课程目标

- 了解什么是全文检索技术？
  - 想明白字典的出现是为了什么？
- 全文检索技术可以用来做什么？
  - 搜索引擎：百度、谷歌、搜狗等
  - **站内搜索**：小说网站、电商网站、论坛等等
  - 文件系统搜索：Windows文件系统搜索
- 有哪些主流的Java全文检索技术？
  - Lucene：这是Java语言全局检索技术的底层实现（开山鼻祖）
  - Solr：基于Lucene，简化开发，提示性能、扩展性。通过SolrCloud可以实现分布式搜索
  - ElasticSearch（ES）：基于Lucene，更倾向于实现实时搜索。
- 这些技术应该如何选择？
  - 需要搞清楚每个技术的特点及缺点。
- 分别学习不同的全文检索技术
  - 是什么？---为了沟通
  - 安装和配置
  - 使用（Java开发）

# 一、全文检索技术

## 什么是全文检索？

什么叫做全文检索呢？这要从我们生活中的数据说起。

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 互联网数据、邮件，word文档等。

**非结构化数据又一种叫法叫全文数据。**

按照数据的分类，搜索也分为两种：

- **对结构化数据的搜索：** 如对数据库的搜索，用SQL语句。再如对元数据的搜索，如利用windows搜索对文件名，类型，修改时间进行搜索等。
- **对非结构化数据的搜索：** 如用Google和百度可以搜索大量内容数据。

对非结构化数据也即全文数据的搜索主要有两种方法：**顺序扫描法和反向索引法**。

- **顺序扫描法**：所谓顺序扫描法，就是顺序扫描每个文档内容，看看是否有要搜索的关键字，实现查找文档的功能，也就是根据文档找词。
- **反向索引法**：所谓反向索引，就是提前将搜索的关键字建成索引，然后再根据索引查找文档，也就是根据词找文档。

**这种先建立`索引`，再对索引进行`搜索`文档的过程就叫`全文检索(Full-text Search)`。**

## 全文检索场景

- 搜索引擎
- 站内搜索
- 系统文件搜索

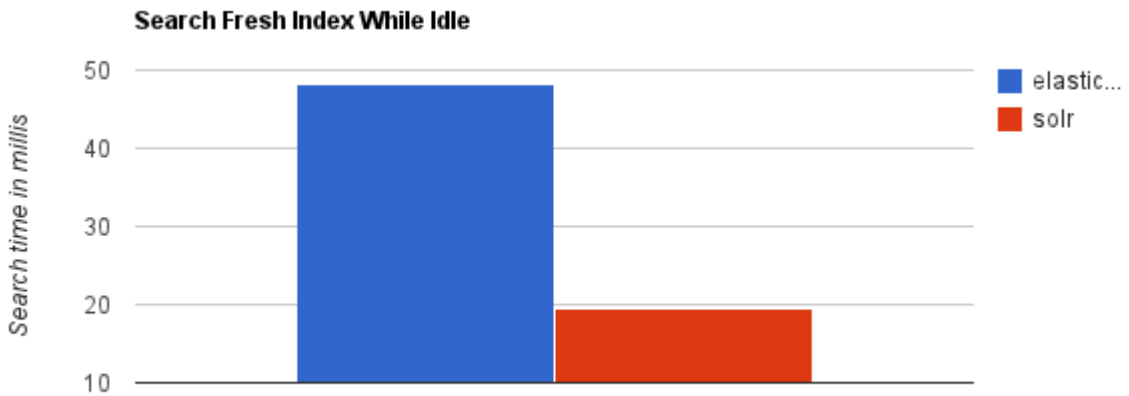
## 全文检索相关技术

1. Lucene：如果使用该技术实现，需要对Lucene的API和底层原理非常了解，而且需要编写大量的Java代码。
2. Solr：使用java实现的一个web应用，可以使用rest方式的http请求，进行远程API的调用。
3. ElasticSearch(ES)：可以使用rest方式的http请求，进行远程API的调用。

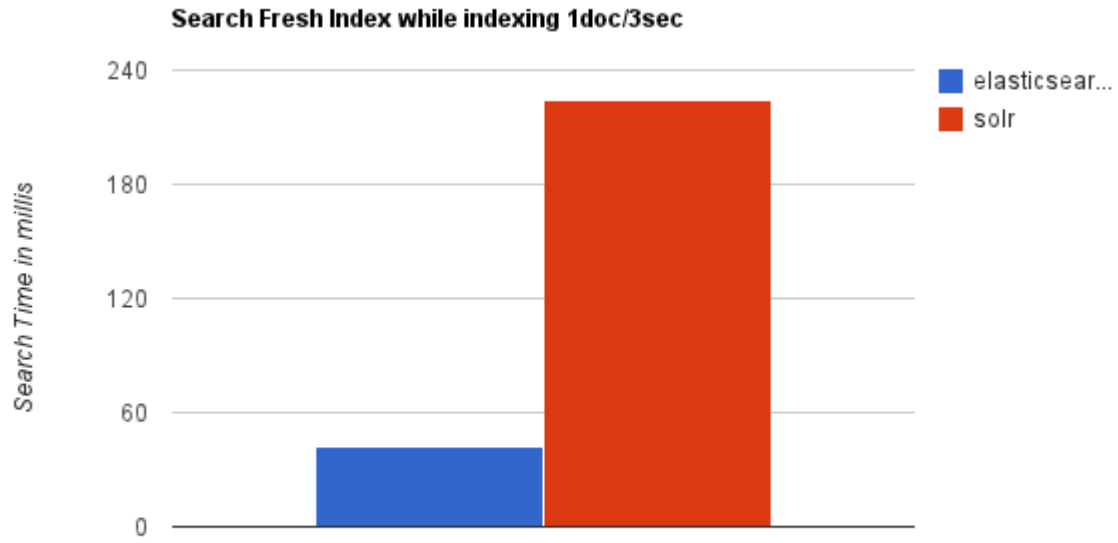
# 二、Solr和ES的比较

## ElasticSearch vs Solr 检索速度

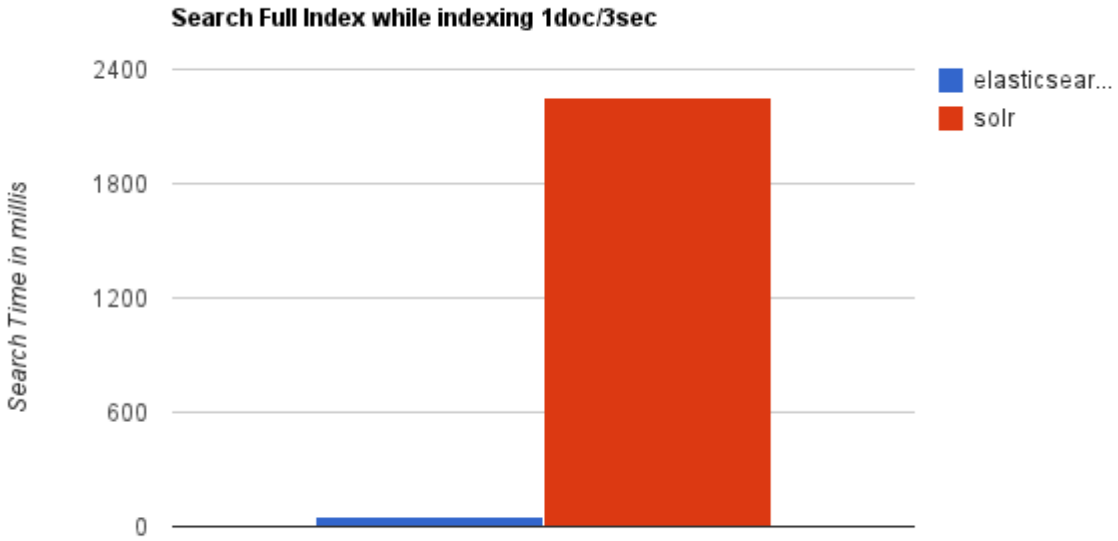
- 当单纯的对**已有数据**进行搜索时，Solr更快。



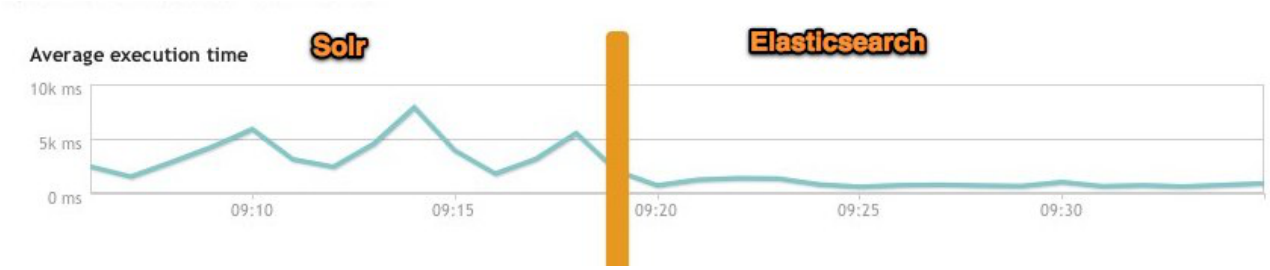

- **当实时建立索引时, Solr会产生io阻塞，查询性能较差, Elasticsearch具有明显的优势。**




- **随着数据量的增加，Solr的搜索效率会变得更低，而Elasticsearch却没有明显的变化。**




- **大型互联网公司，实际生产环境测试，将搜索引擎从Solr转到Elasticsearch以后的平均查询速度有了50倍的提升。**




## Elasticsearch 与 Solr 的比较总结

- 二者安装都很简单；
- Solr 利用 Zookeeper 进行分布式管理，而 Elasticsearch 自身带有分布式协调管理功能;
- Solr 支持更多格式的数据，而 Elasticsearch 仅支持json文件格式；
- Solr 官方提供的功能更多，而 Elasticsearch 本身更注重于核心功能，高级功能多有第三方插件提供；
- Solr 在传统的搜索应用中表现好于 Elasticsearch，但在处理实时搜索应用时效率明显低于 Elasticsearch。

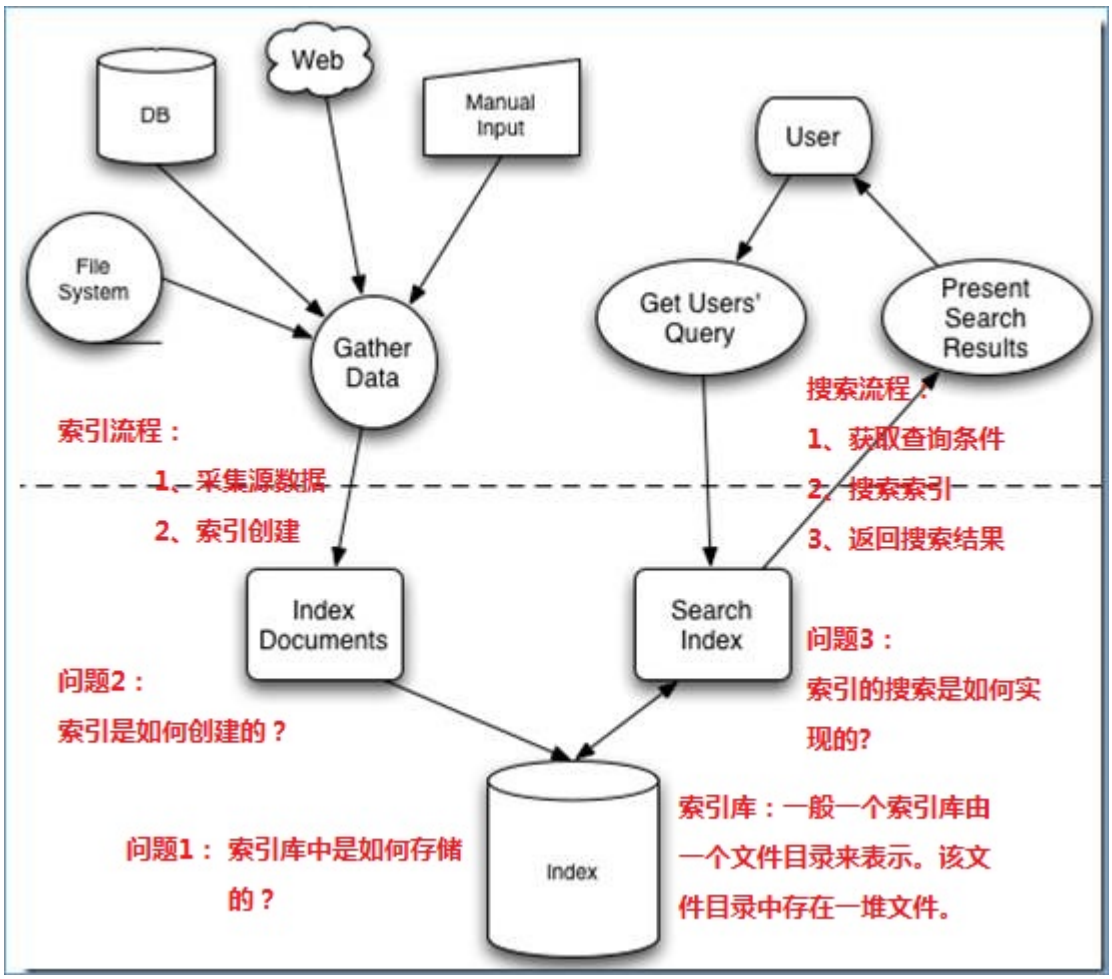
> **最终的结论：**
>
> Solr 是传统搜索应用的有力解决方案，但 Elasticsearch 更适用于新兴的实时搜索应用。

