

前一章介绍了Elasticsearch索引的工作原理，如何创建自定义映射，以及可以使用什么数据类型；还在索引中存储了附加的信息；使用了默认和非默认形式的路由。读完本章，我们将了解以下主题：

- ❑ 查询Elasticsearch并选择要返回的数据；
- ❑ Elasticsearch查询过程的工作机制；
- ❑ 了解Elasticsearch提供的基本查询；
- ❑ 筛选查询结果；
- ❑ 了解高亮的工作原理以及如何使用；
- ❑ 验证查询；
- ❑ 探索复合查询；
- ❑ 数据排序。

3.1 查询 Elasticsearch

到目前为止，我们使用了REST API和简单查询或GET请求来搜索数据。更改索引时，无论想执行的操作是更改映射还是文档索引化，都要用REST API向Elasticsearch发送JSON结构的数据。类似地，如果想发送的不是一个简单的查询，仍然把它封装为JSON结构并发送给Elasticsearch。这就是所谓的查询DSL。从更宏观的角度看，Elasticsearch支持两种类型的查询：基本查询和复合查询。基本查询，如词条查询用于查询实际数据，3.3节将介绍。第二种查询为复合查询，如布尔查询，可以合并多个查询，3.4节将讨论。

然而，这不是全部。除了这两种类型的查询，你还可以用过滤查询，根据一定的条件缩小查询结果。不像其他查询，筛选查询不会影响得分，而且通常非常高效。

更加复杂的情况，查询可以包含其他查询（别担心，我们将试着解释这个内容）。此外，一些查询可以包含过滤器，而其他查询可同时包含查询和过滤器。这并不是全部，但暂时先解释这些工作，3.4节和3.5节将详细介绍。

3.1.1 示例数据

如果没有特别说明，一些映射将用于本章的余下部分：

```
{
  "book" : {
    "_index" : {
      "enabled" : true
    },
    "_id" : {
      "index" : "not_analyzed",
      "store" : "yes"
    },
    "properties" : {
      "author" : {
        "type" : "string"
      },
      "characters" : {
        "type" : "string"
      },
      "copies" : {
        "type" : "long",
        "ignore_malformed" : false
      },
      "otitle" : {
        "type" : "string"
      },
      "tags" : {
        "type" : "string"
      },
      "title" : {
        "type" : "string"
      },
      "year" : {
        "type" : "long",
        "ignore_malformed" : false,
        "index" : "analyzed"
      },
      "available" : {
        "type" : "boolean"
      }
    }
  }
}
```



如果没有特别说明，string类型的字段将被分析。

上述映射（保存为mapping.json文件）用来创建library索引。使用下面的命令来运行：

```
curl -XPOST 'localhost:9200/library'
curl -XPUT 'localhost:9200/library/book/_mapping' -d @mapping.json
```

如果没有特别说明,下面的数据在本章通用:

```
{ "index": { "_index": "library", "_type": "book", "_id": "1" }}
{ "title": "All Quiet on the Western Front","otitle": "Im Westen
nichts Neues","author": "Erich Maria Remarque","year":
1929,"characters": ["Paul Bäumer", "Albert Kropp", "Haie
Westhus", "Fredrich Müller", "Stanislaus Katczinsky",
"Tjaden"],"tags": ["novel"],"copies": 1, "available": true,
"section" : 3}
{ "index": { "_index": "library", "_type": "book", "_id": "2" }}
{ "title": "Catch-22","author": "Joseph Heller","year":
1961,"characters": ["John Yossarian", "Captain Aardvark",
"Chaplain Tappman", "Colonel Cathcart", "Doctor
Daneeka"],"tags": ["novel"],"copies": 6, "available" : false,
"section" : 1}
{ "index": { "_index": "library", "_type": "book", "_id": "3" }}
{ "title": "The Complete Sherlock Holmes",
"author": "Arthur Conan Doyle","year": 1936,"characters":
["Sherlock Holmes","Dr. Watson", "G. Lestrade"],"tags":
[],"copies": 0, "available" : false, "section" : 12}
{ "index": { "_index": "library", "_type": "book", "_id": "4" }}
{ "title": "Crime and Punishment","otitle": "Преступление и
наказание","author": "Fyodor Dostoevsky","year":
1886,"characters": ["Raskolnikov", "Sofia Semyonovna
Marmeladova"],"tags": [],"copies": 0, "available" : true}
```

把上面数据保存在documents.json文件里,使用下面的命令来索引化:

```
curl -s -XPOST 'localhost:9200/_bulk' --data-binary @documents.json
```

该命令执行批量索引,你可以在2.3节中学习更多有关内容。

3.1.2 简单查询

查询Elasticsearch最简单的办法是使用URI请求查询,1.5节已经讨论过。例如,为了搜索title字段中的crime一词,使用下面的命令:

```
curl -XGET 'localhost:9200/library/book/_search?q=title:crime&pretty=true'
```

这种查询方式很简单,但比较局限。如果从Elasticsearch的查询DSL的视点来看,上面的查询是一种query_string查询,它查询title字段中含有crime一词的文档,可以这样写:

```
{
  "query" : {
    "query_string" : { "query" : "title:crime" }
  }
}
```

采用查询DSL来发送查询有点不同,但也不是什么高深的东西。我们和以前一样发送HTTP GET请求到_search这个REST端点,并在请求主体中附上查询。来看看下面的命令:

```
curl -XGET 'localhost:9200/library/book/_search?pretty=true' -d '{
  "query" : {
    "query_string" : { "query" : "title:crime" }
  }
}'
```

可以看到，我们使用请求体（-d参数）把整个JSON格式的查询发到Elasticsearch。pretty=true参数让Elasticsearch以更容易阅读的方式返回响应。上述命令的响应如下所示：

```
{
  "took" : 1,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 1,
    "max_score" : 0.15342641,
    "hits" : [ {
      "_index" : "library",
      "_type" : "book",
      "_id" : "4",
      "_score" : 0.15342641, "_source" : { "title": "Crime and
      Punishment", "otitle": "Преступление и наказание", "author":
      "Fyodor Dostoevsky", "year": 1886, "characters":
      ["Raskolnikov", "Sofia Semyonovna Marmeladova"], "tags":
      [], "copies": 0, "available" : true}
    } ]
  }
}
```

很好！我们得到了使用查询DSL的第一个搜索结果。

3.1.3 分页和结果集大小

正如我们所期望的，Elasticsearch能控制想要的结果数以及想从哪个结果开始。下面是在请求体中添加的两个额外参数。

- ❑ from: 该属性指定我们希望在结果中返回的起始文档。它的默认值是0，表示想要得到从第一个文档开始的结果。
- ❑ size: 该属性指定了一次查询中返回的最大文档数，默认值为10。如果只对切面结果感兴趣，并不关心文档本身，可以把这个参数设置成0。

如果能让查询从第10个文档开始返回20个文档，可以发送如下查询：

```
{
  "from" : 9,
```

```

"size" : 20,
"query" : {
  "query_string" : { "query" : "title:crime" }
}
}

```



下载示例代码。如果你是用账号从<http://www.packtpub.com>买的书，可以到上面下载示例代码。如果你是从别的地方买的书，可以访问<http://www.PacktPub.com/support>并注册，把代码文件直接发到你的电子邮箱。

3.1.4 返回版本值

除了所有返回的信息以外，Elasticsearch还可以返回文档的版本。为此，需要在JSON对象的最上层添加version属性并把值设为true。所以，要求返回版本信息的查询，最终将如下所示：

```

{
  "version" : true,
  "query" : {
    "query_string" : { "query" : "title:crime" }
  }
}

```

执行上面的查询后，得到如下结果：

```

{
  "took" : 2,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 1,
    "max_score" : 0.15342641,
    "hits" : [ {
      "_index" : "library",
      "_type" : "book",
      "_id" : "4",
      "_version" : 1,
      "_score" : 0.15342641, "_source" : { "title": "Crime and
        Punishment", "otitle": "ПреступлѣнХие и наказáние",
        "author": "Fyodor Dostoevsky", "year": 1886,
        "characters": ["Raskolnikov", "Sofia Semyonovna
        Marmeladova"], "tags": [], "copies": 0, "available" : true}
    } ]
  }
}

```

可以看到，_version属性出现在返回的唯一hit对象中。

3.1.5 限制得分

对于非标准用例，Elasticsearch提供一项功能，让我们可以根据文档需要满足的最低得分值，来过滤结果。为了用此功能，必须在JSON顶层提供`min_score`属性和最低得分值。例如，希望我们的查询只返回得分高于0.75的文档，发出以下查询：

```
{
  "min_score" : 0.75,
  "query" : {
    "query_string" : { "query" : "title:crime" }
  }
}
```

执行后得到如下响应：

```
{
  "took" : 1,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 0,
    "max_score" : null,
    "hits" : [ ]
  }
}
```

看下之前那个例子，文档的得分为0.153 426 41，比0.75低，所以这次没有得到任何文档。限制得分一般没太大意义，因为一般来说在查询之间比较得分很困难。也许在某些情况下，你将需要这个功能。

3.1.6 选择需要返回的字段

在请求主体中使用字段数组，可以定义在响应中包含哪些字段。记住，你只能返回那些在用于创建索引的映射中标记为存储的字段，或者你使用了`_source`字段（Elasticsearch使用`_source`字段提供存储字段）。因此，要让每个结果中的文档只返回`title`和`year`字段，发送下面的查询到Elasticsearch：

```
{
  "fields" : [ "title", "year" ],
  "query" : {
    "query_string" : { "query" : "title:crime" }
  }
}
```

在响应中，得到如下输出：

```

{
  "took" : 2,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 1,
    "max_score" : 0.15342641,
    "hits" : [ {
      "_index" : "library",
      "_type" : "book",
      "_id" : "4",
      "_score" : 0.15342641,
      "fields" : {
        "title" : [ "Crime and Punishment" ],
        "year" : [ 1886 ]
      }
    } ]
  }
}

```

可以看到，一切按预期工作。与你分享以下3点：

- ❑ 如果没有定义fields数组，它将用默认值，如果有就返回_source字段；
- ❑ 如果使用_source字段，并且请求一个没有存储的字段，那么这个字段将从_source字段中提取（然而，这需要额外的处理）；
- ❑ 如果想返回所有的存储字段，只需传入星号（*）作为字段名字。



从性能的角度，返回_source字段比返回多个存储字段更好。

部分字段

除可以选择要返回哪些字段外，Elasticsearch允许使用所谓部分字段。可以通过它来控制字段是如何从_source字段加载的。Elasticsearch公开了部分字段对象的include和exclude属性，所以可以基于这些属性来包含或排除字段。例如，为了在查询中包括以titl开头且排除以chara开头的字段，发出以下查询：

```

{
  "partial_fields" : {
    "partial1" : {
      "include" : [ "titl*" ],
      "exclude" : [ "chara*" ]
    }
  },
  "query" : {

```

```

    "query_string" : { "query" : "title:crime" }
  }
}

```

3.1.7 使用脚本字段

可以在Elasticsearch中返回脚本计算字段：在JSON的查询对象中加上script_fields部分，添加上每个想返回的脚本值的名字。若要返回一个叫correctYear的值，它用year字段减去1800计算得来，运行以下查询：

```

{
  "script_fields" : {
    "correctYear" : {
      "script" : "doc['year'].value - 1800"
    }
  },
  "query" : {
    "query_string" : { "query" : "title:crime" }
  }
}

```

我们在上面的示例中使用了doc符号，它让我们捕获了返回结果，从而让脚本执行速度更快，但也导致了更高的内存消耗，并且限制了只能用单个字段的单个值。如果关心内存的使用，或者使用的是更复杂的字段值，可以用_source字段。使用此字段的查询如下所示：

```

{
  "script_fields" : {
    "correctYear" : {
      "script" : "_source.year - 1800"
    }
  },
  "query" : {
    "query_string" : { "query" : "title:crime" }
  }
}

```

这个查询将返回如下响应：

```

{
  "took" : 1,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 1,
    "max_score" : 0.15342641,
    "hits" : [ {
      "_index" : "library",

```



```

    "_type" : "book",
    "_id" : "4",
    "_score" : 0.15342641,
    "fields" : {
      "correctYear" : [ 86 ]
    }
  } ]
}

