Lucene

全文检索技术

**讲师：传智.入云龙**

什么是全文检索

全文检索概念

全文检索是一种将文件中所有文本与**检索项**匹配的检索方法。它可以根据需要获得全文中有关章、节、段、句、词等信息。计算机程序通过扫描文章中的每一个词，对每一个词建立一个**索引**，指明该词在文章中出现的次数和位置，当用户查询时根据建立的索引查找，类似于通过字典的检索字表查字的过程。

经过几年的发展，全文检索从最初的字符串匹配程序已经演进到能对超大文本、语音、图像、活动影像等非结构化数据进行综合管理的大型软件。本教程只讨论文本检索。

**主要应用领域：搜索引擎（百度，搜狗）、站内搜索（微博搜索）、电商网站（京东，淘宝）**

全文检索和数据库like查询的区别

数据查询通常的做法是是通过数据库模糊匹配即Like '%keyword%'的方式，通过它和全文检索对比来分析数据库like模糊查询和全文检索的区别。

数据结构

#### 结构化数据

数据库中存储的数据是结构化数据，即行数据，可以用二维表结构来逻辑表达实现的数据，结构化数据是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

不方便用数据库二维逻辑表来表现的数据即称为非结构化数据，包括所有格式的办公文档、文本、图片、标准通用标记语言下的子集XML、HTML、各类报表、图像和音频/视频信息等等。

* 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。
* 半结构数据：就是介于完全结构化数据（如关系型数据库、面向对象数据库中的数据）和完全无结构的数据（如声音、图像文件等）之间的数据，HTML、XML文档就属于半结构化数据，数据的结构和内容混在一起，没有明显的区分。

搜索原理

#### 顺序扫描

数据库的like查询采用顺序扫描的方法匹配字符串，查找结构化数据中存在某字符串的记录，如下：

查询table表中title字段出现XXXX字符的记录。

select \* from table where title like ‘%XXXX%’

windows的搜索也是顺序扫描，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搜索，对于每一个文档，从头找到尾，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。

顺序扫描问题：如果要查询的目标数据源量大且内容多，采用顺序扫描方法查询过程较慢，比如你有一个几十G的硬盘，如果想在上面找到一个内容包含某字符串的文件，将会非常耗时。

什么时候使用顺序扫描？对于查询的目标数据源量小、内容少的情况时采用顺序扫描是很快的。

#### 全文检索

对于查询目标数据源量大且内容多时，特别是如果查询的数据源为非结构化数据，这时就要采用全文检索方法进行查询。

全文检索首先将要查询的目标数据源中的一部分信息提取出来，组成索引，通过查询索引达到搜索目标数据源的目的，所以速度较快。

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）。

如下图是对文件搜索的索引：

索引

lucene

java

apache

…

…

目标数据源

dataset

XXXXXX.doc、XXXXX.xls、XXXXXX.pdf

上图中，索引栏是从目标数据源中提取出来的词，在进行全文检索时是通过搜索索引（搜索索引中的词）从而找到索引对应的文件即目标数据源。

搜索效果

#### 匹配准确性

使用数据库like搜索关键字“java”会把“javascript”也查询出来，因为javascript和'%java%'匹配。

使用搜索引擎搜索关键字“java”不会把“javascript”查询出来，因为在对“javascript”创建索引时不会把“java”抽取出来放在索引中，而是把“javascript”当成一个整体放在索引中，在进行全文检索时根据“java”在索引中找不到，通过“javascript”是可以找到的。

#### 相关度排序

使用数据库like搜索关键字“java”，查询结果中不会把与关键字相关度最高的记录排在最前边，数据库的排序只能根据由高到低或按字母顺序排序。

使用搜索引擎搜索关键字“java”，查询结果中会把关键字相关度最高的记录排在最前边，在进行全文检索时会计算哪些记录与关键字的相关度最高，最高相关度的记录会排在前边。

#### 搜索速度

使用数据库like搜索，如果目标数据源记录多且内容大，查询速度慢。

使用搜索引擎搜索，速度非常快。

应用领域

#### 数据库like查询

对于数据量不大、数据结构固定的数据可采用关系数据库存储，通过关系数据库提供的模糊匹配方式查询用户需要的数据，比如学校的学生管理系统、企业人事管理系统等。

#### 全文检索

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。

# Lucene实现全文检索

什么是Lucene?

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。

Lucene与搜索引擎的区别

全文检索系统是按照全文检索理论建立起来的用于提供全文检索服务的软件系统。全文检索系统是一个可以运行的系统，包括建立索引、处理查询返回结果集、增加索引、优化索引结构等功能。例如：百度搜索、eclipse帮助搜索、淘宝网商品搜索。

搜索引擎是全文检索技术最主要的一个应用，例如百度。搜索引擎起源于传统的信息全文检索理论，即计算机程序通过扫描每一篇文章中的每一个词，建立以词为单位的倒排文件，检索程序根据检索词在每一篇文章中出现的频率和每一个检索词在一篇文章中出现的概率，对包含这些检索词的文章进行排序，最后输出排序的结果。全文检索技术是搜索引擎的核心支撑技术。

Lucene和搜索引擎不同，Lucene是一套用java写的全文检索的工具包，为应用程序提供了很多个api接口去调用，可以简单理解为是一套实现全文检索的类库，搜索引擎是一个全文检索系统，它是一个单独运行的软件。

