**全文检索技术**

**Lucene&Solr**

**Part2**

# 课程计划

一、索引库的维护

1、 添加文档

2、 删除文档

3、 修改文档

二、Lucene的查询

1、 使用Query的子类查询

1）、MatchAllDocsQuery

2）、TermQuery

3）、NumericRangeQuery

4）、BooleanQuery

2、 使用QueryParser

1）、QueryParser

2）、MulitFieldQueryParser

三、相关度排序

四、什么是solr

五、Solr的安装及配置

六、Solr后台的使用

七、使用solrj维护索引库

# 索引库的维护

## 索引库的添加

### 步骤

向索引库中添加document对象。

第一步：先创建一个indexwriter对象

第二步：创建一个document对象

第三步：把document对象写入索引库

第四步：关闭indexwriter。

### 代码实现

|  |
| --- |
| //添加索引  @Test  **public** **void** addDocument() **throws** Exception {  //索引库存放路径  Directory directory = FSDirectory.*open*(**new** File("D:\\temp\\0108\\index"));    IndexWriterConfig config = **new** IndexWriterConfig(Version.*LATEST*, **new** IKAnalyzer());  //创建一个indexwriter对象  IndexWriter indexWriter = **new** IndexWriter(directory, config);  //创建一个Document对象  Document document = **new** Document();  //向document对象中添加域。  //不同的document可以有不同的域，同一个document可以有相同的域。  document.add(**new** TextField("filename", "新添加的文档", Store.*YES*));  document.add(**new** TextField("content", "新添加的文档的内容", Store.*NO*));  document.add(**new** TextField("content", "新添加的文档的内容第二个content", Store.*YES*));  document.add(**new** TextField("content1", "新添加的文档的内容要能看到", Store.*YES*));  //添加文档到索引库  indexWriter.addDocument(document);  //关闭indexwriter  indexWriter.close();    } |

## Field域的属性

**是否分析**：是否对域的内容进行分词处理。前提是我们要对域的内容进行查询。

**是否索引**：将Field分析后的词或整个Field值进行索引，只有索引方可搜索到。

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

**是否存储**：将Field值存储在文档中，存储在文档中的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。

**是否存储的标准：是否要将内容展示给用户**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Field类 | 数据类型 | Analyzed  是否分析 | 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，但是不会进行分析，会将整个串存储在索引中，比如(订单号,姓名等)  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，进行分析和索引，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. |

## 索引库删除

### 删除全部

|  |
| --- |
| //删除全部索引  @Test  **public** **void** deleteAllIndex() **throws** Exception {  IndexWriter indexWriter = getIndexWriter();  //删除全部索引  indexWriter.deleteAll();  //关闭indexwriter  indexWriter.close();  } |

说明：将索引目录的索引信息全部删除，直接彻底删除，无法恢复。

**此方法慎用！！**

### 指定查询条件删除

|  |
| --- |
| //根据查询条件删除索引  @Test  **public** **void** deleteIndexByQuery() **throws** Exception {  IndexWriter indexWriter = getIndexWriter();  //创建一个查询条件  Query query = **new** TermQuery(**new** Term("filename", "apache"));  //根据查询条件删除  indexWriter.deleteDocuments(query);  //关闭indexwriter  indexWriter.close();  } |

## 索引库的修改

原理就是先删除后添加。

|  |
| --- |
| //修改索引库  @Test  **public** **void** updateIndex() **throws** Exception {  IndexWriter indexWriter = getIndexWriter();  //创建一个Document对象  Document document = **new** Document();  //向document对象中添加域。  //不同的document可以有不同的域，同一个document可以有相同的域。  document.add(**new** TextField("filename", "要更新的文档", Store.*YES*));  document.add(**new** TextField("content", "2013年11月18日 - Lucene 简介 Lucene 是一个基于 Java 的全文信息检索工具包,它不是一个完整的搜索应用程序,而是为你的应用程序提供索引和搜索功能。", Store.*YES*));  indexWriter.updateDocument(**new** Term("content", "java"), document);  //关闭indexWriter  indexWriter.close();  } |

# Lucene索引库查询（重点）

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Field的name为“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

