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 。