安装Lucene

Lucene是开发全文检索功能的工具包，从官方网站下载Lucene4.10.3，并解压。

官方网站：<http://lucene.apache.org/>

版本：lucene4.10.3

Jdk要求：1.7以上

Lucene主要包结构

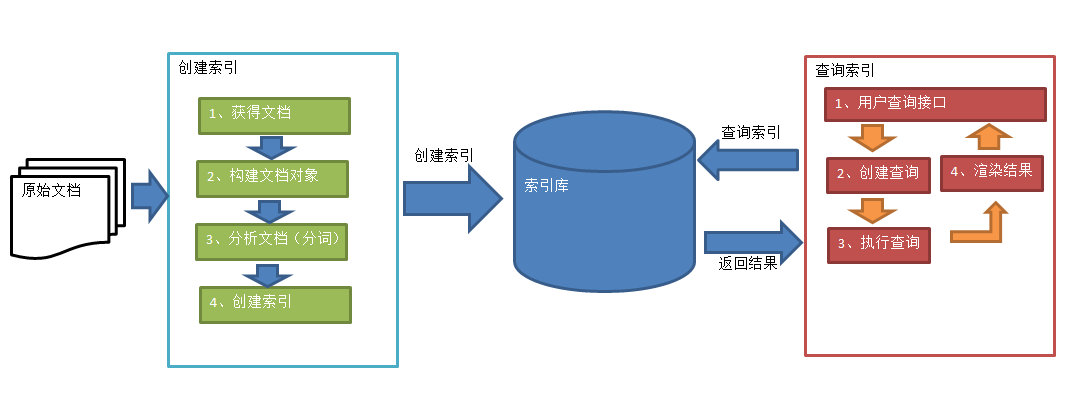
|  |  |
| --- | --- |
| 包名 | 功能 |
| org.apache.lucene.analysis | 语言分析器，主要用于的切词  Lucene提供的分析器实现类在：  lucene-analyzers-common-4.10.3.jar |
| org.apache.lucene.document | 索引存储时的文档结构管理，类似于关系型数据库的表结构 |
| org.apache.lucene.index | 索引管理，包括索引建立、删除等 |
| org.apache.lucene.queryParser | 查询分析器，实现查询关键词间的运算，如与、或、非等, 生成查询表达式， |
| 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，包括一些I/O操作 |
| org.apache.lucene.util | 公用类 |

案例描述

我们以一个案例来研究全文检索系统架构：实现一个资源管理器的搜索功能，通过关键字搜索文件，凡是文件名或文件内容包括关键字的文件都需要找出来。

根据上边的架构图分别从索引和搜索两个过程来描述实现过程。

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

索引流程

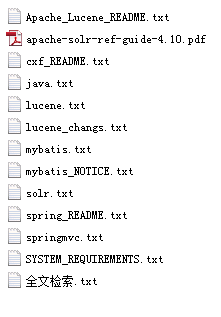
对文档索引的过程，将用户要搜索的文档内容进行索引，索引存储在索引库（index）中。

这里我们要搜索的文档是磁盘上的文本文件，根据案例描述：凡是文件名或文件内容包括关键字的文件都要找出来，这里要对文件名和文件内容创建索引。

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。

本案例中的原始内容就是磁盘上的文件，如下图：



获取原始内容（信息采集）

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：

Solr（<http://lucene.apache.org/solr>） ，solr是apache的一个子项目，支持从关系数据库、xml文档中提取原始数据。

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。

本案例我们要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。

创建文档

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：

Document（文档）

Field（域）

Name：file\_name（文件名称）

Value：XXXXXX.txt

Field（域）

Name：file\_path（文件路径）

Value：e:/…/XXXXXX.txt

Field（域）

Name：file\_content（文件内容）

Value：XXXXXXXXXXXXXXXX

Field（域）

Name：file\_size（文件大小）

Value：346543

其它Field.。。。。。。。。。。。。

**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）**

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、去除常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。。。。

索引文档

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：

Field域

index索引（词典）

Document文档

Document1(springmvc.txt)

Document2(mybatis.txt)

Document3(solr.txt)

springmvc.txt

web

spring

…..

mybatis.txt

file\_name

file\_path

file\_content

根据左边的索引词典可以找到词对应的文档。“springmvc.txt”这个词在Document1(springmvc.txt) ，“web”和“spring”在Document1、Document2中都存在，词是通过Field和Document文档联系起来的。

**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。**

搜索流程

搜索就是用户输入关键字，从索引（index）中进行搜索的过程。根据关键字搜索索引，根据索引找到对应的文档，从而找到要搜索的内容（这里指磁盘上的文件）。

用户

用户可以是自然人，也可以是远程调用的程序。

用户搜索界面

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。

比如：



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。

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，比如：

语法 “fileName:spring.txt”表示要搜索Field域的内容为“spring.txt”的文档

语法 “lucene AND java” 表示要搜索即包括关键字“lucene”也包括“java”的文档。

Lucene语句介绍：<http://lucene.apache.org/core/3_0_3/queryparsersyntax.html>

执行搜索

搜索索引过程：

1.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“lucene AND java”表示搜索出的文档中即要包括lucene也要包括java。

词lucene

文档lucene.txt

文档lucene\_changes.txt

.....

词java

文档solr.txt

文档lucene.txt

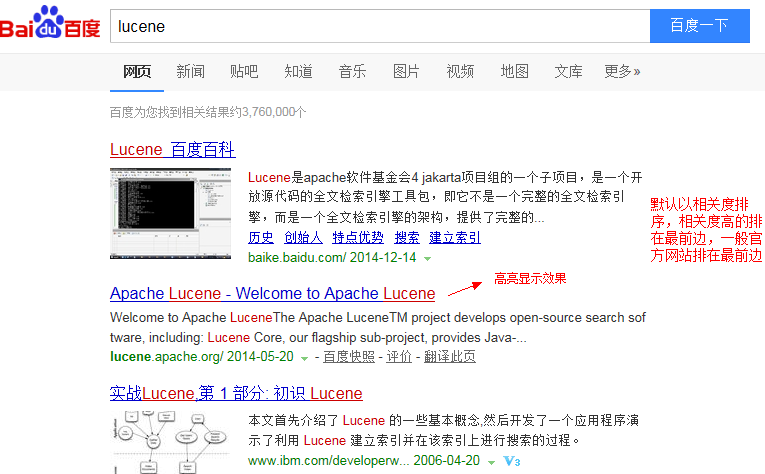
.....

