

Elasticsearch

威指南

序言

我 然清晰地 得那个日子，我 布了 个 源 目第一个版本并在 IRC 聊天室 建一个道，在那个最 的 刻，独自一人，急切地希望和 望着第一个用 的到来。

第一个跳 IRC 道的用 就是 Clint (克林)，当 我好 。好 ... 了一会直到我 Clint 上是 Perl 用 ， 是跟死亡 告 站打交道。我 得(当) 自己 什 他不是来自于更“主流”的社区，像 Ruby 或 Python，亦或是一个 微好点的使用案例。

我真是大 特 ! Clint 最 Elasticsearch 的成功起到了重要作用。他是第一个把 Elasticsearch 到生 境中的人 (是 0.4 的版本 !)，初期与 Clint 的互 于塑造 Elasticsearch 成今天的子非常 。 于什 是 ， Clint 有独特的 解并且他很少出 ， Elasticsearch 从管理、API 和日常使用等各个方面的易用性上面 生了很大的影 。 所以我 公司成立后不久，我 想也没想立即就 系 Clint 他是否 意加入我 。

当我 成立公司 ，我 做的第一件事就是提供公 培 。很 表 我 当 有多 和担心是否真的有人会 名。

但我 了。

培 到 在依然很成功，很多主要城市都 有大量的人等待参加。参加培 的人之中，有一个年 的家 吸引了我 注意，他的名字叫 Zach。 我 知道他有很多 于 Elasticsearch 的博客 (并暗自嫉他用非常 的方式来 述 概念的能力)，他 写了一个 PHP 的客 端。然后我 找到 Zach， 他是否 意加入我的公司。

Clint 和 Zach 是 Elasticsearch 能否成功的 。他 是完美的解 家，从 的上 用到的 (Apache Lucene 的) 底 。在 Elastic 里我 非常珍惜 独特技能。 Clint Elasticsearch Perl 客 端，而 Zach PHP，都是精彩的代 。

最后， 位在 Elasticsearch 目 天都在 生的大多数事情中也扮演着重要的角色。 Elasticsearch 如此受 迎的主要原因之一就是 有与用 通 生共 的能力， Clint 和 Zach 都是 个集体的一子， 一切成 可能。

Shay Banon

前言

个世界已然被数据淹没。多年来，我 系 流 和 生的大量数据已 我 不知所措。 有的技都集中在如何解决数据 存 以及如何 化 些数据。 些看上去都挺美好，直到 需要基于些数据 做决策分析的 时候才 根本不是那 一回事。

Elasticsearch 是一个分布式、可 展、 的搜索与数据分析引 。 它能从 目一 始就 予的数据以搜索、分析和探索的能力， 是通常没有 料到的。 它存在 因 原始数据如果只是 在磁里面根本就 无用 。

无 是需要全文搜索， 是 化数据的 ，或者 者 合， 本指南都能 助 了解其中最基本的概念， 从最基本的操作 始学 Elasticsearch。之后，我 会逐 始探索更加高 的搜索技， 不断提升搜索体 来 足 的用 需求。

Elasticsearch 不只是全文搜索，我将介绍化搜索、数据分析、的言理、地理位置和象系等。我将探如何数据建模来充分利用 Elasticsearch 的水平伸性，以及在生境中如何配置和的集群。

本

本是写任何想要把他数据拿来干活做点事情的人。不管是新起一自从始是了留系改造血，Elasticsearch都能助解决有和新的功能，有些可能是之前没有想到的功能。

本既合初学者也合有 的用。我希望有一定的程基，然不是必的，但有用SQL和系数据会更佳。我会从原理解和基本概念出，助新手在的搜索世界里得一个定的基。

具有搜索背景的者也会受益于本。有 的用将得其所熟悉搜索的概念是如何和具体 的。即使是高用，前面几个章所包含的信息也是非常有用的。

最后，也是一名DevOps，其他部一直尽可能快的往 Elasticsearch 里面灌数据，而是那个阻止 Elasticsearch 服器起火的消防。只要用在内行事，Elasticsearch 集群容相当松。不需知道如何在入生境前搭建一个定的集群，能在凌晨三点能出警告信号，以防止生。前面几章可能不太感趣，但本的最后一部分是非常重要的，包含所有需要知道的用以避免系崩的知。

什 我 要写 本

我写本，因 Elasticsearch 需要更好的述。有的参考文是秀的一前提是知道在什。它假定已熟悉信息索的概念、分布式系原理、Query DSL 和多其他相的概念。

本没有的假。它的目的是写一本即便是一个完全不的初学者（不管是搜索是分布式系）也能拿起它看完几章，就能始搭建一个原型。

我采取一基于求解的方式：是一个，我解决？如何候方案行衡取？我从基知始，循序，一章都建立在前一章之上，同提供必要的用案例和理解。

有的参考文解决了如何使用些功能，我希望本解决的是什和什候使用些功能。

Elasticsearch 版本

本的初始印刷版的是 Elasticsearch 1.4.0，不我一直在不断更新内容和完善示例本的上版本的是 Elasticsearch 2.x。

可以本的 GitHub 来追踪最新化。

如何 本

Elasticsearch 做了很多努力和来的事情得，很大程度上来 Elasticsearch 的成功来源于此。句，搜索以及分布式系是非常的，不早也需要掌握一些来充分利用 Elasticsearch。

恩，是有点，但不是魔法。我向于系如同神奇的子，能外部的咒，但是通常里面的工作很。理解了些程就能散魔法，理解内在能更加明和清晰，而不是寄托于子做想要做的。

本威指南不助学 Elasticsearch，而且接触更深入、更有趣的，如 [集群内的原理](#)、[分布式文存](#)、[行分布式索](#)和[分片内部原理](#)，些然不是必要的却能深入理解其内在机制。

本的第一部分是在按章序，因一章建立在上一章的基础上（尽管也可以才提到的章）。后各章如[近似匹配](#)和[部分匹配](#)相独立，可以根据需要性参。

本 航

本分七个部分：

- 章 知道的，了搜索... 到 [分片内部原理](#) 主要是介 Elasticsearch。介了 Elasticsearch 的数据入出以及 Elasticsearch 如何理的文数据。如何行基本的搜索操作和管理的索引。本章束将学会如何将 Elasticsearch 与的用程序集成。章：[集群内的原理](#)、[分布式文存](#)、[行分布式索](#)和[分片内部原理](#) 附加章，目的是了解分布式理的程，不是必的。
- 章 化搜索 到 控制相度 深入了解搜索，如何索引和的数据，并借助一些更高的特性，如近 (word proximity) 和部分匹配 (partial matching)。将了解相度分是如何工作的以及如何控制它来保第一是返回最佳的搜索果。
- 章 始理各言 到 写 解决如何有效使用分析器和来理言的痛。我会从一个的言分析下手，然后逐深入，如字母表和排序，会及到干提取、停用、同和模糊匹配。
- 章 高概念 到 Doc Values and Fielddata 聚合 (aggregations) 和分析，的数据行摘要化和分来呈体。
- 章 地理坐点 到 地理形状 介 Elasticsearch 支持的地理位置索方式：坐点和的地理形状 (geo-shapes)。
- 章 系理 到 容 到到了如何的数据建模来高效使用 Elasticsearch。在搜索引擎表体的系可能不是那容易，因它不是用来做个的。些章会述如何索引来匹配系中的数据流。
- 最后，章 控 到 部署后 将生境上 的重要配置、控点以及如何断以避免出。

在 源

因本重如何在 Elasticsearch 里解决，而不是法介，所以有候需要 [Elasticsearch 参考手](#) 来取明。可以以下址取最新的 Elasticsearch 参考手和相文：<https://www.elastic.co/guide/>

如果遇到本或者参考手没有收 到的，我建 Elasticsearch 社区来提，学人是如何使用 Elasticsearch 的或者分享自己的：

- [英文社区](#)
- [中文社区](#)

本 定

以下是本 中使用的印刷 :

Italic

表示重点、新的 或概念。

Constant width

用于程序列表以及在段落中引用 量或程序元素如：函数名称、数据 、数据 型、境 量、句和 字。

TIP 个 代表小 士，建 。

NOTE 个 代表一般注意事 。

WARNING 个 代表警告。

使用代 示例

本 的目的是 了 尽快能完成工作。一般来 ，本 提供的示例代 都可以用于 的程序或文 。不需要 系我 来 可，除非 打算 用相当大一部分代 。比如，写一个程序用了一段本 的代不需要 可，但是 或者是 行一 包含所有 O'Reilly 的示例代 的 CD 个就需要 可。引用本 、引用示例代 来回答 不需要 可，将大量的示例代 从 本 中包含到 的 品的文 中， 个需要 可。

于署名出 ，我 欣 但不是必 。一个出 通常包含： 名、作者、出版商和 ISBN。如：
Elasticsearch: The Definitive Guide by Clinton Gormley and Zachary Tong (O'Reilly). Copyright 2015 Elasticsearch BV, 978-1-449-35854-9。

如果 得 的示例代 使用超出合理使用或上面 出的 可，可随 与我 系
permissions@oreilly.com。

什 配偶 是被放到最后一个？但并非是 最不重要！ 在我 心中 无疑 ，有 个最 得我 感的人，他 是 Clinton 期受苦的老婆和 Zach 的未婚妻。 他 照 着我 和 着我 ， 不怠，忍受我 的 席和我 没完没了的抱怨 本 要多久完成，最重要的是， 依然 在我 身 。

感 Shay Banon 在最 始 建了 Elasticsearch，感 Elastic 公司支持本 的工作。也非常感 Elastic 所有的同事，他 助我 透 的了解 Elasticsearch 内部如何工作并且一直 添加完善和修 与他相 的部分。

其中 位同事特 得一提：

- Robert Muir 耐心地分享了他的真知灼 ，特 是 Lucene 搜索方面。有几章段落就是直接出自其智慧珠 。
- Adrien Grand 深入到代 中回答 ，并 我 的解 ，以 保他 合理。

感 O'Reilly 承担 个 目和我 一起工作使 本 免 在 ， 有一直温柔哄 我 的 Brian Anderson 和善良而温柔的 者 Benjamin Devèze、Ivan Brusic 和 Leo Lapworth。 的鼓励，我 充 希望。

感 我 的 者，其中一些我 只有通 各自的 GitHub 才知道他 的身 ， 他 花 告 、 提供修正或提出改 建：

Adam Canady, Adam Gray, Alexander Kahn, Alexander Reelsen, Alaattin Kahramanlar, Ambrose Ludd, Anna Beyer, Andrew Bramble, Baptiste Cabarrou, Bart Vandewoestyne, Bertrand Dechoux, Brian Wong, Brooke Babcock, Charles Mims, Chris Earle, Chris Gilmore, Christian Burgas, Colin Goodheart-Smithe, Corey Wright, Daniel Wiesmann, David Pilato, Duncan Angus Wilkie, Florian Hopf, Gavin Foo, Gilbert Chang, Grégoire Seux, Gustavo Alberola, Igal Sapir, Iskren Ivov Chernev, Itamar Syn-Hershko, Jan Forrest, Jānis Peisenieks, Japheth Thomson, Jeff Myers, Jeff Patti, Jeremy Falling, Jeremy Nguyen, J.R. Heard, Joe Fleming, Jonathan Page, Joshua Gourneau, Josh Schneier, Jun Ohtani, Keiji Yoshida, Kieren Johnstone, Kim Laplume, Kurt Hurtado, Laszlo Balogh, londocr, losar, Lucian Precup, Lukáš Vlček, Malibu Carl, Margirier Laurent, Martijn Dwars, Matt Ruzicka, Matthias Pfeiffer, Mehdy Amazigh, mhemani, Michael Bonfils, Michael Bruns, Michael Salmon, Michael Scharf , Mitar Milutinović, Mustafa K. Isik, Nathan Peck, Patrick Peschlow, Paul Schwarz, Pieter Coucke, Raphaël Flores, Robert Muir, Ruslan Zavacky, Sanglarsh Boudhh, Santiago Gaviria, Scott Wilkerson, Sebastian Kurfürst, Sergii Golubev, Serkan Kucukbay, Thierry Jossermoz, Thomas Cuchietti, Tom Christie, Ulf Reimers, Venkat Somula, Wei Zhu, Will Kahn-Greene 和 Yuri Bakumenko。

感 所有参与本 的中文 者与 校人 ， 他 牺了大量宝 的休息 ， 他 翻 内容仔 斟酌，一 不苟， 修改意 真 待，各抒己 ， 不 其 的 行修改与再次 校， 些 奉献的可 的人分 是： ， <http://github.com/luotitan>[朗]， <http://github.com/pengqiuyuan>[彭秋源]， <http://github.com/richardwei2008>[魏]， <http://github.com/chenrym>[琳]， <http://github.com/looly>[路小]， <http://github.com/michealzh>[michealzh]， <http://github.com/node>[nodexy]， <http://github.com/sdlyjzh>[sdlyjzh]， <http://github.com/wharstr9027>[落英流]， <http://github.com/sunyonggang>， <http://github.com/zhaochenxiao90>[Singham]， <http://github.com/Josephjin>[]， <http://github.com/lephix>[翔]， <http://github.com/lephix>[思]， <http://github.com/blogsit>[]， <http://github.com/倪侃>， <http://github.com/Geolem>[Geolem]， <http://github.com/JessicaWon>[卷]， <http://github.com/kfypmqqw>[kfypmqqw]， <http://github.com/weiqiangyuan>[袁]， <http://github.com/yichao2015>[yi chao]， <http://github.com/小彬>， <http://github.com/leo650>[leo]， <http://github.com/tangmisi>[tangmisi]， <http://github.com/cdma>[Alex]， <http://github.com/abia321>[baifan]， <http://github.com/EvanYellow>[Evan]， <http://github.com/fanyer>[fanyer]， <http://github.com/wwb>， <http://github.com/luoruixing>[瑞星]， <http://github.com/Miranda21>[碧琴]， <http://github.com/weikuo0506>[walker]， <http://github.com/javasgl>[songgl]， <http://github.com/碧琴>， <http://github.com/kankedong>[]， <http://github.com/smilesfc>[杜]， <http://github.com/qindongliang>[秦 亮]， <http://github.com/biyuhao>[biyuhao]， <http://github.com/LiuGangR>[]， <http://github.com/yumo>， <http://github.com/wangxiuwen>[王秀文]， <http://github.com/zcola>[zcola]， <http://github.com/gitqh>[gitqh]， <http://github.com/blackoon>[blackoon]， http://github.com/davidmr_001[David]， <http://github.com/stromdash>[辰]， <http://github.com/echolihao>， <http://github.com/cch123>[Xargin]， http://github.com/sunzh_enya[abel-sun]， <http://github.com/AlixMu>[]， <http://github.com/bsll>， <http://github.com/donglangdtstack>[冬狼]， <http://github.com/destinyfortune>[王]， <http://github.com/medcl>[Medcl]。

基 入

Elasticsearch 是一个 的分布式搜索分析引，它能 以一个之前从未有 的速度和 模，去探索 的数据。它被用作全文 索、 化搜索、分析以及 三个功能的 合：

- Wikipedia 使用 Elasticsearch 提供 有高亮片段的全文搜索， 有 *search-as-you-type* 和 *did-you-mean* 的建 。
- 使用 Elasticsearch 将 社交数据 合到 客日志中， 的 它的 提供公 于新文章的反 。
- Stack Overflow 将地理位置 融入全文 索中去，并且使用 *more-like-this* 接口去 相 的 与答案。
- GitHub 使用 Elasticsearch 1300 行代 行 。

然而 Elasticsearch 不 巨 公司服 。它也 助了很多初 公司，像 Datadog 和 Klout， 助他 将想法用原型 ，并 化 可 展的解决方案。Elasticsearch 能 行在 的 本 上，或者 展到上百台服 器上去 理PB 数据。

Elasticsearch 中没有一个 独的 件是全新的或者是革命性的。全文搜索很久之前就已 可以做到了， 就像早就出 了的分析系 和分布式数据 。 革命性的成果在于将 些 独的，有用的 件融合到一个 一的、一致的、 的 用中。它 于初学者而言有一个 低的 ， 而当 的技能提升或需求 加 ，它也始 能 足 的需求。

如果 在打 本 ，是因 有数据。除非 准 使用它 做些什 ，否 有 些数据将没有意 。

不幸的是，大部分数据 在从 的数据中提取可用知 出乎意料的低效。 当然， 可以通 或精 行 ，但是它 能 行全文 索、 理同 、通 相 性 文 分 ？ 它 从同 的数据中生成分析与聚合数据 ？最重要的是，它 能 地做到上面的那些而不 大型批 理的任 ？

就是 Elasticsearch 脱 而出的地方：Elasticsearch 鼓励 去探索与利用数据，而不是因 数据太困 ，就 它 在数据 里面。

Elasticsearch 将成 最好的朋友。

知道的， 了搜索...

Elasticsearch 是一个 源的搜索引，建立在一个全文搜索引 [Apache Lucene™](#) 基 之上。Lucene 可以 是当下最先 、高性能、全功能的搜索引 —无 是 源 是私有。

但是 Lucene 只是一个 。 了充分 其功能， 需要使用 Java 并将 Lucene 直接集成到 用程序中。更糟 的是， 可能需要 得信息 索学位才能了解其工作原理。Lucene 非常 。

Elasticsearch 也是使用 Java 写的，它的内部使用 Lucene 做索引与搜索，但是它的目的是使全文 索 得 ，通 藏 Lucene 的 性，取而代之的提供一套 一致的 RESTful API。

然而，Elasticsearch 不 是 Lucene，并且也不 只是一个全文搜索引擎 。 它可以被下面 准

的形容：

- 一个分布式的文存，一个字段可以被索引与搜索
- 一个分布式分析搜索引擎
- 能任上百个服务点的扩展，并支持PB的数据化或者非数据化数据

Elasticsearch 将所有的功能打包成一个独服，可以通程序与它提供的 RESTful API 行通信，可以使用自己喜的程言充当 web 客端，甚至可以使用命令行（去充当个客端）。

就 Elasticsearch 而言，起很。于初学者来，它了一些当的，并藏了的搜索理知。它箱即用。只需最少的理解，很快就能具有生产力。

随着知的累，可以利用 Elasticsearch 更多的高特性，它的整个引是可配置并且活的。从多高特性中，挑恰当去修的 Elasticsearch，使它能解决本地遇到的。

可以免下，使用，修改 Elasticsearch。它在 Apache 2 license 下布的，是多活的源之一。Elasticsearch 的源被托管在 Github 上 github.com/elastic/elasticsearch。如果想加入我一个令人奇的 contributors 社区，看里 [Contributing to Elasticsearch](#)。

如果 Elasticsearch 有任何相的，包括特定的特性(specific features)、言客端(language clients)、件(plugins)，可以在里 discuss.elastic.co 加入。

回光

多年前，一个婚的名叫 Shay Banon 的失者，跟着他的妻子去了敦，他的妻子在那里学厨。在第一个工作的候，为了他的妻子做一个食搜索引擎，他始使用 Lucene 的一个早期版本。

直接使用 Lucene 是很的，因此 Shay 始做一个抽象，Java 者使用它可以很的他的程序添加搜索功能。他布了他的第一个源目 Compass。

后来 Shay 得了一工作，主要是高性能，分布式境下的内存数据格。个于高性能，分布式搜索引擎的需求尤突出，他决定重写 Compass，把它一个独立的服并取名 Elasticsearch。

第一个公版本在2010年2月布，从此以后，Elasticsearch 已成了 Github 上最活的目之一，他有超 300 名 contributors(目前 736 名 contributors)。一家公司已始 Elasticsearch 提供商服，并新的特性，但是，Elasticsearch 将永源并所有人可用。

据，Shay 的妻子在等着的食搜索引擎 ...

安装并行 Elasticsearch

想用最的方式去理解 Elasticsearch 能做什，那就是使用它了，我始！

安装 Elasticsearch 之前，需要先安装一个新的版本的 Java，最好的是，可以从 www.java.com 得官方提供的最新版本的 Java。

之后，可以从 elastic 的官 [elasticsearch](http://elastic.co/downloads/elasticsearch) 取最新版本的 Elasticsearch。

要想安装 Elasticsearch，先下 并解 合 操作系 的 Elasticsearch 版本。如果 想了解更多的信息，可以 看 Elasticsearch 参考手 里 的安装部分， 出的 接指向安装 明 [Installation](#)。

TIP 当 准 在生 境安装 Elasticsearch ， 可以在 官 下 地址 到 Debian 或者 RPM 包，除此之外， 也可以使用官方支持的 [Puppet module](#) 或者 [Chef cookbook](#)。

当 解 好了 文件之后，Elasticsearch 已 准 好 行了。按照下面的操作，在前台(foreground) Elasticsearch：

```
cd elasticsearch-<version>
./bin/elasticsearch ① ②
```

① 如果 想把 Elasticsearch 作 一个守 程在后台 行，那 可以在后面添加参数 `-d`。

② 如果 是在 Windows 上面 行 Elasticsearch, 行 `bin\elasticsearch.bat` 而不是 `bin\elasticsearch`。

Elasticsearch 是否 成功，可以打 一个 端， 行以下操作：

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

TIP：如果 是在 Windows 上面 行 Elasticsearch， 可以从 <http://curl.haxx.se/download.html> 中下 cURL。cURL 提供了一 将 求提交到 Elasticsearch 的便捷方式，并且安装 cURL 之后，可以通 制与粘 去 中的 多例子。

得到和下面 似的 (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"
}
```

就意味着 在已 并 行一个 Elasticsearch 点了， 可以用它做 了。 个 点 可以作 一个 行中的 Elasticsearch 的 例。 而一个 集群 是一 有相同 `cluster.name` 的 点， 他 能一起工作并共享数据， 提供容 与可伸 性。(当然，一个 独的 点也 可以 成一个集群) 可以在 `elasticsearch.yml` 配置文件中 修改 `cluster.name`， 文件会在 点 加 (者注： 个重 服 后才会生效)。 于上面的 `cluster.name` 以及其它 [Important Configuration Changes](#) 信息，可以在 本 后面提供的生 部署章 到更多。

TIP : 看到下方的 View in Sense 的例子了 ? [Install the Sense console](#) 使用自己的 Elasticsearch 集群去运行本中的例子，看会有怎样的结果。

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

安装 Sense

Sense 是一个 [Kibana](#) 用它提供交互式的控制台，通过它的界面直接向 Elasticsearch 提交请求。本节的版本包含有一个 View in Sense 的连接，里面有多个示例。当点击的时候，它会打开一个对应的 Sense 控制台。不必安装 Sense，但是它允许在本地的 Elasticsearch 集群上运行示例代码，从而使本节更具有交互性。

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 ①
```

① Windows 上面运行：`bin\kibana.bat plugin --install elastic/sense`。

NOTE：可以直接从这里 <https://download.elastic.co/elastic/sense/sense-latest.tar.gz> 下载 Sense 安装可以看这里 [install it on an offline machine](#)。

2. 运行 Kibana。

```
./bin/kibana ①
```

① Windows 上运行 kibana：`bin\kibana.bat`。

3. 在浏览器中打开 Sense：<http://localhost:5601/app/sense>。

和 Elasticsearch 交互

和 Elasticsearch 的交互方式取决于是否使用 Java

Java API

如果正在使用 Java，在代码中可以使用 Elasticsearch 内置的一个客户端：

点客户端 (*Node client*)

点客户端作为一个非数据点加入到本地集群中。一句，它本身不保存任何数据，但是它知道数据在集群中的一个点中，并且可以把请求发送到正确的点。

客户端 (*Transport client*)

大量的客户端可以将请求送到远程集群。它本身不加入集群，但是它可以将请求发送到集群中的一个点上。

两个 Java 客户端都是通过 9300 端口并使用本地 Elasticsearch 和集群交互。集群中的点通

端口 9300 彼此通信。如果 个端口没有打 , 点将无法形成一个集群。

TIP Java 客 端作 点必 和 Elasticsearch 有相同的 主要 版本；否 ，它 之 将无法互相理解。

更多的 Java 客 端信息可以在 [Elasticsearch Clients](#) 中 到。

RESTful API with JSON over HTTP

所有其他 言可以使用 RESTful API 通 端口 9200 和 Elasticsearch 行通信，可以用 最喜 的 web 客 端 和 Elasticsearch 。事 上，正如 所看到的， 甚至可以使用 curl 命令来和 Elasticsearch 交互。

NOTE Elasticsearch 以下 言提供了官方客 端--Groovy、JavaScript、.NET、PHP、Perl、Python 和 Ruby— 有很多社区提供的客 端和 件，所有 些都可以在 [Elasticsearch Clients](#) 中 到。

一个 Elasticsearch 求和任何 HTTP 求一 由若干相同的部件 成：

```
curl -X<VERB> '<PROTOCOL>://<HOST>:<PORT>/<PATH>?<QUERY_STRING>' -d '<BODY>'
```

被 < > 的部件：

VERB

当的 HTTP 方法 或 : GET、POST、PUT、HEAD 或者 DELETE。

PROTOCOL

http 或者 https (如果 在 Elasticsearch 前面有一个 https 代理)

HOST

Elasticsearch 集群中任意 点的主机名，或者用 localhost 代表本地机器上的 点。

PORT

行 Elasticsearch HTTP 服 的端口号， 是 9200 。

PATH

API 的 端路径 (例如 _count 将返回集群中文 数量)。Path 可能包含多个 件，例如 : _cluster/stats 和 _nodes/stats/jvm 。

QUERY_STRING

任意可 的 字符串参数 (例如 ?pretty 将格式化地 出 JSON 返回 ，使其更容易)

BODY

一个 JSON 格式的 求体 (如果 求需要的)

例如， 算集群中文 的数量，我 可以用 个：

```
curl -XGET 'http://localhost:9200/_count?pretty' -d '  
{  
    "query": {  
        "match_all": {}  
    }  
}'
```

Elasticsearch 返回一个 HTTP 状态（例如：200 OK）和（除`HEAD`请求）一个 JSON 格式的返回。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 信息是因为我没有要求`curl`显示它。想要看到这些信息，需要结合 -i 参数来使用 curl 命令：

```
curl -i -XGET 'localhost:9200/'
```

在 中剩余的部分，我将用写格式来展示一些请求中所有相同的部分，例如主机名、端口号以及用一个完整的请求：

curl示例，所有的写格式就是省略命令本身。而不是像下面所示的那个

```
curl -XGET 'localhost:9200/_count?pretty' -d '  
{  
    "query": {  
        "match_all": {}  
    }  
}'
```

我将用写格式表示：

```
GET /_count  
{  
    "query": {  
        "match_all": {}  
    }  
}
```

事上，Sense 控制台也使用相同的格式。如果正在本的在版本，可以通点 Sense 接在 Sense 上打和行示例代。

面向文

在用程序中象很少只是一个的和的列表。通常，它有更的数据，可能包括日期、地理信息、其他象或者数等。

也有一天想把些象存 在数据中。使用系型数据的行和列存，相当于是一个表力富的象到一个非常大的子表格中：必 将个象扁平化来表—通常一个字段>一列—而且又不得不在次重新造象。

Elasticsearch 是面向文的，意味着它存整个象或文。Elasticsearch 不存文，而且索引个文的内容使之可以被索。在 Elasticsearch 中，文行索引、索、排序和—而不是行列数据。是一完全不同的思考数据的方式，也是 Elasticsearch 能支持全文索的原因。

JSON

Elasticsearch 使用 JavaScript Object Notation 或者 JSON 作文的序列化格式。JSON 序列化被大多数语言所支持，并且已成 NoSQL 域的准格式。它、易于。

考一下一个 JSON 文，它代表了一个 user 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"  
}
```

然原始的 user 象很，但个象的和含在 JSON 版本中都得到了体和保留。在 Elasticsearch 中将象化 JSON 并做索引要比在一个扁平的表中做相同的事情的多。

NOTE 几乎所有的言都有相的模可以将任意的数据或象化成 JSON 格式，只是各不相同。具体看 *serialization* 或者 *marshalling* 一个理 JSON 的模。官方 Elasticsearch 客端自提供 JSON 化。

新境

了 Elasticsearch 能什及其上手容易程度有一个基本印象，我从一个的教程始并介索引、搜索及聚合等基概念。

我将一并介一些新的技和基概念，因此即使无法立即全理解也无妨。在本后内容中，我

将深入介 里提到的所有概念。

接下来尽情享受 Elasticsearch 探索之旅。

建一个雇 目

我 受雇于 *Megacorp* 公司，作 HR 部 新的 “ 无人机” (“We love our drones!”) 激励 目的一部分，我 的任 是 此 建一个雇 目 。 目 当能培 雇 同感及支持 、高效、 作，因此有一些 需求：

- 支持包含多 数 以及全文本的数据
- 索任一雇 的完整信息
- 允 化搜索，比如 30 以上的 工
- 允 的全文搜索以及 的短 搜索
- 支持在匹配文 内容中高亮 示搜索片段
- 支持基于数据 建和管理分析 表

索引雇 文

第一个 需求就是存 雇 数据。 将会以 雇 文 的形式存：一个文 代表一个雇 。存 数据到 Elasticsearch 的行 叫做 索引，但在索引一个文 之前，需要 定将文 存 在 里。

一个 Elasticsearch 集群可以 包含多个 索引，相 的 个索引可以包含多个 型。 些不同的 型存 着多个文 ， 个文 又有 多个属性。

Index Versus Index Versus Index

也 已 注意到 索引 个 在 Elasticsearch 境中包含多重意思， 所以有必要做点儿 明：

索引（名）：

如前所述，一个 索引 似于 系数据 中的一个 数据 ， 是一个存 系型文 的地方。 索引 (*index*) 的 数 *indices* 或 *indexes* 。

索引（ ）：

索引一个文 就是存 一个文 到一个 索引（名） 中以便它可以被 索和 到。 非常 似于 SQL 句中的 **INSERT** ， 除了文 已存在 新文 会替 旧文 情况之外。

倒排索引：

系型数据 通 加一个 索引 比如一个 B (B-tree) 索引 到指定的列上，以便提升数据 索速度。 Elasticsearch 和 Lucene 使用了一个叫做 倒排索引 的 来 到相同的目的。

+ 的，一个文 中的 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 将在 [倒排索引](#) 倒排索引的更多 。

于雇 目 , 我 将做如下操作 :

- 个雇 索引一个文 , 包含 雇 的所有信息。
- 个文 都将是 `employee` 型。
- 型位于 索引 `megacorp` 内。
- 索引保存在我 的 Elasticsearch 集群中。

践中 非常 (尽管看起来有很多) , 我 可以通 一条命令完成所有 些 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

型名称

1

特定雇 的ID

求体 —— JSON 文 —— 包含了 位 工的所有 信息, 他的名字叫 John Smith , 今年 25 , 喜 岩。

很 ! 无需 行 行管理任 , 如 建一个索引或指定 个属性的数据 型之 的, 可以直接只索引一个 文 。Elasticsearch 地完成其他一切, 因此所有必需的管理任 都在后台使用 置完成。

行下一 前, 我 加更多的 工信息到目 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" ]
}
```

索文

目前我已在 Elasticsearch 中存了一些数据，接下来就能注于用的需求了。第一个需求是可以索到一个雇员的数据。

在 Elasticsearch 中很方便。地行一个 HTTP GET 请求并指定文档的地址——索引、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" ]
  }
}
```

TIP

将 HTTP 命令由 `PUT` 改为 `GET` 可以用来索文，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`。

批量搜索

一个 `GET` 是相当直接的，可以直接得到指定的文档。在点儿微高的功能，比如一个批量的搜索！

第一个几乎是最简单的搜索了。我使用下列请求来搜索所有雇员：

```
GET /megacorp/employee/_search
```

可以看到，我仍然使用索引 `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：返回结果不告知匹配了哪些文本，包含了整个文本本身：示搜索结果最通用所需的全部信息。

接下来，一下搜索姓氏 Smith 的雇员。此，我将使用一个高亮搜索，很容易通过命令行完成。一个方法一般涉及到一个字符串（query-string）搜索，因为我通过一个URL参数来信息搜索接口：

```
GET /megacorp/employee/_search?q=last_name:Smith
```

我果然在请求路径中使用 `_search` 端点，并将本身参数 `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" ]
        }
      }
    ]
  }
}
```

使用表式搜索

Query-string 搜索通过命令非常方便地进行个性化的即席搜索，但它有自身的局限性（参见量搜索）。Elasticsearch 提供一个灵活的方言叫做表式，它支持构建更加丰富和健壮的。

域特定语言（DSL），指定了使用一个 JSON 请求。我可以像重写之前的所有 Smith 的搜索：

```

GET /megacorp/employee/_search
{
  "query": {
    "match": {
      "last_name": "Smith"
    }
  }
}

```

返回 果与之前的 一，但 是可以看到有一些 化。其中之一是，不再使用 *query-string* 参数，而是一个 求体替代。 个 求使用 JSON 造，并使用了一个 **match** 型之一，后 将会了解）。

更 的搜索

在 下更 的搜索。 同 搜索姓氏 Smith 的雇 ，但 次我 只需要年 大于 30 的。 需要 作 整，使用 器 **filter**，它支持高效地 行一个 化 。

```

GET /megacorp/employee/_search
{
  "query": {
    "bool": {
      "must": {
        "match": {
          "last_name": "smith" ①
        }
      },
      "filter": {
        "range": {
          "age": { "gt": 30 } ②
        }
      }
    }
  }
}

```

① 部分与我 之前使用的 **match** 一。

② 部分是一个 **range** 器， 它能 到年 大于 30 的文 ，其中 **gt** 表示_大于_(great than)。

目前无需太多担心 法 ，后 会更 地介 。只需明 我 添加了一个 器 用于 行一个 ，并 用之前的 **match** 。 在 果只返回了一个雇 ，叫 Jane Smith, 32 。

```
{
  ...
  "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 都很 : 个姓名, 通 年 。 在 下 微高 点儿的全文搜索———
数据 很 定的任 。

搜索下所有喜 岩 (rock climbing) 的雇 :

```
GET /megacorp/employee/_search
{
  "query": {
    "match": {
      "about": "rock climbing"
    }
  }
}
```

然我 依旧使用之前的 `match` 在`about` 属性上搜索 ``rock climbing''。得到 个匹配的文 :

```
{
...
  "hits": {
    "total": 2,
    "max_score": 0.16273327,
    "hits": [
      {
        ...
        "_score": 0.16273327, ①
        "_source": {
          "first_name": "John",
          "last_name": "Smith",
          "age": 25,
          "about": "I love to go rock climbing",
          "interests": [ "sports", "music" ]
        }
      },
      {
        ...
        "_score": 0.016878016, ①
        "_source": {
          "first_name": "Jane",
          "last_name": "Smith",
          "age": 32,
          "about": "I like to collect rock albums",
          "interests": [ "music" ]
        }
      }
    ]
  }
}
```

① 相 性得分

Elasticsearch 按照相 性得分排序，即 个文 跟 的匹配程度。第一个最高得分的 果很明：John Smith 的 `about` 属性清楚地写着 ``rock climbing''。

但 什 Jane Smith 也作 果返回了 ？原因是 的 `about` 属性里提到了 `rock''`。因 只有 `rock''` 而没有 ``climbing''，所以 的相 性得分低于 John 的。

是一个很好的案例，明了 Elasticsearch 如何 在 全文属性上搜索并返回相 性最 的果。Elasticsearch中的 相 性 概念非常重要，也是完全区 于 系型数据 的一个概念，数据 中的一条 要 匹配要 不匹配。

短 搜索

出一个属性中的独立 是没有 的，但有 候想要精 匹配一系列 或者_短 _。 比如， 我想 行 一个 ， 匹配同 包含 `rock''` 和 `climbing''`， 并且 二者以短 ``rock climbing'' 的形式 挨着的雇 。

此 `match` 作 整，使用一个叫做 `match_phrase` 的：

```
GET /megacorp/employee/_search
{
  "query": {
    "match_phrase": {
      "about": "rock climbing"
    }
  }
}
```

无 念，返 回 果 有 John Smith 的文 。

```
{
  ...
  "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

多 用都 向于在 个搜索 果中 高亮 部分文本片段，以便 用 知道 何 文 符合 条件。在 Elasticsearch 中 索出高亮片段也很容易。

再次 行前面的 ，并 加一个新的 `highlight` 参数：

```

GET /megacorp/employee/_search
{
  "query": {
    "match_phrase": {
      "about": "rock climbing"
    }
  },
  "highlight": {
    "fields": {
      "about": {}
    }
  }
}

```

当行，返回果与之前一，与此同果中多了一个叫做 highlight 的部分。个部分包含了 about 属性匹配的文本片段，并以 HTML `` 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>" ①
          ]
        }
      }
    ]
  }
}

```

① 原始文本中的高亮片段

于高亮搜索片段，可以在 [highlighting reference documentation](#) 了解更多信息。

分析

于到了最后一个需求：支持管理者雇目做分析。Elasticsearch
有一个功能叫聚合（aggregations），允我基于数据生成一些精的分析果。聚合与SQL中的**GROUP BY**似但更大。

个例子，掘出雇中最受迎的趣好：

```
GET /megacorp/employee/_search
{
  "aggs": {
    "all_interests": {
      "terms": { "field": "interests" }
    }
  }
}
```

忽略掉法，直接看看果：

```
{
  ...
  "hits": { ... },
  "aggregations": {
    "all_interests": {
      "buckets": [
        {
          "key": "music",
          "doc_count": 2
        },
        {
          "key": "forestry",
          "doc_count": 1
        },
        {
          "key": "sports",
          "doc_count": 1
        }
      ]
    }
  }
}
```

可以看到，位工音感趣，一位林地感趣，一位感趣。些聚合并非先，而是从匹配当前的文中即生成。如果想知道叫Smith的雇中最受迎的趣好，可以直接添加当的来合：

```
GET /megacorp/employee/_search
{
  "query": {
    "match": {
      "last_name": "smith"
    }
  },
  "aggs": {
    "all_interests": {
      "terms": {
        "field": "interests"
      }
    }
  }
}
```

all_interests 聚合已 只包含匹配 的文 :

```
...
"all_interests": {
  "buckets": [
    {
      "key": "music",
      "doc_count": 2
    },
    {
      "key": "sports",
      "doc_count": 1
    }
  ]
}
```

聚合 支持分 。比如, 特定 趣 好 工的平均年 :

```
GET /megacorp/employee/_search
{
  "aggs" : {
    "all_interests" : {
      "terms" : { "field" : "interests" },
      "aggs" : {
        "avg_age" : {
          "avg" : { "field" : "age" }
        }
      }
    }
  }
}
```

得到的聚合 果有点儿 ，但理解起来 是很 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
      }
    }
  ]
}
```

出基本是第一次聚合的加 版。依然有一个 趣及数量的列表，只不 一个 趣都有了一个附加的 **avg_age** 属性，代表有 个 趣 好的所有 工的平均年 。

即使 在不太理解 些 法也没有 系，依然很容易了解到 聚合及分 通 Elasticsearch 特性 得很完美。可提取的数据 型 无限制。

教程

欣喜的是， 是一个 于 Elasticsearch 基 描述的教程，且 是浅 止，更多 如 suggestions 、geolocation、percolation、fuzzy 与 partial matching 等特性均被省略，以便保持教程的 。但它 突 了 始 建高 搜索功能多 容易。不需要配置——只需要添加数据并 始搜索！

很可能 法会 在某些地方有所困惑，并且 各个方面如何微 也有一些 。没 系！本 后 内容将 个 解 ， 全方位地理解 Elasticsearch 的工作原理。

分布式特性

在本章 ，我 提到 Elasticsearch 可以横向 展至数百（甚至数千）的服 器 点，同 可以理PB 数据。我 的教程 出了一些使用 Elasticsearch 的示例，但并不 及任何内部机制。Elasticsearch 天生就是分布式的，并且在 屏蔽了分布式的 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、分片、集群或其他的各 分布式概念。可以使用 本上的 点 松地 行教程里的程序，但如果 想要在 100 个 点的集群上 行程序，一切依然 。

Elasticsearch 尽可能地屏蔽了分布式系 的 性。 里列 了一些在后台自 行的操作：

- 分配文 到不同的容器 或 分片 中，文 可以 存在一个或多个 点中
- 按集群 点来均衡分配 些分片，从而 索引和搜索 程 行 均衡
- 制 个分片以支持数据冗余，从而防止硬件故障 致的数据 失
- 将集群中任一 点的 求路由到存有相 数据的 点
- 集群 容 无 整合新 点，重新分配分片以便从 群 点恢

当 本 ，将会遇到有 Elasticsearch 分布式特性的 充章 。 些章 将介 有 集群容、故障 移(集群内的原理) 、 文 存 (分布式文 存) 、 行分布式搜索(行分布式 索) ，以及分区 (shard) 及其工作原理(分片内部原理) 。

些章 并非必 ，完全可以无需了解内部机制就使用 Elasticsearch，但是它 将从 一个角度 助 了解更完整的 Elasticsearch 知 。可以根据需要跳 它 ，或者想更完整地理解 再回 也无妨。

后

在 于通 Elasticsearch 能 什 的功能、以及上手的 易程度 有了初 概念。Elasticsearch 力 通 最少的知 和配置做到 箱即用。学 Elasticsearch 的最好方式是投 入 践：尽管 始索引和搜索 ！

然， 于 Elasticsearch 知道得越多，就越有生 效率。告 Elasticsearch 越多的 域知 ，就越容易 行 果 。

本 的后 内容将 助 从新手成 家， 个章 不 述必要的基 知 ，而且包含 家建 。如果 上手， 些建 可能无法立竿 影；但 Elasticsearch 有着合理的 置，在无需干 的情况下通常都能工作得很好。当追求 秒 的性能提升 ，随 可以重温 些章 。

集群内的原理

充章

如前文所述， 是 充章 中第一篇介 Elasticsearch 在分布 式 境中的 行原理。 在 个章 中，我 将会介 cluster 、 node 、 shard 等常用 ， Elasticsearch 的 容机制， 以及如何 理硬件故障的内容。

然 个章 不是必 的— 完全可以在不 注分片、副本和失效切 等内容的情况下 期使用Elasticsearch-- 但是 将 助 了解 Elasticsearch 的内部工作 程。 可以先快速 章 ，将来有需要 再次 看。

ElasticSearch 的主旨是随 可用和按需 容。而 容可以通 性能更 大（垂 直 容，或 水 平 容）或者数量更多的服 器（水 平 容，或 横向 容）来 。

然 Elasticsearch 可以 益于更 大的硬件 ，但是垂直 容是有 限的。 真正的
容能力是来自于水平 容— 集群添加更多的 点，并且将 力和 定性分散到 些 点中。

于大多数的数据 而言，通常需要 用程序 行非常大的改 ，才能利用上横向 容的新 源。
与之相反的是，ElasticSearch天生就是 分布式的 ，它知道如何通 管理多 点来提高 容性和可用性。
也意味着 的 用无需 注 个 。

本章将 述如何按需配置集群、 点和分片，并在硬件故障 保数据安全。

空集群

如果我 了一个 独的 点，里面不包含任何的数据和索引，那我 的集群看起来就是一个
包含空内容 点的集群。

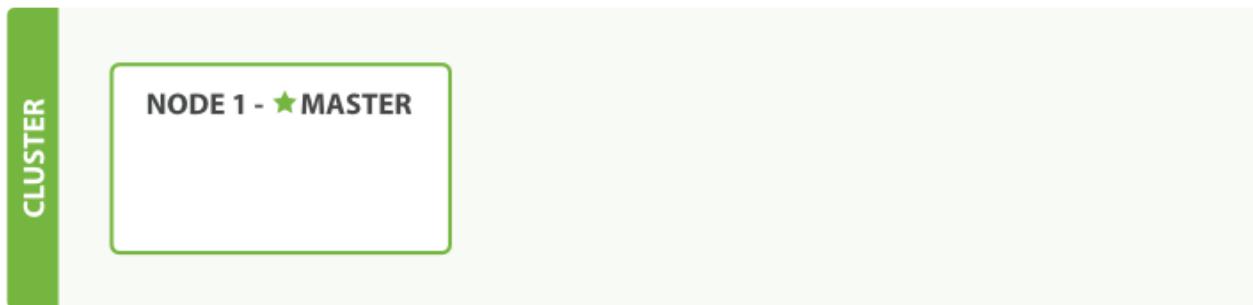


Figure 1. 包含空内容 点的集群

一个 行中的 Elasticsearch 例称 一个 点，而集群是由一个或者多个 有相同 `cluster.name` 配置的
点 成， 它 共同承担数据和 的 力。当有 点加入集群中或者从集群中移除 点
，集群将会重新平均分布所有的数据。

当一个 点被 成 主 点 ， 它将 管理集群 内的所有 更，例如 加、 除索引，或者
加、 除 点等。 而主 点并不需要 及到文 的 更和搜索等操作，所以当集群只 有一个主
点的情况下，即使流量的 加它也不会成 瓶 。 任何 点都可以成 主 点。我
的示例集群就只有一个 点，所以它同 也成 了主 点。

作 用 ，我 可以将 求 送到 集群中的任何 点 ，包括主 点。 个 点都知道任意文 所
的位置，并且能 将我 的 求直接 到存 我 所需文 的 点。 无 我 将 求 送到 个
点，它都能 从各个包含我 所需文 的 点收集回数据，并将最 果返回给客 端。
Elasticsearch 一切的管理都是透明的。

集群健康

Elasticsearch 的集群 控信息中包含了 多的 数据，其中最 重要的一 就是 集群健康 ， 它在
`status` 字段中展示 `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 似于如下所示的返回内容：

```
{
  "cluster_name": "elasticsearch",
  "status": "green", ①
  "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
}
```

① **status** 字段是我最关心的。

status 字段指示着当前集群在体上是否工作正常。它的三色含义如下：

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。

在本章剩余的部分，我将解释什么是主分片和副本分片，以及上面提到的一些颜色的意义。

添加索引

我往 Elasticsearch 添加数据需要用到索引——保存相数据的地方。索引上是指向一个或者多个物理分片的命名空间。

一个分片是一个底层的工作单元，它保存了全部数据中的一部分。在**分片内部机制**中，我将介绍分片是如何工作的，而在我只需知道一个分片是一个 Lucene 的例子，以及它本身就是一个完整的搜索引擎。我的文章被存储和索引到分片内，但是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到集群内各的。分片是数据的容器，文保存在分片内，分片又被分配到集群内的各个点里。当的集群模大或者小，Elasticsearch 会自动的在各点中移分片，使得数据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属于一个主分片，所以主分片的数目决定着索引能保存的最大数据量。

NOTE 技上来，一个主分片最大能存 Integer.MAX_VALUE - 128 个文档，但是最大需要参考的使用场景：包括使用的硬件，文档的大小和程度，索引和文档的方式以及期望的。

一个副本分片只是一个主分片的拷贝。副本分片作硬件故障保证数据不失冗余，并搜索和返

回文 等 操作提供服 。

在索引建立的 候就已 定了主分片数，但是副本分片数可以随 修改。

我 在包含一个空 点的集群内 建名 **blogs** 的索引。 索引在 情况下会被分配5个主分片， 但是 了演示目的，我 将分配3个主分片和一 副本（ 个主分片 有一个副本分片）：

```
PUT /blogs
{
  "settings" : {
    "number_of_shards" : 3,
    "number_of_replicas" : 1
  }
}
```

我 的集群 在是 [有一个索引的 点集群](#)。所有3个主分片都被分配在 **Node 1**。

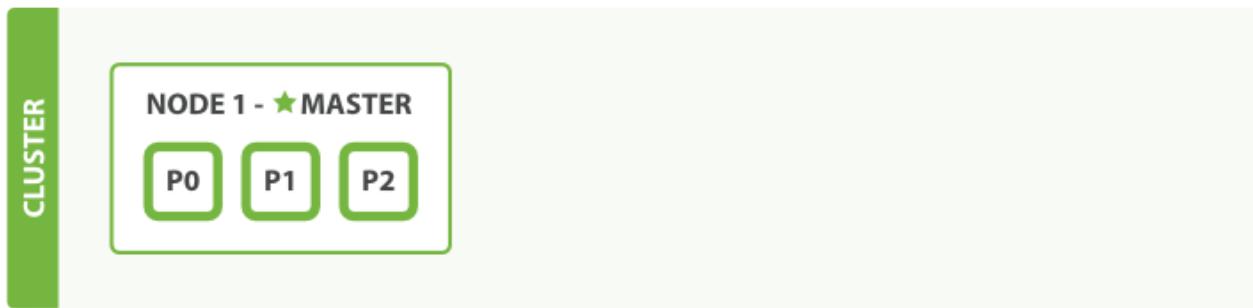


Figure 2. 有一个索引的 点集群

如果我 在 看**集群健康**，我 将看到如下内容：

```
{
  "cluster_name": "elasticsearch",
  "status": "yellow", ①
  "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, ②
  "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
}
```

- ① 集群 status yellow。
- ② 没有被分配到任何 点的副本数。

集群的健康状况 yellow 表示全部 主 分片都正常 行 (集群可以正常服 所有 求), 但是 副本分片没有全部 在正常状 。 上, 所有3个副本分片都是 unassigned —— 它 都没有被分配到任何 点。 在同一个 点上既保存原始数据又保存副本是沒有意 的, 因 一旦失去了那个 点, 我 也将失 点上的所有副本数据。

当前我 的集群是正常 行的, 但是在硬件故障 有 失数据的 。

添加故障 移

当集群中只有一个 点在 行 , 意味着会有一个 点故障 ——没有冗余。 幸 的是, 我 只需再一个 点即可防止数据 失。

第二个 点

了 第二个 点 后的情况, 可以在同一个目 内, 完全依照 第一个 点的方式来一个新 点 (参考[安装并 行 Elasticsearch](#))。多个 点可以共享同一个目 。

当 在同一台机器上 了第二个 点 , 只要它和第一个 点有同 的 配置, 它就会自 集群并加入到其中。 但是在不同机器上 为了加入到同一集群, 需要配置一个可 接到的 播主机列表。 信息 看[最好使用播代替 播](#)

如果 了第二个 点, 我 的集群将会如 有 个 点的集群——所有主分片和副本分片都已被分配所示。

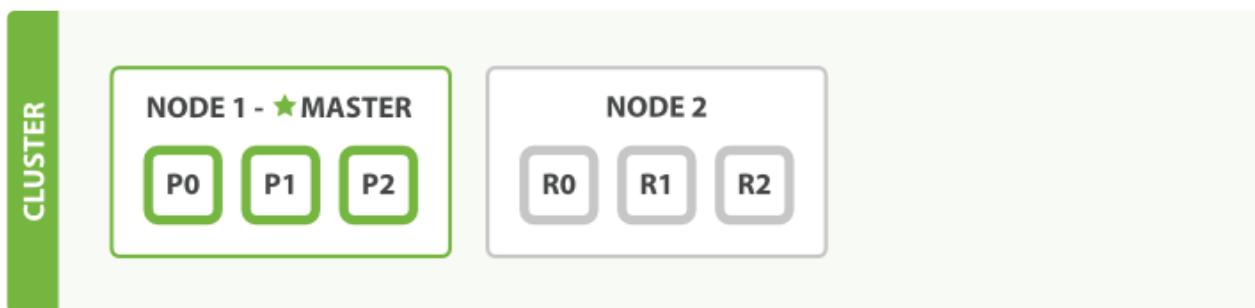


Figure 3. 有 个 点的集群——所有主分片和副本分片都已被分配

当第二个 点加入到集群后, 3个 副本分片 将会分配到 个 点上—— 个主分片 一个副本分片。 意味着当集群内任何一个 点出 , 我 的数据都完好无 。

所有新近被索引的文 都将会保存在主分片上, 然后被并行的 制到 的副本分片上。 就保 了我 既可以从主分片又可以从副本分片上 得文 。

`cluster-health` 在展示的状 green , 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 行。

```
{
  "cluster_name": "elasticsearch",
  "status": "green", ①
  "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
}
```

① 集群 status green。

我的集群 在不行的，并且于始可用的状态。

水平 容

我正在中的用程序按需容？当了第三个点，我的集群将会看起来如有三个点的集群——了分散而分片行重新分配所示。

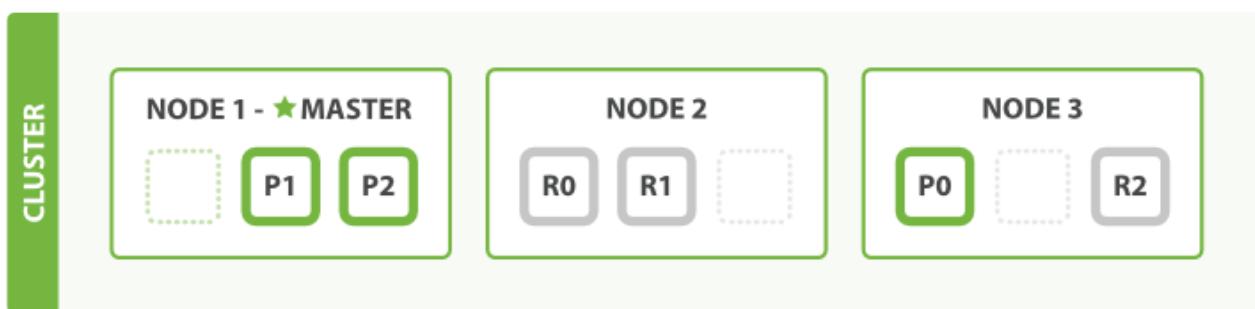


Figure 4. 有三个点的集群——了分散而分片行重新分配

Node 1 和 Node 2 上各有一个分片被移到了新的 Node 3 点，在个点上都 有2个分片，而不是之前的3个。表示个点的硬件源(CPU, RAM, I/O)将被更少的分片所共享，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，它有使用一个点上的所有源的能力。我一个有6个分片(3个主分片和3个副本分片)的索引可以最大容到6个点，一个点上存在一个分片，并且一个分片有所在点的全部源。

更多的 容

但是如果我 想要 容超 6个 点 ?

主分片的数目在索引 建 就已 定了下来。 上， 个数目定 了 个索引能 存 的最大数据量。（ 大小取决于 的数据、硬件和使用 景。） 但是， 操作——搜索和返回数据 ——可以同 被主分片 或 副本分片所 理， 所以当 有越多的副本分片 ， 也将 有越高的 吐量。

在 行中的集群上是可以 整副本分片数目的， 我 可以按需伸 集群。 我 把副本数从 1 加到 2 :

```
PUT /blogs/_settings
{
  "number_of_replicas" : 2
}
```

如将参数 `number_of_replicas` 大到 2所示， `blogs` 索引 在 有9个分片：3个主分片和6个副本分片。 意味着我 可以将集群 容到9个 点， 个 点上一个分片。相比原来3个 点， 集群搜索性能可以提升 3 倍。

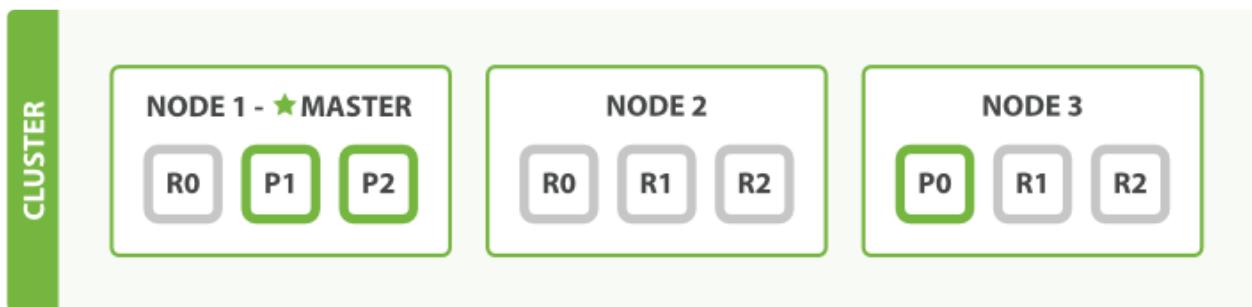


Figure 5. 将参数 `number_of_replicas` 大到 2

当然，如果只是在相同 点数目的集群上 加更多的副本分片并不能提高性能，因 个分 片从 点上 得的 源会 少。 需要 加更多的硬件 源来提升 吐量。

NOTE

但是更多的副本分片提高了数据冗余量：按照上面的 点配置，我 可以在失去2个 点的情况下不 失任何数据。

故障

我 之前 Elasticsearch 可以 点故障， 接下来 我 下 个功能。 如果我 第一个 点， 集群的状 了一个 点后的集群

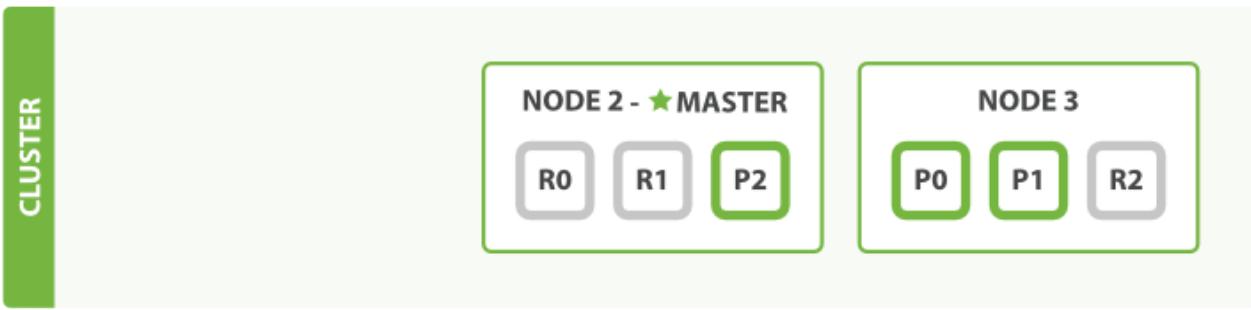


Figure 6. 了一个 点后的集群

我 的 点是一个主 点。而集群必 有一个主 点来保 正常工作，所以 生的第一件事情就是 一个新的主 点：Node 2。

在我 Node 1 的同 也失去了主分片 1 和 2，并且在 失主分片的 候索引也不能正常工作。如果此 来 集群的状况，我 看到的状 将会 red：不是所有主分片都在正常工作。

幸 的是，在其它 点上存在着 个主分片的完整副本， 所以新的主 点立即将 些分片在 Node 2 和 Node 3 上 的副本分片提升 主分片， 此 集群的状 将会 yellow。 个提升主分片的 程是瞬 生的，如同按下一个 一般。

什 我 集群状 是 yellow 而不是 green ？ 然我 有所有的三个主分片，但是同 置了 个主分片需要 2 副本分片，而此 只存在一 副本分片。 所以集群不能 green 的状 ，不 我不必 于担心：如果我 同 了 Node 2， 我 的程序 依然 可以保持在不 任何数据的情况下 行，因 Node 3 一个分片都保留着一 副本。

如果我 重新 Node 1， 集群可以将 失的副本分片再次 行分配，那 集群的状 也将如[将参数 number_of_replicas 大到 2所示](#)。 如果 Node 1 依然 有着之前的分片，它将 去重用它， 同 从主分片 制 生了修改的数据文件。

到目前 止， 分片如何使得 Elasticsearch 行水平 容以及数据保障等知 有了一定了解。接下来我 将 述 于分片生命周期的更多 。

数据 入和 出

无 我 写什 的程序，目的都是一 的：以某 方式 数据服 我 的目的。 但是数据不 由随机位和字 成。我 建立数据元素之 的 系以便于表示 体，或者 世界中存在的 事物 。如果我 知道一个名字和 子 件地址属于同一个人，那 它 将会更有意 。

尽管在 世界中，不是所有的 型相同的 体看起来都是一 的。 一个人可能有一个家庭 号，而 一个人只有一个手机号，再一个人可能 者兼有。 一个人可能有三个 子 件地址，而 一个人却一个都没有。一位西班牙人可能有 个姓，而 英 的人可能只有一个姓。

面向 象 程 言如此流行的原因之一是 象 我 表示和 理 世界具有潜在的 的数据 的 体，到目前 止，一切都很完美！

但是当我 需要存 些 体 来了， 上，我 以行和列的形式存 数据到 系型数据 中，相当 于使用 子表格。正因 我 使用了 不 活的存 媒介 致所有我 使用 象的 活性都 失了。

但是否我 可以将我 的 象按 象的方式来存 ？ 我 就能更加 注于 使用 数据，而不是在子表格的局限性下 我 的 用建模。我 可以重新利用 象的 活性。

一个 象 是基于特定 语的内存的数据 。 了通 送或者存 它，我 需要将它表示成某准的格式。 JSON 是一 以人可 读的文本表示 象的方法。 它已 成 NoSQL 世界交 数据的事准。当一个 象被序列化成 JSON，它被称 一个JSON文 。

Elasticsearch 是分布式的 文 存 。它能存 和 索 的数据 —序列化成 JSON文 —以的方式。 句 ，一旦一个文 被存 在 Elasticsearch 中，它就是可以被集群中的任意 点 索到。

当然，我 不 要存 数据，我 一定 需要 它，成批且快速的 它 。 尽管 存的 NoSQL 解决方案允 我 以文 的形式存 象，但是他 旧需要我 思考如何 我 的数据，以及 定 些 字段需要被索引以加快数据 索。

在 Elasticsearch 中， 个字段的所有数据 都是 被索引的 。 即 个字段都有 了快速 索 置的 用倒排索引。而且，不像其他多数的数据 ，它能在 相同的 中 使用所有 些倒排索引，并以 人的速度返回 果。

在本章中，我 展示了用来 建， 索，更新和 除文 的 API。就目前而言，我 不 心文 中的数据或者 它 。 所有我 心的就是在 Elasticsearch 中 安全的存 文 ，以及如何将文 再次返回。

什 是文 ？

在大多数 用中，多数 体或 象可以被序列化 包含 的 JSON 象。 一个 可以是一个字段或字段的名称，一个 可以是一个字符串，一个数字，一个布 ， 一个 象，一些数 ， 或一些其它特殊 型 如表示日期的字符串，或代表一个地理位置的 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，我使用的象和文是可以互相替换的。不，有一个区别：一个象是似于 hash、hashmap、字典或者数的 JSON 象，象中也可以嵌套其他的象。象可能包含了外一些象。在 Elasticsearch 中，文有着特定的含义。它是指最或者根象，一个根象被序列化成 JSON 并存到 Elasticsearch 中，指定了唯一 ID。

WARNING 字段的名字可以是任何合法的字符串，但不可以包含段。

文元数据

一个文不包含它的数据，也包含元数据。有文的信息。三个必有的元数据元素如下：

_index

文在存放

_type

文表示的象

_id

文唯一

_index

一个索引是因共同的特性被分到一起的文集合。例如，可能存所有的品在索引 products 中，而存所有交易到索引 sales 中。然也允存不相的数据到一个索引中，但通常看作是一个反模式的做法。

TIP

上，在 Elasticsearch 中，我的数据是被存和索引在分片中，而一个索引是上的命名空间，一个命名空间由一个或者多个分片合在一起。然而，是一个内部，我的用程序根本不关心分片，于用程序而言，只需知道文位于一个索引内。Elasticsearch 会理所有的。

我将在索引管理介如何自行建和管理索引，但在我将 Elasticsearch 我建索引。所有需要我做的就是一个索引名，这个名字必小写，不能以下，不能包含逗号。我用 website 作索引名例。

_type

数据可能在索引中只是松散的合在一起，但是通常明定一些数据中的子分区是很有用的。例如，所有的品都放在一个索引中，但是有多不同的品，比如 "electronics"、"kitchen" 和 "lawn-care"。

些文共享一相同的（或非常相似）的模式：他有一个、描述、品代和格。他只是正好属于“品”下的一些子。

Elasticsearch 公了一个称 types（型）的特性，它允的文可能有不同的字段，但最好能非常相似。我将在型和映射中更多的于 types 的一些用和限制。

一个 `_type` 命名可以是大写或者小写，但是不能以下 或者句号 ，不 包含逗号，并且度限制 256个字符。我 使用 `blog` 作 型名 例。

`_id`

`ID` 是一个字符串，当它和 `_index` 以及 `_type` 合就可以唯一 定 Elasticsearch 中的一个文 。当建一个新的文 ，要 提供自己的 `_id`，要 Elasticsearch 生成。

其他元数据

有一些其他的元数据元素，他 在 型和映射 行了介 。通 前面已 列出的元数据元素，我 已能存 文 到 Elasticsearch 中并通 `ID` 索它— 句 ，使用 Elasticsearch 作 文 的存 介 。

索引文

通 使用 `<code>index</code>` API ，文 可以被 `索引` —— 存 和使文 可被搜索。但是首先，我 要 定文 的位置。正如我 的，一个文 的 `<code>_index</code>` 、 `<code>_type</code>` 和 `<code>_id</code>` 唯一 一个文 。 我 可以提供自定 的 `<code>_id</code>` ，或者 `<code>index</code>` API 自 生成。

使用自定 的 ID

如果 的文 有一个自然的 符 （例如，一个 `user_account` 字段或其他 文 的）， 使用如下方式的 `index` API 并提供 自己 `_id`：

```
PUT /{index}/{type}/{id}
{
  "field": "value",
  ...
}
```

个例子，如果我 的索引称 `website`，型称 `blog`，并且 `123` 作 `ID`，那 索引 求 是下面：

```
PUT /website/blog/123
{
  "title": "My first blog entry",
  "text": "Just trying this out...",
  "date": "2014/01/01"
}
```

Elasticsearch 体如下所示：

```
{  
  "_index": "website",  
  "_type": "blog",  
  "_id": "123",  
  "_version": 1,  
  "created": true  
}
```

表明文 已 成功 建， 索引包括 `_index`、`_type` 和 `_id` 元数据， 以及一个新元素：`_version`。

在 Elasticsearch 中 个文 都有一个版本号。当 次 文 行修改 （包括 除）， `_version` 的会 。 在 理冲突 中， 我 了 使用 `_version` 号 保 的 用程序中的一部分修改不会覆一部分所做的修改。

Autogenerating IDs

如果 的数据没有自然的 ID， Elasticsearch 可以 我 自 生成 ID。 求的 整： 不再使用 `PUT`（“使用 个 URL 存 个文 ”）， 而是使用 `POST`（“存 文 在 个 URL 命名空 下”）。

在 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 生成的， 的其他部分和前面的 似：

```
{  
  "_index": "website",  
  "_type": "blog",  
  "_id": "AVFgSgVHUP18jI2wRx0w",  
  "_version": 1,  
  "created": true  
}
```

自 生成的 ID 是 URL-safe、 基于 Base64 且 度 20个字符的 GUID 字符串。 些 GUID 字符串由可修改的 FlakeID 模式生成， 模式允 多个 点并行生成唯一 ID ， 且互相之 的冲突概率几乎 零。

取回一个文

了从 Elasticsearch 中 索出文 ， 我 然使用相同的 `_index`，`_type`， 和 `_id`， 但是 HTTP 更改 `GET`：

```
GET /website/blog/123?pretty
```

体包括目前已熟悉了的元数据元素，再加上 `_source` 字段，一个字段包含我索引数据送 Elasticsearch 的原始 JSON 文：

```
{  
    "_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"  
    }  
}
```

NOTE

在求的串参数中加上 `pretty` 参数，正如前面的例子中看到的，将会用 Elasticsearch 的 *pretty-print* 功能，功能使得 JSON 体更加可读。但是，`_source` 字段不能被格式化打印出来。相反，我得到的 `_source` 字段中的 JSON 串，好是和我一样的一。

GET 求的体包括 `{"found": true}`，为了文已被到。如果我求一个不存在的文，我旧会得到一个 JSON 体，但是 `found` 将会是 `false`。此外，HTTP 将会是 `404 Not Found`，而不是 `200 OK`。

我可以通 `-i` 参数 `curl` 命令，参数能示的部：

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

示部的体在似：

```
HTTP/1.1 404 Not Found  
Content-Type: application/json; charset=UTF-8  
Content-Length: 83
```

```
{  
    "_index": "website",  
    "_type": "blog",  
    "_id": "124",  
    "found": false  
}
```

返回文的一部分

情况下，`GET` 求会返回整个文，一个文正如存 在 `_source` 字段中的一。但是也只其中的 `title` 字段感 趣。一个字段能用 `_source` 参数求得到，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
```

`_source` 字段 在包含的只是我 求的那些字段，并且已 将 `date` 字段 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 只想得到 `_source` 字段，不需要任何元数据，能使用 `_source` 端点：

```
GET /website/blog/123/_source
```

那 返回的的内容如下所示：

```
{
  "title": "My first blog entry",
  "text": "Just trying this out...",
  "date": "2014/01/01"
}
```

文 是否存在

如果只想 一个文 是否存在--根本不想 心内容—那 用 `HEAD` 方法来代替 `GET` 方法。`HEAD` 求没有返回体，只返回一个 HTTP 求：

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

如果文 存在，Elasticsearch 将返回一个 `200 ok` 的状：

```
HTTP/1.1 200 OK
Content-Type: text/plain; charset=UTF-8
Content-Length: 0
```

若文不存在，Elasticsearch 将返回一个 **404 Not Found** 的状：

```
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，一个文是在候不存在，并不意味着一秒之后它也不存在：也同样正好一个程就建了文。

更新整个文

在 Elasticsearch 中文是不可改的，不能修改它。相反，如果想要更新有的文，需要重建索引或者行替，我可以使用相同的 **index API** 行，在索引文中已行了。

```
PUT /website/blog/123
{
  "title": "My first blog entry",
  "text": "I am starting to get the hang of this...",
  "date": "2014/01/02"
}
```

在体中，我能看到 Elasticsearch 已加了 **_version** 字段：

```
{
  "_index": "website",
  "_type": "blog",
  "_id": "123",
  "_version": 2,
  "created": false ①
}
```

① **created** 志置成 **false**，是因相同的索引、型和 ID 的文已存在。

在内部，Elasticsearch 已将旧文已除，并加一个全新的文。尽管不能再旧版本的文行，但它并不会立即消失。当索引更多的数据，Elasticsearch 会在后台清理些已除文。

在本章的后面部分，我会介**update API**，个 API 可以用于 **partial updates to a document**。

然它似乎 文 直接 行了修改，但 上 Elasticsearch 按前述完全相同方式 行以下 程：

1. 从旧文 建 JSON
2. 更改 JSON
3. 除旧文
4. 索引一个新文

唯一的区 在于，`update` API 通 一个客 端 求来 些 ， 而不需要 独的 `get` 和 `index` 求。

建新文

当我 索引一个文 ， 我 正在 建一个完全新的文 ， 而不是覆 有的 ？

住，`_index` 、`_type` 和 `_id` 的 合可以唯一 一个文 。所以，保 建一个新文 的最 法是，使用索引 求的 `POST` 形式 Elasticsearch 自 生成唯一 `_id`：

```
POST /website/blog/  
{ ... }
```

然而，如果已 有自己的 `_id`，那 我 必 告 Elasticsearch，只有在相同的 `_index`、`_type` 和 `_id` 不存在 才接受我 的索引 求。 里有 方式，他 做的 是相同的事情。使用 ， 取决于 使 用起来更方便。

第一 方法使用 `op_type` -字符串参数：

```
PUT /website/blog/123?op_type=create  
{ ... }
```

第二 方法是在 URL 末端使用 `_create`：

```
PUT /website/blog/123/_create  
{ ... }
```

如果 建新文 的 求成功 行，Elasticsearch 会返回元数据和一个 `201 Created` 的 HTTP 。

一方面，如果具有相同的 `_index`、`_type` 和 `_id` 的文 已 存在，Elasticsearch 将会返回 `409 Conflict`，以及如下的 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  
}
```

除文

除文 的 法和我 所知道的 相同，只是使用 **DELETE** 方法：

```
DELETE /website/blog/123
```

如果 到 文 ， Elasticsearch 将要返回一个 **200 ok** 的 HTTP ， 和一个 似以下 的 体。注意，字段 **_version** 已 加：

```
{  
  "found" : true,  
  "_index" : "website",  
  "_type" : "blog",  
  "_id" : "123",  
  "_version" : 3  
}
```

如果文 没有 到，我 将得到 **404 Not Found** 的 和 似 的 体：

```
{  
  "found" : false,  
  "_index" : "website",  
  "_type" : "blog",  
  "_id" : "123",  
  "_version" : 4  
}
```

即使文 存在 (`Found` 是 `false`), `_version` 然会 加。是 Elasticsearch 内部本的一部分, 用来 保 些改 在跨多 点 以正 的 序 行。

正如已 在更新整个文 中提到的, 除文 不会立即将文 从磁 中 除, 只是将文
NOTE 已 除状 。随着 不断的索引更多的数据, Elasticsearch 将会在后台清理
已 除的文 。

理冲突

当我 使用 `index` API 更新文 , 可以一次性 取原始文 , 做我 的修改, 然后重新索引 整个文 。最近的索引 求将 :无 最后 一个文 被索引, 都将被唯一存 在 Elasticsearch 中。如果其他人同 更改 个文 , 他 的更改将 失。

很多 候 是没有 的。也 我 的主数据存 是一个 系型数据 , 我 只是将数据 制到 Elasticsearch 中并使其可被搜索。也 个人同 更改相同的文 的几率很小。或者 于我 的 来偶 失更改并不是很 重的 。

但有 失了一个 更就是非常 重的。想我 使用 Elasticsearch 存 我 上商城商品 存的数量, 次我 一个商品的 候, 我 在 Elasticsearch 中将 存数量 少。

有一天, 管理 决定做一次促 。突然地, 我 一秒要 好几个商品。假 有个 web 程序并行 行, 一个都同 理所有商品的 , 如 [Consequence of no concurrency control](#) 所示。

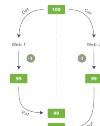


Figure 7. Consequence of no concurrency control

`web_1` `stock_count` 所做的更改已 失, 因 `web_2` 不知道它的 `stock_count` 的拷 已 期。果我 会 有超 商品的 数量的 存, 因 客的 存商品并不存在, 我 将 他 非常失望。

更越 繁, 数据和更新数据的 隙越 , 也就越可能 失 更。

在数据 域中, 有 方法通常被用来 保并 更新 更不会 失 :

悲 并 控制

方法被 系型数据 广泛使用, 它假定有 更冲突可能 生, 因此阻塞 源以防止冲突。一个典型的例子是 取一行数据之前先将其 住, 保只有放置 的 程能 行数据 行修改。

并 控制

Elasticsearch 中使用的 方法假定冲突是不可能 生的, 并且不会阻塞正在 的操作。然而, 如果源数据在 写当中被修改, 更新将会失 。用程序接下来将决定 如何解决冲突。例如, 可以重 更新、使用新的数据、或者将相 情况 告 用 。

并 控制

Elasticsearch 是分布式的。当文 建、更新或 除 , 新版本的文 必 制到集群中的其他点。Elasticsearch 也是 和并 的, 意味着 些 制 求被并行 送, 并且到 目的地 也

序是乱的。 Elasticsearch 需要一 方法 保文 的旧版本不会覆 新的版本。

当我 之前 `index`, `GET` 和 `delete` 求 , 我 指出 个文 都有一个 `_version` (版本) 号, 当文 被修改 版本号 。 Elasticsearch 使用 个 `_version` 号来 保 更以正 序得到 行。如果旧版本的文 在新版本之后到 , 它可以被 忽略。

我 可以利用 `_version` 号来 保 用中相互冲突的 更不会 致数据 失。我 通 指定想要修改文 的 `version` 号来 到 个目的。如果 版本不是当前版本号, 我 的 求将会失 。

我 建一个新的博客文章 :

```
PUT /website/blog/1/_create
{
  "title": "My first blog entry",
  "text": "Just trying this out..."
}
```

体告 我 , 个新 建的文 `_version` 版本号是 1 。 在假 我 想 个文 :我 加 其数据到 web 表 中, 做一些修改, 然后保存新的版本。

首先我 索文 :

```
GET /website/blog/1
```

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..."
  }
}
```

在, 当我 通 重建文 的索引来保存修改, 我 指定 `version` 我 的修改会被 用的版本 :

```
PUT /website/blog/1?version=1 ①
{
  "title": "My first blog entry",
  "text": "Starting to get the hang of this..."
}
```

① 我 想 个在我 索引中的文 只有 在的 `_version` 1 , 本次更新才能成功。

此 求成功, 并且 体告 我 `_version` 已 到 2 :

```
{  
  "_index": "website",  
  "_type": "blog",  
  "_id": "1",  
  "_version": 2  
  "created": false  
}
```

然而, 如果我 重新 行相同的索引 求, 然指定 `version=1`, Elasticsearch 返回 409 Conflict HTTP , 和一个如下所示的 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  
}
```

告 我 在 Elasticsearch 中 个文 的当前 `_version` 号是 2 , 但我 指定的更新版本号 1 。

我 在 做取决于我 的 用需求。我 可以告 用 其他人已 修改了文 , 并且在再次保存之前 些修改内容。 或者, 在之前的商品 `stock_count` 景, 我 可以 取到最新的文 并 重新 用 些修改。

所有文 的更新或 除 API, 都可以接受 `version` 参数, 允 在代 中使用 并 控制, 是一 明智的做法。

通 外部系 使用版本控制

一个常 的 置是使用其它数据 作 主要的数据存 , 使用 Elasticsearch 做数据 索, 意味着主数据 的所有更改 生 都需要被 制到 Elasticsearch , 如果多个 程 一数据同 , 可能遇到 似于之前描述的并 。

如果 的主数据 已 有了版本号—或一个能作 版本号的字段 比如 `timestamp`—那 就可以在 Elasticsearch 中通 加 `version_type=external` 到 字符串的方式重用 些相同的版本号,

版本号必 是大于零的整数，且小于 $9.2E+18$ ——一个 Java 中 long 型的正。

外部版本号的 理方式和我 之前 的内部版本号的 理方式有些不同， Elasticsearch 不是 当前 `version` 和 求中指定的版本号是否相同，而是 当前 `_version` 是否 小于 指定的版本号。如果 求成功，外部的版本号作 文 的新 `_version` 行存。

外部版本号不 在索引和 除 求是可以指定，而且在 建新文 也可以指定。

例如，要 建一个新的具有外部版本号 5 的博客文章，我 可以按以下方法 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 中，我 能看到当前的 `_version` 版本号是 5：

```
{
  "_index": "website",
  "_type": "blog",
  "_id": "2",
  "_version": 5,
  "created": true
}
```

在我 更新 个文 ，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..."
}
```

求成功并将当前 `_version` 10：

```
{
  "_index": "website",
  "_type": "blog",
  "_id": "2",
  "_version": 10,
  "created": false
}
```

如果 要重新 行此 求 ，它将会失，并返回像我 之前看到的同 的冲突， 因 指定的外部版本号不大于 Elasticsearch 的当前版本号。

文 的部分更新

在 `更新整个文`，我 已 介 更新一个文 的方法是 索并修改它，然后重新索引整个文，的如此。然而，使用 `update` API 我 可以部分更新文，例如在某个 求 数器 行累加。

我 也介 文 是不可 的：他 不能被修改，只能被替。`update` API 必 遵循同 的。从外部来看，我 在一个文 的某个位置 行部分更新。然而在内部，`update` API 使用与之前描述相同的 索-修改-重建索引 的 理 程。区 在于 个 程 生在分片内部，就避免了多次 求的 。通 少 索和重建索引 之 的 ，我 也 少了其他 程的更 来冲突的可能性。

`update` 求最 的一 形式是接收文 的一部分作 `doc` 的参数，它只是与 有的文 行合并。象被合并到一起，覆 有的字段， 加新的字段。 例如，我 加字段 `tags` 和 `views` 到我的博客文章，如下所示：

```
POST /website/blog/1/_update
{
  "doc" : {
    "tags" : [ "testing" ],
    "views": 0
  }
}
```

如果 求成功，我 看到 似于 `index` 求的：

```
{
  "_index" : "website",
  "_id" : "1",
  "_type" : "blog",
  "_version" : 3
}
```

索文 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" ], ①  
        "views": 0 ①  
    }  
}
```

① 新的字段已被添加到 `_source` 中。

使用脚本部分更新文

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

```
POST /website/blog/1/_update  
{  
    "script" : "ctx._source.views+=1"  
}
```

用 Groovy 脚本 程

于那些 API 不能 足需求的情况，Elasticsearch 允 使 用脚本 写自定 的 。 多 API都支持脚本的使用，包括搜索、排序、聚合和文 更新。 脚本可以作 求的一部分被 ，从特殊的 .scripts 索引中 索，或者从磁 加 脚本。

的脚本 言 是 **Groovy**，一 快速表 的脚本 言，在 法上与 JavaScript 似。 它在 Elasticsearch V1.3.0 版本首次引入并 行在 沙 中，然而 Groovy 脚本引 存在漏洞，允 攻者通 建 Groovy 脚本，在 Elasticsearch Java VM 行 脱 沙 并 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 被 禁用。 此外， 可以通 置集群中的所有 点的 **config/elasticsearch.yml** 文件来禁用 Groovy 脚本：

```
script.groovy.sandbox.enabled: false
```

将 Groovy 沙 ，从而防止 Groovy 脚本作 求的一部分被接受， 或者从特殊的 .scripts 索引中被 索。当然， 然可以使用存 在 个 点的 **config/scripts/** 目 下的 Groovy 脚本。

如果 的架 和安全性不需要担心漏洞攻 ，例如 的 Elasticsearch 端 暴露和提供 可信 的 用，当它是 的 用需要的特性 ， 可以 重新 用 脚本。

可以在 [scripting reference documentation](#) 取更多 于脚本的 料。

我 也可以通 使用脚本 **tags** 数 添加一个新的 。在 个例子中，我 指定新的 作 参数，而不是硬 到脚本内部。 使得 Elasticsearch 可以重用 个脚本，而不是 次我 想添加 都要 新脚本重新 ！

```
POST /website/blog/1/_update
{
  "script" : "ctx._source.tags+=new_tag",
  "params" : {
    "new_tag" : "search"
  }
}
```

取文 并 示最后 次 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"], ①
    "views": 1 ②
  }
}
```

① `search` 已追加到 `tags` 数 中。

② `views` 字段已 。

我 甚至可以 通 置 `ctx.op delete` 来 除基于其内容的文 :

```
POST /website/blog/1/_update
{
  "script" : "ctx.op = ctx._source.views == count ? 'delete' : 'none'",
  "params" : {
    "count": 1
  }
}
```

更新的文 可能尚不存在

假 我 需要在 Elasticsearch 中存 一个 面 量 数器。 当有用 数器 行累加。但是，如果它是一个新 ， 我 不能 定 数器已 存在。 如果我 更新一个不存在的文 ， 那 更新操作将会失 。

在 的情况下，我 可以使用 `upsert` 参数，指定如果文 不存在就 先 建它：

```
POST /website/pageviews/1/_update
{
  "script" : "ctx._source.views+=1",
  "upsert": {
    "views": 1
  }
}
```

我 第一次 行 个 求 ， `upsert` 作 新文 被索引，初始化 `views` 字段 1 。 在后 的 行中，由于文 已 存在， `script` 更新操作将替代 `upsert` 行 用， `views` 数器 行累加。

更新和冲突

在本的介中，我明索和重建索引的隔越小，更冲突的机会越小。但是它并不能完全消除冲突的可能性。是有可能在 `update` 法重新索引之前，来自一程的求修改了文。

为了避免数据失，`update` API 在索得到文当前的 `version` 号，并版本号到 `_version` 的 `index` 求。如果一个程修改了于索和重新索引之的文，那 `_version` 号将不匹配，更新求将会失。

于部分更新的很多使用景，文已被改也没有系。例如，如果个程都面量数器行操作，它生的先后序其不太重要；如果冲突生了，我唯一需要做的就是再次更新。

可以通置参数 `retry_on_conflict` 来自完成，个参数定了失之前 `update` 重的次数，它的 0。

```
POST /website/pageviews/1/_update?retry_on_conflict=5 ①
{
  "script": "ctx._source.views+=1",
  "upsert": {
    "views": 0
  }
}
```

①失之前重更新5次。

在量操作无序的景，例如数器等个方法十分有效，但是在其他情况下更的序是非常重要的。似 `index API`，`update API` 采用最写入生效的方案，但它也接受一个 `version` 参数来分使用 `optimistic concurrency control` 指定想要更新文的版本。

取回多个文

Elasticsearch的速度已很快了，但甚至能更快。将多个求合并成一个，避免独理个求花的延和。如果需要从Elasticsearch索很多文，那使用 `multi-get` 或者 `mget` API 来将些索求放在一个求中，将比逐个文求更快地索到全部文。

`mget` API 要求有一个 `docs` 数作参数，个元素包含需要索文的元数据，包括 `_index`、`_type` 和 `_id`。如果想索一个或者多个特定的字段，那可以通 `_source` 参数来指定些字段的名字：

```

GET /_mget
{
  "docs" : [
    {
      "_index" : "website",
      "_type" : "blog",
      "_id" : 2
    },
    {
      "_index" : "website",
      "_type" : "pageviews",
      "_id" : 1,
      "_source": "views"
    }
  ]
}

```

体也包含一个 `docs` 数 , 于一个在 求中指定的文 , 个数 中都包含有一个 的 , 且 序与 求中的 序相同。 其中的 一个 都和使用 个 `get request` 求所得到的 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 索的数据都在相同的 `_index` 中 (甚至相同的 `_type` 中) , 可以在 URL 中指定 的 `/_index`

或者 的 `/_index/_type`。

然可以通 独 求覆 些 :

```
GET /website/blog/_mget
{
  "docs" : [
    { "_id" : 2 },
    { "_type" : "pageviews", "_id" : 1 }
  ]
}
```

事 上, 如果所有文 的 `_index` 和 `_type` 都是相同的, 可以只 一个 `ids` 数 , 而不是整个 `docs` 数 :

```
GET /website/blog/_mget
{
  "ids" : [ "2", "1" ]
}
```

注意, 我 求的第二个文 是不存在的。我 指定 型 `blog` , 但是文 ID 1 的 型是 `pageviews` , 一个不存在的情况将在 体中被 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 ①
    }
  ]
}
```

① 未 到 文 。

事 上第二个文 未能 到并不妨碍第一个文 被 索到。 个文 都是 独 索和 告的。

NOTE

即使有某个文 没有 到，上述 求的 HTTP 状 然是 200 。事 上，即使 求 没有 到任何文 ，它的状 依然 是 200 --因 mget 求本身已 成功 行。了 定某个文 是成功或者失 ， 需要 found 。

代 小的批量操作

与 mget 可以使我 一次取回多个文 同 的方式， bulk API 允 在 个 中 行多次 create 、 index 、 update 或 delete 求。如果 需要索引一个数据流比如日志事件，它可以排 和索引数百或数千批次。

bulk 与其他的 求体格式 有不同，如下所示：

```
{ action: { metadata } }\n{ request body } }\n{ action: { metadata } }\n{ request body } }\n...
```

格式 似一个有效的 行 JSON 文 流，它通 行符(\n) 接到一起。注意 个要点：

- 行一定要以 行符(\n) 尾，包括最后一行。 些 行符被用作一个 ，可以有效分隔行。
- 些行不能包含末 的 行符，因 他 将会 解析造成干 。 意味着 个 JSON 不能使用 pretty 参数打印。

TIP 在 什 是有趣的格式？ 中，我 解 什 bulk API 使用 格式。

action/metadata 行指定 一个文 做什 操作。

action 必 是以下 之一：

create

如果文 不存在，那 就 建它。 情 建新文 。

index

建一个新文 或者替 一个 有的文 。 情 索引文 和 更新整个文 。

update

部分更新一个文 。 情 文 的部分更新。

delete

除一个文 。 情 除文 。

metadata 指定被索引、 建、更新或者 除的文 的 _index 、 _type 和 _id 。

例如，一个 delete 求看起来是 的：

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

request body 行由文 的 _source 本身 成一文 包含的字段和 。它是 index 和 create

操作所必需的，是有道理的：必提供文以索引。

它也是 `update` 操作所必需的，并且包含 update API 的相同求体：`doc`、`upsert`、`script` 等等。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" }
```

了把所有的操作合在一起，一个完整的 `bulk` 求有以下形式：

```
POST /_bulk  
{ "delete": { "_index": "website", "_type": "blog", "_id": "123" }} ①  
{ "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"} } ②
```

① 注意 `delete` 作不能有求体, 它后面跟着的是外一个操作。

② 最后一个行符不要落下。

↑ Elasticsearch 包含 `items` 数，个数的内容是以求的序列出来的个求的结果。

```
{
  "took": 4,
  "errors": false, ①
  "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": "EiwfApScQiy7TIKFxRCTw",
        "_version": 1,
        "status": 201
      }},
    { "update": {
        "_index": "website",
        "_type": "blog",
        "_id": "123",
        "_version": 4,
        "status": 200
      }}
  ]
}
```

① 所有的子 求都成功完成。

个子 求都是独立 行，因此某个子 求的失 不会 其他子 求的成功与否造成影 。如果其中任何子 求失 ，最 的 error 志被 置 true，并且在相 的 求 告出 明：

```
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 中，我 看到 create 文 123 失 ，因 它已 存在。但是随后的 index 求，也是 文 123 操作，就成功了：

```
{
  "took": 3,
  "errors": true, ①
  "items": [
    { "create": {
        "_index": "website",
        "_type": "blog",
        "_id": "123",
        "status": 409, ②
        "error": "DocumentAlreadyExistsException ③
[[website][4] [blog][123]:
document already exists]"
      }},
    { "index": {
        "_index": "website",
        "_type": "blog",
        "_id": "123",
        "_version": 5,
        "status": 200 ④
      }}
  ]
}
```

① 一个或者多个 求失 。

② 个 求的HTTP状 告 409 CONFLICT。

③ 解 什 求失 的 信息。

④ 第二个 求成功, 返回 HTTP 状 200 OK。

也意味着 bulk 求不是原子的 : 不能用它来 事 控制。 个 求是 独 理的, 因此一个求的成功或失 不会影 其他的 求。

不要重 指定Index和Type

也 正在批量索引日志数据到相同的 index 和 type 中。 但 一个文 指定相同的元数据是一 浪 。相反, 可以像 mget API 一 , 在 bulk 求的 URL 中接收 的 /_index 或者 /_index/_type :

```
POST /website/_bulk
{ "index": { "_type": "log" } }
{ "event": "User logged in" }
```

然可以覆 元数据行中的 _index 和 _type ,但是它将使用 URL 中的 些元数据 作 :

```
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 求都需要由接收到 求的 点加 到内存中，因此 求越大，其他 求所能 得的内存就越少。 批量 求的大小有一个最佳 ，大于 个 ，性能将不再提升，甚至会下降。 但是最佳 不是一个固定的 。它完全取决于硬件、文 的大小和 度、索引和搜索的 的整体情况。

幸 的是，很容易 到 个 最佳点：通 批量索引典型文 ，并不断 加批量大小 行 。 当性能 始下降，那 的批量大小就太大了。一个好的 法是 始 将 1,000 到 5,000 个文 作 一个批次，如果 的文 非常大，那 就 少批量的文 个数。

密切 注 的批量 求的物理大小往往非常有用，一千个 1KB 的文 是完全不同于一千个 1MB 文 所占的物理大小。一个好的批量大小在 始 理后所占用的物理大小 5-15 MB。

分布式文 存

在前面的章 ，我 介 了如何索引和 数据，不 我 忽略了很多底 的技 ，例如文件是如何分布到集群的，又是如何从集群中 取的。 Elasticsearch 本意就是 藏 些底 ，我 好 注在 中，所以其 不必了解 深入也无妨。

在 个章 中，我 将深入探索 些核心的技 ，能 助 更好地理解数据如何被存 到 个分布式系 中。

注意

个章 包含了一些高 ，上面也提到 ，就算 不 住和理解所有的 然能正常使用 Elasticsearch。如果 有 趣的 ， 个章 可以作 的 外 趣 物， 展 的知 面。

如果 在 个章 的 时候感到很吃力，也不用担心。 个章 只是用来告 Elasticsearch 是如何工作的， 将来在工作中如果 需要用到 个章 提供的知 ，可以再回 来翻 。

路由一个文 到一个分片中

当索引一个文 的 时候，文 会被存 到一个主分片中。 Elasticsearch 如何知道一个文 存放到 个分片中 ？当我 建文 ，它如何决定 个文 当被存 在分片 1 是分片 2 中 ？

首先 肯定不会是随机的，否 将来要 取文 的 时候我 就不知道从何 了。 上， 个 程是根 据下面 个公式决定的：

```
shard = hash(routing) % number_of_primary_shards
```

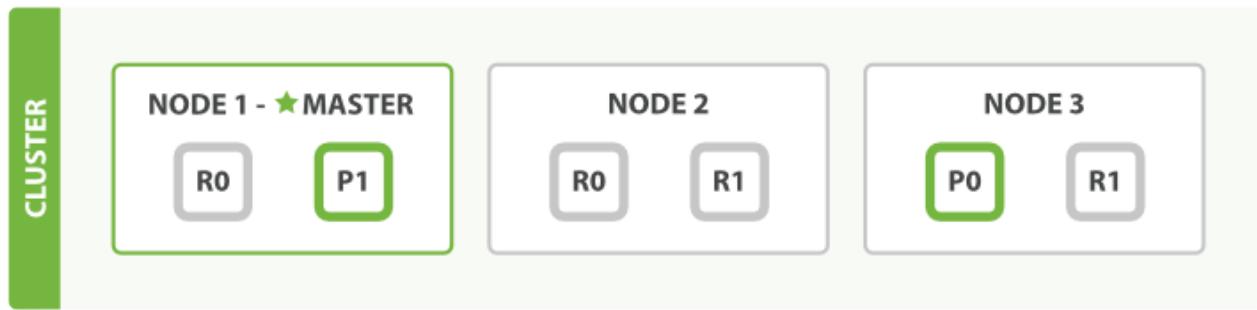
`routing` 是一个可选参数，是文档的 `_id`，也可以设置成一个自定义的。`routing` 通过 `hash` 函数生成一个数字，然后这个数字再除以 `number_of_primary_shards`（主分片的数量）后得到余数。一个分布在 0 到 `number_of_primary_shards-1` 之间的余数，就是我所求的文档所在分片的位置。

就解决了什么要在建索引的时候就定好主分片的数量，并且永远不会改变这个数量：因为如果数量变化了，那所有之前路由的都会无效，文档再也找不到了。

NOTE 可能得由于 Elasticsearch 主分片数量是固定的会使索引以行容。上当需要有很多技巧可以松容。我将会在《容》一章中提到更多有水平扩展的内容。

所有的 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保证所有相同的文档——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。我也会在《容》一章中讨论会有这一需求。

主分片和副本分片如何交互

为了说明目的，我假设有一个集群由三个节点组成。它包含一个叫 `blogs` 的索引，有一个主分片，一个主分片有两个副本分片。相同分片的副本不会放在同一点，所以我集群看起来像  有三个点和一个索引的集群。

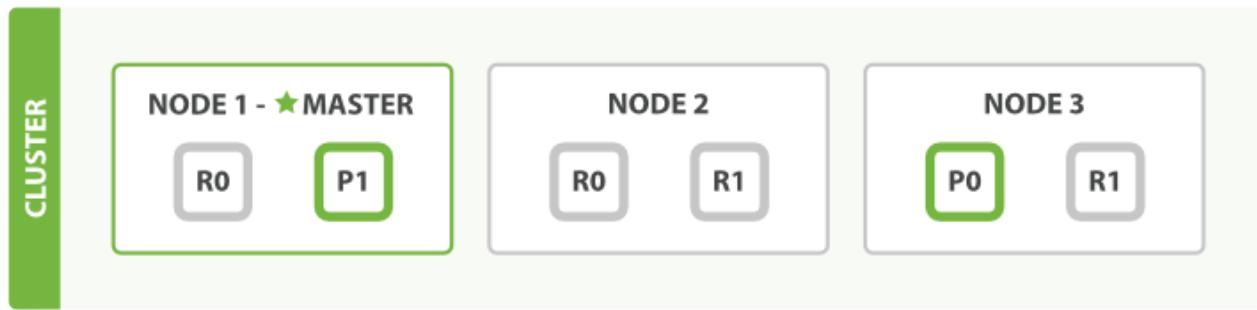


Figure 8. 有三个点和一个索引的集群

我可以发送请求到集群中的任一点。每个点都有能力处理任意请求。每个点都知道集群中任一文档的位置，所以可以直接将请求发送到需要的节点上。在下面的例子中，将所有的请求送到 `Node 1`，我将其称为协调点(*coordinating node*)。

TIP 当发送请求的时候，为了扩展，更好的做法是集群中所有的点。

新建、索引和删除文档

新建、索引和删除都是写操作，必须在主分片上面完成之后才能被复制到其他的副本分片，如下所示 [新建、索引和删除一个文档](#)。

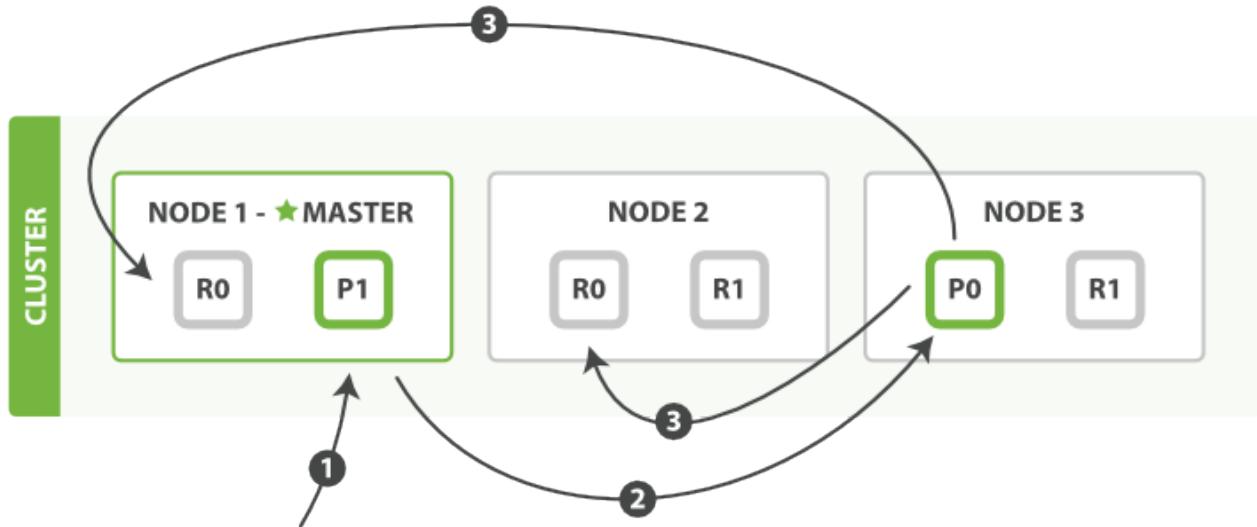


Figure 9. 新建、索引和 除 个文

以下是在主副分片和任何副本分片上面 成功新建，索引和 除文 所需要的 序：

1. 客 端向 Node 1 送新建、索引或者 除 求。
2. 点使用文 的 `_id` 定文 属于分片 0 。 求会被 到 Node 3, 因 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 行 求。如果成功了，它将 求并行 到 Node 1 和 Node 2 的副本分片上。一旦所有的副本分片都 告成功, Node 3 将向 点 告成功, 点向客 端 告成功。

