**搜索引擎**

## 一、一些概念

**1 倒排索引/反向索引**

输入一个关键词，就会出现相关的文档。如果这里有三篇文档，给一个关键词，就通过字符串匹配的方法就可以找到包含该关键词的文档，这很简单。那么如果有一百篇呢，同样对这一百篇文章逐个进行搜索即可，现代计算机对于上百篇文章的检索还是可以毫秒级的时间完成的。那么，网络上数以万计乃至上亿的文章，难道也要这样逐个搜索吗。索引就是解决搜索缓慢的方案，也就是说将每篇文章进行处理，对每一个出现的词建索引。每个词对应的列表是包含这个词的文章的列表，这被称为倒排索引。于是输入一个词，只要查找这张表，就能很快把包含这个词的文章给找出来。那如果有多个词呢比如，在淘宝上搜索“黄色毛衣”，只要把包含“黄色”的商品和包含“毛衣”的商品求个交集。构建倒排索引是搜索引擎的基础。

**2 倒排索引（英语：Inverted index），也常被称为反向索引、置入档案或反向档案**，是一种索引方法，被用来存储在全文搜索下某个单词在一个文档或者一组文档中的存储位置的映射。它是文档检索系统中最常用的数据结构。通过倒排索引，可以根据单词快速获取包含这个单词的文档列表。倒排索引主要由两个部分组成：“单词词典”和“倒排文件”。

**3 分词**

构建倒排索引的单位是词，词代表了语言中最基本的单位。在英文中，可以通过空格对每个词进行分开，而汉语就相对复杂了，不是通过空格分开的了，需要人通过语义进行分开。上面提到了“黄色毛衣”这个query，可以将“黄色”和“毛衣”分成两个基本的语言单位。但是，计算机来进行汉语分词就相对来说比较困难了。好在目前汉语分词技术已经非常成熟了，也有非常成熟的库进行调用，中科院，复旦等科研机构都对汉语分词技术研究得很深入。

**4 排序**

找出这些文章以后，怎么进行排序，哪篇文章靠前，哪篇文章靠后，也是个问题。我们暂且可以这样来进行排序，按照**相关性**来，如果搜索的query跟文档的标题一样，这个相关性就相对来说比在正文中出现这些query的文档高。如果词的顺序都一模一样，那相关性就更高了，如果一字不差，不多字也不少字，当然是相关性最高了。

**搜索引擎需要快速对用户的搜索进行响应，把最好的结果展现给用户。如果说不能在一秒内返回搜索结果，那么基本上就和这一款搜索引擎拜拜了。**

**5 依赖的技术**

**1.高并发架构**

像百度这样的搜索引擎，每秒钟至少要能扛得住上百万次搜索请求。这是工程方面的问题。如果是用户量级上亿的搜索引擎，需要上百乃至上千的机器来处理请求。针对不同区域的用户，处理用户请求的机器都是不同的。由最开始的电信、移动、联通的网关到搜索引擎前端机器的流量划分，再到后端query处理、索引查询，一个搜索请求很可能需要经过几十台机器的处理，才将结果展现给用户，而这些流程，在毫秒级时间就完成了。对此，需要有反向代理、负载均衡这样的技术进行压力的分流，哪些片区需要多少台机器，保证这一片的机房不会因为流量过大而奔溃，都是经过工程师的多年运维经验，预估出来的。并且要考虑过节、突发事件、自然灾害对机房的影响等因素，进行多地备份。这不仅仅是搜索引擎需要解决的问题，淘宝、微信等任何一款用户量较大的产品也需要考虑大量用户访问带来的高并发处理。

**2.缓存技术**

每秒钟上百万次的搜索请求，都进行query处理、索引查表、求交集、重排序等工作显然也是扛不住的，就算有上百万台机器来处理，这显然是很浪费资源的。这样，缓存技术就有用武之地了。搜索query的热门程度，大致满足长尾分布的，对于一些非常热门的关键词，可能每秒钟会出现好多次请求，这样，如果每次请求都重新处理一次，就是资源的浪费。可以在第一次处理完就把搜索的结果缓存起来，如果再次发生同样的请求，就可以到缓存池里直接把结果拿出来展现了。不在缓存中的结果就需要重新处理，这样每次需要重新处理的请求都是一些冷门的长尾query。对于缓存的策略，可以采用LRU算法，按照最近最少使用的方法进行记录的淘汰，如果系统庞大一点，可以设计多级**缓存。**

**3.索引端召回策略**

虽然建立了倒排索引，但是如果文章的量级很大，每个term所包含的文章都有上亿篇，对于求交和重排都是很耗费时间的。这就需要在索引端就对文章进行**粗打分**，如果分数都低于某个阈值，就不会将文档参与求交与精排序。那么另一个问题来了，这个分数怎样衡量，怎样的文档才算分数高。这就需要对query进行上下文的处理，结合其他query以及query本身指标进行粗打分。

**6 爬虫策略技术**

爬虫是靠几个起点网页出发，通过网页上的链接导向下一个网页，顺藤摸瓜，将尽可能多的网页进行获取，这里就需要设计到底是**深度优先搜索还是广度优先搜索方法**来把文章获取到，同时需要对链接进行分析，比如计算网页的PageRank值，计算网页的权威值。同时，考虑到成本，也不是对所有的网页进行爬取，可以选择性挑选一些质量较高的网页优先进行爬取。至于这个网页质量如何评价，也是需要工程师不断在指标查看下进行优化的。

网页内容是不断在变化的，几个月前，屠呦呦这个名字还不是很火，自从获得诺贝尔奖以后，很多相关的新闻就出来了。经常上网的人可能会发现，消息一发布，就很快就能在搜索引擎上检索到相关的新闻。从时效性出发，就需要爬虫对一些更新较频繁的网页进行频繁地访问，比如说新闻类网站。所以，网站更新频繁度的指标也是工程师们不断关注的。某些爬虫可能侧重于优先搜索更新频繁度较高的网页。那么同时，爬虫的访问会对目标网站带来压力，很多网站对有反爬虫机制，如果爬取地较频繁，很可能会被目标网站加入黑名单。需要在时效性与可用性之间寻找一个平衡。

**7 精准性的几个问题：**

**1.query解析**

对于两个城市类型的query，比如说“北京上海”，可以解析出来，并且将相关火车票和机票的结果返回给用户。在地图检索中，对于“中关村肯德基”这样的query，识别出是商圈加上商户的类型，并将所在商圈的商家都返回给用户。

在淘宝中，检索“好看的衣服”，就能识别出“好看”这样的修饰词和“衣服”这样的中心词，至于好看如何定性，那就是另外一回事了。

另外，类似“北京遇上西雅图首映时间”这样的query，就需要实体识别了，因为分词系统将词切分的粒度比较小，像“北京遇上西雅图”这样的实体，就需要一个智能的系统来进行抽取了，同时在离线端，也需要将“北京遇上西雅图”这个实体建立索引。

复杂query的理解，比如说“谢霆锋的是谁的儿子？”这样的query，需要用复杂的自然语言处理技术将query进行分析，转换成相应的格式去知识库中进行检索。

针对这些不同的query模式，就可以对query进行分类，不同的query请求不用的服务器，然后整合自然搜索的结果，融合起来后进行结果的展示。百度率先推出了阿拉丁平台，给予商户一些接口，这样，搜索“北京上海”，不用再转到去哪儿网，就可以把机票火车票结果获取了。

**2.关键词改写**

用户输入的query可能由于用户个人理解记忆或者输入的手误打错，这会严重影响搜索的质量，如果搜索引擎能判断出用户输入错误，并进行纠正，把正确query的结果返回，那用户体验会得到更大的提升。对于一些明显错误的query，搜索引擎给直接把正确query的结果返回。如果是模棱两可的结果，搜索引擎会给出一些建议的query。

query改写系统比较复杂，在离线阶段，需要对用户的搜索日志挖掘一些常用易出错的query term对，或者人工配置一些可能出错的term对，然后建立一个类似于机器翻译的模型。在在线阶段，按照模型进行候选query的计算。检索时，对原query检索的结果打分，结合改写后query的检索结果打分进行判断，到底是给改写后的query，还是给出一些建议，还是不做任何指示。这就需要进行多次检索，同时，结果的选择也可以做成一个分类的模型。

不仅仅是错误的纠正，同义词的扩展也是query改写的一个非常重要的方面，比如说搜索iPhone需要同时把苹果手机相关的结果召回，搜索咖啡厅，同时也需要把咖啡馆相关的结果返回。同时，简称（北大与北京大学）、别名等改写也是需要采纳的。除了用扩展的办法以外，还可以将query与离线端都进行归一化，比如说“的哥”全部归一成“出租车司机”。至于采取哪种策略，需要结合线上的情况进行选择，或者各种方法的融合。

**3.个性化排序**

不同人对文章的要求不一样。喜欢音乐的人搜索李娜，就希望将歌手李娜的结果排前；喜欢运动的人就希望将运动员李娜的结果排前。传统的搜索引擎对于每个人展现的都是一样的结果，而随着人们需求的增大，个性化的搜索就很热门。这就需要搜索引擎对检索结果打分时考虑的不仅仅是相关性的因素，还需要考虑用户维度的特征。通过对用户历史行为的分析，建立用户的画像，每个用户都可以被量化为一个向量，同时每篇文档也被量化为一个向量，还有用户与文档交互维度的也被量化为一个向量，比如说用户曾经点击过此篇文档或者点击过类似的文档。那么这就是一个机器学习问题。

基于机器学习的排序方法在学术界被称为learn to rank，目前比较成熟的方法有：pointwise、pairwise、listwise。

**4.知识库的建立**

前面提到的类似“谢霆锋的是谁的儿子？”这样的query的后面就不是简单的基于倒排索引的技术了，需要对实体构建知识库了，在这case中，谢霆锋就是一个实体，同时谢贤也是一个实体，同时，这两个实体之间有一个父子关系，为了构建一个强大的知识库，这都是需要添加的信息。于是，类似的问答机器人就出现了，一些号称能达到几岁儿童智商的机器人也被炒得很火。你会发现，这样的机器人知识渊博，深知天文下知地理，其实是靠强大的语音识别技术、语言理解技术、知识库技术、搜索技术支撑的。

## 二、Java 全文搜索引擎框架 Lucene

**1 出现背景**

数据库中的搜索很容易实现，通常都是使用sql语句进行查询，而且能很快的得到查询结果。

为什么数据库搜索很容易？因为数据库中的数据存储是有规律的，有行有列而且数据格式、数据长度都是固定的。

我们生活中的数据总体分为两种：**结构化数据和非结构化数据。**

@结构化数据：指具有固定格式或有限长度的数据，如数据库，元数据等。

@非结构化数据：指不定长或无固定格式的数据，如邮件，word文档等磁盘上的文件

**2 非结构化数据查询方法**

**（1）顺序扫描法(Serial Scanning)**

所谓顺序扫描，比如要找内容包含某一个字符串的文件，就是一个文档一个文档的看，对于每一个文档，从头看到尾，如果此文档包含此字符串，则此文档为我们要找的文件，接着看下一个文件，直到扫描完所有的文件。如利用windows的搜索也可以搜索文 件内容，只是相当的慢。