2、由于是AND，所以要对包含lucene或java词语的链表进行交集，得到文档链表应该包括每一个搜索词语

3、获取文档中的Field域数据。

展示结果

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。



Lucene 入门程序

需求

实现一个歌词搜索系统，通过关键字搜索，凡是文件名或文件内容包括关键字的文件都要找出来。

注意：该入门程序只对文本文件(.txt)搜索。

开发环境

Jdk：1.7.0\_72

开发工具：eclipse indigo

Lucene包：

lucene-core-4.10.3.jar

lucene-analyzers-common-4.10.3.jar

lucene-queryparser-4.10.3.jar

其它：

commons-io-2.4.jar

junit-4.9.jar

Index索引

创建数据源目录

创建数据源目录F:\develop\lucene\searchsource ，该目录存放要搜索的原始文件。

创建索引目录

创建索引目录 F:\develop\lucene\indexdata，该目录存放创建的索引文件。

Document和Field

索引的目的是为了搜索，索引前要对搜索的内容创建成Document和Field。

Document

Document文档是Lucene搜索的单位，最终要将文档中的数据展示给用户，文档中包括多个Field，在设计文档对象时，可以将一个Document对应关系数据库的一条记录，字段对应Field，但是要注意：Document结构属于NOSql（非关系），不同的Document包括不同的Field，建议将相同类型数据的Document保持Field一致，方便管理维护，避免将不同类型Field数据融合在一个Document中。

比如：

文件信息Document，Field包括：文件名称、文件类型、文件大小、文件路径、文件内容等，避免加入非文件信息Field。

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。

1. 是否分析(tokenized)

是：将Field值分析出语汇单元即作分词处理，将词进行索引

比如：商品名称、商品简介等，这些内容用户要输入关键字搜索，由于搜索的内容格式大、内容多需要分析后将语汇单元索引。

否：不作分词处理

比如：商品id、订单号、身份证号等

1. 是否索引(indexed)

是：将Field分析后的词或整个Field值进行索引，只有索引方可搜索到。

比如：商品名称、商品简介分析后进行索引，订单号、身份证号不用分析但也要索引，这些将来都要作为查询条件。

否：不索引无法搜索到

比如：商品id、文件路径、图片路径等，不用作为查询条件的不用索引。

1. 是否存储(stored)

是：将Field值存储在文档中，存储在文档中的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，不存储的Field无法通过Document获取

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

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  是否分析 | 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，但是不会进行分析，会将整个串存储在索引中，比如(订单号,姓名等)  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，进行分析和索引，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. |

IndexWriter和Directory

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）。

索引程序代码

分析：

确定文档的各各域的属性和类型：

文件名称：不分析、索引、存储，使用StringField方法

文件路径：不分析、不索引、存储，使用StoredField方法

文件大小：分析、索引、存储，使用LongField方法

文件内容：分析，索引，不存储，使用TextField方法

代码如下：

**public** **class** IndexTest {

// 索引源，即源数据目录

**private** **static** String *searchSource* = "F:\\develop\\lucene\\searchsource";

// 索引目标地址

**private** **static** String *indexFolder* = "F:\\develop\\lucene\\indexdata";

@Test

**public** **void** testCreateIndex() {

**try** {

//从目录中读取文件内容并创建Document文档

List<Document> docs = IndexUtils.*file2Document*(*searchSource*);

//创建分析器，standardAnalyzer标准分析器

Analyzer standardAnalyzer = **new** StandardAnalyzer();

// 指定索引存储目录

Directory directory = FSDirectory.*open*(**new** File(*indexFolder*));

//创建索引操作配置对象

IndexWriterConfig indexWriterConfig = **new** IndexWriterConfig(Version.*LUCENE\_4\_10\_3*,

standardAnalyzer);

// 定义索引操作对象indexWriter

IndexWriter indexWriter = **new** IndexWriter(directory,indexWriterConfig);

// 遍历目录 下的文件生成的文档，调用indexWriter方法创建索引

**for** (Document document : docs) {

indexWriter.addDocument(document);

}

// 索引操作流关闭

indexWriter.close();

} **catch** (IOException e) {

e.printStackTrace();

}

}

}

IndexUtils.file2Document()方法定义如下：

// 从文件创建Document

**public** **static** List<Document> file2Document(String folderPath)

**throws** IOException {

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

File folder = **new** File(folderPath);

**if** (!folder.isDirectory()) {

**return** **null**;

}

// 获取目录 中的所有文件

File[] files = folder.listFiles();

**for** (File file : files) {

//文件名称

String fileName = file.getName();

System.*out*.println(fileName);

**if** (fileName.lastIndexOf(".txt") > 0) {

// 文件内容

String fileContent = FileUtils.*readFileToString*(file);

//文件路径

String filePath = file.getAbsolutePath();

//文件大小

**long** fileSize = FileUtils.*sizeOf*(file);

//创建文档

Document doc = **new** Document();

//创建各各Field域

//文件名

Field field\_fileName = **new** StringField("fileName", fileName, Store.*YES*);

//文件内容

Field field\_fileContent = **new** TextField("fileContent", fileContent, Store.*NO*);

//文件大小

Field field\_fileSize = **new** LongField("fileSize", fileSize, Store.*YES*);

//文件路径

Field field\_filePath = **new** StoredField("filePath", filePath, Store.*YES*);

//将各各Field添加到文档中

doc.add(field\_fileName);

doc.add(field\_fileContent);

doc.add(field\_fileSize);

doc.add(field\_filePath);

list.add(doc);

}

}

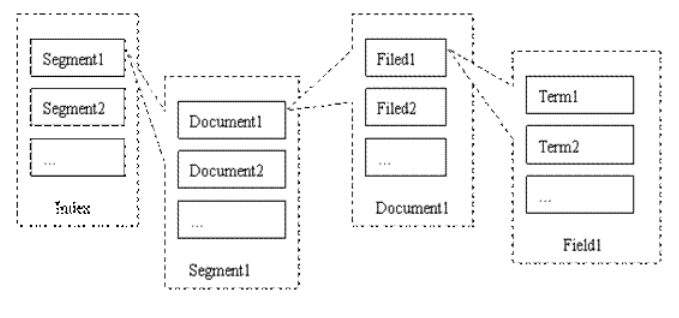
**return** list;

