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　—　<a　anchor="algorithmic-

stemmers"> 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 [configuring-language-analyzers]）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 [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 [using-language-analyzers] 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。