QueryParse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");

## 使用query的子类查询

### MatchAllDocsQuery

使用MatchAllDocsQuery查询索引目录中的所有文档

|  |
| --- |
| @Test  **public** **void** testMatchAllDocsQuery() **throws** Exception {  IndexSearcher indexSearcher = getIndexSearcher();  //创建查询条件  Query query = **new** MatchAllDocsQuery();  //执行查询  printResult(query, indexSearcher);  } |

### TermQuery

TermQuery，通过项查询，TermQuery不使用分析器所以建议匹配不分词的Field域查询，比如订单号、分类ID号等。

指定要查询的域和要查询的关键词。

|  |
| --- |
| //使用Termquery查询  @Test  **public** **void** testTermQuery() **throws** Exception {  IndexSearcher indexSearcher = getIndexSearcher();  //创建查询对象  Query query = **new** TermQuery(**new** Term("content", "lucene"));  //执行查询  TopDocs topDocs = indexSearcher.search(query, 10);  //共查询到的document个数  System.*out*.println("查询结果总数量：" + topDocs.totalHits);  //遍历查询结果  **for** (ScoreDoc scoreDoc : topDocs.scoreDocs) {  Document document = indexSearcher.doc(scoreDoc.doc);  System.*out*.println(document.get("filename"));  //System.out.println(document.get("content"));  System.*out*.println(document.get("path"));  System.*out*.println(document.get("size"));  }  //关闭indexreader  indexSearcher.getIndexReader().close();  } |

### NumericRangeQuery

可以根据数值范围查询。

|  |
| --- |
| //数值范围查询  @Test  **public** **void** testNumericRangeQuery() **throws** Exception {  IndexSearcher indexSearcher = getIndexSearcher();  //创建查询  //参数：  //1.域名  //2.最小值  //3.最大值  //4.是否包含最小值  //5.是否包含最大值  Query query = NumericRangeQuery.*newLongRange*("size", 1l, 1000l, **true**, **true**);  //执行查询  printResult(query, indexSearcher);  } |

### BooleanQuery

可以组合查询条件。

|  |
| --- |
| //组合条件查询  @Test  **public** **void** testBooleanQuery() **throws** Exception {  IndexSearcher indexSearcher = getIndexSearcher();  //创建一个布尔查询对象  BooleanQuery query = **new** BooleanQuery();  //创建第一个查询条件  Query query1 = **new** TermQuery(**new** Term("filename", "apache"));  Query query2 = **new** TermQuery(**new** Term("content", "apache"));  //组合查询条件  query.add(query1, Occur.*MUST*);  query.add(query2, Occur.*MUST*);  //执行查询  printResult(query, indexSearcher);  } |

Occur.MUST：必须满足此条件，相当于and

Occur.SHOULD：应该满足，但是不满足也可以，相当于or

Occur.MUST\_NOT：必须不满足。相当于not

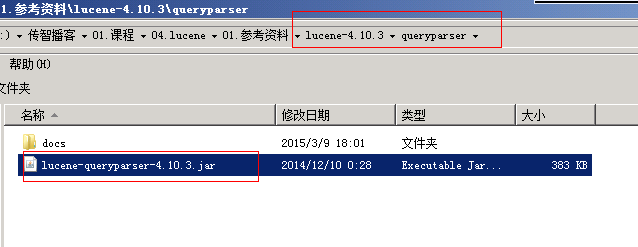
## 使用queryparser查询

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。Query对象执行的查询语法可通过System.out.println(query);查询。

需要使用到分析器。建议创建索引时使用的分析器和查询索引时使用的分析器要一致。

### QueryParser

需要加入queryParser依赖的jar包。



#### 程序实现

|  |
| --- |
| @Test  **public** **void** testQueryParser() **throws** Exception {  IndexSearcher indexSearcher = getIndexSearcher();  //创建queryparser对象  //第一个参数默认搜索的域  //第二个参数就是分析器对象  QueryParser queryParser = **new** QueryParser("content", **new** IKAnalyzer());  Query query = queryParser.parse("Lucene是java开发的");  //执行查询  printResult(query, indexSearcher);  } |

