|  |
| --- |
| Elasticsearch 权威指南 |
| **2017/11/14** |
|  |
|  |

目 录

[第1章 基础入门 1](#_Toc498415171)

[1.1 本书协议约定 1](#_Toc498415172)

[1.2 你知道的, 为了搜索… 1](#_Toc498415173)

[1.2.1 安装并运行 Elasticsearch 1](#_Toc498415174)

[1.2.2 和 Elasticsearch 交互 3](#_Toc498415175)

[1.2.3 面向文档 6](#_Toc498415176)

[1.2.4 适应新环境 7](#_Toc498415177)

[1.2.5 索引雇员文档 7](#_Toc498415178)

[1.2.6 检索文档 9](#_Toc498415179)

[1.2.7 轻量搜索 10](#_Toc498415180)

[1.2.8 使用查询表达式搜索 12](#_Toc498415181)

[1.2.9 更复杂的搜索 13](#_Toc498415182)

[1.2.10 全文搜索 14](#_Toc498415183)

[1.2.11 短语搜索 15](#_Toc498415184)

[1.2.12 高亮搜索 16](#_Toc498415185)

[1.2.13 分析 18](#_Toc498415186)

[1.2.14 教程结语 20](#_Toc498415187)

[1.2.15 分布式特性 21](#_Toc498415188)

[1.2.16 后续步骤 21](#_Toc498415189)

[1.3 集群内的原理 22](#_Toc498415190)

[1.3.1 补充章节 22](#_Toc498415191)

[1.3.2 空集群 22](#_Toc498415192)

[1.3.3 集群健康 23](#_Toc498415193)

[1.3.4 添加索引 24](#_Toc498415194)

[1.3.5 添加故障转移 26](#_Toc498415195)

[1.3.6 水平扩容 27](#_Toc498415196)

[1.3.7 应对故障 29](#_Toc498415197)

[1.4 数据输入和输出 30](#_Toc498415198)

[1.4.1 什么是文档? 31](#_Toc498415199)

[1.4.2 文档元数据 32](#_Toc498415200)

[1.4.3 索引文档 33](#_Toc498415201)

[1.4.4 取回一个文档 35](#_Toc498415202)

[1.4.5 检查文档是否存在 37](#_Toc498415203)

[1.4.6 更新整个文档 37](#_Toc498415204)

[1.4.7 创建新文档 38](#_Toc498415205)

[1.4.8 删除文档 39](#_Toc498415206)

[1.4.9 处理冲突 40](#_Toc498415207)

[1.4.10 乐观并发控制 41](#_Toc498415208)

[1.4.11 文档的部分更新 45](#_Toc498415209)

[1.4.12 取回多个文档 49](#_Toc498415210)

[1.4.13 代价较小的批量操作 51](#_Toc498415211)

[1.5 分布式文档存储 56](#_Toc498415212)

[1.5.1 路由一个文档到一个分片中 56](#_Toc498415213)

[1.5.2 主分片和副本分片如何交互 56](#_Toc498415214)

[1.5.3 新建、索引和删除文档 57](#_Toc498415215)

[1.5.4 取回一个文档 59](#_Toc498415216)

[1.5.5 局部更新文档 60](#_Toc498415217)

[1.5.6 多文档模式 61](#_Toc498415218)

[1.6 搜索——最基本的工具 64](#_Toc498415219)

[1.6.1 空搜索 65](#_Toc498415220)

[1.6.2 多索引，多类型 66](#_Toc498415221)

[1.6.3 分页 67](#_Toc498415222)

[1.6.4 *轻量* 搜索 68](#_Toc498415223)

[1.7 映射和分析 70](#_Toc498415224)

[1.7.1 精确值 VS 全文 71](#_Toc498415225)

[1.7.2 倒排索引 72](#_Toc498415226)

[1.7.3 分析与分析器 74](#_Toc498415227)

[1.7.4 映射 78](#_Toc498415228)

[1.7.5 复杂核心域类型 83](#_Toc498415229)

[1.8 请求体查询 86](#_Toc498415230)

[1.8.1 空查询 86](#_Toc498415231)

[1.9 查询表达式 87](#_Toc498415232)

[1.9.1 查询语句的结构 88](#_Toc498415233)

[1.9.2 合并查询语句 89](#_Toc498415234)

[1.9.3 查询与过滤 90](#_Toc498415235)

[1.9.4 性能差异 91](#_Toc498415236)

[1.9.5 如何选择查询与过滤 91](#_Toc498415237)

[1.9.6 最重要的查询 91](#_Toc498415238)

[1.9.7 组合多查询 93](#_Toc498415239)

[1.9.8 验证查询 96](#_Toc498415240)

[1.10 排序与相关性 98](#_Toc498415241)

[1.10.1 排序 98](#_Toc498415242)

[1.10.2 字符串排序与多字段 101](#_Toc498415243)

[1.10.3 什么是相关性? 103](#_Toc498415244)

[1.10.4 Doc Values 介绍 107](#_Toc498415245)

[1.11 执行分布式检索 108](#_Toc498415246)

[1.11.1 查询阶段 108](#_Toc498415247)

[1.11.2 取回阶段 110](#_Toc498415248)

[1.11.3 搜索选项 111](#_Toc498415249)

[1.11.4 游标查询 *Scroll* 113](#_Toc498415250)

[1.12 索引管理 115](#_Toc498415251)

[1.12.1 创建一个索引 115](#_Toc498415252)

[1.12.2 删除一个索引 115](#_Toc498415253)

[1.12.3 索引设置 116](#_Toc498415254)

[1.12.4 配置分析器 117](#_Toc498415255)

[1.12.5 自定义分析器 118](#_Toc498415256)

[1.12.6 类型和映射 121](#_Toc498415257)

[1.12.7 根对象 124](#_Toc498415258)

[1.12.8 动态映射 128](#_Toc498415259)

[1.12.9 自定义动态映射 129](#_Toc498415260)

[1.12.10 缺省映射 132](#_Toc498415261)

[1.12.11 重新索引你的数据 132](#_Toc498415262)

[1.12.12 索引别名和零停机 133](#_Toc498415263)

[1.13 分片内部原理 135](#_Toc498415264)

[1.13.1 使文本可被搜索 136](#_Toc498415265)

[1.13.2 不变性 136](#_Toc498415266)

[1.13.3 动态更新索引 137](#_Toc498415267)

[1.13.4 近实时搜索 140](#_Toc498415268)

[1.13.5 持久化变更 143](#_Toc498415269)

[1.13.6 段合并 147](#_Toc498415270)

[第2章 深入搜索 150](#_Toc498415271)

[2.1 结构化搜索 150](#_Toc498415272)

[2.1.1 精确值查找 151](#_Toc498415273)

[2.1.2 组合过滤器 156](#_Toc498415274)

[2.1.3 查找多个精确值 161](#_Toc498415275)

[2.1.4 范围 164](#_Toc498415276)

[2.1.5 处理 Null 值 167](#_Toc498415277)

[2.1.6 关于缓存 171](#_Toc498415278)

[2.2 全文搜索 173](#_Toc498415279)

[2.2.1 基于词项与基于全文 173](#_Toc498415280)

[2.2.2 匹配查询 175](#_Toc498415281)

[2.2.3 多词查询 177](#_Toc498415282)

[2.2.4 组合查询 180](#_Toc498415283)

[2.2.5 如何使用布尔匹配 182](#_Toc498415284)

[2.2.6 查询语句提升权重 183](#_Toc498415285)

[2.2.7 控制分析 185](#_Toc498415286)

[2.2.8 被破坏的相关度！ 188](#_Toc498415287)

[2.3 多字段搜索 189](#_Toc498415288)

[2.3.1 多字符串查询 189](#_Toc498415289)

[2.3.2 单字符串查询 191](#_Toc498415290)

[2.3.3 最佳字段 192](#_Toc498415291)

[2.3.4 最佳字段查询调优 195](#_Toc498415292)

[2.3.5 multi\_match 查询 197](#_Toc498415293)

[2.3.6 多数字段 199](#_Toc498415294)

[2.3.7 跨字段实体搜索 203](#_Toc498415295)

[2.3.8 字段中心式查询 204](#_Toc498415296)

[2.3.9 自定义 \_all 字段 206](#_Toc498415297)

[2.3.10 cross-fields 跨字段查询 208](#_Toc498415298)

[2.3.11 Exact-Value 精确值字段 211](#_Toc498415299)

[2.4 近似匹配 211](#_Toc498415300)

[2.4.1 短语匹配 212](#_Toc498415301)

[2.4.2 混合起来 214](#_Toc498415302)

[2.4.3 多值字段 215](#_Toc498415303)

[2.4.4 越近越好 217](#_Toc498415304)

[2.4.5 使用邻近度提高相关度 218](#_Toc498415305)

[2.4.6 性能优化 219](#_Toc498415306)

[2.4.7 寻找相关词 220](#_Toc498415307)

[2.5 部分匹配 226](#_Toc498415308)

[2.5.1 邮编与结构化数据 226](#_Toc498415309)

[2.5.2 prefix 前缀查询 227](#_Toc498415310)

[2.5.3 通配符与正则表达式查询 229](#_Toc498415311)

[2.5.4 查询时输入即搜索 230](#_Toc498415312)

[2.5.5 索引时优化 232](#_Toc498415313)

[2.5.6 Ngrams 在部分匹配的应用 232](#_Toc498415314)

[2.5.7 索引时输入即搜索 232](#_Toc498415315)

[2.5.8 Ngrams 在复合词的应用 239](#_Toc498415316)

[2.6 控制相关度 242](#_Toc498415317)

[2.6.1 相关度评分背后的理论 242](#_Toc498415318)

[2.6.2 Lucene 的实用评分函数 248](#_Toc498415319)

[2.6.3 查询时权重提升 252](#_Toc498415320)

[2.6.4 使用查询结构修改相关度 254](#_Toc498415321)

[2.6.5 Not Quite Not 255](#_Toc498415322)

[2.6.6 忽略 TF/IDF 257](#_Toc498415323)

[2.6.7 function\_score 查询 259](#_Toc498415324)

[2.6.8 按受欢迎度提升权重 260](#_Toc498415325)

[2.6.9 过滤集提升权重 268](#_Toc498415326)

[2.6.10 随机评分 270](#_Toc498415327)

[2.6.11 越近越好 271](#_Toc498415328)

[2.6.12 理解 price 价格语句 274](#_Toc498415329)

[2.6.13 脚本评分 274](#_Toc498415330)

[2.6.14 可插拔的相似度算法 276](#_Toc498415331)

[2.6.15 更改相似度 278](#_Toc498415332)

[2.6.16 调试相关度是最后 10% 要做的事情 280](#_Toc498415333)

[第3章 处理人类语言 280](#_Toc498415334)

[3.1 开始处理各种语言 281](#_Toc498415335)

[3.1.1 使用语言分析器 282](#_Toc498415336)

[3.1.2 配置语言分析器 284](#_Toc498415337)

[3.1.3 混合语言的陷阱 285](#_Toc498415338)

[3.1.4 每份文档一种语言 287](#_Toc498415339)

[3.1.5 每个域一种语言 289](#_Toc498415340)

[3.1.6 混合语言域 291](#_Toc498415341)

[3.2 词汇识别 294](#_Toc498415342)

[3.2.1 标准分析器 295](#_Toc498415343)

[3.2.2 标准分词器 295](#_Toc498415344)

[3.2.3 安装 ICU 插件 296](#_Toc498415345)

[3.2.4 icu\_分词器 297](#_Toc498415346)

[3.2.5 整理输入文本 298](#_Toc498415347)

[3.3 归一化词元 301](#_Toc498415348)

[3.3.1 举个例子 301](#_Toc498415349)

[3.3.2 如果有口音 302](#_Toc498415350)

[3.3.3 Unicode的世界 306](#_Toc498415351)

[3.3.4 Unicode 大小写折叠 307](#_Toc498415352)

[3.3.5 Unicode 字符折叠 309](#_Toc498415353)

[3.3.6 排序和整理 310](#_Toc498415354)

[3.4 将单词还原为词根 319](#_Toc498415355)

[3.4.1 词干提取算法 320](#_Toc498415356)

[3.4.2 字典词干提取器 323](#_Toc498415357)

[3.4.3 Hunspell 词干提取器 324](#_Toc498415358)

[3.4.4 选择一个词干提取器 328](#_Toc498415359)

[3.4.5 控制词干提取 330](#_Toc498415360)

[3.4.6 原形词干提取 333](#_Toc498415361)

[3.5 停用词: 性能与精度 335](#_Toc498415362)

[3.5.1 停用词的优缺点 336](#_Toc498415363)

[3.5.2 使用停用词 336](#_Toc498415364)

[3.5.3 停用词与性能 341](#_Toc498415365)

[3.5.4 词项的分别管理 342](#_Toc498415366)

[3.5.5 停用词与短语查询 346](#_Toc498415367)

[3.5.6 common\_grams 过滤器 348](#_Toc498415368)

[3.5.7 停用词与相关性 352](#_Toc498415369)

[3.6 同义词 352](#_Toc498415370)

[3.6.1 使用同义词 353](#_Toc498415371)

[3.6.2 同义词格式 354](#_Toc498415372)

[3.6.3 扩展或收缩 355](#_Toc498415373)

[3.6.4 同义词和分析链 358](#_Toc498415374)

[3.6.5 多词同义词和短语查询 359](#_Toc498415375)

[3.6.6 符号同义词 363](#_Toc498415376)

[3.7 拼写错误 364](#_Toc498415377)

[3.7.1 模糊性 364](#_Toc498415378)

[3.7.2 模糊查询 365](#_Toc498415379)

[3.7.3 模糊匹配查询 367](#_Toc498415380)

[3.7.4 模糊性评分 368](#_Toc498415381)

[3.7.5 语音匹配 368](#_Toc498415382)

[第4章 聚合 371](#_Toc498415383)

[4.1 高阶概念 372](#_Toc498415384)

[4.1.1 桶 372](#_Toc498415385)

[4.1.2 指标 373](#_Toc498415386)

[4.1.3 桶和指标的组合 373](#_Toc498415387)

[4.2 尝试聚合 373](#_Toc498415388)

[4.2.1 添加度量指标 376](#_Toc498415389)

[4.2.2 嵌套桶 378](#_Toc498415390)

[4.2.3 最后的修改 380](#_Toc498415391)

[4.1 条形图 382](#_Toc498415392)

[4.2 按时间统计 386](#_Toc498415393)

[4.2.1 返回空 Buckets 389](#_Toc498415394)

[4.2.2 扩展例子 391](#_Toc498415395)

[4.2.3 潜力无穷 394](#_Toc498415396)

[4.3 范围限定的聚合 395](#_Toc498415397)

[4.4 过滤和聚合 399](#_Toc498415398)

[4.4.1 过滤 399](#_Toc498415399)

[4.4.2 过滤桶 400](#_Toc498415400)

[4.4.3 后过滤器 401](#_Toc498415401)

[4.4.4 小结 402](#_Toc498415402)

[4.1 多桶排序 403](#_Toc498415403)

[4.1.1 内置排序 403](#_Toc498415404)

[4.1.2 按度量排序 404](#_Toc498415405)

[4.1.3 基于“深度”度量排序 405](#_Toc498415406)

[4.2 近似聚合 406](#_Toc498415407)

[4.2.1 统计去重后的数量 408](#_Toc498415408)

[4.2.2 百分位计算 412](#_Toc498415409)

[4.3 通过聚合发现异常指标 420](#_Toc498415410)

[4.3.1 significant\_terms 演示 421](#_Toc498415411)

[4.4 Doc Values and Fielddata编辑 431](#_Toc498415412)

[4.4.1 Doc Values 431](#_Toc498415413)

[4.4.2 深入文档值 433](#_Toc498415414)

[4.4.3 聚合与分析 436](#_Toc498415415)

[4.4.4 限制内存使用 441](#_Toc498415416)

[4.4.5 Fielddata 的过滤 445](#_Toc498415417)

[4.4.6 预加载 fielddata 447](#_Toc498415418)

[4.4.7 优化聚合查询 452](#_Toc498415419)

[第5章 管理、监控和部署 457](#_Toc498415420)

[5.1 监控 457](#_Toc498415421)

[5.1.1 Marvel 监控 458](#_Toc498415422)

[5.1.2 集群健康 458](#_Toc498415423)

[5.1.3 监控单个节点 462](#_Toc498415424)

[5.1.4 集群统计 471](#_Toc498415425)

[5.1.5 索引统计 471](#_Toc498415426)

[5.1.6 等待中的任务 472](#_Toc498415427)

[5.1.7 cat API 474](#_Toc498415428)

[5.2 部署 477](#_Toc498415429)

[5.2.1 硬件 478](#_Toc498415430)

[5.2.2 Java 虚拟机 479](#_Toc498415431)

[5.2.3 Transport Client 与 Node Client 480](#_Toc498415432)

[5.2.4 配置管理 480](#_Toc498415433)

[5.2.5 重要配置的修改 481](#_Toc498415434)

[5.2.6 不要触碰这些配置！ 484](#_Toc498415435)

[5.2.7 堆内存:大小和交换 486](#_Toc498415436)

[5.2.8 文件描述符和 MMap 489](#_Toc498415437)

[5.3 部署后 490](#_Toc498415438)

[5.3.1 动态变更设置 490](#_Toc498415439)

[5.3.2 日志记录 491](#_Toc498415440)

[5.3.3 索引性能技巧 492](#_Toc498415441)

[5.3.4 推迟分片分配 495](#_Toc498415442)

[5.3.5 滚动重启 496](#_Toc498415443)

[5.3.6 备份你的集群 497](#_Toc498415444)

[5.3.7 从快照恢复 503](#_Toc498415445)

[5.3.8 集群是活着的、呼吸着的生命 505](#_Toc498415446)

# [基础入门](https://www.elastic.co/guide/cn/elasticsearch/guide/current/getting-started.html)

**网址：**

https://www.elastic.co/guide/cn/elasticsearch/guide/current/foreword\_id.html

## 本书协议约定

以下是本书中使用的印刷规范：

斜体

表示重点、新的术语或概念。

等宽字体

用于程序列表以及在段落中引用变量或程序元素如：函数名称、数据库、数据类型、环境变量、语句和关键字。

提示

这个图标代表小贴士，建议。

注意

这个图标代表一般注意事项。

警告

这个图标代表警告。

## [你知道的, 为了搜索…](https://www.elastic.co/guide/cn/elasticsearch/guide/current/intro.html)

### [安装并运行 Elasticsearch](https://www.elastic.co/guide/cn/elasticsearch/guide/current/running-elasticsearch.html)

**安装并运行 Elasticsearch**

想用最简单的方式去理解 Elasticsearch 能为你做什么，那就是使用它了，让我们开始吧！

安装 Elasticsearch 之前，你需要先安装一个较新的版本的 Java，最好的选择是，你可以从 [www.java.com](http://www.java.com/)获得官方提供的最新版本的 Java。

之后，你可以从 elastic 的官网 [elastic.co/downloads/elasticsearch](https://www.elastic.co/downloads/elasticsearch) 获取最新版本的 Elasticsearch。

要想安装 Elasticsearch，先下载并解压适合你操作系统的 Elasticsearch 版本。如果你想了解更多的信息， 可以查看 Elasticsearch 参考手册里边的安装部分，这边给出的链接指向安装说明 [Installation](https://www.elastic.co/guide/en/elasticsearch/reference/master/_installation.html)。

提示

当你准备在生产环境安装 Elasticsearch 时，你可以在 [官网下载地址](http://www.elastic.co/downloads/elasticsearch) 找 到 Debian 或者 RPM 包，除此之外，你也可以使用官方支持的 [Puppet module](https://github.com/elasticsearch/puppet-elasticsearch) 或者 [Chef cookbook](https://github.com/elasticsearch/cookbook-elasticsearch)。

当你解压好了归档文件之后，Elasticsearch 已经准备好运行了。按照下面的操作，在前台(foregroud)启动 Elasticsearch：

cd elasticsearch-<version>

./bin/elasticsearch https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/running-elasticsearch.html#CO1-1) | 如果你想把 Elasticsearch 作为一个守护进程在后台运行，那么可以在后面添加参数 -d 。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/running-elasticsearch.html#CO1-2) | 如果你是在 Windows 上面运行 Elasticseach，你应该运行 bin\elasticsearch.bat 而不是 bin\elasticsearch 。 |

测试 Elasticsearch 是否启动成功，可以打开另一个终端，执行以下操作：

curl 'http://localhost:9200/?pretty'

TIP：如果你是在 Windows 上面运行 Elasticsearch，你可以从 <http://curl.haxx.se/download.html> 中下载 cURL。 cURL 给你提供了一种将请求提交到 Elasticsearch 的便捷方式，并且安装 cURL 之后，你可以通过复制与粘贴去尝试书中的许多例子。

你应该得到和下面类似的响应(response)：

{

"name" : "Tom Foster",

"cluster\_name" : "elasticsearch",

"version" : {

"number" : "2.1.0",

"build\_hash" : "72cd1f1a3eee09505e036106146dc1949dc5dc87",

"build\_timestamp" : "2015-11-18T22:40:03Z",

"build\_snapshot" : false,

"lucene\_version" : "5.3.1"

},

"tagline" : "You Know, for Search"

}

**拷贝为 CURL**[在 SENSE 中查看](http://localhost:5601/app/sense/?load_from=https://www.elastic.co/guide/cn/elasticsearch/guide/current/snippets/010_Intro/10_Info.json)

这就意味着你现在已经启动并运行一个 Elasticsearch 节点了，你可以用它做实验了。 单个 节点 可以作为一个运行中的 Elasticsearch 的实例。 而一个 集群 是一组拥有相同 cluster.name 的节点， 他们能一起工作并共享数据，还提供容错与可伸缩性。(当然，一个单独的节点也可以组成一个集群) 你可以在 elasticsearch.yml 配置文件中 修改 cluster.name ，该文件会在节点启动时加载 (译者注：这个重启服务后才会生效)。 关于上面的 cluster.name 以及其它 [Important Configuration Changes](https://www.elastic.co/guide/cn/elasticsearch/guide/current/important-configuration-changes.html) 信息， 你可以在这本书后面提供的生产部署章节找到更多。

TIP：看到下方的 View in Sense 的例子了么？[Install the Sense console](https://www.elastic.co/guide/cn/elasticsearch/guide/current/running-elasticsearch.html#sense) 使用你自己的 Elasticsearch 集群去运行这本书中的例子， 查看会有怎样的结果。

当 Elastcisearch 在前台运行时，你可以通过按 Ctrl+C 去停止。

**安装 Sense**

Sense 是一个 [Kibana](https://www.elastic.co/guide/en/kibana/current/index.html) 应用 它提供交互式的控制台，通过你的浏览器直接向 Elasticsearch 提交请求。 这本书的在线版本包含有一个 View in Sense 的链接，里面有许多代码示例。当点击的时候，它会打开一个代码示例的Sense控制台。 你不必安装 Sense，但是它允许你在本地的 Elasticsearch 集群上测试示例代码，从而使本书更具有交互性。

安装与运行 Sense：

1. 在 Kibana 目录下运行下面的命令，下载并安装 Sense app：

./bin/kibana plugin --install elastic/sense https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/running-elasticsearch.html#CO2-1) | Windows上面执行: bin\kibana.bat plugin --install elastic/sense 。  NOTE：你可以直接从这里 <https://download.elastic.co/elastic/sense/sense-latest.tar.gz> 下载 Sense 离线安装可以查看这里 [install it on an offline machine](https://www.elastic.co/guide/en/sense/current/installing.html#manual_download) 。 |

1. 启动 Kibana.

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

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/running-elasticsearch.html#CO3-1) | Windows 上启动 kibana: bin\kibana.bat 。 |

1. 在你的浏览器中打开 Sense: http://localhost:5601/app/sense 。

### 和 Elasticsearch 交互

和 Elasticsearch 的交互方式取决于 你是否使用 Java

**Java API**

如果你正在使用 Java，在代码中你可以使用 Elasticsearch 内置的两个客户端：

节点客户端（Node client）

节点客户端作为一个非数据节点加入到本地集群中。换句话说，它本身不保存任何数据，但是它知道数据在集群中的哪个节点中，并且可以把请求转发到正确的节点。

传输客户端（Transport client）

轻量级的传输客户端可以将请求发送到远程集群。它本身不加入集群，但是它可以将请求转发到集群中的一个节点上。

两个 Java 客户端都是通过 *9300* 端口并使用本地 Elasticsearch *传输* 协议和集群交互。集群中的节点通过端口 9300 彼此通信。如果这个端口没有打开，节点将无法形成一个集群。

提示

Java 客户端作为节点必须和 Elasticsearch 有相同的 *主要* 版本；否则，它们之间将无法互相理解。

更多的 Java 客户端信息可以在 [Elasticsearch Clients](https://www.elastic.co/guide/en/elasticsearch/client/index.html) 中找到。

**RESTful API with JSON over HTTP**

所有其他语言可以使用 RESTful API 通过端口 *9200* 和 Elasticsearch 进行通信，你可以用你最喜爱的 web 客户端访问 Elasticsearch 。事实上，正如你所看到的，你甚至可以使用 curl 命令来和 Elasticsearch 交互。

注意

Elasticsearch 为以下语言提供了官方客户端 --Groovy、JavaScript、.NET、 PHP、 Perl、 Python 和 Ruby--还有很多社区提供的客户端和插件，所有这些都可以在 [Elasticsearch Clients](https://www.elastic.co/guide/en/elasticsearch/client/index.html) 中找到。

一个 Elasticsearch 请求和任何 HTTP 请求一样由若干相同的部件组成：

curl -X<VERB> '<PROTOCOL>://<HOST>:<PORT>/<PATH>?<QUERY\_STRING>' -d '<BODY>'

被 < > 标记的部件：

|  |  |
| --- | --- |
| VERB | 适当的 HTTP *方法* 或 *谓词* : GET`、 `POST`、 `PUT`、 `HEAD 或者 `DELETE`。 |
| PROTOCOL | http 或者 https`（如果你在 Elasticsearch 前面有一个 `https 代理） |
| HOST | Elasticsearch 集群中任意节点的主机名，或者用 localhost 代表本地机器上的节点。 |
| PORT | 运行 Elasticsearch HTTP 服务的端口号，默认是 9200 。 |
| PATH | API 的终端路径（例如 \_count 将返回集群中文档数量）。Path 可能包含多个组件，例如：\_cluster/stats 和 \_nodes/stats/jvm 。 |
| QUERY\_STRING | 任意可选的查询字符串参数 (例如 ?pretty 将格式化地输出 JSON 返回值，使其更容易阅读) |
| BODY | 一个 JSON 格式的请求体 (如果请求需要的话) |

例如，计算集群中文档的数量，我们可以用这个:

curl -XGET 'http://localhost:9200/\_count?pretty' -d '

{

"query": {

"match\_all": {}

}

}

'

Elasticsearch 返回一个 HTTP 状态码（例如：200 OK`）和（除`HEAD`请求）一个 JSON 格式的返回值。前面的 `curl 请求将返回一个像下面一样的 JSON 体：

{

"count" : 0,

"\_shards" : {

"total" : 5,

"successful" : 5,

"failed" : 0

}

}

在返回结果中没有看到 HTTP 头信息是因为我们没有要求curl`显示它们。想要看到头信息，需要结合 `-i 参数来使用 curl 命令：

curl -i -XGET 'localhost:9200/'

在书中剩余的部分，我们将用缩写格式来展示这些 curl 示例，所谓的缩写格式就是省略请求中所有相同的部分，例如主机名、端口号以及 curl 命令本身。而不是像下面显示的那样用一个完整的请求：

curl -XGET 'localhost:9200/\_count?pretty' -d '

{

"query": {

"match\_all": {}

}

}'

我们将用缩写格式显示：

GET /\_count

{

"query": {

"match\_all": {}

}

}

事实上， [Sense 控制台](https://www.elastic.co/guide/cn/elasticsearch/guide/current/running-elasticsearch.html#sense) 也使用这样相同的格式。如果你正在阅读这本书的在线版本,可以通过点击 Sense 链接视图在 Sense 上打开和运行示例代码。

### 面向文档

在应用程序中对象很少只是一个简单的键和值的列表。通常，它们拥有更复杂的数据结构，可能包括日期、地理信息、其他对象或者数组等。

也许有一天你想把这些对象存储在数据库中。使用关系型数据库的行和列存储，这相当于是把一个表现力丰富的对象挤压到一个非常大的电子表格中：你必须将这个对象扁平化来适应表结构--通常一个字段>对应一列--而且又不得不在每次查询时重新构造对象。

Elasticsearch 是 *面向文档* 的，意味着它存储整个对象或 *文档\_。Elasticsearch 不仅存储文档，而且 \_索引*每个文档的内容使之可以被检索。在 Elasticsearch 中，你  对文档进行索引、检索、排序和过滤--而不是对行列数据。这是一种完全不同的思考数据的方式，也是 Elasticsearch 能支持复杂全文检索的原因。

**JSON**

Elasticsearch 使用 JavaScript Object Notation 或者 [*JSON*](http://en.wikipedia.org/wiki/Json) 作为文档的序列化格式。JSON 序列化被大多数编程语言所支持，并且已经成为 NoSQL 领域的标准格式。 它简单、简洁、易于阅读。

考虑一下这个 JSON 文档，它代表了一个 user 对象：

{

"email": "john@smith.com",

"first\_name": "John",

"last\_name": "Smith",

"info": {

"bio": "Eco-warrior and defender of the weak",

"age": 25,

"interests": [ "dolphins", "whales" ]

},

"join\_date": "2014/05/01"

}

虽然原始的 user 对象很复杂，但这个对象的结构和含义在 JSON 版本中都得到了体现和保留。在 Elasticsearch 中将对象转化为 JSON 并做索引要比在一个扁平的表结构中做相同的事情简单的多。

注意

几乎所有的语言都有相应的模块可以将任意的数据结构或对象 转化成 JSON 格式，只是细节各不相同。具体请查看 *serialization* 或者 *marshalling* 这两个 处理 JSON 的模块。官方 [Elasticsearch 客户端](https://www.elastic.co/guide/en/elasticsearch/client/index.html) 自动为您提供 JSON 转化。

### 适应新环境

为了对 Elasticsearch 能实现什么及其上手容易程度有一个基本印象，让我们从一个简单的教程开始并介绍索引、搜索及聚合等基础概念。

我们将一并介绍一些新的技术术语和基础概念，因此即使无法立即全盘理解也无妨。在本书后续内容中，我们将深入介绍这里提到的所有概念。

接下来尽情享受 Elasticsearch 探索之旅。

**创建一个雇员目录**

我们受雇于 *Megacorp* 公司，作为 HR 部门新的 *“热爱无人机”* （\_"We love our drones!"\_）激励项目的一部分，我们的任务是为此创建一个雇员目录。该目录应当能培养雇员认同感及支持实时、高效、动态协作，因此有一些业务需求：

* 支持包含多值标签、数值、以及全文本的数据
* 检索任一雇员的完整信息
* 允许结构化搜索，比如查询 30 岁以上的员工
* 允许简单的全文搜索以及较复杂的短语搜索
* 支持在匹配文档内容中高亮显示搜索片段
* 支持基于数据创建和管理分析仪表盘

### 索引雇员文档

第一个业务需求就是存储雇员数据。 这将会以 *雇员文档* 的形式存储：一个文档代表一个雇员。存储数据到 Elasticsearch 的行为叫做 *索引* ，但在索引一个文档之前，需要确定将文档存储在哪里。

一个 Elasticsearch 集群可以 包含多个 *索引* ，相应的每个索引可以包含多个 *类型* 。 这些不同的类型存储着多个 *文档* ，每个文档又有 多个 *属性* 。

**Index Versus Index Versus Index**

你也许已经注意到 *索引* 这个词在 Elasticsearch 语境中包含多重意思， 所以有必要做一点儿说明：

索引（名词）：

如前所述，一个 *索引* 类似于传统关系数据库中的一个 *数据库* ，是一个存储关系型文档的地方。 *索引* (*index*) 的复数词为 *indices* 或 *indexes* 。

索引（动词）：

*索引一个文档* 就是存储一个文档到一个 *索引* （名词）中以便它可以被检索和查询到。这非常类似于 SQL 语句中的 INSERT 关键词，除了文档已存在时新文档会替换旧文档情况之外。

倒排索引：

关系型数据库通过增加一个 *索引* 比如一个 B树（B-tree）索引 到指定的列上，以便提升数据检索速度。Elasticsearch 和 Lucene 使用了一个叫做 *倒排索引* 的结构来达到相同的目的。

+ 默认的，一个文档中的每一个属性都是 *被索引* 的（有一个倒排索引）和可搜索的。一个没有倒排索引的属性是不能被搜索到的。我们将在 [倒排索引](https://www.elastic.co/guide/cn/elasticsearch/guide/current/inverted-index.html) 讨论倒排索引的更多细节。

对于雇员目录，我们将做如下操作：

* 每个雇员索引一个文档，包含该雇员的所有信息。
* 每个文档都将是 employee *类型* 。
* 该类型位于 *索引* megacorp 内。
* 该索引保存在我们的 Elasticsearch 集群中。

实践中这非常简单（尽管看起来有很多步骤），我们可以通过一条命令完成所有这些动作：

PUT /megacorp/employee/1

{

"first\_name" : "John",

"last\_name" : "Smith",

"age" : 25,

"about" : "I love to go rock climbing",

"interests": [ "sports", "music" ]

}

注意，路径 /megacorp/employee/1 包含了三部分的信息：

megacorp

索引名称

employee

类型名称

1

特定雇员的ID

请求体 —— JSON 文档 —— 包含了这位员工的所有详细信息，他的名字叫 John Smith ，今年 25 岁，喜欢攀岩。

很简单！无需进行执行管理任务，如创建一个索引或指定每个属性的数据类型之类的，可以直接只索引一个文档。Elasticsearch 默认地完成其他一切，因此所有必需的管理任务都在后台使用默认设置完成。

进行下一步前，让我们增加更多的员工信息到目录中：

PUT /megacorp/employee/2

{

"first\_name" : "Jane",

"last\_name" : "Smith",

"age" : 32,

"about" : "I like to collect rock albums",

"interests": [ "music" ]

}

PUT /megacorp/employee/3

{

"first\_name" : "Douglas",

"last\_name" : "Fir",

"age" : 35,

"about": "I like to build cabinets",

"interests": [ "forestry" ]

}

### 检索文档

目前我们已经在 Elasticsearch 中存储了一些数据， 接下来就能专注于实现应用的业务需求了。第一个需求是可以检索到单个雇员的数据。

这在 Elasticsearch 中很简单。简单地执行 一个 HTTP GET 请求并指定文档的地址——索引库、类型和ID。 使用这三个信息可以返回原始的 JSON 文档：

GET /megacorp/employee/1

返回结果包含了文档的一些元数据，以及 \_source 属性，内容是 John Smith 雇员的原始 JSON 文档：

{

"\_index" : "megacorp",

"\_type" : "employee",

"\_id" : "1",

"\_version" : 1,

"found" : true,

"\_source" : {

"first\_name" : "John",

"last\_name" : "Smith",

"age" : 25,

"about" : "I love to go rock climbing",

"interests": [ "sports", "music" ]

}

}

提示

将 HTTP 命令由 PUT 改为 GET 可以用来检索文档，同样的，可以使用 DELETE 命令来删除文档，以及使用 HEAD 指令来检查文档是否存在。如果想更新已存在的文档，只需再次 PUT 。

### 轻量搜索

一个 GET 是相当简单的，可以直接得到指定的文档。 现在尝试点儿稍微高级的功能，比如一个简单的搜索！

第一个尝试的几乎是最简单的搜索了。我们使用下列请求来搜索所有雇员：

GET /megacorp/employee/\_search

可以看到，我们仍然使用索引库 megacorp 以及类型 employee`，但与指定一个文档 ID 不同，这次使用 `\_search 。返回结果包括了所有三个文档，放在数组 hits 中。一个搜索默认返回十条结果。

{

"took": 6,

"timed\_out": false,

"\_shards": { ... },

"hits": {

"total": 3,

"max\_score": 1,

"hits": [

{

"\_index": "megacorp",

"\_type": "employee",

"\_id": "3",

"\_score": 1,

"\_source": {

"first\_name": "Douglas",

"last\_name": "Fir",

"age": 35,

"about": "I like to build cabinets",

"interests": [ "forestry" ]

}

},

{

"\_index": "megacorp",

"\_type": "employee",

"\_id": "1",

"\_score": 1,

"\_source": {

"first\_name": "John",

"last\_name": "Smith",

"age": 25,

"about": "I love to go rock climbing",

"interests": [ "sports", "music" ]

}

},

{

"\_index": "megacorp",

"\_type": "employee",

"\_id": "2",

"\_score": 1,

"\_source": {

"first\_name": "Jane",

"last\_name": "Smith",

"age": 32,

"about": "I like to collect rock albums",

"interests": [ "music" ]

}

}

]

}

}

注意：返回结果不仅告知匹配了哪些文档，还包含了整个文档本身：显示搜索结果给最终用户所需的全部信息。

接下来，尝试下搜索姓氏为 ``Smith`` 的雇员。为此，我们将使用一个 *高亮* 搜索，很容易通过命令行完成。这个方法一般涉及到一个 *查询字符串* （\_query-string\_） 搜索，因为我们通过一个URL参数来传递查询信息给搜索接口：

GET /megacorp/employee/\_search?q=last\_name:Smith

我们仍然在请求路径中使用 \_search 端点，并将查询本身赋值给参数 q= 。返回结果给出了所有的 Smith：

{

...

"hits": {

"total": 2,

"max\_score": 0.30685282,

"hits": [

{

...

"\_source": {

"first\_name": "John",

"last\_name": "Smith",

"age": 25,

"about": "I love to go rock climbing",

"interests": [ "sports", "music" ]

}

},

{

...

"\_source": {

"first\_name": "Jane",

"last\_name": "Smith",

"age": 32,

"about": "I like to collect rock albums",

"interests": [ "music" ]

}

}

]

}

}

### 使用查询表达式搜索

Query-string 搜索通过命令非常方便地进行临时性的即席搜索 ，但它有自身的局限性（参见 [*轻量* 搜索](https://www.elastic.co/guide/cn/elasticsearch/guide/current/search-lite.html) ）。Elasticsearch 提供一个丰富灵活的查询语言叫做 *查询表达式* ， 它支持构建更加复杂和健壮的查询。

*领域特定语言* （DSL）， 指定了使用一个 JSON 请求。我们可以像这样重写之前的查询所有 Smith 的搜索 ：

GET /megacorp/employee/\_search

{

"query" : {

"match" : {

"last\_name" : "Smith"

}

}

}

返回结果与之前的查询一样，但还是可以看到有一些变化。其中之一是，不再使用 *query-string* 参数，而是一个请求体替代。这个请求使用 JSON 构造，并使用了一个 match 查询（属于查询类型之一，后续将会了解）。

### 更复杂的搜索

现在尝试下更复杂的搜索。 同样搜索姓氏为 Smith 的雇员，但这次我们只需要年龄大于 30 的。查询需要稍作调整，使用过滤器 *filter* ，它支持高效地执行一个结构化查询。

GET /megacorp/employee/\_search

{

"query" : {

"bool": {

"must": {

"match" : {

"last\_name" : "smith" https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

}

},

"filter": {

"range" : {

"age" : { "gt" : 30 } https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

}

}

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_more_complicated_searches.html#CO4-1) | 这部分与我们之前使用的 match *查询* 一样。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_more_complicated_searches.html#CO4-2) | 这部分是一个 range *过滤器* ， 它能找到年龄大于 30 的文档，其中 gt 表示\_大于*(\_great than*)。 |

目前无需太多担心语法问题，后续会更详细地介绍。只需明确我们添加了一个 *过滤器* 用于执行一个范围查询，并复用之前的 match 查询。现在结果只返回了一个雇员，叫 Jane Smith，32 岁。

{

...

"hits": {

"total": 1,

"max\_score": 0.30685282,

"hits": [

{

...

"\_source": {

"first\_name": "Jane",

"last\_name": "Smith",

"age": 32,

"about": "I like to collect rock albums",

"interests": [ "music" ]

}

}

]

}

}

### 全文搜索

截止目前的搜索相对都很简单：单个姓名，通过年龄过滤。现在尝试下稍微高级点儿的全文搜索——一项传统数据库确实很难搞定的任务。

搜索下所有喜欢攀岩（rock climbing）的雇员：

GET /megacorp/employee/\_search

{

"query" : {

"match" : {

"about" : "rock climbing"

}

}

}

显然我们依旧使用之前的 match 查询在about 属性上搜索 “rock climbing” 。得到两个匹配的文档：

{

...

"hits": {

"total": 2,

"max\_score": 0.16273327,

"hits": [

{

...

"\_score": 0.16273327, https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

"\_source": {

"first\_name": "John",

"last\_name": "Smith",

"age": 25,

"about": "I love to go rock climbing",

"interests": [ "sports", "music" ]

}

},

{

...

"\_score": 0.016878016, https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

"\_source": {

"first\_name": "Jane",

"last\_name": "Smith",

"age": 32,

"about": "I like to collect rock albums",

"interests": [ "music" ]

}

}

]

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_full_text_search.html#CO5-1) [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_full_text_search.html#CO5-2) | 相关性得分 |

Elasticsearch 默认按照相关性得分排序，即每个文档跟查询的匹配程度。第一个最高得分的结果很明显：John Smith 的 about 属性清楚地写着 “rock climbing” 。

但为什么 Jane Smith 也作为结果返回了呢？原因是她的 about 属性里提到了 “rock” 。因为只有 “rock” 而没有 “climbing” ，所以她的相关性得分低于 John 的。

这是一个很好的案例，阐明了 Elasticsearch 如何 *在* 全文属性上搜索并返回相关性最强的结果。Elasticsearch中的 *相关性* 概念非常重要，也是完全区别于传统关系型数据库的一个概念，数据库中的一条记录要么匹配要么不匹配。

### 短语搜索

找出一个属性中的独立单词是没有问题的，但有时候想要精确匹配一系列单词或者*短语* 。 比如， 我们想执行这样一个查询，仅匹配同时包含 “rock” *和* “climbing” ，*并且* 二者以短语 “rock climbing” 的形式紧挨着的雇员记录。

为此对 match 查询稍作调整，使用一个叫做 match\_phrase 的查询：

GET /megacorp/employee/\_search

{

"query" : {

"match\_phrase" : {

"about" : "rock climbing"

}

}

}

毫无悬念，返回结果仅有 John Smith 的文档。

{

...

"hits": {

"total": 1,

"max\_score": 0.23013961,

"hits": [

{

...

"\_score": 0.23013961,

"\_source": {

"first\_name": "John",

"last\_name": "Smith",

"age": 25,

"about": "I love to go rock climbing",

"interests": [ "sports", "music" ]

}

}

]

}

}

### 高亮搜索

许多应用都倾向于在每个搜索结果中 *高亮* 部分文本片段，以便让用户知道为何该文档符合查询条件。在 Elasticsearch 中检索出高亮片段也很容易。

再次执行前面的查询，并增加一个新的 highlight 参数：

GET /megacorp/employee/\_search

{

"query" : {

"match\_phrase" : {

"about" : "rock climbing"

}

},

"highlight": {

"fields" : {

"about" : {}

}

}

}

当执行该查询时，返回结果与之前一样，与此同时结果中还多了一个叫做 highlight 的部分。这个部分包含了 about 属性匹配的文本片段，并以 HTML 标签 <em></em> 封装：

{

...

"hits": {

"total": 1,

"max\_score": 0.23013961,

"hits": [

{

...

"\_score": 0.23013961,

"\_source": {

"first\_name": "John",

"last\_name": "Smith",

"age": 25,

"about": "I love to go rock climbing",

"interests": [ "sports", "music" ]

},

"highlight": {

"about": [

"I love to go <em>rock</em> <em>climbing</em>" https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

]

}

}

]

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/highlighting-intro.html#CO6-1) | 原始文本中的高亮片段 |

关于高亮搜索片段，可以在 [highlighting reference documentation](https://www.elastic.co/guide/en/elasticsearch/reference/master/search-request-highlighting.html) 了解更多信息。

### 分析

终于到了最后一个业务需求：支持管理者对雇员目录做分析。 Elasticsearch 有一个功能叫聚合（aggregations），允许我们基于数据生成一些精细的分析结果。聚合与 SQL 中的 GROUP BY 类似但更强大。

举个例子，挖掘出雇员中最受欢迎的兴趣爱好：

GET /megacorp/employee/\_search

{

"aggs": {

"all\_interests": {

"terms": { "field": "interests" }

}

}

}

暂时忽略掉语法，直接看看结果：

{

...

"hits": { ... },

"aggregations": {

"all\_interests": {

"buckets": [

{

"key": "music",

"doc\_count": 2

},

{

"key": "forestry",

"doc\_count": 1

},

{

"key": "sports",

"doc\_count": 1

}

]

}

}

}

可以看到，两位员工对音乐感兴趣，一位对林地感兴趣，一位对运动感兴趣。这些聚合并非预先统计，而是从匹配当前查询的文档中即时生成。如果想知道叫 Smith 的雇员中最受欢迎的兴趣爱好，可以直接添加适当的查询来组合查询：

GET /megacorp/employee/\_search

{

"query": {

"match": {

"last\_name": "smith"

}

},

"aggs": {

"all\_interests": {

"terms": {

"field": "interests"

}

}

}

}

all\_interests 聚合已经变为只包含匹配查询的文档：

...

"all\_interests": {

"buckets": [

{

"key": "music",

"doc\_count": 2

},

{

"key": "sports",

"doc\_count": 1

}

]

}

聚合还支持分级汇总 。比如，查询特定兴趣爱好员工的平均年龄：

GET /megacorp/employee/\_search

{

"aggs" : {

"all\_interests" : {

"terms" : { "field" : "interests" },

"aggs" : {

"avg\_age" : {

"avg" : { "field" : "age" }

}

}

}

}

}

得到的聚合结果有点儿复杂，但理解起来还是很简单的：

...

"all\_interests": {

"buckets": [

{

"key": "music",

"doc\_count": 2,

"avg\_age": {

"value": 28.5

}

},

{

"key": "forestry",

"doc\_count": 1,

"avg\_age": {

"value": 35

}

},

{

"key": "sports",

"doc\_count": 1,

"avg\_age": {

"value": 25

}

}

]

}

输出基本是第一次聚合的加强版。依然有一个兴趣及数量的列表，只不过每个兴趣都有了一个附加的 avg\_age 属性，代表有这个兴趣爱好的所有员工的平均年龄。

即使现在不太理解这些语法也没有关系，依然很容易了解到复杂聚合及分组通过 Elasticsearch 特性实现得很完美。可提取的数据类型毫无限制。

### 教程结语

欣喜的是，这是一个关于 Elasticsearch 基础描述的教程，且仅仅是浅尝辄止，更多诸如 suggestions、geolocation、percolation、fuzzy 与 partial matching 等特性均被省略，以便保持教程的简洁。但它确实突显了开始构建高级搜索功能多么容易。不需要配置——只需要添加数据并开始搜索！

很可能语法会让你在某些地方有所困惑，并且对各个方面如何微调也有一些问题。没关系！本书后续内容将针对每个问题详细解释，让你全方位地理解 Elasticsearch 的工作原理。

### 分布式特性

在本章开头，我们提到过 Elasticsearch 可以横向扩展至数百（甚至数千）的服务器节点，同时可以处理PB级数据。我们的教程给出了一些使用 Elasticsearch 的示例，但并不涉及任何内部机制。Elasticsearch 天生就是分布式的，并且在设计时屏蔽了分布式的复杂性。

Elasticsearch 在分布式方面几乎是透明的。教程中并不要求了解分布式系统、分片、集群发现或其他的各种分布式概念。可以使用笔记本上的单节点轻松地运行教程里的程序，但如果你想要在 100 个节点的集群上运行程序，一切依然顺畅。

Elasticsearch 尽可能地屏蔽了分布式系统的复杂性。这里列举了一些在后台自动执行的操作：

* 分配文档到不同的容器 或 *分片* 中，文档可以储存在一个或多个节点中
* 按集群节点来均衡分配这些分片，从而对索引和搜索过程进行负载均衡
* 复制每个分片以支持数据冗余，从而防止硬件故障导致的数据丢失
* 将集群中任一节点的请求路由到存有相关数据的节点
* 集群扩容时无缝整合新节点，重新分配分片以便从离群节点恢复

当阅读本书时，将会遇到有关 Elasticsearch 分布式特性的补充章节。这些章节将介绍有关集群扩容、故障转移([*集群内的原理*](https://www.elastic.co/guide/cn/elasticsearch/guide/current/distributed-cluster.html)) 、应对文档存储([*分布式文档存储*](https://www.elastic.co/guide/cn/elasticsearch/guide/current/distributed-docs.html)) 、执行分布式搜索([*执行分布式检索*](https://www.elastic.co/guide/cn/elasticsearch/guide/current/distributed-search.html)) ，以及分区（shard）及其工作原理([*分片内部原理*](https://www.elastic.co/guide/cn/elasticsearch/guide/current/inside-a-shard.html)) 。

这些章节并非必读，完全可以无需了解内部机制就使用 Elasticsearch，但是它们将从另一个角度帮助你了解更完整的 Elasticsearch 知识。可以根据需要跳过它们，或者想更完整地理解时再回头阅读也无妨。

### 后续步骤

现在对于通过 Elasticsearch 能够实现什么样的功能、以及上手的简易程度应该有了初步概念。Elasticsearch 力图通过最少的知识和配置做到开箱即用。学习 Elasticsearch 的最好方式是投入实践：尽管开始索引和搜索吧！

然而，对于 Elasticsearch 知道得越多，就越有生产效率。告诉 Elasticsearch 越多的领域知识，就越容易进行结果调优。

本书的后续内容将帮助你从新手成长为专家，每个章节不仅阐述必要的基础知识，而且包含专家建议。如果刚刚上手，这些建议可能无法立竿见影；但 Elasticsearch 有着合理的默认设置，在无需干预的情况下通常都能工作得很好。当追求毫秒级的性能提升时，随时可以重温这些章节。

## 集群内的原理

### 补充章节

如前文所述，这是补充章节中第一篇介绍 Elasticsearch 在分布式环境中的运行原理。 在这个章节中，我们将会介绍 *cluster* 、 *node* 、 *shard* 等常用术语，Elastisearch 的扩容机制， 以及如何处理硬件故障的内容。

虽然这个章节不是必读的--您完全可以在不关注分片、副本和失效切换等内容的情况下长期使用Elasticsearch-- 但是这将帮助你了解 Elasticsearch 的内部工作过程。您可以先快速阅览该章节，将来有需要时再次查看。

ElasticSearch 的主旨是随时可用和按需扩容。 而扩容可以通过购买性能更强大（ *垂直扩容* ，或 *纵向扩容*）  或者数量更多的服务器（ *水平扩容* ，或 *横向扩容* ）来实现。

虽然 Elasticsearch 可以获益于更强大的硬件设备，但是垂直扩容是有极限的。 真正的扩容能力是来自于水平扩容--为集群添加更多的节点，并且将负载压力和稳定性分散到这些节点中。

对于大多数的数据库而言，通常需要对应用程序进行非常大的改动，才能利用上横向扩容的新增资源。 与之相反的是，ElastiSearch天生就是 *分布式的* ，它知道如何通过管理多节点来提高扩容性和可用性。 这也意味着你的应用无需关注这个问题。

本章将讲述如何按需配置集群、节点和分片，并在硬件故障时确保数据安全。

### 空集群

如果我们启动了一个单独的节点，里面不包含任何的数据和 索引，那我们的集群看起来就是一个 [图 1 “包含空内容节点的集群”](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_an-empty-cluster.html#img-cluster)。

**图 1. 包含空内容节点的集群**



一个运行中的 Elasticsearch 实例称为一个 节点，而集群是由一个或者多个拥有相同 cluster.name 配置的节点组成， 它们共同承担数据和负载的压力。当有节点加入集群中或者从集群中移除节点时，集群将会重新平均分布所有的数据。

当一个节点被选举成为 *主* 节点时， 它将负责管理集群范围内的所有变更，例如增加、删除索引，或者增加、删除节点等。 而主节点并不需要涉及到文档级别的变更和搜索等操作，所以当集群只拥有一个主节点的情况下，即使流量的增加它也不会成为瓶颈。 任何节点都可以成为主节点。我们的示例集群就只有一个节点，所以它同时也成为了主节点。

作为用户，我们可以将请求发送到 *集群中的任何节点* ，包括主节点。 每个节点都知道任意文档所处的位置，并且能够将我们的请求直接转发到存储我们所需文档的节点。 无论我们将请求发送到哪个节点，它都能负责从各个包含我们所需文档的节点收集回数据，并将最终结果返回給客户端。 Elasticsearch 对这一切的管理都是透明的。

### 集群健康

Elasticsearch 的集群监控信息中包含了许多的统计数据，其中最为重要的一项就是 *集群健康* ， 它在 status 字段中展示为 green 、 yellow 或者 red 。

GET /\_cluster/health

在一个不包含任何索引的空集群中，它将会有一个类似于如下所示的返回内容：

{

"cluster\_name": "elasticsearch",

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

"timed\_out": false,

"number\_of\_nodes": 1,

"number\_of\_data\_nodes": 1,

"active\_primary\_shards": 0,

"active\_shards": 0,

"relocating\_shards": 0,

"initializing\_shards": 0,

"unassigned\_shards": 0

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/cluster-health.html#CO7-1) | status 字段是我们最关心的。 |

status 字段指示着当前集群在总体上是否工作正常。它的三种颜色含义如下：

green

所有的主分片和副本分片都正常运行。

yellow

所有的主分片都正常运行，但不是所有的副本分片都正常运行。

red

有主分片没能正常运行。

在本章节剩余的部分，我们将解释什么是 *主* 分片和 *副本* 分片，以及上面提到的这些颜色的实际意义。

### 添加索引

我们往 Elasticsearch 添加数据时需要用到 索引 —— 保存相关数据的地方。 索引实际上是指向一个或者多个物理 分片 的 逻辑命名空间 。

一个 分片 是一个底层的 工作单元 ，它仅保存了 全部数据中的一部分。 在[分片内部机制](https://www.elastic.co/guide/cn/elasticsearch/guide/current/inside-a-shard.html)中，我们将详细介绍分片是如何工作的，而现在我们只需知道一个分片是一个 Lucene 的实例，以及它本身就是一个完整的搜索引擎。 我们的文档被存储和索引到分片内，但是应用程序是直接与索引而不是与分片进行交互。

Elasticsearch 是利用分片将数据分发到集群内各处的。分片是数据的容器，文档保存在分片内，分片又被分配到集群内的各个节点里。 当你的集群规模扩大或者缩小时， Elasticsearch 会自动的在各节点中迁移分片，使得数据仍然均匀分布在集群里。

一个分片可以是 主 分片或者 副本 分片。 索引内任意一个文档都归属于一个主分片，所以主分片的数目决定着索引能够保存的最大数据量。

注意

技术上来说，一个主分片最大能够存储 Integer.MAX\_VALUE - 128 个文档，但是实际最大值还需要参考你的使用场景：包括你使用的硬件， 文档的大小和复杂程度，索引和查询文档的方式以及你期望的响应时长。

一个副本分片只是一个主分片的拷贝。 副本分片作为硬件故障时保护数据不丢失的冗余备份，并为搜索和返回文档等读操作提供服务。

在索引建立的时候就已经确定了主分片数，但是副本分片数可以随时修改。

让我们在包含一个空节点的集群内创建名为 blogs 的索引。 索引在默认情况下会被分配5个主分片， 但是为了演示目的，我们将分配3个主分片和一份副本（每个主分片拥有一个副本分片）：

PUT /blogs

{

"settings" : {

"number\_of\_shards" : 3,

"number\_of\_replicas" : 1

}

}

我们的集群现在是[图 2 “拥有一个索引的单节点集群”](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_add-an-index.html#cluster-one-node)。所有3个主分片都被分配在 Node 1 。

**图 2. 拥有一个索引的单节点集群**



如果我们现在查看[集群健康](https://www.elastic.co/guide/cn/elasticsearch/guide/current/cluster-health.html)， 我们将看到如下内容：

{

"cluster\_name": "elasticsearch",

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

"timed\_out": false,

"number\_of\_nodes": 1,

"number\_of\_data\_nodes": 1,

"active\_primary\_shards": 3,

"active\_shards": 3,

"relocating\_shards": 0,

"initializing\_shards": 0,

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

"delayed\_unassigned\_shards": 0,

"number\_of\_pending\_tasks": 0,

"number\_of\_in\_flight\_fetch": 0,

"task\_max\_waiting\_in\_queue\_millis": 0,

"active\_shards\_percent\_as\_number": 50

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_add-an-index.html#CO8-1) | 集群 status 值为 yellow 。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_add-an-index.html#CO8-2) | 没有被分配到任何节点的副本数。 |

集群的健康状况为 yellow 则表示全部 主 分片都正常运行（集群可以正常服务所有请求），但是 副本 分片没有全部处在正常状态。 实际上，所有3个副本分片都是 unassigned —— 它们都没有被分配到任何节点。 在同一个节点上既保存原始数据又保存副本是没有意义的，因为一旦失去了那个节点，我们也将丢失该节点上的所有副本数据。

当前我们的集群是正常运行的，但是在硬件故障时有丢失数据的风险。

### 添加故障转移

当集群中只有一个节点在运行时，意味着会有一个单点故障问题——没有冗余。 幸运的是，我们只需再启动一个节点即可防止数据丢失。

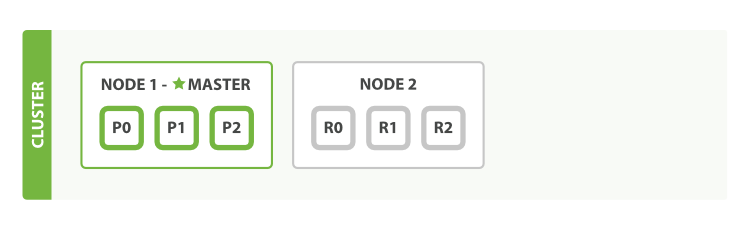
**启动第二个节点**

为了测试第二个节点启动后的情况，你可以在同一个目录内，完全依照启动第一个节点的方式来启动一个新节点（参考[安装并运行 Elasticsearch](https://www.elastic.co/guide/cn/elasticsearch/guide/current/running-elasticsearch.html)）。多个节点可以共享同一个目录。

当你在同一台机器上启动了第二个节点时，只要它和第一个节点有同样的 cluster.name 配置，它就会自动发现集群并加入到其中。 但是在不同机器上启动节点的时候，为了加入到同一集群，你需要配置一个可连接到的单播主机列表。 详细信息请查看[最好使用单播代替组播](https://www.elastic.co/guide/cn/elasticsearch/guide/current/important-configuration-changes.html#unicast)

如果启动了第二个节点，我们的集群将会如[图 3 “拥有两个节点的集群——所有主分片和副本分片都已被分配”](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_add_failover.html#cluster-two-nodes)所示。

**图 3. 拥有两个节点的集群——所有主分片和副本分片都已被分配**



当第二个节点加入到集群后，3个 副本分片 将会分配到这个节点上——每个主分片对应一个副本分片。 这意味着当集群内任何一个节点出现问题时，我们的数据都完好无损。

所有新近被索引的文档都将会保存在主分片上，然后被并行的复制到对应的副本分片上。这就保证了我们既可以从主分片又可以从副本分片上获得文档。

cluster-health 现在展示的状态为 green ，这表示所有6个分片（包括3个主分片和3个副本分片）都在正常运行。

{

"cluster\_name": "elasticsearch",

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

"timed\_out": false,

"number\_of\_nodes": 2,

"number\_of\_data\_nodes": 2,

"active\_primary\_shards": 3,

"active\_shards": 6,

"relocating\_shards": 0,

"initializing\_shards": 0,

"unassigned\_shards": 0,

"delayed\_unassigned\_shards": 0,

"number\_of\_pending\_tasks": 0,

"number\_of\_in\_flight\_fetch": 0,

"task\_max\_waiting\_in\_queue\_millis": 0,

"active\_shards\_percent\_as\_number": 100

}

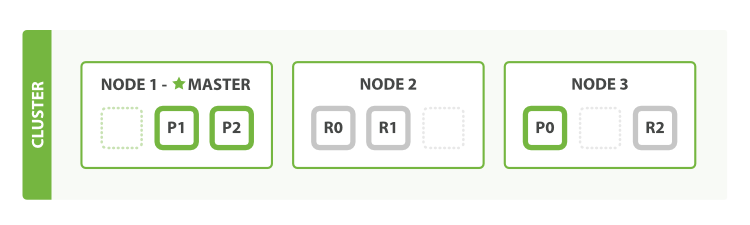
|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_add_failover.html#CO9-1) | 集群 status 值为 green 。 |

我们的集群现在不仅仅是正常运行的，并且还处于 始终可用 的状态。

### 水平扩容

怎样为我们的正在增长中的应用程序按需扩容呢？ 当启动了第三个节点，我们的集群将会看起来如[图 4 “拥有三个节点的集群——为了分散负载而对分片进行重新分配”](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_scale_horizontally.html#cluster-three-nodes)所示。

**图 4. 拥有三个节点的集群——为了分散负载而对分片进行重新分配**



Node 1 和 Node 2 上各有一个分片被迁移到了新的 Node 3 节点，现在每个节点上都拥有2个分片，而不是之前的3个。 这表示每个节点的硬件资源（CPU, RAM, I/O）将被更少的分片所共享，每个分片的性能将会得到提升。

分片是一个功能完整的搜索引擎，它拥有使用一个节点上的所有资源的能力。 我们这个拥有6个分片（3个主分片和3个副本分片）的索引可以最大扩容到6个节点，每个节点上存在一个分片，并且每个分片拥有所在节点的全部资源。

**更多的扩容**

但是如果我们想要扩容超过6个节点怎么办呢？

主分片的数目在索引创建时 就已经确定了下来。实际上，这个数目定义了这个索引能够 存储 的最大数据量。（实际大小取决于你的数据、硬件和使用场景。） 但是，读操作——搜索和返回数据——可以同时被主分片 或 副本分片所处理，所以当你拥有越多的副本分片时，也将拥有越高的吞吐量。

在运行中的集群上是可以动态调整副本分片数目的 ，我们可以按需伸缩集群。让我们把副本数从默认的 1增加到 2 ：

PUT /blogs/\_settings

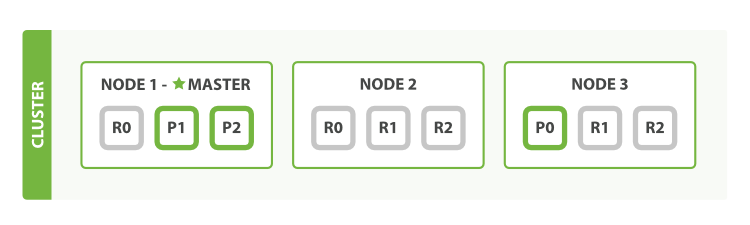
{

"number\_of\_replicas" : 2

}

如[图 5 “将参数 number\_of\_replicas 调大到 2”](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_scale_horizontally.html#cluster-three-nodes-two-replicas)所示， blogs 索引现在拥有9个分片：3个主分片和6个副本分片。 这意味着我们可以将集群扩容到9个节点，每个节点上一个分片。相比原来3个节点时，集群搜索性能可以提升 3 倍。

**图 5. 将参数 number\_of\_replicas 调大到 2**



注意

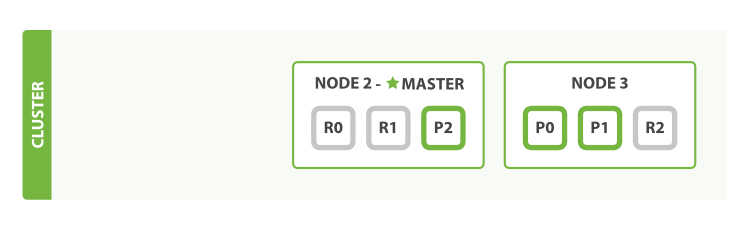
当然，如果只是在相同节点数目的集群上增加更多的副本分片并不能提高性能，因为每个分片从节点上获得的资源会变少。 你需要增加更多的硬件资源来提升吞吐量。

但是更多的副本分片数提高了数据冗余量：按照上面的节点配置，我们可以在失去2个节点的情况下不丢失任何数据。

### 应对故障

我们之前说过 Elasticsearch 可以应对节点故障，接下来让我们尝试下这个功能。 如果我们关闭第一个节点，这时集群的状态为[图 6 “关闭了一个节点后的集群”](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_coping_with_failure.html#cluster-post-kill)

**图 6. 关闭了一个节点后的集群**



我们关闭的节点是一个主节点。而集群必须拥有一个主节点来保证正常工作，所以发生的第一件事情就是选举一个新的主节点： Node 2 。

在我们关闭 Node 1 的同时也失去了主分片 1 和 2 ，并且在缺失主分片的时候索引也不能正常工作。 如果此时来检查集群的状况，我们看到的状态将会为 red ：不是所有主分片都在正常工作。

幸运的是，在其它节点上存在着这两个主分片的完整副本， 所以新的主节点立即将这些分片在 Node 2 和 Node 3 上对应的副本分片提升为主分片， 此时集群的状态将会为 yellow 。 这个提升主分片的过程是瞬间发生的，如同按下一个开关一般。

为什么我们集群状态是 yellow 而不是 green 呢？ 虽然我们拥有所有的三个主分片，但是同时设置了每个主分片需要对应2份副本分片，而此时只存在一份副本分片。 所以集群不能为 green 的状态，不过我们不必过于担心：如果我们同样关闭了 Node 2 ，我们的程序 依然 可以保持在不丢任何数据的情况下运行，因为 Node 3 为每一个分片都保留着一份副本。

如果我们重新启动 Node 1 ，集群可以将缺失的副本分片再次进行分配，那么集群的状态也将如[图 5 “将参数 number\_of\_replicas 调大到 2”](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_scale_horizontally.html#cluster-three-nodes-two-replicas)所示。 如果 Node 1 依然拥有着之前的分片，它将尝试去重用它们，同时仅从主分片复制发生了修改的数据文件。

到目前为止，你应该对分片如何使得 Elasticsearch 进行水平扩容以及数据保障等知识有了一定了解。 接下来我们将讲述关于分片生命周期的更多细节。

## 数据输入和输出

无论我们写什么样的程序，目的都是一样的：以某种方式组织数据服务我们的目的。 但是数据不仅仅由随机位和字节组成。我们建立数据元素之间的关系以便于表示实体，或者现实世界中存在的 事物 。 如果我们知道一个名字和电子邮件地址属于同一个人，那么它们将会更有意义。

尽管在现实世界中，不是所有的类型相同的实体看起来都是一样的。 一个人可能有一个家庭电话号码，而另一个人只有一个手机号码，再一个人可能两者兼有。 一个人可能有三个电子邮件地址，而另一个人却一个都没有。一位西班牙人可能有两个姓，而讲英语的人可能只有一个姓。

面向对象编程语言如此流行的原因之一是对象帮我们表示和处理现实世界具有潜在的复杂的数据结构的实体，到目前为止，一切都很完美！

但是当我们需要存储这些实体时问题来了，传统上，我们以行和列的形式存储数据到关系型数据库中，相当于使用电子表格。 正因为我们使用了这种不灵活的存储媒介导致所有我们使用对象的灵活性都丢失了。

但是否我们可以将我们的对象按对象的方式来存储？ 这样我们就能更加专注于 使用 数据，而不是在电子表格的局限性下对我们的应用建模。 我们可以重新利用对象的灵活性。

一个 对象 是基于特定语言的内存的数据结构。 为了通过网络发送或者存储它，我们需要将它表示成某种标准的格式。 [JSON](http://en.wikipedia.org/wiki/Json) 是一种以人可读的文本表示对象的方法。 它已经变成 NoSQL 世界交换数据的事实标准。当一个对象被序列化成为 JSON，它被称为一个 JSON 文档 。

Elastcisearch 是分布式的 文档 存储。它能存储和检索复杂的数据结构--序列化成为JSON文档--以 实时 的方式。 换句话说，一旦一个文档被存储在 Elasticsearch 中，它就是可以被集群中的任意节点检索到。

当然，我们不仅要存储数据，我们一定还需要查询它，成批且快速的查询它们。 尽管现存的 NoSQL 解决方案允许我们以文档的形式存储对象，但是他们仍旧需要我们思考如何查询我们的数据，以及确定哪些字段需要被索引以加快数据检索。

在 Elasticsearch 中， 每个字段的所有数据 都是 默认被索引的 。 即每个字段都有为了快速检索设置的专用倒排索引。而且，不像其他多数的数据库，它能在 相同的查询中 使用所有这些倒排索引，并以惊人的速度返回结果。

在本章中，我们展示了用来创建，检索，更新和删除文档的 API。就目前而言，我们不关心文档中的数据或者怎样查询它们。 所有我们关心的就是在 Elasticsearch 中怎样安全的存储文档，以及如何将文档再次返回。

### 什么是文档?

在大多数应用中，多数实体或对象可以被序列化为包含键值对的 JSON 对象。 一个 键 可以是一个字段或字段的名称，一个 值 可以是一个字符串，一个数字，一个布尔值， 另一个对象，一些数组值，或一些其它特殊类型诸如表示日期的字符串，或代表一个地理位置的对象：

{

"name": "John Smith",

"age": 42,

"confirmed": true,

"join\_date": "2014-06-01",

"home": {

"lat": 51.5,

"lon": 0.1

},

"accounts": [

{

"type": "facebook",

"id": "johnsmith"

},

{

"type": "twitter",

"id": "johnsmith"

}

]

}

通常情况下，我们使用的术语 对象 和 文档 是可以互相替换的。不过，有一个区别： 一个对象仅仅是类似于 hash 、 hashmap 、字典或者关联数组的 JSON 对象，对象中也可以嵌套其他的对象。 对象可能包含了另外一些对象。在 Elasticsearch 中，术语 文档 有着特定的含义。它是指最顶层或者根对象, 这个根对象被序列化成 JSON 并存储到 Elasticsearch 中，指定了唯一 ID。

警告

字段的名字可以是任何合法的字符串，但不可以包含时间段。

### 文档元数据

一个文档不仅仅包含它的数据 ，也包含 元数据 —— 有关 文档的信息。 三个必须的元数据元素如下：

\_index

文档在哪存放

\_type

文档表示的对象类别

\_id

文档唯一标识

**\_index**

一个 索引 应该是因共同的特性被分组到一起的文档集合。 例如，你可能存储所有的产品在索引 products中，而存储所有销售的交易到索引 sales 中。 虽然也允许存储不相关的数据到一个索引中，但这通常看作是一个反模式的做法。

提示

实际上，在 Elasticsearch 中，我们的数据是被存储和索引在 分片 中，而一个索引仅仅是逻辑上的命名空间， 这个命名空间由一个或者多个分片组合在一起。 然而，这是一个内部细节，我们的应用程序根本不应该关心分片，对于应用程序而言，只需知道文档位于一个 索引内。  Elasticsearch 会处理所有的细节。

我们将在 [索引管理](https://www.elastic.co/guide/cn/elasticsearch/guide/current/index-management.html) 介绍如何自行创建和管理索引，但现在我们将让 Elasticsearch 帮我们创建索引。 所有需要我们做的就是选择一个索引名，这个名字必须小写，不能以下划线开头，不能包含逗号。我们用 website 作为索引名举例。

**\_type**

数据可能在索引中只是松散的组合在一起，但是通常明确定义一些数据中的子分区是很有用的。 例如，所有的产品都放在一个索引中，但是你有许多不同的产品类别，比如 "electronics" 、 "kitchen" 和 "lawn-care"。

这些文档共享一种相同的（或非常相似）的模式：他们有一个标题、描述、产品代码和价格。他们只是正好属于“产品”下的一些子类。

Elasticsearch 公开了一个称为 types （类型）的特性，它允许您在索引中对数据进行逻辑分区。不同 types 的文档可能有不同的字段，但最好能够非常相似。 我们将在 [类型和映射](https://www.elastic.co/guide/cn/elasticsearch/guide/current/mapping.html) 中更多的讨论关于 types 的一些应用和限制。

一个 \_type 命名可以是大写或者小写，但是不能以下划线或者句号开头，不应该包含逗号， 并且长度限制为256个字符. 我们使用 blog 作为类型名举例。

**\_id**

ID 是一个字符串， 当它和 \_index 以及 \_type 组合就可以唯一确定 Elasticsearch 中的一个文档。 当你创建一个新的文档，要么提供自己的 \_id ，要么让 Elasticsearch 帮你生成。

**其他元数据**

还有一些其他的元数据元素，他们在 [类型和映射](https://www.elastic.co/guide/cn/elasticsearch/guide/current/mapping.html) 进行了介绍。通过前面已经列出的元数据元素， 我们已经能存储文档到 Elasticsearch 中并通过 ID 检索它--换句话说，使用 Elasticsearch 作为文档的存储介质。

### 索引文档

通过使用 index API ，文档可以被 索引 —— 存储和使文档可被搜索 。 但是首先，我们要确定文档的位置。正如我们刚刚讨论的，一个文档的 \_index 、 \_type 和 \_id 唯一标识一个文档。 我们可以提供自定义的 \_id 值，或者让 index API 自动生成。

**使用自定义的 ID**

如果你的文档有一个自然的 标识符 （例如，一个 user\_account 字段或其他标识文档的值），你应该使用如下方式的 index API 并提供你自己 \_id ：

PUT /{index}/{type}/{id}

{

"field": "value",

...

}

举个例子，如果我们的索引称为 website ，类型称为 blog ，并且选择 123 作为 ID ，那么索引请求应该是下面这样：

PUT /website/blog/123

{

"title": "My first blog entry",

"text": "Just trying this out...",

"date": "2014/01/01"

}

Elasticsearch 响应体如下所示：

{

"\_index": "website",

"\_type": "blog",

"\_id": "123",

"\_version": 1,

"created": true

}

该响应表明文档已经成功创建，该索引包括 \_index 、 \_type 和 \_id 元数据， 以及一个新元素： \_version 。

在 Elasticsearch 中每个文档都有一个版本号。当每次对文档进行修改时（包括删除）， \_version 的值会递增。 在 [处理冲突](https://www.elastic.co/guide/cn/elasticsearch/guide/current/version-control.html) 中，我们讨论了怎样使用 \_version 号码确保你的应用程序中的一部分修改不会覆盖另一部分所做的修改。

**Autogenerating IDs**

如果你的数据没有自然的 ID， Elasticsearch 可以帮我们自动生成 ID 。 请求的结构调整为： 不再使用PUT 谓词(“使用这个 URL 存储这个文档”)， 而是使用 POST 谓词(“存储文档在这个 URL 命名空间下”)。

现在该 URL 只需包含 \_index 和 \_type :

POST /website/blog/

{

"title": "My second blog entry",

"text": "Still trying this out...",

"date": "2014/01/01"

}

除了 \_id 是 Elasticsearch 自动生成的，响应的其他部分和前面的类似：

{

"\_index": "website",

"\_type": "blog",

"\_id": "AVFgSgVHUP18jI2wRx0w",

"\_version": 1,

"created": true

}

自动生成的 ID 是 URL-safe、 基于 Base64 编码且长度为20个字符的 GUID 字符串。 这些 GUID 字符串由可修改的 FlakeID 模式生成，这种模式允许多个节点并行生成唯一 ID ，且互相之间的冲突概率几乎为零。

### 取回一个文档

为了从 Elasticsearch 中检索出文档 ，我们仍然使用相同的 \_index , \_type , 和 \_id ，但是 HTTP 谓词 更改为 GET :

GET /website/blog/123?pretty

响应体包括目前已经熟悉了的元数据元素，再加上 \_source 字段，这个字段包含我们索引数据时发送给 Elasticsearch 的原始 JSON 文档：

{

"\_index" : "website",

"\_type" : "blog",

"\_id" : "123",

"\_version" : 1,

"found" : true,

"\_source" : {

"title": "My first blog entry",

"text": "Just trying this out...",

"date": "2014/01/01"

}

}

注意

在请求的查询串参数中加上 pretty 参数， 正如前面的例子中看到的，这将会调用 Elasticsearch 的 pretty-print 功能，该功能 使得 JSON 响应体更加可读。但是， \_source字段不能被格式化打印出来。相反，我们得到的 \_source 字段中的 JSON 串，刚好是和我们传给它的一样。

GET 请求的响应体包括 {"found": true} ，这证实了文档已经被找到。 如果我们请求一个不存在的文档，我们仍旧会得到一个 JSON 响应体，但是 found 将会是 false 。 此外， HTTP 响应码将会是 404 Not Found ，而不是 200 OK 。

我们可以通过传递 -i 参数给 curl 命令，该参数 能够显示响应的头部：

curl -i -XGET http://localhost:9200/website/blog/124?pretty

显示响应头部的响应体现在类似这样：

HTTP/1.1 404 Not Found

Content-Type: application/json; charset=UTF-8

Content-Length: 83

{

"\_index" : "website",

"\_type" : "blog",

"\_id" : "124",

"found" : false

}

**返回文档的一部分**

默认情况下， GET 请求 会返回整个文档，这个文档正如存储在 \_source 字段中的一样。但是也许你只对其中的 title 字段感兴趣。单个字段能用 \_source 参数请求得到，多个字段也能使用逗号分隔的列表来指定。

GET /website/blog/123?\_source=title,text

该 \_source 字段现在包含的只是我们请求的那些字段，并且已经将 date 字段过滤掉了。

{

"\_index" : "website",

"\_type" : "blog",

"\_id" : "123",

"\_version" : 1,

"found" : true,

"\_source" : {

"title": "My first blog entry" ,

"text": "Just trying this out..."

}

}

或者，如果你只想得到 \_source 字段，不需要任何元数据，你能使用 \_source 端点：

GET /website/blog/123/\_source

那么返回的的内容如下所示：

{

"title": "My first blog entry",

"text": "Just trying this out...",

"date": "2014/01/01"

}

### 检查文档是否存在

如果只想检查一个文档是否存在 --根本不想关心内容--那么用 HEAD 方法来代替 GET 方法。 HEAD 请求没有返回体，只返回一个 HTTP 请求报头：

curl -i -XHEAD http://localhost:9200/website/blog/123

如果文档存在， Elasticsearch 将返回一个 200 ok 的状态码：

HTTP/1.1 200 OK

Content-Type: text/plain; charset=UTF-8

Content-Length: 0

若文档不存在， Elasticsearch 将返回一个 404 Not Found 的状态码：

curl -i -XHEAD http://localhost:9200/website/blog/124

HTTP/1.1 404 Not Found

Content-Type: text/plain; charset=UTF-8

Content-Length: 0

当然，一个文档仅仅是在检查的时候不存在，并不意味着一毫秒之后它也不存在：也许同时正好另一个进程就创建了该文档。

### 更新整个文档

在 Elasticsearch 中文档是 *不可改变* 的，不能修改它们。 相反，如果想要更新现有的文档，需要 *重建索引*或者进行替换， 我们可以使用相同的 index API 进行实现，在 [索引文档](https://www.elastic.co/guide/cn/elasticsearch/guide/current/index-doc.html) 中已经进行了讨论。

PUT /website/blog/123

{

"title": "My first blog entry",

"text": "I am starting to get the hang of this...",

"date": "2014/01/02"

}

在响应体中，我们能看到 Elasticsearch 已经增加了 \_version 字段值：

{

"\_index" : "website",

"\_type" : "blog",

"\_id" : "123",

"\_version" : 2,

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

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/update-doc.html#CO10-1) | created 标志设置成 false ，是因为相同的索引、类型和 ID 的文档已经存在。 |

在内部，Elasticsearch 已将旧文档标记为已删除，并增加一个全新的文档。 尽管你不能再对旧版本的文档进行访问，但它并不会立即消失。当继续索引更多的数据，Elasticsearch 会在后台清理这些已删除文档。

在本章的后面部分，我们会介绍 update API, 这个 API 可以用于 [partial updates to a document](https://www.elastic.co/guide/cn/elasticsearch/guide/current/partial-updates.html) 。 虽然它似乎对文档直接进行了修改，但实际上 Elasticsearch 按前述完全相同方式执行以下过程：

1. 从旧文档构建 JSON
2. 更改该 JSON
3. 删除旧文档
4. 索引一个新文档

唯一的区别在于, update API 仅仅通过一个客户端请求来实现这些步骤，而不需要单独的 get 和 index 请求。

### 创建新文档

当我们索引一个文档， 怎么确认我们正在创建一个完全新的文档，而不是覆盖现有的呢？

请记住， \_index 、 \_type 和 \_id 的组合可以唯一标识一个文档。所以，确保创建一个新文档的最简单办法是，使用索引请求的 POST 形式让 Elasticsearch 自动生成唯一 \_id :

POST /website/blog/

{ ... }

然而，如果已经有自己的 \_id ，那么我们必须告诉 Elasticsearch ，只有在相同的 \_index 、 \_type 和 \_id 不存在时才接受我们的索引请求。这里有两种方式，他们做的实际是相同的事情。使用哪种，取决于哪种使用起来更方便。

第一种方法使用 op\_type 查询 -字符串参数：

PUT /website/blog/123?op\_type=create

{ ... }

第二种方法是在 URL 末端使用 /\_create :

PUT /website/blog/123/\_create

{ ... }

如果创建新文档的请求成功执行，Elasticsearch 会返回元数据和一个 201 Created 的 HTTP 响应码。

另一方面，如果具有相同的 \_index 、 \_type 和 \_id 的文档已经存在，Elasticsearch 将会返回 409 Conflict 响应码，以及如下的错误信息：

{

"error": {

"root\_cause": [

{

"type": "document\_already\_exists\_exception",

"reason": "[blog][123]: document already exists",

"shard": "0",

"index": "website"

}

],

"type": "document\_already\_exists\_exception",

"reason": "[blog][123]: document already exists",

"shard": "0",

"index": "website"

},

"status": 409

}

### 删除文档

删除文档 的语法和我们所知道的规则相同，只是 使用 DELETE 方法：

DELETE /website/blog/123

**拷贝为 CURL**[在 SENSE 中查看](http://loca5601/app/sense/?load_from=https://www.elastic.co/guide/cn/elasticsearch/guide/current/snippets/030_Data/35_Delete_doc.json)

如果找到该文档，Elasticsearch 将要返回一个 200 ok 的 HTTP 响应码，和一个类似以下结构的响应体。注意，字段 \_version 值已经增加:

{

"found" : true,

"\_index" : "website",

"\_type" : "blog",

"\_id" : "123",

"\_version" : 3

}

如果文档没有 找到，我们将得到 404 Not Found 的响应码和类似这样的响应体：

{

"found" : false,

"\_index" : "website",

"\_type" : "blog",

"\_id" : "123",

"\_version" : 4

}

即使文档不存在（ Found 是 false ）， \_version 值仍然会增加。这是 Elasticsearch 内部记录本的一部分，用来确保这些改变在跨多节点时以正确的顺序执行。

注意

正如已经在[更新整个文档](https://www.elastic.co/guide/cn/elasticsearch/guide/current/update-doc.html)中提到的，删除文档不会立即将文档从磁盘中删除，只是将文档标记为已删除状态。随着你不断的索引更多的数据，Elasticsearch 将会在后台清理标记为已删除的文档。

### 处理冲突

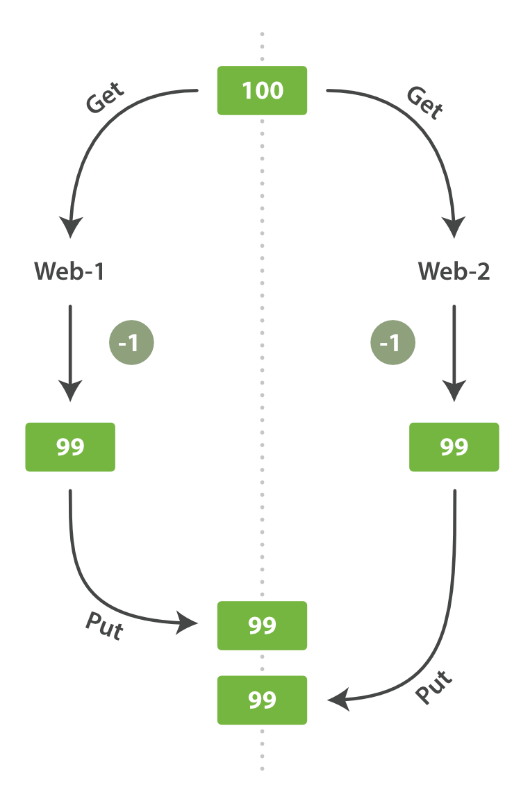
当我们使用 index API 更新文档 ，可以一次性读取原始文档，做我们的修改，然后重新索引 *整个文档* 。 最近的索引请求将获胜：无论最后哪一个文档被索引，都将被唯一存储在 Elasticsearch 中。如果其他人同时更改这个文档，他们的更改将丢失。

很多时候这是没有问题的。也许我们的主数据存储是一个关系型数据库，我们只是将数据复制到 Elasticsearch 中并使其可被搜索。 也许两个人同时更改相同的文档的几率很小。或者对于我们的业务来说偶尔丢失更改并不是很严重的问题。

但有时丢失了一个变更就是 *非常严重的* 。试想我们使用 Elasticsearch 存储我们网上商城商品库存的数量， 每次我们卖一个商品的时候，我们在 Elasticsearch 中将库存数量减少。

有一天，管理层决定做一次促销。突然地，我们一秒要卖好几个商品。 假设有两个 web 程序并行运行，每一个都同时处理所有商品的销售，如图 [图 7 “Consequence of no concurrency control”](https://www.elastic.co/guide/cn/elasticsearch/guide/current/version-control.html#img-data-lww) 所示。

**图 7. Consequence of no concurrency control**



web\_1 对 stock\_count 所做的更改已经丢失，因为 web\_2 不知道它的 stock\_count 的拷贝已经过期。 结果我们会认为有超过商品的实际数量的库存，因为卖给顾客的库存商品并不存在，我们将让他们非常失望。

变更越频繁，读数据和更新数据的间隙越长，也就越可能丢失变更。

在数据库领域中，有两种方法通常被用来确保并发更新时变更不会丢失：

*悲观并发控制*

这种方法被关系型数据库广泛使用，它假定有变更冲突可能发生，因此阻塞访问资源以防止冲突。 一个典型的例子是读取一行数据之前先将其锁住，确保只有放置锁的线程能够对这行数据进行修改。

*乐观并发控制*

Elasticsearch 中使用的这种方法假定冲突是不可能发生的，并且不会阻塞正在尝试的操作。 然而，如果源数据在读写当中被修改，更新将会失败。应用程序接下来将决定该如何解决冲突。 例如，可以重试更新、使用新的数据、或者将相关情况报告给用户。

### 乐观并发控制

Elasticsearch 是分布式的。当文档创建、更新或删除时， 新版本的文档必须复制到集群中的其他节点。Elasticsearch 也是异步和并发的，这意味着这些复制请求被并行发送，并且到达目的地时也许 顺序是乱的。  Elasticsearch 需要一种方法确保文档的旧版本不会覆盖新的版本。

当我们之前讨论 index ， GET 和 delete 请求时，我们指出每个文档都有一个 \_version （版本）号，当文档被修改时版本号递增。 Elasticsearch 使用这个 \_version 号来确保变更以正确顺序得到执行。如果旧版本的文档在新版本之后到达，它可以被简单的忽略。

我们可以利用 \_version 号来确保 应用中相互冲突的变更不会导致数据丢失。我们通过指定想要修改文档的 version 号来达到这个目的。 如果该版本不是当前版本号，我们的请求将会失败。

让我们创建一个新的博客文章：

PUT /website/blog/1/\_create

{

"title": "My first blog entry",

"text": "Just trying this out..."

}

响应体告诉我们，这个新创建的文档 \_version 版本号是 1 。现在假设我们想编辑这个文档：我们加载其数据到 web 表单中， 做一些修改，然后保存新的版本。

首先我们检索文档:

GET /website/blog/1

响应体包含相同的 \_version 版本号 1 ：

{

"\_index" : "website",

"\_type" : "blog",

"\_id" : "1",

"\_version" : 1,

"found" : true,

"\_source" : {

"title": "My first blog entry",

"text": "Just trying this out..."

}

}

现在，当我们尝试通过重建文档的索引来保存修改，我们指定 version 为我们的修改会被应用的版本：

PUT /website/blog/1?version=1 https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

{

"title": "My first blog entry",

"text": "Starting to get the hang of this..."

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/optimistic-concurrency-control.html#CO11-1) | 我们想这个在我们索引中的文档只有现在的 \_version 为 1 时，本次更新才能成功。 |

此请求成功，并且响应体告诉我们 \_version 已经递增到 2 ：

{

"\_index": "website",

"\_type": "blog",

"\_id": "1",

"\_version": 2

"created": false

}

然而，如果我们重新运行相同的索引请求，仍然指定 version=1 ， Elasticsearch 返回 409 ConflictHTTP 响应码，和一个如下所示的响应体：

{

"error": {

"root\_cause": [

{

"type": "version\_conflict\_engine\_exception",

"reason": "[blog][1]: version conflict, current [2], provided [1]",

"index": "website",

"shard": "3"

}

],

"type": "version\_conflict\_engine\_exception",

"reason": "[blog][1]: version conflict, current [2], provided [1]",

"index": "website",

"shard": "3"

},

"status": 409

}

这告诉我们在 Elasticsearch 中这个文档的当前 \_version 号是 2 ，但我们指定的更新版本号为 1 。

我们现在怎么做取决于我们的应用需求。我们可以告诉用户说其他人已经修改了文档，并且在再次保存之前检查这些修改内容。 或者，在之前的商品 stock\_count 场景，我们可以获取到最新的文档并尝试重新应用这些修改。

所有文档的更新或删除 API，都可以接受 version 参数，这允许你在代码中使用乐观的并发控制，这是一种明智的做法。

**通过外部系统使用版本控制**

一个常见的设置是使用其它数据库作为主要的数据存储，使用 Elasticsearch 做数据检索， 这意味着主数据库的所有更改发生时都需要被复制到 Elasticsearch ，如果多个进程负责这一数据同步，你可能遇到类似于之前描述的并发问题。

如果你的主数据库已经有了版本号 — 或一个能作为版本号的字段值比如 timestamp — 那么你就可以在 Elasticsearch 中通过增加 version\_type=external 到查询字符串的方式重用这些相同的版本号， 版本号必须是大于零的整数， 且小于 9.2E+18 — 一个 Java 中 long 类型的正值。

外部版本号的处理方式和我们之前讨论的内部版本号的处理方式有些不同， Elasticsearch 不是检查当前 \_version 和请求中指定的版本号是否相同， 而是检查当前 \_version 是否 小于 指定的版本号。 如果请求成功，外部的版本号作为文档的新 \_version 进行存储。

外部版本号不仅在索引和删除请求是可以指定，而且在 创建 新文档时也可以指定。

例如，要创建一个新的具有外部版本号 5 的博客文章，我们可以按以下方法进行：

PUT /website/blog/2?version=5&version\_type=external

{

"title": "My first external blog entry",

"text": "Starting to get the hang of this..."

}

在响应中，我们能看到当前的 \_version 版本号是 5 ：

{

"\_index": "website",

"\_type": "blog",

"\_id": "2",

"\_version": 5,

"created": true

}

现在我们更新这个文档，指定一个新的 version 号是 10 ：

PUT /website/blog/2?version=10&version\_type=external

{

"title": "My first external blog entry",

"text": "This is a piece of cake..."

}

请求成功并将当前 \_version 设为 10 ：

{

"\_index": "website",

"\_type": "blog",

"\_id": "2",

"\_version": 10,

"created": false

}

如果你要重新运行此请求时，它将会失败，并返回像我们之前看到的同样的冲突错误， 因为指定的外部版本号不大于 Elasticsearch 的当前版本号。

### 文档的部分更新

在 [更新整个文档](https://www.elastic.co/guide/cn/elasticsearch/guide/current/update-doc.html) , 我们已经介绍过 更新一个文档的方法是检索并修改它，然后重新索引整个文档，这的确如此。然而，使用 update API 我们还可以部分更新文档，例如在某个请求时对计数器进行累加。

我们也介绍过文档是不可变的：他们不能被修改，只能被替换。 update API 必须遵循同样的规则。 从外部来看，我们在一个文档的某个位置进行部分更新。然而在内部， update API 简单使用与之前描述相同的 检索-修改-重建索引 的处理过程。 区别在于这个过程发生在分片内部，这样就避免了多次请求的网络开销。通过减少检索和重建索引步骤之间的时间，我们也减少了其他进程的变更带来冲突的可能性。

update 请求最简单的一种形式是接收文档的一部分作为 doc 的参数， 它只是与现有的文档进行合并。对象被合并到一起，覆盖现有的字段，增加新的字段。 例如，我们增加字段 tags 和 views 到我们的博客文章，如下所示：

POST /website/blog/1/\_update

{

"doc" : {

"tags" : [ "testing" ],

"views": 0

}

}

如果请求成功，我们看到类似于 index 请求的响应：

{

"\_index" : "website",

"\_id" : "1",

"\_type" : "blog",

"\_version" : 3

}

检索文档显示了更新后的 \_source 字段：

{

"\_index": "website",

"\_type": "blog",

"\_id": "1",

"\_version": 3,

"found": true,

"\_source": {

"title": "My first blog entry",

"text": "Starting to get the hang of this...",

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

"views": 0 https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/partial-updates.html#CO12-1) [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/partial-updates.html#CO12-2) | 新的字段已被添加到 \_source 中。 |

**使用脚本部分更新文档**

脚本可以在 update API中用来改变 \_source 的字段内容， 它在更新脚本中称为 ctx.\_source 。 例如，我们可以使用脚本来增加博客文章中 views 的数量：

POST /website/blog/1/\_update

{

"script" : "ctx.\_source.views+=1"

}

**用 Groovy 脚本编程**

对于那些 API 不能满足需求的情况，Elasticsearch 允许你使用脚本编写自定义的逻辑。 许多API都支持脚本的使用，包括搜索、排序、聚合和文档更新。 脚本可以作为请求的一部分被传递，从特殊的 .scripts 索引中检索，或者从磁盘加载脚本。

默认的脚本语言 是 [Groovy](http://groovy.codehaus.org/)，一种快速表达的脚本语言，在语法上与 JavaScript 类似。 它在 Elasticsearch V1.3.0 版本首次引入并运行在 沙盒 中，然而 Groovy 脚本引擎存在漏洞， 允许攻击者通过构建 Groovy 脚本，在 Elasticsearch Java VM 运行时脱离沙盒并执行 shell 命令。

因此，在版本 v1.3.8 、 1.4.3 和 V1.5.0 及更高的版本中，它已经被默认禁用。 此外，您可以通过设置集群中的所有节点的 config/elasticsearch.yml 文件来禁用动态 Groovy 脚本：

script.groovy.sandbox.enabled: false

这将关闭 Groovy 沙盒，从而防止动态 Groovy 脚本作为请求的一部分被接受， 或者从特殊的 .scripts 索引中被检索。当然，你仍然可以使用存储在每个节点的 config/scripts/ 目录下的 Groovy 脚本。

如果你的架构和安全性不需要担心漏洞攻击，例如你的 Elasticsearch 终端仅暴露和提供给可信赖的应用， 当它是你的应用需要的特性时，你可以选择重新启用动态脚本。

你可以在 [scripting reference documentation](https://www.elastic.co/guide/en/elasticsearch/reference/master/modules-scripting.html) 获取更多关于脚本的资料。

我们也可以通过使用脚本给 tags 数组添加一个新的标签。在这个例子中，我们指定新的标签作为参数，而不是硬编码到脚本内部。 这使得 Elasticsearch 可以重用这个脚本，而不是每次我们想添加标签时都要对新脚本重新编译：

POST /website/blog/1/\_update

{

"script" : "ctx.\_source.tags+=new\_tag",

"params" : {

"new\_tag" : "search"

}

}

获取文档并显示最后两次请求的效果：

{

"\_index": "website",

"\_type": "blog",

"\_id": "1",

"\_version": 5,

"found": true,

"\_source": {

"title": "My first blog entry",

"text": "Starting to get the hang of this...",

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

"views": 1 https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/partial-updates.html#CO13-1) | search 标签已追加到 tags 数组中。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/partial-updates.html#CO13-2) | views 字段已递增。 |

我们甚至可以选择通过设置 ctx.op 为 delete 来删除基于其内容的文档：

POST /website/blog/1/\_update

{

"script" : "ctx.op = ctx.\_source.views == count ? 'delete' : 'none'",

"params" : {

"count": 1

}

}

**更新的文档可能尚不存在**

假设我们需要 在 Elasticsearch 中存储一个页面访问量计数器。 每当有用户浏览网页，我们对该页面的计数器进行累加。但是，如果它是一个新网页，我们不能确定计数器已经存在。 如果我们尝试更新一个不存在的文档，那么更新操作将会失败。

在这样的情况下，我们可以使用 upsert 参数，指定如果文档不存在就应该先创建它：

POST /website/pageviews/1/\_update

{

"script" : "ctx.\_source.views+=1",

"upsert": {

"views": 1

}

}

我们第一次运行这个请求时， upsert 值作为新文档被索引，初始化 views 字段为 1 。 在后续的运行中，由于文档已经存在， script 更新操作将替代 upsert 进行应用，对 views 计数器进行累加。

**更新和冲突**

在本节的介绍中，我们说明 检索 和 重建索引 步骤的间隔越小，变更冲突的机会越小。 但是它并不能完全消除冲突的可能性。 还是有可能在 update 设法重新索引之前，来自另一进程的请求修改了文档。

为了避免数据丢失， update API 在 检索 步骤时检索得到文档当前的 \_version 号，并传递版本号到 重建索引 步骤的 index 请求。 如果另一个进程修改了处于检索和重新索引步骤之间的文档，那么 \_version 号将不匹配，更新请求将会失败。

对于部分更新的很多使用场景，文档已经被改变也没有关系。 例如，如果两个进程都对页面访问量计数器进行递增操作，它们发生的先后顺序其实不太重要； 如果冲突发生了，我们唯一需要做的就是尝试再次更新。

这可以通过 设置参数 retry\_on\_conflict 来自动完成， 这个参数规定了失败之前 update 应该重试的次数，它的默认值为 0 。

POST /website/pageviews/1/\_update?retry\_on\_conflict=5 https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

{

"script" : "ctx.\_source.views+=1",

"upsert": {

"views": 0

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/partial-updates.html#CO14-1) | 失败之前重试该更新5次。 |

在增量操作无关顺序的场景，例如递增计数器等这个方法十分有效，但是在其他情况下变更的顺序 是 非常重要的。 类似 [index API](https://www.elastic.co/guide/cn/elasticsearch/guide/current/index-doc.html) ， update API 默认采用 最终写入生效 的方案，但它也接受一个 version 参数来允许你使用 [optimistic concurrency control](https://www.elastic.co/guide/cn/elasticsearch/guide/current/optimistic-concurrency-control.html) 指定想要更新文档的版本。

### 取回多个文档

Elasticsearch 的速度已经很快了，但甚至能更快。 将多个请求合并成一个，避免单独处理每个请求花费的网络时延和开销。 如果你需要从 Elasticsearch 检索很多文档，那么使用 *multi-get* 或者 mget API 来将这些检索请求放在一个请求中，将比逐个文档请求更快地检索到全部文档。

mget API 要求有一个 docs 数组作为参数，每个 元素包含需要检索文档的元数据， 包括 \_index 、 \_type和 \_id 。如果你想检索一个或者多个特定的字段，那么你可以通过 \_source 参数来指定这些字段的名字：

GET /\_mget

{

"docs" : [

{

"\_index" : "website",

"\_type" : "blog",

"\_id" : 2

},

{

"\_index" : "website",

"\_type" : "pageviews",

"\_id" : 1,

"\_source": "views"

}

]

}

该响应体也包含一个 docs 数组 ， 对于每一个在请求中指定的文档，这个数组中都包含有一个对应的响应，且顺序与请求中的顺序相同。 其中的每一个响应都和使用单个 [get request](https://www.elastic.co/guide/cn/elasticsearch/guide/current/get-doc.html) 请求所得到的响应体相同：

{

"docs" : [

{

"\_index" : "website",

"\_id" : "2",

"\_type" : "blog",

"found" : true,

"\_source" : {

"text" : "This is a piece of cake...",

"title" : "My first external blog entry"

},

"\_version" : 10

},

{

"\_index" : "website",

"\_id" : "1",

"\_type" : "pageviews",

"found" : true,

"\_version" : 2,

"\_source" : {

"views" : 2

}

}

]

}

如果想检索的数据都在相同的 \_index 中（甚至相同的 \_type 中），则可以在 URL 中指定默认的 /\_index或者默认的 /\_index/\_type 。

你仍然可以通过单独请求覆盖这些值：

GET /website/blog/\_mget

{

"docs" : [

{ "\_id" : 2 },

{ "\_type" : "pageviews", "\_id" : 1 }

]

}

事实上，如果所有文档的 \_index 和 \_type 都是相同的，你可以只传一个 ids 数组，而不是整个 docs 数组：

GET /website/blog/\_mget

{

"ids" : [ "2", "1" ]

}

注意，我们请求的第二个文档是不存在的。我们指定类型为 blog ，但是文档 ID 1 的类型是 pageviews ，这个不存在的情况将在响应体中被报告：

{

"docs" : [

{

"\_index" : "website",

"\_type" : "blog",

"\_id" : "2",

"\_version" : 10,

"found" : true,

"\_source" : {

"title": "My first external blog entry",

"text": "This is a piece of cake..."

