

目录

Elasticsearch 权威指南

Introduction	1.1
序言	1.2
引言	1.3
谁应该读这这本书	1.3.1
为什么我们要写这本书	1.3.2
如何读这本书	1.3.3
本书导航	1.3.4
本书协议约定	1.3.5
基础入门	1.4
基础知识	1.4.1
安装运行elasticsearch	1.4.1.1
和Elasticsearch交互	1.4.1.2
面向文档	1.4.1.3
适应新环境	1.4.1.4
索引员工文档	1.4.1.5
检索文档	1.4.1.6
轻量检索	1.4.1.7
使用查询表达式搜索	1.4.1.8
更复杂的搜索	1.4.1.9
全文搜索	1.4.1.10
短语搜索	1.4.1.11
高亮搜索	1.4.1.12
分析	1.4.1.13
教程结语	1.4.1.14
分布式特性	1.4.1.15
集群内的原理	1.4.2
空集群	1.4.2.1
集群健康	1.4.2.2
添加索引	1.4.2.3
添加故障转移	1.4.2.4
水平扩容	1.4.2.5
应对故障	1.4.2.6
数据输入和输出	1.4.3

什么是文档	1.4.3.1
文档元数据	1.4.3.2
索引文档	1.4.3.3
取回一个文档	1.4.3.4
检查文档是否存在	1.4.3.5
更新整个文档	1.4.3.6
创建新文档	1.4.3.7
删除文档	1.4.3.8
处理冲突	1.4.3.9
乐观并发控制	1.4.3.10
文档的部分更新	1.4.3.11
取回多个文档	1.4.3.12
代价较小的批量操作	1.4.3.13
分布式文档存储	1.4.4
路由一个文档到一个分片中	1.4.4.1
主分片和副本分片如何交互	1.4.4.2
新建、索引和删除文档	1.4.4.3
取回一个文档	1.4.4.4
局部更新文档	1.4.4.5
多文档模式	1.4.4.6
搜索——最基本的工具	1.4.5
空搜索	1.4.5.1
多索引	1.4.5.2
分页	1.4.5.3
轻量 搜索	1.4.5.4
映射和分析	1.4.6
精确值 VS 全文	1.4.6.1
倒排索引	1.4.6.2
分析与分析器	1.4.6.3
映射	1.4.6.4
复杂核心域类型	1.4.6.5
请求体查询	1.4.7
空查询	1.4.7.1
查询表达式	1.4.7.2

- [1. GitBook](#)
- [2. 声明](#)

1. GitBook

Download PDF: [Elasticsearch:权威指南](#)

2. 声明

本书整理自官方文档

中文版本《[Elasticsearch:权威指南](#)》

英文版《[Elasticsearch: The Definitive Guide](#)》

[Elasticsearch官方最新文档](#)

[kibana使用手册官方中文文档](#)

Copyright © Songbai Yang all right reserved, powered by Gitbook该文件修订时间： 2021-03-07

- **1. 序言**

1. 序言

我仍然清晰地记得那个日子，我发布了这个开源项目第一个版本并在 IRC 聊天室创建一个频道，在那个最紧张的时刻，独自一人，急切地希望和盼望着第一个用户的到来。

第一个跳进 IRC 频道的用户就是 Clint（克林顿），当时我欣喜若狂。好吧…直到我发现 Clint 实际上是 Perl 用户啦，而且还是跟死亡讣告网站打交道。我记得（当时）问自己为什么他不是来自于更“主流”的社区，像 Ruby 或 Python，亦或是一个稍微好点的使用案例。

后来发生的一切都证明，我真是大错特错！Clint 最终对 Elasticsearch 的成功起到了重要作用。他是第一个将 Elasticsearch 投入生产环境的人（还是 0.4 的版本！），初期与 Clint 的交流和沟通对于将 Elasticsearch 塑造成今天的样子非常关键。对于什么是简单，Clint 有独特的见解并且他很少出错，这对 Elasticsearch 从管理、API 设计到日常使用等各个方面的易用性产生了深远的影响。所以公司成立不久，我们想也没想立即就联系 Clint，询问他是否愿意加入我们。

公司成立后，我们做的第一件事就是提供公开培训。很难表达我们当时有多么紧张和担心是否真的有人会报名。

但我们错了。

培训到现在依然很成功，很多主要城市都还有大量的人等待参加。参加培训的成员之中，有一个叫 Zach 年轻小伙吸引了我们注意。我们知道他写过很多关于 Elasticsearch 的博客（并暗自嫉妒他能够用非常简洁的方式来阐述复杂概念的能力），他还编写了一个 PHP 的客户端。然后我们发现 Zach 他还是自掏腰包来参加我们的培训！你真的不能要求更多，于是我们找到 Zach，问他是否愿意加入我们的公司。

Clint 和 Zach 是 Elasticsearch 能否成功的关键。他们是完美的解说家，从简单的上层应用到复杂的（Apache Lucene）底层逻辑。在 Elastic 这里我们非常珍惜这种独特技能。Clint 还负责 Elasticsearch Perl 客户端，而 Zach 则负责 PHP，都是精彩的代码。

最后，两位在 Elasticsearch 项目每天的日常事务中也扮演着重要的角色。Elasticsearch 如此受欢迎的主要原因之一，就是它拥有与用户沟通产生共鸣的能力，Clint 和 Zach 都是这个集体的一份子，这让一切成为可能。

Shay Banon

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间：2021-03-06

- [1. 引言](#)

1. 引言

这个世界已然被数据淹没。多年来，我们系统间流转和产生的大量数据已让我们不知所措。现有的技术都集中在如何解决数据仓库存储以及如何结构化这些数据。这些看上去都挺美好，直到你实际需要基于这些数据实时做决策分析的时候才发现根本不是那么一回事。

Elasticsearch 是一个分布式、可扩展、实时的搜索与数据分析引擎。它能从项目一开始就赋予你的数据以搜索、分析和探索的能力，这是通常没有预料到的。它存在还因为原始数据如果只是躺在磁盘里面根本就毫无用处。

无论你是需要全文搜索，还是结构化数据的实时统计，或者两者结合，这本指南都能帮助你了解其中最基本的概念，从最基本的操作开始学习 Elasticsearch。之后，我们还会逐渐开始探索更加高级的搜索技术，不断提升搜索体验来满足你的需求。

Elasticsearch 不仅仅只是全文搜索，我们还将介绍结构化搜索、数据分析、复杂的人类语言处理、地理位置和对象间关联关系等。我们还将探讨为了充分利用 Elasticsearch 的水平伸缩性，应当如何建立数据模型，以及在生产环境中如何配置和监控你的集群。

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间： 2021-03-06

- [1. 谁应该读这这本书](#)

1. 谁应该读这这本书

这本书是写给任何想要把他们的数据拿来干活做点事情的人。不管你是从头构建一个新项目，还是为了给已有的系统改造换血， Elasticsearch 都能够帮助你解决现有问题和开发新的功能，有些可能是你之前没有想到的功能。

这本书既适合初学者也适合有经验的用户。我们希望你有一定的编程基础，虽然不是必须的，但有用过 SQL 和关系数据库会更佳。我们会从原理解释和基本概念出发，帮助新手在复杂的搜索世界里打下稳固的知识基础。

具有搜索背景的读者也会受益于这本书。有经验的用户将懂得其所熟悉搜索的概念在 Elasticsearch 是如何对应和具体实现的。即使是高级用户，前面几个章节所包含的信息也是非常有用的。

最后，也许你是一名 DevOps，其他部门一直尽可能快的往 Elasticsearch 里面灌数据，而你是那个负责防止 Elasticsearch 服务器起火的消防员。只要用户在规则内行事，Elasticsearch 集群扩容相当轻松。不过你需要知道如何在进入生产环境前搭建一个稳定的集群，还能要在凌晨三点钟能识别出警告信号，以防止灾难发生。前面几章你可能不太感兴趣，但这本书的最后一部分是非常重要的，包含所有你需要知道的用以避免系统崩溃的知识。

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间： 2021-03-06

- 1. 为什么要我们写这本书

1. 为什么要我们写这本书

我们写这本书，因为 Elasticsearch 需要更好的阐述。现有的参考文档是优秀的一前提是你知道你在寻找什么。它假定你已经熟悉信息检索、分布式系统原理、Query DSL 和许多其他相关的概念。

这本书没有这样的假设。它的目的是写一本即便是什么都不懂的初学者（不管是对于搜索还是对于分布式系统）也能拿起它简单看完几章，就能开始搭建一个原型。

我们采取一种基于问题求解的方式：这是一个问题，我该怎么解决？如何对候选方案进行权衡取舍？我们从基础知识开始，循序渐进，每一章都建立在前一章之上，同时提供必要的实用案例和理论解释。

现有的参考文档解决了如何使用这些功能，我们希望这本书解决的是 **为什么** 和 **什么时候** 使用这些功能。

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间：2021-03-06

- [1. 如何读这本书](#)

1. 如何读这本书

Elasticsearch 做了很多努力和尝试来让复杂的事情变得简单，很大程度上来说 Elasticsearch 的成功来源于此。换句话说，搜索以及分布式系统是非常复杂的，不过为了充分利用 Elasticsearch，迟早你也需要掌握它们。

恩，是有点复杂，但不是魔法。我们倾向于认为复杂系统如同神奇的黑盒子，能响应外部的咒语，但是通常里面的工作逻辑很简单。理解了这些逻辑过程你就能驱散魔法，理解内在能够让你更加明确和清晰，而不是寄托于黑盒子做你想要做的。

这本权威指南不仅会帮助你学习 Elasticsearch，而且希望能够带你接触一些更深入、更有趣的话题，如 集群内的原理、分布式文档存储、执行分布式检索 和 分片内部原理，这些虽然不是必要的阅读却能让你深入理解其内在机制。

本书的第一部分应该按章节顺序阅读，因为每一章建立在上一章的基础上（尽管你也可以浏览刚才提到的章节）。后续各章节如 近似匹配 和 部分匹配 相对独立，你可以按需选择性参阅。

