查询原理（一）

从本篇文章开始介绍Lucene查询阶段的内容，由于Lucene提供了几十种不同方式的查询，但其核心的查询逻辑是一致的，该系列的文章通过Query的其中的一个子类BooleanQuery，同时也是作者在实际业务中最常使用的，来介绍Lucene的查询原理。

查询方式

下文中先介绍几种常用的查询方式的简单应用：

- TermQuery
- BooleanQuery
- WildcardQuery
- PrefixQuery
- FuzzyQuery
- RegexpQuery
- PhraseQuery
- TermRangeQuery
- PointRangeQuery

TermQuery

图1：

```
Query query = new TermQuery(new Term( fld: "content",  text: "a"));
```

图1中的TermQuery描述的是，我们想要找出包含**域名（FieldName）为“content”，域值（FieldValue）中包含“a”的域（Field）**的文档。

该查询方式的demo见：https://github.com/LuXugang/Lucene-7.5.0/blob/master/LuceneDemo/src/main/java/lucene/query/TermQueryTest.java。

BooleanQuery

图2：

```
BooleanQuery.Builder builder = new BooleanQuery.Builder();
builder.add(new TermQuery(new Term( fld: "content",  text: "h")), BooleanClause.Occur.SHOULD);
builder.add(new TermQuery(new Term( fld: "content",  text: "f")), BooleanClause.Occur.SHOULD);
builder.setMinimumNumberShouldMatch(1);

Query query = builder.build();
```

BooleanQuery为组合查询，图2中给出了最简单的多个TermQuery的组合（允许其他查询方式的组合），上图中描述的是，我们期望的文档必须至少（根据BooleanClause.Occur.SHOULD）满足两个TermQuery中的一个，如果都满足，那么打分更高。

该查询方式的demo见：https://github.com/LuXugang/Lucene-7.5.0/blob/master/LuceneDemo/src/main/java/lucene/query/BooleanQueryTest.java。

WildcardQuery

该查询方式为通配符查询，支持两种通配符：

```
    // 星号通配符 *
    public static final char WILDCARD_STRING = '*';
    // 问号通配符 ?
    public static final char WILDCARD_CHAR = '?';
    // 转义符号 (escape character)
    public static final char WILDCARD_ESCAPE = '\\';
```

星号通配符描述的是匹配零个或多个字符，问号通配符描述的是匹配一个字符，转义符号用来对星号跟问号进行转移，表示这两个作为字符使用，而不是通配符。

图3：

```
Document doc;
// 文档0
doc = new Document();
doc.add(new TextField( name: "content",  value: "good job", Field.Store.YES));
doc.add(new TextField( name: "author",  value: "author1", Field.Store.YES));
indexWriter.addDocument(doc);
```

问号通配符的查询：

图4：

```
Query query = new WildcardQuery(new Term( fld: "content",  text: "g?d"));
```

图4中的查询会匹配**文档3，文档1**。

星号通配符的查询：

图5：

```
Query query = new WildcardQuery(new Term( fld: "content",  text: "g*d"));
```

图4中的查询会匹配**文档0、文档1、文档2、文档3**。

转义符号的使用：

图6：

```
Query query = new WildcardQuery(new Term( fld: "content",  text: "g\\*d"));
```

图4中的查询会匹配**文档3**。

该查询方式的demo见：https://github.com/LuXugang/Lucene-7.5.0/blob/master/LuceneDemo/src/main/java/lucene/query/WildcardQueryTest.java。

PrefixQuery

该查询方式为前缀查询：

图7：

```
Query query = new PrefixQuery(new Term( fld: "content",  text: "go"));
```

图7中的PrefixQuery描述的是，我们想要找出包含**域名为“content”，域值的前缀值为"go"的域**的文档。

以图3为例子，图7的查询会匹配**文档0、文档1**。

该查询方式的demo见：https://github.com/LuXugang/Lucene-7.5.0/blob/master/LuceneDemo/src/main/java/lucene/query/PrefixQueryTest.java。

FuzzyQuery

该查询方式为模糊查询，使用编辑距离来实现模糊匹配，**下面的查询都是以图3作为例子**：

图8：

```
Query query = new FuzzyQuery(new Term( fld: "content",  text: "god"),  maxEdits: 2,  prefixLength: 2,  maxExpansions: 20 ,  transpositions: false);
```