```

可以看到，我们在响应中得到了correctYear这个计算字段。

传参数到脚本字段中

再看一个脚本字段的特性：可传入额外的参数。可以使用一个变量名称，并把值传入params节中，而不是直接把1800写在等式中。这样做以后，查询将如下所示：

```

{
  "script_fields" : {
    "correctYear" : {
      "script" : "_source.year - paramYear",
      "params" : {
        "paramYear" : 1800
      }
    }
  },
  "query" : {
    "query_string" : { "query" : "title:crime" }
  }
}

```

可以看到，我们在脚本的等式中添加了paramYear变量，并在params节中提供了变量的值。

5.2节将介绍关于脚本使用的更多内容。

3.2 理解查询过程

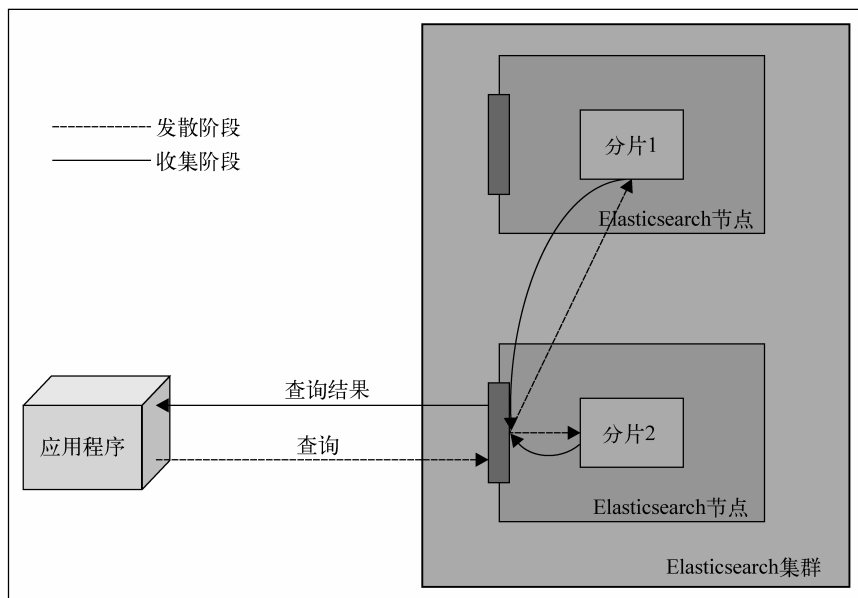
读完上一节后，我们知道了Elasticsearch的查询的工作原理。要知道在大多数情况下，Elasticsearch需要分散查询到多个节点中，得到结果，合并它们，再获取有关文档并返回结果。我们还没谈到的是另外三个定义查询行为的东西：查询重写、搜索类型和查询执行偏好。现在，将注意力集中在Elasticsearch的这些功能上，试着展示查询是如何工作的。

3.2.1 查询逻辑

Elasticsearch是一个分布式搜索引擎，因此提供的所有功能在性质上都必须是分布式的。查询也是如此。既然我们想讨论一些更高级的、关于如何控制查询过程的主题，首先需要知道它是

如何工作的。

默认情况下，如果我们什么都不改变，查询过程将分为两个阶段，如下图所示：



查询发送到Elasticsearch的其中一个节点，这时发生的是一个所谓的发散阶段。查询分布到建立过索引的所有分片上。如果它建立在5个分片和1个副本基础上，那么，这5个实体分片都会被查询（不需要同时查询分片及其副本，因为它们包含相同的数据）。每个查询的分片将只返回文档的标识符和得分。发送分散查询的节点将等待所有的分片完成它们的任务，收集结果并适当排序（在这种情况下，按得分从低到高）。

之后，将发送一个新的请求来生成搜索结果。然而，这次请求将只发送到那些持有响应所需文档的分片上。在大多数情况下，Elasticsearch不会把请求发送到所有的分片，而只是发送给其中的一部分。这是因为通常不需要整个查询结果，只要一部分。这一阶段被称为收集阶段（gather phase）。收集完所有文档，将建立最终响应，并返回查询结果。

当然，上述是Elasticsearch的默认行为，可以改变。以下部分将描述如何更改此行为。

3.2.2 搜索类型

Elasticsearch允许通过指定搜索类型来选择查询在内部如何处理。不同的搜索类型适合不同的情况；可以只在乎性能，但有时查询的关联性可能是最重要的因素。你应该记得每个分片是一个小的Lucene索引，为了返回更多相关的结果，频率等信息需要在分片之间传输。为了控制查询如何执行，可以使用search_type请求参数，并将其设置为下列值之一。

- ❑ `query_then_fetch`: 第一步, 执行查询得到对文档进行排序和分级所需信息。这一步在所有的分片上执行。然后, 只在相关分片上查询文档的实际内容。不同于`query_and_fetch`, 此查询类型返回结果的最大数量等于`size`参数的值。如果没有指定搜索类型, 就默认使用这个类型, 前面描述过。
- ❑ `query_and_fetch`: 这通常是最快也最简单的搜索类型实现。查询在所有分片上并行执行(当然, 任意一个主分片, 只查询一个副本), 所有分片返回等于`size`值的结果数。返回文档的最大数量等于`size`的值乘以分片的数量。
- ❑ `dfs_query_and_fetch`: 这个跟`query_and_fetch`类似, 但相比`query_and_fetch`, 它包含一个额外阶段, 在初始查询中执行分布式词频的计算, 以得到返回文件的更精确的得分, 从而让查询结果更相关。
- ❑ `dfs_query_then_fetch`: 与前一个`dfs_query_and_fetch`一样, `dfs_query_then_fetch`类似于相应的`query_then_fetch`, 但比`query_then_fetch`多了一个额外的阶段, 就像`dfs_query_and_fetch`一样。
- ❑ `count`: 这是一个特殊的搜索, 只返回匹配查询的文档数。如果你只需要结果数量, 而不关心文档, 应该使用这个搜索类型。
- ❑ `scan`: 这是另一个特殊的搜索类型, 只有在要让查询返回大量结果时才用。它跟一般的查询有点不同, 因为在发送第一个请求之后, Elasticsearch响应一个滚动标识符, 类似于关系型数据库中的游标。所有查询需要在`_search/scroll` REST端点运行, 并需要在请求主体中发送返回的滚动标识符。6.7节将介绍它的更多功能。

所以, 如果想使用最简单的搜索类型, 可以执行以下命令:

```
curl -XGET 'localhost:9200/library/book/_search?pretty=true&search_type=query_and_fetch' -d '{
  "query" : {
    "term" : { "title" : "crime" }
  }
}'
```

3.2.3 搜索执行偏好

除了可以控制查询是如何执行的, 也可以控制在哪些分片上执行查询。默认情况下, Elasticsearch使用的分片和副本, 既包含我们已经发送过请求的可用节点, 又包括集群中的其他节点。而且, 在大多数情况下, 默认行为是最佳的查询首选方法。有时我们要更改默认行为, 例如, 可能希望只在主分片上执行搜索。为此, 可以设置偏好请求参数, 设为下面表中的其中一个值:

参 数 值	描 述
<code>_primary</code>	只在主分片上执行搜索, 不使用副本。当想使用索引中最近更新的、还没复制到副本中的信息, 这个是很有用的
<code>_primary_first</code>	如果主分片可用, 只在主分片上执行搜索, 否则才在其他分片上执行

(续)

参 数 值	描 述
<code>_local</code>	在可能的情况下，只在发送请求的节点上的可用分片上执行搜索
<code>_only_node:node_id</code>	只在提供标识符的节点上执行搜索
<code>_prefer_node:node_id</code>	Elasticsearch将尝试在提供标识符的节点上执行搜索。如果该节点不可用，则使用其他的可用节点
<code>_shards:1,2</code>	Elasticsearch将在提供标识符的分片上执行操作(在这个例子中,分片1和2)。 <code>_shards</code> 参数可以和其他首选项合并，但 <code>_shards</code> 标识符必须在前面，比如 <code>_shards:1,2;_local</code>
自定义值	可以传入任何自定义字符串值，具有相同值的请求将在相同的分片上执行

如果只想在本地分片上执行查询，可以执行如下的命令：

```
curl -XGET 'localhost:9200/library/_search?preference=_local' -d '{
  "query" : {
    "term" : { "title" : "crime" }
  }
}'
```

3

3.2.4 搜索分片 API

在讨论搜索偏好时，还想提到Elasticsearch所公开的搜索分片API。此API允许检查将执行查询的分片。需要在 `_search_shards` REST端点执行这个API。若要查看查询如何执行，运行以下命令：

```
curl -XGET 'localhost:9200/library/_search_shards?pretty' -d
'{"query":"match_all":{}}'
```

该命令将返回如下响应：

```
{
  "nodes" : {
    "N0iP_bH3QriX4NpqsqSUA" : {
      "name" : "Oracle",
      "transport_address" : "inet[/192.168.1.19:9300]"
    }
  },
  "shards" : [ [ {
    "state" : "STARTED",
    "primary" : true,
    "node" : "N0iP_bH3QriX4NpqsqSUA",
    "relocating_node" : null,
    "shard" : 0,
    "index" : "library"
  } ], [ {
    "state" : "STARTED",
    "primary" : true,
    "node" : "N0iP_bH3QriX4NpqsqSUA",
    "relocating_node" : null,
```

```

    "shard" : 1,
    "index" : "library"
  } ], [ {
    "state" : "STARTED",
    "primary" : true,
    "node" : "N0iP_bH3QriX4NpqsqSUAq",
    "relocating_node" : null,
    "shard" : 4,
    "index" : "library"
  } ], [ {
    "state" : "STARTED",
    "primary" : true,
    "node" : "N0iP_bH3QriX4NpqsqSUAq",
    "relocating_node" : null,
    "shard" : 3,
    "index" : "library"
  } ], [ {
    "state" : "STARTED",
    "primary" : true,
    "node" : "N0iP_bH3QriX4NpqsqSUAq",
    "relocating_node" : null,
    "shard" : 2,
    "index" : "library"
  } ] ]
}

```

可以看到，在Elasticsearch返回的响应中，有关于在查询过程中使用的分片信息。当然，对于搜索分片API，可以使用所有参数，如routing或preference，看看它对搜索执行的影响。

3.3 基本查询

Elasticsearch具有广泛的搜索和数据分析能力，以不同的查询、筛选和聚合等形式公开。本节将集中讨论Elasticsearch提供的基本查询。

3.3.1 词条查询

词条查询是Elasticsearch中的一个简单查询。它仅匹配在给定字段中含有该词条的文档，而且是确切的、未经分析的词条。最简单的词条查询如下所示：

```

{
  "query" : {
    "term" : {
      "title" : "crime"
    }
  }
}

```

上述查询将匹配title字段中含有crime一词的文档。记住，词条查询是未经分析的，因此需

要提供跟索引文档中的词条完全匹配的词条。请注意，在输入数据中，`title`字段含有Crime and Punishment，但我们使用小写开头的`crime`来搜索。因为Crime一词在建立索引时已经变成了`crime`。

除了想找的词条外，还可以在词条查询中包含加权属性，它影响给定词条的重要程度。5.1节将对此进行详细讨论。现在，只需记住它改变查询中给定词条的重要程度。

为了修改前面的查询，给它一个10.0的加权，可以发送如下查询：

```
{
  "query" : {
    "term" : {
      "title" : {
        "value" : "crime",
        "boost" : 10.0
      }
    }
  }
}
```

3

你可以看到，我们对查询做了点改变。不再是一个简单的词条，嵌套了一个新的JSON对象包含`value`属性和`boost`属性。`value`属性的值包含我们感兴趣的词条，`boost`属性的值是我们想使用的加权值。

3.3.2 多词条查询

多词条查询允许匹配那些在内容中含有某些词条的文档。词条查询允许匹配单个未经分析的词条，多词条查询可以用来匹配多个这样的词条。假设想得到所有在`tags`字段中含有`novel`或`book`的文档。运行以下查询来达到目的：

```
{
  "query" : {
    "terms" : {
      "tags" : [ "novel", "book" ],
      "minimum_match" : 1
    }
  }
}
```

上述查询返回在`tags`字段中包含一个或两个搜索词条的所有文档。为什么？这是因为我们把`minimum_match`属性设置为1；这意味着至少有1个词条应该匹配。如果想要查询匹配所有词条的文档，可以把`minimum_match`属性设置为2。

