

placeholder0

基 入

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

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

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

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

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

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

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

Elasticsearch 将成 最好的朋友。

知道的， 了搜索...

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

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

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

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

的形容：

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

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

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

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

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

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

回光

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

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

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

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

据，Shay 的妻子在等着的食搜索引擎 ...

安装并行 Elasticsearch

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

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

之后，可以从 elastic 的官 elastic.co/downloads/elasticsearch 取最新版本的 Elasticsearch。

要想安装 Elasticsearch，先下 并解 合 操作系 的 Elasticsearch 版本。如果 想了解更多的信息，可以 看 Elasticsearch 参考手 里 的安装部分，出的 接指向安装 明 {ref}/_installation.html[Installation]。

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

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

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

- ① 如果 想把 Elasticsearch 作 一个守 程在后台 行，那 可以在后面添加参数 **-d**。
- ② 如果 是在 Windows 上面 行 Elasticsearch， 行 **bin\elasticsearch.bat** 而不是 **bin\elasticsearch**。

Elasticsearch 是否 成功，可以打 一个 端， 行以下操作：

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

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

得到和下面 似的 (response)：

```
{
  "name" : "Tom Foster",
  "cluster_name" : "elasticsearch",
  "version" : {
    "number" : "2.1.0",
    "build_hash" : "72cd1f1a3eee09505e036106146dc1949dc5dc87",
    "build_timestamp" : "2015-11-18T22:40:03Z",
    "build_snapshot" : false,
    "lucene_version" : "5.3.1"
  },
  "tagline" : "You Know, for Search"
}
```

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

可以在 本 后面提供的生 部署章 到更多。

TIP : 看到下方的 View in Sense 的例子了 ? [Install the Sense console](#) 使用 自己的 Elasticsearch 集群去 行 本 中的例子, 看会有 的 果。

当 Elasticsearch 在前台 行 , 可以通 按 Ctrl+C 去停止。

安装 Sense

Sense 是一个 [Kibana](#) 用 它提供交互式的控制台, 通 的 器直接向 Elasticsearch 提交 求。本 的在 版本包含有一个 View in Sense 的 接, 里面有 多代 示例。当点 的 候, 它会打一个代 示例的Sense控制台。 不必安装 Sense, 但是它允 在本地的 Elasticsearch 集群上 示例代 , 从而使本 更具有交互性。

安装与 行 Sense :

1. 在 Kibana 目 下 行下面的命令, 下 并安装 Sense app :

```
./bin/kibana plugin --install elastic/sense ①
```

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

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

2. Kibana.

```
./bin/kibana ①
```

① Windows 上 kibana: `bin\kibana.bat`。

3. 在 的 器中打 Sense: <http://localhost:5601/app/sense>。

和 Elasticsearch 交互

和 Elasticsearch 的交互方式取决于 是否使用 Java

Java API

如果 正在使用 Java, 在代 中 可以使用 Elasticsearch 内置的 个客 端 :

点客 端 (*Node client*)

点客 端作 一个非数据 点加入到本地集群中。 句 , 它本身不保存任何数据, 但是它知道数据 在集群中的 个 点中, 并且可以把 求 到正 的 点。

客 端 (*Transport client*)

量 的 客 端可以将 求 送到 程集群。它本身不加入集群, 但是它可以将 求 到集群中的 一个 点上。

个 Java 客端都是通过 9300 端口并使用本地 Elasticsearch 和集群交互。集群中的点通过端口 9300 彼此通信。如果一个端口没有打开，点将无法形成一个集群。

TIP Java 客端作品必须和 Elasticsearch 有相同的主版本；否则，它们之间将无法互相理解。

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

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，可以用最喜欢的 web 客端与 Elasticsearch 交互。事实上，正如所看到的，甚至可以使用 curl 命令来和 Elasticsearch 交互。

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

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

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

被 <> 的部件：

VERB

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

PROTOCOL

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

HOST

Elasticsearch 集群中任意一点的主机名，或者用 localhost 代表本地机器上的点。

PORt

行 Elasticsearch HTTP 服务的端口号，是 9200。

PATH

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

QUERY_STRING

任意可选的字符串参数（例如 ?pretty 将格式化地输出 JSON 返回，使其更容易读）

BODY

一个 JSON 格式的请求体（如果请求需要的）

例如，算集群中文的数量，我可以使用一个：

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

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

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

在返回结果中没有看到 HTTP 信息是因为我没有要求`curl`显示它。想要看到这些信息，需要结合 -i 参数来使用 curl 命令：

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

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

curl示例，所有的写格式就是省略命令本身。而不是像下面所示的那个

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

我将用写格式表示：

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

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

面向文

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

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

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

JSON

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

考一下一个 JSON 文，它代表了一个 user 象：

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

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

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

新境

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

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

将深入介 里提到的所有概念。

接下来尽情享受 Elasticsearch 探索之旅。

建一个雇 目

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

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

索引雇 文

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

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

Index Versus Index Versus Index

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

索引（名）：

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

索引（ ）：

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

倒排索引：

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

+ 的，一个文 中的 一个属性都是 被索引 的（有一个倒排索引）和可搜索的。一个没有倒排索引的属性是不能被搜索到的。我 将在 [倒排索引](#) 倒排索引的更多 。

于雇 目 , 我 将做如下操作 :

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

践中 非常 (尽管看起来有很多) , 我 可以通 一条命令完成所有 些 作 :

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

注意, 路径 `/megacorp/employee/1` 包含了三部分的信息 :

megacorp

索引名称

employee

型名称

1

特定雇 的ID

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

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

行下一 前, 我 加更多的 工信息到目 中 :

```

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

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

```

索文

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

在 Elasticsearch 中很 。地行一个 HTTP GET 求并指定文 的地址——索引 、型和ID。使用 三个信息可以返回原始的 JSON 文：

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

返回果包含了文的一些元数据，以及 `_source` 属性，内容是 John Smith 雇员的原始 JSON 文：

```

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

```

TIP

将 HTTP 命令由 `PUT` 改为 `GET` 可以用来索文，同样的，可以使用 `DELETE` 命令来删除文档，以及使用 `HEAD` 指令来检查文档是否存在。如果想更新已存在的文档，只需再次 `PUT`。

批量搜索

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

第一个几乎是最简单的搜索了。我使用下列请求来搜索所有雇员：

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

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

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

注意：返回结果不告知匹配了哪些文本，包含了整个文本本身：示搜索结果最通用所需的全部信息。

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

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

我果然在请求路径中使用 `_search` 端点，并将本身参数 `q=`。返回结果给出了所有的 Smith：

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

使用表式搜索

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

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

```

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

```

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

更 的搜索

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

```

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

```

① 部分与我 之前使用的 **match** 一。

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

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

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

全文搜索

截止目前的搜索相 都很 : 个姓名, 通 年 。 在 下 微高 点儿的全文搜索———
数据 很 定的任 。

搜索下所有喜 岩 (rock climbing) 的雇 :

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

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

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

① 相 性得分

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

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

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

短 搜索

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

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

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

无 念，返回 果 有 John Smith 的文 。

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

高亮搜索

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

再次 行前面的 ，并 加一个新的 `highlight` 参数：

```

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

```

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

```

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

```

① 原始文本中的高亮片段

于高亮搜索片段，可以在[{ref}/search-request-highlighting.html\[highlighting reference documentation\]](#) 了解更多信息。

分析

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

个例子，掘出雇中最受迎的趣好：

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

忽略掉法，直接看看果：

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

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

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

all_interests 聚合已 只包含匹配 的文 :

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

聚合 支持分 。比如, 特定 趣 好 工的平均年 :

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

得到的聚合 果有点儿 ，但理解起来 是很 的：

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

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

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

教程

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

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

分布式特性

在本章 ，我 提到 Elasticsearch 可以横向 展至数百（甚至数千）的服 器 点，同 可以理PB 数据。我 的教程 出了一些使用 Elasticsearch 的示例，但并不 及任何内部机制。Elasticsearch 天生就是分布式的，并且在 屏蔽了分布式的 性。

Elasticsearch 在分布式方面几乎是透明的。教程中并不要求了解分布式系、分片、集群或其他的各 分布式概念。可以使用 本上的 点 松地 行教程里的程序，但如果 想要在 100 个 点的集群上 行程序，一切依然 。

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

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

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

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

后

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

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

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

集群内的原理

充章

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

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

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

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

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

本章将 述如何按需配置集群、 点和分片，并在硬件故障 保数据安全。

空集群

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

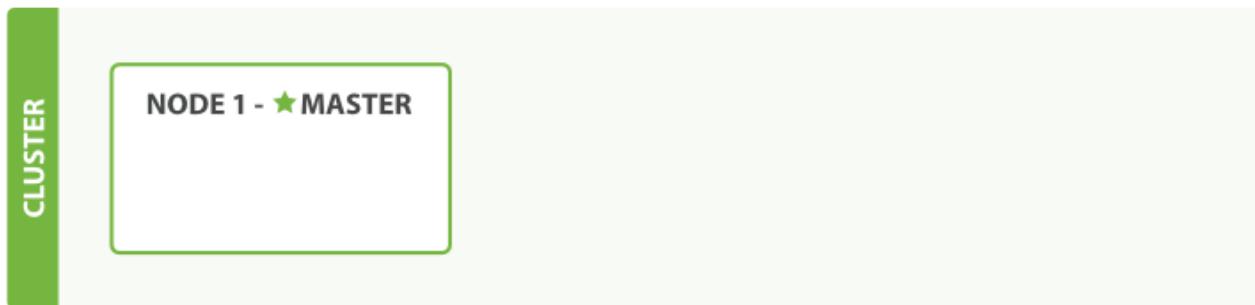


Figure 1. 包含空内容 点的集群

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

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

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

集群健康

Elasticsearch 的集群 控信息中包含了 多的 数据，其中最 重要的一 就是 集群健康 ， 它在
`status` 字段中展示 `green`、`yellow` 或者 `red`。

```
GET /_cluster/health
```

在一个不包含任何索引的空集群中，它将会有一个 似于如下所示的返回内容：

```
{
  "cluster_name": "elasticsearch",
  "status": "green", ①
  "timed_out": false,
  "number_of_nodes": 1,
  "number_of_data_nodes": 1,
  "active_primary_shards": 0,
  "active_shards": 0,
  "relocating_shards": 0,
  "initializing_shards": 0,
  "unassigned_shards": 0
}
```

