**Lucene**

**全文检索技术**

# 课程计划

1. Lucene的Field
2. Lucene的索引库维护
3. lucene的查询
   1. Query子对象
   2. QueryParser
4. Lucene相关度排序
5. solr介绍
6. solr安装配置
7. Solrj的使用

# Field域

## Field属性

Field是文档中的域，包括Field名和Field值两部分，一个文档可以包括多个Field，Document只是Field的一个承载体，Field值即为要索引的内容，也是要搜索的内容。

* **是否分词(tokenized)**

是：作分词处理，即将Field值进行分词，分词的目的是为了索引。

比如：商品名称、商品描述等，这些内容用户要输入关键字搜索，由于搜索的内容格式大、内容多需要分词后将语汇单元建立索引

否：不作分词处理

比如：商品id、订单号、身份证号等

* **是否索引(indexed)**

是：进行索引。将Field分词后的词或整个Field值进行索引，存储到索引域，索引的目的是为了搜索。

比如：商品名称、商品描述分析后进行索引，订单号、身份证号不用分词但也要索引，这些将来都要作为查询条件。

否：不索引。

比如：图片路径、文件路径等，不用作为查询条件的不用索引。

* **是否存储(stored)**

是：将Field值存储在文档域中，存储在文档域中的Field才可以从Document中获取。

比如：商品名称、订单号，凡是将来要从Document中获取的Field都要存储。

否：不存储Field值

比如：商品描述，内容较大不用存储。如果要向用户展示商品描述可以从系统的关系数据库中获取。

## Field常用类型

下边列出了开发中常用 的Filed类型，注意Field的属性，根据需求选择：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Field类 | 数据类型 | Analyzed  是否分词 | Indexed  是否索引 | Stored  是否存储 | 说明 |
| StringField(FieldName, FieldValue,Store.YES)) | 字符串 | N | Y | Y或N | 这个Field用来构建一个字符串Field，但是不会进行分词，会将整个串存储在索引中，比如(订单号,身份证号等)  是否存储在文档中用Store.YES或Store.NO决定 |
| LongField(FieldName, FieldValue,Store.YES) | Long型 | Y | Y | Y或N | 这个Field用来构建一个Long数字型Field，进行分词和索引，比如(价格)  是否存储在文档中用Store.YES或Store.NO决定 |
| StoredField(FieldName, FieldValue) | 重载方法，支持多种类型 | N | N | Y | 这个Field用来构建不同类型Field  不分析，不索引，但要Field存储在文档中 |
| TextField(FieldName, FieldValue, Store.NO)  或  TextField(FieldName, reader) | 字符串  或  流 | Y | Y | Y或N | 如果是一个Reader, lucene猜测内容比较多,会采用Unstored的策略. |

## Field修改

### 修改分析

图书id：

是否分词：不用分词，因为不会根据商品id来搜索商品

是否索引：不索引，因为不需要根据图书ID进行搜索

是否存储：要存储，因为查询结果页面需要使用id这个值。

图书名称：

是否分词：要分词，因为要根据图书名称的关键词搜索。

是否索引：要索引。

是否存储：要存储。

图书价格：

是否分词：要分词，lucene对数字型的值只要有搜索需求的都要分词和索引，因 为lucene对数字型的内容要特殊分词处理，需要分词和索引。

是否索引：要索引

是否存储：要存储

图书图片地址：

是否分词：不分词

是否索引：不索引

是否存储：要存储

图书描述：

是否分词：要分词

是否索引：要索引

是否存储：因为图书描述内容量大，不在查询结果页面直接显示，不存储。

不存储是不在lucene的索引域中记录，节省lucene的索引文件空间。

如果要在详情页面显示描述，解决方案：

从lucene中取出图书的id，根据图书的id查询关系数据库（MySQL）中book表得到描述信息。

### 代码修改

对之前编写的testCreateIndex()方法进行修改。

代码片段

// Document文档中添加域

// 图书Id

// Store.YES:表示存储到文档域中

// 不分词，不索引，储存

document.add(**new** StoredField("id", book.getId().toString()));

// 图书名称

// 分词，索引，储存

document.add(**new** TextField("name", book.getName().toString(), Store.***YES***));

// 图书价格

// 分词，索引，储存

document.add(**new** FloatField("price", book.getPrice(), Store.***YES***));

// 图书图片地址

// 不分词，不索引，储存

document.add(**new** StoredField("pic", book.getPic().toString()));

// 图书描述

// 分词，索引，不储存

document.add(**new** TextField("desc", book.getDesc().toString(), Store.***NO***));

# 索引维护

## 需求

管理人员通过电商系统更改图书信息，这时更新的是关系数据库，如果使用lucene搜索图书信息，需要在数据库表book信息变化时及时更新lucene索引库。

## 添加索引

调用 indexWriter.addDocument（doc）添加索引。

参考入门程序的创建索引。

## 删除索引

### 删除指定索引

根据**Term项**删除索引，满足条件的将全部删除。

@Test

**public** **void** testIndexDelete() **throws** Exception {

// 创建Directory流对象

Directory directory = FSDirectory.*open*(**new** File("C:/itcast/lucene/index"));

IndexWriterConfig config = **new** IndexWriterConfig(Version.***LUCENE\_4\_10\_3***, **null**);

// 创建写入对象

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

// 根据Term删除索引库，name:java

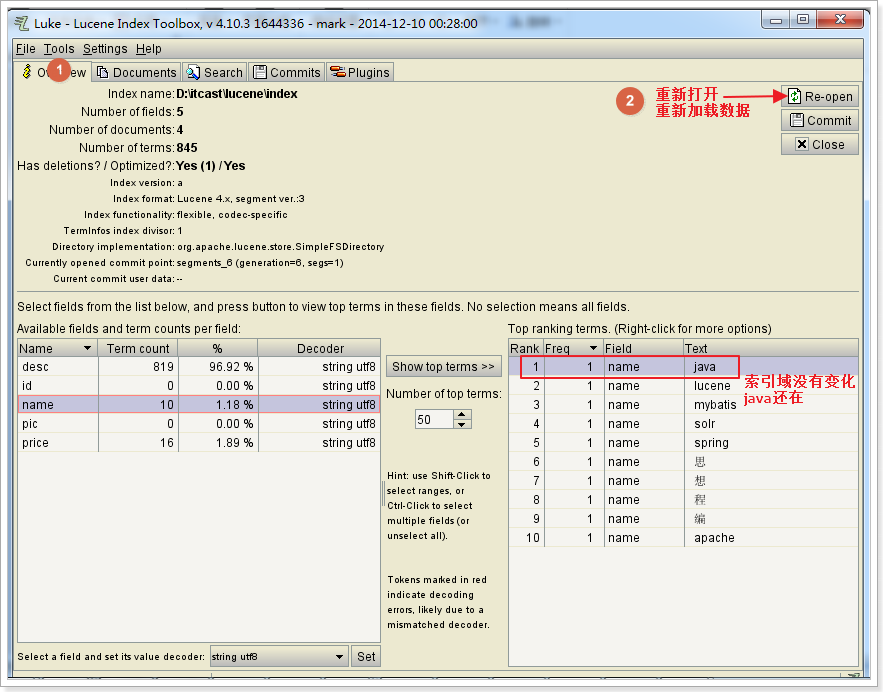
indexWriter.deleteDocuments(**new** Term("name", "java"));