3.3.3 `match_all` 查询

`match_all`查询是Elasticsearch中最简单的查询之一。它使我们能够匹配索引中的所有文件。

如果想得到索引中的所有文档，只需运行以下查询：

```
{
  "query" : {
    "match_all" : {}
  }
}
```

也可以在查询中包含加权值，它将赋给所有跟它匹配的文档。比如，在match_all查询中给所有文档加上2.0的加权，可以发送以下查询：

```
{
  "query" : {
    "match_all" : {
      "boost" : 2.0
    }
  }
}
```

3.3.4 常用词查询

常用词查询是在没有使用停用词（stop word，http://en.wikipedia.org/wiki/Stop_words）的情况下，Elasticsearch为了提高常用词的查询相关性和精确性而提供的一个现代解决方案。例如，“crime and punishment”可以翻译成3个词查询，每一个都有性能上的成本（词越多，查询性能越低）。但“and”这个词非常常见，对文档得分的影响非常低。解决办法是常用词查询，将查询分为两组。第一组包含重要的词，出现的频率较低。第二组包含较高频率的、不那么重要的词。先执行第一个查询，Elasticsearch从第一组的所有词中计算分数。这样，通常都很重要的低频词总是被列入考虑范围。然后，Elasticsearch对第二组中的词执行二次查询，但只为与第一个查询中匹配的文档计算得分。这样只计算了相关文档的得分，实现了更高的性能。

一个常用词查询的例子如下：


```
{
  "query" : {
    "common" : {
      "title" : {
        "query" : "crime and punishment",
        "cutoff_frequency" : 0.001
      }
    }
  }
}
```

查询可使用下列参数。

- ❑ query: 这个参数定义了实际的查询内容。
- ❑ cutoff_frequency: 这个参数定义一个百分比（0.001表示0.1%）或一个绝对值（当此

属性值 ≥ 1 时)。这个值用来构建高、低频词组。此参数设置为0.001意味着频率 $\leq 0.1\%$ 的词将出现在低频词组中。

- ❑ `low_freq_operator`: 这个参数可以设为`or`或`and`, 默认是`or`。它用来指定为低频词组构建查询时用到的布尔运算符。如果希望所有的词都在文档中出现才认为是匹配, 应该把它设置为`and`。
- ❑ `high_freq_operator`: 这个参数可以设为`or`或`and`, 默认是`o`。它用来指定为高频词组构建查询时用到的布尔运算符。如果希望所有的词都在文档中出现才认为是匹配, 那么应该把它设置为`and`。
- ❑ `minimum_should_match`: 不使用`low_freq_operator`和`high_freq_operator`参数的话, 可以使用`minimum_should_match`参数。和其他查询一样, 它允许指定匹配的文档中应该出现的查询词的最小个数。
- ❑ `boost`: 这个参数定义了赋给文档得分的加权值。
- ❑ `analyzer`: 这个参数定义了分析查询文本时用到的分析器名称。默认值为`default analyzer`。
- ❑ `disable_coord`: 此参数的值默认为`false`, 它允许启用或禁用分数因子的计算, 该计算基于文档中包含的所有查询词的分数。把它设置为`true`, 得分不那么精确, 但查询将稍快。

 不像词条查询和多词条查询, 常用词查询是经过Elasticsearch分析的。

3.3.5 match 查询

`match`查询把`query`参数中的值拿出来, 加以分析, 然后构建相应的查询。使用`match`查询时, Elasticsearch将对一个字段选择合适的分析器, 所以可以确定, 传给`match`查询的词条将被建立索引时相同的分析器处理。请记住, `match`查询 (以及将在稍后解释的`multi_match`查询) 不支持Lucene查询语法。但是, 它是完全符合搜索需求的一个查询处理器。最简单也是默认的`match`查询如下所示:

```
{
  "query" : {
    "match" : {
      "title" : "crime and punishment"
    }
  }
}
```

上面的查询将匹配所有在`title`字段含有`crime`、`and`或`punishment`词条的文档。这只是最简单的一个查询, 现在来讨论`match`查询的几种类型。

1. 布尔值匹配查询

布尔匹配查询分析提供的文本,然后做出布尔查询。有几个参数允许控制布尔查询匹配行为,如下所示。

- ❑ **operator**: 此参数可以接受or和and,控制用来连接创建的布尔条件的布尔运算符。默认值是or。如果希望查询中的所有条件都匹配,可以使用and运算符。
- ❑ **analyzer**: 这个参数定义了分析查询文本时用到的分析器的名字。默认值为default analyzer。
- ❑ **fuzziness**: 可以通过提供此参数的值来构建模糊查询(fuzzy query)。它为字符串类型提供从0.0到1.0的值。构造模糊查询时,该参数将用来设置相似性。
- ❑ **prefix_length**: 此参数可以控制模糊查询的行为。有关此参数值的更多信息,请参阅3.3.11节。
- ❑ **max_expansions**: 此参数可以控制模糊查询的行为。有关此参数值的更多信息,请参阅3.3.11节。
- ❑ **zero_terms_query**: 该参数允许指定当所有的词条都被分析器移除时(例如,因为停止词),查询的行为。它可以被设置为none或all,默认值是none。在分析器移除所有查询词条时,该参数设置为none,将没有文档返回;设置为all,则将返回所有文档。
- ❑ **cutoff_frequency**: 该参数允许将查询分解成两组:一组低频词和一组高频词。参阅3.3.4节,看看这个参数怎么用。

这些参数应该封装在运行查询的字段名称里。所以如果想对title字段运行一个简单的布尔匹配查询,发送如下查询:

```
{
  "query" : {
    "match" : {
      "title" : {
        "query" : "crime and punishment",
        "operator" : "and"
      }
    }
  }
}
```

2. match_phrase查询

match_phrase查询类似于布尔查询,不同的是,它从分析后的文本中构建短语查询,而不是布尔子句。该查询可以使用下面几种参数。

- ❑ **slop**: 这是一个整数值,该值定义了文本查询中的词条和词条之间可以有多少个未知词条,以被视为跟一个短语匹配。此参数的默认值是0,这意味着,不允许有额外的词条^①。

^① slop为1时,“a b”和“a and b”被视为匹配。——译者注

□ **analyzer**: 这个参数定义了分析查询文本时用到的分析器的名字。默认值为default analyzer。

下面是一段对title字段进行match_phrase查询的示例代码:

```
{
  "query" : {
    "match_phrase" : {
      "title" : {
        "query" : "crime punishment",
        "slop" : 1
      }
    }
  }
}
```

注意, 我们从查询中移除了and一词, 但因为slop参数设置为1, 它仍将匹配我们的文档。

3. match_phrase_prefix查询

match_query查询的最后一种类型是match_phrase_prefix查询。此查询跟match_phrase查询几乎一样, 但除此之外, 它允许查询文本的最后一个词条只做前缀匹配。此外, 除了match_phrase查询公开的参数, 还公开了一个额外参数max_expansions。这个参数控制有多少前缀将被重写成最后的词条。我们的示例查询若改用match_phrase前缀来写, 将如下所示:

```
{
  "query" : {
    "match_phrase_prefix" : {
      "title" : {
        "query" : "crime and punishm",
        "slop" : 1,
        "max_expansions" : 20
      }
    }
  }
}
```

注意, 我们没有提供完整的“crime and punishment”短语, 而只是提供“crime and punishm”, 该查询仍将匹配我们的文档。

3.3.6 multi_match 查询

multi_match查询和match查询一样, 不同的是它不是针对单个字段, 而是可以通过fields参数针对多个字段查询。当然, match查询中可以使用的参数同样可以在multi_match查询中使用。所以, 如果想修改match查询, 让它针对title和otitle字段运行, 那么运行以下查询:

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

```

    "query" : "crime punishment",
    "fields" : [ "title", "otitle" ]
  }
}

```

除了之前提到的参数，multi_match查询还可以使用以下额外的参数来控制它的行为。

- ❑ use_dis_max: 该参数定义一个布尔值，设置为true时，使用析取最大分查询，设置为false时，使用布尔查询。默认值为true。3.3.18节将讨论更多细节。
- ❑ tie_breaker: 只有在use_dis_max参数设为true时才会使用这个参数。它指定低分数项和最高分数项之间的平衡。3.3.18节将介绍更多细节。

3.3.7 query_string 查询

相比其他可用的查询，query_string查询支持全部的Apache Lucene查询语法，1.5.3节讨论过。它使用一个查询解析器把提供的文本构建成实际的查询，例子如下所示：

```

{
  "query" : {
    "query_string" : {
      "query" : "title:crime^10 +title:punishment -otitle:cat
+author:(+Fyodor +dostoevsky)",
      "default_field" : "title"
    }
  }
}

```

我们已经熟悉了Lucene查询语法的基础知识，所以可以讨论上述查询的工作原理。正如你所看到的，我们想得到在title字段中包含crime词条的文档，并且这些文档应该有10的加权。接下来，我们希望文档在title字段中包含punishment，而在otitle字段中不包含cat。最后，告诉Lucene我们只希望文档的作者字段中包含Fyodor和dostoevsky词条。

像大多数Elasticsearch查询一样，query_string提供下列参数控制查询行为。

- ❑ query: 此参数指定查询文本。
- ❑ default_field: 此参数指定默认的查询字段，默认值由index.query.default_field属性指定，默认为_all。
- ❑ default_operator: 此参数指定默认的逻辑运算符（or或and），默认值是or。
- ❑ allow_leading_wildcard: 此参数指定是否允许通配符作为词条的第一个字符，默认值为true。
- ❑ lowercase_expand_terms: 此参数指定查询重写是否把词条变成小写，默认值为true，意味着重写后的词条将小写。
- ❑ enable_position_increments: 此参数指定查询结果中的位置增量是否打开，默认值是true。

- ❑ `fuzzy_max_expansions`: 使用模糊查询时, 此参数指定模糊查询可被扩展到的最大词条数, 默认值是50。
- ❑ `fuzzy_prefix_length`: 此参数指定生成的模糊查询中的前缀长度, 默认值为0。欲了解更多信息, 请参阅3.3.11节。
- ❑ `fuzzy_min_sim`: 此参数指定模糊查询的最小相似度, 默认值为0.5。欲了解更多信息, 请参阅3.3.11节。
- ❑ `phrase_slop`: 此参数指定短语溢出值, 默认值为0。欲了解更多信息, 请参阅3.3.5节。
- ❑ `boost`: 此参数指定使用的加权值, 默认值为1.0。
- ❑ `analyze_wildcard`: 此参数指定是否应该分析通配符查询生成的词条, 默认为false, 意味着词条不会被分析。
- ❑ `auto_generate_phrase_queries`: 此参数指定是否自动生成短语查询。其默认值为false, 这意味着不会自动生成。
- ❑ `minimum_should_match`: 此参数控制有多少生成的Boolean `should`子句必须与文档匹配, 才能认为它是匹配的。它可以是百分比, 例如50%, 这意味着至少有50%的给定词条必须匹配。它也可以是整数值, 如2, 这意味着至少2个词条必须匹配。
- ❑ `lenient`: 此参数的取值true或false。如果设置为true, 格式方面的失败将被忽略。

DisMax是Disjunction Max的缩写。Disjunction指搜索执行可以跨多个字段, 每个字段可以给予不同的权重。Max意味着, 对于给定词条, 只有最高分会包括在最后的文档评分中, 而不是所有包含该词条的所有字段分数之和(简单的布尔查询才会这样)。

注意, Elasticsearch可以重写`query_string`查询, 正因为如此, Elasticsearch使我们能够传递额外的参数来控制重写方法。有关此过程的详细信息, 请参阅3.2节。

针对多字段的`query_string`查询

针对多个字段做`query_string`查询是可能的。为此, 需要在查询主体中提供一个`fields`参数, 它是个持有字段名称的数组。有两种方法针对多个字段运行`query_string`查询; 默认方法是采用布尔查询来构造查询, 另一种是使用最大分查询。

使用最大分查询要在查询主体中添加`use_dis_max`属性并将其设置为true。示例查询如下:

```
{
  "query" : {
    "query_string" : {
      "query" : "crime punishment",
      "fields" : [ "title", "otitle" ],
      "use_dis_max" : true
    }
  }
}
```

