Elasticsearch 权威指南

深入搜索[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/01_Search_in_depth.asciidoc)

在 [基础入门](https://www.elastic.co/guide/cn/elasticsearch/guide/current/getting-started.html) 中涵盖了基本工具并对它们有足够详细的描述，这让我们能够开始用 Elasticsearch 搜索数据。 用不了多长时间，就会发现我们想要的更多：希望查询匹配更灵活，排名结果更精确，不同问题域下搜索更具体。

1. 想要进阶，只知道如何使用 match 查询是不够的，我们需要理解数据以及如何能够搜索到它们。本章会解释如何索引和查询我们的数据让我们能利用词的相似度（word proximity）、部分匹配（partial matching）、模糊匹配（fuzzy matching）以及语言感知（language awareness）这些优势。

理解每个查询如何贡献相关度评分 \_score 有助于调试我们的查询：确保我们认为的最佳匹配文档出现在结果首页，以及削减结果中几乎不相关的 “长尾（long tail）”。

搜索不仅仅是全文搜索：我们很大一部分数据都是结构化的，如日期和数字。 我们会以说明结构化搜索与全文搜索最高效的结合方式开始本章的内容。

## 结构化搜索[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/080_Structured_Search/00_structuredsearch.asciidoc)

结构化搜索（Structured search） 是指有关探询那些具有内在结构数据的过程。比如日期、时间和数字都是结构化的：它们有精确的格式，我们可以对这些格式进行逻辑操作。比较常见的操作包括比较数字或时间的范围，或判定两个值的大小。

文本也可以是结构化的。如彩色笔可以有离散的颜色集合： 红（red） 、 绿（green） 、 蓝（blue） 。一个博客可能被标记了关键词 分布式（distributed） 和 搜索（search） 。电商网站上的商品都有 UPCs（通用产品码 Universal Product Codes）或其他的唯一标识，它们都需要遵从严格规定的、结构化的格式。

在结构化查询中，我们得到的结果 总是 非是即否，要么存于集合之中，要么存在集合之外。结构化查询不关心文件的相关度或评分；它简单的对文档包括或排除处理。

这在逻辑上是能说通的，因为一个数字不能比其他数字 更 适合存于某个相同范围。结果只能是：存于范围之中，抑或反之。同样，对于结构化文本来说，一个值要么相等，要么不等。没有 更似 这种概念。

### 精确值查找[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/080_Structured_Search/05_term.asciidoc)

当进行精确值查找时， 我们会使用过滤器（filters）。过滤器很重要，因为它们执行速度非常快，不会计算相关度（直接跳过了整个评分阶段）而且很容易被缓存。我们会在本章后面的 [过滤器缓存](https://www.elastic.co/guide/cn/elasticsearch/guide/current/filter-caching.html) 中讨论过滤器的性能优势，不过现在只要记住：请尽可能多的使用过滤式查询。

#### term 查询数字[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/080_Structured_Search/05_term.asciidoc)

我们首先来看最为常用的 term 查询， 可以用它处理数字（numbers）、布尔值（Booleans）、日期（dates）以及文本（text）。

让我们以下面的例子开始介绍，创建并索引一些表示产品的文档，文档里有字段 `price` 和 `productID` （ `价格` 和 `产品ID` ）：

POST /my\_store/products/\_bulk

{ "index": { "\_id": 1 }}

{ "price" : 10, "productID" : "XHDK-A-1293-#fJ3" }

{ "index": { "\_id": 2 }}

{ "price" : 20, "productID" : "KDKE-B-9947-#kL5" }

{ "index": { "\_id": 3 }}

{ "price" : 30, "productID" : "JODL-X-1937-#pV7" }

{ "index": { "\_id": 4 }}

{ "price" : 30, "productID" : "QQPX-R-3956-#aD8" }

我们想要做的是查找具有某个价格的所有产品，有关系数据库背景的人肯定熟悉 SQL，如果我们将其用 SQL 形式表达，会是下面这样：

SELECT document

FROM products

WHERE price = 20

在 Elasticsearch 的查询表达式（query DSL）中，我们可以使用 term 查询达到相同的目的。 term 查询会查找我们指定的精确值。作为其本身， term 查询是简单的。它接受一个字段名以及我们希望查找的数值：

{

"term" : {

"price" : 20

}

}

通常当查找一个精确值的时候，我们不希望对查询进行评分计算。只希望对文档进行包括或排除的计算，所以我们会使用 constant\_score 查询以非评分模式来执行 term 查询并以一作为统一评分。

最终组合的结果是一个 constant\_score 查询，它包含一个 term 查询：

GET /my\_store/products/\_search

{

"query" : {

"constant\_score" : { https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

"filter" : {

"term" : { https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

"price" : 20

}

}

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_finding_exact_values.html#CO42-1) | 我们用 constant\_score 将 term 查询转化成为过滤器 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_finding_exact_values.html#CO42-2) | 我们之前看到过的 term 查询 |

执行后，这个查询所搜索到的结果与我们期望的一致：只有文档 2 命中并作为结果返回（因为只有 2 的价格是 20 ）:

"hits" : [

{

"\_index" : "my\_store",

"\_type" : "products",

"\_id" : "2",

"\_score" : 1.0, https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

"\_source" : {

"price" : 20,

"productID" : "KDKE-B-9947-#kL5"

}

}

]

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_finding_exact_values.html#CO43-1) | 查询置于 filter 语句内不进行评分或相关度的计算，所以所有的结果都会返回一个默认评分 1 。 |

#### term 查询文本[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/080_Structured_Search/05_term.asciidoc)

如本部分开始处提到过的一样 ，使用 term 查询匹配字符串和匹配数字一样容易。如果我们想要查询某个具体 UPC ID 的产品，使用 SQL 表达式会是如下这样：

SELECT product

FROM products

WHERE productID = "XHDK-A-1293-#fJ3"

转换成查询表达式（query DSL），同样使用 term 查询，形式如下：

GET /my\_store/products/\_search

{

"query" : {

"constant\_score" : {

"filter" : {

"term" : {

"productID" : "XHDK-A-1293-#fJ3"

}

}

}

}

}

但这里有个小问题：我们无法获得期望的结果。为什么呢？问题不在 term 查询，而在于索引数据的方式。如果我们使用 analyze API ([分析 API](https://www.elastic.co/guide/cn/elasticsearch/guide/current/analysis-intro.html#analyze-api))，我们可以看到这里的 UPC 码被拆分成多个更小的 token ：

GET /my\_store/\_analyze

{

"field": "productID",

"text": "XHDK-A-1293-#fJ3"

}

{

"tokens" : [ {

"token" : "xhdk",

"start\_offset" : 0,

"end\_offset" : 4,

"type" : "<ALPHANUM>",

"position" : 1

}, {

"token" : "a",

"start\_offset" : 5,

"end\_offset" : 6,

"type" : "<ALPHANUM>",

"position" : 2

}, {

"token" : "1293",

"start\_offset" : 7,

"end\_offset" : 11,

"type" : "<NUM>",

"position" : 3

}, {

"token" : "fj3",

"start\_offset" : 13,

"end\_offset" : 16,

"type" : "<ALPHANUM>",

"position" : 4

} ]

}

这里有几点需要注意：

* Elasticsearch 用 4 个不同的 token 而不是单个 token 来表示这个 UPC 。
* 所有字母都是小写的。
* 丢失了连字符和哈希符（ # ）。

所以当我们用 term 查询查找精确值 XHDK-A-1293-#fJ3 的时候，找不到任何文档，因为它并不在我们的倒排索引中，正如前面呈现出的分析结果，索引里有四个 token 。

显然这种对 ID 码或其他任何精确值的处理方式并不是我们想要的。

为了避免这种问题，我们需要告诉 Elasticsearch 该字段具有精确值，要将其设置成 not\_analyzed 无需分析的。 我们可以在 [自定义字段映射](https://www.elastic.co/guide/cn/elasticsearch/guide/current/mapping-intro.html#custom-field-mappings) 中查看它的用法。为了修正搜索结果，我们需要首先删除旧索引（因为它的映射不再正确）然后创建一个能正确映射的新索引：

DELETE /my\_store https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

PUT /my\_store https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

{

"mappings" : {

"products" : {

"properties" : {

"productID" : {

"type" : "string",

"index" : "not\_analyzed" https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png

}

}

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_finding_exact_values.html#CO44-1) | 删除索引是必须的，因为我们不能更新已存在的映射。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_finding_exact_values.html#CO44-2) | 在索引被删除后，我们可以创建新的索引并为其指定自定义映射。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_finding_exact_values.html#CO44-3) | 这里我们告诉 Elasticsearch ，我们不想对 productID 做任何分析。 |

现在我们可以为文档重建索引：

POST /my\_store/products/\_bulk

{ "index": { "\_id": 1 }}

{ "price" : 10, "productID" : "XHDK-A-1293-#fJ3" }

{ "index": { "\_id": 2 }}

{ "price" : 20, "productID" : "KDKE-B-9947-#kL5" }

{ "index": { "\_id": 3 }}

{ "price" : 30, "productID" : "JODL-X-1937-#pV7" }

{ "index": { "\_id": 4 }}

{ "price" : 30, "productID" : "QQPX-R-3956-#aD8" }

此时， term 查询就能搜索到我们想要的结果，让我们再次搜索新索引过的数据（注意，查询和过滤并没有发生任何改变，改变的是数据映射的方式）：

GET /my\_store/products/\_search

{

"query" : {

"constant\_score" : {

"filter" : {

"term" : {

"productID" : "XHDK-A-1293-#fJ3"

}

}

}

}

}

因为 productID 字段是未分析过的， term 查询不会对其做任何分析，查询会进行精确查找并返回文档 1 。成功！

#### 内部过滤器的操作[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/080_Structured_Search/05_term.asciidoc)

在内部，Elasticsearch 会在运行非评分查询的时执行多个操作：

1. 查找匹配文档.

term 查询在倒排索引中查找 XHDK-A-1293-#fJ3 然后获取包含该 term 的所有文档。本例中，只有文档 1 满足我们要求。

1. 创建 bitset.

过滤器会创建一个 bitset （一个包含 0 和 1 的数组），它描述了哪个文档会包含该 term 。匹配文档的标志位是 1 。本例中，bitset 的值为 [1,0,0,0] 。在内部，它表示成一个 ["roaring bitmap"](https://www.elastic.co/blog/frame-of-reference-and-roaring-bitmaps)，可以同时对稀疏或密集的集合进行高效编码。

1. 迭代 bitset(s)

一旦为每个查询生成了 bitsets ，Elasticsearch 就会循环迭代 bitsets 从而找到满足所有过滤条件的匹配文档的集合。执行顺序是启发式的，但一般来说先迭代稀疏的 bitset （因为它可以排除掉大量的文档）。

1. 增量使用计数.

Elasticsearch 能够缓存非评分查询从而获取更快的访问，但是它也会不太聪明地缓存一些使用极少的东西。非评分计算因为倒排索引已经足够快了，所以我们只想缓存那些我们 知道 在将来会被再次使用的查询，以避免资源的浪费。

为了实现以上设想，Elasticsearch 会为每个索引跟踪保留查询使用的历史状态。如果查询在最近的 256 次查询中会被用到，那么它就会被缓存到内存中。当 bitset 被缓存后，缓存会在那些低于 10,000 个文档（或少于 3% 的总索引数）的段（segment）中被忽略。这些小的段即将会消失，所以为它们分配缓存是一种浪费。

实际情况并非如此（执行有它的复杂性，这取决于查询计划是如何重新规划的，有些启发式的算法是基于查询代价的），理论上非评分查询 先于 评分查询执行。非评分查询任务旨在降低那些将对评分查询计算带来更高成本的文档数量，从而达到快速搜索的目的。

从概念上记住非评分计算是首先执行的，这将有助于写出高效又快速的搜索请求。

### 组合过滤器[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/080_Structured_Search/10_compoundfilters.asciidoc)

前面的两个例子都是单个过滤器（filter）的使用方式。 在实际应用中，我们很有可能会过滤多个值或字段。比方说，怎样用 Elasticsearch 来表达下面的 SQL ？

SELECT product

FROM products

WHERE (price = 20 OR productID = "XHDK-A-1293-#fJ3")

AND (price != 30)

这种情况下，我们需要 bool （布尔）过滤器。 这是个 复合过滤器（compound filter） ，它可以接受多个其他过滤器作为参数，并将这些过滤器结合成各式各样的布尔（逻辑）组合。

#### 布尔过滤器[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/080_Structured_Search/10_compoundfilters.asciidoc)

一个 bool 过滤器由三部分组成：

{

"bool" : {

"must" : [],

"should" : [],

"must\_not" : [],

}

}

must

所有的语句都 必须（must） 匹配，与 AND 等价。

must\_not

所有的语句都 不能（must not） 匹配，与 NOT 等价。

should

至少有一个语句要匹配，与 OR 等价。

就这么简单！ 当我们需要多个过滤器时，只须将它们置入 bool 过滤器的不同部分即可。

注意

一个 bool 过滤器的每个部分都是可选的（例如，我们可以只有一个 must 语句），而且每个部分内部可以只有一个或一组过滤器。

用 Elasticsearch 来表示本部分开始处的 SQL 例子，将两个 term 过滤器置入 bool 过滤器的 should 语句内，再增加一个语句处理 NOT 非的条件：

GET /my\_store/products/\_search

{

"query" : {

"filtered" : { https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

"filter" : {

"bool" : {

"should" : [

{ "term" : {"price" : 20}}, https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

{ "term" : {"productID" : "XHDK-A-1293-#fJ3"}} https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png

],

"must\_not" : {

"term" : {"price" : 30} https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/4.png

}

}

}

}

}

}

**拷贝为 CURL**[在 SENSE 中查看](http://localhost:5601/app/sense/?load_from=https://www.elastic.co/guide/cn/elasticsearch/guide/current/snippets/080_Structured_Search/10_Bool_filter.json)

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/combining-filters.html#CO45-1) | 注意，我们仍然需要一个 filtered 查询将所有的东西包起来。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/combining-filters.html#CO45-2) [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/combining-filters.html#CO45-3) | 在 should 语句块里面的两个 term 过滤器与 bool 过滤器是父子关系，两个 term 条件需要匹配其一。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/4.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/combining-filters.html#CO45-4) | 如果一个产品的价格是 30 ，那么它会自动被排除，因为它处于 must\_not 语句里面。 |

我们搜索的结果返回了 2 个命中结果，两个文档分别匹配了 bool 过滤器其中的一个条件：

"hits" : [

{

"\_id" : "1",

"\_score" : 1.0,

"\_source" : {

"price" : 10,

"productID" : "XHDK-A-1293-#fJ3" https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

}

},

{

"\_id" : "2",

"\_score" : 1.0,

"\_source" : {

"price" : 20, https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

"productID" : "KDKE-B-9947-#kL5"

}

}

]

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/combining-filters.html#CO46-1) | 与 term 过滤器中 productID = "XHDK-A-1293-#fJ3" 条件匹配 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/combining-filters.html#CO46-2) | 与 term 过滤器中 price = 20 条件匹配 |

#### 嵌套布尔过滤器[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/080_Structured_Search/10_compoundfilters.asciidoc)

尽管 bool 是一个复合的过滤器，可以接受多个子过滤器，需要注意的是 bool 过滤器本身仍然还只是一个过滤器。 这意味着我们可以将一个 bool 过滤器置于其他 bool 过滤器内部，这为我们提供了对任意复杂布尔逻辑进行处理的能力。

对于以下这个 SQL 语句：

SELECT document

FROM products

WHERE productID = "KDKE-B-9947-#kL5"

OR ( productID = "JODL-X-1937-#pV7"

AND price = 30 )

我们将其转换成一组嵌套的 bool 过滤器：

GET /my\_store/products/\_search

{

"query" : {

"filtered" : {

"filter" : {

"bool" : {

"should" : [

{ "term" : {"productID" : "KDKE-B-9947-#kL5"}}, https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

{ "bool" : { https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

"must" : [

{ "term" : {"productID" : "JODL-X-1937-#pV7"}}, https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png

{ "term" : {"price" : 30}} https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/4.png

]

}}

]

}

}

}

}

}

**拷贝为 CURL**[在 SENSE 中查看](http://localhost:5601/app/sense/?load_from=https://www.elastic.co/guide/cn/elasticsearch/guide/current/snippets/080_Structured_Search/10_Bool_filter.json)

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/combining-filters.html#CO47-1) [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/combining-filters.html#CO47-2) | 因为 term 和 bool 过滤器是兄弟关系，他们都处于外层的布尔逻辑 should 的内部，返回的命中文档至少须匹配其中一个过滤器的条件。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/combining-filters.html#CO47-3) [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/4.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/combining-filters.html#CO47-4) | 这两个 term 语句作为兄弟关系，同时处于 must 语句之中，所以返回的命中文档要必须都能同时匹配这两个条件。 |

得到的结果有两个文档，它们各匹配 should 语句中的一个条件：

"hits" : [

{

"\_id" : "2",

"\_score" : 1.0,

"\_source" : {

"price" : 20,

"productID" : "KDKE-B-9947-#kL5" https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

}

},

{

"\_id" : "3",

"\_score" : 1.0,

"\_source" : {

"price" : 30, https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

"productID" : "JODL-X-1937-#pV7" https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png

}

}

]

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/combining-filters.html#CO48-1) | 这个 productID 与外层的 bool 过滤器 should 里的唯一一个 term 匹配。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/combining-filters.html#CO48-2) [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/combining-filters.html#CO48-3) | 这两个字段与嵌套的 bool 过滤器 must 里的两个 term 匹配。 |

这只是个简单的例子，但足以展示布尔过滤器可以用来作为构造复杂逻辑条件的基本构建模块。

### 查找多个精确值[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/080_Structured_Search/15_terms.asciidoc)

term 查询对于查找单个值非常有用，但通常我们可能想搜索多个值。 如果我们想要查找价格字段值为 $20 或 $30 的文档该如何处理呢？

不需要使用多个 term 查询，我们只要用单个 terms 查询（注意末尾的 s ）， terms 查询好比是 term 查询的复数形式（以英语名词的单复数做比）。

它几乎与 term 的使用方式一模一样，与指定单个价格不同，我们只要将 term 字段的值改为数组即可：

{

"terms" : {

"price" : [20, 30]

}

}

与 term 查询一样，也需要将其置入 filter 语句的常量评分查询中使用：

GET /my\_store/products/\_search

{

"query" : {

"constant\_score" : {

"filter" : {

"terms" : { https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

"price" : [20, 30]

}

}

}

}

}

**拷贝为 CURL**[在 SENSE 中查看](http://localhost:5601/app/sense/?load_from=https://www.elastic.co/guide/cn/elasticsearch/guide/current/snippets/080_Structured_Search/15_Terms_filter.json)

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_finding_multiple_exact_values.html#CO49-1) | 这个 terms 查询被置于 constant\_score 查询中 |

运行结果返回第二、第三和第四个文档：

"hits" : [

{

"\_id" : "2",

"\_score" : 1.0,

"\_source" : {

"price" : 20,

"productID" : "KDKE-B-9947-#kL5"

}

},

{

"\_id" : "3",

"\_score" : 1.0,

"\_source" : {

"price" : 30,

"productID" : "JODL-X-1937-#pV7"

}

},

{

"\_id": "4",

"\_score": 1.0,

"\_source": {

"price": 30,

"productID": "QQPX-R-3956-#aD8"

}

}

]

#### 包含，而不是相等[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/080_Structured_Search/20_contains.asciidoc)

一定要了解 term 和 terms 是 包含（contains） 操作，而非 等值（equals） （判断）。 如何理解这句话呢？

如果我们有一个 term（词项）过滤器 { "term" : { "tags" : "search" } } ，它会与以下两个文档 同时匹配：

{ "tags" : ["search"] }

{ "tags" : ["search", "open\_source"] } https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_finding_multiple_exact_values.html#CO50-1) | 尽管第二个文档包含除 search 以外的其他词，它还是被匹配并作为结果返回。 |

回忆一下 term 查询是如何工作的？ Elasticsearch 会在倒排索引中查找包括某 term 的所有文档，然后构造一个 bitset 。在我们的例子中，倒排索引表如下：

|  |  |
| --- | --- |
| Token | DocIDs |
| open\_source | 2 |
| search | 1,2 |

当 term 查询匹配标记 search 时，它直接在倒排索引中找到记录并获取相关的文档 ID，如倒排索引所示，这里文档 1 和文档 2 均包含该标记，所以两个文档会同时作为结果返回。

注意

由于倒排索引表自身的特性，整个字段是否相等会难以计算，如果确定某个特定文档是否 只（only） 包含我们想要查找的词呢？首先我们需要在倒排索引中找到相关的记录并获取文档 ID，然后再扫描 倒排索引中的每行记录 ，查看它们是否包含其他的 terms 。

可以想象，这样不仅低效，而且代价高昂。正因如此， term 和 terms 是 必须包含（must contain） 操作，而不是 必须精确相等（must equal exactly） 。

#### 精确相等[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/080_Structured_Search/20_contains.asciidoc)

如果一定期望得到我们前面说的那种行为（即整个字段完全相等），最好的方式是增加并索引另一个字段， 这个字段用以存储该字段包含词项的数量，同样以上面提到的两个文档为例，现在我们包括了一个维护标签数的新字段：

{ "tags" : ["search"], "tag\_count" : 1 }

{ "tags" : ["search", "open\_source"], "tag\_count" : 2 }

**拷贝为 CURL**[在 SENSE 中查看](http://localhost:5601/app/sense/?load_from=https://www.elastic.co/guide/cn/elasticsearch/guide/current/snippets/080_Structured_Search/20_Exact.json)

一旦增加这个用来索引项 term 数目信息的字段，我们就可以构造一个 constant\_score 查询，来确保结果中的文档所包含的词项数量与要求是一致的：

GET /my\_index/my\_type/\_search

{

"query": {

"constant\_score" : {

"filter" : {

"bool" : {

"must" : [

{ "term" : { "tags" : "search" } }, https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

{ "term" : { "tag\_count" : 1 } } https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

]

}

}

}

}

}

**拷贝为 CURL**[在 SENSE 中查看](http://localhost:5601/app/sense/?load_from=https://www.elastic.co/guide/cn/elasticsearch/guide/current/snippets/080_Structured_Search/20_Exact.json)

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_finding_multiple_exact_values.html#CO51-1) | 查找所有包含 term search 的文档。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_finding_multiple_exact_values.html#CO51-2) | 确保文档只有一个标签。 |

这个查询现在只会匹配具有单个标签 search 的文档，而不是任意一个包含 search 的文档。

### 范围[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/080_Structured_Search/25_ranges.asciidoc)

本章到目前为止，对于数字，只介绍如何处理精确值查询。 实际上，对数字范围进行过滤有时会更有用。例如，我们可能想要查找所有价格大于 $20 且小于 $40 美元的产品。

在 SQL 中，范围查询可以表示为：

SELECT document

FROM products

WHERE price BETWEEN 20 AND 40

Elasticsearch 有 range 查询， 不出所料地，可以用它来查找处于某个范围内的文档：

"range" : {

"price" : {

"gte" : 20,

"lte" : 40

}

}

range 查询可同时提供包含（inclusive）和不包含（exclusive）这两种范围表达式，可供组合的选项如下：

* gt: > 大于（greater than）
* lt: < 小于（less than）
* gte: >= 大于或等于（greater than or equal to）
* lte: <= 小于或等于（less than or equal to）

**下面是一个范围查询的例子：.**

GET /my\_store/products/\_search

{

"query" : {

"constant\_score" : {

"filter" : {

"range" : {

"price" : {

"gte" : 20,

"lt" : 40

}

}

}

}

}

}

**拷贝为 CURL**[在 SENSE 中查看](http://localhost:5601/app/sense/?load_from=https://www.elastic.co/guide/cn/elasticsearch/guide/current/snippets/080_Structured_Search/25_Range_filter.json)

如果想要范围无界（比方说 >20 ），只须省略其中一边的限制：

"range" : {

"price" : {

"gt" : 20

}

}

**拷贝为 CURL**[在 SENSE 中查看](http://localhost:5601/app/sense/?load_from=https://www.elastic.co/guide/cn/elasticsearch/guide/current/snippets/080_Structured_Search/25_Range_filter.json)

#### 日期范围[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/080_Structured_Search/25_ranges.asciidoc)

range 查询同样可以应用在日期字段上：

"range" : {

"timestamp" : {

"gt" : "2014-01-01 00:00:00",

"lt" : "2014-01-07 00:00:00"

}

}

当使用它处理日期字段时， range 查询支持对 日期计算（date math） 进行操作，比方说，如果我们想查找时间戳在过去一小时内的所有文档：

"range" : {

"timestamp" : {

"gt" : "now-1h"

}

}

这个过滤器会一直查找时间戳在过去一个小时内的所有文档，让过滤器作为一个时间 滑动窗口（sliding window） 来过滤文档。

日期计算还可以被应用到某个具体的时间，并非只能是一个像 now 这样的占位符。只要在某个日期后加上一个双管符号 (||) 并紧跟一个日期数学表达式就能做到：

"range" : {

"timestamp" : {

"gt" : "2014-01-01 00:00:00",

"lt" : "2014-01-01 00:00:00||+1M" https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_ranges.html#CO52-1) | 早于 2014 年 1 月 1 日加 1 月（2014 年 2 月 1 日 零时） |

日期计算是 日历相关（calendar aware） 的，所以它不仅知道每月的具体天数，还知道某年的总天数（闰年）等信息。更详细的内容可以参考： [时间格式参考文档](https://www.elastic.co/guide/en/elasticsearch/reference/5.6/mapping-date-format.html) 。

#### 字符串范围[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/080_Structured_Search/25_ranges.asciidoc)

range 查询同样可以处理字符串字段， 字符串范围可采用 字典顺序（lexicographically） 或字母顺序（alphabetically）。例如，下面这些字符串是采用字典序（lexicographically）排序的：

* 5, 50, 6, B, C, a, ab, abb, abc, b

注意

在倒排索引中的词项就是采取字典顺序（lexicographically）排列的，这也是字符串范围可以使用这个顺序来确定的原因。

如果我们想查找从 a 到 b （不包含）的字符串，同样可以使用 range 查询语法：

"range" : {

"title" : {

"gte" : "a",

"lt" : "b"

}

}

**注意基数**

数字和日期字段的索引方式使高效地范围计算成为可能。 但字符串却并非如此，要想对其使用范围过滤，Elasticsearch 实际上是在为范围内的每个词项都执行 term 过滤器，这会比日期或数字的范围过滤慢许多。

字符串范围在过滤 低基数（low cardinality） 字段（即只有少量唯一词项）时可以正常工作，但是唯一词项越多，字符串范围的计算会越慢。

### 处理 Null 值[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/080_Structured_Search/30_existsmissing.asciidoc)

回想在之前例子中，有的文档有名为 tags （标签）的字段，它是个多值字段， 一个文档可能有一个或多个标签，也可能根本就没有标签。如果一个字段没有值，那么如何将它存入倒排索引中的呢？

这是个有欺骗性的问题，因为答案是：什么都不存。让我们看看之前内容里提到过的倒排索引：

|  |  |
| --- | --- |
| Token | DocIDs |
| open\_source | 2 |
| search | 1,2 |

如何将某个不存在的字段存储在这个数据结构中呢？无法做到！简单的说，一个倒排索引只是一个 token 列表和与之相关的文档信息，如果字段不存在，那么它也不会持有任何 token，也就无法在倒排索引结构中表现。

最终，这也就意味着 ，null, [] （空数组）和 [null] 所有这些都是等价的，它们无法存于倒排索引中。

显然，世界并不简单，数据往往会有缺失字段，或有显式的空值或空数组。为了应对这些状况，Elasticsearch 提供了一些工具来处理空或缺失值。

#### 存在查询[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/080_Structured_Search/30_existsmissing.asciidoc)

第一件武器就是 exists 存在查询。 这个查询会返回那些在指定字段有任何值的文档，让我们索引一些示例文档并用标签的例子来说明：

POST /my\_index/posts/\_bulk

{ "index": { "\_id": "1" }}

{ "tags" : ["search"] } https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

{ "index": { "\_id": "2" }}

{ "tags" : ["search", "open\_source"] } https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

{ "index": { "\_id": "3" }}

{ "other\_field" : "some data" } https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png

{ "index": { "\_id": "4" }}

{ "tags" : null } https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/4.png

{ "index": { "\_id": "5" }}

{ "tags" : ["search", null] } https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/5.png

**拷贝为 CURL**[在 SENSE 中查看](http://localhost:5601/app/sense/?load_from=https://www.elastic.co/guide/cn/elasticsearch/guide/current/snippets/080_Structured_Search/30_Exists_missing.json)

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_dealing_with_null_values.html#CO53-1) | tags 字段有 1 个值。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_dealing_with_null_values.html#CO53-2) | tags 字段有 2 个值。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_dealing_with_null_values.html#CO53-3) | tags 字段缺失。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/4.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_dealing_with_null_values.html#CO53-4) | tags 字段被置为 null 。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/5.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_dealing_with_null_values.html#CO53-5) | tags 字段有 1 个值和 1 个 null 。 |

以上文档集合中 tags 字段对应的倒排索引如下：

|  |  |
| --- | --- |
| Token | DocIDs |
| open\_source | 2 |
| search | 1,2,5 |

我们的目标是找到那些被设置过标签字段的文档，并不关心标签的具体内容。只要它存在于文档中即可，用 SQL 的话就是用 IS NOT NULL 非空进行查询：

SELECT tags

FROM posts

WHERE tags IS NOT NULL

在 Elasticsearch 中，使用 exists 查询的方式如下：

GET /my\_index/posts/\_search

{

"query" : {

"constant\_score" : {

"filter" : {

"exists" : { "field" : "tags" }

}

}

}

}

**拷贝为 CURL**[在 SENSE 中查看](http://localhost:5601/app/sense/?load_from=https://www.elastic.co/guide/cn/elasticsearch/guide/current/snippets/080_Structured_Search/30_Exists_missing.json)

