placeholder2

理人　　言

全文搜索是一　准率与全率之的量—准率即尽量返回少的无文，而全率尽量返回多的相文。　尽管能精准匹配用　的　，但　然不，我会　很多被用　是相的文。　因此，我需要把撒得更广一些，去搜索那些和原文不是完全匹配但却相的　。

道不期待在搜索“quick brown fox“匹配到包含“fast brown foxed“的文，或是搜索“Johnny Walker“匹配到“Johnnie Walker“，又或是搜索“Arnolt Schwarzenneger“匹配到“Arnold Schwarzenegger“？

如果文　包含用　的内容，那　些文　当出在返回果的最前面，而匹配程度低的文将会排在后的位置。　如果没有任何完全匹配的文，我至少可以用展示一些潜在的匹配果；它甚至可能就是用最初想要的果。

以下列出了一些可化的地方：

- 清除似´，^，¨的音符号，　在搜索 `rôle` 的候也会匹配 `role`，反之亦然。　一化元。
- 通提取　的干，清除数和数之的差—`fox`　与<code>foxes</code>—以及　上的差—`jumping`、<code>jumped</code>与<code>jumps</code>。　将　原　根。
- 清除常用或者 停用，如 `the`，`and`，和 `or`，从而提升搜索性能。　停用：性能与精度。
- 包含同　，　在搜索 `quick` 也可以匹配 `fast`，或者在搜索 `UK` 匹配 `United Kingdom`。同　。
- 　写　和替代写方式，或者　同音型—音一致的不同　，例如<code>their</code>与<code>there</code>，<code>meat</code>、<code>meet</code>与<code>mete</code>。　写　。

在我可以操控个　之前，需要先将文本切分成　，也意味着我需要知道　是由什　成的。我将在　章　个　。

在之前，我看看如何更快更　地始。

始　理各　言

Elasticsearch 很多世界流行言提供良好的、　的、箱即用的言分析器集合：

阿拉伯、美尼　、巴斯克、巴西、保加利　、加泰尼　、中文、捷克、丹麦、荷　、英、　、法、加里西　、　、希　、北印度、匈牙利、印度尼西、　、意大利、日、国、　、威、波斯、葡萄牙、　尼　、俄、西班牙、瑞典、土耳其和泰。

些分析器承担以下四角色：

- 文本拆分　：

The quick brown foxes → [The, quick, brown, foxes]

- 大写 小写：

  The → the

- 移除常用的 停用 ：

  [The, quick, brown, foxes] → [quick, brown, foxes]

- 将 型 （例如 数 ， 去式） 化 根：

  foxes → fox

了更好的搜索性， 个 言的分析器提供了 言 的具体 ：

- 英 分析器移除了所有格 's

  John's → john

- 法 分析器移除了 元音省略 例如 l' 和 qu' 和 音符号 例如 ¨ 或 ^ ：

  l'église → eglis

- 分析器 化了切 ， 将切 中的 ä 和 ae 替 a ， 或将 ß 替 ss ：

  äußerst → ausserst

使用 言分析器

Elasticsearch 的内置分析器都是全局可用的，不需要提前配置，它 也可以在字段映射中直接指定在某字段上：

```
PUT /my_index
{
  "mappings": {
    "blog": {
      "properties": {
        "title": {
          "type":     "string",
          "analyzer": "english" ①
        }
      }
    }
  }
}
```

① title 字段将会用 english（英 ）分析器替 的 standard（ 准）分析器

当然，文本 english 分析 理，我 会 失源数据：

```
GET /my_index/_analyze?field=title ①
I'm not happy about the foxes
```

① 切　　：i'm, happi, about, fox

我　无法分　源文　中是包含　数 fox　是　数 foxes ；　　not 因　是停用　所以被移除了，　所以我　无法分　源文　中是happy about foxes　是not happy about foxes，　然通　使用 english （英）分析器，使得匹配　　更加　松，我　也因此提高了召回率，但却降低了精准匹配文　的能力。

　了　得　方面的　　，我　可以使用multifields（多字段）　　title　字段建立　次索引：　一次使用 english（英　）分析器，　一次使用 standard（　准）分析器:

```
PUT /my_index
{
  "mappings": {
    "blog": {
      "properties": {
        "title": { ①
          "type": "string",
          "fields": {
            "english": { ②
              "type":     "string",
              "analyzer": "english"
            }
          }
        }
      }
    }
  }
}
```

① 主 title 字段使用 standard（　准）分析器。

② title.english 子字段使用 english（英　）分析器。

替　　字段映射后，我　可以索引一些　　文　来展示　　在搜索　使用　个字段：

```
PUT /my_index/blog/1
{ "title": "I'm happy for this fox" }

PUT /my_index/blog/2
{ "title": "I'm not happy about my fox problem" }

GET /_search
{
  "query": {
    "multi_match": {
      "type":     "most_fields", ①
      "query":    "not happy foxes",
      "fields": [ "title", "title.english" ]
    }
  }
}
```

① 使用most_fields query type（多字段搜索　法来）　我　可以用多个字段来匹配同一段文本。

感　title.english 字段的切　，无　我　的文　中是否含有　　foxes 都会被搜索到，第二　文　的相性排行要比第一　高，因　在 title 字段中匹配到了　　not 。

配置　言分析器

言分析器都不需要任何配置，　箱即用，它　中的大多数都允　　控制它　的各方面行　，具体来　：

干提取排除

想象下某个　景，用　　想要搜索 World Health Organization 的　果，但是却被替　　搜索 organ health 的　果。有　个困惑是因　　organ 和 organization 有相同的　根： organ 。通常　不是什　　，但是在一些特殊的文　中就会　致有　　的　果，所以我　希望防止　　organization　和 organizations 被　　干。

自定　停用

英　中　　的停用　列表如下：

```
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
```

于　　no 和 not 有点特　，　　会反　跟在它　后面的　　的含　。或　我　　个很重要，不　　把他　看成停用　。

了自定　　english （英　）分　器的行　，我　需要基于　english （英　）分析器　建一个自定分析器，然后添加一些配置：

```
PUT /my_index
{
  "settings": {
    "analysis": {
      "analyzer": {
        "my_english": {
          "type": "english",
          "stem_exclusion": [ "organization", "organizations" ], ①
          "stopwords": [ ②
            "a", "an", "and", "are", "as", "at", "be", "but", "by", "for",
            "if", "in", "into", "is", "it", "of", "on", "or", "such", "that",
            "the", "their", "then", "there", "these", "they", "this", "to",
            "was", "will", "with"
          ]
        }
      }
    }
  }
}

GET /my_index/_analyze?analyzer=my_english ③
The World Health Organization does not sell organs.
```

① 防止 organization 和 organizations 被 干

② 指定一个自定 停用 列表

③ 切 world 、 health 、 organization 、 does 、 not 、 sell 、 organ

我 在 将 原 根 和 停用 ：性能与精度 中分 了 干提取和停用 。

混合 言的陷

如果 只需要 理一 言，那 很幸 。 到一个正 的策略用于 理多 言文 是一 巨大的挑 。

在索引的 候

多 言文 主要有以下三个 型：

- 一 是 *document* （文 ）有自己的主 言，并包含一些其他 言的片段（参考 文 一言。）
- 一 是 个 *field* （域）有自己的主 言，并包含一些其他 言的片段（参考 个域一 言。）
- 一 是 个 *field* （域）都是混合 言（参考 混合 言域。）

（分 ）目 不 是可以 ，我 当保持将不同 言分隔 。在同一 倒排索引内混合多 言可能造成一些 。

不合理的 干提取

的 干提取 跟英 ，法 ，瑞典 等是不一 的。 不同的 言提供同 的 干提 将会

致有的 的 根 的正 ，有的 的 根 的不正 ，有的 根本 不到 根。 甚至是将不同言的不同含 的 切 同一个 根，合并 些 根的搜索 果会 用 来困 。

提供多 的 干提取器 流切分同一 文 的 果很有可能得到一堆 ，因 下一个 干提取器会 切分一个已 被 干的 ， 加 了上面提到的 。

写方式一 干提取器

只有一 情况， *only-one-stemmer* （唯一 干提取器）会 生，就是 言都有自己的写方式。例如，在以色列就有很大的可能一个文 包含希伯来 ， 阿拉伯 ，俄 （古代斯拉夫 ），和英 。

```
- Предупреждение -          - Warning
```

言使用不同的 写方式，所以一 言的 干提取器就不会干 其他 言的，允 同一 文本提供多 干提取器。

不正 的倒排文 率

在 [relevance-intro] （相 性教程）中，一个 term （ ）在一 文 中出 的 率 高， term（ ）的 重就越低。 了精 的 算相 性， 需要精 的 term-frequency（ ）。

一段 文出 在英 主的文本中会 与 更高的 重， 那 高 重是因 相 来 更稀有。但是如果 文 跟以 主的文 混合在一起，那 段 文就会有很低的 重。

在搜索的 候

然而 考 的文 是不 的 。 也需要考 的用 会 搜索 些文 。 通常 能从用 的 言界面来 定用 的主 言，（例如， `mysite.de` 和 `mysite.fr` ） 或者从用 的 器的HTTP header（HTTP 文件） `accept-language` 定。

用 的搜索也注意有三个方面:

- 用 使用他的主 言搜索。
- 用 使用其他的 言搜索，但希望 取主 言的搜索 果。
- 用 使用其他 言搜索，并希望 取 言的搜索 果。（例如，精通双 的人，或者 的外国 者）。

根据 搜索数据的 型，或 会返回 言的合 果（例如，一个用 在西班牙 站搜索商品），也可能是用 主 言的搜索 果和其他 言的搜索 果混合。

通常来 ， 与用 言偏好的搜索很有意 。一个使用英 的用 搜索 更希望看到英 Wikipedia 面而不是法 Wikipedia 面。

言

很可能已 知道 的文 所 用的 言，或者 的文 只是在 自己的 内 写并被翻 成 定的一系

列 言。人 的 可能是最可 的将 言正 的方法。

然而，或 的文 来自第三方 源且没 言 ，或者是不正 的 。 情况下， 需要一个学 算法来 文 的主 言。幸 的是，一些 言有 成的工具包可以 解决 个 。

内容是来自 Mike McCandless 的 chromium-compact-language-detector 工具包，使用的是google 的基于 (Apache License 2.0)的 源工具包 Compact Language Detector (CLD) 。 它小巧，快速，且精 ，并能根据短短的 句 就可以 160+ 的 言。 它甚至能 文本 多 言。支持多 言包括 Python，Perl，JavaScript，PHP，C#/.NET，和 R 。

定用 搜索 求的 言并不是那 。 CLD 是 了至少 200 字符 的文本 的。字符短的文本，例如搜索 字，会 生不精 的 果。 情况下，或 采取一些 的 式算法会更好些，例如 国家的官方 言，用 的 言，和 HTTP `accept-language` headers （HTTP 文件）。

文 一 言

个主 言文 只需要相当 的 置。 不同 言的文 被分 存放在不同的索引中 — <code>blogs-en</code> 、 <code>blogs-fr</code> ， 如此等等 — 个索引就可以使用相同的 型和相同的域，只是使用不同的分析器：

```
PUT /blogs-en
{
  "mappings": {
    "post": {
      "properties": {
        "title": {
          "type": "string", ①
          "fields": {
            "stemmed": {
              "type":     "string",
              "analyzer": "english" ②
            }
}}}}}}

PUT /blogs-fr
{
  "mappings": {
    "post": {
      "properties": {
        "title": {
          "type": "string", ①
          "fields": {
            "stemmed": {
              "type":     "string",
              "analyzer": "french" ②
            }
}}}}}}
```

① 索引 blogs-en 和 blogs-fr 的 post 型都有一个包含 title 域。

② title.stemmed 子域使用了具体 言的分析器。

个方法干 且 活。新 言很容易被添加 — 是 建一个新索引—因 言都是 底的被分 ，我 不用遭受在 混合 言的陷 中描述的 和 干提取的 。

一 言的文 都可被独立 ，或者通 多 索引来 多 言。 我 甚至可以使用 indices_boost 参数 特定的 言添加 先 ：

```
GET /blogs-*/post/_search ①
{
    "query": {
        "multi_match": {
            "query":   "deja vu",
            "fields":  [ "title", "title.stemmed" ] ②
            "type":    "most_fields"
        }
    },
    "indices_boost": { ③
        "blogs-en": 3,
        "blogs-fr": 2
    }
}
```

① 个 会在所有以 blogs- 的索引中 行。

② title.stemmed 字段使用 个索引中指定的分析器 。

③ 也 用 接受 言 表明，更 向于英 ，然后是法 ，所以相 的，我 会 个索引的 果添加 重。任何其他 言会有一个中性的 重 1 。

外

当然，有些文 含有一些其他 言的 或句子，且不幸的是 些 被切 了正 的 根。 于主 言文 ， 通常并不是主要的 。用 常需要搜索很精 的 —例如，一个其他 言的引用—而不是型 化 的 。召回率 (Recall)可以通 使用 一化 元 中 解的技 提升。

假 有些 例如地名 当能被主 言和原始 言都能 索，例如 *Munich* 和 *München* 。 些 上是我 在 同 解 的同 。

不要 言使用 型

也 很 向于 个 言使用分 的 型，来代替使用分 的索引。 了 到最佳效果， 当避免使用 型。在 [mapping] 解 ，不同 型但有相同域名的域会被索引在相同的倒排索引 中。 意味着不同 型（和不同 言）的 混合在了一起。

了 保一 言的 不会 染其他 言的 ，在后面的章 中会介 到，无 是 个 言使用 独的索引， 是使用 独的域都可以。

个域一　　言

于一些　体　，例如:　品、　影、法律声明，　　通常　　的一　文本会被翻　成不同　言的文　。　然　些不同　言的文　可以　独保存在各自的索引中。但　一　更合理的方式是同一　文本的所有翻　　一保存在一个索引中。。

```
{
   "title":     "Fight club",
   "title_br":  "Clube de Luta",
   "title_cz":  "Klub rváčů",
   "title_en":  "Fight club",
   "title_es":  "El club de la lucha",
   ...
}
```

