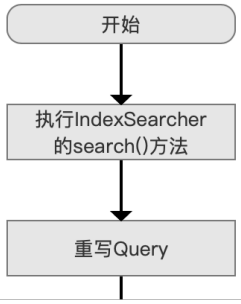
查询原理（二）

在查询原理（一）的文章中，我们介绍了几种常用查询方式的使用方法，从本篇文章开始，通过BooleanQuery来介绍查询原理。

查询原理流程图

图1：



点击查看大图

执行IndexSearcher的search()方法

根据用户提供的不同参数IndexSearcher类提供了多种search( )方法：

- 分页参数：当用户提供了上一次查询的ScoreDoc对象就可以实现分页查询的功能，该内容的实现

方式已经在Collector（二）中介绍，不赘述

- 排序参数：用户通过提供Sort参数使得查询结果按照自定义的规则进行排序，默认使用TopFieldCollector对满足查询条件的结果进行排序。
- Collector参数：用户通过提供Collector收集器来自定义处理那些满足查询条件的文档的方法，如果不提供，那么默认使用TopFieldCollector或者TopScoreDocCollector，如果用户提供了Sort参数，那么使用前者，反之使用后者
- TopN参数：用户通过提供该参数用来描述期望获得的查询结果的最大个数

至于使用上述不同的参数对应哪些不同的search( )就不详细展开了，感兴趣的同学可以结合上述的参数介绍及源码中的几个search( )方法，相信很容易能理解。

重写Query

每一种Query都需要重写（rewrite）才能使得较为友好的接口层面（api level）的Query**完善**成一个"最终的（final）"Query，如果这个Query已经是"最终的"，就不需要重写，这个最终的Query在源码注释中被称为primitive query。

图2中定义了一个TermQuery（接口层面（api level）的Query），它**显示的（explicit）**描述了满足查询条件的文档必须包含一个域名（FieldName）为"content"，域值（FIeldValue）中包含"a"的term（对域值分词后的每一个结果称为term）的域，我们称这种TermQuery是"最终"的Query。在TermQuery中，我们能显示的知道，查询的term为"a"。

图2：

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

图3中定义了一个PrefixQuery（前缀查询，见查询原理（一）），它描述了满足条件的文档必须包含一个域名为"content"，域值中包含前缀为"go"的term的域，相比较TermQuery，这个查询没有**显示的**在用户使用的接口层面（api level）描述我们要查询具体哪个term，我们称之为这不是一个“最终”的Query，故需要通过重写Query来完善成一个新的Query，先找到以"go"为前缀的所有term集合，然后根据这些term重新生成一个Query对象，具体过程在下文中展开。

图3：

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

注意的是上述的介绍只是描述了重写Query的其中一个目的。

根据查询原理（一）中介绍的9种Query，我们分别来讲解这些Query的重写过程。

TermQuery

TermQuery不需要重写。

PointRangeQuery

数值类型的查询，它没有重写的必要。

BooleanQuery

BooleanQuery的重写过程在BooleanQuery的文章中介绍，不赘述。

PhraseQuery

PhraseQuery的重写会生成以下两种新的Query：

- TermQuery：图4中的PhraseQuery，它只有一个域名为“content”，域值为"quick"的term，这种PhraseQuery可以被重写为TermQuery，TermQuery是所有的查询性能最好的查询方式（性能好到Lucene认为这种查询方式都不需要使用缓存机制，见LRUQueryCache），可见这次的重写是一个完善的过程

图4：

```
PhraseQuery.Builder builder = new PhraseQuery.Builder();
builder.add(new Term( fld: "content",  text: "quick"),  position: 0);
builder.setSlop(4);
Query query = builder.build();
```

- PhraseQuery：图5中的PhraseQuery跟图6中的PhraseQuery，他们的查询结果实际是一致的，因为对于图5的PhraseQuery，它会在重写PhraseQuery后变成图6中的PhraseQuery，也就是这种查询方式只关心term之间的相对位置，对于图5中的PhraseQuery，在重写的过程中，"quick"的position参数会被改为0，"fox"的position参数会被改为2，由于本篇文章只是描述PhraseQuery的重写过程，对于为什么要做出这样的重写逻辑，在后面的文章中会展开介绍