在客 端收到成功 ，文 更已 在主分片和所有副本分片 行完成， 更是安全的。

有一些可 的 求参数允 影 个 程，可能以数据安全 代 提升性能。 些 很少使用，因 Elasticsearch已 很快，但是 了完整起，在 里 述如下：

`consistency`

`consistency`, 即一致性。在 置下，即使 是在 行一个`_write`操作之前，主分片都会要求 必 要有 定数量(`quorum`) (或者 法，也即必 要有大多数) 的分片副本 于活 可用状，才会去 行`_write`操作(其中分片副本可以是主分片或者副本分片)。 是 了避免在 生 分区故障 (`network partition`) 的 时候 行`_write`操作， 而 致数据不一致。`_quorum` 即：

```
int( (primary + number_of_replicas) / 2 ) + 1
```

`consistency` 参数的 可以 `one` (只要主分片状 `ok` 就允 行`_write`操作),`all` (必 要主分片和所有副本分片的状 没 才允 行`_write`操作)， 或 `quorum` 。 `quorum` ， 即大多数的分片副本状 没 就允 行`_write`操作。

注意， 定数量 的 算公式中 `number_of_replicas` 指的是在索引 置中的 定副本分片数，而不是指当前 理活 状 的副本分片数。如果 的索引 置中指定了当前索引 有三个副 分片，那 定数量的 算 果即：

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

如果此 只 一个 点，那 于活 状 的分片副本数量就 不到 定数量，也因此 将无法索引和 除任何文 。

timeout

如果没有足 的副本分片会 生什 ？ Elasticsearch会等待，希望更多的分片出 。

情况下，它最多等待1分 。 如果 需要， 可以使用 `timeout` 参数 使它更早 止： `100 100` 秒， `30s` 是30秒。

NOTE 新索引 有 1 个副本分片， 意味着 足 定数量 需要 个活 的分片副本。但是， 些 的 置会阻止我 在 一 点上做任何事情。 了避免 个 ， 要求只有当 `number_of_replicas` 大于1的 候， 定数量才会 行。

取回一个文

可以从主分片或者从其它任意副本分片 索文 ， 如下 所示 取回 个文 。

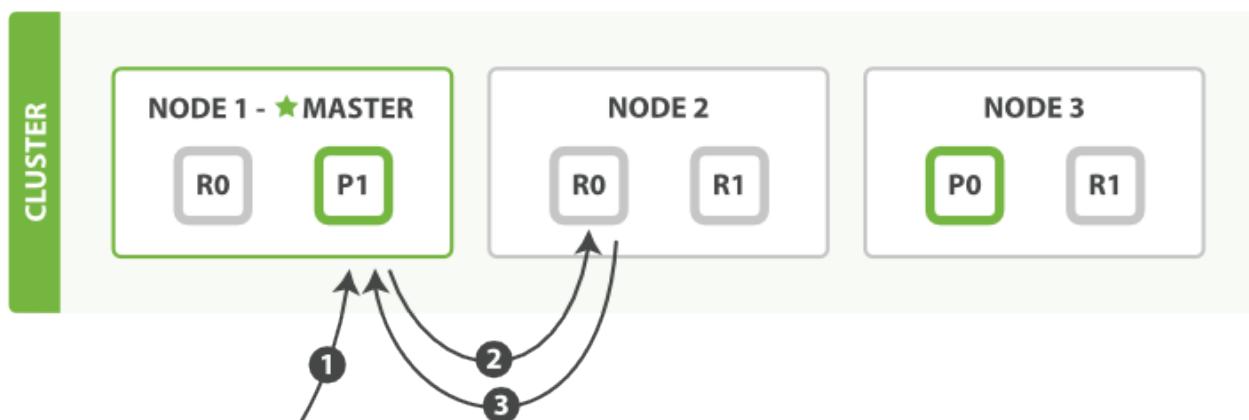


Figure 10. 取回 个文

以下是从主分片或者副本分片 索文 的 序：

- 1、客 端向 Node 1 送 取 求。
- 2、点使用文 的 `_id` 来 定文 属于分片 0 。分片 0 的副本分片存在于所有的三个 点上。 在 情况下，它将 求 到 Node 2。
- 3、Node 2 将文 返回 Node 1，然后将文 返回 客 端。

在 理 取 求 ， 点在 次 求的 候都会通 所有的副本分片来 到 均衡。

在文 被 索 ， 已 被索引的文 可能已 存在于主分片上但是 没有 制到副本分片。 在 情况下，副本分片可能会 告文 不存在，但是主分片可能成功返回文 。 一旦索引 求成功返回 用 ， 文 在主分片和副本分片都是可用的。

局部更新文

如 [局部更新文](#) 所示, `update` API 合了先前 明的 取和写入模式。

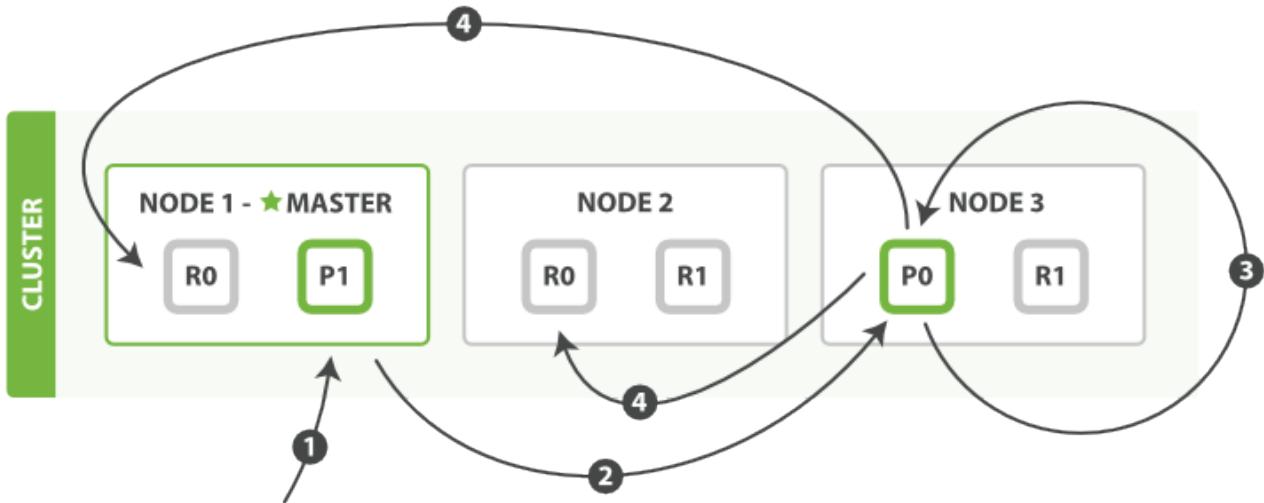


Figure 11. 局部更新文

以下是部分更新一个文 的：

1. 客 端向 `Node 1` 送更新 求。
2. 它将 求 到主分片所在的 `Node 3`。
3. `Node 3` 从主分片 索文 , 修改 `_source` 字段中的 JSON , 并且 重新索引主分片的文 。 如果文 已 被 一个 程修改, 它会重 3 , 超 `retry_on_conflict` 次后放 。
4. 如果 `Node 3` 成功地更新文 , 它将新版本的文 并行 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 点也返回成功, 点向客 端返回成功。

`update` API 接受在 [新建、索引和 除文](#) 章 中介 的 `routing` 、 `replication` 、 `consistency` 和 `timeout` 参数。

基于文 的 制

当主分片把更改 到副本分片 , 它不会 更新 求。 相反, 它 完整文 的新版本。

住, 些更改将会 到副本分片, 并且不能保 它 以 送它 相同的 序到 。

如果Elasticsearch 改变 求, 可能以 的 序 用更改, 致得到 坏的文 。

多文 模式

`mget` 和 `bulk` API 的模式 似于 文 模式。区 在于 点知道 个文 存在于 个分片中。 它将整个多文 求分解成 个分片的多文 求, 并且将 些 求并行 到 个参与 点。

点一旦收到来自 个 点的 答, 就将 个 点的 收集整理成 个 , 返回 客 端, 如 [使用 `mget` 取回多个文](#) 所示。

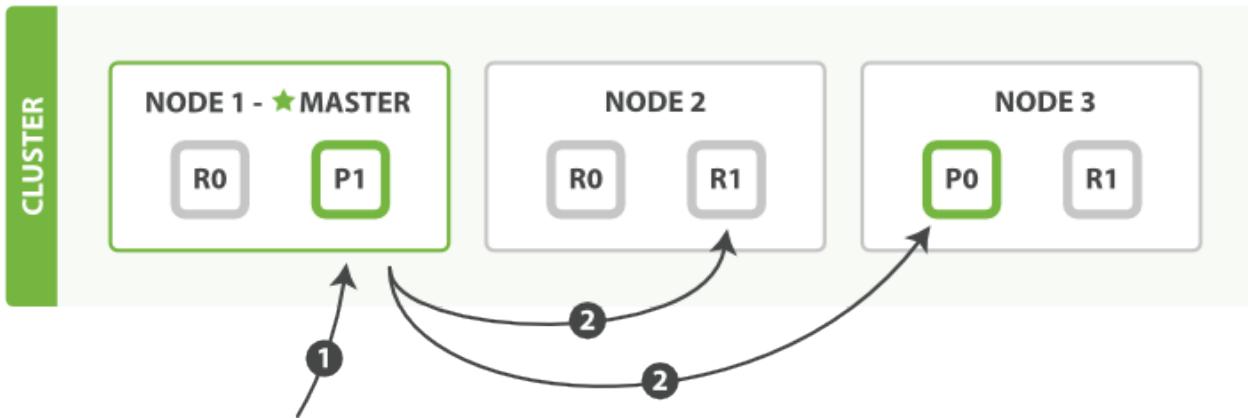


Figure 12. 使用 `mget` 取回多个文

以下是使用一个 `mget` 求取回多个文 所需的 序：

1. 客 端向 Node 1 送 `mget` 求。
2. Node 1 个分片 建多文 取 求, 然后并行 些 求到托管在 个所需的主分片或者副本分片的 点上。一旦收到所有答 , Node 1 建 并将其返回 客 端。

可以 `docs` 数 中 个文 置 `routing` 参数。

bulk API, 如 使用 `bulk` 修改多个文 所示, 允 在 个批量 求中 行多个 建、索引、 除和更新 求。

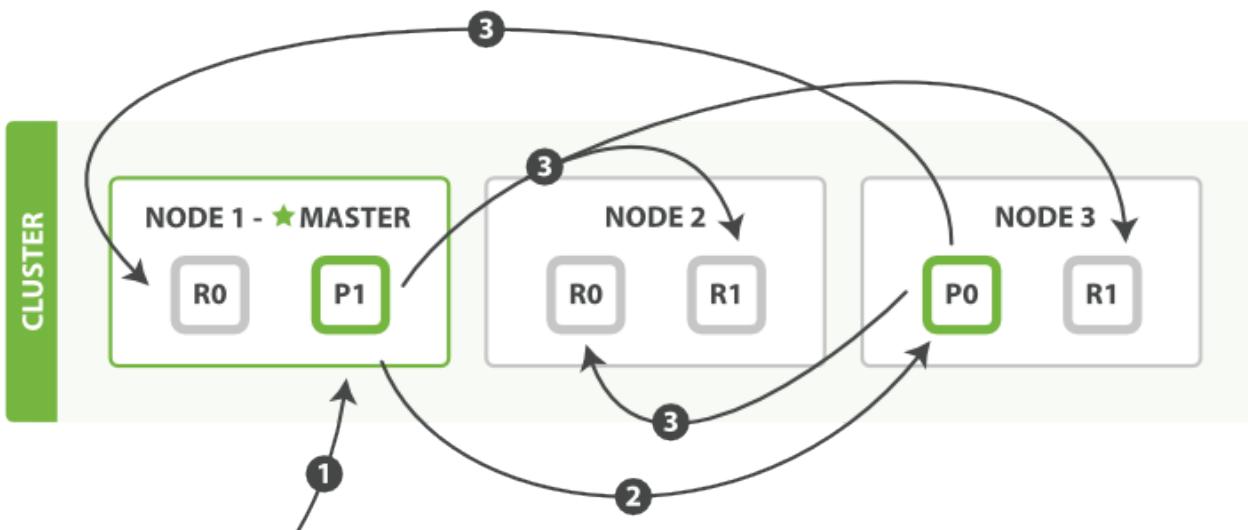


Figure 13. 使用 `bulk` 修改多个文

`bulk` API 按如下 序 行：

1. 客 端向 Node 1 送 `bulk` 求。
2. Node 1 个 点 建一个批量 求, 并将 些 求并行 到 个包含主分片的 点主机。
3. 主分片一个接一个按 序 行 个操作。当 个操作成功 , 主分片并行 新文 (或 除) 到副本 分片, 然后 行下一个操作。 一旦所有的副本分片 告所有操作成功, 点将向 点 告成功, 点将 些 收集整理并返回 客 端。

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

什么是有趣的格式？

当我早些时候在[代小的批量操作](#)章了解批量请求，可能会自己，“什么 `bulk` API 需要有行符的有趣格式，而不是送包装在 JSON 数字中的请求，例如 `mget` API？”。

了回答一点，我需要解一点背景：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集群中的任何点。这意味着批量请求 `bulk` 中的每个操作都需要被送到正点上的正确分片。

如果一个请求被包装在 JSON 数字中，那就意味着我需要执行以下操作：

- 将 JSON 解析为数字（包括文本数据，可以非常大）
- 看一个请求以决定去哪个分片
- 在一个分片建一个请求数
- 将这些数序列化为内部格式
- 将请求送到一个分片

是可行的，但需要大量的 RAM 来存储原本相同的数据的副本，并将构建更多的数据，Java 虚拟机（JVM）将不得不花时间回收。

相反，Elasticsearch 可以直接取被缓冲区接收的原始数据。它使用行符字符来解析小的 `action/metadata` 行来决定一个分片处理一个请求。

些原始请求会被直接送到正确的分片。没有冗余的数据存储，没有浪费的数据。整个请求尽可能在最小的内存中处理。

搜索——最基本的工具

在，我已学会了如何使用 Elasticsearch 作为一个 NoSQL 格的分布式文件存储系统。我可以将一个 JSON 文档到 Elasticsearch 里，然后根据 ID 索引。但 Elasticsearch 真正强大之处在于可以从无规律的数据中找出有意的信息——从“大数据”到“大信息”。

Elasticsearch 不只会存储（stores）文档，了能被搜索到也会为文档添加索引（indexes），也是什么我使用结构化的 JSON 文档，而不是无结构的二进制数据。

文档中的所有字段都将被索引并且可以被查询。不仅如此，在所有（all）一些索引字段，以人的速度返回结果。是永远不会考虑用数据去做的一些事情。

搜索（search）可以做到：

- 在类似于 `gender` 或者 `age` 的字段上使用规范化，`join_date` 的字段上使用排序，就像 SQL 的规范化一样。
- 全文索引，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 箱即用的， 了充分 挖 Elasticsearch 的潜力， 需要理解以下三个概念：

映射 (*Mapping*)

描述数据在 个字段内如何存

分析 (*Analysis*)

全文是如何 理使之可以被搜索的

域特定 言 (*Query DSL*)

Elasticsearch 中 大 活的 言

以上提到的 个点都是一個大 ， 我 将在 [深入搜索](#) 一章 述它 。本章 我 将介 三点的一些基本概念—— 助 大致了解搜索是如何工作的。

我 将使用最 的形式 始介 `search` API。

数据

本章 的 数据可以在 里 到：<https://gist.github.com/clintongormley/8579281>。

可以把 些命令 制到 端中 行来 践本章的例子。

外，如果 的是在 版本，可以 点 个 接 感受下。

空搜索

搜索API的最基 的形式是没有指定任何 的空搜索，它 地返回集群中所有索引下的所有文 ：

```
GET /_search
```

返回的 果（ 了界面 的）像 ：

```
{
  "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 果中最重要的部分是 `hits`，它包含 `total` 字段来表示匹配到的文 数，并且一个 `hits` 数 包含所 果的前十个文 。

在 `hits` 数 中 个 果包含文 的 `_index`、`_type`、`_id`，加上 `_source` 字段。 意味着我 可以直接从返回的搜索 果中使用整个文 。 不像其他的搜索引 ， 返回文 的ID，需要 独去 取文 。

个 果 有一个 `_score`， 它衡量了文 与 的匹配程度。 情况下，首先返回最相 的文 果，就是 ， 返回的文 是按照 `_score` 降序排列的。在 个例子中，我 没有指定任何 ， 故所有的文 具有相同的相 性，因此 所有的 果而言 1是中性的 `_score`。

`max_score` 是与 所匹配文 的 `_score` 的最大 。

took

`took` 告 我 行整个搜索 求耗 了多少 秒。

shards

`_shards` 部分告我 在 中参与分片的 数，以及 些分片成功了多少个失了多少个。正常情况下我 不希望分片失 ，但是分片失 是可能 生的。如果我 遭遇到一 的故障，在 个故障中 失了相同分片的原始数据和副本，那 个分片将没有可用副本 来 搜索 求作出 。假若 ， Elasticsearch 将 告 个分片是失 的，但是会 返回剩余分片的 果。

timeout

`timed_out` 告我 是否超 。 情况下，搜索 求不会超 。如果低 比完成 果更重要， 可以指定 `timeout` 10 或者 10ms (10 秒) ，或者 1s (1秒) :

```
GET /_search?timeout=10ms
```

在 求超 之前，Elasticsearch 将会返回已 成功从 个分片 取的 果。

WARNING

当注意的是 `timeout` 不是停止 行 ，它 是告知正在 的点返回到目前 止收集的 果并且 接。在后台，其他的分片可能 在 行即使是 果已 被 送了。

使用超 是因 SLA(服 等) 是很重要的，而不是因 想去中止行的 。

多索引，多 型

有没有注意到之前的 `empty search` 的 果，不同 型的文 — `<code>user</code>` 和 `<code>tweet</code>` 来自不同的索引— `<code>us</code>` 和 `<code>gb</code>` ?

如果不 某一特殊的索引或者 型做限制，就会搜索集群中的所有文 。 Elasticsearch 搜索 求到一个主分片或者副本分片， 集 出的前10个 果，并且返回 我 。

然而， 常的情况下， 想在一个或多个特殊的索引并且在一个或者多个特殊的 型中 行搜索。我 可以通 在URL中指定特殊的索引和 型 到 效果，如下所示：

`/_search`

在所有的索引中搜索所有的 型

`/gb/_search`

在 `gb` 索引中搜索所有的 型

`/gb,us/_search`

在 `gb` 和 `us` 索引中搜索所有的文

`/g*,u*/_search`

在任何以 `g` 或者 `u` 的索引中搜索所有的 型

`/gb/user/_search`

在 `gb` 索引中搜索 `user` 型

/gb,us/user,tweet/_search

在 gb 和 us 索引中搜索 user 和 tweet 型

/_all/user,tweet/_search

在所有的索引中搜索 user 和 tweet 型

当在一的索引下 行搜索的 候, Elasticsearch

求到索引的

个分片中, 可以是主分片也可以是副本分片, 然后从 个分片中收集 果。多索引搜索恰好也是用相同的方式工作的一只是会 及到更多的分片。

TIP 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 来所 是等 的。

接下来, 将明白 的方式如何 活的根据需求的 化 容 得 。

分

在之前的 空搜索 中 明了集群中有 14 个文 匹配了 (empty) query 。但是在 hits 数 中只有 10 个文 。如何才能看到其他的文 ?

和 SQL 使用 LIMIT 字返回 个 page 果的方法相同, Elasticsearch 接受 from 和 size 参数 :

size

示 返回的 果数量, 是 10

from

示 跳 的初始 果数量, 是 0

如果 展示 5 条 果, 可以用下面方式 求得到 1 到 3 的 果:

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

考 到分 深以及一次 求太多 果的情况, 果集在返回之前先 行排序。 但 住一个 求

常跨越多个分片, 个分片都 生自己的排序 果, 些 果需要 行集中排序以保 整体 序是正 的

。

在分布式系 中深度分

理解 什 深度分 是有 的，我 可以假 在一个有 5 个主分片的索引中搜索。 当我 求 果的第一 果从 1 到 10)， 一个分片 生前 10 的 果，并且返回 点， 点 50 个 果排序得到全部 果的前 10 个。

在假 我 求第 1000 一 果从 10001 到 10010 。所有都以相同的方式工作除了 个分片不得不 生前 10010 个 果以外。 然后 点 全部 50050 个 果排序最后 掉 些 果中的 50040 个 果。

可以看到，在分布式系 中， 果排序的成本随分 的深度成指数上升。 就是 web 搜索引 任何 都不要返回超 1000 个 果的原因。

TIP 在 [重新索引 的数据](#) 中解 了如何能 有效 取大量的文 。

量 搜索

有 形式的 **搜索 API**：一 是 `` 量的'' 字符串 版本，要求在 字符串中 所有的参数， 一 是更完整的 求体 版本，要求使用 JSON 格式和更 富的 表 式作 搜索 言。

字符串搜索非常 用于通 命令行做即席 。例如，在 **tweet** 型中 **tweet** 字段包含 **elasticsearch** 的所有文 ：

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

下一个 在 **name** 字段中包含 **john** 并且在 **tweet** 字段中包含 **mary** 的文 。 的 就是

```
+name:john +tweet:mary
```

但是 字符串参数所需要的 百分比 (者注：URL) 上更加 :

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

+ 前 表示必 与 条件匹配。 似地， **-** 前 表示一定不与 条件匹配。没有 **+** 或者 **-** 的所有其他条件都是可 的——匹配的越多，文 就越相 。

_all 字段

个 搜索返回包含 **mary** 的所有文 :

```
GET /_search?q=mary
```

之前的例子中，我 在 **tweet** 和 **name** 字段中搜索内容。然而， 个 的 果在三个地方提到了 **mary** :

- 有一个用 叫做 Mary
- 6条微博 自 Mary
- 一条微博直接 @mary

Elasticsearch 是如何在三个不同的字段中 到 果的 ？

当索引一个文 的 候， Elasticsearch 取出所有字段的 接成一个大的字符串，作 `_all` 字段 行索引。例如，当索引 个文 ：

```
{
  "tweet": "However did I manage before Elasticsearch?",
  "date": "2014-09-14",
  "name": "Mary Jones",
  "user_id": 1
}
```

就好似 加了一个名叫 `_all` 的 外字段：

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

除非 置特定字段，否 字符串就使用 `_all` 字段 行搜索。

TIP 在 始 一个 用 `_all` 字段是一个很 用的特性。之后，会 如果搜索 用指定字段来代替 `_all` 字段，将会更好控制搜索 果。当 `_all` 字段不再有用的时候，可以将它置 失效，正如在 元数据: `_all` 字段 中所解 的。

更 的

下面的 tweents 型，并使用以下的条件：

- `name` 字段中包含 `mary` 或者 `john`
- `date` 大于 `2014-09-10`
- `all` 字段包含 `aggregations` 或者 `geo`

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

字符串在做了 当的 后，可 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， 量 的 字符串搜索效果 是挺 人 喜的。 它的 法在相 参考文 中有 解 ，以便 的表 很 的 。 于通 命令做一次性 ，或者是在 段 ，都非常方便。

但同 也可以看到， 精 更加晦 和困 。而且很脆弱，一些 字符串中很小的 法 ，像 -，:，/ 或者 " 不匹配等，将会返回 而不是搜索 果。

最后， 字符串搜索允 任何用 在索引的任意字段上 行可能 慢且重量 的 ， 可能会暴露 私信息，甚至将集群 。

TIP

因 些原因，不推 直接向用 暴露 字符串搜索功能，除非 于集群和数据来 非常信任他 。

相反，我 常在生 境中更多地使用功能全面的 *request body* API，除了能完成以上所有功能，有一些附加功能。但在到 那个 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 弄索引里面的数据 ，我 一些奇怪的事情。一些事情看起来被打乱了：在我 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 命中的 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 !
```

什 在 `_all` 字段 日期返回所有推文，而在 `date` 字段只 年 却没有返回 果？ 什 我 在 `_all` 字段和 `date` 字段的 果有差 ？

推 起来， 是因 数据在 `all` 字段与 `date` 字段的索引方式不同。所以，通 求 `gb` 索引中 `tweet` 型的`_映射` (或模式定)， 我 看一看 Elasticsearch 是如何解 我 文 的：

```
GET /gb/_mapping/tweet
```

将得到如下 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 `date` 字段 型的 ， Elasticsearch 我 生了一个映射。 个 告 我 `date` 字段被 是 `date` 型的。由于 `_all` 是 字段，所以没有提及它。但是我 知道 `_all` 字段是 `string` 型的。

所以 `date` 字段和 `string` 字段索引方式不同，因此搜索 果也不一 。 完全不令人吃 。 可能会 核心数据 型 `strings`、`numbers`、`Booleans` 和 `dates` 的索引方式有 不同。没 ， 他 有不同。

但是，到目前 止，最大的差 在于代表 精 （它包括 `string` 字段）的字段和代表 全文 的字段。 个区 非常重要——它将搜索引擎 和所有其他数据 区 来。

精 VS 全文

Elasticsearch 中的数据可以概括的分 : 精 和全文。

精 如它 听起来那 精 。例如日期或者用 ID，但字符串也可以表示精 ， 例如用 名或 箱地址。 于精 来 ， `Foo` 和 `foo` 是不同的， `2014` 和 `2014-09-15` 也是不同的。

一方面，全文 是指文本数据（通常以人 容易 的 言 写）， 例如一个推文的内容或一封 件的内容。

全文通常是指非自然语言的，是致的数据，但里有一个解：自然言是高度化的。在于自然言的是的，致算机以正解析。例如，考一条句：

NOTE

May is fun but June bores me.

它指的是月是人？

精很容易。果是二制的：要匹配，要不匹配。很容易用SQL表示：

```
WHERE name = "John Smith"  
AND user_id = 2  
AND date > "2014-09-15"
```

全文数据要微妙的多。我 的不只是“个文匹配”，而是“文匹配”的程度有多大？”句，文与定的相性如何？

我很少全文型的域做精匹配。相反，我希望在文本型的域中搜索。不如此，我希望搜索能理解我的意：

- 搜索 `UK`，会返回包含 `United Kindom` 的文。
- 搜索 `jump`，会匹配 `jumped`, `jumps`, `jumping`，甚至是 `leap`。
- 搜索 `johnny walker` 会匹配 `Johnnie Walker`, `johnnie depp` 匹配 `Johnny Depp`。
- `fox news hunting` 返回福克斯新（Fox News）中于狩的故事，同，`fox hunting news` 返回于狐的故事。

了促在全文域中的，Elasticsearch首先分析文，之后根据果建倒排索引。在接下来的，我会倒排索引和分析程。

倒排索引

Elasticsearch 使用一称倒排索引的，它用于快速的全文搜索。一个倒排索引由文中所有不重的列表成，于其中个，有一个包含它的文列表。

例如，假我有个文，个文的 `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

了建倒排索引，我首先将个文的 `content` 域拆分成独的（我称它条或 `tokens`），建一个包含所有不重条的排序列表，然后列出个条出在个文。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	

在，如果我 想搜索 **quick brown**，我 只需要 包含 个 条的文：

Term	Doc_1	Doc_2
brown	X	X
quick	X	
Total	2	1

个文 都匹配，但是第一个文 比第二个匹配度更高。如果我 使用 算匹配 条数量的相似性算法，那，我 可以，于我 的相 性来，第一个文 比第二个文 更佳。

但是，我 目前的倒排索引有一些：

- **Quick** 和 **quick** 以独立的 条出，然而用 可能 它 是相同的。
- **fox** 和 **foxes** 非常相似，就像 **dog** 和 **dogs**；他 有相同的 根。
- **jumped** 和 **leap**，尽管没有相同的 根，但他 的意思很相近。他 是同。

使用前面的索引搜索 **+Quick +fox** 不会得到任何匹配文。（住，**+** 前 表明 个 必存在。）只有同 出 **Quick** 和 **fox** 的文 才 足 个 条件，但是第一个文 包含 **quick fox**，第二个文 包含 **Quick foxes**。

我 的用 可以合理的期望 个文 与 匹配。我 可以做的更好。

如果我 将 条 准模式，那 我 可以 到与用 搜索的 条不完全一致，但具有足 相 性的文。例如：

- **Quick** 可以小写化 **quick**。

- `foxes` 可以 干提取 -- 根的格式-- `fox`。 似的, `dogs` 可以 提取 `dog`。
- `jumped` 和 `leap` 是同 , 可以索引 相同的 `jump`。

在索引看上去像 :

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

不 。我 搜索 `+Quick +fox` 然 会失 , 因 在我 的索引中, 已 没有 `Quick` 了。但是, 如果我 搜索的字符串使用与 `content` 域相同的 准化 , 会 成 `+quick +fox` , 个文 都会匹配 !

NOTE 非常重要。 只能搜索在索引中出 的 条, 所以索引文本和 字符串必 准化 相 同的格式。

分 和 准化的 程称 分析 , 我 会在下个章 。

分析与分析器

分析包含下面的 程 :

- 首先, 将一 文本分成 合于倒排索引的独立的 条 ,
- 之后, 将 些 条 一化 准格式以提高它 的“可搜索性”, 或者 *recall*

分析器 行上面的工作。 分析器 上是将三个功能封装到了一个包里 :

字符 器

首先, 字符串按 序通 个 字符 器 。他 的任 是在分 前整理字符串。一个字符 器可以用来去掉HTML, 或者将 `&` 化成 `and`。

分 器

其次, 字符串被 分 器 分 个的 条。一个 的分 器遇到空格和 点的 时候, 可能会将文本拆分成 条。

Token 器

最后, 条按 序通 个 *token* 器 。 个 程可能会改 条 (例如, 小写化 `Quick`), 除 条 (例如, 像 `a`, `and`, `the` 等无用), 或者 加 条 (例如, 像 `jump` 和 `leap` 同) 。

Elasticsearch 提供了 箱即用的字符 器、分 器和 token 器。 些可以 合起来形成自定 的分析器以用于不同的目的。我 会在 [自定 分析器 章](#)。

内置分析器

但是， Elasticsearch 附 了可以直接使用的 包装的分析器。接下来我 会列出最重要的分析器。了 明它 的差 ，我 看看 个分析器会从下面的字符串得到 些 条：

```
"Set the shape to semi-transparent by calling set_trans(5)"
```

准分析器

准分析器是Elasticsearch 使用的分析器。它是分析各 言文本最常用的 。它根据 [Unicode 盟定 的 界 分文本](#)。除 大部分 点。最后，将 条小写。它会 生

```
set, the, shape, to, semi, transparent, by, calling, set_trans, 5
```

分析器

分析器在任何不是字母的地方分隔文本，将 条小写。它会 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 分文本。它会 生

```
Set, the, shape, to, semi-transparent, by, calling, set_trans(5)
```

言分析器

特定 言分析器可用于 很多 言。它 可以考 指定 言的特点。例如， 英 分析器附 了一 英 无用 （常用 ，例如 **and** 或者 **the** ，它 相 性没有多少影 ），它 会被 除。由于理解英 法的 ， 个分 器可以提取英 的 干。

英 分 器会 生下面的 条：

```
set, shape, semi, transpar, call, set_tran, 5
```

注意看 **transparent**、**calling** 和 **set_trans** 已 根格式。

什 时候使用分析器

当我 索引 一个文 ，它的全文域被分析成 条以用来 建倒排索引。但是，当我 在全文域 搜索 的 时候，我 需要将 字符串通 相同的分析 程 ，以保 我 搜索的 条格式与索引中的 条格式一致。

全文 ，理解 个域是如何定 的，因此它 可以做正 的事：

- 当一个全文域，会将字符串用相同的分析器，以生成正确的搜索结果。
- 当一个精确域，不会分析字符串，而是搜索指定的精确值。

在可以理解在 [始章](#) 的什么返回那的结果：

- `date` 域包含一个精确值：唯一的条 `2014-09-15`。
- `_all` 域是一个全文域，所以分析将日期化为三个条：`2014`, `09`, 和 `15`。

当我 在 `_all` 域 `2014`，它匹配所有的12条推文，因为它都含有 `2014`：

```
GET /_search?q=2014 # 12 results
```

当我 在 `all` 域 `2014-09-15`，它首先分析字符串，生成匹配 `2014`, `09`, 或 `15` 中任意一条的值。也会匹配所有12条推文，因为它都含有 `2014`：

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

当我 在 `date` 域 `2014-09-15`，它精确日期，只找到一个推文：

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

当我 在 `date` 域 `2014`，它找不到任何文档，因为没有文档含有一个精确日志：

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

分析器

有时候很难理解分析的过程和被存储到索引中的条，特别是接触Elasticsearch。为了理解发生了什么，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"
}
```

结果中一个元素代表一个唯一的条：

```
{
  "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` 是存到索引中的条。`position` 指明条在原始文本中出的位置。`start_offset` 和 `end_offset` 指明字符在原始字符串中的位置。

TIP 一个分析器的 `type` 都不一，可以忽略它。它在Elasticsearch 中的唯一作用在于<https://www.elastic.co/guide/en/elasticsearch/reference/master/analyzers-keep-types-tokenfilter.html>[`keep_types` token 器]。

`analyze` API 是一个有用的工具，它有助于我理解Elasticsearch索引内部发生了什么，随着深入，我会一一它。

指定分析器

当Elasticsearch在的文 中到一个新的字符串域，它会自 置其一个全文字符串域，使用 准分析器 它 行分析。

不希望 是 。可能 想使用一个不同的分析器， 用于 的数据使用的 言。有 时候 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 入的精 ，例如用 ID或者一个内部的状 域或 。

要做到 一点，我 必 手 指定 些域的映射。

映射

了能 将 域 , 数字域 数字, 字符串域 全文或精 字符串, Elasticsearch 需要知道 个域中数据的 型。 个信息包含在映射中。

如 [数据](#) [入和](#) [出](#) 中解 的, 索引中 个文 都有 型。 型都有它自己的 映射 , 或者 模式定 。映射定 了 型中的域, 个域的数据 型, 以及Elasticsearch如何 理 些域。映射也用于配置与 型 有 的元数据。

我 会在 [型和映射](#) 映射。本 , 我 只 足 入 的内容。

核心 域 型

Elasticsearch 支持如下 域 型 :

字符串: `string`

整数: `byte`, `short`, `integer`, `long`

浮点数: `float`, `double`

布 型: `boolean`

日期: `date`

当 索引一个包含新域的文 —之前未曾出 -- Elasticsearch 会使用 映射 , 通 JSON中基本数据 型, 域 型, 使用如下 :

JSON type

域 type

布 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`

NOTE 意味着如果 通 引号(`"123"`)索引一个数字, 它会被映射 `string` 型, 而不是 `long` 。但是, 如果 个域已 映射 `long` , 那 Elasticsearch 会 将 个字符串 化 `long` , 如果无法 化, 会 出一个 常。

看映射

通过 `/_mapping`，我可以看 Elasticsearch 在一个或多个索引中的一个或多个 型的映射。在 始章，我已取得索引 `gb` 中 型 `tweet` 的映射：

```
GET /gb/_mapping/tweet
```

Elasticsearch 根据我 索引的文 ， 域(称 属性) 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"
          }
        }
      }
    }
  }
}
```

TIP 的映射，例如 将 `age` 域映射 `string` 型，而不是 `integer`，会致出令人困惑的果。

一下！而不是假 的映射是正 的。

自定 域映射

尽管在很多情况下基本域数据 型已 用，但 常需要 独域自定 映射，特 是字符串域。自定映射允 行下面的操作：

- 全文字符串域和精 字符串域的区
- 使用特定 言分析器
- 化域以 部分匹配

- 指定自定 数据格式
- 有更多

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

```
{
  "number_of_clicks": {
    "type": "integer"
  }
}
```

，`string` 型域会被 包含全文。就是 ，它 的 在索引前，会通 一个分析器，于 个域的 在搜索前也会 一个分析器。

`string` 域映射的 个最重要属性是 `index` 和 `analyzer`。

index

`index` 属性控制 索引字符串。它可以是下面三个：

analyzed

首先分析字符串，然后索引它。句 ，以全文索引 个域。

not_analyzed

索引 个域，所以它能 被搜索，但索引的是精 。不会 它 行分析。

no

不索引 个域。 个域不会被搜索到。

`string` 域 `index` 属性 是 `analyzed`。如果我 想映射 个字段 一个精 ，我 需要 置它 `not_analyzed`：

```
{
  "tag": {
    "type": "string",
    "index": "not_analyzed"
  }
}
```

NOTE

其他 型（例如 `long`, `double`, `date` 等）也接受 `index` 参数，但有意 的 只有 `no` 和 `not_analyzed`，因 它 永 不会被分析。

analyzer

于 `analyzed` 字符串域，用 `analyzer` 属性指定在搜索和索引 使用的分析器。 ， Elasticsearch 使用 `standard` 分析器，但 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 分析器](#)，我 会展示 定 和使用自定 分析器。

更新映射

当 首次 建一个索引的 候，可以指定 型的映射。 也可以使用 `/_mapping` 新 型（或者 存在的 型更新映射） 加映射。

NOTE 尽管 可以 加 一个存在的映射， 不能 修改 存在的域映射。如果一个域的映射已 存在，那 域的数据可能已 被索引。如果 意 修改 个域的映射，索引的数据可能 会出 ， 不能被正常的搜索。

我 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 `not_analyzed`。

了描述指定映射的 方式，我 先 除 `gd` 索引：

```
DELETE /gb
```

然后 建一个新索引，指定 `tweet` 域使用 `english` 分析器：

```
PUT /gb ①
{
  "mappings": {
    "tweet" : {
      "properties" : {
        "tweet" : {
          "type" : "string",
          "analyzer": "english"
        },
        "date" : {
          "type" : "date"
        },
        "name" : {
          "type" : "string"
        },
        "user_id" : {
          "type" : "long"
        }
      }
    }
  }
}
```

① 通 消息体中指定的 `mappings` 建了索引。

后，我 决定在 `tweet` 映射 加一个新的名 `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 不需要再次列出所有已存在的域，因 无 如何我 都无法改 它 。新域已 被合并到存在的映射中。

映射

可以使用 `analyze` API 字符串域的映射。比 下面 个 求的 出：

```
GET /gb/_analyze
{
  "field": "tweet",
  "text": "Black-cats" ①
}

GET /gb/_analyze
{
  "field": "tag",
  "text": "Black-cats" ①
}
```

① 消息体里面 我 想要分析的文本。

tweet 域 生 个 条 black 和 cat , tag 域 生 独的 条 Black-cats 。 句 , 我 的映射正常工作。

核心域 型

除了我 提到的 量数据 型, JSON 有 null , 数 , 和 象, 些 Elasticsearch 都是支持的。

多 域

很有可能, 我 希望 tag 域包含多个 。我 可以以数 的形式索引 :

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

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

暗示 数 中所有的 必 是相同数据 型的 。 不能将日期和字符串混在一起。如果 通 索引数 来 建新的域, Elasticsearch 会用数 中第一个 的数据 型作 一个域的 型。

当 从 Elasticsearch 得到一个文 , 个数 的 序和 当初索引文 一 。 得到的 _source 域, 包含与 索引的一模一 的 JSON 文 。

NOTE

但是, 数 是以多 域 索引的—可以搜索, 但是无序的。 在搜索的 时候, 不能指定 “第一个” 或者 “最后一个”。 更 切的 , 把数 想象成 装在袋子里的 。

空域

当然, 数 可以 空。 相当于存在零 。 事 上, 在 Lucene 中是不能存 null 的, 所以我 存在 null 的域 空域。

下面三 域被 是空的, 它 将不会被索引 :

```
"null_value": null,  
"empty_array": [],  
"array_with_null_value": [ null ]
```

多 象

我 的最后一个 JSON 原生数据 是 象 -- 在其他 言中称 哈希, 哈希 map, 字典或者 数 。

内部 象 常用于嵌入一个 体或 象到其它 象中。例如, 与其在 tweet 文 中包含 user_name 和 user_id 域, 我 也可以 写 :

```
{  
    "tweet": "Elasticsearch is very flexible",  
    "user": {  
        "id": "@johnsmith",  
        "gender": "male",  
        "age": 26,  
        "name": {  
            "full": "John Smith",  
            "first": "John",  
            "last": "Smith"  
        }  
    }  
}
```

内部 象的映射

Elasticsearch 会 新的 象域并映射它 象 , 在 properties 属性下列出内部域 :

```
{
  "gb": {
    "tweet": { ①
      "properties": {
        "tweet": { "type": "string" },
        "user": { ②
          "type": "object",
          "properties": {
            "id": { "type": "string" },
            "gender": { "type": "string" },
            "age": { "type": "long" },
            "name": { ②
              "type": "object",
              "properties": {
                "full": { "type": "string" },
                "first": { "type": "string" },
                "last": { "type": "string" }
              }
            }
          }
        }
      }
    }
  }
}
```

① 根 象

② 内部 象

`user` 和 `name` 域的映射 与 `tweet` 型的相同。事 上，`type` 映射只是一 特殊的 象 映射，我 称之 根 象。除了它有一些文 元数据的特殊 域，例如 `_source` 和 `_all` 域，它和其他 象一 。

内部 象是如何索引的

Lucene 不理解内部 象。 Lucene 文 是由一 列表 成的。 了能 Elasticsearch 有效地索引内部 ， 它把我 的文 化成 ：

```
{
  "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 名称引用（例如， `first` ）。 了区分同名的 个域，我 可以使用全 路径 （例如，

`user.name.first`) 或 `type` 名加路径 (`tweet.user.name.first`) 。

NOTE 在前面 扁平的文 中，没有 `user` 和 `user.name` 域。Lucene 索引只有 量和 ，没有 数据 。

内部 象数

最后，考 包含内部 象的数 是如何被索引的。假 我 有个 `followers` 数 :

```
{  
  "followers": [  
    { "age": 35, "name": "Mary White"},  
    { "age": 26, "name": "Alex Jones"},  
    { "age": 19, "name": "Lisa Smith"}  
  ]  
}
```

个文 会像我 之前描述的那 被扁平化 理， 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 的相 性已 失了，因 个多 域只是一包无序的 ，而不是有序数 。 足以 我 ，“有一个26 的追随者？”

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

相 内部 象被称 `nested` 象，可以回答上面的 ，我 后会在 [嵌套 象中介](#) 它。

求体

 易 —search-lite—于用命令行 行点 点 (ad-hoc) 是非常有用的。然而， 了充分利用 的 大功能， 使用 求体 <code>search</code> API, 之所以称之 求体 (Full-Body Search), 因 大部分参数是通 Http 求体而非 字符串来 的。

求体 —下文 称 —不 可以 理自身的 求， 允 果 行片段 (高亮)、 所有或部分 果 行聚合分析，同 可以 出 是不是想 的建 ， 些建 可以引 使用者快速 到他想要的 果。

空

我 以最 的 `search` API 的形式 我 的旅程，空 将返回所有索引 (indices)中的所有文 :

```
GET /_search
```

```
{}
```

① 是一个空的 求体。

只用一个 字符串， 就可以在一个、多个或者 `_all` 索引 (indices) 和一个、多个或者所有types 中：

```
GET /index_2014*/type1,type2/_search
```

```
{}
```

同 可以使用 `from` 和 `size` 参数来分：

```
GET /_search
```

```
{
  "from": 30,
  "size": 10
}
```

一个 求体的 GET 求？

某些特定 言 (特 是 JavaScript) 的 HTTP 是不允 `GET` 求 有 求体的。事 上，一些使用者 于 `GET` 求可以 求体感到非常的吃 。

而事 是 个RFC文 [RFC 7231](http://tools.ietf.org/html/rfc7231#page-24)； 一个 理 HTTP 和内容的文  — 并没有 定一个 有 求体的 `<code>GET</code>` 求 如何 理！ 果是，一些 HTTP 服 器 允 子，而有一些 — 特 是一些用于 存和代理的服 器 —  不允 。

于一个 求， Elasticsearch 的工程 倾向于使用 `GET` 方式，因 他 得它比 `POST` 能更好的描述信息 索 (retrieving information) 的行 。然而，因 求体的 `GET` 求并不被广泛支持，所以 `search API` 同 支持 `POST` 求：

```
POST /_search
```

```
{
  "from": 30,
  "size": 10
}
```

似的 可以 用于任何需要 求体的 `GET API`。

我 将在聚合 [聚合](#) 章 深入介 聚合 (aggregations)，而 在，我 将聚焦在 。

相 于使用晦 的 字符串的方式，一个 求体的 允 我 使用 域特定 言 (`query`

domain-specific language) 或者 Query DSL 来写句。

表 式

表式(Query DSL)是一非常活又富有表达力的语言。Elasticsearch 使用它可以以的 JSON 接口来展示 Lucene 功能的大部分。在的用中，用它来写的句。它可以使的句更活、更精、易和易。

要使用表式，只需将句 `query` 参数：

```
GET /_search
{
  "query": YOUR_QUERY_HERE
}
```

空 (empty search) `{}` 在功能上等同于使用<code>match_all</code>，正如其名字一，匹配所有文：

```
GET /_search
{
  "query": {
    "match_all": {}
  }
}
```

句的

一个句的典型：

```
{
  QUERY_NAME: {
    ARGUMENT: VALUE,
    ARGUMENT: VALUE,...,
  }
}
```

如果是某个字段，那它的如下：

```
{  
    QUERY_NAME: {  
        FIELD_NAME: {  
            ARGUMENT: VALUE,  
            ARGUMENT: VALUE,...  
        }  
    }  
}
```

个例子， 可以使用 `match` 句 来 在 `tweet` 字段中包含 `elasticsearch` 的 tweet：

```
{  
    "match": {  
        "tweet": "elasticsearch"  
    }  
}
```

完整的 求如下：

```
GET /_search  
{  
    "query": {  
        "match": {  
            "tweet": "elasticsearch"  
        }  
    }  
}
```

合并 句

句(*Query clauses*) 就像一些 的 合， 些 合 可以彼此之 合并 成更 的。 些 句可以是如下形式：

- 叶子 句 (*Leaf clauses*) (就像 `match` 句) 被用于将 字符串和一个字段 (或者多个字段) 比。
- 合(*Compound*) 句 主要用于 合并其它 句。 比如，一个 `bool` 句 允 在 需要的 候 合其它 句，无 是 `must` 匹配、 `must_not` 匹配 是 `should` 匹配，同 它可以包含不 分的 器 (filters)：

```
{
  "bool": {
    "must": { "match": { "tweet": "elasticsearch" }},
    "must_not": { "match": { "name": "mary" }},
    "should": { "match": { "tweet": "full text" }},
    "filter": { "range": { "age": { "gt": 30 } } }
  }
}
```

一条合句可以合并任何其它句，包括合句，了解一点是很重要的。就意味着，合句之可以互相嵌套，可以表达非常。

例如，以下是为了出信件正文包含 `business opportunity` 的星件，或者在收件箱正文包含 `business opportunity` 的非件：

```
{
  "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，不必太在意这个例子的，我会在后面解。最重要的是要理解到，一条合句可以将多条句—叶子句和其它合句—合并成一个—的句。

与

Elasticsearch 使用的语（DSL）有一套件，些件可以以无限合的方式行搭配。套件可以在以下情况下使用：情况（filtering context）和情况（query context）。

当使用于情况，被置成一个“不分”或者“”。即，个只是的一个：“篇文是否匹配？”。回答也是非常的，yes 或者 no，二者必居其一。

- `created` 是否在 2013 与 2014 个区？
- `status` 字段是否包含 `published` 个？
- `lat_lon` 字段表示的位置是否在指定点的 10km 内？

当使用于情况，就成了一个“分”的。和不分的似，也要去判断一个文是否匹配，同它需要判断一个文匹配的有多好（匹配程度如何）。此的典型用法是用于以下文：

- 与 full text search 个 最佳匹配的文
- 包含 run 个 , 也能匹配 runs 、 running 、 jog 或者 sprint
- 包含 <code>quick</code> 、 <code>brown</code> 和 <code>fox</code> 几个 ; 之 的越近, 文 相 性越高
- 有 <code>lucene</code> 、 <code>search</code> 或者 <code>java</code> 越多, 相 性越高

一个 分 算 一个文 与此 的 相 程度, 同 将 个相 程度分配 表示相 性的字段 `_score`, 并且按照相 性 匹配到的文 行排序。 相 性的概念是非常 合全文搜索的情况, 因 全文搜索几乎没有完全 ``正 '' 的答案。

自 Elasticsearch 世以来, 与 (queries and filters) 就独自成 Elasticsearch 的 件。但从 Elasticsearch 2.0 始, (filters) 已 从技 上被排除了, 同 所有的 (queries) 有 成不 分 的能力。

NOTE 然而, 了明 和 , 我 用 "filter" 个 表示不 分、只 情况下的 。 可以把 "filter" 、 "filtering query" 和 "non-scoring query" 几个 相同的。

相似的, 如果 独地不加任何修 地使用 "query" 个 , 我 指的是 "scoring query" 。

性能差

(Filtering queries) 只是 的 包含或者排除, 就使得 算起来非常快。考 到至少有一个 (filtering query) 的 果是 “稀少的” (很少匹配的文) , 并且 常使用不分 (non-scoring queries) , 果会被 存到内存中以便快速 取, 所以有各 各 的手段来化 果。

相反, 分 (scoring queries) 不 要 出匹配的文 , 要 算 个匹配文 的相 性, 算相 性使得它 比不 分 力的多。同 , 果并不 存。

多 倒排索引 (inverted index) , 一个 的 分 在匹配少量文 可能与一个涵 百万文 的 filter表 的一 好, 甚至会更好。但是在一般情况下, 一个filter 会比一个 分的query性能更 , 并且 次都表 的很 定。

(filtering) 的目 是 少那些需要通 分 (scoring queries) 行 的文 。

如何 与

通常的 是, 使用 (query) 句来 行 全文 搜索或者其它任何需要影 相 性得分的搜索。除此以外的情况都使用 (filters)。

最重要的

然 Elasticsearch 自 了很多的 , 但 常用到的也就那 几个。我 将在 深入搜索 章 那些 的 , 接下来我 最重要的几个 行 介 。

match_all

`match_all` 的匹配所有文 。在没有指定 方式 ，它是 的 ：

```
{ "match_all": {}}
```

它 常与 filter 合使用—例如， 索收件箱里的所有 件。所有 件被 具有相同的相 性，所以都将 得分 1 的中性 `_score`。

match

无 在任何字段上 行的是全文搜索 是精 ， `match` 是 可用的 准 。

如果 在一个全文字段上使用 `match` ，在 行 前，它将用正 的分析器去分析 字符串：

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

如果在一个精 的字段上使用它，例如数字、日期、布 或者一个 `not_analyzed` 字符串字段，那 它将会精 匹配 定的 ：

```
{ "match": { "age": 26 } }
{ "match": { "date": "2014-09-01" } }
{ "match": { "public": true } }
{ "match": { "tag": "full_text" } }
```

TIP 于精 的 ， 可能需要使用 filter 句来取代 query，因 filter 将会被 存。接下来，我 将看到一些 于 filter 的例子。

不像我 在 量 搜索 章 介 的字符串 (query-string search)，`match` 不使用 似 `+user_id:2 +tweet:search` 的 法。它只是去 定的 。 就意味着将 字段暴露 的用 是安全的； 需要控制那些允 被 字段，不易于 出 法 常。

multi_match

`multi_match` 可以在多个字段上 行相同的 `match` ：

```
{
  "multi_match": {
    "query": "full text search",
    "fields": [ "title", "body" ]
  }
}
```

range

range 指定区间的数字或者：

```
{  
  "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

term 被用于精确匹配，一些精确可能是数字、布或者那些 **not_analyzed** 的字符串：

```
{ "term": { "age": 26 } }  
{ "term": { "date": "2014-09-01" } }  
{ "term": { "public": true } }  
{ "term": { "tag": "full_text" } }
```

term 于入的文本不分析，所以它将定的行精。

terms

terms 和 **term** 一，但它允指定多行匹配。如果个字段包含了指定中的任何一个，那多个文足条件：

```
{ "terms": { "tag": [ "search", "full_text", "nosql" ] } }
```

和 **term** 一，**terms** 于入的文本不分析。它那些精匹配的（包括在大小写、重音、空格等方面的不同）。

exists 和 missing

`exists` 和 `missing` 被用于 那些指定字段中有 (`exists`) 或无 (`missing`) 的文 。 与 SQL中的 `IS_NULL (missing)` 和 `NOT IS_NULL (exists)` 在本 上具有共性：

```
{  
  "exists": {  
    "field": "title"  
  }  
}
```

些 常用于某个字段有 的情况和某个字段 的情况。

合多

的 需求从来都没有那 ；它 需要在多个字段上 多 多 的文本，并且根据一系列的 准 来 。 了 建 似的高 ， 需要一 能 将多 合成 一 的 方法。

可以用 `bool` 来 的需求。 将多 合在一起，成 用 自己想要的布 。 它接收以下参数：

`must`

文 必 匹配 些条件才能被包含 来。

`must_not`

文 必 不 匹配 些条件才能被包含 来。

`should`

如果 足 些 句中的任意 句，将 加 `_score` ，否 ， 无任何影 。它 主要用于修正 个文 的相 性得分。

`filter`

必 匹配，但它以不 分、 模式来 行。 些 句 分没有 献，只是根据 准来排除或包含文 。

由于 是我 看到的第一个包含多个 的 ， 所以有必要 一下相 性得分是如何 合的。 一个子 都独自地 算文 的相 性得分。一旦他 的得分被 算出来， `bool` 就将 些得分 行合并并且返回一个代表整个布 操作的得分。

下面的 用于 `title` 字段匹配 `how to make millions` 并且不被 `spam` 的文 。那些被 `starred` 或在2014之后的文 ， 将比 外那些文 有更高的排名。如果 者 都 足， 那 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" }}}
    ]
  }
}
```

TIP 如果没有 `must` 句，那 至少需要能 匹配其中的一条 `should` 句。但，如果存在至少一条 `must` 句，`should` 句的匹配没有要求。

加 器 (`filtering`) 的

如果我 不想因 文 的 而影 得分，可以用 `filter` 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" }} ①
    }
  }
}
```

① range 已 从 `should` 句中移到 `filter` 句

通 将 range 移到 filter 句中，我 将它 成不 分的 ，将不再影 文 的相 性排名。由于它 在是一个不 分的 ，可以使用各 filter 有效的 化手段来提升性能。

所有 都可以借 方式。将 移到 bool 的 filter 句中， 它就自 的 成一个不 分的 filter 了。

如果 需要通 多个不同的 准来 的文 ， bool 本身也可以被用做不 分的 。 地将它放置到 filter 句中并在内部 建布：

```
{
  "bool": {
    "must": { "match": { "title": "how to make millions" }},
    "must_not": { "match": { "tag": "spam" }},
    "should": [
      { "match": { "tag": "starred" }}
    ],
    "filter": {
      "bool": { ①
        "must": [
          { "range": { "date": { "gte": "2014-01-01" }}},
          { "range": { "price": { "lte": 29.99 }}}
        ],
        "must_not": [
          { "term": { "category": "ebooks" }}
        ]
      }
    }
  }
}
```

① 将 `bool` 包 在 `filter` 句中，我 可以在 准中 加布

通 混合布 ，我 可以在我 的 求中 活地 写 `scoring` 和 `filtering` 。

`constant_score`

尽管没有 `bool` 使用 繁，`constant_score` 也是 工具箱里有用的 工具。它将一个不 的常量 分 用于所有匹配的文 。它被 常用于 只需要 行一个 `filter` 而没有其它 （例如，分 ）的情况下。

可以使用它来取代只有 `filter` 句的 `bool` 。在性能上是完全相同的，但 于提高 性和清晰度有很大 助。

```
{
  "constant_score": {
    "filter": {
      "term": { "category": "ebooks" } ①
    }
  }
}
```

① `term` 被放置在 `constant_score` 中， 成不 分的 `filter`。 方式可以用来取代只有 `filter` 句的 `bool` 。

可以 得非常的 ，尤其和不同的分析器与不同的字段映射 合 ，理解起来就有点困 了。不

validate-query API 可以用来 是否合法。

```
GET /gb/tweet/_validate/query
{
  "query": {
    "tweet" : {
      "match" : "really powerful"
    }
  }
}
```

以上 **validate** 求的 答告 我 个 是不合法的：

```
{
  "valid" : false,
  "_shards" : {
    "total" : 1,
    "successful" : 1,
    "failed" : 0
  }
}
```

理解 信息

了 出 不合法的原因，可以将 **explain** 参数 加到 字符串中：

```
GET /gb/tweet/_validate/query?explain ①
{
  "query": {
    "tweet" : {
      "match" : "really powerful"
    }
  }
}
```

① **explain** 参数可以提供更多 于 不合法的信息。

很明 ， 我 将 型(**match**)与字段名称 (**tweet**) 混了：

```
{
  "valid" : false,
  "_shards" : { ... },
  "explanations" : [ {
    "index" : "gb",
    "valid" : false,
    "error" : "org.elasticsearch.index.query.QueryParsingException:
      [gb] No query registered for [tweet]"
  } ]
}
```

理解句

于合法，使用 `explain` 参数将返回可的描述，准理解 Elasticsearch 是如何解析的 query 是非常有用的：

```
GET /_validate/query?explain
{
  "query": {
    "match" : {
      "tweet" : "really powerful"
    }
  }
}
```

我 的一个 index 都会返回 的 `explanation`，因一个 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` 被重写 个 `tweet` 字段的 single-term，一个single-term 字符串分出来的一个term。

当然，于索引 `us`，一个 term 分是 `really` 和 `powerful`，而于索引 `gb`，term 分是 `realli` 和 `power`。之所以出 个情况，是由于我 将索引 `gb` 中 `tweet` 字段的分析器修改 `english` 分析器。

排序与相 性

情况下，返回的 果是按照 *相 性* 行排序的；最相 的文 排在最前。 在本章的后面部分，我 会解 *相 性* 意味着什 以及它是如何 算的，不 我 首先看看 `sort` 参数以及如何使用它。

排序

了按照相 性来排序，需要将相 性表示 一个数 。在 Elasticsearch 中，相 性得分 由一个浮点数 行表示，并在搜索 果中通 `_score` 参数返回， 排序是 `_score` 降序。

有 ，相 性 分 来 并没有意 。例如，下面的 返回所有 `user_id` 字段包含 1 的 果：

```
GET /_search
{
  "query": {
    "bool": {
      "filter": {
        "term": {
          "user_id": 1
        }
      }
    }
  }
}
```

里没有一个有意 的分数：因 我 使用的是 filter ()， 表明我 只希望 取匹配 `user_id: 1` 的文 ，并没有 定 些文 的相 性。 上文 将按照随机 序返回，并且 个文 都会 零分。

如果 分 零 造成了困 ， 可以使用 `constant_score` 行替代：

```
GET /_search
{
  "query": {
    "constant_score": {
      "filter": {
        "term": {
          "user_id": 1
        }
      }
    }
  }
}
```

NOTE

将 所有文 用一个恒定分数 (1)。它将 行与前述 相同的 ，并且所有的文 将像之前一 随机返回， 些文 只是有了一个分数而不是零分。

按照字段的 排序

在 个案例中，通 来 tweets 行排序是有意 的，最新的 tweets 排在最前。我 可以使用 `sort` 参数 行：

```
GET /_search
{
  "query": {
    "bool": {
      "filter": { "term": { "user_id": 1 } }
    }
  },
  "sort": { "date": { "order": "desc" } }
}
```

会注意到 果中的 个不同点：

```
"hits": {
  "total": 6,
  "max_score": null, ①
  "hits": [ {
    "_index": "us",
    "_type": "tweet",
    "_id": "14",
    "_score": null, ①
    "_source": {
      "date": "2014-09-24",
      ...
    },
    "sort": [ 141151680000 ] ②
  },
  ...
}
```

① `_score` 不被 算，因 它并没有用于排序。

② `date` 字段的 表示 自 epoch (January 1, 1970 00:00:00 UTC)以来的 秒数，通 `sort` 字段的 行返回。

首先我 在 个 果中有一个新的名 `sort` 的元素，它包含了我 用于排序的 。 在 个案例中，我 按照 `date` 行排序，在内部被索引 自 epoch 以来的 秒数。 long 型数 `141151680000` 等于日期字符串 `2014-09-24 00:00:00 UTC`。

其次 `_score` 和 `max_score` 字段都是 `null`。 算 `_score` 的花 巨大，通常 用于排序；我 并不根据相 性排序，所以 `_score` 是没有意 的。如果无 如何 都要 算 `_score`， 可以将 `track_scores` 参数 置 `true`。

一个 便方法是， 可以指定一个字段用来排序：

TIP

```
"sort": "number_of_children"
```

字段将会 升序排序， 而按照 `_score` 的 行降序排序。

多 排序

假定我 想要 合使用 `date` 和 `_score` 行，并且匹配的 果首先按照日期排序， 然后按照相 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 序是很重要的。 果首先按第一个条件排序， 当 果集的第一个 `sort` 完全相同 才会按照第二个条件 行排序， 以此 推。

多 排序并不一定包含 `_score` 。 可以根据一些不同的字段 行排序， 如地理距 或是脚本 算的特定。

Query-string 搜索也支持自定 排序， 可以在 字符串中使用 `sort` 参数：

NOTE

```
GET /_search?sort=date:desc&sort=_score&q=search
```

字段多 的排序

一 情形是字段有多个 的排序， 需要 住 些 并没有固有的 序；一个 多 的字段 是多个 的包装， 个 行排序 ？

于数字或日期， 可以将多 字段 ， 可以通 使用 `min` 、 `max` 、 `avg` 或是 `sum` 排序模式 。 例如 可以按照 个 `date` 字段中的最早日期 行排序， 通 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字段，但是很少会按照想要的方式行排序。如果想分析一个字符串，如 fine old art，包含 3 个。我很可能想要按第一的字母排序，然后按第二的字母排序，如此，但是 Elasticsearch 在排序程序中没有的信息。

可以使用 min 和 max 排序模式（是 min），但是会导致排序以 art 或是 old，任何一个都不是所希望的。

了以字符串字段行排序，个字段包含一：整个 not_analyzed 字符串。但是我需要 analyzed 字段，才能以全文行

一个的方法是用方式同一个字符串行索引，将在文 中包括个字段：analyzed 用于搜索，not_analyzed 用于排序

但是保存相同的字符串次在 source 字段是浪空的。我真正想要做的是一个 _ 字段但是却用方式索引它。所有的 _core_field 型 (strings, numbers, Booleans, dates) 接收一个 fields 参数

参数允化一个的映射如：

```
"tweet": {  
    "type": "string",  
    "analyzer": "english"  
}
```

一个多字段映射如：

```
"tweet": { ①  
    "type": "string",  
    "analyzer": "english",  
    "fields": {  
        "raw": { ②  
            "type": "string",  
            "index": "not_analyzed"  
        }  
    }  
}
```

① tweet 主字段与之前的一：是一个 analyzed 全文字段。

② 新的 tweet.raw 子字段是 not_analyzed.

在，至少只要我 重新索引了我 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"
}
```

WARNING 以全文 `analyzed` 字段排序会消耗大量的内存。 取更多信息 看 [聚合与分析](#)。

什 是相 性？

我 曾 ， 情况下，返回 果是按相 性倒序排列的。但是什 是相 性？相 性如何 算？

个文 都有相 性 分，用一个正浮点数字段 `_score` 来表示。`_score` 的 分越高，相 性越高。

句会 个文 生成一个 `score` 字段。 分的 算方式取决于 型 不同的 句用于不同的目的： `fuzzy` 会 算与 的 写相似程度，`terms` 会 算 到的内容与 成部分匹配的百分比，但是通常我 的 `_relevance` 是我 用来 算全文本字段的 相 于全文本 索 相似程度的算法。

Elasticsearch 的相似度算法被定 索 率/反向文 率， `TF/IDF`，包括以下内容：

索 率

索 在 字段出 的 率？出 率越高，相 性也越高。 字段中出 5 次要比只出 1 次的相 性高。

反向文 率

个 索 在索引中出 的 率？ 率越高，相 性越低。 索 出 在多数文 中会比出 在少数文 中的 重更低。

字段 度准

字段的 度是多少？ 度越 ，相 性越低。 索 出 在一个短的 title 要比同 的 出 在一个 的 content 字段 重更大。

个 可以 合使用 `TF/IDF` 和其他方式，比如短 中 索 的距 或模糊 里的 索 相似度。

相 性并不只是全文本 索的 利。也 用于 `yes|no` 的子句，匹配的子句越多，相 性 分越高。

如果多条 子句被合并 一条 合 句，比如 `bool` ， 个 子句 算得出的 分会被合并到 的相 性 分中。

TIP 我 有一 整章着眼于相 性 算和如何 其配合 的需求 控制相 度。

理解 分 准

当 一条 的 句 , 想要理解 `_score` 究竟是如何 算是比 困 的。Elasticsearch 在 个 句中都有一个 `explain` 参数, 将 `explain true` 就可以得到更 的信息。

```
GET /_search?explain ①
{
  "query" : { "match" : { "tweet" : "honeymoon" } }
}
```

① `explain` 参数可以 返回 果添加一个 `_score` 分的得来依据。

NOTE 加一个 `explain` 参数会 个匹配到的文 生一大堆 外内容, 但是花 去理解它是很有意 的。 如果 在看不明白也没 系一等 需要的 候再回 一 就行。下面我 来一点点的了解 知 点。

首先, 我 看一下普通 返回的元数据 :

```
{
  "_index" : "us",
  "_type" : "tweet",
  "_id" : "12",
  "_score" : 0.076713204,
  "_source" : { ... trimmed ... },
```

里加入了 文 来自于 个 点 个分片上的信息, 我 是比 有 助的, 因 率和 文 率是在 个分片中 算出来的, 而不是 个索引中 :

```
"_shard" : 1,
"_node" : "mzIVYCsqSWCG_M_ZffSs9Q",
```

然后它提供了 `_explanation` 。 个入口都包含一个 `description` 、 `value` 、 `details` 字段, 它分 告 算的 型、 算 果和任何我 需要的 算 。

```

"_explanation": { ①
  "description": "weight(tweet:honeymoon in 0)
                  [PerFieldSimilarity], result of:",
  "value": 0.076713204,
  "details": [
    {
      "description": "fieldWeight in 0, product of:",
      "value": 0.076713204,
      "details": [
        { ②
          "description": "tf(freq=1.0), with freq of:",
          "value": 1,
          "details": [
            {
              "description": "termFreq=1.0",
              "value": 1
            }
          ]
        },
        { ③
          "description": "idf(docFreq=1, maxDocs=1)",
          "value": 0.30685282
        },
        { ④
          "description": "fieldNorm(doc=0)",
          "value": 0.25,
        }
      ]
    }
  ]
}

```

① `honeymoon` 相 性 分 算的

② 索 率

③ 反向文 率

④ 字段 度准

WARNING

出 `explain` 果代 是十分昂 的，它只能用作 工具 。千万不要用于生
境。

第一部分是 于 算的 。告 了我 `honeymoon` 在 `tweet` 字段中的 索 率/反向文 率或TF/IDF, (里的文 ① 是一个内部的 ID, 跟我 没有 系, 可以忽略。)

然后它提供了 重是如何 算的 :

索 率:

索 'honeymoon' 在一个文档的 'tweet' 字段中的出现次数。

反向索引率：

索 'honeymoon' 在索引上所有文档的 'tweet' 字段中出现的次数。

字段度量：

在一个文档中，'tweet' 字段内容的度 -- 内容越少，越小。

的句解也非常少，但是包含的内容与上面例子大致相同。通过这段信息我可以了解搜索结果是如何生成的。

TIP JSON 形式的 `explain` 描述是以 XML 的，但是改成 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` 加到某一文档上，`explain` API 会帮助理解何个文档会被匹配，更重要的是，一个文档为何没有被匹配。

请求路径 `/index/type/id/_explain`，如下所示：

```
GET /us/tweet/12/_explain
{
  "query": {
    "bool": {
      "filter": { "term": { "user_id": 2 } },
      "must": { "match": { "tweet": "honeymoon" } }
    }
  }
}
```

不只是我之前看到的充分解释，我现在有了一个 `description` 元素，它将告诉我：

```
"failure to match filter: cache(user_id:[2 TO 2])"
```

也就是我的 `user_id` 子句使文档不能匹配到。

Doc Values 介绍

本章的最后一个部分是关于 Elasticsearch 内部的一些运行情况。在这里我先不介绍新的知识点，所以我特意提到，Doc Values 是我需要反提到的一个重要部分。

当一个字段行排序，Elasticsearch 需要得到相匹配的文。倒排索引的索性能是非常快的，但是在字段排序却不是理想的。

- 在搜索的时候，我 能通过搜索快速得到果集。
- 当排序的时候，我 需要在倒排索引里面某个字段的集合。句，我 需要倒置倒排索引。

倒置 在其他系中常被称作 **列存**。上，它将所有字段的存 在数据列中，使得其行操作是十分高效的，例如排序。

在 Elasticsearch 中，doc values 就是一列式存，情况下一个字段的 doc values 都是激活的，doc values 是在索引建的，当字段索引，Elasticsearch 了能快速索，会把字段的加入倒排索引中，同 它也会存 字段的 doc values。

Elasticsearch 中的 doc vaules 常被用到以下景：

- 一个字段行排序
- 一个字段行聚合
- 某些，比如地理位置
- 某些与字段相 的脚本 算

因 文 被序列化到磁，我可以依 操作系 的 助来快速。当 working set 小于点的可用内存，系 会自 将所有的文 保存在内存中，使得其写十分高速；当其大于可用内存，操作系 会自 把 doc values 加 到系 的 存中，从而避免了 jvm 堆内存溢出常。

我 后会深入 doc values。在所有 需要知道的是排序 生在索引 建立的平行数据 中。

行分布式 索

在 之前，我 将 道 一下在分布式 境中搜索是 行的。 比我 在 分布式文 存章 的基本的 - 改- (CRUD) 求要 一些。

内容提示

可以根据 趣 本章内容。 并不需要 了使用 Elasticsearch 而理解和 住所有的 。

章的 目的只 初 了解下工作原理，以便将来需要 可以及 到 些知 ， 但是不要被 所困 。

一个 CRUD 操作只 个文 行 理，文 的唯一性由 `_index`, `_type`, 和 `routing` values (通常 是文 的 `_id`) 的 合来 定。 表示我 切的知道集群中 个分片含有此文 。

搜索需要一 更加 的 行模型因 我 不知道 会命中 些文： 些文 有可能在集群的任何分片上。 一个搜索 求必 我 注的索引 (index or indices) 的所有分片的某个副本来 定它 是否含有任何匹配的文 。

但是 到所有的匹配文 完成事情的一半。 在 `search` 接口返回一个 `page` 果之前，多分片中的

果必 合成 个排序列表。此，搜索被 行成一个 段 程，我 称之 *query then fetch*。

段

在初始 段 ， 会广播到索引中 一个分片拷 （主分片或者副本分片）。 个分片在本地 行搜索并 建一个匹配文 的 先 列。

先 列

一个 先 列 是一个存有 *top-n* 匹配文 的有序列表。 先 列的大小取决于分 参数 `from` 和 `size`。例如，如下搜索 求将需要足 大的 先 列来放入100条文 。

```
GET /_search
{
  "from": 90,
  "size": 10
}
```

个 段的 程如 程分布式搜索 所示。

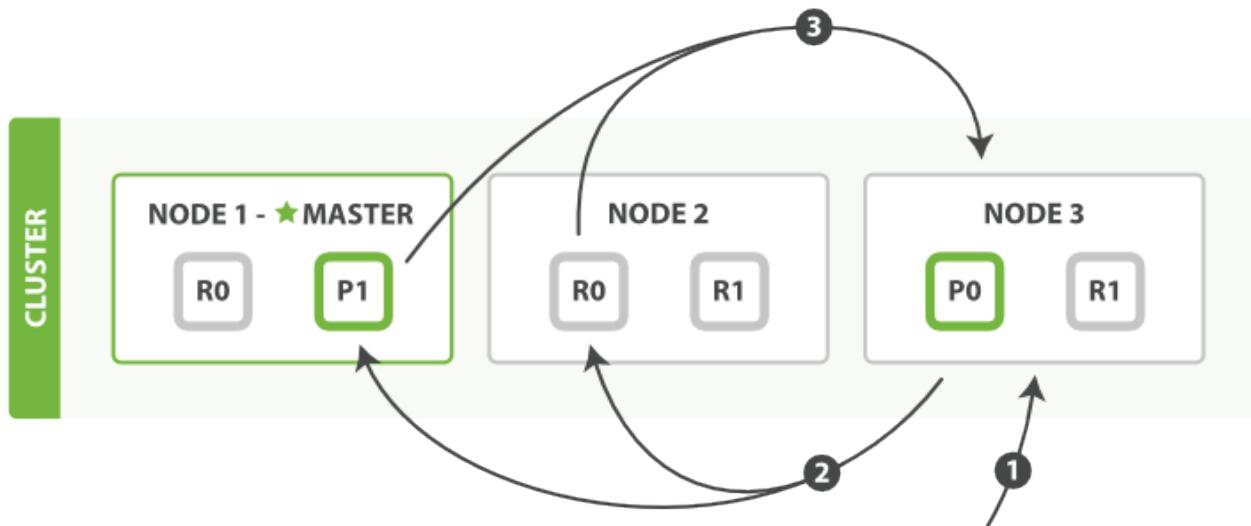


Figure 14. 程分布式搜索

段包含以下三个 ：

1. 客 端 送一个 `search` 求到 Node 3 , Node 3 会 建一个大小 `from + size` 的空 先 列。
2. Node 3 将 求 到索引的 个主分片或副本分片中。 个分片在本地 行 并添加 到大小 `from + size` 的本地有序 先 列中。
3. 个分片返回各自 先 列中所有文 的 ID 和排序 点，也就是 Node 3 ，它合并 些 到自己的 先 列中来 生一个全局排序后的 果列表。

当一个搜索 求被 送到某个 点 ， 个 点就 成了 点。

个 点的任 是广播

求到所有相 分片并将它 的 整合成全局排序后的 果集合， 个 果集合会返回 客 端。

第一 是广播 求到索引中 一个 点的分片拷 。就像 `document GET requests` 所描述的， 求可以被某个主分片或某个副本分片 理， 就是 什 更多的副本（当 合更多的硬件）能 加搜索 吐率。 点将在之后的 求中 所有的分片拷 来分 。

个分片在本地 行 求并且 建一个 度 `<code>from + size</code>` 的 先 列—； 也就是 ， 个分片 建的 果集足 大，均可以 足全局的搜索 求。 分片返回一个 量 的 果列表到 点，它 包含文 ID 集合以及任何排序需要用到的 ，例如 `<code>_score</code>` 。

点将 些分片 的 果合并到自己的有序 先 列里，它代表了全局排序 果集合。至此 程束。

NOTE 一个索引可以由一个或几个主分片 成， 所以一个 个索引的搜索 求需要能 把来自多个分片的 果 合起来。 *multiple* 或者 *all* 索引的搜索工作方式也是完全一致的— 是包含了更多的分片而已。

取回 段

段 些文 足搜索 求，但是我 然需要取回 些文 。 是取回 段的任 ， 正如 [分布式搜索的取回 段](#) 所展示的。

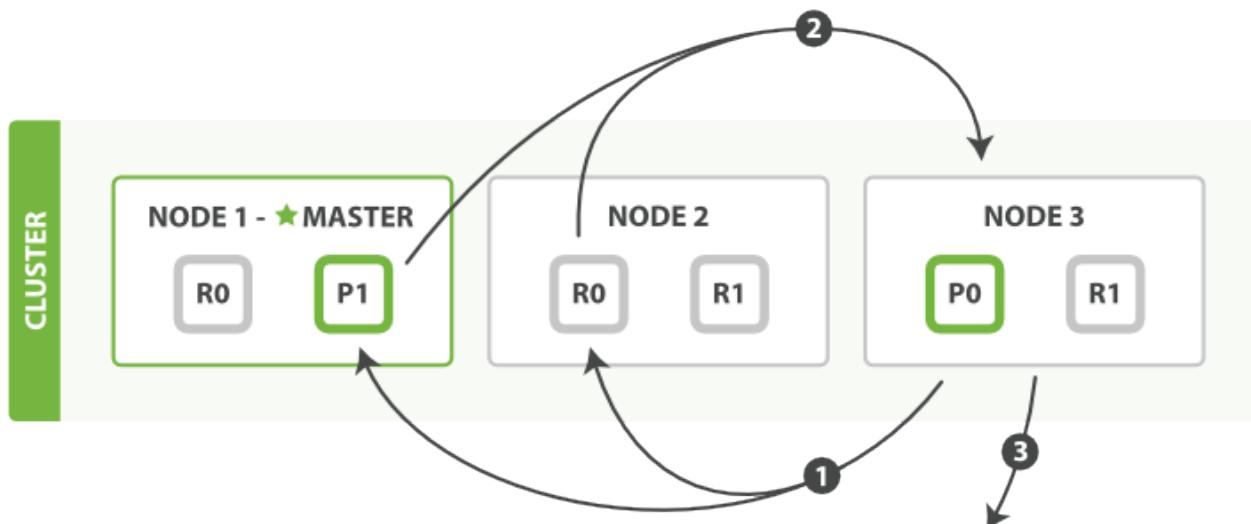


Figure 15. 分布式搜索的取回 段

分布式 段由以下 成：

1. 点 出 些文 需要被取回并向相 的分片提交多个 `GET` 求。
2. 个分片加 并 富文 ，如果有需要的 ，接着返回文 点。
3. 一旦所有的文 都被取回了， 点返回 果 客 端。

点首先决定 些文 需要被取回。例如，如果我 的 指定了 `{ "from": 90, "size": 10 }` ，最初90个 果会被 ，只有从第91个 始的10个 果需要被取回。 些文 可能来自和最初搜索 求有 的一个、多个甚至全部分片。

点 持有相 文 的 个分片 建一个 `multi-get` request , 并 送 求 同 理段的分片副本。

分片加 文 体- `_source` 字段—如果有需要，用元数据和 `search snippet highlighting` 富 果文 。一旦 点接收到所有的 果文 ， 它就 装 些 果 个 返回 客 端。

深分 (Deep Pagination)

先 后取的 程支持用 `from` 和 `size` 参数分 , 但是 是 有限制的 。 要 住需要 信息点的 个分片必 先 建一个 `from + size` 度的 列, 点需要根据 `number_of_shards * (from + size)` 排序文 , 来 到被包含在 `size` 里的文 。

取决于 的文 的大小, 分片的数量和 使用的硬件, 10,000 到 50,000 的 果文 深分 (1,000 到 5,000)是完全可行的。但是使用足 大的 `from` , 排序 程可能会 得非常重, 使用大量的CPU、内存和 。因 个原因, 我 烈建 不要使用深分 。

上, ``深分 '' 很少符合人的行 。当2到3 去以后, 人会停止翻 , 并且改 搜索准。会不知疲倦地一 一 的 取 直到 的服 崩 的罪魁 首一般是机器人或者web spider。

如果 需要从 的集群取回大量的文 , 可以通 用 `scroll` 禁用排序使 个取回行更有效率, 我 会在 `later in this chapter` 行 。

搜索

有几个 参数可以影 搜索 程。

偏好

偏好 个参数 `preference` 允 用来控制由 些分片或 点来 理搜索 求。 它接受像 `_primary`, `_primary_first`, `_local`, `_only_node:xyz`, `_prefer_node:xyz`, 和 `_shards:2,3` 的 , 些 在 `search preference` 文 面被 解 。

但是最有用的 是某些随机字符串, 它可以避免 *bouncing results* 。

Bouncing Results

想象一下有 个文 有同 的 字段, 搜索 果用 `timestamp` 字段来排序。 由于搜索求是在所有有效的分片副本 的, 那就有可能 生主分片 理 求 , 个文 是一 序, 而副本分片 理 求 又是 一 序。

就是所 的 *bouncing results* : 次用 刷新 面, 搜索 果表 是不同的 序。同一个用 始 使用同一个分片, 可以避免 , 可以 置 `preference` 参数一个特定的任意 比如用 会 ID来解决。

超

通常分片 理完它所有的数据后再把 果返回 同 点， 同 点把收到的所有 果合并 最 果。

意味着花 的 是最慢分片的 理 加 果合并的 。如果有 一个 点有 ，就会 致所有 的 慢。

参数 `timeout` 告 分片允 理数据的最大 。如果没有足 的 理所有数据， 个分片的 果可以是部分的，甚至是空数据。

搜索的返回 果会用属性 `timed_out` 明分片是否返回的是部分 果：

```
...  
"timed_out": true, ①  
...
```

① 个搜索 求超 了。

超 然是一个最有效的操作，知道 一点很重要； 很可能 会超 定的超 行 有 个原因：

1. 超 是基于 文 做的。 但是某些 型有大量的工作在文 估之前需要完成。 "setup" 段并不考 超 置，所以太 的建立 会 致超 超 的整体延 。
2. 因 是基于 个文 的，一次 在 个文 上 行并且在下个文 被 估之前不会超 。 也意味着差的脚本（比如 无限循 的脚本）将会永 行下去。

WARNING

路由

在 [路由一个文 到一个分片中](#) 中，我 解 如何定制参数 `routing`，它能 在索引 提供来 保相 的文，比如属于某个用 的文 被存 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 指定几个 `routing` 来限定只搜索几个相 的分片：

```
GET /_search?routing=user_1,user2
```

个技 在 大 模搜索系 就会派上用 ，我在 容 中 它。

搜索 型

省的搜索 型是 `query_then_fetch` 。 在某些情况下， 可能想明 置 `search_type` `dfs_query_then_fetch` 来改善相 性精 度：

```
GET /_search?search_type=dfs_query_then_fetch
```

搜索 型 `dfs_query_then_fetch` 有 段， 个 段可以从所有相 分片 取 来 算全局 。 我 在 被破坏的相 度！会再 它。

游 'Scroll'

scroll 可以用来 Elasticsearch 有效地 行大批量的文 , 而又不用付出深度分 那 代 。

游 允 我 先做 初始化, 然后再批量地拉取 果。 有点儿像 数据 中的 cursor 。

游 会取某个 点的快照数据。 初始话之后索引上的任何 化会被它忽略。 它通 保存旧的数据文件来 个特性, 果就像保留初始化 的索引 ' ' 一 。

深度分 的代 根源是 果集全局排序, 如果去掉全局排序的特性的 果的成本就会很低。 游 用字段 _doc 来排序。 个指令 Elasticsearch 从 有 果的分片返回下一批 果。

用游 可以通 在 的 候 置参数 scroll 的 我 期望的游 的 期 。 游 的 期 会在 次做 的 候刷新, 所以 个 只需要足 理当前批的 果就可以了, 而不是 理 果的所有文 的所需 。 个 期 的参数很重要, 因 保持 个游 口需要消耗 源, 所以我 期望如果不再需要 源就 早点儿 放掉。 置 个超 能 Elasticsearch 在 后空 的 候自 放 部分 源。

```
GET /old_index/_search?scroll=1m ①
{
  "query": { "match_all": {}},
  "sort" : ["_doc"], ②
  "size": 1000
}
```

① 保持游 口一分 。

② 字 _doc 是最有效的排序 序。

个 的返回 果包括一个字段 _scroll_id, 它是一个base64 的 字符串 。 在我 能 字段 _scroll_id 到 _search/scroll 接口 取下一批 果 :

```
GET /_search/scroll
{
  "scroll": "1m", ①
  "scroll_id" :
  "cXVlcnlUaGVuRmV0Y2g7NTsxMDk5NDpkUmpriR2FjOFNhNnlCM1ZDMWpWYnRR0zEw0Tk10mRSamJHYWM4U2E2e
  UIzVkJxalZidFE7MTA50TM6ZFJqYkdhYzhTYTZ5QjNWQzFqVmJ0UTsxMTE5MDpBVUtwN2lx1FLZV8yRGVjWlI
  2QUVB0zEw0Tk20mRSamJHYWM4U2E2eUIzVkJxalZidFE7MDs="
}
```

① 注意再次 置游 期 一分 。

个游 返回的下一批 果。 尽管我 指定字段 size 的 1000, 我 有可能取到超 个 数量的文 。 当 的 候, 字段 size 作用于 个分片, 所以 个批次 返回的文 数量最大 size * number_of_primary_shards 。

NOTE 注意游 次返回一个新字段 `_scroll_id`。 次我 做下一次游 , 我 必 把前一次 返回的字段 `_scroll_id` 去。 当没有更多的 果返回的 时候, 我 就 理完所有匹配的文 了。

TIP 提示 : 某些官方的 Elasticsearch 客 端比如 Python 客 端 和 Perl 客 端 提供了一个功能易用的封装。

索引管理

我 已 看到 Elasticsearch 一个新的 用 得 , 不需要任何 先 或 置。 不 , 要不了多久 就会 始想要 化索引和搜索 程, 以便更好地 合 的特定用例。 些定制几乎 着索引和 型的方方面面, 在本章, 我 将介 管理索引和 型映射的 API 以及一些最重要的 置。

建一个索引

到目前 止, 我 已 通 索引一篇文 建了一个新的索引 。 个索引采用的是 的配置, 新的字段通 映射的方式被添加到 型映射。 在我 需要 个建立索引的 程做更多的控 制: 我 想要 保 个索引有数量 中的主分片, 并且在我 索引任何数据 之前 , 分析器和映射已 被建立好。

了 到 个目的, 我 需要手 建索引, 在 求体里面 入 置或 型映射, 如下所示 :

```
PUT /my_index
{
  "settings": { ... any settings ... },
  "mappings": {
    "type_one": { ... any mappings ... },
    "type_two": { ... any mappings ... },
    ...
  }
}
```

如果 想禁止自 建索引, 可以通 在 config/elasticsearch.yml 的 个 点下添加下面的配置 :

```
action.auto_create_index: false
```

NOTE 我 会在之后 用 索引模板 来 配置 自 建索引。 在索引日志数据的 时候尤其有用 : 将日志数据索引在一个以日期 尾命名的索引上, 子夜 分, 一个 配置 的新索引将会自 行 建。

除一个索引

用以下的 求来 除索引:

```
DELETE /my_index
```

也可以 除多个索引：

```
DELETE /index_one,index_two  
DELETE /index_*
```

甚至可以 除全部索引：

```
DELETE/_all  
DELETE/*
```

一些人来，能用一个命令来除所有数据可能会致可怕的后果。如果想要避免意外的大量除，可以在的 `elasticsearch.yml` 做如下配置：

NOTE `action.destructive_requires_name: true`

个置使除只限于特定名称指向的数据，而不允通指定 `_all` 或通配符来除指定索引。同可以通 [Cluster State API](#) 的更新个置。

索引置

可以通修改配置来自定索引行，配置参照[索引模](#)

TIP Elasticsearch 提供了化好的配置。除非理解些配置的作用并且知道什要去修改，否不要随意修改。

下面是 个最重要的置：

`number_of_shards`

个索引的主分片数，是 5。个配置在索引建后不能修改。

`number_of_replicas`

个主分片的副本数，是 1。于活的索引，个配置可以随修改。

例如，我 可以 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，我可以用 `update-index-settings` API 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 置是 `analysis` 部分，用来配置已存在的分析器或 建新的自定 分析器。

在 [分析与分析器](#)，我 介 了一些内置的分析器，用于将全文字符串 合搜索的倒排索引。

`standard` 分 析器是用于全文字段的 分析器，于大部分西方 系来 是一个不 的 。 它包括了以下几点：

- `standard` 分 器，通 界分割 入的文本。
- `standard` 元 器，目的是整理分 器触 的 元（但是目前什 都没做）。
- `lowercase` 元 器， 所有的 元 小写。
- `stop` 元 器， 除停用 — 搜索相 性影 不大的常用 ，如 `a`, `the`, `and`, `is`。

情况下，停用 器是被禁用的。如需 用它， 可以通 建一个基于 `standard` 分析器的自定 分析器并 置 `stopwords` 参数。 可以 分析器提供一个停用 列表，或者告知使用一个基于特定 言的 定 停用 列表。

在下面的例子中，我 建了一个新的分析器，叫做 `es_std`， 并使用 定 的西班牙 停用 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 存在于我 定 的 `spanish_docs` 索引中。 了使用 `analyze` API 来 它 行 ，我 必 使用特定的索引名：

```
GET /spanish_docs/_analyze?analyzer=es_std
El veloz zorro marrón
```

化的 果 示西班牙 停用 El 已被正 的移除：

```
{
  "tokens": [
    { "token": "veloz", "position": 2 },
    { "token": "zorro", "position": 3 },
    { "token": "marrón", "position": 4 }
  ]
}
```

自定 分析器

然Elasticsearch 有一些 成的分析器，然而在分析器上Elasticsearch真正的 大之 在于， 可以通在一个 合 的特定数据的 置之中 合字符 器、分 器、 元 器来 建自定 的分析器。

在 [分析与分析器](#) 我 ，一个 分析器 就是在一个包里面 合了三 函数的一个包装器， 三 函数按照序被 行：

字符 器

字符 器 用来 整理 一个尚未被分 的字符串。例如，如果我 的文本是 HTML格式的，它会包含像 <code><p></code> 或者 <code><div></code> 的HTML ， 些 是我 不想索引的。我 可以使用 [html清除](https://www.elastic.co/guide/en/elasticsearch/reference/master/analysis-htmlstrip-charfilter.html) 字符 器 来移除掉所有的HTML，并且像把 <code>Á</code> 相 的Unicode字符 <code>Á</code> ， HTML 体。

一个分析器可能有0个或者多个字符 器。

分 器

一个分析器 必 有一个唯一的分 器。 分 器把字符串分解成 个 条或者 元。 **准 分 器** 把一个字符串根据 界分解成 个 条，并且移除掉大部分的点符号，然而 有其他不同行 的分 器存在。

例如， **分 器** 完整地 出 接收到的同 的字符串，并不做任何分 。 **空格 分 器** 只根据空格分割文本。**正 分 器** 根据匹配正 表 式来分割文本。

元 器

分 ，作 果的 元流 会按照指定的 序通 指定的 元 器。

元 器可以修改、添加或者移除 元。我 已 提到 **lowercase** 和 **stop** 器，但是在 Elasticsearch 里面 有很多可供 的 元 器。 干 器 把 制 干。 **ascii_folding** 器移除 音符，把一个像 "très" 的 "tres"。 **ngram** 和 **edge_ngram** 元 器可以 生 合用于部分匹配或者自 全的 元。

在深入搜索，我了在里使用，以及使用分器和器。但是首先，我需要解一下建自定的分析器。

建一个自定 分析器

和我之前配置 `es_std` 分析器一，我可以 `analysis` 下的相位置字符器、分器和元器：

```
PUT /my_index
{
  "settings": {
    "analysis": {
      "char_filter": { ... custom character filters ... },
      "tokenizer": { ... custom tokenizers ... },
      "filter": { ... custom token filters ... },
      "analyzer": { ... custom analyzers ... }
    }
  }
}
```

作示，我一起来建一个自定分析器，个分析器可以做到下面的一些事：

1. 使用 `html` 清除字符器移除HTML部分。
2. 使用一个自定的映射字符器把 `&` 替 `"` 和 `"`：

```
"char_filter": {
  "&_to_and": {
    "type": "mapping",
    "mappings": [ "&=> and "]
  }
}
```

3. 使用准分器分。
4. 小写条，使用小写器理。
5. 使用自定停止器移除自定的停止列表中包含的：

```
"filter": {
  "my_stopwords": {
    "type": "stop",
    "stopwords": [ "the", "a" ]
  }
}
```

我的分析器定用我之前已置好的自定器合了已定好的分器和器：

```
"analyzer": {  
    "my_analyzer": {  
        "type": "custom",  
        "char_filter": [ "html_strip", "&_to_and" ],  
        "tokenizer": "standard",  
        "filter": [ "lowercase", "my_stopwords" ]  
    }  
}
```

起来，完整的 **建索引** 求看起来 像：

```
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 建以后，使用 **analyze** API 来 个新的分析器：

```
GET /my_index/_analyze?analyzer=my_analyzer  
The quick & brown fox
```

下面的 略 果展示出我 的分析器正在正 地 行：

```
{
  "tokens" : [
    { "token" : "quick", "position" : 2 },
    { "token" : "and", "position" : 3 },
    { "token" : "brown", "position" : 4 },
    { "token" : "fox", "position" : 5 }
  ]
}
```

个分析器 在是没有多大用 的，除非我 告诉 Elasticsearch 在 里用上它。我可以像下面 把一个分析器 用在一个 `string` 字段上：

```
PUT /my_index/_mapping/my_type
{
  "properties": {
    "title": {
      "type": "string",
      "analyzer": "my_analyzer"
    }
  }
}
```

型和映射

`型` 在 Elasticsearch 中表示一 相似的文 。 `型`由 `名称` ;比如 `user` 或 `blogpost` ;和 `映射` 成。

`映射`， 就像数据 中的 `schema` ， 描述了文 可能具有的字段或 `属性` 、 个字段的数据 型;比如 `string`, `integer` 或 `date`;以及Lucene是如何索引和存 些字段的。

型可以很好的抽象 分相似但不相同的数据。但由于 Lucene 的 理方式， 型的使用有些限制。

Lucene 如何 理文

在 Lucene 中，一个文 由一 的 成。 个字段都可以有多个 ， 但至少要有一个 。 似的，一个字符串可以通 分析 程 化 多个 。Lucene 不 心 些 是字符串、数字或日期—所有的 都被当做 不透明字 。

当我 在 Lucene 中索引一个文 ， 个字段的 都被添加到相 字段的倒排索引中。 也可以将未理的原始数据存 起来，以便 些原始数据在之后也可以被 索到。

型是如何 的

Elasticsearch 型是以 Lucene 理文 的 个方式 基 来 的。一个索引可以有多个 型， 些 型的文 可以存 在相同的索引中。

Lucene 没有文 型的概念， 个文 的 型名被存 在一个叫 `_type` 的元数据字段上。 当我 要 索某个 型的文 ， Elasticsearch 通 在 `_type` 字段上使用 器限制只返回 个 型的文 。

Lucene 也没有映射的概念。 映射是 Elasticsearch 将 JSON 文 映射 成 Lucene 需要的扁平化数据的方式。

例如，在 `user` 型中， `name` 字段的映射可以声明 个字段是 `string` 型，并且它的 被索引到名叫 `name` 的倒排索引之前，需要通 `whitespace` 分 器分析：

```
"name": {  
    "type": "string",  
    "analyzer": "whitespace"  
}
```

避免 型陷

致了一个有趣的思想：如果有 个不同的 型， 个 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 什 情况？

回答是，Elasticsearch 不会允 定 个映射。当 配置 个映射 ， 将会出 常。

回答是， 个 Lucene 索引中的所有字段都包含一个 一的、扁平的模式。一个特定字段可以映射成 `string` 型也可以是 `number` 型，但是不能 者兼具。因 型是 Elasticsearch 添加的 于 Lucene 的 外机制（以元数据 `_type` 字段的形式），在 Elasticsearch 中的所有 型最 都共享相同的映射。

以 `data` 索引中 型的映射 例：

```
{
  "data": {
    "mappings": {
      "people": {
        "properties": {
          "name": {
            "type": "string",
          },
          "address": {
            "type": "string"
          }
        }
      },
      "transactions": {
        "properties": {
          "timestamp": {
            "type": "date",
            "format": "strict_date_optional_time"
          },
          "message": {
            "type": "string"
          }
        }
      }
    }
  }
}
```

个型定一个字段 (分是 "name"/"address" 和 "timestamp"/"message")。它看起来是相互独立的，但在后台 Lucene 将建一个映射，如：

```
{
  "data": {
    "mappings": {
      "_type": {
        "type": "string",
        "index": "not_analyzed"
      },
      "name": {
        "type": "string"
      }
    },
    "address": {
      "type": "string"
    }
  },
  "timestamp": {
    "type": "long"
  }
}
}
```

注：不是真有效的映射法，只是用于演示

于整个索引，映射在本上被扁平化成一个一的、全局的模式。就是什一个型不能定冲突的字段：当映射被扁平化，Lucene 不知道如何去理。

型

那，个的 是什？技上，多个型可以在相同的索引中存在，只要它的字段不冲突（要因字段是互独占模式，要因它共享相同的字段）。

重要的一点是：型可以很好的区分同一个集合中的不同分。在不同的分中数据的整体模式是相同的（或相似的）。

型不 合 完全不同型的数据 。如果 个型的字段集是互不相同的，就意味着索引中将有一半的数据是空的（字段将是稀疏的），最将致性能。在情况下，最好是使用 个 独的索引。

：

- 正：将 kitchen 和 lawn-care 型放在 products 索引中，因型基本上是相同的模式
- : 将 products 和 logs 型放在 data 索引中，因型互不相同。将它放在不同的索引中。

根 象

映射的最高一 被称 根 象，它可能包含下面几 ：

- 一个 `properties` 点，列出了文 中可能包含的 个字段的映射
- 各 元数据字段，它 都以一个下 ，例如 `_type`、`_id` 和 `_source`
- 置 ，控制如何 理新的字段，例如 `analyzer`、`dynamic_date_formats` 和 `dynamic_templates`
- 其他 置，可以同 用在根 象和其他 `object` 型的字段上，例如 `enabled`、`dynamic` 和 `include_in_all`

属性

我 已 在 核心 域 型 和 核心域 型 章 中介 文 字段和属性的三个最重要的 置：

`type`

字段的数据 型，例如 `string` 或 `date`

`index`

字段是否 当被当成全文来搜索（ `analyzed` ），或被当成一个准 的（ `not_analyzed` ），是完全不可被搜索（ `no` ）

`analyzer`

定在索引和搜索 全文字段使用的 `analyzer`

我 将在本 的后 部分 其他字段 型，例如 `ip`、`geo_point` 和 `geo_shape`。

元数据: `_source` 字段

地，Elasticsearch 在 `_source` 字段存 代表文 体的JSON字符串。和所有被存 的字段一，`_source` 字段在被写入磁 之前先会被 。

个字段的存 几乎 是我 想要的，因 它意味着下面的 些：

- 搜索 果包括了整个可用的文 ——不需要 外的从 一个的数据 来取文 。
- 如果没有 `_source` 字段，部分 `update` 求不会生效。
- 当 的映射改 ， 需要重新索引 的数据，有了`_source`字段 可以直接从Elasticsearch 做，而不必从 一个(通常是速度更慢的) 数据 取回 的所有文 。
- 当 不需要看到整个文 ， 个字段可以从 `_source` 字段提取和通 `get` 或者 `search` 求返回。
- 句更加 ， 因 可以直接看到 个文 包括什 ，而不是从一列id 它 的内容。

然而，存 `_source` 字段的 要使用磁 空 。如果上面的原因 来 没有一个是重要的，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，可以通过在请求体中指定 `_source` 参数，来达到只取特定的字段的效果：

```
GET /_search
{
  "query": { "match_all": {}},
  "_source": [ "title", "created" ]
}
```

这些字段的会从 `_source` 字段被提取和返回，而不是返回整个 `_source`。

Stored Fields 被存字段

之后的索，除了索引一个字段的，可以存原始字段。有Lucene使用背景的用被存字段来他想要在搜索结果里面返回的字段。事实上，`_source`字段就是一个被存的字段。

在Elasticsearch中，文的个字段置存的做法通常不是最好的。整个文已被存 `_source` 字段。使用 `_source` 参数提取需要的字段是更好的。

元数据: `_all` 字段

在量搜索中，我介了 `_all` 字段：一个把其它字段当作一个大字符串来索引的特殊字段。`query_string` 子句(搜索 `?q=john`)在没有指定字段使用 `_all` 字段。

`_all` 字段在新用的探索段，当不清楚文的最是比有用的。可以使用一个字段来做任何，并且有很大可能到需要的文：

```
GET /_search
{
  "match": {
    "_all": "john smith marketing"
  }
}
```

随着用的展，搜索需求得更加明，会自己越来越少使用 `_all` 字段。`_all` 字段是搜索的急之策。通过指定字段，的更加活、大，也可以相性最高的搜索果行更粒度的控制。

NOTE

`relevance algorithm` 考的一个最重要的原是字段的度：字段越短越重要。在短的 `title` 字段中出的短可能比在的 `content` 字段中出的短更加重要。字段度的区在 `_all` 字段中不会出。

如果不再需要 `_all` 字段，可以通下面的映射来禁用：

```
PUT /my_index/_mapping/my_type
{
  "my_type": {
    "_all": { "enabled": false }
  }
}
```

通 `include_in_all` 置来逐个控制字段是否要包含在 `_all` 字段中，是 `true`。在一个象(或根象)上置 `include_in_all` 可以修改一个象中的所有字段的行。

可能想要保留 `_all` 字段作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`。相较于完全禁用 `_all` 字段，可以所有字段禁用 `include_in_all`，在的字段上用：

```
PUT /my_index/my_type/_mapping
{
  "my_type": {
    "include_in_all": false,
    "properties": {
      "title": {
        "type": "string",
        "include_in_all": true
      },
      ...
    }
  }
}
```

住，`_all` 字段是一个分的 `string` 字段。它使用分器来分析它的，不管个原本所在字段指定的分器。就像所有 `string` 字段，可以配置 `_all` 字段使用的分器：

```
PUT /my_index/my_type/_mapping
{
  "my_type": {
    "_all": { "analyzer": "whitespace" }
  }
}
```

元数据：文

文 与四个元数据字段相：

`_id`

文 的 ID 字符串

`_type`

文 的 型名

`_index`

文 所在的索引

`_uid`

`_type` 和 `_id` 接在一起 造成 `type#id`

情况下，`_uid` 字段是被存（可取回）和索引（可搜索）的。`_type` 字段被索引但是没有存，`_id` 和 `_index` 字段既没有被索引也没有被存，意味着它并不是真存在的。

尽管如此，然可以像真字段一样使用 `_id` 字段。Elasticsearch 使用 `_uid` 字段来派生出 `_id`。然可以修改些字段的 `index` 和 `store` 置，但是基本上不需要做。

映射

当 Elasticsearch 遇到文 中以前未遇到的字段，它用 `dynamic mapping` 来定字段的数据型并自把新的字段添加到型映射。

有 是想要的行 有 又不希望 。通常没有人知道以后会有什新字段加到文，但是又希望些字段被自的索引。也只想忽略它。如果Elasticsearch是作重要的数据存，可能就会期望遇到新字段就会出常，能及。

幸的是可以用 `dynamic` 配置来控制行，可接受的如下：

`true`

添加新的字段—省

`false`

忽略新的字段

`strict`

如果遇到新字段出常

配置参数 `dynamic` 可以用在根 `object` 或任何 `object`型的字段上。可以将 `dynamic` 的 `strict`，而只在指定的内部象中 它，例如：

```

PUT /my_index
{
  "mappings": {
    "my_type": {
      "dynamic": "strict", ①
      "properties": {
        "title": { "type": "string"}, 
        "stash": {
          "type": "object",
          "dynamic": true ②
        }
      }
    }
  }
}

```

① 如果遇到新字段，象 `my_type` 就会出常。

② 而内部 象 `stash` 遇到新字段就会建新字段。

使用上述 映射，可以 `stash` 象添加新的可 索的字段：

```

PUT /my_index/my_type/1
{
  "title": "This doc adds a new field",
  "stash": { "new_field": "Success!" }
}

```

但是 根 点 象 `my_type` 行同 的操作会失：

```

PUT /my_index/my_type/1
{
  "title": "This throws a StrictDynamicMappingException",
  "new_field": "Fail!"
}

```

NOTE

把 `dynamic` 置 `false` 一点儿也不会改 `_source` 的字段内容。`_source` 然包含被索引的整个JSON文。只是新的字段不会被加到映射中也不可搜索。

自定 映射

如果 想在 行 加新的字段，可能会 用 映射。然而，有 候， 映射 可能不太智能。幸 的是，我 可以通 置去自定 些 ，以便更好的 用于 的数据。

日期

当 Elasticsearch 遇到一个新的字符串字段，它会 个字段是否包含一个可 的日期，比如 `2014-`

01-01。如果它像日期，一个字段就会被作 `date` 型添加。否，它会被作 `string` 型添加。

有些时候一个行可能致一些。想象下，有如下的一个文：

```
{ "note": "2014-01-01" }
```

假是第一次 `note` 字段，它会被添加 `date` 字段。但是如果下一个文像：

```
{ "note": "Logged out" }
```

然不是一个日期，但已。一个字段已是一个日期型，一个 **不合法的日期** 将会造成一个常。

日期可以通过在根象上置 `date_detection` `false` 来：

```
PUT /my_index
{
  "mappings": {
    "my_type": {
      "date_detection": false
    }
  }
}
```

使用一个映射，字符串将始作 `string` 型。如果需要一个 `date` 字段，必手添加。

NOTE Elasticsearch 判断字符串日期的可以通 `dynamic_date_formats setting` 来置。

模板

使用 `dynamic_templates`，可以完全控制新生成字段的映射。甚至可以通字段名称或数据型来用不同的映射。

个模板都有一个名称，可以用来描述个模板的用途，一个 `mapping` 来指定映射使用，以及至少一个参数(如 `match`)来定个模板用于个字段。

模板按照序来；第一个匹配的模板会被用。例如，我 `string` 型字段定个模板：

- `es`：以 `_es` 尾的字段名需要使用 `spanish` 分器。
- `en`：所有其他字段使用 `english` 分器。

我将 `es` 模板放在第一位，因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", ①
          "match_mapping_type": "string",
          "mapping": {
            "type": "string",
            "analyzer": "spanish"
          }
        }},
        { "en": {
          "match": "*", ②
          "match_mapping_type": "string",
          "mapping": {
            "type": "string",
            "analyzer": "english"
          }
        }}
      ]
    }
  }
}

```

① 匹配字段名以 `_es` 尾的字段。

② 匹配其他所有字符串 型字段。

`match_mapping_type` 允用模板到特定 型的字段上，就像有 准 映射 的一， (例如 `string` 或 `long`)。

`match` 参数只匹配字段名称， `path_match` 参数匹配字段在 象上的完整路径，所以 `address.*.name` 将匹配 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 映射文 。

省映射

通常，一个索引中的所有 型共享相同的字段和 置。 `default` 映射更加方便地指定通用 置，而不是

次 建新 型 都要重 置。 `default` 映射是新 型的模板。在 置 `default` 映射之后 建的所有 型都将 用 些 省的 置，除非 型在自己的映射中明 覆 些 置。

例如，我 可以使用 `default` 映射 所有的 型禁用 `_all` 字段，而只在 `blog` 型 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](#) 的好方法。

重新索引 的数据

尽管可以 加新的 型到索引中，或者 加新的字段到 型中，但是不能添加新的分析器或者 有的字段 做改 。如果 那 做的 ， 果就是那些已 被索引的数据就不正 ， 搜索也不能正常工作。

有数据的 改 最 的 法就是重新索引：用新的 置 建新的索引并把文 从旧的索引 制到新的索引。

字段 `_source` 的一个 点是在Elasticsearch中已 有整个文 。 不必从源数据中重建索引，而且那 通常比 慢。

了有效的重新索引所有在旧的索引中的文 ，用 `scroll` 从旧的索引 索批量文 ， 然后用 `bulk API` 把文 推送到新的索引中。

从Elasticsearch v2.3.0 始， [Reindex API](#) 被引入。它能 文 重建索引而不需要任何 件或外部工具。

批量重新索引

同 并行 行多个重建索引任 , 但是 然不希望 果有重 。正 的做法是按日期或者 的字段作 条件把大的重建索引分成小的任 :

```
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 会有 化, 希望新的索引中也包括那些新加的文 。那就可以 新加的文 做重 新索引, 但 是要用日期 字段 来匹配那些新加的文 。

索引 名和零停机

在前面提到的, 重建索引的 是必 更新 用中的索引名称。索引 名就是用来解决 个 的 !

