Elasticsearch 入门指南

1 Elasticsearch 入门

欢迎来到 Elasticsearch 的世界，它目前是全球最受欢迎的全文搜索引擎。不管你对 Elasticsearch 和全文检索有没有经验都不要紧。我们希望你可以通过这本书，来走进 Elasticsearch 的大门。这本书是为初学者准备的，当然对于中高级的人员也有参考作用，我们首先介绍一些和 Elasticsearch 相关的基础内容。接着介绍一下 Elasticsearch 的安装和配置。此外本章还会介绍如何简单的使用 Elasticsearch，包括 http json 接口和 JAVA 接口。在学习这些接口的过程中，不用陷入太多的细节，后面的章节会逐步展开并细化接口内容。读完本章，你将学到以下内容：

- Elasticsearch 介绍；
- 全文检索；
- Elasticsearch 的基础；
- 安装和配置 Elasticsearch；
- HTTP REST API 接口；
- JAVA 开发接口。

1.1 Elasticsearch 是什么

1.1.1 Elasticsearch 是什么

Elasticsearch 是一个基于 Lucene 构建的开源，分布式，RESTful 全文搜索引擎。Elasticsearch 还是一个分布式文档数据库，其中每个 field 均是被索引的数据且可被搜索，它能够扩展至数以百计的服务器存储及处理 PB 级的数据。它可以在很短的时间内存储，搜索和分析大量的数据。它通常作为具有复杂搜索场景情况下的核心发动机。

Elasticsearch 就是为高可用和可扩展而生的。扩展可以通过购置性能更强的服务器(垂直扩展或者向上扩展，Vertical Scale/Scaling Up)，亦或是通过购置更多的服务器(水平扩展或者向外扩展，Horizontal Scale/Scaling Out)来完成。

尽管 ES 能够利用更强劲的硬件，垂直扩展毕竟还是有它的极限。真正的可扩展性来自于水平扩展，通过向集群中添加更多的节点来分布负载，增加可靠性。

在大多数数据库中，水平扩展通常都需要你对应用进行一次大的重构来利用更多的节点。相反，ES 天生就是分布式的：它知道如何管理多个节点来完成扩展和实现高可用性。这也意味着你的应用不需要在乎这一点。

我们举几个例子来说明 Elasticsearch 能做什么？

当你经营一家网上商店，你可以让你的客户搜索你卖的商品。在这种情况下，你可以使用 Elasticsearch 来存储你的整个产品目录和库存信息，为客户提供精准搜索，可以为客户推荐相关商品。

当你想收集日志或者交易数据的时候，需要分析和挖掘这些数据，寻找趋势，统计，总结，或异常。在这种情况下，你可以使用 LogStash 或者其他工具来进行收集数据，当这些数据存储到 Elasticsearch 中 。你可以搜索和汇总这些数据，找到任何你感兴趣的信息。

当你运行一个价格提醒的平台，可以给客户提供一些规则，如我有兴趣购买一个特定的电子设备，当商品的价格在未来一个月内价格低于多少钱的时候通知我。在这种情况下，你可以把供应商的价格，把他们定期存储到 Elasticsearch 中，使用定时器过滤来匹配客户的需求，当查询到价格低于客户设定的值后给客户发送一条通知。

当有大量数据（千万条以上的记录）时，你有商业智能分析的需求，希望快速调查，分析和可视化。在这种情况下，你可以使用 Elasticsearch 来存储你的数据，然后用 Kibana 建立自定义的仪表板或者任何你熟悉的语言开发展示界面，你可以使用 Elasticsearch 的聚合功能来执行复杂的商业智能与数据查询。

对于码农来说，比较有名的案例是 GitHub，GitHub 的搜索是基于 Elasticsearch 构建的，在 github.com/search 页面，你可以检索项目、用户、issue、pull request，还有代码。共有 40-50 个索引库，分别用于索引网站需要跟踪的各种数据。虽然只索引项目的主分支(master)，但这个数据量依然巨大，20 亿个索引文档，30TB 的索引文件。

1.1.2 Elasticsearch 历史

网上流传着历史是：多年前，一个叫做 Shay Banon 的刚结婚不久的失业开发者，由于妻子要去伦敦学习厨师，他便跟着也去了。在他找工作的过程中，为了给妻子构建一个食谱的搜索引擎，他开始构建一个早期版本的 Lucene。

直接基于 Lucene 工作会比较困难，所以 Shay 开始抽象 Lucene 代码以便 Java 程序员可以在应用中添加搜索功能。他发布了他的第一个开源项目，叫做“Compass”。

后来 Shay 找到一份工作，这份工作处在高性能和内存数据网格的分布式环境中，因此高性能的、实时的、分布式的搜索引擎也是理所当然需要的。然后他决定重写 Compass 库使其成为一个独立的服务叫做 Elasticsearch。

第一个公开版本出现在 2010 年 2 月，在那之后 Elasticsearch 已经成为 Github 上最受欢迎的项目之一，代码贡献者超过 300 人。直到 2016 年 3 月 30 日，Elasticsearch 已经发布了 2.3.0 版本。目前已经成为全球最受欢迎的全文搜索引擎。

那 Elasticsearch 为什么会有如此的魅力呢？我们首先看一下 Elasticsearch 的优点：

横向可扩展性：只需要增加一台服务器，做一点儿配置，启动一下 Elasticsearch 进程就可以并入集群；

分片机制提供更好的分布性：同一个索引分成多个分片（sharding），这点类似于 HDFS 的块机制；分而治之的方式来提升处理效率；

高可用：提供复制（replica）机制，一个分片可以设置多个复制，使得某台服务器在宕机的情况下，集群仍旧可以照常运行，并会把服务器宕机丢失的数据信息复制恢复到其它可用节点上；

使用简单：下载文件，一条命令就可以启动，然后很快可以搭建一个站内搜索引擎。

1.1.3 相关产品

Beats：它是一个代理，将不同类型的数据发送到 Elasticsearch 中。它可以直接将数据发送到 Elasticsearch。Beats 有三部分的内容组成：Filebeat、Topbeat、Packetbeat。Filebeat 用来收集日志。Topbeat 用来收集系统基础设置数据，如 cpu、内存、每个进程的统计信息。Packetbeat 是一个网络包分析工具，统计收集网络信息。这三个是官方提供的。

Shield：它为 Elasticsearch 带来企业级的安全性，加密通信，认证保护整个 Elasticsearch 数据，它是基于角色的访问控制与审计。当今企业对安全需求越来越重视，Shield 可以提供安全的 Elasticsearch 访问，从而保护核心的数据。注意：Shield 是收费的产品。

Watcher：它是 Elasticsearch 的警报和通知工具。它可以主动监测 Elasticsearch 的状态，并在有异常的时候进行提醒，它可以根据你的数据变化情况来采取不同的处理方式。注意：Watcher 也是收费的产品。

Marvel：它是 Elasticsearch 的管理和监控工具。它监测 Elasticsearch 集群的活动，快速诊断问题，并能发现微调方式和优化性能。注意：Marvel 也是收费的产品。

1.2 全文搜索

1.2.1 Lucene 介绍

Lucene 是 Apache 软件基金会一个开放源代码的全文检索引擎工具包，是一个全文检索引擎的架构，提供了完整的查询引擎和索引引擎，部分文本分析引擎。Lucene 的目的是为软件开发人员提供一个简单易用的工具包，以方便在目标系统中实现全文检索的功能，或者是以此为基础建立起完整的全文检索引擎。

Lucene 最初是由 Doug Cutting 所撰写的，是一位资深全文索引/检索专家，曾经是 V-Twin 搜索引擎的主要开发者，后来在 Excite 担任高级系统架构设计师，目前从事于一些 INTERNET 底层架构的研究。他贡献出 Lucene 的目标是为各种中小型应用程式加入全文检索功能。

1.2.2 Lucene 倒排索引

倒排索引源于实际应用中需要根据属性的值来查找记录。这种索引表中的每一项都包括一个属性值和具有该属性值的各记录的地址。由于不是由记录来确定属性值，而是由属性值来确定记录的位置，因而称为倒排索引(inverted index)。带有倒排索引的文件我们称为倒排索引文件，简称倒排文件(inverted file)。

倒排文件（倒排索引），索引对象是文档或者文档集合中的单词等，用来存储这些单词在一个文档或者一组文档中的存储位置，是对文档或者文档集合的一种最常用的索引机制。

搜索引擎的关键步骤就是建立倒排索引，倒排索引一般表示为一个关键词，然后是它的频度（出现的次数），位置（出现在哪一篇文章或网页中，及有关的日期，作者等信息），它相当于为互联网上几千亿页网页做了一个索引，好比一本书的目录、标签一般。读者想看哪一个主题相关的章节，直接根据目录即可找到相关的页面。不必再从书的第一页到最后一页，一页一页的查找。

Lucene 倒排索引原理

Lucene 是一个开放源代码的高性能的 java 全文检索引擎工具包，不是一个完整的全文检索引擎，而是一个全文检索引擎的架构，提供了完整的查询引擎和索引引擎，部分文本分析引擎。目的是为软件开发人员提供一个简单易用的工具包，以方便在目标系统中实现全文检索的功能，或者以此为基础建立起完整的全文检索引擎。

Lucene 使用的是倒排文件索引结构。该结构及相应的生成算法如下：

设有两篇文章 1 和 2：

文章 1 的内容为：Tom lives in Guangzhou,I live in Guangzhou too.

文章 2 的内容为：He once lived in Shanghai.

- 取得关键词

由于 Lucene 是基于关键词索引和查询的，首先我们要取得这两篇文章的关键词，通常我们需要如下处理措施：

a.我们现在有的是文章内容，即一个字符串，我们先要找出字符串中的所有单词，即分词。英文单词由于用空格分隔，比较好处理。中文单词间是连在一起的需要特殊的分词处理。

b.文章中的“in”“once” “too”等词没有什么实际意义，中文中的“的”“是”等字通常也无具体含义，这些不代表概念的词可以过滤掉 。

c.用户通常希望查“He”时能把含“he”，“HE”的文章也找出来，所以所有单词需要统一大小写。

d.用户通常希望查“live”时能把含“lives”，“lived”的文章也找出来，所以需要把“lives”，“lived”还原成“live”。

e.文章中的标点符号通常不表示某种概念，也可以过滤掉 。

在 Lucene 中以上措施由 Analyzer 类完成。 经过上面处理后，

文章 1 的所有关键词为：[tom] [live] [guangzhou] [i] [live] [guangzhou]

文章 2 的所有关键词为：[he] [live] [shanghai]

- 建立倒排索引

有了关键词后，我们就可以建立倒排索引了。上面的对应关系是：“文章号”对“文章中所有关键词”。倒排索引把这个关系倒过来，变成: “关键词”对“拥有该关键词的所有文章号”。

文章 1，2 经过倒排后变成

关键词	文章号
guangzhou	1
he	2
i	1
live	1，2
shanghai	2
tom	1

表格 1.1 倒排索引关键词文章号对应关系示例

通常仅知道关键词在哪些文章中出现还不够，我们还需要知道关键词在文章中出现的次数和位置，通常有两种位置：

a.字符位置，即记录该词是文章中第几个字符（优点是关键词亮显时定位快）；

b.关键词位置，即记录该词是文章中第几个关键词（优点是节约索引空间、词组（phase）查询快），Lucene 中记录的就是这种位置。

加上“出现频率”和“出现位置”信息后，我们的索引结构变为：

关键词	文章号[出现频率]	出现位置
guangzhou	1[2]	3,6
he	2[1]	1
i	1[1]	4
live	1[2] 2[1]	2,5 2
shanghai	2[1]	3
tom	1[1]	1

表格 1.2 倒排索引关键词频率位置示例

以 live 这行为例我们说明一下该结构：live 在文章 1 中出现了 2 次，文章 2 中出现了一次，它的出现位置为“2,5,2”这表示什么呢？我们需要结合文章号和出现频率来分析，文章 1 中出现了 2 次，那么“2,5”就表示 live 在文章 1 中出现的两个位置，文章 2 中出现了一次，剩下的“2”就表示 live 是文章 2 中第 2 个关键字。

以上就是 Lucene 索引结构中最核心的部分。我们注意到关键字是按字符顺序排列的（Lucene 没有使用 B 树结构），因此 Lucene 可以用二元搜索算法快速定位关键词。

- 实现

实现时，Lucene 将上面三列分别作为词典文件（Term Dictionary）、频率文件(frequencies)、位置文件 (positions)保存。其中词典文件不仅保存了每个关键词，还保留了指向频率文件和位置文件的指针，通过指针可以找到该关键字的频率信息和位置信息。

Lucene 中使用了 field 的概念，用于表达信息所在位置（如标题中，文章中，URL 中），在建索引中，该 field 信息也记录在词典文件中，每个关键词都有一个 field 信息(因为每个关键字一定属于一个或多个 field)。

- 压缩算法

为了减小索引文件的大小，Lucene 对索引还使用了压缩技术。

首先，对词典文件中的关键词进行了压缩，关键词压缩为<前缀长度，后缀>，例如：当前词为“阿拉伯语”，上一个词为“阿拉伯”，那么“阿拉伯语”压缩为<3，语>。

其次大量用到的是对数字的压缩，数字只保存与上一个值的差值（这样可以减小数字的长度，进而减少保存该数字需要的字节数）。例如当前文章号是 16389（不压缩要用 3 个字节保存），上一文章号是 16382，压缩后保存 7（只用一个字节）。

- 应用原因

下面我们可以通过对该索引的查询来解释一下为什么要建立索引。

假设要查询单词 “live”，Lucene 先对词典二元查找、找到该词，通过指向频率文件的指针读出所有文章号，然后返回结果。词典通常非常小，因而，整个过程的时间是毫秒级的。

而用普通的顺序匹配算法，不建索引，而是对所有文章的内容进行字符串匹配，这个过程将会相当缓慢，当文章数目很大时，时间往往是无法忍受的。

1.3 基础

1.3.1 Elasticsearch 术语概念

- 索引词(term)

在 Elasticsearch 中索引词(term)是一个能够被索引的精确值。foo，Foo Foo 几个单词是不相同的索引词。索引词(term)是可以通过 term 查询进行准确的搜索。

- 文本(text)

文本是一段普通的非结构化文字。通常，文本会被分析成一个个的索引词，存储在 Elasticsearch 的索引库中。为了让文本能够进行搜索，文本字段需要事先进行分析；当对文本中的关键词进行查询的时候，搜索引擎应该根据搜索条件搜索出原文本。

- 分析(analysis)

分析是将文本转换为索引词的过程，分析的结果依赖于分词器。比如： FOO BAR，Foo-Bar，foo bar 这几个单词有可能会被分析成相同的索引词 foo 和 bar，这些索引词存储在 Elasticsearch 的索引库中。当用 FoO:bAR 进行全文搜索的时候，搜索引擎根据匹配计算也能在索引库中搜索出之前的内容。这就是 Elasticsearch 的搜索分析。

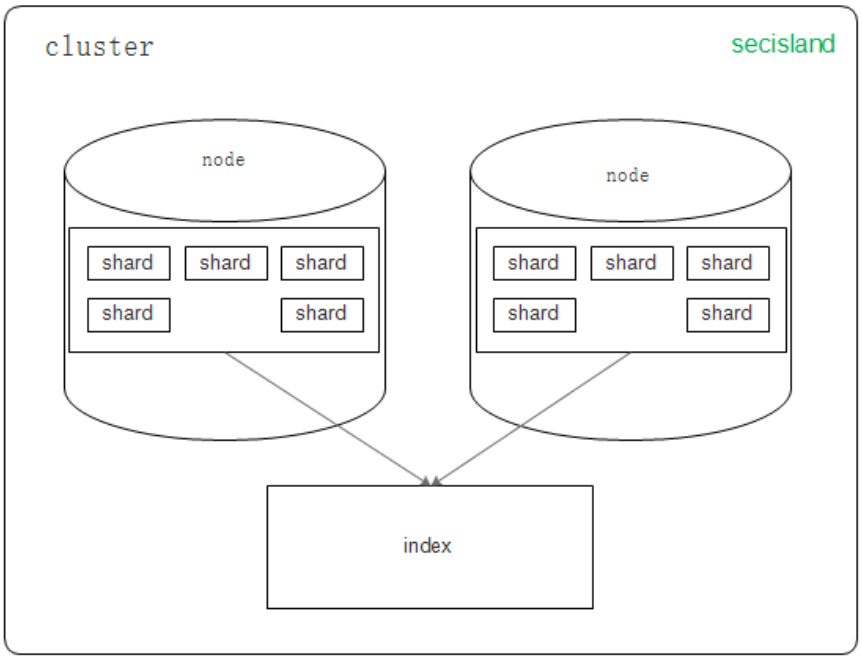

图 1.1　　Elasticsearch 集群术语结构

- 集群(cluster)

集群由一个或多个节点组成，对外提供服务，对外提供索引和搜索功能。在所有节点，一个集群有一个唯一的名称默认为“Elasticsearch”。此名称是很重要的，因为每个节点只能是集群的一部分，当该节点被设置为相同的集群名称时，就会自动加入集群。当需要有多个集群的时候，要确保每个集群的名称不能重复，否则，节点可能会加入错误的集群。请注意，一个节点只能加入一个集群。此外，你还可以拥有多个独立的集群，每个集群都有其不同的集群名称。例如，在开发过程中，你可以建立开发集群库和测试集群库，分别为开发、测试服务。

- 节点(node)

一个节点是一个逻辑上独立的服务，它是集群的一部分，可以存储数据，并参与集群的索引和搜索功能。就像集群一样，节点也有唯一的名字，默认是一个随机的和机器相关的名称，在启动的时候分配。如果你不想要默认名称，你可以定义任何你想要的节点名。这个名字在管理中很重要，在网络中 Elasticsearch 集群通过节点名称进行管理和通信。一个节点可以被配置加入一个特定的集群。默认情况下，每个节点会加入名为 Elasticsearch 的集群中，这意味着如果你在网络上启动多个节点，如果网络畅通，他们能彼此发现并自动加入一个名为 Elasticsearch 的集群中。在一个单一的集群中，你可以拥有多个你想要的节点。当网络没有集群运行的时候，只要启动任何一个节点，这个节点会默认生成一个新的集群，这个集群会有一个节点。

- 路由(routing)

当存储一个文档的时候，它会存储在一个唯一的主分片中，具体哪个分片是通过散列值进行选择。默认情况下，这个值是由文档的 ID 生成。如果文档有一个指定的父文档，则从父文档 ID 中生成，该值可以在存储文档的时候进行修改。

- 分片(shard)

分片是一个单一的 Lucene 实例。这个是 Elasticsearch 管理的比较底层的功能。索引是指向主分片和副本分片的逻辑空间。对于使用，只需要指定分片的数量，其他不需要做过多的事情。在开发使用的过程中，我们对应的对象都是索引，Elasticsearch 会自动管理集群中所有的分片，当发生故障的时候，一个 Elasticsearch 会把分片移动到不同的节点或者添加新的节点。

一个索引可以存储很大的数据，这些空间可以超过一个节点的物理存储的限制。例如，十亿个文档占用磁盘空间为 1TB。仅从单个节点搜索可能会很慢，还有一台物理机器也不一定能存储这么多的数据。为了解决这一问题，Elasticsearch 将索引分解成多个分片。当你创建一个索引，你可以简单地定义你想要的分片数量。每个分片本身是一个全功能的、独立的单元，可以托管在集群中的任何节点。

- 主分片(primary shard)

每个文档都存储在一个分片中，当你存储一个文档的时候，系统会首先存储在主分片中，然后会复制到不同的副本中。默认情况下，一个索引有 5 个主分片。你可以事先制定分片的数量，当分片一旦建立，分片的数量则不能修改。

- 副本分片(replica shard)

每一个分片有零个或多个副本。副本主要是主分片的复制，其中有两个目的：

1、增加高可用性：当主分片失败的时候，可以从副本分片中选择一个作为主分片。

2、提高性能：当查询的时候可以到主分片或者副本分片中进行查询。默认情况下，一个主分片配有一个副本，但副本的数量可以在后面动态地配置增加。副本必须部署在不同的节点上，不能部署在和主分片相同的节点上。

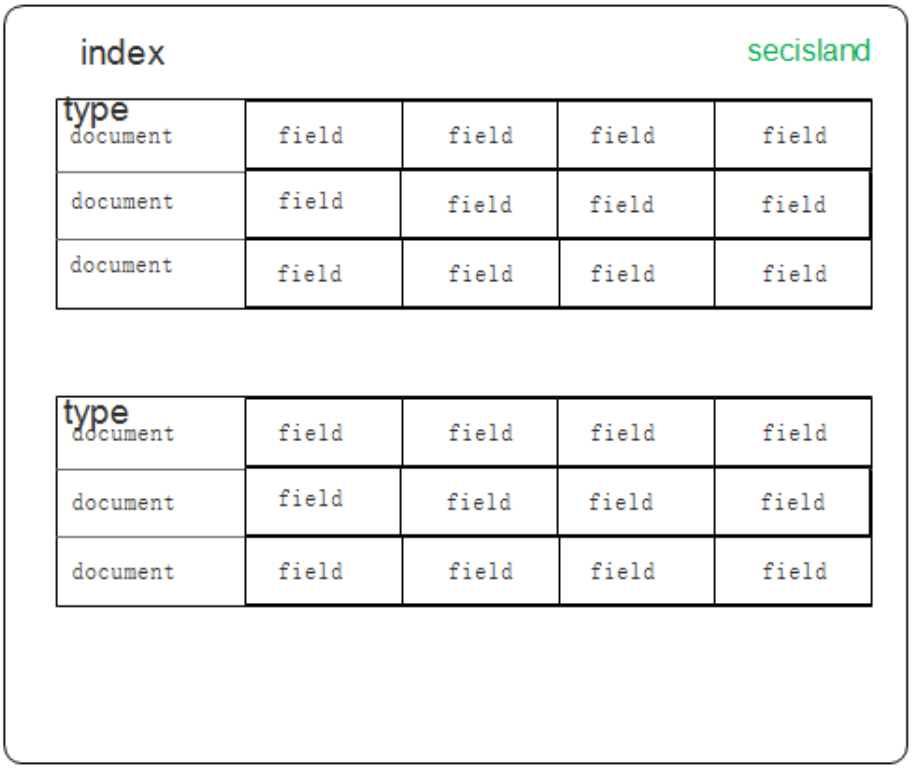

图 1.2 Elasticsearch 索引术语结构

分片主要有两个很重要的原因是：

1、它允许你水平分割扩展你的数据。

2、它允许你分配和并行操作（可能在多个节点上）从而提高性能和吞吐量。

这些很强大的功能对用户来说是透明的，你不需要做什么操作，系统会自动处理。

- 复制(replicas)

复制是一个非常有用的功能，不然会有单点问题。当网络中的某个节点出现问题的时候，复制可以对故障进行转移，保证系统的高可用。因此，Elasticsearch 允许你创建一个或多个拷贝，你的索引分片就形成了所谓的副本或副本分片。

复制是重要的，主要的原因有：

1、它提供了高可用性，当节点失败的时候不受影响。需要注意的是，一个复制的分片不会存储在同一个节点中。

2、它允许你扩展你的搜索量，提高并发量，因为搜索可以在所有副本上并行的执行。

每个索引可以拆分成多个分片。索引可以复制零个或者多个分片。一旦复制，每个索引就有了主分片和副本分片。分片的数量和副本的数量可以在创建索引时定义。当创建索引后，你可以随时改变副本的数量，但你不能改变分片的数量。

默认情况下，每个索引分配 5 个分片和一个副本，这意味着你的集群节点至少要有两个节点，你将拥有 5 个主要的分片和 5 个副本分片共计 10 个分片。

注：每个 Elasticsearch 分片是一个 Lucene 的索引。有文档参考，你可以在一个单一的 Lucene 索引中存储的最大值为 lucene-5843，极限是 2147483519（= integer.max_value - 128）个文档。你可以使用_cat/shards api 监控碎片的大小。

- 索引(index)

索引是具有相同结构的文档集合。例如，可以有一个客户信息的索引，一个是产品目录的索引，一个订单数据的索引。一个索引在系统上是一个全部小写的名字，通过这个名字可以用来执行索引、搜索、更新和删除操作等。在一个单一的集群中，您可以定义多个你想要的索引。

- 类型(type)

在索引中，可以定义一个或多个类型。类型是索引的逻辑分区。在一般情况下，一种类型被定义为具有一组公共字段的文档。例如，让我们假设你运行一个博客平台，并把所有的数据存储在一个索引中。在这个索引中，你可以定义一种类型为用户数据，一种类型为博客数据，另一种类型为评论数据。

- 文档(document)

一个文档是一个 JSON 格式的字符串存储在 Elasticsearch 中。它就像在关系数据库中的表中的一行。每个存储在索引中的一个文件都有一个类型和一个 ID，每个文件都是一个 JSON 对象，存储了零个或者多个字段，或者键值对。原始的 JSON 文档被存储在一个叫做_source 的字段中。当搜索文档的时候默认返回的就是这个字段。

- 映射(mapping)

映射像关系数据库中的表结构，每一个索引都有一个映射，它定义了索引中的每一个字段类型，以及一个索引范围内的设置。一个映射可以事先被定义，或者在第一次存储文档的时候自动识别。

- 字段(field)

一个文档中包含零个或者多个字段，字段可以是一个简单的值（例如字符串、整数、日期），也可以是一个数组或对象的嵌套结构。字段类似于关系数据库中的表中的列。每个字段都对应一个字段类型，例如整数、字符串、对象等。字段还可以指定如何分析该字段的值。

- 来源字段(source field)

默认情况下，你的原文档将被存储在_source 这个字段中，当你查询的时候也是返回这个字段。这允许你可以从搜索结果中访问原始的对象，这个对象返回一个精确的 JSON 字符串，这个对象不显示索引分析后的其他任何数据。

- 主键(id)

ID 是一个文件的唯一标识，如果在存库的时候没有提供 ID，系统会自动生成一个 ID，文档的 index/type/id 必须是唯一的。

1.3.2 JSON 介绍

在 Elasticsearch 中的接口中，大多数都是以 JSON 的格式进行的，那 JSON 是什么呢？JSON(JavaScript Object Notation) 是一种轻量级的数据交换格式。易于人们阅读和编写，同时也易于机器解析和生成。它基于 JavaScript Programming Language, Standard ECMA-262 3rd Edition - December 1999 的一个子集。JSON 采用完全独立于语言的文本格式，但是也使用了类似于 C 语言家族的习惯(包括 C, C++, C#, Java, JavaScript, Perl, Python 等)。这些特性使 JSON 成为理想的数据交换语言。

JSON 建构于两种结构：

“名称/值”对的集合（A collection of name/value pairs）。不同的语言中，它被理解为对象（object），纪录（record），结构（struct），字典（dictionary），哈希表（hash table），有键列表（keyed list），或者关联数组 （associative array）。

值的有序列表（An ordered list of values）。在大部分语言中，它被理解为数组（array）。

这些都是常见的数据结构。事实上大部分现代计算机语言都以某种形式支持它们。这使得一种数据格式在同样基于这些结构的编程语言之间交换成为可能。

JSON 实例：

名称 / 值对

按照最简单的形式，可以用下面这样的 JSON 表示"名称 / 值对"：

```
{"firstName":"Brett"}
```

这个示例非常基本，而且实际上比等效的纯文本"名称 / 值对"占用更多的空间：

```
firstName=Brett
```

但是，当将多个"名称 / 值对"串在一起时，JSON 就会体现出它的价值了。首先，可以创建包含多个"名称 / 值对"的 记录，比如：

```
{"firstName":"Brett","lastName":"McLaughlin","email":"aaaa"}
```

从语法方面来看，这与"名称 / 值对"相比并没有很大的优势，但是在这种情况下 JSON 更容易使用，而且可读性更好。例如，它明确地表示以上三个值都是同一记录的一部分；花括号使这些值有了某种联系。

表示数组

当需要表示一组值时，JSON 不但能够提高可读性，而且可以减少复杂性。例如，假设你希望表示一个人名列表。在 XML 中，需要许多开始标记和结束标记；如果使用典型的名称 / 值对（就像在本系列前面文章中看到的那种名称 / 值对），那么必须建立一种专有的数据格式，或者将键名称修改为 person1-firstName 这样的形式。

如果使用 JSON，就只需将多个带花括号的记录分组在一起：

```
{
    "people":[
        {"firstName":"Brett","lastName":"McLaughlin","email":"aaaa"},
        {"firstName":"Jason","lastName":"Hunter","email":"bbbb"},
        {"firstName":"Elliotte","lastName":"Harold","email":"cccc"}
    ]
```

```
}
```

这不难理解。在这个示例中，只有一个名为 people 的变量，值是包含三个条目的数组，每个条目是一个人的记录，其中包含名、姓和电子邮件地址。上面的示例演示如何用括号将记录组合成一个值。当然，可以使用相同的语法表示多个值（每个值包含多个记录）：

```
{
    "programmers": [{
        "firstName": "Brett",
        "lastName": "McLaughlin",
        "email": "aaaa"
    }, {
        "firstName": "Jason",
        "lastName": "Hunter",
        "email": "bbbb"
    }],
    "authors": [{
        "firstName": "Isaac",
        "lastName": "Asimov",
        "genre": "sciencefiction"
    },{
        "firstName": "Sergei",
        "lastName": "Rachmaninoff",
        "instrument": "piano"
    }]
}
```

这里最值得注意的是，能够表示多个值，每个值进而包含多个值。但是还应该注意，在不同的主条目（programmers、authors 和 musicians）之间，记录中实际的名称 / 值对可以不一样。JSON 是完全动态的，允许在 JSON 结构的中间改变表示数据的方式。

在处理 JSON 格式的数据时，没有需要遵守的预定义的约束。所以，在同样的数据结构中，可以改变表示数据的方式，甚至可以以不同方式表示同一事物。

1.4 安装配置

下面是安装 Elasticsearch 所需要的几个步骤，接下来几节将详细说明。

1.4.1 安装 Java

为了运行 Elasticsearch，第一步是要安装 Java，Elasticsearch 需要 Java 7 或者更高版本的支持。建议使用最新的 Oracle 的 JDK 版本 1.8.0_72。如需了解 Java 的情况，可以到 Oracle 的官网找相关的资料。在你安装 Elasticsearch 前，请检查你的 Java 版本运行：

```
java -version
```

1.4.2 安装 Elasticsearch

当我们设置好 Java 后，下载最新的 Elasticsearch 版本，解压，安装完毕。下载地址：https://www.elastic.co/downloads/past-releases/elasticsearch-2-3-0。

环境变量

Elasticsearch 是 Java 开发的，所以 JVM 的环境变量 JAVA_OPTS 对 Elasticsearch 也是非常重要的。在 JAVA_OPTS 中对 Elasticsearch 最重要的参数是-Xmx 最大可以使用内存的参数，一般情况下大内存更能发挥 Elasticsearch 作用，建议-Xmx 设置为物理内存的一半，为了减少

内存分配带来的性能损耗，最好一开始就设置初始内存和最大内存都为物理内存的一半。即 Xms 和 Xmx 这两个参数。

由于 JAVA_OPTS 大多数时候对整个机器环境起作用，所以最好是保留默认的 JAVA_OPTS，最好用 ES_JAVA_OPTS 环境变量设置来作为 JAVA_OPTS 参数。

默认的配置文件在 elasticsearch/bin/elasticsearch.in.sh 中

```
if [ "x$ES_MIN_MEM" = "x" ]; then
    ES_MIN_MEM=256m
fi
if [ "x$ES_MAX_MEM" = "x" ]; then
    ES_MAX_MEM=1g
fi
if [ "x$ES_HEAP_SIZE" != "x" ]; then
    ES_MIN_MEM=$ES_HEAP_SIZE
    ES_MAX_MEM=$ES_HEAP_SIZE
fi

JAVA_OPTS="$JAVA_OPTS -Xms${ES_MIN_MEM}"
JAVA_OPTS="$JAVA_OPTS -Xmx${ES_MAX_MEM}"
```

ES_HEAP_SIZE 环境变量允许设置被分配到 Elasticsearch java 进程中堆内存的大小。最小和最大值将分配相同的值，可以通过设置 ES_MIN_MEM（默认为 256M）和 ES_MAX_MEM（默认为 1G）对堆内存进行设置

1.4.3 配置

Elasticsearch 配置文件在 elasticsearch/config 文件夹下。在这个文件夹中有两个文件，一个是 Elasticsearch 配置不同模块的配置文件 elasticsearch.yml ，另一个是 Elasticsearch 日志的配置文件 logging.yml。默认配置文件的格式为 YML。

Elasticsearch 提供了多种方式进行设置，在系统内部，都使用命名空间来表示这些设置，根据这些命名空间，系统可以很容易的扩展到其他格式。比如我们设置节点名称：

1、yml 格式为：node.name: node-1

2、JSON 格式为：只需要把 elasticsearch.yml 名修改成 elasticsearch.json

配置的方式为：

```
{
    "node" : {
        "name" : "node-1"
    }
}
```

3、通过 Elasticsearch 命令的参数来设置配置信息。例如：

```
elasticsearch -Des.node.name=node-1
```

4、elasticsearch 还可以通过交互式方式进行设置，通过${prompt.text}或者 ${prompt.secret} 来指定，${prompt.secret} 表示在终端中隐藏输入的值，${prompt.text}表示在终端中显示输入的值，例如：

```
node.name: ${prompt.text}
```

在 Elasticsearch 命令执行时，将提示你输入的实际值，例如下面的提示：

```
Enter value for [node.name]:
```

注意：当 Elasticsearch 作为服务或者在后台启动的时候，这两个参数则不起作用。

elasticsearch.yml 配置说明

集群名称：

cluster.name: my-application

确保在不同的环境中集群的名称不重复，否则，节点可能会连接到错误的集群上。

节点名称：

node.name: node-1

默认情况下，当节点启动时 Elasticsearch 将随机在一份 3000 个名字的列表中随机指定一个。如果机器上只运行一个集群 Elasticsearch 节点，可以用$ {HOSTNAME}设置节点的名称为主机名。

节点描述：

node.rack: r1

索引存储位置：

path.data: /path/to/data

日志存储位置：

path.logs: /path/to/logs

内存分配模式：

bootstrap.mlockall: true

绑定的网卡 IP：

network.host: 192.168.0.1

http 协议端口：

http.port: 9200

开始发现新节点的 IP 地址：

discovery.zen.ping.unicast.hosts: ["host1", "host2"]

最多发现节点的个数：

discovery.zen.minimum_master_nodes: 3

当重启集群节点后最少启动 N 个节点后开始做恢复：

gateway.recover_after_nodes: 3

在一台机器上最多启动的节点数：

node.max_local_storage_nodes: 1

当删除一个索引的时候，需要指定具体索引的名称：

action.destructive_requires_name: true

索引配置说明

在集群中创建的索引可以提供每个索引自己的设置。例如，下面创建一个索引刷新间隔是 5 秒钟而不是默认的刷新间隔（格式可以是 YAML 或 JSON）：

请求：PUT http://localhost:9200/kimchy

参数： index:refresh_interval: 5s

这个索引参数的设置可以设置在节点上，例如，在 elasticsearch.yml 文件中可以设置：

index.refresh_interval: 5s

这意味着除非索引明确定义，这个节点上创建的每个索引的刷新间隔为 5 秒。

当然也可以在启动 elasticsearch 的时候用参数指定：

elasticsearch -Des.index.refresh_interval=5s

日志配置说明

Elasticsearch 内部使用 log4j 记录系统日志，它试图通过使用 YAML 配置方式来简化 log4j 的配置，配置文件位置为 elasticsearch/config/ logging.yml。JSON 格式和键值对的格式也是支持的。可以加载多个配置文件，在启动 Elasticsearch 后系统自动合并多个配置文件。支持不同的后缀格式，例如： (.yml, .yaml, .json or .properties)。

记录器部分包含 java 包和相应的日志级别，在配置里可以省略 org.elasticsearch 前缀。Appender 部分包含日志描述信息。更多内容请参见 log4j。

由于日志比较重要，正常情况下不要禁止日志的产生，如果感觉日志过多，可以提高日志的级别。系统日志会每日生成一个新的文件。当遇到问题的时候，首先要检查一下日志文件，看看是否有出错的信息。

1.4.4 运行

在 windows 下执行 elasticsearch.bat，在 linux 下运行./elasticsearch。

如果一切顺利的话，你应该看到一堆像下面这样的信息：

[2016-02-03 16:53:31,122][INFO][node] [Rintrah] version[2.2.0], pid[6840], build[8ff36d1/2016-01-27T13:32:39Z]

[2016-02-03 16:53:31,122][INFO][node] [Rintrah] initializing ...