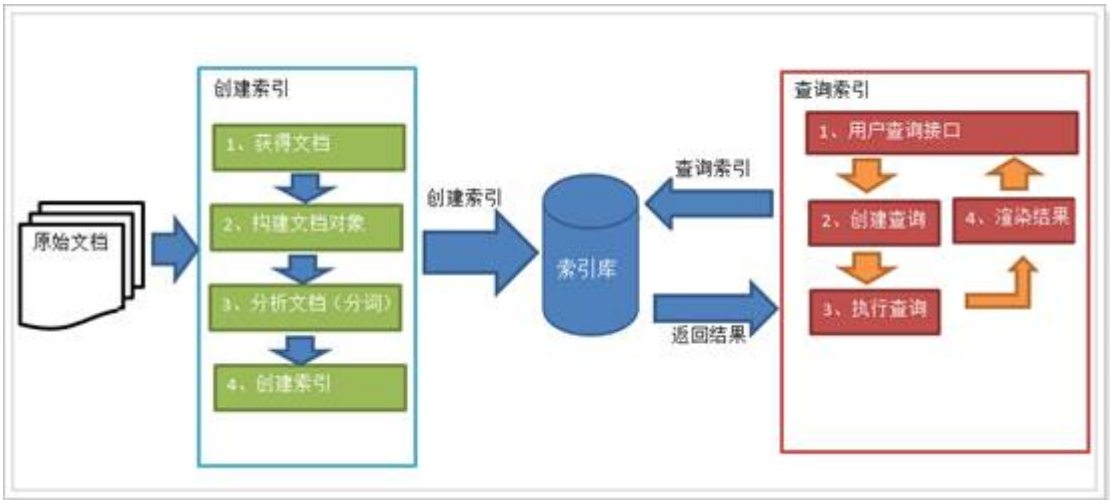
## 实时搜索与传统搜索

> 通常来说，**传统搜索**都是一些“静态”的搜索，即用户搜索的只是从信息库里边筛选出来的信息。而百度推出的**实时搜索功能**，改变了传统意义上的静态搜索模式，用户对于搜索的结果是实时变化的。
>
> 举个例子，用户在搜索“华山”、“峨眉山”等景点时，实时观看各地景区画面。以华山景区为例，当用户在搜索框中输入“华山”时，点击右侧“实时直播——华山”，即可实时观看华山靓丽风景，并能在华山长空栈道、北峰顶、观日台三个视角之间切换。同时，该直播引入广受年轻人欢迎的“弹幕”模式，用户在观看风景时可以同时发表评论，甚至进行聊天互动。

# 三、全文检索的流程分析

## 2.1 流程总览




**全文检索的流程分为两大流程：索引创建、搜索索引**

- 索引创建：将现实世界中所有的结构化和非结构化数据提取信息，创建索引的过程。
- 搜索索引：就是得到用户的查询请求，搜索创建的索引，然后返回结果的过程。

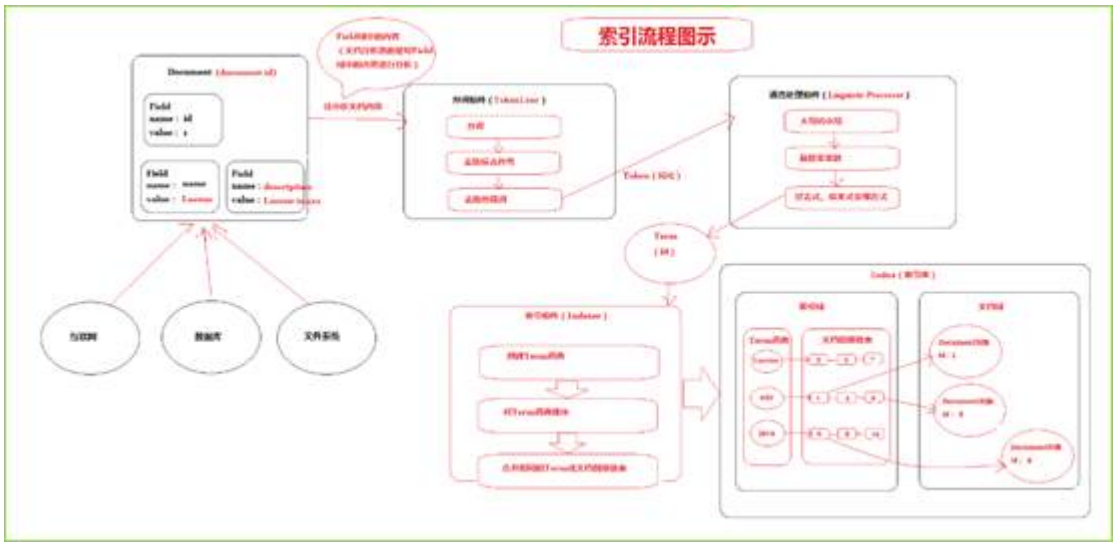
**想搞清楚全文检索，必须要搞清楚下面三个问题：**

**1. 索引库里面究竟存些什么？(Index)**

**2. 如何创建索引？(Indexing)**

**3. 如何对索引进行搜索？(Search)**

# 2.2 创建索引流程




**一次索引，多次使用。**

## 2.2.1 原始内容

**原始内容是指要索引和搜索的内容。**

原始内容包括互联网上的网页、数据库中的数据、磁盘上的文件等。

## 2.2.2 获得文档

也就是采集数据，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。**

**采集数据的目的是为了将原始内容存储到Document对象中。**

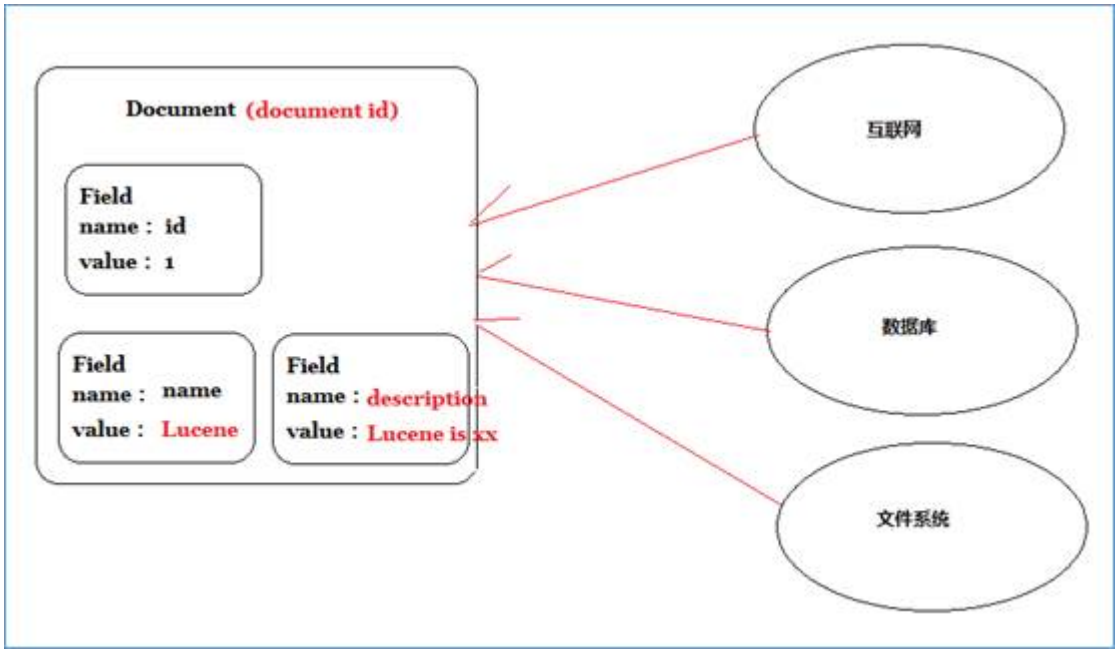
**如何采集数据？**

1. 对于互联网上网页，可以使用工具将网页抓取到本地生成html文件。
2. 数据库中的数据，可以直接连接数据库读取表中的数据。
3. 文件系统中的某个文件，可以通过I/O操作读取文件的内容。

> 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。

## 2.2.3 创建文档

**创建文档的目的是统一数据格式（Document），方便文档分析。**

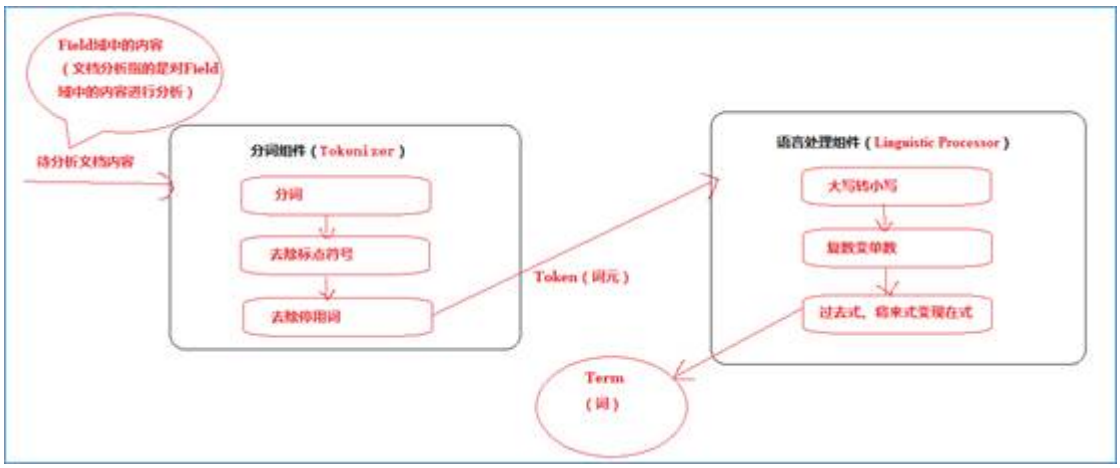



**说明：**

1. 一个Document文档中包括多个域（Field），域（Field）中存储内容。
2. 这里我们可以将数据库中一条记录当成一个Document，一列当成一个Field。

## 2.2.4 分析文档（重点）

**分析文档主要是对Field域进行分析，分析文档的目的是为了索引。**




**说明：**分析文档主要通过**分词组件（Tokenizer）**和**语言处理组件（Linguistic Processor）**完成。

### 分词组件

**分词组件工作流程（此过程称之为Tokenize）**

1. 将Field域中的内容进行分词（不同语言有不同的分词规则）。
2. 去除标点符号。
3. 去除停用词（stop word）。

**经过分词（Tokenize）之后得到的结果成为`词元（Token）`。**

> **所谓停词(Stop word)**就是一种语言中最普通的一些单词，由于没有特别的意义，因而大多数情况下不能成为搜索的关键词，因而创建索引时，这种词会被去掉而减少索引的大小。
>
> 英语中停词(Stop word)如：“the”,“a”，“this”等。
>
> **对于每一种语言的分词组件(Tokenizer)，都有一个停词(stop word)集合。**

**示例(Document1的Field域和Document2的Field域是同名的)：**

- Document1的Field域：

```
Students should be allowed to go out with their   friends, but not allowed to drink beer.
```

- Document2的Field域：

```
My friend Jerry went to school to see his students but   found them drunk which is not allowed.
```

- 在我们的例子中，便得到以下**词元(Token)**：

```
“Students”，“allowed”，“go”，“their”，“friends”，“allowed”，“drink”，“beer”，“My”，“friend”，“Jerry”，“went”，“school”，“see”，“his”，“students”，“found”，“them”，“drunk”，“allowed”。
```

**将得到的词元(Token)传给语言处理组件(Linguistic Processor)。**

### 语言处理组件

*语言处理组件**(linguistic processor)**主要是对得到的词元**(Token)**做一些同语言相关的处理。*

**对于英语，语言处理组件(Linguistic Processor)一般做以下几点：**

1. 变为小写(Lowercase)。
2. 将单词缩减为词根形式，如“cars”到“car”等。这种操作称为：stemming。
3. 将单词转变为词根形式，如“drove”到“drive”等。这种操作称为：lemmatization。

