# 简介

Elasticsearch是一个开源的、实时的分布式搜索分析引擎， 它能让你以一个之前从未有过的速度和规模，去探索你的数据。 它被用作全文检索、结构化搜索、分析以及这三个功能的组合。

Elasticsearch是一个建立在全文搜索引擎 Apache Lucene™ 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，需要使用JAVA，并且在程序中集成Lucene。需要很多的学习了解，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，但是在使用它做全文搜索时，只需要使用统一开发好的API即可，而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。

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

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

Elasticsearch提供RESTful的API，数据以JSON格式为基础。

Elasticsearch 中没有一个单独的组件是全新的或者是革命性的。全文搜索很久之前就已经可以做到了， 就像早就出现了的分析系统和分布式数据库。 革命性的成果在于将这些单独的，有用的组件融合到一个单一的、一致的、实时的应用中。它对于初学者而言有一个较低的门槛， 而当你的技能提升或需求增加时，它也始终能满足你的需求。

目前，有以下组织在使用Elasticsearch：

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

然而 Elasticsearch 不仅仅为巨头公司服务。它也帮助了很多初创公司，像 Datadog 和 Klout， 帮助他们将想法用原型实现，并转化为可扩展的解决方案。Elasticsearch 能运行在你的笔记本电脑上，或者扩展到上百台服务器上去处理PB级数据。

你可以免费下载，使用，修改 Elasticsearch。它在 [Apache 2 license](http://www.apache.org/licenses/LICENSE-2.0.html) 协议下发布的， 这是众多灵活的开源协议之一。Elasticsearch 的源码被托管在 Github 上 [github.com/elastic/elasticsearch](https://github.com/elastic/elasticsearch)。 如果你想加入我们这个令人惊奇的 contributors 社区，看这里 [Contributing to Elasticsearch](https://github.com/elastic/elasticsearch/blob/master/CONTRIBUTING.md)。

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

官网：https://www.elastic.co/products/elasticsearch

# 安装

## Elasticsearch

安装 Elasticsearch 之前，你需要先安装一个较新的版本的 Java。之后，你可以从 elastic 的官网 [elastic.co/downloads/elasticsearch](https://www.elastic.co/downloads/elasticsearch) 获取最新版本的 Elasticsearch。

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

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

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

cd elasticsearch-<version>

./bin/elasticsearch

如果你想把 Elasticsearch 作为一个守护进程在后台运行，那么可以在后面添加参数 -d。

如果你是在 Windows 上面运行 Elasticseach，你应该运行 bin\elasticsearch.bat 而不是 bin\elasticsearch 。

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

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

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

1、下载elasticsearch，解压，运行bin目录下的elasticsearch.bat

2、浏览器访问http://localhost:9200/

## sense

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

1. 下载Kibana：<https://www.elastic.co/downloads/kibana>

由于官方下载速度很慢，这里提供kibana 5.4版本的百度网盘下载地址：

http://pan.baidu.com/s/1dERuDWX 密码：ttty

1. 在kibana目录执行：./bin/kibana plugin --install elastic/sense，windows上bin\kibana.bat plugin --install elastic/sense
2. 启动kibana：./bin/kibana，windows上为：.bin\kibana.bat

4、浏览器访问： http://localhost:5601/app/sense

5、By default, Kibana 4.2 requires an active Elasticsearch instance of version 2.0 or higher. If you want to use Sense without this requirement see instructions here.

离线安装sense

你可以直接从这里 <https://download.elastic.co/elastic/sense/sense-latest.tar.gz> 下载 Sense 离线安装可以查看这里 [install it on an offline machine](https://www.elastic.co/guide/en/sense/current/installing.html#manual_download) 。

注意：启动kibana之前，先确认是否已经启动elasticsearch，否则无法成功启动。

ERROR unknown command plugin

上边的安装步骤适用于kibana5之前的版本，kibana5之后sense改为了Console，而且默认已经安装，如下：

解压之后，启动：

E:\04-opensource\kibana-5.4.1-windows-x86>bin\kibana.bat

log [12:11:28.138] [info][optimize] Optimizing and caching bundles for kibana, timelion and status\_page. This may take a few minutes

log [12:12:49.611] [info][optimize] Optimization of bundles for kibana, timelion and status\_page complete in 81.47 seconds

log [12:12:49.656] [info][status][plugin:kibana@5.4.1] Status changed from uninitialized to green - Ready

log [12:12:49.744] [info][status][plugin:elasticsearch@5.4.1] Status changed from uninitialized to yellow - Waiting for Elasticsearch

log [12:12:49.765] [info][status][plugin:console@5.4.1] Status changed from uninitialized to green - Ready

log [12:12:49.773] [info][status][plugin:metrics@5.4.1] Status changed from uninitialized to green - Ready

log [12:12:49.944] [info][status][plugin:timelion@5.4.1] Status changed from uninitialized to green - Ready

log [12:12:49.948] [info][listening] Server running at http://localhost:5601

log [12:12:49.950] [info][status][ui settings] Status changed from uninitialized to yellow - Elasticsearch plugin is yellow

log [12:12:50.102] [warning] You're running Kibana 5.4.1 with some different versions of Elasticsearch. Update Kibana or Elasticsearch to the same version to prevent compatibility issues: v5.4.0 @ 127.0.0.1:9200 (127.0.0.1)

log [12:12:55.289] [info][status][plugin:elasticsearch@5.4.1] Status changed from yellow to yellow - No existing Kibana index found

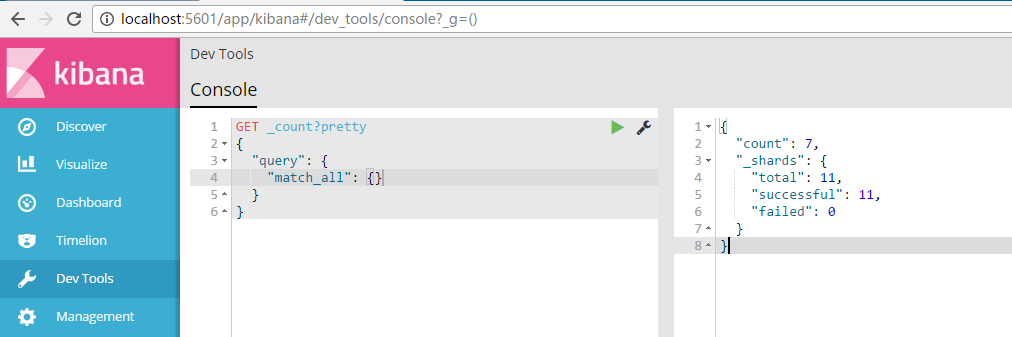
log [12:12:56.184] [info][status][plugin:elasticsearch@5.4.1] Status changed from yellow to green - Kibana index ready

log [12:12:56.186] [info][status][ui settings] Status changed from yellow to green – Ready

然后浏览器访问如下地址：

<http://localhost:5601>

点击左侧菜单的Dev Tools，既可以看到Console界面，点击Get Work按钮，进入Conosle工作界面，即可开始发送请求与ElasticSearch进行交互：



# 交互：

和 Elasticsearch 的交互方式取决于 你是否使用 Java。

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

## JAVA API

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

### 节点客户端（Node client）

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

### 传输客户端（Transport client）

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

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

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

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

该部分内容我们将在后边深入学习，现在，我们看看如何使用CURL来与Elasticsearch的RESTful API进行交互。

## RESTful API with JSON over HTTP

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

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

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

如上命令，其中使用尖括号括起来的部分详细解释如下：

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

例如，计算集群中文档的数量，我们可以用这个:

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

{

"query": {

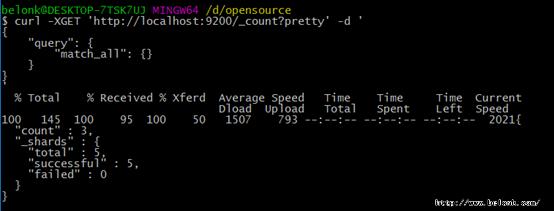
"match\_all": {}

}

}

'

执行情况如下：



如果在返回结果中我们需要看到HTTP头信息，需要使用**-i**参数，如下：

curl -i -XGET 'localhost:9200/'