[2016-02-03 16:53:31,668][INFO][plugins] [Rintrah] modules [lang-groovy, lang-expression], plugins [], sites []

[2016-02-03 16:53:31,684][INFO][env] [Rintrah] using [1] data paths, mounts [[work (D:)]], net usable_space [67.2gb], net total_space [99.9gb], spins? [unknown], types [NTFS]

[2016-02-03 16:53:31,684][INFO][env] [Rintrah] heap size [910.5mb], compressed ordinary object pointers [true]

[2016-02-03 16:53:33,637][INFO][node] [Rintrah] initialized

[2016-02-03 16:53:33,637][INFO][node] [Rintrah] starting ...

[2016-02-03 16:53:33,918][INFO][transport] [Rintrah] publish_address {127.0.0.1:9300}, bound_addresses {[::1]:9300}, {127.0.0.1:9300}

[2016-02-03 16:53:33,934][INFO][discovery] [Rintrah] elasticsearch/1oo5dtelT8ax-3LmnTrs8g

[2016-02-03 16:53:37,982][INFO][cluster.service] [Rintrah] new_master {Rintrah}{1oo5dtelT8ax-3LmnTrs8g}{127.0.0.1}{127.0.0.1:9300}, reason: zen-disco-join(elected_as_master, [0] joins received)

[2016-02-03 16:53:40,363][INFO][gateway] [Rintrah] recovered [0] indices into cluster_state

[2016-02-03 16:53:40,567][INFO][http] [Rintrah] publish_address {127.0.0.1:9200}, bound_addresses {[::1]:9200}, {127.0.0.1:9200}

[2016-02-03 16:53:40,567][INFO][node] [Rintrah] started

这样我们就已经启动了 Elasticsearch，当然我们也可以在启动的时候修改集群的名称和节点的名称。例如：

./elasticsearch --cluster.name my_cluster_name --node.name my_node_name

默认情况下，Elasticsearch 使用 9200 端口提供的 REST API。该端口是可配置的。

在本机访问 http://127.0.0.1:9200/

将会得到以下内容：

```
{
  "name" : "Rintrah",
```

```
  "cluster_name" : "elasticsearch",
  "version" : {
    "number" : "2.2.0",
    "build_hash" : "8ff36d139e16f8720f2947ef62c8167a888992fe",
    "build_timestamp" : "2016-01-27T13:32:39Z",
    "build_snapshot" : false,
    "lucene_version" : "5.4.1"
  },
  "tagline" : "You Know, for Search"
}
```


现在，我们的节点（和集群）将正确运行，下一步就是要了解如何进行使用。Elasticsearch 提供了非常全面和强大的 REST API，通过这些 API，我们可以了解集群的信息。这些 API 可以做如下事情：

1、检查集群，节点和索引的情况、状态和统计

2、管理集群、节点、索引数据和文档数据

3、执行 CRUD（创建，读取，更新和删除）操作，可以对索引进行操作。

4、执行高级搜索操作，如分页、排序、过滤、脚本、聚合及其他操作。

1.4.5 停止

如果需要停止 Elasticsearch，有以下几种方法。

- 第一种方法：如果节点是连接到控制台，按下 Ctrl+C。
- 第二种方法：杀进程(Linux 是 kill，windows 下用任务管理器)。

由于 Elasticsearch 常用在集群中，集群停止相对有些复杂。参考集群升级。

1.4.6 作为服务

linux 下作为服务

Elasticsearch 创建了 debian 安装包和 RMP 安装包，可以在官网的下载页面中进行下载。安装包需要依赖 Java，除此就没有任何依赖。

在 debian 系统下可以使用标准的系统工具，init 脚本放在 /etc/init.d/elasticsearch 下，配置文件默认放在 /etc/default/elasticsearch 下。从 debian 软件包安装好后默认是不启动服务的。其原因是为了防止实例不小心加入集群。安装好后用 dpkg -i 命令来确保，当系统启动后启动 Elasticsearch 需要运行下面的两个命令：

```
sudo update-rc.d elasticsearch defaults 95 10
sudo /etc/init.d/elasticsearch start
```

当用户运行 Debian8 或者 Ubuntu14 或者更高版本的时候，系统需要用 systemd 来代替 update-rc.d，在这种情况下，请使用 systemd 来运行，参见下面的介绍。

Elasticsearch 通常的建议是使用 Oracle 的 JDK。可以用下面的命令安装。

```
sudo add-apt-repository ppa:webupd8team/java
sudo apt-get update
sudo apt-get install oracle-java8-installer
java -version
```

基于 RPM 的系统一般使用 chkconfig 来启用和禁用服务。init 脚本位于 /etc/init.d/elasticsearch 下，配置文件放在/etc/sysconfig/Elasticsearch 下。同 Debian 系统类似，安装好后也不会自动加入自启动服务中。需要手工指定。

```
sudo /sbin/chkconfig --add elasticsearch
sudo service elasticsearch start
```

systemd 服务启动

很多 linux 系统，例如 Debian Jessie，Ubuntu 14 等，系统不使用 chkconfig 来注册服务，取而代之的是用 systemd 来启动和停止服务。命令是 /bin/systemctl 来启动和停止服务。RPM 包安装的配置文件在/etc/sysconfig/elasticsearch 下，deb 包安装的配置文件在 /etc/default/elasticsearch 下。安装 RPM 之后，你必须改变系统配置，然后启动 Elasticsearch。

sudo /bin/systemctl daemon-reload

sudo /bin/systemctl enable elasticsearch.service

sudo /bin/systemctl start elasticsearch.service

同时注意改变在/etc/sysconfig/elasticsearch 中的 MAX_MAP_COUNT 设置是没有任何效果的。需要改变/usr/lib/sysctl.d/elasticsearch.conf 中的配置才起作用。

Windows 下作为服务

Windows 用户可以配置 Elasticsearch 作为服务在后台运行，或在没有任何用户交互启动时自动启动。这可以通过 bin 目录下的 service.bat 脚本来实现，可以安装，卸载，管理或配置服务命令行为：service.bat install|remove|start|stop|manager [SERVICE_ID]

SERVICE_ID 是服务 ID，可以不用指定用默认的值，系统可以安装多个服务。Manager 是启动图形界面的配置。例如运行：service install 后显示的内容如下

Installing service : "elasticsearch-service-x64"

Using JAVA_HOME (64-bit): "C:\Program Files\Java\jdk1.7.0_79"

The service 'elasticsearch-service-x64' has been installed.

安装好后，有两种方法可以对服务进行设置。

1、图形化界面，可以用命令 service manager 来启动图形界面。执行后显示界面如下。

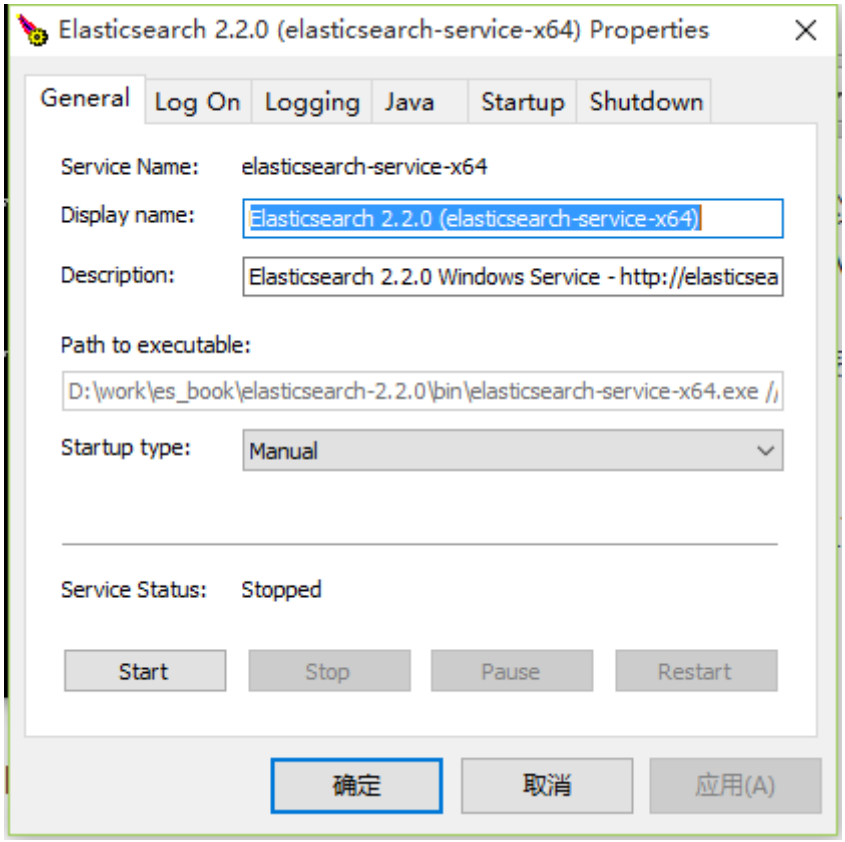

图 1.3 服务设置图形化界面

2、用命令 service start ，service stop 来启动停止服务。

还有一个社区支持的可定制 MSI 安装程序：
https://github.com/salyh/elasticsearch-msi-installer 也可以安装成服务。

1.4.7 版本升级

Elasticsearch 通常可以使用滚动升级过程，导致服务不中断。本文详细介绍如何执行滚动升级与集群的升级重启。Elasticsearch 不是所有版本都可以直接升级。升级前请先查阅相关文档，然后进行数据备份，最好在测试环境下模拟做一遍。升级可以参照以下内容。

从 0.90.x 到 2.x 需要全集群重启。

从 1.x 到 2.x 需要全集群重启。

从 2.x 到 2.y 可以不用全部重启，使用滚动升级完成（y>x）。

插件问题，在升级的时候，同时要考虑到插件问题，最好插件的版本和 Elasticsearch 版本一致。

在每个节点上升级和重启的过程为：

1、关闭 Elasticsearch

2、升级 Elasticsearch

3、升级插件

4、启动 Elasticsearch

集群重启步骤：

1、关闭分片分配。

当我们试图关闭一个节点的时候，Elasticsearch 会立即试图复制这个节点的数据到集群中的其他节点上。这将导致大量的 IO 请求。在关闭该节点的时候可以通过设置一下参数来避免此问题的发生。

```
PUT /_cluster/settings
{
  "transient": {
    "cluster.routing.allocation.enable": "none"
  }
}
```

2、执行一个同步刷新

当停止一个索引的时候，分片的恢复会很快，所以要进行同步刷新请求。

```
POST /_flush/synced
```

同步刷新请求是非常有效的一种操作，当任何索引操作失败的时候，可以执行同步刷新请求，必要的时候可以执行多次。

3、关闭和升级所有节点

停止在集群中的所有节点上的服务。每一个节点都要进行单独升级。这个主要就是文件替换操作，注意保留日志目录。

4、启动集群

如果你有专门的主节点（node.master 节点设置为 true 和 node.data 设置为 false），则先启动主节点。等待它们形成一个集群，然后选择一个主数据节点进行启动。你可以通过查看日志来检查启动情况。通过下面命令可以监控集群的启动情况，检查所有节点是否已成功加入集群。

```
GET _cat/health
GET _cat/nodes
```

5、等待黄色集群状态

当节点加入集群后，它首先恢复存储在本地的主分片数据。最初的时候，通过 _cat/health 请求发现集群的状态是红色，意味着不是所有的主分片都已分配。当每个节点都恢复完成后，集群的状态将会变成黄色，这意味着所有主分片已经被找到，但是并不是所有的副本分片都恢复。

6、重新分配

延迟副本的分配直到所有节点都加入集群，在集群的所有节点，可以重新启用碎片分配：

```
PUT /_cluster/settings
{
  "persistent": {
    "cluster.routing.allocation.enable": "all"
  }
}
```

这个时候集群将开始复制所有副本到数据节点上，这样可以安全的恢复索引和搜索。如果你能延迟索引和搜索直到所有的碎片已经恢复，这样可以加快集群的恢复。可以通过下面 API 监控恢复的进度和健康情况：

```
GET _cat/health
GET _cat/recovery
```

最后当集群的状态出现绿色的时候，表示本次集群升级全部完成。

滚动升级

滚动升级允许 Elasticsearch 集群升级一个节点，同时又不影响系统的使用。在同一个集群中的所有节点的版本最好保持一致，否则可能会产生不可预知的后果。滚动升级的步骤如下：

1、关闭分片分配。

当我们视图关闭一个节点的时候，Elasticsearch 会立即试图复制这个节点的数据到集群中的其它节点上。这将导致大量的 IO 请求。在关闭该节点的时候可以通过设置一下参数来避免此问题的发生。

```
PUT /_cluster/settings
{
  "transient": {
    "cluster.routing.allocation.enable": "none"
  }
}
```

2、停止不必要的索引和执行同步刷新（可选）

你可以在升级过程中继续索引。然而，如果你暂时停止不必要的索引碎片，但它恢复要快得多。所以可以执行同步刷新操作。

```
POST /_flush/synced
```

同步刷新请求是非常有效的一种操作，当任何索引操作失败的时候，可以执行同步刷新请求，必要的时候可以执行多次。

3、停止和升级一个节点

在启动升级前，将节点中的一个节点关闭。可以通过绿色解压安装或者通过 RPM 等安装包安装。

不管是解压安装还是压缩包安装都要保留之前的数据文件不能被破坏。可以在新的目录中安装，把 path.conf 和 path.data 的位置指向之前的数据。

4、启动升级节点

启动“升级”节点，并通过接口检查是否正确：

```
GET _cat/nodes
```

5、启用共享配置

一旦节点加入集群，在节点重新启用碎片分配：

```
PUT /_cluster/settings
{
   "transient": {
      "cluster.routing.allocation.enable": "all"
   }
}
```

6、等待节点恢复

应该在集群下一个节点升级之前完成碎片分配。可以通过以下接口进行检查：

```
GET _cat/health
```

等待的状态栏从黄到绿色。状态绿色意味着所有主分片和副本分片已经完成分配。

重要：在滚动升级期间，主碎片被分配到一个更高版本节点不会有副本分配到一个较低的版本节点，因为新版本的数据结构可能旧版本不能识别。

一旦另一个节点升级，副本将被分配，集群的健康状态将达到绿色。

没有同步刷新碎片可能需要一些时间来恢复。恢复的状态和每个节点的监控可以用以下接口。

```
GET _cat/recovery
```

如果你停止了索引，那么恢复索引的恢复是安全的。

7、重复其他节点

当集群是稳定的，节点已经恢复，重复上述步骤，把所有剩余的节点进行升级。

1.5 对外接口

Elasticsearch 对外提供的 API 是以 Http 协议的方式，通过 JSON 格式以 REST 约定对外提供。

HTTP 配置文件是放在 elasticsearch.yml 中，注意的是所有与 HTTP 配置相关的内容都是静态配置，也就是需要重启后才生效。HTTP 对外接口模块是可以禁用的，只需要设置 http.enabled 为 false。Elasticsearch 集群中的通信是通过内部接口实现的，而不是 HTTP 协议。在集群中不需要所有节点都开启 HTTP 协议，正常情况下，只需要在一个节点上开启 HTTP 协议。

1.5.1 API 约定

多索引参数

大多数 API 支持多索引查询，就是同时可以查询多个索引中的数据，例如，参数 test1，test2，test3 表示同时搜索 test1，test2，test3 三个索引中的数据(或者用_all 全部索引，_all 是内部定义的关键字)。在参数中同时支持通配符的操作，例如 test*表示查询所有以 test 开头的索引。同时也支持排除操作，例如+test*，-test3 表示查询所有 test 开头的索引，排除 test3。同时多索引查询还支持以下参数：

ignore_unavailable：当索引不存在或者关闭的时候，是否忽略这些索引，值为 true 和 false。

allow_no_indices：当使用通配符查询所有索引的时候，当有索引不存在的时候是否返回查询失败。值为 true 和 false。

expand_wildcards ：控制通配符索引表达式匹配具体哪一类的索引，值为 open，close，none，all。open 表示只支持开启状态的索引，close 表示只支持关闭状态的索引，none 表示不可用，all 表示同时支持 open 和 close 索引。

备注：文档操作 API 和索引别名 API 不支持多索引参数。

日期筛选

日期筛选可以限定时间序列索引的搜索范围，而不是搜索全部内容，通过时间限定，可以从集群中减少搜索的内容，提高搜索效率和减少资源占用。例如只搜索最近两天的错误日志。

备注：几乎所有的 API 都支持日期筛选。

日期筛选的语法为：

<static_name{date_math_expr{date_format|time_zone}}>

语法解释：

static_name ：索引的名称；

date_math_expr：动态日期计算表达式；

date_format ：日期格式；

time_zone：时区，默认为 UTC。

例如：

```
curl -XGET 'localhost:9200/<logstash-{now%2Fd-2d}>/_search' {
   "query" : {
      ...
   }
}
```

备注：由于 URL 编码的问题/被替换成了%2F。

假设当前时间为 2013.7.17 日中午，下面列举几个例子：

<secilog-{now/d}>:secilog-2013. 07.17

<secilog-{now/M}>secilog-2013.07.01

<secilog-{now/M{YYYY.MM}}>secilog-2013.07

<secilog-{now/M-1M{YYYY.MM}}>secilog-2013.06

<secilog-{now/d{YYYY.MM.dd|+12:00}}secilog-2013.7.18

备注：如果索引名称有{}，可以通过添加\来转义，如：

<elastic\\{ON\\}-{now/M}>被转换为 elastic{ON}-2013.07.01

时间搜索也可以通过逗号，来选择多个时间，例如，选择最近三天的数据：

```
curl -XGET
'localhost:9200/<secilog-{now%2Fd-2d}>,<secilog-{now%2Fd-1d}>,<secilog-{now%2Fd}>/_search'
{
   "query" : {
      ...
   }
}
```

通用参数

pretty 参数，当你在任何请求中添加了参数?pretty=true 时，请求的返回值是经过格式化后的 JSON 数据，这样阅读起来更加的方便。系统还提供了另一种格式的格式化，?format=yaml，YAML 格式，这将导致返回的结果具有可读的 YAML 格式。

human 参数，对于统计数据，系统支持计算机数据，同时也支持比较适合人类阅读的数据。比如，计算机数据"exists_time_in_millis": 3600000 or "size_in_bytes": 1024。更适合人类阅读的数据："exists_time": "1h" or "size": "1kb"。当?human=false 的时候，只输出计算机数据，当?human=true 的时候输出更适合人类阅读的数据，但这会消耗更多的资源，默认是 false。

日期表达式，大多数参数接受格式化日期表达式 ，如范围查询 gt（大于）和 lt（小于），或在日期聚合中用 from to 来表达时间范围。表达式设定的日期为 now 或者日期字符串加||。

- +1h - 增加一小时
- -1d - 减少一个小时
- /d - 上一个小时

所支持的时间单位为：y（年）、M（月）、w（周）、d（日）、h（小时）、m（分钟）、s（秒）。

例如：

now+1h ： 当前时间加一小时，以毫秒为单位。

now+1h+1m ： 当前时间加一小时和一分钟，以毫秒为单位。

now+1h/d ： 当前时间加一小时，四舍五入到最近的一天。

2015-01-01||+1M/d ： 2015-01-01 加一个月，向下舍入到最接近的一天。

响应过滤(filter_path)。所有的返回值可以通过 filter_path 来减少返回值的内容，多个值可以用逗号分开。例如：

```
curl -XGET 'localhost:9200/_search?pretty&filter_path=took,hits.hits._id,hits.hits._score'{
  "took" : 3,
  "hits" : {
    "hits" : [
      {
        "_id" : "3640",
        "_score" : 1.0
      },
      {
        "_id" : "3642",
        "_score" : 1.0
      }
    ]
  }
}
```

它也支持通配符*匹配任何部分字段的名称，例如：

```
curl -XGET 'localhost:9200/_nodes/stats?filter_path=nodes.*.ho*'{
  "nodes" : {
    "lvJHed8uQQu4brS-SXKsNA" : {
      "host" : "portable"
    }
  }
```

```
}
```

我们可以用两个通配符**来匹配不确定名称的字段，例如我们可以返回 Lucene 版本的段信息：

```
curl 'localhost:9200/_segments?pretty&filter_path=indices.**.version'{
  "indices" : {
    "movies" : {
      "shards" : {
        "0" : [ {
          "segments" : {
            "_0" : {
              "version" : "5.2.0"
            }
          }
        } ],
        "2" : [ {
          "segments" : {
            "_0" : {
              "version" : "5.2.0"
            }
          }
        } ]
      }
    },
    "books" : {
      "shards" : {
        "0" : [ {
          "segments" : {
            "_0" : {
              "version" : "5.2.0"
            }
          }
        } ]
      }
    }
  }
}
```

注意，有时直接返回 Elasticsearch 的某个字段的原始值，如_source 字段。如果你想过滤_source 字段，可以结合_source 字段和 filter_path 参数，例如：

```
curl -XGET 'localhost:9200/_search?pretty&filter_path=hits.hits._source&_source=title'{
  "hits" : {
    "hits" : [ {
      "_source":{"title":"Book #2"}
    }, {
      "_source":{"title":"Book #1"}
```

```
        }, {
          "_source":{"title":"Book #3"}
        } ]
      }
    }
```

紧凑参数 flat_settings，flat_settings 为 true 时候返回的内容更加的紧凑，false 是返回的值更加的容易阅读。例如为 true 的时候：

```
{
  "persistent" : { },
  "transient" : {
    "discovery.zen.minimum_master_nodes" : "1"
  }
}
```

为 false 的时候，默认的情况下为 false：

```
{
  "persistent" : { },
  "transient" : {
    "discovery" : {
      "zen" : {
        "minimum_master_nodes" : "1"
      }
    }
  }
}
```

基于 URL 的访问控制

当多用户通过 URL 访问 Elasticsearch 索引的时候，为了防止用户误删除等操作，可以通过基于 URL 的访问控制来限制用户对某个具体索引的访问。可以在配置文件中添加参数：rest.action.multi.allow_explicit_index: false。这个参数默认为 true。当为 false 的时候。在请求参数中指定具体索引的请求将会被拒绝。

1.5.2 REST 介绍

REST (REpresentational State Transfer)，从字面就是"表述性状态传输"，它通常是开发的一种约定，当所有的开发者都遵从这种约定的时候，可以大大的简化开发的沟通成本。REST 约定用 http 的请求头 POST，GET，PUT，DELETE 正好可以对应 CRUD (Create，Read，Update，Delete) 四种数据操作。如果你设计的应用程序能符合 REST 原则 (REST principles)，这些符合 REST 原则的 REST 服务可称为 "RESTful web service" 也称 "RESTful Web API"。

HTTP 方法	数据处理	说明
POST	Create	新增一个没有 id 的资源
GET	Read	取得一个资源

PUT	Update	更新一个资源。或新增一个含 id 的资源(如果 id 不存在)
DELETE	Delete	删除一个资源

表格 1.3 REST 请求头

1.5.3 Head 插件安装

工欲善其事必先利其器，在学习 Elasticsearch 的过程中也要借助一些工具来进行。官方文档中的模拟工具用的是 curl，这是控制台工具，不是很直观。所以在本书中用到了 head 插件来作为请求的工具。

插件安装，在 Elasticsearc/bin 目录下执行下面的命令：

plugin install mobz/elasticsearch-head

安装成功后，启动 Elasticsearch，然后在浏览器中输入：

http://127.0.0.1:9200/_plugin/head/

可以看到如下界面，表示安装成功。

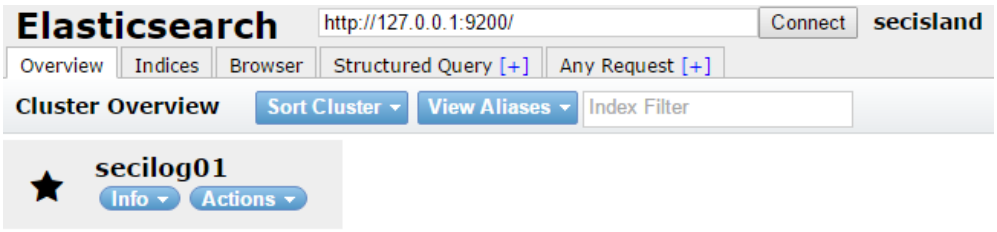

图 1.4 Head 插件整体视图

在本书大多数的接口操作中使用 Any Request 面板进行交互，界面说明如下：

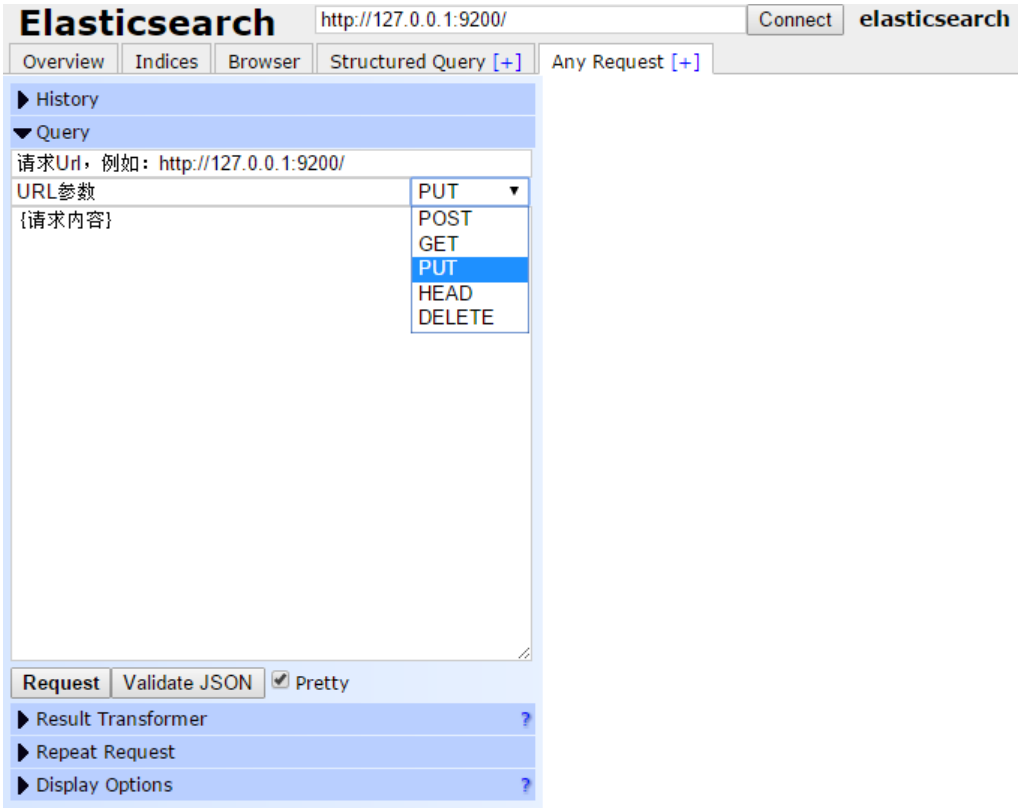

图 1.5　　Head 插件请求视图

注意：在 URL 中要用 127.0.0.1，用 localhost 不起作用。

1.5.4　创建库

在请求 URL 中输入：PUT 127.0.0.1:9200/secisland?pretty

在请求的方法中选择 PUT。

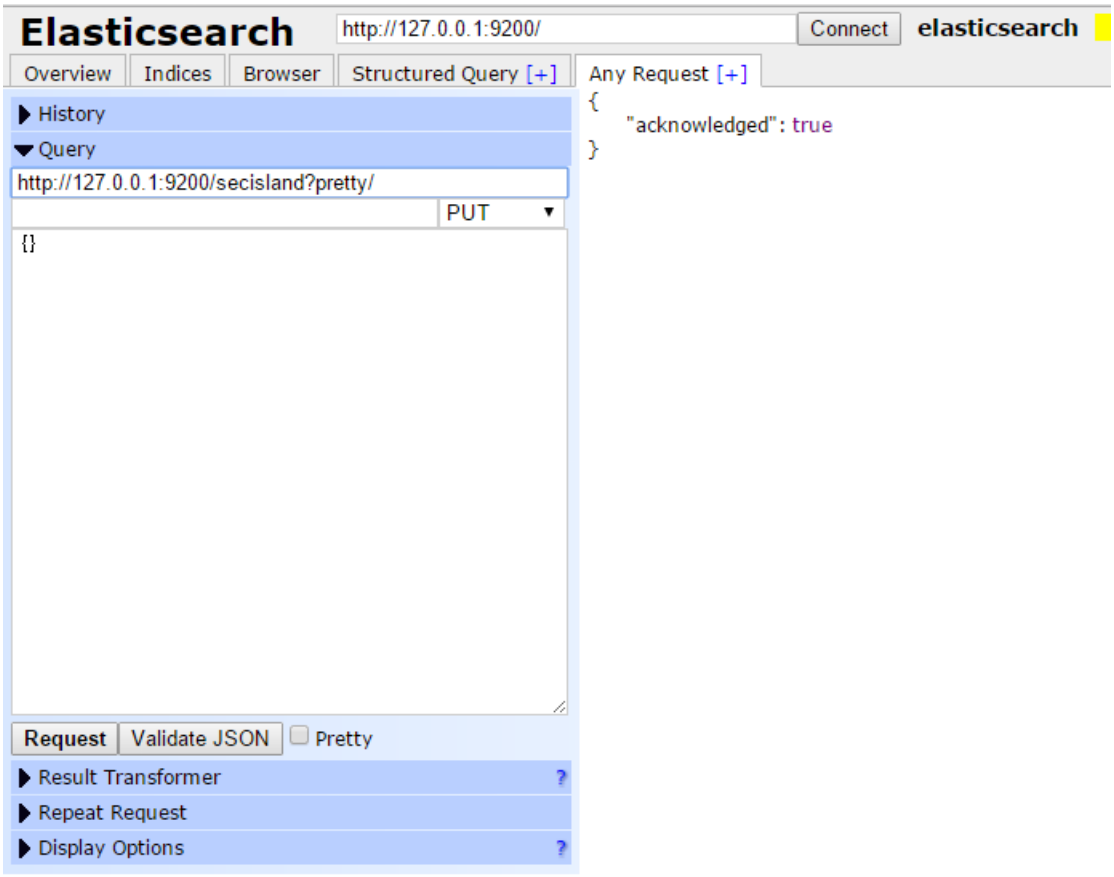

图 1.6　　创建索引库示意图

点击 Request 后可以在右边看到返回的内容如下，表示建库成功：

```
{
  "acknowledged" : true
}
```

执行完建库后然后查询一下库的状态，有两种方式查看，如果用命令查看，可以在浏览器中执行 http://localhost:9200/_cat/indices?v

返回：

health status index pri rep docs.count docs.deleted store.size pri.store.size

yellow open secisland 5 1 0 0 650b 650b

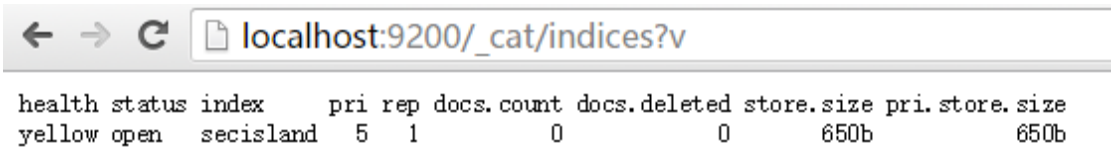

图 1.7　　索引库状态查询结果

表示已经建成了一个索引 secisland，主分片是 5 个，健康度是黄色，状态是活动，文档数为 0。

或者在 head 插件中进行查看：

图 1.8　　索引创建结果视图

备注：在后面的例子中都用下面的方式进行操作。

请求：{请求参数，例如：POST}　{URL：例如：http://127.0.0.1:9200/secisland?pretty}

参数：{参数内容}

返回结果：{返回结果}

1.5.5　　插入数据

请求　PUT http://127.0.0.1:9200/secisland/secilog/1/

参数：

```
{
   "computer": "secisland",
   "message": "secisland is an security company!"
}
```