翻　存　在不同的域中，根据域的　言决定使用相　的分析器：

```
PUT /movies
{
  "mappings": {
    "movie": {
      "properties": {
        "title": { ①
          "type":       "string"
        },
        "title_br": { ②
            "type":     "string",
            "analyzer": "brazilian"
        },
        "title_cz": { ②
            "type":     "string",
            "analyzer": "czech"
        },
        "title_en": { ②
            "type":     "string",
            "analyzer": "english"
        },
        "title_es": { ②
            "type":     "string",
            "analyzer": "spanish"
        }
      }
    }
  }
}
```

① `title` 域含有title的原文，并使用 `standard` （　准）分析器。

② 其他字段使用　合自己　言的分析器。

在 持干 的 方面， 然 *index-per-language* （一 言一 索引的方法），不像 *field-per-language* （一 言一个域的方法）分 索引那 活。但是使用 update-mapping API 添加一个新域也很 ，那些新域需要新的自定 分析器， 些新分析器只能在索引 建 被装配。有一个 通的方案， 可以先 个索引 {ref}/indices-open-close.html[close] ，然后使用 {ref}/indices-update-settings.html[update-settings API] ，重新打 个索引，但是 掉 个索引意味着得停止服 一段 。

文 的一 言可以 独 ，也可以通 多个域来 多 言。我 甚至可以通 特定 言 置偏好来提高字段 先 ：

```
GET /movies/movie/_search
{
    "query": {
        "multi_match": {
            "query":    "club de la lucha",
            "fields": [ "title*", "title_es^2" ], ①
            "type":     "most_fields"
        }
    }
}
```

① 个搜索 所有以 title 前 的域，但是 title_es 域加 重 2 。其他的所有域是中性 重 1 。

混合 言域

通常,那些从源数据中 得的多 言混合在一个域中的文 会超出 的控制， 例如从 上爬取的 面：

```
{ "body": "Page not found / Seite nicht gefunden / Page non trouvée" }
```

正 的 理多 言 型文 是非常困 的。即使 所有的域使用 standard （ 准）分析器， 但 的文 会 得不利于搜索，除非 使用了合 的 干提取器。当然， 不可能只 一个 干提取器。 干提取器是由 言具体决定的。或者， 干提取器是由 言和脚本所具体决定的。像在 写方式一 干提取器 中那 。如果 个 言都使用不同的脚本，那 干提取器就可以合并了。

假 的混合 言使用的是一 的脚本，例如拉丁文， 有三个可用的 ：

- 切分到不同的域
- 行多次分析
- 使用 n-grams

切分到不同的域

在 言 提到 的 的 言 可以告 部分文 属于 言。 可以用 个域一 言 中用 的一 的方法来根据 言切分文本。

行多次分析

如果 主要 理数量有限的 言， 可以使用多个域， 言都分析文本一次。

```
PUT /movies
{
  "mappings": {
    "title": {
      "properties": {
        "title": { ①
          "type": "string",
          "fields": {
            "de": { ②
              "type":     "string",
              "analyzer": "german"
            },
            "en": { ②
              "type":     "string",
              "analyzer": "english"
            },
            "fr": { ②
              "type":     "string",
              "analyzer": "french"
            },
            "es": { ②
              "type":     "string",
              "analyzer": "spanish"
            }
          }
        }
      }
    }
  }
}
```

① 主域 `title` 使用 `standard` （ 准）分析器

② 个子域提供不同的 言分析器来 `title` 域文本 行分析。

使用 n-grams

可以使用 [ngrams-compound-words] 中描述的方法索引所有的 n-grams。 大多数 型化包含 添加一个后 （或在一些 言中添加前 ），所以通 将 拆成 n-grams，有很大的机会匹配到相似但不完全一 的 。 个可以 合 *analyze-multiple times*（多次分析）方法 不支持的 言提供全域 取：

```
PUT /movies
{
  "settings": {
    "analysis": {...} ①
  },
  "mappings": {
    "title": {
      "properties": {
        "title": {
          "type": "string",
          "fields": {
            "de": {
              "type":     "string",
              "analyzer": "german"
            },
            "en": {
              "type":     "string",
              "analyzer": "english"
            },
            "fr": {
              "type":     "string",
              "analyzer": "french"
            },
            "es": {
              "type":     "string",
              "analyzer": "spanish"
            },
            "general": { ②
              "type":     "string",
              "analyzer": "trigrams"
            }
          }
        }
      }
    }
  }
}
```

① 在 analysis 章 ，我 按照 [ngrams-compound-words] 中描述的定 了同 的 trigrams 分析器。

② 在 title.general 域使用 trigrams 分析器索引所有的 言。

当 取所有 general 域 ， 可以使用 minimum_should_match （最少 当匹配数）来 少低 量的匹配。 或 也需要 其他字段 行 微的加 ， 与主 言域的 重要高于其他的在 general 上的域：

```
GET /movies/movie/_search
{
    "query": {
        "multi_match": {
            "query":    "club de la lucha",
            "fields": [ "title*^1.5", "title.general" ], ①
            "type":     "most_fields",
            "minimum_should_match": "75%" ②
        }
    }
}
```

① 所有 title 或 title.* 域 与了比 title.general 域 微高的加 。

② minimum_should_match（最少 当匹配数） 参数 少了低 量匹配的返回数, title.general 域尤其重要。

英 相 而言比 容易 : 之 都是以空格或者（一些） 点隔 。 然而即使在英 中也会有一些争 : *you're* 是一个 是 个？ *o'clock* , *cooperate* , *half-baked* , 或者 *eyewitness* 些 ？

或者荷 把独立的 合并起来 造一个 的合成 如 *Weißkopfseeadler* (white-headed sea eagle) , 但是 了在 Adler (eagle)的 候返回 Weißkopfseeadler 的 果，我 需要 得 将合并 拆成 。

洲的 言更 ：很多 言在 , 句子，甚至段落之 没有空格。 有些 可以用一个字来表 , 但是同 的字在 一个字旁 的 候就是不同意思的 的一部分。

而易 的是没有能 奇 般 理所有人 言的万能分析器，Elasticsearch 很多 言提供了 用的分析器， 其他特殊 言的分析器以 件的形式提供。

然而并不是所有 言都有 用分析器，而且有 候 甚至无法 定 理的是什 言。 情况，我 需要一些忽略 言也能合理工作的 准工具包。

准分析器

任何全文 索的字符串域都 使用 standard 分析器。 如果我 想要一个 自定 分析器 , 可以按照如下定 方式重新 准 分析器：

```
{
    "type":      "custom",
    "tokenizer": "standard",
    "filter":  [ "lowercase", "stop" ]
}
```

在 一化 元 （ 准化 元）和 停用 : 性能与精度 （停用 ）中，我 了 lowercase

（小写字母）和 stop （停用 ） 元 器，但是 在，我 注于 standard *tokenizer* （ 准分器）。

准分 器

分 器 接受一个字符串作 入，将 个字符串拆分成独立的 或 元（*token*） （可能会 一些点符号等字符），然后 出一个 元流（*token stream*）。

有趣的是用于 的算法。 whitespace （空白字符）分 器按空白字符 —— 空格、tabs、行符等等 行 拆分 —— 然后假定 的非空格字符 成了一个 元。例如：

```
GET /_analyze?tokenizer=whitespace
You're the 1st runner home!
```

个 求会返回如下 （terms）：You're、the、1st、runner、home!

letter 分 器，采用 外一 策略，按照任何非字符 行拆分， 将会返回如下 ：You、re、the、st、runner、home。

standard 分 器使用 Unicode 文本分割算法 （定 来源于 Unicode Standard Annex #29）来 之 的界限，并且 出所有界限之 的内容。 Unicode 内含的知 使其可以成功的 包含混合言的文本 行分 。

点符号可能是 的一部分，也可能不是， 取决于它出 的位置：

```
GET /_analyze?tokenizer=standard
You're my 'favorite'.
```

在 个例子中，You're 中的 号被 的一部分，然而 'favorite' 中的 引号 不会被 的一部分， 所以分 果如下：You're、my、favorite。

> **TIP**
>
> uax_url_email 分 器和 standard 分 器工作方式 其相同。 区 只在于它能 email 地址和 URLs 并 出 个 元。standard 分 器 不一 ，会将 email 地址和 URLs 拆分成独立的 。例如，email 地址 joe-bloggs@foo-bar.com 的分 果 joe、bloggs、foo、bar.com。

standard 分 器是大多数 言分 的一个合理的起点，特 是西方 言。 事 上，它 成了大多数特定言分析器的基 ，如 english 、french 和 spanish 分析器。 它也支持 洲 言，只是有些 陷，可以考 通 ICU 件的方式使用 icu_tokenizer 行替 。

安装 ICU 件

Elasticsearch的 ICU 分析器 件 使用 国 化 件 *Unicode* (ICU) 函数 （ 情 看 site.project.org）提供 富的 理 Unicode 工具。 些包含 理 洲 言特 有用的 icu_分 器， 有大量 除英外其他 言 行正 匹配和排序所必 的分 器。

NOTE

ICU 件是 理英 之外 言的必需工具，非常推 安装并使用它，不幸的是，因 是基于 外的 ICU 函数 ， 不同版本的ICU 件可能并不兼容之前的版本，当更新 件的 候， 需要重新索引 的数据。

安装 个 件，第一 先 掉 的Elasticsearch 点，然后在Elasticsearch的主目 行以下命令：

```
./bin/plugin -install elasticsearch/elasticsearch-analysis-icu/$VERSION ①
```

① 当前 `$VERSION` （版本）可以在以下地址 到 *https://github.com/elasticsearch/elasticsearch-analysis-icu*.

一旦安装后，重 Elasticsearch， 将会看到 似如下的一条 日志：

```
[INFO][plugins] [Mysterio] loaded [marvel, analysis-icu], sites [marvel]
```

如果 有很多 点并以集群方式 行的， 需要在集群的 个 点都安装 个 件。

icu_分 器

`icu_分 器` 和 `准分 器` 使用同 的 Unicode 文本分段算法， 只是 了更好的支持 洲 ，添加了泰 、老 、中文、日文、和 文基于 典的 方法，并且可以使用自定 将 和柬埔 文本拆分成音 。

例如，分 比 `准分 器` 和 `icu_分 器` 在分 泰 中的 `'Hello. I am from Bangkok.'` 生的 元：

```
GET /_analyze?tokenizer=standard
```

`准分 器` 生了 个 元， 个句子一个： ， 。 个只是 想搜索整个句子 `'I am from Bangkok.'` 的 候有用，但是如果 想搜索 `'Bangkok.'` 不行。

```
GET /_analyze?tokenizer=icu_tokenizer
```

相反， `icu_分 器` 可以把文本分成独立的 （ ， ， ， ， ）， 使得文 更容易被搜索到。

相 而言， `准分 器` 分 中文和日文的 候“ 度分 ”了， 常将一个完整的 拆分独立的字符，因 之 并没有空格，很 区分 的字符是 隔的 是一个句子中的 字：

- 向的意思是 *facing* （面 ），日的意思是 *sun* （太 ），葵的意思是 *hollyhock*（蜀葵）。当写在一起的 候, 向日葵的意思是 *sunflower*（向日葵）。
- 五的意思是 *five* （五）或者 *fifth* （第五）， 月的意思是 *month* （月 ），雨的意思是 *rain*（下雨）。 第一个和第二个字符写在一起成了五月，意思是 *the month of May*（一年中的五月），

然而添加上第三个字符，五月雨的意思是 *continuous rain*（不断的下雨，梅雨）。当在合并第四个字符，式，意思是 *style*（式），五月雨式个成了一不屈不持不断的西的形容。

然个字符本身可以是一个，但使元保持更大的原始概念比使其作一个的一部分要有意的多：

```
GET /_analyze?tokenizer=standard
向日葵

GET /_analyze?tokenizer=icu_tokenizer
向日葵
```

准分器在前面的例子中将个字符出独的元：向，日，葵。icu_分器会出个元向日葵（sunflower）。

<code>准分器</code>和<code>icu_分器</code>的一个不同的地方是后者会将不同写方式的字符（例如，<code>βeta</code>）拆分成独立的元—<code>β</code>和<code>eta</code>—，而前者会出个元：<code>βeta</code>。

整理入文本

当入文本是干的候分器提供最佳分果，有效文本，里有效指的是遵从 Unicode 算法期望的点符号。然而很多候，我需要理的文本会是除了干文本之外的任何文本。在分之前整理文本会提升出果的量。

HTML 分

将 HTML 通准分器或 icu_分器分将生糟的果。些分器不知道如何理 HTML。例如：

```
GET /_analyze?tokenizer=standard
<p>Some d&eacute;j&agrave; vu <a href="http://somedomain.com">website</a>
```

准分器会混 HTML 和体，并且出以下元：p、Some、d、eacute、j、agrave、vu、a、href、http、somedomain.com、website、a。些元然不知所云！

字符器可以添加分析器中，在将文本分器之前理文本。在情况下，我可以用<code>html_strip</code>字符器移除 HTML 并 HTML 体如<code>é</code>一致的 Unicode 字符。

字符器可以通 analyze API 行，需要在字符串中指明它：

```
GET /_analyze?tokenizer=standard&char_filters=html_strip
<p>Some d&eacute;j&agrave; vu <a href="http://somedomain.com">website</a>
```

想将它 作 分析器的一部分使用，需要把它 添加到 custom 型的自定 分析器里：

```
PUT /my_index
{
    "settings": {
        "analysis": {
            "analyzer": {
                "my_html_analyzer": {
                    "tokenizer":     "standard",
                    "char_filter": [ "html_strip" ]
                }
            }
        }
    }
}
```

一旦自定 分析器 建好之后， 我 新的 my_html_analyzer 就可以用 analyze API :

```
GET /my_index/_analyze?analyzer=my_html_analyzer
<p>Some d&eacute;j&agrave; vu <a href="http://somedomain.com">website</a>
```