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间：2021-03-06

- 1. 本书导航

1. 本书导航

这本书分为七个部分：

- 章节 你知道的, 为了搜索... 到 分片内部原理 主要是介绍 Elasticsearch。介绍了 Elasticsearch 的数据输入输出以及 Elasticsearch 如何处理你的文档数据。如何进行基本的搜索操作和管理你的索引。本章结束你将学会如何将 Elasticsearch 集成到你的应用程序中。章节：集群内的原理、分布式文档存储、执行分布式检索 和 分片内部原理 为附加章节，目的是让你了解分布式处理的过程，不是必读的。
- 章节 结构化搜索 到 控制相关度 带你深入了解搜索，如何借助一些更高级的特性，如邻近词（word proximity）和部分匹配（partial matching）来索引和查询你的数据。你将了解相关度评分是如何工作的以及如何控制它来确保第一页总是返回最佳的搜索结果。
- 章节 开始处理各种语言 到 拼写错误 解决如何有效使用分析器和查询来处理人类语言的棘手问题。我们会从一次简单的语言分析下手，然后逐步深入，如字母表和排序，还会涉及到词干提取、停用词、同义词和模糊匹配。
- 章节 高阶概念 到 Doc Values and Fielddata 讨论聚合（aggregations）和分析，对你的数据进行摘要化和分组来呈现总体趋势。
- 章节 地理坐标点 到 地理形状 介绍 Elasticsearch 支持的两种地理位置检索方式：经纬坐标点和复杂的地理形状（geo-shapes）。
- 章节 关联关系处理 到 扩容设计 谈到了为了高效使用 Elasticsearch，应当如何为你的数据建立模型。在搜索引擎里表达实体间的关系可能不是那么容易，因为它不是用来设计做这个的。这些章节还会阐述如何设计索引来匹配你系统中的数据流。
- 最后，章节 监控 到 部署后 将讨论生产环境上线的重要配置、监控点以及如何诊断以避免出现问题。

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间： 2021-03-06

- **1. 本书协议约定**

1. 本书协议约定

以下是本书中使用的印刷规范：

斜体

表示重点、新的术语或概念。

等宽字体

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

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间： 2021-03-06

- [1. 基础入门](#)

1. 基础入门

Elasticsearch 是一个实时的分布式搜索分析引擎，它能让你以前所未有的速度和规模，去探索你的数据。它被用作全文检索、结构化搜索、分析以及这三个功能的组合：

- Wikipedia 使用 Elasticsearch 提供带有高亮片段的全文搜索，还有 search-as-you-type 和 did-you-mean 的建议。
- 卫报 使用 Elasticsearch 将网络社交数据结合到访客日志中，为它的编辑们提供公众对于新文章的实时反馈。
- Stack Overflow 将地理位置查询融入全文检索中去，并且使用 more-like-this 接口去查找相关的问题和回答。
- GitHub 使用 Elasticsearch 对1300亿行代码进行查询。

Elasticsearch 不仅仅为巨头公司服务。它也帮助了很多初创公司，比如 Datadog 和 Klout，Elasticsearch 帮助他们将想法用原型实现，并转化为可扩展的解决方案。Elasticsearch 能运行在你的笔记本电脑上，或者扩展到数百台服务器上来处理PB级数据。

Elasticsearch 中没有一个单独的组件是全新的或者是革命性的。全文搜索很久之前就已经可以做到了，就像很早之前出现的分析系统和分布式数据库。革命性的成就是在将这些单独的、有用的组件融合到一个单一的、一致的、实时的应用中。对于初学者而言它的门槛相对较低，而当你的技能提升或需求增加时，它也始终能满足你的需求。

如果你现在打开这本书，是因为你拥有数据。除非你准备使用它 做些什么，否则拥有这些数据将没有意义。

不幸的是，大部分数据库在从你的数据中提取可用知识时出乎意料的低效。当然，你可以通过时间戳或精确值进行过滤，但是它们能够全文检索、处理同义词、通过相关性给文档评分么？它们能从同样的数据中生成分析与聚合数据吗？最重要的是，它们能实时地做到上述操作，而不经过大型批处理的任务么？

这就是 Elasticsearch 脱颖而出的地方：Elasticsearch 鼓励你去探索与利用数据，而不是因为查询数据太困难，就让它们烂在数据仓库里面。

Elasticsearch 将成为你最好的朋友。

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间： 2021-03-06

- 1. 基础知识

1. 基础知识

Elasticsearch 是一个开源的搜索引擎，建立在一个全文搜索引擎库 Apache Lucene™ 基础之上。Lucene 可以说是当下最先进、高性能、全功能的搜索引擎库—无论是开源还是私有。

但是 Lucene 仅仅只是一个库。为了充分发挥其功能，你需要使用 Java 并将 Lucene 直接集成到应用程序中。更糟糕的是，您可能需要获得信息检索学位才能了解其工作原理。Lucene 非常 复杂。

Elasticsearch 也是使用 Java 编写的，它的内部使用 Lucene 做索引与搜索，但是它的目的是使全文检索变得简单，通过隐藏 Lucene 的复杂性，取而代之的提供一套简单一致的 RESTful API。

然而，Elasticsearch 不仅仅是 Lucene，并且也仅仅只是一个全文搜索引擎。它可以被下面这样准确的形容：

- 一个分布式的实时文档存储，每个字段 可以被索引与搜索
- 一个分布式实时分析搜索引擎
- 能胜任上百个服务节点的扩展，并支持 PB 级别的结构化或者非结构化数据

Elasticsearch 将所有的功能打包成一个单独的服务，这样你可以通过程序与它提供的简单的 RESTful API 进行通信，可以使用自己喜欢的编程语言充当 web 客户端，甚至可以使用命令行（去充当这个客户端）。

就 Elasticsearch 而言，起步很简单。对于初学者来说，它预设了一些适当的默认值，并隐藏了复杂的搜索理论知识。它 开箱即用。只需最少的理解，你很快就能具有生产力。

随着你知识的积累，你可以利用 Elasticsearch 更多的高级特性，它的整个引擎是可配置并且灵活的。从众多高级特性中，挑选恰当去修饰的 Elasticsearch，使它能解决你本地遇到的问题。

你可以免费下载，使用，修改 Elasticsearch。它在 [Apache 2 license](#) 协议下发布的，这是众多灵活的开源协议之一。Elasticsearch 的源码被托管在 Github 上 github.com/elastic/elasticsearch。如果你想加入我们这个令人惊奇的 contributors 社区，看这里 [Contributing to Elasticsearch](#)。

如果你对 Elasticsearch 有任何相关的问题，包括特定的特性(specific features)、语言客户端(language clients)、插件(plugins)，可以在discuss.elastic.co 加入讨论。

回忆时光

许多年前，一个刚结婚的名叫 Shay Banon 的失业开发者，跟着他的妻子去了伦敦，他的妻子在那里学习厨师。在寻找一个赚钱的工作的时候，为了给他的妻子做一个食谱搜索引擎，他开始使用 Lucene 的一个早期版本。

直接使用 Lucene 是很难的，因此 Shay 开始做一个抽象层，Java 开发者使用它可以很简单的给他们的程序添加搜索功能。他发布了他的第一个开源项目 Compass。

后来 Shay 获得了一份工作，主要是高性能，分布式环境下的内存数据网格。这个对于高性能，实时，分布式搜索引擎的需求尤为突出，他决定重写 Compass，把它变为一个独立的服务并取名 Elasticsearch。

第一个公开版本在2010年2月发布，从此以后，Elasticsearch 已经成为了 Github 上最活跃的项目之一，他拥有超过300名 contributors(目前736名 contributors)。一家公司已经开始围绕 Elasticsearch 提供商业服务，并开发新的特性，但是，Elasticsearch 将永远开源并对所有人可用。

据说，Shay 的妻子还在等着她的食谱搜索引擎...

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间： 2021-03-06

- 1. 安装运行elasticsearch

1. 安装运行elasticsearch

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

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间： 2021-03-07

- [1. 和Elasticsearch交互](#)
 - [1.1. Java Api](#)
 - [1.2. Restful API with Json over HTTP](#)

1. 和Elasticsearch交互

和 Elasticsearch 的交互方式取决于你是否使用 Java。

1.1. Java Api

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

节点客户端 (Node client)

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

传输客户端 (Transport client)

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

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

建议

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

更多的 Java 客户端信息可以在 [Elasticsearch Clients](#) 中找到。

1.2. Restful API with Json over HTTP

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

注意

Elasticsearch 为以下语言提供了官方客户端—Groovy、JavaScript、.NET、PHP、Perl、Python 和 Ruby—还有很多社区提供的客户端和插件, 所有这些都可以在 [Elasticsearch Clients](#) 中找到。

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

```
curl -X<VERB> '<PROTOCOL>://<HOST>:<PORT>/<PATH>?<QUERY_STRING>' -H 'Content-T'
```

被 < > 标记的部件：

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

例如，计算集群中文档的数量，我们可以用这个：

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

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

```
{
  "count" : 0,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  }
}
```

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

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

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

安装运行elasticsearch

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

我们将用缩写格式显示：

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

事实上， kibana DevTool 控制台 也使用这样相同的格式。

Copyright © Songbai Yang all right reserved, powered by Gitbook该文件修订时间： 2021-03-06

- 1. 面向文档
- 2. JSON

1. 面向文档

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

也许有一天你想把这些对象存储在数据库中。使用关系型数据库的行和列存储，这相当于把一个表现力丰富的对象塞到一个非常大的电子表格中：为了适应表结构，你必须设法将这个对象扁平化—通常一个字段对应一列—而且每次查询时又需要将其重新构造为对象。

Elasticsearch 是面向文档的，意味着它存储整个对象或文档。Elasticsearch 不仅存储文档，而且索引每个文档的内容，使之可以被检索。在 Elasticsearch 中，我们对文档进行索引、检索、排序和过滤—而不是对行列数据。这是一种完全不同的思考数据的方式，也是 Elasticsearch 能支持全文检索的原因。

2. JSON

Elasticsearch 使用 JavaScript Object Notation（或者 [JSON](#)）作为文档的序列化格式。JSON 序列化为大多数编程语言所支持，并且已经成为 NoSQL 领域的标准格式。它简单、简洁、易于阅读。

下面这个 JSON 文档代表了一个 user 对象：

```
{  
    "email": "john@smith.com",  
    "first_name": "John",  
    "last_name": "Smith",  
    "info": {  
        "bio": "Eco-warrior and defender of the weak",  
        "age": 25,  
        "interests": [ "dolphins", "whales" ]  
    },  
    "join_date": "2014/05/01"  
}
```

虽然原始的 user 对象很复杂，但这个对象的结构和含义在 JSON 版本中都得到了体现和保留。在 Elasticsearch 中将对象转化为 JSON 后构建索引要比在一个扁平的表结构中要简单的多。

注意

几乎所有的语言都有可以将任意的数据结构或对象转化成 JSON 格式的模块，只是细节各不相同。具体请查看 `serialization` 或者 `marshalling` 这两个处理 JSON 的模块。官方 Elasticsearch 客户端自动为您提供 JSON 转化。

- 1. 适应新环境
- 2. 创建一个雇员目录

1. 适应新环境

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

我们将一并介绍一些新的技术术语，即使无法立即全部理解它们也无妨，因为在本书后续内容中，我们将继续深入介绍这里提到的所有概念。

接下来尽情享受 Elasticsearch 探索之旅。

2. 创建一个雇员目录

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

- 支持包含多值标签、数值、以及全文本的数据
- 检索任一员工的完整信息
- 允许结构化搜索，比如查询 30 岁以上的员工
- 允许简单的全文搜索以及较复杂的短语搜索
- 支持在匹配文档内容中高亮显示搜索片段
- 支持基于数据创建和管理分析仪表盘

