第 5 章

更好的搜索

5

上一章介绍了Elasticsearch如何对非扁平数据建立索引，如何索引树状结构和面向对象结构的数据，以及如何修改已创建索引的结构。最后，还学习了如何使用嵌套文档和父子功能来处理文档之间的关系。本章主要内容如下：

- Apache Lucene评分；
- 使用Elasticsearch的脚本功能；
- 对不同语言的数据索引和搜索；
- 使用不同查询来影响返回文档的得分；
- 使用索引时加权；
- 具有相同意思的词；
- 检查为什么特定文档被返回；
- 检查得分计算细节。

5.1 Apache Lucene 评分简介

当谈到查询及其相关性，我们不能忽略得分以及它从哪里来。但得分是什么？得分是描述文档与查询相关度的一个参数。本节将讨论Apache Lucene的默认评分机制：TF/IDF算法，看看它如何影响返回的文档。

TF/IDF算法不是Elasticsearch公开的唯一可用的算法。有关可用模型的更多信息，请参阅2.2.3节，或我们的书*Mastering ElasticSearch*，Packt出版。

5.1.1 当文档被匹配时

Lucene返回文档时，意味着文档与我们发送的查询匹配，并且对该文档已给出一个分数。得分越高，从搜索引擎的角度来看文档越相关。然而，两个不同的查询将对同一文档计算出不同的分数。正因为如此，在查询之间比较分数通常没什么意义。我们回到评分这个话题。计算文档的

评分属性时，考虑以下因素。

- 文档加权：对文档建立索引时，对文档的加权值。
- 字段加权：查询和索引时，对字段的加权值。
- 协调：基于文档词条数的协调因子。对包含更多查询词条的文档，它提供更大的值。
- 逆文档频率：基于词条的因子，它告诉评分公式，给定词条出现的频率有多低。逆文档频率越高，词条越罕见。
- 长度规范：基于字段的规范化因子，它基于给定字段包含的词条数目。字段越长，该因子给的加权值越小。这基本上意味着更短的文档更受分数的青睐。
- 词频：基于词条的因子，描述给定词条在文档中出现的次数，词频越高，文档的得分越高。
- 查询规范：基于查询的规范化因子，由每个查询词条比重的平方之和计算而成。查询规范用于查询之间的得分比较，但这并不一定很容易，有时甚至做不到。

5.1.2 默认评分公式

TF/IDF算法的实用计算公式如下：

$$score(q,d) = coord(q,d) * queryNorm(q) * \sum_{tinq}(tf(tind) * idf(t)^2 * boost(t) * norm(t,d))$$

为了调整查询相关性，你不需要记住这个等式的细节，但至少要知道它是如何工作的。我们可以看到，文档的评分因子是查询q和文档d的一个函数。还有两个不直接依赖于查询词条的因子，coord和queryNorm。公式中这两个元素跟查询中的每个词计算而得的总和相乘。另一方面，该总和由给定词的词频、逆文档频率、词条加权和规范相乘而来，其中的规范就是我们前面讨论过的长度规范。

注意前面的公式是实用性的，你可以在Lucene Javadocs中查看更多概念公式的信息，网址是：http://lucene.apache.org/core/4_7_0/core/org/apache/lucene/search/similarities/TFIDFSimilarity.html。

上述规则的好处是，你不需要记住全部内容。应该知道的是影响文档评分的因素。下面是一些派生自上述等式的规则：

- 匹配的词条越罕见，文档的得分越高；
- 文档的字段越小，文档的得分越高；
- 字段的加权越高，文档的得分越高；
- 我们可以看到，文档匹配的查询词条数目越高、字段越少（意味着索引的词条越少），Lucene给文档的分数越高。同时，罕见词条比常见词条更受评分的青睐。

5.1.3　相关性的意义

在大多数情况下，我们希望得到最匹配的文档，但最相关的不一定是最匹配的。一些用例定义了非常严格的规则，规定了某些文档应该在结果列表中排位靠前。例如，文档除了被TF/IDF相似度模型完美匹配外，有客户付钱让他们的文档出现在结果中更靠前的位置。基于客户计划，我们想给这样的文档更大的重要性，把付费最高的用户的文档放到搜索结果的最顶部。当然，这就不属于TF/IDF相关性了。

这是一个非常简单的例子，但Elasticsearch查询可以非常复杂。5.4节将讨论。

在进行搜索相关性方面的工作时，你应该永远记住，这不是一次性的过程。随着时间的推移，你的数据将改变，查询也需要相应调整。在大多数情况下，优化查询相关性是持续性的工作，要根据业务规则、需求以及用户行为方式等调整。有一点非常重要：记住这不是设置之后就可以抛诸脑后的一次性过程。

5.2　Elasticsearch 的脚本功能

Elasticsearch有几个可以使用脚本的功能。你已经看过一些例子，如更新文件、过滤和搜索。这似乎有点高级，我们还是来看看Elasticsearch提供的可能性，因为在一些用例中，脚本是非常有价值的。Elasticsearch使用脚本执行的任何请求中，我们会注意到以下相似的属性。

- `script`：此属性包含实际的脚本代码。
- `lang`：这个属性定义了提供脚本语言信息的字段。如果省略，Elasticsearch假定为`mvel`。
- `params`：此对象包含参数及其值。每个定义的参数可以通过指定参数名称在脚本中使用。通过使用参数，我们可以编写更干净的代码。由于可以缓存，使用参数的脚本比嵌入常数的代码执行得更快。

5.2.1　脚本执行过程中可用的对象

在不同的操作过程中，Elasticsearch允许在脚本中使用不同的对象。为开发符合我们用例的脚本，应该熟悉这些对象。

比如，在搜索过程中，下列对象是可用的。

- `_doc`（也可以用`doc`）：这是个`org.elasticsearch.search.lookup.DocLookup`对象的实例。通过它可以访问当前找到的文档，附带计算的得分和字段的值。
- `_source`：这是个`org.elasticsearch. search.lookup.SourceLookup`对象的实例，通过它可以访问当前文档的`source`，以及定义在`source`中的值。

