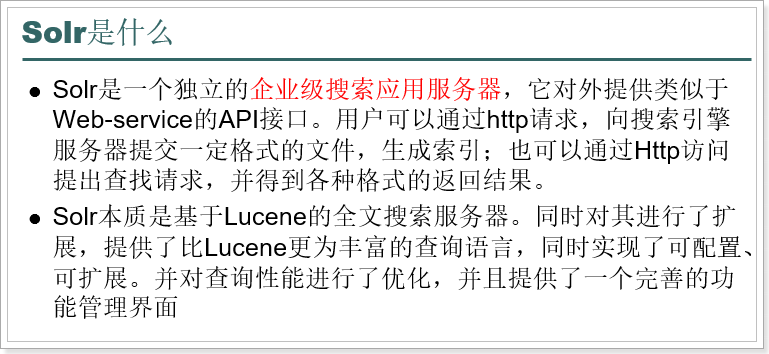
**Solr**

# 课堂笔记

# Solr概述

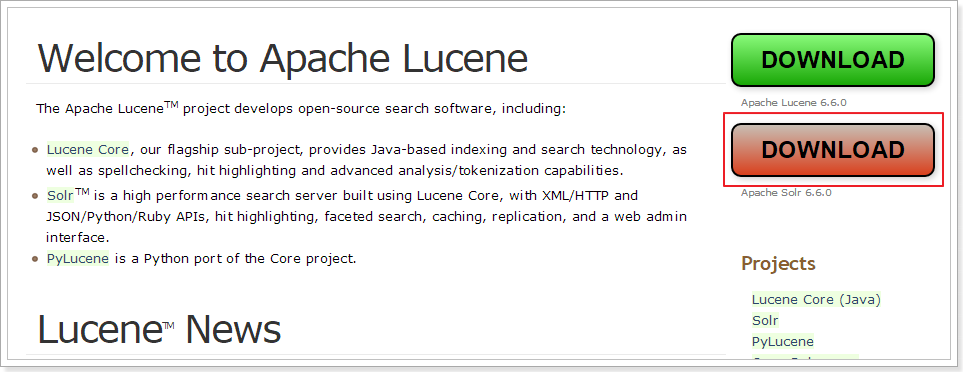
## 什么是Solr



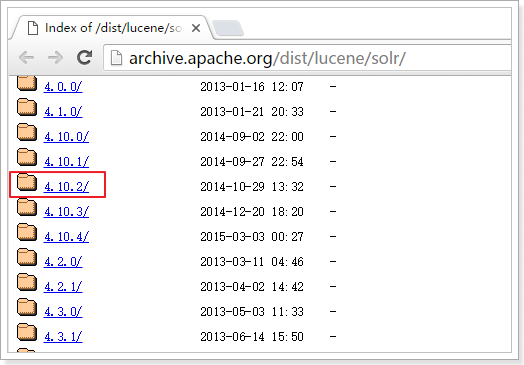
## Solr的下载及安装

1）下载：

官网：[http://lucene.apache.org](http://lucene.apache.org/solr/)



历史版本下载网址：<http://archive.apache.org/dist/lucene/solr/>



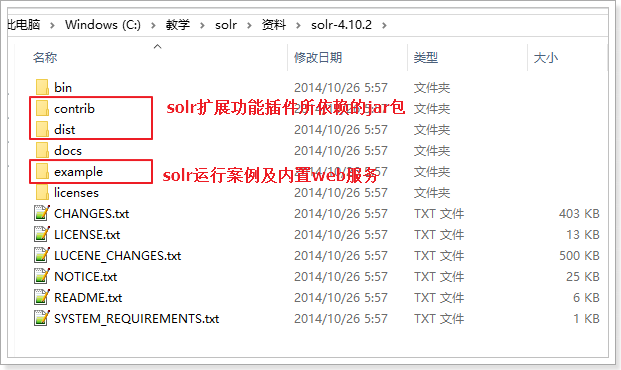
也可以直接使用发给大家的压缩包：



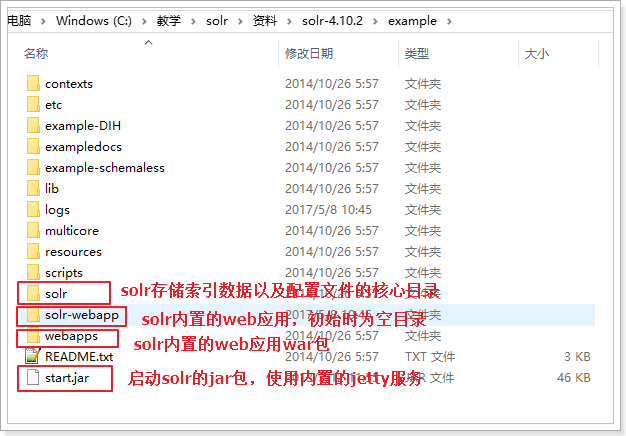
2）安装

右键解压到当前文件夹即可，注意：解压的目录中一定不要中文和空格。

## 目录结构



我们最需要关注的就是：example目录。其中就有搭建Solr服务的详细案例。



Jetty是一个WEB容器

* **Solr目录**：包含全文检索要存储的索引数据以及相关的一些配置文件。
* **solr-webapp和webapps**：其实包含的是一样的内容，都是Solr的内置Web应用，我们可以通过Http请求访问该应用，实现对Solr服务的管理和访问
* **start.jar**：web应用需要一个Web容器来启动，而这个jar包会使用一个叫做Jetty的WEB容器（类似与Tomcat），来启动Solr的Web服务。

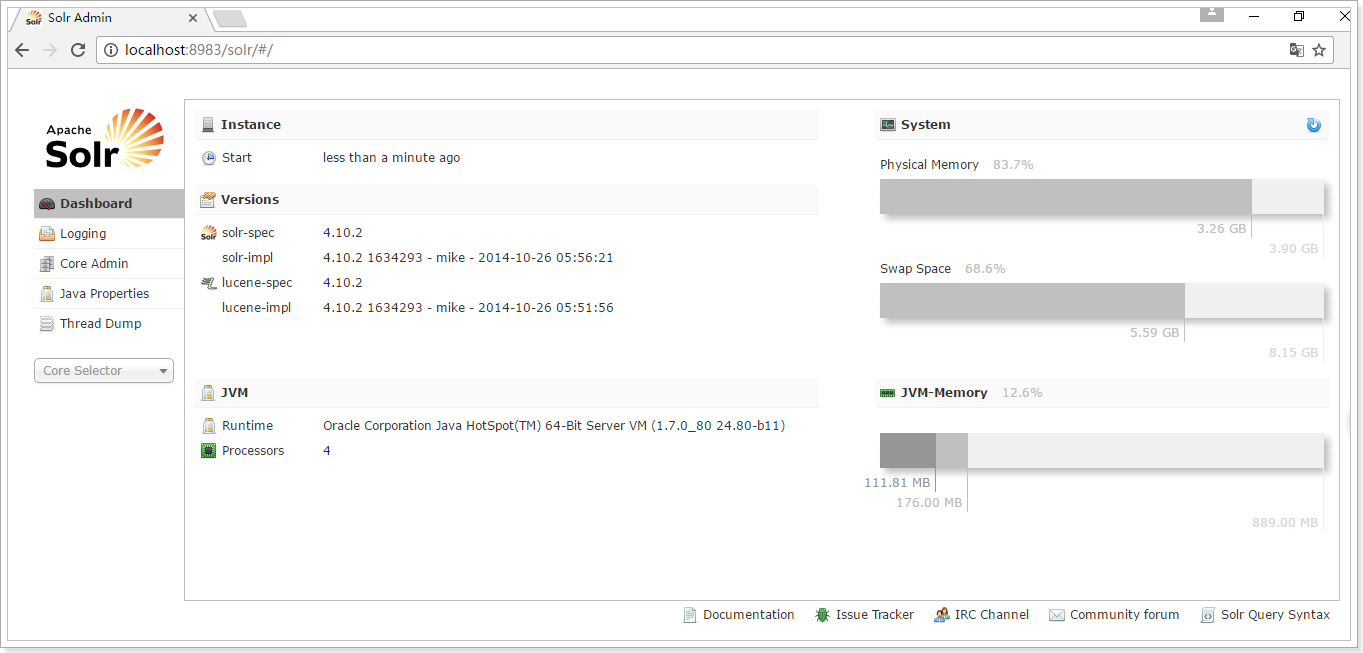
# 运行Solr服务

## 启动Solr服务的方式

### 方式一，Jetty服务器启动Solr（了解）

步骤：

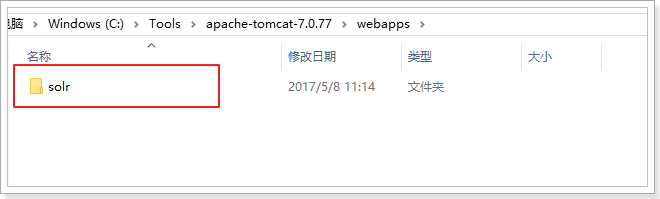
1. 进入solr-4.10.2/example目录
2. 打开命令行，执行java –jar start.jar命令，即可启动Solr服务
3. 打开浏览器，通过<http://localhost:8983/solr>来访问Solr管理页面。（Jetty服务的默认端口是8983）



### 方式二，Tomcat服务器启动

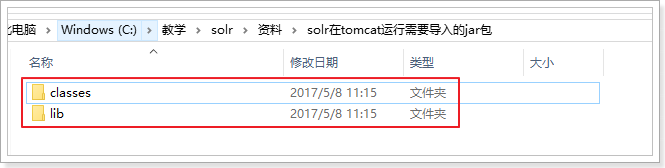
步骤：

1. 部署Web服务，将solr-4.10.2/example/webapps/solr.war复制到自己的tomcat/webapps目录中，并解压，然后删除solr.war文件



1. 在Tomcat中加入相关jar包：将“resource\solr在tomcat运行需要导入的jar包\lib”下的jar包复制tomcat/webapps/solr/WEB-INF/lib下。

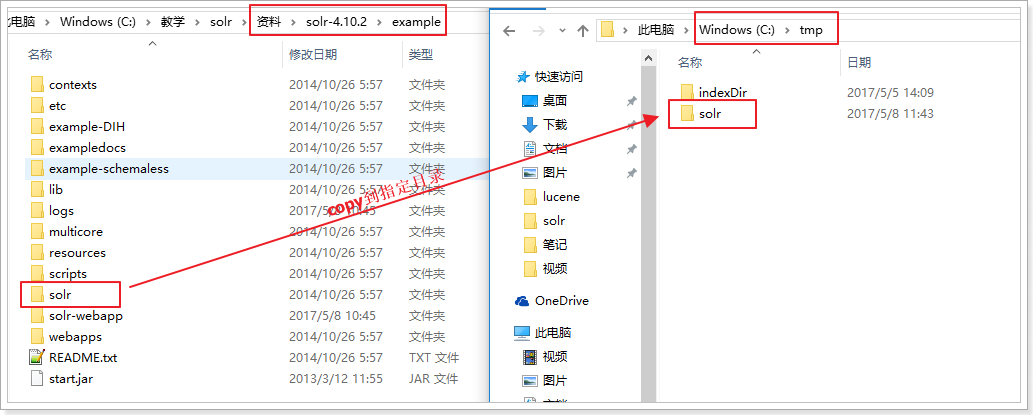
并且把class/log4j.properties复制到tomcat/webapps/solr/WEB-INF下