// 释放资源

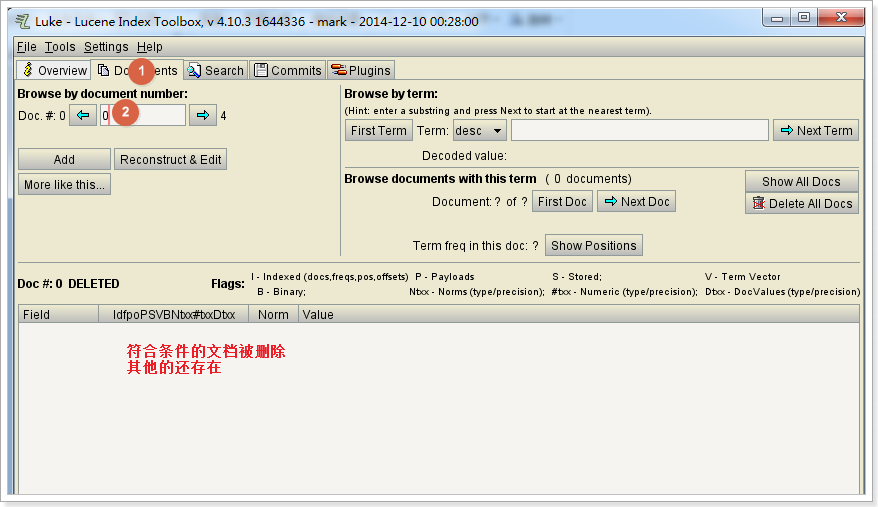
indexWriter.close();

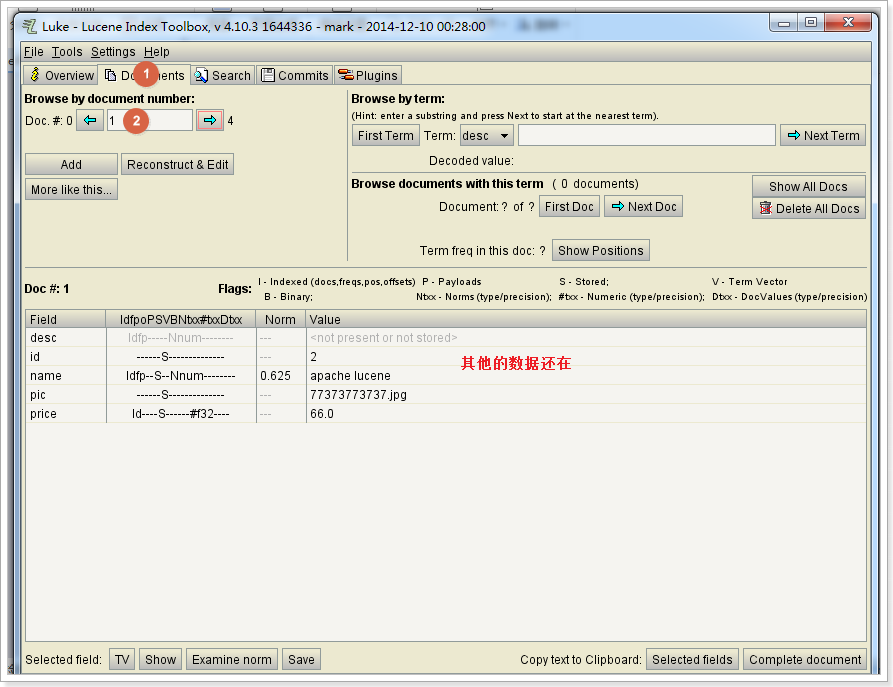
}

效果如下图：索引域没有变化



文档域数据被删除掉





### 删除全部索引（慎用）

将索引目录的索引信息全部删除，直接彻底删除，无法恢复。

建议参照关系数据库基于主键删除方式，所以在创建索引时需要创建一个主键Field，删除时根据此主键Field删除。

索引删除后将放在Lucene的回收站中，Lucene3.X版本可以恢复删除的文档，3.X之后无法恢复。

代码：

@Test

**public** **void** testIndexDelete() **throws** Exception {

// 创建Directory流对象

Directory directory = FSDirectory.*open*(**new** File("D:/itcast/lucene/index"));

IndexWriterConfig config = **new** IndexWriterConfig(Version.***LUCENE\_4\_10\_3***, **null**);

// 创建写入对象

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

// 根据Term删除索引库，name:java

// indexWriter.deleteDocuments(new Term("name", "java"));

// 全部删除

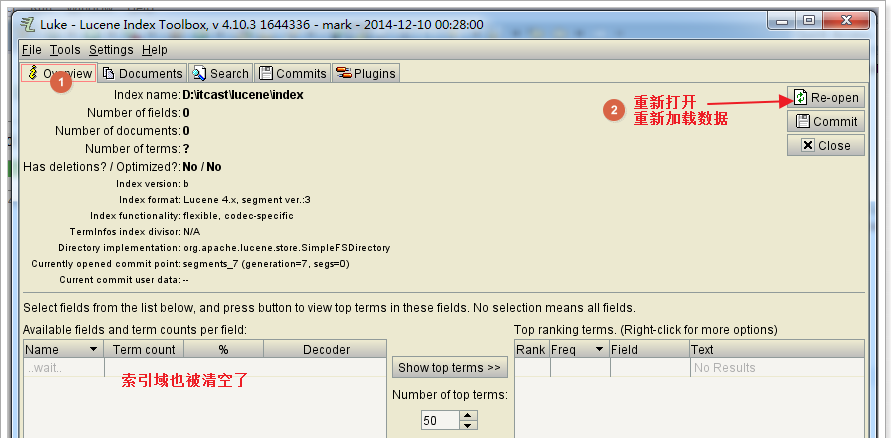
indexWriter.deleteAll();