}

## Search搜索

索引结构

Lucene的索引结构为倒排索引，倒排文件或倒排索引是指索引对象是文档或者文档集合中的单词等，用来存储这些单词在一个文档或者一组文档中的存储位置，是对文档或者文档集合的一种最常用的索引机制。



Lucene索引index由若干段(segment)组成，每一段由若干的文档（document）组成，每一个文档由若干的域（field）组成，每一个域由若干的项（term）组成。

项是最小的索引单位，如果Field进行分析，可能会分析出多个语汇单元（词），每个词就是一个Term项，如果Field不进行分析，整个Field就是一个Term项。

项直接代表了一个字符串以及其在文件中的位置、出现次数等信息。域将项和文档进行关联。

使用Luke查看索引

Luke作为Lucene工具包中的一个工具（http://www.getopt.org/luke/），用于查询、修改lucene的索引文件。

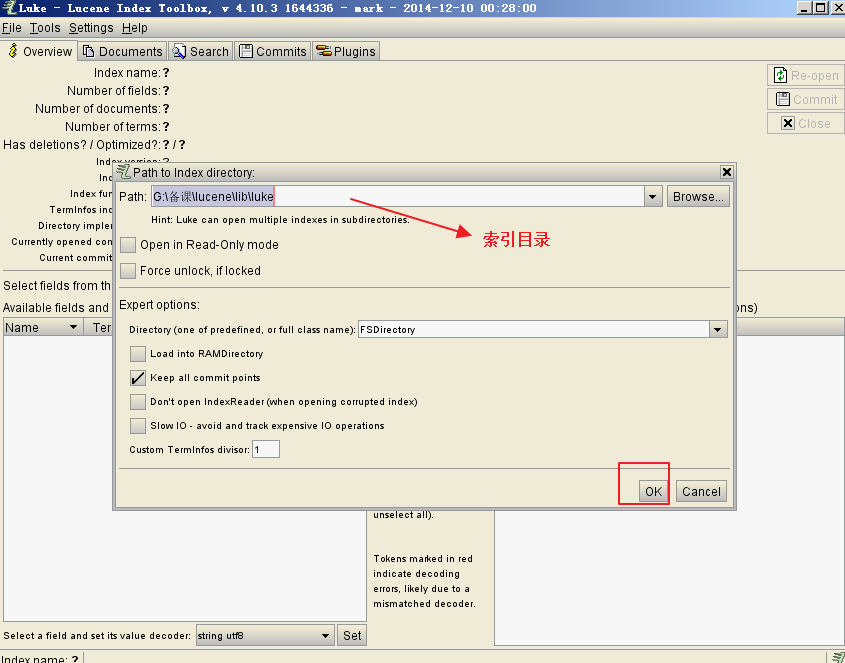
打开Luke方法：

cmd运行：java -jar lukeall-4.10.3.jar

如果需要加载第三方分词器，如果需要加载第三方分词器，需通过java.ext.dirs加载jar包:

可简单的将第三方分词器和lukeall放在一块儿，cmd下运行：。

java -Djava.ext.dirs=. -jar lukeall-4.10.3.jar

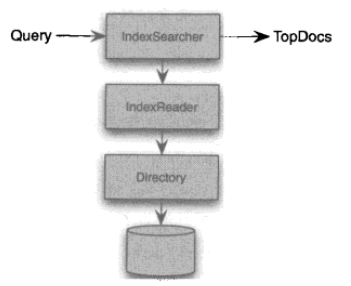


确定索引目录

搜索是要从索引中查找，确定索引目录即上边创建的索引目录 F:\develop\lucene\indexdata。

### IndexSearcher和IndexReader

通过IndexSearcher执行搜索，构建IndexSearcher需要IndexReader读取索引目录，如下图：



代码如下：

// 指定 目录

File folder = **new** File(indexFolder);

Directory directory = FSDirectory.*open*(folder);

// indexReader

IndexReader indexReader = DirectoryReader.*open*(directory);

// 定义indexSearcher

IndexSearcher indexSearcher = **new** IndexSearcher(indexReader);

需要注意：

Indexreader打开需要耗费很大的系统资源，建议使用一个IndexReader，如果索引进行添加、修改或删除需要打开新的Reader才可以搜索到。

IndexSearcher搜索方法如下：

|  |  |
| --- | --- |
| 方法 | 说明 |
| indexSearcher.search(query, n) | 根据Query搜索，返回评分最高的n条记录 |
|  |  |
|  |  |
|  |  |
| indexSearcher.search(query, filter, n) | 根据Query搜索，添加过滤策略，返回评分最高的n条记录 |
| indexSearcher.search(query, n, sort) | 根据Query搜索，添加排序策略，返回评分最高的n条记录 |
| indexSearcher.search(booleanQuery, filter, n, sort) | 根据Query搜索，添加过滤策略，添加排序策略，返回评分最高的n条记录 |

搜索程序代码

**public** **class** SearchTest {

// 索引目录地址

**private** **static** String *indexFolder* = "F:\\develop\\lucene\\indexdata";

//查询方法

@Test

**public** **void** testTermQuery() **throws** IOException {

// 创建查询对象，根据文件名称域搜索匹配文件名称的文档

Query query = **new** TermQuery(**new** Term("fileName", "springmvc\_test.txt"));

// 指定索引目录

Directory directory = FSDirectory.*open*(**new** File(*indexFolder*));

// 定义IndexReader

IndexReader reader = DirectoryReader.*open*(directory);

// 创建indexSearcher

IndexSearcher indexSearcher = **new** IndexSearcher(reader);

// 执行搜索

TopDocs topDocs = indexSearcher.search(query, 100);

// 提取搜索结果

ScoreDoc[] scoreDocs = topDocs.scoreDocs;

System.*out*.println("共搜索到总记录数：" + topDocs.totalHits);