**（2）全文检索(Full-text Search)**

将非结构化数据中的一部分信息提取出来，重新组织，使其变得有一定结构，然后对此有一定结构的数据进行搜索，从而达到搜索相对较快的目的。这部分从非结构化数据中提取出的然后重新组织的信息，我们称之索引。这种先建立索引，再对索引进行搜索的过程就叫全文检索(Full-text Search)。虽然创建索引的过程也是非常耗时的，但是索引一旦创建就可以多次使用，全文检索主要处理的是查询，所以耗时间创建索引是值得的。

**3 实现全文检索**

可以使用Lucene实现全文检索。Lucene是apache下的一个开放源代码的全文检索引擎工具包。提供了完整的查询引擎和索引引擎，部分文本分析引擎。 **Lucene的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。**

**应用场景：**对于数据量大、数据结构不固定的数据可采用全文检索方式搜索，比如百度、Google等搜索引擎、论坛站内搜索、电商网站站内搜索等。

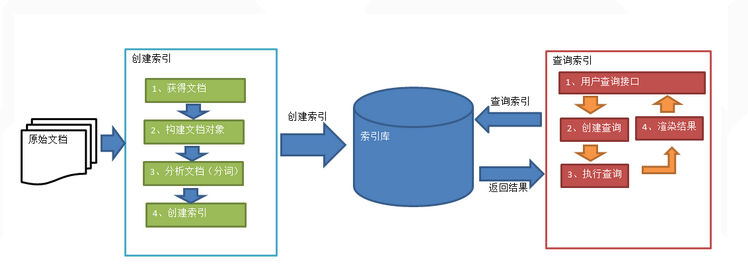
**4 全文索引和搜索流程图**

1、绿色表示索引过程，对要搜索的原始内容进行索引构建一个索引库，索引过程包括：

确定原始内容即要搜索的内容-->采集文档-->创建文档-->分析文档-->索引文档

2、红色表示搜索过程，从索引库中搜索内容，搜索过程包括：

用户通过搜索界面-->创建查询-->执行搜索，从索引库搜索-->渲染搜索结果



**5 创建索引过程**

**1) 获取原始文档**

原始文档 是指要索引和搜索的内容。原始内容包括互联网上的网页（爬虫）、数据库中的数据（sql查询）、磁盘上的文件（IO流获取）等。从互联网上、数据库、文件系统中等获取需要搜索的原始信息，这个过程就是信息采集，信息采集的目的是为了对原始内容进行索引。在Internet上采集信息的软件通常称为爬虫或蜘蛛，也称为网络机器人，爬虫访问互联网上的每一个网页，将获取到的网页内容存储起来。Lucene不提供信息采集的类库，需要自己编写一个爬虫程序实现信息采集，也可以通过一些开源软件实现信息采集，如下：

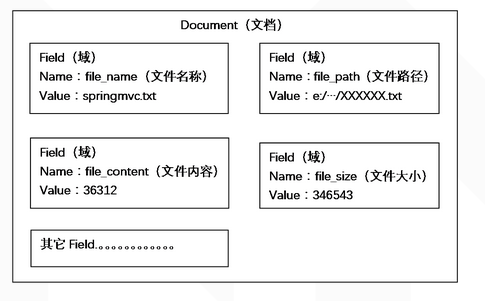
Nutch（http://lucene.apache.org/nutch）, Nutch是apache的一个子项目，包括大规模爬虫工具，能够抓取和分辨web网站数据。

jsoup（http://jsoup.org/ ），jsoup 是一款Java 的HTML解析器，可直接解析某个URL地址、HTML文本内容。它提供了一套非常省力的API，可通过DOM，CSS以及类似于jQuery的操作方法来取出和操作数据。

heritrix（http://sourceforge.net/projects/archive-crawler/files/），Heritrix 是一个由 java 开发的、开源的网络爬虫，用户可以使用它来从网上抓取想要的资源。其最出色之处在于它良好的可扩展性，方便用户实现自己的抓取逻辑。获取磁盘上文件的内容，可以通过文件流来读取文本文件的内容，对于pdf、doc、xls等文件可通过第三方提供的解析工具读取文件内容，比如Apache POI读取doc和xls的文件内容。

**2）创建文档对象**

获取原始内容的目的是为了索引，在索引前需要将原始内容创建成文档（Document），文档中包括一个一个的域（Field），域中存储内容。这里我们可以将磁盘上的一个文件当成一个document，Document中包括一些Field（file\_name文件名称、file\_path文件路径、file\_size文件大小、file\_content文件内容），如下图：注意：每个Document可以有多个Field，不同的Document可以有不同的Field，同一个Document可以有相同的Field（域名和域值都相同）每个文档都有一个唯一的编号，就是文档id。



**3） 分析文档**

将原始内容创建为包含域（Field）的文档（document），需要再对域中的内容进行分析，分析的过程是经过对原始文档提取单词、将字母转为小写、去除标点符号、去除停用词等过程生成最终的语汇单元，可以将语汇单元理解为一个一个的单词。

比如下边的文档经过分析如下：

　　原文档内容：

Lucene is a Java full-text search engine. Lucene is not a complete

application, but rather a code library and API that can easily be used

to add search capabilities to applications.

　　分析后得到的语汇单元：**lucene、java、full、search、engine。。。。**

每个单词叫做一个Term，不同的域中拆分出来的相同的单词是不同的term。term中包含两部分一部分是文档的域名，另一部分是单词的内容。例如：文件名中包含apache和文件内容中包含的apache是不同的term。

将原始内容创建为包含域(Field)的文档(document),需要再对域中的内容进行分析,分析的过程是经过对原始文档提取单词,将字母转为小写,去除标点符号,去除停用词等过程生成最终的语汇单元,可以将语汇单元理解为一个一个的单词;每一个单词叫做一个Term,不同的域中拆分出来的相同的单词是不同的term; term中包含两部分,一部分是文档的域名, 另一部分是单词的内容;

**Field 域的属性**

是否分析: 是否对域的内容进行分词处理;

是否索引: 将 Field 分析后的词或整个 Field 值进行索引,只有建立索引,才能搜索到;

是否存储: 存储在文档中的 Field 才可以从 Document 中获取;



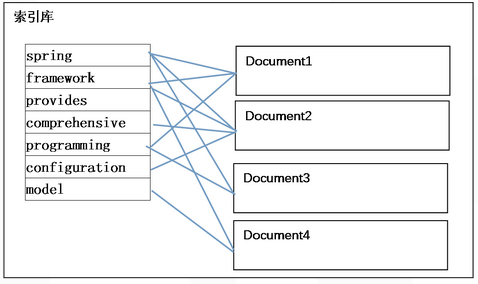
**4） 创建索引**

对所有文档分析得出的语汇单元进行索引，索引的目的是为了搜索，最终要实现只搜索被索引的语汇单元从而找到Document（文档）。

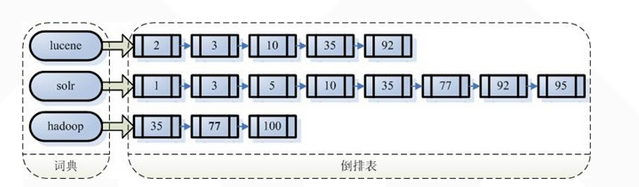
注意：创建索引是对语汇单元索引，通过词语找文档，这种索引的结构叫倒排索引结构。

传统方法是根据文件找到该文件的内容，在文件内容中匹配搜索关键字，这种方法是顺序扫描方法，数据量大、搜索慢。

倒排索引结构是根据内容（词语）找文档，如下图：



　　倒排索引结构也叫反向索引结构，包括索引和文档两部分，索引即词汇表，它的规模较小，而文档集合较大。



**6 查询索引过程**

**1） 用户查询接口**

全文检索系统提供用户搜索的界面供用户提交搜索的关键字，搜索完成展示搜索结果。

Lucene不提供制作用户搜索界面的功能，需要根据自己的需求开发搜索界面。

**2） 创建查询**

用户输入查询关键字执行搜索之前需要先构建一个查询对象，查询对象中可以指定查询要搜索的Field文档域、查询关键字等，查询对象会生成具体的查询语法，

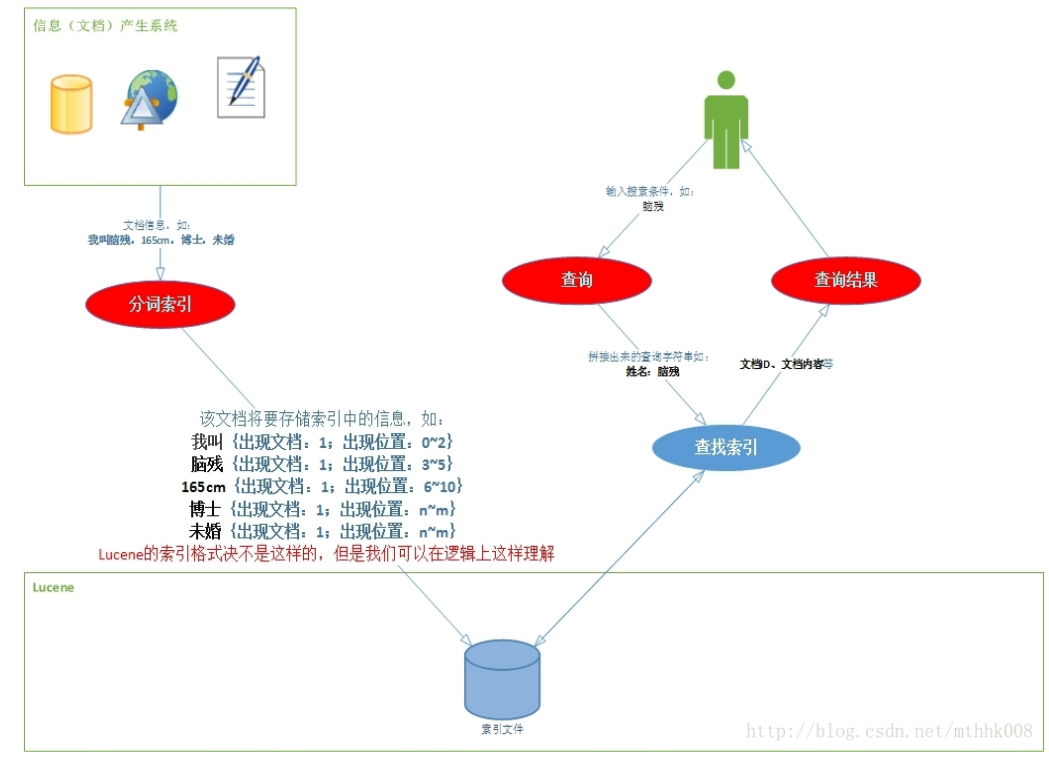
例如：　　语法 “fileName:lucene”表示要搜索Field域的内容为“lucene”的文档

