Solr

# Solr简介

Solr是一个独立的[企业级搜索](http://baike.baidu.com/view/4173560.htm)应用服务器，它对外提供类似于Web-service的API接口。用户可以通过http请求，向搜索引擎服务器提交一定格式的XML文件，生成索引；也可以通过Http Get操作提出查找请求，并得到XML格式的返回结果。Solr是一个高性能，采用Java开发，Solr基于Lucene的全文搜索服务器。同时对其进行了扩展，提供了比Lucene更为丰富的查询语言，同时实现了可配置、可扩展并对查询性能进行了优化，并且提供了一个完善的功能管理界面，是一款非常优秀的[全文搜索引擎](http://baike.baidu.com/view/2007665.htm)。

## solr 工作方式

文档通过Http利用XML 加到一个搜索集合中。Solr查询该集合也是通过http收到一个XML/JSON响应来实现。它的主要特性包括：高效、灵活的缓存功能，垂直搜索功能，高亮显示搜索结果，通过索引复制来提高可用性，提供一套强大Data Schema来定义字段，类型和设置[文本分析](http://baike.baidu.com/view/3488135.htm)，提供基于Web的管理界面等。

## Solr和Lucene的关系

Solr是一个高性能，采用Java开发，基于Lucene的全文搜索服务器。同时对其进行了扩展，提供了比Lucene更为丰富的查询语言，同时实现了可配置、可扩展并对查询性能进行了优化，并且提供了一个完善的功能管理界面，是一款非常优秀的[全文搜索引擎](https://www.baidu.com/s?wd=%E5%85%A8%E6%96%87%E6%90%9C%E7%B4%A2%E5%BC%95%E6%93%8E&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1Y3PHIBnAP-nWIBrjcLrAwh0AP8IA3qPjfsn1bkrjKxmLKz0ZNzUjdCIZwsrBtEXh9GuA7EQhF9pywdQhPEUiqkIyN1IA-EUBt1rHckP1TdnjTzn1TdPWm3rjR)。

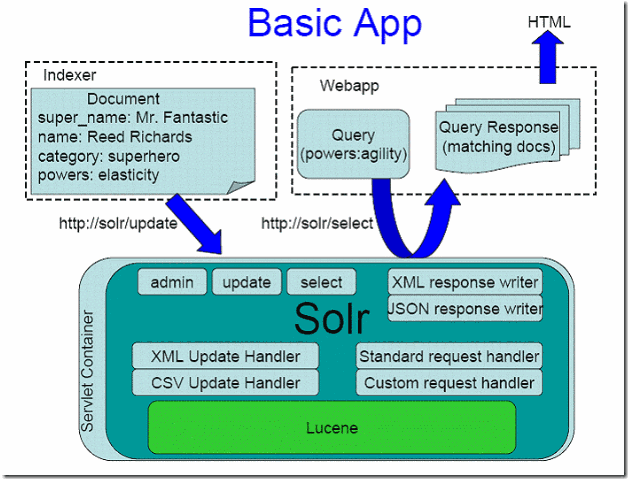
**··** Solr和Lucene的本质区别有以下三点：搜索服务器，企业级和管理。Lucene本质上是搜索库，不是独立的应用程序，而Solr是。Lucene专注于搜索底层的建设，而Solr专注于企业应用。Lucene不负责支撑搜索服务所必须的管理，而Solr负责。所以说，一句话概括Solr: Solr是Lucene面向企业搜索应用的扩展。

## Solr 工作原理

Solr对外提供标准的http接口来实现对数据的索引的增加、删除、修改、查询。

　　在Solr中，用户通过向部署在servlet 容器中的Solr Web应用程序发送 HTTP 请求来启动索引和搜索。

　　Solr接受请求，确定要使用的适当SolrRequestHandler，然后处理请求。通过 HTTP 以同样的方式返回响应。  
　　默认配置返回Solr的标准 XML 响应，也可以配置Solr的备用响应格式。



## SOLR索引原理

可以向Solr索引servlet传递四个不同的索引请求：  
　　1) add/update允许向Solr添加文档或更新文档。直到提交后才能搜索到这些添加和更新。  
　　2) commit 告诉Solr，应该使上次提交以来所做的所有更改都可以搜索到。  
　　3) optimize 重构 Lucene 的文件以改进搜索性能。索引完成后执行一下优化通常比较好。如果更新比较频繁，则应该在使用率较低的时候安排优化。一个索引无需优化也可以正常地运行。优化是一个耗时较多的过程。  
　　4) delete 可以通过 id 或查询来指定。按 id 删除将删除具有指定 id 的文档；按查询删除将删除查询返回的所有文档。

## SOLR CORE

Solr里面的core就像数据库里面的一个表，用来管理索引和相关配置。

core.properties

#Written by CorePropertiesLocator

#Sat Mar 15 15:49:01 CST 2014

name=core0

config=solrconfig.xml

schema=schema.xml

dataDir=data

# solr 配置相关

## schema.xml

schema.xml 是用来定义索引数据中的域的，包括域名称，域类型，域是否索引，是否分词，是否存储，是否标准化即 Norms ，是否存储项向量等等。

schema.xml 配置文件的根元素就是 schema, 有个 name 属性， name 属性值可以随便配，根元素没什么好说的， schema 元素下主要有两个标签元素即 field 和fieldType,field 表示域，用来定义域， fieldType 用来定义域类型。

### field 元素有很多属性可以配置：

**name:**表示域的名称，是强制必须有的属性

**type:**域类型的名称，与 fieldType 元素的 name 属性值对应，也是强制必须有的属性，不可省略

**indexed:** true 即表示需要对该域进行索引，一般如果你需要在该域上进行查询或排序时，则需要配置为 true, 默认值为 false

**stored:** 表示是否需要把域值存储到硬盘上，方便你后续查询时能再次提取出来原样显示给用户

**docValues:**表示此域是否需要添加一个 docValues 域，这对 facet 查询， group 分组，排序， function 查询有好处，尽管这个属性不是必须的，但他能加快索引数据加载，对 NRT 近实时搜索比较友好，且更节省内存，但它也有一些限制，比如当前docValues 域只支持 strField,UUIDField,Trie\*Field 等域，且要求域的域值是单值不能是多值域

**multiValued:**表示这个域是否可以存储多个值，若设置为 true, 即表示这是一个多值域

**omitNorms:** 此属性若设置为 true ，即表示将忽略域值的长度标准化，忽略在索引过程中对当前域的权重设置，且会节省内存。只有全文本域或者你需要在索引创建过程中设置域的权重时才需要把这个值设为 false, 对于基本数据类型且不分词的域如intFeild,longField,StrField 等默认此属性值就是 true, 否则默认就是 false.

**termVectors:**设置为 true 即表示需要为该 field 存储项向量信息，当你需要MoreLikeThis 功能时，则需要将此属性值设为 true ，这样会带来一些性能提升。

**termPositions:** 是否存储 Term 的起始位置信息，这会增大索引的体积，但高亮功能需要依赖此项设置，否则无法高亮

**termOffsets:** 表示是否存储索引的位置偏移量，高亮功能需要此项配置，当你使用SpanQuery 时，此项配置会影响匹配的结果集

field 里还有两个比较难理解的域，是 Solr 扩展的，在 Lucene 中没有的概念，即dynamicField 动态域和 copyField 复制域：

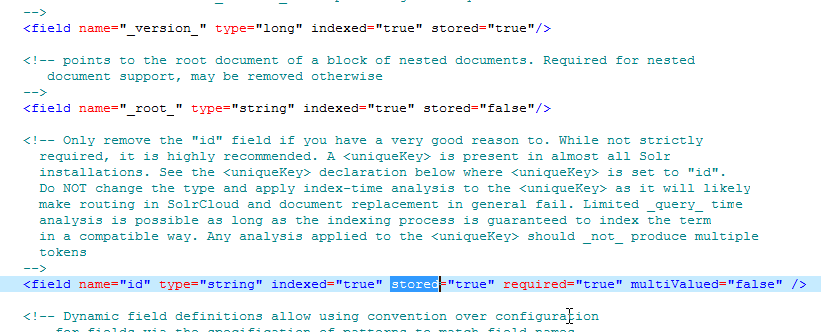
动态域的属性配置跟普通的 field 差不多就不多说了，唯一有点区别就是 name 的属性值，可以用通配符，这样就可以模糊匹配多个域啦，这样设计的目的就是不用频繁的去修改我们的 schema.xml 中的 field 配置去增加 field 域啦，比如之前有个 link\_s域，某一天你想再增加一个 url\_s 域，那你就需要去修改 schema.xml 配置文件，由于schema.xml 修改过后需要重启 tomcat 才能生效，重启即意味着程序的中断，这往往是不可接受的。所以引入动态域来避免频繁添加修改域，但前提是你的域需要符合你提前定义的动态域的域名称命名规则哦

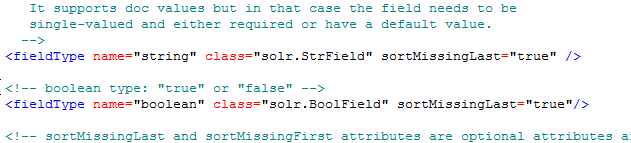
复制域即表示把某个域的值复制到一个目标域上面，那如果把多个域的值复制到一个目标域上面呢，你可以进行多次复制，体现到 XML 配置上就是类似这样的配置：

<copyField source="title" dest="text"/>

<copyField source="body" dest="text"/>

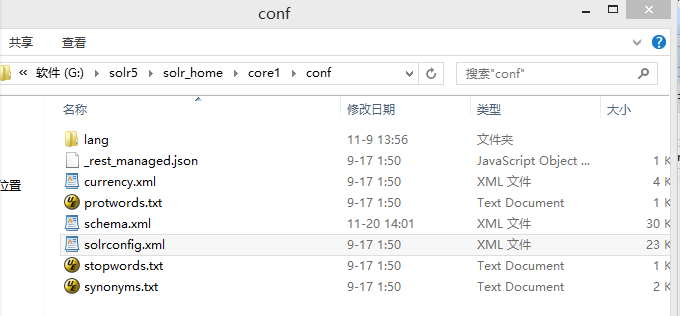
如上配置就表示把 title 和 body 这两个域的值全部复制到 text 这个新域上面，唯一要注意的是，如果你只是复制单个域，那么如果你被复制域本身就是多值域，那么目标域也是多值域，这毋庸置疑，那如果你复制的是多个域，只要其中有一个域是多值域，那么目标域就一定是多值域，这点一定要谨记。





## solrconfig.xml属性介绍

solrconfig.xml  配置文件中包含了很多 solr 自身配置相关的参数 ,solrconfig.xml  配置文件示例可以从 solr home的以下目录找到：



solrconfig.xml 中的配置项主要分以下几大块：

### 依赖的 lucene 版本配置

这决定了你创建的 Lucene 索引结构，因为 Lucene 各版本之间的索引结构并不是完全兼容的，这个需要引起你的注意。

<luceneMatchVersion>5.3.1</luceneMatchVersion>

### 索引创建相关的配置

如索引目录， IndexWriterConfig 类中的相关配置 ( 它决定了你的索引创建性能 )

<filter class="solr.LimitTokenCountFilterFactory" maxTokenCount="10000"/>

maxTokenCount 即在对某个域分词的时候，最多只提取前 10000 个 Token ，后续的域值将被抛弃

<writeLockTimeout>1000</writeLockTimeout>

writeLockTimeout 表示 IndexWriter 实例在获取写锁的时候最大等待超时时间，超过指定的超时时间仍未获取到写锁，则 IndexWriter 写索引操作将会抛出异常。