**语言处理组件(linguistic processor)的结果称为`词(Term)`。Term是索引库的最小单位。**

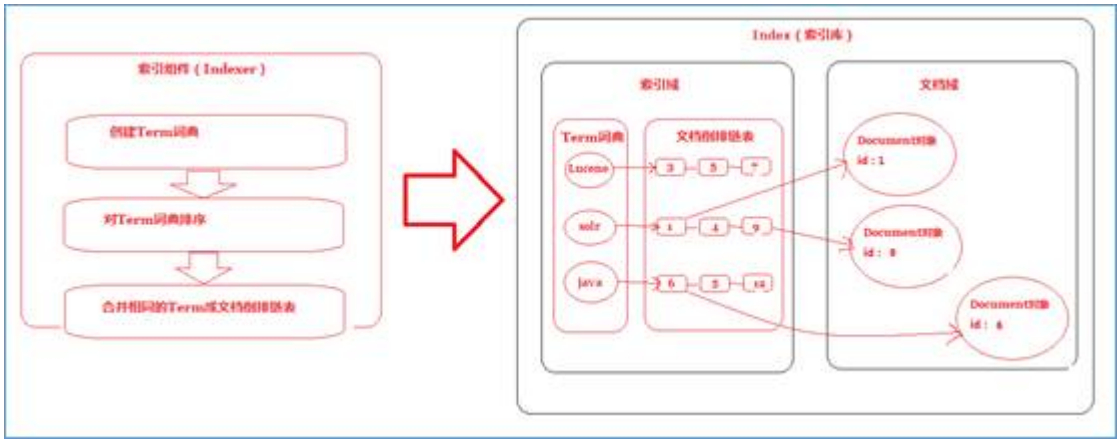
- 在我们的例子中，经过语言处理，得到的词**(Term)**如下：

```
“student”，“allow”，“go”，“their”，“friend”，“allow”，“drink”，“beer”，“my”，“friend”，“jerry”，“go”，“school”，“see”，“his”，“student”，“find”，“them”，“drink”，“allow”。
```

也正是因为有语言处理的步骤，才能使搜索drove，而drive也能被搜索出来。

## 2.2.5 索引文档

**索引的目的是为了搜索。**




**说明：将得到的词(Term)传给索引组件(Indexer)，索引组件(Indexer)主要做以下几件事情：**

## 创建Term字典

在我们的例子中字典如下：

| Term | Document ID |
| --- | --- |
| Student | 1 |
| Allow | 1 |
| Go | 1 |
| Their | 1 |
| Friend | 1 |
| Allow | 1 |
| Drink | 1 |
| Beer | 1 |
| My | 2 |
| Friend | 2 |
| Jerry | 2 |
| Go | 2 |
| School | 2 |
| See | 2 |
| His | 2 |
| Student | 2 |
| Find | 2 |
| Them | 2 |
| Drink | 2 |
| Allow | 2 |

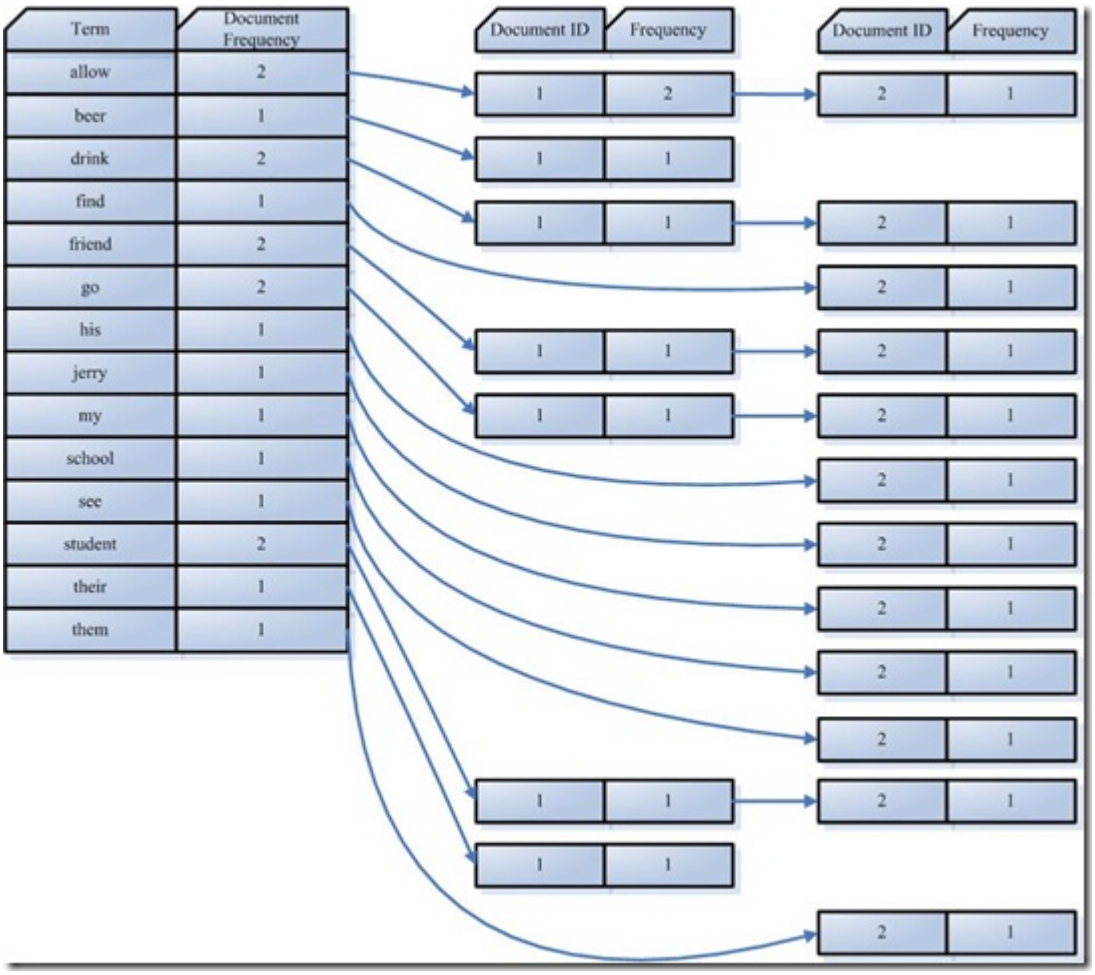
### 排序Term字典

对字典按字母顺序进行排序

| Term | Document ID |
| --- | --- |
| Allow | 1 |
| Allow | 1 |
| Allow | 2 |
| Beer | 1 |
| Drink | 1 |
| Drink | 2 |
| Find | 2 |
| Friend | 1 |
| Friend | 2 |
| Go | 1 |
| Go | 2 |
| His | 2 |
| Jerry | 2 |
| My | 2 |
| School | 2 |
| See | 2 |
| Student | 1 |
| Student | 2 |
| Their | 1 |
| Them | 2 |

## 合并Term字典

合并相同的词(Term)成为文档倒排(Posting List)链表



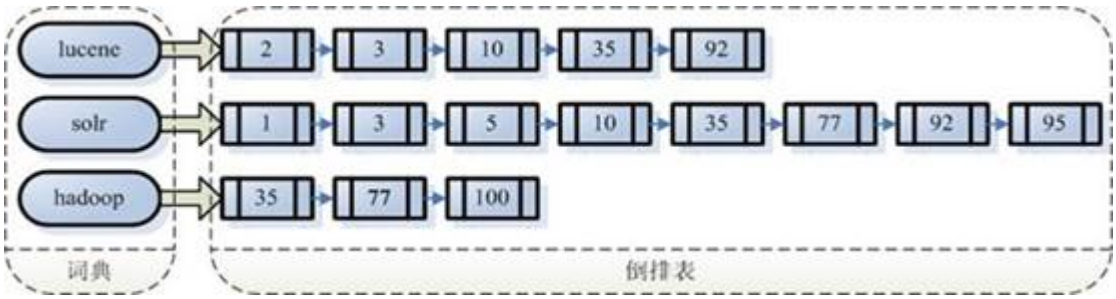

在此表中，有几个定义：

- **Document Frequency** 即文档频次，表示总共有多少文件包含此词(Term)。
- **Frequency** 即词频率，表示此文件中包含了几个此词(Term)。

**到此为止，索引已经创建好了。**

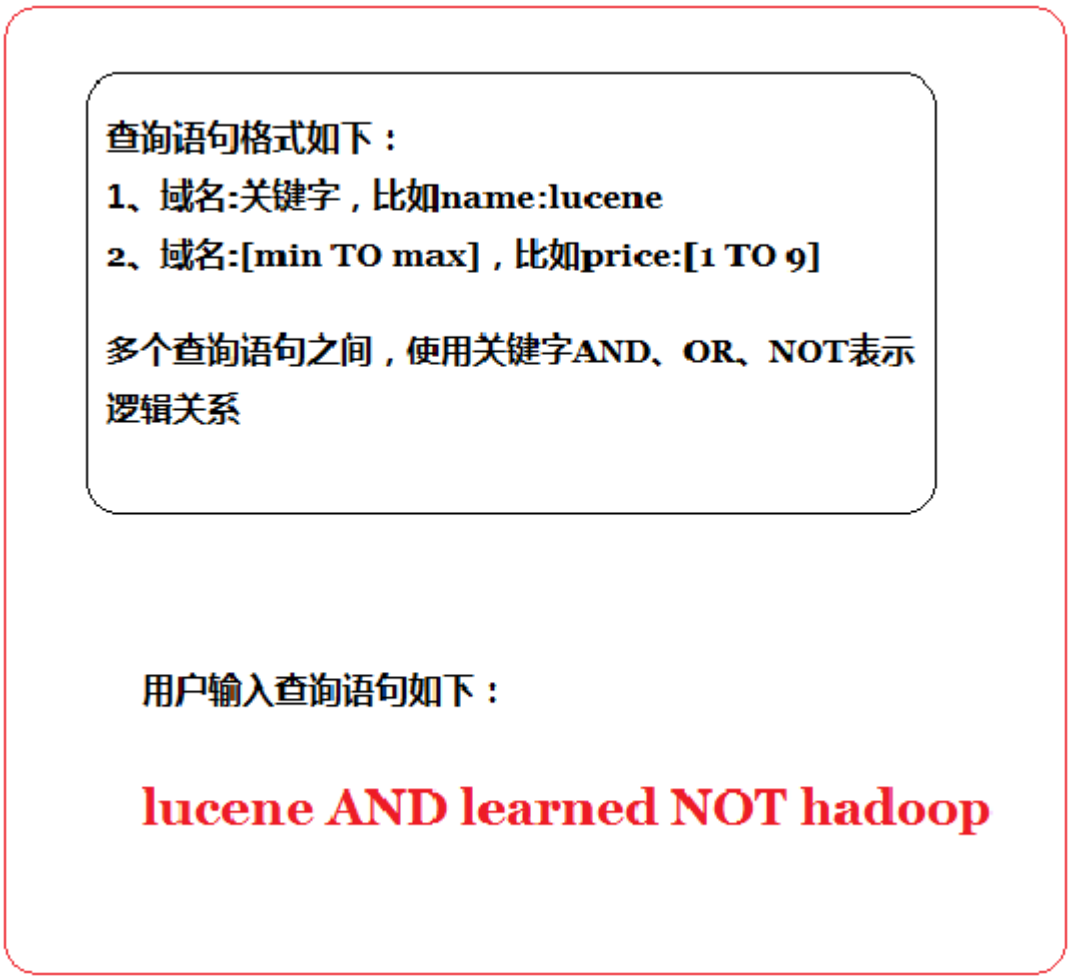
**最终的索引结构是一种倒排索引结构也叫反向索引结构，包括索引和文档两部分，索引即词汇表，它的规模较小，而文档集合较大。**