**3） 执行查询**

根据查询语法在倒排索引词典表中分别找出对应搜索词的索引，从而找到索引所链接的文档链表。比如搜索语法为“fileName:lucene”表示搜索出fileName域中包含Lucene的文档。搜索过程就是在索引上查找域为fileName，并且关键字为Lucene的term，并根据term找到文档id列表。

1. **渲染结果**

以一个友好的界面将查询结果展示给用户，用户根据搜索结果找自己想要的信息，为了帮助用户很快找到自己的结果，提供了很多展示的效果，比如搜索结果中将关键字高亮显示，百度提供的快照等。



|  |  |
| --- | --- |
| 包   名 | 功   能 |
| org.apache.lucene.analysis | 语言分析器，主要用于切词，中文切词可以扩展此类 |
| org.apache.lucene.document | 索引存储时的文档结构管理，类似于关系型数据库的表结构 |
| org.apache.lucene.index | 索引管理，包括索引建立、删除等 |
| org.apache.lucene.queryParser | 查询分析器，实现查询关键词的运算，如与、或、非等 |
| org.apache.lucene.search | 检索管理，根据查询条件，检索得到结果 |
| org.apache.lucene.store | 数据存储管理，主要包括一些底层的I/O操作 |
| org.apache.lucene.util | 一些公用类 |

## 三、java全文搜索服务器 Solr

**1 搜索方案选择**

1、基于Lucene自己进行封装实现站内搜索。工作量及扩展性都较大，不采用。

2、调用Google、Baidu的API实现站内搜索。同第三方搜索引擎绑定太死，无法满足后期业务扩展需要，暂时不采用。

3、 基于Compass+Lucene实现站内搜索。适合于对数据库驱动的应用数据进行索引，尤其是替代传统的like ‘%expression%’来实现对varchar或clob等字段的索引，对于实现站内搜索是一种值得采纳的方案。但在分布式处理、接口封装上尚需要自己进行一定程度的封装，暂时不采用。

4、基于Solr实现站内搜索。封装及扩展性较好，提供了较为完备的解决方案，因此在门户社区中采用此方案，后期加入Compass方案。

**2 Solr简介**

Solr是一个基于Lucene的Java搜索引擎服务器。Solr 提供了层面搜索、命中醒目显示并且支持多种输出格式（包括 XML/XSLT 和 JSON 格式）。它易于安装和配置，而且附带了一个基于 HTTP 的管理界面。Solr已经在众多大型的网站中使用，较为成熟和稳定。Solr 包装并扩展了 Lucene，所以Solr的基本上沿用了Lucene的相关术语。更重要的是，Solr 创建的索引与 Lucene 搜索引擎库完全兼容。通过对Solr 进行适当的配置，某些情况下可能需要进行编码，Solr 可以阅读和使用构建到其他 Lucene 应用程序中的索引。此外，很多 Lucene 工具（如Nutch、 Luke）也可以使用Solr 创建的索引。

**3 Solr4.10安装**

下载地址： http://www.apache.org/dyn/closer.cgi/lucene/solr/

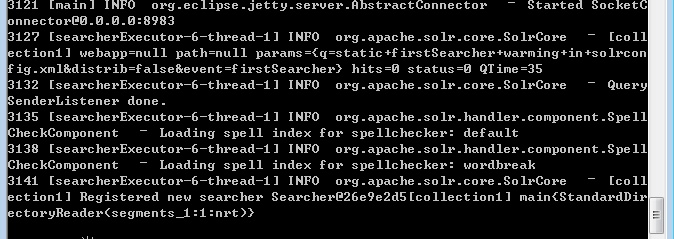
下载得到zip压缩包，下载的版本为4.10.2. 文件大小148MB左右。

1、将solr-4.10.2.zip文件拷贝到E盘；（或者其他盘都可以，只要目录中不要出现中文就行。）

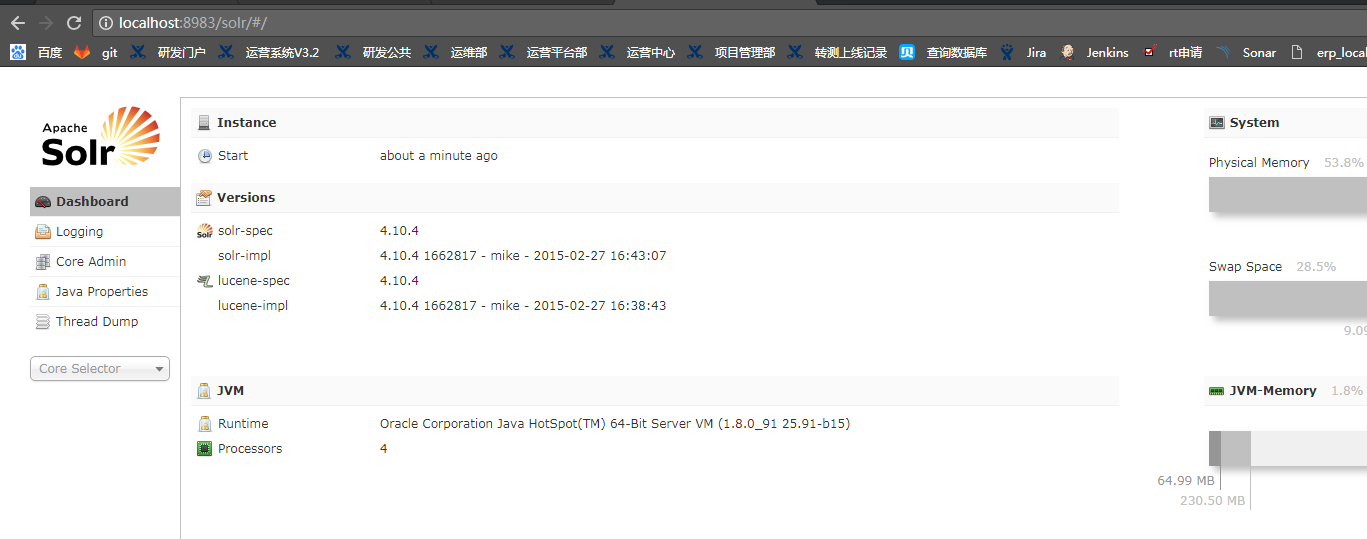
2、解压solr-4.10.2.zip文件，得到solr-4.10.2目录。

3、运行cmd，执行命令进入C:\solr-4.10.2\example

4、执行命令：java -jar start.jar



5、打开浏览器，输入地址：http://localhost:8983/solr/，界面如下：



**4 添加索引数据**

服务启动后，目前你看到的界面没有任何数据，你可以通过POSTing命令向Solr中添加（更新）文档，删除文档，在exampledocs目录包含一些示例文件，运行命令：

**java -jar post.jar solr.xml monitor.xml**

上面的命令是向solr添加了两份文档，打开这两个文件看看里面是什么内容，solr.xml里面的内容是：

<add>

<doc>

<field name="id">SOLR1000</field>

<field name="name">Solr, the Enterprise Search Server</field>

<field name="manu">Apache Software Foundation</field>

<field name="cat">software</field>

<field name="cat">search</field>

<field name="features">Advanced Full-Text Search Capabilities using Lucene</field>

<field name="features">Optimized for High Volume Web Traffic</field>

<field name="features">Standards Based Open Interfaces - XML and HTTP</field>

<field name="features">Comprehensive HTML Administration Interfaces</field>

<field name="features">Scalability - Efficient Replication to other Solr Search Servers</field>

<field name="features">Flexible and Adaptable with XML configuration and Schema</field>

<field name="features">Good unicode support: h&#xE9;llo (hello with an accent over the e)</field>

<field name="price">0</field>

<field name="popularity">10</field>

<field name="inStock">true</field>

<field name="incubationdate\_dt">2006-01-17T00:00:00.000Z</field>

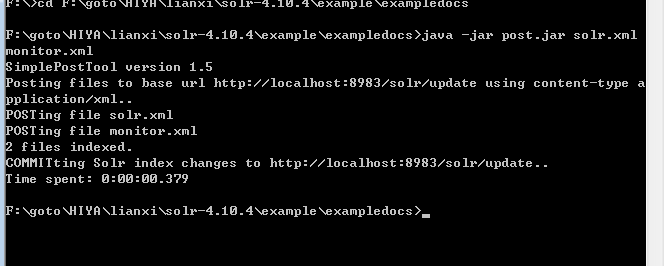
</doc>

</add>

表示向索引中添加一个文档，文档就是用来搜索的数据源，现在就可以通过管理界面搜索关键字”solr”，具体步骤是：

点击页面下的Execute Query按钮后右侧就会显示查询结果，这个结果就是刚才导入进去的solr.xml的json格式的展示结果。

solr支持丰富的查询语法，比如：现在想搜索字段name里面的关键字”Search”就可以用语法name:search，当然如果你搜索name:xxx就没有返回结果了，因为文档中没有这样的内容。