<maxIndexingThreads>8</maxIndexingThreads>

表示创建索引的最大线程数，默认是开辟 8 个线程来创建索引。

<ramBufferSizeMB>100</ramBufferSizeMB>

表示创建索引时内存缓存大小，单位是 MB, 默认最大是 100M 。

<maxBufferedDocs>1000</maxBufferedDocs>

表示在 document 写入到硬盘之前，缓存的 document 最大个数，超过这个最大值会触发索引的 flush 操作。

### solrconfig.xml 中依赖的外部 jar 包加载路径配置

如： <lib dir="./lib" regex=”lucene-\w+\.jar”/>

这里的 dir 表示一个 jar 包目录路径，该目录路径是相对于你当前 core 根目录的；regex 表示一个正则表达式，用来过滤文件名的，符合正则表达式的 jar 文件将会被加载

<dataDir>${solr.data.dir:}</dataDir>

如果solr\_home 下不存在 core 的话，那 dataDir 默认就是相对于 solr\_home

### 缓存相关配置

缓存包括过滤器缓存，查询结果集缓存， Document 缓存，以及自定义缓存等等

<filterCache class="solr.FastLRUCache"

size="512"

initialSize="512"

autowarmCount="0"/>

用来配置 filter 过滤器的缓存相关的参数

<queryResultCache class="solr.LRUCache"

size="512"

initialSize="512"

autowarmCount="0"/>

用来配置对 Query 返回的查询结果集即 TopDocs 的缓存

<documentCache class="solr.LRUCache"

size="512"

initialSize="512"

autowarmCount="0"/>

用来配置对 Document 中存储域的缓存，因为每次从硬盘上加载存储域的值都是很昂贵的操作，这里说的存储域指的是那些 Store.YES 的 Field 。

### updateHandler 配置即索引更新操作相关配置

<updateHandler class="solr.DirectUpdateHandler2">

指定索引更新操作处理类， DirectUpdateHandler2 是一个高性能的索引更新处理类，它支持软提交。

<updateLog>

 <str name="dir">${solr.ulog.dir:}</str>

 </updateLog>

设置索引库更新日志，默认路径为solr home下面的data/tlog。

<autoCommit>

 <maxTime>15000</maxTime>

 <maxDocs>10000</maxDocs>

 <openSearcher>false</openSearcher>

 </autoCommit>

自动硬提交方式:maxTime：设置多长时间提交一次maxDocs：设置达到多少文档提交一次openSearcher：文档提交后是否开启新的searcher，

 如果false，文档只是提交到index索引库，搜索结果中搜不到此次提交的文档；如果true，既提交到index索引库，也能在搜索结果中搜到此次提交的内容。

### RequestHandler 相关配置

即接收客户端 HTTP 请求的处理类配置，输入的请求会通过请求中的路径被转发到特定的处理器

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

<lst name="defaults">

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

<str name="wt">json</str>

<str name="indent">true</str>

</lst>

</requestHandler>

这个 requestHandler 配置的是请求 URL  /query 跟请求处理类 SearcherHandler 之间的一个映射关系，即你访问 [***http://localhost:8080/solr/coreName/query?q=xxx***](http://localhost:8080/solr/coreName/query?q=xxx)时，会交给 SearcherHandler 类来处理这个 http 请求，你可以配置一些参数来干预SearcherHandler 处理细节。

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

<requestHandler name="/dataimport" class="org.apache.solr.handler.dataimport.DataImportHandler">

<requestHandler name="/replication" class="solr.ReplicationHandler">

<requestHandler name="/spell" class="solr.SearchHandler" startup="lazy">

<requestHandler name="/suggest" class="solr.SearchHandler" startup="lazy">

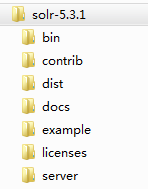
其他的一些 RequestHandler 说明详见实际配置文件。

# 安装

## 单机安装

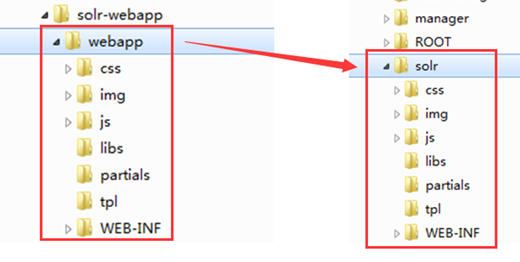
下载地址： <http://archive.apache.org/dist/lucene/solr/>

### 解压

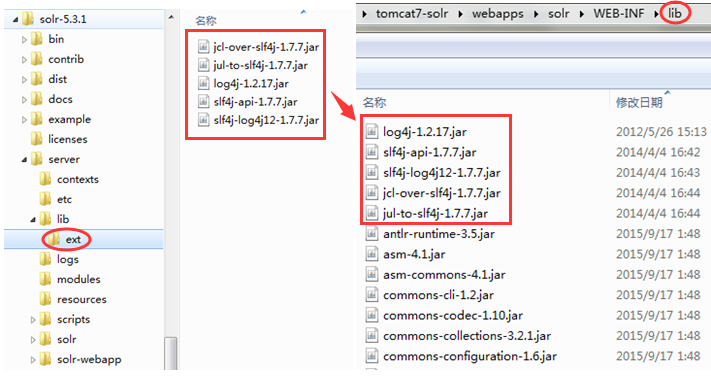


### 拷贝应用

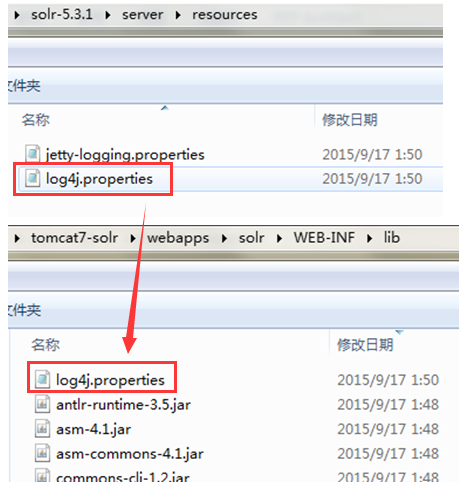
拷贝solr-5.3.1\server\solr-webapp\webapp到tomcat的webapps目录下，并重命名为solr



### 拷贝扩展依赖JAR

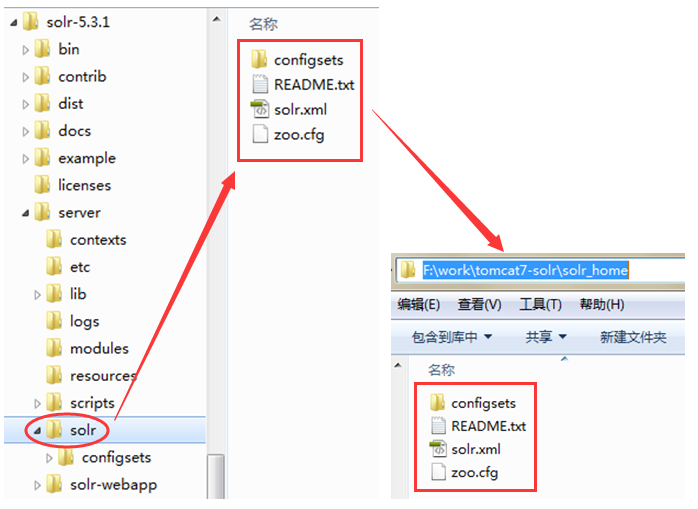


### 拷贝日志配置文件



### 配置HOME目录

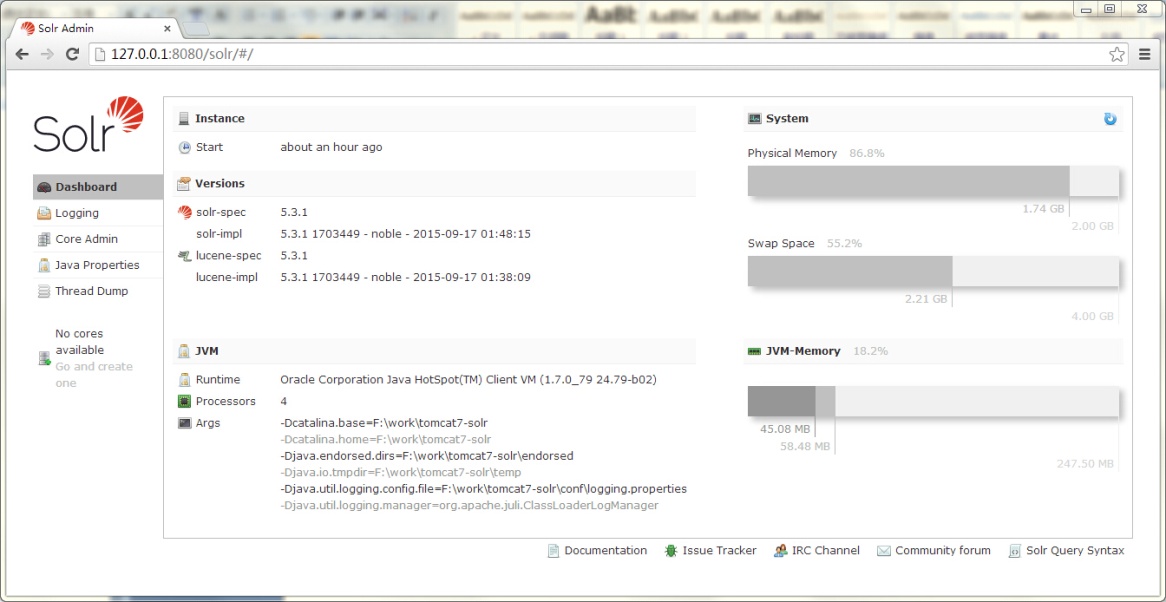
创建HOME目录（D:\denv\solr\_home）并把solr配置文件拷贝到HOME目录中。



### 修改solr应用下的web.xml文件，设置HOME目录路径。



### 启动tomcat，访问：http://127.0.0.1:8080/solr



## 集群安装

solr5.3.1+tomcat8.0.28+zookeeper-3.5.1-alpha

下载地址列表：

<http://mirror.bit.edu.cn/apache/lucene/solr/5.3.1/>

<http://tomcat.apache.org/download-80.cgi>

<http://mirrors.cnnic.cn/apache/zookeeper/zookeeper-3.5.1-alpha/>

本次是在一台服务器上部署3个 zk,4个 tomcat,由于是在同一台服务器上，zk 和 tomcat 的端口不能重复，若分布在不同的服务器上，这一步可以省略。

### 创建目录结构

**1 . 创建目录：F:\solrcloud 可以根据自己需求建立在不同的盘**

**2 . solrcloud 下面目录结构为**

service1   service2    service3

    tomcat1 tomcat2 tomcat3 tomcat4

solr\_home1 solr\_home2 solr\_home3 solr\_home4

**3 . service中是 zk 的目录，三个目录结构一样，目录结构为**

data      datalog       logs       zookeeper-3.5.1-alpha

### ZK安装

**1 . 进入zk的conf目录**

.. solrcloud\service1\zookeeper-3.5.1-alpha\conf

**2 . 新建zoo.cfg并编辑此文件，文件内容如下（或者复制将zoo\_sample.cfg改名为zoo.cfg）**

initLimit=5

syncLimit=2

clientPort=2181

tickTime=2000

dataDir= F:\\solrcloud\\service1\\data

dataLogDir= F:\\solrcloud\\service1\\datalog

//集群信息（服务器编号，服务器地址，LF通信端口，选举端口）

server.1=[localhost](http://www.07net01.com/tags-localhost-0.html):2888:3888

server.2=localhost:2889:3889

server.3=localhost:2890:3890

initLimit：集群中的follower服务器(F)与leader服务器(L)之间初始连接时能容忍的最多心跳数（tickTime的数量）。

tickTime： Zookeeper 服务器之间或客户端与服务器之间维持心跳的时间间隔。

syncLimit：集群中的follower服务器与leader服务器之间请求和应答之间能容忍的最多心跳数。

**3 . 在data目录新建myid文件，并写入值**

文件中写入一个数字，每台zk不能重复，代表zk的唯一性，在zoo.cfg 中需要配置到。

**4 . 启动zk**

分别进入 F:\solrcloud\service1\zookeeper-3.5.1-alpha\bin

F:\solrcloud\service2\zookeeper-3.5.1-alpha\bin

F:\solrcloud\service3\zookeeper-3.5.1-alpha\bin

双击zkServer.cmd启动，启动过程中由于集群未完全正常启动，会报连接错误，可以暂时不管。 zkServer.sh status（查看状态）

### 安装Tomcat

**1 . 复制单机的tomcat:** 复制D:\System\apache-tomcat-8.0.28 至 F:\solrcloud 下 分别更名为tomcat1、tomcat2、tomcat3、tomcat4

**2 . 编辑tomcat中solr项目的web.xml文件，  找到env-entry节点，编辑为以下内容**

<env-entry>

       <env-entry-name>solr/home</env-entry-name>

       <env-entry-value> F:\solrcloud\solr\_home1</env-entry-value>(solr的用户目录，用于存储索引等，接下来会介绍,每个tomcat需要不一样的路径)