这个查询返回 3 个文档：

"hits" : [

{

"\_id" : "1",

"\_score" : 1.0,

"\_source" : { "tags" : ["search"] }

},

{

"\_id" : "5",

"\_score" : 1.0,

"\_source" : { "tags" : ["search", null] } https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

},

{

"\_id" : "2",

"\_score" : 1.0,

"\_source" : { "tags" : ["search", "open source"] }

}

]

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_dealing_with_null_values.html#CO54-1) | 尽管文档 5 有 null 值，但它仍会被命中返回。字段之所以存在，是因为标签有实际值（ search ）可以被索引，所以 null 对过滤不会产生任何影响。 |

显而易见，只要 tags 字段存在项（term）的文档都会命中并作为结果返回，只有 3 和 4 两个文档被排除。

#### 缺失查询[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/080_Structured_Search/30_existsmissing.asciidoc)

这个 missing 查询本质上与 exists 恰好相反： 它返回某个特定 \_无\_ 值字段的文档，与以下 SQL 表达的意思类似：

SELECT tags

FROM posts

WHERE tags IS NULL

我们将前面例子中 exists 查询换成 missing 查询：

GET /my\_index/posts/\_search

{

"query" : {

"constant\_score" : {

"filter": {

"missing" : { "field" : "tags" }

}

}

}

}

**拷贝为 CURL**[在 SENSE 中查看](http://localhost:5601/app/sense/?load_from=https://www.elastic.co/guide/cn/elasticsearch/guide/current/snippets/080_Structured_Search/30_Exists_missing.json)

按照期望的那样，我们得到 3 和 4 两个文档（这两个文档的 tags 字段没有实际值）：

"hits" : [

{

"\_id" : "3",

"\_score" : 1.0,

"\_source" : { "other\_field" : "some data" }

},

{

"\_id" : "4",

"\_score" : 1.0,

"\_source" : { "tags" : null }

}

]

**当 null 的意思是 null**

有时候我们需要区分一个字段是没有值，还是它已被显式的设置成了 null 。在之前例子中，我们看到的默认的行为是无法做到这点的；数据被丢失了。不过幸运的是，我们可以选择将显式的 null 值替换成我们指定 占位符（placeholder） 。

在为字符串（string）、数字（numeric）、布尔值（Boolean）或日期（date）字段指定映射时，同样可以为之设置 null\_value 空值，用以处理显式 null 值的情况。不过即使如此，还是会将一个没有值的字段从倒排索引中排除。

当选择合适的 null\_value 空值的时候，需要保证以下几点：

* 它会匹配字段的类型，我们不能为一个 date 日期字段设置字符串类型的 null\_value 。
* 它必须与普通值不一样，这可以避免把实际值当成 null 空的情况。

#### 对象上的存在与缺失[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/080_Structured_Search/30_existsmissing.asciidoc)

不仅可以过滤核心类型， exists and missing 查询 还可以处理一个对象的内部字段。以下面文档为例：

{

"name" : {

"first" : "John",

"last" : "Smith"

}

}

我们不仅可以检查 name.first 和 name.last 的存在性，也可以检查 name ，不过在 [映射](https://www.elastic.co/guide/cn/elasticsearch/guide/current/mapping.html) 中，如上对象的内部是个扁平的字段与值（field-value）的简单键值结构，类似下面这样：

{

"name.first" : "John",

"name.last" : "Smith"

}

那么我们如何用 exists 或 missing 查询 name 字段呢？ name 字段并不真实存在于倒排索引中。

原因是当我们执行下面这个过滤的时候：

{

"exists" : { "field" : "name" }

}

实际执行的是：

{

"bool": {

"should": [

{ "exists": { "field": "name.first" }},

{ "exists": { "field": "name.last" }}

]

}

}

这也就意味着，如果 first 和 last 都是空，那么 name 这个命名空间才会被认为不存在。

### 关于缓存[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/080_Structured_Search/40_bitsets.asciidoc)

在本章前面（[过滤器的内部操作](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_finding_exact_values.html#_internal_filter_operation)）中，我们已经简单介绍了过滤器是如何计算的。 其核心实际是采用一个 bitset 记录与过滤器匹配的文档。Elasticsearch 积极地把这些 bitset 缓存起来以备随后使用。一旦缓存成功，bitset 可以复用 任何 已使用过的相同过滤器，而无需再次计算整个过滤器。

这些 bitsets 缓存是“智能”的：它们以增量方式更新。当我们索引新文档时，只需将那些新文档加入已有 bitset，而不是对整个缓存一遍又一遍的重复计算。和系统其他部分一样，过滤器是实时的，我们无需担心缓存过期问题。

#### 独立的过滤器缓存[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/080_Structured_Search/40_bitsets.asciidoc)

属于一个查询组件的 bitsets 是独立于它所属搜索请求其他部分的。这就意味着，一旦被缓存，一个查询可以被用作多个搜索请求。bitsets 并不依赖于它所存在的查询上下文。这样使得缓存可以加速查询中经常使用的部分，从而降低较少、易变的部分所带来的消耗。

同样，如果单个请求重用相同的非评分查询，它缓存的 bitset 可以被单个搜索里的所有实例所重用。

让我们看看下面例子中的查询，它查找满足以下任意一个条件的电子邮件：

* 在收件箱中，且没有被读过的
* 不在 收件箱中，但被标注重要的

GET /inbox/emails/\_search

{

"query": {

"constant\_score": {

"filter": {

"bool": {

"should": [

{ "bool": {

"must": [

{ "term": { "folder": "inbox" }}, https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

{ "term": { "read": false }}

]

}},

{ "bool": {

"must\_not": {

"term": { "folder": "inbox" } https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

},

"must": {

"term": { "important": true }

}

}}

]

}

}

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/filter-caching.html#CO55-1) [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/filter-caching.html#CO55-2) | 两个过滤器是相同的，所以会使用同一 bitset 。 |

尽管其中一个收件箱的条件是 must 语句，另一个是 must\_not 语句，但他们两者是完全相同的。这意味着在第一个语句执行后， bitset 就会被计算然后缓存起来供另一个使用。当再次执行这个查询时，收件箱的这个过滤器已经被缓存了，所以两个语句都会使用已缓存的 bitset 。

这点与查询表达式（query DSL）的可组合性结合得很好。它易被移动到表达式的任何地方，或者在同一查询中的多个位置复用。这不仅能方便开发者，而且对提升性能有直接的益处。

#### 自动缓存行为[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/080_Structured_Search/40_bitsets.asciidoc)

在 Elasticsearch 的较早版本中，默认的行为是缓存一切可以缓存的对象。这也通常意味着系统缓存 bitsets 太富侵略性，从而因为清理缓存带来性能压力。不仅如此，尽管很多过滤器都很容易被评价，但本质上是慢于缓存的（以及从缓存中复用）。缓存这些过滤器的意义不大，因为可以简单地再次执行过滤器。

检查一个倒排是非常快的，然后绝大多数查询组件却很少使用它。例如 term 过滤字段 "user\_id" ：如果有上百万的用户，每个具体的用户 ID 出现的概率都很小。那么为这个过滤器缓存 bitsets 就不是很合算，因为缓存的结果很可能在重用之前就被剔除了。

这种缓存的扰动对性能有着严重的影响。更严重的是，它让开发者难以区分有良好表现的缓存以及无用缓存。

为了解决问题，Elasticsearch 会基于使用频次自动缓存查询。如果一个非评分查询在最近的 256 词查询中被使用过（次数取决于查询类型），那么这个查询就会作为缓存的候选。但是，并不是所有的片段都能保证缓存 bitset 。只有那些文档数量超过 10,000 （或超过总文档数量的 3% )才会缓存 bitset 。因为小的片段可以很快的进行搜索和合并，这里缓存的意义不大。

一旦缓存了，非评分计算的 bitset 会一直驻留在缓存中直到它被剔除。剔除规则是基于 LRU 的：一旦缓存满了，最近最少使用的过滤器会被剔除。

## 全文搜索[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/100_Full_Text_Search/00_Intro.asciidoc)

我们已经介绍了搜索结构化数据的简单应用示例，现在来探寻 全文搜索（full-text search） ：怎样在全文字段中搜索到最相关的文档。

全文搜索两个最重要的方面是：

相关性（Relevance）

它是评价查询与其结果间的相关程度，并根据这种相关程度对结果排名的一种能力，这种计算方式可以是 TF/IDF 方法（参见 [相关性的介绍](https://www.elastic.co/guide/cn/elasticsearch/guide/current/relevance-intro.html)）、地理位置邻近、模糊相似，或其他的某些算法。

分析（Analysis）

它是将文本块转换为有区别的、规范化的 token 的一个过程，（参见 [分析的介绍](https://www.elastic.co/guide/cn/elasticsearch/guide/current/analysis-intro.html)） 目的是为了（a）创建倒排索引以及（b）查询倒排索引。

一旦谈论相关性或分析这两个方面的问题时，我们所处的语境是关于查询的而不是过滤。

### 基于词项与基于全文[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/100_Full_Text_Search/00_Intro.asciidoc)

所有查询会或多或少的执行相关度计算，但不是所有查询都有分析阶段。 和一些特殊的完全不会对文本进行操作的查询（如 bool 或 function\_score ）不同，文本查询可以划分成两大家族：

基于词项的查询

如 term 或 fuzzy 这样的底层查询不需要分析阶段，它们对单个词项进行操作。用 term 查询词项 Foo 只要在倒排索引中查找 准确词项 ，并且用 TF/IDF 算法为每个包含该词项的文档计算相关度评分 \_score 。

记住 term 查询只对倒排索引的词项精确匹配，这点很重要，它不会对词的多样性进行处理（如， foo或 FOO ）。这里，无须考虑词项是如何存入索引的。如果是将 ["Foo","Bar"] 索引存入一个不分析的（ not\_analyzed ）包含精确值的字段，或者将 Foo Bar 索引到一个带有 whitespace 空格分析器的字段，两者的结果都会是在倒排索引中有 Foo 和 Bar 这两个词。

基于全文的查询

像 match 或 query\_string 这样的查询是高层查询，它们了解字段映射的信息：

* 如果查询 日期（date） 或 整数（integer） 字段，它们会将查询字符串分别作为日期或整数对待。
* 如果查询一个（ not\_analyzed ）未分析的精确值字符串字段， 它们会将整个查询字符串作为单个词项对待。
* 但如果要查询一个（ analyzed ）已分析的全文字段， 它们会先将查询字符串传递到一个合适的分析器，然后生成一个供查询的词项列表。

一旦组成了词项列表，这个查询会对每个词项逐一执行底层的查询，再将结果合并，然后为每个文档生成一个最终的相关度评分。

我们将会在随后章节中详细讨论这个过程。

我们很少直接使用基于词项的搜索，通常情况下都是对全文进行查询，而非单个词项，这只需要简单的执行一个高层全文查询（进而在高层查询内部会以基于词项的底层查询完成搜索）。

注意

当我们想要查询一个具有精确值的 not\_analyzed 未分析字段之前， 需要考虑，是否真的采用评分查询，或者非评分查询会更好。

单词项查询通常可以用是、非这种二元问题表示，所以更适合用过滤， 而且这样做可以有效利用[缓存](https://www.elastic.co/guide/cn/elasticsearch/guide/current/filter-caching.html)：

GET /\_search

{

"query": {

"constant\_score": {

"filter": {

"term": { "gender": "female" }

}

}

}

}

### 匹配查询[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/100_Full_Text_Search/05_Match_query.asciidoc)

匹配查询 match 是个 核心 查询。无论需要查询什么字段， match 查询都应该会是首选的查询方式。 它是一个高级 全文查询 ，这表示它既能处理全文字段，又能处理精确字段。

这就是说， match 查询主要的应用场景就是进行全文搜索，我们以下面一个简单例子来说明全文搜索是如何工作的：

#### 索引一些数据[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/100_Full_Text_Search/05_Match_query.asciidoc)

首先，我们使用 [bulk API](https://www.elastic.co/guide/cn/elasticsearch/guide/current/bulk.html) 创建一些新的文档和索引：

DELETE /my\_index https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

PUT /my\_index

{ "settings": { "number\_of\_shards": 1 }} https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

POST /my\_index/my\_type/\_bulk

{ "index": { "\_id": 1 }}

{ "title": "The quick brown fox" }

{ "index": { "\_id": 2 }}

{ "title": "The quick brown fox jumps over the lazy dog" }

{ "index": { "\_id": 3 }}

{ "title": "The quick brown fox jumps over the quick dog" }

{ "index": { "\_id": 4 }}

{ "title": "Brown fox brown dog" }

**拷贝为 CURL**[在 SENSE 中查看](http://localhost:5601/app/sense/?load_from=https://www.elastic.co/guide/cn/elasticsearch/guide/current/snippets/100_Full_Text_Search/05_Match_query.json)

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/match-query.html#CO56-1) | 删除已有的索引。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/match-query.html#CO56-2) | 稍后，我们会在 [被破坏的相关性！](https://www.elastic.co/guide/cn/elasticsearch/guide/current/relevance-is-broken.html) 中解释只为这个索引分配一个主分片的原因。 |

#### 单个词查询[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/100_Full_Text_Search/05_Match_query.asciidoc)

我们用第一个示例来解释使用 match 查询搜索全文字段中的单个词：

GET /my\_index/my\_type/\_search

{

"query": {

"match": {

"title": "QUICK!"

}

}

}

**拷贝为 CURL**[在 SENSE 中查看](http://localhost:5601/app/sense/?load_from=https://www.elastic.co/guide/cn/elasticsearch/guide/current/snippets/100_Full_Text_Search/05_Match_query.json)

Elasticsearch 执行上面这个 match 查询的步骤是：

1. 检查字段类型 。

标题 title 字段是一个 string 类型（ analyzed ）已分析的全文字段，这意味着查询字符串本身也应该被分析。

1. 分析查询字符串 。

将查询的字符串 QUICK! 传入标准分析器中，输出的结果是单个项 quick 。因为只有一个单词项，所以 match 查询执行的是单个底层 term 查询。

1. 查找匹配文档 。

用 term 查询在倒排索引中查找 quick 然后获取一组包含该项的文档，本例的结果是文档：1、2 和 3 。

1. 为每个文档评分 。

用 term 查询计算每个文档相关度评分 \_score ，这是种将 词频（term frequency，即词 quick 在相关文档的 title 字段中出现的频率）和反向文档频率（inverse document frequency，即词 quick 在所有文档的 title 字段中出现的频率），以及字段的长度（即字段越短相关度越高）相结合的计算方式。参见 [相关性的介绍](https://www.elastic.co/guide/cn/elasticsearch/guide/current/relevance-intro.html) 。

这个过程给我们以下（经缩减）结果：

"hits": [

{

"\_id": "1",

"\_score": 0.5, https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

"\_source": {

"title": "The quick brown fox"

}

},

{

"\_id": "3",

"\_score": 0.44194174, https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

"\_source": {

"title": "The quick brown fox jumps over the quick dog"

}

},

{

"\_id": "2",

"\_score": 0.3125, https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png

"\_source": {

"title": "The quick brown fox jumps over the lazy dog"

}

}

]

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/match-query.html#CO57-1) | 文档 1 最相关，因为它的 title 字段更短，即 quick 占据内容的一大部分。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/match-query.html#CO57-2) [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/match-query.html#CO57-3) | 文档 3 比 文档 2 更具相关性，因为在文档 2 中 quick 出现了两次。 |

### 多词查询[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/100_Full_Text_Search/10_Multi_word_queries.asciidoc)

如果我们一次只能搜索一个词，那么全文搜索就会不太灵活，幸运的是 match 查询让多词查询变得简单：

GET /my\_index/my\_type/\_search

{

"query": {

"match": {

"title": "BROWN DOG!"

}

}

}

**拷贝为 CURL**[在 SENSE 中查看](http://localhost:5601/app/sense/?load_from=https://www.elastic.co/guide/cn/elasticsearch/guide/current/snippets/100_Full_Text_Search/05_Match_query.json)

上面这个查询返回所有四个文档：

{

"hits": [

{

"\_id": "4",

"\_score": 0.73185337, https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

"\_source": {

"title": "Brown fox brown dog"

}

},

{

"\_id": "2",

"\_score": 0.47486103, https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

"\_source": {

"title": "The quick brown fox jumps over the lazy dog"

}

},

{

"\_id": "3",

"\_score": 0.47486103, https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png

"\_source": {

"title": "The quick brown fox jumps over the quick dog"

}

},

{

"\_id": "1",

"\_score": 0.11914785, https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/4.png

"\_source": {

"title": "The quick brown fox"

}

}

]

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/match-multi-word.html#CO58-1) | 文档 4 最相关，因为它包含词 "brown" 两次以及 "dog" 一次。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/match-multi-word.html#CO58-2) [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/match-multi-word.html#CO58-3) | 文档 2、3 同时包含 brown 和 dog 各一次，而且它们 title 字段的长度相同，所以具有相同的评分。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/4.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/match-multi-word.html#CO58-4) | 文档 1 也能匹配，尽管它只有 brown 没有 dog 。 |

因为 match 查询必须查找两个词（ ["brown","dog"] ），它在内部实际上先执行两次 term 查询，然后将两次查询的结果合并作为最终结果输出。为了做到这点，它将两个 term 查询包入一个 bool 查询中，详细信息见 [布尔查询](https://www.elastic.co/guide/cn/elasticsearch/guide/current/bool-query.html)。

以上示例告诉我们一个重要信息：即任何文档只要 title 字段里包含 指定词项中的至少一个词 就能匹配，被匹配的词项越多，文档就越相关。

#### 提高精度[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/100_Full_Text_Search/10_Multi_word_queries.asciidoc)

用 任意 查询词项匹配文档可能会导致结果中出现不相关的长尾。 这是种散弹式搜索。可能我们只想搜索包含 所有 词项的文档，也就是说，不去匹配 brown OR dog ，而通过匹配 brown AND dog 找到所有文档。

match 查询还可以接受 operator 操作符作为输入参数，默认情况下该操作符是 or 。我们可以将它修改成 and 让所有指定词项都必须匹配：

GET /my\_index/my\_type/\_search

{

"query": {

"match": {

"title": { https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

"query": "BROWN DOG!",

"operator": "and"

}

}

}

}

**拷贝为 CURL**[在 SENSE 中查看](http://localhost:5601/app/sense/?load_from=https://www.elastic.co/guide/cn/elasticsearch/guide/current/snippets/100_Full_Text_Search/05_Match_query.json)

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/match-multi-word.html#CO59-1) | match 查询的结构需要做稍许调整才能使用 operator 操作符参数。 |

这个查询可以把文档 1 排除在外，因为它只包含两个词项中的一个。

#### 控制精度[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/100_Full_Text_Search/10_Multi_word_queries.asciidoc)

在 所有 与 任意 间二选一有点过于非黑即白。 如果用户给定 5 个查询词项，想查找只包含其中 4 个的文档，该如何处理？将 operator 操作符参数设置成 and 只会将此文档排除。

有时候这正是我们期望的，但在全文搜索的大多数应用场景下，我们既想包含那些可能相关的文档，同时又排除那些不太相关的。换句话说，我们想要处于中间某种结果。

match 查询支持 minimum\_should\_match 最小匹配参数， 这让我们可以指定必须匹配的词项数用来表示一个文档是否相关。我们可以将其设置为某个具体数字，更常用的做法是将其设置为一个百分数，因为我们无法控制用户搜索时输入的单词数量：

GET /my\_index/my\_type/\_search

{

"query": {

"match": {

"title": {

"query": "quick brown dog",

"minimum\_should\_match": "75%"

}

}

}

}

**拷贝为 CURL**[在 SENSE 中查看](http://localhost:5601/app/sense/?load_from=https://www.elastic.co/guide/cn/elasticsearch/guide/current/snippets/100_Full_Text_Search/05_Match_query.json)

当给定百分比的时候， minimum\_should\_match 会做合适的事情：在之前三词项的示例中， 75% 会自动被截断成 66.6% ，即三个里面两个词。无论这个值设置成什么，至少包含一个词项的文档才会被认为是匹配的。

注意

参数 minimum\_should\_match 的设置非常灵活，可以根据用户输入词项的数目应用不同的规则。完整的信息参考文档<https://www.elastic.co/guide/en/elasticsearch/reference/5.6/query-dsl-minimum-should-match.html#query-dsl-minimum-should-match>

为了完全理解 match 是如何处理多词查询的，我们就需要查看如何使用 bool 查询将多个查询条件组合在一起。

### 组合查询[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/100_Full_Text_Search/15_Combining_queries.asciidoc)

在 [组合过滤器](https://www.elastic.co/guide/cn/elasticsearch/guide/current/combining-filters.html) 中，我们讨论过如何使用 bool 过滤器通过 and 、 or 和 not 逻辑组合将多个过滤器进行组合。在查询中， bool 查询有类似的功能，只有一个重要的区别。

过滤器做二元判断：文档是否应该出现在结果中？但查询更精妙，它除了决定一个文档是否应该被包括在结果中，还会计算文档的 相关程度 。

与过滤器一样， bool 查询也可以接受 must 、 must\_not 和 should 参数下的多个查询语句。比如：

GET /my\_index/my\_type/\_search

{

"query": {

"bool": {

"must": { "match": { "title": "quick" }},

"must\_not": { "match": { "title": "lazy" }},

"should": [

{ "match": { "title": "brown" }},

{ "match": { "title": "dog" }}

]

}

}

}

**拷贝为 CURL**[在 SENSE 中查看](http://localhost:5601/app/sense/?load_from=https://www.elastic.co/guide/cn/elasticsearch/guide/current/snippets/100_Full_Text_Search/15_Bool_query.json)

以上的查询结果返回 title 字段包含词项 quick 但不包含 lazy 的任意文档。目前为止，这与 bool 过滤器的工作方式非常相似。

区别就在于两个 should 语句，也就是说：一个文档不必包含 brown 或 dog 这两个词项，但如果一旦包含，我们就认为它们 更相关 ：

{

"hits": [

{

"\_id": "3",

"\_score": 0.70134366, https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

"\_source": {

"title": "The quick brown fox jumps over the quick dog"

}

},

{

"\_id": "1",

"\_score": 0.3312608,

"\_source": {

"title": "The quick brown fox"

}

}

]

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/bool-query.html#CO60-1) | 文档 3 会比文档 1 有更高评分是因为它同时包含 brown 和 dog 。 |

#### 评分计算[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/100_Full_Text_Search/15_Combining_queries.asciidoc)

bool 查询会为每个文档计算相关度评分 \_score ， 再将所有匹配的 must 和 should 语句的分数 \_score求和，最后除以 must 和 should 语句的总数。

must\_not 语句不会影响评分； 它的作用只是将不相关的文档排除。

#### 控制精度[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/100_Full_Text_Search/15_Combining_queries.asciidoc)

所有 must 语句必须匹配，所有 must\_not 语句都必须不匹配，但有多少 should 语句应该匹配呢？ 默认情况下，没有 should 语句是必须匹配的，只有一个例外：那就是当没有 must 语句的时候，至少有一个 should 语句必须匹配。

就像我们能控制 [match 查询的精度](https://www.elastic.co/guide/cn/elasticsearch/guide/current/match-multi-word.html#match-precision) 一样，我们可以通过 minimum\_should\_match 参数控制需要匹配的 should 语句的数量， 它既可以是一个绝对的数字，又可以是个百分比：

GET /my\_index/my\_type/\_search

{

"query": {

"bool": {

"should": [

{ "match": { "title": "brown" }},

{ "match": { "title": "fox" }},

{ "match": { "title": "dog" }}

],

"minimum\_should\_match": 2 https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

}

}

}

**拷贝为 CURL**[在 SENSE 中查看](http://localhost:5601/app/sense/?load_from=https://www.elastic.co/guide/cn/elasticsearch/guide/current/snippets/100_Full_Text_Search/15_Bool_query.json)

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/bool-query.html#CO61-1) | 这也可以用百分比表示。 |

这个查询结果会将所有满足以下条件的文档返回： title 字段包含 "brown" AND "fox" 、 "brown" AND "dog" 或 "fox" AND "dog" 。如果有文档包含所有三个条件，它会比只包含两个的文档更相关

### 如何使用布尔匹配[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/100_Full_Text_Search/20_How_match_uses_bool.asciidoc)

目前为止，可能已经意识到[多词 match 查询](https://www.elastic.co/guide/cn/elasticsearch/guide/current/match-multi-word.html)只是简单地将生成的 term 查询包裹 在一个 bool 查询中。如果使用默认的 or 操作符，每个 term 查询都被当作 should 语句，这样就要求必须至少匹配一条语句。以下两个查询是等价的：

{

"match": { "title": "brown fox"}

}

{

"bool": {

"should": [

{ "term": { "title": "brown" }},

{ "term": { "title": "fox" }}

]

}

}

如果使用 and 操作符，所有的 term 查询都被当作 must 语句，所以 所有（all） 语句都必须匹配。以下两个查询是等价的：

{

"match": {

"title": {

"query": "brown fox",

"operator": "and"

}

}

}

{

"bool": {

"must": [

{ "term": { "title": "brown" }},

{ "term": { "title": "fox" }}

]

}

}

如果指定参数 minimum\_should\_match ，它可以通过 bool 查询直接传递，使以下两个查询等价：

{

"match": {

"title": {

"query": "quick brown fox",

"minimum\_should\_match": "75%"

}

}

}

{

"bool": {

"should": [

{ "term": { "title": "brown" }},

{ "term": { "title": "fox" }},

{ "term": { "title": "quick" }}

],

"minimum\_should\_match": 2 https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_how_match_uses_bool.html#CO62-1) | 因为只有三条语句，match 查询的参数 minimum\_should\_match 值 75% 会被截断成 2 。即三条 should 语句中至少有两条必须匹配。 |

当然，我们通常将这些查询用 match 查询来表示，但是如果了解 match 内部的工作原理，我们就能根据自己的需要来控制查询过程。有些时候单个 match 查询无法满足需求，比如为某些查询条件分配更高的权重。我们会在下一小节中看到这个例子。

### 查询语句提升权重[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/100_Full_Text_Search/25_Boosting_clauses.asciidoc)

当然 bool 查询不仅限于组合简单的单个词 match 查询， 它可以组合任意其他的查询，以及其他 bool 查询。 普遍的用法是通过汇总多个独立查询的分数，从而达到为每个文档微调其相关度评分 \_score 的目的。

假设想要查询关于 “full-text search（全文搜索）” 的文档， 但我们希望为提及 “Elasticsearch” 或 “Lucene” 的文档给予更高的 权重 ，这里 更高权重 是指如果文档中出现 “Elasticsearch” 或 “Lucene” ，它们会比没有的出现这些词的文档获得更高的相关度评分 \_score ，也就是说，它们会出现在结果集的更上面。

一个简单的 bool 查询 允许我们写出如下这种非常复杂的逻辑：

GET /\_search

{

"query": {

"bool": {

"must": {

"match": {

"content": { https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

"query": "full text search",

"operator": "and"

}

}

},

"should": [ https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

{ "match": { "content": "Elasticsearch" }},

{ "match": { "content": "Lucene" }}

]

}

}

}

**拷贝为 CURL**[在 SENSE 中查看](http://localhost:5601/app/sense/?load_from=https://www.elastic.co/guide/cn/elasticsearch/guide/current/snippets/100_Full_Text_Search/25_Boost.json)

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_boosting_query_clauses.html#CO63-1) | content 字段必须包含 full 、 text 和 search 所有三个词。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_boosting_query_clauses.html#CO63-2) | 如果 content 字段也包含 Elasticsearch 或 Lucene ，文档会获得更高的评分 \_score 。 |

should 语句匹配得越多表示文档的相关度越高。目前为止还挺好。

但是如果我们想让包含 Lucene 的有更高的权重，并且包含 Elasticsearch 的语句比 Lucene 的权重更高，该如何处理?