**倒排索引结构**是根据内容（词汇）找文档，如下图：




# 2.3 搜索索引流程

## 2.3.1 图1：查询语句



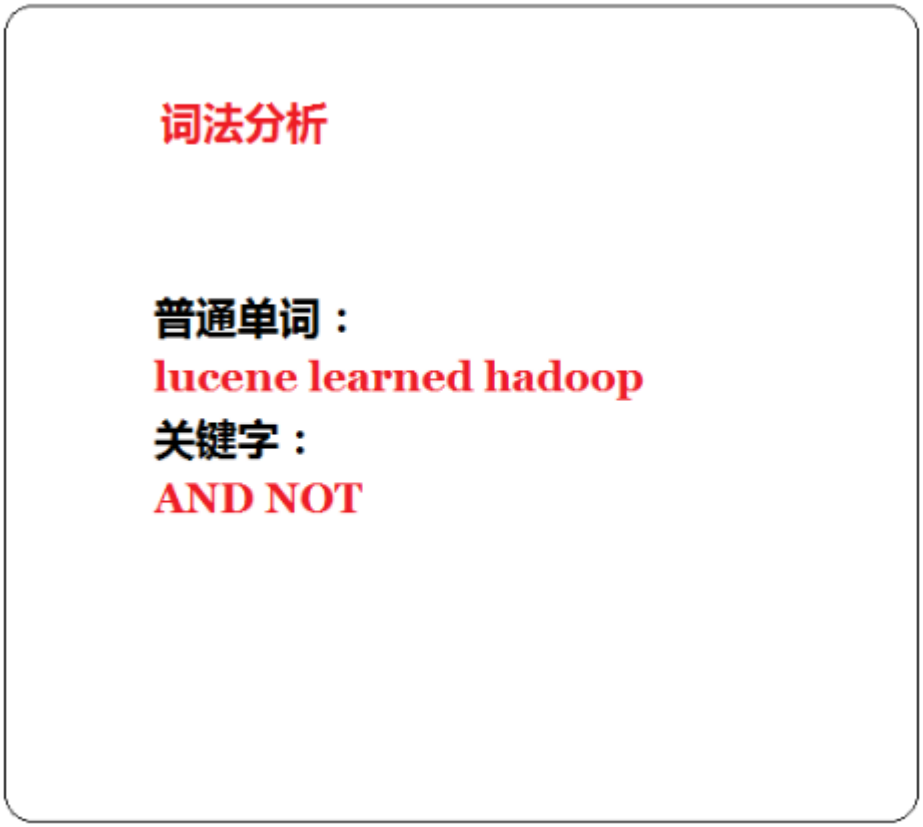

### 2.4.2 图2：执行搜索

**第一步：对查询语句进行词法分析、语法分析及语言处理。**

**1、词法分析**

如上述例子中，经过词法分析，得到单词有lucene，learned，hadoop, 关键字有AND, NOT。

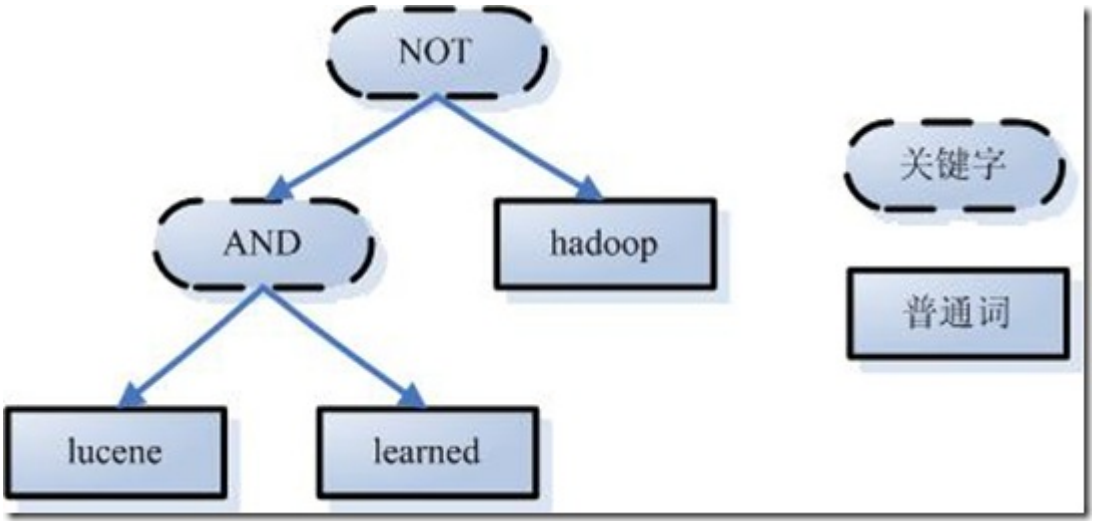
***注意：关键字必须大写，否则就作为普通单词处理。***




**2、语法分析**

如果发现查询语句不满足语法规则，则会报错。如lucene NOT AND learned，则会出错。

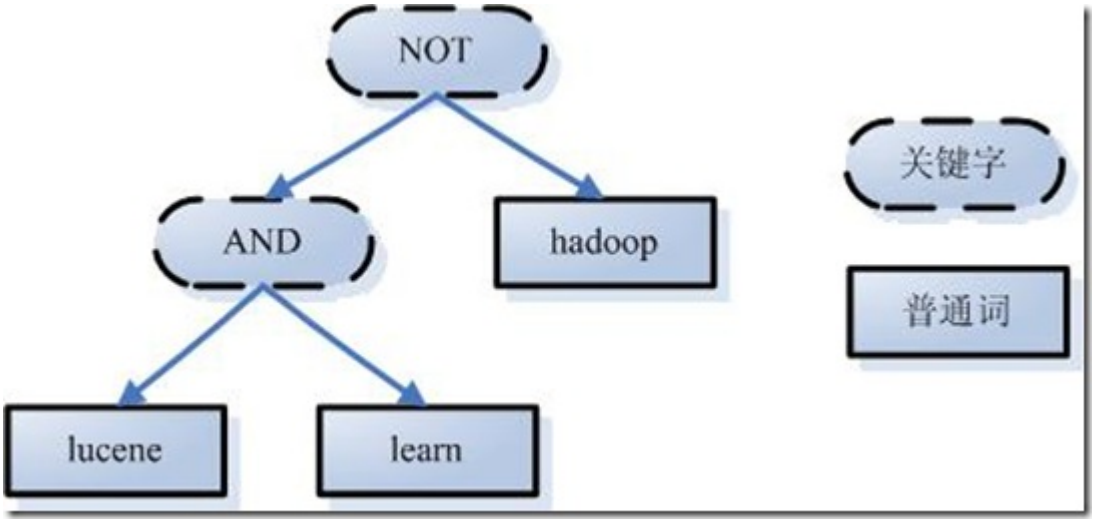
如上述例子，lucene AND learned NOT hadoop形成的语法树如下：




**3、语言处理**

如learned变成learn等。

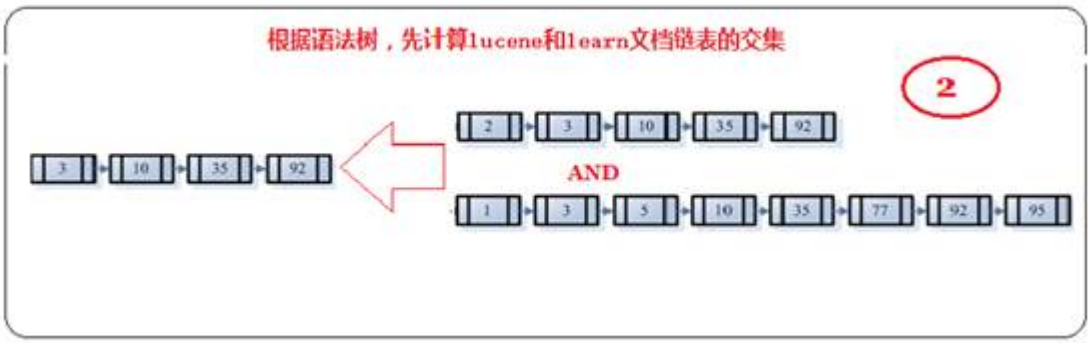
经过第二步，我们得到一棵经过语言处理的语法树。



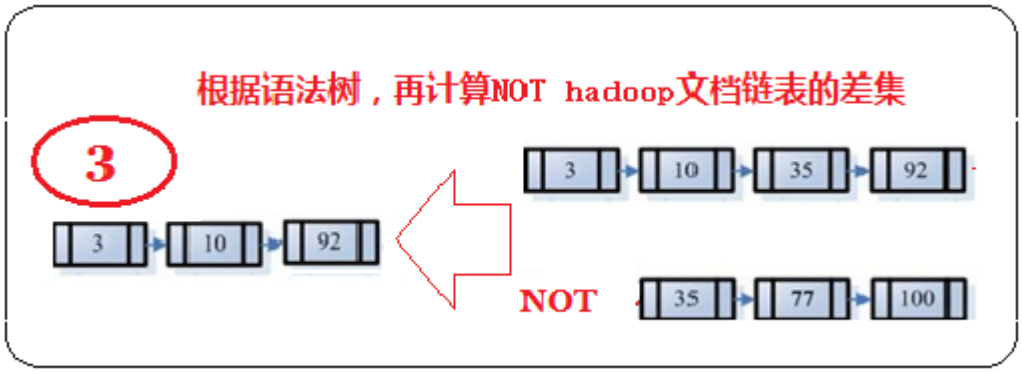

**第二步：搜索索引，得到符号语法树的文档。**

1、 首先，在反向索引表中，分别找出包含lucene，learn，hadoop的文档链表。

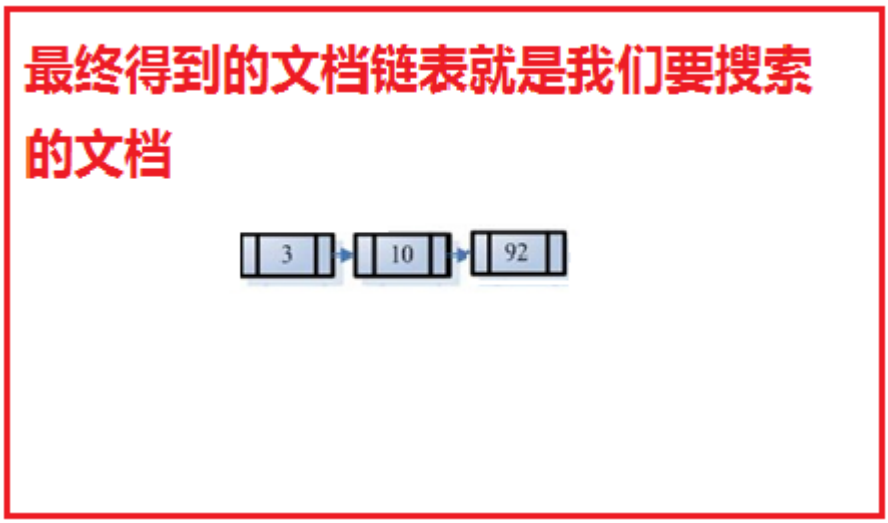
2、 其次，对包含lucene，learn的链表进行合并操作，得到既包含lucene又包含learn的文档链表。


3、 然后，将此链表与hadoop的文档链表进行差操作，去除包含hadoop的文档，从而得到既包含lucene又包含learn而且不包含hadoop的文档链表。




4、 此文档链表就是我们要找的文档。




**第三步：根据得到的文档和查询语句的相关性，对结果进行排序。**

相关度自然打分（权重越高分越高）：

tf越高、权重越高

df越高、权重越低

人为影响分数：

设置Boost值（**加权值**）

## 2.4.3 Lucene相关度排序

### 2.4.3.1 什么是相关度排序

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

### 2.4.3.2 相关度打分

Lucene对查询关键字和索引文档的相关度进行打分，得分高的就排在前边。

如何打分呢？Lucene是在用户进行检索时实时根据搜索的关键字计算出来的，分两步：

1. 计算出词（Term）的权重
2. 根据词的权重值，计算文档相关度得分。

**什么是词的权重？**

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

- **Term Frequency (tf)：**

指此Term在此文档中出现了多少次。**tf 越大说明越重要。**

词(Term)在文档中出现的次数越多，说明此词(Term)对该文档越重要，如“Lucene”这个词，在文档中出现的次数很多，说明该文档主要就是讲Lucene技术的。

- **Document Frequency (df)：**

指有多少文档包含此Term。**df 越大说明越不重要。**

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

### 2.4.3.3 设置boost值影响相关度排序

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

# 四、Lucene应用代码

## 所需依赖

```
    <dependencies>
        <dependency>
            <groupId>org.apache.lucene</groupId>
            <artifactId>lucene-queryparser</artifactId>
            <version>7.5.0</version>
        </dependency>
        <dependency>
            <groupId>org.apache.lucene</groupId>
            <artifactId>lucene-analyzers-common</artifactId>
            <version>7.5.0</version>
        </dependency>

        <!-- 目的是为了数据采集 -->
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>5.1.35</version>
        </dependency>


        <!-- 可以自己安装，也可以使用中央仓库 -->
```

```xml
        <dependency>
            <groupId>com.janeluo</groupId>
            <artifactId>ikanalyzer</artifactId>
            <version>2012_u6</version>
        </dependency>

    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <configuration>
                    <source>1.8</source>
                    <target>1.8</target>
                </configuration>
            </plugin>
        </plugins>
    </build>
```