// 释放资源

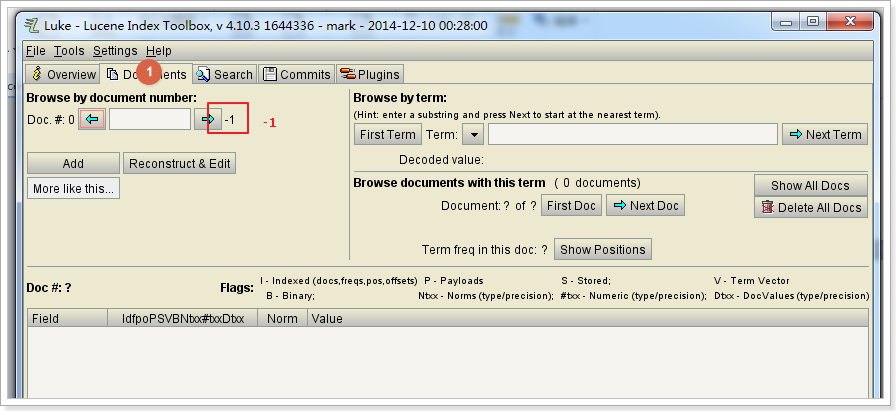
indexWriter.close();

}

索引域数据清空



文档域数据也清空



## 修改索引

更新索引是先删除再添加，建议对更新需求采用此方法并且要保证对已存在的索引执行更新，可以先查询出来，确定更新记录存在执行更新操作。

如果更新索引的目标文档对象不存在，则执行添加。

代码

@Test

**public** **void** testIndexUpdate() **throws** Exception {

// 创建分词器

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

// 创建Directory流对象

Directory directory = FSDirectory.*open*(**new** File("C:/itcast/lucene/index"));

IndexWriterConfig config = **new** IndexWriterConfig(Version.***LUCENE\_4\_10\_3***, analyzer);

// 创建写入对象

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

// 创建Document

Document document = **new** Document();

document.add(**new** TextField("id", "1002", Store.***YES***));

document.add(**new** TextField("name", "lucene测试test 002", Store.***YES***));

// 执行更新，会把所有符合条件的Document删除，再新增。

indexWriter.updateDocument(**new** Term("name", "test"), document);

// 释放资源

indexWriter.close();

}

# 搜索

## 创建查询的两种方法

对要搜索的信息创建Query查询对象，Lucene会根据Query查询对象生成最终的查询语法。类似关系数据库Sql语法一样，Lucene也有自己的查询语法，比如：“name:lucene”表示查询名字为name的Field域中的“lucene”的文档信息。

可通过两种方法创建查询对象：

1）使用Lucene提供Query子类

Query是一个抽象类，lucene提供了很多查询对象，比如TermQuery项精确查询，NumericRangeQuery数字范围查询等。

如下代码：

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

2）使用QueryParse解析查询表达式

QueryParser会将用户输入的查询表达式解析成Query对象实例。

如下代码：

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

Query query = queryParser.parse("name:lucene");

## 通过Query子类搜索

### TermQuery

TermQuery词项查询，TermQuery不使用分析器，搜索关键词进行精确匹配Field域中的词，比如订单号、分类ID号等。

搜索对象创建：

@Test

**public** **void** testSearchTermQuery() **throws** Exception {

// 创建TermQuery搜索对象

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

doSearch(query);

}

抽取搜索逻辑：

**private** **void** doSearch(Query query) **throws** IOException {

// 2. 执行搜索，返回结果集

// 创建Directory流对象

Directory directory = FSDirectory.*open*(**new** File("D:/itcast/lucene/index"));

// 创建索引读取对象IndexReader

IndexReader reader = DirectoryReader.*open*(directory);

// 创建索引搜索对象

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

// 使用索引搜索对象，执行搜索，返回结果集TopDocs

// 第一个参数：搜索对象，第二个参数：返回的数据条数，指定查询结果最顶部的n条数据返回

TopDocs topDocs = searcher.search(query, 10);

System.***out***.println("查询到的数据总条数是：" + topDocs.totalHits);

// 获取查询结果集

ScoreDoc[] docs = topDocs.scoreDocs;

// 解析结果集

**for** (ScoreDoc scoreDoc : docs) {

// 获取文档id

**int** docID = scoreDoc.doc;

Document doc = searcher.doc(docID);

System.***out***.println("======================================");

System.***out***.println("docID:" + docID);

System.***out***.println("bookId:" + doc.get("id"));

System.***out***.println("name:" + doc.get("name"));

System.***out***.println("price:" + doc.get("price"));

System.***out***.println("pic:" + doc.get("pic"));

// System.out.println("desc:" + doc.get("desc"));

}

// 3. 释放资源

reader.close();

}

### NumericRangeQuery

NumericRangeQuery，指定数字范围查询.

@Test

**public** **void** testSearchNumericRangeQuery() **throws** Exception {

// 创建NumericRangeQuery搜索对象,数字范围查询.

// 五个参数分别是：域名、最小值、最大值、是否包含最小值，是否包含最大值

Query query = NumericRangeQuery.*newFloatRange*("price", 54f, 56f, **false**, **true**);

doSearch(query);

}

### BooleanQuery

BooleanQuery，布尔查询，实现组合条件查询。

@Test

**public** **void** testSearchBooleanQuery() **throws** Exception {

// 创建TermQuery搜索对象

Query query1 = **new** TermQuery(**new** Term("name", "lucene"));

// 创建NumericRangeQuery搜索对象,数字范围查询.

// 四个参数分别是：域名、最小值、最大值、是否包含最小值，是否包含最大值

Query query2 = NumericRangeQuery.*newFloatRange*("price", 54f, 66f, **false**, **true**);

// 创建BooleanQuery搜索对象,组合查询条件

BooleanQuery boolQuery = **new** BooleanQuery();

// 组合条件,

// 第一个参数，查询条件，第二个参数，组合方式

boolQuery.add(query1, Occur.***MUST\_NOT***);

boolQuery.add(query2, Occur.***MUST***);

doSearch(boolQuery);

}

组合关系代表的意思如下:

1、MUST和MUST表示“与”的关系，即“交集”。

2、MUST和MUST\_NOT前者包含后者不包含。

3、MUST\_NOT和MUST\_NOT没意义

4、SHOULD与MUST表示MUST，SHOULD失去意义；

5、SHOULD与MUST\_NOT相当于MUST与MUST\_NOT。

6、SHOULD与SHOULD表示“或”的关系，即“并集”。

## 通过QueryParser搜索

通过QueryParser也可以创建Query，QueryParser提供一个Parse方法，此方法可以直接根据查询语法来查询。可以通过打印Query对象的方式，查看生成的查询语句。

### 查询语法

1、基础的查询语法，关键词查询：

域名+“:”+搜索的关键字

例如：name:java

1. 范围查询

域名+“:”+[最小值 TO 最大值]

例如：size:[1 TO 1000]