3.3.8 simple_query_string 查询

simple_query_string查询使用Lucene的最新查询解析器之一：SimpleQueryParser。类似字符串查询，它接受Lucene查询语法；然而不同的是，simple_query_string查询在解析错误时不会抛出异常。它丢弃查询无效的部分，执行其余部分，示例如下：

```
{
  "query" : {
    "simple_query_string" : {
      "query" : "title:crime^10 +title:punishment -otitle:cat
        +author:(+Fyodor +dostoevsky)",
      "default_operator" : "and"
    }
  }
}
```

3.3.9 标识符查询

标识符查询是一个简单的查询，仅用提供的标识符来过滤返回的文档。此查询针对内部的_uid字段运行，所以它不需要启用_id字段。最简单的版本类似于下面的代码：

```
{
  "query" : {
    "ids" : {
      "values" : [ "10", "11", "12", "13" ]
    }
  }
}
```

此查询只返回具有values数组中一个标识符的文档。也可以把标识符查询变得复杂一点，限制文档为特定的类型。例如，只包括book类型的文档，发出以下查询：

```
{
  "query" : {
    "ids" : {
      "type" : "book",
      "values" : [ "10", "11", "12", "13" ]
    }
  }
}
```

可以看到，我们在查询中添加了type属性，并设置其值为我们感兴趣的类型。

3.3.10 前缀查询

前缀查询在配置方面来说跟词条查询类似。前缀查询能让我们匹配这样的文档：它们的特定字段以给定的前缀开始。例如，想找到所有title字段以cri开始的文档，可以运行以下查询：

```
{
  "query" : {
    "prefix" : {
      "title" : "cri"
    }
  }
}
```

与词条查询类似，还可以在前缀查询中包含加权属性；这将影响到给定前缀的重要性。例如，改变之前的查询，并给它增加3.0的加权，发出以下查询：

```
{
  "query" : {
    "prefix" : {
      "title" : {
        "value" : "cri",
        "boost" : 3.0
      }
    }
  }
}
```



Elasticsearch会把前缀查询重写，也允许我们传递额外的参数来控制重写方法。有关此过程的详细信息，请参阅3.2节。

3

3.3.11 fuzzy_like_this 查询

fuzzy_like_this查询类似于more_like_this查询。它查找所有与提供的文本类似的文档，但是它有点不同于more_like_this查询。它利用模糊字符串并选择生成的最佳差分词条。如果针对title和otitle字段的fuzzy_like_this查询来查找所有类似于crime punishment的文档，可以运行以下查询：

```
{
  "query" : {
    "fuzzy_like_this" : {
      "fields" : ["title", "otitle"],
      "like_text" : "crime punishment"
    }
  }
}
```

fuzzy_like_this查询支持以下查询参数。

- ❑ fields: 此参数定义应该执行查询的字段数组，默认值是_all字段。
- ❑ like_text: 这是一个必需参数，包含用来跟文档比较的文本。
- ❑ ignore_tf: 此参数指定在相似度计算期间，是否应忽略词频，默认值为false，意味着将使用词频。

- ❑ `max_query_terms`: 此参数指定生成的查询中能包括的最大查询词条数, 默认值为25。
- ❑ `min_similarity`: 此参数指定差分词条 (differencing terms) 应该有的最小相似性, 默认值为0.5。
- ❑ `prefix_length`: 此参数指定差分词条的公共前缀长度, 默认值为0。
- ❑ `boost`: 此参数指定使用的加权值, 默认值为1.0。
- ❑ `analyzer`: 这个参数定义了分析所提供文本时用到的分析器名称。

3.3.12 `fuzzy_like_this_field` 查询

`fuzzy_like_this_field`查询和`fuzzy_like_this`查询类似, 但它只能对应单个字段。正因为如此, 它不支持多字段属性。作为替代, 应该把查询参数封装到字段名称中。查询`title`字段的一个示例查询类似于下面这样:

```
{
  "query" : {
    "fuzzy_like_this_field" : {
      "title" : {
        "like_text" : "crime and punishment"
      }
    }
  }
}
```

`fuzzy_like_this`查询的其他所有参数也可以用在`fuzzy_like_this_field`中。

3.3.13 `fuzzy` 查询

`fuzzy`查询是模糊查询中的第三种类型, 它基于编辑距离算法来匹配文档。编辑距离的计算基于我们提供的查询词条和被搜索文档。此查询很占用CPU资源, 但当需要模糊匹配时它很有用, 例如, 当用户拼写错误时。在我们的示例中, 假设用户向搜索框中输入单词`crme`, 而不是`crime`, 运行模糊查询的最简单形式如下所示:

```
{
  "query" : {
    "fuzzy" : {
      "title" : "crme"
    }
  }
}
```

查询响应如下所示:

```
{
  "took" : 1,
  "timed_out" : false,
```

```

    "_shards" : {
      "total" : 5,
      "successful" : 5,
      "failed" : 0
    },
    "hits" : {
      "total" : 1,
      "max_score" : 0.15342641,
      "hits" : [ {
        "_index" : "library",
        "_type" : "book",
        "_id" : "4",
        "_score" : 0.15342641, "_source" : { "title": "Crime and
        Punishment", "otitle": "Преступление и наказание",
        "author": "Fyodor Dostoevsky", "year": 1886,
        "characters": ["Raskolnikov", "Sofia Semyonovna
        Marmeladova"], "tags": [], "copies": 0, "available" : true}
      } ]
    }
  }
}

```

即使犯了一个拼写错误，Elasticsearch仍然设法找到我们感兴趣的文档。

可以使用下面的参数来控制fuzzy查询的行为。

- ❑ value: 此参数指定了实际的查询。
- ❑ boost: 此参数指定了查询的加权值，默认为1.0。
- ❑ min_similarity: 此参数指定了一个词条被算作匹配所必须拥有的最小相似度。对字符串字段来说，这个值应该在0到1之间，包含0和1。对于数值型字段，这个值可以大于1，比如查询值是20，min_similarity设为3，则可以得到17~23的值。对于日期字段，可以把min_similarity参数值设为1d、2d、1m等，分别表示1天、2天、1个月。
- ❑ prefix_length: 此参数指定差分词条的公共前缀长度，默认值为0。
- ❑ max_expansions: 此参数指定查询可被扩展到的最大词条数，默认值是无限制。

参数应该封装在查询针对的字段名称里。所以如果想修改前面的查询，并添加额外的参数，查询将如下所示：

```

{
  "query" : {
    "fuzzy" : {
      "title" : {
        "value" : "crme",
        "min_similarity" : 0.2
      }
    }
  }
}

```


3.3.14 通配符查询

通配符查询允许我们在查询值中使用*和?等通配符。此外，通配符查询跟词条查询在内容方面非常类似。可以发送一下查询，来匹配所有包含cr?me词条的文档，这里?表示任意字符：

```
{
  "query" : {
    "wildcard" : {
      "title" : "cr?me"
    }
  }
}
```

这将匹配title字段中包含与cr?me匹配的词条的所有文档。然后，你还可以在通配符查询中包含加权属性；它将影响每个与给定值匹配的词条的重要性。如果要改变之前的查询，给它一个20.0的加权，可以发出以下查询：

```
{
  "query" : {
    "wildcard" : {
      "title" : {
        "value" : "cr?me",
        "boost" : 20.0
      }
    }
  }
}
```



请注意，通配符查询不太注重性能，在可能时应尽量避免，特别是要避免前缀通配符（以通配符开始的词条）。此外，请注意Elasticsearch会重写通配符查询，因此Elasticsearch允许通过一个额外的参数控制重写方法。有关此过程的详细信息，请参阅3.2节。

3.3.15 more_like_this 查询

more_like_this查询让我们能够得到与提供的文本类似的文档。Elasticsearch支持几个参数来定义more_like_this查询如何工作，如下所示。

- ❑ **fields**: 此参数定义应该执行查询的字段数组，默认值是_all字段。
- ❑ **like_text**: 这是一个必需的参数，包含用来跟文档比较的文本。
- ❑ **percent_terms_to_match**: 此参数定义了文档需要有多少百分比的词条与查询匹配才能认为是类似的，默认值为0.3，意思是30%。
- ❑ **min_term_freq**: 此参数定义了文档中词条的最低词频，低于此频率的词条将被忽略，默认值为2。

- ❑ `max_query_terms`: 此参数指定生成的查询中能包括的最大查询词条数, 默认值为25。值越大, 精度越大, 但性能也越低。
- ❑ `stop_words`: 此参数定义了一个单词的数组, 当比较文档和查询时, 这些单词将被忽略, 默认值为空数组。
- ❑ `min_doc_freq`: 此参数定义了包含某词条的文档的最小数目, 低于此数目时, 该词条将被忽略, 默认值为5, 意味着一个词条至少应该出现在5个文档中, 才不会被忽略。
- ❑ `max_doc_freq`: 此参数定义了包含某词条的文档的最大数目, 高于此数目时, 该词条将被忽略, 默认值为无限制。
- ❑ `min_word_len`: 此参数定义了单词的最小长度, 低于此长度的单词将被忽略, 默认值为0。
- ❑ `max_word_len`: 此参数定义了单词的最大长度, 高于此长度的单词将被忽略, 默认值为无限制。
- ❑ `boost_terms`: 此参数定义了用于每个词条的加权值, 默认值为1。
- ❑ `boost`: 此参数定义了用于查询的加权值, 默认值为1。
- ❑ `analyzer`: 此参数指定了针对我们提供的文本的分析器名称。

`more_like_this`查询的一个示例如下所示:

```
{
  "query" : {
    "more_like_this" : {
      "fields" : [ "title", "otitle" ],
      "like_text" : "crime and punishment",
      "min_term_freq" : 1,
      "min_doc_freq" : 1
    }
  }
}
```

3.3.16 `more_like_this_field` 查询

`more_like_this_field`查询与`more_like_this`查询类似, 但它只能针对单个字段。正因为如此, 它不支持多字段属性。我们把查询参数封装到要查询的字段名字中, 而不是指定`fields`参数。所以, 一个查询`title`字段的示例查询如下:

```
{
  "query" : {
    "more_like_this_field" : {
      "title" : {
        "like_text" : "crime and punishment",
        "min_term_freq" : 1,
        "min_doc_freq" : 1
      }
    }
  }
}
```

`more_like_this` 查询中的所有其他参数都可以同样的方式作用于 `more_like_this_filed` 查询。

3.3.17 范围查询

范围查询使我们能够找到在某一字段值在某个范围里的文档，字段可以是数值型，也可以是基于字符串的（将映射到一个不同的 Apache Lucene 查询）。范围查询只能针对单个字段，查询参数应封装在字段名称中。范围查询支持以下参数。

- ❑ `gte`：范围查询将匹配字段值大于或等于此参数值的文档。
- ❑ `gt`：范围查询将匹配字段值大于此参数值的文档。
- ❑ `lte`：范围查询将匹配字段值小于或等于此参数值的文档。
- ❑ `lt`：范围查询将匹配字段值小于此参数值的文档。

因此，举例来说，要找到 `year` 字段从 1700 到 1900 的所有图书，可以运行以下查询：

```
{
  "query" : {
    "range" : {
      "year" : {
        "gte" : 1700,
        "lte" : 1900
      }
    }
  }
}
```

3.3.18 最大分查询

最大分查询非常有用，因为它会生成一个由所有子查询返回的文档组成的并集并将它返回。这个查询好的一面是，我们可以控制较低得分的子查询对文档最后得分的影响。

文档的最后得分是这样计算的：最高分数的子查询的得分之和，加上其余子查询的得分之和乘以 `tie_breaker` 参数的值。所以，可以通过 `tie_breaker` 参数来控制较低得分的查询对最后得分的影响。把 `tie_breaker` 设为 1.0，得到确切的总和；把 `tie_breaker` 设为 0.1，结果，除最高得分的查询外，只有所有查询总得分的 10% 被加到最后得分里。

一个最大分查询的示例如下所示：

```
{
  "query" : {
    "dismax" : {
      "tie_breaker" : 0.99,
      "boost" : 10.0,
      "queries" : [
        {
```

```

        "match" : {
          "title" : "crime"
        }
      },
      {
        "match" : {
          "author" : "fyodor"
        }
      }
    ]
  }
}