返回值：

```
{
   "_index": "secisland",
   "_type": "secilog",
   "_id": "1",
   "_version": 1,
   "_shards": {
            "total": 2,
   "successful": 1,
   "failed": 0
},
        "created": true
}
```

1.5.6　　修改文档

请求　POST http://127.0.0.1:9200/secisland/secilog/1/_update

参数：

```
{
   "doc": {
      "computer": "secisland",
      "message": "secisland is an security computer. It provides log analysis products !"
```

```
    }
}
```

返回值：

```
{
    "_index": "secisland",
    "_type": "secilog",
    "_id": "1",
    "_version": 2,
    "_shards": {
        "total": 2,
        "successful": 1,
        "failed": 0
    }
}
```

1.5.7 查询文档

请求：GET http://127.0.0.1:9200/secisland/secilog/1/

返回值：

```
{
    "_index": "secisland",
    "_type": "secilog",
    "_id": "1",
    "_version": 2,
    "found": true,
    "_source": {
        "computer": "secisland",
        "message": "secisland is an security computer. It provides log analysis products !"
    }
}
```

1.5.8 删除文档

请求：DELETE http://127.0.0.1:9200/secisland/secilog/1/

返回值：

```
{
    "found": true,
    "_index": "secisland",
    "_type": "secilog",
    "_id": "1",
    "_version": 3,
    "_shards": {
        "total": 2,
        "successful": 1,
        "failed": 0
    }
}
```

1.5.9 删除库

请求：DELETE http://127.0.0.1:9200/secisland/

返回值：

```
{
"acknowledged": true
}
```

1.6 JAVA 接口

1.6.1 JAVA 接口说明

Elasticsearch 本身是 JAVA 开发的，天生对 JAVA 的支持能力是最好的，所以用 JAVA 来开发 Elasticsearch 应该是一个不错的选择。可以把 JAVA 开发看成 CS 模式，客户端请求，服务端响应，所有的操作是完全异步的。此外在客户端上操作，很多的操作可以在客户端上来完成，增加了系统效率。JAVA 客户端和服务端都是使用相同的程序。

mvn 仓库配置

在 mvn 项目中可以在 pom.xml 增加 mvn 仓库。

```
<dependency>
    <groupId>org.elasticsearch</groupId>
    <artifactId>elasticsearch</artifactId>
    <version>${es.version}</version>
</dependency>
```

打成独立的 jar 包。如果你想打成独立的 jar 包，并包含所有的依赖。不能使用 mvn 的 maven-assembly-plugin 插件，因为他不能很好的处理 META-INF/services 结构。我们可以用 maven-shade-plugin 插件来进行打包。例如：

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-shade-plugin</artifactId>
    <version>2.4.1</version>
    <executions>
        <execution>
            <phase>package</phase>
            <goals><goal>shade</goal></goals>
            <configuration>
                <transformers>
                    <transformer
implementation="org.apache.maven.plugins.shade.resource.ServicesResourceTransformer"/>
                </transformers>
            </configuration>
        </execution>
    </executions></plugin>
```

如果你有一个 main 函数，可以通过调用 java -jar yourjar.jar 运行时，只需要在插件配置中加入一句话：例如：

```
<transformer
implementation="org.apache.maven.plugins.shade.resource.ManifestResourceTransformer">
    <mainClass>com.secisland.es.demo.main</mainClass>
</transformer>
```

可以使用多种方式来调用 JAVA 客户端。

一种方式是把节点加入集群，不存储数据。

另一种是把节点作为客户端连接到集群。

注意：客户端版本要和集群版本一致，否则有可能出现不可预知的错误。

嵌入式节点客户端

实例化一个基于节点的客户，这是对 Elasticsearch 执行操作的最简单的方法。例如：

```
import static org.elasticsearch.node.NodeBuilder.*;
// on startup
Node node = nodeBuilder().node();
Client client = node.client();
// on shutdown
node.close();
```

当你开始一个节点，它加入了一个 Elasticsearch 集群。你可以通过设置 cluster.name 来配置不同的集群，或显式使用 clusterName 方法来创建对象。在项目工程中可以定义 cluster.name，配置在/src/main/resources/elasticsearch.yml 文件中。只要在 classpath 中能找到 elasticsearch.yml 文件，节点启动的时候就会加载此文件。配置文件中定义：

```
cluster.name: secilog
```

或者在 java 代码中指定：

```
Node node = nodeBuilder().clusterName("secilog").node();
Client client = node.client();
```

使用客户端的好处是依赖 Elasticsearch 服务端的特性把需要执行的操作自动路由到节点，当加入集群节点作为客户端时，最重要的是不能在此节点上增加数据保存，这可以通过配置来完成。把 node.data 设置为 false 或 node.client 设置为 true。这都可以通过 NodeBuilder 的方法来实现。

```
import static org.elasticsearch.node.NodeBuilder.*;

// on startup
//  嵌入式节点可以不用打开 http 端口
Node node =
      nodeBuilder()
            .settings(Settings.settingsBuilder().put("http.enabled", false))
            .client(true)
      .node();

Client client = node.client();

// on shutdown
node.close();
```

另一个常见的客户端用处是用在单元/集成测试中。在这种情况下，只需要启动一个“本地(local)”节点。这里的 local 节点的意思是公用一个 Java 虚拟机(JVM)。在这种情况下两个服务会互相发现自己并形成一个集群。例如：

```
import static org.elasticsearch.node.NodeBuilder.*;
```

```
// on startup
Node node = nodeBuilder().local(true).node();
Client client = node.client();

// on shutdown
node.close();
```

嵌入式节点的缺点：

频繁启动和停止一个或多个节点在集群上会创建不必要的噪声。

就像其它节点一样嵌入式节点客户端会响应外部请求。

注意：用 eclipse 开发的时候有可能会报错。

```
Exception in thread "main" java.lang.IllegalStateException: path.home is not configured
```

解决方法：在 main 方法的类中右键选择 run configurations，在 Arguments 标签页下的 vm arguments 输入：-Des.path.home=-Des.path.home 即可。比如：

```
-Des.path.home=D:\elasticsearch-2.2.0
```

通过连接的客户端

客户端通过 TransportClient 对象可以使用远程连接的方式连接 Elasticsearch 集群。这种情况下不用加入集群，比较像传统的 CS 程序的架构，比如数据库连接。例如：

```
// on startup
Client client = TransportClient.builder().build().addTransportAddress(new
InetSocketTransportAddress(InetAddress.getByName("host1"), 9300))
        .addTransportAddress(new
InetSocketTransportAddress(InetAddress.getByName("host2"), 9300));

// on shutdown
client.close();
```

注意：如果集群名称不是“Elasticsearch”，你必须设置集群名称或者使用 elasticsearch.yml 配置文件在客户端工程中配置。可以连接一个集群中的多个节点。

```
Settings settings = Settings.settingsBuilder()
        .put("cluster.name", "secilog").build();
Client client = TransportClient.builder().settings(settings).build();
```

你可以设置 client.transport.sniff 为 true 来使客户端去嗅探整个集群的状态，把集群中其它机器的 IP 地址加到客户端中。这样做的好处是一般你不用手动设置集群里所有集群的 IP 到连接客户端，它会自动帮你添加，并且自动发现新加入集群的机器。

```
Settings settings = Settings.settingsBuilder()
        .put("client.transport.sniff", true).build();
TransportClient client = TransportClient.builder().settings(settings).build();
```

其他参数说明

参数　　描述

client.transport.ignore_cluster_name　设置为 true 的时候忽略连接节点时的集群名称验证。

client.transport.ping_timeout　　等待一个节点的 ping 响应的时间，默认 5 秒。

client.transport.nodes_sampler_interval　　监听和连接节点的频率，默认 5 秒。

1.6.2 创建索引文档

引入的头文件：

```
import static org.elasticsearch.node.NodeBuilder.nodeBuilder;

import java.io.IOException;
import java.net.InetAddress;
import java.util.Date;
import java.util.Map;
import java.util.Set;

import org.elasticsearch.action.admin.cluster.health.ClusterHealthResponse;
import org.elasticsearch.action.admin.indices.create.CreateIndexRequestBuilder;
import org.elasticsearch.action.admin.indices.create.CreateIndexResponse;
import org.elasticsearch.action.index.IndexResponse;
import org.elasticsearch.client.Client;
import org.elasticsearch.client.ClusterAdminClient;
import org.elasticsearch.client.transport.TransportClient;
import org.elasticsearch.cluster.health.ClusterIndexHealth;
import org.elasticsearch.common.settings.Settings;
import org.elasticsearch.common.transport.InetSocketTransportAddress;
import org.elasticsearch.common.xcontent.XContentBuilder;
import org.elasticsearch.common.xcontent.XContentFactory;
import org.elasticsearch.node.Node;
import static org.elasticsearch.common.xcontent.XContentFactory.*;
//上面的内容对下面的几个示例同样起作用。

XContentBuilder mapping = XContentFactory.jsonBuilder()
        .startObject()
            .startObject("settings")
               .field("number_of_shards", 1)//设置分片数量
               .field("number_of_replicas", 0)//设置副本数量
            .endObject()
        .endObject()
        .startObject()
            .startObject(type)//type 名称
                .startObject("properties") //下面是设置文档列属性。
                    .startObject("type")
                        .field("type", "string")
                        .field("store", "yes")
                    .endObject()
                    .startObject("eventCount")
                          .field("type", "long")
                          .field("store", "yes")
                    .endObject()
```

```
                    .startObject("eventDate")
                        .field("type", "date")
                        .field("format", "dateOptionalTime")
                        .field("store", "yes")
                    .endObject()
                    .startObject("message")
                        .field("type", "string")
                        .field("index", "not_analyzed")
                        .field("store", "yes")
                    .endObject()
                .endObject()
            .endObject()
        .endObject();

    CreateIndexRequestBuilder cirb = client
            .admin()
            .indices()
            .prepareCreate(indexName)//index 名称
            .setSource(mapping);

    CreateIndexResponse response = cirb.execute().actionGet();
    if (response.isAcknowledged()) {
        System.out.println("Index created.");
    } else {
        System.err.println("Index creation failed.");
    }
```

1.6.3 增加文档

```
    IndexResponse response = client
            .prepareIndex(indexName, type, "1")
            .setSource(//这里可以直接用 json 字符串
                    jsonBuilder().startObject()
                        .field("type", "syslog")
                        .field("eventCount", 1)
                        .field("eventDate", new Date())
                        .field("message", "secilog insert doc test")
                    .endObject()).get();
    System.out.println("index:"+response.getIndex()
            +" insert doc id:"+response.getId()
            +" result:"+response.isCreated());
```

1.6.4 修改文档

修改文档有两种方式，一种是直接修改，另一种是如果文档不存在则插入存在再修改。

第一种代码

```
    UpdateRequest updateRequest = new UpdateRequest();
    updateRequest.index(indexName);
```

```
updateRequest.type(type);
updateRequest.id("1");
updateRequest.doc(jsonBuilder()
        .startObject()
            .field("type", "file")
        .endObject());
client.update(updateRequest).get();
```

第二种代码：

```
IndexRequest indexRequest = new IndexRequest(indexName, type, "3")
.source(jsonBuilder()
    .startObject()
        .field("type", "syslog")
        .field("eventCount", 2)
        .field("eventDate", new Date())
        .field("message", "secilog insert doc test")
    .endObject());
UpdateRequest updateRequest = new UpdateRequest(indexName, type, "3")
    .doc(jsonBuilder()
        .startObject()
            .field("type", "file")
        .endObject())
    .upsert(indexRequest);
client.update(updateRequest).get();
```

1.6.5 查询文档

```
GetResponse response = client.prepareGet("secilog", "log", "1").get();
String source = response.getSource().toString();
long version = response.getVersion();
String indexName = response.getIndex();
String type = response.getType();
String id = response.getId();
```

1.6.6 删除文档

```
DeleteResponse dresponse = client.prepareDelete("secilog", "log", "4").get();
boolean isFound = dresponse.isFound(); //文档存在返回 true,不存在返回 false；
```

删除索引

```
DeleteIndexRequest delete = new DeleteIndexRequest("secilog");
client.admin().indices().delete(delete);
```

1.7 小结

本章介绍了全文搜索 Elasticsearch 的基本概念和历史，介绍了需要掌握 Elasticsearch 的相关的基本知识。Elasticsearch 的安装、配置和升级操作。使用 Elasticsearch REST API 和 JAVA 接口来索引、更新、检索，最终删除数据。通过本章的了解，应该对 Elasticsearch 有了基本的认识。

2 索引

在上一章节的例子中，我们介绍了全文搜索和 Elasticsearch 的基础知识。我们创建了一个默认的设置来创建索引，检索数据并更新了数据，同时介绍了 JAVA 开发接口相关的一些知识和例子。本章我们将介绍以下内容：

索引管理的一些高级用法；

索引映射管理；

索引别名管理；

索引详细设置；

索引监控管理；

索引状态管理；

文档操作管理。

2.1 索引管理

前面的介绍已经启动了集群并创建了默认的索引，除了默认的索引配置外，我们还可以通过索引管理来对索引进行更高级别的管理。首先了解一下创建索引的过程。

2.1.1 创建索引

创建索引的时候可以通过修改 number_of_shards 和 number_of_replicas 参数的数量来修改主分片和副本分片的数量。在默认的情况下主分片和副本分片的数量都是 5 个。

例如，创建三个主分片，两个副本分片的索引。

请求：PUT http://127.0.0.1:9200/secisland/

参数：

```
{
 "settings" : {
    "index" : {
       "number_of_shards" : 3,
       "number_of_replicas" : 2
    }
 }
}
```

参数可以简写成：

```
{
   "settings": {
      "number_of_shards": 3,
      "number_of_replicas": 2
   }
}
```

返回值：

```
{
      "acknowledged": true
}
```

然后我们直观的看一下创建后的效果。

图 2.1　　修改分片数量示意图

后面可以通过 update-index-settings API 完成对副本分片数量的修改。例如

请求：PUT http://127.0.0.1:9200/secisland/_settings/

参数：

```
{
  "number_of_replicas": 1
}
```

这样就把副本分片的数量改成 1 个副本。

对于任何 Elasticsearch 文档而言，一个文档会包括一个或者多个字段，任何字段都要有自己的数据类型。例如是 string，integer，date 等。Elasticsearch 中是通过映射来进行字段和数据类型对应的。在默认的情况下 Elasticsearch 会自动识别字段的数据类型。同时 Elasticsearch 提供了 mappings 参数可以显式地进行映射。

我们看一下在 Lucene 中是如何看待文档的。在 Lucene 中的文档包含的是一个简单 field-value 对的列表。一个字段至少要有一个值，但是任何字段都可以拥有多个值。类似的，一个字符串值也可以通过解析阶段而被转换为多个值。Lucene 不管值是字符串类型，还是数值类型或者其它类型，所有的值都会被同等地看做一些不透明的字节(Opaque bytes)。当我们使用 Lucene 对文档进行索引时，每个字段的值都会被添加到倒排索引(Inverted Index)的对应字段中。也可以选择原始值是否会不作修改的被保存到索引中，以此来方便将来的获取。

创建自定义字段类型的例子如下：

请求：PUT http://127.0.0.1:9200/secisland

参数：

```
{
  "settings": {
    "number_of_shards": 3,
    "number_of_replicas": 2
  },
  "mappings": {
    "secilog": {
      "properties": {
        "logType": {
```

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

在这个例子中，我们创建了一个名为 secilog 的类型，类型中有一个字段，字段的名称是 logType，字段的数据类型是 string，而且这个字段是不进行分析的。

2.1.2 删除索引

请求：DELETE http://127.0.0.1:9200/secisland/

上面的示例删除了名为 secisland 的索引。删除索引需要指定索引名称，别名或者通配符。

删除索引可以使用逗号分隔符，或者使用_all 或*号删除全部索引。

注意：_all 或 * 删除全部索引时要谨慎操作。

为了防止误删除，可以设置 elasticsearch.yml 属性 action.destructive_requires_name 为 true，禁止使用通配符或_all 删除索引，必须使用名称或别名才能删除该索引。

2.1.3 获取索引

获取索引接口允许从一个或多个索引中获取信息。

请求：GET http://127.0.0.1:9200/secisland/

通过这个请求会把系统中的信息都显示出来，包括默认的一些配置，例如上面的返回值为：

```
{
    "secisland": {
        "aliases": { },
        "mappings": {
            "secilog": {
                "properties": {
                    "logType": {
                        "type": "string",
                        "index": "not_analyzed"
                    }
                }
            }
        },
        "settings": {
            "index": {
                "creation_date": "1459652280258",
                "uuid": "2EsTzGu-QuCh7ddPGZ9gUA",
                "number_of_replicas": "2",
                "number_of_shards": "3",
                "version": {
                    "created": "2020099"
                }
```

```
            }
        },
        "warmers": { }
    }
}
```


上面的示例获取名为 secisland 的索引。获取索引需要指定索引名称，别名或者通配符。获取索引可以使用通配符获取多个索引，或者使用_all 或*号获取全部索引。

返回结果过滤：

可以自定义返回结果的属性。

请求：GET　http://localhost:9200/secisland/_settings,_mappings

上面示例只返回 secisland 索引的 settings 和 mappings 属性。

可配置的属性包括_settings, _mappings, _warmers and _aliases。

如果索引不存在，系统会返回一个错误内容，例如：

请求：GET　http://127.0.0.1:9200/secilog


```
{
    "error": {
        "root_cause": [
            {
                "type": "index_not_found_exception",
                "reason": "no such index",
                "index": "secilog",
                "resource.type": "index_or_alias",
                "resource.id": "secilog"
            }
        ],
        "type": "index_not_found_exception",
        "reason": "no such index",
        "index": "secilog",
        "resource.type": "index_or_alias",
        "resource.id": "secilog"
    },
    "status": 404
}
```


2.1.4　打开/关闭索引

打开/关闭索引接口允许关闭一个打开的索引或者打开一个已经关闭的索引。关闭的索引只能显示索引元数据信息，不能够进行读写操作。

打开/关闭索引的方式是/{索引名}/_close 或者/{索引名}/_open，完整示例如下：

请求：POST 127.0.0.1:9200/secisland/_close

图 2.2　　关闭索引示意图

请求：POST 127.0.0.1:9200/secisland/_open

可以同时打开或关闭多个索引。如果指向不存在的索引会抛出错误。可以使用配置 ignore_unavailable=true，不显示异常。

全部索引可以使用_all 打开或关闭，或者使用通配符表示全部（比如 *）

设置 config/elasticsearch.yml 属性 action.destructive_requires_name 为 true，禁止使用通配符或者_all 标识索引。

因为关闭的索引会继续占用磁盘空间而不能使用，所以关闭索引接口可能造成磁盘空间的浪费。

禁止使用关闭索引功能，可以设置 settingscluster.indices.close.enable 为 false，默认是 true。

2.2 索引映射管理

在前面例子中 Elasticsearch 创建文档的时候，是没有指定索引参数，这个时候系统会自动判断每个维度的类型，有很多时候需要我们进行一些更高级的设置，比如索引分词，是否存储等。

2.2.1　　增加映射

API 允许你向索引(index)添加文档类型(type)，或者向文档类型(type)中添加字段(field)。

```
PUT secisland
{
  "mappings": {
    "log": {
      "properties": {
        "message": {
          "type": "string"
        }
      }
    }
  }
}
```

以上接口添加索引名为 secisland，文档类型为 log，其中包含字段 message，字段类型是字符串。

```
PUT secisland/_mapping/user
{
  "properties": {
```

```
      "name": {
        "type": "string"
      }
    }
  }
```

向已经存在的索引 secisland 添加文档类型为 user，包含字段 name，字段类型是字符串。

```
PUT secisland/_mapping/log
{
  "properties": {
    "user_name": {
      "type": "string"
    }
  }
}
```

已经存在的索引 secisland，文档类型为 log，添加新的字段 user_name，字段类型是字符串。

- 多个索引设置映射

可以一次向多个索引添加文档类型。

PUT /{index}/_mapping/{type}

{ body }

{index}可以有多种方式，逗号分隔：比如 test1,test2,test3。_all 表示所有索引。通配符*表示所有。test*表示以 test 开头。

{type}需要添加或更新的文档类型。

{body}需要添加的字段或字段类型。

- 更新字段映射

在一般情况下，对现有字段的映射不会更新。对这个规则有一些例外。例如：

新的属性被添加到对象数据类型的字段。

新的多域字段被添加到现有的字段。

doc_values 可以被禁用。

增加了 ignore_above 参数。

例如：

请求：PUT secisland

参数:

```
{
  "mappings": {
    "user": {
      "properties": {
        "name": {
          "properties": {
            "first": {
              "type": "string"
```

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

user 的第一个 name 属性是对象数据类型(Object datatype)字段，对上个索引进行修改：

请求：PUT secisland/_mapping/user

参数：

```
{
  "properties": {
    "name": {
      "properties": {
        "last": {
          "type": "string"
        }
      }
    },
    "user_id": {
      "type": "string",
      "index": "not_analyzed",
      "ignore_above": 100
    }
  }
}
```

修改映射，对第一个对象数据类型增加了一个属性是 last。修改了 user_id, 通过设置 ignore_above 使默认的更新为 100。

- 不同类型之间的冲突

在同一个索引的不同类型(type)中，相同名称的字段中必须有相同的映射，因为他们内部是在同一个领域内，如果试图在这种情况下更新映射参数，系统将会抛出异常。除非在更新的时候指定 update_all_types 参数。在这种情况下它将更新所有同一索引同名称的映射参数。

例如：

请求：PUT secisland

参数：

```
{
  "mappings": {
    "type_one": {
```

```
        "properties": {
          "text": {
            "type": "string",
            "analyzer": "standard"
          }
        }
      },
      "type_two": {
        "properties": {
          "text": {
            "type": "string",
            "analyzer": "standard"
          }
        }
      }
    }
}
```

修改映射

请求：PUT secisland/_mapping/type_one

参数：

```
{
  "properties": {
    "text": {
      "type": "string",
      "analyzer": "standard",
      "search_analyzer": "whitespace"
    }
  }
}
```

这个时候会抛出异常，然后增加参数，update_all_types，这个时候会同时更新两个类型。

请求：PUT secisland/_mapping/type_one?update_all_types

```
{
  "properties": {
    "text": {
      "type": "string",
      "analyzer": "standard",
      "search_analyzer": "whitespace"
    }
  }
}
```

2.2.2 获取映射

获取文档映射接口允许通过索引或者索引和类型来检索。

GET http://localhost:9200/secisland/_mapping/tweet

系统同时支持获取多个索引和类型的语法：

获取文档映射接口一次可以获取多个索引或文档映射类型。该接口通常是如下格式：

host:port/{index}/_mapping/{type}，{index}和{type}可以接受逗号(,)分隔符，也可以使用_all 来表示全部索引。如下所示：

GET http://localhost:9200/_mapping/secisland,kimchy

GET http://localhost:9200/_all/_mapping/tweet,book

第一个省略_all，第二个使用_all，两者都是表示全部索引。也就是说，下面两个是等价的：

GET http://localhost:9200/_all/_mapping

GET http://localhost:9200/_mapping

2.2.3 获取字段映射

获取文档字段接口允许你检索一个或多个字段。这个用来检索想要检索的字段，而不是某个索引或者文档类型的全部内容。

这段请求只返回字段为 text 的内容：

GET http://localhost:9200/secisland/_mapping/tweet/field/text

响应结果如下（假定 text 为 String 类型）

```
{
    "secisland": {
        "tweet": {
            "text": {
                "full_name": "text",
                "mapping": {
                    "text": { "type": "string" }
                }
            }
        }
    }
}
```

获取多索引和类型的字段映射。

获取文档字段映射接口一次可以获取多个索引或文档映射类型。该接口通常是如下格式：host:port/{index}/{type}/_mapping/field/{field}

{index}，{type}，{field}可以使用逗号(,)分隔，也可以使用通配符。

其中{index}可以使用_all 表示全部索引，示例如下：

GET http://localhost:9200/secisland,kimchy/_mapping/field/message

GET http://localhost:9200/_all/_mapping/tweet,book/field/message,user.id

GET http://localhost:9200/_all/_mapping/tw*/field/*.id

指定字段

获取文档字段接口，可以使用逗号(,)分隔符或者通配符(*)。

如下文档示例，如果只使用字段名 id 会产生歧义。

```
{
    "article": {
        "properties": {
```

```
                "id": { "type": "string" },
                "title":    { "type": "string"},
                "abstract": { "type": "string"},
                "author": {
                    "properties": {
                        "id": { "type": "string" },
                        "name": { "type": "string" }
                    }
                }
            }
        }
    }
```

如果想要表示 author 中的 id，name，使用 author.id，author.name。请求如下：

```
curl -XGET "http://localhost:9200/publications/_mapping/article/field/
author.id,abstract,author.name"
```

返回结果如下：

```
{
    "publications": {
        "article": {
            "abstract": {
                "full_name": "abstract",
                "mapping": {
                    "abstract": { "type": "string" }
                }
            },
            "author.id": {
                "full_name": "author.id",
                "mapping": {
                    "id": { "type": "string" }
                }
            },
            "author.name": {
                "full_name": "author.name",
                "mapping": {
                    "name": { "type": "string" }
                }
            }
        }
    }
}
```

2.2.4 判断类型是否存在

检查索引或文档类型是否存在

```
HEAD http://localhost:9200/secisland/tweet
```

存在返回 200，不存在返回 404。

2.3 索引别名

别名管理

在 Elasticsearch 所有的 API 中，对应的是一个或者多个索引。Elasticsearch 可以对一个或者多个索引指定别名，通过别名可以查询到一个或者多个索引的内容。在内部，Elasticsearch 会自动把别名映射到相应的索引上。可以对别名编写过滤器或者路由，在系统中别名不能重复，也不能和索引名重复。其实 Elasticsearch 的别名机制有点像数据库中的视图。例如：把索引 test1 增加一个别名 alias1。

请求：POST http://localhost:9200/_aliases

参数：

```
{
    "actions" : [
        { "add" : { "index" : "test1", "alias" : "alias1" } }
    ]
}
```

删除别名：

请求是一样的，参数不一样。

```
{
    "actions" : [
        { "remove" : { "index" : "test1", "alias" : "alias1" } }
    ]
}
```

注意：别名没有修改的语法，当需要修改别名的时候，可以先删除别名，然后再增加别名，例如：

```
{
    "actions" : [
        { "remove" : { "index" : "test1", "alias" : "alias1" } },
        { "add" : { "index" : "test1", "alias" : "alias2" } }
    ]
}
```

一个别名关联多个索引：

```
{
    "actions" : [
        { "add" : { "index" : "test1", "alias" : "alias1" } },
        { "add" : { "index" : "test2", "alias" : "alias1" } }
    ]
}
```

或者用下面的语法：

```
{
    "actions" : [
        { "add" : { "indices" : ["test1", "test2"], "alias" : "alias1" } }
    ]
}
```

或者使用通配符：

```
{
```

```
    "actions" : [
        { "add" : { "index" : "test*", "alias" : "all_test_indices" } }
    ]
}
```

注意：通配符指定的索引只是在当前生效，后面添加的索引不会被自动添加到别名上。

对某一别名做索引的时候，如果该别名关联多个索引会报错。

- 过滤索引别名

通过过滤索引来指定别名提供了对索引查看的不同视图，该过滤器可以使用查询 DSL 来定义适用于所有的搜索，计数，查询删除等，以及更多类似这样的与此别名的操作。

要创建一个过滤的别名，首先我们需要确保映射中已经存在的字段：

创建一个索引，请求：PUT http://localhost:9200/test1

参数

```
{
  "mappings": {
    "type1": {
      "properties": {
        "user" : {
          "type": "string",
          "index": "not_analyzed"
        }
      }
    }
  }
}
```

创建过滤别名，请求：POST http://localhost:9200/_aliases

参数：

```
{
    "actions" : [
        {
            "add" : {
                "index" : "test1",
                "alias" : "alias2",
                "filter" : { "term" : { "user" : "kimchy" } }
            }
        }
    ]
}
```

通过别名也可以和路由关联，此功能可以和过滤别名命令一起使用，以避免不必要的碎片操作。例如:

请求：POST 'http://localhost:9200/_aliases

参数：