       <env-entry-type>java.lang.String</env-entry-type>

</env-entry>

**3 . 添加solr\_home内容**

复制 D:\work\solr\_home 目录到F:\solrcloud 下，分别命名为solr\_home 1、solr\_home 2、solr\_home 3、solr\_home 4。并清理除原来的core。

**4.修改solr.xml**

 <int name="hostPort">${jetty.port:9091}</int>（修改成你当前的tomcat的运行端口，每个solr\_home对应的tomcat端口不一样，当然是在伪集群的情况下，如果你是多台机器的话请忽略）

**5.新建配置文件目录（用于zk的集群配置管理），我是直接在solr\_home1中新建一个collection1文件夹，如下：**

..\solr\_home1\collection1（此文件夹只需要建一个，不需要在每个机器上建，他会通过zk分发到各个机器）

将solr原包中../solr/server/solr/configsets/basic\_configs/\* 拷贝到 ../solr\_home1/collection1/ 下。

**6 . 修改tomcat的文件catalina.bat文件**

一台添加： set JAVA\_OPTS=-Dsolr.solr.home=D:/solrcloud/solr\_home1 -Dbootstrap\_confdir=F:/solrcloud/solr\_home1/collection1/conf -Dcollection.configName=myconf -DnumShards=2 -DzkHost=localhost:2181,localhost:2182,localhost:2183

其它添加： set JAVA\_OPTS=-Dsolr.solr.home=D:/solrcloud/solr\_home2 -DzkHost=localhost:2181,localhost:2182,localhost:2183

### 启动集群

1 . 依次启动zk和tomcat，开始启动zk的时候会有各种报错，没关系，因为zk是2n+1的机制，当你依次启动好了就可以了。

2 . 访问[**http://localhost:8081/solr/**](http://localhost:8081/solr/)

3 . 创建一个solr的collection

<http://localhost:8081/solr/admin/collections?action=CREATE&name=eSearch&numShards=3&replicationFactor=3&maxShardsPerNode=3&collection.configName=myconf>

4 . 删除一个collection

<http://localhost:8081/solr/admin/collections?action=DELETE&name=eSearch>

# Solr 基础

## HttpSolrClient

旧版本的连接服务HttpSolrServer在solr5版本后已经停用，被新的HttpSolrClient取代，大家可以从下面的url中查看相关的API,了解更详细的变动。

HttpSolrClient server=new HttpSolrClient(url);

//设置对应请求的目标主机线程数为1000条

server.setDefaultMaxConnectionsPerHost(1000); server.setMaxTotalConnections(10000);

server.setConnectionTimeout(60000);//设置连接超时时间（单位毫秒） 1000

server.setSoTimeout(60000);//// 设置读数据超时时间(单位毫秒) 1000

server.setFollowRedirects(false);//遵循从定向

server.setAllowCompression(true);//允许压缩

API：<http://lucene.apache.org/solr/5_3_1/solr-solrj/index.html>

## Solr 添加索引

HttpSolrClient server = SolrServer.*getServer();*

SolrInputDocument doc = **new SolrInputDocument();**

doc.addField("id", "123456");

doc.addField("title\_s", "test");

doc.addField("des\_s", "test des");

**try {**

server.add(doc);

server.commit();

} **catch (Exception e) {**

e.printStackTrace();

}

Id：索引中的唯一ID，定义见配置。

title\_s：动态字符串域定义，静态见配置。

## 查询索引

HttpSolrClient server = SolrServer.*getServer();*

SolrQuery sQuery = **new SolrQuery();**

sQuery.setQuery(str); // 默认\*:\* ， ***des\_s：xxx , 具体见查询语法***

//sQuery.set("q", "des\_s:test\*");// 查询nickname是已chm开头的数据

sQuery.setStart(0); // 检索偏移量

sQuery.setRows(5); // 返回行数

QueryResponse queryResponse;

**try {**

queryResponse = server.query(sQuery);

SolrDocumentList list = queryResponse.getResults();

System.***out.println("query result nums: " + list.getNumFound());***

**for (int i = 0; i < list.size(); i++) {**

System.***out.println(list.get(i).getFieldValue("des\_s"));***

}

} **catch (Exception e) {**

e.printStackTrace();

}

## 删除索引

HttpSolrClient server = SolrServer.*getServer();*

server.deleteById(“123456”); // 单个id

server.commit();

HttpSolrClient server = SolrServer.*getServer();*

List<String> ids = **new ArrayList<String>();**

ids.add(“123456”); // id的集合

server.deleteById(ids);

server.commit();

## 常用语法

q - 查询字符串，必须的。

fl - 指定返回那些字段内容，用逗号或空格分隔多个。    
start - 返回第一条记录在完整找到结果中的偏移位置，0开始。

rows - 指定返回结果最多有多少条记录，配合start来实现分页。

sort - 排序，示例：(inStock desc, price asc)表示先 “inStock”降序, 再 “price”升序，默认是相关性降序。

wt - (writer type)指定输出格式，可以有 xml, json, php, phps。    
 fq - (filter query )过虑查询，作用：在q查询符合结果中同时是fq查询符合的，例如：q=mm&fq=date\_time:[20081001 TO 20091031]，找关键字mm，并且date\_time是20081001到20091031之间的。官方文档：<http://wiki.apache.org/solr/CommonQueryParameters>

q.op - 覆盖schema.xml的defaultOperator（有空格时用“AND”还是用“OR”操作逻辑），一般默认指定  “OR”  
df - 默认的查询字段，一般默认指定    
qt – (query type)指定那个类型来处理查询请求，一般不用指定，默认是standard。

indent - 返回的结果是否缩进，默认关闭，用 indent=true|on 开启，一般调试json,php,phps,ruby输出才有必要用这个参数。    
version - 查询语法的版本，建议不使用它，由服务器指定默认值。

## 检索运算符

“:”指定字段查指定值，如返回所有值\*:\*  
“?”表示单个任意字符的通配   
“\*” 表示多个任意字符的通配（不能在检索的项开始使用\*或者?符号）  
“~”表示模糊检索，如检索拼写类似于”roam”的项这样写：roam~将找到形如foam和roams的单词；roam~0.8，检索返回相似度在0.8以上的记录。   
邻近检索，如检索相隔10个单词的“apache”和”“akarta”，“jakarta apache”~10   
“^”控制相关度检索，如检索jakarta apache，同时希望去让“jakarta”的相关度更加好，那么在其后加上”^”符号和增量值，即jakarta^4 apache

布尔操作符AND、||  
布尔操作符OR、&&   
布尔操作符NOT、!、-（排除操作符不能单独与项使用构成查询）   
“+” 存在操作符，要求符号“+”后的项必须在文档相应的域中存在  
 ( ) 用于构成子查询  
 [ ] 包含范围检索，如检索某时间段记录，包含头尾，date:[200707 TO 200710]   
 { }不包含范围检索，如检索某时间段记录，不包含头尾   
 date:{200707 TO 200710}   
 " 转义操作符，特殊字符包括+ - && || ! ( ) { } [ ] ^ ” ~ \* ? : "

## 其它查询条件

查询某个字段非空的记录 比如：fq=FieldName:[‘’ TO \*] 查询FieldName非空的数据。

查询某个字段为空的记录 比如：查询公司名称为空的记录可以采用如下语法实现(似乎目前为止只有此方法可行):

-company:[\* TO \*]

取反实例：fq=!fstate:1

## Multicore 查询

### 创建multicore

1、复制D:\work\solr\_home\test，命名为core。

2、修改D:\work\solr\_home\test\ core.properties

name=test

config=solrconfig.xml

schema=schema.xml

dataDir=data

改成：

name=core1

3、重启tomcat

### 添加索引

向每个core添加索引：

修改**private static String *url="http://localhost:8080/solr/test";***

***每次添加不同core只需要修改test即可，也可以以参数将core传入。***

添加索引的方法跟上节将的方法一样，在此就不再累述了。

### Multicore查询

HttpSolrClient server = SolrServer.*getServer();*

SolrQuery sQuery = **new SolrQuery();**

sQuery.setQuery(str);

sQuery.setStart(0);

sQuery.setRows(5);

*sQuery.set("shards", "localhost:8081/solr/test,localhost:8081/solr/core1,localhost:8081/solr/core2");*

QueryResponse queryResponse;

**try {**

queryResponse = server.query(sQuery);

List<TestBO> indexBOLists = queryResponse.getBeans(TestBO.**class);**

**for (TestBO bo : indexBOLists) {**

System.***out.println(bo.getDes\_s());***

}

} **catch (Exception e) {**

e.printStackTrace();

}

## 管理页面安全设置

仅针对tomcat：

1、修改G:\solr5\apache-tomcat-8.0.28\conf\tomcat-users.xml

在tomcat-users标签内添加：

<role rolename="solr"/>

<user username="admin" password="admin" roles="solr"/>

2、修改G:\solr5\apache-tomcat-8.0.28\webapps\solr\WEB-INF\web.xml

<security-constraint>

<web-resource-collection>

<web-resource-name>RestrictaccesstoSolradmin</web-resource-name>

<url-pattern>/admin/\*</url-pattern>

<http-method>DELETE</http-method>

<http-method>GET</http-method>

<http-method>POST</http-method>

<http-method>PUT</http-method>

</web-resource-collection>

<auth-constraint>

<description>This applies only to the "tomcat" security role</description>

<role-name>solr</role-name>

<role-name>admin</role-name>

</auth-constraint>

<user-data-constraint>

<transport-guarantee>NONE</transport-guarantee>

</user-data-constraint>

</security-constraint>

<login-config>

<auth-method>BASIC</auth-method>

<realm-name>please input user and password.</realm-name>

</login-config>

## SolrCloud

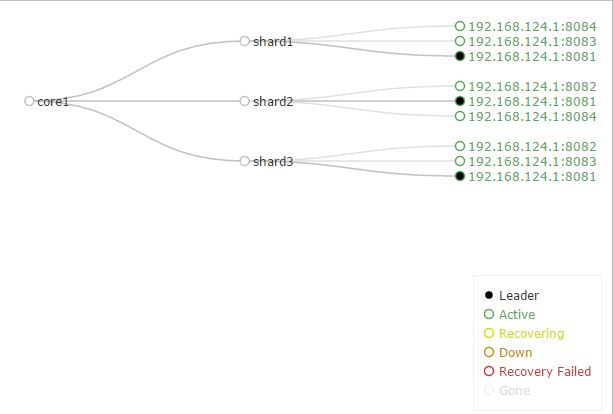
[SolrCloud](http://wiki.apache.org/solr/SolrCloud)是基于Solr和Zookeeper的分布式搜索方案，也是Solr5 的核心组件之一，它的主要思想是使用Zookeeper作为集群的配置信息中心。它有几个特色功能：

