1 Lucene介绍

1.1 什么是lucene

Lucene是Apache的一个全文检索引擎工具包，通过lucene可以让程序员快速开发一个全文检索功能。

引擎：核心组件

工具包：jar包、类库

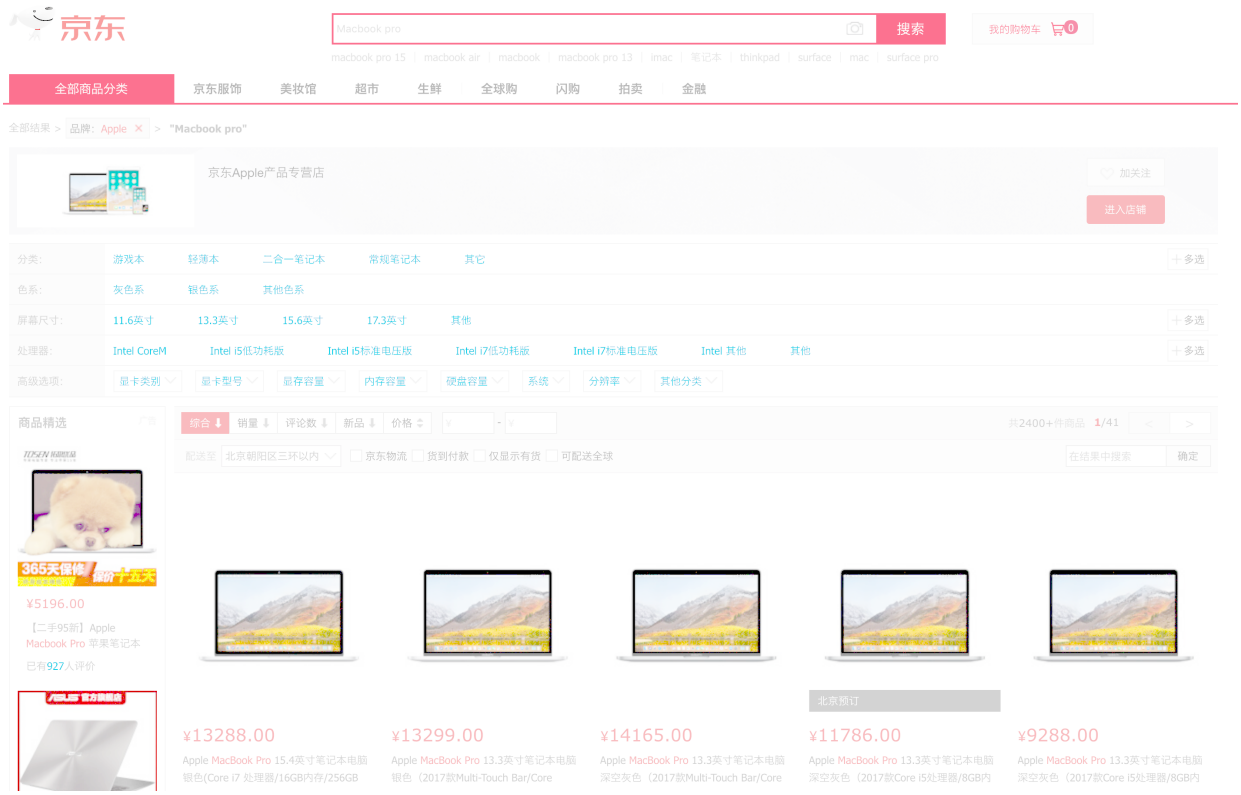
1.2 全文检索的应用场景

1.2.1 搜索引擎

> lucence 可以作用搜索引擎的数据检索实现,但是它本身不是搜索引擎

1.2.2 站内搜索

> 京东,淘宝的搜索



1.2.3 文件系统的搜索

> 操作系统的搜索功能

Lucene和搜索引擎不是一回事
Lucene是一个工具包，它不能独立运行，不能单独对外提供服务。
搜索引擎可以独立运行对外提供搜索服务

1.3全文检索