}

},

{

"\_index" : "website",

"\_type" : "blog",

"\_id" : "1",

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

}

]

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_Retrieving_Multiple_Documents.html#CO15-1) | 未找到该文档。 |

事实上第二个文档未能找到并不妨碍第一个文档被检索到。每个文档都是单独检索和报告的。

注意

即使有某个文档没有找到，上述请求的 HTTP 状态码仍然是 200 。事实上，即使请求 *没有* 找到任何文档，它的状态码依然是 200 --因为 mget 请求本身已经成功执行。 为了确定某个文档查找是成功或者失败，你需要检查 found 标记。

### 代价较小的批量操作

与 mget 可以使我们一次取回多个文档同样的方式， bulk API 允许在单个步骤中进行多次 create 、 index 、 update 或 delete 请求。 如果你需要索引一个数据流比如日志事件，它可以排队和索引数百或数千批次。

bulk 与其他的请求体格式稍有不同，如下所示：

{ action: { metadata }}\n

{ request body }\n

{ action: { metadata }}\n

{ request body }\n

...

这种格式类似一个有效的单行 JSON 文档 *流* ，它通过换行符(\n)连接到一起。注意两个要点：

* 每行一定要以换行符(\n)结尾， *包括最后一行* 。这些换行符被用作一个标记，可以有效分隔行。
* 这些行不能包含未转义的换行符，因为他们将会对解析造成干扰。这意味着这个 JSON *不* 能使用 pretty 参数打印。

提示

在 [为什么是有趣的格式？](https://www.elastic.co/guide/cn/elasticsearch/guide/current/distrib-multi-doc.html#bulk-format) 中， 我们解释为什么 bulk API 使用这种格式。

action/metadata 行指定 *哪一个文档* 做 *什么操作* 。

action 必须是以下选项之一:

create

如果文档不存在，那么就创建它。详情请见 [创建新文档](https://www.elastic.co/guide/cn/elasticsearch/guide/current/create-doc.html)。

index

创建一个新文档或者替换一个现有的文档。详情请见 [索引文档](https://www.elastic.co/guide/cn/elasticsearch/guide/current/index-doc.html) 和 [更新整个文档](https://www.elastic.co/guide/cn/elasticsearch/guide/current/update-doc.html)。

update

部分更新一个文档。详情请见 [文档的部分更新](https://www.elastic.co/guide/cn/elasticsearch/guide/current/partial-updates.html)。

delete

删除一个文档。详情请见 [删除文档](https://www.elastic.co/guide/cn/elasticsearch/guide/current/delete-doc.html)。

metadata 应该 指定被索引、创建、更新或者删除的文档的 \_index 、 \_type 和 \_id 。

例如，一个 delete 请求看起来是这样的：

{ "delete": { "\_index": "website", "\_type": "blog", "\_id": "123" }}

request body 行由文档的 \_source 本身组成--文档包含的字段和值。它是 index 和 create 操作所必需的，这是有道理的：你必须提供文档以索引。

它也是 update 操作所必需的，并且应该包含你传递给 update API 的相同请求体： doc 、 upsert 、 script 等等。 删除操作不需要 request body 行。

{ "create": { "\_index": "website", "\_type": "blog", "\_id": "123" }}

{ "title": "My first blog post" }

如果不指定 \_id ，将会自动生成一个 ID ：

{ "index": { "\_index": "website", "\_type": "blog" }}

{ "title": "My second blog post" }

为了把所有的操作组合在一起，一个完整的 bulk 请求 有以下形式:

POST /\_bulk

{ "delete": { "\_index": "website", "\_type": "blog", "\_id": "123" }} https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

{ "create": { "\_index": "website", "\_type": "blog", "\_id": "123" }}

{ "title": "My first blog post" }

{ "index": { "\_index": "website", "\_type": "blog" }}

{ "title": "My second blog post" }

{ "update": { "\_index": "website", "\_type": "blog", "\_id": "123", "\_retry\_on\_conflict" : 3} }

{ "doc" : {"title" : "My updated blog post"} } https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/bulk.html#CO16-1) | 请注意 delete 动作不能有请求体,它后面跟着的是另外一个操作。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/bulk.html#CO16-2) | 谨记最后一个换行符不要落下。 |

这个 Elasticsearch 响应包含 items 数组， 这个数组的内容是以请求的顺序列出来的每个请求的结果。

{

"took": 4,

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

"items": [

{ "delete": {

"\_index": "website",

"\_type": "blog",

"\_id": "123",

"\_version": 2,

"status": 200,

"found": true

}},

{ "create": {

"\_index": "website",

"\_type": "blog",

"\_id": "123",

"\_version": 3,

"status": 201

}},

{ "create": {

"\_index": "website",

"\_type": "blog",

"\_id": "EiwfApScQiiy7TIKFxRCTw",

"\_version": 1,

"status": 201

}},

{ "update": {

"\_index": "website",

"\_type": "blog",

"\_id": "123",

"\_version": 4,

"status": 200

}}

]

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/bulk.html#CO17-1) | 所有的子请求都成功完成。 |

每个子请求都是独立执行，因此某个子请求的失败不会对其他子请求的成功与否造成影响。 如果其中任何子请求失败，最顶层的 error 标志被设置为 true ，并且在相应的请求报告出错误明细：

POST /\_bulk

{ "create": { "\_index": "website", "\_type": "blog", "\_id": "123" }}

{ "title": "Cannot create - it already exists" }

{ "index": { "\_index": "website", "\_type": "blog", "\_id": "123" }}

{ "title": "But we can update it" }

在响应中，我们看到 create 文档 123 失败，因为它已经存在。但是随后的 index 请求，也是对文档 123操作，就成功了：

{

"took": 3,

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

"items": [

{ "create": {

"\_index": "website",

"\_type": "blog",

"\_id": "123",

"status": 409, https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

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

[[website][4] [blog][123]:

document already exists]"

}},

{ "index": {

"\_index": "website",

"\_type": "blog",

"\_id": "123",

"\_version": 5,

"status": 200 https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/4.png

}}

]

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/bulk.html#CO18-1) | 一个或者多个请求失败。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/bulk.html#CO18-2) | 这个请求的HTTP状态码报告为 409 CONFLICT 。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/bulk.html#CO18-3) | 解释为什么请求失败的错误信息。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/4.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/bulk.html#CO18-4) | 第二个请求成功，返回 HTTP 状态码 200 OK 。 |

这也意味着 bulk 请求不是原子的： 不能用它来实现事务控制。每个请求是单独处理的，因此一个请求的成功或失败不会影响其他的请求。

**不要重复指定Index和Type**

也许你正在批量索引日志数据到相同的 index 和 type 中。 但为每一个文档指定相同的元数据是一种浪费。相反，可以像 mget API 一样，在 bulk 请求的 URL 中接收默认的 /\_index 或者 /\_index/\_type ：

POST /website/\_bulk

{ "index": { "\_type": "log" }}

{ "event": "User logged in" }

你仍然可以覆盖元数据行中的 \_index 和 \_type , 但是它将使用 URL 中的这些元数据值作为默认值：

POST /website/log/\_bulk

{ "index": {}}

{ "event": "User logged in" }

{ "index": { "\_type": "blog" }}

{ "title": "Overriding the default type" }

**多大是太大了？**

整个批量请求都需要由接收到请求的节点加载到内存中，因此该请求越大，其他请求所能获得的内存就越少。 批量请求的大小有一个最佳值，大于这个值，性能将不再提升，甚至会下降。 但是最佳值不是一个固定的值。它完全取决于硬件、文档的大小和复杂度、索引和搜索的负载的整体情况。

幸运的是，很容易找到这个 *最佳点* ：通过批量索引典型文档，并不断增加批量大小进行尝试。 当性能开始下降，那么你的批量大小就太大了。一个好的办法是开始时将 1,000 到 5,000 个文档作为一个批次, 如果你的文档非常大，那么就减少批量的文档个数。

密切关注你的批量请求的物理大小往往非常有用，一千个 1KB 的文档是完全不同于一千个 1MB 文档所占的物理大小。 一个好的批量大小在开始处理后所占用的物理大小约为 5-15 MB。

## [分布式文档存储](https://www.elastic.co/guide/cn/elasticsearch/guide/current/distributed-docs.html)

### 路由一个文档到一个分片中

当索引一个文档的时候，文档会被存储到一个主分片中。 Elasticsearch 如何知道一个文档应该存放到哪个分片中呢？当我们创建文档时，它如何决定这个文档应当被存储在分片 1 还是分片 2 中呢？

首先这肯定不会是随机的，否则将来要获取文档的时候我们就不知道从何处寻找了。实际上，这个过程是根据下面这个公式决定的：

shard = hash(routing) % number\_of\_primary\_shards

routing 是一个可变值，默认是文档的 \_id ，也可以设置成一个自定义的值。 routing 通过 hash 函数生成一个数字，然后这个数字再除以 number\_of\_primary\_shards （主分片的数量）后得到 **余数** 。这个分布在 0 到 number\_of\_primary\_shards-1 之间的余数，就是我们所寻求的文档所在分片的位置。

这就解释了为什么我们要在创建索引的时候就确定好主分片的数量 并且永远不会改变这个数量：因为如果数量变化了，那么所有之前路由的值都会无效，文档也再也找不到了。

注意

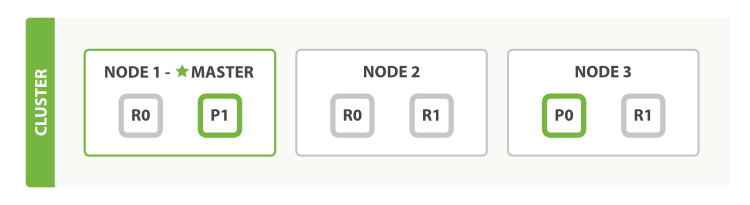
你可能觉得由于 Elasticsearch 主分片数量是固定的会使索引难以进行扩容。实际上当你需要时有很多技巧可以轻松实现扩容。我们将会在[*扩容设计*](https://www.elastic.co/guide/cn/elasticsearch/guide/current/scale.html)一章中提到更多有关水平扩展的内容。

所有的文档 API（ get 、 index 、 delete 、 bulk 、 update 以及 mget ）都接受一个叫做 routing 的路由参数 ，通过这个参数我们可以自定义文档到分片的映射。一个自定义的路由参数可以用来确保所有相关的文档——例如所有属于同一个用户的文档——都被存储到同一个分片中。我们也会在[*扩容设计*](https://www.elastic.co/guide/cn/elasticsearch/guide/current/scale.html)这一章中详细讨论为什么会有这样一种需求。

### 主分片和副本分片如何交互

为了说明目的, 我们 假设有一个集群由三个节点组成。 它包含一个叫 blogs 的索引，有两个主分片，每个主分片有两个副本分片。相同分片的副本不会放在同一节点，所以我们的集群看起来像 [图 8 “有三个节点和一个索引的集群”](https://www.elastic.co/guide/cn/elasticsearch/guide/current/how-primary-and-replica-shards-interact.html#img-distrib)。

**图 8. 有三个节点和一个索引的集群**



我们可以发送请求到集群中的任一节点。 每个节点都有能力处理任意请求。 每个节点都知道集群中任一文档位置，所以可以直接将请求转发到需要的节点上。 在下面的例子中，将所有的请求发送到 Node 1 ，我们将其称为 协调节点(coordinating node) 。

提示

当发送请求的时候， 为了扩展负载，更好的做法是轮询集群中所有的节点。

### 新建、索引和删除文档

新建、索引和删除 请求都是 写 操作， 必须在主分片上面完成之后才能被复制到相关的副本分片，如下图所示 [图 9 “新建、索引和删除单个文档”](https://www.elastic.co/guide/cn/elasticsearch/guide/current/distrib-write.html#img-distrib-write).

**图 9. 新建、索引和删除单个文档**



以下是在主副分片和任何副本分片上面 成功新建，索引和删除文档所需要的步骤顺序：

1. 客户端向 Node 1 发送新建、索引或者删除请求。
2. 节点使用文档的 \_id 确定文档属于分片 0 。请求会被转发到 Node 3`，因为分片 0 的主分片目前被分配在 `Node 3 上。
3. Node 3 在主分片上面执行请求。如果成功了，它将请求并行转发到 Node 1 和 Node 2 的副本分片上。一旦所有的副本分片都报告成功, Node 3 将向协调节点报告成功，协调节点向客户端报告成功。

在客户端收到成功响应时，文档变更已经在主分片和所有副本分片执行完成，变更是安全的。

有一些可选的请求参数允许您影响这个过程，可能以数据安全为代价提升性能。这些选项很少使用，因为Elasticsearch已经很快，但是为了完整起见，在这里阐述如下：

consistency

consistency，即一致性。在默认设置下，即使仅仅是在试图执行一个\_写\_操作之前，主分片都会要求必须要有 \_规定数量(quorum)\_（或者换种说法，也即必须要有大多数）的分片副本处于活跃可用状态，才会去执行\_写\_操作(其中分片副本可以是主分片或者副本分片)。这是为了避免在发生网络分区故障（network partition）的时候进行\_写\_操作，进而导致数据不一致。\_规定数量\_即：

int( (primary + number\_of\_replicas) / 2 ) + 1

consistency 参数的值可以设为 one （只要主分片状态 ok 就允许执行\_写\_操作）,all`（必须要主分片和所有副本分片的状态没问题才允许执行\_写\_操作）, 或 `quorum 。默认值为 quorum , 即大多数的分片副本状态没问题就允许执行\_写\_操作。

注意，规定数量 的计算公式中 number\_of\_replicas 指的是在索引设置中的设定副本分片数，而不是指当前处理活动状态的副本分片数。如果你的索引设置中指定了当前索引拥有三个副本分片，那规定数量的计算结果即：

int( (primary + 3 replicas) / 2 ) + 1 = 3

如果此时你只启动两个节点，那么处于活跃状态的分片副本数量就达不到规定数量，也因此您将无法索引和删除任何文档。

timeout

如果没有足够的副本分片会发生什么？ Elasticsearch会等待，希望更多的分片出现。默认情况下，它最多等待1分钟。 如果你需要，你可以使用 timeout 参数 使它更早终止： 100 100毫秒，30s 是30秒。

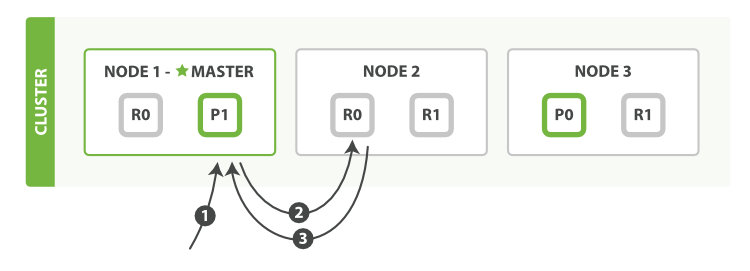
注意

新索引默认有 1 个副本分片，这意味着为满足 规定数量 应该 需要两个活动的分片副本。 但是，这些默认的设置会阻止我们在单一节点上做任何事情。为了避免这个问题，要求只有当 number\_of\_replicas 大于1的时候，规定数量才会执行。

### 取回一个文档

可以从主分片或者从其它任意副本分片检索文档 ，如下图所示 [图 10 “取回单个文档”](https://www.elastic.co/guide/cn/elasticsearch/guide/current/distrib-read.html#img-distrib-read).

**图 10. 取回单个文档**



以下是从主分片或者副本分片检索文档的步骤顺序：

1、客户端向 Node 1 发送获取请求。

2、节点使用文档的 \_id 来确定文档属于分片 0 。分片 0 的副本分片存在于所有的三个节点上。 在这种情况下，它将请求转发到 Node 2 。

3、Node 2 将文档返回给 Node 1 ，然后将文档返回给客户端。

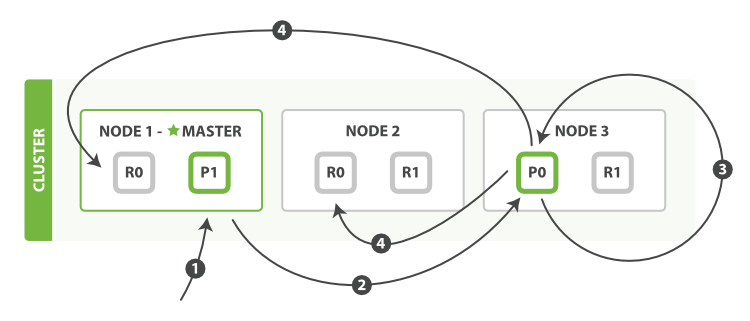
在处理读取请求时，协调结点在每次请求的时候都会通过轮询所有的副本分片来达到负载均衡。

在文档被检索时，已经被索引的文档可能已经存在于主分片上但是还没有复制到副本分片。 在这种情况下，副本分片可能会报告文档不存在，但是主分片可能成功返回文档。 一旦索引请求成功返回给用户，文档在主分片和副本分片都是可用的。

### 局部更新文档

如 [图 11 “局部更新文档”](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_partial_updates_to_a_document.html#img-distrib-update) 所示，update API 结合了先前说明的读取和写入模式 。

**图 11. 局部更新文档**



以下是部分更新一个文档的步骤：

1. 客户端向 Node 1 发送更新请求。
2. 它将请求转发到主分片所在的 Node 3 。
3. Node 3 从主分片检索文档，修改 \_source 字段中的 JSON ，并且尝试重新索引主分片的文档。 如果文档已经被另一个进程修改，它会重试步骤 3 ，超过 retry\_on\_conflict 次后放弃。
4. 如果 Node 3 成功地更新文档，它将新版本的文档并行转发到 Node 1 和 Node 2 上的副本分片，重新建立索引。 一旦所有副本分片都返回成功， Node 3 向协调节点也返回成功，协调节点向客户端返回成功。

update API 还接受在 [新建、索引和删除文档](https://www.elastic.co/guide/cn/elasticsearch/guide/current/distrib-write.html) 章节中介绍的 routing 、 replication 、 consistency 和 timeout 参数。

**基于文档的复制**

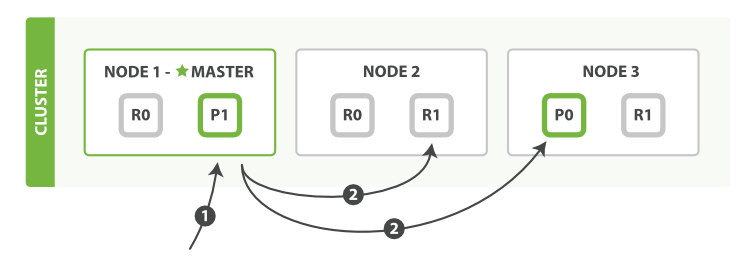
当主分片把更改转发到副本分片时， 它不会转发更新请求。 相反，它转发完整文档的新版本。请记住，这些更改将会异步转发到副本分片，并且不能保证它们以发送它们相同的顺序到达。 如果Elasticsearch仅转发更改请求，则可能以错误的顺序应用更改，导致得到损坏的文档。

### 多文档模式

mget 和 bulk API 的 模式类似于单文档模式。区别在于协调节点知道每个文档存在于哪个分片中。 它将整个多文档请求分解成 *每个分片* 的多文档请求，并且将这些请求并行转发到每个参与节点。

协调节点一旦收到来自每个节点的应答，就将每个节点的响应收集整理成单个响应，返回给客户端，如 [图 12 “使用 mget 取回多个文档”](https://www.elastic.co/guide/cn/elasticsearch/guide/current/distrib-multi-doc.html#img-distrib-mget) 所示。

**图 12. 使用 mget 取回多个文档**



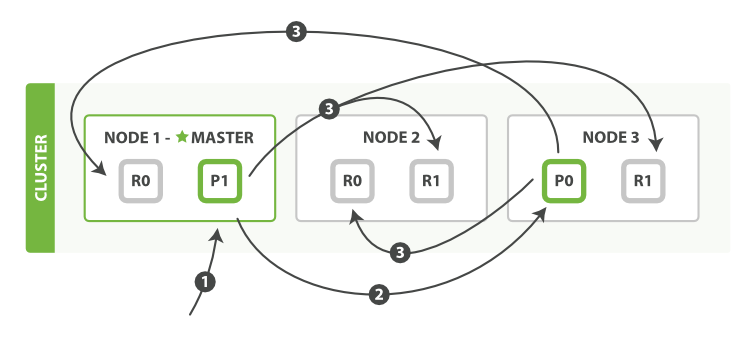
以下是使用单个 mget 请求取回多个文档所需的步骤顺序：

1. 客户端向 Node 1 发送 mget 请求。
2. Node 1 为每个分片构建多文档获取请求，然后并行转发这些请求到托管在每个所需的主分片或者副本分片的节点上。一旦收到所有答复， Node 1 构建响应并将其返回给客户端。

可以对 docs 数组中每个文档设置 routing 参数。

bulk API， 如 [图 13 “使用 bulk 修改多个文档”](https://www.elastic.co/guide/cn/elasticsearch/guide/current/distrib-multi-doc.html#img-distrib-bulk) 所示， 允许在单个批量请求中执行多个创建、索引、删除和更新请求。

**图 13. 使用 bulk 修改多个文档**



bulk API 按如下步骤顺序执行：

1. 客户端向 Node 1 发送 bulk 请求。
2. Node 1 为每个节点创建一个批量请求，并将这些请求并行转发到每个包含主分片的节点主机。
3. 主分片一个接一个按顺序执行每个操作。当每个操作成功时，主分片并行转发新文档（或删除）到副本分片，然后执行下一个操作。 一旦所有的副本分片报告所有操作成功，该节点将向协调节点报告成功，协调节点将这些响应收集整理并返回给客户端。

bulk API 还可以在整个批量请求的最顶层使用 consistency 参数，以及在每个请求中的元数据中使用 routing 参数。

**为什么是有趣的格式？**

当我们早些时候在[代价较小的批量操作](https://www.elastic.co/guide/cn/elasticsearch/guide/current/bulk.html)章节了解批量请求时， 您可能会问自己， "为什么 bulk API 需要有换行符的有趣格式，而不是发送包装在 JSON 数组中的请求，例如 mget API？" 。

为了回答这一点，我们需要解释一点背景：在批量请求中引用的每个文档可能属于不同的主分片， 每个文档可能被分配给集群中的任何节点。这意味着批量请求 bulk 中的每个 *操作* 都需要被转发到正确节点上的正确分片。

如果单个请求被包装在 JSON 数组中，那就意味着我们需要执行以下操作：

* 将 JSON 解析为数组（包括文档数据，可以非常大）
* 查看每个请求以确定应该去哪个分片
* 为每个分片创建一个请求数组
* 将这些数组序列化为内部传输格式
* 将请求发送到每个分片

这是可行的，但需要大量的 RAM 来存储原本相同的数据的副本，并将创建更多的数据结构，Java虚拟机（JVM）将不得不花费时间进行垃圾回收。

相反，Elasticsearch可以直接读取被网络缓冲区接收的原始数据。 它使用换行符字符来识别和解析小的 action/metadata 行来决定哪个分片应该处理每个请求。

这些原始请求会被直接转发到正确的分片。没有冗余的数据复制，没有浪费的数据结构。整个请求尽可能在最小的内存中处理。

## 搜索——最基本的工具

现在，我们已经学会了如何使用 Elasticsearch 作为一个简单的 NoSQL 风格的分布式文档存储系统。我们可以将一个 JSON 文档扔到 Elasticsearch 里，然后根据 ID 检索。但 Elasticsearch 真正强大之处在于可以从无规律的数据中找出有意义的信息——从“大数据”到“大信息”。

Elasticsearch 不只会*存储（stores）* 文档，为了能被搜索到也会为文档添加*索引（indexes）* ，这也是为什么我们使用结构化的 JSON 文档，而不是无结构的二进制数据。

*文档中的每个字段都将被索引并且可以被查询* 。不仅如此，在简单查询时，Elasticsearch 可以使用 *所有（all）* 这些索引字段，以惊人的速度返回结果。这是你永远不会考虑用传统数据库去做的一些事情。

*搜索（search）* 可以做到：

* 在类似于 gender 或者 age 这样的字段 上使用结构化查询，join\_date 这样的字段上使用排序，就像SQL的结构化查询一样。
* 全文检索，找出所有匹配关键字的文档并按照*相关性（relevance）* 排序后返回结果。
* 以上二者兼而有之。

很多搜索都是开箱即用的，为了充分挖掘 Elasticsearch 的潜力，你需要理解以下三个概念：

*映射（Mapping）*

描述数据在每个字段内如何存储

*分析（Analysis）*

全文是如何处理使之可以被搜索的

*领域特定查询语言（Query DSL）*

Elasticsearch 中强大灵活的查询语言

以上提到的每个点都是一个大话题，我们将在 [深入搜索](https://www.elastic.co/guide/cn/elasticsearch/guide/current/search-in-depth.html) 一章详细阐述它们。本章节我们将介绍这三点的一些基本概念——仅仅帮助你大致了解搜索是如何工作的。

我们将使用最简单的形式开始介绍 search API。

**测试数据**

本章节的测试数据可以在这里找到： <https://gist.github.com/clintongormley/8579281> 。

你可以把这些命令复制到终端中执行来实践本章的例子。

另外，如果你读的是在线版本，可以 [点击这个链接](https://www.elastic.co/guide/cn/elasticsearch/guide/current/sense_widget.html?snippets/050_Search/Test_data.json) 感受下。

### 空搜索

搜索API的最基础的形式是没有指定任何查询的空搜索 ，它简单地返回集群中所有索引下的所有文档：

GET /\_search

返回的结果（为了界面简洁编辑过的）像这样：

{

"hits" : {

"total" : 14,

"hits" : [

{

"\_index": "us",

"\_type": "tweet",

"\_id": "7",

"\_score": 1,

"\_source": {

"date": "2014-09-17",

"name": "John Smith",

"tweet": "The Query DSL is really powerful and flexible",

"user\_id": 2

}

},

... 9 RESULTS REMOVED ...

],

"max\_score" : 1

},

"took" : 4,

"\_shards" : {

"failed" : 0,

"successful" : 10,

"total" : 10

},

"timed\_out" : false

}

**hits**

返回结果中最重要的部分是 hits ，它 包含 total 字段来表示匹配到的文档总数，并且一个 hits 数组包含所查询结果的前十个文档。

在 hits 数组中每个结果包含文档的 \_index 、 \_type 、 \_id ，加上 \_source 字段。这意味着我们可以直接从返回的搜索结果中使用整个文档。这不像其他的搜索引擎，仅仅返回文档的ID，需要你单独去获取文档。

每个结果还有一个 \_score ，它衡量了文档与查询的匹配程度。默认情况下，首先返回最相关的文档结果，就是说，返回的文档是按照 \_score 降序排列的。在这个例子中，我们没有指定任何查询，故所有的文档具有相同的相关性，因此对所有的结果而言 1 是中性的 \_score 。

max\_score 值是与查询所匹配文档的 \_score 的最大值。

**took**

took 值告诉我们执行整个搜索请求耗费了多少毫秒。

**shards**

\_shards 部分 告诉我们在查询中参与分片的总数，以及这些分片成功了多少个失败了多少个。正常情况下我们不希望分片失败，但是分片失败是可能发生的。如果我们遭遇到一种灾难级别的故障，在这个故障中丢失了相同分片的原始数据和副本，那么对这个分片将没有可用副本来对搜索请求作出响应。假若这样，Elasticsearch 将报告这个分片是失败的，但是会继续返回剩余分片的结果。

**timeout**

timed\_out 值告诉我们查询是否超时。默认情况下，搜索请求不会超时。 如果低响应时间比完成结果更重要，你可以指定 timeout 为 10 或者 10ms（10毫秒），或者 1s（1秒）：

GET /\_search?timeout=10ms

在请求超时之前，Elasticsearch 将会返回已经成功从每个分片获取的结果。

警告

应当注意的是 timeout 不是停止执行查询，它仅仅是告知正在协调的节点返回到目前为止收集的结果并且关闭连接。在后台，其他的分片可能仍在执行查询即使是结果已经被发送了。

使用超时是因为 SLA(服务等级协议)对你是很重要的，而不是因为想去中止长时间运行的查询。

### 多索引，多类型

你有没有注意到之前的 [empty search](https://www.elastic.co/guide/cn/elasticsearch/guide/current/empty-search.html) 的结果，不同类型的文档 — user 和 tweet 来自不同的索引— us和 gb ？

如果不对某一特殊的索引或者类型做限制，就会搜索集群中的所有文档。Elasticsearch 转发搜索请求到每一个主分片或者副本分片，汇集查询出的前10个结果，并且返回给我们。

然而，经常的情况下，你 想在一个或多个特殊的索引并且在一个或者多个特殊的类型中进行搜索。我们可以通过在URL中指定特殊的索引和类型达到这种效果，如下所示：

/\_search

在所有的索引中搜索所有的类型

/gb/\_search

在 gb 索引中搜索所有的类型

/gb,us/\_search

在 gb 和 us 索引中搜索所有的文档

/g\*,u\*/\_search

在任何以 g 或者 u 开头的索引中搜索所有的类型

/gb/user/\_search

在 gb 索引中搜索 user 类型

/gb,us/user,tweet/\_search

在 gb 和 us 索引中搜索 user 和 tweet 类型

/\_all/user,tweet/\_search

在所有的索引中搜索 user 和 tweet 类型

当在单一的索引下进行搜索的时候，Elasticsearch 转发请求到索引的每个分片中，可以是主分片也可以是副本分片，然后从每个分片中收集结果。多索引搜索恰好也是用相同的方式工作的--只是会涉及到更多的分片。

提示

搜索一个索引有五个主分片和搜索五个索引各有一个分片准确来所说是等价的。

接下来，你将明白这种简单的方式如何灵活的根据需求的变化让扩容变得简单。

[«  空搜索](https://www.elastic.co/guide/cn/elasticsearch/guide/current/empty-search.html)  

### 分页

在之前的 [空搜索](https://www.elastic.co/guide/cn/elasticsearch/guide/current/empty-search.html) 中说明了集群中有 14 个文档匹配了（empty）query 。 但是在 hits 数组中只有 10 个文档。如何才能看到其他的文档？

和 SQL 使用 LIMIT 关键字返回单个 page 结果的方法相同，Elasticsearch 接受 from 和 size 参数：

size

显示应该返回的结果数量，默认是 10

from

显示应该跳过的初始结果数量，默认是 0

如果每页展示 5 条结果，可以用下面方式请求得到 1 到 3 页的结果：

GET /\_search?size=5

GET /\_search?size=5&from=5

GET /\_search?size=5&from=10

考虑到分页过深以及一次请求太多结果的情况，结果集在返回之前先进行排序。 但请记住一个请求经常跨越多个分片，每个分片都产生自己的排序结果，这些结果需要进行集中排序以保证整体顺序是正确的。

**在分布式系统中深度分页**

理解为什么深度分页是有问题的，我们可以假设在一个有 5 个主分片的索引中搜索。 当我们请求结果的第一页（结果从 1 到 10 ），每一个分片产生前 10 的结果，并且返回给 *协调节点* ，协调节点对 50 个结果排序得到全部结果的前 10 个。

现在假设我们请求第 1000 页--结果从 10001 到 10010 。所有都以相同的方式工作除了每个分片不得不产生前10010个结果以外。 然后协调节点对全部 50050 个结果排序最后丢弃掉这些结果中的 50040 个结果。

可以看到，在分布式系统中，对结果排序的成本随分页的深度成指数上升。这就是 web 搜索引擎对任何查询都不要返回超过 1000 个结果的原因。

提示

在 [重新索引你的数据](https://www.elastic.co/guide/cn/elasticsearch/guide/current/reindex.html) 中解释了如何 *能够* 有效获取大量的文档。

### *轻量* 搜索

有两种形式的 搜索 API：一种是 “轻量的” *查询字符串* 版本，要求在查询字符串中传递所有的 参数，另一种是更完整的 *请求体* 版本，要求使用 JSON 格式和更丰富的查询表达式作为搜索语言。

查询字符串搜索非常适用于通过命令行做即席查询。例如，查询在 tweet 类型中 tweet 字段包含 elasticsearch 单词的所有文档：

GET /\_all/tweet/\_search?q=tweet:elasticsearch

下一个查询在 name 字段中包含 john 并且在 tweet 字段中包含 mary 的文档。实际的查询就是这样

+name:john +tweet:mary

但是查询字符串参数所需要的 *百分比编码* （译者注：URL编码）实际上更加难懂：

GET /\_search?q=%2Bname%3Ajohn+%2Btweet%3Amary

+ 前缀表示必须与查询条件匹配。类似地， - 前缀表示一定不与查询条件匹配。没有 + 或者 - 的所有其他条件都是可选的——匹配的越多，文档就越相关。

**\_all 字段**

这个简单搜索返回包含 mary 的所有文档：

GET /\_search?q=mary

之前的例子中，我们在 tweet 和 name 字段中搜索内容。然而，这个查询的结果在三个地方提到了 mary：

* 有一个用户叫做 Mary
* 6条微博发自 Mary
* 一条微博直接 @mary

Elasticsearch 是如何在三个不同的字段中查找到结果的呢？

当索引一个文档的时候，Elasticsearch 取出所有字段的值拼接成一个大的字符串，作为 \_all 字段进行索引。例如，当索引这个文档时：

{

"tweet": "However did I manage before Elasticsearch?",

"date": "2014-09-14",

"name": "Mary Jones",

"user\_id": 1

}

这就好似增加了一个名叫 \_all 的额外字段：

"However did I manage before Elasticsearch? 2014-09-14 Mary Jones 1"

除非设置特定字段，否则查询字符串就使用 \_all 字段进行搜索。

提示

在刚开始开发一个应用时，\_all 字段是一个很实用的特性。之后，你会发现如果搜索时用指定字段来代替 \_all 字段，将会更好控制搜索结果。当 \_all 字段不再有用的时候，可以将它置为失效，正如在 [元数据: \_all 字段](https://www.elastic.co/guide/cn/elasticsearch/guide/current/root-object.html#all-field) 中所解释的。

**更复杂的查询**

下面的查询针对tweents类型，并使用以下的条件：

* name 字段中包含 mary 或者 john
* date 值大于 2014-09-10
* \_all\_ 字段包含 aggregations 或者 geo

+name:(mary john) +date:>2014-09-10 +(aggregations geo)

查询字符串在做了适当的编码后，可读性很差：

?q=%2Bname%3A(mary+john)+%2Bdate%3A%3E2014-09-10+%2B(aggregations+geo)

从之前的例子中可以看出，这种 *轻量* 的查询字符串搜索效果还是挺让人惊喜的。 它的查询语法在相关参考文档中有详细解释，以便简洁的表达很复杂的查询。对于通过命令做一次性查询，或者是在开发阶段，都非常方便。

但同时也可以看到，这种精简让调试更加晦涩和困难。而且很脆弱，一些查询字符串中很小的语法错误，像 - ， : ， / 或者 " 不匹配等，将会返回错误而不是搜索结果。

最后，查询字符串搜索允许任何用户在索引的任意字段上执行可能较慢且重量级的查询，这可能会暴露隐私信息，甚至将集群拖垮。

提示

因为这些原因，不推荐直接向用户暴露查询字符串搜索功能，除非对于集群和数据来说非常信任他们。

相反，我们经常在生产环境中更多地使用功能全面的 *request body* 查询API，除了能完成以上所有功能，还有一些附加功能。但在到达那个阶段之前，首先需要了解数据在 Elasticsearch 中是如何被索引的。

## 映射和分析

当摆弄索引里面的数据时，我们发现一些奇怪的事情。一些事情看起来被打乱了：在我们的索引中有12条推文，其中只有一条包含日期 2014-09-15 ，但是看一看下面查询命中的 总数 （total）：

GET /\_search?q=2014 # 12 results

GET /\_search?q=2014-09-15 # 12 results !

GET /\_search?q=date:2014-09-15 # 1 result

GET /\_search?q=date:2014 # 0 results !

为什么在 [\_all](https://www.elastic.co/guide/cn/elasticsearch/guide/current/search-lite.html#all-field-intro) 字段查询日期返回所有推文，而在 date 字段只查询年份却没有返回结果？为什么我们在 \_all 字段和 date 字段的查询结果有差别？

推测起来，这是因为数据在 \_all 字段与 date 字段的索引方式不同。所以，通过请求 gb 索引中 tweet类型的\_映射\_（或模式定义），让我们看一看 Elasticsearch 是如何解释我们文档结构的：

GET /gb/\_mapping/tweet

这将得到如下结果：

{

"gb": {

"mappings": {

"tweet": {

"properties": {

"date": {

"type": "date",

"format": "strict\_date\_optional\_time||epoch\_millis"

},

"name": {

"type": "string"

},

"tweet": {

"type": "string"

},

"user\_id": {

"type": "long"

}

}

}

}

}

}

基于对字段类型的猜测， Elasticsearch 动态为我们产生了一个映射。这个响应告诉我们 date 字段被认为是 date 类型的。由于 \_all 是默认字段，所以没有提及它。但是我们知道 \_all 字段是 string 类型的。

所以 date 字段和 string 字段 索引方式不同，因此搜索结果也不一样。这完全不令人吃惊。你可能会认为 核心数据类型 strings、numbers、Booleans 和 dates 的索引方式有稍许不同。没错，他们确实稍有不同。

但是，到目前为止，最大的差异在于 代表 *精确值* （它包括 string 字段）的字段和代表 *全文* 的字段。这个区别非常重要——它将搜索引擎和所有其他数据库区别开来。

### 精确值 VS 全文

Elasticsearch 中的数据可以概括的分为两类：精确值和全文。

*精确值* 如它们听起来那样精确。例如日期或者用户 ID，但字符串也可以表示精确值，例如用户名或邮箱地址。对于精确值来讲，Foo 和 foo 是不同的，2014 和 2014-09-15 也是不同的。

另一方面，*全文* 是指文本数据（通常以人类容易识别的语言书写），例如一个推文的内容或一封邮件的内容。

注意

全文通常是指非结构化的数据，但这里有一个误解：自然语言是高度结构化的。问题在于自然语言的规则是复杂的，导致计算机难以正确解析。例如，考虑这条语句：

May is fun but June bores me.

它指的是月份还是人？

精确值很容易查询。结果是二进制的：要么匹配查询，要么不匹配。这种查询很容易用 SQL 表示：

WHERE name = "John Smith"

AND user\_id = 2

AND date > "2014-09-15"

查询全文数据要微妙的多。我们问的不只是“这个文档匹配查询吗”，而是“该文档匹配查询的程度有多大？”换句话说，该文档与给定查询的相关性如何？

我们很少对全文类型的域做精确匹配。相反，我们希望在文本类型的域中搜索。不仅如此，我们还希望搜索能够理解我们的 *意图* ：

* 搜索 UK ，会返回包含 United Kindom 的文档。
* 搜索 jump ，会匹配 jumped ， jumps ， jumping ，甚至是 leap 。
* 搜索 johnny walker 会匹配 Johnnie Walker ， johnnie depp 应该匹配 Johnny Depp 。
* fox news hunting 应该返回福克斯新闻（ Foxs News ）中关于狩猎的故事，同时， fox hunting news 应该返回关于猎狐的故事。

为了促进这类在全文域中的查询，Elasticsearch 首先 *分析* 文档，之后根据结果创建 *倒排索引* 。在接下来的两节，我们会讨论倒排索引和分析过程。

### 倒排索引

Elasticsearch 使用一种称为 *倒排索引* 的结构，它适用于快速的全文搜索。一个倒排索引由文档中所有不重复词的列表构成，对于其中每个词，有一个包含它的文档列表。

例如，假设我们有两个文档，每个文档的 content 域包含如下内容：

1. The quick brown fox jumped over the lazy dog
2. Quick brown foxes leap over lazy dogs in summer

为了创建倒排索引，我们首先将每个文档的 content 域拆分成单独的 词（我们称它为 词条 或 tokens），创建一个包含所有不重复词条的排序列表，然后列出每个词条出现在哪个文档。结果如下所示：

Term Doc\_1 Doc\_2

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

Quick | | X

The | X |

brown | X | X

dog | X |

dogs | | X

fox | X |

foxes | | X

in | | X

jumped | X |

lazy | X | X

leap | | X

over | X | X

quick | X |

summer | | X

the | X |

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

现在，如果我们想搜索 quick brown ，我们只需要查找包含每个词条的文档：

Term Doc\_1 Doc\_2

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

brown | X | X

quick | X |

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

Total | 2 | 1

两个文档都匹配，但是第一个文档比第二个匹配度更高。如果我们使用仅计算匹配词条数量的简单 *相似性算法* ，那么，我们可以说，对于我们查询的相关性来讲，第一个文档比第二个文档更佳。

但是，我们目前的倒排索引有一些问题：

* Quick 和 quick 以独立的词条出现，然而用户可能认为它们是相同的词。
* fox 和 foxes 非常相似, 就像 dog 和 dogs ；他们有相同的词根。
* jumped 和 leap, 尽管没有相同的词根，但他们的意思很相近。他们是同义词。

使用前面的索引搜索 +Quick +fox 不会得到任何匹配文档。（记住，+ 前缀表明这个词必须存在。）只有同时出现 Quick 和 fox 的文档才满足这个查询条件，但是第一个文档包含 quick fox ，第二个文档包含 Quick foxes 。

我们的用户可以合理的期望两个文档与查询匹配。我们可以做的更好。

如果我们将词条规范为标准模式，那么我们可以找到与用户搜索的词条不完全一致，但具有足够相关性的文档。例如：

* Quick 可以小写化为 quick 。
* foxes 可以 *词干提取* --变为词根的格式-- 为 fox 。类似的， dogs 可以为提取为 dog 。
* jumped 和 leap 是同义词，可以索引为相同的单词 jump 。

现在索引看上去像这样：

Term Doc\_1 Doc\_2

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

brown | X | X

dog | X | X

fox | X | X

in | | X

jump | X | X

lazy | X | X

over | X | X

quick | X | X

summer | | X

the | X | X

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

这还远远不够。我们搜索 +Quick +fox *仍然* 会失败，因为在我们的索引中，已经没有 Quick 了。但是，如果我们对搜索的字符串使用与 content 域相同的标准化规则，会变成查询 +quick +fox ，这样两个文档都会匹配！

注意

这非常重要。你只能搜索在索引中出现的词条，所以索引文本和查询字符串必须标准化为相同的格式。

分词和标准化的过程称为 *分析* ， 我们会在下个章节讨论。

### 分析与分析器

分析 包含下面的过程：

* 首先，将一块文本分成适合于倒排索引的独立的 词条 ，
* 之后，将这些词条统一化为标准格式以提高它们的“可搜索性”，或者 recall

分析器执行上面的工作。 分析器 实际上是将三个功能封装到了一个包里：

字符过滤器

首先，字符串按顺序通过每个 字符过滤器 。他们的任务是在分词前整理字符串。一个字符过滤器可以用来去掉HTML，或者将 & 转化成 `and`。

分词器

其次，字符串被 分词器 分为单个的词条。一个简单的分词器遇到空格和标点的时候，可能会将文本拆分成词条。

Token 过滤器

最后，词条按顺序通过每个 token 过滤器 。这个过程可能会改变词条（例如，小写化 Quick ），删除词条（例如， 像 a`， `and`， `the 等无用词），或者增加词条（例如，像 jump 和 leap 这种同义词）。

Elasticsearch提供了开箱即用的字符过滤器、分词器和token 过滤器。 这些可以组合起来形成自定义的分析器以用于不同的目的。我们会在 [自定义分析器](https://www.elastic.co/guide/cn/elasticsearch/guide/current/custom-analyzers.html) 章节详细讨论。

**内置分析器**

但是， Elasticsearch还附带了可以直接使用的预包装的分析器。 接下来我们会列出最重要的分析器。为了证明它们的差异，我们看看每个分析器会从下面的字符串得到哪些词条：

"Set the shape to semi-transparent by calling set\_trans(5)"

标准分析器

标准分析器是Elasticsearch默认使用的分析器。它是分析各种语言文本最常用的选择。它根据 [Unicode 联盟](http://www.unicode.org/reports/tr29/) 定义的 单词边界 划分文本。删除绝大部分标点。最后，将词条小写。它会产生

set, the, shape, to, semi, transparent, by, calling, set\_trans, 5

简单分析器

简单分析器在任何不是字母的地方分隔文本，将词条小写。它会产生

set, the, shape, to, semi, transparent, by, calling, set, trans

空格分析器

空格分析器在空格的地方划分文本。它会产生

Set, the, shape, to, semi-transparent, by, calling, set\_trans(5)

语言分析器

特定语言分析器可用于 [很多语言](https://www.elastic.co/guide/en/elasticsearch/reference/master/analysis-lang-analyzer.html)。它们可以考虑指定语言的特点。例如， 英语 分析器附带了一组英语无用词（常用单词，例如 and 或者 the ，它们对相关性没有多少影响），它们会被删除。 由于理解英语语法的规则，这个分词器可以提取英语单词的 词干 。

英语 分词器会产生下面的词条：

set, shape, semi, transpar, call, set\_tran, 5

注意看 transparent`、 `calling 和 set\_trans 已经变为词根格式。

**什么时候使用分析器**

当我们 索引 一个文档，它的全文域被分析成词条以用来创建倒排索引。 但是，当我们在全文域 搜索 的时候，我们需要将查询字符串通过 相同的分析过程 ，以保证我们搜索的词条格式与索引中的词条格式一致。

全文查询，理解每个域是如何定义的，因此它们可以做 正确的事：

* 当你查询一个 全文 域时， 会对查询字符串应用相同的分析器，以产生正确的搜索词条列表。
* 当你查询一个 精确值 域时，不会分析查询字符串， 而是搜索你指定的精确值。

现在你可以理解在 [开始章节](https://www.elastic.co/guide/cn/elasticsearch/guide/current/mapping-analysis.html) 的查询为什么返回那样的结果：

* date 域包含一个精确值：单独的词条 `2014-09-15`。
* \_all 域是一个全文域，所以分词进程将日期转化为三个词条： `2014`， `09`， 和 `15`。

当我们在 \_all 域查询 2014`，它匹配所有的12条推文，因为它们都含有 `2014 ：

GET /\_search?q=2014 # 12 results

**拷贝为 CURL**[在 SENSE 中查看](http://loca5601/app/sense/?load_from=https://www.elastic.co/guide/cn/elasticsearch/guide/current/snippets/052_Mapping_Analysis/25_Data_type_differences.json)

当我们在 \_all 域查询 2014-09-15`，它首先分析查询字符串，产生匹配 `2014`， `09`， 或 `15 中 任意 词条的查询。这也会匹配所有12条推文，因为它们都含有 2014 ：

GET /\_search?q=2014-09-15 # 12 results !

**拷贝为 CURL**[在 SENSE 中查看](http://loca5601/app/sense/?load_from=https://www.elastic.co/guide/cn/elasticsearch/guide/current/snippets/052_Mapping_Analysis/25_Data_type_differences.json)

当我们在 date 域查询 `2014-09-15`，它寻找 精确 日期，只找到一个推文：

GET /\_search?q=date:2014-09-15 # 1 result

**拷贝为 CURL**[在 SENSE 中查看](http://loca5601/app/sense/?load_from=https://www.elastic.co/guide/cn/elasticsearch/guide/current/snippets/052_Mapping_Analysis/25_Data_type_differences.json)

当我们在 date 域查询 `2014`，它找不到任何文档，因为没有文档含有这个精确日志：

GET /\_search?q=date:2014 # 0 results !

**拷贝为 CURL**[在 SENSE 中查看](http://loca5601/app/sense/?load_from=https://www.elastic.co/guide/cn/elasticsearch/guide/current/snippets/052_Mapping_Analysis/25_Data_type_differences.json)

**测试分析器**

有些时候很难理解分词的过程和实际被存储到索引中的词条，特别是你刚接触 Elasticsearch。为了理解发生了什么，你可以使用 analyze API 来看文本是如何被分析的。在消息体里，指定分析器和要分析的文本：

GET /\_analyze

{

"analyzer": "standard",

"text": "Text to analyze"

}

**拷贝为 CURL**[在 SENSE 中查看](http://loca5601/app/sense/?load_from=https://www.elastic.co/guide/cn/elasticsearch/guide/current/snippets/052_Mapping_Analysis/40_Analyze.json)

结果中每个元素代表一个单独的词条：

{

"tokens": [

{

"token": "text",

"start\_offset": 0,

"end\_offset": 4,

"type": "<ALPHANUM>",

"position": 1

},

{

"token": "to",

"start\_offset": 5,

"end\_offset": 7,

"type": "<ALPHANUM>",

"position": 2

},

{

"token": "analyze",

"start\_offset": 8,

"end\_offset": 15,

"type": "<ALPHANUM>",

"position": 3

}

]

}

token 是实际存储到索引中的词条。 position 指明词条在原始文本中出现的位置。 start\_offset 和 end\_offset 指明字符在原始字符串中的位置。

提示

每个分析器的 type 值都不一样，可以忽略它们。它们在Elasticsearch中的唯一作用在于[keep\_types token 过滤器](https://www.elastic.co/guide/en/elasticsearch/reference/master/analysis-keep-types-tokenfilter.html)。

analyze API 是一个有用的工具，它有助于我们理解Elasticsearch索引内部发生了什么，随着深入，我们会进一步讨论它。

**指定分析器**

当Elasticsearch在你的文档中检测到一个新的字符串域 ，它会自动设置其为一个全文 字符串 域，使用 标准 分析器对它进行分析。

你不希望总是这样。可能你想使用一个不同的分析器，适用于你的数据使用的语言。有时候你想要一个字符串域就是一个字符串域--不使用分析，直接索引你传入的精确值，例如用户ID或者一个内部的状态域或标签。

要做到这一点，我们必须手动指定这些域的映射。

### 映射

为了能够将时间域视为时间，数字域视为数字，字符串域视为全文或精确值字符串， Elasticsearch 需要知道每个域中数据的类型。这个信息包含在映射中。

如 [*数据输入和输出*](https://www.elastic.co/guide/cn/elasticsearch/guide/current/data-in-data-out.html) 中解释的， 索引中每个文档都有 *类型* 。每种类型都有它自己的 *映射* ，或者 *模式定义* 。映射定义了类型中的域，每个域的数据类型，以及Elasticsearch如何处理这些域。映射也用于配置与类型有关的元数据。

我们会在 [类型和映射](https://www.elastic.co/guide/cn/elasticsearch/guide/current/mapping.html) 详细讨论映射。本节，我们只讨论足够让你入门的内容。

**核心简单域类型**

Elasticsearch 支持 如下简单域类型：

* 字符串: string
* 整数 : byte, short, integer, long
* 浮点数: float, double
* 布尔型: boolean
* 日期: date

当你索引一个包含新域的文档--之前未曾出现-- Elasticsearch 会使用 [*动态映射*](https://www.elastic.co/guide/cn/elasticsearch/guide/current/dynamic-mapping.html) ，通过JSON中基本数据类型，尝试猜测域类型，使用如下规则：

|  |  |
| --- | --- |
| **JSON type** | **域 type** |
| 布尔型: true 或者 false | boolean |
| 整数: 123 | long |
| 浮点数: 123.45 | double |
| 字符串，有效日期: 2014-09-15 | date |
| 字符串: foo bar | string |

注意

这意味着如果你通过引号( "123" )索引一个数字，它会被映射为 string 类型，而不是 long。但是，如果这个域已经映射为 long ，那么 Elasticsearch 会尝试将这个字符串转化为 long ，如果无法转化，则抛出一个异常。

**查看映射**

通过 /\_mapping ，我们可以查看 Elasticsearch 在一个或多个索引中的一个或多个类型的映射 。在 [开始章节](https://www.elastic.co/guide/cn/elasticsearch/guide/current/mapping-analysis.html) ，我们已经取得索引 gb 中类型 tweet 的映射：

GET /gb/\_mapping/tweet

Elasticsearch 根据我们索引的文档，为域(称为 *属性* )动态生成的映射。

{

"gb": {

"mappings": {

"tweet": {

"properties": {

"date": {

"type": "date",

"format": "strict\_date\_optional\_time||epoch\_millis"

},

"name": {

"type": "string"

},

"tweet": {

"type": "string"

},

"user\_id": {

"type": "long"

}

}

}

}

}

}

提示

错误的映射，例如 将 age 域映射为 string 类型，而不是 integer ，会导致查询出现令人困惑的结果。

检查一下！而不是假设你的映射是正确的。

**自定义域映射**

尽管在很多情况下基本域数据类型 已经够用，但你经常需要为单独域自定义映射 ，特别是字符串域。自定义映射允许你执行下面的操作：

* 全文字符串域和精确值字符串域的区别
* 使用特定语言分析器
* 优化域以适应部分匹配
* 指定自定义数据格式
* 还有更多

域最重要的属性是 type 。对于不是 string 的域，你一般只需要设置 type ：

{

"number\_of\_clicks": {

"type": "integer"

}

}

默认， string 类型域会被认为包含全文。就是说，它们的值在索引前，会通过 一个分析器，针对于这个域的查询在搜索前也会经过一个分析器。

string 域映射的两个最重要 属性是 index 和 analyzer 。

**index**

index 属性控制怎样索引字符串。它可以是下面三个值：

analyzed

首先分析字符串，然后索引它。换句话说，以全文索引这个域。

not\_analyzed

  索引这个域，所以它能够被搜索，但索引的是精确值。不会对它进行分析。

no

不索引这个域。这个域不会被搜索到。

string 域 index 属性默认是 analyzed 。如果我们想映射这个字段为一个精确值，我们需要设置它为 not\_analyzed ：

{

"tag": {

"type": "string",

"index": "not\_analyzed"

}

}

注意

其他简单类型（例如 long ， double ， date 等）也接受 index 参数，但有意义的值只有 no 和 not\_analyzed ， 因为它们永远不会被分析。

**analyzer**

对于 analyzed 字符串域，用 analyzer 属性指定在搜索和索引时使用的分析器。默认， Elasticsearch 使用 standard 分析器， 但你可以指定一个内置的分析器替代它，例如 whitespace 、 simple 和 `english`：

{

"tweet": {

"type": "string",

"analyzer": "english"

}

}

在 [自定义分析器](https://www.elastic.co/guide/cn/elasticsearch/guide/current/custom-analyzers.html) ，我们会展示怎样定义和使用自定义分析器。

**更新映射**

当你首次 创建一个索引的时候，可以指定类型的映射。你也可以使用 /\_mapping 为新类型（或者为存在的类型更新映射）增加映射。

注意

尽管你可以 *增加\_ 一个存在的映射，你不能 \_修改* 存在的域映射。如果一个域的映射已经存在，那么该域的数据可能已经被索引。如果你意图修改这个域的映射，索引的数据可能会出错，不能被正常的搜索。

我们可以更新一个映射来添加一个新域，但不能将一个存在的域从 analyzed 改为 not\_analyzed 。

为了描述指定映射的两种方式，我们先删除 gd 索引：

DELETE /gb

然后创建一个新索引，指定 tweet 域使用 english 分析器：

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

{

"mappings": {

"tweet" : {

"properties" : {

"tweet" : {

"type" : "string",

"analyzer": "english"

},

"date" : {

"type" : "date"

},

"name" : {

"type" : "string"

},

"user\_id" : {

"type" : "long"

}

}

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/mapping-intro.html#CO19-1) | 通过消息体中指定的 mappings 创建了索引。 |

稍后，我们决定在 tweet 映射增加一个新的名为 tag 的 not\_analyzed 的文本域，使用 \_mapping ：

PUT /gb/\_mapping/tweet

{

"properties" : {

"tag" : {

"type" : "string",

"index": "not\_analyzed"

}

}

}

注意，我们不需要再次列出所有已存在的域，因为无论如何我们都无法改变它们。新域已经被合并到存在的映射中。

**测试映射**

你可以使用 analyze API 测试字符串域的映射。比较下面两个请求的输出：

GET /gb/\_analyze

{

"field": "tweet",

"text": "Black-cats" https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

}

GET /gb/\_analyze

{

"field": "tag",

"text": "Black-cats" https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/mapping-intro.html#CO20-1) [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/mapping-intro.html#CO20-2) | 消息体里面传输我们想要分析的文本。 |

tweet 域产生两个词条 black 和 cat ， tag 域产生单独的词条 Black-cats 。换句话说，我们的映射正常工作。

### 复杂核心域类型

除了我们提到的简单标量数据类型， JSON 还有 null 值，数组，和对象，这些 Elasticsearch 都是支持的。

**多值域**

很有可能，我们希望 tag 域 包含多个标签。我们可以以数组的形式索引标签：

{ "tag": [ "search", "nosql" ]}

对于数组，没有特殊的映射需求。任何域都可以包含0、1或者多个值，就像全文域分析得到多个词条。

这暗示 *数组中所有的值必须是相同数据类型的* 。你不能将日期和字符串混在一起。如果你通过索引数组来创建新的域，Elasticsearch 会用数组中第一个值的数据类型作为这个域的 类型 。

注意

当你从 Elasticsearch 得到一个文档，每个数组的顺序和你当初索引文档时一样。你得到的 \_source 域，包含与你索引的一模一样的 JSON 文档。

但是，数组是以多值域 *索引的*—可以搜索，但是无序的。 在搜索的时候，你不能指定 “第一个” 或者 “最后一个”。 更确切的说，把数组想象成 *装在袋子里的值* 。

**空域**

当然，数组可以为空。 这相当于存在零值。 事实上，在 Lucene 中是不能存储 null 值的，所以我们认为存在 null 值的域为空域。

下面三种域被认为是空的，它们将不会被索引：

"null\_value": null,

"empty\_array": [],

"array\_with\_null\_value": [ null ]

**多层级对象**

我们讨论的最后一个 JSON 原生数据类是 *对象* -- 在其他语言中称为哈希，哈希 map，字典或者关联数组。

*内部对象* 经常用于 嵌入一个实体或对象到其它对象中。例如，与其在 tweet 文档中包含 user\_name 和 user\_id 域，我们也可以这样写：

{

"tweet": "Elasticsearch is very flexible",

"user": {

"id": "@johnsmith",

"gender": "male",

"age": 26,

"name": {

"full": "John Smith",

"first": "John",

"last": "Smith"

}

}

}

**内部对象的映射**

Elasticsearch 会动态 监测新的对象域并映射它们为 对象 ，在 properties 属性下列出内部域：

{

"gb": {

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

"properties": {

"tweet": { "type": "string" },

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

"type": "object",

"properties": {

"id": { "type": "string" },

"gender": { "type": "string" },

"age": { "type": "long" },

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

"type": "object",

"properties": {

"full": { "type": "string" },

"first": { "type": "string" },

"last": { "type": "string" }

}

}

}

}

}

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/complex-core-fields.html#CO21-1) | 根对象 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/complex-core-fields.html#CO21-2) [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/complex-core-fields.html#CO21-3) | 内部对象 |

user 和 name 域的映射结构与 tweet 类型的相同。事实上， type 映射只是一种特殊的 对象 映射，我们称之为 *根对象* 。除了它有一些文档元数据的特殊顶级域，例如 \_source 和 \_all 域，它和其他对象一样。

**内部对象是如何索引的**

Lucene 不理解内部对象。 Lucene 文档是由一组键值对列表组成的。为了能让 Elasticsearch 有效地索引内部类，它把我们的文档转化成这样：

{

"tweet": [elasticsearch, flexible, very],

"user.id": [@johnsmith],

"user.gender": [male],

"user.age": [26],

"user.name.full": [john, smith],

"user.name.first": [john],

"user.name.last": [smith]

}

*内部域* 可以通过名称引用（例如， first ）。为了区分同名的两个域，我们可以使用全 *路径* （例如， user.name.first ） 或 type 名加路径（ tweet.user.name.first ）。

注意

在前面简单扁平的文档中，没有 user 和 user.name 域。Lucene 索引只有标量和简单值，没有复杂数据结构。

**内部对象数组**

最后，考虑包含 内部对象的数组是如何被索引的。 假设我们有个 followers 数组：

{

"followers": [

{ "age": 35, "name": "Mary White"},

{ "age": 26, "name": "Alex Jones"},

{ "age": 19, "name": "Lisa Smith"}

]

}

这个文档会像我们之前描述的那样被扁平化处理，结果如下所示：

{

"followers.age": [19, 26, 35],

"followers.name": [alex, jones, lisa, smith, mary, white]

}

{age: 35} 和 {name: Mary White} 之间的相关性已经丢失了，因为每个多值域只是一包无序的值，而不是有序数组。这足以让我们问，“有一个26岁的追随者？”

但是我们不能得到一个准确的答案：“是否有一个26岁 *名字叫 Alex Jones* 的追随者？”

相关内部对象被称为 *nested* 对象，可以回答上面的查询，我们稍后会在[*嵌套对象*](https://www.elastic.co/guide/cn/elasticsearch/guide/current/nested-objects.html)中介绍它。

## 请求体查询

*简易* 查询 —[query-string search](https://www.elastic.co/guide/cn/elasticsearch/guide/current/search-lite.html)— 对于用命令行进行点对点（ad-hoc）查询是非常有用的。 然而，为了充分利用查询的强大功能，你应该使用 *请求体* search API， 之所以称之为请求体查询(Full-Body Search)，因为大部分参数是通过 Http 请求体而非查询字符串来传递的。

请求体查询 —下文简称 *查询*—不仅可以处理自身的查询请求，还允许你对结果进行片段强调（高亮）、对所有或部分结果进行聚合分析，同时还可以给出 *你是不是想找* 的建议，这些建议可以引导使用者快速找到他想要的结果。

### 空查询

让我们以 最简单的 search API 的形式开启我们的旅程，空查询将返回所有索引库（indices)中的所有文档：

GET /\_search

{}

|  |  |
| --- | --- |
|  | 这是一个空的请求体。 |

只用一个查询字符串，你就可以在一个、多个或者 \_all 索引库（indices）和一个、多个或者所有types中查询：

GET /index\_2014\*/type1,type2/\_search

{}

同时你可以使用 from 和 size 参数来分页：

GET /\_search

{

"from": 30,

"size": 10

}

**一个带请求体的 GET 请求？**

某些特定语言（特别是 JavaScript）的 HTTP 库是不允许 GET 请求带有请求体的。 事实上，一些使用者对于 GET 请求可以带请求体感到非常的吃惊。

而事实是这个RFC文档 [RFC 7231](http://tools.ietf.org/html/rfc7231#page-24)— 一个专门负责处理 HTTP 语义和内容的文档 — 并没有规定一个带有请求体的 GET 请求应该如何处理！结果是，一些 HTTP 服务器允许这样子，而有一些 — 特别是一些用于缓存和代理的服务器 — 则不允许。

对于一个查询请求，Elasticsearch 的工程师偏向于使用 GET 方式，因为他们觉得它比 POST 能更好的描述信息检索（retrieving information）的行为。然而，因为带请求体的 GET 请求并不被广泛支持，所以 search API 同时支持 POST 请求：

POST /\_search

{

"from": 30,

"size": 10

}

类似的规则可以应用于任何需要带请求体的 GET API。

我们将在聚合 [聚合](https://www.elastic.co/guide/cn/elasticsearch/guide/current/aggregations.html) 章节深入介绍聚合（aggregations），而现在，我们将聚焦在查询。

相对于使用晦涩难懂的查询字符串的方式，一个带请求体的查询允许我们使用 查询领域特定语言（query domain-specific language） 或者 Query DSL 来写查询语句。

## 查询表达式

查询表达式(Query DSL)是一种非常灵活又富有表现力的 查询语言。 Elasticsearch 使用它可以以简单的 JSON 接口来展现 Lucene 功能的绝大部分。在你的应用中，你应该用它来编写你的查询语句。它可以使你的查询语句更灵活、更精确、易读和易调试。

要使用这种查询表达式，只需将查询语句传递给 query 参数：

GET /\_search

{

"query": YOUR\_QUERY\_HERE

}

空查询（empty search） —{}— 在功能上等价于使用 match\_all 查询， 正如其名字一样，匹配所有文档：

GET /\_search

{

"query": {

"match\_all": {}

}

}

### 查询语句的结构

一个查询语句 的典型结构：

{

QUERY\_NAME: {

ARGUMENT: VALUE,

ARGUMENT: VALUE,...

}

}

如果是针对某个字段，那么它的结构如下：

{

QUERY\_NAME: {

FIELD\_NAME: {

ARGUMENT: VALUE,

ARGUMENT: VALUE,...

}

}

}

举个例子，你可以使用 match 查询语句 来查询 tweet 字段中包含 elasticsearch 的 tweet：

{

"match": {

"tweet": "elasticsearch"

}

}

完整的查询请求如下：

GET /\_search

{

"query": {

"match": {

"tweet": "elasticsearch"

}

}

}

### 合并查询语句

查询语句(Query clauses) 就像一些简单的组合块 ，这些组合块可以彼此之间合并组成更复杂的查询。这些语句可以是如下形式：

* 叶子语句（Leaf clauses） (就像 match 语句) 被用于将查询字符串和一个字段（或者多个字段）对比。
* 复合(Compound) 语句 主要用于 合并其它查询语句。 比如，一个 bool 语句 允许在你需要的时候组合其它语句，无论是 must 匹配、 must\_not 匹配还是 should 匹配，同时它可以包含不评分的过滤器（filters）：

{

"bool": {

"must": { "match": { "tweet": "elasticsearch" }},

"must\_not": { "match": { "name": "mary" }},

"should": { "match": { "tweet": "full text" }},

"filter": { "range": { "age" : { "gt" : 30 }} }

}

}

一条复合语句可以合并 任何 其它查询语句，包括复合语句，了解这一点是很重要的。这就意味着，复合语句之间可以互相嵌套，可以表达非常复杂的逻辑。

例如，以下查询是为了找出信件正文包含 business opportunity 的星标邮件，或者在收件箱正文包含business opportunity 的非垃圾邮件：

{

"bool": {

"must": { "match": { "email": "business opportunity" }},

"should": [

{ "match": { "starred": true }},

{ "bool": {

"must": { "match": { "folder": "inbox" }},

"must\_not": { "match": { "spam": true }}

}}

],

"minimum\_should\_match": 1

}

}

到目前为止，你不必太在意这个例子的细节，我们会在后面详细解释。最重要的是你要理解到，一条复合语句可以将多条语句 — 叶子语句和其它复合语句 — 合并成一个单一的查询语句。

### 查询与过滤

Elasticsearch 使用的查询语言（DSL） 拥有一套查询组件，这些组件可以以无限组合的方式进行搭配。这套组件可以在以下两种情况下使用：过滤情况（filtering context）和查询情况（query context）。

当使用于 过滤情况 时，查询被设置成一个“不评分”或者“过滤”查询。即，这个查询只是简单的问一个问题：“这篇文档是否匹配？”。回答也是非常的简单，yes 或者 no ，二者必居其一。

* created 时间是否在 2013 与 2014 这个区间？
* status 字段是否包含 published 这个单词？
* lat\_lon 字段表示的位置是否在指定点的 10km 范围内？

当使用于 查询情况 时，查询就变成了一个“评分”的查询。和不评分的查询类似，也要去判断这个文档是否匹配，同时它还需要判断这个文档匹配的有 \_多好\_（匹配程度如何）。 此查询的典型用法是用于查找以下文档：

* 查找与 full text search 这个词语最佳匹配的文档
* 包含 run 这个词，也能匹配 runs 、 running 、 jog 或者 sprint
* 包含 quick 、 brown 和 fox 这几个词 — 词之间离的越近，文档相关性越高
* 标有 lucene 、 search 或者 java 标签 — 标签越多，相关性越高

一个评分查询计算每一个文档与此查询的 \_相关程度\_，同时将这个相关程度分配给表示相关性的字段 `\_score`，并且按照相关性对匹配到的文档进行排序。这种相关性的概念是非常适合全文搜索的情况，因为全文搜索几乎没有完全 “正确” 的答案。

自 Elasticsearch 问世以来，查询与过滤（queries and filters）就独自成为 Elasticsearch 的组件。但从 Elasticsearch 2.0 开始，过滤（filters）已经从技术上被排除了，同时所有的查询（queries）拥有变成不评分查询的能力。

然而，为了明确和简单，我们用 "filter" 这个词表示不评分、只过滤情况下的查询。你可以把 "filter" 、 "filtering query" 和 "non-scoring query" 这几个词视为相同的。

相似的，如果单独地不加任何修饰词地使用 "query" 这个词，我们指的是 "scoring query" 。

### 性能差异

过滤查询（Filtering queries）只是简单的检查包含或者排除，这就使得计算起来非常快。考虑到至少有一个过滤查询（filtering query）的结果是 “稀少的”（很少匹配的文档），并且经常使用不评分查询（non-scoring queries），结果会被缓存到内存中以便快速读取，所以有各种各样的手段来优化查询结果。

相反，评分查询（scoring queries）不仅仅要找出 匹配的文档，还要计算每个匹配文档的相关性，计算相关性使得它们比不评分查询费力的多。同时，查询结果并不缓存。

多亏倒排索引（inverted index），一个简单的评分查询在匹配少量文档时可能与一个涵盖百万文档的filter表现的一样好，甚至会更好。但是在一般情况下，一个filter 会比一个评分的query性能更优异，并且每次都表现的很稳定。

过滤（filtering）的目标是减少那些需要通过评分查询（scoring queries）进行检查的文档。

### 如何选择查询与过滤

通常的规则是，使用 查询（query）语句来进行 全文 搜索或者其它任何需要影响 相关性得分 的搜索。除此以外的情况都使用过滤（filters)。

### 最重要的查询

虽然 Elasticsearch 自带了很多的查询，但经常用到的也就那么几个。我们将在 [深入搜索](https://www.elastic.co/guide/cn/elasticsearch/guide/current/search-in-depth.html) 章节详细讨论那些查询的细节，接下来我们对最重要的几个查询进行简单介绍。

**match\_all 查询**

match\_all 查询简单的 匹配所有文档。在没有指定查询方式时，它是默认的查询：

{ "match\_all": {}}

它经常与 filter 结合使用--例如，检索收件箱里的所有邮件。所有邮件被认为具有相同的相关性，所以都将获得分值为 1 的中性 `\_score`。

**match 查询**

无论你在任何字段上进行的是全文搜索还是精确查询，match 查询是你可用的标准查询。

如果你在一个全文字段上使用 match 查询，在执行查询前，它将用正确的分析器去分析查询字符串：

{ "match": { "tweet": "About Search" }}

如果在一个精确值的字段上使用它， 例如数字、日期、布尔或者一个 not\_analyzed 字符串字段，那么它将会精确匹配给定的值：

{ "match": { "age": 26 }}

{ "match": { "date": "2014-09-01" }}

{ "match": { "public": true }}

{ "match": { "tag": "full\_text" }}

对于精确值的查询，你可能需要使用 filter 语句来取代 query，因为 filter 将会被缓存。接下来，我们将看到一些关于 filter 的例子。

不像我们在 [*轻量* 搜索](https://www.elastic.co/guide/cn/elasticsearch/guide/current/search-lite.html) 章节介绍的字符串查询（query-string search）， match 查询不使用类似 +user\_id:2 +tweet:search 的查询语法。它只是去查找给定的单词。这就意味着将查询字段暴露给你的用户是安全的；你需要控制那些允许被查询字段，不易于抛出语法异常。

**multi\_match 查询**

multi\_match 查询可以在多个字段上执行相同的 match 查询：

{

"multi\_match": {

"query": "full text search",

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

}

}

**range 查询**

range 查询找出那些落在指定区间内的数字或者时间：

{

"range": {

"age": {

"gte": 20,

"lt": 30

}

}

}

被允许的操作符如下：

gt

大于

gte

大于等于

lt

小于

lte

小于等于

**term 查询**

term 查询被用于精确值 匹配，这些精确值可能是数字、时间、布尔或者那些 not\_analyzed 的字符串：

{ "term": { "age": 26 }}

{ "term": { "date": "2014-09-01" }}

{ "term": { "public": true }}

{ "term": { "tag": "full\_text" }}

term 查询对于输入的文本不 *分析* ，所以它将给定的值进行精确查询。

**terms 查询**

terms 查询和 term 查询一样，但它允许你指定多值进行匹配。如果这个字段包含了指定值中的任何一个值，那么这个文档满足条件：

{ "terms": { "tag": [ "search", "full\_text", "nosql" ] }}

和 term 查询一样，terms 查询对于输入的文本不分析。它查询那些精确匹配的值（包括在大小写、重音、空格等方面的差异）。

**exists 查询和 missing 查询**

exists 查询和 missing 查询被用于查找那些指定字段中有值 (exists) 或无值 (missing) 的文档。这与SQL中的 IS\_NULL (missing) 和 NOT IS\_NULL (exists) 在本质上具有共性：

{

"exists": {

"field": "title"

}

}

这些查询经常用于某个字段有值的情况和某个字段缺值的情况。

### 组合多查询

现实的查询需求从来都没有那么简单；它们需要在多个字段上查询多种多样的文本，并且根据一系列的标准来过滤。为了构建类似的高级查询，你需要一种能够将多查询组合成单一查询的查询方法。

你可以用 bool 查询来实现你的需求。这种查询将多查询组合在一起，成为用户自己想要的布尔查询。它接收以下参数：

must

文档 *必须* 匹配这些条件才能被包含进来。

must\_not

文档 *必须不* 匹配这些条件才能被包含进来。

should

如果满足这些语句中的任意语句，将增加 \_score ，否则，无任何影响。它们主要用于修正每个文档的相关性得分。

filter

*必须* 匹配，但它以不评分、过滤模式来进行。这些语句对评分没有贡献，只是根据过滤标准来排除或包含文档。

由于这是我们看到的第一个包含多个查询的查询，所以有必要讨论一下相关性得分是如何组合的。每一个子查询都独自地计算文档的相关性得分。一旦他们的得分被计算出来， bool 查询就将这些得分进行合并并且返回一个代表整个布尔操作的得分。

下面的查询用于查找 title 字段匹配 how to make millions 并且不被标识为 spam 的文档。那些被标识为 starred 或在2014之后的文档，将比另外那些文档拥有更高的排名。如果 \_两者\_ 都满足，那么它排名将更高：

{

"bool": {

"must": { "match": { "title": "how to make millions" }},

"must\_not": { "match": { "tag": "spam" }},

"should": [

{ "match": { "tag": "starred" }},

{ "range": { "date": { "gte": "2014-01-01" }}}

]

}

}

如果没有 must 语句，那么至少需要能够匹配其中的一条 should 语句。但，如果存在至少一条 must 语句，则对 should 语句的匹配没有要求。

**增加带过滤器（filtering）的查询**

如果我们不想因为文档的时间而影响得分，可以用 filter 语句来重写前面的例子：

{

"bool": {

"must": { "match": { "title": "how to make millions" }},

"must\_not": { "match": { "tag": "spam" }},

"should": [

{ "match": { "tag": "starred" }}

],

"filter": {

"range": { "date": { "gte": "2014-01-01" }}

}

}

}

|  |  |
| --- | --- |
|  | range 查询已经从 should 语句中移到 filter 语句 |

通过将 range 查询移到 filter 语句中，我们将它转成不评分的查询，将不再影响文档的相关性排名。由于它现在是一个不评分的查询，可以使用各种对 filter 查询有效的优化手段来提升性能。

所有查询都可以借鉴这种方式。将查询移到 bool 查询的 filter 语句中，这样它就自动的转成一个不评分的 filter 了。

如果你需要通过多个不同的标准来过滤你的文档，bool 查询本身也可以被用做不评分的查询。简单地将它放置到 filter 语句中并在内部构建布尔逻辑：

{

"bool": {

"must": { "match": { "title": "how to make millions" }},

"must\_not": { "match": { "tag": "spam" }},

"should": [

{ "match": { "tag": "starred" }}

],

"filter": {

"bool": {

"must": [

{ "range": { "date": { "gte": "2014-01-01" }}},

{ "range": { "price": { "lte": 29.99 }}}

],

"must\_not": [

{ "term": { "category": "ebooks" }}

]

}

}

}

}

|  |  |
| --- | --- |
|  | 将 bool 查询包裹在 filter 语句中，我们可以在过滤标准中增加布尔逻辑 |

通过混合布尔查询，我们可以在我们的查询请求中灵活地编写 scoring 和 filtering 查询逻辑。

**constant\_score 查询**

尽管没有 bool 查询使用这么频繁，constant\_score 查询也是你工具箱里有用的查询工具。它将一个不变的常量评分应用于所有匹配的文档。它被经常用于你只需要执行一个 filter 而没有其它查询（例如，评分查询）的情况下。

可以使用它来取代只有 filter 语句的 bool 查询。在性能上是完全相同的，但对于提高查询简洁性和清晰度有很大帮助。

{

"constant\_score": {

"filter": {

"term": { "category": "ebooks" }

}

}

}

|  |  |
| --- | --- |
|  | term 查询被放置在 constant\_score 中，转成不评分的 filter。这种方式可以用来取代只有 filter 语句的 bool 查询。 |

### 验证查询

查询可以变得非常的复杂，尤其 和不同的分析器与不同的字段映射结合时，理解起来就有点困难了。不过 validate-query API 可以用来验证查询是否合法。

GET /gb/tweet/\_validate/query

{

"query": {

"tweet" : {

"match" : "really powerful"

}

}

}

以上 validate 请求的应答告诉我们这个查询是不合法的：

{

"valid" : false,

"\_shards" : {

"total" : 1,

"successful" : 1,

"failed" : 0

}

}

**理解错误信息**

为了找出 查询不合法的原因，可以将 explain 参数 加到查询字符串中：

GET /gb/tweet/\_validate/query?explain

{

"query": {

"tweet" : {

"match" : "really powerful"

}

}

}

|  |  |
| --- | --- |
|  | explain 参数可以提供更多关于查询不合法的信息。 |

很明显，我们将查询类型(match)与字段名称 (tweet)搞混了：

{

"valid" : false,

"\_shards" : { ... },

"explanations" : [ {

"index" : "gb",

"valid" : false,

"error" : "org.elasticsearch.index.query.QueryParsingException:

[gb] No query registered for [tweet]"

} ]

}

**理解查询语句**

对于合法查询，使用 explain 参数将返回可读的描述，这对准确理解 Elasticsearch 是如何解析你的 query 是非常有用的：

GET /\_validate/query?explain

{

"query": {

"match" : {

"tweet" : "really powerful"

}

}

}

我们查询的每一个 index 都会返回对应的 explanation ，因为每一个 index 都有自己的映射和分析器：

{

"valid" : true,

"\_shards" : { ... },

"explanations" : [ {

"index" : "us",

"valid" : true,

"explanation" : "tweet:really tweet:powerful"

}, {

"index" : "gb",

"valid" : true,

"explanation" : "tweet:realli tweet:power"

} ]

}

从 explanation 中可以看出，匹配 really powerful 的 match 查询被重写为两个针对 tweet 字段的 single-term 查询，一个single-term查询对应查询字符串分出来的一个term。

当然，对于索引 us ，这两个 term 分别是 really 和 powerful ，而对于索引 gb ，term 则分别是 realli 和 power 。之所以出现这个情况，是由于我们将索引 gb 中 tweet 字段的分析器修改为 english分析器。

## 排序与相关性

默认情况下，返回的结果是按照 *相关性* 进行排序的——最相关的文档排在最前。 在本章的后面部分，我们会解释 *相关性* 意味着什么以及它是如何计算的， 不过让我们首先看看 sort 参数以及如何使用它。

### 排序

为了按照相关性来排序，需要将相关性表示为一个数值。在 Elasticsearch 中， *相关性得分* 由一个浮点数进行表示，并在搜索结果中通过 \_score 参数返回， 默认排序是 \_score 降序。

有时，相关性评分对你来说并没有意义。例如，下面的查询返回所有 user\_id 字段包含 1 的结果：

GET /\_search

{

"query" : {

"bool" : {

"filter" : {

"term" : {

"user\_id" : 1

}

}

}

}

}

这里没有一个有意义的分数：因为我们使用的是 filter （过滤），这表明我们只希望获取匹配 user\_id: 1的文档，并没有试图确定这些文档的相关性。 实际上文档将按照随机顺序返回，并且每个文档都会评为零分。

如果评分为零对你造成了困扰，你可以使用 constant\_score 查询进行替代：

GET /\_search

{

"query" : {

"constant\_score" : {

"filter" : {

"term" : {

"user\_id" : 1

}

}

}

}

}

这将让所有文档应用一个恒定分数（默认为 1 ）。它将执行与前述查询相同的查询，并且所有的文档将像之前一样随机返回，这些文档只是有了一个分数而不是零分。

**按照字段的值排序**

在这个案例中，通过时间来对 tweets 进行排序是有意义的，最新的 tweets 排在最前。 我们可以使用 sort 参数进行实现：

GET /\_search

{

"query" : {

"bool" : {

"filter" : { "term" : { "user\_id" : 1 }}

}

},

"sort": { "date": { "order": "desc" }}

}

你会注意到结果中的两个不同点：

"hits" : {

"total" : 6,

"max\_score" : null,

"hits" : [ {

"\_index" : "us",

"\_type" : "tweet",

"\_id" : "14",

"\_score" : null,

"\_source" : {

"date": "2014-09-24",

...

},

"sort" : [ 1411516800000 ]

},

...

}

|  |  |
| --- | --- |
|  | \_score 不被计算, 因为它并没有用于排序。 |
|  | date 字段的值表示为自 epoch (January 1, 1970 00:00:00 UTC)以来的毫秒数，通过 sort 字段的值进行返回。 |

首先我们在每个结果中有一个新的名为 sort 的元素，它包含了我们用于排序的值。 在这个案例中，我们按照 date 进行排序，在内部被索引为 *自 epoch 以来的毫秒数* 。 long 类型数 1411516800000 等价于日期字符串 2014-09-24 00:00:00 UTC 。

其次 \_score 和 max\_score 字段都是 null 。 计算 \_score 的花销巨大，通常仅用于排序； 我们并不根据相关性排序，所以记录 \_score 是没有意义的。如果无论如何你都要计算 \_score ， 你可以将track\_scores 参数设置为 true 。

一个简便方法是, 你可以 指定一个字段用来排序：

"sort": "number\_of\_children"

字段将会默认升序排序 ，而按照 \_score 的值进行降序排序。

**多级排序**

假定我们想要结合使用 date 和 \_score 进行查询，并且匹配的结果首先按照日期排序，然后按照相关性排序：

GET /\_search

{

"query" : {

"bool" : {

"must": { "match": { "tweet": "manage text search" }},

"filter" : { "term" : { "user\_id" : 2 }}

}

},

"sort": [

{ "date": { "order": "desc" }},

{ "\_score": { "order": "desc" }}

]

}

排序条件的顺序是很重要的。结果首先按第一个条件排序，仅当结果集的第一个 sort 值完全相同时才会按照第二个条件进行排序，以此类推。

多级排序并不一定包含 \_score 。你可以根据一些不同的字段进行排序， 如地理距离或是脚本计算的特定值。

Query-string 搜索 也支持自定义排序，可以在查询字符串中使用 sort 参数：

GET /\_search?sort=date:desc&sort=\_score&q=search

**字段多值的排序**

一种情形是字段有多个值的排序， 需要记住这些值并没有固有的顺序；一个多值的字段仅仅是多个值的包装，这时应该选择哪个进行排序呢？

对于数字或日期，你可以将多值字段减为单值，这可以通过使用 min 、 max 、 avg 或是 sum *排序模式* 。例如你可以按照每个 date 字段中的最早日期进行排序，通过以下方法：

"sort": {

"dates": {

"order": "asc",

"mode": "min"

}

}

### 字符串排序与多字段

被解析的字符串字段也是多值字段， 但是很少会按照你想要的方式进行排序。如果你想分析一个字符串，如 fine old art ， 这包含 3 项。我们很可能想要按第一项的字母排序，然后按第二项的字母排序，诸如此类，但是 Elasticsearch 在排序过程中没有这样的信息。

你可以使用 min 和 max 排序模式（默认是 min ），但是这会导致排序以 art 或是 old ，任何一个都不是所希望的。

为了以字符串字段进行排序，这个字段应仅包含一项： 整个 not\_analyzed 字符串。 但是我们仍需要 analyzed 字段，这样才能以全文进行查询

一个简单的方法是用两种方式对同一个字符串进行索引，这将在文档中包括两个字段： analyzed 用于搜索， not\_analyzed 用于排序

但是保存相同的字符串两次在 \_source 字段是浪费空间的。 我们真正想要做的是传递一个 单字段 但是却用两种方式索引它。所有的 \_core\_field 类型 (strings, numbers, Booleans, dates) 接收一个 fields 参数

该参数允许你转化一个简单的映射如：

"tweet": {

"type": "string",

"analyzer": "english"

}

为一个多字段映射如：

"tweet": {

"type": "string",

"analyzer": "english",

"fields": {

"raw": {

"type": "string",

"index": "not\_analyzed"

}

}

}

|  |  |
| --- | --- |
|  | tweet 主字段与之前的一样: 是一个 analyzed 全文字段。 |
|  | 新的 tweet.raw 子字段是 not\_analyzed. |

现在，至少只要我们重新索引了我们的数据，使用 tweet 字段用于搜索，tweet.raw 字段用于排序：

GET /\_search

{

"query": {

"match": {

"tweet": "elasticsearch"

}

},

"sort": "tweet.raw"

}

以全文 analyzed 字段排序会消耗大量的内存。获取更多信息请看 [聚合与分析](https://www.elastic.co/guide/cn/elasticsearch/guide/current/aggregations-and-analysis.html) 。

### 什么是相关性?

我们曾经讲过，默认情况下，返回结果是按相关性倒序排列的。 但是什么是相关性？ 相关性如何计算？

每个文档都有相关性评分，用一个正浮点数字段 \_score 来表示 。 \_score 的评分越高，相关性越高。

查询语句会为每个文档生成一个 \_score 字段。评分的计算方式取决于查询类型 不同的查询语句用于不同的目的： fuzzy 查询会计算与关键词的拼写相似程度，terms 查询会计算 找到的内容与关键词组成部分匹配的百分比，但是通常我们说的 relevance 是我们用来计算全文本字段的值相对于全文本检索词相似程度的算法。

Elasticsearch 的相似度算法 被定义为检索词频率/反向文档频率， TF/IDF ，包括以下内容：

检索词频率

检索词在该字段出现的频率？出现频率越高，相关性也越高。 字段中出现过 5 次要比只出现过 1 次的相关性高。

反向文档频率

每个检索词在索引中出现的频率？频率越高，相关性越低。检索词出现在多数文档中会比出现在少数文档中的权重更低。

字段长度准则

字段的长度是多少？长度越长，相关性越低。 检索词出现在一个短的 title 要比同样的词出现在一个长的 content 字段权重更大。

单个查询可以联合使用 TF/IDF 和其他方式，比如短语查询中检索词的距离或模糊查询里的检索词相似度。

相关性并不只是全文本检索的专利。也适用于 yes|no 的子句，匹配的子句越多，相关性评分越高。

如果多条查询子句被合并为一条复合查询语句 ，比如 bool 查询，则每个查询子句计算得出的评分会被合并到总的相关性评分中。

我们有一️整章着眼于相关性计算和如何让其配合你的需求 [控制相关度](https://www.elastic.co/guide/cn/elasticsearch/guide/current/controlling-relevance.html)。

**理解评分标准**

当调试一条复杂的查询语句时， 想要理解 \_score 究竟是如何计算是比较困难的。Elasticsearch 在 每个查询语句中都有一个 explain 参数，将 explain 设为 true 就可以得到更详细的信息。

GET /\_search?explain

{

"query" : { "match" : { "tweet" : "honeymoon" }}

}

|  |  |
| --- | --- |
|  | explain 参数可以让返回结果添加一个 \_score 评分的得来依据。 |

增加一个 explain 参数会为每个匹配到的文档产生一大堆额外内容，但是花时间去理解它是很有意义的。 如果现在看不明白也没关系 — 等你需要的时候再来回顾这一节就行。下面我们来一点点的了解这块知识点。

首先，我们看一下普通查询返回的元数据：

{

"\_index" : "us",

"\_type" : "tweet",

"\_id" : "12",

"\_score" : 0.076713204,

"\_source" : { ... trimmed ... },

这里加入了该文档来自于哪个节点哪个分片上的信息，这对我们是比较有帮助的，因为词频率和 文档频率是在每个分片中计算出来的，而不是每个索引中：

"\_shard" : 1,

"\_node" : "mzIVYCsqSWCG\_M\_ZffSs9Q",

然后它提供了 \_explanation 。每个 入口都包含一个 description 、 value 、 details 字段，它分别告诉你计算的类型、计算结果和任何我们需要的计算细节。

"\_explanation": {

"description": "weight(tweet:honeymoon in 0)

[PerFieldSimilarity], result of:",

"value": 0.076713204,

"details": [

{

"description": "fieldWeight in 0, product of:",

"value": 0.076713204,

"details": [

{

"description": "tf(freq=1.0), with freq of:",

"value": 1,

"details": [

{

"description": "termFreq=1.0",

"value": 1

}

]

},

{

"description": "idf(docFreq=1, maxDocs=1)",

"value": 0.30685282

},

{

"description": "fieldNorm(doc=0)",

"value": 0.25,

}

]

}

]

}

|  |  |
| --- | --- |
|  | honeymoon 相关性评分计算的总结 |
|  | 检索词频率 |
|  | 反向文档频率 |
|  | 字段长度准则 |

输出 explain 结果代价是十分昂贵的，它只能用作调试工具 。千万不要用于生产环境。

第一部分是关于计算的总结。告诉了我们 honeymoon 在 tweet 字段中的检索词频率/反向文档频率或TF/IDF， （这里的文档 0 是一个内部的 ID，跟我们没有关系，可以忽略。）

然后它提供了权重是如何计算的细节：

检索词频率:

检索词 `honeymoon` 在这个文档的 `tweet` 字段中的出现次数。

反向文档频率:

检索词 `honeymoon` 在索引上所有文档的 `tweet` 字段中出现的次数。

字段长度准则:

在这个文档中， `tweet` 字段内容的长度 -- 内容越长，值越小。

复杂的查询语句解释也非常复杂，但是包含的内容与上面例子大致相同。 通过这段信息我们可以了解搜索结果是如何产生的。

JSON 形式的 explain 描述是难以阅读的， 但是转成 YAML 会好很多，只需要在参数中加上 format=yaml 。

**理解文档是如何被匹配到的**

当 explain 选项加到某一文档上时， explain api 会帮助你理解为何这个文档会被匹配，更重要的是，一个文档为何没有被匹配。

请求路径为 /index/type/id/\_explain ，如下所示：

GET /us/tweet/12/\_explain

{

"query" : {

"bool" : {

"filter" : { "term" : { "user\_id" : 2 }},

"must" : { "match" : { "tweet" : "honeymoon" }}

}

}

}

不只是我们之前看到的充分解释 ，我们现在有了一个 description 元素，它将告诉我们：

"failure to match filter: cache(user\_id:[2 TO 2])"

也就是说我们的 user\_id 过滤子句使该文档不能匹配到。

### Doc Values 介绍

本章的最后一个话题是关于 Elasticsearch 内部的一些运行情况。在这里我们先不介绍新的知识点，所以我们应该意识到，Doc Values 是我们需要反复提到的一个重要话题。

当你对一个字段进行排序时，Elasticsearch 需要访问每个匹配到的文档得到相关的值。倒排索引的检索性能是非常快的，但是在字段值排序时却不是理想的结构。

* 在搜索的时候，我们能通过搜索关键词快速得到结果集。
* 当排序的时候，我们需要倒排索引里面某个字段值的集合。换句话说，我们需要 ``倒置`` 倒排索引。

``倒置`` 结构在其他系统中经常被称作 ``列存储`` 。实质上，它将所有单字段的值存储在单数据列中，这使得对其进行操作是十分高效的，例如排序。

在 Elasticsearch 中，doc values 就是一种列式存储结构，默认情况下每个字段的 doc values 都是激活的，doc values 是在索引时创建的，当字段索引时，Elasticsearch 为了能够快速检索，会把字段的值加入倒排索引中，同时它也会存储该字段的 doc values。

Elasticsearch 中的 doc vaules 常被应用到以下场景：

* 对一个字段进行排序
* 对一个字段进行聚合
* 某些过滤，比如地理位置过滤
* 某些与字段相关的脚本计算

因为文档值被序列化到磁盘，我们可以依靠操作系统的帮助来快速访问。当 working set 远小于节点的可用内存，系统会自动将所有的文档值保存在内存中，使得其读写十分高速； 当其远大于可用内存，操作系统会自动把 doc values 加载到系统的页缓存中，从而避免了 jvm 堆内存溢出异常。

我们稍后会深入讨论 doc values。现在所有你需要知道的是排序发生在索引时建立的平行数据结构中。

## 执行分布式检索

在继续之前，我们将绕道讨论一下在分布式环境中搜索是怎么执行的。 这比我们在 [分布式文档存储](https://www.elastic.co/guide/cn/elasticsearch/guide/current/distributed-docs.html) 章节讨论的基本的 增-删-改-查 (CRUD)请求要复杂一些。

**内容提示**

你可以根据兴趣阅读本章内容。你并不需要为了使用 Elasticsearch 而理解和记住所有的细节。

这章的阅读目的只为初步了解下工作原理，以便将来需要时可以及时找到这些知识， 但是不要被细节所困扰。

一个 CRUD 操作只对单个文档进行处理，文档的唯一性由 \_index, \_type, 和 [routing values](https://www.elastic.co/guide/cn/elasticsearch/guide/current/routing-value.html) （通常默认是该文档的 \_id ）的组合来确定。 这表示我们确切的知道集群中哪个分片含有此文档。

搜索需要一种更加复杂的执行模型因为我们不知道查询会命中哪些文档: 这些文档有可能在集群的任何分片上。 一个搜索请求必须询问我们关注的索引（index or indices）的所有分片的某个副本来确定它们是否含有任何匹配的文档。

但是找到所有的匹配文档仅仅完成事情的一半。 在 search 接口返回一个 ``page`` 结果之前，多分片中的结果必须组合成单个排序列表。 为此，搜索被执行成一个两阶段过程，我们称之为 query then fetch 。

### 查询阶段

在初始 查询阶段 时， 查询会广播到索引中每一个分片拷贝（主分片或者副本分片）。 每个分片在本地执行搜索并构建一个匹配文档的 \_优先队列\_。

**优先队列**

一个 优先队列 仅仅是一个存有 top-n 匹配文档的有序列表。优先队列的大小取决于分页参数 from 和 size 。例如，如下搜索请求将需要足够大的优先队列来放入100条文档。

GET /\_search

{

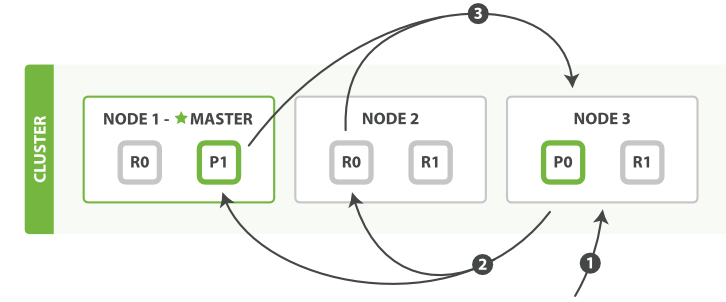
"from": 90,

"size": 10

}

这个查询阶段的过程如图 [图 14 “查询过程分布式搜索”](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_query_phase.html#img-distrib-search) 所示。

**图 14. 查询过程分布式搜索**



查询阶段包含以下三个步骤:

1. 客户端发送一个 search 请求到 Node 3 ， Node 3 会创建一个大小为 from + size 的空优先队列。
2. Node 3 将查询请求转发到索引的每个主分片或副本分片中。每个分片在本地执行查询并添加结果到大小为 from + size 的本地有序优先队列中。
3. 每个分片返回各自优先队列中所有文档的 ID 和排序值给协调节点，也就是 Node 3 ，它合并这些值到自己的优先队列中来产生一个全局排序后的结果列表。

当一个搜索请求被发送到某个节点时，这个节点就变成了协调节点。 这个节点的任务是广播查询请求到所有相关分片并将它们的响应整合成全局排序后的结果集合，这个结果集合会返回给客户端。

第一步是广播请求到索引中每一个节点的分片拷贝。就像 [document GET requests](https://www.elastic.co/guide/cn/elasticsearch/guide/current/distrib-read.html) 所描述的， 查询请求可以被某个主分片或某个副本分片处理， 这就是为什么更多的副本（当结合更多的硬件）能够增加搜索吞吐率。 协调节点将在之后的请求中轮询所有的分片拷贝来分摊负载。

每个分片在本地执行查询请求并且创建一个长度为 from + size 的优先队列—也就是说，每个分片创建的结果集足够大，均可以满足全局的搜索请求。 分片返回一个轻量级的结果列表到协调节点，它仅包含文档 ID 集合以及任何排序需要用到的值，例如 \_score 。

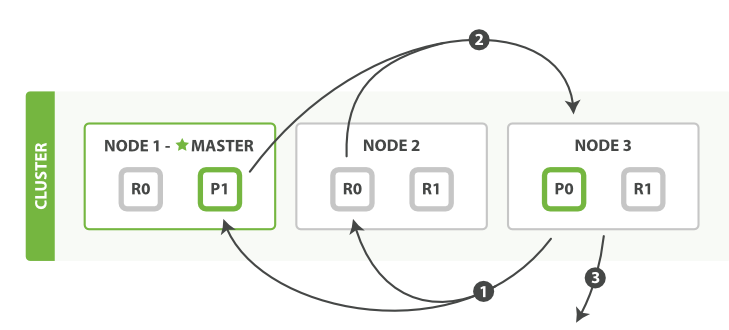
协调节点将这些分片级的结果合并到自己的有序优先队列里，它代表了全局排序结果集合。至此查询过程结束。

一个索引可以由一个或几个主分片组成， 所以一个针对单个索引的搜索请求需要能够把来自多个分片的结果组合起来。 针对 multiple 或者 all 索引的搜索工作方式也是完全一致的--仅仅是包含了更多的分片而已。

### 取回阶段

查询阶段标识哪些文档满足 搜索请求，但是我们仍然需要取回这些文档。这是取回阶段的任务, 正如 [图 15 “分布式搜索的取回阶段”](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_fetch_phase.html#img-distrib-fetch) 所展示的。

**图 15. 分布式搜索的取回阶段**



分布式阶段由以下步骤构成：

1. 协调节点辨别出哪些文档需要被取回并向相关的分片提交多个 GET 请求。
2. 每个分片加载并 丰富 文档，如果有需要的话，接着返回文档给协调节点。
3. 一旦所有的文档都被取回了，协调节点返回结果给客户端。

协调节点首先决定哪些文档 确实 需要被取回。例如，如果我们的查询指定了 { "from": 90, "size": 10 } ，最初的90个结果会被丢弃，只有从第91个开始的10个结果需要被取回。这些文档可能来自和最初搜索请求有关的一个、多个甚至全部分片。

协调节点给持有相关文档的每个分片创建一个 [multi-get request](https://www.elastic.co/guide/cn/elasticsearch/guide/current/distrib-multi-doc.html) ，并发送请求给同样处理查询阶段的分片副本。

分片加载文档体-- \_source 字段--如果有需要，用元数据和 [search snippet highlighting](https://www.elastic.co/guide/cn/elasticsearch/guide/current/highlighting-intro.html) 丰富结果文档。 一旦协调节点接收到所有的结果文档，它就组装这些结果为单个响应返回给客户端。

**深分页（Deep Pagination）**

先查后取的过程支持用 from 和 size 参数分页，但是这是 有限制的 。 要记住需要传递信息给协调节点的每个分片必须先创建一个 from + size 长度的队列，协调节点需要根据 number\_of\_shards \* (from + size) 排序文档，来找到被包含在 size 里的文档。

取决于你的文档的大小，分片的数量和你使用的硬件，给 10,000 到 50,000 的结果文档深分页（ 1,000 到 5,000 页）是完全可行的。但是使用足够大的 from 值，排序过程可能会变得非常沉重，使用大量的CPU、内存和带宽。因为这个原因，我们强烈建议你不要使用深分页。

实际上， “深分页” 很少符合人的行为。当2到3页过去以后，人会停止翻页，并且改变搜索标准。会不知疲倦地一页一页的获取网页直到你的服务崩溃的罪魁祸首一般是机器人或者web spider。

如果你 确实 需要从你的集群取回大量的文档，你可以通过用 scroll 查询禁用排序使这个取回行为更有效率，我们会在 [later in this chapter](https://www.elastic.co/guide/cn/elasticsearch/guide/current/scroll.html) 进行讨论。

### 搜索选项

有几个 查询参数可以影响搜索过程。

**偏好**

偏好这个参数 preference 允许 用来控制由哪些分片或节点来处理搜索请求。 它接受像 \_primary,\_primary\_first, \_local, \_only\_node:xyz, \_prefer\_node:xyz, 和 \_shards:2,3 这样的值, 这些值在[search preference](https://www.elastic.co/guide/en/elasticsearch/reference/master/search-request-preference.html) 文档页面被详细解释。

但是最有用的值是某些随机字符串，它可以避免 bouncing results 问题。

**Bouncing Results**

想象一下有两个文档有同样值的时间戳字段，搜索结果用 timestamp 字段来排序。 由于搜索请求是在所有有效的分片副本间轮询的，那就有可能发生主分片处理请求时，这两个文档是一种顺序， 而副本分片处理请求时又是另一种顺序。

这就是所谓的 bouncing results 问题: 每次用户刷新页面，搜索结果表现是不同的顺序。 让同一个用户始终使用同一个分片，这样可以避免这种问题， 可以设置 preference 参数为一个特定的任意值比如用户会话ID来解决。

**超时问题**

通常分片处理完它所有的数据后再把结果返回给协同节点，协同节点把收到的所有结果合并为最终结果。

这意味着花费的时间是最慢分片的处理时间加结果合并的时间。如果有一个节点有问题，就会导致所有的响应缓慢。

参数 timeout 告诉 分片允许处理数据的最大时间。如果没有足够的时间处理所有数据，这个分片的结果可以是部分的，甚至是空数据。

搜索的返回结果会用属性 timed\_out 标明分片是否返回的是部分结果：

...