① **status** 字段是我最关心的。

status 字段指示着当前集群在体上是否工作正常。它的三色含义如下：

green

所有的主分片和副本分片都正常运行。

yellow

所有的主分片都正常运行，但不是所有的副本分片都正常运行。

red

有主分片没能正常运行。

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

添加索引

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

一个分片是一个底层的工作单元，它保存了全部数据中的一部分。在**分片内部机制**中，我将介绍分片是如何工作的，而在我只需知道一个分片是一个 Lucene 的例子，以及它本身就是一个完整的搜索引擎。我的文章被存储和索引到分片内，但是用程序是直接与索引而不是与分片进行交互。

Elasticsearch 是利用分片将数据分到集群内各的。分片是数据的容器，文保存在分片内，分片又被分配到集群内的各个点里。当的集群模大或者小，Elasticsearch 会自动的在各点中移分片，使得数据然均匀分布在集群里。

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

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

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

回文 等 操作提供服 。

在索引建立的 候就已 定了主分片数，但是副本分片数可以随 修改。

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

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

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

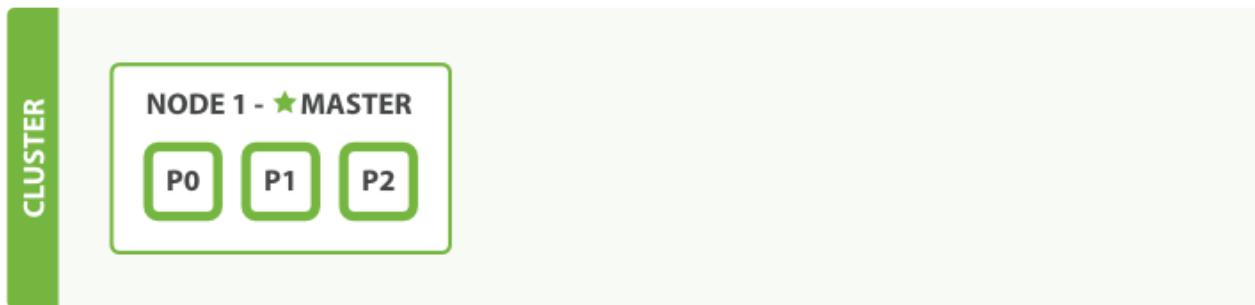


Figure 2. 有一个索引的 点集群

如果我 在 看**集群健康**，我 将看到如下内容：

```
{
  "cluster_name": "elasticsearch",
  "status": "yellow", ①
  "timed_out": false,
  "number_of_nodes": 1,
  "number_of_data_nodes": 1,
  "active_primary_shards": 3,
  "active_shards": 3,
  "relocating_shards": 0,
  "initializing_shards": 0,
  "unassigned_shards": 3, ②
  "delayed_unassigned_shards": 0,
  "number_of_pending_tasks": 0,
  "number_of_in_flight_fetch": 0,
  "task_max_waiting_in_queue_millis": 0,
  "active_shards_percent_as_number": 50
}
```

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

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

当前我 的集群是正常 行的, 但是在硬件故障 有 失数据的 。

添加故障 移

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

第二个 点

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

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

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

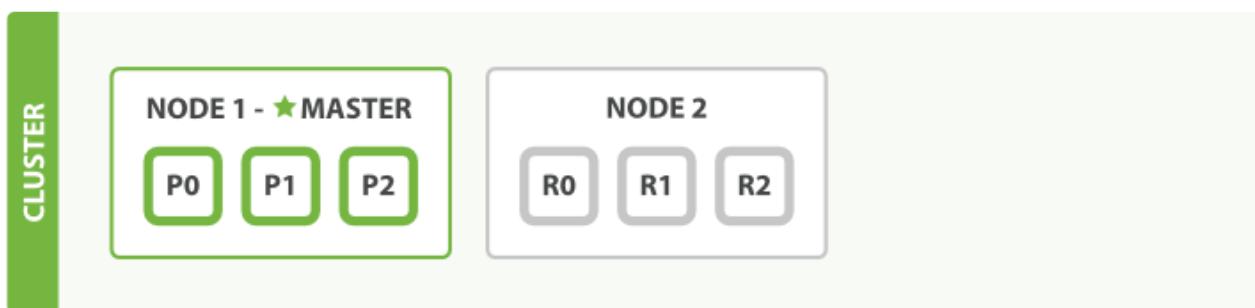


Figure 3. 有 个 点的集群——所有主分片和副本分片都已被分配

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

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

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

```
{
  "cluster_name": "elasticsearch",
  "status": "green", ①
  "timed_out": false,
  "number_of_nodes": 2,
  "number_of_data_nodes": 2,
  "active_primary_shards": 3,
  "active_shards": 6,
  "relocating_shards": 0,
  "initializing_shards": 0,
  "unassigned_shards": 0,
  "delayed_unassigned_shards": 0,
  "number_of_pending_tasks": 0,
  "number_of_in_flight_fetch": 0,
  "task_max_waiting_in_queue_millis": 0,
  "active_shards_percent_as_number": 100
}
```

① 集群 status green。

我的集群 在不行的，并且于始可用的状态。

水平 容

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

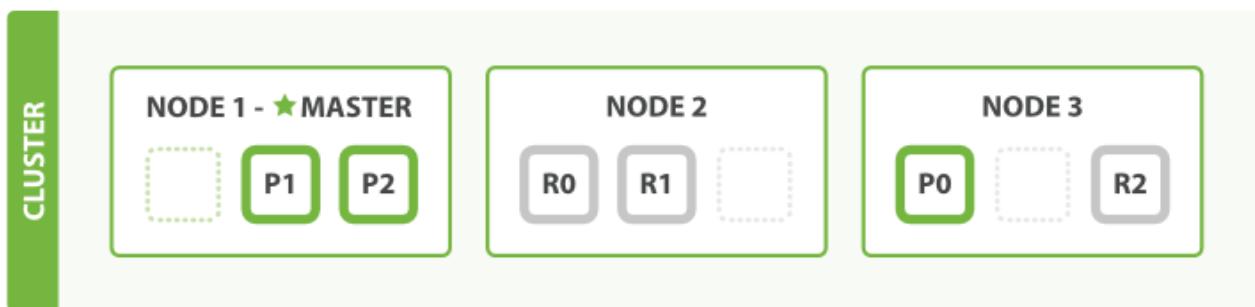


Figure 4. 有三个点的集群——了分散而分片行重新分配

Node 1 和 Node 2 上各有一个分片被移到了新的 Node 3 点，在个点上都 有2个分片，而不是之前的3个。表示个点的硬件源(CPU, RAM, I/O)将被更少的分片所共享，个分片的性能将会得到提升。

分片是一个功能完整的搜索引，它有使用一个点上的所有源的能力。我一个有6个分片(3个主分片和3个副本分片)的索引可以最大容到6个点，一个点上存在一个分片，并且一个分片有所在点的全部源。

更多的 容

但是如果我 想要 容超 6个 点 ?

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

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

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

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

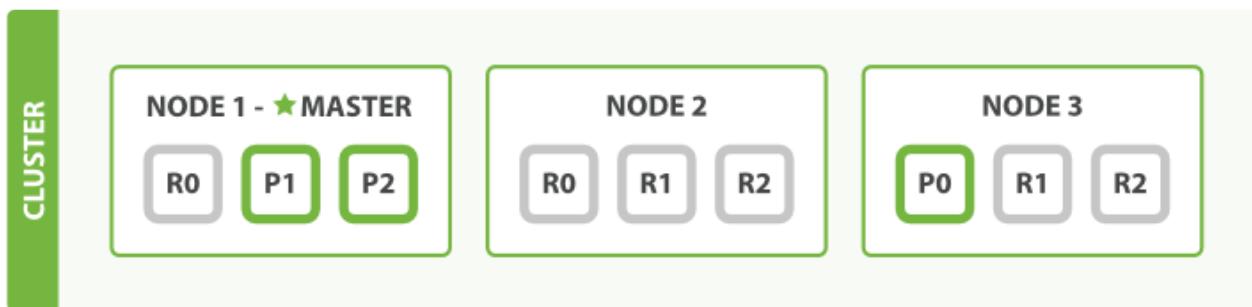


Figure 5. 将参数 `number_of_replicas` 大到 2

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

NOTE

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

故障

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

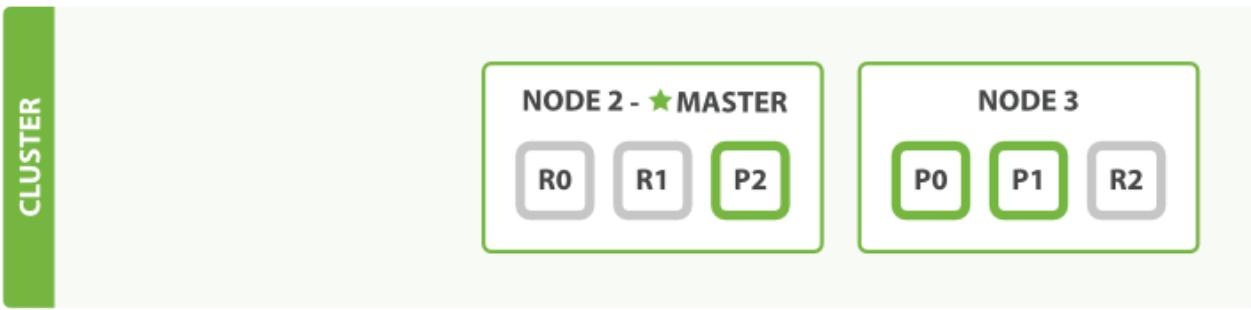


Figure 6. 了一个 点后的集群

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

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

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

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

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

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

数据 入和 出

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

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

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

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

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

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

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

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

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

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

什 是文 ？

在大多数 用中，多数 体或 象可以被序列化 包含 的 JSON 象。 一个 可以是一个字段或字段的名称，一个 可以是一个字符串，一个数字，一个布 ， 一个 象，一些数 ， 或一些其它特殊 型 如表示日期的字符串，或代表一个地理位置的 象：