次 出的 元才是我 期望的： Some , déjà , vu , website 。

整理 点符号

准分 器 和 icu_分 器 都能理解 中的 号 当被 的一部分，然而包 的引号在不 。分 文本 You're my 'favorite' , 会被 出正 的 元 You're , my , favorite 。

不幸的是， Unicode 列出了一些有 会被用 号的字符：

U+0027
号 (<code>'</code>)— 原始 ASCII 符号

U+2018
左 引号 (<code>‘</code>)— 当 引用 作 一个引用的 始

U+2019
右 引号 (<code>’</code>)— 当 引用 座位一个引用的 束，也是 号的首 字符。

当 三个字符出 在 中 的 候， 准分 器 和 icu_分 器 都会将 三个字符 号（会被 的一部分）。然而 有 外三个 得很像 号的字符：

U+201B
Single high-reversed-9 （高反 引号） (<code>‛</code>)— 跟 <code>U+2018</code> 一 ，但是外 上有区

U+0091
ISO-8859-1 中的左 引号 — 不会被用于 Unicode 中

U+0092

ISO-8859-1 中的右 引号 — 不会被用于 Unicode 中

准分 器 和 icu_分 器 把 三个字符 的分界 — 一个将文本拆分 元的位置。不幸的是，一些出版社用 U+201B 作 名字的典型 写方式例如 M‛coy ， 第二个字符或 可以被 的文字 理 件打出来， 取决于 款 件的年 。

即使在使用可以“接受”的引号 ，一个用 引号 写的 — <code>You’re</code> — 也和一个用 号 写的 — <code>You’re</code> — 不一 ，意味着搜索其中的一个 体将会 不到 一个。

幸 的是，可以用 mapping 些混乱的字符 行分 ， 器可以 行我 用 一个字符替 所有例中的一个字符。 情况下，我 可以 的用 U+0027 替 所有的 号 体：

```
PUT /my_index
{
  "settings": {
    "analysis": {
      "char_filter": { ①
        "quotes": {
          "type": "mapping",
          "mappings": [ ②
            "\\u0091=>\\u0027",
            "\\u0092=>\\u0027",
            "\\u2018=>\\u0027",
            "\\u2019=>\\u0027",
            "\\u201B=>\\u0027"
          ]
        }
      },
      "analyzer": {
        "quotes_analyzer": {
          "tokenizer":     "standard",
          "char_filter": [ "quotes" ] ③
        }
      }
    }
  }
}
```

① 我 自定 了一个 char_filter （字符 器）叫做 quotes，提供所有 号 体到 号的映射。

② 了更清晰，我 使用 个字符的 JSON Unicode 句，当然我 也可以使用他 本身字符表示：`"‘⇒'"`。

③ 我 用自定 的 quotes 字符 器 建一个新的分析器叫做 quotes_analyzer。

像以前一 ，我 需要在 建了分析器后 它：

```
GET /my_index/_analyze?analyzer=quotes_analyzer
You're my 'favorite' M'Coy
```

个例子返回如下 元，其中所有的 中的引号 都被替 了 号： You're, my, favorite, M'Coy 。

投入更多的努力 保 的分 器接收到高 量的 入， 的搜索 果 量也将会更好。

一化 元

把文本切割成 元(token)只是 工作的一半。 了 些 元(token)更容易搜索, 些 元(token)需要被 一化(*normalization*)-- 个 程会去除同一个 元(token)的无意 差 ，例如大写和小写的差 。可能我 需要去掉有意 的差 ， esta、ésta 和 está 都能用同一个元(token)来搜索。 会用 déjà vu 来搜索， 是 deja vu?

些都是 元 器的工作。 元 器接收来自分 器(tokenizer)的 元(token)流。 可以一起使用多个 元 器， 一个都有自己特定的 理工作。 一个 元 器都可以 理来自 一个 元 器 出的 流。

个例子

用的最多的 元 器(token filters)是 lowercase 器，它的功能正和 期望的一 ；它将 个元(token) 小写形式：

```
GET /_analyze?tokenizer=standard&filters=lowercase
The QUICK Brown FOX! ①
```

① 得到的 元(token)是 the, quick, brown, fox

只要 和 索的分析 程是一 的，不管用 搜索 fox 是 FOX 都能得到一 的搜索 果。lowercase 器会将 FOX 的 求 fox 的 求， fox 和我 在倒排索引中存 的是同一个 元(token)。

了在分析 程中使用 token 器，我 可以 建一个 custom 分析器：

```
PUT /my_index
{
  "settings": {
    "analysis": {
      "analyzer": {
        "my_lowercaser": {
          "tokenizer": "standard",
          "filter":  [ "lowercase" ]
        }
      }
    }
  }
}
```

我 可以通 analyze API 来 :

```
GET /my_index/_analyze?analyzer=my_lowercaser
The QUICK Brown FOX! ①
```

① 得到的 元是 the, quick, brown, fox

如果有口音

英 用 音符号(例如 ´, ^, 和 ¨) 来 —例如 rôle, déjà, 和 däis —但是是否使用他 通常是可 的. 其他 言 通 音符号来区分 。当然，只是因 在 的索引中 写正 的 并不意味着用 将搜索正 的 写。 去掉 音符号通常是有用的， rôle role， 或者反 来。 于西方 言，可以用 asciifolding 字符 器来 个功能。 上，它不 能去掉 音符号。它会把Unicode字符化 ASCII来表示:

- ß ⇒ ss
- æ ⇒ ae
- ł ⇒ l
- ⇒ m
- ⇒ ??
- ⇒ 2
- ⇒ 6

像 lowercase 器一 ，asciifolding 不需要任何配置，可以被 custom 分析器直接使用:

```
PUT /my_index
{
  "settings": {
    "analysis": {
      "analyzer": {
        "folding": {
          "tokenizer": "standard",
          "filter":  [ "lowercase", "asciifolding" ]
        }
      }
    }
  }
}

GET /my_index?analyzer=folding
My œsophagus caused a débâcle ①
```

① 得到的 元 my, oesophagus, caused, a, debacle

保留原意

理所当然的，去掉 音符号会 失原意。 例如, 参考 三个 西班牙 :

esta
: 形容 *this* 的 性形式, 例如 *esta silla* (this chair) 和 *esta* (this one).

ésta
: esta 的古代用法.

está
: *estar* (to be) 的第三人称形式, 例如 *está feliz* (he is happy).

通常我 会合并前 个形式的 ，而去区分和他 不相同的第三个形式的 。 似的:

sé
: *saber* (to know) 的第一人称形式 例如 *Yo sé* (I know).

se
: 与 多 使用的第三人称反身代 , 例如 *se sabe* (it is known).

不幸的是，没有 的方法，去区分 些 保留 音符号和 些 去掉 音符号。而且很有可能，的用 也不知道.

相反， 我 文本做 次索引: 一次用原文形式，一次用去掉 音符号的形式:

```
PUT /my_index/_mapping/my_type
{
  "properties": {
    "title": { ①
      "type":           "string",
      "analyzer":       "standard",
      "fields": {
        "folded": { ②
          "type":       "string",
          "analyzer":   "folding"
        }
      }
    }
  }
}
```

① 在 title 字段用 standard 分析器，会保留原文的　音符号.

② 在 title.folded 字段用 folding 分析器，会去掉　音符号

可以使用 analyze API 分析 *Esta está loca* (This woman is crazy)　个句子，来　　字段映射:

```
GET /my_index/_analyze?field=title ①
Esta está loca

GET /my_index/_analyze?field=title.folded ②
Esta está loca
```

① 得到的　元 esta, está, loca

② 得到的　元 esta, esta, loca

可以用更多的文　来　　:

```
PUT /my_index/my_type/1
{ "title": "Esta loca!" }

PUT /my_index/my_type/2
{ "title": "Está loca!" }
```

在，我　可以通　　合所有的字段来搜索。在`multi_match`　　中通　　most_fields　mode　模式来合所有字段的　果:

```
GET /my_index/_search
{
  "query": {
    "multi_match": {
      "type":     "most_fields",
      "query":    "esta loca",
      "fields": [ "title", "title.folded" ]
    }
  }
}
```

通 validate-query API 来 行 个 可以 助 理解 是如何 行的:

```
GET /my_index/_validate/query?explain
{
  "query": {
    "multi_match": {
      "type":     "most_fields",
      "query":    "está loca",
      "fields": [ "title", "title.folded" ]
    }
  }
}
```

multi-match 会搜索在 title 字段中原文形式的 (está)，和在 title.folded 字段中去掉音符号形式的 esta:

```
(title:está        title:loca       )
(title.folded:esta title.folded:loca)
```

无 用 搜索的是 esta 是 está; 个文 都会被匹配，因 去掉 音符号形式的 在 title.folded 字段中。然而，只有原文形式的 在 title 字段中。此 外匹配会把包含原文形式 的文 排在果列表前面。

我 用 title.folded 字段来 大我 的 (*widen the net*)来匹配更多的文 ，然后用原文形式的 title 字段来把 度最高的文 排在最前面。在可以 了匹配数量 牲文本原意的情况下， 个技 可以被用在任何分析器里。

TIP

asciifolding 器有一个叫做 preserve_original 的 可以 来做索引，把 的原文 元(original token)和 理—折 后的 元(folded token)放在同一个字段的同一个位置。 了 个 ， 果会像 ：

```
Position 1     Position 2
-------------------------
(ésta,esta)    loca
-------------------------
```

然 个是 空 的好 法，但是也意味着没有 法再 “ 我精 匹配的原文 元”(Give me an exact match on the original word)。包含去掉和不去掉 音符号的 元，会 致不可 的相 性 分。

所以，正如我 一章做的，把 个字段的不同形式分 到不同的字段会 索引更清晰。

Unicode的世界

当Elasticsearch在比 元(token)的 候，它是 行字 (byte) 的比 。 句 ，如果 个元(token)被判定 相同的 ，他 必 是相同的字 (byte) 成的。然而，Unicode允 用不同的字 来写相同的字符。

例如， é 和 é 的不同是什 ？ 取决于 。 于Elasticsearch，第一个是由 <code>0xC3 0xA9</code> 个字 成的，第二个是由 <code>0x65 0xCC 0x81</code> 三个字 成的。

于Unicode，他 的差 和他 的 成没有 系，所以他 是相同的。第一个是 个 é ，第二个是一个 e 和重音符 ´。

如果 的数据有多个来源，就会有可能 生 状况：因 相同的 使用了不同的 ， 致一个形式的 déjà 不能和它的其他形式 行匹配。

幸 的是， 里就有解决 法。 里有4 Unicode 一化形式 (*normalization forms*) ： nfc, nfd, nfkc, nfkd，它 都把Unicode字符 成 准格式，把所有的字符 行字 (byte) 的比 。

Unicode 一化形式 (Normalization Forms)

```
_ 合_ (_composed_) 模式—`nfc` 和 `nfkc`—用尽可能少的字 (byte)来代表字符。
((("composed forms (Unicode normalization)"))) 所以用 `é` 来代表 个字母 `é` 。
_分解_ (_decomposed_) 模式—`nfd` and `nfkd`—用字符的 一部分来代表字符。所以
`é` 分解 `e` 和 `´`。 ((("decomposed forms (Unicode normalization)")))
```

(*canonical*) 模式—nfc 和 nfd&—把 字作 个字符，例如 或者 œ 。 兼容 (*compatibility*) 模式—nfkc 和 nfkd—将 些 合的字符分解成 字符的等 物，例如： f + f + i 或者 o + e.

无 一个 一化(normalization)模式，只要 的文本只用一 模式，那 的同一个 元(token)就会由相同的字 (byte) 成。例如，兼容 (*compatibility*) 模式 可以用 的 化形式 `ffi`来 行 比。

可以使用 icu_normalizer 元 器(token filters) 来保 的所有 元(token)是相同模式：

```
PUT /my_index
{
  "settings": {
    "analysis": {
      "filter": {
        "nfkc_normalizer": { ①
          "type": "icu_normalizer",
          "name": "nfkc"
        }
      },
      "analyzer": {
        "my_normalizer": {
          "tokenizer": "icu_tokenizer",
          "filter":  [ "nfkc_normalizer" ]
        }
      }
    }
  }
}
```

① 用 nfkc 一化(normalization)模式来 一化(Normalize)所有 元(token).

TIP

包括 才提到 的 icu_normalizer 元 器(token filters)在内， 里 有 icu_normalizer 字符 器(*character* filters)。 然它和 元 器做相同的工作，但是会在文本到 器之前做。到底是用`standard` 器， 是 icu_tokenizer 器，其 并不重要。因 器知道 来正 理所有的模式。

但是，如果 使用不同的分 器，例如： ngram，edge_ngram，或者 pattern 分 器，那 在 元 器(token filters)之前使用 icu_normalizer 字符 器就 得有意 了。

通常来 ， 不 想要 一化(normalize) 元(token)的字 (byte) ， 需要把他 成小写字母。 个可以通 icu_normalizer 和定制的 一化(normalization)的模式 nfkc_cf 来 。下一 我会具体 个。

Unicode 大小写折

人 没有 造力的 就不会是人 ， 而人 的 言就恰恰反映了 一点。

理一个 的大小写看起来是一个 的任 ，除非遇到需要 理多 言的情况。

那就 一个例子： 小写 国 ß。把它 成大写是 SS，然后在 成小写就成了 ss。 有一个例子： 希 字母 ς (sigma，在 末尾使用)。把它 成大写是 Σ，然后再 成小写就成了 σ。

把 条小写的核心是 他 看起来更像，而不是更不像。在Unicode中， 个工作是大小写折 (case folding)来完成的，而不是小写化(lowercasing)。 大小写折 (*Case folding*) 把 到一 (通常是小写)形式，是 写法不会影 的比 ，所以 写不需要完全正 。

例如： ß，已 是小写形式了，会被_折 _(*folded*)成 ss。 似的小写的 ς 被折 成 σ， 的 ，无 σ， ς，和 `Σ`出 在 里，他 就都可以比 了。

`icu_normalizer` 元 器 的 一化(normalization)模式是 `nfkc_cf`。它像 `nfkc` 模式一 ：

- 合 (*Composes*) 字符用最短的字 来表示。
- 用 兼容 (*compatibility*）模式，把像 的字符 成 的 ffi

但是，也会 做：

- 大小写折 (*Case-folds*) 字符成一 合比 的形式

句 ， nfkc_cf`等 于 `lowercase 元 器(token filters)，但是却 用于所有的 言。 *on-steroids* 等 于 standard 分析器，例如：

```
PUT /my_index
{
  "settings": {
    "analysis": {
      "analyzer": {
        "my_lowercaser": {
          "tokenizer": "icu_tokenizer",
          "filter":  [ "icu_normalizer" ] ①
        }
      }
    }
  }
}
```

① icu_normalizer 是 nfkc_cf 模式.