**for** (ScoreDoc scoreDoc : scoreDocs) {

// 文档id

**int** docID = scoreDoc.doc;

// 得到文档

Document doc = indexSearcher.doc(docID);

// 输出 文件内容

IndexUtils.*printDocumentOfFile*(doc);

}

}

IndexUtils.printDocumentOfFile()方法：

//从文档中读取文件内容

**public** **static** **void** printDocumentOfFile(Document doc){

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

System.*out*.println("文件名称 =" + doc.get("fileName"));

System.*out*.println("文件大小 =" + doc.get("fileSize"));

System.*out*.println("文件内容 =" + doc.get("fileContent"));

}

TopDocs

Lucene搜索结果可通过TopDocs遍历，TopDocs类提供了少量的属性，如下：

|  |  |
| --- | --- |
| 方法或属性 | 说明 |
| totalHits | 匹配搜索条件的总记录数 |
|  |  |
|  |  |
|  |  |
| scoreDocs | 顶部匹配记录 |

注意：

Search方法需要指定匹配记录数量n：indexSearcher.search(query, n)

TopDocs.totalHits：是匹配索引库中所有记录的数量

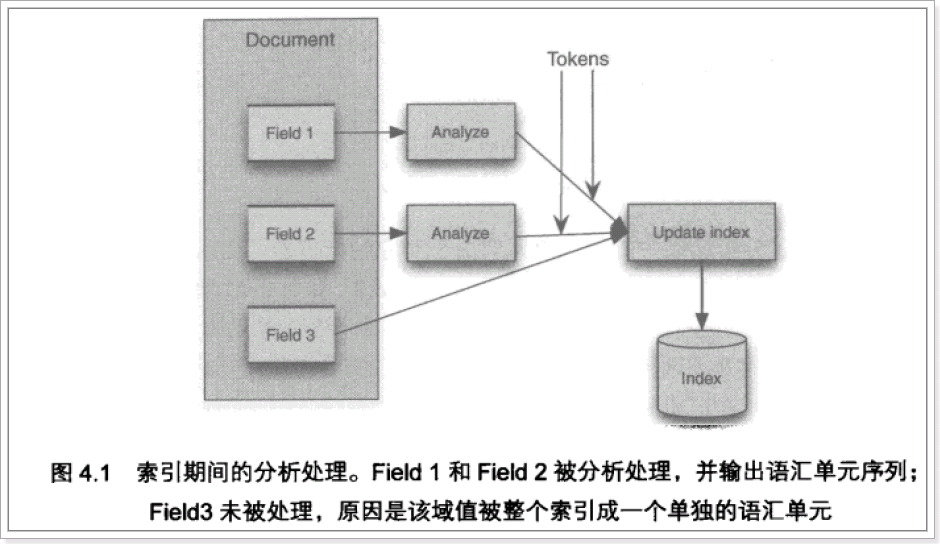
TopDocs.scoreDocs：匹配相关度高的前边记录数组，scoreDocs的长度小于等于search方法指定的参数n

Analyzer分析器

Analyzer使用时机

创建索引时使用Analyzer

输入关键字进行搜索，当需要让该关键字与文档域内容所包含的词进行匹配时需要对文档域内容进行分析，需要经过Analyzer分析器处理生成语汇单元（Token）。分析器分析的对象是文档中的Field域。当Field的属性tokenized（是否分词）为true时会对Field值进行分析，如下图：



对于一些Field可以不用分析：

1、不作为查询条件的内容，比如文件路径

2、不是匹配内容中的词而匹配Field的整体内容，比如订单号、身份证号等。

搜索时使用Analyzer

对搜索关键字进行分析和索引分析一样，使用Analyzer对搜索关键字进行分析、分词处理，使用分析后每个词语进行搜索。比如：搜索关键字：spring web ，经过分析器进行分词，得出：spring web拿词去索引词典表查找 ，找到索引链接到Document，解析Document内容。

对于匹配整体Field域的查询可以在搜索时不分析，比如根据订单号、身份证号查询等。

**注意：搜索使用的分析器要和索引使用的分析器一致。**

Analyzer执行过程

代码分析

Analyzer是一个抽象类，在Lucene的lucene-analyzers-common包中提供了很多分析器，比如：org.apache.lucene.analysis.standard.standardAnalyzer标准分词器，它是Lucene的核心分词器，它对分析文本进行分词、大写转成小写、去除停用词、去除标点符号等操作过程。

什么是停用词？停用词是为节省存储空间和提高搜索效率，搜索引擎在索引页面或处理搜索请求时会自动忽略某些字或词，这些字或词即被称为Stop Words(停用词)。比如语气助词、副词、介词、连接词等，通常自身并无明确的意义，只有将其放入一个完整的句子中才有一定作用，如常见的“的”、“在”、“是”、“啊”等。

如下是org.apache.lucene.analysis.standard.standardAnalyzer的部分源码：

final StandardTokenizer src = new StandardTokenizer(getVersion(), reader);//创建分词器

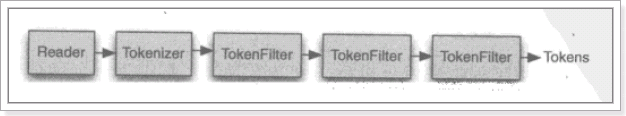
src.setMaxTokenLength(maxTokenLength);

TokenStream tok = new StandardFilter(getVersion(), src);//创建标准分词过滤器

tok = new LowerCaseFilter(getVersion(), tok);//在标准分词过滤器的基础上加大小写转换过滤

tok = new StopFilter(getVersion(), tok, stopwords);//在上边过滤器基础上加停用词过滤

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。

TokenStream

TokenStream是语汇单元流，tokenStream是一个抽象类，它是所有分析器的基类。如下图：

tokenStream

Tokenizer（分词器）

TokenFilter（分词过滤器）

Tokenizer是分词器，负责将reader转换为语汇单元即进行分词，Lucene提供了很多的分词器，也可以使用第三方的分词，比如IKAnalyzer一个中文分词器。

tokenFilter是分词过滤器，负责对语汇单元进行过滤，tokenFilter可以是一个过滤器链儿，Lucene提供了很多的分词器过滤器，比如大小写转换、去除停用词等。