#### 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：content:java

1. 范围查询

域名+“:”+[最小值 TO 最大值]

例如：size:[1 TO 1000]

范围查询在lucene中不支持数值类型，支持字符串类型。在solr中支持数值类型。

1. 组合条件查询

1）+条件1 +条件2：两个条件之间是并且的关系and

例如：+filename:apache +content:apache

1. +条件1 条件2：必须满足第一个条件，应该满足第二个条件

例如：+filename:apache content:apache

1. 条件1 条件2：两个条件满足其一即可。

例如：filename:apache content:apache

4）-条件1 条件2：必须不满足条件1，要满足条件2

例如：-filename:apache content:apache

|  |  |
| --- | --- |
| 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） |

第二种写法：

条件1 AND 条件2

条件1 OR 条件2

条件1 NOT 条件2

### MulitFieldQueryParser

可以指定多个默认搜索域

|  |
| --- |
| @Test  **public** **void** testMultiFiledQueryParser() **throws** Exception {  IndexSearcher indexSearcher = getIndexSearcher();  //可以指定默认搜索的域是多个  String[] fields = {"filename", "content"};  //创建一个MulitFiledQueryParser对象  MultiFieldQueryParser queryParser = **new** MultiFieldQueryParser(fields, **new** IKAnalyzer());  Query query = queryParser.parse("java and apache");  System.*out*.println(query);  //执行查询  printResult(query, indexSearcher);    } |

# 相关度排序

## 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，采用空间向量模型算法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)，搜索也是要从Term中搜索，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)

即有多少文档包含次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影响打分结果

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的Field域设置加权值高，在搜索时匹配到这个Field就可能排在前边。lucene在执行搜索时对某个域进行加权，在进行组合域查询时，匹配到加权值高的域最后计算的相关度得分就高。

如果希望某些文档更重要，当此文档中包含所要查询的词则应该得分较高，这样相关度排序可以排在前边，可以在创建索引时设定文档中某些域（Field）的boost值来实现，如果不进行设定，则Field Boost默认为1.0f。一旦设定，除非删除此文档，否则无法改变。

代码：

field. setBoost(XXXf); XXX即权值。

测试：

可以将springmvc.txt的file\_content加权值设置为10.0f，结果搜索spring时如果内容可以匹配到关键字就可以把springmvc.txt文件排在前边。

代码：

索引时设置boost加权值：

//设置加权值

**if**(file\_name.equals("springmvc.txt")){

//设置比默认值 1.0大的

field\_file\_content.setBoost(20.0f);

}

**if**(file\_name.equals("spring\_README.txt")){

//设置比默认值 1.0大的

field\_file\_content.setBoost(30.0f);

}

//向文档中添加Field

document.add(field\_file\_content);

搜索时：

// 设置组合查询域，如果匹配到一个域就返回记录

String[] fields = { "file\_content" };

//设置评分,文件名称中包括关键字的评分高

/\*Map<String,Float> boosts = new HashMap<String,Float>();

boosts.put("file\_content", 3.0f);\*/

// 创建查询解析器

QueryParser queryParser = **new** MultiFieldQueryParser(fields,

**new** StandardAnalyzer());

// 查询文件名、文件内容中包括“java”关键字的文档

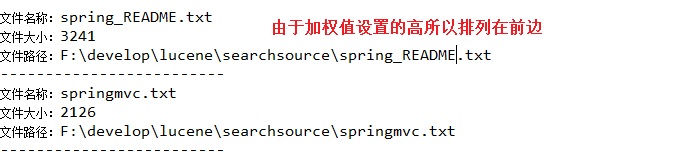
Query query = queryParser.parse("spring");

TopDocs topDocs = indexSearcher.search(query, 100);

ScoreDoc[] scoreDocs = topDocs.scoreDocs;

结果：

springmvc.txt排在最前边



# 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的全文搜索服务器。Solr提供了比Lucene更为丰富的查询语言，同时实现了可配置、可扩展，并对索引、搜索性能进行了优化。

Solr可以独立运行，运行在Jetty、Tomcat等这些Servlet容器中，Solr 索引的实现方法很简单，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根据xml文档添加、删除、更新索引 。Solr 搜索只需要发送 HTTP GET 请求，然后对 Solr 返回Xml、json等格式的查询结果进行解析，组织页面布局。Solr不提供构建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与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引擎，Lucene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引擎。