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间： 2021-03-07

- 1. 索引员工文档

1. 索引员工文档

第一个业务需求是存储员工数据。这将会以 员工文档 的形式存储：一个文档代表一个员工。存储数据到 Elasticsearch 的行为叫做 索引，但在索引一个文档之前，需要确定将文档存储在哪里。

一个 Elasticsearch 集群可以 包含多个 索引，相应的每个索引可以包含多个类型 (在6版本之后一个索引表只允许包含一个类型)。一个类型存储着多个 文档，每个文档又有 多个 属性。

Index Versus Index Versus Index

你也许已经注意到 索引 这个词在 Elasticsearch 语境中有多种含义，这里有必要做一些说明：

索引（名词）：

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

索引（动词）：

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

倒排索引：

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

默认的，一个文档中的每一个属性都是 被索引 的（有一个倒排索引）和可搜索的。一个没有倒排索引的属性是不能被搜索到的。我们将在 [倒排索引](#) 讨论倒排索引的更多细节。

对于员工目录，我们将做如下操作：

- 每个员工索引一个文档，文档包含该员工的所有信息。
- 每个文档都将是 *employee* 类型。
- 该类型位于 索引 *megacorp* 内。
- 该索引保存在我们的 Elasticsearch 集群中。

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

```
PUT /megacorp/employee/1
{
  "first_name" : "John",
  "last_name" : "Smith",
  "age" : 25,
  "about" : "I love to go rock climbing",
  "interests": [ "sports", "music" ]
}
```

注意，路径 /megacorp/employee/1 包含了三部分的信息：

megacorp
索引名称
employee
类型名称
1
特定雇员的ID

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

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

进行下一步前，让我们增加更多的员工信息到目录中：

```
PUT /megacorp/employee/2
{
    "first_name" : "Jane",
    "last_name" : "Smith",
    "age" : 32,
    "about" : "I like to collect rock albums",
    "interests": [ "music" ]
}

PUT /megacorp/employee/3
{
    "first_name" : "Douglas",
    "last_name" : "Fir",
    "age" : 35,
    "about": "I like to build cabinets",
    "interests": [ "forestry" ]
}
```

Copyright © Songbai Yang all right reserved, powered by Gitbook 该文件修订时间：2021-03-07

- 1. 检索文档

1. 检索文档

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

这在 Elasticsearch 中很简单。简单地执行一个 HTTP GET 请求并指定文档的地址——索引库、类型和ID。使用这三个信息可以返回原始的 JSON 文档：

```
GET /megacorp/employee/1
```

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

```
{
  "_index": "megacorp",
  "_type": "employee",
  "_id": "1",
  "_version": 1,
  "found": true,
  "_source": {
    "first_name": "John",
    "last_name": "Smith",
    "age": 25,
    "about": "I love to go rock climbing",
    "interests": [ "sports", "music" ]
  }
}
```

建议

将 HTTP 命令由 PUT 改为 GET 可以用来检索文档，同样的，可以用 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 。

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间： 2021-03-06

- 1. 轻量检索

1. 轻量检索

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

第一个尝试的几乎是最简单的搜索了。我们使用下列请求来搜索所有雇员：

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

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

```
{
  "took":      6,
  "timed_out": false,
  "_shards": { ... },
  "hits": {
    "total":      3,
    "max_score":  1,
    "hits": [
      {
        "_index":      "megacorp",
        "_type":       "employee",
        "_id":        "3",
        "_score":     1,
        "_source": {
          "first_name": "Douglas",
          "last_name":  "Fir",
          "age":        35,
          "about":      "I like to build cabinets",
          "interests": [ "forestry" ]
        }
      },
      {
        "_index":      "megacorp",
        "_type":       "employee",
        "_id":        "1",
        "_score":     1,
        "_source": {
          "first_name": "John",
          "last_name":  "Smith",
          "age":        25,
          "about":      "I love to go rock climbing",
          "interests": [ "sports", "music" ]
        }
      },
      {
        "_index":      "megacorp",
        "_type":       "employee",
        "_id":        "2",
        "_score":     1,
        "_source": {
          "first_name": "Jane",
          "last_name":  "Smith",
          "age":        32,
          "about":      "I like to collect rock albums",
          "interests": [ "music" ]
        }
      }
    ]
  }
}
```

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

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

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

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

```
{  
  ...  
  "hits": {  
    "total": 2,  
    "max_score": 0.30685282,  
    "hits": [  
      {  
        ...  
        "_source": {  
          "first_name": "John",  
          "last_name": "Smith",  
          "age": 25,  
          "about": "I love to go rock climbing",  
          "interests": [ "sports", "music" ]  
        }  
      },  
      {  
        ...  
        "_source": {  
          "first_name": "Jane",  
          "last_name": "Smith",  
          "age": 32,  
          "about": "I like to collect rock albums",  
          "interests": [ "music" ]  
        }  
      }  
    ]  
  }  
}
```

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间： 2021-03-06

- 1. 使用查询表达式搜索

1. 使用查询表达式搜索

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

领域特定语言（DSL），使用 JSON 构造了一个请求。我们可以像这样重写之前的查询所有名为 Smith 的搜索：

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

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

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间： 2021-03-06

- 1. 更复杂的搜索

1. 更复杂的搜索

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

```
GET /megacorp/employee/_search
{
  "query": {
    "bool": {
      "must": {
        "match": {
          "last_name": "smith" (a)
        }
      },
      "filter": {
        "range": {
          "age": { "gt": 30 } (b)
        }
      }
    }
  }
}
```

(a) 这部分与我们之前使用的 match 查询一样。

(b) 这部分是一个 range 过滤器，它能找到年龄大于 30 的文档，其中 gt 表示大于(great than)。

目前无需太多担心语法问题，后续会更详细地介绍。只需明确我们添加了一个过滤器用于执行一个范围查询，并复用之前的 match 查询。现在结果只返回了一名员工，叫 Jane Smith，32 岁。

```
{
  ...
  "hits": {
    "total": 1,
    "max_score": 0.30685282,
    "hits": [
      {
        ...
        "_source": {
          "first_name": "Jane",
          "last_name": "Smith",
          "age": 32,
          "about": "I like to collect rock albums",
          "interests": [ "music" ]
        }
      }
    ]
  }
}
```

- 1. 全文搜索

1. 全文搜索

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

搜索下所有喜欢攀岩（rock climbing）的员工：

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

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

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

(a) 相关性得分

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

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

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

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间：2021-03-07

- 1. 短语搜索

1. 短语搜索

找出一个属性中的独立单词是没有问题的，但有时候想要精确匹配一系列单词或者短语。比如，我们想执行这样一个查询，仅匹配同时包含“rock”和“climbing”，并且二者以短语“rock climbing”的形式紧挨着的雇员记录。

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

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

毫无悬念，返回结果仅有 John Smith 的文档。

```
{
  ...
  "hits": {
    "total": 1,
    "max_score": 0.23013961,
    "hits": [
      {
        ...
        "_score": 0.23013961,
        "_source": {
          "first_name": "John",
          "last_name": "Smith",
          "age": 25,
          "about": "I love to go rock climbing",
          "interests": [ "sports", "music" ]
        }
      }
    ]
  }
}
```

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间：2021-03-06

- 1. 高亮搜索

1. 高亮搜索

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

再次执行前面的查询，并增加一个新的 highlight 参数：

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

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

```
{
  ...
  "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>" ( a)
          ]
        }
      }
    ]
  }
}
```

(a) 原始文本中的高亮片段

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

Copyright © Songbai Yang all right reserved, powered by Gitbook该文件修订时间： 2021-03-06

- 1. 分析

1. 分析

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

举个例子，挖掘出员工中最受欢迎的兴趣爱好：

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

暂时忽略掉语法，直接看看结果：

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

可以看到，两位员工对音乐感兴趣，一位对林业感兴趣，一位对运动感兴趣。这些聚合的结果数据并非预先统计，而是根据匹配当前查询的文档即时生成的。如果想知道叫 Smith 的员工中最受欢迎的兴趣爱好，可以直接构造一个组合查询：

安装运行elasticsearch

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

all_interests 聚合已经变为只包含匹配查询的文档：

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

聚合还支持分级汇总。比如，查询特定兴趣爱好员工的平均年龄：

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

得到的聚合结果有点儿复杂，但理解起来还是很简单的：

```
...
"all_interests": {
  "buckets": [
    {
      "key": "music",
      "doc_count": 2,
      "avg_age": {
        "value": 28.5
      }
    },
    {
      "key": "forestry",
      "doc_count": 1,
      "avg_age": {
        "value": 35
      }
    },
    {
      "key": "sports",
      "doc_count": 1,
      "avg_age": {
        "value": 25
      }
    }
  ]
}
```

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

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

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间： 2021-03-06

- 1. 教程结语

1. 教程结语

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

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

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间：2021-03-06

- 1. 分布式特性

1. 分布式特性

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

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

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

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

当阅读本书时，将会遇到有关 Elasticsearch 分布式特性的补充章节。这些章节将介绍有关集群扩容、故障转移(集群内的原理)、应对文档存储(分布式文档存储)、执行分布式搜索(执行分布式检索)，以及分区 (shard) 及其工作原理(分片内部原理)。

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

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间： 2021-03-06

- 1. 集群内的原理

1. 集群内的原理

补充章节

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

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

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

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

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

本章将讲述如何按需配置集群、节点和分片，并在硬件故障时确保数据安全。

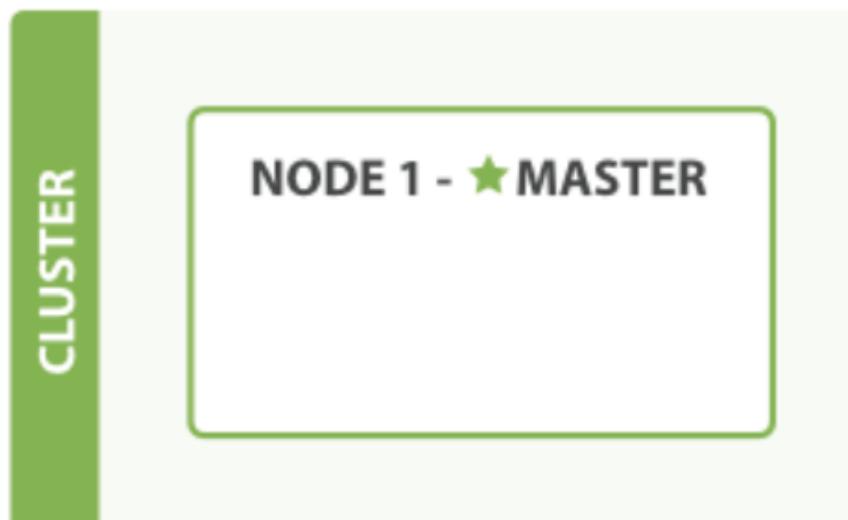
Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间： 2021-03-06

- 1. 空集群

1. 空集群

如果我们启动了一个单独的节点，里面不包含任何的数据和索引，那我们的集群看起来就是一个 Figure 1, “包含空内容节点的集群”。

Figure 1, “包含空内容节点的集群”