**5 数据导入**

导入数据到Solr的方式也是多种多样的：

1、可以使用DIH(DataImportHandler)从数据库导入数据

2、支持CSV文件导入，因此Excel数据也能轻松导入

3、支持JSON格式文档

4、二进制文档比如：Word、PDF

5、还能以编程的方式来自定义导入

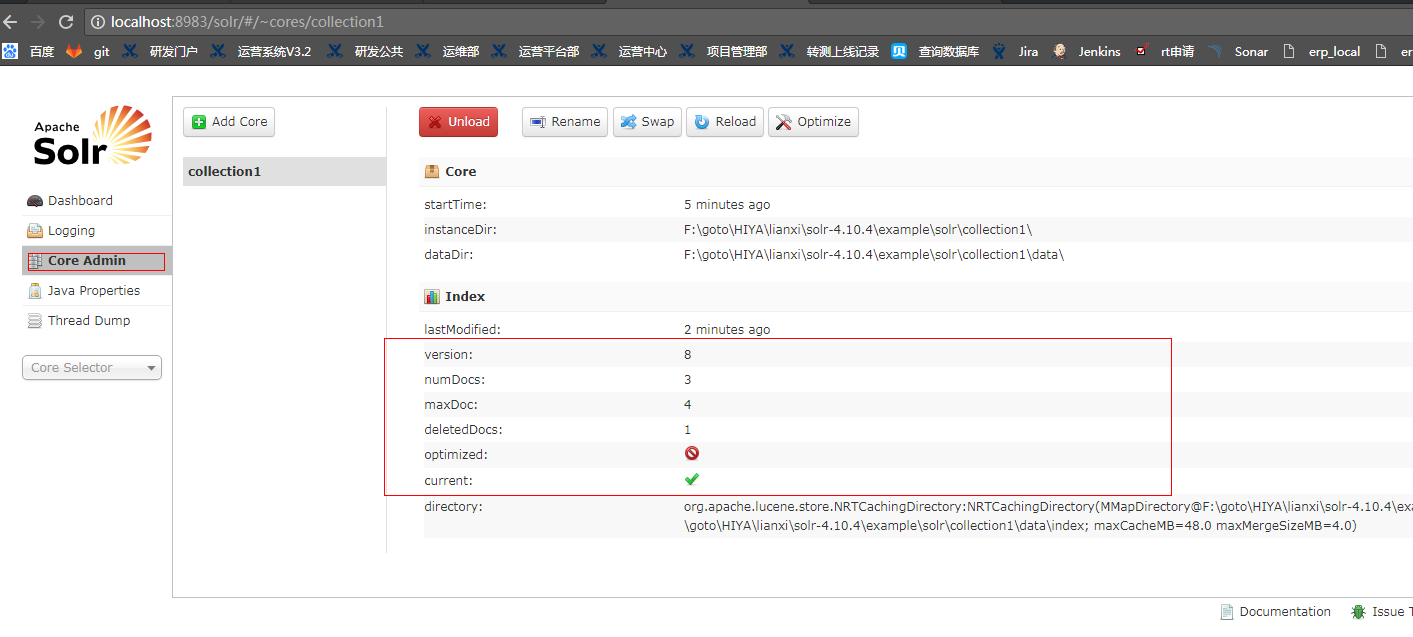
**6 更新数据**

如果同一份文档solr.xml重复导入会出现什么情况呢？实际上solr会根据文档的字段id来唯一标识文档，如果导入的文档的id已经存在solr中，那么这份文档就被最新导入的同id的文档自动替换。

1、numDocs：当前系统中的文档数量，它有可能大于xml文件个数，因为一个xml文件可能有多个doc标签。

2、maxDoc：maxDoc有可能比numDocs的值要大，比如重复post同一份文件后，maxDoc值就增大了。

3、deletedDocs：重复post的文件会替换掉老的文档，同时deltedDocs的值也会加1，不过这只是逻辑上的删除，并没有真正从索引中移除掉



**7 删除数据**

通过id删除指定的文档，或者通过一个查询来删除匹配的文档

java -Ddata=args -jar post.jar "<delete><id>SOLR1000</id></delete>"

java -Ddata=args -jar post.jar "<delete><query>name:DDR</query></delete>"

java -Ddata=args -Dcommit=false -jar post.jar "<delete><id>3007WFP</id></delete>" java -jar post.jar -提交事务，文档就彻底删除了。

此时solr.xml文档从索引中删除了，再次搜”solr”时不再返回结果。当然solr也有数据库中的事务，执行删除命令的时候事务自动提交了，文档就会立即从索引中删除。你也可以把commit设置为false，手动提交事务。

**8 删除所有数据**

http://localhost:8983/solr/collection1/update?stream.body=<delete><query>\*:\*</query></delete>&commit=true

**9 删除指定数据**

http://localhost:8983/solr/collection1/update?stream.body=<delete><query>title:abc</query></delete>&commit=true

**10 多条件删除**

http://localhost:8983/solr/collection1/update?stream.body=<delete><query>title:abc AND name:zhang</query></delete>&commit=true

**11 查询数据**

查询数据都是通过HTTP的GET请求获取的，搜索关键字用参数q指定，另外还可以指定很多可选的参数来控制信息的返回，例如：用fl指定返回的字段，比如f1=name，那么返回的数据就只包括name字段的内容

http://localhost:8983/solr/collection1/select?q=solr&fl=name&wt=json&indent=true

**12 排序**

Solr提供排序的功能，通过参数sort来指定，它支持正序、倒序，或者多个字段排序

q=video&sort=price desc

q=video&sort=price asc

q=video&sort=inStock asc, price desc

**13高亮**

网页搜索中，为了突出搜索结果，可能会对匹配的关键字高亮出来，Solr提供了很好的支持，只要指定参数：

hl=true #开启高亮功能

hl.fl=name #指定需要高亮的字段

**14 文本分析**

文本字段通过把文本分割成单词以及运用各种转换方法（如：小写转换、复数移除、词干提取）后被索引，schema.xml文件中定义了字段在索引中，这些字段将作用于其中.

默认情况下搜索”power-shot”是不能匹配”powershot”的，通过修改schema.xml文件(solr/example/solr/collection1/conf目录)，把features和text字段替换成”text\_en\_splitting”类型，就能索引到了。

<field name="features" type="text\_en\_splitting" indexed="true" stored="true" multiValued="true"/>

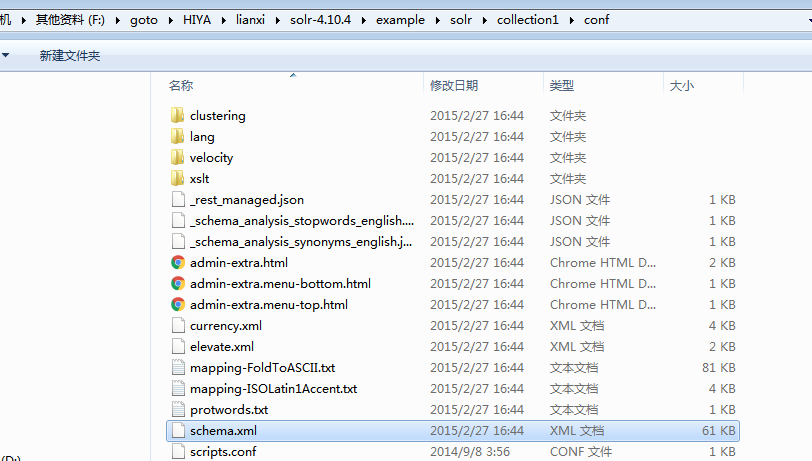
<field name="text" type="text\_en\_splitting" indexed="true" stored="false" multiValued="true"/>

修改完后重启solr，然后重新导入文档，现在就可以匹配了。

**15 Solr相关配置简介**

1、schema.xml

schema.xml这个配置文件可以在你下载solr包的安装解压目录的\solr\example\solr\collection1\conf中找到，它就是solr模式关联的文件。



<?xml version="1.0" encoding="UTF-8" ?>

略...

<!--

这是Solr的schema文件，应该命名为schema.xml，并且在solr home的conf目录下

（如，默认在./solr/conf/schema.xml）.

有关如何根据需要定制化该文件，请参照：

http://wiki.apache.org/solr/SchemaXml 性能须知: 这里包含了很多实际应用不需要的可选项。 为改善性能，你可以：

- 尽量将所有仅用于搜索，而不用于实际返回的字段设置stored="false"；

- 尽量将所有仅用于返回，而不用于搜索的字段设置indexed="false"；

- 去掉所有不需要的copyField 语句；

- 为了达到最佳的索引大小和搜索性能,对所有的文本字段设置indexed="false"，

使用copyField将他们拷贝到“整合字段”name="text"的字段中，使用整合字段进行搜索；

- 使用server模式来运行JVM,同时将log级别调高, 避免输出所有请求的日志。

-->

<schema name="example" version="1.5">

略...

<fields>

<!-- fields各个属性说明:

name: 必须属性 - 字段名

type: 必须属性 - <types>中定义的字段类型

indexed: 如果字段需要被索引（用于搜索或排序），属性值设置为true

stored: 如果字段内容需要被返回，值设置为true

docValues: 如果这个字段应该有文档值（doc values），设置为true。文档值在门

面搜索，分组，排序和函数查询中会非常有用。虽然不是必须的，而且会导致生成

索引变大变慢，但这样设置会使索引加载更快，更加NRT友好，更高的内存使用效率。

然而也有一些使用限制：目前仅支持StrField, UUIDField和所有 Trie\*Fields,

并且依赖字段类型, 可能要求字段为单值（single-valued）的,必须的或者有默认值。

multiValued: 如果这个字段在每个文档中可能包含多个值，设置为true

termVectors: [false] 设置为true后，会保存所给字段的相关向量（vector）

当使用MoreLikeThis时, 用于相似度判断的字段需要设置为stored来达到最佳性能.

termPositions: 保存和向量相关的位置信息，会增加存储开销

termOffsets: 保存 offset 和向量相关的信息，会增加存储开销

required: 字段必须有值，否则会抛异常

default: 在增加文档时，可以根据需要为字段设置一个默认值，防止为空

-->

<!-- 字段名由字母数字下划线组成，且不能以数字开头。两端为下划线的字段为保留字段，

如(\_version\_)。

-->

<field name="id" type="string" indexed="true" stored="true"

required="true" multiValued="false" />

<field name="title" type="text\_general" indexed="true"

stored="true" multiValued="true"/>

<field name="description" type="text\_general" indexed="true" stored="true"/>

<field name="author" type="text\_general" indexed="true" stored="true"/>

<field name="keywords" type="text\_general" indexed="true" stored="true"/>

<field name="category" type="text\_general" indexed="true" stored="true"/>

<field name="url" type="text\_general" indexed="true" stored="true"/>

<field name="last\_modified" type="date" indexed="true" stored="true"/>

<!-- 注意: 为了节省空间,这个字段默认不被索引, 因使用copyField被拷贝到了名为text的字段中

。用于内容返回和高亮。搜索时使用text字段

-->

<field name="content" type="text\_general" indexed="false"

stored="true" multiValued="true"/>

<!-- 整合字段(catchall field), 包含其他可搜索的字段 （通过copyField实现） -->

<field name="text" type="text\_general" indexed="true"

stored="false" multiValued="true"/>

<!-- 保留字段，不能删除，否则报错 -->

<field name="\_version\_" type="long" indexed="true" stored="true"/>

</fields>

<!-- 文档的唯一标识，可理解为主键，除非标识为required="false", 否则值不能为空-->

<uniqueKey>id</uniqueKey>

<!-- 拷贝需要索引的字段到整合字段中 -->

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

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

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

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

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

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

<types>

<!-- 字段类型定义 -->

<fieldType name="string" class="solr.StrField" sortMissingLast="true" />

<fieldType name="boolean" class="solr.BoolField" sortMissingLast="true"/>

<fieldType name="int" class="solr.TrieIntField" precisionStep="0"

positionIncrementGap="0"/>

<fieldType name="float" class="solr.TrieFloatField" precisionStep="0"

positionIncrementGap="0"/>

<fieldType name="long" class="solr.TrieLongField" precisionStep="0"

positionIncrementGap="0"/>

<fieldType name="double" class="solr.TrieDoubleField" precisionStep="0"

positionIncrementGap="0"/>

<fieldType name="date" class="solr.TrieDateField" precisionStep="0"

positionIncrementGap="0"/>

略...