- _fields：这是个org.elasticsearch.search.lookup.FieldsLookup对象的实例，通过它可以访问文档的所有字段。

另一方面，在文档更新过程中，Elasticsearch只通过_source属性公开了ctx对象，通过它可以访问当前文档。

我们之前看到过，在文档字段和字段值的上下文中提到了几种方法。现在让我们通过下面的例子，看看如何获取title字段的值。在括号中，你可以看到Elasticsearch从library索引中为我们的一个示例文档返回的值：

- _doc.title.value(crime)
- _source.title(Crime and Punishment)
- _fields.title.value(null)

有点疑惑，不是吗？在索引期间，一个字段值作为_source文档的一部分被发送到Elasticsearch。Elasticsearch可以存储此信息，而且默认的就是存储。此外，文档被解析，每个被标记成stored的字段可能都存储在索引中（也就是说，store属性设置为true；否则，默认情况下，字段不存储）。最后，字段值可以配置成indexed。这意味着分析该字段值，划分为标记，并放置在索引中。综上所述，一个字段可能以如下方式存储在索引中：

- 作为_source文档的一部分；
- 一个存储并未经解析的值；
- 一个解析成若干标记的值。

在脚本中，除了更新操作，我们可以访问所有这些形式。你可能疑惑应该使用哪个版本。好吧，如果我们要访问处理过的形式，答案很简单，_doc。那_source和_fields呢？在大多数情况下，_source是一个不错的选择，比起从索引中读取原始字段值来说，它通常很快并且磁盘操作较少。

5.2.2 MVEL

Elasticsearch可以在脚本中使用几种语言。如果没有明确说明，默认使用MVEL（MVFLEX Expression Language，MVFLEX表达式语言）。MVEL快速、易于使用和嵌入，是在很多开源项目中使用的一个简单但功能强大的表达式语言。它允许我们使用Java对象，自动映射属性到getter/setter方法、简单类型转换、集合映射、数组和关联数组映射等。更多关于MVEL的信息，请参阅http://mvel.codehaus.org/Language+Guide+for+2.0。

5.2.3 使用其他语言

脚本中使用MVEL是一个简单而充分的解决方案，但你也可以选择JavaScript、Python或者

Groovy。使用其他语言之前，必须安装相应的插件，可以在8.7节中阅读更多关于插件的内容。现在，我们只需从Elasticsearch目录中执行以下命令：

```
bin/plugin -install elasticsearch/elasticsearch-lang-
  javascript/2.0.0.RC1
```

上述命令将安装一个插件，使我们能够使用JavaScript。我们在请求中的唯一修改是添加所使用脚本语言的额外信息，当然，还有修改脚本本身，让它在新的语言中是正确的。看看下面的示例：

```
{
  "query" : {
    "match_all" : { }
  },
  "sort" : {
    "_script" : {
      "script" : "doc.tags.values.length > 0 ? doc.tags.values[0]
        :'\u19999';",
      "lang" : "javascript",
      "type" : "string",
      "order" : "asc"
    }
  }
}
```

可以看到，我们使用JavaScript脚本，而不是默认MVEL。`lang`参数通知Elasticsearch我们正在使用的语言。

5.2.4　使用自定义脚本库

脚本通常都很小，可以很方便地放到请求中。但有时随着应用程序的增长，你想给开发者一些可以在他们的模块中重复使用的东西。如果是大型和复杂的脚本，一般最好将它们放在文件中，并仅在API请求中引用。要做的第一件事情是把脚本用适当的名称放在适当的地方。我们的小脚本应该放在Elasticsearch的config/scripts目录中。把我们的示例脚本文件命名为text_sort.js。请注意，文件的扩展名应该指示用于脚本的语言。在本例中，使用JavaScript。本示例文件的内容很简单，如下所示：

```
doc.tags.values.length > 0 ? doc.tags.values[0] :'\u19999';
```

一个使用上述脚本的查询如下所示：

```
{
  "query" : {
    "match_all" : { }
  },
  "sort" : {
    "_script" : {
```

```
        "script" : "text_sort",
        "type" : "string",
        "order" : "asc"
      }
    }
}
```

你可以看到，现在可以使用text_sort作为脚本的名称。此外，可以省略脚本语言，Elasticsearch会从文件扩展名判断它。

使用本地代码

如果脚本太慢，或者你不喜欢脚本语言，Elasticsearch允许你使用Java类，而不是脚本。

(1) 工厂实现类

需要实现至少两个类来创建新的本地脚本。第一个是我们脚本的工厂。现在，先关注它。下面的代码示例说明了我们的脚本工厂：

```
package pl.solr.elasticsearch.examples.scripts;

import java.util.Map;
import org.elasticsearch.common.Nullable;
import org.elasticsearch.script.ExecutableScript;
import org.elasticsearch.script.NativeScriptFactory;

public class HashCodeSortNativeScriptFactory implements
  NativeScriptFactory {

  @Override
  public ExecutableScript newScript(@Nullable Map <string, Object>
    params) {
    return new HashCodeSortScript(params);
  }
}
```

重要部分是高亮的代码片段。这个类必须实现org.elasticsearch.script.NativeScriptFactory类。该接口强制我们实现newScript()方法。它接收定义在API请求中的参数，返回脚本的一个实例。

(2) 实现本地脚本

现在让我们看看脚本的实现。想法很简单：我们的脚本将用于排序。文档将按照选择字段的hashCode()值来排序，没有定义该字段的文档将是第一个。我们知道此逻辑没有太多意义，但它很简单，是个好例子。本地脚本源代码如下所示：

```
package pl.solr.elasticsearch.examples.scripts;

import java.util.Map;
```

```
import org.elasticsearch.script.AbstractSearchScript;

public class HashCodeSortScript    extends AbstractSearchScript {
  private String field = "name";

  public HashCodeSortScript(Map <string, Object> params) {
    if (params != null && params.containsKey("field")) {
      this.field = params.get("field").toString();
    }
  }

  @Override
  public Object run() {
    Object value = source().get(field);
    if (value != null) {
      return value.hashCode();
    }
    return 0;
  }
}
```

首先，我们的类从`org.elasticsearch.script.AbstractSearchScript`类继承并实现`run()`方法。该方法从当前文档中得到适当的值，并且根据我们的奇怪逻辑来处理后，返回一个结果。你可能注意到了`source()`，没错，它和我们在非本地脚本中碰到的`_source`参数完全一样。`doc()`和`fields()`方法同样是可用的，与之前描述的逻辑一样。