我们可以通过指定 boost 来控制任何查询语句的相对的权重， boost 的默认值为 1 ，大于 1 会提升一个语句的相对权重。所以下面重写之前的查询：

GET /\_search

{

"query": {

"bool": {

"must": {

"match": { https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

"content": {

"query": "full text search",

"operator": "and"

}

}

},

"should": [

{ "match": {

"content": {

"query": "Elasticsearch",

"boost": 3 https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

}

}},

{ "match": {

"content": {

"query": "Lucene",

"boost": 2 https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png

}

}}

]

}

}

}

**拷贝为 CURL**[在 SENSE 中查看](http://localhost:5601/app/sense/?load_from=https://www.elastic.co/guide/cn/elasticsearch/guide/current/snippets/100_Full_Text_Search/25_Boost.json)

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_boosting_query_clauses.html#CO64-1) | 这些语句使用默认的 boost 值 1 。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_boosting_query_clauses.html#CO64-2) | 这条语句更为重要，因为它有最高的 boost 值。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_boosting_query_clauses.html#CO64-3) | 这条语句比使用默认值的更重要，但它的重要性不及 Elasticsearch 语句。 |

注意

boost 参数被用来提升一个语句的相对权重（ boost 值大于 1 ）或降低相对权重（ boost值处于 0 到 1 之间），但是这种提升或降低并不是线性的，换句话说，如果一个 boost 值为 2 ，并不能获得两倍的评分 \_score 。

相反，新的评分 \_score 会在应用权重提升之后被 归一化 ，每种类型的查询都有自己的归一算法，细节超出了本书的范围，所以不作介绍。简单的说，更高的 boost 值为我们带来更高的评分 \_score 。

如果不基于 TF/IDF 要实现自己的评分模型，我们就需要对权重提升的过程能有更多控制，可以使用 [function\_score 查询](https://www.elastic.co/guide/cn/elasticsearch/guide/current/function-score-query.html)操纵一个文档的权重提升方式而跳过归一化这一步骤。

更多的组合查询方式会在下章[多字段搜索](https://www.elastic.co/guide/cn/elasticsearch/guide/current/multi-field-search.html)中介绍，但在此之前，让我们先看另外一个重要的查询特性：文本分析（text analysis）。

### 控制分析[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/100_Full_Text_Search/30_Controlling_analysis.asciidoc)

查询只能查找倒排索引表中真实存在的项， 所以保证文档在索引时与查询字符串在搜索时应用相同的分析过程非常重要，这样查询的项才能够匹配倒排索引中的项。

尽管是在说 文档 ，不过分析器可以由每个字段决定。 每个字段都可以有不同的分析器，既可以通过配置为字段指定分析器，也可以使用更高层的类型（type）、索引（index）或节点（node）的默认配置。在索引时，一个字段值是根据配置或默认分析器分析的。

例如为 my\_index 新增一个字段：

PUT /my\_index/\_mapping/my\_type

{

"my\_type": {

"properties": {

"english\_title": {

"type": "string",

"analyzer": "english"

}

}

}

}

**拷贝为 CURL**[在 SENSE 中查看](http://localhost:5601/app/sense/?load_from=https://www.elastic.co/guide/cn/elasticsearch/guide/current/snippets/100_Full_Text_Search/30_Analysis.json)

现在我们就可以通过使用 analyze API 来分析单词 Foxes ，进而比较 english\_title 字段和 title 字段在索引时的分析结果：

GET /my\_index/\_analyze

{

"field": "my\_type.title", https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

"text": "Foxes"

}

GET /my\_index/\_analyze

{

"field": "my\_type.english\_title", https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

"text": "Foxes"

}

**拷贝为 CURL**[在 SENSE 中查看](http://localhost:5601/app/sense/?load_from=https://www.elastic.co/guide/cn/elasticsearch/guide/current/snippets/100_Full_Text_Search/30_Analysis.json)

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_controlling_analysis.html#CO65-1) | 字段 title ，使用默认的 standard 标准分析器，返回词项 foxes 。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_controlling_analysis.html#CO65-2) | 字段 english\_title ，使用 english 英语分析器，返回词项 fox 。 |

这意味着，如果使用底层 term 查询精确项 fox 时， english\_title 字段会匹配但 title 字段不会。

如同 match 查询这样的高层查询知道字段映射的关系，能为每个被查询的字段应用正确的分析器。 可以使用 validate-query API 查看这个行为：

GET /my\_index/my\_type/\_validate/query?explain

{

"query": {

"bool": {

"should": [

{ "match": { "title": "Foxes"}},

{ "match": { "english\_title": "Foxes"}}

]

}

}

}

**拷贝为 CURL**[在 SENSE 中查看](http://localhost:5601/app/sense/?load_from=https://www.elastic.co/guide/cn/elasticsearch/guide/current/snippets/100_Full_Text_Search/30_Analysis.json)

返回语句的 explanation 结果：

(title:foxes english\_title:fox)

match 查询为每个字段使用合适的分析器，以保证它在寻找每个项时都为该字段使用正确的格式。

#### 默认分析器[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/100_Full_Text_Search/30_Controlling_analysis.asciidoc)

虽然我们可以在字段层级指定分析器， 但是如果该层级没有指定任何的分析器，那么我们如何能确定这个字段使用的是哪个分析器呢？

分析器可以从三个层面进行定义：按字段（per-field）、按索引（per-index）或全局缺省（global default）。Elasticsearch 会按照以下顺序依次处理，直到它找到能够使用的分析器。索引时的顺序如下：

* 字段映射里定义的 analyzer ，否则
* 索引设置中名为 default 的分析器，默认为
* standard 标准分析器

在搜索时，顺序有些许不同：

* 查询自己定义的 analyzer ，否则
* 字段映射里定义的 analyzer ，否则
* 索引设置中名为 default 的分析器，默认为
* standard 标准分析器

有时，在索引时和搜索时使用不同的分析器是合理的。 我们可能要想为同义词建索引（例如，所有 quick出现的地方，同时也为 fast 、 rapid 和 speedy 创建索引）。但在搜索时，我们不需要搜索所有的同义词，取而代之的是寻找用户输入的单词是否是 quick 、 fast 、 rapid 或 speedy 。

为了区分，Elasticsearch 也支持一个可选的 search\_analyzer 映射，它仅会应用于搜索时（ analyzer 还用于索引时）。还有一个等价的 default\_search 映射，用以指定索引层的默认配置。

如果考虑到这些额外参数，一个搜索时的 完整 顺序会是下面这样：

* 查询自己定义的 analyzer ，否则
* 字段映射里定义的 search\_analyzer ，否则
* 字段映射里定义的 analyzer ，否则
* 索引设置中名为 default\_search 的分析器，默认为
* 索引设置中名为 default 的分析器，默认为
* standard 标准分析器

#### 分析器配置实践[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/100_Full_Text_Search/30_Controlling_analysis.asciidoc)

就可以配置分析器地方的数量而言是十分惊人的， 但是实际非常简单。

#### 保持简单[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/100_Full_Text_Search/30_Controlling_analysis.asciidoc)

多数情况下，会提前知道文档会包括哪些字段。最简单的途径就是在创建索引或者增加类型映射时，为每个全文字段设置分析器。这种方式尽管有点麻烦，但是它让我们可以清楚的看到每个字段每个分析器是如何设置的。

通常，多数字符串字段都是 not\_analyzed 精确值字段，比如标签（tag）或枚举（enum），而且更多的全文字段会使用默认的 standard 分析器或 english 或其他某种语言的分析器。这样只需要为少数一两个字段指定自定义分析：或许标题 title 字段需要以支持 输入即查找（find-as-you-type） 的方式进行索引。

可以在索引级别设置中，为绝大部分的字段设置你想指定的 default 默认分析器。然后在字段级别设置中，对某一两个字段配置需要指定的分析器。

注意

对于和时间相关的日志数据，通常的做法是每天自行创建索引，由于这种方式不是从头创建的索引，仍然可以用 [索引模板（Index Template）](https://www.elastic.co/guide/en/elasticsearch/reference/5.6/indices-templates.html) 为新建的索引指定配置和映射。

### 被破坏的相关度！[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/100_Full_Text_Search/35_Relevance_is_broken.asciidoc)

在讨论更复杂的 [多字段搜索](https://www.elastic.co/guide/cn/elasticsearch/guide/current/multi-field-search.html) 之前，让我们先快速解释一下为什么只在主分片上 [创建测试索引](https://www.elastic.co/guide/cn/elasticsearch/guide/current/match-query.html#match-test-data) 。

用户会时不时的抱怨无法按相关度排序并提供简短的重现步骤： 用户索引了一些文档，运行一个简单的查询，然后发现明显低相关度的结果出现在高相关度结果之上。

为了理解为什么会这样，可以设想，我们在两个主分片上创建了索引和总共 10 个文档，其中 6 个文档有单词 foo 。可能是分片 1 有其中 3 个 foo 文档，而分片 2 有其中另外 3 个文档，换句话说，所有文档是均匀分布存储的。

在 [什么是相关度？](https://www.elastic.co/guide/cn/elasticsearch/guide/current/relevance-intro.html)中，我们描述了 Elasticsearch 默认使用的相似度算法，这个算法叫做 词频/逆向文档频率 或 TF/IDF 。词频是计算某个词在当前被查询文档里某个字段中出现的频率，出现的频率越高，文档越相关。 逆向文档频率 将 某个词在索引内所有文档出现的百分数 考虑在内，出现的频率越高，它的权重就越低。

但是由于性能原因， Elasticsearch 不会计算索引内所有文档的 IDF 。 相反，每个分片会根据 该分片 内的所有文档计算一个本地 IDF 。

因为文档是均匀分布存储的，两个分片的 IDF 是相同的。相反，设想如果有 5 个 foo 文档存于分片 1 ，而第 6 个文档存于分片 2 ，在这种场景下， foo 在一个分片里非常普通（所以不那么重要），但是在另一个分片里非常出现很少（所以会显得更重要）。这些 IDF 之间的差异会导致不正确的结果。

在实际应用中，这并不是一个问题，本地和全局的 IDF 的差异会随着索引里文档数的增多渐渐消失，在真实世界的数据量下，局部的 IDF 会被迅速均化，所以上述问题并不是相关度被破坏所导致的，而是由于数据太少。

为了测试，我们可以通过两种方式解决这个问题。第一种是只在主分片上创建索引，正如 [match 查询](https://www.elastic.co/guide/cn/elasticsearch/guide/current/match-query.html) 里介绍的那样，如果只有一个分片，那么本地的 IDF 就是 全局的 IDF。

第二个方式就是在搜索请求后添加 ?search\_type=dfs\_query\_then\_fetch ， dfs 是指 分布式频率搜索（Distributed Frequency Search） ， 它告诉 Elasticsearch ，先分别获得每个分片本地的 IDF ，然后根据结果再计算整个索引的全局 IDF 。

提示

不要在生产环境上使用 dfs\_query\_then\_fetch 。完全没有必要。只要有足够的数据就能保证词频是均匀分布的。没有理由给每个查询额外加上 DFS 这步。

## 多字段搜索[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/110_Multi_Field_Search/00_Intro.asciidoc)

查询很少是简单一句话的 match 匹配查询。通常我们需要用相同或不同的字符串查询一个或多个字段，也就是说，需要对多个查询语句以及它们相关度评分进行合理的合并。

有时候或许我们正查找作者 Leo Tolstoy 写的一本名为 \_War and Peace\_（战争与和平）的书。或许我们正用 “minimum should match” （最少应该匹配）的方式在文档中对标题或页面内容进行搜索，或许我们正在搜索所有名字为 John Smith 的用户。

在本章，我们会介绍构造多语句搜索的工具及在特定场景下应该采用的解决方案

### 多字符串查询[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/110_Multi_Field_Search/05_Multiple_query_strings.asciidoc)

最简单的多字段查询可以将搜索项映射到具体的字段。 如果我们知道 War and Peace 是标题，Leo Tolstoy 是作者，很容易就能把两个条件用 match 语句表示， 并将它们用 [bool 查询](https://www.elastic.co/guide/cn/elasticsearch/guide/current/bool-query.html) 组合起来：

GET /\_search

{

"query": {

"bool": {

"should": [

{ "match": { "title": "War and Peace" }},

{ "match": { "author": "Leo Tolstoy" }}

]

}

}

}

**拷贝为 CURL**[在 SENSE 中查看](http://localhost:5601/app/sense/?load_from=https://www.elastic.co/guide/cn/elasticsearch/guide/current/snippets/110_Multi_Field_Search/05_Multiple_query_strings.json)

bool 查询采取 more-matches-is-better 匹配越多越好的方式，所以每条 match 语句的评分结果会被加在一起，从而为每个文档提供最终的分数 \_score 。能与两条语句同时匹配的文档比只与一条语句匹配的文档得分要高。

当然，并不是只能使用 match 语句：可以用 bool 查询来包裹组合任意其他类型的查询， 甚至包括其他的 bool 查询。我们可以在上面的示例中添加一条语句来指定译者版本的偏好：

GET /\_search

{

"query": {

"bool": {

"should": [

{ "match": { "title": "War and Peace" }},

{ "match": { "author": "Leo Tolstoy" }},

{ "bool": {

"should": [

{ "match": { "translator": "Constance Garnett" }},

{ "match": { "translator": "Louise Maude" }}

]

}}

]

}

}

}

**拷贝为 CURL**[在 SENSE 中查看](http://localhost:5601/app/sense/?load_from=https://www.elastic.co/guide/cn/elasticsearch/guide/current/snippets/110_Multi_Field_Search/05_Multiple_query_strings.json)

为什么将译者条件语句放入另一个独立的 bool 查询中呢？所有的四个 match 查询都是 should 语句，所以为什么不将 translator 语句与其他如 title 、 author 这样的语句放在同一层呢？

答案在于评分的计算方式。 bool 查询运行每个 match 查询，再把评分加在一起，然后将结果与所有匹配的语句数量相乘，最后除以所有的语句数量。处于同一层的每条语句具有相同的权重。在前面这个例子中，包含 translator 语句的 bool 查询，只占总评分的三分之一。如果将 translator 语句与 title 和 author 两条语句放入同一层，那么 title 和 author 语句只贡献四分之一评分。

#### 语句的优先级[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/110_Multi_Field_Search/05_Multiple_query_strings.asciidoc)

前例中每条语句贡献三分之一评分的这种方式可能并不是我们想要的， 我们可能对 title 和 author 两条语句更感兴趣，这样就需要调整查询，使 title 和 author 语句相对来说更重要。

在武器库中，最容易使用的就是 boost 参数。为了提升 title 和 author 字段的权重， 为它们分配的 boost 值大于 1 ：

GET /\_search

{

"query": {

"bool": {

"should": [

{ "match": { https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

"title": {

"query": "War and Peace",

"boost": 2

}}},

{ "match": { https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

"author": {

"query": "Leo Tolstoy",

"boost": 2

}}},

{ "bool": { https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png

"should": [

{ "match": { "translator": "Constance Garnett" }},

{ "match": { "translator": "Louise Maude" }}

]

}}

]

}

}

}

**拷贝为 CURL**[在 SENSE 中查看](http://localhost:5601/app/sense/?load_from=https://www.elastic.co/guide/cn/elasticsearch/guide/current/snippets/110_Multi_Field_Search/05_Multiple_query_strings.json)

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/multi-query-strings.html#CO66-1) [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/multi-query-strings.html#CO66-2) | title 和 author 语句的 boost 值为 2 。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/multi-query-strings.html#CO66-3) | 嵌套 bool 语句默认的 boost 值为 1 。 |

要获取 boost 参数 “最佳” 值，较为简单的方式就是不断试错：设定 boost 值，运行测试查询，如此反复。 boost 值比较合理的区间处于 1 到 10 之间，当然也有可能是 15 。如果为 boost 指定比这更高的值，将不会对最终的评分结果产生更大影响，因为评分是被 [归一化的（normalized）](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_boosting_query_clauses.html#boost-normalization) 。

### 单字符串查询[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/110_Multi_Field_Search/10_Single_query_string.asciidoc)

bool 查询是多语句查询的主干。 它的适用场景很多，特别是当需要将不同查询字符串映射到不同字段的时候。

问题在于，目前有些用户期望将所有的搜索项堆积到单个字段中，并期望应用程序能为他们提供正确的结果。有意思的是多字段搜索的表单通常被称为 高级查询 （Advanced Search） —— 只是因为它对用户而言是高级的，而多字段搜索的实现却非常简单。

对于多词（multiword）、多字段（multifield）查询来说，不存在简单的 万能 方案。为了获得最好结果，需要 了解我们的数据 ，并了解如何使用合适的工具。

#### 了解我们的数据[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/110_Multi_Field_Search/10_Single_query_string.asciidoc)

当用户输入了单个字符串查询的时候，通常会遇到以下三种情形：

最佳字段

当搜索词语具体概念的时候，比如 “brown fox” ，词组比各自独立的单词更有意义。像 title 和 body 这样的字段，尽管它们之间是相关的，但同时又彼此相互竞争。文档在 相同字段 中包含的词越多越好，评分也来自于 最匹配字段 。

多数字段

为了对相关度进行微调，常用的一个技术就是将相同的数据索引到不同的字段，它们各自具有独立的分析链。

主字段可能包括它们的词源、同义词以及 变音词 或口音词，被用来匹配尽可能多的文档。

相同的文本被索引到其他字段，以提供更精确的匹配。一个字段可以包括未经词干提取过的原词，另一个字段包括其他词源、口音，还有一个字段可以提供 [词语相似性](https://www.elastic.co/guide/cn/elasticsearch/guide/current/proximity-matching.html) 信息的瓦片词（shingles）。

其他字段是作为匹配每个文档时提高相关度评分的 信号 ， 匹配字段越多 则越好。

混合字段

对于某些实体，我们需要在多个字段中确定其信息，单个字段都只能作为整体的一部分：

* Person： first\_name 和 last\_name （人：名和姓）
* Book： title 、 author 和 description （书：标题、作者、描述）
* Address： street 、 city 、 country 和 postcode （地址：街道、市、国家和邮政编码）

在这种情况下，我们希望在 任何 这些列出的字段中找到尽可能多的词，这有如在一个大字段中进行搜索，这个大字段包括了所有列出的字段。

上述所有都是多词、多字段查询，但每个具体查询都要求使用不同策略。本章后面的部分，我们会依次介绍每个策略。

### 最佳字段[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/110_Multi_Field_Search/15_Best_field.asciidoc)

假设有个网站允许用户搜索博客的内容， 以下面两篇博客内容文档为例：

PUT /my\_index/my\_type/1

{

"title": "Quick brown rabbits",

"body": "Brown rabbits are commonly seen."

}

PUT /my\_index/my\_type/2

{

"title": "Keeping pets healthy",

"body": "My quick brown fox eats rabbits on a regular basis."

}

**拷贝为 CURL**[在 SENSE 中查看](http://localhost:5601/app/sense/?load_from=https://www.elastic.co/guide/cn/elasticsearch/guide/current/snippets/110_Multi_Field_Search/15_Best_fields.json)

用户输入词组 “Brown fox” 然后点击搜索按钮。事先，我们并不知道用户的搜索项是会在 title 还是在 body 字段中被找到，但是，用户很有可能是想搜索相关的词组。用肉眼判断，文档 2 的匹配度更高，因为它同时包括要查找的两个词：

现在运行以下 bool 查询：

{

"query": {

"bool": {

"should": [

{ "match": { "title": "Brown fox" }},

{ "match": { "body": "Brown fox" }}

]

}

}

}

**拷贝为 CURL**[在 SENSE 中查看](http://localhost:5601/app/sense/?load_from=https://www.elastic.co/guide/cn/elasticsearch/guide/current/snippets/110_Multi_Field_Search/15_Best_fields.json)

但是我们发现查询的结果是文档 1 的评分更高：

{

"hits": [

{

"\_id": "1",

"\_score": 0.14809652,

"\_source": {

"title": "Quick brown rabbits",

"body": "Brown rabbits are commonly seen."

}

},

{

"\_id": "2",

"\_score": 0.09256032,

"\_source": {

"title": "Keeping pets healthy",

"body": "My quick brown fox eats rabbits on a regular basis."

}

}

]

}

为了理解导致这样的原因， 需要回想一下 bool 是如何计算评分的：

1. 它会执行 should 语句中的两个查询。
2. 加和两个查询的评分。
3. 乘以匹配语句的总数。
4. 除以所有语句总数（这里为：2）。

文档 1 的两个字段都包含 brown 这个词，所以两个 match 语句都能成功匹配并且有一个评分。文档 2 的 body 字段同时包含 brown 和 fox 这两个词，但 title 字段没有包含任何词。这样， body 查询结果中的高分，加上 title 查询中的 0 分，然后乘以二分之一，就得到比文档 1 更低的整体评分。

在本例中， title 和 body 字段是相互竞争的关系，所以就需要找到单个 最佳匹配 的字段。

如果不是简单将每个字段的评分结果加在一起，而是将 最佳匹配 字段的评分作为查询的整体评分，结果会怎样？这样返回的结果可能是： 同时 包含 brown 和 fox 的单个字段比反复出现相同词语的多个不同字段有更高的相关度。

#### dis\_max 查询[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/110_Multi_Field_Search/15_Best_field.asciidoc)

不使用 bool 查询，可以使用 dis\_max 即分离 最大化查询（Disjunction Max Query） 。分离（Disjunction）的意思是 或（or） ，这与可以把结合（conjunction）理解成 与（and） 相对应。分离最大化查询（Disjunction Max Query）指的是： 将任何与任一查询匹配的文档作为结果返回，但只将最佳匹配的评分作为查询的评分结果返回 ：

{

"query": {

"dis\_max": {

"queries": [

{ "match": { "title": "Brown fox" }},

{ "match": { "body": "Brown fox" }}

]

}

}

}

**拷贝为 CURL**[在 SENSE 中查看](http://localhost:5601/app/sense/?load_from=https://www.elastic.co/guide/cn/elasticsearch/guide/current/snippets/110_Multi_Field_Search/15_Best_fields.json)

得到我们想要的结果为：

{

"hits": [

{

"\_id": "2",

"\_score": 0.21509302,

"\_source": {

"title": "Keeping pets healthy",

"body": "My quick brown fox eats rabbits on a regular basis."

}

},

{

"\_id": "1",

"\_score": 0.12713557,

"\_source": {

"title": "Quick brown rabbits",

"body": "Brown rabbits are commonly seen."

}

}

]

}

### 最佳字段查询调优[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/110_Multi_Field_Search/20_Tuning_best_field_queries.asciidoc)

当用户搜索 “quick pets” 时会发生什么呢？ 在前面的例子中，两个文档都包含词 quick ，但是只有文档 2 包含词 pets ，两个文档中都不具有同时包含 两个词 的 相同字段 。

如下，一个简单的 dis\_max 查询会采用单个最佳匹配字段， 而忽略其他的匹配：

{

"query": {

"dis\_max": {

"queries": [

{ "match": { "title": "Quick pets" }},

{ "match": { "body": "Quick pets" }}

]

}

}

}

**拷贝为 CURL**[在 SENSE 中查看](http://localhost:5601/app/sense/?load_from=https://www.elastic.co/guide/cn/elasticsearch/guide/current/snippets/110_Multi_Field_Search/15_Best_fields.json)

{

"hits": [

{

"\_id": "1",

"\_score": 0.12713557, https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

"\_source": {

"title": "Quick brown rabbits",

"body": "Brown rabbits are commonly seen."

}

},

{

"\_id": "2",

"\_score": 0.12713557, https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

"\_source": {

"title": "Keeping pets healthy",

"body": "My quick brown fox eats rabbits on a regular basis."

}

}

]

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_tuning_best_fields_queries.html#CO67-1) [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_tuning_best_fields_queries.html#CO67-2) | 注意两个评分是完全相同的。 |

我们可能期望同时匹配 title 和 body 字段的文档比只与一个字段匹配的文档的相关度更高，但事实并非如此，因为 dis\_max 查询只会简单地使用 单个 最佳匹配语句的评分 \_score 作为整体评分。

#### tie\_breaker 参数[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/110_Multi_Field_Search/20_Tuning_best_field_queries.asciidoc)

可以通过指定 tie\_breaker 这个参数将其他匹配语句的评分也考虑其中：

{

"query": {

"dis\_max": {

"queries": [

{ "match": { "title": "Quick pets" }},

{ "match": { "body": "Quick pets" }}

],

"tie\_breaker": 0.3

}

}

}

**拷贝为 CURL**[在 SENSE 中查看](http://localhost:5601/app/sense/?load_from=https://www.elastic.co/guide/cn/elasticsearch/guide/current/snippets/110_Multi_Field_Search/15_Best_fields.json)

结果如下：

{

"hits": [

{

"\_id": "2",

"\_score": 0.14757764, https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

"\_source": {

"title": "Keeping pets healthy",

"body": "My quick brown fox eats rabbits on a regular basis."

}

},

{

"\_id": "1",

"\_score": 0.124275915, https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

"\_source": {

"title": "Quick brown rabbits",

"body": "Brown rabbits are commonly seen."

}

}

]

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_tuning_best_fields_queries.html#CO68-1) [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_tuning_best_fields_queries.html#CO68-2) | 文档 2 的相关度比文档 1 略高。 |

tie\_breaker 参数提供了一种 dis\_max 和 bool 之间的折中选择，它的评分方式如下：

1. 获得最佳匹配语句的评分 \_score 。
2. 将其他匹配语句的评分结果与 tie\_breaker 相乘。
3. 对以上评分求和并规范化。

有了 tie\_breaker ，会考虑所有匹配语句，但最佳匹配语句依然占最终结果里的很大一部分。

注意

tie\_breaker 可以是 0 到 1 之间的浮点数，其中 0 代表使用 dis\_max 最佳匹配语句的普通逻辑， 1 表示所有匹配语句同等重要。最佳的精确值需要根据数据与查询调试得出，但是合理值应该与零接近（处于 0.1 - 0.4 之间），这样就不会颠覆 dis\_max 最佳匹配性质的根本。

### multi\_match 查询[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/110_Multi_Field_Search/25_Multi_match_query.asciidoc)

multi\_match 查询为能在多个字段上反复执行相同查询提供了一种便捷方式。

注意

multi\_match 多匹配查询的类型有多种，其中的三种恰巧与 [了解我们的数据](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_single_query_string.html#know-your-data) 中介绍的三个场景对应，即： best\_fields 、 most\_fields 和 cross\_fields （最佳字段、多数字段、跨字段）。

默认情况下，查询的类型是 best\_fields ， 这表示它会为每个字段生成一个 match 查询，然后将它们组合到 dis\_max 查询的内部，如下：

{

"dis\_max": {

"queries": [

{

"match": {

"title": {

"query": "Quick brown fox",

"minimum\_should\_match": "30%"

}

}

},

{

"match": {

"body": {

"query": "Quick brown fox",

"minimum\_should\_match": "30%"

}

}

},

],

"tie\_breaker": 0.3

}

}

上面这个查询用 multi\_match 重写成更简洁的形式：

{

"multi\_match": {

"query": "Quick brown fox",

"type": "best\_fields", https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

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

"tie\_breaker": 0.3,

"minimum\_should\_match": "30%" https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

}

}

**拷贝为 CURL**[在 SENSE 中查看](http://localhost:5601/app/sense/?load_from=https://www.elastic.co/guide/cn/elasticsearch/guide/current/snippets/110_Multi_Field_Search/25_Best_fields.json)

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/multi-match-query.html#CO69-1) | best\_fields 类型是默认值，可以不指定。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/multi-match-query.html#CO69-2) | 如 minimum\_should\_match 或 operator 这样的参数会被传递到生成的 match 查询中。 |

#### 查询字段名称的模糊匹配[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/110_Multi_Field_Search/25_Multi_match_query.asciidoc)

字段名称可以用模糊匹配的方式给出：任何与模糊模式正则匹配的字段都会被包括在搜索条件中， 例如可以使用以下方式同时匹配 book\_title 、 chapter\_title 和 section\_title （书名、章名、节名）这三个字段：

{

"multi\_match": {

"query": "Quick brown fox",

"fields": "\*\_title"

}

}

#### 提升单个字段的权重[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/110_Multi_Field_Search/25_Multi_match_query.asciidoc)

可以使用 ^ 字符语法为单个字段提升权重，在字段名称的末尾添加 ^boost ， 其中 boost 是一个浮点数：

{

"multi\_match": {

"query": "Quick brown fox",

"fields": [ "\*\_title", "chapter\_title^2" ] https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/multi-match-query.html#CO70-1) | chapter\_title 这个字段的 boost 值为 2 ，而其他两个字段 book\_title 和 section\_title 字段的默认 boost 值为 1 。 |

### 多数字段[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/110_Multi_Field_Search/30_Most_fields.asciidoc)

全文搜索被称作是 召回率（Recall） 与 精确率（Precision） 的战场： 召回率 ——返回所有的相关文档；精确率 ——不返回无关文档。目的是在结果的第一页中为用户呈现最为相关的文档。

为了提高召回率的效果，我们扩大搜索范围 ——不仅返回与用户搜索词精确匹配的文档，还会返回我们认为与查询相关的所有文档。如果一个用户搜索 “quick brown box” ，一个包含词语 fast foxes 的文档被认为是非常合理的返回结果。

如果包含词语 fast foxes 的文档是能找到的唯一相关文档，那么它会出现在结果列表的最上面，但是，如果有 100 个文档都出现了词语 quick brown fox ，那么这个包含词语 fast foxes 的文档当然会被认为是次相关的，它可能处于返回结果列表更下面的某个地方。当包含了很多潜在匹配之后，我们需要将最匹配的几个置于结果列表的顶部。

提高全文相关性精度的常用方式是为同一文本建立多种方式的索引， 每种方式都提供了一个不同的相关度信号 signal 。主字段会以尽可能多的形式的去匹配尽可能多的文档。举个例子，我们可以进行以下操作：

* 使用词干提取来索引 jumps 、 jumping 和 jumped 样的词，将 jump 作为它们的词根形式。这样即使用户搜索 jumped ，也还是能找到包含 jumping 的匹配的文档。
* 将同义词包括其中，如 jump 、 leap 和 hop 。
* 移除变音或口音词：如 ésta 、 está 和 esta 都会以无变音形式 esta 来索引。

尽管如此，如果我们有两个文档，其中一个包含词 jumped ，另一个包含词 jumping ，用户很可能期望前者能排的更高，因为它正好与输入的搜索条件一致。

为了达到目的，我们可以将相同的文本索引到其他字段从而提供更为精确的匹配。一个字段可能是为词干未提取过的版本，另一个字段可能是变音过的原始词，第三个可能使用 shingles 提供 [词语相似性](https://www.elastic.co/guide/cn/elasticsearch/guide/current/proximity-matching.html) 信息。这些附加的字段可以看成提高每个文档的相关度评分的信号 signals ，能匹配字段的越多越好。

一个文档如果与广度匹配的主字段相匹配，那么它会出现在结果列表中。如果文档同时又与 signal 信号字段匹配，那么它会获得额外加分，系统会提升它在结果列表中的位置。

我们会在本书稍后对同义词、词相似性、部分匹配以及其他潜在的信号进行讨论，但这里只使用词干已提取（stemmed）和未提取（unstemmed）的字段作为简单例子来说明这种技术。

#### 多字段映射[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/110_Multi_Field_Search/30_Most_fields.asciidoc)

首先要做的事情就是对我们的字段索引两次： 一次使用词干模式以及一次非词干模式。为了做到这点，采用 multifields 来实现，已经在 [multifields](https://www.elastic.co/guide/cn/elasticsearch/guide/current/multi-fields.html) 有所介绍：