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

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

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

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间：2021-03-06

- 1. 集群健康

1. 集群健康

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

```
GET /_cluster/health
```

在一个不包含任何索引的空集群中，它将会有一个类似于如下所示的返回内容：

```
{
  "cluster_name": "elasticsearch",
  "status": "yellow", (a)
  "timed_out": false,
  "number_of_nodes": 10,
  "number_of_data_nodes": 9,
  "active_primary_shards": 6921,
  "active_shards": 13661,
  "relocating_shards": 0,
  "initializing_shards": 0,
  "unassigned_shards": 181,
  "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": 98.6923854934258
}
```

(a)`status` 字段是我们最关心的。

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

green

所有的主分片和副本分片都正常运行。

yellow

所有的主分片都正常运行，但不是所有的副本分片都正常运行。

red

有主分片没能正常运行。

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

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间： 2021-03-06

- 1. 添加索引

1. 添加索引

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

一个 分片 是一个底层的工作单元，它仅保存了全部数据中的一部分。在分片内部机制中，我们将详细介绍分片是如何工作的，而现在我们只需知道一个分片是一个 Lucene 的实例，以及它本身就是一个完整的搜索引擎。我们的文档被存储和索引到分片内，但是应用程序是直接与索引而不是与分片进行交互。

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

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

注意

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

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

在索引建立的时候就已经确定了主分片数，但是副本分片数可以随时修改。

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

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

我们的集群现在是Figure 2，“拥有一个索引的单节点集群”。所有3个主分片都被分配在 Node 1。

Figure 2. 拥有一个索引的单节点集群



如果我们现在查看[集群健康](#), 我们将看到如下内容:

```
{  
  "cluster_name": "elasticsearch",  
  "status": "yellow",  ( a)  
  "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,  ( b)  
  "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  
}
```

- (a)集群 `status` 值为 `yellow`。
- (b)没有被分配到任何节点的副本数。

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

当前我们的集群是正常运行的，但是在硬件故障时有丢失数据的风险。

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间: 2021-03-06

- 1. 添加故障转移

1. 添加故障转移

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

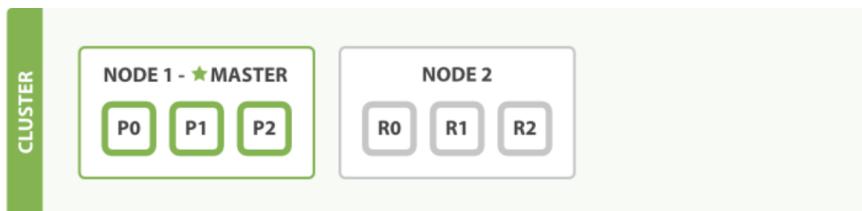
启动第二个节点

为了测试第二个节点启动后的情况，你可以在同一个目录内，完全依照启动第一个节点的方式来启动一个新节点（参考安装并运行 Elasticsearch）。多个节点可以共享同一个目录。

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

如果启动了第二个节点，我们的集群将会如Figure 3，“拥有两个节点的集群——所有主分片和副本分片都已被分配”所示。

Figure 3. 拥有两个节点的集群——所有主分片和副本分片都已被分配



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

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

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

```
{  
  "cluster_name": "elasticsearch",  
  "status": "green",  ( a)  
  "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  
}
```

(a)集群 status 值为 green 。

我们的集群现在不仅仅是正常运行的，并且还处于始终可用 的状态。

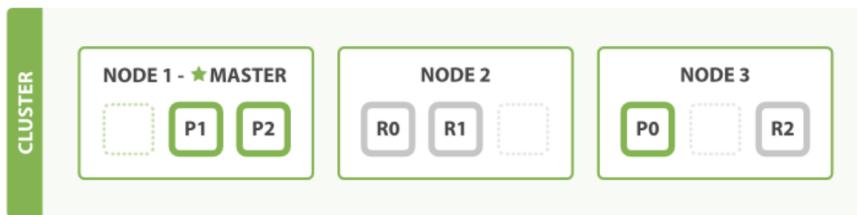
Copyright © Songbai Yang all right reserved, powered by Gitbook该文件修订时间： 2021-03-06

- 1. 水平扩容
- 2. 更多的扩容

1. 水平扩容

怎样为我们的正在增长中的应用程序按需扩容呢？当启动了第三个节点，我们的集群将会看起来如Figure 4，“拥有三个节点的集群——为了分散负载而对分片进行重新分配”所示。

Figure 4. 拥有三个节点的集群——为了分散负载而对分片进行重新分配



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

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

2. 更多的扩容

但是如果我们要扩容超过6个节点怎么办呢？

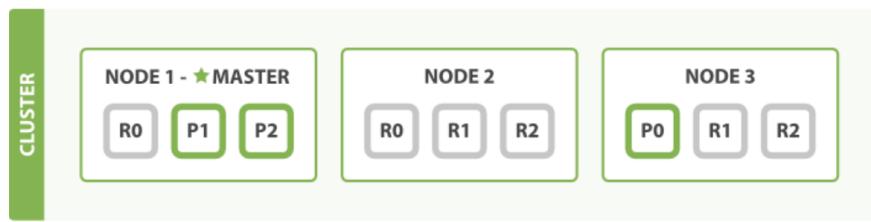
主分片的数目在索引创建时就已经确定了下来。实际上，这个数目定义了这个索引能够存储的最大数据量。（实际大小取决于你的数据、硬件和使用场景。）但是，读操作——搜索和返回数据——可以同时被主分片或副本分片所处理，所以当你拥有越多的副本分片时，也将拥有越高的吞吐量。

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

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

如Figure 5，“将参数 number_of_replicas 调大到 2”所示， blogs 索引现在拥有9个分片：3个主分片和6个副本分片。这意味着我们可以将集群扩容到9个节点，每个节点上一个分片。相比原来3个节点时，集群搜索性能可以提升 3 倍。

Figure 5. 将参数 number_of_replicas 调大到 2



注意

当然，如果只是在相同节点数目的集群上增加更多的副本分片并不能提高性能，因为每个分片从节点上获得的资源会变少。你需要增加更多的硬件资源来提升吞吐量。但是更多的副本分片数提高了数据冗余量：按照上面的节点配置，我们可以在失去2个节点的情况下不丢失任何数据。

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间： 2021-03-06

- 1. 应对故障

1. 应对故障

我们之前说过 Elasticsearch 可以应对节点故障，接下来让我们尝试下这个功能。如果我们关闭第一个节点，这时集群的状态为Figure 6, “关闭了一个节点后的集群”

Figure 6. 关闭了一个节点后的集群



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

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

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

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

如果我们重新启动 Node 1，集群可以将缺失的副本分片再次进行分配，那么集群的状态也将如Figure 5, “将参数 number_of_replicas 调大到 2”所示。如果 Node 1 依然拥有着之前的分片，它将尝试去重用它们，同时仅从主分片复制发生了修改的数据文件。

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

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间：2021-03-06

- 1. 数据输入和输出

1. 数据输入和输出

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

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

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

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

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

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

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

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

在Elasticsearch中，每个字段的所有数据都是默认被索引的。即每个字段都有为了快速检索设置的专用倒排索引。而且，不像其他多数的数据库，它能在同一个查询中使用所有这些倒排索引，并以惊人的速度返回结果。

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

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间：2021-03-06

- 1. 什么是文档

1. 什么是文档

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

```
{  
    "name": "John Smith",  
    "age": 42,  
    "confirmed": true,  
    "join_date": "2014-06-01",  
    "home": {  
        "lat": 51.5,  
        "lon": 0.1  
    },  
    "accounts": [  
        {  
            "type": "facebook",  
            "id": "johnsmith"  
        },  
        {  
            "type": "twitter",  
            "id": "johnsmith"  
        }  
    ]  
}
```

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

⚠ 警示

字段的名字可以是任何合法的字符串，但不可以包含英文句号(.)。

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间：2021-03-07

- **1. 文档元数据**
 - **1.1. _index**
 - **1.2. _type**
 - **1.3. _id**
 - **1.4. 其他元数据**

1. 文档元数据

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

`_index`
文档在哪存放

`_type`
文档表示的对象类别

`_id`
文档唯一标识

1.1. `_index`

一个索引应该是因共同的特性被分组到一起的文档集合。例如，你可能存储所有的产品在索引 `products` 中，而存储所有销售的交易到索引 `sales` 中。虽然也允许存储不相关的数据到一个索引中(6.x版本之前可以这么做)，但这通常看作是一个反模式的做法。

建议

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

我们将在 索引管理 介绍如何自行创建和管理索引，但现在我们将让 Elasticsearch 帮我们创建索引。所有需要我们做的就是选择一个索引名，这个名字必须小写，不能以下划线开头，不能包含逗号。我们用 `website` 作为索引名举例。

1.2. `_type`

数据可能在索引中只是松散的组合在一起，但是通常明确定义一些数据中的子分区是有用的。例如，所有的产品都放在一个索引中，但是你有许多不同的产品类别，比如 `"electronics"`、`"kitchen"` 和 `"lawn-care"`。这些文档共享一种相同的（或非常相似）的模式：他们有一个标题、描述、产品代码和价格。他们只是正好属于“产品”下的一些子类。Elasticsearch 公开了一个称为 `types`（类型）的特性，它允许您在索引中对数据进行逻辑分区。不同 `types` 的文档可能有不同的字段，但最好能够非常相似。我们将在 类型和映射 中更多的讨论关于 `types` 的一些应用和限制。一个 `_type` 命名可以是大写或者小写，但是不能以下划线或者句号开头，不应该包含逗号，并且长度限制为256个字符。我们使用 `blog` 作为类型名举例。

⚠ 警示

在 ES6.x 之后已经不支持单个索引表多个 type， type 已经没有任何实际意义了，在 Elasticsearch7.x 中 type 默认值为 doc

1.3. _id

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

1.4. 其他元数据

还有一些其他的元数据元素，他们在 [类型和映射](#) 进行了介绍。通过前面已经列出的元数据元素，我们已经能存储文档到 Elasticsearch 中并通过 ID 检索它—换句话说，使用 Elasticsearch 作为文档的存储介质。

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间： 2021-03-07

- 1. 索引文档
 - 1.1. 使用自定义的ID
 - 1.2. 自动生成id

1. 紴索引文档

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

1.1. 使用自定义的ID

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

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

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

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

Elasticsearch 响应体如下所示：

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

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

在 Elasticsearch 中每个文档都有一个版本号。当每次对文档进行修改时（包括删除），`_version` 的值会递增。在处理冲突中，我们讨论了怎样使用 `_version` 号码确保你的应用程序中的一部分修改不会覆盖另一部分所做的修改。

1.2. 自动生成id

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

现在该 URL 只需包含 _index 和 _type :

```
POST /website/blog/
{
  "title": "My second blog entry",
  "text": "Still trying this out...",
  "date": "2014/01/01"
}
```