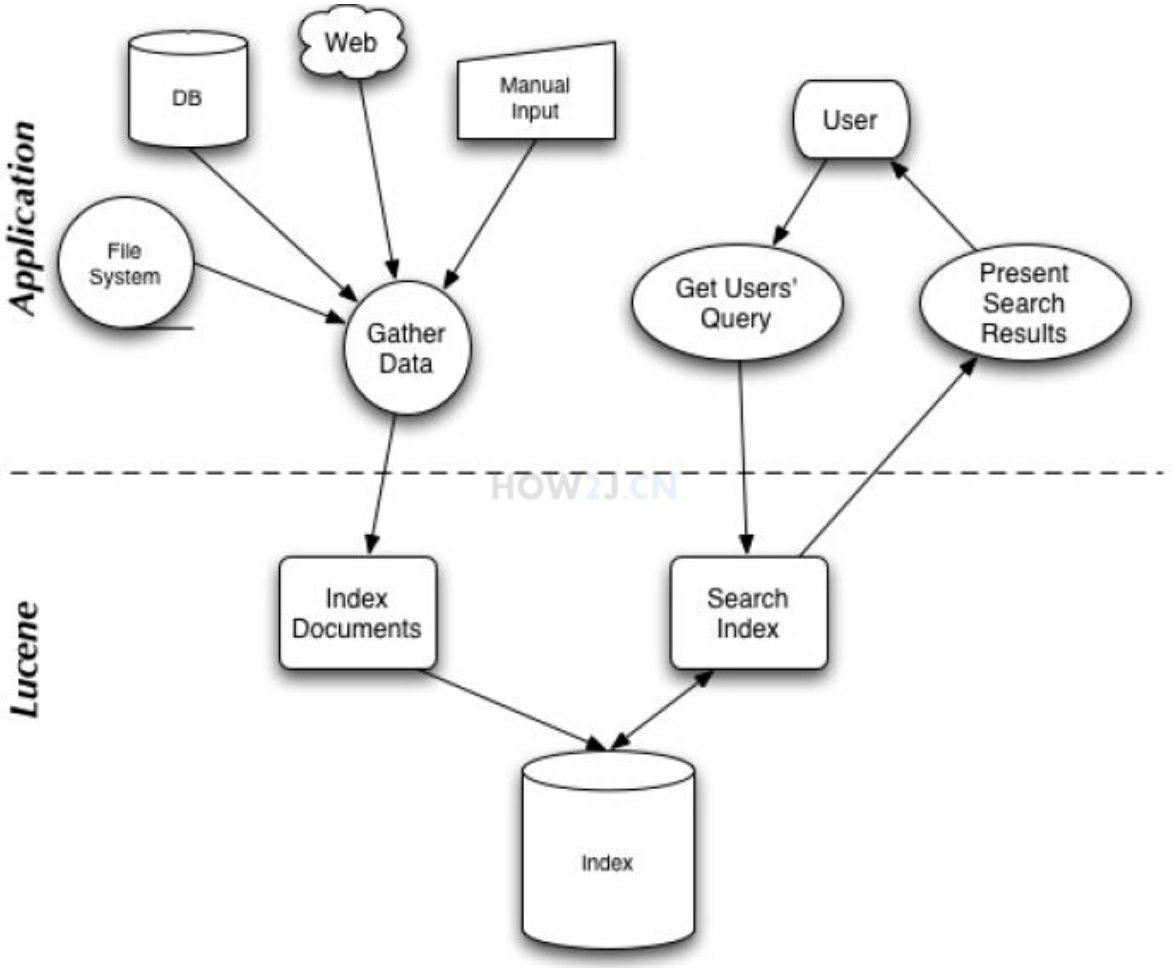
全文检索首先对要搜索的文档进行分词，然后形成索引，通过查询索引来查询文档。

全文检索就是先创建索引，然后根据索引来进行搜索的过程，就叫全文检索。

比如：字典，
字典的偏旁部首页，就类似于luence的索引
　　字典的具体内容，就类似于luence的文档内容

1.4 lucence 执行流程

1. 首先搜集数据 数据可以使文件系统，数据库，网络上，手工输入的，或者像本例直接写在内存上的
2. 通过数据创建索引
3. 用户输入关键字
4. 通过关键字创建查询器
5. 根据查询器到索引里获取数据
6. 然后把查询结果展示在用户面前(渲染需要用我们自己渲染, lucence 本身不负责渲染)