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

通常情况下，我使用的象和文是可以互相替换的。不，有一个区别：一个象是似于 hash、hashmap、字典或者数的 JSON 象，象中也可以嵌套其他的象。象可能包含了外一些象。在 Elasticsearch 中，文有着特定的含义。它是指最或者根象，一个根象被序列化成 JSON 并存到 Elasticsearch 中，指定了唯一 ID。

WARNING 字段的名字可以是任何合法的字符串，但不可以包含段。

文元数据

一个文不包含它的数据，也包含元数据。有文的信息。三个必有的元数据元素如下：

_index

文在存放

_type

文表示的象

_id

文唯一

_index

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

TIP

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

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

_type

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

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

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

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

`_id`

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

其他元数据

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

索引文

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

使用自定 的 ID

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

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

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

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

Elasticsearch 体如下所示：

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

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

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

Autogenerating IDs

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

在 URL 只需包含 `_index` 和 `_type`：

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

除了 `_id` 是 Elasticsearch 自 生成的， 的其他部分和前面的 似：

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

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

取回一个文

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

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

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

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

NOTE

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

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

我可以通 `-i` 参数 `curl` 命令，参数能示的部：

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

示部的体在似：

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

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

返回文的一部分

情况下，`GET` 求会返回整个文，一个文正如存 在 `_source` 字段中的一。但是也只其中的 `title` 字段感 趣。一个字段能用 `_source` 参数求得到，多个字段也能使用逗号分隔的列表来指定。

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

`_source` 字段 在包含的只是我 求的那些字段，并且已 将 `date` 字段 掉了。

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

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

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

那 返回的的内容如下所示：

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

文 是否存在

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

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

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

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

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

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

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

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

更新整个文

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

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

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

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

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

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

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

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

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

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

建新文

当我 索引一个文 ， 我 正在 建一个完全新的文 ， 而不是覆 有的 ？

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

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

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

第一 方法使用 `op_type` -字符串参数：

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

第二 方法是在 URL 末端使用 `_create`：

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

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

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

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

除文

除文 的 法和我 所知道的 相同，只是使用 **DELETE** 方法：

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

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

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

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

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

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

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

理冲突

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

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

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

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

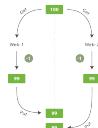


Figure 7. Consequence of no concurrency control

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

更越 繁, 数据和更新数据的 隙越 , 也就越可能 失 更。

在数据 域中, 有 方法通常被用来 保并 更新 更不会 失 :

悲 并 控制

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

并 控制

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

并 控制

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

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

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

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

我 建一个新的博客文章 :

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

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

首先我 索文 :

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

体包含相同的 `_version` 版本号 1 :

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

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

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

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

此 求成功, 并且 体告 我 `_version` 已 到 2 :

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

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

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

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

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

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

通 外部系 使用版本控制

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

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

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

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

外部版本号不 在索引和 除 求是可以指定，而且在 建新文 也可以指定。

例如，要 建一个新的具有外部版本号 5 的博客文章，我 可以按以下方法 行：

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

在 中，我 能看到当前的 `_version` 版本号是 5：

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

在我 更新 个文 ，指定一个新的 `version` 号是 10：

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

求成功并将当前 `_version` 10：

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

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

文 的部分更新

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

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

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

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

如果 求成功，我 看到 似于 `index` 求的：

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

索文 示了更新后的 `_source` 字段：

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

① 新的字段已被添加到 `_source` 中。

使用脚本部分更新文

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

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

用 Groovy 脚本 程

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

的脚本 言 是 [Groovy](#)，一 快速表 的脚本 言，在 法上与 JavaScript 似。 它在 Elasticsearch V1.3.0 版本首次引入并 行在 沙 中，然而 Groovy 脚本引 存在漏洞，允 攻者通 建 Groovy 脚本，在 Elasticsearch Java VM 行 脱 沙 并 行 shell 命令。

因此，在版本 v1.3.8 、 1.4.3 和 V1.5.0 及更高的版本中，它已 被 禁用。 此外， 可以通 置集群中的所有 点的 [config/elasticsearch.yml](#) 文件来禁用 Groovy 脚本：

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

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

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

可以在 [{ref}/modules-scripting.html\[scripting reference documentation\]](#) 取更多 于脚本的 料。

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

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

取文 并 示最后 次 求的效果：

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

① `search` 已追加到 `tags` 数 中。

② `views` 字段已 。

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

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

更新的文 可能尚不存在

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

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

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

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

更新和冲突

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

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

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

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

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

①失之前重更新5次。

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

取回多个文

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

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

```

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

```

体也包含一个 `docs` 数 , 于一个在 求中指定的文 , 个数 中都包含有一个 的 , 且 序与 求中的 序相同。 其中的 一个 都和使用 个 `get request` 求所得到的 体相同 :

```

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

```

如果想 索的数据都在相同的 `_index` 中 (甚至相同的 `_type` 中) , 可以在 URL 中指定 的 `/_index`

或者 的 `/_index/_type`。

然可以通 独 求覆 些 :

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

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

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

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

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

① 未 到 文 。

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

NOTE

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

代 小的批量操作

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

bulk 与其他的 求体格式 有不同，如下所示：

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

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

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

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

action/metadata 行指定 一个文 做什 操作。

action 必 是以下 之一：

create

如果文 不存在，那 就 建它。 情 建新文 。

index

建一个新文 或者替 一个 有的文 。 情 索引文 和 更新整个文 。

update

部分更新一个文 。 情 文 的部分更新。

delete

除一个文 。 情 除文 。

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

例如，一个 delete 求看起来是 的：

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

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

操作所必需的，是有道理的：必提供文以索引。

它也是 `update` 操作所必需的，并且包含 update API 的相同求体：`doc`、`upsert`、`script` 等等。除操作不需要 request body 行。

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

如果不指定 `_id`，将会自动生成一个 ID：

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

了把所有的操作合在一起，一个完整的 `bulk` 求有以下形式：

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

① 注意 `delete` 作不能有求体，它后面跟着的是外一个操作。

② 最后一个行符不要落下。

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

```
{
  "took": 4,
  "errors": false, ①
  "items": [
    { "delete": {
        "_index": "website",
        "_type": "blog",
        "_id": "123",
        "_version": 2,
        "status": 200,
        "found": true
      }},
    { "create": {
        "_index": "website",
        "_type": "blog",
        "_id": "123",
        "_version": 3,
        "status": 201
      }},
    { "create": {
        "_index": "website",
        "_type": "blog",
        "_id": "EiwfApScQiy7TIKFxRCTw",
        "_version": 1,
        "status": 201
      }},
    { "update": {
        "_index": "website",
        "_type": "blog",
        "_id": "123",
        "_version": 4,
        "status": 200
      }}
  ]
}
```

① 所有的子 求都成功完成。

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

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

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

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

① 一个或者多个 求失 。

② 个 求的HTTP状 告 409 CONFLICT。

③ 解 什 求失 的 信息。

④ 第二个 求成功, 返回 HTTP 状 200 OK。

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

不要重 指定Index和Type

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

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

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

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

多大是太大了？

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

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

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

分布式文 存

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

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

注意

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

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

路由一个文 到一个分片中

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

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

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

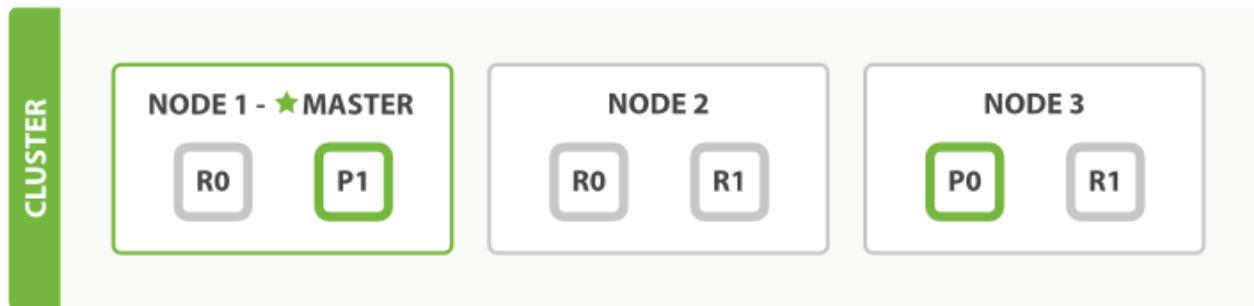
`routing` 是一个可选参数，是文档的 `_id`，也可以设置成一个自定义的。`routing` 通过 `hash` 函数生成一个数字，然后这个数字再除以 `number_of_primary_shards`（主分片的数量）后得到余数。一个分布在 0 到 `number_of_primary_shards-1` 之间的余数，就是我所求的文档所在分片的位置。

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

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

所有的 API（`get`、`index`、`delete`、`bulk`、`update` 以及 `mget`）都接受一个叫做 `routing` 的路由参数，通过这个参数我可以自定义文档到分片的映射。一个自定义的路由参数可以用来保证所有相同的文档——例如所有属于同一个用户的文档——都被存储到同一个分片中。我也会在 [scale] 一章中讨论什么会有这一需求。

