# 停用 ：性能与精度

从早期的信息 索到如今， 我 已 于磁 空 和内存被限制 很小一部分，所以 必 使 的索引尽可能小。 个字 都意味着巨大的性能提升。（ 看 [stemming] ） 干提取的重要性不 是因 它 搜索的内容更广泛、 索的能力更深入， 因 它是 索引空 的工具。

一 最 的 少索引大小的方法就是 索引更少的 。 有些 要比其他 更重要，只索引那些更重要的 来可以大大 少索引的空 。

那 些 条可以被 ？我 可以 分 ：

低 （*Low-frequency terms*）

在文 集合中相 出 少的 ，因 它 稀少，所以它 的 重 更高。

高 （*High-frequency terms*）

在索引下的文 集合中出 多的常用 ，例如 `the`、`and`、和`is`。 些 的 重小， 相 度分影 不大。

| **TIP** | 当然， 率 上是个可以衡量的 尺而不是非 高 即 低 的 。我 可以在尺的任何位置 取一个 准，低于 个 准的属于低 ，高于它的属于高 。 |
|---|---|

到底是低 或是高 取决于它 所 的文 。 `and` 如果在所有都是中文的文 里可能是个低 。在 于数据 的文 集合里， `database` 可能是一个高 ，它 搜索 个特定集合 无助。

言都存在一些非常常 的 ，它 搜索没有太大 。在 Elasticsearch 中，英 的停用 ：

```
a, an, and, are, as, at, be, but, by, for, if, in, into, is, it,
no, not, of, on, or, such, that, the, their, then, there, these,
they, this, to, was, will, with
```

些 停用 通常在索引前就可以被 掉，同 索的 面影 不大。但是 做真的是一个好的解决方案？

# 停用 的 点

在我 有更大的磁 空 ，更多内存，并且 有更好的 算法。 将之前的 33 个常 从索引中移除， 百万文 只能 省 4MB 空 。 所以使用停用 少索引大小不再是一个有效的理由。 (不 法 有一点需要注意，我 在 停用 与短 。)

在此基 上，从索引里将 些 移除会使我 降低某 型的搜索能力。将前面 些所列 移除会 我 以完成以下事情：

- 区分 *happy* 和 *not happy*。
- 搜索 名称 The The。

- 士比 的名句 ``To be, or not to be'' （生存 是 ）。
- 使用 威的国家代 ：no。

移除停用 的最主要好 是性能，假 我 在个具有上百万文 的索引中搜索 fox。或 fox 只在其中 20 个文 中出 ，也就是 Elasticsearch 需要 算 20 个文 的相 度 分 `_score ` 从而排出前十。 在我 把搜索条件改 `the OR fox，几乎所有的文件都包含 the 个 ，也就是 Elasticsearch 需要 所有一百万文 算 分 _score。由此可 第二个 肯定没有第一个的 果好。

幸 的是，我 可以用来保持常用 搜索，同 可以保持良好的性能。首先我 一 学 如何使用停用 。

# 使用停用

移除停用 的工作是由 stop 停用 器完成的，可以通 建自定 的分析器来使用它（参 使用停用 器{ref}/analysis-stop-tokenfilter.html[stop 停用 器])。但是，也有一些自 的分析器 置使用停用 器：

*{ref}/analysis-lang-analyzer.html[ 言分析器]*

个 言分析器 使用与 言相 的停用 列表，例如：english 英 分析器使用 english 停用 列表。

*{ref}/analysis-standard-analyzer.html[standard 准分析器]*

使用空的停用 列表：none， 上是禁用了停用 。

*{ref}/analysis-pattern-analyzer.html[pattern 模式分析器]*

使用空的停用 列表： none，与 standard 分析器 似。

## 停用 和 准分析器（Stopwords and the Standard Analyzer）

了 准分析器能与自定 停用 表 用，我 要做的只需 建一个分析器的配置好的版本，然后将停用 列表 入：

```
PUT /my_index
{
  "settings": {
    "analysis": {
      "analyzer": {
        "my_analyzer": { ①
          "type": "standard", ②
          "stopwords": [ "and", "the" ] ③
        }
      }
    }
  }
}
```

