**7.1请求体查询**

简单查询语句(lite)是一种有效的命令行*adhoc（点对点）*查询。但是，如果你想要善用搜索，你必须使用请求体查询(request body search)API。之所以这么称呼，是因为大多数的参数以JSON格式所容纳而非查询字符串。

请求体查询(下文简称查询)，并不仅仅用来处理查询，而且还可以高亮返回结果中的片段，并且给出帮助你的用户找寻最好结果的相关数据建议。

**空查询**

我们以最简单的 search API开始，空查询将会返回索引中所有的文档。

GET /\_search

{} <1>

* <1> 这是一个空查询数据。

同字符串查询一样，你可以查询一个，多个或\_all索引(indices)或类型(types)：

GET /index\_2014\*/type1,type2/\_search

{}

你可以使用from及size参数进行分页：

GET /\_search

{

"from": 30,

"size": 10

}

携带内容的GET请求？

任何一种语言(特别是js)的HTTP库都不允许GET请求中携带交互数据。 事实上，有些用户很惊讶GET请求中居然会允许携带交互数据。

真实情况是，<http://tools.ietf.org/html/rfc7231#page-24[RFC> 7231]， 一份规定HTTP语义及内容的RFC中并未规定GET请求中允许携带交互数据！ 所以，有些HTTP服务允许这种行为，而另一些(特别是缓存代理)，则不允许这种行为。

Elasticsearch的作者们倾向于使用GET提交查询请求，因为他们觉得这个词相比POST来说，能更好的描述这种行为。 然而，因为携带交互数据的GET请求并不被广泛支持，所以searchAPI同样支持POST请求，类似于这样：

POST /\_search

{

"from": 30,

"size": 10

}

这个原理同样应用于其他携带交互数据的GETAPI请求中。

我们将在后续的章节中讨论聚合查询，但是现在我们把关注点仅放在查询语义上。

相对于神秘的查询字符串方法，请求体查询允许我们使用结构化查询Query DSL(Query Domain Specific Language)

## 7.2结构化查询 Query DSL

结构化查询是一种灵活的，多表现形式的查询语言。 Elasticsearch在一个简单的JSON接口中用结构化查询来展现Lucene绝大多数能力。 你应当在你的产品中采用这种方式进行查询。它使得你的查询更加灵活，精准，易于阅读并且易于debug。

使用结构化查询，你需要传递query参数：

GET /\_search

{

"query": YOUR\_QUERY\_HERE

}

空查询 - {} - 在功能上等同于使用match\_all查询子句，正如其名字一样，匹配所有的文档：

GET /\_search

{

"query": {

"match\_all": {} //match\_all匹配条件，空对象表示匹配所有

}

}

### 查询子句

一个查询子句一般使用这种结构：

{

QUERY\_NAME: {

ARGUMENT: VALUE,

ARGUMENT: VALUE,...

}

}

或指向一个指定的字段：

{

QUERY\_NAME: {

FIELD\_NAME: {

ARGUMENT: VALUE,

ARGUMENT: VALUE,...

}

}

}

例如，你可以使用match查询子句用来找寻在tweet字段中找寻包含elasticsearch的成员：

{

"match": {

"tweet": "elasticsearch"

}

}

完整的查询请求会是这样：

GET /\_search

{

"query": {

"match": {

"tweet": "elasticsearch"

}

}

}

### 合并多子句

查询子句就像是搭积木一样，可以合并简单的子句为一个复杂的查询语句，比如：

* 叶子子句，n:查询语句的最小单元(leaf clauses)(比如match子句)用以在将查询字符串与一个字段(或多字段)进行比较
* 复合子句，n:由多个叶子子句或者其他复合语句组合而成(compound)用以合并其他的子句。例如，bool子句允许你合并其他的合法子句，must，must\_not或者should，如果可能的话：

{

"bool": {

"must": { "match": { "tweet": "elasticsearch" }},

"must\_not": { "match": { "name": "mary" }},

"should": { "match": { "tweet": "full text" }}

}

}

复合子句能合并 **任意**其他查询子句，包括其他的复合子句。 这就意味着复合子句可以相互嵌套，从而实现非常复杂的逻辑。

以下实例查询的是邮件正文中含有“business opportunity”字样的星标邮件或收件箱中正文中含有“business opportunity”字样的非垃圾邮件：