DELETE /my\_index

PUT /my\_index

{

"settings": { "number\_of\_shards": 1 }, https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

"mappings": {

"my\_type": {

"properties": {

"title": { https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

"type": "string",

"analyzer": "english",

"fields": {

"std": { https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png

"type": "string",

"analyzer": "standard"

}

}

}

}

}

}

}

**拷贝为 CURL**[在 SENSE 中查看](http://localhost:5601/app/sense/?load_from=https://www.elastic.co/guide/cn/elasticsearch/guide/current/snippets/110_Multi_Field_Search/30_Most_fields.json)

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/most-fields.html#CO71-1) | 参考 [被破坏的相关度](https://www.elastic.co/guide/cn/elasticsearch/guide/current/relevance-is-broken.html). |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/most-fields.html#CO71-2) | title 字段使用 english 英语分析器来提取词干。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/most-fields.html#CO71-3) | title.std 字段使用 standard 标准分析器，所以没有词干提取。 |

接着索引一些文档：

PUT /my\_index/my\_type/1

{ "title": "My rabbit jumps" }

PUT /my\_index/my\_type/2

{ "title": "Jumping jack rabbits" }

**拷贝为 CURL**[在 SENSE 中查看](http://localhost:5601/app/sense/?load_from=https://www.elastic.co/guide/cn/elasticsearch/guide/current/snippets/110_Multi_Field_Search/30_Most_fields.json)

这里用一个简单 match 查询 title 标题字段是否包含 jumping rabbits （跳跃的兔子）：

GET /my\_index/\_search

{

"query": {

"match": {

"title": "jumping rabbits"

}

}

}

**拷贝为 CURL**[在 SENSE 中查看](http://localhost:5601/app/sense/?load_from=https://www.elastic.co/guide/cn/elasticsearch/guide/current/snippets/110_Multi_Field_Search/30_Most_fields.json)

因为有了 english 分析器，这个查询是在查找以 jump 和 rabbit 这两个被提取词的文档。两个文档的 title 字段都同时包括这两个词，所以两个文档得到的评分也相同：

{

"hits": [

{

"\_id": "1",

"\_score": 0.42039964,

"\_source": {

"title": "My rabbit jumps"

}

},

{

"\_id": "2",

"\_score": 0.42039964,

"\_source": {

"title": "Jumping jack rabbits"

}

}

]

}

如果只是查询 title.std 字段，那么只有文档 2 是匹配的。尽管如此，如果同时查询两个字段，然后使用 bool 查询将评分结果 合并 ，那么两个文档都是匹配的（ title 字段的作用），而且文档 2 的相关度评分更高（ title.std 字段的作用）：

GET /my\_index/\_search

{

"query": {

"multi\_match": {

"query": "jumping rabbits",

"type": "most\_fields", https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

"fields": [ "title", "title.std" ]

}

}

}

**拷贝为 CURL**[在 SENSE 中查看](http://localhost:5601/app/sense/?load_from=https://www.elastic.co/guide/cn/elasticsearch/guide/current/snippets/110_Multi_Field_Search/30_Most_fields.json)

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/most-fields.html#CO72-1) | 我们希望将所有匹配字段的评分合并起来，所以使用 most\_fields 类型。这让 multi\_match 查询用 bool 查询将两个字段语句包在里面，而不是使用 dis\_max 查询。 |

{

"hits": [

{

"\_id": "2",

"\_score": 0.8226396, https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

"\_source": {

"title": "Jumping jack rabbits"

}

},

{

"\_id": "1",

"\_score": 0.10741998, https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

"\_source": {

"title": "My rabbit jumps"

}

}

]

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/most-fields.html#CO73-1) [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/most-fields.html#CO73-2) | 文档 2 现在的评分要比文档 1 高。 |

用广度匹配字段 title 包括尽可能多的文档——以提升召回率——同时又使用字段 title.std 作为 信号将相关度更高的文档置于结果顶部。

每个字段对于最终评分的贡献可以通过自定义值 boost 来控制。比如，使 title 字段更为重要，这样同时也降低了其他信号字段的作用：

GET /my\_index/\_search

{

"query": {

"multi\_match": {

"query": "jumping rabbits",

"type": "most\_fields",

"fields": [ "title^10", "title.std" ] https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

}

}

}

**拷贝为 CURL**[在 SENSE 中查看](http://localhost:5601/app/sense/?load_from=https://www.elastic.co/guide/cn/elasticsearch/guide/current/snippets/110_Multi_Field_Search/30_Most_fields.json)

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/most-fields.html#CO74-1) | title 字段的 boost 的值为 10 使它比 title.std 更重要。 |

[«  多数字段](https://www.elastic.co/guide/cn/elasticsearch/guide/current/most-fields.html)    [字段中心式查询  »](https://www.elastic.co/guide/cn/elasticsearch/guide/current/field-centric.html)

### 跨字段实体搜索[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/110_Multi_Field_Search/35_Entity_search.asciidoc)

现在讨论一种普遍的搜索模式：跨字段实体搜索（cross-fields entity search）。 在如 person 、 product或 address （人、产品或地址）这样的实体中，需要使用多个字段来唯一标识它的信息。 person 实体可能是这样索引的：

{

"firstname": "Peter",

"lastname": "Smith"

}

或地址：

{

"street": "5 Poland Street",

"city": "London",

"country": "United Kingdom",

"postcode": "W1V 3DG"

}

这与之前描述的 [多字符串查询](https://www.elastic.co/guide/cn/elasticsearch/guide/current/multi-query-strings.html) 很像，但这存在着巨大的区别。在 [多字符串查询](https://www.elastic.co/guide/cn/elasticsearch/guide/current/multi-query-strings.html) 中，我们为每个字段使用不同的字符串，在本例中，我们想使用 单个 字符串在多个字段中进行搜索。

我们的用户可能想搜索 “Peter Smith” 这个人，或 “Poland Street W1V” 这个地址，这些词出现在不同的字段中，所以如果使用 dis\_max 或 best\_fields 查询去查找 单个 最佳匹配字段显然是个错误的方式。

#### 简单的方式[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/110_Multi_Field_Search/35_Entity_search.asciidoc)

依次查询每个字段并将每个字段的匹配评分结果相加，听起来真像是 bool 查询：

{

"query": {

"bool": {

"should": [

{ "match": { "street": "Poland Street W1V" }},

{ "match": { "city": "Poland Street W1V" }},

{ "match": { "country": "Poland Street W1V" }},

{ "match": { "postcode": "Poland Street W1V" }}

]

}

}

}

为每个字段重复查询字符串会使查询瞬间变得冗长，可以采用 multi\_match 查询， 将 type 设置成 most\_fields 然后告诉 Elasticsearch 合并所有匹配字段的评分：

{

"query": {

"multi\_match": {

"query": "Poland Street W1V",

"type": "most\_fields",

"fields": [ "street", "city", "country", "postcode" ]

}

}

}

#### most\_fields 方式的问题[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/110_Multi_Field_Search/35_Entity_search.asciidoc)

用 most\_fields 这种方式搜索也存在某些问题，这些问题并不会马上显现：

* 它是为多数字段匹配 任意 词设计的，而不是在 所有字段 中找到最匹配的。
* 它不能使用 operator 或 minimum\_should\_match 参数来降低次相关结果造成的长尾效应。
* 词频对于每个字段是不一样的，而且它们之间的相互影响会导致不好的排序结果。

### 字段中心式查询[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/110_Multi_Field_Search/40_Field_centric.asciidoc)

以上三个源于 most\_fields 的问题都因为它是 字段中心式（field-centric） 而不是 词中心式（term-centric） 的：当真正感兴趣的是匹配词的时候，它为我们查找的是最匹配的 字段 。

注意

best\_fields 类型也是字段中心式的， 它也存在类似的问题。

首先查看这些问题存在的原因，再想如何解决它们。

#### 问题 1 ：在多个字段中匹配相同的词[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/110_Multi_Field_Search/40_Field_centric.asciidoc)

回想一下 most\_fields 查询是如何执行的：Elasticsearch 为每个字段生成独立的 match 查询，再用 bool查询将他们包起来。

可以通过 validate-query API 查看：

GET /\_validate/query?explain

{

"query": {

"multi\_match": {

"query": "Poland Street W1V",

"type": "most\_fields",

"fields": [ "street", "city", "country", "postcode" ]

}

}

}

**拷贝为 CURL**[在 SENSE 中查看](http://localhost:5601/app/sense/?load_from=https://www.elastic.co/guide/cn/elasticsearch/guide/current/snippets/110_Multi_Field_Search/40_Entity_search_problems.json)

生成 explanation 解释：

(street:poland street:street street:w1v)

(city:poland city:street city:w1v)

(country:poland country:street country:w1v)

(postcode:poland postcode:street postcode:w1v)

可以发现， 两个 字段都与 poland 匹配的文档要比一个字段同时匹配 poland 与 street 文档的评分高。

#### 问题 2 ：剪掉长尾[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/110_Multi_Field_Search/40_Field_centric.asciidoc)

在 [匹配精度](https://www.elastic.co/guide/cn/elasticsearch/guide/current/match-multi-word.html#match-precision) 中，我们讨论过使用 and 操作符或设置 minimum\_should\_match 参数来消除结果中几乎不相关的长尾，或许可以尝试以下方式：

{

"query": {

"multi\_match": {

"query": "Poland Street W1V",

"type": "most\_fields",

"operator": "and", https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

"fields": [ "street", "city", "country", "postcode" ]

}

}

}

**拷贝为 CURL**[在 SENSE 中查看](http://localhost:5601/app/sense/?load_from=https://www.elastic.co/guide/cn/elasticsearch/guide/current/snippets/110_Multi_Field_Search/40_Entity_search_problems.json)

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/field-centric.html#CO75-1) | 所有词必须呈现。 |

但是对于 best\_fields 或 most\_fields 这些参数会在 match 查询生成时被传入，这个查询的 explanation 解释如下：

(+street:poland +street:street +street:w1v)

(+city:poland +city:street +city:w1v)

(+country:poland +country:street +country:w1v)

(+postcode:poland +postcode:street +postcode:w1v)

换句话说，使用 and 操作符要求所有词都必须存在于 相同字段 ，这显然是不对的！可能就不存在能与这个查询匹配的文档。

#### 问题 3 ：词频[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/110_Multi_Field_Search/40_Field_centric.asciidoc)

在 [什么是相关](https://www.elastic.co/guide/cn/elasticsearch/guide/current/relevance-intro.html) 中，我们解释过每个词默认使用 TF/IDF 相似度算法计算相关度评分：

词频

一个词在单个文档的某个字段中出现的频率越高，这个文档的相关度就越高。

逆向文档频率

一个词在所有文档某个字段索引中出现的频率越高，这个词的相关度就越低。

当搜索多个字段时，TF/IDF 会带来某些令人意外的结果。

想想用字段 first\_name 和 last\_name 查询 “Peter Smith” 的例子， Peter 是个平常的名 Smith 也是平常的姓，这两者都具有较低的 IDF 值。但当索引中有另外一个人的名字是 “Smith Williams” 时， Smith 作为名来说很不平常，以致它有一个较高的 IDF 值！

下面这个简单的查询可能会在结果中将 “Smith Williams” 置于 “Peter Smith” 之上，尽管事实上是第二个人比第一个人更为匹配。

{

"query": {

"multi\_match": {

"query": "Peter Smith",

"type": "most\_fields",

"fields": [ "\*\_name" ]

}

}

}

**拷贝为 CURL**[在 SENSE 中查看](http://localhost:5601/app/sense/?load_from=https://www.elastic.co/guide/cn/elasticsearch/guide/current/snippets/110_Multi_Field_Search/40_Bad_frequencies.json)

这里的问题是 smith 在名字段中具有高 IDF ，它会削弱 “Peter” 作为名和 “Smith” 作为姓时低 IDF 的所起作用。

#### 解决方案[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/110_Multi_Field_Search/40_Field_centric.asciidoc)

存在这些问题仅仅是因为我们在处理着多个字段，如果将所有这些字段组合成单个字段，问题就会消失。可以为 person 文档添加 full\_name 字段来解决这个问题：

{

"first\_name": "Peter",

"last\_name": "Smith",

"full\_name": "Peter Smith"

}

当查询 full\_name 字段时：

* 具有更多匹配词的文档会比只有一个重复匹配词的文档更重要。
* minimum\_should\_match 和 operator 参数会像期望那样工作。
* 姓和名的逆向文档频率被合并，所以 Smith 到底是作为姓还是作为名出现，都会变得无关紧要。

这么做当然是可行的，但我们并不太喜欢存储冗余数据。取而代之的是 Elasticsearch 可以提供两个解决方案——一个在索引时，而另一个是在搜索时——随后会讨论它们。

### 自定义 \_all 字段[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/110_Multi_Field_Search/45_Custom_all.asciidoc)

在 [all-field](https://www.elastic.co/guide/cn/elasticsearch/guide/current/root-object.html#all-field) 字段中，我们解释过 \_all 字段的索引方式是将所有其他字段的值作为一个大字符串索引的。然而这么做并不十分灵活，为了灵活我们可以给人名添加一个自定义 \_all 字段，再为地址添加另一个 \_all 字段。

Elasticsearch 在字段映射中为我们提供 copy\_to 参数来实现这个功能：

PUT /my\_index

{

"mappings": {

"person": {

"properties": {

"first\_name": {

"type": "string",

"copy\_to": "full\_name" https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

},

"last\_name": {

"type": "string",

"copy\_to": "full\_name" https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

},

"full\_name": {

"type": "string"

}

}

}

}

}

**拷贝为 CURL**[在 SENSE 中查看](http://localhost:5601/app/sense/?load_from=https://www.elastic.co/guide/cn/elasticsearch/guide/current/snippets/110_Multi_Field_Search/45_Custom_all.json)

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/custom-all.html#CO76-1) [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/custom-all.html#CO76-2) | first\_name 和 last\_name 字段中的值会被复制到 full\_name 字段。 |

有了这个映射，我们可以用 first\_name 来查询名，用 last\_name 来查询姓，或者直接使用 full\_name 查询整个姓名。

first\_name 和 last\_name 的映射并不影响 full\_name 如何被索引， full\_name 将两个字段的内容复制到本地，然后根据 full\_name 的映射自行索引。

警告

copy\_to 设置对[multi-field](https://www.elastic.co/guide/cn/elasticsearch/guide/current/multi-fields.html)无效。如果尝试这样配置映射，Elasticsearch 会抛异常。

为什么呢？多字段只是以不同方式简单索引“主”字段；它们没有自己的数据源。也就是说没有可供 copy\_to 到另一字段的数据源。

只要对“主”字段 copy\_to 就能轻而易举的达到相同的效果：

PUT /my\_index

{

"mappings": {

"person": {

"properties": {

"first\_name": {

"type": "string",

"copy\_to": "full\_name", https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

"fields": {

"raw": {

"type": "string",

"index": "not\_analyzed"

}

}

},

"full\_name": {

"type": "string"

}

}

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/custom-all.html#CO77-1) | copy\_to 是针对“主”字段，而不是多字段的 |

### cross-fields 跨字段查询[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/110_Multi_Field_Search/50_Cross_field.asciidoc)

自定义 \_all 的方式是一个好的解决方案，只需在索引文档前为其设置好映射。 不过， Elasticsearch 还在搜索时提供了相应的解决方案：使用 cross\_fields 类型进行 multi\_match 查询。 cross\_fields 使用词中心式（term-centric）的查询方式，这与 best\_fields 和 most\_fields 使用字段中心式（field-centric）的查询方式非常不同，它将所有字段当成一个大字段，并在 每个字段 中查找 每个词 。

为了说明字段中心式（field-centric）与词中心式（term-centric）这两种查询方式的不同， 先看看以下字段中心式的 most\_fields 查询的 explanation 解释：

GET /\_validate/query?explain

{

"query": {

"multi\_match": {

"query": "peter smith",

"type": "most\_fields",

"operator": "and", https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

"fields": [ "first\_name", "last\_name" ]

}

}

}

**拷贝为 CURL**[在 SENSE 中查看](http://localhost:5601/app/sense/?load_from=https://www.elastic.co/guide/cn/elasticsearch/guide/current/snippets/110_Multi_Field_Search/50_Cross_field.json)

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_cross_fields_queries.html#CO78-1) | 所有词都是必须的。 |

对于匹配的文档， peter 和 smith 都必须同时出现在相同字段中，要么是 first\_name 字段，要么 last\_name 字段：

(+first\_name:peter +first\_name:smith)

(+last\_name:peter +last\_name:smith)

词中心式 会使用以下逻辑：

+(first\_name:peter last\_name:peter)

+(first\_name:smith last\_name:smith)

换句话说，词 peter 和 smith 都必须出现，但是可以出现在任意字段中。

cross\_fields 类型首先分析查询字符串并生成一个词列表，然后它从所有字段中依次搜索每个词。这种不同的搜索方式很自然的解决了 [字段中心式](https://www.elastic.co/guide/cn/elasticsearch/guide/current/field-centric.html) 查询三个问题中的二个。剩下的问题是逆向文档频率不同。

幸运的是 cross\_fields 类型也能解决这个问题，通过 validate-query 可以看到：

GET /\_validate/query?explain

{

"query": {

"multi\_match": {

"query": "peter smith",

"type": "cross\_fields", https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

"operator": "and",

"fields": [ "first\_name", "last\_name" ]

}

}

}

**拷贝为 CURL**[在 SENSE 中查看](http://localhost:5601/app/sense/?load_from=https://www.elastic.co/guide/cn/elasticsearch/guide/current/snippets/110_Multi_Field_Search/50_Cross_field.json)

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_cross_fields_queries.html#CO79-1) | 用 cross\_fields 词中心式匹配。 |

它通过 混合 不同字段逆向索引文档频率的方式解决了词频的问题：

+blended("peter", fields: [first\_name, last\_name])

+blended("smith", fields: [first\_name, last\_name])

换句话说，它会同时在 first\_name 和 last\_name 两个字段中查找 smith 的 IDF ，然后用两者的最小值作为两个字段的 IDF 。结果实际上就是 smith 会被认为既是个平常的姓，也是平常的名。

注意

为了让 cross\_fields 查询以最优方式工作，所有的字段都须使用相同的分析器， 具有相同分析器的字段会被分组在一起作为混合字段使用。

如果包括了不同分析链的字段，它们会以 best\_fields 的相同方式被加入到查询结果中。例如：我们将 title 字段加到之前的查询中（假设它们使用的是不同的分析器）， explanation 的解释结果如下：

(+title:peter +title:smith)

(

+blended("peter", fields: [first\_name, last\_name])

+blended("smith", fields: [first\_name, last\_name])

)

当在使用 minimum\_should\_match 和 operator 参数时，这点尤为重要。

#### 按字段提高权重[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/110_Multi_Field_Search/50_Cross_field.asciidoc)

采用 cross\_fields 查询与 [自定义 \_all 字段](https://www.elastic.co/guide/cn/elasticsearch/guide/current/custom-all.html) 相比，其中一个优势就是它可以在搜索时为单个字段提升权重。

这对像 first\_name 和 last\_name 具有相同值的字段并不是必须的，但如果要用 title 和 description字段搜索图书，可能希望为 title 分配更多的权重，这同样可以使用前面介绍过的 ^ 符号语法来实现：

GET /books/\_search

{

"query": {

"multi\_match": {

"query": "peter smith",

"type": "cross\_fields",

"fields": [ "title^2", "description" ] https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_cross_fields_queries.html#CO80-1) | title 字段的权重提升值为 2 ， description 字段的权重提升值默认为 1 。 |

自定义单字段查询是否能够优于多字段查询，取决于在多字段查询与单字段自定义 \_all 之间代价的权衡，即哪种解决方案会带来更大的性能优化就选择哪一种。

### Exact-Value 精确值字段[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/110_Multi_Field_Search/55_Not_analyzed.asciidoc)

在结束多字段查询这个话题之前，我们最后要讨论的是精确值 not\_analyzed 未分析字段。 将 not\_analyzed 字段与 multi\_match 中 analyzed 字段混在一起没有多大用处。

原因可以通过查看查询的 explanation 解释得到，设想将 title 字段设置成 not\_analyzed ：

GET /\_validate/query?explain

{

"query": {

"multi\_match": {

"query": "peter smith",

"type": "cross\_fields",

"fields": [ "title", "first\_name", "last\_name" ]

}

}

}

**拷贝为 CURL**[在 SENSE 中查看](http://localhost:5601/app/sense/?load_from=https://www.elastic.co/guide/cn/elasticsearch/guide/current/snippets/110_Multi_Field_Search/55_Not_analyzed.json)

因为 title 字段是未分析过的，Elasticsearch 会将 “peter smith” 这个完整的字符串作为查询条件来搜索！

title:peter smith

(

blended("peter", fields: [first\_name, last\_name])

blended("smith", fields: [first\_name, last\_name])

)

显然这个项不在 title 的倒排索引中，所以需要在 multi\_match 查询中避免使用 not\_analyzed 字段。

## 近似匹配[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/120_Proximity_Matching/00_Intro.asciidoc)

使用 TF/IDF 的标准全文检索将文档或者文档中的字段作一大袋的词语处理。 match 查询可以告知我们这大袋子中是否包含查询的词条，但却无法告知词语之间的关系。

思考下面这几个句子的不同：

* Sue ate the alligator.
* The alligator ate Sue.
* Sue never goes anywhere without her alligator-skin purse.

用 match 搜索 sue alligator 上面的三个文档都会得到匹配，但它却不能确定这两个词是否只来自于一种语境，甚至都不能确定是否来自于同一个段落。

理解分词之间的关系是一个复杂的难题，我们也无法通过换一种查询方式去解决。但我们至少可以通过出现在彼此附近或者仅仅是彼此相邻的分词来判断一些似乎相关的分词。

每个文档可能都比我们上面这个例子要长： Sue 和 alligator 这两个词可能会分散在其他的段落文字中，我们可能会希望得到尽可能包含这两个词的文档，但我们也同样需要这些文档与分词有很高的相关度。

这就是短语匹配或者近似匹配的所属领域。

提示

在这一章节，我们还是使用在[match 查询](https://www.elastic.co/guide/cn/elasticsearch/guide/current/match-query.html#match-test-data)中使用过的文档作为例子。

### 短语匹配[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/120_Proximity_Matching/05_Phrase_matching.asciidoc)

就像 match 查询对于标准全文检索是一种最常用的查询一样，当你想找到彼此邻近搜索词的查询方法时，就会想到 match\_phrase 查询 。

GET /my\_index/my\_type/\_search

{

"query": {

"match\_phrase": {

"title": "quick brown fox"

}

}

}

**拷贝为 CURL**[在 SENSE 中查看](http://localhost:5601/app/sense/?load_from=https://www.elastic.co/guide/cn/elasticsearch/guide/current/snippets/120_Proximity_Matching/05_Match_phrase_query.json)

类似 match 查询， match\_phrase 查询首先将查询字符串解析成一个词项列表，然后对这些词项进行搜索，但只保留那些包含 全部 搜索词项，且 位置 与搜索词项相同的文档。 比如对于 quick fox 的短语搜索可能不会匹配到任何文档，因为没有文档包含的 quick 词之后紧跟着 fox 。

提示

match\_phrase 查询同样可写成一种类型为 phrase 的 match 查询:

"match": {

"title": {

"query": "quick brown fox",

"type": "phrase"

}

}

**拷贝为 CURL**[在 SENSE 中查看](http://localhost:5601/app/sense/?load_from=https://www.elastic.co/guide/cn/elasticsearch/guide/current/snippets/120_Proximity_Matching/05_Match_phrase_query.json)

#### 词项的位置[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/120_Proximity_Matching/05_Phrase_matching.asciidoc)

当一个字符串被分词后，这个分析器不但会 返回一个词项列表，而且还会返回各词项在原始字符串中的 位置 或者顺序关系：

GET /\_analyze?analyzer=standard

Quick brown fox

**拷贝为 CURL**[在 SENSE 中查看](http://localhost:5601/app/sense/?load_from=https://www.elastic.co/guide/cn/elasticsearch/guide/current/snippets/120_Proximity_Matching/05_Term_positions.json)

返回信息如下：

{

"tokens": [

{

"token": "quick",

"start\_offset": 0,

"end\_offset": 5,

"type": "<ALPHANUM>",

"position": 1 https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

},

{

"token": "brown",

"start\_offset": 6,

"end\_offset": 11,

"type": "<ALPHANUM>",

"position": 2 https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

},

{

"token": "fox",

"start\_offset": 12,

"end\_offset": 15,

"type": "<ALPHANUM>",

"position": 3 https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png

}

]

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/phrase-matching.html#CO81-1) [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/phrase-matching.html#CO81-2)[https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/phrase-matching.html#CO81-3) | position 代表各词项在原始字符串中的位置。 |

位置信息可以被存储在倒排索引中，因此 match\_phrase 查询这类对词语位置敏感的查询， 就可以利用位置信息去匹配包含所有查询词项，且各词项顺序也与我们搜索指定一致的文档，中间不夹杂其他词项。

#### 什么是短语[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/120_Proximity_Matching/05_Phrase_matching.asciidoc)

一个被认定为和短语 quick brown fox 匹配的文档，必须满足以下这些要求：

* quick 、 brown 和 fox 需要全部出现在域中。
* brown 的位置应该比 quick 的位置大 1 。
* fox 的位置应该比 quick 的位置大 2 。

如果以上任何一个选项不成立，则该文档不能认定为匹配。

提示

本质上来讲，match\_phrase 查询是利用一种低级别的 span 查询族（query family）去做词语位置敏感的匹配。 Span 查询是一种词项级别的查询，所以它们没有分词阶段；它们只对指定的词项进行精确搜索。

值得庆幸的是，match\_phrase 查询已经足够优秀，大多数人是不会直接使用 span 查询。 然而，在一些专业领域，例如专利检索，还是会采用这种低级别查询去执行非常具体而又精心构造的位置搜索。

### 混合起来[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/120_Proximity_Matching/10_Slop.asciidoc)

精确短语匹配 或许是过于严格了。也许我们想要包含 “quick brown fox” 的文档也能够匹配 “quick fox,” ， 尽管情形不完全相同。

我们能够通过使用 slop 参数将灵活度引入短语匹配中：

GET /my\_index/my\_type/\_search

{

"query": {

"match\_phrase": {

"title": {

"query": "quick fox",

"slop": 1

}

}

}

}

**拷贝为 CURL**[在 SENSE 中查看](http://localhost:5601/app/sense/?load_from=https://www.elastic.co/guide/cn/elasticsearch/guide/current/snippets/120_Proximity_Matching/10_Slop.json)

slop 参数告诉 match\_phrase 查询词条相隔多远时仍然能将文档视为匹配 。 相隔多远的意思是为了让查询和文档匹配你需要移动词条多少次？

我们以一个简单的例子开始吧。 为了让查询 quick fox 能匹配一个包含 quick brown fox 的文档， 我们需要 slop 的值为 1:

Pos 1 Pos 2 Pos 3

-----------------------------------------------

Doc: quick brown fox

-----------------------------------------------

Query: quick fox

Slop 1: quick ↳ fox

尽管在使用了 slop 短语匹配中所有的单词都需要出现， 但是这些单词也不必为了匹配而按相同的序列排列。 有了足够大的 slop 值， 单词就能按照任意顺序排列了。

为了使查询 fox quick 匹配我们的文档， 我们需要 slop 的值为 3:

Pos 1 Pos 2 Pos 3

-----------------------------------------------

Doc: quick brown fox

-----------------------------------------------

Query: fox quick

Slop 1: fox|quick ↵ https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

Slop 2: quick ↳ fox

Slop 3: quick ↳ fox

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/slop.html#CO82-1) | 注意 fox 和 quick 在这步中占据同样的位置。 因此将 fox quick 转换顺序成 quick fox 需要两步， 或者值为 2 的 slop 。 |

### 多值字段[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/120_Proximity_Matching/15_Multi_value_fields.asciidoc)

对多值字段使用短语匹配时会发生奇怪的事。 想象一下你索引这个文档:

PUT /my\_index/groups/1

{

"names": [ "John Abraham", "Lincoln Smith"]

}

**拷贝为 CURL**[在 SENSE 中查看](http://localhost:5601/app/sense/?load_from=https://www.elastic.co/guide/cn/elasticsearch/guide/current/snippets/120_Proximity_Matching/15_Multi_value_fields.json)

然后运行一个对 Abraham Lincoln 的短语查询:

GET /my\_index/groups/\_search

{

"query": {

"match\_phrase": {

"names": "Abraham Lincoln"

}

}

}

**拷贝为 CURL**[在 SENSE 中查看](http://localhost:5601/app/sense/?load_from=https://www.elastic.co/guide/cn/elasticsearch/guide/current/snippets/120_Proximity_Matching/15_Multi_value_fields.json)

令人惊讶的是， 即使 Abraham 和 Lincoln 在 names 数组里属于两个不同的人名， 我们的文档也匹配了查询。 这一切的原因在Elasticsearch数组的索引方式。

在分析 John Abraham 的时候， 产生了如下信息：

* Position 1: john
* Position 2: abraham

然后在分析 Lincoln Smith 的时候， 产生了：

* Position 3: lincoln
* Position 4: smith

换句话说， Elasticsearch对以上数组分析生成了与分析单个字符串 John Abraham Lincoln Smith 一样几乎完全相同的语汇单元。 我们的查询示例寻找相邻的 lincoln 和 abraham ， 而且这两个词条确实存在，并且它们俩正好相邻， 所以这个查询匹配了。

幸运的是， 在这样的情况下有一种叫做 position\_increment\_gap 的简单的解决方案， 它在字段映射中配置 。

DELETE /my\_index/groups/ https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

PUT /my\_index/\_mapping/groups https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