Administrator@PC-201610162213 MINGW64 ~

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

% Total % Received % Xferd Average Speed Time Time Time Current

Dload Upload Total Spent Left Speed

100 327 100 327 0 0 2616 0 --:--:-- --:--:-- --:--:-- 2616HT TP/1.1 200 OK

content-type: application/json; charset=UTF-8

content-length: 327

{

"name" : "ZQuJdW6",

"cluster\_name" : "elasticsearch",

"cluster\_uuid" : "3kzN71MjSXG69l7O3CNaRQ",

"version" : {

"number" : "5.4.0",

"build\_hash" : "780f8c4",

"build\_date" : "2017-04-28T17:43:27.229Z",

"build\_snapshot" : false,

"lucene\_version" : "6.5.0"

},

"tagline" : "You Know, for Search"

}

## 缩写格式显示请求

Elasticsearch官方提供了一种缩写的请求标准格式，格式如下：

GET /\_count

{

"query": {

"match\_all": {}

}

}

官方推荐的sense工具直接支持以这种格式进行交互请求。sense介绍和安装见上一篇文章，这里不再赘述。

## 总结

本文介绍了如何使用CURL与Elasticsearch交互，这在初期学习Elasticsearch时很实用，更深入的JAVA客户端代码的交互方式我们放在后边来学习。

# 面向文档

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

# JSON格式

Elasticsearch 使用 JavaScript Object Notation 或者 JSON 作为文档的序列化格式。JSON 序列化被大多数编程语言所支持，并且已经成为 NoSQL 领域的标准格式。 它简单、简洁、易于阅读。在 Elasticsearch 中将对象转化为 JSON 并做索引要比在一个扁平的表结构中做相同的事情简单的多。

# 实战

## 需求

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

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

## 索引雇员文档

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

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

### 索引的含义

索引（名词）：

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

索引（动词）：

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

倒排索引：

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

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

### 索引雇员信息

对于雇员目录，我们将做如下操作：

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

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

PUT /megacorp/employee/1

{

"first\_name" : "John",

"last\_name" : "Smith",

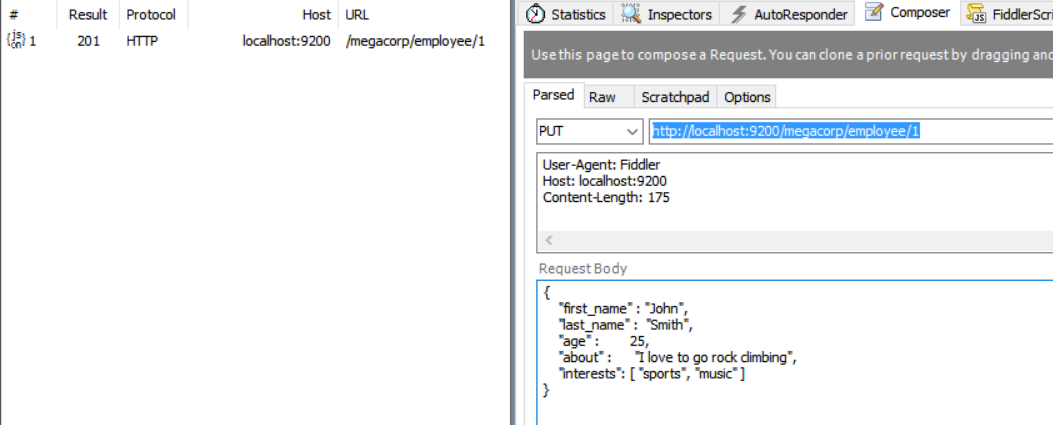
"age" : 25,

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

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

}

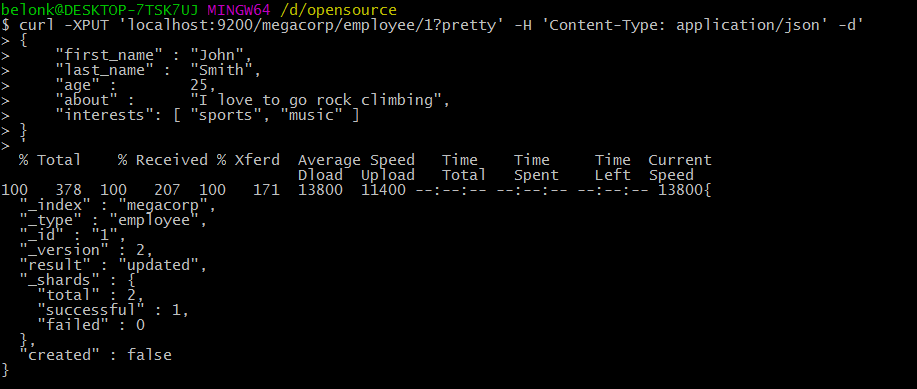
使用fiddler请求：



返回：

{"\_index":"megacorp","\_type":"employee","\_id":"1","\_version":1,"result":"created","\_shards":{"total":2,"successful":1,"failed":0},"created":true}

使用curl请求：



注意，路径 /megacorp/employee/1 包含了三部分的信息：

megacorp

索引名称

employee

类型名称

1

特定雇员的ID

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

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

进行下一步前，让我们增加更多的员工信息到目录中：

PUT /megacorp/employee/2

{

"first\_name" : "Jane",

"last\_name" : "Smith",

"age" : 32,

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

"interests": [ "music" ]

}

PUT /megacorp/employee/3

{

"first\_name" : "Douglas",

"last\_name" : "Fir",

"age" : 35,

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

"interests": [ "forestry" ]

}

### 检索文档

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

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

GET /megacorp/employee/1

$ curl -XGET 'localhost:9200/megacorp/employee/1?pretty'

% Total % Received % Xferd Average Speed Time Time Time Current

Dload Upload Total Spent Left Speed

100 294 100 294 0 0 18375 0 --:--:-- --:--:-- --:--:-- 287k{

"\_index" : "megacorp",

"\_type" : "employee",

"\_id" : "1",

"\_version" : 2,

"found" : true,

"\_source" : {

"first\_name" : "John",

"last\_name" : "Smith",

"age" : 25,

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

"interests" : [

"sports",

"music"

]

}

}

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

belonk@DESKTOP-7TSK7UJ MINGW64 /d/opensource

$ curl -XPUT -i 'http://localhost:9200/megacorp/employee/3' -d '

{

"first\_name" : "Douglas",

"last\_name" : "Fir",

"age" : 35,

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

"interests": [ "forestry1" ]

}

'

% Total % Received % Xferd Average Speed Time Time Time Current

Dload Upload Total Spent Left Speed

100 315 100 146 100 169 9125 10562 --:--:-- --:--:-- --:--:-- 10562HTTP/1.1 200 OK

Warning: 299 Elasticsearch-5.4.0-780f8c4 "Content type detection for rest requests is deprecated. Specify the content type using the [Content-Type] header." "Fri, 02 Jun 2017 10:07:01 GMT"

content-type: application/json; charset=UTF-8

content-length: 146

{"\_index":"megacorp","\_type":"employee","\_id":"3","\_version":3,"result":"updated","\_shards":{"total":2,"successful":1,"failed":0},"created":false}

belonk@DESKTOP-7TSK7UJ MINGW64 /d/opensource

$ curl -XGET -i 'http://localhost:9200/megacorp/employee/3?pretty'

% Total % Received % Xferd Average Speed Time Time Time Current

Dload Upload Total Spent Left Speed

100 281 100 281 0 0 281 0 0:00:01 --:--:-- 0:00:01 274kHTTP/1.1 200 OK

content-type: application/json; charset=UTF-8

content-length: 281

{

"\_index" : "megacorp",

"\_type" : "employee",

"\_id" : "3",

"\_version" : 3,

"found" : true,

"\_source" : {

"first\_name" : "Douglas",

"last\_name" : "Fir",

"age" : 35,

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

"interests" : [

"forestry1"

]

}

}

### 轻量搜索

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

第一个尝试的几乎是最简单的搜索了。我们使用下列请求来搜索所有雇员：

GET /megacorp/employee/\_search

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

belonk@DESKTOP-7TSK7UJ MINGW64 /d/opensource

$ curl -XGET 'localhost:9200/megacorp/employee/\_search?pretty'