① 自定 的分析器名称 my_analyzer 。

② 个分析器是一个 准 standard 分析器， 行了一些自定 配置。

③ 掉的停用 包括 and 和 the 。

**TIP** 任何 言分析器都可以使用相同的方式配置自定 停用 。

## 保持位置（Maintaining Positions）

analyzer API的 出 果很有趣:

```
GET /my_index/_analyze?analyzer=my_analyzer
The quick and the dead
```

```
{
   "tokens": [
      {
         "token":        "quick",
         "start_offset": 4,
         "end_offset":   9,
         "type":         "<ALPHANUM>",
         "position":     1 ①
      },
      {
         "token":        "dead",
         "start_offset": 18,
         "end_offset":   22,
         "type":         "<ALPHANUM>",
         "position":     4
      }
   ]
}
```

① position 个 元的位置。

停用 如我 期望被 掉了，但有趣的是 个 的位置 position 没有 化：quick 是原句子的第二个 ，dead 是第五个。 短 十分重要，因 如果 个 的位置被 整了，一个短 quick dead 会与以上示例中的文 匹配。

## 指定停用 （Specifying Stopwords）

停用 可以以内 的方式 入，就像我 在前面的例子中那 ，通 指定数 ：

```
"stopwords": [ "and", "the" ]
```

特定 言的 停用 ，可以通 使用 lang 符号来指定:

```
"stopwords": "_english_"
```

TIP: Elasticsearch 中 定 的与 言相 的停用 列表可以在文 "languages", "predefined stopword lists for"){ref}/analysis-stop-tokenfilter.html[stop 停用 器] 中 到。

停用 可以通 指定一个特殊列表 none 来禁用。例如，使用 english 分析器而不使用停用 ，可以通 以下方式做到：

```
PUT /my_index
{
  "settings": {
    "analysis": {
      "analyzer": {
        "my_english": {
          "type":      "english", ①
          "stopwords": "_none_" ②
        }
      }
    }
  }
}
```

① my_english 分析器是基于 english 分析器。

② 但禁用了停用 。

最后，停用 可以使用一行一个 的格式保存在文件中。此文件必 在集群的所有 点上，并且通 stopwords_path 参数 置路径:

```
PUT /my_index
{
  "settings": {
    "analysis": {
      "analyzer": {
        "my_english": {
          "type":           "english",
          "stopwords_path": "stopwords/english.txt" ①
        }
      }
    }
  }
}
```

① 停用 文件的路径， 路径相 于 Elasticsearch 的 config 目 。

## 使用停用 器（Using the stop Token Filter）

当 建 custom 分析器 候，可以 合多个 {ref}/analysis-stop-tokenfilter.html[stop 停用 器] 分 器。例如：我 想要 建一个西班牙 的分析器:

- 自定 停用 列表

- light_spanish 干提取器
- 在 asciifolding 元 器中除去附加符号

我 可以通 以下 置完成:

```
PUT /my_index
{
  "settings": {
    "analysis": {
      "filter": {
        "spanish_stop": {
          "type":        "stop",
          "stopwords": [ "si", "esta", "el", "la" ]  ①
        },
        "light_spanish": { ②
          "type":     "stemmer",
          "language": "light_spanish"
        }
      },
      "analyzer": {
        "my_spanish": {
          "tokenizer": "spanish",
          "filter": [ ③
            "lowercase",
            "asciifolding",
            "spanish_stop",
            "light_spanish"
          ]
        }
      }
    }
  }
}
```

① 停用 器采用与 standard 分析器相同的参数 stopwords 和 stopwords_path 。

② 参 算法提取器（Algorithmic Stemmers）。

③ 器的 序非常重要，下面会 行解 。

我 将 spanish_stop 器放置在 asciifolding 器之后. 意味着以下三个 esta 、ésta 、está ，先通 asciifolding 器 掉特殊字符 成了 esta ，随后使用停用 器会将 esta 去除。如果我 只想移除 esta 和 ésta ，但是 está 不想移除。必 将 spanish_stop 器放置在 asciifolding 之前，并且需要在停用 中指定 esta 和 ésta 。

## 更新停用 （Updating Stopwords）

想要更新分析器的停用 列表有多 方式， 分析器在 建索引 ，当集群 点重 候，或者 的索引重新打 的 候。

如果 使用 stopwords 参数以内 方式指定停用 ，那 只能通

索引，更新分析器的配置{ref}/indices-update-settings.html#update-settings-analysis[update index settings API]，然后在重新打 索引才能更新停用 。