```

可以看到，我们在查询中包含了`tie_breaker`和`boost`参数。此外，在`queries`参数中指定了一组查询，这些查询将执行并产生结果文档的并集。

3

3.3.19 正则表达式查询

通过正则表达式查询，可以使用正则表达式来查询文本。请记住，此类查询的性能取决于所选的正则表达式。如果我们的正则表达式匹配许多词条，查询将很慢。一般规则是，正则表达式匹配的词条数越高，查询越慢。

正则表达式查询示例如下所示：

```

{
  "query" : {
    "regexp" : {
      "title" : {
        "value" : "cr.m[ae]",
        "boost" : 10.0
      }
    }
  }
}

```

上述查询将被Elasticsearch重写成若干个词条查询，根据索引中匹配给定正则表达式的内容。查询中加权参数指定了生成的查询将使用的加权值。



完整的Elasticsearch支持的正则表达式语法可以在这里找到：<http://www.elasticsearch.org/guide/en/elasticsearch/reference/current/query-dsl-regexp-query.html#egexp-syntax>。

3.4 复合查询

3.3节讨论了Elasticsearch公开的最简单查询。然而，Elasticsearch提供的不只如此。顾名思义，

复合查询就是支持可以把多个查询连接起来，或者改变其他查询的行为。你可能好奇自己是否需要这样的功能。用一个简单的练习来确定：结合一个简单的词条查询和一个短语查询，得到更好的搜索结果。

3.4.1 布尔查询

可以通过布尔查询来封装无限数量的查询，并通过下面描述的节点之一使用一个逻辑值来连接它们。

- ❑ **should**: 被它封装的布尔查询可能被匹配，也可能不被匹配。被匹配的**should**节点数目由**minimum_should_match**参数控制。
- ❑ **must**: 被它封装的布尔查询必须被匹配，文档才会返回。
- ❑ **must_not**: 被它封装的布尔查询必须不被匹配，文档才会返回。

上述每个节点都可以在单个布尔查询中出现多次。这允许建立非常复杂的查询，有多个嵌套级别（在一个布尔查询中包含另一个布尔查询）。记住，结果文档的得分将由文档匹配的所有封装的查询得分总和计算得到。

除了上述部分以外，还可以在查询主体中添加以下参数控制其行为。

- ❑ **boost**: 此参数指定了查询使用的加权值，默认为1.0。加权值越高，匹配文档的得分越高。
- ❑ **minimum_should_match**: 此参数的值描述了文档被视为匹配时，应该匹配的**should**子句的最少数目。举例来说，它可以是个整数值，比如2，也可以是个百分比，比如75%。更多有关信息，参见 <http://www.elasticsearch.org/guide/en/elasticsearch/reference/current/query-dsl-minimum-should-match.html>。
- ❑ **disable_coord**: 此参数的默认值为false，允许启用或禁用分数因子的计算，该计算是基于文档包含的所有查询词条。如果得分不必太精确，但要查询快点，那么应该将它设置为true。

假设我们想要找到所有这样的文档：在**title**字段中含有**crime**词条，并且**year**字段可以在也可以不在1900~2000的范围里，在**otitle**字段中不可以包含**nothing**词条。用布尔查询的话，类似于下面的代码：

```
{
  "query" : {
    "bool" : {
      "must" : {
        "term" : {
          "title" : "crime"
        }
      },
      "should" : {
```

```

    "range" : {
      "year" : {
        "from" : 1900,
        "to" : 2000
      }
    },
    "must_not" : {
      "term" : {
        "otitle" : "nothing"
      }
    }
  }
}

```



注意: must, should和must_not节点可以包含单一查询,也可以包含多个查询。

3

3.4.2 加权查询

加权查询封装了两个查询,并且降低其中一个查询返回文档的得分。加权查询中有三个节点需要定义: positive部分,包含所返回文档得分不会被改变的查询;negative部分,返回的文档得分将被降低;negative_boost部分,包含用来降低negative部分查询得分的加权值。

加权查询的优点是, positive部分和negative部分包含的查询结果都会出现在搜索结果中,而某些查询的得分将被降低。如果使用布尔查询的must_not节点,将得不到这样的结果。

假设我们想要一个简单的词条查询,查询title字段中含有crime词条,希望这样的文档得分不被改变,同时要year字段在1800~1900内的文档,但这样文档的得分要有一个0.5的加权。这样的查询如下所示:

```

{
  "query" : {
    "boosting" : {
      "positive" : {
        "term" : {
          "title" : "crime"
        }
      },
      "negative" : {
        "range" : {
          "year" : {
            "from" : 1800,
            "to" : 1900
          }
        }
      }
    }
  },
}

```

```

        "negative_boost" : 0.5
    }
}

```

3.4.3 constant_score 查询

constant_score查询封装了另一个查询（或过滤），并为每一个所封装查询（或过滤）返回的文档返回一个常量得分。它允许我们严格控制与一个查询或过滤匹配的文档得分。如果希望title字段包含crime词条的所有文档的得分为2.0，可以发出以下查询：

```

{
  "query" : {
    "constant_score" : {
      "query" : {
        "term" : {
          "title" : "crime"
        }
      },
      "boost" : 2.0
    }
  }
}

```

3.4.4 索引查询

当针对多个索引执行查询时，索引查询很有用。可以通过indices属性提供一个索引的数组以及两个查询，一个通过query属性指定，将执行在指定的索引列表上；另一个通过no_match_query属性指定，将执行在其他所有索引上。假设我们有一个别名：books，它持有两个索引：library和users，我们希望使用别名；然而，我们希望在那些索引上执行不同的查询，为此，发送以下查询：

```

{
  "query" : {
    "indices" : {
      "indices" : [ "library" ],
      "query" : {
        "term" : {
          "title" : "crime"
        }
      }
    },
    "no_match_query" : {
      "term" : {
        "user" : "crime"
      }
    }
  }
}

```

上述查询中，`query`属性中的查询将执行在`library`索引上，`no_match_query`属性中的查询将执行在集群中其他所有索引上。

`no_match_query`属性也可以是个字符串值，而不是一个查询。这个字符串值可以是`all`或者`none`，默认是`all`。设置为`all`，索引中不匹配的所有文档都会返回；设置为`none`，索引中不匹配的文档将不会返回。



Elasticsearch公开的一些查询，如`custom_score`查询、`custom_boost_factor`查询和`custom_filters_scores`查询，已经被`function_score`查询取代，5.4.3节将描述。我们决定省略这些查询的描述，因为它们在未来版本中可能会被删除。

3

3.5 查询结果的过滤

本书已经介绍了如何使用不同的条件和查询来构建查询并搜索数据。我们还熟知了评分（参见1.1.3节），它告诉我们在给定的查询中，哪些文档更重要以及查询文本如何影响排序。然而，有时我们可能要在不影响最后分数的情况下，选择索引中的某个子集，这就要使用过滤器（当然不是唯一的原因）。

老实说，应该尽可能使用过滤器。过滤器不影响评分，而得分计算让搜索变得复杂，而且需要CPU资源。另一方面，过滤是一种相对简单的操作。由于过滤应用在整个索引的内容上，过滤的结果独立于找到的文档，也独立于文档之间的关系。过滤器很容易被缓存，从而进一步提高过滤查询的整体性能。

以下有关过滤器的章节中，我们使用`post_filter`参数保持例子尽可能地简单。然而，请记住，如果可能，应该总是使用`filtered`查询，而不是`post_filter`，因为`filtered`执行起来更快。

3.5.1 使用过滤器

在任何搜索中使用过滤器，只需在`query`节点相同级别上添加一个`filter`节点。如果你只想要过滤器，也可以完全省略`query`节点。来看一个示例查询，在`title`字段搜索`Catch-22`并向其添加过滤器，如下所示：

```
{
  "query" : {
    "match" : { "title" : "Catch-22" }
  },
  "post_filter" : {
    "term" : { "year" : 1961 }
  }
}
```


它返回给定title的所有文档，但结果缩小到仅在1961年出版的书。还有一种在查询中包含过滤器的方法：使用filtered查询。所以前面的查询可以重写如下：

```
{
  "query": {
    "filtered" : {
      "query" : {
        "match" : { "title" : "Catch-22" }
      },
      "filter" : {
        "term" : { "year" : 1961 }
      }
    }
  }
}
```

如果发送curl -XGET localhost:9200/library/book/_search?pretty -d @query.json命令来执行两个查询，你将看到它们的响应完全一样（可能除了响应时间）：

```
{
  "took" : 1,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 1,
    "max_score" : 0.2712221,
    "hits" : [ {
      "_index" : "library",
      "_type" : "book",
      "_id" : "2",
      "_score" : 0.2712221, "_source" : { "title": "Catch-
        22","author": "Joseph Heller","year": 1961,
        "characters": ["John Yossarian", "Captain Aardvark",
        "Chaplain Tappman", "Colonel Cathcart",
        "Doctor Daneeka"],"tags": ["novel"],
        "copies": 6, "available" : false}
    } ]
  }
}
```

这表明两种形式是等效的。其实不对，因为它们应用过滤和搜索的顺序是不同的。在第一种情况下，过滤器应用到查询所发现的所有文档上。第二种情况下，过滤发生在在运行查询之前，性能更好。如前所述，过滤器很快，所以filtered查询效率更高。6.2节将讨论这些内容。

3.5.2 过滤器类型

我们现在已经知道如何使用过滤器，也知道两种过滤方法的区别。现在看看Elasticsearch提

供的过滤器类型。

1. 范围过滤器

范围过滤器可以用来限制只搜索那些字段值在给定边界之间的文档。例如，为了创建一个过滤器来过滤只在1930~1990年之间出版的图书，在查询中加入以下部分：

```
{
  "post_filter" : {
    "range" : {
      "year" : {
        "gte": 1930,
        "lte": 1990
      }
    }
  }
}
```

使用`gte`和`lte`，表明字段的左右边界是包含的。如果想排除左右边界，可以使用`gt`和`lt`参数。如果想要从1930年（含1930）到1990年（不含）的文档，构造下列过滤器：

```
{
  "post_filter" : {
    "range" : {
      "year" : {
        "gte": 1930,
        "lt": 1990
      }
    }
  }
}
```

总结如下。

- `gt`: 大于。
- `lt`: 小于。
- `gte`: 大于或等于。
- `lte`: 小于或等于。

你也可以使用`execution`参数。这是一个对引擎如何执行过滤器的提示，可用值有`fielddata`和`index`。一般的经验是：在范围内有很多值时，`fielddata`能提高性能（以及内存使用）；范围内的值较少时，`index`应该更好。

还有个过滤器的变种：`numeric_filter`。这是一个特别设计的版本，用来过滤数值类型的值。此过滤器更快，但有额外的内存使用，因为Elasticsearch需要加载过滤字段的值。请注意，有时即使不使用范围过滤器，这些值也被加载。这种情况发生在我们使用相同字段做切面或排序时，没有理由不使用此过滤器。

2. exists过滤器

exists过滤器非常简单。它过滤掉给定字段上没有值的文档。比如，考虑下面的代码：

```
{
  "post_filter" : {
    "exists" : { "field": "year" }
  }
}
```

上述过滤器将只返回year字段有值的文档。

3. missing过滤器

missing过滤器跟exists过滤器相反，它过滤掉给定字段上有值的文档。然而，它还有一些额外的功能。除了选择指定字段缺失的文档，可以指定Elasticsearch对空字段的定义。这有助于在输入数据中包含null、EMPTY、not-defined等词条的情况。我们修改先前的例子，找没有定义year字段、或者year字段为0的那些文档。修改过的过滤器将如下所示：

```
{
  "post_filter" : {
    "missing" : {
      "field": "year",
      "null_value": 0,
      "existence": true
    }
  }
}
```

在前面的示例中，有两个额外参数。existence参数告诉Elasticsearch应该检查指定字段上存在值的文档，null_value参数定义了应该被视为空的额外值。如果你没有定义null_value，existence将被设为默认值；所以，在这个例子中可以省略existence。

4. 脚本过滤器

有时，我们想要通过计算值来过滤文档。一个例子是：可以过滤掉所有发表在一个世纪以前的书。使用脚本过滤器来实现，如下所示：

```
{
  "post_filter" : {
    "script" : {
      "script" : "now - doc['year'].value > 100",
      "params" : {
        "now" : 2012
      }
    }
  }
}
```