"timed\_out": true,

...

|  |  |
| --- | --- |
|  | 这个搜索请求超时了。 |

超时仍然是一个最有效的操作，知道这一点很重要； 很可能查询会超过设定的超时时间。这种行为有两个原因：

1. 超时检查是基于每文档做的。 但是某些查询类型有大量的工作在文档评估之前需要完成。 这种 "setup" 阶段并不考虑超时设置，所以太长的建立时间会导致超过超时时间的整体延迟。
2. 因为时间检查是基于每个文档的，一次长时间查询在单个文档上执行并且在下个文档被评估之前不会超时。 这也意味着差的脚本（比如带无限循环的脚本）将会永远执行下去。

**路由**

在 [路由一个文档到一个分片中](https://www.elastic.co/guide/cn/elasticsearch/guide/current/routing-value.html) 中, 我们解释过如何定制参数 routing ，它能够在索引时提供来确保相关的文档，比如属于某个用户的文档被存储在某个分片上。 在搜索的时候，不用搜索索引的所有分片，而是通过指定几个 routing 值来限定只搜索几个相关的分片：

GET /\_search?routing=user\_1,user2

这个技术在设计大规模搜索系统时就会派上用场，我们在 [扩容设计](https://www.elastic.co/guide/cn/elasticsearch/guide/current/scale.html) 中详细讨论它。

**搜索类型**

缺省的搜索类型是 query\_then\_fetch 。 在某些情况下，你可能想明确设置 search\_type 为 dfs\_query\_then\_fetch 来改善相关性精确度：

GET /\_search?search\_type=dfs\_query\_then\_fetch

搜索类型 dfs\_query\_then\_fetch 有预查询阶段，这个阶段可以从所有相关分片获取词频来计算全局词频。 我们在 [被破坏的相关度！](https://www.elastic.co/guide/cn/elasticsearch/guide/current/relevance-is-broken.html) 会再讨论它。

### 游标查询 Scroll

scroll 查询 可以用来对 Elasticsearch 有效地执行大批量的文档查询，而又不用付出深度分页那种代价。

游标查询允许我们 先做查询初始化，然后再批量地拉取结果。 这有点儿像传统数据库中的 cursor 。

游标查询会取某个时间点的快照数据。 查询初始化之后索引上的任何变化会被它忽略。 它通过保存旧的数据文件来实现这个特性，结果就像保留初始化时的索引 视图 一样。

深度分页的代价根源是结果集全局排序，如果去掉全局排序的特性的话查询结果的成本就会很低。 游标查询用字段 \_doc 来排序。 这个指令让 Elasticsearch 仅仅从还有结果的分片返回下一批结果。

启用游标查询可以通过在查询的时候设置参数 scroll 的值为我们期望的游标查询的过期时间。 游标查询的过期时间会在每次做查询的时候刷新，所以这个时间只需要足够处理当前批的结果就可以了，而不是处理查询结果的所有文档的所需时间。 这个过期时间的参数很重要，因为保持这个游标查询窗口需要消耗资源，所以我们期望如果不再需要维护这种资源就该早点儿释放掉。 设置这个超时能够让 Elasticsearch 在稍后空闲的时候自动释放这部分资源。

GET /old\_index/\_search?scroll=1m

{

"query": { "match\_all": {}},

"sort" : ["\_doc"],

"size": 1000

}

|  |  |
| --- | --- |
|  | 保持游标查询窗口一分钟。 |
|  | 关键字 \_doc 是最有效的排序顺序。 |

这个查询的返回结果包括一个字段 \_scroll\_id`， 它是一个base64编码的长字符串 ((("scroll\_id"))) 。 现在我们能传递字段 `\_scroll\_id 到 \_search/scroll 查询接口获取下一批结果：

GET /\_search/scroll

{

"scroll": "1m",

"scroll\_id" : "cXVlcnlUaGVuRmV0Y2g7NTsxMDk5NDpkUmpiR2FjOFNhNnlCM1ZDMWpWYnRROzEwOTk1OmRSamJHYWM4U2E2eUIzVkMxalZidFE7MTA5OTM6ZFJqYkdhYzhTYTZ5QjNWQzFqVmJ0UTsxMTE5MDpBVUtwN2lxc1FLZV8yRGVjWlI2QUVBOzEwOTk2OmRSamJHYWM4U2E2eUIzVkMxalZidFE7MDs="

}

|  |  |
| --- | --- |
|  | 注意再次设置游标查询过期时间为一分钟。 |

这个游标查询返回的下一批结果。 尽管我们指定字段 size 的值为1000，我们有可能取到超过这个值数量的文档。 当查询的时候， 字段 size 作用于单个分片，所以每个批次实际返回的文档数量最大为 size \* number\_of\_primary\_shards 。

注意游标查询每次返回一个新字段 \_scroll\_id`。每次我们做下一次游标查询， 我们必须把前一次查询返回的字段 `\_scroll\_id 传递进去。 当没有更多的结果返回的时候，我们就处理完所有匹配的文档了。

提示：某些官方的 Elasticsearch 客户端比如 [Python 客户端](http://elasticsearch-py.readthedocs.org/en/master/helpers.html#scan) 和 [Perl 客户端](https://metacpan.org/pod/Search::Elasticsearch::Scroll) 提供了这个功能易用的封装。

## 索引管理

我们已经看到 Elasticsearch 让开发一个新的应用变得简单，不需要任何预先计划或设置。 不过，要不了多久你就会开始想要优化索引和搜索过程，以便更好地适合您的特定用例。 这些定制几乎围绕着索引和类型的方方面面，在本章，我们将介绍管理索引和类型映射的 API 以及一些最重要的设置。

### 创建一个索引

到目前为止, 我们已经通过索引一篇文档创建了一个新的索引 。这个索引采用的是默认的配置，新的字段通过动态映射的方式被添加到类型映射。现在我们需要对这个建立索引的过程做更多的控制：我们想要确保这个索引有数量适中的主分片，并且在我们索引任何数据 *之前* ，分析器和映射已经被建立好。

为了达到这个目的，我们需要手动创建索引，在请求体里面传入设置或类型映射，如下所示：

PUT /my\_index

{

"settings": { ... any settings ... },

"mappings": {

"type\_one": { ... any mappings ... },

"type\_two": { ... any mappings ... },

...

}

}

如果你想禁止自动创建索引，你 可以通过在 config/elasticsearch.yml 的每个节点下添加下面的配置：

action.auto\_create\_index: false

我们会在之后讨论你怎么用 [索引模板](https://www.elastic.co/guide/cn/elasticsearch/guide/current/index-templates.html) 来预配置开启自动创建索引。这在索引日志数据的时候尤其有用：你将日志数据索引在一个以日期结尾命名的索引上，子夜时分，一个预配置的新索引将会自动进行创建。

### 删除一个索引

用以下的请求来 删除索引:

DELETE /my\_index

你也可以这样删除多个索引：

DELETE /index\_one,index\_two

DELETE /index\_\*

你甚至可以这样删除 *全部* 索引：

DELETE /\_all

DELETE /\*

对一些人来说，能够用单个命令来删除所有数据可能会导致可怕的后果。如果你想要避免意外的大量删除, 你可以在你的 elasticsearch.yml 做如下配置：

action.destructive\_requires\_name: true

这个设置使删除只限于特定名称指向的数据, 而不允许通过指定 \_all 或通配符来删除指定索引库。你同样可以通过 [Cluster State API](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_changing_settings_dynamically.html) 动态的更新这个设置。

### 索引设置

你可以通过修改配置来自定义索引行为，详细配置参照 [索引模块](https://www.elastic.co/guide/en/elasticsearch/reference/master/index-modules.html)

Elasticsearch 提供了优化好的默认配置。 除非你理解这些配置的作用并且知道为什么要去修改，否则不要随意修改。

下面是两个 最重要的设置：

number\_of\_shards

每个索引的主分片数，默认值是 5 。这个配置在索引创建后不能修改。

number\_of\_replicas

每个主分片的副本数，默认值是 1 。对于活动的索引库，这个配置可以随时修改。

例如，我们可以创建只有 一个主分片，没有副本的小索引：

PUT /my\_temp\_index

{

"settings": {

"number\_of\_shards" : 1,

"number\_of\_replicas" : 0

}

}

然后，我们可以用 update-index-settings API 动态修改副本数：

PUT /my\_temp\_index/\_settings

{

"number\_of\_replicas": 1

}

### 配置分析器

第三个重要的索引设置是 analysis 部分， 用来配置已存在的分析器或针对你的索引创建新的自定义分析器。

在 [分析与分析器](https://www.elastic.co/guide/cn/elasticsearch/guide/current/analysis-intro.html) ，我们介绍了一些内置的 分析器，用于将全文字符串转换为适合搜索的倒排索引。

standard 分析器是用于全文字段的默认分析器， 对于大部分西方语系来说是一个不错的选择。 它包括了以下几点：

* standard 分词器，通过单词边界分割输入的文本。
* standard 语汇单元过滤器，目的是整理分词器触发的语汇单元（但是目前什么都没做）。
* lowercase 语汇单元过滤器，转换所有的语汇单元为小写。
* stop 语汇单元过滤器，删除停用词--对搜索相关性影响不大的常用词，如 a ， the ， and ， is。

默认情况下，停用词过滤器是被禁用的。如需启用它，你可以通过创建一个基于 standard 分析器的自定义分析器并设置 stopwords 参数。 可以给分析器提供一个停用词列表，或者告知使用一个基于特定语言的预定义停用词列表。

在下面的例子中，我们创建了一个新的分析器，叫做 es\_std ， 并使用预定义的 西班牙语停用词列表：

PUT /spanish\_docs

{

"settings": {

"analysis": {

"analyzer": {

"es\_std": {

"type": "standard",

"stopwords": "\_spanish\_"

}

}

}

}

}

es\_std 分析器不是全局的--它仅仅存在于我们定义的 spanish\_docs 索引中。 为了使用 analyze API来对它进行测试，我们必须使用特定的索引名：

GET /spanish\_docs/\_analyze?analyzer=es\_std

El veloz zorro marrón

简化的结果显示西班牙语停用词 El 已被正确的移除：

{

"tokens" : [

{ "token" : "veloz", "position" : 2 },

{ "token" : "zorro", "position" : 3 },

{ "token" : "marrón", "position" : 4 }

]

}

### 自定义分析器

虽然Elasticsearch带有一些现成的分析器，然而在分析器上Elasticsearch真正的强大之处在于，你可以通过在一个适合你的特定数据的设置之中组合字符过滤器、分词器、词汇单元过滤器来创建自定义的分析器。

在 [分析与分析器](https://www.elastic.co/guide/cn/elasticsearch/guide/current/analysis-intro.html) 我们说过，一个 分析器 就是在一个包里面组合了三种函数的一个包装器， 三种函数按照顺序被执行:

字符过滤器

字符过滤器 用来 整理 一个尚未被分词的字符串。例如，如果我们的文本是HTML格式的，它会包含像 <p> 或者 <div> 这样的HTML标签，这些标签是我们不想索引的。我们可以使用 [html清除 字符过滤器](https://www.elastic.co/guide/en/elasticsearch/reference/master/analysis-htmlstrip-charfilter.html)来移除掉所有的HTML标签，并且像把 &Aacute; 转换为相对应的Unicode字符 Á 这样，转换HTML实体。

一个分析器可能有0个或者多个字符过滤器。

分词器

一个分析器 必须 有一个唯一的分词器。 分词器把字符串分解成单个词条或者词汇单元。 标准 分析器里使用的 [标准 分词器](https://www.elastic.co/guide/en/elasticsearch/reference/master/analysis-standard-tokenizer.html) 把一个字符串根据单词边界分解成单个词条，并且移除掉大部分的标点符号，然而还有其他不同行为的分词器存在。

例如， [关键词 分词器](https://www.elastic.co/guide/en/elasticsearch/reference/master/analysis-keyword-tokenizer.html) 完整地输出 接收到的同样的字符串，并不做任何分词。 [空格 分词器](https://www.elastic.co/guide/en/elasticsearch/reference/master/analysis-whitespace-tokenizer.html) 只根据空格分割文本 。 [正则 分词器](https://www.elastic.co/guide/en/elasticsearch/reference/master/analysis-pattern-tokenizer.html) 根据匹配正则表达式来分割文本 。

词单元过滤器

经过分词，作为结果的 词单元流 会按照指定的顺序通过指定的词单元过滤器 。

词单元过滤器可以修改、添加或者移除词单元。我们已经提到过 [lowercase](http://www.elastic.co/guide/en/elasticsearch/reference/current/analysis-lowercase-tokenizer.html)和 [stop 词过滤器](https://www.elastic.co/guide/en/elasticsearch/reference/master/analysis-stop-tokenfilter.html) ，但是在 Elasticsearch 里面还有很多可供选择的词单元过滤器。 [词干过滤器](https://www.elastic.co/guide/en/elasticsearch/reference/master/analysis-stemmer-tokenfilter.html) 把单词 遏制 为 词干。 [ascii\_folding 过滤器](https://www.elastic.co/guide/en/elasticsearch/reference/master/analysis-asciifolding-tokenfilter.html)移除变音符，把一个像 "très" 这样的词转换为 "tres" 。 [ngram](https://www.elastic.co/guide/en/elasticsearch/reference/master/analysis-ngram-tokenfilter.html) 和 [edge\_ngram 词单元过滤器](https://www.elastic.co/guide/en/elasticsearch/reference/master/analysis-edgengram-tokenfilter.html) 可以产生 适合用于部分匹配或者自动补全的词单元。

在 [深入搜索](https://www.elastic.co/guide/cn/elasticsearch/guide/current/search-in-depth.html)，我们讨论了在哪里使用，以及怎样使用分词器和过滤器。但是首先，我们需要解释一下怎样创建自定义的分析器。

**创建一个自定义分析器**

和我们之前配置 es\_std 分析器一样，我们可以在 analysis 下的相应位置设置字符过滤器、分词器和词单元过滤器:

PUT /my\_index

{

"settings": {

"analysis": {

"char\_filter": { ... custom character filters ... },

"tokenizer": { ... custom tokenizers ... },

"filter": { ... custom token filters ... },

"analyzer": { ... custom analyzers ... }

}

}

}

作为示范，让我们一起来创建一个自定义分析器吧，这个分析器可以做到下面的这些事:

1. 使用 html清除 字符过滤器移除HTML部分。
2. 使用一个自定义的 映射 字符过滤器把 & 替换为 " 和 " ：
3. "char\_filter": {
4. "&\_to\_and": {
5. "type": "mapping",
6. "mappings": [ "&=> and "]
7. }

}

1. 使用 标准 分词器分词。
2. 小写词条，使用 小写 词过滤器处理。
3. 使用自定义 停止 词过滤器移除自定义的停止词列表中包含的词：
4. "filter": {
5. "my\_stopwords": {
6. "type": "stop",
7. "stopwords": [ "the", "a" ]
8. }

}

我们的分析器定义用我们之前已经设置好的自定义过滤器组合了已经定义好的分词器和过滤器：

"analyzer": {

"my\_analyzer": {

"type": "custom",

"char\_filter": [ "html\_strip", "&\_to\_and" ],

"tokenizer": "standard",

"filter": [ "lowercase", "my\_stopwords" ]

}

}

汇总起来，完整的 创建索引 请求 看起来应该像这样：

PUT /my\_index

{

"settings": {

"analysis": {

"char\_filter": {

"&\_to\_and": {

"type": "mapping",

"mappings": [ "&=> and "]

}},

"filter": {

"my\_stopwords": {

"type": "stop",

"stopwords": [ "the", "a" ]

}},

"analyzer": {

"my\_analyzer": {

"type": "custom",

"char\_filter": [ "html\_strip", "&\_to\_and" ],

"tokenizer": "standard",

"filter": [ "lowercase", "my\_stopwords" ]

}}

}}}

索引被创建以后，使用 analyze API 来 测试这个新的分析器：

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

The quick & brown fox

下面的缩略结果展示出我们的分析器正在正确地运行：

{

"tokens" : [

{ "token" : "quick", "position" : 2 },

{ "token" : "and", "position" : 3 },

{ "token" : "brown", "position" : 4 },

{ "token" : "fox", "position" : 5 }

]

}

这个分析器现在是没有多大用处的，除非我们告诉 Elasticsearch在哪里用上它。我们可以像下面这样把这个分析器应用在一个 string 字段上：

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

{

"properties": {

"title": {

"type": "string",

"analyzer": "my\_analyzer"

}

}

}

### 类型和映射

类型 在 Elasticsearch 中表示一类相似的文档。 类型由 名称 —比如 user 或 blogpost —和 映射 组成。

映射, 就像数据库中的 schema ，描述了文档可能具有的字段或 属性 、 每个字段的数据类型—比如 string, integer 或 date —以及Lucene是如何索引和存储这些字段的。

类型可以很好的抽象划分相似但不相同的数据。但由于 Lucene 的处理方式，类型的使用有些限制。

**Lucene 如何处理文档**

在 Lucene 中，一个文档由一组简单的键值对组成。 每个字段都可以有多个值，但至少要有一个值。 类似的，一个字符串可以通过分析过程转化为多个值。Lucene 不关心这些值是字符串、数字或日期--所有的值都被当做 不透明字节 。

当我们在 Lucene 中索引一个文档时，每个字段的值都被添加到相关字段的倒排索引中。你也可以将未处理的原始数据 存储 起来，以便这些原始数据在之后也可以被检索到。

**类型是如何实现的**

Elasticsearch 类型是 以 Lucene 处理文档的这个方式为基础来实现的。一个索引可以有多个类型，这些类型的文档可以存储在相同的索引中。

Lucene 没有文档类型的概念，每个文档的类型名被存储在一个叫 \_type 的元数据字段上。 当我们要检索某个类型的文档时, Elasticsearch 通过在 \_type 字段上使用过滤器限制只返回这个类型的文档。

Lucene 也没有映射的概念。 映射是 Elasticsearch 将复杂 JSON 文档 映射 成 Lucene 需要的扁平化数据的方式。

例如，在 user 类型中， name 字段的映射可以声明这个字段是 string 类型，并且它的值被索引到名叫 name 的倒排索引之前，需要通过 whitespace 分词器分析：

"name": {

"type": "string",

"analyzer": "whitespace"

}

**避免类型陷阱**

这导致了一个有趣的思想实验： 如果有两个不同的类型，每个类型都有同名的字段，但映射不同（例如：一个是字符串一个是数字），将会出现什么情况？

简单回答是，Elasticsearch 不会允许你定义这个映射。当你配置这个映射时，将会出现异常。

详细回答是，每个 Lucene 索引中的所有字段都包含一个单一的、扁平的模式。一个特定字段可以映射成 string 类型也可以是 number 类型，但是不能两者兼具。因为类型是 Elasticsearch 添加的 优于 Lucene 的额外机制（以元数据 \_type 字段的形式），在 Elasticsearch 中的所有类型最终都共享相同的映射。

以 data 索引中两种类型的映射为例：

{

"data": {

"mappings": {

"people": {

"properties": {

"name": {

"type": "string",

},

"address": {

"type": "string"

}

}

},

"transactions": {

"properties": {

"timestamp": {

"type": "date",

"format": "strict\_date\_optional\_time"

},

"message": {

"type": "string"

}

}

}

}

}

}

每个类型定义两个字段 (分别是 "name"/"address" 和 "timestamp"/"message" )。它们看起来是相互独立的，但在后台 Lucene 将创建一个映射，如:

{

"data": {

"mappings": {

"\_type": {

"type": "string",

"index": "not\_analyzed"

},

"name": {

"type": "string"

}

"address": {

"type": "string"

}

"timestamp": {

"type": "long"

}

"message": {

"type": "string"

}

}

}

}

注: 这不是真实有效的映射语法，只是用于演示

对于整个索引，映射在本质上被 扁平化 成一个单一的、全局的模式。这就是为什么两个类型不能定义冲突的字段：当映射被扁平化时，Lucene 不知道如何去处理。

**类型结论**

那么，这个讨论的结论是什么？技术上讲，多个类型可以在相同的索引中存在，只要它们的字段不冲突（要么因为字段是互为独占模式，要么因为它们共享相同的字段）。

重要的一点是: 类型可以很好的区分同一个集合中的不同细分。在不同的细分中数据的整体模式是相同的（或相似的）。

类型不适合 完全不同类型的数据 。如果两个类型的字段集是互不相同的，这就意味着索引中将有一半的数据是空的（字段将是 稀疏的 ），最终将导致性能问题。在这种情况下，最好是使用两个单独的索引。

总结：

* **正确:** 将 kitchen 和 lawn-care 类型放在 products 索引中, 因为这两种类型基本上是相同的模式
* **错误:** 将 products 和 logs 类型放在 data 索引中, 因为这两种类型互不相同。应该将它们放在不同的索引中。

### 根对象

映射的最高一层被称为 根对象 ，它可能包含下面几项：

* 一个 properties 节点，列出了文档中可能包含的每个字段的映射
* 各种元数据字段，它们都以一个下划线开头，例如 \_type 、 \_id 和 \_source
* 设置项，控制如何动态处理新的字段，例如 analyzer 、 dynamic\_date\_formats 和dynamic\_templates
* 其他设置，可以同时应用在根对象和其他 object 类型的字段上，例如 enabled 、 dynamic 和 include\_in\_all

**属性**

我们已经在 [核心简单域类型](https://www.elastic.co/guide/cn/elasticsearch/guide/current/mapping-intro.html#core-fields) 和 [复杂核心域类型](https://www.elastic.co/guide/cn/elasticsearch/guide/current/complex-core-fields.html) 章节中介绍过文档字段和属性的三个 最重要的设置：

type

字段的数据类型，例如 string 或 date

index

字段是否应当被当成全文来搜索（ analyzed ），或被当成一个准确的值（ not\_analyzed ），还是完全不可被搜索（ no ）

analyzer

确定在索引和搜索时全文字段使用的 analyzer

我们将在本书的后续部分讨论其他字段类型，例如 ip 、 geo\_point 和 geo\_shape 。

**元数据: \_source 字段**

默认地，Elasticsearch 在 \_source 字段存储代表文档体的JSON字符串。和所有被存储的字段一样， \_source 字段在被写入磁盘之前先会被压缩。

这个字段的存储几乎总是我们想要的，因为它意味着下面的这些：

* 搜索结果包括了整个可用的文档——不需要额外的从另一个的数据仓库来取文档。
* 如果没有 \_source 字段，部分 update 请求不会生效。
* 当你的映射改变时，你需要重新索引你的数据，有了\_source字段你可以直接从Elasticsearch这样做，而不必从另一个（通常是速度更慢的）数据仓库取回你的所有文档。
* 当你不需要看到整个文档时，单个字段可以从 \_source 字段提取和通过 get 或者 search 请求返回。
* 调试查询语句更加简单，因为你可以直接看到每个文档包括什么，而不是从一列id猜测它们的内容。

然而，存储 \_source 字段的确要使用磁盘空间。如果上面的原因对你来说没有一个是重要的，你可以用下面的映射禁用 \_source 字段：

PUT /my\_index

{

"mappings": {

"my\_type": {

"\_source": {

"enabled": false

}

}

}

}

在一个搜索请求里，你可以通过在请求体中指定 \_source 参数，来达到只获取特定的字段的效果：

GET /\_search

{

"query": { "match\_all": {}},

"\_source": [ "title", "created" ]

}

这些字段的值会从 \_source 字段被提取和返回，而不是返回整个 \_source 。

**Stored Fields 被存储字段**

为了之后的检索，除了索引一个字段的值，你 还可以选择 存储 原始字段值。有 Lucene 使用背景的用户使用被存储字段来选择他们想要在搜索结果里面返回的字段。事实上， \_source 字段就是一个被存储的字段。

在Elasticsearch中，对文档的个别字段设置存储的做法通常不是最优的。整个文档已经被存储为 \_source 字段。使用 \_source 参数提取你需要的字段总是更好的。

**元数据: \_all 字段**

在 [轻量 搜索](https://www.elastic.co/guide/cn/elasticsearch/guide/current/search-lite.html) 中，我们介绍了 \_all 字段：一个把其它字段值 当作一个大字符串来索引的特殊字段。 query\_string 查询子句(搜索 ?q=john )在没有指定字段时默认使用 \_all 字段。

\_all 字段在新应用的探索阶段，当你还不清楚文档的最终结构时是比较有用的。你可以使用这个字段来做任何查询，并且有很大可能找到需要的文档：

GET /\_search

{

"match": {

"\_all": "john smith marketing"

}

}

随着应用的发展，搜索需求变得更加明确，你会发现自己越来越少使用 \_all 字段。 \_all 字段是搜索的应急之策。通过查询指定字段，你的查询更加灵活、强大，你也可以对相关性最高的搜索结果进行更细粒度的控制。

[relevance algorithm](https://www.elastic.co/guide/cn/elasticsearch/guide/current/relevance-intro.html) 考虑的一个最重要的原则是字段的长度：字段越短越重要。 在较短的 title 字段中出现的短语可能比在较长的 content 字段中出现的短语更加重要。字段长度的区别在 \_all 字段中不会出现。

如果你不再需要 \_all 字段，你可以通过下面的映射来禁用：

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

{

"my\_type": {

"\_all": { "enabled": false }

}

}

通过 include\_in\_all 设置来逐个控制字段是否要包含在 \_all 字段中，默认值是 true`。在一个对象(或根对象)上设置 `include\_in\_all 可以修改这个对象中的所有字段的默认行为。

你可能想要保留 \_all 字段作为一个只包含某些特定字段的全文字段，例如只包含 title`，`overview`，`summary 和 tags`。 相对于完全禁用 `\_all 字段，你可以为所有字段默认禁用 include\_in\_all 选项，仅在你选择的字段上启用：

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

{

"my\_type": {

"include\_in\_all": false,

"properties": {

"title": {

"type": "string",

"include\_in\_all": true

},

...

}

}

}

记住，\_all 字段仅仅是一个 经过分词的 string 字段。它使用默认分词器来分析它的值，不管这个值原本所在字段指定的分词器。就像所有 string 字段，你可以配置 \_all 字段使用的分词器：

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

{

"my\_type": {

"\_all": { "analyzer": "whitespace" }

}

}

**元数据：文档标识**

文档标识与四个元数据字段 相关：

\_id

文档的 ID 字符串

\_type

文档的类型名

\_index

文档所在的索引

\_uid

\_type 和 \_id 连接在一起构造成 type#id

默认情况下， \_uid 字段是被存储（可取回）和索引（可搜索）的。 \_type 字段被索引但是没有存储，\_id 和 \_index 字段则既没有被索引也没有被存储，这意味着它们并不是真实存在的。

尽管如此，你仍然可以像真实字段一样查询 \_id 字段。Elasticsearch 使用 \_uid 字段来派生出 \_id 。 虽然你可以修改这些字段的 index 和 store 设置，但是基本上不需要这么做。

### 动态映射

当 Elasticsearch 遇到文档中以前 未遇到的字段，它用 [dynamic mapping](https://www.elastic.co/guide/cn/elasticsearch/guide/current/mapping-intro.html) 来确定字段的数据类型并自动把新的字段添加到类型映射。

有时这是想要的行为有时又不希望这样。通常没有人知道以后会有什么新字段加到文档，但是又希望这些字段被自动的索引。也许你只想忽略它们。如果Elasticsearch是作为重要的数据存储，可能就会期望遇到新字段就会抛出异常，这样能及时发现问题。

幸运的是可以用 dynamic 配置来控制这种行为 ，可接受的选项如下：

true

动态添加新的字段--缺省

false

忽略新的字段

strict

如果遇到新字段抛出异常

配置参数 dynamic 可以用在根 object 或任何 object 类型的字段上。你可以将 dynamic 的默认值设置为 strict , 而只在指定的内部对象中开启它, 例如：

PUT /my\_index

{

"mappings": {

"my\_type": {

"dynamic": "strict",

"properties": {

"title": { "type": "string"},

"stash": {

"type": "object",

"dynamic": true

}

}

}

}

}

|  |  |
| --- | --- |
|  | 如果遇到新字段，对象 my\_type 就会抛出异常。 |
|  | 而内部对象 stash 遇到新字段就会动态创建新字段。 |

使用上述动态映射， 你可以给 stash 对象添加新的可检索的字段：

PUT /my\_index/my\_type/1

{

"title": "This doc adds a new field",

"stash": { "new\_field": "Success!" }

}

但是对根节点对象 my\_type 进行同样的操作会失败：

PUT /my\_index/my\_type/1

{

"title": "This throws a StrictDynamicMappingException",

"new\_field": "Fail!"

}

把 dynamic 设置为 false 一点儿也不会改变 \_source 的字段内容。 \_source 仍然包含被索引的整个JSON文档。只是新的字段不会被加到映射中也不可搜索。

### 自定义动态映射

如果你想在运行时增加新的字段，你可能会启用动态映射。 然而，有时候，动态映射 规则 可能不太智能。幸运的是，我们可以通过设置去自定义这些规则，以便更好的适用于你的数据。

**日期检测**

当 Elasticsearch 遇到一个新的字符串字段时，它会检测这个字段是否包含一个可识别的日期，比如 2014-01-01 。 如果它像日期，这个字段就会被作为 date 类型添加。否则，它会被作为 string 类型添加。

有些时候这个行为可能导致一些问题。想象下，你有如下这样的一个文档：

{ "note": "2014-01-01" }

假设这是第一次识别 note 字段，它会被添加为 date 字段。但是如果下一个文档像这样：

{ "note": "Logged out" }

这显然不是一个日期，但为时已晚。这个字段已经是一个日期类型，这个 不合法的日期 将会造成一个异常。

日期检测可以通过在根对象上设置 date\_detection 为 false 来关闭：

PUT /my\_index

{

"mappings": {

"my\_type": {

"date\_detection": false

}

}

}

使用这个映射，字符串将始终作为 string 类型。如果你需要一个 date 字段，你必须手动添加。

Elasticsearch 判断字符串为日期的规则可以通过 [dynamic\_date\_formats setting](https://www.elastic.co/guide/en/elasticsearch/reference/master/dynamic-field-mapping.html#date-detection) 来设置。

**动态模板**

使用 dynamic\_templates ，你可以完全控制 新检测生成字段的映射。你甚至可以通过字段名称或数据类型来应用不同的映射。

每个模板都有一个名称， 你可以用来描述这个模板的用途， 一个 mapping 来指定映射应该怎样使用，以及至少一个参数 (如 match) 来定义这个模板适用于哪个字段。

模板按照顺序来检测；第一个匹配的模板会被启用。例如，我们给 string 类型字段定义两个模板：

* es ：以 \_es 结尾的字段名需要使用 spanish 分词器。
* en ：所有其他字段使用 english 分词器。

我们将 es 模板放在第一位，因为它比匹配所有字符串字段的 en 模板更特殊：

PUT /my\_index

{

"mappings": {

"my\_type": {

"dynamic\_templates": [

{ "es": {

"match": "\*\_es",

"match\_mapping\_type": "string",

"mapping": {

"type": "string",

"analyzer": "spanish"

}

}},

{ "en": {

"match": "\*",

"match\_mapping\_type": "string",

"mapping": {

"type": "string",

"analyzer": "english"

}

}}

]

}}}

|  |  |
| --- | --- |
|  | 匹配字段名以 \_es 结尾的字段。 |
|  | 匹配其他所有字符串类型字段。 |

match\_mapping\_type 允许你应用模板到特定类型的字段上，就像有标准动态映射规则检测的一样， (例如 string 或 long)。

match 参数只匹配字段名称， path\_match 参数匹配字段在对象上的完整路径，所以 address.\*.name 将匹配这样的字段：

{

"address": {

"city": {

"name": "New York"

}

}

}

unmatch 和 path\_unmatch将被用于未被匹配的字段。

更多的配置选项见 [动态映射文档](https://www.elastic.co/guide/en/elasticsearch/reference/master/dynamic-mapping.html) 。

### 缺省映射

通常，一个索引中的所有类型共享相同的字段和设置。 \_default\_ 映射更加方便地指定通用设置，而不是每次创建新类型时都要重复设置。 \_default\_ 映射是新类型的模板。在设置 \_default\_ 映射之后创建的所有类型都将应用这些缺省的设置，除非类型在自己的映射中明确覆盖这些设置。

例如，我们可以使用 \_default\_ 映射为所有的类型禁用 \_all 字段， 而只在 blog 类型启用：

PUT /my\_index

{

"mappings": {

"\_default\_": {

"\_all": { "enabled": false }

},

"blog": {

"\_all": { "enabled": true }

}

}

}

\_default\_ 映射也是一个指定索引 [dynamic templates](https://www.elastic.co/guide/cn/elasticsearch/guide/current/custom-dynamic-mapping.html#dynamic-templates) 的好方法。

### 重新索引你的数据

尽管可以增加新的类型到索引中，或者增加新的字段到类型中，但是不能添加新的分析器或者对现有的字段做改动。 如果你那么做的话，结果就是那些已经被索引的数据就不正确， 搜索也不能正常工作。

对现有数据的这类改变最简单的办法就是重新索引：用新的设置创建新的索引并把文档从旧的索引复制到新的索引。

字段 \_source 的一个优点是在Elasticsearch中已经有整个文档。你不必从源数据中重建索引，而且那样通常比较慢。

为了有效的重新索引所有在旧的索引中的文档，用 [scroll](https://www.elastic.co/guide/cn/elasticsearch/guide/current/scroll.html) 从旧的索引检索批量文档 ， 然后用 [bulk API](https://www.elastic.co/guide/cn/elasticsearch/guide/current/bulk.html) 把文档推送到新的索引中。

从Elasticsearch v2.3.0开始， [Reindex API](https://www.elastic.co/guide/en/elasticsearch/reference/master/docs-reindex.html) 被引入。它能够对文档重建索引而不需要任何插件或外部工具。

**批量重新索引**

同时并行运行多个重建索引任务，但是你显然不希望结果有重叠。正确的做法是按日期或者时间 这样的字段作为过滤条件把大的重建索引分成小的任务：

GET /old\_index/\_search?scroll=1m

{

"query": {

"range": {

"date": {

"gte": "2014-01-01",

"lt": "2014-02-01"

}

}

},

"sort": ["\_doc"],

"size": 1000

}

如果旧的索引持续会有变化，你希望新的索引中也包括那些新加的文档。那就可以对新加的文档做重新索引， 但还是要用日期类字段过滤来匹配那些新加的文档。

### 索引别名和零停机

在前面提到的，重建索引的问题是必须更新应用中的索引名称。 索引别名就是用来解决这个问题的！

索引 *别名* 就像一个快捷方式或软连接，可以指向一个或多个索引，也可以给任何一个需要索引名的API来使用。别名 带给我们极大的灵活性，允许我们做下面这些：

* 在运行的集群中可以无缝的从一个索引切换到另一个索引
* 给多个索引分组 (例如， last\_three\_months)
* 给索引的一个子集创建 视图

在后面我们会讨论更多关于别名的使用。现在，我们将解释怎样使用别名在零停机下从旧索引切换到新索引。

有两种方式管理别名： \_alias 用于单个操作， \_aliases 用于执行多个原子级操作。

在本章中，我们假设你的应用有一个叫 my\_index 的索引。事实上， my\_index 是一个指向当前真实索引的别名。真实索引包含一个版本号： my\_index\_v1 ， my\_index\_v2 等等。

首先，创建索引 my\_index\_v1 ，然后将别名 my\_index 指向它：

PUT /my\_index\_v1

PUT /my\_index\_v1/\_alias/my\_index

|  |  |
| --- | --- |
|  | 创建索引 my\_index\_v1 。 |
|  | 设置别名 my\_index 指向 my\_index\_v1 。 |

你可以检测这个别名指向哪一个索引：

GET /\*/\_alias/my\_index

或哪些别名指向这个索引：

GET /my\_index\_v1/\_alias/\*

两者都会返回下面的结果：

{

"my\_index\_v1" : {

"aliases" : {

"my\_index" : { }

}

}

}

然后，我们决定修改索引中一个字段的映射。当然，我们不能修改现存的映射，所以我们必须重新索引数据。 首先, 我们用新映射创建索引 my\_index\_v2 ：

PUT /my\_index\_v2

{

"mappings": {

"my\_type": {

"properties": {

"tags": {

"type": "string",

"index": "not\_analyzed"

}

}

}

}

}

然后我们将数据从 my\_index\_v1 索引到 my\_index\_v2 ，下面的过程在 [重新索引你的数据](https://www.elastic.co/guide/cn/elasticsearch/guide/current/reindex.html) 中已经描述过。一旦我们确定文档已经被正确地重索引了，我们就将别名指向新的索引。

一个别名可以指向多个索引，所以我们在添加别名到新索引的同时必须从旧的索引中删除它。这个操作需要原子化，这意味着我们需要使用 \_aliases 操作：

POST /\_aliases

{

"actions": [

{ "remove": { "index": "my\_index\_v1", "alias": "my\_index" }},

{ "add": { "index": "my\_index\_v2", "alias": "my\_index" }}

]

}

你的应用已经在零停机的情况下从旧索引迁移到新索引了。

即使你认为现在的索引设计已经很完美了，在生产环境中，还是有可能需要做一些修改的。

做好准备：在你的应用中使用别名而不是索引名。然后你就可以在任何时候重建索引。别名的开销很小，应该广泛使用。

## 分片内部原理

在 [集群内的原理](https://www.elastic.co/guide/cn/elasticsearch/guide/current/distributed-cluster.html), 我们介绍了 分片, 并将它 描述成最小的 工作单元\_。但是究竟什么 \_是 一个分片，它是如何工作的？ 在这个章节，我们回答以下问题:

* 为什么搜索是 近 实时的？
* 为什么文档的 CRUD (创建-读取-更新-删除) 操作是 实时 的?
* Elasticsearch 是怎样保证更新被持久化在断电时也不丢失数据?
* 为什么删除文档不会立刻释放空间？
* refresh, flush, 和 optimize API 都做了什么, 你什么情况下应该是用他们？

最简单的理解一个分片如何工作的方式是上一堂历史课。 我们将要审视提供一个带近实时搜索和分析的 分布式持久化数据存储需要解决的问题。

**内容警告**

本章展示的这些信息仅供您兴趣阅读。为了使用 Elasticsearch 您并不需要理解和记忆所有的细节。 读这个章节是为了了解工作机制，并且为了将来您需要这些信息时，知道这些信息在哪里。但是不要被这些细节所累。

### 使文本可被搜索

必须解决的第一个挑战是如何 使文本可被搜索。 传统的数据库每个字段存储单个值，但这对全文检索并不够。文本字段中的每个单词需要被搜索，对数据库意味着需要单个字段有索引多值(这里指单词)的能力。

最好的支持 一个字段多个值 需求的数据结构是我们在 [倒排索引](https://www.elastic.co/guide/cn/elasticsearch/guide/current/inverted-index.html) 章节中介绍过的 倒排索引 。 倒排索引包含一个有序列表，列表包含所有文档出现过的不重复个体，或称为 词项 ，对于每一个词项，包含了它所有曾出现过文档的列表。

Term | Doc 1 | Doc 2 | Doc 3 | ...

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

brown | X | | X | ...

fox | X | X | X | ...

quick | X | X | | ...

the | X | | X | ...

当讨论倒排索引时，我们会谈到 文档 标引，因为历史原因，倒排索引被用来对整个非结构化文本文档进行标引。 Elasticsearch 中的 文档 是有字段和值的结构化 JSON 文档。事实上，在 JSON 文档中， 每个被索引的字段都有自己的倒排索引。

这个倒排索引相比特定词项出现过的文档列表，会包含更多其它信息。它会保存每一个词项出现过的文档总数， 在对应的文档中一个具体词项出现的总次数，词项在文档中的顺序，每个文档的长度，所有文档的平均长度，等等。这些统计信息允许 Elasticsearch 决定哪些词比其它词更重要，哪些文档比其它文档更重要，这些内容在 [什么是相关性?](https://www.elastic.co/guide/cn/elasticsearch/guide/current/relevance-intro.html) 中有描述。

为了能够实现预期功能，倒排索引需要知道集合中的 所有 文档，这是需要认识到的关键问题。

早期的全文检索会为整个文档集合建立一个很大的倒排索引并将其写入到磁盘。 一旦新的索引就绪，旧的就会被其替换，这样最近的变化便可以被检索到。

### 不变性

倒排索引被写入磁盘后是 不可改变 的:它永远不会修改。 不变性有重要的价值：

* 不需要锁。如果你从来不更新索引，你就不需要担心多进程同时修改数据的问题。
* 一旦索引被读入内核的文件系统缓存，便会留在哪里，由于其不变性。只要文件系统缓存中还有足够的空间，那么大部分读请求会直接请求内存，而不会命中磁盘。这提供了很大的性能提升。
* 其它缓存(像filter缓存)，在索引的生命周期内始终有效。它们不需要在每次数据改变时被重建，因为数据不会变化。
* 写入单个大的倒排索引允许数据被压缩，减少磁盘 I/O 和 需要被缓存到内存的索引的使用量。

当然，一个不变的索引也有不好的地方。主要事实是它是不可变的! 你不能修改它。如果你需要让一个新的文档 可被搜索，你需要重建整个索引。这要么对一个索引所能包含的数据量造成了很大的限制，要么对索引可被更新的频率造成了很大的限制。

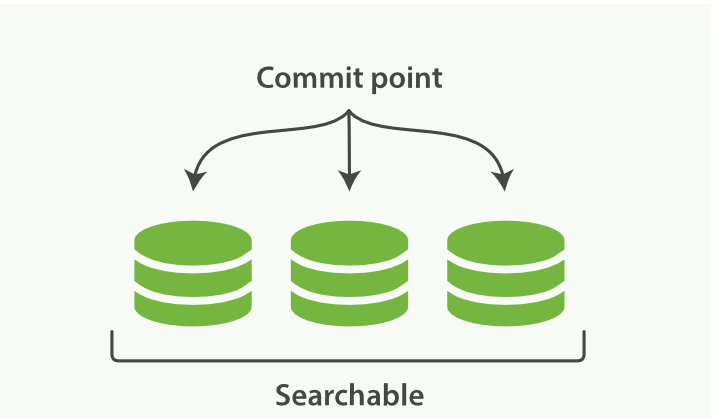
### 动态更新索引

下一个需要被解决的问题是怎样在保留不变性的前提下实现倒排索引的更新？ 答案是: 用更多的索引。

通过增加新的补充索引来反映新近的修改，而不是直接重写整个倒排索引。每一个倒排索引都会被轮流查询到--从最早的开始--查询完后再对结果进行合并。

Elasticsearch 基于 Lucene, 这个 java 库引入了 按段搜索 的概念。 每一 段 本身都是一个倒排索引， 但 索引 在 Lucene 中除表示所有 段 的集合外， 还增加了 提交点 的概念 — 一个列出了所有已知段的文件，就像在 [图 16 “一个 Lucene 索引包含一个提交点和三个段”](https://www.elastic.co/guide/cn/elasticsearch/guide/current/dynamic-indices.html#img-index-segments) 中描绘的那样。 如 [图 17 “一个在内存缓存中包含新文档的 Lucene 索引”](https://www.elastic.co/guide/cn/elasticsearch/guide/current/dynamic-indices.html#img-memory-buffer) 所示，新的文档首先被添加到内存索引缓存中，然后写入到一个基于磁盘的段，如 [图 18 “在一次提交后，一个新的段被添加到提交点而且缓存被清空。”](https://www.elastic.co/guide/cn/elasticsearch/guide/current/dynamic-indices.html#img-post-commit) 所示。

**图 16. 一个 Lucene 索引包含一个提交点和三个段**



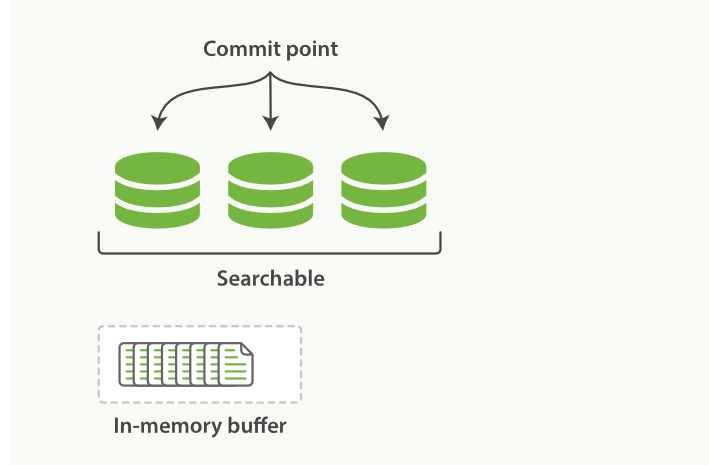
**索引与分片的比较**

被混淆的概念是，一个 Lucene 索引 我们在 Elasticsearch 称作 分片 。 一个 Elasticsearch 索引是分片的集合。 当 Elasticsearch 在索引中搜索的时候， 他发送查询到每一个属于索引的分片(Lucene 索引)，然后像 [执行分布式检索](https://www.elastic.co/guide/cn/elasticsearch/guide/current/distributed-search.html) 提到的那样，合并每个分片的结果到一个全局的结果集。

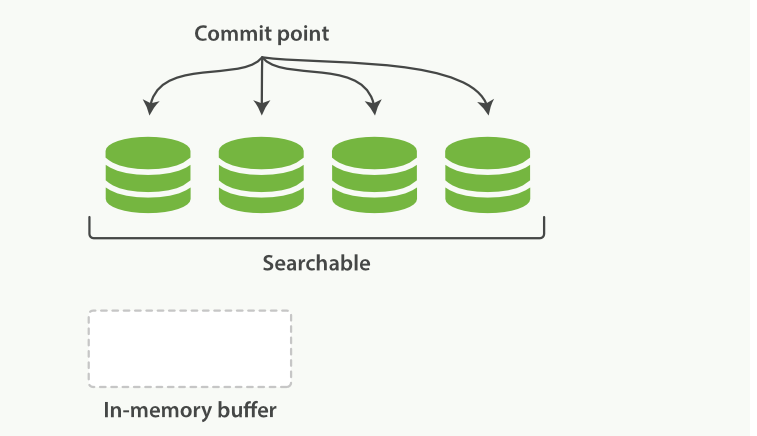
逐段搜索会以如下流程进行工作：

1. 新文档被收集到内存索引缓存， 见 [图 17 “一个在内存缓存中包含新文档的 Lucene 索引”](https://www.elastic.co/guide/cn/elasticsearch/guide/current/dynamic-indices.html#img-memory-buffer) 。
2. 不时地, 缓存被 提交 ：
   * 一个新的段--一个追加的倒排索引--被写入磁盘。
   * 一个新的包含新段名字的 提交点 被写入磁盘。
   * 磁盘进行 同步 — 所有在文件系统缓存中等待的写入都刷新到磁盘，以确保它们被写入物理文件。
3. 新的段被开启，让它包含的文档可见以被搜索。
4. 内存缓存被清空，等待接收新的文档。

**图 17. 一个在内存缓存中包含新文档的 Lucene 索引**



**图 18. 在一次提交后，一个新的段被添加到提交点而且缓存被清空。**



当一个查询被触发，所有已知的段按顺序被查询。词项统计会对所有段的结果进行聚合，以保证每个词和每个文档的关联都被准确计算。 这种方式可以用相对较低的成本将新文档添加到索引。

**删除和更新**

段是不可改变的，所以既不能从把文档从旧的段中移除，也不能修改旧的段来进行反映文档的更新。 取而代之的是，每个提交点会包含一个 .del 文件，文件中会列出这些被删除文档的段信息。

当一个文档被 “删除” 时，它实际上只是在 .del 文件中被 标记 删除。一个被标记删除的文档仍然可以被查询匹配到， 但它会在最终结果被返回前从结果集中移除。

文档更新也是类似的操作方式：当一个文档被更新时，旧版本文档被标记删除，文档的新版本被索引到一个新的段中。 可能两个版本的文档都会被一个查询匹配到，但被删除的那个旧版本文档在结果集返回前就已经被移除。

在 [段合并](https://www.elastic.co/guide/cn/elasticsearch/guide/current/merge-process.html) , 我们展示了一个被删除的文档是怎样被文件系统移除的。

### 近实时搜索

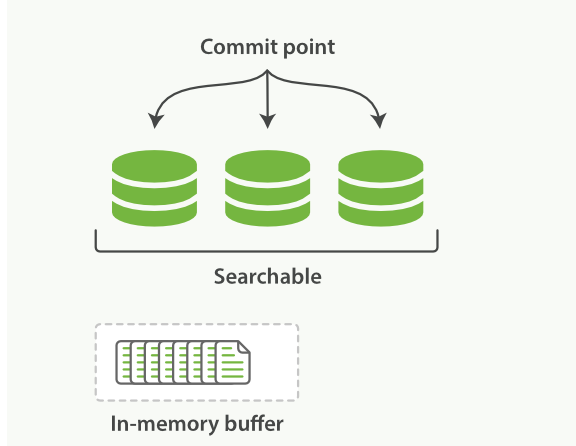
随着按段（per-segment）搜索的发展， 一个新的文档从索引到可被搜索的延迟显著降低了。新文档在几分钟之内即可被检索，但这样还是不够快。

磁盘在这里成为了瓶颈。 提交（Commiting）一个新的段到磁盘需要一个 [fsync](http://en.wikipedia.org/wiki/Fsync) 来确保段被物理性地写入磁盘，这样在断电的时候就不会丢失数据。 但是 fsync 操作代价很大; 如果每次索引一个文档都去执行一次的话会造成很大的性能问题。

我们需要的是一个更轻量的方式来使一个文档可被搜索，这意味着 fsync 要从整个过程中被移除。

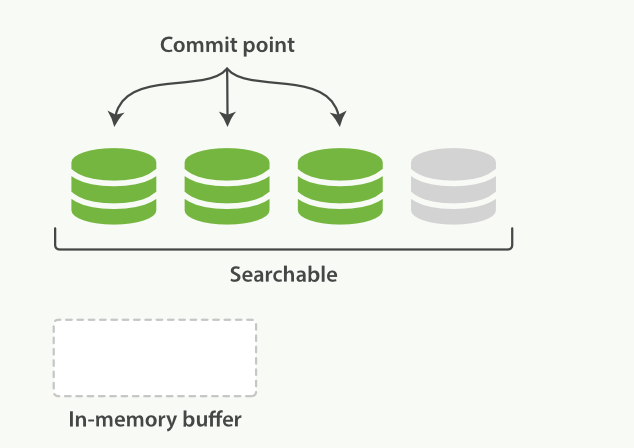
在Elasticsearch和磁盘之间是文件系统缓存。 像之前描述的一样， 在内存索引缓冲区（ [图 19 “在内存缓冲区中包含了新文档的 Lucene 索引”](https://www.elastic.co/guide/cn/elasticsearch/guide/current/near-real-time.html#img-pre-refresh) ）中的文档会被写入到一个新的段中（ [图 20 “缓冲区的内容已经被写入一个可被搜索的段中，但还没有进行提交”](https://www.elastic.co/guide/cn/elasticsearch/guide/current/near-real-time.html#img-post-refresh) ）。 但是这里新段会被先写入到文件系统缓存--这一步代价会比较低，稍后再被刷新到磁盘--这一步代价比较高。不过只要文件已经在缓存中， 就可以像其它文件一样被打开和读取了。

**图 19. 在内存缓冲区中包含了新文档的 Lucene 索引**



Lucene 允许新段被写入和打开--使其包含的文档在未进行一次完整提交时便对搜索可见。 这种方式比进行一次提交代价要小得多，并且在不影响性能的前提下可以被频繁地执行。

**图 20. 缓冲区的内容已经被写入一个可被搜索的段中，但还没有进行提交**



**refresh API**

在 Elasticsearch 中，写入和打开一个新段的轻量的过程叫做 refresh 。 默认情况下每个分片会每秒自动刷新一次。这就是为什么我们说 Elasticsearch 是 近 实时搜索: 文档的变化并不是立即对搜索可见，但会在一秒之内变为可见。

这些行为可能会对新用户造成困惑: 他们索引了一个文档然后尝试搜索它，但却没有搜到。这个问题的解决办法是用 refresh API 执行一次手动刷新:

POST /\_refresh

POST /blogs/\_refresh

|  |  |
| --- | --- |
|  | 刷新（Refresh）所有的索引。 |
|  | 只刷新（Refresh） blogs 索引。 |

尽管刷新是比提交轻量很多的操作，它还是会有性能开销。 当写测试的时候， 手动刷新很有用，但是不要在生产环境下每次索引一个文档都去手动刷新。 相反，你的应用需要意识到 Elasticsearch 的近实时的性质，并接受它的不足。

并不是所有的情况都需要每秒刷新。可能你正在使用 Elasticsearch 索引大量的日志文件， 你可能想优化索引速度而不是近实时搜索， 可以通过设置 refresh\_interval ， 降低每个索引的刷新频率：

PUT /my\_logs

{

"settings": {

"refresh\_interval": "30s"

}

}

|  |  |
| --- | --- |
|  | 每30秒刷新 my\_logs 索引。 |

refresh\_interval 可以在既存索引上进行动态更新。 在生产环境中，当你正在建立一个大的新索引时，可以先关闭自动刷新，待开始使用该索引时，再把它们调回来：

PUT /my\_logs/\_settings

{ "refresh\_interval": -1 }

PUT /my\_logs/\_settings

{ "refresh\_interval": "1s" }

|  |  |
| --- | --- |
|  | 关闭自动刷新。 |
|  | 每秒自动刷新。 |

refresh\_interval 需要一个 持续时间 值， 例如 1s （1 秒） 或 2m （2 分钟）。 一个绝对值 1 表示的是 1毫秒 --无疑会使你的集群陷入瘫痪。

### 持久化变更

如果没有用 fsync 把数据从文件系统缓存刷（flush）到硬盘，我们不能保证数据在断电甚至是程序正常退出之后依然存在。为了保证 Elasticsearch 的可靠性，需要确保数据变化被持久化到磁盘。

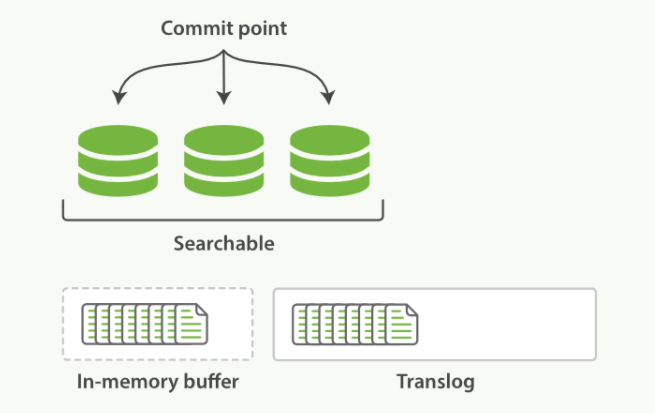
在 [动态更新索引](https://www.elastic.co/guide/cn/elasticsearch/guide/current/dynamic-indices.html)，我们说一次完整的提交会将段刷到磁盘，并写入一个包含所有段列表的提交点。Elasticsearch 在启动或重新打开一个索引的过程中使用这个提交点来判断哪些段隶属于当前分片。

即使通过每秒刷新（refresh）实现了近实时搜索，我们仍然需要经常进行完整提交来确保能从失败中恢复。但在两次提交之间发生变化的文档怎么办？我们也不希望丢失掉这些数据。

Elasticsearch 增加了一个 translog ，或者叫事务日志，在每一次对 Elasticsearch 进行操作时均进行了日志记录。通过 translog ，整个流程看起来是下面这样：

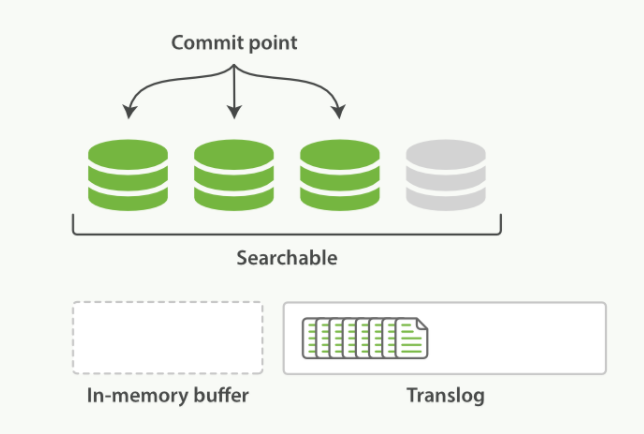
1. 一个文档被索引之后，就会被添加到内存缓冲区，并且 追加到了 translog ，正如 [图 21 “新的文档被添加到内存缓冲区并且被追加到了事务日志”](https://www.elastic.co/guide/cn/elasticsearch/guide/current/translog.html#img-xlog-pre-refresh) 描述的一样。

**图 21. 新的文档被添加到内存缓冲区并且被追加到了事务日志**



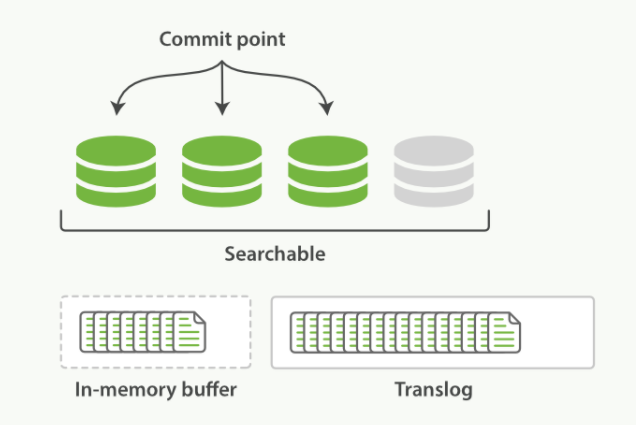
1. 刷新（refresh）使分片处于 [图 22 “刷新（refresh）完成后, 缓存被清空但是事务日志不会”](https://www.elastic.co/guide/cn/elasticsearch/guide/current/translog.html#img-xlog-post-refresh) 描述的状态，分片每秒被刷新（refresh）一次：
   * 这些在内存缓冲区的文档被写入到一个新的段中，且没有进行 fsync 操作。
   * 这个段被打开，使其可被搜索。
   * 内存缓冲区被清空。

**图 22. 刷新（refresh）完成后, 缓存被清空但是事务日志不会**



1. 这个进程继续工作，更多的文档被添加到内存缓冲区和追加到事务日志（见 [图 23 “事务日志不断积累文档”](https://www.elastic.co/guide/cn/elasticsearch/guide/current/translog.html#img-xlog-pre-flush) ）。

**图 23. 事务日志不断积累文档**

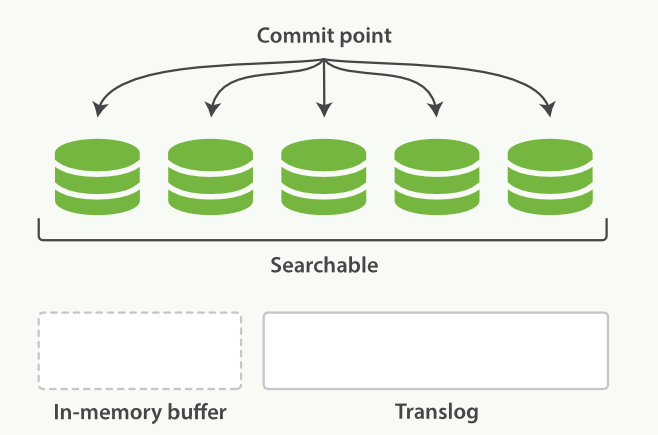


1. 每隔一段时间--例如 translog 变得越来越大--索引被刷新（flush）；一个新的 translog 被创建，并且一个全量提交被执行（见 [图 24 “在刷新（flush）之后，段被全量提交，并且事务日志被清空”](https://www.elastic.co/guide/cn/elasticsearch/guide/current/translog.html#img-xlog-post-flush)）：
   * 所有在内存缓冲区的文档都被写入一个新的段。
   * 缓冲区被清空。
   * 一个提交点被写入硬盘。
   * 文件系统缓存通过 fsync 被刷新（flush）。
   * 老的 translog 被删除。

translog 提供所有还没有被刷到磁盘的操作的一个持久化纪录。当 Elasticsearch 启动的时候， 它会从磁盘中使用最后一个提交点去恢复已知的段，并且会重放 translog 中所有在最后一次提交后发生的变更操作。

translog 也被用来提供实时 CRUD 。当你试着通过ID查询、更新、删除一个文档，它会在尝试从相应的段中检索之前， 首先检查 translog 任何最近的变更。这意味着它总是能够实时地获取到文档的最新版本。

**图 24. 在刷新（flush）之后，段被全量提交，并且事务日志被清空**



**flush API**

这个执行一个提交并且截断 translog 的行为在 Elasticsearch 被称作一次 flush 。 分片每30分钟被自动刷新（flush），或者在 translog 太大的时候也会刷新。请查看 [translog 文档](https://www.elastic.co/guide/en/elasticsearch/reference/master/index-modules-translog.html#_translog_settings) 来设置，它可以用来 控制这些阈值：

[flush API](https://www.elastic.co/guide/en/elasticsearch/reference/master/indices-flush.html) 可以 被用来执行一个手工的刷新（flush）:

POST /blogs/\_flush

POST /\_flush?wait\_for\_ongoing

|  |  |
| --- | --- |
|  | 刷新（flush） blogs 索引。 |
|  | 刷新（flush）所有的索引并且并且等待所有刷新在返回前完成。 |

你很少需要自己手动执行一个的 flush 操作；通常情况下，自动刷新就足够了。

这就是说，在重启节点或关闭索引之前执行 [flush](https://www.elastic.co/guide/cn/elasticsearch/guide/current/translog.html#flush-api) 有益于你的索引。当 Elasticsearch 尝试恢复或重新打开一个索引， 它需要重放 translog 中所有的操作，所以如果日志越短，恢复越快。

**Translog 有多安全?**

translog 的目的是保证操作不会丢失。这引出了这个问题： Translog 有多安全 ？

在文件被 fsync 到磁盘前，被写入的文件在重启之后就会丢失。默认 translog 是每 5 秒被 fsync 刷新到硬盘， 或者在每次写请求完成之后执行(e.g. index, delete, update, bulk)。这个过程在主分片和复制分片都会发生。最终， 基本上，这意味着在整个请求被 fsync 到主分片和复制分片的translog之前，你的客户端不会得到一个 200 OK 响应。

在每次请求后都执行一个 fsync 会带来一些性能损失，尽管实践表明这种损失相对较小（特别是bulk导入，它在一次请求中平摊了大量文档的开销）。

但是对于一些大容量的偶尔丢失几秒数据问题也并不严重的集群，使用异步的 fsync 还是比较有益的。比如，写入的数据被缓存到内存中，再每5秒执行一次 fsync 。

这个行为可以通过设置 durability 参数为 async 来启用：

PUT /my\_index/\_settings

{

"index.translog.durability": "async",

"index.translog.sync\_interval": "5s"

}

这个选项可以针对索引单独设置，并且可以动态进行修改。如果你决定使用异步 translog 的话，你需要 保证 在发生crash时，丢失掉 sync\_interval 时间段的数据也无所谓。请在决定前知晓这个特性。

如果你不确定这个行为的后果，最好是使用默认的参数（ "index.translog.durability": "request" ）来避免数据丢失。

### 段合并

由于自动刷新流程每秒会创建一个新的段 ，这样会导致短时间内的段数量暴增。而段数目太多会带来较大的麻烦。 每一个段都会消耗文件句柄、内存和cpu运行周期。更重要的是，每个搜索请求都必须轮流检查每个段；所以段越多，搜索也就越慢。

Elasticsearch通过在后台进行段合并来解决这个问题。小的段被合并到大的段，然后这些大的段再被合并到更大的段。

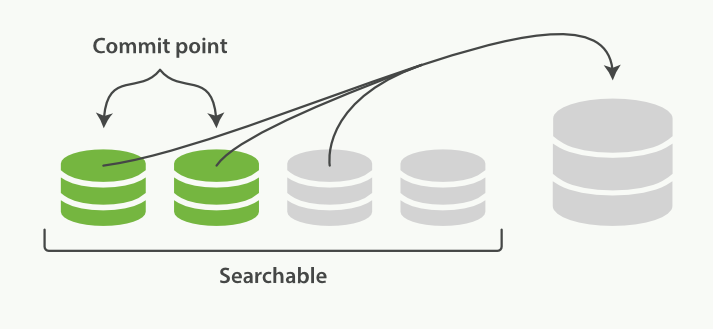
段合并的时候会将那些旧的已删除文档 从文件系统中清除。 被删除的文档（或被更新文档的旧版本）不会被拷贝到新的大段中。

启动段合并不需要你做任何事。进行索引和搜索时会自动进行。这个流程像在 [图 25 “两个提交了的段和一个未提交的段正在被合并到一个更大的段”](https://www.elastic.co/guide/cn/elasticsearch/guide/current/merge-process.html#img-merge) 中提到的一样工作：

1、 当索引的时候，刷新（refresh）操作会创建新的段并将段打开以供搜索使用。

2、 合并进程选择一小部分大小相似的段，并且在后台将它们合并到更大的段中。这并不会中断索引和搜索。

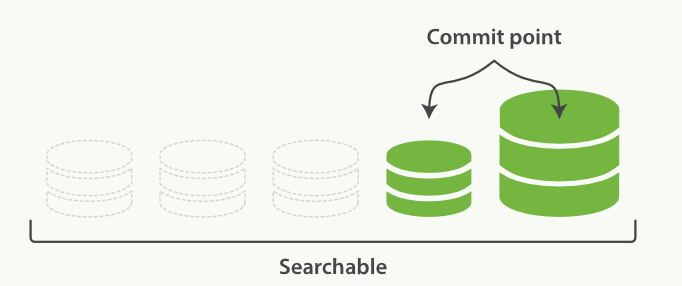
**图 25. 两个提交了的段和一个未提交的段正在被合并到一个更大的段**



3、 [图 26 “一旦合并结束，老的段被删除”](https://www.elastic.co/guide/cn/elasticsearch/guide/current/merge-process.html#img-post-merge) 说明合并完成时的活动：

* 新的段被刷新（flush）到了磁盘。   \*\* 写入一个包含新段且排除旧的和较小的段的新提交点。
* 新的段被打开用来搜索。
* 老的段被删除。

**图 26. 一旦合并结束，老的段被删除**



合并大的段需要消耗大量的I/O和CPU资源，如果任其发展会影响搜索性能。Elasticsearch在默认情况下会对合并流程进行资源限制，所以搜索仍然 有足够的资源很好地执行。

查看 [段和合并](https://www.elastic.co/guide/cn/elasticsearch/guide/current/indexing-performance.html#segments-and-merging) 来为你的实例获取关于合并调整的建议。

**optimize API**

optimize API大可看做是 强制合并 API 。它会将一个分片强制合并到 max\_num\_segments 参数指定大小的段数目。 这样做的意图是减少段的数量（通常减少到一个），来提升搜索性能。

optimize API 不应该 被用在一个动态索引————一个正在被活跃更新的索引。后台合并流程已经可以很好地完成工作。 optimizing 会阻碍这个进程。不要干扰它！

在特定情况下，使用 optimize API 颇有益处。例如在日志这种用例下，每天、每周、每月的日志被存储在一个索引中。 老的索引实质上是只读的；它们也并不太可能会发生变化。

在这种情况下，使用optimize优化老的索引，将每一个分片合并为一个单独的段就很有用了；这样既可以节省资源，也可以使搜索更加快速：

POST /logstash-2014-10/\_optimize?max\_num\_segments=1

|  |  |
| --- | --- |
|  | 合并索引中的每个分片为一个单独的段 |

请注意，使用 optimize API 触发段合并的操作一点也不会受到任何资源上的限制。这可能会消耗掉你节点上全部的I/O资源, 使其没有余裕来处理搜索请求，从而有可能使集群失去响应。 如果你想要对索引执行 `optimize`，你需要先使用分片分配（查看 [迁移旧索引](https://www.elastic.co/guide/cn/elasticsearch/guide/current/retiring-data.html#migrate-indices)）把索引移到一个安全的节点，再执行。

# 深入搜索

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

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

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

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

## 结构化搜索

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

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

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

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

### 精确值查找

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

**term 查询数字**

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

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

POST /my\_store/products/\_bulk

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

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

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

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

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

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

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

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

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

SELECT document

FROM products

WHERE price = 20

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

{

"term" : {

"price" : 20

}

}

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

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

GET /my\_store/products/\_search

{

"query" : {

"constant\_score" : {

"filter" : {

"term" : {

"price" : 20

}

}

}

}

}

|  |  |
| --- | --- |
|  | 我们用 constant\_score 将 term 查询转化成为过滤器 |
|  | 我们之前看到过的 term 查询 |

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

"hits" : [

{

"\_index" : "my\_store",

"\_type" : "products",

"\_id" : "2",

"\_score" : 1.0,

"\_source" : {

"price" : 20,

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

}

}

]

|  |  |
| --- | --- |
|  | 查询置于 filter 语句内不进行评分或相关度的计算，所以所有的结果都会返回一个默认评分 1 。 |

**term 查询文本**

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

SELECT product

FROM products

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

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

GET /my\_store/products/\_search

{

"query" : {

"constant\_score" : {

"filter" : {

"term" : {

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

}

}

}

}

}

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

GET /my\_store/\_analyze

{

"field": "productID",

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

}

{

"tokens" : [ {

"token" : "xhdk",

"start\_offset" : 0,

"end\_offset" : 4,

"type" : "<ALPHANUM>",

"position" : 1

}, {

"token" : "a",

"start\_offset" : 5,

"end\_offset" : 6,

"type" : "<ALPHANUM>",

"position" : 2

}, {

"token" : "1293",

"start\_offset" : 7,

"end\_offset" : 11,

"type" : "<NUM>",

"position" : 3

}, {

"token" : "fj3",

"start\_offset" : 13,

"end\_offset" : 16,

"type" : "<ALPHANUM>",

"position" : 4

} ]

}

这里有几点需要注意：

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

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

显然这种对 ID 码或其他任何精确值的处理方式并不是我们想要的。

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

DELETE /my\_store

PUT /my\_store

{

"mappings" : {

"products" : {

"properties" : {

"productID" : {

"type" : "string",

"index" : "not\_analyzed"

}

}

}

}

}

|  |  |
| --- | --- |
|  | 删除索引是必须的，因为我们不能更新已存在的映射。 |
|  | 在索引被删除后，我们可以创建新的索引并为其指定自定义映射。 |
|  | 这里我们告诉 Elasticsearch ，我们不想对 productID 做任何分析。 |

现在我们可以为文档重建索引：

POST /my\_store/products/\_bulk

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

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

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

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

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

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

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

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

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

GET /my\_store/products/\_search

{

"query" : {

"constant\_score" : {

"filter" : {

"term" : {

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

}

}

}

}

}

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

**内部过滤器的操作**

在内部，Elasticsearch 会在运行非评分查询的时执行多个操作：

1. 查找匹配文档.

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

1. 创建 bitset.

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

1. 迭代 bitset(s)

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

1. 增量使用计数.

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

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

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

从概念上记住非评分计算是首先执行的，这将有助于写出高效又快速的搜索请求。

### 组合过滤器

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

SELECT product

FROM products

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

AND (price != 30)

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

**布尔过滤器**

一个 bool 过滤器由三部分组成：

{

"bool" : {

"must" : [],

"should" : [],

"must\_not" : [],

}

}

must

所有的语句都 必须（must） 匹配，与 AND 等价。

must\_not

所有的语句都 不能（must not） 匹配，与 NOT 等价。

should

至少有一个语句要匹配，与 OR 等价。

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

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

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

GET /my\_store/products/\_search

{

"query" : {

"filtered" : {

"filter" : {

"bool" : {

"should" : [

{ "term" : {"price" : 20}},

{ "term" : {"productID" : "XHDK-A-1293-#fJ3"}}

],

"must\_not" : {

"term" : {"price" : 30}

}

}

}

}

}

}

|  |  |
| --- | --- |
|  | 注意，我们仍然需要一个 filtered 查询将所有的东西包起来。 |
|  | 在 should 语句块里面的两个 term 过滤器与 bool 过滤器是父子关系，两个 term 条件需要匹配其一。 |
|  | 如果一个产品的价格是 30 ，那么它会自动被排除，因为它处于 must\_not 语句里面。 |

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

"hits" : [

{

"\_id" : "1",

"\_score" : 1.0,

"\_source" : {

"price" : 10,

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

}

},

{

"\_id" : "2",

"\_score" : 1.0,

"\_source" : {

"price" : 20,

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

}

}

]

|  |  |
| --- | --- |
|  | 与 term 过滤器中 productID = "XHDK-A-1293-#fJ3" 条件匹配 |
|  | 与 term 过滤器中 price = 20 条件匹配 |

**嵌套布尔过滤器**

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

对于以下这个 SQL 语句：

SELECT document

FROM products

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

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

AND price = 30 )

我们将其转换成一组嵌套的 bool 过滤器：

GET /my\_store/products/\_search

{

"query" : {

"filtered" : {

"filter" : {

"bool" : {

"should" : [

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

{ "bool" : {

"must" : [

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

{ "term" : {"price" : 30}}

]

}}

]

}

}

}

}

}

|  |  |
| --- | --- |
|  | 因为 term 和 bool 过滤器是兄弟关系，他们都处于外层的布尔逻辑 should 的内部，返回的命中文档至少须匹配其中一个过滤器的条件。 |
|  | 这两个 term 语句作为兄弟关系，同时处于 must 语句之中，所以返回的命中文档要必须都能同时匹配这两个条件。 |

得到的结果有两个文档，它们各匹配 should 语句中的一个条件：

"hits" : [

{

"\_id" : "2",

"\_score" : 1.0,

"\_source" : {

"price" : 20,

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

}

},

{

"\_id" : "3",

"\_score" : 1.0,

"\_source" : {

"price" : 30,

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

}

}

]

|  |  |
| --- | --- |
|  | 这个 productID 与外层的 bool 过滤器 should 里的唯一一个 term 匹配。 |
|  | 这两个字段与嵌套的 bool 过滤器 must 里的两个 term 匹配。 |

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

### 查找多个精确值

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

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

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

{

"terms" : {

"price" : [20, 30]

}

}

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

GET /my\_store/products/\_search

{

"query" : {

"constant\_score" : {

"filter" : {

"terms" : {

"price" : [20, 30]

}

}

}

}

}

|  |  |
| --- | --- |
|  | 这个 terms 查询被置于 constant\_score 查询中 |

运行结果返回第二、第三和第四个文档：

"hits" : [

{

"\_id" : "2",

"\_score" : 1.0,

"\_source" : {

"price" : 20,

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

}

},

{

"\_id" : "3",

"\_score" : 1.0,

"\_source" : {

"price" : 30,

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

}

},

{

"\_id": "4",

"\_score": 1.0,

"\_source": {

"price": 30,

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

}

}

]

**包含，而不是相等**

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

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

{ "tags" : ["search"] }

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

|  |  |
| --- | --- |
|  | 尽管第二个文档包含除 search 以外的其他词，它还是被匹配并作为结果返回。 |

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

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

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

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

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

**精确相等**

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

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

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

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

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

{

"query": {

"constant\_score" : {

"filter" : {

"bool" : {

"must" : [

{ "term" : { "tags" : "search" } },

{ "term" : { "tag\_count" : 1 } }

]

}

}

}

}

}

|  |  |
| --- | --- |
|  | 查找所有包含 term search 的文档。 |
|  | 确保文档只有一个标签。 |

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

### 范围

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

在 SQL 中，范围查询可以表示为：

SELECT document

FROM products

WHERE price BETWEEN 20 AND 40

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

"range" : {

"price" : {

"gte" : 20,

"lte" : 40

}

}

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

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

**下面是一个范围查询的例子：.**

GET /my\_store/products/\_search

{

"query" : {

"constant\_score" : {

"filter" : {

"range" : {

"price" : {

"gte" : 20,

"lt" : 40

}

}

}

}

}

}

如果想要范围无界（比方说 >20 ），只须省略其中一边的限制：

"range" : {

"price" : {

"gt" : 20

}

}

**日期范围**

range 查询同样可以应用在日期字段上：

"range" : {

"timestamp" : {

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

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

}

}

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

"range" : {

"timestamp" : {

"gt" : "now-1h"

}

}

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

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

"range" : {

"timestamp" : {

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

"lt" : "2014-01-01 00:00:00||+1M"

}

}

|  |  |
| --- | --- |
|  | 早于 2014 年 1 月 1 日加 1 月（2014 年 2 月 1 日 零时） |

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

**字符串范围**

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

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

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

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

"range" : {

"title" : {

"gte" : "a",

"lt" : "b"

}

}

**注意基数**

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

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

### 处理 Null 值

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

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

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

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

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

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

**存在查询**

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

POST /my\_index/posts/\_bulk

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

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

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

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

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

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

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

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

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

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

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

以上文档集合中 tags 字段对应的倒排索引如下：

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

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

SELECT tags

FROM posts

WHERE tags IS NOT NULL

在 Elasticsearch 中，使用 exists 查询的方式如下：

GET /my\_index/posts/\_search

{

"query" : {

"constant\_score" : {

"filter" : {

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

}

}

}

}

这个查询返回 3 个文档：

"hits" : [

{

"\_id" : "1",

"\_score" : 1.0,

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

},

{

"\_id" : "5",

"\_score" : 1.0,

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

},

{

"\_id" : "2",

"\_score" : 1.0,

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

}

]

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

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

**缺失查询**

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

SELECT tags

FROM posts

WHERE tags IS NULL

我们将前面例子中 exists 查询换成 missing 查询：

GET /my\_index/posts/\_search

{

"query" : {

"constant\_score" : {

"filter": {

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

}

}

}

}

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

"hits" : [

{

"\_id" : "3",

"\_score" : 1.0,

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

},

{

"\_id" : "4",

"\_score" : 1.0,

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

}

]

**当 null 的意思是 null**

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

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

当选择合适的 null\_value 空值的时候，需要保证以下几点：

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

**对象上的存在与缺失**

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

{

"name" : {

"first" : "John",

"last" : "Smith"

}

}

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

{

"name.first" : "John",

"name.last" : "Smith"

}

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

原因是当我们执行下面这个过滤的时候：

{

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

}

实际执行的是：

{

"bool": {

"should": [

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

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

]

}

}

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

### 关于缓存

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

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

**独立的过滤器缓存**

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

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

让我们看看下面例子中的查询，它查找满足以下任意一个条件的电子邮件：

* 在收件箱中，且没有被读过的
* 不在 收件箱中，但被标注重要的

GET /inbox/emails/\_search

{

"query": {

"constant\_score": {

"filter": {

"bool": {

"should": [

{ "bool": {

"must": [

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

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

]

}},

{ "bool": {

"must\_not": {

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

},

"must": {

"term": { "important": true }

}

}}

]

}

}

}

}

}

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

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

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

**自动缓存行为**

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

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

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

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

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

## 全文搜索

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

全文搜索两个最重要的方面是：

相关性（Relevance）

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

分析（Analysis）

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

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

### 基于词项与基于全文

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

基于词项的查询

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

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

基于全文的查询

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

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

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

我们将会在随后章节中详细讨论这个过程。

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

注意

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

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

GET /\_search

{

"query": {

"constant\_score": {

"filter": {

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

}

}

}

}

### 匹配查询

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

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

**索引一些数据**

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

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

PUT /my\_index

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

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

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

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

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

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

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

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

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

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

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

**单个词查询**

我们用第一个示例来解释使用 match 查询搜索全文字段中的单个词：

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

{

"query": {

"match": {

"title": "QUICK!"

}

}

}

Elasticsearch 执行上面这个 match 查询的步骤是：

1. 检查字段类型 。

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

1. 分析查询字符串 。

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

1. 查找匹配文档 。

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

1. 为每个文档评分 。

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

这个过程给我们以下（经缩减）结果：

"hits": [

{

"\_id": "1",

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

"\_source": {

"title": "The quick brown fox"

}

},

{

"\_id": "3",

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

"\_source": {

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

}

},

{

"\_id": "2",

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

"\_source": {

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

}

}

]

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

### 多词查询

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

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

{

"query": {

"match": {

"title": "BROWN DOG!"