如果 使用 stopwords_path 参数指定停用 的文件路径 ，那 更新停用 就 了。 只需更新文件(在 一个集群 点上)，然后通 者之中的任何一个操作来 制重新 建分析器:

- 和重新打 索引 (参考 {ref}/indices-open-close.html[索引的 与 ])，
- 一一重 集群下的 个 点。

当然，更新的停用 不会改 任何已 存在的索引。 些停用 的只 用于新的搜索或更新文 。如果要改 有的文 ， 需要重新索引数据。参加 [reindex] 。

# 停用 与性能

保留停用 最大的 点就影 搜索性能。使用 Elasticsearch 行全文搜索，它需要 所有匹配的文 算相 度 分 _score 从而返回最相 的前 10 个文 。

通常大多数的 在所有文 中出 的 率低于0.1%，但是有少数 （例如 the ）几乎存在于所有的文 中。假 有一个索引含有100万个文 ， quick brown fox ，能 匹配上的可能少于1000个文 。但是如果 the quick brown fox ，几乎需要 索引中的100万个文 行 分和排序，只是 了返回前 10 名最相 的文 。

的 是 <code>the quick brown fox</code> 是 <code>the</code> 或 <code>quick</code> 或 <code>brown</code> 或 <code>fox</code>— 任何文 即使它什 内容都没有而只包含 <code>the</code> 个 也会被包括在 果集中。因此，我 需要 到一 降低待 分文 数量的方法。

## and 操作符 (and Operator)

我 想要 少待 分文 的数量，最 的方式就是在and 操作符 match 使用 and 操作符，可以 所有 都是必 的。

以下是 match :

```
{
    "match": {
        "text": {
            "query":    "the quick brown fox",
            "operator": "and"
             }
    }
}
```

上述 被重写 bool 如下：

```
{
    "bool": {
        "must": [
            { "term": { "text": "the" }},
            { "term": { "text": "quick" }},
            { "term": { "text": "brown" }},
            { "term": { "text": "fox" }}
        ]
    }
}
```

bool 会智能的根据 的 序依次 行 个 term ：它会从最低 的 始。因 所有 都必 匹配，只要包含低 的文 才有可能匹配。使用 and 操作符可以大大提升多 的速度。

### 最少匹配数(minimum_should_match)

在精度匹配[match-precision]的章 里面，我 使用 minimum_should_match 配置去掉 果中次相 的 尾。 然它只 个目的奏效，但是也 我 从 面 来一个好 ，它提供 and 操作符相似的性能。

```
{
    "match": {
        "text": {
            "query": "the quick brown fox",
            "minimum_should_match": "75%"
        }
    }
}
```

在上面 个示例中，四分之三的 都必 匹配， 意味着我 只需考 那些包含最低 或次低 的文 。相比 使用 or 操作符的 ， 我 来了巨大的性能提升。不 我 有 法可以做得更好……

## 的分 管理

在 字符串中的 可以分 更重要（低 ）和次重要（高 ） 。 只与次重要 匹配的文 很有可能不太相 。 上，我 想要文 能尽可能多的匹配那些更重要的 。

match 接受一个参数 cutoff_frequency ，从而可以 它将 字符串里的 分 低 和高 。低 （更重要的 ） 成 bulk 大量 条件，而高 （次重要的 ）只会用来 分，而不参与匹配 程。通 的区分 理，我 可以在之前慢 的基 上 得巨大的速度提升。

域相 的停用 （Domain-Specific Stopwords）

cutoff_frequency 配置的好 是， 在 特定 域 使用停用 不受 束。例如， 于 影 站使用的 *movie* 、 *color* 、 *black* 和 *white* ， 些 我 往往 几乎没有任何意 。使用 stop 元 器， 些特定 域的 必 手 添加到停用 列表中。然而 cutoff_frequency 会 看索引里 的具体 率， 些 会被自 高 。

以下面 例：

```
{
  "match": {
    "text": {
      "query": "Quick and the dead",
      "cutoff_frequency": 0.01 ①
    }
}
```

① 任何 出 在文 中超 1%，被 是高 。cutoff_frequency 配置可以指定 一个分数（ 0.01 ）或者一个正整数（ 5 ）。

此 通 cutoff_frequency 配置，将 条件 分 低 （ quick ， dead ）和高 （ and ， the ）。然后，此 会被重写 以下的 bool ：

```
{
  "bool": {
    "must": { ①
      "bool": {
        "should": [
          { "term": { "text": "quick" }},
          { "term": { "text": "dead"  }}
        ]
      }
    },
    "should": { ②
      "bool": {
        "should": [
          { "term": { "text": "and" }},
          { "term": { "text": "the" }}
        ]
      }
    }
  }
}
```