可以看到，我们使用了一个简单的脚本来计算值，并从中过滤数据。5.2节将讨论Elasticsearch

的更多脚本功能。

5. 类型过滤器

类型过滤器专门用来限制文档的类型。当查询运行在多个索引上，或单个索引但有很多类型时，可以使用这个过滤器。例如，想限制返回文档的类型为book，使用下列过滤器：

```
{
  "post_filter" : {
    "type": {
      "value" : "book"
    }
  }
}
```

6. 限定过滤器

限定过滤器限定单个分片返回的文档数目。不要把它跟size参数混在一起。作为例子，我们看看下面的过滤器：

```
{
  "post_filter" : {
    "limit" : {
      "value" : 1
    }
  }
}
```

当我们对分片数量使用默认设置时，上述过滤器返回5个文档。因为在Elasticsearch中，索引默认分为5个分片。查询分别运行在各个分片上，而每个分片最多返回1个文档。

7. 标识符过滤器

需要过滤成若干具体的文档时，可以使用标识符过滤器。例如，要排除标识符等于1的文档，可使用类似下面的代码：

```
{
  "post_filter": {
    "ids" : {
      "type": ["book"],
      "values": [1]
    }
  }
}
```

注意，type参数不是必需的。然而，当我们在几个索引中搜索，又想指定一个感兴趣的类型时，它还是有用的。

8. 如果还不够

目前为止，我们讨论了几个在Elasticsearch使用过滤器的例子。然而，这只是冰山一角。你

可以在过滤器中封装几乎所有查询。例如，让我们来看看下面的查询：

```
{
  "query" : {
    "multi_match" : {
      "query" : "novel erich",
      "fields" : [ "tags", "author" ]
    }
  }
}
```

上述示例显示了一个我们熟知的简单multi_match查询。可以用过滤器按如下方式重写此查询：

```
{
  "post_filter" : {
    "query" : {
      "multi_match" : {
        "query" : "novel erich",
        "fields" : [ "tags", "author" ]
      }
    }
  }
}
```

当然，唯一的区别是得分。过滤器返回的每个文档得分都是1.0。注意，Elasticsearch有几个专用过滤器是这样工作的（比如与词条查询对应的词条过滤器）。所以，你不必总是使用封装查询语法。事实上，你应该尽量使用专用的过滤器版本。

Elasticsearch支持下列专用过滤器：

- ❑ bool过滤器；
- ❑ geo_shape过滤器；
- ❑ has_child过滤器；
- ❑ has_parent过滤器；
- ❑ ids过滤器；
- ❑ indices过滤器；
- ❑ match_all过滤器；
- ❑ nested过滤器；
- ❑ prefix过滤器；
- ❑ range过滤器；
- ❑ regexp过滤器；
- ❑ term过滤器；
- ❑ terms过滤器。

9. 组合过滤器

现在，是时候把一些过滤器组合在一起了。第一个选择是使用bool过滤器，它能够在3.4.1节所述原理的基础上组合过滤器。第二种选择是使用and、or和not过滤器。and过滤器使用一个过滤器数组并返回与该数组中的所有过滤器匹配的文档。or过滤器也使用一个数组，但它返回与数组中任何一个过滤器匹配的文档。至于not过滤器，则返回与所封装的过滤器不匹配的文档。当然，所有这些过滤器可以嵌套使用，如以下示例所示：

```
{
  "post_filter": {
    "not": {
      "and": [
        {
          "term": {
            "title": "Catch-22"
          }
        },
        {
          "or": [
            {
              "range": {
                "year": {
                  "gte": 1930,
                  "lte": 1990
                }
              }
            },
            {
              "term": {
                "available": true
              }
            }
          ]
        }
      ]
    }
  }
}
```

● 关于bool过滤器

当然，你可能会问bool过滤器与and、or、not过滤器之间的区别。首先，这些过滤器可以互换使用。当然，从返回的结果角度是对的，但从性能角度则不然。

我们可以看到Elasticsearch在内部为每个过滤器都建立了一个叫bitset的结构，它保存着索引中的后续文档是否跟过滤器匹配的信息。bitset很容易被缓存并在使用相同过滤器的所有查询中重用。这是Elasticsearch的一种简便、高效的方案。总之，尽可能使用bool过滤器。可惜，现实生活不总是这么简单。某些类型的过滤器没有能力直接创建bitset。在这罕见的情形下，bool筛选

器将更低效。你已经知道有两个这样的过滤器：数值型的范围过滤器和脚本过滤器。第三个是使用地理坐标的整组过滤器，6.2.10将对此进行讨论。

10. 命名过滤器

设置过滤器可能很复杂，所以有时候，如果知道哪些过滤器用来决定查询应该返回哪些文档，无疑是很有帮助的。幸好，可以给每个过滤器命名，名字将随着匹配文档返回。来看看它是如何工作的。以下查询将返回所有可用并且标签为novel，或者来自19世纪的书：

```
{
  "query": {
    "filtered" : {
      "query": { "match_all" : {} },
      "filter" : {
        "or" : [
          { "and" : [
            { "term": { "available" : true } },
            { "term": { "tags" : "novel" } }
          ] },
          { "range" : { "year" : { "gte": 1800, "lte" : 1899 } } }
        ]
      }
    }
  }
}
```

我们使用查询的filtered版本，因为它是Elasticsearch中唯一可以添加过滤器信息的版本。我们重写该查询以添加每个过滤器的名字，如下所示：

```
{
  "query": {
    "filtered" : {
      "query": { "match_all" : {} },
      "filter" : {
        "or" : {
          "filters" : [
            {
              "and" : {
                "filters" : [
                  {
                    "term": {
                      "available" : true,
                      "_name" : "avail"
                    }
                  },
                  {
                    "term": {
                      "tags" : "novel",
                      "_name" : "tag"
                    }
                  }
                ]
              }
            }
          ]
        }
      }
    }
  },
}
```



```

    "Punishment", "otitle": "Преступление и наказание",
    "author": "Fyodor Dostoevsky", "year": 1886,
    "characters": ["Raskolnikov", "Sofia Semyonovna
    Marmeladova"], "tags": [], "copies": 0, "available" : true},
    "matched_queries" : [ "or", "year", "avail" ]
  } ]
}
}

```

你可以看到,除了标准信息以外,每个文档包含一个表,包含与该特定文档匹配的过滤器名称。



记住在大多数情况下, `filtered` 查询比 `post_filter` 更快。所以在可能的情况下,尽可能使用 `filtered` 查询。

3.5.3 过滤器的缓存

关于过滤器最后要提到的是缓存。缓存加速了使用过滤器的查询,代价是第一次执行过滤器时的内存成本和查询时间。因此,缓存的最佳选择是那些可以重复使用的过滤器,例如,经常会使用并包括参数值的那些。

缓存可以在 `and`、`bool`、`or` 过滤器上打开(但通常,缓存它们所附的过滤器才是更好的主意)。在这种情况下,所需的语法与前面命名过滤器所描述的一样,如下所示:

```

{
  "post_filter" : {
    "script" : {
      "_cache": true,
      "script" : "now - doc['year'].value > 100",
      "params" : {
        "now" : 2012
      }
    }
  }
}

```

有些过滤器不支持 `_cache` 参数,因为它们的结果总是被缓存。默认情况下是下面这些:

- ☐ `exists`
- ☐ `missing`
- ☐ `range`
- ☐ `term`
- ☐ `terms`

可通过关闭缓存来修改此行为,代码如下所示:

```

{
  "post_filter": {

```

```

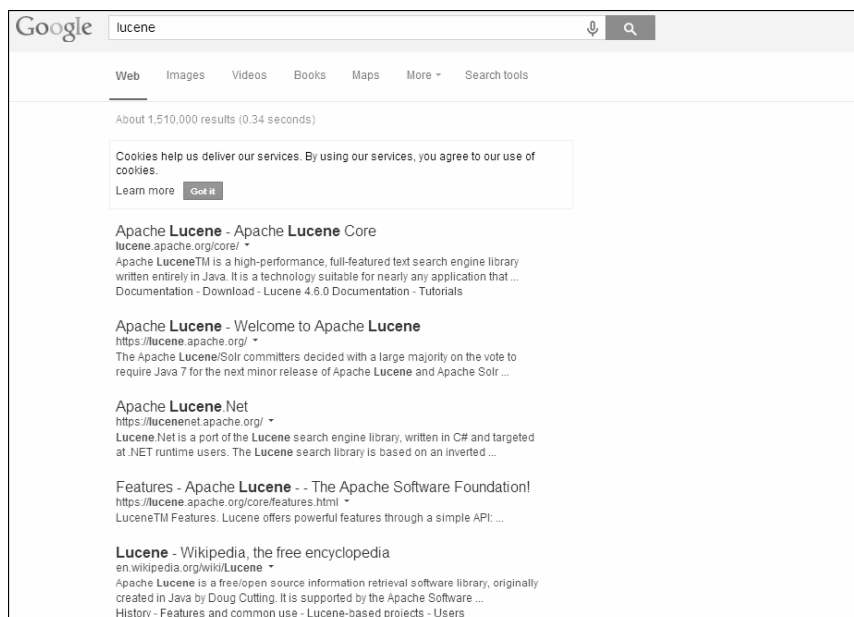
    "term": {
      "_cache": false,
      "year": 1961
    }
  }
}

```

对于ids过滤器、match_all过滤器和limit过滤器，缓存是无效的。

3.6 高亮显示

你可能听说过高亮显示，即使不熟悉这个名字，也可能在你访问过的网页中看过高亮显示的结果。高亮显示是在结果文档中显示查询中的哪个或哪些单词被匹配的过程。例如，在谷歌搜索lucene这个词，我们会看到它在结果列表中以粗体显示，如下面的截图所示：



本章将展示如何使用Elasticsearch高亮能力来加强我们的应用，使结果高亮显示。

3.6.1 高亮显示入门

要展示高亮显示是如何工作的，最好的办法是创建一个查询，看看Elasticsearch返回的结果。所以假设我们想高亮显示在title字段中匹配的单词，以改善用户的搜索体验。我们还是来搜索crime一词，为了得到具有高亮的结果，发送如下查询：

```
{
  "query" : {
    "term" : {
      "title" : "crime"
    }
  },
  "highlight" : {
    "fields" : {
      "title" : {}
    }
  }
}
```

这个查询的响应看起来如下所示:

```
{
  "took" : 2,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 1,
    "max_score" : 0.19178301,
    "hits" : [ {
      "_index" : "library",
      "_type" : "book",
      "_id" : "4",
      "_score" : 0.19178301, "_source" : { "title": "Crime and
        Punishment", "otitle": "Преступление и наказание",
        "author": "Fyodor Dostoevsky", "year": 1886,

        "characters": ["Raskolnikov", "Sofia Semyonovna
        Marmeladova"], "tags": [], "copies": 0, "available" : true},
      "highlight" : {
        "title" : [ "<em>Crime</em> and Punishment" ]
      }
    } ]
  }
}
```

可以看到,除了从Elasticsearch得到的标准信息,有一个新的名为highlight的部分。Elasticsearch使用这个HTML标签来包含高亮部分。这是Elasticsearch的默认行为,我们将学习如何去改变它。

3.6.2 字段配置

为了执行高亮显示,需要呈现字段的原始内容:我们必须把这些用来高亮显示的字段设为stored,或者把这些字段包含在_source字段中。

3.6.3 深入底层

Elasticsearch在底层使用Apache Lucene，而高亮显示是Lucene库的功能之一。Lucene提供了三种类型的高亮实现：标准类型，就是我们刚刚使用的；第二种叫FastVectorHighlighter，它需要词向量和位置才能工作；第三种叫PostingsHighlighter，本章的最后将讨论它。Elasticsearch自动选择正确的高亮实现方式：如果字段的配置中，term_vector属性设成了with_positions_offsets，则将使用FastVectorHighlighter。