{

"properties": {

"names": {

"type": "string",

"position\_increment\_gap": 100

}

}

}

**拷贝为 CURL**[在 SENSE 中查看](http://localhost:5601/app/sense/?load_from=https://www.elastic.co/guide/cn/elasticsearch/guide/current/snippets/120_Proximity_Matching/15_Multi_value_fields.json)

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_multivalue_fields_2.html#CO83-1) | 首先删除映射 groups 以及这个类型内的所有文档。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_multivalue_fields_2.html#CO83-2) | 然后创建一个有正确值的新的映射 groups 。 |

position\_increment\_gap 设置告诉 Elasticsearch 应该为数组中每个新元素增加当前词条 position 的指定值。 所以现在当我们再索引 names 数组时，会产生如下的结果：

* Position 1: john
* Position 2: abraham
* Position 103: lincoln
* Position 104: smith

现在我们的短语查询可能无法匹配该文档因为 abraham 和 lincoln 之间的距离为 100 。 为了匹配这个文档你必须添加值为 100 的 slop 。

### 越近越好[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/120_Proximity_Matching/20_Scoring.asciidoc)

鉴于一个短语查询仅仅排除了不包含确切查询短语的文档， 而 邻近查询 — 一个 slop 大于 0— 的短语查询将查询词条的邻近度考虑到最终相关度 \_score 中。 通过设置一个像 50 或者 100 这样的高 slop 值, 你能够排除单词距离太远的文档， 但是也给予了那些单词临近的的文档更高的分数。

下列对 quick dog 的邻近查询匹配了同时包含 quick 和 dog 的文档， 但是也给了与 quick 和 dog 更加临近的文档更高的分数 ：

POST /my\_index/my\_type/\_search

{

"query": {

"match\_phrase": {

"title": {

"query": "quick dog",

"slop": 50 https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

}

}

}

}

**拷贝为 CURL**[在 SENSE 中查看](http://localhost:5601/app/sense/?load_from=https://www.elastic.co/guide/cn/elasticsearch/guide/current/snippets/120_Proximity_Matching/20_Scoring.json)

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_closer_is_better.html#CO84-1) | 注意高 slop 值。 |

{

"hits": [

{

"\_id": "3",

"\_score": 0.75, https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

"\_source": {

"title": "The quick brown fox jumps over the quick dog"

}

},

{

"\_id": "2",

"\_score": 0.28347334, https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

"\_source": {

"title": "The quick brown fox jumps over the lazy dog"

}

}

]

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_closer_is_better.html#CO85-1) | 分数较高因为 quick 和 dog 很接近 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_closer_is_better.html#CO85-2) | 分数较低因为 quick 和 dog 分开较远 |

### 使用邻近度提高相关度[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/120_Proximity_Matching/25_Relevance.asciidoc)

虽然邻近查询很有用， 但是所有词条都出现在文档的要求过于严格了。 我们讨论 [全文搜索](https://www.elastic.co/guide/cn/elasticsearch/guide/current/full-text-search.html) 一章的 [控制精度](https://www.elastic.co/guide/cn/elasticsearch/guide/current/match-multi-word.html#match-precision) 也是同样的问题： 如果七个词条中有六个匹配， 那么这个文档对用户而言就已经足够相关了， 但是 match\_phrase 查询可能会将它排除在外。

相比将使用邻近匹配作为绝对要求， 我们可以将它作为 信号— 使用， 作为许多潜在查询中的一个， 会对每个文档的最终分值做出贡献 (参考 [多数字段](https://www.elastic.co/guide/cn/elasticsearch/guide/current/most-fields.html))。

实际上我们想将多个查询的分数累计起来意味着我们应该用 bool 查询将它们合并。

我们可以将一个简单的 match 查询作为一个 must 子句。 这个查询将决定哪些文档需要被包含到结果集中。 我们可以用 minimum\_should\_match 参数去除长尾。 然后我们可以以 should 子句的形式添加更多特定查询。 每一个匹配成功的都会增加匹配文档的相关度。

GET /my\_index/my\_type/\_search

{

"query": {

"bool": {

"must": {

"match": { https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

"title": {

"query": "quick brown fox",

"minimum\_should\_match": "30%"

}

}

},

"should": {

"match\_phrase": { https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

"title": {

"query": "quick brown fox",

"slop": 50

}

}

}

}

}

}

**拷贝为 CURL**[在 SENSE 中查看](http://localhost:5601/app/sense/?load_from=https://www.elastic.co/guide/cn/elasticsearch/guide/current/snippets/120_Proximity_Matching/25_Relevance.json)

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/proximity-relevance.html#CO86-1) | must 子句从结果集中包含或者排除文档。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/proximity-relevance.html#CO86-2) | should 子句增加了匹配到文档的相关度评分。 |

我们当然可以在 should 子句里面添加其它的查询， 其中每一个查询只针对某一特定方面的相关度。

### 性能优化[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/120_Proximity_Matching/30_Performance.asciidoc)

短语查询和邻近查询都比简单的 query 查询代价更高 。 一个 match 查询仅仅是看词条是否存在于倒排索引中，而一个 match\_phrase 查询是必须计算并比较多个可能重复词项的位置。

[Lucene nightly benchmarks](http://people.apache.org/~mikemccand/lucenebench/) 表明一个简单的 term 查询比一个短语查询大约快 10 倍，比邻近查询(有 slop 的短语 查询)大约快 20 倍。当然，这个代价指的是在搜索时而不是索引时。

提示

通常，短语查询的额外成本并不像这些数字所暗示的那么吓人。事实上，性能上的差距只是证明一个简单的 term 查询有多快。标准全文数据的短语查询通常在几毫秒内完成，因此实际上都是完全可用，即使是在一个繁忙的集群上。

在某些特定病理案例下，短语查询可能成本太高了，但比较少见。一个典型例子就是DNA序列，在序列里很多同样的词项在很多位置重复出现。在这里使用高 slop 值会到导致位置计算大量增加。

那么我们应该如何限制短语查询和邻近近查询的性能消耗呢？一种有用的方法是减少需要通过短语查询检查的文档总数。

#### 结果集重新评分[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/120_Proximity_Matching/30_Performance.asciidoc)

在[先前的章节中](https://www.elastic.co/guide/cn/elasticsearch/guide/current/proximity-relevance.html) ，我们讨论了而使用邻近查询来调整相关度，而不是使用它将文档从结果列表中添加或者排除。 一个查询可能会匹配成千上万的结果，但我们的用户很可能只对结果的前几页感兴趣。

一个简单的 match 查询已经通过排序把包含所有含有搜索词条的文档放在结果列表的前面了。事实上，我们只想对这些 顶部文档 重新排序，来给同时匹配了短语查询的文档一个额外的相关度升级。

search API 通过 重新评分 明确支持该功能。重新评分阶段支持一个代价更高的评分算法--比如 phrase 查询--只是为了从每个分片中获得前 K 个结果。 然后会根据它们的最新评分 重新排序。

该请求如下所示：

GET /my\_index/my\_type/\_search

{

"query": {

"match": { https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

"title": {

"query": "quick brown fox",

"minimum\_should\_match": "30%"

}

}

},

"rescore": {

"window\_size": 50, https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

"query": { https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png

"rescore\_query": {

"match\_phrase": {

"title": {

"query": "quick brown fox",

"slop": 50

}

}

}

}

}

}

**拷贝为 CURL**[在 SENSE 中查看](http://localhost:5601/app/sense/?load_from=https://www.elastic.co/guide/cn/elasticsearch/guide/current/snippets/120_Proximity_Matching/30_Performance.json)

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_Improving_Performance.html#CO87-1) | match 查询决定哪些文档将包含在最终结果集中，并通过 TF/IDF 排序。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_Improving_Performance.html#CO87-2) | window\_size 是每一分片进行重新评分的顶部文档数量。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_Improving_Performance.html#CO87-3) | 目前唯一支持的重新打分算法就是另一个查询，但是以后会有计划增加更多的算法。 |

### 寻找相关词[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/120_Proximity_Matching/35_Shingles.asciidoc)

短语查询和邻近查询都很好用，但仍有一个缺点。它们过于严格了：为了匹配短语查询，所有词项都必须存在，即使使用了 slop 。

用 slop 得到的单词顺序的灵活性也需要付出代价，因为失去了单词对之间的联系。即使可以识别 sue 、 alligator 和 ate 相邻出现的文档，但无法分辨是 Sue ate 还是 alligator ate 。

当单词相互结合使用的时候，表达的含义比单独使用更丰富。两个子句 I’m not happy I’m working 和 I’m happy I’m not working 包含相同 的单词，也拥有相同的邻近度，但含义截然不同。

如果索引单词对而不是索引独立的单词，就能对这些单词的上下文尽可能多的保留。

对句子 Sue ate the alligator ，不仅要将每一个单词（或者 unigram ）作为词项索引

["sue", "ate", "the", "alligator"]

也要将每个单词 以及它的邻近词 作为单个词项索引：

["sue ate", "ate the", "the alligator"]

这些单词对（或者 bigrams ）被称为 shingles 。

提示

Shingles 不限于单词对；你也可以索引三个单词（ trigrams ）：

["sue ate the", "ate the alligator"]

Trigrams 提供了更高的精度，但是也大大增加了索引中唯一词项的数量。在大多数情况下，Bigrams 就够了。

当然，只有当用户输入的查询内容和在原始文档中顺序相同时，shingles 才是有用的；对 sue alligator的查询可能会匹配到单个单词，但是不会匹配任何 shingles 。

幸运的是，用户倾向于使用和搜索数据相似的构造来表达搜索意图。但这一点很重要：只是索引 bigrams 是不够的；我们仍然需要 unigrams ，但可以将匹配 bigrams 作为增加相关度评分的信号。

#### 生成 Shingles[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/120_Proximity_Matching/35_Shingles.asciidoc)

Shingles 需要在索引时作为分析过程的一部分被创建。 我们可以将 unigrams 和 bigrams 都索引到单个字段中， 但将它们分开保存在能被独立查询的字段会更清晰。unigrams 字段将构成我们搜索的基础部分，而 bigrams 字段用来提高相关度。

首先，我们需要在创建分析器时使用 shingle 语汇单元过滤器：

DELETE /my\_index

PUT /my\_index

{

"settings": {

"number\_of\_shards": 1, https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

"analysis": {

"filter": {

"my\_shingle\_filter": {

"type": "shingle",

"min\_shingle\_size": 2, https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

"max\_shingle\_size": 2, https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png

"output\_unigrams": false https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/4.png

}

},

"analyzer": {

"my\_shingle\_analyzer": {

"type": "custom",

"tokenizer": "standard",

"filter": [

"lowercase",

"my\_shingle\_filter" https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/5.png

]

}

}

}

}

}

**拷贝为 CURL**[在 SENSE 中查看](http://localhost:5601/app/sense/?load_from=https://www.elastic.co/guide/cn/elasticsearch/guide/current/snippets/120_Proximity_Matching/35_Shingles.json)

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/shingles.html#CO88-1) | 参考 [被破坏的相关度！](https://www.elastic.co/guide/cn/elasticsearch/guide/current/relevance-is-broken.html) 。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/shingles.html#CO88-2) [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/shingles.html#CO88-3) | 默认最小/最大的 shingle 大小是 2 ，所以实际上不需要设置。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/4.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/shingles.html#CO88-4) | shingle 语汇单元过滤器默认输出 unigrams ，但是我们想让 unigrams 和 bigrams 分开。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/5.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/shingles.html#CO88-5) | my\_shingle\_analyzer 使用我们常规的 my\_shingles\_filter 语汇单元过滤器。 |

首先，用 analyze API 测试下分析器：

GET /my\_index/\_analyze?analyzer=my\_shingle\_analyzer

Sue ate the alligator

果然， 我们得到了 3 个词项：

* sue ate
* ate the
* the alligator

现在我们可以继续创建一个使用新的分析器的字段。

#### 多字段[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/120_Proximity_Matching/35_Shingles.asciidoc)

我们曾谈到将 unigrams 和 bigrams 分开索引更清晰，所以 title 字段将创建成一个多字段（参考 [字符串排序与多字段](https://www.elastic.co/guide/cn/elasticsearch/guide/current/multi-fields.html) ）：

PUT /my\_index/\_mapping/my\_type

{

"my\_type": {

"properties": {

"title": {

"type": "string",

"fields": {

"shingles": {

"type": "string",

"analyzer": "my\_shingle\_analyzer"

}

}

}

}

}

}

通过这个映射， JSON 文档中的 title 字段将会被以 unigrams (title)和 bigrams (title.shingles)被索引，这意味着可以独立地查询这些字段。

最后，我们可以索引以下示例文档:

POST /my\_index/my\_type/\_bulk

{ "index": { "\_id": 1 }}

{ "title": "Sue ate the alligator" }

{ "index": { "\_id": 2 }}

{ "title": "The alligator ate Sue" }

{ "index": { "\_id": 3 }}

{ "title": "Sue never goes anywhere without her alligator skin purse" }

#### 搜索 Shingles[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/120_Proximity_Matching/35_Shingles.asciidoc)

为了理解添加 shingles 字段的好处 ，让我们首先来看 The hungry alligator ate Sue 进行简单 match查询的结果：

GET /my\_index/my\_type/\_search

{

"query": {

"match": {

"title": "the hungry alligator ate sue"

}

}

}

这个查询返回了所有的三个文档， 但是注意文档 1 和 2 有相同的相关度评分因为他们包含了相同的单词：

{

"hits": [

{

"\_id": "1",

"\_score": 0.44273707, https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

"\_source": {

"title": "Sue ate the alligator"

}

},

{

"\_id": "2",

"\_score": 0.44273707, https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

"\_source": {

"title": "The alligator ate Sue"

}

},

{

"\_id": "3", https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png

"\_score": 0.046571054,

"\_source": {

"title": "Sue never goes anywhere without her alligator skin purse"

}

}

]

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/shingles.html#CO89-1) [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/shingles.html#CO89-2) | 两个文档都包含 the 、 alligator 和 ate ，所以获得相同的评分。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/shingles.html#CO89-3) | 我们可以通过设置 minimum\_should\_match 参数排除文档 3 ，参考 [控制精度](https://www.elastic.co/guide/cn/elasticsearch/guide/current/match-multi-word.html#match-precision) 。 |

现在在查询里添加 shingles 字段。不要忘了在 shingles 字段上的匹配是充当一 种信号--为了提高相关度评分--所以我们仍然需要将基本 title 字段包含到查询中：

GET /my\_index/my\_type/\_search

{

"query": {

"bool": {

"must": {

"match": {

"title": "the hungry alligator ate sue"

}

},

"should": {

"match": {

"title.shingles": "the hungry alligator ate sue"

}

}

}

}

}

仍然匹配到了所有的 3 个文档， 但是文档 2 现在排到了第一名因为它匹配了 shingled 词项 ate sue.

{

"hits": [

{

"\_id": "2",

"\_score": 0.4883322,

"\_source": {

"title": "The alligator ate Sue"

}

},

{

"\_id": "1",

"\_score": 0.13422975,

"\_source": {

"title": "Sue ate the alligator"

}

},

{

"\_id": "3",

"\_score": 0.014119488,

"\_source": {

"title": "Sue never goes anywhere without her alligator skin purse"

}

}

]

}

即使查询包含的单词 hungry 没有在任何文档中出现，我们仍然使用单词邻近度返回了最相关的文档。

#### Performance性能[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/120_Proximity_Matching/35_Shingles.asciidoc)

shingles 不仅比短语查询更灵活， 而且性能也更好。 shingles 查询跟一个简单的 match 查询一样高效，而不用每次搜索花费短语查询的代价。只是在索引期间因为更多词项需要被索引会付出一些小的代价， 这也意味着有 shingles 的字段会占用更多的磁盘空间。 然而，大多数应用写入一次而读取多次，所以在索引期间优化我们的查询速度是有意义的。

这是一个在 Elasticsearch 里会经常碰到的话题：不需要任何前期进行过多的设置，就能够在搜索的时候有很好的效果。 一旦更清晰的理解了自己的需求，就能在索引时通过正确的为你的数据建模获得更好结果和性能。

## 部分匹配[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/130_Partial_Matching/00_Intro.asciidoc)

敏锐的读者会注意，目前为止本书介绍的所有查询都是针对整个词的操作。为了能匹配，只能查找倒排索引中存在的词，最小的单元为单个词。

但如果想匹配部分而不是全部的词该怎么办？ 部分匹配 允许用户指定查找词的一部分并找出所有包含这部分片段的词。

与想象的不太一样，对词进行部分匹配的需求在全文搜索引擎领域并不常见，但是如果读者有 SQL 方面的背景，可能会在某个时候实现一个 低效的全文搜索 用下面的 SQL 语句对全文进行搜索：

WHERE text LIKE "%quick%"

AND text LIKE "%brown%"

AND text LIKE "%fox%" https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/partial-matching.html#CO90-1) | \*fox\* 会与 “fox” 和 “foxes” 匹配。 |

当然， Elasticsearch 提供分析过程，倒排索引让我们不需要使用这种粗笨的技术。为了能应对同时匹配 “fox” 和 “foxes” 的情况，只需简单的将它们的词干作为索引形式，没有必要做部分匹配。

也就是说，在某些情况下部分匹配会比较有用， 常见的应用如下：

* 匹配邮编、产品序列号或其他 not\_analyzed 未分析值，这些值可以是以某个特定前缀开始，也可以是与某种模式匹配的，甚至可以是与某个正则式相匹配的。
* 输入即搜索（search-as-you-type） ——在用户键入搜索词过程的同时就呈现最可能的结果。
* 匹配如德语或荷兰语这样有长组合词的语言，如： Weltgesundheitsorganisation （世界卫生组织，英文 World Health Organization）。

本章始于检验 not\_analyzed 精确值字段的前缀匹配。

### 邮编与结构化数据[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/130_Partial_Matching/05_Postcodes.asciidoc)

我们会使用美国目前使用的邮编形式（United Kingdom postcodes 标准）来说明如何用部分匹配查询结构化数据。 这种邮编形式有很好的结构定义。例如，邮编 W1V 3DG 可以分解成如下形式：

* W1V ：这是邮编的外部，它定义了邮件的区域和行政区：
  + W 代表区域（ 1 或 2 个字母）
  + 1V 代表行政区（ 1 或 2 个数字，可能跟着一个字符）
* 3DG ：内部定义了街道或建筑：
  + 3 代表街区区块（ 1 个数字）
  + DG 代表单元（ 2 个字母）

假设将邮编作为 not\_analyzed 的精确值字段索引，所以可以为其创建索引，如下：

PUT /my\_index

{

"mappings": {

"address": {

"properties": {

"postcode": {

"type": "string",

"index": "not\_analyzed"

}

}

}

}

}

**拷贝为 CURL**[在 SENSE 中查看](http://localhost:5601/app/sense/?load_from=https://www.elastic.co/guide/cn/elasticsearch/guide/current/snippets/130_Partial_Matching/10_Prefix_query.json)

然后索引一些邮编：

PUT /my\_index/address/1

{ "postcode": "W1V 3DG" }

PUT /my\_index/address/2

{ "postcode": "W2F 8HW" }

PUT /my\_index/address/3

{ "postcode": "W1F 7HW" }

PUT /my\_index/address/4

{ "postcode": "WC1N 1LZ" }

PUT /my\_index/address/5

{ "postcode": "SW5 0BE" }

**拷贝为 CURL**[在 SENSE 中查看](http://localhost:5601/app/sense/?load_from=https://www.elastic.co/guide/cn/elasticsearch/guide/current/snippets/130_Partial_Matching/10_Prefix_query.json)

现在这些数据已可查询。

### prefix 前缀查询[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/130_Partial_Matching/10_Prefix_query.asciidoc)

为了找到所有以 W1 开始的邮编，可以使用简单的 prefix 查询：

GET /my\_index/address/\_search

{

"query": {

"prefix": {

"postcode": "W1"

}

}

}

**拷贝为 CURL**[在 SENSE 中查看](http://localhost:5601/app/sense/?load_from=https://www.elastic.co/guide/cn/elasticsearch/guide/current/snippets/130_Partial_Matching/10_Prefix_query.json)

prefix 查询是一个词级别的底层的查询，它不会在搜索之前分析查询字符串，它假定传入前缀就正是要查找的前缀。

提示

默认状态下， prefix 查询不做相关度评分计算，它只是将所有匹配的文档返回，并为每条结果赋予评分值 1 。它的行为更像是过滤器而不是查询。 prefix 查询和 prefix 过滤器这两者实际的区别就是过滤器是可以被缓存的，而查询不行。

之前已经提过：“只能在倒排索引中找到存在的词”，但我们并没有对这些邮编的索引进行特殊处理，每个邮编还是以它们精确值的方式存在于每个文档的索引中，那么 prefix 查询是如何工作的呢？

回想倒排索引包含了一个有序的唯一词列表（本例是邮编）。 对于每个词，倒排索引都会将包含词的文档 ID 列入 倒排表（postings list） 。与示例对应的倒排索引是：

Term: Doc IDs:

-------------------------

"SW5 0BE" | 5

"W1F 7HW" | 3

"W1V 3DG" | 1

"W2F 8HW" | 2

"WC1N 1LZ" | 4

-------------------------

为了支持前缀匹配，查询会做以下事情：

1. 扫描词列表并查找到第一个以 W1 开始的词。
2. 搜集关联的文档 ID 。
3. 移动到下一个词。
4. 如果这个词也是以 W1 开头，查询跳回到第二步再重复执行，直到下一个词不以 W1 为止。

这对于小的例子当然可以正常工作，但是如果倒排索引中有数以百万的邮编都是以 W1 开头时，前缀查询则需要访问每个词然后计算结果！

前缀越短所需访问的词越多。如果我们要以 W 作为前缀而不是 W1 ，那么就可能需要做千万次的匹配。

小心

prefix 查询或过滤对于一些特定的匹配是有效的，但使用方式还是应当注意。 当字段中词的集合很小时，可以放心使用，但是它的伸缩性并不好，会对我们的集群带来很多压力。可以使用较长的前缀来限制这种影响，减少需要访问的量。

本章后面会介绍另一个索引时的解决方案，这个方案能使前缀匹配更高效，不过在此之前，需要先看看两个相关的查询： wildcard 和 regexp （模糊和正则）。

### 通配符与正则表达式查询[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/130_Partial_Matching/15_WildcardRegexp.asciidoc)

与 prefix 前缀查询的特性类似， wildcard 通配符查询也是一种底层基于词的查询， 与前缀查询不同的是它允许指定匹配的正则式。它使用标准的 shell 通配符查询： ? 匹配任意字符， \* 匹配 0 或多个字符。

这个查询会匹配包含 W1F 7HW 和 W2F 8HW 的文档：

GET /my\_index/address/\_search

{

"query": {

"wildcard": {

"postcode": "W?F\*HW" https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

}

}

}

**拷贝为 CURL**[在 SENSE 中查看](http://localhost:5601/app/sense/?load_from=https://www.elastic.co/guide/cn/elasticsearch/guide/current/snippets/130_Partial_Matching/15_Wildcard_regexp.json)

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_wildcard_and_regexp_queries.html#CO91-1) | ? 匹配 1 和 2 ， \* 与空格及 7 和 8 匹配。 |

设想如果现在只想匹配 W 区域的所有邮编，前缀匹配也会包括以 WC 开头的所有邮编，与通配符匹配碰到的问题类似，如果想匹配只以 W 开始并跟随一个数字的所有邮编， regexp 正则式查询允许写出这样更复杂的模式：

GET /my\_index/address/\_search

{

"query": {

"regexp": {

"postcode": "W[0-9].+" https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

}

}

}

**拷贝为 CURL**[在 SENSE 中查看](http://localhost:5601/app/sense/?load_from=https://www.elastic.co/guide/cn/elasticsearch/guide/current/snippets/130_Partial_Matching/15_Wildcard_regexp.json)

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_wildcard_and_regexp_queries.html#CO92-1) | 这个正则表达式要求词必须以 W 开头，紧跟 0 至 9 之间的任何一个数字，然后接一或多个其他字符。 |

wildcard 和 regexp 查询的工作方式与 prefix 查询完全一样，它们也需要扫描倒排索引中的词列表才能找到所有匹配的词，然后依次获取每个词相关的文档 ID ，与 prefix 查询的唯一不同是：它们能支持更为复杂的匹配模式。

这也意味着需要同样注意前缀查询存在性能问题，对有很多唯一词的字段执行这些查询可能会消耗非常多的资源，所以要避免使用左通配这样的模式匹配（如： \*foo 或 .\*foo 这样的正则式）。

数据在索引时的预处理有助于提高前缀匹配的效率，而通配符和正则表达式查询只能在查询时完成，尽管这些查询有其应用场景，但使用仍需谨慎。

小心

prefix 、 wildcard 和 regexp 查询是基于词操作的，如果用它们来查询 analyzed 字段，它们会检查字段里面的每个词，而不是将字段作为整体来处理。

比方说包含 “Quick brown fox” （快速的棕色狐狸）的 title 字段会生成词： quick 、 brown 和 fox 。

会匹配以下这个查询：

{ "regexp": { "title": "br.\*" }}

但是不会匹配以下两个查询：

{ "regexp": { "title": "Qu.\*" }} https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

{ "regexp": { "title": "quick br\*" }} https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_wildcard_and_regexp_queries.html#CO93-1) | 在索引里的词是 quick 而不是 Quick 。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_wildcard_and_regexp_queries.html#CO93-2) | quick 和 brown 在词表中是分开的。 |

### 查询时输入即搜索[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/130_Partial_Matching/20_Match_phrase_prefix.asciidoc)

把邮编的事情先放一边，让我们先看看前缀查询是如何在全文查询中起作用的。 用户已经渐渐习惯在输完查询内容之前，就能为他们展现搜索结果，这就是所谓的 即时搜索（instant search） 或 输入即搜索（search-as-you-type） 。不仅用户能在更短的时间内得到搜索结果，我们也能引导用户搜索索引中真实存在的结果。

例如，如果用户输入 johnnie walker bl ，我们希望在它们完成输入搜索条件前就能得到：Johnnie Walker Black Label 和 Johnnie Walker Blue Label 。

生活总是这样，就像猫的花色远不只一种！我们希望能找到一种最简单的实现方式。并不需要对数据做任何准备，在查询时就能对任意的全文字段实现 输入即搜索（search-as-you-type） 的查询。

在 [短语匹配](https://www.elastic.co/guide/cn/elasticsearch/guide/current/phrase-matching.html) 中，我们引入了 match\_phrase 短语匹配查询，它匹配相对顺序一致的所有指定词语，对于查询时的输入即搜索，可以使用 match\_phrase 的一种特殊形式， match\_phrase\_prefix 查询：

{

"match\_phrase\_prefix" : {

"brand" : "johnnie walker bl"

}

}

**拷贝为 CURL**[在 SENSE 中查看](http://localhost:5601/app/sense/?load_from=https://www.elastic.co/guide/cn/elasticsearch/guide/current/snippets/130_Partial_Matching/20_Match_phrase_prefix.json)

这种查询的行为与 match\_phrase 查询一致，不同的是它将查询字符串的最后一个词作为前缀使用，换句话说，可以将之前的例子看成如下这样：

* johnnie
* 跟着 walker
* 跟着以 bl 开始的词

如果通过 validate-query API 运行这个查询查询，explanation 的解释结果为：

"johnnie walker bl\*"

与 match\_phrase 一样，它也可以接受 slop 参数（参照 [slop](https://www.elastic.co/guide/cn/elasticsearch/guide/current/slop.html) ）让相对词序位置不那么严格：

{

"match\_phrase\_prefix" : {

"brand" : {

"query": "walker johnnie bl", https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

"slop": 10

}

}

}

**拷贝为 CURL**[在 SENSE 中查看](http://localhost:5601/app/sense/?load_from=https://www.elastic.co/guide/cn/elasticsearch/guide/current/snippets/130_Partial_Matching/20_Match_phrase_prefix.json)

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_query_time_search_as_you_type.html#CO94-1) | 尽管词语的顺序不正确，查询仍然能匹配，因为我们为它设置了足够高的 slop 值使匹配时的词序有更大的灵活性。 |

但是只有查询字符串的最后一个词才能当作前缀使用。

在之前的 [前缀查询](https://www.elastic.co/guide/cn/elasticsearch/guide/current/prefix-query.html) 中，我们警告过使用前缀的风险，即 prefix 查询存在严重的资源消耗问题，短语查询的这种方式也同样如此。 前缀 a 可能会匹配成千上万的词，这不仅会消耗很多系统资源，而且结果的用处也不大。

可以通过设置 max\_expansions 参数来限制前缀扩展的影响， 一个合理的值是可能是 50 ：

{

"match\_phrase\_prefix" : {

"brand" : {

"query": "johnnie walker bl",

"max\_expansions": 50

}

}

}

**拷贝为 CURL**[在 SENSE 中查看](http://localhost:5601/app/sense/?load_from=https://www.elastic.co/guide/cn/elasticsearch/guide/current/snippets/130_Partial_Matching/20_Match_phrase_prefix.json)

参数 max\_expansions 控制着可以与前缀匹配的词的数量，它会先查找第一个与前缀 bl 匹配的词，然后依次查找搜集与之匹配的词（按字母顺序），直到没有更多可匹配的词或当数量超过 max\_expansions 时结束。

不要忘记，当用户每多输入一个字符时，这个查询又会执行一遍，所以查询需要快，如果第一个结果集不是用户想要的，他们会继续输入直到能搜出满意的结果为止。

### 索引时优化[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/130_Partial_Matching/25_Index_time.asciidoc)

到目前为止，所有谈论过的解决方案都是在 查询时（query time） 实现的。 这样做并不需要特殊的映射或特殊的索引模式，只是简单使用已经索引的数据。

查询时的灵活性通常会以牺牲搜索性能为代价，有时候将这些消耗从查询过程中转移到别的地方是有意义的。在实时 web 应用中， 100 毫秒可能是一个难以忍受的巨大延迟。