```
{
    "actions" : [
```

```
            {
                "add" : {
                        "index" : "test",
                        "alias" : "alias1",
                        "routing" : "1"
                }
            }
        ]
    }
```

同时可以指定搜索路由或者查询路由，例如参数：

```
    {
        "actions" : [
            {
                "add" : {
                        "index" : "test",
                        "alias" : "alias2",
                        "search_routing" : "1,2",
                        "index_routing" : "2"
                }
            }
        ]
    }
```

注意：搜索路由可以指定多个值，索引路由只能指定一个值。如果使用路由别名操作的同时还有路由参数，则结果是别名路由和路由的交集。例如以下命令将使用“2”作为路由值：

GET http://localhost:9200/alias2/_search?q=user:kimchy&routing=2,3

通过参数添加别名

语法：PUT /{index}/_alias/{name}

条件解释：

index：参照的索引，可以使用 * ， _all ， 正则表达式或者逗号分开的多个 name1, name2, …

name：别名的名称。

routing：别名对应的路由。

filter：指定别名时候的过滤条件。

例如：PUT localhost:9200/logs_201305/_alias/2013

例如一个索引。

PUT localhost:9200/users

```
    {
        "mappings" : {
            "user" : {
                "properties" : {
                    "user_id" : {"type" : "integer"}
                }
```

```
            }
        }
    }
```

指定别名：

```
PUT localhost:9200/users/_alias/user_12
{
    "routing" : "12",
    "filter" : {
        "term" : {
            "user_id" : 12
        }
    }
}
```

建索引的时候同时指定别名

例如：

```
PUT localhost:9200/logs_20142801
{
    "mappings" : {
        "type" : {
            "properties" : {
                "year" : {"type" : "integer"}
            }
        }
    },
    "aliases" : {
        "current_day" : {},
        "2014" : {
            "filter" : {
                "term" : {"year" : 2014 }
            }
        }
    }
}
```

- 删除别名

语法：DELETE /{index}/_alias/{name}

例如：DELETE localhost:9200/users/_alias/user_12

- 查询现有的别名

可以通过索引名或者别名进行查询。参数：

index：索引别名的名称。部分名称支持通配符，用逗号分隔也可以指定多个索引名称，还可以使用索引的别名名称。

alias：在响应中返回的别名名称。 该参数支持通配符和用逗号分隔的多个别名。

ignore_unavailable：如果一个指定的索引名称不存在该怎么办。如果设置为 true，那么这些索引将被忽略。

语法：GET /{index}/_alias/{alias}

例如：GET localhost:9200/users/_alias/*

返回值：


```
{
  "users" : {
    "aliases" : {
      "user_13" : {
        "filter" : {
          "term" : {
            "user_id" : 13
          }
        },
        "index_routing" : "13",
        "search_routing" : "13"
      },
      "user_14" : {
        "filter" : {
          "term" : {
            "user_id" : 14
          }
        },
        "index_routing" : "14",
        "search_routing" : "14"
      }
    }
  }
}
```


下面的例子中包括所有别名为 2013 的。

GET localhost:9200/_alias/2013

返回：


```
{
  "logs_201304" : {
    "aliases" : {
      "2013" : { }
    }
  },
  "logs_201305" : {
    "aliases" : {
      "2013" : { }
    }
  }
}
```


GET localhost:9200/_alias/2013_01*

返回：


```
{
```

```
  "logs_20130101" : {
    "aliases" : {
      "2013_01" : { }
    }
  }
}
```

用 HEAD 也可以检查别名是否存在，语法和 GET 类似，例如：

HEAD localhost:9200/_alias/2013

HEAD localhost:9200/_alias/2013_01*

HEAD localhost:9200/users/_alias/*

2.4 索引设置

2.4.1 更新索引设置

在 REST 风格的 URL 设置中设置 /_settings (所有索引)或者{index}/_settings，可以设置一个或者多个索引，例如：

请求：PUT localhost:9200/secisland/_settings

参数：

```
{
    "index" : {
        "number_of_replicas" : 4
    }
}
```

更新分词器

创建索引后可以添加新的分析器。添加分析器之前必须关闭索引，添加之后再打开索引。

POST localhost:9200/myindex/_close

PUT localhost:9200/myindex/_settings

参数：

```
{
  "analysis" : {
    "analyzer":{
      "content":{
        "type":"custom",
        "tokenizer":"whitespace"
      }
    }
  }
}
```

POST localhost:9200/myindex/_open

如上示例，先关闭 myindex 索引，然后添加自定义分析器，分析器策略是空格分析器(whitespace)，就是按照空格进行分词。

2.4.2 获取设置

GET http://localhost:9200/secisland/_settings

获取索引配置参数的请求格式如下：

host:port/{index}/_settings

host：主机名

port：端口号

{index}：接收多种格式，* | _all | name1, name2, …

过滤配置参数结果

GET http://localhost:9200/secisland/_settings/name=index.number_*

name=index.number_*设置将只返回 number_of_replicas,number_of_shards 两个参数详情。

2.4.3 索引分析

索引分析(analysis)是这样一个过程：

首先，把一个文本块分析成一个个单独的词(term)，为了后面的倒排索引做准备；

然后标准化这些词为标准形式，提高它们的“可搜索性”；

这些工作是分析器(analyzers)完成的。一个分析器(analyzers)是一个组合，用于将三个功能放到一起：

字符过滤器：

字符串经过字符过滤器(character filter)处理，它们的工作是在标记化之前处理字符串。字符过滤器能够去除 HTML 标记，或者转换“&”为“and”。

分词器：

分词器(tokenizer)被标记化成独立的词。一个简单的分词器(tokenizer)可以根据空格或逗号将单词分开。

标记过滤器：

每个词都通过所有标记过滤(token filters)处理，它可以修改词（例如将"Quick"转为小写），去掉词（例如连接词像“a”、“and”、“the”等等），或者增加词（例如同义词像“jump”和“leap”）。

Elasticsearch 提供很多内置的字符过滤器，分词器和标记过滤器。这些可以组合起来创建自定义的分析器以应对不同的需求。

测试分析器：

```
GET localhost:9200/_analyze
{
  "analyzer" : "standard",
  "text" : "this is a test"
}
```

该结果将返回“this is a test”使用 standard 分析器后词的解析情况。

在该分析器下，将会分析成 this，is，a，test 四个词。

自定义分析器

```
GET localhost:9200/_analyze
{
  "tokenizer" : "keyword",
  "token_filters" : ["lowercase"],
  "char_filters" : ["html_strip"],
  "text" : "this is a <b>test</b>"
}'
```

使用 keyword 分词器，lowercase 分词过滤，字符过滤器是 html_strip，这 3 部分构成一个分词器。

上面示例返回分词结果是 this is a test，其中 html_strip 过滤掉了 html 字符。

也可以指定索引进行分词。URL 格式如下：

localhost:9200/test/_analyze

索引详情

如果想获取分析器分析的更多细节，设置 explain 属性为 true(默认是 false)，将输出分词器的分词详情。请求格式如下：

请求：GET test/_analyze

参数：

```
{
   "tokenizer" : "standard",
   "token_filters" : ["snowball"],
   "text" : "detailed output",
   "explain" : true,
   "attributes" : ["keyword"]
}
```

返回结果如下：

```
{
   "detail" : {
      "custom_analyzer" : true,
      "charfilters" : [ ],
      "tokenizer" : {
         "name" : "standard",
         "tokens" : [ {
            "token" : "detailed",
            "start_offset" : 0,
            "end_offset" : 8,
            "type" : "<ALPHANUM>",
            "position" : 0
         }, {
            "token" : "output",
            "start_offset" : 9,
            "end_offset" : 15,
            "type" : "<ALPHANUM>",
            "position" : 1
         } ]
      },
      "tokenfilters" : [ {
         "name" : "snowball",
         "tokens" : [ {
            "token" : "detail",
            "start_offset" : 0,
            "end_offset" : 8,
            "type" : "<ALPHANUM>",
            "position" : 0,
            "keyword" : false
```

```
        }, {
          "token" : "output",
          "start_offset" : 9,
          "end_offset" : 15,
          "type" : "<ALPHANUM>",
          "position" : 1,
          "keyword" : false
        } ]
      } ]
    }
  }
```

2.4.4 索引模板

创建索引模板

索引模板就是创建好一个索引参数设置(settings)和映射（mapping）的模板，在创建新索引的时候指定模板名称就可以使用模板定义好的参数设置和映射。例子如下：

请求：PUT localhost:9200/_template/template_1

```
{
    "template" : "te*",
    "settings" : {
        "number_of_shards" : 1
    },
    "mappings" : {
        "type1" : {
            "_source" : { "enabled" : false }
        }
    }
}
```

定义好模板可使用 te*来适配，分片数量为 1，默认文档类型 type1，_source 的 enabled 为 false。

也可以在模板中定义别名等其它属性。

删除索引模板

DELETE localhost:9200/_template/template_1

template_1 为之前创建的索引模板名称。

获取索引模板

GET localhost:9200/_template/template_1

使用通配符或逗号分隔符

GET localhost:9200/_template/temp*

GET localhost:9200/_template/template_1,template_2

获取所有索引模板

GET localhost:9200/_template/

判断索引模板是否存在

HEAD -i localhost:9200/_template/template_1

多个模板匹配

有这样一种情况：template_1，template_2 两个模板，使用 te*会匹配 2 个模板，最后合并两个模板的配置。如果配置重复，这时应该设置 order 属性，order 是从 0 开始的数字，先匹配 order 数字小的，再匹配数字大的，如果有相同的属性配置，后匹配的会覆盖之前的配置。

2.4.5 复制配置

警告：这个功能是实验性的，可能在未来的版本中移除。

如果使用共享文件系统，可以使用复制配置来选择索引数据保存在磁盘上的位置，以及 Elasticsearch 应该如何在索引副本分片上重复操作。

为了充分利用 index.data_path 和 index.shadow_replicas 设置，要让 Elasticsearch 使用相同数据目录为多个实例服务，需要在 elasticsearch.yml 中设置 node.add_id_to_custom_path 为 false。

node.add_id_to_custom_path: false

可以在 elasticsearch.yml 中设置 path.shared_data 权限管理标明自定义索引的位置，以便应用正确的权限。

path.shared_data: /opt/data

这意味着 Elasticsearch 可以读写 path.shared_data 设置中所有子目录中的文档。

可以在自定义数据路径中创建索引，每个节点使用这个路径来存储数据：

```
PUT 'localhost:9200/secisland
{
    "index" : {
        "number_of_shards" : 1,
        "number_of_replicas" : 4,
        "data_path": "/opt/data/secisland",
        "shadow_replicas": true
    }
}'
```

索引创建时，index.shadow_replicas 设置为“true”不会在任何副本分片上重复文档操作，反而，会不停的刷新。

下面是可以使用设置更新接口修改的设置列表：

index.data_path(String)	索引数据使用的路径。Elasticsearch 默认附加节点序数到路径中，确保相同机器上的多重实例不会共享数据目录。
index.shadow_replicas	布尔值，指示索引是否应该使用副本。默认为 false。
index.shared_filesystem	布尔值，指示索引使用共享文件系统。如果 index.shadow_replicas 设置为 true，默认值为 true，其他情况下的默认值为 false。
index.shared_filesystem.recover_on_any_node	布尔值，指示索引的主分片是否可以恢复到集群中的任何节点。默认值为 false。

表格 2.1 设置更新接口参数

2.4.6 重建索引

重建索引是 2.3.0 新增加的接口。这个接口是实验性质的，在未来有可能会改变。

重建索引的最基本的功能是拷贝文件从一个索引到另一个索引，例如：

```
POST /_reindex
{
```

```
  "source": {
    "index": "secisland"
  },
  "dest": {
    "index": "new_secisland"
  }
}
```

返回的内容如下：

```
{
  "took" : 639,
  "updated": 112,
  "batches": 130,
  "version_conflicts": 0,
  "failures" : [ ],
  "created": 12344
}
```

took：从开始到结束的整个操作的毫秒数。

updated：已成功更新的文档数。

created：成功创建的文档数。

batches：从重建索引拉回的滚动响应的数量。

version_conflicts：重建索引中版本冲突数的数量。

failures：所有索引失败的数组。如果这是非空的，则请求将被中止。

由于_reindex 是获取源索引的快照，而且目标索引是不同的索引，所以基本上不太可能产生冲突。在接口参数中可以增加 dest 来进行乐观并发控制。如果 version_type 设置为 internal 会导致 Elasticsearch 盲目转储文件到目标索引，任何具有相同类型和 ID 的文档将被重写。例如：

```
POST /_reindex
{
  "source": {
    "index": "twitter"
  },
  "dest": {
    "index": "new_twitter",
    "version_type": "internal"
  }
}
```

如果设置 version_type 为 external 将会导致 Elasticsearch 保护源索引的版本，如果在目标索引中有一个比源索引旧的版本，则会更新文档。对于源文件中丢失的文档在目标中也会被创建。

```
POST /_reindex
{
  "source": {
    "index": "twitter"
```

```
    },
    "dest": {
      "index": "new_twitter",
      "version_type": "external"
    }
  }
```

设置 op_type 为 create 将导致_reindex 在目标索引中仅创建丢失的文件。所有现有的文件将导致版本冲突。

```
POST /_reindex
{
  "source": {
    "index": "twitter"
  },
  "dest": {
    "index": "new_twitter",
    "op_type": "create"
  }
}
```

正常情况下当发生冲突的时候_reindex 过程将被终止，可以在请求体中设置 "conflicts": "proceed" ，可以只进行计算。

```
POST /_reindex
{
  "conflicts": "proceed",
  "source": {
    "index": "twitter"
  },
  "dest": {
    "index": "new_twitter",
    "op_type": "create"
  }
}
```

可以通过向源添加一个类型或者增加一个查询来限制文档的数量，比如只复制用户名为 kimchy 的文档。

```
POST /_reindex
{
  "source": {
    "index": "twitter",
    "type": "tweet",
    "query": {
      "term": {
        "user": "kimchy"
      }
    }
  },
```

```
  "dest": {
    "index": "new_twitter"
  }
}
```

在请求接口中可以列出源索引和类型，可以在一个接口中复制多个源。例如下面的例子将在 twitter 和 blog 索引中的 tweet 和 post 类型中拷贝数据，这包括 twitter 索引中的“tweet”和“post”类型，也包括 blog 索引中的“tweet”和“post”类型。如果需要更具体的文档可以使用查询。当 id 产生冲突的时候是没有办法处理的，因为执行的顺序是随机的，所以目标索引将无法确定应该保存哪些文档。

```
POST /_reindex
{
  "source": {
    "index": ["twitter", "blog"],
    "type": ["tweet", "post"]
  },
  "dest": {
    "index": "all_together"
  }
}
```

也可以通过设置大小来限制处理文档的数量。这只会复制一个文件到 new_twitter 索引中。

```
POST /_reindex
{
  "size": 1,
  "source": {
    "index": "twitter"
  },
  "dest": {
    "index": "new_twitter"
  }
}
```

如果你想要复制特定的文档，可以使用排序。排序会降低效率，但在某些情况下，它是有意义的。如果可能的话，可以选择性的查询来确定复制的大小和排序。下面将从 twitter 索引中复制 10000 文档到 new_twitter 中。

```
POST /_reindex
{
  "size": 10000,
  "source": {
    "index": "twitter",
    "sort": { "date": "desc" }
  },
  "dest": {
    "index": "new_twitter"
  }
```

```
}
```

_reindex 同时支持使用脚本来修改文档。例如：

```
POST /_reindex
{
  "source": {
    "index": "twitter",
  },
  "dest": {
    "index": "new_twitter",
    "version_type": "external"
  }
  "script": {
    "internal": "if (ctx._source.foo == 'bar') {ctx._version++; ctx._source.remove('foo')}"
  }
}
```

可以修改的元字段包括_id、_type、_index、_version、_routing、_parent、_timestamp、_ttl。所以用脚本复制能力非常强大。

默认情况下，如果_reindex 可以设置文档路由来进行路由保存，除非是通过脚本进行了改变。你可以设置 dest 参数来进行路由设置：

keep：设置在匹配的每一个路由上发送请求的路由。这个是默认设置。

discard：为每一个匹配发送的请求设置为空。

=<some text>：设置为每一个匹配的文本请求发送路由上的路由。例如，你可以使用以下要求复制所有文件从源索引为“company”等于“cat”的查询中复制到目标索引，目标索引的路由设置为 cat。

```
POST /_reindex
{
  "source": {
    "index": "source"
    "query": {
      "match": {
        "company": "cat"
      }
    }
  }
  "dest": {
    "index": "dest",
    "routing": "=cat"
  }
}
```

默认情况下_reindex 每次处理的大小为 100。可以在源参数中用 size 参数来进行修改：

```
POST /_reindex
{
  "source": {
```

```
    "index": "source",
    "size": 1500
  },
  "dest": {
    "index": "dest"
  }
}
```

URL 参数：

_reindex 接口除了接收标准的参数例如 pretty，还支持 refresh，wait_for_completion，consistency，timeout 参数。

在 URL 参数中发送 refresh 会导致所有写入请求的索引被刷新。与索引接口中的 refresh 参数不同，只有接收到新数据的分片会进行刷新。

如果请求包含 wait_for_completion = false，Elasticsearch 将进行执行前检查后启动请求，然后返回一个 task 可用于任务 API 取消或得到任务的状态，现在一旦请求完成任务就不见了，找任务的最终结果的唯一地方是在 Elasticsearch 日志文件中，这个问题将会在未来的版本中修复。

consistency 控制每次写请求必须多少份分片被响应。

timeout 控制每个写请求等待可用的分片的时间。

任务查看：

GET /_tasks/?pretty&detailed=true&actions=*reindex

返回的结果类似

```
{
  "nodes" : {
    "r1A2WoRbTwKZ516z6NEs5A" : {
      "name" : "Tyrannus",
      "transport_address" : "127.0.0.1:9300",
      "host" : "127.0.0.1",
      "ip" : "127.0.0.1:9300",
      "attributes" : {
        "testattr" : "test",
        "portsfile" : "true"
      },
      "tasks" : {
        "r1A2WoRbTwKZ516z6NEs5A:36619" : {
          "node" : "r1A2WoRbTwKZ516z6NEs5A",
          "id" : 36619,
          "type" : "transport",
          "action" : "indices:data/write/reindex",
          "status" : {
            "total" : 6154,
            "updated" : 3500,
            "created" : 0,
            "deleted" : 0,
            "batches" : 36,
```

```
                "version_conflicts" : 0,
                "noops" : 0
              },
              "description" : ""
            }
          }
        }
      }
    }
```

_reindex 可以用来建立索引的时候同时也可以修改列的名称，例如，源索引类型为：

```
POST test/test/1?refresh&pretty
{
  "text": "words words",
  "flag": "foo"
}
```

可以通过_reindex 来修改列的名称，例如：

```
POST _reindex?pretty
{
  "source": {
    "index": "test"
  },
  "dest": {
    "index": "test2"
  },
  "script": {
    "inline": "ctx._source.tag = ctx._source.remove(\"flag\")"
  }
}
```

然后我们看一下 test2 的结构，可以看出 flag 字段名称修改成了 tag。

```
GET test2/test/1?pretty
{
  "text": "words words",
  "tag": "foo"
}
```

2.5 索引监控

2.5.1 索引统计

索引统计接口提供索引中不同内容的统计数据（其中的大多数统计数据也可以从节点级别范围取得）。

获取所有聚合以及索引的统计数据：

- GET localhost:9200/_stats

获取指定索引的统计数据:

- GET localhost:9200/index1,index2/_stats

默认情况返回所有统计数据，也可以在 URL 中指定需要返回的特定统计数据。可以是以下任何一项：

docs	文档/删除文档（没有合并的文档）的数量。注意，受索引刷新的影响。
store	索引的大小。
indexing	索引统计数据，可以结合用逗号分隔的类型列表来提供文档类型级别的统计数据。
get	获取统计数据，包含缺失统计。
search	搜索统计数据。可以通过添加额外的 groups 参数（搜索操作可以关联一个或多个分组）包含自定义分组的统计数据。groups 参数接受逗号分隔的组名列表。使用_all 来返回所有分组的统计数据。
completion	完成建议统计数据。
fielddata	字段数据统计数据。
flush	冲刷统计数据。
merge	混合统计数据。
request_cache	分片请求缓存统计数据。
refresh	刷新统计数据。
suggest	建议统计数据。
translog	事务日志统计数据。

表格 2.2 统计数据选项

一些统计数据可以作用在字段粒度上，接受逗号分隔的字段列表。默认包含所有字段：

fields	包含在统计数据中的字段列表。用作默认列表，除非提供了更明确的列表。
completion_fields	包含在完成建议统计数据中的字段列表。
fielddata_fields	包含在字段数据统计数据中的字段列表。

表格 2.3 字段参数选项

举一些例子：

- 获取所有索引的混合和刷新统计数据

GET localhost:9200/_stats/merge,refresh

- 获取名为 secisland 索引中类型为 type1 和 type2 的文档统计数据

GET localhost:9200/secisland/_stats/indexing?types=type1,type2

- 获取分组 group1 和 group2 的搜索统计数据

GET localhost:9200/_stats/search?groups=group1,group2

返回的统计数据在索引级别发生聚合，生成名为 primaries 和 total 的聚合。其中，primaries 仅包含主分片的值，total 是主分片和从分片的累计值。

为了获取分片级别统计数据，需要设置 level 参数为 shards。

注意，当分片在集群中移动的时候，它们的统计数据会被清除，视作它们在其他节点中被创建。另一方面，即使分片“离开”了一个节点，那个节点仍然会保存分片之前的统计数据。

2.5.2 索引碎片

提供 Lucene 索引所在的分片信息。可以用来提供分片和索引的更多统计信息，可能是优化信息，删除的“垃圾”数据等等。

端点包括特定的索引，多个索引或者所有索引的分片：

- GET http://localhost:9200/test/_segments
- GET http://localhost:9200/test1,test2/_segments
- GET http://localhost:9200/_segments

响应：

```
{
    ...
        "_3": {
            "generation": 3,
            "num_docs": 1121,
            "deleted_docs": 53,
            "size_in_bytes": 228288,
            "memory_in_bytes": 3211,
            "committed": true,
            "search": true,
            "version": "4.6",
            "compound": true
        }
    ...
}
```

_3	JSON 文档的键名，代表分片的名称。这个名用来生成文档名：分片目录中所有以分片名开头的文档属于这个分片。
generation	需要写新的分片时生成的一个数字，基本上是递增的。分片名从这个生成的数字派生出来。
num_docs	存储在分片中没被删除的文档数量。
deleted_docs	存储在分片中被删除的文档数量。如果这个数字大于 0 也是没有问题的，磁盘空间会在分片融合的时候被回收。
size_in_bytes	用字节表示分片使用的磁盘空间数量。
memory_in_bytes	分片需要在内存中存储一些数据来使搜索性能更加高效。这个数字表示用于这个目的的字节数量。如果返回的值为-1，表示 Elasticsearch 无法计算这个值。
committed	表示分片在磁盘上是否同步。提交的分片会在硬重启中存活下来。如果值为 false 也不需要担心，未提交的分片数据也会存储在事务日志中，在 Elasticsearch 下一次启动时可以重做修改。
search	分片是否可以进行搜索。如果值为 false，可能意味着分片已经被写入磁盘但是没有经过刷新使之可以进行搜索。
version	用来写这个分片的 Lucene 版本。
compound	分片是否存储在符合文件中。如果值为 true，意味着 Lucene 将分片中的所有文档融合为一个用来保存文档的描述符。

表格 2.4　索引分片信息

2.5.3　索引恢复

索引恢复接口提供正在进行恢复的索引分片信息。可以报告指定索引或者集群范围的恢复状态。

例如，获取 index1 和 index2 两个索引的恢复信息：

```
GET http://localhost:9200/index1,index2/_recovery
```

去掉索引名可以查看集群范围的恢复状态：

GET http://localhost:9200/_recovery?pretty&human

选项列表：

detailed　　显示详细的视图。主要用来查看物理索引文件的恢复。默认为 false。

active_only　显示那些现在正在进行的恢复。默认为 false。

输出字段描述：

id	分片 ID。
type	恢复类型：包括存储、快照、复制、迁移。
stage	恢复阶段：包括初始化（恢复没有开始）、索引（读取索引元字段并且从源到目的地复制字节）、开始（启动恢复；开启使用的索引）、事务日志（重做事务日志）、完成（清理）、结束。
primary	如果分片是主分片，值为 true；分片是从分片，值为 false。
start_time	恢复开始的时间戳。
stop_time	恢复结束的时间戳。
total_time_in_millis	以毫秒来表示恢复分片的整个时间。
source	恢复源：如果从快照中恢复，描述备份仓库；其他情况，描述源节点。
target	目标节点。
index	物理索引恢复的统计数据。
translog	事务日志恢复的统计数据。
start	打开和启动索引的时间统计数据。

表格 2.5　索引恢复输出字段

2.5.4　索引分片存储

提供索引分片副本的存储信息。存储信息报告分片副本存在的节点、分片副本版本、指示分片副本最近的状态以及在开启分片索引时遭遇的任何异常。

默认的，只列出至少有一项未分配副本的分片的存储信息。

端点包括特定的索引，多个索引或者所有索引的分片：

- GET http://localhost:9200/test/_shard_stores
- GET http://localhost:9200/test1,test2/_shard_stores
- GET http://localhost:9200/_shard_stores

列出存储信息的分片范围可以通过 status 参数进行修改。默认是 yellow 和 red。使用 green 参数来列出所有指定副本分片的存储信息:

GET http://localhost:9200/_shard_stores?status=green

2.6 状态管理

2.6.1　清除缓存

清除缓存接口可以清除所有缓存或者关联一个或更多索引的特定缓存。

POST http://localhost:9200/secisland/_cache/clear

接口默认清理所有缓存，可以明确设置 query、fielddata 和 request 来清理特定缓存。

所有关联特定字段的缓存也可以被清理，通过逗号分隔的相关字段列表来指定 fields 参数。

2.6.2　索引刷新

刷新接口可以明确地刷新一个或多个索引，使之前最后一次刷新之后的所有操作被执行。（接近）实时能力取决于使用的搜索引擎。例如，内部的一个请求刷新被调用，但是默认刷新是一个周期性的安排。

POST http://localhost:9200/secisland/_refresh

刷新接口可以通过一条请求应用在多个索引上，或者所有索引：

POST http://localhost:9200/kimchy,elasticsearch/_refresh

POST http://localhost:9200/_refresh

2.6.3 Flush

冲洗接口可以通过接口冲洗一个或多个索引。索引主要通过执行冲洗将数据保存到索引存储并且清除内部事务日志，以此来释放索引的内存空间。默认的，Elasticsearch 使用内存启发式算法来自动触发冲洗操作的请求来清理内存。

POST /secisland/_flush

2.6.4 合并索引

合并接口可以强制合并一个或更多索引。合并分片数量和每个分片保存的 Lucene 索引。强制合并可以通过合并来减少分片数量。

调用会被阻塞直到合并完成。如果 http 连接丢失，请求会在后台继续执行，任何新的请求都会被阻塞直到前一个强制合并完成。

POST http://localhost:9200/secisland/_forcemerge

合并接口接受下面请求参数：

max_num_segments	用于合并的分片数量。为了充分合并索引，设置它的值为 1。默认简单地检查是否需要执行合并，如果是，执行合并。
only_expunge_deletes	合并过程是否只删除分片中被删除的文档。在 Lucene 中，文档不会从分片中删除，只是标记为删除。通过执行分片合并，一个不包含这些被删除的文档的新分片会被创建。这个标识可以只合并拥有删除文档的分片。默认值为 false。注意 index.merge.policy.expunge_deletes_allowed 阈值不会被覆盖。
flush	强制合并之后是否执行冲洗。默认为 true。

表格 2.6 合并索引接口参数

合并接口可以通过单次应用到多个索引，或者所有索引：

POST http://localhost:9200/kimchy,elasticsearch/_forcemerge

POST http://localhost:9200/_forcemerge

2.7 文档管理

文档是具体的数据，一个文档有点像数据库中的一条记录，文档必须包含在一个索引中。

2.7.1 增加文档

新增文档是把一条新的文档增加到索引中，使之能够进行搜索，文档的格式是 JSON 格式。 注意：在 Elasticsearch 中如果有相同 ID 的文档存在，则更新此文档。例如我们在索引 secilog 中下面增加一条文档：

请求：PUT http://localhost:9200/secilog/log/1

参数：

```
{
    "collect_type" : "syslog",
    "collect_date" : "2016-01-11T09:32:12",
    "message" : "Failed password for root from 192.168.21.1 port 50790 ssh2"
}
```

返回结果：

```
{
    "_index": "secilog",
    "_type": "log",
```

```
    "_id": "1",
    "_version": 1,
    "_shards": {
        "total": 2,
        "successful": 1,
        "failed": 0
    },
    "created": true
}
```

返回结果中的_shards 说明，_shards 提供了索引创建的过程信息:

total	文档被创建的时候，在多少个分片中进行了操作，包括主分片和副本分片。
successful	成功建立索引分片的数量，当创建成功后，成功创建索引分片的数量最少是 1。
failed	失败建立索引分片的数量。

表格 2.7 新增文档返回结果

自动创建索引

当创建文档的时候，如果索引不存在，则会自动创建该索引。自动创建的索引会自动映射每个字段的类型。自动创建字段类型是非常灵活的，新的字段类型将会自动匹配字段对象的类型。比如字符串类型,日期类型。可以通过配置文件设置 action.auto_create_index 为 false 在所有节点的配置文件中禁用自动创建索引。自动映射的字段类型可以通过配置文件设置 index.mapper.dynamic 为 false 禁用。自动创建索引可以通过模板设置索引名称，例如：可以设置 action.auto_create_index 为 +aaa*,-bbb*,+ccc*,-* (+表示准许，-表示禁止)。

版本号

每个文档都有一个版本号，版本号的具体值放在创建索引的返回值中("_version":)。通过版本号参数可以达到并发控制的效果。当在操作文档的过程中指定版本号，如果和版本号不一致的时候操作会被拒绝。版本号常用在对事务的处理中。例如，更新刚才创建的文档：

请求：PUT http://localhost:9200/secilog/log/1?version=2&pretty

参数：

```
{
    "message" : "elasticsearch now has versioning support!"
}
```

返回内容为更新失败：

```
{
  "error" : {
    "root_cause" : [ {
      "type" : "version_conflict_engine_exception",
      "reason" : "[log][1]: version conflict, current [-1], provided [2]",
      "index" : "secilog",
      "shard" : "2"
    } ],
    "type" : "version_conflict_engine_exception",
    "reason" : "[log][1]: version conflict, current [-1], provided [2]",
    "index" : "secilog",
    "shard" : "2"
  },
```

```
  "status" : 409
}
```

注意：版本号是实时更新的，不会存在缓存现象。当操作的时候不指定版本号，则系统不会对版本号是否一致进行检查。

默认情况下对文档的操作版本号从 1 开始递增，包括修改文档和删除文档。当然版本号还可以从外部获取，比如从数据库中获取，要启用此功能，version_type 应设置为 external，这个值必须是一个大于 0 小于 9.2e+18 的数字。当使用外部版本号来代替自动生成的版本号时，在操作文档的时候，系统通过对比参数中的版本号是否大于文档中的版本号来做判断，当参数中的版本号大于系统中的版本号，则执行此操作，并更新版本号。反之则拒绝操作(包括小于或者等于)。

版本号同时产生了一个比较实用的功能，只要版本号从源数据库中使用，在异步索引操作的时候就不需要对源数据库的变化执行严格排序。任何操作都只会对最新的版本号起作用，不管这个版本号是内部的还是从外部获取的。

操作类型

系统同时支持通过 op_type=create 参数强制命令执行创建操作，只有系统中不存在此文档的时候才会创建成功。如果不指定此操作类型，如果存在此文档，则会更新此文档。例如再次创建文档：

请求：PUT http://localhost:9200/secilog/log/1?op_type=create&pretty

参数：