## 索引流程代码

```java
public class IndexDemo {

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

        // 1. 数据采集
        ItemDao itemDao = new ItemDaoImpl();
        List<Item> itemList = itemDao.queryItemList();

        // 2. 创建Document文档对象
        List<Document> documents = new ArrayList<>();
        for (Item item : itemList) {
            Document document = new Document();

            // Document文档中添加Field域
            // 商品Id
            // Store.YES:表示存储到文档域中
            document.add(new TextField("id", item.getId().toString(), Store.YES));
            // 商品名称
            document.add(new TextField("name", item.getName().toString(),
Store.YES));
            // 商品价格
            document.add(new TextField("price", item.getPrice().toString(),
Store.YES));
            // 商品图片地址
            document.add(new TextField("pic", item.getPic().toString(), Store.YES));
            // 商品描述
```

```
            document.add(new TextField("description",
item.getDescription().toString(), Store.YES));

            // 把Document放到list中
            documents.add(document);
        }

        // 指定分词器：标准分词器（此处可以改为中文分词器）
        // Analyzer analyzer = new StandardAnalyzer();
        Analyzer analyzer = new IKAnalyzer();
        // 配置文件
        IndexWriterConfig iwc = new IndexWriterConfig(analyzer);

        // 指定索引库路径
        String indexPath = "E:\\11-index\\vip01\\";
        // 指定索引库对象
        Directory dir = FSDirectory.open(Paths.get(indexPath));
        // 创建索引写对象
        IndexWriter writer = new IndexWriter(dir, iwc);
        // 3. 分词并创建索引文件
        writer.addDocuments(documents);

        // 释放资源
        writer.close();

    }
}
```

# Luke工具

Luke作为Lucene工具包中的一个工具，可以通过界面来进行索引文件的查询、修改。

# 搜索流程代码

```
public class SearchDemo {

    public static void main(String[] args) throws Exception {
        // 指定索引库路径
        String indexPath = "E:\\11-index\\vip01\\";
        // 指定索引库对象
        Directory dir = FSDirectory.open(Paths.get(indexPath));

        // 索引读对象
        IndexReader reader = DirectoryReader.open(dir);
        // 索引搜索器
        IndexSearcher searcher = new IndexSearcher(reader);

        Analyzer analyzer = new StandardAnalyzer();
        // 通过QueryParser解析查询语法，获取Query对象
        QueryParser parser = new QueryParser("description", analyzer);
```

```
        // 参数是查询语法
        Query query = parser.parse("lucene");
        TopDocs topDocs = searcher.search(query, 100);

        ScoreDoc[] scoreDocs = topDocs.scoreDocs;

        for (ScoreDoc scoreDoc : scoreDocs) {
            Document document = searcher.doc(scoreDoc.doc);
            System.out.println("name : "+document.get("name"));
        }

        reader.close();
    }

}
```

# 五、Lucene的Field域

## 1.1. Field属性

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

- **是否分词(tokenized)**
  - 是：作分词处理，即将Field值进行分词，分词的目的是为了索引。

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

    比如：商品id、订单号、身份证号等
- **是否索引(indexed)**
  - 是：进行索引。将Field分词后的词或整个Field值进行索引，存储到索引域，索引的目的是为了搜索。

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

    比如：图片路径、文件路径等，不用作为查询条件的不用索引
- **是否存储(stored)**
  - 是：将Field值存储在文档域中，存储在文档域中的Field才可以从Document中获取。

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

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

## 1.2. Field常用类型

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

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

# 1.3. Field设计

**Field域如何设计，取决于需求，比如搜索条件有哪些？显示结果有哪些？**

- 商品id：

  是否分词：不用分词，因为不会根据商品id来搜索商品

  是否索引：不索引，因为不需要根据商品ID进行搜索

  是否存储：要存储，因为查询结果页面需要使用id这个值。

- 商品名称：

  是否分词：要分词，因为要根据商品名称的关键词搜索。

  是否索引：要索引。

  是否存储：要存储。

- 商品价格：

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

  是否索引：要索引

  是否存储：要存储

- 商品图片地址：

  是否分词：不分词

  是否索引：不索引

  是否存储：要存储

- 商品描述：

  是否分词：要分词

  是否索引：要索引

  是否存储：因为商品描述内容量大，不在查询结果页面直接显示，不存储。

**常见问题：**

不存储是指不在lucene的索引域中记录，目的是为了节省lucene的索引文件空间。

> **如果要在详情页面显示描述，解决方案：**
>
> 从lucene中取出商品的id，根据商品的id查询关系数据库（MySQL）中item表得到描述信息。

# 六、中文分词器IkAnalyzer

## 什么是中文分词

学过英文的都知道，英文是以单词为单位的，单词与单词之间以空格或者逗号句号隔开。所以对于英文，我们可以简单以空格判断某个字符串是否为一个单词，比如I love China，love 和 China很容易被程序区分开来。

而中文则以字为单位，字又组成词，字和词再组成句子。中文“**詹哥在开课吧讲课**”就不一样了，电脑不知道“**詹哥**”是一个词语还是“**哥在**”是一个词语。

**把中文的句子切分成有意义的词，就是中文分词，也称切词。**

## 使用IkAnalyzer

IKAnalyzer继承Lucene的Analyzer抽象类，使用IKAnalyzer和Lucene自带的分析器方法一样，将Analyzer测试代码改为IKAnalyzer测试中文分词效果。

如果使用中文分词器ik-analyzer，就需要在索引和搜索程序中使用一致的分词器：IK-analyzer。

### 添加依赖

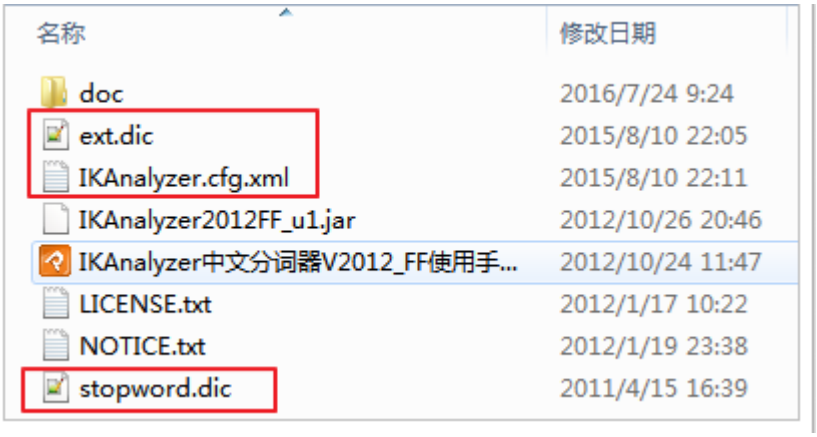
```
    <dependency>
        <groupId>com.janeluo</groupId>
        <artifactId>ikanalyzer</artifactId>
        <version>2012_u6</version>
    </dependency>
```

### 修改代码

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

## 扩展中文词库

- **第一步：从IkAnalyzer的资料包中，拷贝以下三个文件到目标项目的资源目录下**




- **第二步：修改ext.dic文件，添加扩展词。**

  **注意事项：**

  - 一行就是一个词
  - 最好使用IDE内置的文本编辑器进行编辑。
- **第三步：删除索引库中的文件，重新创建索引数据。**

# 七、Solr介绍

## 2.1. 什么是solr

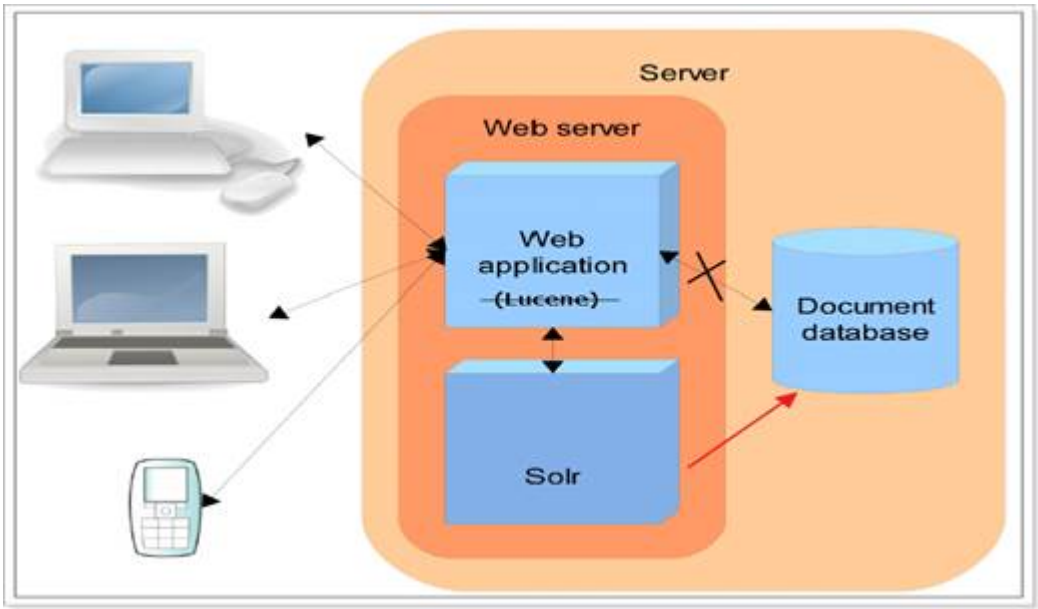
> Solr是一个独立的企业级搜索应用服务器，它对外提供类似于Web-service的API接口。
>
> - 用户可以通过HTTP的**POST**请求，向Solr服务器提交一定格式的XML或者JSON文件，Solr服务器解析文件之后，根据具体需求对索引库执行增删改操作；
> - 用户可以通过HTTP的**GET**请求，向Solr服务器发送搜索请求，并得到XML/JSON格式的返回结果。
>
> Solr 是Apache下的一个顶级开源项目，采用Java开发，基于Lucene。
>
> Solr可以独立运行在**Jetty、Tomcat**等这些Servlet容器中。
>
> Solr提供了比Lucene更为丰富的查询语言，同时实现了可配置、可扩展，并对索引、搜索性能进行了优化。

## 2.2. Solr和Lucene的区别

Lucene是一个开放源代码的全文检索引擎工具包，它不是一个完整的全文检索应用。

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

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




# 八、Solr安装配置

## 下载安装

- 第一步：下载solr压缩包

```
wget http://archive.apache.org/dist/lucene/solr/4.10.4/solr-4.10.4.tgz
```

- 第二步：解压缩

```
tar -xf solr-4.10.4.tgz
```

## 默认使用Jetty部署

Solr默认提供Jetty（java写的Servlet容器）启动solr服务器。

**使用jetty启动：**

1. 进入example目录
2. 执行命令：java -jar start.jar
3. 访问地址：http://192.168.10.136:8983/solr

但是企业中一般使用Tomcat作为服务器，所以下面我们一起来看看如何将solr部署在tomcat中。

## 手动部署到Tomcat

### 配置SolrHome

### 1.1.1. SolrHome和SolrCore

**SolrHome是Solr服务器运行的主目录，Solr服务的所有配置信息包括索引库都是在该目录下。**

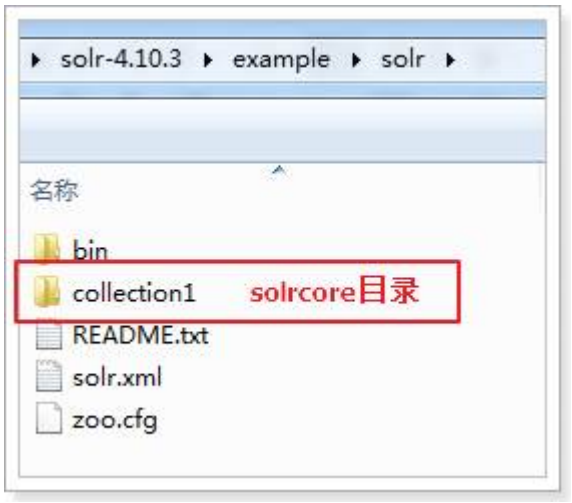
该目录中包括了多个SolrCore目录。

SolrHome和SolrCore是Solr服务器中最重要的两个目录。

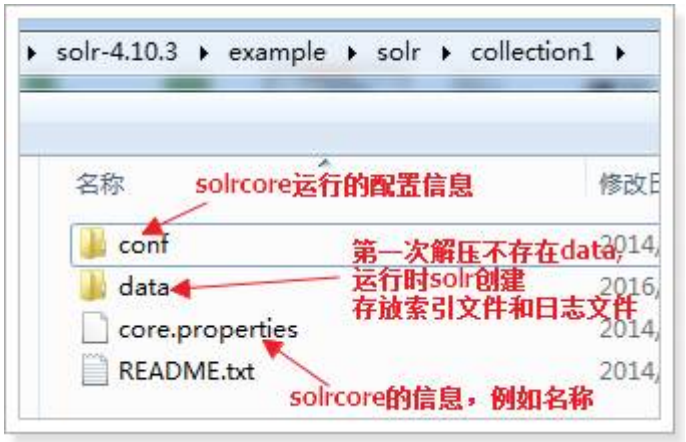
SolrCore就是collection1目录，该目录中包含搜索和索引时需要的配置文件和数据文件（比如索引库中的文件）。