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

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

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

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间： 2021-03-07

- [1. 取回一个文档](#)
- [2. 返回文档的一部分](#)

1. 取回一个文档

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

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

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

```
{
  "_index": "website",
  "_type": "blog",
  "_id": "123",
  "_version": 1,
  "found": true,
  "_source": {
    "title": "My first blog entry",
    "text": "Just trying this out...",
    "date": "2014/01/01"
  }
}
```

注意

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

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

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

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

显示响应头部的响应体现在类似这样：

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

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

2. 返回文档的一部分

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

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

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

```
{
  "_index": "website",
  "_type": "blog",
  "_id": "123",
  "_version": 1,
  "found": true,
  "_source": {
    "title": "My first blog entry",
    "text": "Just trying this out..."
  }
}
```

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

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

那么返回的内容如下所示：

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

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间：2021-03-07

- 1. 检查文档是否存在

1. 检查文档是否存在

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

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

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

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

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

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

```
HTTP/1.1 404 Not Found
Content-Type: text/plain; charset=UTF-8
Content-Length: 0
```

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

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间: 2021-03-08

- 1. 更新整个文档

1. 更新整个文档

在 Elasticsearch 中文档是不可改变的，不能修改它们。相反，如果想要更新现有的文档，需要重建索引或者进行替换，我们可以使用相同的 index API 进行实现，在 [索引文档](#) 中已经进行了讨论。

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

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

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

(a) created 标志设置成 false，是因为相同的索引、类型和 ID 的文档已经存在。

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

在本章的后面部分，我们会介绍 update API，这个 API 可以用于 partial updates to a document。虽然它似乎对文档直接进行了修改，但实际上 Elasticsearch 按前述完全相同方式执行以下过程：

- 从旧文档构建 JSON
- 更改该 JSON
- 删除旧文档
- 索引一个新文档

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

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间：2021-03-08

- 1. 创建新文档

1. 创建新文档

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

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

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

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

第一种方法使用 `op_type` 查询-字符串参数：

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

第二种方法是在 URL 末端使用 `_create`：

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

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

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

```
{  
  "error": {  
    "root_cause": [  
      {  
        "type": "document_already_exists_exception",  
        "reason": "[blog][123]: document already exists",  
        "shard": "0",  
        "index": "website"  
      }  
    ],  
    "type": "document_already_exists_exception",  
    "reason": "[blog][123]: document already exists",  
    "shard": "0",  
    "index": "website"  
  },  
  "status": 409  
}
```

- 1. 删除文档

1. 删除文档

删除文档的语法和我们所知道的规则相同，只是使用 DELETE 方法

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

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

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

如果文档没有找到，我们将得到 404 Not Found 的响应码和类似这样的响应体：
`$xslt` 即使文档不存在（Found 是 false），_version 值仍然会增加。这是 Elasticsearch 内部记录本的一部分，用来确保这些改变在跨多节点时以正确的顺序执行。

Note

正如已经在[更新整个文档](#)中提到的，删除文档不会立即将文档从磁盘中删除，只是将文档标记为已删除状态。随着你不断的索引更多的数据，Elasticsearch 将会在后台清理标记为已删除的文档。

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间：2021-03-08

- 1. 处理冲突

1. 处理冲突

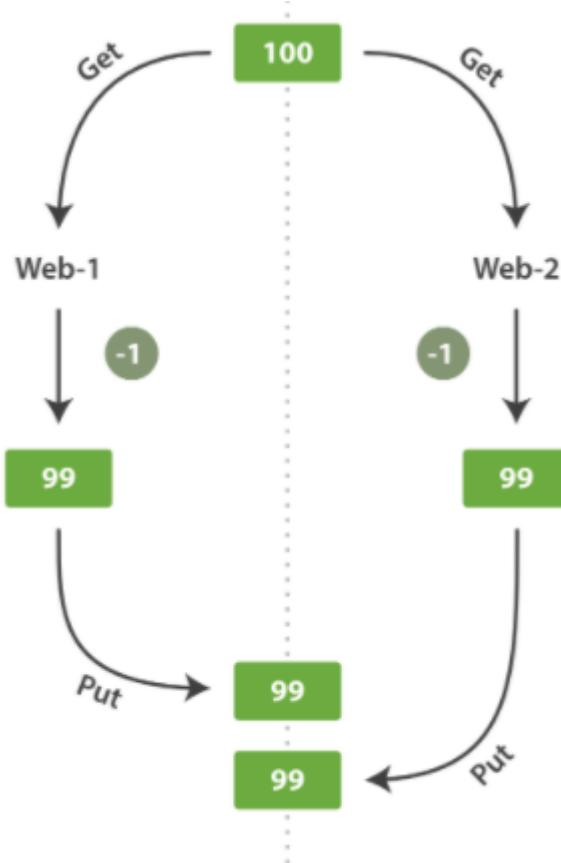
当我们使用 index API 更新文档，可以一次性读取原始文档，做我们的修改，然后重新索引整个文档。最近的索引请求将获胜：无论最后哪一个文档被索引，都将被唯一存储在 Elasticsearch 中。如果其他人同时更改这个文档，他们的更改将丢失。

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

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

有一天，管理层决定做一次促销。突然地，我们一秒要卖好几个商品。假设有两个 web 程序并行运行，每一个都同时处理所有商品的销售，如图 Figure 7，“Consequence of no concurrency control” 所示。

Figure 7. Consequence of no concurrency control



web_1 对 stock_count 所做的更改已经丢失，因为 web_2 不知道它的 stock_count 的拷贝已经过期。结果我们会认为有超过商品的实际数量的库存，因为卖给顾客的库存商品并不存在，我们将让他们非常失望。

变更越频繁，读数据和更新数据的间隙越长，也就越可能丢失变更。

在数据库领域中，有两种方法通常被用来确保并发更新时变更不会丢失：

悲观并发控制

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

乐观并发控制

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

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间： 2021-03-08

- 1. 乐观并发控制
- 2. 通过外部系统使用版本控制

1. 乐观并发控制

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

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

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

让我们创建一个新的博客文章：

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

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

首先我们检索文档：

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

响应体包含相同的 `_version` 版本号 1：

```
{
  "_index": "website",
  "_type": "blog",
  "_id": "1",
  "_version": 1,
  "found": true,
  "_source": {
    "title": "My first blog entry",
    "text": "Just trying this out..."
  }
}
```

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

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

(a) 我们想这个在我们索引中的文档只有现在的 `_version` 为 1 时，本次更新才能成功。

此请求成功，并且响应体告诉我们 `_version` 已经递增到 2：

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

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

```
{
  "error": {
    "root_cause": [
      {
        "type": "version_conflict_engine_exception",
        "reason": "[blog][1]: version conflict, current [2], provided [1]",
        "index": "website",
        "shard": "3"
      }
    ],
    "type": "version_conflict_engine_exception",
    "reason": "[blog][1]: version conflict, current [2], provided [1]",
    "index": "website",
    "shard": "3"
  },
  "status": 409
}
```

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

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

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

2. 通过外部系统使用版本控制

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

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

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

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

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

```
PUT /website/blog/2?version=5&version_type=external
{
  "title": "My first external blog entry",
  "text": "Starting to get the hang of this..."
}
```

在响应中， 我们能看到当前的 `_version` 版本号是 5 :

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

现在我们更新这个文档， 指定一个新的 version 号是 10 :

```
PUT /website/blog/2?version=10&version_type=external
{
  "title": "My first external blog entry",
  "text": "This is a piece of cake..."
}
```

请求成功并将当前 `_version` 设为 10 :

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

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

Copyright © Songbai Yang all right reserved, powered by Gitbook该文件修订时间： 2021-03-08

- 1. 文档的部分更新
- 2. 使用脚本部分更新文档
- 3. 更新的文档可能尚不存在
- 4. 更新和冲突

1. 文档的部分更新

在更新整个文档，我们已经介绍过更新一个文档的方法是检索并修改它，然后重新索引整个文档，这的确如此。然而，使用 `update API` 我们还可以部分更新文档，例如在某个请求时对计数器进行累加。

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

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

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

如果请求成功，我们看到类似于 `index` 请求的响应：

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

检索文档显示了更新后的 `_source` 字段：

```
{
  "_index": "website",
  "_type": "blog",
  "_id": "1",
  "_version": 3,
  "found": true,
  "_source": {
    "title": "My first blog entry",
    "text": "Starting to get the hang of this...",
    "tags": [ "testing" ], ( a )
    "views": 0 ( a )
  }
}
```

(a) 新的字段已被添加到 `_source` 中

2. 使用脚本部分更新文档

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

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

用 Groovy 脚本编程

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

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

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

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

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

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

你可以在 scripting reference documentation 获取更多关于脚本的资料。

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

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

获取文档并显示最后两次请求的效果：

```
{
  "_index": "website",
  "_type": "blog",
  "_id": "1",
  "_version": 5,
  "found": true,
  "_source": {
    "title": "My first blog entry",
    "text": "Starting to get the hang of this...",
    "tags": ["testing", "search"], (a)
    "views": 1 (b)
  }
}
```

(a) search 标签已追加到 tags 数组中。
(b) views 字段已递增。

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

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

3. 更新的文档可能尚不存在

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

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

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

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

4. 更新和冲突

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

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

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

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

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

(a) 失败之前重试该更新5次。

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

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间：2021-03-09

- 1. 取回多个文档

1. 取回多个文档

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

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

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

该响应体也包含一个 `docs` 数组，对于每一个在请求中指定的文档，这个数组中都包含有一个对应的响应，且顺序与请求中的顺序相同。其中的每一个响应都和使用单个 `get request` 请求所得到的响应体相同：

```
{  
  "docs" : [  
    {  
      "_index" : "website",  
      "_id" : "2",  
      "_type" : "blog",  
      "found" : true,  
      "_source" : {  
        "text" : "This is a piece of cake...",  
        "title" : "My first external blog entry"  
      },  
      "_version" : 10  
    },  
    {  
      "_index" : "website",  
      "_id" : "1",  
      "_type" : "pageviews",  
      "found" : true,  
      "_version" : 2,  
      "_source" : {  
        "views" : 2  
      }  
    }  
  ]  
}
```

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

你仍然可以通过单独请求覆盖这些值：

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

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

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

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

```
{  
  "docs" : [  
    {  
      "_index" : "website",  
      "_type" : "blog",  
      "_id" : "2",  
      "_version" : 10,  
      "found" : true,  
      "_source" : {  
        "title": "My first external blog entry",  
        "text": "This is a piece of cake..."  
      }  
    },  
    {  
      "_index" : "website",  
      "_type" : "blog",  
      "_id" : "1",  
      "found" : false  
    }  
  ]  
}
```