注意：QueryParser不支持对数字范围的搜索，它支持字符串范围。数字范围搜索建议使用NumericRangeQuery。

1. 组合条件查询

|  |  |
| --- | --- |
| Occur.MUST 查询条件必须满足，相当于AND | +（加号） |
| Occur.SHOULD 查询条件可选，相当于OR | 空（不用符号） |
| Occur.MUST\_NOT 查询条件不能满足，相当于NOT非 | -（减号） |

### QueryParser

@Test

**public** **void** testSearchIndex() **throws** Exception {

// 创建分词器

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

// 1. 创建Query搜索对象

// 创建搜索解析器，第一个参数：默认Field域，第二个参数：分词器

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

// 创建搜索对象

// Query query = queryParser.parse("desc:java学习");

Query query = queryParser.parse("desc:java AND lucene");

// 打印生成的搜索语句

System.***out***.println(query);

// 执行搜索

doSearch(query);

}

### MultiFieldQueryParser

通过MultiFieldQueryParse对多个域查询。

@Test

**public** **void** testSearchMultiFieldQueryParser() **throws** Exception {

// 创建分词器

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

// 1. 创建MultiFieldQueryParser搜索对象

String[] fields = { "name", "desc" };

MultiFieldQueryParser multiFieldQueryParser = **new** MultiFieldQueryParser(fields, analyzer);

// 创建搜索对象

Query query = multiFieldQueryParser.parse("lucene");

// 打印生成的搜索语句

System.***out***.println(query);

// 执行搜索

doSearch(query);

}

生成的查询语句：

name:lucene desc:lucene

## TopDocs

Lucene搜索结果可通过TopDocs遍历，TopDocs类提供了少量的属性，如下：

|  |  |
| --- | --- |
| 方法或属性 | 说明 |
| totalHits | 匹配搜索条件的总记录数 |
| scoreDocs | 顶部匹配记录 |

注意：

Search方法需要指定匹配记录数量n：indexSearcher.search(query, n)

TopDocs.totalHits：是匹配索引库中所有记录的数量

TopDocs.scoreDocs：匹配相关度高的前边记录数组，scoreDocs的长度小于等于search方法指定的参数n

# 相关度排序(了解)

## 什么是相关度排序

相关度排序是查询结果按照与查询关键字的相关性进行排序，越相关的越靠前。比如搜索“Lucene”关键字，与该关键字最相关的文章应该排在前边。

## 相关度打分

Lucene对查询关键字和索引文档的相关度进行打分，得分高的就排在前边。如何打分呢？Lucene是在用户进行检索时实时根据搜索的关键字计算出来的，分两步：

1）计算出词（Term）的权重

2）根据词的权重值，计算文档相关度得分。

什么是词的权重？

通过索引部分的学习，明确索引的最小单位是一个Term(索引词典中的一个词)。搜索也是从索引域中查询Term，再根据Term找到文档。Term对文档的重要性称为权重，影响Term权重有两个因素：

* Term Frequency (tf)：

指此Term在此文档中出现了多少次。tf 越大说明越重要。

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“Lucene”这个词，在文档中出现的次数很多，说明该文档主要就是讲Lucene技术的。

* Document Frequency (df)：

指有多少文档包含此Term。df 越大说明越不重要。

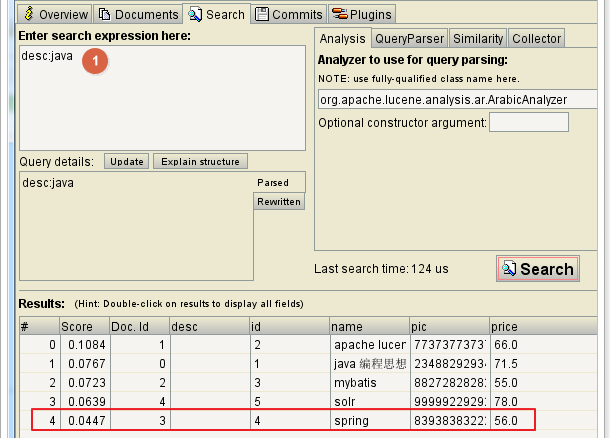
比如，在一篇英语文档中，this出现的次数更多，就说明越重要吗？不是的，有越多的文档包含此词(Term), 说明此词(Term)太普通，不足以区分这些文档，因而重要性越低。

## 设置boost值影响相关度排序

boost是一个加权值（默认加权值为1.0f），它可以影响权重的计算。在索引时对某个文档中的field设置加权值，设置越高，在搜索时匹配到这个文档就可能排在前边。

未设置权重：

希望把name为spring的排名提高



先清空索引库，然后修改创建索引的代码，添加设置加权值的逻辑

修改创建索引代码：

@Test

public void testCreateIndex() throws Exception {

// 1. 采集数据

BookDao bookDao = new BookDaoImpl();

List<Book> bookList = bookDao.queryBookList();

// 2. 创建Document文档对象

List<Document> documents = new ArrayList<>();

for (Book book : bookList) {

Document document = new Document();

// Document文档中添加域

// 图书Id

// Store.YES:表示存储到文档域中

// 不分词，不索引，储存

document.add(new StoredField("id", book.getId().toString()));

// 图书名称

// 分词，索引，储存

document.add(new TextField("name", book.getName().toString(), Store.*YES*));

// 图书价格

// 分词，索引，储存

document.add(new FloatField("price", book.getPrice(), Store.*YES*));

// 图书图片地址

// 不分词，不索引，储存

document.add(new StoredField("pic", book.getPic().toString()));

// 图书描述

// 分词，索引，不储存

TextField descField = new TextField("desc", book.getDesc().toString(), Store.*NO*);

// 给id为4的文档设置加权值

if (4 == book.getId()) {

descField.setBoost(100f);

}

document.add(descField);

// 把Document放到list中

documents.add(document);

}

// 3. 创建Analyzer分词器,分析文档，对文档进行分词

Analyzer analyzer = new StandardAnalyzer();

// 4. 创建Index Write，需要directory流对象

// 创建流对象

Directory directory = FSDirectory.*open*(new File("D:/itcast/lucene/index"));

// 创建IndexWriteConfig对象

IndexWriterConfig config = new IndexWriterConfig(Version.*LUCENE\_4\_10\_3*, analyzer);

// 创建IndexWriter写入对象

IndexWriter indexWriter = new IndexWriter(directory, config);

// 通过IndexWriter添加文档对象document

for (Document doc : documents) {

indexWriter.addDocument(doc);

