elasticsearch.
权威指南

Elasticsearch 权威指南

项目信息

GITHUB 仓库

GITBOOK 在线阅读 国外用户

SAE 在线阅读 国内用户

译者前言

译者现在的工作项目中需要用到elasticsearch，但是在网络中找了很多的相关的内容都很不完善，中文的文档更是寥寥无几，所以我决定边研究边翻译一下官方推出的权威手册。在这里要先感谢原作者们！如果各位在这里发现了错误之处，请大家在Issue中提出或者pr这个项目。

原作名字：elasticsearch - the definitive guide

原作作者：clinton gormley，zachary tong

译者：Gavin Foo fuxiaopang@gmail.com

前言

这本书还在不断地添加内容中，我们会陆陆续续地在这里添加新的章节。这本书中的内容针对的是Elasticsearch的最新版本。

欢迎反馈 – 如果这里出现了错误，或者你有什么建议可以到我们GitHub项目中 新建一个issue。

这个世界已经被数据淹没。我们创造的系统所产生的数据可以瞬间轻而易举地将我们压垮，现有的科技一直致力于如何存储数据，并能将拥有大量信息的数据仓库结构化。而当你准备开始从大量的数据中得出结论做决策的时候，美好的一天就要被毁灭了……

Elasticsearch是一个分布式可扩展的实时搜索和分析引擎。它能帮助你搜索、分析和浏览数据，而往往大家并没有在某个项目一开始就预料到需要这些功能。Elasticsearch之所以出现就是为了重新赋予硬盘中看似无用的原始数据新的活力。

无论你是需要全文搜索、机构化数据的实时统计，还是两者的结合，这本指南都会帮助你了解其中最基本的概念，从最基本的操作开始学习Elasticsearch。之后，我们还会逐渐开始探索更加复杂的搜索技术，你可以根据自身的学习的步伐。

Elasticsearch并不是单纯的全文搜索这么简单。我们将向你介绍讲解结构化搜索、统计、查询过滤、地理定位、自动完成以及你是不是要查找的提示。我们还将探讨如何给数据建模能提升Elasticsearch的性能，以及在生产环境中如何配置、监视你的集群。

入门

Elasticsearch是一个实时的分布式搜索和分析引擎。它可以帮助你用前所未有的速度去处理大规模数据。

它可以用于全文搜索，结构化搜索以及分析，当然你也可以将这三者进行组合

- 维基百科使用Elasticsearch来进行全文搜做并高亮显示关键词，以及提供search-as-you-type、did-you-mean等搜索建议功能。
- 英国卫报使用Elasticsearch来处理访客日志，以便能将公众对不同文章的反应实时地反馈给各位编辑。
- StackOverflow将全文搜索与地理位置和相关信息进行结合，以提供more-like-this相关问题的展现。
- GitHub使用Elasticsearch来检索超过1300亿行代码。
- 每天，Goldman Sachs使用它来处理5TB数据的索引，还有很多投行使用它来分析股票市场的变动。

但是Elasticsearch并不只是面向大型企业的，它还帮助了很多类似DataDog以及Klout的创业公司进行了功能的扩展。Elasticsearch可以运行在你的笔记本上，也可以部署到成千上万的服务器上，处理PB级别的数据。

Elasticsearch每一个独立的部分都不是新创的。比如全文搜索早就已经被实现，统计系统和分布式数据库也早已存在。但是革命之处在于能将这些独立的功能结合成一个连贯、实时处理的整体。对于新用户，它的门槛也很低，当然他也会因为你的强大而变得更强大。

你之所以拿起这本书，就是因为你眼前有很多的数据，但是你并不知道如何使用他们，接下来我们将开始探讨有关处理数据的事情。

很不幸的是，目前的大部分数据库在提取数据方面都是非常的薄弱的。虽然它们可以通过精准的时间戳或者确切的数值来进行内容的筛选，但是它们可以在全文搜索时做到同义词或者相关性搜索吗？他们可以汇总相同内容数据吗？最重要的是，每对如此巨大的数据量，它们能做到实时处理吗？

这便是Elasticsearch如此突出的理由：Elasticsearch可以帮助你浏览并利用已经快要烂在数据库里的那些极难查询的数据。

> **Elasticsearch**将会成为你一生的小伙伴。

了解搜索

Elasticsearch是一个建立在全文搜索引擎Apache Lucene(TM)基础上的搜索引擎，可以说Lucene是当今最先进，最高效的全功能开源搜索引擎框架。

但是Lucene只是一个框架，要充分利用它的功能，你需要使用JAVA，并且在你的程序中集成Lucene。更糟的是，你需要做很多的学习了解，才能明白它是如何运行的，Lucene确实非常复杂。

Elasticsearch使用Lucene作为内部引擎，但是在你使用它做全文搜索时，只需要使用统一开发好的API即可，请不需要了解其背后复杂的Lucene的运行原理。

当然Elasticsearch并不仅仅是Lucene这么简单，它不但包括了全文搜索功能，还可以进行以下工作:

- 分布式实时文件存储，并将每一个字段都编入索引，使其可以被搜索。
- 实时分析的分布式搜索引擎。
- 可以扩展到上百台服务器，处理PB级别的结构化或非结构化数据。

这么多的功能被集成到一台服务器上，你可以轻松地通过客户端或者任何你喜欢的程序语言与ES的RESTful API进行交流。

Elasticsearch的上手是非常简单的。它附带了很多非常合理的默认值，这让初学者很好地避免一上手就要面对复杂的理论，它安装好了就可以使用了，用很小的学习成本就可以变得很有生产力。

随着你越学越深入，你还可以利用Elasticsearch更多高级的功能，整个引擎可以很灵活地进行配置。你可以根据自身需求来定制属于你自己的Elasticsearch。

安装JAVA

```
yum install java-1.7.0-openjdk -y
```

安装Elasticsearch

最简单的了解Elasticsearch的方法就是去尽情的玩儿它（汗），准备好了我们就开始吧。

安装Elasticsearch只有一个要求，就是要安装最新版本的JAVA。你可以到官方网站下载它：www.java.com.

你可以在这里下载到最新版本的Elasticsearch： elasticsearch.org/download.

```
curl -L -O http://download.elasticsearch.org/PATH/TO/LATEST/$VERSION.zip
unzip elasticsearch-$VERSION.zip
cd  elasticsearch-$VERSION
```

提示: 当你安装Elasticsearch时，你可以到下载页面选择Debian或者RP安装包。或者你也可以使用官方提供的 Puppet module 或者 Chef cookbook.

安装Marvel

这是个付费的监控插件 暂时先不翻译

Marvel is a management and monitoring tool for Elasticsearch which is free for development use. It comes with an interactive console called Sense which makes it very easy to talk to Elasticsearch directly from your browser.

Many of the code examples in this book include a ``View in Sense" link. When clicked, it will open up a working example of the code in the Sense console. You do not have to install Marvel, but it will make this book much more interactive by allowing you to experiment with the code samples on your local Elasticsearch cluster.

