 7;
  }
  writeByte((byte)i);
}
```

2. PackedInt：
   位压缩
   Byte,short,int,long
   整型数组的类型从固定大小(8,16,32,64位)4种类型，扩展到了[1-64]位共64种类型

```
switch (format) {
  case PACKED_SINGLE_BLOCK:
    return Packed64SingleBlock.create(in, valueCount, bitsPerValue);
  case PACKED:
    switch (bitsPerValue) {
      case 8:
        return new Direct8(version, in, valueCount);
      case 16:
        return new Direct16(version, in, valueCount);
      case 32:
        return new Direct32(version, in, valueCount);
      case 64:
        return new Direct64(version, in, valueCount);
      case 24:
        if (valueCount <= Packed8ThreeBlocks.MAX_SIZE) {
          return new Packed8ThreeBlocks(version, in, valueCount);
        }
        break;
      case 48:
        if (valueCount <= Packed16ThreeBlocks.MAX_SIZE) {
          return new Packed16ThreeBlocks(version, in, valueCount);
        }
        break;
```

Packed64SingleBLock,看起来像 Packed64，但实际上利用位填充来做代替跨多个块的数据存储。 (每个值最大占用32 bits),

Packed8ThreeBlocks和 Packed16ThreeBlocks, 用 3 bytes (每个值24 bits) or 3 shorts (每个值48 bits)来存储数据。

3. skip table
   下面的代码是将跳跃表缓存写入到文件里：

```
public long writeSkip(IndexOutput output) throws IOException {
  long skipPointer = output.getFilePointer();
  //System.out.println("skipper.writeSkip fp=" + skipPointer);
  if (skipBuffer == null || skipBuffer.length == 0) return skipPointer;

  for (int level = numberOfSkipLevels - 1; level > 0; level--) {
    long length = skipBuffer[level].getFilePointer();
    if (length > 0) {
      output.writeVLong(length);
      skipBuffer[level].writeTo(output);
    }
  }
  skipBuffer[0].writeTo(output);

  return skipPointer;
}
}
```

可以看出 高层在前，低层在后，除最低层外，其他层都有长度保存

4. FST
   FST（Finite State Transducer）。空间占用小。通过对词典中单词前缀和后缀的重复利用，压缩了存储空间。查询速度快。

http://examples.mikemccandless.com/fst.py?terms=hello world hi&cmd=Build+it!

共享前缀，后缀

.fdx .fdt

正向信息
缓存满了， flash到硬盘，使用了LZ4算法
http://cyan4973.github.io/lz4/

差值存储

负数的符号位都在最高位，而且PackedInts无法存储负数，因此需要对数据进行转码，转码方式就是zigzag转码

ZIgZag编码：
就是把负数位，移到最低位，节省空间
1000 0001变成0000 0011
32位 (i >> 31) ^ (i << 1)
64位 (l >> 63) ^ (l << 1)

```
public int readZInt() throws IOException {
  return BitUtil.zigZagDecode(readVInt());
}
```

merge

```
protected final void flush(boolean triggerMerge, boolean applyAllDeletes) throws IOException {

  // NOTE: this method cannot be sync'd because
  // maybeMerge() in turn calls mergeScheduler.merge which
  // in turn can take a long time to run and we don't want
  // to hold the lock for that.  In the case of
  // ConcurrentMergeScheduler this can lead to deadlock
  // when it stalls due to too many running merges.

  // We can be called during close, when closing==true, so we must pass false to ensureOpen:
  ensureOpen(false);
  if (doFlush(applyAllDeletes) && triggerMerge) {
    maybeMerge(config.getMergePolicy(), MergeTrigger.FULL_FLUSH, UNBOUNDED_MAX_MERGE_SEGMENTS);
  }
}
```

flush会触发是否进行段合并
MergePolicy 默认策略 LogMergePolicy， 选择哪些段应该参与合并
MergeScheduler 将选择出来的段合并成新段

