Lucene

(讲师：程道)

主要课程内容

Lucene 是一个开源的 高性能的 信息搜索库（IR Information Retrieval Library）

- **第一部分：Lucene基础回顾**

  传统搜索的问题 改进之后的搜索 全文数据 全文数据的查询方法 全文检索和倒排索引 Lucene 相关产品 特性 逻辑模块

- **第二部分：Lucene 应用实战**

  索引创建和搜索的流程 索引创建和搜索的代码实现 luke工具 Field域的使用 索引的维护 分词器

  Query子类查询 TermQuery BooleanQuery 范围查询 短语查询 跨度查询 模糊查询 数值查询

  QueryParser MultiFieldQueryParser StandardQueryParser

- **第三部分：Lucene 底层高级**

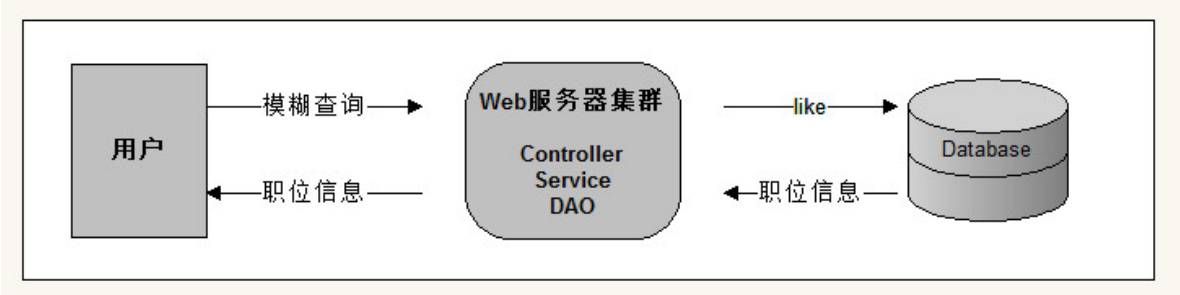
  底层的存储结构 相关度排序 相关度排序公式 VSM BM25 Lucene优化 使用的注意事项

一.Lucene基础回顾

1.数据检索的问题

原始方式实现搜索功能

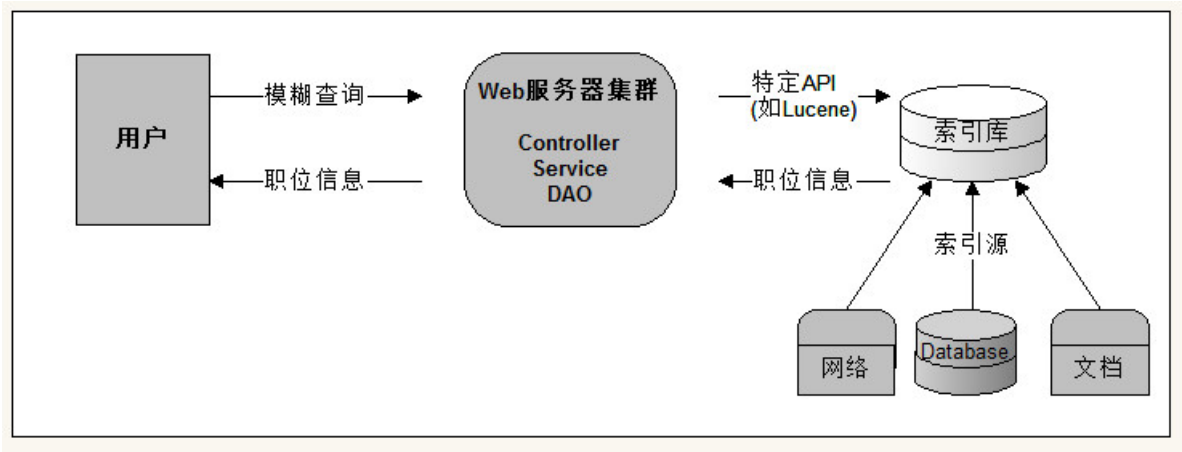
流程如下图：



上图就是原始搜索技术，如果用户比较少而且数据库的数据量比较小，那么这种方式实现搜索功能在企业中是没有问题的。但是数据量过多时，数据库的压力就会变得很大，查询速度会变得非常慢。我们需要使用更好的解决方案来分担数据库的压力。

改进后的查询

为了解决数据库压力和速度的问题，我们的数据库就变成了索引库，我们使用Lucene的API的来操作服务器上的索引库。这样完全和数据库进行了隔离。



全文数据

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，如数据库中的数据，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。
```

非结构化数据又一种叫法叫全文数据。

数据库适合结构化数据的精确查询，而不适合半结构化、非结构化数据的模糊查询及灵活搜索（特别是数据量大时），无法提供想要的实时性。

2.全文数据查询方法

顺序扫描法

所谓顺序扫描，就是要找内容包含一个字符串的文件，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。
这种方法是顺序扫描方法，数据量大就搜索慢。

全文检索

全文检索是指计算机索引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，检索程序就根据事先建立的索引进行查找，并将查找的结果反馈给用户的检索方式。这个过程类似于通过字典中的检索字表查字的过程。

全文检索的基本思路，就是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，然后对这个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) 。

具体应用的有 单机软件的搜索（word中的搜索） 站内搜索 （ 京东、 taobao、拉勾职位搜索） 专业搜索引擎公司 （ google、baidu ）的搜索

全文检索 与 倒排索引

全文检索通常使用倒排索引（inverted index）来实现。倒排索引同B树索引一样，也是一种索引结构。其中存储了单词与单词自身在一个或多个文档中所在位置之间的映射。

正排索引(forward index)：

正排索引是指文档ID为key，表中记录每个关键词出现的次数 位置等，查找时扫描表中的每个文档中字的信息，直到找到所有包含查询关键字的文档。

“文档1”的ID > 单词1：出现次数，出现位置列表；单词2：出现次数，出现位置列表；.....。

“文档2”的ID > 此文档出现的关键词列表。

正常的索引一般是指关系型数据库里的索引。 把不同的数据存放到不同的字段中。如果要实现baidu或google那种搜索，就需要与一条记录的多个字段进行比对，需要 全表扫描，如果数据量比较大的话，性能就很低。

当用户在主页上搜索关键词“华为手机”时，假设只存在正向索引（forward index），那么就需要扫描索引库中的所有文档，找出所有包含关键词“华为手机”的文档，再根据打分模型进行打分，排出名次后呈现给用户。因为互联网上收录在搜索引擎中的文档的数目是个天文数字，这样的索引结构根本无法满足实时返回排名结果的要求。

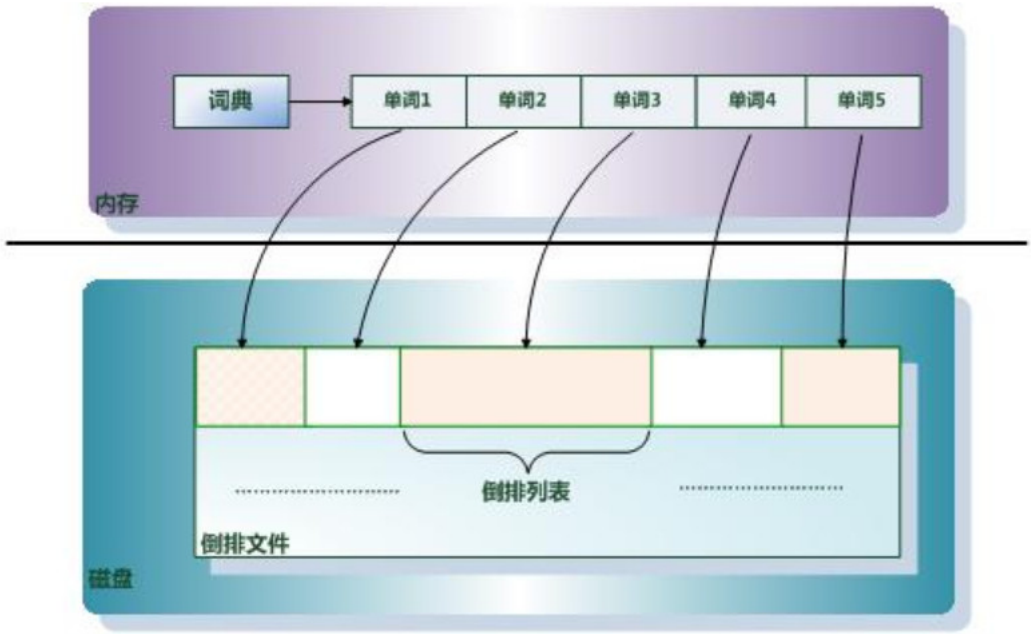
倒排索引(Inverted Index)：

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。

“关键词1”：“文档1”的ID 出现次数 出现的位置，“文档2”的ID 出现次数 出现的位置，.....。

“关键词2”：带有此关键词的文档ID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.Lucene 简介

什么是Lucene

Lucene的作者Doug Cutting是资深的全文索引/检索专家，最开始发布在他本人的主页上，2000年开源，2001年10月贡献给APACHE，成为APACHE基金的一个子项目。官网https://lucene.apache.org/core。现在是开源全文检索方案的重要选择。
Lucene是非常优秀的成熟的开源的免费的纯java语言的全文索引检索工具包。
Lucene是一个高性能、可伸缩的信息搜索(IR)库。 Information Retrieval (IR) library.它可以为你的应用程序添加索引和搜索能力。
Lucene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，或者是以此为基础建立起完整的全文检索引擎。由Apache软件基金会支持和提供,Lucene提供了一个简单却强大的应用程序接口，能够做全文索引和搜索。Lucene是当前以及最近几年非常受欢迎的免费Java信息检索程序库。

Lucene的实现产品

作为一个开放源代码项目，Lucene从问世之后，引发了开放源代码社群的巨大反响，程序员们不仅使用它构建具体的全文检索应用，而且将之集成到各种系统软件中去，以及构建Web应用，甚至某些商业软件也采用了Lucene作为其内部全文检索子系统的核心。

Nutch：Apache顶级开源项目，包含网络爬虫和搜索引擎(基于lucene)的系统（同 百度、google）。Hadoop因它而生。

Solr : Lucene下的子项目，基于Lucene构建的独立的企业级开源搜索平台，一个服务。它提供了基于xml/JSON/http的api供外界访问，还有web管理界面。

Elasticsearch：基于Lucene的企业级分布式搜索平台，它对外提供restful-web接口，让程序员可以轻松、方便使用搜索平台。

还有大家所熟知的OSChina、Eclipse、MyEclipse、JForum等等都是使用了Lucene做搜索框架来实现自己的搜索部分内容，在我们自己的项目中很有必要加入他的搜索能力，可以大大提高我们开发系统的搜索体验度。

Lucene 特性

1、稳定、索引性能高

每小时能够索引150GB以上的数据。

对内存的要求小 只需要1MB的堆内存

增量索引和批量索引一样快。

索引的大小约为索引文本大小的20%~30%。

2、高效、准确、高性能的搜索算法

范围搜索 - 优先返回最佳结果很多强大的

良好的搜索排序。

强大的查询方式支持：短语查询、通配符查询、临近查询、范围查询等。

支持字段搜索（如标题、作者、内容）。

可根据任意字段排序

支持多个索引查询结果合并

支持更新操作和查询操作同时进行

支持高亮、join、分组结果功能

速度快

可扩展排序模块，内置包含向量空间模型、BM25模型可选

可配置存储引擎

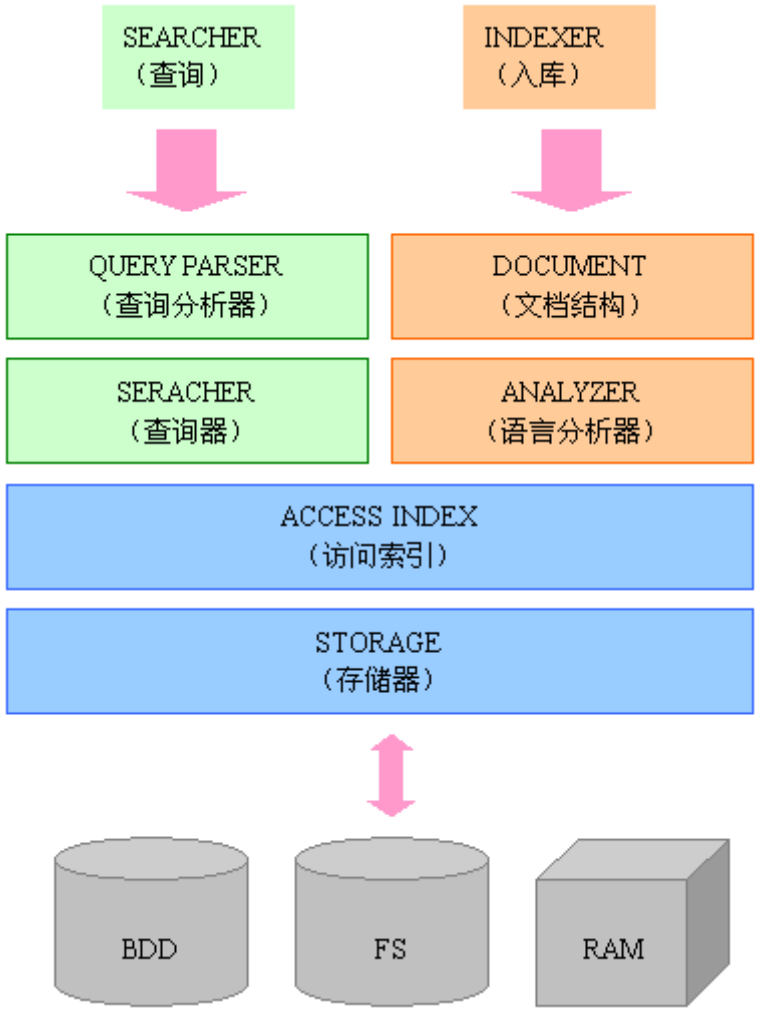
3、跨平台

纯java编写。

作为Apache开源许可下的开源项目，你可在商业或开源项目中使用。

Lucene有多种语言实现版可选(如C、C++、Python等），不只是JAVA。

Lucene模块构成

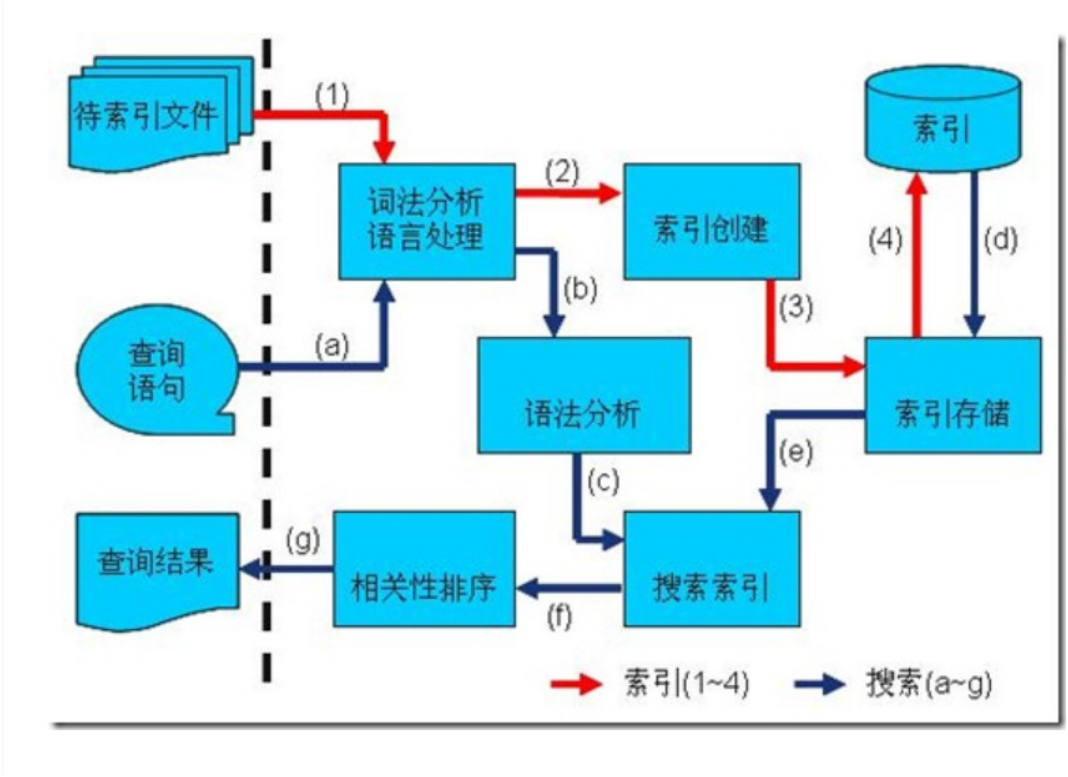


Lucene是一个用Java写的高性能、可伸缩的全文检索引擎工具包，它可以方便的嵌入到各种应用中实现针对应用的全文索引/检索功能。Lucene的目标是为各种中小型应用程序加入全文检索功能。