主分片和副本分片如何交互

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

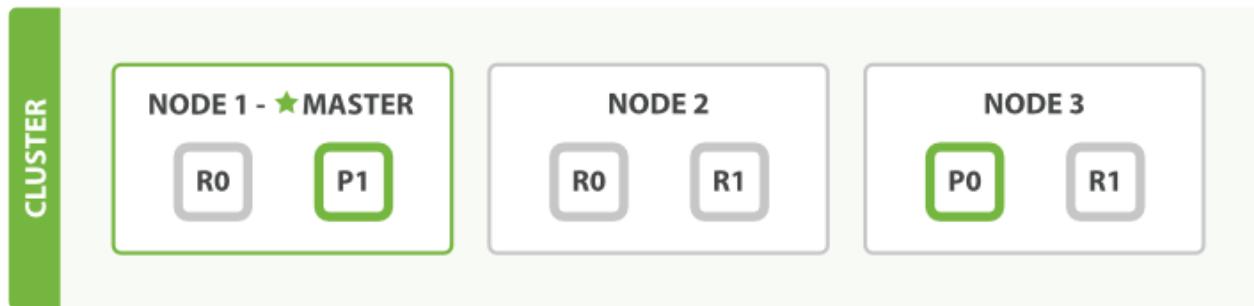


Figure 8. 有三个点和一个索引的集群

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

TIP 当发送请求的时候，为了展示，更好的做法是集群中所有的点。

新建、索引和删除文档

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

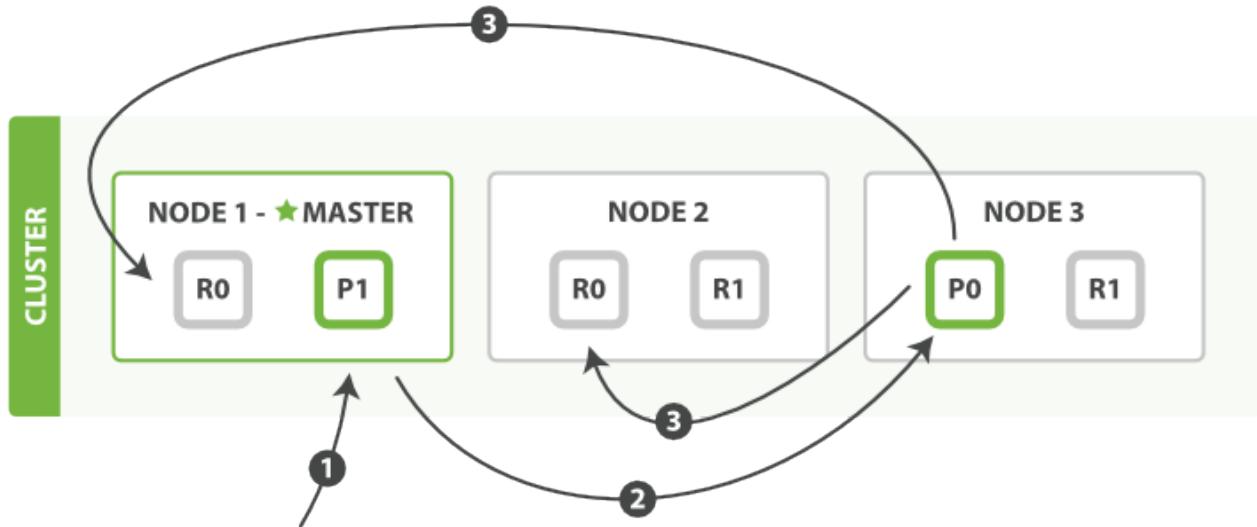


Figure 9. 新建、索引和 除 文

以下是在主副分片和任何副本分片上面 成功新建，索引和 除文 所需要的 序：

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

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

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

`consistency`

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

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

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

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

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

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

timeout

如果没有足 的副本分片会 生什 ？ Elasticsearch会等待，希望更多的分片出 。 情况下，它最多等待1分 。 如果 需要， 可以使用 `timeout` 参数 使它更早 止： `100 100` 秒， `30s` 是30秒。

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

取回一个文

可以从主分片或者从其它任意副本分片 索文 ， 如下 所示 取回 个文 。

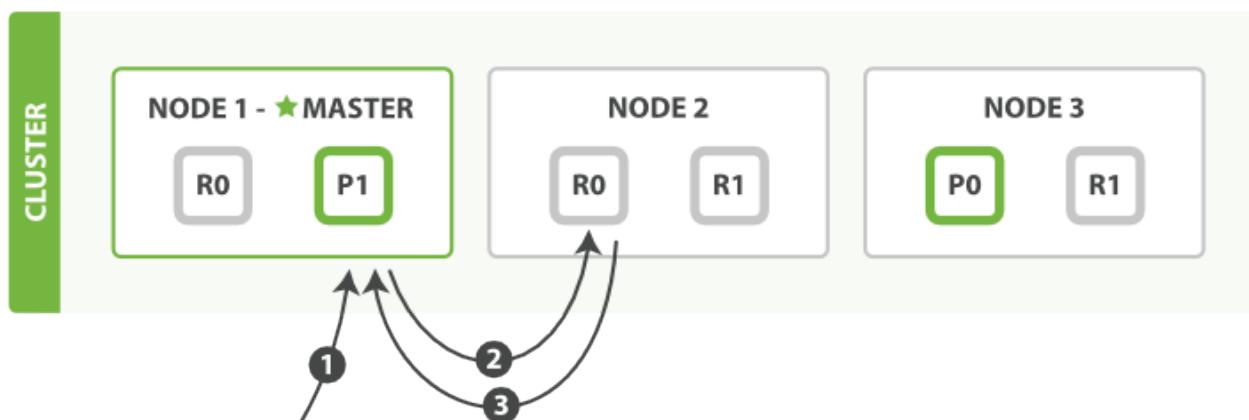


Figure 10. 取回 个文

以下是从主分片或者副本分片 索文 的 序：

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

在 理 取 求 ， 点在 次 求的 候都会通 所有的副本分片来 到 均衡。

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

局部更新文

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

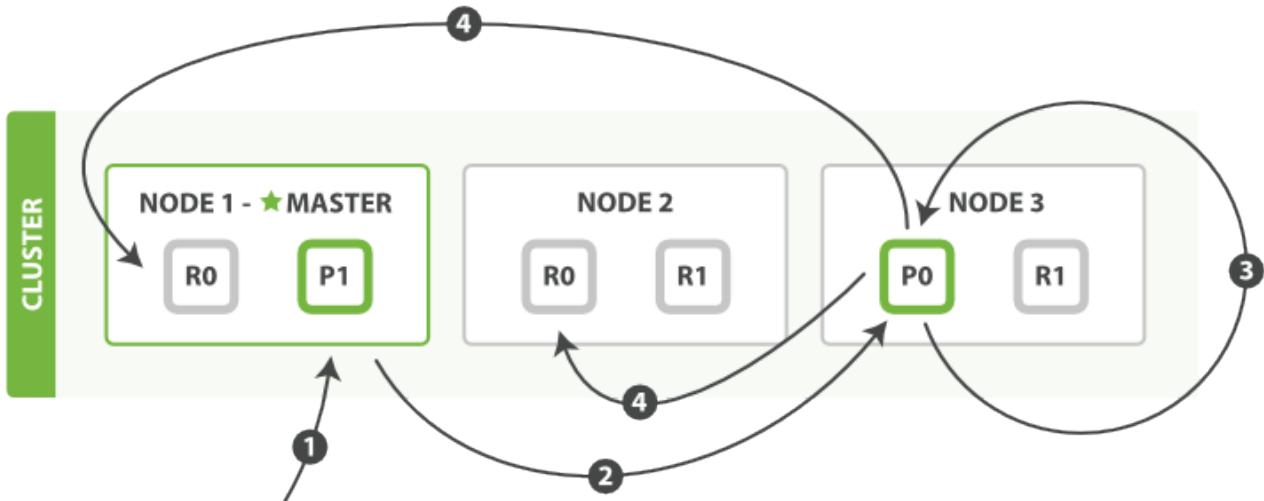


Figure 11. 局部更新文

以下是部分更新一个文 的：

1. 客 端向 `Node 1` 送更新 求。
2. 它将 求 到主分片所在的 `Node 3`。
3. `Node 3` 从主分片 索文 , 修改 `_source` 字段中的 JSON , 并且 重新索引主分片的文 。 如果文 已 被 一个 程修改, 它会重 3 , 超 `retry_on_conflict` 次后放 。
4. 如果 `Node 3` 成功地更新文 , 它将新版本的文 并行 到 `Node 1` 和 `Node 2` 上的副本分片, 重新建立索引。 一旦所有副本分片都返回成功, `Node 3` 向 点也返回成功, 点向客 端返回成功。

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

基于文 的 制

当主分片把更改 到副本分片 , 它不会 更新 求。 相反, 它 完整文 的新版本。 住, 些更改将会 到副本分片, 并且不能保 它 以 送它 相同的 序到 。 如果Elasticsearch 更改 求, 可能以 的 序 用更改, 致得到 坏的文 。