2 入门使用

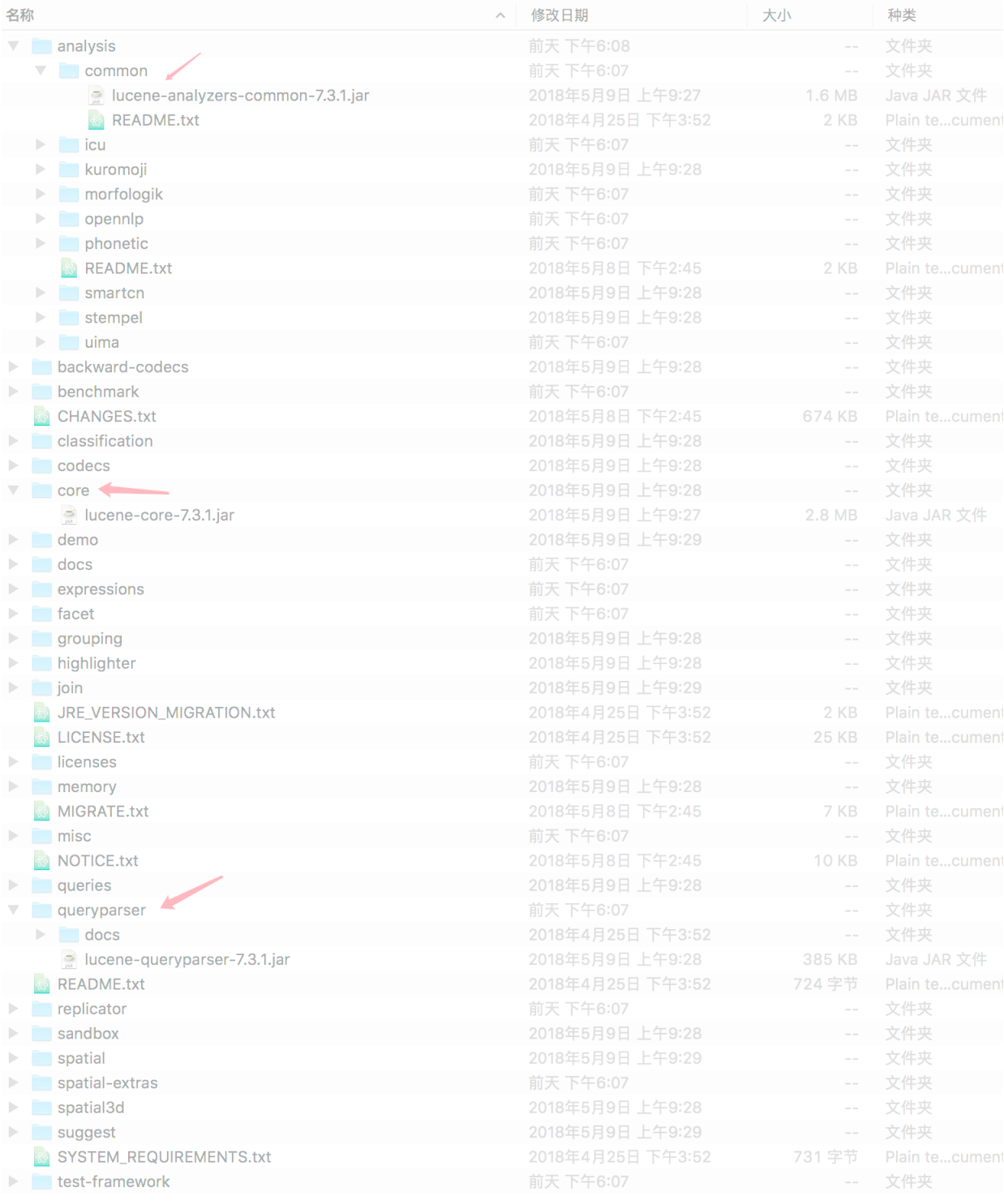
2.1 环境

> JDK 1.8+
>
> lucence 7.X

2.2 下载

官方网站：http://lucene.apache.org/

2.3 目录&所需内容



2.4 流程

2.4.1 获取数据

全文检索搜索的内容的格式是多种多样的，比如：视频、mp3、图片、文档等等。对于这种格式不同的数据，需要先将他们采集到本地，然后统一封装到lucene的文档对象中，也就是说需要将存储的内容进行统一才能对它进行查询
获取方式可以是从自己的数据库,网络采集等

2.4.2 索引结构

文档域
文档域存储的信息就是采集到的信息，通过Document对象来存储，具体说是通过Document对象中field域来存储数据。

比如：数据库中一条记录会存储一个一个Document对象，数据库中一列会存储成Document中一个field域。

文档域中，Document对象之间是没有关系的。而且每个Document中的field域也不一定一样。

索引域

索引域主要是为了搜索使用的。索引域内容是经过lucene分词之后存储的。

倒排索引表

传统方法是先找到文件，如何在文件中找内容，在文件内容中匹配搜索关键字，这种方法是顺序扫描方法，数据量大就搜索慢
倒排索引结构是根据内容（词语）找文档，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，它是在索引中匹配搜索关键字，由于索引内容量有限并且采用固定优化算法搜索速度很快，找到了索引中的词汇，词汇与文档关联，从而最终找到了文档,一个内容后续跟着很多 id
比如关键字 世界杯->id1->id2.....