```
{
    "collect_type" : "syslog",
    "collect_date" : "2016-01-11T09:32:12",
    "message" : "Failed password for root from 192.168.21.2 port 50790 ssh2"
}
```

返回值，表示创建失败：

```
{
  "error" : {
    "root_cause" : [ {
      "type" : "document_already_exists_exception",
      "reason" : "[log][1]: document already exists",
      "index" : "secilog",
      "shard" : "3"
    } ],
    "type" : "document_already_exists_exception",
    "reason" : "[log][1]: document already exists",
    "index" : "secilog",
    "shard" : "3"
  },
  "status" : 409
}
```

当不指定 op_type=create 时候，则更新此文档。

创建操作的另一个写法为：http://localhost:9200/secilog/log/1/_create?pretty

自动创建 ID

当创建文档的时候，如果不指定 ID，系统会自动创建 ID。自动生成的 ID 是一个不会重复的随机数。例如：

请求：POST http://localhost:9200/secilog/log/?op_type=create&pretty

参数：

```
{
    "collect_type" : "syslog",
    "collect_date" : "2016-01-11T09:32:12",
    "message" : "Failed password for root from 192.168.21.2 port 50790 ssh2"
}
```

返回值：

```
{
  "_index" : "secilog",
  "_type" : "log",
  "_id" : "AVLOV99W6rG7Qqt6i_gk",
  "_version" : 1,
  "_shards" : {
    "total" : 2,
    "successful" : 1,
    "failed" : 0
  },
  "created" : true
}
```

从中可以看出_id 自动生成了一个随机数。

分片选择

默认情况下，分片的选择是通过 ID 的散列值进行控制。这个只可以通过 router 参数进行手动的控制。可以在每个操作的基础上直接通过哈希函数的值来指定分片的选择。例如：

请求：POST http://localhost:9200/secilog/log/?routing=secisland&pretty

在上面的例子中，分片的选择是通过指定 routing=secisland 参数的哈希值来确定的。

其他说明

分布式：索引操作主要是针对主节点的分片进行，当主节点完成索引操作后，如果有副本节点，则分发到副本中。

一致性：为了防止当网络出现问题的时候写入不一致，系统只有在有效节点的数量大于一定数量的时候生效(总结点数/2+1)，该值可以通过 action.write_consistency 参数进行修改。

刷新：更新的时候可以指定 refresh 参数为 true 来立即刷新所有的副本，当 refresh 设置为 true 的时候，系统做了充分的优化，不会对系统产生任何影响，需要注意的是查询操作 refresh 参数没有任何的意义。

空操作：当文档内容没有任何改变的时候，更新文档操作也会生效，具体体现在版本号会发生变化。如果不希望此情况发生，在更新的时候指定 detect_noop 为 true。这个参数在创建索引的时候无效。

超时：默认情况下系统的超时时间是 1 分钟。可以通过设置 timeout 来修改超时的时间，例如 timeout=5m，表示超时的时间是 5 分钟。

2.7.2 更新删除文档

Elasticsearch 的更新文档 API 准许通过脚本操作来更新文档。更新操作从索引中获取文档，执行脚本，然后获得返回结果。它使用版本号来控制文档获取或者重建索引。

备注：在 Elasticsearch 中的更新操作是完全重新索引文件。

我们新建一个文档：

请求：PUT http://localhost:9200/test/type1/1?pretty

参数：

```
{
    "counter" : 1,
    "tags" : ["red"]
}
```

脚本开启功能

在最新版本的 Elasticsearch 中，基于安全考虑(如果用不到，请保持禁用)，默认禁用了动态脚本功能。如果被禁用，在使用脚本的时候则报以下的错误：

scripts of type [inline], operation [update] and lang [groovy] are disabled

可以用以下方式完全开启动态脚本功能，在 config/elasticsearch.yml 文件最后添加以下代码：

```
script.inline: on
script.indexed: on
script.file: on
```

配置后，重启 Elasticsearch。

下面我们用脚本来更新此文档。

请求：POST http://localhost:9200/test/type1/1/_update?pretty

参数：

```
{
    "script" : {
        "inline": "ctx._source.counter += count",
        "params" : {
            "count" : 4
        }
    }
}
```

执行完后，我们再查询一下文档内容，可以发现 counter 的值为 5：

```
{
  "_index" : "test",
  "_type" : "type1",
  "_id" : "1",
  "_version" : 5,
  "found" : true,
  "_source" : {
    "counter" : 5,
```

```
        "tags" : [ "red" ]
      }
    }
```

再看下面的更新操作：

请求：POST http://localhost:9200/test/type1/1/_update?pretty

参数：

```
{
    "script" : {
        "inline": "ctx._source.tags += tag",
        "params" : {
            "tag" : "blue"
        }
    }
}
```

返回的内容为，表示更新成功，我们看一下_version 为 6，比刚才的值增加了 1：

```
{
  "_index" : "test",
  "_type" : "type1",
  "_id" : "1",
  "_version" : 6,
  "_shards" : {
    "total" : 2,
    "successful" : 1,
    "failed" : 0
  }
}
```

然后我们再查询一下文档内容：

```
{
  "_index" : "test",
  "_type" : "type1",
  "_id" : "1",
  "_version" : 6,
  "found" : true,
  "_source" : {
    "counter" : 5,
    "tags" : [ "red", "blue" ]
  }
}
```

在脚本中除了_source 外其他内置参数也可以使用，例如_index、_type、_id、 _version、_routing、_parent、_timestamp、_ttl。

下面我们通过脚本增加一列。

请求：POST http://localhost:9200/test/type1/1/_update?pretty

参数：

```
{
    "script" : "ctx._source.name_of_new_field = \"value_of_new_field\""
}
```

然后查询此文档：

```
{
  "_index" : "test",
  "_type" : "type1",
  "_id" : "1",
  "_version" : 7,
  "found" : true,
  "_source" : {
    "counter" : 5,
    "tags" : [ "red", "blue" ],
    "name_of_new_field" : "value_of_new_field"
  }
}
```

从中可以看出，文档中又增加了一列。

删除一列，请求和刚才的一样，参数变为：

```
{
    "script" : "ctx._source.remove(\"name_of_new_field\")"
}
```

甚至可以通过表达式来判断做某些事情。例如：下面的示例将删除 tag 字段包含 blue 的文件：

请求参数：

```
{
    "script" : {
        "inline": "ctx._source.tags.contains(tag) ? ctx.op = \"delete\" : ctx.op = \"none\"",
        "params" : {
            "tag" : "blue"
        }
    }
}
```

部分文档更新：

该更新接口还支持更新部分文档，将文档合并到现有文档中（简单的递归合并、对象的内部合并、替换核心的“键/值”和数组）。例如：

```
{
    "doc" : {
        "name" : "new_name"
    }
}
```

更新后，可以发现文档中多了一列 name。

```
{
```

```
  "_index" : "test",
  "_type" : "type1",
  "_id" : "1",
  "_version" : 23,
  "found" : true,
  "_source" : {
    "counter" : 5,
    "tags" : [ "red", "blue" ],
    "name" : "new_name"
  }
}
```

当文档指定的值与现有的_source 合并，当新的文档和老的文档不一致的时候，文档将会被重新建立索引。当新旧文档一样的时候，则不进行重建索引的操作。可以通过设置 detect_noop 为 false，让任何情况下都重新建立索引，例如下面的更新操作：

```
{
    "doc" : {
        "name" : "new_name"
    },
    "detect_noop": false
}
```

删除文档

删除文档相对比较简单：

请求：DELETE http://localhost:9200/test/type1/1

返回的内容为：

```
{
    "found": true,
    "_index": "test",
    "_type": "type1",
    "_id": "1",
    "_version": 24,
    "_shards": {
        "total": 2,
        "successful": 1,
        "failed": 0
    }}
```

则表示删除了此文档。

2.7.3 查询文档

Elasticsearch 查询文档 API 准许用户通过文档的 ID 来查询具体的某一个文档，例如下面查询索引为 secilog，type 为 log，id 为 1 的文档：

请求：GET http://localhost:9200/secilog/log/1?pretty

返回的内容为：

```
{
  "_index" : "secilog",
  "_type" : "log",
```

```
  "_id" : "1",
  "_version" : 2,
  "found" : true,
  "_source" : {
    "collect_type" : "syslog",
    "collect_date" : "2016-01-11T09:32:12",
    "message" : "Failed password for root from 192.168.21.2 port 50790 ssh2"
  }
}
```


从返回的内容中可以有很多有用的信息，当然也包括原始的文档信息，放在_source 字段中。

默认情况下，查询获得的数据接口是实时的，并且不受索引的刷新率影响，为了禁用实时性，可以将参数 realtime 设置为 false，或全局设置 action.get.realtime 为 false。

备注：在查询中，中间的_type 是可选的，当不指定具体 type 的时候，可以用_all 来代替。

默认情况下，查询操作会返回_source 字段，当然_source 可以被禁用。例如：

请求：GET http://localhost:9200/secilog/log/1?_source=false&pretty

返回的内容为：


```
{
  "_index" : "secilog",
  "_type" : "log",
  "_id" : "1",
  "_version" : 2,
  "found" : true
}
```


当然如果你想获取 source 中的一部分内容，可以用_source_include 或者_source_exclude 来包含或者过滤其中的某些字段，例如：

请求：GET http://localhost:9200/secilog/log/1?_source_include=message&pretty

返回的内容为：


```
{
  "_index" : "secilog",
  "_type" : "log",
  "_id" : "1",
  "_version" : 2,
  "found" : true,
  "_source" : {
    "message" : "Failed password for root from 192.168.21.2 port 50790 ssh2"
  }
}
```


当一个文档内容非常多的时候，用包含或者过滤可以减少很多的网络负担。如果有多个，可以用逗号分开，或者用*通配符。例如：

请求：GET http://localhost:9200/secilog/log/1?_source_include=message,collect_date&pretty

返回的内容为：

```
{
   "_index" : "secilog",
   "_type" : "log",
   "_id" : "1",
   "_version" : 2,
   "found" : true,
   "_source" : {
      "message" : "Failed password for root from 192.168.21.2 port 50790 ssh2",
      "collect_date" : "2016-01-11T09:32:12"
   }
}
```

通过 fields 字段过滤，可以从存储中查询一组字段，例如：

请求：GET http://localhost:9200/secilog/log/1?fields=message,collect_date&pretty

返回的内容：

```
{
   "_index" : "secilog",
   "_type" : "log",
   "_id" : "1",
   "_version" : 2,
   "found" : true,
   "fields" : {
      "message" : [ "Failed password for root from 192.168.21.2 port 50790 ssh2" ],
      "collect_date" : [ "2016-01-11T09:32:12" ]
   }
}
```

备注：从返回值可以看出，返回的字段是数组类型的，但_routing 字段和_parent 字段是没有数组返回的。只有子字段可以从 fields 中进行查询，对象数据是不生效的。

如果建立索引后还没有来得及刷新，查询得到的内容是事务的日志。但有些字段只有在索引的时候才会产生，当访问这些字段的时候，系统会抛出一个异常。可以通过设置 ignore_errors_on_generated_fields=true 来忽略这些字段。

只获取文档内容

可以通过 /{index}/{type}/{id}/_source 的方式只获取文档内容，例如：

请求：GET http://localhost:9200/secilog/log/1/_source?pretty

返回的内容：

```
{
   "collect_type" : "syslog",
   "collect_date" : "2016-01-11T09:32:12",
   "message" : "Failed password for root from 192.168.21.2 port 50790 ssh2"
}
```

同理，这种方式的查询也可以通过之前的过滤方式来选择具体的字段。

分片选择(routing)

可以在查询的时候指定路由选择(routing)，当路由不存在的时候，返回为空值，此实例是事先做了路由的操作，例如：

请求：GET http://localhost:9200/secilog/log/1?routing=secisland&pretty

返回的内容：


```
{
  "_index" : "secilog",
  "_type" : "log",
  "_id" : "1",
  "_version" : 1,
  "_routing" : "secisland",
  "found" : true,
  "_source" : {
    "collect_type" : "syslog",
    "collect_date" : "2016-01-11T09:32:12",
    "message" : "Failed password for root from 192.168.21.2 port 50790 ssh2"
  }
}
```


通过参数控制，查询的时候可以指定查询是在主节点上查询还是在副本节点上查询。

_primary：在主节点进行查询；

_local：尽可能在本地节点上进行查询；

刷新参数：

refresh 参数可以被设置为 true，使之在搜索操作前刷新相关的分片保证可以及时查询到。但这个参数会消耗系统的资源，除非有必要，正常情况下不需要设置。

2.7.4 多文档操作(bulk)

在 Elasticsearch 对文档的操作中，之前介绍的都是对单个文档进行操作，其实 Elasticsearch 可以对多个文档同时操作。下面介绍多文档查询。

多文档查询

多文档查询可以在同一个接口中查询多个文档，可以分别指定 index，type，id 来进行多个文档的查询。响应包括所有查询到的文档数组，每个元素在结构上类似于单个文档查询。例如：

请求：POST http://localhost:9200/_mget?pretty

参数：


```
{
    "docs" : [
        {
            "_index" : "secilog",
            "_type" : "log",
            "_id" : "1"
        },
        {
            "_index" : "secilog",
            "_type" : "log",
            "_id" : "2"
        }
```

```
        ]
    }
```

返回结果：

```
{
  "docs" : [ {
    "_index" : "secilog",
    "_type" : "log",
    "_id" : "1",
    "_version" : 3,
    "found" : true,
    "_source" : {
      "collect_type" : "syslog",
      "collect_date" : "2016-01-11T09:32:12",
      "message" : "Failed password for root from 192.168.21.2 port 50790 ssh2"
    }
  }, {
    "_index" : "secilog",
    "_type" : "log",
    "_id" : "2",
    "_version" : 1,
    "found" : true,
    "_source" : {
      "collect_type" : "syslog",
      "collect_date" : "2016-01-12T09:32:12",
      "message" : "secisland mget test!"
    }
  } ]
}
```

从中可以看出，一次查询了两个文档。

在查询的时候，index，type 可以在 URL 中直接填写。例如下面两个请求和之前的是等价的。

请求：POST http://localhost:9200/secilog/_mget?pretty

参数：

```
{
    "docs" : [
        {
            "_type" : "log",
            "_id" : "1"
        },
        {
            "_type" : "log",
            "_id" : "2"
        }
    ]
```

}

请求：POST http://localhost:9200/secilog/log/_mget?pretty

参数：

```
{
    "docs" : [
        {
            "_id" : "1"
        },
        {
            "_id" : "2"
        }
    ]
}
```

对于上一种，可以用更加简化的方式查询：

请求：POST http://localhost:9200/secilog/log/_mget?pretty

参数：

```
{
    "ids" : ["1","2" ]
}
```

从上面的例子可以看出，Elasticsearch 的多文档查询还是很灵活的。

type 参数说明

在多文档查询中，_type 允许为空，当他设置为空或者_all 的时候，系统会匹配第一个查询到的结果。如果不设置_type，当有许多文件有相同的_id 的时候，系统最终得到的只有第一个匹配的文档。例如：

请求：POST http://localhost:9200/secilog/_mget?pretty

参数：

```
{
    "ids" : ["1","2" ]
}
```

上面的查询当有多个 type 中都有 1，2 两个 id 的时候，系统只会返回第一个找到的文档。如果想要多个，就需要把 type 在请求参数中指出来。

默认情况下，_source 字段将在每个文件中返回（如果存储）。类似单个文档的查询，可以在 URL 中指定_source， _source_include 或者_source_exclude 来对查询的结果进行过滤。例如：

请求：POST http://localhost:9200/secilog/log/_mget?pretty

参数：

```
{
    "docs" : [
        {
            "_id" : "1",
            "_source" : false
        },
        {
            "_id" : "2",
```

```
                "_source" : ["collect_type", "collect_date"]
            }
        ]
}
```

返回结果：

```
{
  "docs" : [ {
    "_index" : "secilog",
    "_type" : "log",
    "_id" : "1",
    "_version" : 3,
    "found" : true
  }, {
    "_index" : "secilog",
    "_type" : "log",
    "_id" : "2",
    "_version" : 1,
    "found" : true,
    "_source" : {
      "collect_date" : "2016-01-12T09:32:12",
      "collect_type" : "syslog"
    }
  } ]
}
```

类似单个文档查询，在请求的 URL 中或者参数的 docs 中可以指定 field，routing 参数。

块操作

块操作可以在一个接口中处理文档的内容，包括创建文档，删除文档，和修改文档。用块操作方式操作多个文档可以提高系统的效率。例如：

请求：POST http://localhost:9200/_bulk?pretty

参数：

```
{ "index" : { "_index" : "test", "_type" : "type1", "_id" : "10" } }
{ "field1" : "value1" }
{ "index" : { "_index" : "test", "_type" : "type1", "_id" : "13" } }
{ "field1" : "value3" }
{ "delete" : { "_index" : "test", "_type" : "type1", "_id" : "12" } }
}
```

返回结果：

```
{
  "took" : 1,
  "errors" : false,
  "items" : [ {
    "index" : {
```

```
            "_index" : "test",
            "_type" : "type1",
            "_id" : "10",
            "_version" : 6,
            "_shards" : {
              "total" : 2,
              "successful" : 1,
              "failed" : 0
            },
            "status" : 200
          }
        }, {
          "index" : {
            "_index" : "test",
            "_type" : "type1",
            "_id" : "13",
            "_version" : 1,
            "_shards" : {
              "total" : 2,
              "successful" : 1,
              "failed" : 0
            },
            "status" : 201
          }
        }, {
          "delete" : {
            "_index" : "test",
            "_type" : "type1",
            "_id" : "12",
            "_version" : 2,
            "_shards" : {
              "total" : 2,
              "successful" : 1,
              "failed" : 0
            },
            "status" : 404,
            "found" : false
          }
        } ]
      }
```


和批量查询类似， /_bulk， /{index}/_bulk，and {index}/{type}/_bulk 这三种方式都可以执行，只需要在请求的参数中做出相应的对应。

2.7.5 索引词频率

term vector 是在 Lucene 中的一个概念，就是对于文档的某一列，如 title、body 这种文本类型的，建立词频的多维向量空间，每一个词就是一个维度，这个维度的值就是这个词在这个列中的频率。在 Elasticsearch 中 termvectors 返回在索引中特定文档字段的统计信息，termvectors 在 Elasticsearch 中是实时分析的，如果要想不实时分析，可以设置 realtime 参数为 false。默认情况下索引词频率统计是关闭的，需要在建索引的时候手工打开。

注意：在 Elasticsearch2.0 版本以上用_termvectors 代替_termvector。

下面我们建一个打开了索引词统计的索引。

请求：PUT http://localhost:9200/secilog/

参数：

```
{
  "mappings": {
    "log": {
      "properties": {
        "type": {
          "type": "string",
          "term_vector": "with_positions_offsets_payloads",
          "store" : true,
          "analyzer" : "fulltext_analyzer"
         },
         "message": {
          "type": "string",
          "term_vector": "with_positions_offsets_payloads",
          "analyzer" : "fulltext_analyzer"
        }
      }
    }
  },
  "settings" : {
    "index" : {
      "number_of_shards" : 1,
      "number_of_replicas" : 0
    },
    "analysis": {
      "analyzer": {
        "fulltext_analyzer": {
          "type": "custom",
          "tokenizer": "whitespace",
          "filter": [
            "lowercase",
            "type_as_payload"
          ]
        }
      }
```

```
      }
    }
}
```

然后我们插入两条数据：

请求：PUT http://localhost:9200/secilog/log/1/?pretty

参数：

```
{
   "type" : "syslog",
   "message" : "secilog test test test "
}
```

请求：PUT http://localhost:9200/secilog/log/2/?pretty

参数：

```
{
   "type" : "file",
   "message" : "Another secilog test "
}
```

当创建两条日志成功后，我们用_termvectors 来查询统计结果。

请求：GET http://localhost:9200/secilog/log/1/_termvectors?pretty=true

返回结果如下：

```
{
   "_index" : "secilog",
   "_type" : "log",
   "_id" : "1",
   "_version" : 1,
   "found" : true,
   "took" : 2,
   "term_vectors" : {
      "message" : {
         "field_statistics" : {
            "sum_doc_freq" : 5,
            "doc_count" : 2,
            "sum_ttf" : 7
         },
         "terms" : {
            "secilog" : {
               "term_freq" : 1,
               "tokens" : [ {
                  "position" : 0,
                  "start_offset" : 0,
                  "end_offset" : 7,
                  "payload" : "d29yZA=="
               } ]
            },
            "test" : {
```

```
            "term_freq" : 3,
            "tokens" : [ {
              "position" : 1,
              "start_offset" : 8,
              "end_offset" : 12,
              "payload" : "d29yZA=="
            }, {
              "position" : 2,
              "start_offset" : 13,
              "end_offset" : 17,
              "payload" : "d29yZA=="
            }, {
              "position" : 3,
              "start_offset" : 18,
              "end_offset" : 22,
              "payload" : "d29yZA=="
            } ]
          }
        }
      },
      "type" : {
        "field_statistics" : {
          "sum_doc_freq" : 2,
          "doc_count" : 2,
          "sum_ttf" : 2
        },
        "terms" : {
          "syslog" : {
            "term_freq" : 1,
            "tokens" : [ {
              "position" : 0,
              "start_offset" : 0,
              "end_offset" : 6,
              "payload" : "d29yZA=="
            } ]
          }
        }
      }
    }
  }
```


从中可以看出，每个字段，每个单词出现的次数和位置。需要注意的是对这些字段统计不是完全精确的，已删除的文件未被考虑在内，信息统计所请求的文档只统计所在的分片，除非 DFS 设置为 true。因此，索引词的统计数据对于了解索引词的频率有参考意义，默认情

况下当情况索引词频率查询的时候，系统会随机的指定一个分片进行统计，如果使用 routing 可以查询具体某个分片的统计情况。对于索引词统计，还可以指定参数查询，例如：

请求：POST http://localhost:9200/secilog/log/1/_termvectors?pretty=true

参数：

```
{
   "fields" : ["message"],
   "offsets" : true,
   "payloads" : true,
   "positions" : true,
   "term_statistics" : true,
   "field_statistics" : true
}
```

返回结果：

```
{
   "_index" : "secilog",
   "_type" : "log",
   "_id" : "1",
   "_version" : 1,
   "found" : true,
   "took" : 2,
   "term_vectors" : {
      "message" : {
         "field_statistics" : {
            "sum_doc_freq" : 5,
            "doc_count" : 2,
            "sum_ttf" : 7
         },
         "terms" : {
            "secilog" : {
               "doc_freq" : 2,
               "ttf" : 2,
               "term_freq" : 1,
               "tokens" : [ {
                  "position" : 0,
                  "start_offset" : 0,
                  "end_offset" : 7,
                  "payload" : "d29yZA=="
               } ]
            },
            "test" : {
               "doc_freq" : 2,
               "ttf" : 4,
               "term_freq" : 3,
               "tokens" : [ {
```

```
                "position" : 1,
                "start_offset" : 8,
                "end_offset" : 12,
                "payload" : "d29yZA=="
              }, {
                "position" : 2,
                "start_offset" : 13,
                "end_offset" : 17,
                "payload" : "d29yZA=="
              }, {
                "position" : 3,
                "start_offset" : 18,
                "end_offset" : 22,
                "payload" : "d29yZA=="
              } ]
            }
          }
        }
      }
    }
```


从上面的查询中可以看出，对统计进行了过滤，只查询了一部分的统计。

需要注意的是打开了索引词频率会增加系统的负担，除非特别有必要才需要打开统计。

2.7.6 查询更新接口

查询更新接口是 2.3.0 增加新接口，这个接口目前是实验性的。此接口可能会在未来的版本中改变。此接口在索引中更新每一个文档，而文档的内容并没有改变的情况下使用，这在增加新的属性或者修改映射的时候非常有用。例如：

请求：POST /secisland/_update_by_query?conflicts=proceed

返回结果为：


```
{
  "took" : 639,
  "updated": 1235,
  "batches": 13,
  "version_conflicts": 2,
  "failures" : [ ]
}
```


_update_by_query 当系统开始或者正在用内部版本号进行索引的时候获得一个索引的快照。当快照被处理时或者当索引请求被处理时，如果文档发生了变化，系统将会产生版本冲突。当匹配版本的文档更新后，文档的版本号会递增。当更新和查询失败时，_update_by_query 将会中止并在返回值中返回失败的原因。已执行的更新将会保持。也就是说，这个过程是不回滚的，只有中止。当第一条失败时，会导致程序中止，这意味着所有的更新失败，这时候会返回大量的失败元素。

如果你想当版本冲突时只进行简单的计数，而不中止，可以在 URL 中设置 conflicts=proceed 或者在请求内容中设置"conflicts": "proceed"，在 API 中，可以只针对索引中的一个类型进行操作。例如：

请求：POST /secisland/tweet/_update_by_query?conflicts=proceed

同样可以采用 DSL 语法进行查询，例如：

请求：POST /secisland/_update_by_query?conflicts=proceed

```
{
  "query": {
    "term": {
      "user": "kimchy"
    }
  }
}
```

这里的搜索和其他搜索的语法是一致的。

之前介绍的都是没有改变文档内容的，其实_update_by_query 是可以支持脚本对文档内容的更新。例如：

请求：POST /secisland/_update_by_query

```
{
  "script": {
    "inline": "ctx._source.likes++"
  },
  "query": {
    "term": {
      "user": "kimchy"
    }
  }
}
```

这个接口可以在多个索引和多个类型中同时操作，例如：

请求：POST /secisland,blog/tweet,post/_update_by_query

同时，也可以指定路由查询，例如：

请求：POST /secisland/_update_by_query?routing=1

默认情况下，_update_by_query 采用滚动 100 批次。你可以在 URL 参数用 scroll_size 来改变批次的大小，例如：

请求：POST /secisland/_update_by_query?scroll_size=1000

除了标注的参数外，_update_by_query 还支持 refresh，wait_for_completion，consistency，timeout。

刷新(refresh)操作只是当请求完成时再更新所有分片。这不同于更新索引时的刷新，索引的时候当收到新的数据时，只针对当前数据分片进行刷新。如果请求中包含 wait_for_completion=true，Elasticsearch 将进行执行前检查，启动请求，然后返回任务，用这个任务可以取消操作或获得任务的状态。一旦请求完成任务就结束了，唯一有记录的地方是在 Elasticsearch 日志文件中有任务的执行结果，这个问题将在以后的版本修复。一致性(consistency)控制每次请求时有多少分片的拷贝被响应，超时控制每次请求等待的时间。在 Bulk API 中可以精确的知道他们是如何工作的。超时控制多久每批等待成为目标碎片。

响应介绍，每次响应的内容大概是：

```
{
  "took" : 639,
  "updated": 0,
```

```
    "batches": 1,
    "version_conflicts": 2,
    "failures" : [ ]
}
```

took：从开始到结束的整个操作的毫秒数。

updated：已成功更新的文档数。

batches：滚动响应的数量。

version_conflicts：通过查询命中更新的版本冲突的数量。

failures：所有索引失败的数组。如果这是非空的，则请求中止。

当查询更新操作发生后，可以使用任务接口来获取它们的状态，例如：

请求：POST /_tasks/?pretty&detailed=true&action=*byquery

响应：

```
{
  "nodes" : {
    "r1A2WoRbTwKZ516z6NEs5A" : {
      "name" : "Tyrannus",
      "transport_address" : "127.0.0.1:9300",
      "host" : "127.0.0.1",
      "ip" : "127.0.0.1:9300",
      "attributes" : {
        "testattr" : "test",
        "portsfile" : "true"
      },
      "tasks" : {
        "r1A2WoRbTwKZ516z6NEs5A:36619" : {
          "node" : "r1A2WoRbTwKZ516z6NEs5A",
          "id" : 36619,
          "type" : "transport",
          "action" : "indices:data/write/update/byquery",
          "status" : {
            "total" : 6154,
            "updated" : 3500,
            "created" : 0,
            "deleted" : 0,
            "batches" : 36,
            "version_conflicts" : 0,
            "noops" : 0
          },
          "description" : ""
        }
      }
    }
  }
}
```

该对象包含实际状态。里面有个重要的参数是 total，预计重建执行总的操作数。你可以通过添加更新、创建和删除字段来估计这个进展。当它们的总和等于总的列时，该请求将完成。

当创建了一个没有动态映射的索引，如果有新的数据内容，系统会添加新的映射值来匹配数据中的多个字段，例如：

请求：PUT test，数据结构只有 test

```
{
  "mappings": {
    "test": {
      "dynamic": false,
      "properties": {
        "text": {"type": "string"}
      }
    }
  }
}
```

插入数据：

请求：POST test/test?refresh

```
{
  "text": "words words",
  "flag": "bar"
}
```

然后得到数据结构：

请求：PUT test/_mapping/test

```
{
  "properties": {
    "text": {"type": "string"},
    "flag": {"type": "string", "analyzer": "keyword"}
  }
}
```

这个时候会自动添加一个 flag 映射。

但这个时候查询数据，不会找到任何值：

请求：POST test/_search?filter_path=hits.total

参数：

```
{
  "query": {
    "match": {
      "flag": "bar"
    }
  }
}
```

返回：

```
{
  "hits" : {
```

```
      "total" : 0
    }
  }
```

这个时候我们可以用更新的查询请求来获得数据，例如：

请求：POST test/_update_by_query?refresh&conflicts=proceed

POST test/_search?filter_path=hits.total

参数：

```
{
  "query": {
    "match": {
      "flag": "foo"
    }
  }
}
```

返回：

```
{
  "hits" : {
    "total" : 1
  }
}
```

2.8 小结

本章介绍了 Elasticsearch 的索引相关的知识，包括索引管理，索引映射管理等索引高级用法；索引别名管理，类似数据库的索引可以管理多个索引；索引监控，索引状态管理；最后介绍了文档的详细操作，通过以上介绍基本上可以对 Elasticsearch 进行基本的使用。

下一章节重点介绍 Elasticsearch 的映射，通过映射可以了解 Elasticsearch 内部结构。

3 映射

映射是定义存储和索引的文档类型以及字段的过程。索引中的每一个文档都有一个类型，每种类型都有它自己的映射。一个映射定义了文档结构内的每个字段的数据类型。映射是通过配置来定义字段类型与该类型相关联的元数据的关系。例如，可以通过映射来定义日期类型的格式，数字类型的格式或者文档中所有字段的值是否应该被_all 字段索引等。

3.1 概念

A. 映射类型

每个索引拥有一个或更多的映射类型，用来在索引中将文档划分为不同的逻辑组。

每个映射类型拥有：

- 元字段

元字段用来定义如何处理文档的元数据。元字段包括文档的_index 字段，_type 字段，_id 字段和_source 字段。

- 字段或属性

每个映射类型包含与类型相关的字段或属性列表。同一索引中不同映射类型的相同名称字段必须拥有相同的映射。

B. 字段数据类型

每个字段拥有一个数据类型，可以是：

简单数据类型，比如字符串型（String），日期型（date），长整型（long），双精度浮点型（double），布尔型（boolean）或者 IP。

支持 JSON 的层次性类型，比如对象（object）、嵌套（nested）或者特定的类型，比如地理点（geo_point），地理模型（geo_shape）。

基于不同的目的对同一个字段进行不同方式的索引是很有用的。例如，一个字符串类型字段可以在全文搜索中作为分析字段，在排序或聚合时作为不分析的字段。或者，你可以通过标准分析器，英文分析器或者法语分析器对字符串字段进行索引。

一个数据类型通过 fields 参数支持多字段。

C. 动态映射

字段和映射类型在使用前不需要被事先定义。依靠动态映射，通过索引文档，新的映射类型和字段名会被自动添加。新的字段可以添加到顶级映射类型或者映射内部的对象和嵌入字段。

动态映射可以配置自定义映射用于新类型或者新字段。

D. 显式映射

相对于 Elasticsearch 来说，我们对于数据类型的掌控更加全面，所以我们可以指定显示映射而不是使用动态映射。

当我们创建索引的时候，可以创建映射类型和字段。也可以在当前的索引中通过映射创建接口添加映射类型和字段。

E. 更新当前映射

除了记录，现有的映射类型和字段不能被更新。修改映射意味着废弃已经索引的文档，我们反而应该根据映射创建新的索引并且重新索引数据。

F. 映射类型之间共享字段

映射类型在每个索引中是唯一的，就是在一个索引的多个类型中，如果多个类型中的映射名称一样，则它必须是相同的类型。例如：

如果一个 title 字段同时存在于 user 和 blogpost 映射类型，title 字段在每个类型中必须拥有相同的映射。这个规则的唯一例外是：对于 copy_to 参数，dynamic 参数，enabled 参数，ignore_above 参数，include_in_all 参数，每个不同映射类型中的字段拥有不同的参数设置。

通常，相同名称的字段由相同类型的数据构成，所以拥有相同的索引是没有问题的。当产生类型冲突的时候，可以选择更详细的命名，比如 user_title 和 blog_title。

G. 映射示例

当创建索引的时候，可以指定映射：