图5：

```
PhraseQuery.Builder builder = new PhraseQuery.Builder();
builder.add(new Term( fld: "content",  text: "quick"),  position: 4);
builder.add(new Term( fld: "content",  text: "fox"),  position: 6);
builder.setSlop(4);
Query query = builder.build();
```

图6：

```
PhraseQuery.Builder builder = new PhraseQuery.Builder();
builder.add(new Term( fld: "content",  text: "quick"),  position: 0);
```

FuzzyQuery、WildcardQuery、PrefixQuery、RegexpQuery、TermRangeQuery

这几种Query的重写逻辑是一致的，在重写的过程中，找到所有的term，每一个生成对应的TermQuery，并用BooleanQuery封装。

他们的差异在于不同的Query还会对BooleanQuery进行再次封装，不过这不是我们本篇文章关心的。

下面用一个例子来说明上面的描述：

图7：

```
Document doc;
```

图8：

图8中我们使用TermRangeQuery对图7中的内容进行查询。

图9：

```
BooleanQuery.Builder builder = new BooleanQuery.Builder();
```

图9中我们使用BooleanQuery对图7中的内容进行查询。

图8中TermRangeQuery在重写的过程中，会先找到"bc" ~ "gc"之间的所有term（查找方式见Automaton），这些term即"bcd"、"ga"、"gc"，然后将他们生成对应的TermQuery，并用BooleanQuery进行封装，所以图8中的TermRangeQuery相当于图9中的BooleanQuery。

不得不提的是，TermRangeQuery最终重写后的Query对象不仅仅如此，生成BooleanQuery只是其中最重要，最关键的一步，本篇文章中我们只需要了解到这个程度即可，因为在后面的文章会详细介绍TermRangeQuery。

所有的Query在查询的过程中都会执行该流程点，但不是重写Query唯一执行的地方，在构建Weight的过程中，可能还会执行重写Query的操作。

生成Weight

不同的Query生成Weight的逻辑各不相同，由于无法介绍所有的情况，故挑选了最最常用的一个查询BooleanQuery来作介绍。

图10：

图11：

```
// 查询阶段
IndexReader reader = DirectoryReader.open(indexWriter);
```

图10跟图11分别是索引阶段跟查询阶段的内容，我们在查询阶段定义了一个BooleanQuery，封装了3个TermQuery，该查询条件描述的是：我们期望获得的文档中至少包含三个term，"h"、"f"、"a"中的一个。

BooleanWeight

对于上述的例子中，该BooleanQuery生成的Weight对象如下所示：

图12：

BooleanQuery生成的Weight对象即BooleanWeight对象，它由三个TermWeight对象组合而成，TermWeight即图11中封装的三个**TermQuery**对应生成的Weight对象。

TermWeight

图13中列出了TermWeight中至少包含的几个关键对象：

图13：

Similarity

Similarity描述的是当前查询使用的文档打分规则，当前Lucene7.5.0中默认使用BM25Similarity。用户可以使用自定义的打分规则，可以在构造IndexSearcher后，执行IndexSearcher的search()方法前，调用IndexSearcher.setSimilarity(Similarity)的方法设置。Lucene的文档打分规则在后面的文章中会展开介绍。

SimWeight

图14：

图14中描述的是SimWeight中包含的几个重要的信息，这些信息在后面的流程中用来作为文档打分的参数，由于SimWeight是一个抽象类，在使用BM25Similarity的情况下，SimWeight类的具体实现是BM25Stats类。

我们以下图中红框标识的TermQuery为例子来介绍SimWeight中的各个信息

图15：

field

该值描述的是TermQuery中的域名，在图15中，field的值即“content”。

idf

idf即逆文档频率因子，它的计算公式如下：

```
(float) Math.log(1 + (docCount - docFreq + 0.5D)/(docFreq + 0.5D))
```