(a) 未找到该文档

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

Note

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

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间： 2021-03-09

- **1. 代价较小的批量操作**
 - **1.1. 不要重复指定Index和Type**
 - **1.2. 多大是太大?**

1. 代价较小的批量操作

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

`bulk` 与其他的请求体格式稍有不同，如下所示：

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

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

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

Tip

在 [为什么是有趣的格式？](#) 中，我们解释为什么 bulk API 使用这种格式。

`action/metadata` 行指定 哪一个文档 做 什么操作 。

`action` 必须是以下选项之一：

`create`

如果文档不存在，那么就创建它。详情请见 [创建新文档](#)。

`index`

创建一个新文档或者替换一个现有的文档。详情请见 [索引文档](#) 和 [更新整个文档](#)。

`update`

部分更新一个文档。详情请见 [文档的部分更新](#)。

`delete`

删除一个文档。详情请见 [删除文档](#)。

`metadata`

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

例如，一个 `delete` 请求看起来是这样的：

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

`request body` 行由文档的 `_source` 本身组成—文档包含的字段和值。它是 `index` 和 `create` 操作所必需的，这是有道理的：你必须提供文档以索引。它也是 `update` 操作所必需的，并且应该包含你传递给 `update API` 的相同请求体：`doc`、`upsert`、`script` 等等。删除操作不需要 `request body` 行。

```
{ "create": { "_index": "website", "_type": "blog", "_id": "123" }}  
{ "title": "My first blog post" }
```

如果不指定 _id , 将会自动生成一个 ID :

```
{ "index": { "_index": "website", "_type": "blog" }}  
{ "title": "My second blog post" }
```

为了把所有的操作组合在一起，一个完整的 bulk 请求 有以下形式:

```
POST /_bulk  
{ "delete": { "_index": "website", "_type": "blog", "_id": "123" }} (a)  
{ "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"} } (b)
```

(a) 请注意 delete 动作不能有请求体, 它后面跟着的是另外一个操作。

(b) 谨记最后一个换行符不要落下。

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

```
{
  "took": 4,
  "errors": false, ( a )
  "items": [
    { "delete": {
        "_index": "website",
        "_type": "blog",
        "_id": "123",
        "_version": 2,
        "status": 200,
        "found": true
      }},
    { "create": {
        "_index": "website",
        "_type": "blog",
        "_id": "123",
        "_version": 3,
        "status": 201
      }},
    { "create": {
        "_index": "website",
        "_type": "blog",
        "_id": "EiwfApScQiiy7TIKFxRCTw",
        "_version": 1,
        "status": 201
      }},
    { "update": {
        "_index": "website",
        "_type": "blog",
        "_id": "123",
        "_version": 4,
        "status": 200
      }}
  ]
}
```

(a) 所有的子请求都成功完成。

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

```
POST /_bulk
{ "create": { "_index": "website", "_type": "blog", "_id": "123" }}
{ "title": "Cannot create - it already exists" }
{ "index": { "_index": "website", "_type": "blog", "_id": "123" }}
{ "title": "But we can update it" }
```

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

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

- (a) 一个或者多个请求失败。
- (b) 这个请求的HTTP状态码报告为 409 CONFLICT 。
- (c) 解释为什么请求失败的错误信息。
- (d) 第二个请求成功，返回 HTTP 状态码 200 OK 。

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

1.1. 不要重复指定Index和Type

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

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

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

```
POST /website/log/_bulk
{ "index": {}}
{ "event": "User logged in" }
{ "index": { "_type": "blog" }}
{ "title": "Overriding the default type" }
```

1.2. 多大是太大？

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

档的大小和复杂度、索引和搜索的负载的整体情况。

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

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

Copyright © Songbai Yang all right reserved, powered by Gitbook该文件修订时间： 2021-03-09

- 1. 分布式文档存储

1. 分布式文档存储

在前面的章节，我们介绍了如何索引和查询数据，不过我们忽略了很多底层的技术细节，例如文件是如何分布到集群的，又是如何从集群中获取的。Elasticsearch 本意就是隐藏这些底层细节，让我们好专注在业务开发中，所以其实你不必了解这么深入也无妨。

在这个章节中，我们将深入探索这些核心的技术细节，这能帮助你更好地理解数据如何被存储到这个分布式系统中。

NOTE

这个章节包含了一些高级话题，上面也提到过，就算你不记住和理解所有的细节仍然能正常使用 Elasticsearch。如果你有兴趣的话，这个章节可以作为你的课外兴趣读物，扩展你的知识面。

如果你在阅读这个章节的时候感到很吃力，也不用担心。这个章节仅仅只是用来告诉你 Elasticsearch 是如何工作的，将来在工作中如果你需要用到这个章节提供的知识，可以再回过头来翻阅。

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间： 2021-03-09

- 1. 路由一个文档到一个分片中

1. 路由一个文档到一个分片中

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

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

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

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

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

NOTE

你可能觉得由于 Elasticsearch 主分片数量是固定的会使索引难以进行扩容。实际上当你需要时有很多技巧可以轻松实现扩容。我们将会在扩容设计一章中提到更多有关水平扩展的内容。

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

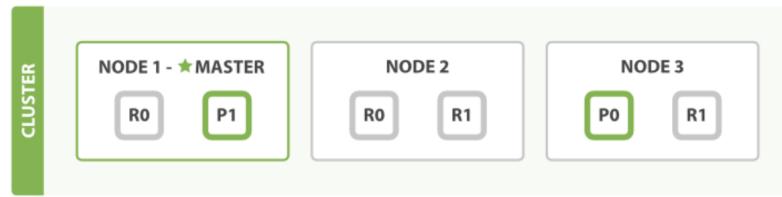
Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间：2021-03-09

- 1. 主分片和副本分片如何交互

1. 主分片和副本分片如何交互

为了说明目的, 我们假设有一个集群由三个节点组成。它包含一个叫 blogs 的索引, 有两个主分片, 每个主分片有两个副本分片。相同分片的副本不会放在同一节点, 所以我们的集群看起来像 Figure 8, “有三个节点和一个索引的集群”。

Figure 8. 有三个节点和一个索引的集群



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

Note

当发送请求的时候, 为了扩展负载, 更好的做法是轮询集群中所有的节点。

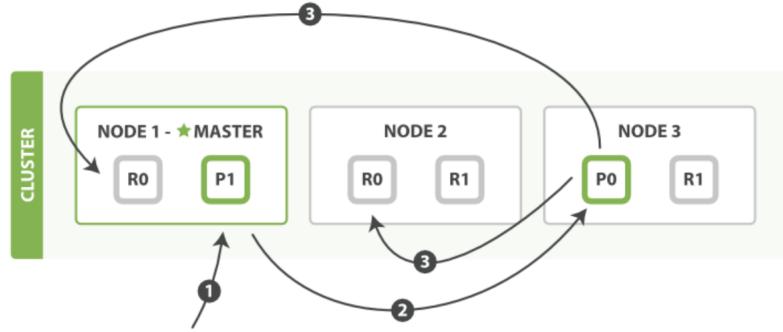
Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间: 2021-03-09

- 1. 新建、索引和删除文档

1. 新建、索引和删除文档

新建、索引和删除 请求都是 写 操作， 必须在主分片上面完成之后才能被复制到相关的副本分片， 如下图所示 Figure 9, “新建、索引和删除单个文档”.

Figure 9. 新建、索引和删除单个文档



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

客户端向 Node 1 发送新建、索引或者删除请求。 1 节点使用文档的 `_id` 确定文档属于分片 0 。请求会被转发到 Node 3，因为分片 0 的主分片 目前被分配在 Node 3 上。 2 Node 3 在主分片上面执行请求。如果成功了，它将请求并行转发到 Node 1 和 Node 2 的副本分片上。 3 一旦所有的副本分片都报告成功, Node 3 将向协调节点报告成功，协调节点向客户端报告成功。在客户端收到成功响应时，文档变更已经在主分片和所有副本分片执行完成，变更是安全的。

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

consistency

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

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

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

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

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

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

timeout

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

NOTE

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

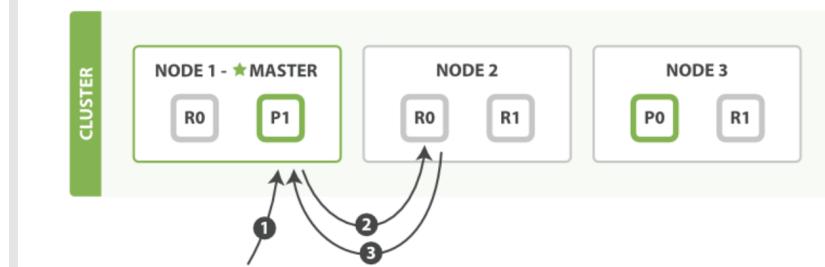
Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间： 2021-03-10

- 1. 取回一个文档

1. 取回一个文档

可以从主分片或者从其它任意副本分片检索文档，如下图所示 Figure 10, “取回单个文档”。

Figure 10. 取回单个文档



以下是从主分片或者副本分片检索文档的步骤顺序：

- 1、客户端向 Node 1 发送获取请求。
- 2、节点使用文档的 _id 来确定文档属于分片 0。分片 0 的副本分片存在于所有的三个节点上。在这种情况下，它将请求转发到 Node 2。
- 3、Node 2 将文档返回给 Node 1，然后将文档返回给客户端。

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

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

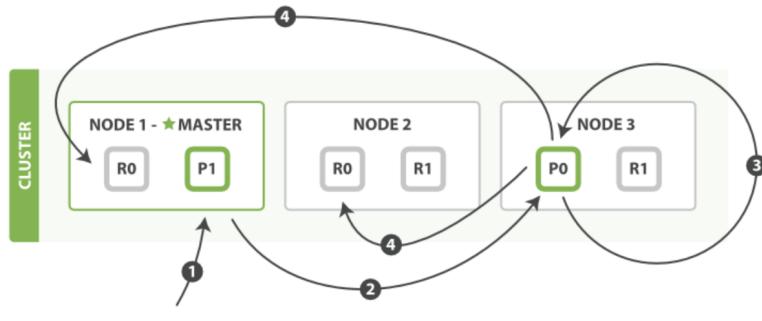
Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间： 2021-03-10

- 1. 局部更新文档

1. 局部更新文档

如 Figure 11, “局部更新文档” 所示, update API 结合了先前说明的读取和写入模式。

Figure 11. 局部更新文档



以下是部分更新一个文档的步骤:

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

update API 还接受在 新建、索引和删除文档 章节中介绍的 `routing`、
`replication`、`consistency` 和 `timeout` 参数。

基于文档的复制

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

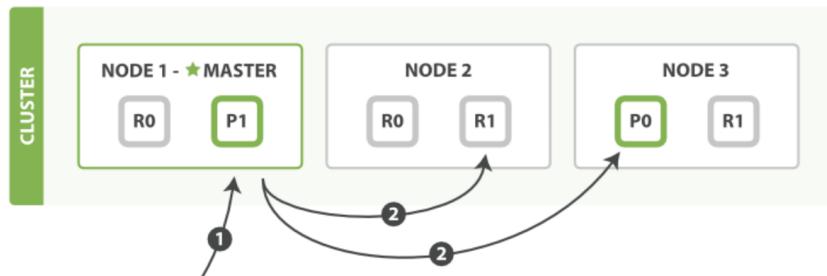
Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间: 2021-03-10