Solr的目标是打造一款企业级的搜索引擎系统，它是一个搜索引擎服务，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安装及配置

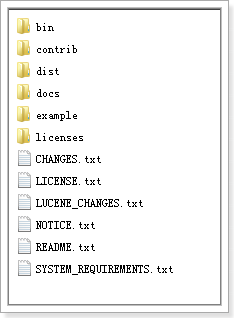
## Solr的下载

从Solr官方网站（http://lucene.apache.org/solr/ ）下载Solr4.10.3，根据Solr的运行环境，Linux下需要下载lucene-4.10.3.tgz，windows下需要下载lucene-4.10.3.zip。

Solr使用指南可参考：https://wiki.apache.org/solr/FrontPage。

## Solr的文件夹结构

将solr-4.10.3.zip解压：



bin：solr的运行脚本

contrib：solr的一些贡献软件/插件，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：

该目录是一个包含了默认配置信息的Solr的Core目录。

* 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），本教程使用Tocmat作为Servlet容器，环境如下：

Solr：Solr4.10.3

Jdk：jdk1.7.0\_72

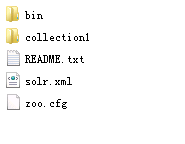
Tomcat：apache-tomcat-7.0.53

## Solr整合tomcat

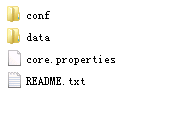
### Solr Home与SolrCore

创建一个Solr home目录，SolrHome是Solr运行的主目录，目录中包括了运行Solr实例所有的配置文件和数据文件，Solr实例就是SolrCore，一个SolrHome可以包括多个SolrCore（Solr实例），每个SolrCore提供单独的搜索和索引服务。

example\solr是一个solr home目录结构，如下：



上图中“collection1”是一个SolrCore（Solr实例）目录 ，目录内容如下所示：



说明：

collection1：叫做一个Solr运行实例SolrCore，SolrCore名称不固定，一个solr运行实例对外单独提供索引和搜索接口。

solrHome中可以创建多个solr运行实例SolrCore。

一个solr的运行实例对应一个索引目录。

conf是SolrCore的配置文件目录 。

data目录存放索引文件需要创建

### 整合步骤

第一步：安装tomcat。D:\temp\apache-tomcat-7.0.53

第二步：把solr的war包复制到tomcat 的webapp目录下。

把\solr-4.10.3\dist\solr-4.10.3.war复制到D:\temp\apache-tomcat-7.0.53\webapps下。

改名为solr.war  
第三步：solr.war解压。使用压缩工具解压或者启动tomcat自动解压。解压之后删除solr.war

第四步：把\solr-4.10.3\example\lib\ext目录下的所有的jar包添加到solr工程中

第五步：配置solrHome和solrCore。

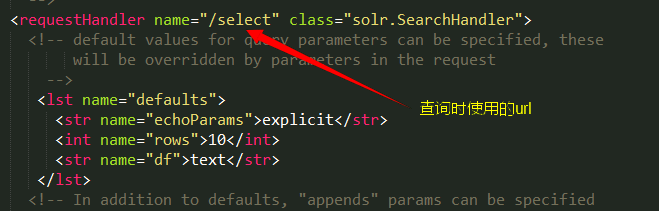
1. 创建一个solrhome（存放solr所有配置文件的一个文件夹）。\solr-4.10.3\example\solr目录就是一个标准的solrhome。
2. 把\solr-4.10.3\example\solr文件夹复制到D:\temp\0108路径下，改名为solrhome，改名不是必须的，是为了便于理解。
3. 在solrhome下有一个文件夹叫做collection1这就是一个solrcore。就是一个solr的实例。一个solrcore相当于mysql中一个数据库。Solrcore之间是相互隔离。
   * 1. 在solrcore中有一个文件夹叫做conf，包含了索引solr实例的配置信息。
     2. 在conf文件夹下有一个solrconfig.xml。配置实例的相关信息。**如果使用默认配置可以不用做任何修改。**