从TokenStream中获取语汇单元信息，如下代码：

//创建分析器

Analyzer analyzer = **new** StandardAnalyzer();

//得到TokenStream

TokenStream tokenStream = analyzer.tokenStream("content", **new** StringReader("Lucene is a Java full-text search engine"));

//设置tokenStream初始状态，否则会抛异常

tokenStream.reset();

//设置要获取分词的偏移量

OffsetAttribute offsetAttribute = tokenStream.addAttribute(OffsetAttribute.**class**);

//设置要获取分词的项

CharTermAttribute charTermAttribute = tokenStream.addAttribute(CharTermAttribute.**class**);

**while**(tokenStream.incrementToken()){

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

//起始偏移量

System.*out*.print("-->"+offsetAttribute.startOffset());

//截止偏移量

System.*out*.print("-->"+offsetAttribute.endOffset());

//分词项的值

System.*out*.println("-->"+**new** String(charTermAttribute.toString()));

}

中文分词器

什么是中文分词器

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。而中文则以字为单位，字又组成词，字和词再组成句子。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；但中文“我爱中国”就不一样了，电脑不知道“中国”是一个词语还是“爱中”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。我爱中国，分词的结果是：我 爱 中国。

Lucene自带中文分词器

* StandardAnalyzer：

单字分词：就是按照中文一个字一个字地进行分词。如：“我爱中国”，  
效果：“我”、“爱”、“中”、“国”。

* CJKAnalyzer

二分法分词：按两个字进行切分。如：“我是中国人”，效果：“我是”、“是中”、“中国”“国人”。

上边两个分词器无法满足需求。

* SmartChineseAnalyzer

对中文支持稍好，但扩展性差，扩展词库，禁用词库和同义词库等不好处理

第三方产品介绍

* ik-analyzer：

IK Analyzer是一个开源的，基于java语言开发的轻量级的中文分词工具包。从2006年12月推出1.0版开始， IKAnalyzer已经推出了4个大版本。最初，它是以开源项目Luence为应用主体的，结合词典分词和文法分析算法的中文分词组件。从3.0版本开始，IK发展为面向Java的公用分词组件，独立于Lucene项目，同时提供了对Lucene的默认优化实现。在2012版本中，IK实现了简单的分词歧义排除算法，标志着IK分词器从单纯的词典分词向模拟语义分词衍化。

* mmseg4j

用 Chih-Hao Tsai 的 MMSeg 算法(http://technology.chtsai.org/mmseg/)实现的中文分词器，并实现 lucene 的 analyzer 和 solr 的TokenizerFactory 以方便在Lucene和Solr中使用。MMSeg 算法有两种分词方法：Simple和Complex，都是基于正向最大匹配。Complex 加了四个规则过滤。官方说：词语的正确识别率达到了 98.41%。mmseg4j 已经实现了这两种分词算法。

* imdict-chinese-analyzer

imdict-chinese-analyzer 是 imdict智能词典 的智能中文分词模块，算法基于隐马尔科夫模型(Hidden Markov Model, HMM)，是中国科学院计算技术研究所的ictclas中文分词程序的重新实现（基于Java），可以直接为lucene搜索引擎提供简体中文分词支持。

* 庖丁解牛分词

PaodingAnalzyer，最后版本是2.0.4，更新时间是2008-06-03，不支持Lucene3.0，目前已不再更新。

使用中文分词器

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。

测试代码如下：

//创建分析器

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

//得到TokenStream

TokenStream tokenStream = analyzer.tokenStream("content", **new** StringReader("我是中国人，我爱传智播客！"));

自定义词库

在classpath下定义IKAnalyzer.cfg.xml文件，如下：

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

<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">

<properties>

<comment>IK Analyzer 扩展配置</comment>

<!-- 用户可以在这里配置自己的扩展字典 -->

<entry key=*"ext\_dict"*>dicdata/mydict.dic</entry>

<!-- 用户可以在这里配置自己的扩展停用词字典 -->

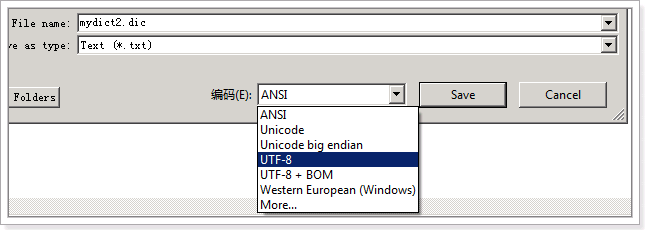
<entry key=*"ext\_stopwords"*>dicdata/ext\_stopword.dic</entry>

</properties>

在classpath下的编辑dicdata/mydict.dic文件，此文件中存储扩展词库，在dicdata/ext\_stopword.dic文件中存放停用词。

注意：mydict.dic和ext\_stopword.dic文件的格式为UTF-8，注意是无BOM 的UTF-8 编码。

使用EditPlus.exe保存为无BOM 的UTF-8 编码格式，如下图：



# 索引维护

添加索引

步骤：

1）创建存放索引的目录Directory

2）创建索引器配置管理类IndexWriterConfig

3）使用索引目录和配置管理类创建索引器

4）使用索引器将Document写到索引文件中

代码：

// 定义索引存储目录

Directory directory = FSDirectory.open(new File(indexFolder));

//标准分析器

Analyzer standardAnalyzer = new StandardAnalyzer();

IndexWriterConfig indexWriterConfig = new IndexWriterConfig(Version.LUCENE\_4\_10\_3,

standardAnalyzer);

// 定义索引操作对象

IndexWriter indexWriter = new IndexWriter(directory,indexWriterConfig);

//遍历目录下的文档，创建Document

List<Document> docs = IndexUtils.file2Document(indexSource);