Marvel is available as a plugin. To download and install it, run this command in the Elasticsearch directory:

```
./bin/plugin -i elasticsearch/marvel/latest
```

You probably don't want Marvel to monitor your local cluster, so you can disable data collection with this command:

```
echo 'marvel.agent.enabled: false' >> ./config/elasticsearch.yml
```

运行Elasticsearch

Elasticsearch已经蓄势待发，现在你便可以运行它了：

```
./bin/elasticsearch
```

如果你想让它在后台保持运行的话可以在命令后面再加一个 `-d`

开启后你就可以使用另一个终端窗口来进行测试了:

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

你应该看到如下提示：

```
{
   "status": 200,
   "name": "Shrunken Bones",
   "version": {
      "number": "1.1.0",
      "lucene_version": "4.7"
   },
   "tagline": "You Know, for Search"
}
```

这就说明你的Elasticsearch 群集 已经上线运行了，这是我们就可以进行各种实验了。

> 集群和节点

节点是Elasticsearch运行的实例。集群是一组有着同样`cluster.name`的节点，它们协同工作，互相分享数据，提供了故障转移和扩展的功能。当然一个节点也可以是一个集群。

与Elasticsearch通信

如何与Elasticsearch通信要取决于你是否使用JAVA。

Java API

如果你使用的是JAVA，Elasticsearch内置了两个客户端，你可以在你的代码中使用：

节点客户端:: 节点客户端以一个无数据节点的身份加入了一个集群。换句话说，它自身是没有任何数据的，但是他知道什么数据在集群中的哪一个节点上，然后就可以请求转发到正确的节点上并进行连接。

传输客户端:: 更加轻量的传输客户端可以被用来向远程集群发送请求。他并不加入集群本身，而是把请求转发到集群中的节点。

这两个客户端都使用Elasticsearch的传输协议，通过**9300**端口与java客户端进行通信。集群中的各个节点也是通过9300端口进行通信。如果这个端口被禁止了，那么你的节点们将不能组成一个集群。

> **TIP**

Java的客户端的版本号必须要与Elasticsearch节点所用的版本号一样，不然他们之间可能无法识别。

更多关于Java API的说明可以在这里找到 Guide.

通过HTTP向RESTful API传送json

其他的语言可以通过*9200*端口与Elasticsearch的RESTful API进行通信。事实上，如你所见，你甚至可以使用行命令`curl`来与Elasticsearch通信。

Elasticsearch官方提供了很多种编程语言的客户端，也有和许多社区化软件的集成插件，这些都可以在Guide里面找到。

向Elasticsearch发出的请求和其他所有的HTTP请求的组成部分是一致的。例如，计算集群中文件的数量，我们就可以使用：

```
      <1>     <2>                    <3>    <4>
curl -XGET 'http://localhost:9200/_count?pretty' -d '
{  <5>
    "query": {
        "match_all": {}
    }
}
'
```

1. 相应的HTTP请求方法或者变量: GET, POST, PUT, HEAD 或者 DELETE。
2. 集群中任意一个节点的访问协议、主机名以及端口。
3. 请求的路径。
4. 任意一个查询后再加上`?pretty`就可以生成更加美观的JSON反馈，以增强可读性。
5. 一个JSON编码的请求主体（如果需要的话）。

Elasticsearch将会返回一个HTTP状态码类似于'200 OK'，以及一个JSON格式的主体（除了单纯的'HEAD'请求），上面的请求会得到下方的JSON主体：

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

在反馈中，我们并没有看见HTTP的头部信息，因为我们没有告知`curl`显示这些内容。如果你想看到头部信息，可以在使用`curl`命令的时候再加上`-i`这个参数：

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

从现在开始，本书里所有涉及`curl`命令的部分我们都会进行简写，因为主机、端口等信息都是相同的，缩减前的样子:

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

我们将会简写成这样:

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

面向文档

程序中的对象很少是单纯的键值与数值的列表。更多的时候它拥有一个复杂的结构，比如包含了日期、地理位置、对象、数组等。

迟早你会把这些对象存储在数据库中。你会试图将这些丰富而又庞大的数据都放到一个由行与列组成的关系数据库中，然后你不得不根据每个字段的格式来调整数据，然后每次重建它你都要检索一遍数据。

Elasticsearch是面向文档型数据库，这意味着它存储的是整个对象或者文档，它不但会存储它们，还会为他们建立索引，这样你就可以搜索他们了。你可以在Elasticsearch中索引、搜索、排序和过滤这些文档。不需要成行成列的数据。这将会是完全不同的一种面对数据的思考方式，这也是为什么Elasticsearch可以执行复杂的全文搜索的原因。

JSON

Elasticsearch使用*JSON* (或称作JavaScript Object Notation)作为文档序列化的格式。JSON已经被大多数语言支持，也成为NoSQL领域的一个标准格式。它简单、简洁、易于阅读。

把这个JSON想象成一个用户对象:

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

虽然user这个对象非常复杂，但是它的结构和含义都被保留到JSON中了。在Elasticsearch中，将对象转换为JSON并作为索引要比在表结构中做相同的事情简单多了。

> 将你的数据转换为**JSON**

几乎所有的语言都有将任意数据转换、机构化成JSON，或者将对象转换为JSON的模块。查看`serialization`以及`marshalling`两个JSON模块。The official Elasticsearch clients 也可以帮你自动结构化JSON。

启程

为了能让你感受一下Elasticsearch能做什么以及它是有多么的易用，我们会先为你简单展示一下，其中包括了基本的创建索引，搜索以及聚合。

我们会在这里向你介绍一些新的术语以及简单的概念，即使你没有马上接受这些概念也没有关系。我们会在之后的章节中逐渐帮你理解它们。

所以，准备开始享受Elasticsearch的学习之旅把！

建立一个员工名单

想象我们正在为一个名叫megacorp的公司的HR部门制作一个新的员工名单系统，这些名单应该可以满足实时协同工作，所以它应该可以满足以下要求：

- 数据可以包含多个值的标签、数字以及纯文本内容，
- 可以检索任何职员的所有数据。
- 允许结构化搜索。例如，查找30岁以上的员工。
- 允许简单的全文搜索以及相对复杂的短语搜索。
- 在返回的匹配文档中高亮关键字。
- 拥有数据统计与管理的后台。

为员工档案创建索引

这个项目的第一步就是存储员工的数据。这样你就需要一个“员工档案”的表单，这样每个文档都代表着一个员工。在Elasticsearch中，存储数据的行为就叫做索引*(indexing)*，但是在我们索引数据前，我们需要决定将数据存储在哪里。

在Elasticsearch中，文档术语一种类型*(type)*，各种各样的类型存在于一个索引中。你也可以通过类比传统的关系数据库得到一些大致的相似之处：

```
关系数据库      ⇒ 数据库 ⇒ 表     ⇒ 行     ⇒ 列(Columns)
Elasticsearch  ⇒ 索引   ⇒ 类型  ⇒ 文档   ⇒ 字段(Fields)
```

一个Elasticsearch集群可以包含多个索引（数据库），也就是说其中包含了很多类型（表）。这些类型中包含了很多的文档（行），然后每个文档中又包含了很多的字段（列）。

> 索引 索引 索引

你可能发现在Elasticsearch中，索引这个词汇已经被赋予了太多意义，所以在这里我们有必要澄清一下：

> 索引 (名词)

如上文所说，一个索引就类似于传统关系型数据库中的数据库。这里就是存储相关文档的的地方。

> 索引 (动词)

为一个文档创建索引是把一个文档存储到一个索引*(名词)*中的过程，这样它才能被检索。这个过程非常类似于SQL中的`INSERT`命令，如果已经存在文档，新的文档将会覆盖旧的文档。

> 反向索引

在关系数据库中的某列添加一个索引，比如多路搜索树(B-Tree)索引，就可以加速数据的取回速度， Elasticsearch以及Lucene使用的是一个叫做反向索引*(inverted index)*的结构来实现相同的功能。

> 通常，每个文档中的字段都被创建了索引（拥有一个反向索引），因此他们可以被搜索。如果一个字段缺失了反向索引的话，它将不能被搜索。我们将会在之后的《反向索引》章节中详细介绍它。

所以为了创建员工名单，我们需要进行如下操作：

- 为每一个员工的文档创建索引，每个文档都包含了一个员工的所有信息。
- 每个文档都会被标记为`employee`类型。
- 这种类型将存活在`employee`这个索引中。
- 这个索引将会存储在Elasticsearch的集群中

在实际的操作中，这些操作是非常简单的（及时看起来有这么多步骤）。我们可以把如此之多的操作通过一个命令来完成：

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

注意在`/megacorp/employee/1`路径下，包含了三个部分：

名字　　内容

megacorp 索引的名字
employee 类型的名字
1 当前员工的ID

请求部分，也就是JSON文档，在这里包含了关于这名员工的所有信息。他的名字是``John Smith"，他已经25岁了，他很喜欢攀岩。

怎么样？很简单吧！我们在操作前不需要进行任何管理操作，比如创建索引，或者为字段指定数据的类型。我们就这么直接地为一个文档创建了索引。Elasticsearch会在创建的时候为它们设定默认值，所以所有管理操作已经在后台被默默地完成了。