}

}

}

上面这个查询返回所有四个文档：

{

"hits": [

{

"\_id": "4",

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

"\_source": {

"title": "Brown fox brown dog"

}

},

{

"\_id": "2",

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

"\_source": {

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

}

},

{

"\_id": "3",

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

"\_source": {

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

}

},

{

"\_id": "1",

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

"\_source": {

"title": "The quick brown fox"

}

}

]

}

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

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

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

**提高精度**

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

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

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

{

"query": {

"match": {

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

"query": "BROWN DOG!",

"operator": "and"

}

}

}

}

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

这个查询可以把文档 1 排除在外，因为它只包含两个词项中的一个。

**控制精度**

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

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

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

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

{

"query": {

"match": {

"title": {

"query": "quick brown dog",

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

}

}

}

}

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

注意

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

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

### 组合查询

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

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

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

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

{

"query": {

"bool": {

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

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

"should": [

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

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

]

}

}

}

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

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

{

"hits": [

{

"\_id": "3",

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

"\_source": {

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

}

},

{

"\_id": "1",

"\_score": 0.3312608,

"\_source": {

"title": "The quick brown fox"

}

}

]

}

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

**评分计算**

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

must\_not 语句不会影响评分； 它的作用只是将不相关的文档排除。

**控制精度**

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

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

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

{

"query": {

"bool": {

"should": [

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

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

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

],

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

}

}

}

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

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

### 如何使用布尔匹配

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

{

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

}

{

"bool": {

"should": [

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

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

]

}

}

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

{

"match": {

"title": {

"query": "brown fox",

"operator": "and"

}

}

}

{

"bool": {

"must": [

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

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

]

}

}

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

{

"match": {

"title": {

"query": "quick brown fox",

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

}

}

}

{

"bool": {

"should": [

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

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

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

],

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

}

}

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

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

### 查询语句提升权重

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

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

一个简单的 bool *查询* 允许我们写出如下这种非常复杂的逻辑：

GET /\_search

{

"query": {

"bool": {

"must": {

"match": {

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

"query": "full text search",

"operator": "and"

}

}

},

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

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

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

]

}

}

}

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

should 语句匹配得越多表示文档的相关度越高。目前为止还挺好。

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

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

GET /\_search

{

"query": {

"bool": {

"must": {

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

"content": {

"query": "full text search",

"operator": "and"

}

}

},

"should": [

{ "match": {

"content": {

"query": "Elasticsearch",

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

}

}},

{ "match": {

"content": {

"query": "Lucene",

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

}

}}

]

}

}

}

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

注意

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

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

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

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

### 控制分析

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

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

例如为 my\_index 新增一个字段：

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

{

"my\_type": {

"properties": {

"english\_title": {

"type": "string",

"analyzer": "english"

}

}

}

}

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

GET /my\_index/\_analyze

{

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

"text": "Foxes"

}

GET /my\_index/\_analyze

{

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

"text": "Foxes"

}

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

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

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

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

{

"query": {

"bool": {

"should": [

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

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

]

}

}

}

返回语句的 explanation 结果：

(title:foxes english\_title:fox)

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

**默认分析器**

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

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

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

在搜索时，顺序有些许不同：

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

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

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

如果考虑到这些额外参数，一个搜索时的 *完整* 顺序会是下面这样：

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

**分析器配置实践**

就可以配置分析器地方的数量而言是十分惊人的， 但是实际非常简单。

**保持简单**

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

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

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

注意

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

### 被破坏的相关度！

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

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

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

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

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

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

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

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

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

提示

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

## 多字段搜索

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

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

在本章，我们会介绍构造多语句搜索的工具及在特定场景下应该采用的解决方案。

### 多字符串查询

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

GET /\_search

{

"query": {

"bool": {

"should": [

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

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

]

}

}

}

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

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

GET /\_search

{

"query": {

"bool": {

"should": [

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

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

{ "bool": {

"should": [

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

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

]

}}

]

}

}

}

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

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

**语句的优先级**

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

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

GET /\_search

{

"query": {

"bool": {

"should": [

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

"title": {

"query": "War and Peace",

"boost": 2

}}},

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

"author": {

"query": "Leo Tolstoy",

"boost": 2

}}},

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

"should": [

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

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

]

}}

]

}

}

}

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

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

### 单字符串查询

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

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

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

**了解我们的数据**

当用户输入了单个字符串查询的时候，通常会遇到以下三种情形：

最佳字段

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

多数字段

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

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

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

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

混合字段

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

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

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

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

### 最佳字段

假设有个网站允许用户搜索博客的内容， 以下面两篇博客内容文档为例：

PUT /my\_index/my\_type/1

{

"title": "Quick brown rabbits",

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

}

PUT /my\_index/my\_type/2

{

"title": "Keeping pets healthy",

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

}

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

现在运行以下 bool 查询：

{

"query": {

"bool": {

"should": [

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

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

]

}

}

}

但是我们发现查询的结果是文档 1 的评分更高：

{

"hits": [

{

"\_id": "1",

"\_score": 0.14809652,

"\_source": {

"title": "Quick brown rabbits",

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

}

},

{

"\_id": "2",

"\_score": 0.09256032,

"\_source": {

"title": "Keeping pets healthy",

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

}

}

]

}

为了理解导致这样的原因， 需要回想一下 bool 是如何计算评分的：

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

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

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

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

**dis\_max 查询**

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

{

"query": {

"dis\_max": {

"queries": [

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

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

]

}

}

}

得到我们想要的结果为：

{

"hits": [

{

"\_id": "2",

"\_score": 0.21509302,

"\_source": {

"title": "Keeping pets healthy",

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

}

},

{

"\_id": "1",

"\_score": 0.12713557,

"\_source": {

"title": "Quick brown rabbits",

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

}

}

]

}

### 最佳字段查询调优

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

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

{

"query": {

"dis\_max": {

"queries": [

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

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

]

}

}

}

{

"hits": [

{

"\_id": "1",

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

"\_source": {

"title": "Quick brown rabbits",

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

}

},

{

"\_id": "2",

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

"\_source": {

"title": "Keeping pets healthy",

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

}

}

]

}

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

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

**tie\_breaker 参数**

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

{

"query": {

"dis\_max": {

"queries": [

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

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

],

"tie\_breaker": 0.3

}

}

}

结果如下：

{

"hits": [

{

"\_id": "2",

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

"\_source": {

"title": "Keeping pets healthy",

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

}

},

{

"\_id": "1",

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

"\_source": {

"title": "Quick brown rabbits",

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

}

}

]

}

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

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

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

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

注意

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

### multi\_match 查询

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

注意

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

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

{

"dis\_max": {

"queries": [

{

"match": {

"title": {

"query": "Quick brown fox",

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

}

}

},

{

"match": {

"body": {

"query": "Quick brown fox",

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

}

}

},

],

"tie\_breaker": 0.3

}

}

上面这个查询用 multi\_match 重写成更简洁的形式：

{

"multi\_match": {

"query": "Quick brown fox",

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

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

"tie\_breaker": 0.3,

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

}

}

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

**查询字段名称的模糊匹配**

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

{

"multi\_match": {

"query": "Quick brown fox",

"fields": "\*\_title"

}

}

**提升单个字段的权重**

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

{

"multi\_match": {

"query": "Quick brown fox",

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

}

}

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

### 多数字段

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

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

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

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

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

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

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

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

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

**多字段映射**

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

DELETE /my\_index

PUT /my\_index

{

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

"mappings": {

"my\_type": {

"properties": {

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

"type": "string",

"analyzer": "english",

"fields": {

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

"type": "string",

"analyzer": "standard"

}

}

}

}

}

}

}

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

接着索引一些文档：

PUT /my\_index/my\_type/1

{ "title": "My rabbit jumps" }

PUT /my\_index/my\_type/2

{ "title": "Jumping jack rabbits" }

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

GET /my\_index/\_search

{

"query": {

"match": {

"title": "jumping rabbits"

}

}

}

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

{

"hits": [

{

"\_id": "1",

"\_score": 0.42039964,

"\_source": {

"title": "My rabbit jumps"

}

},

{

"\_id": "2",

"\_score": 0.42039964,

"\_source": {

"title": "Jumping jack rabbits"

}

}

]

}

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

GET /my\_index/\_search

{

"query": {

"multi\_match": {

"query": "jumping rabbits",

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

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

}

}

}

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

{

"hits": [

{

"\_id": "2",

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

"\_source": {

"title": "Jumping jack rabbits"

}

},

{

"\_id": "1",

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

"\_source": {

"title": "My rabbit jumps"

}

}

]

}

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

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

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

GET /my\_index/\_search

{

"query": {

"multi\_match": {

"query": "jumping rabbits",

"type": "most\_fields",

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

}

}

}

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

### 跨字段实体搜索

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

{

"firstname": "Peter",

"lastname": "Smith"

}

或地址：

{

"street": "5 Poland Street",

"city": "London",

"country": "United Kingdom",

"postcode": "W1V 3DG"

}

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

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

**简单的方式**

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

{

"query": {

"bool": {

"should": [

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

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

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

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

]

}

}

}

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

{

"query": {

"multi\_match": {

"query": "Poland Street W1V",

"type": "most\_fields",

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

}

}

}

**most\_fields 方式的问题**

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

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

### 字段中心式查询

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

注意

best\_fields 类型也是字段中心式的， 它也存在类似的问题。

首先查看这些问题存在的原因，再想如何解决它们。

**问题 1 ：在多个字段中匹配相同的词**

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

可以通过 validate-query API 查看：

GET /\_validate/query?explain

{

"query": {

"multi\_match": {

"query": "Poland Street W1V",

"type": "most\_fields",

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

}

}

}

生成 explanation 解释：

(street:poland street:street street:w1v)

(city:poland city:street city:w1v)

(country:poland country:street country:w1v)

(postcode:poland postcode:street postcode:w1v)

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

**问题 2 ：剪掉长尾**

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

{

"query": {

"multi\_match": {

"query": "Poland Street W1V",

"type": "most\_fields",

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

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

}

}

}

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

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

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

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

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

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

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

**问题 3 ：词频**

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

词频

一个词在单个文档的某个字段中出现的频率越高，这个文档的相关度就越高。

逆向文档频率

一个词在所有文档某个字段索引中出现的频率越高，这个词的相关度就越低。

当搜索多个字段时，TF/IDF 会带来某些令人意外的结果。

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

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

{

"query": {

"multi\_match": {

"query": "Peter Smith",

"type": "most\_fields",

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

}

}

}

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

**解决方案**

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

{

"first\_name": "Peter",

"last\_name": "Smith",

"full\_name": "Peter Smith"

}

当查询 full\_name 字段时：

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

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

### 自定义 \_all 字段

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

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

PUT /my\_index

{

"mappings": {

"person": {

"properties": {

"first\_name": {

"type": "string",

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

},

"last\_name": {

"type": "string",

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

},

"full\_name": {

"type": "string"

}

}

}

}

}

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

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

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

警告

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

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

只要对“主”字段 copy\_to 就能轻而易举的达到相同的效果：

PUT /my\_index

{

"mappings": {

"person": {

"properties": {

"first\_name": {

"type": "string",

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

"fields": {

"raw": {

"type": "string",

"index": "not\_analyzed"

}

}

},

"full\_name": {

"type": "string"

}

}

}

}

}

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

### cross-fields 跨字段查询

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

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

GET /\_validate/query?explain

{

"query": {

"multi\_match": {

"query": "peter smith",

"type": "most\_fields",

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

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

}

}

}

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

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

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

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

*词中心式* 会使用以下逻辑：

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

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

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

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

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

GET /\_validate/query?explain

{

"query": {

"multi\_match": {

"query": "peter smith",

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

"operator": "and",

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

}

}

}

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

它通过 *混合* 不同字段逆向索引文档频率的方式解决了词频的问题：

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

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

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

注意

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

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

(+title:peter +title:smith)

(

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

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

)

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

**按字段提高权重**

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

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

GET /books/\_search

{

"query": {

"multi\_match": {

"query": "peter smith",

"type": "cross\_fields",

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

}

}

}

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

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

### Exact-Value 精确值字段

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

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

GET /\_validate/query?explain

{

"query": {

"multi\_match": {

"query": "peter smith",

"type": "cross\_fields",

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

}

}

}

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

title:peter smith

(

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

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

)

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

## 近似匹配

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

思考下面这几个句子的不同：

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

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

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

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

这就是短语匹配或者近似匹配的所属领域。

提示

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

### 短语匹配

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

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

{

"query": {

"match\_phrase": {

"title": "quick brown fox"

}

}

}

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

提示

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

"match": {

"title": {

"query": "quick brown fox",

"type": "phrase"

}

}

**词项的位置**

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

GET /\_analyze?analyzer=standard

Quick brown fox

返回信息如下：

{

"tokens": [

{

"token": "quick",

"start\_offset": 0,

"end\_offset": 5,

"type": "<ALPHANUM>",

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

},

{

"token": "brown",

"start\_offset": 6,

"end\_offset": 11,

"type": "<ALPHANUM>",

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

},

{

"token": "fox",

"start\_offset": 12,

"end\_offset": 15,

"type": "<ALPHANUM>",

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

}

]

}

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

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

**什么是短语**

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

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

如果以上任何一个选项不成立，则该文档不能认定为匹配。

提示

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

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

### 混合起来

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

我们能够通过使用 slop 参数将灵活度引入短语匹配中：

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

{

"query": {

"match\_phrase": {

"title": {

"query": "quick fox",

"slop": 1

}

}

}

}

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

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

Pos 1 Pos 2 Pos 3

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

Doc: quick brown fox

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

Query: quick fox

Slop 1: quick ↳ fox

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

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

Pos 1 Pos 2 Pos 3

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

Doc: quick brown fox

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

Query: fox quick

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

Slop 2: quick ↳ fox

Slop 3: quick ↳ fox

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

### 多值字段

对多值字段使用短语匹配时会发生奇怪的事。 想象一下你索引这个文档:

PUT /my\_index/groups/1

{

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

}

然后运行一个对 Abraham Lincoln 的短语查询:

GET /my\_index/groups/\_search

{

"query": {

"match\_phrase": {

"names": "Abraham Lincoln"

}

}

}

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

在分析 John Abraham 的时候， 产生了如下信息：

* Position 1: john
* Position 2: abraham

然后在分析 Lincoln Smith 的时候， 产生了：

* Position 3: lincoln
* Position 4: smith

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

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

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

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

{

"properties": {

"names": {

"type": "string",

"position\_increment\_gap": 100

}

}

}

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

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

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

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

### 越近越好

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

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

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

{

"query": {

"match\_phrase": {

"title": {

"query": "quick dog",

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

}

}

}

}

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

{

"hits": [

{

"\_id": "3",

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

"\_source": {

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

}

},

{

"\_id": "2",

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

"\_source": {

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

}

}

]

}

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

### 使用邻近度提高相关度

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

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

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

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

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

{

"query": {

"bool": {

"must": {

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

"title": {

"query": "quick brown fox",

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

}

}

},

"should": {

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

"title": {

"query": "quick brown fox",

"slop": 50

}

}

}

}

}

}

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

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

### 性能优化

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

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

提示

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

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

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

**结果集重新评分**

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

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

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

该请求如下所示：

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

{

"query": {

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

"title": {

"query": "quick brown fox",

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

}

}

},

"rescore": {

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

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

"rescore\_query": {

"match\_phrase": {

"title": {

"query": "quick brown fox",

"slop": 50

}

}

}

}

}

}

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

### 寻找相关词

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

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

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

如果索引单词对而不是索引独立的单词，就能对这些单词的上下文尽可能多的保留。

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

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

也要将每个单词 *以及它的邻近词* 作为单个词项索引：

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

这些单词对（或者 *bigrams* ）被称为 *shingles* 。

提示

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

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

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

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

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

**生成 Shingles**

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

首先，我们需要在创建分析器时使用 shingle 语汇单元过滤器：

DELETE /my\_index

PUT /my\_index

{

"settings": {

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

"analysis": {

"filter": {

"my\_shingle\_filter": {

"type": "shingle",

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

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

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

}

},

"analyzer": {

"my\_shingle\_analyzer": {

"type": "custom",

"tokenizer": "standard",

"filter": [

"lowercase",

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

]

}

}

}

}

}

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

首先，用 analyze API 测试下分析器：

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

Sue ate the alligator

果然， 我们得到了 3 个词项：

* sue ate
* ate the
* the alligator

现在我们可以继续创建一个使用新的分析器的字段。

**多字段**

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

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

{

"my\_type": {

"properties": {

"title": {

"type": "string",

"fields": {

"shingles": {

"type": "string",

"analyzer": "my\_shingle\_analyzer"

}

}

}

}

}

}

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

最后，我们可以索引以下示例文档:

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

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

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

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

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

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

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

**搜索 Shingles**

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

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

{

"query": {

"match": {

"title": "the hungry alligator ate sue"

}

}

}

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

{

"hits": [

{

"\_id": "1",

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

"\_source": {

"title": "Sue ate the alligator"

}

},

{

"\_id": "2",

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

"\_source": {

"title": "The alligator ate Sue"

}

},

{

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

"\_score": 0.046571054,

"\_source": {

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

}

}

]

}

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

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

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

{

"query": {

"bool": {

"must": {

"match": {

"title": "the hungry alligator ate sue"

}

},

"should": {

"match": {

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

}

}

}

}

}

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

{

"hits": [

{

"\_id": "2",

"\_score": 0.4883322,

"\_source": {

"title": "The alligator ate Sue"

}

},

{

"\_id": "1",

"\_score": 0.13422975,

"\_source": {

"title": "Sue ate the alligator"

}

},

{

"\_id": "3",

"\_score": 0.014119488,

"\_source": {

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

}

}

]

}

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

**Performance性能**

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

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

## 部分匹配

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

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

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

WHERE text LIKE "%quick%"

AND text LIKE "%brown%"

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

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

当然， Elasticsearch 提供分析过程，倒排索引让我们不需要使用这种粗笨的技术。为了能应对同时匹配 “fox” 和 “foxes” 的情况，只需简单的将它们的词干作为索引形式，没有必要做部分匹配。

也就是说，在某些情况下部分匹配会比较有用， 常见的应用如下：

* 匹配邮编、产品序列号或其他 not\_analyzed 未分析值，这些值可以是以某个特定前缀开始，也可以是与某种模式匹配的，甚至可以是与某个正则式相匹配的。
* *输入即搜索（search-as-you-type）* ——在用户键入搜索词过程的同时就呈现最可能的结果。
* 匹配如德语或荷兰语这样有长组合词的语言，如： *Weltgesundheitsorganisation* （世界卫生组织，英文 World Health Organization）。

本章始于检验 not\_analyzed 精确值字段的前缀匹配。

### 邮编与结构化数据

我们会使用美国目前使用的邮编形式（United Kingdom postcodes 标准）来说明如何用部分匹配查询结构化数据。 这种邮编形式有很好的结构定义。例如，邮编 W1V 3DG 可以分解成如下形式：

* W1V ：这是邮编的外部，它定义了邮件的区域和行政区：
  + W 代表区域（ 1 或 2 个字母）
  + 1V 代表行政区（ 1 或 2 个数字，可能跟着一个字符）
* 3DG ：内部定义了街道或建筑：
  + 3 代表街区区块（ 1 个数字）
  + DG 代表单元（ 2 个字母）

假设将邮编作为 not\_analyzed 的精确值字段索引，所以可以为其创建索引，如下：

PUT /my\_index

{

"mappings": {

"address": {

"properties": {

"postcode": {

"type": "string",

"index": "not\_analyzed"

}

}

}

}

}

然后索引一些邮编：

PUT /my\_index/address/1

{ "postcode": "W1V 3DG" }

PUT /my\_index/address/2

{ "postcode": "W2F 8HW" }

PUT /my\_index/address/3

{ "postcode": "W1F 7HW" }

PUT /my\_index/address/4

{ "postcode": "WC1N 1LZ" }

PUT /my\_index/address/5

{ "postcode": "SW5 0BE" }

现在这些数据已可查询。

### prefix 前缀查询

为了找到所有以 W1 开始的邮编，可以使用简单的 prefix 查询：

GET /my\_index/address/\_search

{

"query": {

"prefix": {

"postcode": "W1"

}

}

}

prefix 查询是一个词级别的底层的查询，它不会在搜索之前分析查询字符串，它假定传入前缀就正是要查找的前缀。

提示

默认状态下， prefix 查询不做相关度评分计算，它只是将所有匹配的文档返回，并为每条结果赋予评分值 1 。它的行为更像是过滤器而不是查询。 prefix 查询和 prefix 过滤器这两者实际的区别就是过滤器是可以被缓存的，而查询不行。

之前已经提过：“只能在倒排索引中找到存在的词”，但我们并没有对这些邮编的索引进行特殊处理，每个邮编还是以它们精确值的方式存在于每个文档的索引中，那么 prefix 查询是如何工作的呢？

回想倒排索引包含了一个有序的唯一词列表（本例是邮编）。 对于每个词，倒排索引都会将包含词的文档 ID 列入 *倒排表（postings list）* 。与示例对应的倒排索引是：

Term: Doc IDs:

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

"SW5 0BE" | 5

"W1F 7HW" | 3

"W1V 3DG" | 1

"W2F 8HW" | 2

"WC1N 1LZ" | 4

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

为了支持前缀匹配，查询会做以下事情：

1. 扫描词列表并查找到第一个以 W1 开始的词。
2. 搜集关联的文档 ID 。
3. 移动到下一个词。
4. 如果这个词也是以 W1 开头，查询跳回到第二步再重复执行，直到下一个词不以 W1 为止。

这对于小的例子当然可以正常工作，但是如果倒排索引中有数以百万的邮编都是以 W1 开头时，前缀查询则需要访问每个词然后计算结果！

前缀越短所需访问的词越多。如果我们要以 W 作为前缀而不是 W1 ，那么就可能需要做千万次的匹配。

小心

prefix 查询或过滤对于一些特定的匹配是有效的，但使用方式还是应当注意。 当字段中词的集合很小时，可以放心使用，但是它的伸缩性并不好，会对我们的集群带来很多压力。可以使用较长的前缀来限制这种影响，减少需要访问的量。

本章后面会介绍另一个索引时的解决方案，这个方案能使前缀匹配更高效，不过在此之前，需要先看看两个相关的查询： wildcard 和 regexp （模糊和正则）。

### 通配符与正则表达式查询

与 prefix 前缀查询的特性类似， wildcard 通配符查询也是一种底层基于词的查询， 与前缀查询不同的是它允许指定匹配的正则式。它使用标准的 shell 通配符查询： ? 匹配任意字符， \* 匹配 0 或多个字符。

这个查询会匹配包含 W1F 7HW 和 W2F 8HW 的文档：

GET /my\_index/address/\_search

{

"query": {

"wildcard": {

"postcode": "W?F\*HW" https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_wildcard_and_regexp_queries.html#CO91-1) | ? 匹配 1 和 2 ， \* 与空格及 7 和 8 匹配。 |

设想如果现在只想匹配 W 区域的所有邮编，前缀匹配也会包括以 WC 开头的所有邮编，与通配符匹配碰到的问题类似，如果想匹配只以 W 开始并跟随一个数字的所有邮编， regexp 正则式查询允许写出这样更复杂的模式：

GET /my\_index/address/\_search

{

"query": {

"regexp": {

"postcode": "W[0-9].+" https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_wildcard_and_regexp_queries.html#CO92-1) | 这个正则表达式要求词必须以 W 开头，紧跟 0 至 9 之间的任何一个数字，然后接一或多个其他字符。 |

wildcard 和 regexp 查询的工作方式与 prefix 查询完全一样，它们也需要扫描倒排索引中的词列表才能找到所有匹配的词，然后依次获取每个词相关的文档 ID ，与 prefix 查询的唯一不同是：它们能支持更为复杂的匹配模式。

这也意味着需要同样注意前缀查询存在性能问题，对有很多唯一词的字段执行这些查询可能会消耗非常多的资源，所以要避免使用左通配这样的模式匹配（如： \*foo 或 .\*foo 这样的正则式）。

数据在索引时的预处理有助于提高前缀匹配的效率，而通配符和正则表达式查询只能在查询时完成，尽管这些查询有其应用场景，但使用仍需谨慎。

小心

prefix 、 wildcard 和 regexp 查询是基于词操作的，如果用它们来查询 analyzed 字段，它们会检查字段里面的每个词，而不是将字段作为整体来处理。

比方说包含 “Quick brown fox” （快速的棕色狐狸）的 title 字段会生成词： quick 、 brown 和 fox 。

会匹配以下这个查询：

{ "regexp": { "title": "br.\*" }}

但是不会匹配以下两个查询：

{ "regexp": { "title": "Qu.\*" }} https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

{ "regexp": { "title": "quick br\*" }} https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_wildcard_and_regexp_queries.html#CO93-1) | 在索引里的词是 quick 而不是 Quick 。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_wildcard_and_regexp_queries.html#CO93-2) | quick 和 brown 在词表中是分开的。 |

### 查询时输入即搜索

把邮编的事情先放一边，让我们先看看前缀查询是如何在全文查询中起作用的。 用户已经渐渐习惯在输完查询内容之前，就能为他们展现搜索结果，这就是所谓的 *即时搜索（instant search）* 或 *输入即搜索（search-as-you-type）* 。不仅用户能在更短的时间内得到搜索结果，我们也能引导用户搜索索引中真实存在的结果。

例如，如果用户输入 johnnie walker bl ，我们希望在它们完成输入搜索条件前就能得到：Johnnie Walker Black Label 和 Johnnie Walker Blue Label 。

生活总是这样，就像猫的花色远不只一种！我们希望能找到一种最简单的实现方式。并不需要对数据做任何准备，在查询时就能对任意的全文字段实现 *输入即搜索（search-as-you-type）* 的查询。

在 [短语匹配](https://www.elastic.co/guide/cn/elasticsearch/guide/current/phrase-matching.html) 中，我们引入了 match\_phrase 短语匹配查询，它匹配相对顺序一致的所有指定词语，对于查询时的输入即搜索，可以使用 match\_phrase 的一种特殊形式， match\_phrase\_prefix 查询：

{

"match\_phrase\_prefix" : {

"brand" : "johnnie walker bl"

}

}

这种查询的行为与 match\_phrase 查询一致，不同的是它将查询字符串的最后一个词作为前缀使用，换句话说，可以将之前的例子看成如下这样：

* johnnie
* 跟着 walker
* 跟着以 bl 开始的词

如果通过 validate-query API 运行这个查询查询，explanation 的解释结果为：

"johnnie walker bl\*"

与 match\_phrase 一样，它也可以接受 slop 参数（参照 [slop](https://www.elastic.co/guide/cn/elasticsearch/guide/current/slop.html) ）让相对词序位置不那么严格：

{

"match\_phrase\_prefix" : {

"brand" : {

"query": "walker johnnie bl", https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

"slop": 10

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_query_time_search_as_you_type.html#CO94-1) | 尽管词语的顺序不正确，查询仍然能匹配，因为我们为它设置了足够高的 slop 值使匹配时的词序有更大的灵活性。 |

但是只有查询字符串的最后一个词才能当作前缀使用。

在之前的 [前缀查询](https://www.elastic.co/guide/cn/elasticsearch/guide/current/prefix-query.html) 中，我们警告过使用前缀的风险，即 prefix 查询存在严重的资源消耗问题，短语查询的这种方式也同样如此。 前缀 a 可能会匹配成千上万的词，这不仅会消耗很多系统资源，而且结果的用处也不大。

可以通过设置 max\_expansions 参数来限制前缀扩展的影响， 一个合理的值是可能是 50 ：

{

"match\_phrase\_prefix" : {

"brand" : {

"query": "johnnie walker bl",

"max\_expansions": 50

}

}

}

参数 max\_expansions 控制着可以与前缀匹配的词的数量，它会先查找第一个与前缀 bl 匹配的词，然后依次查找搜集与之匹配的词（按字母顺序），直到没有更多可匹配的词或当数量超过 max\_expansions 时结束。

不要忘记，当用户每多输入一个字符时，这个查询又会执行一遍，所以查询需要快，如果第一个结果集不是用户想要的，他们会继续输入直到能搜出满意的结果为止。

### 索引时优化

到目前为止，所有谈论过的解决方案都是在 *查询时（query time）* 实现的。 这样做并不需要特殊的映射或特殊的索引模式，只是简单使用已经索引的数据。

查询时的灵活性通常会以牺牲搜索性能为代价，有时候将这些消耗从查询过程中转移到别的地方是有意义的。在实时 web 应用中， 100 毫秒可能是一个难以忍受的巨大延迟。

可以通过在索引时处理数据提高搜索的灵活性以及提升系统性能。为此仍然需要付出应有的代价：增加的索引空间与变慢的索引能力，但这与每次查询都需要付出代价不同，索引时的代价只用付出一次。

用户会感谢我们。

### Ngrams 在部分匹配的应用

之前提到：“只能在倒排索引中找到存在的词。” 尽管 prefix 、 wildcard 、 regexp 查询告诉我们这种说法并不完全正确，但单个词的查找 *确实* 要比在词列表中盲目挨个查找的效率要高得多。 在搜索之前准备好供部分匹配的数据可以提高搜索的性能。

在索引时准备数据意味着要选择合适的分析链，这里部分匹配使用的工具是 *n-gram* 。可以将 *n-gram* 看成一个在词语上 *滑动窗口* ， *n* 代表这个 “窗口” 的长度。如果我们要 n-gram quick 这个词 —— 它的结果取决于 *n* 的选择长度：

* 长度 1（unigram）： [ q, u, i, c, k ]
* 长度 2（bigram）： [ qu, ui, ic, ck ]
* 长度 3（trigram）： [ qui, uic, ick ]
* 长度 4（four-gram）： [ quic, uick ]
* 长度 5（five-gram）： [ quick ]

朴素的 n-gram 对 *词语内部的匹配* 非常有用，即在 [Ngram 匹配复合词](https://www.elastic.co/guide/cn/elasticsearch/guide/current/ngrams-compound-words.html) 介绍的那样。但对于输入即搜索（search-as-you-type）这种应用场景，我们会使用一种特殊的 n-gram 称为 *边界 n-grams* （edge n-grams）。所谓的边界 n-gram 是说它会固定词语开始的一边，以单词 quick 为例，它的边界 n-gram 的结果为：

* q
* qu
* qui
* quic
* quick

可能会注意到这与用户在搜索时输入 “quick” 的字母次序是一致的，换句话说，这种方式正好满足即时搜索（instant search）！

### 索引时输入即搜索

设置索引时输入即搜索的第一步是需要定义好分析链， 我们已在 [配置分析器](https://www.elastic.co/guide/cn/elasticsearch/guide/current/configuring-analyzers.html) 中讨论过，这里会对这些步骤再次说明。

**准备索引**

第一步需要配置一个自定义的 edge\_ngram token 过滤器，称为 autocomplete\_filter ：

{

"filter": {

"autocomplete\_filter": {

"type": "edge\_ngram",

"min\_gram": 1,

"max\_gram": 20

}

}

}

这个配置的意思是：对于这个 token 过滤器接收的任意词项，过滤器会为之生成一个最小固定值为 1 ，最大为 20 的 n-gram 。

然后会在一个自定义分析器 autocomplete 中使用上面这个 token 过滤器：

{

"analyzer": {

"autocomplete": {

"type": "custom",

"tokenizer": "standard",

"filter": [

"lowercase",

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

]

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_index_time_search_as_you_type.html#CO95-1) | 自定义的 edge-ngram token 过滤器。 |

这个分析器使用 standard 分词器将字符串拆分为独立的词，并且将它们都变成小写形式，然后为每个词生成一个边界 n-gram，这要感谢 autocomplete\_filter 起的作用。

创建索引、实例化 token 过滤器和分析器的完整示例如下：

PUT /my\_index

{

"settings": {

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

"analysis": {

"filter": {

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

"type": "edge\_ngram",

"min\_gram": 1,

"max\_gram": 20

}

},

"analyzer": {

"autocomplete": {

"type": "custom",

"tokenizer": "standard",

"filter": [

"lowercase",

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

]

}

}

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_index_time_search_as_you_type.html#CO96-1) | 参考 [被破坏的相关度](https://www.elastic.co/guide/cn/elasticsearch/guide/current/relevance-is-broken.html) 。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_index_time_search_as_you_type.html#CO96-2) | 首先自定义 token 过滤器。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_index_time_search_as_you_type.html#CO96-3) | 然后在分析器中使用它。 |

可以拿 analyze API 测试这个新的分析器确保它行为正确：

GET /my\_index/\_analyze?analyzer=autocomplete

quick brown

结果表明分析器能正确工作，并返回以下词：

* q
* qu
* qui
* quic
* quick
* b
* br
* bro
* brow
* brown

可以用 update-mapping API 将这个分析器应用到具体字段：

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

{

"my\_type": {

"properties": {

"name": {

"type": "string",

"analyzer": "autocomplete"

}

}

}

}

现在创建一些测试文档：

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

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

{ "name": "Brown foxes" }

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

{ "name": "Yellow furballs" }

**查询字段**

如果使用简单 match 查询测试查询 “brown fo” ：

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

{

"query": {

"match": {

"name": "brown fo"

}

}

}

可以看到两个文档同时 *都能* 匹配，尽管 Yellow furballs 这个文档并不包含 brown 和 fo ：

{

"hits": [

{

"\_id": "1",

"\_score": 1.5753809,

"\_source": {

"name": "Brown foxes"

}

},

{

"\_id": "2",

"\_score": 0.012520773,

"\_source": {

"name": "Yellow furballs"

}

}

]

}

如往常一样， validate-query API 总能提供一些线索：

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

{

"query": {

"match": {

"name": "brown fo"

}

}

}

explanation 表明查询会查找边界 n-grams 里的每个词：

name:b name:br name:bro name:brow name:brown name:f name:fo

name:f 条件可以满足第二个文档，因为 furballs 是以 f 、 fu 、 fur 形式索引的。回过头看这并不令人惊讶，相同的 autocomplete 分析器同时被应用于索引时和搜索时，这在大多数情况下是正确的，只有在少数场景下才需要改变这种行为。

我们需要保证倒排索引表中包含边界 n-grams 的每个词，但是我们只想匹配用户输入的完整词组（ brown和 fo ）， 可以通过在索引时使用 autocomplete 分析器，并在搜索时使用 standard 标准分析器来实现这种想法，只要改变查询使用的搜索分析器 analyzer 参数即可：

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

{

"query": {

"match": {

"name": {

"query": "brown fo",

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

}

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_index_time_search_as_you_type.html#CO97-1) | 覆盖了 name 字段 analyzer 的设置。 |

换种方式，我们可以在映射中，为 name 字段分别指定 index\_analyzer 和 search\_analyzer 。因为我们只想改变 search\_analyzer ，这里只要更新现有的映射而不用对数据重新创建索引：

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

{

"my\_type": {

"properties": {

"name": {

"type": "string",

"index\_analyzer": "autocomplete", https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

"search\_analyzer": "standard" https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

}

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_index_time_search_as_you_type.html#CO98-1) | 在索引时，使用 autocomplete 分析器生成边界 n-grams 的每个词。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_index_time_search_as_you_type.html#CO98-2) | 在搜索时，使用 standard 分析器只搜索用户输入的词。 |

如果再次请求 validate-query API ，当前的解释为：

name:brown name:fo

再次执行查询就能正确返回 Brown foxes 这个文档。

因为大多数工作是在索引时完成的，所有的查询只要查找 brown 和 fo 这两个词，这比使用 match\_phrase\_prefix 查找所有以 fo 开始的词的方式要高效许多。

**补全提示（Completion Suggester）**

使用边界 n-grams 进行输入即搜索（search-as-you-type）的查询设置简单、灵活且快速，但有时候它并不够快，特别是当试图立刻获得反馈时，延迟的问题就会凸显，很多时候不搜索才是最快的搜索方式。

Elasticsearch 里的 [completion suggester](https://www.elastic.co/guide/en/elasticsearch/reference/master/search-suggesters-completion.html) 采用与上面完全不同的方式，需要为搜索条件生成一个所有可能完成的词列表，然后将它们置入一个 *有限状态机（finite state transducer）* 内，这是个经优化的图结构。为了搜索建议提示，Elasticsearch 从图的开始处顺着匹配路径一个字符一个字符地进行匹配，一旦它处于用户输入的末尾，Elasticsearch 就会查找所有可能结束的当前路径，然后生成一个建议列表。

本数据结构存于内存中，能使前缀查找非常快，比任何一种基于词的查询都要快很多，这对名字或品牌的自动补全非常适用，因为这些词通常是以普通顺序组织的：用 “Johnny Rotten” 而不是 “Rotten Johnny” 。

当词序不是那么容易被预见时，边界 n-grams 比完成建议者（Completion Suggester）更合适。即使说不是所有猫都是一个花色，那这只猫的花色也是相当特殊的。

**边界 n-grams 与邮编**

边界 n-gram 的方式可以用来查询结构化的数据， 比如 [本章之前示例](https://www.elastic.co/guide/cn/elasticsearch/guide/current/prefix-query.html) 中的邮编（postcode）。当然 postcode 字段需要 analyzed 而不是 not\_analyzed ，不过可以用 keyword 分词器来处理它，就好像他们是 not\_analyzed 的一样。

提示

keyword 分词器是一个非操作型分词器，这个分词器不做任何事情，它接收的任何字符串都会被原样发出，因此它可以用来处理 not\_analyzed 的字段值，但这也需要其他的一些分析转换，如将字母转换成小写。

下面示例使用 keyword 分词器将邮编转换成 token 流，这样就能使用边界 n-gram token 过滤器：

{

"analysis": {

"filter": {

"postcode\_filter": {

"type": "edge\_ngram",

"min\_gram": 1,

"max\_gram": 8

}

},

"analyzer": {

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

"tokenizer": "keyword",

"filter": [ "postcode\_filter" ]

},

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

"tokenizer": "keyword"

}

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_index_time_search_as_you_type.html#CO99-1) | postcode\_index 分析器使用 postcode\_filter 将邮编转换成边界 n-gram 形式。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_index_time_search_as_you_type.html#CO99-2) | postcode\_search 分析器可以将搜索词看成 not\_analyzed 未分析的。 |

### Ngrams 在复合词的应用

最后，来看看 n-gram 是如何应用于搜索复合词的语言中的。 德语的特点是它可以将许多小词组合成一个庞大的复合词以表达它准确或复杂的意义。例如：

*Aussprachewörterbuch*

发音字典（Pronunciation dictionary）

*Militärgeschichte*

战争史（Military history）

*Weißkopfseeadler*

秃鹰（White-headed sea eagle, or bald eagle）

*Weltgesundheitsorganisation*

世界卫生组织（World Health Organization）

*Rindfleischetikettierungsüberwachungsaufgabenübertragungsgesetz*

法案考虑代理监管牛和牛肉的标记的职责（The law concerning the delegation of duties for the supervision of cattle marking and the labeling of beef）

有些人希望在搜索 “Wörterbuch”（字典）的时候，能在结果中看到 “Aussprachewörtebuch”（发音字典）。同样，搜索 “Adler”（鹰）的时候，能将 “Weißkopfseeadler”（秃鹰）包括在结果中。

处理这种语言的一种方式可以用 [组合词 token 过滤器（compound word token filter）](https://www.elastic.co/guide/en/elasticsearch/reference/master/analysis-compound-word-tokenfilter.html) 将复合词拆分成各自部分，但这种方式的结果质量依赖于组合词字典的质量。

另一种方式就是将所有的词用 n-gram 进行处理，然后搜索任何匹配的片段——能匹配的片段越多，文档的相关度越大。

假设某个 n-gram 是一个词上的滑动窗口，那么任何长度的 n-gram 都可以遍历这个词。我们既希望选择足够长的值让拆分的词项具有意义，又不至于因为太长而生成过多的唯一词。一个长度为 3 的 *trigram* 可能是一个不错的开始：

PUT /my\_index

{

"settings": {

"analysis": {

"filter": {

"trigrams\_filter": {

"type": "ngram",

"min\_gram": 3,

"max\_gram": 3

}

},

"analyzer": {

"trigrams": {

"type": "custom",

"tokenizer": "standard",

"filter": [

"lowercase",

"trigrams\_filter"

]

}

}

}

},

"mappings": {

"my\_type": {

"properties": {

"text": {

"type": "string",

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

}

}

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/ngrams-compound-words.html#CO100-1) | text 字段用 trigrams 分析器索引它的内容，这里 n-gram 的长度是 3 。 |

使用 analyze API 测试 trigram 分析器：

GET /my\_index/\_analyze?analyzer=trigrams

Weißkopfseeadler

返回以下词项：

wei, eiß, ißk, ßko, kop, opf, pfs, fse, see, eea,ead, adl, dle, ler

索引前述示例中的复合词来测试：

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

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

{ "text": "Aussprachewörterbuch" }

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

{ "text": "Militärgeschichte" }

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

{ "text": "Weißkopfseeadler" }

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

{ "text": "Weltgesundheitsorganisation" }

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

{ "text": "Rindfleischetikettierungsüberwachungsaufgabenübertragungsgesetz" }

“Adler”（鹰）的搜索转化为查询三个词 adl 、 dle 和 ler ：

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

{

"query": {

"match": {

"text": "Adler"

}

}

}

正好与 “Weißkopfsee-*adler*” 相匹配：

{

"hits": [

{

"\_id": "3",

"\_score": 3.3191128,

"\_source": {

"text": "Weißkopfseeadler"

}

}

]

}

类似查询 “Gesundheit”（健康）可以与 “Welt-gesundheit-sorganisation” 匹配，同时也能与 “Militär-*ges*-chichte” 和 “Rindfleischetikettierungsüberwachungsaufgabenübertragungs-*ges*-etz” 匹配，因为它们同时都有 trigram 生成的 ges ：

使用合适的 minimum\_should\_match 可以将这些奇怪的结果排除，只有当 trigram 最少匹配数满足要求时，文档才能被认为是匹配的：

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

{

"query": {

"match": {

"text": {

"query": "Gesundheit",

"minimum\_should\_match": "80%"

}

}

}

}

这有点像全文搜索中霰弹枪式的策略，可能会导致倒排索引内容变多，尽管如此，在索引具有很多复合词的语言，或词之间没有空格的语言（如：泰语）时，它仍不失为一种通用且有效的方法。

这种技术可以用来提升 *召回率* ——搜索结果中相关的文档数。它通常会与其他技术一起使用，例如 shingles（参见 [shingles 瓦片词](https://www.elastic.co/guide/cn/elasticsearch/guide/current/shingles.html) ），以提高精度和每个文档的相关度评分。

## 控制相关度

处理结构化数据（比如：时间、数字、字符串、枚举）的数据库， 只需检查文档（或关系数据库里的行）是否与查询匹配。

布尔的是/非匹配是全文搜索的基础，但不止如此，我们还要知道每个文档与查询的相关度，在全文搜索引擎中不仅需要找到匹配的文档，还需根据它们相关度的高低进行排序。

全文相关的公式或 *相似算法（similarity algorithms）* 会将多个因素合并起来，为每个文档生成一个相关度评分 \_score 。本章中，我们会验证各种可变部分，然后讨论如何来控制它们。

当然，相关度不只与全文查询有关，也需要将结构化的数据考虑其中。可能我们正在找一个度假屋，需要一些的详细特征（空调、海景、免费 WiFi ），匹配的特征越多相关度越高。可能我们还希望有一些其他的考虑因素，如回头率、价格、受欢迎度或距离，当然也同时考虑全文查询的相关度。

所有的这些都可以通过 Elasticsearch 强大的评分基础来实现。

本章会先从理论上介绍 Lucene 是如何计算相关度的，然后通过实际例子说明如何控制相关度的计算过程。

### 相关度评分背后的理论

Lucene（或 Elasticsearch）使用 [*布尔模型（Boolean model）*](http://en.wikipedia.org/wiki/Standard_Boolean_model) 查找匹配文档， 并用一个名为 [*实用评分函数（practical scoring function）*](https://www.elastic.co/guide/cn/elasticsearch/guide/current/practical-scoring-function.html) 的公式来计算相关度。这个公式借鉴了 [*词频/逆向文档频率（term frequency/inverse document frequency）*](http://en.wikipedia.org/wiki/Tfidf) 和 [*向量空间模型（vector space model）*](http://en.wikipedia.org/wiki/Vector_space_model)，同时也加入了一些现代的新特性，如协调因子（coordination factor），字段长度归一化（field length normalization），以及词或查询语句权重提升。

注意

不要紧张！这些概念并没有像它们字面看起来那么复杂，尽管本小节提到了算法、公式和数学模型，但内容还是让人容易理解的，与理解算法本身相比，了解这些因素如何影响结果更为重要。

**布尔模型**

*布尔模型（Boolean Model）* 只是在查询中使用 AND 、 OR 和 NOT （与、或和非）这样的条件来查找匹配的文档，以下查询：

full AND text AND search AND (elasticsearch OR lucene)

会将所有包括词 full 、 text 和 search ，以及 elasticsearch 或 lucene 的文档作为结果集。

这个过程简单且快速，它将所有可能不匹配的文档排除在外。

**词频/逆向文档频率（TF/IDF）**

当匹配到一组文档后，需要根据相关度排序这些文档，不是所有的文档都包含所有词，有些词比其他的词更重要。一个文档的相关度评分部分取决于每个查询词在文档中的 *权重* 。

词的权重由三个因素决定，在 [什么是相关](https://www.elastic.co/guide/cn/elasticsearch/guide/current/relevance-intro.html) 中已经有所介绍，有兴趣可以了解下面的公式，但并不要求记住。

**词频**

词在文档中出现的频度是多少？ 频度越高，权重 *越高* 。 5 次提到同一词的字段比只提到 1 次的更相关。词频的计算方式如下：

tf(t in d) = √frequency https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/scoring-theory.html#CO101-1) | 词 t 在文档 d 的词频（ tf ）是该词在文档中出现次数的平方根。 |

如果不在意词在某个字段中出现的频次，而只在意是否出现过，则可以在字段映射中禁用词频统计：

PUT /my\_index

{

"mappings": {

"doc": {

"properties": {

"text": {

"type": "string",

"index\_options": "docs" https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

}

}

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/scoring-theory.html#CO102-1) | 将参数 index\_options 设置为 docs 可以禁用词频统计及词频位置，这个映射的字段不会计算词的出现次数，对于短语或近似查询也不可用。要求精确查询的 not\_analyzed 字符串字段会默认使用该设置。 |

**逆向文档频率**

词在集合所有文档里出现的频率是多少？频次越高，权重 *越低* 。 常用词如 and 或 the 对相关度贡献很少，因为它们在多数文档中都会出现，一些不常见词如 elastic 或 hippopotamus 可以帮助我们快速缩小范围找到感兴趣的文档。逆向文档频率的计算公式如下：

idf(t) = 1 + log ( numDocs / (docFreq + 1)) https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/scoring-theory.html#CO103-1) | 词 t 的逆向文档频率（ idf ）是：索引中文档数量除以所有包含该词的文档数，然后求其对数。 |

**字段长度归一值**

字段的长度是多少？ 字段越短，字段的权重 *越高* 。如果词出现在类似标题 title 这样的字段，要比它出现在内容 body 这样的字段中的相关度更高。字段长度的归一值公式如下：

norm(d) = 1 / √numTerms https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/scoring-theory.html#CO104-1) | 字段长度归一值（ norm ）是字段中词数平方根的倒数。 |

字段长度的归一值对全文搜索非常重要， 许多其他字段不需要有归一值。无论文档是否包括这个字段，索引中每个文档的每个 string 字段都大约占用 1 个 byte 的空间。对于 not\_analyzed 字符串字段的归一值默认是禁用的，而对于 analyzed 字段也可以通过修改字段映射禁用归一值：

PUT /my\_index

{

"mappings": {

"doc": {

"properties": {

"text": {

"type": "string",

"norms": { "enabled": false } https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

}

}

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/scoring-theory.html#CO105-1) | 这个字段不会将字段长度归一值考虑在内，长字段和短字段会以相同长度计算评分。 |

对于有些应用场景如日志，归一值不是很有用，要关心的只是字段是否包含特殊的错误码或者特定的浏览器唯一标识符。字段的长度对结果没有影响，禁用归一值可以节省大量内存空间。

**结合使用**

以下三个因素——词频（term frequency）、逆向文档频率（inverse document frequency）和字段长度归一值（field-length norm）——是在索引时计算并存储的。 最后将它们结合在一起计算单个词在特定文档中的 *权重* 。

提示

前面公式中提到的 *文档* 实际上是指文档里的某个字段，每个字段都有它自己的倒排索引，因此字段的 TF/IDF 值就是文档的 TF/IDF 值。

当用 explain 查看一个简单的 term 查询时（参见 [explain](https://www.elastic.co/guide/cn/elasticsearch/guide/current/relevance-intro.html#explain) ），可以发现与计算相关度评分的因子就是前面章节介绍的这些：

PUT /my\_index/doc/1

{ "text" : "quick brown fox" }

GET /my\_index/doc/\_search?explain

{

"query": {

"term": {

"text": "fox"

}

}

}

以上请求（简化）的 explanation 解释如下：

weight(text:fox in 0) [PerFieldSimilarity]: 0.15342641 https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

result of:

fieldWeight in 0 0.15342641

product of:

tf(freq=1.0), with freq of 1: 1.0 https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

idf(docFreq=1, maxDocs=1): 0.30685282 https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png

fieldNorm(doc=0): 0.5 https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/4.png

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/scoring-theory.html#CO106-1) | 词 fox 在文档的内部 Lucene doc ID 为 0 ，字段是 text 里的最终评分。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/scoring-theory.html#CO106-2) | 词 fox 在该文档 text 字段中只出现了一次。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/scoring-theory.html#CO106-3) | fox 在所有文档 text 字段索引的逆向文档频率。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/4.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/scoring-theory.html#CO106-4) | 该字段的字段长度归一值。 |

当然，查询通常不止一个词，所以需要一种合并多词权重的方式——向量空间模型（vector space model）。

**向量空间模型**

*向量空间模型（vector space model）* 提供一种比较多词查询的方式，单个评分代表文档与查询的匹配程度，为了做到这点，这个模型将文档和查询都以 *向量（vectors）* 的形式表示：

向量实际上就是包含多个数的一维数组，例如：

[1,2,5,22,3,8]

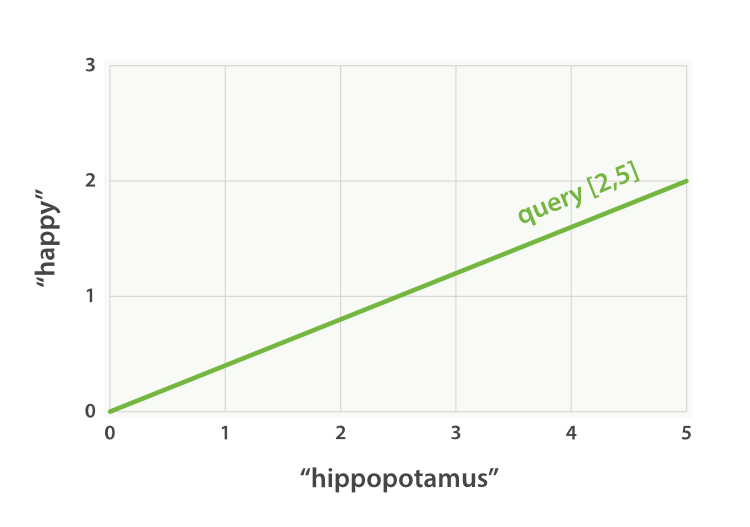
在向量空间模型里， 向量空间模型里的每个数字都代表一个词的 *权重* ，与 [词频/逆向文档频率（term frequency/inverse document frequency）](https://www.elastic.co/guide/cn/elasticsearch/guide/current/scoring-theory.html#tfidf) 计算方式类似。

提示

尽管 TF/IDF 是向量空间模型计算词权重的默认方式，但不是唯一方式。Elasticsearch 还有其他模型如 Okapi-BM25 。TF/IDF 是默认的因为它是个经检验过的简单又高效的算法，可以提供高质量的搜索结果。

设想如果查询 “happy hippopotamus” ，常见词 happy 的权重较低，不常见词 hippopotamus 权重较高，假设 happy 的权重是 2 ， hippopotamus 的权重是 5 ，可以将这个二维向量—— [2,5] ——在坐标系下作条直线，线的起点是 (0,0) 终点是 (2,5) ，如图 [图 27 “表示 “happy hippopotamus” 的二维查询向量”](https://www.elastic.co/guide/cn/elasticsearch/guide/current/scoring-theory.html#img-vector-query) 。

**图 27. 表示 “happy hippopotamus” 的二维查询向量**



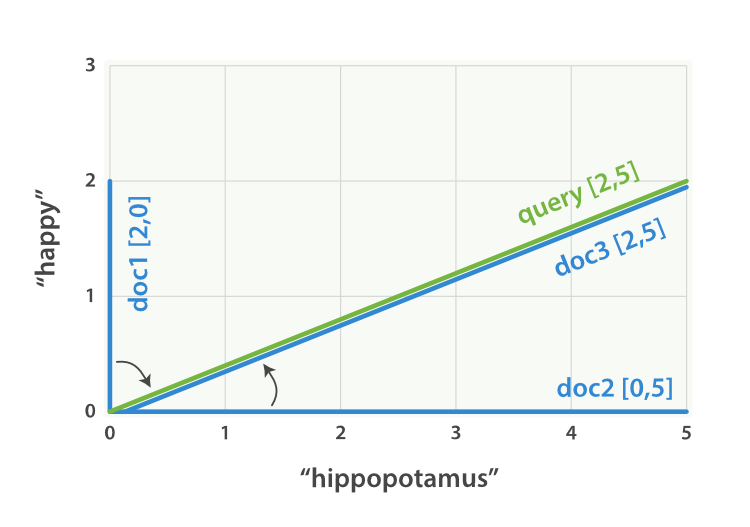
现在，设想我们有三个文档：

1. I am *happy* in summer 。
2. After Christmas I’m a *hippopotamus* 。
3. The *happy hippopotamus* helped Harry 。

可以为每个文档都创建包括每个查询词—— happy 和 hippopotamus ——权重的向量，然后将这些向量置入同一个坐标系中，如图 [图 28 ““happy hippopotamus” 查询及文档向量”](https://www.elastic.co/guide/cn/elasticsearch/guide/current/scoring-theory.html#img-vector-docs) ：

* 文档 1： (happy,\_\_\_\_\_\_\_\_\_\_\_\_) —— [2,0]
* 文档 2： ( \_\_\_ ,hippopotamus) —— [0,5]
* 文档 3： (happy,hippopotamus) —— [2,5]

**图 28. “happy hippopotamus” 查询及文档向量**



向量之间是可以比较的，只要测量查询向量和文档向量之间的角度就可以得到每个文档的相关度，文档 1 与查询之间的角度最大，所以相关度低；文档 2 与查询间的角度较小，所以更相关；文档 3 与查询的角度正好吻合，完全匹配。

提示

在实际中，只有二维向量（两个词的查询）可以在平面上表示，幸运的是， *线性代数* ——作为数学中处理向量的一个分支——为我们提供了计算两个多维向量间角度工具，这意味着可以使用如上同样的方式来解释多个词的查询。

关于比较两个向量的更多信息可以参考 [*余弦近似度（cosine similarity）*](http://en.wikipedia.org/wiki/Cosine_similarity)。

现在已经讲完评分计算的基本理论，我们可以继续了解 Lucene 是如何实现评分计算的。

### Lucene 的实用评分函数

对于多词查询， Lucene 使用 [布尔模型（Boolean model）](https://www.elastic.co/guide/cn/elasticsearch/guide/current/scoring-theory.html#boolean-model) 、 [TF/IDF](https://www.elastic.co/guide/cn/elasticsearch/guide/current/scoring-theory.html#tfidf) 以及 [向量空间模型（vector space model）](https://www.elastic.co/guide/cn/elasticsearch/guide/current/scoring-theory.html#vector-space-model) ，然后将它们组合到单个高效的包里以收集匹配文档并进行评分计算。

一个多词查询

GET /my\_index/doc/\_search

{

"query": {

"match": {

"text": "quick fox"

}

}

}

会在内部被重写为：

GET /my\_index/doc/\_search

{

"query": {

"bool": {

"should": [

{"term": { "text": "quick" }},

{"term": { "text": "fox" }}

]

}

}

}

bool 查询实现了布尔模型，在这个例子中，它会将包括词 quick 和 fox 或两者兼有的文档作为查询结果。

只要一个文档与查询匹配，Lucene 就会为查询计算评分，然后合并每个匹配词的评分结果。这里使用的评分计算公式叫做 *实用评分函数（practical scoring function）* 。看似很高大上，但是别被吓到——多数的组件都已经介绍过，下一步会讨论它引入的一些新元素。

score(q,d) = https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

queryNorm(q) https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

· coord(q,d) https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png

· ∑ ( https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/4.png

tf(t in d) https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/5.png

· idf(t)² https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/6.png

· t.getBoost() https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/7.png

· norm(t,d) https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/8.png

) (t in q) https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/9.png

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/practical-scoring-function.html#CO107-1) | score(q,d) 是文档 d 与查询 q 的相关度评分。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/practical-scoring-function.html#CO107-2) | queryNorm(q) 是 [*查询归一化* 因子](https://www.elastic.co/guide/cn/elasticsearch/guide/current/practical-scoring-function.html#query-norm) （新）。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/practical-scoring-function.html#CO107-3) | coord(q,d) 是 [*协调* 因子](https://www.elastic.co/guide/cn/elasticsearch/guide/current/practical-scoring-function.html#coord) （新）。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/4.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/practical-scoring-function.html#CO107-4) [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/9.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/practical-scoring-function.html#CO107-9) | 查询 q 中每个词 t 对于文档 d 的权重和。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/5.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/practical-scoring-function.html#CO107-5) | tf(t in d) 是词 t 在文档 d 中的 [词频](https://www.elastic.co/guide/cn/elasticsearch/guide/current/scoring-theory.html#tf) 。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/6.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/practical-scoring-function.html#CO107-6) | idf(t) 是词 t 的 [逆向文档频率](https://www.elastic.co/guide/cn/elasticsearch/guide/current/scoring-theory.html#idf) 。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/7.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/practical-scoring-function.html#CO107-7) | t.getBoost() 是查询中使用的 [*boost*](https://www.elastic.co/guide/cn/elasticsearch/guide/current/query-time-boosting.html)（新）。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/8.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/practical-scoring-function.html#CO107-8) | norm(t,d) 是 [字段长度归一值](https://www.elastic.co/guide/cn/elasticsearch/guide/current/scoring-theory.html#field-norm) ，与 [索引时字段层 boost](https://www.elastic.co/guide/cn/elasticsearch/guide/current/practical-scoring-function.html#index-boost) （如果存在）的和（新）。 |

上节已介绍过 score 、 tf 和 idf 。现在来介绍 queryNorm 、 coord 、 t.getBoost 和 norm 。

我们会在本章后面继续探讨 [查询时的权重提升](https://www.elastic.co/guide/cn/elasticsearch/guide/current/query-time-boosting.html) 的问题，但是首先需要了解查询归一化、协调和索引时字段层面的权重提升等概念。

**查询归一因子**

*查询归一因子* （ queryNorm ）试图将查询 *归一化* ， 这样就能将两个不同的查询结果相比较。

提示

尽管查询归一值的目的是为了使查询结果之间能够相互比较，但是它并不十分有效，因为相关度评分 \_score 的目的是为了将当前查询的结果进行排序，比较不同查询结果的相关度评分没有太大意义。

这个因子是在查询过程的最前面计算的，具体的计算依赖于具体查询，一个典型的实现如下：

queryNorm = 1 / √sumOfSquaredWeights https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/practical-scoring-function.html#CO108-1) | sumOfSquaredWeights 是查询里每个词的 IDF 的平方和。 |

提示

相同查询归一化因子会被应用到每个文档，不能被更改，总而言之，可以被忽略。

**查询协调**

*协调因子* （ coord ） 可以为那些查询词包含度高的文档提供奖励，文档里出现的查询词越多，它越有机会成为好的匹配结果。

设想查询 quick brown fox ，每个词的权重都是 1.5 。如果没有协调因子，最终评分会是文档里所有词权重的总和。例如：

* 文档里有 fox → 评分： 1.5
* 文档里有 quick fox → 评分： 3.0
* 文档里有 quick brown fox → 评分： 4.5

协调因子将评分与文档里匹配词的数量相乘，然后除以查询里所有词的数量，如果使用协调因子，评分会变成：

* 文档里有 fox → 评分： 1.5 \* 1 / 3 = 0.5
* 文档里有 quick fox → 评分： 3.0 \* 2 / 3 = 2.0
* 文档里有 quick brown fox → 评分： 4.5 \* 3 / 3 = 4.5

协调因子能使包含所有三个词的文档比只包含两个词的文档评分要高出很多。

回想将查询 quick brown fox 重写成 bool 查询的形式：

GET /\_search

{

"query": {

"bool": {

"should": [

{ "term": { "text": "quick" }},

{ "term": { "text": "brown" }},

{ "term": { "text": "fox" }}

]

}

}

}

bool 查询默认会对所有 should 语句使用协调功能，不过也可以将其禁用。为什么要这样做？通常的回答是——无须这样。查询协调通常是件好事，当使用 bool 查询将多个高级查询如 match 查询包裹的时候，让协调功能开启是有意义的，匹配的语句越多，查询请求与返回文档间的重叠度就越高。

但在某些高级应用中，将协调功能关闭可能更好。设想正在查找同义词 jump 、 leap 和 hop 时，并不关心会出现多少个同义词，因为它们都表示相同的意思，实际上，只有其中一个同义词会出现，这是不使用协调因子的一个好例子：

GET /\_search

{

"query": {

"bool": {

"disable\_coord": true,

"should": [

{ "term": { "text": "jump" }},

{ "term": { "text": "hop" }},

{ "term": { "text": "leap" }}

]

}

}

}

当使用同义词的时候（参照： [同义词](https://www.elastic.co/guide/cn/elasticsearch/guide/current/synonyms.html) ），Lucene 内部是这样的：重写的查询会禁用同义词的协调功能。大多数禁用操作的应用场景是自动处理的，无须为此担心。

**索引时字段层权重提升**

我们会讨论 [查询时的权重提升](https://www.elastic.co/guide/cn/elasticsearch/guide/current/query-time-boosting.html)，让字段 *权重提升* 就是让某个字段比其他字段更重要。 当然在索引时也能做到如此。实际上，权重的提升会被应用到字段的每个词，而不是字段本身。

将提升值存储在索引中无须更多空间，这个字段层索引时的提升值与字段长度归一值（参见 [字段长度归一值](https://www.elastic.co/guide/cn/elasticsearch/guide/current/scoring-theory.html#field-norm) ）一起作为单个字节存于索引， norm(t,d) 是前面公式的返回值。

警告

我们不建议在建立索引时对字段提升权重，有以下原因：

* 将提升值与字段长度归一值合在单个字节中存储会丢失字段长度归一值的精度，这样会导致 Elasticsearch 不知如何区分包含三个词的字段和包含五个词的字段。
* 要想改变索引时的提升值，就必须重新为所有文档建立索引，与此不同的是，查询时的提升值可以随着每次查询的不同而更改。
* 如果一个索引时权重提升的字段有多个值，提升值会按照每个值来自乘，这会导致该字段的权重急剧上升。

[查询时赋予权重](https://www.elastic.co/guide/cn/elasticsearch/guide/current/query-time-boosting.html) 是更为简单、清楚、灵活的选择。

了解了查询归一化、协同和索引时权重提升这些方式后，可以进一步了解相关度计算最有用的工具：查询时的权重提升。

### 查询时权重提升

在 [语句优先级（Prioritizing Clauses）](https://www.elastic.co/guide/cn/elasticsearch/guide/current/multi-query-strings.html#prioritising-clauses) 中，我们解释过如何在搜索时使用 boost 参数让一个查询语句比其他语句更重要。 例如：

GET /\_search

{

"query": {

"bool": {

"should": [

{

"match": {

"title": {

"query": "quick brown fox",

"boost": 2 https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

}

}

},

{

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

"content": "quick brown fox"

}

}

]

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/query-time-boosting.html#CO109-1) | title 查询语句的重要性是 content 查询的 2 倍，因为它的权重提升值为 2 。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/query-time-boosting.html#CO109-2) | 没有设置 boost 的查询语句的值为 1 。 |

*查询时的权重提升* 是可以用来影响相关度的主要工具，任意类型的查询都能接受 boost 参数。 将 boost设置为 2 ，并不代表最终的评分 \_score 是原值的两倍；实际的权重值会经过归一化和一些其他内部优化过程。尽管如此，它确实想要表明一个提升值为 2 的句子的重要性是提升值为 1 语句的两倍。

在实际应用中，无法通过简单的公式得出某个特定查询语句的 “正确” 权重提升值，只能通过不断尝试获得。需要记住的是 boost 只是影响相关度评分的其中一个因子；它还需要与其他因子相互竞争。在前例中， title 字段相对 content 字段可能已经有一个 “缺省的” 权重提升值，这因为在 [字段长度归一值](https://www.elastic.co/guide/cn/elasticsearch/guide/current/scoring-theory.html#field-norm) 中，标题往往比相关内容要短，所以不要想当然的去盲目提升一些字段的权重。选择权重，检查结果，如此反复。

**提升索引权重**

当在多个索引中搜索时， 可以使用参数 indices\_boost 来提升整个索引的权重，在下面例子中，当要为最近索引的文档分配更高权重时，可以这么做：

GET /docs\_2014\_\*/\_search https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

{

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

"docs\_2014\_10": 3,

"docs\_2014\_09": 2

},

"query": {

"match": {

"text": "quick brown fox"

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/query-time-boosting.html#CO110-1) | 这个多索引查询涵盖了所有以字符串 docs\_2014\_ 开始的索引。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/query-time-boosting.html#CO110-2) | 其中，索引 docs\_2014\_10 中的所有文件的权重是 3 ，索引 docs\_2014\_09 中是 2 ，其他所有匹配的索引权重为默认值 1 。 |

**t.getBoost()**

这些提升值在 Lucene 的 [实用评分函数](https://www.elastic.co/guide/cn/elasticsearch/guide/current/practical-scoring-function.html) 中可以通过 t.getBoost() 获得。 权重提升不会被应用于它在查询表达式中出现的层，而是会被合并下转至每个词中。 t.getBoost() 始终返回当前词的权重或当前分析链上查询的权重。

提示

实际上，要想解读 [explain](https://www.elastic.co/guide/cn/elasticsearch/guide/current/relevance-intro.html#explain) 的输出是相当复杂的，在 explanation 里面完全看不到 boost值，也完全无法访问上面提到的 t.getBoost() 方法，权重值融合在 [queryNorm](https://www.elastic.co/guide/cn/elasticsearch/guide/current/practical-scoring-function.html#query-norm) 中并应用到每个词。尽管说， queryNorm 对于每个词都是相同的，还是会发现一个权重提升过的词的 queryNorm 值要高于一个没有提升过的。

### 使用查询结构修改相关度

Elasticsearch 的查询表达式相当灵活， 可以通过调整查询结构中查询语句的所处层次，从而或多或少改变其重要性，比如，设想下面这个查询：

quick OR brown OR red OR fox

可以将所有词都放在 bool 查询的同一层中：

GET /\_search

{

"query": {

"bool": {

"should": [

{ "term": { "text": "quick" }},

{ "term": { "text": "brown" }},

{ "term": { "text": "red" }},

{ "term": { "text": "fox" }}

]

}

}

}

这个查询可能最终给包含 quick 、 red 和 brown 的文档评分与包含 quick 、 red 、 fox 文档的评分相同，这里 *Red* 和 *brown* 是同义词，可能只需要保留其中一个，而我们真正要表达的意思是想做以下查询：

quick OR (brown OR red) OR fox

根据标准的布尔逻辑，这与原始的查询是完全一样的，但是我们已经在 [组合查询（Combining Queries）](https://www.elastic.co/guide/cn/elasticsearch/guide/current/bool-query.html)中看到， bool 查询不关心文档匹配的 *程度* ，只关心是否能匹配。

上述查询有个更好的方式：

GET /\_search

{

"query": {

"bool": {

"should": [

{ "term": { "text": "quick" }},

{ "term": { "text": "fox" }},

{

"bool": {

"should": [

{ "term": { "text": "brown" }},

{ "term": { "text": "red" }}

]

}

}

]

}

}

}

现在， red 和 brown 处于相互竞争的层次， quick 、 fox 以及 red OR brown 则是处于顶层且相互竞争的词。

我们已经讨论过如何使用 [match](https://www.elastic.co/guide/cn/elasticsearch/guide/current/match-query.html) 、[multi\_match](https://www.elastic.co/guide/cn/elasticsearch/guide/current/multi-match-query.html) 、[term](https://www.elastic.co/guide/cn/elasticsearch/guide/current/term-vs-full-text.html) 、[bool](https://www.elastic.co/guide/cn/elasticsearch/guide/current/bool-query.html) 和 [dis\_max](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_best_fields.html#dis-max-query) 查询修改相关度评分。本章后面的内容会介绍另外三个与相关度评分有关的查询： boosting 查询、 constant\_score 查询和 function\_score 查询。

### Not Quite Not

在互联网上搜索 “Apple”，返回的结果很可能是一个公司、水果和各种食谱。 我们可以在 bool 查询中用 must\_not 语句来排除像 pie 、 tart 、 crumble 和 tree 这样的词，从而将查询结果的范围缩小至只返回与 “Apple” （苹果）公司相关的结果：

GET /\_search

{

"query": {

"bool": {

"must": {

"match": {

"text": "apple"

}

},

"must\_not": {

"match": {

"text": "pie tart fruit crumble tree"

}

}

}

}

}

但谁又敢保证在排除 tree 或 crumble 这种词后，不会错失一个与苹果公司特别相关的文档呢？有时， must\_not 条件会过于严格。

**权重提升查询**

[boosting 查询](https://www.elastic.co/guide/en/elasticsearch/reference/master/query-dsl-boosting-query.html) 恰恰能解决这个问题。 它仍然允许我们将关于水果或甜点的结果包括到结果中，但是使它们降级——即降低它们原来可能应有的排名：

GET /\_search

{

"query": {

"boosting": {

"positive": {

"match": {

"text": "apple"

}

},

"negative": {

"match": {

"text": "pie tart fruit crumble tree"

}

},

"negative\_boost": 0.5

}

}

}

它接受 positive 和 negative 查询。只有那些匹配 positive 查询的文档罗列出来，对于那些同时还匹配 negative 查询的文档将通过文档的原始 \_score 与 negative\_boost 相乘的方式降级后的结果。

为了达到效果， negative\_boost 的值必须小于 1.0 。在这个示例中，所有包含负向词的文档评分 \_score都会减半。

### 忽略 TF/IDF

有时候我们根本不关心 TF/IDF ， 只想知道一个词是否在某个字段中出现过。可能搜索一个度假屋并希望它能尽可能有以下设施：

* WiFi
* Garden（花园）
* Pool（游泳池）

这个度假屋的文档如下：

{ "description": "A delightful four-bedroomed house with ... " }

可以用简单的 match 查询进行匹配：

GET /\_search

{

"query": {

"match": {

"description": "wifi garden pool"

}

}

}

但这并不是真正的 *全文搜索* ，此种情况下，TF/IDF 并无用处。我们既不关心 wifi 是否为一个普通词，也不关心它在文档中出现是否频繁，关心的只是它是否曾出现过。实际上，我们希望根据房屋不同设施的数量对其排名——设施越多越好。如果设施出现，则记 1 分，不出现记 0 分。

**constant\_score 查询**

在 [constant\_score](https://www.elastic.co/guide/en/elasticsearch/reference/master/query-dsl-constant-score-query.html) 查询中，它可以包含查询或过滤，为任意一个匹配的文档指定评分 1 ，忽略 TF/IDF 信息：

GET /\_search

{

"query": {

"bool": {

"should": [

{ "constant\_score": {

"query": { "match": { "description": "wifi" }}

}},

{ "constant\_score": {

"query": { "match": { "description": "garden" }}

}},

{ "constant\_score": {

"query": { "match": { "description": "pool" }}

}}

]

}

}

}

或许不是所有的设施都同等重要——对某些用户来说有些设施更有价值。如果最重要的设施是游泳池，那我们可以为更重要的设施增加权重：

GET /\_search

{

"query": {

"bool": {

"should": [

{ "constant\_score": {

"query": { "match": { "description": "wifi" }}

}},

{ "constant\_score": {

"query": { "match": { "description": "garden" }}

}},

{ "constant\_score": {

"boost": 2 https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

"query": { "match": { "description": "pool" }}

}}

]

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/ignoring-tfidf.html#CO111-1) | pool 语句的权重提升值为 2 ，而其他的语句为 1 。 |

注意

最终的评分并不是所有匹配语句的简单求和， [协调因子（coordination factor）](https://www.elastic.co/guide/cn/elasticsearch/guide/current/practical-scoring-function.html#coord) 和 [查询归一化因子（query normalization factor）](https://www.elastic.co/guide/cn/elasticsearch/guide/current/practical-scoring-function.html#query-norm) 仍然会被考虑在内。

我们可以给 features 字段加上 not\_analyzed 类型来提升度假屋文档的匹配能力：

{ "features": [ "wifi", "pool", "garden" ] }

默认情况下，一个 not\_analyzed 字段会禁用 [字段长度归一值（field-length norms）](https://www.elastic.co/guide/cn/elasticsearch/guide/current/scoring-theory.html#field-norm) 的功能， 并将 index\_options 设为 docs 选项，禁用 [词频](https://www.elastic.co/guide/cn/elasticsearch/guide/current/scoring-theory.html#tf) ，但还是存在问题：每个词的 [倒排文档频率](https://www.elastic.co/guide/cn/elasticsearch/guide/current/scoring-theory.html#idf) 仍然会被考虑。

可以采用与之前相同的方法 constant\_score 查询来解决这个问题：

GET /\_search

{

"query": {

"bool": {

"should": [

{ "constant\_score": {

"query": { "match": { "features": "wifi" }}

}},

{ "constant\_score": {

"query": { "match": { "features": "garden" }}

}},

{ "constant\_score": {

"boost": 2

"query": { "match": { "features": "pool" }}

}}

]

}

}

}

实际上，每个设施都应该看成一个过滤器，对于度假屋来说要么具有某个设施要么没有——过滤器因为其性质天然合适。而且，如果使用过滤器，我们还可以利用缓存。

这里的问题是：过滤器无法计算评分。这样就需要寻求一种方式将过滤器和查询间的差异抹平。 function\_score 查询不仅正好可以扮演这个角色，而且有更强大的功能。

### function\_score 查询

[function\_score 查询](https://www.elastic.co/guide/en/elasticsearch/reference/master/query-dsl-function-score-query.html) 是用来控制评分过程的终极武器，它允许为每个与主查询匹配的文档应用一个函数，以达到改变甚至完全替换原始查询评分 \_score 的目的。

实际上，也能用过滤器对结果的 *子集* 应用不同的函数，这样一箭双雕：既能高效评分，又能利用过滤器缓存。

Elasticsearch 预定义了一些函数：

weight

为每个文档应用一个简单而不被规范化的权重提升值：当 weight 为 2 时，最终结果为 2 \* \_score 。

field\_value\_factor

使用这个值来修改 \_score ，如将 popularity 或 votes （受欢迎或赞）作为考虑因素。

random\_score

为每个用户都使用一个不同的随机评分对结果排序，但对某一具体用户来说，看到的顺序始终是一致的。

*衰减函数* —— linear 、 exp 、 gauss

将浮动值结合到评分 \_score 中，例如结合 publish\_date 获得最近发布的文档，结合 geo\_location 获得更接近某个具体经纬度（lat/lon）地点的文档，结合 price 获得更接近某个特定价格的文档。

script\_score

如果需求超出以上范围时，用自定义脚本可以完全控制评分计算，实现所需逻辑。

如果没有 function\_score 查询，就不能将全文查询与最新发生这种因子结合在一起评分，而不得不根据评分 \_score 或时间 date 进行排序；这会相互影响抵消两种排序各自的效果。这个查询可以使两个效果融合：可以仍然根据全文相关度进行排序，但也会同时考虑最新发布文档、流行文档、或接近用户希望价格的产品。正如所设想的，查询要考虑所有这些因素会非常复杂，让我们先从简单的例子开始，然后顺着梯子慢慢向上爬，增加复杂度。

### 按受欢迎度提升权重

设想有个网站供用户发布博客并且可以让他们为自己喜欢的博客点赞， 我们希望将更受欢迎的博客放在搜索结果列表中相对较上的位置，同时全文搜索的评分仍然作为相关度的主要排序依据，可以简单的通过存储每个博客的点赞数来实现它：

PUT /blogposts/post/1

{

"title": "About popularity",

"content": "In this post we will talk about...",

"votes": 6

}

在搜索时，可以将 function\_score 查询与 field\_value\_factor 结合使用， 即将点赞数与全文相关度评分结合：

GET /blogposts/post/\_search

{

"query": {

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

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

"multi\_match": {

"query": "popularity",

"fields": [ "title", "content" ]

}

},

"field\_value\_factor": { https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png

"field": "votes" https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/4.png

}

}

}

}

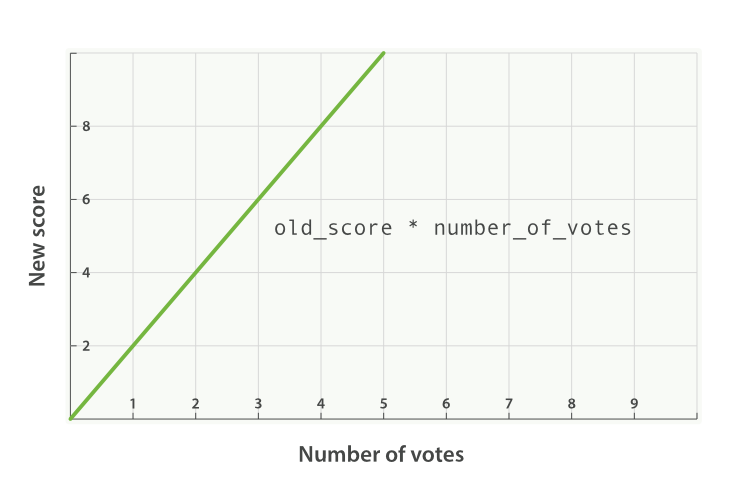
|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/boosting-by-popularity.html#CO112-1) | function\_score 查询将主查询和函数包括在内。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/boosting-by-popularity.html#CO112-2) | 主查询优先执行。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/boosting-by-popularity.html#CO112-3) | field\_value\_factor 函数会被应用到每个与主 query 匹配的文档。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/4.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/boosting-by-popularity.html#CO112-4) | 每个文档的 votes 字段都 *必须* 有值供 function\_score 计算。如果 *没有* 文档的 votes 字段有值，那么就 *必须* 使用 [missing 属性](https://www.elastic.co/guide/en/elasticsearch/reference/master/query-dsl-function-score-query.html#function-field-value-factor) 提供的默认值来进行评分计算。 |

在前面示例中，每个文档的最终评分 \_score 都做了如下修改：

new\_score = old\_score \* number\_of\_votes

然而这并不会带来出人意料的好结果，全文评分 \_score 通常处于 0 到 10 之间，如下图 [图 29 “受欢迎度的线性关系基于 \_score 的原始值 2.0”](https://www.elastic.co/guide/cn/elasticsearch/guide/current/boosting-by-popularity.html#img-popularity-linear) 中，有 10 个赞的博客会掩盖掉全文评分，而 0 个赞的博客的评分会被置为 0 。

**图 29. 受欢迎度的线性关系基于 \_score 的原始值 2.0**



**modifier**

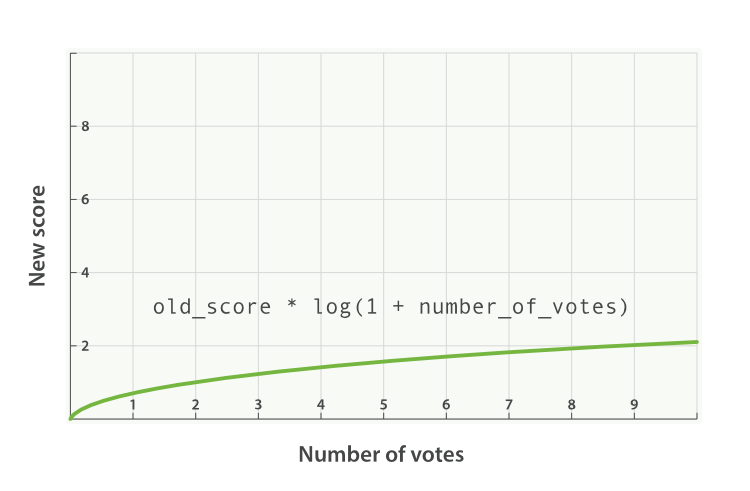
一种融入受欢迎度更好方式是用 modifier 平滑 votes 的值。 换句话说，我们希望最开始的一些赞更重要，但是其重要性会随着数字的增加而降低。 0 个赞与 1 个赞的区别应该比 10 个赞与 11 个赞的区别大很多。

对于上述情况，典型的 modifier 应用是使用 log1p 参数值，公式如下：

new\_score = old\_score \* log(1 + number\_of\_votes)

log 对数函数使 votes 赞字段的评分曲线更平滑，如图 [图 30 “受欢迎度的对数关系基于 \_score 的原始值 2.0”](https://www.elastic.co/guide/cn/elasticsearch/guide/current/boosting-by-popularity.html#img-popularity-log) ：

**图 30. 受欢迎度的对数关系基于 \_score 的原始值 2.0**



带 modifier 参数的请求如下：

GET /blogposts/post/\_search

{

"query": {

"function\_score": {

"query": {

"multi\_match": {

"query": "popularity",

"fields": [ "title", "content" ]

}

},

"field\_value\_factor": {

"field": "votes",

"modifier": "log1p" https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

}

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/boosting-by-popularity.html#CO113-1) | modifier 为 log1p 。 |

修饰语 modifier 的值可以为： none （默认状态）、 log 、 log1p 、 log2p 、 ln 、 ln1p 、 ln2p 、 square 、 sqrt 以及 reciprocal 。想要了解更多信息请参照： [field\_value\_factor 文档](https://www.elastic.co/guide/en/elasticsearch/reference/master/query-dsl-function-score-query.html#function-field-value-factor).

**factor**

可以通过将 votes 字段与 factor 的积来调节受欢迎程度效果的高低：

GET /blogposts/post/\_search

{

"query": {

"function\_score": {

"query": {

"multi\_match": {

"query": "popularity",

"fields": [ "title", "content" ]

}

},

"field\_value\_factor": {

"field": "votes",

"modifier": "log1p",

"factor": 2 https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

}

}

}

}

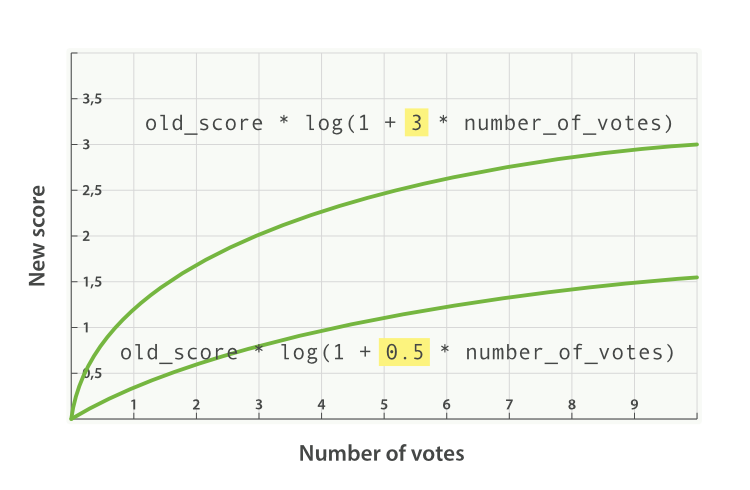
|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/boosting-by-popularity.html#CO114-1) | 双倍效果。 |

添加了 factor 会使公式变成这样：

new\_score = old\_score \* log(1 + factor \* number\_of\_votes)

factor 值大于 1 会提升效果， factor 值小于 1 会降低效果，如图 [图 31 “受欢迎度的对数关系基于多个不同因子”](https://www.elastic.co/guide/cn/elasticsearch/guide/current/boosting-by-popularity.html#img-popularity-factor) 。

**图 31. 受欢迎度的对数关系基于多个不同因子**



**boost\_mode**

或许将全文评分与 field\_value\_factor 函数值乘积的效果仍然可能太大， 我们可以通过参数 boost\_mode来控制函数与查询评分 \_score 合并后的结果，参数接受的值为：

multiply

评分 \_score 与函数值的积（默认）

sum

评分 \_score 与函数值的和

min

评分 \_score 与函数值间的较小值

max

评分 \_score 与函数值间的较大值

replace

函数值替代评分 \_score

与使用乘积的方式相比，使用评分 \_score 与函数值求和的方式可以弱化最终效果，特别是使用一个较小 factor 因子时：

GET /blogposts/post/\_search

{

"query": {

"function\_score": {

"query": {

"multi\_match": {

"query": "popularity",

"fields": [ "title", "content" ]

}

},

"field\_value\_factor": {

"field": "votes",

"modifier": "log1p",

"factor": 0.1

},

"boost\_mode": "sum" https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

}

}

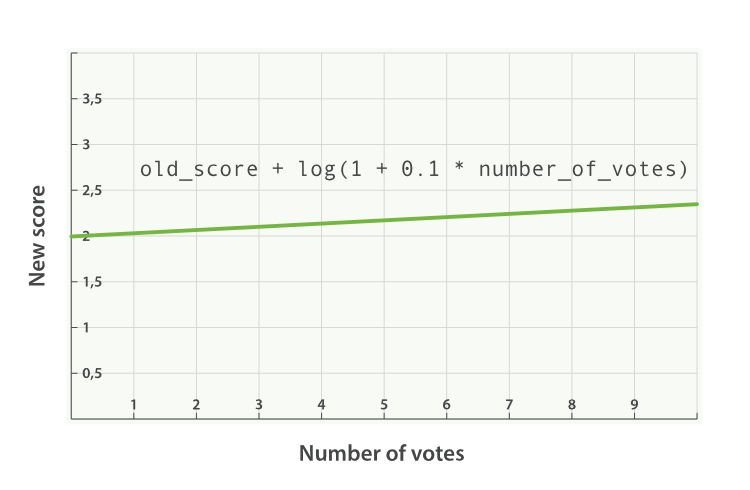
}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/boosting-by-popularity.html#CO115-1) | 评分 \_score 与函数值的积。 |

之前请求的公式现在变成下面这样（参见 [图 32 “使用 sum 结合受欢迎程度”](https://www.elastic.co/guide/cn/elasticsearch/guide/current/boosting-by-popularity.html#img-popularity-sum) ）：

new\_score = old\_score + log(1 + 0.1 \* number\_of\_votes)

**图 32. 使用 sum 结合受欢迎程度**



**max\_boost**

最后，可以使用 max\_boost 参数限制一个函数的最大效果：

GET /blogposts/post/\_search

{

"query": {

"function\_score": {

"query": {

"multi\_match": {

"query": "popularity",

"fields": [ "title", "content" ]

}

},

"field\_value\_factor": {

"field": "votes",

"modifier": "log1p",

"factor": 0.1

},

"boost\_mode": "sum",

"max\_boost": 1.5 https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/boosting-by-popularity.html#CO116-1) | 无论 field\_value\_factor 函数的结果如何，最终结果都不会大于 1.5 。 |

注意

max\_boost 只对函数的结果进行限制，不会对最终评分 \_score 产生直接影响。

### 过滤集提升权重

回到 [忽略 TF/IDF](https://www.elastic.co/guide/cn/elasticsearch/guide/current/ignoring-tfidf.html) 里处理过的问题，我们希望根据每个度假屋的特性数量来评分， 当时我们希望能用缓存的过滤器来影响评分，现在 function\_score 查询正好可以完成这件事情。

到目前为止，我们展现的都是为所有文档应用单个函数的使用方式，现在会用过滤器将结果划分为多个子集（每个特性一个过滤器），并为每个子集使用不同的函数。

在下面例子中，我们会使用 weight 函数，它与 boost 参数类似可以用于任何查询。有一点区别是 weight没有被 Luence 归一化成难以理解的浮点数，而是直接被应用。

查询的结构需要做相应变更以整合多个函数：

GET /\_search

{

"query": {

"function\_score": {

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

"term": { "city": "Barcelona" }

},

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

{

"filter": { "term": { "features": "wifi" }}, https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png

"weight": 1

},

{

"filter": { "term": { "features": "garden" }}, https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/4.png

"weight": 1

},

{

"filter": { "term": { "features": "pool" }}, https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/5.png

"weight": 2 https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/6.png

}

],

"score\_mode": "sum", https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/7.png

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/function-score-filters.html#CO117-1) | function\_score 查询有个 filter 过滤器而不是 query 查询。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/function-score-filters.html#CO117-2) | functions 关键字存储着一个将被应用的函数列表。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/function-score-filters.html#CO117-3) [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/4.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/function-score-filters.html#CO117-4)[https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/5.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/function-score-filters.html#CO117-5) | 函数会被应用于和 filter 过滤器（可选的）匹配的文档。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/6.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/function-score-filters.html#CO117-6) | pool 比其他特性更重要，所以它有更高 weight 。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/7.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/function-score-filters.html#CO117-7) | score\_mode 指定各个函数的值进行组合运算的方式。 |

这个新特性需要注意的地方会在以下小节介绍。

**过滤 vs. 查询**

首先要注意的是 filter 过滤器代替了 query 查询， 在本例中，我们无须使用全文搜索，只想找到 city字段中包含 Barcelona 的所有文档，逻辑用过滤比用查询表达更清晰。过滤器返回的所有文档的评分 \_score 的值为 1 。 function\_score 查询接受 query 或 filter ，如果没有特别指定，则默认使用 match\_all 查询。

**函数 functions**

functions 关键字保持着一个将要被使用的函数列表。 可以为列表里的每个函数都指定一个 filter 过滤器，在这种情况下，函数只会被应用到那些与过滤器匹配的文档，例子中，我们为与过滤器匹配的文档指定权重值 weight 为 1 （为与 pool 匹配的文档指定权重值为 2 ）。

**评分模式 score\_mode**

每个函数返回一个结果，所以需要一种将多个结果缩减到单个值的方式，然后才能将其与原始评分 \_score合并。评分模式 score\_mode 参数正好扮演这样的角色， 它接受以下值：

multiply

函数结果求积（默认）。

sum

函数结果求和。

avg

函数结果的平均值。

max

函数结果的最大值。

min

函数结果的最小值。

first

使用首个函数（可以有过滤器，也可能没有）的结果作为最终结果

在本例中，我们将每个过滤器匹配结果的权重 weight 求和，并将其作为最终评分结果，所以会使用 sum评分模式。

不与任何过滤器匹配的文档会保有其原始评分， \_score 值的为 1 。

### 随机评分

你可能会想知道 *一致随机评分（consistently random scoring）* 是什么，又为什么会使用它。 之前的例子是个很好的应用场景，前例中所有的结果都会返回 1 、 2 、 3 、 4 或 5 这样的最终评分 \_score ，可能只有少数房子的评分是 5 分，而有大量房子的评分是 2 或 3 。

作为网站的所有者，总会希望让广告有更高的展现率。在当前查询下，有相同评分 \_score 的文档会每次都以相同次序出现，为了提高展现率，在此引入一些随机性可能会是个好主意，这能保证有相同评分的文档都能有均等相似的展现机率。

我们想让每个用户看到不同的随机次序，但也同时希望如果是同一用户翻页浏览时，结果的相对次序能始终保持一致。这种行为被称为 *一致随机（consistently random）* 。

random\_score 函数会输出一个 0 到 1 之间的数， 当种子 seed 值相同时，生成的随机结果是一致的，例如，将用户的会话 ID 作为 seed ：

GET /\_search

{

"query": {

"function\_score": {

"filter": {

"term": { "city": "Barcelona" }

},

"functions": [

{

"filter": { "term": { "features": "wifi" }},

"weight": 1

},

{

"filter": { "term": { "features": "garden" }},

"weight": 1

},

{

"filter": { "term": { "features": "pool" }},

"weight": 2

},

{

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

"seed": "the users session id" https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

}

}

],

"score\_mode": "sum"

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/random-scoring.html#CO118-1) | random\_score 语句没有任何过滤器 filter ，所以会被应用到所有文档。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/random-scoring.html#CO118-2) | 将用户的会话 ID 作为种子 seed ，让该用户的随机始终保持一致，相同的种子 seed 会产生相同的随机结果。 |

当然，如果增加了与查询匹配的新文档，无论是否使用一致随机，其结果顺序都会发生变化。

### 越近越好

很多变量都可以影响用户对于度假屋的选择， 也许用户希望离市中心近点，但如果价格足够便宜，也有可能选择一个更远的住处，也有可能反过来是正确的：愿意为最好的位置付更多的价钱。

如果我们添加过滤器排除所有市中心方圆 1 千米以外的度假屋，或排除所有每晚价格超过 £100 英镑的，我们可能会将用户愿意考虑妥协的那些选择排除在外。

function\_score 查询会提供一组 *衰减函数（decay functions）* ， 让我们有能力在两个滑动标准，如地点和价格，之间权衡。

有三种衰减函数—— linear 、 exp 和 gauss （线性、指数和高斯函数），它们可以操作数值、时间以及经纬度地理坐标点这样的字段。所有三个函数都能接受以下参数：

origin

*中心点* 或字段可能的最佳值，落在原点 origin 上的文档评分 \_score 为满分 1.0 。

scale

衰减率，即一个文档从原点 origin 下落时，评分 \_score 改变的速度。（例如，每 £10 欧元或每 100 米）。

decay

从原点 origin 衰减到 scale 所得的评分 \_score ，默认值为 0.5 。

offset

以原点 origin 为中心点，为其设置一个非零的偏移量 offset 覆盖一个范围，而不只是单个原点。在范围 -offset <= origin <= +offset 内的所有评分 \_score 都是 1.0 。

这三个函数的唯一区别就是它们衰减曲线的形状，用图来说明会更为直观（参见 [图 33 “衰减函数曲线”](https://www.elastic.co/guide/cn/elasticsearch/guide/current/decay-functions.html#img-decay-functions)）。

**图 33. 衰减函数曲线**

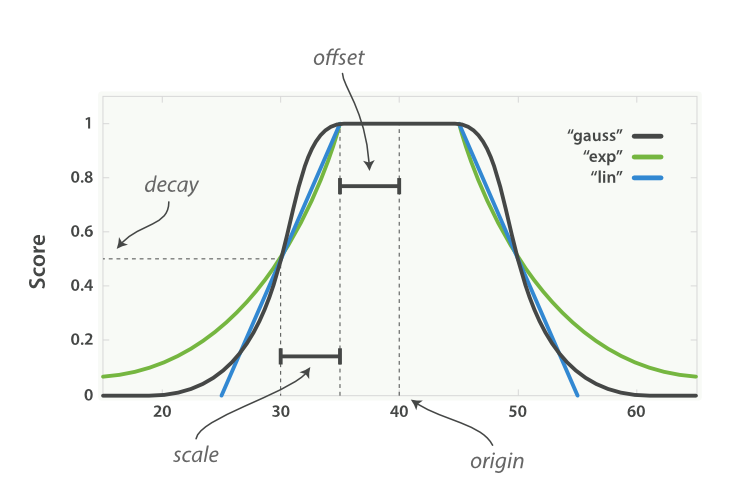


图 [图 33 “衰减函数曲线”](https://www.elastic.co/guide/cn/elasticsearch/guide/current/decay-functions.html#img-decay-functions) 中所有曲线的原点 origin （即中心点）的值都是 40 ， offset 是 5 ，也就是在范围 40 - 5 <= value <= 40 + 5 内的所有值都会被当作原点 origin 处理——所有这些点的评分都是满分 1.0 。

在此范围之外，评分开始衰减，衰减率由 scale 值（此例中的值为 5 ）和 衰减值 decay （此例中为默认值 0.5 ）共同决定。结果是所有三个曲线在 origin +/- (offset + scale) 处的评分都是 0.5 ，即点 30和 50 处。

linear 、 exp 和 gauss （线性、指数和高斯）函数三者之间的区别在于范围（ origin +/- (offset + scale) ）之外的曲线形状：

* linear 线性函数是条直线，一旦直线与横轴 0 相交，所有其他值的评分都是 0.0 。
* exp 指数函数是先剧烈衰减然后变缓。
* gauss 高斯函数是钟形的——它的衰减速率是先缓慢，然后变快，最后又放缓。

选择曲线的依据完全由期望评分 \_score 的衰减速率来决定，即距原点 origin 的值。

回到我们的例子：用户希望租一个离伦敦市中心近（ { "lat": 51.50, "lon": 0.12} ）且每晚不超过 £100 英镑的度假屋，而且与距离相比， 我们的用户对价格更为敏感，这样查询可以写成：

GET /\_search

{

"query": {

"function\_score": {

"functions": [

{

"gauss": {

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

"origin": { "lat": 51.5, "lon": 0.12 },

"offset": "2km",

"scale": "3km"

}

}

},

{

"gauss": {

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

"origin": "50", https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png

"offset": "50",

"scale": "20"

}

},

"weight": 2 https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/4.png

}

]

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/decay-functions.html#CO119-1) | location 字段以地理坐标点 geo\_point 映射。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/decay-functions.html#CO119-2) | price 字段是数值。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/decay-functions.html#CO119-3) | 参见 [理解价格语句](https://www.elastic.co/guide/cn/elasticsearch/guide/current/Understanding-the-price-Clause.html) ，理解 origin 为什么是 50 而不是 100 。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/4.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/decay-functions.html#CO119-4) | price 语句是 location 语句权重的两倍。 |

location 语句可以简单理解为：

* 以伦敦市中作为原点 origin 。
* 所有距原点 origin 2km 范围内的位置的评分是 1.0 。
* 距中心 5km （ offset + scale ）的位置的评分是 0.5 。

### 理解 price 价格语句

price 语句使用了一个小技巧：用户希望选择 £100 英镑以下的度假屋，但是例子中的原点被设置成 £50 英镑，价格不能为负，但肯定是越低越好，所以 £0 到 £100 英镑内的所有价格都认为是比较好的。

如果我们将原点 origin 被设置成 £100 英镑，那么低于 £100 英镑的度假屋的评分会变低，与其这样不如将原点 origin 和偏移量 offset 同时设置成 £50 英镑，这样就能使只有在价格高于 £100 英镑（ origin + offset ）时评分才会变低。

提示

weight 参数可以被用来调整每个语句的贡献度，权重 weight 的默认值是 1.0 。这个值会先与每个句子的评分相乘，然后再通过 score\_mode 的设置方式合并。

### 脚本评分

最后，如果所有 function\_score 内置的函数都无法满足应用场景，可以使用 script\_score 函数自行实现逻辑。

举个例子，想将利润空间作为因子加入到相关度评分计算，在业务中，利润空间和以下三点相关：

* price 度假屋每晚的价格。
* 会员用户的级别——某些等级的用户可以在每晚房价高于某个 threshold 阀值价格的时候享受折扣 discount 。
* 用户享受折扣后，经过议价的每晚房价的利润 margin 。

计算每个度假屋利润的算法如下：

if (price < threshold) {

profit = price \* margin

} else {

profit = price \* (1 - discount) \* margin;

}

我们很可能不想用绝对利润作为评分，这会弱化其他如地点、受欢迎度和特性等因子的作用，而是将利润用目标利润 target 的百分比来表示，高于 目标的利润空间会有一个正向评分（大于 1.0 ），低于目标的利润空间会有一个负向分数（小于 1.0 ）：

if (price < threshold) {

profit = price \* margin

} else {

profit = price \* (1 - discount) \* margin

}

return profit / target

Elasticsearch 里使用 [Groovy](http://groovy.codehaus.org/) 作为默认的脚本语言，它与JavaScript很像， 上面这个算法用 Groovy 脚本表示如下：

price = doc['price'].value https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

margin = doc['margin'].value https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

if (price < threshold) { https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png

return price \* margin / target

}

return price \* (1 - discount) \* margin / target https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/4.png

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/script-score.html#CO120-1) [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/script-score.html#CO120-2) | price 和 margin 变量可以分别从文档的 price 和 margin 字段提取。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/script-score.html#CO120-3) [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/4.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/script-score.html#CO120-4) | threshold 、 discount 和 target 是作为参数 params 传入的。 |

最终我们将 script\_score 函数与其他函数一起使用：

GET /\_search

{

"function\_score": {

"functions": [

{ ...location clause... }, https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

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

{

"script\_score": {

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

"threshold": 80,

"discount": 0.1,

"target": 10

},

"script": "price = doc['price'].value; margin = doc['margin'].value;

if (price < threshold) { return price \* margin / target };

return price \* (1 - discount) \* margin / target;" https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/4.png

}

}

]

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/script-score.html#CO121-1) [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/script-score.html#CO121-2) | location 和 price 语句在 [衰减函数](https://www.elastic.co/guide/cn/elasticsearch/guide/current/decay-functions.html) 中解释过。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/script-score.html#CO121-3) | 将这些变量作为参数 params 传递，我们可以查询时动态改变脚本无须重新编译。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/4.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/script-score.html#CO121-4) | JSON 不能接受内嵌的换行符，脚本中的换行符可以用 \n 或 ; 符号替代。 |

这个查询根据用户对地点和价格的需求，返回用户最满意的文档，同时也考虑到我们对于盈利的要求。

提示

script\_score 函数提供了巨大的灵活性， 可以通过脚本访问文档里的所有字段、当前评分 \_score 甚至词频、逆向文档频率和字段长度规范值这样的信息（参见 see [脚本对文本评分](https://www.elastic.co/guide/en/elasticsearch/reference/master/modules-advanced-scripting.html)）。

有人说使用脚本对性能会有影响，如果确实发现脚本执行较慢，可以有以下三种选择：

* 尽可能多的提前计算各种信息并将结果存入每个文档中。
* Groovy 很快，但没 Java 快。 可以将脚本用原生的 Java 脚本重新实现。（参见 [原生 Java 脚本](https://www.elastic.co/guide/en/elasticsearch/reference/master/modules-scripting-native.html)）。
* 仅对那些最佳评分的文档应用脚本，使用 [重新评分](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_Improving_Performance.html#rescore-api) 中提到的 rescore 功能。

### 可插拔的相似度算法

在进一步讨论相关度和评分之前，我们会以一个更高级的话题结束本章节的内容：可插拔的相似度算法（Pluggable Similarity Algorithms）。 Elasticsearch 将 [实用评分算法](https://www.elastic.co/guide/cn/elasticsearch/guide/current/practical-scoring-function.html) 作为默认相似度算法，它也能够支持其他的一些算法，这些算法可以参考 [相似度模块](https://www.elastic.co/guide/en/elasticsearch/reference/master/index-modules-similarity.html#configuration) 文档。

**Okapi BM25**

能与 TF/IDF 和向量空间模型媲美的就是 [*Okapi BM25*](http://en.wikipedia.org/wiki/Okapi_BM25) ，它被认为是 *当今最先进的* 排序函数。 BM25 源自 [概率相关模型（probabilistic relevance model）](http://en.wikipedia.org/wiki/Probabilistic_relevance_model) ，而不是向量空间模型，但这个算法也和 Lucene 的实用评分函数有很多共通之处。

BM25 同样使用词频、逆向文档频率以及字段长归一化，但是每个因子的定义都有细微区别。与其详细解释 BM25 公式，倒不如将关注点放在 BM25 所能带来的实际好处上。

**词频饱和度**

TF/IDF 和 BM25 同样使用 [逆向文档频率](https://www.elastic.co/guide/cn/elasticsearch/guide/current/scoring-theory.html#idf) 来区分普通词（不重要）和非普通词（重要）， 同样认为（参见 [词频](https://www.elastic.co/guide/cn/elasticsearch/guide/current/scoring-theory.html#tf) ）文档里的某个词出现次数越频繁，文档与这个词就越相关。

不幸的是，普通词随处可见， 实际上一个普通词在同一个文档中大量出现的作用会由于该词在 *所有* 文档中的大量出现而被抵消掉。

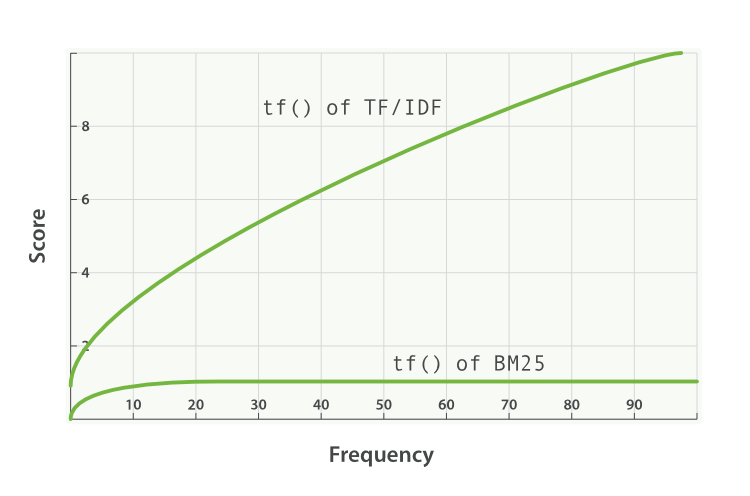
曾经有个时期，将 *最* 普通的词（或 *停用词* ，参见 [停用词](https://www.elastic.co/guide/cn/elasticsearch/guide/current/stopwords.html)）从索引中移除被认为是一种标准实践， TF/IDF 正是在这种背景下诞生的。TF/IDF 没有考虑词频上限的问题，因为高频停用词已经被移除了。

Elasticsearch 的 standard 标准分析器（ string 字段默认使用）不会移除停用词，因为尽管这些词的重要性很低，但也不是毫无用处。这导致：在一个相当长的文档中，像 the 和 and 这样词出现的数量会高得离谱，以致它们的权重被人为放大。

另一方面，BM25 有一个上限，文档里出现 5 到 10 次的词会比那些只出现一两次的对相关度有着显著影响。但是如图 [TF/IDF 与 BM25 的词频饱和度](https://www.elastic.co/guide/cn/elasticsearch/guide/current/pluggable-similarites.html#img-bm25-saturation) 所见，文档中出现 20 次的词几乎与那些出现上千次的词有着相同的影响。

这就是 *非线性词频饱和度（nonlinear term-frequency saturation）* 。

**图 34. TF/IDF 与 BM25 的词频饱和度**



**字段长度归一化（Field-length normalization）**

在 [字段长归一化](https://www.elastic.co/guide/cn/elasticsearch/guide/current/scoring-theory.html#field-norm) 中，我们提到过 Lucene 会认为较短字段比较长字段更重要：字段某个词的频度所带来的重要性会被这个字段长度抵消，但是实际的评分函数会将所有字段以同等方式对待。它认为所有较短的 title 字段比所有较长的 body 字段更重要。

BM25 当然也认为较短字段应该有更多的权重，但是它会分别考虑每个字段内容的平均长度，这样就能区分短 title 字段和 长 title 字段。

小心

在 [查询时权重提升](https://www.elastic.co/guide/cn/elasticsearch/guide/current/query-time-boosting.html) 中，已经说过 title 字段因为其长度比 body 字段 *自然* 有更高的权重提升值。由于字段长度的差异只能应用于单字段，这种自然的权重提升会在使用 BM25 时消失。

**BM25 调优**

不像 TF/IDF ，BM25 有一个比较好的特性就是它提供了两个可调参数：

k1

这个参数控制着词频结果在词频饱和度中的上升速度。默认值为 1.2 。值越小饱和度变化越快，值越大饱和度变化越慢。

b

这个参数控制着字段长归一值所起的作用， 0.0 会禁用归一化， 1.0 会启用完全归一化。默认值为 0.75 。

在实践中，调试 BM25 是另外一回事， k1 和 b 的默认值适用于绝大多数文档集合，但最优值还是会因为文档集不同而有所区别，为了找到文档集合的最优值，就必须对参数进行反复修改验证。

### 更改相似度

相似度算法可以按字段指定， 只需在映射中为不同字段选定即可：

PUT /my\_index

{

"mappings": {

"doc": {

"properties": {

"title": {

"type": "string",

"similarity": "BM25" https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

},

"body": {

"type": "string",

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

}

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/changing-similarities.html#CO122-1) | title 字段使用 BM25 相似度算法。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/changing-similarities.html#CO122-2) | body 字段用默认相似度算法（参见 [实用评分函数](https://www.elastic.co/guide/cn/elasticsearch/guide/current/practical-scoring-function.html)）。 |

目前，Elasticsearch 不支持更改已有字段的相似度算法 similarity 映射，只能通过为数据重新建立索引来达到目的。

**配置 BM25**

配置相似度算法和配置分析器很相似， 自定义相似度算法可以在创建索引时指定，例如：

PUT /my\_index

{

"settings": {

"similarity": {

"my\_bm25": { https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

"type": "BM25",

"b": 0 https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

}

}

},

"mappings": {

"doc": {

"properties": {

"title": {

"type": "string",

"similarity": "my\_bm25" https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png

},

"body": {

"type": "string",

"similarity": "BM25" https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/4.png

}

}

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/changing-similarities.html#CO123-1) | 创建一个基于内置 BM25 ，名为 my\_bm25 的自定义相似度算法。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/changing-similarities.html#CO123-2) | 禁用字段长度规范化（field-length normalization）。参见 [调试 BM25](https://www.elastic.co/guide/cn/elasticsearch/guide/current/pluggable-similarites.html#bm25-tunability) 。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/changing-similarities.html#CO123-3) | title 字段使用自定义相似度算法 my\_bm25 。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/4.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/changing-similarities.html#CO123-4) | 字段 body 使用内置相似度算法 BM25 。 |

提示

自定义的相似度算法可以通过关闭索引，更新索引设置，开启索引这个过程进行更新。这样可以无须重建索引又能试验不同的相似度算法配置。

### 调试相关度是最后 10% 要做的事情

本章介绍了 Lucene 是如何基于 TF/IDF 生成评分的。理解评分过程是非常重要的， 这样就可以根据具体的业务对评分结果进行调试、调节、减弱和定制。

实践中，简单的查询组合就能提供很好的搜索结果，但是为了获得 *具有成效* 的搜索结果，就必须反复推敲修改前面介绍的这些调试方法。

通常，经过对策略字段应用权重提升，或通过对查询语句结构的调整来强调某个句子的重要性这些方法，就足以获得良好的结果。有时，如果 Lucene 基于词的 TF/IDF 模型不再满足评分需求（例如希望基于时间或距离来评分），则需要更具侵略性的调整。

除此之外，相关度的调试就有如兔子洞，一旦跳进去就很难再出来。 *最相关* 这个概念是一个难以触及的模糊目标，通常不同人对文档排序又有着不同的想法，这很容易使人陷入持续反复调整而没有明显进展的怪圈。

我们强烈建议不要陷入这种怪圈，而要监控测量搜索结果。监控用户点击最顶端结果的频次，这可以是前 10 个文档，也可以是第一页的；用户不查看首次搜索的结果而直接执行第二次查询的频次；用户来回点击并查看搜索结果的频次，等等诸如此类的信息。

这些都是用来评价搜索结果与用户之间相关程度的指标。如果查询能返回高相关的文档，用户会选择前五中的一个，得到想要的结果，然后离开。不相关的结果会让用户来回点击并尝试新的搜索条件。

一旦有了这些监控手段，想要调试查询就并不复杂，稍作调整，监控用户的行为改变并做适当反复尝试。本章介绍的一些工具就只是工具而已，要想物尽其用并将搜索结果提高到 *极高的* 水平，唯一途径就是需要具备能评价度量用户行为的强大能力。

# 处理人类语言

|  |  |  |
| --- | --- | --- |
|  | “我认识这句话里的所有单词，但并不能理解全句。” |  |
|  | -- Matt Groening | |

全文搜索是一场 查准率 与 查全率 之间的较量—查准率即尽量返回较少的无关文档，而查全率则尽量返回较多的相关文档。 尽管能够精准匹配用户查询的单词，但这仍然不够，我们会错过很多被用户认为是相关的文档。 因此，我们需要把网撒得更广一些，去搜索那些和原文不是完全匹配但却相关的单词。

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

如果文档 确实 包含用户查询的内容，那么这些文档应当出现在返回结果的最前面，而匹配程度较低的文档将会排在靠后的位置。 如果没有任何完全匹配的文档，我们至少可以给用户展示一些潜在的匹配结果；它们甚至可能就是用户最初想要的结果。

以下列出了一些可优化的地方：

* 清除类似 ´ ， ^ ， ¨ 的变音符号，这样在搜索 rôle 的时候也会匹配 role ，反之亦然。请见 [归一化词元](https://www.elastic.co/guide/cn/elasticsearch/guide/current/token-normalization.html)。
* 通过提取单词的词干，清除单数和复数之间的差异—fox 与 foxes—以及时态上的差异—jumping、 jumped 与 jumps 。请见 [将单词还原为词根](https://www.elastic.co/guide/cn/elasticsearch/guide/current/stemming.html)。
* 清除常用词或者 停用词 ，如 the ， and ， 和 or ，从而提升搜索性能。请见 [停用词: 性能与精度](https://www.elastic.co/guide/cn/elasticsearch/guide/current/stopwords.html)。
* 包含同义词，这样在搜索 quick 时也可以匹配 fast ，或者在搜索 UK 时匹配 United Kingdom 。 请见 [同义词](https://www.elastic.co/guide/cn/elasticsearch/guide/current/synonyms.html)。
* 检查拼写错误和替代拼写方式，或者 同音异型词 —发音一致的不同单词，例如 their 与 there ， meat 、 meet 与 mete 。 请见 [拼写错误](https://www.elastic.co/guide/cn/elasticsearch/guide/current/fuzzy-matching.html)。

在我们可以操控单个单词之前，需要先将文本切分成单词， 这也意味着我们需要知道 单词 是由什么组成的。我们将在 [词汇识别](https://www.elastic.co/guide/cn/elasticsearch/guide/current/identifying-words.html) 章节阐释这个问题。

在这之前，让我们看看如何更快更简单地开始。

## 开始处理各种语言

Elasticsearch 为很多世界流行语言提供良好的、简单的、开箱即用的语言分析器集合：

阿拉伯语、亚美尼亚语、巴斯克语、巴西语、保加利亚语、加泰罗尼亚语、中文、捷克语、丹麦、荷兰语、英语、芬兰语、法语、加里西亚语、德语、希腊语、北印度语、匈牙利语、印度尼西亚、爱尔兰语、意大利语、日语、韩国语、库尔德语、挪威语、波斯语、葡萄牙语、罗马尼亚语、俄语、西班牙语、瑞典语、土耳其语和泰语。

这些分析器承担以下四种角色：

* 文本拆分为单词：

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

* 大写转小写：

The → the

* 移除常用的 \_停用词\_：

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

* 将变型词（例如复数词，过去式）转化为词根：

foxes → fox

为了更好的搜索性，每个语言的分析器提供了该语言词汇的具体转换规则：

* 英语 分析器移除了所有格 's

John's → john

* 法语 分析器移除了 元音省略 例如 l' 和 qu' 和 变音符号 例如 ¨ 或 ^ ：

l'église → eglis

* 德语 分析器规范化了切词， 将切词中的 ä 和 ae 替换为 a ， 或将 ß 替换为 ss ：

äußerst → ausserst

### 使用语言分析器

Elasticsearch 的内置分析器都是全局可用的，不需要提前配置， 它们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" https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

}

}

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/using-language-analyzers.html#CO124-1) | title 字段将会用 `english`（英语）分析器替换默认的 `standard`（标准）分析器 |

当然，文本经过 english 分析处理，我们会丢失源数据：

GET /my\_index/\_analyze?field=title https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

I'm not happy about the foxes

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/using-language-analyzers.html#CO125-1) | 切词为: i'm`，`happi`，`about`，`fox |

我们无法分辨源文档中是包含单数 fox 还是复数 foxes ；单词 not 因为是停用词所以被移除了， 所以我们无法分辨源文档中是happy about foxes还是not happy about foxes，虽然通过使用 english （英语）分析器，使得匹配规则更加宽松，我们也因此提高了召回率，但却降低了精准匹配文档的能力。

为了获得两方面的优势，我们可以使用[multifields](https://www.elastic.co/guide/cn/elasticsearch/guide/current/multi-fields.html)（多字段）对 title 字段建立两次索引： 一次使用`english`（英语）分析器，另一次使用 `standard`（标准）分析器:

PUT /my\_index

{

"mappings": {

"blog": {

"properties": {

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

"type": "string",

"fields": {

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

"type": "string",

"analyzer": "english"

}

}

}

}

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/using-language-analyzers.html#CO126-1) | 主 title 字段使用 `standard`（标准）分析器。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/using-language-analyzers.html#CO126-2) | title.english 子字段使用 `english`（英语）分析器。 |

替换为该字段映射后，我们可以索引一些测试文档来展示怎么在搜索时使用两个字段：

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

"query": "not happy foxes",

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

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/using-language-analyzers.html#CO127-1) | 使用[most\_fields](https://www.elastic.co/guide/cn/elasticsearch/guide/current/most-fields.html) query type（多字段搜索语法来）让我们可以用多个字段来匹配同一段文本。 |

感谢 title.english 字段的切词，无论我们的文档中是否含有单词 foxes 都会被搜索到，第二份文档的相关性排行要比第一份高， 因为在 title 字段中匹配到了单词 not 。

### 配置语言分析器

语言分析器都不需要任何配置，开箱即用， 它们中的大多数都允许你控制它们的各方面行为，具体来说：

词干提取排除

想象下某个场景，用户们想要搜索 World Health Organization 的结果， 但是却被替换为搜索 organ health 的结果。有这个困惑是因为 organ 和 organization 有相同的词根： organ 。 通常这不是什么问题，但是在一些特殊的文档中就会导致有歧义的结果，所以我们希望防止单词 organization 和 organizations 被缩减为词干。

自定义停用词

英语中默认的停用词列表如下：

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

关于单词 no 和 not 有点特别，这俩词会反转跟在它们后面的词汇的含义。或许我们应该认为这两个词很重要，不应该把他们看成停用词。

为了自定义 english （英语）分词器的行为，我们需要基于 english （英语）分析器创建一个自定义分析器，然后添加一些配置：

PUT /my\_index

{

"settings": {

"analysis": {

"analyzer": {

"my\_english": {

"type": "english",

"stem\_exclusion": [ "organization", "organizations" ], https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

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

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

The World Health Organization does not sell organs.