1）集中式的配置信息

2）自动容错

3）近实时搜索

4）查询时自动负载均衡 。



基本可以用这幅图来概述，这是一个拥有4个Solr节点的集群，索引分布在三个Shard里面，每个Shard包含三个Solr节点，一个是Leader节点，另外两个是Replica节点，此外集群中有一个负责维护集群状态信息的Overseer节点，它是一个总控制器。集群的所有状态信息都放在Zookeeper集群中统一维护。从图中还可以看到，任何一个节点都可以接收索引更新的请求，然后再将这个请求转发到文档所应该属于的那个Shard的Leader节点，Leader节点更新结束完成，最后将版本号和文档转发给同属于一个Shard的replicas节点。

*具体节点信息我们在后续进入Zookeeper管理节点查看*

### 创建与删除集群索引库

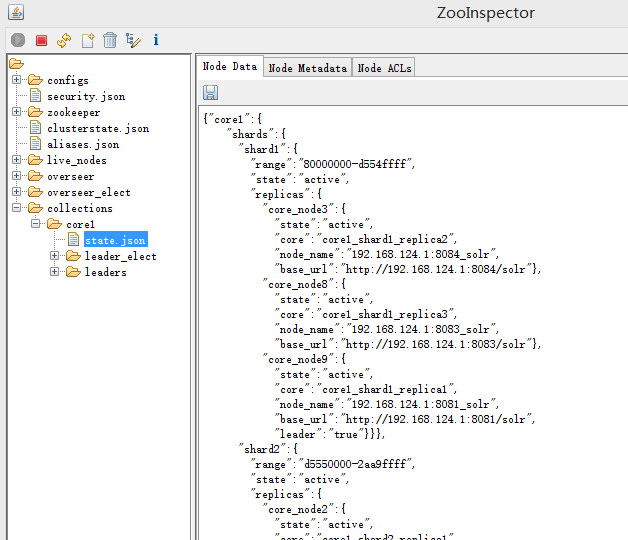
**1 . 创建一个solr的collection**

<http://localhost:8081/solr/admin/collections?action=CREATE&name=eSearch&numShards=3&replicationFactor=3&maxShardsPerNode=3&collection.configName=myconf>

**2 . 删除一个collection**

<http://localhost:8081/solr/admin/collections?action=DELETE&name=eSearch>

使用命令或ZooInspector工具查看集群信息



## Solrj操作SolrCloud

### 获取CloudSolrClient

public static CloudSolrClient getServer() {

final String zkHost = "localhost:2181,localhost:2182,localhost:2183";

final String defaultCollection = "core1";

final int zkClientTimeout = 20000;

final int zkConnectTimeout = 1000;

CloudSolrClient *cloudSolrServer = new CloudSolrClient(zkHost);*

System.*out.println("The Cloud SolrServer Instance has benn created!");*

cloudSolrServer.setDefaultCollection(defaultCollection);

cloudSolrServer.setZkClientTimeout(zkClientTimeout);

cloudSolrServer.setZkConnectTimeout(zkConnectTimeout);

cloudSolrServer.connect();

return cloudSolrServer;

}

Add index:

CloudSolrClient solrServer = *getServer();*

SolrInputDocument doc1 = **new SolrInputDocument();**

doc1.addField("id", "1");

doc1.addField("name", "王海");

SolrInputDocument doc2 = **new SolrInputDocument();**

doc2.addField("id", "2");

doc2.addField("name", "张芳");

SolrInputDocument doc3 = **new SolrInputDocument();**

doc3.addField("id", "3");

doc3.addField("name", "刘俊");

Collection<SolrInputDocument> docs = **new ArrayList<SolrInputDocument>();**

docs.add(doc1);

docs.add(doc2);

docs.add(doc3);

solrServer.add(docs);

solrServer.commit();

*其它删除和查询都与单机一样，仅连接不同。*

Query:

SolrQuery query = **new SolrQuery();**

query.setQuery(String);

**try {**

CloudSolrClient solrServer = *getServer();*

QueryResponse response = solrServer.query(query);

SolrDocumentList docs = response.getResults();

System.***out.println("文档个数：" + docs.getNumFound());***

System.***out.println("查询时间：" + response.getQTime());***

**for (SolrDocument doc : docs) {**

String name = (String) doc.getFieldValue("name");

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

}

} **catch (SolrServerException e) {**

e.printStackTrace();

} **catch (Exception e) {**

e.printStackTrace();

}

Delete Index:

public static void delteIndex() {

try {

CloudSolrClient server = SolrCloudServer.*getServer();*

server.deleteByQuery("\*:\*");

server.commit();

} catch (Exception e) {

e.printStackTrace();

}

}

## Solr 主从复制

SOLR复制模式，是一种在分布式环境下用于同步主从服务器的一种实现方式，因之前提到的基于rsync的SOLR不同方式部署成本过高，取而代之的就是基于HTTP协议的索引文件传输机制，该方式部署简单，只需配置一个文件即可。以下讲解具体操作步骤：   
 步骤分主服务器和从服务器，允许有多个从服务器，即从服务器的配置一样。

### 主服务器修改solrConfig.xml

<requestHandler name="/replication" class="solr.ReplicationHandler">

<lst name="master">

<str name="replicateAfter">commit</str>

<str name="replicateAfter">startup</str>

<str name="confFiles">schema.xml,stopwords.txt</str>

<str name="commitReserveDuration">00:01:00</str>

</lst>

</requestHandler>

说明：   
replicateAfter : SOLR会自行在以下操作行为发生后执行复制： 'commit', 'startup' 'optimize',这里我们选择commit , 即SOLR每一次接受到commit请求后，会执行复制策略。

confFiles : 待分发的配置文件，solr 也会将主服务器上的字段配置文件：schema.xml和stopwords.txt，固排文件： elevate.xml同步到辅服务器上。

commitReserveDuration: 每次commit之后，保留增量索引的周期时间，这里设置为1分钟。

### 从服务器修改solrConfig.xml

<requestHandler name="/replication" class="solr.ReplicationHandler">

<lst name="slave">

<str name="masterUrl">http://localhost:8081/solr/core1/replication</str>

<str name="pollInterval">00:01:00</str>

<str name="compression">internal</str>

<str name="httpConnTimeout">2000</str>

<str name="httpReadTimeout">6000</str>

</lst>

</requestHandler>

**说明：**   
masterUrl : 主服务器同步URL地址

pollInterval:从服务器同步间隔，即每隔多长时间同步一次主服务器

httpConnTimeout:设置连接超时（单位：毫秒）

httpReadTimeout:如果设置同步索引文件过大，则应适当提高此值。（单位：毫秒）

httpBasicAuthUser:验证用户名，需要和主服务器一致

httpBasicAuthPassword:验证密码，需和主服务器一致

compression:external or internal 使用SOLR自己的压缩算法或应用容器的

# Solr进阶

## Facet

Facet 是 solr 的高级搜索功能之一 ，可以给用户提供更友好的搜索体验，在搜索关键字的同时 , 能够按照 Facet 的字段进行分组并统计。

比如你上淘宝，输入“笔记本”进行搜索，就会出现品牌分类，价格范围等分类，这个就叫facet了。这个例子也许不是那么准确的描述facet，不过基本上就是这个意思。对输入关键字后搜索出来的结果再进行分类。

solr有几种facet:   
 普通facet，比如从厂商品牌的维度建立fact 。  
 查询facet，比如根据价格查询时，将根据价格，设置多个区间，比如0-10, 10-20, 20-30等。  
 日期facet， 也是一种特殊的范围查询，比如按照月份进行facet。



### Facet查询

进行Facet查询需要在请求参数中加入“facet=on”或“facet=true”只有这样Facet组件才起作用。

1. Field Facet

Facet字段通过在请求中加入“facet.field”参数加以声明，如果需要对多个字段进行Facet查询,那么将该参数声明多次.比如

http://localhost:8081/solr/select?q=联想&facet=on&facet.field=cpu&facet.field=videoCard

各个Facet字段互不影响,且可以针对每个Facet字段设置查询参数.以下介绍的参数既可以应用于所有的Facet字段,也可以应用于每个单独的Facet字段.应用于单独的字段时通过 f.字段名.参数名=参数值，这种方式调用。比如facet.prefix参数应用于cpu字段,可以采用如下形式

f.cpu.facet.prefix=Intel

### 实例分析

sQuery.setFacet(true);//设置facet=on

sQuery.addFacetField(new String[] { "subMajor\_s","brand\_s" });//设置需要facet的字段

sQuery.setFacetLimit(1000);//-1 // 限制facet返回的数量

sQuery.setFacetMissing(false);// 不统计null的值

sQuery.setFacetMinCount(1);// 设置返回的数据中每个分组的数据最小值，比如设置为1，则统计数量最小为1，不然不显示

sQuery.addFacetQuery("price\_i:[1000 TO 10000]");

sQuery.setQuery(para);

QueryResponse queryResponse;

queryResponse = solrServer.query(sQuery);

List<FacetField> facets = queryResponse.getFacetFields();// 返回的facet列表

<http://wiki.apache.org/solr/SimpleFacetParameters>

for (FacetField facet :facets) {

System.*out.println(facet.getName());*

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

List<Count>counts = facet.getValues();

for (Count count : counts){

System.*out.println(count.getName()+":"+ count.getCount());*

}

System.*out.println();*

}

*注：建立索引时加上价格区间的字段，就可以实现检索时显示动态的价 格筛选条件。*

<http://wiki.apache.org/solr/SimpleFacetParameters>

## Facet.pivot

***Facet.pivot*** 就是按照多个维度进行分组查询，是Facet的加强，在实际运用中经常用到，一个典型的例子就是商品目录树。



HttpSolrClient solrServer = SolrServer.*getServer();*

SolrQuery sQuery = **new SolrQuery();**

sQuery.setFacet(**true);**

sQuery.add("facet.pivot", "major\_s, subMajor\_s");//两维度来分组查询

sQuery.setFacetLimit(1000);

sQuery.setQuery(para);

QueryResponse response = solrServer.query(sQuery,SolrRequest.METHOD.***POST);***

NamedList<List<PivotField>> namedList = response.getFacetPivot();

NamedList解释：

NamedList，一个有序的name/value容器，NamedList不像Map，他具有以下特点：

1、名字可以重复

      2、NamedList中的element保持这有序状态

      3、可以下标的形式访问Elements

      4、name和value都可以为null

List<PivotField> pivotList = **null;**

**for(int i=0;i<namedList.size();i++){ //此处的多次循环可以用递归**

pivotList = namedList.getVal(i);

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

**for(PivotField pivot:pivotList){**

System.***out.println("一级 : "+pivot.getValue() + " " + pivot.getCount());***

List<PivotField> fieldList = pivot.getPivot();

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

**for(PivotField field:fieldList){**

System.***out.println("二级 : "+field.getValue() + " " + field.getCount());***

List<PivotField> fieldList1 = field.getPivot();

**if(fieldList1 != null){**

**for(PivotField field1:fieldList1){**

System.***out.println("三级 : "+field1.getValue() + " " + field1.getCount());***

}}}}}}}

## GROUP

facet的查询结果主要是分组信息：有什么分组，每个分组包括多少记录；但是分组中有哪些数据是不可知道的，只有进一步搜索。  
 group则类似于关系数据库的group by，可以用于一个或者几个字段去重、显示一个group的前几条记录等。

再细说点就是如果你想查询归查询聚类归聚类，那么使用facet，如果想使用类似采集的效果，每个group分组采集多少个，那么使用group查询。

### 示例

HttpSolrClient solrServer = SolrServer.*getServer();*

SolrQuery query = **new SolrQuery("brand\_s:海尔");//"subMajor\_s:空调" "\*:\*"**

// 设置通过Group查询为true，表示查询时使用Group机制

query.setParam(GroupParams.***GROUP, true);***

// 设置Group查询针对的域

query.setParam(GroupParams.***GROUP\_FIELD, "brand\_s");***

// 设置每个组最多返回的记录数，可用于数据采集;若只需要数量，可以设置为0

query.setParam(GroupParams.***GROUP\_LIMIT, "5");***

// 设置返回的行数

query.setRows(10);