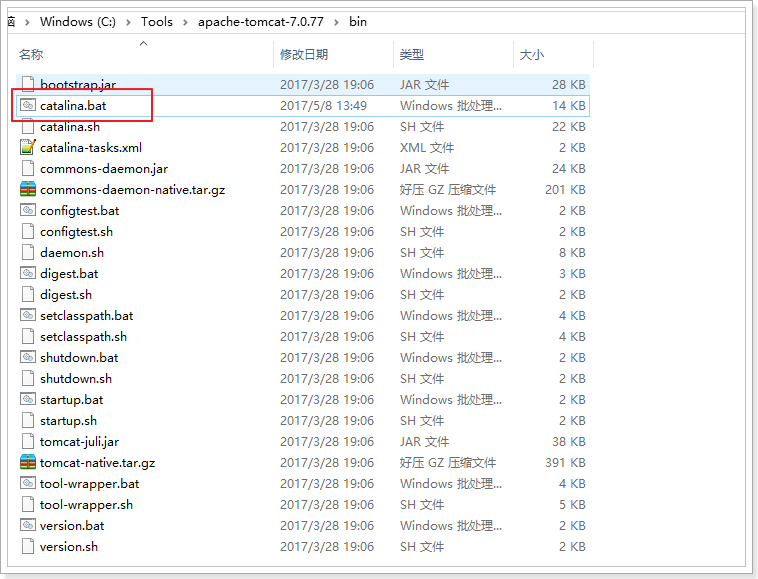
1. 修改Tomcat配置文件，指向Solr的索引库及配置目录。

注意，这里可以指向solr-4.10.2/example/solr目录，如果想独立出来，也可以将这个solr文件夹复制出来到任意位置（**不要出现中文**），例如：例如:C:/tmp/solr

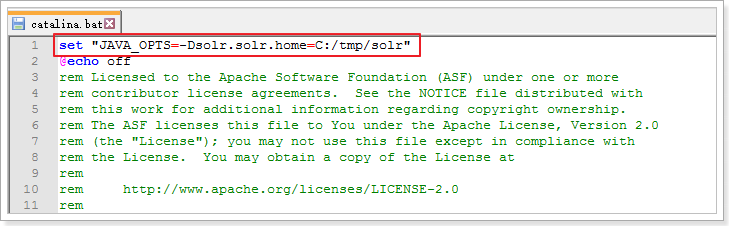
我们这里就把这个文件夹复制出来，独立使用



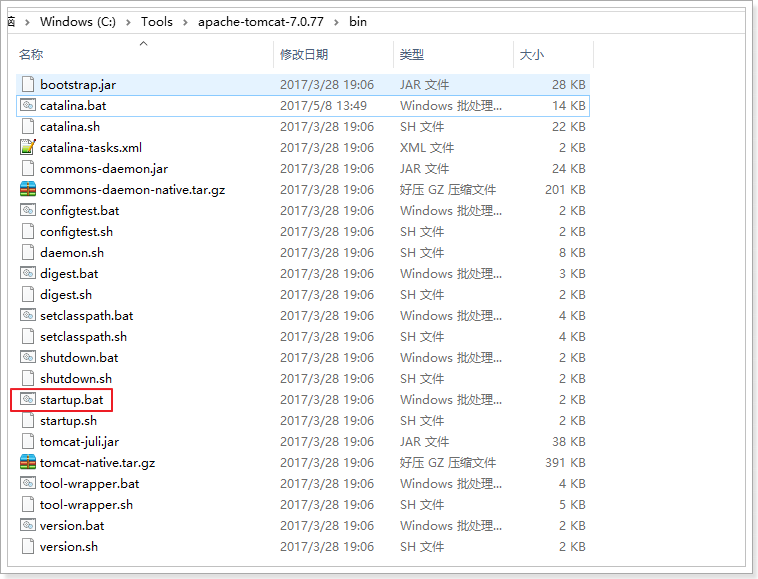
进入Tomcat文件夹，用记事本打开：tomcat/bin/catalina.bat文件

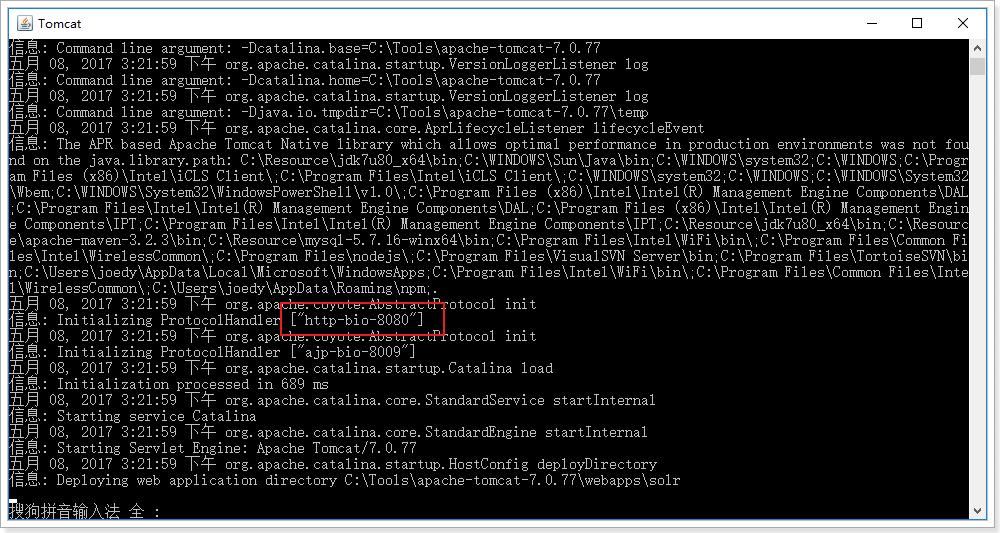


添加一条配置信息，指向我们的索引库及配置目录：set "JAVA\_OPTS=-Dsolr.solr.home=C:/tmp/solr"