2.4.3 pom.xml

```
    <dependencies>
        <!-- https://mvnrepository.com/artifact/org.apache.lucene/lucene-core -->
        <dependency>
            <groupId>org.apache.lucene</groupId>
            <artifactId>lucene-core</artifactId>
            <version>7.3.1</version>
        </dependency>
        <!-- https://mvnrepository.com/artifact/org.apache.lucene/lucene-
queryparser -->
        <dependency>
            <groupId>org.apache.lucene</groupId>
            <artifactId>lucene-queryparser</artifactId>
            <version>7.3.1</version>
        </dependency>
        <!-- https://mvnrepository.com/artifact/org.apache.lucene/lucene-
analyzers-common -->
        <dependency>
            <groupId>org.apache.lucene</groupId>
            <artifactId>lucene-analyzers-common</artifactId>
```

```
            <version>7.3.1</version>
        </dependency>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.12</version>
        </dependency>
        <!-- https://mvnrepository.com/artifact/com.janeluo/ikanalyzer
    此版本不支持最新 lucence 和 solr,需自己添加依赖新包到项目或者私服

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

2.4.4 数据源对象

```
public class Book {
    private Integer id;
    private String name;
    private Float price;
    private String pic;
    private String description;
    public Integer getId() {
        return id;
    }
    public void setId(Integer id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public Float getPrice() {
        return price;
    }
    public void setPrice(Float price) {
        this.price = price;
    }
    public String getPic() {
        return pic;
    }
    public void setPic(String pic) {
```

```
        this.pic = pic;
    }
    public String getDescription() {
        return description;
    }
    public void setDescription(String description) {
        this.description = description;
    }

}
```

2.4.5 生成数据

> 此处模拟数据,未从数据库加载

```
public class BookDaoImpl {

    @Override
    public List<Book> queryBooks() {
        List<Book> list = new ArrayList<>();
        Book book = new Book();
        book.setId(1);
        book.setName("Java是世界上最好的编程语言");
        book.setPic("111.jpg");
        book.setDescription("Java描述");
        book.setPrice(20.54f);


        Book book1 = new Book();
        book1.setId(2);
        book1.setName("Java和Spring");
        book1.setPic("222.jpg");
        book1.setDescription("Spring描述");
        book1.setPrice(31.54f);

        list.add(book);
        list.add(book1);
        return list;
    }

}
```

2.4.6 创建索引

> 创建索引流程 创建我们的 Document 文档对象-->通过analyzer进行分词--->通过 indexWriter 索引对象写入到目录对象Directory,最终到达索引库中

> **IndexWriter**是索引过程的核心组件，通过IndexWriter可以**创建新索引、更新索引、删除索引**操作。IndexWriter需要通过**Directory**对索引进行存储操作。
>
> **Directory**描述了索引的存储位置，底层封装了I/O操作，负责对索引进行存储。它是一个抽象类，它的子类常用的包括**FSDirectory**（在文件系统存储索引）、**RAMDirectory**（在内存存储索引）。

2.4.7 创建索引

```
public class IndexManager {

    @Test
    public void createIndex() throws Exception {
        // 采集数据
        BookDao dao = new BookDaoImpl();
        List<Book> list = dao.queryBooks();

        // 将采集到的数据封装到Document对象中
        List<Document> docList = new ArrayList<>();
        Document document;
        for (Book book : list) {
            document = new Document();
            // store:如果是yes，则说明存储到文档域中
            // 图书ID
            // 不分词、索引、存储 StringField
            Field id = new StringField("id", book.getId().toString(), Store.YES);
            // 图书名称
            // 分词、索引、存储 TextField
            Field name = new TextField("name", book.getName(), Store.YES);
            // 图书价格
            // 分词、索引、存储 但是是数字类型，所以使用FloatField
            // Field price = new FloatField("price", book.getPrice(), Store.YES);
            FloatDocValuesField price = new
FloatDocValuesField("price",book.getPrice());
            // 图书图片地址
            // 不分词、不索引、存储 StoredField
            Field pic = new StoredField("pic", book.getPic());
            // 图书描述
            // 分词、索引、不存储 TextField
            Field description = new TextField("description",
                    book.getDescription(), Store.NO);

            // 设置boost值
            /*if (book.getId() == 4)
                description.setBoost(100f);*/

            // 将field域设置到Document对象中
            document.add(id);
            document.add(name);
```