索引 名 就像一个快捷方式或 接, 可以指向一个或多个索引, 也可以 任何一个需要索引名的 API来使用。 名 我 大的 活性, 允 我 做下面 些:

- 在 行的集群中可以无 的从一个索引切 到 一个索引
- 多个索引分 (例如, `last_three_months`)
- 索引的一个子集 建

在后面我 会 更多 于 名的使用。 在, 我 将解 使用 名在零停机下从旧索引切 到新索引 。

有 方式管理 名: `_alias` 用于 个操作, `_aliases` 用于 行多个原子 操作。

在本章中, 我 假 的 用有一个叫 `my_index` 的索引。事 上, `my_index` 是一个指向当前真 索引的 名。真 索引包含一个版本号: `my_index_v1`, `my_index_v2` 等等。

首先, 建索引 `my_index_v1`, 然后将 名 `my_index` 指向它 :

```
PUT /my_index_v1 ①
PUT /my_index_v1/_alias/my_index ②
```

① 建索引 `my_index_v1`。

② 置 名 `my_index` 指向 `my_index_v1`。

可以 一个 名指向 一个索引：

```
GET /*/_alias/my_index
```

或 些 名指向 个索引：

```
GET /my_index_v1/_alias/*
```

者都会返回下面的 果：

```
{
  "my_index_v1" : {
    "aliases" : {
      "my_index" : { }
    }
  }
}
```

然后，我 决定修改索引中一个字段的映射。当然，我 不能修改 存的映射，所以我 必 重新索引数据。首先，我 用新映射 建索引 `my_index_v2`：

```
PUT /my_index_v2
{
  "mappings": {
    "my_type": {
      "properties": {
        "tags": {
          "type": "string",
          "index": "not_analyzed"
        }
      }
    }
  }
}
```

然后我 将数据从 `my_index_v1` 索引到 `my_index_v2`，下面的 程在 [重新索引 的数据](#) 中已 描述。一旦我 定义 已 被正 地重索引了，我 就将 名指向新的索引。

一个 名可以指向多个索引，所以我 在添加 名到新索引的同 必 从旧的索引中 除它。 个操作需要原子化， 意味着我 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" }}
  ]
}

```

的 用已 在零停机的情况下从旧索引 移到新索引了。

即使 在的索引 已 很完美了，在生 境中， 是有可能需要做一些修改的。

TIP 做好准：在 的 用中使用 名而不是索引名。然后 就可以在任何 时候重建索引。 名的 很小， 广泛使用。

分片内部原理

在 集群内的原理， 我 介 了 分片， 并将它 描述成最小的 工作 元。但是究竟什 是一个分片，它是如何工作的？在 个章 ，我 回答以下 ：

- 什 搜索是近 的？
- 什 文 的 CRUD(建- 取-更新- 除)操作是 的？
- Elasticsearch 是 保 更新被持久化在断 也不 失数据？
- 什 除文 不会立刻 放空 ？
- `refresh`, `flush`, 和 `optimize` API 都做了什 ， 什 情况下 是用他 ？

最 的理解一个分片如何工作的方式是上一堂 史 。 我 将要 提供一个 近 搜索和分析的 分布式持久化数据存 需要解决的 。

内容警告

本章展示的 些信息 供 趣 。 了使用 Elasticsearch 并不需要理解和 所有的 。 个章 是 了了解工作机制，并且 了将来 需要 些信息 ，知道 些信息在 里。但是不要 被 些 所累。

使文本可被搜索

必 解决的第一个挑 是如何使文本可被搜索。 的数据 个字段存 个 ，但 全文 索并不 。文本字段中的 个 需要被搜索， 数据 意味着需要 个字段有索引多 (里指)的能力。

最好的支持 一个字段多个 需求的数据 是我 在 倒排索引 章 中介 的 倒排索引 。 倒排索引包含一个有序列表，列表包含所有文 出 的不重 个体，或称 ， 包含了它所有曾出 文 的列表。

Term	Doc 1	Doc 2	Doc 3	...
brown	X		X	...
fox	X	X	X	...
quick	X	X		...
the	X		X	...

NOTE 当倒排索引，我会到文引，因史原因，倒排索引被用来整个非化文本行引。Elasticsearch 中的文是有字段和的化 JSON 文。事上，在 JSON 文中，一个被索引的字段都有自己的倒排索引。

个倒排索引相比特定出的文列表，会包含更多其它信息。它会保存一个出的文数，在的文中的一个具体出的次数，在文中的序，个文的度，所有文的平均度，等等。些信息允 Elasticsearch 决定些比其它更重要，些文比其它文更重要，些内容在什 是相 性？中有描述。

了能期功能，倒排索引需要知道集合中的所有文，是需要到的。

早期的全文索会整个文集合建立一个很大的倒排索引并将其写入到磁。一旦新的索引就，旧的就会被其替，最近的化便可以被索到。

不性

倒排索引被写入磁后是不可改的：它永不会修改。不性有重要的：

- 不需要。如果从来不更新索引，就不需要担心多程同修改数据的。
- 一旦索引被入内核的文件系存，便会留在里，由于其不性。只要文件系存中有足够的空，那大部分求会直接求内存，而不会命中磁。提供了很大的性能提升。
- 其它存(像filter存)，在索引的生命周期内始有效。它不需要在次数据改被重建，因数据不会化。
- 写入个大的倒排索引允数据被，少磁I/O和需要被存到内存的索引的使用量。

当然，一个不的索引也有不好的地方。主要事是它是不可的！不能修改它。如果需要一个新的文可被搜索，需要重建整个索引。要一个索引所能包含的数据量造成了很大的限制，要索引可被更新的率造成了很大的限制。

更新索引

下一个需要被解决的是在保留不性的前提下倒排索引的更新？答案是：用更多的索引。

通常加新的充索引来反映新近的修改，而不是直接重写整个倒排索引。一个倒排索引都会被流到—从最早的始—完后再果行合并。

Elasticsearch 基于 Lucene，个 java 引入了按段搜索的概念。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，加了 *提交点* 的概念；一个列出了所有已知段的文件，就像在 [一个 Lucene 索引包含一个提交点和三个段](#) 中描的那。如 [一个 anchor="img-memory-segments"](#)

buffer">一个在内存 存中包含新文 的 Lucene 索引 所示，新的文 首先被添加到内存索引 存中，然后写入到一个基于磁 的段，如 在一次提交后，一个新的段被添加到提交点而且 存被清空。 所示。

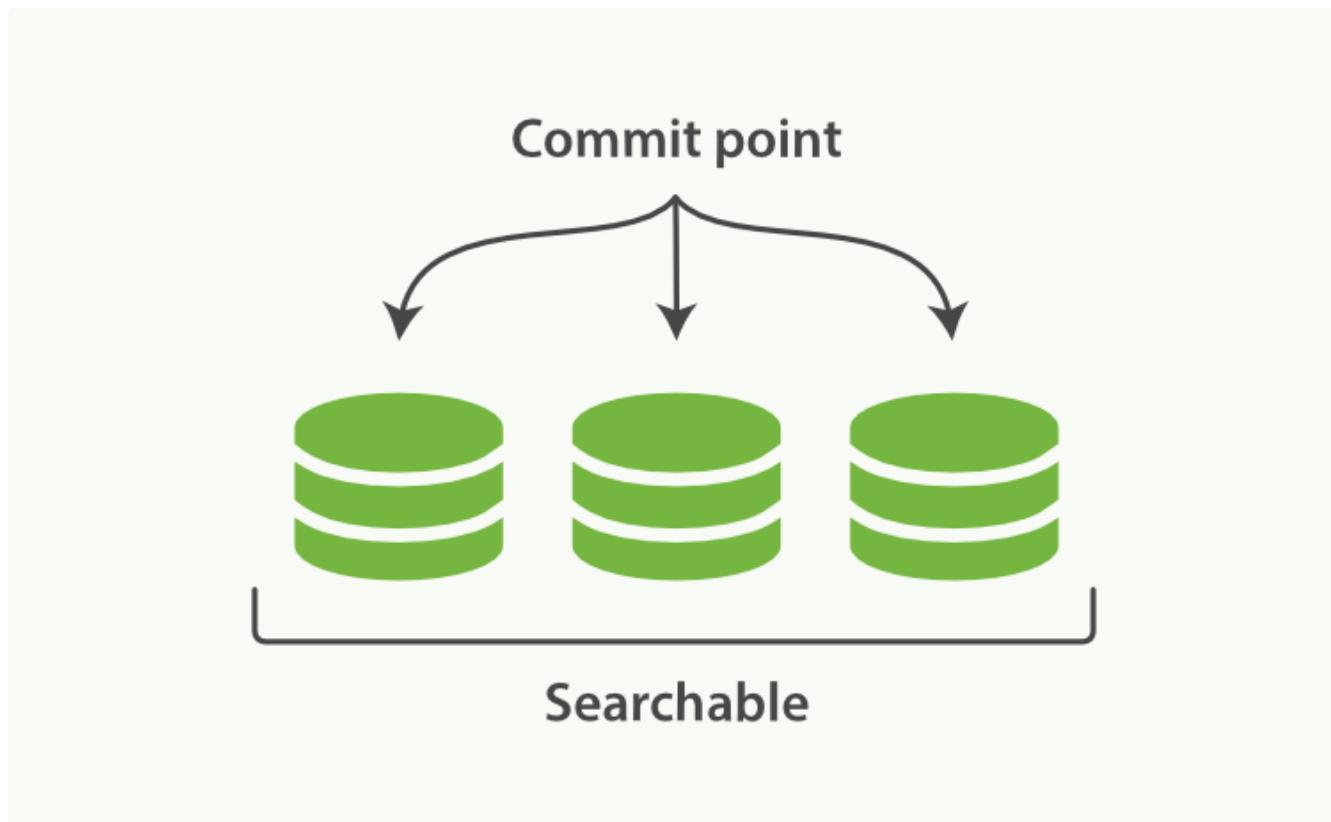


Figure 16. 一个 Lucene 索引包含一个提交点和三个段

索引与分片的比

被混 的概念是，一个 Lucene 索引 我 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 送 到 一个属于索引的分片(Lucene 索引)，然后像 行分布式 索 提到的那 ，合并 个分片的 果到一个全局的 果集。

逐段搜索会以如下流程 行工作：

1. 新文 被收集到内存索引 存， 一个在内存 存中包含新文 的 Lucene 索引 。
2. 不 地， 存被 提交：
 - 一个新的段—一个追加的倒排索引—被写入磁 。
 - 一个新的包含新段名字的 提交点 被写入磁 。
 - 磁 行 同 — 所有在文件系 存中等待的写入都刷新到磁 ，以 保它 被写入物理文件。
3. 新的段被 ， 它包含的文 可 以被搜索。
4. 内存 存被清空， 等待接收新的文 。

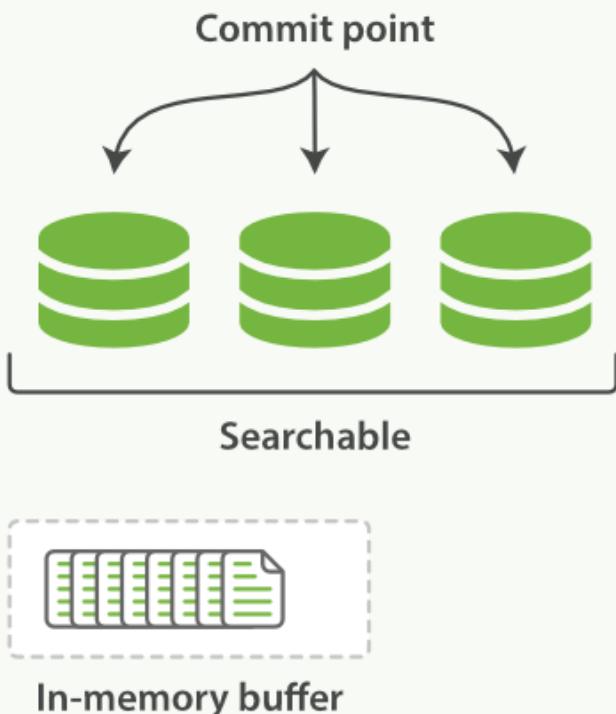


Figure 17. 一个在内存 存中包含新文 的 Lucene 索引

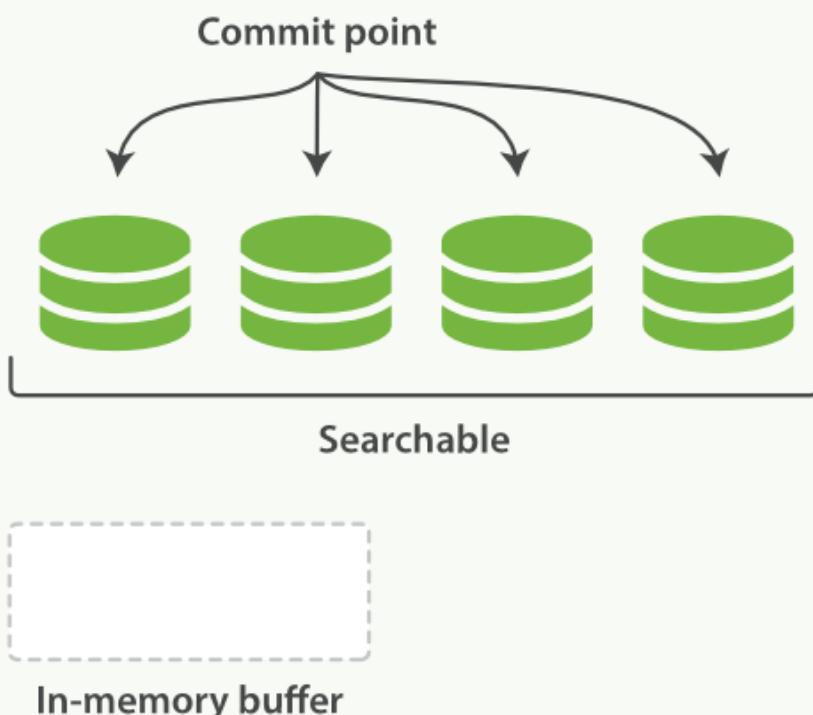


Figure 18. 在一次提交后，一个新的段被添加到提交点而且 存被清空。

当一个 被触 ，所有已知的段按 序被 。 会 所有段的 果 行聚合，以保 个 和 个文 的 都被准 算。 方式可以用相 低的成本将新文 添加到索引。

除和更新

段是不可改的，所以既不能从把文 从旧的段中移除，也不能修改旧的段来 行反映文 的更新。取而代之的是， 个提交点会包含一个 `.del` 文件，文件中会列出 些被 除文 的段信息。

当一个文 被“ 除” ，它 上只是在 `.del` 文件中被 除。一个被 除的文 然可以被匹配到，但它会在最 果被返回前从 果集中移除。

文 更新也是 似的操作方式：当一个文 被更新 ，旧版本文 被 除，文 的新版本被索引到一个新的段中。 可能 个版本的文 都会被一个 匹配到，但被 除的那个旧版本文 在 果集返回前就已被移除。

在 段合并，我 展示了一个被 除的文 是 被文件系 移除的。

近 搜索

随着按段 (per-segment) 搜索的 展，一个新的文 从索引到可被搜索的延 著降低了。新文 在几分 之内即可被 索，但 是不 快。

磁 在 里成 了瓶 。提交 (Committing) 一个新的段到磁 需要一个 `fsync` 来保段被物理性地写入磁 ， 在断 的 候就不会 失数据。 但是 `fsync` 操作代 很大； 如果次索引一个文 都去 行一次的 会造成很大的性能 。

我 需要的是一个更 量的方式来使一个文 可被搜索， 意味着 `fsync` 要从整个 程中被移除。

在Elasticsearch和磁 之 是文件系 存。 像之前描述的一 ， 在内存索引 冲区（ 在内存冲区中包含了新文 的 Lucene 索引 ）中的文 会被写入到一个新的段中（ 冲区的内容已被写入一个可被搜索的段中，但 没有 行提交 ）。 但是 里新段会被先写入到文件系 存— 一代 会比 低， 后再被刷新到磁 — 一代 比 高。不 只要文件已 在 存中， 就可以像其它文件一 被打 和 取了。

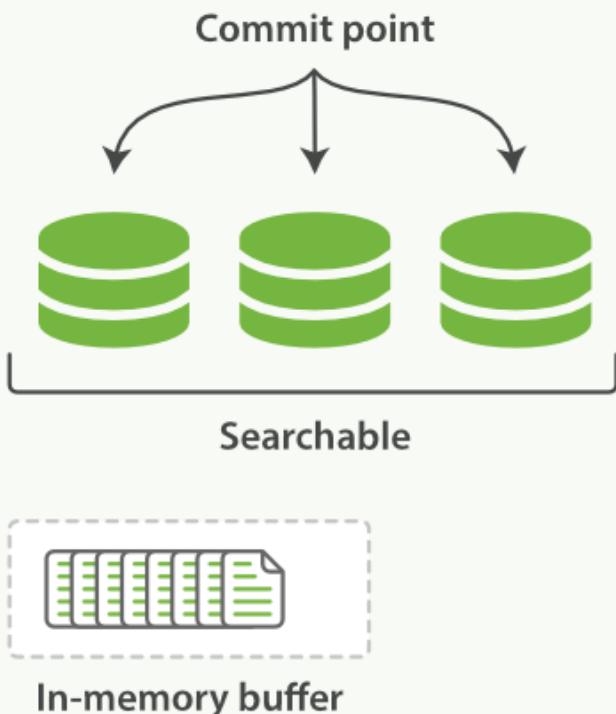


Figure 19. 在内存缓冲区中包含了新文档的Lucene索引

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便能搜索到。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。

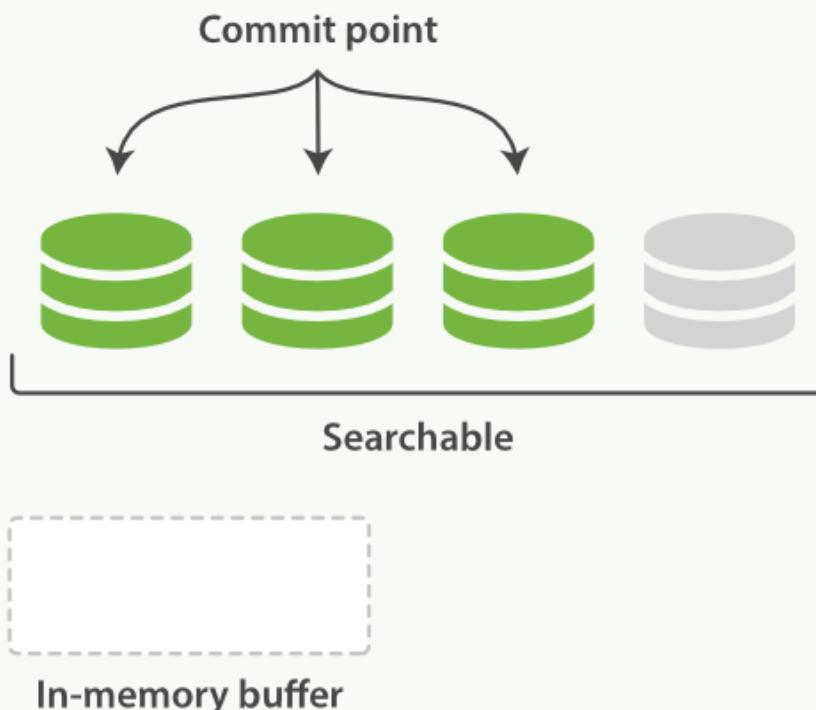


Figure 20. 缓冲区的内容已被写入一个可被搜索的段中，但没有进行提交

refresh API

在 Elasticsearch 中，写入和打一个新段的量的程叫做 `refresh`。情况下个分片会秒自刷新一次。就是什我 Elasticsearch 是近搜索：文的化并不是立即搜索可，但会在一秒之内可。

些行可能会新用造成困惑：他索引了一个文然后搜索它，但却没有搜到。个的解决法是用 `refresh` API 行一次手刷新：

```
POST /_refresh ①  
POST /blogs/_refresh ②
```

① 刷新（Refresh）所有的索引。

② 只刷新（Refresh）`blogs` 索引。

TIP 尽管刷新是比提交量很多的操作，它是会有性能。当写的时候，手刷新很有用，但是不要在生境下次索引一个文都去手刷新。相反，的用需要意到 Elasticsearch 的近的性，并接受它的不足。

并不是所有的情况都需要秒刷新。可能正在使用 Elasticsearch 索引大量的日志文件，可能想化索引速度而不是近搜索，可以通置 `refresh_interval`，降低个索引的刷新率：

```
PUT /my_logs  
{  
  "settings": {  
    "refresh_interval": "30s" ①  
  }  
}
```

① 30秒刷新 `my_logs` 索引。

`refresh_interval`可以在既存索引上行更新。在生境中，当正在建立一个大的新索引，可以先自刷新，待始使用索引，再把它回来：

```
PUT /my_logs/_settings  
{ "refresh_interval": -1 } ①  
  
PUT /my_logs/_settings  
{ "refresh_interval": "1s" } ②
```

① 自刷新。

② 秒自刷新。

CAUTION `refresh_interval` 需要一个持，例如 `1s` (1秒) 或 `2m` (2分钟)。一个 `1` 表示的是 1 秒 -- 无疑会使的集群陷入。

持久化 更

如果没有用 `fsync` 把数据从文件系 存刷 (flush) 到硬 盘，我 不能保 证 数据在断 甚至是程序正常退出之后依然存在。为了保 持 Elasticsearch 的可 用 性，需要 保数据 化被持久化到磁 盘。

在 [更新索引](#)，我 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 或重新打 一个索引的 程中使用 个提交点来判断 些段隶属于当前分片。

即使通 秒刷新 (refresh) 了近 搜索，我 然需要 常 行完整提交来 保能从失 中恢 。但在 次提交之 生 化的文 ？我 也不希望 失掉 些数据。

Elasticsearch 加了一个 `translog`，或者叫事 日志，在 一次 Elasticsearch 行操作 均 行了日志 。通 `translog`，整个流程看起来是下面 ：

1. 一个文 被索引之后，就会被添加到内存 冲区，并且 追加到了 `translog`，正如 新的文 被添加到内存 冲区并且被追加到了事 日志 描述的一 。

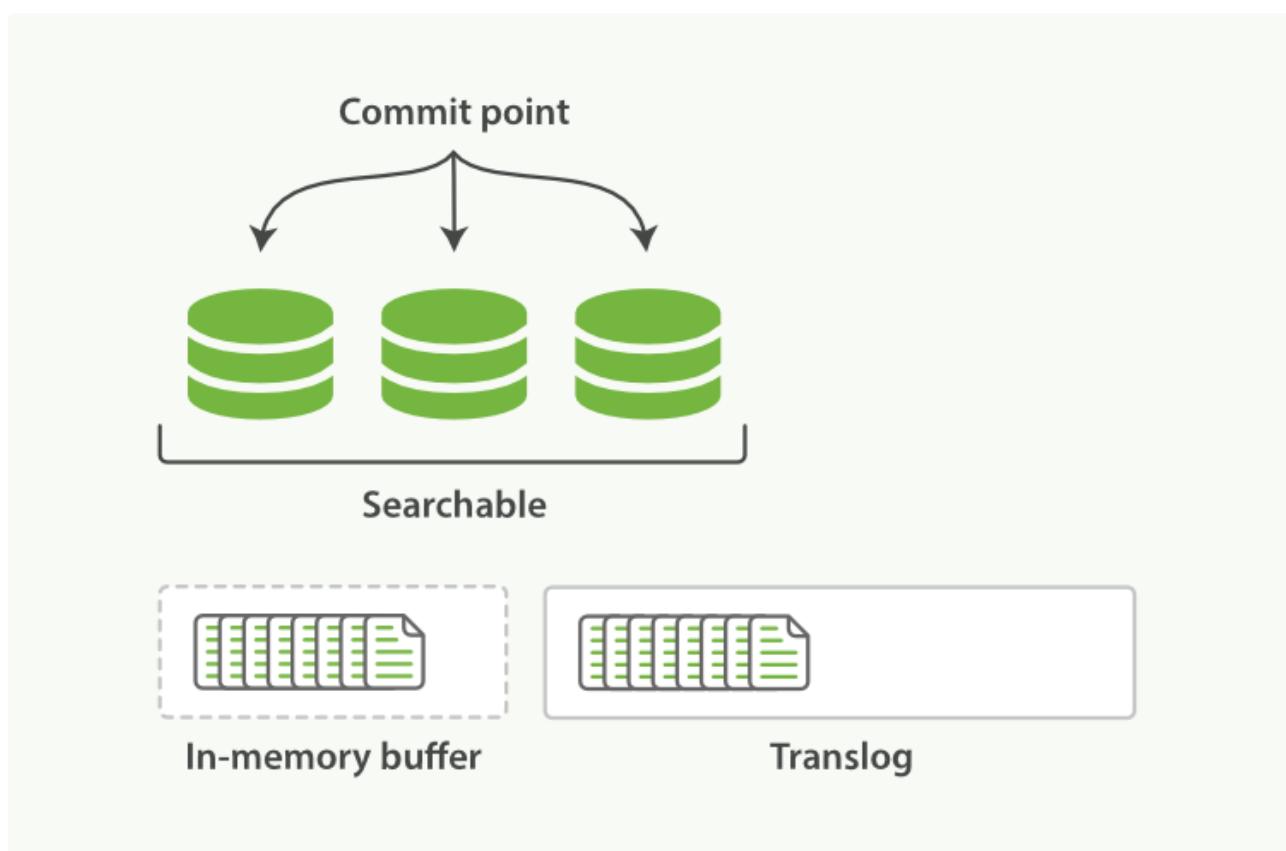


Figure 21. 新的文 被添加到内存 冲区并且被追加到了事 日志

2. 刷新 (refresh) 使分片 于 刷新 (refresh) 完成后，存被清空但是事 日志不会 描述的状 ，分片 秒被刷新 (refresh) 一次：

- 些在内存 冲区的文 被写入到一个新的段中，且没有 行 `fsync` 操作。
- 个段被打 ，使其可被搜索。
- 内存 冲区被清空。

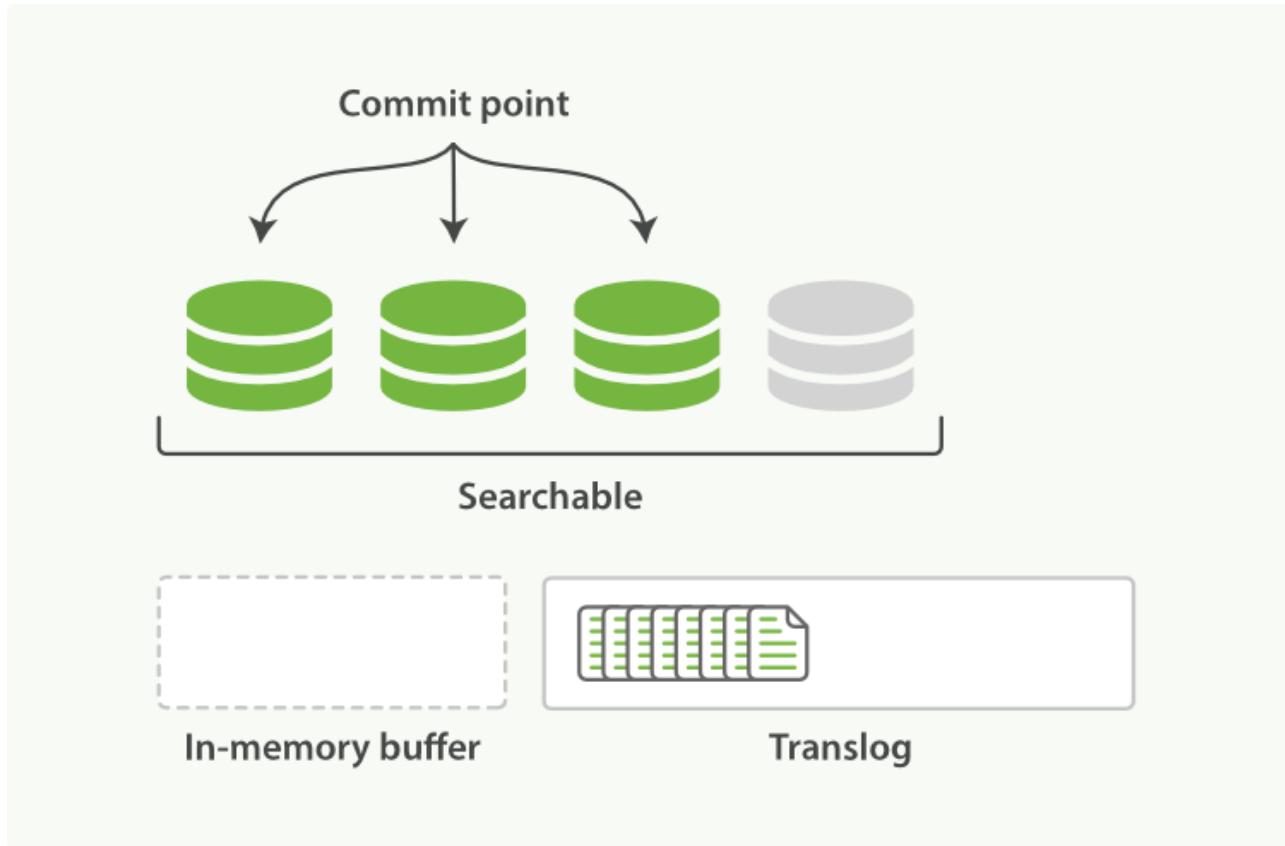


Figure 22. 刷新 (refresh) 完成后，存被清空但是事 日志不会

3. 个 程 工作，更多的文 被添加到内存 冲区和追加到事 日志（ 事 日志不断 累文 ）。

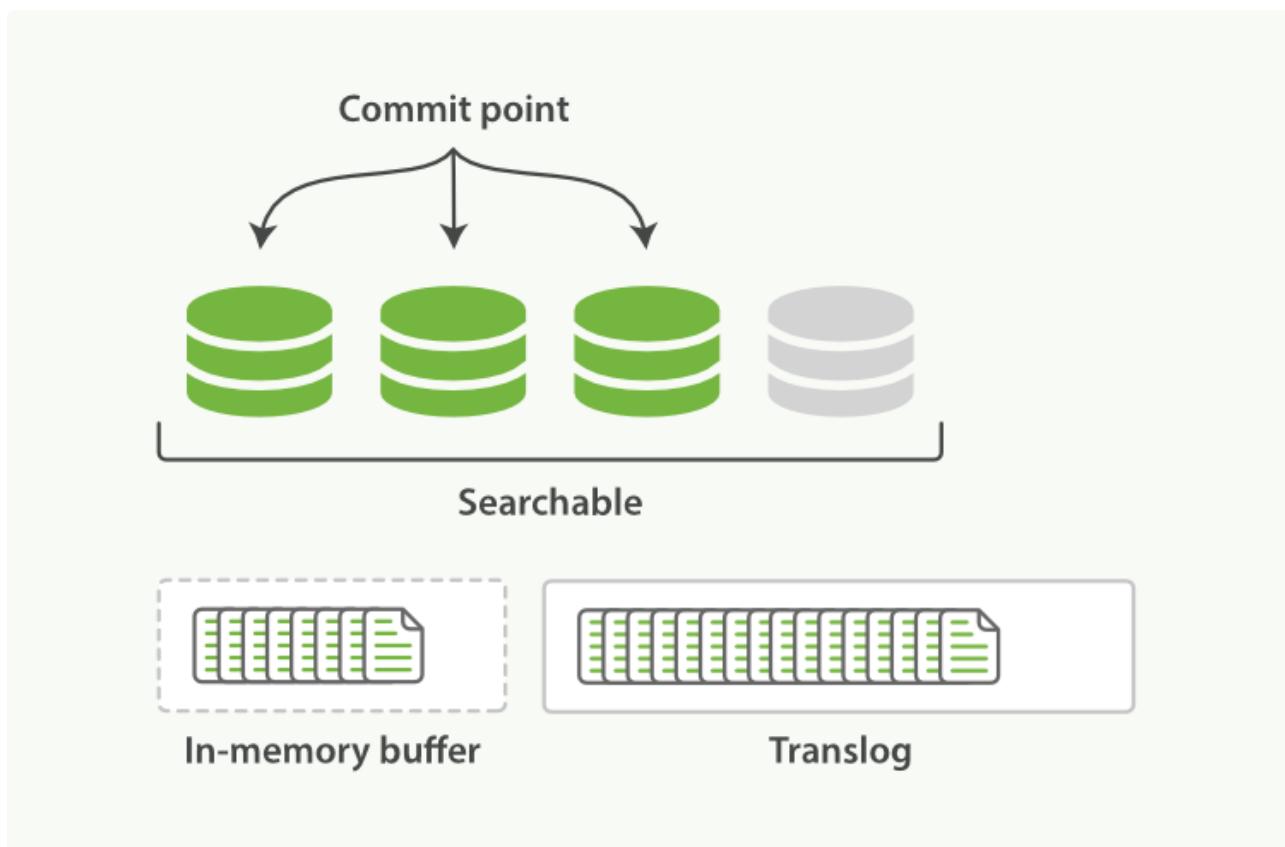


Figure 23. 事 日志不断 累文

4. 隔一段 —例如 translog 得越来越大—索引被刷新 (flush)；一个新的 translog 被建，并且一个全量提交被 行（ 在刷新 (flush) 之后，段被全量提交， 并且事 日志被清空 ）：

- 所有在内存 冲区的文 都被写入一个新的段。
- 冲区被清空。
- 一个提交点被写入硬 。
- 文件系 存通 `fsync` 被刷新 (flush) 。
- 老的 translog 被 除。

translog 提供所有 没有被刷到磁 的操作的一个持久化 。当 Elasticsearch 的 候， 它会从磁 中使用最后一个提交点去恢 已知的段，并且会重放 translog 中所有在最后一次提交后 生的 更操作。

translog 也被用来提供 CRUD 。当 着通 ID 、更新、 除一个文 ， 它会在 从相 的段中 索之前， 首先 translog 任何最近的 更。 意味着它 是能 地 取到文 的最新版本。

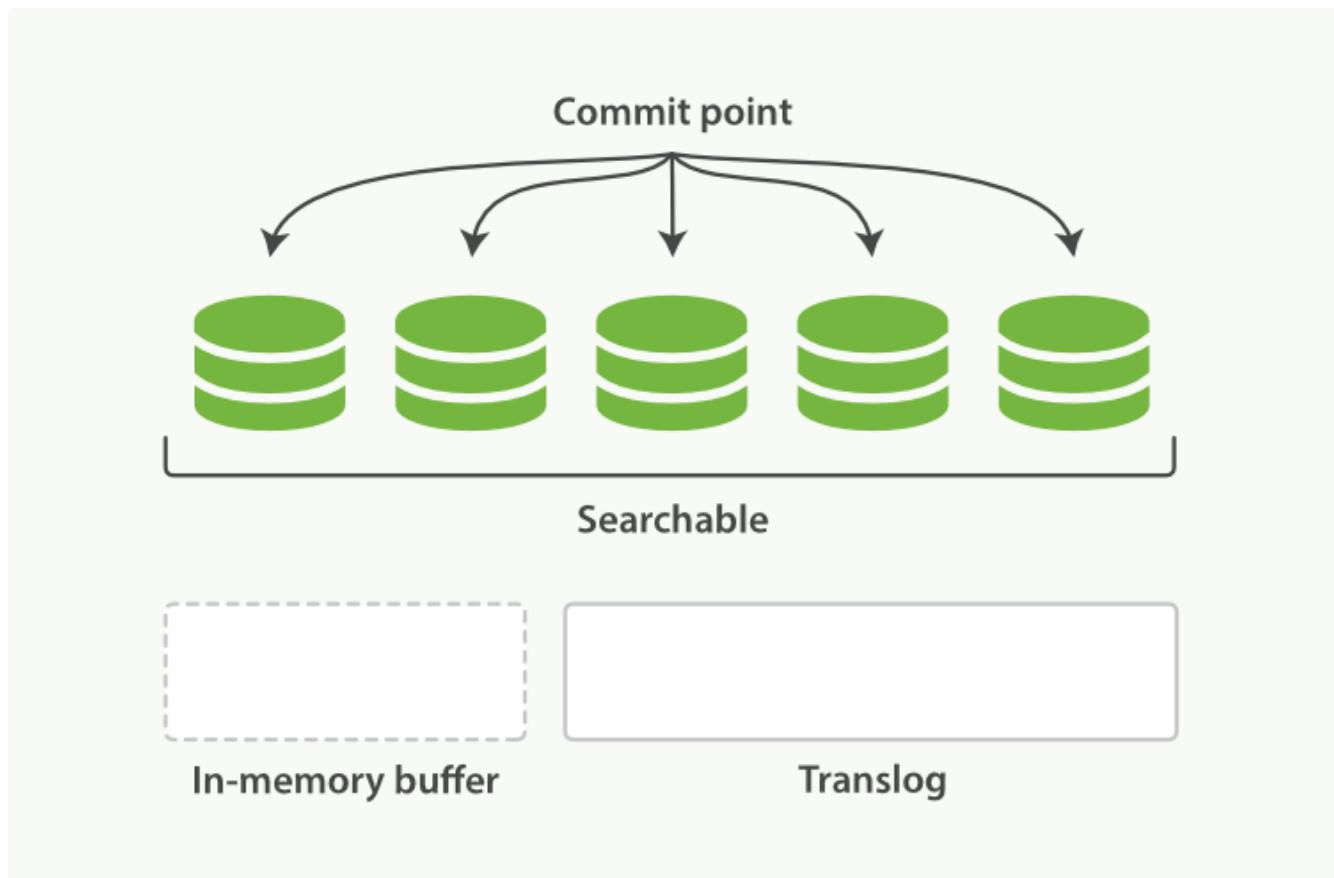


Figure 24. 在刷新 (flush) 之后，段被全量提交，并且事 日志被清空

flush API

个 行一个提交并且截断 translog 的行 在 Elasticsearch 被称作一次 `flush` 。 分片 30分 被自 刷新 (flush) ， 或者在 translog 太大的 候也会刷新。 看 `translog 文` 来 置， 它可以用来 控制 些 ：

`flush API` 可以被用来 行一个手工的刷新 (flush) :

```
POST /blogs/_flush ①
```

```
POST /_flush?wait_for_ongoing ②
```

① 刷新 (flush) `blogs` 索引。

② 刷新 (flush) 所有的索引并且等待所有刷新在返回前完成。

很少需要自己手动一个的 `flush` 操作；通常情况下，自动刷新就足够了。

就是，在重建点或索引之前执行 `flush` 有益于的索引。当 Elasticsearch 做快照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。引出了一个问题：Translog 有多安全？

在文件被 `fsync` 到磁盘前，被写入的文件在重写之后就会丢失。translog 是 5 秒被 `fsync` 刷新到硬盘，或者在一次写入完成之后（e.g. `index`, `delete`, `update`, `bulk`）。一个请求在主分片和副本分片都会产生。最坏情况下，意味着在整个请求被 `fsync` 到主分片和副本分片的 translog 之前，客户端不会得到一个 200 OK 响应。

在一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 入，它在一次请求中平均处理了大量文档）。

但是对于一些大容量的偶发性几秒数据丢失并不严重的集群，使用自动的 `fsync` 是比较有益的。比如，写入的数据被存到内存中，再每 5 秒执行一次 `fsync`。

一个请求可以通过设置 `durability` 参数为 `async` 来使用：

```
PUT /my_index/_settings
{
  "index.translog.durability": "async",
  "index.translog.sync_interval": "5s"
}
```

一个请求可以对索引独占，并且可以进行修改。如果决定使用自动的 translog，需要保证在发生 crash，丢失掉 `sync_interval` 范围内的数据也无所损失。在决定前知道这个特性。

如果不希望一个请求的后果，最好是使用 `request` 的参数（`"index.translog.durability": "request"`）来避免数据丢失。

段合并

由于自动生成流程每秒会建立一个新的段，会致短时间内段数量暴增。而段数目太多会带来很大的麻烦。一个段都会消耗文件句柄、内存和 CPU 行周期。更重要的是，一个搜索请求都需要流过多个段；所以段越多，搜索也就越慢。

Elasticsearch 通过后台进行段合并来解决这个问题。小的段被合并到大的段，然后一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件从文件系统中清除。被删除的文件（或被更新文件的旧版本）不会被拷贝到新的大段中。

段合并不需要做任何事。索引和搜索会自行进行。这个流程像在一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以供搜索使用。
- 2、合并进程将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。并不会中断索引和搜索。

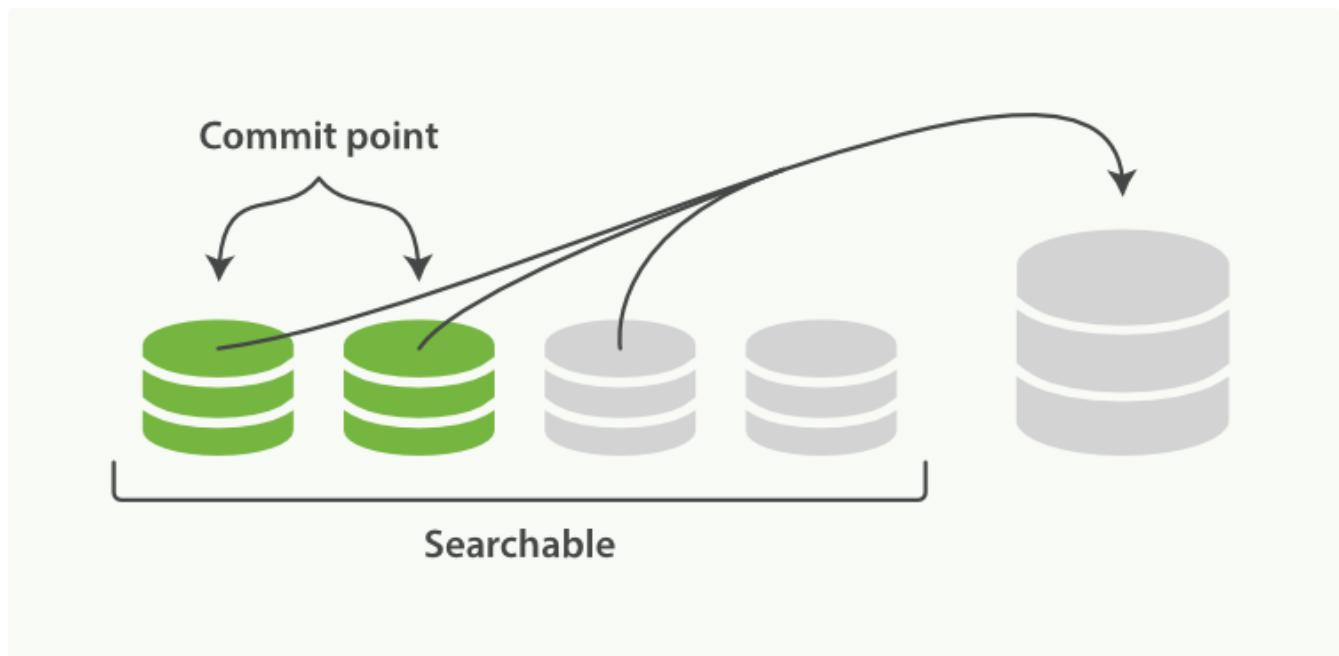


Figure 25. 一个提交了的段和一个未提交的段正在被合并到一个更大的段

- 3、一旦合并结束，老的段被删除。明合并完成的活：

- 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小的段的新提交点。
- 新的段被打以供搜索。
- 老的段被删除。

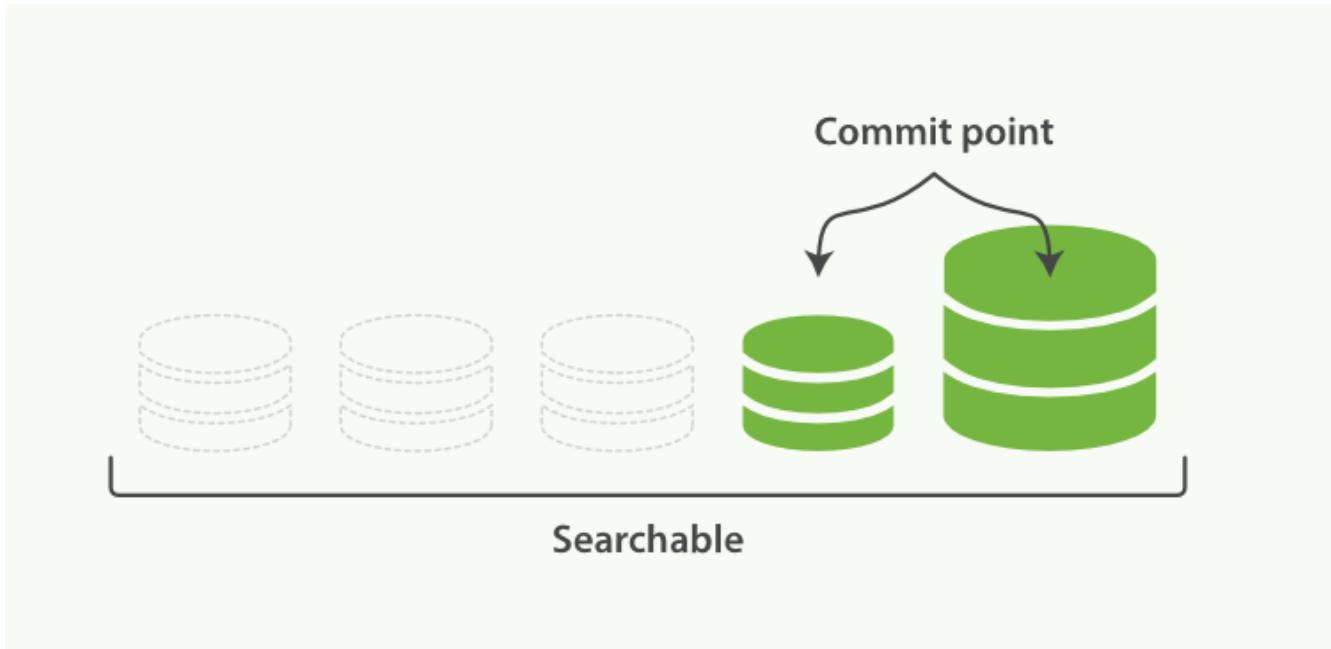


Figure 26. 一旦合并结束，老的段被除

合并大的段需要消耗大量的I/O和CPU资源，如果任其展会影响搜索性能。Elasticsearch在情况下会合并流程行源限制，所以搜索依然有足够的源很好地行。

TIP 看段和合并来好的例子取于合并的整体的建议。

optimize API

`optimize` API大可看做是制合并API。它会将一个分片制合并到`max_num_segments`参数指定大小的段数目。做的意思是少段的数量（通常少到一个），来提升搜索性能。

WARNING `optimize` API不被用在一个索引——一个正在被活更新的索引。后台合并流程已可以很好地完成工作。`optimizing` 会阻碍个程。不要干它！

在特定情况下，使用`optimize` API有益。例如在日志用例下，天、周、月的日志被存在一个索引中。老的索引上是只的；它也并不太可能会生化。

在情况下，使用`optimize`化老的索引，将一个分片合并一个独的段就很有用了；既可以省源，也可以使搜索更加快速：

```
POST /logstash-2014-10/_optimize?max_num_segments=1 ①
```

① 合并索引中的一个分片一个独的段

WARNING 注意，使用`optimize` API触段合并的操作一点也不会受到任何源上的限制。可能会消耗掉点上全部的I/O源，使其没有余裕来理搜索求，从而有可能使集群失去。如果想要索引行`optimize`，需要先使用分片分配（看`移旧索引`）把索引移到一个安全的点，再行。

深入搜索

在 **基础** 中涵了基本工具并有足的描述，我能够始用 Elasticsearch 搜索数据。用不了多，就会我想要的更多：希望匹配更活，排名果更精，不同域下搜索更具体。

想要，只知道如何使用 **match** 是不的，我需要理解数据以及如何能搜索到它。本章会解如何索引和我的数据我利用的相似度（word proximity）、部分匹配（partial matching）、模糊匹配（fuzzy matching）以及言感知（language awareness）些。

理解个如何献相度分 **_score** 有助于我的：保我最佳匹配文出在果首，以及削果中几乎不相的“尾（long tail）”。

搜索不全文搜索：我很大一部分数据都是化的，如日期和数字。我会以明化搜索与全文搜索最高效的合方式始本章的内容。

化搜索

化搜索（*Structured search*）是指有探那些具有内在数据的程。比如日期、和数字都是化的：它有精的格式，我可以些格式行操作。比常的操作包括比数字或的，或判定个的大小。

文本也可以是化的。如彩色可以有散的色集合：**(red)**、**(green)**、**(blue)**。一个博客可能被了**分布式（distributed）**和**搜索（search）**。商站上的商品都有UPCs（通用品Universal Product Codes）或其他的唯一，它都需要遵从格定的、化的格式。

在化中，我得到的果是非是即否，要存于集合之中，要存在集合之外。化不关心文件的相度或分；它的文包括或排除理。

在上是能通的，因一个数字不能比其他数字更合存于某个相同。果只能是：存于之中，抑或反之。同，于化文本来，一个要相等，要不等。没有更似概念。

精

当行精，我会使用器（filters）。器很重要，因它行速度非常快，不会算相度（直接跳了整个分段）而且很容易被存。我会在本章后面的器存中器的性能，不在只要住：尽可能多的使用式。

term 数字

我首先来看最常用的**term**，可以用它理数字（numbers）、布（Booleans）、日期（dates）以及文本（text）。

我以下面的例子始介，建并索引一些表示品的文，文里有字段‘`price`’和‘`productID`’（‘格’和‘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" }
```

我想要做的是具有某个格的所有品，有系数据背景的人肯定熟悉SQL，如果我将其用SQL形式表，会是下面：

```
SELECT document
FROM products
WHERE price = 20
```

在Elasticsearch的表式(query DSL)中，我可以使用term到相同的目的。term会是我指定的精。作其本身，term是的。它接受一个字段名以及我希望的数：

```
{
  "term" : {
    "price" : 20
  }
}
```

通常当一个精的候，我不希望行分算。只希望文行包括或排除的算，所以我使用constant_score以非分模式来行term并以一作一分。

最合的果是一个constant_score，它包含一个term：

```
GET /my_store/products/_search
{
  "query" : {
    "constant_score" : { ①
      "filter" : {
        "term" : { ②
          "price" : 20
        }
      }
    }
  }
}
```

① 我用constant_score将term化成器

② 我之前看到的 term

行后，一个所搜索到的结果与我期望的一致：只有文 2 命中并作 果返回（因 只有 2 的格是 20）：

```
"hits" : [
  {
    "_index" : "my_store",
    "_type" : "products",
    "_id" : "2",
    "_score" : 1.0, ①
    "_source" : {
      "price" : 20,
      "productID" : "KDKE-B-9947-#kL5"
    }
  }
]
```

① 置于 filter 句内不行 分或相 度的 算，所以所有的 果都会返回一个 分 1。

term 文本

如本部分始提到的一 ，使用 term 匹配字符串和匹配数字一 容易。如果我想要某个具体 UPC ID 的 品，使用 SQL 表 式会是如下：

```
SELECT product
FROM products
WHERE productID = "XHDK-A-1293-#fJ3"
```

成 表 式 (query DSL)，同 使用 term ，形式如下：

```
GET /my_store/products/_search
{
  "query" : {
    "constant_score" : {
      "filter" : {
        "term" : {
          "productID" : "XHDK-A-1293-#fJ3"
        }
      }
    }
  }
}
```

但 里有个小 ！我 无法 得期望的 果。什 ？ 不在 term ，而在于索引数据的方式。如果我 使用 analyze API (分析 API)，我 可以看到 里的 UPC 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
  } ]
}
```

里有几点需要注意：

- Elasticsearch 用 4 个不同的 token 而不是 1 个 token 来表示 1 个 UPC。
- 所有字母都是小写的。
- 失了 # 字符和哈希符 (#)。

所以当我用 term 精 XHDK-A-1293-#fJ3 的时候，找不到任何文，因为它并不在我的倒排索引中，正如前面呈现出的分析结果，索引里有四个 token。

然 ID 或其他任何精的理方式并不是我想要的。

为了避免 ，我需要告诉 Elasticsearch 字段具有精，要将其置成 not_analyzed 无需分析的。我可以在 自定字段映射 中看它的用法。为了修正搜索结果，我需要首先删除旧索引（因它的映射不再正）然后建一个能正映射的新索引：

```

DELETE /my_store ①

PUT /my_store ②
{
  "mappings" : {
    "products" : {
      "properties" : {
        "productID" : {
          "type" : "string",
          "index" : "not_analyzed" ③
        }
      }
    }
  }
}

```

- ① 除索引是必 的，因 我 不能更新已存在的映射。
- ② 在索引被 除后，我 可以 建新的索引并 其指定自定 映射。
- ③ 里我 告 Elasticsearch，我 不想 **productID** 做任何分析。

在我 可以 文 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 ， **term** 就能搜索到我 想要的 果， 我 再次搜索新索引 的数据（注意， 和 并没有 生任何改 ， 改 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 `productID` 字段是未分析 的， `term` 不会 其做任何分析， 会 行精 并返回文 1 。成功！

内部 器的操作

在内部， Elasticsearch 会在 行非 分 的 行多个操作：

1. 匹配文 .

`term` 在倒排索引中 `XHDK-A-1293-#fJ3` 然后 取包含 `term` 的所有文 。本例中，只有文 1 足我 要求。

2. 建 `bitset`.

器会 建一个 `bitset` （一个包含 0 和 1 的数 ），它描述了 个文 会包含 `term` 。匹配文 的 志位是 1 。本例中，`bitset` 的 `[1,0,0,0]` 。在内部，它表示成一个 “roaring bitmap” ，可以同 稀疏或密集的集合 行高效 。

3. 迭代 `bitset(s)`

一旦 个 生成了 `bitsets` ， Elasticsearch 就会循 迭代 `bitsets` 从而 到 足所有 条件的匹配文 的集合。 行 序是 式的，但一般来 先迭代稀疏的 `bitset` （因 它可以排除掉大量的文 ）。

4. 量使用 数.

Elasticsearch 能 存非 分 从而 取更快的 ，但是它也会不太 明地 存一些使用 少的 西。非 分 算因 倒排索引已 足 快了，所以我 只想 存那些我 知道 在将来会被再次使用的 ，以避免 源的浪 。

了 以上 想， Elasticsearch 会 个索引跟踪保留 使用的 史状 。如果 在最近的 256 次 中会被用到，那 它就会被 存到内存中。当 `bitset` 被 存后， 存会在那些低于 10,000 个文 （或少于 3% 的 索引数）的段（segment）中被忽略。 些小的段即将会消失，所以 它 分配 存是一 浪 。

情况并非如此（ 行有它的 性， 取决于 是如何重新 的，有些 式的算法是基于 代 的），理 上非 分 先于 分 行。非 分 任 旨在降低那些将 分 算

来更高成本的文 数量，从而 到快速搜索的目的。

从概念上 住非 分 算是首先 行的， 将有助于写出高效又快速的搜索 求。

合 器

前面的 个例子都是 个 器 (filter) 的使用方式。 在 用中，我 很有可能会 多个 或字段。比方 ， 用 Elasticsearch 来表 下面的 SQL ?

```
SELECT product
FROM products
WHERE (price = 20 OR productID = "XHDK-A-1293-#fJ3")
AND (price != 30)
```

情况下，我 需要 **bool** (布) 器。 是个 合 器 (*compound filter*) ，它可以接受多个其他 器作 参数，并将 些 器 合成各式各 的布 () 合。

布 器

一个 **bool** 器由三部分 成：

```
{
  "bool" : {
    "must" : [],
    "should" : [],
    "must_not" : []
  }
}
```

must

所有的 句都必 (must) 匹配，与 AND 等 。

must_not

所有的 句都不能 (must not) 匹配，与 NOT 等 。

should

至少有一个 句要匹配，与 OR 等 。

就 ! 当我 需要多个 器 ，只 将它 置入 **bool** 器的不同部分即可。

NOTE 一个 **bool** 器的 个部分都是可 的 (例如，我 可以只有一个 **must** 句)，而且 个部分内部可以只有一个或一 器。

用 Elasticsearch 来表示本部分 始 的 SQL 例子，将 个 **term** 器置入 **bool** 器的 **should** 句内，再 加一个 句 理 NOT 非的条件：

```

GET /my_store/products/_search
{
  "query": {
    "filtered": { ①
      "filter": {
        "bool": {
          "should": [
            { "term": {"price": 20}}, ②
            { "term": {"productID": "XHDK-A-1293-#fJ3"} } ②
          ],
          "must_not": {
            "term": {"price": 30} ③
          }
        }
      }
    }
  }
}

```

① 注意，我 然需要一个 `filtered` 将所有的 西包起来。

② 在 `should` 句 里面的 个 `term` 器与 `bool` 器是父子 系， 个 `term` 条件需要匹配其一。

③ 如果一个 品的 格是 30， 那 它会自 被排除，因 它 于 `must_not` 句里面。

我 搜索的 果返回了 2 个命中 果， 个文 分 匹配了 `bool` 器其中的一个条件：

```

"hits": [
  {
    "_id": "1",
    "_score": 1.0,
    "_source": {
      "price": 10,
      "productID": "XHDK-A-1293-#fJ3" ①
    }
  },
  {
    "_id": "2",
    "_score": 1.0,
    "_source": {
      "price": 20, ②
      "productID": "KDKE-B-9947-#kL5"
    }
  }
]

```

① 与 `term` 器中 `productID = "XHDK-A-1293-#fJ3"` 条件匹配

② 与 `term` 器中 `price = 20` 条件匹配

嵌套布 器

尽管 `bool` 是一个 合的 器，可以接受多个子 器，需要注意的是 `bool` 器本身 然 只是一个 器。 意味着我 可以将一个 `bool` 器置于其他 `bool` 器内部， 我 提供了 任意 布 行 理的能力。

于以下 个 SQL 句：

```
SELECT document
FROM products
WHERE productID      = "KDKE-B-9947-#kL5"
  OR (    productID = "JODL-X-1937-#pV7"
    AND price       = 30 )
```

我 将其 成一 嵌套的 `bool` 器：

```
GET /my_store/products/_search
{
  "query" : {
    "filtered" : {
      "filter" : {
        "bool" : {
          "should" : [
            { "term" : {"productID" : "KDKE-B-9947-#kL5"}}, ①
            { "bool" : { ①
              "must" : [
                { "term" : {"productID" : "JODL-X-1937-#pV7"}}, ②
                { "term" : {"price" : 30}} ②
              ]
            }
          ]
        }
      }
    }
  }
}
```

① 因 `term` 和 `bool` 器是兄弟 系，他 都 于外 的布 `should` 的内部，返回的命中文 至少 匹配其中一个 器的条件。

② 一个 `term` 句作 兄弟 系，同 于 `must` 句之中，所以返回的命中文 要必 都能同 匹配 个条件。

得到的 果有 个文 ，它 各匹配 `should` 句中的一个条件：

```

"hits" : [
  {
    "_id" : "2",
    "_score" : 1.0,
    "_source" : {
      "price" : 20,
      "productID" : "KDKE-B-9947-#kL5" ①
    }
  },
  {
    "_id" : "3",
    "_score" : 1.0,
    "_source" : {
      "price" : 30, ②
      "productID" : "JODL-X-1937-#pV7" ②
    }
  }
]

```

① 一个 `productID` 与外层的 `bool` 器 `should` 里的唯一一个 `term` 匹配。

② 一个字段与嵌套的 `bool` 器 `must` 里的一个 `term` 匹配。

只是个例子，但足以展示布尔器可以用来作构造条件的基本建模。

多个精

`term` 于一个非常有用，但通常我可能想搜索多个。如果我想要价格字段 \$20 或 \$30 的文如何理？

不需要使用多个 `term`，我只要用一个 `terms`（注意末尾的 `s`），`terms` 好比是 `term` 的复数形式（以英文名的复数做比）。

它几乎与 `term` 的使用方式一模一，与指定一个格不同，我只要将 `term` 字段的改数值即可：

```
{
  "terms" : {
    "price" : [20, 30]
  }
}
```

与 `term` 一，也需要将其置入 `filter` 句的常量分中使用：

```

GET /my_store/products/_search
{
  "query" : {
    "constant_score" : {
      "filter" : {
        "terms" : { ①
          "price" : [20, 30]
        }
      }
    }
  }
}

```

① ↑ terms 被置于 constant_score 中

行 果返回第二、第三和第四个文：

```

"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 (equals) (判断)。如何理解 句？

如果我 有一个 term () 器 { "term" : { "tags" : "search" } } , 它会与以下 个文 同匹配：

```
{ "tags" : ["search"] }
{ "tags" : ["search", "open_source"] } ①
```

① 尽管第二个文 包含除 search 以外的其他 , 它 是被匹配并作 果返回。

回 一下 term 是如何工作的？ Elasticsearch 会在倒排索引中 包括某 term 的所有文 , 然后造一个 bitset 。在我 的例子中，倒排索引表如下：

Token	DocIDs
open_source	2
search	1,2

当 term 匹配 search , 它直接在倒排索引中 到 并 取相 的文 ID, 如倒排索引所示, 里文 1 和文 2 均包含 , 所以 个文 会同 作 果返回。

NOTE

由于倒排索引表自身的特性，整个字段是否相等会 以 算，如果 定某个特定文 是否只 (only) 包含我 想要 的 ?首先我 需要在倒排索引中 到相 的 并 取文 ID, 然后再 描倒排索引中的 行 , 看它 是否包含其他的 terms 。

可以想象， 不 低效，而且代 高昂。正因如此， term 和 terms 是 必 包含 (must contain) 操作，而不是 必 精 相等 (must equal exactly) 。

精 相等

如果定期望得到我 前面 的那 行 (即整个字段完全相等) , 最好的方式是 加并索引 一个字段, 个字段用以存 字段包含 的数量, 同 以上面提到的 个文 例, 在我 包括了一个数的新字段 :

```
{ "tags" : ["search"], "tag_count" : 1 }
{ "tags" : ["search", "open_source"], "tag_count" : 2 }
```

一旦 加 个用来索引 term 数目信息的字段, 我 就可以 造一个 constant_score , 来 保 果中的文 所包含的 数量与要求是一致的 :

```

GET /my_index/my_type/_search
{
  "query": {
    "constant_score" : {
      "filter" : {
        "bool" : {
          "must" : [
            { "term" : { "tags" : "search" } }, ①
            { "term" : { "tag_count" : 1 } } ②
          ]
        }
      }
    }
  }
}

```

① 所有包含 term `search` 的文 。

② 保文 只有一个 。

个 在只会匹配具有 个 `search` 的文 , 而不是任意一个包含 `search` 的文 。

本章到目前 止, 于数字, 只介 如何 理精 。 上, 数字 行 有 会更有用。例
如, 我 可能想要 所有 格大于 \$20 且小于 \$40 美元的 品。

在 SQL 中, 可以表示 :

```

SELECT document
FROM products
WHERE price BETWEEN 20 AND 40

```

Elasticsearch 有 `range` , 不出所料地, 可以用它来 于某个 内的文 :

```

"range" : {
  "price" : {
    "gte" : 20,
    "lte" : 40
  }
}

```

`range` 可同 提供包含 (inclusive) 和不包含 (exclusive) 表 式, 可供 合的 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 **range** 的例子：

```
GET /my_store/products/_search
{
  "query": {
    "constant_score": {
      "filter": {
        "range": {
          "price": {
            "gte": 20,
            "lt": 40
          }
        }
      }
    }
  }
}
```

如果想要 无界 (比方 >20) , 只 省略其中一 的限制：

```
"range": {
  "price": {
    "gt": 20
  }
}
```

日期

range 同 可以 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 理日期字段 , **range** 支持 日期 算 (date math) 行操作, 比方 , 如果我想 在 去一小 内的所有文 :

```
"range" : {  
    "timestamp" : {  
        "gt" : "now-1h"  
    }  
}
```

个 器会一直 在 去一个 小 内的所有文 , 器作 一个 滑 口 (*sliding window*) 来 文 。

日期 算 可以被 用到某个具体的 , 并非只能是一个像 now 的占位符。只要在某个日期后加上一个双管符号 (||) 并 跟一个日期数学表 式就能做到 :

```
"range" : {  
    "timestamp" : {  
        "gt" : "2014-01-01 00:00:00",  
        "lt" : "2014-01-01 00:00:00||+1M" ①  
    }  
}
```

① 早于 2014 年 1 月 1 日加 1 月 (2014 年 2 月 1 日 零)

日期 算是 日 相 (*calendar aware*) 的, 所以它不 知道 月的具体天数, 知道某年的 天数 (年) 等信息。更 的内容可以参考 : [格式参考文](#) 。

字符串

`range` 同 可以 理字符串字段, 字符串 可采用 字典 序 (*lexicographically*) 或字母序 (*alphabetically*) 。例如, 下面 些字符串是采用字典序 (*lexicographically*) 排序的 :

- 5, 50, 6, B, C, a, ab, abb, abc, b

NOTE

在倒排索引中的 就是采取字典 序 (*lexicographically*) 排列的, 也是字符串 可以使用 个 序来 定的原因。

如果我 想 从 **a** 到 **b** (不包含) 的字符串, 同 可以使用 `range` 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，Elasticsearch 会比日期或数字的慢多。

算成 可能。但字符串却并非如此，要想 其使用

上是在 内的 个 都 行 term 器，

字符串 在 低基数 (low cardinality) 字段 (即只有少量唯一) 可以正常工作，但是唯一 越多，字符串 的 算会越慢。

理 Null

回想在之前例子中，有的文 有名

tags () 的字段，它是个多 字段，一个文

可能有一个或多个 ，也可能根本就没有 ？

。如果一个字段没有 ，那 如何将它存入倒排索引中的

是个有欺 性的 ，因 答案是：什 都不存。 我 看看之前内容里提到 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 在 个数据 中 ？无法做到！ 的 ，一个倒排索引只是一个 token 列表和与之相 的文 信息，如果字段不存在，那 它也不会持有任何 token，也就无法在倒排索引 中表 。

最 ， 也就意味着，`null, []` (空数) 和 `[null]` 所有 些都是等 的，它 无法存于倒排索引中。

然，世界并不 ，数据往往会有 失字段，或有 式的空 或空数 。 了 些状况，Elasticsearch 提供了一些工具来 理空或 失 。

存在

第一件武器就是 `exists` 存在 。 个 会返回那些在指定字段有任何 的文 ， 我 索引一些示例文 并用 的例子来 明：

```
POST /my_index/posts/_bulk
{ "index": { "_id": "1" } }
{ "tags" : ["search"] } ①
{ "index": { "_id": "2" } }
{ "tags" : ["search", "open_source"] } ②
{ "index": { "_id": "3" } }
{ "other_field" : "some data" } ③
{ "index": { "_id": "4" } }
{ "tags" : null } ④
{ "index": { "_id": "5" } }
{ "tags" : ["search", null] } ⑤
```

- ① tags 字段有 1 个。
- ② tags 字段有 2 个。
- ③ tags 字段 失。
- ④ tags 字段被置 null。
- ⑤ tags 字段有 1 个 和 1 个 null。

以上文 集合中 tags 字段 的倒排索引如下：

Token	DocIDs
open_source	2
search	1,2,5

我 的目 是 到那些被 置 字段的文 , 并不 心 的具体内容。只要它存在于文 中即可, 用 SQL 的 就是用 IS NOT NULL 非空 行 :

```
SELECT tags
FROM posts
WHERE tags IS NOT NULL
```

在 Elasticsearch 中, 使用 exists 的方式如下:

```
GET /my_index/posts/_search
{
  "query": {
    "constant_score": {
      "filter": {
        "exists": { "field": "tags" }
      }
    }
  }
}
```

个 返回 3 个文 :

```

"hits" : [
  {
    "_id" : "1",
    "_score" : 1.0,
    "_source" : { "tags" : ["search"] }
  },
  {
    "_id" : "5",
    "_score" : 1.0,
    "_source" : { "tags" : ["search", null] } ①
  },
  {
    "_id" : "2",
    "_score" : 1.0,
    "_source" : { "tags" : ["search", "open source"] }
  }
]

```

① 尽管文 5 有 `null`，但它 会被命中返回。字段之所以存在，是因 有（`search`）可以被索引，所以 `null` 不会 生任何影 。

而易 ，只要 `tags` 字段存在（term）的文 都会命中并作 果返回，只有 3 和 4 个文 被排除。

失

个 `missing` 本 上与 `exists` 恰好相反：它返回某个特定 无 字段的文 ，与以下 SQL 表 的意思似：

```

SELECT tags
FROM posts
WHERE tags IS NULL

```

我 将前面例子中 `exists` 成 `missing` :

```

GET /my_index/posts/_search
{
  "query" : {
    "constant_score" : {
      "filter": {
        "missing" : { "field" : "tags" }
      }
    }
  }
}

```

按照期望的那 ，我 得到 3 和 4 个文 （ 个文 的 `tags` 字段没有 ）：

```

"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 需要区分一个字段是没有 , 是它已被 式的 置成了 `null` 。在之前例子中, 我 看到的 行 是无法做到 点的; 数据被 失了。不 幸 的是, 我 可以 将 式的 `null` 替 成我 指定占位符 (*placeholder*) 。

在 字符串 (string) 、数字 (numeric) 、布 (Boolean) 或日期 (date) 字段指定映射 , 同 可以 之 置 `null_value` 空 , 用以 理 式 `null` 的情况。不 即使如此, 是会将一个没有 的字段从倒排索引中排除。

当 合 的 `null_value` 空 的 候, 需要保 以下几点 :

- 它会匹配字段的 型, 我 不能 一个 `date` 日期字段 置字符串 型的 `null_value` 。
- 它必 与普通 不一 , 可以避免把 当成 `null` 空的情况。

象上的存在与 失

不 可以 核心 型, `exists` and `missing` 可以 理一个 象的内部字段。以下面文 例 :

```

{
  "name" : {
    "first" : "John",
    "last" : "Smith"
  }
}

```

我 不 可以 `name.first` 和 `name.last` 的存在性, 也可以 `name` , 不 在 映射 中, 如上 象的内部是个扁平的字段与 (field-value) 的 , 似下面 :

```
{  
  "name.first" : "John",  
  "name.last" : "Smith"  
}
```

那我如何用 `exists` 或 `missing` 检查 `name` 字段？`name` 字段并不真存在于倒排索引中。

原因是当我运行下面一个的时候：

```
{  
  "exists" : { "field" : "name" }  
}
```

行的是：

```
{  
  "bool": {  
    "should": [  
      { "exists": { "field": "name.first" }},  
      { "exists": { "field": "name.last" }}  
    ]  
  }  
}
```

也就意味着，如果 `first` 和 `last` 都是空，那么 `name` 个命名空间才会被不存在。

于存

在本章前面（[器的内部操作](#)）中，我已经介绍了器是如何算的。其核心是采用一个 `bitset` 与器匹配的文。Elasticsearch 地把些 `bitset` 存起来以随后使用。一旦存成功，`bitset` 可以用任何已使用的相同器，而无需再次算整个器。

些 `bitsets` 存是“智能”的：它以量方式更新。当我索引新文，只需将那些新文加入已有 `bitset`，而不是整个存一遍又一遍的重算。和系其他部分一，器是的，我无需担心存期。

独立的器存

属于一个件的 `bitsets` 是独立于它所属搜索求其他部分的。就意味着，一旦被存，一个可以被用作多个搜索求。`bitsets` 并不依于它所存在的上下文。使得存可以加速中常使用的部分，从而降低少、易的部分所来的消耗。

同，如果一个求重用相同的非分，它存的 `bitset` 可以被个搜索里的所有例所重用。

我看看下面例子中的，它足以以下任意一个条件的子件：

- 在收件箱中，且没有被的

- 不在收件箱中，但被注重要的

```
GET /inbox/emails/_search
{
  "query": {
    "constant_score": {
      "filter": {
        "bool": {
          "should": [
            { "bool": {
                "must": [
                  { "term": { "folder": "inbox" }}, ①
                  { "term": { "read": false }}
                ]
              }},
            { "bool": {
                "must_not": {
                  "term": { "folder": "inbox" } ①
                },
                "must": {
                  "term": { "important": true }
                }
              }}
            ]
          }
        }
      }
    }
}
```

① 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` 句，一个是 `must_not` 句，但两者是完全相同的。意味着在第一个句行后，bitset 就会被算然后存起来供一个使用。当再次行一个，收件箱的两个过滤器已被存了，所以两个句都会使用已存的 bitset。

点与表式 (query DSL) 的可合性合得很好。它易被移到表达式的任何地方，或者在同一中的多个位置用。不能方便者，而且提升性能有直接的益。

自存行

在 Elasticsearch 的早版本中，的行是存一切可以存的形象。也通常意味着系存 bitsets 太富侵略性，从而因清理存来性能力。不如此，尽管很多过滤器都很容易被，但本上是慢于存的（以及从存中用）。存些过滤器的意不大，因可以地再次行过滤器。

一个倒排是非常快的，然后大多数文件却很少使用它。例如 `term` 字段 `"user_id"`：如果有上百万的用，个具体的用 ID 出的概率都很小。那两个过滤器存 bitsets 就不是很合算，因存的结果很可能在重用之前就被除了。

存的性能有着重的影。更重的是，它者以区分有良好表的存以及无用存

。了解解决，Elasticsearch 会基于使用次自存。如果一个非分在最近的 256 中被使用（次数取决于型），那个就会作存的候。但是，并不是所有的片段都能保存 bitset。只有那些文数量超 10,000（或超文数量的 3%）才会存 bitset。因小的片段可以很快的行搜索和合并，里存的意不大。

一旦存了，非分算的 bitset 会一直留在存中直到它被除。除是基于 LRU 的：一旦存了，最近最少使用的器会被除。

全文搜索

我已介了搜索化数据的用示例，在来探全文搜索（full-text search）：

在全文字段中搜索到最相的文。

全文搜索个最重要的方面是：

相性 (Relevance)

它是与其果的相程度，并根据相程度果排名的一能力，算方式可以是 TF/IDF 方法（参 [相性的介](#)）、地理位置近、模糊相似，或其他的某些算法。

分析 (Analysis)

它是将文本有区的、化的 token 的一个程，（参 [分析的介](#)）目的是了（a）建倒排索引以及（b）倒排索引。

一旦相性或分析个方面的，我所的境是于的而不是。

基于与基于全文

所有会或多或少的行相度算，但不是所有都有分析段。和一些特殊的完全不会文本行操作的（如 `bool` 或 `function_score`）不同，文本可以分成大家族：

基于的

如 `term` 或 `fuzzy` 的底不需要分析段，它一个行操作。用 `term` `Foo` 只要在倒排索引中准，并且用 TF/IDF 算法一个包含的文算相度分 `_score`。

住 `term` 只倒排索引的精匹配，点很重要，它不会的多性行理（如，`foo` 或 `FOO`）。里，无考是如何存入索引的。如果是将 `["Foo", "Bar"]` 索引存入一个不分析的（`not_analyzed`）包含精的字段，或者将 `Foo Bar` 索引到一个有 `whitespace` 空格分析器的字段，者的果都会是在倒排索引中有 `Foo` 和 `Bar` 个。

基于全文的

像 `match` 或 `query_string` 的是高，它了解字段映射的信息：

- 如果日期（`date`）或整数（`integer`）字段，它会将字符串分作日期或整数待。
- 如果一个（`not_analyzed`）未分析的精字符串字段，它会将整个字符串作一个待。
- 但如果要一个（`analyzed`）已分析的全文字段，它会先将字符串到一个合

的分析器，然后生成一个供 的 列表。

一旦 成了 列表， 个 会 个 逐一 行底 的 ， 再将 果合并， 然后 个文 生成 一个最 的相 度 分。

我 将会在随后章 中 个 程。

我 很少直接使用基于 的搜索，通常情况下都是 全文 行 ， 而非 个 ， 只需要 的 行 一个高 全文 （ 而在高 内部会以基于 的底 完成搜索）。

当我 想要 一个具有精 的 `not_analyzed` 未分析字段之前， 需要考虑， 是否真的采用 分 ， 或者非 分 会更好。

通常可以用是、非 二元 表示， 所以更 合用 ， 而且 做可以有效 利用 存：

```
NOTE
GET /_search
{
  "query": {
    "constant_score": {
      "filter": {
        "term": { "gender": "female" }
      }
    }
  }
}
```

匹配

匹配 `match` 是个 核心 。无 需要 什 字段， `match` 都 会是首 的 方式。它是一个高 全文 ， 表示它既能 理全文字段， 又能 理精 字段。

就是 ， `match` 主要的 用 景就是 行全文搜索， 我 以下面一个 例子来 明全文搜索是如何工作的：

索引一些数据

首先，我 使用 `bulk API` 建一些新的文 和索引：

```

DELETE /my_index ①

PUT /my_index
{ "settings": { "number_of_shards": 1 }} ②

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" }

```

① 除已有的索引。

② 后，我 会在 [被破坏的相 性！](#) 中解 只 一个索引分配一个主分片的原因。

↑

我 用第一个示例来解 使用 `match` 搜索全文字段中的 个 ：

```

GET /my_index/my_type/_search
{
  "query": {
    "match": {
      "title": "QUICK!"
    }
  }
}

```

Elasticsearch 行上面 个 `match` 的 是：

1. 字段 型。

`title` 字段是一个 `string` 型 (`analyzed`) 已分析的全文字段， 意味着 字符串本身也 被分析。

2. 分析 字符串。

将 的字符串 `QUICK!` 入 准分析器中， 出的 果是 个 `quick`。因 只有一个 ， 所以 `match` 行的是 个底 `term`。

3. 匹配文 。

用 `term` 在倒排索引中 `quick` 然后 取一 包含 的文 ， 本例的 果是文 : 1、2 和 3。

4. 个文 分。

用 term 算 个文 相 度 分 `_score`，是 将 (term frequency, 即 `quick` 在相文 的 `title` 字段中出 的 率) 和反向文 率 (inverse document frequency, 即 `quick` 在所有文 的 `title` 字段中出 的 率)，以及字段的 度 (即字段越短相 度越高) 相 合的算方式。参 相 性的介 。

个 程 我 以 下 () 果：

```
"hits": [
  {
    "_id": "1",
    "_score": 0.5, ①
    "_source": {
      "title": "The quick brown fox"
    }
  },
  {
    "_id": "3",
    "_score": 0.44194174, ②
    "_source": {
      "title": "The quick brown fox jumps over the quick dog"
    }
  },
  {
    "_id": "2",
    "_score": 0.3125, ②
    "_source": {
      "title": "The quick brown fox jumps over the lazy dog"
    }
  }
]
```

① 文 1 最相 ，因 它的 `title` 字段更短，即 `quick` 占据内容的一大部分。

② 文 3 比文 2 更具相 性，因 在文 2 中 `quick` 出 了 次。

多

如果我 一 次 只 能 搜 索 一 个 ，那 全 文 搜 索 就 会 不 太 活，幸 幸 的 是 `match` 多 得 ！

```
GET /my_index/my_type/_search
{
  "query": {
    "match": {
      "title": "BROWN DOG!"
    }
  }
}
```

上面 个 返回所有四个文 :

```
{  
  "hits": [  
    {  
      "_id": "4",  
      "_score": 0.73185337, ①  
      "_source": {  
        "title": "Brown fox brown dog"  
      }  
    },  
    {  
      "_id": "2",  
      "_score": 0.47486103, ②  
      "_source": {  
        "title": "The quick brown fox jumps over the lazy dog"  
      }  
    },  
    {  
      "_id": "3",  
      "_score": 0.47486103, ②  
      "_source": {  
        "title": "The quick brown fox jumps over the quick dog"  
      }  
    },  
    {  
      "_id": "1",  
      "_score": 0.11914785, ③  
      "_source": {  
        "title": "The quick brown fox"  
      }  
    }  
  ]  
}
```

① 文 4 最相 , 因 它包含 "brown" 次以及 "dog" 一次。

② 文 2、3 同 包含 brown 和 dog 各一次, 而且它 title 字段的 度相同, 所以具有相同的 分。

③ 文 1 也能匹配, 尽管它只有 brown 没有 dog 。

因 match 必 个 (["brown", "dog"]), 它在内部 上先 行 次 term , 然后将 次 的 果合并作 最 果 出。 了做到 点, 它将 个 term 包入一个 bool 中, 信息 布 。

以上示例告 我 一个重要信息: 即任何文 只要 title 字段里包含 指定 中的至少一个 就能匹配, 被匹配的 越多, 文 就越相 。

提高精度

用 任意 匹配文 可能会 致 果中出 不相 的 尾。 是 散 式搜索。可能我

只想搜索包含 所有 的文，也就是，不去匹配 brown OR dog，而通 匹配 brown AND dog 到所有文。

match 可以接受 operator 操作符作 入参数，情况下 操作符是 or。我 可以将它修改成 and 所有指定 都必 匹配：

```
GET /my_index/my_type/_search
{
  "query": {
    "match": {
      "title": {①
        "query": "BROWN DOG!",
        "operator": "and"
      }
    }
  }
}
```

① match 的 需要做 整才能使用 operator 操作符参数。

个 可以把文 1 排除在外，因 它只包含 个 中的一个。

控制精度

在 所有 与 任意 二 一有点 于非 即白。如果用 定 5 个 ，想 只包含其中 4 个的文， 如何 理？将 operator 操作符参数 置成 and 只会将此文 排除。

有 候 正是我 期望的，但在全文搜索的大多数 用 景下，我 既想包含那些可能相 的文，同 又排除那些不太相 的。 句 ，我 想要 于中 某 果。

match 支持 minimum_should_match 最小匹配参数， 我 可以指定必 匹配的数用来表示一个文 是否相 。我 可以将其 置 某个具体数字，更常用的做法是将其 置 一个百分数，因 我 无法控制用 搜索 入的 数量：

```
GET /my_index/my_type/_search
{
  "query": {
    "match": {
      "title": {
        "query": "quick brown dog",
        "minimum_should_match": "75%"
      }
    }
  }
}
```

当 定百分比的 候， minimum_should_match 会做合 的事情：在之前三 的示例中， 75% 会自被截断成 66.6%，即三个里面 个。无 个 置成什，至少包含一个 的文 才会被是匹配的。

NOTE

参数 `minimum_should_match` 的置非常活，可以根据用入的数目用不同的。完整的信息参考文
<https://www.elastic.co/guide/en/elasticsearch/reference/master/query-dsl-minimum-should-match.html#query-dsl-minimum-should-match>

了完全理解 `match` 是如何理多的，我就需要看如何使用 `bool` 将多个条件合在一起。

合

在 合 器中，我如何使用 `bool` 器通 `and`、`or` 和 `not` 合将多个器行合。在 中，`bool` 有似功能，只有一个重要的区。

器做二元判断：文是否出在果中？但更精妙，它除了决定一个文是否被包括在果中，会算文的相程度。

与器一，`bool` 也可以接受 `must`、`must_not` 和 `should` 参数下的多个句。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" } }
      ]
    }
  }
}
```

以上的果返回 `title` 字段包含 `quick` 但不包含 `lazy` 的任意文。目前止，与 `bool`器的工作方式非常相似。

区就在于个 `should` 句，也就是：一个文不必包含 `brown` 或 `dog` 个，但如果一旦包含，我就它更相：

```
{
  "hits": [
    {
      "_id": "3",
      "_score": 0.70134366, ①
      "_source": {
        "title": "The quick brown fox jumps over the quick dog"
      }
    },
    {
      "_id": "1",
      "_score": 0.3312608,
      "_source": {
        "title": "The quick brown fox"
      }
    }
  ]
}
```

① 文 3 会比文 1 有更高 分是因 它同 包含 brown 和 dog 。

分 算

`bool` 会 个文 算相 度 分 `_score` , 再将所有匹配的 `must` 和 `should` 句的分数 `_score` 求和, 最后除以 `must` 和 `should` 句的 数。

`must_not` 句不会影 分 ; 它的作用只是将不相 的文 排除。

控制精度

所有 `must` 句必 匹配, 所有 `must_not` 句都必 不匹配, 但有多少 `should` 句 匹配 ? 情况下, 没有 `should` 句是必 匹配的, 只有一个例外 : 那就是当没有 `must` 句的 时候, 至少有一个 `should` 句必 匹配。

就像我 能控制 `match` 的精度 一 , 我 可以通 `minimum_should_match` 参数控制需要匹配的 `should` 句的数量, 它既可以是一个 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 ①
    }
  }
}

```

① 也可以用百分比表示。

如果会将所有 足以下条件的文 返回： `title` 字段包含 "brown" AND "fox" 、 "brown" AND "dog" 或 "fox" AND "dog" 。如果有文 包含所有三个条件，它会比只包含 个的文 更相 。

如何使用布 匹配

目前 止，可能已 意 到多 `match` 只是 地将生成的 `term` 包 在一个 `bool` 中。如果使用 的 `or` 操作符， 个 `term` 都被当作 `should` 句， 就要求必至少匹配一条 句。以下 个 是等 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` 都被当作 `must` 句，所以 所有 (all) 句都必 匹配。以下 个 是等 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 `bool` 直接 ，使以下 个 等：

```
{  
  "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 ①  
  }  
}
```

① 因 只有三条 句， `match` 的参数 `minimum_should_match` 75% 会被截断成 2 。即三条 `should` 句中至少有 条必 匹配。

当然，我 通常将 些 用 `match` 来表示，但是如果了解 `match` 内部的工作原理，我 就能根据自己的需要来控制 程。有些 候 个 `match` 无法 足需求，比如 某些 条件分配更高的 重。我 会在下一小 中看到 个例子。

句提升 重

当然 `bool` 不限于合的个 `match`，它可以合任意其他的，以及其他 `bool`。普遍的用法是通多个独立的分数，从而到一个文微其相度分 `_score` 的目的。

假想要于“full-text search（全文搜索）”的文，但我希望提及“Elasticsearch”或“Lucene”的文予更高的重，里更高重是指如果文出“Elasticsearch”或“Lucene”，它会比没有的出些的文得更高的相度分 `_score`，也就是，它会出在果集的更上面。

一个的 `bool` 允我写出如下非常 的：

```
GET /_search
{
  "query": {
    "bool": {
      "must": {
        "match": {
          "content": { ①
            "query": "full text search",
            "operator": "and"
          }
        }
      },
      "should": [ ②
        { "match": { "content": "Elasticsearch" } },
        { "match": { "content": "Lucene" } }
      ]
    }
  }
}
```

① `content` 字段必包含 `full`、`text` 和 `search` 所有三个。

② 如果 `content` 字段也包含 `Elasticsearch` 或 `Lucene`，文会得更高的分 `_score`。

`should` 句匹配得越多表示文的相度越高。目前止挺好。

但是如果我想包含 `Lucene` 的有更高的重，并且包含 `Elasticsearch` 的句比 `Lucene` 的重更高，如何理？

我可以通指定 `boost` 来控制任何句的相的重，`boost` 的 1，大于 1 会提升一个句的相重。所以下面重写之前的：

```

GET /_search
{
  "query": {
    "bool": {
      "must": {
        "match": { ①
          "content": {
            "query": "full text search",
            "operator": "and"
          }
        }
      },
      "should": [
        { "match": {
          "content": {
            "query": "Elasticsearch",
            "boost": 3 ②
          }
        }},
        { "match": {
          "content": {
            "query": "Lucene",
            "boost": 2 ③
          }
        }}
      ]
    }
  }
}

```

① 些 句使用 的 `boost 1`。

② 条 句更 重要，因 它有最高的 `boost`。

③ 条 句比使用 的更重要，但它的的重要性不及 `Elasticsearch` 句。

`boost` 参数被用来提升一个 句的相 重（`boost` 大于 1）或降低相 重（`boost` 于 0 到 1 之），但是 提升或降低并不是 性的， 句 ，如果一个 `boost` 2，并不能 得 倍的 分 `_score`。

NOTE 相反，新的 分 `score` 会在 用 重提升之后被 _ 一化， 型的 都有自己的 一算法， 超出了本 的 ，所以不作介 。 的 ，更高的 `boost` 我 来更高的 分 `_score`。

如果不基于 TF/IDF 要 自己的 分模型，我 就需要 重提升的 程能有更多控制，可以使用 `function_score` 操 一个文 的 重提升方式而跳 一化 一 。

更多的 合 方式会在下章[多字段搜索](#)中介，但在此之前，我 先看 外一个重要的 特性：文本分析（text analysis）。

控制分析

只能 倒排索引表中真 存在的 ，所以保 文 在索引 与 字符串在搜索 用相同的分析 程非常重 要， 的 才能 匹配倒排索引中的 。

尽管是在 文 ，不 分析器可以由 个字段决定。 个字段都可以有不同的分析器，既可以通 配置 字段指定分析器，也可以使用更高 的 型 (type)、索引 (index) 或 点 (node) 的 配置。在索 引 ，一个字段 是根据配置或 分析器分析的。

例如 `my_index` 新 一个字段：

```
PUT /my_index/_mapping/my_type
{
  "my_type": {
    "properties": {
      "english_title": {
        "type": "string",
        "analyzer": "english"
      }
    }
  }
}
```

在我 就可以通 使用 `analyze` API 来分析 `Foxes`，而比 `english_title` 字段和 `title` 字段在索引 的分析 果：

```
GET /my_index/_analyze
{
  "field": "my_type.title", ①
  "text": "Foxes"
}

GET /my_index/_analyze
{
  "field": "my_type.english_title", ②
  "text": "Foxes"
}
```

① 字段 `title`，使用 的 `standard` 准分析器，返回 `foxes`。

② 字段 `english_title`，使用 `english` 英 分析器，返回 `fox`。

意味着，如果使用底 `term` 精 `fox`，`english_title` 字段会匹配但 `title` 字段不会。

如同 `match` 的高 知道字段映射的 系，能 个被 的字段 用正 的分析器。可以使用 `validate-query` API 看 个行：