```
PUT secisland
{
  "mappings": {
    "user": {
      "_all":          { "enabled": false  },
      "properties": {
        "title":      { "type": "string"  },
        "name":       { "type": "string"  },
        "age":        { "type": "integer" }
      }
    },
```

```
    "blogpost": {
      "properties": {
        "title":       { "type": "string"    },
        "body":        { "type": "string"    },
        "user_id":   {
          "type":     "string",
          "index":    "not_analyzed"
        },
        "created":   {
          "type":     "date",
          "format": "strict_date_optional_time||epoch_millis"
        }
      }
    }
  }
```

上面的接口意思是创建一个名为 secisland 的索引，在索引中添加名为 user 和 blogpost 的映射类型。user 映射类型取消元字段_all，指定了每个映射类型的字段或属性，指定了每个字段的数据类型和映射。

3.2 字段数据类型

Elasticsearch 支持一系列不同的数据类型来定义文档字段：

A. 核心数据类型：
- 字符串数据类型：string
- 数字型数据类型：long，integer，short，byte，double，float
- 日期型数据类型：date
- 布尔型数据类型：boolean
- 二进制数据类型：binary

B. 复杂数据类型：
- 数组数据类型：不需要专门的类型来定义数组
- 对象数据类型：object 单独的 JSON 对象
- 嵌套数据类型：nested 关于 JSON 对象的数组

C. 地理数据类型：
- 地理点数据类型：geo_point 经纬点
- 地理模型数据类型：geo_shape 多边形的复杂地理模型

D. 专门数据类型
- IPv4 数据类型：ip 协议为 IPv4 的 IP 地址
- 完成数据类型：completion 提供自动补全的建议
- 词元计数数据类型：token_count 统计字符串中的词元数量

3.2.1 字符串数据类型

字符串类型的字段接收文本值。字符串可以分为：

全文本

全文本值，通常被用于基于文本的相关性搜索，比如：找到相关度最高的文档，匹配查询“棕色狐狸”。

这些字段可以进行分词，就是说在索引执行之前通过一个分词器将字符串转换为单词列表。分词操作使 Elasticsearch 可以在全文本字段上搜索单词。全文本字段不用于排序而且很少用于聚合。

关键字

关键字是个精准值，通常被用于过滤（例如，获取所有 status 字段值为 published 的博客文章），排序以及参与聚合。关键字字段不参与分词（not_analyzed）。

全文本（可以分词）字符串字段和关键字（不可以分词）字符串字段映射示例：

```
PUT secisland
{
  "mappings": {
    "my_type": {
      "properties": {
        "full_name": {
          "type":   "string"
        },
        "status": {
          "type":   "string",
          "index": "not_analyzed"
        }
      }
    }
  }
}
```

full_name 字段是一个可分词的全文本类型字段——index：默认是 analyzed。

status 字段是一个不可分词的关键字字段。

同一个字段同时拥有全文本和关键字两个版本，通常是很有用的：一个用于全文本搜索，另一个用于聚合和排序。这可以通过多字段来实现。

字符串型字段可以接受以下参数：

analyzer	分词器可以被用于可分词的字符串型字段。默认为默认的索引分词器，或者标准分词器。
boost	字段级索引加权。接受浮点型数字，默认值是 1.0。
doc_values	定义字段是否应该以列跨度的方式存储在磁盘上，以便用于排序，聚合或者脚本？接受 true 或 false 参数。对于不可分词字段，默认值是 true。可分词字段不支持这个参数。
fielddate	决定字段是否可以使用内存字段值进行排序，聚合或者在脚本中使用。接受 disabled 或者 paged_bytes（默认）参数。没有分析过的字段会优先使用文档值。
ignore_above	不要索引或执行任何长于这个值的字符串。默认为 0（禁用）。
include_in_all	决定字段是否应该被包含在_all 字段中。接受 true 或 false 参数。如果索引被设置为 no 或者父对象字段设置 include_in_all 为 false，参数默认值为 false；其它情况下，默认值为 true。
index	决定字段是否可以被用户搜索。接受参数 analyzed（默认，视为全文字段），not_analyzed（作为关键字字段）以及 no。
index_options	定义存储在索引中，用于搜索和突出用途的信息。

norms	计算查询得分的时候是否应该考虑字段长度。默认依赖于索引设置： analyzed 字段默认{ "enabled": true, "loading": "lazy" }。not_analyzed 字段默认{ "enabled": false }。
null_value	接受一个字符串值替换所有 null 值。默认为 null，意味着字段被作为缺失字段。如果字段是可分词（analyzed）的，null_value 也会被分词。
position_increment_gap	定义字符串数组中应该插入的虚拟索引词的数量。默认值为 100，以一个较合理的值来阻止短语查询在跨字段匹配索引词的时候溢出。
store	决定字段值是否应该被存储以及从_source 字段分别获取。接受参数 true 或 false（默认）。
search_analyzer	指定搜索时用在可分词字段上的分词器。
search_quote_analyzer	指定搜索短语时使用的分词器。
similarity	指定使用的相似度评分算法，默认为 TF/IDF。
term_vector	定义一个可分词字段是否应该存储索引词向量。默认为 no。

表格 3.1 字符串型字段接受的参数

3.2.2 数字型数据类型

数字型数据类型支持下列数字类型：

long	一个有符号的 64 位整数，最小值为-2^{63}，最大值为 2^{63}-1。
integer	一个有符号的 32 位整数，最小值为-2^{31}，最大值为 2^{31}-1。
short	一个有符号的 16 位整数，最小值为-32768，最大值为 32767。
byte	一个有符号的 8 位整数，最小值为-128，最大值为 127。
double	64 位双精度浮点数。
float	32 位单精度浮点数。

表格 3.2 数字型数据类型

数字型字段映射配置示例：

```
PUT secisland
{
  "mappings": {
    "my_type": {
      "properties": {
        "number_of_bytes": {
          "type": "integer"
        },
        "time_in_seconds": {
          "type": "float"
        }
      }
    }
  }
}
```

coerce	试着将字符串型数据转换为整数型数字数据。
boost	字段级索引加权，接受浮点型数字参数，默认为 1.0。

doc_values	定义字段是否应该以列跨度的方式存储在磁盘上，以便用于排序，聚合或者脚本？接受 true（默认）或 false 参数。
ignore_malformed	如果是 true，畸形的数字会被忽略。如果是 false（默认），畸形数字会抛出异常并丢弃整个文档。
include_in_all	决定字段是否应该被包含在_all 字段中。接受 true 或 false 参数。如果索引被设置为 no 或者父对象字段设置 include_in_all 为 false，参数默认值为 false；其他情况下，默认值为 true。
index	决定字段是否可以被用户搜索。接受参数 not_analyzed（默认）以及 no。
null_value	接受与字段同类型的数字型值来代替 null 值。默认是 null，意味着字段被作为缺失字段。
precision_step	控制索引的额外索引词的数量来使范围查询更快速。默认值取决于数字类型。
store	决定字段值是否应该被存储以及从_source 字段分别获取。接受参数 true 或 false（默认）。

表格 3.3 数字型字段参数

3.2.3 日期型数据类型

JSON 没有日期型数据类型，所以在 Elasticsearch 中，日期可以是：

包含格式化日期的字符串，例如“2015-01-01”或者“2015/01/01 12:10:30”

代表时间毫秒数的长整型数字。

代表时间秒数的整数。

通常，日期被转换为 UTC（如果时区被指定）但是存储为代表时间毫秒数的长整数。

可以自定义时间格式，如果没有指定格式，则使用默认值：

"strict_date_optional_time||epoch_millis"

这意味着接受任意时间戳的日期值，例如：

```
PUT secisland
{
  "mappings": {
    "my_type": {
      "properties": {
        "date": {
          "type": "date"
        }
      }
    }
  }
}
```

创建映射之后，可以放置日期型数据：

```
PUT secisland/my_type/1
{ "date": "2015-01-01" }

PUT secisland/my_type/2
{ "date": "2015-01-01T12:10:30Z" }

PUT secisland/my_type/3
```

{ "date": 1420070400001 }

A. 多日期格式

使用双竖线（||）分隔，可以指定多个日期格式。每个格式会被依次尝试，直到找到匹配的格式。第一个格式会被用于将时间毫秒数值转换为字符串。

```
PUT secisland
{
  "mappings": {
    "my_type": {
      "properties": {
        "date": {
          "type":    "date",
          "format": "yyy-MM-dd HH:mm:ss||yyyy-MM-dd||epoch_millis"
        }
      }
    }
  }
}
```

B. 日期型字段参数

boost	字段级索引加权，接受浮点型数字参数，默认为 1.0。
doc_values	定义字段是否应该以列跨度的方式存储在磁盘上，以便用于排序，聚合或者脚本？接受 true（默认）或 false 参数。
format	可解析的日期格式。默认为 strict_date_optional_time||epoch_millis。
ignore_malformed	如果是 true，畸形的日期会被忽略。如果是 false（默认），畸形日期会抛出异常并丢弃整个文档。
include_in_all	决定字段是否应该被包含在_all 字段中。接受 true 或 false 参数。如果索引被设置为 no 或者父对象字段设置 include_in_all 为 false，参数默认值为 false；其他情况下，默认值为 true。
index	决定字段是否可以被用户搜索。接受参数 not_analyzed（默认）以及 no。
null_value	接受日期型值来代替 null 值。默认是 null，意味着字段被作为缺失字段。
precision_step	控制索引的额外索引词的数量来使范围查询更快速。默认值为 16。
store	决定字段值是否应该被存储以及从_source 字段分别获取。接受参数 true 或 false（默认）。

表格 3.4 日期型字段参数

3.2.4 布尔数据类型

布尔型字段接受 true 或 false 值，也可以接受代表真或假的字符串和数字：

假值 false，“false”，“off”，“no”，“0”，“”（空字符串），0，0.0

真值 其他任何非假的值

示例：

```
PUT secisland
{
  "mappings": {
    "my_type": {
      "properties": {
        "is_published": {
```

```
            "type": "boolean"
          }
        }
      }
    }
}

POST secisland/my_type/1
{
  "is_published": true
}
```

索引词聚合之类的聚合使用 1 和 0 作为 key，使用字符串“true”和“false”作为 key_as_string。使用脚本时，布尔字段返回 1 和 0。

boost	字段级索引加权，接受浮点型数字参数，默认为 1.0。
doc_values	定义字段是否应该以列跨度的方式存储在磁盘上，以便用于排序，聚合或者脚本？接受 true（默认）或 false 参数。
index	决定字段是否可以被用户搜索。接受参数 not_analyzed（默认）以及 no。
null_value	接受布尔型值来代替 null 值。默认是 null，意味着字段被作为缺失字段。
store	决定字段值是否应该被存储以及从_source 字段分别获取。接受参数 true 或 false（默认）。

表格 3.5 布尔型字段参数

3.2.5 二进制数据类型

二进制数据类型接受 Base64 编码字符串的二进制值。字段不以默认方式存储而且不能被搜索:

```
PUT secisland
{
  "mappings": {
    "my_type": {
      "properties": {
        "name": {
          "type": "string"
        },
        "blob": {
          "type": "binary"
        }
      }
    }
  }
}

PUT secisland/my_type/1
{
  "name": "Some binary blob",
  "blob": "U29tZSBiaW5hcnkgYmxvYg=="
```

}

Base64 编码二进制值不能嵌入换行符\n。

doc_values	定义字段是否应该以列跨度的方式存储在磁盘上，以便用于排序，聚合或者脚本？接受 true（默认）或 false 参数。
store	决定字段值是否应该被存储以及从_source 字段分别获取。接受参数 true 或 false（默认）。

表格 3.6　二进制型字段参数

3.2.6　数组数据类型

在 Elasticsearch 中，没有专门的数组类型。每个字段默认可以包含零个或更多的值，然而，数组中所有的值都必须是相同的数据类型。例如：

字符串数组：["one", "two"]

整数数组：[1,2]

由数组组成的数组：[1,[2,3]]，等同于[1,2,3]

对象数组：[{ "name": "Mary", "age": 12 }, { "name": "John", "age": 10 }]

注解：无法对数组中的每一个对象进行单独的查询。

当动态添加字段的时候，数组中第一个元素的值决定了字段类型，随后的所有值必须是相同的数据类型或者可以强制转换为相同的数据类型。

不支持混合数据类型的数组，比如：[10, "some string"]

数组可能包含 null 值，会被 null_value 配置替换掉或者忽略掉。一个空数组[]被当作缺失字段——没有值的字段。

3.2.7　对象数据类型

JSON 文档是天然分层的：文档可以包含内部对象，同样，内部对象也可以包含内部对象。

```
PUT secisland/my_type/1
{
  "region": "US",
  "manager": {
    "age":      30,
    "name": {
      "first": "John",
      "last":   "Smith"
    }
  }
}
```

本质上，文档被简单地索引为键值对的列表，形如：

```
{
  "region":                  "US",
  "manager.age":             30,
  "manager.name.first": "John",
  "manager.name.last":   "Smith"
}
```

上面文档的映射结构为：

```
PUT secisland
{
```

```
"mappings": {
  "my_type": {
    "properties": {
      "region": {
        "type": "string",
        "index": "not_analyzed"
      },
      "manager": {
        "properties": {
          "age":    { "type": "integer" },
          "name": {
            "properties": {
              "first": { "type": "string" },
              "last":    { "type": "string" }
            }
          }
        }
      }
    }
  }
}
```

映射类型是一种对象类型，拥有参数字段，manager 字段是一个内部对象字段，manager.name 字段是 manager 字段中的内部对象字段。

可以明确地设置 type 字段为 object（默认值）。

dynamic	定义新的参数是否应该被动态加入到已经存在的对象中。接受 true（默认），false 和 strict。
enabled	赋值给对象字段的 JSON 值应该被解析和索引（true，默认）还是完全忽略（false）。
include_in_all	为对象内的所有属性设置 include_in_all 值。对象本身不添加到_all 字段。
properties	对象内的字段，可以是任意数据类型，包括对象数据类型。新的属性可以被添加到已存在的对象中。

表格 3.7　对象字段参数

3.2.8　嵌套数据类型

嵌套数据类型是对象数据类型的一个专门的版本，用来使一组对象被单独地索引和查询。

A.　对象数组是如何摊平的

Lucene 没有内部对象的概念，所以 Elasticsearch 利用简单的列表存储字段名和值，将对象层次摊平。例如，下面的文档：

```
PUT secisland/my_type/1
{
  "group" : "fans",
  "user" : [
    {
```

```
        "first" : "John",
        "last" :   "Smith"
      },
      {
        "first" : "Alice",
        "last" :   "White"
      }
    ]
}
```

内部转换为文档，结构如下：

```
{
  "group" :              "fans",
  "user.first" : [ "alice", "john" ],
  "user.last" :   [ "smith", "white" ]
}
```

user.first 和 user.last 字段被存在多值字段中，alice 和 white 的关联性丢失了。这个文档可能错误地匹配到关于 alice 和 smith 的查询。

B. 对一组对象使用嵌套字段

如果需要对一组对象进行索引而且保留数组中每个对象的独立性，可以使用嵌套数据类型而不是对象数据类型。本质上，嵌套对象将数组中的每个对象作为分离出来的隐藏文档进行索引。这也意味着每个嵌套对象可以独立于其它对象被查询:

创建映射

```
PUT secisland
{
  "mappings": {
    "my_type": {
      "properties": {
        "user": {
          "type": "nested"
        }
      }
    }
  }
}
```

插入数据

```
PUT secisland/my_type/1
{
  "group" : "fans",
  "user" : [
    {
      "first" : "John",
      "last" :   "Smith"
    },
    {
```

```
            "first" : "Alice",
            "last" :    "White"
        }
    ]
}
```

搜索 1

```
GET secisland/_search
{
  "query": {
    "nested": {
      "path": "user",
      "query": {
        "bool": {
          "must": [
            { "match": { "user.first": "Alice" }},
            { "match": { "user.last":    "Smith" }}
          ]
        }
      }
    }
  }
}
```

搜索 2

```
GET secisland/_search
{
  "query": {
    "nested": {
      "path": "user",
      "query": {
        "bool": {
          "must": [
            { "match": { "user.first": "Alice" }},
            { "match": { "user.last":    "White" }}
          ]
        }
      }
    }
  }
}
```

user 字段作为嵌套类型添加到索引中。搜索 1 匹配不到结果，因为 Alice 和 Smith 不在同一个嵌套对象中；搜索 2 匹配到搜索结果，因为 Alice 和 White 在同一个嵌套对象中。

C. 嵌套字段参数

dynamic	定义新的参数是否应该被动态加入到已经存在的对象中。接受 true（默认），false 和 strict。

inclue_in_all	设置所有嵌套对象属性的 include_in_all 值。嵌套文档没有他们自身的_all 字段，取而代之的是，值被添加到“根”文档的_all 字段中。
properties	嵌套对象的字段可以是任何数据类型，包括嵌套对象类型。新的属性可以被添加到已经存在的嵌套对象中。

表格 3.8 嵌套字段参数

3.2.9 地理点数据类型

地理点类型字段接受经纬度对，可以被用来：

查找一定范围内的地理点，这个范围可以是相对于一个中心点的固定距离，也可以是多边形或者地理散列单元。

通过地理位置或者相对于中心点的距离聚合文档。

整合距离到文档的相关性评分中。

通过距离对文档进行排序。

可以通过 4 种方式指定地理点：

指定字段类型为地理点数据类型

```
PUT secisland
{
  "mappings": {
    "my_type": {
      "properties": {
        "location": {
          "type": "geo_point"
        }
      }
    }
  }
}
```

有 4 种不同方式指定地理点。

1、PUT secisland/my_type/1

```
{
  "text": "Geo-point as an object",
  "location": {
    "lat": 41.12,
    "lon": -71.34
  }
}
```

地理点参数形如对象参数，拥有纬度和经度键值对。

2、PUT secisland/my_type/2

```
{
  "text": "Geo-point as a string",
  "location": "41.12,-71.34"
}
```

字符串地理点参数的格式为“纬度,经度”。

3、PUT secisland/my_type/3

```
{
```

```
    "text": "Geo-point as a geohash",
    "location": "drm3btev3e86"
}
```

地理散列地理点参数

```
4、PUT secisland/my_type/4
{
    "text": "Geo-point as an array",
    "location": [ -71.34, 41.12 ]
}
```

地理点数组参数，格式为“[经度,纬度]”。

重要信息：注意字符串地理点参数顺序为（纬度,经度），然而地理点数组参数的顺序为（经度,纬度）。

coerce	基于标准的-180:180/-90:90 坐标系统的经度和纬度值。接受 true 和 false（默认）。
doc_values	定义字段是否应该以列跨度的方式存储在磁盘上，以便用于排序，聚合或者脚本？接受 true（默认）或 false 参数。
geohash	定义地理点是否应该作为地理散列值在子字段.geohash 中被索引。默认值为 false，除非 geohash_prefix 参数值为 true。
geohash_precision	用于 geohash 和 geohash_prefix 选项的地理散列最大长度。
geohash_prefix	定义地理点是否应该作为添加前缀的地理散列来进行索引。默认值为 false。
ignore_malformed	如果是 true，畸形的地理点会被忽略。如果是 false（默认），畸形地理点会抛出异常并丢弃整个文档。
lat_lon	定义地理点是否应该在子字段.lat 和.lon 中被索引到。接受 true 和 false（默认）。
precision_step	控制每个经纬点被索引的额外索引词的数量。默认值为 16。与 lat_lon 参数的值无关。

表格 3.9 地理点字段参数

3.2.10 地理形态数据类型

3.2.11 地理形状数据类型

地理形状数据类型有利于索引和搜索任意地理形状，例如矩形和多边形。无论是数据被索引还是在查询执行的过程中，都可以使用地理形状数据类型在地理点的基础上包含地理形状。

利用地理形状查询（geo_shape）来查询文档，可以使用地理形状数据类型。

A. 映射选项

地理形状类型映射将 geo_json 几何对象映射成地理形状类型。为了启用，需要明确映射字段为地理形状类型。

选项	描述	默认
tree	引用的前缀树名：geohash 或 quadtree	geohash
precision	这个参数可以用来代替 tree_levels 来设置一个适当的 tree_levels 参数值。指定一个适当的精确度，Elasticsearch 会计算匹配精确度的最佳 tree_levels 值。这个值应该是一个数字，跟着一个可选的距离单元。可用的距离单元包括：in、	meters

	inch、yd、yard、mi、miled、km、kilometers、m、meters、cm、centimeters、mm、millimeters。	
tree_levels	前缀树使用的层的最大数量。可以用来控制形状表示的精度以及因此索引的索引词数量。默认是选取前缀树引用的默认值。因为这个参数需要一定程度的底层实现的理解，可以使用 precision 参数进行替代。然而，即使使用 precision 参数，Elasticsearch 本质上只会使用并通过映射接口返回 tree_levels。	50m
strategy	定义了如何在索引和搜索时表示形状。会影响可用的功能，所以建议让 Elasticsearch 自动设置这个参数。有两种可用的值：recursive 和 term。term 仅支持点类型（points_only 参数会自动设置为 true），recursive 支持所有的形状类型。	recursive
distance_error_pct	用于示意前缀树应该使用的精确值。默认是 0.025（2.5%），最大支持的值为 0.5。	0.025
orientation	定义了如何解读多边形/多边形集合顶点的顺序。这个参数定义两种坐标系统规则（右手或左手）中的一个，每一种坐标系统可以用三种方式进行指定。1.右手规则：right、ccw、counterclockwise，2.左手规则：left、cw、clockwise。默认方向（counterclockwise）遵循 OGC 标准，定义了外环顶点按照逆时针的方向内环顶点按顺时针方向。在映射中对地理形状字段设置这个参数，可以明确设置坐标列表中的顶点顺序。	ccw
points_only	设置这个选项为 true（默认为 false），只对点形状配置地理形状字段类型（不支持多个点）。	false

表格 3.10 地理形状数据类型映射选项

B. 前缀树

在索引中高效地表示形状，形状被转换到一系列表示为方格（通常被称为“栅格”）的散列用于实现一个前缀树。树的概念来自该前缀树使用多层网格，每层增加精度级别来表示陆地。这可以被看作是增加地图的缩放级别或图像的细节水平。

C. 空间策略

前缀树的实现基于空间策略用来分解提供的形状为近似方格。每个策略解决这些问题：

- 哪种类型的形状可以被索引。
- 形状可以用于哪种类型的查询操作。
- 每个字段是否可以保存多个形状。

提供这些策略实现（具有相应的功能）：

策略	支持的形状	支持的查询	多形状
recursive	所有	INTERSECTS、DISJOINT、WITHIN、CONTAINS	是
term	点	INTERSECTS	是

表格 3.11 空间策略

D. 准确性

地理形状不提供 100%的准确性，并且取决于匹配值，可能对确定的查询返回一些误判或漏判的结果。为了缓和这个问题，需要为 tree_levels 参数选择一个合适的值来适应相应的预期。

例如:

```
{
    "properties": {
        "location": {
            "type": "geo_shape",
            "tree": "quadtree",
            "precision": "1m"
        }
    }
}
```

这个映射将 location 字段映射到地理形状类型，使用 quad 前缀树并且精度为 1m。Elasticsearch 转换这个精度到 tree_levels 设置为 26。

E. 性能方面的考虑

Elasticsearch 使用前缀树中的路径作为索引和查询中的索引词。更高级别的精确值，会生成更多的索引词。计算索引词、加载到内存、保存到磁盘也需要额外的性能花销。

索引大小和合理水平的精确值的折中是 50m。

F. 输入结构

用于表示形状的 GeoJSON 格式作为输入：

GeoJSON 类型	Elasticsearch 类型	描述
Point	point	单独地理坐标点。
LineString	linestring	一个任意的线，给出两个或更多的点。
Polygon	polygon	一个封闭的多边形，第一个点和最后一个点必须匹配，需要 N+1 个顶点创建一个 N 多边形，最少需要 4 个顶点。
MultiPoint	multipoint	一组不连续的，但是可能相关的点。
MultiLineString	multilinestring	一组分离的线
MultiPolygon	multipolygon	一组分离的多边形
GeometryCollection	geometrycollection	一种类似于 multi*形状的 GeoJSON 形状，多种类型可以共存（例如，一个点和一条线）。
N/A	envelope	一个封闭的矩形，通过指定左上角和右下角来确定。
N/A	circle	一个圆，通过指定中心点和带单位的半径来确认，默认单位为 METERS。

表格 3.12 GeoJSON 格式

1. Point

Point 是一个单独的地理坐标点，比如当前建筑的位置或者智能手机地理定位接口提供的确切位置。

```
{
    "location" : {
        "type" : "point",
        "coordinates" : [-77.03653, 38.897676]
    }
}
```

2. LineString

通过两个或更多的一组位置定义。只指定两个点，linestring 会表示一条直线。指定更多的点，可以创建任意的线。

```
{
    "location" : {
        "type" : "linestring",
        "coordinates" : [[-77.03653, 38.897676], [-77.009051, 38.889939]]
    }
}
```

3. Polygon

通过一列地理点列表进行定义。每个列表（外环）中的第一个点和最后一个点必须相同（多边形必须是封闭的）。

```
{
    "location" : {
        "type" : "polygon",
        "coordinates" : [
            [ [100.0, 0.0], [101.0, 0.0], [101.0, 1.0], [100.0, 1.0], [100.0, 0.0] ]
        ]
    }
}
```

第一个数组表示多边形的外环边界，其他数组表示内部形状（“孔”）。

```
{
    "location" : {
        "type" : "polygon",
        "coordinates" : [
            [ [100.0, 0.0], [101.0, 0.0], [101.0, 1.0], [100.0, 1.0], [100.0, 0.0] ],
            [ [100.2, 0.2], [100.8, 0.2], [100.8, 0.8], [100.2, 0.8], [100.2, 0.2] ]
        ]
    }
}
```

注意事项：GeoJSON 不强制指定顶点的顺序，而在国际日期变更线和极点附近的多边形是极有可能造成混乱的。为了解决这种混乱，开放地理空间信息联盟（OGC）简单特征访问规范定义了这些顶点顺序：

- 外环：逆时针方向
- 内环/孔：顺时针方向

对于不跨越国际日期变更线的多边形，顶点的顺序对于 Elasticsearch 来说是没有关系的。Elasticsearch 完全按照 OGC 制定的规范请求顶点，否则，一个意外的多边形会被创建，并且返回不符合预期的查询/过滤结果。

orientation 参数可以在地理形状数据类型字段创建映射的时候进行定义，规定顶点的顺序。也可以在每个文档中进行重写：

```
{
    "location" : {
        "type" : "polygon",
        "orientation" : "clockwise",
```

```
            "coordinates" : [
                [ [-177.0, 10.0], [176.0, 15.0], [172.0, 0.0], [176.0, -15.0], [-177.0, -10.0],
[-177.0, 10.0] ],
                [ [178.2, 8.2], [-178.8, 8.2], [-180.8, -8.8], [178.2, 8.8] ]
            ]
        }
    }
```

4. MultiPoint

GeoJSON 点的列表。

```
    {
        "location" : {
            "type" : "multipoint",
            "coordinates" : [
                [102.0, 2.0], [103.0, 2.0]
            ]
        }
    }
```

5. MultiLineString

GeoJSON 线的列表。

```
    {
        "location" : {
            "type" : "multilinestring",
            "coordinates" : [
                [ [102.0, 2.0], [103.0, 2.0], [103.0, 3.0], [102.0, 3.0] ],
                [ [100.0, 0.0], [101.0, 0.0], [101.0, 1.0], [100.0, 1.0] ],
                [ [100.2, 0.2], [100.8, 0.2], [100.8, 0.8], [100.2, 0.8] ]
            ]
        }
    }
```

6. MultiPolygon

GeoJSON 多边形的列表。

```
    {
        "location" : {
            "type" : "multipolygon",
            "coordinates" : [
                [ [[102.0, 2.0], [103.0, 2.0], [103.0, 3.0], [102.0, 3.0], [102.0, 2.0]] ],

                [ [[100.0, 0.0], [101.0, 0.0], [101.0, 1.0], [100.0, 1.0], [100.0, 0.0]],
                  [[100.2, 0.2], [100.8, 0.2], [100.8, 0.8], [100.2, 0.8], [100.2, 0.2]] ]
            ]
        }
    }
```

7. GeometryCollection

GeoJSON 地理几何对象的集合。

```
{
    "location" : {
        "type": "geometrycollection",
        "geometries": [
            {
                "type": "point",
                "coordinates": [100.0, 0.0]
            },
            {
                "type": "linestring",
                "coordinates": [ [101.0, 0.0], [102.0, 1.0] ]
            }
        ]
    }
}
```

8. Envelope

包含矩形左上角和右下角的坐标值来表示矩形:

```
{
    "location" : {
        "type" : "envelope",
        "coordinates" : [ [-45.0, 45.0], [45.0, -45.0] ]
    }
}
```

9. Circle

包含中心点和半径：

```
{
    "location" : {
        "type" : "circle",
        "coordinates" : [-45.0, 45.0],
        "radius" : "100m"
    }
}
```

注意：radius 是必要字段。距离单位默认为米（METERS）。

G. 排序和取回形状索引

由于形状复杂的输入结构和索引表示，当前不能直接通过字段排序或取回形状。地理形状值只能通过_source 字段取回。

3.2.12 IPv4 数据类型

IPv4 字段本质上是一个长整型字段，接受 IPv4 地址并作为长整型值进行索引。

添加映射

```
PUT secisland
{
  "mappings": {
    "my_type": {
      "properties": {
```

```
            "ip_addr": {
              "type": "ip"
            }
          }
        }
      }
    }
```

插入数据

```
PUT secisland/my_type/1
{
  "ip_addr": "192.168.1.1"
}
```

boost	字段级索引加权，接受浮点型数字参数，默认为 1.0。
doc_values	定义字段是否应该以列跨度的方式存储在磁盘上，以便用于排序，聚合或者脚本？接受 true（默认）或 false 参数。
include_in_all	决定字段是否应该被包含在_all 字段中。接受 true 或 false 参数。如果索引被设置为 no 或者父对象字段设置 include_in_all 为 false，参数默认值为 false；其他情况下，默认值为 true。
index	决定字段是否可以被用户搜索。接受参数 not_analyzed（默认）以及 no。
null_value	接受一个 IPv4 值替换所有 null 值。默认为 null，意味着字段被作为缺失字段。
precision_step	控制索引的额外索引词的数量来使范围查询更快速。默认值为 16。
store	决定字段值是否应该被存储以及从_source 字段分别获取。接受参数 true 或 false（默认）。

表格 3.13 IP 字段参数

注解：IPv6 地址现在还不提供支持。

3.2.13 单词计数数据类型

单词计数型字段本质上是一个整数型字段，接受并分析字符串值，然后索引字符串中单词的个数。

例如：

添加映射

```
PUT secisland
{
  "mappings": {
    "my_type": {
      "properties": {
        "name": {
          "type": "string",
          "fields": {
            "length": {
              "type":       "token_count",
              "analyzer": "standard"
            }
          }
        }
```

