# Lucene

## Lucene的主要构件

### 索引库Directory

Directory directory = **new** FSDirectory(Paths.*get*("E:\\lucene"));

### 分词器Analyzer

Analyzer analyzer = **new** StandardAnalyzer();

### 索引写入器IndexWriter

IndexWriterConfig config = **new** IndexWriterConfig(analyzer);

IndexWriter iwriter = **new** IndexWriter(directory, config);

### 索引文档Document

Document doc = **new** Document();

doc.add(**new** Field("name", ff.getName(), TextField.TYPE\_STORED));

iwriter.addDocument(doc);

iwriter.close();

### 索引查询库DirectoryReader

DirectoryReader ireader = DirectoryReader.open(directory);

### 索引查询器IndexSearcher

IndexSearcher isearcher = **new** IndexSearcher(ireader);

### QueryParser

QueryParser parser = **new** QueryParser("context", analyzer);

### Query

Query query = parser.parse("create");

TopDocs search = isearcher.search(query, 1000);

ScoreDoc[] scoreDocs = search.scoreDocs;

Document doc =isearcher.doc(scoreDocs[i].doc);

## ==================索引======================

## Directory 索引库

### 文件系统索引库FSDirectory

Directory directory = FSDirectory.*open*(Paths.*get*("E:/lucene "));

此方法将会自动打开一个最适合当前系统的的FSDirectory的具体实现类

### 内存索引库RAMDirectory

Directory directory = **new** RAMDirectory();

此方法开启一个内存索引库，好处是速度快，但是无法持久化

## Documen 文档

相当于数据库表里面的每一条记录

## Field 域 文档的域

相当于数据库记录中的列

### StringField 字符域

StringField field = **new** StringField(name, value, Store.YES);

字段索引,使用此域将会 不使用 分词器 来对字符值 进行分词,即不会对值进行任何拆分来进行索引，适用于 id 等字段，且推荐使用缓存Stroe.YES

### TextField 文本域

TextField text = **new** TextField(name, value, Store.NO);

大文本域，使用此域 默认使用分词器进行分词，使用于大文本的索引，由于是大文本，最好不要将数据全部缓存至索引库，推荐使用Store.NO，(默认使用分词器进行分词)

### 数值类型 域

IntField

LongField

FloatField

DoubleField

### Date类型的域 使用DateTools

## 查询索引器DirectoryReader

推荐使用单类模式

DirectoryReaderold = DirectoryReader.*open*(directory);

DirectoryReader new = DirectoryReader.*openIfChanged*(old);

**if** (**null** != new) {

old.close();

old = new;

}

根据索引库开启一个索引查询器

*openIfChanged*根据旧的索引器开启一个新的索引器，发生在当创建索引器成功后，又对索引库数据进行更新，此时所以器还是旧的索引器，需要重新开启，此方法会返回一个新的索引器，所过索引库未更新，则返回null;如果返回新的索引器组需要将旧索引器关闭，重新赋值为新索引器。

## 查询索引数据

IndexSearcher searcher = **new** IndexSearcher(*directoryReader*);

根据查询索引器开启索引查询

## 索引查询条件Query

### 基于分词器索引查询Query

QueryParser queryParser = **new** QueryParser("name", analyzer);

Query query = queryParser.parse(seachLike);

基于分词器的索引查询，索引库根据分词器生成的索引，如TextField生成的索引，需要基于此查询来进行查询，类似与 like 查询

### 不基于简单查询TermQuery

Query query = **new** TermQuery(**new** Term("name", seach));

此查询将使用简单的查询来进行查询，如果生成索引 未使用分词器，则此方法需要全部匹配才能查询到结果

## 索引的删除基于IndexWriter

indexWriter.deleteAll();

此方法将会删除索引库的全部索引，请慎用此方法 5.0版本不能被恢复。

在indexWriter.Commit()或indexWriter. Close()后将不能被恢复

在为进行数据提交前 可以使用 indexWriter.rollback();进行回滚

## 索引的更新IndexWriter

*indexWriter*.updateDocument(term, doc);

索引的更新 相当于 先删除Term查询条件索引 后创建新的索引，删除的索引会位于回收站的位置，不会彻底删除

*indexWriter*.forceMergeDeletes();

会将回收站的索引彻底删除(测试未成功)

## 为索引添加时进行加权操作setBoot

TextField text = **new** TextField("context", body, Store.*NO*);

textField.setBoost(float1);

为域进行加权操作，加权值为float 默认为1f

## ==================查询==========================

## 基于Term的简单查询

TopDocs search = indexSearch.search(query, num);

num为最大查询的文档数量

**int** totalHits = search.totalHits;

totalHits 匹配的文档数量

### 简单查询TermQuery

IndexSearcher indexSearch = **new** IndexSearcher(*reader*);

Query query = **new** TermQuery(**new** Term(field, value));

如果生成索引时，未使用分词器，如StringField生成的索引，在查询时，必须匹配全部字段，才会有查询结果

### 11.2 简单区间查询TermRangeQuery

IndexSearcher indexSearch = **new** IndexSearcher(*reader*);

Query query = TermRangeQuery.*newStringRange*(field, start, end, **true**, **true**);

### 简单区间数值类型查询NumericRangeQuery

IndexSearcher indexSearch = **new** IndexSearcher(*reader*);

Query query = NumericRangeQuery.*newLongRange*(field, start, end, **true**, **true**);