}

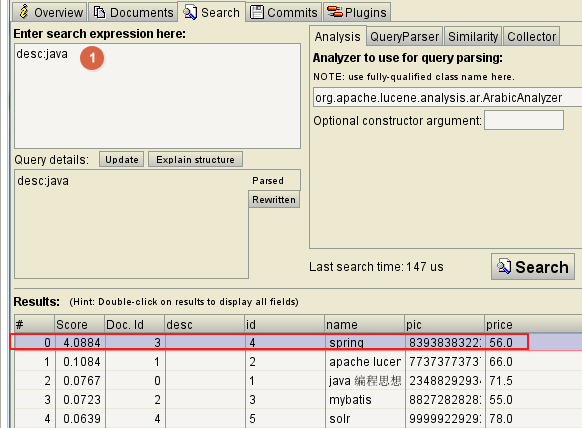
// 释放资源

indexWriter.close();

}

执行创建索引的逻辑，使用luke重载新生成的索引库，再次查询spring在第一

查询结果：



# Solr介绍

## 什么是solr

Solr 是Apache下的一个顶级开源项目，采用Java开发，它是基于Lucene的全文搜索服务。Solr可以独立运行在Jetty、Tomcat等这些Servlet容器中。

使用Solr 进行创建索引和搜索索引的实现方法很简单，如下：

* 创建索引：客户端（可以是浏览器可以是Java程序）用 POST 方法向 Solr 服务器发送一个描述 Field 及其内容的 XML 文档，Solr服务器根据xml文档添加、删除、更新索引 。
* 搜索索引：客户端（可以是浏览器可以是Java程序）用 GET方法向 Solr 服务器发送请求，然后对 Solr服务器返回Xml、json等格式的查询结果进行解析。Solr不提供构建页面UI的功能。Solr提供了一个管理界面，通过管理界面可以查询Solr的配置和运行情况。

Solr是一个可以独立运行的搜索服务器，使用solr进行全文检索服务的话，只需要通过http请求访问该服务器即可。

## Solr和Lucene的区别

Lucene是一个开放源代码的全文检索引擎工具包，它不是一个完整的全文检索应用。Lucene仅提供了完整的查询引擎和索引引擎，目的是为软件开发人员提供一个简单易用的工具包，以方便的在目标系统中实现全文检索的功能，或者以Lucene为基础构建全文检索应用。

Solr的目标是打造一款企业级的搜索引擎系统，它是基于Lucene一个搜索引擎服务，可以独立运行，通过Solr可以非常快速的构建企业的搜索引擎，通过Solr也可以高效的完成站内搜索功能。



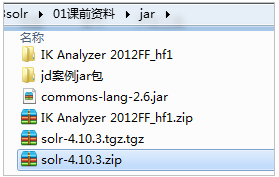
# Solr安装配置

## 下载solr

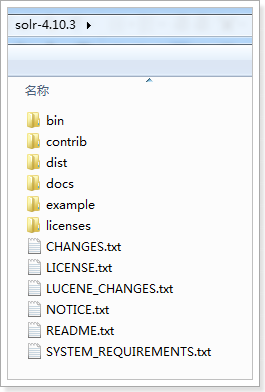
Solr和lucene的版本是同步更新的，本课程使用的版本：4.10.3

下载地址：http://archive.apache.org/dist/lucene/solr/

Linux下需要solr-4.10.3.tgz，windows下需要solr-4.10.3.zip。



解压solr-4.10.3.zip：



bin：solr的运行脚本

contrib：solr的一些扩展jar包，用于增强solr的功能。

dist：该目录包含build过程中产生的war和jar文件，以及相关的依赖文件。

docs：solr的API文档

example：solr工程的例子目录：

* example/solr：

该目录是一个标准的SolrHome，它包含一个默认的SolrCore

* example/multicore：

该目录包含了在Solr的multicore中设置的多个Core目录。

* example/webapps：

该目录中包括一个solr.war，该war可作为solr的运行实例工程。

licenses：solr相关的一些许可信息

## 运行环境

solr 需要运行在一个Servlet容器中，Solr4.10.3要求jdk使用1.7以上，Solr默认提供Jetty（java写的Servlet容器），

使用jetty启动：

使用cmd命令行，进入example文件夹启动

启动命令java -jar start.jar



启动后访问地址：http://127.0.0.1:8983/solr

但是企业中一般使用Tomcat作为服务器，本课程也是一样，

相关环境如下：

* Solr：4.10.3
* Jdk环境：1.7（solr4.10 不能使用jdk1.7以下）
* 服务器：Tomcat 7

## SolrCore配置

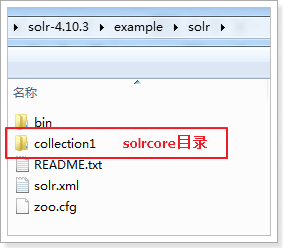
### SolrHome和SolrCore

SolrHome是Solr服务运行的主目录，该目录中包括了多个SolrCore目录。SolrCore目录中包含了运行Solr实例所有的配置文件和数据文件，Solr实例就是SolrCore。

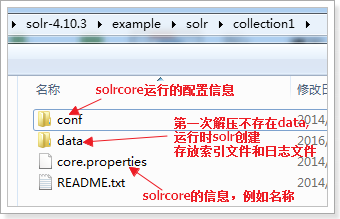
每个SolrCore提供单独的搜索和索引服务。

#### 目录结构

SolrHome目录：



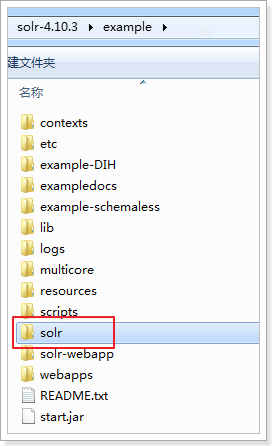
SolrCore目录：



### 创建SolrCore

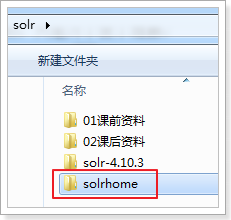
创建SolrCore先要创建SolrHome。在solr解压包下solr-4.10.3\example\solr文件夹就是一个标准的SolrHome，只需要将它复制到指定的目录下即可。

拷贝solr解压包下solr-4.10.3\example\solr文件夹。

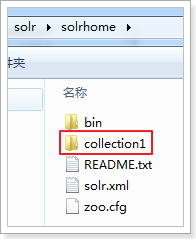


复制该文件夹到本地的一个目录，把文件名称改为solrhome。

改名不是必须的，只是为了便于理解

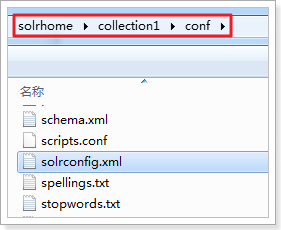


打开solrhome目录确认solrcore



### 配置SolrCore

其实就是配置SolrCore目录下的conf/solrconfig.xml。

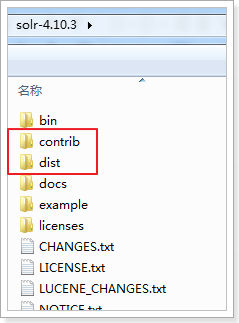


这个文件是来配置SolrCore实例的相关信息。如果使用默认配置可以不用做任何修改。它里面包含了不少标签，但是我们经常使用的标签为：lib标签、datadir标签、requestHandler标签。