% Total % Received % Xferd Average Speed Time Time Time Current

Dload Upload Total Spent Left Speed

100 1277 100 1277 0 0 8185 0 --:--:-- --:--:-- --:--:-- 8185{

"took" : 67,

"timed\_out" : false,

"\_shards" : {

"total" : 5,

"successful" : 5,

"failed" : 0

},

"hits" : {

"total" : 3,

"max\_score" : 1.0,

"hits" : [

{

"\_index" : "megacorp",

"\_type" : "employee",

"\_id" : "2",

"\_score" : 1.0,

"\_source" : {

"first\_name" : "Jane",

"last\_name" : "Smith",

"age" : 32,

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

"interests" : [

"music"

]

}

},

{

"\_index" : "megacorp",

"\_type" : "employee",

"\_id" : "1",

"\_score" : 1.0,

"\_source" : {

"first\_name" : "John",

"last\_name" : "Smith",

"age" : 25,

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

"interests" : [

"sports",

"music"

]

}

},

{

"\_index" : "megacorp",

"\_type" : "employee",

"\_id" : "3",

"\_score" : 1.0,

"\_source" : {

"first\_name" : "Douglas",

"last\_name" : "Fir",

"age" : 35,

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

"interests" : [

"forestry1"

]

}

}

]

}

}

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

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

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

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

belonk@DESKTOP-7TSK7UJ MINGW64 /d/opensource

$ curl -XGET 'localhost:9200/megacorp/employee/\_search?q=last\_name:Smith&pretty'

% Total % Received % Xferd Average Speed Time Time Time Current

Dload Upload Total Spent Left Speed

100 941 100 941 0 0 20021 0 --:--:-- --:--:-- --:--:-- 29406{

"took" : 29,

"timed\_out" : false,

"\_shards" : {

"total" : 5,

"successful" : 5,

"failed" : 0

},

"hits" : {

"total" : 2,

"max\_score" : 0.2876821,

"hits" : [

{

"\_index" : "megacorp",

"\_type" : "employee",

"\_id" : "2",

"\_score" : 0.2876821,

"\_source" : {

"first\_name" : "Jane",

"last\_name" : "Smith",

"age" : 32,

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

"interests" : [

"music"

]

}

},

{

"\_index" : "megacorp",

"\_type" : "employee",

"\_id" : "1",

"\_score" : 0.2876821,

"\_source" : {

"first\_name" : "John",

"last\_name" : "Smith",

"age" : 25,

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

"interests" : [

"sports",

"music"

]

}

}

]

}

}

### 使用查询表达式搜索

Elasticsearch 提供一个丰富灵活的查询语言叫做 *查询表达式* ， 它支持构建更加复杂和健壮的查询。

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

GET /megacorp/employee/\_search

{

"query" : {

"match" : {

"last\_name" : "Smith"

}

}

}

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

$ curl -XGET 'localhost:9200/megacorp/employee/\_search?pretty' -H 'Content-Type: application/json' -d'

> {

> "query" : {

> "match" : {

}

> "last\_name" : "Smith"

> }

> }

> }

> '

% Total % Received % Xferd Average Speed Time Time Time Current

Dload Upload Total Spent Left Speed

100 1031 100 940 100 91 15161 1467 --:--:-- --:--:-- --:--:-- 15161{

"took" : 4,

"timed\_out" : false,

"\_shards" : {

"total" : 5,

"successful" : 5,

"failed" : 0

},

"hits" : {

"total" : 2,

"max\_score" : 0.2876821,

"hits" : [

{

"\_index" : "megacorp",

"\_type" : "employee",

"\_id" : "2",

"\_score" : 0.2876821,

"\_source" : {

"first\_name" : "Jane",

"last\_name" : "Smith",

"age" : 32,

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

"interests" : [

"music"

]

}

},

{

"\_index" : "megacorp",

"\_type" : "employee",

"\_id" : "1",

"\_score" : 0.2876821,

"\_source" : {

"first\_name" : "John",

"last\_name" : "Smith",

"age" : 25,

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

"interests" : [

"sports",

"music"

]

}

}

]

}

}

### 更复杂的搜索

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

GET /megacorp/employee/\_search

{

"query" : {

"bool": {

"must": {

"match" : {

"last\_name" : "smith" // 这部分与我们之前使用的 match 查询 一样。

}

},

"filter": {

"range" : {

"age" : { "gt" : 30 } // 这部分是一个 range 过滤器 ， 它能找到年龄大于 30 的文档，其中 gt 表示\_大于(\_great than)。

}

}

}

}

}

$ curl -XGET 'localhost:9200/megacorp/employee/\_search?pretty' -H 'Content-Type: application/json' -d'

> {

> "query" : {

> "bool": {

> "must": {

> "match" : {

> "last\_name" : "smith"

> }

> },

> "filter": {

> "range" : {

> "age" : { "gt" : 30 }

> }

> }

> }

> }

> }

> '

% Total % Received % Xferd Average Speed Time Time Time Current

Dload Upload Total Spent Left Speed

100 871 100 563 100 308 3608 1974 --:--:-- --:--:-- --:--:-- 3608{

"took" : 117,

"timed\_out" : false,

"\_shards" : {

"total" : 5,

"successful" : 5,

"failed" : 0

},

"hits" : {

"total" : 1,

"max\_score" : 0.2876821,

"hits" : [

{

"\_index" : "megacorp",

"\_type" : "employee",

"\_id" : "2",

"\_score" : 0.2876821,

"\_source" : {

"first\_name" : "Jane",

"last\_name" : "Smith",

"age" : 32,

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

"interests" : [

"music"

]

}

}

]

}

}

### 全文搜索

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

搜索下所有喜欢攀岩（rock climbing）的雇员：

GET /megacorp/employee/\_search

{

"query" : {

"match" : {

"about" : "rock climbing"

}

}

}

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

Administrator@PC-201610162213 MINGW64 ~

$ curl -XGET 'localhost:9200/megacorp/employee/\_search?pretty' -H 'Content-Type: application/json' -d'

> {

> "query" : {

> "match" : {

> "about" : "rock climbing"

> }

> }

> }

> '

% Total % Received % Xferd Average Speed Time Time Time Current

Dload Upload Total Spent Left Speed

100 1039 100 944 100 95 29500 2968 --:--:-- --:--:-- --:--:-- 29500{

"took" : 17,

"timed\_out" : false,

"\_shards" : {

"total" : 5,

"successful" : 5,

"failed" : 0

},

"hits" : {

"total" : 2,

"max\_score" : 0.53484553,

"hits" : [

{

"\_index" : "megacorp",

"\_type" : "employee",

"\_id" : "1",

"\_score" : 0.53484553, // 相关性得分

"\_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" : 0.26742277, // 相关性得分

"\_source" : {

"first\_name" : "Jane",

"last\_name" : "Smith",

"age" : 32,

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

"interests" : [

"music"

]

}

}

]

}

}

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

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

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

### 短语搜索

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

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

GET /megacorp/employee/\_search

{

"query" : {

"match\_phrase" : {

"about" : "rock climbing"

}

}

}

毫无悬念，返回结果仅有 John Smith 的文档。

belonk@DESKTOP-7TSK7UJ MINGW64 ~

$ curl -XGET 'localhost:9200/megacorp/employee/\_search?pretty' -H 'Content-Type: application/json' -d'

> {

> "query" : {

> "match\_phrase" : {

> "about" : "rock climbing"

> }

> }

> }

> '

% Total % Received % Xferd Average Speed Time Time Time Current

Dload Upload Total Spent Left Speed

100 684 100 582 100 102 18774 3290 --:--:-- --:--:-- --:--:-- 36375{

"took" : 2,

"timed\_out" : false,

"\_shards" : {

"total" : 5,

"successful" : 5,

"failed" : 0

},

"hits" : {

"total" : 1,

"max\_score" : 0.53484553,

"hits" : [

{

"\_index" : "megacorp",

"\_type" : "employee",

"\_id" : "1",

"\_score" : 0.53484553,

"\_source" : {

"first\_name" : "John",

"last\_name" : "Smith",

"age" : 25,

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

"interests" : [

"sports",

"music"

]

}

}

]

}

}

### 高亮搜索

高亮搜索即是在检索结果中将匹配的内容高亮显示。在 Elasticsearch 中检索出高亮片段也很容易：

GET /megacorp/employee/\_search

{

"query" : {

"match\_phrase" : {

"about" : "rock climbing"