// 获取查询响应

QueryResponse response = solrServer.query(query);

**if (response != null) {**// 获取group查询响应

GroupResponse groupResponse = response.getGroupResponse();

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

List<GroupCommand> groupList = groupResponse.getValues();

**for (GroupCommand groupCommand : groupList) {**

List<Group> groups = groupCommand.getValues();

**for (Group group : groups) {**

System.***out.println(group.getResult().get(0).getFirstValue("subMajor\_s")***

+" "+ group.getResult().get(1).getFirstValue("subMajor\_s"));

System.***out.println("group查询..." + group.getGroupValue() + " 数量为：" + group.getResult().getNumFound() + " " + group.getResult().get(0).getFieldValue("model\_s"));***

}

}

}

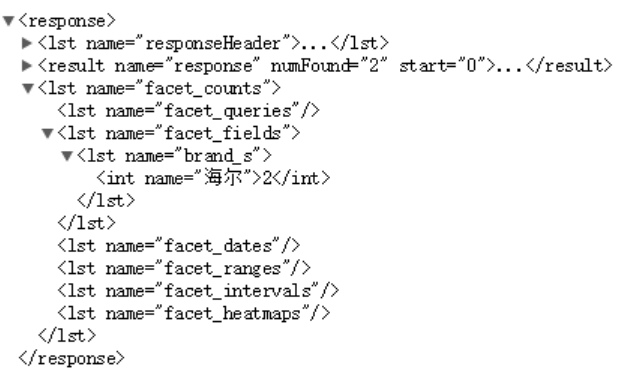
}

}**catch(Exception e){e.printStackTrace();}**

<http://localhost:8081/solr/test/select?q=brand_s:xx&group=true&group.field=brand_s&group.limit=5&rows=10>



<http://localhost:8081/solr/test/select?q=brand_s:xx&facet=true&facet.field=brand_s&facet.limit=10&facet.missing=false&facet.mincount=1>



## 高亮显示

高亮显示在搜索中使用的比较多，比较常用的有三种使用方式,如果要对某field做高亮显示，必须对该field设置stored=true 。   
      第一种是普通的高亮显示Highlighter，根据查询的docIdSet，获取Documents，并获取当前document的需要高亮的field的value，根据query的term和该field的value做匹配算法。   
      第二种是，快速高亮显示FastVectorHighlighter，效率比普通的高亮显示要高；需要定义termvector(占用空间和IO)，包括position和offset，根据query term的termvector到field value中做快速的定位标记，进而实现快速的高亮显示。  
      第三种是在solr外部做高亮显示，由于高亮显示需要对field设置为store=true，所有对于单节点数据量比较大并且该字段比较大的话，会消耗大量的IO操作，那么可以把该字段存储在另外的地方，比如Hbase，在外部做高亮显示的匹配。

### 参数说明

hl.fl: 用空格或逗号隔开的字段列表。要启用某个字段的highlight功能，就得保证该字段在schema中是stored。如果该参数未被给出，那么就会高亮默认字段 standard handler会用df参数，dismax字段用qf参数。你可以使用星号去方便的高亮所有字段。如果你使用了通配符，那么要考虑启用hl.requiredFieldMatch选项。

hl.requireFieldMatch: 如果置为true，除非用hl.fl指定了该字段，查询结果才会被高亮。它的默认值是false。

hl.usePhraseHighlighter: 如果一个查询中含有短语（引号框起来的）那么会保证一定要完全匹配短语的才会被高亮。

hl.highlightMultiTerm :如果使用通配符和模糊搜索，那么会确保与通配符匹配的term会高亮。默认为false，同时hl.usePhraseHighlighter要为true。

hl.fragsize: 返回的最大字符数。默认是100.如果为0，那么该字段不会被fragmented且整个字段的值会被返回。

常用的就这几个：参数详细地址<http://wiki.apache.org/solr/HighlightingParameters/>

SolrQuery query = **new SolrQuery();**

query.setQuery(queryString);

query.setHighlight(**true);//开启高亮功能**

query.addHighlightField("title");//高亮字段

query.setHighlightSimplePre("<font color='red'>");//渲染标签

query.setHighlightSimplePost("</font>");//渲染标签

query.setStart((pageNum-1)\*pageSize);

query.setRows(pageSize);

QueryResponse response = **null;**

**try {**

response = server.query(query);

} **catch (Exception e) {**

e.printStackTrace();

**return null;**

}

SolrDocumentList lists = response.getResults();//查询结果集

List<ProductBO> items = **new ArrayList<ProductBO>();//对象结果集**

String tmpId = "";

Map<String,Map<String,List<String>>>highlightMap=response.getHighlighting();

**for (SolrDocument solrDocument : lists) {**

ProductBO at = **new ProductBO();**

tmpId = solrDocument.getFieldValue("id").toString();

at.setId(tmpId);

at.setTitle(solrDocument.getFieldValue("title").toString());

List<String> titleList = highlightMap.get(tmpId).get("title");

**if(titleList != null && titleList.size() > 0){**

at.setTitle(titleList.get(0));//获取并设置高亮的字段title

}**else{**

at.setTitle(solrDocument.getFieldValue("title").toString());

}

System.***out.println(at.getSubMajor\_s() + " | " + at.getTitle());***

}

## Solr相似匹配

在网页搜索或电商产品搜索结果页面，很多时候会看到一个相似文档、相似产品或找相似的链接。Solr 使用 MoreLikeThisComponent（MLT）和 MoreLikeThisHandler 实现了一样的功能。如上所述，MLT 是与标准 SolrRequestHandler 集成在一起的；MoreLikeThisHandler 与 MLT 结合在一起，并添加了一些其他选项，但它要求发布一个单一的请求。我将着重讲述 MLT，因为使用它的可能性更大一些。

　　MLT 要求字段被储存或使用检索词向量，检索词向量以一种以文档为中心的方式储存信息。MLT 通过文档的内容来计算文档中关键词语，然后使用原始查询词语和这些新词语创建一个新的查询。提交新查询就会返回其他查询结果。所有这些都可以用检索词向量来完成：只需将 termVectors="true" 添加到 schema.xml 中的 <field> 声明。

### 两种实现

第一种：SearchHandler中的MoreLikeThisComponent，MoreLikeThis以组件的身份出现，适于简单应用。  
第二种：MoreLikeThisHandler，MoreLikeThis作为一个单独的Handler来处理，可以应用过滤等较复杂操作。

### 查询参数

**id**，文档主键，或使用其他唯一键；  
**fl**，需要返回的字段  
**mtl.fl**，根据哪些字段判断相似度  
**mlt.mindf**，最小文档频率，所在文档的个数小于这个值的词将不用于相似判断  
**mlt.mintf**，最小分词频率，在单个文档中出现频率小于这个值的词将不用于相似判断  
**mlt.count**，返回相似文章个数

*需要注意的是 mlt.fl 中的 field 的* termVectors*=true 才有效果。*

<field  name=“title" type="text\_smartcn" indexed="false" stored="true"  multiValued="false" termVectors="true"/>

### 实现一：

HttpSolrClient solrServer = SolrServer.*getServer();*

SolrQuery query = **new SolrQuery();**

List<ProductBO> articles = **new ArrayList<ProductBO>();**

**try {**

query.setQuery("id:" + id)

.setParam("fl", "id,title,brand\_s")

.setParam("mlt", "true")

.setParam("mlt.fl", "title")

.setParam("mlt.mindf", "1")

.setParam("mlt.mintf", "1")

.setParam("mlt.count", String.*valueOf(count));*

QueryResponse response = solrServer.query(query);

**if (response == null) return articles;**

SimpleOrderedMap<SolrDocumentList> mltResults = (SimpleOrderedMap<SolrDocumentList>) response.getResponse().get("moreLikeThis");

**for (int i = 0; i < mltResults.size(); i++) {**

SolrDocumentList items = mltResults.getVal(i);

**for (SolrDocument doc : items) {**

String idStr = doc.getFieldValue("id").toString();

**if (StringUtils.*equals(idStr, id)) continue;// 排除本身***

ProductBO article = **new ProductBO();**

article.setId(idStr);

article.setTitle(doc.getFieldValue("title").toString());

article.setBrand\_s(doc.getFieldValue("brand\_s").toString());

articles.add(article);

}

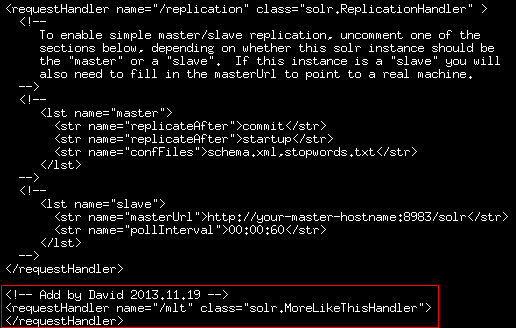
}

### 实现二

配置MoreLikeThisHandler，修改solrconfig.xml

在最后一个requestHandler下新增/mlt

<requestHandler name="/mlt" class="solr.MoreLikeThisHandler">   
</requestHandler>



SolrQuery query = new SolrQuery();  
        // 指定配置的RequestHandler，默认使用/select  
        query.setRequestHandler("/mlt");  
        List<Article> articles = new ArrayList<Article>();  
        try {  
            query.setQuery("id:" + id)  
                .setParam("fl", "id,title,score")  
                .setParam("mlt", "true")  
                .setParam("mlt.fl", "title")  
                .setParam("mlt.mindf", "1")  
                .setParam("mlt.mintf", "1");  
            query.addFilterQuery("status:" + Article.STATUS\_PUBLISHED);  
            query.setRows(count);// mlt.count无效，需要此方法控制返回条数  
            QueryResponse response =  solrServer.query(query);  
            if (response == null) return articles;

### 参考资料

<http://wiki.apache.org/solr/MoreLikeThis>

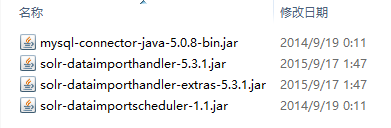
<http://www.boyunjian.com/javadoc/org.apache.servicemix.bundles/org.apache.servicemix.bundles.solr-solrj/4.7.0_1/_/org/apache/solr/client/solrj/util/package-summary.html>

## DataImport

Solr 除利用solrj建立索引外，还可以由列式数据库hbase 触发器添加索引、自动全量或增量索引关系数据库数据等。

Dataimport 可以配置从任何关系数据库导入索引，这里我们以mysql为例进行配置讲解。

主要涉及jar包如下：( 拷贝到D:\System\apache-tomcat-8.0.28\webapps\solr\WEB-INF\lib下 )



### web.xml 配置

修改：D:\System\apache-tomcat-8.0.28\webapps\solr\WEB-INF\web.xml 添加调度监听：

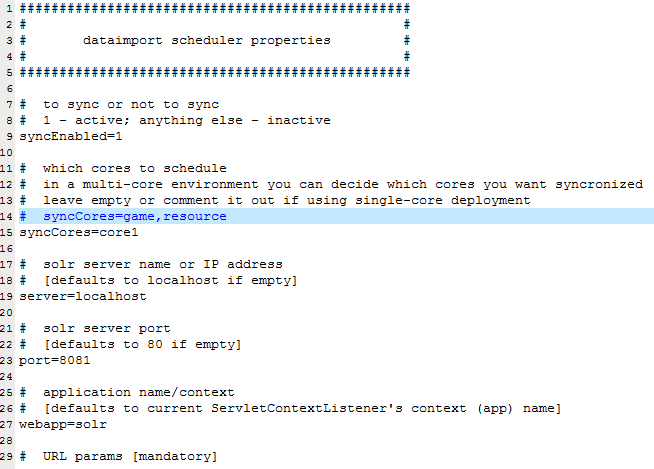
<listener>

<listener-class>org.apache.solr.handler.dataimport.scheduler.ApplicationListener</listener-class>