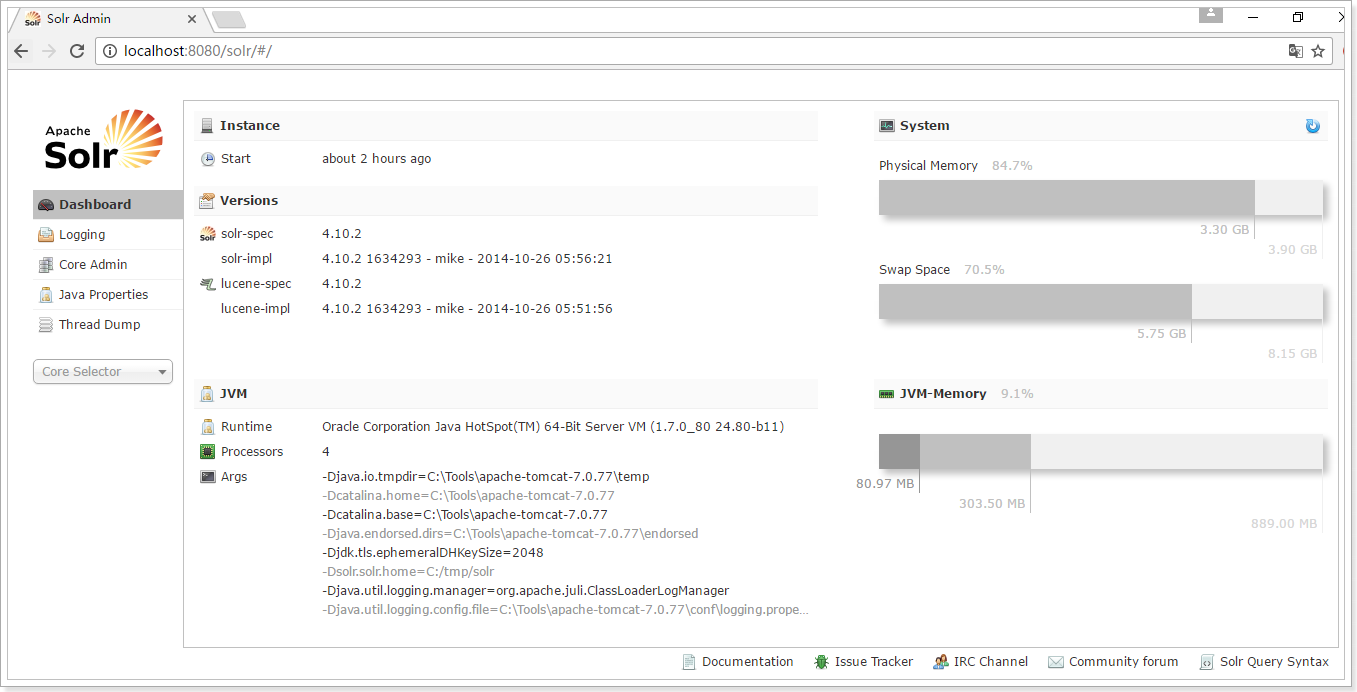


1. 进入tomcat/bin目录，双击 startup.bat文件启动服务器



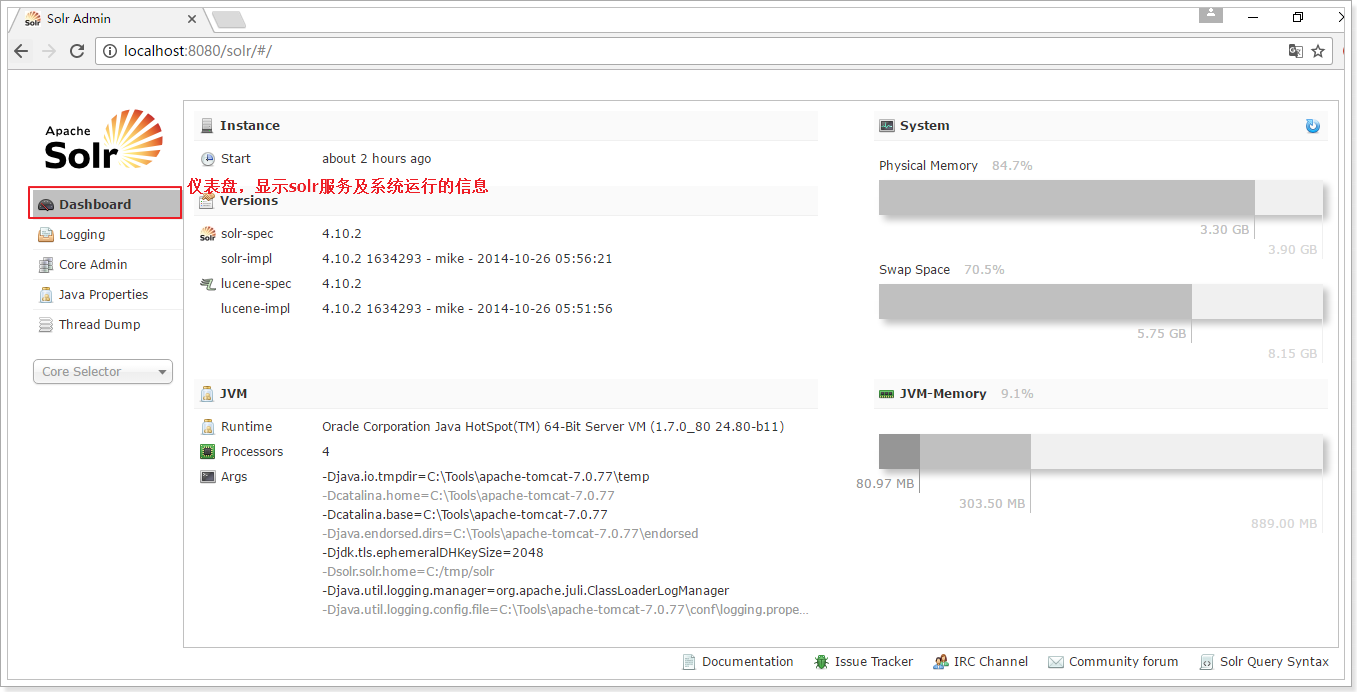


1. 打开浏览器，访问 <http://localhost:8080/solr> 进入Solr管理页面

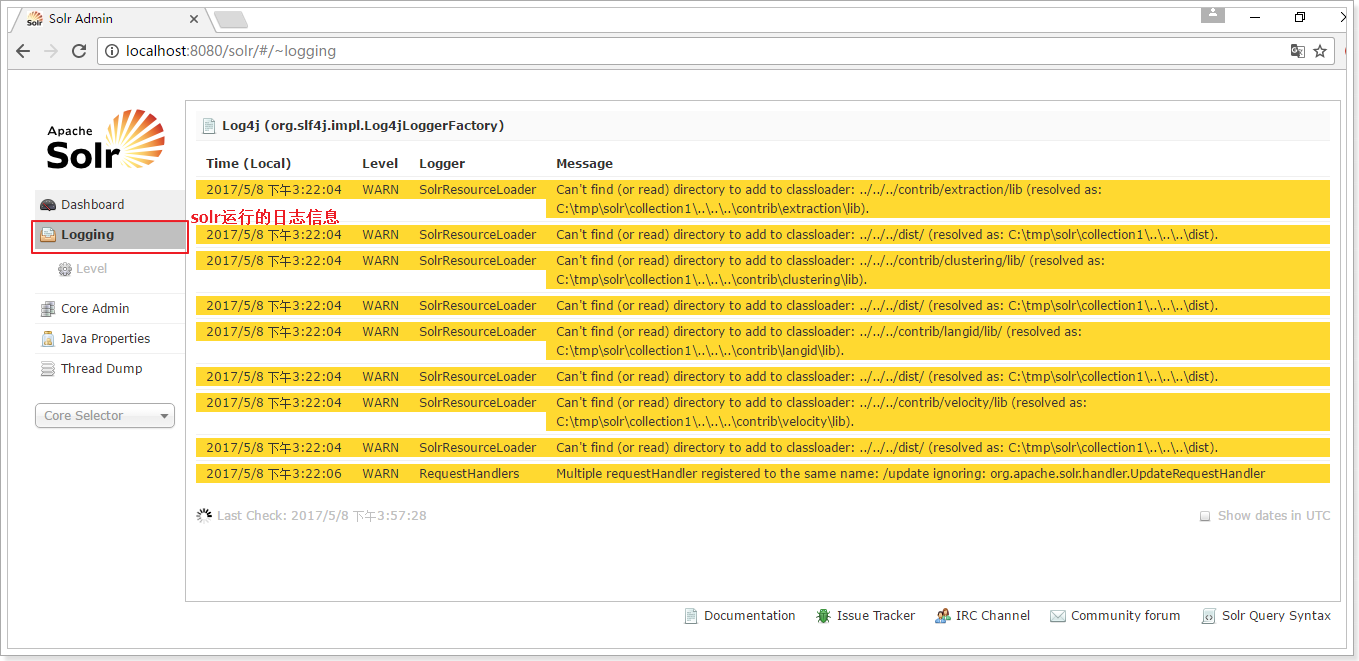


## Solr管理页面

### DashBoard（仪表盘）



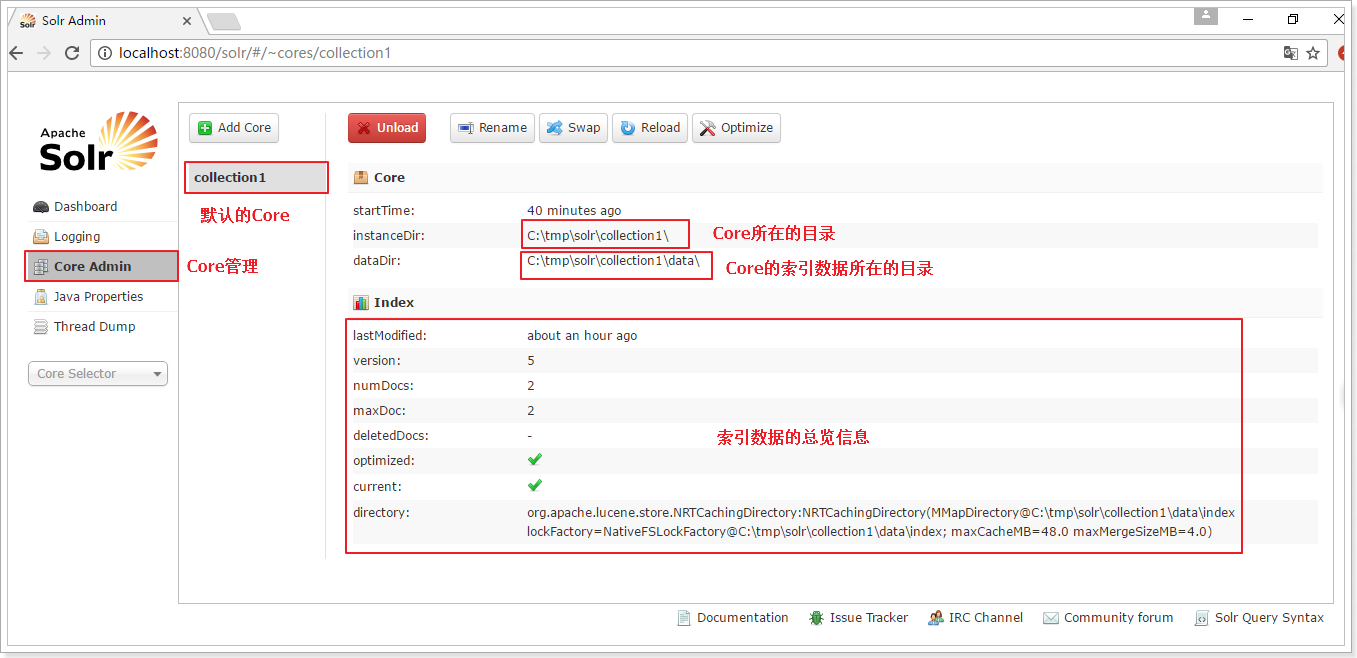
### Logging（日志）



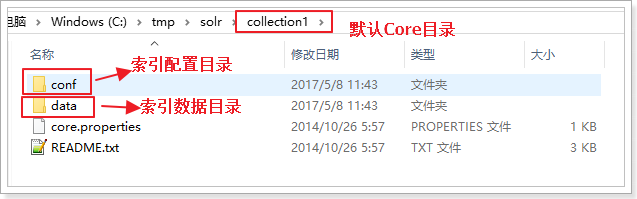
### Core Admin（Core管理）

在Solr中，每一个Core，代表一个索引库，里面包含索引数据及其配置信息。

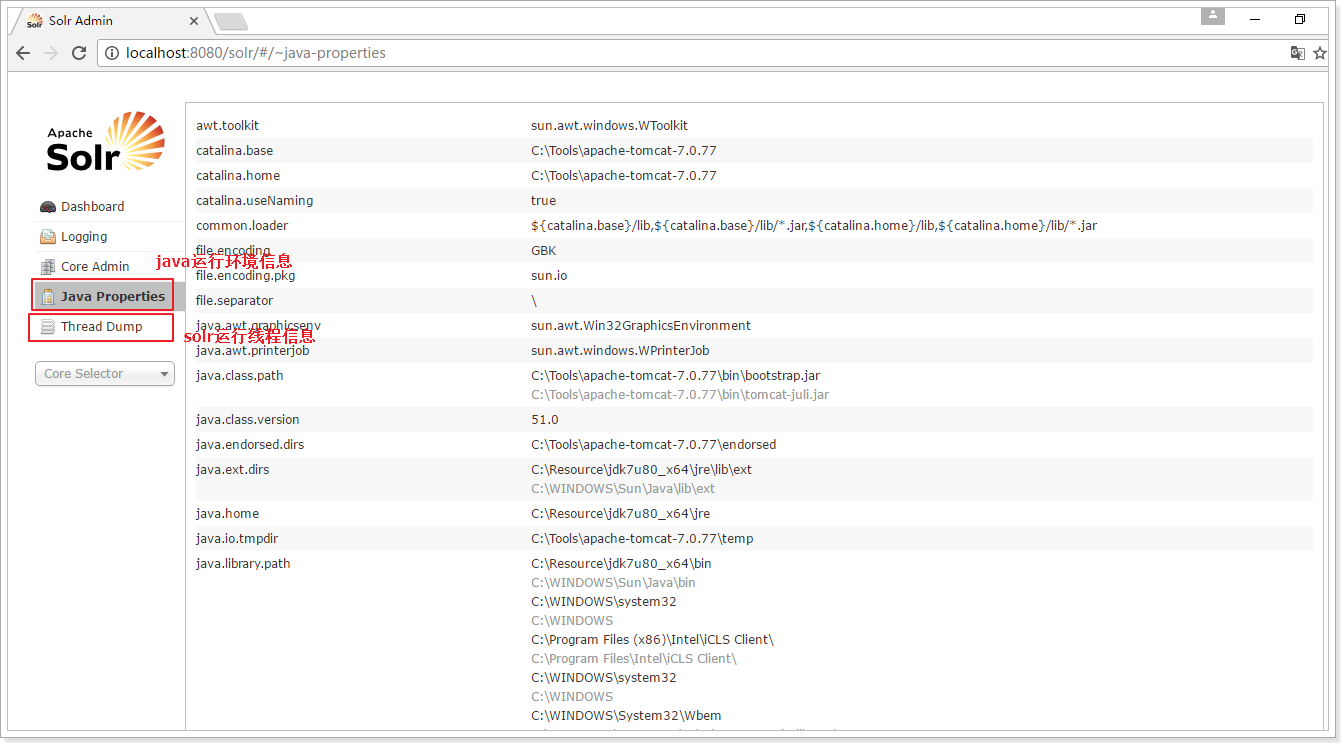
Solr中可以拥有多个Core，也就同时管理多个索引库！就像在MySQL中可以有多个database一样！