<!-- Thai，泰语类型字段 -->

<fieldType name="text\_th" class="solr.TextField" positionIncrementGap="100">

<analyzer>

<tokenizer class="solr.StandardTokenizerFactory"/>

<filter class="solr.LowerCaseFilterFactory"/>

<filter class="solr.ThaiWordFilterFactory"/>

<filter class="solr.StopFilterFactory" ignoreCase="true"

words="lang/stopwords\_th.txt" />

</analyzer>

</fieldType>

<!-- Turkish，土耳其语类型字段 -->

<fieldType name="text\_tr" class="solr.TextField" positionIncrementGap="100">

<analyzer>

<tokenizer class="solr.StandardTokenizerFactory"/>

<filter class="solr.TurkishLowerCaseFilterFactory"/>

<filter class="solr.StopFilterFactory" ignoreCase="false"

words="lang/stopwords\_tr.txt" />

<filter class="solr.SnowballPorterFilterFactory" language="Turkish"/>

</analyzer>

</fieldType>

<!-- Chinese，需要我们自己配置，整合mmseg4j就配置在这里 -->

</types>

<!-- 文档相似度判断依赖于文档相似度得分。 一个自定义的 Similarity 或 SimilarityFactory

可以在这里指定, 但是默认的设置已经适合大多数应用。可以参考:

http://wiki.apache.org/solr/SchemaXml#Similarity

-->

<!--

<similarity class="com.example.solr.CustomSimilarityFactory">

<str name="paramkey">param value</str>

</similarity>

-->

</schema>

**16 性能优化**

1、 将所有只用于搜索的，而不需要作为结果的field（特别是一些比较大的field）的stored设置为false；

2、 将不需要被用于搜索的，而只是作为结果返回的field的indexed设置为false；

3、 删除所有不必要的copyField声明为了索引字段的最小化和搜索的效率；

4、 将所有的 text fields的index都设置成false，然后使用copyField将他们都复制到一个总的 text field上，然后进行搜索。

**17 solrconfig.xml配置文件主要定义了SOLR的一些处理规则，包括索引数据的存放位置，更新，删除，查询的一些规则配置。**

<dataDir>${solr.data.dir:d:/Server/Solr/data}</dataDir> 定义了索引数据和日志文件的存放位置

<luceneMatchVersion>4.10.1</luceneMatchVersion> 表示solr底层使用的是lucene4.8

<lib dir="../../../contrib/extraction/lib"regex=".\*\.jar"/> 表示solr引用包的位置,当dir对应的目录不存在时候，会忽略此属性

**18 directoryFactory索引存储方案，共有以下存储方案**

1、 solr.StandardDirectoryFactory,这是一个基于文件系统存储目录的工厂，它会试图选择最好的实现基于你当前的操作系统和Java虚拟机版本。

2、 solr.SimpleFSDirectoryFactory,适用于小型应用程序，不支持大数据和多线程。

3、 solr.NIOFSDirectoryFactory,适用于多线程环境，但是不适用在windows平台（很慢），是因为JVM还存在bug。

4、solr.MMapDirectoryFactory,这个是solr3.1到4.0版本在linux64位系统下默认的实现。它是通过使用虚拟内存和内核特性调用mmap去访问存储在磁盘中的索引文件。它允许lucene或solr直接访问I/O缓存。如果不需要近实时搜索功能，使用此工厂是个不错的方案。

5、 solr.NRTCachingDirectoryFactory,此工厂设计目的是存储部分索引在内存中，从而加快了近实时搜索的速度。

6、 solr.RAMDirectoryFactory,这是一个内存存储方案，不能持久化存储，在系统重启或服务器crash时数据会丢失。且不支持索引复.

<directoryFactory class="${solr.directoryFactory:solr.NRTCachingDirectoryFactory}" name="DirectoryFactory">

<str name="solr.hdfs.home">${solr.hdfs.home:}</str>

<str name="solr.hdfs.confdir">${solr.hdfs.confdir:}</str>

<str name="solr.hdfs.blockcache.enabled">${solr.hdfs.blockcache.enabled:true}</str>

<str name="solr.hdfs.blockcache.global">${solr.hdfs.blockcache.global:true}</str>

</directoryFactory>

**codecFactory编解码工厂允许使用自定义的编解码器。**

例如：如果想启动per-field DocValues格式, 可以在solrconfig.xml里面设置SchemaCodecFactory：

docValuesFormat=”Lucene42”: 这是默认设置，所有数据会被加载到堆内存中。

docValuesFormat=”Disk”: 这是另外一个实现，将部分数据存储在磁盘上。

docValuesFormat=”SimpleText”: 文本格式，非常慢，用于学习。

<codecFactory class="solr.SchemaCodecFactory"/>

<schemaFactory class="ClassicIndexSchemaFactory"/>

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

//限制token最大长度

<writeLockTimeout>1000</writeLockTimeout>

//IndexWriter等待解锁的最长时间（毫秒）。

<maxIndexingThreads>8</maxIndexingThreads>

//indexWrite生成索引时使用的最大线程数

<useCompoundFile>false</useCompoundFile>

//solr默认为false。如果为true,索引文件减少,检索性能降低,追求平衡。

<ramBufferSizeMB>100</ramBufferSizeMB>

//缓存

<maxBufferedDocs>1000</maxBufferedDocs>

//同上。两个同时定义时命中较低的那个。

<mergePolicy class="org.apache.lucene.index.TieredMergePolicy">

<int name="maxMergeAtOnce">10</int>

<int name="segmentsPerTier">10</int>

</mergePolicy>

//合并策略。

<mergeFactor>10</mergeFactor>

//合并因子,每次合并多少个segments。

<mergeScheduler class="org.apache.lucene.index.ConcurrentMergeScheduler"/>

//合并调度器。

<lockType>${solr.lock.type:native}</lockType>

//锁工厂。

设置索引库的锁方式，主要有三种：

1. single：适用于只读的索引库，即索引库是定死的，不会再更改

2. native：使用本地操作系统的文件锁方式，不能用于多个solr服务共用同一个索引库。Solr3.6 及后期版本使用的默认锁机制。

3. simple：使用简单的文件锁机制

<unlockOnStartup>false</unlockOnStartup>

//是否启动时先解锁。

<termIndexInterval>128</termIndexInterval>

//Lucene loads terms into memory 间隔

<reopenReaders>true</reopenReaders>

//重新打开,替代先关闭-再打开。

<deletionPolicy class="solr.SolrDeletionPolicy">

//提交删除策略,必须实现org.apache.lucene.index.IndexDeletionPolicy

<infoStream file="INFOSTREAM.txt">false</infoStream>

//相当于把创建索引时的日志输出。

<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。随着索引库的频繁更新，tlog文件会越来越大，所以建议提交索引时采用硬提交方式，即批量提交。

<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。

<updateHandler class="solr.DirectUpdateHandler2">

<!-- 允许事务日志 -->

<updateLog>

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

</updateLog>

<!--

在满足一定条件时自动提交。maxDocs/maxTime/openSearcher

-->

<autoCommit>

<maxTime>15000</maxTime>

<openSearcher>false</openSearcher>

</autoCommit>

<!-- 软提交VS硬提交 -->

<!--

<autoSoftCommit>

<maxTime>1000</maxTime>

</autoSoftCommit>

-->

<!--

更新相关事件监听器

postCommit - fired after every commit or optimize command

postOptimize - fired after every optimize command

-->

<!-- The RunExecutableListener executes an external command from a

hook such as postCommit or postOptimize.

exe - the name of the executable to run

dir - dir to use as the current working directory. (default=".")

wait - the calling thread waits until the executable returns.

(default="true")

args - the arguments to pass to the program. (default is none)

env - environment variables to set. (default is none)

-->

<!--

<listener event="postCommit" class="solr.RunExecutableListener">

<str name="exe">solr/bin/snapshooter</str>

<str name="dir">.</str>

<bool name="wait">true</bool>

<arr name="args"> <str>arg1</str> <str>arg2</str> </arr>

<arr name="env"> <str>MYVAR=val1</str> </arr>

</listener>

-->

</updateHandler>

**19 Query 查询**

<maxBooleanClauses>1024</maxBooleanClauses>

设置boolean 查询中，最大条件数。在范围搜索或者前缀搜索时，会产生大量的 boolean 条件，如果条件数达到这个数值时，将抛出异常，限制这个条件数，可以防止条件过多查询等待时间过长。

<filterCache class="solr.FastLRUCache" size="512" initialSize="512" autowarmCount="0"/>

<queryResultCache class="solr.LRUCache" size="512" initialSize="512" autowarmCount="0"/>

<documentCache class="solr.LRUCache" size="512" initialSize="512" autowarmCount="0"/>

<queryResultMaxDocsCached>200</queryResultMaxDocsCached>

<maxWarmingSearchers>2</maxWarmingSearchers>

**20加入IK中文分词器,进行中文分词**

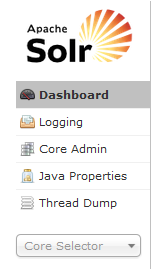
首先将IKAnalyzer-2012-4x.jar拷贝到example\solr-webapp\webapp\WEB-INF\lib下；然后在schema.xml文件中添加fieldType：

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

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

</fieldType>

**21界面管理**



1、Dashboard：就是上面全图就是这个按钮的功能；

2、Logging：查看日志；

3、CoreAdmin：添加core用户，这个可重要也可不重要，默认是collection1，这个就是添加一个和collection1具有一样功能的用户，用来检索和查询。点击之后会出现下面的截图：