{

"bool": {

"must": { "match": { "email": "business opportunity" }},

"should": [

{ "match": { "starred": true }},

{ "bool": {

"must": { "folder": "inbox" }},

"must\_not": { "spam": true }}

}}

],

"minimum\_should\_match": 1

}

}

不用担心这个例子的细节，我们将在后面详细解释它。 重点是复合子句可以合并多种子句为一个单一的查询，无论是叶子子句还是其他的复合子句。

## 7.3查询与过滤

前面我们讲到的是关于结构化查询语句，事实上我们可以使用两种结构化语句： 结构化查询（Query DSL）和结构化过滤（Filter DSL）。 查询与过滤语句非常相似，但是它们由于使用目的不同而稍有差异。

一条过滤语句会询问每个文档的字段值是否包含着特定值：

* created 的日期范围是否在 2013 到 2014 ?
* status 字段中是否包含单词 "published" ?
* lat\_lon 字段中的地理位置与目标点相距是否不超过10km ?

一条查询语句与过滤语句相似，但问法不同：

查询语句会询问每个文档的字段值与特定值的匹配程度如何？

查询语句的典型用法是为了找到文档：

* 查找与 full text search 这个词语最佳匹配的文档
* 查找包含单词 run ，但是也包含runs, running, jog 或 sprint的文档
* 同时包含着 quick, brown 和 fox --- 单词间离得越近，该文档的相关性越高
* 标识着 lucene, search 或 java --- 标识词越多，该文档的相关性越高

一条查询语句会计算每个文档与查询语句的相关性，会给出一个相关性评分 \_score，并且 按照相关性对匹配到的文档进行排序。 这种评分方式非常适用于一个没有完全配置结果的全文本搜索。

## 性能差异

使用过滤语句得到的结果集 -- 一个简单的文档列表，快速匹配运算并存入内存是十分方便的， 每个文档仅需要1个字节。这些缓存的过滤结果集与后续请求的结合使用是非常高效的。

查询语句不仅要查找相匹配的文档，还需要计算每个文档的相关性，所以一般来说查询语句要比 过滤语句更耗时，并且查询结果也不可缓存。

幸亏有了倒排索引，一个只匹配少量文档的简单查询语句在百万级文档中的查询效率会与一条经过缓存 的过滤语句旗鼓相当，甚至略占上风。 但是一般情况下，一条经过缓存的过滤查询要远胜一条查询语句的执行效率。

过滤语句的目的就是缩小匹配的文档结果集，所以需要仔细检查过滤条件。

## 什么情况下使用

原则上来说，使用查询语句做全文本搜索或其他需要进行相关性评分的时候，剩下的全部用过滤语句

## 7.4最重要的查询过滤语句

Elasticsearch 提供了丰富的查询过滤语句，而有一些是我们较常用到的。 我们将会在后续的《深入搜索》中展开讨论，现在我们快速的介绍一下 这些最常用到的查询过滤语句。

## term 过滤

term主要用于精确匹配哪些值，比如数字，日期，布尔值或 not\_analyzed的字符串(未经分析的文本数据类型)：

{ "term": { "age": 26 }}

{ "term": { "date": "2014-09-01" }}

{ "term": { "public": true }}

{ "term": { "tag": "full\_text" }}

## terms 过滤

terms 跟 term 有点类似，但 terms 允许指定多个匹配条件。 如果某个字段指定了多个值，那么文档需要一起去做匹配：

{

"terms": {

"tag": [ "search", "full\_text", "nosql" ]

}

}

## range 过滤

range过滤允许我们按照指定范围查找一批数据：

{

"range": {

"age": {

"gte": 20,

"lt": 30

}

}

}

范围操作符包含：

gt :: 大于

gte:: 大于等于

lt :: 小于

lte:: 小于等于

## exists 和 missing 过滤

exists 和 missing 过滤可以用于查找文档中是否包含指定字段或没有某个字段，类似于SQL语句中的IS\_NULL条件

{

"exists": {

"field": "title"