每个SolrCore都可以提供单独的搜索和索引服务。

**SolrHome目录：**

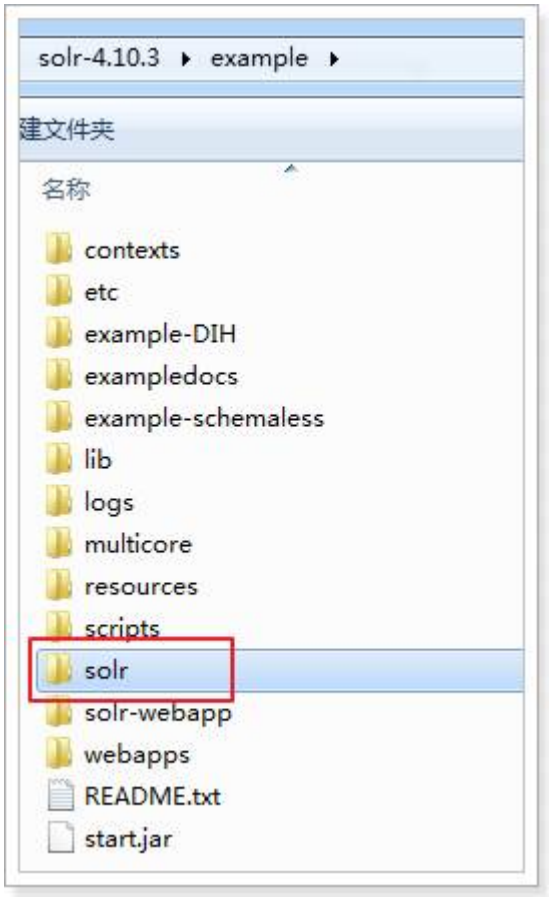



**SolrCore目录：**



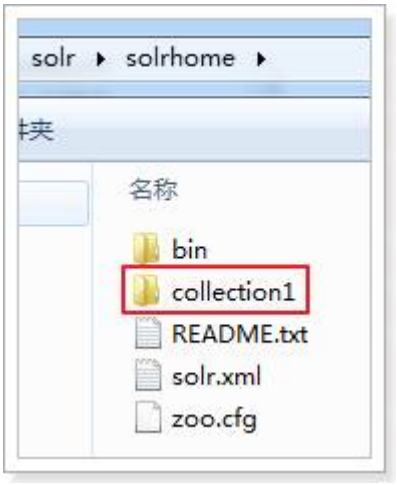

### 1.1.2. 创建SolrHome和SolrCore

下图中的solr目录就是一个solrhome目录，它就是jetty启动的solr服务使用的solrhome目录。



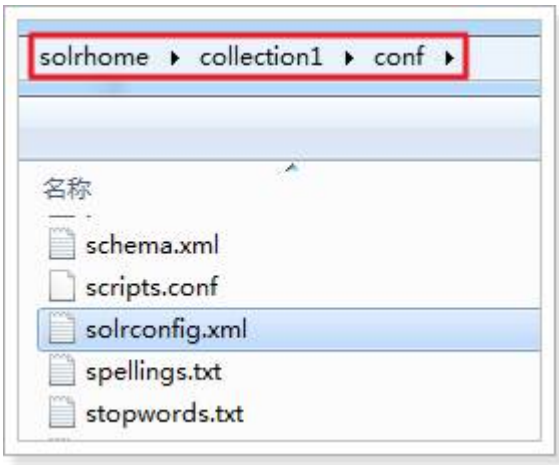

复制该文件夹到本地的一个目录，把文件名称改为solrhome。（*改名不是必须的，只是为了便于理解*）


打开solrhome目录确认是否拥有solrcore




### 1.1.3. 配置SolrCore（了解）

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




这个文件是来配置SolrCore实例的相关运行时信息。**如果使用默认配置可以不用做任何修改**。该配置文件中包含了不少标签，但是我们经常使用的标签有：**lib标签、datadir标签、requestHandler标签。**

#### 1.1.3.1. lib 标签

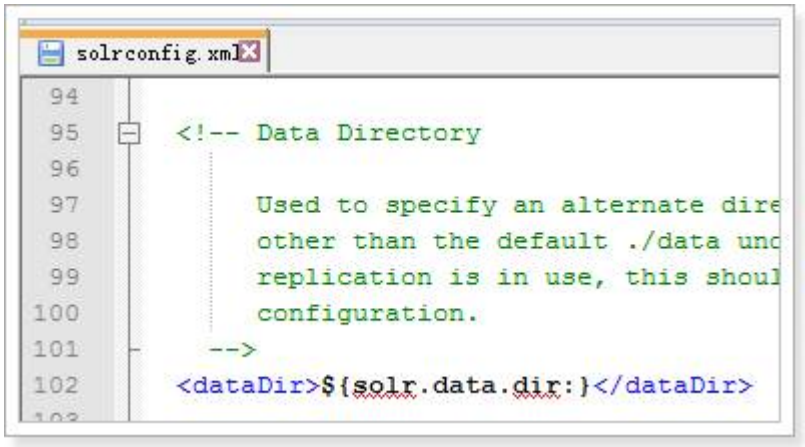
**lib标签可以配置扩展功能的一些jar，用以增强solr本身没有的功能。**

比如solr自身没有『数据导入索引库』功能，如果需要使用，则首先要把这些jar复制到指定的目录，然后通过该配置文件进行相关配置，后面会具体讲解如何配置。

#### 1.1.3.2. datadir标签

**dataDir数据目录data。data目录用来存放索引文件和tlog日志文件。**

**solr.data.dir**表示${SolrCore}/data的目录位置




如果不想使用默认的目录也可以通过solrconfig.xml更改索引目录

例如：

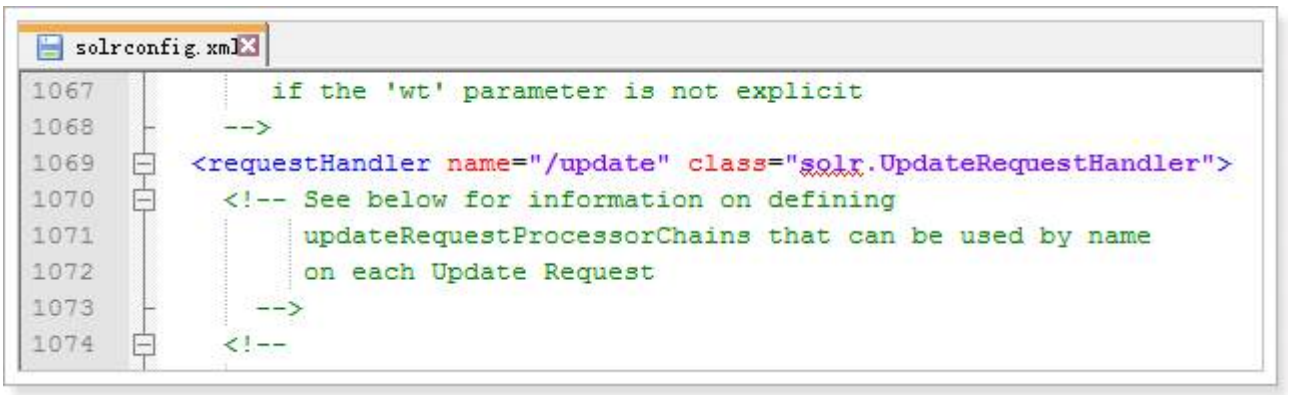
```
<dataDir>${solr.data.dir:F:/develop/solr/collection1/data}</dataDir>
```

(建议不修改，否则配置多个SolrCore会报错 )

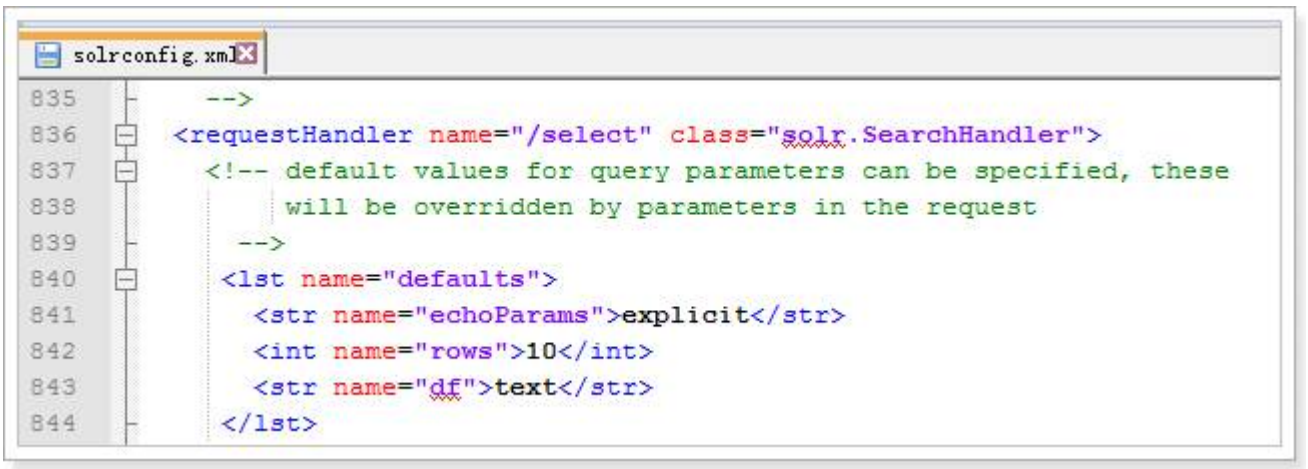
#### 1.1.3.3. requestHandler标签

**requestHandler请求处理器，定义了索引和搜索的访问方式。**

通过/update维护索引，可以完成索引的添加、修改、删除操作。




通过/select搜索索引。




设置搜索参数完成搜索，搜索参数也可以设置一些默认值，如下：

```
<requestHandler name="/select" class="solr.SearchHandler">
    <!-- 设置默认的参数值，可以在请求地址中修改这些参数-->
    <lst name="defaults">
        <str name="echoParams">explicit</str>
        <int name="rows">10</int><!--显示数量-->
        <str name="wt">json</str><!--显示格式-->
        <str name="df">text</str><!--默认搜索字段-->
    </lst>
</requestHandler>
```

## 配置IKAnalyzer中文分词器

第一步：把IKAnalyzer2012FF_u1.jar添加到solr/WEB-INF/lib目录下。

```
cp   /root/IK\ Analyzer\ 2012FF_hf1/IKAnalyzer2012FF_u1.jar
/kkb/server/solr/tomcat-solr/webapps/solr/WEB-INF/lib/
```

第二步：复制IKAnalyzer的配置文件和自定义词典和停用词词典到solr的classes目录下。

```
cp   /root/IK\ Analyzer\ 2012FF_hf1/IKAnalyzer.cfg.xml    /kkb/server/solr/tomcat-
solr/webapps/solr/WEB-INF/classes/
cp   /root/IK\ Analyzer\ 2012FF_hf1/ext.dic    /kkb/server/solr/tomcat-
solr/webapps/solr/WEB-INF/classes/
cp   /root/IK\ Analyzer\ 2012FF_hf1/stopword.dic    /kkb/server/solr/tomcat-
solr/webapps/solr/WEB-INF/classes/
```

第三步：在schema.xml中添加一个自定义的fieldType，使用中文分析器。

```
<!--   IKAnalyzer-->
<fieldType   name="text_ik" class="solr.TextField">
    <analyzer   class="org.wltea.analyzer.lucene.IKAnalyzer"/>
</fieldType>
```

## Tomcat部署

### 第一步：安装Tomcat

### 第二步：部署solr.war

### 第三步：解压缩solr.war

### 第四步：添加solr扩展jar包

### 第五步：添加log4j文件

**第六步：配置solrhome路径**

**第七步：启动Tomcat**

# 九、Solrcloud讲解

## 1.1 Solrcloud介绍

### 1.1.1 什么是solrcloud

**SolrCloud是Solr提供的分布式搜索方案。**

**什么时候使用SolrCloud呢？**

- 当你需要大规模，容错，分布式索引和检索能力时使用 SolrCloud。
- 当索引量很大，搜索请求并发很高时，同样需要使用SolrCloud来满足这些需求。
- 不过当一个系统的索引数据量少的时候是不需要使用SolrCloud的。

**SolrCloud是基于Solr和Zookeeper的分布式搜索方案。**它的主要思想是使用Zookeeper作为SolrCloud集群的配置信息中心，统一管理solrcloud的配置，比如solrconfig.xml和schema.xml。