① 必 匹配至少一个低 /更重要的 。

② 高 /次重要性 是非必 的。

<code>must</code> 意味着至少有一个低 — <code>quick</code> 或者 <code>dead</code> —必 出 在被匹配文 中。所有其他的文 被排除在外。 <code>should</code> 句 高

<code>and</code> 和 <code>the</code> ，但也只是在 <code>must</code> 句 的 果集文中 。 <code>should</code> 句的唯一的工作就是在 如 <code>Quick and the dead</code> 和 <code>The quick but dead</code> 句 行 分，前者得分比后者高。 方式可以大大 少需要 行 分 算的文 数量。

| TIP | 将操作符参数 置成 and 会要求所有低 都必 匹配，同 包含所有高 的文 予更高 分。但是，在匹配文 ，并不要求文 必 包含所有高 。如果希望文 包含所有的低 和高 ，我 使用一个 bool 来替代。正如我 在and 操作符 (and Operator)中看到的，它的 效率已 很高了。 |
| --- | --- |

## 控制精度

minimum_should_match 参数可以与 cutoff_frequency 合使用，但是此参数 用与低 。如以下 :

```
{
  "match": {
    "text": {
      "query": "Quick and the dead",
      "cutoff_frequency": 0.01,
      "minimum_should_match": "75%"
    }
}
```

将被重写 如下所示:

```
{
  "bool": {
    "must": {
      "bool": {
        "should": [
          { "term": { "text": "quick" }},
          { "term": { "text": "dead"  }}
        ],
        "minimum_should_match": 1 ①
      }
    },
    "should": { ②
      "bool": {
        "should": [
          { "term": { "text": "and" }},
          { "term": { "text": "the" }}
        ]
      }
    }
  }
}
```

① 因　只有　个　，原来的75%向下取整　1，意思是：必　匹配低　　的　者之一。

② 高　　可　的，并且　用于　分使用。

### 高

当使用 <code>or</code>　　高　　条，如— <code>To be, or not to be</code> — 行　　性能最差。只是　了返回最匹配的前十个　果就　只是包含　些　的所有文　　行　分是盲目的。我　真正的意　是　　整个　条出　的文　，所以在　　情况下，不存低　所言，　个　　需要重写　所有高　　条都必　：

```
{
  "bool": {
    "must": [
      { "term": { "text": "to" }},
      { "term": { "text": "be" }},
      { "term": { "text": "or" }},
      { "term": { "text": "not" }},
      { "term": { "text": "to" }},
      { "term": { "text": "be" }}
    ]
  }
}
```

### 常用　使用更多控制（More Control with Common Terms）

尽管高　/低　的功能在　`match`　　中是有用的，有　我　　希望能　它有更多的控制，想控制它　高和低　　分　的行　。　`match`　　`common`　　提供了一　功能。

例如，我　可以　所有低　　都必　匹配，而只　那些包括超　75% 的高　　文　　行　分：

```
{
  "common": {
    "text": {
      "query":                  "Quick and the dead",
      "cutoff_frequency":       0.01,
      "low_freq_operator":      "and",
      "minimum_should_match": {
        "high_freq":            "75%"
      }
    }
  }
}
```

更多配置　参　　{ref}/query-dsl-common-terms-query.html[`common` terms query]。

# 停用　与短

所有　　中　[phrase-matching]　大　占到5%，但是在慢　　里面它　又占大部分。　短　　性能相　　差，特　是当短　中包括常用　的　候，如 “To be, or not to be” 短　全部由停用　　成，是一　　端情况。原因在于几乎需要匹配全量的数据。