值得一看的是我们使用这些参数的方式。假设用户会填充`field`参数，告诉我们文档的哪个字段将被用来操纵。我们还提供了此参数的默认值。

(3) 安装脚本

现在来安装本地脚本。在把已编译的类封装成JAR归档后，应该把它放在Elasticsearch的lib目录，这使我们的代码对类加载器可见。我们应该做的是注册脚本，可以通过Setting API调用来实现或在elasticsearch.yml配置文件添加一行。这里选择使用elasticsearch.yml脚本，将下面这行添加到上述配置文件：

```
script.native.native_sort.type:
  pl.solr.elasticsearch.examples.scripts.
    HashCodeSortNativeScriptFactory
```

注意`native_sort`片段。这是在请求期间使用的脚本名称，它将会被传递给`script`参数。此属性的值是工厂实现的完整类名，它将用来初始化脚本。最后，需要重新启动Elasticsearch。

(4) 执行脚本

已经重新启动Elasticsearch，所以，可以开始使用本地脚本发送查询。例如，发送一个查询，使用之前的`library`索引中的数据。示例查询如下所示：

```
{
  "query" : {
    "match_all" : { }
  },
  "sort" : {
    "_script" : {
      "script" : "native_sort",
      "params" : {
        "field" : "otitle"
      },
      "lang" : "native",
      "type" : "string",
      "order" : "asc"
    }
  }
}
```

注意查询的params部分，在这个请求中，要对otitle字段排序。提供native_sort作为脚本名字、native作为脚本语言，这是必需的。如果一切顺利，应该看到结果按自定义排序逻辑排序。在Elasticsearch的响应中，我们会看到，没有otitle字段的文档将出现在结果列表的最上面，它们的sort值为0。

5.3 搜索不同语言的内容

此前，在讨论语言的分析时，我们主要谈理论，还没看到一个关于语言分析、处理数据可能包含的多语言例子。现在，我们将讨论如何处理多语言的数据。

5.3.1 区分处理不同语言

你已经知道，Elasticsearch能为数据提供不同的分析器，可以让数据以空白符分割、小写，等等。这通常可以处理不同语言的数据：应该可以把基于空白符的分词器用在英语、德语和波兰语（但不适用于中文）。然而，如果你只想发送单词cat到Elasticsearch，找包含cat和cats的文档，应该怎么办？这是语言分析起作用的地方，使用不同语言的词干提取算法，分析单词，回退到词根状态。

现在最糟糕的部分是，不能用一个通用的词干提取算法来处理世界上所有的语言；要选择一个合适的语言。以下章节将帮助你理解语言分析过程的某些部分。

5.3.2 多语言处理

Elasticsearch中有几种处理多语言的方法，它们都有利有弊。我们不会讨论每一个，为了让你有所了解，列出如下方法：

- 把不同语言的文档存储成不同的类型；
- 把不同语言的文档存储到单独的索引中；
- 对单一文档的字段存储多个版本，每个版本包含不同的语言。

不过，我们会将重心放在一个能够将多语言文档存储在单个索引的方法。将注意力集中在这个问题上：文档只有一个类型，但它们可能来自世界各地，因此可以有多种语言。同时，我们希望用户在不同语言上使用所有的分析功能，如词干提取和停止词，而不仅是英语。

请注意，不同语言上的词干提取算法是不一样的：不管是分析性能还是词条结果。例如，英语词干分析器很好，但在欧洲语言（比如德语）上执行时，可能会有问题。

5.3.3　检测文档的语言

如果你不知道文档或查询的语言（大多时候是这样），可以使用语言检测软件，在一定程度上可以检测文档或查询的语言。如果使用Java，可以使用几个可用的语言检测库之一。比如下面这些：

- Apache Tika（http://tika.apache.org/）；
- Language detection（http://code.google.com/p/language-detection/）。

Language detection库声称支持53种语言并提供99%的准确度，可以说很多了。

应该记住，文本越长语言检测越准确。然而，由于查询的文本通常很短，你可能会在查询语言识别过程中遇到一定程度的错误。

5.3.4　示例文档

先介绍一个示例文档，如下所示：

```
{
  "title" : "First test document",
  "content" : "This is a test document",
  "lang" : "english"
}
```

可以看到，该文档很简单，包含如下3个字段。

- `title`：该字段存储文档的标题。
- `content`：该字段存储文档的实际内容。
- `lang`：该字段定义识别语言。

前两个字段从用户的文档创建而来，第三个字段是我们的假想用户在上传文档时选择的。

为了通知Elasticsearch选择哪个分析器，我们把lang字段映射到Elasticsearch的分析器之一（分析器的完整列表可以在这里找到：http://www.elasticsearch.org/guide/en/elasticsearch/reference/current/analysis-lang-analyzer.html）。如果用户输入了一个不支持的语言，就不设置lang字段，让Elasticsearch使用默认分析器。

5.3.5 映射文件

来看看为上述文档创建的映射，把它们存储在mappings.json文件中，如下所示：

```
{
  "mappings" : {
    "doc" : {
      "_analyzer" : {
        "path" : "lang"
      },
      "properties" : {
        "title" : {
          "type" : "string",
          "index" : "analyzed",
          "store" : "no",
          "fields" : {
            "default" : {
              "type" : "string",
              "index" : "analyzed",
              "store" : "no",
              "analyzer" : "simple"
            }
          }
        },
        "content" : {
          "type" : "string",
          "index" : "analyzed",
          "store" : "no",
          "fields" : {
            "default" : {
              "type" : "string",
              "index" : "analyzed",
              "store" : "no",
              "analyzer" : "simple"
            }
          }
        },
        "lang" : {
          "type" : "string",
          "index" : "not_analyzed",
          "store" : "yes"
        }
      }
```

```
      }
    }
  }
```

上面的映射中，我们最感兴趣的是分析器定义，以及title和description字段（如果你还不熟悉映射，请参阅2.2节）。我们希望分析器基于lang字段，因此，要在lang字段添加一个与Elasticsearch知道的分析器的名字相同的值，可以是默认的，也可以是自定义的。