// 遍历目录 下的文件生成的文档，调用indexWriter方法创建索引

for (Document document : docs) {

indexWriter.addDocument(document);

}

// 索引操作流关闭

indexWriter.close();

删除索引

1）删除符合条件的索引

// 索引目录

Directory directory = FSDirectory.open(new File(indexFolder));

// 定义索引操作对象

IndexWriter indexWriter = new IndexWriter(directory,

new IndexWriterConfig(Version.LUCENE\_4\_10\_3,

new StandardAnalyzer()));

// 删除索引，文件名称等于“springmvc.txt”的索引删除

indexWriter.deleteDocuments(new Term("fileName", "springmvc.txt"));

//索引删除后无法恢复

indexWriter.commit();

// 关闭索引操作流

indexWriter.close();

说明：根据Term项删除索引，满足条件的将全部删除，建议参照关系数据库基于主键删除方式，所以在创建索引时需要创建一个主键Field，删除时根据此主键Field删除。

索引删除后将放在Lucene的回收站中，Lucene3.X版本可以恢复删除的文档，3.X之后无法恢复。

2）删除全部索引

// 索引目录

Directory directory = FSDirectory.open(new File(indexFolder));

// 定义索引操作对象

IndexWriter indexWriter = new IndexWriter(directory,

new IndexWriterConfig(Version.LUCENE\_4\_10\_3,

new StandardAnalyzer()));

//删除全部索引

indexWriter.deleteAll();

说明：将索引目录的索引信息全部删除，直接彻底删除，无法恢复。

更新索引

// 索引目录

Directory directory = FSDirectory.open(new File(indexFolder));

// 定义索引操作对象

IndexWriter indexWriter = new IndexWriter(directory,

new IndexWriterConfig(Version.LUCENE\_4\_10\_3,

new StandardAnalyzer()));

// 创建更新文档

Document doc\_update = new Document();

Field field\_fileName = new TextField("fileName", "springmvc.txt",

Store.YES);

doc\_update.add(field\_fileName);

// 根据条件更新索引,将fileName等于“springmvc\_test.txt”的更新为上边的文档

// 更新的机制 是先删除再添加，如果原根据fileName没有找到“springmvc\_test.txt”，则添加新文档

indexWriter.updateDocument(new Term("fileName", "springmvc\_test.txt"),

doc\_update);

//提交

indexWriter.commit();

//关闭

indexWriter.close();

说明：更新索引是先删除再添加，建议对更新需求采用此方法并且要保证对已存在的索引执行更新，可以先查询出来，确定更新记录存在执行更新操作。

Query和QueryParser

创建查询的两种方法

创建查询

对要搜索的信息创建Query查询对象，Lucene会根据Query查询对象生成最终的查询语法，类似关系数据库Sql语法一样Lucene也有自己的查询语法，比如：“name:lucene”表示查询Field的name为“lucene”的文档信息。

可通过两种方法创建查询对象：

1）使用Lucene提供Query子类

Query是一个抽象类，lucene提供了很多查询对象，比如TermQuery项精确查询，NumericRangeQuery数字范围查询等。

如下代码：

Query query = **new** TermQuery(**new** Term("name", "lucene"));

2）使用QueryParse解析查询表达式

QueryParse会将用户输入的查询表达式解析成Query对象实例。

如下代码：

QueryParser queryParser = **new** QueryParser("name", **new** IKAnalyzer());

Query query = queryParser.parse("name:lucene");

通过Query 搜索

TermQuery

TermQuery，通过项查询，TermQuery不使用分析器所以建议匹配不分词的Field域查询，比如订单号、分类ID号等。

//创建查询对象

Query query = new TermQuery(new Term("fileName", "springmvc\_test.txt"));

//搜索索引 目录

Directory directory = FSDirectory.open(new File(indexFolder));

//定义IndexReader

IndexReader reader = DirectoryReader.open(directory);

//创建indexSearcher

IndexSearcher indexSearcher = new IndexSearcher(reader);

//执行搜索

TopDocs topDocs = indexSearcher.search(query, 100);

NumericRangeQuery

NumericRangeQuery，指定数字范围查询，如下：

//文件大小在0到1024的文件

NumericRangeQuery<Long> numericRangeQuery = NumericRangeQuery

.*newLongRange*("fileSize", 0l, 1024l, **true**, **true**);

BooleanQuery

BooleanQuery，布尔查询，实现组合条件查询，如下：

// 数字范围查询

NumericRangeQuery<Long> numericRangeQuery = NumericRangeQuery

.newLongRange("fileSize", 0l, 1024l, true, true);

//定义Boolean查询

BooleanQuery booleanQuery = new BooleanQuery();

//必须满足fileSize在0到1024范围的条件

booleanQuery.add(numericRangeQuery, Occur.MUST);

//根据文件名搜索

Query query = new TermQuery(new Term("fileName", "springmvc.txt"));

//不查询文件名为“springmvc\_test.txt”

booleanQuery.add(query, Occur.MUST);

IndexReader reader = DirectoryReader.open(directory);

IndexSearcher searcher = new IndexSearcher(reader);

TopDocs topDocs = searcher.search(booleanQuery, 100);

说明：

Occur.MUST 查询条件必须满足，相当于and

Occur.SHOULD 查询条件可选，相当于or

Occur.MUST\_NOT 查询条件不能满足，相当于not非

MatchAllDocsQuery

使用MatchAllDocsQuery查询索引目录中的所有文档，如果只存在过滤而没有查询条件时可以使用此类定义Query，如下：

//查询全部文档

Query query = new MatchAllDocsQuery();

通过QueryParser搜索

通过QueryParser也可以创建Query，QueryParser提供一个Parse方法，此方法可以直接根据查询语法来查询。Query对象执行的查询语法可通过System.out.println(query);查询。

**Lucene查询语法官方介绍：**

<http://lucene.apache.org/core/4_0_0/queryparser/org/apache/lucene/queryparser/classic/package-summary.html>