}

},

"highlight": {

"fields" : {

"about" : {}

}

}

}

当执行该查询时，中还多了一个叫做 highlight 的部分。这个部分包含了 about 属性匹配的文本片段，并以 HTML 标签em封装：

belonk@DESKTOP-7TSK7UJ MINGW64 ~

$ curl -XGET 'localhost:9200/megacorp/employee/\_search?pretty' -H 'Content-Type: application/json' -d'

> {

> "query" : {

> "match\_phrase" : {

> "about" : "rock climbing"

},

> }

> },

> "highlight": {

> "fields" : {

> "about" : {}

> }

> }

> }

> '

% Total % Received % Xferd Average Speed Time Time Time Current

Dload Upload Total Spent Left Speed

100 894 100 710 100 184 44375 11500 --:--:-- --:--:-- --:--:-- 693k{

"took" : 3,

"timed\_out" : false,

"\_shards" : {

"total" : 5,

"successful" : 5,

"failed" : 0

},

"hits" : {

"total" : 1,

"max\_score" : 0.53484553,

"hits" : [

{

"\_index" : "megacorp",

"\_type" : "employee",

"\_id" : "1",

"\_score" : 0.53484553,

"\_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>"

]

}

}

]

}

}

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

## 分析

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

举个例子，挖掘出雇员中最受欢迎的兴趣爱好：

GET /megacorp/employee/\_search

{

"aggs": {

"all\_interests": {

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

}

}

}

暂时忽略掉语法，直接看看结果：

belonk@DESKTOP-7TSK7UJ MINGW64 ~

$ curl -XGET 'localhost:9200/megacorp/employee/\_search?pretty' -H 'Content-Type: application/json' -d'

> {

> "aggs": {

> "all\_interests": {

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

> }

> }

> }

> '

% Total % Received % Xferd Average Speed Time Time Time Current

Dload Upload Total Spent Left Speed

100 1751 100 1661 100 90 21025 1139 --:--:-- --:--:-- --:--:-- 21025{

"took" : 68,

"timed\_out" : false,

"\_shards" : {

"total" : 5,

"successful" : 5,

"failed" : 0

},

"hits" : {

"total" : 3,

"max\_score" : 1.0,

"hits" : [

…… // 此处省略命中结果

]

},

"aggregations" : {

"all\_interests" : {

"doc\_count\_error\_upper\_bound" : 0,

"sum\_other\_doc\_count" : 0,

"buckets" : [

{

"key" : "music",

"doc\_count" : 2

},

{

"key" : "forestry1",

"doc\_count" : 1

},

{

"key" : "sports",

"doc\_count" : 1

}

]

}

}

}

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

GET /megacorp/employee/\_search

{

"query": {

"match": {

"last\_name": "smith"

}

},

"aggs": {

"all\_interests": {

"terms": {

"field": "interests"

}

}

}

}

all\_interests 聚合已经变为只包含匹配查询的文档：

belonk@DESKTOP-7TSK7UJ MINGW64 ~

$ curl -XGET 'localhost:9200/megacorp/employee/\_search?pretty' -H 'Content-Type: application/json' -d'

> {

> "query": {

> "match": {

> "last\_name": "smith"

> }

> },

> "aggs": {

> "all\_interests": {

> "terms": {

> "field": "interests"

> }

> }

> }

> }

> '

% Total % Received % Xferd Average Speed Time Time Time Current

Dload Upload Total Spent Left Speed

100 1417 100 1247 100 170 15987 2179 --:--:-- --:--:-- --:--:-- 15987{

"took" : 56,

"timed\_out" : false,

"\_shards" : {

"total" : 5,

"successful" : 5,

"failed" : 0

},

"hits" : {

……

]

},

"aggregations" : {

"all\_interests" : {

"doc\_count\_error\_upper\_bound" : 0,

"sum\_other\_doc\_count" : 0,

"buckets" : [

{

"key" : "music",

"doc\_count" : 2

},

{

"key" : "sports",

"doc\_count" : 1

}

]

}

}

}

聚合还支持分级汇总 。比如，查询特定兴趣爱好员工的平均年龄：

GET /megacorp/employee/\_search

{

"aggs" : {

"all\_interests" : {

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

"aggs" : {

"avg\_age" : {

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

}

}

}

}

}

得到的聚合结果有点儿复杂，但理解起来还是很简单的：

belonk@DESKTOP-7TSK7UJ MINGW64 ~

$ curl -XGET 'localhost:9200/megacorp/employee/\_search?pretty' -H 'Content-Type: application/json' -d'

> {

> "aggs" : {

> "all\_interests" : {

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

> "aggs" : {

> "avg\_age" : {

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

> }

> }

> }

> }

> }

> '

% Total % Received % Xferd Average Speed Time Time Time Current

Dload Upload Total Spent Left Speed

100 2099 100 1853 100 246 59774 7935 --:--:-- --:--:-- --:--:-- 59774{

"took" : 23,

"timed\_out" : false,

"\_shards" : {

"total" : 5,

"successful" : 5,

"failed" : 0

},

"hits" : {

"total" : 3,

"max\_score" : 1.0,

"hits" : [

……

]

},

"aggregations" : {

"all\_interests" : {

"doc\_count\_error\_upper\_bound" : 0,

"sum\_other\_doc\_count" : 0,

"buckets" : [

{

"key" : "music",

"doc\_count" : 2,

"avg\_age" : {

"value" : 28.5

}

},

{

"key" : "forestry1",

"doc\_count" : 1,

"avg\_age" : {

"value" : 35.0

}

},

{

"key" : "sports",

"doc\_count" : 1,

"avg\_age" : {

"value" : 25.0

}

}

]

}

}

}

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

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

### Fielddata is disabled on text fields by default

在测试上边的分析代码时，出现了一个400的错误信息：

Fielddata is disabled on text fields by default. Set fielddata=true on [your\_field\_name] in order to load fielddata in memory by uninverting the inverted index. Note that this can however use significant memory.

#### 原因

官方的关于fielddata的文档如下：

<https://www.elastic.co/guide/en/elasticsearch/reference/current/fielddata.html>

Fielddata can consume a **lot** of heap space, especially when loading high cardinality text fields. Once fielddata has been loaded into the heap, it remains there for the lifetime of the segment. Also, loading fielddata is an expensive process which can cause users to experience latency hits. This is why fielddata is disabled by default.

If you try to sort, aggregate, or access values from a script on a text field, you will see this exception:

Fielddata is disabled on text fields by default. Set fielddata=true on [your\_field\_name] in order to load fielddata in memory by uninverting the inverted index. Note that this can however use significant memory.

大概的意思是说：

Fielddata会耗费大量的堆内存，尤其是在加载大文件时。一旦fielddata被加载到内存中，他会在segment的生命周期一直存在。而且，搜索时处理fielddata还会导致速度下降甚至严重延迟，所以默认fielddata被禁用。

如果你试图对text字段进行排序、聚合或者访问值等操作，你将会遇到如下异常：

Fielddata is disabled on text fields by default. Set fielddata=true on [your\_field\_name] in order to load fielddata in memory by uninverting the inverted index. Note that this can however use significant memory.

#### 解决办法

由于我们是在测试，需要对文本字段开启fielddata，需要使用 [PUT mapping API](https://www.elastic.co/guide/en/elasticsearch/reference/current/indices-put-mapping.html)，如下命令:

PUT my\_index/\_mapping/my\_type

{

"properties": {

"my\_field": {

"type": "text",

"fielddata": true

}

}

}

其中my\_field字段是你需要进行排序、聚合等操作的包含现有映射的字段。

上边的实例我们开启fielddata如下：

belonk@DESKTOP-7TSK7UJ MINGW64 ~

$ curl -XPUT 'localhost:9200/megacorp/\_mapping/employee' -d '

{

"properties": {

"interests": {

"type": "text",

"fielddata": true

}

}

}'

% Total % Received % Xferd Average Speed Time Time Time Current

Dload Upload Total Spent Left Speed

100 123 100 21 100 102 338 1645 --:--:-- --:--:-- --:--:-- 1645{"acknowledged":true}