在进行下一步之前，我们再为这个目录添加更多的员工信息吧：

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

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

检索文档

现在，我们已经在Elasticsearch中存储了一些数据，我们可以开始根据这个项目的需求进行工作了。第一个需求就是要能搜索每一个员工的数据。

对于Elasticsearch来说，这是非常简单的。我们只需要执行一次HTTP GET请求，然后指出文档的地址，也就是索引、类型以及ID即可。通过这三个部分，我们就可以得到原始的JSON文档：

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

返回的内容包含了这个文档的元数据信息，而John Smith的原始JSON文档也在`_source`字段中出现了：

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

我们通过将HTTP后的请求方式由GET改变为PUT来获取文档，同理，我们也可以将其更换为DELETE来删除这个文档，HEAD是用来查询这个文档是否存在的。如果你想替换一个已经存在的文档，你只需要使用PUT再次发出请求即可。

简易搜索

GET命令真的相当简单，你只需要告诉它你要什么即可。接下来，我们来试一下稍微复杂一点的搜索。

我们首先要完成一个最简单的搜索命令来搜索全部员工：

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

你可以发现我们正在使用megacorp索引，employee类型，但是我们我们并没有指定文档的ID，我们现在使用的是_search端口。你可以再返回的hits中发现我们录入的三个文档。搜索会默认返回最前的10个数值。

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

注意：反馈值中不仅会到诉你匹配到那些文档，同时也会把这个文档都会包含到其中：我们需要搜索的用户的所有信息。

接下来，我们将要尝试着实现搜索一下哪些员工的姓氏中包含Smith。为了实现这个，我们需要使用一种轻量的搜索方法。这种方法经常被称做查询字符串*(query string)*搜索，因为我们通过URL来传递查询的关键字：

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

我们依旧使用_search端口，然后可以将参数传入给q=。这样我们就可以得到姓Smith的结果：

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

使用Query DSL搜索

查询字符串是通过命令语句完成点对点*(ad hoc)*的搜索，但是这也有它的局限性（可参阅《搜索局限性》章节）。Elasticsearch提供了更加丰富灵活的查询语言，它被称作*Query DSL*，通过它你可以完成更加复杂、强大的搜索任务。

DSL (*Domain Specific Language* 领域特定语言)需要使用JSON作为主体，我们还可以这样查询姓Smith的员工：

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

这个请求会返回同样的结果。你会发现我们在这里没有使用查询字符串，而是使用了一个由JSON构成的请求体，其中使用了match查询法，随后我们还将会学习到其他的查询类型。

更加复杂的搜索

接下来，我们再提高一点儿搜索的难度。我们依旧要寻找出姓Smith的员工，但是我们还将添加一个年龄大于30岁的限定条件。我们的查询语句将会有一些细微的调整来以识别结构化搜索的限定条件 *filter*（过滤器）：

```
GET /megacorp/employee/_search
{
    "query" : {
        "filtered" : {
            "filter" : {
                "range" : {
                    "age" : { "gt" : 30 } <1>
                }
            },
            "query" : {
                "match" : {
                    "last_name" : "Smith" <2>
                }
            }
        }
    }
}
```

1. 这一部分的语句是`range`*filter*，它可以查询所有超过30岁的数据 -- `gt`代表 **greater than** （大于）。
2. 这一部分我们前一个操作的`match`*query* 是一样的

先不要被这么多的语句吓到，我们将会在之后带你逐渐了解他们的用法。你现在只需要知道我们添加了一个*filter*，可以在`match`的搜索基础上再来实现区间搜索。现在，我们的只会显示32岁的名为**Jane Smith**的员工了：

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

全文搜索

上面的搜索都很简单：名字搜索、通过年龄过滤。接下来我们来学习一下更加复杂的搜索，全文搜索——一项在传统数据库很难实现的功能。 我们将会搜索所有喜欢**rock climbing**的员工:

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

你会发现我们同样使用了match查询来搜索about字段中的**rock climbing**。我们会得到两个匹配的文档：

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

1. 相关评分

通常情况下，Elasticsearch会通过相关性来排列顺序，第一个结果中，John Smith的about字段中明确地写到**rock climbing**。而在Jane Smith的about字段中，提及到了**rock**，但是并没有提及到**climbing**，所以后者的_score就要比前者的低。

这个例子很好地解释了Elasticsearch是如何执行全文搜索的。对于Elasticsearch来说，相关性的感念是很重要的，而这也是它与传统数据库在返回匹配数据时最大的不同之处。

段落搜索

能够找出每个字段中的独立单词最然很好，但是有的时候你可能还需要去匹配精确的短语或者段落。例如，我们只需要查询到about字段只包含**rock climbing**的短语的员工。

为了实现这个效果，我们将对match查询变为match_phrase查询：

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

这样，系统会没有异议地返回John Smith的文档：

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

高亮我们的搜索

很多程序希望能在搜索结果中高亮匹配到的关键字来告诉用户这个文档是如何匹配他们的搜索的。在Elasticsearch中找到高亮片段是非常容易的。

让我们回到之前的查询，但是添加一个`highlight`参数：

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

当我们运行这个查询后，相同的命中结果会被返回，但是我们会得到一个新的名叫`highlight`的部分。在这里包含了`about`字段中的匹配单词，并且会被`<em></em>`HTML字符包裹住：

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

1. 在原有文本中高亮关键字。

统计

最后，我们还有一个需求需要完成：可以让老板在职工目录中进行统计。Elasticsearch把这项功能称作汇总 *(aggregations)*，通过这个功能，我们可以针对你的数据进行复杂的统计。这个功能有些类似于SQL中的`GROUP BY`，但是要比它更加强大。

例如，让我们找一下员工中最受欢迎的兴趣是什么：

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

请忽略语法，让我们先来看一下结果：

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

我们可以发现有两个员工喜欢音乐，还有一个喜欢森林，还有一个喜欢运动。这些数据并没有被预先计算好，它们是在文档被查询的同时实时计算得出的。如果你想要查询姓Smith的员工的兴趣汇总情况，你就可以执行如下查询：

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

这样，all_interests的统计结果就只会包含满足查询的文档了：

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

汇总还允许多个层面的统计。比如我们还可以统计每一个兴趣下的平均年龄：

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

```
}
```

虽然这次返回的汇总结果变得更加复杂了，但是它依旧很容易理解：

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

在这个丰富的结果中，我们不但可以看到兴趣的统计数据，还能针对不同的兴趣来分析喜欢这个兴趣的平均年龄。

即使你现在还不能很好地理解语法，但是相信你还是能发现，用这个功能来实现如此复杂的统计工作是这样的简单。你的极限取决于你存入了什么样的数据哟！

小结

希望上面的几个小教程可以很好地向你解释Elasticsearch可以实现什么功能。为了保持教程简短，这里只提及了一些基础，除此之外还有很多功能，比如建议、地理定位、过滤、模糊以及部分匹配等。但是相信你也发现了，在这里你只需要很简单的操作就可以完成复杂的操作。无需配置，添加数据就可以开始搜索！

可能前面有一些语法会让你觉得很难理解，你可能对如何调整优化它们还有很多疑问。那么，本书之后的章节将会帮助你逐步解开疑问，让你对Elasticsearch是如何工作的有一个全面的了解。

分布式特性

在最开始的章节中，我们曾经提到Elasticsearch可以被扩展到上百台（甚至上千台）服务器上，来处理PB级别的数据。我们的教程只提及了如何使用它，但是并没有提及到服务器方面的内容。Elasticsearch是自动分布的，它在设计时就考虑到可以隐藏分布操作的复杂性。