}

}

这两个过滤只是针对已经查出一批数据来，但是想区分出某个字段是否存在的时候使用。

## bool 过滤

bool 过滤可以用来合并多个过滤条件查询结果的布尔逻辑，它包含一下操作符：

must :: 多个查询条件的完全匹配,相当于 and。

must\_not :: 多个查询条件的相反匹配，相当于 not。

should :: 至少有一个查询条件匹配, 相当于 or。

这些参数可以分别继承一个过滤条件或者一个过滤条件的数组：

{

"bool": {

"must": { "term": { "folder": "inbox" }},

"must\_not": { "term": { "tag": "spam" }},

"should": [

{ "term": { "starred": true }},

{ "term": { "unread": true }}

]

}

}

## match\_all 查询

使用match\_all 可以查询到所有文档，是没有查询条件下的默认语句。

{

"match\_all": {}

}

此查询常用于合并过滤条件。 比如说你需要检索所有的邮箱,所有的文档相关性都是相同的，所以得到的\_score为1

## match 查询

match查询是一个标准查询，不管你需要全文本查询还是精确查询基本上都要用到它。

如果你使用 match 查询一个全文本字段，它会在真正查询之前用分析器先分析match一下查询字符：

{

"match": {

"tweet": "About Search"

}

}

如果用match下指定了一个确切值，在遇到数字，日期，布尔值或者not\_analyzed 的字符串时，它将为你搜索你给定的值：

{ "match": { "age": 26 }}

{ "match": { "date": "2014-09-01" }}

{ "match": { "public": true }}

{ "match": { "tag": "full\_text" }}

**提示**： 做精确匹配搜索时，你最好用过滤语句，因为过滤语句可以缓存数据。

不像我们在《简单搜索》中介绍的字符查询，match查询不可以用类似"+usid:2 +tweet:search"这样的语句。 它只能就指定某个确切字段某个确切的值进行搜索，而你要做的就是为它指定正确的字段名以避免语法错误。

## multi\_match 查询

multi\_match查询允许你做match查询的基础上同时搜索多个字段：

{

"multi\_match": {

"query": "full text search",

"fields": [ "title", "body" ]

}

}

## bool 查询

bool 查询与 bool 过滤相似，用于合并多个查询子句。不同的是，bool 过滤可以直接给出是否匹配成功， 而bool 查询要计算每一个查询子句的 \_score （相关性分值）。

must:: 查询指定文档一定要被包含。

must\_not:: 查询指定文档一定不要被包含。

should:: 查询指定文档，有则可以为文档相关性加分。

以下查询将会找到 title 字段中包含 "how to make millions"，并且 "tag" 字段没有被标为 spam。 如果有标识为 "starred" 或者发布日期为2014年之前，那么这些匹配的文档将比同类网站等级高：

{

"bool": {

"must": { "match": { "title": "how to make millions" }},

"must\_not": { "match": { "tag": "spam" }},

"should": [

{ "match": { "tag": "starred" }},

{ "range": { "date": { "gte": "2014-01-01" }}}

]

}

}

**提示**： 如果bool 查询下没有must子句，那至少应该有一个should子句。但是 如果有must子句，那么没有should子句也可以进行查询。

## 7.5查询与过滤条件的合并

查询语句和过滤语句可以放在各自的上下文中。 在 ElasticSearch API 中我们会看到许多带有 query 或 filter 的语句。 这些语句既可以包含单条 query 语句，也可以包含一条 filter 子句。 换句话说，这些语句需要首先创建一个query或filter的上下文关系。

复合查询语句可以加入其他查询子句，复合过滤语句也可以加入其他过滤子句。 通常情况下，一条查询语句需要过滤语句的辅助，全文本搜索除外。

所以说，查询语句可以包含过滤子句，反之亦然。 以便于我们切换 query 或 filter 的上下文。这就要求我们在读懂需求的同时构造正确有效的语句。

## 带过滤的查询语句

#### 过滤一条查询语句

比如说我们有这样一条查询语句:

{

"match": {

"email": "business opportunity"

}

}

然后我们想要让这条语句加入 term 过滤，在收信箱中匹配邮件：

{

"term": {

"folder": "inbox"