```
            document.add(price);
            document.add(pic);
            document.add(description);

            docList.add(document);
        }

        // 创建分词器，标准分词器
         //Analyzer analyzer = new StandardAnalyzer();
        // 使用ikanalyzer
        Analyzer analyzer = new IKAnalyzer();

        // 创建IndexWriter
        IndexWriterConfig cfg = new IndexWriterConfig(analyzer);
        // 指定索引库的地址
        //File indexFile = new File("/Users/jackiechan/Documents/index");
        Directory directory =
FSDirectory.open(Paths.get("/Users/jackiechan/Documents/index"));
        IndexWriter writer = new IndexWriter(directory, cfg);

        // 通过IndexWriter对象将Document写入到索引库中
        for (Document doc : docList) {
            writer.addDocument(doc);
        }

        // 关闭writer
        writer.close();
    }
}
```

Lucene中分词主要分为两个步骤：分词、过滤

分词：将field域中的内容一个个的分词。

过滤：将分好的词进行过滤，比如去掉标点符号、大写转小写、词的型还原（复数转单数、过去式转成现在式）、停用词过滤

停用词：单独应用没有特殊意义的词。比如的、啊、等，英文中的this is a the等等。

比如要分词的内容

```
Lucene is a Java full-text search engine.
```

分词

```
Lucene
```

```
is

a

Java

Full

-

text

search

engine

.
```

过滤

```
去掉标点符号

Lucene

is

a

Java

Full

text

search

engine
```

```
去掉停用词

Lucene
```

```
Java

Full

text

search

engine


大写转小写

lucene

java

full

text

search

engine
```

如下图是语汇单元的生成过程：

从一个Reader字符流开始，创建一个基于Reader的Tokenizer分词器，经过三个TokenFilter生成语汇单元Token。

同一个域中相同的语汇单元（Token）对应同一个Term（词），它记录了语汇单元的内容及所在域的域名等，还包括来该token出现的频率及位置。

l 不同的域中拆分出来的相同的单词对应不同的term。**

l 相同的域中拆分出来的相同的单词对应相同的term。**

例如：图书信息里面，图书名称中的java和图书描述中的java对应不同的term

3 Field域

3.1 Field的属性

Ø 是否分词（Tokenized）

是：对该field存储的内容进行分词，分词的目的，就是为了索引。

比如：商品名称、商品描述、商品价格

否：不需要对field存储的内容进行分词，不分词，不代表不索引，而是将整个内容进行索引。

比如：商品id

Ø 是否索引（Indexed）

是：将分好的词进行索引，索引的目的，就是为了搜索。

比如：商品名称、商品描述、商品价格、商品id

否：不索引，也就是不对该field域进行搜索。

Ø 是否存储（Stored）

是：将field域中的内容存储到文档域中。存储的目的，就是为了搜索页面显示取值用的。

比如：商品名称、商品价格、商品id、商品图片地址

否：不将field域中的内容存储到文档域中。不存储，则搜索页面中没法获取该field域的值。

比如：商品描述，由于商品描述在搜索页面中不需要显示，再加上商品描述的内容比较多，所以就不需要进行存储。

如果需要商品描述，则根据搜索出的商品ID去数据库中查询，然后显示出商品描述信息即可。

3.2 Field的常用类型

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.

4 索引维护和查询

数据库中的数据发生变化后我们也需要更新索引库,如增删改
删除可以根据条件删除,也可以删除所有有
查询以下两种方式

1通过Query子类来创建查询对象
Query子类常用的有：TermQuery、NumericRangeQuery、BooleanQuery
优缺点不能输入lucene的查询语法，不需要指定分词器

2 通过QueryParser来创建查询对象（常用）
QueryParser、MultiFieldQueryParser
优缺点可以输入lucene的查询语法、可以指定分词器

4.1索引维护

```
package com.qianfeng.lucene.first;

import com.qianfeng.lucene.dao.BookDao;
import com.qianfeng.lucene.dao.BookDaoImpl;
import com.qianfeng.lucene.po.Book;
import org.apache.lucene.analysis.Analyzer;
```

```java
import org.apache.lucene.analysis.standard.StandardAnalyzer;
import org.apache.lucene.document.*;
import org.apache.lucene.document.Field.Store;
import org.apache.lucene.index.IndexWriter;
import org.apache.lucene.index.IndexWriterConfig;
import org.apache.lucene.index.Term;
import org.apache.lucene.store.Directory;
import org.apache.lucene.store.FSDirectory;
import org.junit.Test;
import org.wltea.analyzer.lucene.IKAnalyzer;

import java.nio.file.Paths;
import java.util.ArrayList;
import java.util.List;


public class IndexManager {

    @Test
    public void createIndex() throws Exception {
        // 采集数据
        BookDao dao = new BookDaoImpl();
        List<Book> list = dao.queryBooks();