默认core的目录

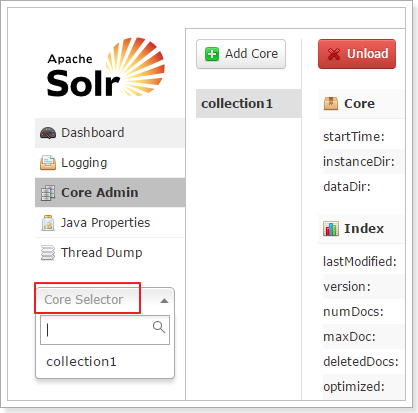


### JavaProperties和ThreadDump

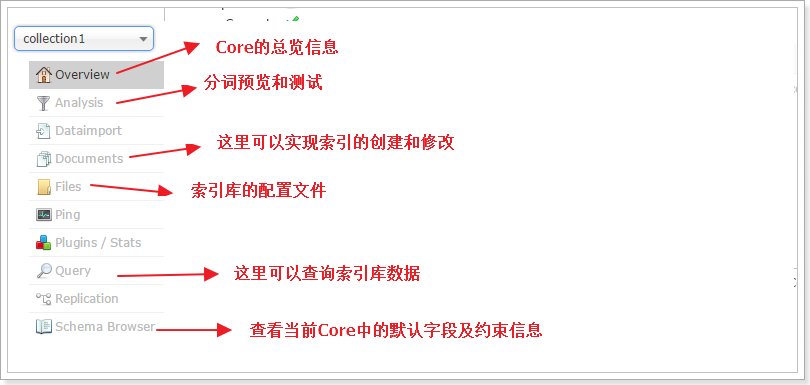


### CoreSelector（Core选择器）

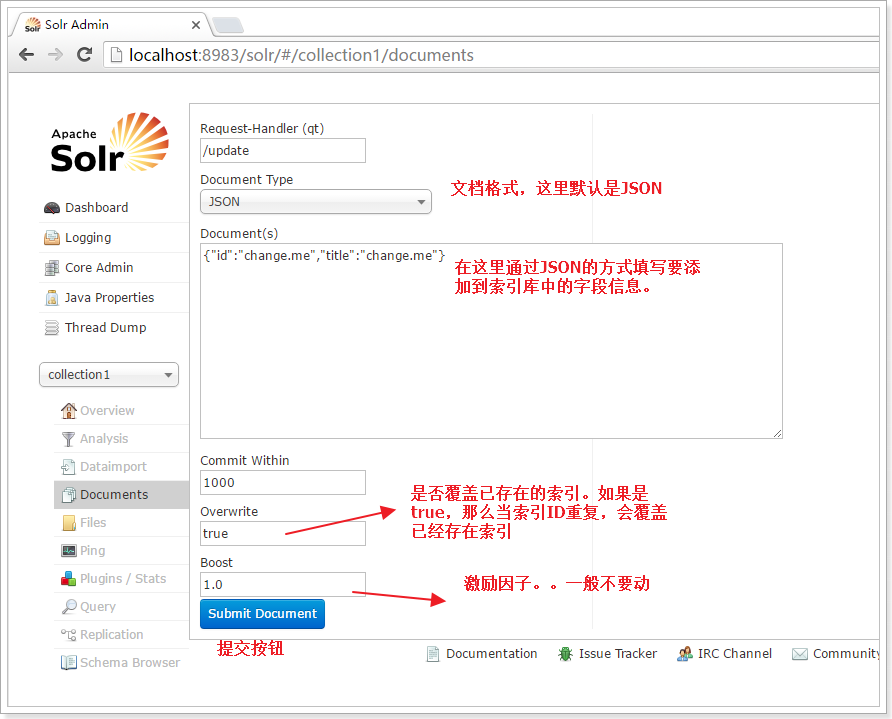
#### 界面预览



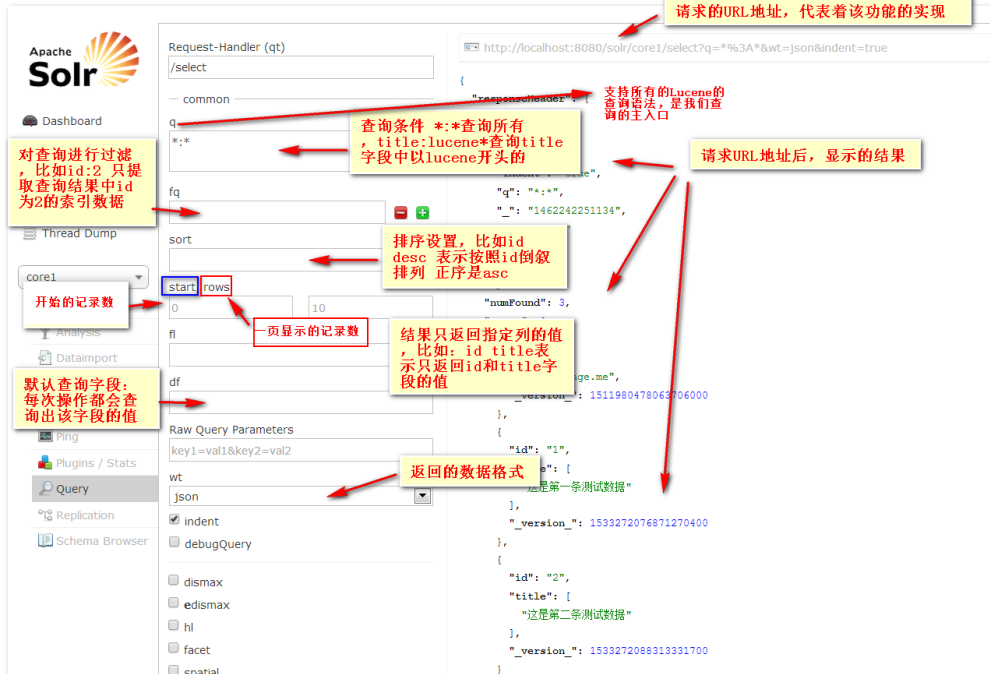
这里可以看到目前存在的所有Core，并且可以选择其中一个，进行更详细的操作：

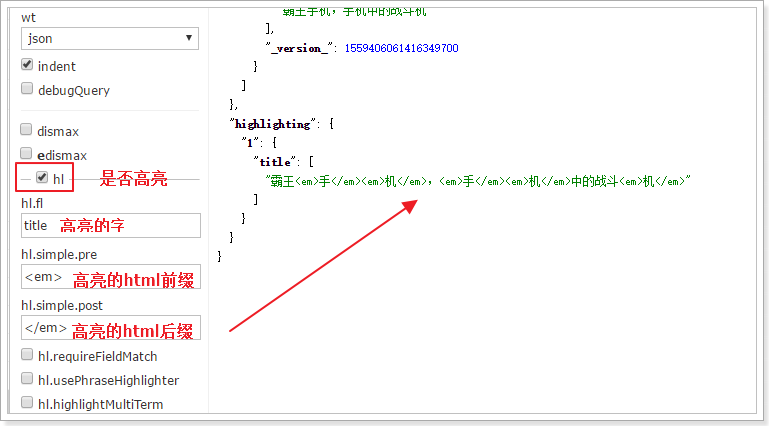


#### 添加修改索引数据



#### 查询索引数据





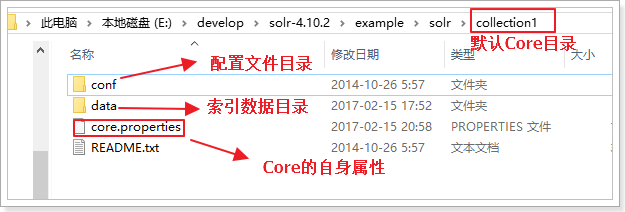
# Solr中的Core详解

## Core的概念

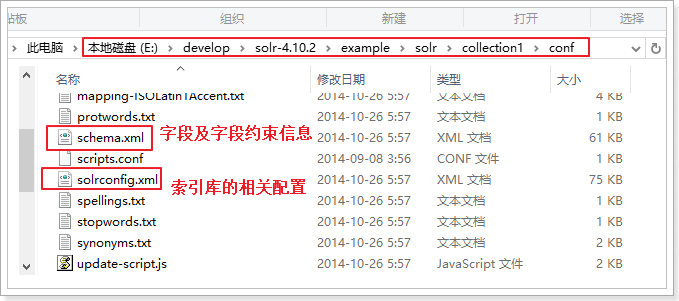


## 目录结构

Core中有两个重要目录：conf和data



conf目录中有两个非常重要的配置文件：schema.xml和solrconfig.xml



## Core的配置文件详解

总共有3个需要我们知道：

core.properties

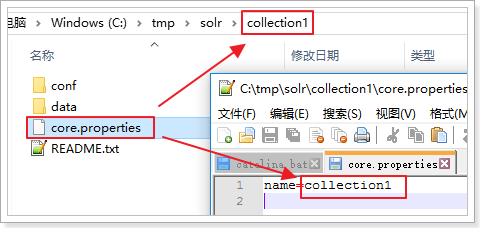
schemal.xml

solrconfig.xml

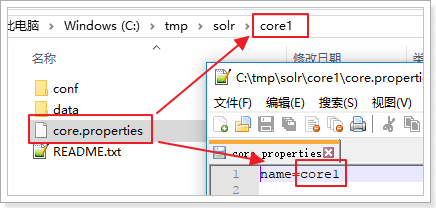
### core.properties

#### core.properties 的作用

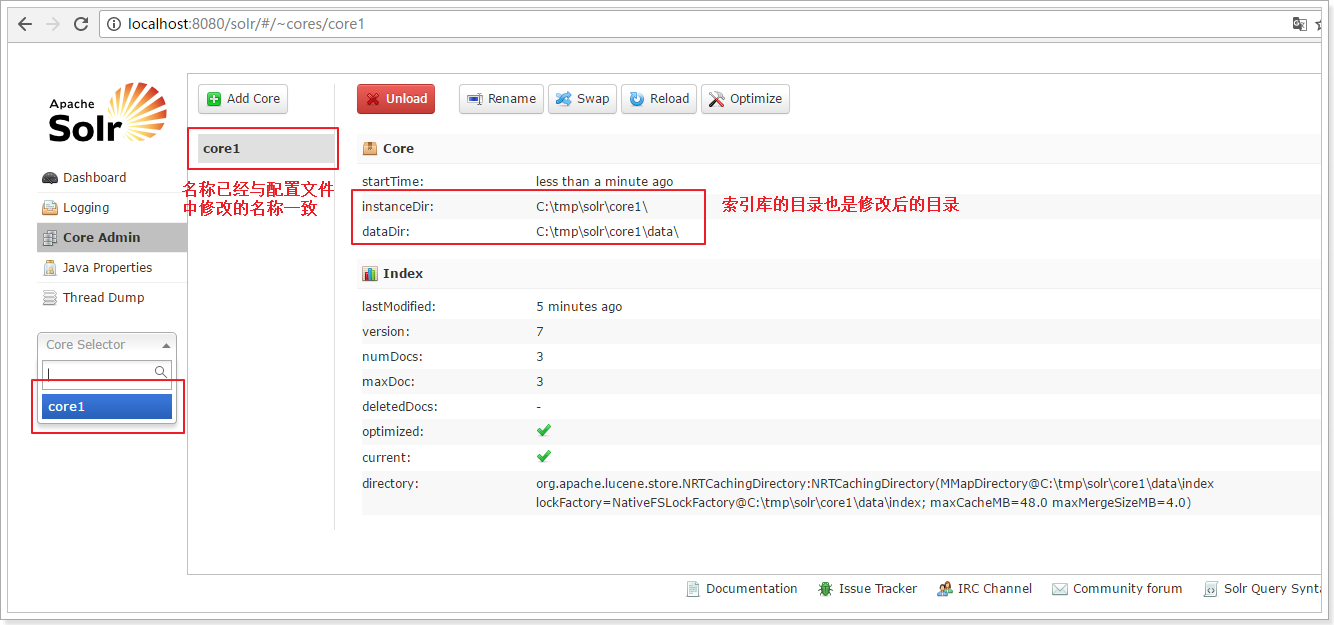
Core的属性文件，记录当前core的名称、索引位置、配置文件名称等信息，也可以不写。



一般要求Core名称跟Core的文件夹名称一致！这里都是collection1，我们可以手动修改这个属性，把Core的名字改成我们喜欢的。

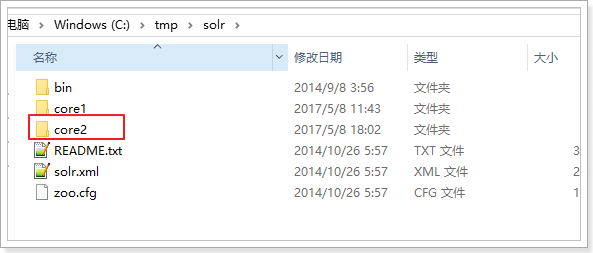


此时重启Tomcat，可以看到core的名字已经改变！

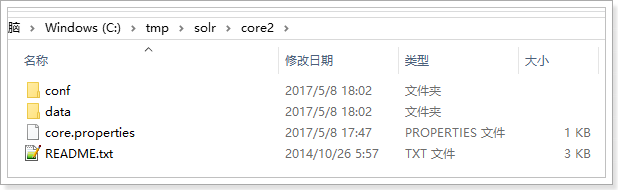


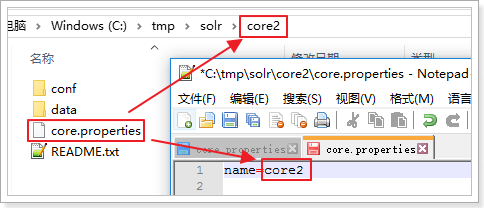
#### 添加多个core

1）在solr目录下创建新的文件夹core2,作为新的core目录