Elasticsearch的分布式部分很简单。你甚至不需要关于分布式系统的任何内容，比如分片、集群、发现等成堆的分布式概念。你可能在你的笔记本中运行着刚才的教程，如果你想在一个拥有100个节点的集群中运行教程，你会发现操作是完全一样的。

Elasticsearch很努力地在避免复杂的分布式系统，很多操作都是自动完成的：

- 可以将你的文档分区到不同容器或者分片中，这些文档可能被存在一个节点或者多个节点。
- 跨界点平衡集群中节点间的索引与搜索负载。
- 自动复制你的数据以提供冗余副本，防止硬件错误导致数据丢失。
- 自动在节点之间路由，以帮助你找到你想要的数据。
- 无缝扩展或者恢复你的集群。

当你在阅读这本书时，你会发现到有关Elasticsearch的分布式特性分布式特性的补充章节。在这些章节中你会了解到如何扩展集群以及故障转移（《分布式集群》），如何处理文档存储（《分布式文档》），如何执行分布式搜索(《分布式搜索》）

这一部分不是必须要看的——你不懂它们也能正常使用Elasticsearch。但是帮助你更加全面完整地了解Elasticsearch。你也可以在之后需要的时候再回来翻阅它们。

总结

到目前为止，你应该已经知道Elasticsearch可以实现哪些功能，入门上手是非常简单的。只需要最少的知识和配置就可以开始使用Elasticsearch也是它的追求。学习Elasticsearch最好的方法就是开始使用它：进行的检索与搜索吧！

当然，学得越多，你的生产力就越高。你也就能对特定的内容进行微调，得到更适合你的结果。

之后的章节，我们将会引领你从新手晋级到专家。每一个站街都会解释一个要点，同时我们也会提供专家级别的小提示。如果你只是刚刚起步，这些提示可能并不是很适合你。Elasticsearch 会在一开始设置很多合理的默认值。你可以在需要提升性能的时候再重新回顾它们。

集群

本章将会在主要章节翻译结束后再继续翻译

> 补充章节

正如前文提到的，这就是第个补充的章节，这里会介绍Elasticsearch如何在分布式环境中运行。 本章解释了常用术语，比如集群 *(cluster)*, 节点 *(node)* 以及分片 *(shard)*，以及如何横向扩展主机，如何处理硬件故障。

尽管这一章不是必读章节 —— 你可以完全不用理会分片，复制以及故障恢复就能长时间使用Elasticsearch。你可以先跳过这一章节，然后在你需要的时候再回来。

你可以随时根据你的需要扩展Elasticsearch。你可以购买配置更好的主机(*vertical scale* or *scaling up*) 或者购买过多的主机 (*horizontal scale* or *scaling out*) 来达到扩展的目的。

硬件越强大，Elasticsearch运行的也就越快，但是垂直扩展(vertical scale)方式也有它的局限性。真正的扩展来自于横向扩展 (horizontal scale)方式，在集群中添加更多的节点，这样能在节点之间分配负载。

对于大多数数据库来说，横向扩展意味着你的程序往往需要大改，以充分使用这些新添加的设备。相比而言，Elasticsearch自带分布式功能：他知道如何管理多个节点并提供高可用性。这也就意味着你的程序根本不需要为扩展做任何事情。

在这一章节，我们将要探索如何根据你的需要创建你的集群，节点以及分片，并保障硬件故障后，你的数据依旧的安全。

空集群

If we start a single node, with no data and no indices, our cluster looks like 

A *node* is a running instance of Elasticsearch, while a *cluster* consists of one or more nodes with the same `cluster.name` that are working together to share their data and workload. As nodes are added to or removed from the cluster, the cluster reorganizes itself to spread the data evenly.

One node in the cluster is elected to be the *master* node, which is in charge of managing cluster-wide changes like creating or deleting an index, or adding or removing a node from the cluster. The master node does not need to be involved in document level changes or searches, which means that having just one master node will not become a bottleneck as traffic grows. Any node can become the master. Our example cluster has only one node, so it performs the master role.

As users, we can talk to *any node in the cluster*, including the master node. Every node knows where each document lives and can forward our request directly to the nodes that hold the data we are interested in. Whichever node we talk to manages the process of gathering the response from the node or nodes holding the data and returning the final response to the client. It is all managed transparently by Elasticsearch.

集群健康

There are many statistics that can be monitored in an Elasticsearch cluster but the single most important one is the *cluster health*, which reports a `status` of either `green`, `yellow` or `red`:

```
GET /_cluster/health
```

which, on an empty cluster with no indices, will return something like:

```
{
   "cluster_name":          "elasticsearch",
   "status":                "green", <1>
   "timed_out":             false,
   "number_of_nodes":       1,
```

```
   "number_of_data_nodes":  1,
   "active_primary_shards": 0,
   "active_shards":         0,
   "relocating_shards":     0,
   "initializing_shards":   0,
   "unassigned_shards":     0
}
```

1. The `status` field is the one we're most interested in.

The `status` field provides an overall indication of how the cluster is functioning. The meaning of the three colors are provided here for reference:

[horizontal] `green`:: All primary and replica shards are active. `yellow`:: All primary shards are active, but not all replica shards are active. `red`:: Not all primary shards are active.

In the rest of this chapter we explain what *primary* and *replica* shards are and explain the practical implications of each of the above colors.

=== Add an index

To add data to Elasticsearch, we need an *index* -- a place to store related data. In reality, an index is just a ``logical namespace'' which points to one or more physical *shards*.

A *shard* is a low-level ``worker unit''. Each shard is a single instance of Lucene, and is a complete search engine in its own right. Our documents are stored and indexed in shards, but our applications don't talk to them directly. Instead, they talk to an index.

Shards are how Elasticsearch distributes data around your cluster. Think of shards as containers for data. Documents are stored in shards, and shards are allocated to nodes in your cluster. As your cluster grows or shrinks, Elasticsearch will automatically migrate shards between nodes so that the cluster remains balanced.

A shard can be either a *primary* shard or a *replica* shard. Each document in your index belongs to a single primary shard, so the number of primary shards that you have determines the maximum amount of data that your index can hold.

While there is no theoretical limit to the amount of data that a primary shard can hold, there is a practical limit. What constitutes the maximum shard size depends entirely on your use case: the hardware you have, the size and complexity of your documents, how you index and query your documents, and your expected response times.

A replica shard is just a copy of a primary shard. Replicas are used to provide redundant copies of your data to protect against hardware failure, and to serve read requests like searching or retrieving a document.

The number of primary shards in an index is fixed at the time that an index is created, but the number of replica shards can be changed at any time.

Let's create an index called `blogs` in our empty one-node cluster. By default, indices are assigned 5 primary shards, but for the purpose of this demonstration, we'll assign just 3 primary shards and 1 replica (one replica of every primary shard):

[source,js]

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

}

// SENSE: 020_Distributed_Cluster/15_Add_index.json

[[cluster-one-node]] .A single-node cluster with an index image::images/02-02_one_node.png["A single-node cluster with an index"]

Our cluster now looks like <> -- all 3 primary shards have been allocated to `Node 1`. If we were to check the <> now, we would see this:

[source,js]

{ "cluster_name": "elasticsearch", "status": "yellow", <1> "timed_out": false, "number_of_nodes": 1, "number_of_data_nodes": 1, "active_primary_shards": 3, "active_shards": 3, "relocating_shards": 0, "initializing_shards": 0, "unassigned_shards": 3 <2>

}

<1> Cluster `status` is `yellow`.

<2> Our three replica shards have not been allocated to a node.

A cluster health of `yellow` means that all *primary* shards are up and running -- the cluster is capable of serving any request successfully -- but not all *replica* shards are active. In fact all three of our replica shards are currently `unassigned` -- they haven't been allocated to a node. It doesn't make sense to store copies of the same data on the same node. If we were to lose that node, we would lose all copies of our data.

Currently our cluster is fully functional but at risk of data loss in case of hardware failure.