这里写图片描述

补充：

name：给core随便起个名字；

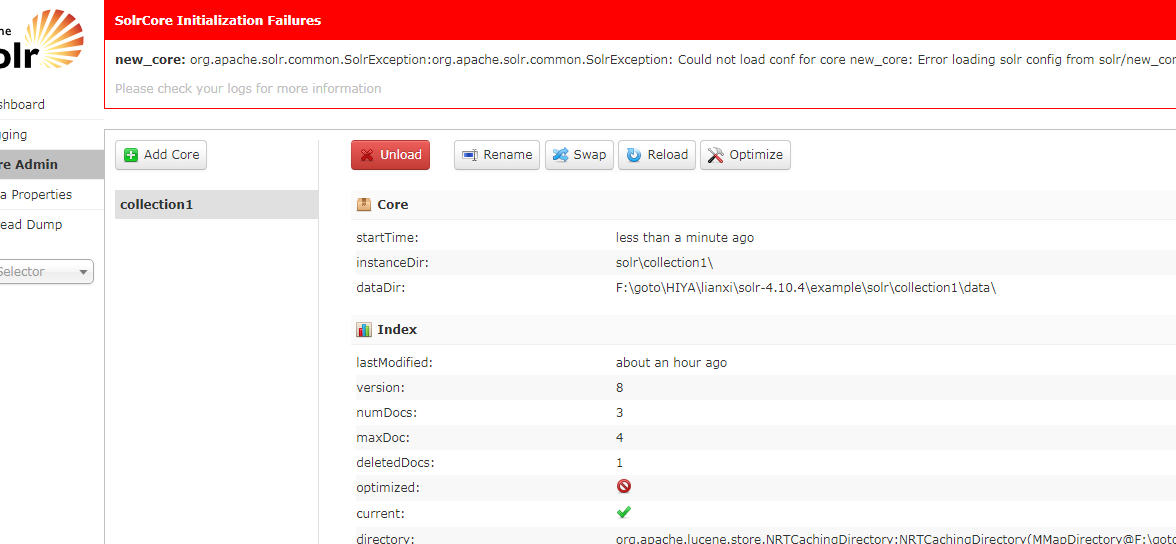
instanceDir：core的安装目录，这里就是之前在tomcat/solrhome/目录下创建的core1文件夹；

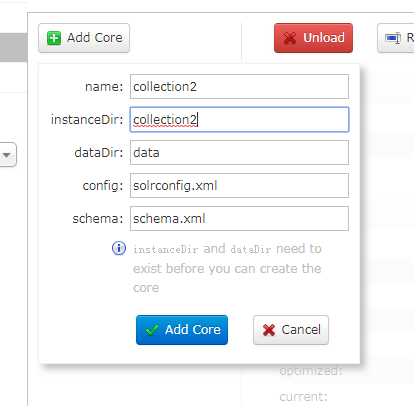
dataDir：指定用于存放lucene索引和log日志文件的目录路径，该路径是相对于core根目录(在单core模式下，就直接是相对于solr\_home了)，默认值是当前core目录下的data；

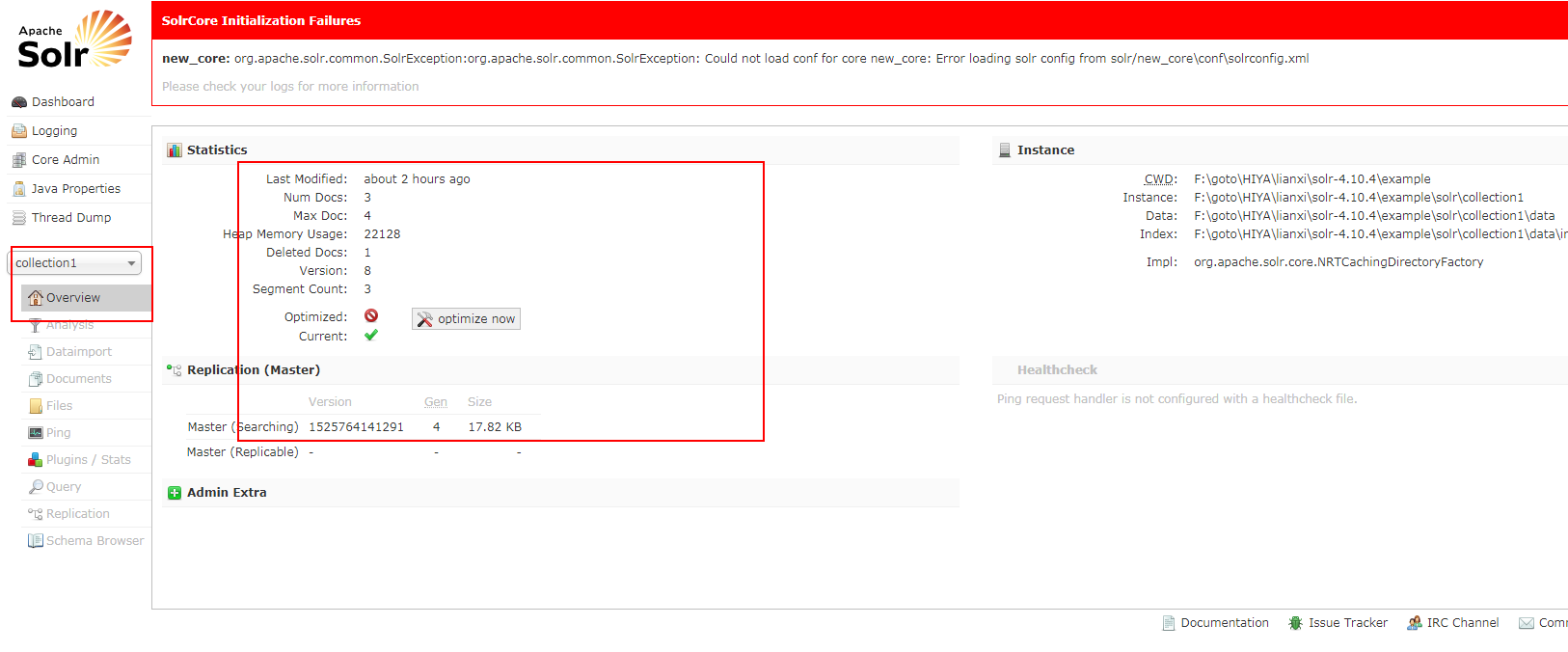
config：用于指定solrconfig.xml配置文件的文件名，启动时会去core1/config目录下去查找；

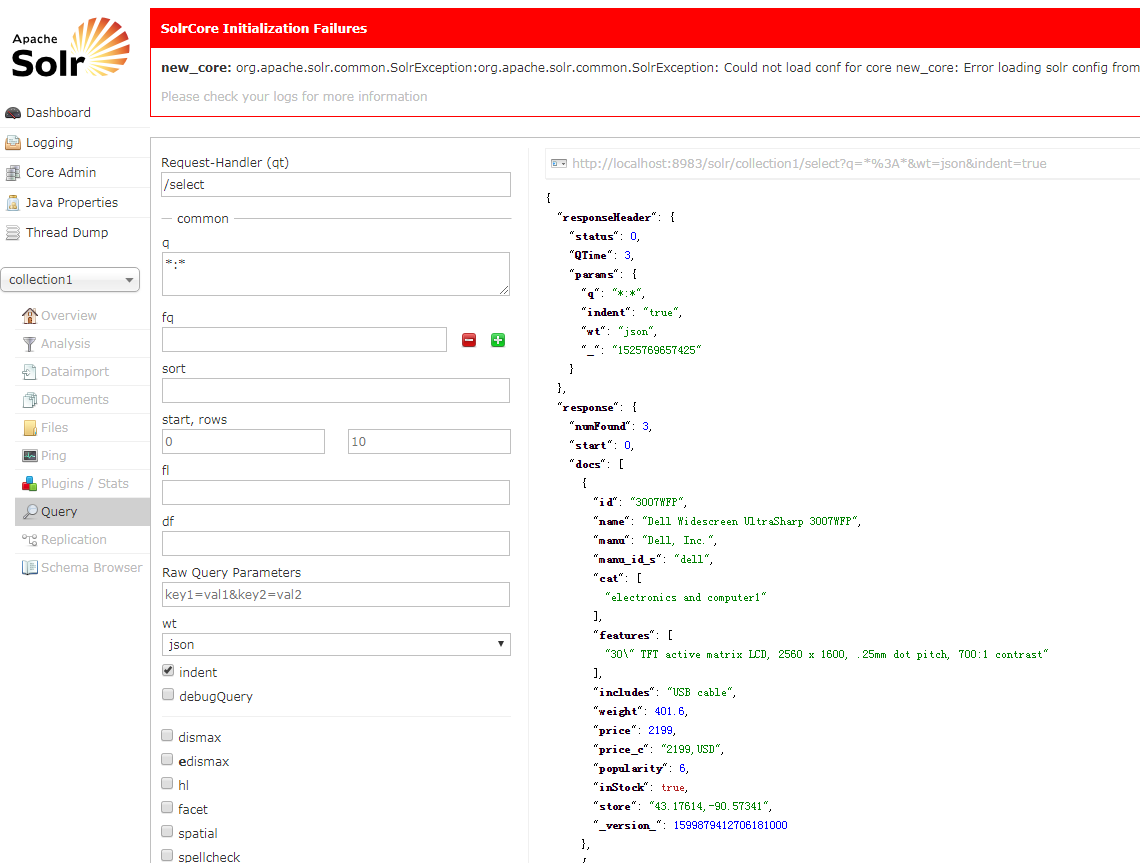
schema：即用来配置你的schema.xml配置文件的文件名的，schema.xml配置文件应该存放在当前core目录下的conf目录下。但是下载的solr里没有这个文件，所以我也不管了；

属性都填上，然后点击AddCore，就创建完成了。











q：查询字符串，必须写，格式为：“索引属性：属性值”，必须遵守这种格式，否则查不出来。

fq：filterquery。使用Filter Query可以充分利用Filter QueryCache，提高检索性能。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 TO20091031] 找关键字mm，并且date\_time是20081001到20091031之间的。

fl：fieldlist。指定返回结果字段。以空格“ ”或逗号“,”分隔。

start：用于分页定义结果起始记录数，默认为0。

rows：用于分页定义结果每页返回记录数，默认为10。

sort：排序，格式:

sort=<fieldname>+<desc|asc>[,<field name>+<desc|asc>]… 。

示例：（inStockdesc, price asc）表示先 “inStock” 降序, 再 “price” 升序，默认相关性降序。

df：默认的查询字段，一般默认指定。

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。必须大写

wt：writertype。指定查询输出结构格式，默认为“xml”。在solrconfig.xml中定义了查询输出格式：xml、json、Python、ruby、PHP、phps、custom。

qt：querytype，指定查询使用的Query Handler，默认为“standard”。

explainOther：设置当debugQuery=true时，显示其他的查询说明。

defType：设置查询解析器名称。

timeAllowed：设置查询超时时间。

omitHeader：设置是否忽略查询结果返回头信息，默认为“false”。

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。

debugQuery：设置返回结果是否显示Debug信息。

**22 基本查询**

　　q 查询的关键字，此参数最为重要，例如，q=id:1，默认为q=\*:\*，

　　fl 指定返回哪些字段，用逗号或空格分隔，注意：字段区分大小写，例如，fl= id,title,sort

　　start 返回结果的第几条记录开始，一般分页用，默认0开始

　　rows 指定返回结果最多有多少条记录，默认值为 10，配合start实现分页

　　sort 排序方式，例如id desc 表示按照 “id” 降序

　　wt (writer type)指定输出格式，有 xml, json, php等

　　fq （filter query）过虑查询，提供一个可选的筛选器查询。返回在q查询符合结果中同时符合的fq条件的查询结果，例如：q=id:1&fq=sort:[1 TO 5]，找关键字id为1 的，并且sort是1到5之间的。