我 来比 Weißkopfseeadler`和 `WEISSKOPFSEEADLER(大写形式) 分 通 `standard`分析器和我的Unicode自 (Unicode-aware)分析器 理得到的 果：

```
GET /_analyze?analyzer=standard ①
Weißkopfseeadler WEISSKOPFSEEADLER

GET /my_index/_analyze?analyzer=my_lowercaser ②
Weißkopfseeadler WEISSKOPFSEEADLER
```

① 得到的 元(token)是 weißkopfseeadler, weisskopfseeadler

② 得到的 元(token)是 weisskopfseeadler, weisskopfseeadler

> `standard`分析器得到了 个不同且不可比 的 元(token)，而我 定制化的分析器得到了个相同但是不符合原意的 元(token)。

Unicode 字符折

在多 言(("Unicode", "character folding")))(("tokens", "normalizing", "Unicode character folding"))) 理中，`lowercase` 元 器(token filters)是一个很好的 始。但是作 比的 ，也只是 于整个巴 塔的 一瞥。所以 <<asciifolding-token-filter,`asciifolding` token filter>> 需要更有效的Unicode _字符折 _ (_character-folding_)工具来 理全世界的各 言。(("asciifolding token filter")))

`icu_folding` 元 器(token filters) (provided by the <<icu-plugin,`icu` plugin>>)的功能和 `asciifolding` 器一 ， (("icu_folding token filter")))但是它 展到了非ASCII 的 言，例如：希 ，希伯来 ， 。它把 些 言都 拉丁文字，甚至包含它 的各 各 的 数符号，象形符号和 点符号。

`icu_folding` 元 器(token filters)自 使用 `nfkc_cf` 模式来 行大小写折和Unicode 一化(normalization)，所以不需要使用 `icu_normalizer` ：

```
PUT /my_index
{
  "settings": {
    "analysis": {
      "analyzer": {
        "my_folder": {
          "tokenizer": "icu_tokenizer",
          "filter":  [ "icu_folding" ]
        }
      }
    }
  }
}

GET /my_index/_analyze?analyzer=my_folder
           ①
```

① 阿拉伯数字 被折 成等 的拉丁数字: 12345.

如果 有指定的字符不想被折 ， 可以使用 *UnicodeSet*(像字符的正 表 式) 来指定 些Unicode才可以被折 。例如：瑞典 å,ä, ö, Å, Ä, 和 Ö 不能被折 ， 就可以 定 ： [^åäöÅÄÖ] (^表示 不包含)。 就会 于所有的Unicode字符生效。

```
PUT /my_index
{
  "settings": {
    "analysis": {
      "filter": {
        "swedish_folding": { ①
          "type": "icu_folding",
          "unicodeSetFilter": "[^åäöÅÄÖ]"
        }
      },
      "analyzer": {
        "swedish_analyzer": { ②
          "tokenizer": "icu_tokenizer",
          "filter":  [ "swedish_folding", "lowercase" ]
        }
      }
    }
  }
}
```

① `swedish_folding` 元 器(token filters) 定制了 `icu_folding` 元 器(token filters)来不 理那些大写和小写的瑞典 。

② swedish 分析器首先分 ，然后用`swedish_folding` 元 器来折 ，最后把他 走小写，除了被排除在外的 ：Å, Ä, 或者 Ö。

排序和整理

本章到目前 止，我 已 了解了 以搜索 目的去 化 元。 本章 中要考 的最用例是字符串排序。

在 [multi-fields] （ 数域）中，我 解 了 Elasticsearch 什 不能在 analyzed （分析 ）的字符串字段上排序，并演示了如何 同一个域 建 数域索引 ，其中 analyzed 域用来搜索，not_analyzed 域用来排序。

analyzed 域无法排序并不是因 使用了分析器，而是因 分析器将字符串拆分成了很多 元，就像一个 袋， 所以 Elasticsearch 不知道使用那一个 元排序。

依 于 not_analyzed 域来排序的 不是很 活： 允 我 使用原始字符串 一 定的排序。然而我 可以 使用分析器来 外一 排序 ，只要 的分析器 是 个字符串出有且 有一个的 元。

大小写敏感排序

想象下我 有三个 用 文 ，文 的 姓名 域分 含有 Boffey 、 BROWN 和 bailey 。首先我 将使用在 [multi-fields] 中提到的技 ，使用 not_analyzed 域来排序：

```
PUT /my_index
{
  "mappings": {
    "user": {
      "properties": {
        "name": { ①
          "type": "string",
          "fields": {
            "raw": { ②
              "type":  "string",
              "index": "not_analyzed"
            }
          }
        }
      }
    }
  }
}
```

① analyzed name 域用来搜索。

② not_analyzed name.raw 域用来排序。

我 可以索引一些文 用来 排序：

```
PUT /my_index/user/1
{ "name": "Boffey" }

PUT /my_index/user/2
{ "name": "BROWN" }

PUT /my_index/user/3
{ "name": "bailey" }

GET /my_index/user/_search?sort=name.raw
```

行 个搜索 求将会返回 的文 排序： BROWN 、 Boffey 、 bailey 。 个是 典排序 跟 字符串排序相反。基本上就是大写字母 的字 要比小写字母 的字 重低，所以 些姓名是按照最低 先排序。

可能 算机是合理的，但是 人来 并不是那 合理，人 更期望 些姓名按照字母 序排序，忽略大小写。 了 个，我 需要把 个姓名按照我 想要的排序的 序索引。

句 来 ，我 需要一个能 出 个小写 元的分析器：

```
PUT /my_index
{
  "settings": {
    "analysis": {
      "analyzer": {
        "case_insensitive_sort": {
          "tokenizer": "keyword",    ①
          "filter":  [ "lowercase" ] ②
        }
      }
    }
  }
}
```

① keyword 分　器将　入的字符串原封不　的　出。

② lowercase 分　　器将　　元　化　小写字母。

使用 大小写不敏感排序 分析器替　后，　在我　可以将其用在我　的　数域：

```
PUT /my_index/_mapping/user
{
  "properties": {
    "name": {
      "type": "string",
      "fields": {
        "lower_case_sort": { ①
          "type":     "string",
          "analyzer": "case_insensitive_sort"
        }
      }
    }
  }
}

PUT /my_index/user/1
{ "name": "Boffey" }

PUT /my_index/user/2
{ "name": "BROWN" }

PUT /my_index/user/3
{ "name": "bailey" }

GET /my_index/user/_search?sort=name.lower_case_sort
```

① name.lower_case_sort 域将会　我　提供大小写不敏感排序。

行　个搜索　求会得到我　想要的文　排序： bailey 、 Boffey 、 BROWN 。

但是 个 序是正 的 ？它符合我 的期望所以看起来像是正 的， 但我 的期望可能受到 个事的影 ： 本 是英文的，我 的例子中使用的所有字母都属于到英 字母表。

如果我 添加一个 姓名 *Böhm* 会 ？

在我 的姓名会返回 的排序： `bailey` 、 `Boffey` 、 `BROWN` 、 `Böhm` 。 `Böhm` 会排在 `BROWN` 后面的原因是 些 依然是按照它 表 的字 排序的。 `r` 所存 的字 `0x72` ，而 `ö` 存的字 `0xF6` ，所以 `Böhm` 排在最后。 个字符的字 都是 史的意外。

然， 排序 序 于除 英 之外的任何事物都是无意 的。事 上，没有完全“正 ”的排序 。完全取决于 使用的 言。

言之 的区

言都有自己的排序 ，并且 有 候甚至有多 排序 。 里有几个例子，我 前一小中的四个名字在不同的上下文中是 排序的：

- 英 ：`bailey`、 `boffey`、 `böhm`、 `brown`
- ：`bailey`、 `boffey`、 `böhm`、 `brown`
- 簿：`bailey`、 `böhm`、 `boffey`、 `brown`
- 瑞典 ：`bailey`, `boffey`, `brown`, `böhm`

NOTE	簿将 `böhm` 放在 `boffey` 的原因是 `ö` 和 `oe` 在 理名字和地点的 候会被看成同 ，所以 `böhm` 在排序 像是被写成了 `boehm` 。

Unicode 算法

是将文本按 定 序排序的 程。 *Unicode* 算法 或称 UCA （参 *www.unicode.org/reports/tr10* ） 定 了一 将字符串按照在 元表中定 的序排序的方法（通常称 排序 ）。

UCA 定 了 *Unicode* 排序 元素表 或称 *DUCET* ， *DUCET* 无 任何 言的所有 Unicode 字符定 了 排序。如 所 ，没有惟一一个正 的排序 ，所以 DUCET 更少的人感到 ，且 尽可能的小，但它 不是解决所有排序 的万能 。

而且，明 几乎 言都有自己的排序 。大多 候使用 DUCET 作 起点并且添加一些自定 用来 理 言的特性。

UCA 将字符串和排序 作 入，并 出二 制排序 。 将根据指定的排序 字符串集合行排序 化 其二 制排序 的 比 。

Unicode 排序

TIP	本 中描述的方法可能会在未来版本的 Elasticsearch 中更改。 看 `icu` plugin 文 的最新信息。

`icu_collation` 分 器 使用 DUCET 排序 。 已 是 排序的改 了。想要使用 `icu_collation` 我 需要 建一个使用 `icu_collation` 器的分析器：

```
PUT /my_index
{
  "settings": {
    "analysis": {
      "analyzer": {
        "ducet_sort": {
          "tokenizer": "keyword",
          "filter": [ "icu_collation" ] ①
        }
      }
    }
  }
}
```

① 使用 DUCET 。

通常，我 想要排序的字段就是我 想要搜索的字段， 因此我 使用与在 大小写敏感排序 中使用的相同的 数域方法：

```
PUT /my_index/_mapping/user
{
  "properties": {
    "name": {
      "type": "string",
      "fields": {
        "sort": {
          "type": "string",
          "analyzer": "ducet_sort"
        }
      }
    }
  }
}
```

使用 个映射， `name.sort` 域将会含有一个 用来排序的 。我 没有指定某 言，所以它会 会使用 DUCET collation 。

在，我 可以重新索引我 的案例文 并 排序：

```
PUT /my_index/user/_bulk
{ "index": { "_id": 1 }}
{ "name": "Boffey" }
{ "index": { "_id": 2 }}
{ "name": "BROWN" }
{ "index": { "_id": 3 }}
{ "name": "bailey" }
{ "index": { "_id": 4 }}
{ "name": "Böhm" }

GET /my_index/user/_search?sort=name.sort
```

NOTE 注意，个文返回的 sort ，在前面的例子中看起来像 brown 和 böhm ，在看起来像天 : 乏 \u0001 。原因是 icu_collation 器出用于有效分 ，不用于任何其他目的。

行个搜索求反的文排序 : bailey 、 Boffey 、 Böhm 、 BROWN 。个排序英和来都正 ，已是一 ，但是它簿和瑞典来不正 。下一我不同的言自定映射。

指定言

可以特定的言配置使用表的 icu_collation 器，例如一个国家特定版本的言，或者像 簿之的子集。个可以按照如下所示通使用 language 、 country 、 和 variant 参数来建自定版本的分器：

英

```
{ "language": "en" }
```

```
{ "language": "de" }
```

奥地利

```
{ "language": "de", "country": "AT" }
```

簿

```
{ "language": "de", "variant": "@collation=phonebook" }
```

TIP 可以在一下址更多的 ICU 本地支持： http://userguide.icu-project.org/locale.