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/configuring-language-analyzers.html#CO128-1) | 防止 organization 和 organizations 被缩减为词干 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/configuring-language-analyzers.html#CO128-2) | 指定一个自定义停用词列表 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/configuring-language-analyzers.html#CO128-3) | 切词为 world 、 health 、 organization 、 does 、 not 、 sell 、 organ |

我们在 [将单词还原为词根](https://www.elastic.co/guide/cn/elasticsearch/guide/current/stemming.html) 和 [停用词: 性能与精度](https://www.elastic.co/guide/cn/elasticsearch/guide/current/stopwords.html) 中分别详细讨论了词干提取和停用词。

### 混合语言的陷阱

如果你只需要处理一种语言，那么你很幸运。找到一个正确的策略用于处理多语言文档是一项巨大的挑战。

**在索引的时候**

多语言文档主要有以下三个类型：

* 一种是每份 document （文档）有自己的主语言，并包含一些其他语言的片段（参考 [每份文档一种语言](https://www.elastic.co/guide/cn/elasticsearch/guide/current/one-lang-docs.html)。）
* 一种是每个 field （域）有自己的主语言, 并包含一些其他语言的片段（参考 [每个域一种语言](https://www.elastic.co/guide/cn/elasticsearch/guide/current/one-lang-fields.html)。）
* 一种是每个 field （域）都是混合语言（参考 [混合语言域](https://www.elastic.co/guide/cn/elasticsearch/guide/current/mixed-lang-fields.html)。）

（分词）目标不总是可以实现，我们应当保持将不同语言分隔开。在同一份倒排索引内混合多种语言可能造成一些问题。

**不合理的词干提取**

德语的词干提取规则跟英语，法语，瑞典语等是不一样的。 为不同的语言提供同样的词干提规则 将会导致有的词的词根找的正确，有的词的词根找的不正确，有的词根本找不到词根。 甚至是将不同语言的不同含义的词切为同一个词根，合并这些词根的搜索结果会给用户带来困恼。

提供多种的词干提取器轮流切分同一份文档的结果很有可能得到一堆垃圾，因为下一个词干提取器会尝试切分一个已经被缩减为词干的单词，这加剧了上面提到的问题。

**每种书写方式一种词干提取器**

只有一种情况, only-one-stemmer （唯一词干提取器）会发生，就是每种语言都有自己的书写方式。例如，在以色列就有很大的可能一个文档包含希伯来语， 阿拉伯语，俄语（古代斯拉夫语），和英语。

אזהרה - Предупреждение - تحذير - Warning

每种语言使用不同的书写方式，所以一种语言的词干提取器就不会干扰其他语言的，允许为同一份文本提供多种词干提取器。

**不正确的倒排文档频率**

在 [什么是相关性?](https://www.elastic.co/guide/cn/elasticsearch/guide/current/relevance-intro.html) （相关性教程）中，一个 term （词）在一份文档中出现的频率约高，该term（词）的权重就越低。 为了精确的计算相关性，你需要精确的统计 term-frequency （词频）。

一段德文出现在英语为主的文本中会给与德语单词更高的权重，给那么高权重是因为德语单词相对来说更稀有。 但是如果这份文档跟以德语为主的文档混合在一起，那么这段德文就会有很低的权重。

**在搜索的时候**

然而仅仅考虑你的文档是不够的 。你也需要考虑你的用户会怎么搜索这些文档。 通常你能从用户选择的语言界面来确定用户的主语言，（例如， mysite.de 和 mysite.fr ） 或者从用户的浏览器的HTTP header（HTTP头文件） [accept-language](http://www.w3.org/International/questions/qa-lang-priorities.en.php) 确定。

用户的搜索也注意有三个方面:

* 用户使用他的主语言搜索。
* 用户使用其他的语言搜索，但希望获取主语言的搜索结果。
* 用户使用其他语言搜索，并希望获取该语言的搜索结果。（例如，精通双语的人，或者网络咖啡馆的外国访问者）。

根据你搜索数据的类型，或许会返回单语言的合适结果（例如，一个用户在西班牙网站搜索商品），也可能是用户主语言的搜索结果和其他语言的搜索结果混合。

通常来说，给与用户语言偏好的搜索很有意义。一个使用英语的用户搜索时更希望看到英语 Wikipedia 页面而不是法语 Wikipedia 页面。

**语言识别**

你很可能已经知道你的文档所选用的语言，或者你的文档只是在你自己的组织内编写并被翻译成确定的一系列语言。人类的预识别可能是最可靠的将语言正确归类的方法。

然而，或许你的文档来自第三方资源且没经过语言归类，或者是不正确的归类。这种情况下，你需要一个学习算法来归类你文档的主语言。幸运的是，一些语言有现成的工具包可以帮你解决这个问题。

详细内容是来自 [Mike McCandless](http://blog.mikemccandless.com/2013/08/a-new-version-of-compact-language.html) 的 [chromium-compact-language-detector](https://github.com/mikemccand/chromium-compact-language-detector) 工具包，使用的是google开发的基于 ([Apache License 2.0](http://www.apache.org/licenses/LICENSE-2.0))的开源工具包 [Compact Language Detector](https://code.google.com/p/cld2/) (CLD) 。 它小巧，快速，且精确，并能根据短短的两句话就可以检测 160+ 的语言。 它甚至能对单块文本检测多种语言。支持多种开发语言包括 Python，Perl，JavaScript，PHP，C#/.NET，和 R 。

确定用户搜索请求的语言并不是那么简单。 CLD 是为了至少 200 字符长的文本设计的。字符短的文本，例如搜索关键字，会产生不精确的结果。 这种情况下，或许采取一些简单的启发式算法会更好些，例如该国家的官方语言，用户选择的语言，和 HTTP accept-language headers （HTTP头文件）。

### 每份文档一种语言

每个主语言文档 只需要相当简单的设置。 不同语言的文档被分别存放在不同的索引中 — blogs-en 、blogs-fr ， 如此等等 — 这样每个索引就可以使用相同的类型和相同的域，只是使用不同的分析器：

PUT /blogs-en

{

"mappings": {

"post": {

"properties": {

"title": {

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

"fields": {

"stemmed": {

"type": "string",

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

}

}}}}}}

PUT /blogs-fr

{

"mappings": {

"post": {

"properties": {

"title": {

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

"fields": {

"stemmed": {

"type": "string",

"analyzer": "french" https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/4.png

}

}}}}}}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/one-lang-docs.html#CO129-1) [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/one-lang-docs.html#CO129-3) | 索引 blogs-en 和 blogs-fr 的 post 类型都有一个包含 title 域。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/one-lang-docs.html#CO129-2) [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/4.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/one-lang-docs.html#CO129-4) | title.stemmed 子域使用了具体语言的分析器。 |

这个方法干净且灵活。新语言很容易被添加 — 仅仅是创建一个新索引--因为每种语言都是彻底的被分开， 我们不用遭受在 [混合语言的陷阱](https://www.elastic.co/guide/cn/elasticsearch/guide/current/language-pitfalls.html) 中描述的词频和词干提取的问题。

每一种语言的文档都可被独立查询，或者通过查询多种索引来查询多种语言。 我们甚至可以使用 indices\_boost 参数为特定的语言添加优先权 ：

GET /blogs-\*/post/\_search https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

{

"query": {

"multi\_match": {

"query": "deja vu",

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

"type": "most\_fields"

}

},

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

"blogs-en": 3,

"blogs-fr": 2

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/one-lang-docs.html#CO130-1) | 这个查询会在所有以 blogs- 开头的索引中执行。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/one-lang-docs.html#CO130-2) | title.stemmed 字段使用每个索引中指定的分析器查询。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/one-lang-docs.html#CO130-3) | 也许用户接受语言标头表明，更倾向于英语，然后是法语，所以相应的，我们会为每个索引的结果添加权重。任何其他语言会有一个中性的权重 1 。 |

**外语单词**

当然，有些文档含有一些其他语言的单词或句子，且不幸的是这些单词被切为了正确的词根。对于主语言文档，这通常并不是主要的问题。用户经常需要搜索很精确的单词--例如，一个其他语言的引用--而不是语型变化过的单词。召回率 (Recall)可以通过使用 [归一化词元](https://www.elastic.co/guide/cn/elasticsearch/guide/current/token-normalization.html) 中讲解的技术提升。

假设有些单词例如地名应当能被主语言和原始语言都能检索，例如 Munich 和 München 。 这些单词实际上是我们在 [同义词](https://www.elastic.co/guide/cn/elasticsearch/guide/current/synonyms.html) 解释过的同义词。

**不要对语言使用类型**

你也许很倾向于为每个语言使用分开的类型， 来代替使用分开的索引。 为了达到最佳效果，你应当避免使用类型。在 [类型和映射](https://www.elastic.co/guide/cn/elasticsearch/guide/current/mapping.html) 解释过，不同类型但有相同域名的域会被索引在 相同的倒排索引 中。这意味着不同类型（和不同语言）的词频混合在了一起。

为了确保一种语言的词频不会污染其他语言的词频，在后面的章节中会介绍到，无论是为每个语言使用单独的索引，还是使用单独的域都可以。

### 每个域一种语言

对于一些实体类，例如:产品、电影、法律声明， 通常这样的一份文本会被翻译成不同语言的文档。虽然这些不同语言的文档可以单独保存在各自的索引中。但另一种更合理的方式是同一份文本的所有翻译统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",

...

}

每份翻译存储在不同的域中，根据域的语言决定使用相应的分析器：

PUT /movies

{

"mappings": {

"movie": {

"properties": {

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

"type": "string"

},

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

"type": "string",

"analyzer": "brazilian"

},

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

"type": "string",

"analyzer": "czech"

},

"title\_en": { https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/4.png

"type": "string",

"analyzer": "english"

},

"title\_es": { https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/5.png

"type": "string",

"analyzer": "spanish"

}

}

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/one-lang-fields.html#CO131-1) | title 域含有title的原文，并使用 standard （标准）分析器。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/one-lang-fields.html#CO131-2) [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/one-lang-fields.html#CO131-3)[https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/4.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/one-lang-fields.html#CO131-4) [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/5.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/one-lang-fields.html#CO131-5) | 其他字段使用适合自己语言的分析器。 |

在维持干净的词频方面，虽然 index-per-language （一种语言一份索引的方法），不像 field-per-language （一种语言一个域的方法）分开索引那么灵活。但是使用 [update-mapping API](https://www.elastic.co/guide/cn/elasticsearch/guide/current/mapping-intro.html#updating-a-mapping) 添加一个新域也很简单，那些新域需要新的自定义分析器，这些新分析器只能在索引创建时被装配。有一个变通的方案，你可以先关闭这个索引 [close](https://www.elastic.co/guide/en/elasticsearch/reference/master/indices-open-close.html) ，然后使用 [update-settings API](https://www.elastic.co/guide/en/elasticsearch/reference/master/indices-update-settings.html) ，重新打开这个索引，但是关掉这个索引意味着得停止服务一段时间。

文档的 一种语言可以单独查询，也可以通过查询多个域来查询多种语言。我们甚至可以通过对特定语言设置偏好来提高字段优先级：

GET /movies/movie/\_search

{

"query": {

"multi\_match": {

"query": "club de la lucha",

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

"type": "most\_fields"

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/one-lang-fields.html#CO132-1) | 这个搜索查询所有以 title 为前缀的域，但是对 title\_es 域加权重 2 。其他的所有域是中性权重 1 。 |

### 混合语言域

通常,那些从源数据中获得的多种语言混合在一个域中的文档会超出你的控制， 例如 从网上爬取的页面：

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

正确的处理多语言类型文档是非常困难的。即使你简单对所有的域使用 standard （标准）分析器， 但你的文档会变得不利于搜索，除非你使用了合适的词干提取器。当然，你不可能只选择一个词干提取器。 词干提取器是由语言具体决定的。或者，词干提取器是由语言和脚本所具体决定的。像在 [每种书写方式一种词干提取器](https://www.elastic.co/guide/cn/elasticsearch/guide/current/language-pitfalls.html#different-scripts) 讨论中那样。 如果每个语言都使用不同的脚本，那么词干提取器就可以合并了。

假设你的混合语言使用的是一样的脚本，例如拉丁文，你有三个可用的选择：

* 切分到不同的域
* 进行多次分析
* 使用 n-grams

**切分到不同的域**

在 [语言识别](https://www.elastic.co/guide/cn/elasticsearch/guide/current/language-pitfalls.html#identifying-language) 提到过的紧凑的语言检测 可以告诉你哪部分文档属于哪种语言。 你可以用 [每个域一种语言](https://www.elastic.co/guide/cn/elasticsearch/guide/current/one-lang-fields.html) 中用过的一样的方法来根据语言切分文本。

**进行多次分析**

如果你主要处理数量有限的语言， 你可以使用多个域，每种语言都分析文本一次。

PUT /movies

{

"mappings": {

"title": {

"properties": {

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

"type": "string",

"fields": {

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

"type": "string",

"analyzer": "german"

},

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

"type": "string",

"analyzer": "english"

},

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

"type": "string",

"analyzer": "french"

},

"es": { https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/5.png

"type": "string",

"analyzer": "spanish"

}

}

}

}

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/mixed-lang-fields.html#CO133-1) | 主域 title 使用 standard （标准）分析器 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/mixed-lang-fields.html#CO133-2) [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/mixed-lang-fields.html#CO133-3)[https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/4.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/mixed-lang-fields.html#CO133-4) [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/5.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/mixed-lang-fields.html#CO133-5) | 每个子域提供不同的语言分析器来对 title 域文本进行分析。 |

**使用 n-grams**

你可以使用 [Ngrams 在复合词的应用](https://www.elastic.co/guide/cn/elasticsearch/guide/current/ngrams-compound-words.html) 中描述的 方法索引所有的词汇为 n-grams。 大多数语型变化包含给单词添加一个后缀（或在一些语言中添加前缀），所以通过将单词拆成 n-grams，你有很大的机会匹配到相似但不完全一样的单词。 这个可以结合 analyze-multiple times （多次分析）方法为不支持的语言提供全域抓取：

PUT /movies

{

"settings": {

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

},

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

"type": "string",

"analyzer": "trigrams"

}

}

}

}

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/mixed-lang-fields.html#CO134-1) | 在 analysis 章节, 我们按照 [Ngrams 在复合词的应用](https://www.elastic.co/guide/cn/elasticsearch/guide/current/ngrams-compound-words.html) 中描述的定义了同样的 trigrams 分析器。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/mixed-lang-fields.html#CO134-2) | 在 title.general 域使用 trigrams 分析器索引所有的语言。 |

当查询抓取所有 general 域时，你可以使用 minimum\_should\_match （最少应当匹配数）来减少低质量的匹配。 或许也需要对其他字段进行稍微的加权，给与主语言域的权重要高于其他的在 general 上的域：

GET /movies/movie/\_search

{

"query": {

"multi\_match": {

"query": "club de la lucha",

"fields": [ "title\*^1.5", "title.general" ], https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

"type": "most\_fields",

"minimum\_should\_match": "75%" https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/mixed-lang-fields.html#CO135-1) | 所有 title 或 title.\* 域给与了比 title.general 域稍微高的加权。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/mixed-lang-fields.html#CO135-2) | minimum\_should\_match`（最少应当匹配数） 参数减少了低质量匹配的返回数, 这对 `title.general 域尤其重要。 |

## 词汇识别

英语单词相对而言比较容易辨认：单词之间都是以空格或者（一些）标点隔开。 然而即使在英语词汇中也会有一些争议： you’re 是一个单词还是两个？ o’clock ， cooperate ， half-baked ，或者 eyewitness 这些呢？

德语或者荷兰语把独立的单词合并起来创造一个长的合成词如 Weißkopfseeadler (white-headed sea eagle) , 但是为了在查询 Adler (eagle)的时候返回查询 Weißkopfseeadler 的结果，我们需要懂得怎么将合并词拆成词组。

亚洲的语言更复杂：很多语言在单词，句子，甚至段落之间没有空格。 有些词可以用一个字来表达，但是同样的字在另一个字旁边的时候就是不同意思的长词的一部分。

显而易见的是没有能够奇迹般处理所有人类语言的万能分析器，Elasticsearch 为很多语言提供了专用的分析器， 其他特殊语言的分析器以插件的形式提供。

然而并不是所有语言都有专用分析器，而且有时候你甚至无法确定处理的是什么语言。这种情况，我们需要一些忽略语言也能合理工作的标准工具包。

### 标准分析器

任何全文检索的字符串域都默认使用 standard 分析器。 如果我们想要一个 [自定义 分析器](https://www.elastic.co/guide/cn/elasticsearch/guide/current/custom-analyzers.html) ，可以按照如下定义方式重新实现 标准 分析器：

{

"type": "custom",

"tokenizer": "standard",

"filter": [ "lowercase", "stop" ]

}

在 [归一化词元](https://www.elastic.co/guide/cn/elasticsearch/guide/current/token-normalization.html) （标准化词汇单元）和 [停用词: 性能与精度](https://www.elastic.co/guide/cn/elasticsearch/guide/current/stopwords.html) （停用词）中，我们讨论了 lowercase （小写字母）和 stop （停用词） 词汇单元过滤器 ，但是现在，我们专注于 standard tokenizer （标准分词器）。

### 标准分词器

分词器 接受一个字符串作为输入，将 这个字符串拆分成独立的词或 语汇单元（token） （可能会丢弃一些标点符号等字符），然后输出一个 语汇单元流（token stream） 。

有趣的是用于词汇 识别 的算法。 whitespace （空白字符）分词器按空白字符 —— 空格、tabs、换行符等等进行简单拆分 —— 然后假定连续的非空格字符组成了一个语汇单元。例如：

GET /\_analyze?tokenizer=whitespace

You're the 1st runner home!

这个请求会返回如下词项（terms）： You're 、 the 、 1st 、 runner 、 home!

letter 分词器 ，采用另外一种策略，按照任何非字符进行拆分， 这样将会返回如下单词： You 、 re 、 the 、 st 、 runner 、 home 。

standard 分词器使用 Unicode 文本分割算法 （定义来源于 [Unicode Standard Annex #29](http://unicode.org/reports/tr29/)）来寻找单词 之间 的界限，并且输出所有界限之间的内容。 Unicode 内含的知识使其可以成功的对包含混合语言的文本进行分词。

标点符号 可能是单词的一部分，也可能不是，这取决于它出现的位置：

GET /\_analyze?tokenizer=standard

You're my 'favorite'.

在这个例子中，You're 中的撇号被视为单词的一部分，然而 'favorite' 中的单引号则不会被视为单词的一部分， 所以分词结果如下： You're 、 my 、 favorite 。

提示

uax\_url\_email 分词器和 standard 分词器工作方式极其相同。 区别只在于它能识别 email 地址和 URLs 并输出为单个语汇单元。 standard 分词器则不一样，会将 email 地址和 URLs 拆分成独立的单词。 例如，email 地址 joe-bloggs@foo-bar.com 的分词结果为 joe 、 bloggs 、 foo 、 bar.com 。

standard 分词器是大多数语言分词的一个合理的起点，特别是西方语言。 事实上，它构成了大多数特定语言分析器的基础，如 english 、french 和 spanish 分析器。 它也支持亚洲语言，只是有些缺陷，你可以考虑通过 ICU 插件的方式使用 icu\_tokenizer 进行替换。

### 安装 ICU 插件

Elasticsearch的 [ICU 分析器插件](https://github.com/elasticsearch/elasticsearch-analysis-icu) 使用 国际化组件 Unicode (ICU) 函数库（详情查看 [site.project.org](http://site.icu-project.org/) ）提供丰富的处理 Unicode 工具。 这些包含对处理亚洲语言特别有用的 icu\_分词器 ，还有大量对除英语外其他语言进行正确匹配和排序所必须的分词过滤器。

注意

ICU 插件是处理英语之外语言的必需工具，非常推荐你安装并使用它，不幸的是，因为是基于额外的 ICU 函数库， 不同版本的ICU插件可能并不兼容之前的版本，当更新插件的时候，你需要重新索引你的数据。

安装这个插件，第一步先关掉你的Elasticsearch节点，然后在Elasticsearch的主目录运行以下命令：

./bin/plugin -install elasticsearch/elasticsearch-analysis-icu/$VERSION https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/icu-plugin.html#CO136-1) | 当前 $VERSION （版本）可以在以下地址找到 https://github.com/elasticsearch/elasticsearch-analysis-icu. |

一旦安装后，重启Elasticsearch，你将会看到类似如下的一条启动日志：

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

如果你有很多节点并以集群方式运行的，你需要在集群的每个节点都安装这个插件。

### icu\_分词器

icu\_分词器 和 标准分词器 使用同样的 Unicode 文本分段算法， 只是为了更好的支持亚洲语，添加了泰语、老挝语、中文、日文、和韩文基于词典的词汇识别方法，并且可以使用自定义规则将缅甸语和柬埔寨语文本拆分成音节。

例如，分别比较 标准分词器 和 icu\_分词器 在分词泰语中的 'Hello. I am from Bangkok.' 产生的词汇单元：

GET /\_analyze?tokenizer=standard

สวัสดี ผมมาจากกรุงเทพฯ

标准分词器 产生了两个词汇单元，每个句子一个： สวัสดี ， ผมมาจากกรุงเทพฯ 。这个只是你想搜索整个句子 'I am from Bangkok.' 的时候有用，但是如果你仅想搜索 'Bangkok.' 则不行。

GET /\_analyze?tokenizer=icu\_tokenizer

สวัสดี ผมมาจากกรุงเทพฯ

相反， icu\_分词器 可以把文本分成独立的单词（ สวัสดี ， ผม ， มา ， จาก ， กรุงเทพฯ ），这使得文档更容易被搜索到。

相较而言, 标准分词器 分词中文和日文的时候“过度分词”了，经常将一个完整的词拆分为独立的字符，因为单词之间并没有空格，很难区分连续的字符是间隔的单词还是一个句子中的单字：

* 向的意思是 facing （面对）， 日的意思是 sun （太阳），葵的意思是 hollyhock （蜀葵）。当写在一起的时候, 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 （月份），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 （连续不断的下雨,梅雨）。当在合并第四个字符， 式， 意思是 style （样式），五月雨式这个单词则成了一种不屈不挠持续不断的东西的形容词。

虽然每个字符本身可以是一个单词，但使词汇单元保持更大的原始概念比使其仅作为一个词组的一部分要有意义的多：

GET /\_analyze?tokenizer=standard

向日葵

GET /\_analyze?tokenizer=icu\_tokenizer

向日葵

标准分词器 在前面的例子中将每个字符输出为单独的词汇单元： 向 ， 日 ， 葵 。 icu\_分词器 会输出单个词汇单元 向日葵 （sunflower） 。

标准分词器 和 icu\_分词器 的另一个不同的地方是后者会将不同书写方式的字符（例如，βeta ）拆分成独立的词汇单元 — β 和 eta— ，而前者则会输出单个词汇单元： βeta 。

### 整理输入文本

当输入文本是干净的时候分词器提供最佳分词结果，有效文本，这里 有效 指的是遵从 Unicode 算法期望的标点符号规则 。 然而很多时候，我们需要处理的文本会是除了干净文本之外的任何文本。在分词之前整理文本会提升输出结果的质量。

**HTML 分词**

将 HTML 通过 标准分词器 或 icu\_分词器 分词将产生糟糕的结果。这些分词器不知道如何处理 HTML 标签。例如：

GET /\_analyze?tokenizer=standard

<p>Some d&eacute;j&agrave; vu <a href="http://somedomain.com>">website</a>

标准分词器 会混淆 HTML 标签和实体，并且输出以下词汇单元： p 、 Some 、 d 、 eacute 、 j 、 agrave 、 vu 、 a 、 href 、 http 、 somedomain.com 、 website 、 a 。这些词汇单元显然不知所云！

字符过滤器 可以添加进分析器中，在将文本传给分词器之前预处理该文本。在这种情况下，我们可以用 html\_strip 字符过滤器 移除 HTML 标签并编码 HTML 实体如 &eacute; 为一致的 Unicode 字符。

字符过滤器可以通过 analyze API 进行测试，这需要在查询字符串中指明它们：

GET /\_analyze?tokenizer=standard&char\_filters=html\_strip

<p>Some d&eacute;j&agrave; vu <a href="http://somedomain.com>">website</a>

想将它们作为分析器的一部分使用，需要把它们添加到 custom 类型的自定义分析器里：

PUT /my\_index

{

"settings": {

"analysis": {

"analyzer": {

"my\_html\_analyzer": {

"tokenizer": "standard",

"char\_filter": [ "html\_strip" ]

}

}

}

}

}

一旦自定义分析器创建好之后， 我们新的 my\_html\_analyzer 就可以用 analyze API 测试：

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

<p>Some d&eacute;j&agrave; vu <a href="http://somedomain.com>">website</a>

这次输出的词汇单元才是我们期望的： Some ， déjà ， vu ， website 。

**整理标点符号**

标准分词器 和 icu\_分词器 都能理解单词中的撇号应当被视为单词的一部分，然而包围单词的单引号在不应该。 分词文本 You're my 'favorite' ， 会被输出正确的词汇单元 You're ， my ， favorite 。

不幸的是， Unicode 列出了一些有时会被用为撇号的字符：

U+0027

撇号标记为 (')— 原始 ASCII 符号

U+2018

左单引号标记为 (‘)— 当单引用时作为一个引用的开始

U+2019

右单引号标记为 (’)— 当单引用时座位一个引用的结束，也是撇号的首选字符。

当这三个字符出现在单词中间的时候， 标准分词器 和 icu\_分词器 都会将这三个字符视为撇号（这会被视为单词的一部分）。 然而还有另外三个长得很像撇号的字符：

U+201B

Single high-reversed-9 （高反单引号）标记为 (‛)— 跟 U+2018 一样，但是外观上有区别

U+0091

ISO-8859-1 中的左单引号 — 不会被用于 Unicode 中

U+0092

ISO-8859-1 中的右单引号 — 不会被用于 Unicode 中

标准分词器 和 icu\_分词器 把这三个字符视为单词的分界线 — 一个将文本拆分为词汇单元的位置。不幸的是，一些出版社用 U+201B 作为名字的典型书写方式例如 M‛coy ， 第二个俩字符或许可以被你的文字处理软件打出来，这取决于这款软件的年纪。

即使在使用可以“接受”的引号标记时，一个用单引号书写的词 — You’re — 也和一个用撇号书写的词 — You're — 不一样，这意味着搜索其中的一个变体将会找不到另一个。

幸运的是，可以用 mapping 对这些混乱的字符进行分类， 该过滤器可以运行我们用另一个字符替换所有实例中的一个字符。这种情况下，我们可以简单的用 U+0027 替换所有的撇号变体：

PUT /my\_index

{

"settings": {

"analysis": {

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

"quotes": {

"type": "mapping",

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

"\\u0091=>\\u0027",

"\\u0092=>\\u0027",

"\\u2018=>\\u0027",

"\\u2019=>\\u0027",

"\\u201B=>\\u0027"

]

}

},

"analyzer": {

"quotes\_analyzer": {

"tokenizer": "standard",

"char\_filter": [ "quotes" ] https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png

}

}

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/char-filters.html#CO137-1) | 我们自定义了一个 char\_filter （字符过滤器）叫做 quotes ，提供所有撇号变体到简单撇号的映射。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/char-filters.html#CO137-2) | 为了更清晰，我们使用每个字符的 JSON Unicode 转义语句，当然我们也可以使用他们本身字符表示： "‘=>'" 。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/char-filters.html#CO137-3) | 我们用自定义的 quotes 字符过滤器创建一个新的分析器叫做 quotes\_analyzer 。 |

像以前一样，我们需要在创建了分析器后测试它：

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

You’re my ‘favorite’ M‛Coy

这个例子返回如下词汇单元，其中所有的单词中的引号标记都被替换为了撇号： You're, my, favorite, M'Coy 。

投入更多的努力确保你的分词器接收到高质量的输入，你的搜索结果质量也将会更好。

## 归一化词元

把文本切割成词元(token)只是这项工作 的一半。为了让这些词元(token)更容易搜索, 这些词元(token)需要被 *归一化*(*normalization*)--这个过程会去除同一个词元(token)的无意义差别，例如大写和小写的差别。可能我们还需要去掉有意义的差别, 让 esta`、`ésta 和 está 都能用同一个词元(token)来搜索。你会用 déjà vu 来搜索，还是 deja vu?

这些都是语汇单元过滤器的工作。语汇单元过滤器接收来自分词器(tokenizer)的词元(token)流。还可以一起使用多个语汇单元过滤器，每一个都有自己特定的处理工作。每一个语汇单元过滤器都可以处理来自另一个语汇单元过滤器输出的单词流。

### 举个例子

用的最多的语汇单元过滤器(token filters)是 lowercase 过滤器，它的功能正和你期望的一样；它将每个词元(token)转换为小写形式：

GET /\_analyze?tokenizer=standard&filters=lowercase

The QUICK Brown FOX! https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/lowercase-token-filter.html#CO138-1) | 得到的词元(token)是 the, quick, brown, fox |

只要查询和检索的分析过程是一样的，不管用户搜索 fox 还是 FOX 都能得到一样的搜索结果。lowercase过滤器会将查询 FOX 的请求转换为查询 fox 的请求， fox 和我们在倒排索引中存储的是同一个词元(token)。

为了在分析过程中使用 token 过滤器 ，我们可以创建一个 custom 分析器 ：

PUT /my\_index

{

"settings": {

"analysis": {

"analyzer": {

"my\_lowercaser": {

"tokenizer": "standard",

"filter": [ "lowercase" ]

}

}

}

}

}

我们可以通过 analyze API 来验证:

GET /my\_index/\_analyze?analyzer=my\_lowercaser

The QUICK Brown FOX! https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/lowercase-token-filter.html#CO139-1) | 得到的词元是 the, quick, brown, fox |

### 如果有口音

英语用变音符号(例如 ´, ^, 和 ¨) 来强调单词—​例如 rôle, déjà, 和 däis —​但是是否使用他们通常是可选的. 其他语言则通过变音符号来区分单词。当然，只是因为在你的索引中拼写正确的单词并不意味着用户将搜索正确的拼写。 去掉变音符号通常是有用的，让 rôle 对应 role, 或者反过来。 对于西方语言，可以用 asciifolding 字符过滤器来实现这个功能。 实际上，它不仅仅能去掉变音符号。它会把Unicode字符转化为ASCII来表示:

* ß ⇒ ss
* æ ⇒ ae
* ł ⇒ l
* ɰ ⇒ m
* ⁇ ⇒ ??
* ❷ ⇒ 2
* ⁶ ⇒ 6

像 lowercase 过滤器一样, 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 https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/asciifolding-token-filter.html#CO140-1) | 得到的词元 my, oesophagus, caused, a, debacle |

**保留原意**

理所当然的，去掉变音符号会丢失原意。 例如, 参考 这三个 西班牙单词:

esta

形容词 this 的阴性形式, 例如 esta silla (this chair) 和 esta (this one).

ésta

esta 的古代用法.

está

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

通常我们会合并前两个形式的单词，而去区分和他们不相同的第三个形式的单词。类似的:

sé

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

se

与许多动词使用的第三人称反身代词, 例如 se sabe (it is known).

不幸的是，没有简单的方法，去区分哪些词应该保留变音符号和哪些词应该去掉变音符号。而且很有可能，你的用户也不知道.

相反， 我们对文本做两次索引: 一次用原文形式，一次用去掉变音符号的形式 :

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

{

"properties": {

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

"type": "string",

"analyzer": "standard",

"fields": {

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

"type": "string",

"analyzer": "folding"

}

}

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/asciifolding-token-filter.html#CO141-1) | 在 title 字段用 standard 分析器，会保留原文的变音符号. |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/asciifolding-token-filter.html#CO141-2) | 在 title.folded 字段用 folding 分析器，会去掉变音符号 |

你可以使用 analyze API 分析 Esta está loca (This woman is crazy)这个句子，来验证字段映射:

GET /my\_index/\_analyze?field=title https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

Esta está loca

GET /my\_index/\_analyze?field=title.folded https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

Esta está loca

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/asciifolding-token-filter.html#CO142-1) | 得到的词元 esta, está, loca |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/asciifolding-token-filter.html#CO142-2) | 得到的词元 esta, esta, loca |

可以用更多的文档来测试:

PUT /my\_index/my\_type/1

{ "title": "Esta loca!" }

PUT /my\_index/my\_type/2

{ "title": "Está loca!" }

现在，我们可以通过联合所有的字段来搜索。在multi\_match`查询中通过 <<most-fields,`most\_fieldsmode>> 模式来联合所有字段的结果:

GET /my\_index/\_search

{

"query": {

"multi\_match": {

"type": "most\_fields",

"query": "esta loca",

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

}

}

}

通过 validate-query API 来执行这个查询可以帮助你理解查询是如何执行的:

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

{

"query": {

"multi\_match": {

"type": "most\_fields",

"query": "está loca",

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

}

}

}

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

(title:está title:loca )

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

无论用户搜索的是 esta 还是 está; 两个文档都会被匹配，因为去掉变音符号形式的单词在 title.folded字段中。然而，只有原文形式的单词在 title 字段中。此额外匹配会把包含原文形式单词的文档排在结果列表前面。

我们用 title.folded 字段来 扩大我们的网 (widen the net)来匹配更多的文档，然后用原文形式的 title字段来把关联度最高的文档排在最前面。在可以为了匹配数量牺牲文本原意的情况下，这个技术可以被用在任何分析器里。

提示

asciifolding 过滤器有一个叫做 preserve\_original 的选项可以让你这样来做索引 ，把词的原文词元(original token)和处理--折叠后的词元(folded token)放在同一个字段的同一个位置。开启了这个选项，结果会像这样:

Position 1 Position 2

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

(ésta,esta) loca

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

虽然这个是节约空间的好办法，但是也意味着没有办法再说“给我精确匹配的原文词元”(Give me an exact match on the original word)。包含去掉和不去掉变音符号的词元，会导致不可靠的相关性评分。

所以，正如我们这一章做的，把每个字段的不同形式分开到不同的字段会让索引更清晰。

### Unicode的世界

当Elasticsearch在比较词元(token)的时候，它是进行字节(byte)级别的比较。 换句话说，如果两个词元(token)被判定为相同的话，他们必须是相同的字节(byte)组成的。然而，Unicode允许你用不同的字节来写相同的字符。

例如， é 和 é 的不同是什么？这取决于你问谁。对于Elasticsearch，第一个是由 0xC3 0xA9 这两个字节组成的，第二个是由 0x65 0xCC 0x81 这三个字节组成的。

对于Unicode，他们的差异和他们的怎么组成没有关系，所以他们是相同的。第一个是单个单词 é ，第二个是一个简单 e 和重音符 +´+。

如果你的数据有多个来源，就会有可能发生这种状况：因为相同的单词使用了不同的编码，导致一个形式的 déjà 不能和它的其他形式进行匹配。

幸运的是，这里就有解决办法。这里有4种Unicode 归一化形式 (normalization forms) : nfc, nfd, nfkc, `nfkd`，它们都把Unicode字符转换成对应标准格式，把所有的字符 进行字节(byte)级别的比较。

**Unicode归一化形式 (Normalization Forms)**

\_组合\_ (\_composed\_) 模式—`nfc` 和 `nfkc`—用尽可能少的字节(byte)来代表字符。 ((("composed forms (Unicode normalization)"))) 所以用 `é` 来代表单个字母 `é` 。 \_分解\_ （\_decomposed\_） 模式—`nfd` and `nfkd`—用字符的每一部分来代表字符。所以 `é` 分解为 `e` 和 `´`。 ((("decomposed forms (Unicode normalization)")))

规范 (canonical) 模式—nfc 和 nfd&—把连字作为单个字符，例如 ﬃ 或者 œ 。 兼容(compatibility) 模式—nfkc 和 nfkd`—将这些组合的字符分解成简单字符的等价物，例如： `f + f + i 或者 o + e.

无论你选择哪一个归一化(normalization)模式，只要你的文本只用一种模式，那你的同一个词元(token)就会由相同的字节(byte)组成。例如，兼容 (compatibility) 模式 可以用连词 ﬃ 的简化形式 `ffi`来进行对比。

你可以使用 icu\_normalizer 语汇单元过滤器(token filters) 来保证你的所有词元(token)是相同模式：

PUT /my\_index

{

"settings": {

"analysis": {

"filter": {

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

"type": "icu\_normalizer",

"name": "nfkc"

}

},

"analyzer": {

"my\_normalizer": {

"tokenizer": "icu\_tokenizer",

"filter": [ "nfkc\_normalizer" ]

}

}

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/unicode-normalization.html#CO143-1) | 用 nfkc 归一化(normalization)模式来归一化(Normalize)所有词元(token). |

提示

包括刚才提到过的 icu\_normalizer 语汇单元过滤器(token filters)在内，这里还有 icu\_normalizer 字符 过滤器(character filters)。虽然它和语汇单元过滤器做相同的工作，但是会在文本到达过滤器之前做。到底是用standard 过滤器，还是 icu\_tokenizer 过滤器，其实并不重要。因为过滤器知道怎么来正确处理所有的模式。

但是，如果你使用不同的分词器，例如： ngram, edge\_ngram, 或者 pattern 分词器，那么在语汇单元过滤器(token filters)之前使用 icu\_normalizer 字符过滤器就变得有意义了。

通常来说，你不仅仅想要归一化(normalize)词元(token)的字节(byte)规则，还需要把他们转成小写字母。这个可以通过 icu\_normalizer 和定制的归一化(normalization)的模式 nfkc\_cf 来实现。下一节我们会具体讲这个。

### Unicode 大小写折叠

人类没有创造力的话就不会是人类， 而人类的语言就恰恰反映了这一点。

处理一个单词的大小写看起来是一个简单的任务，除非遇到需要处理多语言的情况。

那就举一个例子：转换小写德国单词 ß`。把它转换成大写是 `SS`，然后在转换成小写就成了 `ss`。还有一个例子：转换希腊字母 `ς (sigma, 在单词末尾使用)。把它转换成大写是 `Σ`，然后再转换成小写就成了 `σ`。

把词条小写的核心是让他们看起来更像，而不是更不像。在Unicode中，这个工作是大小写折叠(case folding)来完成的，而不是小写化(lowercasing)。 大小写折叠\_ (\_Case folding) 把单词转换到一种(通常是小写)形式，是让写法不会影响单词的比较，所以拼写不需要完全正确。

例如：单词 ß`，已经是小写形式了，会被\_折叠\_(\_folded\_)成 `ss`。类似的小写的 `ς 被折叠成 `σ`，这样的话，无论 `σ`， `ς`， 和 `Σ`出现在哪里， 他们就都可以比较了。

`icu\_normalizer` 语汇单元过滤器默认的归一化(normalization)模式是 `nfkc\_cf`。它像 `nfkc` 模式一样：

* 组合 (Composes) 字符用最短的字节来表示。
* 用 兼容 (\_compatibility\_）模式，把像 ﬃ 的字符转换成简单的 ffi

但是，也会这样做：

* 大小写折叠\_ (\_Case-folds) 字符成一种适合比较的形式

换句话说， nfkc\_cf`等价于 `lowercase 语汇单元过滤器(token filters)，但是却适用于所有的语言。 on-steroids 等价于 standard 分析器，例如：

PUT /my\_index

{

"settings": {

"analysis": {

"analyzer": {

"my\_lowercaser": {

"tokenizer": "icu\_tokenizer",

"filter": [ "icu\_normalizer" ] https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

}

}

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/case-folding.html#CO144-1) | icu\_normalizer 默认是 nfkc\_cf 模式. |

我们来比较 Weißkopfseeadler`和 `WEISSKOPFSEEADLER(大写形式) 分别通过 `standard`分析器和我们的Unicode自识别(Unicode-aware)分析器处理得到的结果：

GET /\_analyze?analyzer=standard https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

Weißkopfseeadler WEISSKOPFSEEADLER

GET /my\_index/\_analyze?analyzer=my\_lowercaser https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

Weißkopfseeadler WEISSKOPFSEEADLER

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/case-folding.html#CO145-1) | 得到的词元(token)是 weißkopfseeadler, weisskopfseeadler |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/case-folding.html#CO145-2) | 得到的词元(token)是 weisskopfseeadler, weisskopfseeadler  `standard`分析器得到了两个不同且不可比较的词元(token)，而我们定制化的分析器得到了两个相同但是不符合原意的词元(token)。 |

### Unicode 字符折叠

在多语言((("Unicode", "character folding")))((("tokens", "normalizing", "Unicode character folding")))处理中，`lowercase` 语汇单元过滤器(token filters)是一个很好的开始。但是作为对比的话，也只是对于整个巴别塔的惊鸿一瞥。所以 <<asciifolding-token-filter,`asciifolding` token filter>> 需要更有效的Unicode \_字符折叠\_ (\_character-folding\_)工具来处理全世界的各种语言。((("asciifolding token filter")))

`icu\_folding` 语汇单元过滤器(token filters) (provided by the <<icu-plugin,`icu` plug-in>>)的功能和 `asciifolding` 过滤器一样， ((("icu\_folding token filter")))但是它扩展到了非ASCII编码的语言，例如：希腊语，希伯来语，汉语。它把这些语言都转换对应拉丁文字，甚至包含它们的各种各样的计数符号，象形符号和标点符号。

`icu\_folding` 语汇单元过滤器(token filters)自动使用 `nfkc\_cf` 模式来进行大小写折叠和Unicode归一化(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

١٢٣٤٥ https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/character-folding.html#CO146-1) | 阿拉伯数字 ١٢٣٤٥ 被折叠成等价的拉丁数字: 12345. |

如果你有指定的字符不想被折叠，你可以使用 [*UnicodeSet*](http://icu-project.org/apiref/icu4j/com/ibm/icu/text/UnicodeSet.html)(像字符的正则表达式) 来指定哪些Unicode才可以被折叠。例如：瑞典单词 å,ä, ö, Å, Ä, 和 Ö 不能被折叠，你就可以设定为： [^åäöÅÄÖ] (^ 表示 *不包含*)。这样就会对于所有的Unicode字符生效。

PUT /my\_index

{

"settings": {

"analysis": {

"filter": {

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

"type": "icu\_folding",

"unicodeSetFilter": "[^åäöÅÄÖ]"

}

},

"analyzer": {

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

"tokenizer": "icu\_tokenizer",

"filter": [ "swedish\_folding", "lowercase" ]

}

}

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/character-folding.html#CO147-1) | `swedish\_folding`语汇单元过滤器(token filters) 定制了 `icu\_folding`语汇单元过滤器(token filters)来不处理那些大写和小写的瑞典单词。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/character-folding.html#CO147-2) | swedish 分析器首先分词，然后用swedish\_folding`语汇单元过滤器来折叠单词，最后把他们走转换为小写，除了被排除在外的单词： ++Å++, `Ä, 或者 `Ö`。 |

### 排序和整理

本章到目前为止，我们已经了解了怎么以搜索为目的去规范化词汇单元。 本章节中要考虑的最终用例是字符串排序。

在 [字符串排序与多字段](https://www.elastic.co/guide/cn/elasticsearch/guide/current/multi-fields.html) （复数域）中，我们解释了 Elasticsearch 为什么不能在 analyzed （分析过）的字符串字段上排序，并演示了如何为同一个域创建 复数域索引 ，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为使用了分析器，而是因为分析器将字符串拆分成了很多词汇单元，就像一个 词汇袋 ，所以 Elasticsearch 不知道使用那一个词汇单元排序。

依赖于 not\_analyzed 域来排序的话不是很灵活：这仅仅允许我们使用原始字符串这一确定的值排序。然而我们 可以 使用分析器来实现另外一种排序规则，只要你选择的分析器总是为每个字符串输出有且仅有一个的词汇单元。

**大小写敏感排序**

想象下我们有三个 用户 文档，文档的 姓名 域分别含有 Boffey 、 BROWN 和 bailey 。首先我们将使用在 [字符串排序与多字段](https://www.elastic.co/guide/cn/elasticsearch/guide/current/multi-fields.html) 中提到的技术，使用 not\_analyzed 域来排序：

PUT /my\_index

{

"mappings": {

"user": {

"properties": {

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

"type": "string",

"fields": {

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

"type": "string",

"index": "not\_analyzed"

}

}

}

}

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/sorting-collations.html#CO148-1) | analyzed name 域用来搜索。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/sorting-collations.html#CO148-2) | not\_analyzed name.raw 域用来排序。 |

我们可以索引一些文档用来测试排序：

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

运行这个搜索请求将会返回这样的文档排序： BROWN 、 Boffey 、 bailey 。 这个是 词典排序 跟 字符串排序 相反。基本上就是大写字母开头的字节要比小写字母开头的字节权重低，所以这些姓名是按照最低值优先排序。

这可能对计算机是合理的，但是对人来说并不是那么合理，人们更期望这些姓名按照字母顺序排序，忽略大小写。为了实现这个，我们需要把每个姓名按照我们想要的排序的顺序索引。

换句话来说，我们需要一个能输出单个小写词汇单元的分析器：

PUT /my\_index

{

"settings": {

"analysis": {

"analyzer": {

"case\_insensitive\_sort": {

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

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

}

}

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/sorting-collations.html#CO149-1) | keyword 分词器将输入的字符串原封不动的输出。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/sorting-collations.html#CO149-2) | lowercase 分词过滤器将词汇单元转化为小写字母。 |

使用 大小写不敏感排序 分析器替换后，现在我们可以将其用在我们的复数域：

PUT /my\_index/\_mapping/user

{

"properties": {

"name": {

"type": "string",

"fields": {

"lower\_case\_sort": { https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

"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

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/sorting-collations.html#CO150-1) | name.lower\_case\_sort 域将会为我们提供大小写不敏感排序。 |

运行这个搜索请求会得到我们想要的文档排序： bailey 、 Boffey 、 BROWN 。

但是这个顺序是正确的么？它符合我门的期望所以看起来像是正确的， 但我们的期望可能受到这个事实的影响：这本书是英文的，我们的例子中使用的所有字母都属于到英语字母表。

如果我们添加一个德语姓名 Böhm 会怎样呢？

现在我们的姓名会返回这样的排序： bailey 、 Boffey 、 BROWN 、 Böhm 。 Böhm 会排在 BROWN 后面的原因是这些单词依然是按照它们表现的字节值排序的。 r 所存储的字节为 0x72 ，而 ö 存储的字节值为 0xF6 ，所以 Böhm 排在最后。每个字符的字节值都是历史的意外。

显然，默认排序顺序对于除简单英语之外的任何事物都是无意义的。事实上，没有完全“正确”的排序规则。这完全取决于你使用的语言。

**语言之间的区别**

每门语言都有自己的排序规则，并且 有时候甚至有多种排序规则。 这里有几个例子，我们前一小节中的四个名字在不同的上下文中是怎么排序的：

* 英语： bailey 、 boffey 、 böhm 、 brown
* 德语： bailey 、 boffey 、 böhm 、 brown
* 德语电话簿： bailey 、 böhm 、 boffey 、 brown
* 瑞典语： bailey, boffey, brown, böhm

注意

德语电话簿将 böhm 放在 boffey 的原因是 ö 和 oe 在处理名字和地点的时候会被看成同义词，所以 böhm 在排序时像是被写成了 boehm 。

**Unicode 归类算法**

归类是将文本按预定义顺序排序的过程。 Unicode 归类算法 或称为 UCA （参见[www.unicode.org/reports/tr10](http://www.unicode.org/reports/tr10/) ） 定义了一种将字符串按照在归类单元表中定义的顺序排序的方法（通常称为排序规则）。

UCA 还定义了 默认 Unicode 排序规则元素表 或称为 DUCET ， DUCET 为无论任何语言的所有 Unicode 字符定义了默认排序。如你所见，没有惟一一个正确的排序规则，所以 DUCET 让更少的人感到烦恼，且烦恼尽可能的小，但它还远不是解决所有排序烦恼的万能药。

而且，明显几乎每种语言都有 自己的排序规则。大多时候使用 DUCET 作为起点并且添加一些自定义规则用来处理每种语言的特性。

UCA 将字符串和排序规则作为输入，并输出二进制排序键。 将根据指定的排序规则对字符串集合进行排序转化为对其二进制排序键的简单比较。

**Unicode 排序**

提示

本节中描述的方法可能会在未来版本的 Elasticsearch 中更改。请查看 [icu plugin](https://www.elastic.co/guide/cn/elasticsearch/guide/current/icu-plugin.html) 文档的最新信息。

icu\_collation 分词过滤器默认使用 DUCET 排序规则。这已经是对默认排序的改进了。想要使用 icu\_collation 我们仅需要创建一个使用默认 icu\_collation 过滤器的分析器：

PUT /my\_index

{

"settings": {

"analysis": {

"analyzer": {

"ducet\_sort": {

"tokenizer": "keyword",

"filter": [ "icu\_collation" ] https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

}

}

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/sorting-collations.html#CO151-1) | 使用默认 DUCET 归类。 |

通常，我们想要排序的字段就是我们想要搜索的字段， 因此我们使用与在 [大小写敏感排序](https://www.elastic.co/guide/cn/elasticsearch/guide/current/sorting-collations.html#case-insensitive-sorting) 中使用的相同的复数域方法：

PUT /my\_index/\_mapping/user

{

"properties": {

"name": {

"type": "string",

"fields": {

"sort": {

"type": "string",

"analyzer": "ducet\_sort"

}

}

}

}

}

使用这个映射， name.sort 域将会含有一个仅用来排序的键。我们没有指定某种语言，所以它会默认会使用 [DUCET collation](https://www.elastic.co/guide/cn/elasticsearch/guide/current/sorting-collations.html#uca) 。

现在，我们可以重新索引我们的案例文档并测试排序：

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

注意

注意，每个文档返回的 sort 键，在前面的例子中看起来像 brown 和 böhm ，现在看起来像天书： ᖔ乏昫တ倈⠀\u0001 。原因是 icu\_collat​​ion 过滤器输出键 仅用于有效分类，不用于任何其他目的。

运行这个搜索请求反问的文档排序为： bailey 、 Boffey 、 Böhm 、 BROWN 。这个排序对英语和德语来说都正确，这已经是一种进步，但是它对德语电话簿和瑞典语来说还不正确。下一步我们为不同的语言自定义映射。

**指定语言**

可以为特定的语言配置 使用归类表的 icu\_collation 过滤器，例如一个国家特定版本的语言，或者像德语电话簿之类的子集。 这个可以按照如下所示通过 使用 language 、 country 、 和 variant 参数来创建自定义版本的分词过滤器：

英语

{ "language": "en" }

德语

{ "language": "de" }

奥地利德语

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

德语电话簿

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

提示

你可以在一下网址阅读更多的 ICU 本地支持： <http://userguide.icu-project.org/locale>.

这个例子演示怎么创建德语电话簿排序规则：

PUT /my\_index

{

"settings": {

"number\_of\_shards": 1,

"analysis": {

"filter": {

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

"type": "icu\_collation",

"language": "de",

"country": "DE",

"variant": "@collation=phonebook"

}

},

"analyzer": {

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

"tokenizer": "keyword",

"filter": [ "german\_phonebook" ]

}

}

}

},

"mappings": {

"user": {

"properties": {

"name": {

"type": "string",

"fields": {

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

"type": "string",

"analyzer": "german\_phonebook"

}

}

}

}

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/sorting-collations.html#CO152-1) | 首先我们为德语电话薄创建一个自定义版本的 icu\_collation 。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/sorting-collations.html#CO152-2) | 之后我们将其包装在自定义的分析器中。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/sorting-collations.html#CO152-3) | 并且为我们的 name.sort 域配置它。 |

像我们之前那样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

现在返回的文档排序为： bailey 、 Böhm 、 Boffey 、 BROWN 。在德语电话簿归类中， Böhm 等同于 Boehm ，所以排在 Boffey 前面。

**多排序规则**

每种语言都可以使用复数域 来支持对同一个域进行多规则排序：

PUT /my\_index/\_mapping/\_user

{

"properties": {

"name": {

"type": "string",

"fields": {

"default": {

"type": "string",

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

},

"french": {

"type": "string",

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

},

"german": {

"type": "string",

"analyzer": "german\_phonebook" https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png

},

"swedish": {

"type": "string",

"analyzer": "swedish" https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/4.png

}

}

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/sorting-collations.html#CO153-1) [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/sorting-collations.html#CO153-2)[https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/sorting-collations.html#CO153-3) [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/4.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/sorting-collations.html#CO153-4) | 我们需要为每个排序规则创建相应的分析器。 |

使用这个映射，只要按照 name.french 、 name.german 或 name.swedish 域排序，就可以为法语、德语和瑞典语用户正确的排序结果了。不支持的语言可以回退到使用 name.default 域，它使用 DUCET 排序顺序。

**自定义排序**

icu\_collation 分词过滤器提供很多 选项，不止 language 、 country 、和 variant ，这些选项可以用于定制排序算法。可用的选项有以下作用：

* 忽略变音符号
* 顺序大写排先或排后，或忽略大小写
* 考虑或忽略标点符号和空白
* 将数字按字符串或数字值排序
* 自定义现有归类或定义自己的归类

这些选项的详细信息超出了本书的范围，更多的信息可以查询 [ICU plug-in documentation](https://github.com/elasticsearch/elasticsearch-analysis-icu) 和 [ICU project collation documentation](http://userguide.icu-project.org/collation/concepts) 。

## 将单词还原为词根

大多数语言的单词都可以 词形变化 ，意味着 下列单词可以改变它们的形态用来表达不同的意思：

* 单复数变化 ： fox 、foxes
* 时态变化 ： pay 、 paid 、 paying
* 性别变化 ： waiter 、 waitress
* 动词人称变化 ： hear 、 hears
* 代词变化 ： I 、 me 、 my
* 不规则变化 ： ate 、 eaten
* 情景变化 ： so be it 、 were it so

虽然词形变化有助于表达，但它干扰了检索，一个单一的词根 词义 （或意义）可能被很多不同的字母序列表达。 英语是一种弱词形变化语言（你可以忽略词形变化并且能得到合理的搜索结果），但是一些其他语言是高度词形变化的并且需要额外的工作来保证高质量的搜索结果。

词干提取 试图移除单词的变化形式之间的差别，从而达到将每个词都提取为它的词根形式。 例如 foxes可能被提取为词根 fox ，移除单数和复数之间的区别跟我们移除大小写之间的区别的方式是一样的。

单词的词根形式甚至有可能不是一个真的单词，单词 jumping 和 jumpiness 或许都会被提取词干为 jumpi。 这并没有什么问题--只要在索引时和搜索时产生相同的词项，搜索会正常的工作。

如果词干提取很容易的话，那只要一个插件就够了。不幸的是，词干提取 是一种遭受两种困扰的模糊的技术：词干弱提取和词干过度提取。

词干弱提取 就是无法将同样意思的单词缩减为同一个词根。例如， jumped 和 jumps 可能被提取为 jump， 但是 jumping 可能被提取为 jumpi 。弱词干提取会导致搜索时无法返回相关文档。

词干过度提取 就是无法将不同含义的单词分开。例如， general 和 generate 可能都被提取为 gener 。 词干过度提取会降低精准度：不相干的文档会在不需要他们返回的时候返回。

**词形还原**

原词是一组相关词的规范形式，或词典形式 — paying 、 paid 和 pays 的原词是 pay 。 通常原词很像与其相关的词，但有时也不像 — is 、 was 、 am 和 being 的原词是 be 。

词形还原，很像词干提取，试图归类相关单词，但是它比词干提取先进一步的是它企图按单词的 词义 ，或意义归类。 同样的单词可能表现出两种意思—例如， wake 可以表现为 to wake up 或 a funeral 。然而词形还原试图区分两个词的词义，词干提取却会将其混为一谈。

词形还原是一种更复杂和高资源消耗的过程，它需要理解单词出现的上下文来决定词的意思。实践中，词干提取似乎比词形还原更高效，且代价更低。

首先我们会讨论下两个 Elasticsearch 使用的经典词干提取器 — [词干提取算法](https://www.elastic.co/guide/cn/elasticsearch/guide/current/algorithmic-stemmers.html) 和 [字典词干提取器](https://www.elastic.co/guide/cn/elasticsearch/guide/current/dictionary-stemmers.html) — 并且在 [选择一个词干提取器](https://www.elastic.co/guide/cn/elasticsearch/guide/current/choosing-a-stemmer.html) 讨论了怎么根据你的需要选择合适的词干提取器。 最后将在 [控制词干提取](https://www.elastic.co/guide/cn/elasticsearch/guide/current/controlling-stemming.html) 和 [原形词干提取](https://www.elastic.co/guide/cn/elasticsearch/guide/current/stemming-in-situ.html) 中讨论如何裁剪词干提取。

### 词干提取算法

Elasticsearch 中的大部分 stemmers （词干提取器）是基于算法的，它们提供了一系列规则用于将一个词提取为它的词根形式，例如剥离复数词末尾的 s 或 es 。提取单词词干时并不需要知道该词的任何信息。

这些基于算法的 stemmers 优点是：可以作为插件使用，速度快，占用内存少，有规律的单词处理效果好。缺点是：没规律的单词例如 be 、 are 、和 am ，或 mice 和 mouse 效果不好。

最早的一个基于算法 的英文词干提取器是 Porter stemmer ，该英文词干提取器现在依然推荐使用。 Martin Porter 后来为了开发词干提取算法创建了 [Snowball language](http://snowball.tartarus.org/) 网站， 很多 Elasticsearch 中使用的词干提取器就是用 Snowball 语言写的。

提示

[kstem token filter](https://www.elastic.co/guide/en/elasticsearch/reference/master/analysis-kstem-tokenfilter.html) 是一款合并了词干提取算法和内置词典的英语分词过滤器。为了避免模糊词不正确提取，这个词典包含一系列根词单词和特例单词。 kstem 分词过滤器相较于 Porter 词干提取器而言不那么激进。

**使用基于算法的词干提取器**

你 可以使用 [porter\_stem](https://www.elastic.co/guide/en/elasticsearch/reference/master/analysis-porterstem-tokenfilter.html) 词干提取器或直接使用 [kstem](https://www.elastic.co/guide/en/elasticsearch/reference/master/analysis-kstem-tokenfilter.html) 分词过滤器，或使用 [snowball](https://www.elastic.co/guide/en/elasticsearch/reference/master/analysis-snowball-tokenfilter.html) 分词过滤器创建一个具体语言的 Snowball 词干提取器。所有基于算法的词干提取器都暴露了用来接受 语言 参数的统一接口： [stemmer token filter](https://www.elastic.co/guide/en/elasticsearch/reference/master/analysis-stemmer-tokenfilter.html) 。

例如，假设你发现 英语 分析器使用的默认词干提取器太激进并且 你想使它不那么激进。首先应在 [language analyzers](https://www.elastic.co/guide/en/elasticsearch/reference/master/analysis-lang-analyzer.html) 查看 英语 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", https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

"keywords": []

},

"english\_stemmer": {

"type": "stemmer",

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

},

"english\_possessive\_stemmer": {

"type": "stemmer",

"language": "possessive\_english" https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png

}

},

"analyzer": {

"english": {

"tokenizer": "standard",

"filter": [

"english\_possessive\_stemmer",

"lowercase",

"english\_stop",

"english\_keywords",

"english\_stemmer"

]

}

}

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/algorithmic-stemmers.html#CO154-1) | keyword\_marker 分词过滤器列出那些不用被词干提取的单词。这个过滤器默认情况下是一个空的列表。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/algorithmic-stemmers.html#CO154-2) [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/algorithmic-stemmers.html#CO154-3) | english 分析器使用了两个词干提取器： possessive\_english 词干提取器和 english 词干提取器。所有格词干提取器会在任何词传递到 english\_stop 、 english\_keywords 和 english\_stemmer 之前去除 's 。 |

重新审视下现在的配置，添加上以下修改，我们可以把这份配置当作新分析器的基本配置：

* 修改 english\_stemmer ，将 english （[porter\_stem](https://www.elastic.co/guide/en/elasticsearch/reference/master/analysis-porterstem-tokenfilter.html) 分词过滤器的映射）替换为 light\_english（非激进的 [kstem](https://www.elastic.co/guide/en/elasticsearch/reference/master/analysis-kstem-tokenfilter.html) 分词过滤器的映射）。
* 添加 [asciifolding](https://www.elastic.co/guide/cn/elasticsearch/guide/current/asciifolding-token-filter.html) 分词过滤器用以移除外语的附加符号。
* 移除 keyword\_marker 分词过滤器，因为我们不需要它。（我们会在 [控制词干提取](https://www.elastic.co/guide/cn/elasticsearch/guide/current/controlling-stemming.html) 中详细讨论它）

新定义的分析器会像下面这样:

PUT /my\_index

{

"settings": {

"analysis": {

"filter": {

"english\_stop": {

"type": "stop",

"stopwords": "\_english\_"

},

"light\_english\_stemmer": {

"type": "stemmer",

"language": "light\_english" https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

},

"english\_possessive\_stemmer": {

"type": "stemmer",

"language": "possessive\_english"

}

},

"analyzer": {

"english": {

"tokenizer": "standard",

"filter": [

"english\_possessive\_stemmer",

"lowercase",

"english\_stop",

"light\_english\_stemmer", https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

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

]

}

}

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/algorithmic-stemmers.html#CO155-1) [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/algorithmic-stemmers.html#CO155-2) | 将 english 词干提取器替换为非激进的 light\_english 词干提取器 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/algorithmic-stemmers.html#CO155-3) | 添加 asciifolding 分词过滤器 |

### 字典词干提取器

字典词干提取器 在工作机制上与 [算法化词干提取器](https://www.elastic.co/guide/cn/elasticsearch/guide/current/algorithmic-stemmers.html) 完全不同。 不同于应用一系列标准规则到每个词上，字典词干提取器只是简单地在字典里查找词。理论上可以给出比算法化词干提取器更好的结果。一个字典词干提取器应当可以：

* 返回不规则形式如 feet 和 mice 的正确词干
* 区分出词形相似但词义不同的情形，比如 organ and organization

实践中一个好的算法化词干提取器一般优于一个字典词干提取器。应该有以下两大原因：

字典质量

一个字典词干提取器再好也就跟它的字典一样。 据牛津英语字典网站估计，英语包含大约75万个单词（包含变音变形词）。电脑上的大部分英语字典只包含其中的 10% 。

词的含义随时光变迁。mobility 提取词干 mobil 先前可能讲得通，但现在合并进了手机可移动性的含义。字典需要保持最新，这是一项很耗时的任务。通常等到一个字典变得好用后，其中的部分内容已经过时。

字典词干提取器对于字典中不存在的词无能为力。而一个基于算法的词干提取器，则会继续应用之前的相同规则，结果可能正确或错误。

大小与性能

字典词干提取器需要加载所有词汇、 所有前缀，以及所有后缀到内存中。这会显著地消耗内存。找到一个词的正确词干，一般比算法化词干提取器的相同过程更加复杂。

依赖于不同的字典质量，去除前后缀的过程可能会更加高效或低效。低效的情形可能会明显地拖慢整个词干提取过程。

另一方面，算法化词干提取器通常更简单、轻量和快速。

提示

如果你所使用的语言有比较好的算法化词干提取器，这通常是比一个基于字典的词干提取器更好的选择。对于算法化词干提取器效果比较差（或者压根没有）的语言，可以使用拼写检查（Hunspell）字典词干提取器，下一个章节会讨论。

### Hunspell 词干提取器

Elasticsearch 提供了基于词典提取词干的 [hunspell 语汇单元过滤器（token filter）](https://www.elastic.co/guide/en/elasticsearch/reference/master/analysis-hunspell-tokenfilter.html). Hunspell [hunspell.github.io](http://hunspell.github.io/) 是一个 Open Office、LibreOffice、Chrome、Firefox、Thunderbird 等众多其它开源项目都在使用的拼写检查器。

可以从这里获取 Hunspell 词典 ：

* [extensions.openoffice.org](http://extensions.openoffice.org/): 下载解压 .oxt 后缀的文件。
* [addons.mozilla.org](http://mzl.la/157UORf): 下载解压 .xpi 扩展文件。
* [OpenOffice archive](http://download.services.openoffice.org/contrib/dictionaries/): 下载解压 .zip 文件。

一个 Hunspell 词典由两个文件组成 — 具有相同的文件名和两个不同的后缀 — 如 en\_US—和下面的两个后缀的其中一个：

.dic

包含所有词根，采用字母顺序，再加上一个代表所有可能前缀和后缀的代码表 【集体称之为词缀( affixes 】

.aff

包含实际 .dic 文件每一行代码表对应的前缀和后缀转换

**安装一个词典**

Hunspell 语汇单元过滤器 在特定的 Hunspell 目录里寻找词典， 默认目录是 ./config/hunspell/ 。 .dic 文件和 .aff 文件应该要以子目录且按语言/区域的方式来命名。 例如，我们可以为美式英语创建一个 Hunspell 词干提取器，目录结构如下：

config/

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

└ en\_US/ https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

├ en\_US.dic

├ en\_US.aff

└ settings.yml https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/hunspell.html#CO156-1) | Hunspell 目录位置可以通过编辑 config/elasticsearch.yml 文件的：indices.analysis.hunspell.dictionary.location 设置来修改。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/hunspell.html#CO156-2) | en\_US 是这个区域的名字，也是我们传给 hunspell 语汇单元过滤器参数 language 值。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/hunspell.html#CO156-3) | 一个语言一个设置文件，下面的章节会具体介绍。 |

**按语言设置**

在语言的目录设置文件 settings.yml 包含适用于所有字典内的语言目录的设置选项。

---

ignore\_case: true

strict\_affix\_parsing: true

这些选项的意思如下：

ignore\_case

Hunspell 目录默认是区分大小写的，如，姓氏 Booker 和名词 booker 是不同的词，所以应该分别进行词干提取。 也许让 hunspell 提取器区分大小写是一个好主意，不过也可能让事情变得复杂：

* 一个句子的第一个词可能会被大写，因此感觉上会像是一个名词。
* 输入的文本可能全是大写，如果这样那几乎一个词都找不到。
* 用户也许会用小写来搜索名字，在这种情况下，大写开头的词将找不到。

一般来说，设置参数 ignore\_case 为 true 是一个好主意。

strict\_affix\_parsing

词典的质量千差万别。 一些网上的词典的 .aff 文件有很多畸形的规则。 默认情况下，如果 Lucene 不能正常解析一个词缀(affix)规则， 它会抛出一个异常。 你可以通过设置 strict\_affix\_parsing 为 false 来告诉 Lucene 忽略错误的规则。

**自定义词典**

如果一个目录放置了多个词典 (.dic 文件)， 他们会在加载时合并到一起。这可以让你以自定义的词典的方式对下载的词典进行定制：

config/

└ hunspell/

└ en\_US/ https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

├ en\_US.dic

├ en\_US.aff https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

├ custom.dic

└ settings.yml

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/hunspell.html#CO157-1) | custom 词典和 en\_US 词典将合并到一起。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/hunspell.html#CO157-2) | 多个 .aff 文件是不允许的，因为会产生规则冲突。 |

.dic 文件和 .aff 文件的格式在这里讨论： [Hunspell 词典格式](https://www.elastic.co/guide/cn/elasticsearch/guide/current/hunspell.html#hunspell-dictionary-format) 。

**创建一个 Hunspell 语汇单元过滤器**

一旦你在所有节点上安装好了词典，你就能像这样定义一个 hunspell 语汇单元过滤器 ：

PUT /my\_index

{

"settings": {

"analysis": {

"filter": {

"en\_US": {

"type": "hunspell",

"language": "en\_US" https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

}

},

"analyzer": {

"en\_US": {

"tokenizer": "standard",

"filter": [ "lowercase", "en\_US" ]

}

}

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/hunspell.html#CO158-1) | 参数 language 和目录下对应的名称相同。 |

你可以通过 analyze API 来测试这个新的分析器， 然后和 english 分析器比较一下它们的输出：

GET /my\_index/\_analyze?analyzer=en\_US https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

reorganizes

GET /\_analyze?analyzer=english https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

reorganizes

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/hunspell.html#CO159-1) | 返回 organize |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/hunspell.html#CO159-2) | 返回 reorgan |

在前面的例子中，hunspell 提取器有一个有意思的事情，它不仅能移除前缀还能移除后缀。大多数算法词干提取仅能移除后缀。

提示

Hunspell 词典会占用几兆的内存。幸运的是，Elasticsearch 每个节点只会创建一个词典的单例。 所有的分片都会使用这个相同的 Hunspell 分析器。

**Hunspell 词典格式**

尽管使用 hunspell 不必了解 Hunspell 词典的格式， 不过了解格式可以帮助我们编写自己的自定义的词典。其实很简单。

例如，在美式英语词典（US English dictionary），en\_US.dic 文件包含了一个包含词 analyze 的实体，看起来如下：

analyze/ADSG

en\_US.aff 文件包含了一个针对标记 A 、 G 、D 和 S 的前后缀的规则。 其中应该只有一个能匹配，每一个规则的格式如下：

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

例如，下面的后缀 (SFX) 规则 D 。它是说，当一个词由一个辅音 (除了 a 、e 、i 、o 或 u 外的任意音节) 后接一个 y ，那么它可以移除 y 和添加 ied 结尾 （如，ready → readied ）。

SFX D y ied [^aeiou]y

前面提到的 A 、 G 、D 和 S 标记对应规则如下：

SFX D Y 4

SFX D 0 d e https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

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

SFX G Y 2

SFX G e ing e https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png

SFX G 0 ing [^e]

PFX A Y 1

PFX A 0 re . https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/4.png

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/hunspell.html#CO160-1) | analyze 以一个 e 结尾，所以它可以添加一个 d 变成 analyzed 。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/hunspell.html#CO160-2) | analyze 不是由 s 、x 、z 、h 或 y 结尾，所以，它可以添加一个 s 变成 analyzes 。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/hunspell.html#CO160-3) | analyze 以一个 e 结尾，所以，它可以移除 e 和添加 ing 然后变成 analyzing 。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/4.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/hunspell.html#CO160-4) | 可以添加前缀 re 来形成 reanalyze 。这个规则可以组合后缀规则一起形成： reanalyzes 、reanalyzed 、 reanalyzing 。 |

了解更多关于 Hunspell 的语法，可以前往 [Hunspell 文档](http://sourceforge.net/projects/hunspell/files/Hunspell/Documentation/) 。

### 选择一个词干提取器

在文档 [stemmer](https://www.elastic.co/guide/en/elasticsearch/reference/master/analysis-stemmer-tokenfilter.html) token filter 里面列出了一些针对语言的若干词干提取器。 就英语来说我们有如下提取器：

english

[porter\_stem](https://www.elastic.co/guide/en/elasticsearch/reference/master/analysis-porterstem-tokenfilter.html) 语汇单元过滤器（token filter）。

light\_english

[kstem](https://www.elastic.co/guide/en/elasticsearch/reference/master/analysis-kstem-tokenfilter.html) 语汇单元过滤器（token filter）。

minimal\_english

Lucene 里面的 EnglishMinimalStemmer ，用来移除复数。

lovins

基于 [Snowball](https://www.elastic.co/guide/en/elasticsearch/reference/master/analysis-snowball-tokenfilter.html) 的 [Lovins](http://snowball.tartarus.org/algorithms/lovins/stemmer.html) 提取器, 第一个词干提取器。

porter

基于 [Snowball](https://www.elastic.co/guide/en/elasticsearch/reference/master/analysis-snowball-tokenfilter.html) 的 [Porter](http://snowball.tartarus.org/algorithms/porter/stemmer.html) 提取器。

porter2

基于 [Snowball](https://www.elastic.co/guide/en/elasticsearch/reference/master/analysis-snowball-tokenfilter.html) 的 [Porter2](http://snowball.tartarus.org/algorithms/english/stemmer.html) 提取器。

possessive\_english

Lucene 里面的 EnglishPossessiveFilter ，移除 's

Hunspell 词干提取器也要纳入到上面的列表中，还有多种英文的词典可用。

有一点是可以肯定的：当一个问题存在多个解决方案的时候，这意味着没有一个解决方案充分解决这个问题。 这一点同样体现在词干提取上 — 每个提取器使用不同的方法不同程度的对单词进行了弱提取或是过度提取。

在 stemmer 文档 中，使用粗体高亮了每一个语言的推荐的词干提取器， 通常是因为它提供了一个在性能和质量之间合理的妥协。也就是说，推荐的词干提取器也许不适用所有场景。 关于哪个是最好的词干提取器，不存在一个唯一的正确答案 — 它要看你具体的需求。 这里有3个方面的因素需要考虑在内： 性能、质量、程度。

**提取性能**

算法提取器一般来说比 Hunspell 提取器快4到5倍。 “Handcrafted” 算法提取器通常（不是永远） 要比 Snowball 快或是差不多。 比如，porter\_stem 语汇单元过滤器（token filter）就明显要比基于 Snowball 实现的 Porter 提取器要快的多。

Hunspell 提取器需要加载所有的词典、前缀和后缀表到内存，可能需要消耗几兆的内存。而算法提取器，由一点点代码组成，只需要使用很少内存。

**提取质量**

所有的语言，除了世界语（Esperanto）都是不规范的。 最日常用语使用的词往往不规则，而更正式的书面用语则往往遵循规律。 一些提取算法经过多年的开发和研究已经能够产生合理的高质量的结果了，其他人只需快速组装做很少的研究就能解决大部分的问题了。

虽然 Hunspell 提供了精确地处理不规则词语的承诺，但在实践中往往不足。 一个基于词典的提取器往往取决于词典的好坏。如果 Hunspell 碰到的这个词不在词典里，那它什么也不能做。 Hunspell 需要一个广泛的、高质量的、最新的词典以产生好的结果；这样级别的词典可谓少之又少。 另一方面，一个算法提取器，将愉快的处理新词而不用为新词重新设计算法。

如果一个好的算法词干提取器可用于你的语言，那明智的使用它而不是 Hunspell。它会更快并且消耗更少内存，并且会产生和通常一样好或者比 Hunspell 等价的结果.