```
Lucene 的index模块主要负责索引的创建，里面有IndexWriter。
Lucene 的search模块主要负责对索引的搜索。
Lucene 的QueryParser主要负责语法分析。
Lucene 的Document相当于一个要进行索引的单元，任何可以想要被索引的文件都必须转化为Document对象才能进行索引。代表一个虚拟文档与字段，其中字段是可包含在物理文档的内容，元数据等对象。
Lucene 的analysis 模块主要负责词法分析及语言处理而形成Term。
Lucene 的store模块主要负责索引的读写。
```

二.Lucene 应用实战

1 索引创建和搜索流程



索引创建流程

第一步：一些要索引的原文档(Document)数据

采集数据分类：
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。

比如我们需要分析的数据内容是:

```
Lucene Core is a Java library providing powerful indexing and search features,
as well as spellchecking, hit highlighting and advanced analysis/tokenization
capabilities. The PyLucene sub project provides Python bindings for Lucene Core.
```

```
Solr is highly scalable, providing fully fault tolerant distributed indexing,
search and analytics. It exposes Lucene's features through easy to use JSON/HTTP
interfaces or native clients for Java and other languages.
```

```
The Apache Hadoop software library is a framework that allows for the
distributed processing of large data sets across clusters of computers using
simple programming models.
```

第二步 创建文档对象 进行词法分析 语言处理 将原文档传给分词器(Tokenizer) 形成一系列词(Term)

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。每个Document可以有多个Field。

每个文档都有一个唯一的编号，就是文档id

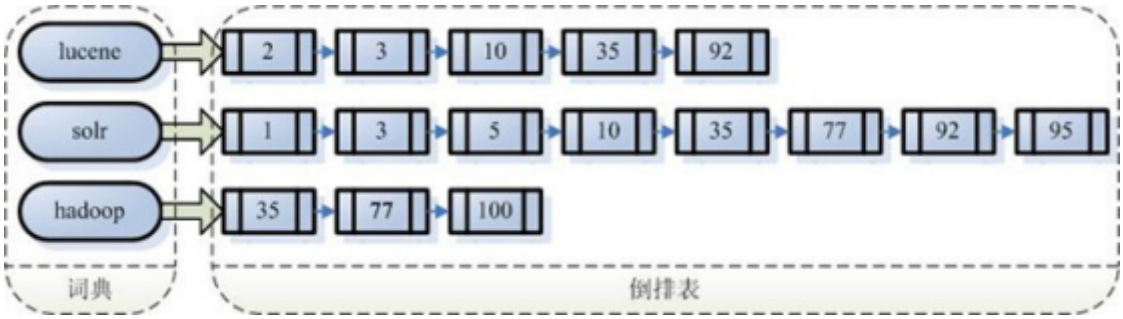
当然这里传给分词器的操作在addDocument后会自行调用 ，将原始内容创建为包含域（Field）的文档（document），需要再对域中的内容进行分析，分析成为一个一个的单词(Term)。

第三步: 索引创建 将得到的词(Term)传给索引组件(Indexer) 形成倒排索引结构

对所有文档分析得出的词汇单元进行索引，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）。

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。

倒排索引结构是根据内容（词汇）找文档，如下图：



第四步: 通过索引存储器 将索引写入到磁盘

搜索过程

查询索引一般会经过以下步骤：

a) 用户输入查询语句

b) 对查询语句经过词法分析和语言分析得到一系列词(Term)

c) 通过语法分析得到一个查询树

d) 通过索引存储将索引读到内存

e) 利用查询树搜索索引，从而得到每个词(Term)的文档列表，对文档列表进行交、差、并得到结果文档

f) 将搜索到的结果文档按照对查询语句的相关性进行排序

g) 返回查询结果给用户

2.Lucene 索引创建和搜索实现

```
 <dependencies>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.8.2</version>
        </dependency>

        <dependency>
            <groupId>org.apache.lucene</groupId>
            <artifactId>lucene-core</artifactId>
            <version>7.7.3</version>
        </dependency>

        <dependency>
            <groupId>org.apache.lucene</groupId>
            <artifactId>lucene-queryparser</artifactId>
            <version>7.7.3</version>
        </dependency>

        <dependency>
            <groupId>org.apache.lucene</groupId>
            <artifactId>lucene-analyzers-common</artifactId>
            <version>7.7.3</version>
        </dependency>

        <dependency>
            <groupId>commons-io</groupId>
            <artifactId>commons-io</artifactId>
            <version>2.5</version>
        </dependency>