=== Add failover

Running a single node means that you have a single point of failure -- there is no redundancy. Fortunately all we need to do to protect ourselves from data loss is to start another node. A new node will join the cluster automatically as long as it has the same cluster name set in its config file, and it can talk to the other nodes.

If we start a second node, our cluster would look like <>.

[[cluster-two-nodes]] .A two-node cluster -- all primary and replica shards are allocated image::images/02-03_two_nodes.png["A two-node cluster"]

The second node has joined the cluster and three *replica shards* have been allocated to it -- one for each primary shard. That means that we can lose either node and all of our data will be intact.

Any newly indexed document will first be stored on a primary shard, then copied in parallel to the associated replica shard(s). This ensures that our document can be retrieved from a primary shard or from any of its replicas.

The `cluster-health` now shows a status of `green`, which means that all 6 shards (all 3 primary shards and all 3 replica shards) are active:

[source,js]

{ "cluster_name": "elasticsearch", "status": "green", <1> "timed_out": false, "number_of_nodes": 2, "number_of_data_nodes": 2, "active_primary_shards": 3, "active_shards": 6, "relocating_shards": 0, "initializing_shards": 0, "unassigned_shards": 0

}

<1> Cluster `status` is `green`.

Our cluster is not only fully functional but also *always available.*

=== Scale horizontally

What about scaling as the demand for our application grows? If we start a third node, our cluster reorganizes itself to look like <>.

[[cluster-three-nodes]] .A three-node cluster -- shards have been reallocated to spread the load image::images/02-04_three_nodes.png["A three-node cluster"]

One shard each from `Node 1` and `Node 2` have moved to the new `Node 3` and we have two shards per node, instead of three. This means that the hardware resources (CPU, RAM, I/O) of each node are being shared between fewer shards, allowing each shard to perform better.

A shard is a fully fledged search engine in its own right, and is capable of using all of the resources of a single node. With our total of 6 shards (3 primaries and 3 replicas) our index is capable of scaling out to a maximum of 6 nodes, with one shard on each node and each shard having access to 100% of its node's resources.

==== Then scale some more

But what if we want to scale our search to more than 6 nodes?

The number of primary shards is fixed at the moment an index is created. Effectively, that number defines the maximum amount of data that can be *stored* in the index. (The actual number depends on your data, your hardware and your use case). However, read requests -- searches or document retrieval -- can be handled by a primary *or* a replica shard, so the more copies of data that you have, the more search throughput we can handle.

The number of replica shards can be changed dynamically on a live cluster, allowing us to scale up or down as demand requires. Let's increase the number of replicas from the default of `1` to `2`:

[source,js]

PUT /blogs/_settings { "number_of_replicas" : 2

}

// SENSE: 020_Distributed_Cluster/30_Replicas.json

[[cluster-three-nodes-two-replicas]] .Increasing the `number_of_replicas` to 2 image::images/02-05_replicas.png["A three-node cluster with two replica shards"]

As can be seen in <>, the `blogs` index now has 9 shards: 3 primaries and 6 replicas. If we were to add another three nodes to our 6 node cluster, we would again have one shard per node, and our cluster would be able to handle *50%* more search requests than before.

[NOTE]

Of course, just having more replica shards on the same number of nodes doesn't increase our performance at all because each shard has access to a smaller fraction of its node's resources. You need to add hardware to increase throughput.

But these extra replicas do mean that we have more redundancy. With the node configuration above, we can now afford to lose two nodes without losing any data.

==========

错误恢复

We've said that Elasticsearch can cope when nodes fail, so let's go ahead and try it out. If we kill the first node our cluster looks like <>.

[[cluster-post-kill]] .Cluster after killing one node image::images/02-06_node_failure.png["The cluster after killing one node"]

The node we killed was the master node. A cluster must have a master node in order to function correctly, so the first thing that happened was that the nodes elected a new master: `Node 2`.

Primary shards `1` and `2` were lost when we killed `Node 1` and our index cannot function properly if it is missing primary shards. If we had checked the cluster health at this point, we would have seen status `red`: not all primary shards are active!

Fortunately a complete copy of the two lost primary shards exists on other nodes, so the first thing that the new master node did was to promote the replicas of these shards on `Node 2` and `Node 3` to be primaries, putting us back into cluster health `yellow`. This promotion process was instantaneous, like the flick of a switch.

So why is our cluster health `yellow` and not `green`? We have all 3 primary shards, but we specified that we wanted two replicas of each primary and currently only one replica is assigned. This prevents us from reaching `green`, but we're not too worried here: were we to kill `Node 2` as well, our application could *still* keep running without data loss because `Node 3` contains a copy of every shard.

By now you should have a reasonable idea of how shards allow Elasticsearch to scale horizontally and to ensure that your data is safe. Later we will examine the life-cycle of a shard in more detail.

存入 取出

不论是哪一个程序，目的都是一致的：根据我们的需要来组织数据。数据并不是只有随机的比特和字节。我们需要在多个代表现实中的事物或者实体的元素之间建立关系。如果我们知道一个名字和一个电子邮件同属于一个人的话，我们就能得到更有意义的东西。

生活中，有很多同种类的实体也存在着不同。一个人可能有一个座机号码，而另一个人可能只有手机号码，当然有的人还会同时拥有两者。有的人有三个邮箱地址，有的则可能一个都没有。一个西班牙人可能有两个姓，但是我们可能就只有一个姓。

面向对象编程语言之所以受到大家欢迎，其中一个原因就是对象能帮助我们去代替现实生活中的复杂的实体。

但是当我们存储这些实体时就会出现问题。通常，我们把我们的数据存储在关系数据库的列(columns)与行(rows)中，这相当于在电子表格中排列我们的数据。这样我们对象本来的灵活性就不复存在了。

但是，如果我们就单纯地把对象存储为对象呢？相比于在电子表中的各种限制，我们应该重新着眼于使用数据。把对象本来应有的灵活性找回来。

对象**(object)**是特定语言(language-specific)，包含数据(in-memory)的数据结构。通过网络发送或者存储它，我们需要一个标准格式来代表它。JSON (JavaScript Object Notation)这是一种可读文本的对象形式。它已经成为NoSQL世界中的标准。当一个对象被序列化到JSON时，它就被成为**JSON**文档了。

Elasticsearch 是一个分布式文档存储器。它可以实时地存储并检索复杂的数据结构（被序列化后的JSON文档）。换句话说，当文档被存储到Elasticsearch 后，它就可以被集群中的任意节点搜索了。

当然，我们不但需要存储数据，还需要能够大量快速地进行查询。虽然已经有很多的NOSQl解决方案，可以让我们将对象存储为文档，但是他们还是需要我们去考虑如何查询我们的数据以及哪些字段需要我们来做索引，以便快速搜索。

在Elasticsearch中，每一个字段都会默认被建立索引。也就是说，每一个字段都会有一个反向索引以便快速搜索。而且，与大多数其他数据库不同的是ES可以在同一个查询中使用所有的反向索引，以惊人的速度返回查询结果。

在本章中，我们将探讨如何使用API来创建、搜索、更新、删除文档。目前，我们并不用关心数据是如何存储在文档中的，我们只需要关心我们的文档是如何被安全地存储在Elasticsearch中以及我们如何能再次获取到他们就可以了。

文档是什么？

在很多程序中，大部分实体或者对象都被序列化为包含键和值的JSON对象。键是一个字段或者属性的名字，值可以是一个字符串、数字、布尔值、对象、数组或者是其他的特殊类型，比如代表日期的字符串或者单表地理位置的对象：

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

通常情况下，我们使用可以互换对象和文档。然而，还是有一个区别的。对象(object)仅仅是一个JSON对象,类似于哈希，哈希映射，字典或关联数组。对象(Objects)则可以包含其他对象(Objects)。

在Elasticsearch中，文档这个单词有特殊的含义。它指的是在Elasticsearch中被存储到唯一ID下的由最高级或者根对象 *(root object)*序列化而来的JSON。