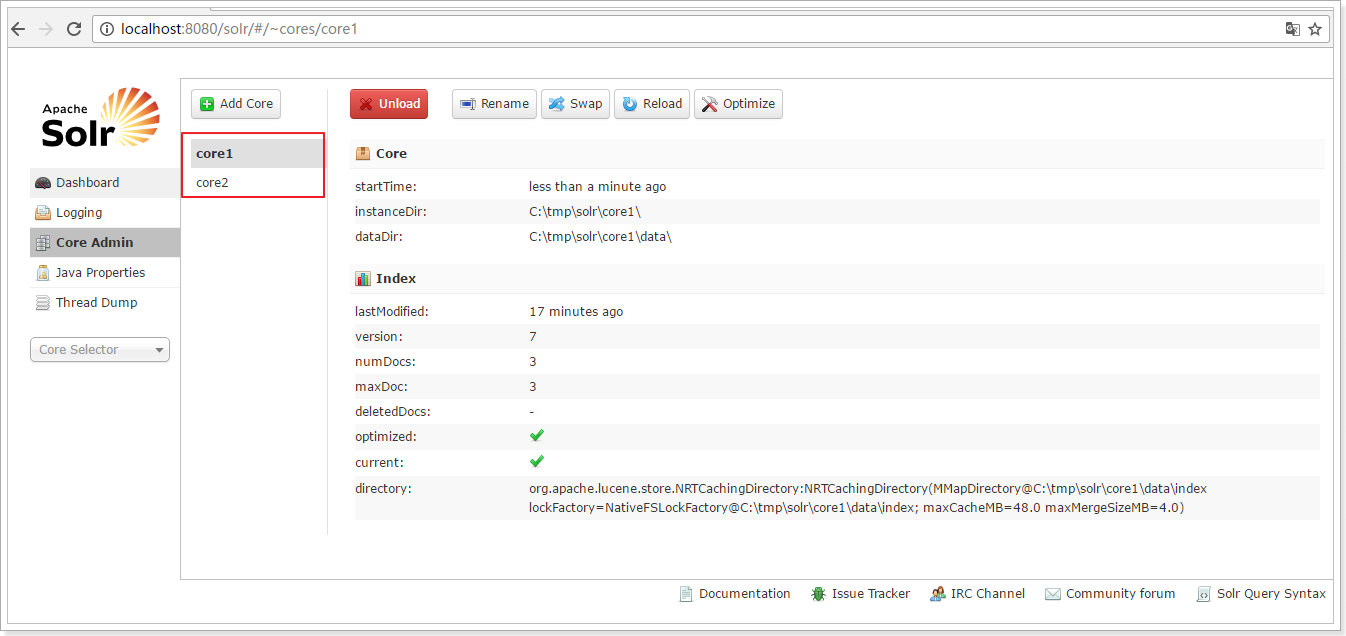
2）在core2下创建conf目录和data目录，并且创建文件core.properties，添加属性：name=core2





3）从core1/conf目录下复制配置文件core2/conf/下

4）重启Tomcat，访问管理页面

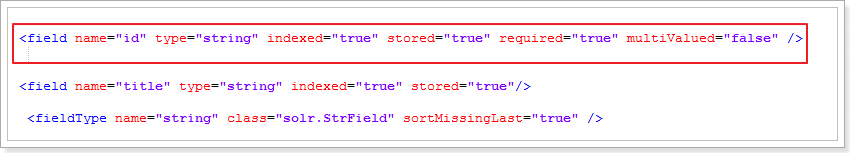


### schema.xml

#### 配置详解

Solr中会提前对文档中的字段进行定义，并且在schema.xml中对这些字段的属性进行约束，例如：字段数据类型、字段是否索引、是否存储、是否分词等等

1）通过Field字段定义字段的属性信息：



属性及含义：

name：字段名称，最好以下划线或者字母开头

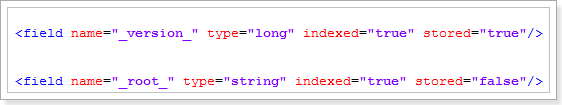
type：字段类型，指向的是本文件中的<fieldType>标签

indexed：是否创建索引

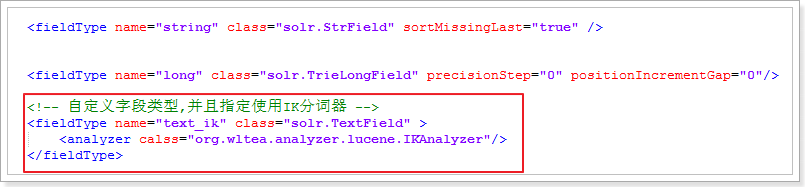
stored：是否被存储

multiValued：是否可以有多个值，如果字段可以有多个值，设置为true

注意：在本文件中，有两个字段是Solr自带的字段，绝对不要删除：\_version\_节点和\_root\_节点



2）通过FieldType指定数据类型



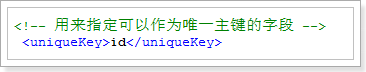
name：字段类型的名称，可以自定义，<field>标签的type属性可以引用该字段，来指定数据类型

class：字段类型在Solr中的类。StrField可索引不可分词。TextField字段可索引，可以分词，所以需要指定分词器

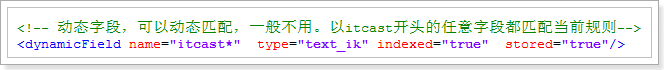
<analyzer>：这个子标签用来指定分词器

3）唯一主键

Lucene中本来是没有主键的。删除和修改都需要根据词条进行匹配。而Solr却可以设置一个字段为唯一主键，这样删改操作都可以根据主键来进行！



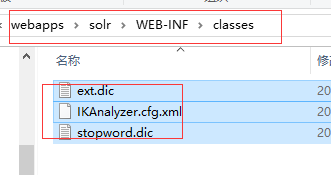
4）动态字段



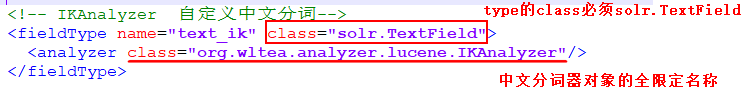
#### 引入IK分词器

1. 引入IK分词器的Jar包

在我们的课前资料中（已在之前引入），三个配置文件在Lucene代码中复制到 **{Tomcat}\webapps\solr\WEB-INF\classes** 文件中