如果精度和可定制性对你很重要，那么你需要（和有精力）来维护一个自定义的词典，那么 Hunspell 会给你比算法提取器更大的灵活性。 (查看 [控制词干提取](https://www.elastic.co/guide/cn/elasticsearch/guide/current/controlling-stemming.html) 来了解可用于任何词干提取器的自定义技术。)

**提取程度**

不同的词干提取器会将词弱提取或过度提取到一定的程度 。 light\_ 提取器提干力度不及标准的提取器。 minimal\_ 提取器同样也不那么积极。Hunspell 提取力度要激进一些。

是否想要积极提取还是轻量提取取决于你的场景。如果你的搜索结果是要用于聚类算法，你可能会希望匹配的更广泛一点（因此，提取力度要更大一点）。 如果你的搜索结果是面向最终用户，轻量的提取一般会产生更好的结果。对搜索来说，将名称和形容词提干比动词提干更重要，当然这也取决于语言。

另外一个要考虑的因素就是你的文档集的大小。 一个只有 10,000 个产品的小集合，你可能要更激进的提干来确保至少匹配到一些文档。 如果你的文档集很大，使用轻量的弱提取可能会得到更好的匹配结果。

**做一个选择**

从推荐的一个词干提取器出发，如果它工作的很好，那没有什么需要调整的。如果不是，你将需要花点时间来调查和比较该语言可用的各种不同提取器， 来找到最适合你目的的那一个。

### 控制词干提取

开箱即用的词干提取方案永远也不可能完美。 尤其是算法提取器，他们可以愉快的将规则应用于任何他们遇到的词，包含那些你希望保持独立的词。 也许，在你的场景，保持独立的 skies 和 skiing 是重要的，你不希望把他们提取为 ski （正如 english 分析器那样）。

语汇单元过滤器 [keyword\_marker](https://www.elastic.co/guide/en/elasticsearch/reference/master/analysis-keyword-marker-tokenfilter.html) 和 [stemmer\_override](https://www.elastic.co/guide/en/elasticsearch/reference/master/analysis-stemmer-override-tokenfilter.html) 能让我们自定义词干提取过程。

**阻止词干提取**

语言分析器（查看 [配置语言分析器](https://www.elastic.co/guide/cn/elasticsearch/guide/current/configuring-language-analyzers.html)）的参数 [stem\_exclusion](https://www.elastic.co/guide/cn/elasticsearch/guide/current/configuring-language-analyzers.html#stem-exclusion) 允许我们指定一个词语列表，让他们不被词干提取。

在内部，这些语言分析器使用 [keyword\_marker 语汇单元过滤器](https://www.elastic.co/guide/en/elasticsearch/reference/master/analysis-keyword-marker-tokenfilter.html) 来标记这些词语列表为 *keywords* ，用来阻止后续的词干提取过滤器来触碰这些词语。

例如，我们创建一个简单自定义分析器，使用 [porter\_stem](https://www.elastic.co/guide/en/elasticsearch/reference/master/analysis-porterstem-tokenfilter.html) 语汇单元过滤器，同时阻止 skies 的词干提取：

PUT /my\_index

{

"settings": {

"analysis": {

"filter": {

"no\_stem": {

"type": "keyword\_marker",

"keywords": [ "skies" ] https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

}

},

"analyzer": {

"my\_english": {

"tokenizer": "standard",

"filter": [

"lowercase",

"no\_stem",

"porter\_stem"

]

}

}

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/controlling-stemming.html#CO161-1) | 参数 keywords 可以允许接收多个词语。 |

使用 analyze API 来测试，可以看到词 skies 没有被提取：

GET /my\_index/\_analyze?analyzer=my\_english

sky skies skiing skis https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/controlling-stemming.html#CO162-1) | 返回: sky, skies, ski, ski |

提示

虽然语言分析器只允许我们通过参数 stem\_exclusion 指定一个词语列表来排除词干提取，不过 keyword\_marker 语汇单元过滤器同样还接收一个 keywords\_path 参数允许我们将所有的关键字存在一个文件。 这个文件应该是每行一个字，并且存在于集群的每个节点。查看 [更新停用词（Updating Stopwords）](https://www.elastic.co/guide/cn/elasticsearch/guide/current/using-stopwords.html#updating-stopwords) 了解更新这些文件的提示。

**自定义提取**

在上面的例子中，我们阻止了 skies 被词干提取，但是也许我们希望他能被提干为 sky 。 The[stemmer\_override](https://www.elastic.co/guide/en/elasticsearch/reference/master/analysis-stemmer-override-tokenfilter.html) 语汇单元过滤器允许我们指定自定义的提取规则。 与此同时，我们可以处理一些不规则的形式，如：mice 提取为 mouse 和 feet 到 foot ：

PUT /my\_index

{

"settings": {

"analysis": {

"filter": {

"custom\_stem": {

"type": "stemmer\_override",

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

"skies=>sky",

"mice=>mouse",

"feet=>foot"

]

}

},

"analyzer": {

"my\_english": {

"tokenizer": "standard",

"filter": [

"lowercase",

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

"porter\_stem"

]

}

}

}

}

}

GET /my\_index/\_analyze?analyzer=my\_english

The mice came down from the skies and ran over my feet https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/controlling-stemming.html#CO163-1) | 规则来自 original=>stem 。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/controlling-stemming.html#CO163-2) | stemmer\_override 过滤器必须放置在词干提取器之前。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/controlling-stemming.html#CO163-3) | 返回 the, mouse, came, down, from, the, sky, and, ran, over, my, foot 。 |

提示

正如 keyword\_marker 语汇单元过滤器，规则可以被存放在一个文件中，通过参数 rules\_path 来指定位置。

### 原形词干提取

为了完整地 完成本章的内容，我们将讲解如何将已提取词干的词和原词索引到同一个字段中。举个例子，分析句子 The quick foxes jumped 将会得到以下词项：

Pos 1: (the)

Pos 2: (quick)

Pos 3: (foxes,fox) https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

Pos 4: (jumped,jump) https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/stemming-in-situ.html#CO164-1) [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/stemming-in-situ.html#CO164-2) | 已提取词干的形式和未提取词干的形式位于相同的位置。 |

Warning：使用此方法前请先阅读 [原形词干提取是个好主意吗](https://www.elastic.co/guide/cn/elasticsearch/guide/current/stemming-in-situ.html#stemming-in-situ-good-idea) 。

为了归档词干提取出的 原形 ，我们将使用 [keyword\_repeat](http://www.elastic.co/guide/en/elasticsearch/reference/current/analysis-keyword-repeat-tokenfilter.html) 过滤器，跟 keyword\_marker 过滤器 ( see [阻止词干提取](https://www.elastic.co/guide/cn/elasticsearch/guide/current/controlling-stemming.html#preventing-stemming) ) 一样，它把每一个词项都标记为关键词，以防止后续词干提取器对其修改。但是，它依然会在相同位置上重复词项，并且这个重复的词项 **是** 提取的词干。

单独使用 keyword\_repeat token 过滤器将得到以下结果：

Pos 1: (the,the) https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

Pos 2: (quick,quick) https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

Pos 3: (foxes,fox)

Pos 4: (jumped,jump)

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/stemming-in-situ.html#CO165-1) [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/stemming-in-situ.html#CO165-2) | 提取词干前后的形式一样，所以只是不必要的重复。 |

为了防止提取和未提取词干形式相同的词项中的无意义重复，我们增加了组合的 [unique](https://www.elastic.co/guide/en/elasticsearch/reference/master/analysis-unique-tokenfilter.html) 语汇单元过滤器：

PUT /my\_index

{

"settings": {

"analysis": {

"filter": {

"unique\_stem": {

"type": "unique",

"only\_on\_same\_position": true https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

}

},

"analyzer": {

"in\_situ": {

"tokenizer": "standard",

"filter": [

"lowercase",

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

"porter\_stem",

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

]

}

}

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/stemming-in-situ.html#CO166-1) | 设置 unique 类型语汇单元过滤器，是为了只有当重复语汇单元出现在相同位置时，移除它们。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/stemming-in-situ.html#CO166-2) | 语汇单元过滤器必须出现在词干提取器之前。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/stemming-in-situ.html#CO166-3) | unique\_stem 过滤器是在词干提取器完成之后移除重复词项。 |

**原形词干提取是个好主意吗**

用户喜欢 原形 词干提取这个主意：“如果我可以只用一个组合字段，为什么还要分别存一个未提取词干和已提取词干的字段呢？” 但这是一个好主意吗？答案一直都是否定的。因为有两个问题：

第一个问题是无法区分精准匹配和非精准匹配。本章中，我们看到了多义词经常会被展开成相同的词干词：organs 和 organization 都会被提取为 organ 。

在 [使用语言分析器](https://www.elastic.co/guide/cn/elasticsearch/guide/current/using-language-analyzers.html) 我们展示了如何整合一个已提取词干属性的查询(为了增加召回率)和一个未提取词干属性的查询（为了提升相关度）。 当提取和未提取词干的属性相互独立时，单个属性的贡献可以通过给其中一个属性增加boost值来优化(参见 [语句的优先级](https://www.elastic.co/guide/cn/elasticsearch/guide/current/multi-query-strings.html#prioritising-clauses) )。相反地，如果已提取和未提取词干的形式置于同一个属性，就没有办法来优化搜索结果了。

第二个问题是，必须搞清楚 相关度分值是否如何计算的。在 [什么是相关性?](https://www.elastic.co/guide/cn/elasticsearch/guide/current/relevance-intro.html) 我们解释了部分计算依赖于逆文档频率（IDF）—— 即一个词在索引库的所有文档中出现的频繁程度。 在一个包含文本 jump jumped jumps 的文档上使用原形词干提取，将得到下列词项：

Pos 1: (jump)

Pos 2: (jumped,jump)

Pos 3: (jumps,jump)

jumped 和 jumps 各出现一次，所以有正确的IDF值；jump 出现了3次，作为一个搜索词项，与其他未提取词干的形式相比，这明显降低了它的IDF值。

基于这些原因，我们不推荐使用原形词干提取。

## 停用词: 性能与精度

从早期的信息检索到如今， 我们已习惯于磁盘空间和内存被限制为很小一部分，所以 必须使你的索引尽可能小。 每个字节都意味着巨大的性能提升。 (查看 [*将单词还原为词根*](https://www.elastic.co/guide/cn/elasticsearch/guide/current/stemming.html) ) 词干提取的重要性不仅是因为它让搜索的内容更广泛、让检索的能力更深入，还因为它是压缩索引空间的工具。

一种最简单的减少索引大小的方法就是 \_索引更少的词\_。 有些词要比其他词更重要，只索引那些更重要的词来可以大大减少索引的空间。

那么哪些词条可以被过滤呢？ 我们可以简单分为两组:

低频词（Low-frequency terms）

在文档集合中相对出现较少的词，因为它们稀少，所以它们的权重值更高。

高频词（High-frequency terms）

在索引下的文档集合中出现较多的常用词，例如 `the`、`and`、和`is`。 这些词的权重小，对相关度评分影响不大。

提示

当然，频率实际上是个可以衡量的标尺而不是非 *高* 即 *低* 的标签。我们可以在标尺的任何位置选取一个标准，低于这个标准的属于低频词，高于它的属于高频词。

词项到底是低频或是高频取决于它们所处的文档。单词 and 如果在所有都是中文的文档里可能是个低频词。在关于数据库的文档集合里，单词 database 可能是一个高频词项，它对搜索这个特定集合毫无帮助。

每种语言都存在一些非常常见的单词，它们对搜索没有太大价值。在 Elasticsearch 中，英语默认的停用词为:

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

这些 *停用词* 通常在索引前就可以被过滤掉，同时对检索的负面影响不大。但是这样做真的是一个较好的解决方案？

### 停用词的优缺点

现在我们拥有更大的磁盘空间，更多内存，并且 还有更好的压缩算法。 将之前的 33 个常见词从索引中移除，每百万文档只能节省 4MB 空间。 所以使用停用词减少索引大小不再是一个有效的理由。 (不过这种说法还有一点需要注意，我们在 [停用词与短语查询](https://www.elastic.co/guide/cn/elasticsearch/guide/current/stopwords-phrases.html) 讨论。)

在此基础上，从索引里将这些词移除会使我们降低某种类型的搜索能力。将前面这些所列单词移除会让我们难以完成以下事情：

* 区分 happy 和 \_not happy\_。
* 搜索乐队名称 The The。
* 查找莎士比亚的名句 “To be, or not to be” （生存还是毁灭)。
* 使用挪威的国家代码: `no`。

移除停用词的最主要好处是性能，假设我们在个具有上百万文档的索引中搜索单词 fox`。或许 `fox 只在其中 20 个文档中出现，也就是说 Elasticsearch 需要计算 20 个文档的相关度评分 `\_score `从而排出前十。现在我们把搜索条件改为 `the OR fox`，几乎所有的文件都包含 `the 这个词，也就是说 Elasticsearch 需要为所有一百万文档计算评分 `\_score`。 由此可见第二个查询肯定没有第一个的结果好。

幸运的是，我们可以用来保持常用词搜索，同时还可以保持良好的性能。首先我们一块学习如何使用停用词。

### 使用停用词

移除停用词的工作是由 stop 停用词过滤器完成的，可以通过创建自定义的分析器来使用它（参见 使用停用词过滤器[stop 停用词过滤器](https://www.elastic.co/guide/en/elasticsearch/reference/master/analysis-stop-tokenfilter.html))。但是，也有一些自带的分析器预置使用停用词过滤器：

[语言分析器](https://www.elastic.co/guide/en/elasticsearch/reference/master/analysis-lang-analyzer.html)

每个语言分析器默认使用与该语言相适的停用词列表，例如：english 英语分析器使用 \_english\_ 停用词列表。

[standard 标准分析器](https://www.elastic.co/guide/en/elasticsearch/reference/master/analysis-standard-analyzer.html)

默认使用空的停用词列表：\_none\_ ，实际上是禁用了停用词。

[pattern 模式分析器](https://www.elastic.co/guide/en/elasticsearch/reference/master/analysis-pattern-analyzer.html)

默认使用空的停用词列表：为 \_none\_ ，与 standard 分析器类似。

**停用词和标准分析器（Stopwords and the Standard Analyzer）**

为了让 标准分析器能与 自定义停用词表连用，我们要做的只需创建一个分析器的配置好的版本，然后将停用词列表传入：

PUT /my\_index

{

"settings": {

"analysis": {

"analyzer": {

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

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

"stopwords": [ "and", "the" ] https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png

}

}

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/using-stopwords.html#CO167-1) | 自定义的分析器名称为 my\_analyzer 。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/using-stopwords.html#CO167-2) | 这个分析器是一个标准 standard 分析器，进行了一些自定义配置。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/using-stopwords.html#CO167-3) | 过滤掉的停用词包括 and 和 the 。 |

提示

任何语言分析器都可以使用相同的方式配置自定义停用词。

**保持位置（Maintaining Positions）**

analyzer API 的输出结果很有趣:

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

},

{

"token": "dead",

"start\_offset": 18,

"end\_offset": 22,

"type": "<ALPHANUM>",

"position": 4

}

]

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/using-stopwords.html#CO168-1) | position 标记每个词汇单元的位置。 |

停用词如我们期望被过滤掉了，但有趣的是两个词项的位置 position 没有变化：quick 是原句子的第二个词，dead 是第五个。这对短语查询十分重要，因为如果每个词项的位置被调整了，一个短语查询 quick dead 会与以上示例中的文档错误匹配。

**指定停用词（Specifying Stopwords）**

停用词可以以内联的方式传入，就像我们在前面的 例子中那样，通过指定数组:

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

特定语言的默认停用词，可以通过使用 \_lang\_ 符号来指定:

"stopwords": "\_english\_"

TIP: Elasticsearch 中预定义的与语言相关的停用词列表可以在文档"languages"[stop 停用词过滤器](https://www.elastic.co/guide/en/elasticsearch/reference/master/analysis-stop-tokenfilter.html) 中找到。

停用词可以通过指定一个特殊列表 \_none\_ 来禁用。例如，使用 \_english\_ 分析器而不使用停用词，可以通过以下方式做到：

PUT /my\_index

{

"settings": {

"analysis": {

"analyzer": {

"my\_english": {

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

"stopwords": "\_none\_" https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

}

}

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/using-stopwords.html#CO169-1) | my\_english 分析器是基于 english 分析器。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/using-stopwords.html#CO169-2) | 但禁用了停用词。 |

最后，停用词还可以使用一行一个单词的格式保存在文件中。此文件必须在集群的所有节点上，并且通过 stopwords\_path 参数设置路径:

PUT /my\_index

{

"settings": {

"analysis": {

"analyzer": {

"my\_english": {

"type": "english",

"stopwords\_path": "stopwords/english.txt" https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

}

}

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/using-stopwords.html#CO170-1) | 停用词文件的路径，该路径相对于 Elasticsearch 的 config 目录。 |

**使用停用词过滤器（Using the stop Token Filter）**

当你创建 custom 分析器时候，可以组合多个 [stop 停用词过滤器](https://www.elastic.co/guide/en/elasticsearch/reference/master/analysis-stop-tokenfilter.html) 分词器 。例如：我们想要创建一个西班牙语 的分析器:

* 自定义停用词列表
* light\_spanish 词干提取器
* 在 asciifolding 词汇单元过滤器中除去附加符号

我们可以通过以下设置完成:

PUT /my\_index

{

"settings": {

"analysis": {

"filter": {

"spanish\_stop": {

"type": "stop",

"stopwords": [ "si", "esta", "el", "la" ] https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

},

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

"type": "stemmer",

"language": "light\_spanish"

}

},

"analyzer": {

"my\_spanish": {

"tokenizer": "spanish",

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

"lowercase",

"asciifolding",

"spanish\_stop",

"light\_spanish"

]

}

}

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/using-stopwords.html#CO171-1) | 停用词过滤器采用与 standard 分析器相同的参数 stopwords 和 stopwords\_path 。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/using-stopwords.html#CO171-2) | 参见 算法提取器（Algorithmic Stemmers）。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/using-stopwords.html#CO171-3) | 过滤器的顺序非常重要，下面会进行解释。 |

我们将 spanish\_stop 过滤器放置在 asciifolding 过滤器之后.这意味着以下三个词组 esta 、ésta 、++está++ ，先通过 asciifolding 过滤器过滤掉特殊字符变成了 esta ，随后使用停用词过滤器会将 esta 去除。 如果我们只想移除 esta 和 ésta ，但是 ++está++ 不想移除。必须将 spanish\_stop 过滤器放置在 asciifolding 之前，并且需要在停用词中指定 esta 和 ésta 。

**更新停用词（Updating Stopwords）**

想要更新分析器的停用词列表有多种方式， 分析器在创建索引时，当集群节点重启时候，或者关闭的索引重新打开的时候。

如果你使用 stopwords 参数以内联方式指定停用词，那么你只能通过关闭索引，更新分析器的配置[update index settings API](https://www.elastic.co/guide/en/elasticsearch/reference/master/indices-update-settings.html#update-settings-analysis)，然后在重新打开索引才能更新停用词。

如果你使用 stopwords\_path 参数指定停用词的文件路径 ，那么更新停用词就简单了。你只需更新文件(在每一个集群节点上)，然后通过两者之中的任何一个操作来强制重新创建分析器:

* 关闭和重新打开索引 (参考 [索引的开与关](https://www.elastic.co/guide/en/elasticsearch/reference/master/indices-open-close.html))，
* 一一重启集群下的每个节点。

当然，更新的停用词不会改变任何已经存在的索引。这些停用词的只适用于新的搜索或更新文档。如果要改变现有的文档，则需要重新索引数据。参加 [重新索引你的数据](https://www.elastic.co/guide/cn/elasticsearch/guide/current/reindex.html) 。

### 停用词与性能

保留停用词最大的缺点就影响搜索性能。使用 Elasticsearch 进行 全文搜索，它需要为所有匹配的文档计算相关度评分 \_score 从而返回最相关的前 10 个文档。

通常大多数的单词在所有文档中出现的频率低于0.1％，但是有少数词（例如 the ）几乎存在于所有的文档中。假设有一个索引含有100万个文档，查询 quick brown fox 词组，能够匹配上的可能少于1000个文档。但是如果查询 the quick brown fox 词组，几乎需要对索引中的100万个文档进行评分和排序，只是为了返回前 10 名最相关的文档。

问题的关键是 the quick brown fox 词组实际是查询 the 或 quick 或 brown 或 fox— 任何文档即使它什么内容都没有而只包含 the 这个词也会被包括在结果集中。因此，我们需要找到一种降低待评分文档数量的方法。

**and 操作符 (and Operator)**

我们想要减少待评分文档的数量，最简单的方式就是在[and 操作符](https://www.elastic.co/guide/cn/elasticsearch/guide/current/match-multi-word.html#match-improving-precision) match 查询时使用 and 操作符， 这样可以让所有词都是必须的。

以下是 match 查询：

{

"match": {

"text": {

"query": "the quick brown fox",

"operator": "and"

}

}

}

上述查询被重写为 bool 查询如下：

{

"bool": {

"must": [

{ "term": { "text": "the" }},

{ "term": { "text": "quick" }},

{ "term": { "text": "brown" }},

{ "term": { "text": "fox" }}

]

}

}

bool 查询会智能的根据较优的顺序依次执行每个 term 查询：它会从最低频的词开始。因为所有词项都必须匹配，只要包含低频词的文档才有可能匹配。使用 and 操作符可以大大提升多词查询的速度。

**最少匹配数(minimum\_should\_match)**

在精度匹配[控制精度](https://www.elastic.co/guide/cn/elasticsearch/guide/current/match-multi-word.html#match-precision)的章节里面，我们讨论过使用 minimum\_should\_match 配置去掉结果中次相关的长尾。虽然它只对这个目的奏效，但是也为我们从侧面带来一个好处，它提供 and 操作符相似的性能。

{

"match": {

"text": {

"query": "the quick brown fox",

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

}

}

}

在上面这个示例中，四分之三的词都必须匹配，这意味着我们只需考虑那些包含最低频或次低频词的文档。 相比默认使用 or 操作符的简单查询，这为我们带来了巨大的性能提升。不过我们有办法可以做得更好……

### 词项的分别管理

在查询字符串中的词项可以分为更重要（低频词）和次重要（高频词）这两类。 只与次重要词项匹配的文档很有可能不太相关。实际上，我们想要文档能尽可能多的匹配那些更重要的词项。

match 查询接受一个参数 cutoff\_frequency ，从而可以让它将查询字符串里的词项分为低频和高频两组。 低频组（更重要的词项）组成 bulk 大量查询条件，而高频组（次重要的词项）只会用来评分，而不参与匹配过程。通过对这两组词的区分处理，我们可以在之前慢查询的基础上获得巨大的速度提升。

领域相关的停用词（Domain-Specific Stopwords）

cutoff\_frequency 配置的好处是，你在 特定领域 使用停用词不受约束。 例如，关于电影网站使用的词 movie 、 color 、 black 和 white ，这些词我们往往认为几乎没有任何意义。使用 stop词汇单元过滤器，这些特定领域的词必须手动添加到停用词列表中。然而 cutoff\_frequency 会查看索引里词项的具体频率，这些词会被自动归类为 高频词汇 。

以下面查询为例：

{

"match": {

"text": {

"query": "Quick and the dead",

"cutoff\_frequency": 0.01 https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/common-terms.html#CO172-1) | 任何词项出现在文档中超过1%，被认为是高频词。cutoff\_frequency 配置可以指定为一个分数（ 0.01 ）或者一个正整数（ 5 ）。 |

此查询通过 cutoff\_frequency 配置，将查询条件划分为低频组（ quick , dead ）和高频组（ and , the）。然后，此查询会被重写为以下的 bool 查询：

{

"bool": {

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

"bool": {

"should": [

{ "term": { "text": "quick" }},

{ "term": { "text": "dead" }}

]

}

},

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

"bool": {

"should": [

{ "term": { "text": "and" }},

{ "term": { "text": "the" }}

]

}

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/common-terms.html#CO173-1) | 必须匹配至少一个低频／更重要的词项。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/common-terms.html#CO173-2) | 高频/次重要性词项是非必须的。 |

must 意味着至少有一个低频词— quick 或者 dead —必须出现在被匹配文档中。所有其他的文档被排除在外。 should 语句查找高频词 and 和 the ，但也只是在 must 语句查询的结果集文档中查询。 should 语句的唯一的工作就是在对如 Quick \_and the\_ dead 和 \_The\_ quick but　dead 语句进行评分时，前者得分比后者高。这种方式可以大大减少需要进行评分计算的文档数量。

提示

将操作符参数设置成 and 会要求所有低频词都必须匹配，同时对包含所有高频词的文档给予更高评分。但是，在匹配文档时，并不要求文档必须包含所有高频词。如果希望文档包含所有的低频和高频词，我们应该使用一个 bool 来替代。正如我们在[and 操作符 (and Operator)](https://www.elastic.co/guide/cn/elasticsearch/guide/current/stopwords-performance.html#stopwords-and)中看到的，它的查询效率已经很高了。

**控制精度**

minimum\_should\_match 参数可以与 cutoff\_frequency 组合使用，但是此参数仅适用与低频词。 如以下查询：

{

"match": {

"text": {

"query": "Quick and the dead",

"cutoff\_frequency": 0.01,

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

}

}

将被重写为如下所示:

{

"bool": {

"must": {

"bool": {

"should": [

{ "term": { "text": "quick" }},

{ "term": { "text": "dead" }}

],

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

}

},

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

"bool": {

"should": [

{ "term": { "text": "and" }},

{ "term": { "text": "the" }}

]

}

}

}

}

<1>　因为只有两个词，原来的75%向下取整为 1 ，意思是：必须匹配低频词的两者之一。 <2>　高频词仍可选的，并且仅用于评分使用。

**高频词**

当使用 or 查询高频词条 ，如— To be, or not to be —进行查询时性能最差。只是为了返回最匹配的前十个结果就对只是包含这些词的所有文档进行评分是盲目的。我们真正的意图是查询整个词条出现的文档，所以在这种情况下，不存低频所言，这个查询需要重写为所有高频词条都必须：

{

"bool": {

"must": [

{ "term": { "text": "to" }},

{ "term": { "text": "be" }},

{ "term": { "text": "or" }},

{ "term": { "text": "not" }},

{ "term": { "text": "to" }},

{ "term": { "text": "be" }}

]

}

}

**对常用词使用更多控制（More Control with Common Terms）**

尽管高频/低频的功能在 match 查询中是有用的，有时我们还希望能对它 有更多的控制，想控制它对高频和低频词分组的行为。　match 查询针对 common 词项查询提供了一组功能。

例如，我们可以让所有低频词都必须匹配，而只对那些包括超过 75% 的高频词文档进行评分：

{

"common": {

"text": {

"query": "Quick and the dead",

"cutoff\_frequency": 0.01,

"low\_freq\_operator": "and",

"minimum\_should\_match": {

"high\_freq": "75%"

}

}

}

}

更多配置项参见　[common terms query](https://www.elastic.co/guide/en/elasticsearch/reference/master/query-dsl-common-terms-query.html)。

### 停用词与短语查询

所有查询中 [短语匹配](https://www.elastic.co/guide/cn/elasticsearch/guide/current/phrase-matching.html) 大约占到5%，但是在慢查询里面它们又占大部分。 短语查询性能相对较差，特别是当短语中包括常用词的时候，如 “To be, or not to be” 短语全部由停用词组成，这是一种极端情况。原因在于几乎需要匹配全量的数据。

在 停用词的两面 [停用词的优缺点](https://www.elastic.co/guide/cn/elasticsearch/guide/current/pros-cons-stopwords.html),中，我们提到移除停用词只能节省倒排索引中的一小部分空间。这句话只部分正确，一个典型的索引会可能包含部分或所有以下数据：

词项字典（Terms dictionary）

索引中所有文档内所有词项的有序列表，以及包含该词的文档数量。

倒排表（Postings list）

包含每个词项的文档（ID）列表。

词频（Term frequency）

每个词项在每个文档里出现的频率。

位置（Positions）

每个词项在每个文档里出现的位置，供短语查询或近似查询使用。

偏移（Offsets）

每个词项在每个文档里开始与结束字符的偏移，供词语高亮使用，默认是禁用的。

规范因子（Norms）

用来对字段长度进行规范化处理的因子，给较短字段予以更多权重。

将停用词从索引中移除会节省 词项字典 和 倒排表 里的少量空间，但 位置 和 偏移 是另一码事。位置和偏移数据很容易变成索引大小的两倍、三倍、甚至四倍。

**位置信息**

analyzed 字符串字段的位置信息默认是开启的， 所以短语查询能随时使用到它。 词项出现的越频繁，用来存储它位置信息的空间就越多。在一个大的文档集合中，对于那些非常常见的词，它们的位置信息可能占用成百上千兆的空间。

运行一个针对高频词 the 的短语查询可能会导致从磁盘读取好几G的数据。这些数据会被存储到内核文件系统的缓存中，以提高后续访问的速度，这看似是件好事，但这可能会导致其他数据从缓存中被剔除，进一步使后续查询变慢。

这显然是我们需要解决的问题。

**索引选项**

我们首先应该问自己：是否真的需要使用短语查询 或 近似查询 ？

答案通常是：不需要。在很多应用场景下，比如说日志，我们需要知道一个词 是否 在文档中（这个信息由倒排表提供）而不是关心词的位置在哪里。或许我们要对一两个字段使用短语查询，但是我们完全可以在其他 analyzed 字符串字段上禁用位置信息。

index\_options 参数 允许我们控制索引里为每个字段存储的信息。 可选值如下:

docs

只存储文档及其包含词项的信息。这对 not\_analyzed 字符串字段是默认的。

freqs

存储 docs 信息，以及每个词在每个文档里出现的频次。词频是完成[TF/IDF](https://www.elastic.co/guide/cn/elasticsearch/guide/current/relevance-intro.html) 相关度计算的必要条件，但如果只想知道一个文档是否包含某个特定词项，则无需使用它。

positions

存储 docs 、 freqs 、 analyzed ，以及每个词项在每个文档里出现的位置。 这对 analyzed 字符串字段是默认的，但当不需使用短语或近似匹配时，可以将其禁用。

offsets

存储 docs,freqs,positions, 以及每个词在原始字符串中开始与结束字符的偏移信息( [postings highlighter](http://www.elastic.co/guide/en/elasticsearch/reference/current/search-request-highlighting.html#postings-highlighter))。这个信息被用以高亮搜索结果，但它默认是禁用的。

我们可以在索引创建的时候为字段设置 index\_options 选项，或者在使用 put-mapping API新增字段映射的时候设置。我们无法修改已有字段的这个设置：

PUT /my\_index

{

"mappings": {

"my\_type": {

"properties": {

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

"type": "string"

},

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

"type": "string",

"index\_options": "freqs"

}

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/stopwords-phrases.html#CO174-1) [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/stopwords-phrases.html#CO174-3) | title 字段使用默认的 positions 设置，所以它适于短语或近似查询。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/stopwords-phrases.html#CO174-2) [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/stopwords-phrases.html#CO174-4) | content 字段的位置设置是禁用的，所以它无法用于短语或近似查询。 |

**停用词**

删除停用词是能显著降低位置信息所占空间的一种方式。 一个被删除停用词的索引仍然可以使用短语查询，因为剩下的词的原始位置仍然被保存着，这正如 [保持位置（Maintaining Positions）](https://www.elastic.co/guide/cn/elasticsearch/guide/current/using-stopwords.html#maintaining-positions) 中看到的那样。 尽管如此，将词项从索引中排除终究会降低搜索能力，这使我们难以区分 Man in the moon 与 Man on the moon 这两个短语。

幸运的是，鱼与熊掌是可以兼得的：请查看 [common\_grams 过滤器](https://www.elastic.co/guide/cn/elasticsearch/guide/current/common-grams.html)。

### common\_grams 过滤器

common\_grams 过滤器是针对短语查询能更高效的使用停用词而设计的。 它与 shingles 过滤器 类似（参见 查找相关词（[寻找相关词](https://www.elastic.co/guide/cn/elasticsearch/guide/current/shingles.html))), 为每个相邻词对生成 ，用示例解释更为容易。

common\_grams 过滤器根据 query\_mode 设置的不同而生成不同输出结果：false （为索引使用） 或 true（为搜索使用），所以我们必须创建两个独立的分析器：

PUT /my\_index

{

"settings": {

"analysis": {

"filter": {

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

"type": "common\_grams",

"common\_words": "\_english\_" https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

},

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

"type": "common\_grams",

"common\_words": "\_english\_", https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/4.png

"query\_mode": true

}

},

"analyzer": {

"index\_grams": { https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/5.png

"tokenizer": "standard",

"filter": [ "lowercase", "index\_filter" ]

},

"search\_grams": { https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/6.png

"tokenizer": "standard",

"filter": [ "lowercase", "search\_filter" ]

}

}

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/common-grams.html#CO175-1) [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/common-grams.html#CO175-3) | 首先我们基于 common\_grams 过滤器创建两个过滤器： index\_filter 在索引时使用（此时 query\_mode 的默认设置是 false ）， search\_filter 在查询时使用（此时 query\_mode 的默认设置是 true ）。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/common-grams.html#CO175-2) [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/4.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/common-grams.html#CO175-4) | common\_words 参数可以接受与 stopwords 参数同样的选项（参见 指定停用词 [指定停用词（Specifying Stopwords）](https://www.elastic.co/guide/cn/elasticsearch/guide/current/using-stopwords.html#specifying-stopwords) ）。这个过滤器还可以接受参数 common\_words\_path ，使用存于文件里的常用词。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/5.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/common-grams.html#CO175-5) [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/6.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/common-grams.html#CO175-6) | 然后我们使用过滤器各创建一个索引时分析器和查询时分析器。 |

有了自定义分析器，我们可以创建一个字段在索引时使用 index\_grams 分析器：

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

{

"properties": {

"text": {

"type": "string",

"analyzer": "index\_grams", https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

"search\_analyzer": "standard" https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/common-grams.html#CO176-1) [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/common-grams.html#CO176-2) | text 字段索引时使用 index\_grams 分析器，但是在搜索时默认使用 standard 分析器，稍后我们会解释其原因。 |

**索引时（At Index Time）**

如果我们对 短语 The quick and brown fox 进行拆分，它生成如下词项：

Pos 1: the\_quick

Pos 2: quick\_and

Pos 3: and\_brown

Pos 4: brown\_fox

新的 index\_grams 分析器生成以下词项：

Pos 1: the, the\_quick

Pos 2: quick, quick\_and

Pos 3: and, and\_brown

Pos 4: brown

Pos 5: fox

所有的词项都是以 unigrams 形式输出的（the、quick 等等），但是如果一个词本身是常用词或者跟随着常用词，那么它同时还会在 unigram 同样的位置以 bigram 形式输出：the\_quick ， quick\_and ， and\_brown 。

**单字查询（Unigram Queries）**

因为索引包含 unigrams ，可以使用与其他字段相同的技术进行查询，例如：

GET /my\_index/\_search

{

"query": {

"match": {

"text": {

"query": "the quick and brown fox",

"cutoff\_frequency": 0.01

}

}

}

}

上面这个查询字符串是通过为文本字段配置的 search\_analyzer 分析器 --本例中使用的是 standard 分析器-- 进行分析的， 它生成的词项为： the ， quick ， and ， brown ， fox 。

因为 text 字段的索引中包含与 standard 分析去生成的一样的 unigrams ，搜索对于任何普通字段都能正常工作。

**二元语法短语查询（Bigram Phrase Queries）**

但是，当我们进行短语查询时， 我们可以用专门的 search\_grams 分析器让整个过程变得更高效：

GET /my\_index/\_search

{

"query": {

"match\_phrase": {

"text": {

"query": "The quick and brown fox",

"analyzer": "search\_grams" https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

}

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/common-grams.html#CO177-1) | 对于短语查询，我们重写了默认的 search\_analyzer 分析器，而使用 search\_grams 分析器。 |

search\_grams 分析器会生成以下词项：

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词的 unigrams`，只留下常用词的 `bigrams 以及低频的 unigrams`。如 `the\_quick这样的 bigrams 比单个词项 the 更为少见，这样有两个好处：

* the\_quick 的位置信息要比 the 的小得多，所以它读取磁盘更快，对系统缓存的影响也更小。
* 词项 the\_quick 没有 the 那么常见，所以它可以大量减少需要计算的文档。

**两词短语（Two-Word Phrases）**

我们的优化可以更进一步， 因为大多数的短语查询只由两个词组成，如果其中一个恰好又是常用词，例如：

GET /my\_index/\_search

{

"query": {

"match\_phrase": {

"text": {

"query": "The quick",

"analyzer": "search\_grams"

}

}

}

}

那么 search\_grams 分析器会输出单个语汇单元：the\_quick 。这将原来昂贵的查询（查询 the 和 quick）转换成了对单个词项的高效查找。

### 停用词与相关性

在结束停用词 相关内容之前，最后一个话题是关于相关性的。在索引中保留停用词会降低相关度计算的准确性，特别是当我们的文档非常长时。

正如我们在 [词频饱和度](https://www.elastic.co/guide/cn/elasticsearch/guide/current/pluggable-similarites.html#bm25-saturation) 已经讨论过的， 原因在于 [词频饱和度](https://www.elastic.co/guide/cn/elasticsearch/guide/current/pluggable-similarites.html#bm25-saturation) 并没有强制对词频率的影响设置上限 。 基于逆文档频率的影响，非常常用的词可能只有很低的权重，但是在长文档中，单个文档出现的绝对数量很大的停用词会导致这些词被不自然的加权。

可以考虑对包含停用词的较长字段使用 [Okapi BM25](https://www.elastic.co/guide/cn/elasticsearch/guide/current/pluggable-similarites.html#bm25) 相似度算法，而不是默认的 Lucene 相似度。

## 同义词

词干提取是通过简化他们的词根形式来扩大搜索的范围，同义词 通过相关的观念和概念来扩大搜索范围。 也许没有文档匹配查询 “英国女王“ ，但是包含 “英国君主” 的文档可能会被认为是很好的匹配。

用户搜索 “美国” 并且期望找到包含 *美利坚合众国* 、 *美国* 、 *美洲* 、或者 *美国各州* 的文档。 然而，他们不希望搜索到关于 国事 或者 政府机构 的结果。

这个例子提供了宝贵的经验，它向我们阐述了，区分不同的概念对于人类是多么简单而对于纯粹的机器是多么棘手的事情。通常我们会对语言中的每一个词去尝试提供同义词以确保任何一个文档都是可发现的，以保证不管文档之间有多么微小的关联性都能够被检索出来。

这样做是不对的。就像我们更喜欢不用或少用词根而不是过分使用词根一样，同义词也应该只在必要的时候使用。 这是因为用户可以理解他们的搜索结果受限于他们的搜索词，如果搜索结果看上去几乎是随机时，他们就会变得无法理解（注：大规模使用同义词会导致查询结果趋向于让人觉得是随机的）。

同义词可以用来合并几乎相同含义的词，如 跳 、 跳越 或者 单脚跳行 ，和 小册子 、 传单 或者 资料手册。 或者，它们可以用来让一个词变得更通用。例如， 鸟 可以作为 猫头鹰 或 鸽子 的通用代名词，还有， 成人 可以被用于 男人 或者 女人 。

同义词似乎是一个简单的概念，但是正确的使用它们却是非常困难的。在这一章，我们会介绍使用同义词的技巧和讨论它的局限性和陷阱。

提示

同义词扩大了一个匹配文件的范围。正如 [词干提取](https://www.elastic.co/guide/cn/elasticsearch/guide/current/stemming.html) 或者 [部分匹配](https://www.elastic.co/guide/cn/elasticsearch/guide/current/partial-matching.html) ，同义词的字段不应该被单独使用，而应该与一个针对主字段的查询操作一起使用，这个主字段应该包含纯净格式的原始文本。 在使用同义词时，参阅 [多数字段](https://www.elastic.co/guide/cn/elasticsearch/guide/current/most-fields.html) 的解释来维护相关性。

### 使用同义词

同义词可以取代现有的语汇单元或 通过使用 [同义词 语汇单元过滤器](https://www.elastic.co/guide/en/elasticsearch/reference/master/analysis-synonym-tokenfilter.html)，添加到语汇单元流中：

PUT /my\_index

{

"settings": {

"analysis": {

"filter": {

"my\_synonym\_filter": {

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

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

"british,english",

"queen,monarch"

]

}

},

"analyzer": {

"my\_synonyms": {

"tokenizer": "standard",

"filter": [

"lowercase",

"my\_synonym\_filter" https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png

]

}

}

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/using-synonyms.html#CO178-1) | 首先，我们定义了一个 同义词 类型的语汇单元过滤器。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/using-synonyms.html#CO178-2) | 我们在 [同义词格式](https://www.elastic.co/guide/cn/elasticsearch/guide/current/synonym-formats.html) 中讨论同义词格式。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/using-synonyms.html#CO178-3) | 然后我们创建了一个使用 my\_synonym\_filter 的自定义分析器。 |

提示

同义词可以使用 synonym 参数来内嵌指定，或者必须 存在于集群每一个节点上的同义词文件中。 同义词文件路径由 synonyms\_path 参数指定，应绝对或相对于 Elasticsearch config目录。参照 [更新停用词（Updating Stopwords）](https://www.elastic.co/guide/cn/elasticsearch/guide/current/using-stopwords.html#updating-stopwords) 的技巧，可以用来刷新的同义词列表。

通过 analyze API 来测试我们的分析器，显示如下：

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

Pos 5: (queen,monarch) https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/using-synonyms.html#CO179-1) [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/using-synonyms.html#CO179-2) | 所有同义词与原始词项占有同一个位置。 |

这样的一个文件将匹配任何以下的查询： English queen 、British queen 、 English monarch 或 British monarch 。 即使是一个短语查询也将会工作，因为每个词项的位置已被保存。

提示

在索引和搜索中使用相同的同义词语汇单元过滤器是多余的。 如果在索引的时候，我们用 english 和 british 这两个术语代替 English ， 然后在搜索的时候，我们只需要搜索这些词项中的一个。或者，如果在索引的时候我们不使用同义词，然后在搜索的时候，我们将需要把对 English 的查询转换为 english 或者 british 的查询。

是否在搜索或索引的时候做同义词扩展可能是一个困难的选择。我们将探索更多的选择 [扩展或收缩](https://www.elastic.co/guide/cn/elasticsearch/guide/current/synonyms-expand-or-contract.html)。

### 同义词格式

同义词最简单的表达形式是 逗号分隔：

"jump,leap,hop"

如果遇到这些词项中的任何一项，则将其替换为所有列出的同义词。例如：

原始词项: 取代:

────────────────────────────────

jump → (jump,leap,hop)

leap → (jump,leap,hop)

hop → (jump,leap,hop)

或者, 使用 => 语法，可以指定一个词项列表（在左边），和一个或多个替换（右边）的列表：

"u s a,united states,united states of america => usa"

"g b,gb,great britain => britain,england,scotland,wales"

原始词项: 取代:

────────────────────────────────

u s a → (usa)

united states → (usa)

great britain → (britain,england,scotland,wales)

如果多个规则指定同一个同义词，它们将被合并在一起，且顺序无关，否则使用最长匹配。以下面的规则为例：

"united states => usa",

"united states of america => usa"

如果这些规则相互冲突，Elasticsearch 会将 United States of America 转换为词项 (usa),(of),(america) 。否则，会使用最长的序列，即最终得到词项 (usa) 。

### 扩展或收缩

在 [同义词格式](https://www.elastic.co/guide/cn/elasticsearch/guide/current/synonym-formats.html) 中，我们看到了 可以通过 简单扩展 、 简单收缩 、或类型扩展 来指明同义词规则。 本章节我们将在这三者间做个权衡比较。

提示

本节仅处理单词同义词。多词同义词又增添了一层复杂性，在 [多词同义词和短语查询](https://www.elastic.co/guide/cn/elasticsearch/guide/current/multi-word-synonyms.html) 中，我们将会讨论。

**简单扩展**

通过 简单扩展 ， 我们可以把同义词列表中的任意一个词扩展成同义词列表 所有 的词：

"jump,hop,leap"

扩展可以应用在索引阶段或查询阶段。两者都有优点 (⬆)︎ 和缺点 (⬇)︎。到底要在哪个阶段使用，则取决于性能与灵活性：

|  | **索引** | **查询** |
| --- | --- | --- |
| **索引的大小** | ⬇︎ 大索引。因为所有的同义词都会被索引，所以索引的大小相对会变大一些。 | ⬆︎ 正常大小。 |
| **关联** | ⬇︎ 所有同义词都有相同的 IDF（至于什么是 IDF ，参见 [什么是相关性?](https://www.elastic.co/guide/cn/elasticsearch/guide/current/relevance-intro.html)），这意味着通用的词和较常用的词都拥有着相同的权重。 | ⬆︎ 每个同义词 IDF 都和原来一样。 |
| **性能** | ⬆︎ 查询只需要找到查询字符串中指定单个词项。 | ⬇︎ 对一个词项的查询重写来查找所有的同义词，从而降低性能。 |
| **灵活性** | ⬇︎ 同义词规则不能改变现有的文件。对于有影响的新规则，现有的文件都要重建（注：重新索引一次文档）。 | ⬆︎ 同义词规则可以更新不需要索引文件。 |

**简单收缩**

简单收缩 ，把 左边的多个同义词映射到了右边的单个词：

"leap,hop => jump"

它必须同时应用于索引和查询阶段，以确保查询词项映射到索引中存在的同一个值。

相对于简单扩展方法，这种方法也有一些优点和一些缺点：

索引的大小

⬆︎ 索引大小是正常的，因为只有单一词项被索引。

关联

⬇︎ 所有词项的 IDF 是一样的，所以你不能区分比较常用的词、不常用的单词。

性能

⬆︎ 查询只需要在索引中找到单词的出现。

灵活性

⬆︎ 新同义词可以添加到规则的左侧并在查询阶段使用。例如，我们想添加 bound 到先前指定的同义词规则中。那么下面的规则将作用于包含 bound 的查询或包含 bound 的文档索引：

"leap,hop,bound => jump"

似乎对旧有的文档不起作用是么？其实我们可以把上面这个同义词规则改写下，以便对旧有文档同样起作用：

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

当你重建索引文件，你可以恢复到上面的规则（注： leap,hop,bound => jump ）来获得查询单个词项的性能优势（注：因为上面那个规则相比这个而言，查询阶段就只要查询一个词了）。

**类型扩展**

类型扩展是完全不同于简单收缩 或扩张， 并不是平等看待所有的同义词，而是扩大了词的意义，使被拓展的词更为通用。以这些规则为例：

"cat => cat,pet",

"kitten => kitten,cat,pet",

"dog => dog,pet"

"puppy => puppy,dog,pet"

通过在索引阶段使用类型扩展：

* 一个关于 kitten 的查询会发现关于 kittens 的文档。
* 查询一个 cat 会找到关于 kittens 和 cats 的文档。
* 一个 pet 的查询将发现有关的 kittens、cats、puppies、dogs 或者 pets 的文档。

或者在查询阶段使用类型扩展， kitten 的查询结果就会被拓展成涉及到 kittens、cats、dogs。

您也可以有两全其美的办法，通过在索引阶段应用类型扩展同义词规则，以确保类型在索引中存在。然后，在查询阶段， 你可以选择不采用同义词（使 kitten 查询只返回 kittens 的文件）或采用同义词， kitten 的查询操作就会返回包括 kittens、cats、pets（也包括 dogs 和 puppies）的相关结果。

前面的示例规则，对 kitten 的 IDF 将是正确的，而 cat 和 pet 的 IDF 将会被 Elasticsearch 降权。然而, 这是对你有利的，当一个针对 kitten 的查询被拓展成了针对 kitten OR cat OR pet 的查询， 那么 kitten 相关的文档就应该排在最上方，其次是 cat 的文件， pet 的文件将被排在最底部。

### 同义词和分析链

在 [同义词格式](https://www.elastic.co/guide/cn/elasticsearch/guide/current/synonym-formats.html) 一章中，我们使用 u s a 来举例阐述一些同义词相关的知识。那么为什么 我们使用的不是 U.S.A. 呢？原因是， 这个 同义词 的语汇单元过滤器只能接收到在它前面的语汇单元过滤器或者分词器的输出结果（这里看不到原始文本）。

假设我们有一个分析器，它由 standard 分词器、 lowercase 的语汇单元过滤器、 synonym 的语汇单元过滤器组成。文本 U.S.A. 的分析过程，看起来像这样的：

original string（原始文本） → "U.S.A."

standard tokenizer（分词器） → (U),(S),(A)

lowercase token filter（语汇单元过滤器） → (u),(s),(a)

synonym token filter（语汇单元过滤器） → (usa)

如果我们有指定的同义词 U.S.A. ，它永远不会匹配任何东西。因为， my\_synonym\_filter 看到词项的时候，句号已经被移除了，并且字母已经被小写了。

这其实是一个非常需要注意的地方。如果我们想同时使用同义词特性与词根提取特性，那么 jumps 、 jumped 、 jump 、 leaps 、 leaped 和 leap 这些词是否都会被索引成一个 jump ？ 我们 可以把同义词过滤器放置在词根提取之前，然后把所有同义词以及词形变化都列举出来：

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

但更简洁的方式将同义词过滤器放置在词根过滤器之后，然后把词根形式的同义词列举出来：

"leap => jump"

**大小写敏感的同义词**

通常，我们把同义词过滤器放置在 lowercase 语汇单元过滤器之后，因此，所有的同义词 都是小写。 但有时会导致奇怪的合并。例如， CAT 扫描和一只 cat 有很大的不同，或者 PET （正电子发射断层扫描）和 pet 。 就此而言，姓 Little 也是不同于形容词 little 的 (尽管当一个句子以它开头时，首字母会被大写)。

如果根据使用情况来区分词义，则需要将同义词过滤器放置在 lowercase 筛选器之前。当然，这意味着同义词规则需要列出所有想匹配的变化（例如， Little、LITTLE、little ）。

相反,可以有两个同义词过滤器：一个匹配大小写敏感的同义词，一个匹配大小写不敏感的同义词。例如，大小写敏感的同义词规则可以是这个样子：

"CAT,CAT scan => cat\_scan"

"PET,PET scan => pet\_scan"

"Johnny Little,J Little => johnny\_little"

"Johnny Small,J Small => johnny\_small"

大小不敏感的同义词规则可以是这个样子：

"cat => cat,pet"

"dog => dog,pet"

"cat scan,cat\_scan scan => cat\_scan"

"pet scan,pet\_scan scan => pet\_scan"

"little,small"

大小写敏感的同义词规则不仅会处理 CAT scan ，而且有时候也可能会匹配到 CAT scan 中的 CAT （注：从而导致 CAT scan 被转化成了同义词 cat\_scan scan ）。出于这个原因，在大小写敏感的同义词列表中会有一个针对较坏替换情况的特异规则 cat\_scan scan 。

提示： 可以看到它们可以多么轻易地变得复杂。同平时一样， analyze API 是帮手，用它来检查分析器是否正确配置。参阅 [测试分析器](https://www.elastic.co/guide/cn/elasticsearch/guide/current/analysis-intro.html#analyze-api)。

### 多词同义词和短语查询

至此，同义词看上去还挺简单的。然而不幸的是，复杂的部分才刚刚开始。 为了能使 [短语查询](https://www.elastic.co/guide/cn/elasticsearch/guide/current/phrase-matching.html) 正常工作， Elasticsearch 需要知道每个词在初始文本中的位置。多词同义词会严重破坏词的位置信息，尤其当新增的同义词标记长度各不相同的时候。

我们创建一个同义词语汇单元过滤器，然后使用下面这样的同义词规则：

"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 会输出下面这样的结果：

Pos 1: (the)

Pos 2: (usa,united,u,united)

Pos 3: (states,s,states)

Pos 4: (is,a,of)

Pos 5: (wealthy,america)

如果你用上面这个同义词语汇单元过滤器索引一个文档，然后执行一个短语查询，那你就会得到惊人的结果，下面这些短语都不会匹配成功：

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

但是这些短语会：

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

如果你是在查询阶段使同义词，那你就会看到更加诡异的匹配结果。看下这个 validate-query 查询：

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

{

"query": {

"match\_phrase": {

"text": {

"query": "usa is wealthy",

"analyzer": "my\_synonyms"

}

}

}

}

查询关键字会被同义词语汇单元过滤器处理成类似这样的信息：

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

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

提示

多词同义 对高亮匹配结果也会造成影响。一个针对 USA 的查询，返回的结果可能却高亮了： The United States is wealthy 。

**使用简单收缩进行短语查询**

避免这种混乱的方法是使用 [简单收缩](https://www.elastic.co/guide/cn/elasticsearch/guide/current/synonyms-expand-or-contract.html#synonyms-contraction)， 用单个词项 表示所有的同义词， 然后在查询阶段，就只需要针对这单个词进行查询了：

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

上面那个查询信息就会被处理成类似下面这样：

Pos 1: (the)

Pos 2: (usa)

Pos 3: (is)

Pos 5: (wealthy)

现在我们再次执行我们之前做过的那个 validate-query 查询，就会输出一个简单又合理的结果：

"usa is wealthy"

这个方法的缺点是，因为把 united states of america 转换成了同义词 usa, 你就不能使用 united states of america 去搜索出 united 或者 states 。 你需要使用一个额外的字段并用另一个解析器链来达到这个目的。

**同义词与 query\_string 查询**

本书很少谈论到 query\_string 查询， 因为真心不推荐你用它。 在 [复杂查询](https://www.elastic.co/guide/cn/elasticsearch/guide/current/search-lite.html#query-string-query) 一节中有提到，由于 query\_string 查询支持一个精简的 查询语法 ，因此，可能这会导致它搜出一些出人意料的结果或者甚至是含有语法错误的结果。

这种查询方式存在不少问题，而其中之一便与多词同义有关。为了支持它的查询语法，你必须用指定的、该语法所能识别的操作符号来标示，比如 AND 、 OR 、 + 、 - 、 field: 等等。 (更多相关内容参阅 [query\_string 语法](https://www.elastic.co/guide/en/elasticsearch/reference/master/query-dsl-query-string-query.html#query-string-syntax) 。)

而在这种语法的解析过程中，解析动作会把查询文本在空格符处作切分，然后分别把每个切分出来的词传递给相关性解析器。 这也即意味着你的同义词解析器永远都不可能收到类似 United States 这样的多个单词组成的同义词。由于不会把 United States 作为一个原子性的文本，所以同义词解析器的输入信息永远都是两个被切分开的词 United 和 States 。

所幸， match 查询相比而言就可靠得多了，因为它不支持上述语法，所以多个字组成的同义词不会被切分开，而是会完整地交给解析器处理。

### 符号同义词

最后一节内容我们来阐述下怎么对符号进行同义词处理，这和我们前面讲的同义词 处理不太一样。 符号同义词 是用别名来表示这个符号，以防止它在分词过程中被误认为是不重要的标点符号而被移除。

虽然绝大多数情况下，符号对于全文搜索而言都无关紧要，但是字符组合而成的表情，或许又会是很有意义的东西，甚至有时候会改变整个句子的含义，对比一下这两句话：

* 我很高兴能在星期天工作。
* 我很高兴能在星期天工作 :( （注：难过的表情）

标准 （注：standard）分词器或许会简单地消除掉第二个句子里的字符表情，致使两个原本意思相去甚远的句子变得相同。

我们可以先使用 [映射](https://www.elastic.co/guide/en/elasticsearch/reference/master/analysis-mapping-charfilter.html)字符过滤器，在文本被递交给分词器处理之前， 把字符表情替换成 符号同义词 emoticon\_happy 或者 emoticon\_sad ：

PUT /my\_index

{

"settings": {

"analysis": {

"char\_filter": {

"emoticons": {

"type": "mapping",

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

":)=>emoticon\_happy",

":(=>emoticon\_sad"

]

}

},

"analyzer": {

"my\_emoticons": {

"char\_filter": "emoticons",

"tokenizer": "standard",

"filter": [ "lowercase" ]

]

}

}

}

}

}

GET /my\_index/\_analyze?analyzer=my\_emoticons

I am :) not :( https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/symbol-synonyms.html#CO180-1) | 映射 过滤器把字符从 => 左边的格式转变成右边的样子。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/symbol-synonyms.html#CO180-2) | 输出： i 、 am 、 emoticon\_happy 、 not 、 emoticon\_sad 。 |

很少有人会搜 emoticon\_happy 这个词，但是确保类似字符表情的这类重要符号被存储到索引中是非常好的做法，在进行情感分析的时候会很有用。当然，我们也可以用真实的词汇来处理符号同义词，比如： happy 或者 sad 。

提示： 映射 字符过滤器是个非常有用的过滤器，它可以用来对一些已有的字词进行替换操作， 你如果想要采用更灵活的正则表达式去替换字词的话，那你可以使用 [pattern\_replace](https://www.elastic.co/guide/en/elasticsearch/reference/master/analysis-pattern-replace-charfilter.html)字符过滤器。

## 拼写错误

我们期望在类似时间和价格的结构化数据上执行一个查询来返回精确匹配的文档。 然而，好的全文检索不应该是完全相同的限定逻辑。 相反，我们可以扩大范围以包括 可能 的匹配，而根据相关性得分将更好的匹配推到结果集的顶部。

事实上，只能完全匹配的全文搜索可能会困扰你的用户。 难道不希望在搜索 quick brown fox 时匹配一个包含 fast brown foxes 的文档， 搜索 Johnny Walker 同时匹配 Johnnie Walker ，搜索 Arnold Shcwarzenneger 同时匹配 Arnold Schwarzenegger ?

如果存在完全符合用户查询的文档，他们应该出现在结果集的顶部，而较弱的匹配可以被包含在列表的后面。 如果没有精确匹配的文档，至少我们可以显示有可能匹配用户要求的文档，它们甚至可能是用户最初想要的！

我们已经在 [归一化词元](https://www.elastic.co/guide/cn/elasticsearch/guide/current/token-normalization.html) 看过自由变音匹配， [将单词还原为词根](https://www.elastic.co/guide/cn/elasticsearch/guide/current/stemming.html) 中的词干， [同义词](https://www.elastic.co/guide/cn/elasticsearch/guide/current/synonyms.html) 中的同义词， 但所有这些方法假定单词拼写正确，或者每个单词拼写只有唯一的方法。

Fuzzy matching 允许查询时匹配错误拼写的单词，而语音语汇单元过滤器可以在索引时用来进行 近似读音匹配。

### 模糊性

模糊匹配 对待 “模糊” 相似的两个词似乎是同一个词。 首先，我们需要对我们所说的 模糊性 进行定义。

在1965年，Vladimir Levenshtein 开发出了 [Levenshtein distance](http://en.wikipedia.org/wiki/Levenshtein_distance)， 用来度量从一个单词转换到另一个单词需要多少次单字符编辑。他提出了三种类型的单字符编辑：

* 一个字符 替换 另一个字符： \_f\_ox → \_b\_ox
* 插入 一个新的字符：sic → sic\_k\_
* 删除 一个字符：b\_l\_ack → back

[Frederick Damerau](http://en.wikipedia.org/wiki/Frederick_J._Damerau) 后来在这些操作基础上做了一个扩展：

* 相邻两个字符的 换位 ： \_st\_ar → \_ts\_ar

举个例子，将单词 bieber 转换成 beaver 需要下面几个步骤：

1. 把 b 替换成 v ：bie\_b\_er → bie\_v\_er
2. 把 i 替换成 a ：b\_i\_ever → b\_a\_ ever
3. 把 e 和 a 进行换位：b\_ae\_ver → b\_ea\_ver

这三个步骤表示 [Damerau-Levenshtein edit distance](https://en.wikipedia.org/wiki/Damerau%E2%80%93Levenshtein_distance) 编辑距离为 3 。

显然，从 beaver 转换成 bieber 是一个很长的过程—他们相距甚远而不能视为一个简单的拼写错误。 Damerau 发现 80% 的拼写错误编辑距离为 1 。换句话说， 80% 的拼写错误可以对原始字符串用 单次编辑 进行修正。

Elasticsearch 指定了 fuzziness 参数支持对最大编辑距离的配置，默认为 ２ 。

当然，单次编辑对字符串的影响取决于字符串的长度。对单词 hat 两次编辑能够产生 mad ， 所以对一个只有 3 个字符长度的字符串允许两次编辑显然太多了。 fuzziness 参数可以被设置为 AUTO ，这将导致以下的最大编辑距离：

* 字符串只有 1 到 2 个字符时是 0
* 字符串有 3 、 4 或者 5 个字符时是 1
* 字符串大于 5 个字符时是 2

当然，你可能会发现编辑距离 2 仍然是太多了，返回的结果似乎并不相关。 把最大 fuzziness 设置为 1，你可以得到更好的结果和更好的性能。

### 模糊查询

[fuzzy 查询](https://www.elastic.co/guide/en/elasticsearch/reference/master/query-dsl-fuzzy-query.html)是 term 查询的模糊等价。 也许你很少直接使用它，但是理解它是如何工作的，可以帮助你在更高级别的 match 查询中使用模糊性。

为了解它是如何运作的，我们首先索引一些文档：

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."}

现在我们可以为词 surprize 运行一个 fuzzy 查询：

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

{

"query": {

"fuzzy": {

"text": "surprize"

}

}

}

fuzzy 查询是一个词项级别的查询，所以它不做任何分析。它通过某个词项以及指定的 fuzziness 查找到词典中所有的词项。 fuzziness 默认设置为 AUTO 。

在我们的例子中， surprise 比较 surprise 和 surprised 都在编辑距离 2 以内， 所以文档 1 和 3 匹配。通过以下查询，我们可以减少匹配度到仅匹配 surprise ：

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

{

"query": {

"fuzzy": {

"text": {

"value": "surprize",

"fuzziness": 1

}

}

}

}

**提高性能**

fuzzy 查询的工作原理是给定原始词项及构造一个 编辑自动机— 像表示所有原始字符串指定编辑距离的字符串的一个大图表。

然后模糊查询使用这个自动机依次高效遍历词典中的所有词项以确定是否匹配。 一旦收集了词典中存在的所有匹配项，就可以计算匹配文档列表。

当然，根据存储在索引中的数据类型，一个编辑距离 2 的模糊查询能够匹配一个非常大数量的词项同时执行效率会非常糟糕。 下面两个参数可以用来限制对性能的影响：

prefix\_length

不能被 “模糊化” 的初始字符数。 大部分的拼写错误发生在词的结尾，而不是词的开始。 例如通过将 prefix\_length 设置为 3 ，你可能够显著降低匹配的词项数量。

max\_expansions

如果一个模糊查询扩展了三个或四个模糊选项， 这些新的模糊选项也许是有意义的。如 果它产生 1000 个模糊选项，那么就基本没有意义了。 设置 max\_expansions 用来限制将产生的模糊选项的总数量。模糊查询将收集匹配词项直到达到 max\_expansions 的限制。

### 模糊匹配查询

match 查询支持 开箱即用的模糊匹配：

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

{

"query": {

"match": {

"text": {

"query": "SURPRIZE ME!",

"fuzziness": "AUTO",

"operator": "and"

}

}

}

}

查询字符串首先进行分析，会产生词项 [surprize, me] ，并且每个词项根据指定的 fuzziness 进行模糊化。

同样， multi\_match 查询也 支持 fuzziness ，但只有当执行查询时类型是 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 查询都支持 prefix\_length 和 max\_expansions 参数。

提示

模糊性（Fuzziness）只能在 match and multi\_match 查询中使用。不能使用在短语匹配、常用词项或 cross\_fields 匹配。

### 模糊性评分

用户喜欢模糊查询。他们认为这种查询会魔法般的找到正确拼写组合。 很遗憾，实际效果平平。

假设我们有1000个文档包含 ``Schwarzenegger`` ，只是一个文档的出现拼写错误 ``Schwarzeneger`` 。 根据 [term frequency/inverse document frequency](https://www.elastic.co/guide/cn/elasticsearch/guide/current/scoring-theory.html#tfidf) 理论，这个拼写错误文档比拼写正确的相关度更高，因为错误拼写出现在更少的文档中！

换句话说，如果我们对待模糊匹配 类似其他匹配方法，我们将偏爱错误的拼写超过了正确的拼写，这会让用户抓狂。

提示

模糊匹配不应用于参与评分--只能在有拼写错误时扩大匹配项的范围。

默认情况下， match 查询给定所有的模糊匹配的恒定评分为1。这可以满足在结果列表的末尾添加潜在的匹配记录，并且没有干扰非模糊查询的相关性评分。

提示

在模糊查询最初出现时很少能单独使用。他们更好的作为一个 ``bigger`` 场景的部分功能特性，如 search-as-you-type [完成 建议](https://www.elastic.co/guide/en/elasticsearch/reference/master/search-suggesters-completion.html)或 did-you-mean [短语 建议](https://www.elastic.co/guide/en/elasticsearch/reference/master/search-suggesters-phrase.html)。

### 语音匹配

最后，在尝试任何其他匹配方法都无效后，我们可以求助于搜索发音相似的词， 即使他们的拼写不同。

有一些用于将词转换成语音标识的算法。 [Soundex](http://en.wikipedia.org/wiki/Soundex) 算法是这些算法的鼻祖， 而且大多数语音算法是 Soundex 的改进或者专业版本，例如 [Metaphone](http://en.wikipedia.org/wiki/Metaphone) 和 [Double Metaphone](http://en.wikipedia.org/wiki/Metaphone#Double_Metaphone) （扩展了除英语以外的其他语言的语音匹配）， [Caverphone](http://en.wikipedia.org/wiki/Caverphone) 算法匹配了新西兰的名称， [Beider-Morse](https://en.wikipedia.org/wiki/Daitch%E2%80%93Mokotoff_Soundex#Beider.E2.80.93Morse_Phonetic_Name_Matching_Algorithm) 算法吸收了 Soundex 算法为了更好的匹配德语和依地语名称， [Kölner Phonetik](http://de.wikipedia.org/wiki/K%C3%B6lner_Phonetik) 为了更好的处理德语词汇。

值得一提的是，语音算法是相当简陋的， 他们设计初衷针对的语言通常是英语或德语。这限制了他们的实用性。 不过，为了某些明确的目标，并与其他技术相结合，语音匹配能够作为一个有用的工具。

首先，你需要从 <https://www.elastic.co/guide/en/elasticsearch/plugins/current/analysis-phonetic.html>获取语音分析插件并在集群的每个节点安装， 然后重启每个节点。

然后，您可以创建一个使用语音语汇单元过滤器的自定义分析器，并尝试下面的方法：

PUT /my\_index

{

"settings": {

"analysis": {

"filter": {

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

"type": "phonetic",

"encoder": "double\_metaphone"

}

},

"analyzer": {

"dbl\_metaphone": {

"tokenizer": "standard",

"filter": "dbl\_metaphone" https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

}

}

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/phonetic-matching.html#CO181-1) | 首先，配置一个自定义 phonetic 语汇单元过滤器并使用 double\_metaphone 编码器。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/phonetic-matching.html#CO181-2) | 然后在自定义分析器中使用自定义语汇单元过滤器。 |

现在我们可以通过 analyze API 来进行测试：

GET /my\_index/\_analyze?analyzer=dbl\_metaphone

Smith Smythe

每个 Smith 和 Smythe 在同一位置产生两个语汇单元： SM0 和 XMT 。 通过分析器播放 John ， Jon 和 Johnnie 将产生两个语汇单元 JN 和 AN ，而 Jonathon 产生语汇单元 JN0N 和 ANTN 。

语音分析器可以像任何其他分析器一样使用。 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": { https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

"type": "string",

"analyzer": "dbl\_metaphone"

}

}

}

}

}

PUT /my\_index/my\_type/1

{

"name": "John Smith"

}

PUT /my\_index/my\_type/2

{

"name": "Jonnie Smythe"

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/phonetic-matching.html#CO182-1) | name.phonetic 字段使用自定义 dbl\_metaphone 分析器。 |

可以使用 match 查询来进行搜索：

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

{

"query": {

"match": {

"name.phonetic": {

"query": "Jahnnie Smeeth",

"operator": "and"

}

}

}

}

这个查询返回全部两个文档，演示了如何进行简陋的语音匹配。 用语音算法计算评分是没有价值的。 语音匹配的目的不是为了提高精度，而是要提高召回率--以扩展足够的范围来捕获可能匹配的文档。

通常更有意义的使用语音算法是在检索到结果后，由另一台计算机进行消费和后续处理，而不是由人类用户直接使用。

# 聚合

在这之前，本书致力于搜索。 通过搜索，如果我们有一个查询并且希望找到匹配这个查询的文档集，就好比在大海捞针。

通过聚合，我们会得到一个数据的概览。我们需要的是分析和总结全套的数据而不是寻找单个文档：

* 在大海里有多少针？
* 针的平均长度是多少？
* 按照针的制造商来划分，针的长度中位值是多少？
* 每月加入到海中的针有多少？

聚合也可以回答更加细微的问题：

* 你最受欢迎的针的制造商是什么？
* 这里面有异常的针么？

聚合允许我们向数据提出一些复杂的问题。虽然功能完全不同于搜索，但它使用相同的数据结构。这意味着聚合的执行速度很快并且就像搜索一样几乎是实时的。

这对报告和仪表盘是非常强大的。你可以实时显示你的数据，让你立即回应，而不是对你的数据进行汇总（ 需要一周时间去运行的 Hadoop 任务 ），您的报告随着你的数据变化而变化，而不是预先计算的、过时的和不相关的。

最后，聚合和搜索是一起的。 这意味着你可以在单个请求里同时对相同的数据进行搜索/过滤和分析。并且由于聚合是在用户搜索的上下文里计算的，你不只是显示四星酒店的数量，而是显示匹配查询条件的四星酒店的数量。

聚合是如此强大以至于许多公司已经专门为数据分析建立了大型 Elasticsearch 集群。

## 高阶概念

类似于 DSL 查询表达式， 聚合也有 可组合 的语法：独立单元的功能可以被混合起来提供你需要的自定义行为。这意味着只需要学习很少的基本概念，就可以得到几乎无尽的组合。

要掌握聚合，你只需要明白两个主要的概念：

桶（Buckets）

满足特定条件的文档的集合

指标（Metrics）

对桶内的文档进行统计计算

这就是全部了！每个聚合都是一个或者多个桶和零个或者多个指标的组合。翻译成粗略的SQL语句来解释吧：

SELECT COUNT(color) https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

FROM table

GROUP BY color https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/aggs-high-level.html#CO183-1) | COUNT(color) 相当于指标。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/aggs-high-level.html#CO183-2) | GROUP BY color 相当于桶。 |

桶在概念上类似于 SQL 的分组（GROUP BY），而指标则类似于 COUNT() 、 SUM() 、 MAX() 等统计方法。

让我们深入这两个概念 并且了解和这两个概念相关的东西。

### 桶

桶 简单来说就是满足特定条件的文档的集合：

* 一个雇员属于 男性 桶或者 女性 桶
* 奥尔巴尼属于 纽约 桶
* 日期2014-10-28属于 十月 桶

当聚合开始被执行，每个文档里面的值通过计算来决定符合哪个桶的条件。如果匹配到，文档将放入相应的桶并接着进行聚合操作。

桶也可以被嵌套在其他桶里面，提供层次化的或者有条件的划分方案。例如，辛辛那提会被放入俄亥俄州这个桶，而 整个 俄亥俄州桶会被放入美国这个桶。

Elasticsearch 有很多种类型的桶，能让你通过很多种方式来划分文档（时间、最受欢迎的词、年龄区间、地理位置等等）。其实根本上都是通过同样的原理进行操作：基于条件来划分文档。

### 指标

桶能让我们划分文档到有意义的集合， 但是最终我们需要的是对这些桶内的文档进行一些指标的计算。分桶是一种达到目的的手段：它提供了一种给文档分组的方法来让我们可以计算感兴趣的指标。

大多数 指标 是简单的数学运算（例如最小值、平均值、最大值，还有汇总），这些是通过文档的值来计算。在实践中，指标能让你计算像平均薪资、最高出售价格、95%的查询延迟这样的数据。

### 桶和指标的组合

聚合 是由桶和指标组成的。 聚合可能只有一个桶，可能只有一个指标，或者可能两个都有。也有可能有一些桶嵌套在其他桶里面。例如，我们可以通过所属国家来划分文档（桶），然后计算每个国家的平均薪酬（指标）。

由于桶可以被嵌套，我们可以实现非常多并且非常复杂的聚合：

1.通过国家划分文档（桶）

2.然后通过性别划分每个国家（桶）

3.然后通过年龄区间划分每种性别（桶）

4.最后，为每个年龄区间计算平均薪酬（指标）

最后将告诉你每个 <国家, 性别, 年龄> 组合的平均薪酬。所有的这些都在一个请求内完成并且只遍历一次数据！

## 尝试聚合

我们可以用以下几页定义不同的聚合和它们的语法， 但学习聚合的最佳途径就是用实例来说明。 一旦我们获得了聚合的思想，以及如何合理地嵌套使用它们，那么语法就变得不那么重要了。

注意

聚合的桶操作和度量的完整用法可以在 [Elasticsearch 参考](https://www.elastic.co/guide/en/elasticsearch/reference/master/search-aggregations.html) 中找到。本章中会涵盖其中很多内容，但在阅读完本章后查看它会有助于我们对它的整体能力有所了解。

所以让我们先看一个例子。我们将会创建一些对汽车经销商有用的聚合，数据是关于汽车交易的信息：车型、制造商、售价、何时被出售等。

首先我们批量索引一些数据：

POST /cars/transactions/\_bulk

{ "index": {}}

{ "price" : 10000, "color" : "red", "make" : "honda", "sold" : "2014-10-28" }

{ "index": {}}

{ "price" : 20000, "color" : "red", "make" : "honda", "sold" : "2014-11-05" }

{ "index": {}}

{ "price" : 30000, "color" : "green", "make" : "ford", "sold" : "2014-05-18" }

{ "index": {}}

{ "price" : 15000, "color" : "blue", "make" : "toyota", "sold" : "2014-07-02" }

{ "index": {}}

{ "price" : 12000, "color" : "green", "make" : "toyota", "sold" : "2014-08-19" }

{ "index": {}}

{ "price" : 20000, "color" : "red", "make" : "honda", "sold" : "2014-11-05" }

{ "index": {}}

{ "price" : 80000, "color" : "red", "make" : "bmw", "sold" : "2014-01-01" }

{ "index": {}}

{ "price" : 25000, "color" : "blue", "make" : "ford", "sold" : "2014-02-12" }

有了数据，开始构建我们的第一个聚合。汽车经销商可能会想知道哪个颜色的汽车销量最好，用聚合可以轻易得到结果，用 terms 桶操作：

GET /cars/transactions/\_search

{

"size" : 0,

"aggs" : { https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

"popular\_colors" : { https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

"terms" : { https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png

"field" : "color"

}

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_aggregation_test_drive.html#CO184-1) | 聚合操作被置于顶层参数 aggs 之下（如果你愿意，完整形式 aggregations 同样有效）。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_aggregation_test_drive.html#CO184-2) | 然后，可以为聚合指定一个我们想要名称，本例中是： popular\_colors 。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_aggregation_test_drive.html#CO184-3) | 最后，定义单个桶的类型 terms 。 |

聚合是在特定搜索结果背景下执行的， 这也就是说它只是查询请求的另外一个顶层参数（例如，使用 /\_search 端点）。 聚合可以与查询结对，但我们会晚些在 [限定聚合的范围（Scoping Aggregations）](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_scoping_aggregations.html) 中来解决这个问题。

注意

可能会注意到我们将 size 设置成 0 。我们并不关心搜索结果的具体内容，所以将返回记录数设置为 0 来提高查询速度。 设置 size: 0 与 Elasticsearch 1.x 中使用 count 搜索类型等价。

然后我们为聚合定义一个名字，名字的选择取决于使用者，响应的结果会以我们定义的名字为标签，这样应用就可以解析得到的结果。

随后我们定义聚合本身，在本例中，我们定义了一个单 terms 桶。 这个 terms 桶会为每个碰到的唯一词项动态创建新的桶。 因为我们告诉它使用 color 字段，所以 terms 桶会为每个颜色动态创建新桶。

让我们运行聚合并查看结果：

{

...

"hits": {

"hits": [] https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

},

"aggregations": {

"popular\_colors": { https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

"buckets": [

{

"key": "red", https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png

"doc\_count": 4 https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/4.png

},

{

"key": "blue",

"doc\_count": 2

},

{

"key": "green",

"doc\_count": 2

}

]

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_aggregation_test_drive.html#CO185-1) | 因为我们设置了 size 参数，所以不会有 hits 搜索结果返回。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_aggregation_test_drive.html#CO185-2) | popular\_colors 聚合是作为 aggregations 字段的一部分被返回的。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_aggregation_test_drive.html#CO185-3) | 每个桶的 key 都与 color 字段里找到的唯一词对应。它总会包含 doc\_count 字段，告诉我们包含该词项的文档数量。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/4.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_aggregation_test_drive.html#CO185-4) | 每个桶的数量代表该颜色的文档数量。 |

响应包含多个桶，每个对应一个唯一颜色（例如：红 或 绿）。每个桶也包括 聚合进 该桶的所有文档的数量。例如，有四辆红色的车。

前面的这个例子完全是实时执行的：一旦文档可以被搜到，它就能被聚合。这也就意味着我们可以直接将聚合的结果源源不断的传入图形库，然后生成实时的仪表盘。 不久，你又销售了一辆银色的车，我们的图形就会立即动态更新银色车的统计信息。

瞧！这就是我们的第一个聚合！

### 添加度量指标

前面的例子告诉我们每个桶里面的文档数量，这很有用。 但通常，我们的应用需要提供更复杂的文档度量。 例如，每种颜色汽车的平均价格是多少？

为了获取更多信息，我们需要告诉 Elasticsearch 使用哪个字段，计算何种度量。 这需要将度量 嵌套 在桶内， 度量会基于桶内的文档计算统计结果。

让我们继续为汽车的例子加入 average 平均度量：

GET /cars/transactions/\_search

{

"size" : 0,

"aggs": {

"colors": {

"terms": {

"field": "color"

},

"aggs": { https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

"avg\_price": { https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

"avg": {

"field": "price" https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png

}

}

}

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_adding_a_metric_to_the_mix.html#CO186-1) | 为度量新增 aggs 层。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_adding_a_metric_to_the_mix.html#CO186-2) | 为度量指定名字： avg\_price 。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_adding_a_metric_to_the_mix.html#CO186-3) | 最后，为 price 字段定义 avg 度量。 |

正如所见，我们用前面的例子加入了新的 aggs 层。这个新的聚合层让我们可以将 avg 度量嵌套置于 terms 桶内。实际上，这就为每个颜色生成了平均价格。

正如 颜色 的例子，我们需要给度量起一个名字（ avg\_price ）这样可以稍后根据名字获取它的值。最后，我们指定度量本身（ avg ）以及我们想要计算平均值的字段（ price ）：

{

...

"aggregations": {

"colors": {

"buckets": [

{

"key": "red",

"doc\_count": 4,

"avg\_price": { https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

"value": 32500

}

},

{

"key": "blue",

"doc\_count": 2,

"avg\_price": {

"value": 20000

}

},

{

"key": "green",

"doc\_count": 2,

"avg\_price": {

"value": 21000

}

}

]

}

}

...

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_adding_a_metric_to_the_mix.html#CO187-1) | 响应中的新字段 avg\_price 。 |

尽管响应只发生很小改变，实际上我们获得的数据是增长了。之前，我们知道有四辆红色的车，现在，红色车的平均价格是 $32，500 美元。这个信息可以直接显示在报表或者图形中。

### 嵌套桶

在我们使用不同的嵌套方案时，聚合的力量才能真正得以显现。 在前例中，我们以及看到如何将一个度量嵌入桶中，它的功能已经十分强大了。

但真正令人激动的分析来自于将桶嵌套进 另外一个桶 所能得到的结果。 现在，我们想知道每个颜色的汽车制造商的分布：

GET /cars/transactions/\_search

{

"size" : 0,

"aggs": {

"colors": {

"terms": {

"field": "color"

},

"aggs": {

"avg\_price": { https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

"avg": {

"field": "price"

}

},

"make": { https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

"terms": {

"field": "make" https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png

}

}

}

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_buckets_inside_buckets.html#CO188-1) | 注意前例中的 avg\_price 度量仍然保持原位。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_buckets_inside_buckets.html#CO188-2) | 另一个聚合 make 被加入到了 color 颜色桶中。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_buckets_inside_buckets.html#CO188-3) | 这个聚合是 terms 桶，它会为每个汽车制造商生成唯一的桶。 |

这里发生了一些有趣的事。 首先，我们可能会观察到之前例子中的 avg\_price 度量完全没有变化，还在原来的位置。 一个聚合的每个 层级 都可以有多个度量或桶， avg\_price 度量告诉我们每种颜色汽车的平均价格。它与其他的桶和度量相互独立。

这对我们的应用非常重要，因为这里面有很多相互关联，但又完全不同的度量需要收集。聚合使我们能够用一次数据请求获得所有的这些信息。

另外一件值得注意的重要事情是我们新增的这个 make 聚合，它是一个 terms 桶（嵌套在 colors 、 terms 桶内）。这意味着它 会为数据集中的每个唯一组合生成（ color 、 make ）元组。

让我们看看返回的响应（为了简单我们只显示部分结果）：

{

...

"aggregations": {

"colors": {

"buckets": [

{

"key": "red",

"doc\_count": 4,

"make": { https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

"buckets": [

{

"key": "honda", https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

"doc\_count": 3

},

{

"key": "bmw",

"doc\_count": 1

}

]

},

"avg\_price": {

"value": 32500 https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png

}

},

...

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_buckets_inside_buckets.html#CO189-1) | 正如期望的那样，新的聚合嵌入在每个颜色桶中。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_buckets_inside_buckets.html#CO189-2) | 现在我们看见按不同制造商分解的每种颜色下车辆信息。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_buckets_inside_buckets.html#CO189-3) | 最终，我们看到前例中的 avg\_price 度量仍然维持不变。 |

响应结果告诉我们以下几点：

* 红色车有四辆。
* 红色车的平均售价是 $32，500 美元。
* 其中三辆是 Honda 本田制造，一辆是 BMW 宝马制造。

### 最后的修改

让我们回到话题的原点，在进入新话题之前，对我们的示例做最后一个修改， 为每个汽车生成商计算最低和最高的价格：

GET /cars/transactions/\_search

{

"size" : 0,

"aggs": {

"colors": {

"terms": {

"field": "color"

},

"aggs": {

"avg\_price": { "avg": { "field": "price" }

},

"make" : {

"terms" : {

"field" : "make"

},

"aggs" : { https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

"min\_price" : { "min": { "field": "price"} }, https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

"max\_price" : { "max": { "field": "price"} } https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png

}

}

}

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_one_final_modification.html#CO190-1) | 我们需要增加另外一个嵌套的 aggs 层级。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_one_final_modification.html#CO190-2) | 然后包括 min 最小度量。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_one_final_modification.html#CO190-3) | 以及 max 最大度量。 |

得到以下输出（只显示部分结果）：

{

...

"aggregations": {

"colors": {

"buckets": [

{

"key": "red",

"doc\_count": 4,

"make": {

"buckets": [

{

"key": "honda",

"doc\_count": 3,

"min\_price": {

"value": 10000 https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

},

"max\_price": {

"value": 20000 https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

}

},

{

"key": "bmw",

"doc\_count": 1,

"min\_price": {

"value": 80000

},

"max\_price": {

"value": 80000

}

}

]

},

"avg\_price": {

"value": 32500

}

},

...

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_one_final_modification.html#CO191-1) [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_one_final_modification.html#CO191-2) | min 和 max 度量现在出现在每个汽车制造商（ make ）下面。 |

有了这两个桶，我们可以对查询的结果进行扩展并得到以下信息：

* 有四辆红色车。
* 红色车的平均售价是 $32，500 美元。
* 其中三辆红色车是 Honda 本田制造，一辆是 BMW 宝马制造。
* 最便宜的红色本田售价为 $10，000 美元。
* 最贵的红色本田售价为 $20，000 美元。

## 条形图

聚合还有一个令人激动的特性就是能够十分容易地将它们转换成图表和图形。 本章中， 我们正在通过示例数据来来完成各种各样的聚合分析，最终，我们将会发现聚合功能是非常强大的。

直方图 histogram 特别有用。 它本质上是一个条形图，如果有创建报表或分析仪表盘的经验，那么我们会毫无疑问的发现里面有一些图表是条形图。 创建直方图需要指定一个区间，如果我们要为售价创建一个直方图，可以将间隔设为 20,000。这样做将会在每个 $20,000 档创建一个新桶，然后文档会被分到对应的桶中。

对于仪表盘来说，我们希望知道每个售价区间内汽车的销量。我们还会想知道每个售价区间内汽车所带来的收入，可以通过对每个区间内已售汽车的售价求和得到。

可以用 histogram 和一个嵌套的 sum 度量得到我们想要的答案：

GET /cars/transactions/\_search

{

"size" : 0,

"aggs":{

"price":{

"histogram":{ https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

"field": "price",

"interval": 20000

},

"aggs":{

"revenue": {

"sum": { https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

"field" : "price"

}

}

}

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_building_bar_charts.html#CO192-1) | histogram 桶要求两个参数：一个数值字段以及一个定义桶大小间隔。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_building_bar_charts.html#CO192-2) | sum 度量嵌套在每个售价区间内，用来显示每个区间内的总收入。 |

如我们所见，查询是围绕 price 聚合构建的，它包含一个 histogram 桶。它要求字段的类型必须是数值型的同时需要设定分组的间隔范围。 间隔设置为 20,000 意味着我们将会得到如 [0-19999, 20000-39999, ...] 这样的区间。

接着，我们在直方图内定义嵌套的度量，这个 sum 度量，它会对落入某一具体售价区间的文档中 price 字段的值进行求和。 这可以为我们提供每个售价区间的收入，从而可以发现到底是普通家用车赚钱还是奢侈车赚钱。

响应结果如下：

{

...

"aggregations": {

"price": {

"buckets": [

{

"key": 0,

"doc\_count": 3,

"revenue": {

"value": 37000

}

},

{

"key": 20000,

"doc\_count": 4,

"revenue": {

"value": 95000

}

},

{

"key": 80000,

"doc\_count": 1,

"revenue": {

"value": 80000

}

}

]

}

}

}

结果很容易理解，不过应该注意到直方图的键值是区间的下限。键 0 代表区间 0-19，999 ，键 20000 代表区间 20，000-39，999 ，等等。

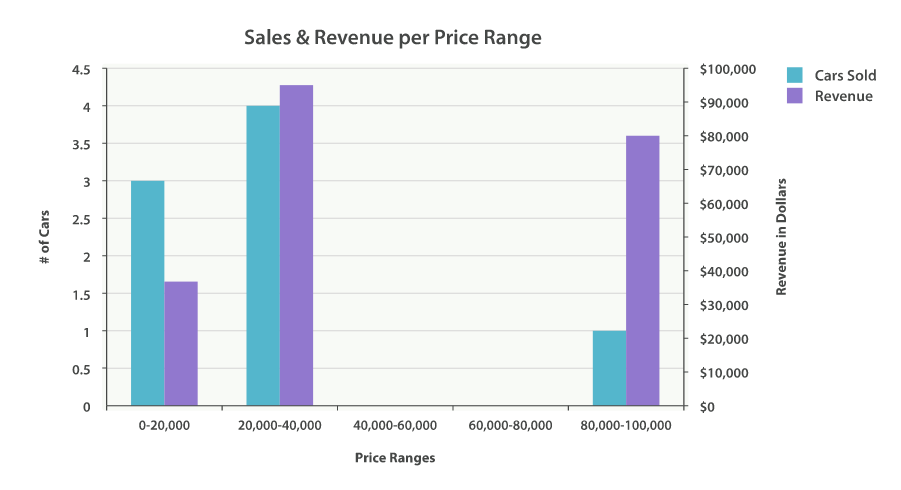
注意

我们可能会注意到空的区间，比如：$40，000-60，000，没有出现在响应中。 histogram桶默认会忽略它，因为它有可能会导致不希望的潜在错误输出。

我们会在下一小节中讨论如何包括空桶。返回空桶 [返回空 Buckets](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_returning_empty_buckets.html) 。

可以在图 [图 35 “Sales and Revenue per price bracket”](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_building_bar_charts.html#barcharts-histo1) 中看到以上数据直方图的图形化表示。

**图 35. Sales and Revenue per price bracket**



当然，我们可以为任何聚合输出的分类和统计结果创建条形图，而不只是 直方图 桶。让我们以最受欢迎 10 种汽车以及它们的平均售价、标准差这些信息创建一个条形图。 我们会用到 terms 桶和 extended\_stats 度量：

GET /cars/transactions/\_search

{

"size" : 0,

"aggs": {

"makes": {

"terms": {

"field": "make",

"size": 10

},

"aggs": {

"stats": {

"extended\_stats": {

"field": "price"

}

}

}

}

}

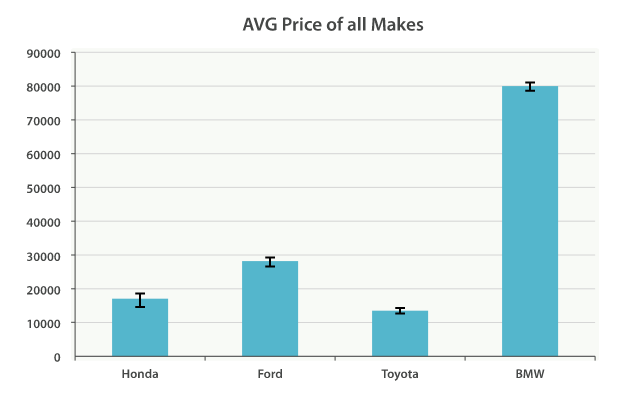
}

上述代码会按受欢迎度返回制造商列表以及它们各自的统计信息。我们对其中的 stats.avg 、 stats.count 和 stats.std\_deviation 信息特别感兴趣，并用 它们计算出标准差：

std\_err = std\_deviation / count

创建图表如图 [图 36 “Average price of all makes, with error bars”](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_building_bar_charts.html#barcharts-bar1) 。

**图 36. Average price of all makes, with error bars**



## 按时间统计

如果搜索是在 Elasticsearch 中使用频率最高的，那么构建按时间统计的 date\_histogram 紧随其后。 为什么你会想用 date\_histogram 呢？

假设你的数据带时间戳。 无论是什么数据（Apache 事件日志、股票买卖交易时间、棒球运动时间）只要带有时间戳都可以进行 date\_histogram 分析。当你的数据有时间戳，你总是想在 时间 维度上构建指标分析：

* 今年每月销售多少台汽车？
* 这只股票最近 12 小时的价格是多少？
* 我们网站上周每小时的平均响应延迟时间是多少？

虽然通常的 histogram 都是条形图，但 date\_histogram 倾向于转换成线状图以展示时间序列。 许多公司用 Elasticsearch \_仅仅\_ 只是为了分析时间序列数据。 date\_histogram 分析是它们最基本的需要。

date\_histogram 与 通常的 histogram 类似。 但不是在代表数值范围的数值字段上构建 buckets，而是在时间范围上构建 buckets。 因此每一个 bucket 都被定义成一个特定的日期大小 (比如， 1个月 或 2.5 天)。

**可以用通常的 histogram 进行时间分析吗？**

从技术上来讲，是可以的。 通常的 histogram bucket（桶）是可以处理日期的。 但是它不能自动识别日期。 而用 date\_histogram ，你可以指定时间段如 1 个月 ，它能聪明地知道 2 月的天数比 12 月少。 date\_histogram 还具有另外一个优势，即能合理地处理时区，这可以使你用客户端的时区进行图标定制，而不是用服务器端时区。

通常的 histogram 会把日期看做是数字，这意味着你必须以微秒为单位指明时间间隔。另外聚合并不知道日历时间间隔，使得它对于日期而言几乎没什么用处。

我们的第一个例子将构建一个简单的折线图来回答如下问题： 每月销售多少台汽车？

GET /cars/transactions/\_search

{

"size" : 0,

"aggs": {

"sales": {

"date\_histogram": {

"field": "sold",

"interval": "month", https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

"format": "yyyy-MM-dd" https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

}

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_looking_at_time.html#CO193-1) | 时间间隔要求是日历术语 (如每个 bucket 1 个月)。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_looking_at_time.html#CO193-2) | 我们提供日期格式以便 buckets 的键值便于阅读。 |

我们的查询只有一个聚合，每月构建一个 bucket。这样我们可以得到每个月销售的汽车数量。 另外还提供了一个额外的 format 参数以便 buckets 有 "好看的" 键值。 然而在内部，日期仍然是被简单表示成数值。这可能会使得 UI 设计者抱怨，因此可以提供常用的日期格式进行格式化以更方便阅读。

结果既符合预期又有一点出人意料（看看你是否能找到意外之处）：

{

...

"aggregations": {

"sales": {

"buckets": [

{

"key\_as\_string": "2014-01-01",

"key": 1388534400000,

"doc\_count": 1

},

{

"key\_as\_string": "2014-02-01",

"key": 1391212800000,

"doc\_count": 1

},

{

"key\_as\_string": "2014-05-01",

"key": 1398902400000,

"doc\_count": 1

},

{

"key\_as\_string": "2014-07-01",

"key": 1404172800000,

"doc\_count": 1

},

{

"key\_as\_string": "2014-08-01",

"key": 1406851200000,

"doc\_count": 1

},

{

"key\_as\_string": "2014-10-01",

"key": 1412121600000,

"doc\_count": 1

},

{

"key\_as\_string": "2014-11-01",

"key": 1414800000000,

"doc\_count": 2

}

]

...

}

聚合结果已经完全展示了。正如你所见，我们有代表月份的 buckets，每个月的文档数目，以及美化后的 key\_as\_string 。

### 返回空 Buckets

注意到结果末尾处的奇怪之处了吗？

是的，结果没错。 我们的结果少了一些月份！ date\_histogram （和 histogram 一样）默认只会返回文档数目非零的 buckets。

这意味着你的 histogram 总是返回最少结果。通常，你并不想要这样。对于很多应用，你可能想直接把结果导入到图形库中，而不想做任何后期加工。

事实上，即使 buckets 中没有文档我们也想返回。可以通过设置两个额外参数来实现这种效果：

GET /cars/transactions/\_search

{

"size" : 0,

"aggs": {

"sales": {

"date\_histogram": {

"field": "sold",

"interval": "month",

"format": "yyyy-MM-dd",

"min\_doc\_count" : 0, https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

"extended\_bounds" : { https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

"min" : "2014-01-01",

"max" : "2014-12-31"

}

}

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_returning_empty_buckets.html#CO194-1) | 这个参数强制返回空 buckets。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_returning_empty_buckets.html#CO194-2) | 这个参数强制返回整年。 |

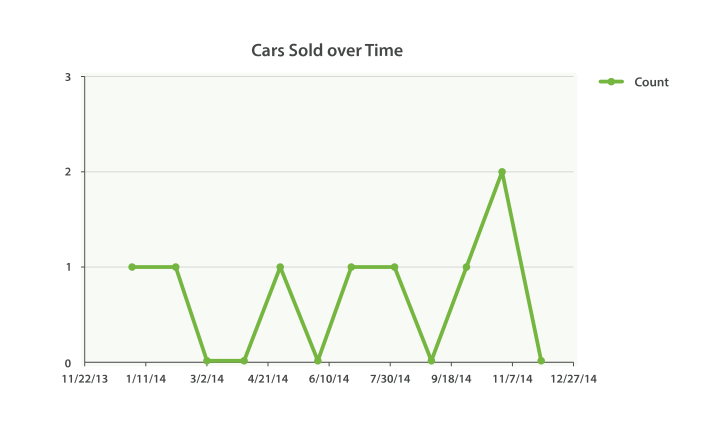
这两个参数会强制返回一年中所有月份的结果，而不考虑结果中的文档数目。 min\_doc\_count 非常容易理解：它强制返回所有 buckets，即使 buckets 可能为空。

extended\_bounds 参数需要一点解释。 min\_doc\_count 参数强制返回空 buckets，但是 Elasticsearch 默认只返回你的数据中最小值和最大值之间的 buckets。

因此如果你的数据只落在了 4 月和 7 月之间，那么你只能得到这些月份的 buckets（可能为空也可能不为空）。因此为了得到全年数据，我们需要告诉 Elasticsearch 我们想要全部 buckets， 即便那些 buckets 可能落在最小日期 之前 或 最大日期 之后 。

extended\_bounds 参数正是如此。一旦你加上了这两个设置，你可以把得到的结果轻易地直接插入到你的图形库中，从而得到类似 [图 37 “汽车销售时间图”](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_returning_empty_buckets.html#date-histo-ts1) 的图表。

**图 37. 汽车销售时间图**



### 扩展例子

正如我们已经见过很多次，buckets 可以嵌套进 buckets 中从而得到更复杂的分析。 作为例子，我们构建聚合以便按季度展示所有汽车品牌总销售额。同时按季度、按每个汽车品牌计算销售总额，以便可以找出哪种品牌最赚钱：

GET /cars/transactions/\_search

{

"size" : 0,

"aggs": {

"sales": {

"date\_histogram": {

"field": "sold",

"interval": "quarter", https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

"format": "yyyy-MM-dd",

"min\_doc\_count" : 0,

"extended\_bounds" : {

"min" : "2014-01-01",

"max" : "2014-12-31"

}

},

"aggs": {

"per\_make\_sum": {

"terms": {

"field": "make"

},

"aggs": {

"sum\_price": {

"sum": { "field": "price" } https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

}

}

},

"total\_sum": {

"sum": { "field": "price" } https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png

}

}

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_extended_example.html#CO195-1) | 注意我们把时间间隔从 month 改成了 quarter 。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_extended_example.html#CO195-2) | 计算每种品牌的总销售金额。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_extended_example.html#CO195-3) | 也计算所有全部品牌的汇总销售金额。 |

得到的结果（截去了一大部分）如下：

{

....

"aggregations": {

"sales": {

"buckets": [

{

"key\_as\_string": "2014-01-01",

"key": 1388534400000,

"doc\_count": 2,

"total\_sum": {

"value": 105000

},

"per\_make\_sum": {

"buckets": [

{

"key": "bmw",

"doc\_count": 1,

"sum\_price": {

"value": 80000

}

},

{

"key": "ford",

"doc\_count": 1,

"sum\_price": {

"value": 25000

}

}

]

}

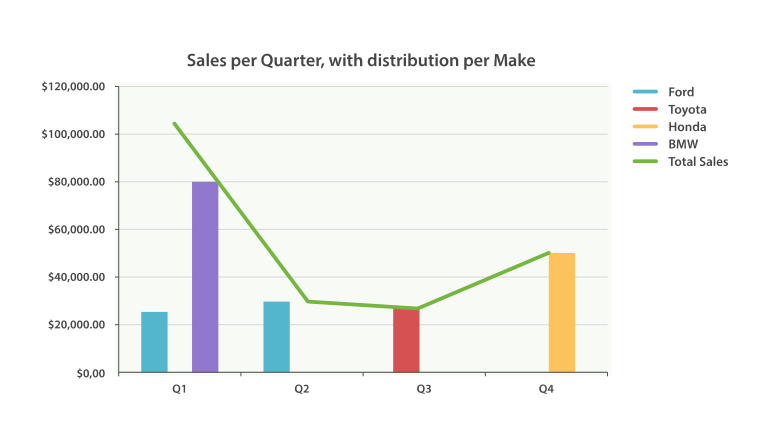
},

...

}

我们把结果绘成图，得到如 [图 38 “按品牌分布的每季度销售额”](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_extended_example.html#date-histo-ts2) 所示的总销售额的折线图和每个品牌（每季度）的柱状图。

**图 38. 按品牌分布的每季度销售额**



### 潜力无穷

这些很明显都是简单例子，但图表聚合其实是潜力无穷的。 如 [图 39 “Kibana--用聚合构建的实时分析面板”](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_the_sky_8217_s_the_limit.html#kibana-img)展示了 Kibana 中用各种聚合构建的面板。

**图 39. Kibana--用聚合构建的实时分析面板**



因为聚合的实时性，类似这样的面板很容易查询、操作和交互。这使得它们成为需要分析数据又不会构建 Hadoop 作业的非技术人员的理想工具。

当然，为了构建类似 Kibana 这样的强大面板，你可能需要更深的知识，比如基于范围、过滤以及排序的聚合。

## 范围限定的聚合

所有聚合的例子到目前为止，你可能已经注意到，我们的搜索请求省略了一个 query 。 整个请求只不过是一个聚合。

聚合可以与搜索请求同时执行，但是我们需要理解一个新概念： 范围 。 默认情况下，聚合与查询是对同一范围进行操作的，也就是说，聚合是基于我们查询匹配的文档集合进行计算的。

让我们看看第一个聚合的示例：

GET /cars/transactions/\_search

{

"size" : 0,

"aggs" : {

"colors" : {

"terms" : {

"field" : "color"

}

}

}

}

我们可以看到聚合是隔离的。现实中，Elasticsearch 认为 "没有指定查询" 和 "查询所有文档" 是等价的。前面这个查询内部会转化成下面的这个请求：

GET /cars/transactions/\_search

{

"size" : 0,

"query" : {

"match\_all" : {}

},

"aggs" : {

"colors" : {

"terms" : {

"field" : "color"

}

}

}

}

因为聚合总是对查询范围内的结果进行操作的，所以一个隔离的聚合实际上是在对 match\_all 的结果范围操作，即所有的文档。

一旦有了范围的概念，我们就能更进一步对聚合进行自定义。我们前面所有的示例都是对 所有 数据计算统计信息的：销量最高的汽车，所有汽车的平均售价，最佳销售月份等等。

利用范围，我们可以问“福特在售车有多少种颜色？”诸如此类的问题。可以简单的在请求中加上一个查询（本例中为 match 查询）：

GET /cars/transactions/\_search

{

"query" : {

"match" : {

"make" : "ford"

}

},

"aggs" : {

"colors" : {

"terms" : {

"field" : "color"

}

}

}

}

因为我们没有指定 "size" : 0 ，所以搜索结果和聚合结果都被返回了：

{

...