　　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。

**23 Solr的检索运算符**

　　“:” 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。

　　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²

　　( ) 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:[201507 TO 201510]

　　{} 不包含范围检索，如检索某时间段记录，不包含头尾date:{201507 TO 201510}

**24 高亮**

　　h1 是否高亮，hl=true，表示采用高亮

　　hl.fl 设定高亮显示的字段，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。

## 四、Java 操作solr

**1 SolrJ是操作Solr的JAVA客户端**，它提供了增加、修改、删除、查询Solr索引的JAVA接口。SolrJ针对 Solr提供了Rest 的HTTP接口进行了封装， SolrJ底层是通过使用httpClient中的方法来完成Solr的操作。

<dependency>

<groupId>org.apache.solr</groupId>

<artifactId>solr-solrj</artifactId>

<version>5.3.1</version>

</dependency>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.11</version>

</dependency>

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-log4j12</artifactId>

<version>1.7.7</version>

</dependency>

<dependency>

<groupId>commons-logging</groupId>

<artifactId>commons-logging</artifactId>

<version>1.1.3</version>

</dependency>

package entity;

import java.io.IOException;

import java.util.List;

import org.apache.solr.client.solrj.SolrQuery;

import org.apache.solr.client.solrj.SolrServerException;

import org.apache.solr.client.solrj.impl.HttpSolrClient;

import org.apache.solr.client.solrj.response.QueryResponse;

import org.apache.solr.common.SolrDocument;

import org.apache.solr.common.SolrDocumentList;

import org.apache.solr.common.SolrInputDocument;

public class SolrJTest {

//指定solr服务器的地址

private final static String SOLR\_URL = "http://localhost:8080/solr/";

/\*\*

\* 创建SolrServer对象

\*

\* 该对象有两个可以使用，都是线程安全的

\* 1、CommonsHttpSolrServer：启动web服务器使用的，通过http请求的

\* 2、 EmbeddedSolrServer：内嵌式的，导入solr的jar包就可以使用了

\* 3、solr 4.0之后好像添加了不少东西，其中CommonsHttpSolrServer这个类改名为HttpSolrClient

\*

\* @return

\*/

public HttpSolrClient createSolrServer(){

HttpSolrClient solr = null;

solr = new HttpSolrClient(SOLR\_URL);

return solr;

}

/\*\*

\* 往索引库添加文档

\* @throws IOException

\* @throws SolrServerException

\*/

public void addDoc() throws SolrServerException, IOException{

//构造一篇文档

SolrInputDocument document = new SolrInputDocument();

//往doc中添加字段,在客户端这边添加的字段必须在服务端中有过定义

document.addField("id", "8");

document.addField("name", "周新星");

document.addField("description", "一个灰常牛逼的军事家");

//获得一个solr服务端的请求，去提交 ,选择具体的某一个solr core

HttpSolrClient solr = new HttpSolrClient(SOLR\_URL + "my\_core");

solr.add(document);

solr.commit();

solr.close();

}

/\*\*

\* 根据id从索引库删除文档

\*/

public void deleteDocumentById() throws Exception {

//选择具体的某一个solr core

HttpSolrClient server = new HttpSolrClient(SOLR\_URL+"my\_core");

//删除文档

server.deleteById("8");

//删除所有的索引

//solr.deleteByQuery("\*:\*");

//提交修改

server.commit();

server.close();

}

/\*\*

\* 查询

\* @throws Exception

\*/

public void querySolr() throws Exception{

HttpSolrClient solrServer = new HttpSolrClient(SOLR\_URL+"my\_core/");

SolrQuery query = new SolrQuery();

//下面设置solr查询参数

//query.set("q", "\*:\*");// 参数q 查询所有

query.set("q","周星驰");//相关查询，比如某条数据某个字段含有周、星、驰三个字 将会查询出来 ，这个作用适用于联想查询

//参数fq, 给query增加过滤查询条件

query.addFilterQuery("id:[0 TO 9]");//id为0-4

//给query增加布尔过滤条件

//query.addFilterQuery("description:演员"); //description字段中含有“演员”两字的数据

//参数df,给query设置默认搜索域

query.set("df", "name");

//参数sort,设置返回结果的排序规则

query.setSort("id",SolrQuery.ORDER.desc);

//设置分页参数

query.setStart(0);

query.setRows(10);//每一页多少值

//参数hl,设置高亮

query.setHighlight(true);

//设置高亮的字段

query.addHighlightField("name");

//设置高亮的样式

query.setHighlightSimplePre("<font color='red'>");

query.setHighlightSimplePost("</font>");

//获取查询结果

QueryResponse response = solrServer.query(query);

//两种结果获取：得到文档集合或者实体对象

//查询得到文档的集合

SolrDocumentList solrDocumentList = response.getResults();

System.out.println("通过文档集合获取查询的结果");

System.out.println("查询结果的总数量：" + solrDocumentList.getNumFound());

//遍历列表

for (SolrDocument doc : solrDocumentList) {

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

}

//得到实体对象

List<Person> tmpLists = response.getBeans(Person.class);

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

System.out.println("通过文档集合获取查询的结果");

for(Person per:tmpLists){

System.out.println("id:"+per.getId()+" name:"+per.getName()+" description:"+per.getDescription());

}

}

}

public static void main(String[] args) throws Exception {

SolrJTest solr = new SolrJTest();

//solr.createSolrServer();

solr.addDoc();

solr.deleteDocumentById();

solr.querySolr();

}

}

**五、Solr实际应用**

当我们在前台页面搜索商品名称关键词时, 我们这时是在Solr库中去查找相应的商品信息, 然后将搜索关键词高亮.当我们在给商品上架的时候, 将商品信息update 到mysql数据库中的bbs\_product表中, 然后同样的将相应的信息 添加到Solr库中.

**1 商品上架**

我们在这里点击上架按钮:

list.jsp:

<div style="margin-top:15px;"><input class="del-button" type="button" value="删除" onclick="optDelete();"/>

<input class="add" type="button" value="上架" onclick="isShow('${name}', '${brandId }', '${isShow }' ,'${pagination.pageNo }')"/>

<input class="del-button" type="button" value="下架" onclick="isHide();"/></div>

1 <script type="text/javascript">

2 //上架

3 function isShow(name,brandId,isShow,pageNo){

4 //请至少选择一个

5 var size = $("input[name='ids']:checked").size();

6 if(size == 0){

7 alert("请至少选择一个");

8 return;

9 }

10 //你确定上架吗

11 if(!confirm("你确定上架吗")){

12 return;

13 }

14 //提交 Form表单

15 $("#jvForm").attr("action","/product/isShow.do?name="+ name +"&brandId="+brandId+"&isShow="+isShow+"&pageNo="+pageNo);

16 $("#jvForm").attr("method","post");

17 $("#jvForm").submit();

18

19 }

20 </script>

**2 接着到Controller层:**

ProductController.java:

1 //添加页面

2 @RequestMapping("/isShow.do")

3 public String isShow(Long[] ids, Model model){

4 productService.isShow(ids);

5 return "forward:/product/list.do";

6 }

**3 接着看Service层:**

ProdcutServiceImpl.java:

1 //上架

　　@Autowired

private SolrServer solrServer;

2 public void isShow(Long[] ids){

3 Product product = new Product();

4 product.setIsShow(true);

5 for (Long id : ids) {

6 //上下架状态

7 product.setId(id);

8 productDao.updateByPrimaryKeySelective(product);

9

10 //TODO 保存商品信息到Solr服务器

11 SolrInputDocument doc = new SolrInputDocument();

12 //ID

13 doc.setField("id", id);

14 //名称

15 Product p = productDao.selectByPrimaryKey(id);

16 doc.setField("name\_ik", p.getName());

17 //图片URL

18 doc.setField("url", p.getImgUrls()[0]);

19 //品牌 ID

20 doc.setField("brandId", p.getBrandId());

21 //价格 sql查询语句: select price from bbs\_sku where product\_id = ? order by price asc limit 1

22 SkuQuery skuQuery = new SkuQuery();

23 skuQuery.createCriteria().andProductIdEqualTo(id);

24 skuQuery.setOrderByClause("price asc");

25 skuQuery.setPageNo(1);

26 skuQuery.setPageSize(1);

27 List<Sku> skus = skuDao.selectByExample(skuQuery);

28 doc.setField("price", skus.get(0).getPrice());

29 //...时间等 剩下的省略

30

31 try {

32 solrServer.add(doc);

33 solrServer.commit();

34 } catch (Exception e) {

35 // TODO Auto-generated catch block

36 e.printStackTrace();

37 }

38 //TODO 静态化

39 }

40 }

这里使用SolrInputDocument 来保存商品信息, 其中doc.setField("name\_ik", p.getName());的name\_ik 是我们在solr 配置文件配置的IK 分词器的字段, doc.setField("url", p.getImgUrls()[0]); 这里我们也只是取第一张图片的url用来展示.

这里我们还用到了skuQuery, 因为一个商品中不同的颜色不同的尺码都可能有不同的价格, 我们在这里 是取到同一个productId下价格最小的来给显示~

然后再就是将我们已经设置好的SolrInputDocument 通过SolrServer 来提交到Solr服务器. SolrServer是已经在spring中注册好了的, 在这里直接注入即可使用.

spring来管理Solr:

到了这里上架的功能就做好了, 这也是给后面Solr查询做好铺垫.

**4 前台使用Solr查询**

到了这里就开始查看前台页面了, 前台页面是扒的网上的, 具体业务逻辑是自己修改的, 页面如下:

这里需要特殊说明一下, 我们配置的全局拦截器变成了: / , 而且过滤掉静态资源, 配置如下:

首先是babasport-portal project下的web.xml文件:

View Code

第二个就是babasport-portal project下的spring配置文件中设置过滤掉静态资源:

1 <!-- 过滤静态资源 -->

2 <mvc:resources location="/js/" mapping="/js/\*.\*"/>

3 <mvc:resources location="/css/" mapping="/css/\*.\*"/>

4 <mvc:resources location="/images/" mapping="/images/\*.\*"/>

这样就就可以直接访问了.