        // 将采集到的数据封装到Document对象中
        List<Document> docList = new ArrayList<>();
        Document document;
        for (Book book : list) {
            document = new Document();
            // store:如果是yes，则说明存储到文档域中
            // 图书ID
            // 不分词、索引、存储 StringField
            Field id = new StringField("id", book.getId().toString(), Store.YES);
            // 图书名称
            // 分词、索引、存储 TextField
            Field name = new TextField("name", book.getName(), Store.YES);
            // 图书价格
            // 分词、索引、存储 但是是数字类型，所以使用FloatField
            // Field price = new FloatField("price", book.getPrice(), Store.YES);
            FloatDocValuesField price = new
FloatDocValuesField("price",book.getPrice());
            // 图书图片地址
            // 不分词、不索引、存储 StoredField
            Field pic = new StoredField("pic", book.getPic());
            // 图书描述
            // 分词、索引、不存储 TextField
            Field description = new TextField("description",
                    book.getDescription(), Store.NO);
```

```
            // 设置boost值
            /*if (book.getId() == 4)
                description.setBoost(100f);*/

            // 将field域设置到Document对象中
            document.add(id);
            document.add(name);
            document.add(price);
            document.add(pic);
            document.add(description);

            docList.add(document);
        }

        // 创建分词器，标准分词器
         //Analyzer analyzer = new StandardAnalyzer();
        // 使用ikanalyzer
        Analyzer analyzer = new IKAnalyzer();

        // 创建IndexWriter
        IndexWriterConfig cfg = new IndexWriterConfig(analyzer);
        // 指定索引库的地址
        //File indexFile = new File("/Users/jackiechan/Documents/index");
        Directory directory =
FSDirectory.open(Paths.get("/Users/jackiechan/Documents/index"));
        IndexWriter writer = new IndexWriter(directory, cfg);

        // 通过IndexWriter对象将Document写入到索引库中
        for (Document doc : docList) {
            writer.addDocument(doc);
        }

        // 关闭writer
        writer.close();
    }

    @Test
    public void deleteIndex() throws Exception {
        // 创建分词器，标准分词器
        Analyzer analyzer = new StandardAnalyzer();

        // 创建IndexWriter
        IndexWriterConfig cfg = new IndexWriterConfig(analyzer);
        Directory directory = FSDirectory
                .open(Paths.get("/Users/jackiechan/Documents/index"));
        // 创建IndexWriter
        IndexWriter writer = new IndexWriter(directory, cfg);

        // Terms
```

```
        // writer.deleteDocuments(new Term("id", "1"));//删除特定条件的

        // 删除全部（慎用）
        writer.deleteAll();

        writer.close();
    }

    @Test
    public void updateIndex() throws Exception {
        // 创建分词器，标准分词器
        Analyzer analyzer = new StandardAnalyzer();

        // 创建IndexWriter
        IndexWriterConfig cfg = new IndexWriterConfig(analyzer);

        Directory directory = FSDirectory
                .open(Paths.get("/Users/jackiechan/Documents/index"));
        // 创建IndexWriter
        IndexWriter writer = new IndexWriter(directory, cfg);

        // 第一个参数：指定查询条件
        // 第二个参数：修改之后的对象
        // 修改时如果根据查询条件，可以查询出结果，则将以前的删掉，然后覆盖新的Document
对象，如果没有查询出结果，则新增一个Document
        // 修改流程即：先查询，再删除，在添加
        Document doc = new Document();
        doc.add(new TextField("name", "lisi", Store.YES));
        writer.updateDocument(new Term("name", "zhangsan"), doc);//有则更新,没有则创
建
        writer.close();
    }
}
```

4.2 查询

```
package com.qianfeng.lucene.first;

import org.apache.lucene.analysis.Analyzer;
import org.apache.lucene.analysis.standard.StandardAnalyzer;
import org.apache.lucene.document.Document;
import org.apache.lucene.index.DirectoryReader;
import org.apache.lucene.index.IndexReader;
import org.apache.lucene.index.Term;
import org.apache.lucene.queryparser.classic.QueryParser;
import org.apache.lucene.search.*;
import org.apache.lucene.store.Directory;
```