个例子演示建簿排序 :

```
PUT /my_index
{
  "settings": {
    "number_of_shards": 1,
    "analysis": {
      "filter": {
        "german_phonebook": { ①
          "type":     "icu_collation",
          "language": "de",
          "country":  "DE",
          "variant":  "@collation=phonebook"
        }
      },
      "analyzer": {
        "german_phonebook": { ②
          "tokenizer": "keyword",
          "filter":  [ "german_phonebook" ]
        }
      }
    }
  },
  "mappings": {
    "user": {
      "properties": {
        "name": {
          "type": "string",
          "fields": {
            "sort": { ③
              "type":     "string",
              "analyzer": "german_phonebook"
            }
          }
        }
      }
    }
  }
}
```

① 首先我　　薄　建一个自定　版本的 `icu_collation` 。

② 之后我　将其包装在自定　的分析器中。

③ 并且　我　的 `name.sort` 域配置它。

像我　之前那　重新索引并重新搜索：

```
PUT /my_index/user/_bulk
{ "index": { "_id": 1 }}
{ "name": "Boffey" }
{ "index": { "_id": 2 }}
{ "name": "BROWN" }
{ "index": { "_id": 3 }}
{ "name": "bailey" }
{ "index": { "_id": 4 }}
{ "name": "Böhm" }

GET /my_index/user/_search?sort=name.sort
```

在返回的文 排序 ：bailey 、 Böhm 、 Boffey 、 BROWN 。在 簿 中， Böhm 等同于 Boehm ，所以排在 Boffey 前面。

多排序

言都可以使用 数域来支持 同一个域 行多 排序：

```
PUT /my_index/_mapping/_user
{
  "properties": {
    "name": {
      "type": "string",
      "fields": {
        "default": {
          "type":     "string",
          "analyzer": "ducet" ①
        },
        "french": {
          "type":     "string",
          "analyzer": "french" ①
        },
        "german": {
          "type":     "string",
          "analyzer": "german_phonebook" ①
        },
        "swedish": {
          "type":     "string",
          "analyzer": "swedish" ①
        }
      }
    }
  }
}
```

① 我 需要 个排序 建相 的分析器。

使用 个映射，只要按照 name.french 、 name.german 或 name.swedish 域排序，就可以 法 、

和瑞典 用 正 的排序 果了。不支持的 言可以回退到使用 `name.default` 域，它使用 DUCET 排序 序。

自定 排序

`icu_collation` 分 器提供很多 ，不止 `language` 、 `country` 、和 `variant` ， 些 可以用于定制排序算法。可用的 有以下作用：

- 忽略 音符号
- 序大写排先或排后，或忽略大小写
- 考 或忽略 点符号和空白
- 将数字按字符串或数字 排序
- 自定 有 或定 自己的

些 的 信息超出了本 的 ，更多的信息可以 ICU plug-in documentation 和 ICU project collation documentation 。

将 原 根

大多数 言的 都可以 形 化，意味着下列 可以改 它 的形 用来表 不同的意思：

- 数 化 ： fox 、 foxes
- 化 ： pay 、 paid 、 paying
- 性 化 ： waiter 、 waitress
- 人称 化 ： hear 、 hears
- 代 化 ： I 、 me 、 my
- 不 化 ： ate 、 eaten
- 情景 化 ： so be it 、 were it so

然 形 化有助于表 ，但它干 了 索，一个 一的 根 （或意 ）可能被很多不同的字母序列表 。 英 是一 弱 形 化 言（ 可以忽略 形 化并且能得到合理的搜索 果），但是一些其他 言是高度 形 化的并且需要 外的工作来保 高 量的搜索 果。

干提取 移除 的 化形式之 的差 ，从而 到将 个 都提取 它的 根形式。 例如 `foxes` 可能被提取 根 `fox` ，移除 数和 数之 的区 跟我 移除大小写之 的区 的方式是一 的。

的 根形式甚至有可能不是一个真的 ， `jumping` 和 `jumpiness` 或 都会被提取 干 `jumpi` 。 并没有什 —只要在索引 和搜索 生相同的 ，搜索会正常的工作。

如果 干提取很容易的 ，那只要一个 件就 了。不幸的是， 干提取是一 遭受 困 的模糊的技 ： 干弱提取和 干 度提取。

干弱提取 就是无法将同 意思的 同一个 根。例如， `jumped` 和 `jumps` 可能被提取 `jump` ，但是 `jumping` 可能被提取 `jumpi` 。弱 干提取会 致搜索 无法返回相 文 。

干 度提取 就是无法将不同含 的 分 。例如， general 和 generate 可能都被提取 gener 。干 度提取会降低精准度：不相干的文 会在不需要他 返回的 候返回。

> 形 原
>
> 原 是一 相 的 形式，或 典形式—paying 、 paid 和 pays 的原 是 pay 。 通常原 很像与其相 的 ，但有 也不像—is 、 was 、 am 和 being 的原 是 be 。
>
> 形 原，很像 干提取， 相 ，但是它比 干提取先 一 的是它企 按 的   ，或意 。 同 的 可能表 出 意思—例如，wake 可以表 to wake up 或 a funeral 。然而 形原 区分 个 的 ， 干提取却会将其混 一 。
>
> 形 原是一 更 和高 源消耗的 程，它需要理解 出 的上下文来决定 的意思。 践中 ， 干提取似乎比 形 原更高效，且代 更低。

首先我 会 下 个 Elasticsearch 使用的 典 干提取器 — 干提取算法 和 字典 干提取器 — 并且在 一个 干提取器 了 根据 的需要 合 的 干提取器。 最后将在 控制 干提取 和 原形 干提取 中 如何裁剪 干提取。

干提取算法

Elasticsearch 中的大部分 stemmers （ 干提取器）是基于算法的，它 提供了一系列 用于将一个 提取 它的 根形式，例如剥 数 末尾的 s 或 es 。提取 干 并不需要知道 的任何信息。

些基于算法的 stemmers 点是：可以作 件使用，速度快，占用内存少，有 律的 理效果好。 点是：没 律的 例如 be 、 are 、和 am ， 或 mice 和 mouse 效果不好。

最早的一个基于算法的英文 干提取器是 Porter stemmer ， 英文 干提取器 在依然推 使用。Martin Porter 后来 了 干提取算法 建了 Snowball language 站， 很多 Elasticsearch 中使用的 干提取器就是用 Snowball 言写的。

TIP

{ref}/analysis-kstem-tokenfilter.html[kstem token filter] 是一款合并了 干提取算法和内置 典的英 分 器。 了避免模糊 不正 提取， 个 典包含一系列根 和特例 。 kstem 分 器相 于 Porter 干提取器而言不那 激 。

使用基于算法的 干提取器

可以使用 {ref}/analysis-porterstem-tokenfilter.html[porter_stem] 干提取器或直接使用 {ref}/analysis-kstem-tokenfilter.html[kstem] 分 器，或使用 {ref}/analysis-snowball-tokenfilter.html[snowball] 分 器 建一个具体 言的 Snowball 干提取器。所有基于算法的 干提取器都暴露了用来接受 言 参数的 一接口： {ref}/analysis-stemmer-tokenfilter.html[stemmer token filter] 。

例如，假 英 分析器使用的 干提取器太激 并且 想使它不那 激 。首先 在

{ref}/analysis-lang-analyzer.html[language analyzers] 看 英 分析器配置文件，配置文件展示如下：

```
{
  "settings": {
    "analysis": {
      "filter": {
        "english_stop": {
          "type":       "stop",
          "stopwords":  "_english_"
        },
        "english_keywords": {
          "type":       "keyword_marker", ①
          "keywords":   []
        },
        "english_stemmer": {
          "type":       "stemmer",
          "language":   "english" ②
        },
        "english_possessive_stemmer": {
          "type":       "stemmer",
          "language":   "possessive_english" ②
        }
      },
      "analyzer": {
        "english": {
          "tokenizer":  "standard",
          "filter": [
            "english_possessive_stemmer",
            "lowercase",
            "english_stop",
            "english_keywords",
            "english_stemmer"
          ]
        }
      }
    }
  }
}
```

① keyword_marker 分 器列出那些不用被 干提取的 。 个 器 情况下是一个空的列表。

② english 分析器使用了 个 干提取器： possessive_english 干提取器和 english 干提取器。所有格 干提取器会在任何 到 english_stop 、 english_keywords 和 english_stemmer 之前去除 's 。

重新 下 在的配置，添加上以下修改，我 可以把 配置当作新分析器的基本配置：

- 修改 english_stemmer ，将 english （{ref}/analysis-porterstem-tokenfilter.html[porter_stem] 分 器的映射）替 light_english （非激 的 {ref}/analysis-kstem-tokenfilter.html[kstem] 分 器的映射）。

- 添加 asciifolding 分　　器用以移除外　的附加符号。
- 移除 keyword_marker 分　　器，因　我　不需要它。（我　会在 控制　干提取 中　　它）

新定　的分析器会像下面　　：

```
PUT /my_index
{
  "settings": {
    "analysis": {
      "filter": {
        "english_stop": {
          "type":       "stop",
          "stopwords":  "_english_"
        },
        "light_english_stemmer": {
          "type":       "stemmer",
          "language":   "light_english" ①
        },
        "english_possessive_stemmer": {
          "type":       "stemmer",
          "language":   "possessive_english"
        }
      },
      "analyzer": {
        "english": {
          "tokenizer":  "standard",
          "filter": [
            "english_possessive_stemmer",
            "lowercase",
            "english_stop",
            "light_english_stemmer", ①
            "asciifolding" ②
          ]
        }
      }
    }
  }
}
```

① 将 english　干提取器替　　非激　的 light_english　干提取器

② 添加 asciifolding 分　　器

字典　干提取器

字典　干提取器　在工作机制上与　算法化　干提取器　完全不同。　不同于　用一系列　准　　到　个　上，字典　干提取器只是　　地在字典里　　。理　上可以　出比算法化　干提取器更好的　果。一个字典　干提取器　当可以：

- 返回不　　形式如 feet 和 mice 的正　　干

- 区分出　形相似但　　不同的情形，比如 `organ` and `organization`

践中一个好的算法化　干提取器一般　于一个字典　干提取器。　　有以下　大原因：

字典　量

一个字典　干提取器再好也就跟它的字典一　。　　据牛津英　字典　站估　，英　包含大　75万个　（包含　音　形　）。　　上的大部分英　字典只包含其中的 10% 。

的含　随　光　　。`mobility`　提取　干　`mobil`　先前可能　得通，但　在合并　了手机可移　性的含　。字典需要保持最新，　是一　很耗　的任　。通常等到一个字典　得好用后，其中的部分内容已　。

字典　干提取器　于字典中不存在的　无能　力。而一个基于算法的　干提取器，　会　　用之前的相同　，　果可能正　或　　。

大小与性能

字典　干提取器需要加　所有　、　　所有前　，以及所有后　到内存中。　会　著地消耗内存。　到一个　的正　　干，一般比算法化　干提取器的相同　程更加　　。

依　于不同的字典　量，去除前后　的　程可能会更加高效或低效。低效的情形可能会明　地　慢整个　干提取　程。

一方面，算法化　干提取器通常更　　、　量和快速。

TIP	如果　所使用的　言有比　好的算法化　干提取器，　通常是比一个基于字典的　干提取器更好的　　。　于算法化　干提取器效果比　差（或者　根没有）的　言，可以使用　写　　（Hunspell）字典　干提取器，下一个章　会　　。

Hunspell　干提取器

Elasticsearch　提供了基于　典提取　干的　{ref}/analysis-hunspell-tokenfilter.html[`hunspell` 元　　器（token filter）]. Hunspell *hunspell.github.io* 是一个 Open Office、LibreOffice、Chrome、Firefox、Thunderbird 等　多其它　源　目都在使用的　写　　器。

可以从　里　取 Hunspell　典：

- *extensions.openoffice.org*: 下　解　`.oxt` 后　的文件。
- *addons.mozilla.org*: 下　解　`.xpi`　展文件。
- OpenOffice archive: 下　解　`.zip` 文件。

一个　　Hunspell　　典由　个文件　成 — 具有相同的文件名和个不同的后　 — 如 <code>en_US</code>—和下面的　个后　的其中一个：

`.dic`

包含所有　根，采用字母　序，再加上一个代表所有可能前　和后　的代　表【集体称之　　（*affixes*）】

`.aff`

包含　　`.dic` 文件　一行代　表　　的前　和后

安装一个　典

Hunspell　　元　　器在特定的 Hunspell 目　里　　典，　　目　是 `./config/hunspell/` 。 `.dic` 文件和　`.aff`　文件　　要以子目　且按　言/区域的方式来命名。　例如，我　可以　美式英　　建一个 Hunspell　干提取器，目　　如下：

```
config/
   └ hunspell/ ①
       └ en_US/ ②
           ├ en_US.dic
           ├ en_US.aff
           └ settings.yml ③
```

① Hunspell　　目　位置可以通　　`config/elasticsearch.yml`　　文件的：`indices.analysis.hunspell.dictionary.location`　置来修改。

② `en_US` 是　个区域的名字，也是我　　`hunspell`　　元　　器参数 `language`　。

③ 一个　言一个　置文件，下面的章　会具体介　。

按　言　置

在　言的目　　置文件 `settings.yml` 包含　用于所有字典内的　言目　的　置　　。

```
---
ignore_case:          true
strict_affix_parsing: true
```

　些　　的意思如下：

`ignore_case`
　Hunspell 目　　是区分大小写的，如，姓氏 `Booker` 和名　`booker` 是不同的　，所以　　分　　行干提取。也　　`hunspell` 提取器区分大小写是一个好主意，不　也可能　事情　得　　：

- 一个句子的第一个　可能会被大写，因此感　上会像是一个名　。
- 　入的文本可能全是大写，如果　　那几乎一个　都　不到。
- 用　也　会用小写来搜索名字，在　　情况下，大写　　的　将　不到。

　一般来　，　置参数 `ignore_case`　`true` 是一个好主意。

`strict_affix_parsing`
　典的　量千差万　。　一些　上的　典的 `.aff` 文件有很多畸形的　　。　　情况下，如果 Lucene 不能正常解析一个　(affix)　，　它会　出一个　常。　　可以通　　置 `strict_affix_parsing` `false` 来告　Lucene 忽略　　的　　。

自定　　典

如果一个目　放置了多个　典　(.dic　文件)，　他　会在加　　合并到一起。　可以　　以自定　的　典的方式　下　的　典　行定制：

```
config/
   └ hunspell/
       └ en_US/  ①
           ├ en_US.dic
           ├ en_US.aff ②
           ├ custom.dic
           └ settings.yml
```

① custom　典和 en_US　典将合并到一起。

② 多个 .aff 文件是不允　的，因　会　生　　冲突。

.dic 文件和 .aff 文件的格式在　里　　：Hunspell　典格式。

建一个 Hunspell　　元　　器

一旦　在所有　点上安装好了　典，　就能像　　定　一个 hunspell　　元　　器：

```
PUT /my_index
{
  "settings": {
    "analysis": {
      "filter": {
        "en_US": {
          "type":     "hunspell",
          "language": "en_US" ①
        }
      },
      "analyzer": {
        "en_US": {
          "tokenizer":  "standard",
          "filter":   [ "lowercase", "en_US" ]
        }
      }
    }
  }
}
```

① 参数 language 和目　下　　的名称相同。

可以通　analyze API 来　　个新的分析器，　然后和 english 分析器比　一下它　的　出：

```
GET /my_index/_analyze?analyzer=en_US ①
reorganizes

GET /_analyze?analyzer=english ②
reorganizes
```