# 基本概念

## Near realtime(NRT)

## Cluster

## Node

## Index

## Type

## Document

## Shards & Replicas

# 基本操作

## 健康检查

## 创建索引

## 索引和查询文档

## 查询所有索引

## 删除索引

## 修改文档

## 删除文档

## 批量处理

# 数据检索

## 搜索API

## 查询语言

## 执行搜索

## 执行过滤

## 执行聚合

# 集群内的原理

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

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

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

本章将讲述如何按需配置集群、节点和分片，并在硬件故障时确保数据安全。

## 空集群

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

**图 1. 包含空内容节点的集群**



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

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

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

## 集群健康

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

GET /\_cluster/health

在一个不包含任何索引的空集群中，它将会有一个类似于如下所示的返回内容：

{

"cluster\_name": "elasticsearch",

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

"timed\_out": false,

"number\_of\_nodes": 1,

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

"active\_primary\_shards": 0,

"active\_shards": 0,

"relocating\_shards": 0,

"initializing\_shards": 0,

"unassigned\_shards": 0

}

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

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

green

所有的主分片和副本分片都正常运行。

yellow

所有的主分片都正常运行，但不是所有的副本分片都正常运行。

red

有主分片没能正常运行。

也可以使用\_cat命令来检查集群健康状况：

GET /\_cat/health?v

belonk@DESKTOP-7TSK7UJ MINGW64 ~

$ curl -XGET 'localhost:9200/\_cat/health?v&pretty'

epoch timestamp cluster status node.total node.data shards pri relo init unassign pending\_tasks max\_task\_wait\_time active\_shards\_percent

1496648088 15:34:48 my-application green 3 3 26 13 0 0 0 0 - 100.0%

列出集群中的节点：

GET /\_cat/nodes?v

belonk@DESKTOP-7TSK7UJ MINGW64 ~

$ curl -XGET 'localhost:9200/\_cat/nodes?v&pretty'

ip heap.percent ram.percent cpu load\_1m load\_5m load\_15m node.role master name

127.0.0.1 13 42 8 mdi - ZQuJdW6

127.0.0.1 15 42 9 mdi \* ZKbVKef

127.0.0.1 14 42 10 mdi - wwj6PD\_

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

## 添加索引

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

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

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

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

注意

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

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

在索引建立的时候就已经确定了主分片数，但是副本分片数可以随时修改。

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

PUT /blogs

{

"settings" : {

"number\_of\_shards" : 3,

"number\_of\_replicas" : 1

}

}

belonk@DESKTOP-7TSK7UJ MINGW64 ~

$ curl -XPUT 'localhost:9200/blogs?pretty' -H 'Content-Type: application/json' -d'

> {

> "settings" : {

> "number\_of\_shards" : 3,

> "number\_of\_replicas" : 1

> }

> }

> '

% Total % Received % Xferd Average Speed Time Time Time Current

Dload Upload Total Spent Left Speed

100 149 100 60 100 89 256 380 --:--:-- --:--:-- --:--:-- 380{

"acknowledged" : true,

"shards\_acknowledged" : true

}

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

**图 2. 拥有一个索引的单节点集群**



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

{

"cluster\_name": "elasticsearch",

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

"timed\_out": false,

"number\_of\_nodes": 1,

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

"active\_primary\_shards": 3,

"active\_shards": 3,

"relocating\_shards": 0,

"initializing\_shards": 0,

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

"delayed\_unassigned\_shards": 0,

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

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

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

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

}

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

belonk@DESKTOP-7TSK7UJ MINGW64 ~

$ curl -XGET 'localhost:9200/\_cluster/health?pretty'

% Total % Received % Xferd Average Speed Time Time Time Current

Dload Upload Total Spent Left Speed

100 469 100 469 0 0 29312 0 --:--:-- --:--:-- --:--:-- 458k{

"cluster\_name" : "elasticsearch",

"status" : "yellow",

"timed\_out" : false,

"number\_of\_nodes" : 1,

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

"active\_primary\_shards" : 13,

"active\_shards" : 13,

"relocating\_shards" : 0,

"initializing\_shards" : 0,

"unassigned\_shards" : 13,

"delayed\_unassigned\_shards" : 0,

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

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

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

"active\_shards\_percent\_as\_number" : 50.0

}

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

当前我们的集群是正常运行的，但是在硬件故障时有丢失数据的风险。

## 添加故障转移

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

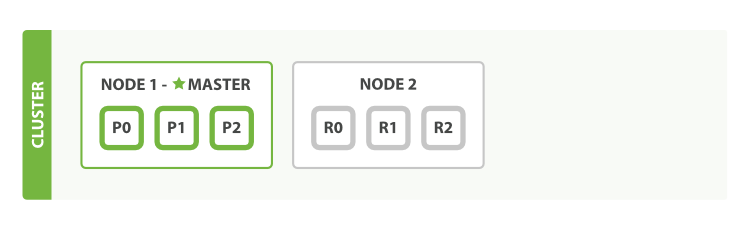
**启动第二个节点**

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

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

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

**图 3. 拥有两个节点的集群——所有主分片和副本分片都已被分配**



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

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

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

{

"cluster\_name": "elasticsearch",

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

"timed\_out": false,

"number\_of\_nodes": 2,

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

"active\_primary\_shards": 3,

"active\_shards": 6,

"relocating\_shards": 0,

"initializing\_shards": 0,

"unassigned\_shards": 0,

"delayed\_unassigned\_shards": 0,

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

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

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

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

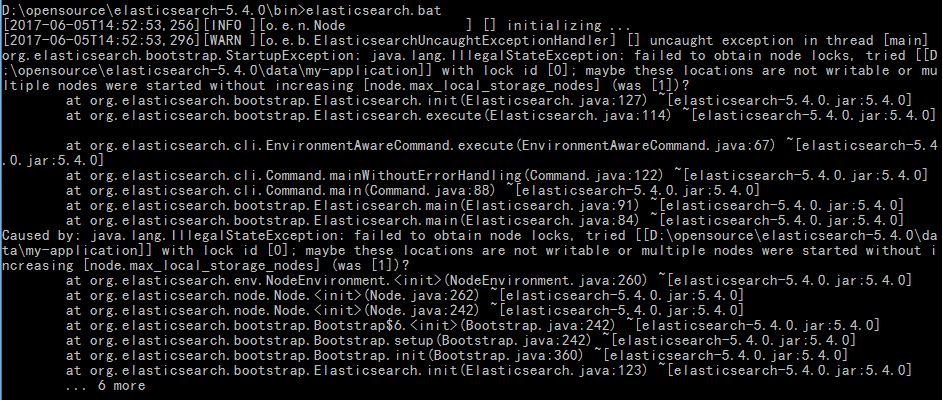
}

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

我们的集群现在不仅仅是正常运行的，并且还处于 始终可用 的状态。

### [elasticsearch start more than one nodes on the same data folder](http://blog.csdn.net/the_conquer_zzy/article/details/72818896)

在同一个目录下启动第二个节点时，出现如下异常：



解决方法是：在 config/elasticsearch.yml文件中新增一个配置变量：

node.max\_local\_storage\_nodes: 10

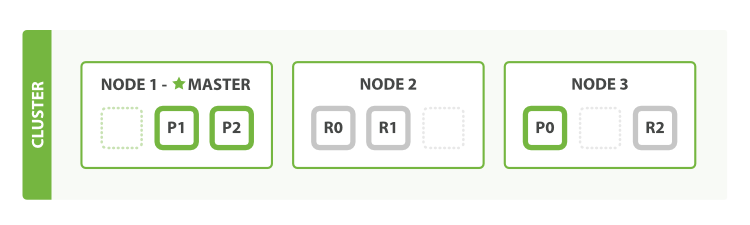
配置一个大于1的整数

[应对故障  »](https://www.elastic.co/guide/cn/elasticsearch/guide/current/_coping_with_failure.html)

## 水平扩容

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

**图 4. 拥有三个节点的集群——为了分散负载而对分片进行重新分配**



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

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

## 更多的扩容

但是如果我们想要扩容超过6个节点怎么办呢？

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

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

PUT /blogs/\_settings

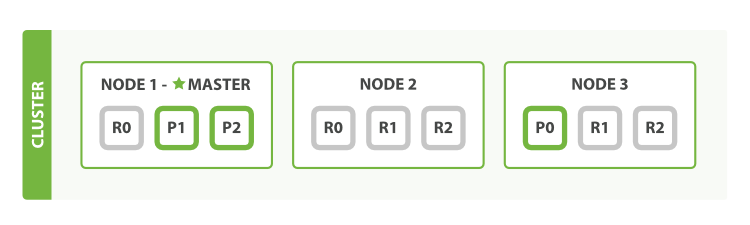
{

"number\_of\_replicas" : 2

}

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

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



注意

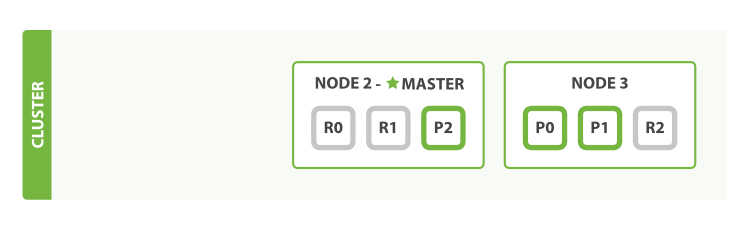
当然，如果只是在相同节点数目的集群上增加更多的副本分片并不能提高性能，因为每个分片从节点上获得的资源会变少。 你需要增加更多的硬件资源来提升吞吐量。