可以通过在索引时处理数据提高搜索的灵活性以及提升系统性能。为此仍然需要付出应有的代价：增加的索引空间与变慢的索引能力，但这与每次查询都需要付出代价不同，索引时的代价只用付出一次。

用户会感谢我们。

[«  索引时优化](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_index_time_optimizations.html)    [索引时输入即搜索  »](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_index_time_search_as_you_type.html)

### Ngrams 在部分匹配的应用[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/130_Partial_Matching/30_Ngram_intro.asciidoc)

之前提到：“只能在倒排索引中找到存在的词。” 尽管 prefix 、 wildcard 、 regexp 查询告诉我们这种说法并不完全正确，但单个词的查找 确实 要比在词列表中盲目挨个查找的效率要高得多。 在搜索之前准备好供部分匹配的数据可以提高搜索的性能。

在索引时准备数据意味着要选择合适的分析链，这里部分匹配使用的工具是 n-gram 。可以将 n-gram 看成一个在词语上 滑动窗口 ， n 代表这个 “窗口” 的长度。如果我们要 n-gram quick 这个词 —— 它的结果取决于 n 的选择长度：

* 长度 1（unigram）： [ q, u, i, c, k ]
* 长度 2（bigram）： [ qu, ui, ic, ck ]
* 长度 3（trigram）： [ qui, uic, ick ]
* 长度 4（four-gram）： [ quic, uick ]
* 长度 5（five-gram）： [ quick ]

朴素的 n-gram 对 词语内部的匹配 非常有用，即在 [Ngram 匹配复合词](https://www.elastic.co/guide/cn/elasticsearch/guide/current/ngrams-compound-words.html) 介绍的那样。但对于输入即搜索（search-as-you-type）这种应用场景，我们会使用一种特殊的 n-gram 称为 边界 n-grams （edge n-grams）。所谓的边界 n-gram 是说它会固定词语开始的一边，以单词 quick 为例，它的边界 n-gram 的结果为：

* q
* qu
* qui
* quic
* quick

可能会注意到这与用户在搜索时输入 “quick” 的字母次序是一致的，换句话说，这种方式正好满足即时搜索（instant search）！

### 索引时输入即搜索[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/130_Partial_Matching/35_Search_as_you_type.asciidoc)

设置索引时输入即搜索的第一步是需要定义好分析链， 我们已在 [配置分析器](https://www.elastic.co/guide/cn/elasticsearch/guide/current/configuring-analyzers.html) 中讨论过，这里会对这些步骤再次说明。

#### 准备索引[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/130_Partial_Matching/35_Search_as_you_type.asciidoc)

第一步需要配置一个自定义的 edge\_ngram token 过滤器，称为 autocomplete\_filter ：

{

"filter": {

"autocomplete\_filter": {

"type": "edge\_ngram",

"min\_gram": 1,

"max\_gram": 20

}

}

}

这个配置的意思是：对于这个 token 过滤器接收的任意词项，过滤器会为之生成一个最小固定值为 1 ，最大为 20 的 n-gram 。

然后会在一个自定义分析器 autocomplete 中使用上面这个 token 过滤器：

{

"analyzer": {

"autocomplete": {

"type": "custom",

"tokenizer": "standard",

"filter": [

"lowercase",

"autocomplete\_filter" https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

]

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_index_time_search_as_you_type.html#CO95-1) | 自定义的 edge-ngram token 过滤器。 |

这个分析器使用 standard 分词器将字符串拆分为独立的词，并且将它们都变成小写形式，然后为每个词生成一个边界 n-gram，这要感谢 autocomplete\_filter 起的作用。

创建索引、实例化 token 过滤器和分析器的完整示例如下：

PUT /my\_index

{

"settings": {

"number\_of\_shards": 1, https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

"analysis": {

"filter": {

"autocomplete\_filter": { https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

"type": "edge\_ngram",

"min\_gram": 1,

"max\_gram": 20

}

},

"analyzer": {

"autocomplete": {

"type": "custom",

"tokenizer": "standard",

"filter": [

"lowercase",

"autocomplete\_filter" https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png

]

}

}

}

}

}

**拷贝为 CURL**[在 SENSE 中查看](http://localhost:5601/app/sense/?load_from=https://www.elastic.co/guide/cn/elasticsearch/guide/current/snippets/130_Partial_Matching/35_Search_as_you_type.json)

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_index_time_search_as_you_type.html#CO96-1) | 参考 [被破坏的相关度](https://www.elastic.co/guide/cn/elasticsearch/guide/current/relevance-is-broken.html) 。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_index_time_search_as_you_type.html#CO96-2) | 首先自定义 token 过滤器。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_index_time_search_as_you_type.html#CO96-3) | 然后在分析器中使用它。 |

可以拿 analyze API 测试这个新的分析器确保它行为正确：

GET /my\_index/\_analyze?analyzer=autocomplete

quick brown

**拷贝为 CURL**[在 SENSE 中查看](http://localhost:5601/app/sense/?load_from=https://www.elastic.co/guide/cn/elasticsearch/guide/current/snippets/130_Partial_Matching/35_Search_as_you_type.json)

结果表明分析器能正确工作，并返回以下词：

* q
* qu
* qui
* quic
* quick
* b
* br
* bro
* brow
* brown

可以用 update-mapping API 将这个分析器应用到具体字段：

PUT /my\_index/\_mapping/my\_type

{

"my\_type": {

"properties": {

"name": {

"type": "string",

"analyzer": "autocomplete"

}

}

}

}

**拷贝为 CURL**[在 SENSE 中查看](http://localhost:5601/app/sense/?load_from=https://www.elastic.co/guide/cn/elasticsearch/guide/current/snippets/130_Partial_Matching/35_Search_as_you_type.json)

现在创建一些测试文档：

POST /my\_index/my\_type/\_bulk

{ "index": { "\_id": 1 }}

{ "name": "Brown foxes" }

{ "index": { "\_id": 2 }}

{ "name": "Yellow furballs" }

**拷贝为 CURL**[在 SENSE 中查看](http://localhost:5601/app/sense/?load_from=https://www.elastic.co/guide/cn/elasticsearch/guide/current/snippets/130_Partial_Matching/35_Search_as_you_type.json)

#### 查询字段[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/130_Partial_Matching/35_Search_as_you_type.asciidoc)

如果使用简单 match 查询测试查询 “brown fo” ：

GET /my\_index/my\_type/\_search

{

"query": {

"match": {

"name": "brown fo"

}

}

}

**拷贝为 CURL**[在 SENSE 中查看](http://localhost:5601/app/sense/?load_from=https://www.elastic.co/guide/cn/elasticsearch/guide/current/snippets/130_Partial_Matching/35_Search_as_you_type.json)

可以看到两个文档同时 都能 匹配，尽管 Yellow furballs 这个文档并不包含 brown 和 fo ：

{

"hits": [

{

"\_id": "1",

"\_score": 1.5753809,

"\_source": {

"name": "Brown foxes"

}

},

{

"\_id": "2",

"\_score": 0.012520773,

"\_source": {

"name": "Yellow furballs"

}

}

]

}

如往常一样， validate-query API 总能提供一些线索：

GET /my\_index/my\_type/\_validate/query?explain

{

"query": {

"match": {

"name": "brown fo"

}

}

}

**拷贝为 CURL**[在 SENSE 中查看](http://localhost:5601/app/sense/?load_from=https://www.elastic.co/guide/cn/elasticsearch/guide/current/snippets/130_Partial_Matching/35_Search_as_you_type.json)

explanation 表明查询会查找边界 n-grams 里的每个词：

name:b name:br name:bro name:brow name:brown name:f name:fo

name:f 条件可以满足第二个文档，因为 furballs 是以 f 、 fu 、 fur 形式索引的。回过头看这并不令人惊讶，相同的 autocomplete 分析器同时被应用于索引时和搜索时，这在大多数情况下是正确的，只有在少数场景下才需要改变这种行为。

我们需要保证倒排索引表中包含边界 n-grams 的每个词，但是我们只想匹配用户输入的完整词组（ brown和 fo ）， 可以通过在索引时使用 autocomplete 分析器，并在搜索时使用 standard 标准分析器来实现这种想法，只要改变查询使用的搜索分析器 analyzer 参数即可：

GET /my\_index/my\_type/\_search

{

"query": {

"match": {

"name": {

"query": "brown fo",

"analyzer": "standard" https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

}

}

}

}

**拷贝为 CURL**[在 SENSE 中查看](http://localhost:5601/app/sense/?load_from=https://www.elastic.co/guide/cn/elasticsearch/guide/current/snippets/130_Partial_Matching/35_Search_as_you_type.json)

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_index_time_search_as_you_type.html#CO97-1) | 覆盖了 name 字段 analyzer 的设置。 |

换种方式，我们可以在映射中，为 name 字段分别指定 index\_analyzer 和 search\_analyzer 。因为我们只想改变 search\_analyzer ，这里只要更新现有的映射而不用对数据重新创建索引：

PUT /my\_index/my\_type/\_mapping

{

"my\_type": {

"properties": {

"name": {

"type": "string",

"index\_analyzer": "autocomplete", https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

"search\_analyzer": "standard" https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

}

}

}

}

**拷贝为 CURL**[在 SENSE 中查看](http://localhost:5601/app/sense/?load_from=https://www.elastic.co/guide/cn/elasticsearch/guide/current/snippets/130_Partial_Matching/35_Search_as_you_type.json)

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_index_time_search_as_you_type.html#CO98-1) | 在索引时，使用 autocomplete 分析器生成边界 n-grams 的每个词。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_index_time_search_as_you_type.html#CO98-2) | 在搜索时，使用 standard 分析器只搜索用户输入的词。 |

如果再次请求 validate-query API ，当前的解释为：

name:brown name:fo

再次执行查询就能正确返回 Brown foxes 这个文档。

因为大多数工作是在索引时完成的，所有的查询只要查找 brown 和 fo 这两个词，这比使用 match\_phrase\_prefix 查找所有以 fo 开始的词的方式要高效许多。

**补全提示（Completion Suggester）**

使用边界 n-grams 进行输入即搜索（search-as-you-type）的查询设置简单、灵活且快速，但有时候它并不够快，特别是当试图立刻获得反馈时，延迟的问题就会凸显，很多时候不搜索才是最快的搜索方式。

Elasticsearch 里的 [completion suggester](https://www.elastic.co/guide/en/elasticsearch/reference/5.6/search-suggesters-completion.html) 采用与上面完全不同的方式，需要为搜索条件生成一个所有可能完成的词列表，然后将它们置入一个 有限状态机（finite state transducer） 内，这是个经优化的图结构。为了搜索建议提示，Elasticsearch 从图的开始处顺着匹配路径一个字符一个字符地进行匹配，一旦它处于用户输入的末尾，Elasticsearch 就会查找所有可能结束的当前路径，然后生成一个建议列表。

本数据结构存于内存中，能使前缀查找非常快，比任何一种基于词的查询都要快很多，这对名字或品牌的自动补全非常适用，因为这些词通常是以普通顺序组织的：用 “Johnny Rotten” 而不是 “Rotten Johnny” 。

当词序不是那么容易被预见时，边界 n-grams 比完成建议者（Completion Suggester）更合适。即使说不是所有猫都是一个花色，那这只猫的花色也是相当特殊的。

#### 边界 n-grams 与邮编[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/130_Partial_Matching/35_Search_as_you_type.asciidoc)

边界 n-gram 的方式可以用来查询结构化的数据， 比如 [本章之前示例](https://www.elastic.co/guide/cn/elasticsearch/guide/current/prefix-query.html) 中的邮编（postcode）。当然 postcode 字段需要 analyzed 而不是 not\_analyzed ，不过可以用 keyword 分词器来处理它，就好像他们是 not\_analyzed 的一样。

提示

keyword 分词器是一个非操作型分词器，这个分词器不做任何事情，它接收的任何字符串都会被原样发出，因此它可以用来处理 not\_analyzed 的字段值，但这也需要其他的一些分析转换，如将字母转换成小写。

下面示例使用 keyword 分词器将邮编转换成 token 流，这样就能使用边界 n-gram token 过滤器：

{

"analysis": {

"filter": {

"postcode\_filter": {

"type": "edge\_ngram",

"min\_gram": 1,

"max\_gram": 8

}

},

"analyzer": {

"postcode\_index": { https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

"tokenizer": "keyword",

"filter": [ "postcode\_filter" ]

},

"postcode\_search": { https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

"tokenizer": "keyword"

}

}

}

}

**拷贝为 CURL**[在 SENSE 中查看](http://localhost:5601/app/sense/?load_from=https://www.elastic.co/guide/cn/elasticsearch/guide/current/snippets/130_Partial_Matching/35_Postcodes.json)

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_index_time_search_as_you_type.html#CO99-1) | postcode\_index 分析器使用 postcode\_filter 将邮编转换成边界 n-gram 形式。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_index_time_search_as_you_type.html#CO99-2) | postcode\_search 分析器可以将搜索词看成 not\_analyzed 未分析的。 |

### Ngrams 在复合词的应用[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/130_Partial_Matching/40_Compound_words.asciidoc)

最后，来看看 n-gram 是如何应用于搜索复合词的语言中的。 德语的特点是它可以将许多小词组合成一个庞大的复合词以表达它准确或复杂的意义。例如：

Aussprachewörterbuch

发音字典（Pronunciation dictionary）

Militärgeschichte

战争史（Military history）

Weißkopfseeadler

秃鹰（White-headed sea eagle, or bald eagle）

Weltgesundheitsorganisation

世界卫生组织（World Health Organization）

Rindfleischetikettierungsüberwachungsaufgabenübertragungsgesetz

法案考虑代理监管牛和牛肉的标记的职责（The law concerning the delegation of duties for the supervision of cattle marking and the labeling of beef）

有些人希望在搜索 “Wörterbuch”（字典）的时候，能在结果中看到 “Aussprachewörtebuch”（发音字典）。同样，搜索 “Adler”（鹰）的时候，能将 “Weißkopfseeadler”（秃鹰）包括在结果中。

处理这种语言的一种方式可以用 [组合词 token 过滤器（compound word token filter）](https://www.elastic.co/guide/en/elasticsearch/reference/5.6/analysis-compound-word-tokenfilter.html) 将复合词拆分成各自部分，但这种方式的结果质量依赖于组合词字典的质量。

另一种方式就是将所有的词用 n-gram 进行处理，然后搜索任何匹配的片段——能匹配的片段越多，文档的相关度越大。

假设某个 n-gram 是一个词上的滑动窗口，那么任何长度的 n-gram 都可以遍历这个词。我们既希望选择足够长的值让拆分的词项具有意义，又不至于因为太长而生成过多的唯一词。一个长度为 3 的 trigram 可能是一个不错的开始：

PUT /my\_index

{

"settings": {

"analysis": {

"filter": {

"trigrams\_filter": {

"type": "ngram",

"min\_gram": 3,

"max\_gram": 3

}

},

"analyzer": {

"trigrams": {

"type": "custom",

"tokenizer": "standard",

"filter": [

"lowercase",

"trigrams\_filter"

]

}

}

}

},

"mappings": {

"my\_type": {

"properties": {

"text": {

"type": "string",

"analyzer": "trigrams" https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

}

}

}

}

}

**拷贝为 CURL**[在 SENSE 中查看](http://localhost:5601/app/sense/?load_from=https://www.elastic.co/guide/cn/elasticsearch/guide/current/snippets/130_Partial_Matching/40_Compound_words.json)

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/ngrams-compound-words.html#CO100-1) | text 字段用 trigrams 分析器索引它的内容，这里 n-gram 的长度是 3 。 |

使用 analyze API 测试 trigram 分析器：

GET /my\_index/\_analyze?analyzer=trigrams

Weißkopfseeadler

**拷贝为 CURL**[在 SENSE 中查看](http://localhost:5601/app/sense/?load_from=https://www.elastic.co/guide/cn/elasticsearch/guide/current/snippets/130_Partial_Matching/40_Compound_words.json)

返回以下词项：

wei, eiß, ißk, ßko, kop, opf, pfs, fse, see, eea,ead, adl, dle, ler

索引前述示例中的复合词来测试：

POST /my\_index/my\_type/\_bulk

{ "index": { "\_id": 1 }}

{ "text": "Aussprachewörterbuch" }

{ "index": { "\_id": 2 }}

{ "text": "Militärgeschichte" }

{ "index": { "\_id": 3 }}

{ "text": "Weißkopfseeadler" }

{ "index": { "\_id": 4 }}

{ "text": "Weltgesundheitsorganisation" }

{ "index": { "\_id": 5 }}

{ "text": "Rindfleischetikettierungsüberwachungsaufgabenübertragungsgesetz" }

**拷贝为 CURL**[在 SENSE 中查看](http://localhost:5601/app/sense/?load_from=https://www.elastic.co/guide/cn/elasticsearch/guide/current/snippets/130_Partial_Matching/40_Compound_words.json)

“Adler”（鹰）的搜索转化为查询三个词 adl 、 dle 和 ler ：

GET /my\_index/my\_type/\_search

{

"query": {

"match": {

"text": "Adler"

}

}

}

**拷贝为 CURL**[在 SENSE 中查看](http://localhost:5601/app/sense/?load_from=https://www.elastic.co/guide/cn/elasticsearch/guide/current/snippets/130_Partial_Matching/40_Compound_words.json)

正好与 “Weißkopfsee-adler” 相匹配：

{

"hits": [

{

"\_id": "3",

"\_score": 3.3191128,

"\_source": {

"text": "Weißkopfseeadler"

}

}

]

}

**拷贝为 CURL**[在 SENSE 中查看](http://localhost:5601/app/sense/?load_from=https://www.elastic.co/guide/cn/elasticsearch/guide/current/snippets/130_Partial_Matching/40_Compound_words.json)

类似查询 “Gesundheit”（健康）可以与 “Welt-gesundheit-sorganisation” 匹配，同时也能与 “Militär-ges-chichte” 和 “Rindfleischetikettierungsüberwachungsaufgabenübertragungs-ges-etz” 匹配，因为它们同时都有 trigram 生成的 ges ：

使用合适的 minimum\_should\_match 可以将这些奇怪的结果排除，只有当 trigram 最少匹配数满足要求时，文档才能被认为是匹配的：

GET /my\_index/my\_type/\_search

{

"query": {

"match": {

"text": {

"query": "Gesundheit",

"minimum\_should\_match": "80%"

}

}

}

}

**拷贝为 CURL**[在 SENSE 中查看](http://localhost:5601/app/sense/?load_from=https://www.elastic.co/guide/cn/elasticsearch/guide/current/snippets/130_Partial_Matching/40_Compound_words.json)

这有点像全文搜索中霰弹枪式的策略，可能会导致倒排索引内容变多，尽管如此，在索引具有很多复合词的语言，或词之间没有空格的语言（如：泰语）时，它仍不失为一种通用且有效的方法。

这种技术可以用来提升 召回率 ——搜索结果中相关的文档数。它通常会与其他技术一起使用，例如 shingles（参见 [shingles 瓦片词](https://www.elastic.co/guide/cn/elasticsearch/guide/current/shingles.html) ），以提高精度和每个文档的相关度评分。

## 控制相关度[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/170_Relevance/05_Intro.asciidoc)

处理结构化数据（比如：时间、数字、字符串、枚举）的数据库， 只需检查文档（或关系数据库里的行）是否与查询匹配。

布尔的是/非匹配是全文搜索的基础，但不止如此，我们还要知道每个文档与查询的相关度，在全文搜索引擎中不仅需要找到匹配的文档，还需根据它们相关度的高低进行排序。

全文相关的公式或 相似算法（similarity algorithms） 会将多个因素合并起来，为每个文档生成一个相关度评分 \_score 。本章中，我们会验证各种可变部分，然后讨论如何来控制它们。

当然，相关度不只与全文查询有关，也需要将结构化的数据考虑其中。可能我们正在找一个度假屋，需要一些的详细特征（空调、海景、免费 WiFi ），匹配的特征越多相关度越高。可能我们还希望有一些其他的考虑因素，如回头率、价格、受欢迎度或距离，当然也同时考虑全文查询的相关度。

所有的这些都可以通过 Elasticsearch 强大的评分基础来实现。

本章会先从理论上介绍 Lucene 是如何计算相关度的，然后通过实际例子说明如何控制相关度的计算过程

# Delete the `my\_index` index

DELETE /my\_index

# Create `my\_index` with a single primary shard

PUT /my\_index

{ "settings": { "number\_of\_shards": 1 }}

# Index some example docs

POST /my\_index/my\_type/\_bulk

{ "index": { "\_id": 1 }}

{ "title": "The quick brown fox" }

{ "index": { "\_id": 2 }}

{ "title": "The quick brown fox jumps over the lazy dog" }

{ "index": { "\_id": 3 }}

{ "title": "The quick brown fox jumps over the quick dog" }

{ "index": { "\_id": 4 }}

{ "title": "Brown fox brown dog" }

### 相关度评分背后的理论[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/170_Relevance/10_Scoring_theory.asciidoc)

Lucene（或 Elasticsearch）使用 [布尔模型（Boolean model）](http://en.wikipedia.org/wiki/Standard_Boolean_model) 查找匹配文档， 并用一个名为 [实用评分函数（practical scoring function）](https://www.elastic.co/guide/cn/elasticsearch/guide/current/practical-scoring-function.html) 的公式来计算相关度。这个公式借鉴了 [词频/逆向文档频率（term frequency/inverse document frequency）](http://en.wikipedia.org/wiki/Tfidf) 和 [向量空间模型（vector space model）](http://en.wikipedia.org/wiki/Vector_space_model)，同时也加入了一些现代的新特性，如协调因子（coordination factor），字段长度归一化（field length normalization），以及词或查询语句权重提升。

不要紧张！这些概念并没有像它们字面看起来那么复杂，尽管本小节提到了算法、公式和数学模型，但内容还是让人容易理解的，与理解算法本身相比，了解这些因素如何影响结果更为重要。

#### 布尔模型[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/170_Relevance/10_Scoring_theory.asciidoc)

布尔模型（Boolean Model） 只是在查询中使用 AND 、 OR 和 NOT （与、或和非）这样的条件来查找匹配的文档，以下查询：

full AND text AND search AND (elasticsearch OR lucene)

会将所有包括词 full 、 text 和 search ，以及 elasticsearch 或 lucene 的文档作为结果集。

这个过程简单且快速，它将所有可能不匹配的文档排除在外。

#### 词频/逆向文档频率（TF/IDF）[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/170_Relevance/10_Scoring_theory.asciidoc)

当匹配到一组文档后，需要根据相关度排序这些文档，不是所有的文档都包含所有词，有些词比其他的词更重要。一个文档的相关度评分部分取决于每个查询词在文档中的 权重 。

词的权重由三个因素决定，在 [什么是相关](https://www.elastic.co/guide/cn/elasticsearch/guide/current/relevance-intro.html) 中已经有所介绍，有兴趣可以了解下面的公式，但并不要求记住。

##### 词频[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/170_Relevance/10_Scoring_theory.asciidoc)

词在文档中出现的频度是多少？ 频度越高，权重 越高 。 5 次提到同一词的字段比只提到 1 次的更相关。词频的计算方式如下：

tf(t in d) = √frequency

|  |  |
| --- | --- |
|  | 词 t 在文档 d 的词频（ tf ）是该词在文档中出现次数的平方根。 |

如果不在意词在某个字段中出现的频次，而只在意是否出现过，则可以在字段映射中禁用词频统计：

PUT /my\_index

{

"mappings": {

"doc": {

"properties": {

"text": {

"type": "string",

"index\_options": "docs"

}

}

}

}

}

|  |  |
| --- | --- |
|  | 将参数 index\_options 设置为 docs 可以禁用词频统计及词频位置，这个映射的字段不会计算词的出现次数，对于短语或近似查询也不可用。要求精确查询的 not\_analyzed 字符串字段会默认使用该设置。 |

##### 逆向文档频率[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/170_Relevance/10_Scoring_theory.asciidoc)

词在集合所有文档里出现的频率是多少？频次越高，权重 越低 。 常用词如 and 或 the 对相关度贡献很少，因为它们在多数文档中都会出现，一些不常见词如 elastic 或 hippopotamus 可以帮助我们快速缩小范围找到感兴趣的文档。逆向文档频率的计算公式如下：

idf(t) = 1 + log ( numDocs / (docFreq + 1))

|  |  |
| --- | --- |
|  | 词 t 的逆向文档频率（ idf ）是：索引中文档数量除以所有包含该词的文档数，然后求其对数。 |

##### 字段长度归一值[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/170_Relevance/10_Scoring_theory.asciidoc)

字段的长度是多少？ 字段越短，字段的权重 越高 。如果词出现在类似标题 title 这样的字段，要比它出现在内容 body 这样的字段中的相关度更高。字段长度的归一值公式如下：

norm(d) = 1 / √numTerms

|  |  |
| --- | --- |
|  | 字段长度归一值（ norm ）是字段中词数平方根的倒数。 |

字段长度的归一值对全文搜索非常重要， 许多其他字段不需要有归一值。无论文档是否包括这个字段，索引中每个文档的每个 string 字段都大约占用 1 个 byte 的空间。对于 not\_analyzed 字符串字段的归一值默认是禁用的，而对于 analyzed 字段也可以通过修改字段映射禁用归一值：

PUT /my\_index

{

"mappings": {

"doc": {

"properties": {

"text": {

"type": "string",

"norms": { "enabled": false }

}

}

}

}

}

|  |  |
| --- | --- |
|  | 这个字段不会将字段长度归一值考虑在内，长字段和短字段会以相同长度计算评分。 |

对于有些应用场景如日志，归一值不是很有用，要关心的只是字段是否包含特殊的错误码或者特定的浏览器唯一标识符。字段的长度对结果没有影响，禁用归一值可以节省大量内存空间。

##### 结合使用[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/170_Relevance/10_Scoring_theory.asciidoc)

以下三个因素——词频（term frequency）、逆向文档频率（inverse document frequency）和字段长度归一值（field-length norm）——是在索引时计算并存储的。 最后将它们结合在一起计算单个词在特定文档中的 权重 。

前面公式中提到的 文档 实际上是指文档里的某个字段，每个字段都有它自己的倒排索引，因此字段的 TF/IDF 值就是文档的 TF/IDF 值。

当用 explain 查看一个简单的 term 查询时（参见 [explain](https://www.elastic.co/guide/cn/elasticsearch/guide/current/relevance-intro.html#explain) ），可以发现与计算相关度评分的因子就是前面章节介绍的这些：

PUT /my\_index/doc/1

{ "text" : "quick brown fox" }

GET /my\_index/doc/\_search?explain

{

"query": {

"term": {

"text": "fox"

}

}

}

以上请求（简化）的 explanation 解释如下：

weight(text:fox in 0) [PerFieldSimilarity]: 0.15342641

result of:

fieldWeight in 0 0.15342641

product of:

tf(freq=1.0), with freq of 1: 1.0

idf(docFreq=1, maxDocs=1): 0.30685282

fieldNorm(doc=0): 0.5

|  |  |
| --- | --- |
|  | 词 fox 在文档的内部 Lucene doc ID 为 0 ，字段是 text 里的最终评分。 |
|  | 词 fox 在该文档 text 字段中只出现了一次。 |
|  | fox 在所有文档 text 字段索引的逆向文档频率。 |
|  | 该字段的字段长度归一值。 |

当然，查询通常不止一个词，所以需要一种合并多词权重的方式——向量空间模型（vector space model）。

#### 向量空间模型[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/170_Relevance/10_Scoring_theory.asciidoc)

向量空间模型（vector space model） 提供一种比较多词查询的方式，单个评分代表文档与查询的匹配程度，为了做到这点，这个模型将文档和查询都以 向量（vectors） 的形式表示：

向量实际上就是包含多个数的一维数组，例如：

[1,2,5,22,3,8]

在向量空间模型里， 向量空间模型里的每个数字都代表一个词的 权重 ，与 [词频/逆向文档频率（term frequency/inverse document frequency）](https://www.elastic.co/guide/cn/elasticsearch/guide/current/scoring-theory.html#tfidf) 计算方式类似。

尽管 TF/IDF 是向量空间模型计算词权重的默认方式，但不是唯一方式。Elasticsearch 还有其他模型如 Okapi-BM25 。TF/IDF 是默认的因为它是个经检验过的简单又高效的算法，可以提供高质量的搜索结果。

设想如果查询 “happy hippopotamus” ，常见词 happy 的权重较低，不常见词 hippopotamus 权重较高，假设 happy 的权重是 2 ， hippopotamus 的权重是 5 ，可以将这个二维向量—— [2,5] ——在坐标系下作条直线，线的起点是 (0,0) 终点是 (2,5) ，如图 [图 27 “表示 “happy hippopotamus” 的二维查询向量”](https://www.elastic.co/guide/cn/elasticsearch/guide/current/scoring-theory.html#img-vector-query) 。

**图 27. 表示 “happy hippopotamus” 的二维查询向量**

现在，设想我们有三个文档：

1. I am happy in summer 。
2. After Christmas I’m a hippopotamus 。
3. The happy hippopotamus helped Harry 。

可以为每个文档都创建包括每个查询词—— happy 和 hippopotamus ——权重的向量，然后将这些向量置入同一个坐标系中，如图 [图 28 ““happy hippopotamus” 查询及文档向量”](https://www.elastic.co/guide/cn/elasticsearch/guide/current/scoring-theory.html#img-vector-docs) ：

* 文档 1： (happy,\_\_\_\_\_\_\_\_\_\_\_\_) —— [2,0]
* 文档 2： ( \_\_\_ ,hippopotamus) —— [0,5]
* 文档 3： (happy,hippopotamus) —— [2,5]