> 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

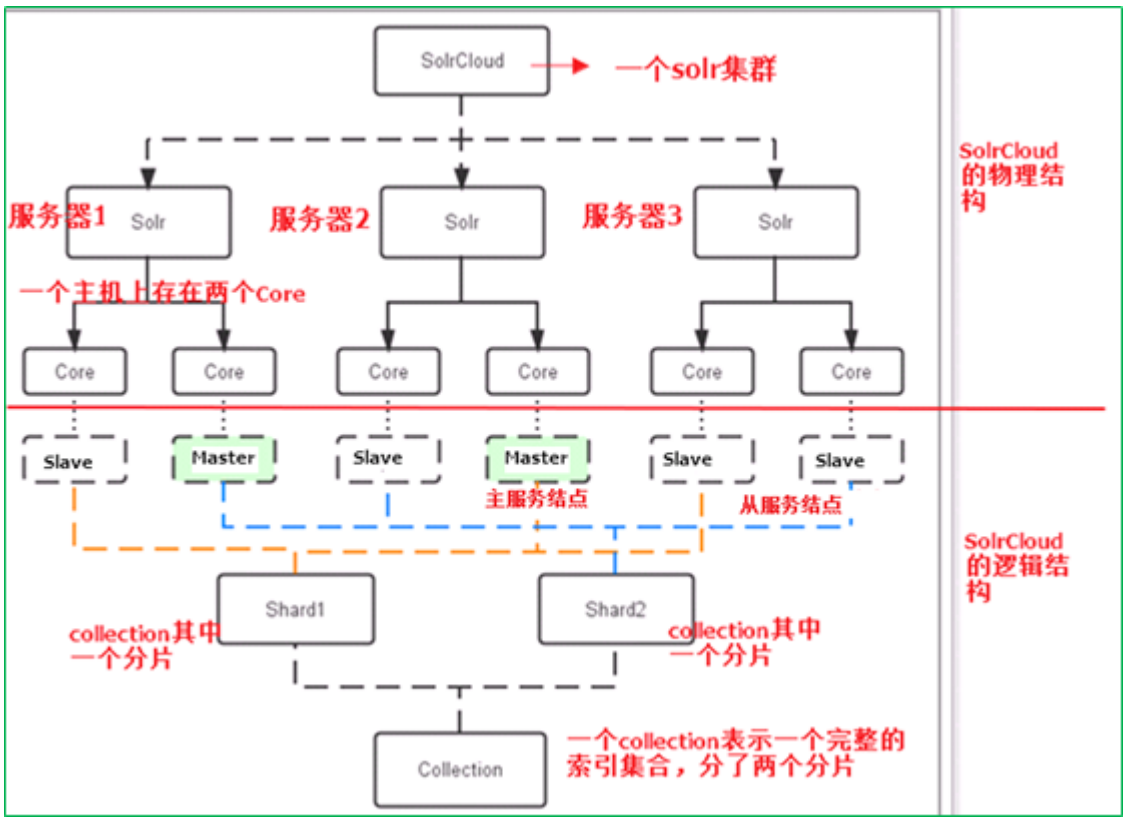
Zookeeper的功能有很多：目录服务（注册中心）、配置中心

### 1.1.2 Solrcloud的结构

solrcloud为了降低单机的处理压力，需要由多台服务器共同来完成索引和搜索任务。实现的思路是**将索引数据进行Shard分片，每个分片由多台服务器共同完成，当一个索引或搜索请求过来时会分别从不同的Shard的服务器中操作索引。**

**solrcloud是基于solr和zookeeper部署**，zookeeper是一个集群管理软件，solrcloud需要由多台solr服务器组成，然后由zookeeper来进行协调管理。

下图是一个SolrCloud应用的例子：




总结：solrcloud的架构分为逻辑结构和物理结构。

**物理结构**：

一个solrcloud集群由多个物理机器或者虚拟机组成，每个虚拟机中可以包含多个solrcore，**一个solrcore对应一个tomcat。**

**逻辑结构**：

一个solrcloud集群可以看成是一个collection（可以将solr集群看成是一个存储量更大、并发量支持更高的一个单机版的solr），一个collection从逻辑上可以被分成多个片（shard）、每个片又有多个solrcore组成。同一个片中的solrcore又会被分为一主多从。

不同的shard可以增强solrcloud的存储功能。

同一个shard中不同的solrcore可以解决单点故障问题，以及可以解决高并发问题。

### 1.1.2.1 物理结构

从物理结构来看，solrcloud需要三台solr服务器，每台服务器包括两个solrcore实例，共同组成一个solrcloud。

### 1.1.2.2 逻辑结构

从逻辑结构来说，整个solrcloud就看成一个大的solrcore，也就是一个collection。而一个collection被分成两个shard分片（shard1和shard2）。

shard1和shard2又分别由三个solrcore组成，其中一个Leader两个Replication。Leader是由zookeeper选举产生，zookeeper控制每个shard上三个Core的索引数据一致，解决高可用问题。

用户发起索引请求分别从shard1和shard2上获取，解决高并发问题。

#### 1.1.2.2.1 Collection

Collection在Solrcloud集群中是**一个逻辑意义上的完整的索引结构**。它常常被划分为一个或多个shard分片，这些shard分片使用相同的配置信息。

比如：针对商品信息搜索可以创建一个collection。

collection=shard1+shard2+....+shardX

**1.1.2.2.2 Shard**

Shard是Collection的逻辑分片。每个Shard被化成一个或者多个replication，通过选举确定哪个是Leader。

**1.1.2.2.3 Core**

每个Core都是Solr中一个独立运行单位，提供索引和搜索服务。**一个shard需要由一个Core或多个Core组成。由于collection由多个shard组成，一个shard由多个core组成，所以也可以说collection一般由多个core组成。**

**1.1.2.2.4 Master或Slave**

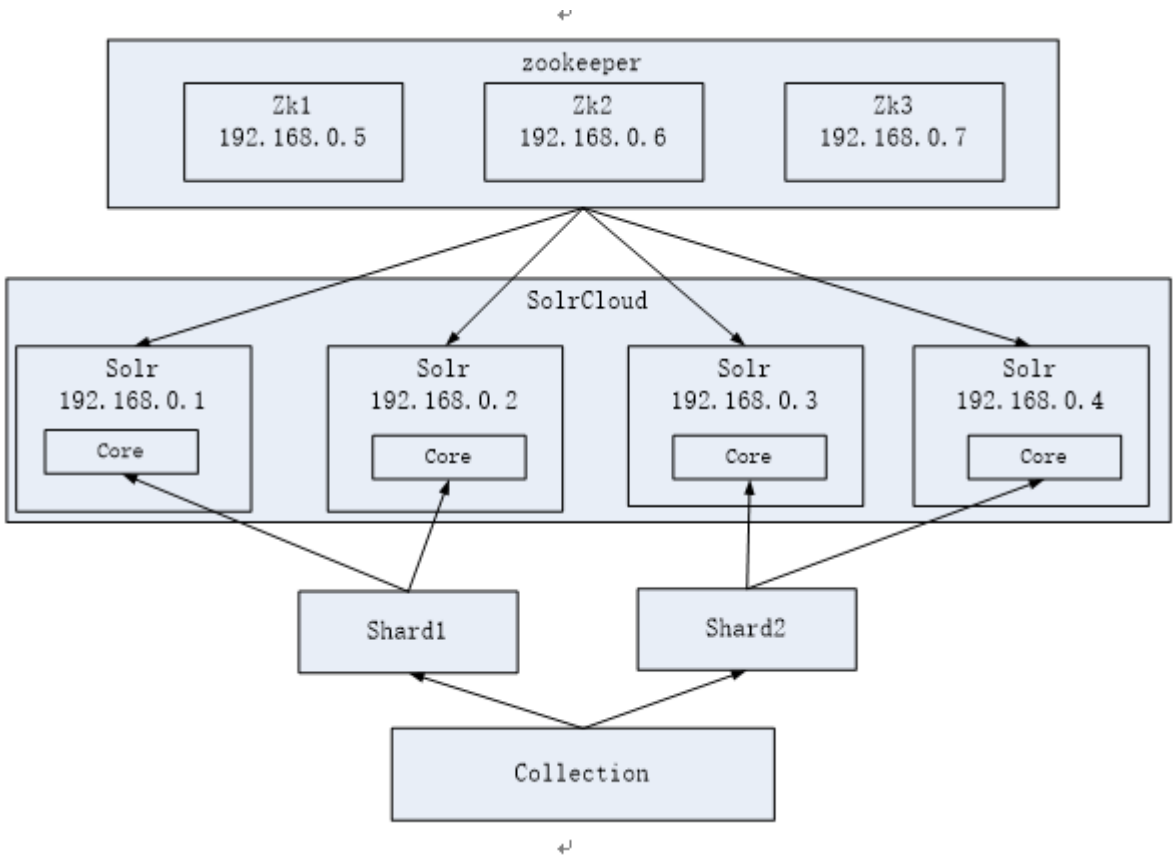
Master是master-slave结构中的主结点（通常说主服务器），Slave是master-slave结构中的从结点（通常说从服务器或备服务器）。**同一个Shard下master和slave存储的数据是一致的，这是为了达到高可用目的。**

# 1.2 solrcloud搭建

注意： 1、 solrcloud是通过zookeeper统一管理配置文件（solrconfig.xml、schema.xml等），所以搭建solrcloud之前，需要先搭建zookeeper。 2、 由于solrcloud一般都是解决大数据量、大并发的搜索服务，所以搭建solrcloud，对zookeeper也需要搭建集群。

本教程的solrcloud是搭建在一台机器中，指定不同的端口。而真实生产环境搭建solrcloud时，只需要修改ip地址即可。

Solrcloud示例结构图如下：



## 1.2.1 环境准备

l Linux CentOS 7

l Jdk 1.8

l Tomcat 8

l solr-4.10.4.tgz

l zookeeper-3.4.6z

## 1.2.2 zookeeper集群搭建

**需要三台zookeeper、分别是zk1、zk2、zk3，对应的端口分别为2281、2282、2283**

**第一步：安装jdk，zookeeper是使用java开发的。**

**第二步：上传zookeeper-3.4.6.tar.gz到linux指定目录，解压缩并改名为zk1**

```
tar -zxf zookeeper-3.4.6.tar.gz
mv zookeeper-3.4.6 zk1
```

**第三步：创建zoo.cfg，进入zk1/conf目录，将zoo_sample.cfg改为zoo.cfg**

```
cp zoo_sample.cfg zoo.cfg
```

**第四步：修改zoo.cfg，编写 zookeeper集群配置**

**注意：客户端端口、通信端口、选举端口，在同一台机器搭建集群时，需要与集群中其他zookeeper服务的端口区分。**

```
dataDir=/usr/local/solrcloud/zk1/data
# the port at which the clients will connect
clientPort=2281
#集群中每台机器都是以下配置
#2881系列端口是zookeeper通信端口
#3881系列端口是zookeeper投票选举端口
server.1=192.168.10.139:2881:3881
server.2=192.168.10.139:2882:3882
server.3=192.168.10.139:2883:3883
```

**第五步：在dataDir目录下创建myid文件，文件内容为1，对应server.1中的1。**

**第六步：拷贝zk1，复制两个目录zk2、zk3。并修改zoo.cfg和myid两个文件。**

Zk2的zoo.cfg修改内容

```
dataDir=/usr/local/solrcloud/zk2/data
# the port at which the clients will connect
clientPort=2282
```

Zk3的zoo.cfg修改内容

```
dataDir=/usr/local/solrcloud/zk3/data
# the port at which the clients will connect
clientPort=2283
```

创建zk2和zk3的myid文件，其内容分别为**2**和**3**。

**第七步：启动3台zookeeper服务**

```
/usr/local/solrcloud/zk1/bin/zkServer.sh start

/usr/local/solrcloud/zk2/bin/zkServer.sh start

/usr/local/solrcloud/zk3/bin/zkServer.sh start
```

**第八步：查看zookeeper状态**

```
/usr/local/solrcloud/zk1/bin/zkServer.sh status

/usr/local/solrcloud/zk2/bin/zkServer.sh status

/usr/local/solrcloud/zk3/bin/zkServer.sh status
```

## 1.2.3 solrcloud搭建

solrcloud由zookeeper统一管理solr的配置文件（主要是schema.xml、solrconfig.xml），solrcloud各个solrcore节点都使用zookeeper管理的统一配置文件。

**注意：solrcloud启动之前，需要先启动zookeeper集群。**

Solrcloud安装步骤：

**第一步：复制4个单机版solr服务对应的tomcat，并分别改端口为： 8280、8380、8480、8580。**

**第二步：复制4个solrhome，分别为solrhome8280、solrhome8380、solrhome8480、solrhome8580。一个solr实例对应一个solrhome。**

**第三步：修改每个solr服务的web.xml，分别指定对应的solrhome路径。**

```
<env-entry>
   <env-entry-name>solr/home</env-entry-name>
   <env-entry-value>/usr/local/solrcloud/solrhome4</env-entry-value>
   <env-entry-type>java.lang.String</env-entry-type>
</env-entry>
```

**第四步：修改每个solrhome下的solr.xml，指定对应solr服务的tomcat的ip和端口。**

```
<solrcloud>
  <str name="host">192.168.242.139</str>
  <int name="hostPort">8280</int>
  <str name="hostContext">${hostContext:solr}</str>
  <int name="zkClientTimeout">${zkClientTimeout:30000}</int>
  <bool name="genericCoreNodeNames">${genericCoreNodeNames:true}</bool>
```