"hits": {

"total": 2,

"max\_score": 1.6931472,

"hits": [

{

"\_source": {

"price": 25000,

"color": "blue",

"make": "ford",

"sold": "2014-02-12"

}

},

{

"\_source": {

"price": 30000,

"color": "green",

"make": "ford",

"sold": "2014-05-18"

}

}

]

},

"aggregations": {

"colors": {

"buckets": [

{

"key": "blue",

"doc\_count": 1

},

{

"key": "green",

"doc\_count": 1

}

]

}

}

}

看上去这并没有什么，但却对高大上的仪表盘来说至关重要。 加入一个搜索栏可以将任何静态的仪表板变成一个实时数据搜索设备。 这让用户可以搜索数据，查看所有实时更新的图形（由于聚合的支持以及对查询范围的限定）。 这是 Hadoop 无法做到的！

**全局桶**

通常我们希望聚合是在查询范围内的，但有时我们也想要搜索它的子集，而聚合的对象却是 所有 数据。

例如，比方说我们想知道福特汽车与 所有 汽车平均售价的比较。我们可以用普通的聚合（查询范围内的）得到第一个信息，然后用 全局 桶获得第二个信息。

全局 桶包含 所有 的文档，它无视查询的范围。因为它还是一个桶，我们可以像平常一样将聚合嵌套在内：

GET /cars/transactions/\_search

{

"size" : 0,

"query" : {

"match" : {

"make" : "ford"

}

},

"aggs" : {

"single\_avg\_price": {

"avg" : { "field" : "price" } https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

},

"all": {

"global" : {}, https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

"aggs" : {

"avg\_price": {

"avg" : { "field" : "price" } https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png

}

}

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_scoping_aggregations.html#CO196-1) | 聚合操作在查询范围内（例如：所有文档匹配 ford ） |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_scoping_aggregations.html#CO196-2) | global 全局桶没有参数。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_scoping_aggregations.html#CO196-3) | 聚合操作针对所有文档，忽略汽车品牌。 |

single\_avg\_price 度量计算是基于查询范围内所有文档，即所有 福特 汽车。avg\_price 度量是嵌套在 全局 桶下的，这意味着它完全忽略了范围并对所有文档进行计算。聚合返回的平均值是所有汽车的平均售价。

如果能一直坚持读到这里，应该知道我们有个真言：尽可能的使用过滤器。它同样可以应用于聚合，在下一章中，我们会展示如何对聚合结果进行过滤而不是仅对查询范围做限定。

## 过滤和聚合

聚合范围限定还有一个自然的扩展就是过滤。因为聚合是在查询结果范围内操作的，任何可以适用于查询的过滤器也可以应用在聚合上。

### 过滤

如果我们想找到售价在 $10,000 美元之上的所有汽车同时也为这些车计算平均售价， 可以简单地使用一个 constant\_score 查询和 filter 约束：

GET /cars/transactions/\_search

{

"size" : 0,

"query" : {

"constant\_score": {

"filter": {

"range": {

"price": {

"gte": 10000

}

}

}

}

},

"aggs" : {

"single\_avg\_price": {

"avg" : { "field" : "price" }

}

}

}

这正如我们在前面章节中讨论过那样，从根本上讲，使用 non-scoring 查询和使用 match 查询没有任何区别。查询（包括了一个过滤器）返回一组文档的子集，聚合正是操作这些文档。使用 filtering query 会忽略评分，并有可能会缓存结果数据等等。

### 过滤桶

但是如果我们只想对聚合结果过滤怎么办？ 假设我们正在为汽车经销商创建一个搜索页面， 我们希望显示用户搜索的结果，但是我们同时也想在页面上提供更丰富的信息，包括（与搜索匹配的）上个月度汽车的平均售价。

这里我们无法简单的做范围限定，因为有两个不同的条件。搜索结果必须是 ford ，但是聚合结果必须满足 ford AND sold > now - 1M 。

为了解决这个问题，我们可以用一种特殊的桶，叫做 filter （注：过滤桶） 。 我们可以指定一个过滤桶，当文档满足过滤桶的条件时，我们将其加入到桶内。

查询结果如下：

GET /cars/transactions/\_search

{

"size" : 0,

"query":{

"match": {

"make": "ford"

}

},

"aggs":{

"recent\_sales": {

"filter": { https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

"range": {

"sold": {

"from": "now-1M"

}

}

},

"aggs": {

"average\_price":{

"avg": {

"field": "price" https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

}

}

}

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_filter_bucket.html#CO197-1) | 使用 过滤 桶在 查询 范围基础上应用过滤器。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_filter_bucket.html#CO197-2) | avg 度量只会对 ford 和上个月售出的文档计算平均售价。 |

因为 filter 桶和其他桶的操作方式一样，所以可以随意将其他桶和度量嵌入其中。所有嵌套的组件都会 "继承" 这个过滤，这使我们可以按需针对聚合过滤出选择部分。

### 后过滤器

目前为止，我们可以同时对搜索结果和聚合结果进行过滤（不计算得分的 filter 查询），以及针对聚合结果的一部分进行过滤（ filter 桶）。

我们可能会想，"只过滤搜索结果，不过滤聚合结果呢？" 答案是使用 post\_filter 。

它是接收一个过滤器的顶层搜索请求元素。这个过滤器在查询 之后 执行（这正是该过滤器的名字的由来：它在查询之后 post 执行）。正因为它在查询之后执行，它对查询范围没有任何影响，所以对聚合也不会有任何影响。

我们可以利用这个行为对查询条件应用更多的过滤器，而不会影响其他的操作，就如 UI 上的各个分类面。让我们为汽车经销商设计另外一个搜索页面，这个页面允许用户搜索汽车同时可以根据颜色来过滤。颜色的选项是通过聚合获得的：

GET /cars/transactions/\_search

{

"size" : 0,

"query": {

"match": {

"make": "ford"

}

},

"post\_filter": { https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

"term" : {

"color" : "green"

}

},

"aggs" : {

"all\_colors": {

"terms" : { "field" : "color" }

}

}

}

**拷贝为 CURL**[在 SENSE 中查看](http://localhost:5601/app/sense/?load_from=https://www.elastic.co/guide/cn/elasticsearch/guide/current/snippets/300_Aggregations/45_filtering.json)

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_post_filter.html#CO198-1) | post\_filter 元素是 top-level 而且仅对命中结果进行过滤。 |

查询 部分找到所有的 ford 汽车，然后用 terms 聚合创建一个颜色列表。因为聚合对查询范围进行操作，颜色列表与福特汽车有的颜色相对应。

最后， post\_filter 会过滤搜索结果，只展示绿色 ford 汽车。这在查询执行过 后 发生，所以聚合不受影响。

这通常对 UI 的连贯一致性很重要，可以想象用户在界面商选择了一类颜色（比如：绿色），期望的是搜索结果已经被过滤了，而 不是 过滤界面上的选项。如果我们应用 filter 查询，界面会马上变成 只 显示 绿色 作为选项，这不是用户想要的！

警告

**性能考虑（Performance consideration）**

当你需要对搜索结果和聚合结果做不同的过滤时，你才应该使用 post\_filter ， 有时用户会在普通搜索使用 post\_filter 。

不要这么做！ post\_filter 的特性是在查询 之后 执行，任何过滤对性能带来的好处（比如缓存）都会完全失去。

在我们需要不同过滤时， post\_filter 只与聚合一起使用。

### 小结

选择合适类型的过滤（如：搜索命中、聚合或两者兼有）通常和我们期望如何表现用户交互有关。选择合适的过滤器（或组合）取决于我们期望如何将结果呈现给用户。

* 在 filter 过滤中的 non-scoring 查询，同时影响搜索结果和聚合结果。
* filter 桶影响聚合。
* post\_filter 只影响搜索结果。

## 多桶排序

多值桶（ terms 、 histogram 和 date\_histogram ）动态生成很多桶。 Elasticsearch 是如何决定这些桶展示给用户的顺序呢？

默认的，桶会根据 doc\_count 降序排列。这是一个好的默认行为，因为通常我们想要找到文档中与查询条件相关的最大值：售价、人口数量、频率。但有些时候我们希望能修改这个顺序，不同的桶有着不同的处理方式。

### 内置排序

这些排序模式是桶 固有的 能力：它们操作桶生成的数据 ，比如 doc\_count 。 它们共享相同的语法，但是根据使用桶的不同会有些细微差别。

让我们做一个 terms 聚合但是按 doc\_count 值的升序排序：

GET /cars/transactions/\_search

{

"size" : 0,

"aggs" : {

"colors" : {

"terms" : {

"field" : "color",

"order": {

"\_count" : "asc" https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

}

}

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_intrinsic_sorts.html#CO199-1) | 用关键字 \_count ，我们可以按 doc\_count 值的升序排序。 |

我们为聚合引入了一个 order 对象， 它允许我们可以根据以下几个值中的一个值进行排序：

\_count

按文档数排序。对 terms 、 histogram 、 date\_histogram 有效。

\_term

按词项的字符串值的字母顺序排序。只在 terms 内使用。

\_key

按每个桶的键值数值排序（理论上与 \_term 类似）。 只在 histogram 和 date\_histogram 内使用。

### 按度量排序

有时，我们会想基于度量计算的结果值进行排序。 在我们的汽车销售分析仪表盘中，我们可能想按照汽车颜色创建一个销售条状图表，但按照汽车平均售价的升序进行排序。

我们可以增加一个度量，再指定 order 参数引用这个度量即可：

GET /cars/transactions/\_search

{

"size" : 0,

"aggs" : {

"colors" : {

"terms" : {

"field" : "color",

"order": {

"avg\_price" : "asc" https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

}

},

"aggs": {

"avg\_price": {

"avg": {"field": "price"} https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

}

}

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_sorting_by_a_metric.html#CO200-2) | 计算每个桶的平均售价。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_sorting_by_a_metric.html#CO200-1) | 桶按照计算平均值的升序排序。 |

我们可以采用这种方式用任何度量排序，只需简单的引用度量的名字。不过有些度量会输出多个值。 extended\_stats 度量是一个很好的例子：它输出好几个度量值。

如果我们想使用多值度量进行排序， 我们只需以关心的度量为关键词使用点式路径：

GET /cars/transactions/\_search

{

"size" : 0,

"aggs" : {

"colors" : {

"terms" : {

"field" : "color",

"order": {

"stats.variance" : "asc" https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

}

},

"aggs": {

"stats": {

"extended\_stats": {"field": "price"}

}

}

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_sorting_by_a_metric.html#CO201-1) | 使用 . 符号，根据感兴趣的度量进行排序。 |

在上面这个例子中，我们按每个桶的方差来排序，所以这种颜色售价方差最小的会排在结果集最前面。

### 基于“深度”度量排序

在前面的示例中，度量是桶的直接子节点。平均售价是根据每个 term 来计算的。 在一定条件下，我们也有可能对 更深 的度量进行排序，比如孙子桶或从孙桶。

我们可以定义更深的路径，将度量用尖括号（ > ）嵌套起来，像这样： my\_bucket>another\_bucket>metric 。

需要提醒的是嵌套路径上的每个桶都必须是 单值 的。 filter 桶生成 一个单值桶：所有与过滤条件匹配的文档都在桶中。 多值桶（如：terms ）动态生成许多桶，无法通过指定一个确定路径来识别。

目前，只有三个单值桶： filter 、 global 和 reverse\_nested 。让我们快速用示例说明，创建一个汽车售价的直方图，但是按照红色和绿色（不包括蓝色）车各自的方差来排序：

GET /cars/transactions/\_search

{

"size" : 0,

"aggs" : {

"colors" : {

"histogram" : {

"field" : "price",

"interval": 20000,

"order": {

"red\_green\_cars>stats.variance" : "asc" https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

}

},

"aggs": {

"red\_green\_cars": {

"filter": { "terms": {"color": ["red", "green"]}}, https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

"aggs": {

"stats": {"extended\_stats": {"field" : "price"}} https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png

}

}

}

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_sorting_based_on_deep_metrics.html#CO202-1) | 按照嵌套度量的方差对桶的直方图进行排序。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_sorting_based_on_deep_metrics.html#CO202-2) | 因为我们使用单值过滤器 filter ，我们可以使用嵌套排序。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_sorting_based_on_deep_metrics.html#CO202-3) | 按照生成的度量对统计结果进行排序。 |

本例中，可以看到我们如何访问一个嵌套的度量。 stats 度量是 red\_green\_cars 聚合的子节点，而 red\_green\_cars 又是 colors 聚合的子节点。 为了根据这个度量排序，我们定义了路径 red\_green\_cars>stats.variance 。我们可以这么做，因为 filter 桶是个单值桶。

## 近似聚合

如果所有的数据都在一台机器上，那么生活会容易许多。 CS201 课上教的经典算法就足够应付这些问题。如果所有的数据都在一台机器上，那么也就不需要像 Elasticsearch 这样的分布式软件了。不过一旦我们开始分布式存储数据，就需要小心地选择算法。

有些算法可以分布执行，到目前为止讨论过的所有聚合都是单次请求获得精确结果的。这些类型的算法通常被认为是 高度并行的 ，因为它们无须任何额外代价，就能在多台机器上并行执行。比如当计算 max 度量时，以下的算法就非常简单：

1. 把请求广播到所有分片。
2. 查看每个文档的 price 字段。如果 price > current\_max ，将 current\_max 替换成 price 。
3. 返回所有分片的最大 price 并传给协调节点。
4. 找到从所有分片返回的最大 price 。这是最终的最大值。

这个算法可以随着机器数的线性增长而横向扩展，无须任何协调操作（机器之间不需要讨论中间结果），而且内存消耗很小（一个整型就能代表最大值）。

不幸的是，不是所有的算法都像获取最大值这样简单。更加复杂的操作则需要在算法的性能和内存使用上做出权衡。对于这个问题，我们有个三角因子模型：大数据、精确性和实时性。

我们需要选择其中两项：

精确 + 实时

数据可以存入单台机器的内存之中，我们可以随心所欲，使用任何想用的算法。结果会 100% 精确，响应会相对快速。

大数据 + 精确

传统的 Hadoop。可以处理 PB 级的数据并且为我们提供精确的答案，但它可能需要几周的时间才能为我们提供这个答案。

大数据 + 实时

近似算法为我们提供准确但不精确的结果。

Elasticsearch 目前支持两种近似算法（ cardinality 和 percentiles ）。 它们会提供准确但不是 100% 精确的结果。 以牺牲一点小小的估算错误为代价，这些算法可以为我们换来高速的执行效率和极小的内存消耗。

对于 大多数 应用领域，能够 实时 返回高度准确的结果要比 100% 精确结果重要得多。乍一看这可能是天方夜谭。有人会叫 “我们需要精确的答案！” 。但仔细考虑 0.5% 误差所带来的影响：

* 99% 的网站延时都在 132ms 以下。
* 0.5% 的误差对以上延时的影响在正负 0.66ms 。
* 近似计算的结果会在毫秒内返回，而“完全正确”的结果就可能需要几秒，甚至无法返回。

只要简单的查看网站的延时情况，难道我们会在意近似结果是 132.66ms 而不是 132ms 吗？当然，不是所有的领域都能容忍这种近似结果，但对于绝大多数来说是没有问题的。接受近似结果更多的是一种 文化观念上 的壁垒而不是商业或技术上的需要。

### 统计去重后的数量

Elasticsearch 提供的首个近似聚合是 cardinality （注：基数）度量。 它提供一个字段的基数，即该字段的 distinct 或者 unique 值的数目。 你可能会对 SQL 形式比较熟悉：

SELECT COUNT(DISTINCT color)

FROM cars

去重是一个很常见的操作，可以回答很多基本的业务问题：

* 网站独立访客是多少？
* 卖了多少种汽车？
* 每月有多少独立用户购买了商品？

我们可以用 cardinality 度量确定经销商销售汽车颜色的数量：

GET /cars/transactions/\_search

{

"size" : 0,

"aggs" : {

"distinct\_colors" : {

"cardinality" : {

"field" : "color"

}

}

}

}

返回的结果表明已经售卖了三种不同颜色的汽车：

...

"aggregations": {

"distinct\_colors": {

"value": 3

}

}

...

可以让我们的例子变得更有用：每月有多少颜色的车被售出？为了得到这个度量，我们只需要将一个 cardinality 度量嵌入一个 date\_histogram ：

GET /cars/transactions/\_search

{

"size" : 0,

"aggs" : {

"months" : {

"date\_histogram": {

"field": "sold",

"interval": "month"

},

"aggs": {

"distinct\_colors" : {

"cardinality" : {

"field" : "color"

}

}

}

}

}

}

**学会权衡**

正如我们本章开头提到的， cardinality 度量是一个近似算法。 它是基于 [HyperLogLog++](http://static.googleusercontent.com/media/research.google.com/en/pubs/archive/40671.pdf) （HLL）算法的。 HLL 会先对我们的输入作哈希运算，然后根据哈希运算的结果中的 bits 做概率估算从而得到基数。

我们不需要理解技术细节（如果确实感兴趣，可以阅读这篇论文）， 但我们最好应该关注一下这个算法的 特性 ：

* 可配置的精度，用来控制内存的使用（更精确 ＝ 更多内存）。
* 小的数据集精度是非常高的。
* 我们可以通过配置参数，来设置去重需要的固定内存使用量。无论数千还是数十亿的唯一值，内存使用量只与你配置的精确度相关。

要配置精度，我们必须指定 precision\_threshold 参数的值。 这个阈值定义了在何种基数水平下我们希望得到一个近乎精确的结果。参考以下示例：

GET /cars/transactions/\_search

{

"size" : 0,

"aggs" : {

"distinct\_colors" : {

"cardinality" : {

"field" : "color",

"precision\_threshold" : 100 https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

}

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/cardinality.html#CO203-1) | precision\_threshold 接受 0–40,000 之间的数字，更大的值还是会被当作 40,000 来处理。 |

示例会确保当字段唯一值在 100 以内时会得到非常准确的结果。尽管算法是无法保证这点的，但如果基数在阈值以下，几乎总是 100% 正确的。高于阈值的基数会开始节省内存而牺牲准确度，同时也会对度量结果带入误差。

对于指定的阈值，HLL 的数据结构会大概使用 precision\_threshold \* 8 字节的内存，所以就必须在牺牲内存和获得额外的准确度间做平衡。

在实际应用中， 100 的阈值可以在唯一值为百万的情况下仍然将误差维持 5% 以内。

**速度优化**

如果想要获得唯一值的数目， 通常 需要查询整个数据集合（或几乎所有数据）。 所有基于所有数据的操作都必须迅速，原因是显然的。 HyperLogLog 的速度已经很快了，它只是简单的对数据做哈希以及一些位操作。

但如果速度对我们至关重要，可以做进一步的优化。 因为 HLL 只需要字段内容的哈希值，我们可以在索引时就预先计算好。 就能在查询时跳过哈希计算然后将哈希值从 fielddata 直接加载出来。

注意

预先计算哈希值只对内容很长或者基数很高的字段有用，计算这些字段的哈希值的消耗在查询时是无法忽略的。

尽管数值字段的哈希计算是非常快速的，存储它们的原始值通常需要同样（或更少）的内存空间。这对低基数的字符串字段同样适用，Elasticsearch 的内部优化能够保证每个唯一值只计算一次哈希。

基本上说，预先计算并不能保证所有的字段都更快，它只对那些具有高基数和/或者内容很长的字符串字段有作用。需要记住的是，预计算只是简单的将查询消耗的时间提前转移到索引时，并非没有任何代价，区别在于你可以选择在 什么时候 做这件事，要么在索引时，要么在查询时。

要想这么做，我们需要为数据增加一个新的多值字段。我们先删除索引，再增加一个包括哈希值字段的映射，然后重新索引：

DELETE /cars/

PUT /cars/

{

"mappings": {

"transactions": {

"properties": {

"color": {

"type": "string",

"fields": {

"hash": {

"type": "murmur3" https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

}

}

}

}

}

}

}

POST /cars/transactions/\_bulk

{ "index": {}}

{ "price" : 10000, "color" : "red", "make" : "honda", "sold" : "2014-10-28" }

{ "index": {}}

{ "price" : 20000, "color" : "red", "make" : "honda", "sold" : "2014-11-05" }

{ "index": {}}

{ "price" : 30000, "color" : "green", "make" : "ford", "sold" : "2014-05-18" }

{ "index": {}}

{ "price" : 15000, "color" : "blue", "make" : "toyota", "sold" : "2014-07-02" }

{ "index": {}}

{ "price" : 12000, "color" : "green", "make" : "toyota", "sold" : "2014-08-19" }

{ "index": {}}

{ "price" : 20000, "color" : "red", "make" : "honda", "sold" : "2014-11-05" }

{ "index": {}}

{ "price" : 80000, "color" : "red", "make" : "bmw", "sold" : "2014-01-01" }

{ "index": {}}

{ "price" : 25000, "color" : "blue", "make" : "ford", "sold" : "2014-02-12" }

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/cardinality.html#CO204-1) | 多值字段的类型是 murmur3 ，这是一个哈希函数。 |

现在当我们执行聚合时，我们使用 color.hash 字段而不是 color 字段：

GET /cars/transactions/\_search

{

"size" : 0,

"aggs" : {

"distinct\_colors" : {

"cardinality" : {

"field" : "color.hash" https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

}

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/cardinality.html#CO205-1) | 注意我们指定的是哈希过的多值字段，而不是原始字段。 |

现在 cardinality 度量会读取 "color.hash" 里的值（预先计算的哈希值），取代动态计算原始值的哈希。

单个文档节省的时间是非常少的，但是如果你聚合一亿数据，每个字段多花费 10 纳秒的时间，那么在每次查询时都会额外增加 1 秒，如果我们要在非常大量的数据里面使用 cardinality ，我们可以权衡使用预计算的意义，是否需要提前计算 hash，从而在查询时获得更好的性能，做一些性能测试来检验预计算哈希是否适用于你的应用场景。。

### 百分位计算

Elasticsearch 提供的另外一个近似度量就是 percentiles 百分位数度量。 百分位数展现某以具体百分比下观察到的数值。例如，第95个百分位上的数值，是高于 95% 的数据总和。

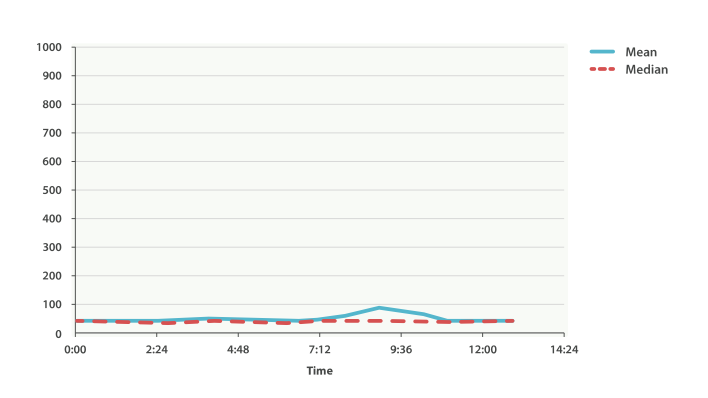
百分位数通常用来找出异常。在（统计学）的正态分布下，第 0.13 和 第 99.87 的百分位数代表与均值距离三倍标准差的值。任何处于三倍标准差之外的数据通常被认为是不寻常的，因为它与平均值相差太大。

更具体的说，假设我们正运行一个庞大的网站，一个很重要的工作是保证用户请求能得到快速响应，因此我们就需要监控网站的延时来判断响应是否能保证良好的用户体验。

在此场景下，一个常用的度量方法就是平均响应延时。 但这并不是一个好的选择（尽管很常用），因为平均数通常会隐藏那些异常值， 中位数有着同样的问题。 我们可以尝试最大值，但这个度量会轻而易举的被单个异常值破坏。

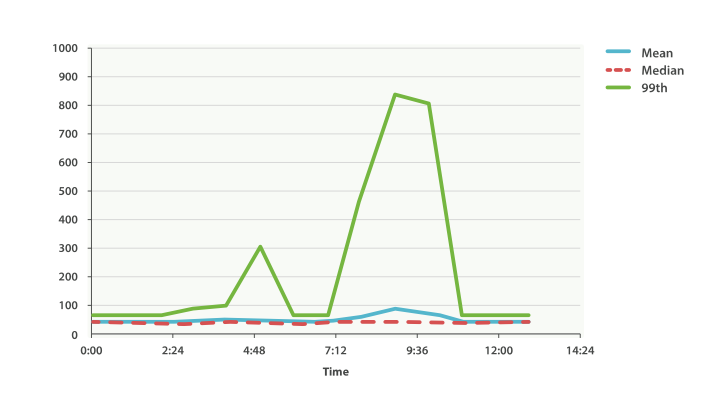
在图 [图 40 “Average request latency over time”](https://www.elastic.co/guide/cn/elasticsearch/guide/current/percentiles.html#percentile-mean-median) 查看问题。如果我们依靠如平均值或中位数这样的简单度量，就会得到像这样一幅图 [图 40 “Average request latency over time”](https://www.elastic.co/guide/cn/elasticsearch/guide/current/percentiles.html#percentile-mean-median) 。

**图 40. Average request latency over time**



一切正常。 图上有轻微的波动，但没有什么值得关注的。 但如果我们加载 99 百分位数时（这个值代表最慢的 1% 的延时），我们看到了完全不同的一幅画面，如图 [图 41 “Average request latency with 99th percentile over time”](https://www.elastic.co/guide/cn/elasticsearch/guide/current/percentiles.html#percentile-mean-median-percentile) 。

**图 41. Average request latency with 99th percentile over time**



令人吃惊！在上午九点半时，均值只有 75ms。如果作为一个系统管理员，我们都不会看他第二眼。 一切正常！但 99 百分位告诉我们有 1% 的用户碰到的延时超过 850ms，这是另外一幅场景。 在上午4点48时也有一个小波动，这甚至无法从平均值和中位数曲线上观察到。

这只是百分位的一个应用场景，百分位还可以被用来快速用肉眼观察数据的分布，检查是否有数据倾斜或双峰甚至更多。

**百分位度量**

让我加载一个新的数据集（汽车的数据不太适用于百分位）。我们要索引一系列网站延时数据然后运行一些百分位操作进行查看：

POST /website/logs/\_bulk

{ "index": {}}

{ "latency" : 100, "zone" : "US", "timestamp" : "2014-10-28" }

{ "index": {}}

{ "latency" : 80, "zone" : "US", "timestamp" : "2014-10-29" }

{ "index": {}}

{ "latency" : 99, "zone" : "US", "timestamp" : "2014-10-29" }

{ "index": {}}

{ "latency" : 102, "zone" : "US", "timestamp" : "2014-10-28" }

{ "index": {}}

{ "latency" : 75, "zone" : "US", "timestamp" : "2014-10-28" }

{ "index": {}}

{ "latency" : 82, "zone" : "US", "timestamp" : "2014-10-29" }

{ "index": {}}

{ "latency" : 100, "zone" : "EU", "timestamp" : "2014-10-28" }

{ "index": {}}

{ "latency" : 280, "zone" : "EU", "timestamp" : "2014-10-29" }

{ "index": {}}

{ "latency" : 155, "zone" : "EU", "timestamp" : "2014-10-29" }

{ "index": {}}

{ "latency" : 623, "zone" : "EU", "timestamp" : "2014-10-28" }

{ "index": {}}

{ "latency" : 380, "zone" : "EU", "timestamp" : "2014-10-28" }

{ "index": {}}

{ "latency" : 319, "zone" : "EU", "timestamp" : "2014-10-29" }

数据有三个值：延时、数据中心的区域以及时间戳。让我们对数据全集进行 百分位 操作以获得数据分布情况的直观感受：

GET /website/logs/\_search

{

"size" : 0,

"aggs" : {

"load\_times" : {

"percentiles" : {

"field" : "latency" https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

}

},

"avg\_load\_time" : {

"avg" : {

"field" : "latency" https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

}

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/percentiles.html#CO206-1) | percentiles 度量被应用到 latency 延时字段。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/percentiles.html#CO206-2) | 为了比较，我们对相同字段使用 avg 度量。 |

默认情况下，percentiles 度量会返回一组预定义的百分位数值： [1, 5, 25, 50, 75, 95, 99] 。它们表示了人们感兴趣的常用百分位数值，极端的百分位数在范围的两边，其他的一些处于中部。在返回的响应中，我们可以看到最小延时在 75ms 左右，而最大延时差不多有 600ms。与之形成对比的是，平均延时在 200ms 左右， 信息并不是很多：

...

"aggregations": {

"load\_times": {

"values": {

"1.0": 75.55,

"5.0": 77.75,

"25.0": 94.75,

"50.0": 101,

"75.0": 289.75,

"95.0": 489.34999999999985,

"99.0": 596.2700000000002

}

},

"avg\_load\_time": {

"value": 199.58333333333334

}

}

所以显然延时的分布很广，让我们看看它们是否与数据中心的地理区域有关：

GET /website/logs/\_search

{

"size" : 0,

"aggs" : {

"zones" : {

"terms" : {

"field" : "zone" https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

},

"aggs" : {

"load\_times" : {

"percentiles" : { https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

"field" : "latency",

"percents" : [50, 95.0, 99.0] https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png

}

},

"load\_avg" : {

"avg" : {

"field" : "latency"

}

}

}

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/percentiles.html#CO207-1) | 首先根据区域我们将延时分到不同的桶中。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/percentiles.html#CO207-2) | 再计算每个区域的百分位数值。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/percentiles.html#CO207-3) | percents 参数接受了我们想返回的一组百分位数，因为我们只对长的延时感兴趣。 |

在响应结果中，我们发现欧洲区域（EU）要比美国区域（US）慢很多，在美国区域（US），50 百分位与 99 百分位十分接近，它们都接近均值。

与之形成对比的是，欧洲区域（EU）在 50 和 99 百分位有较大区分。现在，显然可以发现是欧洲区域（EU）拉低了延时的统计信息，我们知道欧洲区域的 50% 延时都在 300ms+。

...

"aggregations": {

"zones": {

"buckets": [

{

"key": "eu",

"doc\_count": 6,

"load\_times": {

"values": {

"50.0": 299.5,

"95.0": 562.25,

"99.0": 610.85

}

},

"load\_avg": {

"value": 309.5

}

},

{

"key": "us",

"doc\_count": 6,

"load\_times": {

"values": {

"50.0": 90.5,

"95.0": 101.5,

"99.0": 101.9

}

},

"load\_avg": {

"value": 89.66666666666667

}

}

]

}

}

...

**百分位等级**

这里有另外一个紧密相关的度量叫 percentile\_ranks 。 percentiles 度量告诉我们落在某个百分比以下的所有文档的最小值。例如，如果 50 百分位是 119ms，那么有 50% 的文档数值都不超过 119ms。 percentile\_ranks 告诉我们某个具体值属于哪个百分位。119ms 的 percentile\_ranks 是在 50 百分位。 这基本是个双向关系，例如：

* 50 百分位是 119ms。
* 119ms 百分位等级是 50 百分位。

所以假设我们网站必须维持的服务等级协议（SLA）是响应时间低于 210ms。然后，开个玩笑，我们老板警告我们如果响应时间超过 800ms 会把我开除。可以理解的是，我们希望知道有多少百分比的请求可以满足 SLA 的要求（并期望至少在 800ms 以下！）。

为了做到这点，我们可以应用 percentile\_ranks 度量而不是 percentiles 度量：

GET /website/logs/\_search

{

"size" : 0,

"aggs" : {

"zones" : {

"terms" : {

"field" : "zone"

},

"aggs" : {

"load\_times" : {

"percentile\_ranks" : {

"field" : "latency",

"values" : [210, 800] https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

}

}

}

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/percentiles.html#CO208-1) | percentile\_ranks 度量接受一组我们希望分级的数值。 |

在聚合运行后，我们能得到两个值：

"aggregations": {

"zones": {

"buckets": [

{

"key": "eu",

"doc\_count": 6,

"load\_times": {

"values": {

"210.0": 31.944444444444443,

"800.0": 100

}

}

},

{

"key": "us",

"doc\_count": 6,

"load\_times": {

"values": {

"210.0": 100,

"800.0": 100

}

}

}

]

}

}

这告诉我们三点重要的信息：

* 在欧洲（EU），210ms 的百分位等级是 31.94% 。
* 在美国（US），210ms 的百分位等级是 100% 。
* 在欧洲（EU）和美国（US），800ms 的百分位等级是 100% 。

通俗的说，在欧洲区域（EU）只有 32% 的响应时间满足服务等级协议（SLA），而美国区域（US）始终满足服务等级协议的。但幸运的是，两个区域所有响应时间都在 800ms 以下，所以我们还不会被炒鱿鱼（至少目前不会）。

percentile\_ranks 度量提供了与 percentiles 相同的信息，但它以不同方式呈现，如果我们对某个具体数值更关心，使用它会更方便。

**学会权衡**

和基数一样，计算百分位需要一个近似算法。 朴素的 实现会维护一个所有值的有序列表， 但当我们有几十亿数据分布在几十个节点时，这几乎是不可能的。

取而代之的是 percentiles 使用一个 TDigest 算法，（由 Ted Dunning 在 [Computing Extremely Accurate Quantiles Using T-Digests](https://github.com/tdunning/t-digest/blob/master/docs/t-digest-paper/histo.pdf) 里面提出的）。 与 HyperLogLog 一样，不需要理解完整的技术细节，但有必要了解算法的特性：

* 百分位的准确度与百分位的 极端程度 相关，也就是说 1 或 99 的百分位要比 50 百分位要准确。这只是数据结构内部机制的一种特性，但这是一个好的特性，因为多数人只关心极端的百分位。
* 对于数值集合较小的情况，百分位非常准确。如果数据集足够小，百分位可能 100% 精确。
* 随着桶里数值的增长，算法会开始对百分位进行估算。它能有效在准确度和内存节省之间做出权衡。 不准确的程度比较难以总结，因为它依赖于 聚合时数据的分布以及数据量的大小。

与 cardinality 类似，我们可以通过修改参数 compression 来控制内存与准确度之间的比值。

TDigest 算法用节点近似计算百分比：节点越多，准确度越高（同时内存消耗也越大），这都与数据量成正比。 compression 参数限制节点的最大数目为 20 \* compression 。

因此，通过增加压缩比值，可以以消耗更多内存为代价提高百分位数准确性。更大的压缩比值会使算法运行更慢，因为底层的树形数据结构的存储也会增长，也导致操作的代价更高。默认的压缩比值是 100 。

一个节点大约使用 32 字节的内存，所以在最坏的情况下（例如，大量数据有序存入），默认设置会生成一个大小约为 64KB 的 TDigest。 在实际应用中，数据会更随机，所以 TDigest 使用的内存会更少。

## 通过聚合发现异常指标

significant\_terms （SigTerms）聚合 与其他聚合都不相同。 目前为止我们看到的所有聚合在本质上都是简单的数学计算。将不同这些构造块相互组合在一起，我们可以创建复杂的聚合以及数据报表。

significant\_terms 有着不同的工作机制。对有些人来说，它甚至看起来有点像机器学习。significant\_terms 聚合可以在你数据集中找到一些 异常 的指标。

如何解释这些 不常见 的行为？ 这些异常的数据指标通常比我们预估出现的频次要更频繁，这些统计上的异常指标通常象征着数据里的某些有趣信息。

例如，假设我们负责检测和跟踪信用卡欺诈，客户打电话过来抱怨他们信用卡出现异常交易，它们的帐户已经被盗用。这些交易信息只是更严重问题的症状。在最近的某些地区，一些商家有意的盗取客户的信用卡信息，或者它们自己的信息无意中也被盗取。

我们的任务是找到 危害的共同点 ，如果我们有 100 个客户抱怨交易异常，他们很有可能都属于同一个商户，而这家商户有可能就是罪魁祸首。

当然，这里面还有一些特例。例如，很多客户在它们近期交易历史记录中会有很大的商户如亚马逊，我们可以将亚马逊排除在外，然而，在最近一些有问题的信用卡的商家里面也有亚马逊。

这是一个 普通的共同 商户的例子。每个人都共享这个商户，无论有没有遭受危害。我们对它并不感兴趣。

相反，我们有一些很小商户比如街角的一家药店，它们属于 普通但不寻常 的情况，只有一两个客户有交易记录。我们同样可以将这些商户排除，因为所有受到危害的信用卡都没有与这些商户发生过交易，我们可以肯定它们不是安全漏洞的责任方。

我们真正想要的是 不普通的共同 商户。所有受到危害的信用卡都与它们发生过交易，但是在未受危害的背景噪声下，它们并不明显。这些商户属于统计异常，它们比应该出现的频率要高。这些不普通的共同商户很有可能就是需要调查的。

significant\_terms 聚合就是做这些事情。它分析统计你的数据并通过对比正常数据找到可能有异常频次的指标。

暴露的异常指标 代表什么 依赖的你数据。对于信用卡数据，我们可能会想找出信用卡欺诈。 对于电商数据，我们可能会想找出未被识别的人口信息，从而进行更高效的市场推广。 如果我们正在分析日志，我们可能会发现一个服务器会抛出比它本应抛出的更多异常。 significant\_terms 的应用还远不止这些。

### significant\_terms 演示

因为 significant\_terms 聚合 是通过分析统计信息来工作的， 需要为数据设置一个阀值让它们更有效。也就是说无法通过只索引少量示例数据来展示它。

正因如此，我们准备了一个大约 8000 个文档的数据集，并将它的快照保存在一个公共演示库中。可以通过以下步骤在集群中还原这些数据：

1. 在 elasticsearch.yml 配置文件中增加以下配置， 以便将演示库加入到白名单中：

repositories.url.allowed\_urls: ["http://download.elastic.co/\*"]

1. 重启 Elasticsearch。
2. 运行以下快照命令。（更多使用快照的信息，参见 [备份集群（Backing Up Your Cluster）](https://www.elastic.co/guide/cn/elasticsearch/guide/current/backing-up-your-cluster.html)）。
3. PUT /\_snapshot/sigterms https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png
4. {
5. "type": "url",
6. "settings": {
7. "url": "http://download.elastic.co/definitiveguide/sigterms\_demo/"
8. }
9. }
10. GET /\_snapshot/sigterms/\_all https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png
11. POST /\_snapshot/sigterms/snapshot/\_restore https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png

GET /mlmovies,mlratings/\_recovery https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/4.png

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_significant_terms_demo.html#CO209-1) | 注册一个新的只读地址库，并指向演示快照。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_significant_terms_demo.html#CO209-2) | （可选）检查库内关于快照的详细信息。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_significant_terms_demo.html#CO209-3) | 开始还原过程。会在集群中创建两个索引： mlmovies 和 mlratings 。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/4.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_significant_terms_demo.html#CO209-4) | （可选）使用 Recovery API 监控还原过程。 |

注意

数据集有 50 MB 会需要一些时间下载。

在本演示中，会看看 MovieLens 里面用户对电影的评分。在 MovieLens 里，用户可以推荐电影并评分，这样其他用户也可以找到新的电影。 为了演示，会基于输入的电影采用 significant\_terms 对电影进行推荐。

让我们看看示例中的数据，感受一下要处理的内容。本数据集有两个索引， mlmovies 和 mlratings 。首先查看 mlmovies ：

GET mlmovies/\_search https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

{

"took": 4,

"timed\_out": false,

"\_shards": {...},

"hits": {

"total": 10681,

"max\_score": 1,

"hits": [

{

"\_index": "mlmovies",

"\_type": "mlmovie",

"\_id": "2",

"\_score": 1,

"\_source": {

"offset": 2,

"bytes": 34,

"title": "Jumanji (1995)"

}

},

....

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_significant_terms_demo.html#CO210-1) | 执行一个不带查询条件的搜索，以便能看到一组随机演示文档。 |

mlmovies 里的每个文档表示一个电影，数据有两个重要字段：电影ID \_id 和电影名 title 。可以忽略offset 和 bytes 。它们是从原始 CSV 文件抽取数据的过程中产生的中间属性。数据集中有 10，681 部影片。

现在来看看 mlratings ：

GET mlratings/\_search

{

"took": 3,

"timed\_out": false,

"\_shards": {...},

"hits": {

"total": 69796,

"max\_score": 1,

"hits": [

{

"\_index": "mlratings",

"\_type": "mlrating",

"\_id": "00IC-2jDQFiQkpD6vhbFYA",

"\_score": 1,

"\_source": {

"offset": 1,

"bytes": 108,

"movie": [122,185,231,292,

316,329,355,356,362,364,370,377,420,

466,480,520,539,586,588,589,594,616

],

"user": 1

}

},

...

这里可以看到每个用户的推荐信息。每个文档表示一个用户，用 ID 字段 user 来表示， movie 字段维护一个用户观看和推荐过的影片列表。

**基于流行程度推荐（Recommending Based on Popularity）**

可以采取的首个策略就是基于流行程度向用户推荐影片。 对于某部影片，找到所有推荐过它的用户，然后将他们的推荐进行聚合并获得推荐中最流行的五部。

我们可以很容易的通过一个 terms 聚合 以及一些过滤来表示它，看看 Talladega Nights（塔拉迪加之夜）这部影片，它是 Will Ferrell 主演的一部关于全国运动汽车竞赛（NASCAR racing）的喜剧。 在理想情况下，我们的推荐应该找到类似风格的喜剧（很有可能也是 Will Ferrell 主演的）。

首先需要找到影片 Talladega Nights 的 ID：

GET mlmovies/\_search

{

"query": {

"match": {

"title": "Talladega Nights"

}

}

}

...

"hits": [

{

"\_index": "mlmovies",

"\_type": "mlmovie",

"\_id": "46970", https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

"\_score": 3.658795,

"\_source": {

"offset": 9575,

"bytes": 74,

"title": "Talladega Nights: The Ballad of Ricky Bobby (2006)"

}

},

...

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_significant_terms_demo.html#CO211-1) | Talladega Nights 的 ID 是 46970 。 |

有了 ID，可以过滤评分，再应用 terms 聚合从喜欢 Talladega Nights 的用户中找到最流行的影片：

GET mlratings/\_search

{

"size" : 0, https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

"query": {

"filtered": {

"filter": {

"term": {

"movie": 46970 https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

}

}

}

},

"aggs": {

"most\_popular": {

"terms": {

"field": "movie", https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png

"size": 6

}

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_significant_terms_demo.html#CO212-1) | 这次查询 mlratings ， 将结果内容 大小设置 为 0 因为我们只对聚合的结果感兴趣。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_significant_terms_demo.html#CO212-2) | 对影片 Talladega Nights 的 ID 使用过滤器。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_significant_terms_demo.html#CO212-3) | 最后，使用 terms 桶找到最流行的影片。 |

在 mlratings 索引下搜索，然后对影片 Talladega Nights 的 ID 使用过滤器。由于聚合是针对查询范围进行操作的，它可以有效的过滤聚合结果从而得到那些只推荐 Talladega Nights 的用户。 最后，执行 terms聚合得到最流行的影片。 请求排名最前的六个结果，因为 Talladega Nights 本身很有可能就是其中一个结果（并不想重复推荐它）。

返回结果就像这样：

{

...

"aggregations": {

"most\_popular": {

"buckets": [

{

"key": 46970,

"key\_as\_string": "46970",

"doc\_count": 271

},

{

"key": 2571,

"key\_as\_string": "2571",

"doc\_count": 197

},

{

"key": 318,

"key\_as\_string": "318",

"doc\_count": 196

},

{

"key": 296,

"key\_as\_string": "296",

"doc\_count": 183

},

{

"key": 2959,

"key\_as\_string": "2959",

"doc\_count": 183

},

{

"key": 260,

"key\_as\_string": "260",

"doc\_count": 90

}

]

}

}

...

通过一个简单的过滤查询，将得到的结果转换成原始影片名：

GET mlmovies/\_search

{

"query": {

"filtered": {

"filter": {

"ids": {

"values": [2571,318,296,2959,260]

}

}

}

}

}

最后得到以下列表：

1. Matrix, The（黑客帝国）
2. Shawshank Redemption（肖申克的救赎）
3. Pulp Fiction（低俗小说）
4. Fight Club（搏击俱乐部）
5. Star Wars Episode IV: A New Hope（星球大战 IV：曙光乍现）

好吧，这肯定不是一个好的列表！我喜欢所有这些影片。但问题是：几乎 每个人 都喜欢它们。这些影片本来就受大众欢迎，也就是说它们出现在每个人的推荐中都会受欢迎。 这其实是一个流行影片的推荐列表，而不是和影片 Talladega Nights 相关的推荐。

可以通过再次运行聚合轻松验证，而不需要对影片 Talladega Nights 进行过滤。会提供最流行影片的前五名列表：

GET mlratings/\_search

{

"size" : 0,

"aggs": {

"most\_popular": {

"terms": {

"field": "movie",

"size": 5

}

}

}

}

返回列表非常相似：

1. Shawshank Redemption（肖申克的救赎）
2. Silence of the Lambs, The（沉默的羔羊）
3. Pulp Fiction（低俗小说）
4. Forrest Gump（阿甘正传）
5. Star Wars Episode IV: A New Hope（星球大战 IV：曙光乍现）

显然，只是检查最流行的影片是不能足以创建一个良好而又具鉴别能力的推荐系统。

**基于统计的推荐（Recommending Based on Statistics）**

现在场景已经设定好，使用 significant\_terms 。 significant\_terms 会分析喜欢影片 Talladega Nights的用户组（ 前端 用户组），并且确定最流行的电影。 然后为每个用户（ 后端 用户）构造一个流行影片列表，最后将两者进行比较。

统计异常就是与统计背景相比在前景特征组中过度展现的那些影片。理论上讲，它应该是一组喜剧，因为喜欢 Will Ferrell 喜剧的人给这些影片的评分会比一般人高。

让我们试一下：

GET mlratings/\_search

{

"size" : 0,

"query": {

"filtered": {

"filter": {

"term": {

"movie": 46970

}

}

}

},

"aggs": {

"most\_sig": {

"significant\_terms": { https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

"field": "movie",

"size": 6

}

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_significant_terms_demo.html#CO213-1) | 设置几乎一模一样，只是用 significant\_terms 替代了 terms 。 |

正如所见，查询也几乎是一样的。过滤出喜欢影片 Talladega Nights 的用户，他们组成了前景特征用户组。默认情况下， significant\_terms 会使用整个索引里的数据作为统计背景，所以不需要特别的处理。

与 terms 类似，结果返回了一组桶，不过有更多的元数据信息：

...

"aggregations": {

"most\_sig": {

"doc\_count": 271, https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

"buckets": [

{

"key": 46970,

"key\_as\_string": "46970",

"doc\_count": 271,

"score": 256.549815498155,

"bg\_count": 271

},

{

"key": 52245, https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

"key\_as\_string": "52245",

"doc\_count": 59, https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png

"score": 17.66462367106966,

"bg\_count": 185 https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/4.png

},

{

"key": 8641,

"key\_as\_string": "8641",

"doc\_count": 107,

"score": 13.884387742677438,

"bg\_count": 762

},

{

"key": 58156,

"key\_as\_string": "58156",

"doc\_count": 17,

"score": 9.746428133759462,

"bg\_count": 28

},

{

"key": 52973,

"key\_as\_string": "52973",

"doc\_count": 95,

"score": 9.65770100311672,

"bg\_count": 857

},

{

"key": 35836,

"key\_as\_string": "35836",

"doc\_count": 128,

"score": 9.199001116457955,

"bg\_count": 1610

}

]

...

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_significant_terms_demo.html#CO214-1) | 顶层 doc\_count 展现了前景特征组里文档的数量。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_significant_terms_demo.html#CO214-2) | 每个桶里面列出了聚合的键值（例如，影片的ID）。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_significant_terms_demo.html#CO214-3) | 桶内文档的数量 doc\_count 。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/4.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_significant_terms_demo.html#CO214-4) | 背景文档的数量，表示该值在整个统计背景里出现的频度。 |

可以看到，获得的第一个桶是 Talladega Nights 。它可以在所有 271 个文档中找到，这并不意外。让我们看下一个桶：键值 52245 。

这个 ID 对应影片 Blades of Glory（荣誉之刃） ，它是一部关于男子学习滑冰的喜剧，也是由 Will Ferrell 主演。可以看到喜欢 Talladega Nights 的用户对它的推荐是 59 次。 这也意味着 21% 的前景特征用户组推荐了影片 Blades of Glory （ 59 / 271 = 0.2177 ）。

形成对比的是， Blades of Glory 在整个数据集合中仅被推荐了 185 次， 只占 0.26% （ 185 / 69796 = 0.00265 ）。因此 Blades of Glory 是一个统计异常：它在喜欢 Talladega Nights 的用户中是显著的共性（注：uncommonly common）。这样就找到了一个好的推荐！

如果看完整的列表，它们都是好的喜剧推荐（其中很多也是由 Will Ferrell 主演）：

1. Blades of Glory（荣誉之刃）
2. Anchorman: The Legend of Ron Burgundy（王牌播音员）
3. Semi-Pro（半职业选手）
4. Knocked Up（一夜大肚）
5. 40-Year-Old Virgin, The（四十岁的老处男）

这只是 significant\_terms 它强大的一个示例，一旦开始使用 significant\_terms ，可能碰到这样的情况，我们不想要最流行的，而想要显著的共性（注：uncommonly common）。这个简单的聚合可以揭示出一些数据里出人意料的复杂趋势。

## Doc Values and Fielddata编辑

### Doc Values

聚合使用一个叫 doc values 的数据结构（在 [Doc Values 介绍](https://www.elastic.co/guide/cn/elasticsearch/guide/current/docvalues-intro.html) 里简单介绍）。 Doc values 可以使聚合更快、更高效并且内存友好，所以理解它的工作方式十分有益。

Doc values 的存在是因为倒排索引只对某些操作是高效的。 倒排索引的优势 在于查找包含某个项的文档，而对于从另外一个方向的相反操作并不高效，即：确定哪些项是否存在单个文档里，聚合需要这种次级的访问模式。

对于以下倒排索引：

Term Doc\_1 Doc\_2 Doc\_3

------------------------------------

brown | X | X |

dog | X | | X

dogs | | X | X

fox | X | | X

foxes | | X |

in | | X |

jumped | X | | X

lazy | X | X |

leap | | X |

over | X | X | X

quick | X | X | X

summer | | X |

the | X | | X

------------------------------------

如果我们想要获得所有包含 brown 的文档的词的完整列表，我们会创建如下查询：

GET /my\_index/\_search

{

"query" : {

"match" : {

"body" : "brown"

}

},

"aggs" : {

"popular\_terms": {

"terms" : {

"field" : "body"

}

}

}

}

查询部分简单又高效。倒排索引是根据项来排序的，所以我们首先在词项列表中找到 brown ，然后扫描所有列，找到包含 brown 的文档。我们可以快速看到 Doc\_1 和 Doc\_2 包含 brown 这个 token。

然后，对于聚合部分，我们需要找到 Doc\_1 和 Doc\_2 里所有唯一的词项。 用倒排索引做这件事情代价很高： 我们会迭代索引里的每个词项并收集 Doc\_1 和 Doc\_2 列里面 token。这很慢而且难以扩展：随着词项和文档的数量增加，执行时间也会增加。

Doc values 通过转置两者间的关系来解决这个问题。倒排索引将词项映射到包含它们的文档，doc values 将文档映射到它们包含的词项：

Doc Terms

-----------------------------------------------------------------

Doc\_1 | brown, dog, fox, jumped, lazy, over, quick, the

Doc\_2 | brown, dogs, foxes, in, lazy, leap, over, quick, summer

Doc\_3 | dog, dogs, fox, jumped, over, quick, the

-----------------------------------------------------------------

当数据被转置之后，想要收集到 Doc\_1 和 Doc\_2 的唯一 token 会非常容易。获得每个文档行，获取所有的词项，然后求两个集合的并集。

因此，搜索和聚合是相互紧密缠绕的。搜索使用倒排索引查找文档，聚合操作收集和聚合 doc values 里的数据。

注意

Doc values 不仅可以用于聚合。 任何需要查找某个文档包含的值的操作都必须使用它。 除了聚合，还包括排序，访问字段值的脚本，父子关系处理（参见 [父-子关系文档](https://www.elastic.co/guide/cn/elasticsearch/guide/current/parent-child.html) ）。

### 深入文档值

在上一节一开头我们就说文档值（doc values） 是 "更快、更高效并且内存友好" 。 听起来好像是不错的营销术语，不过话说回来文档值到底是如何工作的呢？

文档值是在索引时与倒排索引同时产生的。也就是说文档值是按段来产生的并且是不可变的，正如用于搜索的倒排索引一样。 同样，和倒排索引一样，文档值也序列化到磁盘。这些对于性能和伸缩性很重要。

通过序列化一个持久化的数据结构到磁盘，我们可以依赖于操作系统的缓存来管理内存，而不是在 JVM 堆栈里驻留数据。 当 “工作集（working set）” 数据要小于系统可用内存的情况下，操作系统会自然的将文档值驻留在内存，这将会带来和直接使用 JVM 堆栈数据结构相同的性能。

不过，如果你的工作集远大于可用内存，操作系统会开始根据需要对文档值进行分页开/关。这会显著慢于纯内存驻留的数据结构，当然，它也拥有使用远大于服务器内存容量的伸缩性的好处。 如果这些数据结构是纯粹的存储于 JVM 堆内存，那么唯一的选项只能是随着内存溢出（OutOfMemory）而崩溃（或是实现一个分页模式，正如操作系统的那样）。

注意

因为文档值不是由 JVM 来管理，所以 Elasticsearch 服务器可以配置一个很小的 JVM 堆栈。 这会给操作系统带来更多的内存来做缓存。同时也带来一个好处就是让 JVM 的垃圾回收器工作在一个很小的堆栈，结果就是更快更高效的回收周期。

传统上，我们会建议分配机器内存的 50% 来给 JVM 堆栈。随着文档值的引入，这个建议开始不再适用。 在 64gb 内存的机器上，也许可以考虑给堆栈分配 4-16gb 的内存，而不是之前建议的 32gb。

有关更详细的讨论，查看 [堆内存:大小和交换](https://www.elastic.co/guide/cn/elasticsearch/guide/current/heap-sizing.html).

**列式存储的压缩**

从广义来说，文档值本质上是一个序列化的 列式存储 。 正如我们上一节所讨论的，列式存储 擅长某些操作，因为这些数据的存储天然适合这些查询。

而且，他们也同样擅长数据压缩，特别是数字。 这对于节省磁盘空间和快速访问很重要。现代 CPU 的处理速度要比磁盘快几个数量级（尽管即将到来的 NVMe 驱动器正在迅速缩小差距）。 这意味着减少必须从磁盘读取的数据量总是有益的，尽管需要额外的 CPU 运算来进行解压。

要了解它如何帮助压缩数据，来看一组数字类型的文档值：

Doc Terms

-----------------------------------------------------------------

Doc\_1 | 100

Doc\_2 | 1000

Doc\_3 | 1500

Doc\_4 | 1200

Doc\_5 | 300

Doc\_6 | 1900

Doc\_7 | 4200

-----------------------------------------------------------------

按列布局意味着我们有一个连续的数据块： [100,1000,1500,1200,300,1900,4200] 。因为我们已经知道他们都是数字（而不是像文档或行中看到的异构集合），所以我们可以使用统一的偏移来将他们紧紧排列。

而且，针对这样的数字有很多种压缩技巧。 你会注意到这里每个数字都是 100 的倍数，文档值会检测一个段里面的所有数值，并使用一个 最大公约数 ，方便做进一步的数据压缩。

如果我们保存 100 作为此段的除数，我们可以对每个数字都除以 100，然后得到：[1,10,15,12,3,19,42] 。现在这些数字变小了，只需要很少的位就可以存储下，也减少了磁盘存放的大小。

文档值正是使用了像这样的一些技巧。它会按依次检测以下压缩模式:

1. 如果所有的数值各不相同（或缺失），设置一个标记并记录这些值
2. 如果这些值小于 256，将使用一个简单的编码表
3. 如果这些值大于 256，检测是否存在一个最大公约数
4. 如果没有存在最大公约数，从最小的数值开始，统一计算偏移量进行编码

你会发现这些压缩模式不是传统的通用的压缩方式，比如 DEFLATE 或是 LZ4。 因为列式存储的结构是严格且良好定义的，我们可以通过使用专门的模式来达到比通用压缩算法（如 LZ4 ）更高的压缩效果。

注意

你也许会想 "好吧，貌似对数字很好，不知道字符串怎么样？" 通过借助顺序表（ordinal table），字符类型也是类似进行编码的。字符类型是去重之后存放到顺序表的，通过分配一个 ID，然后这些 ID 和数值类型的文档值一样使用。 也就是说，字符类型和数值类型一样拥有相同的压缩特性。

顺序表本身也有很多压缩技巧，比如固定长度、变长或是前缀字符编码等等。

**禁用文档值**

文档值默认对所有字段启用，除了分析字符类型字段。也就是说所有的数字、地理坐标、日期、IP 和不分析（ not\_analyzed ）字符类型。

分析字符类型暂时还不使用文档值。分析流程会产生很多新的 token，这会让文档值不能高效的工作。我们将在 [聚合与分析](https://www.elastic.co/guide/cn/elasticsearch/guide/current/aggregations-and-analysis.html) 讨论如何使用分析字符类型来做聚合。

因为文档值默认启用，你可以选择对你数据集里面的大多数字段进行聚合和排序操作。但是如果你知道你永远也不会对某些字段进行聚合、排序或是使用脚本操作？

尽管罕见，但当这些情况出现时，你还是希望有办法来为特定的字段禁用文档值。这回为你节省磁盘空间（因为文档值再也没有序列化到磁盘），也许还能提升些许索引速度（因为不需要生成文档值）。

要禁用文档值，在字段的映射（mapping）设置 doc\_values: false 即可。例如，这里我们创建了一个新的索引，字段 "session\_id" 禁用了文档值：

PUT my\_index

{

"mappings": {

"my\_type": {

"properties": {

"session\_id": {

"type": "string",

"index": "not\_analyzed",

"doc\_values": false https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

}

}

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_deep_dive_on_doc_values.html#CO215-1) | 通过设置 doc\_values: false ，这个字段将不能被用于聚合、排序以及脚本操作 |

反过来也是可以进行配置的：让一个字段可以被聚合，通过禁用倒排索引，使它不能被正常搜索，例如：

PUT my\_index

{

"mappings": {

"my\_type": {

"properties": {

"customer\_token": {

"type": "string",

"index": "not\_analyzed",

"doc\_values": true, https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

"index": "no" https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

}

}

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_deep_dive_on_doc_values.html#CO216-1) | 文档值被启用来允许聚合 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_deep_dive_on_doc_values.html#CO216-2) | 索引被禁用了，这让该字段不能被查询/搜索 |

通过设置 doc\_values: true 和 index: no ，我们得到一个只能被用于聚合/排序/脚本的字段。无可否认，这是一个非常罕见的需求，但有时很有用。

### 聚合与分析

有些聚合，比如 terms 桶， 操作字符串字段。字符串字段可能是 analyzed 或者 not\_analyzed ， 那么问题来了，分析是怎么影响聚合的呢？

答案是影响“很多”，有两个原因：分析影响聚合中使用的 tokens ，并且 doc values 不能使用于 分析字符串。

让我们解决第一个问题：分析 tokens 的产生如何影响聚合。首先索引一些代表美国各个州的文档：

POST /agg\_analysis/data/\_bulk

{ "index": {}}

{ "state" : "New York" }

{ "index": {}}

{ "state" : "New Jersey" }

{ "index": {}}

{ "state" : "New Mexico" }

{ "index": {}}

{ "state" : "New York" }

{ "index": {}}

{ "state" : "New York" }

我们希望创建一个数据集里各个州的唯一列表，并且计数。 简单，让我们使用 terms 桶：

GET /agg\_analysis/data/\_search

{

"size" : 0,

"aggs" : {

"states" : {

"terms" : {

"field" : "state"

}

}

}

}

得到结果：

{

...

"aggregations": {

"states": {

"buckets": [

{

"key": "new",

"doc\_count": 5

},

{

"key": "york",

"doc\_count": 3

},

{

"key": "jersey",

"doc\_count": 1

},

{

"key": "mexico",

"doc\_count": 1

}

]

}

}

}

宝贝儿，这完全不是我们想要的！没有对州名计数，聚合计算了每个词的数目。背后的原因很简单：聚合是基于倒排索引创建的，倒排索引是 后置分析（ post-analysis ）的。

当我们把这些文档加入到 Elasticsearch 中时，字符串 "New York" 被分析/分析成 ["new", "york"] 。这些单独的 tokens ，都被用来填充聚合计数，所以我们最终看到 new 的数量而不是 New York 。

这显然不是我们想要的行为，但幸运的是很容易修正它。

我们需要为 state 定义 multifield 并且设置成 not\_analyzed 。这样可以防止 New York 被分析，也意味着在聚合过程中它会以单个 token 的形式存在。让我们尝试完整的过程，但这次指定一个 raw multifield：

DELETE /agg\_analysis/

PUT /agg\_analysis

{

"mappings": {

"data": {

"properties": {

"state" : {

"type": "string",

"fields": {

"raw" : {

"type": "string",

"index": "not\_analyzed"https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

}

}

}

}

}

}

}

POST /agg\_analysis/data/\_bulk

{ "index": {}}

{ "state" : "New York" }

{ "index": {}}

{ "state" : "New Jersey" }

{ "index": {}}

{ "state" : "New Mexico" }

{ "index": {}}

{ "state" : "New York" }

{ "index": {}}

{ "state" : "New York" }

GET /agg\_analysis/data/\_search

{

"size" : 0,

"aggs" : {

"states" : {

"terms" : {

"field" : "state.raw" https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

}

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/aggregations-and-analysis.html#CO217-1) | 这次我们显式映射 state 字段并包括一个 not\_analyzed 辅字段。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/aggregations-and-analysis.html#CO217-2) | 聚合针对 state.raw 字段而不是 state 。 |

现在运行聚合，我们得到了合理的结果：

{

...

"aggregations": {

"states": {

"buckets": [

{

"key": "New York",

"doc\_count": 3

},

{

"key": "New Jersey",

"doc\_count": 1

},

{

"key": "New Mexico",

"doc\_count": 1

}

]

}

}

}

在实际中，这样的问题很容易被察觉，我们的聚合会返回一些奇怪的桶，我们会记住分析的问题。 总之，很少有在聚合中使用分析字段的实例。当我们疑惑时，只要增加一个 multifield 就能有两种选择。

**分析字符串和 Fielddata（Analyzed strings and Fielddata）**

当第一个问题涉及如何聚合数据并显示给用户，第二个问题主要是技术和幕后。

Doc values 不支持 analyzed 字符串字段，因为它们不能很有效的表示多值字符串。 Doc values 最有效的是，当每个文档都有一个或几个 tokens 时， 但不是无数的，分析字符串（想象一个 PDF ，可能有几兆字节并有数以千计的独特 tokens）。

出于这个原因，doc values 不生成分析的字符串，然而，这些字段仍然可以使用聚合，那怎么可能呢？

答案是一种被称为 fielddata 的数据结构。与 doc values 不同，fielddata 构建和管理 100% 在内存中，常驻于 JVM 内存堆。这意味着它本质上是不可扩展的，有很多边缘情况下要提防。 本章的其余部分是解决在分析字符串上下文中 fielddata 的挑战。

注意

从历史上看，fielddata 是 所有 字段的默认设置。但是 Elasticsearch 已迁移到 doc values 以减少 OOM 的几率。分析的字符串是仍然使用 fielddata 的最后一块阵地。 最终目标是建立一个序列化的数据结构类似于 doc values ，可以处理高维度的分析字符串，逐步淘汰 fielddata。

**高基数内存的影响（High-Cardinality Memory Implications）**

避免分析字段的另外一个原因就是：高基数字段在加载到 fielddata 时会消耗大量内存。 分析的过程会经常（尽管不总是这样）生成大量的 token，这些 token 大多都是唯一的。 这会增加字段的整体基数并且带来更大的内存压力。

有些类型的分析对于内存来说 极度 不友好，想想 n-gram 的分析过程， New York 会被 n-gram 分析成以下 token：

* ne
* ew
* w
* y
* yo
* or
* rk

可以想象 n-gram 的过程是如何生成大量唯一 token 的，特别是在分析成段文本的时候。当这些数据加载到内存中，会轻而易举的将我们堆空间消耗殆尽。

因此，在聚合字符串字段之前，请评估情况：

* 这是一个 not\_analyzed 字段吗？如果是，可以通过 doc values 节省内存 。
* 否则，这是一个 analyzed 字段，它将使用 fielddata 并加载到内存中。这个字段因为 ngrams 有一个非常大的基数？如果是，这对于内存来说极度不友好。

### 限制内存使用

一旦分析字符串被加载到 fielddata ，他们会一直在那里，直到被驱逐（或者节点崩溃）。由于这个原因，留意内存的使用情况，了解它是如何以及何时加载的，怎样限制对集群的影响是很重要的。

Fielddata 是 延迟 加载。如果你从来没有聚合一个分析字符串，就不会加载 fielddata 到内存中。此外，fielddata 是基于字段加载的， 这意味着只有很活跃地使用字段才会增加 fielddata 的负担。

然而，这里有一个令人惊讶的地方。假设你的查询是高度选择性和只返回命中的 100 个结果。大多数人认为 fielddata 只加载 100 个文档。

实际情况是，fielddata 会加载索引中（针对该特定字段的） **所有的** 文档，而不管查询的特异性。逻辑是这样：如果查询会访问文档 X、Y 和 Z，那很有可能会在下一个查询中访问其他文档。

与 doc values 不同，fielddata 结构不会在索引时创建。相反，它是在查询运行时，动态填充。这可能是一个比较复杂的操作，可能需要一些时间。 将所有的信息一次加载，再将其维持在内存中的方式要比反复只加载一个 fielddata 的部分代价要低。

JVM 堆 是有限资源的，应该被合理利用。 限制 fielddata 对堆使用的影响有多套机制，这些限制方式非常重要，因为堆栈的乱用会导致节点不稳定（感谢缓慢的垃圾回收机制），甚至导致节点宕机（通常伴随 OutOfMemory 异常）。

**选择堆大小（Choosing a Heap Size）**

在设置 Elasticsearch 堆大小时需要通过 $ES\_HEAP\_SIZE 环境变量应用两个规则：

不要超过可用 RAM 的 50%

Lucene 能很好利用文件系统的缓存，它是通过系统内核管理的。如果没有足够的文件系统缓存空间，性能会受到影响。 此外，专用于堆的内存越多意味着其他所有使用 doc values 的字段内存越少。

不要超过 32 GB

如果堆大小小于 32 GB，JVM 可以利用指针压缩，这可以大大降低内存的使用：每个指针 4 字节而不是 8 字节。

更详细和更完整的堆大小讨论，请参阅 [堆内存:大小和交换](https://www.elastic.co/guide/cn/elasticsearch/guide/current/heap-sizing.html)

**Fielddata的大小**

indices.fielddata.cache.size 控制为 fielddata 分配的堆空间大小。 当你发起一个查询，分析字符串的聚合将会被加载到 fielddata，如果这些字符串之前没有被加载过。如果结果中 fielddata 大小超过了指定 大小 ，其他的值将会被回收从而获得空间。

默认情况下，设置都是 unbounded ，Elasticsearch 永远都不会从 fielddata 中回收数据。

这个默认设置是刻意选择的：fielddata 不是临时缓存。它是驻留内存里的数据结构，必须可以快速执行访问，而且构建它的代价十分高昂。如果每个请求都重载数据，性能会十分糟糕。

一个有界的大小会强制数据结构回收数据。我们会看何时应该设置这个值，但请首先阅读以下警告：

警告

这个设置是一个安全卫士，而非内存不足的解决方案。

如果没有足够空间可以将 fielddata 保留在内存中，Elasticsearch 就会时刻从磁盘重载数据，并回收其他数据以获得更多空间。内存的回收机制会导致重度磁盘I/O，并且在内存中生成很多垃圾，这些垃圾必须在晚些时候被回收掉。

设想我们正在对日志进行索引，每天使用一个新的索引。通常我们只对过去一两天的数据感兴趣，尽管我们会保留老的索引，但我们很少需要查询它们。不过如果采用默认设置，旧索引的 fielddata 永远不会从缓存中回收！ fieldata 会保持增长直到 fielddata 发生断熔（请参阅 [断路器](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_limiting_memory_usage.html#circuit-breaker)），这样我们就无法载入更多的 fielddata。

这个时候，我们被困在了死胡同。但我们仍然可以访问旧索引中的 fielddata，也无法加载任何新的值。相反，我们应该回收旧的数据，并为新值获得更多空间。

为了防止发生这样的事情，可以通过在 config/elasticsearch.yml 文件中增加配置为 fielddata 设置一个上限：

indices.fielddata.cache.size: 20% https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_limiting_memory_usage.html#CO218-1) | 可以设置堆大小的百分比，也可以是某个值，例如： 5gb 。 |

有了这个设置，最久未使用（LRU）的 fielddata 会被回收为新数据腾出空间。

警告

可能发现在线文档有另外一个设置： indices.fielddata.cache.expire 。

这个设置 永远都不会 被使用！它很有可能在不久的将来被弃用。

这个设置要求 Elasticsearch 回收那些 过期 的 fielddata，不管这些值有没有被用到。

这对性能是件 很糟糕 的事情。回收会有消耗性能，它刻意的安排回收方式，而没能获得任何回报。

没有理由使用这个设置：我们不能从理论上假设一个有用的情形。目前，它的存在只是为了向前兼容。我们只在很有以前提到过这个设置，但不幸的是网上各种文章都将其作为一种性能调优的小窍门来推荐。

它不是。永远不要使用！

**监控 fielddata（Monitoring fielddata）**

无论是仔细监控 fielddata 的内存使用情况， 还是看有无数据被回收都十分重要。高的回收数可以预示严重的资源问题以及性能不佳的原因。

Fielddata 的使用可以被监控：

* 按索引使用 [indices-stats API](http://www.elastic.co/guide/en/elasticsearch/reference/current/indices-stats.html) ：

GET /\_stats/fielddata?fields=\*

* 按节点使用 [nodes-stats API](https://www.elastic.co/guide/en/elasticsearch/reference/master/cluster-nodes-stats.html) ：

GET /\_nodes/stats/indices/fielddata?fields=\*

* 按索引节点：

GET /\_nodes/stats/indices/fielddata?level=indices&fields=\*

使用设置 ?fields=\* ，可以将内存使用分配到每个字段。

**断路器**

机敏的读者可能已经发现 fielddata 大小设置的一个问题。fielddata 大小是在数据加载 之后 检查的。 如果一个查询试图加载比可用内存更多的信息到 fielddata 中会发生什么？答案很丑陋：我们会碰到 OutOfMemoryException 。

Elasticsearch 包括一个 fielddata 断熔器 ，这个设计就是为了处理上述情况。 断熔器通过内部检查（字段的类型、基数、大小等等）来估算一个查询需要的内存。它然后检查要求加载的 fielddata 是否会导致 fielddata 的总量超过堆的配置比例。

如果估算查询的大小超出限制，就会 触发 断路器，查询会被中止并返回异常。这都发生在数据加载 之前，也就意味着不会引起 OutOfMemoryException 。

**可用的断路器（Available Circuit Breakers）**

Elasticsearch 有一系列的断路器，它们都能保证内存不会超出限制：

indices.breaker.fielddata.limit

fielddata 断路器默认设置堆的 60% 作为 fielddata 大小的上限。

indices.breaker.request.limit

request 断路器估算需要完成其他请求部分的结构大小，例如创建一个聚合桶，默认限制是堆内存的 40%。

indices.breaker.total.limit

total 揉合 request 和 fielddata 断路器保证两者组合起来不会使用超过堆内存的 70%。

断路器的限制可以在文件 config/elasticsearch.yml 中指定，可以动态更新一个正在运行的集群：

PUT /\_cluster/settings

{

"persistent" : {

"indices.breaker.fielddata.limit" : "40%" https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_limiting_memory_usage.html#CO219-1) | 这个限制是按对内存的百分比设置的。 |

最好为断路器设置一个相对保守点的值。 记住 fielddata 需要与 request 断路器共享堆内存、索引缓冲内存和过滤器缓存。Lucene 的数据被用来构造索引，以及各种其他临时的数据结构。 正因如此，它默认值非常保守，只有 60% 。过于乐观的设置可能会引起潜在的堆栈溢出（OOM）异常，这会使整个节点宕掉。

另一方面，过度保守的值只会返回查询异常，应用程序可以对异常做相应处理。异常比服务器崩溃要好。这些异常应该也能促进我们对查询进行重新评估：为什么单个查询需要超过堆内存的 60% 之多？

提示

在 [Fielddata的大小](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_limiting_memory_usage.html#fielddata-size) 中，我们提过关于给 fielddata 的大小加一个限制，从而确保旧的无用 fielddata 被回收的方法。 indices.fielddata.cache.size 和 indices.breaker.fielddata.limit 之间的关系非常重要。 如果断路器的限制低于缓存大小，没有数据会被回收。为了能正常工作，断路器的限制 必须 要比缓存大小要高。

值得注意的是：断路器是根据总堆内存大小估算查询大小的，而 非 根据实际堆内存的使用情况。 这是由于各种技术原因造成的（例如，堆可能看上去是满的但实际上可能只是在等待垃圾回收，这使我们难以进行合理的估算）。但作为终端用户，这意味着设置需要保守，因为它是根据总堆内存必要的，而 不是 可用堆内存。

### Fielddata 的过滤

设想我们正在运行一个网站允许用户收听他们喜欢的歌曲。 为了让他们可以更容易的管理自己的音乐库，用户可以为歌曲设置任何他们喜欢的标签，这样我们就会有很多歌曲被附上 rock（摇滚） 、 hiphop（嘻哈） 和 electronica（电音） ，但也会有些歌曲被附上 my\_16th\_birthday\_favorite\_anthem 这样的标签。

现在设想我们想要为用户展示每首歌曲最受欢迎的三个标签，很有可能 rock 这样的标签会排在三个中的最前面，而 my\_16th\_birthday\_favorite\_anthem 则不太可能得到评级。 尽管如此，为了计算最受欢迎的标签，我们必须强制将这些一次性使用的项加载到内存中。

感谢 fielddata 过滤，我们可以控制这种状况。我们 知道 自己只对最流行的项感兴趣，所以我们可以简单地避免加载那些不太有意思的长尾项：

PUT /music/\_mapping/song

{

"properties": {

"tag": {

"type": "string",

"fielddata": { https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

"filter": {

"frequency": { https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

"min": 0.01, https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png

"min\_segment\_size": 500 https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/4.png

}

}

}

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_fielddata_filtering.html#CO220-1) | fielddata 关键字允许我们配置 fielddata 处理该字段的方式。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_fielddata_filtering.html#CO220-2) | frequency 过滤器允许我们基于项频率过滤加载 fielddata。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_fielddata_filtering.html#CO220-3) | 只加载那些至少在本段文档中出现 1% 的项。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/4.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_fielddata_filtering.html#CO220-4) | 忽略任何文档个数小于 500 的段。 |

有了这个映射，只有那些至少在 本段 文档中出现超过 1% 的项才会被加载到内存中。我们也可以指定一个 最大 词频，它可以被用来排除 常用 项，比如 [停用词](https://www.elastic.co/guide/cn/elasticsearch/guide/current/stopwords.html) 。

这种情况下，词频是按照段来计算的。这是实现的一个限制：fielddata 是按段来加载的，所以可见的词频只是该段内的频率。但是，这个限制也有些有趣的特性：它可以让受欢迎的新项迅速提升到顶部。

比如一个新风格的歌曲在一夜之间受大众欢迎，我们可能想要将这种新风格的歌曲标签包括在最受欢迎列表中，但如果我们倚赖对索引做完整的计算获取词频，我们就必须等到新标签变得像 rock 和 electronica ）一样流行。 由于频度过滤的实现方式，新加的标签会很快作为高频标签出现在新段内，也当然会迅速上升到顶部。

min\_segment\_size 参数要求 Elasticsearch 忽略某个大小以下的段。 如果一个段内只有少量文档，它的词频会非常粗略没有任何意义。 小的分段会很快被合并到更大的分段中，某一刻超过这个限制，将会被纳入计算。

提示

通过频次来过滤项并不是唯一的选择，我们也可以使用正则式来决定只加载那些匹配的项。例如，我们可以用 regex 过滤器 处理 twitte 上的消息只将以 # 号开始的标签加载到内存中。 这假设我们使用的分析器会保留标点符号，像 whitespace 分析器。

Fielddata 过滤对内存使用有 巨大的 影响，权衡也是显而易见的：我们实际上是在忽略数据。但对于很多应用，这种权衡是合理的，因为这些数据根本就没有被使用到。内存的节省通常要比包括一个大量而无用的长尾项更为重要。

### 预加载 fielddata

Elasticsearch 加载内存 fielddata 的默认行为是 延迟 加载 。 当 Elasticsearch 第一次查询某个字段时，它将会完整加载这个字段所有 Segment 中的倒排索引到内存中，以便于以后的查询能够获取更好的性能。

对于小索引段来说，这个过程的需要的时间可以忽略。但如果我们有一些 5 GB 的索引段，并希望加载 10 GB 的 fielddata 到内存中，这个过程可能会要数十秒。 已经习惯亚秒响应的用户很难会接受停顿数秒卡着没反应的网站。

有三种方式可以解决这个延时高峰：

* 预加载 fielddata
* 预加载全局序号
* 缓存预热

所有的变化都基于同一概念：预加载 fielddata ，这样在用户进行搜索时就不会碰到延迟高峰。

**预加载 fielddata（Eagerly Loading Fielddata）**

第一个工具称为 预加载 （与默认的 延迟加载相对）。随着新分段的创建（通过刷新、写入或合并等方式）， 启动字段预加载可以使那些对搜索不可见的分段里的 fielddata 提前 加载。

这就意味着首次命中分段的查询不需要促发 fielddata 的加载，因为 fielddata 已经被载入到内存。避免了用户遇到搜索卡顿的情形。

预加载是按字段启用的，所以我们可以控制具体哪个字段可以预先加载：

PUT /music/\_mapping/\_song

{

"tags": {

"type": "string",

"fielddata": {

"loading" : "eager" https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/preload-fielddata.html#CO221-1) | 设置 fielddata.loading: eager 可以告诉 Elasticsearch 预先将此字段的内容载入内存中。 |

Fielddata 的载入可以使用 update-mapping API 对已有字段设置 lazy 或 eager 两种模式。

警告

预加载只是简单的将载入 fielddata 的代价转移到索引刷新的时候，而不是查询时，从而大大提高了搜索体验。

体积大的索引段会比体积小的索引段需要更长的刷新时间。通常，体积大的索引段是由那些已经对查询可见的小分段合并而成的，所以较慢的刷新时间也不是很重要。

**全局序号（Global Ordinals）**

有种可以用来降低字符串 fielddata 内存使用的技术叫做 序号 。

设想我们有十亿文档，每个文档都有自己的 status 状态字段，状态总共有三种： status\_pending 、 status\_published 、 status\_deleted 。如果我们为每个文档都保留其状态的完整字符串形式，那么每个文档就需要使用 14 到 16 字节，或总共 15 GB。

取而代之的是我们可以指定三个不同的字符串，对其排序、编号：0，1，2。

Ordinal | Term

-------------------

0 | status\_deleted

1 | status\_pending

2 | status\_published

序号字符串在序号列表中只存储一次，每个文档只要使用数值编号的序号来替代它原始的值。

Doc | Ordinal

-------------------------

0 | 1 # pending

1 | 1 # pending

2 | 2 # published

3 | 0 # deleted

这样可以将内存使用从 15 GB 降到 1 GB 以下！

但这里有个问题，记得 fielddata 是按分 段 来缓存的。如果一个分段只包含两个状态（ status\_deleted和 status\_published ）。那么结果中的序号（0 和 1）就会与包含所有三个状态的分段不一样。

如果我们尝试对 status 字段运行 terms 聚合，我们需要对实际字符串的值进行聚合，也就是说我们需要识别所有分段中相同的值。一个简单粗暴的方式就是对每个分段执行聚合操作，返回每个分段的字符串值，再将它们归纳得出完整的结果。 尽管这样做可行，但会很慢而且大量消耗 CPU。

取而代之的是使用一个被称为 全局序号 的结构。 全局序号是一个构建在 fielddata 之上的数据结构，它只占用少量内存。唯一值是 跨所有分段 识别的，然后将它们存入一个序号列表中，正如我们描述过的那样。

现在， terms 聚合可以对全局序号进行聚合操作，将序号转换成真实字符串值的过程只会在聚合结束时发生一次。这会将聚合（和排序）的性能提高三到四倍。

**构建全局序号（Building global ordinals）**

当然，天下没有免费的晚餐。 全局序号分布在索引的所有段中，所以如果新增或删除一个分段时，需要对全局序号进行重建。 重建需要读取每个分段的每个唯一项，基数越高（即存在更多的唯一项）这个过程会越长。

全局序号是构建在内存 fielddata 和 doc values 之上的。实际上，它们正是 doc values 性能表现不错的一个主要原因。

和 fielddata 加载一样，全局序号默认也是延迟构建的。首个需要访问索引内 fielddata 的请求会促发全局序号的构建。由于字段的基数不同，这会导致给用户带来显著延迟这一糟糕结果。一旦全局序号发生重建，仍会使用旧的全局序号，直到索引中的分段产生变化：在刷新、写入或合并之后。

**预构建全局序号（Eager global ordinals）**

单个字符串字段 可以通过配置预先构建全局序号：

PUT /music/\_mapping/\_song

{

"song\_title": {

"type": "string",

"fielddata": {

"loading" : "eager\_global\_ordinals" https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/preload-fielddata.html#CO222-1) | 设置 eager\_global\_ordinals 也暗示着 fielddata 是预加载的。 |

正如 fielddata 的预加载一样，预构建全局序号发生在新分段对于搜索可见之前。

注意

序号的构建只被应用于字符串。数值信息（integers（整数）、geopoints（地理经纬度）、dates（日期）等等）不需要使用序号映射，因为这些值自己本质上就是序号映射。

因此，我们只能为字符串字段预构建其全局序号。

也可以对 Doc values 进行全局序号预构建：

PUT /music/\_mapping/\_song

{

"song\_title": {

"type": "string",

"doc\_values": true,

"fielddata": {

"loading" : "eager\_global\_ordinals" https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/preload-fielddata.html#CO223-1) | 这种情况下，fielddata 没有载入到内存中，而是 doc values 被载入到文件系统缓存中。 |

与 fielddata 预加载不一样，预建全局序号会对数据的 实时性 产生影响，构建一个高基数的全局序号会使一个刷新延时数秒。 选择在于是每次刷新时付出代价，还是在刷新后的第一次查询时。如果经常索引而查询较少，那么在查询时付出代价要比每次刷新时要好。如果写大于读，那么在选择在查询时重建全局序号将会是一个更好的选择。

提示

针对实际场景优化全局序号的重建频次。如果我们有高基数字段需要花数秒钟重建，增加 refresh\_interval 的刷新的时间从而可以使我们的全局序号保留更长的有效期，这也会节省 CPU 资源，因为我们重建的频次下降了。

**索引预热器（Index Warmers）**

最后我们谈谈 索引预热器 。预热器早于 fielddata 预加载和全局序号预加载之前出现，它们仍然尤其存在的理由。一个索引预热器允许我们指定一个查询和聚合须要在新分片对于搜索可见之前执行。 这个想法是通过预先填充或 预热缓存 让用户永远无法遇到延迟的波峰。

原来，预热器最重要的用法是确保 fielddata 被预先加载，因为这通常是最耗时的一步。现在可以通过前面讨论的那些技术来更好的控制它，但是预热器还是可以用来预建过滤器缓存，当然我们也还是能选择用它来预加载 fielddata。

让我们注册一个预热器然后解释发生了什么：

PUT /music/\_warmer/warmer\_1 https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

{

"query" : {

"bool" : {

"filter" : {

"bool": {

"should": [ https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

{ "term": { "tag": "rock" }},

{ "term": { "tag": "hiphop" }},

{ "term": { "tag": "electronics" }}

]

}

}

}

},

"aggs" : {

"price" : {

"histogram" : {

"field" : "price", https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png

"interval" : 10

}

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/preload-fielddata.html#CO224-1) | 预热器被关联到索引（ music ）上，使用接入口 \_warmer 以及 ID （ warmer\_1 ）。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/preload-fielddata.html#CO224-2) | 为三种最受欢迎的曲风预建过滤器缓存。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/preload-fielddata.html#CO224-3) | 字段 price 的 fielddata 和全局序号会被预加载。 |

预热器是根据具体索引注册的， 每个预热器都有唯一的 ID ，因为每个索引可能有多个预热器。

然后我们可以指定查询，任何查询。它可以包括查询、过滤器、聚合、排序值、脚本，任何有效的查询表达式都毫不夸张。 这里的目的是想注册那些可以代表用户产生流量压力的查询，从而将合适的内容载入缓存。

当新建一个分段时，Elasticsearch 将会执行注册在预热器中的查询。执行这些查询会强制加载缓存，只有在所有预热器执行完，这个分段才会对搜索可见。

警告

与预加载类似，预热器只是将冷缓存的代价转移到刷新的时候。当注册预热器时，做出明智的决定十分重要。 为了确保每个缓存都被读入，我们 可以 加入上千的预热器，但这也会使新分段对于搜索可见的时间急剧上升。

实际中，我们会选择少量代表大多数用户的查询，然后注册它们。

有些管理的细节（比如获得已有预热器和删除预热器）没有在本小节提到，剩下的详细内容可以参考 [预热器文档（warmers documentation）](https://www.elastic.co/guide/en/elasticsearch/reference/master/indices-warmers.html) 。

### 优化聚合查询

“elasticsearch 里面桶的叫法和 SQL 里面分组的概念是类似的，一个桶就类似 SQL 里面的一个 group，多级嵌套的 aggregation， 类似 SQL 里面的多字段分组（group by field1,field2, …..），注意这里仅仅是概念类似，底层的实现原理是不一样的。 －译者注”

terms 桶基于我们的数据动态构建桶；它并不知道到底生成了多少桶。 大多数时候对单个字段的聚合查询还是非常快的， 但是当需要同时聚合多个字段时，就可能会产生大量的分组，最终结果就是占用 es 大量内存，从而导致 OOM 的情况发生。

假设我们现在有一些关于电影的数据集，每条数据里面会有一个数组类型的字段存储表演该电影的所有演员的名字。

{

"actors" : [

"Fred Jones",

"Mary Jane",

"Elizabeth Worthing"

]

}

如果我们想要查询出演影片最多的十个演员以及与他们合作最多的演员，使用聚合是非常简单的：

{

"aggs" : {

"actors" : {

"terms" : {

"field" : "actors",

"size" : 10

},

"aggs" : {

"costars" : {

"terms" : {

"field" : "actors",

"size" : 5

}

}

}

}

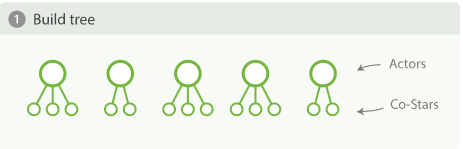
}

}

这会返回前十位出演最多的演员，以及与他们合作最多的五位演员。这看起来是一个简单的聚合查询，最终只返回 50 条数据！

但是， 这个看上去简单的查询可以轻而易举地消耗大量内存，我们可以通过在内存中构建一个树来查看这个 terms 聚合。 actors 聚合会构建树的第一层，每个演员都有一个桶。然后，内套在第一层的每个节点之下， costar 聚合会构建第二层，每个联合出演一个桶，请参见 [图 42 “Build full depth tree”](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_preventing_combinatorial_explosions.html#depth-first-1) 所示。这意味着每部影片会生成 n2 个桶！

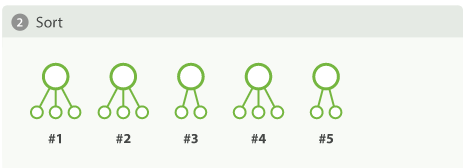
**图 42. Build full depth tree**



用真实点的数据，设想平均每部影片有 10 名演员，每部影片就会生成 102 == 100 个桶。如果总共有 20，000 部影片，粗率计算就会生成 2，000，000 个桶。

现在，记住，聚合只是简单的希望得到前十位演员和与他们联合出演者，总共 50 条数据。为了得到最终的结果，我们创建了一个有 2，000，000 桶的树，然后对其排序，取 top10。 图 [图 43 “Sort tree”](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_preventing_combinatorial_explosions.html#depth-first-2) 和图 [图 44 “Prune tree”](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_preventing_combinatorial_explosions.html#depth-first-3) 对这个过程进行了阐述。

**图 43. Sort tree**



**图 44. Prune tree**



这时我们一定非常抓狂，在 2 万条数据下执行任何聚合查询都是毫无压力的。如果我们有 2 亿文档，想要得到前 100 位演员以及与他们合作最多的 20 位演员，作为查询的最终结果会出现什么情况呢？

可以推测聚合出来的分组数非常大，会使这种策略难以维持。世界上并不存在足够的内存来支持这种不受控制的聚合查询。

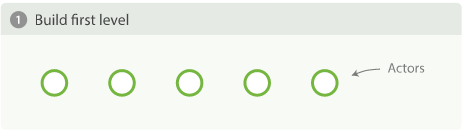
**深度优先与广度优先（Depth-First Versus Breadth-First）**

Elasticsearch 允许我们改变聚合的 集合模式 ，就是为了应对这种状况。 我们之前展示的策略叫做 深度优先 ，它是默认设置， 先构建完整的树，然后修剪无用节点。 深度优先 的方式对于大多数聚合都能正常工作，但对于如我们演员和联合演员这样例子的情形就不太适用。

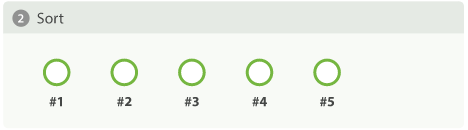
为了应对这些特殊的应用场景，我们应该使用另一种集合策略叫做 广度优先 。这种策略的工作方式有些不同，它先执行第一层聚合， 再 继续下一层聚合之前会先做修剪。 图 [图 45 “Build first level”](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_preventing_combinatorial_explosions.html#breadth-first-1) 和图 [图 47 “Prune first level”](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_preventing_combinatorial_explosions.html#breadth-first-3) 对这个过程进行了阐述。

在我们的示例中， actors 聚合会首先执行，在这个时候，我们的树只有一层，但我们已经知道了前 10 位的演员！这就没有必要保留其他的演员信息，因为它们无论如何都不会出现在前十位中。

**图 45. Build first level**



**图 46. Sort first level**

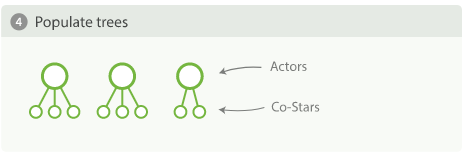


**图 47. Prune first level**



因为我们已经知道了前十名演员，我们可以安全的修剪其他节点。修剪后，下一层是基于 它的 执行模式读入的，重复执行这个过程直到聚合完成，如图 [图 48 “Populate full depth for remaining nodes”](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_preventing_combinatorial_explosions.html#breadth-first-4) 所示。 这种场景下，广度优先可以大幅度节省内存。

**图 48. Populate full depth for remaining nodes**



要使用广度优先，只需简单 的通过参数 collect 开启：

{

"aggs" : {

"actors" : {

"terms" : {

"field" : "actors",

"size" : 10,

"collect\_mode" : "breadth\_first" https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

},

"aggs" : {

"costars" : {

"terms" : {

"field" : "actors",

"size" : 5

}

}

}

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_preventing_combinatorial_explosions.html#CO225-1) | 按聚合来开启 breadth\_first 。 |

广度优先仅仅适用于每个组的聚合数量远远小于当前总组数的情况下，因为广度优先会在内存中缓存裁剪后的仅仅需要缓存的每个组的所有数据，以便于它的子聚合分组查询可以复用上级聚合的数据。

广度优先的内存使用情况与裁剪后的缓存分组数据量是成线性的。对于很多聚合来说，每个桶内的文档数量是相当大的。 想象一种按月分组的直方图，总组数肯定是固定的，因为每年只有12个月，这个时候每个月下的数据量可能非常大。这使广度优先不是一个好的选择，这也是为什么深度优先作为默认策略的原因。

针对上面演员的例子，如果数据量越大，那么默认的使用深度优先的聚合模式生成的总分组数就会非常多，但是预估二级的聚合字段分组后的数据量相比总的分组数会小很多所以这种情况下使用广度优先的模式能大大节省内存，从而通过优化聚合模式来大大提高了在某些特定场景下聚合查询的成功率。

1. 管理、监控和部署

本书大部分介绍了使用 Elasticsearch 作为后端创建应用程序。本章节稍微不同。在这里，你将学习到如何管理 Elasticsearch 自身。Elasticsearch 是一个复杂的软件，有许多可移动组件，大量的 API 设计用来帮助管理你的 Elasticsearch 部署。

在这个章节，我们涵盖三个主题：

* 根据监控你的集群重要数据的统计，去了解哪些行为是正常的，哪些应该引起警告，并解释 Elasticsearch 提供的各种统计信息。
* 部署你的集群到生产环境，包括最佳实践和应该（或不应该！）修改的重要配置。
* 部署后的维护，如 Rolling Restart 或备份你的集群

## 监控

Elasticsearch 经常以多节点集群的方式部署。 有多种 API 让你可以管理和监控集群本身，而不用和集群里存储的数据打交道。

和 Elasticsearch 里绝大多数功能一样，我们有一个总体的设计目标，即任务应该通过 API 执行，而不是通过修改静态的配置文件。这一点在你的集群扩容时尤为重要。即便通过配置管理系统（比如 Puppet，Chef 或者 Ansible），一个简单的 HTTP API 调用，也比往上百台物理设备上推送新配置文件简单多了。

因此，本章将介绍各种可以让你动态调整、调优和调配集群的 API。同时，还会介绍一系列提供集群自身统计数据的 API，你可以用这些接口来监控集群健康状态和性能。

### Marvel 监控

[Marvel](https://www.elastic.co/guide/en/marvel/current/index.html) 让你可以很简单的通过 Kibana 监控 Elasticsearch。你可以实时查看你的集群健康状态和性能，也可以分析过去的集群、索引和节点指标。

虽然你可以通过本章介绍的 API 查看大量的指标数据，但是它们展示的都是当前时间点的即时情况。了解这个瞬间的内存占用比当然很有用，但是了解内存占用比 *随时间轴的趋势* 更加有用。Marvel 会查询并聚合这些数据，你可以通过可视化效果看到自己集群随时间的变化，这样可以很容易的发现发展的趋势。

随着你集群规模的发展，统计 API 的输出内容会复杂得让人完全没法看。当你有一大把节点，比如说一百个，再阅读这个输出的 JSON 就非常乏味了。而 Marvel 可以让你交互式的探索这些数据，更容易于集中关注特定节点或者索引上发生了什么。

Marvel 使用公开的 API，和你自己能找到的一样 — 它没有暴露任何你通过 API 访问不到的统计信息。但是，Marvel 极大的简化了这些统计信息的采集和可视化工作。

Marvel 可以免费使用（包括生产环境上！），所以你现在就开始用起来吧！安装介绍，参阅 [Marvel 入门](https://www.elastic.co/guide/en/marvel/current/getting-started.html)。

### 集群健康

一个 Elasticsearch 集群至少包括一个节点和一个索引。或者它 可能有一百个数据节点、三个单独的主节点，以及一小打客户端节点——这些共同操作一千个索引（以及上万个分片）。

不管集群扩展到多大规模，你都会想要一个快速获取集群状态的途径。Cluster Health API 充当的就是这个角色。你可以把它想象成是在一万英尺的高度鸟瞰集群。它可以告诉你安心吧一切都好，或者警告你集群某个地方有问题。

让我们执行一下 cluster-health API 然后看看响应体是什么样子的：

GET \_cluster/health

和 Elasticsearch 里其他 API 一样，cluster-health 会返回一个 JSON 响应。这对自动化和告警系统来说，非常便于解析。响应中包含了和你集群有关的一些关键信息：

{

"cluster\_name": "elasticsearch\_zach",

"status": "green",

"timed\_out": false,

"number\_of\_nodes": 1,

"number\_of\_data\_nodes": 1,

"active\_primary\_shards": 10,

"active\_shards": 10,

"relocating\_shards": 0,

"initializing\_shards": 0,

"unassigned\_shards": 0

}

响应信息中最重要的一块就是 status 字段。状态可能是下列三个值之一：

green

所有的主分片和副本分片都已分配。你的集群是 100% 可用的。

yellow

所有的主分片已经分片了，但至少还有一个副本是缺失的。不会有数据丢失，所以搜索结果依然是完整的。不过，你的高可用性在某种程度上被弱化。如果 *更多的* 分片消失，你就会丢数据了。把 yellow 想象成一个需要及时调查的警告。

red

至少一个主分片（以及它的全部副本）都在缺失中。这意味着你在缺少数据：搜索只能返回部分数据，而分配到这个分片上的写入请求会返回一个异常。

green/yellow/red 状态是一个概览你的集群并了解眼下正在发生什么的好办法。剩下来的指标给你列出来集群的状态概要：

* number\_of\_nodes 和 number\_of\_data\_nodes 这个命名完全是自描述的。
* active\_primary\_shards 指出你集群中的主分片数量。这是涵盖了所有索引的汇总值。
* active\_shards 是涵盖了所有索引的\_所有\_分片的汇总值，即包括副本分片。
* relocating\_shards 显示当前正在从一个节点迁往其他节点的分片的数量。通常来说应该是 0，不过在 Elasticsearch 发现集群不太均衡时，该值会上涨。比如说：添加了一个新节点，或者下线了一个节点。
* initializing\_shards 是刚刚创建的分片的个数。比如，当你刚创建第一个索引，分片都会短暂的处于 initializing 状态。这通常会是一个临时事件，分片不应该长期停留在 initializing 状态。你还可能在节点刚重启的时候看到 initializing 分片：当分片从磁盘上加载后，它们会从 initializing 状态开始。
* unassigned\_shards 是已经在集群状态中存在的分片，但是实际在集群里又找不着。通常未分配分片的来源是未分配的副本。比如，一个有 5 分片和 1 副本的索引，在单节点集群上，就会有 5 个未分配副本分片。如果你的集群是 red 状态，也会长期保有未分配分片（因为缺少主分片）。

**钻更深点：找到问题索引**

想象一下某天碰到问题了， 而你发现你的集群健康状态看起来像是这样：

{

"cluster\_name": "elasticsearch\_zach",

"status": "red",

"timed\_out": false,

"number\_of\_nodes": 8,

"number\_of\_data\_nodes": 8,

"active\_primary\_shards": 90,

"active\_shards": 180,

"relocating\_shards": 0,

"initializing\_shards": 0,

"unassigned\_shards": 20

}

好了，从这个健康状态里我们能推断出什么来？嗯，我们集群是 red ，意味着我们缺数据（主分片 + 副本分片）了。我们知道我们集群原先有 10 个节点，但是在这个健康状态里列出来的只有 8 个数据节点。有两个数据节点不见了。我们看到有 20 个未分配分片。

这就是我们能收集到的全部信息。那些缺失分片的情况依然是个谜。我们是缺了 20 个索引，每个索引里少 1 个主分片？还是缺 1 个索引里的 20 个主分片？还是 10 个索引里的各 1 主 1 副本分片？具体是哪个索引？

要回答这个问题，我们需要使用 level 参数让 cluster-health 答出更多一点的信息：

GET \_cluster/health?level=indices

这个参数会让 cluster-health API 在我们的集群信息里添加一个索引清单，以及有关每个索引的细节（状态、分片数、未分配分片数等等）：

{

"cluster\_name": "elasticsearch\_zach",

"status": "red",

"timed\_out": false,

"number\_of\_nodes": 8,

"number\_of\_data\_nodes": 8,

"active\_primary\_shards": 90,

"active\_shards": 180,

"relocating\_shards": 0,

"initializing\_shards": 0,

"unassigned\_shards": 20

"indices": {

"v1": {

"status": "green",

"number\_of\_shards": 10,

"number\_of\_replicas": 1,

"active\_primary\_shards": 10,

"active\_shards": 20,

"relocating\_shards": 0,

"initializing\_shards": 0,

"unassigned\_shards": 0

},

"v2": {

"status": "red", https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

"number\_of\_shards": 10,

"number\_of\_replicas": 1,

"active\_primary\_shards": 0,

"active\_shards": 0,

"relocating\_shards": 0,

"initializing\_shards": 0,

"unassigned\_shards": 20 https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

},

"v3": {

"status": "green",

"number\_of\_shards": 10,

"number\_of\_replicas": 1,

"active\_primary\_shards": 10,

"active\_shards": 20,

"relocating\_shards": 0,

"initializing\_shards": 0,

"unassigned\_shards": 0

},

....

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_cluster_health.html#CO298-1) | 我们可以看到 v2 索引就是让集群变 red 的那个索引。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_cluster_health.html#CO298-2) | 由此明确了，20 个缺失分片全部来自这个索引。 |

一旦我们询问要索引的输出，哪个索引有问题立马就很清楚了：v2 索引。我们还可以看到这个索引曾经有 10 个主分片和一个副本，而现在这 20 个分片全不见了。可以推测，这 20 个索引就是位于从我们集群里不见了的那两个节点上。

level 参数还可以接受其他更多选项：

GET \_cluster/health?level=shards

shards 选项会提供一个详细得多的输出，列出每个索引里每个分片的状态和位置。这个输出有时候很有用，但是由于太过详细会比较难用。如果你知道哪个索引有问题了，本章讨论的其他 API 显得更加有用一点。

**阻塞等待状态变化**

当构建单元和集成测试时，或者实现和 Elasticsearch 相关的自动化脚本时，cluster-health API 还有另一个小技巧非常有用。你可以指定一个 wait\_for\_status 参数，它只有在状态达标之后才会返回。比如：

GET \_cluster/health?wait\_for\_status=green

这个调用会 *阻塞* （不给你的程序返回控制权）住直到 cluster-health 变成 green ，也就是说所有主分片和副本分片都分配下去了。这对自动化脚本和测试非常重要。

如果你创建一个索引，Elasticsearch 必须在集群状态中向所有节点广播这个变更。那些节点必须初始化这些新分片，然后响应给主节点说这些分片已经 Started 。这个过程很快，但是因为网络延迟，可能要花 10–20ms。

如果你有个自动化脚本是 (a) 创建一个索引然后 (b) 立刻写入一个文档，这个操作会失败。因为索引还没完全初始化完成。在 (a) 和 (b) 两步之间的时间可能不到 1ms —— 对网络延迟来说这可不够。

比起使用 sleep 命令，直接让你的脚本或者测试使用 wait\_for\_status 参数调用 cluster-health 更好。当索引完全创建好，cluster-health 就会变成 green ，然后这个调用就会把控制权交还给你的脚本，然后你就可以开始写入了。

有效的选项是： green 、 yellow 和 red 。这个调回会在达到你要求（或者『更高』）的状态时返回。比如，如果你要求的是 yellow ，状态变成 yellow 或者 green 都会打开调用。

### 监控单个节点

集群健康 就像是光谱的一端——对集群的所有信息进行高度概述。 而 节点统计值 API 则是在另一端。它提供一个让人眼花缭乱的统计数据的数组，包含集群的每一个节点统计值。

节点统计值 提供的统计值如此之多，在完全熟悉它之前，你可能都搞不清楚哪些指标是最值得关注的。我们将会高亮那些最重要的监控指标（但是我们鼓励你记录接口提供的所有指标——或者用 Marvel ——因为你永远不知道何时需要某个或者另一个值）。

节点统计值 API 可以通过如下命令执行：

GET \_nodes/stats

在输出内容的开头，我们可以看到集群名称和我们的第一个节点：

{

"cluster\_name": "elasticsearch\_zach",

"nodes": {

"UNr6ZMf5Qk-YCPA\_L18BOQ": {

"timestamp": 1408474151742,

"name": "Zach",

"transport\_address": "inet[zacharys-air/192.168.1.131:9300]",

"host": "zacharys-air",

"ip": [

"inet[zacharys-air/192.168.1.131:9300]",

"NONE"

],

...