**图 28. “happy hippopotamus” 查询及文档向量**

向量之间是可以比较的，只要测量查询向量和文档向量之间的角度就可以得到每个文档的相关度，文档 1 与查询之间的角度最大，所以相关度低；文档 2 与查询间的角度较小，所以更相关；文档 3 与查询的角度正好吻合，完全匹配。

在实际中，只有二维向量（两个词的查询）可以在平面上表示，幸运的是， 线性代数 ——作为数学中处理向量的一个分支——为我们提供了计算两个多维向量间角度工具，这意味着可以使用如上同样的方式来解释多个词的查询。

关于比较两个向量的更多信息可以参考 [余弦近似度（cosine similarity）](http://en.wikipedia.org/wiki/Cosine_similarity)。

现在已经讲完评分计算的基本理论，我们可以继续了解 Lucene 是如何实现评分计算的。

### Lucene 的实用评分函数[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/170_Relevance/15_Practical_scoring.asciidoc)

对于多词查询， Lucene 使用 [布尔模型（Boolean model）](https://www.elastic.co/guide/cn/elasticsearch/guide/current/scoring-theory.html#boolean-model) 、 [TF/IDF](https://www.elastic.co/guide/cn/elasticsearch/guide/current/scoring-theory.html#tfidf) 以及 [向量空间模型（vector space model）](https://www.elastic.co/guide/cn/elasticsearch/guide/current/scoring-theory.html#vector-space-model) ，然后将它们组合到单个高效的包里以收集匹配文档并进行评分计算。

一个多词查询

GET /my\_index/doc/\_search

{

"query": {

"match": {

"text": "quick fox"

}

}

}

会在内部被重写为：

GET /my\_index/doc/\_search

{

"query": {

"bool": {

"should": [

{"term": { "text": "quick" }},

{"term": { "text": "fox" }}

]

}

}

}

bool 查询实现了布尔模型，在这个例子中，它会将包括词 quick 和 fox 或两者兼有的文档作为查询结果。

只要一个文档与查询匹配，Lucene 就会为查询计算评分，然后合并每个匹配词的评分结果。这里使用的评分计算公式叫做 实用评分函数（practical scoring function） 。看似很高大上，但是别被吓到——多数的组件都已经介绍过，下一步会讨论它引入的一些新元素。

score(q,d) =

queryNorm(q)

· coord(q,d)

· ∑ (

tf(t in d)

· idf(t)²

· t.getBoost()

· norm(t,d)

) (t in q)

|  |  |
| --- | --- |
|  | score(q,d) 是文档 d 与查询 q 的相关度评分。 |
|  | queryNorm(q) 是 [查询归一化 因子](https://www.elastic.co/guide/cn/elasticsearch/guide/current/practical-scoring-function.html#query-norm) （新）。 |
|  | coord(q,d) 是 [协调 因子](https://www.elastic.co/guide/cn/elasticsearch/guide/current/practical-scoring-function.html#coord) （新）。 |
|  | 查询 q 中每个词 t 对于文档 d 的权重和。 |
|  | tf(t in d) 是词 t 在文档 d 中的 [词频](https://www.elastic.co/guide/cn/elasticsearch/guide/current/scoring-theory.html#tf) 。 |
|  | idf(t) 是词 t 的 [逆向文档频率](https://www.elastic.co/guide/cn/elasticsearch/guide/current/scoring-theory.html#idf) 。 |
|  | t.getBoost() 是查询中使用的 [boost](https://www.elastic.co/guide/cn/elasticsearch/guide/current/query-time-boosting.html)（新）。 |
|  | norm(t,d) 是 [字段长度归一值](https://www.elastic.co/guide/cn/elasticsearch/guide/current/scoring-theory.html#field-norm) ，与 [索引时字段层 boost](https://www.elastic.co/guide/cn/elasticsearch/guide/current/practical-scoring-function.html#index-boost) （如果存在）的和（新）。 |

上节已介绍过 score 、 tf 和 idf 。现在来介绍 queryNorm 、 coord 、 t.getBoost 和 norm 。

我们会在本章后面继续探讨 [查询时的权重提升](https://www.elastic.co/guide/cn/elasticsearch/guide/current/query-time-boosting.html) 的问题，但是首先需要了解查询归一化、协调和索引时字段层面的权重提升等概念。

#### 查询归一因子[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/170_Relevance/15_Practical_scoring.asciidoc)

查询归一因子 （ queryNorm ）试图将查询 归一化 ， 这样就能将两个不同的查询结果相比较。

尽管查询归一值的目的是为了使查询结果之间能够相互比较，但是它并不十分有效，因为相关度评分 \_score 的目的是为了将当前查询的结果进行排序，比较不同查询结果的相关度评分没有太大意义。

这个因子是在查询过程的最前面计算的，具体的计算依赖于具体查询，一个典型的实现如下：

queryNorm = 1 / √sumOfSquaredWeights

|  |  |
| --- | --- |
|  | sumOfSquaredWeights 是查询里每个词的 IDF 的平方和。 |

相同查询归一化因子会被应用到每个文档，不能被更改，总而言之，可以被忽略。

#### 查询协调[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/170_Relevance/15_Practical_scoring.asciidoc)

协调因子 （ coord ） 可以为那些查询词包含度高的文档提供奖励，文档里出现的查询词越多，它越有机会成为好的匹配结果。

设想查询 quick brown fox ，每个词的权重都是 1.5 。如果没有协调因子，最终评分会是文档里所有词权重的总和。例如：

* 文档里有 fox → 评分： 1.5
* 文档里有 quick fox → 评分： 3.0
* 文档里有 quick brown fox → 评分： 4.5

协调因子将评分与文档里匹配词的数量相乘，然后除以查询里所有词的数量，如果使用协调因子，评分会变成：

* 文档里有 fox → 评分： 1.5 \* 1 / 3 = 0.5
* 文档里有 quick fox → 评分： 3.0 \* 2 / 3 = 2.0
* 文档里有 quick brown fox → 评分： 4.5 \* 3 / 3 = 4.5

协调因子能使包含所有三个词的文档比只包含两个词的文档评分要高出很多。

回想将查询 quick brown fox 重写成 bool 查询的形式：

GET /\_search

{

"query": {

"bool": {

"should": [

{ "term": { "text": "quick" }},

{ "term": { "text": "brown" }},

{ "term": { "text": "fox" }}

]

}

}

}

bool 查询默认会对所有 should 语句使用协调功能，不过也可以将其禁用。为什么要这样做？通常的回答是——无须这样。查询协调通常是件好事，当使用 bool 查询将多个高级查询如 match 查询包裹的时候，让协调功能开启是有意义的，匹配的语句越多，查询请求与返回文档间的重叠度就越高。

但在某些高级应用中，将协调功能关闭可能更好。设想正在查找同义词 jump 、 leap 和 hop 时，并不关心会出现多少个同义词，因为它们都表示相同的意思，实际上，只有其中一个同义词会出现，这是不使用协调因子的一个好例子：

GET /\_search

{

"query": {

"bool": {

"disable\_coord": true,

"should": [

{ "term": { "text": "jump" }},

{ "term": { "text": "hop" }},

{ "term": { "text": "leap" }}

]

}

}

}

当使用同义词的时候（参照： [同义词](https://www.elastic.co/guide/cn/elasticsearch/guide/current/synonyms.html) ），Lucene 内部是这样的：重写的查询会禁用同义词的协调功能。大多数禁用操作的应用场景是自动处理的，无须为此担心。

#### 索引时字段层权重提升[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/170_Relevance/15_Practical_scoring.asciidoc)

我们会讨论 [查询时的权重提升](https://www.elastic.co/guide/cn/elasticsearch/guide/current/query-time-boosting.html)，让字段 权重提升 就是让某个字段比其他字段更重要。 当然在索引时也能做到如此。实际上，权重的提升会被应用到字段的每个词，而不是字段本身。

将提升值存储在索引中无须更多空间，这个字段层索引时的提升值与字段长度归一值（参见 [字段长度归一值](https://www.elastic.co/guide/cn/elasticsearch/guide/current/scoring-theory.html#field-norm) ）一起作为单个字节存于索引， norm(t,d) 是前面公式的返回值。

我们不建议在建立索引时对字段提升权重，有以下原因：

* 将提升值与字段长度归一值合在单个字节中存储会丢失字段长度归一值的精度，这样会导致 Elasticsearch 不知如何区分包含三个词的字段和包含五个词的字段。
* 要想改变索引时的提升值，就必须重新为所有文档建立索引，与此不同的是，查询时的提升值可以随着每次查询的不同而更改。
* 如果一个索引时权重提升的字段有多个值，提升值会按照每个值来自乘，这会导致该字段的权重急剧上升。

[查询时赋予权重](https://www.elastic.co/guide/cn/elasticsearch/guide/current/query-time-boosting.html) 是更为简单、清楚、灵活的选择。

了解了查询归一化、协同和索引时权重提升这些方式后，可以进一步了解相关度计算最有用的工具：查询时的权重提升。

### 查询时权重提升[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/170_Relevance/20_Query_time_boosting.asciidoc)

在 [语句优先级（Prioritizing Clauses）](https://www.elastic.co/guide/cn/elasticsearch/guide/current/multi-query-strings.html#prioritising-clauses) 中，我们解释过如何在搜索时使用 boost 参数让一个查询语句比其他语句更重要。 例如：

GET /\_search

{

"query": {

"bool": {

"should": [

{

"match": {

"title": {

"query": "quick brown fox",

"boost": 2 https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

}

}

},

{

"match": { https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

"content": "quick brown fox"

}

}

]

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/query-time-boosting.html#CO109-1) | title 查询语句的重要性是 content 查询的 2 倍，因为它的权重提升值为 2 。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/query-time-boosting.html#CO109-2) | 没有设置 boost 的查询语句的值为 1 。 |

查询时的权重提升 是可以用来影响相关度的主要工具，任意类型的查询都能接受 boost 参数。 将 boost设置为 2 ，并不代表最终的评分 \_score 是原值的两倍；实际的权重值会经过归一化和一些其他内部优化过程。尽管如此，它确实想要表明一个提升值为 2 的句子的重要性是提升值为 1 语句的两倍。

在实际应用中，无法通过简单的公式得出某个特定查询语句的 “正确” 权重提升值，只能通过不断尝试获得。需要记住的是 boost 只是影响相关度评分的其中一个因子；它还需要与其他因子相互竞争。在前例中， title 字段相对 content 字段可能已经有一个 “缺省的” 权重提升值，这因为在 [字段长度归一值](https://www.elastic.co/guide/cn/elasticsearch/guide/current/scoring-theory.html#field-norm) 中，标题往往比相关内容要短，所以不要想当然的去盲目提升一些字段的权重。选择权重，检查结果，如此反复。

#### 提升索引权重[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/170_Relevance/20_Query_time_boosting.asciidoc)

当在多个索引中搜索时， 可以使用参数 indices\_boost 来提升整个索引的权重，在下面例子中，当要为最近索引的文档分配更高权重时，可以这么做：

GET /docs\_2014\_\*/\_search https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

{

"indices\_boost": { https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

"docs\_2014\_10": 3,

"docs\_2014\_09": 2

},

"query": {

"match": {

"text": "quick brown fox"

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/query-time-boosting.html#CO110-1) | 这个多索引查询涵盖了所有以字符串 docs\_2014\_ 开始的索引。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/query-time-boosting.html#CO110-2) | 其中，索引 docs\_2014\_10 中的所有文件的权重是 3 ，索引 docs\_2014\_09 中是 2 ，其他所有匹配的索引权重为默认值 1 。 |

#### t.getBoost()[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/170_Relevance/20_Query_time_boosting.asciidoc)

这些提升值在 Lucene 的 [实用评分函数](https://www.elastic.co/guide/cn/elasticsearch/guide/current/practical-scoring-function.html) 中可以通过 t.getBoost() 获得。 权重提升不会被应用于它在查询表达式中出现的层，而是会被合并下转至每个词中。 t.getBoost() 始终返回当前词的权重或当前分析链上查询的权重。

提示

实际上，要想解读 [explain](https://www.elastic.co/guide/cn/elasticsearch/guide/current/relevance-intro.html#explain) 的输出是相当复杂的，在 explanation 里面完全看不到 boost值，也完全无法访问上面提到的 t.getBoost() 方法，权重值融合在 [queryNorm](https://www.elastic.co/guide/cn/elasticsearch/guide/current/practical-scoring-function.html#query-norm) 中并应用到每个词。尽管说， queryNorm 对于每个词都是相同的，还是会发现一个权重提升过的词的 queryNorm 值要高于一个没有提升过的。

### 使用查询结构修改相关度[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/170_Relevance/25_Query_scoring.asciidoc)

Elasticsearch 的查询表达式相当灵活， 可以通过调整查询结构中查询语句的所处层次，从而或多或少改变其重要性，比如，设想下面这个查询：

quick OR brown OR red OR fox

可以将所有词都放在 bool 查询的同一层中：

GET /\_search

{

"query": {

"bool": {

"should": [

{ "term": { "text": "quick" }},

{ "term": { "text": "brown" }},

{ "term": { "text": "red" }},

{ "term": { "text": "fox" }}

]

}

}

}

这个查询可能最终给包含 quick 、 red 和 brown 的文档评分与包含 quick 、 red 、 fox 文档的评分相同，这里 Red 和 brown 是同义词，可能只需要保留其中一个，而我们真正要表达的意思是想做以下查询：

quick OR (brown OR red) OR fox

根据标准的布尔逻辑，这与原始的查询是完全一样的，但是我们已经在 [组合查询（Combining Queries）](https://www.elastic.co/guide/cn/elasticsearch/guide/current/bool-query.html)中看到， bool 查询不关心文档匹配的 程度 ，只关心是否能匹配。

上述查询有个更好的方式：

GET /\_search

{

"query": {

"bool": {

"should": [

{ "term": { "text": "quick" }},

{ "term": { "text": "fox" }},

{

"bool": {

"should": [

{ "term": { "text": "brown" }},

{ "term": { "text": "red" }}

]

}

}

]

}

}

}

现在， red 和 brown 处于相互竞争的层次， quick 、 fox 以及 red OR brown 则是处于顶层且相互竞争的词。

我们已经讨论过如何使用 [match](https://www.elastic.co/guide/cn/elasticsearch/guide/current/match-query.html) 、[multi\_match](https://www.elastic.co/guide/cn/elasticsearch/guide/current/multi-match-query.html) 、[term](https://www.elastic.co/guide/cn/elasticsearch/guide/current/term-vs-full-text.html) 、[bool](https://www.elastic.co/guide/cn/elasticsearch/guide/current/bool-query.html) 和 [dis\_max](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_best_fields.html#dis-max-query) 查询修改相关度评分。本章后面的内容会介绍另外三个与相关度评分有关的查询： boosting 查询、 constant\_score 查询和 function\_score 查询。

### Not Quite Not[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/170_Relevance/30_Not_quite_not.asciidoc)

在互联网上搜索 “Apple”，返回的结果很可能是一个公司、水果和各种食谱。 我们可以在 bool 查询中用 must\_not 语句来排除像 pie 、 tart 、 crumble 和 tree 这样的词，从而将查询结果的范围缩小至只返回与 “Apple” （苹果）公司相关的结果：

GET /\_search

{

"query": {

"bool": {

"must": {

"match": {

"text": "apple"

}

},

"must\_not": {

"match": {

"text": "pie tart fruit crumble tree"

}

}

}

}

}

但谁又敢保证在排除 tree 或 crumble 这种词后，不会错失一个与苹果公司特别相关的文档呢？有时， must\_not 条件会过于严格。

#### 权重提升查询[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/170_Relevance/30_Not_quite_not.asciidoc)

[boosting 查询](https://www.elastic.co/guide/en/elasticsearch/reference/5.6/query-dsl-boosting-query.html) 恰恰能解决这个问题。 它仍然允许我们将关于水果或甜点的结果包括到结果中，但是使它们降级——即降低它们原来可能应有的排名：

GET /\_search

{

"query": {

"boosting": {

"positive": {

"match": {

"text": "apple"

}

},

"negative": {

"match": {

"text": "pie tart fruit crumble tree"

}

},

"negative\_boost": 0.5

}

}

}

它接受 positive 和 negative 查询。只有那些匹配 positive 查询的文档罗列出来，对于那些同时还匹配 negative 查询的文档将通过文档的原始 \_score 与 negative\_boost 相乘的方式降级后的结果。

为了达到效果， negative\_boost 的值必须小于 1.0 。在这个示例中，所有包含负向词的文档评分 \_score都会减半。

### 忽略 TF/IDF[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/170_Relevance/35_Ignoring_TFIDF.asciidoc)

有时候我们根本不关心 TF/IDF ， 只想知道一个词是否在某个字段中出现过。可能搜索一个度假屋并希望它能尽可能有以下设施：

* WiFi
* Garden（花园）
* Pool（游泳池）

这个度假屋的文档如下：

{ "description": "A delightful four-bedroomed house with ... " }

可以用简单的 match 查询进行匹配：

GET /\_search

{

"query": {

"match": {

"description": "wifi garden pool"

}

}

}

但这并不是真正的 全文搜索 ，此种情况下，TF/IDF 并无用处。我们既不关心 wifi 是否为一个普通词，也不关心它在文档中出现是否频繁，关心的只是它是否曾出现过。实际上，我们希望根据房屋不同设施的数量对其排名——设施越多越好。如果设施出现，则记 1 分，不出现记 0 分。

#### constant\_score 查询[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/170_Relevance/35_Ignoring_TFIDF.asciidoc)

在 [constant\_score](https://www.elastic.co/guide/en/elasticsearch/reference/5.6/query-dsl-constant-score-query.html) 查询中，它可以包含查询或过滤，为任意一个匹配的文档指定评分 1 ，忽略 TF/IDF 信息：

GET /\_search

{

"query": {

"bool": {

"should": [

{ "constant\_score": {

"query": { "match": { "description": "wifi" }}

}},

{ "constant\_score": {

"query": { "match": { "description": "garden" }}

}},

{ "constant\_score": {

"query": { "match": { "description": "pool" }}

}}

]

}

}

}

或许不是所有的设施都同等重要——对某些用户来说有些设施更有价值。如果最重要的设施是游泳池，那我们可以为更重要的设施增加权重：

GET /\_search

{

"query": {

"bool": {

"should": [

{ "constant\_score": {

"query": { "match": { "description": "wifi" }}

}},

{ "constant\_score": {

"query": { "match": { "description": "garden" }}

}},

{ "constant\_score": {

"boost": 2 https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

"query": { "match": { "description": "pool" }}

}}

]

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/ignoring-tfidf.html#CO111-1) | pool 语句的权重提升值为 2 ，而其他的语句为 1 。 |

注意

最终的评分并不是所有匹配语句的简单求和， [协调因子（coordination factor）](https://www.elastic.co/guide/cn/elasticsearch/guide/current/practical-scoring-function.html#coord) 和 [查询归一化因子（query normalization factor）](https://www.elastic.co/guide/cn/elasticsearch/guide/current/practical-scoring-function.html#query-norm) 仍然会被考虑在内。

我们可以给 features 字段加上 not\_analyzed 类型来提升度假屋文档的匹配能力：

{ "features": [ "wifi", "pool", "garden" ] }

默认情况下，一个 not\_analyzed 字段会禁用 [字段长度归一值（field-length norms）](https://www.elastic.co/guide/cn/elasticsearch/guide/current/scoring-theory.html#field-norm) 的功能， 并将 index\_options 设为 docs 选项，禁用 [词频](https://www.elastic.co/guide/cn/elasticsearch/guide/current/scoring-theory.html#tf) ，但还是存在问题：每个词的 [倒排文档频率](https://www.elastic.co/guide/cn/elasticsearch/guide/current/scoring-theory.html#idf) 仍然会被考虑。

可以采用与之前相同的方法 constant\_score 查询来解决这个问题：

GET /\_search

{

"query": {

"bool": {

"should": [

{ "constant\_score": {

"query": { "match": { "features": "wifi" }}

}},

{ "constant\_score": {

"query": { "match": { "features": "garden" }}

}},

{ "constant\_score": {

"boost": 2

"query": { "match": { "features": "pool" }}

}}

]

}

}

}

实际上，每个设施都应该看成一个过滤器，对于度假屋来说要么具有某个设施要么没有——过滤器因为其性质天然合适。而且，如果使用过滤器，我们还可以利用缓存。

这里的问题是：过滤器无法计算评分。这样就需要寻求一种方式将过滤器和查询间的差异抹平。 function\_score 查询不仅正好可以扮演这个角色，而且有更强大的功能。

### function\_score 查询[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/170_Relevance/40_Function_score_query.asciidoc)

[function\_score 查询](https://www.elastic.co/guide/en/elasticsearch/reference/5.6/query-dsl-function-score-query.html) 是用来控制评分过程的终极武器，它允许为每个与主查询匹配的文档应用一个函数，以达到改变甚至完全替换原始查询评分 \_score 的目的。

实际上，也能用过滤器对结果的 子集 应用不同的函数，这样一箭双雕：既能高效评分，又能利用过滤器缓存。

Elasticsearch 预定义了一些函数：

weight

为每个文档应用一个简单而不被规范化的权重提升值：当 weight 为 2 时，最终结果为 2 \* \_score 。

field\_value\_factor

使用这个值来修改 \_score ，如将 popularity 或 votes （受欢迎或赞）作为考虑因素。

random\_score

为每个用户都使用一个不同的随机评分对结果排序，但对某一具体用户来说，看到的顺序始终是一致的。

衰减函数 —— linear 、 exp 、 gauss

将浮动值结合到评分 \_score 中，例如结合 publish\_date 获得最近发布的文档，结合 geo\_location 获得更接近某个具体经纬度（lat/lon）地点的文档，结合 price 获得更接近某个特定价格的文档。

script\_score

如果需求超出以上范围时，用自定义脚本可以完全控制评分计算，实现所需逻辑。

如果没有 function\_score 查询，就不能将全文查询与最新发生这种因子结合在一起评分，而不得不根据评分 \_score 或时间 date 进行排序；这会相互影响抵消两种排序各自的效果。这个查询可以使两个效果融合：可以仍然根据全文相关度进行排序，但也会同时考虑最新发布文档、流行文档、或接近用户希望价格的产品。正如所设想的，查询要考虑所有这些因素会非常复杂，让我们先从简单的例子开始，然后顺着梯子慢慢向上爬，增加复杂度。

### 按受欢迎度提升权重[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/170_Relevance/45_Popularity.asciidoc)

设想有个网站供用户发布博客并且可以让他们为自己喜欢的博客点赞， 我们希望将更受欢迎的博客放在搜索结果列表中相对较上的位置，同时全文搜索的评分仍然作为相关度的主要排序依据，可以简单的通过存储每个博客的点赞数来实现它：

PUT /blogposts/post/1

{

"title": "About popularity",

"content": "In this post we will talk about...",

"votes": 6

}

在搜索时，可以将 function\_score 查询与 field\_value\_factor 结合使用， 即将点赞数与全文相关度评分结合：

GET /blogposts/post/\_search

{

"query": {

"function\_score": { https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

"query": { https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

"multi\_match": {

"query": "popularity",

"fields": [ "title", "content" ]

}

},

"field\_value\_factor": { https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png

"field": "votes" https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/4.png

}

}

}

}

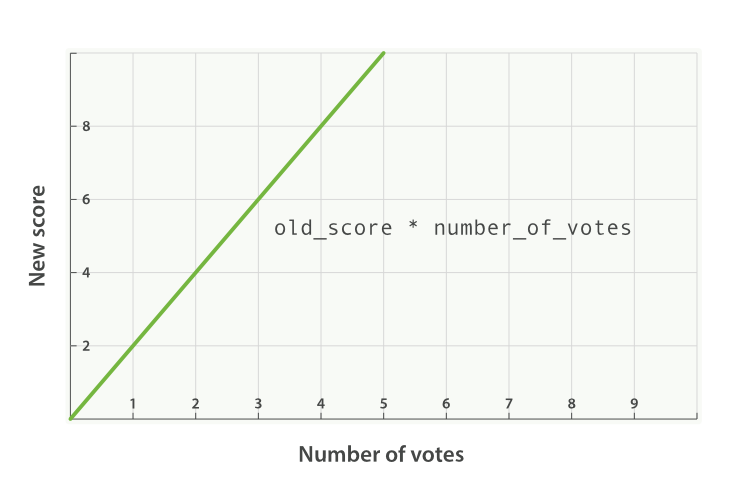
|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/boosting-by-popularity.html#CO112-1) | function\_score 查询将主查询和函数包括在内。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/boosting-by-popularity.html#CO112-2) | 主查询优先执行。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/boosting-by-popularity.html#CO112-3) | field\_value\_factor 函数会被应用到每个与主 query 匹配的文档。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/4.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/boosting-by-popularity.html#CO112-4) | 每个文档的 votes 字段都 必须 有值供 function\_score 计算。如果 没有 文档的 votes 字段有值，那么就 必须 使用 [missing 属性](https://www.elastic.co/guide/en/elasticsearch/reference/5.6/query-dsl-function-score-query.html#function-field-value-factor) 提供的默认值来进行评分计算。 |

在前面示例中，每个文档的最终评分 \_score 都做了如下修改：

new\_score = old\_score \* number\_of\_votes

然而这并不会带来出人意料的好结果，全文评分 \_score 通常处于 0 到 10 之间，如下图 [图 29 “受欢迎度的线性关系基于 \_score 的原始值 2.0”](https://www.elastic.co/guide/cn/elasticsearch/guide/current/boosting-by-popularity.html#img-popularity-linear) 中，有 10 个赞的博客会掩盖掉全文评分，而 0 个赞的博客的评分会被置为 0 。

**图 29. 受欢迎度的线性关系基于 \_score 的原始值 2.0**



#### modifier[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/170_Relevance/45_Popularity.asciidoc)

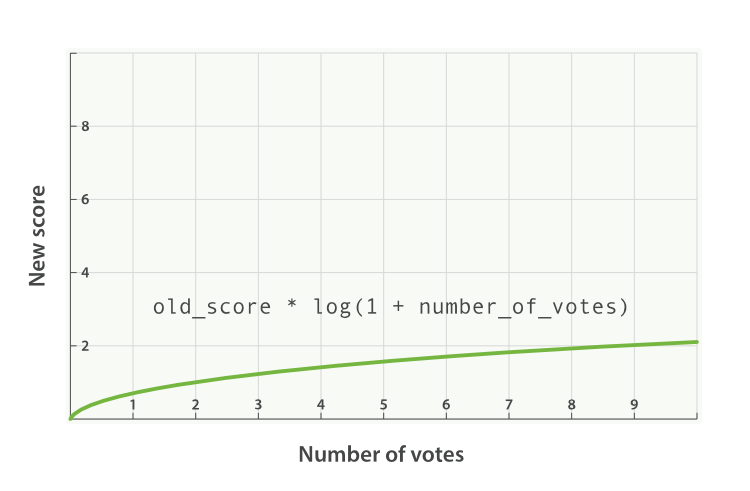
一种融入受欢迎度更好方式是用 modifier 平滑 votes 的值。 换句话说，我们希望最开始的一些赞更重要，但是其重要性会随着数字的增加而降低。 0 个赞与 1 个赞的区别应该比 10 个赞与 11 个赞的区别大很多。

对于上述情况，典型的 modifier 应用是使用 log1p 参数值，公式如下：

new\_score = old\_score \* log(1 + number\_of\_votes)

log 对数函数使 votes 赞字段的评分曲线更平滑，如图 [图 30 “受欢迎度的对数关系基于 \_score 的原始值 2.0”](https://www.elastic.co/guide/cn/elasticsearch/guide/current/boosting-by-popularity.html#img-popularity-log) ：

**图 30. 受欢迎度的对数关系基于 \_score 的原始值 2.0**



带 modifier 参数的请求如下：

GET /blogposts/post/\_search

{

"query": {

"function\_score": {

"query": {

"multi\_match": {

"query": "popularity",

"fields": [ "title", "content" ]

}

},

"field\_value\_factor": {

"field": "votes",

"modifier": "log1p" https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

}

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/boosting-by-popularity.html#CO113-1) | modifier 为 log1p 。 |

修饰语 modifier 的值可以为： none （默认状态）、 log 、 log1p 、 log2p 、 ln 、 ln1p 、 ln2p 、 square 、 sqrt 以及 reciprocal 。想要了解更多信息请参照： [field\_value\_factor 文档](https://www.elastic.co/guide/en/elasticsearch/reference/5.6/query-dsl-function-score-query.html#function-field-value-factor).