- 1. 多文档模式
 - 1.1. 为什么是有趣的格式

1. 多文档模式

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

协调节点一旦收到来自每个节点的应答，就将每个节点的响应收集整理成单个响应，返回给客户端，如 Figure 12，“使用 `mget` 取回多个文档”所示。



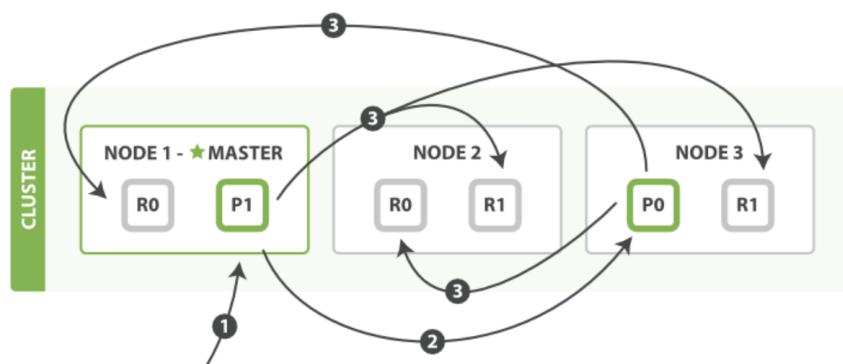
以下是使用单个 `mget` 请求取回多个文档所需的步骤顺序：

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

可以对 `docs` 数组中每个文档设置 `routing` 参数。

`bulk` API，如 Figure 13，“使用 `bulk` 修改多个文档”所示，允许在单个批量请求中执行多个创建、索引、删除和更新请求。

Figure 13. 使用 `bulk` 修改多个文档



`bulk` API 按如下步骤顺序执行：

- 1 客户端向 Node 1 发送 `bulk` 请求。
- 2 Node 1 为每个节点创建一个批量请求，并将这些请求并行转发到每个包含主分片的节点主机。
- 3 主分片一个接一个按顺序执行每个操作。当每个操作成功时，主分片并行

转发新文档（或删除）到副本分片，然后执行下一个操作。一旦所有的副本分片报告所有操作成功，该节点将向协调节点报告成功，协调节点将这些响应收集整理并返回给客户端。

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

1.1. 为什么是有趣的格式

当我们早些时候在[代价较小的批量操作](#)章节了解批量请求时，您可能会问自己，“为什么 bulk API 需要有换行符的有趣格式，而不是发送包装在 JSON 数组中的请求，例如 mget API？”。

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

如果单个请求被包装在 JSON 数组中，那就意味着我们需要执行以下操作：

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

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

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

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

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间：2021-03-10

- 1. 搜索——最基本的工具

1. 搜索——最基本的工具

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

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

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

搜索 (search) 可以做到：

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

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

映射 (Mapping)

描述数据在每个字段内如何存储

分析 (Analysis)

全文是如何处理使之可以被搜索的

领域特定查询语言 (Query DSL)

Elasticsearch 中强大灵活的查询语言

以上提到的每个点都是一个大话题，我们将在深入搜索一章详细阐述它们。

本章节我们将介绍这三点的一些基本概念——仅仅帮助你大致了解搜索是如何工作的。

我们将使用最简单的形式开始介绍 search API。

测试数据

本章节的测试数据可以在这里找到：

<https://gist.github.com/clintongormley/8579281>。

你可以把这些命令复制到终端中执行来实践本章的例子。

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间：2021-03-10

- [1. 空搜索](#)
 - [1.1. hits](#)
 - [1.2. took](#)
 - [1.3. shards](#)
 - [1.4. timeout](#)

1. 空搜索

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

```
GET /_search
```

返回的结果（为了界面简洁编辑过的）像这样：

```
{
  "hits": {
    "total": 14,
    "hits": [
      {
        "_index": "us",
        "_type": "tweet",
        "_id": "7",
        "_score": 1,
        "_source": {
          "date": "2014-09-17",
          "name": "John Smith",
          "tweet": "The Query DSL is really powerful and flexible",
          "user_id": 2
        }
      },
      ...
      9 RESULTS REMOVED ...
    ],
    "max_score": 1
  },
  "took": 4,
  "_shards": {
    "failed": 0,
    "successful": 10,
    "total": 10
  },
  "timed_out": false
}
```

1.1. hits

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

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

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

`max_score` 值是与查询所匹配文档的 `_score` 的最大值。

1.2. took

`took` 值告诉我们执行整个搜索请求耗费了多少毫秒。

1.3. shards

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

1.4. timeout

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

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

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

警示!

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

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

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间：2021-03-10

- 1. 多索引

1. 多索引

搜索API的最基础的形式是没有指定任何查询的空搜索，它简单地返回集群中所有索引下的所有文档：你有没有注意到之前的 `empty search` 的结果，不同类型的数据—`user` 和 `tweet` 来自不同的索引—`us` 和 `gb`？

搜索API的最基础的形式是没有指定任何查询的空搜索，它简单地返回集群中所有索引下的所有文档：如果不对某一特殊的索引或者类型做限制，就会搜索集群中的所有文档。Elasticsearch 转发搜索请求到每一个主分片或者副本分片，汇集查询出的前10个结果，并且返回给我们。

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

<code>/_search</code>	在所有的索引中搜索所有的类型
<code>/gb/_search</code>	在 <code>gb</code> 索引中搜索所有的类型
<code>/gb, us/_search</code>	在 <code>gb</code> 和 <code>us</code> 索引中搜索所有的文档
<code>/g*, u*/_search</code>	在任何以 <code>g</code> 或者 <code>u</code> 开头的索引中搜索所有的类型
<code>/gb/user/_search</code>	在 <code>gb</code> 索引中搜索 <code>user</code> 类型
<code>/gb, us/user, tweet/_search</code>	在 <code>gb</code> 和 <code>us</code> 索引中搜索 <code>user</code> 和 <code>tweet</code> 类型
<code>/all/user, tweet/_search</code>	在所有的索引中搜索 <code>user</code> 和 <code>tweet</code> 类型

当在单一的索引下进行搜索的时候，Elasticsearch 转发请求到索引的每个分片中，可以是主分片也可以是副本分片，然后从每个分片中收集结果。多索引搜索恰好也是用相同的方式工作的一只是会涉及到更多的分片。

警示 !

搜索一个索引有五个主分片和搜索五个索引各有一个分片准确来说是等价的。Elasticsearch6版本之后

接下来，你将明白这种简单的方式如何灵活的根据需求的变化让扩容变得简单。

Copyright © Songbai Yang all right reserved, powered by Gitbook该文件修订时间：2021-03-10

- 1. 分页

1. 分页

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

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

size

显示应该返回的结果数量， 默认是 10

from

显示应该跳过的初始结果数量， 默认是 0

如果每页展示 5 条结果，可以用下面方式请求得到 1 到 3 页的结果：

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

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

在分布式系统中深度分页

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

现在假设我们请求第 1000 页—结果从 10001 到 10010 。所有都以相同的方式工作除了每个分片不得不产生前 1000 个结果以外。然后协调节点对全部 5000 个结果排序最后丢弃掉这些结果中的 5000 个结果。

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

Tip

在 重新索引你的数据 中解释了如何能够有效获取大量的文档。

Copyright © Songbai Yang all right reserved, powered by Gitbook该文件修订时间： 2021-03-10

- [1. 轻量 搜索](#)
- [2. `_all`](#)
 - [2.1. 更复杂的查询](#)

1. 轻量 搜索

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

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

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

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

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

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

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

2. `_all`

这个简单搜索返回包含 `mary` 的所有文档：

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

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

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

Elasticsearch 是如何在三个不同的字段中查找到结果的呢？

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

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

这就好似增加了一个名叫 `_all` 的额外字段：

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

Tip

在刚开始开发一个应用时，`_all` 字段是一个很实用的特性。之后，你会发现如果搜索时用指定字段来代替 `_all` 字段，将会更好控制搜索结果。当 `_all` 字段不再有用的时候，可以将它置为失效，正如在元数据: `_all` 字段 中所解释的。

2.1. 更复杂的查询

下面的查询针对 `tweets` 类型，并使用以下的条件：

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

查询字符串在做了适当的编码后，可读性很差：

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

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

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

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

Tip

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

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

Copyright © Songbai Yang all right reserved, powered by Gitbook该文件修订时间：2021-03-10

- 1. 映射和分析

1. 映射和分析

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

```
GET /_search?q=2014          # 12 results
GET /_search?q=2014-09-15    # 12 results !
GET /_search?q=date:2014-09-15 # 1 result
GET /_search?q=date:2014      # 0 results !
```

为什么在 `_all` 字段查询日期返回所有推文，而在 `date` 字段只查询年份却没有返回结果？为什么我们在 `_all` 字段和 `date` 字段的查询结果有差别？

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

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

这将得到如下结果：

```
{
  "gb": {
    "mappings": {
      "tweet": {
        "properties": {
          "date": {
            "type": "date",
            "format": "strict_date_optional_time|epoch_millis"
          },
          "name": {
            "type": "string"
          },
          "tweet": {
            "type": "string"
          },
          "user_id": {
            "type": "long"
          }
        }
      }
    }
  }
}
```

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

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

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

Copyright © Songbai Yang all right reserved, powered by Gitbook该文件修订时间： 2021-03-10

- 1. 精确值 VS 全文

1. 精确值 VS 全文

Elasticsearch 中的数据可以概括的分为两类：精确值和全文。

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

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

NOTE

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

```
May is fun but June bores me.
```

它指的是月份还是人？

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

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

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

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

- 搜索 UK，会返回包含 United Kingdom 的文档。
- 搜索 jump，会匹配 jumped，jumps，jumping，甚至是 leap。
- 搜索 johnny walker 会匹配 Johnnie Walker，johnnie depp 应该匹配 Johnny Depp。

fox news hunting 应该返回福克斯新闻（ Fox News ）中关于狩猎的故事，同时， fox hunting news 应该返回关于猎狐的故事。为了促进这类在全文域中的查询， Elasticsearch 首先 分析 文档，之后根据结果创建 倒排索引。在接下来的两节，我们会讨论倒排索引和分析过程。

Copyright © Songbai Yang all right reserved, powered by Gitbook 该文件修订时间：2021-03-13

- 1. 倒排索引

1. 倒排索引

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

例如，假设我们有两个文档，每个文档的 `content` 域包含如下内容：

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

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

Term	Doc_1	Doc_2
Quick		X
The	X	
brown	X	X
dog	X	
dogs		X
fox	X	
foxes		X
in		X
jumped	X	
lazy	X	X
leap		X
over	X	X
quick	X	
summer		X
the	X	