1. 在schemal.xml中自定义fieldType，引入IK分词器

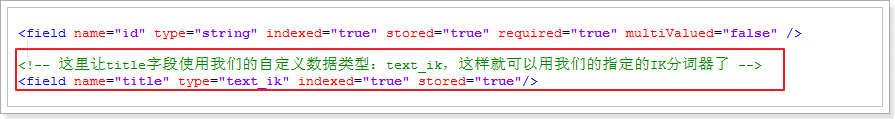


<fieldType name="text\_ik" class="solr.TextField">

<analyzer class="org.wltea.analyzer.lucene.IKAnalyzer"/>

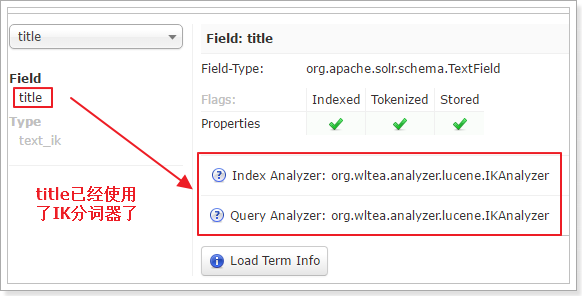
</fieldType>

1. 让字段使用我们的自定义数据类型，引入IK分词器



可以正确分词了：





### solrconfig.xml

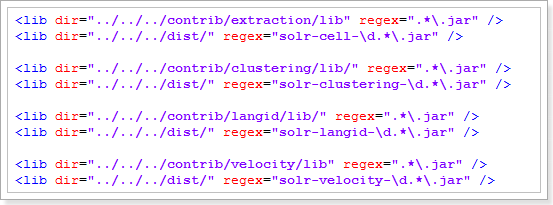
这个配置文件主要配置跟索引库和请求处理相关的配置。solr服务的优化主要通过这个配置文件进行

**<lib/>标签：**

* 用途：配置插件依赖的jar包
* 注意事项：
  + 如果引入多个jar包，要注意包和包的依赖关系，被依赖的包配置在前面
  + 这里的jar包目录如果是相对路径，那么是相对于core所在目录

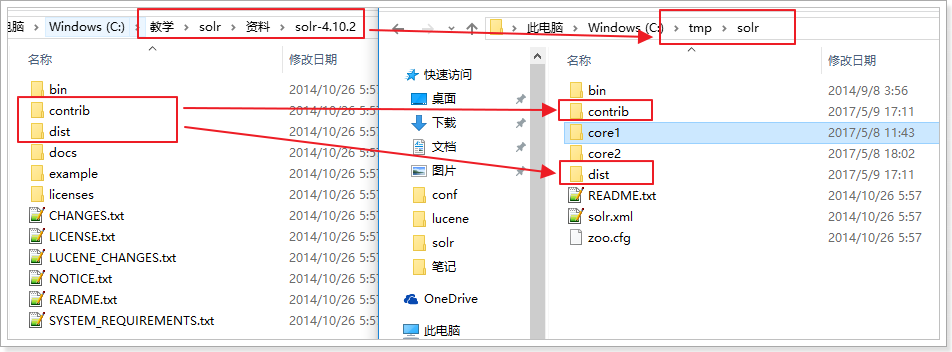
例如，在配置文件中默认有以下配置：

这些是Solr中插件所依赖的Jar包

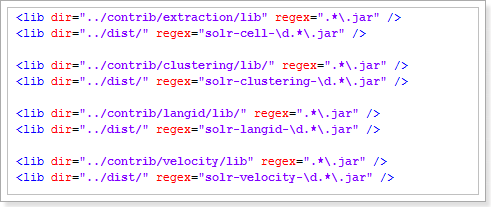


这里的相对路径其实是相对于 core所在目录。

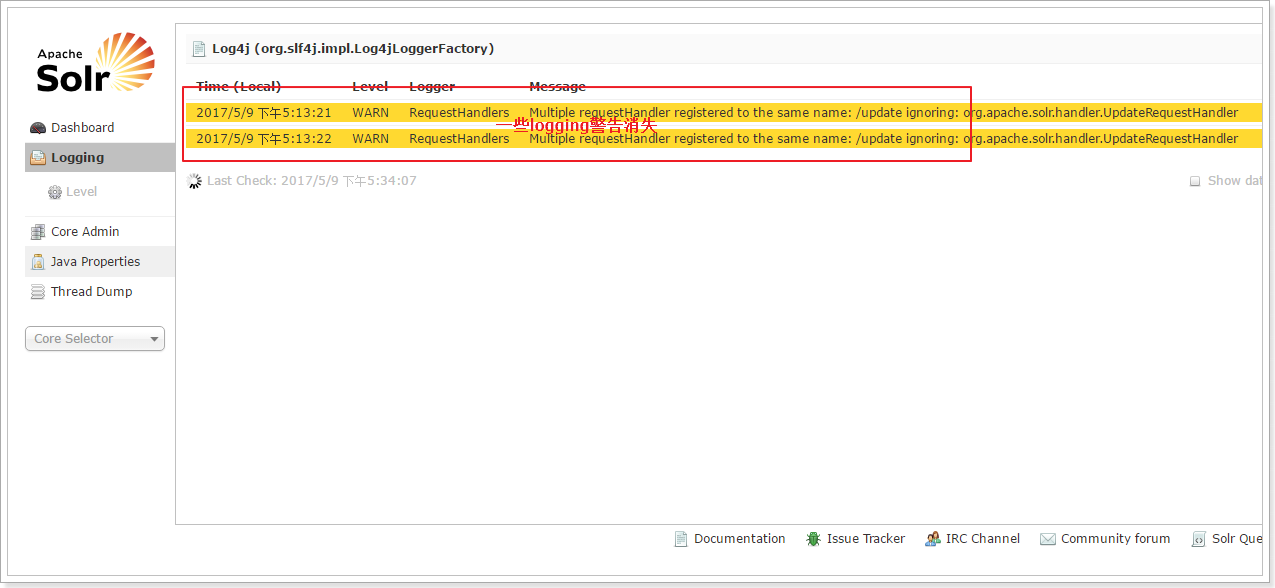
我们把所以来的Jar包复制到solr的HOME中：



然后修改配置文件：



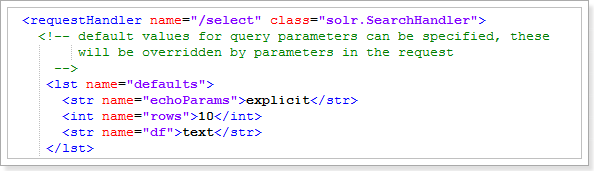
效果：



**<requestHandler/>标签：**

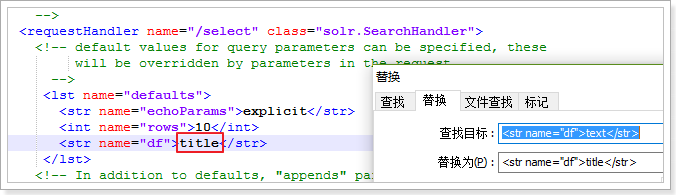
* 用途：配置Solr处理各种请求（搜索/select、更新索引/[update](http://www.07net01.com/tags-update-0.html)、等）的各种参数
* 主要参数：
  + name：请求类型，例如：select、query、get、update
  + class：处理请求的类
  + initParams：可选。引用<initParams>标签中的配置
  + <lst name="defaults">：定义各种缺省的配置，比如缺省的parser、缺省返回条数

例子： 负责搜索请求的Handler



这里有一个默认选项：rows=10 ，就是设置默认查10条数据

<str name=”df”> text</str> 这个是设置默认搜索的字段为：text，我们可以设置为title,因为我们没有text字段



**<initParams/>标签：**

* 用途：为一些requestHandlers定义通用的配置，以便在一个地方修改后，所有地方都生效
* 主要参数：
  + path：指明该配置应用于哪些请求路径，多个 的话用逗号分开，可以用通配符（\*表示一层子路径，\*\*表示无限层）
  + name：如果不指定path，可以指定一个name，然后在<requestHander>配置中可以引用这个name

例子（配置一个缺省的df）：  <initParams path="/update/\*\*,/query,/select,/tvrh,/elevate,/spell,/browse">    <lst name="defaults">      <str name="df">\_text\_</str>    </lst>  </initParams>  
如果配置name，在<requestHander>中引用的例子：<requestHandler name="/dump1" class="DumpRequestHandler"  initParams="myParams"/>

**<updateHandler/>标签：**

* 用途：定义一些更新索引相关的参数，比如定义commit的时机
* 主要参数：
  + autoCommit：定义自动commit的触发条件。如果没配置这个参数，则每次都必须手动commit
    - maxDocs
    - maxTime（毫秒）
    - openSearcher：autoCommit结束后，是否开启一个新的searcher让更改生效。缺省为false
  + autoSoftCommit：定义自动softCommit的触发条件。相关参数同autoCommit
  + listener：配置事件监听器
    - event：监听哪个事件，比如：event="postCommit", event="postOptimize"
    - class：处理的类，可以是自己的实现类。如果是RunExecutableListener，可以配置下面的参数：
      * exe：可[执行文件](http://www.07net01.com/tags-%E6%89%A7%E8%A1%8C%E6%96%87%E4%BB%B6-0.html" \t "_blank)，包括Solr Home的相对路径和文件名
      * dir：工作目录，缺省是“.”
      * wait： 调用者是否等待可执行文件执行结束，缺省是true
      * args：传递给可执行文件的参数
      * env：其他所需要的环境变量
  + updateLog：配置log的保存路径、等
    - dir：保存路径
    - numRecordsToKeep：一个log保存的记录数，缺省为100
    - maxNumLogsToKeep：log的数量，缺省为10
    - num[version](http://www.07net01.com/tags-version-0.html)Buckets：追踪max version的bucket数量（？），缺省为65535

配置这些参数要考虑到搜索的准确度和性能的平衡。^\_^  
注：commit和softCommit：

* commit：正式提交、对索引的修改会被保存到永久[存储](http://www.07net01.com/storage_networking/" \t "_blank)中（比如磁盘），会比较耗时
* softCommit：软提交，对索引的修改会被立即应用到工作中的索引中，即立即生效，但没有保存进磁盘