多文 模式

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

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

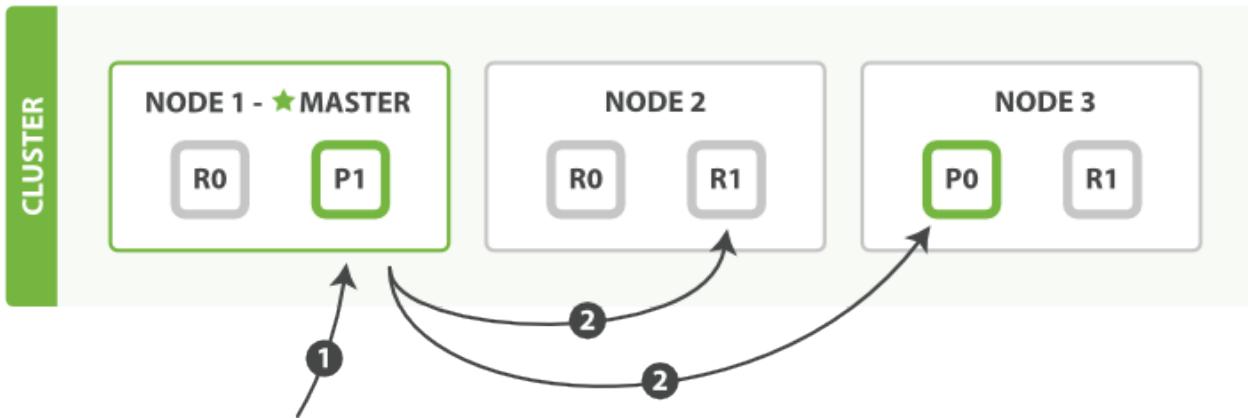


Figure 12. 使用 `mget` 取回多个文

以下是使用一个 `mget` 求取回多个文 所需的 序：

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

可以 `docs` 数 中 个文 置 `routing` 参数。

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

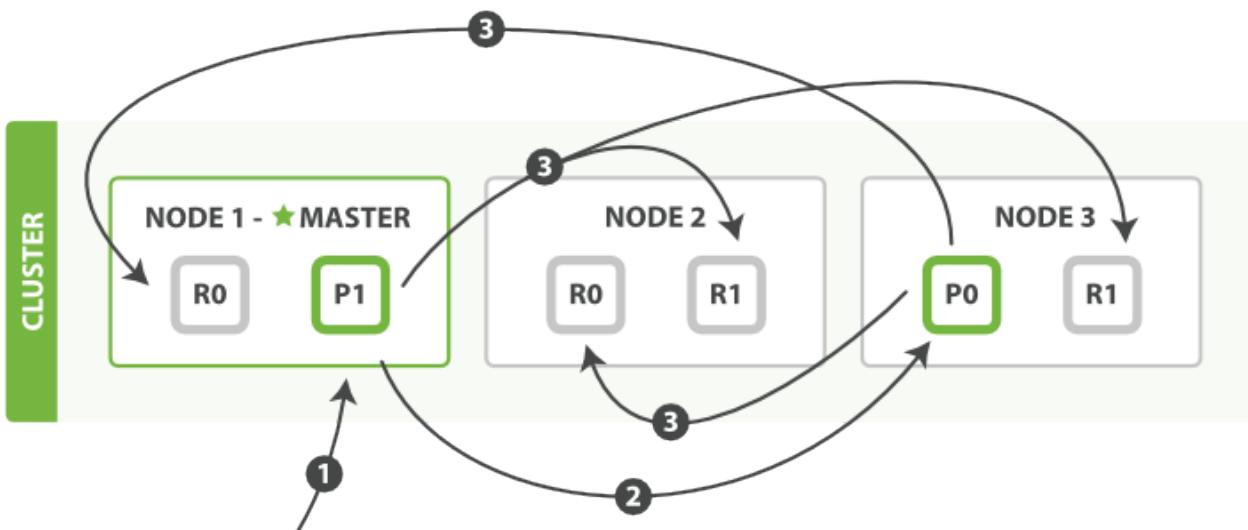


Figure 13. 使用 `bulk` 修改多个文

`bulk` API 按如下 序 行：

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

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

什么是有趣的格式？

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

了回答一点，我需要解一点背景：在批量请求中引用的个文档可能属于不同的主分片，一个文档可能被分配集群中的任何点。这意味着批量请求 `bulk` 中的每个操作都需要被送到正点上的正确分片。

如果一个请求被包装在 JSON 数字中，那就意味着我需要执行以下操作：

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

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

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

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

搜索——最基本的工具

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

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

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

搜索（search）可以做到：

- 在类似于 `gender` 或者 `age` 的字段上使用规范化，`join_date` 的字段上使用排序，就像 SQL 的规范化一样。
- 全文索引，找出所有匹配关键字的文档并按照相关性（relevance）排序后返回结果。
- 以上二者兼而有之。

很多搜索都是 箱即用的， 了充分 挖 Elasticsearch 的潜力， 需要理解以下三个概念：

映射 (*Mapping*)

描述数据在 个字段内如何存

分析 (*Analysis*)

全文是如何 理使之可以被搜索的

域特定 言 (*Query DSL*)

Elasticsearch 中 大 活的 言

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

我 将使用最 的形式 始介 `search` API。

数据

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

可以把 些命令 制到 端中 行来 践本章的例子。

外，如果 的是在 版本，可以 点 个 接 感受下。

空搜索

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

```
GET /_search
```

返回的 果（ 了界面 的）像 ：

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

hits

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

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

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

`max_score` 是与 所匹配文 的 `_score` 的最大 。

took

`took` 告 我 行整个搜索 求耗 了多少 秒。

shards

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

timeout

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

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

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

WARNING

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

使用超 是因 SLA(服 等) 是很重要的，而不是因 想去中止行的 。

多索引，多 型

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

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

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

`/_search`

在所有的索引中搜索所有的 型

`/gb/_search`

在 `gb` 索引中搜索所有的 型

`/gb,us/_search`

在 `gb` 和 `us` 索引中搜索所有的文

`/g*,u*/_search`

在任何以 `g` 或者 `u` 的索引中搜索所有的 型

`/gb/user/_search`

在 `gb` 索引中搜索 `user` 型

/gb,us/user,tweet/_search

在 gb 和 us 索引中搜索 user 和 tweet 型

/_all/user,tweet/_search

在所有的索引中搜索 user 和 tweet 型

当在一的索引下 行搜索的 候, Elasticsearch

求到索引的

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

TIP 搜索一个索引有五个主分片和搜索五个索引各有一个分片准 来所 是等 的。

接下来, 将明白 的方式如何 活的根据需求的 化 容 得 。

分

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

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

size

示 返回的 果数量, 是 10

from

示 跳 的初始 果数量, 是 0

如果 展示 5 条 果, 可以用下面方式 求得到 1 到 3 的 果:

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

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

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

。

在分布式系 中深度分

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

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

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

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

量 搜索

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

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

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

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

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

但是 字符串参数所需要的 百分比 (者注：URL) 上更加 :

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

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

_all 字段

个 搜索返回包含 **mary** 的所有文 :

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

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

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

Elasticsearch 是如何在三个不同的字段中 到 果的 ？

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

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

就好似 加了一个名叫 `_all` 的 外字段：

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

除非 置特定字段，否 字符串就使用 `_all` 字段 行搜索。

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

更 的

下面的 tweents 型，并使用以下的条件：

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

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

字符串在做了 当的 后，可 性很差：

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

从之前的例子中可以看出， 量 的 字符串搜索效果 是挺 人 喜的。 它的 法在相 参考文 中有 解 ，以便 的表 很 的 。 于通 命令做一次性 ，或者是在 段 ，都非常方便。

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

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

TIP

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

相反，我 常在生 境中更多地使用功能全面的 *request body* API，除了能完成以上所有功能，有一些附加功能。但在到 那个 段之前，首先需要了解数据在 Elasticsearch 中是如何被索引的。

映射和分析

当 弄索引里面的数据 ，我 一些奇怪的事情。一些事情看起来被打乱了：在我 的索引中有12条推文，其中只有一条包含日期 **2014-09-15**，但是看一看下面 命中的 数 (total)：

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

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

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

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

将得到如下 果：

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

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

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

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

精 VS 全文

Elasticsearch 中的数据可以概括的分 : 精 和全文。

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

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

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

NOTE

May is fun but June bores me.

它指的是月是人？

精很容易。果是二制的：要匹配，要不匹配。很容易用SQL表示：

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

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

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

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

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

倒排索引

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

例如，假我有个文，个文的 `content` 域包含如下内容：

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

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

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

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

Term	Doc_1	Doc_2
brown	X	X
quick	X	
Total	2	1

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

但是，我 目前的倒排索引有一些：

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

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

我 的用 可以合理的期望 个文 与 匹配。我 可以做的更好。

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

- **Quick** 可以小写化 **quick**。

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

在索引看上去像 :

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

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

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

分 和 准化的 程称 分析 , 我 会在下个章 。

分析与分析器

分析包含下面的 程 :

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

分析器 行上面的工作。 分析器 上是将三个功能封装到了一个包里 :

字符 器

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

分 器

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

Token 器

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

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

内置分析器

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

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

准分析器

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

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

分析器

分析器在任何不是字母的地方分隔文本，将 条小写。它会 生

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

空格分析器

空格分析器在空格的地方 分文本。它会 生

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

言分析器

特定 言分析器可用于 [{ref}/analysis-lang-analyzer.html\[很多 言\]](#)。它 可以考 指定 言的特点。例如， **英** 分析器附 了一 英 无用 （常用 ， 例**and** 或者 **the**， 它 相 性没有多少影 ），它 会被 除。由于理解英 法的 ， 个分 器可以提取英 的 干 。

英 分 器会 生下面的 条：

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

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

什 时候使用分析器

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

全文域，理解一个域是如何定义的，因此它可以做正确的事：

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

在可以理解在始章的什么返回那的结果：

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

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

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

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

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

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

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

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

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

分析器

有时候很难理解分析的过程和被存储到索引中的条，特别是接触Elasticsearch。为了理解发生了什么，可以使用 `analyze` API 来看文本是如何被分析的。在消息体里，指定分析器和要分析的文本：

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

结果中一个元素代表一个唯一的条：

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

`token` 是 存 到索引中的 条。 `position` 指明 条在原始文本中出 的位置。 `start_offset` 和 `end_offset` 指明字符在原始字符串中的位置。

TIP 一个分析器的 `type` 都不一 , 可以忽略它 。它 在Elasticsearch 中的唯一作用在于[{ref}/analysis-keep-types-tokenfilter.html](#)[`keep_types` token 器]。

`analyze` API 是一个有用的工具, 它有助于我 理解Elasticsearch索引内部 生了什 , 随着深入, 我会 一 它。

指定分析器

当Elasticsearch在 的文 中 到一个新的字符串域, 它会自 置其 一个全文 字符串 域, 使用 准 分析器 它 行分析。

不希望 是 。可能 想使用一个不同的分析器, 用于 的数据使用的 言。有 时候 想要一个字符串域就是一个字符串域—不使用分析, 直接索引 入的精 , 例如用 ID或者一个内部的状 域或 。

要做到 一点, 我 必 手 指定 些域的映射。

映射

了能 将 域 , 数字域 数字, 字符串域 全文或精 字符串, Elasticsearch

需要知道 个域中数据的 型。 个信息包含在映射中。

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

我 会在 [型和映射](#) 映射。本 ，我 只 足 入 的内容。

核心 域 型

Elasticsearch 支持如下 域 型：

字符串: `string`

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

浮点数: `float`, `double`

布 型: `boolean`

日期: `date`

当 索引一个包含新域的文 一之前未曾出 -- Elasticsearch 会使用 映射， 通 JSON中基本数据 型， 域 型， 使用如下：

JSON type

域 type

布 型: `true` 或者 `false`

`boolean`

整数: `123`

`long`

浮点数: `123.45`

`double`

字符串, 有效日期: `2014-09-15`

`date`

字符串: `foo bar`

`string`

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

看映射

通 `/_mapping`，我 可以 看 Elasticsearch 在一个或多个索引中的一个或多个 型的映射。在 [始章](#)，我 已 取得索引 `gb` 中 型 `tweet` 的映射：