    </dependencies>
```

```
package com.lagou.bean;

public class Book {
    @Override
```

```
    public String toString() {
        return "Book{" +
                "id=" + id +
                ", name='" + name + '\'' +
                ", price=" + price +  '\'' +
                ", desc='" + desc + '\'' +
                '}';
    }

    public Book() {
    }

    public Book(Integer id, String name, Float price, String desc) {
        this.id = id;
        this.name = name;
        this.price = price;
        this.desc = desc;
    }

    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 getDesc() {
        return desc;
    }

    public void setDesc(String desc) {
        this.desc = desc;
    }

        // 图书ID
        private Integer id;
        // 图书名称
        private String name;
        // 图书价格
        private Float price;
```

```
        // 图书描述
        private String desc;


}
```

java代码实现 索引创建

```
    public void testCreateIndex() throws Exception {
        // 1. 采集数据
        List<Book> bookList = new ArrayList<Book>();
        Book  booka  = new Book();
        booka.setId(1);
        booka.setDesc("Lucene Core is a Java library providing powerful indexing
and search features, as well as spellchecking, hit highlighting and advanced
analysis/tokenization capabilities. The PyLucene sub project provides Python
bindings for Lucene Core. ");
        booka.setName("Lucene");
        booka.setPrice(100.45f);
        bookList.add(booka);

        Book  bookb  = new Book();
        bookb.setId(11);
        bookb.setDesc("Solr is highly scalable, providing fully fault tolerant
distributed indexing, search and analytics. It exposes Lucene's features through
easy to use JSON/HTTP interfaces or native clients for Java and other languages.
");
        bookb.setName("Solr");
        bookb.setPrice(320.45f);
        bookList.add(bookb);
        Book  bookc  = new Book();
        bookc.setId(21);
        bookc.setDesc("The Apache Hadoop software library is a framework that
allows for the distributed processing of large data sets across clusters of
computers using simple programming models.");
        bookc.setName("Hadoop");
        bookc.setPrice(620.45f);
        bookList.add(bookc);
        // 2. 创建Document文档对象
        List<Document> documents = new ArrayList<>();
        for (Book book : bookList) {
            Document document = new Document();
            // Document文档中添加Field域
            // 图书Id    Store.YES:表示存储到文档域中
            document.add(new TextField("id", book.getId().toString(),
Field.Store.YES));
            document.add(new TextField("name", book.getName(),
                 Field.Store.YES));
            document.add(new TextField("price", book.getPrice().toString(),
                 Field.Store.YES));
            document.add(new TextField("desc", book.getDesc(),
                 Field.Store.YES));
            // 把Document放到list中
            documents.add(document);
```

```
        }
        // 3. 创建Analyzer分词器,分析文档，对文档进行分词
        Analyzer analyzer = new StandardAnalyzer();
        //  创建Directory对象,声明索引库的位置
        Directory directory = FSDirectory.open(Paths.get("D:/lucene/index"));
        //  创建IndexWriteConfig对象，写入索引需要的配置
        IndexWriterConfig config = new IndexWriterConfig(analyzer);
        // 4.创建IndexWriter写入对象  添加文档对象document
        IndexWriter indexWriter = new IndexWriter(directory, config);

        for (Document doc : documents) {
            indexWriter.addDocument(doc);
        }
        // 释放资源
        indexWriter.close();
    }
```

使用Luke查看索引

Luke作为Lucene工具包中的一个工具（https://github.com/DmitryKey/luke/releases），可以通过界面来进行索引文件的查询、修改

Lucene可以通过query对象输入查询语句。同数据库的sql一样，lucene也有固定的查询语法：

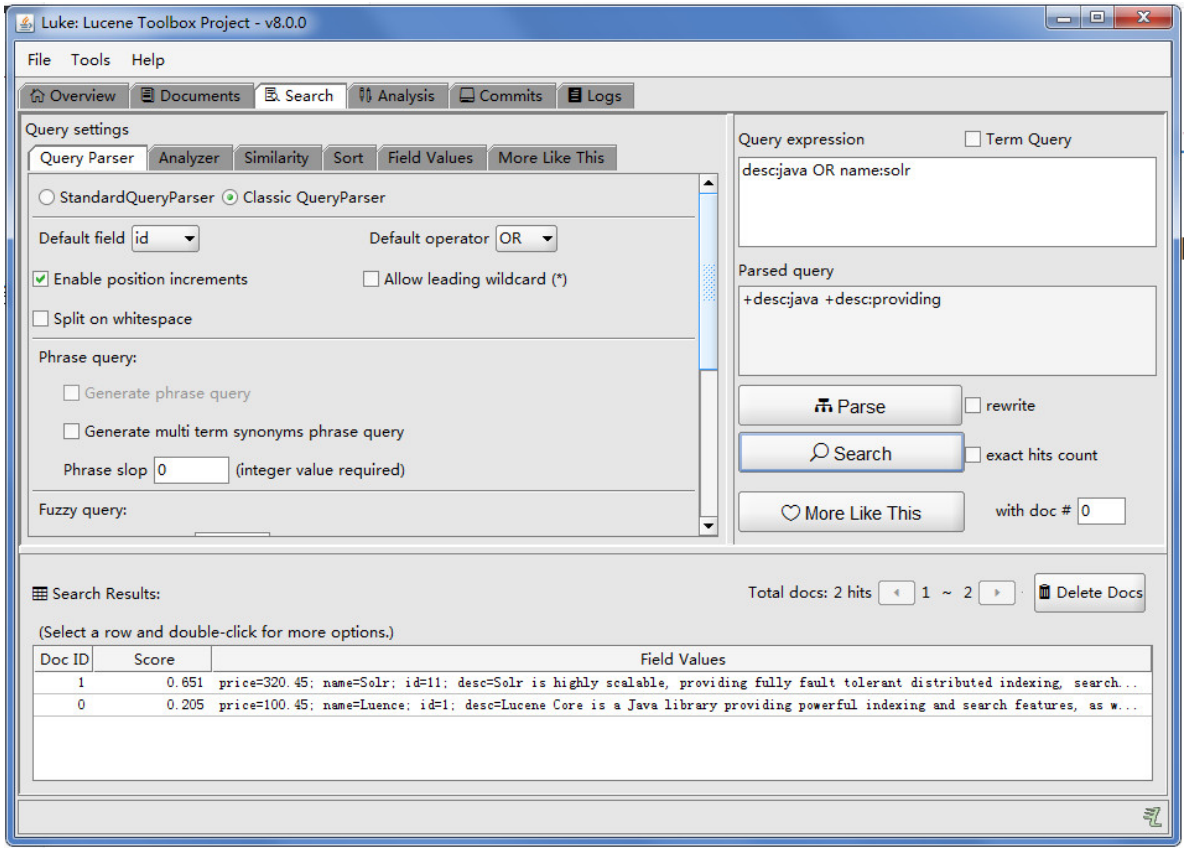
最基本的有比如：AND, OR, NOT 等（必须大写）

举个例子:

用户想找一个desc中包括java关键字和lucene关键字的文档。

它对应的查询语句：desc:java AND desc:lucene

如下图是使用luke搜索的例子



java代码实现 搜索

```
 @Test
public void testSearchIndex() throws Exception {
    // 1. 创建Query搜索对象
    // 创建分词器
    Analyzer analyzer = new StandardAnalyzer();
    // 创建搜索解析器，第一个参数：默认Field域，第二个参数：分词器
    QueryParser queryParser = new QueryParser("id", analyzer);
    // 创建搜索对象
    Query query = queryParser.parse("desc:java OR name:solr");
    // 2. 创建Directory流对象,声明索引库位置
    Directory directory = FSDirectory.open(Paths.get("D:/lucene/index"));
    // 3. 创建索引读取对象IndexReader
    IndexReader reader = DirectoryReader.open(directory);
    // 4. 创建索引搜索对象
    IndexSearcher searcher = new IndexSearcher(reader);
    // 5. 使用索引搜索对象，执行搜索，返回结果集TopDocs
    // 第一个参数：搜索对象，第二个参数：返回的数据条数，指定查询结果最顶部的n条数据返回
    TopDocs topDocs = searcher.search(query, 10);
    System.out.println("查询到的数据总条数是：" + topDocs.totalHits);
    // 获取查询结果集
    ScoreDoc[] docs = topDocs.scoreDocs;
    // 6. 解析结果集
    for (ScoreDoc scoreDoc : docs) {
        // 获取文档
        int docID = scoreDoc.doc;
        Document doc = searcher.doc(docID);
        System.out.println("============================");
        System.out.println("score:"+scoreDoc.score);
        System.out.println("docID:" + docID);
        System.out.println("bookId:" + doc.get("id"));
        System.out.println("name:" + doc.get("name"));
        System.out.println("price:" + doc.get("price"));
        System.out.println("desc:" + doc.get("desc"));
    }
    // 释放资源
    reader.close();
}
```

3.Field 域的使用

1.Field属性

Lucene存储对象是以Document为存储单元，对象中相关的属性值则存放到Field中。

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。

Field的三大属性：

```
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不固定、内容多需要分词后将语汇单元索引。
否：不作分词处理
比如：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，索引的目的是为了搜索。
比如：商品名称、商品简介分词后进行索引，订单号、身份证号不用分词但要索引，这些将来都要作为查询条件。
否：不索引。该域的内容无法搜索到
比如：文件路径、图片路径等，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。
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。

2.Field常用类型

Field对应的类是`org.apache.lucene.document.Field`，该类实现了`org.apache.lucene.document.IndexableField`接口，代表用于indexing的一个字段。Field类比较底层一些，所以Lucene实现了许多Field子类，用于不同的场景。

Field类型	数据类型	是否分词	是否索引	是否存储	说明
StringField(FieldName, FieldValue, Store.YES)	字符串	N	Y	Y/N	字符串类型Field, 不分词, 作为一个整体进行索引(如: 身份证号, 订单编号), 是否需要存储由Store.YES或Store.NO决定
TextField(FieldName, FieldValue, Store.NO)	文本类型	Y	Y	Y/N	文本类型Field, 分词并且索引, 是否需要存储由Store.YES或Store.NO决定
LongField(FieldName, FieldValue, Store.YES) 或 LongPoint(String name, int... point)等	数值型代表	Y	Y	Y/N	在Lucene 6.0中，LongField替换为LongPoint，IntField替换为IntPoint，FloatField替换为FloatPoint，DoubleField替换为DoublePoint。对数值型字段索引，索引不存储。要存储结合StoredField即可。
StoredField(FieldName, FieldValue)	支持多种类型	N	N	Y	构建不同类型的Field, 不分词, 不索引, 要存储. (如: 商品图片路径)