文档元数据

一个文档不只包含了数据。它还包含了元数据*(metadata)* —— 关于文档的信息。有三个元数据元素是必须存在的，它们是：

名字	说明
_index	文档存储的地方
_type	文档代表的对象种类
_id	文档的唯一编号

_index

索引 类似于传统数据库中的"数据库"——也就是我们存储并且索引相关数据的地方。

> **TIP**：

在Elasticsearch中，我们的数据都在分片中被存储以及索引，索引只是一个逻辑命名空间，它可以将一个或多个分片组合在一起。然而，这只是一个内部的运作原理——我们的程序可以根本不用关心分片。对于我们的程序来说，我们的文档存储在索引中。剩下的交给Elasticsearch就可以了。

我们将会在《索引管理》章节中探讨如何创建并管理索引。但是现在，我们只需要让Elasticsearch帮助我们创建索引。我们只需要选择一个索引的名字。这个名称必须要全部小写，也不能以下划线开头，不能包含逗号。我们可以用`website`作为我们索引的名字。

_type

在程序中，我们使用对象代表“物品”，比如一个用户、一篇博文、一条留言或者一个邮件。每一个对象都属于一种类型，类型定义了对象的属性或者与数据的关联。用户类的对象可能就会包含名字、性别、年龄以及邮箱地址等。

在传统的数据库中，我们总是将同类的数据存储在同一个表中，因为它们的数据格式是相同的。同理，在Elasticsearch中，我们使用同样类型的文档来代表同类“事物”，也是因为它们的数据结构是相同的。

每一个类型都拥有自己的映射(mapping)或者结构定义，它们定义了当前类型下的数据结构，类似于数据库表中的列。所有类型下的文档会被存储在同一个索引下，但是映射会告诉Elasticsearch不同的数据应该如何被索引。

我们将会在《映射》中探讨如何制定或者管理影视，但是目前为止，我们只需要依靠Elasticsearch来自动处理数据结构。

_id

*id*是一个字符串，当它与_index以及_type组合时，就可以来代表Elasticsearch中一个特定的文档。我们创建了一个新的文档时，你可以自己提供一个_id，或者也可以让Elasticsearch帮你生成一个。

其他元数据

在文档中还有一些其他的元数据，我们将会在《映射》章节中详细讲解。使用上面罗列的元素，我们已经可以在Elasticsearch中存储文档或者通过ID来搜索已经保存的文档了。

索引一个文档

文档通过索引API被索引——存储并使其可搜索。但是最开始我们需要决定我们将文档存储在那里。正如之前提到的，一篇文档通过_index, _type以及_id来确定它的唯一性。我们可以自己提供一个_id，或者也使用indexAPI 帮我们生成一个。

使用自己的**ID**

如果你的文档拥有天然的标示符（例如`user_account`字段或者文档中其他的标识值），这时你就可以提供你自己的`_id`，这样使用`index`API：

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

几个例子。如果我们的索引叫做`"website"`，我们的类型叫做`"blog"`，然后我们选择`"123"`作为ID的编号。这时，请求就是这样的：

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

Elasticsearch回馈内容：

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

这个回馈意味着我们的索引请求已经被成功创建，其中还包含了`_index`, `_type`以及`_id`的元数据，以及一个新的元素`_version`。

在Elasticsearch中，每一个文档都有一个版本号码。每当文档产生变化时（包括删除），`_version`就会增大。在《版本控制》中，我们将会详细讲解如何使用`_version`的数字来确认你的程序不会随意替换掉不想覆盖的数据。

自增**ID**

如果我们的数据中没有天然的标示符，我们可以让Elasticsearch为我们自动生成一个。请求的结构发生了变化：我们把`PUT`——“把文档存储在这个地址中”变量变成了`POST`——“把文档存储在这个地址下”。

这样一来，请求中就只包含`_index`和`_type`了：

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

这次的反馈和之前基本一样，只有`_id`改成了系统生成的自增值:

```
{
   "_index":    "website",
   "_type":     "blog",
   "_id":       "wM0OSFhDQXGZAWDf0-drSA",
   "_version":  1,
   "created":   true
}
```

自生成ID是由22个字母组成的，安全 *universally unique identifiers* 或者被称为UUIDs。

搜索文档

要从Elasticsearch中获取文档，我们需要使用同样的`_index`，`_type`以及`_id`但是不同的HTTP变量`GET`：

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

返回结果包含了之前提到的内容，以及一个新的字段_source，它包含我们在最初创建索引时的原始JSON文档。

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

---

pretty

在任意的查询字符串中添加pretty参数，类似上面的请求，Elasticsearch就可以得到优美打印的更加易于识别的JSON结果。_source字段不会执行优美打印，它的样子取决于我们录入的样子。

GET请求的返回结果中包含{"found": true}。这意味着这篇文档确实被找到了。如果我们请求了一个不存在的文档，我们依然会得到JSON反馈，只是found的值会变为false。

同样，HTTP返回码也会由'200 OK'变为'404 Not Found'。我们可以在curl后添加-i，这样你就能得到反馈头文件：

```
curl -i -XGET /website/blog/124?pretty
```

反馈结果就会是这个样子：

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

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

检索文档中的一部分

通常，GET请求会将整个文档放入_source字段中一并返回。但是可能你只需要title字段。你可以使用_source得到指定字段。如果需要多个字段你可以使用逗号分隔：

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

现在_source字段中就只会显示你指定的字段：

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

或者你只想得到_source字段而不要其他的元数据，你可以这样请求：

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

这样结果就只返回:

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

检查文档是否存在

如果确实想检查一下文档是否存在，你可以试用HEAD来替代GET方法，这样就是会返回HTTP头文件：

```
curl -i -XHEAD /website/blog/123
```

如果文档存在，Elasticsearch将会返回200 OK的状态码：

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

如果不存在将会返回404 Not Found状态码：

```
curl -i -XHEAD /website/blog/124
```

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

当然，这个反馈只代表了你查询的那一刻文档不存在，但是不代表几毫秒后它是否存在，很可能与此同时，另一个进程正在穿件文档。

更新整个文档

在Documents中的文档是不可改变的。所以如果我们需要改变已经存在的文档，我们可以使用《索引》中提到的indexAPI来重新索引或者替换掉它：

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

在反馈中，我们可以发现Elasticsearch已经将_version数值增加了：

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

1. created被标记为 false是因为在同索引、同类型下已经存在同ID的文档。

在内部，Elasticsearch已经将旧文档标记为删除并且添加了新的文档。旧的文档并不会立即消失，但是你也无法访问他。Elasticsearch会在你继续添加更多数据的时候在后台清理已经删除的文件。

在本章的后面，我们将会在《局部更新》中介绍最新更新的API。这个API允许你修改局部，但是原理和下方的完全一样：

1. 从旧的文档中检索JSON
2. 修改它
3. 删除修的文档
4. i索引一个新的文档

唯一不同的是，使用了updateAPI你就不需要再使用get然后再操作index请求了。

创建一个文档

我们是如何确定当我们索引一个文档时，我们是创建了一个新的文档还是覆盖了一个已经存在的文档呢？

请牢记_index,_type以及_id组成了唯一的文档标记，所以为了确定我们创建的内容是全新的的最简单的方法就是使用POST方法，让Elasticsearch自动创建不同的_id：

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

然而，我们我们可能已经决定好了_id，所以我们需要告诉Elasticsearch只有当同_index，_type以及_id的文档不存在时才接受我们的请求。实现这个目的有两种方法，他们实质上是一样的，你可以选择你认为方便的那种：

第一种是在查询中中添加op_type参数：

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

或者在请求最后添加 /_create:

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

如果成功创建了新的文档，Elasticsearch将会返回常见的元数据以及201 Created的HTTP反馈码。

而如果存在同名文件，Elasticsearch将会返回一个409 Conflict的HTTP反馈码，以及如下方的错误信息：

```
{
  "error" : "DocumentAlreadyExistsException[[website][4] [blog][123]:
             document already exists]",
  "status" : 409
}
```