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

Elasticsearch 根据我 索引的文 , 域(称 属性) 生成的映射。

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

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

一下! 而不是假 的映射是正 的。

自定 域映射

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

- 全文字符串域和精 字符串域的区
- 使用特定 语言分析器
- 化域以 部分匹配
- 指定自定 数据格式
- 有更多

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

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

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

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

index

index 属性控制 索引字符串。它可以是下面三个：

analyzed

首先分析字符串，然后索引它。句 ， 以全文索引 个域。

not_analyzed

索引 个域，所以它能 被搜索，但索引的是精 。不会 它 行分析。

no

不索引 个域。 个域不会被搜索到。

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

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

NOTE

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

analyzer

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

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

在 [自定 分析器](#)，我 会展示 定 和使用自定 分析器。

更新映射

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

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

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

了描述指定映射的 方式，我 先 除 `gd` 索引：

```
DELETE /gb
```

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

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

① 通 消息体中指定的 `mappings` 建了索引。

后，我 决定在 `tweet` 映射 加一个新的名 `tag` 的 `not_analyzed` 的文本域，使用 `_mapping` :

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

注意，我 不需要再次列出所有已存在的域，因 无 如何我 都无法改 它 。新域已 被合并到存在的映射中。

映射

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

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

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

① 消息体里面 我 想要分析的文本。

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

核心域 型

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

多 域

很有可能, 我 希望 tag 域包含多个 。我 可以以数 的形式索引 :

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

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

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

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

NOTE

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

空域

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

下面三 域被 是空的, 它 将不会被索引 :

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

多 象

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

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

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

内部 象的映射

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

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

① 根 象

② 内部 象

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

内部 象是如何索引的

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

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

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

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

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

内部 象数

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

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

个文 会像我 之前描述的那 被扁平化 理， 果如下所示：

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

`{age: 35}` 和 `{name: Mary White}` 之 的相 性已 失了，因 个多 域只是一包无序的 ，而不是有序数 。 足以 我 ，“有一个26 的追随者？”

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

相 内部 象被称 `nested` 象，可以回答上面的 ，我 后会在[nested-objects]中介 它。

求体

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

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

空

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

```
GET /_search
```

```
{}
```

① 是一个空的 求体。

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

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

```
{}
```

同 可以使用 `from` 和 `size` 参数来分：

```
GET /_search
```

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

一个 求体的 GET 求？

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

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

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

```
POST /_search
```

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

似的 可以 用于任何需要 求体的 `GET API`。

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

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

domain-specific language) 或者 Query DSL 来写句。

表 式

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

要使用表式，只需将句 `query` 参数：

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

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

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

句的

一个句的典型：

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

如果是某个字段，那它的如下：

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

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

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

完整的 求如下：

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

合并 句

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

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

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

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

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

```
{
  "bool": {
    "must": { "match": { "email": "business opportunity" }},
    "should": [
      { "match": { "starred": true }},
      { "bool": {
        "must": { "match": { "folder": "inbox" }},
        "must_not": { "match": { "spam": true } }
      }}
    ],
    "minimum_should_match": 1
  }
}
```

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

与

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

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

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

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

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

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

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

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

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

性能差

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

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

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

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

如何 与

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

最重要的

然 Elasticsearch 自 了很多的 , 但 常用到的也就那 几个。我 将在 [search-in-depth] 章 那些 的 , 接下来我 最重要的几个 行 介 。

match_all

`match_all` 的匹配所有文 。在没有指定 方式 ，它是 的：

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

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

match

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

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

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

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

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

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

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

multi_match

`multi_match` 可以在多个字段上 行相同的 `match` ：

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

range

range 指出那些落在指定区间的数字或者：

```
{  
  "range": {  
    "age": {  
      "gte": 20,  
      "lt": 30  
    }  
  }  
}
```

被允许的操作符如下：

gt

大于

gte

大于等于

lt

小于

lte

小于等于

term

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

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

term 于输入的文本不分析，所以它将决定的精确。

terms

terms 和 **term** 一样，但它允许多个字段进行匹配。如果一个字段包含了指定中的任何一个，那么这个文本满足条件：

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

和 **term** 一样，**terms** 于输入的文本不分析。它匹配那些精确匹配的（包括在大小写、重音、空格等方面的不同）。

exists 和 missing

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

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

些 常用于某个字段有 的情况和某个字段 的情况。

合多

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

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

must

文 必 匹配 些条件才能被包含 来。

must_not

文 必 不 匹配 些条件才能被包含 来。

should

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

filter

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

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

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

```
{
  "bool": {
    "must": { "match": { "title": "how to make millions" }},
    "must_not": { "match": { "tag": "spam" }},
    "should": [
      { "match": { "tag": "starred" }},
      { "range": { "date": { "gte": "2014-01-01" }}}
    ]
  }
}
```

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

加 器 (`filtering`) 的

如果我 不想因 文 的 而影 得分，可以用 `filter` 句来重写前面的例子：

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

① range 已 从 `should` 句中移到 `filter` 句

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

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

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

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

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

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

`constant_score`

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

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

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

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

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

validate-query API 可以用来 是否合法。

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

以上 **validate** 求的 答告 我 个 是不合法的：

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

理解 信息

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

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

① **explain** 参数可以提供更多 于 不合法的信息。

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

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

理解句

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

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

我 的一个 index 都会返回 的 `explanation`，因一个 index 都有自己的映射和分析器：

```
{
  "valid" : true,
  "_shards" : { ... },
  "explanations" : [ {
    "index" : "us",
    "valid" : true,
    "explanation" : "tweet:really tweet:powerful"
  }, {
    "index" : "gb",
    "valid" : true,
    "explanation" : "tweet:realli tweet:power"
  } ]
}
```

从 `explanation` 中可以看出，匹配 `really powerful` 的 `match` 被重写 个 `tweet` 字段的 single-term，一个single-term 字符串分出来的一个term。

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

排序与相 性

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

排序

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

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

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

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

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

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

NOTE

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

按照字段的 排序

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

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

会注意到 果中的 个不同点：

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

① `_score` 不被 算，因 它并没有用于排序。

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

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

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

一个 便方法是， 可以指定一个字段用来排序：

TIP

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

字段将会 升序排序， 而按照 `_score` 的 行降序排序。

多 排序

假定我 想要 合使用 `date` 和 `_score` 行，并且匹配的 果首先按照日期排序， 然后按照相 性排序：

```
GET /_search
{
  "query": {
    "bool": {
      "must": { "match": { "tweet": "manage text search" }},
      "filter": { "term": { "user_id": 2 }}
    }
  },
  "sort": [
    { "date": { "order": "desc" }},
    { "_score": { "order": "desc" }}
  ]
}
```

排序条件的 序是很重要的。 果首先按第一个条件排序， 当 果集的第一个 `sort` 完全相同 才会按照第二个条件 行排序， 以此 推。

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

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

NOTE

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

字段多 的排序

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

于数字或日期， 可以将多 字段 ， 可以通 使用 `min` 、 `max` 、 `avg` 或是 `sum` 排序模式 。 例如 可以按照 个 `date` 字段中的最早日期 行排序， 通 以下方法：

```
"sort": {  
    "dates": {  
        "order": "asc",  
        "mode": "min"  
    }  
}
```

字符串排序与多字段

被解析的字符串字段也是多字段，但是很少会按照想要的方式行排序。如果想分析一个字符串，如 fine old art，包含 3 个。我很可能想要按第一的字母排序，然后按第二的字母排序，如此，但是 Elasticsearch 在排序程序中没有的信息。

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

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

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

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

参数允化一个的映射如：

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

一个多字段映射如：

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

① tweet 主字段与之前的一：是一个 analyzed 全文字段。

② 新的 tweet.raw 子字段是 not_analyzed.

在，至少只要我 重新索引了我 的数据，使用 `tweet` 字段用于搜索，`tweet.raw` 字段用于排序：

```
GET /_search
{
  "query": {
    "match": {
      "tweet": "elasticsearch"
    }
  },
  "sort": "tweet.raw"
}
```

WARNING 以全文 `analyzed` 字段排序会消耗大量的内存。 取更多信息 看 [\[aggregations-and-analysis\]](#)。

什 是相 性？

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

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

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

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

索 率

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

反向文 率

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

字段 度准

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

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

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

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

TIP 我 有一 整章着眼于相 性 算和如何 其配合 的需求 [\[controlling-relevance\]](#)。

理解 分 准

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

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

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

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

首先, 我 看一下普通 返回的元数据 :

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

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

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

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

```

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

```

① `honeymoon` 相性分算的

② 索率

③ 反向文率

④ 字段度准

WARNING

出 `explain` 果代是十分昂的，它只能用作工具。千万不要用于生境。

第一部分是于算的。告了我 `honeymoon` 在 `tweet` 字段中的索率/反向文率或TF/IDF，（里的文 ① 是一个内部的 ID，跟我没有系，可以忽略。）

然后它提供了重是如何算的：

索率：

索 'honeymoon' 在 个文 的 'tweet' 字段中的出 次数。

反向文 率：

索 'honeymoon' 在索引上所有文 的 'tweet' 字段中出 的次数。

字段 度准：

在 个文 中， 'tweet' 字段内容的 度 -- 内容越 ， 越小。

的 句解 也非常 ， 但是包含的内容与上面例子大致相同。 通 段信息我 可以了解搜索结果是如何 生的。

TIP JSON 形式的 `explain` 描述是 以 的， 但是 成 YAML 会好很多，只需要在参数中加上 `format=yaml`。

理解文 是如何被匹配到的

当 `explain` 加到某一文 上 ， `explain` api 会 助 理解 何 个文 会被匹配，更重要的是，一个文 何没有被匹配。

求路径 `/index/type/id/_explain`，如下所示：

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

不只是我 之前看到的充分解 ， 我 在有了一个 `description` 元素，它将告 我 ：

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

也就是 我 的 `user_id` 子句使 文 不能匹配到。

Doc Values 介

本章的最后一个 是 于 Elasticsearch 内部的一些 行情况。在 里我 先不介 新的知识，所以我 意 到，Doc Values 是我 需要反 提到的一个重要 。

当一个字段行排序，Elasticsearch 需要一个匹配到的文得到相的。倒排索引的索性能是非常快的，但是在字段排序却不是理想的。

- 在搜索的候，我 能通 搜索 快速得到 果集。
- 当排序的候，我 需要倒排索引里面某个字段的集合。句，我 需要 **倒置** 倒排索引。

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

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

Elasticsearch 中的 doc vaules 常被用到以下景：

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

因 文被序列化到磁，我可以依操作系的帮助来快速。当 **working set** 小于点的可用内存，系会自将所有的文保存在内存中，使得其写十分高速；当其大于可用内存，操作系会自把 doc values 加到系的存中，从而避免了 jvm 堆内存溢出常。