```
public abstract class LogMergePolicy extends MergePolicy {
```

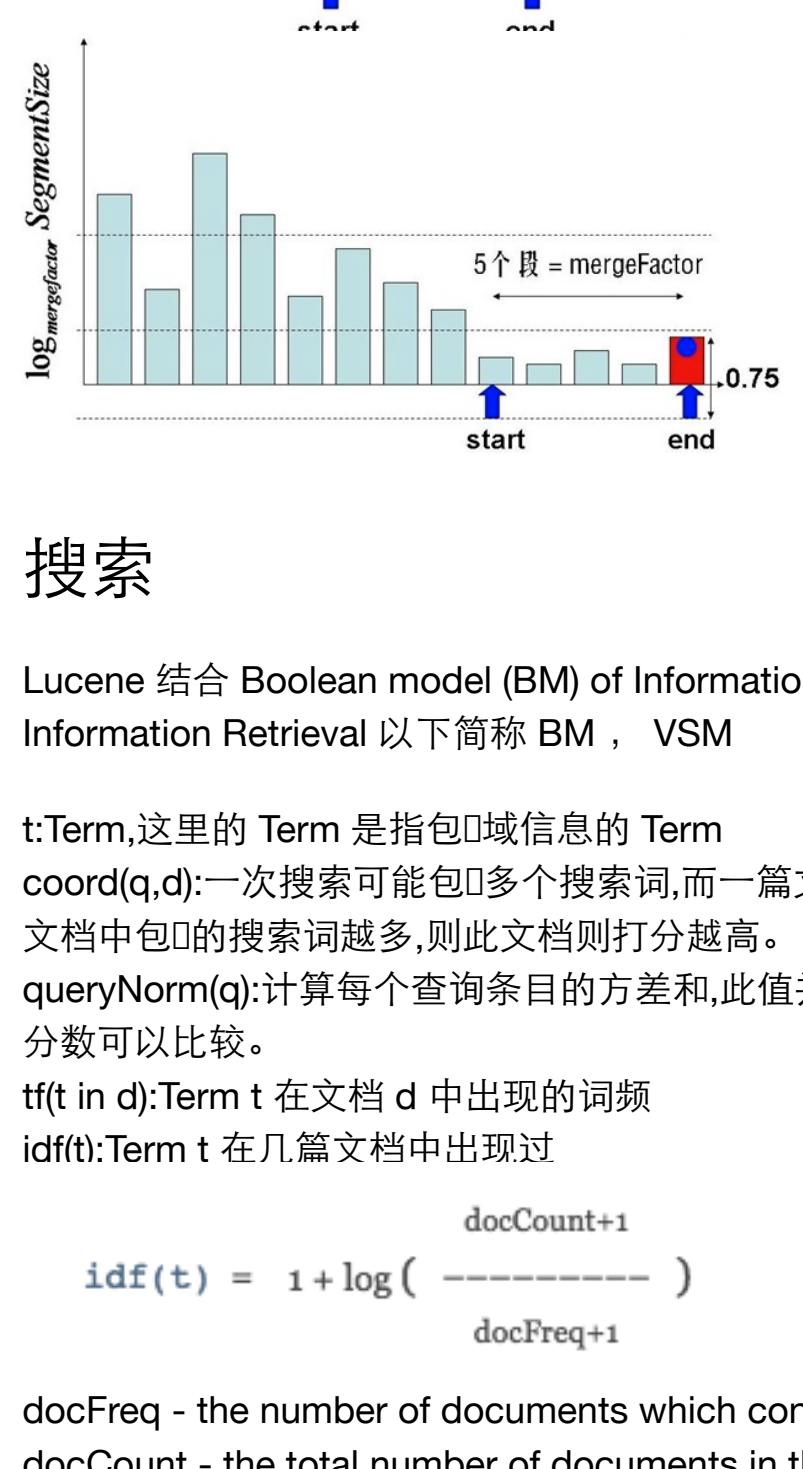
Defines the allowed range of log(size) for each level. A level is computed by taking the max segment log size, minus LEVEL_LOG_SPAN, and finding all segments falling within that range.

```
public static final double LEVEL_LOG_SPAN = 0.75;
```

Default merge factor, which is how many segments are merged at a time

```
public static final int DEFAULT_MERGE_FACTOR = 10;
```

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。



搜索

Lucene 结合 Boolean model (BM) of Information Retrieval 和 Vector Space Model (VSM) of Information Retrieval 以下简称 BM， VSM

t:Term,这里的 Term 是指包含域信息的 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。
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

$$\mathrm{idf}(t) = 1 + \log\left(\frac{\mathrm{docCount}+1}{\mathrm{docFreq}+1}\right)$$

docFreq - the number of documents which contain the term
docCount - the total number of documents in the collection
norm(t, d):标准化因子

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

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

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
f.getBoost: 此域权重越大,说明此域越重要

$$\text{cosine-similarity}(q,d) = \frac{V(q) \cdot V(d)}{|V(q)|\,|V(d)|}$$

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)*
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)

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

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

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

$$|\vec{V_d}| = \sqrt{w(t_1,d)^2 + w(t_2,d)^2 + \ldots + w(t_n,d)^2} = \sqrt{\sum_{t \in d} w(t,d)^2}$$

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1.所以得出文档长度：

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

打分公式变成了：

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

再加上各种boost和coord， lucene的打分公式：

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

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