- docCount：该值描述是包含域名为“content”的域的文档的数量，从图10中可以看出，文档0~文档9都包含，故docCount的值为10
- docFreq：该值描述的是包含域值"h"的文档的数量，从图10中可以看出，只有文档0、文档8包含，故docFreq的值为2
- 0.5D：平滑值

boost

该值描述的是查询权值（即图17中打分公式的第三部分），boost值越高，通过该查询获得的文档的打分会更高。

默认情况下boost的值为1，如果我们期望查询返回的文档尽量是通过某个查询获得的，那么我们就可以在查询（搜索）阶段指定这个查询的权重，如下图所示：

图16：

相比较图15，在图16中，我们使用BoostQuery封装了TermQuery，并显示的指定这个查询的boost值为100。

图16中的查询条件表达了这么一个意愿：我们更期待在执行搜索后，能获得包含"h"的文档。

avgdl

avgdl（average document length，即图17中打分公式第二部分中参数K中的avgdl变量）描述的是平均每篇文档（一个段中的文档）的长度，并且使用域名为"content"的term的个数来描述平均每篇文档的长度。

例如图7中的文档3，在使用空格分词器（WhitespaceAnalyzer）的情况下，域名为"content"，域值为"a c e"的域，在分词后，文档3中就包含了3个域名为"content"的term，这三个term分别是"a"、"c"、"e"。

avgdl的计算公式如下：

```
(float) (sumTotalTermFreq / (double) docCount)
```

- sumTotalTermFreq：域名为"content"的term的总数，图7中，文档0中有1个、文档1中有1个，文档2中有2个，文档3中有3个，文档4中有1个，文档5中有2个，文档6中有3个，文档7中有1个，文档8中有8个，文档9中有6个，故sumTotalTermFreq的值为（1 + 1 + 2 + 3 + 1 + 2 + 3 + 1 + 8 + 6）= 28
- docCount：同idf中的docCount，不赘述，该值为10

cache

cache是一个数组，数组中的元素会作为BM25Similarity打分公式中的一个参数K（图17打分公式第二部分的参数K），具体cache的含义会在介绍BM25Similarity的文章中展开，在这里我们只需要了解cache这个数组是在生成Weight时生成的。

weight

该值计算公式如下：

```
idf * boost
```

图17是BM25Similarity的打分公式，它由三部分组成，在Lucene的实现中，第一部分即idf，第三部分即boost，**至此我们发现，在生成Weight的阶段，除了文档号跟term在文档中的词频这两个参数，我们已经获得了计算文档分数的其他条件**，至于为什么需要文档号，不是本篇文章关心的部分，再介绍打分公式的文章中会展开介绍，另外 idf跟boost即SimWeight中的信息，不赘述。

图17：

图17源自于<<这就是搜索引擎>>，作者：张俊林。

TermContext

图18：

图18中描述的是TermContext中包含的几个重要的信息，**其中红框标注表示生成Weight阶段需要用到的值**，这些信息通过读取索引文件.tip、tim中的内容获得，其读取过程不再这里赘述（因为太复杂~），不过会在以后的文章中介绍，而每个变量的含义都在.tip、tim中详细介绍了，不赘述。

图19中是索引文件.tim的数据结构：

图19：

点击查看大图

上文中，计算idf（DocCount、DocFreq）跟avgdl（SumTotalTemFreq、DocCount）需要用到的信息在图19中用红框标注。

最后给出完整的BooleanWeight包含的主要信息：

图20：

关于Weight的一些其他介绍

生成Weight的目的是为了不更改Query的属性，使得Query可以复用。

从Weight包含的主要信息可以看出，生成这些信息的目的就是为了文档打分，那如果我们不关心文档的打分，生成Weight的过程又是如何呢?

这个问题包含了两个子问题：

问题一：如何设置不对文档进行打分：

- 我们在执行IndexSearcher的search()方法时，需要提供自定义的Collector，并且通过Collector.needsScores( )来设置为不对文档进行打分

问题二：生成的Weight有什么不同：

- 由于不需要对文档进行打分，所以不需要用到TermContext，即TermContext为null，同时也不需要SimWeight，这两个信息都是为文档打分准备的
- 如果设置了查询缓存（queryCache，默认开启），那么在不对文档打分的前提下，我们还可以使用查询缓存机制，当然使用缓存机制的前提是有要求的，感兴趣的同学可以看LRUQueryCache

结语

基于篇幅，本篇只介绍了图1中的三个流程点`执行IndexSearcher的search()方法`、`重写Query`、`生成Weight`，从本文的内容可以看出，想要深入了解查询逻辑的前提是熟悉索引文件的数据结构。

点击下载附件