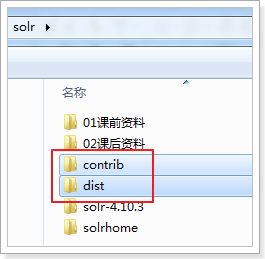
#### lib 标签

在solrconfig.xml中可以加扩展载一些的jar，如果需要使用，则首先要把这些jar复制到指定的目录，我们复制到SolrHome同级目录

复制之前解压的文件夹中的contrib和dist文件夹



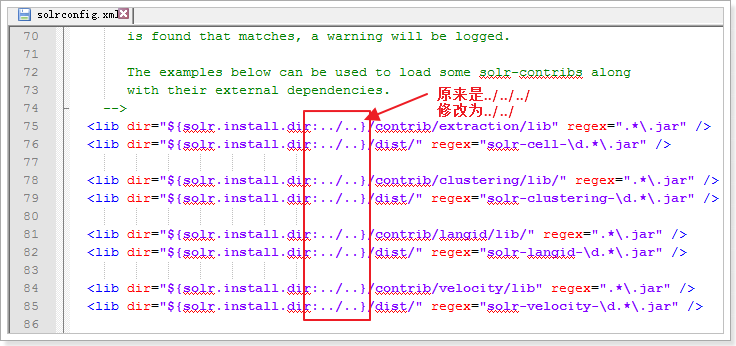
粘贴到SolrHome同级目录下。



修改solrconfig.xml配置文件加载扩展的jar。

configsolr.install.dir表示${SolrCore}的目录位置，需要如下修改：

./ 表示当前目录 ../表示上一级目录

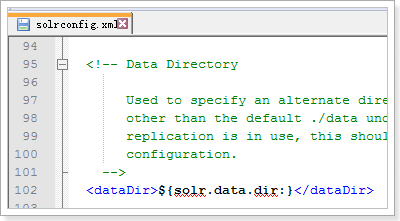


#### datadir标签

配置SolrCore的data目录。

data目录用来存放SolrCore的索引文件和tlog日志文件

solr.data.dir表示${SolrCore}/data的目录位置



如果不想使用默认的目录也可以通过solrconfig.xml更改索引目录 ，

例如：

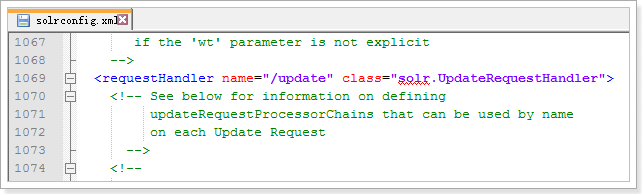


(建议不修改，否则配置多个SolrCore会报错）

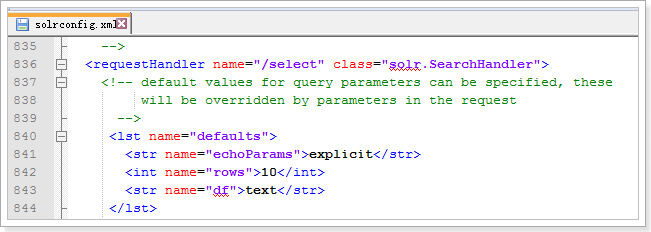
#### requestHandler标签

requestHandler请求处理器，定义了索引和搜索的访问方式。

通过/update维护索引，可以完成索引的添加、修改、删除操作。



通过/select搜索索引。



设置搜索参数完成搜索，搜索参数也可以设置一些默认值，如下：

<requestHandler name="/select" class="solr.SearchHandler">

<!-- 设置默认的参数值，可以在请求地址中修改这些参数-->

<lst name="defaults">

<str name="echoParams">explicit</str>

<int name="rows">10</int><!--显示数量-->

<str name="wt">json</str><!--显示格式-->

<str name="df">text</str><!--默认搜索字段-->

</lst>

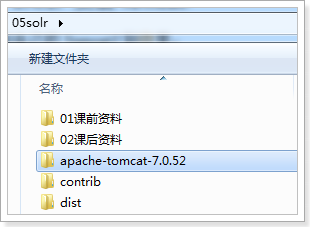
</requestHandler>

## Solr工程部署

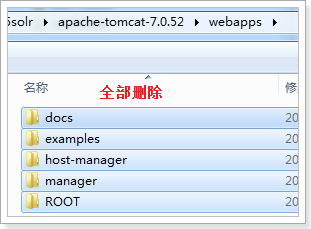
由于在项目中用到的web服务器大多数是用的Tomcat，所以就进行solr和Tomcat的整合。

### 安装Tomcat

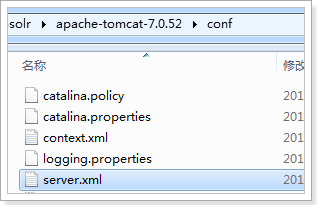
复制自己的Tomcat7到这里



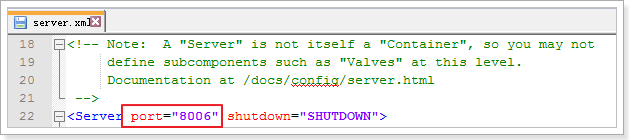
删除不用的应用（可以不删）

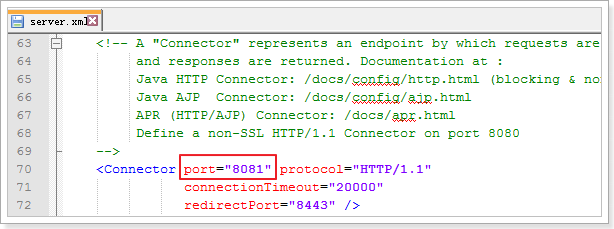


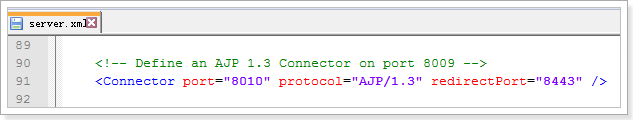
修改server.xml配置文件里面的端口号（否则后面eclipse使用Tomcat会冲突）