在　停用　的　面　停用　的　　点,中，我　提到移除停用　只能　省倒排索引中的一小部分空　。　句　只部分正　，一个典型的索引会可能包含部分或所有以下数据：

字典（*Terms dictionary*）
索引中所有文　内所有　　的有序列表，以及包含　　的文　数量。

倒排表（*Postings list*）
包含　个　　的文　（ID）列表。

（*Term frequency*）
个　　在　个文　里出　的　率。

位置（*Positions*）
个　　在　个文　里出　的位置，供短　　或近似　　使用。

偏移（*Offsets*）
个　　在　个文　里　始与　束字符的偏移，供　　高亮使用，　　是禁用的。

因子（*Norms*）
用来　字段　度　行　　化　理的因子，　　短字段予以更多　重。

将停用　从索引中移除会　省　　字典　和　倒排表　里的少量空　，但　位置　和　偏移　是　一　事。位置和偏移数据很容易　成索引大小的　倍、三倍、甚至四倍。

## 位置信息

`analyzed`　　字符串字段的位置信息　　是　　的，　　所以短　　能随　使用到它。　　出　的越　繁，用来存　它位置信息的空　就越多。在一个大的文　集合中，　于那些非常常　的　，它　的位置信息可能占用成百上千兆的空　。

　行一个　　高　　`the`　　的短　　可能会　致从磁　　取好几G的数据。　些数据会被存到内核文件系　的　存中，以提高后　　的速度，　看似是件好事，但　可能会　致其他数据从　存中被　除，　一　使后　　慢。

然是我　需要解决的　　。

## 索引

我　首先　　自己：是否真的需要使用短　　或 近似　　？

答案通常是：不需要。在很多　用　景下，比如　日志，我　需要知道一个　　是否　　在文　中（　个信息由倒排表提供）而不是　心　的位置在　里。或　我　要　一　个字段使用短　　，但是我　完全可以在其他 `analyzed` 字符串字段上禁用位置信息。

index_options 参数 允 我 控制索引里 个字段存 的信息。 可 如下:

docs
只存 文 及其包含 的信息。 not_analyzed 字符串字段是 的。

freqs
存 docs 信息，以及 个 在 个文 里出 的 次。 是完成TF/IDF 相 度 算的必要条件，但如果只想知道一个文 是否包含某个特定 ， 无需使用它。

positions
存 docs 、 freqs 、 analyzed ，以及 个 在 个文 里出 的位置。 analyzed 字符串字段是 的，但当不需使用短 或近似匹配 ，可以将其禁用。

offsets
存 docs,freqs,positions, 以及 个 在原始字符串中 始与 束字符的偏移信息( postings highlighter )。 个信息被用以高亮搜索 果，但它 是禁用的。

我 可以在索引 建的 候 字段 置 index_options ，或者在使用 put-mapping API新字段映射的 候 置。我 无法修改已有字段的 个 置：

```
PUT /my_index
{
  "mappings": {
    "my_type": {
      "properties": {
        "title": { ①
          "type":          "string"
        },
        "content": { ②
          "type":          "string",
          "index_options": "freqs"
        }
      }
    }
  }
}
```

① title 字段使用 的 positions 置，所以它 于短 或近似 。

② content 字段的位置 置是禁用的，所以它无法用于短 或近似 。

## 停用

除停用 是能 著降低位置信息所占空 的一 方式。 一个被 除停用 的索引 然可以使用短 ，因 剩下的 的原始位置 然被保存着， 正如 保持位置（Maintaining Positions）中看到的那 。尽管如此，将 从索引中排除 究会降低搜索能力， 使我 以区分 *Man in the moon* 与 *Man on the moon* 个短 。

幸 的是， 与熊掌是可以兼得的： 看 common_grams 器。

# common_grams 器

common_grams 器是 短 能更高效的使用停用 而 的。 它与 shingles 器 似（参 相 （[shingles]）), 个相 生成，用示例解 更 容易。

common_grams 器根据 query_mode 置的不同而生成不同 出 果：false （ 索引使用） 或 true （ 搜索使用），所以我 必 建 个独立的分析器：

```
PUT /my_index
{
  "settings": {
    "analysis": {
      "filter": {
        "index_filter": { ①
          "type":         "common_grams",
          "common_words": "_english_" ②
        },
        "search_filter": { ①
          "type":         "common_grams",
          "common_words": "_english_", ②
          "query_mode":   true
        }
      },
      "analyzer": {
        "index_grams": { ③
          "tokenizer":  "standard",
          "filter":   [ "lowercase", "index_filter" ]
        },
        "search_grams": { ③
          "tokenizer": "standard",
          "filter":  [ "lowercase", "search_filter" ]
        }
      }
    }
  }
}
```