删除一个文档

删除文档的基本模式和之前的基本一样，只不过是需要更换成DELETE方法：

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

如果文档存在，那么Elasticsearch就会返回一个200 OK的HTTP相应码，返回的结果就会像下面展示的一样。请注意_version的数字已经增加了。

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

如果文档不存在，那么我们就会得到一个404 Not Found的响应码，返回的内容就会是这样的：

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

尽管文档并不存在（"found"值为false），但是_version的数值仍然增加了。这个。这个就是内部管理的一部分，它保证了我们在多个节点间的不同操作都被标记了正确的顺序。

正如我在《更新》一章中提到的，删除一个文档也不会立即生效，它只是被标记成已删除。Elasticsearch将会在你之后添加更多索引的时候才会在后台进行删除内容的清理。

处理冲突

当你使用索引API来更新一个文档时，我们先看到了原始文档，然后修改它，最后一次性地将整个新文档进行再次索引处理。Elasticsearch会根据请求发出的顺序来选择出最新的一个文档进行保存。但是，如果在你修改文档的同时其他人也发出了指令，那么他们的修改将会丢失。

很长时间以来，这其实都不是什么大问题。或许我们的主要数据还是存储在一个关系数据库中，而我们只是将为了可以搜索，才将这些数据拷贝到Elasticsearch中。或许发生多个人同时修改一个文件的概率很小，又或者这些偶然的数据丢失并不会影响到我们的正常使用。

但是有些时候如果我们丢失了数据就会出大问题。想象一下，如果我们使用Elasticsearch来存储一个网店的商品数量。每当我们卖出一件，我们就会将这个数量减少一个。

突然有一天，老板决定来个大促销。瞬间，每秒就产生了多笔交易。并行处理，多个进程来处理交易：

web_1中库存量的变化丢失的原因是web_2并不知道它所得到的库存量数据是是过期的。这样就会导致我们误认为还有很多货存，最终顾客就会对我们的行为感到失望。

当我们对数据修改得越频繁，或者在读取和更新数据间有越长的空闲时间，我们就越容易丢失掉我们的数据。

以下是两种能避免在并发更新时丢失数据的方法：

悲观并发控制（**PCC**）

这一点在关系数据库中被广泛使用。假设这种情况很容易发生，我们就可以组织对这一资源的访问。典型的例子就是当我们在读取一个数据前先锁定这一行，然后确保只有读取到数据的这个线程可以修改这一行数据。

乐观并发控制（**OCC**）

Elasticsearch所使用的。假设这种情况并不会经常发生，也不会去组织某一数据的访问。然而，如果基础数据在我们读取和写入的间隔中发生了变化，更新就会失败。这时候就由程序来决定如何处理这个冲突。例如，它可以重新读取新数据来进行更新，又或者它可以将这一情况直接反馈给用户。

乐观并发控制

Elasticsearch是分布式的。当文档被创建、更新或者删除时，新版本的文档就会被复制到集群中的其他节点上。Elasticsearch即是同步的又是异步的，也就是说复制的请求被平行发送出去，然后可能会混乱地到达目的地。这就需要一种方法能够保证新的数据不会被旧数据所覆盖。

我们在上文提到每当有索引、put和删除的操作时，无论文档有没有变化，它的_version都会增加。Elasticsearch使用_version来确保所有的改变操作都被正确排序。如果一个旧的版本出现在新版本之后，它就会被忽略掉。

我们可以利用_version的优点来确保我们程序修改的数据冲突不会造成数据丢失。我们可以按照我们的想法来指定_version的数字。如果数字错误，请求就是失败。

我们来创建一个新的博文:

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

反馈告诉我们这是一个新建的文档，它的_version是1。假设我们要编辑它，把这个数据加载到网页表单中，修改完毕然后保存新版本。

首先我们先要得到文档：

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

返回结果显示_version为1：

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

现在，我们试着重新索引文档以保存变化，我们这样指定了version的数字：

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

1. 我们只希望当索引中文档的_version是1时，更新才生效。

请求成功相应，返回内容告诉我们_version已经变成了2：

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

然而，当我们再执行同样的索引请求，并依旧指定version=1时，Elasticsearch就会返回一个409 Conflict的响应码，返回内容如下：

```
{
  "error" : "VersionConflictEngineException[[website][2] [blog][1]:
             version conflict, current [2], provided [1]]",
  "status" : 409
}
```

这里面指出了文档当前的_version数字是2，而我们要求的数字是1。

我们需要做什么取决于我们程序的需求。比如我们可以告知用户已经有其它人修改了这个文档，你应该再保存之前看一下变化。而对于上文提到的库存量问题，我们可能需要重新读取一下最新的文档，然后显示新的数据。

所有的有关于更新或者删除文档的API都支持version这个参数，有了它你就通过修改你的程序来使用乐观并发控制。

使用外部系统的版本

还有一种常见的情况就是我们还是使用其他的数据库来存储数据，而Elasticsearch只是帮我们检索数据。这也就意味着主数据库只要发生的变更，就需要将其拷贝到Elasticsearch中。如果多个进程同时发生，就会产生上文提到的那些并发问题。

如果你的数据库已经存在了版本号码，或者也可以代表版本的时间戳。这是你就可以在Elasticsearch的查询字符串后面添加version_type=external来使用这些号码。版本号码必须要是大于零小于9.2e+18（Java中long的最大正值）的整数。

Elasticsearch在处理外部版本号时会与对内部版本号的处理有些不同。它不再是检查_version是否与制定中的数组相同,而是检查当前的_version是否比指定的数值小。如果请求成功，那么外部的版本号就会被存储到文档中的_version中。

外部版本号不仅可以在索引和删除请求时使用，还可以在创建时使用。

例如，创建一篇使用外部版本号为5的博文，我们可以这样操作：

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

在返回结果中，我们可以发现_version是5：

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

现在我们更新这个文档，并指定version为10：

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

请求被成功执行并且version也变成了10：

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

如果你再次执行这个命令，你会得到之前的错误提示信息，因为你所指定的版本号并没有大于当前Elasticsearch中的版本号。

更新文档中的一部分

在《更新》一章中，我们讲到了要是想更新一个文档，那么就需要去取回数据，更改数据然后将整个文档进行重新索引。当然，你还可以通过使用更新API来做部分更新，比如增加一个计数器。

正如我们提到的，文档不能被修改，它们只能被替换掉。更新API也必须遵循这一法则。从表边看来，貌似是文档被替换了。对内而言，它必须按照找回-修改-索引的流程来进行操作与管理。不同之处在于这个流程是在一个片(shard) 中完成的，因此可以节省多个请求所带来的网络开销。除了节省了步骤，同时我们也能减少多个进程造成冲突的可能性。

使用更新请求最简单的一种用途就是添加新数据。新的数据会被合并到现有数据中，而如果存在相同的字段，就会被新的数据所替换。例如我们可以为我们的博客添加tags和views字段：

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

如果请求成功，我们就会收到一个类似于索引时返回的内容:

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

再次取回数据，你可以在_source中看到更新的结果：

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

1. 新的数据已经添加到了字段_source中。

使用脚本进行更新

我们将会在《脚本》一章中学习更详细的内容，我们现在只需要了解一些在Elasticsearch中使用API无法直接完成的自定义行为。默认的脚本语言叫做MVEL，但是Elasticsearch也支持JavaScript, Groovy 以及 Python。

MVEL是一个简单高效的JAVA基础动态脚本语言，它的语法类似于Javascript。你可以在Elasticsearch scripting docs 以及 MVEL website了解更多关于MVEL的信息。

脚本语言可以在更新API中被用来修改_source中的内容，而它在脚本中被称为ctx._source。例如，我们可以使用脚本来增加博文中views的数字：

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

我们同样可以使用脚本在tags数组中添加新的tag。在这个例子中，我们把新的tag生命为一个变量，而不是将他写死在脚本中。这样Elasticsearch就可以重新使用这个脚本进行tag的添加，而不用再次重新编写脚本了：