```

GET /my_index/my_type/_validate/query?explain
{
  "query": {
    "bool": {
      "should": [
        { "match": { "title": "Foxes" } },
        { "match": { "english_title": "Foxes" } }
      ]
    }
  }
}

```

返回 句的 explanation 果：

```
(title:foxes english_title:fox)
```

match 个字段使用合 的分析器，以保 它在 个 都 字段使用正 的格式。

分析器

然我 可以在字段 指定分析器，但是如果 没有指定任何的分析器，那 我 如何能 定 个字
段使用的是 个分析器 ？

分析器可以从三个 面 行定：按字段 (per-field)、按索引 (per-index) 或全局 省 (global default)。Elasticsearch 会按照以下 序依次 理，直到它 到能 使用的分析器。索引 的 序如下：

- 字段映射里定 的 **analyzer**，否
- 索引 置中名 **default** 的分析器，
- **standard** 准分析器

在搜索 ， 序有些 不同：

- 自己定 的 **analyzer**，否
- 字段映射里定 的 **analyzer**，否
- 索引 置中名 **default** 的分析器，
- **standard** 准分析器

有 ， 在索引 和搜索 使用不同的分析器是合理的。我 可能要想 同 建索引（例如，所有 **quick** 出 的地方，同 也 **fast**、**rapid** 和 **speedy** 建索引）。但在搜索 ， 我 不需要搜索所有的同 ， 取而代之的是 用 入的 是否是 **quick**、**fast**、**rapid** 或 **speedy**。

了区分，Elasticsearch 也支持一个可 的 **search_analyzer** 映射，它 会 用于搜索 （ **analyzer** 用于索引 ）。 有一个等 的 **default_search** 映射，用以指定索引 的 配置。

如果考 到 些 外参数，一个搜索 的完整 序会是下面 ：

- 自己定 的 `analyzer`, 否
- 字段映射里定 的 `search_analyzer`, 否
- 字段映射里定 的 `analyzer`, 否
- 索引 置中名 `default_search` 的分析器,
- 索引 置中名 `default` 的分析器,
- `standard` 准分析器

分析器配置 践

就可以配置分析器地方的数量而言是十分 人的，但是 非常 。

保持

多数情况下，会提前知道文 会包括 些字段。最 的途径就是在 建索引或者 加 型映射 ， 个全文字段 置分析器。 方式尽管有点麻 ，但是它 我 可以清楚的看到 个字段 个分析器是如何置的。

通常，多数字符串字段都是 `not_analyzed` 精 字段，比如 `(tag)` 或枚 (enum)，而且更多的全文字段会使用 的 `standard` 分析器或 `english` 或其他某 言的分析器。只需要 少数一 个字段指定自定 分析：或 `title` 字段需要以支持 入即 (*find-as-you-type*) 的方式 行索引。

可以在索引 置中， 大部分的字段 置 想指定的 `default` 分析器。然后在字段 置中， 某一 个字段配置需要指定的分析器。

NOTE

于和 相 的日志数据，通常的做法是 天自行 建索引，由于 方式不是从 建 的索引， 然可以用 索引模板 (Index Template) 新建的索引指定配置和映射。

被破坏的相 度！

在 更 的 多字段搜索 之前， 我 先快速解 一下 什 只在主分片上 建 索引。

用 会 不 的抱怨无法按相 度排序并提供 短的重 用 索引了一些文 ， 行一个 的，然后 明 低相 度的 果出 在高相 度 果之上。

了理解 什 会 ，可以 想，我 在 个主分片上 建了索引和 共 10 个文 ，其中 6 个文 有 `foo`。可能是分片 1 有其中 3 个 `foo` 文 ，而分片 2 有其中 外 3 个文 ， 句 ，所有文 是均 分布存 的。

在 什 是相 度？ 中，我 描述了 Elasticsearch 使用的相似度算法， 个算法叫做 /逆向文 率 或 TF/IDF 。 是 算某个 在当前被 文 里某个字段中出 的 率，出 的 率越高，文 越相 。 逆向文 率 将 某个 在索引内所有文 出 的百分数 考 在内，出 的 率越高，它的 重就越低。

但是由于性能原因， Elasticsearch 不会 算索引内所有文 的 IDF 。相反， 个分片会根据 分片 内的所有文 算一个本地 IDF 。

因 文 是均 分布存 的， 个分片的 IDF 是相同的。相反， 想如果有 5 个 `foo` 文 存于分片 1，而第

6 个文 存于分片 2 , 在 景下, `foo` 在一个分片里非常普通(所以不那 重要), 但是在 一个分片里非常出 很少(所以会 得更重要)。些 IDF 之 的差 会 致不正 的 果。

在 用中, 并不是一个 , 本地和全局的 IDF 的差 会随着索引里文 数的 多 消失, 在真 世界的数量下, 局部的 IDF 会被迅速均化, 所以上述 并不是相 度被破坏所 致的, 而是由于数据太少。

了 , 我 可以通 方式解决 个 。第一 是只在主分片上 建索引, 正如 `match` 里介 的那 , 如果只有一个分片, 那 本地的 IDF 就是 全局的 IDF。

第二个方式就是在搜索 求后添加 `?search_type=dfs_query_then_fetch` , `dfs` 是指 分布式 率搜索 (*Distributed Frequency Search*) , 它告 Elasticsearch , 先分 得 个分片本地的 IDF , 然后根据 果再 算整个索引的全局 IDF 。

TIP 不要在生 境上使用 `dfs_query_then_fetch` 。完全没有必要。只要有足 的数据就能保 是均 分布的。没有理由 个 外加上 DFS 。

多字段搜索

很少是 一句 的 `match` 匹配 。通常我 需要用相同或不同的字符串 一个或多个字段, 也就是 , 需要 多个 句以及它 相 度 分 行合理的合并。

有 候或 我 正 作者 Leo Tolstoy 写的一本名 *War and Peace* (争与和平)的 。或 我 正用 “minimum should match” (最少 匹配)的方式在文 中 或 面内容 行搜索, 或 我 正在搜索所有名字 John Smith 的用 。

在本章, 我 会介 造多 句搜索的工具及在特定 景下 采用的解决方案。

多字符串

最 的多字段 可以将搜索 映射到具体的字段。如果我 知道 *War and Peace* 是 , Leo Tolstoy 是作者, 很容易就能把 个条件用 `match` 句表示, 并将它 用 `bool` 合起来 :

```
GET /_search
{
  "query": {
    "bool": {
      "should": [
        { "match": { "title": "War and Peace" }},
        { "match": { "author": "Leo Tolstoy" }}
      ]
    }
  }
}
```

`bool` 采取 *more-matches-is-better* 匹配越多越好的方式, 所以 条 `match` 句的 分 果会被加在一起, 从而 个文 提供最 的分数 `_score` 。能与 条 句同 匹配的文 比只与一条 句匹配的文 得分要高。

当然，并不是只能使用 `match` 句：可以用 `bool` 来包含任意其他类型的句，甚至包括其他的 `bool`。我可以在上面的示例中添加一条 `bool` 句来指定或者版本的偏好：

```
GET /_search
{
  "query": {
    "bool": {
      "should": [
        { "match": { "title": "War and Peace" }},
        { "match": { "author": "Leo Tolstoy" }},
        { "bool": {
          "should": [
            { "match": { "translator": "Constance Garnett" }},
            { "match": { "translator": "Louise Maude" }}
          ]
        }}
      ]
    }
  }
}
```

什么将或者条件句放入一个独立的 `bool` 中？所有的四个 `match` 都是 `should` 句，所以为什么不将 `translator` 句与其他如 `title`、`author` 的句放在同一呢？

答案在于分的算方式。`bool` 行一个 `match`，再把分加在一起，然后将果与所有匹配的句数量相乘，最后除以所有的句数量。于同一的条句具有相同的重。在前面个例子中，包含 `translator` 句的 `bool`，只占分的三分之一。如果将 `translator` 句与 `title` 和 `author` 条句放入同一，那 `title` 和 `author` 句只献四分之一分。

句的先

前例中条句献三分之一分的方式可能并不是我想要的，我可能 `title` 和 `author` 条句更感趣，就需要整，使 `title` 和 `author` 句相来更重要。

在武器中，最容易使用的就是 `boost` 参数。了提升 `title` 和 `author` 字段的重，它分配的 `boost` 大于 1：

```

GET /_search
{
  "query": {
    "bool": {
      "should": [
        { "match": { ①
          "title": {
            "query": "War and Peace",
            "boost": 2
          }
        }},
        { "match": { ①
          "author": {
            "query": "Leo Tolstoy",
            "boost": 2
          }
        }},
        { "bool": { ②
          "should": [
            { "match": { "translator": "Constance Garnett" } },
            { "match": { "translator": "Louise Maude" } }
          ]
        }}
      ]
    }
  }
}

```

① title 和 author 句的 boost 2。

② 嵌套 bool 句 的 boost 1。

要 取 boost 参数 “最佳” , 的方式就是不断 : 定 boost , 行 , 如此反 。 boost 比 合理的区 于 1 到 10 之 , 当然也有可能是 15 。如果 boost 指定比 更高的 , 将不会 最 的 分 果 生更大影 , 因 分是被 一化的 (normalized) 。

字符串

bool 是多 句 的主干。它的 用 景很多, 特 是当需要将不同 字符串映射到不同字段的 候。

在于, 目前有些用 期望将所有的搜索 堆 到 个字段中, 并期望 用程序能 他 提供正 的 果。有意思的是多字段搜索的表 通常被称 高 (Advanced Search) —— 只是因 它 用 而言是高 的, 而多字段搜索的 却非常 。

于多 (multiword) 、多字段 (multifield) 来 , 不存在 的 万能 方案。 了 得最好 果, 需要 了解我 的数据, 并了解如何使用合 的工具。

了解我 的数据

当用 入了 个字符串 的 候, 通常会遇到以下三 情形 :

最佳字段

当搜索具体概念的时候，比如“brown fox”，比各自独立的更有意。像 `title` 和 `body` 的字段，尽管它们是相同的，但同时又彼此相互竞争。文本在相同字段中包含的越多越好，部分也来自于最匹配字段。

多数字段

为了相度行微，常用的一个技巧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。

主字段可能包括它的源、同音以及音或口音，被用来匹配尽可能多的文本。

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干提取的原词，一个字段包括其他源、口音，有一个字段可以提供相似性信息的瓦片（shingles）。

其他字段是作匹配个体文本提高相度分的信号，匹配字段越多越好。

混合字段

于某些主体，我需要在多个字段中指定其信息，各个字段都只能作整体的一部分：

- Person：`first_name` 和 `last_name`（人：名和姓）
- Book：`title`、`author` 和 `description`（书名：作者、描述）
- Address：`street`、`city`、`country` 和 `postcode`（地址：街道、市、国家和邮政编码）

在这些情况下，我希望在任何一些列出的字段中找到尽可能多的词，有如在一个大字段中进行搜索，一个大字段包括了所有列出的字段。

上述所有都是多、多字段，但每个具体字段都要求使用不同策略。本章后面的部分，我会依次介绍一个策略。

最佳字段

假设有网站允许用搜索博客的内容，以下面两篇博客内容为例：

```
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“Brown fox”然后点击搜索按钮。事先，我并不知道用哪个搜索词会在 `title` 或是在 `body` 字段中被找到，但是，用很有可能是想搜索相同的词。用肉眼判断，文字段 2 的匹配度更高，因为它同时包括要找的这个词：

在 行以下 `bool` :

```
{  
  "query": {  
    "bool": {  
      "should": [  
        { "match": { "title": "Brown fox" }},  
        { "match": { "body": "Brown fox" }}  
      ]  
    }  
  }  
}
```

但是我 的 果是文 1 的 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."  
      }  
    }  
  ]  
}
```

了理解 致 的原因，需要回想一下 `bool` 是如何 算 分的：

1. 它会 行 `should` 句中的 个 。
2. 加和 个 的 分。
3. 乘以匹配 句的 数。
4. 除以所有 句 数 (里 :2) 。

文 1 的 个字段都包含 `brown` 个 ，所以 个 `match` 句都能成功匹配并且有一个 分。文 2 的 `body` 字段同 包含 `brown` 和 `fox` 个 ，但 `title` 字段没有包含任何 。 ， `body` 果中的高分，加上 `title` 中的 0 分，然后乘以二分之一，就得到比文 1 更低的整体 分。

在本例中，`title` 和 `body` 字段是相互 竞争的 系，所以就需要 到 一个最佳匹配 的字段。

如果不是 将 个字段的 分 果加在一起，而是将 最佳匹配 字段的 分作 的整体 分，果会 ？ 返回的 果可能是： 同 包含 `brown` 和 `fox` 的 个字段比反 出 相同 的多个不同字段有更高的相 度。

dis_max

不使用 `bool` ， 可以使用 `dis_max` 即分 最大化 (*Disjunction Max Query*) 。分 (Disjunction) 的意思是 或 (or) ， 与可以把 合 (conjunction) 理解成 与 (and) 相 。分最大化 (Disjunction Max Query) 指的是： 将任何与任一 匹配的文 作 果返回，但只将最佳匹配的 分作 的 分 果返回：

```
{  
  "query": {  
    "dis_max": {  
      "queries": [  
        { "match": { "title": "Brown fox" }},  
        { "match": { "body": "Brown fox" }}  
      ]  
    }  
  }  
}
```

得到我 想要的 果：

```
{  
  "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

当用 搜索 “quick pets” 会 生什 ？在前面的例子中， 个文 都包含 `quick`，但是只有文 2 包含 `pets`， 个文 中都不具有同 包含 个 的相同字段。

如下，一个 的 `dis_max` 会采用 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, ①  
      "_source": {  
        "title": "Quick brown rabbits",  
        "body": "Brown rabbits are commonly seen."  
      }  
    },  
    {  
      "_id": "2",  
      "_score": 0.12713557, ①  
      "_source": {  
        "title": "Keeping pets healthy",  
        "body": "My quick brown fox eats rabbits on a regular basis."  
      }  
    }  
  ]  
}
```

① 注意 个 分是完全相同的。

我 可能期望同 匹配 `title` 和 `body` 字段的文 比只与一个字段匹配的文 的相 度更高，但事 并非如此，因 `dis_max` 只会 地使用 个 最佳匹配 句的 分 `_score` 作 整体 分。

`tie_breaker` 参数

可以通 指定 `tie_breaker` 个参数将其他匹配 句的 分也考 其中：

```
{
  "query": {
    "dis_max": {
      "queries": [
        { "match": { "title": "Quick pets" }},
        { "match": { "body": "Quick pets" }}
      ],
      "tie_breaker": 0.3
    }
  }
}
```

果如下：

```
{
  "hits": [
    {
      "_id": "2",
      "_score": 0.14757764, ①
      "_source": {
        "title": "Keeping pets healthy",
        "body": "My quick brown fox eats rabbits on a regular basis."
      }
    },
    {
      "_id": "1",
      "_score": 0.124275915, ①
      "_source": {
        "title": "Quick brown rabbits",
        "body": "Brown rabbits are commonly seen."
      }
    }
  ]
}
```

① 文 2 的相 度比文 1 略高。

`tie_breaker` 参数提供了一 `dis_max` 和 `bool` 之 的折中 ，它的 分方式如下：

1. 得最佳匹配 句的 分 `_score`。
2. 将其他匹配 句的 分 果与 `tie_breaker` 相乘。
3. 以上 分求和并 化。

有了 `tie_breaker`，会考 所有匹配 句，但最佳匹配 句依然占最 果里的很大一部分。

NOTE

`tie_breaker` 可以是 0 到 1 之 的浮点数，其中 0 代表使用 `dis_max` 最佳匹配句的普通，1 表示所有匹配句同等重要。最佳的精 需要根据数据与得出，但是合理 与零接近（于 0.1 - 0.4 之），就不会 覆 `dis_max` 最佳匹配性 的根本。

multi_match

`multi_match` 能在多个字段上反 行相同 提供了一 便捷方式。

NOTE

`multi_match` 多匹配 的 型有多 ，其中的三 恰巧与 了解我 的数据 中介 的三个 景 ，即：`best_fields` 、 `most_fields` 和 `cross_fields` （最佳字段、多数字段、跨字段）。

情况下， 的 型是 `best_fields`， 表示它会 个字段生成一个 `match` ，然后将它 合到 `dis_max` 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 个 用 `multi_match` 重写成更 的形式：

```
{
  "multi_match": {
    "query": "Quick brown fox",
    "type": "best_fields", ①
    "fields": [ "title", "body" ],
    "tie_breaker": 0.3,
    "minimum_should_match": "30%" ②
  }
}
```

① `best_fields` 型是 ，可以不指定。

② 如 `minimum_should_match` 或 `operator` 的参数会被 到生成的 `match` 中。

字段名称的模糊匹配

字段名称可以用模糊匹配的方式 出：任何与模糊模式正 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 匹配 `book_title` 、 `chapter_title` 和 `section_title` （ 名、章名、 名）三个字段：

```
{
  "multi_match": {
    "query": "Quick brown fox",
    "fields": "*_title"
  }
}
```

提升 个字段的 重

可以使用 ^ 字符 法 个字段提升 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" ] ①
  }
}
```

① `chapter_title` 个字段的 `boost` 2，而其他 个字段 `book_title` 和 `section_title` 字段的 `boost` 1。

多字段

全文搜索被称作是 召回率 (*Recall*) 与 精 率 (*Precision*) 的 ；召回率 ——返回所有的相 文 ；精 率 ——不返回无 文 。目的是在 果的第一 中 用 呈 最 相 的文 。

了提高召回率的效果，我 大搜索 ——不 返回与用 搜索 精 匹配的文 ， 会返回我 与 相 的所有文 。如果一个用 搜索 “quick brown box”，一个包含 fast foxes 的文 被

是非常合理的返回 果。

如果包含 **fast foxes** 的文 是能 到的唯一相 文 ，那 它会出 在 果列表的最上面，但是，如果有 100 个文 都出 了 **quick brown fox**，那 个包含 **fast foxes** 的文 当然会被 是次相 的，它可能 于返回 果列表更下面的某个地方。当包含了很多潜在匹配之后，我 需要将最匹配的几个置于 果列表的 部。

提高全文相 性精度的常用方式是 同一文本建立多 方式的索引， 方式都提供了一个不同的相 度信 号 *signal*。主字段会以尽可能多的形式去匹配尽可能多的文 。 个例子，我 可以 行以下操作：

- 使用 干提取来索引 **jumps**、**jumping** 和 **jumped** 的 ，将 **jump** 作 它 的 根形式。 即使用 搜索 **jumped**，也 是能 到包含 **jumping** 的匹配的文 。
- 将同 包括其中，如 **jump**、**leap** 和 **hop** 。
- 移除 音或口音：如 **ésta**、**está** 和 **esta** 都会以无 音形式 **esta** 来索引。

尽管如此，如果我 有 个文 ，其中一个包含 **jumped**，一个包含 **jumping**，用 很可能期望前者能排的更高，因 它正好与 入的搜索条件一致。

了 到目的，我 可以将相同的文本索引到其他字段从而提供更 精 的匹配。一个字段可能是 干未 提取 的版本， 一个字段可能是 音 的原始 ，第三个可能使用 **shingles** 提供 相似性 信息。 些附加的字段可以看成提高 个文 的相 度 分的信号 *signals*，能匹配字段的越多越好。

一个文 如果与广度匹配的主字段相匹配，那 它会出 在 果列表中。如果文 同 又与 *signal* 信号字段匹配，那 它会 得 外加分，系 会提升它在 果列表中的位置。

我 会在本 后 同 、 相似性、部分匹配以及其他潜在的信号 行 ，但 里只使用 干已提取 (stemmed) 和未提取 (unstemmed) 的字段作 例子来 明 技 。

多字段映射

首先要做的事情就是 我 的字段索引 次：一次使用 干模式以及一次非 干模式。 了做到 点，采用 **multifields** 来 ，已 在 **multifields** 有所介 ：

```

DELETE /my_index

PUT /my_index
{
  "settings": { "number_of_shards": 1 }, ①
  "mappings": {
    "my_type": {
      "properties": {
        "title": { ②
          "type": "string",
          "analyzer": "english",
          "fields": {
            "std": { ③
              "type": "string",
              "analyzer": "standard"
            }
          }
        }
      }
    }
  }
}

```

① 参考 被破坏的相 度。

② title 字段使用 english 英 分析器来提取 干。

③ title.std 字段使用 standard 准分析器，所以没有 干提取。

接着索引一些文 :

```

PUT /my_index/my_type/1
{ "title": "My rabbit jumps" }

PUT /my_index/my_type/2
{ "title": "Jumping jack rabbits" }

```

里用一个 match title 字段是否包含 jumping rabbits (跳 的兔子) :

```

GET /my_index/_search
{
  "query": {
    "match": {
      "title": "jumping rabbits"
    }
  }
}

```

因 有了 english 分析器， 个 是在 以 jump 和 rabbit 个被提取 的文 。 个文 的 title

字段都同 包括 个 , 所以 个文 得到的 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 `title.std` 字段, 那 只有文 2 是匹配的。尽管如此, 如果同 个字段, 然后使用 `bool` 将 分果合并, 那 个文 都是匹配的 (`title` 字段的作用), 而且文 2 的相 度分更高 (`title.std` 字段的作用) :

```
GET /my_index/_search  
{  
  "query": {  
    "multi_match": {  
      "query": "jumping rabbits",  
      "type": "most_fields", ①  
      "fields": [ "title", "title.std" ]  
    }  
  }  
}
```

① 我 希望将所有匹配字段的 分合并起来, 所以使用 `most_fields` 型。`multi_match` 用 `bool` 将 个字段 句包在里面, 而不是使用 `dis_max` 。

```
{
  "hits": [
    {
      "_id": "2",
      "_score": 0.8226396, ①
      "_source": {
        "title": "Jumping jack rabbits"
      }
    },
    {
      "_id": "1",
      "_score": 0.10741998, ①
      "_source": {
        "title": "My rabbit jumps"
      }
    }
  ]
}
```

① 文 2 在的 分要比文 1 高。

用广度匹配字段 `title` 包括尽可能多的文——以提升召回率——同 又使用字段 `title.std` 作 信号 将相 度更高的文 置于 果 部。

个字段 于最 分的 献可以通 自定 `boost` 来控制。比如，使 `title` 字段更 重要， 同 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" ] ①
    }
  }
}
```

① `title` 字段的 `boost` 的 10 使它比 `title.std` 更重要。

跨字段 体搜索

在 一 普遍的搜索模式：跨字段 体搜索（cross-fields entity search）。在如 `person`、`product` 或 `address`（人、 品或地址） 的 体中，需要使用多个字段来唯一 它的信息。`person` 体可能是 索引的：

```
{  
  "firstname": "Peter",  
  "lastname": "Smith"  
}
```

或地址：

```
{  
  "street": "5 Poland Street",  
  "city": "London",  
  "country": "United Kingdom",  
  "postcode": "W1V 3DG"  
}
```

与之前描述的 多字符串 很像，但 存在着巨大的区 。在 多字符串 中，我 个字段使用不同的字符串，在本例中，我 想使用 个字符串在多个字段中 行搜索。

我 的用 可能想搜索 “Peter Smith” 个人，或 “Poland Street W1V” 个地址， 些 出 在不同的字段中，所以如果使用 `dis_max` 或 `best_fields` 去 一个 最佳匹配字段 然是个 的方式。

的方式

依次 个字段并将 个字段的匹配 分 果相加，听起来真像是 `bool` :

```
{  
  "query": {  
    "bool": {  
      "should": [  
        { "match": { "street": "Poland Street W1V" }},  
        { "match": { "city": "Poland Street W1V" }},  
        { "match": { "country": "Poland Street W1V" }},  
        { "match": { "postcode": "Poland Street W1V" }}  
      ]  
    }  
  }  
}
```

个字段重 字符串会使 瞬 得冗 ，可以采用 `multi_match` ，将 `type` 置成 `most_fields` 然后告 Elasticsearch 合并所有匹配字段的 分：

```
{
  "query": {
    "multi_match": {
      "query": "Poland Street W1V",
      "type": "most_fields",
      "fields": [ "street", "city", "country", "postcode" ]
    }
  }
}
```

most_fields 方式的

用 most_fields 方式搜索也存在某些问题，一些问题并不会马上：

- 它是多数字段匹配任意一个的，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尾效应。
- 由于一个字段是唯一的，而且它之间的相互影响会导致不好的排序结果。

字段中心式

以上三个源于 most_fields 的问题都因它是字段中心式 (field-centric) 而不是术语中心式 (term-centric) 的：当真正感兴趣的匹配的时候，它只关心的是最匹配的字段。

NOTE best_fields 型也是字段中心式的，它也存在类似的问题。

首先看这些问题存在的原因，再想如何解决它们。

1：在多个字段中匹配相同的

回想一下 most_fields 是如何工作的：Elasticsearch 为每个字段生成独立的 match，再用 bool 将它们包起来。

可以通过 validate-query API 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，一个字段都与 `poland` 匹配的文 要比一个字段同 匹配 `poland` 与 `street` 文 的 分高。

2 : 剪掉 尾

在 [匹配精度](#) 中，我 使用 `and` 操作符或 置 `minimum_should_match` 参数来消除 果中几乎不相的 尾，或 可以 以下方式：

```
{
  "query": {
    "multi_match": {
      "query": "Poland Street W1V",
      "type": "most_fields",
      "operator": "and", ①
      "fields": [ "street", "city", "country", "postcode" ]
    }
  }
}
```

① 所有 必 呈 。

但是 于 `best_fields` 或 `most_fields` 些参数会在 `match` 生成 被 入， 个 的 `explanation` 解 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)
```

句 ， 使用 `and` 操作符要求所有 都必 存在于 相同字段 ， 然是不 的！可能就不存在能与 个 匹配的文 。

3 :

在 [什 是相](#) 中，我 解 个 使用 TF/IDF 相似度算法 算相 度 分：

一个 在 个文 的某个字段中出 的 率越高， 个文 的相 度就越高。

逆向文 率

一个 在所有文 某个字段索引中出 的 率越高， 个 的相 度就越低。

当搜索多个字段 ， TF/IDF 会 来某些令人意外的 果。

想想用字段 `first_name` 和 `last_name` “Peter Smith”的例子，Peter 是个平常的名 Smith 也是平常的姓，二者都具有低的 IDF。但当索引中有外一个人的名字是“Smith Williams”，Smith 作名来很不平常，以致它有一个高的 IDF！

下面一个的可能会在果中将“Smith Williams”置于“Peter Smith”之上，尽管事实上是第二个人比第一个人更匹配。

```
{  
  "query": {  
    "multi_match": {  
      "query": "Peter Smith",  
      "type": "most_fields",  
      "fields": ["*_name"]  
    }  
  }  
}
```

里的 `smith` 在名字段中具有高 IDF，它会削弱“Peter”作名和“Smith”作姓低 IDF 的所起作用。

解决方案

存在一些是因我在理着多个字段，如果将所有这些字段合成一个字段，就会消失。可以 `person` 文添加 `full_name` 字段来解决这个问题：

```
{  
  "first_name": "Peter",  
  "last_name": "Smith",  
  "full_name": "Peter Smith"  
}
```

当 `full_name` 字段：

- 具有更多匹配的文会比只有一个重匹配的文更重要。
- `minimum_should_match` 和 `operator` 参数会像期望那样工作。
- 姓和名的逆向文率被合并，所以 Smith 到底是作姓是作名出，都会得无必要。

做当然是可行的，但我并不太喜欢存冗余数据。取而代之的是 Elasticsearch 可以提供一个解决方案——一个在索引，而一个是在搜索——随后会把它。

自定 `_all` 字段

在 `all-field` 字段中，我解`_all`字段的索引方式是将所有其他字段的作一个大字符串索引的。然而做并不十分活，为了活我可以人名添加一个自定`_all`字段，再地址添加一个`_all`字段。

Elasticsearch 在字段映射中我提供 `copy_to` 参数来这个功能：

```
PUT /my_index
{
  "mappings": {
    "person": {
      "properties": {
        "first_name": {
          "type": "string",
          "copy_to": "full_name" ①
        },
        "last_name": {
          "type": "string",
          "copy_to": "full_name" ①
        },
        "full_name": {
          "type": "string"
        }
      }
    }
  }
}
```

① `first_name` 和 `last_name` 字段中的 值会被 复制到 `full_name` 字段。

有了 一个映射，我 可以用 `first_name` 来 姓，用 `last_name` 来 姓，或者直接使用 `full_name` 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 一个字段的内容复制到本地，然后根据 `full_name` 的映射自行索引。

`copy_to` 置 `multi-field`无效。如果配置映射，Elasticsearch 会常。

什？多字段只是以不同方式索引“主”字段；它没有自己的数据源。也就是没有可供 `copy_to` 到一字段的数据源。

只要“主”字段 `copy_to` 就能而易的到相同的效果：

```
PUT /my_index
{
  "mappings": {
    "person": {
      "properties": {
        "first_name": {
          "type": "string",
          "copy_to": "full_name", ①
        },
        "fields": {
          "raw": {
            "type": "string",
            "index": "not_analyzed"
          }
        }
      },
      "full_name": {
        "type": "string"
      }
    }
  }
}
```

WARNING

① `copy_to` 是“主”字段，而不是多字段的

cross-fields 跨字段

自定 `all` 的方式是一个好的解决方案，只需在索引文前其置好映射。不，在搜索提供了相的解决方案：使用 `cross_fields` 型行 `multi_match`。`cross_fields` 使用中心式 (term-centric) 的方式，与 `best_fields` 和 `most_fields` 使用字段中心式 (field-centric) 的方式非常不同，它将所有字段当成一个大字段，并在一个字段中一个。

了明字段中心式 (field-centric) 与中心式 (term-centric) 方式的不同，先看看以下字段中心式的 `most_fields` 的 explanation 解：

```

GET /_validate/query?explain
{
  "query": {
    "multi_match": {
      "query": "peter smith",
      "type": "most_fields",
      "operator": "and", ①
      "fields": [ "first_name", "last_name" ]
    }
  }
}

```

① 所有 都是必 的。

于匹配的文 , `peter` 和 `smith`都必 同 出 在相同字段中, 要 是 `first_name` 字段, 要 `last_name` 字段 :

```

(+first_name:peter +first_name:smith)
(+last_name:peter +last_name:smith)

```

中心式会使用以下 :

```

+(first_name:peter last_name:peter)
+(first_name:smith last_name:smith)

```

句 , `peter` 和 `smith`都必 出 , 但是可以出 在任意字段中。

`cross_fields` 型首先分析 字符串并生成一个 列表, 然后它从所有字段中依次搜索 个 。不同的搜索方式很自然的解决了 [字段中心式](#) 三个 中的二个。剩下的 是逆向文率不同。

幸 的是 `cross_fields` 型也能解决 个 , 通 `validate-query` 可以看到 :

```

GET /_validate/query?explain
{
  "query": {
    "multi_match": {
      "query": "peter smith",
      "type": "cross_fields", ①
      "operator": "and",
      "fields": [ "first_name", "last_name" ]
    }
  }
}

```

① 用 `cross_fields` 中心式匹配。

它通过混合不同字段逆向索引文率的方式解决了 的：

```
+blended("peter", fields: [first_name, last_name])
+blended("smith", fields: [first_name, last_name])
```

句，它会同时在 `first_name` 和 `last_name` 个字段中 `smith` 的 IDF，然后用者的最小作一个字段的 IDF。果上就是 `smith` 会被既是个平常的姓，也是平常的名。

了 `cross_fields` 以最 方式工作，所有的字段都使用相同的分析器，具有相同分析器的字段会被分在一起作混合字段使用。

如果包括了不同分析的字段，它会以 `best_fields` 的相同方式被加入到果中。例如：我将 `title` 字段加到之前的 中（假 它使用的是不同的分析器），`explanation` 的解 果如下：

NOTE

```
(+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，点尤 重要。

按字段提高重

采用 `cross_fields` 与 自定 `_all` 字段 相比，其中一个 就是它可以在搜索 个字段提升重。

像 `first_name` 和 `last_name` 具有相同的字段并不是必 的，但如果要用 `title` 和 `description` 字段搜索，可能希望 `title` 分配更多的 重，同 可以使用前面介 的 ^ 符号 法来：

```
GET /books/_search
{
  "query": {
    "multi_match": {
      "query": "peter smith",
      "type": "cross_fields",
      "fields": [ "title^2", "description" ] ①
    }
  }
}
```

① `title` 字段的 重提升 2，`description` 字段的 重提升 1。

自定 字段 是否能 于多字段 ，取决于在多字段 与 字段自定 `_all` 之 代 的衡，即 解决方案会 来更大的性能 化就 一。

Exact-Value 精 字段

在 束多字段 个 之前，我 最后要 的是精 `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。

原因可以通 看 的 explanation 解 得到， 想将 `title` 字段 置成 `not_analyzed`：

```
GET /_validate/query?explain
{
  "query": {
    "multi_match": {
      "query": "peter smith",
      "type": "cross_fields",
      "fields": [ "title", "first_name", "last_name" ]
    }
  }
}
```

因 `title` 字段是未分析 的，Elasticsearch 会将“peter smith” 个完整的字符串作 条件来搜索！

```
title:peter smith
(
  blended("peter", fields: [first_name, last_name])
  blended("smith", fields: [first_name, last_name])
)
```

然 个 不在 `title` 的倒排索引中，所以需要在 `multi_match` 中避免使用 `not_analyzed` 字段。

近似匹配

使用 TF/IDF 的 准全文 索将文 或者文 中的字段作一大袋的 理。`match` 可以告知我 大袋子中是否包含 的 条，但却无法告知 之 的 系。

思考下面 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 都会得到匹配，但它却不能 定 个 是否只来自于一 境，甚至都不能 定是否来自于同一个段落。

理解分 之 的 系是一个 的 ，我 也无法通 一 方式去解决。但我 至少可以通 出 在彼此附近或者 是彼此相 的分 来判断一些似乎相 的分 。

个文 可能都比我 上面 个例子要 : `Sue` 和 `alligator` 个 可能会分散在其他的段落文字中，我 可能会希望得到尽可能包含 个 的文 ，但我 也同 需要 些

文 与分 有很高的相 度。

就是短 匹配或者近似匹配的所属 域。

TIP 在 一 章 , 我 是 使用 在 `match` 中 使用 的 文 作 例 子。

短 匹 配

就 像 `match` 于 全 文 索 是 一 最 常 用 的 一 , 当 想 到 彼 此 近 搜 索 的 方 法 , 就 会 想 到 `match_phrase` 。

```
GET /my_index/my_type/_search
{
  "query": {
    "match_phrase": {
      "title": "quick brown fox"
    }
  }
}
```

似 `match` , `match_phrase` 首先将 字符串 解析成一个 列 表 , 然后 些 行 搜 索 , 但 只 保 留 那 些 包 含 全 部 搜 索 , 且 位 置 与 搜 索 相 同 的 文 。 比 如 于 `quick fox` 的 短 搜 索 可 能 不 会 匹 配 到 任 何 文 , 因 没 有 文 包 含 的 `quick` 之 后 跟 着 `fox` 。

`match_phrase` 同 可 写 成 一 型 `phrase` 的 `match` :

TIP

```
"match": {
  "title": {
    "query": "quick brown fox",
    "type": "phrase"
  }
}
```

的 位 置

当 一 个 字 符 串 被 分 后 , 一 个 分 析 器 不 但 会 返回 一 个 列 表 , 而 且 会 返回 各 在 原 始 字 符 串 中 的 位 置 或 者 序 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 ①
    },
    {
      "token": "brown",
      "start_offset": 6,
      "end_offset": 11,
      "type": "<ALPHANUM>",
      "position": 2 ①
    },
    {
      "token": "fox",
      "start_offset": 12,
      "end_offset": 15,
      "type": "<ALPHANUM>",
      "position": 3 ①
    }
  ]
}
```

① position 代表各 在原始字符串中的位置。

位置信息可以被存 在倒排索引中，因此 **match_phrase** 位置敏感的，
就可以利用位置信息去匹配包含所有 ，且各 序也与我 搜索指定一致的文 中 不 其他 。

什 是短

一个被 定 和短 **quick brown fox** 匹配的文 ，必 足以下 些要求：

- **quick**、**brown** 和 **fox** 需要全部出 在域中。
- **brown** 的位置 比 **quick** 的位置大 1。
- **fox** 的位置 比 **quick** 的位置大 2。

如果以上任何一个 不成立， 文 不能 定 匹配。

本上来，`match_phrase`是利用一低的span族(query family)去做位置敏感的匹配。Span是一的，所以它没有分段；它只指定的行精搜索。

TIP

得幸的是，`match_phrase`已足秀，大多数人是不会直接使用span。然而，在一些域，例如利索，是会采用低去行非常具体而又精心造的位置搜索。

混合起来

精短匹配或是于格了。也我想要包含`quick brown fox''`的文也能匹配`quick fox,"`，尽管情形不完全相同。

我能通使用`slop`参数将活度引入短匹配中：

```
GET /my_index/my_type/_search
{
  "query": {
    "match_phrase": {
      "title": {
        "query": "quick fox",
        "slop": 1
      }
    }
  }
}
```

`slop`参数告`match_phrase`条相隔多然能将文匹配。相隔多的意思是了和文匹配需要移条多少次？

我以一个的例子始。了`quick fox`能匹配一个包含`quick brown fox`的文，我需要`slop`的1:

	Pos 1	Pos 2	Pos 3
<hr/>			
Doc:	quick	brown	fox
<hr/>			
Query:	quick	fox	
Slop 1:	quick		fox

尽管在使用了`slop`短匹配中所有的都需要出，但是些也不必了匹配而按相同的序列排列。有了足大的`slop`，就能按照任意序排列了。

了使`fox quick`匹配我的文，我需要`slop`的3:

	Pos 1	Pos 2	Pos 3
Doc:	quick	brown	fox
Query:	fox	quick	
Slop 1:	fox quick	①	
Slop 2:	quick	fox	
Slop 3:	quick		fox

① 注意 `fox` 和 `quick` 在 中占据同 的位置。因此将 `fox quick` 序成 `quick fox` 需要 ，或者 2 的 `slop`。

多 字段

多 字段使用短 匹配 会 生奇怪的事。想象一下 索引 个文：

```
PUT /my_index/groups/1
{
  "names": [ "John Abraham", "Lincoln Smith"]
}
```

然后 行一个 `Abraham Lincoln` 的短：

```
GET /my_index/groups/_search
{
  "query": {
    "match_phrase": {
      "names": "Abraham Lincoln"
    }
  }
}
```

令人 的是， 即使 `Abraham` 和 `Lincoln` 在 `names` 数 里属于 个不同的人名， 我 的文 也匹配了 。 一切的原因在Elasticsearch数 的索引方式。

在分析 `John Abraham` 的 候， 生了如下信息：

- Position 1: `john`
- Position 2: `abraham`

然后在分析 `Lincoln Smith` 的 候， 生了：

- Position 3: `lincoln`
- Position 4: `smith`

句 ， Elasticsearch 以上数 分析生成了与分析 个字符串 `John Abraham Lincoln Smith` —

几乎完全相同的元。我的示例相的 lincoln 和 abraham，而且一个条存在，并且它正好相，所以一个匹配了。

幸的是，在的情况下有一叫做 position_increment_gap 的的解决方案，它在字段映射中配置。

```
DELETE /my_index/groups/ ①
```

```
PUT /my_index/_mapping/groups ②
```

```
{  
    "properties": {  
        "names": {  
            "type": "string",  
            "position_increment_gap": 100  
        }  
    }  
}
```

①首先除映射 groups 以及一个型内的所有文。

②然后建一个有正的新的映射 groups。

position_increment_gap 置告 Elasticsearch 数中个新元素加当前条 position 的指定。所以在我再索引 names 数，会生如下的果：

- Position 1: john
- Position 2: abraham
- Position 103: lincoln
- Position 104: smith

在我短可能无法匹配文因 abraham 和 lincoln 之距 100。为了匹配个文必添加 100 的 slop。

越近越好

于一个短排除了不包含切短的文，而近一个 `slop` 大于 0 的短将条的近度考到最相度 `_score` 中。通常置一个像 50 或者 100 的高 `slop`，能排除距太的文，但是也予了那些近的文更高的分数。

下列 quick dog 的近匹配了同包含 quick 和 dog 的文，但是也与 quick 和 dog 更加近的文更高的分数：

```

POST /my_index/my_type/_search
{
  "query": {
    "match_phrase": {
      "title": {
        "query": "quick dog",
        "slop": 50 ①
      }
    }
  }
}

```

① 注意高 slop。

```

{
  "hits": [
    {
      "_id": "3",
      "_score": 0.75, ①
      "_source": {
        "title": "The quick brown fox jumps over the quick dog"
      }
    },
    {
      "_id": "2",
      "_score": 0.28347334, ②
      "_source": {
        "title": "The quick brown fox jumps over the lazy dog"
      }
    }
  ]
}

```

① 分数 高因 `quick` 和 `dog` 很接近

② 分数 低因 `quick` 和 `dog` 分

使用 近度提高相 度

然 近 很有用，但是所有 条都出 在文 的要求 于 格了。我 全文搜索 一章的 控制精度 也是同 的：如果七个 条中有六个匹配，那 个文 用 而言就已 足 相 了，但是 `match_phrase` 可能会将它排除在外。

相比将使用 近匹配作 要求，我 可以将它作 信号— 使用， 作 多潜在 中的一个，会 个文 的最 分 做出 献(参考 多数字段)。

上我 想将多个 的分数累 起来意味着我 用 `bool` 将它 合并。

我 可以将一个 的 `match` 作 一个 `must` 子句。 个 将决定 些文 需要被包含到 果集中。

我可以用 `minimum_should_match` 参数去除尾。然后我可以以 `should` 子句的形式添加更多特定。一个匹配成功的都会加匹配文的相度。

```
GET /my_index/my_type/_search
{
  "query": {
    "bool": {
      "must": {
        "match": { ①
          "title": {
            "query": "quick brown fox",
            "minimum_should_match": "30%"
          }
        }
      },
      "should": {
        "match_phrase": { ②
          "title": {
            "query": "quick brown fox",
            "slop": 50
          }
        }
      }
    }
  }
}
```

① `must` 子句从果集中包含或者排除文本。

② `should` 子句加入了匹配到文本的相度分。

我当然可以在 `should` 子句里面添加其它的，其中一个只某一特定方面的相度。

性能化

短和近都比的 `query` 代价更高。一个 `match` 是看条是否存在倒排索引中，而一个 `match_phrase` 是必算并比多个可能重的位置。

[Lucene nightly benchmarks](#) 表明一个的 `term` 比一个短大快 10 倍，比近（有 `slop` 的短）大快 20 倍。当然，一个代价指的是在搜索而不是索引。

通常，短的外成本并不像些数字所暗示的那人。事实上，性能上的差距只是明一个的 `term` 有多快。准全文数据的短通常在几秒内完成，因此上都是完全可用，即使是在一个繁忙的集群上。

TIP

在某些特定病理案例下，短可能成本太高了，但比少。一个典型例子就是DNA序列，在序列里很多同的在很多位置重出。在里使用高 `slop` 会到致位置算大量加。

那我如何限制短和近的性能消耗？一有用的方法是少需要通短

的文 数。

果集重新 分

在先前的章 中 ，我 了而使用 近 来 整相 度，而不是使用它将文 从 果列表中添加或者排除。一个 可能会匹配成千上万的 果，但我 的用 很可能只 果的前几 感 趣。

一个 的 `match` 已 通 排序把包含所有含有搜索 条的文 放在 果列表的前面了。事 上，我 只想 些 部文 重新排序，来 同 匹配了短 的文 一个 外的相 度升 。

`search` API 通 重新 分 明 支持 功能。重新 分 段支持一个代 更高的 分算法—比如 `phrase` 一只是 了从 个分片中 得前 `K` 个 果。然后会根据它 的最新 分 重新排序。

求如下所示：

```
GET /my_index/my_type/_search
{
  "query": {
    "match": { ①
      "title": {
        "query": "quick brown fox",
        "minimum_should_match": "30%"
      }
    },
    "rescore": {
      "window_size": 50, ②
      "query": { ③
        "rescore_query": {
          "match_phrase": {
            "title": {
              "query": "quick brown fox",
              "slop": 50
            }
          }
        }
      }
    }
  }
}
```

① `match` 决定 些文 将包含在最 果集中，并通 TF/IDF 排序。

② `window_size` 是 一分片 行重新 分的 部文 数量。

③ 目前唯一支持的重新打分算法就是 一个 ，但是以后会有 加更多的算法。

相

短 和 近 都很好用，但 有一个 点。它 于 格了： 了匹配短 ，所有 都必 存

在，即使使用了 `slop`。

用 `slop` 得到的 序的 活性也需要付出代，因 失去了 之 的 系。即使可以 `sue`、`alligator` 和 `ate` 相 出 的文，但无法分 是 `Sue ate` 是 `alligator ate`。

当 相互 合使用的 候，表 的含 比 独使用更 富。 个子句 `I'm not happy I'm working` 和 `I'm happy I'm not working` 包含相同的 ，也 有相同的 近度，但含 截然不同。

如果索引 而不是索引独立的 ，就能 些 的上下文尽可能多的保留。

句子 `Sue ate the alligator`，不 要将 一个 （或者 *unigram*）作 索引

```
["sue", "ate", "the", "alligator"]
```

也要将 个 以及它的 近 作 个 索引：

```
["sue ate", "ate the", "the alligator"]
```

些 （或者 *bigrams*）被称 *shingles*。

Shingles 不限于 ； 也可以索引三个 （*trigrams*）：

TIP

```
["sue ate the", "ate the alligator"]
```

Trigrams 提供了更高的精度，但是也大大 加了索引中唯一的数量。在大多数情况下，Bigrams 就 了。

当然，只有当用 入的 内容和在原始文 中 序相同，shingles 才是有用的； `sue alligator` 的 可能会匹配到 个 ，但是不会匹配任何 shingles。

幸 的是，用 向于使用和搜索数据相似的 造来表 搜索意 。但 一点很重要：只是索引 bigrams 是不 的；我 然需要 unigrams，但可以将匹配 bigrams 作 加相 度 分的信号。

生成 Shingles

Shingles 需要在索引 作 分析 程的一部分被 建。我 可以将 unigrams 和 bigrams 都索引到 个字段中， 但将它 分 保存在能被独立 的字段会更清晰。unigrams 字段将 成我 搜索的基 部分，而 bigrams 字段用来提高相 度。

首先，我 需要在 建分析器 使用 `shingle` 元 器：

```

DELETE /my_index

PUT /my_index
{
  "settings": {
    "number_of_shards": 1, ①
    "analysis": {
      "filter": {
        "my_shingle_filter": {
          "type": "shingle",
          "min_shingle_size": 2, ②
          "max_shingle_size": 2, ②
          "output_unigrams": false ③
        }
      },
      "analyzer": {
        "my_shingle_analyzer": {
          "type": "custom",
          "tokenizer": "standard",
          "filter": [
            "lowercase",
            "my_shingle_filter" ④
          ]
        }
      }
    }
  }
}

```

① 参考 被破坏的相 度！。

② 最小/最大的 shingle 大小是 2，所以 上不需要 置。

③ shingle 元 器 出 unigrams，但是我 想 unigrams 和 bigrams 分 。

④ my_shingle_analyzer 使用我 常 的 my_shingles_filter 元 器。

首先，用 analyze API 下分析器：

```

GET /my_index/_analyze?analyzer=my_shingle_analyzer
Sue ate the alligator

```

果然，我 得到了 3 个：

- sue ate
- ate the
- the alligator

在我 可以 建一个使用新的分析器的字段。

多字段

我曾到将 unigrams 和 bigrams 分索引更清晰，所以 `title` 字段将建成一个多字段（参考[字符串排序与多字段](#)）：

```
PUT /my_index/_mapping/my_type
{
  "my_type": {
    "properties": {
      "title": {
        "type": "string",
        "fields": {
          "shingles": {
            "type": "string",
            "analyzer": "my_shingle_analyzer"
          }
        }
      }
    }
  }
}
```

通过一个映射，JSON 文中的 `title` 字段将会被以 unigrams (`title`) 和 bigrams (`title.shingles`) 被索引，意味着可以独立地一些字段。

最后，我可以索引以下示例文：

```
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，我首先来看 `The hungry alligator ate Sue` 行的后果：

```

GET /my_index/my_type/_search
{
  "query": {
    "match": {
      "title": "the hungry alligator ate sue"
    }
  }
}

```

↑ 返回了所有的三个文档，但是注意文档 1 和 2 有相同的相 度 分因为它们包含了相同的：

```

{
  "hits": [
    {
      "_id": "1",
      "_score": 0.44273707, ①
      "_source": {
        "title": "Sue ate the alligator"
      }
    },
    {
      "_id": "2",
      "_score": 0.44273707, ①
      "_source": {
        "title": "The alligator ate Sue"
      }
    },
    {
      "_id": "3", ②
      "_score": 0.046571054,
      "_source": {
        "title": "Sue never goes anywhere without her alligator skin purse"
      }
    }
  ]
}

```

① 三个文档都包含 `the`、`alligator` 和 `ate`，所以 得相同的 分。

② 我 可以通过 置 `minimum_should_match` 参数排除文档 3，参考 [控制精度](#)。

在 在 里添加 `shingles` 字段。不要忘了在 `shingles` 字段上的匹配是充当一个信号— 为了提高相 度 分—所以我 然需要将基本 `title` 字段包含到 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"
        }
      }
    }
  }
}

```

然匹配到了所有的 3 个文 , 但是文 2 在排到了第一名因 它匹配了 shingled 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 包含的 hungry 没有在任何文 中出 , 我 然使用 近度返回了最相 的文 。

Performance性能

shingles 不 比短 更 活，而且性能也更好。 shingles 跟一个 的 `match` 一高效，而不用 次搜索花 短 的代 。只是在索引期 因 更多 需要被索引会付出一些小的代， 也意味着有 shingles 的字段会占用更多的磁 空 。 然而，大多数 用写入一次而取多次，所以在索引期 化我 的 速度是有意 的。

是一个在 Elasticsearch 里会 常 到的：不需要任何前期 行 多的 置，就能 在搜索的候有很好的效果。 一旦更清晰的理解了自己的需求，就能在索引 通 正 的 的数据建模 得更好果和性能。

部分匹配

敏 的 者会注意，目前 止本 介 的所有 都是 整个 的操作。 了能匹配，只能 倒排索引中存在的 ，最小的 元 个 。

但如果想匹配部分而不是全部的 ？ 部分匹配 允 用 指定 的一部分并 出所有包含部分片段的 。

与想象的不太一 ， 行部分匹配的需求在全文搜索引擎并不常 ，但是如果 者有 方面的背景，可能会在某个 候 一个 低效的全文搜索 用下面的 SQL 句 全文 行搜索：

```
WHERE text LIKE "%quick%"  
      AND text LIKE "%brown%"  
      AND text LIKE "%fox%" ①
```

① `fox` 会与 “fox” 和 “foxes” 匹配。

当然， Elasticsearch 提供分析 程，倒排索引 我 不需要使用 粗 的技 。 了能 同 匹配 “fox” 和 “foxes” 的情况，只需 的将它 的 干作 索引形式，没有必要做部分匹配。

也就是 ， 在某些情况下部分匹配会比 有用，常 的 用如下：

- 匹配 、 品序列号或其他 `not_analyzed` 未分析 ， 些 可以是以某个特定前始，也可以是与某 模式匹配的，甚至可以是与某个正 式相匹配的。
- 入即搜索 (*search-as-you-type*) ——在用 入搜索 程的同 就呈 最可能的 果。
- 匹配如 或荷 有 合 的 言，如：*Weltgesundheitsorganisation* （世界 生， 英文 World Health Organization）。

本章始于 `not_analyzed` 精 字段的前 匹配。

与 化数据

我 会使用美国目前使用的 形式（United Kingdom postcodes 标准）来 明如何用部分匹配化数据。 形式有很好的 定 。例如， `W1V 3DG` 可以分解成如下形式：

- `W1V` : 是 的外部，它定 了 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**：内部定了街道或建筑：

- **3** 代表街区区（1个数字）
- **DG** 代表元（2个字母）

假设将作 **not_analyzed** 的精字段索引，所以可以为其建索引，如下：

```
PUT /my_index
{
  "mappings": {
    "address": {
      "properties": {
        "postcode": {
          "type": "string",
          "index": "not_analyzed"
        }
      }
    }
  }
}
```

然后索引一些：

```
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" }
```

在这些数据已可。

prefix 前

到了所有以 **W1** 始的，可以使用 **prefix**：

```

GET /my_index/address/_search
{
  "query": {
    "prefix": {
      "postcode": "W1"
    }
  }
}

```

prefix 是一个 的底 的 ，它不会在搜索之前分析 字符串，它假定 入前 就正是要的前 。

TIP 状 下， **prefix** 不做相 度 分 算，它只是将所有匹配的文 返回，并 条果 予 分 1 。它的行 更像是 器而不是 。 **prefix** 和 **prefix** 器者 的区 就是 器是可以被 存的，而 不行。

之前已 提：“只能在倒排索引中 到存在的 ”，但我 并没有 些 的索引 行特殊 理， 个是以它 精 的方式存在于 个文 的索引中，那 **prefix** 是如何工作的 ？

回想倒排索引包含了一个有序的唯一 列表（本例是 ）。 于 个 ，倒排索引都会将包含 的文 ID 列入倒排表（*postings list*）。与示例 的倒排索引是：

Term:	Doc IDs:
"SW5 0BE"	5
"W1F 7HW"	3
"W1V 3DG"	1
"W2F 8HW"	2
"WC1N 1LZ"	4

了支持前 匹配， 会做以下事情：

1. 描 列表并 到第一个以 **W1** 始的 。
2. 搜集 的文 ID 。
3. 移 到下一个 。
4. 如果 个 也是以 **W1** ， 跳回到第二 再重 行，直到下一个 不以 **W1** 止。

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 都是以 **W1** ， 前 需要 个 然后 算 果！

前 越短所需 的 越多。如果我 要以 **W** 作 前 而不是 **W1** ，那 就可能需要做千万次的匹配。

CAUTION **prefix** 或 于一些特定的匹配是有效的，但使用方式 是 当注意。当字段中 的集合很小 ，可以放心使用，但是它的伸 性并不好，会 我 的集群 来很多 力。可以使用 的前 来限制 影 ， 少需要 的量。

本章后面会介绍一个索引的解决方案，一个方案能使前匹配更高效，不在此之前，需要先看看一个相似的：`wildcard` 和 `regexp`（模糊和正式）。

通配符与正表式

与 `prefix` 前的特性类似，`wildcard` 通配符也是一底基于的，与前不同的是它允许指定匹配的正式。它使用标准的 shell 通配符：`? 匹配任意字符，* 匹配 0 或多个字符。`

一个会匹配包含 `W1F 7HW` 和 `W2F 8HW` 的文：

```
GET /my_index/address/_search
{
  "query": {
    "wildcard": {
      "postcode": "W?F*HW" ①
    }
  }
}
```

① `? 匹配 1 和 2，* 与空格及 7 和 8 匹配。`

想如果只想匹配 `W` 区域的所有，前匹配也会包括以 `WC` 的所有，与通配符匹配到的相似，如果想匹配只以 `W` 始并跟随一个数字的所有，`regexp 正式` 允写出更的模式：

```
GET /my_index/address/_search
{
  "query": {
    "regexp": {
      "postcode": "W[0-9].+" ①
    }
  }
}
```

① 一个正表式要求必以 `W`，跟 0 至 9 之任何一个数字，然后接一或多个其他字符。

`wildcard` 和 `regexp` 的工作方式与 `prefix` 完全一，它也需要倒排索引中的列表才能到所有匹配的，然后依次取一个相的文 ID，与 `prefix` 的唯一不同是：它能支持更的匹配模式。

这意味着需要同注意前存在性能，有很多唯一的字段行些可能会消耗非常多的源，所以要避免使用左通配的模式匹配（如：`*foo` 或 `.*foo` 的正式）。

数据在索引的理有助于提高前匹配的效率，而通配符和正表式只能在完成，尽管些有其用景，但使用需慎。

`prefix`、`wildcard` 和 `regexp` 是基于 `analyzed` 操作的，如果用它来分析字段，它会分析字段里面的各个词，而不是将字段作整体来处理。

比方包含“Quick brown fox”（快速的棕色狐狸）的 `title` 字段会生成：`quick`、`brown` 和 `fox`。

会匹配以下一个：

```
{ "regexp": { "title": "br.*" }}
```

CAUTION

但是不会匹配以下一个：

```
{ "regexp": { "title": "Qu.*" } } ①  
{ "regexp": { "title": "quick br*" } } ②
```

① 在索引里的 `Qu` 是 `quick` 而不是 `Quick`。

② `quick` 和 `brown` 在表中是分词的。

入即搜索

把事情先放一旁，我先看看前文是如何在全文搜索中起作用的。用已分析的内容之前，就能扩展搜索结果，这就是所谓的即时搜索 (*instant search*) 或入即搜索 (*search-as-you-type*)。不用能在更短的时间内得到搜索结果，我也能引用搜索索引中真正存在的结果。

例如，如果用入 `johnnie walker bl`，我希望在它完成全文搜索条件前就能得到：Johnnie Walker Black Label 和 Johnnie Walker Blue Label。

生活是美好的，就像猫的花色不只一种！我希望能做到一蹴而就的方式。并不需要对数据做任何准备，在任何时候就能任意的全文字段进行入即搜索 (*search-as-you-type*) 的操作。

在短匹配中，我引入了 `match_phrase` 短匹配，它匹配相序一致的所有指定词于入即搜索，可以使用 `match_phrase` 的一个特殊形式，`match_phrase_prefix`：

```
{  
  "match_phrase_prefix": {  
    "brand": "johnnie walker bl"  
  }  
}
```

的行与 `match_phrase` 一致，不同的是它将字符串的最后一个词放在前使用，句，可以将之前的例子看成如下：

- `johnnie`
- 跟着 `walker`

- 跟着以 `bl` 始的

如果通过 `validate-query API` 行一个 , explanation 的结果 :

```
"johnnie walker bl*"
```

与 `match_phrase` 一样，它也可以接受 `slop` 参数（参照 `slop`）相 序位置不严格：

```
{
  "match_phrase_prefix": {
    "brand": {
      "query": "walker johnnie bl", ①
      "slop": 10
    }
  }
}
```

① 尽管 序不正，然能匹配，因 我 它 置了足 高的 `slop` 使匹配 的序有更大的 活性。

但是只有 字符串的最后一个 才能当作前 使用。

在之前的 前 中，我 警告 使用前 的 ，即 `prefix` 存在 重的 源消耗 ，短的方式也同 如此。前 `a` 可能会匹配成千上万的 ， 不 会消耗很多系 源，而且 果的用也不大。

可以通 置 `max_expansions` 参数来限制前 展的影响，一个合理的 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 匹配的 的数量，它会先 第一个与前 `bl` 匹配的，然后依次 搜集与之匹配的 （按字母 序），直到没有更多可匹配的 或当数量超 `max_expansions` 束。

不要忘 ，当用 多 入一个字符 ， 个 又会 行一遍，所以 需要快，如果第一个 果集不是用 想要的，他 会 入直到能搜出 意的 果 止。

索引 化

到目前 止，所有 的解决方案都是在 `(query time)` 的。

做并不需要特殊的映射或特殊的索引模式，只是 使用已 索引的数据。

的 活性通常会以 牺搜索性能 代 ，有 时候将 些消耗从 程中 移到 的地方是有意 的。在 web 用中，100 秒可能是一个 以忍受的巨大延 。

可以通 在索引 理数据提高搜索的 活性以及提升系 性能。此 然需要付出 有的代 加的索 引空 与 慢的索引能力，但 与 次 都需要付出代 不同，索引 的代 只用付出一次。

用 会感 我 。

Ngrams 在部分匹配的 用

之前提到：“只能在倒排索引中 到存在的 。” 尽管 `prefix` 、 `wildcard` 、 `regexp` 告 我 法并不完全正 ，但 个 的 要比在 列表中盲目挨个 效率要高得多。在搜索之前准 好供部分匹配的数据可以提高搜索的性能。

在索引 准 数据意味着要 合 的分析 ， 里部分匹配使用的工具是 *n-gram* 。可以将 *n-gram* 看成一个在 上滑 口， *n* 代表 个“ 口”的 度。如果我 要 n-gram `quick` 个 —— 它的 果取决于 *n* 的 度：

度 1 (unigram) : [`q`, `u`, `i`, `c`, `k`]

度 2 (bigram) : [`qu`, `ui`, `ic`, `ck`]

度 3 (trigram) : [`qui`, `uic`, `ick`]

度 4 (four-gram) : [`quic`, `uick`]

度 5 (five-gram) : [`quick`]

朴素的 n-gram 内部的匹配 非常有用，即在 Ngram 匹配 合 介 的那 。但 于 入即搜索 (search-as-you-type) 用 景，我 会使用一 特殊的 n-gram 称 界 *n-grams* (edge n-grams)。所 的 界 n-gram 是 它会固定 始的一 ，以 `quick` 例，它的 界 n-gram 的 果 :

- `q`
- `qu`
- `qui`
- `quic`
- `quick`

可能会注意到 与用 在搜索 入 “quick” 的字母次序是一致的， 句 ， 方式正好 足即 搜索 (instant search) !

索引 入即搜索

置索引 入即搜索的第一 是需要定 好分析 ，我 已在 配置分析器 中 ， 里会 些 再次 明。

准 索引

第一 需要配置一个自定 的 `edge_ngram` token 器，称 `autocomplete_filter`：

```
{  
  "filter": {  
    "autocomplete_filter": {  
      "type": "edge_ngram",  
      "min_gram": 1,  
      "max_gram": 20  
    }  
  }  
}
```

个配置的意思是：于 个 token 器接收的任意 ， 器会 之生成一个最小固定 1 , 最大 20 的 n-gram。

然后会在一个自定 分析器 `autocomplete` 中使用上面 个 token 器：

```
{  
  "analyzer": {  
    "autocomplete": {  
      "type": "custom",  
      "tokenizer": "standard",  
      "filter": [  
        "lowercase",  
        "autocomplete_filter" ①  
      ]  
    }  
  }  
}
```

① 自定 的 `edge-ngram` token 器。

个分析器使用 `standard` 分 器将字符串拆分 独立的 ， 并且将它 都 成小写形式，然后 个 生成一个 界 n-gram， 要感 `autocomplete_filter` 起的作用。

建索引、 例化 token 器和分析器的完整示例如下：

```

PUT /my_index
{
  "settings": {
    "number_of_shards": 1, ①
    "analysis": {
      "filter": {
        "autocomplete_filter": { ②
          "type": "edge_ngram",
          "min_gram": 1,
          "max_gram": 20
        }
      },
      "analyzer": {
        "autocomplete": {
          "type": "custom",
          "tokenizer": "standard",
          "filter": [
            "lowercase",
            "autocomplete_filter" ③
          ]
        }
      }
    }
  }
}

```

① 参考 被破坏的相 度。

② 首先自定 token 器。

③ 然后在分析器中使用它。

可以拿 `analyze` API 个新的分析器 保它行 正：

```

GET /my_index/_analyze?analyzer=autocomplete
quick brown

```

果表明分析器能正 工作，并返回以下：

- `q`
- `qu`
- `qui`
- `quic`
- `quick`
- `b`
- `br`
- `bro`
- `brow`

- brown

可以用 `update-mapping` API 将一个分析器用到具体字段：

```
PUT /my_index/_mapping/my_type
{
  "my_type": {
    "properties": {
      "name": {
        "type": "string",
        "analyzer": "autocomplete"
      }
    }
  }
}
```

在 建一些 文：

```
POST /my_index/my_type/_bulk
{ "index": { "_id": 1 } }
{ "name": "Brown foxes" }
{ "index": { "_id": 2 } }
{ "name": "Yellow furballs" }
```

字段

如果使用 `match` “brown fo”：

```
GET /my_index/my_type/_search
{
  "query": {
    "match": {
      "name": "brown fo"
    }
  }
}
```

可以看到 个文 同 都能匹配，尽管 `Yellow furballs` 个文 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，`validate-query` API 能提供一些 索：

```
GET /my_index/my_type/_validate/query?explain
{
  "query": {
    "match": {
      "name": "brown fo"
    }
  }
}
```

`explanation` 表明 会 界 n-grams 里的 个：

```
name:b name:br name:bro name:brow name:brown name:f name:fo
```

`name:f` 条件可以 足第二个文，因 `furballs` 是以 `f`、`fu`、`fur` 形式索引的。回 看 并不令人 相同的 `autocomplete` 分析器同 被 用于索引 和搜索， 在大多数情况下是正 的，只有在少数 景下才需要改 行。

我 需要保 倒排索引表中包含 界 n-grams 的 个，但是我 只想匹配用 入的完整 (`brown` 和 `fo`)，可以通 在索引 使用 `autocomplete` 分析器，并在搜索 使用 `standard` 准分析器来 想法，只要改 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" ①
      }
    }
  }
}

```

① 覆盖了 name 字段 analyzer 的设置。

方式，我可以在映射中，name 字段分指定 index_analyzer 和 search_analyzer。因 我只想改 search_analyzer，里只要更新 有的映射而不用 数据重新 建索引：

```

PUT /my_index/my_type/_mapping
{
  "my_type": {
    "properties": {
      "name": {
        "type": "string",
        "index_analyzer": "autocomplete", ①
        "search_analyzer": "standard" ②
      }
    }
  }
}

```

① 在索引，使用 autocomplete 分析器生成 索引 n-grams 的 个。

② 在搜索，使用 standard 分析器只搜索用 入的。

如果再次 求 validate-query API，当前的解：

```
name:brown name:fo
```

再次 行 就能正 返回 Brown foxes 个文。

因 大多数工作是在索引 完成的，所有的 只要 brown 和 fo 个，比使用 match_phrase_prefix 所有以 fo 始的 的方式要高效 多。

全提示 (Completion Suggester)

使用 界 n-grams 行 入即搜索 (search-as-you-type) 的 置、活且快速，但有 时候它并不 快，特 是当 立刻 得反，延 的 就会凸，很多 时候不搜索才是最快的 搜索方式。

Elasticsearch 里的 completion suggester 采用与上面完全不同的方式，需要 搜索条件生成一个所有可能完成的 列表，然后将它 置入一个 有限状机 (*finite state transducer*) 内，是个 化的 。 为了搜索建 提示，Elasticsearch 从 的 始着匹配路径一个字符一个字符地 行匹配，一旦它 于用 入的末尾，Elasticsearch 就会 所有可能 束的当前路径，然后生成一个建 列表。

本数据 存于内存中，能使前 非常快，比任何一 基于 的 都要快很多， 名字或品 牌的自 全非常 用，因 些 通常是以普通 序 的：用 “Johnny Rotten” 而不是 “Rotten Johnny”。

当 序不是那 容易被 ， 界 n-grams 比完成建 者 (Completion Suggester) 更合 。即使 不是所有猫都是一个花色，那 只猫的花色也是相当特殊的。

界 n-grams 与

界 n-gram 的方式可以用来 化的数据，比如 本章之前示例 中的 (postcode)。当然 postcode 字段需要 analyzed 而不是 not_analyzed，不 可以用 keyword 分 器来 理它，就好像他 是 not_analyzed 的一 。

TIP **keyword** 分 器是一个非操作型分 器， 个分 器不做任何事情，它接收的任何字符串都会被原 出，因此它可以用来 理 not_analyzed 的字段 ，但 也需要其他的一些分析 ，如将字母 成小写。

下面示例使用 keyword 分 器将 成 token 流， 就能使用 界 n-gram token 器：

```
{
    "analysis": {
        "filter": {
            "postcode_filter": {
                "type": "edge_ngram",
                "min_gram": 1,
                "max_gram": 8
            }
        },
        "analyzer": {
            "postcode_index": { ①
                "tokenizer": "keyword",
                "filter": [ "postcode_filter" ]
            },
            "postcode_search": { ②
                "tokenizer": "keyword"
            }
        }
    }
}
```

① `postcode_index` 分析器使用 `postcode_filter` 将成界 n-gram 形式。

② `postcode_search` 分析器可以将搜索看成 `not_analyzed` 未分析的。

Ngrams 在合的用

最后，来看看 n-gram 是如何用于搜索合的言中的。的特点是它可以将多小合成一个大的合以表它准或的意。例如：

Aussprachewörterbuch

音字典 (Pronunciation dictionary)

Militärgeschichte

争史 (Military history)

Weißkopfseeadler

(White-headed sea eagle, or bald eagle)

Weltgesundheitsorganisation

世界生 (World Health Organization)

Rindfleischetikettierungsaufgabenübertragungsgesetz

法案考代理管牛和牛肉的的 (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)的候，能在果中看到“Aussprachewörterbuch”(音字典)。同，搜索“Adler”()的候，能将“Weißkopfseeadler”()包括在果中。

理 言的一 方式可以用 合 token 器 (compound word token filter) 将 合 拆分成各自部分，但 方式的 果 量依 于 合 字典的 量。

一 方式就是将所有的 用 n-gram 行 理，然后搜索任何匹配的片段——能匹配的片段越多，文 的相 度越大。

假 某个 n-gram 是一个 上的滑 口，那 任何 度的 n-gram 都可以遍 个 。我 既希望 足 的 拆分的 具有意 ，又不至于因 太 而生成 多的唯一 。一个 度 3 的 trigram 可能是一个不 的 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" ①
          }
        }
      }
    }
  }
}
```

① text 字段用 trigrams 分析器索引它的内容， 里 n-gram 的 度是 3。

使用 analyze API trigram 分析器：

```
GET /my_index/_analyze?analyzer=trigrams  
Weißkopfseeadler
```

返回以下：

```
wei, eiß, ißk, ßko, kop, opf, pfs, fse, see, eea, ead, adl, dle, ler
```

索引前述示例中的 合 来：

```
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”（ ）的搜索 化 三个 **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"  
            }  
        }  
    ]  
}
```

似 “Gesundheit”（健康）可以与 “Welt-gesundheit-sorganisation” 匹配，同 也能与 “Militär-ges-chichte” 和 “Rindfleischetikettierungsüberwachungsaufgabenübertragungs-ges-etz” 匹配，因 它 同 都有 trigram 生成的 ges :

使用合 的 `minimum_should_match` 可以将 些奇怪的 果排除，只有当 trigram 最少匹配数 足要求 ，文 才能被 是匹配的：

```
GET /my_index/my_type/_search
{
  "query": {
    "match": {
      "text": {
        "query": "Gesundheit",
        "minimum_should_match": "80%"
      }
    }
  }
}
```

有点像全文搜索中霰 式的策略，可能会 致倒排索引内容 多，尽管如此，在索引具有很多 合 的 言，或 之 没有空格的 言（如：泰 ），它 不失 一 通用且有效的方法。

技 可以用来提升 召回率 —— 搜索 果中相 的文 数。它通常会与其他技 一起使用，例如 shingles（参 [shingles 瓦片](#) ），以提高精度和 个文 的相 度 分。

控制相 度

理 化数据（比如： 、数字、字符串、枚 ）的数据 ，只需 文 （或 系数据 里的行）是 否与 匹配。

布 的是/非匹配是全文搜索的基 ，但不止如此，我 要知道 个文 与 的相 度，在全文搜索引擎 中不 需要 到匹配的文 ， 需根据它 相 度的高低 行排序。

全文相 的公式或 相似算法 (*similarity algorithms*) 会将多个因素合并起来， 个文 生成一个相 度 分 `_score`。本章中，我 会 各 可 部分，然后 如何来控制它 。

当然，相 度不只与全文 有 ，也需要将 化的数据考 其中。可能我 正在 一个度假屋，需要一些的 特征（空 、海景、免 WiFi ），匹配的特征越多相 度越高。可能我 希望有一些其他的考 因素，如回 率、 格、受 迎度或距 ，当然也同 考 全文 的相 度。

所有的 些都可以通 Elasticsearch 大的 分基 来 。

本章会先从理 上介 Lucene 是如何 算相 度的，然后通 例子 明如何控制相 度的 算 程。

相 度 分背后的理

Lucene（或 Elasticsearch）使用 布 模型 (*Boolean model*) 匹配文 ，并用一个名 用 分函数 (*practical scoring function*) 的公式来 算相 度。 个公式借 了 /逆向文 率 (*term*

frequency/inverse document frequency) 和 *向量空 模型 (vector space model)* , 同 也加入了一些代的新特性, 如 因子 (coordination factor) , 字段 度 一化 (field length normalization) , 以及 或 句 重提升。

NOTE 不要 ! 些概念并没有像它 字面看起来那 , 尽管本小 提到了算法、公式和数 学模型, 但内容 是 人容易理解的, 与理解算法本身相比, 了解 些因素如何影 果更 重要。

布 模型

布 模型 (*Boolean Model*) 只是在 中使用 **AND** 、 **OR** 和 **NOT** (与、或和非) 的条件来 匹配的文 , 以下 :

```
full AND text AND search AND (elasticsearch OR lucene)
```

会将所有包括 **full**、**text** 和 **search** , 以及 **elasticsearch** 或 **lucene** 的文 作 果集。

个 程 且快速, 它将所有可能不匹配的文 排除在外。

/逆向文 率 (TF/IDF)

当匹配到一 文 后, 需要根据相 度排序 些文 , 不是所有的文 都包含所有 , 有些 比其他的 更 重要。一个文 的相 度 分部分取决于 个 在文 中的 重。

的 重由三个因素决定, 在 什 是相 中已 有所介 , 有 趣可以了解下面的公式, 但并不要求 住。

在文 中出 的 度是多少? 度越高, 重 越高 。 5 次提到同一 的字段比只提到 1 次的更相 。 的 算方式如下 :

```
tf(t in d) = √frequency ①
```

① t 在文 d 的 (tf) 是 在文 中出 次数的平方根。

如果不在意 在某个字段中出 的 次, 而只在意是否出 , 可以在字段映射中禁用 :

```

PUT /my_index
{
  "mappings": {
    "doc": {
      "properties": {
        "text": {
          "type": "string",
          "index_options": "docs" ①
        }
      }
    }
  }
}

```

① 将参数 `index_options` 置 `docs` 可以禁用位置，一个映射的字段不会算的出次数，于短或近似也不可用。要求精的 `not_analyzed` 字符串字段会使用置。

逆向文率

在集合所有文里出的率是多少？次越高，重越低。常用如 `and` 或 `the` 相度献很少，因它在多数文中都会出，一些不常如 `elastic` 或 `hippopotamus` 可以助我快速小到感趣的文。逆向文率的算公式如下：

$$idf(t) = 1 + \log(\text{numDocs} / (\text{docFreq} + 1)) \quad ①$$

① `t` 的逆向文率 (`idf`) 是：索引中文数量除以所有包含的文数，然后求其数。

字段度一

字段的度是多少？字段越短，字段的重越高。如果出在似 `title` 的字段，要比它出在内容 `body` 的字段中的相度更高。字段度的一公式如下：

$$\text{norm}(d) = 1 / \sqrt{\text{numTerms}} \quad ①$$

① 字段度一 (`norm`) 是字段中数平方根的倒数。

字段度的一全文搜索非常重要，多其他字段不需要有一。无文是否包括个字段，索引中个文的个 `string` 字段都大占用1个byte的空。于 `not_analyzed` 字符串字段的一是禁用的，而于 `analyzed` 字段也可以通过修改字段映射禁用一：

```

PUT /my_index
{
  "mappings": {
    "doc": {
      "properties": {
        "text": {
          "type": "string",
          "norms": { "enabled": false } ①
        }
      }
    }
  }
}

```

① 个字段不会将字段 度 一 考 在内， 字段和短字段会以相同 度 算 分。

于有些 用 景如日志， 一 不是很有用，要 心的只是字段是否包含特殊的 或者特定的 器 唯一 符。字段的 度 果没有影 ，禁用 一 可以 省大量内存空 。

合使用

以下三个因素—— (term frequency)、逆向文 率 (inverse document frequency) 和字段 度 一 (field-length norm) ——是在索引 算并存 的。最后将它 合在一起 算 个 在特定文 中的 重。

TIP 前面公式中提到的 文 上是指文 里的某个字段，
个字段都有它自己的倒排索引，因此字段的 TF/IDF 就是文 的 TF/IDF 。

当用 `explain` 看一个 的 `term` (参 [explain](#))，可以 与 算相 度 分的因子就是前面章 介 的 些：

```

PUT /my_index/doc/1
{ "text" : "quick brown fox" }

GET /my_index/doc/_search?explain
{
  "query": {
    "term": {
      "text": "fox"
    }
  }
}

```

以上 求(化)的 `explanation` 解 如下：

```

weight(text:fox in 0) [PerFieldSimilarity]: 0.15342641 ①
result of:
  fieldWeight in 0                      0.15342641
  product of:
    tf(freq=1.0), with freq of 1:        1.0 ②
    idf(docFreq=1, maxDocs=1):           0.30685282 ③
    fieldNorm(doc=0):                   0.5 ④

```

① fox 在文 的内部 Lucene doc ID 0，字段是 text 里的最 分。

② fox 在 文 text 字段中只出 了一次。

③ fox 在所有文 text 字段索引的逆向文 率。

④ 字段的字段 度 一 。

当然， 通常不止一个 ， 所以需要一 合并多 重的方式——向量空 模型（vector space model）。

向量空 模型

向量空 模型（vector space model） 提供一 比 多 的方式， 个 分代表文 与 的匹配程度， 了做到 点， 个模型将文 和 都以 向量（vectors） 的形式表示：

向量 上就是包含多个数的一 数 ， 例如：

[1,2,5,22,3,8]

在向量空 模型里， 向量空 模型里的 个数字都代表一个 的 重 ， 与 /逆向文 率（term frequency/inverse document frequency） 算方式 似。

TIP 尽管 TF/IDF 是向量空 模型 算 重的 方式， 但不是唯一方式。Elasticsearch 有其他模型如 Okapi-BM25 。TF/IDF 是 的因 它是个 的 又高效的算法， 可以提供高 量的搜索 果。

想如果 “happy hippopotamus” ， 常 happy 的 重 低， 不常 hippopotamus 重 高， 假 happy 的 重是 2 ， hippopotamus 的 重是 5 ， 可以将 个二 向量—— [2,5] ——在坐 系下作条直 ， 的起点是 (0,0) 点是 (2,5) ， 如 表示 “happy hippopotamus” 的二 向量 。

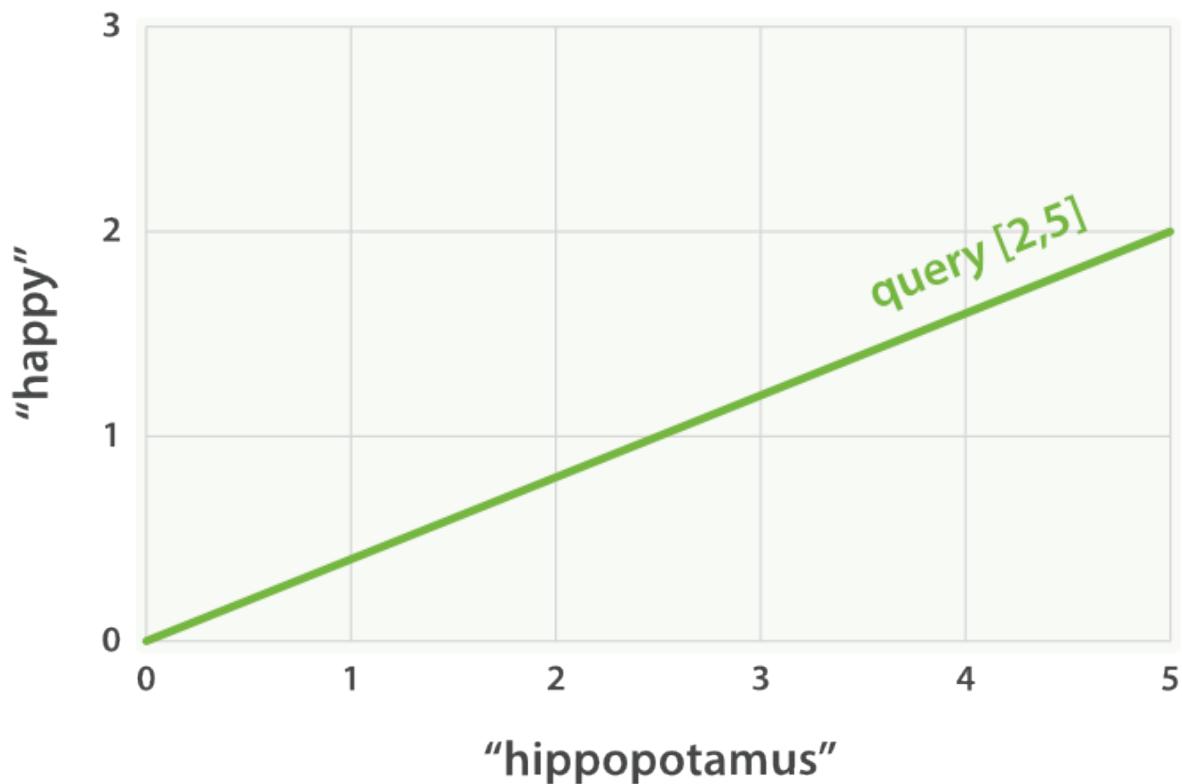


Figure 27. 表示 “*happy hippopotamus*” 的二向量

在， 想我 有三个文：

1. I am *happy* in summer .
2. After Christmas I'm a *hippopotamus* .
3. The *happy hippopotamus* helped Harry .

可以 个文 都 建包括 个 —— *happy* 和 *hippopotamus* —— 重的向量，然后将 些向量置入同一个坐 系中，如 “*happy hippopotamus*” 及文 向量：

- 文 1 : (*happy*,_) —— [2,0]
- 文 2 : (_ ,*hippopotamus*) —— [0,5]
- 文 3 : (*happy*,*hippopotamus*) —— [2,5]

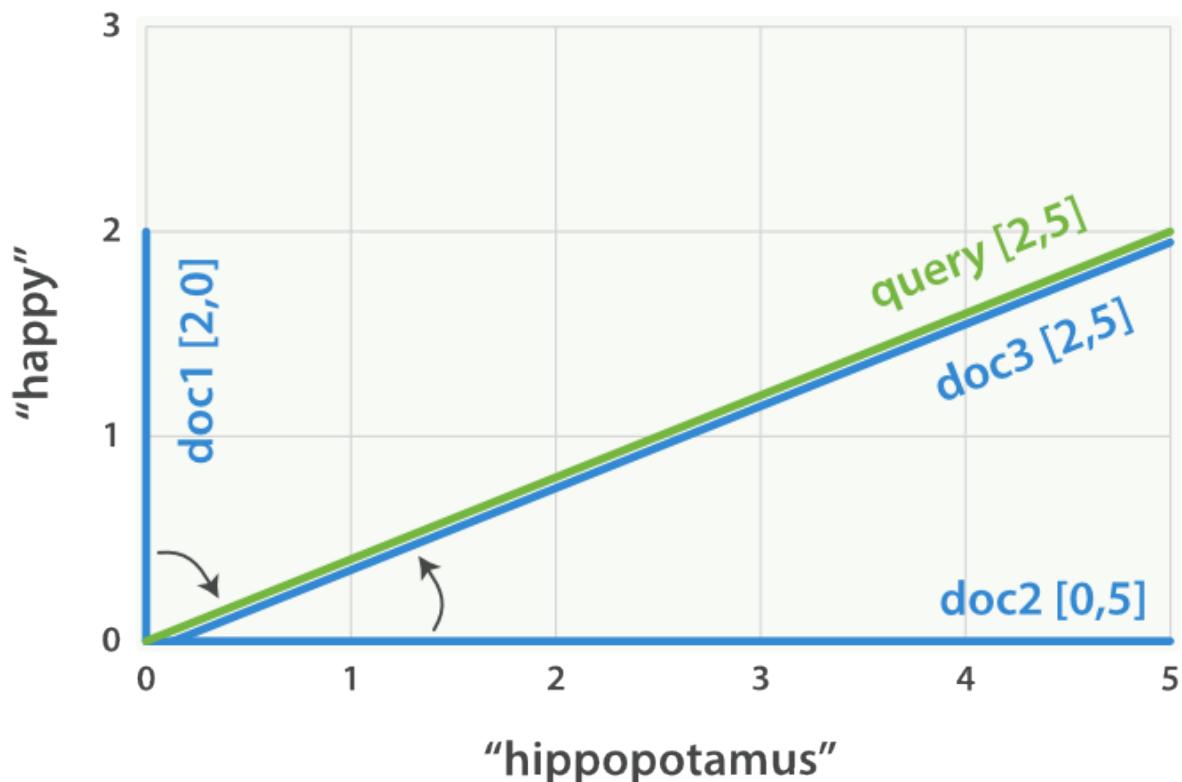


Figure 28. “happy hippopotamus” 及文 向量

向量之 是可以比 的，只要 量 向量和文 向量之 的角度就可以得到 个文 的相 度，文 1 与 之 的角度最大，所以相 度低；文 2 与 的角度 小，所以更相 ；文 3 与 的角度正好吻合，完全匹配。

TIP 在 中，只有二 向量（ 个 的 ）可以在平面上表示，幸 的是， 性代数 —— 作 数学中 理向量的一个分支—— 我 提供了 算 个多 向量 角度工具， 意味着可 以使用如上同 的方式来解 多个 的 。

于比 个向量的更多信息可以参考 [余弦近似度 \(cosine similarity\)](#) 。

在已 完 分 算的基本理 ，我 可以 了解 Lucene 是如何 分 算的。

Lucene 的 用 分函数

于多 ， Lucene 使用 布 模型 (Boolean model) 、 TF/IDF 以及 向量空 模型 (vector space model) ，然后将它 合到 个高效的包里以收集匹配文 并 行 分 算。

一个

```
GET /my_index/doc/_search
{
  "query": {
    "match": {
      "text": "quick fox"
    }
  }
}
```

会在内部被重写：

```
GET /my_index/doc/_search
{
  "query": {
    "bool": {
      "should": [
        {"term": { "text": "quick" }},
        {"term": { "text": "fox" }}
      ]
    }
  }
}
```

bool 布了布模型，在个例子中，它会将包括 **quick** 和 **fox** 或者兼有的文作果。

只要一个文与匹配，Lucene就会算分，然后合并个匹配的分果。里使用的分算公式叫做用分函数(*practical scoring function*)。看似很高大上，但是被到—多数的件都已介，下一会它引入的一些新元素。

```
score(q,d) = ①
  queryNorm(q) ②
  · coord(q,d) ③
  ·  $\sum$  ( ④
    tf(t in d) ⑤
    · idf(t)2 ⑥
    · t.getBoost() ⑦
    · norm(t,d) ⑧
  ) (t in q) ④
```

① **score(q,d)** 是文 **d** 与 **q** 的相度分。

② **queryNorm(q)** 是一化因子(新)。

③ **coord(q,d)** 是因子(新)。

④ **q** 中个 **t** 于文 **d** 的重和。

⑤ **tf(t in d)** 是 **t** 在文 **d** 中的。

⑥ **idf(t)** 是 **t** 的逆向文率。

⑦ `t.getBoost()` 是 中使用的 `boost` (新)。

⑧ `norm(t,d)` 是 字段 度 一 , 与 索引 字段 `boost` (如果存在) 的和 (新)。

上 已介 `score`、`tf` 和 `idf`。 在来介 `queryNorm`、`coord`、`t.getBoost` 和 `norm`。

我 会在本章后面 探 的 重提升 的 , 但是首先需要了解 一化、 和索引 字段 面的 重提升等概念。

一因子

一因子 (`queryNorm`) 将 一化 , 就能将 个不同的 果相比 。

TIP 尽管 一 的目的是 了使 果之 能 相互比 , 但是它并不十分有效, 因 相 度 分 `_score` 的目的是 了将当前 的 果 行排序, 比 不同 果的相 度 分没有太大意 。

个因子是在 程的最前面 算的, 具体的 算依 于具体 , 一个典型的 如下 :

```
queryNorm = 1 / √sumOfSquaredWeights ①
```

① `sumOfSquaredWeights` 是 里 个 的 IDF 的平方和。

TIP 相同 一化因子会被 用到 个文 , 不能被更改, 而言之, 可以被忽略。

因子 (`coord`) 可以 那些 包含度高的文 提供 励, 文 里出 的 越多, 它越有机会成 好的匹配 果。

想 `quick brown fox` , 个 的 重都是 1.5 。如果没有 因子, 最 分会是文 里所有 重的 和。例如 :

- 文 里有 `fox` → 分 : 1.5
- 文 里有 `quick fox` → 分 : 3.0
- 文 里有 `quick brown fox` → 分 : 4.5

因子将 分与文 里匹配 的数量相乘, 然后除以 里所有 的数量, 如果使用 因子, 分会 成 :

- 文 里有 `fox` → 分 : $1.5 * 1 / 3 = 0.5$
- 文 里有 `quick fox` → 分 : $3.0 * 2 / 3 = 2.0$
- 文 里有 `quick brown fox` → 分 : $4.5 * 3 / 3 = 4.5$

因子能使包含所有三个 的文 比只包含 个 的文 分要高出很多。

回想将 `quick brown fox` 重写成 `bool` 的形式 :

```

GET /_search
{
  "query": {
    "bool": {
      "should": [
        { "term": { "text": "quick" } },
        { "term": { "text": "brown" } },
        { "term": { "text": "fox" } }
      ]
    }
  }
}

```

`bool` 会所有 `should` 句使用功能，不也可以将其禁用。什要做？通常的回答是——无。通常是件好事，当使用 `bool` 将多个高如 `match` 包的候，功能是有意的，匹配的句越多，求与返回文的重度就越高。

但在某些高用中，将功能可能更好。想正在同 `jump`、`leap` 和 `hop`，并不心会出多少个同，因它都表示相同的意思，上，只有其中一个同会出，是不使用因子的一个好例子：

```

GET /_search
{
  "query": {
    "bool": {
      "disable_coord": true,
      "should": [
        { "term": { "text": "jump" } },
        { "term": { "text": "hop" } },
        { "term": { "text": "leap" } }
      ]
    }
  }
}

```

当使用同的候（参照：[同](#)），Lucene 内部是的：重写的会禁用同的功能。大多数禁用操作的用景是自理的，无此担心。

索引字段重提升

我会的重提升，字段重提升就是某个字段比其他字段更重要。当然在索引也能做到如此。上，重的提升会被用到字段的个，而不是字段本身。

将提升存 在索引中无更多空，个字段索引的提升与字段度一（参 [字段度一](#)）一起作一个字存于索引，`norm(t,d)` 是前面公式的返回。

WARNING

我不建议在建立索引时提升字段，有以下原因：

- 将提升与字段度合在一起，一个字中存会失字段度一的精度，会致 Elasticsearch 不知如何区分包含三个的字段和包含五个的字段。
- 要想改索引的提升，就必重新所有文建立索引，与此不同的是，的提升可以随着次的不同而更改。
- 如果一个索引重提升的字段有多个，提升会按照个来自乘，会致字段的重急上升。

予重是更、清楚、活的。

了解了一化、同和索引重提升些方式后，可以一了解相度算最有用的工具：的重提升。

重提升

在句先(Prioritizing Clauses)中，我解如何在搜索使用boost参数一个句比其他句更重要。例如：

```
GET /_search
{
  "query": {
    "bool": {
      "should": [
        {
          "match": {
            "title": {
              "query": "quick brown fox",
              "boost": 2 ①
            }
          }
        },
        {
          "match": { ②
            "content": "quick brown fox"
          }
        }
      ]
    }
  }
}
```

① title句的重要性是 content 的 2 倍，因它的重提升 2。

② 没有置 boost 的句的 1。

的重提升是可以用来影相度的主要工具，任意型的都能接受 boost 参数。将 boost 置 2，并不代表最的分 score 是原的倍；的重会一化和一些其他内部化

程。尽管如此，它想要表明一个提升 2 的句子的重要性是提升 1 句的 倍。

在 用中，无法通 的公式得出某个特定 句的 正 '' 重提升 ，只能通 不断 得。需要 住的是 boost 只是影 相 度 分的其中一个因子；它 需要与其他因子相 争。在前例中， title 字段相 content 字段可能已 有一个 省的" 重提升 ，因 在 字段 度 一 中，往往比相 内容要短，所以不要想当然的去盲目提升一些字段的 重。 重， 果，如此反 。

提升索引 重

当在多个索引中搜索 ，可以使用参数 `indices_boost` 来提升整个索引的 重，在下面例子中，当要 最近索引的文 分配更高 重 ，可以 做：

```
GET /docs_2014_*/_search ①
{
  "indices_boost": { ②
    "docs_2014_10": 3,
    "docs_2014_09": 2
  },
  "query": {
    "match": {
      "text": "quick brown fox"
    }
  }
}
```

① 个多索引 涵 了所有以字符串 `docs_2014_` 始的索引。

② 其中，索引 `docs_2014_10` 中的所有文件的 重是 3 ，索引 `docs_2014_09` 中是 2 ，其他所有匹配的索引 重 1 。

`t.getBoost()`

些提升 在 Lucene 的 用 分函数 中可以通 `t.getBoost()` 得。 重提升不会被 用于它在 表 式中出 的，而是会被合并下 至 个 中。`t.getBoost()` 始 返回当前 的 重或当前分析 上 的 重。

TIP

上，要想解 `explain` 的 出是相当 的，在 `explanation` 里面完全看不到 `boost`，也完全无法 上面提到的 `t.getBoost()` 方法， 重 融合在 `queryNorm` 中并 用到 个 。尽管 ， `queryNorm` 于 个 都是相同的， 是会 一个 重提升 的 的 `queryNorm` 要高于一个没有提升 的。

使用 修改相 度

Elasticsearch 的 表 式相当 活，可以通 整 中 句的所 次，从而或多或少改 其重要性，比如， 想下面 个 ：

```
quick OR brown OR red OR fox
```

可以将所有 都放在 **bool** 的同一 中：

```
GET /_search
{
  "query": {
    "bool": {
      "should": [
        { "term": { "text": "quick" } },
        { "term": { "text": "brown" } },
        { "term": { "text": "red" } },
        { "term": { "text": "fox" } }
      ]
    }
  }
}
```

个 可能最 包含 **quick**、**red** 和 **brown** 的文 分与包含 **quick**、**red**、**fox** 文 的 分相同，里 **Red** 和 **brown** 是同 ，可能只需要保留其中一个，而我 真正要表 的意思是想做以下：

```
quick OR (brown OR red) OR fox
```

根据 准的布 ， 与原始的 是完全一 的，但是我 已 在 合 (Combining Queries) 中看到， **bool** 不 心文 匹配的 程度，只 心是否能匹配。

上述 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" } }
            ]
          }
        }
      ]
    }
  }
}
```

在， red 和 brown 于相互 纠争的 次， quick 、 fox 以及 red OR brown 是 于 且相互 纠争的。

我 已 知道 如何使用 match 、 multi_match 、 term 、 bool 和 dis_max 修改相 度 分。本章后面的内容会介 绍另外三个与相 度 分有 关的 方 法： boosting 、 constant_score 和 function_score 。

Not Quite Not

在互 联网上搜索 “Apple”，返回的 果很可能是一个公司、水果和各 食 。我 可以在 bool 中用 must_not 句来排除像 pie 、 tart 、 crumble 和 tree 等 的 ，从而将 果的 相 度 小至只返回与 “Apple” (果) 公司相 符的 果：

```

GET /_search
{
  "query": {
    "bool": {
      "must": {
        "match": {
          "text": "apple"
        }
      },
      "must_not": {
        "match": {
          "text": "pie tart fruit crumble tree"
        }
      }
    }
  }
}

```

但又敢保在排除 `tree` 或 `crumble` 后，不会失一个与果公司特相的文？有，`must_not` 条件会于格。

重提升

`boosting` 恰恰能解决个。它然允我将于水果或甜点的果包括到果中，但是使它降——即降低它原来可能有的排名：

```

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`。只有那些匹配 `positive` 的文列出来，于那些同匹配 `negative` 的文将通文的原始 `_score` 与 `negative_boost` 相乘的方式降后的果。

了 到效果， `negative_boost` 的 必 小于 `1.0` 。在 个示例中，所有包含 向 的文 分 `_score` 都会 半。

忽略 TF/IDF

有 候我 根本不 心 TF/IDF ， 只想知道一个 是否在某个字段中出 。可能搜索一个度假屋并希望它能尽可能有以下 施：

- WiFi
- Garden (花)
- Pool (游泳池)

个度假屋的文 如下：

```
{ "description": "A delightful four-bedroomed house with ... " }
```

可以用 的 `match` 行匹配：

```
GET /_search
{
  "query": {
    "match": {
      "description": "wifi garden pool"
    }
  }
}
```

但 并不是真正的 全文搜索 ，此 情况下，TF/IDF 并无用 。我 既不 心 `wifi` 是否 一个普通 ，也不 心它在文 中出 是否 繁， 心的只是它是否曾出 。 上，我 希望根据房屋不同 施的数量 其排名—— 施越多越好。如果 施出 ， 1 分，不出 0 分。

constant_score

在 `constant_score` 中，它可以包含 或 ， 任意一个匹配的文 指定 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 不是所有的 施都同等重要—— 某些用 来 有些 施更有 。如果最重要的 施是游泳池，那我可以 更重要的 施 加 重：

```

GET /_search
{
  "query": {
    "bool": {
      "should": [
        { "constant_score": {
          "query": { "match": { "description": "wifi" } }
        }},
        { "constant_score": {
          "query": { "match": { "description": "garden" } }
        }},
        { "constant_score": {
          "boost": 2 ①
          "query": { "match": { "description": "pool" } }
        }}
      ]
    }
  }
}

```

① pool 句的 重提升 2，而其他的 句 1。

NOTE 最 的 分并不是所有匹配 句的 求和，因子 (coordination factor) 和 一化因子 (query normalization factor) 然会被考 在内。

我 可以 **features** 字段加上 **not_analyzed** 型来提升度假屋文 的匹配能力：

```
{ "features": [ "wifi", "pool", "garden" ] }
```

情况下，一个 `not_analyzed` 字段会禁用 `字段度一 (field-length norms)` 的功能，并将 `index_options` `docs`，禁用，但是存在：个的倒排文率然会被考。