① 返回 organize

② 返回 reorgan

在前面的例子中，hunspell 提取器有一个有意思的事情，它不 能移除前 能移除后 。大多数算法 干提取 能移除后 。

TIP	Hunspell 典会占用几兆的内存。幸 的是，Elasticsearch 个 点只会 建一个 典的 例。 所有的分片都会使用 个相同的 Hunspell 分析器。

Hunspell 典格式

尽管使用 hunspell 不必了解 Hunspell 典的格式， 不 了解格式可以 助我 写自己的自定 的 典。其 很 。

例如，在美式英 典（US English dictionary），en_US.dic 文件包含了一个包含 analyze 的 体，看起来如下：

```
analyze/ADSG
```

en_US.aff 文件包含了一个 A 、 G 、D 和 S 的前后 的 。 其中 只有一个能匹配，一个 的格式如下：

```
[type] [flag] [letters to remove] [letters to add] [condition]
```

例如，下面的后 (SFX) D 。它是 ，当一个 由一个 音 (除了 a 、e 、i 、o 或 u 外的任意音) 后接一个 y ，那 它可以移除 y 和添加 ied 尾 （如，ready → readied ）。

```
SFX    D      y    ied  [^aeiou]y
```

前面提到的 A 、 G 、D 和 S 如下：

```
SFX D Y 4
SFX D   0     d          e ①
SFX D   y     ied        [^aeiou]y
SFX D   0     ed         [^ey]
SFX D   0     ed         [aeiou]y

SFX S Y 4
SFX S   y     ies        [^aeiou]y
SFX S   0     s          [aeiou]y
SFX S   0     es         [sxzh]
SFX S   0     s          [^sxzhy] ②

SFX G Y 2
SFX G   e     ing        e ③
SFX G   0     ing        [^e]

PFX A Y 1
PFX A   0     re         . ④
```

① analyze 以一个 e 尾，所以它可以添加一个 d 成 analyzed 。

② analyze 不是由 s 、 x 、 z 、 h 或 y 尾，所以，它可以添加一个 s 成 analyzes 。

③ analyze 以一个 e 尾，所以，它可以移除 e 和添加 ing 然后 成 analyzing 。

④ 可以添加前 re 来形成 reanalyze 。 个 可以 合后 一起形成： reanalyzes 、 reanalyzed 、 reanalyzing 。

了解更多 于 Hunspell 的 法，可以前往 Hunspell 文 。

一个 干提取器

在文 {ref}/analysis-stemmer-tokenfilter.html[stemmer] token filter 里面列出了一些 言的若干干提取器。 就英 来 我 有如下提取器：

`english`
: {ref}/analysis-porterstem-tokenfilter.html[porter_stem] 元 器（token filter）。

`light_english`
: {ref}/analysis-kstem-tokenfilter.html[kstem] 元 器（token filter）。

`minimal_english`
: Lucene 里面的 EnglishMinimalStemmer ，用来移除 数。

`lovins`
: 基于 {ref}/analysis-snowball-tokenfilter.html[Snowball] 的 Lovins 提取器, 第一个 干提取器。

`porter`
: 基于 {ref}/analysis-snowball-tokenfilter.html[Snowball] 的 Porter 提取器。

`porter2`
: 基于 {ref}/analysis-snowball-tokenfilter.html[Snowball] 的 Porter2 提取器。

`possessive_english`
Lucene 里面的 `EnglishPossessiveFilter`，移除 `'s`

Hunspell 干提取器也要 入到上面的列表中， 有多 英文的 典可用。

有一点是可以肯定的：当一个 存在多个解决方案的 候， 意味着没有一个解决方案充分解决 个 。 一点同 体 在 干提取上 — 个提取器使用不同的方法不同程度的 行了弱提取或是 度提取。

在 `stemmer` 文 中，使用粗体高亮了 一个 言的推 的 干提取器， 通常是因它提供了一个在性能和 量之 合理的妥 。也就是 ，推 的 干提取器也 不 用所有 景。 于 个是最好的 干提取器，不存在一个唯一的正 答案 — 它要看 具体的需求。 里有3个方面的因素需要考 在内： 性能、 量、程度。

提取性能

算法提取器一般来 比 Hunspell 提取器快4到5倍。 `Handcrafted'' 算法提取器通常（不是永 ） 要比 Snowball 快或是差不多。 比如，`porter_stem 元 器（token filter）就明 要比基于 Snowball 的 Porter 提取器要快的多。

Hunspell 提取器需要加 所有的 典、前 和后 表到内存，可能需要消耗几兆的内存。而算法提取器，由一点点代 成，只需要使用很少内存。

提取 量

所有的 言，除了世界 （Esperanto）都是不 的。 最日常用 使用的 往往不 ，而更正式的 面用 往往遵循 律。 一些提取算法 多年的 和研究已 能 生合理的高 量的 果了，其他人只需快速 装做很少的研究就能解决大部分的 了。

然 Hunspell 提供了精 地 理不 的承 ，但在 践中往往不足。 一个基于 典的提取器往往取决于 典的好坏。如果 Hunspell 到的 个 不在 典里，那它什 也不能做。Hunspell 需要一个广泛的、高 量的、最新的 典以 生好的 果； 的 典可 少之又少。 一方面，一个算法提取器，将愉快的 理新 而不用 新 重新 算法。

如果一个好的算法 干提取器可用于 的 言，那明智的使用它而不是 Hunspell。它会更快并且消耗更少内存，并且会 生和通常一 好或者比 Hunspell 等 的 果.

如果精度和可定制性 很重要，那 需要（和有精力）来 一个自定 的 典，那 Hunspell 会 比算法提取器更大的 活性。(看 控制 干提取 来了解可用于任何 干提取器的自定 技 。)

提取程度

不同的 干提取器会将 弱提取或 度提取到一定的程度。 `light_` 提取器提干力度不及 准的提取器。`minimal_` 提取器同 也不那 。Hunspell 提取力度要激 一些。

是否想要 提取 是 量提取取决于 的 景。如果 的搜索 果是要用于聚 算法， 可能会希望匹配的更广泛一点（因此，提取力度要更大一点）。 如果 的搜索 果是面向最 用 ， 量的提取一般会 生更好的 果。 搜索来 ，将名称和形容 提干比 提干更重要，当然 也取决于 言。

外一个要考 的因素就是 的文 集的大小。 一个只有 10,000 个 品的小集合， 可能要更激 的提干来 保至少匹配到一些文 。 如果 的文 集很大，使用 量的弱提取可能会得到更好的匹配

果。

做一个

从推 的一个 干提取器出 ，如果它工作的很好，那没有什 需要 整的。如果不是， 将需要花点 来 和比 言可用的各 不同提取器， 来 到最 合 目的的那一个。

控制 干提取

箱即用的 干提取方案永 也不可能完美。 尤其是算法提取器，他 可以愉快的将 用于任何他 遇到的 ，包含那些 希望保持独立的 。 也 ，在 的 景，保持独立的 `skies` 和 `skiing` 是重要的， 不希望把他 提取 `ski` （正如 `english` 分析器那 ）。

元 器 {ref}/analysis-keyword-marker-tokenfilter.html[`keyword_marker`] 和 {ref}/analysis-stemmer-override-tokenfilter.html[`stemmer_override`] 能 我 自定 干提取 程。

阻止 干提取

言分析器（ 看 配置 言分析器）的参数 `stem_exclusion` 允 我 指定一个 列表， 他 不被干提取。

在内部， 些 言分析器使用 {ref}/analysis-keyword-marker-tokenfilter.html[`keyword_marker` 元 器] 来 些 列表 *keywords*，用来阻止后 的 干提取 器来触 些 。

例如，我 建一个 自定 分析器，使用 {ref}/analysis-porterstem-tokenfilter.html[`porter_stem`] 元 器，同 阻止 `skies` 的 干提取：

```
PUT /my_index
{
  "settings": {
    "analysis": {
      "filter": {
        "no_stem": {
          "type": "keyword_marker",
          "keywords": [ "skies" ] ①
        }
      },
      "analyzer": {
        "my_english": {
          "tokenizer": "standard",
          "filter": [
            "lowercase",
            "no_stem",
            "porter_stem"
          ]
        }
      }
    }
  }
}
```

① 参数 keywords 可以允　接收多个　　。

使用 analyze API 来　　，可以看到　skies 没有被提取：

```
GET /my_index/_analyze?analyzer=my_english
sky skies skiing skis ①
```

① 返回: sky, skies, ski, ski

TIP

　然　言分析器只允　我　通　参数　stem_exclusion　指定一个　　列表来排除　干提取，不　　keyword_marker　　元　　器同　　接收一个　keywords_path　参数允　我将所有的　　字存在一个文件。　　个文件　　是　行一个字，并且存在于集群的　个　点。　看 更新停用　（Updating Stopwords）了解更新　些文件的提示。

自定　提取

在上面的例子中，我　阻止了　skies　被　干提取，但是也　我　希望他能被提干　　sky　。　The {ref}/analysis-stemmer-override-tokenfilter.html[stemmer_override]　　元　　器允　我指定自定　的提取　　。与此同　，我　可以　理一些不　　的形式，如：mice 提取　mouse 和 feet 到 foot ：

```
PUT /my_index
{
  "settings": {
    "analysis": {
      "filter": {
        "custom_stem": {
          "type": "stemmer_override",
          "rules": [ ①
            "skies=>sky",
            "mice=>mouse",
            "feet=>foot"
          ]
        }
      },
      "analyzer": {
        "my_english": {
          "tokenizer": "standard",
          "filter": [
            "lowercase",
            "custom_stem", ②
            "porter_stem"
          ]
        }
      }
    }
  }
}

GET /my_index/_analyze?analyzer=my_english
The mice came down from the skies and ran over my feet ③
```

① 来自 original⇒stem 。

② stemmer_override 器必 放置在 干提取器之前。

③ 返回 the, mouse, came, down, from, the, sky, and, ran, over, my, foot 。

TIP 正如 keyword_marker 元 器， 可以被存放在一个文件中，通 参数 rules_path 来指定位置。

原形 干提取

了完整地 完成本章的内容，我 将 解如何将已提取 干的 和原 索引到同一个字段中。个例子，分析句子 *The quick foxes jumped* 将会得到以下 ：

```
Pos 1: (the)
Pos 2: (quick)
Pos 3: (foxes,fox) ①
Pos 4: (jumped,jump) ①
```

① 已提取　干的形式和未提取　干的形式位于相同的位置。

Warning：使用此方法前　先　　原形　干提取是个好主意　。

了　　干提取出的 原形 ，我　将使用 keyword_repeat　　器，跟 keyword_marker　　器（see 阻止干提取　　）　　一　，它把　一个　　都　　，以防止后　　干提取器其修改。但是，它依然会在相同位置上重　　，并且　个重　的　　是 提取的　干。

独使用 keyword_repeat token　　器将得到以下　果：

```
Pos 1: (the,the) ①
Pos 2: (quick,quick) ①
Pos 3: (foxes,fox)
Pos 4: (jumped,jump)
```

① 提取　干前后的形式一　，所以只是不必要的重　。

了防止提取和未提取　干形式相同的　　中的无意　重　，我　　加了　合的　　{ref}/analysis-unique-tokenfilter.html[unique]　　元　　器：

```
PUT /my_index
{
  "settings": {
    "analysis": {
      "filter": {
        "unique_stem": {
          "type": "unique",
          "only_on_same_position": true ①
        }
      },
      "analyzer": {
        "in_situ": {
          "tokenizer": "standard",
          "filter": [
            "lowercase",
            "keyword_repeat", ②
            "porter_stem",
            "unique_stem" ③
          ]
        }
      }
    }
  }
}
```

①　置 unique　型　　元　　器，是　了只有当重　　元出　在相同位置　，移除它　。

②　　元　　器必　出　在　干提取器之前。

③ unique_stem　　器是在　干提取器完成之后移除重　　。

原形 干提取是个好主意

用 喜 原形 干提取 个主意：``如果我可以只用一个 合字段， 什 要分 存一个未提取 干和已提取 干的字段 ？"但 是一个好主意 ？答案一直都是否定的。因 有 个 ：

第一个 是无法区分精准匹配和非精准匹配。本章中，我 看到了多 常会被展 成相同的 干 ：organs 和 organization 都会被提取 organ 。

在 使用 言分析器 我 展示了如何整合一个已提取 干属性的 （ 了 加召回率)和一个未提取 干属性的 （ 了提升相 度）。 当提取和未提取 干的属性相互独立 ， 个属性的 献可以通 其中一个属性 加boost 来 化(参 [prioritising-clauses])。相反地，如果已提取和未提取 干的形式置于同一个属性，就没有 法来 化搜索 果了。

第二个 是，必 清楚 相 度分 是否如何 算的。在 [relevance-intro] 我 解 了部分 算依 于逆文 率（IDF）—— 即一个 在索引 的所有文 中出 的 繁程度。 在一个包含文本 jump jumped jumps 的文 上使用原形 干提取，将得到下列 ：

```
Pos 1: (jump)
Pos 2: (jumped,jump)
Pos 3: (jumps,jump)
```

jumped 和 jumps 各出 一次，所以有正 的IDF ；jump 出 了3次，作 一个搜索 ，与其他未提取 干的形式相比， 明 降低了它的IDF 。

基于 些原因，我 不推 使用原形 干提取。

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 将 原 根 ） 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。

同

干提取是通 化他 的 根形式来 大搜索的 ，同 通 相 的 念和概念来 大搜索 。也 没有文 匹配 “英国女王“，但是包含 “英国君主” 的文 可能会被 是很好的匹配。

用 搜索 “美国” 并且期望 到包含 美利 合 国 、 美国 、 美洲 、或者 美国各州 的文 。 然而，他 不希望搜索到 于 国事 或者 政府机 的 果。

个例子提供了宝 的 ，它向我 述了，区分不同的概念 于人 是多 而 于 粹的机器是多棘手的事情。通常我 会 言中的 一个 去 提供同 以 保任何一个文 都是可 的，以保不管文 之 有多 微小的 性都能 被 索出来。