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

## 应对故障

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

**图 6. 关闭了一个节点后的集群**



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

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

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

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

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

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

# 数据输入和输出

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

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

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

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

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

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

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

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

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

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

## 什么是文档?

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

{

"name": "John Smith",

"age": 42,

"confirmed": true,

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

"home": {

"lat": 51.5,

"lon": 0.1

},

"accounts": [

{

"type": "facebook",

"id": "johnsmith"

},

{

"type": "twitter",

"id": "johnsmith"

}

]

}

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

警告

字段的名字可以是任何合法的字符串，但不可以包含时间段。

## 文档元数据

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

\_index

文档在哪存放

\_type

文档表示的对象类别

\_id

文档唯一标识

### \_index

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

提示

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

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

### \_type

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

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

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

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

### \_id

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

### 其他元数据

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

## 索引文档

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

### 使用自定义的 ID

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

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

{

"field": "value",

...

}

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

PUT /website/blog/123

{

"title": "My first blog entry",

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

"date": "2014/01/01"

}

Elasticsearch 响应体如下所示：

{

"\_index": "website",

"\_type": "blog",

"\_id": "123",

"\_version": 1,

"created": true

}

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

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

### Autogenerating IDs

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

现在该 URL 只需包含 \_index 和 \_type :

POST /website/blog/

{

"title": "My second blog entry",

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

"date": "2014/01/01"

}

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

{

"\_index": "website",

"\_type": "blog",

"\_id": "AVFgSgVHUP18jI2wRx0w",

"\_version": 1,

"created": true

}

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

## 取回一个文档

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

GET /website/blog/123?pretty

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

{

"\_index" : "website",

"\_type" : "blog",

"\_id" : "123",

"\_version" : 1,

"found" : true,

"\_source" : {

"title": "My first blog entry",

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

"date": "2014/01/01"

}

}

注意

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

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

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

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

显示响应头部的响应体现在类似这样：

HTTP/1.1 404 Not Found

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

Content-Length: 83

{

"\_index" : "website",

"\_type" : "blog",

"\_id" : "124",

"found" : false

}

### 返回文档的一部分

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

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

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

{

"\_index" : "website",

"\_type" : "blog",

"\_id" : "123",

"\_version" : 1,

"found" : true,

"\_source" : {

"title": "My first blog entry" ,

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

}

}

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

GET /website/blog/123/\_source

那么返回的的内容如下所示：

{

"title": "My first blog entry",

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

"date": "2014/01/01"

}

## 检查文档是否存在

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

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

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

HTTP/1.1 200 OK

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

Content-Length: 0

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

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

HTTP/1.1 404 Not Found

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

Content-Length: 0

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

belonk@DESKTOP-7TSK7UJ MINGW64 ~

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

Warning: Setting custom HTTP method to HEAD with -X/--request may not work the

Warning: way you want. Consider using -I/--head instead.

% Total % Received % Xferd Average Speed Time Time Time Current

Dload Upload Total Spent Left Speed

0 185 0 0 0 0 0 0 --:--:-- 0:00:33 --:--:-- 0

Git bash中-XHEAD请求不成功，出现警告信息，修改为-I参数。

belonk@DESKTOP-7TSK7UJ MINGW64 ~

$ curl -i -IHEAD http://localhost:9200/website/blog/123

% Total % Received % Xferd Average Speed Time Time Time Current

Dload Upload Total Spent Left Speed

0 185 0 0 0 0 0 0 --:--:-- --:--:-- --:--:-- 0HTTP/1.1 200 OK

content-type: application/json; charset=UTF-8

content-length: 185

belonk@DESKTOP-7TSK7UJ MINGW64 ~

$ curl -i -IHEAD "http://localhost:9200/website/blog/124"

% Total % Received % Xferd Average Speed Time Time Time Current

Dload Upload Total Spent Left Speed

0 61 0 0 0 0 0 0 --:--:-- --:--:-- --:--:-- 0HTTP/1.1 404 Not Found

content-type: application/json; charset=UTF-8

content-length: 61

## 更新整个文档

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

PUT /website/blog/123

{

"title": "My first blog entry",

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

"date": "2014/01/02"

}

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

{

"\_index" : "website",

"\_type" : "blog",

"\_id" : "123",

"\_version" : 2,

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

}

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

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

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

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

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

## 创建新文档

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

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

POST /website/blog/

{ ... }

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

第一种方法使用 op\_type 查询 -字符串参数：

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

{ ... }

第二种方法是在 URL 末端使用 /\_create :

PUT /website/blog/123/\_create

{ ... }

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

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

{

"error": {

"root\_cause": [

{

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

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

"shard": "0",

"index": "website"

}

],

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

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

"shard": "0",

"index": "website"

},

"status": 409

}

## 删除文档

删除文档的语法和我们所知道的规则相同，只是 使用 DELETE 方法：

DELETE /website/blog/123

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

{

"found" : true,

"\_index" : "website",

"\_type" : "blog",

"\_id" : "123",

"\_version" : 3

}

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

{

"found" : false,

"\_index" : "website",

"\_type" : "blog",

"\_id" : "123",

"\_version" : 4

}

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

注意

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

## 处理冲突

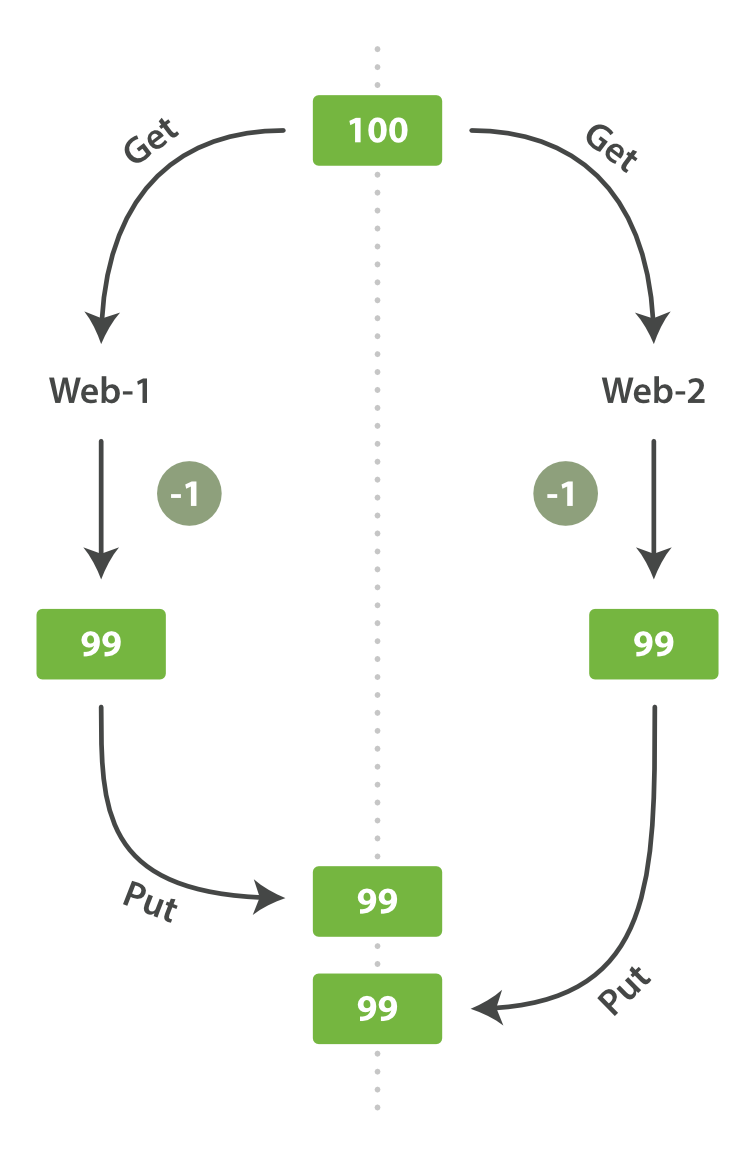
当我们使用 index API 更新文档 ，可以一次性读取原始文档，做我们的修改，然后重新索引 整个文档 。 最近的索引请求将获胜：无论最后哪一个文档被索引，都将被唯一存储在 Elasticsearch 中。如果其他人同时更改这个文档，他们的更改将丢失。