可以采用与之前相同的方法 `constant_score` 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" } }
          }}
      ]
    }
  }
}
```

上，个施都看成一个器，于度假屋来要具有某个施要没有——器因其性天然合。而且，如果使用器，我可以利用存。

里的 是：器无法算分。就需要求一方式将器和的差抹平。
`function_score` 不正好可以扮演个角色，而且有更大的功能。

function_score

`function_score`是用来控制分程的武器，它允一个与主匹配的文用一个函数，以到改甚至完全替原始分`_score`的目的。

上，也能用器果的子集用不同的函数，一箭双：既能高效分，又能利用器存。

Elasticsearch 定了一些函数：

`weight`

个文用一个而不被化的重提升：当 `weight 2`，最果 `2 * _score`。

`field_value_factor`

使用个来修改`_score`，如将 `popularity` 或 `votes`（受迎或）作考因素。

random_score

个用 都使用一个不同的随机 分 果排序，但 某一具体用 来 ，看到的 序始 是一致的。

衰 函数——linear、exp、gauss

将浮 合到 分 _score 中，例如 合 publish_date 得最近 布的文 ， 合 geo_location 得更接近某个具体 度 (lat/lon) 地点的文 ， 合 price 得更接近某个特定 格的文 。

script_score

如果需求超出以上 ， 用自定 脚本可以完全控制 分 算， 所需 。

如果没有 function_score ， 就不能将全文 与最新 生 因子 合在一起 分，而不得不根据 分 _score 或 date 行排序； 会相互影 抵消 排序各自的效果。 个 可以使 个效果融合：可以 然根据全文相 度 行排序，但也会同 考 最新 布文 、流行文 、或接近用 希望 格的 品。正如所 想的， 要考 所有 些因素会非常 ， 我 先从 的例子 始，然后 着梯子慢慢向上爬， 加 度。

按受 迎度提升 重

想有个 站供用 布博客并且可以 他 自己喜 的博客点 ， 我 希望将更受 迎的博客放在搜索 果列表中相 上的位置，同 全文搜索的 分 然作 相 度的主要排序依据，可以 的通 存 个博客的点 数来 它：

```
PUT /blogposts/post/1
{
  "title": "About popularity",
  "content": "In this post we will talk about...",
  "votes": 6
}
```

在搜索 ， 可以将 function_score 与 field_value_factor 合使用，即将点 数与全文相 度 分 合：

```
GET /blogposts/post/_search
{
  "query": {
    "function_score": { ①
      "query": { ②
        "multi_match": {
          "query": "popularity",
          "fields": [ "title", "content" ]
        }
      },
      "field_value_factor": { ③
        "field": "votes" ④
      }
    }
  }
}
```

① `function_score` 将主查询和函数包括在内。

② 主查询先行。

③ `field_value_factor` 函数会被用到一个与主 `query` 匹配的文档。

④ 每个文档的 `votes` 字段都必须有可供 `function_score` 算。如果没有文档的 `votes` 字段，那就必须使用 `missing` 属性提供的值来行分算。

在前面示例中，每个文档的最绢单分 `_score` 都做了如下修改：

```
new_score = old_score * number_of_votes
```

然而，并不会出人意料的好果，全文单分 `_score` 通常位于 0 到 10 之间，如下所示。受热度的系数基于 `_score` 的原始 2.0 中，有 10 个的博客会掩掉全文分，而 0 个的博客的分会被置 0。

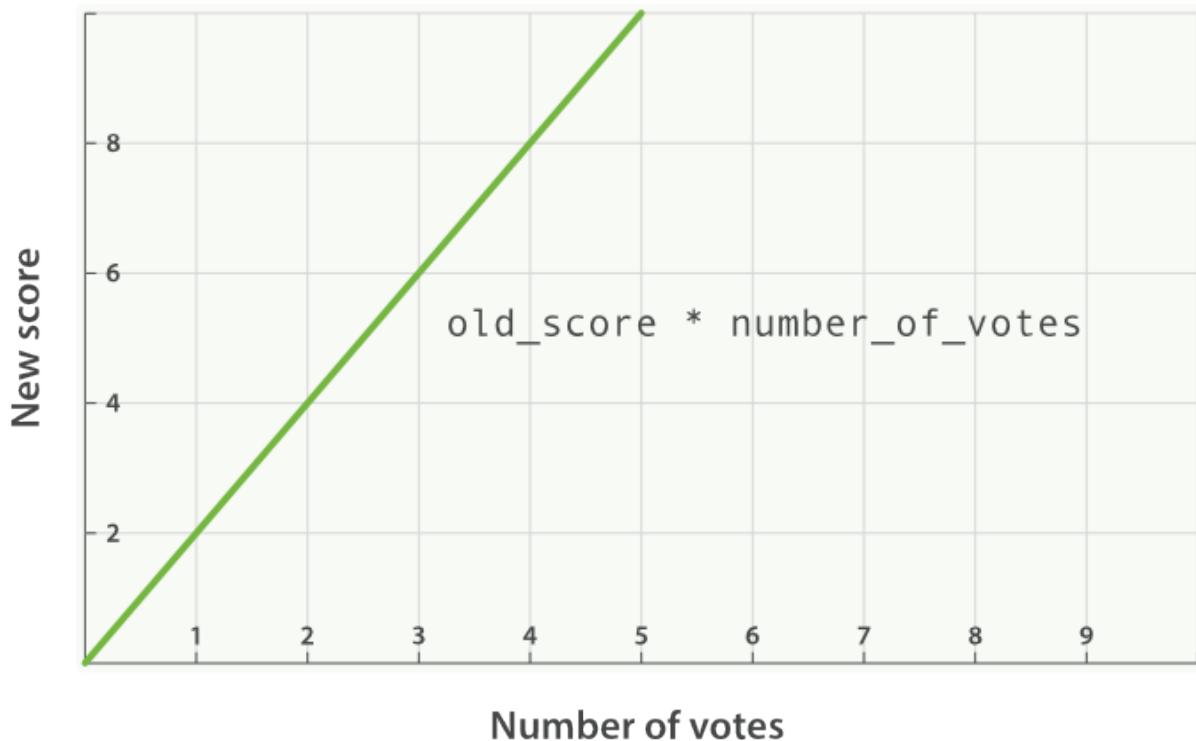


Figure 29. 受迎度的性系基于 `_score` 的原始 2.0

modifier

融入受迎度更好方式是用 `modifier` 平滑 `votes` 的。句，我希望最始的一些更重要，但是其重要性会随着数字的加而降低。0个与1个的区比10个与11个的区大很多。

于上述情况，典型的 `modifier` 用是使用 `log1p` 参数，公式如下：

```
new_score = old_score * log(1 + number_of_votes)
```

`log` 数函数使 `votes` 字段的分曲更平滑，如受迎度的数系基于 `_score` 的原始 2.0：

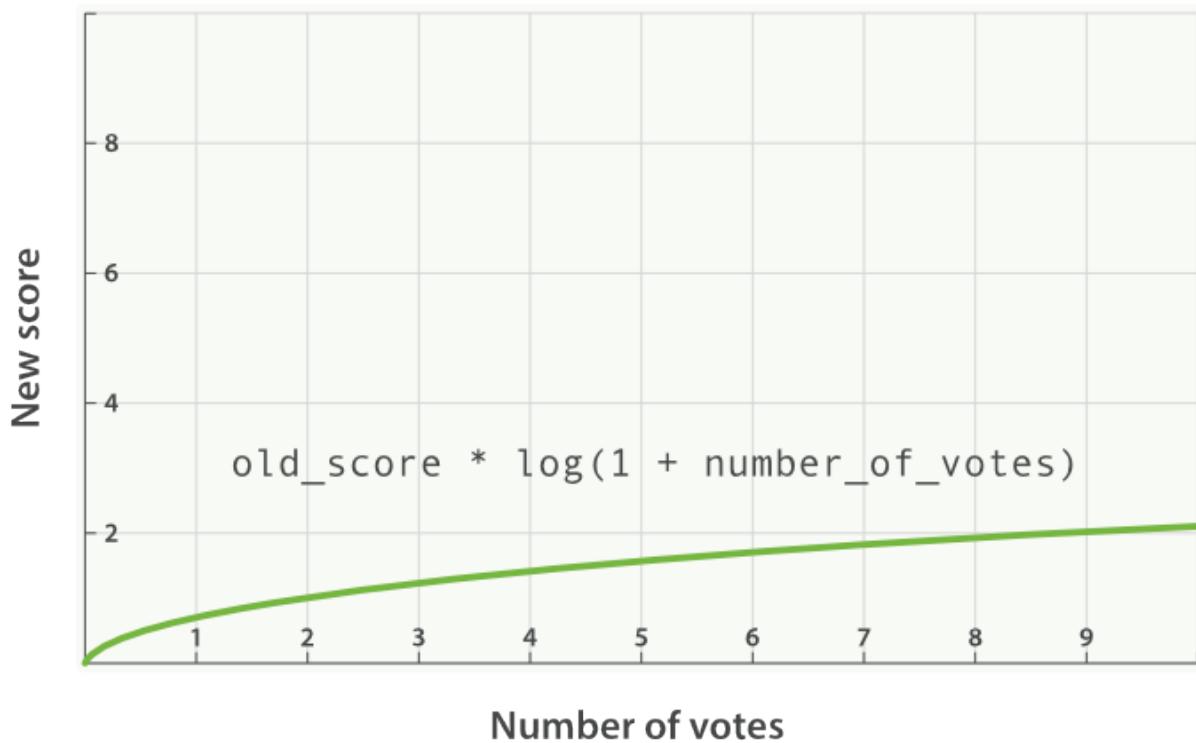


Figure 30. 受迎度的数系基于 `_score` 的原始 2.0

`modifier` 参数的求如下：

```
GET /blogposts/post/_search
{
  "query": {
    "function_score": {
      "query": {
        "multi_match": {
          "query": "popularity",
          "fields": [ "title", "content" ]
        }
      },
      "field_value_factor": {
        "field": "votes",
        "modifier": "log1p" ①
      }
    }
  }
}
```

① `modifier log1p`。

修改 `modifier` 的可以：`none`（状）、`log`、`log1p`、`log2p`、`ln`、`ln1p`、`ln2p`、`square`、`sqrt` 以及 `reciprocal`。想要了解更多信息 参照：`field_value_factor` 文。

factor

可以通 将 votes 字段与 factor 的 来 受 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 ①
      }
    }
  }
}
```

① 双倍效果。

添加了 factor 会使公式 成 :

```
new_score = old_score * log(1 + factor * number_of_votes)
```

factor 大于 1 会提升效果, factor 小于 1 会降低效果, 如 受 迎度的 数 系基于多个不同因子。

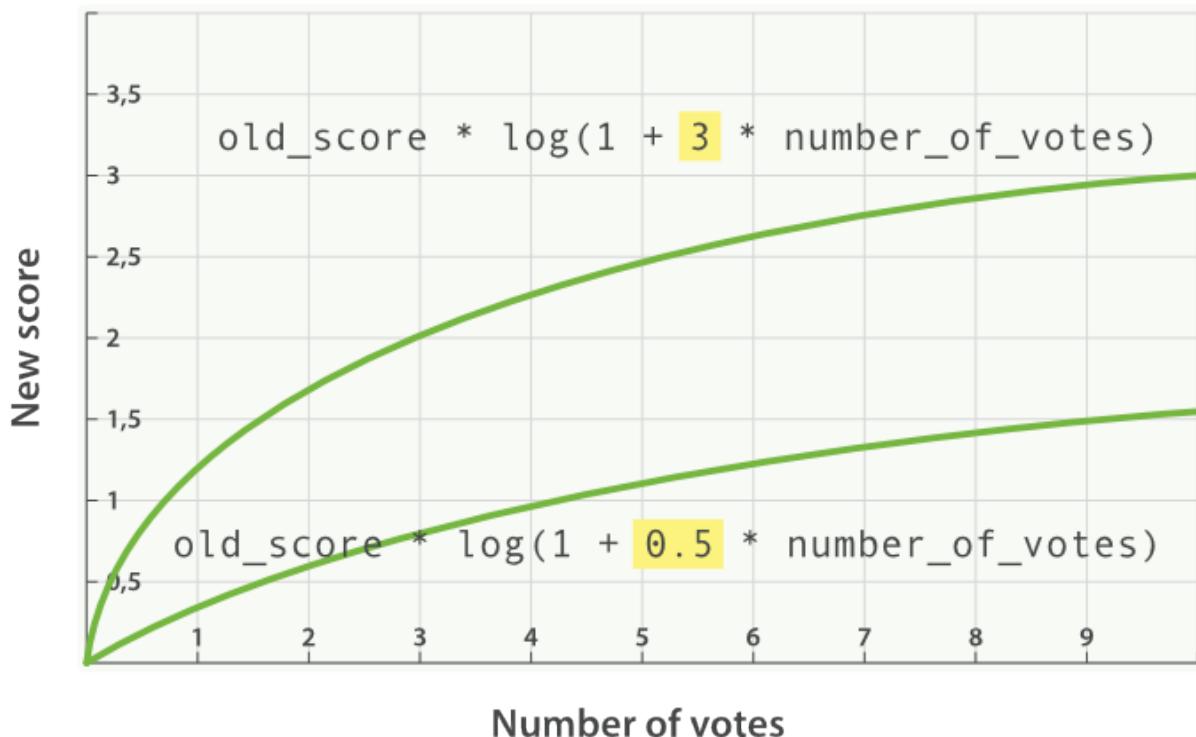


Figure 31. 受 迎度的 数 系基于多个不同因子

boost_mode

或 将全文 分与 `field_value_factor` 函数 乘 的效果 然可能太大，我 可以通 参数 `boost_mode` 来控制函数与 分 `_score` 合并后的 果，参数接受的：

`multiply`

分 `_score` 与函数 的 ()

`sum`

分 `_score` 与函数 的和

`min`

分 `_score` 与函数 的 小

`max`

分 `_score` 与函数 的 大

`replace`

函数 替代 分 `_score`

与使用乘 的方式相比，使用 分 `_score` 与函数 求和的方式可以弱化最 效果，特 是使用一个 小 `factor` 因子：

```

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" ①
    }
  }
}

```

① 分 `_score` 与函数 的 。

之前 求的公式 在 成下面 (参 使用 `sum` 合受 迎程度) :

```
new_score = old_score + log(1 + 0.1 * number_of_votes)
```

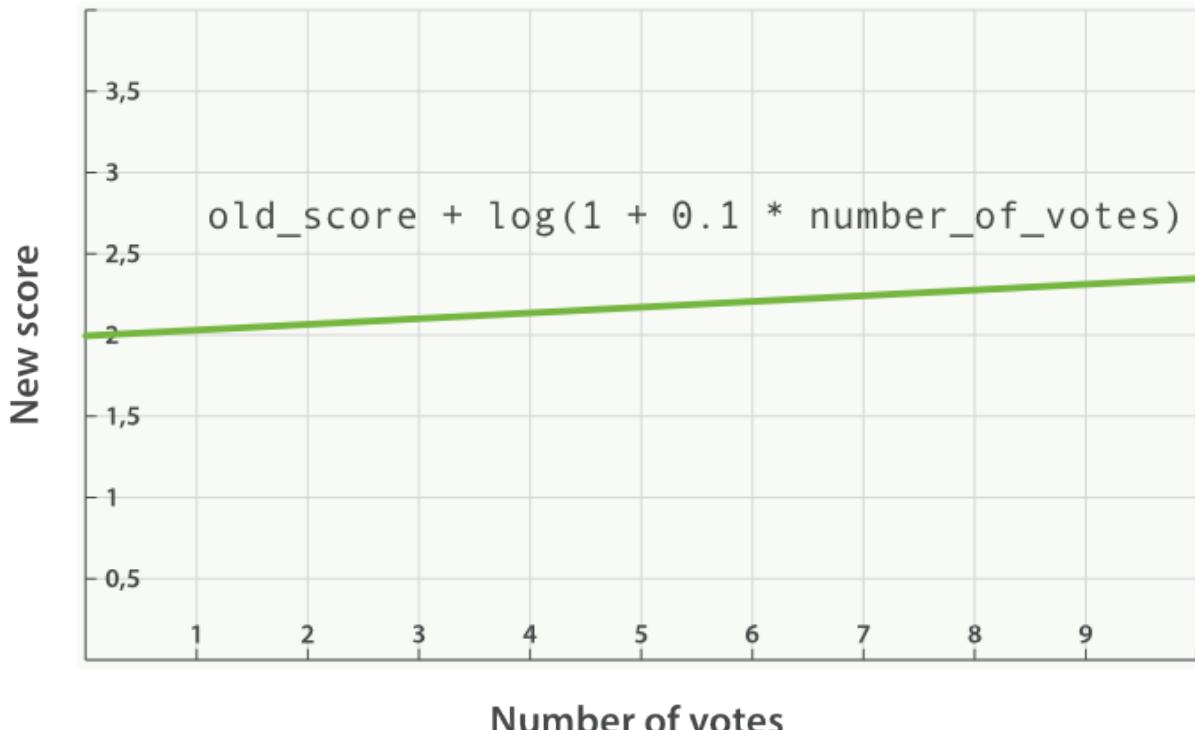


Figure 32. 使用 `sum` 合受 迎程度

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 ①
    }
  }
}
```

① 无 `field_value_factor` 函数的 果如何，最 果都不会大于 1.5。

NOTE `max_boost` 只 函数的 果 行限制，不会 最 分 `_score` 生直接影 。

集提升 重

回到 忽略 TF/IDF 里 理 的 ，我 希望根据 个度假屋的特性数量来 分，当 我 希望能用存的 器来影 分，在 `function_score` 正好可以完成 件事情。

到目前 止，我 展 的都是 所有文 用 个函数的使用方式，在会用 器将 果 分 多个子集（ 个特性一个 器），并 个子集使用不同的函数。

在下面例子中，我 会使用 `weight` 函数，它与 `boost` 参数 似可以用于任何 。有一点区 是 `weight` 没有被 Luence 一化成 以理解的浮点数，而是直接被 用。

的 需要做相 更以整合多个函数：

```

GET /_search
{
  "query": {
    "function_score": {
      "filter": { ①
        "term": { "city": "Barcelona" }
      },
      "functions": [ ②
        {
          "filter": { "term": { "features": "wifi" }}, ③
          "weight": 1
        },
        {
          "filter": { "term": { "features": "garden" }}, ③
          "weight": 1
        },
        {
          "filter": { "term": { "features": "pool" }}, ③
          "weight": 2 ④
        }
      ],
      "score_mode": "sum", ⑤
    }
  }
}

```

① `function_score` 有个 `filter` 器而不是 `query`。

② `functions` 字存 着一个将被 用的函数列表。

③ 函数会被 用于和 `filter` 器（可 的）匹配的文 。

④ `pool` 比其他特性更重要，所以它有更高 `weight`。

⑤ `score_mode` 指定各个函数的 行 合 算的方式。

个新特性需要注意的地方会在以下小 介 。

VS.

首先要注意的是 `filter` 器代替了 `query`，在本例中，我 无 使用全文搜索，只想 到 `city` 字段中包含 `Barcelona` 的所有文 ， 用 比用 表 更清晰。 器返回的所有文 的 分 `_score` 的 1。 `function_score` 接受 `query` 或 `filter`，如果没有特 指定， 使用 `match_all` 。

函数 `functions`

`functions` 字保持着一个将要被使用的函数列表。可以 列表里的 个函数都指定一个 `filter` 器，在 情况下，函数只会被 用到那些与 器匹配的文 ，例子中，我 与 器匹配的文 指定 重 `weight` 1 (与 `pool` 匹配的文 指定 重 2)。

分模式 score_mode

个函数返回一个 果，所以需要一 将多个 果 到 个 的方式，然后才能将其与原始 分 `_score` 合并。 分模式 `score_mode` 参数正好扮演 的角色，它接受以下：

`multiply`

函数 果求 ()。

`sum`

函数 果求和。

`avg`

函数 果的平均 。

`max`

函数 果的最大 。

`min`

函数 果的最小 。

`first`

使用首个函数（可以有 器，也可能没有）的 果作 最 果

在本例中，我 将 个 器匹配 果的 重 `weight` 求和，并将其作 最 分 果，所以会使用 `sum` 分模式。

不与任何 器匹配的文 会保有其原始 分，`_score` 的 1。

随机 分

可能会想知道 一致随机 分 (*consistently random scoring*) 是什，又 什 会使用它。之前的例子是个很好的 用 景，前例中所有的 果都会返回 1、2、3、4 或 5 的最 分 `_score`，可能只有少数房子的 分是 5 分，而有大量房子的 分是 2 或 3。

作 站的所有者， 会希望 广告有更高的展 率。在当前 下，有相同 分 `_score` 的文 会 次都以相同次序出 ， 了提高展 率，在此引入一些随机性可能会是个好主意， 能保 有相同 分的 文 都能有均等相似的展 机率。

我 想 个用 看到不同的随机次序，但也同 希望如果是同一用 翻 果的相 次序能始 保持一致。 行 被称 一致随机 (*consistently random*) 。

`random_score` 函数会 出一个 0 到 1 之 的数，当 子 `seed` 相同 ，生成的随机 果是一致的，例如，将用 的会 ID 作 `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": { ①
            "seed": "the user's session id" ②
          }
        }
      ],
      "score_mode": "sum"
    }
  }
}

```

① `random_score` 句没有任何 器 `filter`，所以会被 用到所有文 。

② 将用 的会 ID 作 子 `seed`， 用 的随机始 保持一致，相同的 子 `seed` 会 生相同的随机 果。

当然，如果 加了与 匹配的新文 ，无 是否使用一致随机，其 果 序都会 生 化。

越近越好

很多 量都可以影 用 于度假屋的 ，也 用 希望 市中心近点，但如果 格足 便宜，也有可能 一个更 的住 ，也有可能反 来是正 的： 意 最好的位置付更多的 。

如果我 添加 器排除所有市中心方 1 千米以外的度假屋，或排除所有 格超 £100 英 的，我 可能会将用 意考 妥 的那些 排除在外。

`function_score` 会提供一 衰 函数 (`decay functions`) ， 我 有能力在 个滑 准，如地点和 格，之 衡。

有三 衰 函数—— `linear` 、 `exp` 和 `gauss` （ 性、指数和高斯函数），它 可以操作数 、

以及度地理坐点的字段。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，落在原点 origin 上的文分 _score 分 1.0。

scale

衰率，即一个文从原点 origin 下落，分 _score 改的速度。（例如，£10 欧元或 100 米）。

decay

从原点 origin 衰到 scale 所得的分 _score , 0.5。

offset

以原点 origin 中心点，其置一个非零的偏移量 offset 覆一个，而不只是个原点。在 $-offset \leq origin \leq +offset$ 内的所有分 _score 都是 1.0。

三个函数的唯一区别就是它衰曲的形状，用来看会更直（参见衰函数曲）。

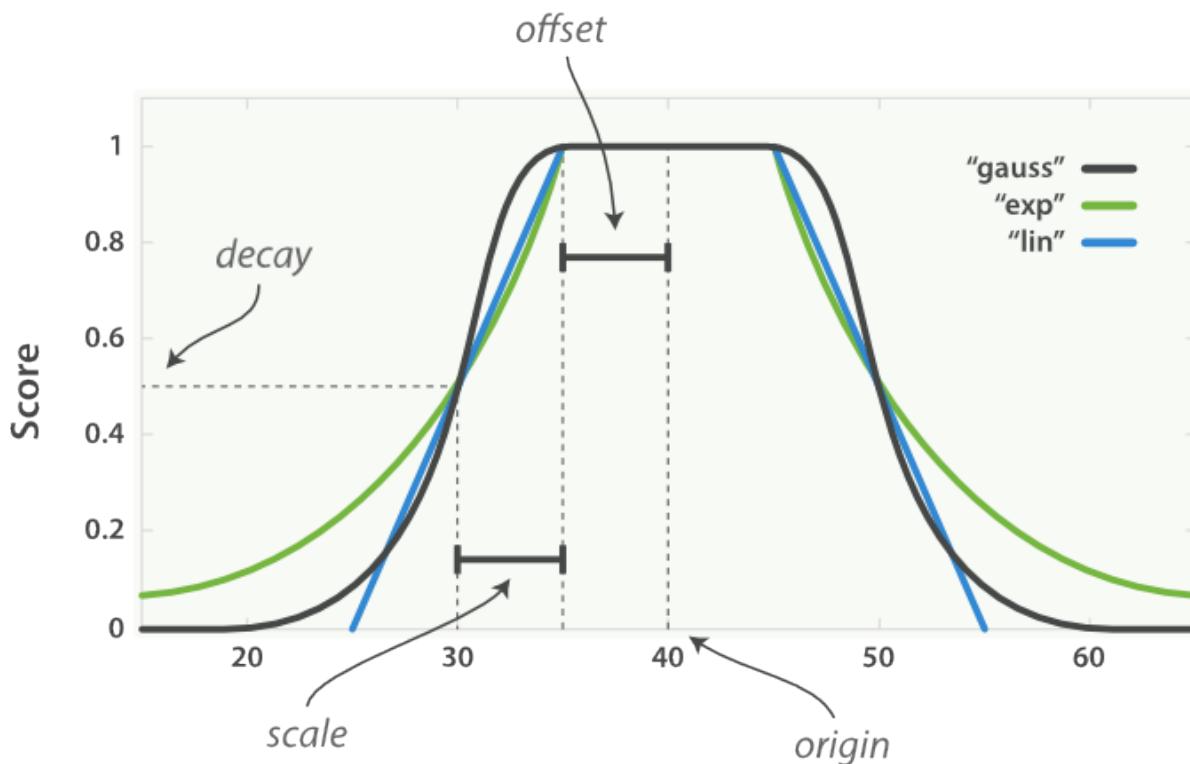


Figure 33. 衰函数曲

衰函数曲中所有曲线的原点 origin (即中心点) 的都是 40, offset 是 5，也就是在 $40 - 5 \leq value \leq 40 + 5$ 内的所有都会被当作原点 origin 理——所有些点的分都是分 1.0。

在此之外，分始衰，衰率由 scale (此例中的 5) 和衰 decay (此例中 0.5) 共同决定。果是所有三个曲在 $origin +/- (offset + scale)$ 的分都是 0.5，即点 30 和 50。

linear、exp 和 gauss (线性、指数和高斯) 函数三者之的区别在于 (origin +/- (offset + scale)) 之外的曲形状：

- **linear** 性函数是条直，一旦直与横 0 相交，所有其他 的 分都是 0.0。
- **exp** 指数函数是先 烈衰 然后 。
- **gauss** 高斯函数是 形的——它的衰 速率是先 慢，然后 快，最后又放 。

曲 的依据完全由期望 分 `_score` 的衰 速率来决定，即距原点 `origin` 的 。

回到我 的例子：用 希望租一个 敦市中心近 ({ "lat": 51.50, "lon": 0.12}) 且 不超 £100 英 的度假屋，而且与距 相比，我 的用 格更 敏感， 可以写成：

```
GET /_search
{
  "query": {
    "function_score": {
      "functions": [
        {
          "gauss": {
            "location": { ①
              "origin": { "lat": 51.5, "lon": 0.12 },
              "offset": "2km",
              "scale": "3km"
            }
          }
        },
        {
          "gauss": {
            "price": { ②
              "origin": "50", ③
              "offset": "50",
              "scale": "20"
            }
          },
          "weight": 2 ④
        }
      ]
    }
  }
}
```

- ① `location` 字段以地理坐 点 `geo_point` 映射。
- ② `price` 字段是数 。
- ③ 参 理解 格 句，理解 `origin` 什 是 50 而不是 100。
- ④ `price` 句是 `location` 句 重的 倍。

`location` 句可以 理解：

- 以 敦市中作 原点 `origin`。
- 所有距原点 `origin` 2km 内的位置的 分是 1.0。

- 距中心 5km (`offset + scale`) 的位置的 分是 0.5。

理解 price 格 句

`price` 句使用了一个小技巧：用 希望 £100 英 以下的度假屋，但是例子中的原点被 置成 £50 英 ， 格不能 ， 但肯定是越低越好，所以 £0 到 £100 英 内的所有 格都 是比 好的。

如果我 将原点 `origin` 被 置成 £100 英 ， 那 低于 £100 英 的度假屋的 分会 低，与其 不如将原点 `origin` 和偏移量 `offset` 同 置成 £50 英 ， 就能使只有在 格高于 £100 英 (`origin + offset`) 分才会 低。

TIP `weight` 参数可以被用来 整 个 句的 献度，重 `weight` 的 是 1.0 。 个 会先与 个句子的 分相乘，然后再通 `score_mode` 的 置方式合并。

脚本 分

最后，如果所有 `function_score` 内置的函数都无法 足 用 景，可以使用 `script_score` 函数自行 。

个例子，想将利 空 作 因子加入到相 度 分 算，在 中，利 空 和以下三点相 ：

- `price` 度假屋 的 格。
- 会 用 的 ——某些等 的用 可以在 房 高于某个 `threshold` 格的 时候享受折扣 `discount`。
- 用 享受折扣后， 的 房 的利 `margin`。

算 个度假屋利 的算法如下：

```
if (price < threshold) {
    profit = price * margin
} else {
    profit = price * (1 - discount) * margin;
}
```

我 很可能不想用 利 作 分， 会弱化其他如地点、受 迎度和特性等因子的作用，而是将利 用 目 利 `target` 的百分比来表示，高于 目 的利 空 会有一个正向 分(大于 1.0)，低于目 的利 空 会有一个 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` 作 的脚本 言，它与JavaScript很像，上面 个算法用 `Groovy`

脚本表示如下：

```
price = doc['price'].value ①
margin = doc['margin'].value ①

if (price < threshold) { ②
    return price * margin / target
}
return price * (1 - discount) * margin / target ②
```

① `price` 和 `margin` 量可以分 从文 的 `price` 和 `margin` 字段提取。

② `threshold`、`discount` 和 `target` 是作 参数 `params` 入的。

最 我 将 `script_score` 函数与其他函数一起使用：

```
GET /_search
{
  "function_score": {
    "functions": [
      { ...location clause... }, ①
      { ...price clause... }, ①
      {
        "script_score": {
          "params": { ②
            "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;" ③
        }
      }
    ]
  }
}
```

① `location` 和 `price` 句在 衰 函数 中解 。

② 将 些 量作 参数 `params` ， 我 可以 改 脚本无 重新 。

③ JSON 不能接受内嵌的 行符，脚本中的 行符可以用 \n 或 ; 符号替代。

个 根据用 地点和 格的需求，返回用 最 意的文 ，同 也考 到我 于盈利的要求。

`script_score` 函数提供了巨大的活性，可以通过脚本文里的所有字段、当前分 `_score` 甚至逆向文率和字段度的信息（参 see [脚本文本分](#)）。

有人使用脚本性能会有影响，如果脚本运行慢，可以有以下三：

TIP

- 尽可能多的提前算各信息并将结果存入一个文本中。
- Groovy很快，但没Java快。可以将脚本用原生的Java脚本重新。参 [原生Java脚本](#)。
- 那些最佳分的文本用脚本，使用[重新分](#)中提到的`rescore`功能。

可的相似度算法

在一相度和分之前，我会以一个更高的束本章的内容：可的相似度算法（Pluggable Similarity Algorithms）。Elasticsearch 将用分算法作相似度算法，它也能支持其他的一些算法，这些算法可以参考[相似度模型文](#)。

Okapi BM25

能与TF/IDF和向量空模型美的就是 [Okapi BM25](#)，它被是当今最先的排序函数。BM25源自概率相模型（probabilistic relevance model），而不是向量空模型，但这个算法也和Lucene的用分函数有很多共通之。

BM25同使用、逆向文率以及字段一化，但是个因子的定都有微区。与其解BM25公式，倒不如将注点放在BM25所能来的号上。

和度

TF/IDF和BM25同使用逆向文率来区分普通（不重要）和非普通（重要），同（参）文里的某个出次数越繁，文与个就越相。

不幸的是，普通随可，上一个普通在同一个文中大量出的作用会由于在所有文中的大量出而被抵消掉。

曾有个期，将最普通的（或停用，参[停用](#)）从索引中移除被是一准践，TF/IDF正是在背景下生的。TF/IDF没有考上限的，因高停用已被移除了。

Elasticsearch的[standard](#)准分析器（`string`字段使用）不会移除停用，因尽管些的重要性很低，但也不是无用。致：在一个相当的文中，像`the`和`and`出的数量会高得，以致它的重被人放大。

一方面，BM25有一个上限，文里出5到10次的会比那些只出一次的相度有着显著影。但是如[TF/IDF与BM25的和度所](#)，文中共出20次的几乎与那些出上千次的有着相同的影。

就是非性和度（*nonlinear term-frequency saturation*）。

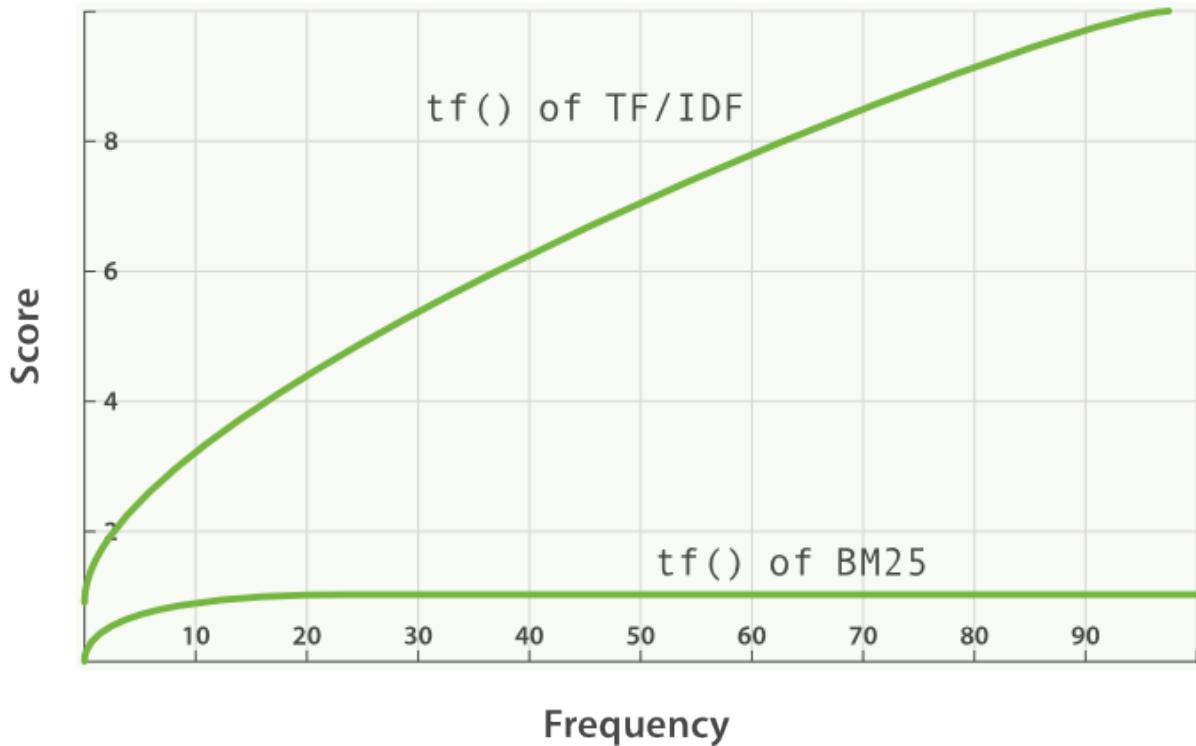


Figure 34. TF/IDF 与 BM25 的和度

字段 度 一化 (Field-length normalization)

在 字段 一化 中，我 提到 Lucene 会 短字段比 字段更重要：字段某个 的 度所来的重要性会被 个字段 度抵消，但是 的 分函数会将所有字段以同等方式 待。它 所有 短的 title 字段比所有 的 body 字段更重要。

BM25 当然也 短字段 有更多的 重，但是它会分 考 个字段内容的平均 度，就能区分短 title 字段和 title 字段。

CAUTION 在 重提升 中，已 title 字段因 其 度比 body 字段 自然 有更高的 重提升 。由于字段 度的差 只能 用于 字段， 自然的 重提升会在使用 BM25 消失。

BM25

不像 TF/IDF，BM25 有一个比 好的特性就是它提供了 个可 参数：

k1

个参数控制着 果在 和度中的上升速度。 1.2 。 越小 和度 化越快， 越大 和度 化越慢。

b

个参数控制着字段 一 所起的作用， 0.0 会禁用 一化， 1.0 会 用完全 一化。 0.75 。

在 践中， BM25 是 外一回事， k1 和 b 的 用于 大多数文 集合，但最 是会因

文 集不同而有所区 , 了 到文 集合的最 , 就必 反参数 行修改 。

更改相似度

相似度算法可以按字段指定, 只需在映射中 不同字段 定即可 :

```
PUT /my_index
{
  "mappings": {
    "doc": {
      "properties": {
        "title": {
          "type": "string",
          "similarity": "BM25" ①
        },
        "body": {
          "type": "string",
          "similarity": "default" ②
        }
      }
    }
  }
}
```

① `title` 字段使用 BM25 相似度算法。

② `body` 字段用 相似度算法 (参 用 分函数)。

目前, Elasticsearch 不支持更改已有字段的相似度算法 `similarity` 映射, 只能通过数据重新建立索引来 到目的。

配置 BM25

配置相似度算法和配置分析器很相似, 自定 相似度算法可以在 建索引 指定, 例如 :

```

PUT /my_index
{
  "settings": {
    "similarity": {
      "my_bm25": { ①
        "type": "BM25",
        "b": 0 ②
      }
    }
  },
  "mappings": {
    "doc": {
      "properties": {
        "title": {
          "type": "string",
          "similarity": "my_bm25" ③
        },
        "body": {
          "type": "string",
          "similarity": "BM25" ④
        }
      }
    }
  }
}

```

① 建一个基于内置 BM25，名 my_bm25 的自定 相似度算法。

② 禁用字段 度 化 (field-length normalization)。参 [BM25](#)。

③ title 字段使用自定 相似度算法 my_bm25。

④ 字段 body 使用内置相似度算法 BM25。

TIP 自定 的相似度算法可以通 索引，更新索引 置， 索引 个 程 行更新。 可以无 重建索引又能 不同的相似度算法配置。

相 度是最后 10% 要做的事情

本章介 了 Lucene 是如何基于 TF/IDF 生成 分的。理解 分 程是非常重要的，就可以根据具体的 分 果 行 、 、 弱和定制。

践中， 的 合就能提供很好的搜索 果，但是 了 得 具有成效 的搜索 果，就必 反推敲修改前面介 的 些 方法。

通常， 策略字段 用 重提升，或通 句 的 整来 某个句子的重要性 些方法，就足以 得良好的 果。有 ，如果 Lucene 基于 的 TF/IDF 模型不再 足 分需求（例如希望基于 或距 来 分）， 需要更具侵略性的 整。

除此之外，相 度的 就有如兔子洞，一旦跳 去就很 再出来。 最相 个概念是一个

以触及的模糊目，通常不同人 文 排序又有着不同的想法，很容易使人陷入持 反 整而没有明
展的怪圈。

我 烈建 不要陷入 怪圈，而要 控 量搜索 果。控用 点 最 端 果的 次，可以是前 10
个文，也可以是第一 的；用 不 看首次搜索的 果而直接 行第二次 的 次；用 来回点 并
看搜索 果的 次，等等 如此 的信息。

些都是用来 搜索 果与用 之 相 程度的指 。如果 能返回高相 的文，用 会 前五中
的一个，得到想要的 果，然后 。不相 的 果会 用 来回点 并 新的搜索条件。

一旦有了 些 控手段，想要 就并不 ，作 整，控用 的行 改 并做 当反 。本
章介 的一些工具就只是工具而已，要想物尽其用并将搜索 果提高到
水平，唯一途径就是需要具 能 度量用 行 的 大能力。

理人 言

``我 句 里的所有 ，但并不能理解全句。"

— Matt Groening

全文搜索是一 准率 与 全率 之 的 量— 准率即尽量返回
少的无 文，而 全率 尽量返回 多的相 文。 尽管能 精准匹配用 的，但 然不
，我 会 很多被用 是相 的文。 因此，我 需要把
撒得更广一些，去搜索那些和原文不是完全匹配但却相 的。

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

如果文 包含用 的内容，那 些文 当出 在返回 果的最前面，而匹配程度 低的文
将会排在 后的位置。 如果没有任何完全匹配的文，我 至少可以 用 展示一些潜在的匹配
果；它 甚至可能就是用 最初想要的 果。

以下列出了一些可 化的地方：

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

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

在之前，我看看如何更快更地始。

始 理各 言

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

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

些分析器承担以下四角色：

- 文本拆分：

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

- 大写 小写：

The → the

- 移除常用的停用：

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

- 将型（例如数，去式）化根：

foxes → fox

了更好的搜索性，个言的分析器提供了言的具体：

- 英分析器移除了所有格's

John's → john

- 法分析器移除了元音省略例如 l' 和 qu' 和 音符号例如 " 或 ^：

l'église → eglis

- 分析器化了切，将切中的 ä 和 ae 替 a，或将 ß 替 ss：

äußerst → ausserst

使用言分析器

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

```

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

```

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

当然，文本 english 分析 理，我 会 失源数据：

```

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

```

① 切 : i'm, happy, about, fox

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

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

```

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

```

- ① 主 `title` 字段使用 `standard` (准) 分析器。
② `title.english` 子字段使用 `english` (英) 分析器。

替 字段映射后，我 可以索引一些 文 来展示 在搜索 使用 个字段：

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

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

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

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

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

配置 言分析器

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

干提取排除

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

自定 停用

英 中 的停用 列表如下：

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

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

了自定 `english` (英) 分 器的行 ， 我 需要基于 `english` (英) 分析器 建一个自定

分析器，然后添加一些配置：

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

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

① 防止 organization 和 organizations 被干

② 指定一个自定 停用 列表

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

我在将原根和停用：性能与精度中分了干提取和停用。

混合言的陷

如果只需要理一言，那就很幸。到一个正的策略用于理多言文是一巨大的挑。

在索引的候

多言文主要有以下三个型：

- 一是 document (文) 有自己的主言，并包含一些其他言的片段（参考 文一言。）
- 一是个 field (域) 有自己的主言，并包含一些其他言的片段（参考 个域一言。）
- 一是个 field (域) 都是混合言（参考 混合言域。）

(分)目不是可以，我当保持将不同言分隔。在同一倒排索引内混合多言可能造成一些。

不合理的 干提取

的 干提取 跟英 , 法 , 瑞典 等是不一 的。 不同的 言提供同 的 干提 将会致有的 的 根 的正 , 有的 的 根 的不正 , 有的 根本 不到 根。 甚至是将不同言的不同含 的 切 同一个 根, 合并 些 根的搜索 果会 用 来困 。

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

写方式一 干提取器

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

- Предупреждение -

- Warning

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

不正 的倒排文 率

在 [什 是相 性?](#) (相 性教程) 中, 一个 term () 在一 文 中出 的 率 高, term () 的重就越低。 为了精 的 算相 性, 需要精 的 term-frequency () 。

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

在搜索的 候

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

用 的搜索也注意有三个方面:

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

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

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

言

很可能已 知道 的文 所 用的 言，或者 的文 只是在 自己的 内 写并被翻 成 定的一系
列 言。人 的 可能是最可 的将 言正 的方法。

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

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

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

文 一 言

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

```

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

```

```

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

```

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

② `title.stemmed` 子域使用了具体 言的分析器。

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

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

```

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

```

① 个 会在所有以 `blogs-` 的索引中 行。

② `title.stemmed` 字段使用 个索引中指定的分析器 。

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

外

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

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

不要 言使用 型

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

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

个域一 言

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

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

翻 存 在不同的域中，根据域的 言决定使用相 的分析器：

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

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

② 其他字段使用 合自己 言的分析器。

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

文的一言可以独，也可以通过多个域来多言。我甚至可以通过特定言置偏好来提高字段先：

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

① 一个搜索所有以 title 前的域，但是 title_es 域加重 2。其他的所有域是中性重 1。

混合言域

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

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

正的理多言型文是非常困的。即使所有的域使用 standard (准) 分析器，但的文会得不利于搜索，除非使用了合的干提取器。当然，不可能只一个干提取器。

干提取器是由言具体决定的。或者，干提取器是由言和脚本所具体决定的。像在

写方式一 干提取器中那。如果个言都使用不同的脚本，那干提取器就可以合并了。

假的混合言使用的是同一的脚本，例如拉丁文，有三个可用的：

- 切分到不同的域
- 行多次分析
- 使用 n-grams

切分到不同的域

在言提到的的言可以告部分文属于言。可以用**一个域一言**中用的一的方法来根据言切分文本。

行多次分析

如果主要理数量有限的言，可以使用多个域，言都分析文本一次。

```

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

```

① 主域 `title` 使用 `standard` (准) 分析器

② 个子域提供不同的 言分析器来 `title` 域文本 行分析。

使用 n-grams

可以使用 `Ngrams` 在 合 的 用 中描述的方法索引所有的
化包含 添加一个后 (或在一些 言中添加前)，所以通 将
有很大的机会匹配到相似但不完全一 的 。 个可以 合
(多次分析) 方法 不支持的 言提供全域 取：

n-grams。 大多数 型
拆成 n-grams,
analyze-multiple times

```

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

```

① 在 `analysis` 章，我 按照 Ngrams 在 合 的 用 中描述的定 了同 的 `trigrams` 分析器。

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

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

```

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

```

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

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

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

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

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

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

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

准分析器

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

```

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

```

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

(小写字母) 和 `stop` (停用) 元器, 但是 在, 我 注于 `standard tokenizer` (准分器)。

准分 器

分器接受一个字符串作入, 将一个字符串拆分成独立的或元(`token`) (可能会一些点符号等字符), 然后出一个元流(`token stream`)。

有趣的是用于的算法。`whitespace` (空白字符) 分器按空白字符——空格、`tabs`、行符等等行拆分——然后假定的非空格字符成了一个元。例如：

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

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

`letter` 分器, 采用外一策略, 按照任何非字符行拆分, 将会返回如下：`You`、`re`、`the`、`st`、`runner`、`home`。

`standard` 分器使用 Unicode 文本分割算法 (定来源于 [Unicode Standard Annex #29](#)) 来之的界限, 并且出所有界限之的内容。Unicode 内含的知使其可以成功的包含混合言的文本行分。

点符号可能是的一部分, 也可能不是, 取决于它出的位置：

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

在个例子中, `You're` 中的号被的一部分, 然而'`favorite`'中的引号不会被的一部分, 所以分果如下：`You're`、`my`、`favorite`。

TIP `uax_url_email` 分器和 `standard` 分器工作方式其相同。区只在于它能email地址和 URLs 并出一个元。`standard` 分器不一, 会将 email 地址和 URLs 拆分成独立的。例如, email 地址 `joe-bloggs@foo-bar.com` 的分果 `joe`、`bloggs`、`foo`、`bar.com`。

`standard` 分器是大多数言分的一个合理的起点, 特是西方言。事上, 它成了大多数特定言分析器的基, 如 `english`、`french` 和 `spanish` 分析器。它也支持洲言, 只是有些陷, 可以考通ICU件的方式使用 `icu_tokenizer` 行替。

安装 ICU 件

Elasticsearch的 [ICU 分析器](#) 件使用国化件 `Unicode` (ICU) 函数 (情看 [site.project.org](#)) 提供富的理 Unicode 工具。些包含理洲言特有用的 `icu 分器`, 有大量除英外其他言行正匹配和排序所必的分器。

NOTE ICU 件是理英之外言的必需工具，非常推安装并使用它，不幸的是，因是基于外的ICU函数，不同版本的ICU件可能并不兼容之前的版本，当更新件的时候，需要重新索引的数据。

安装个件，第一先掉的Elasticsearch点，然后在Elasticsearch的主目行以下命令：

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

①当前\$VERSION（版本）可以在以下地址到<https://github.com/elasticsearch/elasticsearch-analysis-icu>。

一旦安装后，重Elasticsearch，将会看到似如下的一条日志：

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

如果有很多点并以集群方式行的，需要在集群的个点都安装个件。

icu_分 器

icu_分器和准分器使用同的Unicode文本分段算法，只是了更好的支持洲，添加了泰、老、中文、日文、和文基于典的方法，并且可以使用自定将和柬埔寨文本拆分成音。

例如，分比准分器和icu_分器在分泰中的'Hello. I am from Bangkok.'生的元：

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

准分器生了个元，个句子一个：

个只是想搜索整个句子'I am from Bangkok.'的时候有用，但是如果想搜索'Bangkok.'不行。

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

相反，icu_分器可以把文本分成独立的（），使得文更容易被搜索到。

相而言，准分器分中文和日文的时候“度分”了，常将一个完整的拆分独立的字符，因为并没有空格，很难区分的字符是隔的，是一个句子中的字：

- 向的意思是facing（面），日的意思是sun（太），葵的意思是hollyhock（蜀葵）。当写在一起的时候，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），雨的意思是rain（下雨）。第一个和第二个字符写在一起成了五月，意思是the month of May（一年中的五月），

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

然一个字符本身可以是一个，但使元保持更大的原始概念比使其作一个的一部分要有意的多：

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

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

准分器 在前面的例子中将一个字符输出为独立的元：向，日，葵。**icu分器**会输出一个元 **向日葵** (*sunflower*)。

<code>准分器</code> 和 <code>icu分器</code> 的一个不同的地方是后者会将不同写方式的字符（例如，<code>β</code>）拆分成独立的元 — <code>β</code> 和 <code>ε</code>—，而前者会输出一个元：<code>βε</code>。

整理入文本

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

HTML 分

将 HTML 通**准分器**或**icu分器**分将生糟的果。些分器不知道如何理 HTML。例如：

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

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

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

字符器可以通**analyze** API 行，需要在字符串中指明它：

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

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

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

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

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

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

整理 点符号

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

不幸的是， Unicode 列出了一些有 会被用 号的字符：

`U+0027`

号 (<code>'</code>)— 原始 ASCII 符号

`U+2018`

左 引号 (<code>‘</code>)— 当 引用 作 一个引用的 始

`U+2019`

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

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

`U+201B`

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

`U+0091`

ISO-8859-1 中的左 引号 — 不会被用于 Unicode 中

ISO-8859-1 中的右 引号 — 不会被用于 Unicode 中

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

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

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

```
PUT /my_index
{
  "settings": {
    "analysis": {
      "char_filter": { ①
        "quotes": {
          "type": "mapping",
          "mappings": [ ②
            "\u0091=>\u0027",
            "\u0092=>\u0027",
            "\u2018=>\u0027",
            "\u2019=>\u0027",
            "\u201b=>\u0027"
          ]
        }
      },
      "analyzer": {
        "quotes_analyzer": {
          "tokenizer": "standard",
          "char_filter": [ "quotes" ] ③
        }
      }
    }
  }
}
```

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

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

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

像以前一 ， 我 需要在 建了分析器后 它：

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

个例子返回如下元，其中所有的中的引号都被替了号： You're, my, favorite, M'Coy。

投入更多的努力保的分器接收到高量的入，的搜索果量也将更好。

一化 元

把文本切割成元(token)只是工作的一半。了些元(token)更容易搜索，些元(token)需要被一化(normalization)--个程会去除同一个元(token)的无意差，例如大写和小写的差。可能我需要去掉有意的差，esta、ésta 和 está 都能用同一个元(token)来搜索。会用 déjà vu 来搜索，是 déjà vu?

些都是元器的工作。元器接收来自分器(tokenizer)的元(token)流。可以一起使用多个元器，一个都有自己特定的理工作。一个元器都可以理来自一个元器出的流。

个例子

用的最多的元器(token filters)是 lowercase 器，它的功能正和期望的一；它将一个元(token)小写形式：

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

① 得到的元(token)是 the, quick, brown, fox

只要和索的分析程是一的，不管用搜索 fox 是 FOX 都能得到一的搜索果。lowercase 器会将 FOX 的求 fox 的求，fox 和我 在倒排索引中存的是同一个元(token)。

了在分析程中使用 token 器，我可以建一个 custom 分析器：

```

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

```

我 可以通 `analyze` API 来 :

```

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

```

① 得到的 元是 `the, quick, brown, fox`

如果有口音

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

- $\beta \Rightarrow ss$
- $\ae \Rightarrow ae$
- $\dot{t} \Rightarrow l$
- $\ddot{m} \Rightarrow m$
- $\ddot{?} \Rightarrow ??$
- $\ddot{2} \Rightarrow 2$
- $\ddot{6} \Rightarrow 6$

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

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

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

① 得到的 元 my, œsophagus, caused, a, debâcle

保留原意

理所当然的，去掉 音符号会 失原意。例如，参考 三个西班牙 ：

esta

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

ésta

esta 的古代用法.

está

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

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

sé

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

se

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

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

相反， 我 文本做 次索引：一次用原文形式，一次用去掉 音符号的形式：

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

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

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

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

```
GET /my_index/_analyze?field=title ①
```

Esta está loca

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

Esta está loca

① 得到的 元 `esta, está, loca`

② 得到的 元 `esta, esta, loca`

可以用更多的文 来 :

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

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

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

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

通过 `validate-query` API 来运行一个可以助理解是如何行的:

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

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

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

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

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

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

TIP

Position 1	Position 2
(ésta,esta)	loca

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

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

Unicode的世界

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

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

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

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

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

Unicode 一化形式 (Normalization Forms)

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

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

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

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

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

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

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

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

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

Unicode 大小写折

人 没有 造力的 就不会是人 ， 而人 的 言就恰恰反映了 一点。

理一个 的大小写看起来是一个 的任 ， 除非遇到需要 理多 言的情况。

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

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

例如：ß，已是小写形式了，会被折_(folded)成ss。似的小写的S被折成o，的，无o，S，和`Σ`出在里，他就可以比了。

'icu_normalizer'元器的一化(normalization)模式是'nfc_cf'。它像'nfc'模式一：

- 合(Composes)字符用最短的字来表示。
- 用兼容(compatibility)模式，把像的字符成的ffi

但是，也会做：

- 大小写折(Case-folds)字符成一合比的形式

句，'nfc_cf'等于'lowercase'元器(token filters)，但是却用于所有的言。on-steroids等于standard分析器，例如：

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

① icu_normalizer是nfc_cf模式。

我来比'Weißenkopfseeadler'和'WEISSKOPFSEEADLER'(大写形式)分通'standard'分析器和我的Unicode自(Unicode-aware)分析器理得到的结果：

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

```
GET /my_index/_analyze?analyzer=my_lowecaser ②
Weißenkopfseeadler WEISSKOPFSEEADLER
```

① 得到的元(token)是weißenkopfseeadler, weisskopfseeadler

② 得到的元(token)是weisskopfseeadler, weisskopfseeadler

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

Unicode 字符折

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

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

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

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

GET /my_index/_analyze?analyzer=my_folder

①

① 阿拉伯数字 被折 成等 的拉丁数字: 12345.

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

```

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

```

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

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

排序和整理

本章到目前为止，我已了解了以搜索为目的去化元。本章中要考的最用例是字符串排序。

在 [字符串排序与多字段](#) (数域) 中，我解除了 Elasticsearch 什不能在 [analyzed](#) (分析) 的字符串字段上排序，并演示了如何同一个域建数域索引，其中 [analyzed](#) 域用来搜索，[not_analyzed](#) 域用来排序。

[analyzed](#) 域无法排序并不是因使用了分析器，而是因分析器将字符串拆分成了很多元，就像一个袋，所以 Elasticsearch 不知道使用那一个元排序。

依于 [not_analyzed](#) 域来排序的不是很活：允我使用原始字符串一定的排序。然而我可以使用分析器来外一排序，只要的分析器是字符串出且有一个的元。

大小写敏感排序

想象下我有三个用文，文的姓名域分含有 [Boffey](#)、[BROWN](#) 和 [bailey](#)。首先我将使用在 [字符串排序与多字段](#) 中提到的技，使用 [not_analyzed](#) 域来排序：

```

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

```

① analyzed name 域用来搜索。

② not_analyzed name.raw 域用来排序。

我 可以索引一些文 用 来 排序：

```

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

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

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

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

```

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

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

句 来 ，我 需要一个能 出 个小写 元的分析器：

```

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

```

① keyword 分 器将 入的字符串原封不 的 出。

② lowercase 分 器将 元 化 小写字母。

使用 **大小写不敏感排序** 分析器替 后， 在我 可以将其用在我 的 数域：

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

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

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

但是一个序是正的？它符合我的期望所以看起来像是正的，但我对的期望可能受到一个事的影：本是英文的，我的例子中使用的所有字母都属于到英字母表。

如果我添加一个姓名 *Böhm* 会？

在我姓名会返回的排序：*bailey*、*Boffey*、*BROWN*、*Böhm*。*Böhm* 会排在 *BROWN*后面的原因是一些依然是按照它表的字排序的。*r* 所存的字 *0x72*，而 *ö* 存的字 *0xF6*，所以 *Böhm* 排在最后。个字符的字都是史的意外。

然，排序序于除英之外的任何事物都是无意的。事上，没有完全“正”的排序。完全取决于使用的言。

言之的区

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

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

NOTE

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

Unicode 算法

是将文本按定序排序的程。*Unicode* 算法或称 UCA (参 www.unicode.org/reports/tr10) 定了一将字符串按照在元表中定的序排序的方法(通常称排序)。

UCA定了*Unicode*排序元素表或称 *DUCET*，*DUCET*无任何言的所有 Unicode 字符定了排序。如所，没有惟一个正的排序，所以 *DUCET*更少的人感到，且尽可能的小，但它不是解决所有排序的万能。

而且，明几乎言都有自己的排序。大多时候使用 *DUCET*作起点并且添加一些自定用来理言的特性。

UCA 将字符串和排序作入，并出二制排序。将根据指定的排序字符串集合行排序化其二制排序的比。

Unicode 排序

TIP

本中描述的方法可能会在未来版本的 Elasticsearch 中更改。看 [icu plugin](#) 文的最新信息。

icu_collation 分器使用 *DUCET* 排序。已是排序的改了。想要使用 *icu_collation* 我需要建一个使用 *icu_collation* 器的分析器：

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

① 使用 DUCET 。

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

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

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

在，我 可以重新索引我 的案例文 并 排序：

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

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

NOTE 注意，一个文档返回的 `sort`，在前面的例子中看起来像 `brown` 和 `böhm`，在看起来像天：乏 `\u0001`。原因是 `icu_collation` 器出用于有效分，不用于任何其他目的。

行一个搜索求反的文排序：`bailey`、`Boffey`、`Böhm`、`BROWN`。一个排序英和来都正，已是一，但是它簿和瑞典来不正。下一我不同的言自定映射。

指定言

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

英

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

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

奥地利

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

簿

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

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

个例子演示建簿排序：

```

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

```

① 首先我 薄 建一个自定 版本的 `icu_collation`。

② 之后我 将其包装在自定 的分析器中。

③ 并且 我 的 `name.sort` 域配置它。

像我 之前那 重新索引并重新搜索：

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

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

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

多排序

言都可以使用 数域来支持 同一个域 行多 排序 :

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

① 我 需要 个排序 建相 的分析器。

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

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

自定 排序

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

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

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

将 原 根

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

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

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

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

的 根形式甚至有可能不是一个真的 ， jumping 和 jumpiness 或 都会被提取 干 jumpi 。 并没有什 一只要在索引 和搜索 生相同的 ， 搜索会正常的工作。

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

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

干度提取就是无法将不同含义的分。例如，`general` 和 `generate` 可能都被提取为 `gener`。干度提取会降低精准度：不相干的文体会在不需要他返回的时候返回。

形 原

原是一样的形式，或典形式—`paying`、`paid` 和 `pays` 的原是 `pay`。通常原很像与其相的，但有也不像—`is`、`was`、`am` 和 `being` 的原是 `be`。

形原，很像干提取，相，但是它比干提取先一的是它企按的
` `，或意。同的可能表出意思；例如，
`wake` 可以表 `to wake up` 或 `a funeral`。然而形原区分个的，干提取却会将其混一。

形原是一更和高源消耗的程，它需要理解出的上下文来决定的意思。践中，干提取似乎比形原更高效，且代价更低。

首先我会下个 Elasticsearch 使用的典干提取器；`` 干提取算法和 `` 字典干提取器；并且在 `` 一个干提取器；为了根据的需要合的干提取器。最后将在 `` 控制干提取和 `` 原形干提取中如何裁剪干提取。

干提取算法

Elasticsearch 中的大部分 stemmers（干提取器）是基于算法的，它提供了一系列用于将一个提取它的根形式，例如剥数末尾的 `s` 或 `es`。提取干并不需要知道的任何信息。

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

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

TIP `kstem token filter` 是一款合并了干提取算法和内置典的英分器。为了避免模糊不正提取，个典包含一系列根和特例。`kstem` 分器相较于 Porter 干提取器而言不那么激。

使用基于算法的干提取器

可以使用 `porter_stem` 干提取器或直接使用 `kstem` 分器，或使用 `snowball` 分器建一个具体言的 Snowball 干提取器。所有基于算法的干提取器都暴露了用来接受言参数的一接口：`stemmer token filter`。

例如，假英分析器使用的干提取器太激并且想使它不那么激。首先在 `language analyzers` 看英分析器配置文件，配置文件展示如下：

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

① keyword_marker 分析器列出那些不用被干提取的词。一个分析器情况下是一个空的列表。

② english 分析器使用了一个干提取器： possessive_english 干提取器和 english 干提取器。所有格干提取器会在任何到 english_stop、english_keywords 和 english_stemmer 之前去除's'。

重新一下 在的配置，添加上以下修改，我 可以把 配置当作新分析器的基本配置：

- 修改 `english_stemmer`，将 `english` ([https://www.elastic.co/guide/en/elasticsearch/reference/master/analysis-porterstem-tokenfilter.html\[port_stem\]](https://www.elastic.co/guide/en/elasticsearch/reference/master/analysis-porterstem-tokenfilter.html[port_stem]) 分析器的映射) 替换为 `light_english` (非激活的 `kstem` 分析器的映射)。
- 添加 `asciifolding` 分析器用以移除外文的附加符号。

• 移除 keyword_marker 分 器，因 我 不需要它。（我 会在 控制 干提取 中 它）

新定 的分析器会像下面 :

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

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

② 添加 asciifolding 分 器

字典 干提取器

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

- 返回不 形式如 feet 和 mice 的正 干
- 区分出 形相似但 不同的情形，比如 organ and organization

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

字典 量

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

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

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

大小与性能

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

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

一方面， 算法化 干提取器通常更 、 量和快速。

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

Hunspell 干提取器

Elasticsearch 提供了基于 典提取 干的 **hunspell** 元 器 (**token filter**)。 Hunspell [hunspell.github.io](https://github.com/hunspell/hunspell.github.io) 是一个 Open Office、 LibreOffice、 Chrome、 Firefox、 Thunderbird 等 多其它 源 目都在使用的 写 器。

可以从 里 取 Hunspell 典 :

- extensions.openoffice.org: 下 解 **.oxt** 后 的文件。
- addons.mozilla.org: 下 解 **.xpi** 展文件。
- [OpenOffice archive](https://archive.org/details/OpenOfficeArchive): 下 解 **.zip** 文件。

一个 Hunspell 典由 个文件 成“**.dic**”和“**.aff**”，具有相同的文件名和 个不同的后 如 `<code>en_US</code>` 和下面的 个后 的其中一个：

.dic

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

.aff

包含 **.dic** 文件 一行代 表 的前 和后

安装一个 典

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

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

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

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

③ 一个 言一个 置文件，下面的章 会具体介 。

按 言 置

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

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

些 的意思如下：

`ignore_case`

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

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

一般来 ， 置参数 `ignore_case true` 是一个好主意。

`strict_affix_parsing`

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

自定 典

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

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

① custom 典和 en_US 典将合并到一起。

② 多个 .aff 文件是不允 的, 因 会 生 冲突。

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

建一个 Hunspell 元 器

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

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

① 参数 language 和目 下 的名称相同。

可以通 analyze API 来 一个新的分析器, 然后和 english 分析器比 一下它 的 出：

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

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

① 返回 `organize`

② 返回 `reorganizes`

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

TIP Hunspell 字典会占用几兆的内存。幸运的是，Elasticsearch 每个点只会建一个字典的实例。所有的分片都会使用一个相同的 Hunspell 分析器。

Hunspell 字典格式

尽管使用 `hunspell` 不必了解 Hunspell 字典的格式，不了解格式可以帮助我编写自己的自定义字典。其很简单。

例如，在美式英语字典（US English dictionary），`en_US.dic` 文件包含了一个包含 `analyze` 的主体，看起来如下：

```
analyze/ADSG
```

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

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

例如，下面的后缀（SFX）是 D。它是，当一个由一个元音（除了 a、e、i、o 或 u 外的任意元音）后接一个 y，那它可以移除 y 和添加 ied 尾（如，`ready` → `readied`）。

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

前面提到的 A、G、D 和 S 如下：

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

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

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

PFX A Y 1			
PFX A 0	re	. ④	

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

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

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

④ 可以添加前 `re` 来形成 `reanalyze`。一个 `re` 可以 合后 `analyze` 一起形成：`reanalyzes`、`reanalyzed`、`reanalyzing`。

了解更多 于 Hunspell 的 法，可以前往 [Hunspell 文](#)。

一个 干提取器

在文 `stemmer` token filter 里面列出了一些 言的若干 干提取器。就英 来 我 有如下提取器：

`english`
`porter_stem` 元 器 (token filter)。

`light_english`
`kstem` 元 器 (token filter)。

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

`lovins`
基于 Snowball 的 Lovins 提取器，第一个 干提取器。

`porter`
基于 Snowball 的 Porter 提取器。

`porter2`
基于 Snowball 的 Porter2 提取器。

possessive_english

Lucene 里面的 EnglishPossessiveFilter，移除 's

Hunspell 干提取器也要 入到上面的列表中， 有多 英文的 典可用。

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

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

提取性能

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

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

提取 量

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

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

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

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

提取程度

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

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

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

果。

做一个

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

控制 干提取

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

元 器 `keyword_marker` 和 `stemmer_override` 能 我 自定 干提取 程。

阻止 干提取

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

在内部, 些 言分析器使用 `keyword_marker` 元 器 来 些 列表 `keywords`, 用来阻止后 的 干提取 器来触 些 。

例如, 我 建一个 自定 分析器, 使用 `porter_stem` 元 器, 同 阻止 `skies` 的 干提取 :

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

① 参数 `keywords` 可以允 收多个 。

使用 `analyze` API 来 ， 可以看到 `skies` 没有被提取：

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

① 返回: `sky, skies, ski, ski`

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

自定 提取

在上面的例子中，我 阻止了 `skies` 被 干提取，但是也 我 希望他能被提干 `sky` 。 The `stemmer_override` 元 器允 我 指定自定 的提取 。 与此同 ， 我 可以 理一些不 的形式，如：`mice` 提取 `mouse` 和 `feet` 到 `foot` :

```

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

```

GET /my_index/_analyze?analyzer=my_english

The mice came down from the skies and ran over my feet ③

① 来自 `original⇒stem`。

② `stemmer_override` 器必 放置在 干提取器之前。

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

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

原形 干提取

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

```

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

```

① 已提取 干的形式和未提取 干的形式位于相同的位置。

Warning：使用此方法前 先 原形 干提取是个好主意。

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

独使用 keyword_repeat token 器将得到以下 果：

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

① 提取 干前后的形式一，所以只是不必要的重。

了防止提取和未提取 干形式相同的 中的无意 重，我 加了 合的 unique 元 器：

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

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

② 元 器必 出 在 干提取器之前。

③ unique_stem 器是在 干提取器完成之后移除重。

原形 干提取是个好主意

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

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

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

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

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

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

基于 些原因，我 不推 使用原形 干提取。

停用 : 性能与精度

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

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

那 些 条可以被 ? 我 可以 分 :

低 (*Low-frequency terms*)

在文 集合中相 出 少的 ，因 它 稀少，所以它 的 重 更高。

高 (*High-frequency terms*)

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

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

到底是低 或是高 取决于它 所 的文 。 `and` 如果在所有都是中文的文 里可能是个低

。在 于数据 的文 集合里， database 可能是一个高 ，它 搜索 个特定集合 无助。

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

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

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

停用 的 点

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

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

- 区分 happy 和 not happy。
- 搜索 名称 The The。
- 士比 的名句 ``To be, or not to be" (生存 是)。
- 使用 威的国家代 :no。

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

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

使用停用

移除停用 的工作是由 stop 停用 器完成的，可以通 建自定 的分析器来使用它（参 使用停用 器<https://www.elastic.co/guide/en/elasticsearch/reference/master/analysis-stop-tokenfilter.html>[stop 停用 器])。但是，也有一些自 的分析器 置使用停用 器：

言分析器

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

standard 准分析器

使用空的停用列表：`none`，`None` 上是禁用了停用。

pattern 模式分析器

使用空的停用列表：`none`，与 `standard` 分析器似。

停用和准分析器 (Stopwords and the Standard Analyzer)

了准分析器能与自定停用表用，我要做的只需建一个分析器的配置好的版本，然后将停用列表入：

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

① 自定的分析器名称 `my_analyzer`。

② 个分析器是一个准 `standard` 分析器，行了一些自定配置。

③ 掉的停用包括 `and` 和 `the`。

TIP 任何语言分析器都可以使用相同的方式配置自定停用。

保持位置 (Maintaining Positions)

`analyzer` API的出果很有趣：

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

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

① position 个 元的位置。

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

指定停用 (Specifying Stopwords)

停用 可以以内 的方式 入，就像我 在前面的例子中那 ，通 指定数：

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

特定 言的 停用 ，可以通 使用 lang 符号来指定：

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

TIP: Elasticsearch 中 定 的与 言相 的停用 列表可以在文 "languages", "predefined_stopword_lists_for")stop 停用 器 中 到。

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

```

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

```

① my_english 分析器是基于 english 分析器。

② 但禁用了停用词。

最后，停用词可以使用一行一个的格式保存在文件中。此文件必须在集群的所有点上，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" ①
        }
      }
    }
  }
}

```

① 停用词文件的路径，路径相对于 Elasticsearch 的 config 目录。

使用停用词器 (Using the stop Token Filter)

当创建 custom 分析器时，可以组合多个 stop 停用词器分器。例如：我想要建一个西班牙语的分析器：

- 自定义停用词列表
- light_spanish 干提取器
- 在 asciifolding 元分器中除去附加符号

我可以通过以下配置完成：

```

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

```

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

② 参 算法提取器（Algorithmic Stemmers）。

③ 器的 序非常重要，下面会 行解。

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

更新停用 (Updating Stopwords)

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

如果 使用 `stopwords` 参数以内 方式指定停用 ，那 只能通 索引，更新分析器的配置<https://www.elastic.co/guide/en/elasticsearch/reference/master/indices-update-settings.html#update-settings-analysis>[`update index settings API`]，然后在重新打 索引才能更新停用 。

如果 使用 `stopwords_path` 参数指定停用 的文件路径 ，那 更新停用 就 了。 只需更新文件(在

一个集群 点上), 然后通 者之中的任何一个操作来 制重新 建分析器:

- 和重新打 索引(参考 [索引的 与](#)),
- 一重 集群下的 个 点。

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

停用 与性能

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

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

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

and 操作符 (and Operator)

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

以下是 `match` :

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

上述 被重写 `bool` 如下 :

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

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

最少匹配数(minimum_should_match)

在精度匹配控制精度的章 里面, 我 使用 **minimum_should_match** 配置去掉 果中次相 的尾。然它只 个目的奏效, 但是也 我 从 面 来一个好 , 它提供 **and** 操作符相似的性能。

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

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

的分 管理

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

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

域相 的停用 (Domain-Specific Stopwords)

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

以下面 例：

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

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

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

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

① 必 匹配至少一个低 / 更重要的 。

② 高 /次重要性 是非必 的。

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

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

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

控制精度

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

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

将被重写 如下所示：

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

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

② 高 可 的, 并且 用于 分使用。

高

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

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

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

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

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

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

更多配置 参 <https://www.elastic.co/guide/en/elasticsearch/reference/master/query-dsl-common-terms-query.html>[`common terms query`]。

停用与短

所有中短匹配大占到5%，但是在慢里面它又占大部分。短性能相差，特别是当短中包括常用的时候，如“*To be, or not to be*”短全部由停用组成，是一端情况。原因在于几乎需要匹配全量的数据。

在停用的面停用的点中，我提到移除停用只能省倒排索引中的一小部分空。句只部分正，一个典型的索引会可能包含部分或所有以下数据：

字典 (*Terms dictionary*)

索引中所有文内所有有序列表，以及包含的文数量。

倒排表 (*Postings list*)

包含个文ID列表。

(*Term frequency*)

个在个文里出的率。

位置 (*Positions*)

个在个文里出的位置，供短或近似使用。

偏移 (*Offsets*)

个在个文里始与束字符的偏移，供高亮使用，是禁用的。

因子 (*Norms*)

用来字段度行化理的因子，短字段予以更多重。

将停用从索引中移除会省字典和倒排表里的少量空，但位置和偏移是一事。位置和偏移数据很容易成索引大小的倍、三倍、甚至四倍。

位置信息

analyzed字符串字段的位置信息是的，所以短能随使用到它。输出的越繁，用来存它位置信息的空就越多。在一个大的文集合中，于那些非常常的，它的位置信息可能占用成百上千兆的空。

行一个高the短可能会致从磁取好几G的数据。些数据会被存到内核文件系的存中，以提高后的速度，看似是件好事，但可能会致其他数据从存中被除，一使后慢。

然是我需要解决的。

索引

我首先自己：是否真的需要使用短或近似？

答案通常是：不需要。在很多用景下，比如日志，我需要知道一个是否在文中（个信息由倒排表提供）而不是心的位置在里。或我要一个字段使用短，但是我完全可以在其他**analyzed**字符串字段上禁用位置信息。

`index_options` 参数允许我控制索引里一个字段存储的信息。可如下:

docs

只存文本及其包含的信息。`not_analyzed` 字符串字段是的。

freqs

存储 `docs` 信息，以及一个在文本里出现的次数。是完成TF/IDF 相度计算的必要条件，但如果只想知道一个文本是否包含某个特定词，无需使用它。

positions

存储 `docs`、`freqs`、`analyzed`，以及一个在文本里出现的位置。`analyzed` 字符串字段是的，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`，以及一个在原始字符串中始与结束字符的偏移信息(`postings highlighter`)。该信息被用以高亮搜索结果，但它也是禁用的。

我可以在索引建的候字段设置 `index_options`，或者在使用 `put-mapping` API 新字段映射的候置。我无法修改已有字段的个置：

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

① `title` 字段使用 `positions` 置，所以它可用于短语或近似匹配。

② `content` 字段的位置置是禁用的，所以它无法用于短语或近似匹配。

停用

除停用是能显著降低位置信息所占空间的一种方式。一个被停用的索引仍然可以使用短语，因剩下的原始位置仍然被保存着，正如 [保持位置 \(Maintaining Positions\)](#) 中看到的那样。尽管如此，将一个短语从索引中排除终究会降低搜索能力，使我得以区分 *Man in the moon* 与 *Man on the moon* 这两个短语。

幸的是，与熊掌是可以兼得的：看 [common_grams](#) 器。

common_grams 器

common_grams 器是短能更高效的使用停用的。它与 shingles 器似(参考相(相)),一个相生成,用示例解更容易。

common_grams 器根据 query_mode 置的不同而生成不同出果: false (索引使用) 或 true (搜索使用),所以我必建个独立的分析器:

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

①首先我基于 common_grams 器建个器: index_filter 在索引使用(此 query_mode 的置是 false), search_filter 在使用(此 query_mode 的置是 true)。

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

③然后我使用器各建一个索引分析器和分析器。

有了自定分析器,我可以建一个字段在索引使用 index_grams 分析器:

```

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

```

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

索引 (At Index Time)

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

```

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

```

新的 `index_grams` 分析器生成以下：

```

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

```

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

字 (Unigram Queries)

因 索引包含 `unigrams`，可以使用与其他字段相同的技 行，例如：

```

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

```

上面一个字符串是通过文本字段配置的 `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：`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`，搜索于任何普通字段都能正常工作。

二元法短语 (Bigram Phrase Queries)

但是，当我运行短语，我可以用新的 `search_grams` 分析器整个编程得更高效：

```

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

```

① 于短语，我重写了新的 `search_analyzer` 分析器，而使用 `search_grams` 分析器。

`search_grams` 分析器会生成以下：

```

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

```

分析器排除了所有常用的 `unigrams`，只留下常用的 `bigrams` 以及低频的 `unigrams`。如 `the_quick` 的 `bigrams` 比个别 `the` 更少，有一个好：

- `the_quick` 的位置信息要比 `the` 的小得多，所以它取磁更快，系统存的影也更小。

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

短 (Two-Word Phrases)

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

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

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

停用 与相 性

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

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

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

同

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

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

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

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

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

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

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

使用同

同 可以取代 有的 元或通 使用 同 元 器，添加到 元流中：

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

①首先，我 定 了一个 同 型的 元 器。

②我 在 同 格式 中 同 格式。

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

TIP

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

通 `analyze` API 来 我 的分析器， 示如下：

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

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

① 所有同 与原始 占有同一个位置。

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

TIP

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

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

同 格式

同 最 的表 形式是 逗号分隔：

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

如果遇到 些 中的任何一 ， 将其替 所有列出的同 。例如：

原始 : 取代:

```
jump      → (jump,leap,hop)
leap      → (jump,leap,hop)
hop       → (jump,leap,hop)
```

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

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

原始 : 取代:

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

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

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

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

展或收

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

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

展

通 展 , 我 可以把同 列表中的任意一个 展成同 列表所有的 :

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

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

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

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

收

收 , 把 左 的多个同 映射到了右 的 个 :

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

它必 同 用于索引和 段, 以 保 映射到索引中存在的同一个 。

相 于 展方法, 方法也有一些 点和一些 点:

索引的大小

索引大小是正常的, 因 只有 一 被索引。

所有 的 IDF 是一 的, 所以 不能区分比 常用的 、不常用的 。

性能

只需要在索引中 到 的出 。

活性

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

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

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

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

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

型 展

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

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

通 在索引 段使用 型 展 :

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

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

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

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

同 和分析

在 同 格式 一章中, 我 使用 u s a 来 例 述一些同 相 的知 。那 什 我 使用的不是 U.S.A. ?原因是, 一个 同 的 元 器只能接收到在它前面的 元 器或者分 器的 出 果 (里看不到原始文本) 。

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

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

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

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

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

但更 的方式将同 器放置在 根 器之后，然后把 根形式的同 列 出来：

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

大小写敏感的同

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

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

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

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

大小不敏感的同 可以是 个 子：

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

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

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

多 同 和短

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

我 建一个同 元 器，然后使用下面 的同 ：

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

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

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

解析器会 出下面 的 果：

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

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

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

但是 些短 会：

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

如果是在段使同，那就会看到更加的匹配果。看下个 validate-query：

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

字会被同元器理成似的信息：

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

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

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

使用收行短

避免混乱的方法是使用收，用个表示所有的同，然后在段，就只需要个行了：

```

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

```

```

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

```

上面那个信息就会被理成似下面：

```

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

```

在我再次行我之前做那个 validate-query，就会出一个又合理的果：

```
"usa is wealthy"
```

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

同与 query_string

本很少到 query_string，因真心不推用它。在一中有提到，由于 query_string 支持一个精的法，因此，可能会影响到它搜出一些出人意料的

果或者甚至是含有 法 的 果。

方式存在不少 ，而其中之一便与多 同 有 。为了支持它的 法，必 用指定的、法所能 的操作符号来 示，比如 AND 、 OR 、 + 、 - 、 field: 等等。（更多相 内容参 query_string 法。）

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

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

符号同

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

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

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

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

我 可以先使用 映射字符 器，在文本被 交 分 器 理之前， 把字符表情替 成符号同 emoticon_happy 或者 emoticon_sad :

```

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

```

```

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

```

① 映射 器把字符从 \Rightarrow 左 的格式 成右 的 子。

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

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

提示： 映射 字符 器是个非常有用的 器，它可以用来 一些已有的字 行替 操作， 如果想要采用更 活的正 表 式去替 字 的 ，那 可以使用 pattern_replace 字符 器。

写

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

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

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

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

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

模糊性

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

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

- 一个字符 替 一个字符 : _f_ox → _b_ox
- 入一个新的字符 : sic → sic_k_
- 除一个字符 : b_l_ack → back

Frederick Damerau 后来在 些操作基 上做了一个 展 :

- 相 个字符的 位 : _st_ar → _ts_ar

个例子, 将 bieber 成 beaver 需要下面几个 :

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

三个 表示 Damerau-Levenshtein edit distance 距 3 。

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

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

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

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

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

模糊

fuzzy 是 term 的模糊等。也很少直接使用它，但是理解它是如何工作的，可以帮助在更高的 match 中使用模糊性。

了解它是如何工作的，我首先索引一些文档：

```
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."}
```

在我可以 surprise 行一个 fuzzy：

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

fuzzy 是一个的，所以它不做任何分析。它通过某个以及指定的 fuzziness 到典中所有的。fuzziness 置 AUTO。

在我例子中，surprise 比 surprise 和 surprised 都在距 2 以内，所以文档 1 和 3 匹配。通过以下，我可以少匹配度到匹配 surprise：

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

提高性能

<code>fuzzy</code>的工作原理是定原始及造一个*自机*—

像表示所有原始字符串指定 距 的字符串的一个大 表。

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

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

prefix_length

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

max_expansions

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

模糊匹配

match 支持 箱即用的模糊匹配：

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

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

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

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

`match` 和 `multi_match` 都支持 `prefix_length` 和 `max_expansions` 参数。

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

模糊性 分

用 喜 模糊 。他 会魔法般的 到正 写 合。很 憾， 效果平平。

假 我 有1000个文 包含 `Schwarzenegger`，只一个文 的出 写 `Schwarzeneger`。根据 `term frequency/inverse document frequency` 理， 个 写 文 比 写正 的相 度更高，因写出 在更少的文 中！

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

TIP 模糊匹配不 用于参与 分—只能在有 写 大匹配 的 。

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

TIP 在模糊 最初出 很少能 独使用。他 更好的作 一个 `bigger` 景的部分功能特性，如 `search-as-you-type` 完成 建 或 `did-you-mean` 短 建 。

音匹配

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

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

得一提的是， 音算法是相当 的，他 初衷 的 言通常是英 或 。 限制了他 的 用性。不 ， 了某些明 的目 ， 并与其他技 相 合， 音匹配能 作 一个有用的工具。

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

然后，可以 建一个使用 音 元 器的自定 分析器，并 下面的方法：

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

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

② 然后在自定 分析器中使用自定 元 器。

在我 可以通 `analyze` API 来 行 ：

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

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

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

```

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

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

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

```

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

可以使用 `match` 来 行搜索：

```

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

```

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

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

聚合

在之前，本致力于搜索。通过搜索，如果我有一个并且希望到匹配一个的文集，就好比在大海中。

通过聚合，我会得到一个数据的概。我需要的是分析和全套的数据而不是一个文：

- 在大海里有多少？
- 的平均度是多少？
- 按照的制造商来分，的度中位是多少？
- 月加入到海中的有多少？

聚合也可以回答更加微的：

- 最受欢迎的制造商是什么？
- 里面有常的？

聚合允许我向数据提出一些的。然功能完全不同于搜索，但它使用相同的数据。意味着聚合的行速度很快并且就像搜索一样几乎是的。

告和表是非常大的。可以表示的数据，立即回，而不是的数据行（需要一周去行的Hadoop任务），的告随着的数据化而化，而不是先算的、的和不相的。

最后，聚合和搜索是一起的。这意味着可以在一个求里同相同的数据行搜索/和分析。并且由于聚合是在用搜索的上下文里算的，不只是示四星酒店的数量，而是示匹配条件的四星酒店的数量。

聚合是如此大以至于多公司已数据分析建立了大型Elasticsearch集群。

高概念

似于DSL表式，聚合也有可合的方法：独立元的功能可以被混合起来提供需要的自定行。意味着只需要学很少的基本概念，就可以得到几乎无尽的合。

要掌握聚合，只需要明白个主要的概念：

桶(Buckets)

足特定条件的文的集合

指(Metrics)

桶内的文行算

就是全部了！一个聚合都是一个或者多个桶和零个或者多个指的合。翻成粗略的SQL句来解：

```
SELECT COUNT(color) ①  
FROM table  
GROUP BY color ②
```

① COUNT(color) 相当于指。

② GROUP BY color 相当于桶。

桶在概念上 似于 SQL 的分 (GROUP BY)，而指 似于 COUNT()、SUM()、MAX() 等 方法。

我 深入 一个概念 并且了解和 一个概念相 的 西。

桶

桶 来 就是 足特定条件的文 的集合：

- 一个雇 属于 男性 桶或者 女性 桶
- 奥 巴尼属于 桶
- 日期2014-10-28属于 十月 桶

当聚合 始被 行， 个文 里面的 通 算来决定符合 个桶的条件。如果匹配到，文 将放入相 的 桶并接着 行聚合操作。

桶也可以被嵌套在其他桶里面，提供 次化的或者有条件的 分方案。例如，辛辛那提会被放入俄亥俄州 个桶，而 整个 俄亥俄州桶会被放入美国 个桶。

Elasticsearch 有很多 型的桶，能 通 很多 方式来 分文 (、最受 迎的 、年 区 、地理位置等等)。其 根本上都是通 同 的原理 行操作：基于条件来 分文 。

指

桶能 我 分文 到有意 的集合，但是最 我 需要的是 些桶内的文 行一些指 的 算。分桶 是一 到目的的手段：它提供了一 文 分 的方法来 我 可以 算感 趣的指 。

大多数 指 是 的数学 算(例如最小 、平均 、最大 ， 有)， 些是通 文 的 来 算。在 践中，指 能 算像平均薪 、最高出 格、95%的 延 的数据。

桶和指 的 合

聚合 是由桶和指 成的。 聚合可能只有一个桶，可能只有一个指，或者可能 个都有。也有可能有一些桶嵌套在其他桶里面。例如，我 可以通 所属国家来 分文 (桶)，然后 算 个国家的平均薪酬 (指)。

由于桶可以被嵌套，我 可以 非常多并且非常 的聚合：

1.通 国家 分文 (桶)

2.然后通 性 分 个国家 (桶)

3.然后通 年 区 分 性 (桶)

4.最后， 个年 区 算平均薪酬 (指)

最后将告 个 <国家, 性 , 年 > 合的平均薪酬。所有的 些都在一个 求内完成并且只遍一次数据！

聚合

我 可以用以下几 定 不同的聚合和它 的 法， 但学 聚合的最佳途径就是用 例来 明。一旦我 得了聚合的思想，以及如何合理地嵌套使用它 ，那 法就 得不那 重要了。

NOTE

聚合的桶操作和度量的完整用法可以在 [Elasticsearch 参考](#) 中 到。本章中会涵 其中很多内容，但在 完本章后 看它会有助于我 它的整体能力有所了解。

所以 我 先看一个例子。我 将会 建一些 汽 商有用的聚合，数据是 于汽 交易的信息：型、制造商、 、何 被出 等。

首先我 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， 始 建我 的第一个聚合。汽 商可能会想知道 个 色的汽 量最好，用聚合可以 易得到 果，用 `terms` 桶操作：

```
GET /cars/transactions/_search
{
  "size" : 0,
  "aggs" : { ①
    "popular_colors" : { ②
      "terms" : { ③
        "field" : "color"
      }
    }
  }
}
```

① 聚合操作被置于参数 `aggs` 之下（如果意，完整形式 `aggregations` 同有效）。

② 然后，可以聚合指定一个我想要名称，本例中是：`popular_colors`。

③ 最后，定一个桶的型 `terms`。

聚合是在特定搜索果背景下行的，也就是它只是求的外一个参数（例如，使用 `/_search` 端点）。聚合可以与，但我会些在限定聚合的（Scoping Aggregations）中来解决个。

NOTE 可能会注意到我将 `size` 置成 0。我并不心搜索果的具体内容，所以将返回数置 0 来提高速度。置 `size: 0` 与 Elasticsearch 1.x 中使用 `count` 搜索型等。

然后我聚合定一个名字，名字的取决于使用者，的果会以我定的名字，用就可以解析得到的果。

随后我定聚合本身，在本例中，我定了一个 `terms` 桶。一个 `terms` 桶会一个到的唯一建新的桶。因我告它使用 `color` 字段，所以 `terms` 桶会一个色建新桶。

我行聚合并看果：

```
{
...
  "hits": {
    "hits": [] ①
  },
  "aggregations": {
    "popular_colors": { ②
      "buckets": [
        {
          "key": "red", ③
          "doc_count": 4 ④
        },
        {
          "key": "blue",
          "doc_count": 2
        },
        {
          "key": "green",
          "doc_count": 2
        }
      ]
    }
  }
}
```

① 因 我 置了 `size` 参数，所以不会有 hits 搜索 果返回。

② `popular_colors` 聚合是作 `aggregations` 字段的一部分被返回的。

③ 个桶的 `key` 都与 `color` 字段里 到的唯一 。它 会包含 `doc_count` 字段，告 我 包含 的文 数量。

④ 个桶的数量代表 色的文 数量。

包含多个桶， 个 一个唯一 色（例如： 或 ）。 个桶也包括 聚合 桶的所有文 的数量。例如，有四 色的 。

前面的 个例子完全是 行的：一旦文 可以被搜到，它就能被聚合。 也就意味着我 可以直接将聚 合的 果源源不断的 入 形 ，然后生成 的 表 。 不久， 又 了一 色的 ，我 的 形就会立即 更新 色 的 信息。

！ 就是我 的第一个聚合！

添加度量指

前面的例子告 我 个桶里面的文 数量， 很有用。但通常，我 的 用需要提供更 的文 度量。 例如， 色汽 的平均 格是多少？

了 取更多信息，我 需要告 Elasticsearch 使用 个字段， 算何 度量。 需要将度量 嵌套 在桶内， 度量会基于桶内的文 算 果。

我 汽 的例子加入 `average` 平均度量：

```
GET /cars/transactions/_search
{
  "size" : 0,
  "aggs": {
    "colors": {
      "terms": {
        "field": "color"
      },
      "aggs": { ①
        "avg_price": { ②
          "avg": {
            "field": "price" ③
          }
        }
      }
    }
  }
}
```

① 度量新 `aggs`。

② 度量指定名字：`avg_price`。

③ 最后，`price` 字段定 `avg` 度量。

正如所，我用前面的例子加入了新的 `aggs`。一个新的聚合 我可以将 `avg` 度量嵌套置于 `terms` 桶内。上，就一个色生成了平均格。

正如 色 的例子，我需要度量起一个名字（`avg_price`）可以后根据名字取它的。最后，我指定度量本身（`avg`）以及我想要算平均的字段（`price`）：

```
{
...
  "aggregations": {
    "colors": {
      "buckets": [
        {
          "key": "red",
          "doc_count": 4,
          "avg_price": { ①
            "value": 32500
          }
        },
        {
          "key": "blue",
          "doc_count": 2,
          "avg_price": {
            "value": 20000
          }
        },
        {
          "key": "green",
          "doc_count": 2,
          "avg_price": {
            "value": 21000
          }
        }
      ]
    }
  }
...
}
```

① 中的新字段 `avg_price`。

尽管 只 生很小改， 上我 得的数据是 了。之前，我 知道有四 色的，在，色的平均 格是 \$32,500 美元。 个信息可以直 接 示在 表或者 形中。

嵌套桶

在我 使用不同的嵌套方案，聚合的力量才能真正得以 。
以及看到如何将一个度量嵌入桶中，它的功能已 十分 大了。

在前例中，我

但真正令人激 的分析来自于将桶嵌套 外一个桶 所能得到的 果。 在，我 想知道 个 色的汽
制造商的分布：

```

GET /cars/transactions/_search
{
  "size" : 0,
  "aggs": {
    "colors": {
      "terms": {
        "field": "color"
      },
      "aggs": {
        "avg_price": { ①
          "avg": {
            "field": "price"
          }
        },
        "make": { ②
          "terms": {
            "field": "make" ③
          }
        }
      }
    }
  }
}

```

① 注意前例中的 `avg_price` 度量 然保持原位。

② 一个聚合 `make` 被加入到了 `color` 色桶中。

③ 一个聚合是 `terms` 桶，它会 个汽 制造商生成唯一的桶。

里 生了一些有趣的事。首先，我 可能会 察到之前例子中的 `avg_price` 度量完全没有 化，在原来的位置。一个聚合的 个 都可以有多个度量或桶，`avg_price` 度量告 我 色汽的平均 格。它与其他的桶和度量相互独立。

我的 用非常重要，因 里面有很多相互 ，但又完全不同的度量需要收集。聚合使我 能 用一次数据 求 得所有的 些信息。

外一件 得注意的重要事情是我 新 的 个 `make` 聚合，它是一个 `terms` 桶（嵌套在 `colors` 、 `terms` 桶内）。意味着它会 数据集中的 个唯一 合生成 (`color`、`make`) 元 。

我 看看返回的 (了 我 只 示部分 果) :

```
{
...
"aggregations": {
  "colors": {
    "buckets": [
      {
        "key": "red",
        "doc_count": 4,
        "make": { ①
          "buckets": [
            {
              "key": "honda", ②
              "doc_count": 3
            },
            {
              "key": "bmw",
              "doc_count": 1
            }
          ]
        },
        "avg_price": {
          "value": 32500 ③
        }
      },
      ...
    ]
  }
}
}
```

① 正如期望的那样，新的聚合嵌入在 一个 色桶中。

② 在我 看 按不同制造商分解的 色下 信息。

③ 最后，我 看到前例中的 `avg_price` 度量 然而 持不 住。

果告 我 以下几点：

- 色 有四 种。
- 色 的平均 价 是 \$32,500 美元。
- 其中三 是 Honda 本田制造，一 是 BMW 宝马 制造。

最后的修改

我 回到 原点，在 入新 之前， 我 的示例做最后一个修改， 算最低和最高的 价 格：

个汽 生成商

```

GET /cars/transactions/_search
{
  "size" : 0,
  "aggs": {
    "colors": {
      "terms": {
        "field": "color"
      },
      "aggs": {
        "avg_price": { "avg": { "field": "price" } }
      },
      "make" : {
        "terms" : {
          "field" : "make"
        },
        "aggs" : { ①
          "min_price" : { "min": { "field": "price"} }, ②
          "max_price" : { "max": { "field": "price"} } ③
        }
      }
    }
  }
}

```

① 我 需要 加 外一个嵌套的 `aggs` 。

② 然后包括 `min` 最小度量。

③ 以及 `max` 最大度量。

得到以下 出（只 示部分 果）：

```
{
...
"aggregations": {
  "colors": {
    "buckets": [
      {
        "key": "red",
        "doc_count": 4,
        "make": {
          "buckets": [
            {
              "key": "honda",
              "doc_count": 3,
              "min_price": {
                "value": 10000 ①
              },
              "max_price": {
                "value": 20000 ①
              }
            },
            {
              "key": "bmw",
              "doc_count": 1,
              "min_price": {
                "value": 80000
              },
              "max_price": {
                "value": 80000
              }
            }
          ]
        },
        "avg_price": {
          "value": 32500
        }
      },
      ...
    ]
  }
}
```

① min 和 max 度量 在出 在 个汽 制造商 (make) 下面。

有了 个桶，我 可以 的 果 行 展并得到以下信息：

- 有四 色 。
- 色 的平均 是 \$32, 500 美元。
- 其中三 色 是 Honda 本田制造，一 是 BMW 宝 制造。
- 最便宜的 色本田 \$10, 000 美元。
- 最 的 色本田 \$20, 000 美元。

条形

聚合 有一个令人激 的特性就是能 十分容易地将它 成 表和 形。本章中， 我 正在通
示例数据来完成各 各 的聚合分析，最 ，我 将会 聚合功能是非常 大的。

直方 histogram 特 有用。 它本 上是一个条形 ，如果有 建 表或分析 表 的 ，那 我
会 无疑 的 里面有一些 表是条形 。 建直方 需要指定一个区 ，如果我 要
建一个直方 ，可以将 隔 20,000。 做将会在 个 \$20,000 建一个新桶，然后文
会被分到 的桶中。

于 表 来 ，我 希望知道 个 区 内汽 的 量。我 会想知道 个 区 内汽 所 来的
收入，可以通 个区 内已 汽 的 求和得到。

可以用 `histogram` 和一个嵌套的 `sum` 度量得到我 想要的答案：

```
GET /cars/transactions/_search
{
  "size" : 0,
  "aggs":{
    "price":{
      "histogram":{ ①
        "field": "price",
        "interval": 20000
      },
      "aggs":{
        "revenue": {
          "sum": { ②
            "field" : "price"
          }
        }
      }
    }
  }
}
```

① `histogram` 桶要求 个参数：一个数 字段以及一个定 桶大小 隔。

② `sum` 度量嵌套在 个 区 内，用来 示 个区 内的 收入。

如我 所 ， 是 `price` 聚合 建的，它包含一个 `histogram` 桶。它要求字段的 型必 是数
型的同 需要 定分 的 隔 。 隔 置 20,000 意味着我 将会得到如 [0-19999, 20000-39999,
…] 的区 。

接着，我 在直方 内定 嵌套的度量， 个 `sum` 度量，它会 落入某一具体 区 的文 中 `price`
字段的 行求和。 可以 我 提供 个 区 的收入，从而可以 到底是普通家用
是奢 。

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
        }
      }
    ]
  }
}
}
```

果很容易理解，不注意到直方的区是区的下限。`0` 代表区 `0-19, 999`，`20000` 代表区 `20, 000-39, 999`，等等。

NOTE

我可能会注意到空的区，比如：`$40, 000-60, 000`，没有出在中。`histogram` 桶会忽略它，因为它有可能会致不希望的潜在出。

我会在下一节中如何包括空桶。返回空桶 [返回空 Buckets](#)。

可以在 [Sales and Revenue per price bracket](#) 中看到以上数据直方的形化表示。

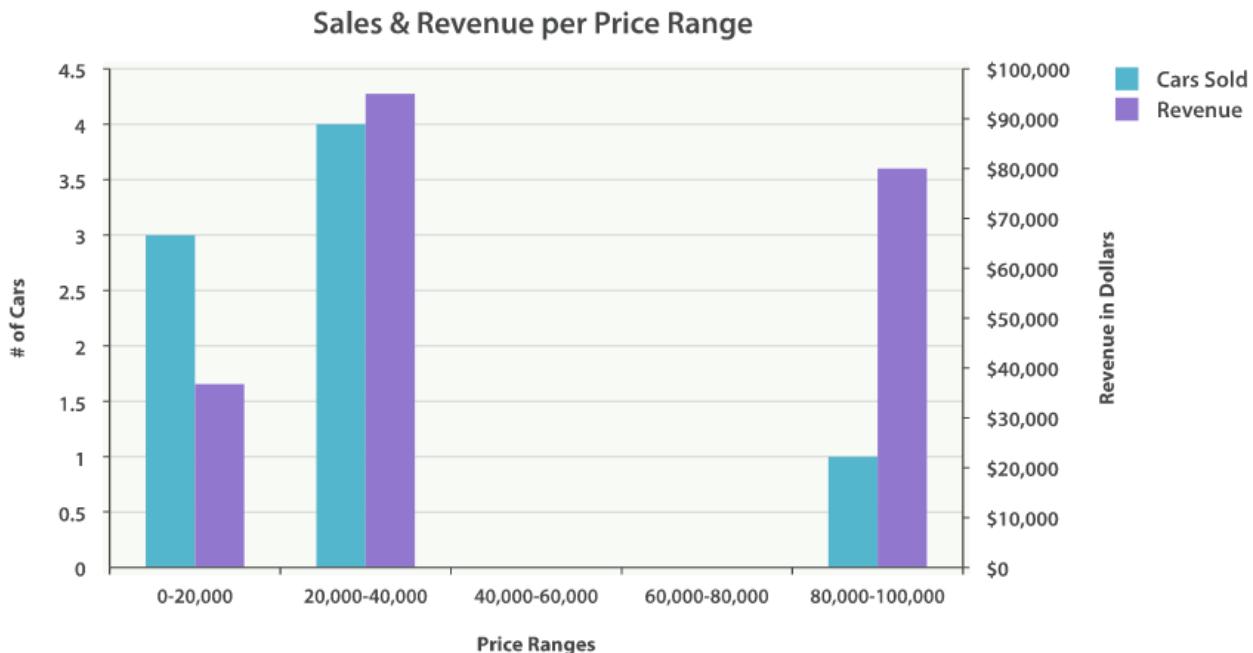


Figure 35. Sales and Revenue per price bracket

当然，我 可以 任何聚合 出的分 和 果 建条形 ，而不只是 直方 桶。 我 以最受 迎 10 汽 以及它 的平均 、 准差 些信息 建一个条形 。 我 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 会按度返回制造商列表以及它 各自的 信息。我 其中的 `stats.avg` 、 `stats.count` 和 `stats.std_deviation` 信息特 感 趣，并用它 算出 准差：

```
std_err = std_deviation / count
```

建 表如 [Average price of all makes, with error bars](#)。

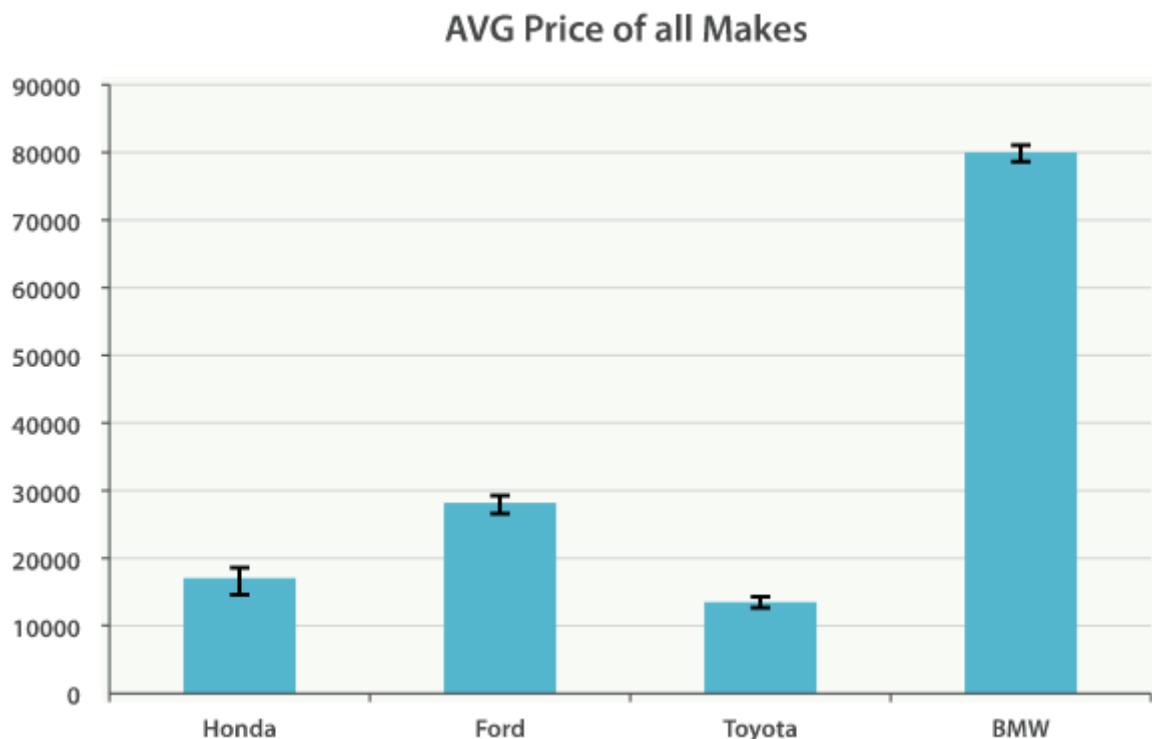


Figure 36. Average price of all makes, with error bars

按

如果搜索是在 Elasticsearch 中使用 率最高的，那 建按 的 date_histogram 随其后。什 会想用 date_histogram ？

假 的数据 。 无 是什 数据 (Apache 事件日志、股票 交易 、棒球) 只要 有 都可以 行 date_histogram 分析。当 的数据有 ， 是想在 度上 建指 分析：

- 今年 月 多少台汽 ？
- 只股票最近 12 小 的 格是多少？
- 我 站上周 小 的平均 延 是多少？

然通常的 histogram 都是条形 ，但 date_histogram 向于 成 状 以展示 序列。 多公司用 Elasticsearch 只是 分析 序列数据。[date_histogram](#) 分析是它 最基本的需要。

[date_histogram](#) 与 通常的 [histogram](#) 似。 但不是在代表数 的数 字段上 建 buckets , 而是在 上 建 buckets。 因此 一个 bucket 都被定 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 行 分析 ?