节点是排列在一个哈希里，以节点的 UUID 作为键名。还显示了节点网络属性的一些信息（比如传输层地址和主机名）。这些值对调试诸如节点未加入集群这类自动发现问题很有用。通常你会发现是端口用错了，或者节点绑定在错误的 IP 地址/网络接口上了。

**索引部分**

索引(indices) 部分列出了这个节点上所有索引的聚合过的统计值 ：

"indices": {

"docs": {

"count": 6163666,

"deleted": 0

},

"store": {

"size\_in\_bytes": 2301398179,

"throttle\_time\_in\_millis": 122850

},

返回的统计值被归入以下部分：

* docs 展示节点内存有多少文档，包括还没有从段里清除的已删除文档数量。
* store 部分显示节点耗用了多少物理存储。这个指标包括主分片和副本分片在内。如果限流时间很大，那可能表明你的磁盘限流设置得过低（在[段和合并](https://www.elastic.co/guide/cn/elasticsearch/guide/current/indexing-performance.html#segments-and-merging)里讨论过）。

"indexing": {

"index\_total": 803441,

"index\_time\_in\_millis": 367654,

"index\_current": 99,

"delete\_total": 0,

"delete\_time\_in\_millis": 0,

"delete\_current": 0

},

"get": {

"total": 6,

"time\_in\_millis": 2,

"exists\_total": 5,

"exists\_time\_in\_millis": 2,

"missing\_total": 1,

"missing\_time\_in\_millis": 0,

"current": 0

},

"search": {

"open\_contexts": 0,

"query\_total": 123,

"query\_time\_in\_millis": 531,

"query\_current": 0,

"fetch\_total": 3,

"fetch\_time\_in\_millis": 55,

"fetch\_current": 0

},

"merges": {

"current": 0,

"current\_docs": 0,

"current\_size\_in\_bytes": 0,

"total": 1128,

"total\_time\_in\_millis": 21338523,

"total\_docs": 7241313,

"total\_size\_in\_bytes": 5724869463

},

* indexing 显示已经索引了多少文档。这个值是一个累加计数器。在文档被删除的时候，数值不会下降。还要注意的是，在发生内部 *索引* 操作的时候，这个值也会增加，比如说文档更新。

还列出了索引操作耗费的时间，正在索引的文档数量，以及删除操作的类似统计值。

* get 显示通过 ID 获取文档的接口相关的统计值。包括对单个文档的 GET 和 HEAD 请求。
* search 描述在活跃中的搜索（ open\_contexts ）数量、查询的总数量、以及自节点启动以来在查询上消耗的总时间。用 query\_time\_in\_millis / query\_total 计算的比值，可以用来粗略的评价你的查询有多高效。比值越大，每个查询花费的时间越多，你应该要考虑调优了。

fetch 统计值展示了查询处理的后一半流程（query-then-fetch 里的 *fetch* ）。如果 fetch 耗时比 query 还多，说明磁盘较慢，或者获取了太多文档，或者可能搜索请求设置了太大的分页（比如， size: 10000 ）。

* merges 包括了 Lucene 段合并相关的信息。它会告诉你目前在运行几个合并，合并涉及的文档数量，正在合并的段的总大小，以及在合并操作上消耗的总时间。

在你的集群写入压力很大时，合并统计值非常重要。合并要消耗大量的磁盘 I/O 和 CPU 资源。如果你的索引有大量的写入，同时又发现大量的合并数，一定要去阅读[索引性能技巧](https://www.elastic.co/guide/cn/elasticsearch/guide/current/indexing-performance.html)。

注意：文档更新和删除也会导致大量的合并数，因为它们会产生最终需要被合并的段 *碎片* 。

"filter\_cache": {

"memory\_size\_in\_bytes": 48,

"evictions": 0

},

"fielddata": {

"memory\_size\_in\_bytes": 0,

"evictions": 0

},

"segments": {

"count": 319,

"memory\_in\_bytes": 65812120

},

...

* filter\_cache 展示了已缓存的过滤器位集合所用的内存数量，以及过滤器被驱逐出内存的次数。过多的驱逐数 *可能* 说明你需要加大过滤器缓存的大小，或者你的过滤器不太适合缓存（比如它们因为高基数而在大量产生，就像是缓存一个 now 时间表达式）。

不过，驱逐数是一个很难评定的指标。过滤器是在每个段的基础上缓存的，而从一个小的段里驱逐过滤器，代价比从一个大的段里要廉价的多。有可能你有很大的驱逐数，但是它们都发生在小段上，也就意味着这些对查询性能只有很小的影响。

把驱逐数指标作为一个粗略的参考。如果你看到数字很大，检查一下你的过滤器，确保他们都是正常缓存的。不断驱逐着的过滤器，哪怕都发生在很小的段上，效果也比正确缓存住了的过滤器差很多。

* field\_data 显示 fielddata 使用的内存， 用以聚合、排序等等。这里也有一个驱逐计数。和 filter\_cache 不同的是，这里的驱逐计数是很有用的：这个数应该或者至少是接近于 0。因为 fielddata 不是缓存，任何驱逐都消耗巨大，应该避免掉。如果你在这里看到驱逐数，你需要重新评估你的内存情况，fielddata 限制，请求语句，或者这三者。
* segments 会展示这个节点目前正在服务中的 Lucene 段的数量。 这是一个重要的数字。大多数索引会有大概 50–150 个段，哪怕它们存有 TB 级别的数十亿条文档。段数量过大表明合并出现了问题（比如，合并速度跟不上段的创建）。注意这个统计值是节点上所有索引的汇聚总数。记住这点。

memory 统计值展示了 Lucene 段自己用掉的内存大小。 这里包括底层数据结构，比如倒排表，字典，和布隆过滤器等。太大的段数量会增加这些数据结构带来的开销，这个内存使用量就是一个方便用来衡量开销的度量值。

**操作系统和进程部分**

OS 和 Process 部分基本是自描述的，不会在细节中展开讲解。它们列出来基础的资源统计值，比如 CPU 和负载。OS 部分描述了整个操作系统，而 Process 部分只显示 Elasticsearch 的 JVM 进程使用的资源情况。

这些都是非常有用的指标，不过通常在你的监控技术栈里已经都测量好了。统计值包括下面这些：

* CPU
* 负载
* 内存使用率
* Swap 使用率
* 打开的文件描述符

**JVM 部分**

jvm 部分包括了运行 Elasticsearch 的 JVM 进程一些很关键的信息。 最重要的，它包括了垃圾回收的细节，这对你的 Elasticsearch 集群的稳定性有着重大影响。

**垃圾回收入门**

在我们描述统计值之前，先上一门速成课程讲解垃圾回收以及它对 Elasticsearch 的影响是非常有用的。如果你对 JVM 的垃圾回收很熟悉，请跳过这段。

Java 是一门 *垃圾回收* 语言，也就是说程序员不用手动管理内存分配和回收。程序员只管写代码，然后 Java 虚拟机（JVM）按需分配内存，然后在稍后不再需要的时候清理这部分内存。

当内存分配给一个 JVM 进程，它是分配到一个大块里，这个块叫做 *堆* 。JVM 把堆分成两组，用 *代* 来表示：

新生代（或者伊甸园）

新实例化的对象分配的空间。新生代空间通常都非常小，一般在 100 MB–500 MB。新生代也包含两个 *幸存* 空间。

老生代

较老的对象存储的空间。这些对象预计将长期留存并持续上很长一段时间。老生代通常比新生代大很多。Elasticsearch 节点可以给老生代用到 30 GB。

当一个对象实例化的时候，它被放在新生代里。当新生代空间满了，就会发生一次新生代垃圾回收（GC）。依然是"存活"状态的对象就被转移到一个幸存区内，而"死掉"的对象被移除。如果一个对象在多次新生代 GC 中都幸存了，它就会被"终身"置于老生代了。

类似的过程在老生代里同样发生：空间满的时候，发生一次垃圾回收，死掉的对象被移除。

不过，天下没有免费的午餐。新生代、老生代的垃圾回收都有一个阶段会“停止时间”。在这段时间里，JVM 字面意义上的停止了程序运行，以便跟踪对象图，收集死亡对象。在这个时间停止阶段，一切都不会发生。请求不被服务，ping 不被回应，分片不被分配。整个世界都真的停止了。

对于新生代，这不是什么大问题；那么小的空间意味着 GC 会很快执行完。但是老生代大很多，而这里面一个慢 GC 可能就意味着 1 秒乃至 15 秒的暂停——对于服务器软件来说这是不可接受的。

JVM 的垃圾回收采用了 *非常* 精密的算法，在减少暂停方面做得很棒。而且 Elasticsearch 非常努力的变成对 *垃圾回收友好* 的程序，比如内部智能的重用对象，重用网络缓冲，以及默认启用 [Doc Values](https://www.elastic.co/guide/cn/elasticsearch/guide/current/docvalues.html) 功能。但最终，GC 的频率和时长依然是你需要去观察的指标。因为它是集群不稳定的头号嫌疑人。

一个经常发生长 GC 的集群就会因为内存不足而处于高负载压力下。这些长 GC 会导致节点短时间内从集群里掉线。这种不稳定会导致分片频繁重定位，因为 Elasticsearch 会尝试保持集群均衡，保证有足够的副本在线。这接着就导致网络流量和磁盘 I/O 的增加。而所有这些都是在你的集群努力服务于正常的索引和查询的同时发生的。

总而言之，长时间 GC 总是不好的，需要尽可能的减少。

因为垃圾回收对 Elasticsearch 是如此重要，你应该非常熟悉 node-stats API 里的这部分内容：

"jvm": {

"timestamp": 1408556438203,

"uptime\_in\_millis": 14457,

"mem": {

"heap\_used\_in\_bytes": 457252160,

"heap\_used\_percent": 44,

"heap\_committed\_in\_bytes": 1038876672,

"heap\_max\_in\_bytes": 1038876672,

"non\_heap\_used\_in\_bytes": 38680680,

"non\_heap\_committed\_in\_bytes": 38993920,

* jvm 部分首先列出一些和 heap 内存使用有关的常见统计值。你可以看到有多少 heap 被使用了，多少被指派了（当前被分配给进程的），以及 heap 被允许分配的最大值。理想情况下，heap\_committed\_in\_bytes 应该等于 heap\_max\_in\_bytes 。如果指派的大小更小，JVM 最终会被迫调整 heap 大小——这是一个非常昂贵的操作。如果你的数字不相等，阅读 [堆内存:大小和交换](https://www.elastic.co/guide/cn/elasticsearch/guide/current/heap-sizing.html)学习如何正确的配置它。

heap\_used\_percent 指标是值得关注的一个数字。Elasticsearch 被配置为当 heap 达到 75% 的时候开始 GC。如果你的节点一直 >= 75%，你的节点正处于 *内存压力* 状态。这是个危险信号，不远的未来可能就有慢 GC 要出现了。

如果 heap 使用率一直 >=85%，你就麻烦了。Heap 在 90–95% 之间，则面临可怕的性能风险，此时最好的情况是长达 10–30s 的 GC，最差的情况就是内存溢出（OOM）异常。

"pools": {

"young": {

"used\_in\_bytes": 138467752,

"max\_in\_bytes": 279183360,

"peak\_used\_in\_bytes": 279183360,

"peak\_max\_in\_bytes": 279183360

},

"survivor": {

"used\_in\_bytes": 34865152,

"max\_in\_bytes": 34865152,

"peak\_used\_in\_bytes": 34865152,

"peak\_max\_in\_bytes": 34865152

},

"old": {

"used\_in\_bytes": 283919256,

"max\_in\_bytes": 724828160,

"peak\_used\_in\_bytes": 283919256,

"peak\_max\_in\_bytes": 724828160

}

}

},

* 新生代(young) 、 幸存区(survivor) 和 老生代(old) 部分分别展示 GC 中每一个代的内存使用情况。这些统计值很方便观察其相对大小，但是在调试问题的时候，通常并不怎么重要。

"gc": {

"collectors": {

"young": {

"collection\_count": 13,

"collection\_time\_in\_millis": 923

},

"old": {

"collection\_count": 0,

"collection\_time\_in\_millis": 0

}

}

}

* gc 部分显示新生代和老生代的垃圾回收次数和累积时间。大多数时候你可以忽略掉新生代的次数：这个数字通常都很大。这是正常的。

与之相反，老生代的次数应该很小，而且 collection\_time\_in\_millis 也应该很小。这些是累积值，所以很难给出一个阈值表示你要开始操心了（比如，一个跑了一整年的节点，即使很健康，也会有一个比较大的计数）。这就是像 Marvel 这类工具很有用的一个原因。GC 计数的 *时间趋势* 是个重要的考虑因素。

GC 花费的时间也很重要。比如，在索引文档时，一系列垃圾生成了。这是很常见的情况，每时每刻都会导致 GC。这些 GC 绝大多数时候都很快，对节点影响很小：新生代一般就花一两毫秒，老生代花一百多毫秒。这些跟 10 秒级别的 GC 是很不一样的。

我们的最佳建议是定期收集 GC 计数和时长（或者使用 Marvel）然后观察 GC 频率。你也可以开启慢 GC 日志记录，在 [日志记录](https://www.elastic.co/guide/cn/elasticsearch/guide/current/logging.html) 小节已经讨论过。

**线程池部分**[**编辑**](https://github.com/elasticsearch-cn/elasticsearch-definitive-guide/edit/cn/500_Cluster_Admin/30_node_stats.asciidoc)

Elasticsearch 在内部维护了线程池。 这些线程池相互协作完成任务，有必要的话相互间还会传递任务。通常来说，你不需要配置或者调优线程池，不过查看它们的统计值有时候还是有用的，可以洞察你的集群表现如何。

这有一系列的线程池，但以相同的格式输出：

"index": {

"threads": 1,

"queue": 0,

"active": 0,

"rejected": 0,

"largest": 1,

"completed": 1

}

每个线程池会列出已配置的线程数量（ threads ），当前在处理任务的线程数量（ active ），以及在队列中等待处理的任务单元数量（ queue ）。

如果队列中任务单元数达到了极限，新的任务单元会开始被拒绝，你会在 rejected 统计值上看到它反映出来。这通常是你的集群在某些资源上碰到瓶颈的信号。因为队列满意味着你的节点或集群在用最高速度运行，但依然跟不上工作的蜂拥而入。

**批量操作的被拒绝数**

如果你碰到了队列被拒，一般来说都是批量索引请求导致的。 通过并发导入程序发送大量批量请求非常简单。越多越好嘛，对不？

事实上，每个集群都有它能处理的请求上限。一旦这个阈值被超过，队列会很快塞满，然后新的批量请求就被拒绝了。

这是一件 \_好事情\_ 。队列的拒绝在回压方面是有用的。它们让你知道你的集群已经在最大容量了。这比把数据塞进内存队列要来得好。增加队列大小并不能增加性能，它只是隐藏了问题。当你的集群只能每秒钟处理 10000 个文档的时候，无论队列是 100 还是 10000000 都没关系——你的集群还是只能每秒处理 10000 个文档。

队列只是隐藏了性能问题，而且带来的是真实的数据丢失的风险。在队列里的数据都是还没处理的，如果节点挂掉，这些请求都会永久的丢失。此外，队列还要消耗大量内存，这也是不理想的。

在你的应用中，优雅的处理来自满载队列的回压，才是更好的选择。当你收到拒绝响应的时候，你应该采取如下几步：

1. 暂停导入线程 3–5 秒。
2. 从批量操作的响应里提取出来被拒绝的操作。因为可能很多操作还是成功的。响应会告诉你哪些成功，哪些被拒绝了。
3. 发送一个新的批量请求，只包含这些被拒绝过的操作。
4. 如果依然碰到拒绝，再次从步骤 1 开始。

通过这个流程，你的代码可以很自然的适应你集群的负载，做到自动回压。

拒绝不是错误：它们只是意味着你要稍后重试。

这里的一系列的线程池，大多数你可以忽略，但是有一小部分还是值得关注的：

indexing

普通的索引请求的线程池

bulk

批量请求，和单条的索引请求不同的线程池

get

Get-by-ID 操作

search

所有的搜索和查询请求

merging

专用于管理 Lucene 合并的线程池

**文件系统和网络部分**

继续向下阅读 node-stats API，你会看到一串和你的文件系统相关的统计值：可用空间，数据目录路径，磁盘 I/O 统计值，等等。如果你没有监控磁盘可用空间的话，可以从这里获取这些统计值。磁盘 I/O 统计值也很方便，不过通常那些更专门的命令行工具（比如 iostat ）会更有用些。

显然，Elasticsearch 在磁盘空间满的时候很难运行——所以请确保不会这样。

还有两个跟 网络统计值相关的部分：

"transport": {

"server\_open": 13,

"rx\_count": 11696,

"rx\_size\_in\_bytes": 1525774,

"tx\_count": 10282,

"tx\_size\_in\_bytes": 1440101928

},

"http": {

"current\_open": 4,

"total\_opened": 23

},

* transport 显示和 *传输地址* 相关的一些基础统计值。包括节点间的通信（通常是 9300 端口）以及任意传输客户端或者节点客户端的连接。如果看到这里有很多连接数不要担心；Elasticsearch 在节点之间维护了大量的连接。
* http 显示 HTTP 端口（通常是 9200）的统计值。如果你看到 total\_opened 数很大而且还在一直上涨，这是一个明确信号，说明你的 HTTP 客户端里有没启用 keep-alive 长连接的。持续的 keep-alive 长连接对性能很重要，因为连接、断开套接字是很昂贵的（而且浪费文件描述符）。请确认你的客户端都配置正确。

**断路器**

终于，我们到了最后一段：跟 fielddata 断路器（在 [断路器](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_limiting_memory_usage.html#circuit-breaker) 介绍过）相关的统计值：

"fielddata\_breaker": {

"maximum\_size\_in\_bytes": 623326003,

"maximum\_size": "594.4mb",

"estimated\_size\_in\_bytes": 0,

"estimated\_size": "0b",

"overhead": 1.03,

"tripped": 0

}

这里你可以看到断路器的最大值（比如，一个请求申请更多的内存时会触发断路器）。这个部分还会让你知道断路器被触发了多少次，以及当前配置的间接开销。间接开销用来铺垫评估，因为有些请求比其他请求更难评估。

主要需要关注的是 tripped 指标。如果这个数字很大或者持续上涨，这是一个信号，说明你的请求需要优化，或者你需要添加更多内存（单机上添加，或者通过添加新节点的方式）。

### 集群统计

集群统计 API 提供了和 节点统计 相似的输出。 但有一个重要的区别：节点统计显示的是每个节点上的统计值，而 集群统计 展示的是对于单个指标，所有节点的总和值。

这里面提供一些很值得一看的统计值。比如说你可以看到，整个集群用了 50% 的堆内存，或者说过滤器缓存的驱逐情况不严重。这个接口主要用途是提供一个比 集群健康 更详细、但又没有 节点统计 那么详细的快速概览。对于非常大的集群来说也很有用，因为那时候 节点统计 的输出已经非常难于阅读了。

这个 API 可以像下面这样调用：

GET \_cluster/stats

### 索引统计

到目前为止，我们看到的都是以 *节点为中心* 的统计值： 节点有多少内存？用了多少 CPU ？正在服务多少个搜索？

有时候从 *索引为中心* 的角度看统计值也很有用：*这个索引* 收到了多少个搜索请求？*那个索引* 获取文档耗费了多少时间？

要做到这点，选择你感兴趣的索引 （ 或者多个索引 ）然后执行一个索引级别的 统计 API：

GET my\_index/\_stats https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

GET my\_index,another\_index/\_stats https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

GET \_all/\_stats https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_index_stats.html#CO299-1) | 统计 my\_index 索引。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_index_stats.html#CO299-2) | 使用逗号分隔索引名可以请求多个索引统计值。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_index_stats.html#CO299-3) | 使用特定的 \_all 可以请求全部索引的统计值 |

返回的统计信息和 节点统计 的输出很相似：search 、 fetch 、 get 、 index 、 bulk 、 segment counts 等等。

索引为中心的统计在有些时候很有用，比如辨别或验证集群中的 *热门* 索引，或者试图找出某些索引比其他索引更快或者更慢的原因。

实践中，节点为中心的统计还是显得更有用些。瓶颈往往是针对整个节点而言，而不是对于单个索引。因为索引一般是分布在多个节点上的，这导致以索引为中心的统计值通常不是很有用，因为它们是从不同环境的物理机器上汇聚的数据。

索引为中心的统计作为一个有用的工具可以保留在你的技能表里，但是通常它不会是第一个用的上的工具。

### 等待中的任务

有一些任务只能由主节点去处理，比如创建一个新的 索引或者在集群中移动分片。由于一个集群中只能有一个主节点，所以只有这一节点可以处理集群级别的元数据变动。在 99.9999% 的时间里，这不会有什么问题。元数据变动的队列基本上保持为零。

在一些 罕见 的集群里，元数据变动的次数比主节点能处理的还快。这会导致等待中的操作会累积成队列。

等待中的任务 API 会给你展示队列中（如果有的话）等待的集群级别的元数据变更操作：

GET \_cluster/pending\_tasks

通常，响应都是像这样的：

{

"tasks": []

}

这意味着没有等待中的任务。如果你有一个罕见的集群在主节点出现瓶颈了，等待中的任务列表可能会像这样：

{

"tasks": [

{

"insert\_order": 101,

"priority": "URGENT",

"source": "create-index [foo\_9], cause [api]",

"time\_in\_queue\_millis": 86,

"time\_in\_queue": "86ms"

},

{

"insert\_order": 46,

"priority": "HIGH",

"source": "shard-started ([foo\_2][1], node[tMTocMvQQgGCkj7QDHl3OA], [P],

s[INITIALIZING]), reason [after recovery from gateway]",

"time\_in\_queue\_millis": 842,

"time\_in\_queue": "842ms"

},

{

"insert\_order": 45,

"priority": "HIGH",

"source": "shard-started ([foo\_2][0], node[tMTocMvQQgGCkj7QDHl3OA], [P],

s[INITIALIZING]), reason [after recovery from gateway]",

"time\_in\_queue\_millis": 858,

"time\_in\_queue": "858ms"

}

]

}

可以看到任务都被指派了优先级（ 比如说 URGENT 要比 HIGH 更早的处理 ），任务插入的次序、操作进入队列多久，以及打算处理什么。在上面的列表中，有一个 创建索引(create-index) 和两个 启动分片(shard-started) 的操作在等待。

**什么时候应该担心等待中的任务？**

就像曾经提到过的，主节点很少会成为集群的瓶颈。唯一可能成为瓶颈的是集群状态非常大 *而且* 更新频繁。

例如，如果你允许客户按照他们的意愿创建任意的动态字段，而且每个客户每天都有一个独立索引，那么你的集群状态会变得非常大。集群状态包括 ( 但不限于 ) 所有索引及其类型，以及每个索引的全部字段。

所以如果你有 100000 客户，然后每个客户平均有 1000 个字段，而且数据有 90 天的保留期—这就有九十亿个字段需要保存在集群状态中。不管它何时发生变更，所有的节点都需要被通知。

主节点必须处理这些变动，这需要不小的 CPU 开销，加上推送更新的集群状态到所有节点的网络开销。

这就是那些可以看到集群状态操作队列上涨的集群。没有简单的办法可以解决这个问题，不过你有三个选择：

* 使用一个更强大的主节点。不幸的是，这种垂直扩展只是延迟这种必然结果出现而已。
* 通过某些方式限定文档的动态性质来限制集群状态的大小。
* 到达某个阈值后组建另外一个集群。

### cat API

如果经常在命令行环境下工作，cat API 对你会非常有用。 用 Linux 的 cat 命令命名，这些 API 也就设计成像 \*nix 命令行工具一样工作了。

他们提供的统计和前面已经讨论过的 API ( 健康、节点统计 等等 ) 是一样的。但是输出以表格的形式提供，而不是 JSON。对于系统管理员来说这是 *非常* 方便的，你仅仅想浏览一遍集群或者找出内存使用偏高的节点而已。

通过 GET 请求发送 cat 命名可以列出所有可用的 API：

GET /\_cat

=^.^=

/\_cat/allocation

/\_cat/shards

/\_cat/shards/{index}

/\_cat/master

/\_cat/nodes

/\_cat/indices

/\_cat/indices/{index}

/\_cat/segments

/\_cat/segments/{index}

/\_cat/count

/\_cat/count/{index}

/\_cat/recovery

/\_cat/recovery/{index}

/\_cat/health

/\_cat/pending\_tasks

/\_cat/aliases

/\_cat/aliases/{alias}

/\_cat/thread\_pool

/\_cat/plugins

/\_cat/fielddata

/\_cat/fielddata/{fields}

许多 API 看起来很熟悉了 ( 是的，顶上还有一只猫:) )。让我们看看 cat 的健康检查 API：

GET /\_cat/health

1408723713 12:08:33 elasticsearch\_zach yellow 1 1 114 114 0 0 114

首先你会注意到的是响应是表格样式的纯文本，而不是 JSON。其次你会注意到各列默认是没有表头的。这都是模仿 \*nix 工具设计的，因为它假设一旦你对输出熟悉了，你就再也不想看见表头了。

要启用表头，添加 ?v 参数即可：

GET /\_cat/health?v

epoch time cluster status node.total node.data shards pri relo init

1408[..] 12[..] el[..] 1 1 114 114 0 0 114

unassign

嗯，好多了。我们现在看到 时间戳、集群名称、状态、集群中节点的数量等等—所有信息和 集群健康 API 返回的都一样。

让我们再看看 cat API 里面的 节点统计 ：

GET /\_cat/nodes?v

host ip heap.percent ram.percent load node.role master name

zacharys-air 192.168.1.131 45 72 1.85 d \* Zach

我们看到集群里节点的一些统计，不过和完整的 节点统计 输出相比而言是非常基础的。你可以包含更多的指标，但是比起查阅文档，让我们直接问 cat API 有哪些可用的吧。

你可以过对任意 API 添加 ?help 参数来做到这点：

GET /\_cat/nodes?help

id | id,nodeId | unique node id

pid | p | process id

host | h | host name

ip | i | ip address

port | po | bound transport port

version | v | es version

build | b | es build hash

jdk | j | jdk version

disk.avail | d,disk,diskAvail | available disk space

heap.percent | hp,heapPercent | used heap ratio

heap.max | hm,heapMax | max configured heap

ram.percent | rp,ramPercent | used machine memory ratio

ram.max | rm,ramMax | total machine memory

load | l | most recent load avg

uptime | u | node uptime

node.role | r,role,dc,nodeRole | d:data node, c:client node

master | m | m:master-eligible, \*:current master

...

...

( 注意这个输出为了页面简洁而被截断了 )。

第一列显示完整的名称，第二列显示缩写，第三列提供了关于这个参数的简介。现在我们知道了一些列名了，我们可以用 ?h 参数来明确指定显示这些指标：

GET /\_cat/nodes?v&h=ip,port,heapPercent,heapMax

ip port heapPercent heapMax

192.168.1.131 9300 53 990.7mb

因为 cat API 试图像 \*nix 工具一样工作，你可以使用管道命令将结果传递给其他工具，比如 sort 、 grep 或者 awk 。例如，通过以下方式可以找到集群中最大的索引：

% curl 'localhost:9200/\_cat/indices?bytes=b' | sort -rnk8

yellow test\_names 5 1 3476004 0 376324705 376324705

yellow .marvel-2014.08.19 1 1 263878 0 160777194 160777194

yellow .marvel-2014.08.15 1 1 234482 0 143020770 143020770

yellow .marvel-2014.08.09 1 1 222532 0 138177271 138177271

yellow .marvel-2014.08.18 1 1 225921 0 138116185 138116185

yellow .marvel-2014.07.26 1 1 173423 0 132031505 132031505

yellow .marvel-2014.08.21 1 1 219857 0 128414798 128414798

yellow .marvel-2014.07.27 1 1 75202 0 56320862 56320862

yellow wavelet 5 1 5979 0 54815185 54815185

yellow .marvel-2014.07.28 1 1 57483 0 43006141 43006141

yellow .marvel-2014.07.21 1 1 31134 0 27558507 27558507

yellow .marvel-2014.08.01 1 1 41100 0 27000476 27000476

yellow kibana-int 5 1 2 0 17791 17791

yellow t 5 1 7 0 15280 15280

yellow website 5 1 12 0 12631 12631

yellow agg\_analysis 5 1 5 0 5804 5804

yellow v2 5 1 2 0 5410 5410

yellow v1 5 1 2 0 5367 5367

yellow bank 1 1 16 0 4303 4303

yellow v 5 1 1 0 2954 2954

yellow p 5 1 2 0 2939 2939

yellow b0001\_072320141238 5 1 1 0 2923 2923

yellow ipaddr 5 1 1 0 2917 2917

yellow v2a 5 1 1 0 2895 2895

yellow movies 5 1 1 0 2738 2738

yellow cars 5 1 0 0 1249 1249

yellow wavelet2 5 1 0 0 615 615

通过添加 ?bytes=b ，我们关闭了人类可读的数字格式化，强制它们以字节数输出。随后通过管道命令将输出传递给 sort 让索引按大小（ 第八列 ）排序

不幸的是，你会注意到 Marval 索引也出现在结果中，但是我们目前并不真正在意这些索引。让我们把结果传递给 grep 命令来移除提到 Marval 的数据：

% curl 'localhost:9200/\_cat/indices?bytes=b' | sort -rnk8 | grep -v marvel

yellow test\_names 5 1 3476004 0 376324705 376324705

yellow wavelet 5 1 5979 0 54815185 54815185

yellow kibana-int 5 1 2 0 17791 17791

yellow t 5 1 7 0 15280 15280

yellow website 5 1 12 0 12631 12631

yellow agg\_analysis 5 1 5 0 5804 5804

yellow v2 5 1 2 0 5410 5410

yellow v1 5 1 2 0 5367 5367

yellow bank 1 1 16 0 4303 4303

yellow v 5 1 1 0 2954 2954

yellow p 5 1 2 0 2939 2939

yellow b0001\_072320141238 5 1 1 0 2923 2923

yellow ipaddr 5 1 1 0 2917 2917

yellow v2a 5 1 1 0 2895 2895

yellow movies 5 1 1 0 2738 2738

yellow cars 5 1 0 0 1249 1249

yellow wavelet2 5 1 0 0 615 615

瞧！在传递给 grep ( 通过 -v 来过滤掉不需要匹配的数据 ) 之后，我们得到了一个没有 Marval 混杂的索引排序列表了。

这只是命令行上 cat 的灵活性的一个简单示例。一旦你习惯了使用 cat ，你会发现它和其他所有 \*nix 工具一样并且开始疯狂的使用管道、排序和过滤。如果你是一个系统管理员并且永远都是 SSH 登录到设备上，那么当然要花些时间来熟悉 cat API 了。

## 部署

如果你按照书中步骤做到了这一步，希望你已经学到了一两件关于 Elasticsearch 的事情并且准备把你的集群部署到生产环境。 这一章不是在生产中运行集群的详尽指南，但是它涵盖了集群上线之前需要考虑的关键事项。

主要包括三个方面：

* 后勤方面的考虑，如硬件和部署策略的建议
* 更适合于生产环境的配置更改
* 部署后的考虑，例如安全，最大限度的索引性能和备份

### 硬件

按照正常的流程， 你可能已经在自己的笔记本电脑或集群上使用了 Elasticsearch。 但是当要部署 Elasticsearch 到生产环境时，有一些建议是你需要考虑的。这里没有什么必须要遵守的准则，Elasticsearch 被用于在众多的机器上处理各种任务。基于我们在生产环境使用 Elasticsearch 集群的经验，这些建议可以为你提供一个好的起点。

**内存**

如果有一种资源是最先被耗尽的，它可能是内存。 排序和聚合都很耗内存，所以有足够的堆空间来应付它们是很重要的。即使堆空间是比较小的时候， 也能为操作系统文件缓存提供额外的内存。因为 Lucene 使用的许多数据结构是基于磁盘的格式，Elasticsearch 利用操作系统缓存能产生很大效果。

64 GB 内存的机器是非常理想的， 但是32 GB 和16 GB 机器也是很常见的。少于8 GB 会适得其反（你最终需要很多很多的小机器），大于64 GB 的机器也会有问题， 我们将在 [堆内存:大小和交换](https://www.elastic.co/guide/cn/elasticsearch/guide/current/heap-sizing.html) 中讨论。

**CPUs**

大多数 Elasticsearch 部署往往对 CPU 要求不高。因此， 相对其它资源，具体配置多少个（CPU）不是那么关键。你应该选择具有多个内核的现代处理器，常见的集群使用两到八个核的机器。

如果你要在更快的 CPUs 和更多的核心之间选择，选择更多的核心更好。多个内核提供的额外并发远胜过稍微快一点点的时钟频率。

**硬盘**

硬盘对所有的集群都很重要， 对大量写入的集群更是加倍重要（例如那些存储日志数据的）。硬盘是服务器上最慢的子系统，这意味着那些写入量很大的集群很容易让硬盘饱和，使得它成为集群的瓶颈。

如果你负担得起 SSD，它将远远超出任何旋转介质（注：机械硬盘，磁带等）。 基于 SSD 的节点，查询和索引性能都有提升。如果你负担得起，SSD 是一个好的选择。

**检查你的 I/O 调度程序**

如果你正在使用 SSDs，确保你的系统 I/O 调度程序是配置正确的。 当你向硬盘写数据，I/O 调度程序决定何时把数据实际发送到硬盘。 大多数默认 \*nix 发行版下的调度程序都叫做 `cfq`（完全公平队列）。

调度程序分配 *时间片* 到每个进程。并且优化这些到硬盘的众多队列的传递。但它是为旋转介质优化的： 机械硬盘的固有特性意味着它写入数据到基于物理布局的硬盘会更高效。

这对 SSD 来说是低效的，尽管这里没有涉及到机械硬盘。但是，deadline 或者 noop 应该被使用。deadline 调度程序基于写入等待时间进行优化， noop 只是一个简单的 FIFO 队列。

这个简单的更改可以带来显著的影响。仅仅是使用正确的调度程序，我们看到了500倍的写入能力提升。

如果你使用旋转介质，尝试获取尽可能快的硬盘（高性能服务器硬盘，15k RPM 驱动器）。

使用 RAID 0 是提高硬盘速度的有效途径，对机械硬盘和 SSD 来说都是如此。没有必要使用镜像或其它 RAID 变体，因为高可用已经通过 replicas 内建于 Elasticsearch 之中。

最后，避免使用网络附加存储（NAS）。人们常声称他们的 NAS 解决方案比本地驱动器更快更可靠。除却这些声称， 我们从没看到 NAS 能配得上它的大肆宣传。NAS 常常很慢，显露出更大的延时和更宽的平均延时方差，而且它是单点故障的。

**网络**

快速可靠的网络显然对分布式系统的性能是很重要的 。 低延时能帮助确保节点间能容易的通讯，大带宽能帮助分片移动和恢复。现代数据中心网络（1 GbE, 10 GbE）对绝大多数集群都是足够的。

即使数据中心们近在咫尺，也要避免集群跨越多个数据中心。绝对要避免集群跨越大的地理距离。

Elasticsearch 假定所有节点都是平等的--并不会因为有一半的节点在150ms 外的另一数据中心而有所不同。更大的延时会加重分布式系统中的问题而且使得调试和排错更困难。

和 NAS 的争论类似，每个人都声称他们的数据中心间的线路都是健壮和低延时的。这是真的--直到它不是时（网络失败终究是会发生的，你可以相信它）。 从我们的经验来看，处理跨数据中心集群的麻烦事是根本不值得的。

**总则**

获取真正的高配机器在今天是可能的： 成百 GB 的 RAM 和几十个 CPU 核心。 反之，在云平台上串联起成千的小虚拟机也是可能的，例如 EC2。哪种方式是最好的?

通常，选择中配或者高配机器更好。避免使用低配机器， 因为你不会希望去管理拥有上千个节点的集群，而且在这些低配机器上运行 Elasticsearch 的开销也是显著的。

与此同时，避免使用真正的高配机器。它们通常会导致资源使用不均衡（例如，所有的内存都被使用，但 CPU 却没有）而且在单机上运行多个节点时，会增加逻辑复杂度。

### Java 虚拟机

你应该始终运行最新版本的 Java 虚拟机（ JVM ）， 除非 Elasticsearch 网站上另有说明。Elasticsearch， 特别是 Lucene，是一个高要求的软件。Lucene 的单元测试和集成测试经常暴露出 JVM 本身的 bug。这些 bug 的范围从轻微的麻烦到严重段错误，所以，最好尽可能的使用最新版本的 JVM。

Java 8 强烈优先选择于 Java 7。不再支持 Java 6。Oracle 或者 OpenJDK 是可以接受的，它们在性能和稳定性也差不多。

如果你的应用程序是用 Java 编写并正在使用传输客户端（注：Transport Client，下同）或节点客户端（注：Node Client，下同），请确保运行你应用程序的 JVM 和服务器的 JVM 是完全一样的。 在 Elasticsearch 的几个地方，使用 Java 的本地序列化（ IP 地址、异常等等）。不幸的是，Oracle 的 JVM 在几个小版本之间有修改序列化格式，从而导致奇怪的错误。 这种情况很少见，但最佳实践是客户端和服务器使用相同版本 JVM。

**请不要调整 JVM 设置**

JVM 暴露出几十个（甚至数百）的设置、参数和配置。 它们允许你进行微调 JVM 几乎是每一个方面。 当遇到一个旋钮，要打开它是人的本性。我们恳求你压制这个本性，而 *不要* 去调整 JVM 参数。Elasticsearch 是复杂的软件，并且我们根据多年的实际使用情况调整了当前 JVM 设置。 它很容易开始转动旋钮，并产生难以衡量的、未知的影响，并最终使集群进入一个缓慢的、不稳定的混乱的效果。当调试集群时，第一步往往是去除所有的自定义配置。多数情况下，仅此就可以恢复稳定和性能。

### Transport Client 与 Node Client

如果你使用的是 Java，你可能想知道何时使用传输客户端（注：Transport Client，下同）与节点客户端（注：Node Client，下同）。 在书的开头所述， 传输客户端作为一个集群和应用程序之间的通信层。它知道 API 并能自动帮你在节点之间轮询，帮你嗅探集群等等。但它是集群 *外部的* ，和 REST 客户端类似。

另一方面，节点客户端，实际上是一个集群中的节点（但不保存数据，不能成为主节点）。因为它是一个节点，它知道整个集群状态（所有节点驻留，分片分布在哪些节点，等等）。 这意味着它可以执行 APIs 但少了一个网络跃点。

这里有两个客户端案例的使用情况：

* 如果要将应用程序和 Elasticsearch 集群进行解耦，传输客户端是一个理想的选择。例如，如果您的应用程序需要快速的创建和销毁到集群的连接，传输客户端比节点客户端”轻”，因为它不是一个集群的一部分。

类似地，如果您需要创建成千上万的连接，你不想有成千上万节点加入集群。传输客户端（ TC ）将是一个更好的选择。

* 另一方面，如果你只需要有少数的、长期持久的对象连接到集群，客户端节点可以更高效，因为它知道集群的布局。但是它会使你的应用程序和集群耦合在一起，所以从防火墙的角度，它可能会构成问题。

### 配置管理

如果你已经使用配置管理（ Puppet，Chef，Ansible），则可以跳过此提示。

如果你没有使用配置管理工具，那么应该注意了！通过 parallel-ssh 管理少量服务器现在可能正常工作，但伴随着集群的增长它将成为一场噩梦。 在不犯错误的情况下手动编辑 30 个配置文件几乎是不可能的。

配置管理工具通过自动化更改配置的过程保持集群的一致性。这可能需要一点时间来建立和学习，但它本身，随着时间的推移会有丰厚的回报。

### 重要配置的修改

Elasticsearch 已经有了 *很好* 的默认值， 特别是涉及到性能相关的配置或者选项。 如果你有疑问，最好就不要动它。我们已经目睹了数十个因为错误的设置而导致毁灭的集群， 因为它的管理者总认为改动一个配置或者选项就可以带来 100 倍的提升。

注意

请阅读整节文章，所有的配置项都同等重要，和描述顺序无关，请阅读所有的配置选项，并应用到你的集群中。

其它数据库可能需要调优，但总得来说，Elasticsearch 不需要。 如果你遇到了性能问题，解决方法通常是更好的数据布局或者更多的节点。 在 Elasticsearch 中很少有“神奇的配置项”， 如果存在，我们也已经帮你优化了！

另外，有些 *逻辑上的* 配置在生产环境中是应该调整的。 这些调整可能会让你的工作更加轻松，又或者因为没办法设定一个默认值（它取决于你的集群布局）。

**指定名字**

Elasticsearch 默认启动的集群名字叫 elasticsearch 。 你最好给你的生产环境的集群改个名字，改名字的目的很简单， 就是防止某人的笔记本电脑加入了集群这种意外。简单修改成 elasticsearch\_production会很省心。

你可以在你的 elasticsearch.yml 文件中修改：

cluster.name: elasticsearch\_production

同样，最好也修改你的节点名字。就像你现在可能发现的那样， Elasticsearch 会在你的节点启动的时候随机给它指定一个名字。你可能会觉得这很有趣，但是当凌晨 3 点钟的时候， 你还在尝试回忆哪台物理机是 Tagak the Leopard Lord 的时候，你就不觉得有趣了。

更重要的是，这些名字是在启动的时候产生的，每次启动节点， 它都会得到一个新的名字。这会使日志变得很混乱，因为所有节点的名称都是不断变化的。

这可能会让你觉得厌烦，我们建议给每个节点设置一个有意义的、清楚的、描述性的名字，同样你可以在 elasticsearch.yml 中配置：

node.name: elasticsearch\_005\_data

**路径**

默认情况下， Elasticsearch 会把插件、日志以及你最重要的数据放在安装目录下。这会带来不幸的事故， 如果你重新安装 Elasticsearch 的时候不小心把安装目录覆盖了。如果你不小心，你就可能把你的全部数据删掉了。

不要笑，这种情况，我们见过很多次了。

最好的选择就是把你的数据目录配置到安装目录以外的地方， 同样你也可以选择转移你的插件和日志目录。

可以更改如下：

path.data: /path/to/data1,/path/to/data2 https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

# Path to log files:

path.logs: /path/to/logs

# Path to where plugins are installed:

path.plugins: /path/to/plugins

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/important-configuration-changes.html#CO300-1) | 注意：你可以通过逗号分隔指定多个目录。 |

数据可以保存到多个不同的目录， 如果将每个目录分别挂载不同的硬盘，这可是一个简单且高效实现一个软磁盘阵列（ RAID 0 ）的办法。Elasticsearch 会自动把条带化（注：RAID 0 又称为 Stripe（条带化），在磁盘阵列中,数据是以条带的方式贯穿在磁盘阵列所有硬盘中的） 数据分隔到不同的目录，以便提高性能。

警告

**多个数据路径的安全性和性能**

如同任何磁盘阵列（ RAID 0 ）的配置，只有单一的数据拷贝保存到硬盘驱动器。如果你失去了一个硬盘驱动器，你 *肯定* 会失去该计算机上的一部分数据。 运气好的话你的副本在集群的其他地方，可以用来恢复数据和最近的备份。

Elasticsearch 试图将全部的条带化分片放到单个驱动器来保证最小程度的数据丢失。这意味着 分片 0 将完全被放置在单个驱动器上。 Elasticsearch 没有一个条带化的分片跨越在多个驱动器，因为一个驱动器的损失会破坏整个分片。

这对性能产生的影响是：如果您添加多个驱动器来提高一个单独索引的性能，可能帮助不大，因为 大多数节点只有一个分片和这样一个积极的驱动器。多个数据路径只是帮助如果你有许多索引／分片在单个节点上。

多个数据路径是一个非常方便的功能，但到头来，Elasticsearch 并不是软磁盘阵列（ software RAID ）的软件。如果你需要更高级的、稳健的、灵活的配置， 我们建议你使用软磁盘阵列（ software RAID ）的软件，而不是多个数据路径的功能。

**最小主节点数**

minimum\_master\_nodes 设定对你的集群的稳定 *极其* 重要。 当你的集群中有两个 masters（注：主节点）的时候，这个配置有助于防止 *脑裂* ，一种两个主节点同时存在于一个集群的现象。

如果你的集群发生了脑裂，那么你的集群就会处在丢失数据的危险中，因为主节点被认为是这个集群的最高统治者，它决定了什么时候新的索引可以创建，分片是如何移动的等等。如果你有 *两个* masters 节点， 你的数据的完整性将得不到保证，因为你有两个节点认为他们有集群的控制权。

这个配置就是告诉 Elasticsearch 当没有足够 master 候选节点的时候，就不要进行 master 节点选举，等 master 候选节点足够了才进行选举。

此设置应该始终被配置为 master 候选节点的法定个数（大多数个）。法定个数就是 ( master 候选节点个数 / 2) + 1 。 这里有几个例子：

* 如果你有 10 个节点（能保存数据，同时能成为 master），法定数就是 6 。
* 如果你有 3 个候选 master 节点，和 100 个 data 节点，法定数就是 2 ，你只要数数那些可以做 master 的节点数就可以了。
* 如果你有两个节点，你遇到难题了。法定数当然是 2 ，但是这意味着如果有一个节点挂掉，你整个集群就不可用了。 设置成 1 可以保证集群的功能，但是就无法保证集群脑裂了，像这样的情况，你最好至少保证有 3 个节点。

你可以在你的 elasticsearch.yml 文件中这样配置：

discovery.zen.minimum\_master\_nodes: 2

但是由于 ELasticsearch 是动态的，你可以很容易的添加和删除节点， 但是这会改变这个法定个数。 你不得不修改每一个索引节点的配置并且重启你的整个集群只是为了让配置生效，这将是非常痛苦的一件事情。

基于这个原因， minimum\_master\_nodes （还有一些其它配置）允许通过 API 调用的方式动态进行配置。 当你的集群在线运行的时候，你可以这样修改配置：

PUT /\_cluster/settings

{

"persistent" : {

"discovery.zen.minimum\_master\_nodes" : 2

}

}

这将成为一个永久的配置，并且无论你配置项里配置的如何，这个将优先生效。当你添加和删除 master 节点的时候，你需要更改这个配置。

**集群恢复方面的配置**

当你集群重启时，几个配置项影响你的分片恢复的表现。 首先，我们需要明白如果什么也没配置将会发生什么。

想象一下假设你有 10 个节点，每个节点只保存一个分片，这个分片是一个主分片或者是一个副本分片，或者说有一个有 5 个主分片／1 个副本分片的索引。有时你需要为整个集群做离线维护（比如，为了安装一个新的驱动程序）， 当你重启你的集群，恰巧出现了 5 个节点已经启动，还有 5 个还没启动的场景。

假设其它 5 个节点出问题，或者他们根本没有收到立即重启的命令。不管什么原因，你有 5 个节点在线上，这五个节点会相互通信，选出一个 master，从而形成一个集群。 他们注意到数据不再均匀分布，因为有 5 个节点在集群中丢失了，所以他们之间会立即启动分片复制。

最后，你的其它 5 个节点打开加入了集群。这些节点会发现 *它们* 的数据正在被复制到其他节点，所以他们删除本地数据（因为这份数据要么是多余的，要么是过时的）。 然后整个集群重新进行平衡，因为集群的大小已经从 5 变成了 10。

在整个过程中，你的节点会消耗磁盘和网络带宽，来回移动数据，因为没有更好的办法。对于有 TB 数据的大集群, 这种无用的数据传输需要 *很长时间* 。如果等待所有的节点重启好了，整个集群再上线，所有的本地的数据都不需要移动。

现在我们知道问题的所在了，我们可以修改一些设置来缓解它。 首先我们要给 ELasticsearch 一个严格的限制：

gateway.recover\_after\_nodes: 8

这将阻止 Elasticsearch 在存在至少 8 个节点（数据节点或者 master 节点）之前进行数据恢复。 这个值的设定取决于个人喜好：整个集群提供服务之前你希望有多少个节点在线？这种情况下，我们设置为 8，这意味着至少要有 8 个节点，该集群才可用。

现在我们要告诉 Elasticsearch 集群中 *应该* 有多少个节点，以及我们愿意为这些节点等待多长时间：

gateway.expected\_nodes: 10

gateway.recover\_after\_time: 5m

这意味着 Elasticsearch 会采取如下操作：

* 等待集群至少存在 8 个节点
* 等待 5 分钟，或者10 个节点上线后，才进行数据恢复，这取决于哪个条件先达到。

这三个设置可以在集群重启的时候避免过多的分片交换。这可能会让数据恢复从数个小时缩短为几秒钟。

注意：这些配置只能设置在 config/elasticsearch.yml 文件中或者是在命令行里（它们不能动态更新）它们只在整个集群重启的时候有实质性作用。

**最好使用单播代替组播**

Elasticsearch 默认被配置为使用单播发现，以防止节点无意中加入集群。只有在同一台机器上运行的节点才会自动组成集群。

虽然组播仍然 [作为插件提供](https://www.elastic.co/guide/en/elasticsearch/plugins/current/discovery-multicast.html)， 但它应该永远不被使用在生产环境了，否在你得到的结果就是一个节点意外的加入到了你的生产环境，仅仅是因为他们收到了一个错误的组播信号。 对于组播 *本身* 并没有错，组播会导致一些愚蠢的问题，并且导致集群变的脆弱（比如，一个网络工程师正在捣鼓网络，而没有告诉你，你会发现所有的节点突然发现不了对方了）。

使用单播，你可以为 Elasticsearch 提供一些它应该去尝试连接的节点列表。 当一个节点联系到单播列表中的成员时，它就会得到整个集群所有节点的状态，然后它会联系 master 节点，并加入集群。

这意味着你的单播列表不需要包含你的集群中的所有节点， 它只是需要足够的节点，当一个新节点联系上其中一个并且说上话就可以了。如果你使用 master 候选节点作为单播列表，你只要列出三个就可以了。 这个配置在 elasticsearch.yml 文件中：

discovery.zen.ping.unicast.hosts: ["host1", "host2:port"]

关于 Elasticsearch 节点发现的详细信息，请参阅 [Zen Discovery](https://www.elastic.co/guide/en/elasticsearch/reference/current/modules-discovery-zen.html) Elasticsearch 文献。

### 不要触碰这些配置！

在 Elasticsearch 中有一些热点，人们可能不可避免的会碰到。 我们理解的，所有的调整就是为了优化，但是这些调整，你真的不需要理会它。因为它们经常会被乱用，从而造成系统的不稳定或者糟糕的性能，甚至两者都有可能。

**垃圾回收器**

这里已经简要介绍了 [垃圾回收入门](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_monitoring_individual_nodes.html#garbage_collector_primer)，JVM 使用一个垃圾回收器来释放不再使用的内存。 这篇内容的确是上一篇的一个延续， 但是因为重要，所以值得单独拿出来作为一节。

不要更改默认的垃圾回收器！

Elasticsearch 默认的垃圾回收器（ GC ）是 CMS。 这个垃圾回收器可以和应用并行处理，以便它可以最小化停顿。 然而，它有两个 stop-the-world 阶段，处理大内存也有点吃力。

尽管有这些缺点，它还是目前对于像 Elasticsearch 这样低延迟需求软件的最佳垃圾回收器。官方建议使用 CMS。

现在有一款新的垃圾回收器，叫 G1 垃圾回收器（ G1GC ）。 这款新的 GC 被设计，旨在比 CMS 更小的暂停时间，以及对大内存的处理能力。 它的原理是把内存分成许多区域，并且预测哪些区域最有可能需要回收内存。通过优先收集这些区域（ *garbage first* ），产生更小的暂停时间，从而能应对更大的内存。

听起来很棒！遗憾的是，G1GC 还是太新了，经常发现新的 bugs。这些错误通常是段（ segfault ）类型，便造成硬盘的崩溃。 Lucene 的测试套件对垃圾回收算法要求严格，看起来这些缺陷 G1GC 并没有很好地解决。

我们很希望在将来某一天推荐使用 G1GC，但是对于现在，它还不能足够稳定的满足 Elasticsearch 和 Lucene 的要求。

**线程池**

许多人 *喜欢* 调整线程池。 无论什么原因，人们都对增加线程数无法抵抗。索引太多了？增加线程！搜索太多了？增加线程！节点空闲率低于 95％？增加线程！

Elasticsearch 默认的线程设置已经是很合理的了。对于所有的线程池（除了 搜索 ），线程个数是根据 CPU 核心数设置的。 如果你有 8 个核，你可以同时运行的只有 8 个线程，只分配 8 个线程给任何特定的线程池是有道理的。

搜索线程池设置的大一点，配置为 int（（ 核心数 ＊ 3 ）／ 2 ）＋ 1 。

你可能会认为某些线程可能会阻塞（如磁盘上的 I／O 操作），所以你才想加大线程的。对于 Elasticsearch 来说这并不是一个问题：因为大多数 I／O 的操作是由 Lucene 线程管理的，而不是 Elasticsearch。

此外，线程池通过传递彼此之间的工作配合。你不必再因为它正在等待磁盘写操作而担心网络线程阻塞， 因为网络线程早已把这个工作交给另外的线程池，并且网络进行了响应。

最后，你的处理器的计算能力是有限的，拥有更多的线程会导致你的处理器频繁切换线程上下文。 一个处理器同时只能运行一个线程。所以当它需要切换到其它不同的线程的时候，它会存储当前的状态（寄存器等等），然后加载另外一个线程。 如果幸运的话，这个切换发生在同一个核心，如果不幸的话，这个切换可能发生在不同的核心，这就需要在内核间总线上进行传输。

这个上下文的切换，会给 CPU 时钟周期带来管理调度的开销；在现代的 CPUs 上，开销估计高达 30 μs。也就是说线程会被堵塞超过 30 μs，如果这个时间用于线程的运行，极有可能早就结束了。

人们经常稀里糊涂的设置线程池的值。8 个核的 CPU，我们遇到过有人配了 60、100 甚至 1000 个线程。 这些设置只会让 CPU 实际工作效率更低。

所以，下次请不要调整线程池的线程数。如果你真 *想调整* ， 一定要关注你的 CPU 核心数，最多设置成核心数的两倍，再多了都是浪费。

### 堆内存:大小和交换

Elasticsearch 默认安装后设置的堆内存是 1 GB。 对于任何一个业务部署来说， 这个设置都太小了。如果你正在使用这些默认堆内存配置，您的集群可能会出现问题。

这里有两种方式修改 Elasticsearch 的堆内存。最简单的一个方法就是指定 ES\_HEAP\_SIZE 环境变量。服务进程在启动时候会读取这个变量，并相应的设置堆的大小。 比如，你可以用下面的命令设置它：

export ES\_HEAP\_SIZE=10g

此外，你也可以通过命令行参数的形式，在程序启动的时候把内存大小传递给它，如果你觉得这样更简单的话：

./bin/elasticsearch -Xmx10g -Xms10g https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/heap-sizing.html#CO301-1) | 确保堆内存最小值（ Xms ）与最大值（ Xmx ）的大小是相同的，防止程序在运行时改变堆内存大小， 这是一个很耗系统资源的过程。 |

通常来说，设置 ES\_HEAP\_SIZE 环境变量，比直接写 -Xmx -Xms 更好一点。

**把你的内存的（少于）一半给 Lucene**

一个常见的问题是给 Elasticsearch 分配的内存 *太* 大了。 假设你有一个 64 GB 内存的机器， 天啊，我要把 64 GB 内存全都给 Elasticsearch。因为越多越好啊！

当然，内存对于 Elasticsearch 来说绝对是重要的，它可以被许多内存数据结构使用来提供更快的操作。但是说到这里， 还有另外一个内存消耗大户 *非堆内存* （off-heap）：Lucene。

Lucene 被设计为可以利用操作系统底层机制来缓存内存数据结构。 Lucene 的段是分别存储到单个文件中的。因为段是不可变的，这些文件也都不会变化，这是对缓存友好的，同时操作系统也会把这些段文件缓存起来，以便更快的访问。

Lucene 的性能取决于和操作系统的相互作用。如果你把所有的内存都分配给 Elasticsearch 的堆内存，那将不会有剩余的内存交给 Lucene。 这将严重地影响全文检索的性能。

标准的建议是把 50％ 的可用内存作为 Elasticsearch 的堆内存，保留剩下的 50％。当然它也不会被浪费，Lucene 会很乐意利用起余下的内存。

如果你不需要对分词字符串做聚合计算（例如，不需要 [fielddata](https://www.elastic.co/guide/cn/elasticsearch/guide/current/aggregations-and-analysis.html) ）可以考虑降低堆内存。堆内存越小，Elasticsearch（更快的 GC）和 Lucene（更多的内存用于缓存）的性能越好。

**不要超过 32 GB！**

这里有另外一个原因不分配大内存给 Elasticsearch。事实上 ， JVM 在内存小于 32 GB 的时候会采用一个内存对象指针压缩技术。

在 Java 中，所有的对象都分配在堆上，并通过一个指针进行引用。 普通对象指针（OOP）指向这些对象，通常为 CPU *字长* 的大小：32 位或 64 位，取决于你的处理器。指针引用的就是这个 OOP 值的字节位置。

对于 32 位的系统，意味着堆内存大小最大为 4 GB。对于 64 位的系统， 可以使用更大的内存，但是 64 位的指针意味着更大的浪费，因为你的指针本身大了。更糟糕的是， 更大的指针在主内存和各级缓存（例如 LLC，L1 等）之间移动数据的时候，会占用更多的带宽。

Java 使用一个叫作 [内存指针压缩（compressed oops）](https://wikis.oracle.com/display/HotSpotInternals/CompressedOops)的技术来解决这个问题。 它的指针不再表示对象在内存中的精确位置，而是表示 *偏移量* 。这意味着 32 位的指针可以引用 40 亿个 *对象* ， 而不是 40 亿个字节。最终， 也就是说堆内存增长到 32 GB 的物理内存，也可以用 32 位的指针表示。

一旦你越过那个神奇的 ~32 GB 的边界，指针就会切回普通对象的指针。 每个对象的指针都变长了，就会使用更多的 CPU 内存带宽，也就是说你实际上失去了更多的内存。事实上，当内存到达 40–50 GB 的时候，有效内存才相当于使用内存对象指针压缩技术时候的 32 GB 内存。

这段描述的意思就是说：即便你有足够的内存，也尽量不要 超过 32 GB。因为它浪费了内存，降低了 CPU 的性能，还要让 GC 应对大内存。

**到底需要低于 32 GB多少，来设置我的 JVM？**

遗憾的是，这需要看情况。确切的划分要根据 JVMs 和操作系统而定。 如果你想保证其安全可靠，设置堆内存为 31 GB 是一个安全的选择。 另外，你可以在你的 JVM 设置里添加 -XX:+PrintFlagsFinal 用来验证 JVM 的临界值， 并且检查 UseCompressedOops 的值是否为 true。对于你自己使用的 JVM 和操作系统，这将找到最合适的堆内存临界值。

例如，我们在一台安装 Java 1.7 的 MacOSX 上测试，可以看到指针压缩在被禁用之前，最大堆内存大约是在 32600 mb（~31.83 gb）：

$ JAVA\_HOME=`/usr/libexec/java\_home -v 1.7` java -Xmx32600m -XX:+PrintFlagsFinal 2> /dev/null | grep UseCompressedOops

bool UseCompressedOops := true

$ JAVA\_HOME=`/usr/libexec/java\_home -v 1.7` java -Xmx32766m -XX:+PrintFlagsFinal 2> /dev/null | grep UseCompressedOops

bool UseCompressedOops = false

相比之下，同一台机器安装 Java 1.8，可以看到指针压缩在被禁用之前，最大堆内存大约是在 32766 mb（~31.99 gb）：

$ JAVA\_HOME=`/usr/libexec/java\_home -v 1.8` java -Xmx32766m -XX:+PrintFlagsFinal 2> /dev/null | grep UseCompressedOops

bool UseCompressedOops := true

$ JAVA\_HOME=`/usr/libexec/java\_home -v 1.8` java -Xmx32767m -XX:+PrintFlagsFinal 2> /dev/null | grep UseCompressedOops

bool UseCompressedOops = false

这个例子告诉我们，影响内存指针压缩使用的临界值， 是会根据 JVM 的不同而变化的。 所以从其他地方获取的例子，需要谨慎使用，要确认检查操作系统配置和 JVM。

如果使用的是 Elasticsearch v2.2.0，启动日志其实会告诉你 JVM 是否正在使用内存指针压缩。 你会看到像这样的日志消息：

[2015-12-16 13:53:33,417][INFO ][env] [Illyana Rasputin] heap size [989.8mb], compressed ordinary object pointers [true]

这表明内存指针压缩正在被使用。如果没有，日志消息会显示 [false] 。

**我有一个 1 TB 内存的机器！**

这个 32 GB 的分割线是很重要的。那如果你的机器有很大的内存怎么办呢？ 一台有着 512–768 GB内存的服务器愈发常见。

首先，我们建议避免使用这样的高配机器（参考 [硬件](https://www.elastic.co/guide/cn/elasticsearch/guide/current/hardware.html)）。

但是如果你已经有了这样的机器，你有三个可选项：

* 你主要做全文检索吗？考虑给 Elasticsearch 4 - 32 GB 的内存， 让 Lucene 通过操作系统文件缓存来利用余下的内存。那些内存都会用来缓存 segments，带来极速的全文检索。
* 你需要更多的排序和聚合？而且大部分的聚合计算是在数字、日期、地理点和 非分词 字符串上？你很幸运，你的聚合计算将在内存友好的 doc values 上完成！ 给 Elasticsearch 4 到 32 GB 的内存，其余部分为操作系统缓存内存中的 doc values。
* 你在对分词字符串做大量的排序和聚合（例如，标签或者 SigTerms，等等）不幸的是，这意味着你需要 fielddata，意味着你需要堆空间。考虑在单个机器上运行两个或多个节点，而不是拥有大量 RAM 的一个节点。仍然要坚持 50％ 原则。

假设你有个机器有 128 GB 的内存，你可以创建两个节点，每个节点内存分配不超过 32 GB。 也就是说不超过 64 GB 内存给 ES 的堆内存，剩下的超过 64 GB 的内存给 Lucene。

如果你选择这一种，你需要配置 cluster.routing.allocation.same\_shard.host: true 。 这会防止同一个分片（shard）的主副本存在同一个物理机上（因为如果存在一个机器上，副本的高可用性就没有了）。

**Swapping 是性能的坟墓**

这是显而易见的， 但是还是有必要说的更清楚一点：内存交换 到磁盘对服务器性能来说是 *致命* 的。想想看：一个内存操作必须能够被快速执行。

如果内存交换到磁盘上，一个 100 微秒的操作可能变成 10 毫秒。 再想想那么多 10 微秒的操作时延累加起来。 不难看出 swapping 对于性能是多么可怕。

最好的办法就是在你的操作系统中完全禁用 swap。这样可以暂时禁用：

sudo swapoff -a

如果需要永久禁用，你可能需要修改 /etc/fstab 文件，这要参考你的操作系统相关文档。

如果你并不打算完全禁用 swap，也可以选择降低 swappiness 的值。 这个值决定操作系统交换内存的频率。 这可以预防正常情况下发生交换，但仍允许操作系统在紧急情况下发生交换。

对于大部分Linux操作系统，可以在 sysctl 中这样配置：

vm.swappiness = 1 https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/heap-sizing.html#CO302-1) | swappiness 设置为 1 比设置为 0 要好，因为在一些内核版本 swappiness 设置为 0 会触发系统 OOM-killer（注：Linux 内核的 Out of Memory（OOM）killer 机制）。 |

最后，如果上面的方法都不合适，你需要打开配置文件中的 mlockall 开关。 它的作用就是允许 JVM 锁住内存，禁止操作系统交换出去。在你的 elasticsearch.yml 文件中，设置如下：

bootstrap.mlockall: true

### 文件描述符和 MMap

Lucene 使用了 *大量的* 文件。 同时，Elasticsearch 在节点和 HTTP 客户端之间进行通信也使用了大量的套接字（注：sockets）。 所有这一切都需要足够的文件描述符。

可悲的是，许多现代的 Linux 发行版本，每个进程默认允许一个微不足道的 1024 文件描述符。这对一个小的 Elasticsearch 节点来说实在是太 *低* 了，更不用说一个处理数以百计索引的节点。

你应该增加你的文件描述符，设置一个很大的值，如 64,000。这个过程困难得让人恼火，它高度依赖于你的特定操作系统和分布。请参考你操作系统文档来确定如何最好地修改允许的文件描述符数量。

一旦你认为已经改变了它，检查 Elasticsearch，以确保它的真的起作用并且有足够的文件描述符：

GET /\_nodes/process

{

"cluster\_name": "elasticsearch\_\_zach",

"nodes": {

"TGn9iO2\_QQKb0kavcLbnDw": {

"name": "Zach",

"transport\_address": "inet[/192.168.1.131:9300]",

"host": "zacharys-air",

"ip": "192.168.1.131",

"version": "2.0.0-SNAPSHOT",

"build": "612f461",

"http\_address": "inet[/192.168.1.131:9200]",

"process": {

"refresh\_interval\_in\_millis": 1000,

"id": 19808,

"max\_file\_descriptors": 64000, https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

"mlockall": true

}

}

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_file_descriptors_and_mmap.html#CO303-1) | max\_file\_descriptors 字段显示 Elasticsearch 进程可以访问的可用文件描述符数量。 |

Elasticsearch 对各种文件混合使用了 NioFs（ 注：非阻塞文件系统）和 MMapFs （ 注：内存映射文件系统）。请确保你配置的最大映射数量，以便有足够的虚拟内存可用于 mmapped 文件。这可以暂时设置：

sysctl -w vm.max\_map\_count=262144

或者你可以在 /etc/sysctl.conf 通过修改 vm.max\_map\_count 永久设置它。

## 部署后

一旦将集群部署到生产环境后，就需要有一些工具及最佳实践经验来保证集群运行在最佳状态。本章将探讨动态配置、日志调优、索引性能优化以及集群备份。

### 动态变更设置

Elasticsearch 里很多设置都是动态的，可以通过 API 修改。需要强制重启节点（或者集群）的配置修改都要极力避免。 而且虽然通过静态配置项也可以完成这些变更，我们建议你还是用 API 来实现。

集群更新 API 有两种工作模式：

临时（Transient）

这些变更在集群重启之前一直会生效。一旦整个集群重启，这些配置就被清除。

永久（Persistent）

这些变更会永久存在直到被显式修改。即使全集群重启它们也会存活下来并覆盖掉静态配置文件里的选项。

临时或永久配置需要在 JSON 体里分别指定：

PUT /\_cluster/settings

{

"persistent" : {

"discovery.zen.minimum\_master\_nodes" : 2 https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

},

"transient" : {

"indices.store.throttle.max\_bytes\_per\_sec" : "50mb" https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_changing_settings_dynamically.html#CO304-1) | 这个永久设置会在全集群重启时存活下来。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_changing_settings_dynamically.html#CO304-2) | 这个临时设置会在第一次全集群重启后被移除。 |

### 日志记录

Elasticsearch 会输出很多日志，都放在 ES\_HOME/logs 目录下。默认的日志记录等级是 INFO 。 它提供了适度的信息，但是又设计好了不至于让你的日志太过庞大。

当调试问题的时候，特别是节点发现相关的问题（因为这个经常依赖于各式过于繁琐的网络配置），提高日志记录等级到 DEBUG 是很有帮助的。

你 *可以* 修改 logging.yml 文件然后重启你的节点——但是这样做即繁琐还会导致不必要的宕机时间。作为替代，你可以通过 cluster-settings API 更新日志记录级别，就像我们前面刚学过的那样。

要实现这个更新，选择你感兴趣的日志器，然后在前面补上 logger. 。对根日志器你可以用 logger.\_root 来表示。

让我们调高节点发现的日志记录级别：

PUT /\_cluster/settings

{

"transient" : {

"logger.discovery" : "DEBUG"

}

}

设置失效，Elasticsearch 将开始输出 discovery 模块的 DEBUG 级别的日志。

提示

避免使用 TRACE 。这个级别非常的详细，详细到日志反而不再有用了。

**慢日志**

还有另一个日志叫 *慢日志* 。这个日志的目的是捕获那些超过指定时间阈值的查询和索引请求。这个日志用来追踪由用户产生的很慢的请求很有用。

默认情况，慢日志是不开启的。要开启它，需要定义具体动作（query，fetch 还是 index），你期望的事件记录等级（ WARN 、 DEBUG 等），以及时间阈值。

这是一个索引级别的设置，也就是说可以独立应用给单个索引：

PUT /my\_index/\_settings

{

"index.search.slowlog.threshold.query.warn" : "10s", https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

"index.search.slowlog.threshold.fetch.debug": "500ms", https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

"index.indexing.slowlog.threshold.index.info": "5s" https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/logging.html#CO305-1) | 查询慢于 10 秒输出一个 WARN 日志。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/logging.html#CO305-2) | 获取慢于 500 毫秒输出一个 DEBUG 日志。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/logging.html#CO305-3) | 索引慢于 5 秒输出一个 INFO 日志。 |

你也可以在 elasticsearch.yml 文件里定义这些阈值。没有阈值设置的索引会自动继承在静态配置文件里配置的参数。

一旦阈值设置过了，你可以和其他日志器一样切换日志记录等级：

PUT /\_cluster/settings

{

"transient" : {

"logger.index.search.slowlog" : "DEBUG", https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

"logger.index.indexing.slowlog" : "WARN" https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/logging.html#CO306-1) | 设置搜索慢日志为 DEBUG 级别。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/logging.html#CO306-2) | 设置索引慢日志为 WARN 级别。 |

### 索引性能技巧

如果你是在一个索引负载很重的环境， 比如索引的是基础设施日志，你可能愿意牺牲一些搜索性能换取更快的索引速率。在这些场景里，搜索常常是很少见的操作，而且一般是由你公司内部的人发起的。他们也愿意为一个搜索等上几秒钟，而不像普通消费者，要求一个搜索必须毫秒级返回。

基于这种特殊的场景，我们可以有几种权衡办法来提高你的索引性能。

**这些技巧仅适用于 Elasticsearch 1.3 及以后的版本**

本书是为最新几个版本的 Elasticsearch 写的，虽然大多数内容在更老的版本也也有效。

不过，本节提及的技巧， *只* 针对 1.3 及以后版本。该版本后有不少性能提升和故障修复是直接影响到索引的。事实上，有些建议在老版本上反而会因为故障或性能缺陷而 *降低* 性能。

**科学的测试性能**

性能测试永远是复杂的，所以在你的方法里已经要尽可能的科学。 随机摆弄旋钮以及写入开关可不是做性能调优的好办法。如果有太多种 *可能* ，我们就无法判断到底哪一种有最好的 *效果* 。合理的测试方法如下：

1. 在单个节点上，对单个分片，无副本的场景测试性能。
2. 在 100% 默认配置的情况下记录性能结果，这样你就有了一个对比基线。
3. 确保性能测试运行足够长的时间（30 分钟以上）这样你可以评估长期性能，而不是短期的峰值或延迟。一些事件（比如段合并，GC）不会立刻发生，所以性能概况会随着时间继续而改变的。
4. 开始在基线上逐一修改默认值。严格测试它们，如果性能提升可以接受，保留这个配置项，开始下一项。

**使用批量请求并调整其大小**

显而易见的，优化性能应该使用批量请求。 批量的大小则取决于你的数据、分析和集群配置，不过每次批量数据 5–15 MB 大是个不错的起始点。注意这里说的是物理字节数大小。文档计数对批量大小来说不是一个好指标。比如说，如果你每次批量索引 1000 个文档，记住下面的事实：

* 1000 个 1 KB 大小的文档加起来是 1 MB 大。
* 1000 个 100 KB 大小的文档加起来是 100 MB 大。

这可是完完全全不一样的批量大小了。批量请求需要在协调节点上加载进内存，所以批量请求的物理大小比文档计数重要得多。

从 5–15 MB 开始测试批量请求大小，缓慢增加这个数字，直到你看不到性能提升为止。然后开始增加你的批量写入的并发度（多线程等等办法）。

用 Marvel 以及诸如 iostat 、 top 和 ps 等工具监控你的节点，观察资源什么时候达到瓶颈。如果你开始收到 EsRejectedExecutionException ，你的集群没办法再继续了：至少有一种资源到瓶颈了。或者减少并发数，或者提供更多的受限资源（比如从机械磁盘换成 SSD），或者添加更多节点。

注意

写数据的时候，要确保批量请求是轮询发往你的全部数据节点的。不要把所有请求都发给单个节点，因为这个节点会需要在处理的时候把所有批量请求都存在内存里。

**存储**

磁盘在现代服务器上通常都是瓶颈。Elasticsearch 重度使用磁盘，你的磁盘能处理的吞吐量越大，你的节点就越稳定。这里有一些优化磁盘 I/O 的技巧：

* 使用 SSD。就像其他地方提过的， 他们比机械磁盘优秀多了。
* 使用 RAID 0。条带化 RAID 会提高磁盘 I/O，代价显然就是当一块硬盘故障时整个就故障了。不要使用镜像或者奇偶校验 RAID 因为副本已经提供了这个功能。
* 另外，使用多块硬盘，并允许 Elasticsearch 通过多个 path.data 目录配置把数据条带化分配到它们上面。
* 不要使用远程挂载的存储，比如 NFS 或者 SMB/CIFS。这个引入的延迟对性能来说完全是背道而驰的。
* 如果你用的是 EC2，当心 EBS。即便是基于 SSD 的 EBS，通常也比本地实例的存储要慢。

**段和合并**

段合并的计算量庞大， 而且还要吃掉大量磁盘 I/O。合并在后台定期操作，因为他们可能要很长时间才能完成，尤其是比较大的段。这个通常来说都没问题，因为大规模段合并的概率是很小的。

不过有时候合并会拖累写入速率。如果这个真的发生了，Elasticsearch 会自动限制索引请求到单个线程里。这个可以防止出现 *段爆炸* 问题，即数以百计的段在被合并之前就生成出来。如果 Elasticsearch 发现合并拖累索引了，它会会记录一个声明有 now throttling indexing 的 INFO 级别信息。

Elasticsearch 默认设置在这块比较保守：不希望搜索性能被后台合并影响。不过有时候（尤其是 SSD，或者日志场景）限流阈值太低了。

默认值是 20 MB/s，对机械磁盘应该是个不错的设置。如果你用的是 SSD，可以考虑提高到 100–200 MB/s。测试验证对你的系统哪个值合适：

PUT /\_cluster/settings

{

"persistent" : {

"indices.store.throttle.max\_bytes\_per\_sec" : "100mb"

}

}

如果你在做批量导入，完全不在意搜索，你可以彻底关掉合并限流。这样让你的索引速度跑到你磁盘允许的极限：

PUT /\_cluster/settings

{

"transient" : {

"indices.store.throttle.type" : "none" https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/indexing-performance.html#CO307-1) | 设置限流类型为 none 彻底关闭合并限流。等你完成了导入，记得改回 merge 重新打开限流。 |

如果你使用的是机械磁盘而非 SSD，你需要添加下面这个配置到你的 elasticsearch.yml 里：

index.merge.scheduler.max\_thread\_count: 1

机械磁盘在并发 I/O 支持方面比较差，所以我们需要降低每个索引并发访问磁盘的线程数。这个设置允许 max\_thread\_count + 2 个线程同时进行磁盘操作，也就是设置为 1 允许三个线程。

对于 SSD，你可以忽略这个设置，默认是 Math.min(3, Runtime.getRuntime().availableProcessors() / 2) ，对 SSD 来说运行的很好。

最后，你可以增加 index.translog.flush\_threshold\_size 设置，从默认的 512 MB 到更大一些的值，比如 1 GB。这可以在一次清空触发的时候在事务日志里积累出更大的段。而通过构建更大的段，清空的频率变低，大段合并的频率也变低。这一切合起来导致更少的磁盘 I/O 开销和更好的索引速率。当然，你会需要对应量级的 heap 内存用以积累更大的缓冲空间，调整这个设置的时候请记住这点。

**其他**

最后，还有一些其他值得考虑的东西需要记住：

* 如果你的搜索结果不需要近实时的准确度，考虑把每个索引的 index.refresh\_interval 改到 30s。如果你是在做大批量导入，导入期间你可以通过设置这个值为 -1 关掉刷新。别忘记在完工的时候重新开启它。
* 如果你在做大批量导入，考虑通过设置 index.number\_of\_replicas: 0关闭副本。文档在复制的时候，整个文档内容都被发往副本节点，然后逐字的把索引过程重复一遍。这意味着每个副本也会执行分析、索引以及可能的合并过程。

相反，如果你的索引是零副本，然后在写入完成后再开启副本，恢复过程本质上只是一个字节到字节的网络传输。相比重复索引过程，这个算是相当高效的了。

* 如果你没有给每个文档自带 ID，使用 Elasticsearch 的自动 ID 功能。 这个为避免版本查找做了优化，因为自动生成的 ID 是唯一的。
* 如果你在使用自己的 ID，尝试使用一种 [Lucene 友好的](http://blog.mikemccandless.com/2014/05/choosing-fast-unique-identifier-uuid.html) ID。包括零填充序列 ID、UUID-1 和纳秒；这些 ID 都是有一致的，压缩良好的序列模式。相反的，像 UUID-4 这样的 ID，本质上是随机的，压缩比很低，会明显拖慢 Lucene。

### 推迟分片分配

正如我们在 [水平扩容](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_scale_horizontally.html) 讨论过， Elasticsearch 将自动在可用节点间进行分片均衡，包括新节点的加入和现有节点的离线。

理论上来说，这个是理想的行为，我们想要提拔副本分片来尽快恢复丢失的主分片。 我们同时也希望保证资源在整个集群的均衡，用以避免热点。

然而，在实践中，立即的再均衡所造成的问题会比其解决的更多。举例来说，考虑到以下情形：

1. Node（节点） 19 在网络中失联了（某个家伙踢到了电源线)
2. Master 立即注意到了这个节点的离线，它决定在集群内提拔其他拥有 Node 19 上面的主分片对应的副本分片为主分片
3. 在副本被提拔为主分片以后，master 节点开始执行恢复操作来重建缺失的副本。集群中的节点之间互相拷贝分片数据，网卡压力剧增，集群状态尝试变绿。
4. 由于目前集群处于非平衡状态，这个过程还有可能会触发小规模的分片移动。其他不相关的分片将在节点间迁移来达到一个最佳的平衡状态

与此同时，那个踢到电源线的倒霉管理员，把服务器插好电源线进行了重启，现在节点 Node 19 又重新加入到了集群。不幸的是，这个节点被告知当前的数据已经没有用了， 数据已经在其他节点上重新分配了。所以 Node 19 把本地的数据进行删除，然后重新开始恢复集群的其他分片（然后这又导致了一个新的再平衡）

如果这一切听起来是不必要的且开销极大，那就对了。是的，不过前提是你知道这个节点会很快回来。如果节点 Node 19 真的丢了，上面的流程确实正是我们想要发生的。

为了解决这种瞬时中断的问题，Elasticsearch 可以推迟分片的分配。这可以让你的集群在重新分配之前有时间去检测这个节点是否会再次重新加入。

**修改默认延时**

默认情况，集群会等待一分钟来查看节点是否会重新加入，如果这个节点在此期间重新加入，重新加入的节点会保持其现有的分片数据，不会触发新的分片分配。

通过修改参数 delayed\_timeout ，默认等待时间可以全局设置也可以在索引级别进行修改:

PUT /\_all/\_settings https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

{

"settings": {

"index.unassigned.node\_left.delayed\_timeout": "5m" https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_delaying_shard_allocation.html#CO308-1) | 通过使用 \_all 索引名，我们可以为集群里面的所有的索引使用这个参数 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_delaying_shard_allocation.html#CO308-2) | 默认时间被修改成了 5 分钟 |

这个配置是动态的，可以在运行时进行修改。如果你希望分片立即分配而不想等待，你可以设置参数： delayed\_timeout: 0.

注意

延迟分配不会阻止副本被提拔为主分片。集群还是会进行必要的提拔来让集群回到 yellow状态。缺失副本的重建是唯一被延迟的过程。

**自动取消分片迁移**

如果节点在超时之后再回来，且集群还没有完成分片的移动，会发生什么事情呢？在这种情形下， Elasticsearch 会检查该机器磁盘上的分片数据和当前集群中的活跃主分片的数据是不是一样 — 如果两者匹配， 说明没有进来新的文档，包括删除和修改 — 那么 master 将会取消正在进行的再平衡并恢复该机器磁盘上的数据。

之所以这样做是因为本地磁盘的恢复永远要比网络间传输要快，并且我们保证了他们的分片数据是一样的，这个过程可以说是双赢。

如果分片已经产生了分歧（比如：节点离线之后又索引了新的文档），那么恢复进程会继续按照正常流程进行。重新加入的节点会删除本地的、过时的数据，然后重新获取一份新的。

### 滚动重启

总有一天你会需要做一次集群的滚动重启——保持集群在线和可操作，但是逐一把节点下线。

常见的原因：Elasticsearch 版本升级，或者服务器自身的一些维护操作（比如操作系统升级或者硬件相关）。不管哪种情况，都要有一种特别的方法来完成一次滚动重启。

正常情况下，Elasticsearch 希望你的数据被完全的复制和均衡的分布。如果你手动关闭了一个节点，集群会立刻发现节点的丢失并开始再平衡。如果节点的维护是短期工作的话，这一点就很烦人了，因为大型分片的再平衡需要花费相当的时间（想想尝试复制 1TB 的数据——即便在高速网络上也是不一般的事情了）。

我们需要的是，告诉 Elasticsearch 推迟再平衡，因为对外部因子影响下的集群状态，我们自己更了解。操作流程如下：

1. 可能的话，停止索引新的数据。虽然不是每次都能真的做到，但是这一步可以帮助提高恢复速度。
2. 禁止分片分配。这一步阻止 Elasticsearch 再平衡缺失的分片，直到你告诉它可以进行了。如果你知道维护窗口会很短，这个主意棒极了。你可以像下面这样禁止分配：
3. PUT /\_cluster/settings
4. {
5. "transient" : {
6. "cluster.routing.allocation.enable" : "none"
7. }

}

1. 关闭单个节点。
2. 执行维护/升级。
3. 重启节点，然后确认它加入到集群了。
4. 用如下命令重启分片分配：
5. PUT /\_cluster/settings
6. {
7. "transient" : {
8. "cluster.routing.allocation.enable" : "all"
9. }

}

分片再平衡会花一些时间。一直等到集群变成 绿色 状态后再继续。

1. 重复第 2 到 6 步操作剩余节点。
2. 到这步你可以安全的恢复索引了（如果你之前停止了的话），不过等待集群完全均衡后再恢复索引，也会有助于提高处理速度。

### 备份你的集群

使用无论哪个存储数据的软件，定期备份你的数据都是很重要的。 Elasticsearch 副本提供了高可靠性；它们让你可以容忍零星的节点丢失而不会中断服务。

但是，副本并不提供对灾难性故障的保护。对这种情况，你需要的是对集群真正的备份——在某些东西确实出问题的时候有一个完整的拷贝。

要备份你的集群，你可以使用 snapshot API。这个会拿到你集群里当前的状态和数据然后保存到一个共享仓库里。这个备份过程是"智能"的。你的第一个快照会是一个数据的完整拷贝，但是所有后续的快照会保留的是已存快照和新数据之间的差异。随着你不时的对数据进行快照，备份也在增量的添加和删除。这意味着后续备份会相当快速，因为它们只传输很小的数据量。

要使用这个功能，你必须首先创建一个保存数据的仓库。有多个仓库类型可以供你选择：

* 共享文件系统，比如 NAS
* Amazon S3
* HDFS (Hadoop 分布式文件系统)
* Azure Cloud

**创建仓库**

让我部署一个共享 文件系统仓库：

PUT \_snapshot/my\_backup https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

{

"type": "fs", https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

"settings": {

"location": "/mount/backups/my\_backup" https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/backing-up-your-cluster.html#CO309-1) | 给我们的仓库取一个名字，在本例它叫 my\_backup 。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/backing-up-your-cluster.html#CO309-2) | 我们指定仓库的类型应该是一个共享文件系统。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/backing-up-your-cluster.html#CO309-3) | 最后，我们提供一个已挂载的设备作为目的地址。 |

注意：共享文件系统路径必须确保集群所有节点都可以访问到。

这步会在挂载点创建仓库和所需的元数据。还有一些其他的配置你可能想要配置的，这些取决于你的节点、网络的性能状况和仓库位置：

max\_snapshot\_bytes\_per\_sec

当快照数据进入仓库时，这个参数控制这个过程的限流情况。默认是每秒 20mb 。

max\_restore\_bytes\_per\_sec

当从仓库恢复数据时，这个参数控制什么时候恢复过程会被限流以保障你的网络不会被占满。默认是每秒 `20mb`。

假设我们有一个非常快的网络，而且对额外的流量也很 OK，那我们可以增加这些默认值：

POST \_snapshot/my\_backup/ https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

{

"type": "fs",

"settings": {

"location": "/mount/backups/my\_backup",

"max\_snapshot\_bytes\_per\_sec" : "50mb", https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

"max\_restore\_bytes\_per\_sec" : "50mb"

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/backing-up-your-cluster.html#CO310-1) | 注意我们用的是 POST 而不是 PUT 。这会更新已有仓库的设置。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/backing-up-your-cluster.html#CO310-2) | 然后添加我们的新设置。 |

**快照所有打开的索引**

一个仓库可以包含多个快照。 每个快照跟一系列索引相关（比如所有索引，一部分索引，或者单个索引）。当创建快照的时候，你指定你感兴趣的索引然后给快照取一个唯一的名字。

让我们从最基础的快照命令开始：

PUT \_snapshot/my\_backup/snapshot\_1

这个会备份所有打开的索引到 my\_backup 仓库下一个命名为 snapshot\_1 的快照里。这个调用会立刻返回，然后快照会在后台运行。

提示

通常你会希望你的快照作为后台进程运行，不过有时候你会希望在你的脚本中一直等待到完成。这可以通过添加一个 wait\_for\_completion 标记实现：

PUT \_snapshot/my\_backup/snapshot\_1?wait\_for\_completion=true

这个会阻塞调用直到快照完成。注意大型快照会花很长时间才返回。

**快照指定索引**

默认行为是备份所有打开的索引。 不过如果你在用 Marvel，你不是真的想要把所有诊断相关的 .marvel索引也备份起来。可能你就压根没那么大空间备份所有数据。

这种情况下，你可以在快照你的集群的时候指定备份哪些索引：

PUT \_snapshot/my\_backup/snapshot\_2

{

"indices": "index\_1,index\_2"

}

这个快照命令现在只会备份 index1 和 index2 了。

**列出快照相关的信息**

一旦你开始在你的仓库里积攒起快照了，你可能就慢慢忘记里面各自的细节了 ——特别是快照按照时间划分命名的时候（比如， backup\_2014\_10\_28 ）。

要获得单个快照的信息，直接对仓库和快照名发起一个 GET 请求：

GET \_snapshot/my\_backup/snapshot\_2

这个会返回一个小响应，包括快照相关的各种信息：

{

"snapshots": [

{

"snapshot": "snapshot\_1",

"indices": [

".marvel\_2014\_28\_10",

"index1",

"index2"

],

"state": "SUCCESS",

"start\_time": "2014-09-02T13:01:43.115Z",

"start\_time\_in\_millis": 1409662903115,

"end\_time": "2014-09-02T13:01:43.439Z",

"end\_time\_in\_millis": 1409662903439,

"duration\_in\_millis": 324,

"failures": [],

"shards": {

"total": 10,

"failed": 0,

"successful": 10

}

}

]

}

要获取一个仓库中所有快照的完整列表，使用 \_all 占位符替换掉具体的快照名称：

GET \_snapshot/my\_backup/\_all

**删除快照**

最后，我们需要一个命令来删除所有不再有用的旧快照 。这只要对仓库/快照名称发一个简单的 DELETEHTTP 调用：

DELETE \_snapshot/my\_backup/snapshot\_2

用 API 删除快照很重要，而不能用其他机制（比如手动删除，或者用 S3 上的自动清除工具）。因为快照是增量的，有可能很多快照依赖于过去的段。delete API 知道哪些数据还在被更多近期快照使用，然后会只删除不再被使用的段。

但是，如果你做了一次人工文件删除，你将会面临备份严重损坏的风险，因为你在删除的是可能还在使用中的数据。

**监控快照进度**

wait\_for\_completion 标记提供了一个监控的基础形式，但哪怕只是对一个中等规模的集群做快照恢复的时候，它都真的不够用。

另外两个 API 会给你有关快照状态更详细的信息。首先你可以给快照 ID 执行一个 `GET`，就像我们之前获取一个特定快照的信息时做的那样：

GET \_snapshot/my\_backup/snapshot\_3

如果你调用这个命令的时候快照还在进行中，你会看到它什么时候开始，运行了多久等等信息。不过要注意，这个 API 用的是快照机制相同的线程池。如果你在快照非常大的分片，状态更新的间隔会很大，因为 API 在竞争相同的线程池资源。

更好的方案是拽取 \_status API 数据：

GET \_snapshot/my\_backup/snapshot\_3/\_status

\_status API 立刻返回，然后给出详细的多的统计值输出：

{

"snapshots": [

{

"snapshot": "snapshot\_3",

"repository": "my\_backup",

"state": "IN\_PROGRESS", https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

"shards\_stats": {

"initializing": 0,

"started": 1, https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

"finalizing": 0,

"done": 4,

"failed": 0,

"total": 5

},

"stats": {

"number\_of\_files": 5,

"processed\_files": 5,

"total\_size\_in\_bytes": 1792,

"processed\_size\_in\_bytes": 1792,

"start\_time\_in\_millis": 1409663054859,

"time\_in\_millis": 64

},

"indices": {

"index\_3": {

"shards\_stats": {

"initializing": 0,

"started": 0,

"finalizing": 0,

"done": 5,

"failed": 0,

"total": 5

},

"stats": {

"number\_of\_files": 5,

"processed\_files": 5,

"total\_size\_in\_bytes": 1792,

"processed\_size\_in\_bytes": 1792,

"start\_time\_in\_millis": 1409663054859,

"time\_in\_millis": 64

},

"shards": {

"0": {

"stage": "DONE",

"stats": {

"number\_of\_files": 1,

"processed\_files": 1,

"total\_size\_in\_bytes": 514,

"processed\_size\_in\_bytes": 514,

"start\_time\_in\_millis": 1409663054862,

"time\_in\_millis": 22

}

},

...

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/backing-up-your-cluster.html#CO311-1) | 一个正在运行的快照会显示 IN\_PROGRESS 作为状态。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/backing-up-your-cluster.html#CO311-2) | 这个特定快照有一个分片还在传输（另外四个已经完成）。 |

响应包括快照的总体状况，但也包括下钻到每个索引和每个分片的统计值。这个给你展示了有关快照进展的非常详细的视图。分片可以在不同的完成状态：

INITIALIZING

分片在检查集群状态看看自己是否可以被快照。这个一般是非常快的。

STARTED

数据正在被传输到仓库。

FINALIZING

数据传输完成；分片现在在发送快照元数据。

DONE

快照完成！

FAILED

快照处理的时候碰到了错误，这个分片/索引/快照不可能完成了。检查你的日志获取更多信息。

**取消一个快照**

最后，你可能想取消一个快照或恢复。 因为它们是长期运行的进程，执行操作的时候一个笔误或者过错就会花很长时间来解决——而且同时还会耗尽有价值的资源。

要取消一个快照，在他进行中的时候简单的删除快照就可以：

DELETE \_snapshot/my\_backup/snapshot\_3

这个会中断快照进程。然后删除仓库里进行到一半的快照。

### 从快照恢复

一旦你备份过了数据，恢复它就简单了：只要在你希望恢复回集群的快照 ID 后面加上 \_restore 即可：

POST \_snapshot/my\_backup/snapshot\_1/\_restore

默认行为是把这个快照里存有的所有索引都恢复。如果 snapshot\_1 包括五个索引，这五个都会被恢复到我们集群里。 和 snapshot API 一样，我们也可以选择希望恢复具体哪个索引。

还有附加的选项用来重命名索引。这个选项允许你通过模式匹配索引名称，然后通过恢复进程提供一个新名称。如果你想在不替换现有数据的前提下，恢复老数据来验证内容，或者做其他处理，这个选项很有用。让我们从快照里恢复单个索引并提供一个替换的名称：

POST /\_snapshot/my\_backup/snapshot\_1/\_restore

{

"indices": "index\_1", https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

"rename\_pattern": "index\_(.+)", https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

"rename\_replacement": "restored\_index\_$1" https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_restoring_from_a_snapshot.html#CO312-1) | 只恢复 index\_1 索引，忽略快照中存在的其余索引。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_restoring_from_a_snapshot.html#CO312-2) | 查找所提供的模式能匹配上的正在恢复的索引。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_restoring_from_a_snapshot.html#CO312-3) | 然后把它们重命名成替代的模式。 |

这个会恢复 index\_1 到你及群里，但是重命名成了 restored\_index\_1 。

提示

和快照类似， restore 命令也会立刻返回，恢复进程会在后台进行。如果你更希望你的 HTTP 调用阻塞直到恢复完成，添加 wait\_for\_completion 标记：

POST \_snapshot/my\_backup/snapshot\_1/\_restore?wait\_for\_completion=true

**监控恢复操作**

从仓库恢复数据借鉴了 Elasticsearch 里已有的现行恢复机制。 在内部实现上，从仓库恢复分片和从另一个节点恢复是等价的。

如果你想监控恢复的进度，你可以使用 recovery API。这是一个通用目的的 API，用来展示你集群中移动着的分片状态。

这个 API 可以为你在恢复的指定索引单独调用：

GET restored\_index\_3/\_recovery

或者查看你集群里所有索引，可能包括跟你的恢复进程无关的其他分片移动：

GET /\_recovery/

输出会跟这个类似（注意，根据你集群的活跃度，输出可能会变得非常啰嗦！）：

{

"restored\_index\_3" : {

"shards" : [ {

"id" : 0,

"type" : "snapshot", https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

"stage" : "index",

"primary" : true,

"start\_time" : "2014-02-24T12:15:59.716",

"stop\_time" : 0,

"total\_time\_in\_millis" : 175576,

"source" : { https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

"repository" : "my\_backup",

"snapshot" : "snapshot\_3",

"index" : "restored\_index\_3"

},

"target" : {

"id" : "ryqJ5lO5S4-lSFbGntkEkg",

"hostname" : "my.fqdn",

"ip" : "10.0.1.7",

"name" : "my\_es\_node"

},

"index" : {

"files" : {

"total" : 73,

"reused" : 0,

"recovered" : 69,

"percent" : "94.5%" https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png

},

"bytes" : {

"total" : 79063092,

"reused" : 0,

"recovered" : 68891939,

"percent" : "87.1%"

},

"total\_time\_in\_millis" : 0

},

"translog" : {

"recovered" : 0,

"total\_time\_in\_millis" : 0

},

"start" : {

"check\_index\_time" : 0,

"total\_time\_in\_millis" : 0

}

} ]

}

}

|  |  |
| --- | --- |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_restoring_from_a_snapshot.html#CO313-1) | type 字段告诉你恢复的本质；这个分片是在从一个快照恢复。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_restoring_from_a_snapshot.html#CO313-2) | source 哈希描述了作为恢复来源的特定快照和仓库。 |
| [https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/3.png](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_restoring_from_a_snapshot.html#CO313-3) | percent 字段让你对恢复的状态有个概念。这个特定分片目前已经恢复了 94% 的文件；它就快完成了。 |

输出会列出所有目前正在经历恢复的索引，然后列出这些索引里的所有分片。每个分片里会有启动/停止时间、持续时间、恢复百分比、传输字节数等统计值。

**取消一个恢复**

要取消一个恢复，你需要删除正在恢复的索引。 因为恢复进程其实就是分片恢复，发送一个 删除索引 API 修改集群状态，就可以停止恢复进程。比如：

DELETE /restored\_index\_3

如果 restored\_index\_3 正在恢复中，这个删除命令会停止恢复，同时删除所有已经恢复到集群里的数据。

### 集群是活着的、呼吸着的生命

一旦你的集群投入生产，你会发现他就开始了他自己的一生。 Elasticsearch 努力工作来保证集群自给自足而且 *真就在工作* 。不过一个集群也还要有日常照料和投喂，比如日常备份和升级。

Elasticsearch 以非常快的速度发布新版本，进行错误修复和性能增强。保持你的集群采用最新版总是一个好主意。类似的，Lucene 持续在发现 JVM 自身的新的和令人惊讶的错误，这意味着你需要尽量保持你的 JVM 是最新的。

这意味着最好是拥有一个标准化的、日常的方案来操作你集群的滚动重启和升级。升级应该是一个日常程序，而不是一个需要好多个小时的精细规划下的年度『惨剧』。

类似的，拥有一个灾备计划是很重要的。请对你的集群做频繁的快照——而且通过执行真实恢复的方式定期 *测试* 这些快照！有些组织做日常备份却从不测试他们的恢复机制，这简直太常见了。通常你会在第一次演练真实恢复的时候发现明显的缺陷（比如用户不知道应该挂载哪个磁盘）。比起在凌晨 3 点真的发生危机的时候，在日常测试中暴露出这些问题总是更好的。