</listener>

### dataimport.properties配置一

存放路径：D:\work\solr\_home\conf 若不存在，需要新建。



## dataimport.properties配置二



### 添加data-config.xml配置

在目录D:\work\solr\_home\test2\conf下添加data-config.xml配置，内容如下：

<dataConfig>

<dataSource type="JdbcDataSource"

driver="com.mysql.jdbc.Driver"

url="jdbc:mysql://192.168.1.106:3306/test\_db"

user="root"

password="etimes"/>

<document>

<entity name="id"

query="select id,word from t\_word">

</entity>

</document>

</dataConfig>

### 修改schema.xml、solrconfig.xml

修改D:\work\solr\_home\test2\conf\ schema.xml 添加：

<field name="word" type="string" indexed="true" stored="true" />

修改D:\work\solr\_home\test2\conf\ solrconfig.xml 添加：

<requestHandler name="/dataimport" class="org.apache.solr.handler.dataimport.DataImportHandler">

<lst name="defaults">

<str name="config">data-config.xml</str>

</lst>

</requestHandler>

# IK分词

IKAnalyzer是一个开源的，基于java语言开发的轻量级的中文分词工具包。

a. 算法采用“正向迭代最细粒度切分算法”，支持细粒度和最大词长两种分词方式，速度最大支持80W字/秒（1600KB/秒）。

b. 支持多子处理器分析模式：中文、数字、字母，并兼容日文、韩文。

c. 较小的内存占用，优化词库占有空间，用户可自定义扩展词库。

d. 扩展lucene的扩展实现，采用歧义分析算法优化查询关键字的搜索排列组合，提高lucene检索命中率。

## 关键类：

**org.wltea.analyzer.lucene.IKAnalyzer**

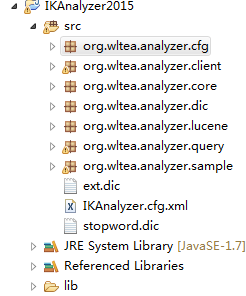
IK分词主类，基于Lucene的Analyzer接口实现。

**org.wltea.analyzer.lucene.IKQueryParser**

IK分词器提供的Query解析、构造工具类，其中parseMultiField 函数(所有的重载函数)为关键函数。

**org.wltea.analyzer.IKSegmentation**

IK分词器的核心类，真正分词的实现类。





## 算法理解

根据作者官方说法IK分词器采用“正向迭代最细粒度切分算法”，分析它的源代码，可以看到分词工具类IKQueryParser起至关重要的作用，它对搜索关键词采用从最大词到最小词层层迭代检索方式切分，比如搜索词：“中华人民共和国成立了”，首先到词库中检索该搜索词中最大分割词，即分割为：“中华人民共和国”和“成立了”，然后对“中华人民共和国”切分为“中华人民”和“人民共和国”，以此类推。最后，“中华人民共和国成立了”切分为：“中华人民 | 中华 | 华人 | 人民 | 人民共和国 | 共和国 | 共和 | 成立 | 立了”，当然，该切分方式为默认的细粒度切分，若按最大词长切分，结果为：“中华人民共和国 | 成立 | 立了”。

## 其它分词器

目前的分词器有：mmseg4j、paoding、ik-analyzer、imdict、Ansj；目前任在持续更新的有mmseg4j、ik-analyzer和Ansj，这三类分词器都有java的实现，ansj是一个ictclas的java实现，ictclas是中科院计算技术研究所的一个中文分词器，目前用的比较广泛的还是ik-analyzer分词器和mmseg4j 分词器，我们对比下：

mmseg4j ：自带sogou词库，支持名为 wordsxxx.dic， utf8文本格式的用户自定义词库，一行一词。不支持自动检测。  
ik ：支持api级的用户词库加载，和配置级的词库文件指定，无 BOM 的 UTF-8 编码，rn 分割。不支持自动检测。

mmseg4j ： 在complex基础上实现了最多分词(max-word)，但是还不成熟，还有很多需要改进的地方。  
ik ：针对Lucene全文检索优化的查询分析器IKQueryParser

## Demo

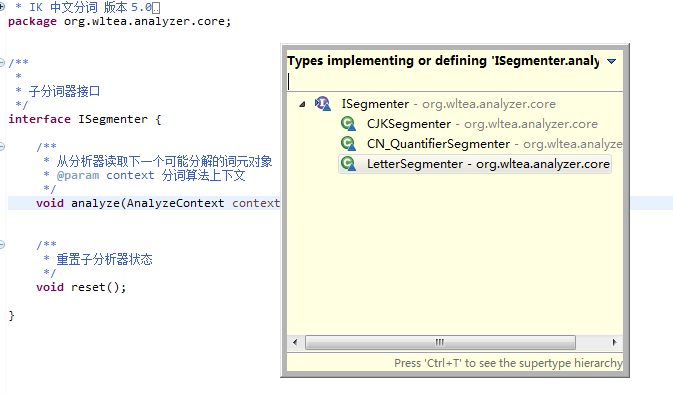
1、TokenStream的作用是从给入的文本中不断解析出Token，运行以上demo，可以看出，ik的大体流程，是先加载词库（我没有输出main2012.dic,其实它是先加载这个内置的词库，然后再来加载我们加入的扩展词库，最后再加载停用词词库）,将这些词都放在字典树中。

2、IK分词器主类（IKSegmenter.java）下的next()方法是用来读取待处理的文本信息遍历分词器，进行分词处理，这里是最核心的流程之一，将待匹配文本生成分词候选集，总共有三种分词器CJKSegmenter（中文分词器）、CN\_QuantifierSegment（数量词分词器）、LetterSegment（字母数字分词器），每种分词器的分词方法是独立的，各自生成自己的分词结果，放到分词候选集里。

3、生成分词候选集之后，进行歧义处理，歧义处理方法区分智能和非智能，也就是在初始化IKSegment时传递的第二个参数IKSegmenter(Readerinput, boolean useSmart)。

## IK 三大分词器

IK里的分词器主要是三个分词器：CJKSegmenter（中文分词），CN\_QuantifierSegmenter（数量词分词），LetterSegmenter（字母分词）。这三个分词器都继承了 ISegmenter接口，思路相差不大，其中采用的结构也比较容易理解，采用字典树（CJK使用）或其他简单数据结构 （CN\_QuantifierSegmenter和LetterSegmenter）匹配文本中的当前字符，将匹配到的字符加入到分词候选集



### 歧义处理（智能分词）

分词的歧义处理是IK分词的一个重要的核心模块，主要使用组合遍历的方式进行处理。从子分词器中取出不相交的分词集合，例如分词结果为abcd（abcd 代表词），abcd是按其在文本中出现的位置排序的，从前到后。假如a与b相交，b与c相交，c与d不相交，则将分词结果切成abc和d两个块分别处理。

当在分词的时候使用的是智能分词，那么便从相交的块中选出最优的结果，这个由IK分词歧义裁决器（ IKArbitrator.java）judge方法来进行处理。

## IK与solr集成

### copy

ik-analyzer-solr5-5.x.jar to ..\apache-tomcat-8.0.28\webapps\solr\WEB-INF\lib

### 配置schemla.xml

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

<analyzer type="index">

<tokenizer class="org.wltea.analyzer.lucene.IKTokenizerFactory" useSmart="false" />

</analyzer>

<analyzer type="query">

<tokenizer class="org.wltea.analyzer.lucene.IKTokenizerFactory" useSmart="true" />

</analyzer>

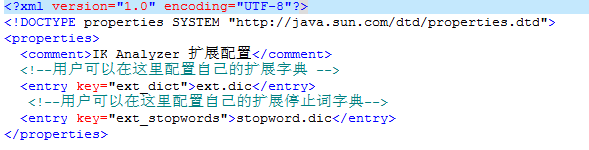
</fieldType>

*注：IKTokenizerFactory在原jar包中是不存在的，需要自己添加。*

### 添加扩展词库

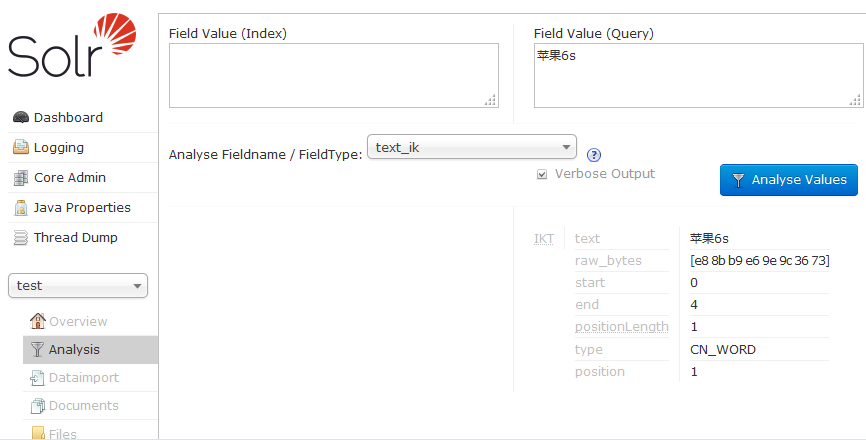
在..\apache-tomcat-8.0.28\webapps\solr\WEB-INF\classes添加ext.dic、IKAnalyzer.cfg.xml、stopword.dic文件，也可以添加多个扩展词库文件，

IKAnalyzer.cfg.xml内容如下：



*注：扩展词库文件可以添加多个，每个之间用分号隔开；扩展词库文件的保存utf-8无BOM格式。*

### 控制台测试



<https://github.com/EugenePig/ik-analyzer-solr5>

## 如何动态加载词库

在实际运用中经常会碰到如下情况：

1、修改了扩展词库需要重启solr服务

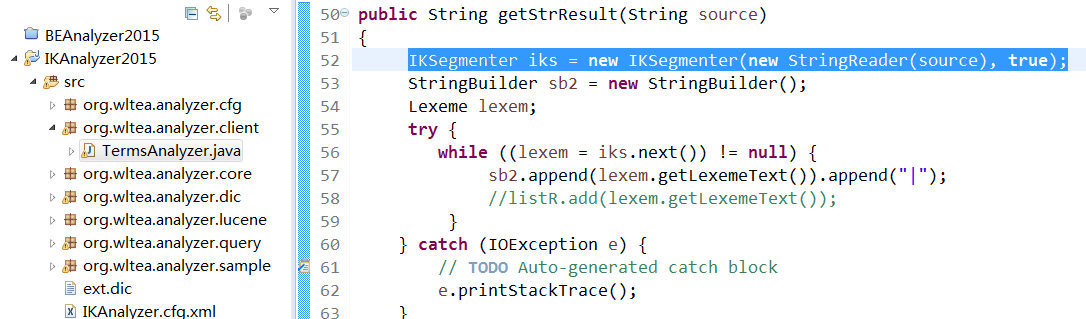
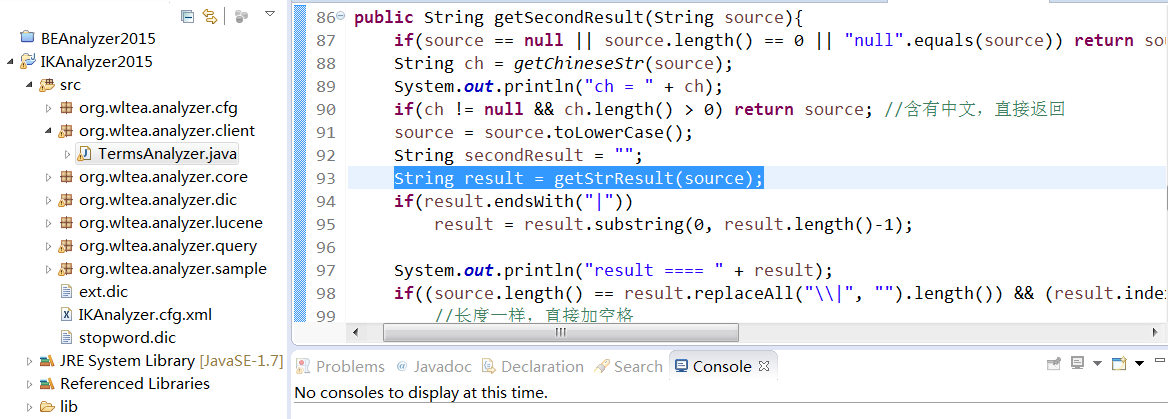
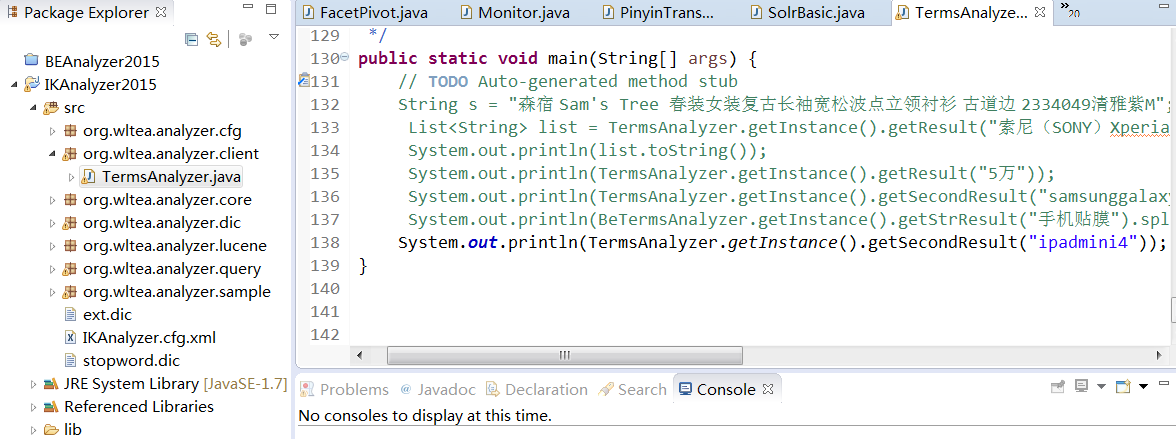
2、需要从关系数据库中加载词到词库