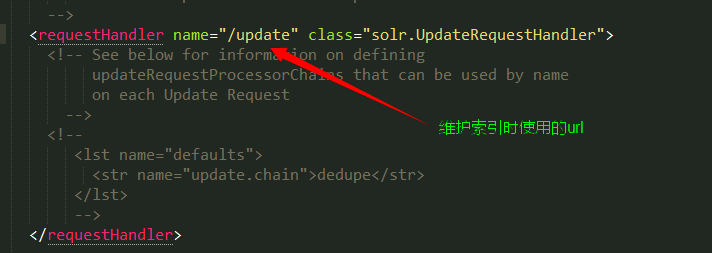
**Xml的配置信息：**

**Lib**：solr服务依赖的扩展包，默认的路径是collection1\lib文件夹，如果没有 就创建一个

**dataDir**：配置了索引库的存放路径。默认路径是collection1\data文件夹，如 果data文件夹，会自动创建。

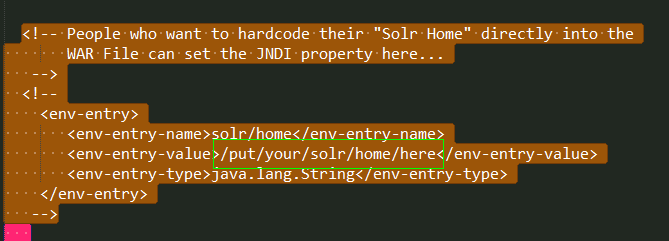
**requestHandler：**

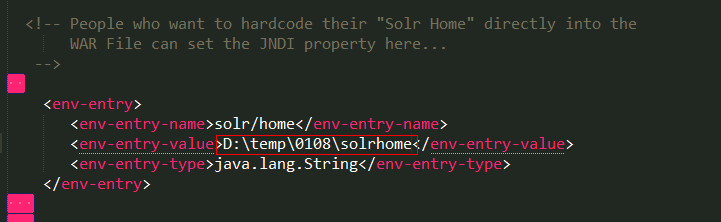




第六步：告诉solr服务器配置文件也就是solrHome的位置。修改web.xml使用jndi的方式告诉solr服务器。

**Solr/home名称必须是固定的。**



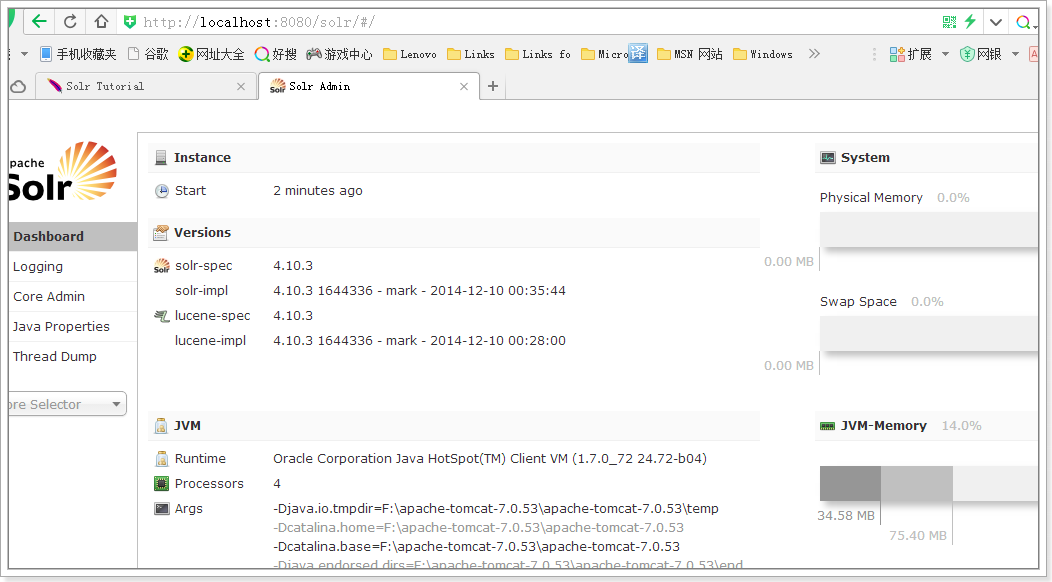


第七步：启动tomcat

第八步：访问http://localhost:8080/solr/

## Solr后台管理

### 管理界面



### 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

### Core Admin

Solr Core的管理界面。Solr Core 是Solr的一个独立运行实例单位，它可以对外提供索引和搜索服务，一个Solr工程可以运行多个SolrCore（Solr实例），一个Core对应一个索引目录。

添加solrcore：

第一步：复制collection1改名为collection2

第二步：修改core.properties。name=collection2

第三步：重启tomcat

### java properties

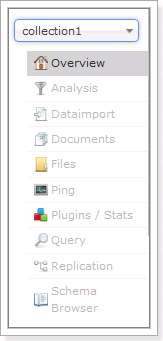
Solr在JVM 运行环境中的属性信息，包括类路径、文件编码、jvm内存设置等信息。