**<query/>标签：**

* 用途：配置Solr如何处理和返回搜索的相关参数
* 主要参数：
  + filterCache：当搜索带有“fq”参数时，使用这个配置，它保存未经过排序的所有文档
    - class：实现类，有三种：solr.search.LRUCache, solr.search.FastLRUCache, solr.search.LFUCache
    - size：最大保存的记录数量
    - initialSize：初始数量
    - autowarmCount：新Index Searcher启动的时候从旧的Index Searcher缓存拷贝过来的数据量
  + queryResultCache：存储最终的搜索结果（排序后的、有范围的文档id）
    - class：实现类，有三种：solr.search.LRUCache, solr.search.FastLRUCache, solr.search.LFUCache
    - size：最大保存的记录数量
    - initialSize：初始数量
    - autowarmCount：新Index Searcher启动的时候从旧的Index Searcher缓存拷贝过来的数据量
    - maxRamMB：最大分配的容量（兆）
  + documentCache：缓存Lucene Document对象（就是每个文档的fields）
    - class：实现类，有三种：solr.search.LRUCache, solr.search.FastLRUCache, solr.search.LFUCache
    - size：最大保存的记录数量
    - initialSize：初始数量
    - autowarmCount：因为Lucene的内部文档 id 是临时的，所以这个缓存不应该被auto-warm，这个值应该为“0”
  + cache：配置自定义的缓存，通过SolrIndexSearcher类的getCache()方法和name参数调用这个缓存
    - name：被调用时的标识
    - 其他参数同上
  + maxBooleanClauses：BooleanQuery的最大子句数量
  + enableLazyFieldLoading：没有知道被请求的field是否懒加载，true/false
  + useFilterForSortedQuery：如果不是按照score排序，是否从filterCache中获取数据
  + queryResult[Windows](http://www.07net01.com/)ize：配合queryResultCache使用，缓存一个超集。如果搜索请求第10到19条记录，而这个参数是50，那么会缓存0到49条记录
  + queryResultMaxDocsCached：queryResultCache缓存的最大文档数量
  + useColdSearcher：但一个新searcher正在warm-up的时候，新请求是使用旧是searcher（true）还是等待新的search（false）
  + maxWarmingSearchers：定义同时在warn-up的searcher的最大数量
  + listener：监听一些事件并指定处理的类，比如在solr启动时加载一些数据到缓存中，相关参数：
    - event：被监听的事件，比如：firstSearcher是第一个searcher启动、也就是solr启动的事件，newSearcher是当已经有searcher在运行的时候有新searcher启动的事件
    - class：处理类
    - name：="queries"就是需要处理的是query
      * lst, name：针对哪些搜索条件需要处理

**<requestDispatcher/>标签：**

* 用途：控制Solr HTTP RequestDispatche r响应请求的方式，比如：是否处理/select url、是否支持对流的处理、上传文件的大小、如何处理带有cache头的HTTP请求、等等
* 主要参数：
  + handleSelect：true/false，如果是false，则由requestHandler来处理/select请求。因为现在的requestHandler中/select是标配，所以这里应该填false
  + requestParsers：
    - enableRemoteStreaming：是否接受流格式的内容，缺省为ture
    - multipartUploadLimitInKB：multi-part POST请求，上传文件的大小上限（K）
    - formdataUploadLimitInKB：HTTP POST的form data大小上限（K）
    - addHttpRequestToContext：原始的HttpServletRequest对象是否应该被包含在SolrQueryRequest的httpRequest中……一般自定义的插件使用这个参数……
  + httpCaching：如何处理带有cache control头的HTTP请求
    - nerver304：如果设为true（开发阶段），则就算所请求的内容没被修改，也不会返回304，并且下面两个参数会失效
    - lastModFrom：最后修改时间的计算方式，openTime：Searcher启动的时刻；dirLastMod：索引更新的时刻
    - etagSeed：HTTP返回的ETag头内容
    - cacheControl：HTTP返回的Cache-Control头内容

**<updateProcessor/>和<updateProcessorChain/>标签：**

* 用途：配置处理update请求的[处理器](http://www.07net01.com/tags-%E5%A4%84%E7%90%86%E5%99%A8-0.html" \t "_blank)、处理器链。如果不配置的话，Solr会使用缺省的三个处理器：
  + LogUpdateProcessorFactory：追踪和记录日志
  + DistributedUpdateProcessorFactory：分流update请求到不同的node，比如SolrCloud的情况下把请求分配给一个shard的leader，然后把更新应用到所有replica中
  + RunUpdateProcessorFactory：调用Solr的内部API执行update操作
* 如果需要自定义update处理器：
  + updateProcessor：
    - class：负责处理的类
    - name：名字，给updateProcessorChain引用是使用
  + updateProcessorChain：
    - name：自己的名字标记
    - processor：指定updateProcessor的name，多个的话用逗号“,”分开

# SolrJ的使用

## 概述

SolrJ是Apache官方提供的一套Java开发的，访问Solr服务的API，通过这套API可以让我们的程序与Solr服务产生交互，让我们的程序可以实现对Solr索引库的增删改查！

SolrJ的官方wiki地址：<https://wiki.apache.org/solr/Solrj>

添加依赖：

|  |
| --- |
| <dependencies>  <!-- Junit单元测试 -->  <dependency>  <groupId>junit</groupId>  <artifactId>junit</artifactId>  <version>4.12</version>  </dependency>  <dependency>  <groupId>org.apache.solr</groupId>  <artifactId>solr-solrj</artifactId>  <version>4.10.2</version>  </dependency>  <!-- Solr底层会使用到slf4j日志系统 -->  <dependency>  <groupId>org.slf4j</groupId>  <artifactId>slf4j-log4j12</artifactId>  <version>1.7.22</version>  </dependency>  <dependency>  <groupId>commons-logging</groupId>  <artifactId>commons-logging</artifactId>  <version>1.2</version>  </dependency>  </dependencies>  <build>  <plugins>  <!-- java编译插件 -->  <plugin>  <groupId>org.apache.maven.plugins</groupId>  <artifactId>maven-compiler-plugin</artifactId>  <version>3.2</version>  <configuration>  <source>1.7</source>  <target>1.7</target>  <encoding>UTF-8</encoding>  </configuration>  </plugin>  </plugins>  </build> |

## 添加或修改索引库数据

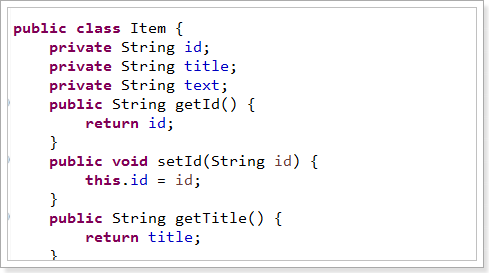
### Document形式



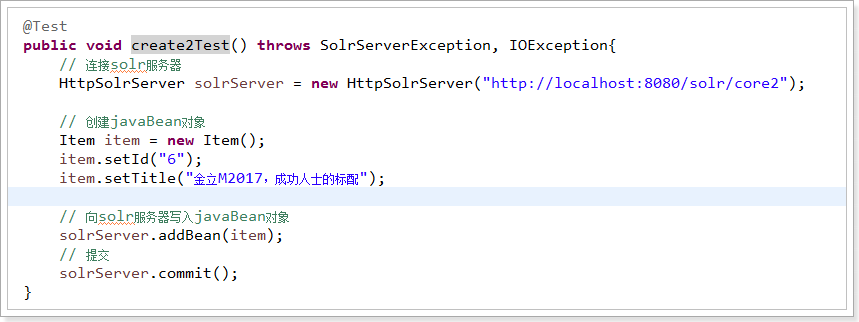
这种方式已经比lucene的实现方式更加简单，但是依然存在可优化的空间，比如：本地数据库中的数据查询出以后被包装成javaBean，如果采用此方式，必须把javaBean拆解到documen对象中。那么solr有没有可解决的方案呢？答案是肯定的，solrServer除了提供add方法外，还提供了addBean方法。

### 注解和JavaBean形式

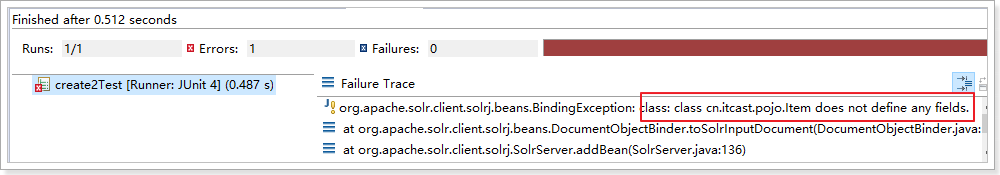
1）创建JavaBean



2）直接通过SolrServer添加JavaBean

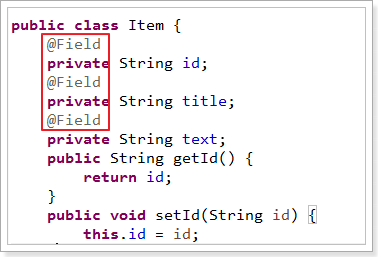


3）执行测试



这是因为javaBean没有使用注解标注那些属性是solr的Field字段

1. 修改javaBean，添加@Field注解



## 删除索引库数据

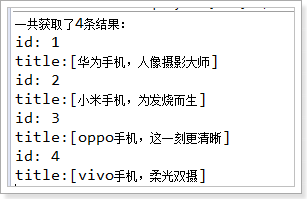
删除索引可以根据ID删除，也可以写一个查询条件，匹配到条件的都会被删除

|  |
| --- |
| @Test  **public** **void** deleteTest() **throws** SolrServerException, IOException {  // 连接solr服务器  HttpSolrServer solrServer = **new** HttpSolrServer("http://localhost:8080/solr/core2");  // 删除索引  // solrServer.deleteById("6");  solrServer.deleteByQuery("title:金立");  // 提交  solrServer.commit();  } |

## 查询索引库数据

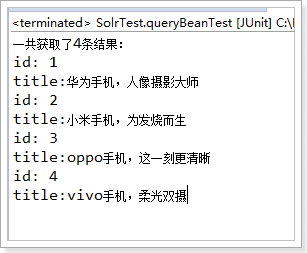
### 以Document形式返回查询结果

|  |
| --- |
| @Test  **public** **void** queryTest() **throws** SolrServerException, IOException {  // 连接solr服务器  HttpSolrServer solrServer = **new** HttpSolrServer("http://localhost:8080/solr/core2");    // 创建查询条件对象  SolrQuery params = **new** SolrQuery("\*:\*");  // 执行查询，获取响应数据  QueryResponse response = solrServer.query(params);  // 获取结果集数据  SolrDocumentList list = response.getResults();  // 处理结果集  System.***out***.println("一共获取了" + list.size()+"条结果：");  **for** (SolrDocument solrDocument : list) {  System.***out***.println("id: " + solrDocument.getFieldValue("id"));  System.***out***.println("title:" + solrDocument.getFieldValue("title"));  }  } |



### 以JavaBean形式返回查询结果

|  |
| --- |
| @Test  **public** **void** queryBeanTest() **throws** SolrServerException, IOException {  // 连接solr服务器  HttpSolrServer solrServer = **new** HttpSolrServer("http://localhost:8080/solr/core2");    // 创建查询条件对象  SolrQuery params = **new** SolrQuery("\*:\*");  // 执行查询，获取响应数据  QueryResponse response = solrServer.query(params);  // 获取结果集数据  List<Item> list = response.getBeans(Item.**class**);  // 处理结果集  System.***out***.println("一共获取了" + list.size()+"条结果：");  **for** (Item item : list) {  System.***out***.println("id: " + item.getId());  System.***out***.println("title:" + item.getTitle());  }  } |



### SolrQuery对象的特殊查询

在创建SolrQuery时，我们填写的Query语句，可以有以下高级写法：

查询语句中如果有特殊字符，需要转义，可以使用：””