3、在程序中根据不同类型动态调用不同的词库

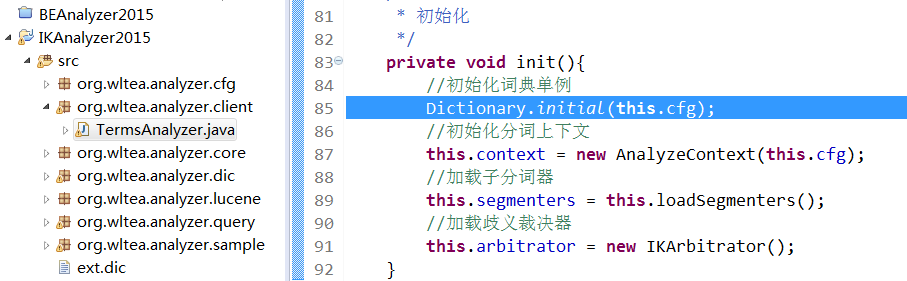
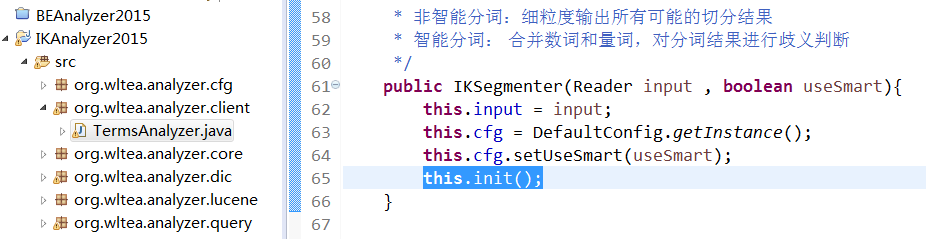
以上要求仅凭IK原始分词包是无法完成的，我们必须修改源码来完成相关的功能，这些都根据项目具体需求进行修改，不过必须在清楚源码结构和分词实现的主要流程条件下方可完成，在此，建议大家修改源码时尽量以添加接口或方法的形式修改，不要大面积修改已有的源码结构，下面我会以第一种情况讲解，第二、三种情况类似。

### Ik词库加载跟踪

我们可以根据客户端调用接口开始跟踪，找到初始化扩展词库的位置：



根据代码反推，找到IK分词器主类IKSegmenter，在主类下可以找到词库初始化的地方：



### 词典初始化

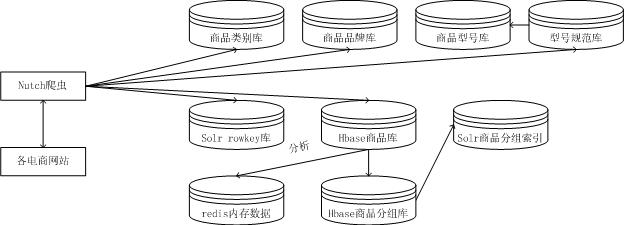


在词典初始化代码里可以加入线程，实时加载远程、本地的文本或关系数据库的词典

# 比比看

## 架构设计

比比看项目采用了Hadoop大数据生态环境，主要用到Nutch爬虫框架、Solr搜索组件、Hdfs文件系统、Hbase列式数据存储、Mapreduce离线数据分析、Redis内存数据库和Kafka Storm实时数据分析等技术。总体架构如下图：



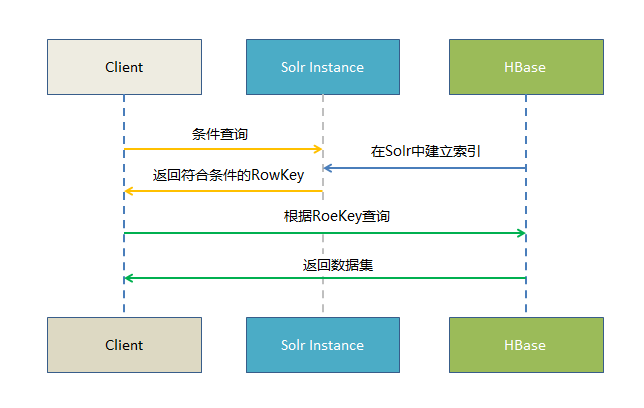
## 商品分组

商品分组即根据商品的类别、品牌、型号进行商品的组合，用户在进行商品搜索时，呈现给用户的将是一组可以比较的数据，检索流程如下图：



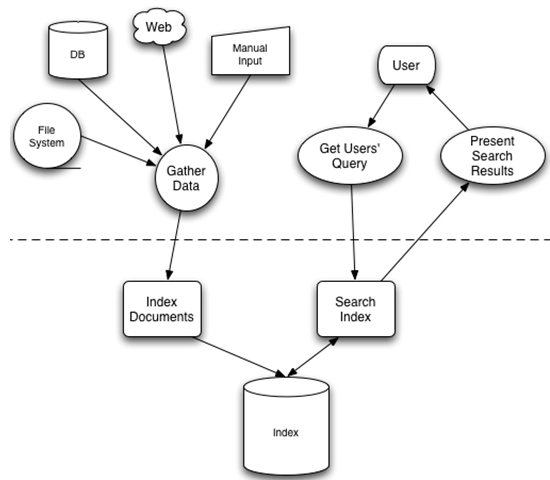
## 详情查询

商品详情查询时，由于分组信息中包含多条同类商品数据，是没有存储在索引库中，所以需要到原始库中查询，如下图：



## 索引设计基本原则

Solr原本是做全文检索的，但在许多场景中都可以利用Solr的实时检索功能完成信息检索需求，在本项目中就利用Solr和Hbase的结合完成分组商品的列表检索和详情检索，在分组索引创建中我们尽量将要检索的信息放到同一文本域中，减少多字段的OR查询，提高检索效率，对单条文本过大的数据我们采取关键字索引，数据存放在Hbase中，以减少Solr存储空间的消耗。



## 索引字段解析

price：商品价格，也是本组商品的最低价格；

maintitle：商品标题，也是本组第一条商品的标题；

brand：商品原始品牌；

model：商品原始型号；

domain\_s：电商域名；

subtype\_s：商品类别；

typenum\_s：类别编号；

quote\_i：商品报价数；

ontitle：分组商品所有标题，以;;分割；

url\_s：商品地址；

productSales\_i：商品销量；

model\_s：分组商品型号；

totalurl：分组商品所有地址，以;;分割；

sourceImgUrl\_s：商品列表页面图片；

thisFilter\_s：商品筛选条件；

newBrand\_s：商品规范后品牌；

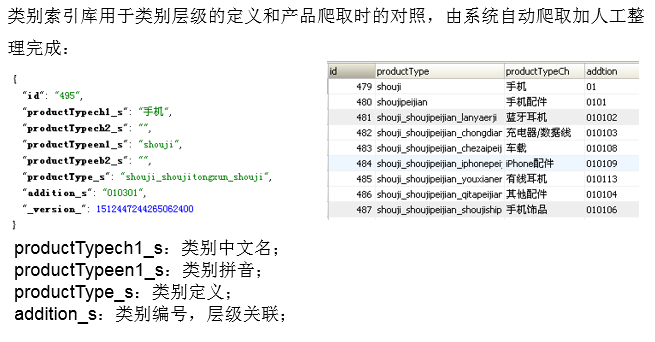
newModel\_s：商品规范后型号；

detailBigImgUrl\_s：商品详细页面图片；

allDomains\_s：本组所有电商域名；

brandtypeweight\_f：品牌所处权重；

### 类别索引库

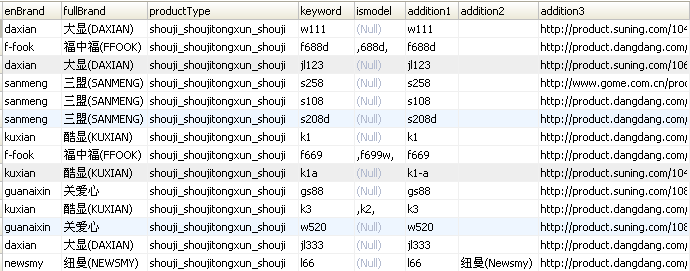


### 品牌索引库



### 型号索引库

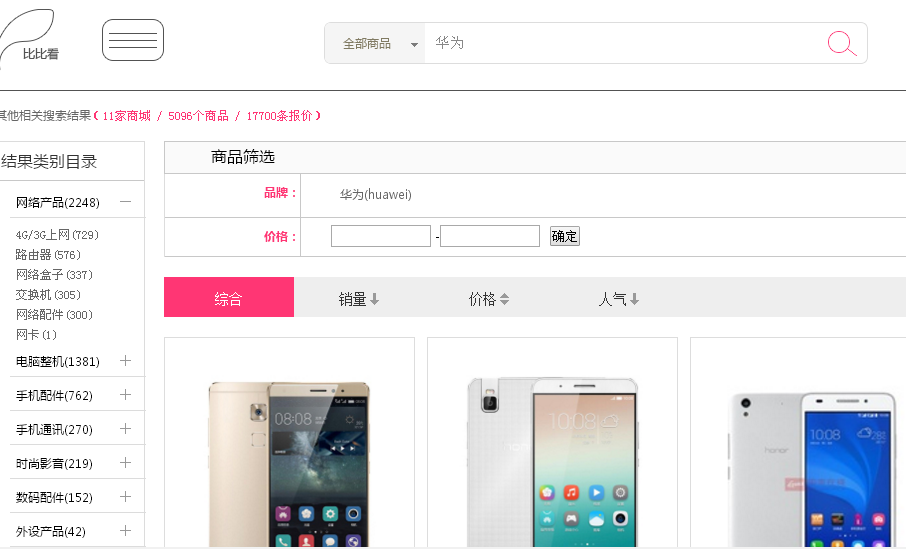
对于型号库，我们暂时未索引起来，型号的统一也直接决定了分组的准确性，所以型号库在系统爬取后需要人工细致的处理，表结构如下：