3.Field应用代码

```
@Test
```

```
public void createIndex() throws Exception {
    // 1.采集数据
    Book  booka  = new Book();
    List<Book> bookList = new ArrayList<Book>();
    booka.setId(1);
    booka.setDesc("Lucene Core is a Java library providing powerful indexing and
search features, as well as spellchecking, hit highlighting and advanced
analysis/tokenization capabilities. The PyLucene sub project provides Python
bindings for Lucene Core. ");
    booka.setName("Lucene");
    booka.setPrice(100.45f);
    bookList.add(booka);

    Book  bookb  = new Book();
    bookb.setId(11);
    bookb.setDesc("Solr is highly scalable, providing fully fault tolerant
distributed indexing, search and analytics. It exposes Lucene's features through
easy to use JSON/HTTP interfaces or native clients for Java and other languages.
");
    bookb.setName("Solr");
    bookb.setPrice(320.45f);
    bookList.add(bookb);
    Book  bookc  = new Book();
    bookc.setId(21);
    bookc.setDesc("The Apache Hadoop software library is a framework that allows
for the distributed processing of large data sets across clusters of computers
using simple programming models.");
    bookc.setName("Hadoop");
    bookc.setPrice(620.45f);
    bookList.add(bookc);

    // 2.将采集到的数据封装到Document对象中
    List<Document> docList = new ArrayList<>();
    Document document;
    for (Book book : bookList) {
        document = new Document();
        // IntPoint  分词 索引 不存储 存储结合  StoredField
        Field id = new IntPoint("id", book.getId());
        Field id_v  = new StoredField("id", book.getId());
        // 分词、索引、存储 TextField
        Field name = new TextField("name", book.getName(), Field.Store.YES);
        // 分词、索引、不存储 但是是数字类型，所以使用FloatPoint
        Field price = new FloatPoint("price", book.getPrice());
        // 分词、索引、不存储 TextField
        Field desc = new TextField("desc",
                                   book.getDesc(), Field.Store.NO);

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

        docList.add(document);
    }
    //3.创建Analyzer 分词器 对文档进行分词
    Analyzer  analyzer  = new StandardAnalyzer();
```

```
    // 创建Directory   和 IndexWriterConfig 对象
    Directory  directory = FSDirectory.open(Paths.get("D:/lucene/index3"));
    IndexWriterConfig indexWriterConfig = new IndexWriterConfig(analyzer);
    // 4.创建IndexWriter 写入对象
    IndexWriter  indexWriter = new IndexWriter(directory,indexWriterConfig);
    // 添加文档对象
    for (Document doc : docList) {
        indexWriter.addDocument(doc);
    }
    // 释放资源
    indexWriter.close();
}
```

4.索引库的维护

索引添加

indexWriter.addDocument(doc);

```
public void indexCreate() throws Exception {
    // 创建分词器
    Analyzer analyzer = new StandardAnalyzer();
    // 创建Directory流对象
    Directory directory = FSDirectory.open(Paths.get("D:/lucene/index3"));
    IndexWriterConfig config = new IndexWriterConfig(analyzer);
    // 创建写入对象
    IndexWriter indexWriter = new IndexWriter(directory, config);

    // 创建Document
    Document document = new Document();
    document.add(new TextField("id", "1001", Field.Store.YES));
    document.add(new TextField("name", "game", Field.Store.YES));
    document.add(new TextField("desc", "one world one dream", Field.Store.NO));
    // 添加文档 完成索引添加
    indexWriter.addDocument(document);

    // 释放资源
    indexWriter.close();
}
```

索引删除

根据Term项删除索引，满足条件的将全部删除。

indexWriter.deleteDocuments(new Term("name", "game"));

全部删除

indexWriter.deleteAll();

更新索引

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。

如果更新索引的目标文档对象不存在，则执行添加。

```
public void indexUpdate() throws Exception {
    // 创建分词器
    Analyzer analyzer = new StandardAnalyzer();
    // 创建Directory流对象
    Directory directory = FSDirectory.open(Paths.get("D:/lucene/index3"));
    IndexWriterConfig config = new IndexWriterConfig(analyzer);
    // 创建写入对象
    IndexWriter indexWriter = new IndexWriter(directory, config);

    // 创建Document
    Document document = new Document();
    document.add(new TextField("id", "1002", Field.Store.YES));
    document.add(new TextField("name", "lucene测试test update1002",
Field.Store.YES));

    // 执行更新，会把所有符合条件的Document删除，再新增。
    indexWriter.updateDocument(new Term("name", "test"), document);

    // 释放资源
    indexWriter.close();
}
```

5.分词器

分词器相关概念

分词器：采集到的数据会存储到Document对象的Field域中，分词器就是将Document中Field的value值切分成一个一个的词。

停用词:停用词是为节省存储空间和提高搜索效率，搜索程序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”、a、an、the 等。

扩展词: 扩展词 就是分词器默认不会切出的词 但我们希望分词器切出这样的词 。

过滤：包括去除标点符号过滤、去除停用词过滤（的、是、a、an、the等）、大写转小写、词的形还原（复数形式转成单数形参、过去式转成现在式。。。）等。

分词器案例

对于分词来说，不同的语言，分词规则不同。Lucene作为一个工具包提供不同国家的分词器，本例子使用StandardAnalyzer，它可以对用英文进行分词。

```
@Test
public void testCreateIndex() throws Exception {
    // 1. 采集数据   和 之前完全相同
```

```
    List<Book> bookList = new ArrayList<Book>();
    Book booka = new Book();
    booka.setId(1);
    booka.setDesc("Lucene is  java  full  text  search  lib");
    booka.setName("lucene");
    booka.setPrice(100.45f);
    bookList.add(booka);
    // 2. 创建Document文档对象
    List<Document> documents = new ArrayList<>();
    for (Book book : bookList) {
        Document document = new Document();
        // IntPoint  分词 索引 不存储 存储结合  StoredField
        Field id = new IntPoint("id", book.getId());
        Field id_v  = new StoredField("id", book.getId());
        // 分词、索引、存储 TextField
        Field name = new TextField("name", book.getName(), Field.Store.YES);
        // 分词、索引、不存储 但是是数字类型，所以使用FloatPoint
        Field price = new FloatPoint("price", book.getPrice());
        // 分词、索引、存储 TextField 为了看到分词效果设置成存储
        Field desc = new TextField("desc",
                                   book.getDesc(), Field.Store.YES);

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

        documents.add(document);
    }

    //3.创建StandardAnalyzer 分词器 对文档进行分词
    Analyzer  analyzer  = new StandardAnalyzer();
    // 创建Directory   和 IndexWriterConfig 对象
    Directory  directory = FSDirectory.open(Paths.get("D:/lucene/index5"));
    IndexWriterConfig indexWriterConfig = new IndexWriterConfig(analyzer);
    // 4.创建IndexWriter 写入对象
    IndexWriter  indexWriter = new IndexWriter(directory,indexWriterConfig);
    // 添加文档对象
    for (Document doc : documents) {
        indexWriter.addDocument(doc);
    }
    // 释放资源
    indexWriter.close();
}
```

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

```
protected TokenStreamComponents createComponents(String fieldName) {
        final StandardTokenizer src = new StandardTokenizer();
        src.setMaxTokenLength(this.maxTokenLength);
        TokenStream tok = new LowerCaseFilter(src);
        TokenStream tok = new StopFilter(tok, this.stopwords);
        return new TokenStreamComponents(src, tok) {
            protected void setReader(Reader reader) {
                src.setMaxTokenLength(StandardAnalyzer.this.maxTokenLength);
                super.setReader(reader);
            }
        };
    }
```

Tokenizer就是分词器，负责将reader转换为语汇单元即进行分词处理，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。

如下图是语汇单元的生成过程：

创建一个Tokenizer分词器，经过三个TokenFilter生成语汇单元Token。

Tokenizer --->TokenFilter(标准过滤)--->TokenFilter(大小写过滤)--->TokenFilter(停用词过滤)---->Tokens

比如下边的文档经过分析器分析如下：

原文档内容：

Lucene is java full text search lib

分析后得到的多个语汇单元：

lucene java full text search lib

注意:搜索使用的分词器要和索引使用的分词器一致。

中文分词器

什么是中文分词器

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“我爱拉勾”就不一样了，电脑不知道“拉勾”是一个词语还是“爱拉”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。我爱拉勾，分词的结果是：我、爱、我爱、拉勾。

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：“我是拉勾人啊”，
效果：“我”、“是”、“拉”、“勾”、“人”、“啊”。

CJKAnalyzer

二分法分词：按两个字进行切分。如：“我是拉勾人啊”，效果：“我是”、“是拉”、“拉勾”、 “勾人”、 “人啊”。

上边两个分词器无法满足需求。

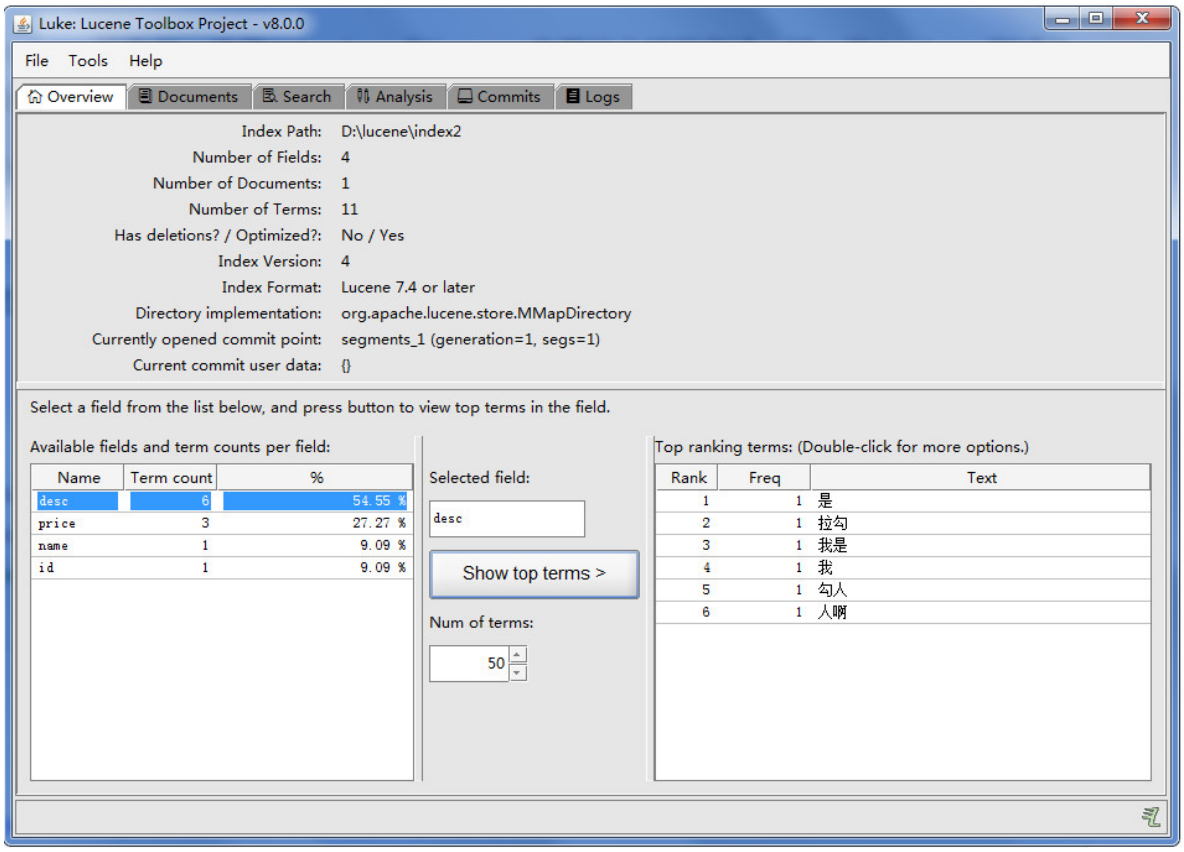
中文分词器IKAnalyzer

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。

我是拉勾人啊

分词结果:



中文分词器案例

```
<dependency>
    <groupId>com.github.magese</groupId>
    <artifactId>ik-analyzer</artifactId>
    <version>7.7.1</version>
</dependency>
```

```java
@Test
public void testCreateIndex() throws Exception {
    // 1. 采集数据  和 之前完全相同
    List<Book> bookList = new ArrayList<Book>();
    Book booka = new Book();
    booka.setId(1);
    booka.setDesc("我是拉勾人啊");
    booka.setName("lucene");
    booka.setPrice(100.45f);
    bookList.add(booka);
    // 2. 创建Document文档对象
    List<Document> documents = new ArrayList<>();
    for (Book book : bookList) {
        Document document = new Document();
        // IntPoint  分词 索引 不存储 存储结合  StoredField
        Field id = new IntPoint("id", book.getId());
        Field id_v  = new StoredField("id", book.getId());
        // 分词、索引、存储 TextField
        Field name = new TextField("name", book.getName(), Field.Store.YES);
        // 分词、索引、不存储 但是是数字类型，所以使用FloatPoint
        Field price = new FloatPoint("price", book.getPrice());
        // 分词、索引、存储 TextField 为了看到分词效果设置成存储
        Field desc = new TextField("desc",
                                   book.getDesc(), Field.Store.YES);

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

        documents.add(document);
    }

    // 3.创建Analyzer 分词器  把之前的StandardAnalyzer 换成 IKAnalyzer
    //Analyzer  analyzer  = new StandardAnalyzer();
    Analyzer analyzer = new IKAnalyzer();
    // 创建Directory   和 IndexWriterConfig 对象
    Directory  directory = FSDirectory.open(Paths.get("D:/lucene/index5"));
    IndexWriterConfig indexWriterConfig = new IndexWriterConfig(analyzer);
    // 4.创建IndexWriter 写入对象
    IndexWriter  indexWriter = new IndexWriter(directory,indexWriterConfig);
    // 添加文档对象
    for (Document doc : documents) {
        indexWriter.addDocument(doc);
    }
    // 释放资源
    indexWriter.close();
}
```

扩展中文词库

如果想配置扩展词和停用词，就创建扩展词的文件和停用词的文件。

注意：不要用window自带的记事本保存扩展词文件和停用词文件，那样的话，格式中是含有bom的。

从ikanalyzer包中拷贝配置文件 拷贝到资源文件夹中

IKAnalyzer.cfg.xml配置文件

```
<?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">ext.dic;</entry>

    <!--用户可以在这里配置自己的扩展停止词字典-->
    <entry key="ext_stopwords">stopword.dic;</entry>

</properties>
```

中文词库，添加新词的地方 ext.dic

stopword.dic是存放停用词的地方

最终分词效果 和之前的分词器对比

6.搜索实战

环境准备

用之前讲Field 的案例 来创建查询环境即可

为了搜索效果明显 我们把desc 也进行存储

创建查询的两种方式

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，WildcardQuery统配查询 各种Point的数字范围查询等。

2）使用QueryParser解析查询表达式

Lucene QueryPaser包中提供了两类查询解析器：

传统的解析器：QueryParser和MultiFieldQueryParser

基于新的 flexible 框架的解析器：StandardQueryParser

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：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

Query子类搜索

TermQuery

TermQuery词项查询，TermQuery不使用分词器，搜索关键词进行精确匹配Field域中的词，比如订单号、分类ID号等。

搜索对象创建：

```
@Test
public void testSearchTermQuery() throws Exception {
    // 创建TermQuery搜索对象
    Query query = new TermQuery(new Term("name", "lucene"));
    doSearch(query);
}
```

```
private void doSearch(Query query) throws Exception {
    //  执行搜索，返回结果集
    // 创建Directory流对象
    Directory directory = FSDirectory.open(Paths.get("D:/lucene/index"));

    // 创建索引读取对象IndexReader
    IndexReader reader = DirectoryReader.open(directory);
    // 创建索引搜索对象
    IndexSearcher searcher = new IndexSearcher(reader);
    // 使用索引搜索对象，执行搜索，返回结果集TopDocs
    // 第一个参数：搜索对象，第二个参数：返回的数据条数，指定查询结果最顶部的n条数据返回
    TopDocs topDocs = searcher.search(query, 10);
    System.out.println("查询到的数据总条数是：" + topDocs.totalHits);
    // 获取查询结果集
    ScoreDoc[] docs = topDocs.scoreDocs;

    // 解析结果集
    for (ScoreDoc scoreDoc : docs) {
        // 获取文档id
        int docID = scoreDoc.doc;
        Document doc = searcher.doc(docID);
        System.out.println("docID:" + docID);
        System.out.println("bookId:" + doc.get("id"));
        System.out.println("name:" + doc.get("name"));
        System.out.println("price:" + doc.get("price"));
        System.out.println("desc:" + doc.get("desc"));
    }
    reader.close();
}
```

BooleanQuery

BooleanQuery，布尔查询，实现组合条件查询。

```
@Test
public void testSearchBooleanQuery() throws Exception {
    // 创建两个 TermQuery搜索对象
    Query query1 = new TermQuery(new Term("name", "lucene"));
    Query query2 = new TermQuery(new Term("desc", "java"));
```

```
    // 创建BooleanQuery搜索对象,组合查询条件
    BooleanQuery.Builder  boolQuery = new BooleanQuery.Builder();
    // 组合条件,
    // 第一个参数，查询条件，第二个参数，组合方式
    boolQuery.add(query1, BooleanClause.Occur.SHOULD);
    boolQuery.add(query2, BooleanClause.Occur.SHOULD);

    doSearch(boolQuery.build());
}
```

组合关系代表的意思如下:

1、MUST和MUST表示“与”的关系，即“交集”。

2、MUST和MUST_NOT前者包含后者不包含。

3、MUST_NOT和MUST_NOT没意义

4、SHOULD与MUST表示MUST，SHOULD失去意义；

5、SHOULD与MUST_NOT相当于MUST与MUST_NOT。

6、SHOULD与SHOULD表示“或”的关系，即“并集”。

短语查询

```
PhraseQuery phraseQuery = new PhraseQuery("desc","lucene");
或者
PhraseQuery phraseQuery = new PhraseQuery(3,"desc","lucene","java");
```

跨度查询

```
SpanTermQuery tq1 = new SpanTermQuery(new Term("desc", "lucene"));
SpanTermQuery tq2 = new SpanTermQuery(new Term("desc", "java"));
SpanNearQuery  spanNearQuery = new SpanNearQuery(new SpanQuery[] { tq1, tq2
},4,true);
```

模糊查询

WildcardQuery：通配符查询， *表示0个或多个字符，?表示1个字符，\是转义符。通配符查询可能会比较慢，不可以通配符开头（那样就是所有词项了）

```
WildcardQuery wildcardQuery = new WildcardQuery( new Term("name", "so*"));
```

```
FuzzyQuery fuzzyQuery = new FuzzyQuery(new Term("name", "slors"), 2);
```

数值查询

通过 IntPoint, LongPoint, FloatPoint, DoublePoint 中的方法构建对应的查询。

```
Query pointRangeQuery = IntPoint.newRangeQuery("id", 1,11);
```

QueryParser搜索

查询语法

1、基础的查询语法，关键词查询：

域名+“:”+搜索的关键字

例如：name:java

2、范围查询

域名+“:”+[最小值 TO 最大值]

例如：size:[1 TO 1000]

注意：QueryParser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对应的Point。

3、组合条件查询

第一种写法:

1）+条件1 + 条件2：两个条件之间是并且的关系and

例如：+filename:lucene + content:lucene

2）+条件1 条件2：必须满足第一个条件，应该满足第二个条件

例如：+filename:lucene content:lucene

3）条件1 条件2：两个条件满足其一即可。

例如：filename:lucene content:lucene

4）-条件1条件2：必须不满足条件1，要满足条件2

例如：-filename:lucene content:lucene

逻辑	实现
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）

第二种写法：

条件1 AND 条件2

条件1 OR 条件2

条件1 NOT 条件2

QueryParser

```
@Test
```

```
public void testSearchIndex() throws Exception {
    // 创建分词器
    Analyzer analyzer = new StandardAnalyzer();
    // 1. 创建Query搜索对象
    // 创建搜索解析器，第一个参数：默认Field域，第二个参数：分词器
    QueryParser queryParser = new QueryParser("desc", analyzer);
    // 创建搜索对象
    Query query = queryParser.parse("desc:java AND name:lucene");

    // 打印生成的搜索语句
    System.out.println(query);
    // 执行搜索
    doSearch(query);
}
```

MultiFieldQueryParser

```
@Test
public void testSearchMultiFieldQueryParser() throws Exception {
    // 创建分词器
    Analyzer analyzer = new IKAnalyzer();
    // 1. 创建MultiFieldQueryParser搜索对象
    String[] fields = { "name", "desc" };
    MultiFieldQueryParser multiFieldQueryParser = new
MultiFieldQueryParser(fields, analyzer);
    // 创建搜索对象
    Query query = multiFieldQueryParser.parse("lucene");
    // 打印生成的搜索语句
    System.out.println(query);
    // 执行搜索
    doSearch(query);
}
```

生成的查询语句： name:lucene desc:lucene

StandardQueryParser

```
@Test
public void testStandardQueryParser() throws Exception {
    // 创建分词器
    Analyzer analyzer = new StandardAnalyzer();
    // 1. 创建Query搜索对象
    // 创建搜索解析器 传入分词器
    StandardQueryParser  parser  = new StandardQueryParser(analyzer);
    // 创建搜索对象
    Query query = parser.parse("desc:java AND name:lucene","desc");

    // 打印生成的搜索语句
    System.out.println(query);
    // 执行搜索
    doSearch(query);
}
```