1、匹配所有文档：\*:\* （通配符？和\*：“\*”表示匹配任意字符；“？”表示匹配出现的位置）

2、布尔操作：AND、OR和NOT布尔操作（推荐使用大写，区分普通字段）

3、子表达式查询（子查询）：可以使用“()”构造子查询。 比如：(query1 AND query2) OR (query3 AND query4)

4、相似度查询：

（1）默认相似度查询：title:appla~，此时默认编辑距离是2

（2）指定编辑距离的相似度查询：对模糊查询可以设置编辑距离，可选0~2的整数：title:appla~1。

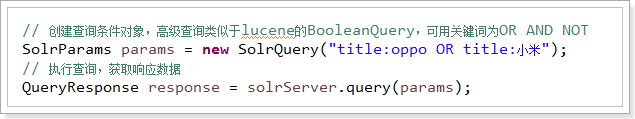
5、范围查询（Range Query）：Lucene支持对数字、日期甚至文本的范围查询，并且两端范围。结束的范围可以使用“\*”通配符。

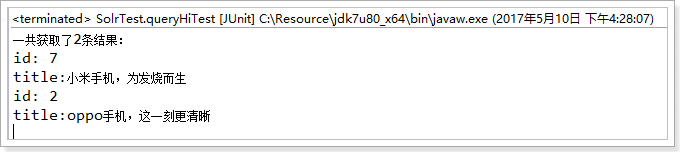
（1）日期范围（ISO-8601 时间GMT）：a\_begin\_date:[1990-01-01T00:00:00.000Z TO 1999-12-31T24:59:99.999Z]

（2）数字：salary:[2000 TO \*]

（3）文本：entryNm:[a TO a]

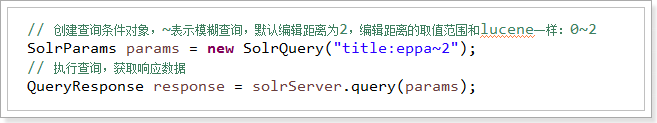
布尔查询：

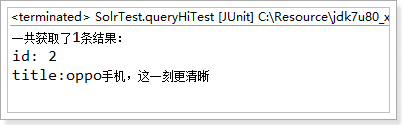




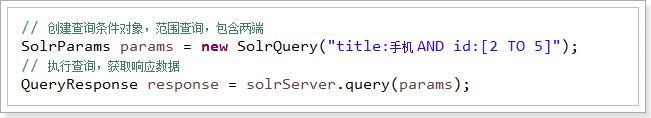
相似度查询：

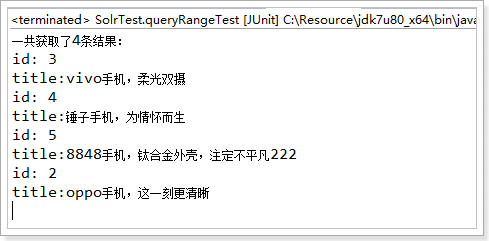
Solr的相似度查询与Lucene一样的！





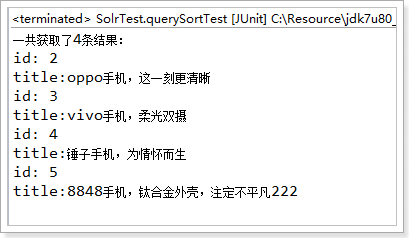
范围查询：两个边界都是包含的！





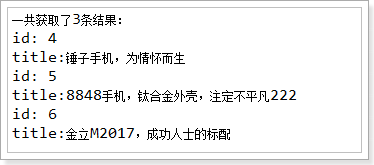
### SolrQuery实现排序

|  |
| --- |
| @Test  **public** **void** querySortTest() **throws** SolrServerException, IOException {  // 连接solr服务器  HttpSolrServer solrServer = **new** HttpSolrServer("http://localhost:8080/solr/core2");    // 创建查询条件对象，范围查询，包含两端  SolrQuery query = **new** SolrQuery("age:[2 TO 5]");  // 设置查询的排序参数，1-排序的字段名，2-排序方式（ORDER：asc desc）  query.setSort("id", ORDER.***asc***);  // 执行查询，获取响应数据  QueryResponse response = solrServer.query(query);    // 获取结果集数据  List<Item> list = response.getBeans(Item.**class**);  // 处理结果集  System.out.println("总记录数 numFound："+response.getResults().getNumFound());  **for** (Item item : list) {  System.***out***.println("id: " + item.getId());  System.***out***.println("title:" + item.getTitle());  }  } |



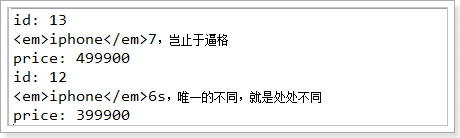
### SolrQuery实现分页

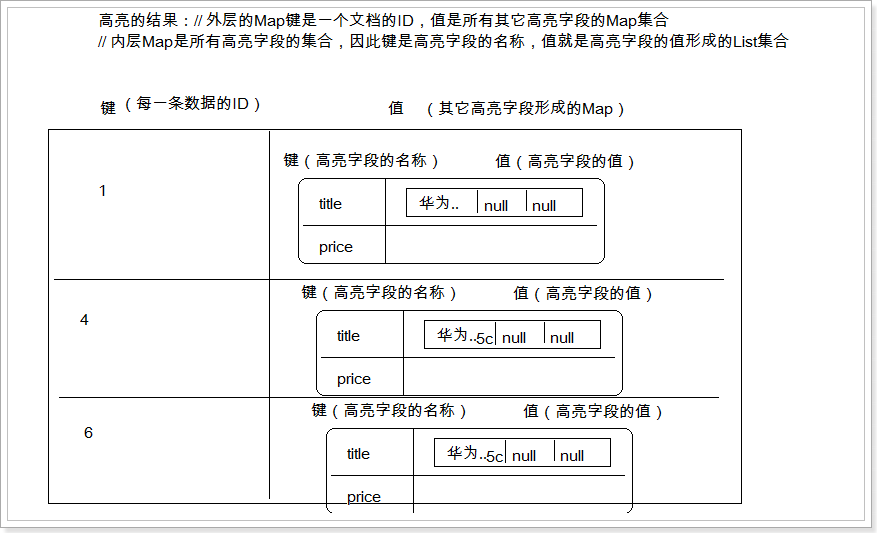
|  |
| --- |
| @Test  **public** **void** queryPageTest() **throws** SolrServerException, IOException {  // 准备分页参数  **int** pageNum = 2; //页码  **int** pageSize = 3; //每页条数    // 连接solr服务器  HttpSolrServer solrServer = **new** HttpSolrServer("http://localhost:8080/solr/core2");    // 创建查询条件对象，范围查询，包含两端  SolrQuery query = **new** SolrQuery("title:\*");  // 设置查询的排序参数，1-排序的字段名，2-排序方式（ORDER：asc desc）  query.setSort("id", ORDER.***asc***);  query.setStart((pageNum-1)\*pageSize);//设置起始条数  query.setRows(pageSize);//设置每页条数  // 执行查询，获取响应数据  QueryResponse response = solrServer.query(query);    // 获取结果集数据  List<Item> list = response.getBeans(Item.**class**);  // 处理结果集  System.out.println("总记录数 numFound："+response.getResults().getNumFound());  **for** (Item item : list) {  System.***out***.println("id: " + item.getId());  System.***out***.println("title:" + item.getTitle());  }  } |



### SolrQuery实现高亮显示

|  |
| --- |
| @Test  **public** **void** testHighLightingQuery() **throws** SolrServerException, IOException{    // 初始化solrj服务  HttpSolrServer server = **new** HttpSolrServer("http://localhost:8080/solr/core2");  // 设置查询条件  SolrQuery query = **new** SolrQuery("title:iphone");  // 设置前置标签  query.setHighlightSimplePre("<em>");  // 设置后置标签  query.setHighlightSimplePost("</em>");  // 添加高亮字段  query.addHighlightField("title");  // 执行查询  QueryResponse queryResponse = server.query(query);    // 处理返回结果  List<Item> items = queryResponse.getBeans(Item.**class**);  // 外层的Map，key：id，value：id以外的其他高亮字段，可能有多个，也是一个Map  // 内层的Map，key：高亮字段的名称，value：字段的内容，集合  Map<String, Map<String, List<String>>> highlighting = queryResponse.getHighlighting();    // 遍历map，获取结果  /\*Set<String> ids = highlighting.keySet();  for (String id : ids) {  System.out.println("id: " + id);;  // 获取高亮字段的集合  Map<String, List<String>> map = highlighting.get(id);  // 获取高亮字段  System.out.println(map.get("title").get(0));  // 因为price不是高亮字段，所以打印出的内容为null  System.out.println(map.get("price"));  }\*/  **for** (Item item : items) {  System.***out***.println("id: " + item.getId());  // 根据item的id获取对应的高亮字段集合：Map，根据字段名称title获取对应的高亮内容集合：List，title只有一条数据，所以获取list集合中的第0条数据  System.***out***.println(highlighting.get(item.getId()).get("title").get(0));  System.***out***.println("price: " + item.getPrice());  }  } |





### 复杂查询

|  |
| --- |
| **public** **void** testSearchFuzaIndex() **throws** Exception {  SolrServer solrServer = **new** HttpSolrServer("http://localhost:8080/solr/");  SolrQuery query = **new** SolrQuery();  // q ： 设置筛选条件  query.setQuery("台灯");  // fq ： 设置过滤条件，可以有多个  query.addFilterQuery("user\_age:[20 TO 50]", "user\_name:豆豆");  // sort ： 设置排序 desc asc  query.setSort("user\_age", ORDER.***desc***);  // start, rows ： 分页条件  query.setStart(0);  query.setRows(5);  // fl ： 设置查询显示的列表  query.setFields("id", "user\_name", "user\_age");    // df ：设置默认搜索域  query.set("df", "user\_keywords");  // hl ： 设置开启高亮  query.setHighlight(**true**);  // hl.fl ： 设置高亮显示的域  query.addHighlightField("user\_name");  // hl.simple.pre ： 设置高亮显示前缀  query.setHighlightSimplePre("<em>");  // hl.simple.post ： 设置高亮显示的后缀  query.setHighlightSimplePost("</em>");  // 取结果  QueryResponse response = solrServer.query(query);  SolrDocumentList list = response.getResults();  System.***out***.println("总记录： " + list.getNumFound());  **for** (SolrDocument solrDocument : list) {  System.***out***.println(solrDocument.get("id"));  // 取高亮结果集  Map<String, Map<String, List<String>>> highlighting = response.getHighlighting();  List<String> hlist = highlighting.get(solrDocument.get("id")).get("user\_name");  String productName = "";  **if** (hlist != **null** && hlist.size() > 0) {  productName = hlist.get(0);  } **else** {  productName = (String) solrDocument.get("user\_name");  }  System.***out***.println(productName);  System.***out***.println(solrDocument.get("user\_age"));  }  } |