## 类别目录

类别目录在几乎所有电商网站都存在，主要方便用户快速选择查找，起到导航作用，这里主要讲解Facet和FacetPivot两种实现方式，利用Facet需要结合计算

完成；FacetPivot则可以直接返回想要的目录结构。



本项目的实现方式，缺陷是需要计算才能得出结果，核心实现如下：

sQuery.setFacet(**true);**

sQuery.addFacetField(**new String[] { "subtype\_s" });**

QueryResponse queryResponse;

**try {**

queryResponse = server.query(sQuery);

List<FacetField> facets = queryResponse.getFacetFields();// 返回的facet列表

**for (FacetField facet : facets) {**

List<Count> counts = facet.getValues();

String facetField = counts.get(0).getFacetField().toString().substring(0,

counts.get(0).getFacetField().toString().indexOf(":"));

**for (Count count : counts) {**

**if ("subtype\_s".equals(facetField)) {**

**if (StringUtils.*isNotBlank(count.getName()) && count.getCount() > 0) {***

**if (StringUtils.*isNotBlank(count.getName()))***

map.put(count.getName(), count.getCount());

}

}

}

}

} **catch (Exception e) {**

e.printStackTrace();

}

Set<String> set = map.keySet();

Iterator<String> it = set.iterator();

**while (it.hasNext()) {**

String key = it.next();

**long value = map.get(key);**

String ptype = key.substring(0, key.lastIndexOf("\_"));

**if (resultMap.get(ptype) == null) {**

List<String> list = **new ArrayList<String>();**

list.add(key + ":" + value);

resultMap.put(ptype, list);

} **else {**

List<String> list = resultMap.get(ptype);

list.add(key + ":" + value);

resultMap.remove(ptype);

resultMap.put(ptype, list);

}

}

Set<String> pset = resultMap.keySet();

Iterator<String> pit = pset.iterator();

**while (pit.hasNext()) {**

String key = pit.next();

List<String> list = resultMap.get(key);

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

**for(String s:list)**

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

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

}

FacetPivot方式是比较好的实现方式，这个必须在索引设计时将各级类别加到索引库中，比比看项目由于索引字段较多，在设计时考虑不够周到，所以未使用此方法。

## 设计方案简介

筛选条件会根据不同的查询条件，列出不同的筛选条件，所查询内容必须属于某个类别，如在查询“羽绒服”时，会动态列出如下条件：



### 品牌筛选

以上条件是如何产生的呢？下面我们一一讲解，首先看下商品筛选中的品牌，这个是用facet分组查询实现的，在分组查询结果集中进行了品牌的排序，核心代码如下：

sQuery.setFacet(**true);**

sQuery.addFacetField(**new String[] { "price", "brand", "thisFilter\_s" });**

sQuery.setFacetLimit(SearchConstant.***FACET\_LIMIT);***

QueryResponse queryResponse;

**try {**

queryResponse = server.query(sQuery);

List<FacetField> facets = queryResponse.getFacetFields();// 返回的facet列表

**for (FacetField facet : facets) {**

List<Count> counts = facet.getValues();

String facetField = counts.get(0).getFacetField().toString().substring(0,

counts.get(0).getFacetField().toString().indexOf(":"));

**for (Count count : counts) {**

**if (“brand”.equals(facetField)** && StringUtils.isBlank(brand)) {

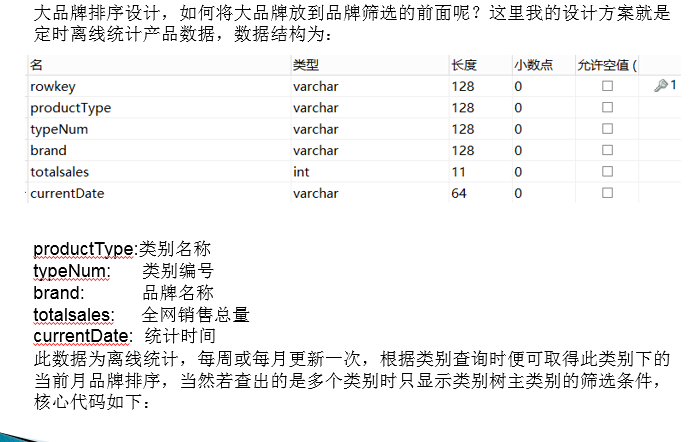
**if (StringUtils.isNotBlank(count.getName())**&& count.getCount() > 0) {

**if (count.getName().contains("其他"))**

**continue;**

abrandMap.put(count.getName().toLowerCase(),count.getName());

} }



TreeMap<String, Integer> big = MysqlDao.*getTopTwentyBrandByProductType(major);*

SortMapByValue bvc = **new SortMapByValue(big);**

List<String> newList = **new ArrayList<String>(big.keySet());**

Collections.*sort(newList, bvc);*

**for (String key : newList) {**

**if (StringUtils.isNotBlank(abrandMap.get(key))) {**

rbrandList.add(key);

abrandMap.remove(key);

}

}

**int i = 0;**

Set<String> seta = abrandMap.keySet();

Iterator<String> ita = seta.iterator();

**while (ita.hasNext()) {**

++i;

**if (i > 990)**

**break;**

String key = ita.next();

rbrandList.add(key);

}

### 过滤条件的筛选

要动态生成过滤条件，那么这个条件就需要索引到索引库中，跟数据的查询变化而变化，那么这个过滤条件是怎么产生的呢？这里我定义了一张条件过滤表，用于原始产品数据分组时分析出包含的过滤关键词，在索引分组数据时同时加入索引库，表结构如下：



以上是我们添加到索引库的筛选数据，那筛选条件怎么产生的呢？先通过facet获取thisFilter\_s的分组统计结果，再根据数据库的顺序动态组合成筛选条件，核心代码如下：

Map<String, String> mapnew = **new HashMap<String, String>();**

Set<String> set = mapFilter.keySet();

Iterator<String> it = set.iterator();

**while (it.hasNext()) {**

String key = it.next();

**if (StringUtils.isBlank(key)) continue;**

String[] s = key.split(":");

**if (s.length < 2) continue;**

**if (StringUtils.isBlank(s[0]) || StringUtils.isBlank(s[1])) continue;**

**if (keys.indexOf("|" + s[0] + "|") >= 0) continue;**

**if (StringUtils.isNotBlank(mapnew.get(s[0]))) {**

String value = mapnew.get(s[0]);

String usevalue = "|" + value;

String[] ss = s[1].split("\\|");

**for (String item : ss) {**

**if (StringUtils.isBlank(item)) continue;**

**if (usevalue.indexOf("|" + item + "|") >= 0) continue;**

value = value + item + "|";

}

mapnew.remove(s[0]);

mapnew.put(s[0], value);

} **else {**

mapnew.put(s[0], s[1]);

}

}

JSONObject tmpJson = **new JSONObject();**

Set<String> setFilter = mapnew.keySet();

Iterator<String> itFilter = setFilter.iterator();

String filters = MysqlDao.*getFilterByAddtion(major, addtion); // 将来考虑放在memcache*

**while (itFilter.hasNext()) {**

String key = itFilter.next();

**if ("价格".equals(key)) continue; // 过滤条件存在价格**

String value = mapnew.get(key);

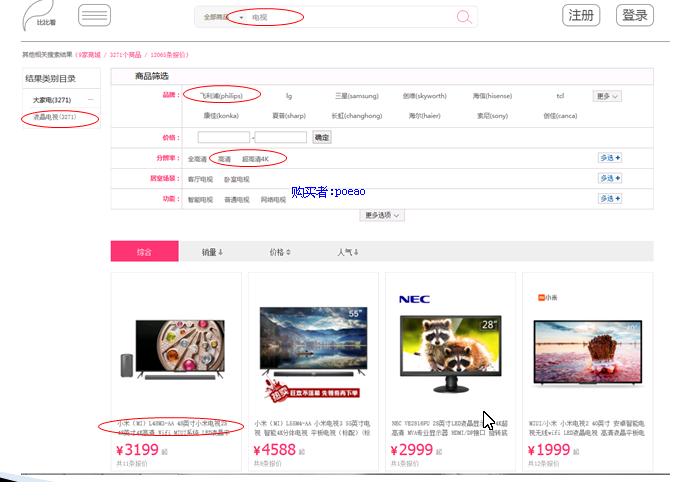
value = *sortFilter(filters, key, value); // 这个过滤的前提是前面检索的是major下的数据*

tmpJson.put(key, value.substring(0, value.length() - 1));

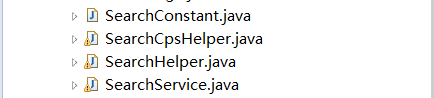
}

## 搜索关键字来源

不管用户在页面上点击或在输入框中输入关键字产生的搜索动作都会触及到搜索引擎，我们来看看在此项目中会有哪些搜索动作呢？



从以上我们可以看出搜索的入口很多，除从关键词搜索外，从类目树、筛选条件和商品详列表点击进入也会促发搜索，不过从类目树和商品列表进入的搜索比较简单，根据类目树点击进入时只需要从索引库中取出对应类别的商品，从商品列表点击进入只需要根据索引rowkey取出商品的详细信息，若根据关键字查询，我们则需要判断输入的关键字是否属于类别、品牌或型号，否则就直接从标题检索，若点击筛选条件过滤，我们则需要结合前面输入的关键字或类目树综合查询获取结果，在接下来的这两节课中我们主要详细分析下商品搜索实现源码



## 获取筛选条件

用户查询时前端发送两次请求，一次返回过筛选条件，一次获取结果数据集，核心代码如下：

if ("1".equals(filter)) {

getFilterValue(request, response);

} else {

……

public void getFilterValue(HttpServletRequest request,

HttpServletResponse response) throws InterruptedException,

Exception {

……

sQuery.setFacet(true);

sQuery.addFacetField(new String[] { "price", "brand", "thisFilter\_s" });

sQuery.setFacetLimit(SearchConstant.*FACET\_LIMIT);// 1000000*

QueryResponse queryResponse;

try {

……

}

}

## 获取结果数据

**if(StringUtils.*isBlank(isProductType)){***

//若是品牌需要判断以品牌方式查询是否有数据，若有跳到品牌查询方法，无就继续本方法

String ss = smodel.toLowerCase().trim();

**long timeb1 = System.*currentTimeMillis();***

isbrand = CacheUtils.*getCached(ss); //品牌放在缓存中*

**long timeb2 = System.*currentTimeMillis();***

//System.out.println("timeb2 - timeb1 = " + (timeb2 - timeb1));

**if(StringUtils.*isNotBlank(isbrand)){***

**if(isbrand.indexOf(";;") >= 0)**

isbrand = isbrand.replaceAll(";;", " OR brand:");

String rs = getSearchByBrandForTitle(request, response, isbrand);

**if(StringUtils.*isNotBlank(rs)){***

response.setContentType("text/html;charset=UTF-8");

response.getWriter().write(rs);

**return;**

}

}

//判断输入的词是否为型号，若为型号就按型号字段检索

**if(StringUtils.*isBlank(isbrand)){***

ismodel = *getModel(smodel,server); //暂时放开*

**if(StringUtils.*isNotBlank(ismodel)){***

String rsm = getSearchByModel(request, response, ismodel);

**if(StringUtils.*isNotBlank(rsm)){***

response.setContentType("text/html;charset=UTF-8");

response.getWriter().write(rsm);

**return;**

}**else{**

**if(StringUtils.*isNotBlank(otherParam) || priceStart > 0 || priceEnd > 0){***

response.setContentType("text/html;charset=UTF-8");

response.getWriter().write(rsm);

**return;**

}

}

}

}

//根据输入条件复合查询

String cpss = SearchCpsHelper.*getSearchByComposites(request, response, smodel, server, major);*

**if(StringUtils.*isNotBlank(cpss)){***

response.setContentType("text/html;charset=UTF-8");

response.getWriter().write(cpss);

**return;**

}

}