从技 上来 , 是可以的。 通常的 histogram bucket (桶) 是可以 理日期的。但是它不能自 日期。而用 date_histogram , 可以指定 段如 1 个月 , 它能 明地知道 2 月的天数比 12 月少。 date_histogram 具有 外一个 , 即能合理地 理 区, 可以使 用客 端的 区 行 定制, 而不是用服 器端 区。

通常的 histogram 会把日期看做是数字, 意味着 必 以微秒 位指明 隔。外聚合并不知道日 隔, 使得它 于日期而言几乎没什 用 。

我 的第一个例子将 建一个 的折 来回答如下 : 月 多少台汽 ?

```
GET /cars/transactions/_search
{
  "size" : 0,
  "aggs": {
    "sales": {
      "date_histogram": {
        "field": "sold",
        "interval": "month", ①
        "format": "yyyy-MM-dd" ②
      }
    }
  }
}
```

① 隔要求是日 (如 个 bucket 1 个月)。

② 我 提供日期格式以便 buckets 的 便于 。

我 的 只有一个聚合, 月 建一个 bucket。 我 可以得到 个月 的汽 数量。 外 提供了一个 外的 format 参数以便 buckets 有 "好看的" 。 然而在内部, 日期 然是被 表示成数 。 可能会使得 UI 者抱怨, 因此可以提供常用的日期格式 行格式化以更方便 。

果既符合 期又有一点出人意料 (看看 是否能 到意外之) :

```
{
...
"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
      }
    ]
  ...
}
}
```

聚合结果已完全展示了。正如所见，我有代表月的 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 果末尾 的奇怪之 了 ？

是的， 果没 。 我 的 果少了一些月 ！ `date_histogram` (和 `histogram` 一) 只会返回文
数目非零的 buckets。

意味着 的 histogram 是返回最少 果。通常， 并不想要 。 于很多 用， 可能想直接把
果 入到 形 中，而不想做任何后期加工。

事 上，即使 buckets 中没有文 我 也想返回。可以通 置 个 外参数来 效果：

```
GET /cars/transactions/_search
{
  "size" : 0,
  "aggs": {
    "sales": {
      "date_histogram": {
        "field": "sold",
        "interval": "month",
        "format": "yyyy-MM-dd",
        "min_doc_count" : 0, ①
        "extended_bounds" : { ②
          "min" : "2014-01-01",
          "max" : "2014-12-31"
        }
      }
    }
  }
}
```

① 个参数 制返回空 buckets。

② 个参数 制返回整年。

个参数会 制返回一年中所有月 的 果，而不考 果中的文 数目。 `min_doc_count`
非常容易理解：它 制返回所有 buckets，即使 buckets 可能 空。

`extended_bounds` 参数需要一点解 。 `min_doc_count` 参数 制返回空 buckets，但是 Elasticsearch
只返回 的数据中最小 和最大 之 的 buckets。

因此如果 的数据只落在了 4 月和 7 月之 ，那 只能得到 些月 的 buckets (可能 空也可能不
空)。因此 了得到全年数据，我 需要告 Elasticsearch 我 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 加上了 个 置， 可以把得到的 果 易地直接 入到 的
形 中，从而得到 似汽 的 表。

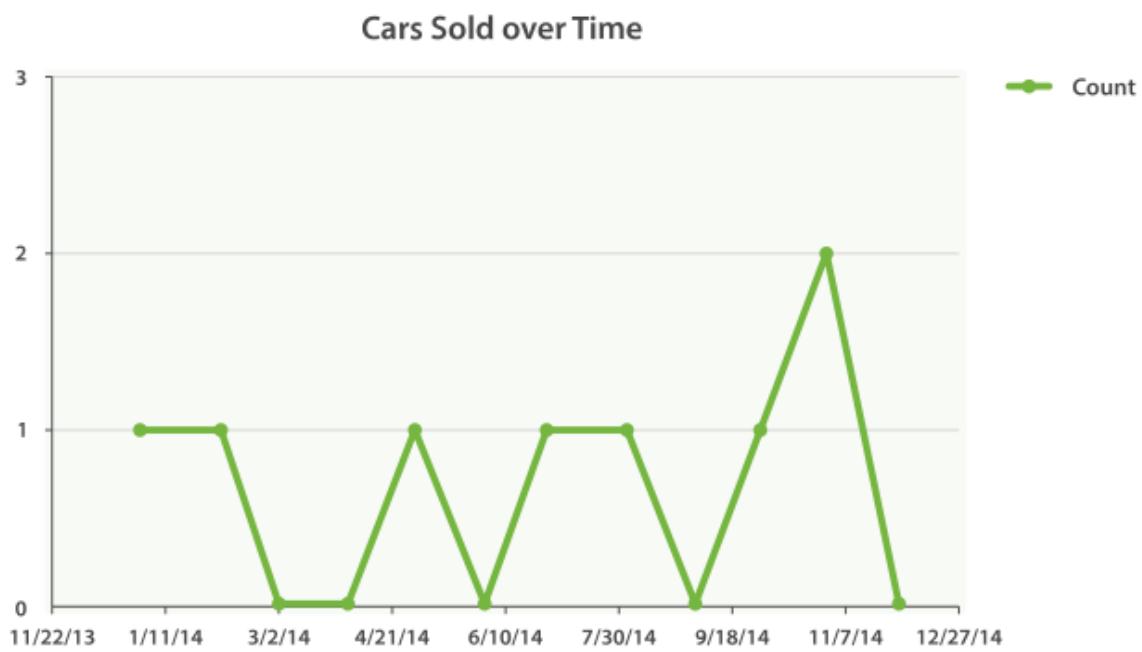


Figure 37. 汽

展例子

正如我 已 很多次, buckets 可以嵌套 buckets 中从而得到更 的分析。 作 例子, 我 建聚合以便按季度展示所有汽 品牌 。同 按季度、按 个汽 品牌 算 , 以便可以 出 品牌最 :

```

GET /cars/transactions/_search
{
  "size" : 0,
  "aggs": {
    "sales": {
      "date_histogram": {
        "field": "sold",
        "interval": "quarter", ①
        "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" } ②
            }
          }
        },
        "total_sum": {
          "sum": { "field": "price" } ③
        }
      }
    }
  }
}

```

① 注意我把隔从 month 改成了 quarter。

② 算品牌的金。

③ 也算所有全部品牌的金。

得到的果（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
              }
            }
          ]
        }
      },
      ...
    ]
  }
}
```

我把果成，得到如按品牌分布的季度所示的的折和个品牌(季度)的柱状。

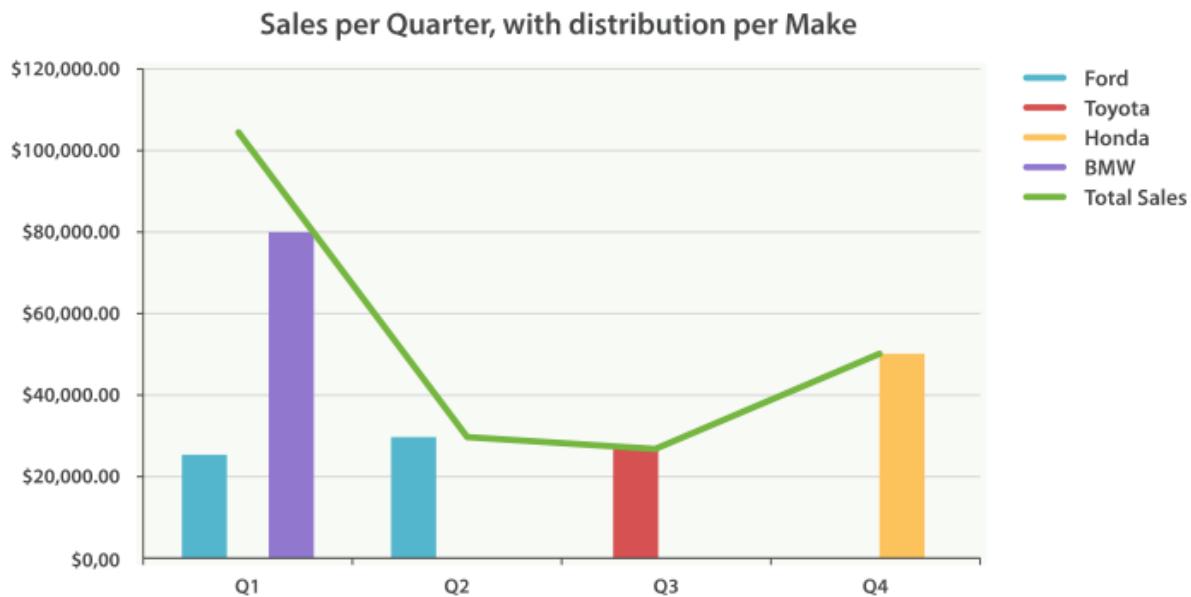


Figure 38. 按品牌分布的 季度

潜力无

些很明 都是 例子，但 表聚合其 是潜力无 的。如 Kibana—用聚合 建的 分析面板 展示了 Kibana 中用各 聚合 建的面板。



Figure 39. Kibana—用聚合 建的 分析面板

因 聚合的 性，似 的面板很容易 、操作和交互。 使得它 成 需要分析数据又不会 建 Hadoop 作 的非技 人 的理想工具。

当然， 了 建 似

Kibana

的 大面板， 可能需要更深的知 ， 比如基于 、

以及排序的聚合。

限定的聚合

所有聚合的例子到目前为止，可能已注意到，我的搜索请求省略了一个 `query`。整个请求只是一个聚合。

聚合可以与搜索请求同行，但是我需要理解一个新概念：。情况下，聚合与是同一行操作的，也就是，聚合是基于我匹配的文集合作用的。

我看看第一个聚合的示例：

```
GET /cars/transactions/_search
{
  "size" : 0,
  "aggs" : {
    "colors" : {
      "terms" : {
        "field" : "color"
      }
    }
  }
}
```

我可以看到聚合是隔开的。其中，Elasticsearch没有指定“和”所有文”是等的。前面一个内部会化成下面的个请求：

```
GET /cars/transactions/_search
{
  "size" : 0,
  "query" : {
    "match_all" : {}
  },
  "aggs" : {
    "colors" : {
      "terms" : {
        "field" : "color"
      }
    }
  }
}
```

因为聚合是在内的结果行操作的，所以一个隔开的聚合上是在`match_all`的结果操作，即所有的文档。

一旦有了的概念，我就能更进一步聚合自行定制。我前面所有的示例都是所有数据计算信息的：量最高的汽车，所有汽车的平均数，最佳月份等等。

利用 ，我 可以 “福特在 有多少 色？” 如此 的 。可以 的在 求中加上一个 (本例中 `match`)：

```
GET /cars/transactions/_search
{
  "query" : {
    "match" : {
      "make" : "ford"
    }
  },
  "aggs" : {
    "colors" : {
      "terms" : {
        "field" : "color"
      }
    }
  }
}
```

因 我 没有指定 `"size" : 0`，所以搜索 果和聚合 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 并没有什 , 但却 高大上的 表 来 至 重要。 加入一个搜索 可以将任何静 的 表板 成一个 数据搜索 。 用 可以搜索数据, 看所有 更新的 形(由于聚合的支持以及 的限定)。 是 Hadoop 无法做到的 !

全局桶

通常我 希望聚合是在 内的, 但有 我 也想要搜索它的子集, 而聚合的 象却是 所有 数据。

例如, 比方 我 想知道福特汽 与 所有 汽 平均 的比 。我 可以用普通的聚合 (

内的) 得到第一个信息, 然后用 全局桶 得第二个信息。

全局桶包含所有的文 , 它无 的。因 它 是一个桶, 我 可以像平常一 将聚合嵌套在内 :

```
GET /cars/transactions/_search
{
  "size" : 0,
  "query" : {
    "match" : {
      "make" : "ford"
    }
  },
  "aggs" : {
    "single_avg_price": {
      "avg" : { "field" : "price" } ①
    },
    "all": {
      "global" : {}, ②
      "aggs" : {
        "avg_price": {
          "avg" : { "field" : "price" } ③
        }
      }
    }
  }
}
```

① 聚合操作在 内 (例如: 所有文 匹配 ford)

② global 全局桶没有参数。

③ 聚合操作 所有文 , 忽略汽 品牌。

single_avg_price 度量 算是基于 内所有文 , 即所有 福特 汽 。avg_price 度量是嵌套在 全局 桶下的, 意味着它完全忽略了 并 所有文 行 算。聚合返回的平均 是所有汽 的平均 。

如果能一直 持 到 里, 知道我 有个真言: 尽可能的使用 器。它同 可以 用于聚合, 在下一 章中, 我 会展示如何 聚合 果 行 而不是 做限定。

和聚合

聚合 限定 有一个自然的 展就是 。因 聚合是在 果 内操作的, 任何可以 用于 的 器也可以 用在聚合上。

如果我 想 到 在 \$10,000 美元之上的所有汽 同 也 些 算平均 , 可以 地使用一个 constant_score 和 filter 束 :

```

GET /cars/transactions/_search
{
  "size" : 0,
  "query" : {
    "constant_score": {
      "filter": {
        "range": {
          "price": {
            "gte": 10000
          }
        }
      }
    },
    "aggs" : {
      "single_avg_price": {
        "avg" : { "field" : "price" }
      }
    }
  }
}

```

正如我 在前面章 中 那 , 从根本上 , 使用 non-scoring 和使用 match 没有任何区 。 (包括了一个 器) 返回一 文 的子集, 聚合正是操作 些文 。使用 filtering query 会忽略 分, 并有可能会 存 果数据等等。

桶

但是如果我 只想 聚合 果 ? 假 我 正在 汽 商 建一个搜索 面, 我 希望 示用 搜索的 果, 但是我 同 也想在 面上提供更 富的信息, 包括 (与搜索匹配的) 上个月度汽 的平均 。

里我 无法 的做 限定, 因 有 个不同的条件。搜索 果必 是 ford , 但是聚合 果必 足 ford AND sold > now - 1M 。

了解决 个 , 我 可以用一 特殊的桶, 叫做 filter (注: 桶) 。 我 可以指定一个 桶, 当文 足 桶的条件 , 我 将其加入到桶内。

果如下 :

```

GET /cars/transactions/_search
{
  "size" : 0,
  "query": {
    "match": {
      "make": "ford"
    }
  },
  "aggs": {
    "recent_sales": {
      "filter": { ①
        "range": {
          "sold": {
            "from": "now-1M"
          }
        }
      },
      "aggs": {
        "average_price": {
          "avg": {
            "field": "price" ②
          }
        }
      }
    }
  }
}

```

① 使用 桶在 基 上 用 器。

② avg 度量只会 ford 和上个月 出的文 算平均 。

因 filter 桶和其他桶的操作方式一，所以可以随意将其他桶和度量嵌入其中。所有嵌套的 件都会 " 承" 个 ，使我 可以按需 聚合 出 部分。

后 器

目前 止，我 可以同 搜索 果和聚合 果 行 （不 算得分的 filter ），以及 聚合 果的一部分 行 （ filter 桶）。

我 可能会想，"只 搜索 果，不 聚合 果 ？" 答案是使用 post_filter 。

它是接收一个 器的 搜索 求元素。 个 器在 之后 行（ 正是 器的名字的由来：它在 之后 post 行）。正因 它在 之后 行，它 没有任何影 ，所以 聚合也不会有任何影 。

我 可以利用 个行 条件 用更多的 器，而不会影 其他的操作，就如 UI 上的各个分 面。 我 汽 商 外一个搜索 面， 个 面允 用 搜索汽 同 可以根据 色来 。 色的 是通 聚合 得的：

```

GET /cars/transactions/_search
{
    "size" : 0,
    "query": {
        "match": {
            "make": "ford"
        }
    },
    "post_filter": { ①
        "term" : {
            "color" : "green"
        }
    },
    "aggs" : {
        "all_colors": {
            "terms" : { "field" : "color" }
        }
    }
}

```

① `post_filter` 元素是 top-level 而且 命中 果 行 。

部分 到所有的 `ford` 汽 , 然后用 `terms` 聚合 建一个 色列表。因 聚合 行操作, 色列表与福特汽 有的 色相 。

最后, `post_filter` 会 搜索 果, 只展示 色 `ford` 汽 。 在 行 后 生, 所以聚合不受影 。

通常 UI 的 一致性很重要, 可以想象用 在界面商 了一 色 (比如: 色), 期望的是搜索 果已 被 了, 而 不是 界面上的 。如果我 用 `filter` , 界面会 上 成只 示 色作 , 不是用 想要的 !

性能考 (*Performance consideration*)

当 需要 搜索 果和聚合 果做不同的 , 才 使用 `post_filter` , 有 用 会在普通搜索使用 `post_filter` 。

WARNING

不要 做! `post_filter` 的特性是在 之后 行, 任何 性能 来的好 (比如 存) 都会完全失去。

在我 需要不同 , `post_filter` 只与聚合并一起使用。

小

合 型的 (如: 搜索命中、聚合或 者兼有) 通常和我 期望如何表 用 交互有 。 合 的 器 (或 合) 取决于我 期望如何将 果呈 用 。

- 在 `filter` 中的 `non-scoring` , 同 影 搜索 果和聚合 果。
- `filter` 桶影 聚合。

- `post_filter` 只影响搜索结果。

多桶排序

多桶（`terms`、`histogram` 和 `date_histogram`）生成很多桶。Elasticsearch 是如何决定一些桶展示用的顺序？

的，桶会根据 `doc_count` 降序排列。是一个好的行，因通常我想要到文中与条件相匹配的最大值：人口数量、率。但有些时候我希望能修改一个顺序，不同的桶有着不同的理方式。

内置排序

些排序模式是桶固有的能力：它操作桶生成的数据，比如 `doc_count`。它共享相同的方法，但是根据使用桶的不同会有些微差。

我做一个 `terms` 聚合但是按 `doc_count` 的升序排序：

```
GET /cars/transactions/_search
{
  "size" : 0,
  "aggs" : {
    "colors" : {
      "terms" : {
        "field" : "color",
        "order": {
          "_count" : "asc" ①
        }
      }
    }
  }
}
```

① 用字 `_count`，我可以按 `doc_count` 的升序排序。

我聚合引入了一个 `order` 象，它允许我可以根据以下几个中的一个行排序：

`_count`

按文档数排序。`terms`、`histogram`、`date_histogram` 有效。

`_term`

按字符串的字母序排序。只在 `terms` 内使用。

`_key`

按一个桶的数排序（理论上与 `_term` 似）。只在 `histogram` 和 `date_histogram` 内使用。

按度量排序

有，我会想基于度量算的结果行排序。在我我的汽分析表中，我可能想按照汽车建一个条状表，但按照汽车平均的升序行排序。

我可以加一个度量，再指定 order 参数引用一个度量即可：

```
GET /cars/transactions/_search
{
  "size" : 0,
  "aggs" : {
    "colors" : {
      "terms" : {
        "field" : "color",
        "order": {
          "avg_price" : "asc" ②
        }
      },
      "aggs": {
        "avg_price": {
          "avg": {"field": "price"} ①
        }
      }
    }
  }
}
```

① 算一个桶的平均。

② 桶按照算平均的升序排序。

我可以采用方式用任何度量排序，只需的引用度量的名字。不有些度量会出多个。
`extended_stats` 度量是一个很好的例子：它出好几个度量。

如果我想使用多度量行排序，我只需以心的度量使用点式路径：

```

GET /cars/transactions/_search
{
  "size" : 0,
  "aggs" : {
    "colors" : {
      "terms" : {
        "field" : "color",
        "order": {
          "stats.variance" : "asc" ①
        }
      },
      "aggs": {
        "stats": {
          "extended_stats": {"field": "price"}
        }
      }
    }
  }
}

```

① 使用 . 符号，根据感 趣的度量 行排序。

在上面 个例子中，我 按 个桶的方差来排序，所以 色 方差最小的会排在 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 点。平均 是根据 个 term 来 算的。 在一定条件下，我 也有可能 更深 的度量 行排序，比如 子桶或从 桶。

我 可以定 更深的路径，将度量用尖括号 (>) 嵌套起来，像 : my_bucket>another_bucket>metric 。

需要提醒的是嵌套路径上的 个桶都必 是 的。 filter 桶生成 一个 桶：所有与 条件匹配的文 都在桶中。多 桶（如：terms） 生成 多桶，无法通 指定一个 定路径来 。

目前，只有三个 桶： filter 、 global 和 reverse_nested 。 我 快速用示例 明，建一个汽 的直方 ，但是按照 色和 色（不包括 色） 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" ①
        }
      },
      "aggs": {
        "red_green_cars": {
          "filter": { "terms": {"color": ["red", "green"]}}}, ②
          "aggs": {
            "stats": {"extended_stats": {"field" : "price"}} ③
          }
        }
      }
    }
}

```

① 按照嵌套度量的方差 桶的直方 行排序。

② 因 我 使用 器 `filter`，我 可以使用嵌套排序。

③ 按照生成的度量 果 行排序。

本例中，可以看到我 如何 一个嵌套的度量。`stats` 度量是 `red_green_cars` 聚合的子 点，而 `red_green_cars` 又是 `colors` 聚合的子 点。 根据 个度量排序，我 定 了路径 `red_green_cars>stats.variance`。我 可以 做，因 `filter` 桶是个 桶。

近似聚合

如果所有的数据都在一台机器上，那 生活会容易 多。 CS201 上教的 典算法就足 付 些 。如果所有的数据都在一台机器上，那 也就不需要像 Elasticsearch 的分布式 件了。不 一旦我 始分布式存 数据，就需要小心地 算法。

有些算法可以分布 行，到目前 止 的所有聚合都是 次 求 得精 果的。 些 型的算法通常 被 是 高度并行的，因 它 无 任何 外代，就能在多台机器上并行 行。比如当 算 `max` 度量 ，以下的算法就非常 ：

1. 把 求广播到所有分片。
2. 看 个文 的 `price` 字段。如果 `price > current_max`，将 `current_max` 替 成 `price`。
3. 返回所有分片的最大 `price` 并 点。
4. 到从所有分片返回的最大 `price`。 是最 的最大 。

个算法可以随着机器数的性而横向展，无任何操作（机器之不需要中果），而且内存消耗很小（一个整型就能代表最大）。

不幸的是，不是所有的算法都像取最大。更加的操作需要在算法的性能和内存使用上做出衡。于个，我有个三角因子模型：大数据、精性和性。

我需要其中：

精 +

数据可以存入台机器的内存之中，我可以随心所欲，使用任何想用的算法。果会100%精，会相快速。

大数据 + 精

的Hadoop。可以理PB的数据并且我提供精的答案，但它可能需要几周的才能我提供个答案。

大数据 +

近似算法我提供准但不精的果。

Elasticsearch目前支持近似算法（`cardinality` 和 `percentiles`）。它会提供准但不是100%精的果。以牲一点小小的估算代，些算法可以我来高速的行效率和小的内存消耗。

于大多数用域，能返回高度准的果要比100%精果重要得多。乍一看可能是天方夜。有人会叫“我需要精的答案！”。但仔考0.5%差所来的影：

- 99%的站延都在132ms以下。
- 0.5%的差以上延的影在正0.66ms。
- 近似算的果会在秒内返回，而“完全正”的果就可能需要几秒，甚至无法返回。

只要的看站的延情况，道我会在意近似果是132.66ms而不是132ms？当然，不是所有的域都能容忍近似果，但于大多数来是没有的。接受近似果更多的是一文化念上的壁而不是商或技上的需要。

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，即字段的`distinct`或者`unique`的数目。可能会SQL形式比熟悉：

```
SELECT COUNT(DISTINCT color)
FROM cars
```

去重是一个很常的操作，可以回答很多基本的：

- 站独立客是多少？
- 了多少汽？
- 月有多少独立用了商品？

我可以用 `cardinality` 度量 定 商 汽 色的数量：

```
GET /cars/transactions/_search
{
  "size" : 0,
  "aggs" : {
    "distinct_colors" : {
      "cardinality" : {
        "field" : "color"
      }
    }
  }
}
```

返回的 果表明已 了三 不同 色的汽 ：

```
...
"aggregations": {
  "distinct_colors": {
    "value": 3
  }
}
...
```

可以 我 的例子 得更有用： 月有多少 色的 被 出？ 了得到 个度量，我 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 衡

正如我 本章 提到的， cardinality 度量是一个近似算法。 它是基于 HyperLogLog++ (HLL) 算法的。 HLL 会先 我 的 入作哈希 算，然后根据哈希 算的 果中的 bits 做概率估算从而得到基数。

我 不需要理解技 (如果 感 趣, 可以 篇 文), 但我 最好 注一下 个算法的 特性 :

- 可配置的精度, 用来控制内存的使用 (更精 = 更多内存)。
- 小的数据集精度是非常高的。
- 我 可以通 配置参数, 来 置去重需要的固定内存使用量。无 数千 是数十 的唯一 , 内存使用 量只与 配置的精 度相 。

要配置精度, 我 必 指定 precision_threshold 参数的 。 个 定 了在何 基数水平下我 希望得到一个近乎精 的 果。参考以下示例 :

```
GET /cars/transactions/_search
{
  "size" : 0,
  "aggs" : {
    "distinct_colors" : {
      "cardinality" : {
        "field" : "color",
        "precision_threshold" : 100 ①
      }
    }
  }
}
```

① precision_threshold 接受 0–40,000 之 的数字, 更大的 是会被当作 40,000 来 理。

示例会 保当字段唯一 在 100 以内 会得到非常准 的 果。尽管算法是无法保 点的, 但如果基数在 以下, 几乎 是 100% 正 的。高于 的基数会 始 省内存而 牺准 度, 同 也会 度量 果 入 差。

于指定的 , HLL 的数据 会大概使用 precision_threshold * 8 字 的内存, 所以就必 在 牺内存和 得 外的准 度 做平衡。

在 用中, 100 的 可以在唯一 百万的情况下 然将 差 持 5% 以内。

速度 化

如果想要 得唯一 的数目, 通常 需要 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 迅速, 原因是 然的。 HyperLogLog 的速度已 很快了, 它只是 的 数据做哈希以及一些位操作。

但如果速度 我 至 重要, 可以做 一 的 化。 因 HLL 只需要字段内容的哈希 , 我 可以在索引 就 先 算好。就能在 跳 哈希 算然后将哈希 从 fielddata 直接加 出来。

NOTE

先 算哈希 只 内容很 或者基数很高的字段有用， 算 些字段的哈希 的消耗在 是无法忽略的。

尽管数 字段的哈希 算是非常快速的，存 它 的原始 通常需要同 （或更少）的内存 空 。 低基数的字符串字段同 用， Elasticsearch 的内部 化能 保 个唯一 只 算一次哈希。

基本上 ， 先 算并不能保 所有的字段都更快，它只 那些具有高基数和/或者内容很 的字符串字段有作用。需要 住的是， 算只是 的将 消耗的 提前 移到索 引 ， 并非没有任何代 ， 区 在于 可以 在 什 时候 做 件事，要 在索引 ， 要 在 。

要想 做，我 需要 数据 加一个新的多 字段。我 先 除索引，再 加一个包括哈希 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" ①  
            }  
          }  
        }  
      }  
    }  
  }  
}
```

```
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" }
```

① 多 字段的 型是 `murmur3`， 是一个哈希函数。

在当我 行聚合，我 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" ①
      }
    }
  }
}

```

① 注意我 指定的是哈希 的多 字段，而不是原始字段。

在 `cardinality` 度量会 取 `"color.hash"` 里的 （先 算的哈希），取代 算原始 的哈希。

个文 省的 是非常少的，但是如果 聚合一 数据， 个字段多花 10 秒的 ，那 在 次 都会 外 加 1 秒，如果我 要在非常大量的数据里面使用 `cardinality`，我 可以 衡使用 算的意，是否需要提前 算 hash，从而在 得更好的性能，做一些性能 来 算哈希是否 用于 的 用 景。。

百分位 算

Elasticsearch 提供的 外一个近似度量就是 `percentiles` 百分位数度量。 百分位数展 某以具体百分比下 察到的数 。例如，第95个百分位上的数 ，是高于 95% 的数据 和。

百分位数通常用来 出 常。在（ 学）的正 分布下，第 0.13 和 第 99.87 的百分位数代表与均 距 三倍 准差的 。任何 于三倍 准差之外的数据通常被 是不 常的，因 它与平均 相差太大。

更具体的，假 我 正 行一个 大的 站，一个很重要的工作是保 用 求能得到快速 ，因此我 就需要 控 站的延 来判断 是否能保 良好的用 体 。

在此 景下，一个常用的度量方法就是平均 延 。 但 并不是一个好的 （尽管很常用），因 平均数通常会 藏那些 常 ， 中位数有着同 的 。 我 可以 最大 ，但 个度量会 而易 的被 个 常 破坏。

在 `Average request latency over time` 看 。如果我 依 如平均 或中位数 的 度量，就会得到像 一幅 `Average request latency over time` 。

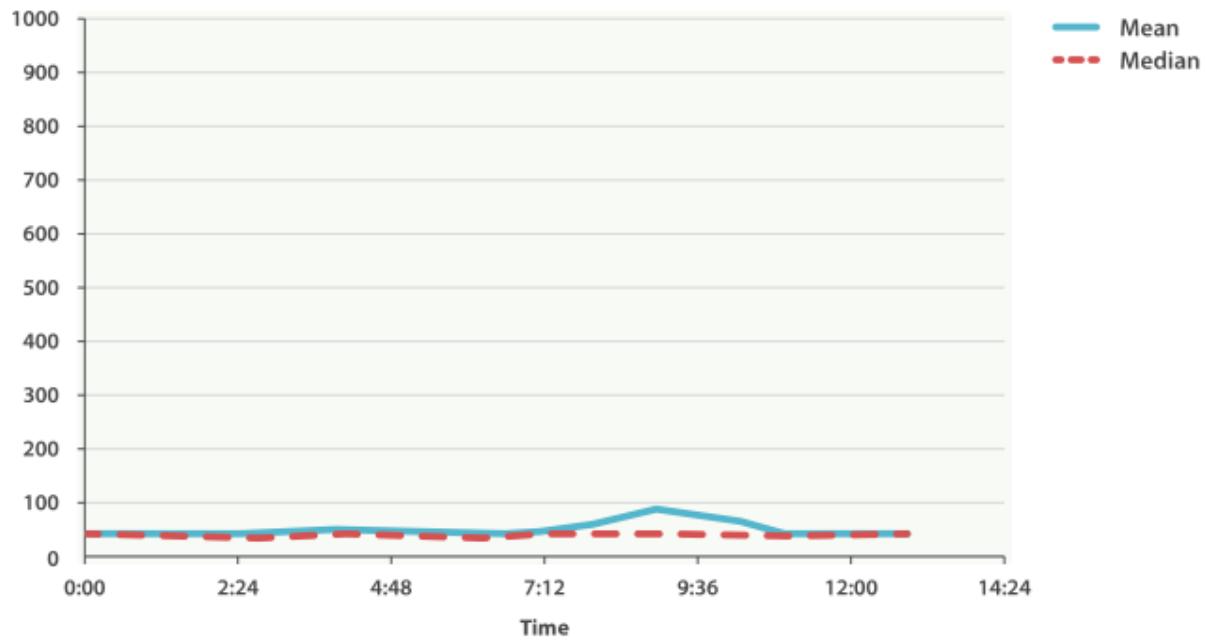


Figure 40. Average request latency over time

一切正常。 上有 微的波 , 但没有什 得 注的。 但如果我 加 99 百分位数 (个 代表最慢的 1% 的延) , 我 看到了完全不同的一幅画面, 如 [Average request latency with 99th percentile over time](#) 。

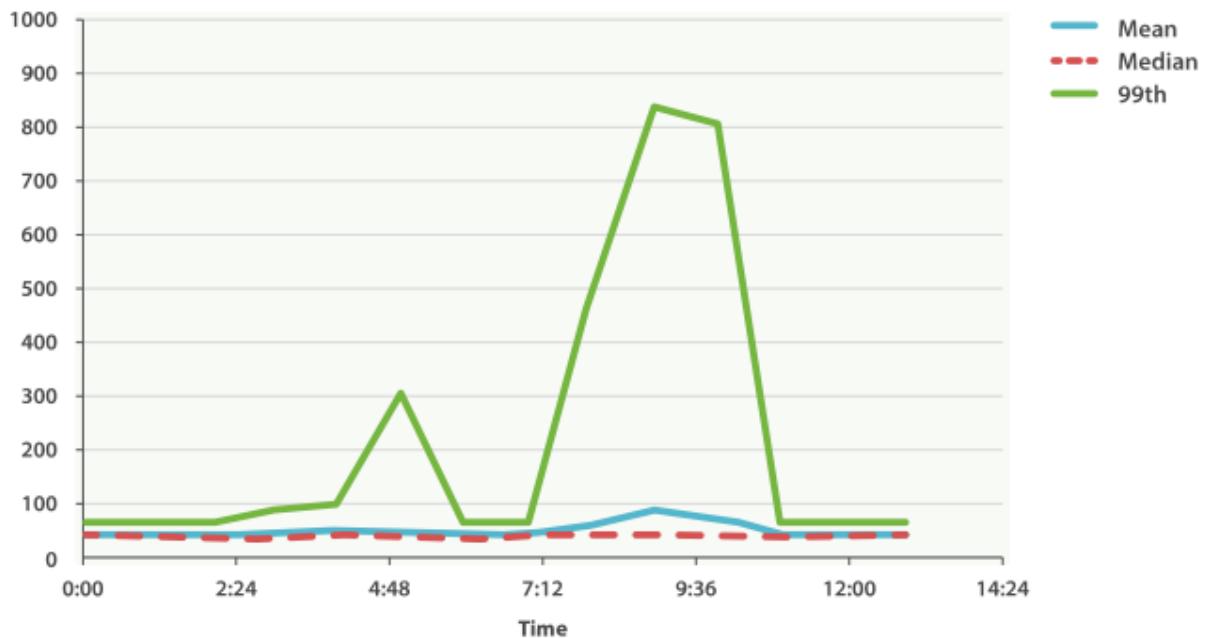


Figure 41. Average request latency with 99th percentile over time

令人吃 ! 在上午九点半 , 均 只有 75ms。如果作 一个系 管理 , 我 都不会看他第二眼。 一切正常 ! 但 99 百分位告 我 有 1% 的用 到的延 超 850ms, 是 外一幅 景。 在上午4 点48 也有一个小波 , 甚至无法从平均 和中位数曲 上 察到。

只是百分位的一个用景，百分位可以被用来快速用肉眼察数据的分布，是否有数据斜或双峰甚至更多。

百分位度量

我加一个新的数据集（汽的数据不太用于百分位）。我要索引一系列站延数据然后行一些百分位操作行看：

```
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：延、数据中心的区域以及。我数据全集行百分位操作以得数据分布情况的直感受：

```

GET /website/logs/_search
{
  "size" : 0,
  "aggs" : {
    "load_times" : {
      "percentiles" : {
        "field" : "latency" ①
      }
    },
    "avg_load_time" : {
      "avg" : {
        "field" : "latency" ②
      }
    }
  }
}

```

① percentiles 度量被用到 latency 延时字段。

② 除了比，我相同字段使用 avg 度量。

情况下，percentiles 度量会返回一定百分位数：[1, 5, 25, 50, 75, 95, 99]。它表示了人感兴趣的常用百分位数，低端的百分位数在低的，其他的一些位于中部。在返回的中，我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": 489.3499999999985,
      "99.0": 596.2700000000002
    }
  },
  "avg_load_time": {
    "value": 199.5833333333334
  }
}

```

所以虽然延时的分布很广，我看下它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" ①
      },
      "aggs" : {
        "load_times" : {
          "percentiles" : { ②
            "field" : "latency",
            "percents" : [50, 95.0, 99.0] ③
          }
        },
        "load_avg" : {
          "avg" : {
            "field" : "latency"
          }
        }
      }
    }
  }
}

```

① 首先根据区域我 将延 分到不同的桶中。

② 再 算 个区域的百分位数 。

③ percents 参数接受了我 想返回的一 百分位数，因 我 只 的延 感 趣。

在 果中，我 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 都接近均 。

与之形成 比的是，欧洲区域（EU）在 50 和 99 百分位有 大区分。 在， 然可以 是欧洲区域（EU）拉低了延 的 信息，我 知道欧洲区域的 50% 延 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

里有 外一个 密相 的度量叫 `percentile_ranks` 。 `percentiles` 度量告 我 落在某个百分比以下的所有文 的最小 。例如，如果 50 百分位是 119ms, 那 有 50% 的文 数 都不超 119ms。 `percentile_ranks` 告 我 某个具体 属于 个百分位。119ms 的 `percentile_ranks` 是在 50 百分位。 基本是个双向 系，例如：

- 50 百分位是 119ms。
- 119ms 百分位等 是 50 百分位。

所以假 我 站必 持的服 等 (SLA) 是 低于 210ms。然后， 个玩笑，我 老板警告我 如果 超 800ms 会把我 除。可以理解的是，我 希望知道有多少百分比的

求可以 足 SLA 的要求（并期望至少在 800ms 以下！）。

了做到 点，我 可以 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] ①
          }
        }
      }
    }
  }
}
```

① `percentile_ranks` 度量接受一 我 希望分 的数 。

在聚合 行后，我 能得到 个 :

```

"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
          }
        }
      }
    ]
  }
}

```

告 我 三点重要的信息：

- 在欧洲 (EU) , 210ms 的百分位等 是 31.94% 。
- 在美国 (US) , 210ms 的百分位等 是 100% 。
- 在欧洲 (EU) 和美国 (US) , 800ms 的百分位等 是 100% 。

通俗的 , 在欧洲区域 (EU) 只有 32% 的 足服 等 (SLA) , 而美国区域 (US) 始足服 等 的。但幸 的是, 个区域所有 都在 800ms 以下, 所以我 不会被炒 (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 , 如果我 某个具体数更 心, 使用它会更方便。

学会 衡

和基数一 , 算百分位需要一个近似算法。朴素的 会 一个所有 的有序列表, 但当我有几十 数据分布在几十个 点 , 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 里面提出的)。与 HyperLogLog 一 , 不需要理解完整的技 , 但有必要了解算法的特性 :

- 百分位的准确度与百分位的端程度相，也就是 1 或 99 的百分位要比 50 百分位要准。只是数据内部机制的一特性，但是一个好的特性，因多数人只关心端的百分位。
- 于数集合小的情况下，百分位非常准。如果数据集足够小，百分位可能 100% 精确。
- 随着桶里数的增加，算法会开始百分位行估算。它能有效在准确度和内存省之间做出平衡。不准的程度比以前，因为它基于聚合数据的分布以及数据量的大小。

与 `cardinality` 似，我可以通过修改参数 `compression` 来控制内存与准确度之比。

`TDigest` 算法用点近似算百分比：点越多，准确度越高（同内存消耗也越大），都与数据量成正比。`compression` 参数限制点的最大数目 $20 * \text{compression}$ 。

因此，通过增加比，可以消耗更多内存来提高百分位数准确性。更大的比会使算法运行更慢，因为底的数据存储也会增加，导致操作的代价更高。这个比是 **100**。

一个点大使用 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），置会生成一个大小为 64KB 的 `TDigest`。在使用中，数据会更随机，所以 `TDigest` 使用的内存会更少。

通用聚合常指

`significant_terms` (`SigTerms`) 聚合与其他聚合都不相同。目前止我看到的所有聚合本质上都是数学计算。将不同一些造相互合在一起，我可以建立的聚合以及数据表。

`significant_terms` 有着不同的工作机制。有些人来说，它甚至看起来有点像机器学习。`significant_terms` 聚合可以在数据集中找到一些常指。

如何解一些不常指的行？一些常的数据指通常比我估出的次要更繁，这些上的常指通常象征着数据里的某些有趣信息。

例如，假如我和跟踪信用欺，客户打来抱怨他信用输出异常交易，它的已被盗用。些交易信息只是更重的症状。在最近的某些地区，一些商家有意的盗取客户的信用信息，或者它自己的信息无意中也被盗取。

我的任务是找到危害的共同点，如果我有 100 个客户抱怨交易异常，他们很有可能都属于同一个商家，而商家有可能就是罪魁首。

当然，里面有一些特例。例如，很多客户在它近期交易史中会有很大的商如，我可以将排除在外，然而，在最近一些有信用的商家里面也有。

是一个普通的共同商的例子。个人都共享一个商家，无论有没有遭受危害。我并不感兴趣。

相反，我有一些很小商比如街角的一家店，它属于普通但不常的情况，只有一个客户有交易。我同样可以将这些商排除，因为所有受到危害的信用都没有与这些商发生交易，我可以肯定它不是安全漏洞的任方。

我真正想要的是不普通的共同商。所有受到危害的信用都与它发生交易，但是在未受危害的背景噪声下，它并不明显。这些商属于常，它比输出的率要高。一些不普通的共同商很有可能就是需要的。

`significant_terms` 聚合就是做些事情。它分析的数据并通过比正常数据到可能有常

次的指 。

暴露的 常指 代表什 依 的 数据。 于信用 数据，我 可能会想 出信用 欺 。 于 商数据，我 可能会想 出来被 的人口信息，从而 行更高效的市 推广。 如果我 正在分析日志，我 可能会 一个服 器会 出比它本 出的更多 常。 `significant_terms` 的 用 不止 些。

significant_terms 演示

因 `significant_terms` 聚合 是通 分析 信息来工作的， 需要 数据 置一个 它 更有效。也就是 无法通 只索引少量示例数据来展示它。

正因如此，我 准 了一个大 8000 个文 的数据集，并将它的快照保存在一个公共演示 中。可以通 以下 在集群中 原 些数据：

1. 在 `elasticsearch.yml` 配置文件中 加以下配置，以便将演示 加入到白名 中：

```
repositories.url.allowed_urls: ["http://download.elastic.co/*"]
```

2. 重 Elasticsearch。

3. 行以下快照命令。（更多使用快照的信息，参 集群（Backing Up Your Cluster））。

```
PUT /_snapshot/sigterms ①
{
  "type": "url",
  "settings": {
    "url": "http://download.elastic.co/definitiveguide/sigterms_demo/"
  }
}

GET /_snapshot/sigterms/_all ②

POST /_snapshot/sigterms/snapshot/_restore ③

GET /mlmovies,mlratings/_recovery ④
```

① 注 一个新的只 地址 ， 并指向演示快照。

② (可) 内 于快照的 信息。

③ 始 原 程。会在集群中 建 个索引： `mlmovies` 和 `mlratings` 。

④ (可) 使用 Recovery API 控 原 程。

NOTE 数据集有 50 MB 会需要一些 下 。

在本演示中，会看看 MovieLens 里面用 影的 分。在 MovieLens 里，用 可以推 影并 分， 其他用 也可以 到新的 影。 了演示，会基于 入的 影采用 `significant_terms` 影 行推 。

我看看示例中的数据，感受一下要理的内容。本数据集有 个索引，`mlmovies` 和 `mlratings`。首先看 `mlmovies`：

```
GET mlmovies/_search ①

{
  "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)"
        }
      },
      ...
    ]
  }
}
```

① 行一个不 条件的搜索，以便能看到一 随机演示文。

`mlmovies` 里的 个文 表示一个 影，数据有 个重要字段：影ID `_id` 和 影名 `title`。可以忽略 `offset` 和 `bytes`。它 是从原始 CSV 文件抽取数据的 程中 生的中 属性。数据集中有 10, 681 部影片。

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  
                }  
            },  
            ...  
        ]  
    },  
}
```

里可以看到 个用 的推 信息。 个文 表示一个用 ，用 ID 字段 `user` 来表示， `movie` 字段 一个用 看和推 的影片列表。

基于流行程度推 (Recommending Based on Popularity)

可以采取的首个策略就是基于流行程度向用 推 影片。 于某部影片， 到所有推 它的用 ，然后将他 的推 行聚合并 得推 中最流行的五部。

我 可以很容易的通 一个 `terms` 聚合 以及一些 来表示它，看看 *Talladega Nights* (塔拉 加之夜) 部影片，它是 Will Ferrell 主演的一部 于全国 汽 (NASCAR racing) 的喜 。 在理想情况下，我 的推 到 似 格的喜 (很有可能也是 Will Ferrell 主演的)。

首先需要 到影片 *Talladega Nights* 的 ID :

```

GET mlmovies/_search
{
  "query": {
    "match": {
      "title": "Talladega Nights"
    }
  }
}

...
"hits": [
  {
    "_index": "mlmovies",
    "_type": "mlmovie",
    "_id": "46970", ①
    "_score": 3.658795,
    "_source": {
      "offset": 9575,
      "bytes": 74,
      "title": "Talladega Nights: The Ballad of Ricky Bobby (2006)"
    }
  },
  ...

```

① *Talladega Nights* 的 ID 是 46970。

有了 ID，可以分，再用 terms 聚合从喜 *Talladega Nights* 的用中到最流行的影片：

```

GET mlratings/_search
{
  "size" : 0, ①
  "query": {
    "filtered": {
      "filter": {
        "term": {
          "movie": 46970 ②
        }
      }
    }
  },
  "aggs": {
    "most_popular": {
      "terms": {
        "field": "movie", ③
        "size": 6
      }
    }
  }
}

```

① 次 `mlratings`, 将 果内容 大小 置 0 因 我 只 聚合的 果感 趣。

② 影片 *Talladega Nights* 的 ID 使用 器。

③ 最后, 使用 `terms` 桶 到最流行的影片。

在 `mlratings` 索引下搜索, 然后 影片 *Talladega Nights* 的 ID 使用 器。由于聚合是行操作的, 它可以有效的 聚合 果从而得到那些只推 *Talladega Nights* 的用。

最后, 行 `terms` 聚合得到最流行的影片。求排名最前的六个 果, 因 *Talladega Nights* 本身很有可能就是其中一个 果(并不想重 推 它)。

返回 果就像 :

```
{  
...  
  "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  
        }  
      ]  
    }  
  }  
...  
}
```

通一个的，将得到的结果成原始影片名：

```

GET mlmovies/_search
{
  "query": {
    "filtered": {
      "filter": {
        "ids": {
          "values": [2571,318,296,2959,260]
        }
      }
    }
  }
}

```

最后得到以下列表：

1. Matrix, The (客帝国)
2. Shawshank Redemption (肖申克的救)
3. Pulp Fiction (低俗小)
4. Fight Club (搏 部)
5. Star Wars Episode IV: A New Hope (星球大 IV : 曙光乍)

好 , 肯定不是一个好的列表 ! 我喜 所有 些影片。但 是 : 几乎 个人 都喜 它 。
 些影片本来就受大 迎, 也就是 它 出 在 个人的推 中都会受 迎。
 是一个流行影片的推 列表, 而不是和影片 *Talladega Nights* 相 的推 。

可以通 再次 行聚合 松 , 而不需要 影片 *Talladega Nights* 行
 。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)
2. Silence of the Lambs, The (的 羊)
3. Pulp Fiction (低俗小)

4. Forrest Gump (阿甘正传)

5. Star Wars Episode IV: A New Hope (星球大战 IV: 曙光乍现)

然，只是最流行的影片是不能足以建一个良好而又具能力的推荐系统。

基于统计的推荐 (Recommending Based on Statistics)

在场景已定好，使用 `significant_terms`。`significant_terms` 会分析喜剧影片 *Talladega Nights* 的用（前端用），并且判定最流行的影片。然后会用（后端用）造一个流行影片列表，最后将两者行比。

常就是与背景相比在前景特征中度展现的那些影片。理论上，它是一喜剧，因喜 Will Ferrell 喜的人这些影片的分会比一般人高。

我一下：

```
GET mlratings/_search
{
  "size": 0,
  "query": {
    "filtered": {
      "filter": {
        "term": {
          "movie": 46970
        }
      }
    }
  },
  "aggs": {
    "most_sig": {
      "significant_terms": { ①
        "field": "movie",
        "size": 6
      }
    }
  }
}
```

① 置几乎一模一样，只是用 `significant_terms` 替代了 `terms`。

正如所，也几乎是一样的。输出喜剧影片 *Talladega Nights* 的用，他成了前景特征用。情况下，`significant_terms` 会使用整个索引里的数据作背景，所以不需要特的理。

与 `terms` 似，果返回了一桶，不有更多的元数据信息：

```
...
"aggregations": {
  "most_sig": {
    "doc_count": 271, ①
    "buckets": [
      {
        "key": 46970,
        "key_as_string": "46970",
        "doc_count": 271,
        "score": 256.549815498155,
        "bg_count": 271
      },
      {
        "key": 52245, ②
        "key_as_string": "52245",
        "doc_count": 59, ③
        "score": 17.66462367106966,
        "bg_count": 185 ④
      },
      {
        "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
      }
    ]
  ...
}
```

- ① `doc_count` 展示了前景特征里文的数量。
- ② 一个桶里面列出了聚合的（例如，影片的ID）。
- ③ 桶内文的数量 `doc_count`。
- ④ 背景文的数量，表示在整个背景里出现的度。

可以看到，得的第一个桶是 *Talladega Nights*。它可以在所有 271 个文本中找到，并不意外。我看下一个桶：`52245`。

一个 ID 影片 *Blades of Glory*（*誉之刃*），它是一部关于男子学滑稽的喜剧，也是由 Will Ferrell 主演。可以看到喜剧 *Talladega Nights* 的用法它的推荐是 59 次。这也意味着 21% 的前景特征用推荐了影片 *Blades of Glory* ($59 / 271 = 0.2177$)。

形成比的是，*Blades of Glory* 在整个数据集合中被推荐了 185 次，只占 0.26% ($185 / 69796 = 0.00265$)。因此 *Blades of Glory* 是一个非常：它在喜剧 *Talladega Nights* 的用法中是著名的共性（注：uncommonly common）。就到了一个好的推荐！

如果看完整的列表，它都是好的喜剧推荐（其中很多也是由 Will Ferrell 主演）：

1. *Blades of Glory*（*誉之刃*）
2. *Anchorman: The Legend of Ron Burgundy*（王牌播音）
3. *Semi-Pro*（半职业手）
4. *Knocked Up*（一夜大肚）
5. *40-Year-Old Virgin, The*（四十岁的老男）

只是 `significant_terms` 它大的一个示例，一旦开始使用 `significant_terms`，可能遇到的情况，我不想最流行的，而想要著名的共性（注：uncommonly common）。一个聚合可以显示出一些数据里出人意料的。

Doc Values and Fielddata

Doc Values

聚合使用一个叫 `doc values` 的数据（在 `Doc Values` 介里）。`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某些操作是高效的。倒排索引的在于包含某个的文本，而由于从外一个方向的相反操作并不高效，即：定位一些是否存在一个文本里，聚合需要次的模式。

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 想要 得所有包含 brown 的文 的 的完整列表，我 会 建如下：

```
GET /my_index/_search
{
  "query": {
    "match": {
      "body": "brown"
    }
  },
  "aggs": {
    "popular_terms": {
      "terms": {
        "field": "body"
      }
    }
  }
}
```

部分 又高效。倒排索引是根据 来排序的，所以我 首先在 列表中 到 brown ， 然后 描所有列， 到包含 brown 的文 。我 可以快速看到 Doc_1 和 Doc_2 包含 brown 个 token。

然后， 于聚合部分，我 需要 到 Doc_1 和 Doc_2 里所有唯一的 。用倒排索引做 件事情代 很高： 我 会迭代索引里的 个 并收集 Doc_1 和 Doc_2 列里面 token。 很慢而且 以 展：随着 和文 的数量 加， 行 也会 加。

Doc values 通 置 者 的 系来解决 个 。倒排索引将 映射到包含它 的文 ， doc values 将文 映射到它 包含的 ！

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 置之后，想要收集到 `Doc_1` 和 `Doc_2` 的唯一 token 会非常容易。得 个文 行，取所有的 ，然后求 个集合的并集。

因此，搜索和聚合是相互 密 的。搜索使用倒排索引 文 ，聚合操作收集和聚合 doc values 里的数据。

NOTE

Doc values 不 可以用于聚合。任何需要 某个文 包含的 的操作都必 使用它。除了聚合， 包括排序， 字段 的脚本，父子 系 理（参 父-子 系文 ）。

深入文

在上一 我 就 文 (doc values) 是 “更快、更高效并且内存友好” 。听起来好像是不 的 ，不 回来文 到底是如何工作的 ？

文 是在索引 与倒排索引同 生的。也就是 文 是按段来 生的并且是不可 的，正如用于搜索的倒排索引一 。同 ，和倒排索引一 ，文 也序列化到磁 。些 于性能和伸 性很重要。

通 序列化一个持久化的数据 到磁 ，我 可以依 于操作系 的 存来管理内存，而不是在 JVM 堆 里 留数据。当 “工作集 (working set)” 数据要小于系 可用内存的情况下，操作系 会自然的将文 留在内存， 将会 来和直接使用 JVM 堆 数据 相同的性能。

不 ，如果 的工作集 大于可用内存，操作系 会 始根据需要 文 行分 / 。会 著慢于 内存 留的数据 ，当然，它也 有使用 大于服 器内存容量的伸 性的好 。如果 些数据 是 粹的存 于 JVM 堆内存，那 唯一的 只能是随着内存溢出 (OutOfMemory) 而崩 (或是一个分 模式，正如操作系 的那)。

因 文 不是由 JVM 来管理，所以 Elasticsearch 服 器可以配置一个很小的 JVM 堆 。会 操作系 来更多的内存来做 存。同 也 来一个好 就是 JVM 的 回收器工作在一个很小的堆 ， 果就是更快更高效的回收周期。

NOTE

上，我 会建 分配机器内存的 50% 来 JVM 堆 。随着文 的引入， 个建 始不再 用。在 64gb 内存的机器上，也 可以考 堆 分配 4-16gb 的内存，而不是之前建 的 32gb。

有 更 的 ， 看 [堆内存:大小和交](#) 。

列式存 的

从广 来 ，文 本 上是一个序列化的 列式存 。正如我 上一 所 的，列式存 擅 某些操作，因 些数据的存 天然 合 些 。

而且，他也同擅数据，特别是数字。于省磁空和快速很重要。代理CPU的理速度要比磁快几个数量（尽管即将到来的NVMe器正在迅速缩小差距）。这意味着少必从磁取的数据量是有益的，尽管需要外的CPU算来行解。

要了解它如何助数据，来看一数字型的文：

Doc Terms

Doc_1	100
Doc_2	1000
Doc_3	1500
Doc_4	1200
Doc_5	300
Doc_6	1900
Doc_7	4200

按列布局意味着我有一个的数据：[100, 1000, 1500, 1200, 300, 1900, 4200]。因我已知道他都是数字（而不是像文或行中看到的集合），所以我可以使用一的偏移来将他排列。

而且，的数字有很多技巧。会注意到里个数字都是100的倍数，文会一个段里面的所有数，并使用一个最大公数，方便做一的数据。

如果我保存100作此段的除数，我可以个数字都除以100，然后得到：[1, 10, 15, 12, 3, 19, 42]。在些数字小了，只需要很少的位就可以存下，也少了磁存放的大小。

文正是使用了像的一些技巧。它会按依次以下模式：

1. 如果所有的数各不相同（或失），置一个并一些
2. 如果些小于256，将使用一个的表
3. 如果些大于256，是否存在一个最大公数
4. 如果没有存在最大公数，从最小的数始，一算偏移量行

会些模式不是的通用的方式，比如DEFLATE或是LZ4。因列式存的是格且良好定的，我可以通使用的模式来比通用算法（如LZ4）更高的效果。

NOTE

也想“好，貌似数字很好，不知道字符串？”通借助序表（ordinal table），字符型也是似行的。字符型是去重之后存放到序表的，通分配一个ID，然后些ID和数型的文一使用。也就是，字符型和数型一有相同的特性。

序表本身也有很多技巧，比如固定度、或是前字符等等。

禁用文

文所有字段用，除了分析字符型字段。也就是所有的数字、地理坐、日期、IP和不分析（not_analyzed）字符型。

分析字符型不使用全文。分析流程会生很多新的token，全文不能高效的工作。我将在[聚合与分析](#)如何使用分析字符型来做聚合。

因为全文用，可以对数据集里面的大多数组字段行聚合和排序操作。但是如果知道永远也不会某些字段行聚合、排序或是使用脚本操作？

尽管，但当些情况出现，是希望有办法来特定的字段禁用全文。回省磁空（因为全文再也没有序列化到磁盘），也能提升些索引速度（因为不需要生成全文）。

要禁用全文，在字段的映射(mapping)置`doc_values: false`即可。例如，里我建了一个新的索引，字段`"session_id"`禁用了全文：

```
PUT my_index
{
  "mappings": {
    "my_type": {
      "properties": {
        "session_id": {
          "type": "string",
          "index": "not_analyzed",
          "doc_values": false ①
        }
      }
    }
  }
}
```

①通过置`doc_values: false`，个字段将不能被用于聚合、排序以及脚本操作

反过来也是可以行配置的：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, ①
          "index": "no" ②
        }
      }
    }
  }
}
```

①全文被用来允聚合

②索引被禁用了，字段不能被 /搜索

通 置 `doc_values: true` 和 `index: no`，我 得到一个只能被用于聚合/排序/脚本的字段。无可否，是一个非常 的需求，但有 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`， 那来了，分析是 影 聚合的 ？

答案是影 “很多”，有 个原因：分析影 聚合中使用的 `tokens`， 并且 `doc values` 不能使用于分析字符串。

我 解决第一个 ！：分析 `tokens` 的 生如何影 聚合。首先索引一些代表美国各个州的文：

```
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 希望 建一个数据集里各个州的唯一列表，并且 数。 ， 我 使用 `terms` 桶：

```
GET /agg_analysis/data/_search
{
  "size" : 0,
  "aggs" : {
    "states" : {
      "terms" : {
        "field" : "state"
      }
    }
  }
}
```

得到 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 儿， 完全不是我 想要的！没有 州名 数，聚合 算了 个 的数目。背后的原因很 : 聚合是 基于倒排索引 建的，倒排索引是 后置分析 (post-analysis) 的。

当我 把 些文 加入到 Elasticsearch 中 ，字符串 "New York" 被分析/分析成 ["new", "york"] 。 些 独的 tokens，都被用来填充聚合 数，所以我 最 看到 new 的数量而不是 New York 。

然不是我 想要的行 ，但幸 的是很容易修正它。

我 需要 state 定 multfield 并且 置成 not_analyzed 。 可以防止 New York 被分析，也意味着在聚合 程中它会以 个 token 的形式存在。 我 完整的 程，但 次指定一个 raw multfield :

```

DELETE /agg_analysis/
PUT /agg_analysis
{
  "mappings": {
    "data": {
      "properties": {
        "state" : {
          "type": "string",
          "fields": {
            "raw" : {
              "type": "string",
              "index": "not_analyzed"①
            }
          }
        }
      }
    }
  }
}

```

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" ②
      }
    }
  }
}

```

① 次我 式映射 state 字段并包括一个 not_analyzed 字段。

② 聚合 state.raw 字段而不是 state。

在 行聚合，我 得到了合理的 果：

```
{
...
"aggregations": {
  "states": {
    "buckets": [
      {
        "key": "New York",
        "doc_count": 3
      },
      {
        "key": "New Jersey",
        "doc_count": 1
      },
      {
        "key": "New Mexico",
        "doc_count": 1
      }
    ]
  }
}
}
```

在 中， 的 很容易被察 ，我 的聚合会返回一些奇怪的桶，我 会 住分析的 。之，很少有在聚合中使用分析字段的 例。当我 疑惑 ，只要 加一个 multfield 就能有 。

分析字符串和 Fielddata (Analyzed strings and Fielddata)

当第一个 及如何聚合数据并 示 用 ，第二个 主要是技 和幕后。

Doc values 不支持 **analyzed** 字符串字段，因 它 不能很有效的表示多 字符串。 Doc values 最有效的是，当 个文 都有一个或几个 tokens ， 但不是无数的，分析字符串（想象一个 PDF ，可能有几兆字 并有数以千 的独特 tokens）。

出于 个原因， doc values 不生成分析的字符串，然而， 些字段 然可以使用聚合，那 可能 ？

答案是一 被称 **fielddata** 的数据 。与 doc values 不同， fielddata 建和管理 100% 在内存中，常 于 JVM 内存堆。 意味着它本 上是不可 展的，有很多 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 。

NOTE

从 史上看， fielddata 是所有字段的 置。但是 Elasticsearch 已 移到 doc values 以 少 OOM 的几率。分析的字符串是 然使用 fielddata 的最后一 地。最 目 是建立一个序列化的数据 似于 doc values ，可以 理高 度的分析字符串，逐 淘汰 fielddata。

高基数内存的影 (High-Cardinality Memory Implications)

避免分析字段的 外一个原因就是：高基数字段在加 到 fielddata 会消耗大量内存。 分析的 程会 常（尽管不 是 ）生成大量的 token， 些 token 大多都是唯一的。 会 加字段的整体基数并且 来更大的内存 力。

有些 型的分析 于内存来 度 不友好，想想 n-gram 的分析 程， New York 会被 n-gram 分析成以下 token：

- ne
- ew
- w{nbsp}
- {nbsp}y
- yo
- or
- rk

可以想象 n-gram 的 程是如何生成大量唯一 token 的，特 是在分析成段文本的 候。当 些数据加 到内存中，会 而易 的将我 堆空 消耗殆尽。

因此，在聚合字符串字段之前， 估情况：

- 是一个 not_analyzed 字段 ？如果是，可以通 doc values 省内存。
- 否， 是一个 analyzed 字段，它将使用 fielddata 并加 到内存中。 个字段因 ngrams 有一个非常大的基数？如果是， 于内存来 度不友好。

限制内存使用

一旦分析字符串被加 到 fielddata ，他 会一直在那里，直到被 逐（或者 点崩 ）。由于 个原因，留意内存的使用情况，了解它是如何以及何 加 的， 限制 集群的影 是很重要的。

Fielddata 是 延 加 。如果 从来没有聚合一个分析字符串，就不会加 fielddata 到内存中。此外， fielddata 是基于字段加 的， 意味着只有很活 地使用字段才会 加 fielddata 的 担。

然而， 里有一个令人 的地方。假 的 是高度 性和只返回命中的 100 个 果。大多数人 fielddata 只加 100 个文 。

情况是， fielddata 会加 索引中（ 特定字段的） 所有的 文 ，而不管 的特 性。 是：如果 会 文 X、Y 和 Z，那很有可能会在下一个 中 其他文 。

与 doc values 不同， fielddata 不会在索引 建。相反，它是在 行 ， 填充。 可能是一个比 的操作，可能需要一些 。 将所有的信息一次加 ，再将其 持在内存中的方式要比反 只加 一个 fielddata 的部分代 要低。

JVM 堆 是有限 源的， 被合理利用。 限制 fielddata 堆使用的影 有多套机制， 些限制方式非常重要，因 堆 的乱用会 致 点不 定（感 慢的 回收机制），甚至 致 点宕 机（通常伴随 OutOfMemory 常）。

堆大小 (Choosing a Heap Size)

在 置 Elasticsearch 堆大小 需要通 `$ES_HEAP_SIZE` 境 量 用 个 :

不要超 可用 RAM 的 50%

Lucene 能很好利用文件系 的 存, 它是通 系 内核管理的。如果没有足 的文件系 存空 , 性能会受到影 。 此外, 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 32 GB

如果堆大小小于 32 GB, JVM 可以利用指 , 可以大大降低内存的使用 : 个指 4 字 而不是 8 字 。

更 和更完整的堆大小 , 参 [堆内存:大小和交](#)

Fielddata的大小

`indices.fielddata.cache.size` 控制 fielddata 分配的堆空 大小。 当 起一个 , 分析字符串的聚合将会被加 到 fielddata, 如果 些字符串之前没有被加 。如果 果中 fielddata 大小超 了指定 大小 , 其他的 将会被回收从而 得空 。

情况下, 置都是 *unbounded*, Elasticsearch 永 都不会从 fielddata 中回收数据。

个 置是刻意 的: fielddata 不是 存。它是 留内存里的数据 , 必 可以快速 行 , 而且 建它的代 十分高昂。如果 个 求都重 数据, 性能会十分糟 。

一个有界的大小会 制数据 回收数据。我 会看何 置 个 , 但 首先 以下警告 :

个 置是一个安全 土, 而非内存不足的解决方案。

WARNING 如果没有足 空 可以将 fielddata 保留在内存中, Elasticsearch 就会 刻从磁 重 数据, 并回收其他数据以 得更多空 。内存的回收机制会 致重度磁 I/O, 并 且在内存中生成很多 , 些 必 在 些 时候被回收掉。

想我 正在 日志 行索引, 天使用一个新的索引。通常我 只 去一 天的数据感 趣, 尽管我 会保留老的索引, 但我 很少需要 它 。不 如果采用 置, 旧索引的 fielddata 永 不会从 存中回收 ! fielddata 会保持 直到 fielddata 生断熔 (参 [断路器](#)), 我 就无法 入更多的 fielddata。

个 时候, 我 被困在了死胡同。但我 然可以 旧索引中的 fielddata, 也无法加 任何新的 。相反, 我 回收旧的数据, 并 新 得更多空 。

了防止 生 的事情, 可以通 在 `config/elasticsearch.yml` 文件中 加配置 fielddata 置一个上限 :

`indices.fielddata.cache.size: 20% ①`

① 可以 置堆大小的百分比, 也可以是某个 , 例如: `5gb` 。

有了一个置，最久未使用（LRU）的 fielddata 会被回收。新数据出空。

可能在文有外一个置：`indices.fielddata.cache.expire`。

一个置永都不会被使用！它很有可能在不久的将来被用。

一个置要求 Elasticsearch 回收那些期的 fielddata，不管些有没有被用到。

WARNING

性能是件很糟的事情。回收会有消耗性能，它刻意的安排回收方式，而没能得到任何回。

没有理由使用一个置：我不能从理上假一个有用的情形。目前，它的存在只是了向前兼容。我只在很有以前提到一个置，但不幸的是上各文章都将其作一性能的小来推。

它不是。永远不要使用！

控 fielddata (Monitoring fielddata)

无是仔控 fielddata 的内存使用情况，是看有无数据被回收都十分重要。高的回收数可以示重的源以及性能不佳的原因。

Fielddata 的使用可以被控：

- 按索引使用 `indices-stats API`：

```
GET /_stats/fielddata?fields=*
```

- 按点使用 `nodes-stats API`：

```
GET /_nodes/stats/indices/fielddata?fields=*
```

- 按索引点：

```
GET /_nodes/stats/indices/fielddata?level=indices&fields=*
```

使用置 `?fields=*`，可以将内存使用分配到个字段。

断路器

机敏的者可能已 fielddata 大小置的一个。fielddata 大小是在数据加之后的。如果一个加比可用内存更多的信息到 fielddata 中会生什？答案很丑：我会到 `OutOfMemoryException`。

Elasticsearch 包括一个 `fielddata` 断熔器，个就是了理上述情况。断熔器通内部（字段的型、基数、大小等等）来估算一个需要的内存。它然后要求加的 fielddata 是否会致 fielddata 的量超堆的配置比例。

如果估算的大小超出限制，就会触断路器，会被中止并返回常。都生在数据加之前，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，它都能保内存不会超出限制：

`indices.breaker.fielddata.limit`

fielddata 断路器置堆的 60% 作 fielddata 大小的上限。

`indices.breaker.request.limit`

request 断路器估算需要完成其他求部分的大小，例如建一个聚合桶，限制是堆内存的 40%。

`indices.breaker.total.limit`

total 拢合 request 和 fielddata 断路器保者合起来不会使用超堆内存的 70%。

断路器的限制可以在文件 `config/elasticsearch.yml` 中指定，可以更新一个正在行的集群：

```
PUT /_cluster/settings
{
  "persistent" : {
    "indices.breaker.fielddata.limit" : "40%" ①
  }
}
```

① 个限制是按内存的百分比置的。

最好断路器置一个相保守点的。住 fielddata 需要与 request 断路器共享堆内存、索引冲内存和器存。Lucene 的数据被用来造索引，以及各其他的数据。正因如此，它非常保守，只有 60%。于的置可能会引起潜在的堆溢出 (OOM) 常，会使整个点宕掉。

一方面，度保守的只会返回常，用程序可以常做相理。常比服器崩要好。些常也能促我行重新估：什一个需要超堆内存的 60% 之多？

TIP

在 [Fielddata的大小](#) 中，我提于 fielddata 的大小加一个限制，从而保旧的无用 fielddata 被回收的方法。`indices.fielddata.cache.size` 和 `indices.breaker.fielddata.limit` 之的系非常重要。如果断路器的限制低于存大小，没有数据会被回收。了能正常工作，断路器的限制必要比存大小要高。

得注意的是：断路器是根据堆内存大小估算大小的，而非根据堆内存的使用情况。是由于各技原因造成的（例如，堆可能看上去是的但上可能只是在等待回收，使我以行合理的估算）。但作端用，意味着置需要保守，因它是根据堆内存必要的，而不是可用堆内存。

Fielddata 的

想我 正在 行一个 站允 用 收听他 喜 的歌曲。 了 他 可以更容易的管理自己的音 , 用 可以 歌曲 置任何他 喜 的 , 我 就会有很多歌曲被附上 rock () 、 hiphop (哈) 和 electronica (音) , 但也会有些歌曲被附上 my_16th_birthday_favorite_anthem 的 。

在 想我 想要 用 展示 首歌曲最受 迎的三个 , 很有可能 rock 的 会排在三个中的最前面, 而 my_16th_birthday_favorite_anthem 不太可能得到 。 尽管如此, 了 算最受 迎的 , 我 必 制将 些一次性使用的 加 到内存中。

感 fielddata , 我 可以控制 状况。我 知道 自己只 最流行的 感 趣, 所以我 可以 地避免加 那些不太有意思的 尾 :

```
PUT /music/_mapping/song
{
  "properties": {
    "tag": {
      "type": "string",
      "fielddata": { ①
        "filter": {
          "frequency": { ②
            "min": 0.01, ③
            "min_segment_size": 500 ④
          }
        }
      }
    }
  }
}
```

① fielddata 字允 我 配置 fielddata 理 字段的方式。

② frequency 器允 我 基于 率 加 fielddata。

③ 只加 那些至少在本段文 中出 1% 的 。

④ 忽略任何文 个数小于 500 的段。

有了 个映射, 只有那些至少在 本段 文 中出 超 1% 的 才会被加 到内存中。我 也可以指定一个 最大 , 它可以被用来排除 常用 , 比如 停用 。

情况下, 是按照段来 算的。 是 的一个限制 : fielddata 是按段来加 的, 所以可 的 只是 段内的 率。但是, 个限制也有些有趣的特性 : 它可以 受 迎的新 迅速提升到 部。

比如一个新 格的歌曲在一夜之 受大 迎, 我 可能想要将 新 格的歌曲 包括在最受 迎列表 中, 但如果我 倚 索引做完整的 算 取 , 我 就必 等到新 得像 rock 和 electronica) 一 流行。 由于 度 的 方式, 新加的 会很快作 高 出 在新段内, 也当然会迅速上升到 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 , 它的

会非常粗略没有任何意。小的分段会很快被合并到更大的分段中，某一刻超一个限制，将会被入 算。

TIP 通 次来 并不是唯一的 ，我 也可以使用正 式来决定只加 那些匹配的 。例 如，我 可以用 `regex` 器 理 twitte 上的消息只将以 # 号 始的 加 到内存中。假 我 使用的分析器会保留 点符号，像 `whitespace` 分析器。

Fielddata 内存使用有 巨大的 影 ， 衡也是 而易 的：我 上是在忽略数据。但 于很多 用， 衡是合理的，因 些数据根本就没有被使用到。内存的 省通常要比包括一个大量而无用的 尾 更 重要。

加 fielddata

Elasticsearch 加 内存 fielddata 的 行 是 延 加 。当 Elasticsearch 第一次 某个字段 ，它将会完整加 个字段所有 Segment 中的倒排索引到内存中，以便于以后的 能 取更好的性能。

于小索引段来 ， 个 程的需要的 可以忽略。但如果我 有一些 5 GB 的索引段，并希望加 10 GB 的 fielddata 到内存中， 个 程可能会要数十秒。 已 秒 的用 很 会接受停 数秒 着没反 的 站。

有三 方式可以解决 个延 高峰：

- 加 fielddata
- 加 全局序号
- 存

所有的 化都基于同一概念： 加 fielddata， 在用 行搜索 就不会 到延 高峰。

加 fielddata (Eagerly Loading Fielddata)

第一个工具称 加 (与 的 延 加 相)。随着新分段的 建(通 刷新、写入或合并等方式)， 字段 加 可以使那些 搜索不可 的分段里的 fielddata 提前加 。

就意味着首次命中分段的 不需要促 fielddata 的加 ，因 fielddata 已 被 入到内存。避免了用 遇到搜索 的情形。

加 是按字段 用的，所以我 可以控制具体 个字段可以 先加 ：

```

PUT /music/_mapping/_song
{
  "tags": {
    "type": "string",
    "fielddata": {
      "loading" : "eager" ①
    }
  }
}

```

① 置 `fielddata.loading: eager` 可以告 Elasticsearch 先将此字段的内容 入内存中。

Fielddata 的 入可以使用 `update-mapping API` 已有字段 置 `lazy` 或 `eager` 模式。

加 只是 的将 入 fielddata 的代 移到索引刷新的 候，而不是
，从而大大提高了搜索体 。

WARNING

体 大的索引段会比体 小的索引段需要更 的刷新 。通常，体 大的索引段是
由那些已 可 的小分段合并而成的，所以 慢的刷新 也很重要。

全局序号 (Global Ordinals)

有 可以用来降低字符串 fielddata 内存使用的技 叫做 序号 。

想我 有十 文 ， 个文 都有自己的 `status` 状 字段，状 共有三 : `status_pending` 、
`status_published` 、 `status_deleted` 。如果我 个文 都保留其状 的完整字符串形式，那
个文 就需要使用 14 到 16 字 ，或 共 15 GB。

取而代之的是我 可以指定三个不同的字符串， 其排序、 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 一次， 个文 只要使用数 号的序号来替代它原始的 。

Doc Ordinal

0 1 # pending
1 1 # pending
2 2 # published
3 0 # deleted

可以将内存使用从 15 GB 降到 1 GB 以下！

但 里有个 ， 得 fielddata 是按分 段 来 存的。如果一个分段只包含 个状 (`status_deleted` 和 `status_published`)。那 果中的序号 (0 和 1) 就会与包含所有三个状 的分段不一 。

如果我 `status` 字段 行 `terms` 聚合，我 需要 字符串的 行聚合，也就是 我 需要 所有分段中相同的 。一个 粗暴的方式就是 个分段 行聚合操作，返回 个分段的字符串 ，再将它 得出完整的 果。尽管 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 全局序号 的 。全局序号是一个 建在 fielddata 之上的数据 ，它只占用少量内存。唯一 是 跨所有分段 的，然后将它 存入一个序号列表中，正如我 描述 的那 。

在， `terms` 聚合可以 全局序号 行聚合操作，将序号 成真 字符串 的 程只会在聚合 束 生一次。 会将聚合 (和排序) 的性能提高三到四倍。

建全局序号 (Building global ordinals)

当然，天下没有免 的 餐。 全局序号分布在索引的所有段中，所以如果新 或 除一个分段 ，需要 全局序号 行重建。 重建需要 取 个分段的 个唯一 ，基数越高 (即存在更多的唯一) 个 程会越 。

全局序号是 建在内存 fielddata 和 doc values 之上的。 上，它 正是 doc values 性能表 不 的一个主要原因。

和 fielddata 加 一 ，全局序号 也是延 建的。首个需要 索引内 fielddata 的 求会促 全局序号的 建。由于字段的基数不同， 会 致 用 来 著延 一糟 果。一旦全局序号 生 重建， 会 使用旧的全局序号，直到索引中的分段 生 化：在刷新、写入或合并之后。

建全局序号 (Eager global ordinals)

个字符串字段 可以通 配置 先 建全局序号：

```
PUT /music/_mapping/_song
{
  "song_title": {
    "type": "string",
    "fielddata": {
      "loading" : "eager_global_ordinals" ①
    }
  }
}
```

① 置 `eager_global_ordinals` 也暗示着 fielddata 是 加 的。

正如 fielddata 的 加 一 ， 建全局序号 生在新分段 于搜索可 之前。

NOTE

序号的 建只被 用于字符串。数 信息 (integers (整数)、geopoints (地理 度) 、dates (日期) 等等) 不需要使用序号映射，因 些 自己本 上就是序号映射。

因此，我 只能 字符串字段 建其全局序号。

也可以 Doc values 行全局序号 建：

```
PUT /music/_mapping/_song
{
  "song_title": {
    "type": "string",
    "doc_values": true,
    "fielddata": {
      "loading": "eager_global_ordinals" ①
    }
  }
}
```

① 情况下，fielddata 没有 入到内存中，而是 doc values 被 入到文件系 存中。

与 fielddata 加 不一，建全局序号会 数据的 性 生影，
建一个高基数的全局序号会使一个刷新延 数秒。 在于是 次刷新 付出代，
是在刷新后的第一次 。如果 常索引而 少，那 在 付出代 要比 次刷新 要好。如
果写大于 ，那 在 在 重建全局序号将会是一个更好的 。

TIP 景化全局序号的重建 次。如果我 有高基数组需要花数秒 重建，加
refresh_interval 的刷新的 从而可以使我的全局序号保留更 的有效期， 也会 省
CPU 源，因 我 重建的 次下降了。

索引 器 (Index Warmers)

最后我 索引 器。 器早于 fielddata 加 和全局序号 加 之前出，它
然尤其存在的理由。一个索引 器允 我 指定一个 和聚合 要在新分片 于搜索可 之前 行
。 个想法是通 先填充或 存 用 永 无法遇到延 的波峰。

原来， 器最重要的用法是 保 fielddata 被 先加，因 通常是最耗 的一。 在可以通
前面 的那些技 来更好的控制它，但是 器 是可以用来 建 器 存，当然我 也 是能 用
它来 加 fielddata。

我 注 一个 器然后解 生了什：

```

PUT /music/_warmer/warmer_1 ①
{
  "query" : {
    "bool" : {
      "filter" : {
        "bool": {
          "should": [ ②
            { "term": { "tag": "rock" } },
            { "term": { "tag": "hiphop" } },
            { "term": { "tag": "electronics" } }
          ]
        }
      }
    }
  },
  "aggs" : {
    "price" : {
      "histogram" : {
        "field" : "price", ③
        "interval" : 10
      }
    }
  }
}

```

① 器被 到索引 (`music`) 上，使用接入口 `_warmer` 以及 ID (`warmer_1`) 。

② 三 最受 迎的曲 建 器 存。

③ 字段 price 的 fielddata 和全局序号会被 加 。

器是根据具体索引注 的， 个 器都有唯一的 ID， 因 个索引可能有多个 器。

然后我 可以指定 ， 任何 。它可以包括 、 器、聚合、排序 、脚本，任何有效的 表 式都 不夸 。 里的目的是想注 那些可以代表用 生流量 力的 ， 从而将合 的内容 入 存。

当新建一个分段 ， Elasticsearch 将会 行注 在 器中的 。 行 些 会 制加 存， 只有在所有 器 行完， 个分段才会 搜索可 。

与 加 似， 器只是将冷 存的代 移到刷新的 候。当注 器 ， 做 出明智的决定十分重要。 了 保 个 存都被 入， 我 可以 加入上千的 器， 但 也会使新分段 于搜索可 的 急 上升。

中， 我 会 少量代表大多数用 的 ， 然后注 它 。

有些管理的 (比如 得已有 器和 除 器) 没有在本小 提到， 剩下的 内容可以参考 器文 ([warmers documentation](#)) 。

WARNING

化聚合

“elasticsearch 里面桶的叫法和 SQL 里面分的概念是似的，一个桶就似 SQL 里面的一个 group，多嵌套的 aggregation，似 SQL 里面的多字段分 (group by field1,field2,)，注意里是概念似，底的原理是不一的。—者注”

terms 桶基于我的数据建桶；它并不知道到底生成了多少桶。大多数时候一个字段的聚合是非常快的，但是当需要同聚合多个字段，就可能会生大量的分，最果就是占用 es 大量内存，从而致 OOM 的情况生。

假我有一些于影的数据集，一条数据里面会有一个数型的字段存表演影的所有演的名字。

```
{  
  "actors" : [  
    "Fred Jones",  
    "Mary Jane",  
    "Elizabeth Worthing"  
  ]  
}
```

如果我想要出演影片最多的十个演以及与他合作最多的演，使用聚合是非常的：

```
{  
  "aggs" : {  
    "actors" : {  
      "terms" : {  
        "field" : "actors",  
        "size" : 10  
      },  
      "aggs" : {  
        "costars" : {  
          "terms" : {  
            "field" : "actors",  
            "size" : 5  
          }  
        }  
      }  
    }  
  }  
}
```

会返回前十位出演最多的演，以及与他合作最多的五位演。看起来是一个的聚合，最只返回 50 条数据！

但是，个看上去的可以而易地消耗大量内存，我可以通在内存中建一个来看个 terms 聚合。**actors** 聚合会建的第一，个演都有一个桶。然后，内套在第一的个点之下，**costar** 聚合会建第二，个合出演一个桶，参 [Build full depth tree](#) 所示。意味着

部影片会生成 n^2 个桶！

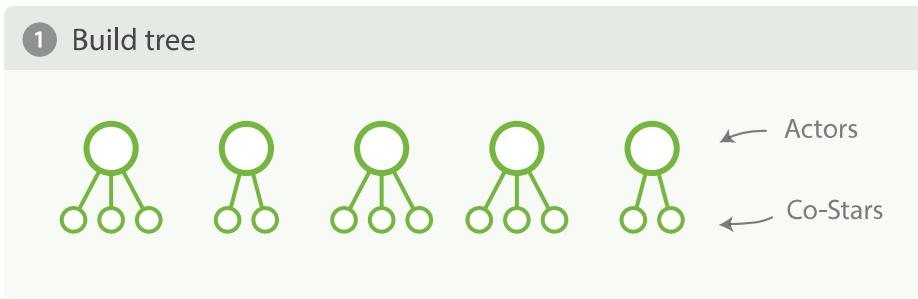


Figure 42. Build full depth tree

用真 点的数据， 想平均 部影片有 10 名演 ， 部影片就会生成 $10^2 == 100$ 个桶。如果 共有 20 , 000 部影片，粗率 算就会生成 2, 000, 000 个桶。

在， 住， 聚合只是 的希望得到前十位演 和与他 合出演者， 共 50 条数据。 了得到最 的 果，我 建了一个有 2, 000, 000 桶的 ，然后 其排序，取 top10。 Sort tree 和 Prune tree 个 程 行了 述。

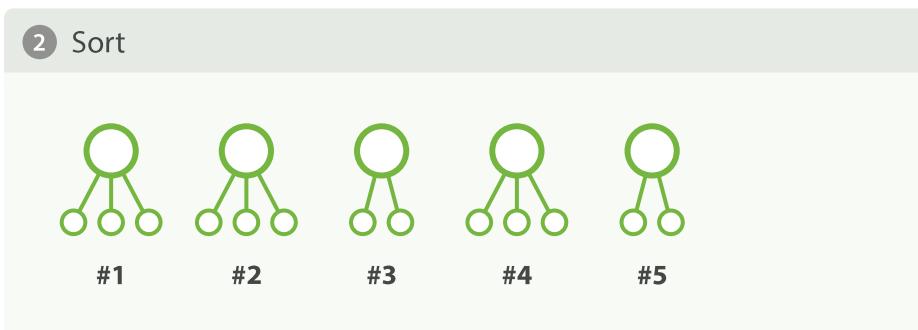


Figure 43. Sort tree

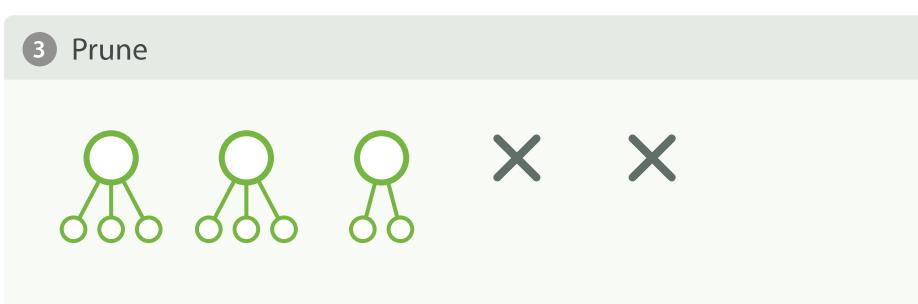


Figure 44. Prune tree

我 一定非常 狂，在 2 万条数据下 行任何聚合 都是 无 力的。如果我 有 2 文 ，想要得到前 100 位演 以及与他 合作最多的 20 位演 ，作 的最 果会出 什 情况 ？

可以推 聚合出来的分 数非常大，会使 策略 以 持。世界上并不存在足 的内存来支持 不受控 制的聚合 。

深度 先与广度 先 (Depth-First Versus Breadth-First)

Elasticsearch 允许我改聚合的集合模式，就是了 状况。我之前展示的策略叫做 深度先 ，它是 置， 先 建完整的 ，然后修剪无用 点。 深度先 的方式于大多数聚合都能正常工作，但于如我演 和 合演 例子的情形就不太用。

了些特殊的用景，我 使用一集合策略叫做 广度先 。 策略的工作方式有些不同，它先行第一聚合，再下一聚合之前会先做修剪。[Build first level](#) 和 [Prune first level](#) 个程行了述。

在我 的示例中，`actors` 聚合会首先 行，在 个候，我的 只有一，但我 已 知道了前 10 位的演！ 就没有必要保留其他的演 信息，因 它 无 如何都不会出 在前十位中。

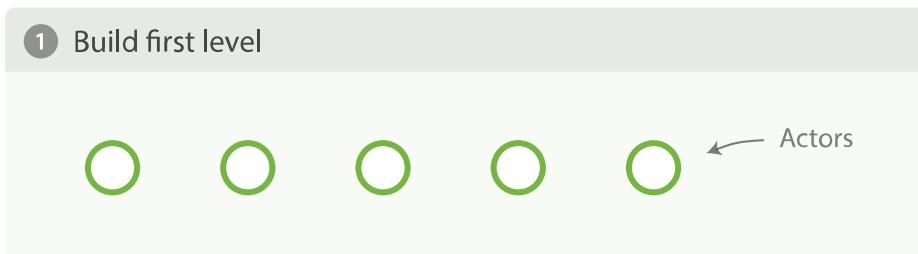


Figure 45. Build first level

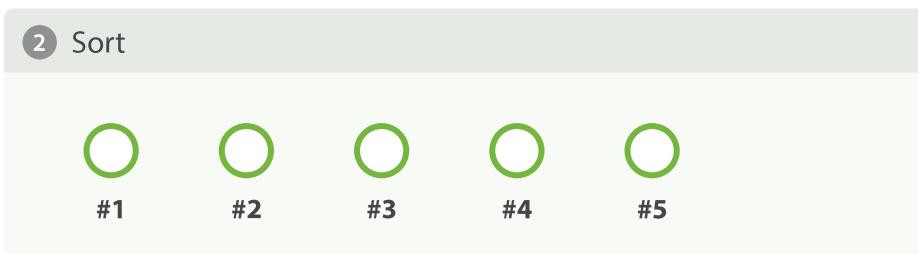


Figure 46. Sort first level

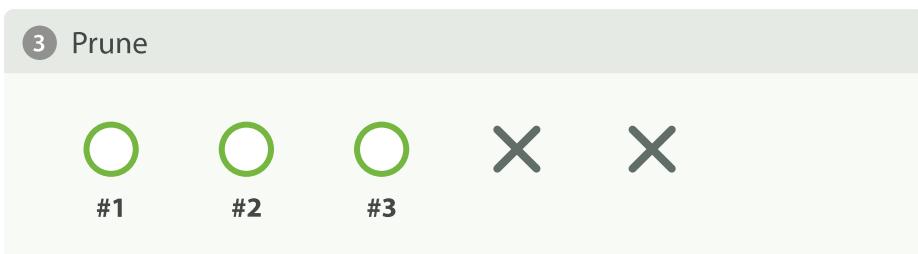


Figure 47. Prune first level

因 我 已 知道了前十名演，我 可以安全的修剪其他 点。修剪后，下一 是基于 它的 行模式

入的，重行个程直到聚合完成，如 [Populate full depth for remaining nodes](#) 所示。景下，广度先可以大幅度省内存。

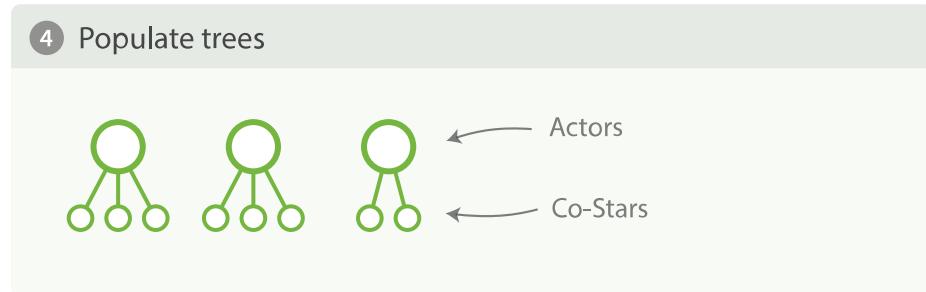


Figure 48. Populate full depth for remaining nodes

要使用广度先，只需的通参数 `collect`：

```
{
  "aggs" : {
    "actors" : {
      "terms" : {
        "field" : "actors",
        "size" : 10,
        "collect_mode" : "breadth_first" ①
      },
      "aggs" : {
        "costars" : {
          "terms" : {
            "field" : "actors",
            "size" : 5
          }
        }
      }
    }
  }
}
```

① 按聚合来 `breadth_first`。

广度先用于个的聚合数量小于当前数的情况下，因广度先会在内存中存裁剪后的需要存的个的所有数据，以便于它的子聚合分可以用上聚合的数据。

广度先的内存使用情况与裁剪后的存分数据量是成性的。于很多聚合来，个桶内的文数量是相当大的。想象一按月分的直方，数肯定是固定的，因年只有12个月，个候个月下的数据量可能非常大。使广度先不是一个好的，也是什深度先作策略的原因。

上面演的例子，如果数据量越大，那的使用深度先的聚合模式生成的分数就会非常多，但是估二的聚合字段分后的数据量相比的分数会小很多所以情况下使用广度先的模式能大大省内存，从而通化聚合模式来大大提高了在某些特定景下聚合的成功率。

本小节涵了多基本理以及很多深入的技。聚合 Elasticsearch 来了以言的大能力和活性。桶与度量的嵌套能力，基数与百分位数的快速估算能力，定位信息中常的能力，所有的些都在近乎的情况下操作的，而且全文搜索是并行的，它改了很多企游。

聚合是一功能特性：一旦我始使用它，我就能到很多其他的可用景。表与分析于很多来都是核心功能（无是用于商智能是服器日志）。

Elasticsearch 大多数字段用 doc values，所以在一些搜索景大大的省了内存使用量，但是需要注意的是只有不分的 string 型的字段才能使用特性。

内存的管理形式可以有多形式，取决于我特定的用景：

- 在，好数据，使聚合行在 not_analyzed 字符串而不是 analyzed 字符串，可以有效的利用 doc values。
- 在，分析不会在之后的聚合算中建高基数组。
- 在搜索，合理利用近似聚合和数据。
- 在点，置硬内存大小以及的断熔限制。
- 在用，通控集群内存的使用情况和 Full GC 的生率，来整是否需要集群源添加更多的机器点

大多数施会用到以上一或几方法。切的合方式与我特定的系境高度相。

无论采取何方式，于有的行估，并同建短期和长期，都十分重要。先决定当前内存的使用情况和需要做的事情（如果有），通估数据速度，来决定未来半年或者一年的集群的，使用何方式来展。

最好在建立集群之前就好些内容，而不是在我集群堆内存使用 90% 的时候再抱佛脚。

地理位置

我拿着地漫城市的日子一去不返了。得益于智能手机，我在是可以知道自己所的准位置，也料到站会使用些信息。我想知道从当前位置行 5 分内可到的那些餐，敦更大内的其他餐并不感趣。

但地理位置功能是 Elasticsearch 的山一角，Elasticsearch 的妙在于，它可以把地理位置、全文搜索、化搜索和分析合到一起。

例如：告我提到 vitello tonnato 食物、行 5 分内可到、且上 11 点的餐，然后合用、距、格排序。一个例子：我展示一幅整个城市 8 月可用假期出租物的地，并算出个区域的平均格。

Elasticsearch 提供了表示地理位置的方式：用度—度表示的坐点使用 geo_point 字段型，以 GeoJSON 格式定的地理形状，使用 geo_shape 字段型。

Geo-points 允许到距一个坐点一定内的坐点、算出点之的距来排序或行相性打分、或者聚合到示在地上的一个格。一方面，Geo-shapes 纯粹是用来的。它

可以用来判断一个地理形状是否有重合或者某个地理形状是否完全包含了其他地理形状。

地理坐 点

地理坐 点 是指地球表面可以用 度描述的一个点。 地理坐 点可以用来 算 个坐 的距 ， 可以判断一个坐 是否在一个区域中，或在聚合中。

地理坐 点不能被 映射 (dynamic mapping) 自 ， 而是需要 式声明 字段 型 geo-point :

```
PUT /attractions
{
  "mappings": {
    "restaurant": {
      "properties": {
        "name": {
          "type": "string"
        },
        "location": {
          "type": "geo_point"
        }
      }
    }
  }
}
```

度坐 格式

如上例， location 字段被声明 geo_point 后，我 就可以索引包含了 度信息的文 了。
度信息的形式可以是字符串、数 或者 象：

```

PUT /attractions/restaurant/1
{
  "name": "Chipotle Mexican Grill",
  "location": "40.715, -74.011" ①
}

PUT /attractions/restaurant/2
{
  "name": "Pala Pizza",
  "location": { ②
    "lat": 40.722,
    "lon": -73.989
  }
}

PUT /attractions/restaurant/3
{
  "name": "Mini Munchies Pizza",
  "location": [ -73.983, 40.719 ] ③
}

```

① 字符串形式以半角逗号分割，如 "`lat,lon`"。

② 象形式 式命名 `lat` 和 `lon`。

③ 数 形式表示 `[lon,lat]`。

可能所有人都至少一次 个坑：地理坐 点用字符串形式表示 是 度在前， 度 在后（`<code>"latitude,longitude"</code>`），而数 形式表示 是 度在前， 度在后（`<code>[longitude,latitude]</code>`）序 好相反。

CAUTION 其，在 Elasticsearch 内部，不管字符串形式 是数 形式，都是 度在前， 度在后。不 早期 了 配 GeoJSON 的格式 ， 整了数 形式的表示方式。

因此，在使用地理位置的路上就出 了 一个“捕熊器”， 坑那些不了解 个陷 的使用者。

通 地理坐 点

有四 地理坐 点相 的 器可以用来 中或者排除文 ：

`geo_bounding_box`

出落在指定矩形 中的点。

`geo_distance`

出与指定位在 定距 内的点。

`geo_distance_range`

出与指定点距 在 定最小距 和最大距 之 的点。

geo_polygon

出落在多形中的点。一个器使用代价很大。当得自己需要使用它，最好先看看 geo-shapes。

些器判断点是否落在指定区域的算方法有不同，但相似。指定的区域被成一系列以quad/geohash前的tokens，并被用来在倒排索引中搜索有相同tokens的文。

地理坐器使用代价昂贵—所以最好在文集合尽可能少的景下使用。可以先使用那些快捷的器，比如 term 或 range，来掉尽可能多的文，最后才交地理坐器理。

TIP

布型器 bool filter 会自做件事。它会先那些基于“bitset”的器(于存)来掉尽可能多的文，然后依次才是更昂贵的地理坐器或者脚本的器。

地理坐模型器

是目前为止最有效的地理坐器了，因为它算起来非常快。指定一个矩形的部，底部，左界，和右界，然后器只需判断坐的度是否在左右界之间，度是否在上下界之间：

```
GET /attractions/restaurant/_search
{
  "query": {
    "filtered": {
      "filter": {
        "geo_bounding_box": {
          "location": {①
            "top_left": {
              "lat": 40.8,
              "lon": -74.0
            },
            "bottom_right": {
              "lat": 40.7,
              "lon": -73.0
            }
          }
        }
      }
    }
  }
}
```

① 些坐也可以用 bottom_left 和 top_right 来表示。

化模型

地理坐模型器不需要把所有坐点都加到内存里。因它要做的只是判断 lat 和 lon 坐数是否在定的内，可以用倒排索引做一个 range 来目。

要使用化方式，需要把 `geo_point` 字段用 `lat` 和 `lon` 的方式分映射到索引中：

```
PUT /attractions
{
  "mappings": {
    "restaurant": {
      "properties": {
        "name": {
          "type": "string"
        },
        "location": {
          "type": "geo_point",
          "lat_lon": true ①
        }
      }
    }
  }
}
```

① `location.lat` 和 `location.lon` 字段将被分索引。它们可以被用于搜索，但是不会在搜索结果中返回。

然后，需要告诉 Elasticsearch 使用已索引的 `lat` 和 `lon`：

```
GET /attractions/restaurant/_search
{
  "query": {
    "filtered": {
      "filter": {
        "geo_bounding_box": {
          "type": "indexed", ①
          "location": {
            "top_left": {
              "lat": 40.8,
              "lon": -74.0
            },
            "bottom_right": {
              "lat": 40.7,
              "lon": -73.0
            }
          }
        }
      }
    }
  }
}
```

① 置 `type` 参数 `indexed`（替代 `memory`）来明 告 Elasticsearch 个
器使用倒排索引。

CAUTION

`geo_point` 型的字段可以包含多个地理坐标点，但是度分索引的化方式只包含一个坐标点的字段有效。

地理距离器

地理距离器 (`geo_distance`) 以定位中心画一个圆，来找出那些地理坐标落在其中的文字：

```
GET /attractions/restaurant/_search
{
  "query": {
    "filtered": {
      "filter": {
        "geo_distance": {
          "distance": "1km", ①
          "location": { ②
            "lat": 40.715,
            "lon": -73.988
          }
        }
      }
    }
  }
}
```

① 找出所有与指定点距离在 1km 内的 `location` 字段。看所支持的距离单位。

② 中心点可以表示字符串、数字或者（如示例中的）对象。坐标格式。

地理距离器 算代价高昂。为了优化性能，Elasticsearch 先画一个矩形来包围整个形状，就可以先用消耗较少的模型算方式来排除掉尽可能多的文本。然后只落在模型内的部分点用地理距离算方式处理。

TIP

需要判断的用，是否需要如此精确的使用模型来做距离？通常使用矩形模型 `bounding box` 是比地理距离更高效的方式，并且往往也能满足需求。

更快的地理距离算

点的距离算，有多性能取精度的算法：

arc

最慢但最精确的是 `arc` 算方式，这种方式把世界当作球体来处理。不过这种方式的精度有限，因为世界并不是完全的球体。

plane

`plane` 算方式把地球当成是平坦的，这种方式快一些但是精度略差。在赤道附近的位置精度最好，而近。

sloppy_arc

如此命名，是因为它使用了 Lucene 的 `SloppyMath`。是一用精度取速度的算方式，它使用

Haversine formula 来 算距 。它比 arc 算方式快 4 到 5 倍，并且距 精度 99.9%。也是 的 算方式。

可以参考下例来指定不同的 算方式：

```
GET /attractions/restaurant/_search
{
  "query": {
    "filtered": {
      "filter": {
        "geo_distance": {
          "distance": "1km",
          "distance_type": "plane", ①
          "location": {
            "lat": 40.715,
            "lon": -73.988
          }
        }
      }
    }
  }
}
```

① 使用更快但精度 差的 plane 算方法。

TIP

真的会在意一个餐 落在指定 形区域数米之外 ？一些地理位置相 的 用会有 高的精度要求；但大部分 用 景中，使用精度 低但 更快的 算方式可能更好。

地理距 区 器

`geo_distance` 和 `geo_distance_range` 器的唯一差 在于后者是一个 状的，它会排除掉落在内圈中的那部分文 。

指定到中心点的距 也可以 — 表示方式：指定一个最小距 （使用 `gt` 或者 `gte` ）和最大距 （使用 `lt` 和 `lte` ），就像使用 `range` 器—：

```
GET /attractions/restaurant/_search
{
  "query": {
    "filtered": {
      "filter": {
        "geo_distance_range": {
          "gte": "1km", ①
          "lt": "2km", ①
          "location": {
            "lat": 40.715,
            "lon": -73.988
          }
        }
      }
    }
  }
}
```

① 匹配那些距 中心点大于等于 1km 而小于 2km 的位置。

按距 排序

索 果可以按与指定点的距 排序：

TIP 当 可以 按距 排序 , 按距 打分 通常是一个更好的解决方案。

```

GET /attractions/restaurant/_search
{
  "query": {
    "filtered": {
      "filter": {
        "geo_bounding_box": {
          "type": "indexed",
          "location": {
            "top_left": {
              "lat": 40.8,
              "lon": -74.0
            },
            "bottom_right": {
              "lat": 40.4,
              "lon": -73.0
            }
          }
        }
      }
    }
  },
  "sort": [
    {
      "_geo_distance": {
        "location": {①
          "lat": 40.715,
          "lon": -73.998
        },
        "order": "asc",
        "unit": "km", ②
        "distance_type": "plane" ③
      }
    }
  ]
}

```

① 算 个文 中 `location` 字段与指定的 `lat/lon` 点 的距 。

② 将距 以 `km` 位写入到 个返回 果的 `sort` 中。

③ 使用快速但精度略差的 `plane` 算方式。

可能想 : 什 要制定距 的 位 ? 用于排序的 , 我 并不 心比 距 的尺度是英里、公里 是光年。原因是, 个用于排序的 会 置在 个返回 果的 `sort` 元素中。

```

...
"hits": [
  {
    "_index": "attractions",
    "_type": "restaurant",
    "_id": "2",
    "_score": null,
    "_source": {
      "name": "New Malaysia",
      "location": {
        "lat": 40.715,
        "lon": -73.997
      }
    },
    "sort": [
      0.08425653647614346 ①
    ]
  },
  ...