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

获取文档，后两项发生了变化：

```
{
   "_index":    "website",
   "_type":     "blog",
   "_id":       "1",
```

```
   "_version":  5,
   "found":     true,
   "_source": {
      "title":  "My first blog entry",
      "text":   "Starting to get the hang of this...",
      "tags":  ["testing", "search"], <1>
      "views":  1 <2>
   }
}
```

1. tags数组中出现了search。
2. views字段增加了。

我们甚至可以使用ctx.op来根据内容选择是否删除一个文档：

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

更新一篇可能不存在的文档

想象一下，我们可能需要在Elasticsearch中存储一个页面计数器。每次用户访问这个页面，我们就增加一下当前页面的计数器。但是如果这是个新的页面，我饿们不能确保这个计数器已经存在。如果我们试着去更新一个不存在的文档，更新操作就会失败。

为了防止上述情况的发生，我们可以使用upsert参数来设定文档不存在时，它应该被创建的内同：

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

首次运行这个请求时，upsert的内容会被索引成新的文档，它将views字段初始化为1。当之后再请求时，文档已经存在，所以脚本更新就会被执行，views计数器就会增加。

更新和冲突

在本节的开篇我们提到了当取回与重新索引两个步骤间的时间越少，发生改变冲突的可能性就越小。但它并不能被完全消除，在更新的过程中还能可能存在另一个进程进行重新索引的可能性。

为了避免丢失数据，更新API会在获取步骤中获取当前文档中的_version，然后将其传递给重新索引步骤中的索引请求。如果其他的进程在这两部之间修改了这个文档，那么_version就会不同，这样更新就会失败。

对于很多的局部更新来说，文档有没有发生变化实际上是不重要的。例如，两个进行都要增加页面浏览的计数器，谁先谁后其实并不重要 —— 发生冲突是之需要重新来过即可。

你可以通过设定retry_on_conflict参数来设置自动完成这项请求的次数，它的默认值是0。

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

1. 失败前重新尝试5次

这个参数非常适用于类似于增加技术起这种无关顺序的请求，但是还有些情况就属于顺序很重要的。例如上一节提到的情况，你可以参考乐观并发控制以及悲观并发控制来设定文档的版本号。

获取多个文档

索然Elasticsearch已经很高效了，但是它依旧可以更快。你可以将多个请求合并到一个请求中以节省网络开销。如果你需要从

Elasticsearch中获取多个文档，你可以使用*multi-get* 或者 `mget` API来取代一篇又一篇的获取。

`mget`API需要一个`docs`数组，每一个元素包含你想要的文档的`_index`, `_type`以及`_id`。你也可以指定`_source`参数来设定你所需要的字段：

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

这个请求包含了一个`docs`数组，其中的每一个参数都和《GET》一节中的方法相同：

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

如果你所需要的文档都在同一个`_index`或者同一个`_type`中，你就可以在URL中声明一个`/_index`或者`/_index`。你也可以在单独的请求中重写这个参数：

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

事实上，如果所有的文档拥有相同的`_index` 以及 `_type`，直接在请求中添加`ids`的数组即可：

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

请注意，我们所请求的第二篇文档不存在，这是就会返回如下内容：

```
{
```

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

1. 文档没有被找到。

当第二篇文档没有被找到的时候也不会影响到其它文档的获取结果。每一个文档都会被独立展示。

注意：上方请求的HTTP代码依旧是200尽管有的文档没有被找到。事实上，及时所有的文档都没有被找到，响应码也依旧是200。这是因为mget这个请求本身已经成功完成。要求定文档是否被成功找到，你需要检查一下found标识。

bulk更方便

In the same way that `mget` allows us to retrieve multiple documents at once, the `bulk` API allows us to make multiple `create`, `index`, `update` or `delete` requests in a single step. This is particularly useful if you need to index a data stream such as log events, which can be queued up and indexed in batches of hundreds or thousands.

The `bulk` request body has the following, slightly unusual, format:

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

This format is like a *stream* of valid one-line JSON documents joined together by newline `"\n"` characters. Two important points to note:

- Every line must end with a newline character `"\n"`, *including the last line*. These are used as markers to allow for efficient line separation.
- The lines cannot contain unescaped newline characters, as they would interfere with parsing -- that means that the JSON must *not* be pretty-printed.

TIP: In <> we explain why the `bulk` API uses this format.

The *action/metadata* line specifies *what action* to do to *which document*.

The *action* must be one of `index`, `create`, `update` or `delete`. The *metadata* should specify the `_index`, `_type` and `_id` of the document to be indexed, created, updated or deleted.

For instance, a `delete` request could look like this:

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

The *request body* line consists of the document `_source` itself -- the fields and values that the document contains. It is required for `index` and `create` operations, which makes sense: you must supply the document to index.

It is also required for `update` operations and should consist of the same request body that you would pass to the `update` API: `doc`, `upsert`, `script` etc. No *request body* line is required for a delete.

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

If no `_id` is specified, then an ID will be auto-generated:

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

To put it all together, a complete `bulk` request has this form:

```
POST /_bulk
{ "delete": { "_index": "website", "_type": "blog", "_id": "123" }} <1>
{ "create": { "_index": "website", "_type": "blog", "_id": "123" }}
{ "title":    "My first blog post" }
{ "index":  { "_index": "website", "_type": "blog" }}
{ "title":    "My second blog post" }
{ "update": { "_index": "website", "_type": "blog", "_id": "123", "_retry_on_conflict" : 3} }
{ "doc" : {"title" : "My updated blog post"} } <2>
```

1. Notice how the `delete` *action* does not have a *request body*, it is followed immediately by another *action*.
2. Remember the final newline character.

The Elasticsearch response contains the `items` array which lists the result of each request, in the same order as we requested them:

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

1. All sub-requests completed successfully.

Each sub-request is executed independently, so the failure of one sub-request won't affect the success of the others. If any of the requests fail, then the top-level `error` flag is set to `true` and the error details will be reported under the relevant request:

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

In the response we can see that it failed to `create` document `123` because it already exists, but the subsequent `index` request, also on document `123`, succeeded:

```
{
   "took": 3,
```

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

1. One or more requests has failed.
2. The HTTP status code for this request reports `409 CONFLICT`.
3. The error message explaining why the request failed.
4. The second request succeeded with an HTTP status code of `200 OK`.

That also means that `bulk` requests are not atomic -- they cannot be used to implement transactions. Each request is processed separately, so the success or failure of one request will not interfere with the others.

Don't repeat yourself

Perhaps you are batch indexing logging data into the same `index`, and with the same `type`. Having to specify the same metadata for every document is a waste. Instead, just as for the `mget` API, the `bulk` request accepts a default `/_index` or `/_index/_type` in the URL:

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

You can still override the `_index` and `_type` in the metadata line, but it will use the values in the URL as defaults:

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

How big is too big?

The entire bulk request needs to be loaded into memory by the node which receives our request, so the bigger the request, the less memory available for other requests. There is an optimal size of `bulk` request. Above that size, performance no longer improves and may even drop off.

The optimal size, however, is not a fixed number. It depends entirely on your hardware, your document size and complexity, and your indexing and search load. Fortunately, it is easy to find this *sweetspot*:

Try indexing typical documents in batches of increasing size. When performance starts to drop off, your batch size is too big. A good place to start is with batches of between 1,000 and 5,000 documents or, if your documents are very large, with even smaller batches.

It is often useful to keep an eye on the physical size of your bulk requests. One thousand 1kB documents is very different than one thousand 1MB documents. A good bulk size to start playing with is around 5-15MB in size.

总结

By now you know how to treat Elasticsearch as a distributed document store. You can store documents, update them, retrieve them and delete them, and you know how to do it safely. This is already very, very useful, even though we have not yet looked at the more exciting aspect of how to search *within* your documents. But first let's talk about the internal processes that Elasticsearch uses to manage your documents safely in a distributed environment.

Table of Contents