```
        }
      }
    }
}
```

插入数据

```
PUT secisland/my_type/1
{ "name": "John Smith" }
```

注解：严格来说，单词计数类型计算位置增量而不是统计单词。这意味着即使分析器过滤掉一部分单词，它们也会被包含在计数中。

analyzer	必要字段，定义用来分析字符串值的分析器。
boost	字段级索引加权，接受浮点型数字参数，默认为 1.0。
doc_values	定义字段是否应该以列跨度的方式存储在磁盘上，以便用于排序，聚合或者脚本？接受 true（默认）或 false 参数。
include_in_all	决定字段是否应该被包含在_all 字段中。接受 true 或 false 参数。如果索引被设置为 no 或者父对象字段设置 include_in_all 为 false，参数默认值为 false；其他情况下，默认值为 true。
index	决定字段是否可以被用户搜索。接受参数 not_analyzed（默认）以及 no。
null_value	接受一个与字段相同类型的数字型值替换所有 null 值。默认为 null，意味着字段被作为缺失字段。
precision_step	控制索引的额外索引词的数量来使范围查询更快速。默认值为 32。
store	决定字段值是否应该被存储以及从_source 字段分别获取。接受参数 true 或 false（默认）。

表格 3.14 单词计数型字段参数

3.3 元字段

每个文档都有与之关联的元数据，比如_index，映射_type 和_id 元字段。当映射类型被创建的时候，可以自定义一些元字段的行为。

A. 标识元字段

_index	文档所属的索引。
_uid	包含_type 和_id 的混合字段。
_type	文档的映射类型。
_id	文档的 ID

表格 3.15 标识元字段

B. 文档来源元字段

_source	作为文档内容的原始 JSON。
_size	_source 元字段占用的字节数，通过 mapper-size 插件提供。

表格 3.16 文档来源元字段

C. 索引元字段

_all	索引所有字段的值
_field_names	文档中所有包含非空值的字段。
_timestamp	关联文章的时间戳，可以手动指定或者自动生成。
_ttl	定义文档被自动删除之前的存活时间。

表格 3.17 索引元字段

D. 路由元字段

_parent	用于在映射类型之间创建父子关系。
_routing	一个自定义的路由值，路由文档到一个特定的分片。

表格 3.18 路由元字段

E. 其他元字段

_meta	应用特定的元字段。

表格 3.19 其他元字段

3.3.1 _all 字段

_all 字段是一个特殊的包含全部内容的字段，在一个大字符串中关联所有其他字段的值，使用空格作为分隔符。可以被分析和索引但不会被存储。

使用_all 字段可以对文档的值进行搜索而不必知道包含所需值的字段名。当面对一个新的数据集的时候，_all 字段是非常有用的选项。

插入数据

```
PUT secisland/user/1
{
   "first_name":      "John",
   "last_name":       "Smith",
   "date_of_birth": "1970-10-24"
}
```

利用_all 字段进行搜索

```
GET secisland/_search
{
   "query": {
      "match": {
         "_all": "john smith 1970"
      }
   }
}
```

_all 字段包含的索引词：["john", "smith", "1970", "10", "24"]

注解：date_of_birth 字段作为日期型字段，会索引一个索引词 1970-10-24 00:00:00 UTC。但是，_all 字段将所有的值作为字符串，所以日期值作为三个字符串被索引："1970", "10", "24"。

_all 字段就是一个字符串类型字段，接受与字符串型字段相同的参数，包括 analyzer，index_options 和 store。

_all 字段关联字段值的时候，丢失了短字段（高相关性）和长字段（低相关性）之间的区别。当相关性是重要搜索条件的时候，应该明确指出查询字段。

_all 字段的使用需要额外的处理器周期，并且耗费更多的磁盘空间。如果不需要的话，可以完全禁用或者在每个字段的基础上自定义。

3.3.2 _field_names 字段

_field_names 字段索引文档中所有包含非空值的字段名称。_field_names 字段用于存在查询和缺失查询来查找指定字段拥有非空值的文档是否存在。

_field_name 字段的值可以用于查询，聚合以及脚本：

```
PUT secisland/my_type/1
```

```
{
  "title": "This is a document"
}

PUT secisland/my_type/1
{
  "title": "This is another document",
  "body": "This document has a body"
}

GET secisland/_search
{
  "query": {
    "terms": {
      "_field_names": [ "title" ]
    }
  },
  "aggs": {
    "Field names": {
      "terms": {
        "field": "_field_names",
        "size": 10
      }
    }
  },
  "script_fields": {
    "Field names": {
      "script": "doc['_field_names']"
    }
  }
}
```

3.3.3 _id 字段

每个被索引的文档都关联一个_type 字段和一个_id 字段。_id 字段没有索引，它的值可以从_uid 字段自动生成。

_id 字段的值可以在查询以及脚本中访问，但是在聚合或者排序的时候，要使用_uid 字段而不能用_id 字段。

在查询和脚本中使用_id 字段

```
GET secisland/_search
{
  "query": {
    "terms": {
      "_id": [ "1", "2" ]
    }
  },
```

```
    "script_fields": {
      "UID": {
        "script": "doc['_id']"
      }
    }
}
```

3.3.4 _index 字段

在多个索引中执行查询的时候，有时需要添加查询子句来关联特定的索引文档。_index 字段可以匹配包含某个文档的索引。在 term 或 terms 查询、聚合、脚本以及排序的时候，可以访问_index 字段的值。

注解：_index 是一个虚拟字段，不作为一个真实的字段添加到 Lucene 索引中。这意味着可以在 term 或 terms 查询（或任何重写 term 查询的查询，比如 match、query_string 或者 simple_query_string 查询）中使用_index 字段，但是不支持 prefix、wildcard、regexp 或 fuzzy 查询。

```
GET index_1,index_2/_search
{
  "query": {
    "terms": {
      "_index": ["index_1", "index_2"]
    }
  },
  "aggs": {
    "indices": {
      "terms": {
        "field": "_index",
        "size": 10
      }
    }
  },
  "sort": [
    {
      "_index": {
        "order": "asc"
      }
    }
  ],
  "script_fields": {
    "index_name": {
      "script": "doc['_index']"
    }
  }
}
```

3.3.5 _meta 字段

每个映射类型都可以拥有自定义的元数据。这些元数据对 Elasticsearch 来说毫无用处，但是可以用来存储应用程序的特定元数据：

```
PUT secisland
{
  "mappings": {
    "user": {
      "_meta": {
        "class": "MyApp::User",
        "version": {
          "min": "1.0",
          "max": "1.3"
        }
      }
    }
  }
}
```

3.3.6 _parent 字段

在同一个索引中通过创建映射类型可以在文档间建立父子关系。

创建映射

```
PUT secisland
{
  "mappings": {
    "my_parent": {},
    "my_child": {
      "_parent": {
        "type": "my_parent"
      }
    }
  }
}
```

插入父文档

```
PUT secisland/my_parent/1
{
  "text": "This is a parent document"
}
```

插入子文档，并指出父文档

```
PUT secisland/my_child/2?parent=1
{
  "text": "This is a child document"
}

PUT secisland/my_child/3?parent=1
{
```

```
  "text": "This is another child document"
}
```

A. 父子限制

父类型和子类型必须是不同的——父子关系不能建立在相同类型的文档之间。

_parent 的 type 设置只能指向一个当前不存在的类型。这意味着一个类型被创建之后就无法成为父类型。

父子文档必须索引在相同的分片上。parent 编号被用来作为子文档的路由值，确保子文档被索引到父文档所在的分片中。这意味着当获取、删除或更新子文档的时候，需要提供相同的 parent 值。

B. 整体序数

使用整体序数可以加快建立父子关系。分片发生任何改变之后，整体序数都需要进行重建。分片中存储的父编码值越多，为_parent 字段重建整体序数所花的时间就越长。

整体序数在默认情况下属于懒创建：刷新之后的第一次父子查询或聚合会触发整体序数的创建，这可能会给用户的使用引入一个明显的延迟。可以使用参数将整体序数的创建时间由查询触发改到刷新触发:

```
PUT secisland
{
  "mappings": {
    "my_parent": {},
    "my_child": {
      "_parent": {
        "type": "my_parent",
        "fielddata": {
          "loading": "eager_global_ordinals"
        }
      }
    }
  }
}
```

3.3.7 _routing 字段

文档在索引中利用下面的公式路由到特定的分片：

```
shard_num = hash(_routing) % num_primary_shards
```

_routing 字段的默认值使用的是文档的_id 字段。如果存在父文档，则使用文档的_parent 编号。

可以通过为每个文档指定一个自定义的路由值来实现自定义的路由方式:

```
PUT secisland/my_type/1?routing=user1
{
  "title": "This is a document"
}
```

这个文档使用 user1 作为路由值，而不是它的 ID，在获取、删除和更新文档的时候需要提供相同的路由值。

_routing 字段可以在查询、聚合、脚本以及排序的时候访问:

```
GET secisland/_search
{
```

```
  "query": {
    "terms": {
      "_routing": [ "user1" ]
    }
  },
  "aggs": {
    "Routing values": {
      "terms": {
        "field": "_routing",
        "size": 10
      }
    }
  },
  "sort": [
    {
      "_routing": {
        "order": "desc"
      }
    }
  ],
  "script_fields": {
    "Routing value": {
      "script": "doc['_routing']"
    }
  }
}
```

A. 利用自定义路由进行搜索

自定义路由可以降低搜索压力。搜索请求可以仅仅发送到匹配指定路由值的分片而不是广播到所有分片。

```
GET secisland/_search?routing=user1,user2
{
  "query": {
    "match": {
      "title": "document"
    }
  }
}
```

搜索请求仅在关联路由值为 user1 和 user2 的分片上执行。

B. 使路由值成为必选项

使用自定义路由索引、获取、删除或更新文档时，提供路由值是很重要的。

忘记路由值会导致文档被一个以上的分片索引。作为保障，_routing 字段可以被设置，使自定义路由值成为所有 CRUD 操作的必选项:

```
PUT secisland
{
```

```
  "mappings": {
    "my_type": {
      "_routing": {
        "required": true
      }
    }
  }
}
```

C. 自定义路由下的唯一编码

当索引指定了自定义路由的文档时，不能保障所有分片中文档_id 的唯一性。事实上，拥有相同_id 的文档会根据不同的路由存储在不同的分片中。

只能依靠用户来确保编码的唯一性。

3.3.8 _source 字段

_source 字段包含索引时原始的 JSON 文档内容，字段本身不建立索引（因此无法搜索）但是会被存储，所以当执行获取请求的时候可以返回_source 字段。

虽然很方便，但是_source 字段确实会对索引产生存储开销。因此，可以禁用_source 字段：

```
PUT tweets
{
  "mappings": {
    "tweet": {
      "_source": {
        "enabled": false
      }
    }
  }
}
```

警告:

如果_source 字段被禁用，会造成大量的功能无法使用：

更新接口

高亮显示功能

重建索引的功能，不论是修改映射或分析，还是升级索引到一个新版本。

通过查看索引时的原始文档对查询或聚合进行调试的功能。

自动修复索引的功能。

提示：如果磁盘空间是个问题，可以提高压缩等级来实现节约存储空间。可以用包含/排除字段的特性在保存之前减少_source 字段的内容。

警告：

从_source 字段中移除内容相当于精简版的禁用功能，尤其是无法重建文档索引。

includes/excludes 参数（可以使用通配符）使用示例：

```
PUT logs
{
  "mappings": {
    "event": {
      "_source": {
        "includes": [
          "*.count",
          "meta.*"
        ],
        "excludes": [
          "meta.description",
          "meta.other.*"
        ]
      }
    }
  }
}
```

移除的字段不会被存储在_source 字段中，但我们仍然可以搜索这些字段。

3.3.9 _type 字段

每个索引的文档都包含_type 和_id 字段，索引_type 字段的目的是通过类型名加快搜索进度。

_type 字段的值可以在查询、聚合、脚本以及排序时访问:

```
PUT secisland/type_1/1
{
  "text": "Document with type 1"
}

PUT secisland/type_2/2
{
  "text": "Document with type 2"
}

GET secisland/_search/type_*
{
  "query": {
    "terms": {
      "_type": [ "type_1", "type_2" ]
```

```
        }
      },
      "aggs": {
        "types": {
          "terms": {
            "field": "_type",
            "size": 10
          }
        }
      },
      "sort": [
        {
          "_type": {
            "order": "desc"
          }
        }
      ],
      "script_fields": {
        "type": {
          "script": "doc['_type']"
        }
      }
    }
```

3.3.10 _uid 字段

每个索引的文档都包含_type 和_id 字段，这两个值结合为{type}#{id}并且作为_uid 字段被索引。

_uid 字段的值可以在查询、聚合、脚本以及排序时访问:

```
PUT secisland/my_type/1
{
  "text": "Document with ID 1"
}

PUT secisland/my_type/2
{
  "text": "Document with ID 2"
}

GET secisland/_search
{
  "query": {
    "terms": {
      "_uid": [ "my_type#1", "my_type#2" ]
    }
  },
```

```
  "aggs": {
    "UIDs": {
      "terms": {
        "field": "_uid",
        "size": 10
      }
    }
  },
  "sort": [
    {
      "_uid": {
        "order": "desc"
      }
    }
  ],
  "script_fields": {
    "UID": {
      "script": "doc['_uid']"
    }
  }
}
```

3.4 映射参数

用于字段映射的映射参数，对部分或全部字段数据类型是通用的。

3.4.1 analyzer 参数

可分词的字符串型字段的值通过一个分析器将字符串转换为一连串的索引词。例如，字符串“The quick Brown Foxes.”，取决于所使用的分析器，可能被分词为“quick,brown,fox”。这些是字段创建索引时的实际索引词，可以使大段文本中的特定单词的搜索更加高效。

分析的过程不仅发生在索引阶段，也发生在查询阶段：查询字符串需要通过相同（或相似）的分析器进行分析。所以，查询中的索引词和索引中索引词具有相同的格式。

Elasticsearch 附带一系列预定义的分析器，不需要更多的配置就可以使用。 也附带一些字符过滤器、分词器、词元过滤器，可以结合起来为每个索引配置自定义分析器。

每个查询、每个字段或每个索引都可以指定分析器。在创建索引时，Elasticsearch 会以这个顺序查找分析器：

- 在字段映射中定义的分析器。
- 在索引设置中名为 default 的分析器。
- 标准分析器。
- 在查询时，有更多的层次。
- 在全文查询中定义的分析器。
- 在字段映射中定义的搜索分析器。
- 在字段映射中定义的分析器。
- 在索引设置中名为 default_search 的分析器。
- 在索引设置中名为 default 的分析器。
- 标准分析器。

为特定字段指定分词器最简单的方式是在字段映射中定义：

```
PUT secisland
{
  "mappings": {
    "my_type": {
      "properties": {
        "text": {
          "type": "string",
          "fields": {
            "english": {
              "type":      "string",
              "analyzer": "english"
            }
          }
        }
      }
    }
  }
}
```

search_quote_analyzer 参数设置可以对短语指定一个分析器，当处理禁用短语连接词的查询时特别有用。

为了禁用短语连接词，字段需要利用三个分析器设置：

1. analyzer 设置，用于索引包括连接词在内的所有索引词。
2. search_analyzer 设置，用于移除连接词的非短语查询。
3. search_quote_analyzer 设置，用于包括连接词在内的短语查询。

```
PUT /secisland
{
    "settings":{
        "analysis":{
            "analyzer":{
                "my_analyzer":{
                    "type":"custom",
                    "tokenizer":"standard",
                    "filter":[
                        "lowercase"
                    ]
                },
                "my_stop_analyzer":{
                    "type":"custom",
                    "tokenizer":"standard",
                    "filter":[
                        "lowercase",
                        "english_stop"
                    ]
                }
```

```
                },
                "filter":{
                    "english_stop":{
                        "type":"stop",
                        "stopwords":"_english_"
                    }
                }
            }
        },
        "mappings":{
            "my_type":{
                "properties":{
                    "title": {
                        "type":"string",
                        "analyzer":"my_analyzer",
                        "search_analyzer":"my_stop_analyzer",
                        "search_quote_analyzer":"my_analyzer"
                      }
                    }
                }
            }
        }
}
```

插入数据

```
PUT secisland/my_type/1
{
    "title":"The Quick Brown Fox"
}

PUT secisland/my_type/2
{
    "title":"A Quick Brown Fox"
}
```

进行短语查询

```
GET secisland/my_type/_search
{
    "query":{
        "query_string":{
            "query":"\"the quick brown fox\""
        }
    }
}
```

当查询被包含在引号中时，表示这是一个短语查询，因此 search_quote_analyzer 生效并确保查询中包含连接词。如果没有 search_quote_analyzer，短语查询不可能被精准匹配：因为短语查询中移除了连接词，所以两个文档都会被匹配到。

3.4.2 boost 参数

在索引的时候，通过 boost 参数可以对一个字段进行加权：对相关性得分计数更多：

```
PUT secisland
{
  "mappings": {
    "my_type": {
      "properties": {
        "title": {
          "type": "string",
          "boost": 2
        },
        "content": {
          "type": "string"
        }
      }
    }
  }
}
```

匹配 title 字段会有两倍的加权，而 content 字段的 boost 参数，是默认值 1.0。

需要注意的是，title 字段通常会比 content 字段的长度短。默认的相关性计算需要考虑字段长度，因此短字段 title 会有一个更高的加权。

> ***警告：***
>
> ***索引时最好不要进行加权，主要有这几个原因：***
>
> ***除非重索引所有的文档，索引加权值不会发生改变。***
>
> ***每个查询都支持查询加权，会产生同样的效果。不同的地方在于不需要重索引就可以调整加权值。***
>
> ***索引加权值作为 norm 的一部分，只有一个字节。这降低了字段长度归一化因子的分辨率，会导致低质量的相关性计算。***

索引加权的唯一优势在于作用于_all 字段。这意味着，当查询_all 字段的时候，源于 title 字段的词比源于 content 字段的词有更高的分数。这个功能是以计算成本为代价的：当索引加权被使用时，查询_all 字段会变得更加缓慢。

3.4.3 coerce 参数

数据不都是干净的。一个数字取决于如何产生，可能通过 JSON 体中确定的 JSON 数值进行提供，比如，5；也可能通过一个字符串进行提供，比如“5”。或者，整型数据可能被提供浮点型数据，例如 5.0 或者“5.0”。

强制尝试清理脏值来匹配字段的数据类型。例如：

字符串会被强制转换为数字。
浮点型数据会被截取为整型数据。
经纬地理点数据会归一化到标准-180:180 / -90:90 坐标系统。
举例：

```
PUT secisland
{
  "mappings": {
    "my_type": {
      "properties": {
        "number_one": {
          "type": "integer"
        },
        "number_two": {
          "type": "integer",
          "coerce": false
        }
      }
    }
  }
}

PUT secisland/my_type/1
{
  "number_one": "10"
}

PUT secisland/my_type/2
{
  "number_two": "10"
}
```

number_one 字段会包含整数 10。
因为 number_two 字段的强制类型转换被禁用，所以文档 2 会被丢弃。
同一个索引中相同名称的字段可以拥有不同的 coerce 设置。可以利用 PUT 映射接口来更新已存在字段的 coerce 值。
默认索引级别
index.mapping.coerce 设置可以在索引级别上对所有映射类型整体禁用强制类型转换:

```
PUT secisland
{
  "settings": {
    "index.mapping.coerce": false
  },
  "mappings": {
    "my_type": {
      "properties": {
```

```
            "number_one": {
              "type": "integer"
            },
            "number_two": {
              "type": "integer",
              "coerce": true
            }
          }
        }
      }
    }

    PUT secisland/my_type/1
    { "number_one": "10" }

    PUT secisland/my_type/2
    { "number_two": "10" }
```

number_one 字段继承索引级别的强制类型转换设置，number_two 字段覆盖索引级别的设置来启用强制类型转换。

3.4.4 copy_to 参数

利用 copy_to 参数可以创建自定义的_all 字段。换句话说，多个字段的值可以被复制到一组字段，然后可以作为单个字段进行查询。例如：

```
PUT /secisland
{
  "mappings": {
    "my_type": {
      "properties": {
        "first_name": {
          "type": "string",
          "copy_to": "full_name"
        },
        "last_name": {
          "type": "string",
          "copy_to": "full_name"
        },
        "full_name": {
          "type": "string"
        }
      }
    }
  }
}
```

```
PUT /secisland/my_type/1
{
  "first_name": "John",
  "last_name": "Smith"
}

GET /secisland/_search
{
  "query": {
    "match": {
      "full_name": {
        "query": "John Smith",
        "operator": "and"
      }
    }
  }
}
```

first_name 和 last_name 字段的值会被复制到 full_name 字段中。

first_name 和 last_name 字段仍然可以单独进行查询，而且 full_name 可以用来同时查询两个字段的内容。

一些重要的点：

字段的值是复制过来的，并不是分组（分析过程的结果）。

原始的_source 字段不会被修改来展示复制的值。

利用"copy_to": ["field_1", "field_2"]，可以将相同的值复制到多个字段。

3.4.5 doc_values 参数

大多数的字段默认被索引，使它们可以被搜索到。反向索引允许查询请求在唯一的索引词有序列表中寻找搜索的索引词，找到之后立即访问包含索引词的文档列表。

排序，聚合以及在脚本中访问字段值需要一个不同的数据访问模式。我们需要能够查找文档并在一个字段中寻找存在的索引词。

文档值（Doc Values）是磁盘上的数据结构，在文档索引阶段创建，使上面这种数据访问模式成为可能。doc_values 支持几乎所有字段类型。

所有的字段默认包含在文档值中。如果确定一个字段不需要排序、聚合或者在脚本中访问字段值，可以禁用文档值来节省存储空间：

```
PUT secisland
{
  "mappings": {
    "my_type": {
      "properties": {
        "status_code": {
          "type":          "string",
          "index":         "not_analyzed"
        },
        "session_id": {
          "type":          "string",
```

```
                "index":          "not_analyzed",
                "doc_values": false
            }
        }
    }
  }
}
```

status_code 字段默认拥有文档值。

session_id 字段值禁用了文档值，但仍然可以被查询。

贴士：在同一个索引中同名的字段可以设置不同的 doc_values。可以通过 PUT 映射接口对存在的字段禁用（设置为 false）文档值。

3.4.6 dynamic 参数

默认，字段可以被动态地添加到一个文档中，或者添加到文档内部对象中，仅仅通过索引一个包含新字段的文档。例如：

dynamic 设置控制新字段是否可以被动态添加。接受三种设置：

true	新检测到的字段会被添加到映射中（默认）。
false	新检测到的字段会被忽略。新字段必须明确添加。
strict	如果新字段被检测到，一个异常会被抛出而且文档会被丢弃。

表格 3.20 dynamic 参数设置

dynamic 可以在映射类型级别以及每个内部对象被设置。内部对象从它们的父对象或映射类型中继承设置值。例如：

```
PUT secisland
{
  "mappings": {
    "my_type": {
      "dynamic": false,
      "properties": {
        "user": {
          "properties": {
            "name": {
              "type": "string"
            },
            "social_networks": {
              "dynamic": true,
              "properties": {}
            }
          }
        }
      }
    }
  }
}
```

动态映射在类型级别上被禁用，所以新的顶级字段不会被自动添加。

user 对象继承了类型级别的设置。

user.social_networks 对象启用动态映射，所以新字段可以被添加到这个内部对象。

3.4.7 enabled 参数

Elasticsearch 尝试索引给出的所有字段，但是一些情况下仅仅需要存储字段而不进行索引。

enabled 设置，仅可以被应用于映射类型和对象字段，导致 Elasticsearch 跳过字段内容的分解。JSON 仍然可以从_source 字段取回，但是不能搜索或以其他方式存储:

```
PUT secisland
{
  "mappings": {
    "session": {
      "properties": {
        "user_id": {
          "type":    "string",
          "index": "not_analyzed"
        },
        "last_updated": {
          "type": "date"
        },
        "session_data": {
          "enabled": false
        }
      }
    }
  }
}

PUT secisland/session/session_1
{
  "user_id": "kimchy",
  "session_data": {
    "arbitrary_object": {
      "some_array": [ "foo", "bar", { "baz": 2 } ]
    }
  },
  "last_updated": "2015-12-06T18:20:22"
}

PUT secisland/session/session_2
{
  "user_id": "jpountz",
  "session_data": "none",
  "last_updated": "2015-12-06T18:22:13"
}
```

session_data 字段设置了 enabled 参数为 false，任意的数据都可以被存储在 session_data 字段，不过 session_data 会忽略非 JSON 对象的值。

整个映射类型都可以被禁用，在这种情况下文档被存储在_source 字段，意味着它可以被取回但是没有任何内容以任何方式被索引：

```
PUT secisland
{
  "mappings": {
    "session": {
      "enabled": false
    }
  }
}

PUT secisland/session/session_1
{
  "user_id": "kimchy",
  "session_data": {
    "arbitrary_object": {
      "some_array": [ "foo", "bar", { "baz": 2 } ]
    }
  },
  "last_updated": "2015-12-06T18:20:22"
}
```

整个 session 映射类型被禁用，存储的文档可以被取回但是没有字段被添加到里面。

3.4.8 fielddata 参数

大多数的字段默认被索引，使它们可以被搜索到。反向索引允许查询请求在唯一的索引词有序列表中寻找搜索的索引词，找到之后立即访问包含索引词的文档列表。

排序，聚合以及在脚本中访问字段值需要一个不同的数据访问模式。我们需要能够查找文档并在一个字段中寻找存在的索引词。

大多数的字段可以使用索引时存储在磁盘上的 doc_values 值来实现这种数据访问模式，但是分析过的字符串字段不支持 doc_values。

相对的，分词的字符串在查询时利用一个称为字段数据（fielddata）的数据结构。这个数据结构在字段被用于聚合、排序或者脚本访问的第一时间创建。从磁盘上为每个分片读取整个反向索引，倒置索引词和文档的对应关系，然后将结果存储到内存中 Java 虚拟机堆内存中。

加载字段数据是一个昂贵的过程，一但被加载，就会在分片的生命周期内驻留在内存中。

警告：

字段数据会消耗大量的堆内存空间，特别是加载高基数分词字符串字段。大多数时候，在分词字符串字段上进行排序或聚合没有意义（除了显著分组聚合）。经常考虑字段是否不可分词（可以使用 doc_values）可以更好地适应应用场景。

贴士：同一个索引中的同名字段必须拥有相同的 fielddata.*设置。

- **fielddata.format**

对于分词字符串字段，字段数据 format 参数用来控制字段数据是否应该被启用。接受参数：disabled 和 paged_bytes（启用，默认值）。禁用字段数据加载，可以使用下面的设置：

```
PUT secisland
{
  "mappings": {
    "my_type": {
      "properties": {
        "text": {
          "type": "string",
          "fielddata": {
            "format": "disabled"
          }
        }
      }
    }
  }
}
```

通过上面的配置，test 字段不能被用于排序、聚合或者脚本引用。

- **fielddata.loading**

这是针对每个字段的设置，控制什么时候字段数据会被加载到内存。接受三个参数：

lazy	字段数据仅在需要的时候加载到内存中（默认）。
eager	在新的搜索分片对搜索可见之前，字段数据就被加载到内存中。如果用户搜索请求一个大容量分片，比起触发被动加载，这么做可以减少请求延迟。
eager_global_ordinals	仅加载必须的部分到内存中。加载之后，Elasticsearch 创建整体序数数据结构来组成一个所有索引词的列表。默认情况下，整体序数是被动创建的。如果字段有非常高的基数，在这种情况下就可以使用主动加载。

表格 3.21 字段数据加载参数

- **fielddata.filter**

字段数据过滤可以用来减少加载到内存中的索引词的数量，这么做可以减少内存使用量。索引词可以通过频率或正则表达式以及两者的配合进行过滤:

- 通过频率过滤

通过频率过滤可以加载部分索引词，这些索引词的频率落在最小值和最大值之间。频率可以表示为确切的数字（当数值比 1.0 大）或者作为一个百分比的数（例如 0.01 是 1%，1.0 是 100%）。每个分片都会计算频率。频率基于有字段值的文档数量，相对于分片中所有的文档。

可以通过 min_segment_size 参数指定分片应该包含的最少文件量来完全排除小容量分片：

```
PUT secisland
{
  "mappings": {
    "my_type": {
```

```
        "properties": {
          "tag": {
            "type": "string",
            "fielddata": {
              "filter": {
                "frequency": {
                  "min": 0.001,
                  "max": 0.1,
                  "min_segment_size": 500
                }
              }
            }
          }
        }
      }
    }
```

- 通过正则表达式过滤

索引词也可以通过正则表达式进行过滤：只有匹配正则表达式的值会被加载。注意：正则表达式被应用于字段中的每个索引词，而不是整个字段值。

```
PUT secisland
{
  "mappings": {
    "my_type": {
      "properties": {
        "tweet": {
          "type": "string",
          "analyzer": "whitespace",
          "fielddata": {
            "filter": {
              "regex": {
                "pattern": "^#.*"
              }
            }
          }
        }
      }
    }
  }
}
```

已有的字段映射可以更新这些过滤而且会影响分片下一次加载字段数据。利用清理缓存接口来用新的过滤条件重载字段数据。

3.4.9 format 参数

在 JSON 格式文档中，日期被表示为字符串。Elasticsearch 利用一系列的预先设定的格式来识别和分析这些字符串成为一个长整型的值，代表世界标准时间的毫秒数。

除了内置格式之外，也可以使用通俗的 yyyy/MM/dd 语法来指定自定义格式：