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

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

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

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



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

变更越频繁，读数据和更新数据的间隙越长，也就越可能丢失变更。

在数据库领域中，有两种方法通常被用来确保并发更新时变更不会丢失：

悲观并发控制

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

乐观并发控制

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

## 乐观并发控制

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

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

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

让我们创建一个新的博客文章：

PUT /website/blog/1/\_create

{

"title": "My first blog entry",

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

}

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

首先我们检索文档:

GET /website/blog/1

响应体包含相同的 \_version 版本号 1 ：

{

"\_index" : "website",

"\_type" : "blog",

"\_id" : "1",

"\_version" : 1,

"found" : true,

"\_source" : {

"title": "My first blog entry",

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

}

}

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

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

{

"title": "My first blog entry",

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

}

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

此请求成功，并且响应体告诉我们 \_version 已经递增到 2 ：

{

"\_index": "website",

"\_type": "blog",

"\_id": "1",

"\_version": 2

"created": false

}

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

{

"error": {

"root\_cause": [

{

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

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

"index": "website",

"shard": "3"

}

],

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

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

"index": "website",

"shard": "3"

},

"status": 409

}

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

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

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

### 通过外部系统使用版本控制

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

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

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

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

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

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

{

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

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

}

在响应中，我们能看到当前的 \_version 版本号是 5 ：

{

"\_index": "website",

"\_type": "blog",

"\_id": "2",

"\_version": 5,

"created": true

}

现在我们更新这个文档，指定一个新的 version 号是 10 ：

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

{

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

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

}

请求成功并将当前 \_version 设为 10 ：

{

"\_index": "website",

"\_type": "blog",

"\_id": "2",

"\_version": 10,

"created": false

}

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

## 文档的部分更新

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

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

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

POST /website/blog/1/\_update

{

"doc" : {

"tags" : [ "testing" ],

"views": 0

}

}

如果请求成功，我们看到类似于 index 请求的响应：

{

"\_index" : "website",

"\_id" : "1",

"\_type" : "blog",

"\_version" : 3

}

检索文档显示了更新后的 \_source 字段：

{

"\_index": "website",

"\_type": "blog",

"\_id": "1",

"\_version": 3,

"found": true,

"\_source": {

"title": "My first blog entry",

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

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

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

}

}

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

### 使用脚本部分更新文档

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

POST /website/blog/1/\_update

{

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

}

**用 Groovy 脚本编程**

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

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

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

script.groovy.sandbox.enabled: false

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

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

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

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

POST /website/blog/1/\_update

{

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

"params" : {

"new\_tag" : "search"

}

}

获取文档并显示最后两次请求的效果：

{

"\_index": "website",

"\_type": "blog",

"\_id": "1",

"\_version": 5,

"found": true,

"\_source": {

"title": "My first blog entry",

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

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

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

}

}

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

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

POST /website/blog/1/\_update

{

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

"params" : {

"count": 1

}

}

### 更新的文档可能尚不存在

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

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

POST /website/pageviews/1/\_update

{

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

"upsert": {

"views": 1

}

}

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

### 更新和冲突

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

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

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

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

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

{

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

"upsert": {

"views": 0

}

}

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

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

## 取回多个文档

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

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

GET /\_mget

{

"docs" : [

{

"\_index" : "website",

"\_type" : "blog",

"\_id" : 2

},

{

"\_index" : "website",

"\_type" : "pageviews",

"\_id" : 1,

"\_source": "views"

}

]

}

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

{

"docs" : [

{

"\_index" : "website",

"\_id" : "2",

"\_type" : "blog",

"found" : true,

"\_source" : {

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

"title" : "My first external blog entry"

},

"\_version" : 10

},

{

"\_index" : "website",

"\_id" : "1",

"\_type" : "pageviews",

"found" : true,

"\_version" : 2,

"\_source" : {

"views" : 2

}

}

]

}

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

你仍然可以通过单独请求覆盖这些值：

GET /website/blog/\_mget

{

"docs" : [

{ "\_id" : 2 },

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

]

}

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

GET /website/blog/\_mget

{

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

}

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

{

"docs" : [

{

"\_index" : "website",

"\_type" : "blog",

"\_id" : "2",

"\_version" : 10,

"found" : true,

"\_source" : {

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

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

}

},

{

"\_index" : "website",

"\_type" : "blog",

"\_id" : "1",

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

}

]

}

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

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

注意

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

## 代价较小的批量操作

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

bulk 与其他的请求体格式稍有不同，如下所示：

{ action: { metadata }}\n

{ request body }\n

{ action: { metadata }}\n

{ request body }\n

...

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

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

提示

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

action/metadata 行指定 哪一个文档 做 什么操作 。

action 必须是以下选项之一:

create

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

index

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

update

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

delete

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

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

例如，一个 delete 请求看起来是这样的：

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

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

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

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

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

如果不指定 \_id ，将会自动生成一个 ID ：

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

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

为了把所有的操作组合在一起，一个完整的 bulk 请求 有以下形式:

POST /\_bulk

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

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

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

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

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

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

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

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

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

{

"took": 4,

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

"items": [

{ "delete": {

"\_index": "website",

"\_type": "blog",

"\_id": "123",

"\_version": 2,

"status": 200,

"found": true

}},

{ "create": {

"\_index": "website",

"\_type": "blog",

"\_id": "123",

"\_version": 3,

"status": 201

}},

{ "create": {

"\_index": "website",

"\_type": "blog",

"\_id": "EiwfApScQiiy7TIKFxRCTw",

"\_version": 1,

"status": 201

}},

{ "update": {

"\_index": "website",

"\_type": "blog",

"\_id": "123",

"\_version": 4,

"status": 200

}}

]

}

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

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

POST /\_bulk

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

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

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

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

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

{

"took": 3,

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

"items": [

{ "create": {

"\_index": "website",

"\_type": "blog",

"\_id": "123",

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

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

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

document already exists]"

}},

{ "index": {

"\_index": "website",

"\_type": "blog",

"\_id": "123",

"\_version": 5,

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

}}

]

}

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

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

### 不要重复指定Index和Type

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

POST /website/\_bulk

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

{ "event": "User logged in" }

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

POST /website/log/\_bulk

{ "index": {}}

{ "event": "User logged in" }

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

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

### 多大是太大了？

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

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

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

# 分布式文档存储

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

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

**注意**

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

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

## 路由一个文档到一个分片中

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

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

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

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

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

注意

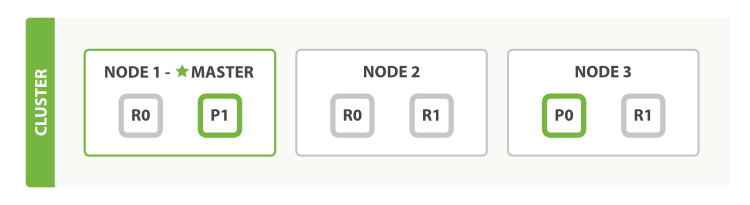
你可能觉得由于 Elasticsearch 主分片数量是固定的会使索引难以进行扩容。实际上当你需要时有很多技巧可以轻松实现扩容。我们将会在[扩容设计](https://www.elastic.co/guide/cn/elasticsearch/guide/current/scale.html)一章中提到更多有关水平扩展的内容。

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

## 主分片和副本分片如何交互

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

**图 8. 有三个节点和一个索引的集群**



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

提示

当发送请求的时候， 为了扩展负载，更好的做法是轮询集群中所有的节点。

## 新建、索引和删除文档

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

**图 9. 新建、索引和删除单个文档**



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

1. 客户端向 Node 1 发送新建、索引或者删除请求。
2. 节点使用文档的 \_id 确定文档属于分片 0。请求会被转发到 Node3，因为分片 0 的主分片目前被分配在 Node3 上。
3. Node 3 在主分片上面执行请求。如果成功了，它将请求并行转发到 Node 1 和 Node 2 的副本分片上。一旦所有的副本分片都报告成功, Node 3 将向协调节点报告成功，协调节点向客户端报告成功。