```
import org.apache.lucene.store.FSDirectory;
import org.apache.lucene.util.BytesRef;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;

import java.nio.file.Paths;

public class IndexSearch {

    private Directory dir;
    private IndexReader reader;
    private IndexSearcher is;

    @Before
    public void setUp() throws Exception {
        dir=FSDirectory.open(Paths.get("/Users/jackiechan/Documents/index"));
        reader=DirectoryReader.open(dir);
        is=new IndexSearcher(reader);
    }

    @After
    public void tearDown() throws Exception {
        reader.close();
    }

    /**
     * 对特定项搜索
     *  按词条搜索—TermQuery
     *TermQuery是最简单、也是最常用的Query。TermQuery可以理解成为“词条搜索”，
     * 在搜索引擎中最基本的搜索就是在索引中搜索某一词条，而TermQuery就是用来完成这项工作
的。
     * 在Lucene中词条是最基本的搜索单位，从本质上来讲一个词条其实就是一个名/值对。
     * 只不过这个“名”是字段名，而“值”则表示字段中所包含的某个关键字。
     * @throws Exception
     */
    @Test
    public void testTermQuery()throws Exception{
        String searchField="name";
        String q="java";
        Term t=new Term(searchField,q);
        Query query=new TermQuery(t);
        TopDocs hits=is.search(query, 10);
        System.out.println("匹配 '"+q+"', 总共查询到"+hits.totalHits+"个文档");
        for(ScoreDoc scoreDoc:hits.scoreDocs){
            Document doc=is.doc(scoreDoc.doc);
            System.out.println(doc.get("name"));
        }
    }
```

```
    /**
     * “多条件查询”搜索–BooleanQuery
     * BooleanQuery也是实际开发过程中经常使用的一种Query。
     * 它其实是一个组合的Query，在使用时可以把各种Query对象添加进去并标明它们之间的逻辑关系。
     * 在本节中所讨论的所有查询类型都可以使用BooleanQuery综合起来。
     * BooleanQuery本身来讲是一个布尔子句的容器，它提供了专门的API方法往其中添加子句，
     * 并标明它们之间的关系，以下代码为BooleanQuery提供的用于添加子句的API接口：
     * @throws Exception
     */
    @Test
    public void testBooleanQuery()throws Exception{
        String searchField="name";
        String q1="java";
        String q2="spring";
        Query query1=new TermQuery(new Term(searchField,q1));
        Query query2=new TermQuery(new Term(searchField,q2));
        BooleanQuery.Builder  builder=new BooleanQuery.Builder();
        //  1. MUST和MUST：取得连个查询子句的交集。
        //  2. MUST和MUST_NOT：表示查询结果中不能包含MUST_NOT所对应得查询子句的检索结果。
        // 3. SHOULD与MUST_NOT：连用时，功能同MUST和MUST_NOT。
        // 4. SHOULD与MUST连用时，结果为MUST子句的检索结果,但是SHOULD可影响排序。
        // 5. SHOULD与SHOULD：表示“或”关系，最终检索结果为所有检索子句的并集。
        // 6. MUST_NOT和MUST_NOT：无意义，检索无结果。
        builder.add(query1, BooleanClause.Occur.MUST);
        builder.add(query2, BooleanClause.Occur.MUST);
        BooleanQuery  booleanQuery=builder.build();
        TopDocs hits=is.search(booleanQuery, 10);
        System.out.println("匹配 "+q1 +"And"+q2+", 总共查询到"+hits.totalHits+"个文档");
        for(ScoreDoc scoreDoc:hits.scoreDocs){
            Document doc=is.doc(scoreDoc.doc);
            System.out.println(doc.get("name"));
        }
    }



    /**
     * TermRangeQuery 范围查询
     *TermRangeQuery是用于字符串范围查询的，既然涉及到范围必然需要字符串比较大小，
     * 字符串比较大小其实比较的是ASC码值，即ASC码范围查询。
     * 一般对于英文来说，进行ASC码范围查询还有那么一点意义，
     * 中文汉字进行ASC码值比较没什么太大意义，所以这个TermRangeQuery了解就行，
     * 用途不太大，一般数字范围查询NumericRangeQuery用的比较多一点，
     * 比如价格，年龄，金额，数量等等都涉及到数字，数字范围查询需求也很普遍。
     * @throws Exception
```