其次是拥有实际数据的两个字段的定义。你可以看到，我们使用多字段定义来索引title和description字段。多字段的第一个使用lang字段指定的分析器（因为没有指定分析器名字，所以使用全局定义的那个）来建立索引，如果查询时知道指定的语言，也将使用这个字段。多字段的第二项使用simple分析器，不知道查询语言时，使用这个字段。simple分析器只是个例子，你同样可以使用标准分析器或其他任意分析器。

为使用这个映射文件创建一个叫docs的简单索引，执行如下命令：

```
curl -XPUT 'localhost:9200/docs' -d @mappings.json
```

5.3.6　查询

现在看看如何查询数据。把查询分成下面两种情况。

1. 用识别语言查询

第一种情况是确定了查询语言。假设识别的语言是英语，我们知道英语与english分析器匹配。在这种情况下，查询如下所示：

```
curl -XGET 'localhost:9200/docs/_search?pretty=true ' -d '{
  "query" : {
    "match" : {
      "content" : {
        "query" : "documents",
        "analyzer" : "english"
      }
    }
  }
}'
```

analyzer参数指明了我们想用的分析器。将该参数设置为识别语言所对应的分析器的名字。注意，我们正在寻找的词条是documents，而在文档中的词条是document。english分析器可以处理这个情况并找到文档。Elasticsearch返回的响应如下所示：

```
{
  "took" : 2,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
```

```
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 1,
    "max_score" : 0.19178301,
    "hits" : [ {
      "_index" : "docs",
      "_type" : "doc",
      "_id" : "1",
      "_score" : 0.19178301
    } ]
  }
}
```

2. 用未知语言查询

假设不知道用户查询使用的语言。此时，不能使用由lang字段指定的分析器，因为我们不想用一个特定于语言的分析器来分析查询。在这种情况下，用标准的简单分析器，发送查询到contents.default字段，而不是content字段。查询如下所示：

```
curl -XGET 'localhost:9200/docs/_search?pretty=true ' -d '{
  "query" : {
    "match" : {
      "content.default" : {
        "query" : "documents",
        "analyzer" : "simple"
      }
    }
  }
}'
```

然而，这次没有得到任何结果，因为搜索单词的复数形式时，simple分析器不能处理它的单数形式。

3. 组合查询

为了对完美匹配默认分析器的文档额外加权，可以把上述两个查询用bool查询组合起来，如下所示：

```
curl -XGET 'localhost:9200/docs/_search?pretty=true ' -d '{
  "query" : {
    "bool" : {
      "minimum_should_match" : 1,
      "should" : [
        {
          "match" : {
            "content" : {
              "query" : "documents",
              "analyzer" : "english"
            }
```

```
            }
          },
          {
            "match" : {
              "content.default" : {
                "query" : "documents",
                "analyzer" : "simple"
              }
            }
          }
        ]
      }
    }
}'
```

返回的文档至少必须匹配一个定义的查询。如果两者都匹配，文档将具有更高的分数值，并在结果中放置得更高。

前面的组合查询有一个额外的优势：如果我们的语言分析器找不到一个文档（例如，所用的分析跟在索引期间使用的不一样时），第二个查询有机会找到只用空白字符分词和小写形式的词条。

5.4　使用查询加权影响得分

上一章介绍了什么是得分以及Elasticsearch如何计算它。随着应用程序的增长，提高搜索质量的需求也进一步增大。我们把它叫做搜索体验。我们需要知道什么对用户更重要，关注用户如何使用搜索功能。这导致不同的结论，例如，有些文档比其他的更重要，或特定查询需强调一个字段而弱化其他字段。这就是可以用到加权的地方。

5.4.1　加权

*加权*是一个评分过程中额外使用的值。我们已经知道它适用于下列地方。

- `query`：这可以通知搜索引擎，给定查询是复杂查询的一部分，而且比其他部分更重要。
- `field`：有几个文档字段对用户非常重要。例如，以Bill搜索电子邮件,应该首先列出那些发送自Bill的邮件，紧随其后列出主题中含有Bill的邮件，然后是内容中提到Bill的邮件。

指定给查询或字段的加权值只是计算分数时的众多因素之一，我们都应意识到这一点。现在，看几个查询的例子。

5.4.2　为查询添加加权

假设索引有两个文档，第一个文档如下所示：

```
{
  "id" : 1,
  "to" : "John Smith",
  "from" : "David Jones",
  "subject" : "Top secret!"
}
```

第二个文档如下所示：

```
{
  "id" : 2,
  "to" : "David Jones",
  "from" : "John Smith",
  "subject" : "John, read this document"
}
```

数据很简单，但它应该很好地描述了我们的问题。现在，假设有以下查询：

```
{
  "query" : {
    "query_string" : {
       "query" : "john",
       "use_dis_max" : false
    }
  }
}
```

在这个例子中，Elasticsearch将为_all字段创建一个查询，并将查找包含所需文字的文档。通过把use_dis_max参数设为false，告诉Elasticsearch不希望使用disjunction查询（如果你不记得disjunction查询，请参阅3.3节中的最大分查询和字符串查询部分）。很容易猜到，我们的两条记录都将返回，标识符等于2的那个将第一个返回。这是由于John分别出现在from字段和subject字段。检查一下结果：

```
"hits" : {
  "total": 2,
  "max_score": 0.13561106,
  "hits": [{
    "_index": "messages",
    "_type": "email",
    "_id": "2",
    "_score": 0.13561106,"_source":
    {"id": 1,"to": "David Jones","from":
    "John Smith","subject": "John, read this document"}
  }, {
    "_index": "messages",
    "_type": "email",
    "_id": "1",
    "_score": 0.11506981,"_source":
    {"id": 2,"to": "John Smith","from":
    "David Jones","subject": "Top secret!"}
  } ]
}
```

一切都正常吗？技术上来说，是的。但我认为第二个文档应该出现在结果列表的第一位，因为搜索时，在许多情况下，最重要的因素是匹配人，而不是消息主题。你可能不同意，但这正好解释了为什么全文搜索的相关性是一个困难的课题：有时，很难判断在特定情况下哪个排序更好。我们能做什么？首先，重写查询，间接告知Elasticsearch应使用哪些字段搜索，如下所示：

```
{
  "query" : {
    "query_string" : {
      "fields" : ["from", "to", "subject"],
      "query" : "john",
      "use_dis_max" : false
    }
  }
}
```