QueryParser查询

QueryParser使用方法：

//f是默认搜索的域

QueryParser queryParser = **new** QueryParser("f", analyzer);

// 指定查询语法 ，如果不指定fileName就搜索默认的域

Query query2 = queryParser.parse("fileName:springmvc.txt");

或：

Query query2 = queryParser.parse("spring AND web");

上边介绍的基于类的查询方法，使用QueryParser可用下边的查询语法 实现，

* 项查询：

FieldName : value

* 范围查询：

FieldName:[min TO max]

注意：QueryParse不支持对数字范围的搜索，它支持字符串范围。数字范围搜索建议使用NumericRangeQuery。

* 组合查询：

上边BooleanQuery例子的查询表达式如下：

+fileSize:[0 TO 1024] +fileName:springmvc.txt

上边的表达式表示fileSize的大小在0和1024之间且fileName为“springmvc.txt”，必须满足的条件使用+（加号）表示。

与BooleanQuery中Occur对应的符号如下：

|  |  |
| --- | --- |
| Occur.MUST 查询条件必须满足，相当于and | +（加号） |
| Occur.SHOULD 查询条件可选，相当于or | 空（不用符号） |
| Occur.MUST\_NOT 查询条件不能满足，相当于not非 | -（减号） |

* 关键字查询

AND：关键字1 AND 关键字2

两个关键字都匹配上条件满足。

OR：关键字1 OR 关键字2

两个关键字匹配一个条件满足

NOT：关键字1 NOT 关键字2

关键字1满足，关键字2不满足

MultiFieldQueryParser组合域查询

通过MuliFieldQueryParse对多个域查询，比如商品信息查询，输入关键字需要从商品名称和商品内容中查询。

代码：

//设置组合查询域

String[] fields = {"fileName","fileContent"};

//创建查询解析器

QueryParser queryParser = new MultiFieldQueryParser(fields, new IKAnalyzer());

//查询文件名、文件内容中包括“java”关键字的文档

Query query = queryParser.parse("java");

# 相关度排序

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。

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，采用空间向量模型算法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)，搜索也是要从Term中搜索，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)

即有多少文档包含次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。

设置boost影响打分结果

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的Field域设置加权值高，在搜索时匹配到这个Field就可能排在前边。

lucene在执行搜索时对某个域进行加权，在进行组合域查询时，匹配到加权值高的域最后计算的相关度得分就高。

索引时设置boost加权值

如果希望某些文档更重要，当此文档中包含所要查询的词则应该得分较高，这样相关度排序可以排在前边，可以在创建索引时设定文档中某些域（Field）的boost值来实现，如果不进行设定，则Field Boost默认为1.0f。一旦设定，除非删除此文档，否则无法改变。

代码：

field. setBoost(XXXf); XXX即权值。

测试：

可以将springmvc.txt的file\_content加权值设置为10.0f，结果搜索spring时如果内容可以匹配到关键字就可以把springmvc.txt文件排在前边。

代码：

索引时设置boost加权值：

//设置加权值

**if**(file\_name.equals("springmvc.txt")){

//设置比默认值 1.0大的

field\_file\_content.setBoost(20.0f);

}

**if**(file\_name.equals("spring\_README.txt")){

//设置比默认值 1.0大的

field\_file\_content.setBoost(30.0f);

}

//向文档中添加Field

document.add(field\_file\_content);

搜索时：

// 设置组合查询域，如果匹配到一个域就返回记录

String[] fields = { "file\_content" };

//设置评分,文件名称中包括关键字的评分高

/\*Map<String,Float> boosts = new HashMap<String,Float>();

boosts.put("file\_content", 3.0f);\*/

// 创建查询解析器

QueryParser queryParser = **new** MultiFieldQueryParser(fields,

**new** StandardAnalyzer());

// 查询文件名、文件内容中包括“java”关键字的文档

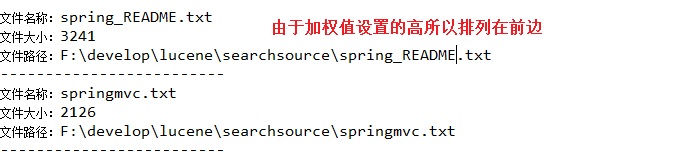
Query query = queryParser.parse("spring");

TopDocs topDocs = indexSearcher.search(query, 100);

ScoreDoc[] scoreDocs = topDocs.scoreDocs;

结果：

springmvc.txt排在最前边



搜索时设置boost加权值

在执行搜索时对某个域进行加权，在进行组合域查询时，匹配到加权值高的域最后计算的相关度得分就高。通常把标题、书名等域的加权值设置高点。

//设置组合查询域

String[] fields = {"file\_name","file\_content"};

//设置评分,文件名称中包括关键字的评分高

Map<String,Float> boosts = new HashMap<String,Float>();

boosts.put("file\_name", 10.0f);

//创建查询解析器

QueryParser queryParser = new MultiFieldQueryParser(fields, new IKAnalyzer(), boosts);

//查询文件名、文件内容中包括“springmvc.txt”关键字的文档，由于设置了文件名称域的加权值高，所以名称中匹配到关键字的应该排在前边

Query query = queryParser.parse("springmvc.txt");

测试：

搜索springmvc.txt，由于文件名中为springmvc.txt应该让它排在前边，内容中有springmvc.txt应该排在后边。

springmvc.txt：

测试结果它应该排在前边

spring\_README.txt

内容中包括springmvc.txt，也能匹配到但要排在后边。

代码：

//设置评分,文件名称中包括关键字的评分高

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

boosts.put("file\_name", 30.0f);

// 创建查询解析器

QueryParser queryParser = **new** MultiFieldQueryParser(fields,

**new** StandardAnalyzer(),boosts);

// 查询文件名、文件内容中包括“java”关键字的文档

Query query = queryParser.parse("springmvc.txt");

TopDocs topDocs = indexSearcher.search(query, 100);

ScoreDoc[] scoreDocs = topDocs.scoreDocs;

测试结果：