注意此处的查询，必须匹配创建索引时的索引域 值的类型，如创建索引 域类型为IntField 查询索引域类型为 LongRanage 即使 查询的值在int范围内也查询不到结果

### 前缀搜索PrefixQuery

Query query = **new** PrefixQuery(**new** Term(field, value));

基于索引值的前缀搜索，此Query会对索引值进行前缀匹配，对于未进行分词器生成的索引 仍然 可以搜索到结果

### 通配符搜索WildcardQuery

Query query = **new** WildcardQuery(**new** Term(field, wild));

支持通配符的查询 前缀搜索即基于此实现

此查询速度较慢，需要遍历很多方面，建议不要以 通配符 为开始的关键字查询

\*表示任意字符， ?表示单个字符， \表示转义字符

### 连接多条件的BooleanQuery

Builder builder = **new** BooleanQuery.Builder();

builder.add(**new** TermQuery(**new** Term("context", "select")), Occur.*MUST*);

builder.add(**new** TermQuery(**new** Term("context", "like")), Occur.*MUST*);

BooleanQuery booleanQuery = builder.build();

BooleanQuery连接多个条件进行查询

Occur.*MUST* 必须存在

Occur.*SHOULD* 可以存在

Occur*.MUST\_NOT* 必须不存在

### 短语查询PhraseQuery

PhraseQuery query = **new** PhraseQuery(1, "context", "public", "double");

此方法适用于 知道前后的几个值，不知道中间的几个值，第一个int类型参数，表示中间可以 跳词 的数量，第二个表示查询的字段 后面的表示查询的值

### 模糊查询FuzzyQuery

FuzzyQuery query = **new** FuzzyQuery(**new** Term("context", "seleck"));

此方法使用与 索引的实际值 与传入的查询值可以不匹配,最大支持一个 词中有2个字符不匹配

## 基于QueryParser的搜索

### QueryParser的创建

QueryParser queryParser = **new** QueryParser("context", **new** StandardAnalyzer());

创建Queryparser，并且设置默认的搜索域

### 基于QueryParser的各种方式查询

// 包含以下内容 为OR 的 或 关系 默认空格为OR

query = queryParser.parse("select from");

// 可以更改默认为AND 关系

// queryParser.setDefaultOperator(Operator.AND);

// 包含以下内容 为 AND 的 并且 关系

query = queryParser.parse("select AND from AND appr");

// 改变搜索域的位置

query = queryParser.parse("name:select.sql");

// 使用通配符进行 查询 ? \*

query = queryParser.parse("selec?");

// + 表示域中必须有 -必须不含有

query = queryParser.parse(" + name:教师\*");

// 闭区间查询

query = queryParser.parse("[public select]");

// 开区间查询

query = queryParser.parse("{public select}");

// 完全匹配字符串

query = queryParser.parse("\"stu.historyclassinfo like\"");

// 匹配 字之间 有一个 距离

query = queryParser.parse("\"LEFT BASE\_USERINFO\"~1");

// 模糊查询

query = queryParser.parse("BASE\_USERIxX~");

## 查询分页

### 使用subList思想实现分页

### 使用SearchAfter实现分页

**public** **void** seachePage(String query, **int** pageIndex, **int** pageSize) {

**try** {

IndexSearcher indexSearch = **new** IndexSearcher(*reader*);

QueryParser parser = **new** QueryParser("context", *analyzer*);

Query q = parser.parse(query);

**int** prePageLastIndex = (pageIndex - 1) \* pageSize;

**if** (prePageLastIndex == 0) {

prePageLastIndex = pageSize;

}

TopDocs topDocs = indexSearch.search(q, prePageLastIndex);

ScoreDoc scoreDoc = topDocs.scoreDocs[prePageLastIndex - 1];

**if** (pageIndex == 1) {

scoreDoc = **null**;

}

TopDocs searchAfter = indexSearch.searchAfter(scoreDoc, q, pageSize);

looger.info("一个匹配到了" + searchAfter.totalHits + "篇文档");

ScoreDoc[] scoreDocs = searchAfter.scoreDocs;

**for** (ScoreDoc sd : scoreDocs) {

Document doc = indexSearch.doc(sd.doc);

looger.info("查询到文档[" + doc.get("name") + "]");

}

} **catch** (ParseException e) {

e.printStackTrace();

} **catch** (IOException e) {

e.printStackTrace();

}

}

## ===================项目使用=================

## 是否进行实时搜索

### 完全的实时搜索

只要数据库一变动，马上要更新索引，需要writer.commit()来进行操作，效率不高。

### 近实时搜索

当修改了信息之后，先报索引保存到内存中，然后在一个统一的时间对内存中的所有的索引进行提交操作。

NRTCachingDirectory cacheFsDir = **new** NRTCachingDirectory(directory, 5d, 60d);

IndexWriter writer = **new** IndexWriter(cacheFsDir, **new** IndexWriterConfig(analyzer));

TrackingIndexWriter trackingIndexWriter = **new** TrackingIndexWriter(writer);

SearcherManager *searcherManager* =

**new** SearcherManager(writer, **true**, **new** SearcherFactory());

ControlledRealTimeReopenThread<IndexSearcher> crtThread =

**new** ControlledRealTimeReopenThread<IndexSearcher>

(trackingIndexWriter, *searcherManager*, 5.0, 0.025);

crtThread.setDaemon(**true**);

crtThread.setName("ControlledRealTimeReopenThread");

crtThread.start();