这和上一个示例查询不完全一样。运行它，将得到同样的结果，但如果仔细观察，你会发现在得分上的差异。在上一个示例中，Elasticsearch只使用一个字段，`_all`。现在我们在3个字段上搜索。这意味着有些因素改变了，如字段长度等。不过，这在我们的例子中不是那么重要。在底层，Elasticsearch生成由3个查询组成的复杂查询：每个字段一个查询。当然，每个查询的贡献得分取决于这个字段上找到的词条数以及这个字段的长度。我们介绍一些字段之间的差异。将下面的查询同前面的查询相比较：

```
{
  "query" : {
    "query_string" : {
      "fields" : ["from^5", "to^10", "subject"],
      "query" : "john",
      "use_dis_max" : false
    }
  }
}
```

看看高亮显示的部分（^5和^10）。通过这种方式，可以告诉Elasticsearch给定字段的重要程度。我们看最重要的字段是`to`，其次是`from`。`subject`字段的`boost`为默认值，即`1.0`。永远记住，这个值只是各种因素之一。你可能想知道为什么选择5，而不是1000或1.23。嗯，这取决于我们想要达到的效果，有什么样的查询，更重要的是在索引中有什么样的数据。通常，当数据有意义的部分发生变化时，也许我们应该再次检查和调整相关性。最后，看一个类似的例子，但这次使用`bool`查询，如下所示：

```
{
  "query" : {
    "bool" : {
      "should" : [
        { "term" : { "from": { "value" : "john", "boost" : 5 }}},
        { "term" : { "to": { "value" : "john", "boost" : 10  }}},
        { "term" : { "subject": { "value" : "john" }}}
      ]
```

```
      }
    }
}
```

5.4.3 修改得分

前面的示例演示了如何通过加权特定查询组件来影响结果列表。另一种方法是运行一个查询来影响匹配文档的得分。接下来几节将总结Elasticsearch提供的几种可能性。我们将在例子中使用第3章中用过的数据。

1. constant_score查询

constant_score查询允许我们对任何过滤器或查询明确设置一个被用作得分的值，它将通过加权参数赋给每个匹配文档。

乍看起来，此查询并不实际。但考虑到建立复杂查询的情况，这种查询允许我们设置多少个匹配查询的文档可以影响总分。看看下面的示例：

```
{
  "query" : {
    "constant_score" : {
      "query": {
        "query_string" : {
          "query" : "available:false author:heller"
        }
      }
    }
  }
}
```

在我们的数据中，有两个文档的available字段为false，其中一个在author字段上有值。但由于constant_score查询，Elasticsearch将在评分时忽略此信息，两个文档的得分都是1.0。

2. 加权查询

下一个与加权相关的查询是加权查询。它允许我们定义一个查询的额外部分，用于降低匹配文档的得分。下面的例子列出所有图书，但E.M.Remarque写的书得分将低10倍：

```
{
  "query" : {
    "boosting" : {
      "positive" : {
        "term" : {
          "available" : true
        }
      },
      "negative" : {
        "match" : {
          "author" : "remarque"
```

```
          }
        },
        "negative_boost" : 0.1
      }
    }
}
```

3. function_score查询

我们已经看过两个允许改变查询返回文档得分的例子。第三个例子，我们想谈谈function_score查询，它比之前的查询更复杂。这个查询在得分计算成本高昂时非常有用，因为它将计算过滤后文档的得分。

(1) *函数查询的结构*

函数查询的结果很简单，看上去如下所示：

```
{
  "query" : {
    "function_score" : {
      "query" : { ... },
      "filter" : { ... },
      "functions" : [
       {
         "filter" : { ... },
         "FUNCTION" : { ... }
       }
     ],
      "boost_mode" : " ... ",
      "score_mode" : " ... ",
      "max_boost" : " ... ",
      "boost" : " ... "
    }
  }
}
```

一般来说，function_score查询中可以使用查询、过滤、函数和附加参数。每个函数可以有一个过滤器定义要应用的过滤结果。如果一个函数没有定义过滤器，它将应用到所有文档。

function_score查询背后的逻辑很简单。首先，函数匹配文档并基于score_mode参数计算得分。然后，文档的查询得分由函数计算所得分数结合而成，结合时基于boost_mode参数。

我们现在来讨论以下参数。

- boost_mode：boost_mode参数允许定义如何将函数查询所计算分数与查询分数结合起来。可以将它设置成下列值。
  - multiply：这是默认行为，查询得分将与函数计算所得分相乘。
  - replace：导致查询得分全部被忽略，文档得分等于函数计算所得分。

- sum：文档得分等于查询得分和函数得分的总和。
- avg：文档得分等于查询得分和函数得分的平均值。
- max：文档得分等于查询得分和函数得分的最大值。
- min：文档得分等于查询得分和函数得分的最小值。

❑ score_mode：该参数定义了函数计算所得分是如何结合在一起的。以下是该参数可以设置的值。

- multiply：这是默认行为，结果是查询得分乘以函数的得分。
- sum：把所定义的函数的得分相加。
- avg：函数得分等于所有匹配函数得分的平均值。
- first：把第一个拥有匹配文档过滤器的函数的得分返回。
- max：返回所有函数得分的最大值。
- min：返回所有函数得分的最小值。

要记住，可以通过使用function_score查询中的max_boost参数来限制最大计算得分。默认情况下，这个参数的值被设为Float.MAX_VALUE，意思是最大浮点值。

boost参数允许我们为文档设置一个查询范围的加权值。

我们还没谈到可以包含到查询的functions节点中的函数，下面是目前可用的函数。

❑ boost_factor函数：这个函数允许我们把文档分数乘以一个给定值。boost_factor参数的值不会被范式化，而是原原本本的值。下面是使用boost_factor参数的一个例子：

```
{
  "query" : {
    "function_score" : {
      "query" : {
        "term" : {
          "available" : true
        }
      },
      "functions" : [
        { "boost_factor" : 20 }
      ]
    }
  }
}
```