#### factor[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/170_Relevance/45_Popularity.asciidoc)

可以通过将 votes 字段与 factor 的积来调节受欢迎程度效果的高低：

GET /blogposts/post/\_search

{

"query": {

"function\_score": {

"query": {

"multi\_match": {

"query": "popularity",

"fields": [ "title", "content" ]

}

},

"field\_value\_factor": {

"field": "votes",

"modifier": "log1p",

"factor": 2 https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

}

}

}

}

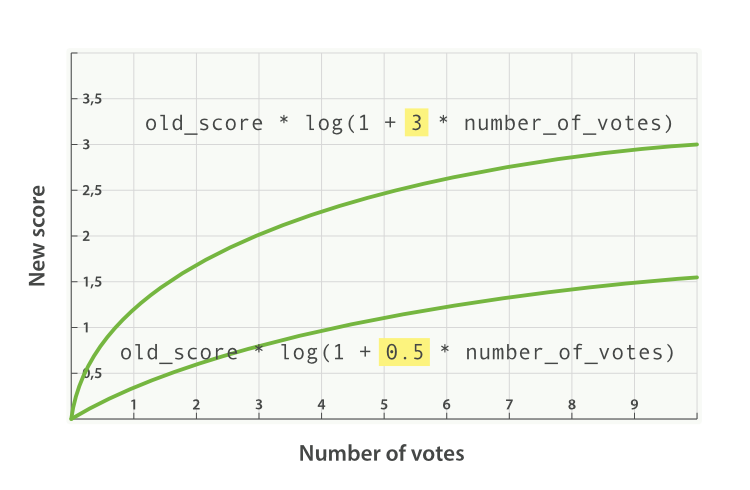
|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/boosting-by-popularity.html#CO114-1) | 双倍效果。 |

添加了 factor 会使公式变成这样：

new\_score = old\_score \* log(1 + factor \* number\_of\_votes)

factor 值大于 1 会提升效果， factor 值小于 1 会降低效果，如图 [图 31 “受欢迎度的对数关系基于多个不同因子”](https://www.elastic.co/guide/cn/elasticsearch/guide/current/boosting-by-popularity.html#img-popularity-factor) 。

**图 31. 受欢迎度的对数关系基于多个不同因子**



#### boost\_mode[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/170_Relevance/45_Popularity.asciidoc)

或许将全文评分与 field\_value\_factor 函数值乘积的效果仍然可能太大， 我们可以通过参数 boost\_mode来控制函数与查询评分 \_score 合并后的结果，参数接受的值为：

multiply

评分 \_score 与函数值的积（默认）

sum

评分 \_score 与函数值的和

min

评分 \_score 与函数值间的较小值

max

评分 \_score 与函数值间的较大值

replace

函数值替代评分 \_score

与使用乘积的方式相比，使用评分 \_score 与函数值求和的方式可以弱化最终效果，特别是使用一个较小 factor 因子时：

GET /blogposts/post/\_search

{

"query": {

"function\_score": {

"query": {

"multi\_match": {

"query": "popularity",

"fields": [ "title", "content" ]

}

},

"field\_value\_factor": {

"field": "votes",

"modifier": "log1p",

"factor": 0.1

},

"boost\_mode": "sum" https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

}

}

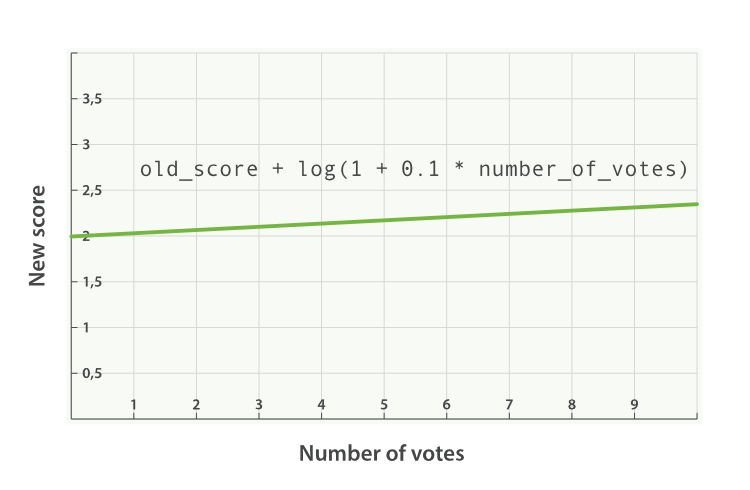
}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/boosting-by-popularity.html#CO115-1) | 评分 \_score 与函数值的积。 |

之前请求的公式现在变成下面这样（参见 [图 32 “使用 sum 结合受欢迎程度”](https://www.elastic.co/guide/cn/elasticsearch/guide/current/boosting-by-popularity.html#img-popularity-sum) ）：

new\_score = old\_score + log(1 + 0.1 \* number\_of\_votes)

**图 32. 使用 sum 结合受欢迎程度**



#### max\_boost[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/170_Relevance/45_Popularity.asciidoc)

最后，可以使用 max\_boost 参数限制一个函数的最大效果：

GET /blogposts/post/\_search

{

"query": {

"function\_score": {

"query": {

"multi\_match": {

"query": "popularity",

"fields": [ "title", "content" ]

}

},

"field\_value\_factor": {

"field": "votes",

"modifier": "log1p",

"factor": 0.1

},

"boost\_mode": "sum",

"max\_boost": 1.5 https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/boosting-by-popularity.html#CO116-1) | 无论 field\_value\_factor 函数的结果如何，最终结果都不会大于 1.5 。 |

注意

max\_boost 只对函数的结果进行限制，不会对最终评分 \_score 产生直接影响。

### 过滤集提升权重[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/170_Relevance/50_Boosting_filtered_subsets.asciidoc)

回到 [忽略 TF/IDF](https://www.elastic.co/guide/cn/elasticsearch/guide/current/ignoring-tfidf.html) 里处理过的问题，我们希望根据每个度假屋的特性数量来评分， 当时我们希望能用缓存的过滤器来影响评分，现在 function\_score 查询正好可以完成这件事情。

到目前为止，我们展现的都是为所有文档应用单个函数的使用方式，现在会用过滤器将结果划分为多个子集（每个特性一个过滤器），并为每个子集使用不同的函数。

在下面例子中，我们会使用 weight 函数，它与 boost 参数类似可以用于任何查询。有一点区别是 weight没有被 Luence 归一化成难以理解的浮点数，而是直接被应用。

查询的结构需要做相应变更以整合多个函数：

GET /\_search

{

"query": {

"function\_score": {

"filter": { https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

"term": { "city": "Barcelona" }

},

"functions": [ https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

{

"filter": { "term": { "features": "wifi" }}, https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png

"weight": 1

},

{

"filter": { "term": { "features": "garden" }}, https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/4.png

"weight": 1

},

{

"filter": { "term": { "features": "pool" }}, https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/5.png

"weight": 2 https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/6.png

}

],

"score\_mode": "sum", https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/7.png

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/function-score-filters.html#CO117-1) | function\_score 查询有个 filter 过滤器而不是 query 查询。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/function-score-filters.html#CO117-2) | functions 关键字存储着一个将被应用的函数列表。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/function-score-filters.html#CO117-3) [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/4.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/function-score-filters.html#CO117-4)[https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/5.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/function-score-filters.html#CO117-5) | 函数会被应用于和 filter 过滤器（可选的）匹配的文档。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/6.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/function-score-filters.html#CO117-6) | pool 比其他特性更重要，所以它有更高 weight 。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/7.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/function-score-filters.html#CO117-7) | score\_mode 指定各个函数的值进行组合运算的方式。 |

这个新特性需要注意的地方会在以下小节介绍。

#### 过滤 vs. 查询[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/170_Relevance/50_Boosting_filtered_subsets.asciidoc)

首先要注意的是 filter 过滤器代替了 query 查询， 在本例中，我们无须使用全文搜索，只想找到 city字段中包含 Barcelona 的所有文档，逻辑用过滤比用查询表达更清晰。过滤器返回的所有文档的评分 \_score 的值为 1 。 function\_score 查询接受 query 或 filter ，如果没有特别指定，则默认使用 match\_all 查询。

#### 函数 functions[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/170_Relevance/50_Boosting_filtered_subsets.asciidoc)

functions 关键字保持着一个将要被使用的函数列表。 可以为列表里的每个函数都指定一个 filter 过滤器，在这种情况下，函数只会被应用到那些与过滤器匹配的文档，例子中，我们为与过滤器匹配的文档指定权重值 weight 为 1 （为与 pool 匹配的文档指定权重值为 2 ）。

#### 评分模式 score\_mode[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/170_Relevance/50_Boosting_filtered_subsets.asciidoc)

每个函数返回一个结果，所以需要一种将多个结果缩减到单个值的方式，然后才能将其与原始评分 \_score合并。评分模式 score\_mode 参数正好扮演这样的角色， 它接受以下值：

multiply

函数结果求积（默认）。

sum

函数结果求和。

avg

函数结果的平均值。

max

函数结果的最大值。

min

函数结果的最小值。

first

使用首个函数（可以有过滤器，也可能没有）的结果作为最终结果

在本例中，我们将每个过滤器匹配结果的权重 weight 求和，并将其作为最终评分结果，所以会使用 sum评分模式。

不与任何过滤器匹配的文档会保有其原始评分， \_score 值的为 1 。

### 随机评分[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/170_Relevance/55_Random_scoring.asciidoc)

你可能会想知道 一致随机评分（consistently random scoring） 是什么，又为什么会使用它。 之前的例子是个很好的应用场景，前例中所有的结果都会返回 1 、 2 、 3 、 4 或 5 这样的最终评分 \_score ，可能只有少数房子的评分是 5 分，而有大量房子的评分是 2 或 3 。

作为网站的所有者，总会希望让广告有更高的展现率。在当前查询下，有相同评分 \_score 的文档会每次都以相同次序出现，为了提高展现率，在此引入一些随机性可能会是个好主意，这能保证有相同评分的文档都能有均等相似的展现机率。

我们想让每个用户看到不同的随机次序，但也同时希望如果是同一用户翻页浏览时，结果的相对次序能始终保持一致。这种行为被称为 一致随机（consistently random） 。

random\_score 函数会输出一个 0 到 1 之间的数， 当种子 seed 值相同时，生成的随机结果是一致的，例如，将用户的会话 ID 作为 seed ：

GET /\_search

{

"query": {

"function\_score": {

"filter": {

"term": { "city": "Barcelona" }

},

"functions": [

{

"filter": { "term": { "features": "wifi" }},

"weight": 1

},

{

"filter": { "term": { "features": "garden" }},

"weight": 1

},

{

"filter": { "term": { "features": "pool" }},

"weight": 2

},

{

"random\_score": { https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

"seed": "the users session id" https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

}

}

],

"score\_mode": "sum"

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/random-scoring.html#CO118-1) | random\_score 语句没有任何过滤器 filter ，所以会被应用到所有文档。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/random-scoring.html#CO118-2) | 将用户的会话 ID 作为种子 seed ，让该用户的随机始终保持一致，相同的种子 seed 会产生相同的随机结果。 |

当然，如果增加了与查询匹配的新文档，无论是否使用一致随机，其结果顺序都会发生变化。

### 越近越好[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/170_Relevance/60_Decay_functions.asciidoc)

很多变量都可以影响用户对于度假屋的选择， 也许用户希望离市中心近点，但如果价格足够便宜，也有可能选择一个更远的住处，也有可能反过来是正确的：愿意为最好的位置付更多的价钱。

如果我们添加过滤器排除所有市中心方圆 1 千米以外的度假屋，或排除所有每晚价格超过 £100 英镑的，我们可能会将用户愿意考虑妥协的那些选择排除在外。

function\_score 查询会提供一组 衰减函数（decay functions） ， 让我们有能力在两个滑动标准，如地点和价格，之间权衡。

有三种衰减函数—— linear 、 exp 和 gauss （线性、指数和高斯函数），它们可以操作数值、时间以及经纬度地理坐标点这样的字段。所有三个函数都能接受以下参数：

origin

中心点 或字段可能的最佳值，落在原点 origin 上的文档评分 \_score 为满分 1.0 。

scale

衰减率，即一个文档从原点 origin 下落时，评分 \_score 改变的速度。（例如，每 £10 欧元或每 100 米）。

decay

从原点 origin 衰减到 scale 所得的评分 \_score ，默认值为 0.5 。

offset

以原点 origin 为中心点，为其设置一个非零的偏移量 offset 覆盖一个范围，而不只是单个原点。在范围 -offset <= origin <= +offset 内的所有评分 \_score 都是 1.0 。

这三个函数的唯一区别就是它们衰减曲线的形状，用图来说明会更为直观（参见 [图 33 “衰减函数曲线”](https://www.elastic.co/guide/cn/elasticsearch/guide/current/decay-functions.html#img-decay-functions)）。

**图 33. 衰减函数曲线**

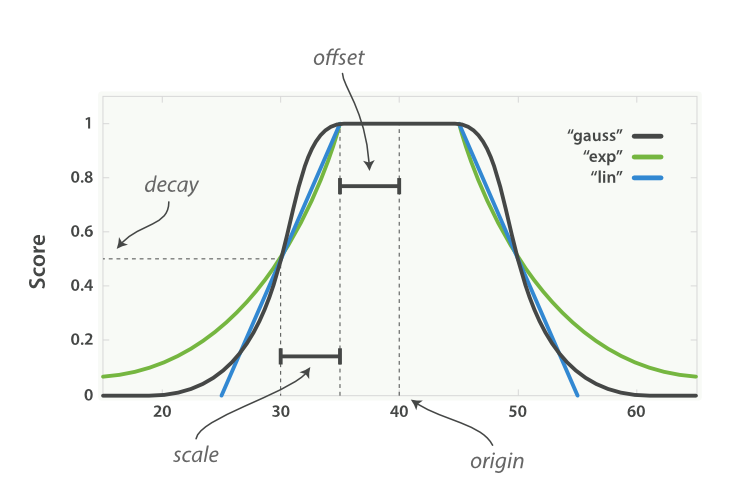


图 [图 33 “衰减函数曲线”](https://www.elastic.co/guide/cn/elasticsearch/guide/current/decay-functions.html#img-decay-functions) 中所有曲线的原点 origin （即中心点）的值都是 40 ， offset 是 5 ，也就是在范围 40 - 5 <= value <= 40 + 5 内的所有值都会被当作原点 origin 处理——所有这些点的评分都是满分 1.0 。

在此范围之外，评分开始衰减，衰减率由 scale 值（此例中的值为 5 ）和 衰减值 decay （此例中为默认值 0.5 ）共同决定。结果是所有三个曲线在 origin +/- (offset + scale) 处的评分都是 0.5 ，即点 30和 50 处。

linear 、 exp 和 gauss （线性、指数和高斯）函数三者之间的区别在于范围（ origin +/- (offset + scale) ）之外的曲线形状：

* linear 线性函数是条直线，一旦直线与横轴 0 相交，所有其他值的评分都是 0.0 。
* exp 指数函数是先剧烈衰减然后变缓。
* gauss 高斯函数是钟形的——它的衰减速率是先缓慢，然后变快，最后又放缓。

选择曲线的依据完全由期望评分 \_score 的衰减速率来决定，即距原点 origin 的值。

回到我们的例子：用户希望租一个离伦敦市中心近（ { "lat": 51.50, "lon": 0.12} ）且每晚不超过 £100 英镑的度假屋，而且与距离相比， 我们的用户对价格更为敏感，这样查询可以写成：

GET /\_search

{

"query": {

"function\_score": {

"functions": [

{

"gauss": {

"location": { https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

"origin": { "lat": 51.5, "lon": 0.12 },

"offset": "2km",

"scale": "3km"

}

}

},

{

"gauss": {

"price": { https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

"origin": "50", https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png

"offset": "50",

"scale": "20"

}

},

"weight": 2 https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/4.png

}

]

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/decay-functions.html#CO119-1) | location 字段以地理坐标点 geo\_point 映射。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/decay-functions.html#CO119-2) | price 字段是数值。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/decay-functions.html#CO119-3) | 参见 [理解价格语句](https://www.elastic.co/guide/cn/elasticsearch/guide/current/Understanding-the-price-Clause.html) ，理解 origin 为什么是 50 而不是 100 。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/4.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/decay-functions.html#CO119-4) | price 语句是 location 语句权重的两倍。 |

location 语句可以简单理解为：

* 以伦敦市中作为原点 origin 。
* 所有距原点 origin 2km 范围内的位置的评分是 1.0 。
* 距中心 5km （ offset + scale ）的位置的评分是 0.5 。
* [«  越近越好](https://www.elastic.co/guide/cn/elasticsearch/guide/current/decay-functions.html)    [脚本评分  »](https://www.elastic.co/guide/cn/elasticsearch/guide/current/script-score.html)

### 理解 price 价格语句[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/170_Relevance/60_Decay_functions.asciidoc)

* price 语句使用了一个小技巧：用户希望选择 £100 英镑以下的度假屋，但是例子中的原点被设置成 £50 英镑，价格不能为负，但肯定是越低越好，所以 £0 到 £100 英镑内的所有价格都认为是比较好的。
* 如果我们将原点 origin 被设置成 £100 英镑，那么低于 £100 英镑的度假屋的评分会变低，与其这样不如将原点 origin 和偏移量 offset 同时设置成 £50 英镑，这样就能使只有在价格高于 £100 英镑（ origin + offset ）时评分才会变低。
* 提示
* weight 参数可以被用来调整每个语句的贡献度，权重 weight 的默认值是 1.0 。这个值会先与每个句子的评分相乘，然后再通过 score\_mode 的设置方式合并。

### 脚本评分[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/170_Relevance/65_Script_score.asciidoc)

最后，如果所有 function\_score 内置的函数都无法满足应用场景，可以使用 script\_score 函数自行实现逻辑。

举个例子，想将利润空间作为因子加入到相关度评分计算，在业务中，利润空间和以下三点相关：

* price 度假屋每晚的价格。
* 会员用户的级别——某些等级的用户可以在每晚房价高于某个 threshold 阀值价格的时候享受折扣 discount 。
* 用户享受折扣后，经过议价的每晚房价的利润 margin 。

计算每个度假屋利润的算法如下：

if (price < threshold) {

profit = price \* margin

} else {

profit = price \* (1 - discount) \* margin;

}

我们很可能不想用绝对利润作为评分，这会弱化其他如地点、受欢迎度和特性等因子的作用，而是将利润用目标利润 target 的百分比来表示，高于 目标的利润空间会有一个正向评分（大于 1.0 ），低于目标的利润空间会有一个负向分数（小于 1.0 ）：

if (price < threshold) {

profit = price \* margin

} else {

profit = price \* (1 - discount) \* margin

}

return profit / target

Elasticsearch 里使用 [Groovy](http://groovy.codehaus.org/) 作为默认的脚本语言，它与JavaScript很像， 上面这个算法用 Groovy 脚本表示如下：

price = doc['price'].value https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

margin = doc['margin'].value https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

if (price < threshold) { https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png

return price \* margin / target

}

return price \* (1 - discount) \* margin / target https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/4.png

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/script-score.html#CO120-1) [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/script-score.html#CO120-2) | price 和 margin 变量可以分别从文档的 price 和 margin 字段提取。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/script-score.html#CO120-3) [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/4.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/script-score.html#CO120-4) | threshold 、 discount 和 target 是作为参数 params 传入的。 |

最终我们将 script\_score 函数与其他函数一起使用：

GET /\_search

{

"function\_score": {

"functions": [

{ ...location clause... }, https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

{ ...price clause... }, https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

{

"script\_score": {

"params": { https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png

"threshold": 80,

"discount": 0.1,

"target": 10

},

"script": "price = doc['price'].value; margin = doc['margin'].value;

if (price < threshold) { return price \* margin / target };

return price \* (1 - discount) \* margin / target;" https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/4.png

}

}

]

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/script-score.html#CO121-1) [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/script-score.html#CO121-2) | location 和 price 语句在 [衰减函数](https://www.elastic.co/guide/cn/elasticsearch/guide/current/decay-functions.html) 中解释过。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/script-score.html#CO121-3) | 将这些变量作为参数 params 传递，我们可以查询时动态改变脚本无须重新编译。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/4.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/script-score.html#CO121-4) | JSON 不能接受内嵌的换行符，脚本中的换行符可以用 \n 或 ; 符号替代。 |

这个查询根据用户对地点和价格的需求，返回用户最满意的文档，同时也考虑到我们对于盈利的要求。

提示

script\_score 函数提供了巨大的灵活性， 可以通过脚本访问文档里的所有字段、当前评分 \_score 甚至词频、逆向文档频率和字段长度规范值这样的信息（参见 see [脚本对文本评分](https://www.elastic.co/guide/en/elasticsearch/reference/5.6/modules-advanced-scripting.html)）。

有人说使用脚本对性能会有影响，如果确实发现脚本执行较慢，可以有以下三种选择：

* 尽可能多的提前计算各种信息并将结果存入每个文档中。
* Groovy 很快，但没 Java 快。 可以将脚本用原生的 Java 脚本重新实现。（参见 [原生 Java 脚本](https://www.elastic.co/guide/en/elasticsearch/reference/5.6/modules-scripting-native.html)）。
* 仅对那些最佳评分的文档应用脚本，使用 [重新评分](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_Improving_Performance.html#rescore-api) 中提到的 rescore 功能。

### 可插拔的相似度算法[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/170_Relevance/70_Pluggable_similarities.asciidoc)

在进一步讨论相关度和评分之前，我们会以一个更高级的话题结束本章节的内容：可插拔的相似度算法（Pluggable Similarity Algorithms）。 Elasticsearch 将 [实用评分算法](https://www.elastic.co/guide/cn/elasticsearch/guide/current/practical-scoring-function.html) 作为默认相似度算法，它也能够支持其他的一些算法，这些算法可以参考 [相似度模块](https://www.elastic.co/guide/en/elasticsearch/reference/5.6/index-modules-similarity.html#configuration) 文档。

#### Okapi BM25[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/170_Relevance/70_Pluggable_similarities.asciidoc)

能与 TF/IDF 和向量空间模型媲美的就是 [Okapi BM25](http://en.wikipedia.org/wiki/Okapi_BM25) ，它被认为是 当今最先进的 排序函数。 BM25 源自 [概率相关模型（probabilistic relevance model）](http://en.wikipedia.org/wiki/Probabilistic_relevance_model) ，而不是向量空间模型，但这个算法也和 Lucene 的实用评分函数有很多共通之处。

BM25 同样使用词频、逆向文档频率以及字段长归一化，但是每个因子的定义都有细微区别。与其详细解释 BM25 公式，倒不如将关注点放在 BM25 所能带来的实际好处上。

#### 词频饱和度[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/170_Relevance/70_Pluggable_similarities.asciidoc)

TF/IDF 和 BM25 同样使用 [逆向文档频率](https://www.elastic.co/guide/cn/elasticsearch/guide/current/scoring-theory.html#idf) 来区分普通词（不重要）和非普通词（重要）， 同样认为（参见 [词频](https://www.elastic.co/guide/cn/elasticsearch/guide/current/scoring-theory.html#tf) ）文档里的某个词出现次数越频繁，文档与这个词就越相关。

不幸的是，普通词随处可见， 实际上一个普通词在同一个文档中大量出现的作用会由于该词在 所有 文档中的大量出现而被抵消掉。

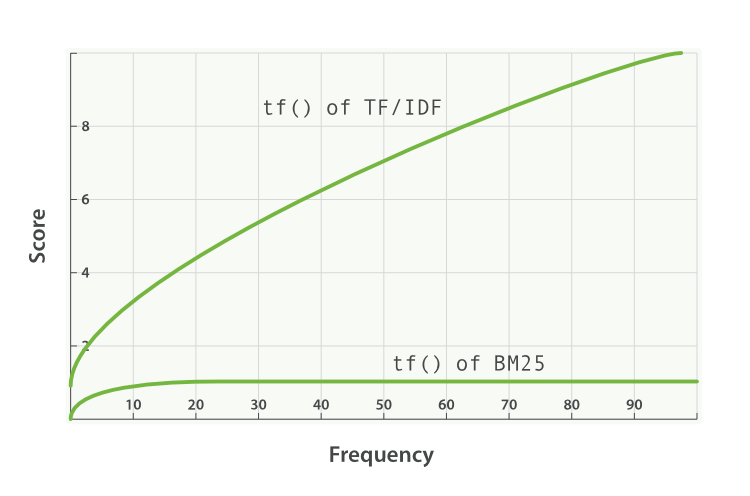
曾经有个时期，将 最 普通的词（或 停用词 ，参见 [停用词](https://www.elastic.co/guide/cn/elasticsearch/guide/current/stopwords.html)）从索引中移除被认为是一种标准实践， TF/IDF 正是在这种背景下诞生的。TF/IDF 没有考虑词频上限的问题，因为高频停用词已经被移除了。

Elasticsearch 的 standard 标准分析器（ string 字段默认使用）不会移除停用词，因为尽管这些词的重要性很低，但也不是毫无用处。这导致：在一个相当长的文档中，像 the 和 and 这样词出现的数量会高得离谱，以致它们的权重被人为放大。

另一方面，BM25 有一个上限，文档里出现 5 到 10 次的词会比那些只出现一两次的对相关度有着显著影响。但是如图 [TF/IDF 与 BM25 的词频饱和度](https://www.elastic.co/guide/cn/elasticsearch/guide/current/pluggable-similarites.html#img-bm25-saturation) 所见，文档中出现 20 次的词几乎与那些出现上千次的词有着相同的影响。

这就是 非线性词频饱和度（nonlinear term-frequency saturation） 。

**图 34. TF/IDF 与 BM25 的词频饱和度**



#### 字段长度归一化（Field-length normalization）[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/170_Relevance/70_Pluggable_similarities.asciidoc)

在 [字段长归一化](https://www.elastic.co/guide/cn/elasticsearch/guide/current/scoring-theory.html#field-norm) 中，我们提到过 Lucene 会认为较短字段比较长字段更重要：字段某个词的频度所带来的重要性会被这个字段长度抵消，但是实际的评分函数会将所有字段以同等方式对待。它认为所有较短的 title 字段比所有较长的 body 字段更重要。

BM25 当然也认为较短字段应该有更多的权重，但是它会分别考虑每个字段内容的平均长度，这样就能区分短 title 字段和 长 title 字段。

小心

在 [查询时权重提升](https://www.elastic.co/guide/cn/elasticsearch/guide/current/query-time-boosting.html) 中，已经说过 title 字段因为其长度比 body 字段 自然 有更高的权重提升值。由于字段长度的差异只能应用于单字段，这种自然的权重提升会在使用 BM25 时消失。

#### BM25 调优[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/170_Relevance/70_Pluggable_similarities.asciidoc)

不像 TF/IDF ，BM25 有一个比较好的特性就是它提供了两个可调参数：

k1

这个参数控制着词频结果在词频饱和度中的上升速度。默认值为 1.2 。值越小饱和度变化越快，值越大饱和度变化越慢。

b

这个参数控制着字段长归一值所起的作用， 0.0 会禁用归一化， 1.0 会启用完全归一化。默认值为 0.75 。

在实践中，调试 BM25 是另外一回事， k1 和 b 的默认值适用于绝大多数文档集合，但最优值还是会因为文档集不同而有所区别，为了找到文档集合的最优值，就必须对参数进行反复修改验证。

### 更改相似度[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/170_Relevance/75_Changing_similarities.asciidoc)

相似度算法可以按字段指定， 只需在映射中为不同字段选定即可：

PUT /my\_index

{

"mappings": {

"doc": {

"properties": {

"title": {

"type": "string",

"similarity": "BM25" https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

},

"body": {

"type": "string",

"similarity": "default" https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

}

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/changing-similarities.html#CO122-1) | title 字段使用 BM25 相似度算法。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/changing-similarities.html#CO122-2) | body 字段用默认相似度算法（参见 [实用评分函数](https://www.elastic.co/guide/cn/elasticsearch/guide/current/practical-scoring-function.html)）。 |

目前，Elasticsearch 不支持更改已有字段的相似度算法 similarity 映射，只能通过为数据重新建立索引来达到目的。

#### 配置 BM25[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/170_Relevance/75_Changing_similarities.asciidoc)

配置相似度算法和配置分析器很相似， 自定义相似度算法可以在创建索引时指定，例如：

PUT /my\_index

{

"settings": {

"similarity": {

"my\_bm25": { https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

"type": "BM25",

"b": 0 https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

}

}

},

"mappings": {

"doc": {

"properties": {

"title": {

"type": "string",

"similarity": "my\_bm25" https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png

},

"body": {

"type": "string",

"similarity": "BM25" https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/4.png

}

}

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/changing-similarities.html#CO123-1) | 创建一个基于内置 BM25 ，名为 my\_bm25 的自定义相似度算法。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/changing-similarities.html#CO123-2) | 禁用字段长度规范化（field-length normalization）。参见 [调试 BM25](https://www.elastic.co/guide/cn/elasticsearch/guide/current/pluggable-similarites.html#bm25-tunability) 。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/changing-similarities.html#CO123-3) | title 字段使用自定义相似度算法 my\_bm25 。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/4.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/changing-similarities.html#CO123-4) | 字段 body 使用内置相似度算法 BM25 。 |

提示

自定义的相似度算法可以通过关闭索引，更新索引设置，开启索引这个过程进行更新。这样可以无须重建索引又能试验不同的相似度算法配置。

### 调试相关度是最后 10% 要做的事情[编辑](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/170_Relevance/80_Conclusion.asciidoc)

本章介绍了 Lucene 是如何基于 TF/IDF 生成评分的。理解评分过程是非常重要的， 这样就可以根据具体的业务对评分结果进行调试、调节、减弱和定制。

实践中，简单的查询组合就能提供很好的搜索结果，但是为了获得 具有成效 的搜索结果，就必须反复推敲修改前面介绍的这些调试方法。

通常，经过对策略字段应用权重提升，或通过对查询语句结构的调整来强调某个句子的重要性这些方法，就足以获得良好的结果。有时，如果 Lucene 基于词的 TF/IDF 模型不再满足评分需求（例如希望基于时间或距离来评分），则需要更具侵略性的调整。

除此之外，相关度的调试就有如兔子洞，一旦跳进去就很难再出来。 最相关 这个概念是一个难以触及的模糊目标，通常不同人对文档排序又有着不同的想法，这很容易使人陷入持续反复调整而没有明显进展的怪圈。

我们强烈建议不要陷入这种怪圈，而要监控测量搜索结果。监控用户点击最顶端结果的频次，这可以是前 10 个文档，也可以是第一页的；用户不查看首次搜索的结果而直接执行第二次查询的频次；用户来回点击并查看搜索结果的频次，等等诸如此类的信息。

这些都是用来评价搜索结果与用户之间相关程度的指标。如果查询能返回高相关的文档，用户会选择前五中的一个，得到想要的结果，然后离开。不相关的结果会让用户来回点击并尝试新的搜索条件。

一旦有了这些监控手段，想要调试查询就并不复杂，稍作调整，监控用户的行为改变并做适当反复尝试。本章介绍的一些工具就只是工具而已，要想物尽其用并将搜索结果提高到 极高的 水平，唯一途径就是需要具备能评价度量用户行为的强大能力。