```
     */
    @Test
    public void testTermRangeQuery()throws Exception{
        String searchField="price";
        String q="0000001----1000002";
        String lowerTermString = "0000001";
        String upperTermString = "1000003";
        /**
         * field  字段
         * lowerterm -范围的下端的文字
         *upperterm -范围的上限内的文本
         *includelower -如果真的lowerterm被纳入范围。
         *includeupper -如果真的upperterm被纳入范围。
         *https://yq.aliyun.com/articles/45353
         */
        Query query=new TermRangeQuery(searchField,new
BytesRef(lowerTermString),new BytesRef(upperTermString),true,true);
        TopDocs hits=is.search(query, 10);
        System.out.println("匹配 '"+q+"', 总共查询到"+hits.totalHits+"个文档");
        for(ScoreDoc scoreDoc:hits.scoreDocs){
            Document doc=is.doc(scoreDoc.doc);
            System.out.println(doc.get("fullPath"));
        }
    }


    /**
     * PrefixQuery  PrefixQuery用于匹配其索引开始以指定的字符串的文档。就是文档中存在
xxx%
     *
     * @throws Exception
     */
    @Test
    public void testPrefixQuery()throws Exception{
        String searchField="name";
        String q="ja";
        Term t=new Term(searchField,q);
        Query query=new PrefixQuery(t);
        TopDocs hits=is.search(query, 10);
        System.out.println("匹配 '"+q+"', 总共查询到"+hits.totalHits+"个文档");

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

    /**
     *  所谓PhraseQuery，就是通过短语来检索，比如我想查“big car”这个短语，
```

```
     *  那么如果待匹配的document的指定项里包含了"big car"这个短语，
     *  这个document就算匹配成功。可如果待匹配的句子里包含的是“big black car”，
     *  那么就无法匹配成功了，如果也想让这个匹配，就需要设定slop，
     *  先给出slop的概念：slop是指两个项的位置之间允许的最大间隔距离
     * @throws Exception
     */
    @Test
    public void testPhraseQuery()throws Exception{
        String searchField="name";
        String q1="java";
        String q2="spring";
        Term t1=new Term(searchField,q1);
        Term t2=new Term(searchField,q2);
        PhraseQuery.Builder builder=new PhraseQuery.Builder();
        builder.add(t1);
        builder.add(t2);
        builder.setSlop(10);//设置中间间隔的内容最大长度
        PhraseQuery query=builder.build();
        TopDocs hits=is.search(query, 10);
        System.out.println("匹配 '"+q1+q2+"之间的几个字段"+"，总共查询
到"+hits.totalHits+"个文档");

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


    /**
     * 相近词语的搜索—FuzzyQuery
     * FuzzyQuery是一种模糊查询，它可以简单地识别两个相近的词语。
     * @throws Exception
     */
    @Test
    public void testFuzzyQuery()throws Exception{
        String searchField="name";
        String q="js";
        Term t=new Term(searchField,q);
        Query query=new FuzzyQuery(t);
        TopDocs hits=is.search(query, 10);
        System.out.println("匹配 '"+q+"'，总共查询到"+hits.totalHits+"个文档");

        for(ScoreDoc scoreDoc:hits.scoreDocs){
            Document doc=is.doc(scoreDoc.doc);
            System.out.println(doc.get("name"));
        }
    }
```

```
    /**
     * 使用通配符搜索—WildcardQuery
     * Lucene也提供了通配符的查询，这就是WildcardQuery。
     * 通配符“?”代表1个字符，而“*”则代表0至多个字符。 通配符的完整内容代表绝对内容匹配 比如 java?? 只能匹配内容 java 开头然后后面有两个内容的
     * @throws Exception
     */
    @Test
    public void testWildcardQuery()throws Exception{
        String searchField="name";
        String q="java*";
        Term t=new Term(searchField,q);
        Query query=new WildcardQuery(t);
        TopDocs hits=is.search(query, 10);
        System.out.println("匹配 '"+q+"', 总共查询到"+hits.totalHits+"个文档");

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

    /**
     * 解析查询表达式
     * QueryParser实际上就是一个解析用户输入的工具，可以通过扫描用户输入的字符串，生成Query对象，以下是一个代码示例：
     * @throws Exception
     */
    @Test
    public void testQueryParser()throws Exception{
        Analyzer analyzer=new StandardAnalyzer(); // 标准分词器
        String searchField="name";
        String q="java";
        //指定搜索字段和分析器
        QueryParser parser=new QueryParser(searchField, analyzer);
        //用户输入内容
        Query query=parser.parse(q);
        TopDocs hits=is.search(query, 100);
        System.out.println("匹配 "+q+"查询到"+hits.totalHits+"个记录");
        for(ScoreDoc scoreDoc:hits.scoreDocs){
            Document doc=is.doc(scoreDoc.doc);
            System.out.println(doc.get("name"));
        }
    }

}
```