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

行分布式 索

在 之前，我 将道一下在分布式境中搜索是行的。比我 在 [分布式文存](#) 章 的基本的 - 改- (CRUD) 求要一些。

内容提示

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

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

一个 CRUD 操作只 个文 行理，文 的唯一性由 **_index**, **_type**, 和 **routing values** (通常 是文的 **_id**) 的合来定。表示我一切的知道集群中 个分片含有此文。

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

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

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

段

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

先 列

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

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

个 段的 程如 程分布式搜索 所示。

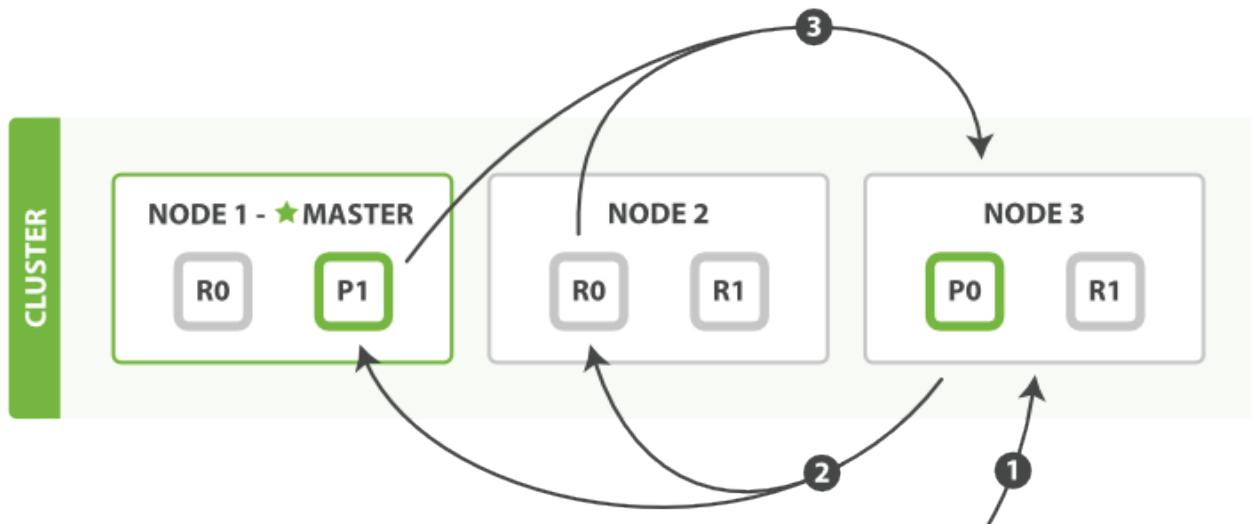


Figure 14. 程分布式搜索

段包含以下三个 ：

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

当一个搜索 求被 送到某个 点 ， 个 点就 成了 点。

个 点的任 是广播

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

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

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

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

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

取回 段

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

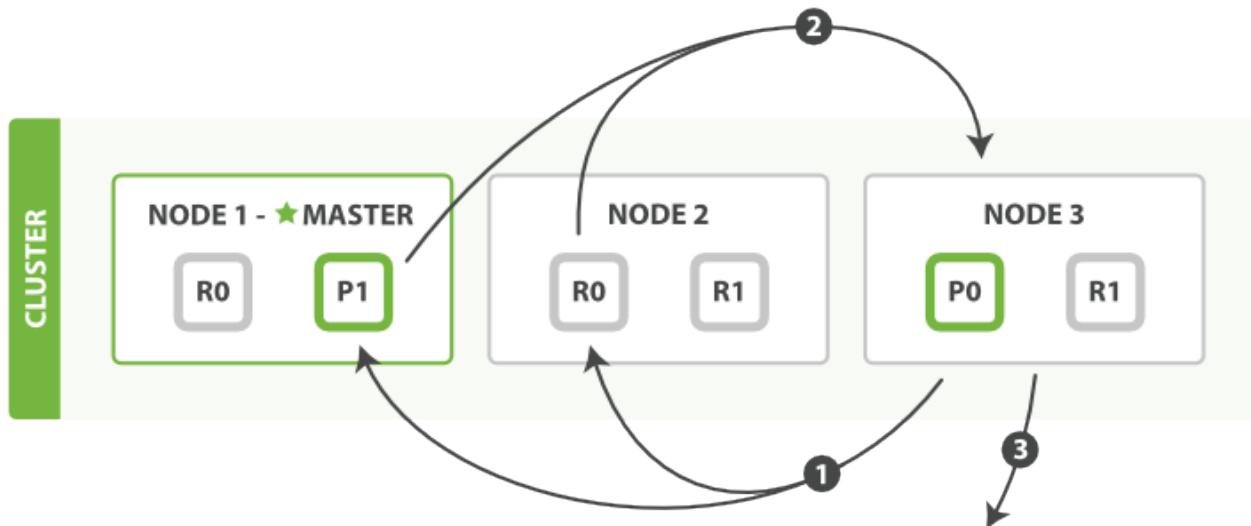


Figure 15. 分布式搜索的取回 段

分布式 段由以下 成：

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

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

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

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

深分 (Deep Pagination)

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

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

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

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

搜索

有几个 参数可以影 搜索 程。

偏好

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

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

Bouncing Results

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

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

超

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

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

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

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

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

① 个搜索 求超 了。

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

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

WARNING

路由

在 [路由一个文 到一个分片中](#) 中，我 解 如何定制参数 `routing`，它能 在索引 提供来 保相 的文，比如属于某个用 的文 被存 在某个分片上。 在搜索的 时候，不用搜索索引的所有分片，而是通 指定几个 `routing` 来限定只搜索几个相 的分片：

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

个技 在 大 模搜索系 就会派上用 ，我 在 [\[scale\]](#) 中 它。

搜索 型

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

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

搜索 型 `dfs_query_then_fetch` 有 段， 个 段可以从所有相 分片 取 来 算全局 。 我 在 [\[relevance-is-broken\]](#) 会再 它。

游 'Scroll'

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

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

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

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

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

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

① 保持游 口一分 。

② 字 _doc 是最有效的排序 序。

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

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

① 注意再次 置游 期 一分 。

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

NOTE 注意每次返回一个新字段 `_scroll_id`。每次我做下一次游，我必须把前一次返回的字段 `_scroll_id` 去。当没有更多的果返回的时候，我就理完所有匹配的文了。

TIP 提示：某些官方的 Elasticsearch 客端比如 Python 客端和 Perl 客端提供了个功能易用的封装。

索引管理

我已看到 Elasticsearch 一个新的用得，不需要任何先或置。不久，要不了多久就会始想要化索引和搜索程，以便更好地合的特定用例。些定制几乎着索引和型的方方面面，在本章，我将介管理索引和型映射的 API 以及一些最重要的置。

建一个索引

到目前止，我已通索引一篇文建了一个新的索引。个索引采用的是配置，新的字段通映射的方式被添加到型映射。在我需要个建立索引的程做更多的控制：我想要保个索引有数量中的主分片，并且在我索引任何数据之前，分析器和映射已被建立好。

了到个目的，我需要手建索引，在求体里面入置或型映射，如下所示：

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

如果想禁止自建索引，可以通在 config/elasticsearch.yml 的一个点下添加下面的配置：

```
action.auto_create_index: false
```

NOTE 我会在之后用 [index-templates] 来配置自建索引。在索引日志数据的时候尤其有用：将日志数据索引在一个以日期尾命名的索引上，子夜分，一个配置的新索引将会自行建。

除一个索引

用以下的求来除索引：

```
DELETE /my_index
```

也可以 除多个索引：

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

甚至可以 除全部索引：

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

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

NOTE `action.destructive_requires_name: true`

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

索引置

可以通修改配置来自定索引行，配置参照 [{ref}/index-modules.html\[索引模 \]](#)

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

下面是 个最重要的置：

`number_of_shards`

个索引的主分片数，是 5。个配置在索引建后不能修改。

`number_of_replicas`

个主分片的副本数，是 1。于活的索引，个配置可以随修改。

例如，我 可以 建只有一个主分片，没有副本的小索引：

```
PUT /my_temp_index  
{  
  "settings": {  
    "number_of_shards": 1,  
    "number_of_replicas": 0  
  }  
}
```

然后，我可以用 `update-index-settings` API 修改副本数：

```
PUT /my_temp_index/_settings
{
  "number_of_replicas": 1
}
```

配置分析器

第三个重要的索引 置是 `analysis` 部分，用来配置已存在的分析器或 建新的自定 分析器。

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

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

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

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

在下面的例子中，我 建了一个新的分析器，叫做 `es_std`， 并使用 定 的西班牙 停用 列表：

```
PUT /spanish_docs
{
  "settings": {
    "analysis": {
      "analyzer": {
        "es_std": {
          "type": "standard",
          "stopwords": "_spanish_"
        }
      }
    }
  }
}
```

`es_std` 分析器不是全局的—它 存在于我 定 的 `spanish_docs` 索引中。 了使用 `analyze` API 来 它 行 ，我 必 使用特定的索引名：

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

化的 果 示西班牙 停用 El 已被正 的移除：

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

自定 分析器

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

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

字符 器

字符 器 用来 <code>整理</code> 一个尚未被分 的字符串。例如，如果我 的文本是 HTML格式的，它会包含像 <code><p></code> 或者 <code><div></code> 的HTML ， 些 是我 不想索引的。我 可以使用 {ref}/analysis-htmlstrip-charfilter.html[<code>html清除</code> 字符 器] 来移除掉所有的HTML ， 并且像把 <code>Á</code> 相 的Unicode字符 <code>Á</code> ， HTML 体。

一个分析器可能有0个或者多个字符 器。

分 器

一个分析器 必 有一个唯一的分 器。 分 器把字符串分解成 个 条或者 元。 **准 分析器**里使用的 {ref}/analysis-standard-tokenizer.html[**准 分 器**] 把一个字符串根据 界分解成 个 条，并且移除掉大部分的 点符号，然而 有其他不同行 的分 器存在。