❑ script_score函数：这个函数允许我们使用一个脚本来计算得分，用于函数返回的得分（然后将落到boost_mode参数定义的行为中去）。使用script_score函数的例子如下所示：

```
{
  "query" : {
```

```
      "function_score" : {
        "query" : {
          "term" : {
            "available" : true
          }
        },
        "functions" : [
          {
            "script_score" : {
              "script" : "_score * _source.copies *
                parameter1",
              "params" : {
                "parameter1" : 12
              }
            }
          }
        ]
      }
    }
}
```

- random_score函数：使用这个函数，可以通过指定seed值来生成一个伪随机分数。为了模拟随机性，每次都应指定一个新的seed。一个使用此功能的例子如下所示：

```
{
  "query" : {
    "function_score" : {
     "query" : {
       "term" : {
         "available" : true
       }
      },
      "functions" : [
        {
          "random_score" : {
            "seed" : 12345
          }
        }
      ]
    }
  }
}
```

- decay函数：除了前面提到的评分函数以外，Elasticsearch包含了额外的decay函数。前述函数给出的分数随着距离变大而变低，该函数则不同。距离基于一个单值的数值型字段（比如日期、地理位置点或标准的数值型字段）计算而来。最容易想到的例子是基于与一个给定点的距离来加权文档。

假设有一个point字段，存储着位置信息，我们希望文档的得分受与用户所在位置的距离影响（比如，用户用手机设备发送请求），用户在52，21这个位置，我们可以发送如下查询：

```
{
  "query" : {
    "function_score" : {
      "query" : {
        "term" : {
          "available" : true
        }
      },
      "functions" : [
        {
          "linear" : {
            "point" : {
              "origin" : "52, 21",
              "scale" : "1km",
              "offset" : 0,
              "decay" : 0.2
            }
          }
        }
      ]
    }
  }
}
```

在前面的示例中，linear是decay函数的名称。使用它时，值将线性衰退。其他可能的值是gauss和exp。我们选择了linear衰减函数，因为当字段值两次超过给定值时，它把分数设置为0。当你想把较低的值赋予太远的文档时，这是非常有用的。

在此给出相关方程，让你了解得分是如何通过给定函数计算的。linear衰减函数使用以下公式来计算得分文档：

$$score = \max\left(0, \frac{scale - |\ field\ value - origin\ |}{scale}\right)$$

gauss衰减函数使用以下公式来计算得分文档：

$$score = \exp\left(-\frac{(field\ value - origin)^2}{2scale^2}\right)$$

exp衰减函数使用以下公式来计算得分文档：

$$score = \exp\left(-\frac{|\ field\ value - origin\ |}{scale}\right)$$

当然，你不需要每次都使用纸笔计算文档，但偶尔需要时，这些函数很方便。

现在，让我们讨论一下查询结构的其余部分。我们想用point字段作得分计算。如果文档没

有该字段的值，在计算时会得到一个值为1。

此外，我们提供了额外的参数。origin和scale参数是必需的。origin是中心点，计算从此点开始。scale是衰变率。默认情况下，offset参数设置为0；如果定义了该参数，decay函数将只计算文档值比此参数的值大的文档得分。decay参数告诉Elasticsearch应该降低多少分数，默认设置为0.5。在我们的例子中，我们要求，在1公里的距离，得分应该会减少20%（0.2）。

我们期望新版的Elasticsearch会扩展可用函数的数量，建议跟进官方文档，function_score查询的专属页面可访问这里：http://www.elasticsearch.org/guide/en/elasticsearch/reference/current/query-dsl- function-score-query.html。

4. 弃用查询

在介绍function_score查询之后，custom_boost、custom_score和custom_filters_score查询被弃用。以下部分演示如何使用function_score查询实现与这些查询相同的结果。本节为想从Elasticsearch旧版本迁移并通过修改来移除废弃查询的人提供一个参考。

(1) 更换custom_boost_factor查询

假设我们有如下的custom_boost_factor查询：

```
{
  "query" : {
    "custom_boost_factor" : {
      "query": {
        "term" : { "author" : "heller" }
      },
      "boost_factor": 5.0
    }
  }
}
```

为使用function_score查询来取代上述查询，可以使用下列查询：

```
{
  "query" : {
    "function_score" : {
      "query": {
        "term" : { "author" : "heller" }
      },
      "functions" : [
        { "boost_factor": 5.0 }
      ]
    }
  }
}
```

(2) 更换custom_score查询

第二种废弃查询是custom_score，假设有如下的custom_score查询：

```
{
  "query" : {
    "custom_score" : {
      "query" : { "match_all" : {} },
      "script" : "_source.copies * 0.5"
    }
  }
}
```

如果想用function_score查询取代它，将如下所示：

```
{
  "query" : {
    "function_score" : {
      "boost_mode" : "replace",
      "query" : { "match_all" : {} },
      "functions" : [
        {
          "script_score" : {
            "script" : "_source.copies * 0.5"
          }
        }
      ]
    }
  }
}
```

(3) 更换custom_filters_score查询

最后要讨论的是custom_filters_score查询，假设有如下查询：

```
{
  "query" : {
    "custom_filters_score" : {
      "query" : { "match_all" : {} },
      "filters" : [
        {
          "filter" : { "term" : { "available" : true }},
          "boost" : 10
        }
      ],
      "score_mode" : "first"
    }
  }
}
```

如果想用function_score查询取代它，将如下所示：

```
{
  "query" : {
    "function_score" : {
      "query" : { "match_all" : {} },
      "functions" : [
        {
          "filter" : { "term" : { "available" : true }},
          "boost_factor" : 10
        }
      ],
      "score_mode" : "first"
    }
  }
}
```

5.5　索引时加权何时有意义

前一节讨论了对查询进行加权。这种类型的加权非常方便和强大，在大多数情况下满足需求。然而，如果当我们在建立索引时就知道哪些文档重要，更方便的方法是使用索引时加权。

我们获得独立于查询的一个加权，成本是重建索引（在加权值变化时，我们需要重建索引）。此外，由于加权过程中已经在索引时计算，性能会稍好一些。Elasticsearch把加权的信息存储为规范化信息的一部分。很重要的是，如果把`omit_norms`设置为`true`，就不能使用索引时加权。