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

Term	Doc_1	Doc_2
brown	X	X
quick	X	
Total	2	1

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

但是，我们目前的倒排索引有一些问题：

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

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

我们的用户可以合理的期望两个文档与查询匹配。我们可以做的更好。

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

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

现在索引看上去像这样：

Term	Doc_1	Doc_2
brown	X	X
dog	X	X
fox	X	X
in		X
jump	X	X
lazy	X	X
over	X	X
quick	X	X
summer		X
the	X	X

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

NOTE

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

分词和标准化的过程称为 分析，我们在下个章节讨论。

Copyright © Songbai Yang all right reserved, powered by Gitbook该文件修订时间： 2021-03-13

- [1. 分析与分析器](#)
 - [1.1. 内置分析器](#)
 - [1.2. 什么时候使用分词器](#)
 - [1.3. 测试分词器](#)
 - [1.4. 指定分词器](#)

1. 分析与分析器

分析 包含下面的过程：

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

分析器执行上面的工作。分析器 实际上是将三个功能封装到了一个包里：

字符过滤器

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

其次，字符串被 分词器 分为单个的词条。一个简单的分词器遇到空格和标点的时候，可能会将文本拆分成词条。 Token 过滤器 最后，词条按顺序通过每个 token 过滤器。这个过程可能会改变词条（例如，小写化 Quick），删除词条（例如，像 a， and， the 等无用词），或者增加词条（例如，像 jump 和 leap 这种同义词）。

Elasticsearch提供了开箱即用的字符过滤器、分词器和token 过滤器。这些可以组合起来形成自定义的分析器以用于不同的目的。我们会在 [自定义分析器](#) 章节详细讨论。

1.1. 内置分析器

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

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

标准分析器

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

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

简单分析器

简单分析器在任何不是字母的地方分隔文本，将词条小写。它会产生

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

空格分析器

空格分析器在空格的地方划分文本。它会产生

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

语言分析器

特定语言分析器可用于很多语言。它们可以考虑指定语言的特点。例如，英语分析器附带了一组英语无用词（常用单词，例如 `and` 或者 `the`，它们对相关性没有多少影响），它们会被删除。由于理解英语语法的规则，这个分词器可以提取英语单词的词干。

英语分词器会产生下面的词条：

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

注意看 `transparent`、`calling` 和 `set_trans` 已经变为词根格式。

1.2. 什么时候使用分词器

当我们索引一个文档，它的全文域被分析成词条以用来创建倒排索引。但是，当我们在全文域搜索的时候，我们需要将查询字符串通过相同的分析过程，以保证我们搜索的词条格式与索引中的词条格式一致。

全文查询，理解每个域是如何定义的，因此它们可以做正确的事：

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

现在你可以理解在开始章节的查询为什么返回那样的结果：

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

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

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

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

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

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

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

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

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

1.3. 测试分词器

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

```
GET /_analyze
{
  "analyzer": "standard",
  "text": "Text to analyze"
}
```

结果中每个元素代表一个单独的词条：

```
{
  "tokens": [
    {
      "token": "text",
      "start_offset": 0,
      "end_offset": 4,
      "type": "<ALPHANUM>",
      "position": 1
    },
    {
      "token": "to",
      "start_offset": 5,
      "end_offset": 7,
      "type": "<ALPHANUM>",
      "position": 2
    },
    {
      "token": "analyze",
      "start_offset": 8,
      "end_offset": 15,
      "type": "<ALPHANUM>",
      "position": 3
    }
  ]
}
```

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

NOTE

每个分析器的 `type` 值都不一样，可以忽略它们。它们在 Elasticsearch 中的唯一作用在于 [keep_types token](#) 过滤器。

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

1.4. 指定分词器

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

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

要做到这一点，我们必须手动指定这些域的映射。

Copyright © Songbai Yang all right reserved, powered by Gitbook该文件修订时间： 2021-03-14

- [1. 映射](#)
 - [1.1. 核心简单域类型](#)
 - [1.2. 查看映射](#)
 - [1.3. 自定义域映射](#)
 - [1.4. index](#)
 - [1.5. analyzer](#)
 - [1.6. 更新映射](#)
 - [1.7. 测试映射](#)

1. 映射

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

如 [数据输入和输出](#) 中解释的，索引中每个文档都有 [类型](#)。每种类型都有它自己的 [映射](#)，或者 [模式定义](#)。映射定义了类型中的域，每个域的数据类型，以及 Elasticsearch 如何处理这些域。映射也用于配置与类型有关的元数据。

我们会在 [类型和映射](#) 详细讨论映射。本节，我们只讨论足够让你入门的内容。

1.1. 核心简单域类型

Elasticsearch 支持如下简单域类型：

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

当你索引一个包含新域的文档—之前未曾出现-- Elasticsearch 会使用 [动态映射](#)，通过 JSON 中基本数据类型，尝试猜测域类型，使用如下规则：

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

NOTE

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

1.2. 查看映射

通过 `_mapping`，我们可以查看 Elasticsearch 在一个或多个索引中的一个或多个类型的映射。在开始章节，我们已经取得索引 `gb` 中类型 `tweet` 的映射：

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

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

```
{
  "gb": {
    "mappings": {
      "tweet": {
        "properties": {
          "date": {
            "type": "date",
            "format": "strict_date_optional_time|epoch_millis"
          },
          "name": {
            "type": "string"
          },
          "tweet": {
            "type": "string"
          },
          "user_id": {
            "type": "long"
          }
        }
      }
    }
  }
}
```

Tip

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

检查一下！而不是假设你的映射是正确的。

1.3. 自定义域映射

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

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

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

```
type :
```

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

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

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

1.4. index

`index` 属性控制怎样索引字符串。它可以是下面三个值：

analyzed

首先分析字符串，然后索引它。换句话说，以全文索引这个域。

not_analyzed

索引这个域，所以它能够被搜索，但索引的是精确值。不会对它进行分析。

no

不索引这个域。这个域不会被搜索到。

`string` 域 `index` 属性默认是 `analyzed`。如果我们想映射这个字段为一个精确值，我们需要设置它为 `not_analyzed`：

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

其他简单类型（例如 `long`，`double`，`date` 等）也接受 `index` 参数，但有意义的值只有 `no` 和 `not_analyzed`，因为它们永远不会被分析。

1.5. analyzer

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

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

在自定义分析器，我们会展示怎样定义和使用自定义分析器。

1.6. 更新映射

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

>

尽管你可以增加一个存在的映射，你不能修改存在的域映射。如果一个域的映射已经存在，那么该域的数据可能已经被索引。如果你意图修改这个域的映射，索引的数据可能会出错，不能被正常的搜索。

我们可以更新一个映射来添加一个新域，但不能将一个存在的域从 `analyzed` 改为 `not_analyzed`。

为了描述指定映射的两种方式，我们先删除 `gb` 索引：

```
DELETE /gb
```

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

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

(a) 通过消息体中指定的 `mappings` 创建了索引。

稍后，我们决定在 `tweet` 映射增加一个新的名为 `tag` 的 `not_analyzed` 的文本域，使用 `_mapping`：

```
PUT /gb/_mapping/tweet
{
  "properties" : {
    "tag" : {
      "type" : "string",
      "index": "not_analyzed"
    }
  }
}
```

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

1.7. 测试映射

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

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

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

(a) 消息体里面传输我们想要分析的文本。

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

Copyright © Songbai Yang all right reserved, powered by Gitbook该文件修订时间： 2021-03-14

- **1. 复杂核心域类型**
 - **1.1. 多值域**
 - **1.2. 空域**
 - **1.3. 多层级对象**
 - **1.4. 内部对象的映射**
 - **1.5. 内部对象是如何索引的**
 - **1.6. 内部对象是如何被索引的**

1. 复杂核心域类型

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

1.1. 多值域

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

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

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

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

NOTE

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

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

1.2. 空域

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

下面三种域被认为是空的，它们将不会被索引：

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

1.3. 多层级对象

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

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

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

1.4. 内部对象的映射

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

```
{
  "gb": {
    "gb": {
      "tweet": { (a)
        "properties": {
          "tweet": { "type": "string" },
          "user": { (b)
            "type": "object",
            "properties": {
              "id": { "type": "string" },
              "gender": { "type": "string" },
              "age": { "type": "long" },
              "name": { (b)
                "type": "object",
                "properties": {
                  "full": { "type": "string" },
                  "first": { "type": "string" },
                  "last": { "type": "string" }
                }
              }
            }
          }
        }
      }
    }
  }
}
```

(a) 根对象

(b) 内部对象

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

1.5. 内部对象是如何索引的

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

```
{
  "tweet": ["elasticsearch", "flexible", "very"],
  "user.id": "@johnsmith",
  "user.gender": "male",
  "user.age": [26],
  "user.name.full": "john smith",
  "user.name.first": "john",
  "user.name.last": "smith"
}
```

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

Note

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

1.6. 内部对象是如何被索引的

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

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

这个文档会像我们之前描述的那样被扁平化处理，结果如下所示：

```
{
  "followers.age": [19, 26, 35],
  "followers.name": [alex, jones, lisa, smith, mary, white]
}
```

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

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

相关内部对象被称为 `nested` 对象，可以回答上面的查询，我们稍后会在嵌套对象中介绍它。

安装运行elasticsearch

Copyright © Songbai Yang all right reserved, powered by Gitbook该文件修订时间： 2021-03-14

- 1. 请求体查询

1. 请求体查询

简易查询—query-string search—对于用命令行进行即席查询（ad-hoc）是非常有用的。然而，为了充分利用查询的强大功能，你应该使用请求体 search API，之所以称之为请求体查询(Full-Body Search)，因为大部分参数是通过Http请求体而非查询字符串来传递的。

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

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间：2021-03-14

- 1. 空查询

1. 空查询

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

```
GET /_search
{ } ( a)
```

(a) 这是一个空的请求体。

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

```
GET /index_2014*/type/_search
{ }
```

同时你可以使用 `from` 和 `size` 参数来分页：

```
GET /_search
{
  "from": 30,
  "size": 10
}
```

一个带请求体的 GET 请求？

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

而事实是这个RFC文档 RFC 7231—一个专门负责处理 HTTP 语义和内容的文档—并没有规定一个带有请求体的 GET 请求应该如何处理！结果是，一些 HTTP 服务器允许这样子，而有一些—特别是那些用于缓存和代理的服务器—则不允许。

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

```
POST /_search
{
  "from": 30,
  "size": 10
}
```

类似的规则可以应用于任何需要带请求体的 GET API。

我们将在聚合 `聚合` 章节深入介绍聚合 (`aggregations`)，而现在，我们将聚焦在查询。

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

Copyright © Songbai Yang all right reserved, powered by Gitbook该文件修订时间：2021-03-14

- 1. 查询表达式

1. 查询表达式

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

要使用这种查询表达式，只需将查询语句传递给 `query` 参数：

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

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

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

Copyright © Songbai Yang all right reserved, powered by Gitbook该文件修订时间： 2021-03-14