StandardQueryParser 其它查询举例

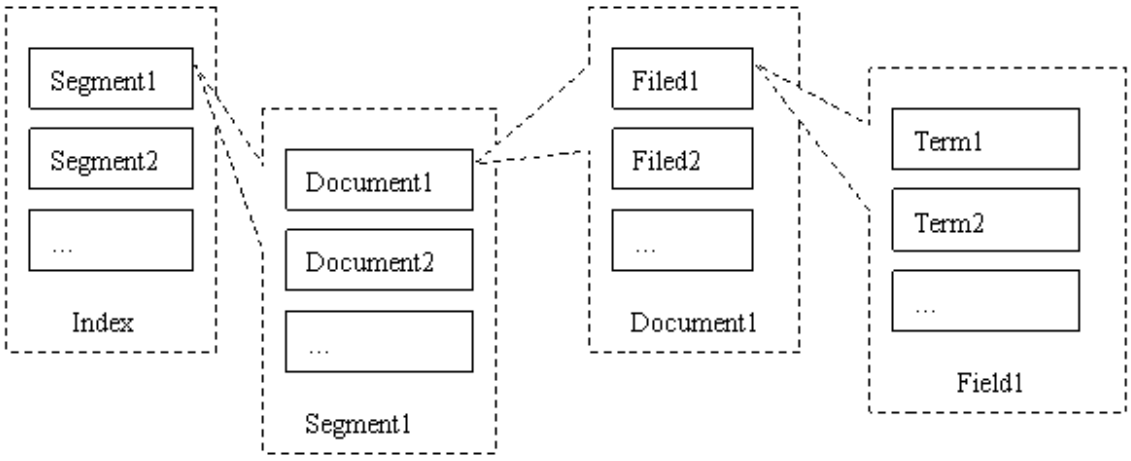
```
//通配符匹配  建议通配符在后  通配符在前效率低
query = parser.parse("name:L*","desc");
query = parser.parse("name:L???","desc");

//模糊匹配
query = parser.parse("lucene~","desc");
//区间查询
query = parser.parse("id:[1 TO 100]","desc");

//跨度查询  ~2表示词语之间包含两个词语
query= parser.parse("\"lucene java\"~2","desc");
```

第三部分 Lucene 底层高级

1.底层存储结构



Lucene的索引结构是有层次结构的，主要分以下几个层次：

索引(Index)：

一个目录一个索引，在Lucene中一个索引是放在一个文件夹中的。

同一文件夹中的所有的文件构成一个Lucene索引。

段(Segment)：

一个索引可以包含多个段，段与段之间是独立的，添加新文档可以生成新的段，不同的段可以合并。

在建立索引的时候对性能影响最大的地方就是在向索引写入文档的时候，所以在具体应用的时候就需要对

此加以控制，段(Segment) 就是实现这种控制的 。

Lucene默认情况是每加入10份文档(Document)就从内存往index文件写入并生成一个段(Segment) ，

然后每10个段(Segment)就合并成一个段(Segment). 这些控制的变量如下：

IndexWriter属性	默认值	描述
MergeFactory	10	控制segment合并的频率和大小
MaxMergeDocs	Int32.MaxValue	限制每个segment中包含的文档数
MinMergeDocs	10	当内存中的文档达到多少的时候再写入segment

文档(Document)：

文档是我们建索引的基本单位，不同的文档是保存在不同的段中的，一个段可以包含多篇文档。

新添加的文档是单独保存在一个新生成的段中，随着段的合并，不同的文档合并到同一个段中。

域(Field)：

一篇文档包含不同类型的信息，可以分开索引，比如标题，内存，作者等，都可以保存在不同的域里。

不同域的索引方式可以不同。

词(Term)：

词是索引的最小单位，是经过词法分析和语言处理后的字符串。

Lucene的索引结构中，即保存了正向信息，也保存了反向信息。

所谓正向信息：

按层次保存了从索引一直到词的包含关系：索引(Index) -> 段(segment) -> 文档(Document) -> 域(Field) -> 词(Term)

也即此索引包含了那些段，每个段包含了那些文档，每个文档包含了那些域，每个域包含了那些词。

既然是层次结构，则每个层次都保存了本层次的信息以及下一层次的元信息，也即属性信息，比如一本介绍中国地理的书，应该首先介绍中国地理的概况，以及中国包含多少个省，每个省介绍本省的基本概况及包含多少个市，每个市介绍本市的基本概况及包含多少个县，每个县具体介绍每个县的具体情况。

所谓反向信息：

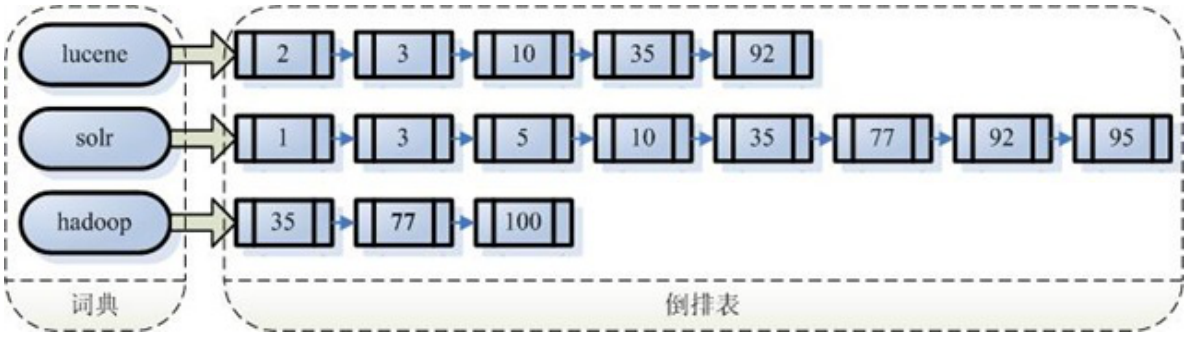
保存了词典到倒排表的映射：词(Term) -> 文档(Document)

词典的构建：

为何Lucene大数据量搜索快, 要分两部分来看：

一点是因为底层的倒排索引存储结构

另一点就是查询关键字的时候速度快 , 因为词典的索引结构



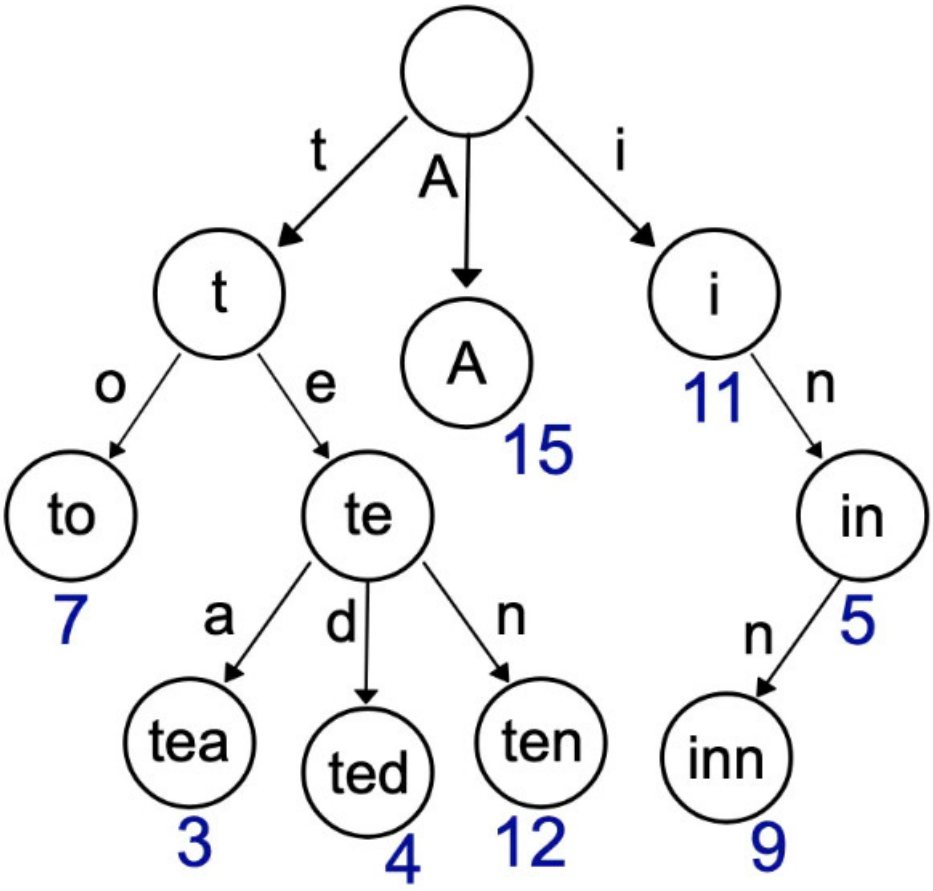
词典数据结构对比：

倒排索引中的词典位于内存，其结构尤为重要，有很多种词典结构，各有各的优缺点，最简单如排序数组，通过二分查找来检索数据，更快的有哈希表，磁盘查找有B树、B+树，但一个能支持TB级数据的倒排索引结构需要在时间和空间上有个平衡，下图列了一些常见词典的优缺点：

很多数据结构均能完成字典功能，总结如下。

数据结构	说明
排序列表Array/List	使用二分法查找，不平衡
HashMap/TreeMap	性能高，内存消耗大，几乎是原始数据的三倍
Skip List	跳跃表，可快速查找词语，在lucene、redis、Hbase等均有实现。相对于TreeMap等结构，特别适合高并发场景
Trie	适合英文词典，如果系统中存在大量字符串且这些字符串基本没有公共前缀，则相应的trie树将非常消耗内存
Double Array Trie	适合做中文词典，内存占用小，很多分词工具均采用此种算法
Ternary Search Tree	三叉树，每一个node有3个节点，兼具省空间和查询快的优点
Finite State Transducers (FST)	一种有限状态转移机，Lucene 4有开源实现，并大量使用

Trie(单词查找树)

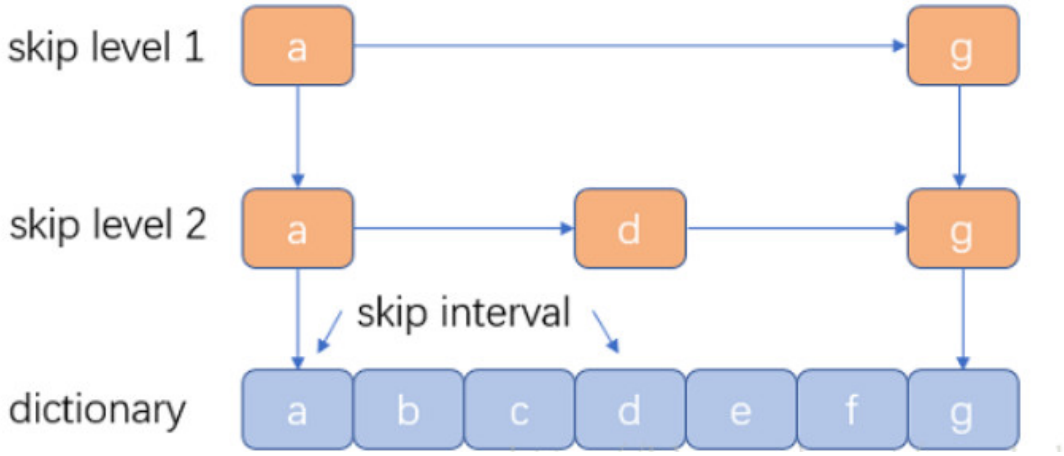


优点：最大限度的减少无谓的比较，查询效率比哈希高

缺点： 空间换时间，利用字符串的公共前缀来降低查询时间的开销已达到提高查询的目的。 空间消耗比较大

跳表结构

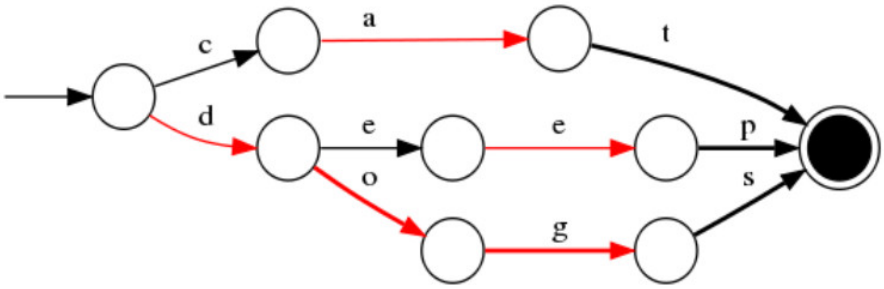
Lucene3.0之前使用的也是跳跃表结构 之后换成了FST，但跳跃表在Lucene其他地方还有应用如倒排表合并和文档号索引。



优点 ：结构简单、跳跃间隔、级数可控

缺点 ：模糊查询支持不好.