图8中的各个参数介绍如下：

- maxEdits：编辑距离的最大编辑值
- prefixLength：模糊匹配到的term的至少跟图8中的域值"god"有两个相同的前缀值，即term的前缀要以"go"开头
- maxExpansions：在maxEidts跟prefixLength条件下，可能匹配到很多个term，但是只允许处理最多20个term
- transpositions：该值在本篇文档中不做介绍，需要了解确定型有穷自动机的知识

图8中的查询会匹配**文档0、文档1**。

图9：

```
Query query = new FuzzyQuery(new Term( fld: "content",  text: "god"),  maxEdits: 2,  prefixLength: 2);
```

图9中的方法最终会调用图8的构造方法，即maxExpansions跟transpositions的值会使用默认值：

- maxExpansions：默认值为50
- transpositions：默认值为true

图9中的查询会匹配**文档0、文档1**。

图10：

```
Query query = new FuzzyQuery(new Term( fld: "content",  text: "god"),  maxEdits: 2);
```

图10中的方法最终会调用图8的构造方法，即prefixLength、maxExpansions跟transpositions的值会使用默认值：

- prefixLength：默认值为0
- maxExpansions：默认值为50
- transpositions：默认值为true

图10中的查询会匹配**文档0、文档1、文档2、文档3**。

图11：

```
Query query = new FuzzyQuery(new Term( fld: "content",  text: "god"));
```

图11中的方法最终会调用图8的构造方法，即maxEdits、maxEprefixLength、maxExpansions跟transpositions的值会使用默认值：

- maxEdits：默认值为2
- prefixLength：默认值为0
- maxExpansions：默认值为50
- transpositions：默认值为true

图10中的查询会匹配**文档0、文档1、文档2、文档3**。

该查询方式的demo见：https://github.com/LuXugang/Lucene-7.5.0/blob/master/LuceneDemo/src/main/java/lucene/query/FuzzyQueryTest.java。

RegexpQuery

该查询方式为正则表达式查询，使用正则表达式来匹配域的域值：

图12：

```
Query query = new RegexpQuery(new Term( fld: "content",  text: "g[o]*d"));
```

图12中的RegexpQuery描述的是，我们想要找出包含**域名（FieldName）为“content”，域值（FieldValue）中包含以"g"开头，以"d"结尾，中间包含零个或多个"o"的域（Field）**的文档。

图12中的查询会匹配**文档0、文档1、文档2**。

该查询方式的demo见：https://github.com/LuXugang/Lucene-7.5.0/blob/master/LuceneDemo/src/main/java/lucene/query/RegexpQueryTest.java。

PhraseQuery

图13：

```
Document doc;
// 文档0
doc = new Document();
doc.add(new TextField( name: "content",  value: "a quick black fox", Field.Store.YES));
doc.add(new TextField( name: "author",  value: "author1", Field.Store.YES));
indexWriter.addDocument(doc);
// 文档1
doc = new Document();
doc.add(new TextField( name: "content",  value: "a really quick red fox", Field.Store.YES));
doc.add(new TextField( name: "author",  value: "author2", Field.Store.YES));
indexWriter.addDocument(doc);
// 文档2
doc = new Document();
doc.add(new TextField( name: "content",  value: "quick fox", Field.Store.YES));
doc.add(new TextField( name: "author",  value: "author3", Field.Store.YES));
indexWriter.addDocument(doc);
// 文档3
doc = new Document();
doc.add(new TextField( name: "content",  value: "fox quick", Field.Store.YES));
doc.add(new TextField( name: "author",  value: "author3", Field.Store.YES));
indexWriter.addDocument(doc);
indexWriter.commit();
// 索引阶段结束
```

该查询方式为短语查询：

图14：

```
PhraseQuery.Builder builder = new PhraseQuery.Builder();
builder.add(new Term( fld: "content",  text: "quick"),  position: 1);
builder.add(new Term( fld: "content",  text: "fox"),  position: 3);
builder.setSlop(1);
Query query = builder.build();
```

图14中，我们定义了两个Term，域值分别为"quick"、"fox"，期望获得的这样文档：文档中必须包含这两个term，这两个term在文档中的位置分别是1、3（如果不满足也没关系，影响的是文档打分值会较低），同时两个term之间的相对位置为2 （3 - 1），并且允许编辑距离最大为1，编辑距离用来**调整**两个term的相对位置（必须满足）。