```
PUT secisland
{
  "mappings": {
    "my_type": {
      "properties": {
        "date": {
          "type":     "date",
          "format": "yyyy-MM-dd"
        }
      }
    }
  }
}
```

许多支持日期值的接口也支持日期匹配表达式，例如 now-1m/d——当前的时间减去一个月，并且四舍五入到最近的一天。

贴士：在同一个索引中的同名字段必须有相同的格式设置。已有的字段可以利用 PUT 映射接口来更新格式的值。

内置格式

大多数的日期都有严格的对应日期的格式，这意味着年份、月份以及一个星期的某一天都必须用零补全格式来使日期有效。日期格式为 45/11/1 会被认为是无效的，需要指定完整的日期，例如 2045/11/01。所以需要指定格式为 strict_date_optional_time 而不是 date_optional_time。

下面列举支持的所有国际标准化组织（ISO）规定的日期格式：

- epoch_millis

从纪元开始的毫秒数。注意，这个时间戳的最大长度为 13 个字符，所以仅支持 1653 年到 2286 年之间的日期。在这种格式下你应该使用不同的时间格式来表示日期。

- epoch_second

从纪元开始的秒数。注意，这个时间戳最大长度为 10 个字符，所以仅支持 1653 年到 2286 年之间的日期。在这种格式下你应该使用不同的时间格式来表示日期。

- date_optional_time 或者 strict_date_optional_time

一个通用的 ISO 日期时间分析器，日期（date）是必须的，时间（time）是可选的。

- basic_date

一个完整的日期基本格式表示为四位数年份、两位数月份以及两位数当前月份的具体天数：yyyyMMdd。

- basic_date_time

融合基础日期和时间的基本格式，利用 T 来分隔：yyyyMMdd’T’HHmmss.SSSZ。

- basic_ordinal_date

全序日期的格式，表示为四位数年份和三位数当前年份的具体天数：yyyyDDD。

- basic_ordinal_date_time

融合全序日期和时间的格式，利用 T 来分隔：yyyyDDD’T’HHmmss.SSSZ。

- basic_ordinal_date_time_no_millis

忽略毫秒数的全序日期和时间:yyyyDDD’T’HHmmssZ。

- basic_time

基础时间格式，表示为两位当日的小时数，两位小时里的分钟数，两位分钟里的秒数，三位毫秒数以及时间区间偏移：HHmmss.SSSZ。

- basic_time_no_millis

忽略毫秒数的基础时间格式：HHmmssZ。

- basic_t_time

基础时区时间格式，表示为两位当日的小时数，两位小时里的分钟数，两位分钟里的秒数，三位毫秒数以及时间区间偏移前缀 T：’T’HHmmss.SSSZ。

- basic_t_time_no_millis

忽略毫秒数的基础时区时间格式：’T’HHmmssZ。

- basic_week_date 或 strict_basic_week_date

全日期周年格式，表示为四位周年数，两位周年里的周数以及一位本周的日数：xxxx’W’wwe。

- basic_week_date_time 或 strict_basic_week_date_time

融合周年和时间的格式，利用 T 来分隔：xxxx’W’wwe’T’HHmmss.SSSZ。

- basic_week_date_time_no_millis 或 strict_basic_week_date_time_no_millis

忽略毫秒数的周年时间格式：xxxx’W’wwe’T’HHmmssZ。

- date 或 strict_date

日期格式，表示为四位数年份，两位数月份以及两位数当前月经过的天数：yyyy-MM-dd。

- date_hour 或 strict_date_hour

融合日期格式和两位当日小时数：yyyy-MM-dd’T’HH。

- date_hour_minute 或 strict_date_hour_minute

融合日期格式和两位小时数，两位分钟数：yyyy-MM-dd’T’HH:mm。

- date_hour_minute_second 或 strict_date_hour_minute_second

融合日期格式和两位小时数，两位分钟数以及两位秒数：yyyy-MM-dd’T’HH:mm:ss。

- date_hour_minute_second_fraction 或 strict_date_hour_minute_second_fraction

融合日期格式和两位小时数，两位分钟数，两位秒数以及三位毫秒部分：yyyy-MM-dd’T’HH:mm:ss.SSS。

- date_hour_minute_second_millis or strict_date_hour_minute_second_millis

融合日期格式和两位小时数，两位分钟数，两位秒数以及三位毫秒部分：yyyy-MM-dd’T’HH:mm:ss.SSS。

- date_time 或 strict_date_time

融合全日期和时间，利用 T 分隔：yyyy-MM-dd’T’HH:mm:ss.SSSZZ。

- date_time_no_millis 或 strict_date_time_no_millis

融合日期和时间，忽略毫秒数，利用 T 分隔：yyyy-MM-dd’T’HH:mm:ssZZ。

- hour 或 strict_hour

当日两位小时数的格式：HH。

- hour_minute 或 strict_hour_minute

两位小时数，两位分钟数：HH:mm。

- hour_minute_second 或 strict_hour_minute_second

两位小时数，两位分钟数，两位秒数：HH:mm:ss。

- hour_minute_second_fraction 或 strict_hour_minute_second_fraction

两位小时数，两位分钟数，两位秒数，三位毫秒数：HH:mm:ss.SSS。

- hour_minute_second_millis 或 strict_hour_minute_second_millis

两位小时数，两位分钟数，两位秒数，三位毫秒数：HH:mm:ss.SSS。

- ordinal_date 或 strict_ordinal_date

全序日期格式，表示为四位数年份和三位当年已过的天数：yyyy-DDD。

- ordinal_date_time 或 strict_ordinal_date_time

融合全序日期和时间：yyyy-DDD’T’HH:mm:ss.SSSZZ。

- ordinal_date_time_no_millis 或 strict_ordinal_date_time_no_millis

忽略毫秒数的全序日期和时间：yyyy-DDD’T’HH:mm:ssZZ

- time 或 strict_time

时间格式，表示为两位当日的小时数，两位小时里的分钟数，两位分钟里的秒数，三位毫秒数以及时间区间偏移：HH:mm:ss.SSSZZ。

- time_no_millis 或 strict_time_no_mills

忽略毫秒数的时间格式：HH:mm:ssZZ。

- t_time 或 strict_t_time

时区时间格式，表示为两位当日的小时数，两位小时里的分钟数，两位分钟里的秒数，三位毫秒数以及时间区间偏移前缀 T：’T’HH:mm:ss.SSSZZ。

- t_time_no_millis 或 strict_t_time_no_millis

忽略毫秒数的时区时间格式：’T’HH:mm:ssZZ。

- week_date 或 strict_week_date

全周年日期表示为四位周年数，两位当年的周数，以及一位本周的日数：xxxx-‘W’ww-e。

- week_date_time 或 strict_week_date_time

融合全周年日期和时间，利用 T 分隔：xxxx-‘W’ww-e’T’HH:mm:ss.SSSZZ。

- weekyear 或 strict_weekyear

周年格式，表示为四位周年数：xxxx。

- weekyear_week 或 strict_weekyear_week

融合周年格式和两位周年的周数：xxxx-‘W’ww。

- weekyear_week_day 或 strict_weekyear_week_day

融合周年格式和两位周年的周数，以及一位本周的日数：xxxx-‘W’ww-e。

- year 或 strict_year

四位年数的格式：yyyy。

- year_month 或 strict_year_month

四位年数和两位月数：yyyy-MM。

- year_month_day 或 strict_year_month_day

四位年数，两位月数，两位日数：yyyy-MM-dd。

3.4.10 geohash 参数

地理散列是把地球划分为网格的经纬度编码。网格的每个单元格被表现为地理散列字符串。每个单元格可以被进一步划分为用更长字符串表示的更小单元格。所以地理散列越长，单元格越小（精确度越高）。

因为地理散列就是个字符串，可以像其他字符串一样被存储在一个反向索引中，这使得索引地理散列效率很高。

如果启用地理散列选项，一个地理散列类型的“子字段”会被索引，例如.geohash。地理散列的长度通过 geohash_precision 参数控制。

如果启用 geohash_prefix 选项，geohash 选项会被自动启用。

例如：

```
PUT secisland
{
  "mappings": {
    "my_type": {
      "properties": {
        "location": {
          "type": "geo_point",
          "geohash": true
        }
      }
    }
  }
}

PUT secisland/my_type/1
{
  "location": {
    "lat": 41.12,
    "lon": -71.34
  }
}
```

每个地理点都会索引一个 location.geohash 字段。

地理散列可以从 doc_values 中取回。

3.4.11 geohash_precision 参数

当启用地理散列选项的时候，地理精度设置可以控制地理散列的长度；当地理散列前缀选项启用的时候，地理精度设置可以控制地理散列的最大长度。

接受参数：

介于 1 到 12（默认）之间的数字，代表地理散列的长度。

一段距离，例如 1km。

如果指定了一段距离，会被作为最小的地理散列长度提供请求分辨率。

例如：

```
PUT secisland
{
  "mappings": {
    "my_type": {
      "properties": {
        "location": {
          "type": "geo_point",
          "geohash_prefix": true,
          "geohash_precision": 6
        }
      }
```

```
        }
      }
    }

    PUT secisland/my_type/1
    {
      "location": {
        "lat": 41.12,
        "lon": -71.34
      }
    }
```

一个 geohash_precision 值为 6 的地理散列单元格的面积近似为 1.26km x 0.6km。

3.4.12 geohash_prefix 参数

当启用地理散列选项，以一定的精度将地理散列作为经纬度点索引的时候，geohash_prefix 选项也会索引到所有的地理散列单元格中。

例如，一个 drm3btev3e86 地理散列会索引下列所有的索引词：[d, dr, drm, drm3, drm3b, drm3bt, drm3bte, drm3btev, drm3btev3, drm3btev3e, drm3btev3e8, drm3btev3e86]。

在 geohash_cell 查询中可以使用地理散列前缀来查找特定地理散列中的点，或者它的相邻点:

```
    PUT secisland
    {
      "mappings": {
        "my_type": {
          "properties": {
            "location": {
              "type": "geo_point",
              "geohash_prefix": true,
              "geohash_precision": 6
            }
          }
        }
      }
    }

    PUT secisland/my_type/1
    {
      "location": {
        "lat": 41.12,
        "lon": -71.34
      }
    }

    GET secisland/_search?fielddata_fields=location.geohash
    {
```

```
    "query": {
      "geohash_cell": {
        "location": {
          "lat": 41.02,
          "lon": -71.48
        },
        "precision": 4,
        "neighbors": true
      }
    }
}
```

3.4.13 ignore_above 参数

比 ignore_above 设置长的字符串不会被分词或者索引。这主要用于不分词的字符串字段，这些字符串通常用来过滤、聚合以及排序。这些都是结构化的字段，让这些字段索引非常长的索引词通常是不明智的。

```
PUT secisland
{
  "mappings": {
    "my_type": {
      "properties": {
        "message": {
          "type": "string",
          "index": "not_analyzed",
          "ignore_above": 20
        }
      }
    }
  }
}
```

message 字段会忽略任何长度超过 20 个字符的字符串。

```
PUT secisland/my_type/1
{
  "message": "Syntax error"
}
```

这篇文档会被成功索引。

```
PUT secisland/my_type/2
{
  "message": "Syntax error with some long stacktrace"
}
```

这篇文档会被索引，但是不会索引 message 字段的值。

这个选项也可以用于防止 Lucene 限制索引词的字节长度为 32766。

注意：

ignore_above 的值是字符数，而 Lucene 是字节计数。如果使用 UTF-8 文本与许多非 ASCII 字符，可能需要设置限制为 32766 / 3 = 10922。因为 UTF-8 字符会占用最多 3 个字节。

3.4.14 ignore_malformed 参数

通常对于收到的数据没有做更多的控制。一个用户发送的 login 字段可能是日期，另一个发送的 login 字段可能是电子邮件地址。

默认情况下，在试着索引错误的数据类型的时候会抛出异常并拒绝整个文档。如果 ignore_malformed 参数被设置为 true，异常会被忽略。错误字段不会被索引，但文档中的其他字段会正常处理。

```
PUT secisland
{
  "mappings": {
    "my_type": {
      "properties": {
        "number_one": {
          "type": "integer"
        },
        "number_two": {
          "type": "integer",
          "ignore_malformed": true
        }
      }
    }
  }
}

PUT secisland/my_type/1
{
  "text":          "Some text value",
  "number_one": "foo"
}
```

这篇文档会被拒绝索引。

```
PUT secisland/my_type/2
{
  "text":          "Some text value",
  "number_two": "foo"
}
```

这篇文档会仅索引 text 字段。

index-level 默认

index.mapping.ignore_malformed 设置可以用在索引级别上，在所有映射类型中忽略错误内容：

```
PUT secisland
{
  "settings": {
    "index.mapping.ignore_malformed": true
  },
  "mappings": {
```

```
        "my_type": {
          "properties": {
            "number_one": {
              "type": "byte"
            },
            "number_two": {
              "type": "integer",
              "ignore_malformed": false
            }
          }
        }
      }
```

3.4.15 include_in_all 参数

include_in_all 参数对每个字段进行控制是否被包含在_all 字段中。默认值为 true，除非索引被设为 no。

include_in_all 参数还可以被设置在类型级别以及对象或嵌入字段，所有的子字段会继承这个设置，例如：

```
PUT secisland
{
  "mappings": {
    "my_type": {
      "include_in_all": false,
      "properties": {
        "title":                { "type": "string" },
        "author": {
          "include_in_all": true,
          "properties": {
            "first_name": { "type": "string" },
            "last_name":    { "type": "string" }
          }
        },
        "editor": {
          "properties": {
            "first_name": { "type": "string" },
            "last_name":    { "type": "string", "include_in_all": true }
          }
        }
      }
    }
  }
}
```

3.4.16 index 参数

index 选项控制字段值如何被索引，以及如何搜索。接受三种值：

no	不要在索引中加入这个字段的值。利用这个设置，字段不会被查询到。
not_analyzed	字段值原封不动地添加到索引中，作为单一索引词。除了字符串字段之外的所有字段默认支持这个选项。
analyzed	这个选项仅仅用在字符串字段上，作为默认选项。字符串值首先被分词为一组索引词，然后被索引。在搜索的时候，查询字符串会通过相同的分词器生成相同格式的索引词。正是由于这个过程，使得全文搜索成为可能。

表格 3.22 index 参数可选值

举例，创建一个不参与分词的字符串字段:

```
PUT /secisland
{
  "mappings": {
    "my_type": {
      "properties": {
        "status_code": {
          "type": "string",
          "index": "not_ analyzed"
        }
      }
    }
  }
}
```

3.4.17 index_options 参数

index_options 参数控制什么信息将被添加到反向索引，用于搜索和强调的目的。接受下面的参数：

docs	只有被索引的文档数量。可以解决“字段中是否包含这个索引词”的问题。
freqs	被索引的文档数量和索引词频率。索引词频率用来使重复索引词的得分高于单个索引词。
positions	文档数量，索引词频率以及索引词位置。位置可以被用于邻近或短语查询。
offsets	文档数量，索引词频率、位置以及开始和结束字符偏移量（映射索引词到原始字符串）。

表格 3.23 index_options 参数可选值

分词字符串字段利用 positions 作为默认值，其他字段利用 docs 作为默认值。

```
PUT secisland
{
  "mappings": {
    "my_type": {
      "properties": {
        "text": {
          "type": "string",
          "index_options": "offsets"
        }
      }
    }
```

```
    }
  }
```

3.4.18 lat_lon 参数

地理查询通过添加每个地理点字段的值到一个公式中，执行这个公式来决定地理点是否落入请求的区域中。不像大多数查询，反向索引并不复杂。

设置 lat_lon 为 true 会使纬度和经度的值作为数字型字段（称为.lat 和.lon）被索引。这些字段可以被用于地理范围查询和地理距离查询代替执行内存运算。

```
PUT secisland
{
  "mappings": {
    "my_type": {
      "properties": {
        "location": {
          "type": "geo_point",
          "lat_lon": true
        }
      }
    }
  }
}
```

设置 lat_lon 为 true，会在 location.lat 和 location.lon 字段索引地理点。

```
GET secisland/_search
{
  "query": {
    "geo_distance": {
      "location": {
        "lat": 41,
        "lon": -71
      },
      "distance": "50km",
      "optimize_bbox": "indexed"
    }
  }
}
```

indexed 参数会让地理距离查询利用反向索引值而不是内存运算。

是执行内存还是索引操作更好，取决于数据集和执行的查询类型。

注意：lat_lon 选项仅对单值的地理点字段有意义。不会对地理点数组产生效果。

3.4.19 fields 参数

多字段的目的是基于不同的目的用不同的方法索引相同的字段。例如，一个字符串字段可以作为分词字段被索引用于全文搜索，也可以作为不可分词字段用于排序或聚合:

```
PUT /secisland
{
  "mappings": {
    "my_type": {
```

```
        "properties": {
          "city": {
            "type": "string",
            "fields": {
              "raw": {
                "type":    "string",
                "index": "not_analyzed"
              }
            }
          }
        }
      }
    }
```

city.raw 字段是 city 字段不可分词的版本。

city 字段是分词字段，可以用来进行全文搜索。

city.raw 字段可以用来排序或聚合。

使用多重分词器

另一种多字段的情况是用不同的方式对相同的字段进行分词来达到更好的相关性。例如，我们可以索引一个字段，利用标准分词器将文本划分为单词，然后利用英文分词器把单词划分为它们的词根形式：

```
PUT secisland
{
  "mappings": {
    "my_type": {
      "properties": {
        "text": {
          "type": "string",
          "fields": {
            "english": {
              "type":         "string",
              "analyzer": "english"
            }
          }
        }
      }
    }
  }
}
```

text 字段使用标准分词器，text.english 字段使用英文分词器。

```
PUT secisland/my_type/1
{ "text": "quick brown fox" }

PUT secisland/my_type/2
```

```
{ "text": "quick brown foxes" }
```

索引两个文档，第一个文档中的 text 字段包含 fox 索引词，第二个文档中的 text 字段包含 foxes 索引词。两个文档的 text.english 字段均包含 fox 索引词，因为 foxes 的词根是 fox。

3.4.20 norms 参数

norms 存储各种标准化系数（一个数字），表示相关字段的长度和索引时相关性加权设置。将会用在查询的时候计算文档对于查询条件的相关性得分。

虽然对于计算相关性得分非常有用，但是 norms 也需要大量的内存。所以，如果不需要在一个特殊的字段上计算相关性得分，应该在字段上禁用 norms。在这种特别的情况下，字段仅仅用于过滤或聚合。

norms 可以利用 PUT 映射接口取消（不能被重新启用）：

```
PUT secisland/_mapping/my_type
{
  "properties": {
    "title": {
      "type": "string",
      "norms": {
        "enabled": false
      }
    }
  }
}
```

注意：标准值不会立即移除，但是索引新文件、旧分片融入新分片时，标准值会被移除。因为一些文档没有标准值，另外的仍然有标准值，任何移除已有标准值的字段计算出的得分可能会返回不同的结果。

标准值延迟加载

当新分片上线，标准值可以被优先（eager）加载到内存，或者当字段被查询时，标准值被延迟加载（lazy，默认）。

```
PUT secisland/_mapping/my_type
{
  "properties": {
    "title": {
      "type": "string",
      "norms": {
        "loading": "eager"
      }
    }
  }
}
```

3.4.21 null_value 参数

一个空值不能被索引或搜索。当一个字段被设置为 null（或者是一个空数组或者 null 值的数组），这个字段会被当作没有值的字段。

null_value 参数可以用指定的值替换掉确切的空值，以便可以被索引和搜索。例如：

```
PUT secisland
```

```
{
  "mappings": {
    "my_type": {
      "properties": {
        "status_code": {
          "type":          "string",
          "index":         "not_analyzed",
          "null_value": "NULL"
        }
      }
    }
  }
}

PUT secisland/my_type/1
{
  "status_code": null
}
```

利用索引词 NULL 替换掉确切的 null 值。

```
PUT secisland/my_type/2
{
  "status_code": []
}
```

因为空数组不包含确切的 null 值，所以不会被 null_value 的值替换。

重要：null_value 需要设置和字段相同的数据类型。

3.4.22 position_increment_gap 参数

为了可以支持短语查询，需要保存可分词字符串字段中分词的位置。当字符串字段索引多个值，一个“虚拟”缺口会被加到各个值之间来防止短语查询跨值匹配。缺口的大小可以利用 position_increment_gap 配置，默认值是 100。

例如：

```
PUT secisland
{
  "mappings": {
    "my_type": {
      "properties": {
        "names": {
          "type": "string",
          "position_increment_gap": 0
        }
      }
    }
  }
}
```

```
PUT /secisland/ my_type /1
{
    "names": [ "John Abraham", "Lincoln Smith"]
}
```

下一个元素数组的第一个索引词与上一个元素数组的最后一个索引词之间会有 0 个索引词间隔。

```
POST /secisland/ my_type /_search
{
    "query": {
        "match_phrase": {
            "names": "Abraham Lincoln"
        }
    }
}
```

短语查询匹配文档是怪异的，但是这正是我们在映射中要求的。

3.4.23 precision_step 参数

大多数的数字型数据类型索引额外的索引词表示每个数字的范围，使范围查询更加便捷：

```
"range": {
    "number": {
      "gte": 0
      "lte": 321
    }
  }
```

这从本质上作为索引词查询，结构如下：

```
"terms": {
    "number": [
      "0-255",
      "256-319"
      "320",
      "321"
    ]
  }
```

precision_step 的默认值取决于数字型字段的类型。

3.4.24 properties 参数

类型映射，对象字段和嵌套类型字段包含子字段，称为属性。这些属性可以是任何数据类型，包含对象和嵌套类型。属性可以被添加：

创建索引的时候明确定义。

利用创建映射接口添加或修改映射类型的时候明确定义。

索引包含新字段的文档可以动态添加。

向映射类型中添加属性，包含一个对象字段和嵌套类型字段的例子：

```
PUT secisland
{
  "mappings": {
```

```
    "my_type": {
      "properties": {
        "manager": {
          "properties": {
            "age":    { "type": "integer" },
            "name": { "type": "string"    }
          }
        },
        "employees": {
          "type": "nested",
          "properties": {
            "age":    { "type": "integer" },
            "name": { "type": "string"    }
          }
        }
      }
    }
  }
}
```

点符号

查询、聚合等方法可以使用点符号获取内部字段:

```
GET secisland/_search
{
  "query": {
    "match": {
      "manager.name": "Alice White"
    }
  },
  "aggs": {
    "Employees": {
      "nested": {
        "path": "employees"
      },
      "aggs": {
        "Employee Ages": {
          "histogram": {
            "field": "employees.age",
            "interval": 5
          }
        }
      }
    }
  }
}
```

重要：必须指定内部字段的完整路径。

3.4.25 search_analyzer 参数

通常，索引时和搜索时应该使用相同的分词器，确保查询时的索引词和反向索引中的索引词有相同的格式。

但有些时候，在搜索时使用不同的分词器是有意义的，比如当使用 edge_ngram 标记器来自动完成。

默认情况下，查询会使用字段映射中定义的分词器，但是可以利用 search_analyzer 设置来重写配置：

```
PUT /secisland
{
  "settings": {
    "analysis": {
      "filter": {
        "autocomplete_filter": {
          "type": "edge_ngram",
          "min_gram": 1,
          "max_gram": 20
        }
      },
      "analyzer": {
        "autocomplete": {
          "type": "custom",
          "tokenizer": "standard",
          "filter": [
            "lowercase",
            "autocomplete_filter"
          ]
        }
      }
    }
  },
  "mappings": {
    "my_type": {
      "properties": {
        "text": {
          "type": "string",
          "analyzer": "autocomplete",
          "search_analyzer": "standard"
        }
      }
    }
  }
}
```

分析器设置定义了自定义的自动完成分词器。

text 字段在索引的时候使用自动完成分词器，在搜索的时候使用标准分词器。

3.4.26 similarity 参数

Elasticsearch 允许对每个字段配置得分算法或者相似算法。similarity 设置提供了一个简单的方式来选择不同于默认的 TF/IDF 相似算法，例如 BM25。

相似算法多数用于字符串字段，特别是可分词的字符串字段，但是也可以应用于其他字段类型。

自定义相似算法可以通过调整内置相似算法的参数进行配置。

不用更多配置就可以直接使用的相似算法只有：

- default：Elasticsearch 和 Lucene 默认使用的 TF/IDF 算法。
- BM25：Okapi BM25 算法。

相似算法可以在字段第一次创建的时候在字段级别上进行设置：

```
PUT secisland
{
  "mappings": {
    "my_type": {
      "properties": {
        "default_field": {
          "type": "string"
        },
        "bm25_field": {
          "type": "string",
          "similarity": "BM25"
        }
      }
    }
  }
}
```

3.4.27 store 参数

默认情况下，字段值被索引来确保可以被搜索，但是他们不会被存储。这意味着字段可以被查询，但是原始字段值无法被取回。

通常这没有什么问题。字段值早已是默认存储的_source 字段的一部分。如果仅仅想取回单个字段或一些字段的值，而不是整个_source 字段，可以通过数据源过滤来实现：

```
PUT /secisland
{
  "mappings": {
    "my_type": {
      "properties": {
        "title": {
          "type": "string",
          "store": true
        },
        "date": {
          "type": "date",
          "store": true
```

```
            },
            "content": {
               "type": "string"
            }
         }
      }
   }
}
```

title 和 date 字段会被存储。

```
POST secisland/_search
{
   "fields": [ "title", "date" ]
}
```

这个请求可以取回 title 和 date 的字段值。

注意：

存储的字段作为数组返回

为了保持一致性，存储的字段作为一个数组返回。因为没有办法知道原始的字段值是一个单值、多值还是个空数组。

如果需要原始值，应该从_source 字段中取回。

3.4.28 term_vector 参数

索引词向量包含分析过程产生的索引词信息，包括：

- 索引词列表
- 每个索引词的位置（或顺序）
- 映射索引词到原始字符串中的原始位置中开始和结束字符的偏移量

这些索引词向量会被存储，所以可以作为一个特殊文档取回。

term_vector 设置接受以下参数：

no	不存储索引词向量（默认）。
yes	只存储字段的索引词。
with_positions	索引词和位置将会被存储。
with_offsets	索引词和字符偏移量会被存储。
with_positions_offsets	索引词，位置以及字符偏移量都会被存储。

表格 3.24 term_vector 参数可选值

警告：设置 with_positions_offsets 会使字段索引的大小翻倍。

映射示例：

```
PUT secisland
{
   "mappings": {
      "my_type": {
         "properties": {
            "text": {
               "type":          "string",
               "term_vector": "with_positions_offsets"
            }
```

```
        }
      }
    }
}
```

3.5 动态映射

3.5.1 概念

Elasticsearch 最重要的一个特种就是尝试跳出固有的方式，使用户尽可能快地检索数据。索引一篇文档，不需要事先创建索引、定义映射类型以及字段——仅需要直接索引一篇文档，然后索引、类型和字段会自动生成：

```
PUT data/counters/1
{ "count": 5 }
```

自动创建索引为 data，映射类型为 counters，字段名为 count，类型为长整型（long）

自动检测添加新类型和字段，被称为动态映射。可以根据目的自定义动态映射的规则:

- _default_映射：用于创建新映射类型的基础映射。
- 动态字段映射：控制动态字段检测规则。
- 动态模板：自定义规则来配置动态添加字段的映射。

禁用动态类型创建

动态类型创建可以通过设置 index.mapper.dynamic 为 false 来禁用，无论是在配置文件 config/elasticsearch.yml 中进行设置还是在每个索引中进行设置：

```
PUT /_settings
{
  "index.mapper.dynamic":false
}
```

对所有的索引禁用动态类型创建。

无论设置的值是什么，映射类型仍然可以在创建索引或者使用添加映射接口显式添加。

3.5.2 _default_映射

默认映射，用作每个新映射类型的基础映射，可以通过在索引中增加名为_default_的映射类型来自定义：

```
PUT secisland
{
  "mappings": {
    "_default_": {
      "_all": {
        "enabled": false
      }
    },
    "user": {},
    "blogpost": {
      "_all": {
        "enabled": true
      }
    }
  }
}
```

- _default_映射禁用了_all 字段。
- user 映射从_default_映射中继承了设置。
- blogpost 映射重写了默认设置，启用了_all 字段。

_default_映射可以在索引创建之后进行修改，新的默认映射只会影响到之后创建的映射类型。

3.5.3 动态字段映射

默认情况下，在文档中发现新字段的时候，Elasticsearch 会添加新字段到类型映射中。这种行为可以被禁用，通过设置动态参数为 false 或 strict。

如果动态字段映射被启用，一些简单的规则被用来决定字段应该是哪一种数据类型：

JSON 数据类型	Elasticsearch 数据类型
null	不添加字段
true 或 false	boolean 类型字段
浮点型数字	双精度浮点型字段
整数	长整型字段
对象	对象类型字段
数组	取决于数组中第一个非空值的类型
字符串	可能是日期型字段（值通过日期检查），双精度或长整型字段（值通过数字检查），可分词字符串型字段

表格 3.25 动态字段映射对应的数据类型

这些是仅有的可以动态添加的字段数据类型。所有其他的数据类型都必须被明确地添加到映射中。

除了下面列出的选项，可以通过动态模板自定义更多的动态字段映射规则。

- 日期检查

如果启用日期检查（默认），新的字符串字段的内容会被检查是否匹配任何 dynamic_date_formats 中指定的日期格式。如果发现匹配，新的日期字段会以相同的格式添加。

dynamic_date_formats 默认值是：

["strict_date_optional_time","yyyy/MM/dd HH:mm:ss Z||yyyy/MM/dd Z"]

- 禁用日期检查

通过设置 date_detection 的值为 false 来禁用动态日期检查：

```
PUT secisland
{
  "mappings": {
    "my_type": {
      "date_detection": false
    }
  }
}
```

- 自定义日期检查格式

动态日期格式可以自定义：

```
PUT secisland
{
  "mappings": {
    "my_type": {
```

```
            "dynamic_date_formats": ["MM/dd/yyyy"]
        }
      }
    }
```

- 数字检查

因为 JSON 支持浮点型和整型数据类型，一些应用或语言有时会作为字符串传递数字。通常的解决方式是明确定义这些字段映射，但是可以启用数字检查（默认禁用）来自动做类型转换：

```
PUT secisland
{
  "mappings": {
    "my_type": {
      "numeric_detection": true
    }
  }
}
```

3.5.4 动态模板

动态模板可以定义自定义映射用来动态添加字段，基于：

- 通过 Elasticsearch 进行数据类型检查，利用 match_mapping_type 参数。
- 字段名，利用 match and unmatch 或 match_pattern 参数。
- 字段全路径，利用 path_match and path_unmatch 参数。

原始字段名{name}和数据类型检查{dynamic_type}模板变量可以作为占位符用在映射标准中。

重要：动态字段映射仅在字段包含具体值（不为 null 或空数组）的时候添加。这意味着如果在 dynamic_template 中有 null_value 选项，第一个文档中被索引的字段有具体的值之后，null_value 才会起作用。

动态模板被指定为命名的对象数组：

```
"dynamic_templates": [
      {
        "my_template_name": {
          ...   match conditions ...
          "mapping": { ... }
        }
      },
      ...
  ]
```

- 模板名可以是任何字符串值。
- 匹配条件可以包含所有的：match_mapping_type，match，match_pattern，unmatch，path_match，path_unmatch。
- mapping 包含匹配到的字段需要使用的映射值。

模板顺序执行——第一个匹配到的模板起作用。新的模板可以用增加映射接口添加到列表的末尾。如果新的模板和已经存在的模板有相同的名字，旧版本会被替换掉。

- **match_mapping_type**

match_mapping_type 匹配数据类型，通过动态字段映射检查，换句话说就是 Elasticsearch 认为字段需要拥有的数据类型。只有下面的数据类型可以被自动检查：boolean，date，double，long，object，string。也可以接受*来匹配所有的数据类型。

举个例子，我们需要映射所有整数型字段为 integer 而不是 long，以及所有的可分词和不可分词的字符串型字段，我们可以使用下面的模板：

```
PUT secisland
{
  "mappings": {
    "my_type": {
      "dynamic_templates": [
        {
          "integers": {
            "match_mapping_type": "long",
            "mapping": {
              "type": "integer"
            }
          }
        },
        {
          "strings": {
            "match_mapping_type": "string",
            "mapping": {
              "type": "string",
              "fields": {
                "raw": {
                  "type":    "string",
                  "index": "not_analyzed",
                  "ignore_above": 256
                }
              }
            }
          }
        }
      ]
    }
  }
}
```

- **match 和 unmatch**

match 参数使用匹配字段名称的方式，unmatch 使用排除 match 匹配的字段的方式。

举个例子，匹配所有字符串型字段名字以 long_开头并排除以_text 结束的字段，索引这些字段为长整型字段：

```
PUT secisland
{
  "mappings": {
```

```
      "my_type": {
        "dynamic_templates": [
          {
            "longs_as_strings": {
              "match_mapping_type": "string",
              "match":     "long_*",
              "unmatch": "*_text",
              "mapping": {
                "type": "long"
              }
            }
          }
        ]
      }
    }
```

- **match_pattern**

match_pattern 参数支持完整的 Java 正则表达式匹配字段名而不是简单的通配符：

```
"match_pattern": "regex"
```

- **path_match 和 path_unmatch**

工作方式与 match 和 unmatch 相同，但是在字段全路径上进行操作，不仅仅是在最终的名字上：

```
PUT secisland
{
  "mappings": {
    "my_type": {
      "dynamic_templates": [
        {
          "full_name": {
            "path_match":     "name.*",
            "path_unmatch": "*.middle",
            "mapping": {
              "type":            "string",
              "copy_to":       "full_name"
            }
          }
        }
      ]
    }
  }
}
```

{name}和{dynamic_type}

占位符会被映射中的字段名和字段类型替换。举个例子，设置所有的字符型字段使用与字段同名的分析器，禁用所有非字符型字段的 doc_values 参数:

```
PUT secisland
{
  "mappings": {
    "my_type": {
      "dynamic_templates": [
        {
          "named_analyzers": {
            "match_mapping_type": "string",
            "match": "*",
            "mapping": {
              "type": "string",
              "analyzer": "{name}"
            }
          }
        },
        {
          "no_doc_values": {
            "match_mapping_type":"*",
            "mapping": {
              "type": "{dynamic_type}",
              "doc_values": false
            }
          }
        }
      ]
    }
  }
}
```

3.5.5 重写默认模板

可以重写所有索引的默认映射，也可以通过在索引模板中指定_default_类型映射重写映射类型。

例如，为了对所有新索引中的类型禁用_all 字段，可以创建下面这样的索引模板：

```
PUT _template/disable_all_field
{
  "disable_all_field": {
    "order": 0,
    "template": "*",
    "mappings": {
      "_default_": {
        "_all": {
          "enabled": false
        }
      }
    }
```

```
        }
    }
```

3.6 小结

本章介绍了 Elasticsearch 的映射，映射是 Elasticsearch 内部结构对外的一个展现方式，通过本章的学习可以了解 Elasticsearch 支持的字段的数据类型，以及 Elasticsearch 内置的元数据的知识；每一个类型都支持很多的参数，通过参考可以控制 Elasticsearch 的很多行为，比如是否索引等；同时 Elasticsearch 也是非常智能的，默认情况下，映射可以自动创建，并且可以很好的工作。