### 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



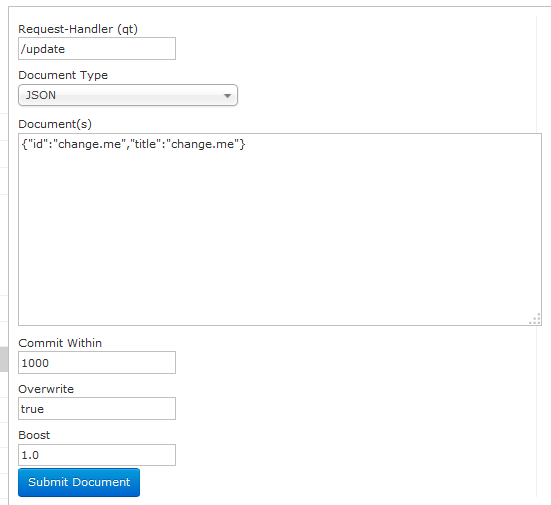
通过此界面可以测试索引分析器和搜索分析器的执行情况。

### 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。

### Document

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：



/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。

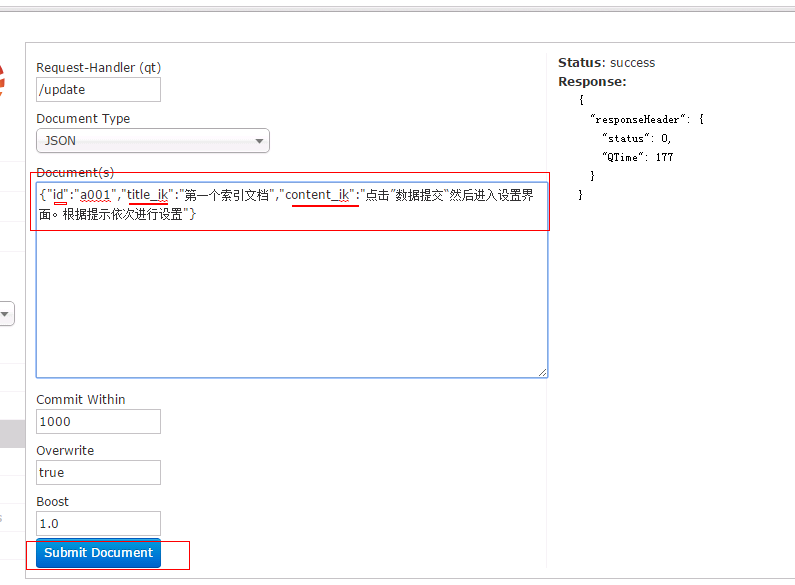
### Query



通过/select执行搜索索引，必须指定“q”查询条件方可搜索。

# Solr管理索引库

## 添加/更新文档



## 删除文档

删除索引格式如下：

1） 删除制定ID的索引

<delete>

<id>8</id>

</delete>

<commit/>

2） 删除查询到的索引数据

<delete>

<query>product\_catalog\_name:幽默杂货</query>

</delete>

<commit/>

3） 删除所有索引数据

<delete>

<query>\*:\*</query>

</delete>

<commit/>

## 查询索引

# 使用SolrJ管理索引库

## 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，SolrJ通常在嵌入在业务系统中，通过SolrJ的API接口操作Solr服务，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