当我们输入2016 点击查询后会出现什么? 我把已经做好的页面展示一下:

那么就进入到实际的开发当中:

当我们在搜索框输入2016 且点击 搜索时:

然后到Controller层去找到search方法:

1 @Autowired

2 private SearchService searchService;

3

4 //去首页

5 @RequestMapping(value="/")

6 public String index(){

7 return "index";

8 }

9

10 //搜索

11 @RequestMapping(value="/search")

12 public String search(Integer pageNo, String keyword, String price, Long brandId, Model model){

13 //品牌结果集 Redis中

14 List<Brand> brands = searchService.selectBrandListFromRedis();

15 model.addAttribute("brands", brands);

16

17 //map 装已经选择的条件

18 Map<String, String> map = new HashMap<String, String>();

19 if(null != brandId){

20 for (Brand brand : brands) {

21 if(brandId.equals(brand.getId())){

22 map.put("品牌", brand.getName());

23 break;

24 }

25 }

26 }

27 //价格 0-99 1600以上

28 if(null != price){

29 String[] split = price.split("-");

30 //如果切割后的长度等于2 就说明这是一个价格区间

31 if(split.length == 2){

32 map.put("价格", price);

33 }else {

34 map.put("价格", price + "以上");

35 }

36 }

37 model.addAttribute("map", map);

38

39 Pagination pagination = searchService.selectPaginationFromSolr(pageNo, keyword, price, brandId);

40 model.addAttribute("pagination", pagination);

41 model.addAttribute("keyword", keyword);

42 model.addAttribute("price", price);

43 model.addAttribute("brandId", brandId);

44

45 return "search";

46 }

提示: 这里使用到了SolrService, 相信看我以前博文的朋友都知道这个地方还需要配置dubbo, 就是服务提供方和适用方, 这里为了简便直接略过, 实际开发中是必须要配置的, 否则就调用不了SolrService中的方法了.

这个controller 中往search.jsp中put了很多东西, 具体这些东西什么用我们可以先不管, 我们先看下search.jsp页面.

而且这个controller中查询brand 是从redis中查询出来的, 我们会在下面讲到这个.

1 <c:if test="${fn:length(map) != 0 }">

2 <div class="sl-b-selected J\_brandSelected">

3 <span class="crumbs-arrow">已选条件：</span>

4 <c:forEach items="${map }" var="m">

5 <a title="依琦莲（yiqilian）" href="javascript:;" class="crumb-select-item">

6 <b>${m.key }：</b><em>${m.value }</em><i></i>

7 </a>

8 </c:forEach>

9 </div>

10 </c:if>

上面这个地方就是为何要在controller设置map值了, 这个是显示已选择的过滤条件.

1 <c:if test="${empty brandId }">

2 <div class="J\_selectorLine s-brand">

3 <div class="sl-wrap">

4 <div class="sl-key"><strong>品牌：</strong></div>

5 <div class="sl-value">

6 <div class="sl-v-list">

7 <ul class="J\_valueList v-fixed">

8 <c:forEach items="${brands }" var="brand">

9 <li id="brand-38118" data-initial="j" style="display:block;">

10 <a href="javascript:;" onclick="fqBrand('${brand.id }')" title="${brand.name }"><i></i>${brand.name }</a>

11 </li>

12 </c:forEach>

13 </ul>

14 </div>

15 </div>

16 </div>

17 </div>

18 </c:if>

19 <c:if test="${empty price }">

20 <div id="J\_selectorPrice" class="J\_selectorLine s-line">

21 <div class="sl-wrap">

22 <div class="sl-key"><span>价格：</span></div>

23 <div class="sl-value">

24 <div class="sl-v-list">

25 <ul class="J\_valueList">

26 <li>

27 <a href="javascript:;" onclick="fqPrice('0-99')"><i></i>0-99</a>

28 </li>

29 <li>

30 <a href="javascript:;" onclick="fqPrice('100-299')"><i></i>100-299</a>

31 </li>

32 <li>

33 <a href="javascript:;" onclick="fqPrice('300-599')"><i></i>300-599</a>

34 </li>

35 <li>

36 <a href="javascript:;" onclick="fqPrice('600-999')"><i></i>600-999</a>

37 </li>

38 <li>

39 <a href="javascript:;" onclick="fqPrice('1000-1599')"><i></i>1000-1599</a>

40 </li>

41 <li>

42 <a href="javascript:;" onclick="fqPrice('1600')"><i></i>1600以上</a>

43 </li>

44 </ul>

45 </div>

46 </div>

47 </div>

48 </div>

49 </c:if>

接下来我们来看下对应的js方法:

1 <script type="text/javascript">

2 var price = '${price}';

3 var brandId = '${brandId}';

4 //过滤品牌id

5 function fqBrand(id){

6 if('' != price){

7 window.location.href="/search?keyword="+ ${keyword} + "&brandId="+ id+"&price="+price;

8 }else{

9 window.location.href="/search?keyword="+ ${keyword} + "&brandId="+ id;

10 }

11 }

12

13 //过滤价格

14 function fqPrice(id){

15 if('' != brandId){

16 window.location.href = "/search?keyword=${keyword}" + "&brandId=" + brandId + "&price=" + id;

17 }else{

18 window.location.href = "/search?keyword=${keyword}" + "&price=" + id;

19 }

20 }

21 </script>

这个就可以实现 添加 过滤条件的选项了.

**5 使用Redis 取出商品品牌列表**

首先 当我们在后台添加或者修改品牌时, 我们应该同样将这个品牌添加到Redis中, 格式类似于: {"brandId":"brandName"}

controller层:(当我们在后台添加或者修改品牌)

1 @Autowired

2 private Jedis jedis;

3 //修改

4 public void updateBrandById(Brand brand){

5 //保存或修改 时修改Redis中的品牌, hmset适合批量添加品牌

6 /\*Map<String, String> map = new HashMap<String,String>();

7 map.put(String.valueOf(brand.getId()), brand.getName());

8 jedis.hmset("brand", map);\*/

9 jedis.hset("brand", String.valueOf(brand.getId()), brand.getName());

10 brandDao.updateBrandById(brand);

11 }

**6 redis中有了品牌列表后, 然后就是查询了:**

1 @Autowired

2 private Jedis jedis;

3 //查询Redis中的品牌结果集

4 public List<Brand> selectBrandListFromRedis(){

5 List<Brand> brands = new ArrayList<Brand>();

6 Map<String, String> hgetAll = jedis.hgetAll("brand");

7 Set<Entry<String, String>> entrySet = hgetAll.entrySet();

8 for (Entry<String, String> entry : entrySet) {

9 Brand brand = new Brand();

10 brand.setId(Long.parseLong(entry.getKey()));

11 brand.setName(entry.getValue());

12 brands.add(brand);

13 }

14

15 return brands;

16 }

**7 到了这里redis查询brand就完成了, 那么继续看下关于solr 是如何加入过滤条件的吧:**

1 @Autowired

2 private SolrServer solrServer;

3 //查询商品信息从Solr

4 public Pagination selectPaginationFromSolr(Integer pageNo, String keyword, String price, Long brandId){

5 ProductQuery productQuery = new ProductQuery();

6 //当前页

7 productQuery.setPageNo(Pagination.cpn(pageNo));

8 //每页数

9 productQuery.setPageSize(8);

10

11 SolrQuery solrQuery = new SolrQuery();

12 //关键词 商品名称

13 solrQuery.set("q", "name\_ik:"+keyword);

14 //回显数据

15 StringBuilder params = new StringBuilder();

16 params.append("keyword=").append(keyword);

17

18 //排序

19 solrQuery.addSort("price", ORDER.asc);

20

21 //高亮

22 //1,设置, 打开高亮的开关

23 solrQuery.setHighlight(true);

24 //2, 设置高亮字段

25 solrQuery.addHighlightField("name\_ik");

26 //3, 设置关键字高亮的样式 <span style='color:red'>2016</span>

27 //设置前缀和后缀

28 solrQuery.setHighlightSimplePre("<span style='color:red'>");

29 solrQuery.setHighlightSimplePost("</span>");

30

31 //过滤条件 品牌

32 if(null != brandId){

33 solrQuery.addFilterQuery("brandId:"+brandId);

34 params.append("&brandId=").append(brandId);

35 }

36 //过滤价格 0-99 1600

37 if(null != price){

38 String[] split = price.split("-");

39 //如果切割后的长度等于2 就说明这是一个价格区间

40 if(split.length == 2){

41 solrQuery.addFilterQuery("price:["+split[0]+" TO "+split[1]+"]");

42 }else {

43 solrQuery.addFilterQuery("price:["+split[0]+" TO \*]");

44 }

45 params.append("&price=").append(price);

46 }

47

48 //分页 limit 开始行,每页数

49 solrQuery.setStart(productQuery.getStartRow());

50 solrQuery.setRows(productQuery.getPageSize());

51

52 QueryResponse response = null;

53 try {

54 response = solrServer.query(solrQuery);

55

56 } catch (Exception e) {

57 e.printStackTrace();

58 }

59 //分析这个Map

60 //第一层Map: Key String == ID : Value: Map

61 //第二层Map: Key String == name\_ik : Value: List

62 //获取到List: String 0,1,2....

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

64

65

66 List<Product> products = new ArrayList<Product>();

67 //结果集

68 SolrDocumentList docs = response.getResults();

69 //总条数

70 long numFound = docs.getNumFound();

71 for (SolrDocument doc : docs) {

72 Product product = new Product();

73 //商品的ID

74 String id = (String)doc.get("id");

75 product.setId(Long.parseLong(id));

76

77 //取第二层Map

78 Map<String, List<String>> map = highlighting.get(id);

79 //取List集合

80 List<String> list = map.get("name\_ik");

81

82 //商品名称

83 //String name = (String)doc.get("name\_ik");

84 //product.setName(name);

85 product.setName(list.get(0)); //list.get(0) 中的name是已经设置为高亮的

86

87 //图片

88 String url = (String)doc.get("url");

89 product.setImgUrl(url);

90 //价格 这里的价格本身是保存在bbs\_sku表中的, 而我们在这里将price属性直接添加到了Product中

91 //因为我们在做上架的时候, 查询的是bbs\_sku中price最小的值 然后保存到solr中的, 所以这里我们就直接将price属性添加到product中了

92 //这里的价格只有一个值

93 //Float price = (Float)doc.get("price");

94 product.setPrice((Float)doc.get("price"));

95 //品牌ID

96 //Integer brandId = (Integer)doc.get("brandId");

97 product.setBrandId(Long.parseLong(String.valueOf((Integer)doc.get("brandId"))));

98 products.add(product);

99 }

100

101 Pagination pagination = new Pagination(

102 productQuery.getPageNo(),

103 productQuery.getPageSize(),

104 (int)numFound,

105 products

106 );

107 //页面展示

108 String url = "/search";

109 pagination.pageView(url, params.toString());

110

111 return pagination;

112 }