然而，必须记住，使用词向量将导致索引变大，但高亮显示的执行需要更少的时间。此外，对于存储了大量数据的字段来说，推荐使用FastVectorHighlighter

3.6.4 配置 HTML 标签

3

我们已经提到，改变默认的HTML标签成我们想用的标签是可能的。例如，假设要使用标准的HTML标签来高亮。为此，我们应分别设置pre_tags和post_tags这些属性（它们是数组）为和。既然提到的两个属性为数组，可以包含多个标签，Elasticsearch将使用定义每个标签来高亮显示不同的单词。所以，我们的示例查询如下所示：

```
{
  "query" : {
    "term" : {
      "title" : "crime"
    }
  },
  "highlight" : {
    "pre_tags" : [ "<b>" ],
    "post_tags" : [ "</b>" ],
    "fields" : {
      "title" : {}
    }
  }
}
```

对上述查询，Elasticsearch返回的结果如下所示：

```
{
  "took" : 2,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 1,
    "max_score" : 0.19178301,
    "hits" : [ {
```

```

    "_index" : "library",
    "_type" : "book",
    "_id" : "4",
    "_score" : 0.19178301, "_source" : { "title": "Crime and
    Punishment", "otitle": "Преступление и наказание",
    "author": "Fyodor Dostoevsky", "year": 1886,
    "characters": ["Raskolnikov", "Sofia Semyonovna
    Marmeladova"], "tags": [], "copies": 0, "available" : true},
    "highlight" : {
      "title" : [ "<b>Crime</b> and Punishment" ]
    }
  }
}
}

```

可以看到，title字段中的Crime，被我们选择的标签包围。

3.6.5 控制高亮片段

Elasticsearch允许我们控制高亮片段的数量以及它们的大小，为此公开了两个属性供使用。第一个，number_of_fragments，定义Elasticsearch返回的片段数量，默认值为5。把这个属性设置为0，将导致整个字段被返回，这对短字段来说是很便利的；然而，对长字段来说代价较大。第二个属性是fragment_size，用来指定高亮片段的最大字符长度，默认值是100。

3.6.6 全局设置与局部设置

前面讨论的高亮显示的属性，可以设在全局范围，也可以设在每个字段上。全局设置将用在没有做局部设置的所有字段上，它跟高亮对象的fields节点设在同一个级别上，如下所示：

```

{
  "query" : {
    "term" : {
      "title" : "crime"
    }
  },
  "highlight" : {
    "pre_tags" : [ "<b>" ],
    "post_tags" : [ "</b>" ],
    "fields" : {
      "title" : {}
    }
  }
}

```

也可以为每个字段设置这些属性。比如，除了title字段，我们想对其他字段保持默认行为，可以使用下面的代码：

```

{
  "query" : {
    "term" : {

```

```

        "title" : "crime"
      }
    },
    "highlight" : {
      "fields" : {
        "title" : {
          "pre_tags" : [ "<b>" ], "post_tags" : [ "</b>" ]
        }
      }
    }
  }
}

```

可以看到，我们把它放在指定title字段行为的空JSON对象里面，而不是与fields同一水平线上的部分中。当然，每个字段可配置为不同的属性。

3.6.7 需要匹配

3

有时，尤其是在使用多个高亮字段时，只需要显示跟查询匹配的字段。为了触发此行为，要把require_field_match属性设为true。把该属性设为false将导致所有词条都高亮显示，即使是在跟查询不匹配的字段中。

为了看它是如何工作的，创建一个新的索引users，并创建一个文档到这个索引中，发送如下命令：

```

curl -XPUT 'http://localhost:9200/users/user/1' -d '{
  "name" : "Test user",
  "description" : "Test document"
}'

```

现在，假想高亮显示name和description字段，查询如下所示：

```

{
  "query" : {
    "term" : {
      "name" : "test"
    }
  },
  "highlight" : {
    "fields" : {
      "name" : { "pre_tags" : [ "<b>" ], "post_tags" : [ "</b>" ]
    },
    "description" : { "pre_tags" : [ "<b>" ], "post_tags" : [
      "</b>" ] }
  }
}

```

上述查询的结果如下所示：

```

{
  "took" : 3,
  "timed_out" : false,

```

```

    "_shards" : {
      "total" : 5,
      "successful" : 5,
      "failed" : 0
    },
    "hits" : {
      "total" : 1,
      "max_score" : 0.19178301,
      "hits" : [ {
        "_index" : "users",
        "_type" : "user",
        "_id" : "1",
        "_score" : 0.19178301, "_source" : {"name" : "Test
          user","description" : "Test document"},
        "highlight" : {
          "description" : [ "<b>Test</b> document" ],
          "name" : [ "<b>Test</b> user" ]
        }
      } ]
    }
  }
}

```

注意，即使我们只匹配name字段，得到的结果却高亮显示了上述两个字段。在大多数情况下，我们不希望这样。所以现在修改查询，使用require_field_match属性如下：

```

{
  "query" : {
    "term" : {
      "name" : "test"
    }
  },
  "highlight" : {
    "require_field_match" : "true",
    "fields" : {
      "name" : { "pre_tags" : [ "<b>" ], "post_tags" : [ "<b>" ]
    },
    "description" : { "pre_tags" : [ "<b>" ], "post_tags" : [
      "</b>" ] }
  }
}

```

看看修改过的查询得到的结果，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" : "users",
      "_type" : "user",
      "_id" : "1",
      "_score" : 0.19178301, "_source" : { "name" : "Test
        user", "description" : "Test document" },
      "highlight" : {
        "name" : [ "<b>Test</b> user" ]
      }
    } ]
  }
}

```

可以看到，Elasticsearch只在高亮中返回匹配字段，在我们的例子中是name字段。

3

3.6.8 信息高亮器

现在来讨论一下Elasticsearch中的第三种高亮显示类型。它在Elasticsearch 0.90.6中被加入，与前两种类型略有不同，我们通过下面的例子来看看这些区别。当字段定义中的index_options属性设成offsets时，自动运用PostingsHighlighter。所以，为了演示PostingsHighlighter是如何工作的，我们将用正确的映射创建一个简单的索引。为此，执行以下命令：

```

curl -XPUT 'localhost:9200/hl_test'
curl -XPOST 'localhost:9200/hl_test/doc/_mapping' -d '{
  "doc" : {
    "properties" : {
      "contents" : {
        "type" : "string",
        "fields" : {
          "ps" : { "type" : "string", "index_options" : "offsets" }
        }
      }
    }
  }
},

```



请记住，与FastVectorHighlighter类似，PostingsHighlighter需要的offset会导致索引大小的增加，但这种增加比使用词向量时更小。此外，索引offset比索引词向量更快，且PostingsHighlighter的查询性能更好。

如果一切顺利，我们将有一个新的索引和映射。映射定义了两个字段：一个名叫contents，另一个叫contents.ps。在这个例子中，使用index_options属性打开了偏移。这意味着Elasticsearch将对contents contents.ps字段使用信息高亮器。

为了看其中的差别，我们索引一个文档，该文档包含维基百字段使用标准的高亮类型，而对

科中描述伯明翰历史的片段。为此，执行以下命令：

```
curl -XPUT localhost:9200/hl_test/doc/1 -d '{
  "contents" : "Birmingham\'s early history is that of a remote and
    marginal area. The main centers of population, power and wealth
    in the pre-industrial English Midlands lay in the fertile and
    accessible river valleys of the Trent, the Severn and the Avon.
    The area of modern Birmingham lay in between, on the upland
    Birmingham Plateau and within the densely wooded and sparsely
    populated Forest of Arden."
}'
```

最后一步是使用两个高亮类型来发送查询请求。为此，可以使用如下命令来发送单个请求：

```
curl 'localhost:9200/hl_test/_search?pretty' -d '{
  "query": {
    "term": {
      "contents": "modern"
    }
  },
  "highlight": {
    "fields": {
      "contents": {},
      "contents.ps" : {}
    }
  }
}'
```

如果一切顺利，可以在响应中找到如下片段：

```
"highlight" : {
  "contents" : [ " valleys of the Trent, the Severn and the
    Avon. The area of <em>modern</em> Birmingham lay in
    between, on the upland" ],
  "contents.ps" : [ "The area of <em>modern</em> Birmingham lay
    in between, on the upland Birmingham Plateau and within the
    densely wooded and sparsely populated Forest of Arden." ]
}
```

可以看到，两种高亮实现都找到了所需要的单词，不同的是，信息高亮器返回的片段更聪明，它检测了句子的边界。

用下面的命令再来试一个查询：

```
curl 'localhost:9200/hl_test/_search?pretty' -d '{
  "query": {
    "match_phrase": {
      "contents": "centers of"
    }
  },
  "highlight": {
    "fields": {
      "contents": {},

```

```

    "contents.ps": {}
  }
}
}'

```

搜索一个特定的短语centers of。正如你预料的那样，这两种高亮实现的结果不一样。对标准高亮实现，将在响应中找到如下的短语：

```

"Birmingham's early history is that of a remote and marginal area.
The main <em>centers</em> <em>of</em> population"

```

可以清楚地看到，标准高亮实现分割了给定短语，高亮单独的词条。不是所有的centers和of词条都被高亮，只有来自这个短语的才被高亮。

另一方面，信息高亮器返回如下高亮片段：

```

"Birmingham's early history is that <em>of</em> a remote and marginal
area.",
"The main <em>centers</em> <em>of</em> population, power and wealth
in the pre-industrial English Midlands lay in the fertile and
accessible river valleys <em>of</em> the Trent, the Severn and the
Avon.",
"The area <em>of</em> modern Birmingham lay in between, on the upland
Birmingham Plateau and within the densely wooded and sparsely
populated Forest <em>of</em> Arden."

```

这是个显著的差异：信息高亮器高亮了所有跟查询词条匹配的词条，而不仅是组成短语的那些词条。

3.7 验证查询

有时，应用程序发送到Elasticsearch的查询是自动从多个条件生成的，甚至更糟，它们是通过某种向导生成，最终用户可以在其中创建复杂的查询。问题是，查询的正确与否，有时并不容易分辨。为了解决这个问题，Elasticsearch公开了验证API。

使用验证 API

验证API非常简单，我们把它发送到_validate_query端点，而不是_search端点就行了。来看看下面的查询：

```

{
  "query" : {
    "bool" : {
      "must" : {
        "term" : {
          "title" : "crime"
        }
      }
    }
  }
}

```

```
    }
  },
  "should" : {
    "range" : {
      "year" : {
        "from" : 1900,
        "to" : 2000
      }
    }
  },
  "must_not" : {
    "term" : {
      "otitle" : "nothing"
    }
  }
}
}
```

本书中已经用过这个查询，我们知道该查询一切正常。但要通过下面的命令来验证一下（已经把查询保存到query.json文件中）：

```
curl -XGET 'localhost:9200/library/_validate/query?pretty' -d
@query.json
```

查询看上去是对的，但来看看验证API怎么说。Elasticsearch返回的响应如下所示：

```
{
  "valid" : false,
  "_shards" : {
    "total" : 1,
    "successful" : 1,
    "failed" : 0
  }
}
```

看下valid属性，它被设成了false。有些地方出错了。再执行一次验证查询，这次加入explain参数，如下所示：

```
curl -XGET 'localhost:9200/library/_validate/query?pretty&explain' --
data-binary @query.json
```

这次，Elasticsearch返回的结果更加详细，如下所示：

```
{
  "valid" : false,
  "_shards" : {
    "total" : 1,
    "successful" : 1,
    "failed" : 0
  },
  "explanations" : [ {
    "index" : "library",
```

```

    "valid" : false,
    "error" : "org.elasticsearch.index.query.QueryParseException:
      [library] Failed to parse;
      org.elasticsearch.common.jackson.core.JsonParseException:
      Illegal unquoted character ((CTRL-CHAR, code 10)): has to be
      escaped using backslash to be included in name\n at [Source:
      [B@6456919f; line: 10, column: 18]"
  } ]
}

```

现在一切都清楚了，在我们的例子中，range属性的引号少了一个。



你可能想知道为什么我们在curl查询中使用--data-binary参数。此参数可在发送查询到Elasticsearch时保留换行符。这意味着行列数都将是完好的，这样更容易发现错误。在其他情况下，-d参数更为方便，因为它更短。

3

验证API还可以检测其他错误，例如，格式不正确的数字或其他映射相关的问题。可惜，由于我们的应用程因为缺乏错误信息中的结构，不是很容易发现问题。

3.8 数据排序

我们现在知道了如何构建查询和过滤结果，也知道搜索类型以及为什么它们重要。可以把这些查询发送到Elasticsearch，并分析返回的数据分析。现在，这个数据的组织顺序是由得分决定的。在大多数情况下，这正是我们想要的。搜索操作首先应该给我们最相关的文档。然而，如果想让查询更像一个数据库，或想设置一个更复杂的算法对数据排序，该怎么做呢？来看看Elasticsearch的排序功能可以做什么。

3.8.1 默认排序

看下面的查询，它返回至少含有一个指定单词的所有书：