## 依赖的jar包

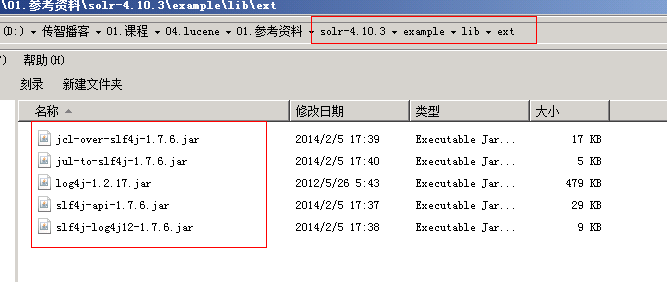


## 添加文档

### 实现步骤

第一步：创建一个java工程

第二步：导入jar包。包括solrJ的jar包。还需要



第三步：和Solr服务器建立连接。HttpSolrServer对象建立连接。

第四步：创建一个SolrInputDocument对象，然后添加域。

第五步：将SolrInputDocument添加到索引库。

第六步：提交。

### 代码实现

|  |
| --- |
| //向索引库中添加索引  @Test  **public** **void** addDocument() **throws** Exception {  //和solr服务器创建连接  //参数：solr服务器的地址  SolrServer solrServer = **new** HttpSolrServer("http://localhost:8080/solr/collection1");  //创建一个文档对象  SolrInputDocument document = **new** SolrInputDocument();  //向文档中添加域  //第一个参数：域的名称，域的名称必须是在schema.xml中定义的  //第二个参数：域的值  document.addField("id", "c0001");  document.addField("title\_ik", "使用solrJ添加的文档");  document.addField("content\_ik", "文档的内容");  document.addField("product\_name", "商品名称");  //把document对象添加到索引库中  solrServer.add(document);  //提交修改  solrServer.commit();    } |

## 删除文档

### 根据id删除

|  |
| --- |
| //删除文档，根据id删除  @Test  **public** **void** deleteDocumentByid() **throws** Exception {  //创建连接  SolrServer solrServer = **new** HttpSolrServer("http://localhost:8080/solr");  //根据id删除文档  solrServer.deleteById("c0001");  //提交修改  solrServer.commit();  } |

### 根据查询删除

查询语法完全支持Lucene的查询语法。

|  |
| --- |
| //根据查询条件删除文档  @Test  **public** **void** deleteDocumentByQuery() **throws** Exception {  //创建连接  SolrServer solrServer = **new** HttpSolrServer("http://localhost:8080/solr");  //根据查询条件删除文档  solrServer.deleteByQuery("\*:\*");  //提交修改  solrServer.commit();  } |

## 修改文档

在solrJ中修改没有对应的update方法，只有add方法，只需要添加一条新的文档，和被修改的文档id一致就，可以修改了。本质上就是先删除后添加。

## 查询文档

|  |
| --- |
| //查询索引  @Test  **public** **void** queryIndex() **throws** Exception {  //创建连接  SolrServer solrServer = **new** HttpSolrServer("http://localhost:8080/solr");  //创建一个query对象  SolrQuery query = **new** SolrQuery();  //设置查询条件  query.setQuery("\*:\*");  //执行查询  QueryResponse queryResponse = solrServer.query(query);  //取查询结果  SolrDocumentList solrDocumentList = queryResponse.getResults();  //共查询到商品数量  System.*out*.println("共查询到商品数量:" + solrDocumentList.getNumFound());  //遍历查询的结果  **for** (SolrDocument solrDocument : solrDocumentList) {  System.*out*.println(solrDocument.get("id"));  System.*out*.println(solrDocument.get("product\_name"));  System.*out*.println(solrDocument.get("product\_price"));  System.*out*.println(solrDocument.get("product\_catalog\_name"));  System.*out*.println(solrDocument.get("product\_picture"));    }  } |