FST的结构



优点：内存占用率低，压缩率一般在3倍~20倍之间、模糊查询支持好、查询快

缺点：结构复杂、输入要求有序、更新不易

2.Lucene 相关度排序

相关度排序

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。

Lucene是在用户进行检索时实时根据搜索的关键字计算出相关度得分。

相关度排序公式

Lucene的打分公式非常复杂，如下：

$$score(q,d) = coord(q,d) \times queryNorm(q) \times \sum_{t\ in\ q} (tf(t\ in\ d) \times idf(t)^2 \times t.getBoost() \times norm(t,d))$$

在推导之前，先逐个介绍每部分的意义：

- score(q,d) : 文档d对查询q的相关性得分
- coord(q,d)：协调因子 一次搜索可能包含多个搜索词，而一篇文档中也可能包含多个搜索词，此项表示，当一篇文档中包含的搜索词越多，则此文档则打分越高。
- queryNorm(q)：计算每个查询条目的方差和，使得不同的query之间的分数可以比较。其公式如下：

$$queryNorm(q) = \frac{1}{\sqrt{q.getBoost()^2 \times \sum_{t in q} (idf(t) \times t.getBoost())^2}}$$

- t：Term，这里的Term是指包含域信息的Term，也即title:hello和content:hello是不同的Term
- tf(t in d)：Term t在文档d中出现的词频 一般来说tf越大 得分越高
- idf(t)：Inverse Document Frequency 逆文档频率 Term t在几篇文档中出现过 值大得分低
- norm(t, d)： 标准化因子，是字段长度归一值，与检索时字段的Boost（如果存在）相结合。

  它包括三个参数：
  - Document boost：此值越大，说明此文档越重要。
  - Field boost：此域越大，说明此域越重要。
  - lengthNorm(field) = (1.0 / Math.sqrt(numTerms))：一个域中包含的Term总数越多，也即文档越长，此值越小，文档越短，此值越大。是字段中词数平方根的倒数。

$$norm(t,d) = d.getBoost() \times lengthNorm(field) \times \prod_{field\ f\ in d} f.getBoost()$$

$$lengthNorm(f) = \frac{1}{\sqrt{num\ of\ terms\ in\ field\ f}}$$

- 各类Boost值
  - term boost：查询语句中每个词的权重，可以在查询中设定某个词更加重要。
  - document boost：文档权重，在索引阶段写入文件，表明某些文档比其他文档更重要。
  - field boost：域的权重，在索引阶段写入文件，表明某些域比其他的域更重要。

然而上面各部分为什么要这样计算在一起呢？这么复杂的公式是怎么得出来的呢？下面我们来推导。

首先，将以上各部分代入score(q, d)公式，将得到一个非常复杂的公式，让我们忽略所有的boost，因为这些属于人为的调整，也省略coord，这和公式所要表达的原理无关。得到下面的公式：

$$score(q,d) = \frac{1}{\sqrt{\sum_{t in q} idf(t)^2}} \times \sum_{t in q} tf(t\ in\ d) \times idf(t)^2 \times \frac{1}{\sqrt{num\ of\ terms\ in\ field\ f}}$$

去掉boost信息的lengthNorm部分　　去掉boost信息的norm部分

向量空间模型 VSM（Vector Space Model）

Lucene的打分机制是采用向量空间模型的： 我们把文档看作一系列词(Term)，每一个词(Term)都有一个权重(Term weight)，不同的词(Term)根据自己在文档中的权重来影响文档相关性的打分计算。

$$w_{t,d} = tf_{t,d} \times \log(n / df_t)$$

```
W(t,d):the weight of term t in  document d
tf(t,d):frequency of term t in document d
n      :total number of  documents
df(t):the number of documents that contain term t
```

于是我们把所有此文档中词(term)的权重(term weight) 看作一个向量。

Document = {term1, term2, …… ,term N}

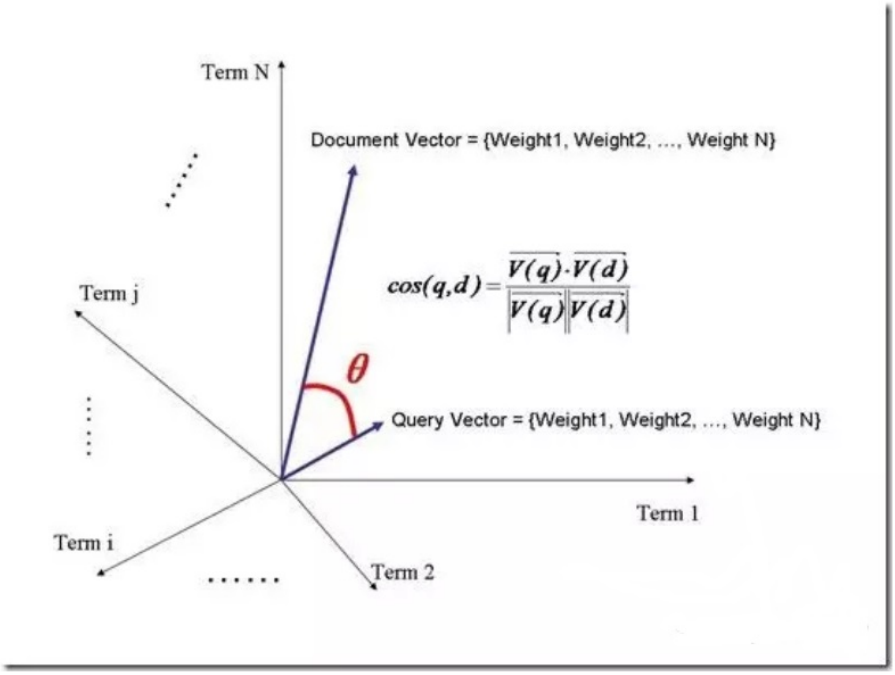
Document Vector = {weight1, weight2, …… ,weight N}

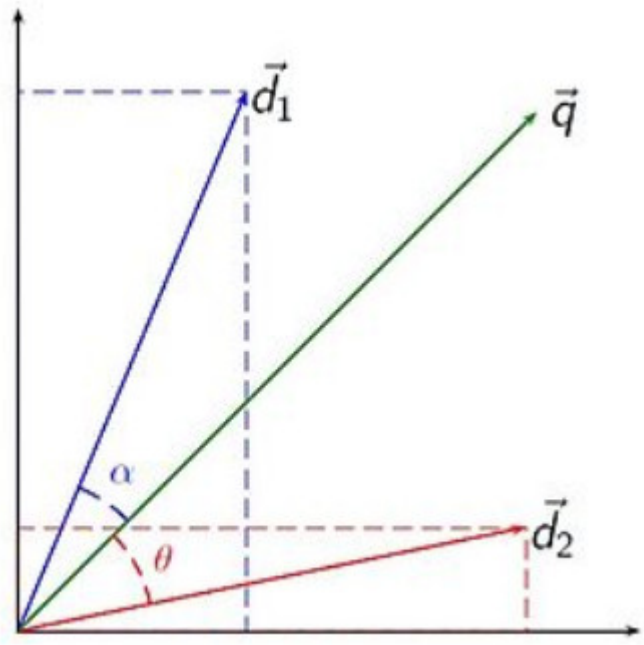
同样我们把查询语句看作一个简单的文档，也用向量来表示。

Query = {term1, term 2, …… , term N}

Query Vector = {weight1, weight2, …… , weight N}

我们把所有搜索出的文档向量及查询向量放到一个N维空间中，每个词(term)是一维。如图：





下面列出计算方法：

设d1=(x1,y1),q=(x2,y2)

我们根据余弦定理，cos(α)=cos(A-B)
=cos(A)cos(B)+sin(A)sin(B)
=(x1/sqr(x1*x1+y1*y1))(x2/sqr(x1*x1+y1*y1))+(y1/sqr(x1*x1+y1*y1)
(y2/sqr(x1*x1+y1*y1))
合并同类项，即向量d1与向量q的内积/向量模。

我们认为两个向量之间的夹角越小，相关性越大。

所以我们计算夹角的余弦值作为相关性的打分，夹角越小，余弦值越大，打分越高，相关性越大。

有人可能会问，查询语句一般是很短的，包含的词(Term)是很少的，因而查询向量的维数很小，而文档很长，包含词(Term)很多，文档向量维数很大。你的图中两者维数怎么都是N呢？

在这里，既然要放到相同的向量空间，自然维数是相同的，不同时，取二者的并集，如果不含某个词(Term)时，则权重(Term Weight)为0。

举个例子，查询语句有11个Term，共有三篇文档搜索出来。其中各自的权重(Term weight)，如下表格。

```
   t1     t2     t3       t4      t5     t6       t7      t8     t9      t10
t11
D1  0      0     0.477     0      0.477   0.176    0      0      0        0.176
   0
D2  0      0.176 0         0.477  0       0        0      0      0.954    0
   0.176  D3  0      0.176 0         0       0       0.176    0      0      0
   0.176    0.176
Q   0      0     0         0      0       0.176    0      0      0.477    0
   0.176
```

相关性打分公式：

$$score(q,d)=\cos(\theta)=\frac{\vec{V_q}\bullet\vec{V_d}}{|\vec{V_q}|\times|\vec{V_d}|}=\frac{1}{\sqrt{\sum_{t in q} idf(t)^2}}\times\sum_{t in q}(tf(t,d)\times idf(t)^2\times\frac{1}{\sqrt{num\,of\ terms\ in\ field\ f}})$$