① 首先我 基于 common_grams 器 建 个 器： index_filter 在索引 使用（此 query_mode 的 置是 false ）， search_filter 在 使用（此 query_mode 的 置是 true ）。

② common_words 参数可以接受与 stopwords 参数同 的 （参 指定停用 指定停用 （Specifying Stopwords） ）。 个 器 可以接受参数 common_words_path ，使用存于文件里的常用 。

③ 然后我 使用 器各 建一个索引 分析器和 分析器。

有了自定 分析器，我 可以 建一个字段在索引 使用 index_grams 分析器：

```
PUT /my_index/_mapping/my_type
{
  "properties": {
    "text": {
      "type":            "string",
      "analyzer":  "index_grams", ①
      "search_analyzer": "standard" ①
    }
  }
}
```

① `text` 字段索引 使用 `index_grams` 分析器，但是在搜索 使用 `standard` 分析器， 后我 会解 其原因。

### 索引 （At Index Time）

如果我 短 *The quick and brown fox* 行拆分，它生成如下 ：

```
Pos 1: the_quick
Pos 2: quick_and
Pos 3: and_brown
Pos 4: brown_fox
```

新的 `index_grams` 分析器生成以下 ：

```
Pos 1: the, the_quick
Pos 2: quick, quick_and
Pos 3: and, and_brown
Pos 4: brown
Pos 5: fox
```

所有的 都是以 `unigrams` 形式 出的（the、quick 等等），但是如果一个 本身是常用 或者跟随着常用 ，那 它同 会在 `unigram` 同 的位置以 `bigram` 形式 出：`the_quick` ， `quick_and` ， `and_brown` 。

### 字 （Unigram Queries）

因 索引包含 `unigrams` ，可以使用与其他字段相同的技 行 ，例如：

```
GET /my_index/_search
{
  "query": {
    "match": {
      "text": {
        "query": "the quick and brown fox",
        "cutoff_frequency": 0.01
      }
    }
  }
}
```

上面 个 字符串是通 文本字段配置的 search_analyzer 分析器 --本例中使用的是 standard 分析器--行分析的， 它生成的 ： the， quick， and， brown， fox 。

因 text 字段的索引中包含与 standard 分析去生成的一 的 unigrams ，搜索于任何普通字段都能正常工作。

## 二元 法短 （Bigram Phrase Queries）

但是，当我 行短 ，我 可以用 的 search_grams 分析器 整个 程 得更高效：

```
GET /my_index/_search
{
  "query": {
    "match_phrase": {
      "text": {
        "query":    "The quick and brown fox",
        "analyzer": "search_grams" ①
      }
    }
  }
}
```

① 于短 ，我 重写了 的 search_analyzer 分析器，而使用 search_grams 分析器。

search_grams 分析器会生成以下 ：

```
Pos 1: the_quick
Pos 2: quick_and
Pos 3: and_brown
Pos 4: brown
Pos 5: fox
```

分析器排除了所有常用 的 unigrams，只留下常用 的 bigrams 以及低 的 unigrams。如 the_quick 的 bigrams 比 个 the 更 少 ， 有 个好 ：

- the_quick 的位置信息要比 the 的小得多，所以它 取磁 更快， 系 存的影 也更小。

- `the_quick` 没有 `the` 那 常 ，所以它可以大量 少需要 算的文 。

### 短 （Two-Word Phrases）

我 的 化可以更 一 ，因 大多数的短 只由 个 成，如果其中一个恰好又是常用 ，例如：

```
GET /my_index/_search
{
  "query": {
    "match_phrase": {
      "text": {
        "query":    "The quick",
        "analyzer": "search_grams"
      }
    }
  }
}
```

那 `search_grams` 分析器会 出 个 元：`the_quick` 。 将原来昂 的 （ `the` 和 `quick` ） 成了 个 的高效 。

## 停用 与相 性

在 束停用 相 内容之前，最后一个 是 于相 性的。在索引中保留停用 会降低相 度 算的准性，特 是当我 的文 非常 。

正如我 在 [bm25-saturation] 已 的， 原因在于 [bm25-saturation] 并没有 制率的影 置上限 。 基于逆文 率的影 ，非常常用的 可能只有很低的 重，但是在 文 中， 个文 出 的 数量很大的停用 会 致 些 被不自然的加 。

可以考 包含停用 的 字段使用 Okapi BM25 相似度算法，而不是 的 Lucene 相似度。