5.5.1　在输入数据中定义字段加权

我们看一个典型的文档定义，如下所示：

```
{
  "title" : "The Complete Sherlock Holmes",
  "author" : "Arthur Conan Doyle",
  "year" : 1936
}
```

如果想为这个特定文档的`author`字段加权，结构应该会略有变化，文档看起来应该如下所示：

```
{
  "title" : "The Complete Sherlock Holmes",
  "author" : {
    "_value" : "Arthur Conan Doyle",
    "_boost" : 10.0,
  },
  "year": 1936
}
```

就是这些。在上述文档被编制到索引后，我们会让Elasticsearch知道`author`字段的重要性大于其他字段。

在旧版本Elasticsearch中，设置文档范围的加权是可能的。然而，从4.0开始，Lucene不支持文档范围的加权，Elasticsearch靠加权所有字段来模拟文档的加权。Elasticsearch 1.0废弃了文档提升，我们决定不写它，因为它将被删除。

5.5.2 在映射中定义加权

值得一提的是可以直接在映射文件中定义字段的加权。下面的示例演示了这一点：

```
{
  "mappings" : {
    "book" : {
      "properties" : {
        "title" : { "type" : "string" },
        "author" : { "type" : "string", "boost" : 10.0 }
      }
    }
  }
}
```

因为上述加权，所有查询将对以author命名的所有字段加权。这也适用于使用_all字段的查询。

5.6 同义词

你应该听说过同义词：有相同或相近意思的词语。有时，当一个词输入搜索框时，我们希望其他一些词也被匹配。回忆一下3.1.1节，有本书叫Crime and Punishment。如果我们希望这本书不仅在搜索crime或punishment时匹配到，也可使用criminality和abuse搜索到，就要使用同义词。

5.6.1 同义词过滤器

为了使用同义词过滤器，我们需要定义自己的分析器，称为synonym，使用空格分词器和一个叫synonym的过滤器。该过滤器的类型属性必须设置为synonym，它告诉Elasticsearch，该过滤器是一个同义词过滤器。此外，我们希望忽略大小写，对大写和小写的同义词一视同仁（设置ignore_case属性为true）。自定义一个使用同义词过滤器的分析器，需要以下映射：

```
{
  "index" : {
    "analysis" : {
      "analyzer" : {
        "synonym" : {
          "tokenizer" : "whitespace",
```

```
            "filter" : [
              "synonym"
            ]
          }
        },
        "filter" : {
          "synonym" : {
            "type" : "synonym",
            "ignore_case" : true,
            "synonyms" : [
              "crime => criminality"
            ]
          }
        }
      }
    }
}
```

1. 映射中的同义词

在前面的定义中，我们在映射中指定了发给Elasticsearch的同义词规则。为此，需要添加synonyms属性，这是一个同义词规则的数组。例如，下面映射的一部分定义了一个同义词规则：

```
"synonyms" : [
  "crime => criminality"
]
```

我们马上将讨论如何定义同义词规则。

2. 存储在文件系统中的同义词

Elasticsearch还允许我们使用基于文件的同义词。为使用文件，需要指定synonyms_path属性，而不是synonyms属性。synonyms_path属性应该设成含有同义词定义的文件名，文件路径应该相对于Elasticsearch的config目录。所以，如果我们将同义词保存在config目录的synonyms.txt文件中，为了使用它，应该把synonyms_path设置成synonyms.txt。

如果要使用存储在文件中的同义词，前面映射文件中的同义词过滤器将如下所示：

```
"filter" : {
  "synonym" : {
    "type" : "synonym",
    "synonyms_path" : "synonyms.txt"
  }
}
```

5.6.2　定义同义词规则

到现在为止，我们讨论了要做什么才能在Elasticsearch中使用同义词扩展。现在，让我们看看允许同义词使用哪些格式。

1. 使用Apache Solr同义词

Apache Lucene世界中最常见的同义词结构可能是Apache Solr用的那个，Apache Solr是基于Lucene的一个搜索引擎，就像Elasticsearch一样。这是Elasticsearch处理同义词的默认方法，以下各节讨论定义一个新同义词的可能性。

(1) 显式同义词

简单的映射允许把一个单词列表映射到其他单词。所以，在我们的例子中，如果要criminality映射到crime、abuse映射到punishment，需要定义以下条目：

```
criminality => crime
abuse => punishment
```

当然，一个单词可以映射到多个单词，多个单词也能被映射到单个单词，如下所示：

```
star wars, wars => starwars
```

前面的示例意味着，star wars和wars将被synonym过滤器换成starwars。

(2) 等效同义词

除了显式映射以外，Elasticsearch允许我们使用等效同义词。例如，下面的定义会使所有单词可交换，你可以使用其中一个单词，来匹配包含其中一个单词的文档：

```
star, wars, star wars, starwars
```

(3) 扩展同义词

使用Apache Solr格式的时候，同义词过滤器允许我们使用一个额外的expand属性。当expand属性设置为true（默认为false），Elasticsearch将所有同义词扩大到所有等价形式。假设我们有以下过滤器配置：

```
"filter" : {
  "synonym" : {
    "type" : "synonym",
    "expand": false,
    "synonyms" : [
      "one, two, three"
    ]
  }
}
```

Elasticsearch将把前面的同义词定义映射如下：

```
one, two, thee => one
```

这意味着，one、two、three将被更改为one。如果expand属性设置为true，相同的同义

词定义将以下列方式解释：

```
one, two, three => one, two, three
```

这基本上意味着左侧的每个单词将扩展为右侧的所有单词。

2. 使用WordNet同义词

如果想使用WordNet结构的同义词（学习更多关于WordNet，请访问http://wordnet.princeton.edu），我们需要为同义词过滤器提供额外的`format`属性，应将其值设置为`wordnet`，以便Elasticsearch理解。

5.6.3　查询时或索引时的同义词扩展

与所有分析器一样，你可能会问我们应该在什么时候使用同义词过滤器：在索引期间，在查询期间或者两者皆可？当然，这取决于你的需求。但是记住，如果使用索引时同义词，在每个同义词更改后，需要重新索引数据。那是因为它们需要重新应用到所有文档。如果只使用查询时同义词，我们可以更新同义词列表，在查询时应用。