}

}

search API中只能包含 query 语句，所以我们需要用 filtered 来同时包含 "query" 和 "filter" 子句：

{

"filtered": {

"query": { "match": { "email": "business opportunity" }},

"filter": { "term": { "folder": "inbox" }}

}

}

我们在外层再加入 query 的上下文关系：

GET /\_search

{

"query": {

"filtered": {

"query": { "match": { "email": "business opportunity" }},

"filter": { "term": { "folder": "inbox" }}

}

}

}

## 单条过滤语句

在 query 上下文中，如果你只需要一条过滤语句，比如在匹配全部邮件的时候，你可以 省略 query 子句：

GET /\_search

{

"query": {

"filtered": {

"filter": { "term": { "folder": "inbox" }}

}

}

}

如果一条查询语句没有指定查询范围，那么它默认使用 match\_all 查询，所以上面语句 的完整形式如下：

GET /\_search

{

"query": {

"filtered": {

"query": { "match\_all": {}},

"filter": { "term": { "folder": "inbox" }}

}

}

}

## 查询语句中的过滤

有时候，你需要在 filter 的上下文中使用一个 query 子句。下面的语句就是一条带有查询功能 的过滤语句， 这条语句可以过滤掉看起来像垃圾邮件的文档：

GET /\_search

{

"query": {

"filtered": {

"filter": {

"bool": {

"must": { "term": { "folder": "inbox" }},

"must\_not": {

"query": { <1>

"match": { "email": "urgent business proposal" }

}

}

}

}

}

}

}

<1> 过滤语句中可以使用query查询的方式代替 bool 过滤子句。

**提示**： 我们很少用到的过滤语句中包含查询，保留这种用法只是为了语法的完整性。 只有在过滤中用到全文本匹配的时候才会使用这种结构。

## 7.6验证查询

查询语句可以变得非常复杂，特别是与不同的分析器和字段映射相结合后，就会有些难度。

validate API 可以验证一条查询语句是否合法。

GET /gb/tweet/\_validate/query

{

"query": {

"tweet" : {

"match" : "really powerful"

}

}

}

以上请求的返回值告诉我们这条语句是非法的：

{

"valid" : false,

"\_shards" : {

"total" : 1,

"successful" : 1,

"failed" : 0

}

}

## 理解错误信息

想知道语句非法的具体错误信息，需要加上 explain 参数：

GET /gb/tweet/\_validate/query?explain <1>

{

"query": {

"tweet" : {

"match" : "really powerful"

}

}

}

<1> explain 参数可以提供语句错误的更多详情。

很显然，我们把 query 语句的 match 与字段名位置弄反了：

{

"valid" : false,

"\_shards" : { ... },

"explanations" : [ {

"index" : "gb",

"valid" : false,

"error" : "org.elasticsearch.index.query.QueryParsingException:

[gb] No query registered for [tweet]"

} ]

}

## 理解查询语句

如果是合法语句的话，使用 explain 参数可以返回一个带有查询语句的可阅读描述， 可以帮助了解查询语句在ES中是如何执行的：

GET /\_validate/query?explain

{

"query": {

"match" : {

"tweet" : "really powerful"

}

}

}

explanation 会为每一个索引返回一段描述，因为每个索引会有不同的映射关系和分析器：

{

"valid" : true,

"\_shards" : { ... },

"explanations" : [ {

"index" : "us",

"valid" : true,

"explanation" : "tweet:really tweet:powerful"

}, {

"index" : "gb",

"valid" : true,

"explanation" : "tweet:really tweet:power"

} ]

}

从返回的 explanation 你会看到 match 是如何为查询字符串 "really powerful" 进行查询的， 首先，它被拆分成两个独立的词分别在 tweet 字段中进行查询。

而且，在索引us中这两个词为"really"和"powerful"，在索引gb中被拆分成"really" 和 "power"。 这是因为我们在索引gb中使用了english分析器。

## 结语

这一章详细介绍了如何在项目中使用常见的查询语句。

也就是说，想要完全掌握搜索和结构化查询，还需要在工作中花费大量的时间来理解ES的工作方式。

更高级的部分，我们将会在《深入搜索》中详细讲解，但是在讲解之前，你还需要理解查询结果是如何进行排序的，

下一章我们将学习如何根据相关性对查询结果进行排序以及指定排序过程。