```

① 餐 到我 指定的位置距 是 0.084km。

可以通 置 位 (unit) 来 返回 的形式，匹配 用中需要的。

TIP 地理距 排序可以 多个坐 点来使用，不管 (些坐 点) 是在文 中 是排序参数中。使 用 sort_mode 来指定是否需要使用位置集合的 最小 (min) 最大 (max) 或者 平均 (avg) 距 。如此就可以返回 `` 我的工作地和家最近的朋友" 的 果了。

按距 打分

有可能距 是决定返回 果排序的唯一重要因素，不 更常 的情况是距 会和其它因素，比如全文 索匹 配度、流行程度或者 格一起决定排序 果。

遇到 景 需要在 功能 分 中指定方式 我 把 些因子 理后得到一个 合分。 越近越好 中有个一个例子就是介 地理距 影 排序得分的。

外按距 排序 有个 点就是性能：需要 一个匹配到的文 都 行距 算。而 function_score , 在 rescore 句 中可以限制只 前 n 个 果 行 算。

Geohashes

Geohashes 是一 将 度坐 (lat/lon) 成字符串的方式。 做的初衷只是 了 地理位置在 url 上呈 的形式更加友好，但 在 geohashes 已 成一 在数据 中有效索引地理坐 点和地理形状的方式。

Geohashes 把整个世界分 32 个 元的格子 —— 4 行 8 列 —— 一个格子都用一个字母或者数字 。比如 9 个 元覆 了半个格林 ， 的全部和大不列 的大部分。 一个 元 可以 一 被分解成新的 32 个 元， 些 元又可以 被分解成 32 个更小的 元，不断重 下去。 gc 个

元覆盖了和英格，`gcp` 覆盖了敦的大部分和部分南英格，`gcpuuz94k` 是白金的入口，精到 5 米。

句，`geohash` 的度越，它的精度就越高。如果一个`geohashes` 有一个共同的前缀`—`，`<code>gcpuuz</code>—` 就表示他挨得很近。共同的前缀越长，距离就越近。

也意味着，一个好相的位置，可能会有完全不同的`geohash`。比如，敦 `Millenium Dome` 的`geohash` 是 `u10hbp`，因为它落在了 `u` 个元里，而挨着它的最大的元是 `g`。

地理坐标点可以自索引相同的`geohashes`，更重要的是，他也可以索引所有的`geohashes`。如索引白金入口位置——度`<code>51.501568</code>`，度`<code>-0.141257</code>—`，将会索引下面表格中列出的所有`geohashes`，表格中也列出了各个`geohash`元的近似尺寸：

Geohash	Level	Dimensions
<code>g</code>	1	~ 5,004km x 5,004km
<code>gc</code>	2	~ 1,251km x 625km
<code>gcp</code>	3	~ 156km x 156km
<code>gcpu</code>	4	~ 39km x 19.5km
<code>gcpuu</code>	5	~ 4.9km x 4.9km
<code>gcpuuz</code>	6	~ 1.2km x 0.61km
<code>gcpuuz9</code>	7	~ 152.8m x 152.8m
<code>gcpuuz94</code>	8	~ 38.2m x 19.1m
<code>gcpuuz94k</code>	9	~ 4.78m x 4.78m
<code>gcpuuz94kk</code>	10	~ 1.19m x 0.60m
<code>gcpuuz94kkp</code>	11	~ 14.9cm x 14.9cm
<code>gcpuuz94kkp5</code>	12	~ 3.7cm x 1.8cm

`geohash` 元素器可以使用一些`geohash` 前缀来指出与指定坐标点 (`lat/lon`) 相同的位置。

Geohashes 映射

首先，需要决定使用什么精度。当然也可以使用 12 位的精度来索引所有的地理坐标点，但是真的需要精确到数厘米？如果把精度控制在一个更小一些的，比如 1km，那么可以省大量的索引空间：

```

PUT /attractions
{
  "mappings": {
    "restaurant": {
      "properties": {
        "name": {
          "type": "string"
        },
        "location": {
          "type": "geo_point",
          "geohash_prefix": true, ①
          "geohash_precision": "1km" ②
        }
      }
    }
  }
}

```

- ① 将 `geohash_prefix` 置为 `true` 来告诉 Elasticsearch 使用指定精度来索引 geohash 的前 7 位。
- ② 精度可以是一个具体的数字，代表的 geohash 的精度，也可以是一个距离。`1km` 的精度对应的 geohash 的精度是 7。

通常如上设置，geohash 前 7 中 1 到 7 的部分将被索引，所能提供的精度大约在 150 米。

Geohash 元

`geohash_cell` 做的事情非常简单：把纬度坐标位置根据指定精度变成一个 geohash，然后所有包含这个 geohash 的位置——是非常高效的。

```

GET /attractions/restaurant/_search
{
  "query": {
    "constant_score": {
      "filter": {
        "geohash_cell": {
          "location": {
            "lat": 40.718,
            "lon": -73.983
          },
          "precision": "2km" ①
        }
      }
    }
  }
}

```

- ① `precision` 字段设置的精度不能高于映射 `geohash_precision` 字段指定的精度。

此将 `<code>lat/lon</code>` 坐点成度的 geohash —— 本例中 `<code>dr5rsk</code>`;然后所有包含一个短的位置。

然而，如上例中的写法可能不会返回 2km 内所有的餐。要知道 geohash 上是个矩形，而指定的点可能位于一个矩形中的任何位置。有可能一个点好落在了 geohash 元的附近，但器会排除那些落在相邻的餐。

为了修一个，我可以通置 `neighbors` 参数 `true`，把周的元也包含来：

```
GET /attractions/restaurant/_search
{
  "query": {
    "constant_score": {
      "filter": {
        "geohash_cell": {
          "location": {
            "lat": 40.718,
            "lon": -73.983
          },
          "neighbors": true, ①
          "precision": "2km"
        }
      }
    }
  }
}
```

① 此将会的 geohash 和包它的 geohashes。

明的，`2km` 精度的 geohash 加上周的元，最致一个大的搜索区域。此不是精度而生，但是它非常有效率，而且可以作更高精度的地理位置器的前置器。

TIP 将 `precision` 参数置一个距可能会有性。`2km` 的 `precision` 会被成度 6 的 geohash。上它的尺寸是 $1.2\text{km} \times 0.6\text{km}$ 。可能会明的置度 5 或 6 会更容易理解。

此的一个点是，相比 `geo_bounding_box`，它支持一个字段中有多个坐位置的情况。我在化模型中，置 `lat_lon` 也是一个很有效的方式，但是它只在个字段只有个坐点的情况下有效。

地理位置聚合

然按照地理位置果行或者打分很有用，但是在地呈信息用通常更加有用。一个可能会返回太多果以至于不能独地展一个地理坐点，但是地理位置聚合可以用来将地理坐聚集到更加容易管理的 buckets 中。

理 `geo_point` 型字段的三聚合：

地理位置距

将文按照距一个中心点来分。

geohash 格

将文本按照 geohash 来分隔，用来表示在地球上。

地理位置 界

返回一个包含所有地理位置坐标点的地理界的度坐标，表示地放比例的非常有用。

地理距离聚合

`geo_distance` 聚合一些搜索非常有用，例如到所有距离我 1km 以内的披萨店。搜索结果也被限制在用指定 1km 内，但是我可以添加在 2km 内到的其他结果：

```
GET /attractions/restaurant/_search
{
  "query": {
    "bool": {
      "must": {
        "match": { ①
          "name": "pizza"
        }
      },
      "filter": {
        "geo_bounding_box": {
          "location": { ②
            "top_left": {
              "lat": 40.8,
              "lon": -74.1
            },
            "bottom_right": {
              "lat": 40.4,
              "lon": -73.7
            }
          }
        }
      }
    }
  },
  "aggs": {
    "per_ring": {
      "geo_distance": { ③
        "field": "location",
        "unit": "km",
        "origin": {
          "lat": 40.712,
          "lon": -73.988
        },
        "ranges": [
          { "from": 0, "to": 1 },
          { "from": 1, "to": 2 }
        ]
      }
    }
  }
}
```

```
        },
    },
    "post_filter": { ④
        "geo_distance": {
            "distance": "1km",
            "location": {
                "lat": 40.712,
                "lon": -73.988
            }
        }
    }
}
```

- ① 主 名称中含有 pizza 的 店。
② geo_bounding_box 那些只在 区域的 果。
③ geo_distance 聚合 距 用 1km 以内, 1km 到 2km 的 果的数量。
④ 最后, post_filter 将 果 小至那些在用 1km 内的 店。

前面的 求 如下：

```

"hits": {
    "total": 1,
    "max_score": 0.15342641,
    "hits": [ ①
        {
            "_index": "attractions",
            "_type": "restaurant",
            "_id": "3",
            "_score": 0.15342641,
            "_source": {
                "name": "Mini Munchies Pizza",
                "location": [
                    -73.983,
                    40.719
                ]
            }
        }
    ]
},
"aggregations": {
    "per_ring": { ②
        "buckets": [
            {
                "key": "*-1.0",
                "from": 0,
                "to": 1,
                "doc_count": 1
            },
            {
                "key": "1.0-2.0",
                "from": 1,
                "to": 2,
                "doc_count": 1
            }
        ]
    }
}

```

① `post_filter` 已 将搜索 果 小至 在用 1km 以内的披 店。

② 聚合包括搜索 果加上其他在用 2km 以内的披 店。

在 个例子中，我 算了落在 个同心 内的 店数量。当然，我 可以在 `per_rings` 聚合下面嵌套子聚合来 算 个 的平均 格、最受 迎程度，等等。

Geohash 格聚合

通 一个 返回的 果数量 在地 上 独的 示 一个位置点而言可能太多了。 `geohash_grid` 按照 定 的精度 算 一个点的 geohash 而将附近的位置聚合成一起。

果是一个 格——一个 元格表示一个可以 示在地 上的 geohash 。通 改 geohash 的精度，可以按国家或者城市街区来概括全世界。

聚合是稀疏的——它 返回那些含有文 的 元。如果 geohashes 太精 ，将 生成太多的 buckets ，它将 返回那些包含了大量文 、最密集的10000个 元。然而， 了 算 些是最密集的 Top10000 ，它 是需要 生成 所有的 buckets 。可以通 以下方式来控制 buckets 的 生成数量：

1. 使用 geo_bounding_box 来限制 果。
2. 的 界大小 一个 当的 precision (精度)

```
GET /attractions/restaurant/_search
{
  "size" : 0,
  "query": {
    "constant_score": {
      "filter": {
        "geo_bounding_box": {
          "location": { ①
            "top_left": {
              "lat": 40.8,
              "lon": -74.1
            },
            "bottom_right": {
              "lat": 40.4,
              "lon": -73.7
            }
          }
        }
      }
    },
    "aggs": {
      "new_york": {
        "geohash_grid": { ②
          "field": "location",
          "precision": 5
        }
      }
    }
  }
}
```

① 界 将搜索限制在大 区的

② Geohashes 精度 5 大 是 5km x 5km。

Geohashes 精度 5 ， 个 25平方公里，所以10000个 元按 个精度将覆 250000平方公里。我 指定的 界 ， 44km x 33km，或 1452平方公里，所以我 的 界在安全 内；我 不会在内存中 建了太多的 buckets。

前面的 求 看起来是 的：

```
...
"aggregations": {
  "new_york": {
    "buckets": [ ①
      {
        "key": "dr5rs",
        "doc_count": 2
      },
      {
        "key": "dr5re",
        "doc_count": 1
      }
    ]
  }
}
...
```

① 一个 bucket 包含作 `key` 的 geohash

同，我也没有指定任何子聚合，所以我得到是文档数。如果需要，我也可以了解一些 buckets 中受迎的餐型、平均格或其他。

TIP 要在地球上制些 buckets，需要一个将 geohash 成同等界或中心点的。JavaScript 和其他言已有的会行一个，但也可以从使用 `geo-bounds-agg` 的信息来行似的工作。

地理 界聚合

在我 [之前的例子](#)中，我通一个覆大区的来果。然而，我的果全部都位于曼哈市中心。当我的用示一个地的候，放大包含数据的区域是有意的；展示大量的空白空是没有任何意的。

`geo_bounds` 正好是的：它算封装所有地理位置点需要的最小界：

```
GET /attractions/restaurant/_search
{
  "size" : 0,
  "query": {
    "constant_score": {
      "filter": {
        "geo_bounding_box": {
          "location": {
            "top_left": {
              "lat": 40.8,
              "lon": -74.1
            },
            "bottom_right": {
              "lat": 40.4,
              "lon": -73.9
            }
          }
        }
      }
    }
  },
  "aggs": {
    "new_york": {
      "geohash_grid": {
        "field": "location",
        "precision": 5
      }
    },
    "map_zoom": {①
      "geo_bounds": {
        "field": "location"
      }
    }
  }
}
```

① geo_bounds 聚合将 算封装所有匹配 文 所需要的最小 界 。

在包括了一个可以用来 放地 的 界 。

```
...
"aggregations": {
  "map_zoom": {
    "bounds": {
      "top_left": {
        "lat": 40.722,
        "lon": -74.011
      },
      "bottom_right": {
        "lat": 40.715,
        "lon": -73.983
      }
    }
  },
  ...
}
```

事 上，我 甚至可以在 一个 geohash 元内部使用 `geo_bounds` 聚合， 以免一个 元内的地理位置点 集中在 元的一部分上：

```
GET /attractions/restaurant/_search
{
  "size" : 0,
  "query": {
    "constant_score": {
      "filter": {
        "geo_bounding_box": {
          "location": {
            "top_left": {
              "lat": 40.8,
              "lon": -74.1
            },
            "bottom_right": {
              "lat": 40.4,
              "lon": -73.9
            }
          }
        }
      }
    }
  },
  "aggs": {
    "new_york": {
      "geohash_grid": {
        "field": "location",
        "precision": 5
      },
      "aggs": {
        "cell": { ①
          "geo_bounds": {
            "field": "location"
          }
        }
      }
    }
  }
}
```

① `cell_bounds` 子聚合会
↑ geohash 元 算 界 。

在在 个 元里的点有一个 界 。

```

...
"aggregations": {
  "new_york": {
    "buckets": [
      {
        "key": "dr5rs",
        "doc_count": 2,
        "cell": {
          "bounds": {
            "top_left": {
              "lat": 40.722,
              "lon": -73.989
            },
            "bottom_right": {
              "lat": 40.719,
              "lon": -73.983
            }
          }
        }
      },
      ...
    ],
    ...
  }
}

```

地理形状

地理形状（Geo-shapes）使用与地理坐标完全不同的方法。我们在计算机屏幕上看到的形状并不是由完美的点组成的。而是用一个个着色像素点画出的一个近似。地理形状的工作方式就与此相似。

的形状——比如点集、线形、多边形、多边形、中空多边形——都是通过 geohash 元素画出来的，一些形状会演化成一个被它所覆盖到的 geohash 的集合。

NOTE 上，类型的格可以被用于 geo-shapes：使用我之前提到的 geohash，外有一维是四叉树。四叉树与 geohash 类似，除了四叉树只有一个只有 4 个元素，而不是 32。不同取决于方式的。

成一个形状的 geohash 都作为一个元素被索引在一起。有了些信息，通过看是否有相同的 geohash 元素，就可以很容易地判断一个形状是否有交集。

geo-shapes 有以下作用：判断一个形状与索引的形状的关系；这些关系可能是以下之一：

intersects

的形状与索引的形状有重叠（）。

disjoint

的形状与索引的形状完全不重叠。

within

索引的形状完全被包含在另一个形状中。

Geo-shapes 不能用于 算距 、排序、打分以及聚合。

映射地理形状

与 `geo_point` 型的字段相似， 地理形状也必 在使用前明 映射：

```
PUT /attractions
{
  "mappings": {
    "landmark": {
      "properties": {
        "name": {
          "type": "string"
        },
        "location": {
          "type": "geo_shape"
        }
      }
    }
  }
}
```

需要考 修改 个非常重要的 置： 精度 和 距 差。

精度

精度（ `precision` ）参数 用 来控制生成的 geohash 的最大 度。 精度 9 , 等 同于 尺寸在 5m x 5m 的 `geohash` 。 一个 精度 可能 比 需要 的 精 得 多。

精度越低， 需要 索引 的 元就 越少， 索 也 会 更 快。 当然， 精度越低， 地理形状的准 性就 越 差。 需要 考 自己 的 地理形状 所 需要 的 精度—— 即使 少 1-2 个 等 的 精度 也 能 来 明 的 消耗 收 益。

可以 使用 距 来 指定 精度 —— 如 `<code>50m</code>` 或 `<code>2km</code>—` 不 些 距 最 也 会 成 的 [Geohashes](#) 等 。

距 差

当 索引 一 个多 形 ， 中 区域 很 容易 用 一 个 短 geohash 来 表示。 麻 的 是 部 分， 些 地方 需要 使用 更精 的 `geohashes` 才 能 表示。

当 在 索引 一 个小 地 ， 会 希望 它的 界 比 精 。 些 念碑 一 个 着 一 个 可 不 好。 当 索引 整个 国 家 ， 就 不 需要 高 的 精度 了。 差 个 50 米 左右 也 不可能 引 争。

距 差 指定 地理形状 可以 接受 的 最大 率。 它的 是 `0.025` , 即 2.5% 。 也 就是 ， 大 的 地理形状 （ 比如 国家 ） 相比 小 的 地理形状 （ 比如 念碑 ） 来 ， 容 更加 模糊 的 界。

`0.025` 是 一 个 不 的 初始 。 不 如果 我 容 更大的 率， 地理形状 需要 索引 的 元 就 越少。

索引地理形状

地理形状通常通过[GeoJSON](http://geojson.org/)来表示，是一种将使用JSON的二进制形状方式。一个形状都包含了形状类型：`point`, `line`, `polygon`, `envelope`和一个或多个点度数集合的数组。

CAUTION 在GeoJSON里，度数表示方式通常是度数在前，经度在后。

例如，我用一个多边形来索引阿姆斯特丹的广场：

```
PUT /attractions/landmark/dam_square
{
  "name" : "Dam Square, Amsterdam",
  "location" : {
    "type" : "polygon", ①
    "coordinates" : [[ ②
      [ 4.89218, 52.37356 ],
      [ 4.89205, 52.37276 ],
      [ 4.89301, 52.37274 ],
      [ 4.89392, 52.37250 ],
      [ 4.89431, 52.37287 ],
      [ 4.89331, 52.37346 ],
      [ 4.89305, 52.37326 ],
      [ 4.89218, 52.37356 ]
    ]]
  }
}
```

① `type`参数指明了度数坐标集表示的形状类型。

② `lon/lat`列表描述了多边形的形状。

上例中大量的方括号可能看起来令人困惑，不过GeoJSON的方法非常直观：

1. 用一个数组表示一个度数坐标点：

```
[lon, lat]
```

2. 一个坐标点放到一个数组来表示一个多边形：

```
[[lon, lat], [lon, lat], ... ]
```

3. 一个多边形（`polygon`）形状可以包含多个多边形；第一个表示多边形的外轮廓，后面的多边形表示第一个多边形内部的空洞：

```
[  
  [[lon,lat],[lon,lat], ... ], # main polygon  
  [[lon,lat],[lon,lat], ... ], # hole in main polygon  
  ...  
]
```

参 [Geo-shape mapping documentation](#) 了解更多支持的形状。

地理形状

`geo_shape` 不常的地方在于，它允我使用形状来做，而不是坐点。

个例子，当我用出阿姆斯特丹中央火站，我可以用如下方式，出方1km内的所有地：

```
GET /attractions/landmark/_search  
{  
  "query": {  
    "geo_shape": {  
      "location": {①  
        "shape": {②  
          "type": "circle", ③  
          "radius": "1km",  
          "coordinates": [④  
            4.89994,  
            52.37815  
          ]  
        }  
      }  
    }  
  }  
}
```

- ① 使用 `location` 字段中的地理形状。
- ② 中的形状是由 `shape` 的内容表示。
- ③ 形状是一个半径 1km 的形。
- ④ 安姆斯特丹中央火站入口的坐点。

的，（或者器——工作方式相同）会从已索引的形状中与指定形状有交集的部分。此外，可以把 `relation` 字段置 `disjoint` 来与指定形状不相交的部分，或者置 `within` 来完全落在形状中的。

个例子，我可以阿姆斯特丹中心区域所有的地：

```

GET /attractions/landmark/_search
{
  "query": {
    "geo_shape": {
      "location": {
        "relation": "within", ①
        "shape": {
          "type": "polygon",
          "coordinates": [[ ②
            [4.88330,52.38617],
            [4.87463,52.37254],
            [4.87875,52.36369],
            [4.88939,52.35850],
            [4.89840,52.35755],
            [4.91909,52.36217],
            [4.92656,52.36594],
            [4.93368,52.36615],
            [4.93342,52.37275],
            [4.92690,52.37632],
            [4.88330,52.38617]
          ]]
        }
      }
    }
  }
}

```

① 只匹配完全落在 形状中的已索引的形状。

② 个多 形表示安海斯丹中心区域。

在 中使用已索引的形状

于那些 常会在 中使用的形状，可以把它 索引起来以便在 中可以方便地直接引用名字。以之前的阿海斯丹中部 例，我 可以把它存 成一个 型 neighborhood 的文 。

首先，我 照之前 置 landmark 的方式建立映射：

```

PUT /attractions/_mapping/neighborhood
{
  "properties": {
    "name": {
      "type": "string"
    },
    "location": {
      "type": "geo_shape"
    }
  }
}

```

然后我 索引阿姆斯特丹中部 的形状：

```
PUT /attractions/neighborhood/central_amsterdam
{
  "name" : "Central Amsterdam",
  "location" : {
    "type" : "polygon",
    "coordinates" : [
      [4.88330,52.38617],
      [4.87463,52.37254],
      [4.87875,52.36369],
      [4.88939,52.35850],
      [4.89840,52.35755],
      [4.91909,52.36217],
      [4.92656,52.36594],
      [4.93368,52.36615],
      [4.93342,52.37275],
      [4.92690,52.37632],
      [4.88330,52.38617]
    ]
  }
}
```

形状索引好之后，我 就可以在 中通 `index`，`type` 和 `id` 来引用它了：

```
GET /attractions/landmark/_search
{
  "query": {
    "geo_shape": {
      "location": {
        "relation": "within",
        "indexed_shape": { ①
          "index": "attractions",
          "type": "neighborhood",
          "id": "central_amsterdam",
          "path": "location"
        }
      }
    }
  }
}
```

① 指定 `indexed_shape` 而不是 `shape`，Elasticsearch 就知道需要从指定的文 和 `path` 索出的形状了。

阿姆斯特丹中部 个形状没有什 特 的。同 地，我 也可以在 中使用已 索引好的 姆广 。 个可以 出与 姆广 有交集的 近点：

```

GET /attractions/neighborhood/_search
{
  "query": {
    "geo_shape": {
      "location": {
        "indexed_shape": {
          "index": "attractions",
          "type": "landmark",
          "id": "dam_square",
          "path": "location"
        }
      }
    }
  }
}

```

数据建模

Elasticsearch 是如此与 不同，特 是如果 来自 SQL 的世界。 Elasticsearch 有非常多的点：高性能、可 展、近 搜索，并支持大数据量的数据分析。一切都很容易！ 只需下 并始使用它。

但它不是魔法。 为了充分利用 Elasticsearch， 需要了解它的工作机制，以及如何 它如 所需的行工作。

和 用的 系型数据存 有所不同，Elasticsearch 并没有 理 体之 的 系 出直接的方法。 一个系数据 的黄金法 是 -- 化 的数据（ 式）-- 但 不 用于 Elasticsearch。 在 系理、[嵌套象](#) 和 [父-子 系文](#) 我 了 些提供的方法的 点和 点。

然后在 容 我 提供的快速、 活的 容能力。 当然容并没有一个放之四海而皆准的方案。 需要考 些通 系 生的数据流的具体特点， 据此的模型。例如日志事件或者社交 流 些 序列数据 型，和静 文 集合在 理模型上有着很 大的不同。

最后，我 聊一下 Elasticsearch 里面不能伸 的一件事。

系 理

世界有很多重要的 系：博客帖子有一些 行 有多个交易 行，客 有多个 行， 有多个 明，文件目 有多个文件和子目 。

系型数据 被明 — 不意外—用来 行 系管理：

- 个 体（或行，在 系世界中）可以被主 唯一 。
- 体 化（ 式）。唯一 体的数据只存 一次，而相 体只存 它的主。只能在一个具体位置修改 个 体的数据。
- 体可以 行 ，可以跨 体搜索。

- 个体的化是原子的，一致的，隔的，和持久的。（可以在 *ACID Transactions* 中看更多。）
- 大多数系数据支持跨多个体的 ACID 事。

但是系型数据有其局限性，包括全文索有限的支持能力。体消耗是很昂的，的越多，消耗就越昂。特别是跨服器行体成本其昂，基本不可用。但个的服务器上又存在数据量的限制。

Elasticsearch 和大多数 NoSQL 数据似，是扁平化的。索引是独立文的集合体。文是否匹配搜索求取决于它是否包含所有的所需信息。

Elasticsearch 中个文的数据更是 *ACIDic* 的，而及多个文的事不是。当一个事部分失，无法回索引数据到前一个状。

扁平化有以下：

- 索引程是快速和无的。
- 搜索程是快速和无的。
- 因个文相互都是独立的，大模数据可以在多个点上行分布。

但系然非常重要。某些时候，我需要小扁平化和世界系模型的差。以下四常用的方法，用来在 Elasticsearch 中行系型数据的管理：

- [Application-side joins](#)
- [Data denormalization](#)
- [Nested objects](#)
- [Parent/child relationships](#)

通常都需要合其中的某几个方法来得到最的解决方案。

用接

我通在我用的程序中接可以（部分）模系数据。例如，比方我正在用和她的博客文章行索引。在系世界中，我会来操作：

```
PUT /my_index/user/1 ①
{
  "name": "John Smith",
  "email": "john@smith.com",
  "dob": "1970/10/24"
}
```

```
PUT /my_index/blogpost/2 ①
{
  "title": "Relationships",
  "body": "It's complicated...",
  "user": 1 ②
}
```

① 个文 的 index, type, 和 id 一起 造成主 。

② blogpost 通 用 的 id 接到用 。index 和 type 并不需要因 在我 的 用 程序中已 硬 。

通 用 的 ID 1 可以很容易的 到博客帖子。

```
GET /my_index/blogpost/_search
{
  "query": {
    "filtered": {
      "filter": {
        "term": { "user": 1 }
      }
    }
  }
}
```

了 到用 叫做 John 的博客帖子，我 需要 行 次 : 第一次会 所有叫做 John 的用 从而 取他 的 ID 集合，接着第二次会将 些 ID 集合放到 似于前面一个例子的 :

```

GET /my_index/user/_search
{
  "query": {
    "match": {
      "name": "John"
    }
  }
}

GET /my_index/blogpost/_search
{
  "query": {
    "filtered": {
      "filter": {
        "terms": { "user": [1] } ①
      }
    }
  }
}

```

① 行第一个 得到的 果将填充到 terms 器中。

用 接的主要 点是可以 数据 行 准化 理。只能在 user 文 中修改用 的名称。 点是，了在搜索 接文 ，必 行 外的 。

在 个例子中，只有一个用 匹配我 的第一个 ，但在 世界中，我 可以很 易的遇到数以百万 的叫做 John 的用 。包含所有 些用 的 IDs 会 生一个非常大的 ， 是一个数百万 的 。

方法 用于第一个 体（例如，在 个例子中 user ）只有少量的文 的情况，并且最好它 很少改 。 将允 用程序 果 行 存，并避免 常 行第一次 。

非 化 的数据

使用 Elasticsearch 得到最好的搜索性能的方法是有目的的通 在索引 行非 化 denormalizing 。 个文 保持一定数量的冗余副本可以在需要 避免 行 。

如果我 希望能 通 某个用 姓名 到他写的博客文章，可以在博客文 中包含 个用 的姓名：

```

PUT /my_index/user/1
{
  "name": "John Smith",
  "email": "john@smith.com",
  "dob": "1970/10/24"
}

PUT /my_index/blogpost/2
{
  "title": "Relationships",
  "body": "It's complicated...",
  "user": {
    "id": 1,
    "name": "John Smith" ①
  }
}

```

① 部分用 的字段数据已被冗余到 blogpost 文 中。

在，我 通 次 就能 通 relationships 到用 John 的博客文章。

```

GET /my_index/blogpost/_search
{
  "query": {
    "bool": {
      "must": [
        { "match": { "title": "relationships" } },
        { "match": { "user.name": "John" } }
      ]
    }
  }
}

```

数据非 化的 点是速度快。因 个文 都包含了所需的所有信息，当 些信息需要在 行匹配，并不需要 行昂 的 接操作。

字段折

一个普遍的需求是需要通 特定字段 行分 。例如我 需要按照用 名称 分 返回最相 的博客文章。按照用 名分 意味着 行 terms 聚合。 能 按照用 整体 名称 行分 ，名称字段 保持 not_analyzed 的形式，具体 明参考 [聚合与分析](#)：

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

① user.name 字段将用来 行全文 索。

② user.name.raw 字段将用来通 terms 聚合 行分 。

然后添加一些数据:

```

PUT /my_index/user/1
{
  "name": "John Smith",
  "email": "john@smith.com",
  "dob": "1970/10/24"
}

PUT /my_index/blogpost/2
{
  "title": "Relationships",
  "body": "It's complicated...",
  "user": {
    "id": 1,
    "name": "John Smith"
  }
}

PUT /my_index/user/3
{
  "name": "Alice John",
  "email": "alice@john.com",
  "dob": "1979/01/04"
}

PUT /my_index/blogpost/4
{
  "title": "Relationships are cool",
  "body": "It's not complicated at all...",
  "user": {
    "id": 3,
    "name": "Alice John"
  }
}

```

在我 来 包含 `relationships` 并且作者名包含 `John` 的博客， 果再按作者名分 ， 感 `top_hits aggregation` 提供了按照用 行分 的功能：

```

GET /my_index/blogpost/_search
{
  "size" : 0, ①
  "query": { ②
    "bool": {
      "must": [
        { "match": { "title": "relationships" }},
        { "match": { "user.name": "John" } }
      ]
    }
  },
  "aggs": {
    "users": {
      "terms": {
        "field": "user.name.raw", ③
        "order": { "top_score": "desc" } ④
      },
      "aggs": {
        "top_score": { "max": { "script": "_score" } }, ④
        "blogposts": { "top_hits": { "_source": "title", "size": 5 } } ⑤
      }
    }
  }
}

```

① 我感兴趣的博客文章是通过 `blogposts` 聚合返回的，所以我可以通过将 `size` 置成 0 来禁止 hits 常搜索。

② `query` 返回通过 `relationships` 名称 `John` 的用户的博客文章。

③ `terms` 聚合通过一个 `user.name.raw` 建一个桶。

④ `top_score` 聚合通过 `users` 聚合得到的一个桶按照分数进行排序。

⑤ `top_hits` 聚合通过返回五个最相关的博客文章的 `title` 字段。

结果：

```

...
"hits": {
  "total": 2,
  "max_score": 0,
  "hits": [] ①
},
"aggregations": {
  "users": {
    "buckets": [
      {
        "key": "John Smith", ②
        "doc_count": 1,
        "blogposts": {
          "hits": { ③
            "total": 1,
            "max_score": 0.35258877,
            "hits": [
              {
                "_index": "my_index",
                "_type": "blogpost",
                "_id": "2",
                "_score": 0.35258877,
                "_source": {
                  "title": "Relationships"
                }
              }
            ]
          }
        },
        "top_score": { ④
          "value": 0.3525887727737427
        }
      }
    ],
    ...
  }
}

```

① 因我置 size 0，所以 hits 数是空的。

② 在果中出的每一个用都会有一个的桶。

③ 在一个用桶下面都会有一个 blogposts.hits 数包含一个用的果。

④ 用桶按照一个用最相的博客文章行排序。

使用 top_hits 聚合等效行一个返回些用的名字和他最相的博客文章，然后一个用行相同的，以得最好的博客。但前者的效率要好很多。

一个桶返回的命中果是基于最初主行的一个量迷果集。个迷提供了一些期望的常用特性，例如高亮示以及分功能。

非化和并

当然，数据非化也有弊端。第一个点是索引会更大因为一个博客文章文的将会更大，并且里有很多的索引字段。通常不是一个大。数据写到磁将会上被高度，而且磁已很廉了。Elasticsearch 可以愉快地付些外的数据。

更重要的 是，如果用 改了他的名字，他所有的博客文章也需要更新了。幸的是，用不常更改名称。即使他做了，用也不可能写超几千篇博客文章，所以更新博客文章通 scroll 和 bulk APIs 大概耗不到一秒。

然而，我考一个更的景，其中的化很常，影深，而且非常重要，并。

在个例子中，我将在 Elasticsearch 模一个文件系的目，非常似 Linux 文件系：根目是 /，个目可以包含文件和子目。

我希望能搜索到一个特定目下的文件，等效于：

```
grep "some text" /clinton/projects/elasticsearch/*
```

就要求我索引文件所在目 的路径：

```
PUT /fs/file/1
{
  "name": "README.txt", ①
  "path": "/clinton/projects/elasticsearch", ②
  "contents": "Starting a new Elasticsearch project is easy..."
}
```

① 文件名

② 文件所在目 的全路径

NOTE 事实上，我也当索引 directory 文，如此我可以在此目内列出所有的文件和子目，但为了，我将忽略一个需求。

我也希望能搜索到一个特定目下的目 包含的任何文件，相当于此：

```
grep -r "some text" /clinton
```

了支持一点，我需要路径 次 行索引：

- /clinton
- /clinton/projects
- /clinton/projects/elasticsearch

次能通 path 字段使用 path_hierarchy tokenizer 自生成：

```

PUT /fs
{
  "settings": {
    "analysis": {
      "analyzer": {
        "paths": { ①
          "tokenizer": "path_hierarchy"
        }
      }
    }
  }
}

```

① 自定 的 `paths` 分析器在 置中使用 `path_hierarchy tokenizer`。

`file` 型的映射看起来如下所示：

```

PUT /fs/_mapping/file
{
  "properties": {
    "name": { ①
      "type": "string",
      "index": "not_analyzed"
    },
    "path": { ②
      "type": "string",
      "index": "not_analyzed",
      "fields": {
        "tree": { ②
          "type": "string",
          "analyzer": "paths"
        }
      }
    }
  }
}

```

① `name` 字段将包含 切名称。

② `path` 字段将包含 切的目 名称，而 `path.tree` 字段将包含路径 次 。

一旦索引建立并且文件已被 入索引，我 可以 行一个搜索，在 `/clinton/projects/elasticsearch` 目 中包含 `elasticsearch` 的文件，如下所示：

```

GET /fs/file/_search
{
  "query": {
    "filtered": {
      "query": {
        "match": {
          "contents": "elasticsearch"
        }
      },
      "filter": {
        "term": { ①
          "path": "/clinton/projects/elasticsearch"
        }
      }
    }
  }
}

```

① 在 目 中 文件。

所有在 `/clinton` 下面的任何子目 存放的文件将在 `path.tree` 字段中包含 `/clinton` 。所以我 能 搜索 `/clinton` 的任何子目 中的所有文件，如下所示：

```

GET /fs/file/_search
{
  "query": {
    "filtered": {
      "query": {
        "match": {
          "contents": "elasticsearch"
        }
      },
      "filter": {
        "term": { ①
          "path.tree": "/clinton"
        }
      }
    }
  }
}

```

① 在 个目 或其下任何子目 中 文件。

重命名文件和目

到目前 止一切 利。 重命名一个文件很容易—所需要的只是一个 的 `update` 或 `index` 求。 甚至可以使用 `optimistic concurrency control` 保 的 化不会与其他用 的 化 生冲突：

```

PUT /fs/file/1?version=2 ①
{
  "name": "README.asciidoc",
  "path": "/clinton/projects/elasticsearch",
  "contents": "Starting a new Elasticsearch project is easy..."
}

```

① version 号 保 更改 用于 索引中具有此相同的版本号的文 。

我 甚至可以重命名一个目 , 但 意味着更新所有存在于 目 下路径 次 中的所有文件。

可能快速或 慢, 取决于有多少文件需要更新。我 所需要做的就是使用 scroll 来 索所有的文件, 以及 bulk API 来更新它 。 个 程不是原子的, 但是所有的文件将会迅速 移到他 的新存放位置。

解决并

当我 允 多个人 同 重命名文件或目 , 就来了。 想一下, 正在 一个包含了成百上千文件的目 /clinton 行重命名操作。同 , 一个用 个目 下的 个文件 /clinton/projects/elasticsearch/README.txt 行重命名操作。 个用 的修改操作, 尽管在 的操作后 始, 但可能会更快的完成。

以下有 情况可能出 :

- 决定使用 version (版本) 号, 在 情况下, 当与 README.txt 文件重命名的版本号 生冲突 , 的批量重命名操作将会失 。
- 没有使用版本控制, 的 更将覆 其他用 的 更。

的原因是 Elasticsearch 不支持 ACID 事 。 个文件的 更是 ACIDic 的, 但包含多个文 的 更不支持。

如果 的主要数据存 是 系数据 , 并且 Elasticsearch 作 一个搜索引擎 或一 提升性能的方法, 可以首先在数据 中 行 更 作, 然后在完成后将 些 更 制到 Elasticsearch。通 方式, 将受益于数据 ACID 事 支持, 并且在 Elasticsearch 中以正 的 序 生 更。 并 在 系数据 中得到了 理。

如果 不使用 系型存 , 些并 就需要在 Elasticsearch 的事 水准 行 理。 以下是三个切 可行的使用 Elasticsearch 的解决方案, 它 都 及某 形式的 :

- 全局
- 文
-

TIP 当使用一个外部系 替代 Elasticsearch , 本 中所描述的解决方案可以通 相同的原 来 。

全局

通 在任何 只允 一个 程来 行 更 作, 我 可以完全避免并 。 大多数的 更只

及少量文件，会很快完成。一个 目 的重命名操作会 其他 更造成 的阻塞，但可能很少做。

因 在 Elasticsearch 文 的 更支持 ACIDic，我 可以使用一个文 是否存在的状 作 一个全局 。 了 求得到 ， 我 **create** 全局 文 :

```
PUT /fs/lock/global/_create
{}
```

如果 个 **create** 求因冲突 常而失 ， 明 一个 程已被授予全局 ， 我 将不得不 后再 。 如果 求成功了，我 自豪的成 全局 的主人，然后可以 完成我 的 更。一旦完成，我 就必 通 除全局 文 来 放 :

```
DELETE /fs/lock/global
```

根据 更的 繁程度以及 消耗，一个全局 能 系 造成大幅度的性能限制。 我 可以通 我 的 更 粒度的方式来 加并行度。

文

我 可以使用前面描述相同的方法技 来 定个体文 ， 而不是 定整个文件系 。 我 可以使用 **scrolled search** 索所有的文 ， 些文 会被 更影 因此 一个文 都 建了一个 文件：

```
PUT /fs/lock/_bulk
{ "create": { "_id": 1}} ①
{ "process_id": 123      } ②
{ "create": { "_id": 2}}
{ "process_id": 123      }
```

① **lock** 文 的 ID 将与 被 定的文件的 ID 相同。

② **process_id** 代表要 行 更 程的唯一 ID。

如果一些文件已被 定，部分的 **bulk** 求将失 ， 我 将不得不再次 。

当然，如果我 再次 定 所有 的文件， 我 前面使用的 **create** 句将会失 ， 因所有文件都已被我 定！ 我 需要一个 **update** 求 **upsert** 参数以及下面 个 **script** ， 而不是一个 的 **create** 句：

```
if ( ctx._source.process_id != process_id ) { ①
  assert false; ②
}
ctx.op = 'noop'; ③
```

① **process_id** 是 到脚本的一个参数。

② **assert false** 将引 常， 致更新失 。

③ 将 `op` 从 `update` 更新到 `noop` 防止更新 求作出任何改 , 但 返回成功。

完整的 `update` 求如下所示 :

```
POST /fs/lock/1/_update
{
  "upsert": { "process_id": 123 },
  "script": "if ( ctx._source.process_id != process_id )
{ assert false }; ctx.op = 'noop';"
  "params": {
    "process_id": 123
  }
}
```

如果文 并不存在, `upsert` 文 将会被 入—和前面 `create` 求相同。但是, 如果 文件 存在, 脚本会 看存 在文 上的 `process_id` 。如果 `process_id` 匹配, 更新不会 行(`noop`)但脚本会返回成功。如果 者并不匹配, `assert false` 出一个 常, 也知道了 取 的 已失 。

一旦所有 已成功 建, 就可以 行 的 更。

之后, 必 放所有的 , 通 索所有的 文 并 行批量 除, 可以完成 的 放 :

```
POST /fs/_refresh ①

GET /fs/lock/_search?scroll=1m ②
{
  "sort" : ["_doc"],
  "query": {
    "match" : {
      "process_id" : 123
    }
  }
}

PUT /fs/lock/_bulk
{ "delete": { "_id": 1}}
{ "delete": { "_id": 2}}
```

① `refresh` 用 保所有 `lock` 文 搜索 求可 。

② 当 需要在 次搜索 求返回大量的 索 果集 , 可以使用 `scroll` 。

文 可以 粒度的 控制, 但是 数百万文 建 文件 也很大。 在某些情况下, 可以用少得多的工作量 粒度的 定, 如以下目 景中所示。

在前面的例子中, 我 可以 定的目 的一部分, 而不是 定 一个 及的文 。 我 将需要独占

我要重命名的文件或目录，它可以通过独占文件来实现：

```
{ "lock_type": "exclusive" }
```

同理我需要共享锁定所有的父目录，通过共享文件：

```
{
  "lock_type": "shared",
  "lock_count": 1 ①
}
```

① `lock_count` 持有共享锁的数量。

`/clinton/projects/elasticsearch/README.txt` 行重命名的程序需要在该文件上有独占锁，以及在 `/clinton`、`/clinton/projects` 和 `/clinton/projects/elasticsearch` 目录下有共享锁。

一个普通的 `create` 请求将满足独占锁的要求，但共享锁需要脚本的更新来做一些额外的处理：

```
if (ctx._source.lock_type == 'exclusive') {
  assert false; ①
}
ctx._source.lock_count++ ②
```

① 如果 `lock_type` 是 `exclusive`（独占）的，`assert` 句将抛出一个异常，导致更新请求失败。

② 否则，我将 `lock_count` 行量处理。

一个脚本处理了 `lock` 文档已存在的情况，但我也需要一个用来处理的文档不存在情况的 `upsert` 文档。完整的更新请求如下：

```
POST /fs/lock/%2Fclinton/_update ①
{
  "upsert": { ②
    "lock_type": "shared",
    "lock_count": 1
  },
  "script": "if (ctx._source.lock_type == 'exclusive')
{ assert false }; ctx._source.lock_count++"
}
```

① 文档的 ID 是 `/clinton`，URL 后成为 `%2fclinton`。

② `upsert` 文档如果不存在，会被插入。

一旦我成功地在所有的父目录中获得一个共享锁，我将在文件本身 `create` 一个独占锁：

```
PUT /fs/lock/%2Fclinton%2fprojects%2felasticsearch%2fREADME.txt/_create
{ "lock_type": "exclusive" }
```

在，如果有其他人想要重新命名 `/clinton` 目，他 将不得不在 条路径上 得一个独占：

```
PUT /fs/lock/%2Fclinton/_create
{ "lock_type": "exclusive" }
```

个 求将失，因 一个具有相同 ID 的 `lock` 文 已 存在。 一个用 将不得不等待我 的操作完成以及 放我 的 。独占 只能 被 除：

```
DELETE /fs/lock/%2Fclinton%2fprojects%2felasticsearch%2fREADME.txt
```

共享 需要 一个脚本 `lock_count`，如果 数下降到零， 除 `lock` 文：

```
if (--ctx._source.lock_count == 0) {
  ctx.op = 'delete' ①
}
```

① 一旦 `lock_count` 到0， `ctx.op` 会从 `update` 被修改成 `delete`。

此更新 求将 父目 由下至上的 行，从最 路径到最短路径：

```
POST /fs/lock/%2Fclinton%2fprojects%2felasticsearch/_update
{
  "script": "if (--ctx._source.lock_count == 0) { ctx.op = 'delete' } "
}
```

用最小的代 提供了 粒度的并 控制。当然，它不 用于所有的情况—数据模型必 有 似于目的 序 路径才能使用。

三个方案—全局、文 或 —都没有 理 最棘手的：如果持有 的 程死了 ？

一个 程的意外死亡 我 留下了2个 ！

NOTE

- 我 如何知道我 可以 放的死亡 程中所持有的 ？
- 我 如何清理死去的 程没有完成的 更？

些主 超出了本 的 ，但是如果 决定使用 ， 需要 他 行一些思考。

当非 化成 很多 目的一个很好的 ，采用 方案的需求会 来 的 。 作 替代方案，Elasticsearch 提供 个模型 助我 理相 的 体：嵌套的 象和父子 系。

嵌套象

由于在 Elasticsearch 中，一个文档的修改都是原子性操作，那将相关数据都存 在同一文档中也就理所当然。比如，我可以将及其明的数据存 在一个文档中。又比如，我可以将一篇博客文章的以一个 `comments` 数的形式和博客文章放在一起：

```
PUT /my_index/blogpost/1
{
  "title": "Nest eggs",
  "body": "Making your money work...",
  "tags": [ "cash", "shares" ],
  "comments": [ ①
    {
      "name": "John Smith",
      "comment": "Great article",
      "age": 28,
      "stars": 4,
      "date": "2014-09-01"
    },
    {
      "name": "Alice White",
      "comment": "More like this please",
      "age": 31,
      "stars": 5,
      "date": "2014-10-22"
    }
  ]
}
```

① 如果我 依 字段自 映射，那 `comments` 字段会自 映射 `object` 型。

由于所有的信息都在一个文档中，当我 就没有必要去 合文章和 文， 效率就很高。

但是当我 使用如下 ，上面的文 也会被当做是符合条件的 果：

```
GET /_search
{
  "query": {
    "bool": {
      "must": [
        { "match": { "name": "Alice" } },
        { "match": { "age": 28 } } ①
      ]
    }
  }
}
```

① Alice 是31，不是28！

正如我在象数中的一出上面的原因是 JSON 格式的文被理成如下的扁平式 的。

```
{  
    "title": [ eggs, nest ],  
    "body": [ making, money, work, your ],  
    "tags": [ cash, shares ],  
    "comments.name": [ alice, john, smith, white ],  
    "comments.comment": [ article, great, like, more, please, this ],  
    "comments.age": [ 28, 31 ],  
    "comments.stars": [ 4, 5 ],  
    "comments.date": [ 2014-09-01, 2014-10-22 ]  
}
```

Alice 和 31、John 和 2014-09-01 之的相性信息不再存在。然 object 型(参 内部象)在存一象非常有用,但于象数的搜索而言,无用。

嵌套象就是来解决个的。将 comments 字段型置 nested 而不是 object 后,一个嵌套象都会被索引一个藏的独立文,例如下:

```
{ ①  
    "comments.name": [ john, smith ],  
    "comments.comment": [ article, great ],  
    "comments.age": [ 28 ],  
    "comments.stars": [ 4 ],  
    "comments.date": [ 2014-09-01 ]  
}  
{ ②  
    "comments.name": [ alice, white ],  
    "comments.comment": [ like, more, please, this ],  
    "comments.age": [ 31 ],  
    "comments.stars": [ 5 ],  
    "comments.date": [ 2014-10-22 ]  
}  
{ ③  
    "title": [ eggs, nest ],  
    "body": [ making, money, work, your ],  
    "tags": [ cash, shares ]  
}
```

① 第一个嵌套文

② 第二个嵌套文

③ 根文 或者也可称父文

在独立索引一个嵌套象后,象中个字段的相性得以保留。我 ,也返回那些真正符合条件的文。

不如此,由于嵌套文直接存 在文内部,嵌套文 和根文合成本很低,速度和独存几乎一

嵌套文 是 藏存 的,我 不能直接 取。如果要 改一个嵌套 象,我 必 把整个文 重新索引才可以。 得注意的是, 的 候返回的是整个文 ,而不是嵌套文 本身。

嵌套 象映射

置一个字段 `nested` 很 一 只需要将字段 型 `object` 替 `nested` 即可：

```
PUT /my_index
{
  "mappings": {
    "blogpost": {
      "properties": {
        "comments": {
          "type": "nested", ①
          "properties": {
            "name": { "type": "string" },
            "comment": { "type": "string" },
            "age": { "type": "short" },
            "stars": { "type": "short" },
            "date": { "type": "date" }
          }
        }
      }
    }
  }
}
```

① `nested` 字段 型的 置参数与 `object` 相同。

就是需要 置的一切。至此, 所有 `comments` 象会被索引在独立的嵌套文 中。可以 看 `nested` 型参考文 取更多 信息。

嵌套 象

由于嵌套 象 被索引在独立 藏的文 中, 我 无法直接 它 。相 地, 我 必 使用 `nested` 去 取它 :

```

GET /my_index/blogpost/_search
{
  "query": {
    "bool": {
      "must": [
        {
          "match": {
            "title": "eggs" ①
          }
        },
        {
          "nested": {
            "path": "comments", ②
            "query": {
              "bool": {
                "must": [ ③
                  {
                    "match": {
                      "comments.name": "john"
                    }
                  },
                  {
                    "match": {
                      "comments.age": 28
                    }
                  }
                ]
              }
            }
          }
        }
      ]
    }
  }
}

```

① `title` 子句是 根文 的。

② `nested` 子句作用于嵌套字段 `comments`。在此 中，既不能 根文 字段，也不能 其他嵌套文。

③ `comments.name` 和 `comments.age` 子句操作在同一个嵌套文 中。

TIP `nested` 字段可以包含其他的 `nested` 字段。同 地，`nested` 也可以包含其他的 `nested`。而嵌套的 次会按照 所期待的被 用。

`nested` 肯定可以匹配到多个嵌套的文 。一个匹配的嵌套文 都有自己的相 度得分，但是 多的分数最 需要 聚 可供根文 使用的一个分数。

情况下，根文 的分数是 些嵌套文 分数的平均 。可以通 置 `score_mode` 参数来控制 个得分策略，相 策略有 `avg`(平均)，`max`(最大)，`sum`(加和) 和 `none`(直接返回 1.0 常数 分数)。

```

GET /my_index/blogpost/_search
{
  "query": {
    "bool": {
      "must": [
        {
          "match": {
            "title": "eggs"
          }
        },
        {
          "nested": {
            "path": "comments",
            "score_mode": "max", ①
            "query": {
              "bool": {
                "must": [
                  {
                    "match": {
                      "comments.name": "john"
                    }
                  },
                  {
                    "match": {
                      "comments.age": 28
                    }
                  }
                ]
              }
            }
          }
        }
      ]
    }
  }
}

```

① 返回最匹配嵌套文的 `_score` 根文 使用。

NOTE

如果 `nested` 放在一个布的 `filter` 子句中，其表就像一个 `nested`，只是 `score_mode` 参数不再生效。因它被用于不打分的 中 — 只是符合或不符合条件，不必打分 — 那 `score_mode` 就没有任何意，因根本就没有要打分的地方。

使用嵌套字段排序

尽管嵌套字段的存于独立的嵌套文中，但依然有方法按照嵌套字段的排序。我添加一个，以使得果更有意思：

```
PUT /my_index/blogpost/2
{
  "title": "Investment secrets",
  "body": "What they don't tell you ...",
  "tags": [ "shares", "equities" ],
  "comments": [
    {
      "name": "Mary Brown",
      "comment": "Lies, lies, lies",
      "age": 42,
      "stars": 1,
      "date": "2014-10-18"
    },
    {
      "name": "John Smith",
      "comment": "You're making it up!",
      "age": 28,
      "stars": 2,
      "date": "2014-10-16"
    }
  ]
}
```

假如我 想要 在10月 收到 的博客文章，并且按照 stars 数的最小 来由小到大排序，那句如下：

```

GET /_search
{
  "query": {
    "nested": { ①
      "path": "comments",
      "filter": {
        "range": {
          "comments.date": {
            "gte": "2014-10-01",
            "lt": "2014-11-01"
          }
        }
      }
    },
    "sort": {
      "comments.stars": { ②
        "order": "asc", ②
        "mode": "min", ②
        "nested_path": "comments", ③
        "nested_filter": {
          "range": {
            "comments.date": {
              "gte": "2014-10-01",
              "lt": "2014-11-01"
            }
          }
        }
      }
    }
  }
}

```

① 此 的 nested 将 果限定 在10月 收到 的博客文章。

② 果按照匹配的 中 comment.stars 字段的最小 (min) 来由小到大 (asc) 排序。

③ 排序子句中的 nested_path 和 nested_filter 和 query 子句中的 nested 相同，原因在下面有解。

我 什 要用 nested_path 和 nested_filter 重 条件 ？原因在于，排序 生在 行之后。 条件限定了只在10月 收到 的博客文 ，但返回整个博客文 。如果我 不在排序子句中加入 nested_filter ， 那 我 博客文 的排序将基于博客文 的所有 ，而不是 在10月 接收到的 。

嵌套聚合

在 的 候，我 使用 nested 就可以 取嵌套 象的信息。同理， nested 聚合允 我 嵌套 象里的字段 行聚合操作。

```

GET /my_index/blogpost/_search
{
    "size" : 0,
    "aggs": {
        "comments": { ①
            "nested": {
                "path": "comments"
            },
            "aggs": {
                "by_month": {
                    "date_histogram": { ②
                        "field":      "comments.date",
                        "interval":  "month",
                        "format":    "yyyy-MM"
                    },
                    "aggs": {
                        "avg_stars": {
                            "avg": { ③
                                "field": "comments.stars"
                            }
                        }
                    }
                }
            }
        }
    }
}

```

① nested 聚合 ' 入' 嵌套的 'comments' 象。

② comment 象根据 comments.date 字段的月 被分到不同的桶。

③ 算 个桶内star的平均数量。

从下面的 果可以看出聚合是在嵌套文 面 行的：

```

...
"aggregations": {
  "comments": {
    "doc_count": 4, ①
    "by_month": {
      "buckets": [
        {
          "key_as_string": "2014-09",
          "key": 1409529600000,
          "doc_count": 1, ①
          "avg_stars": {
            "value": 4
          }
        },
        {
          "key_as_string": "2014-10",
          "key": 1412121600000,
          "doc_count": 3, ①
          "avg_stars": {
            "value": 2.6666666666666665
          }
        }
      ]
    }
  }
}
...

```

① 共有4个 `comments` 象：1个 象在9月的桶里，3个 象在10月的桶里。

逆向嵌套聚合

`nested` 聚合 只能 嵌套文 的字段 行操作。 根文 或者其他嵌套文 的字段 它是不可 行的。 然而，通 `reverse_nested` 聚合，我 可以走出嵌套 ，回到父 文 行操作。

例如，我 要基于 者的年 出 者感 趣 `tags` 的分布。`comment.age` 是一个嵌套字段，但 `tags` 在根文 中：

```

GET /my_index/blogpost/_search
{
  "size" : 0,
  "aggs": {
    "comments": {
      "nested": { ①
        "path": "comments"
      },
      "aggs": {
        "age_group": {
          "histogram": { ②
            "field": "comments.age",
            "interval": 10
          },
          "aggs": {
            "blogposts": {
              "reverse_nested": {}, ③
              "aggs": {
                "tags": {
                  "terms": { ④
                    "field": "tags"
                  }
                }
              }
            }
          }
        }
      }
    }
  }
}

```

① nested 聚合 入 comments 象。

② histogram 聚合基于 comments.age 做分 , 10年一个分 。

③ reverse_nested 聚合退回根文 。

④ terms 聚合 算 个分 年 段的 者最常用的 。

略 果如下所示 :

```

...
"aggregations": {
  "comments": {
    "doc_count": 4, ①
    "age_group": {
      "buckets": [
        {
          "key": 20, ②
          "doc_count": 2, ②
          "blogposts": {
            "doc_count": 2, ③
            "tags": {
              "doc_count_error_upper_bound": 0,
              "buckets": [④
                { "key": "shares", "doc_count": 2 },
                { "key": "cash", "doc_count": 1 },
                { "key": "equities", "doc_count": 1 }
              ]
            }
          }
        },
      ],
    }
  },
}
...

```

① 一共有4条。

② 在20 到30 之 共有 条。

③ 些 包含在 篇博客文章中。

④ 在 些博客文章中最 的 是 **shares**、**cash**、**equities**。

嵌套 象的使用 机

嵌套 象 在只有一个主要 体 非常有用，一个主要 体包含有限个 密 但又不是很重要的 体，例如我 的 **blogpost** 象包含 象。 在基于 的内容 博客文章 ， **nested** 有很大的用，并且可以提供更快的 效率。

嵌套模型的 点如下：

- 当 嵌套文 做 加、修改或者 除 ，整个文 都要重新被索引。嵌套文 越多， 来的成本就越大。
- 果返回的是整个文 ，而不 是匹配的嵌套文 。尽管目前有 支持只返回根文 中最佳匹 配的嵌套文 ，但目前 不支持。

有 需要在主文 和其 体之 做一个完整的隔 。 个隔 是由父子 提供的。

父-子 系文

父-子 系文 在 上 似于 **nested model**：允 将一个 象 体和 外一个 象 体 起来。而 型的主要区 是：在 **nested objects** 文 中，所有 象都是在同一个文 中，而在父-子

系文 中，父 象和子 象都是完全独立的文 。

父-子 系的主要作用是允 把一个 type 的文 和 外一个 type 的文 起来，成一 多的系：一个父文 可以 多个子文 。与 nested objects 相比，父-子 系的主要 有：

- 更新父文 ， 不会重新索引子文 。
- 建，修改或 除子文 ， 不会影 父文 或其他子文 。 一点在 景下尤其有用：子文 数量 多，并且子文 建和修改的 率高 。
- 子文 可以作 搜索 果独立返回。

Elasticsearch 了一个父文 和子文 的映射 系，得益于 个映射，父-子文 操作非常快。但是 个映射也 父-子文 系有个限制条件：父文 和其所有子文 ，都必 要存 在同一个分片中。

父-子文 ID映射存 在 Doc Values 中。当映射完全在内存中 ， Doc Values 提供 映射的快速 理能力， 一方面当映射非常大 ，可以通 溢出到磁 提供足 的 展能力

父-子 系文 映射

建立父-子文 映射 系 只需要指定某一个文 type 是 一个文 type 的父 。 系可以在如下 个 点 置：1) 建索引 ；2) 在子文 type 建之前更新父文 的 mapping。

例 明，有一个公司在多个城市有分公司，并且 一个分公司下面都有很多 工。有 的需求：按照分 公司、 工的 度去搜索，并且把 工和他 工作的分公司 系起来。 需求，用嵌套模型是无法 的。当然，如果使用 application-side-joins 或者 data denormalization 也是可以 的，但是 了演示的目的，在 里我 使用父-子文 。

我 需要告 Elasticsearch，在 建 工 employee 文 type ， 指定分公司 branch 的文 type 其父 。

```
PUT /company
{
  "mappings": {
    "branch": {},
    "employee": {
      "_parent": {
        "type": "branch" ①
      }
    }
  }
}
```

① employee 文 是 branch 文 的子文 。

建父-子文 索引

父文 建索引与 普通文 建索引没有区 。父文 并不需要知道它有 些子文 。

```
POST /company/branch/_bulk
{ "index": { "_id": "london" }}
{ "name": "London Westminster", "city": "London", "country": "UK" }
{ "index": { "_id": "liverpool" }}
{ "name": "Liverpool Central", "city": "Liverpool", "country": "UK" }
{ "index": { "_id": "paris" }}
{ "name": "Champs Élysées", "city": "Paris", "country": "France" }
```

建子文，用必通 `parent` 参数来指定子文的父文 ID：

```
PUT /company/employee/1?parent=london ①
{
  "name": "Alice Smith",
  "dob": "1970-10-24",
  "hobby": "hiking"
}
```

① 当前 `employee` 文的父文 ID 是 `london`。

父文 ID 有个作用：建了父文和子文之系，并且保了父文和子文都在同一个分片上。

在 [路由一个文到一个分片中](#) 中，我解了 Elasticsearch 如何通路由来决定文属于一个分片，路由文的 `_id`。分片路由的算公式如下：

```
shard = hash(routing) % number_of_primary_shards
```

如果指定了父文的 ID，那就会使用父文的 ID 行路由，而不会使用当前文 `_id`。也就是，如果父文和子文都使用相同的行路由，那父文和子文都会定分布在同一个分片上。

在行文的求需要指定父文的 ID，文求包括：通 `GET` 求取一个子文；建、更新或除一个子文。而行搜索求是不需要指定父文的 ID，是因搜索求是向一个索引中的所有分片起求，而文的操作是只会向存文的分片送求。因此，如果操作个子文不指定父文的 ID，那很有可能会把求送到的分片上。

父文的 ID 在 `bulk` API 中指定

```
POST /company/employee/_bulk
{ "index": { "_id": 2, "parent": "london" }}
{ "name": "Mark Thomas", "dob": "1982-05-16", "hobby": "diving" }
{ "index": { "_id": 3, "parent": "liverpool" }}
{ "name": "Barry Smith", "dob": "1979-04-01", "hobby": "hiking" }
{ "index": { "_id": 4, "parent": "paris" }}
{ "name": "Adrien Grand", "dob": "1987-05-11", "hobby": "horses" }
```

WARNING

如果想要改一个子文的 `parent`，通过更新一个子文是不行的，因为新的父文有可能在外一个分片上。因此，必须要先把子文删除，然后再重新索引一个子文。

通 子文 父文

`has_child` 的 和 可以通过子文的内容来 父文。例如，我根据如下，找出所有80后 工所在的分公司：

```
GET /company/branch/_search
{
  "query": {
    "has_child": {
      "type": "employee",
      "query": {
        "range": {
          "dob": {
            "gte": "1980-01-01"
          }
        }
      }
    }
  }
}
```

似于 `nested query`，`has_child` 可以匹配多个子文，并且一个子文的 分都不同。但是由于一个子文都有 分，这些 分如何 成父文的 得分取决于 `score_mode` 一个参数。参数有多取策略：`none`，会忽略子文的 分，并且会父文 分置 `1.0`；除此以外可以置成 `avg`、`min`、`max` 和 `sum`。

下面的 将会同 返回 `london` 和 `liverpool`，不 由于 `Alice Smith` 要比 `Barry Smith` 更加匹配条件，因此 `london` 会得到一个更高的 分。

```
GET /company/branch/_search
{
  "query": {
    "has_child": {
      "type": "employee",
      "score_mode": "max",
      "query": {
        "match": {
          "name": "Alice Smith"
        }
      }
    }
  }
}
```

TIP `score_mode` 的 `none`，会著地比其模式要快，是因 Elasticsearch 不需要算一个子文的分。只有当真正需要关心分果，才需要 `source_mode`，例如成 `avg`、`min`、`max` 或 `sum`。

min_children 和 max_children

`has_child` 的和都可以接受一个参数：`min_children` 和 `max_children`。使用一个参数，只有当子文数量在指定内，才会返回父文。

如下只会返回至少有 个雇的分公司：

```
GET /company/branch/_search
{
  "query": {
    "has_child": {
      "type": "employee",
      "min_children": 2, ①
      "query": {
        "match_all": {}
      }
    }
  }
}
```

① 至少有 个雇的分公司才会符合 条件。

有 `min_children` 和 `max_children` 参数的 `has_child` 或，和允 分的 `has_child` 的性能非常接近。

has_child Filter

`has_child` 和 在 行机制上似，区是 `has_child` 不支持 `source_mode` 参数。`has_child` 用于 内容—如内部的一个 `filtered` —和其他 行似：包含或者排除，但没有 行 分。

`has_child` 的果没有被 存，但是 `has_child` 内部的 方法 用于通常的 存 。

通 父文 子文

然 `nested` 只能返回最 的文，但是父文 和子文 本身是彼此独立并且可被 独 的。我使用 `has_child` 句可以基于子文 来 父文，使用 `has_parent` 句可以基于父文 来 子文。

`has_parent` 和 `has_child` 非常相似，下面的 将会返回所有在 UK 工作的雇：

```
GET /company/employee/_search
{
  "query": {
    "has_parent": {
      "type": "branch", ①
      "query": {
        "match": {
          "country": "UK"
        }
      }
    }
  }
}
```

① 返回父文 type 是 branch 的所有子文

has_parent 也支持 score_mode 个参数，但是 参数只支持 : none () 和 score 。
个子文 都只有一个父文 ，因此 里不存在将多个 分 一个的情况， score_mode 的取
score 和 none 。

不 分的 has_parent

当 has_parent 用于非 分模式（比如 filter 句） ， score_mode
参数就不再起作用了。因 模式只是 地包含或排除文 ，没有 分，那 score_mode
参数也就没有意 了。

子文 聚合

在父-子文 中支持 子文 聚合， 一点和 嵌套聚合 似。但是， 于父文 的聚合 是不支持的（和 reverse_nested 似）。

我 通 下面的例子来演示按照国家 度 看最受雇 迎的 余 好：

```

GET /company/branch/_search
{
  "size" : 0,
  "aggs": {
    "country": {
      "terms": { ①
        "field": "country"
      },
      "aggs": {
        "employees": {
          "children": { ②
            "type": "employee"
          },
          "aggs": {
            "hobby": {
              "terms": { ③
                "field": "hobby"
              }
            }
          }
        }
      }
    }
  }
}

```

① country 是 branch 文 的一个字段。

② 子文 聚合 通 employee type 的子文 将其父文 聚合在一起。

③ hobby 是 employee 子文 的一个字段。

祖 与 系

父子 系可以延展到更多代 系，比如生活中 与祖 的 系 — 唯一的要求是 足 些 系的文 必 在同一个分片上被索引。

我 把上一个例子中的 country 型 定 branch 型的父：

```

PUT /company
{
  "mappings": {
    "country": {},
    "branch": {
      "_parent": {
        "type": "country" ①
      }
    },
    "employee": {
      "_parent": {
        "type": "branch" ②
      }
    }
  }
}

```

① branch 是 country 的子 。

② employee 是 branch 的子 。

country 和 branch 之 是一 的父子 系，所以我 的操作 与之前保持一致：

```

POST /company/country/_bulk
{ "index": { "_id": "uk" }}
{ "name": "UK" }
{ "index": { "_id": "france" }}
{ "name": "France" }

POST /company/branch/_bulk
{ "index": { "_id": "london", "parent": "uk" }}
{ "name": "London Westminster" }
{ "index": { "_id": "liverpool", "parent": "uk" }}
{ "name": "Liverpool Central" }
{ "index": { "_id": "paris", "parent": "france" }}
{ "name": "Champs Élysées" }

```

parent ID 使得 一个 branch 文 被路由到与其父文 country 相同的分片上 行操作。然而，当我 使用相同的方法来操作 employee 个 文 ，会 生什 ？

```

PUT /company/employee/1?parent=london
{
  "name": "Alice Smith",
  "dob": "1970-10-24",
  "hobby": "hiking"
}

```

employee 文 的路由依 其父文 ID — 也就是 <code>london</code> — 但是

<code>london</code> 文 的路由却依 其本身的 父文 ID — 也就是<code>uk</code> 。此 情况下， 文 很有可能最 和父 、祖 文 不在同一分片上， 致不足祖 和 文 必 在同一个分片上被索引的要求。

解决方案是添加一个 外的 `routing` 参数，将其 置 祖 的文 ID ，以此来保 三代文 路由到同一个分片上。索引 求如下所示：

```
PUT /company/employee/1?parent=london&routing=uk ①
{
  "name": "Alice Smith",
  "dob": "1970-10-24",
  "hobby": "hiking"
}
```

① `routing` 的 会取代 `parent` 的 作 路由 。

`parent` 参数的 然可以 `employee` 文 与其父文 的 系，但是 `routing` 参数保 文 被存 到其父 和祖 的分片上。`routing` 在所有的文 求中都要添加。

合多代文 行 和聚合是可行的，只需要一代代的 行 定即可。例如，我 要 到喜 足的雇 者的城市，此 需要 合 `country` 和 `branch`，以及 `branch` 和 `employee`：

```
GET /company/country/_search
{
  "query": {
    "has_child": {
      "type": "branch",
      "query": {
        "has_child": {
          "type": "employee",
          "query": {
            "match": {
              "hobby": "hiking"
            }
          }
        }
      }
    }
  }
}
```

使用中的一些建

当文 索引性能 比 性能重要的 时候，父子 系是非常有用的，但是它也是有巨大代 的。其 速度 会比同等的嵌套 慢5到10倍!

全局序号和延

父子 系使用了全局序数 来加速文 的 合。不管父子 系映射是否使用了内存 存或基于硬 的 doc values, 当索引 更 , 全局序数要重建。

一个分片中父文 越多, 那 全局序数的重建就需要更多的 。父子 系更 合于父文 少、子文 多的情况。

全局序数 情况下是延 建的: 在refresh后的第一个父子 会触 全局序数的 建。而 个 建会致用 使用 感受到明 的 。 可以使用全局序数 加 来将全局序数 建的 由query 段移到refresh 段, 置如下:

```
PUT /company
{
  "mappings": {
    "branch": {},
    "employee": {
      "_parent": {
        "type": "branch",
        "fielddata": {
          "loading": "eager_global_ordinals" ①
        }
      }
    }
  }
}
```

① 在一个新的段可搜索前, `_parent` 字段的全局序数会被 建。

当父文 多 , 全局序数的 建会耗 很多 。此 可以通 加 `refresh_interval` 来 少 refresh 的次数, 延 全局序数的有效 , 也很大程度上 小了全局序数 秒重建的cpu消耗。

多代使用和

多代文 的 合 (看祖 与 系) 然看起来很吸引人, 但必 考 如下的代 :

- 合越多, 性能越差。
- 一代的父文 都要将其字符串 型的 `_id` 字段存 在内存中, 会占用大量内存。

当 考 父子 系是否 合 有 系模型 , 考 下面 些建 :

- 尽量少地使用父子 系, 在子文 多于父文 使用。
- 避免在一个 中使用多个父子 合 句。
- 在 `has_child` 中使用 filter 上下文, 或者 置 `score_mode` `none` 来避免 算文 得分。
- 保 父 IDs 尽量短, 以便在 doc values 中更好地 , 被 占用更少的内存。

最重要的是: 先考 下我 之前 的其他方式来 到父子 系的效果。

容

一些公司 天使用 Elasticsearch 索引 索 PB 数据， 但我 中的大多数都起 于 模 的 目。即使我 立志成 下一个 Facebook， 我 的 行 余 却也跟不上梦想的脚 。 我 需要 今日所需而 建， 但也要允 我 可以 活而又快速地 行水平 展。

Elasticsearch 了可 展性而生。它可以良好地 行于 的 本 又或者一个 有数百 点的集群， 同 用 体 基本相同。 由小 模集群 大 模集群的 程几乎完全自 化并且无痛。由大 模集群 超大 模集群需要一些 和 ， 但 是相 地无痛。

当然 一切并不是魔法。Elasticsearch 也有它的局限性。如果 了解 些局限性并能 与之相 ， 集群 容的 程将会是愉快的。如果 Elasticsearch 理不当， 那 将 于一个充 痛苦的世界。

Elasticsearch 的 置会伴 走 很 的一段路， 但 了 它最大的效用， 需要考虑 数据是如何流 的系 的。 我 将 常 的数据流： 序数据（ 相 性， 例如日志或社交 数据流）， 以及基于用 的数据（ 有很大的文 集但可以按用 或客 分）。

一章将 助 在遇到不愉快之前做出正 的 。

容的 元

在 [更新索引](#)， 我 介 了一个分片即一个 *Lucene* 索引 ， 一个 Elasticsearch 索引即一系列分片的集合。 的 用程序与索引 行交互， Elasticsearch 助 将 求路由至相 的分片。

一个分片即 容的 元 。一个最小的索引 有一个分片。 可能已 完全 足 的需求了 一 个分片即可存 大量的数据—但 限制了 的可 展性。

想象一下我 的集群由一个 点 成，在集群内我 有一个索引， 个索引只含一个分片：

```
PUT /my_index
{
  "settings": {
    "number_of_shards": 1, ①
    "number_of_replicas": 0
  }
}
```

① 建一个 有 1 主分片 0 个副本分片的索引。

个 置 也 很小， 但它 足我 当前的需求而且 行代 低。

NOTE 当前我 只 主分片。我 将在 [副本分片](#) 副本 分片。

在美好的一天， 互 了我 ， 一个 点再也承受不了我 的流量。 我 决定根据 [一个只有一个分片的索引无 容因子](#) 添加一个 点。 将会 生什 ？

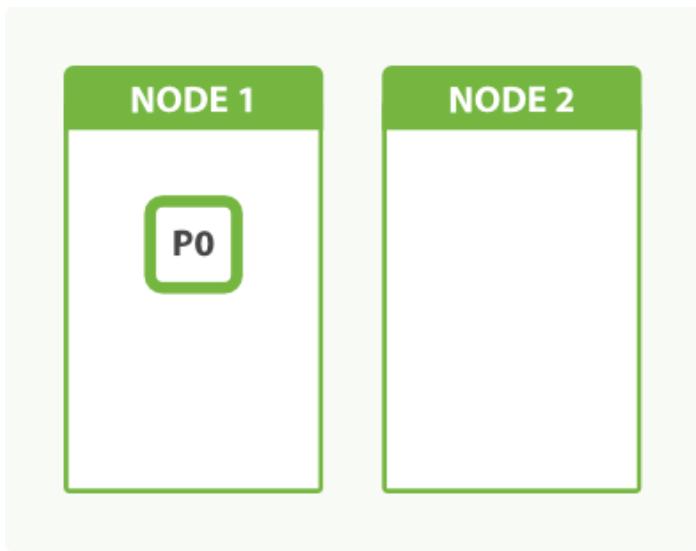


Figure 49. 一个只有一个分片的索引无 容因子

答案是：什 都不会 生。因 我 只有一个分片，已 没有什 可以放在第二个 点上的了。 我 不能 加索引的分片数因 它是 [route documents to shards](#) 算法中的重要元素：

```
shard = hash(routing) % number_of_primary_shards
```

我 当前的 只有一个就是将数据重新索引至一个 有更多分片的一个更大的索引，但 做将消耗的 是我 无法提供的。通 事先 ，我 可以使用 分配 的方式来完全避免 个 。

分片 分配

一个分片存在于 个 点，但一个 点可以持有多个分片。想象一下我 建 有 个主分片的索引而不是 一个：

```
PUT /my_index
{
  "settings": {
    "number_of_shards": 2, ①
    "number_of_replicas": 0
  }
}
```

① 建 有 个主分片无副本分片的索引。

当只有一个 点 ， 个分片都将被分配至相同的 点。 从我 用程序的角度来看，一切都和之前一 作着。 用程序和索引 行通 ， 而不是分片， 在 是只有一个索引。

，我 加入第二个 点，Elasticsearch 会自 将其中一个分片移 至第二个 点，如 一个 有 个分片的索引可以利用第二个 点 描 的那 ， 当重新分配完成后， 个分片都将接近至 倍于之前的 算能力。

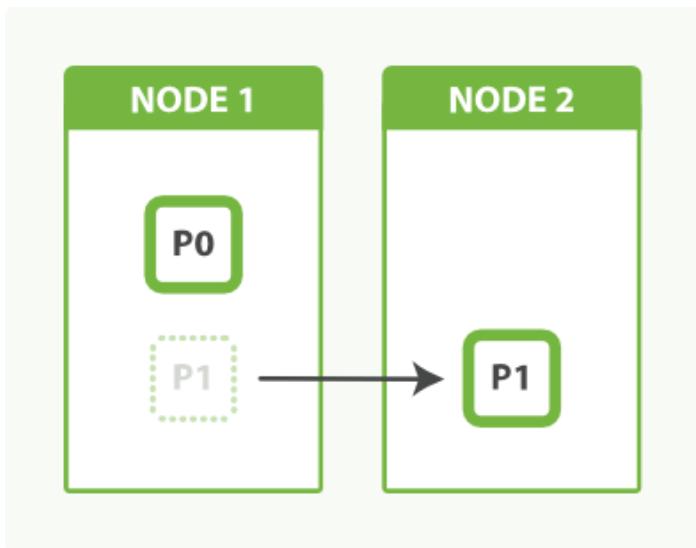


Figure 50. 一个有 个分片的索引可以利用第二个点

我已可以通 地将一个分片通 制到一个新的 点来加倍我 的理能力。最棒的是，我 零停机地做到了 一点。在分片移 程中，所有的索引搜索 求均在正常 行。

在 Elasticsearch 中新添加的索引 被指定了五个主分片。 意味着我 最多可以将那个索引分散到五个 点上， 个 点一个分片。 它具有很高的 理能力， 未等 去思考 一切就已 做到了！

分片分裂

用 常在 ， 什 Elasticsearch 不支持 分片分裂 (shard-splitting) —; 将 个分片分裂 个或更多部分的能力。原因就是分片分裂是一个糟 的想法：

- 分裂一个分片几乎等于重新索引 的数据。它是一个比 将分片从一个 点 制到 一个 点 更重量 的操作。
- 分裂是指数的。起初 有一个分片，然后分裂 个，然后四个，八个，十六个，等等。分裂 并不会 好地把 的 理能力提升 50%。
- 分片分裂需要 有足 的能力支 一 索引的拷 。通常来 ，当 意 到 需要横向 展 ， 已 没有足 的剩余空 来做分裂了。

Elasticsearch 通 一 方式来支持分片分裂。 是可以把 的数据重新索引至一个 有 当分片个数的新索引 (参 [重新索引 的数据](#))。 和移 分片比起来 依然是一个更加密集的操作，依然需要足 的剩余空 来完成，但至少 可以控制新索引的分片个 数了。

海量分片

当新手 在了解 分片 分配 之后做的第一件事就是 自己 :

我不知道 个索引将来会 得多大，并且 后我也不能更改索引的大小，所以 了 保 起 ， 是 它 1000 个分片 ...