修改以下三个端口号



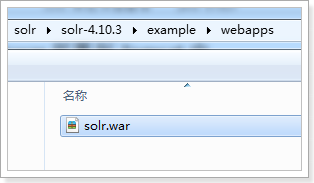




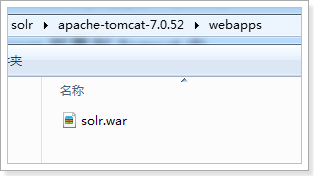
### 部署solr.war到Tomcat中

1. 从solr解压包下的solr-4.10.3\example\webapps目录中拷贝solr.war

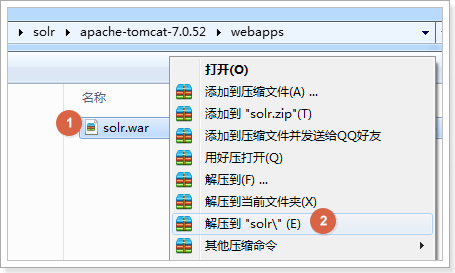
复制solr.war



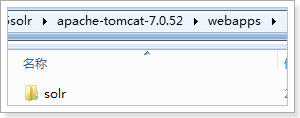
粘贴到自己Tomcat的webapps里

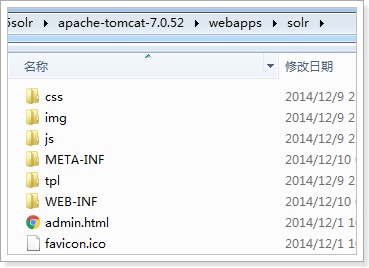


在Tomcat的webapps里，把war解压到当前路径，并删除solr.war



效果：

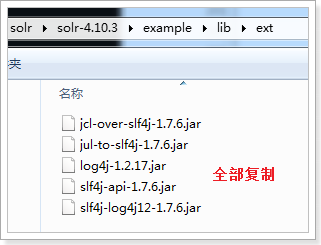




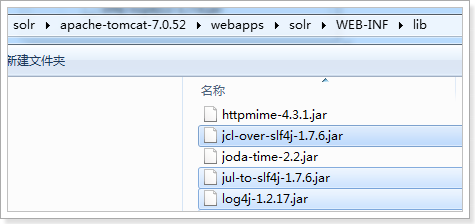
### 添加solr服务的扩展jar包（日志包）

把solr解压包下solr-4.10.3\example\lib\ext目录下的所有jar包拷贝到Tomcat部署的solr的WEB-INF/lib文件夹

复制扩展jar包

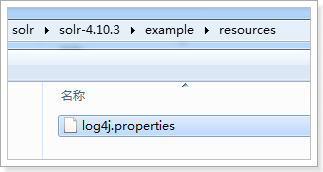


粘贴到Tomcat的webapps的solr工程的WEB-INF\lib目录



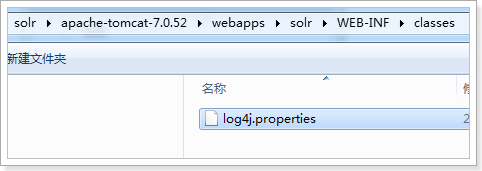
### 添加log4j.properties

把solr解压包下solr-4.10.3\example\resources\log4j.properties文件进行复制



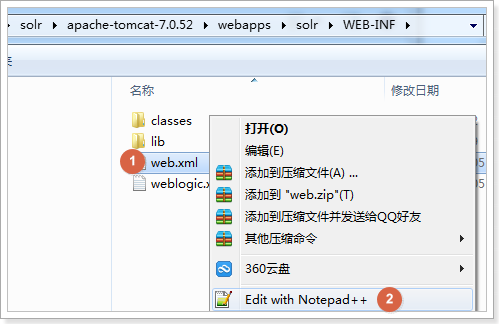
粘贴到Tomcat的webapps的solr的WEB-INF\classes目录下

这里没有classes文件夹创建一个即可



### 配置solr应用的web.xml

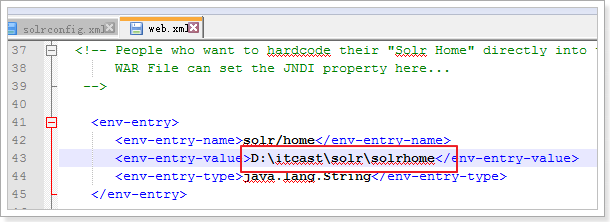
需要修改web.xml，让Tomcat使用JNDI的方式告诉solr服务器SolrHome在哪。



修改内容：

第42行的Solr/home名称必须是固定的，修

改第43行，如下图

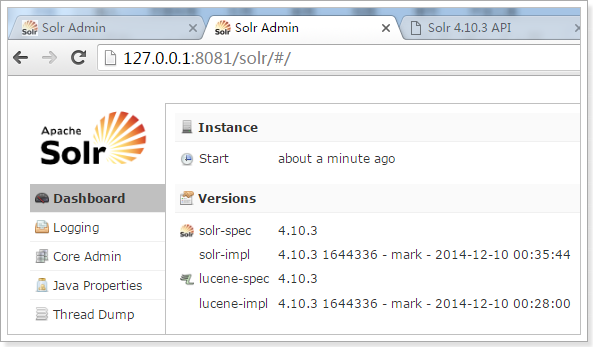


### 启动Tomcat进行访问

访问

http://localhost:8081/solr/

出现以下界面则说明solr安装成功！！！



## 管理界面功能介绍

### Dashboard

仪表盘，显示了该Solr实例开始启动运行的时间、版本、系统资源、jvm等信息。

### Logging

Solr运行日志信息

### Cloud

Cloud即SolrCloud，即Solr云（集群），当使用Solr Cloud模式运行时会显示此菜单，该部分功能在第二个项目，即电商项目会演示。

### Core Admin

Solr Core的管理界面。在这里可以添加SolrCore实例（有bug，不推荐使用浏览器界面添加SolrCore）。

### java properties

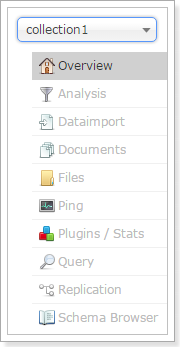
Solr在JVM 运行环境中的属性信息，包括类路径、文件编码、jvm内存设置等信息。