```

{
  "query" : {
    "terms" : {
      "title" : [ "crime", "front", "punishment" ],
      "minimum_match" : 1
    }
  }
}

```

在底层，Elasticsearch把它当作如下查询：

```

{
  "query" : {
    "terms" : {

```

```

    "title" : [ "crime", "front", "punishment" ],
    "minimum_match" : 1
  }
},
"sort" : { "_score" : "desc" }
}

```

注意上述查询中高亮的部分，这是Elasticsearch的默认排序。更详细来说，这个片段如下所示：

```

"sort" : [
  { "_score" : "desc" }
]

```

前一节定义了结果列表中的文档应该如何排序。在这个例子中，Elasticsearch在结果列表的顶部显示最高分的文档。最简单的修改是旋转sort部分，以达到反向排序，如下所示：

```

"sort" : [
  { "_score" : "asc" }
]

```

3.8.2 选择用于排序的字段

默认排序很无聊，不是吗？所以，我们让它根据文档中的一个字段排序，如下所示：

```

"sort" : [
  { "title" : "asc" }
]

```

可惜，这并不会如预期一样工作。虽然Elasticsearch对文档做了排序，但文档顺序有些奇怪。仔细查看响应，Elasticsearch对每个文档返回了排序信息，例如，对Catch-22这本书，返回文档类似于以下代码：

```

{
  "_index": "library",
  "_type": "book",
  "_id": "2",
  "_score": null,
  "_source": {
    "title": "Catch-22",
    "author": "Joseph Heller",
    "year": 1961,
    "characters": [
      "John Yossarian",
      "Captain Aardvark",
      "Chaplain Tappman",
      "Colonel Cathcart",
      "Doctor Daneeka"
    ],
    "tags": [
      "novel"
    ]
  },
  "tags": [
    "novel"
  ]
}

```

```

    "copies": 6,
    "available": false,
    "section": 1
  },
  "sort": [
    "22"
  ]
}

```

如果你比较一下title字段和返回的排序信息，一切应该都清楚了。Elasticsearch在分析过程中把字段拆分为几个标记。因为排序是使用单个标记，Elasticsearch在产生的标记中选择一个。这种做法是最好的，因为它可以通过对这些标记按照字母顺序排序并选择了第一个。这就是为什么在排序的值中，我们只能找到一个单词而不是title字段的全部内容。在空余时间，你可以看看Elasticsearch对字符字段排序时的表现。

一般来说，对一个未经分析的字段排序是一个好主意。我们可以对具有多个值的字段排序，但在大多数情况下，没有多大意义，因此使用有限。例如，我们使用两个不同的字段，一个作为排序，另一个作为搜索，修改title字段。更改过的title字段定义可能类似于下面的代码：

```

"title" : {
  "type": "string",
  "fields": {
    "sort": { "type" : "string", "index": "not_analyzed" }
  }
}

```

在映射中修改title字段后，本章开头已经展示过，可以尝试对title.sort字段排序，看看它是否工作。为此，需要发送以下查询：

```

{
  "query" : {
    "match_all" : { }
  },
  "sort" : [
    { "title.sort" : "asc" }
  ]
}

```

现在，它正常了。正如你所看到的，我们用新的字段title.sort。已经将其设置为未经分析，因此，在索引中它是个单值字段。

在Elasticsearch响应中，每个文档包含用于排序的值的有关信息，如下所示：

```

"_index" : "library",
"_type" : "book",
"_id" : "1",
"_score" : null, "_source" : { "title": "All Quiet on the
Western Front","otitle": "Im Westen nichts Neues",
"author": "Erich Maria Remarque","year":
1929,"characters": ["Paul Bäumer", "Albert Kropp",
"Haie Westhus", "Fredrich Müller", "Stanislaus
Katzinsky", "Tjaden"],"tags": ["novel"],"copies": 1,

```

```
"available": true, "section" : 3},
"sort" : [ "All Quiet on the Western Front" ]
```

请注意，`sort`在请求和响应中，都是一个数组。这表明我们可以使用几种不同的排序。对于前一个字段值相同的文档，Elasticsearch将使用下一个数组里的元素来确定文档的顺序。所以，如果文档的`title`字段相同，将使用我们指定的下一个字段排序。

3.8.3 指定缺少字段的行为

当有些与查询匹配的文档没有我们要排序的字段时，会怎么样？默认情况下，没有给定字段的文档，如果是升序排，则出现在第一个；如果是降序排，则出现在最后一个。然而，有时这不是我们想要的。

使用数字字段排序时，可以更改Elasticsearch对缺少字段的文档的默认行为。例如以下查询：

```
{
  "query" : {
    "match_all" : { }
  },
  "sort" : [
    { "section" : { "order" : "asc", "missing" : "_last" } }
  ]
}
```

注意，查询中`sort`节点的扩展部分添加了`missing`参数。通过把`missing`参数设为`_last`，Elasticsearch将把缺乏给定字段的文档放在结果列表的底部。设置为`_first`，则会把缺乏给定字段的文档放在结果列表的顶部。值得一提的是，除了`_last`和`_first`值，Elasticsearch允许我们使用任意数字。在这种情况下，一个没有给定字段的文档将被视为该文档具有给定的值。

3.8.4 动态条件

上一节提到，Elasticsearch允许使用具有多个值的字段排序。可以使用脚本来控制排序时进行的比较，通过告诉Elasticsearch如何计算应用于排序的值来达到目的。假设要通过`tags`字段中的第一个值来排序。看看下面的示例查询：

```
{
  "query" : {
    "match_all" : { }
  },
  "sort" : {
    "_script" : {
      "script" : "doc['tags'].values.length > 0 ?
        doc['tags'].values[0] : '\u1999'",
      "type" : "string",
      "order" : "asc"
    }
  }
}
```

在上面的示例中，我们把每一个不存在的值替换成一个Unicode字符，该字符在列表中应该处于足够低的位置。此代码的主要想法是检查tags数组中是否包含至少一个元素。如果是，那么返回数组中的第一个值。如果数组为空，返回Unicode字符，该字符应该放在结果列表的底部。除了script参数，还需要指定order参数（在我们的例子中是升序），以及用于比较的type参数（我们的脚本返回的是string）。

3.8.5 排序规则和国家特有字符

如果想使用英语以外的语言，可能要面对字符顺序不正确的问题。这是因为许多语言有不同的字母顺序定义。Elasticsearch支持多种语言，但需要额外的插件来支持适当的排序规则。它很容易安装和配置，8.7节将进一步讨论。

3

3.9 查询重写

基本上，任何涉及多词条的查询，比如前缀查询和通配符查询，都使用查询重写。Elasticsearch这样做是基于性能方面的原因。重写过程把原始的、昂贵的查询修改成一组Lucene认为不太昂贵的查询。

3.9.1 重写过程示例

要说明重写过程在内部是如何进行的，最好的方式是看一个例子，看看哪些词条代替了原始的查询词条。假设在索引中有以下数据：

```
curl -XPOST 'localhost:9200/library/book/1' -d '{"title": "Solr 4 Cookbook"}'
curl -XPOST 'localhost:9200/library/book/2' -d '{"title": "Solr 3.1 Cookbook"}'
curl -XPOST 'localhost:9200/library/book/3' -d '{"title": "Mastering Elasticsearch"}'
```

我们需要找到以字母s开头的所有文档。就这么简单，对该library索引执行以下查询：

```
curl -XGET 'localhost:9200/library/_search?pretty' -d '{
  "query": {
    "prefix": {
      "title": "s",
      "rewrite": "constant_score_boolean"
    }
  }
}'
```

这里，使用一个简单的前缀查询。我们说过要找到所有在title字段中含有字母s的文档。我们还使用了rewrite属性来指定查询重写方法，但先跳过它，因为会在本节的后半部分讨论此参数的可能值。作为对上述查询的响应，得到以下输出：


```

{
  "took" : 22,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 2,
    "max_score" : 1.0,
    "hits" : [ {
      "_index" : "library",
      "_type" : "book",
      "_id" : "2",
      "_score" : 1.0, "_source" : {"title": "Solr 3.1 Cookbook"}
    }, {
      "_index" : "library",
      "_type" : "book",
      "_id" : "1",
      "_score" : 1.0, "_source" : {"title": "Solr 4 Cookbook"}
    } ]
  }
}

```

可以看到,在响应中,我们有两个文档,它们的标题字段内容以期望的字符开头。看看Lucene级别的查询,我们注意到,前缀查询被重写成类似于下面这样的查询:

```
ConstantScore(title:solr)
```

这是因为solr是唯一以字母s开头的词条。这就是查询重写的一切:找到相关的词条,把查询重写为性能更好的查询,而不是执行昂贵的查询。

3.9.2 查询重写的属性

我们已经说过,可以在任何多项词条查询(比如Elasticsearch的前缀查询和通配符查询)中使用rewrite参数来控制查询如何被改写。把rewrite参数添加到负责实际查询的JSON对象中,如下所示:

```

{
  "query" : {
    "prefix" : {
      "title" : "s",
      "rewrite" : "constant_score_boolean"
    }
  }
}

```

现在,来看看此参数的值有哪些选项。

❑ scoring_boolean: 这种重写方法把生成的每个词条翻译成布尔查询中的一个should子

句。此查询重写方法可能是CPU密集型（因为它计算并存储每个词条的得分），如果查询许多词条，可能超过布尔查询极限，也就是1024。此外，此查询会存储计算所得的分数。

- ❑ `constant_score_boolean`: 这种重写方法类似于上面描述的`scoring_boolean`重写方法，但是对CPU要求较低，因为不需要计算得分。相反，每个词条都得到一个与查询加权相等的得分，默认是1，可以通过加权属性进行设置。与`scoring_boolean`重写方法类似，该方法也可能达到布尔查询的最高限制。
- ❑ `constant_score_filter`: 就像Apache Lucene的Javadocs声明的那样，这个重写方法按顺序访问每个词条，标记该词条的所有文档，并创建一个私有过滤器来重写查询。匹配的文档都被赋予一个与查询加权相等的常量得分。当匹配词条或文档的数量很大时，此方法比`scoring_boolean`和`constant_score_boolean`快。
- ❑ `top_terms_N`: 这种重写方法把生成的每个词条翻译成布尔查询中的一个`should`子句，并保持查询计算所得的分数。然而，与`scoring_boolean`重写方法不同，它只会保留N个最高得分的词条，以免达到布尔查询的最大限制。
- ❑ `top_terms_boost_N`: 这是一种类似`top_terms_N`的重写方法。然而，与`top_terms_N`重写方法不同，分数只由加权计算而来，而非查询。



当重写属性设置为`constant_score_auto`或根本没有设置时，根据查询以及构造方式的不同，将选择使用`constant_score_filter`或`constant_score_boolean`。

结束本章查询重写部分之前，我们应该问自己最后一个问题，“什么时候使用哪种类型的重写”？这个问题的答案在很大程度上取决于我们的用例，但是总结来说，如果能忍受低精度（但性能更好），使用`top N`重写方法。如果需要高精度（但性能较低），选择布尔方法。

3.10 小结

本章介绍了Elasticsearch查询如何工作，以及如何选择要返回的数据；讨论了查询重写如何工作，有哪些搜索类型，什么是搜索偏好；展示了Elasticsearch中可用的基本查询，并使用过滤器来过滤结果。此外，还讨论了高亮显示功能，让它来高亮文档中匹配的部分。我们验证了查询。了解了复合查询，将多个查询组合在一起，最后，看到了如何根据需求配置排序。

下一章重点介绍索引，但还会涉及其他内容。我们会学到如何索引树状结构；看到如何在Elasticsearch中存储JSON对象，来索引非扁平的数据，以及如何修改一个已经创建的索引的结构。下一章还将阐述如何使用嵌套文档和主从功能来处理文档之间的关系。