—一个新手的

一千个分片——当真？在来一千个点之前，不得不可能需要再三思考的数据模型然后将它重新索引？

一个分片并不是没有代价的。记住：

- 一个分片的代价即一个Lucene索引，会消耗一定文件句柄、内存、以及CPU。
- 一个搜索请求都需要命中索引中的一个分片，如果一个分片都位于不同的点好，但如果多个分片都需要在同一个点上争使用相同的源就有些糟了。
- 用于算相度的信息是基于分片的。如果有多个分片，一个都只有很少的数据会致很低的相度。

TIP

适当的分配是好的。但上千个分片就有些糟。我很去定分片是否多了，取决于它的大小以及如何去使用它。一百个分片但很少使用好，一个分片但非常繁地使用有可能就有点多了。控制的点保证它留有足够的空源来理一些特殊情况。

横向扩展当分段行。下一段准备好足够的源。只有当进入到下一个段，才有思考需要作出些改来到个段。

容量

如果一个分片太少而1000个又太多，那我只知道我需要多少分片？一般情况下是一个无法回答的。因为在有太多相关的因素了：使用的硬件、文档的大小和密度、文档的索引分析方式、行的类型、行的聚合以及的数据模型等等。

幸的是，在特定景下是一个容易回答的，尤其是自己的景：

1. 基于准确用于生产环境的硬件建一个有N个点的集群。
2. 建一个和准确用于生产环境相同配置和分析器的索引，但它只有一个主分片无副本分片。
3. 索引的文档（或者尽可能接近）。
4. 行的和聚合（或者尽可能接近）。

基本来，需要制真环境的使用方式并将它全部放到一个分片上直到它挂掉。“上挂掉的定位也取决于：一些用需要所有在50秒内返回；一些大于等于5秒。

一旦定好了个分片的容量，很容易就可以推算出整个索引的分片数。用需要索引的数据数加上一部分预期的，除以个分片的容量，结果就是需要的主分片个数。

TIP

容量不当作的第一。

先看看有没有办法化Elasticsearch的使用方式。也有低效的，少足的内存，又或者满了swap？

我有一些新手于初始性能感到沮丧，立即就着手回收或者是进程数，而不是清理例如去掉通配符。

副本分片

目前 止我 只 主分片，但我 身 有 一个工具：副本分片。 副本分片的主要目的就是了故障 移，正如在 集群内的原理 中 的：如果持有主分片的 点挂掉了，一个副本分片就会晋升主分片的角色。

在索引写入，副本分片做着与主分片相同的工作。新文 首先被索引 主分片然后再同 到其它所有的副本分片。 加副本数并不会 加索引容量。

无 如何，副本分片可以服 于 求，如果 的索引也如常 的那 是偏向 使用的，那 可以通加副本的数目来提升 性能，但也要 此 加 外的硬件 源。

我 回到那个有着 个主分片索引的例子。我通 加第二个 点来提升索引容量。 加 外的点不会 助我 提升索引写入能力，但我 可以通 加副本数在搜索 利用 外的硬件：

```
PUT /my_index/_settings
{
  "number_of_replicas": 1
}
```

有 个主分片，加上 个主分片的一个副本， 共 予我 四个分片： 个 点一个，如 所示 一个有 个主分片一 副本的索引可以在四个 点中横向 展。

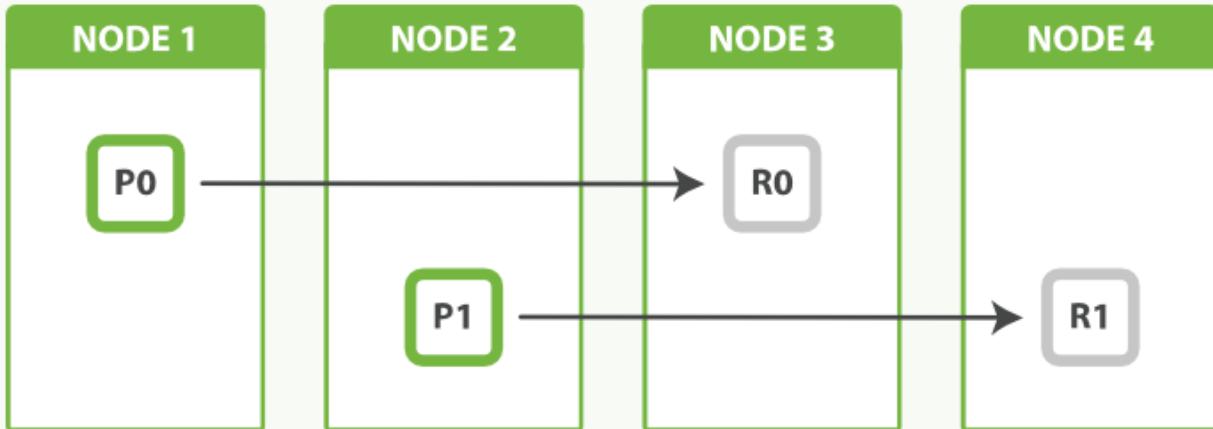


Figure 51. 一个 有 个主分片一 副本的索引可以在四个 点中横向 展

通 副本 行 均衡

搜索性能取决于最慢的 点的 ，所以 均衡所有 点的 是一个好想法。 如果我 只是加一个 点而不是 个，最 我 会有 个 点各持有一个分片，而 一个持有 个分片做着 倍的工作。

我 可以通 整副本数量来平衡 些。通 分配 副本而不是一个，最 我 会 有六个分片， 好可以平均分 三个 点，如 所示 通 整副本数来均衡 点 ：

```

PUT /my_index/_settings
{
  "number_of_replicas": 2
}

```

作 励，我 同 提升了我 的可用性。我 可以容忍 失 个 点而 然保持一 完整数据的拷 。

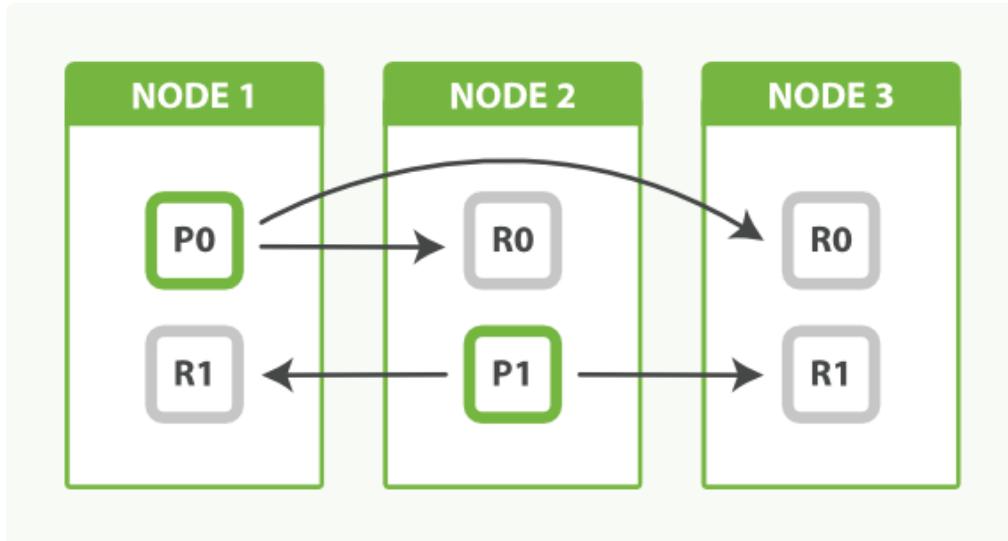


Figure 52. 通 整副本数来均衡 点

NOTE 事 上 点 3 持有
一个副本分片，然而没有主分片并不重要。副本分片与主分片做着相同的工作；它 只是扮演着略微不同的角色。没有必要 保主分片均 地分布在所有 点中。

多索引

最后，住没有任何 限制 的 用程序只使用一个索引。 当我 起一个搜索 求 ，它被 至索引中 个分片的一 拷 （一个主分片或一个副本分片），如果我 向多个索引 出同 的 求， 会 生完全相同的事情——只不 会 及更多的分片。

TIP 搜索 1 个有着 50 个分片的索引与搜索 50 个 个都有 1 个分片的索引完全等 ；搜索 求均命中 50 个分片。

当 需要在不停服 的情况下 加容量 ，下面有一些有用的建 。相 于将数据 移到更大的索引中， 可以 做下面 些操作：

- 建一个新的索引来存 新的数据。
- 同 搜索 个索引来 取新数据和旧数据。

上，通 一点 先 ，添加一个新索引可以通 一 完全透明的方式完成， 的 用程序根本不会察 到任何的改 。

在 索引 名和零停机，我 提到 使用索引 名来指向当前版本的索引。 例来 ， 的索引命名 tweets_v1 而不是 tweets 。 的 用程序会与 tweets 行交互，但事 上它是一个指向 tweets_v1 的 名。 允 将 名切 至一个更新版本的索引而保持服 。

我 可以使用一个 似的技 通 加一个新索引来 展容量。 需要一点点 ， 因 需要 个 名： 一个用于搜索 一个用于索引数据：

```
PUT /tweets_1/_alias/tweets_search ①  
PUT /tweets_1/_alias/tweets_index ①
```

① `tweets_search` 与 `tweets_index` 个 名都指向索引 `tweets_1`。

新文 当索引至 `tweets_index`，同，搜索 求 当 名 `tweets_search` 出。目前， 个 名指向同一个索引。

当我 需要 外容量 ， 我 可以 建一个名 `tweets_2` 的索引，并且像 更新 名：

```
POST /_aliases  
{  
  "actions": [  
    { "add": { "index": "tweets_2", "alias": "tweets_search" }}, ①  
    { "remove": { "index": "tweets_1", "alias": "tweets_index" }}, ②  
    { "add": { "index": "tweets_2", "alias": "tweets_index" }} ②  
  ]  
}
```

① 添加索引 `tweets_2` 到 名 `tweets_search`。

② 将 名 `tweets_index` 由 `tweets_1` 切 至 `tweets_2`。

一个搜索 求可以以多个索引 目，所以将搜索 名指向 `tweets_1` 以及 `tweets_2` 是完全有效的。 然而，索引写入 求只能以 个索引 目。因此，我 必 将索引写入的 名只指向新的索引。

TIP 一个文 `GET` 求，像一个索引写入 求那，只能以 个索引 目。 致在通 ID 取文 的 景下有一点 。作 代替， 可以 `tweets_1` 以及 `tweets_2` 行一个 `ids` 搜索 求，或者 `multi-get` 求。

在服 行中使用多索引来 展索引容量 于一些使用 景有着特 的好 ，像我 将在下一 中 的基 于 的数据例如日志或社交事件流。

基于 的数据

Elasticsearch 的常用案例之一便是日志 ， 它 在太常 了以至于 Elasticsearch 提供了一个集成的日志平台叫做 `ELK stack` Elasticsearch, Logstash, 以及 Kibana ——来 工作 得 。

`Logstash` 采集、解析日志并在将它 写入Elasticsearch之前格式化。 `Elasticsearch` 扮演了一个集中式的日志服 角色， `Kibana` 是一个 形化前端可以很容易地 的 化。

搜索引 中大多数使用 景都是 慢相 定的文 集合。搜索 最相 的文 ，而不 心它是何 建的。

日志——以及其他基于 的数据流例如社交 活 —— 上有很大不同。 索引中文 数量迅速 ，通常随 加速。 文 几乎不会更新，基本以最近文 搜索目 。随着 推移，文 逐失去 。

我 需要 整索引 使其能 工作于 基于 的数据流。

按 索引

如果我 此 型的文 建立一个超大索引，我 可能会很快耗尽存 空 。日志事件会不断的 来，不 会停 也不会中断。 我 可以使用 scroll 和批量 除来 除旧的事件。但 方法 非常低效 。当 除一个文 ，它只会被 被 除(参 除和更新)。 在包含它的段被合并之前不会被物理 除。

替代方案是，我 使用一个 索引。 可以着手于一个按年的索引 (logs_2014) 或按月的索引 (logs_2014-10) 。也 当 的 得十分繁忙 ， 需要切 到一个按天的索引 (logs_2014-10-24) 。 除旧数据十分 :只需要 除旧的索引。

方法有 的 点，允 在需要的 时候 行 容。 不需要 先做任何 的决定。 天都是一个新的机会来 整 的索引 来 当前需求。 用相同的 到决定 个索引的大小上。起初也 需要的 是 周一个主分片。 一 子，也 需要 天五个主分片。 都不重要——任何 都 可以 整到新的 境。

名可以 助我 更加透明地在索引 切 。 当 建索引 ， 可以将 logs_current 指向当前索引来接收新的日志事件，当 索 ， 更新 last_3_months 来指向所有最近三个月的索引：

```
POST /_aliases
{
  "actions": [
    { "add": { "alias": "logs_current", "index": "logs_2014-10" }}, ①
    { "remove": { "alias": "logs_current", "index": "logs_2014-09" }}, ①
    { "add": { "alias": "last_3_months", "index": "logs_2014-10" }}, ②
    { "remove": { "alias": "last_3_months", "index": "logs_2014-07" }} ②
  ]
}
```

① 将 logs_current 由九月切 至十月。

② 将十月添加到 last_3_months 并且 剔掉七月。

索引模板

Elasticsearch 不要求 在使用一个索引前 建它。 于日志 用，依 于自 建索引比手 建要更加方便。

Logstash 使用事件中的 来生成索引名。 天被索引至不同的索引中，因此一个 @timestamp 2014-10-01 00:00:01 的事件将被 送至索引 logstash-2014.10.01 中。 如果那个索引不存在，它将被自 建。

通常我 想要控制一些新建索引的 置 (settings) 和映射 (mappings)。也 我 想要限制分片数 1，并且禁用 _all 域。索引模板可以用于控制何 置 (settings) 当被 用于新 建的索引：

```

PUT /_template/my_logs ①
{
  "template": "logstash-*", ②
  "order": 1, ③
  "settings": {
    "number_of_shards": 1 ④
  },
  "mappings": {
    "_default_": { ⑤
      "_all": {
        "enabled": false
      }
    }
  },
  "aliases": {
    "last_3_months": {} ⑥
  }
}

```

① 建一个名 `my_logs` 的模板。

② 将 个模板 用于所有以 `logstash-` 起始的索引。

③ 个模板将会覆 的 `logstash` 模板，因 模板的 `order` 更低。

④ 限制主分片数量 1。

⑤ 所有 型禁用 `_all` 域。

⑥ 添加 个索引至 `last_3_months` 名中。

个模板指定了所有名字以 `logstash-` 起始的索引的 置，不 它是手 是自 建的。如果我 明天的索引需要比今天更大的容量，我 可以更新 个索引以使用更多的分片。

个模板 将新建索引添加至了 `last_3_months` 名中，然而从那个 名中 除旧的索引 需要手 行。

数据 期

随着 推移，基于 数据的相 度逐 降低。 有可能我 会想要 看上周、上个月甚至上一年度 生了什 ，但是大多数情况，我 只 心当前 生的。

按 索引 来的一个好 是可以方便地 除旧数据：只需要 除那些 得不重要的索引就可以了。

```
DELETE /logs_2013*
```

除整个索引比 除 个文 要更加高效：Elasticsearch 只需要 除整个文件 。

但是 除索引是 手段。在我 决定完全 除它之前 有一些事情可以做来 助数据更加 雅地 期。

移旧索引

随着数据被，很有可能存在一个点索引——今日的索引。所有新文都会被加到那个索引，几乎所有都以它目。那个索引当使用最好的硬件。

Elasticsearch是如何得知台是最好的服务器？可以通台服务器指定任意的来告它。例如，可以像一个点：

```
./bin/elasticsearch --node.box_type strong
```

`box_type`参数是完全随意的——可以将它随意命名只要喜——但可以用些任意的来告Elasticsearch将一个索引分配至何。

我可以通过按以下配置建今日的索引来保它被分配到我最好的服务器上：

```
PUT /logs_2014-10-01
{
  "settings": {
    "index.routing.allocation.include.box_type" : "strong"
  }
}
```

昨日的索引不再需要我最好的服务器了，我可以通更新索引置将它移到`medium`的点上：

```
POST /logs_2014-09-30/_settings
{
  "index.routing.allocation.include.box_type" : "medium"
}
```

索引化(Optimize)

昨日的索引不大可能会改。日志事件是静的：已生的往不会再改了。如果我将一个分片合并至一个段(Segment)，它会占用更少的源更快地。我可以通[optimize API](#)来做到。

分配在`strong`主机上的索引行化(Optimize)操作将会是一个糟的想法，因化操作将消耗点上大量I/O并索引今日日志造成冲。但是`medium`的点没有做太多似的工作，我可以安全地在上面行化。

昨日的索引有可能有副本分片。如果我下一个化(Optimize)求，它会化主分片和副本分片，有些浪。然而，我可以移除副本分片，行化，然后再恢副本分片：

```
POST /logs_2014-09-30/_settings
{ "number_of_replicas": 0 }

POST /logs_2014-09-30/_optimize?max_num_segments=1

POST /logs_2014-09-30/_settings
{ "number_of_replicas": 1 }
```

当然，没有副本我 将面 磁 故障而 致 失数据的 。 可能想要先通 <https://www.elastic.co/guide/en/elasticsearch/reference/master/modules-snapshots.html>[snapshot-restore API] 数据。

旧索引

当索引 得更“老”，它 到 一个几乎不会再被 的 点。 我 可以在 个 段 除它 ，但也要想将它 留在 里以防万一有人在半年后 想要 它 。

些索引可以被 。它 会存在于集群中，但它 不会消耗磁 空 以外的 源。重新打 一个索引要比从 中恢 快得多。

在 之前， 得我 去刷写索引来 保没有事 残留在事 日志中。一个空白的事 日志会使得索引在重新打 恢 得更快：

```
POST /logs_2014-01-*/_flush ①
POST /logs_2014-01-*/_close ②
POST /logs_2014-01-*/_open ③
```

① 刷写 (Flush) 所有一月的索引来清空事 日志。

② 所有一月的索引。

③ 当 需要再次 它 ， 使用 open API 来重新打 它 。

旧索引

最后，非常旧的索引可以通 <https://www.elastic.co/guide/en/elasticsearch/reference/master/modules-snapshots.html>[snapshot-restore API] 至 期存 例如共享磁 或者 Amazon S3，以防日后可能需要 它 。当存在 我 就可以将索引从集群中 除了。

基于用 的数据

通常来 ，用 使用 Elasticsearch 的原因是它 需要添加全文 索或者需要分析一个已 存在的 用。他 建一个索引来存 所有文 。公司里的其他人也逐 了 Elasticsearch 来的好 ，也想把他 的数据添加到 Elasticsearch 中去。

幸 的是，Elasticsearch 支持<http://en.wikipedia.org/wiki/Multitenancy>[多租]所以 个用 可以在相同的集群中 有自己的索引。 有人偶 会想要搜索所有用 的文 ， 情况可以通 搜索所有索引 ，但大多数情况下用 只 心它 自己的文 。

一些用 有着比其他人更多的文 ，一些用 可能有比其他人更多的搜索次数， 所以 指定

个索引主分片和副本分片数量能力的需要

很合使用“一个用一个索引”的模式。

似地，

繁忙的索引可以通 分片分配 指定到高配的 点。（参 [移旧索引](#)。）

TIP

不要 个索引都使用 的主分片数。想想看它需要存 多少数据。有可能 需要一个分片——再多的都只是浪 源。

大多数 Elasticsearch 的用 到 里就已 了。 的“一个用 一个索引” 大多数 景都可以 足了。

于例外的 景， 可能会 需要支持很大数量的用 ， 都是相似的需求。一个例子可能是 一个 有几千个 箱 的 提供搜索服 。 一些 可能有巨大的流量，但大多数都很小。将一个有着 个分片的索引用于一个小 模 已 是足 的了——一个分片可以承 很多个 的数据。

我 需要的是一 可以在用 共享 源的方法， 个用 他 有自己的索引 印象，而在小用 上浪 源。

共享索引

我 可以 多的小 使用一个大的共享的索引， 将 索引 一个字段并且将它用作一个 器：

```
PUT /forums
{
  "settings": {
    "number_of_shards": 10 ①
  },
  "mappings": {
    "post": {
      "properties": {
        "forum_id": { ②
          "type": "string",
          "index": "not_analyzed"
        }
      }
    }
  }
}
```

```
PUT /forums/post/1
{
  "forum_id": "baking", ②
  "title": "Easy recipe for ginger nuts",
  ...
}
```

① 建一个足 大的索引来存 数千个小 的数据。

② 个帖子都必 包含一个 `forum_id` 来 它属于 个 。

我 可以把 `forum_id` 用作一个 器来 个 行搜索。 个 器可以排除索引中 大部分的数据（属于其它 的数据）， 存会保 快速的 ！

```

GET /forums/post/_search
{
  "query": {
    "bool": {
      "must": {
        "match": {
          "title": "ginger nuts"
        }
      },
      "filter": {
        "term": {
          "forum_id": {
            "baking"
          }
        }
      }
    }
  }
}

```

个 法行得通，但我 可以做得更好。 来自于同一个 的帖子可以 地容 于 个分片，但它 在被打散到了 个索引的所有十个分片中。 意味着 个搜索 求都必 被 至所有十个分片的一个主分片或者副本分片。 如果能 保 所有来自于同一个 的所有帖子都被存 于同一个分片可能会是个好想法。

在 [路由一个文 到一个分片中](#)，我 一个文 将通 使用如下公式来分配到一个指定分片：

```
shard = hash(routing) % number_of_primary_shards
```

`routing` 的 文 的 `_id`，但我 可以覆 它并且提供我 自己自定 的路由 ，例如 `forum_id`。 所有有着相同 `routing` 的文 都将被存 于相同的分片：

```

PUT /forums/post/1?routing=baking ①
{
  "forum_id": "baking", ①
  "title": "Easy recipe for ginger nuts",
  ...
}

```

① 将 `forum_id` 用于路由 保 所有来自相同 的帖子都存 于相同的分片。

当我 搜索一个指定 的帖子 ，我 可以 相同的 `routing` 来保 搜索 求 在存有我 文 的分片上 行：

```

GET /forums/post/_search?routing=baking ①
{
  "query": {
    "bool": {
      "must": {
        "match": {
          "title": "ginger nuts"
        }
      },
      "filter": {
        "term": { ②
          "forum_id": {
            "baking"
          }
        }
      }
    }
  }
}

```

① 求 在 于 `routing` 的分片上 行。

② 我 是需要 (Filter) ， 因 一个分片可以存 来自于很多 的帖子。

多个 可以通 一个逗号分隔的列表来指定 `routing` ， 然后将 个 `forum_id` 包含于一个 `terms`

```

GET /forums/post/_search?routing=baking,cooking,recipes
{
  "query": {
    "bool": {
      "must": {
        "match": {
          "title": "ginger nuts"
        }
      },
      "filter": {
        "terms": {
          "forum_id": [
            "baking", "cooking", "recipes"
          ]
        }
      }
    }
  }
}

```

方式从技 上来 比 高效，由于要 一个 或者索引 求指定 `routing` 和 `terms` 的
看起来有一点的 拙。索引 名可以 解决 些！

利用名 一个用 一个索引

了保持 的，我 想 我 的 用 我 个用 都有一个 的索引——或者按照我 的例
子 个 一个——尽管 上我 用的是一个大的 `shared index`。因此，我 需要一 方式将 `routing`
及 器 含于 `forum_id` 中。

索引 名可以 做到 些。当 将一个 名与一个索引 起来， 可以指定一个 器和一个路由：

```
PUT /forums/_alias/baking
{
  "routing": "baking",
  "filter": {
    "term": {
      "forum_id": "baking"
    }
  }
}
```

在我 可以将 `baking` 名 一个 独的索引。索引至 `baking` 名的文 会自 地 用我 自定
的路由：

```
PUT /baking/post/1 ①
{
  "forum_id": "baking", ①
  "title": "Easy recipe for ginger nuts",
  ...
}
```

① 我 是需要 器指定 `forum_id` 字段，但自定 路由 已 是 含的了。

`baking` 名上的 只会在自定 路由 的分片上 行，并且 果也自 按照我 指定的 器
行了：

```
GET /baking/post/_search
{
  "query": {
    "match": {
      "title": "ginger nuts"
    }
  }
}
```

当 多个 行搜索 可以指定多个 名：

```
GET /baking_recipes/post/_search ①
{
  "query": {
    "match": {
      "title": "ginger nuts"
    }
  }
}
```

① 一个 `routing` 的都会用，返回结果会匹配任意一个器。

一个大的用

大模流行都是从小起的。有一天我会我共享索引中的一个分片要比其它分片更加繁忙，因一个分片中一个的文得更加。那个需要属于它自己的索引。

我用来提供一个用一个索引的索引名了我一个的移方式。

第一就是那个建一个新的索引，并其分配合理的分片数，可以足一定期的数据：

```
PUT /baking_v1
{
  "settings": {
    "number_of_shards": 3
  }
}
```

第二就是将共享的索引中的数据移到用的索引中，可以通 `scroll` 和 `bulk` API 来。当移完成，可以更新索引名指向那个新的索引：

```
POST /_aliases
{
  "actions": [
    { "remove": { "alias": "baking", "index": "forums" } },
    { "add": { "alias": "baking", "index": "baking_v1" } }
  ]
}
```

更新索引名的操作是原子性的；就像在一个。的应用程序是在与 `baking` API 交互并且于它已指向一个用的索引无感知。

用的索引不再需要器或者自定的路由了。我可以依于 Elasticsearch 使用的 `_id` 字段来做分区。

最后一是从共享的索引中除旧的文，可以通搜索之前的路由以及 ID 然后行批量除操作来。

一个用一个索引模型的雅之在于它允少源消耗，保持快速的，同有在需要零宕机容的能力。

容并不是无限的

整个章我了多Elasticsearch可以做到的容方式。大多数的容可以通过添加点来解决。但有一源是有限制的，因此得我真待：集群状。

集群状是一数据，存下列集群的信息：

- 集群的置
- 集群中的点
- 索引以及它的置、映射、分析器、器(Warmers)和名
- 与一个索引的分片以及它分配到的点

可以通过如下求看当前的集群状：

```
GET /_cluster/state
```

集群状存在于集群中的个点，包括客端点。就是什任何一个点都可以将求直接至被求数据的点——个点都知道个文在里。

只有主点被允更新集群状。想象一下一个索引求引入了一个之前未知的字段。持有那个文的主分片所在的点必将其新的映射到主点上。主点把更改合并到集群状中，然后向所有集群中的所有点宣布一个新的版本。

搜索求使用集群状，但它不会生修改。同，文的改求也不会集群状生修改。当然，除非它引入了一个需要更新映射的新的字段了。的来，集群状是静的不会成瓶。

然而，需要住的是相同的数据需要在个点的内存中保存，并且当它生更改必布到一个点。集群状的数据量越大，个操作就会越久。

我最常的集群状就是引入了太多的字段。一个用可能会决定一个IP地址或者个refererURL使用一个独的字段。下面个例子通一个唯一的referer使用一个不同的字段名来保持面量的数：

```
POST /counters/pageview/home_page/_update
{
  "script": "ctx._source[referer]++;
  "params": {
    "referer": "http://www.foo.com/links?bar=baz"
  }
}
```

方式十分的糟！它会生成数百万个字段，些都需要被存 在集群状 中。当到一个新的

referer , 都有一个新的字段需要加入那个已 膨 的集群状 中， 都需要被 布到集群的 个 点中去。

更好的方式是使用nested objects， 它使用一个字段作 参数名`referer`， 一个字段作 的 `count`：

```
"counters": [
  { "referer": "http://www.foo.com/links?bar=baz", "count": 2 },
  { "referer": "http://www.linkbait.com/article_3", "count": 10 },
  ...
]
```

嵌套的方式有可能会 加文 数量，但 Elasticsearch 生来就是 了解决它的。重要的是保持集群状 小而敏捷。

最 ， 不管 的初衷有多好， 可能会 集群 点数量、索引、映射 于一个集群来 是太大了。 此 ， 可能有必要将 个 拆分到多个集群中了。感 [https://www.elastic.co/guide/en/elasticsearch/reference/master/modules-tribe.html\[tribe nodes\]](https://www.elastic.co/guide/en/elasticsearch/reference/master/modules-tribe.html[tribe nodes])， 甚至可以向多个集群 出搜索 求，就好像我 有一个巨大的集群那 。

管理、 控和部署

本 大部分介 了使用 Elasticsearch 作 后端 建 用程序。本章 微不同。在 里， 将学 到如何管理 Elasticsearch 自身。Elasticsearch 是一个 的 件，有 多可移 件，大量的 API 用来 助管理 的 Elasticsearch 部署。

在 个章 ， 我 涵 三个主 ：

- 根据 控 的集群重要数据的 ， 去了解 些行 是正常的， 些 引起警告，并解 Elasticsearch 提供的各 信息。
- 部署 的集群到生 境，包括最佳 践和 （或不 ！）修改的重要配置。
- 部署后的 ， 如 Rolling Restart 或 的集群

控

Elasticsearch 常以多 点集群的方式部署。有多 API 可以管理和 控集群本身，而不用和集群里存 的数据打交道。

和 Elasticsearch 里 大多数功能一 ，我 有一个 体的 目，即任 通 API 行，而不是通 修改静 的配置文件。 一点在 的集群 容 尤 重要。即便通 配置管理系 （比如 Puppet, Chef 或者 Ansible），一个 的 HTTP API 用，也比往上百台物理上推送新配置文件 多了。

因此，本章将介 各 可以 整、 和 配集群的 API。同 ， 会介 一系列提供集群自身 数据的 API， 可以用 些接口来 控集群健康状 和性能。

Marvel 控

Marvel 可以很 的通 Kibana 控 Elasticsearch。可以 看 的集群健康状 和性能，也可以分析 去的集群、索引和 点指 。

然 可以通 本章介 的 API 看大量的指 数据，但是它 展示的都是当前 点的即 情况。了解 个瞬 的内存占用比当然很有用，但是了解内存占用比 随 的 更加有用。Marvel 会 并聚合 些数据， 可以通 可 化效果看到自己集群随 的 化， 可以很容易的 展的 。

随着 集群 模的 展， API 的 出内容会 得 人完全没法看。当 有一大把 点，比如 一百个，再 一个 出的 JSON 就非常乏味了。而 Marvel 可以 交互式的探索 些数据，更容易于集中 注特定 点或者索引上 生了什 。

Marvel 使用公 的 API，和 自己能 到的一 —— 它没有暴露任何 通 API 不到的 信息。但是，Marvel 大的 化了 些 信息的采集和可 化工作。

Marvel 可以免 使用（包括生 境上！），所以 在就 始用起来 ！安装介 ，参 [Marvel 入](#) 。

集群健康

一个 Elasticsearch 集群至少包括一个 点和一个索引。或者它可能有一百个数据 点、三个 独的主 点，以及一小打客 端 点—— 些共同操作一千个索引（以及上万个分片）。

不管集群 展到多大 模， 都会想要一个快速 取集群状 的途径。[Cluster Health API](#) 充当的就是 个角色。 可以把它想象成是在一万英尺的高度 瞰集群。它可以告 安心 一切都好，或者警告 集群某个地方有 。

我 行一下 `cluster-health` API 然后看看 体是什 子的：

```
GET _cluster/health
```

和 Elasticsearch 里其他 API 一， `cluster-health` 会返回一个 JSON 。 自 化和告警系 来，非常便于解析。 中包含了和 集群有 的一些 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
}
```

信息中最重要的就是 `status` 字段。状态可能是下列三个之一：

green

所有的主分片和副本分片都已分配。集群是 100% 可用的。

yellow

所有的主分片已分片了，但至少有一个副本是失的。不会有数据失，所以搜索结果依然是完整的。不过，高的高可用性在某种程度上被弱化。如果更多的分片消失，就会丢失数据了。把 `yellow` 想象成一个需要及的警告。

red

至少一个主分片（以及它的全部副本）都在失中。意味着在少数据：搜索只能返回部分数据，而分配到一个分片上的写入请求会返回一个常。

`green/yellow/red` 状态是一个概的集群并了解眼下正在生什的好法。剩下的指列出来集群的状态概要：

- `number_of_nodes` 和 `number_of_data_nodes` 个命名完全是自描述的。
- `active_primary_shards` 指出集群中的主分片数量。是涵了所有索引的。
- `active_shards` 是涵了所有索引的所有分片的，即包括副本分片。
- `relocating_shards` 表示目前正在从一个点往其他点的分片的数量。通常来是 0，不 在 Elasticsearch 集群不太均衡，会上。比如：添加了一个新点，或者下了一个点。
- `initializing_shards` 是新建的分片的个数。比如，当新建第一个索引，分片都会短的于 `initializing` 状态。通常会是一个事件，分片不期停留在 `initializing` 状态。可能在点重的时候看到 `initializing` 分片：当分片从磁上加后，它会从 `initializing` 状态开始。
- `unassigned_shards` 是已在集群状态中存在的分片，但是不在集群里又不着。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，在点集群上，就会有 5 个未分配副本分片。如果集群是 red 状态，也会期保有未分配分片（因少主分片）。

更深点：到索引

想象一下某天到到了，而的集群健康状态看起来像是：

```
{  
  "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 个健康状 里我 能推断出什 来？ ，我 集群是 red ，意味着我 数据（主分片 + 副本分片）了。我 知道我 集群原先有 10 个 点，但是在 个健康状 里列出来的只有 8 个数据 点。有 个数据 点不 了。我 看到有 20 个未分配分片。

就是我 能收集到的全部信息。那些 失分片的情况依然是个 。我 是 了 20 个索引， 个索引里少 1 个主分片？ 是 1 个索引里的 20 个主分片？ 是 10 个索引里的各 1 主 1 副本分片？具体是 个索引？

要回答 个 ，我 需要使用 level 参数 cluster-health 答出更多一点的信息：

```
GET _cluster/health?level=indices
```

个参数会 cluster-health API 在我 的集群信息里添加一个索引清 ，以及有 个索引的 （状 、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", ①
      "number_of_shards": 10,
      "number_of_replicas": 1,
      "active_primary_shards": 0,
      "active_shards": 0,
      "relocating_shards": 0,
      "initializing_shards": 0,
      "unassigned_shards": 20 ②
    },
    "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
    },
    ....
  }
}
```

① 我 可以看到 v2 索引就是 集群 red 的那个索引。

② 由此明 了，20 个 失分片全部来自 一个索引。

一旦我 要索引的 出， 个索引有 立 就很清楚了：v2 索引。我 可以看到 个索引曾 有

10 个主分片和一个副本，而在 20 个分片全不了。可以推，20 个索引就是位于从我集群里不的那一个点上。

`level` 参数 可以接受其他更多：

```
GET _cluster/health?level=shards
```

`shards` 会提供一个 得多的 出，列出 个索引里 个分片的状 和位置。 个 出有候很有用，但是由于太 会比 用。如果 知道 个索引有 了，本章 的其他 API 得更加有用一点。

阻塞等待状态化

当 建 元和集成 ，或者 和 Elasticsearch 相 的自 化脚本，`cluster-health` API 有一个小技巧非常有用。 可以指定一个 `wait_for_status` 参数，它只有在状 之后才会返回。比如：

```
GET _cluster/health?wait_for_status=green
```

个 用会 阻塞 （不 的程序返回控制 ）住直到 `cluster-health` 成 `green` ，也就是所有主分片和副本分片都分配下去了。 自 化脚本和 非常重要。

如果 建一个索引，Elasticsearch 必 在集群状 中向所有 点广播 个 更。那些 点必 初始化些新分片，然后 主 点 些分片已 `<code>Started</code>` 。 个 程很快，但是因延，可能要花 10–20ms。

如果 有个自 化脚本是 (a) 建一个索引然后 (b) 立刻写入一个文 ， 个操作会失 。因 索引 没完全初始化完成。在 (a) 和 (b) 之 的 可能不到 1ms—— 延 来 可不 。

比起使用 `sleep` 命令，直接 的脚本或者 使用 `wait_for_status` 参数 用 `cluster-health` 更好。当索引完全 建好，`cluster-health` 就会 成 `green` ，然后 个 用就会把控制 交 的脚本，然后 就可以 始写入了。

有效的 是：`green` 、 `yellow` 和 `red` 。 个 回会在 到 要求（或者『更高』）的状 返回。比如，如果 要求的是 `yellow`，状 成 `yellow` 或者 `green` 都会打 用。

控 个 点

集群健康 就像是光 的一端—— 集群的所有信息 行高度概述。而 点 API 是在一端。它提供一个 人眼花 乱的 数据的数 ，包含集群的 一个 点 。

点 提供的 如此之多，在完全熟悉它之前，可能都 不清楚 些指 是最 得 注的。我 将会高亮那些最重要的 控指 （但是我 鼓励 接口提供的所有指 ——或者用 Marvel — 因 永 不知道何 需要某个或者 一个 ）。

点 API 可以通 如下命令 行：

```
GET _nodes/stats
```

在 出内容的 ，我 可以看到集群名称和我 的第一个 点：

```
{
  "cluster_name": "elasticsearch_zach",
  "nodes": [
    "UNr6ZMf5Qk-YCPA_L18B0Q": {
      "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"
      ],
    },
    ...
  ]
```

点是排列在一个哈希里，以 点的 UUID 作 名。 示了 点 属性的一些信息（比如 地址和主机名）。 些 如 点未加入集群 自 很有用。通常 会 是端口 用 了，或者 点 定在 的 IP 地址/ 接口上了。

索引部分

索引(**indices**) 部分列出了 个 点上所有索引的聚合 的：

```
"indices": {
  "docs": {
    "count": 6163666,
    "deleted": 0
  },
  "store": {
    "size_in_bytes": 2301398179,
    "throttle_time_in_millis": 122850
  },
}
```

返回的 被 入以下部分：

- **docs** 展示 点内存有多少文 ， 包括 没有从段里清除的已 除文 数量。
- **store** 部分 示 点耗用了多少物理存 。 个指 包括主分片和副本分片在内。如果限流 很大，那可能表明 的磁 限流 置得 低（在[段和合并](#)里 ）。

```

    "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** 表示已 索引了多少文 。 个 是一个累加 数器。在文 被 除的 时候，数 不会下降。要注意的是，在 生内部索引操作的 时候， 个 也会 加，比如 文 更新。

列出了索引操作耗 的 ，正在索引的文 数量，以及 除操作的 似 。

- **get** 表示通 ID 取文 的接口相 的 。包括 个文 的 GET 和 HEAD 求。
- **search** 描述在活 中的搜索（ open_contexts ）数量、 的 数量、以及自 点 以来在 上消耗的 。用 query_time_in_millis / query_total 算的比 ，可以用来粗略的 的 有多高效。比 越大， 个 花 的 越多， 考 了。

fetch 展示了 理的后一半流程 (query-then-fetch 里的 fetch)。如果 fetch 耗 比 query 多， 明磁 慢，或者 取了太多文 ，或者可能搜索 求 置了太大的分 (比如， size: 10000)。

- **merges** 包括了 Lucene 段合并相的信息。它会告目前在行几个合并，合并及的文数量，正在合并的段的大小，以及在合并操作上消耗的。

在 的集群写入力很大，合并非常重要。合并要消耗大量的磁 I/O 和 CPU 源。如果的索引有大量的写入，同又大量的合并数，一定要去 [索引性能技巧](#)。

注意：文更新和除也会致大量的合并数，因它会生最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**展示了已存的器位集合所用的内存数量，以及器被逐出内存的次数。多的逐数可能明需要加大器存的大小，或者的器不太合存（比如它因高基数而在大量生，就像是存一个 now 表式）。

不，逐数是一个很定的指。器是在个段的基上存的，而从一个小的段里逐器，代比从一个大的段里要廉的多。有可能有很大的逐数，但是它都生在小段上，也就意味着些性能只有很小的影。

把逐数指作一个粗略的参考。如果看到数字很大，一下的器，保他都是正常存的。不断逐着的器，怕都生在很小的段上，效果也比正存住了的器差很多。

- **field_data**示 fielddata 使用的内存，用以聚合、排序等等。里也有一个逐数。和 filter_cache 不同的是，里的逐数是很有用的：个数或者至少是接近于 0。因 fielddata 不是存，任何逐都消耗巨大，避免掉。如果在里看到逐数，需要重新估的内存情况，fielddata 限制，求句，或者三者。

- <code>segments</code>会展示个点目前正在服中的Lucene段的数量。是一个重要的数字。大多数索引会有大概 50&x2013;150 个段，怕它存有 TB 的数十条文。段数量大表明合并出了（比如，合并速度跟不上段的建）。注意个是点上所有索引的聚数。住点。

memory展示了Lucene段自己用掉的内存大小。里包括底数据，比如倒排表，字典，和布隆器等。太大的段数量会加些数据来的，个内存使用量就是一个方便用来衡量的度量。

操作系 和 程部分

OS 和 **Process** 部分基本是自描述的，不会在 中展 解。它 列出来基 的 源 ，比如 CPU 和 。**OS** 部分描述了整个操作系 ，而 **Process** 部分只 示 Elasticsearch 的 JVM 程使用的 源情况。

些都是非常有用的指 ，不 通常在 的 控技 里已 都 量好了。 包括下面 些：

- CPU
- 内存使用率
- Swap 使用率
- 打 的文件描述符

JVM 部分

jvm 部分包括了 行 Elasticsearch 的 JVM 程一些很 的信息。最重要的，它包括了 回收的 ， 的 Elasticsearch 集群的 定性有着重大影 。

回收入

在我 描述 之前，先上一 速成 程 解 回收以及它 是非常有用的。如果 JVM 的 回收很熟悉，跳 段。

Java 是一 回收 言，也就是 程序 不用手 管理内存分配和回收。程序 只管写代 ，然后 Java 虚 机 (JVM) 按需分配内存，然后在 后不再需要的 候清理 部分内存。

当内存分配 一个 JVM 程，它是分配到一个大 里， 个 叫做 堆 。JVM 把堆分成 ，用 代 来表示：

新生代 (或者伊)

新 例化的 象分配的空 。新生代空 通常都非常小，一般在 100 MB。新生代也包含 个幸存 空 。

老生代

老的 象存 的空 。些 象 将 期留存并持 上很 一段 。老生代通常比新生代大 很多。Elasticsearch 点可以 老生代用到 30 GB。

当一个 象 例化的 候，它被放在新生代里。当新生代空 了，就会 生一次新生代 回收 (GC)。依然是"存活"状 的 象就被 移到一个幸存区内，而"死掉"的 象被移除。如果一个 象在 多次新生代 GC 中都幸存了，它就会被" 身"置于老生代了。

似的 程在老生代里同 生：空 的 候， 生一次 回收，死掉的 象被移除。

不 ，天下没有免 的午餐。新生代、老生代的 回收都有一个 段会“停止 ”。在 段 里 ，JVM 字面意 上的停止了程序 行，以便跟踪 象 ，收集死亡 象。在 个 停止 段，一切都不会 生。 求不被服 ，ping 不被回 ，分片不被分配。整个世界都真的停止了。

于新生代， 不是什 大 ；那 小的空 意味着 GC 会很快 行完。但是老生代大很多，而 里面一个慢 GC 可能就意味着 1 秒乃至 15 秒的 停—— 于服 器 件来 是不可接受的。

JVM 的 回收采用了 非常 精密的算法，在 少 停方面做得很棒。而且 Elasticsearch 非常努力的 成 回收友好 的程序，比如内部智能的重用 象，重用 冲，以及 用 Doc Values 功能。但最 ，GC 的 率和 依然是 需要去 察的指 。因 它是集群不 定的 号嫌疑人。

一个 常 生 GC 的集群就会因 内存不足而 于高 力下。 些 GC 会 致 点短 内从集群里掉 。 不 定会 致分片 繁重定位，因 Elasticsearch 会 保持集群均衡，保 有足 的副本在 。 接着就 致 流量和磁 I/O 的 加。而所有 些都是在 的集群努力服 于正常的索引和 的同 生的。

而言之， GC 是不好的，需要尽可能的 少。

因 回收 Elasticsearch 是如此重要， 非常熟悉 node-stats API 里的 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 内存使用有关的常量。可以看到有多少 heap 被使用了，多少被指派了（当前被分配的），以及 heap 被允许分配的最大。理想情况下，`heap_committed_in_bytes` 等于 `heap_max_in_bytes`。如果指派的大小更小，JVM 最会被迫调整 heap 大小——是一个非常昂贵的操作。如果这两个数字不相等，[堆内存：大小和交换](#) 学如何正确的配置它。

`heap_used_percent` 指的是得注意的一个数字。Elasticsearch 被配置当 heap 到 75% 的时候开始 GC。如果这个点一直 $\geq 75\%$ ，这个点正处于内存压力状态。是个危险信号，不久的未来可能就有慢 GC 要出来了。

如果 heap 使用率一直 $>= 85\%$ ，就麻烦了。Heap 在 90~95% 之间，面对可怕的性能，此最好的情况是 10~30s 的 GC，最差的情况就是内存溢出（OOM）常常。

```

"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)** 部分分别展示了 GC 中一个代的内存使用情况。

些 很方便 察其相 大小，但是在 的 候，通常并不 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 示新生代和老生代的 回收次数和累 。大多数 候 可以忽略掉新生代的次数：个数字通常都很大。 是正常的。

与之相反，老生代的次数 很小，而且 **collection_time_in_millis** 也 很小。 些是累 ，所以很 出一个 表示 要 始操心了（比如，一个 了一整年的 点，即使很健康，也会 有一个比 大的 数）。 就是像 Marvel 工具很有用的一个原因。GC 数的 是个重要的考 因素。

GC 花 的 也很重要。比如，在索引文 ，一系列 生成了。 是很常 的情况， 刻都会 致 GC。 些 GC 大多数 候都很快， 点影 很小：新生代一般就花一 秒，老生代花一百多 秒。 些跟 10 秒 的 GC 是很不一 的。

我 的最佳建 是定期收集 GC 数和 （或者使用 Marvel）然后 察 GC 率。 也可以 慢 GC 日志 ，在 日志 小 已 。

程池部分

Elasticsearch 在内部 了 程池。 些 程池相互 作完成任 ，有必要的 相互 会 任 。通常来 ， 不需要配置或者 程池，不 看它 的 有 候 是有用的，可以洞察 的集 群表 如何。

有一系列的 程池，但以相同的格式 出：

```
"index": {  
  "threads": 1,  
  "queue": 0,  
  "active": 0,  
  "rejected": 0,  
  "largest": 1,  
  "completed": 1  
}
```

个 程池会列出已配置的 程数量（ **threads** ），当前在 理任 的 程数量（ **active** ），以及在

列中等待 理的任 元数量 (`queue`)。

如果 列中任 元数 到了 限，新的任 元会 始被拒 ， 会在 rejected 上看到它反映出来。 通常 是 的集群在某些 源上 到瓶 的信号。因 列 意味着 的 点或 集群在用最高速度 行，但依然跟不上工作的蜂 而入。

批量操作的被拒 数

如果 到了 列被拒，一般来 都是批量索引 求 致的。通 并 入程序 送大量批量 求非 常 。越多越好 ， 不？

事 上， 个集群都有它能 理的 求上限。一旦 个 被超 ， 列会很快塞 ， 然后新的批量 求就被拒 了。

是一件 好事情 。 列的拒 在回 方面是有用的。它 知道 的集群已 在最大容量了。 比把数据塞 内存 列要来得好。 加 列大小并不能 加性能，它只是 藏了 。当 的集群 只能 秒 理 10000 个文 的 候，无 列是 100 是 10000000 都没 系—— 的集群 是只能 秒 理 10000 个文 。

列只是 藏了性能 ， 而且 来的是真 的数据 失的 。在 列里的数据都是 没 理的， 如果 点挂掉， 些 求都会永久的 失。此外， 列 要消耗大量内存， 也是不理想的。

在 的 用中， 雅的 理来自 列的回 ， 才是更好的 。当 收到拒 的 候， 采取如下几 :

1. 停 入 程 3–5 秒。
2. 从批量操作的 里提取出来被拒 的操作。因 可能很多操作 是成功的。 会告 些 成功， 些被拒 了。
3. 送一个新的批量 求，只包含 些被拒 的操作。
4. 如果依然 到拒 ， 再次从 1 始。

通 个流程， 的代 可以很自然的 集群的 ， 做到自 回 。

拒 不是 : 它 只是意味着 要 后重 。

里的一系列的 程池，大多数 可以忽略，但是有一小部分 是 得 注的：

`indexing`

普通的索引 求的 程池

`bulk`

批量 求，和 条的索引 求不同的 程池

`get`

Get-by-ID 操作

`search`

所有的搜索和 求

merging

用于管理 Lucene 合并的 程池

文件系 和 部分

向下 **node-stats** API, 会看到一串和 的文件系 相 的 : 可用空 , 数据目 路径, 磁 I/O , 等等。如果 没有 控磁 可用空 的 , 可以从 里 取 些 。磁 I/O 也很方便, 不 通常那些更 的命令行工具 (比如 **iostat**) 会更有用些。

然, Elasticsearch 在磁 空 的 候很 行——所以 保不会 。

有 个跟 相 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** 示和 地址 相 的一些基 。包括 点 的通信 (通常是 9300 端口) 以及任意 客 端或者 点客 端的 接。如果看到 里有很多 接数不要担心 ; Elasticsearch 在 点之 了大量的 接。
- **http** 示 HTTP 端口 (通常是 9200) 的 。如果 看到 **total_opened** 数很大而且 在一直上 , 是一个明 信号, 明 的 HTTP 客 端里有没 用 keep-alive 接的。持 的 keep-alive 接 性能很重要, 因 接、断 套接字是很昂 的 (而且浪 文件描述符)。 的客 端 都配置正 。

断路器

于, 我 到了最后一段 : 跟 fielddata 断路器 (在 [断路器 介](#)) 相 的 :

```
"fielddata_breaker": {  
    "maximum_size_in_bytes": 623326003,  
    "maximum_size": "594.4mb",  
    "estimated_size_in_bytes": 0,  
    "estimated_size": "0b",  
    "overhead": 1.03,  
    "tripped": 0  
}
```

里 可以看到断路器的最大 (比如, 一个 求申 更多的内存 会触 断路器)。 个部分 会 知道断路器被触 了多少次, 以及当前配置的 接 。 接 用来 估, 因 有些 求比其他 求更

估。

主要需要 注的是 `tripped` 指 。如果 个数字很大或者持 上 ， 是一个信号， 明 的 求需要化， 或者 需要添加更多内存（ 机上添加， 或者通 添加新 点的方式）。

集群

集群 API 提供了和 点 相似的 出。但有一个重要的区 : 点 表示的是 个点上的 ， 而 **集群** 展示的是 于 个指 ， 所有 点的 和 。

里面提供一些很 得一看的 。比如 可以看到， 整个集群用了 50% 的堆内存， 或者 器存的 逐情况不 重。 个接口主要用途是提供一个比 **集群健康** 更 、 但又没有 点 那的快速概 。 于非常大的集群来 也很有用， 因 那 候 点 的 出已 非常 于 了。

个 API 可以像下面 用：

```
GET _cluster/stats
```

索引

到目前 止， 我 看到的都是以 点 中心 的 : 点有多少内存？用了多少 CPU ？正在服多少个搜索？

有 候从 索引 中心 的角度看 也很有用： 个索引 收到了多少个搜索 求？那个索引 取文 耗了多少 ？

要做到 点， 感 趣的索引（或者多个索引）然后 行一个索引 的 API：

```
GET my_index/_stats ①
```

```
GET my_index,another_index/_stats ②
```

```
GET _all/_stats ③
```

① `my_index` 索引。

② 使用逗号分隔索引名可以 求多个索引 。

③ 使用特定的 `_all` 可以 求全部索引的

返回的 信息和 点 的 出很相似：`search`、`fetch`、`get`、`index`、`bulk`、`segment counts` 等等。

索引 中心的 在有些 候很有用， 比如 或 集群中的 索引， 或者 出某些索引比其他索引更快或者更慢的原因。

践中， 点 中心的 是 得更有用些。瓶 往往是 整个 点而言， 而不是 于 个索引。因 索引一般是分布在多个 点上的， 致以索引 中心的 通常不是很有用， 因 它 是从不同 境的 物理机器上 聚的数据。

索引 中心的 作 一个有用的工具可以保留在 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

有一些任 只能由主 点去 理，比如 建一个新的索引或者在集群中移 分片。由于一个集群中只能有一个主 点，所以只有 一 点可以 理集群 的元数据 。在 99.9999% 的 里， 不会有什 元数据 的 列基本上保持 零。

在一些 的集群里，元数据 的次数比主 点能 理的 快。 会 致等待中的操作会累 成 列。

等待中的任 API 会 展示 列中（如果有的 ）等待的集群 的元数据 更操作：

```
GET _cluster/pending_tasks
```

通常， 都是像 的：

```
{
  "tasks": []
}
```

意味着没有等待中的任 。如果 有一个 的集群在主 点出 瓶 了，等待中的任 列表可能会像

:

```
{
  "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 都被指派了 先 （ 比如 **URGENT** 要比 **HIGH** 更早的 理 ），任 入的次序、操作 入 列多久，以及打算 理什 。在上面的列表中，有一个 建索引(**create-index**) 和 个 分片(**shard-started**) 的操作在等待。

什 候 担心等待中的任 ？

就像曾 提到 的，主 点很少会成 集群的瓶 。唯一可能成 瓶 的是集群状 非常大 而且更新 繁。

例如，如果 允 客 按照他 的意 建任意的 字段，而且 个客 天都有一个独立索引，那 的集群状 会 得非常大。集群状 包括 （ 但不限于 ） 所有索引及其 型，以及 个索引的全部字段。

所以如果 有 100000 客 ，然后 个客 平均有 1000 个字段，而且数据有 90 天的保留期 — 就有九十 个字段需要保存在集群状 中。不管它何 生 更，所有的 点都需要被通知。

主 点必 理 些 ， 需要不小的 CPU ， 加上推送更新的集群状 到所有 点的 。

就是那些可以看到集群状 操作 列上 的集群。没有 的 法可以解决 个 ，不 有三 个 :

- 使用一个更 大的主 点。不幸的是， 垂直 展只是延 必然 果出 而已。
- 通 某些方式限定文 的 性 来限制集群状 的大小。
- 到 某个 后 建 外一个集群。

cat API

如果 常在命令行 境下工作，`cat` API 会非常有用。用 Linux 的 `cat` 命令命名， 些 API 也就 成像 *nix 命令行工具一 工作了。

他 提供的 和前面已 的 API （ 健康、 点 等等 ） 是一 的。但是 出以表格的形式提供，而不是 JSON。 于系 管理 来 是 非常 方便的， 想 一遍集群或者 出内存使用偏高的 点而已。

通 `GET` 求 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}
```

多 API 看起来很熟悉了（是的，`上` 有一只猫:））。我看看 `cat` 的健康 API：

```
GET /_cat/health
```

```
1408723713 12:08:33 elasticsearch_zach yellow 1 1 114 114 0 0 114
```

首先 会注意到的是 是表格 式的 文本，而不是 JSON。其次 会注意到各列 是没有表 的。都是模 *nix 工具 的，因 它假 一旦 出熟悉了， 就再也不想看 表 了。

要 用表 ，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          114 114    0    0    114
unassign
```

，好多了。我 在看到 、集群名称、状 、集群中 点的数量等等—所有信息和 集群健康 API 返回的都一 。

我 再看看 `cat` API 里面的 点 ：

```
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 看到集群里 点的一些 ，不 和完整的 点 出相比而言是非常基 的。
可以包含更多的指 ，但是比起 文 ， 我 直接 cat API 有 些可用的 。

可以 任意 API 添加 ?help 参数来做到 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 个 出 了 面 而被截断了)。

第一列 示完整的名称，第二列 示 写，第三列提供了 于 个参数的 介。 在我 知道了一些列名了
，我 可以用 ?h 参数来明 指定 示 些指 ：

```
GET /_cat/nodes?v&h=ip,port,heapPercent,heapMax
```

ip	port	heapPercent	heapMax
192.168.1.131	9300	53	990.7mb

因 cat API 像 *nix 工具一 工作， 可以使用管道命令将 果 其他工具，比如 sort 、 grep 或者 awk 。例如，通 以下方式可以 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 添加 `?bytes=b` , 我 人 可 的数字格式化, 制它 以字 数 出。随后通 管道命令将
出 `sort` 索引按大小 (第八列) 排序

不幸的是, 会注意到 Marval 索引也出 在 果中, 但是我 目前并不真正在意 些索引。我 把
果 `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
```

! 在 `grep` (通 `-v` 来 掉不需要匹配的数据) 之后, 我 得到了一个没有 Marval 混的索引排序列表了。

只是命令行上 `cat` 的 活性的一个 示例。一旦 了使用 `cat` , 会 它和其他所有 *nix 工具一 并且 始 狂的使用管道、排序和 。如果 是一个系 管理 并且永 都是 SSH 登 到 上, 那 当然要花些 来熟悉 `cat` API 了。

部署

如果 按照 中 做到了 一 , 希望 已 学到了一 件 于 Elasticsearch 的事情并且准 把 的集群部署到生 境。 一章不是在生 中 行集群的 尽指南, 但是它涵 了集群上 之前需要考 的 事 。

主要包括三个方面 :

- 后勤方面的考 , 如硬件和部署策略的建
- 更 合于生 境的配置更改
- 部署后的考 , 例如安全, 最大限度的索引性能和

硬件

按照正常的流程, 可能已 在自己的 本 或集群上使用了 Elasticsearch。 但是当要部署 Elasticsearch 到生 境 , 有一些建 是 需要考 的。 里没有什 必 要遵守的准 , Elasticsearch 被用于在 多的机器上 理各 任 。基于我 在生 境使用 Elasticsearch 集群的 , 些建 可以 提供一个好的起点。

内存

如果有一 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 来 付它是很重要的。即使堆空 是比 小的 候， 也能 操作系 文件 存提供 外的内存。因 Lucene 使用的 多数据 是基于磁 的格式，Elasticsearch 利用操作系 存能 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 的。少于8 GB 会 得其反（ 最需要很多很多的小机器），大于64 GB 的机器也会有 ，我 将在 [堆内存:大小和交](#) 中 。

CPUs

大多数 Elasticsearch 部署往往 CPU 要求不高。因此，相 其它 源，具体配置多少个 (CPU) 不是那 。 具有多个内核的 代 球器，常 的集群使用 到八个核的机器。

如果 要在更快的 CPUs 和更多的核心之 ， 更多的核心更好。多个内核提供的 外并 微快一点点的 率。

硬

硬 所有的集群都很重要， 大量写入的集群更是加倍重要（例如那些存 日志数据的）。硬 是服 器上最慢的子系 ， 意味着那些写入量很大的集群很容易 硬 和，使得它成 集群的瓶 。

如果 担得起 SSD，它将 超出任何旋 介 （注：机械硬 ， 磁 等）。 基于 SSD 的 点， 和索引性能都有提升。如果 担得起， SSD 是一个好的 。

的 I/O 度程序

如果 正在使用 SSDs，保 的系 I/O 度程序是配置正 的。当 向硬 写数据，I/O 度程序决定何 把数据 送到硬 。 大多数 *nix 行版下的 度程序都叫做 cfq (完全公平 列)。

度程序分配 片 到 个 程。并且 化 些到硬 的 多 列的 。但它是 旋 介 化的：机械硬 的固有特性意味着它写入数据到基于物理布局的硬 会更高效。

SSD 来 是低效的，尽管 里没有 及到机械硬 。但是， deadline 或者 noop 被使用。deadline 度程序基于写入等待 行 化， noop 只是一个 的 FIFO 列。

个 的更改可以 来 著的影 。 是使用正 的 度程序，我 看到了500倍的写入能力提升。

如果 使用旋 介 ， 取尽可能快的硬 (高性能服 器硬 ， 15k RPM 器)。

使用 RAID 0 是提高硬 速度的有效途径， 机械硬 和 SSD 来 都是如此。没有必要使用 像或其它 RAID 体，因 高可用已 通 replicas 内建于 Elasticsearch 之中。

最后，避免使用 附加存 (NAS)。人 常声称他 的 NAS 解决方案比本地 器更快更可 。除却 些声称， 我 从没看到 NAS 能配得上它的大肆宣 。NAS 常常很慢， 露出更大的延 和更的平均延 方差，而且它是 点故障的。

快速可靠的。然分布式的性能是很重要的。低延能助保点能容易的通，大能助分片移和恢。代数据中心(1 GbE, 10 GbE)大多数集群都是足的。

即使数据中心近在咫尺，也要避免集群跨越多个数据中心。要避免集群跨越大的地理距。

Elasticsearch假定所有点都是平等的一并不会因有一半的点在150ms外的数据中心而有所不同。更大的延会加重分布式系中的而且使得和排更困。

和NAS的竞争似，个人都声称他的数据中心的路都是健壮和低延的。是真的一直到它不是(失究竟是会生的，可以相信它)。从我的来看，理跨数据中心集群的麻事是根本不得的。

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，在云平台上串起成千的小虚机也是可能的，例如EC2。方式是最好的？

通常，中配或者高配机器更好。避免使用低配机器，因不会希望去管理有上千个点的集群，而且在些低配机器上运行Elasticsearch的也是著的。

与此同，避免使用真正的高配机器。它通常会致源使用不均衡（例如，所有的内存都被使用，但CPU却没有）而且在机上行多个点，会加度。

Java虚机

始行最新版本的Java虚机(JVM)，除非Elasticsearch站上有明。Elasticsearch，特是Lucene，是一个高要求的件。Lucene的元和集成常暴露出JVM本身的bug。些bug的从微的麻到重段，所以，最好尽可能的使用最新版本的JVM。

Java 8烈先于Java 7。不再支持Java 6。Oracle或者OpenJDK是可以接受的，它在性能和定性也差不多。

如果的用程序是用Java写并正在使用客端(注：Transport Client，下同)或点客端(注：Node Client，下同)，保行用程序的JVM和服器的JVM是完全一的。在Elasticsearch的几个地方，使用Java的本地序列化(IP地址、常等等)。不幸的是，Oracle的JVM在几个小版本之有修改序列化格式，从而致奇怪的。情况很少，但最佳践是客端和服器使用相同版本JVM。

不要整JVM置

JVM暴露出几十个(甚至数百)的置、参数和配置。它允行微JVM几乎是各个方面。当遇到一个旋，要打它是人的本性。我求制个本性，而不要去整JVM参数。Elasticsearch是的件，并且我根据多年的使用情况整了当前JVM置。它很容易始旋，并生以衡量的、未知的影，并最使集群入一个慢的、不定的混乱的效果。当集群，第一往往是去除所有的自定配置。多数情况下，此就可以恢定和性能。

Transport Client 与 Node Client

如果 使用的是 Java, 可能想知道何 使用 客 端 (注 : Transport Client, 下同) 与 点客 端 (注 : Node Client, 下同)。 在 的 所述, 客 端作 一个集群和 用程序之 的通信 。它知道 API 并能自 在 点之 , 嗅探集群等等。但它是集群 外部的 , 和 REST 客 端 似。

一方面, 点客 端, 上是一个集群中的 点 (但不保存数据, 不能成 主 点)。因 它是一个 点, 它知道整个集群状 (所有 点 留, 分片分布在 些 点, 等等)。 意味着它可以 行 APIs 但少了一个 点。

里有 个客 端案例的使用情况 :

- 如果要将 用程序和 Elasticsearch 集群 行解 , 客 端是一个理想的 。例如, 如果 的 用程序需要快速的 建和 到集群的 接, 客 端比 “点客 端” ”, 因 它不是一个集群的一部分。
- 似地, 如果 需要 建成千上万的 接, 不想有成千上万 点加入集群。 客 端 () 将是一个更好的 。
- 一方面, 如果 只需要有少数的、 期持久的 象 接到集群, 客 端 点可以更高效, 因 它知道 集群的布局。但是它会使 的 用程序和集群 合在一起, 所以从防火 的角度, 它可能会 成 。

配置管理

如果 已 使用配置管理 (Puppet, Chef, Ansible), 可以跳 此提示。

如果 没有使用配置管理工具, 那 注意了 ! 通 parallel-ssh 管理少量服 器 在可能正常工作, 但伴随着集群的 它将成 一 梦。 在不犯 的情况下手 30 个配置文件几乎是不可能的。

配置管理工具通 自 化更改配置的 程保持集群的一致性。 可能需要一点 来建立和学 , 但它本身 , 随着 的推移会有 厚的回 。

重要配置的修改

Elasticsearch 已 有了 很好 的 , 特 是 及到性能相 的配置或者 。 如果 有疑 , 最好就不要 它。我 已 目 了数十个因 的 置而 致 的集群, 因 它的管理者 改 一个配置或者 就可以 来 100 倍的提升。

NOTE

整 文章, 所有的配置 都同等重要, 和描述 序无 , 所有的配置 , 并 用到 的集群中。

其它数据 可能需要 , 但 得来 , Elasticsearch 不需要。 如果 遇到了性能 , 解决方法通常是更好的数据布局或者更多的 点。 在 Elasticsearch 中很少有“神奇的配置 ”, 如果存在, 我 也已 化了 !

外, 有些 上的 配置在生 境中是 整的。 些 整可能会 的工作更加 松, 又或者因 没 法 定一个 (它取决于 的集群布局) 。

指定名字

Elasticsearch 的集群名字叫 `elasticsearch`。最好 的生境的集群改个名字，改名字的目的很 ，就是防止某人的 本 加入了集群 意外。修改成 `elasticsearch_production` 会很省心。

可以在 的 `elasticsearch.yml` 文件中修改：

```
cluster.name: elasticsearch_production
```

同 ，最好也修改 的 点名字。就像 在可能 的那 ， Elasticsearch 会在 的 点 的时候随机 它指定一个名字。可能会 得 很有趣，但是当凌晨 3 点 的 候，在 回台物理机是 Tagak the Leopard Lord 的 候，就不 得有趣了。

更重要的是， 些名字是在 的 候 生的， 次 点， 它都会得到一个新的名字。会使日志 很混乱，因 所有 点的名称都是不断 化的。

可能会 得 ，我 建 个 点 置一个有意 的、清楚的、描述性的名字，同 可以在 `elasticsearch.yml` 中配置：

```
node.name: elasticsearch_005_data
```

路径

情况下，Elasticsearch 会把 件、日志以及 最重要的数据放在安装目 下。会 来不幸的故事，如果 重新安装 Elasticsearch 的 候不小心把安装目 覆 了。如果不小心，就可能把 的全部数据 掉了。

不要笑， 情况，我 很多次了。

最好的 就是把 的数据目 配置到安装目 以外的地方，同 也可以 移 的 件和日志目 。

可以更改如下：

```
path.data: /path/to/data1,/path/to/data2 ①
```

```
# Path to log files:  
path.logs: /path/to/logs
```

```
# Path to where plugins are installed:  
path.plugins: /path/to/plugins
```

① 注意： 可以通 逗号分隔指定多个目 。

数据可以保存到多个不同的目 ， 如果将 个目 分 挂 不同的硬 ， 可是一个 且高效 一个磁 列（ RAID 0 ）的 法。Elasticsearch 会自 把条 化（注：RAID 0 又称 Stripe（条化），在磁 列中，数据是以条 的方式 穿在磁 列所有硬 中的） 数据分隔到不同的目 ，以便提高性能。

WARNING

多个数据路径的安全性和性能

如同任何磁列（RAID 0）的配置，只有一的数据拷贝保存到硬盘。如果失去了一个硬盘，肯定会失去计算机上的一部分数据。气好的副本在集群的其他地方，可以用来恢复数据和最近的。

Elasticsearch 将全部的条化分片放到一个器来保最小程度的数据失。意味着分片 0 将完全被放置在一个器上。Elasticsearch 没有一个条化的分片跨越在多个器，因一个器的失会破坏整个分片。

性能生的影是：如果添加多个器来提高一个独索引的性能，可能助不大，因大多数点只有一个分片和一个器。多个数据路径只是助如果有多索引／分片在个点上。

多个数据路径是一个非常方便的功能，但到后来，Elasticsearch 并不是磁列（software RAID）的件。如果需要更高的、健的、活的配置，我建议使用磁列（software RAID）的件，而不是多个数据路径的功能。

最小主点数

`minimum_master_nodes` 定义的集群的法定其重要。当的集群中有个 masters（注：主点）的时候，一个配置有助于防止裂，一个主点同存在于一个集群的象。

如果的集群生了裂，那的集群就会在失数据的危中，因主点被是这个集群的最高治者，它决定了什么时候新的索引可以建，分片是如何移的等等。如果有多个 masters 点，数据的完整性将得不到保，因有多个点他有集群的控制。

个配置就是告 Elasticsearch 当没有足够 master 候点的时候，就不要执行 master 点，等 master 候点足了才行。

此置始被配置 master 候点的法定个数（大多数个）。法定个数就是 `(master 候点个数 / 2) + 1`。里有几个例子：

- 如果有 10 个点（能保存数据，同能成 master），法定数就是 6。
- 如果有 3 个候 master 点，和 100 个 data 点，法定数就是 2，只要数数那些可以做 master 的点数就可以了。
- 如果有个点，遇到了。法定数当然是 2，但是意味着如果有一个点挂掉，整个集群就不可用了。置成 1 可以保集群的功能，但是就无法保集群裂了，像的情况，最好至少保有 3 个点。

可以在的 `elasticsearch.yml` 文件中配置：

```
discovery.zen.minimum_master_nodes: 2
```

但是由于 Elasticsearch 是的，可以很容易的添加和除点，但是会改一个法定个数。不得不修改一个索引点的配置并且重置的整个集群只是为了配置生效，将是非常痛苦的一件事情。

基于一个原因，`minimum_master_nodes`（有一些其它配置）允许通过 API 用的方式行配置。

当 的集群在 行的 候， 可以 修改配置：

```
PUT /_cluster/settings
{
  "persistent": {
    "discovery.zen.minimum_master_nodes": 2
  }
}
```

将成 一个永久的配置，并且无 配置 里配置的如何， 个将 先生效。当 添加和 除 master 点的 候， 需要更改 个配置。

集群恢 方面的配置

当 集群重 ， 几个配置 影 的分片恢 的表 。首先，我 需要明白如果什 也没配置将会 生什 。

想象一下假 有 10 个 点， 个 点只保存一个分片， 个分片是一个主分片或者是一个副本分片，或者 有一个有 5 个主分片 /1 个副本分片的索引。有 需要 整个集群做 （比如， 了安装一个新的 程序）， 当 重 的集群，恰巧出 了 5 个 点已 ， 有 5 个 没 的 景。

假 其它 5 个 点出 ， 或者他 根本没有收到立即重 的命令。不管什 原因， 有 5 个 点在 上， 五个 点会相互通信， 出一个 master， 从而形成一个集群。 他 注意到数据不再均 分布， 因 有 5 个 点在集群中 失了， 所以他 之 会立即 分片 制。

最后， 的其它 5 个 点打 加入了集群。 些 点会 它 的数据正在被 制到其他 点， 所以他 除本地数据（因 数据要 是多余的， 要 是 的）。 然后整个集群重新 行平衡， 因 集群的大小已 从 5 成了 10。

在整个 程中， 的 点会消耗磁 和 ， 来回移 数据， 因 没有更好的 法。 于有 TB 数据的大集群， 无用的数据 需要 很 。 如果等待所有的 点重 好了， 整个集群再上 ， 所有的本地的数据都不需要移 。

在我 知道 的所在了， 我 可以修改一些 置来 解它。 首先我 要 Elasticsearch 一个 格的限制：

```
gateway.recover_after_nodes: 8
```

将阻止 Elasticsearch 在存在至少 8 个 点（数据 点或者 master 点）之前 行数据恢 。 个 的 定取决于个人喜好：整个集群提供服 之前 希望有多少个 点在 ？ 情况下，我 置 8， 意味着至少要有 8 个 点， 集群才可用。

在我 要告 Elasticsearch 集群中 有多少个 点， 以及我 意 些 点等待多 ：

```
gateway.expected_nodes: 10
gateway.recover_after_time: 5m
```

意味着 Elasticsearch 会采取如下操作：

- 等待集群至少存在 8 个 点
- 等待 5 分 , 或者 10 个 点上 后, 才 行数据恢 , 取决于 个条件先 到。

三个 置可以在集群重 的 候避免 多的分片交 。 可能会 数据恢 从数个小 短 几秒 。

注意： 些配置只能 置在 config/elasticsearch.yml 文件中或者是在命令行里（它 不能 更新）它 只在整个集群重 的 候有 性作用。

最好使用 播代替 播

Elasticsearch 被配置 使用 播 , 以防止 点无意中加入集群。只有在同一台机器上 行的 点才会自 成集群。

然 播 然 作 件提供, 但它 永 不被使用在生 境了, 否在 得到的 果就是一个 点意外的加入了 的生 境, 是因 他 收到了一个 的 播信号。 于 播 本身 并没有 , 播会 致一些愚蠢的 , 并且 致集群 的脆弱 (比如, 一个 工程 正在 鼓 , 而没有告 , 会 所有的 点突然 不了 方了)。

使用 播, 可以 Elasticsearch 提供一些它 去 接的 点列表。 当一个 点 系到 播列表中的成 , 它就会得到整个集群所有 点的状 , 然后它会 系 master 点, 并加入集群。

意味着 的 播列表不需要包含 的集群中的所有 点, 它只是需要足 的 点, 当一个新 点 系上其中一个并且 上 就可以了。如果 使用 master 时候 点作 播列表, 只要列出三个就可以了。 个配置在 elasticsearch.yml 文件中：

```
discovery.zen.ping.unicast.hosts: ["host1", "host2:port"]
```

于 Elasticsearch 点 的 信息, 参 [Zen Discovery](#) Elasticsearch 文献。

不要触 些配置 !

在 Elasticsearch 中有一些 点, 人 可能不可避免的会 到。 我 理解的, 所有的 整就是 了 化, 但是 些 整, 真的不需要理会它。因 它 常会被乱用, 从而造成系 的不 定或者糟 的性 能, 甚至 者都有可能。

回收器

里已 要介 了 回收入 , JVM 使用一个 回收器来 放不再使用的内存。 这篇内容的是上一篇的一个延 , 但是因 重要, 所以 得 独拿出来作 一 。

不要更改 的 回收器 !

Elasticsearch 的 回收器 (GC) 是 CMS。 个 回收器可以和 用并行 理, 以便它可以最小化停 。然而, 它有 个 stop-the-world 段, 理大内存也有点吃力。

尽管有 些 点, 它 是目前 于像 Elasticsearch 低延 需求 件的最佳 回收器。官方建 使用

CMS。

在有一款新的回收器，叫 G1 回收器（G1GC）。新款的 GC 被，旨在比 CMS 更小的停，以及大内存的理能力。它的原理是把内存分成多区域，并且一些区域最有可能需要回收内存。通过先收集些区域（garbage first），生成更小的停，从而能更大的内存。

听起来很棒！憾的是，G1GC 是太新了，常新的 bugs。些通常是段（segfault）型，便造成硬的崩。Lucene 的套件回收算法要求格，看起来些陷 G1GC 并没有很好地解决。

我 很希望在将来某一天推 使用 G1GC，但是于在，它不能足定的足 Elasticsearch 和 Lucene 的要求。

程池

多人 喜 整 程池。无什原因，人都 加 程数无法抵抗。索引太多了？加 程！搜索太多了？加 程！点空 率低于 95%？加 程！

Elasticsearch 的程置已是很合理的了。于所有的程池（除了 [搜索](#)），程个数是根据 CPU 核心数置的。如果有 8 个核，可以同行的只有 8 个程，只分配 8 个程任何特定的程池是有道理的。

搜索 程池 置的大一点，配置 `int((核心数 * 3) / 2) + 1`。

可能会 某些 程可能会阻塞（如磁 上的 I/O 操作），所以 才想加大 程的。于 Elasticsearch 来 并不是一个：因 大多数 I/O 的操作是由 Lucene 程管理的，而不是 Elasticsearch。

此外，程池通 彼此之 的工作配合。不必再因 它正在等待磁 写操作而担心 程阻塞，因 程早已把 个工作交 外的 程池，并且 行了 。

最后，的理器的算能力是有限的，有更多的 程会致的理器繁切 程上下文。一个理器同 只能行一个 程。所以当它需要切 到其它不同的 程的候，它会存 当前的状（寄存器等等），然后加 外一个 程。如果幸 的，个切 生在同一个核心，如果不幸的，个切可能 生在不同的核心，就需要在内核 上 行 。

个上下文的切，会 CPU 周期 来管理 度的 ；在 代的 CPUs 上，估高 30 μs 。也就是 程会被堵塞超 30 μs ，如果 个 用于 程的 行，有可能早就 束了。

人 常稀里糊 的 置 程池的 。8 个核的 CPU，我 遇到 有人配了 60、100 甚至 1000 个 程。些 置只会 CPU 工作效率更低。

所以，下次 不要 整 程池的 程数。如果 真 想 整 ，一定要 注 的 CPU 核心数，最多 置成核心数的 倍，再多了都是浪 。

堆内存:大小和交

Elasticsearch 安装后 置的堆内存是 1 GB。于任何一个 部署来，个 置都太小了。如果正在使用 些 堆内存配置，的集群可能会出 。

里有 方式修改 Elasticsearch 的堆内存。最 的一个方法就是指定 `ES_HEAP_SIZE` 境 量。服

程在候会取个量，并相的置堆的大小。比如，可以用下面的命令置它：

```
export ES_HEAP_SIZE=10g
```

此外，也可以通过命令行参数的形式，在程序的候把内存大小它，如果得更的：

```
./bin/elasticsearch -Xmx10g -Xms10g ①
```

① 保堆内存最小（`Xms`）与最大（`Xmx`）的大小是相同的，防止程序在行改堆内存大小，是一个很耗系源的程。

通常来，置`ES_HEAP_SIZE`境量，比直接写`-Xmx -Xms`更好一点。

把的内存的（少于）一半 Lucene

一个常的是 Elasticsearch 分配的内存太大了。假有一个 64 GB 内存的机器，天，我要把 64 GB 内存全都 Elasticsearch。因 越多越好！

当然，内存于 Elasticsearch 来是重要的，它可以被多内存数据使用来提供更快的操作。但是到里，有外一个内存消耗大非堆内存（off-heap）：Lucene。

Lucene 被可以利用操作系底机制来存内存数据。Lucene 的段是分存到个文件中的。因 段是不可的，些文件也都不会化，是存友好的，同操作系也会把些段文件存起来，以便更快的。

Lucene 的性能取决于和操作系的相互作用。如果 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 Lucene。将重地影全文索的性能。

准的建 是把 50% 的可用内存作 Elasticsearch 的堆内存，保留剩下的 50%。当然它也不会被浪，Lucene 会很意利用起余下的内存。

如果 不需要分字符串做聚合算（例如，不需要`fielddata`）可以考降低堆内存。堆内存越小，Elasticsearch（更快的 GC）和 Lucene（更多的内存用于存）的性能越好。

不要超 32 GB！

里有外一个原因不分配大内存 Elasticsearch。事上，JVM 在内存小于 32 GB 的时候会采用一个内存象指技。

在 Java 中，所有的象都分配在堆上，并通一个指行引用。普通象指（OOP）指向些象，通常 CPU 字的大小：32 位或 64 位，取决于的理器。指引用的就是个 OOP 的字位置。

于 32 位的系，意味着堆内存大小最大 4 GB。于 64 位的系，可以使用更大的内存，但是 64 位的指意味着更大的浪，因的指本身大了。更糟的是，更大的指在主内存和各存（例如 LLC，L1 等）之移数据的时候，会占用更多的。

Java 使用一个叫作 **内存指针** (**compressed oops**) 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 个对象，而不是 40 个字节。最，也就是堆内存映射到 32 GB 的物理内存，也可以用 32 位的指针表示。

一旦越过那个神奇的 ~32 GB 的界限，指针就会切回普通对象的指针。一个对象的指针都变了，就会使用更多的 CPU 内存，也就是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时候的 32 GB 内存。

这段描述的意思就是：即便有足够的内存，也尽量不要超过 32 GB。因为它浪费了内存，降低了 CPU 的性能，需要 GC 大内存。

到底需要低于 32 GB 多少，来配置我的 JVM？

憾的是，需要看情况。一切的分区要根据 JVMs 和操作系统而定。如果想保证其安全可靠，设置堆内存 31 GB 是一个安全的。另外，可以在 JVM 配置里添加 `-XX:+PrintFlagsFinal` 来使用 JVM 的界限，并且 `UseCompressedOops` 的是否为 true。于自己使用的 JVM 和操作系统，将达到最合适的堆内存界限。

例如，我在一台安装 Java 1.7 的 Mac OSX 上，可以看到指针在被禁用之前，最大堆内存大小是在 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，可以看到指针在被禁用之前，最大堆内存大小是在 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，影响内存指针使用的界限，是会根据 JVM 的不同而变化的。所以从其他地方取的例子，需要慎使用，要根据操作系统的配置和 JVM。

如果使用的是 Elasticsearch v2.2.0，日志其会告诉你 JVM 是否正在使用内存指针。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]
```

表明内存指 正在被使用。如果没有，日志消息会 示 [false]。

我有一个 1 TB 内存的机器！

个 32 GB 的分割 是很重要的。那如果 的机器有很大的内存 ？ 一台有着 512–768 GB 内存的服 器愈 常 。

首先，我 建 避免使用 的高配机器（参考 [硬件](#)）。

但是如果 已 有了 的机器， 有三个可 ：

- 主要做全文 索 ？考 Elasticsearch 4 - 32 GB 的内存， Lucene 通 操作系 文件 存来利用余下的内存。那些内存都会用来 存 segments， 来 速的全文 索。
- 需要更多的排序和聚合？而且大部分的聚合 算是在数字、日期、地理点和 非分 字符串上？ 很幸 ， 的聚合 算将在内存友好的 doc values 上完成！ Elasticsearch 4 到 32 GB 的内存，其余部分 操作系 存内存中的 doc values。
- 在 分 字符串做大量的排序和聚合（例如， 或者 SigTerms, 等等）不幸的是， 意味着 需要 fielddata，意味着 需要堆空 。考 在 个机器上 行 个或多个 点，而不是 有大量 RAM 的一个 点。 然要 持 50% 原 。

假 有个机器有 128 GB 的内存， 可以 建 个 点， 个 点内存分配不超 32 GB。 也就是 不超 64 GB 内存 ES 的堆内存，剩下的超 64 GB 的内存 Lucene。

如果 一 ， 需要配置 `cluster.routing.allocation.same_shard.host: true` 。 会防止同一个分片（shard）的主副本存在同一个物理机上（因 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 墓

是 而易 的，但是 是有必要 的更清楚一点：内存交 到磁 服 器性能来 是 致命 的。想想看：一个内存操作必 能 被快速 行。

如果内存交 到磁 上，一个 100 微秒的操作可能 成 10 秒。 再想想那 多 10 微秒的操作 延累加起来。不 看出 swapping 于性能是多 可怕。

最好的 法就是在 的操作系 中完全禁用 swap。 可以 禁用：

```
sudo swapoff -a
```

如果需要永久禁用， 可能需要修改 `/etc/fstab` 文件， 要参考 的操作系 相 文 。

如果 并不打算完全禁用 swap，也可以 降低 swappiness 的 。 个 决定操作系 交 内存的 率。 可以 防正常情况下 生交 ，但 允 操作系 在 急情况下 生交 。

于大部分Linux操作系 ，可以在 `sysctl` 中 配置：

```
vm.swappiness = 1 ①
```

① `swappiness` 置 1 比 置 0 要好，因 在一些内核版本 `swappiness` 置 0 会触 系 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，需要打 配置文件中的 `mlockall`。它的作用就是允 JVM 住内存，禁止操作系 交 出去。在 的 `elasticsearch.yml` 文件中，置如下：

```
bootstrap.mlockall: true
```

文件描述符和 MMap

Lucene 使用了大量的 文件。同，Elasticsearch 在 点和 HTTP 客 端之行通信也使用了大量的套接字 (注：sockets)。所有 一切都需要足 的文件描述符。

可悲的是，多 代的 Linux 行版本，个 程 允 一个微不足道的 1024 文件描述符。一个小小的 Elasticsearch 点来 在是太低了，更不用 一个 理数以百 索引的 点。

加 的文件描述符，置一个很大的 ，如 64,000。个 程困 得 人 火，它高度依 于的特定操作系 和分布。参考 操作系 文 来 定如何最好地修改允 的文件描述符数量。

一旦 已 改 了它， Elasticsearch，以 保它的真的起作用并且有足 的文件描述符：

```
GET /_nodes/process

{
  "cluster_name": "elasticsearch_zach",
  "nodes": {
    "TGn9i02_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, ①
        "mlockall": true
      }
    }
  }
}
```

① `max_file_descriptors` 字段 示 Elasticsearch 程可以 的可用文件描述符数量。

Elasticsearch 各文件混合使用了 NioFs（注：非阻塞文件系）和 MMapFs（注：内存映射文件系）。保配置的最大映射数量，以便有足够的虚内存可用于 mmapped 文件。可以置：

```
sysctl -w vm.max_map_count=262144
```

或者可以在 `/etc/sysctl.conf` 通修改 `vm.max_map_count` 永久置它。

在生之前，重温个列表

在入生之前，可能了本。本章中及的非常好，一般是可以知道的，但是，正部署到生境之前需要重温个列表。

一些会地阻止（如：可用的文件描述符太少）。因他很快出来，些都是容易的。其他的一些，如裂和内存置，只有在糟的事情生之后才可。在一点上，解决法往往是凌乱和繁的。

在生之前，通当配置集群来主阻止些情况生，是更好的。所以如果想要从整本的一个部分折角（或保存），本章将是一个很好的。在部署到生境的前一周，地里出的列表，并所有的建。

部署后

一旦将集群部署到生境后，就需要有一些工具及最佳践来保集群行在最佳状。本章将探配置、日志、索引性能化以及集群。

更置

Elasticsearch里很多置都是的，可以通 API修改。需要制重点（或者集群）的配置修改都要力避免。而且然通静配置也可以完成些更，我建是用 API 来。

集群更新 API 有工作模式：

(Transient)

些更在集群重之前一直会生效。一旦整个集群重，些配置就被清除。

永久 (Persistent)

些更会永久存在直到被式修改。即使全集群重它也会存活下来并覆掉静配置文件里的。

或永久配置需要在 JSON 体里分指定：

```

PUT /_cluster/settings
{
  "persistent": {
    "discovery.zen.minimum_master_nodes": 2 ①
  },
  "transient": {
    "indices.store.throttle.max_bytes_per_sec": "50mb" ②
  }
}

```

① 个永久 置会在全集群重 存活下来。

② 个 置会在第一次全集群重 后被移除。

可以 更新的 置的完整清 , [online reference docs](#)。

日志

Elasticsearch 会 出很多日志，都放在 `ES_HOME/logs` 目 下。 的日志 等 是 `INFO` 。它提供了 度的信息，但是又 好了不至于 的日志太 大。

当 的 候，特 是 点 相 的 （因 个 常依 于各式 于繁 的 配置）， 提高日 志 等 到 `DEBUG` 是很有 助的。

可以 修改 `logging.yml` 文件然后重 的 点——但是 做即繁 会 致不必要的宕机 。作 替代， 可以通 `cluster-settings` API 更新日志 ， 就像我 前面 学 的那 。

要 个更新， 感 趣的日志器，然后在前面 上 `logger.` 。 根日志器 可以用 `logger._root` 来表示。

我 高 点 的日志 :

```

PUT /_cluster/settings
{
  "transient": {
    "logger.discovery": "DEBUG"
  }
}

```

置失效，Elasticsearch 将 始 出 `discovery` 模 的 `DEBUG` 的日志。

TIP 避免使用 `TRACE`。 个 非常的 ， 到日志反而不再有用了。

慢日志

有 一个日志叫 慢日志 。 个日志的目的是捕 那些超 指定 的 和索引 求。 个日志用来追踪由用 生的很慢的 求很有用。

情况，慢日志是不 的。要 它，需要定 具体 作 (`query, fetch` 是 `index`) ,

期望的事件 等 (WARN、DEBUG 等), 以及。

是一个索引 的 置, 也就是 可以独立 用 一个索引:

```
PUT /my_index/_settings
{
    "index.search.slowlog.threshold.query.warn" : "10s", ①
    "index.search.slowlog.threshold.fetch.debug": "500ms", ②
    "index.indexing.slowlog.threshold.index.info": "5s" ③
}
```

① 慢于 10 秒 出一个 WARN 日志。

② 取慢于 500 秒 出一个 DEBUG 日志。

③ 索引慢于 5 秒 出一个 INFO 日志。

也可以在 `elasticsearch.yml` 文件里定 些 。没 有 置的索引会自 承在静 配置文件里配置的参数。

一旦 置 了, 可以和其他日志器一 切 日志 等 :

```
PUT/_cluster/settings
{
    "transient": {
        "logger.index.search.slowlog" : "DEBUG", ①
        "logger.index.indexing.slowlog" : "WARN" ②
    }
}
```

① 置搜索慢日志 DEBUG 。

② 置索引慢日志 WARN 。

索引性能技巧

如果 是在一个索引 很重的 境, 比如索引的是基 施日志, 可能 意 性一些搜索性能 取更快 的索引速率。在 些 景里, 搜索常常是很少 的操作, 而且一般是由 公司内部的人 起的。他 也 意 一个搜索等上几秒 , 而不像普通消 者, 要求一个搜索必 秒 返回。

基于 特殊的 景, 我 可以有几 衡 法来提高 的索引性能。

些技巧 用于 Elasticsearch 1.3 及以后的版本

本 是 最新几个版本的 Elasticsearch 写的, 然大多数内容在更老的版本也也有效。

不 , 本 提及的技巧, 只 1.3 及以后版本。 版本后有不少性能提升和故障修 是直接影 到索引的。事 上, 有些建 在老版本上反而会因 故障或性能 陷而 降低 性能。

科学的 性能

性能 永 是 的，所以在 的方法里已 要尽可能的科学。随机 弄旋 以及写入 好 法。如果有太多 可能，我 就无法判断到底 一 有最好的效果。合理的 可不是做性能 方法如下：

1. 在 个 点上， 个分片，无副本的 景 性能。
2. 在 100% 配置的情况下 性能 果， 就有了一个 比基 。
3. 保性能 行足 的 (30 分 以上) 可以 估 期性能，而不是短期的峰 或延 。一些事件（比如段合并，GC）不会立刻 生，所以性能概况会随着 而改 的。
4. 始在基 上逐一修改 。 格 它，如果性能提升可以接受，保留 个配置， 始下一 。

使用批量 求并 整其大小

而易 的， 化性能 使用批量 求。批量的大小 取决于 的数据、分析和集群配置，不 次批量 数据 5–15 MB 大是个不 的起始点。注意 里 的是物理字 数大小。文 数 批量大小来 不是一个好指 。比如 ，如果 次批量索引 1000 个文 ， 住下面的事：

- 1000 个 1 KB 大小的文 加起来是 1 MB 大。
- 1000 个 100 KB 大小的文 加起来是 100 MB 大。

可是完完全全不一 的批量大小了。批量 求需要在 点上加 内存，所以批量 求的物理大小比 文 数重要得多。

从 5–15 MB 始 批量 求大小， 慢 加 个数字，直到 看不到性能提升 止。然后 始 加 的批量写入的并 度（多 稨等等 法）。

用 Marvel 以及 如 `iostat` 、 `top` 和 `ps` 等工具 控 的 点，察 源什 候 到瓶 。如果 始收到 `EsRejectedExecutionException` ， 的集群没 法再 了：至少有一 源到瓶 了。或者 少并 数，或者提供更多的受限 源（比如从机械磁 成 SSD），或者添加更多 点。

NOTE

写数据的 候，要 保批量 求是 往 的全部数据 点的。不要把所有 求都 个 点，因 个 点会需要在 理的 候把所有批量 求都存在内存里。

存

磁 在 代服 器上通常都是瓶 。Elasticsearch 重使用磁 ， 的磁 能 理的 吐量越大， 的 点就越 定。 里有一些 化磁 I/O 的技巧：

- 使用 SSD。就像其他地方提 的，他 比机械磁 秀多了。
- 使用 RAID 0。条 化 RAID 会提高磁 I/O，代 然就是当一 硬 故障 整个就故障了。不要使用 像或者奇偶校 RAID 因 副本已 提供了 个功能。
- 外，使用多 硬，并允 Elasticsearch 通 多个 `path.data` 目 配置把数据条 化分配到它 上面。
- 不要使用 程挂 的存 ，比如 NFS 或者 SMB/CIFS。 个引入的延 性能来 完全是背道而 的。
- 如果 用的是 EC2，当心 EBS。即便是基于 SSD 的 EBS，通常也比本地 例的存 要慢。

段和合并

段合并的 算量 大，而且 要吃掉大量磁 I/O。合并在后台定期操作，因 他 可能要很 才能完成，尤其是比 大的段。 个通常来 都没 ，因 大 模段合并的概率是很小的。

不 有 候合并会 累写入速率。如果 个真的 生了，Elasticsearch 会自 限制索引 求到 个 程里。 个可以防止出 段爆炸 ，即数以百 的段在被合并之前就生成出来。如果 Elasticsearch 合并 累索引了，它会会 一个声明有 now throttling indexing 的 INFO 信息。

Elasticsearch 置在 比 保守：不希望搜索性能被后台合并影 。不 有 候（尤其是 SSD，或者日志 景）限流 太低了。

是 20 MB/s， 机械磁 是个不 的 置。如果 用的是 SSD，可以考 提高到 100–200 MB/s。 的系 个 合：

```
PUT /_cluster/settings
{
  "persistent": {
    "indices.store.throttle.max_bytes_per_sec": "100mb"
  }
}
```

如果 在做批量 入，完全不在意搜索， 可以 底 掉合并限流。 的索引速度 到 磁 允 的 限：

```
PUT /_cluster/settings
{
  "transient": {
    "indices.store.throttle.type": "none" ①
  }
}
```

① 置限流 型 none 底 合并限流。等 完成了 入， 得改回 merge 重新打 限流。

如果 使用的是机械磁 而非 SSD， 需要添加下面 个配置到 的 `elasticsearch.yml` 里：

```
index.merge.scheduler.max_thread_count: 1
```

机械磁 在并 I/O 支持方面比 差，所以我 需要降低 个索引并 磁 的 程数。 个 置允 `max_thread_count + 2` 个 程同 行磁 操作，也就是 置 1 允 三个 程。

于 SSD， 可以忽略 个 置， 是 `Math.min(3, Runtime.getRuntime().availableProcessors() / 2)`， SSD 来 行的很好。

最后， 可以 加 `index.translog.flush_threshold_size` 置，从 的 512 MB 到更大一些的 ，比如 1 GB。 可以在一次清空触 的 候在事 日志里 累出更大的段。而通 建更大的段，清空的 率 低，大段合并的 率也 低。 一切合起来 致更少的磁 I/O 和更好的索引速率。当然， 会需要 量 的 heap 内存用以 累更大的 冲空 ， 整 个 置的 候 住 点。

其他

最后，有一些其他得考的西需要住：

- 如果的搜索果不需要近的准度，考把个索引的 `index.refresh_interval` 改到 `30s`。如果是在做大批量入，入期可以通置个 `-1` 掉刷新。忘在完工的时候重新它。
- 如果在做大批量入，考通置 `index.number_of_replicas: 0` 副本。文在制的时候，整个文内容都被往副本点，然后逐字的把索引程重一遍。意味着个副本也会行分析、索引以及可能的合并程。

相反，如果的索引是零副本，然后在写入完成后再副本，恢程本上只是一个字到字的。相比重索引程，个算是相当高效的了。

- 如果没有个文自 `ID`，使用 Elasticsearch 的自 `ID` 功能。个避免版本做了化，因自生成的 `ID` 是唯一的。
- 如果在使用自己的 `ID`，使用一 [Lucene 友好的 ID](#)。包括零填充序列 `ID`、`UUID-1` 和秒；些 `ID` 都是一致的，良好的序列模式。相反的，像 `UUID-4` 的 `ID`，本上是随机的，比很低，会明慢 Lucene。

推分片分配

正如我在[水平容](#)，Elasticsearch 将自在可用点行分片均衡，包括新点的加入和有点的。

理上来，个是理想的行，我想要提副本分片来尽快恢失的主分片。我同也希望保源在整个集群的均衡，用以避免点。

然而，在践中，立即的再均衡所造成的会比其解决的更多。例来，考到以下情形：

1. Node（点）19 在中失了（某个家到了源）
2. Master 立即注意到了个点的，它决定在集群内提其他有 Node 19 上面的主分片的副本分片主分片
3. 在副本被提主分片以后，master 点始行恢操作来重建失的副本。集群中的点之互相拷分片数据，力，集群状。
4. 由于目前集群于非平衡状，个程有可能会触小模的分片移。其他不相的分片将在点移来到一个最佳的平衡状

与此同，那个到源的倒管理，把服器好源行了重，在点 Node 19 又重新加入了集群。不幸的是，个点被告知当前的数据已没有用了，数据已在其他点上重新分配了。所以 Node 19 把本地的数据行除，然后重新始恢集群的其他分片（然后又致了一个新的再平衡）

如果一切听起来是不必要的且大，那就了。是的，不前提是知道个点会很快回来。如果点 Node 19 真的了，上面的流程正是我想要生的。

了解决瞬中断的，Elasticsearch 可以推分片的分配。可以的集群在重新分配之前有去个点是否会再次重新加入。

修改 延

情况，集群会等待一分 来 看 点是否会重新加入，如果 个 点在此期 重新加入，重新加入的点会保持其 有的分片数据，不会触 新的分片分配。

通 修改参数 `delayed_timeout`， 等待 可以全局 置也可以在索引 行修改：

```
PUT /_all/_settings ①
{
  "settings": {
    "index.unassigned.node_left.delayed_timeout": "5m" ②
  }
}
```

① 通 使用 `_all` 索引名，我 可以 集群里面的所有的索引使用 个参数

② 被修改成了 5 分

个配置是 的，可以在 行 行修改。如果 希望分片立即分配而不想等待， 可以 置参数：
`delayed_timeout: 0`.

NOTE

延 分配不会阻止副本被提 主分片。集群 是会 行必要的提 来 集群回到 `yellow` 状 。 失副本的重建是唯一被延 的 程。

自 取消分片 移

如果 点在超 之后再回来，且集群 没有完成分片的移 ，会 生什 事情 ？在 情形下， Elasticsearch 会 机器磁 上的分片数据和当前集群中的活 主分片的数据是不是— —如果 者匹配， 明没有 来新的文 ，包括 除和修改—那 master 将会取消正在 行的再平衡并恢 机器磁 上的数据。

之所以 做是因 本地磁 的恢 永 要比 要快，并且我 保 了他 的分片数据是一 的， 个 程可以 是双 。

如果分片已 生了分 （比如： 点 之后又索引了新的文 ），那 恢 程会 按照正常流程 行。重新加入的 点会 除本地的、 的数据，然后重新 取一 新的。

重

有一天 会需要做一次集群的 重 ——保持集群在 和可操作，但是逐一把 点下 。

常 的原因：Elasticsearch 版本升 ，或者服 器自身的一些 操作（比如操作系 升 或者硬件相 ）。不管 情况，都要有一 特 的方法来完成一次 重 。

正常情况下，Elasticsearch 希望 的数据被完全的 制和均衡的分布。如果 手 了一个 点，集群会立刻 点的 失并 始再平衡。如果 点的 是短期工作的 ， 一点就很 人了，因 大型分片的再平衡需要花 相当的 （想想 制 1TB 的数据——即便在高速 上也是不一般的事情了）。

我 需要的是，告 Elasticsearch 推 再平衡，因 外部因子影 下的集群状 ， 我 自己更了解。操作流程如下：

1. 可能的 ， 停止索引新的数据。 然不是 次都能真的做到，但是 一 可以 助提高恢 速度。
2. 禁止分片分配。 一 阻止 Elasticsearch 再平衡 失的分片，直到 告 它可以 行了。如果 知道 口会很短， 个主意棒 了。 可以像下面 禁止分配：

```
PUT /_cluster/settings
{
  "transient": {
    "cluster.routing.allocation.enable": "none"
  }
}
```

3. 个 点。
4. 行 /升 。
5. 重 点，然后 它加入到集群了。
6. 用如下命令重 分片分配：

```
PUT /_cluster/settings
{
  "transient": {
    "cluster.routing.allocation.enable": "all"
  }
}
```

分片再平衡会花一些 。一直等到集群 成 色 状 后再 。

7. 重 第 2 到 6 操作剩余 点。
8. 到 可以安全的恢 索引了（如果 之前停止了的 ），不 等待集群完全均衡后再恢 索引，也 有助于提高 理速度。

的集群

使用无 个存 数据的 件，定期 的数据都是很重要的。Elasticsearch 副本提供了高可 性；它 可以容忍零星的 点 失而不会中断服 。

但是，副本并不提供 性故障的保 。 情况， 需要的是 集群真正的 ——在某些 西 出 的 时候有一个完整的拷 。

要 的集群， 可以使用 snapshot API。 个会拿到 集群里当前的状 和数据然后保存到一个共享 里。 个 程是"智能"的。 的第一个快照会是一个数据的完整拷 ， 但是所有后 的快照会保留的是已存快照和新数据之 的差 。随着 不 的 数据 行快照， 也在 量的添加和 除。 意味着后 会相当快速，因 它 只 很小的数据量。

要使用 个功能， 必 首先 建一个保存数据的 。有多个 型可以供 ：

- 共享文件系，比如 NAS
- Amazon S3
- HDFS (Hadoop 分布式文件系)
- Azure Cloud

建

我部署一个共享文件系：

```
PUT _snapshot/my_backup ①
{
  "type": "fs", ②
  "settings": {
    "location": "/mount/backups/my_backup" ③
  }
}
```

- ① 我的取一个名字，在本例它叫 `my_backup`。
 ② 我指定的型是一个共享文件系。
 ③ 最后，我提供一个已挂的作目的地址。

注意：共享文件系路径必须保证集群所有点都可以到。

会在挂点建立和所需的元数据。有一些其他的配置可能想要配置的，一些取决于的点、的性能状况和位置：

`max_snapshot_bytes_per_sec`

当快照数据入，一个参数控制个程的限流情况。是秒 `20mb`。

`max_restore_bytes_per_sec`

当从恢数据，一个参数控制什候恢程会被限流以保障的不会被占。是秒 `20mb`。

假我有一个非常快的，而且外的流量也很OK，那我可以加些：

```
POST _snapshot/my_backup/ ①
{
  "type": "fs",
  "settings": {
    "location": "/mount/backups/my_backup",
    "max_snapshot_bytes_per_sec": "50mb", ②
    "max_restore_bytes_per_sec": "50mb"
  }
}
```

- ① 注意我用的是 `POST` 而不是 `PUT`。会更新已有 的置。
 ② 然后添加我新的置。

快照所有打 的索引

一个 可以包含多个快照。 个快照跟一系列索引相 (比如所有索引, 一部分索引, 或者 个索引)。当 建快照的 候, 指定 感 趣的索引然后 快照取一个唯一的名字。

我 从最基 的快照命令 始 :

```
PUT _snapshot/my_backup/snapshot_1
```

个会 所有打 的索引到 my_backup 下一个命名 snapshot_1 的快照里。 个用会立刻返回, 然后快照会在后台 行。

通常 会希望 的快照作 后台 程 行, 不 有 候 会希望在 的脚本中一直等待到完成。 可以通 添加一个 wait_for_completion :

TIP

```
PUT _snapshot/my_backup/snapshot_1?wait_for_completion=true
```

个会阻塞 用直到快照完成。注意大型快照会花很 才返回。

快照指定索引

行 是 所有打 的索引。不 如果 在用 Marvel, 不是真的想要把所有 断相 的 .marvel 索引也 起来。可能 就 根没那 大空 所有数据。

情况下, 可以在快照 的集群的 候指定 些索引 :

```
PUT _snapshot/my_backup/snapshot_2
{
  "indices": "index_1,index_2"
}
```

个快照命令 在只会 index1 和 index2 了。

列出快照相 的信息

一旦 始在 的 里 起快照了, 可能就慢慢忘 里面各自的 了——特 是快照按照 分命名的 候 (比如, backup_2014_10_28)。

要 得 个快照的信息, 直接 和快照名 起一个 GET 求 :

```
GET _snapshot/my_backup/snapshot_2
```

个会返回一个小 , 包括快照相 的各 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 取一个 中所有快照的完整列表，使用 `_all` 占位符替 掉具体的快照名称：

```
GET _snapshot/my_backup/_all
```

除快照

最后，我 需要一个命令来 除所有不再有用的旧快照。 只要 /快照名称 一个 的 `DELETE` HTTP 用：

```
DELETE _snapshot/my_backup/snapshot_2
```

用 API 除快照很重要，而不能用其他机制（比如手 除，或者用 S3 上的自 清除工具）。因 快照是 量的，有可能很多快照依 于 去的段。`delete` API 知道 些数据 在被更多近期快照使用，然后会只 除不再被使用的段。

但是，如果 做了一次人工文件 除， 将会面 重 坏的 ，因 在 除的是可能 在使用中的数据。

控快照 度

`wait_for_completion` 提供了一个 控的基 形式，但 怕只是 一个中等 模的集群做快照恢 的 候，它都真的不 用。

外一个 API 会有快照状态更新的信息。首先可以快照 ID 行一个 GET，就像我之前取一个特定快照的信息做的那样：

```
GET _snapshot/my_backup/snapshot_3
```

如果用一个命令的候快照在行中，会看到它什么时候开始，花了多久等等信息。不要注意，一个 API 用的是快照机制相同的线程池。如果在快照非常大的分片，状态更新的间隔会很大，因为 API 在争相同的线程池资源。

更好的方案是取 `_status` API 数据：

```
GET _snapshot/my_backup/snapshot_3/_status
```

`_status` API 立刻返回，然后输出的多的是：

```
{
  "snapshots": [
    {
      "snapshot": "snapshot_3",
      "repository": "my_backup",
      "state": "IN_PROGRESS", ①
      "shards_stats": {
        "initializing": 0,
        "started": 1, ②
        "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
      }
    },
    ...
  }
}

```

- ① 一个正在 行的快照会 示 IN_PROGRESS 作 状 。
- ② 个特定快照有一个分片 在 (外四个已 完成)。

包括快照的 体状况，但也包括下 到 个索引和 个分片的 。 个 展示了有 快照 展的 非常 的 。 分片可以在不同的完成状 :

INITIALIZING

分片在 集群状 看看自己是否可以被快照。 个一般是非常快的。

STARTED

数据正在被 到 。

FINALIZING

数据 完成；分片 在在 送快照元数据。

DONE

快照完成！

FAILED

快照 理的 候 到了 ， 个分片/索引/快照不可能完成了。 的日志 取更多信息。

取消一个快照

最后， 可能想取消一个快照或恢 。 因 它 是 期 行的 程， 行操作的 候一个 或者 就会 花很 来解决——而且同 会耗尽有 的 源。

要取消一个快照，在他 行中的 候 的 除快照就可以：

```
DELETE _snapshot/my_backup/snapshot_3
```

个会中断快照 程。然后 除 里 行到一半的快照。

从快照恢

一旦 了数据，恢 它就 了：只要在 希望恢 回集群的快照 ID后面加上 `_restore` 即可：

```
POST _snapshot/my_backup/snapshot_1/_restore
```

行 是把 个快照里存有的所有索引都恢 。如果 `snapshot_1` 包括五个索引， 五个都会被恢 到我 集群里。和 `snapshot` API 一 ，我 也可以 希望恢 具体 个索引。

有附加的 用来重命名索引。 个 允 通 模式匹配索引名称，然后通 恢 程提供一个新名 称。如果 想在不替 有数据的前提下，恢 老数据来 内容，或者做其他 理， 个 很有用。 我 从快照里恢 个索引并提供一个替 的名称：

```
POST /_snapshot/my_backup/snapshot_1/_restore
{
  "indices": "index_1", ①
  "rename_pattern": "index_(.+)", ②
  "rename_replacement": "restored_index_$1" ③
}
```

① 只恢 `index_1` 索引，忽略快照中存在的其余索引。

② 所提供的模式能匹配上的正在恢 的索引。

③ 然后把它 重命名成替代的模式。

个会恢 `index_1` 到 及群里，但是重命名成了 `restored_index_1`。

和快照 似， `restore` 命令也会立刻返回，恢 程会在后台 行。如果 更希望 的 HTTP 用阻塞直到恢 完成，添加 `wait_for_completion` :

TIP

```
POST _snapshot/my_backup/snapshot_1/_restore?wait_for_completion=true
```

控恢 操作

从 恢 数据借 了 Elasticsearch 里已有的 行恢 机制。在内部 上，从 恢 分片和从 一个 点恢 是等 的。

如果 想 控恢 的 度， 可以使用 `recovery` API。 是一个通用目的的 API，用来展示 集群中移 着的分片状 。

个 API 可以 在恢 的指定索引 独 用：

```
GET restored_index_3/_recovery
```

或者 看 集群里所有索引，可能包括跟 的恢 程无 的其他分片移 ！：

```
GET /_recovery/
```

出会跟 个 似（注意，根据 集群的活 度， 出可能会 得非常 ！）：

```
{
  "restored_index_3" : {
    "shards" : [ {
      "id" : 0,
      "type" : "snapshot", ①
      "stage" : "index",
      "primary" : true,
      "start_time" : "2014-02-24T12:15:59.716",
      "stop_time" : 0,
      "total_time_in_millis" : 175576,
      "source" : { ②
        "repository" : "my_backup",
        "snapshot" : "snapshot_3",
        "index" : "restored_index_3"
      },
      "target" : {
        "id" : "ryqJ5l05S4-lSFbGntkEkg",
        "hostname" : "my.fqdn",
        "ip" : "10.0.1.7",
        "name" : "my_es_node"
      },
      "index" : {
        "files" : {
          "total" : 73,
          "reused" : 0,
          "recovered" : 69,
          "percent" : "94.5%" ③
        },
        "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
      }
    } ]
  }
}
```

① type 字段告 恢 的本 ； 个分片是在从一个快照恢 。

② source 哈希描述了作 恢 来源的特定快照和 。

③ `percent` 字段 恢 的状 有个概念。一个特定分片目前已 恢 了 94% 的文件；它就快完成了。

出会列出所有目前正在 恢 的索引，然后列出 些索引里的所有分片。一个分片里会有 /停止、持 、恢 百分比、字 数等 。

取消一个恢

要取消一个恢 ，需要 除正在恢 的索引。因 恢 程其 就是分片恢 ，送一个 除索引 API 修改集群状 ，就可以停止恢 程。比如：

```
DELETE /restored_index_3
```

如果 `restored_index_3` 正在恢 中，一个 除命令会停止恢 ，同 除所有已 恢 到集群里的数据。

集群是活着的、呼吸着的生命

一旦 的集群投入生 ，会 他就 始了他自己的一生。Elasticsearch 努力工作来保 集群自足而且 真就在工作。不 一个集群也 要有日常照料和投 ，比如日常 和升 。

Elasticsearch 以非常快的速度 布新版本，行 修 和性能 。保持 的集群采用最新版 是一个好主意。似的，Lucene 持 在 JVM 自身的新的和令人 的 ，意味着需要尽量保持 的 JVM 是最新的。

意味着最好是 有一个 准化的、日常的方案来操作 集群的 重 和升 。升 是一个日常程序，而不是一个需要好多个小 的精 下的年度『惨 』。

似的， 有一个 是很重要的。 的集群做 繁的快照——而且通 行真 恢 的方式定期 些快照！有些 做日常 却从不 他 的恢 机制，直太常 了。通常 会在第一次演 真 恢 的 候 明 的 陷（比如用 不知道 挂 个磁 ）。比起在凌晨 3 点真的 生危机的 候，在日常 中暴露出 些 是更好的。