**第五步：设置tomcat的启动参数，在每个tomcat目录下的bin/catalina.sh，添加以下内容：**

```
JAVA_OPTS="-DzkHost=192.168.10.139:2281,192.168.10.139:2282,192.168.10.139:2283"
```

**第六步：将solr配置文件上传到zookeeper中，进行统一管理。**

使用/usr/local/solr-4.10.3/example/scripts/cloud-scripts下的zkcli.sh命令
将/usr/local/solrcloud/solrhome8280/collection1/conf目录上传到zookeeper进行配置。

**命令如下**：

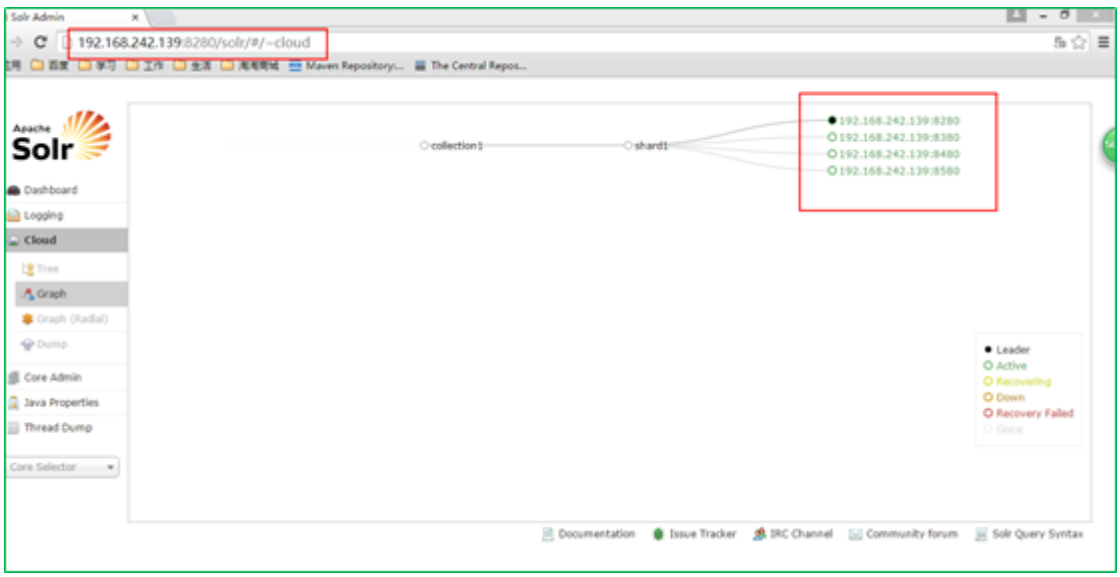
```
./zkcli.sh -zkhost 192.168.10.139:2281,192.168.10.139:2282,192.168.10.139:2283   -cmd upconfig -confdir /usr/local/solrcloud/solrhome8280/collection1/conf   -confname myconf
```

使用zookeeper自带的zkCli.sh命令连接zookeeper集群，查看上传的配置文件。

```
./zkCli.sh -server localhost:2281

[zk: localhost:2181(CONNECTED) 0] ls /
[configs, zookeeper]
[zk: localhost:2181(CONNECTED) 1] ls /configs
[myconf]
[zk: localhost:2181(CONNECTED) 2] ls /configs/myconf
```

**第七步：启动所有solr服务**




# 1.2.4 集群分片

## 1.2.4.1 创建collection

- **需求**：

创建新的集群，名称为collection2，集群中有四个solr节点，将集群分为两片，每片两个副本。

- **HTTP命令：**

```
http://192.168.10.139:8280/solr/admin/collections?
action=CREATE&name=collection2&numShards=2&replicationFactor=2
```


### 1.2.4.2 删除collection

- **需求：**

删除名称为collection1的集群。

- **HTTP命令：**

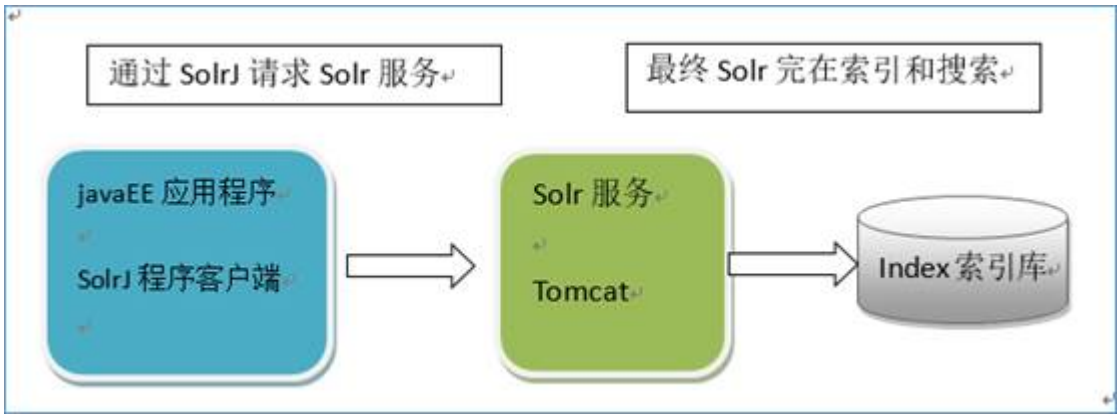
```
http://192.168.10.139:8280/solr/admin/collections?action=DELETE&name=collection1
```

- 原来的collection1被删除了


# 十、Solrj的使用

## 1.1. 什么是solrj

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




# 1.2. 需求

使用solrj的API调用远程Solr服务器，实现对索引库的增删改操作。

# 1.3. 添加jar

Solrj的包，\solr解压缩目录\dist\

solrj依赖包，\solr解压缩目录\dist\solrj-lib\

Solr服务的依赖包，\solr解压缩目录\example\lib\ext

# 1.4. 代码实现

## 1.4.1. 添加&修改索引

### 1.5.1.1. 步骤

1、创建HttpSolrServer对象，通过它和Solr服务器建立连接。

2、创建SolrInputDocument对象，然后通过它来添加域。

3、通过HttpSolrServer对象将SolrInputDocument添加到索引库。

4、提交。

### 1.4.1.2. 代码

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

```
@Test
public void testCreateAndUpdateIndex() throws Exception {
        // 1. 创建HttpSolrServer对象
        // 设置solr服务接口,浏览器客户端地址http://127.0.0.1:8081/solr/#/
        String baseURL = "http://127.0.0.1:8081/solr";
        HttpSolrServer httpSolrServer = new HttpSolrServer(baseURL);
```

```
        // 2. 创建SolrInputDocument对象
        SolrInputDocument document = new SolrInputDocument();
        document.addField("id", "kkb01");
        document.addField("content ", "hello world , hello solr");
        // 3. 把SolrInputDocument对象添加到索引库中
        httpSolrServer.add(document);
        // 4. 提交
        httpSolrServer.commit();
}
```

#### 1.4.1.3. 查询测试

## 1.4.2. 删除索引

### 1.4.2.1. 代码

抽取HttpSolrServer 的创建代码

```
private HttpSolrServer httpSolrServer;

// 提取HttpSolrServer创建
@Before
public void init() {
        // 1. 创建HttpSolrServer对象
        // 设置solr服务接口,浏览器客户端地址http://127.0.0.1:8081/solr/#/
        String baseURL = "http://127.0.0.1:8081/solr/";
        this.httpSolrServer = new HttpSolrServer(baseURL);
}
```

**删除索引逻辑，两种：**

- 根据id删除
- 根据条件删除

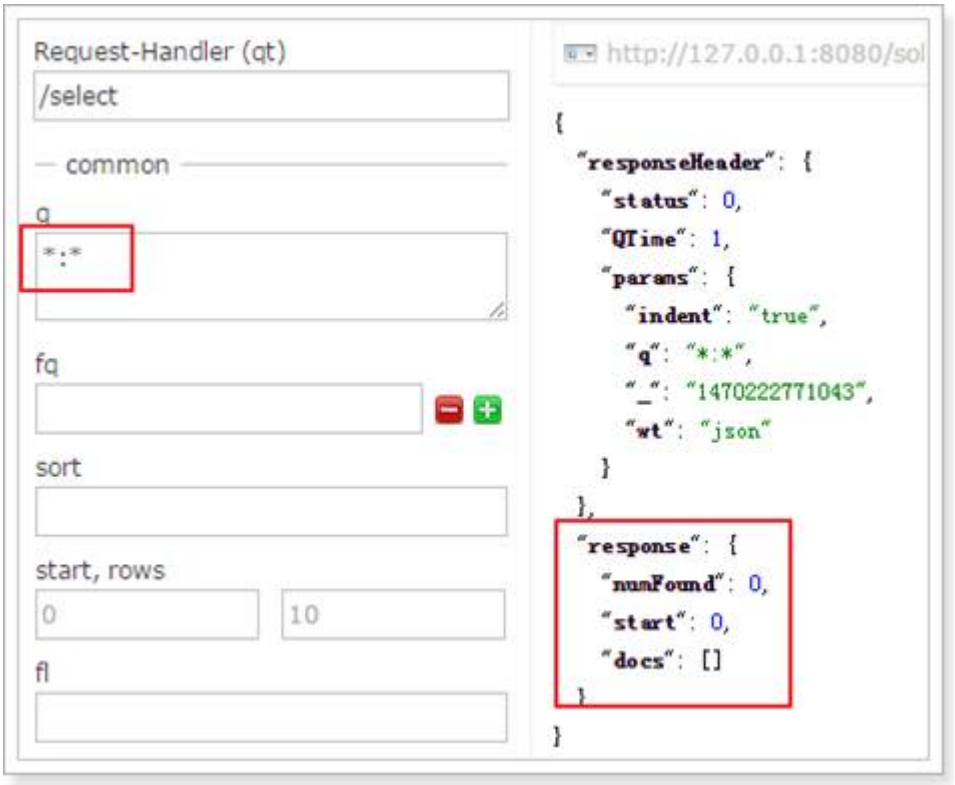
可以使用:作为条件，就是删除所有数据（慎用）

```
@Test
public void testDeleteIndex() throws Exception {
        // 根据id删除索引数据
        // this.httpSolrServer.deleteById("c1001");

        // 根据条件删除（如果是*:*就表示全部删除，慎用）
        this.httpSolrServer.deleteByQuery("*:*");

        // 提交
        this.httpSolrServer.commit();
}
```

#### 1.4.2.2. 查询测试




# solrJ访问solrCloud

```
1
2  public class SolrCloudTest {
3      // zookeeper地址
4      private static String zkHostString =
   "192.168.101.7:2181,192.168.101.8:2181,192.168.101.9:2181";
5      // collection默认名称，比如我的solr服务器上的collection是
   collection2_shard1_replica1，就是去掉“_shard1_replica1”的名称
6      private static String defaultCollection = "collection2";
7      // 客户端连接超时时间
8      private static int zkClientTimeout = 3000;
9      // zookeeper连接超时时间
10     private static int zkConnectTimeout = 3000;
11
12     // cloudSolrServer实际
13     private CloudSolrServer cloudSolrServer;
14
15     // 测试方法之前构造 CloudSolrServer
16     @Before
17     public void init() {
18        cloudSolrServer = new CloudSolrServer(zkHostString);
19        cloudSolrServer.setDefaultCollection(defaultCollection);
20        cloudSolrServer.setZkClientTimeout(zkClientTimeout);
21        cloudSolrServer.setZkConnectTimeout(zkConnectTimeout);
22        cloudSolrServer.connect();
```

```
    }

    // 向solrCloud上创建索引
    @Test
    public void testCreateIndexToSolrCloud() throws SolrServerException,
            IOException {

        SolrInputDocument document = new SolrInputDocument();
        document.addField("id", "100001");
        document.addField("title", "李四");
        cloudSolrServer.add(document);
        cloudSolrServer.commit();

    }

    // 搜索索引
    @Test
    public void testSearchIndexFromSolrCloud() throws Exception {

        SolrQuery query = new SolrQuery();
        query.setQuery(":");
        try {
            QueryResponse response = cloudSolrServer.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) {
                ArrayList title = (ArrayList) doc.getFieldValue("title");
                String id = (String) doc.getFieldValue("id");
                System.out.println("id: " + id);
                System.out.println("title: " + title);
                System.out.println();
            }
        } catch (SolrServerException e) {
            e.printStackTrace();
        } catch (Exception e) {
            System.out.println("Unknowned Exception!!!!");
            e.printStackTrace();
        }
    }

    // 删除索引
    @Test
    public void testDeleteIndexFromSolrCloud() throws SolrServerException,
IOException {

        // 根据id删除
        UpdateResponse response = cloudSolrServer.deleteById("zhangsan");
        // 根据多个id删除
        // cloudSolrServer.deleteById(ids);
        // 自动查询条件删除
```

```
75          // cloudSolrServer.deleteByQuery("product_keywords:教程");
76          // 提交
77          cloudSolrServer.commit();
78      }
79  }
```