于是计算，三篇文档同查询语句的相关性打分分别为：

$$SC(Q,D_1)=\frac{(0.176)(0.176)}{\sqrt{0.477^2+0.477^2+0.176^2+0.176^2}\sqrt{0.176^2+0.477^2+0.176^2}}\approx 0.08$$

$$SC(Q,D_2)=\frac{(0.954)(0.477)+(0.176)^2}{\sqrt{0.176^2+0.477^2+0.954^2+0.176^2}\sqrt{0.176^2+0.477^2+0.176^2}}\approx 0.825$$

$$SC(Q,D_3)=\frac{(0.176)^2+(0.176)^2}{\sqrt{0.176^2+0.176^2+0.176^2+0.176^2}\sqrt{0.176^2+0.477^2+0.176^2}}\approx 0.327$$

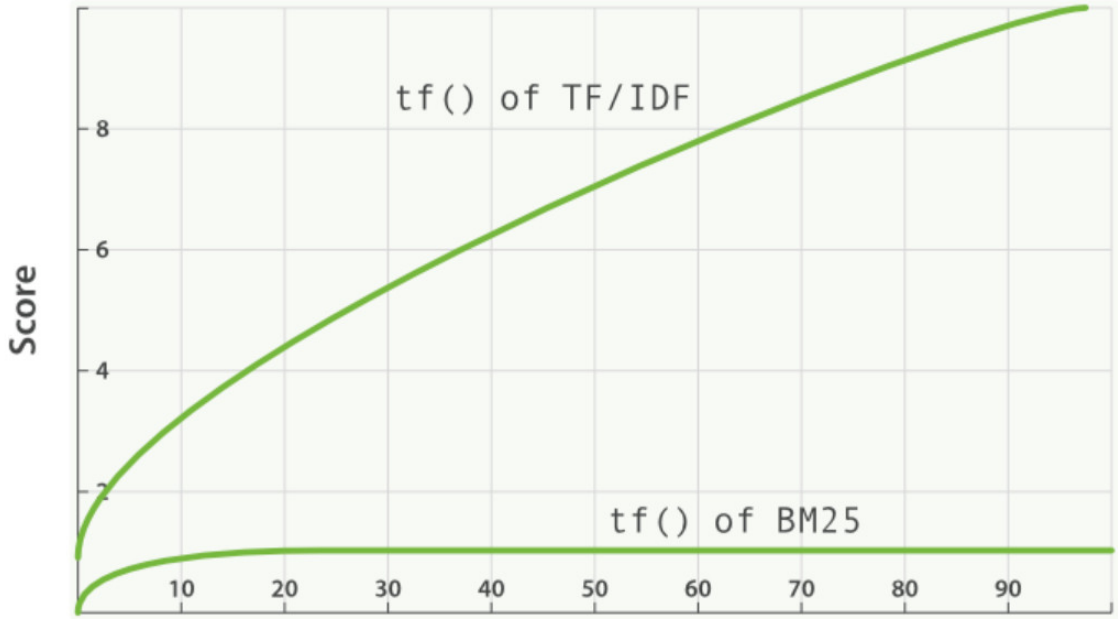
于是文档二相关性最高，先返回，其次是文档三，最后是文档一。

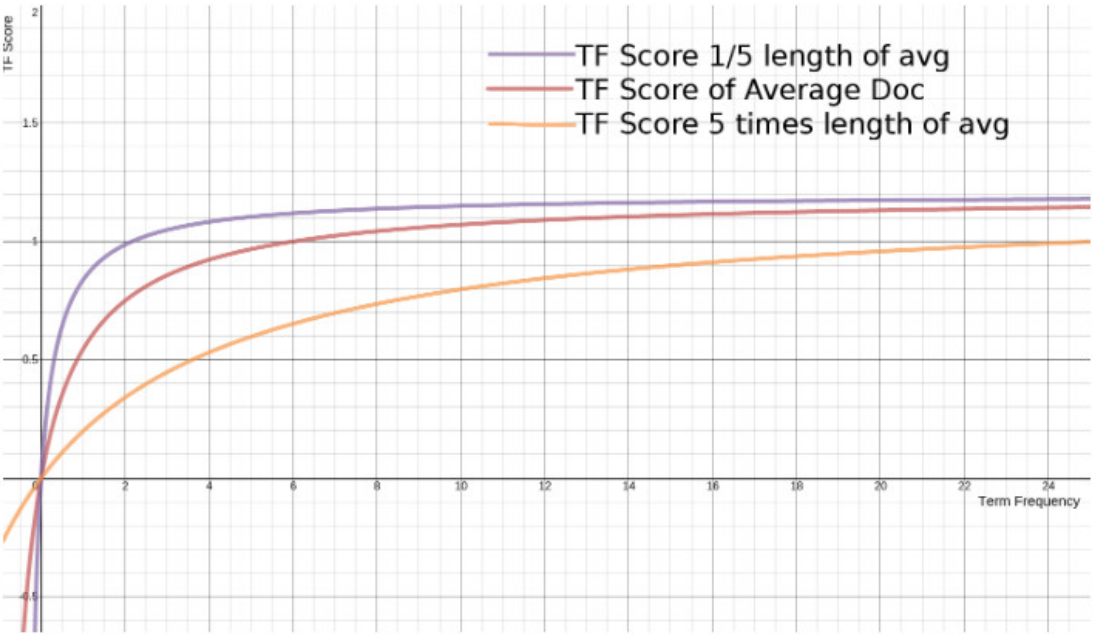
到此为止，我们可以找到我们最想要的文档了

BM25

BM25 源自 概率相关模型（probabilistic relevance model） ，而不是向量空间模型，但这个算法也和 Lucene 的实用评分函数有很多共通之处。加入了词汇在查询向量中的权值以及在文档中的权值还有一系列经验因子。
传统的TF值理论上是可以无限大的。而BM25与之不同，它在TF计算方法中增加了一个常量k，用来限制TF值的增长极限。
BM25还引入了平均文档长度的概念，单个文档长度对相关性的影响力与它和平均长度的比值有关系。

词频饱和度图





从图上可以看到，文档越短，它逼近上限的速度越快，反之则越慢。这是可以理解的，对于只有几个词的内容，比如文章“标题”，只需要匹配很少的几个词，就可以确定相关性。而对于大篇幅的内容，比如一本书的内容，需要匹配很多词才能知道它的重点是讲什么。

BM25 公式

$$\text{score}(D,Q) = \sum_{i=1}^{n} \text{IDF}(q_i) \cdot \frac{f(q_i, D) \cdot (k_1 + 1)}{f(q_i, D) + k_1 \cdot \left(1 - b + b \cdot \frac{|D|}{\text{avgdl}}\right)},$$

$$\text{IDF}(q_i) = \log \frac{N - n(q_i) + 0.5}{n(q_i) + 0.5},$$

> 该公式"."的前部分就是 IDF 的算法，后部分就是 TF+Norm 的算法。
>
> k1：这个参数控制着词频结果在**词频饱和度中的上升速度。**默认值为1.2。值越小饱和度变化越快，值越大饱和度变化越慢。
> b：这个参数**控制着字段长归一值所起的作用，**0.0会禁用归一化，1.0会启用完全归一化。默认值为0.75。

3.Lucene 优化

过滤请求

有些单词不在索引库里，但还需要进索引库查询，发起不必要的IO请求。

使用布隆过滤器，预先判断单词是不是在该索引库里。布隆过滤器原理很简单，对一单词哈希，并映射到相应bit，设置为1，判断时同样做哈希，并去相应bit位取值，若为1，则可能存在，进库查询，若为0，则肯定不存在，不需进库查询。

通过设置IndexWriter的参数优化

现在需要通过 IndexWriterConfig 配置
setMaxBufferedDocs(int maxBufferedDocs)
控制写入一个新的segment前内存中保存的document的数目，设置较大的数目可以加快建索引速度 但会消耗更多的内存，默认为10。
在MergePolicy的实现类中设置
maxMergeDocs
控制一个segment中可以保存的最大document数目，将这个参数设置为比较大的值可以提高检索速度，由于该参数的默认值是整型的最大值，所以我们一般不需要改动这个参数。
mergeFactor
合并因子 是用于控制索引片段的数量,当大于mergeFactor 设定的值时索引将合并成一个大片段。默认是10。
更高的值意味着索引期间更低的段合并开销，但同时也意味着更慢的搜索速度，因为此时的索引通常会包含更多的段。故建议对程序分别进行高低多种值的测试，利用计算机的实际性能来告诉你最优值。

选择合适的分词器

不同的分词器分词效果不同, 所用时间也不同。虽然StandardAnalyzer切分词速度快过IKAnalyzer, 但是由于StandardAnalyzer对中文支持不好, 所以为了追求好的分词效果, 为了追求查询时的准确率, 最好用IKAnalyzer分词器, IKAnalyzer支持停用词典和扩展词典, 可以通过调整两个词典中的内容, 来提升查询匹配的精度。

如果有条件也可以把索引文件放入固态硬盘SSD存放，加速磁盘IO。

屏蔽打分/排序机制

如果业务不需要打分排序机制则可以屏蔽打分排序机制

```
indexSearcher.setSimilarity(indexSearcher.getSimilarity(false));
```

选择合适的对象存放索引库：

类	写操作	读操作	特点
SimpleFSDirectory	java.io.RandomAccessFile	java.io.RandomAccessFile	简单实现，并发能力差
NIOFSDirectory	java.nio.FileChannel	FSDirectory.FSIndexOutput	并发能力强, windows平台下有重大bug
MMapDirectory	内存映射	FSDirectory.FSIndexOutput	读取操作基于内存

建议使用最后一种，第一次查询会慢些，但下次查询是直接从内存中查询，速度加快

Directory directory = MMapDirectory.open(Paths.get("目录路径"));

4.Lucene 使用注意事项

关键词区分大小写：

OR AND TO等关键词是区分大小写的，lucene只认大写的，小写的当做普通单词。

读写互斥性：

同一时刻只能有一个对索引的写操作，在写的同时可以进行搜索

文件锁：

写索引的过程中强行退出将在tmp目录留下一个lock文件，使以后的写操作无法进行，可以将其手工删除

时间格式：

lucene只支持一种时间格式yyMMddHHmmss，所以你传一个yy-MM-dd HH:mm:ss的时间给lucene它 是不会当作时间来处理的。

```
DateTools.dateToString(new Date(), DateTools.Resolution.DAY) 可以把日期处理成yyMMdd
```

设置 boost：

```
有些时候在搜索时某个字段的权重需要大一些，例如你可能认为标题中出现关键词的文章比正文中出现关键词的文章更有价值，你可以把标题的boost设置的更大，那么搜索结果会优先显示标题中出现关键词的文章.
    Query termQuery = new TermQuery(new Term("name","lucene"));
    BoostQuery query =new BoostQuery(termQuery, 3.5f);
```