### Tread Dump

显示Solr Server中当前活跃线程信息，同时也可以跟踪线程运行栈信息。

### Core selector

选择一个SolrCore进行详细操作，如下：



#### Analysis

通过此界面可以测试索引分析器和搜索分析器的执行情况



#### dataimport

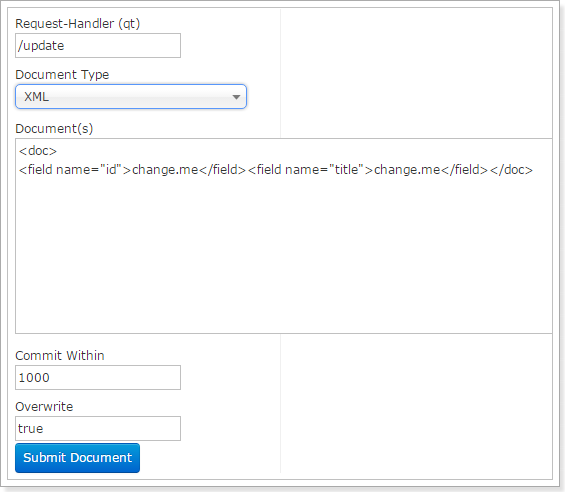
可以定义数据导入处理器，从关系数据库将数据导入到Solr索引库中。

默认没有配置，需要手工配置。

#### Document

通过/update表示更新索引，solr默认根据id（唯一约束）域来更新Document的内容，如果根据id值搜索不到id域则会执行添加操作，如果找到则更新。

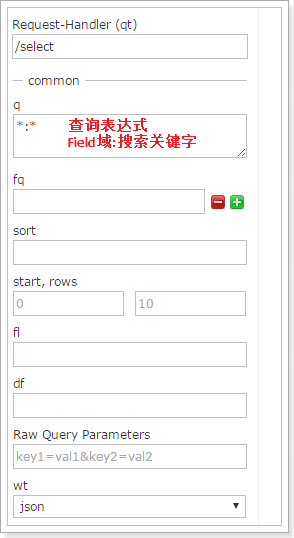
通过此菜单可以创建索引、更新索引、删除索引等操作，界面如下：



* overwrite="true" ： solr在做索引的时候，如果文档已经存在，就用xml中的文档进行替换
* commitWithin="1000" ： solr 在做索引的时候，每隔1000（1秒）毫秒，做一次文档提交。为了方便测试也可以在Document中立即提交，</doc>后添加“<commit/>”

#### Query

通过/select执行搜索索引，必须指定“q”查询条件方可搜索。



# Solrj的使用

## 什么是solrj

solrj是访问Solr服务的java客户端，提供索引和搜索的请求方法，如下图：

Index索引库

javaEE应用程序

SolrJ程序客户端

Solr服务

Tomcat

通过SolrJ请求Solr服务

最终Solr完在索引和搜索

Solrj和图形界面操作的区别就类似于数据库中使用jdbc和mysql客户端的区别一样。

## 需求

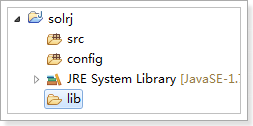
使用solrj调用solr服务实现对索引库的增删改查操作。

## 环境准备

* Solr：4.10.3
* Jdk环境：1.7
* IDE环境：Eclipse Mars2

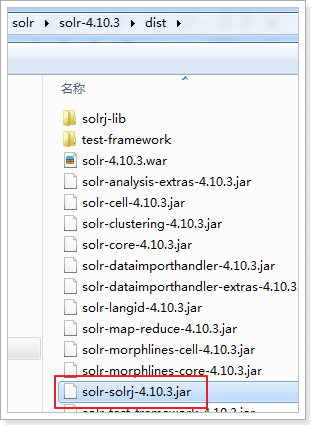
## 工程搭建

### 创建java工程



### 添加jar

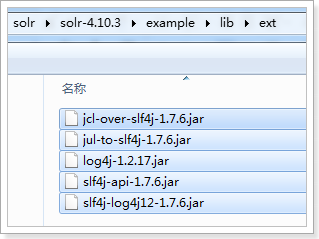
Solrj的包，\solr-4.10.3\dist\目录下



solrj依赖包，\solr-4.10.3\dist\solrj-lib



Solr服务的依赖包，\solr\example\lib\ext



## 代码实现

### 添加&修改索引

#### 步骤

1. 创建HttpSolrServer对象，通过它和Solr服务器建立连接。
2. 创建SolrInputDocument对象，然后通过它来添加域。
3. 通过HttpSolrServer对象将SolrInputDocument添加到索引库。
4. 提交。

#### 代码

说明：根据id（唯一约束）域来更新Document的内容，如果根据id值搜索不到id域则会执行添加操作，如果找到则更新。

@Test

**public** **void** testCreateAndUpdateIndex() **throws** Exception {

// 1. 创建HttpSolrServer对象

// 设置solr服务接口,浏览器客户端地址http://127.0.0.1:8081/solr/#/

String baseURL = "http://127.0.0.1:8081/solr";

HttpSolrServer httpSolrServer = **new** HttpSolrServer(baseURL);

// 2. 创建SolrInputDocument对象

SolrInputDocument document = **new** SolrInputDocument();

document.addField("id", "c1001");

document.addField("content ", "Hello world!");

// 3. 把SolrInputDocument对象添加到索引库中

httpSolrServer.add(document);

// 4. 提交

httpSolrServer.commit();

}

### 删除索引

#### 代码

抽取HttpSolrServer 的创建代码

**private** HttpSolrServer httpSolrServer;

// 提取HttpSolrServer创建

@Before

**public** **void** init() {

// 1. 创建HttpSolrServer对象

// 设置solr服务接口,浏览器客户端地址http://127.0.0.1:8081/solr/#/

String baseURL = "http://127.0.0.1:8081/solr/";

**this**.httpSolrServer = **new** HttpSolrServer(baseURL);

}

删除索引逻辑，两种：

根据id删除

根据条件删除，根据条件删除

可以使用\*:\*作为条件，就是删除所有数据（慎用）

@Test

**public** **void** testDeleteIndex() **throws** Exception {

// 根据id删除索引数据

// this.httpSolrServer.deleteById("c1001");

// 根据条件删除（如果是\*:\*就表示全部删除，慎用）

**this**.httpSolrServer.deleteByQuery("\*:\*");

// 提交

**this**.httpSolrServer.commit();

}

### 查询索引

#### 简单查询

/\*\*

\* 简单搜索

\*

\* @throws Exception

\*/

@Test

public void testSearchIndex1() throws Exception {

// 创建搜索对象

SolrQuery query = new SolrQuery();

// 设置搜索条件

query.setQuery("\*:\*");

// 发起搜索请求

QueryResponse response = this.httpSolrServer.query(query);

// 处理搜索结果

SolrDocumentList results = response.getResults();

System.*out*.println("搜索到的结果总数：" + results.getNumFound());

// 遍历搜索结果

for (SolrDocument solrDocument : results) {

System.*out*.println("----------------------------------------------------");

System.*out*.println("id：" + solrDocument.get("id"));

System.*out*.println("content" + solrDocument.get("content"));

}

}