故根据图13的例子，图14中的查询会匹配**文档0**、**文档1**、**文档2**。

图15：

```
PhraseQuery.Builder builder = new PhraseQuery.Builder();
builder.add(new Term( fld: "content",  text: "quick"),  position: 1);
builder.add(new Term( fld: "content",  text: "fox"),  position: 3);
builder.setSlop(0);
Query query = builder.build();
```

图15中，我们另编辑距离为0，那么改查询只会匹配**文档0**、**文档1**。

图16：

```
PhraseQuery.Builder builder = new PhraseQuery.Builder();
builder.add(new Term( fld: "content",  text: "quick"),  position: 1);
builder.add(new Term( fld: "content",  text: "fox"),  position: 3);
builder.setSlop(4);
Query query = builder.build();
```

图16中，我们另编辑距离为4，此时查询会匹配**文档0**、**文档1**、**文档2**、**文档3**。

这里简单说下短语查询的匹配逻辑：

- 步骤一：找出同时包含"quick"跟"fox"的文档
- 步骤二：计算"quick"跟"fox"之间的相对位置能否在经过编辑距离调整后达到查询的条件

该查询方式的demo见：https://github.com/LuXugang/Lucene-7.5.0/blob/master/LuceneDemo/src/main/java/lucene/query/PhraseQueryTest.java。

TermRangeQuery

图17：

```
Document doc;
// 文档0
doc = new Document();
doc.add(new TextField( name: "content",  value: "a", Field.Store.YES));
doc.add(new TextField( name: "name",  value: "Cris", Field.Store.YES));
indexWriter.addDocument(doc);
// 文档1
doc = new Document();
doc.add(new TextField( name: "content",  value: "bcd", Field.Store.YES));
doc.add(new TextField( name: "name",  value: "Andy", Field.Store.YES));
indexWriter.addDocument(doc);
// 文档2
doc = new Document();
doc.add(new TextField( name: "content",  value: "ga", Field.Store.YES));
doc.add(new TextField( name: "name",  value: "Jack", Field.Store.YES));
indexWriter.addDocument(doc);

// 文档4
doc = new Document();
doc.add(new TextField( name: "content",  value: "gc", Field.Store.YES));
doc.add(new TextField( name: "name",  value: "Pony", Field.Store.YES));
indexWriter.addDocument(doc);
// 文档5
doc = new Document();
doc.add(new TextField( name: "content",  value: "gch", Field.Store.YES));
doc.add(new TextField( name: "name",  value: "Jolin", Field.Store.YES));
indexWriter.addDocument(doc);
// 文档6
doc = new Document();
doc.add(new TextField( name: "content",  value: "gchb", Field.Store.YES));
doc.add(new TextField( name: "name",  value: "Jay", Field.Store.YES));
indexWriter.addDocument(doc);
indexWriter.commit();
// 索引阶段结束
```

该查询方式为范围查询：

图18：

```
Query query = new TermRangeQuery( field: "content", new BytesRef( text: "bc"), new BytesRef( text: "gc"),  includeLower: true,  includeUpper: true);
```

图18中的查询会匹配**文档1、文档2、文档3**。

在后面的文章中会详细介绍TermRangeQuery，对这个查询方法感兴趣的同学可以先看Automaton，它通过确定型有穷自动机的机制来找到查询条件范围内的所有term。

该查询方式的demo见：https://github.com/LuXugang/Lucene-7.5.0/blob/master/LuceneDemo/src/main/java/lucene/query/TermRangeQueryTest.java。

PointRangeQuery

图19：

```
Document doc;
// 文档0
doc = new Document();
doc.add(new IntPoint( name: "coordinate",  ...point: 2, 8));
indexWriter.addDocument(doc);
// 文档1
doc = new Document();
doc.add(new IntPoint( name: "coordinate",  ...point: 4, 6));
indexWriter.addDocument(doc);
// 文档2
doc = new Document();
doc.add(new IntPoint( name: "coordinate",  ...point: 6, 7));
indexWriter.addDocument(doc);
indexWriter.commit();
// 文档3
doc = new Document();
doc.add(new IntPoint( name: "coordinate",  ...point: 4, 3));
indexWriter.addDocument(doc);
indexWriter.commit();
```

该查询方式为域值是数值类型的范围查询（多维度查询）：

图20：

```
IndexReader reader = DirectoryReader.open(indexWriter);
IndexSearcher searcher = new IndexSearcher(reader);
int [] lowValue = {1, 5};
int [] upValue = {4, 7};
Query query = IntPoint.newRangeQuery( field: "coordinate", lowValue, upValue);
```