5.7　理解解释信息

与数据库相比，使用能执行全文搜索的系统往往不那么显而易见。我们可以同时搜索多个字段，在索引中的数据可以和提供的文档字段值不同（因为分析过程、同义词、缩写等）。甚至更糟的是，默认情况下，搜索引擎按照数据的相关性排序：一个表示文档相对于查询的相似性的数字，这里的关键是相似性。我们已经讨论过，得分需要考虑很多因素：在文档中发现了搜索词，词出现的频率，字段中包含多少词条，等等。这看上去很复杂，而且要想知道为什么一个文档被找到，为什么另一个文档“更好”是不容易的。好在Elasticsearch的一些工具可以回答这些问题，我们现在来看看。

5.7.1　理解字段分析

最常见的问题之一是为什么找不到给定文档。在许多情况下，问题在于映射的定义和分析流程的配置。为了调试分析过程，Elasticsearch提供一个专门的REST API端点，`_analyze`。

让我们先看Elasticsearch的默认分析器返回的信息。运行以下命令：

```
curl -XGET 'localhost:9200/_analyze?pretty' -d 'Crime and Punishment'
```

响应中，我们将得到如下数据：

```
{
  "tokens" : [ {
    "token" : "crime",
    "start_offset" : 0,
    "end_offset" : 5,
    "type" : "<ALPHANUM>",
    "position" : 1
  }, {
    "token" : "punishment",
    "start_offset" : 10,
    "end_offset" : 20,
    "type" : "<ALPHANUM>",
    "position" : 3
  } ]
}
```

我们可以看到，Elasticsearch把输入短语划分为两个标记。处理时，普通词and被省略（因为它在停用词列表中），而其他单词变成了小写。这显示了在分析过程中到底会发生什么。我们也可以提供分析器的名称，例如，可以将前面的命令修改成下面这样：

```
curl -XGET 'localhost:9200/_analyze?analyzer=standard&pretty' -d
  'Crime and Punishment'
```

上述命令使我们能够检查标准分析器是如何分析数据的（会跟我们前面看到的响应有点不同）。

值得注意的是，有另一种分析API的可用形式：我们能够提供分词器和过滤器。要在创建目标映射之前实验一下配置时，它非常方便。调用的示例如下所示：

```
curl -XGET
  'localhost:9200/library/_analyze?tokenizer=whitespace&
    filters=lowercase,kstem&pretty' -d 'John Smith'
```

上面的例子使用了一个分析器，它由whitespace分词器和两个过滤器（lowercase和kstem）组成。

可以看到，分析API可以非常有效地跟踪映射配置中的错误，对解决查询和搜索相关性问题也非常有用。它可以告诉我们分析器如何工作，它们生成的词条是什么，这些词条的属性是什么。有了这些资料，分析查询问题会更容易追踪。

5.7.2 解释查询

除了看在分析期间发生了什么，Elasticsearch让我们可以解释特定的查询和文档是如何计算得分的。看下面的示例：

```
curl -XGET 'localhost:9200/library/book/1/_explain?pretty&q=quiet'
```

上面的命令将返回如下的响应结果：

```
{
  "_index" : "library",
  "_type" : "book",
  "_id" : "1",
  "matched" : true,
  "explanation" : {
    "value" : 0.057534903,
    "description" : "weight(_all:quiet in 0) [PerFieldSimilarity],
      result of:",
    "details" : [ {
      "value" : 0.057534903,
      "description" : "fieldWeight in 0, product of:",
      "details" : [ {
        "value" : 1.0,
        "description" : "tf(freq=1.0), with freq of:",
        "details" : [ {
          "value" : 1.0,
          "description" : "termFreq=1.0"
        } ]
      }, {
        "value" : 0.30685282,
        "description" : "idf(docFreq=1, maxDocs=1)"
      }, {
        "value" : 0.1875,
        "description" : "fieldNorm(doc=0)"
      } ]
    } ]
  }
}
```

看起来很复杂，好吧，确实很复杂！更糟糕的是，这只是一个简单的查询！Elasticsearch，更确切地说，是Lucene库，会显示关于评分过程的内部信息。我们只蜻蜓点水地解释一下上述响应中最重要的东西。

最重要的部分是为文档计算的总分（`explanation`对象的`value`属性）。如果等于`0`，说明文档与给定查询不匹配。另一个重要的元素是`description`部分，告诉我们采用了哪种相似度模型。在我们的示例中，查询`quiet`词条，在`_all`字段被找到。这很直观，因为我们在默认字段中搜索，就是`_all`（参见2.4节）。

`details`部分提供了有关组件的信息，以及我们应在哪里寻求解释：为什么我们的文档与查询匹配。说到评分，我们有一个对象：单一的负责文档得分计算的组件。`value`属性是由此组件计算的得分，然后再次看到`description`和`detail`节点。正如你在`description`字段中看到的，最后的得分（`fieldWeight in 0, product of`）是内部`details`数组中的每个元素所计算得分的组成值（`1.0 * 0.30685282 * 0.1875`）。

在内部的`details`数组里，我们可以看到3个对象。第一个显示了给定字段上的词频信息（在

我们的例子中是1），这意味着字段上只出现一次搜索词条。第二个对象显示了逆文档频率。注意maxDocs属性，它等于1，意味着给定词条只找到了1个文档。第三个对象负责字段的field norm。

请注意，每个查询的上述响应是不同的。而且，查询越复杂，返回信息也会越复杂。

5.8 小结

本章介绍了Apache Lucene评分在内部如何工作，如何使用Elasticsearch的脚本功能，以及如何索引和搜索不同语言的文档。我们使用了不同的查询来改变文档的得分，并且修改查询来让它适合我们的用例。学到了索引时的加权，什么是同义词，以及它们如何帮助我们。最后，学习了如何检查为什么特定的文档是结果集的一部分，以及得分是如何计算的。

下一章将介绍全文搜索之外的内容。看看聚合是什么，以及如何使用它们来分析数据。我们也会看到切面，它能够聚合数据并为其带来意义。使用建议器执行拼写检查和自动完成，采用前瞻性搜索找到匹配特定查询的文档。索引二进制文档，并使用地理空间功能，使用地理数据搜索。最后，我们将使用卷动API有效获取大量结果，还会看到如何让Elasticsearch在查询中使用词条列表（一个自动加载的列表）。