做是不 的。就像我 更喜 不用或少用 根而不是 分使用 根一 ，同 也 只在必要的 候使用。 是因 用 可以理解他 的搜索 果受限于他 的搜索 ，如果搜索 果看上去几乎是随机 ，他 就会 得无法理解（注：大 模使用同 会 致 果 向于 人 得是随机的）。

同　　可以用来合并几乎相同含　的　，如 跳 、 跳越 或者　 脚跳行 ，和 小　子 、　　或者　 料手　 。或者，它　可以用来　一个　　得更通用。例如，　　可以作　 猫　　或　 子 的通用代名　，　有， 成人 可以被用于 男人 或者 女人 。

同　　似乎是一个　　的概念，但是正　的使用它　却是非常困　的。在　一章，我　会介　使用同　　的技巧和　　它的局限性和陷　。

TIP

同　　大了一个匹配文件的　　。正如　 干提取 或者 部分匹配 ，同　　的字段不　　被　独使用，而　　与一个　　主字段的　　操作一起使用，　个主字段　　包含　　格式的原始文本。 在使用同　　，参　 [most-fields] 的解　来　　相　性。

使用同

同　　可以取代　有的　　元或　通　使用　{ref}/analysis-synonym-tokenfilter.html[同元　　器]，添加到　　元流中：

```
PUT /my_index
{
  "settings": {
    "analysis": {
      "filter": {
        "my_synonym_filter": {
          "type": "synonym", ①
          "synonyms": [ ②
            "british,english",
            "queen,monarch"
          ]
        }
      },
      "analyzer": {
        "my_synonyms": {
          "tokenizer": "standard",
          "filter": [
            "lowercase",
            "my_synonym_filter" ③
          ]
        }
      }
    }
  }
}
```

① 首先，我　定　了一个 同　　型的　　元　　器。

② 我　在 同　　格式 中　　同　　格式。

③ 然后我　　建了一个使用 `my_synonym_filter` 的自定　分析器。

TIP

同　可以使用　synonym　参数来内嵌指定，或者必　存在于集群　一个　点上的同　文件中。同　文件路径由 synonyms_path 参数指定，　或相　于 Elasticsearch config　目　。参照　更新停用　（Updating　Stopwords）　的技巧，可以用来刷新的同　列表。

通　analyze API 来　我　的分析器，　示如下：

```
GET /my_index/_analyze?analyzer=my_synonyms
Elizabeth is the English queen
```

```
Pos 1: (elizabeth)
Pos 2: (is)
Pos 3: (the)
Pos 4: (british,english) ①
Pos 5: (queen,monarch) ①
```

① 所有同　与原始　占有同一个位置。

　的一个文件将匹配任何以下的　：English queen、British queen、English monarch 或 British monarch。即使是一个短　也将会工作，因　个　的位置已被保存。

TIP

在索引和搜索中使用相同的同　元　器是多余的。　如果在索引的　候，我　用 english 和 british　个　代替 English，然后在搜索的　候，我　只需要搜索　些　中的一个。或者，如果在索引的　候我　不使用同　，然后在搜索的　候，我　将需要把　English 的　english 或者 british 的　。

是否在搜索或索引的　候做同　展可能是一个困　的　。我　将探索更多的　展或收　。

同　格式

同　最　的表　形式是 逗号分隔：

```
"jump,leap,hop"
```

如果遇到　些　中的任何一　，　将其替　所有列出的同　。例如：

```
原始　:　取代:
————————————————————————————————————————————————
jump            → (jump,leap,hop)
leap            → (jump,leap,hop)
hop             → (jump,leap,hop)
```

或者, 使用 ⇒　法，可以指定一个　列表（在左　），和一个或多个替　（右　）的列表：

```
"u s a,united states,united states of america => usa"
"g b,gb,great britain => britain,england,scotland,wales"
```

```
原始    :   取代:
————————————————————————————————————————————————————————

u s a           → (usa)
united states   → (usa)
great britain   → (britain,england,scotland,wales)
```

如果多个 指定同一个同 ，它 将被合并在一起，且 序无 ，否 使用最 匹配。以下面的例：

```
"united states            => usa",
"united states of america => usa"
```

如果 些 相互冲突，Elasticsearch 会将 United States of America (usa),(of),(america) 。否 ，会使用最 的序列，即最 得到 (usa) 。

展或收

在 同 格式 中，我 看到了可以通 展 、 收 、或_ 型 展_ 来指明同 。 本章 我 将在 三者 做个 衡比 。

TIP 本 理 同 。多 同 又 添了一 性，在 多 同 和短 中，我 将会 。

展

通 展，我 可以把同 列表中的任意一个 展成同 列表 所有 的 ：

```
"jump,hop,leap"
```

展可以 用在索引 段或 段。 者都有 点 ( ) 和 点 ( ) 。到底要在 个 段使用，取决于性能与 活性：

	索引	
索引的大小	大索引。因 所有的同 都会被索引，所以索引的大小相 会 大一些。	正常大小。

	索引	
	所有同 都有相同的 IDF（至于什 是 IDF，参 [relevance-intro]），意味着通用的 和 常用的 都有着相同的 重。	个同 IDF 都和原来一 。
性能	只需要 到 字符串中指定 个 。	一个 的 重写来 所有的同 ，从而降低性能。
活性	同 不能改 有的文件。 于有影 的新 ， 有的文件都要重建（注：重新索引一次文 ）。	同 可以更新不需要索引文件。

收

收 ，把 左 的多个同 映射到了右 的 个 ：

```
"leap,hop => jump"
```

它必 同 用于索引和 段，以 保 映射到索引中存在的同一个 。

相 于 展方法， 方法也有一些 点和一些 点：

索引的大小

索引大小是正常的，因 只有 一 被索引。

所有 的 IDF 是一 的，所以 不能区分比 常用的 、不常用的 。

性能

只需要在索引中 到 的出 。

活性

新同 可以添加到 的左 并在 段使用。例如，我 想添加 bound 到先前指定的同 中。那 下面的 将作用于包含 bound 的 或包含 bound 的文 索引：

```
"leap,hop,bound => jump"
```

似乎 旧有的文 不起作用是 ？其 我 可以把上面 个同 改写下，以便 旧有文 同 起作用：

```
"leap,hop,bound => jump,bound"
```

当 重建索引文件， 可以恢 到上面的 （注： `leap,hop,bound ⇒ jump` ）来 得 个 的性能 （注：因 上面那个 相比 个而言， 段就只要 一个 了）。

型　展

型　展是完全不同于　　收　　　　或　　，　　　　并不是平等看待所有的同　　，而是　大了　的意　，使被拓展的　更　通用。以　些　　例：

```
"cat    => cat,pet",
"kitten => kitten,cat,pet",
"dog    => dog,pet"
"puppy  => puppy,dog,pet"
```

通　在索引　段使用　型　展：

- 一个　于 kitten 的　　会　　于 kittens 的文　。
- 　　一个 cat 会　到　于 kittens 和 cats 的文　。
- 一个 pet 的　　将　　有　的 kittens、cats、puppies、dogs 或者 pets 的文　。

或者在　　段使用　型　展，kitten 的　　果就会被拓展成　及到 kittens、cats、dogs。

　也可以有　全其美的　法，通　在索引　段　用　型　展同　　，以　保　型在索引中存在。然后，在　　段，　　可以　　不采用同　　（使 kitten　　只返回 kittens 的文件）或采用同　　，kitten 的　　操作就会返回包括 kittens、cats、pets（也包括 dogs 和 puppies）的相　　果。

前面的示例　　，　　kitten 的 IDF 将是正　的，而 cat 和 pet 的 IDF 将会被 Elasticsearch 降　。然而，　是　　有利的，当一个　　kitten 的　　被拓展成了　　kitten OR cat OR pet 的　　，那　kitten 相　的文　就　　排在最上方，其次是 cat 的文件，pet 的文件将被排在最底部。

同　　和分析

在 同　　格式 一章中，我　使用 u s a 来　例　述一些同　　相　的知　。那　　什　　我　使用的不是 U.S.A.　？原因是，　　个 同　　的　　元　　器只能接收到在它前面的　　元　　器或者分　器的　出　果（　里看不到原始文本）。

假　我　有一个分析器，它由 standard　分　器、　lowercase　的　　元　　器、　synonym　的　　元　　器　成。文本 U.S.A. 的分析　程，看起来像　　的：

```
original string（原始文本）                    → "U.S.A."
standard          tokenizer（分　器）          → (U),(S),(A)
lowercase         token filter（　　元　　器）  → (u),(s),(a)
synonym           token filter（　　元　　器）  → (usa)
```

如果我　有指定的同　　U.S.A.　，它永　不会匹配任何　西。因　，　my_synonym_filter　看到　　的　候，句号已　被移除了，并且字母已　被小写了。

　其　是一个非常需要注意的地方。如果我　想同　使用同　　特性与　根提取特性，那　jumps 、 jumped 、 jump 、 leaps 、 leaped 和 leap　　些　是否都会被索引成一个 jump ？我　　可以把同　器放置在　根提取之前，然后把所有同　　以及　形　化都列　出来：

```
"jumps,jumped,leap,leaps,leaped => jump"
```

但更　的方式将同　　器放置在　根　器之后，然后把　根形式的同　列　出来：

```
"leap => jump"
```

大小写敏感的同

通常，我　把同　　器放置在　lowercase　　元　器之后，因此，所有的同　　都是小写。但有　会　致奇怪的合并。例如，CAT　描和一只 cat 有很大的不同，或者 PET （正　子　射断　描）和 pet 。就此而言，姓 Little 也是不同于形容　little 的 (尽管当一个句子以它　　，首字母会被大写)。

如果根据使用情况来区分　，　需要将同　　器放置在　lowercase　　器之前。当然，意味着同　　需要列出所有想匹配的　化（例如，Little、LITTLE、little ）。

相反,可以有　个同　　器：一个匹配大小写敏感的同　，一个匹配大小写不敏感的同　。例如，大小写敏感的同　　可以是　个　子：

```
"CAT,CAT scan           => cat_scan"
"PET,PET scan           => pet_scan"
"Johnny Little,J Little => johnny_little"
"Johnny Small,J Small   => johnny_small"
```

大小不敏感的同　　可以是　个　子：

```
"cat                    => cat,pet"
"dog                    => dog,pet"
"cat scan,cat_scan scan => cat_scan"
"pet scan,pet_scan scan => pet_scan"
"little,small"
```

大小写敏感的同　　不　会　理　CAT　scan　，而且有　候也可能会匹配到　CAT　scan　中的　CAT（注：从而　致　CAT　scan　被　化成了同　　cat_scan　scan　）。出于　个原因，在大小写敏感的同　列表中会有一个　　坏替　情况的特　　cat_scan scan 。

提示：　可以看到它　可以多　易地　得　。同平　一　，　analyze　API　是　手，用它来　分析器是否正　配置。参　[analyze-api]。

多　同　　和短

至此，同　看上去　挺　的。然而不幸的是，　的部分才　　始。　了能使 短　　正常工作，Elasticsearch　　需要知道　个　在初始文本中的位置。多　同　会　重破坏　的位置信息，尤其当新　的同　　度各不相同的　候。

我　建一个同　　元　器，然后使用下面　的同　　：

```
"usa,united states,u s a,united states of america"
```

```
PUT /my_index
{
  "settings": {
    "analysis": {
      "filter": {
        "my_synonym_filter": {
          "type": "synonym",
          "synonyms": [
            "usa,united states,u s a,united states of america"
          ]
        }
      },
      "analyzer": {
        "my_synonyms": {
          "tokenizer": "standard",
          "filter": [
            "lowercase",
            "my_synonym_filter"
          ]
        }
      }
    }
  }
}

GET /my_index/_analyze?analyzer=my_synonyms&text=
The United States is wealthy
```

解析器 会 出下面 的 果：

```
Pos 1:  (the)
Pos 2:  (usa,united,u,united)
Pos 3:  (states,s,states)
Pos 4:  (is,a,of)
Pos 5:  (wealthy,america)
```

如果 用上面 个同 元 器索引一个文 ，然后 行一个短 ，那 就会得到 人的 果 ，下面 些短 都不会匹配成功：

- The usa is wealthy
- The united states of america is wealthy
- The U.S.A. is wealthy

但是 些短 会：

- United states is wealthy
- Usa states of wealthy
- The U.S. of wealthy
- U.S. is america

如果 是在 段使同 ，那 就会看到更加 的匹配 果。看下 个 validate-query ：

```
GET /my_index/_validate/query?explain
{
  "query": {
    "match_phrase": {
      "text": {
        "query": "usa is wealthy",
        "analyzer": "my_synonyms"
      }
    }
  }
}
```

字会被同 元 器 理成 似 的信息：

```
"(usa united u united) (is states s states) (wealthy a of) america"
```

会匹配包含有 u is of america 的文 ，但是匹配不出任何含有 america 的文 。

TIP 多 同 高亮匹配 果也会造成影 。一个 USA 的 ，返回的 果可能却高亮了：The *United States is wealthy* 。

使用 收 行短

避免 混乱的方法是使用 收 ， 用 个 表示所有的同 ， 然后在段，就只需要 个 行 了：

```
PUT /my_index
{
  "settings": {
    "analysis": {
      "filter": {
        "my_synonym_filter": {
          "type": "synonym",
          "synonyms": [
            "united states,u s a,united states of america=>usa"
          ]
        }
      },
      "analyzer": {
        "my_synonyms": {
          "tokenizer": "standard",
          "filter": [
            "lowercase",
            "my_synonym_filter"
          ]
        }
      }
    }
  }
}

GET /my_index/_analyze?analyzer=my_synonyms
The United States is wealthy
```

上面那个　信息就会被　理成　似下面　：

```
Pos 1:  (the)
Pos 2:  (usa)
Pos 3:  (is)
Pos 5:  (wealthy)
```