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

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

consistency

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

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

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

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

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

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

timeout

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

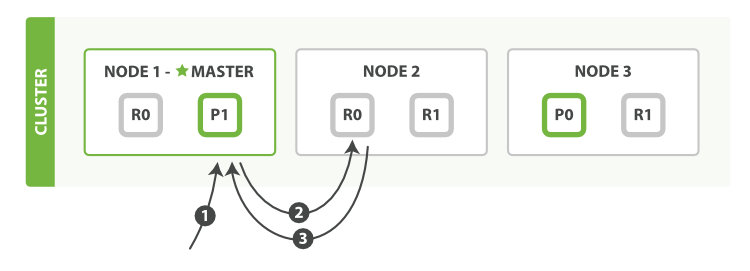
注意

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

## 取回一个文档

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

**图 10. 取回单个文档**



以下是从主分片或者副本分片检索文档的步骤顺序：

1、客户端向 Node 1 发送获取请求。

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

3、Node 2 将文档返回给 Node 1 ，然后将文档返回给客户端。

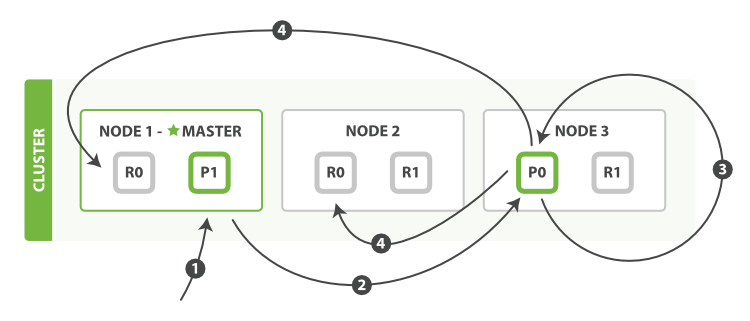
为了读取请求，协调节点在每次请求的时候将选择不同的副本分片来达到负载均衡；通过轮询所有的副本分片。

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

## 局部更新文档

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

**图 11. 局部更新文档**



以下是部分更新一个文档的步骤：

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

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

**基于文档的复制**

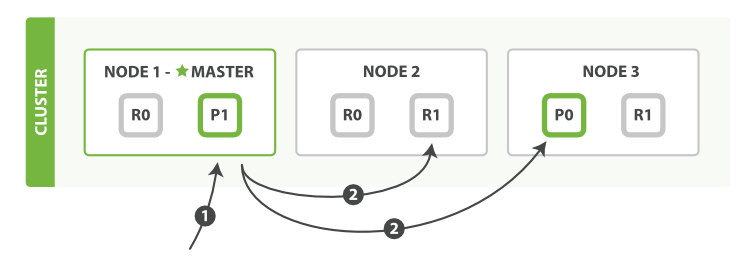
当主分片把更改转发到副本分片时， 它不会转发更新请求。 相反，它转发完整文档的新版本。请记住，这些更改将会异步转发到副本分片，并且不能保证它们以发送它们相同的顺序到达。 如果Elasticsearch仅转发更改请求，则可能以错误的顺序应用更改，导致得到损坏的文档。

## 多文档模式

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

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

**图 12. 使用 mget 取回多个文档**



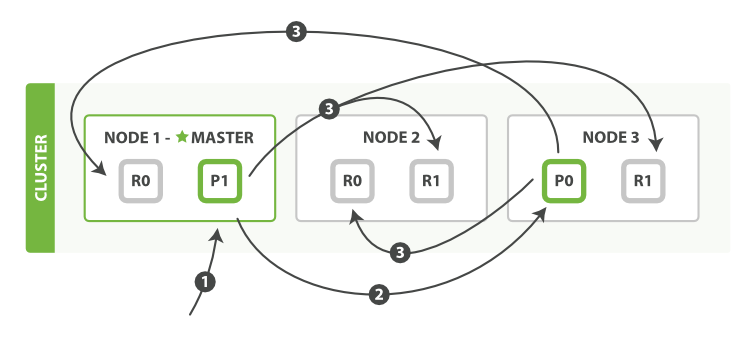
以下是使用单个 mget 请求取回多个文档所需的步骤顺序：

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

可以对 docs 数组中每个文档设置 routing 参数。

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

**图 13. 使用 bulk 修改多个文档**



bulk API 按如下步骤顺序执行：

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

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

### 为什么是有趣的格式？

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

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

如果单个请求被包装在 JSON 数组中，那就意味着我们需要执行以下操作：

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

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

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

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

# 搜索——最基本的工具

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

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

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

搜索（search） 可以做到：

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

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

映射（Mapping）

描述数据在每个字段内如何存储

分析（Analysis）

全文是如何处理使之可以被搜索的

领域特定查询语言（Query DSL）

Elasticsearch 中强大灵活的查询语言

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

我们将使用最简单的形式开始介绍 search API。

**测试数据**

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

你可以把这些命令复制到终端中执行来实践本章的例子。

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

## 空搜索

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

GET /\_search

返回的结果（为了界面简洁编辑过的）像这样：

{

"hits" : {

"total" : 14,

"hits" : [

{

"\_index": "us",

"\_type": "tweet",

"\_id": "7",

"\_score": 1,

"\_source": {

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

"name": "John Smith",

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

"user\_id": 2

}

},

... 9 RESULTS REMOVED ...

],

"max\_score" : 1

},

"took" : 4,

"\_shards" : {

"failed" : 0,

"successful" : 10,

"total" : 10

},

"timed\_out" : false

}

### hits

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

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

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

max\_score 值是与查询所匹配文档的 \_score 的最大值。

### took

took 值告诉我们执行整个搜索请求耗费了多少毫秒。

### shards

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

### timeout

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

GET /\_search?timeout=10ms

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

警告

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

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

## 多索引，多类型

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

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

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

/\_search

在所有的索引中搜索所有的类型

/gb/\_search

在 gb 索引中搜索所有的类型

/gb,us/\_search

在 gb 和 us 索引中搜索所有的文档

/g\*,u\*/\_search

在任何以 g 或者 u 开头的索引中搜索所有的类型

/gb/user/\_search

在 gb 索引中搜索 user 类型

/gb,us/user,tweet/\_search

在 gb 和 us 索引中搜索 user 和 tweet 类型

/\_all/user,tweet/\_search

在所有的索引中搜索 user 和 tweet 类型

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

提示

搜索一个索引有五个主分片和搜索五个索引各有一个分片准确来所说是等价的。

接下来，你将明白这种简单的方式如何灵活的根据需求的变化让扩容变得简单。

## 分页

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

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

size

显示应该返回的结果数量，默认是 10

from

显示应该跳过的初始结果数量，默认是 0

如果每页展示 5 条结果，可以用下面方式请求得到 1 到 3 页的结果：

GET /\_search?size=5

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

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

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

**在分布式系统中深度分页**

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

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

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

提示

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

## 轻量搜索

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

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

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

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

+name:john +tweet:mary

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

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

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

### \_all字段

这个简单搜索返回包含 mary 的所有文档：

GET /\_search?q=mary

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

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

Elasticsearch 是如何在三个不同的字段中查找到结果的呢？

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

{

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

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

"name": "Mary Jones",

"user\_id": 1

}

这就好似增加了一个名叫 \_all 的额外字段：

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

除非设置特定字段，否则查询字符串就使用 \_all 字段进行搜索。

提示

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

### 更复杂的查询

下面的查询针对tweents类型，并使用以下的条件：

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

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

查询字符串在做了适当的编码后，可读性很差：

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

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

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

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

提示

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

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

## 映射和分析

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

GET /\_search?q=2014 # 12 results

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

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

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

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

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

GET /gb/\_mapping/tweet

这将得到如下结果：

{

"gb": {

"mappings": {

"tweet": {

"properties": {

"date": {

"type": "date",

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

},

"name": {

"type": "string"

},

"tweet": {

"type": "string"

},

"user\_id": {

"type": "long"

}

}

}

}

}

}

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

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

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