PointRangeQuery用来实现多维度查询，在图19中，文档0中域名为"coordinate"，域值为"2, 8"的IntPoint域，可以把该域的域值看做是直角坐标系中一个x轴值为2，y轴值为8的一个坐标点。

故文档1中域名为"coordinate"的域，它的域值的个数描述的是维度的维数值。

在图20中，lowValue描述的是x轴的值在[1, 5]的区间，upValue描述的y轴的值在[4, 7]的区间，我们期望找出由lowValue和upValue组成的一个矩形内的点对应的文档。

图21：

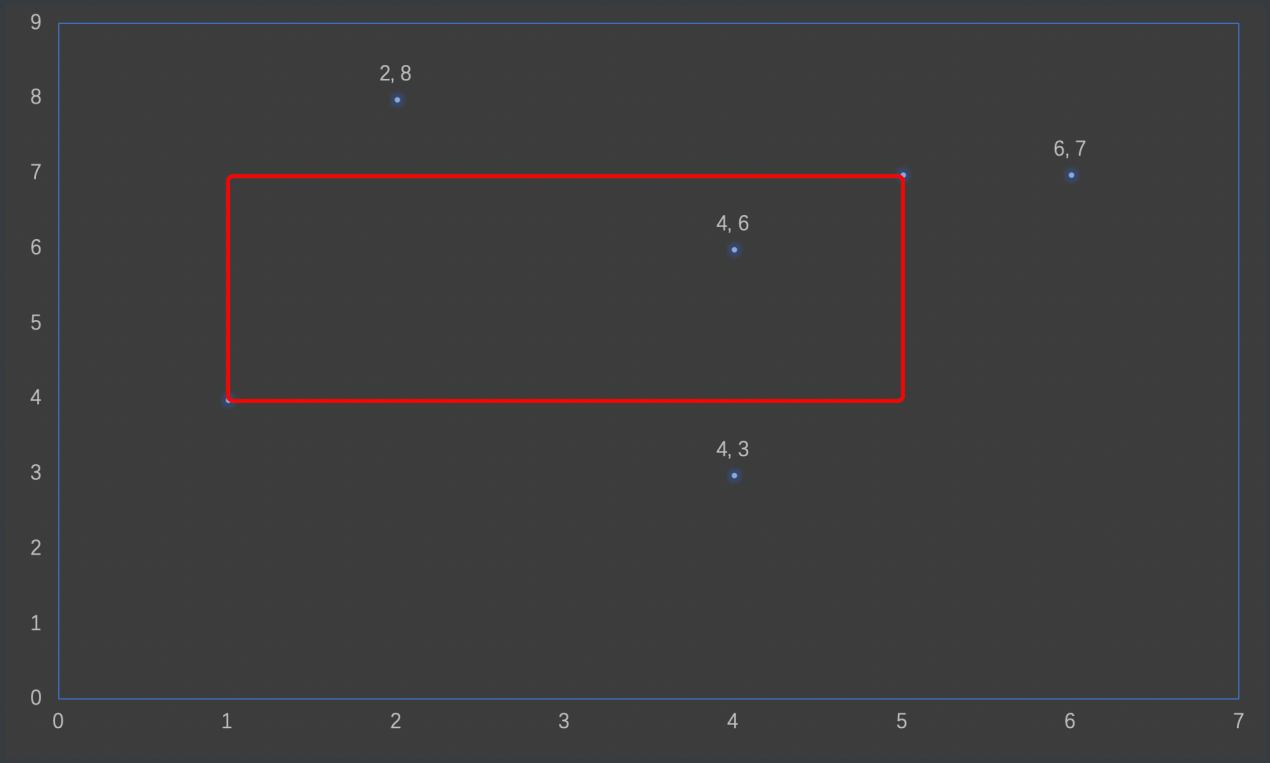

图21中红框描述的是lowValue跟upValue组成的矩形。

故图20中的查询会匹配**文档1**。

在后面的文章中会详细介绍PointRangeQuery的查询过程，对这个查询方法感兴趣的同学可以先看Bkd-Tree以及索引文件之dim&&dii，这两篇文章介绍了在索引阶段如何存储数值类型的索引信息。

该查询方式的demo见：https://github.com/LuXugang/Lucene-7.5.0/blob/master/LuceneDemo/src/main/java/lucene/query/PointRangeQueryTest.java。

结语

无

点击下载附件