在我　再次　行我　之前做　的那个 validate-query　，就会　出一个　又合理的　果：

```
"usa is wealthy"
```

个方法的　点是，因　把 united states of america　成了同　usa，　就不能使用 united states of america 去搜索出 united 或者 states 。　需要使用一个　外的字段并用　一个解析器　来　到个目的。

同　与 query_string

本　很少　到 query_string　，因　真心不推　用它。　在　一　中有提到，由于 query_string　支持一个精　的　法　，因此，可能　会　致它搜出一些出人意料的

果或者甚至是含有 法 的 果。

方式存在不少 ，而其中之一便与多 同 有 。 了支持它的 法， 必 用指定的、法所能 的操作符号来 示，比如 AND 、 OR 、 + 、 - 、 field: 等等。 (更多相 内容参 {ref}/query-dsl-query-string-query.html#query-string-syntax[query_string 法] 。)

而在 法的解析 程中，解析 作会把 文本在空格符 作切分，然后分 把 个切分出来的相 性解析器。 也即意味着 的同 解析器永 都不可能收到 似 United States 的多个 成的同 。由于不会把 United States 作 一个原子性的文本，所以同 解析器的 入信息永都是 个被切分 的 United 和 States 。

所幸， match 相比而言就可 得多了，因 它不支持上述 法，所以多个字 成的同 不会被切分，而是会完整地交 解析器 理。

符号同

最后一 内容我 来 述下 符号 行同 理， 和我 前面 的同 理不太一 。符号同 是用 名来表示 个符号，以防止它在分 程中被 是不重要的 点符号而被移除。

然 大多数情况下，符号 于全文搜索而言都无 要，但是字符 合而成的表情，或 又会是很有意的 西，甚至有 候会改 整个句子的含 ， 比一下 句 ：

- 我很高 能在星期天工作。
- 我很高 能在星期天工作 :(（注： 的表情）

准 （注：standard）分 器或 会 地消除掉第二个句子里的字符表情，致使 个原本意思相去甚的句子 得相同。

我 可以先使用 {ref}/analysis-mapping-charfilter.html[映射]字符 器，在文本被 交 分 器 理之前， 把字符表情替 成符号同 emoticon_happy 或者 emoticon_sad ：

```
PUT /my_index
{
  "settings": {
    "analysis": {
      "char_filter": {
        "emoticons": {
          "type": "mapping",
          "mappings": [ ①
            ":)=>emoticon_happy",
            ":(=>emoticon_sad"
          ]
        }
      },
      "analyzer": {
        "my_emoticons": {
          "char_filter": "emoticons",
          "tokenizer":   "standard",
          "filter":    [ "lowercase" ]
          ]
        }
      }
    }
  }
}

GET /my_index/_analyze?analyzer=my_emoticons
I am :) not :( ②
```

① 映射　　器把字符从 ⇒ 左　的格式　　成右　的　子。

②　出：i、am、emoticon_happy、not、emoticon_sad。

很少有人会搜　　emoticon_happy　　个　，但是　保　似字符表情的　　重要符号被存　到索引中是非常好的做法，在　行情感分析的　候会很有用。当然，我　也可以用真　的　　来　理符号同　　，比如：happy 或者 sad。

提示：　　映射　　字符　　器是个非常有用的　　器，它可以用来　一些已有的字　　行替　操作，　如果想要采用更　活的正　表　式去替　字　的　，那　可以使用　　{ref}/analysis-pattern-replace-charfilter.html[pattern_replace]字符　　器。

写

我　期望在　似　　和　格的　　化数据上　行一个　　来返回精　匹配的文　。　　然而，好的全文索不　　是完全相同的限定　　。　　相反，我　可以　大　　以包括　　可能　　的匹配，而根据相　性得分将更好的匹配推到　果集的　部。

事　上，只能完全匹配的全文搜索可能会困　　的用　。　　道不希望在搜索　quick　brown　fox　匹配一个包含 fast brown foxes 的文　，　搜索 Johnny Walker 同　匹配 Johnnie Walker，搜索 Arnold Shcwarzenneger 同　匹配 Arnold Schwarzenegger？

如果存在完全符合用　　的文　，他　　出　在　果集的　部，而　弱的匹配可以被包含在列表的后面。　　如果没有精　匹配的文　，至少我　可以　示有可能匹配用　要求的文　，它　甚至可能是用　最初想要的！

我　已　在　一化　元 看　自由　音匹配，　将　　原　　根 中的　干，　同　　中的同　　，　但所有　些方法假定　　写正　，或者　个　　写只有唯一的方法。

Fuzzy matching 允　　匹配　　写的　　，而　音　　元　　器可以在索引　用来　行 近似　音匹配。

模糊性

模糊匹配　待 “模糊” 相似的　个　似乎是同一个　。首先，我　需要　我　所　的 模糊性　行定　。

在1965年，Vladimir　Levenshtein　　出了　Levenshtein　distance，　用来度量从一个　　到一个　　需要多少次　字符　　。他提出了三　　型的　字符　　：

- 一个字符 替　　一个字符：_f_ox → _b_ox
- 　入 一个新的字符：sic → sic_k_
- 　除 一个字符：b_l_ack → back

Frederick Damerau 后来在　些操作基　上做了一个　展：

- 相　　个字符的　位　：_st_ar → _ts_ar

个例子，将　　`bieber`　　成 `beaver` 需要下面几个　　：

1. 把 `b` 替　成 `v` ：bie_b_er → bie_v_er
2. 把 `i` 替　成 `a` ：b_i_ever → b_a_ ever
3. 把 `e` 和 `a`　行　位：b_ae_ver → b_ea_ver

三个　　表示 Damerau-Levenshtein edit distance　　距　　3 。

然，从 <code>beaver</code>　　成 <code>bieber</code> 是一个很　的　程—他　相距甚而不能　　一个　　的　写　　。 Damerau　　80% 的　写　　距　　1 。　句　　，　80% 的　写　　可以　原始字符串用 　次　　　行修正。

Elasticsearch 指定了 `fuzziness` 参数支持　最大　　距　的配置，　　2 。

当然，　次　　字符串的影　取决于字符串的　度。　　`hat`　次　　能　　生 `mad`，所以　一个只有 3　个字符　度的字符串允　　次　　然太多了。　`fuzziness`　参数可以被　置　　`AUTO`　，　将致以下的最大　　距　：

- 字符串只有 1 到 2 个字符　是 `0`
- 字符串有 3 、 4 或者 5 个字符　是 `1`
- 字符串大于 5 个字符　是 `2`

当然，　可能会　　距　`2`　然是太多了，返回的　果似乎并不相　。 把最大 `fuzziness`　置　`1`，可以得到更好的　果和更好的性能。

模糊

{ref}/query-dsl-fuzzy-query.html[fuzzy]是 term 的模糊等 。 也 很少直接使用它，但是理解它是如何工作的，可以 助 在更高 的 match 中使用模糊性。

了解它是如何 作的，我 首先索引一些文 ：

```
POST /my_index/my_type/_bulk
{ "index": { "_id": 1 }}
{ "text": "Surprise me!"}
{ "index": { "_id": 2 }}
{ "text": "That was surprising."}
{ "index": { "_id": 3 }}
{ "text": "I wasn't surprised."}
```

在我 可以 surprize 行一个 fuzzy ：

```
GET /my_index/my_type/_search
{
  "query": {
    "fuzzy": {
      "text": "surprize"
    }
  }
}
```

fuzzy 是一个 的 ，所以它不做任何分析。它通 某个 以及指定的 fuzziness 到 典中所有的 。 fuzziness 置 AUTO 。

在我 的例子中， surprise 比 surprise 和 surprised 都在 距 2 以内， 所以文 1 和 3 匹配。通 以下 ，我 可以 少匹配度到 匹配 surprise ：

```
GET /my_index/my_type/_search
{
  "query": {
    "fuzzy": {
      "text": {
        "value": "surprize",
        "fuzziness": 1
      }
    }
  }
}
```

提高性能

<code>fuzzy</code> 的工作原理是 定原始 及 造一个 自 机—

像表示所有原始字符串指定　　距　的字符串的一个大　表。

然后模糊　　使用　个自　机依次高效遍　　典中的所有　　以　定是否匹配。　　一旦收集了　典中存在的所有匹配　，就可以　算匹配文　列表。

当然，根据存　在索引中的数据　型，一个　　距　　2　　的模糊　　能　匹配一个非常大数量的　　同　　行效率会非常糟　。下面　个参数可以用来限制　性能的影　：

`prefix_length`
　不能被`模糊化''　的初始字符数。　大部分的　写　　生在　的　尾，而不是　的　始。　例如通　将`prefix_length`　置　3，　可能　　著降低匹配的　　数量。

`max_expansions`
　如果一个模糊　　展了三个或四个模糊　　，　　些新的模糊　　也　是有意　的。如　果它　生　1000　个模糊　　，那　就基本没有意　了。　　置　　`max_expansions`　　用来限制将　生的模糊　　的　数量。模糊　　将收集匹配　　直到　到`max_expansions`的限制。

模糊匹配

`match`　　支持　箱即用的模糊匹配：

```
GET /my_index/my_type/_search
{
  "query": {
    "match": {
      "text": {
        "query":     "SURPRIZE ME!",
        "fuzziness": "AUTO",
        "operator":  "and"
      }
    }
  }
}
```

　字符串首先　行分析，会　生　　`[surprize,　　me]`　，并且　个　　根据指定的　`fuzziness`　行模糊化。

同　，`multi_match`　　也支持`fuzziness`，但只有当　行　　型是`best_fields`或者`most_fields`：

```
GET /my_index/my_type/_search
{
  "query": {
    "multi_match": {
      "fields":  [ "text", "title" ],
      "query":     "SURPRIZE ME!",
      "fuzziness": "AUTO"
    }
  }
}
```

match 和 multi_match 都支持 prefix_length 和 max_expansions 参数。

TIP 模糊性（Fuzziness）只能在 match and multi_match 中使用。不能使用在短匹配、常用 或 cross_fields 匹配。

模糊性 分

用 喜 模糊 。他 会魔法般的 到正 写 合。很 憾， 效果平平。

假 我 有1000个文 包含 Schwarzenegger，只是一个文 的出 写 Schwarzeneger 。根据 term frequency/inverse document frequency 理 ， 个 写 文 比 写正 的相 度更高，因 写出 在更少的文 中！

句 ，如果我 待模糊匹配 似其他匹配方法，我 将偏 的 写超 了正 的 写， 会 用 狂。

TIP 模糊匹配不 用于参与 分—只能在有 写 大匹配 的 。

情况下， match 定所有的模糊匹配的恒定 分 1。 可以 足在果列表的末尾添加潜在的匹配 ，并且没有干 非模糊 的相 性 分。

TIP 在模糊 最初出 很少能 独使用。他 更好的作 一个 bigger 景的部分功能特性，如 *search-as-you-type* {ref}/search-suggesters-completion.html[完成 建]或 *did-you-mean* {ref}/search-suggesters-phrase.html[短 建]。

音匹配

最后，在 任何其他匹配方法都无效后，我 可以求助于搜索 音相似的 ，即使他 的 写不同。

有一些用于将 成 音 的算法。 Soundex 算法是 些算法的鼻祖， 而且大多数 音算法是 Soundex 的改 或者 版本，例如 Metaphone 和 Double Metaphone （ 展了除英 以外的其他言的 音匹配）， Caverphone 算法匹配了新西 的名称， Beider-Morse 算法吸收了 Soundex 算法了更好的匹配 和依地 名称， Kölner Phonetik 了更好的 理 。

得一提的是， 音算法是相当 的，他 初衷 的 言通常是英 或 。 限制了他 的 用

性。不 ， 了某些明 的目 ，并与其他技 相 合， 音匹配能 作 一个有用的工具。

首先， 需要从 https://www.elastic.co/guide/en/elasticsearch/plugins/current/analysis-phonetic.html 取 音分析 件并在集群的 个 点安装， 然后重 个 点。

然后， 可以 建一个使用 音 元 器的自定 分析器，并 下面的方法：

```
PUT /my_index
{
  "settings": {
    "analysis": {
      "filter": {
        "dbl_metaphone": { ①
          "type":    "phonetic",
          "encoder": "double_metaphone"
        }
      },
      "analyzer": {
        "dbl_metaphone": {
          "tokenizer": "standard",
          "filter":    "dbl_metaphone" ②
        }
      }
    }
  }
}
```

① 首先，配置一个自定 phonetic 元 器并使用 double_metaphone 器。

② 然后在自定 分析器中使用自定 元 器。

在我 可以通 analyze API 来 行 ：

```
GET /my_index/_analyze?analyzer=dbl_metaphone
Smith Smythe
```

个 Smith 和 Smythe 在同一位置 生 个 元： SM0 和 XMT 。 通 分析器播放 John， Jon 和 Johnnie 将 生 个 元 JN 和 AN，而 Jonathon 生 元 JN0N 和 ANTN 。

音分析器可以像任何其他分析器一 使用。首先映射一个字段来使用它，然后索引一些数据：

```
PUT /my_index/_mapping/my_type
{
  "properties": {
    "name": {
      "type": "string",
      "fields": {
        "phonetic": { ①
          "type":     "string",
          "analyzer": "dbl_metaphone"
        }
      }
    }
  }
}

PUT /my_index/my_type/1
{
  "name": "John Smith"
}

PUT /my_index/my_type/2
{
  "name": "Jonnie Smythe"
}
```

① `name.phonetic` 字段使用自定 `dbl_metaphone` 分析器。

可以使用 `match` 来 行搜索：

```
GET /my_index/my_type/_search
{
  "query": {
    "match": {
      "name.phonetic": {
        "query": "Jahnnie Smeeth",
        "operator": "and"
      }
    }
  }
}
```

个 返回全部 个文 ，演示了如何 行 的 音匹配。 用 音算法 算 分是没有 的。音匹配的目的不是 了提高精度，而是要提高召回率—以 展足 的 来捕 可能匹配的文 。

通常更有意 的使用 音算法是在 索到 果后，由 一台 算机 行消 和后 理，而不是由人 用直接使用。