例如， {ref}/analysis-keyword-tokenizer.html[分 器] 完整地 出 接收到的同 的字符串，并不做任何分 。 {ref}/analysis-whitespace-tokenizer.html[**空格 分 器**] 只根据空格分割文本 。 {ref}/analysis-pattern-tokenizer.html[**正 分 器**] 根据匹配正 表 式来分割文本 。

元 器

分 ，作 果的 元流 会按照指定的 序通 指定的 元 器 。

元 器可以修改、添加或者移除 元。我 已 提到 **lowercase** 和 {ref}/analysis-stop-tokenfilter.html[**stop 器**]，但是在 Elasticsearch 里面 有很多可供 的 元 器。 {ref}/analysis-stemmer-tokenfilter.html[干 器] 把 制 干。 {ref}/analysis-asciifolding-tokenfilter.html[**ascii_folding 器**]移除 音符，把一个像 "très" 的

"`tres`"。[{ref}/analysis-ngram-tokenfilter.html\[ngram\]](#) 和 [{ref}/analysis-edgengram-tokenfilter.html\[edge_ngram\]](#) 元 器] 可以 生 合用于部分匹配或者自 全的 元。

在 [\[search-in-depth\]](#), 我 了在 里使用, 以及 使用分 器和 器。但是首先, 我 需要解一下 建自定 的分析器。

建一个自定 分析器

和我 之前配置 `es_std` 分析器一 , 我 可以在 `analysis` 下的相 位置 置字符 器、分 器和 元 器:

```
PUT /my_index
{
  "settings": {
    "analysis": {
      "char_filter": { ... custom character filters ... },
      "tokenizer": { ... custom tokenizers ... },
      "filter": { ... custom token filters ... },
      "analyzer": { ... custom analyzers ... }
    }
  }
}
```

作 示 , 我 一起来 建一个自定 分析器 , 个分析器可以做到下面的 些事:

1. 使用 `html`清除 字符 器移除HTML部分。
2. 使用一个自定 的 `映射`字符 器把 `&`替 "`和`" :

```
"char_filter": {
  "&_to_and": {
    "type": "mapping",
    "mappings": [ "&=> and "]
  }
}
```

3. 使用 `准分` 器分 。
4. 小写 条, 使用 `小写` 器 理。
5. 使用自定 `停止` 器移除自定 的停止 列表中包含的 :

```
"filter": {
  "my_stopwords": {
    "type": "stop",
    "stopwords": [ "the", "a" ]
  }
}
```

我 的分析器定 用我 之前已 置好的自定 器 合了已 定 好的分 器和 器：

```
"analyzer": {  
    "my_analyzer": {  
        "type": "custom",  
        "char_filter": [ "html_strip", "&_to_and" ],  
        "tokenizer": "standard",  
        "filter": [ "lowercase", "my_stopwords" ]  
    }  
}
```

起来，完整的 建索引 求看起来 像：

```
PUT /my_index  
{  
    "settings": {  
        "analysis": {  
            "char_filter": {  
                "&_to_and": {  
                    "type": "mapping",  
                    "mappings": [ "&=> and " ]  
                }},  
            "filter": {  
                "my_stopwords": {  
                    "type": "stop",  
                    "stopwords": [ "the", "a" ]  
                }},  
            "analyzer": {  
                "my_analyzer": {  
                    "type": "custom",  
                    "char_filter": [ "html_strip", "&_to_and" ],  
                    "tokenizer": "standard",  
                    "filter": [ "lowercase", "my_stopwords" ]  
                }  
            }  
        }  
    }}}
```

索引被 建以后，使用 `analyze` API 来 个新的分析器：

```
GET /my_index/_analyze?analyzer=my_analyzer  
The quick & brown fox
```

下面的 略 果展示出我 的分析器正在正 地 行：

```
{
  "tokens" : [
    { "token" : "quick", "position" : 2 },
    { "token" : "and", "position" : 3 },
    { "token" : "brown", "position" : 4 },
    { "token" : "fox", "position" : 5 }
  ]
}
```

个分析器 在是没有多大用 的，除非我 告诉 Elasticsearch 在 里用上它。我可以像下面 把一个分析器 用在一个 `string` 字段上：

```
PUT /my_index/_mapping/my_type
{
  "properties": {
    "title": {
      "type": "string",
      "analyzer": "my_analyzer"
    }
  }
}
```

型和映射

`型` 在 Elasticsearch 中表示一 相似的文 。 `型`由 `名称` ;比如 `user` 或 `blogpost` ;和 `映射` 成。

`映射`， 就像数据 中的 `schema` ， 描述了文 可能具有的字段或 `属性` 、
个字段的数据 型;比如 `string`, `integer` 或 `date`;以及Lucene是如何索引和存 些字段的。

型可以很好的抽象 分相似但不相同的数据。但由于 Lucene 的 理方式， 型的使用有些限制。

Lucene 如何 理文

在 Lucene 中，一个文 由一 的 成。 个字段都可以有多个 ， 但至少要有一个 。
似的，一个字符串可以通 分析 程 化 多个 。Lucene 不 心 些 是字符串、数字或日期—
所有的 都被当做 不透明字 。

当我 在 Lucene 中索引一个文 ， 个字段的 都被添加到相 字段的倒排索引中。 也可以将未
理的原始数据存 起来，以便 些原始数据在之后也可以被 索到。

型是如何 的

Elasticsearch 型是以 Lucene 理文 的 个方式 基 来 的。一个索引可以有多个 型， 些
型的文 可以存 在相同的索引中。

Lucene 没有文 型的概念， 个文 的 型名被存 在一个叫 `_type` 的元数据字段上。 当我 要 索某个 型的文 ， Elasticsearch 通 在 `_type` 字段上使用 器限制只返回 个 型的文 。

Lucene 也没有映射的概念。 映射是 Elasticsearch 将 JSON 文 映射 成 Lucene 需要的扁平化数据的方式。

例如，在 `user` 型中， `name` 字段的映射可以声明 个字段是 `string` 型，并且它的 被索引到名叫 `name` 的倒排索引之前，需要通 `whitespace` 分 器分析：

```
"name": {  
    "type": "string",  
    "analyzer": "whitespace"  
}
```

避免 型陷

致了一个有趣的思想：如果有 个不同的 型， 个 型都有同名的字段，但映射不同（例如：一个是字符串一个是数字），将会出 什 情况？

回答是，Elasticsearch 不会允 定 个映射。当 配置 个映射 ，将会出 常。

回答是， 个 Lucene 索引中的所有字段都包含一个 一的、扁平的模式。一个特定字段可以映射成 `string` 型也可以是 `number` 型，但是不能 者兼具。因 型是 Elasticsearch 添加的 于 Lucene 的 外机制（以元数据 `_type` 字段的形式），在 Elasticsearch 中的所有 型最 都共享相同的映射。

以 `data` 索引中 型的映射 例：

```
{
  "data": {
    "mappings": {
      "people": {
        "properties": {
          "name": {
            "type": "string",
          },
          "address": {
            "type": "string"
          }
        }
      },
      "transactions": {
        "properties": {
          "timestamp": {
            "type": "date",
            "format": "strict_date_optional_time"
          },
          "message": {
            "type": "string"
          }
        }
      }
    }
  }
}
```

个型定一个字段 (分是 "name"/"address" 和 "timestamp"/"message")。它看起来是相互独立的，但在后台 Lucene 将建一个映射，如：

```
{
  "data": {
    "mappings": {
      "_type": {
        "type": "string",
        "index": "not_analyzed"
      },
      "name": {
        "type": "string"
      }
    },
    "address": {
      "type": "string"
    }
  },
  "timestamp": {
    "type": "long"
  }
}
}
```

注：不是真有效的映射法，只是用于演示

于整个索引，映射在本上被扁平化成一个一的、全局的模式。就是什一个型不能定冲突的字段：当映射被扁平化，Lucene 不知道如何去理。

型

那，个的 是什？技上，多个型可以在相同的索引中存在，只要它的字段不冲突（要因字段是互独占模式，要因它共享相同的字段）。

重要的一点是：型可以很好的区分同一个集合中的不同分。在不同的分中数据的整体模式是相同的（或相似的）。

型不 合 完全不同型的数据 。如果 个型的字段集是互不相同的，就意味着索引中将有一半的数据是空的（字段将是稀疏的），最将致性能。在情况下，最好是使用 个 独的索引。

：

- 正：将 kitchen 和 lawn-care 型放在 products 索引中，因型基本上是相同的模式
- : 将 products 和 logs 型放在 data 索引中，因型互不相同。将它放在不同的索引中。

根 象

映射的最高一 被称 根 象，它可能包含下面几 ：

- 一个 `properties` 点，列出了文 中可能包含的 个字段的映射
- 各 元数据字段，它 都以一个下 ，例如 `_type`、`_id` 和 `_source`
- 置 ，控制如何 理新的字段，例如 `analyzer`、`dynamic_date_formats` 和 `dynamic_templates`
- 其他 置，可以同 用在根 象和其他 `object` 型的字段上，例如 `enabled`、`dynamic` 和 `include_in_all`

属性

我 已 在 核心 域 型 和 核心域 型 章 中介 文 字段和属性的三个最重要的 置：

`type`

字段的数据 型，例如 `string` 或 `date`

`index`

字段是否 当被当成全文来搜索（ `analyzed` ），或被当成一个准 的（ `not_analyzed` ），是完全不可被搜索（ `no` ）

`analyzer`

定在索引和搜索 全文字段使用的 `analyzer`

我 将在本 的后 部分 其他字段 型，例如 `ip`、`geo_point` 和 `geo_shape`。

元数据: `_source` 字段

地，Elasticsearch 在 `_source` 字段存 代表文 体的JSON字符串。和所有被存 的字段一，`_source` 字段在被写入磁 之前先会被 。

个字段的存 几乎 是我 想要的，因 它意味着下面的 些：

- 搜索 果包括了整个可用的文 ——不需要 外的从 一个的数据 来取文 。
- 如果没有 `_source` 字段，部分 `update` 求不会生效。
- 当 的映射改 ， 需要重新索引 的数据，有了`_source`字段 可以直接从Elasticsearch 做，而不必从 一个(通常是速度更慢的) 数据 取回 的所有文 。
- 当 不需要看到整个文 ， 个字段可以从 `_source` 字段提取和通 `get` 或者 `search` 求返回。
- 句更加 ， 因 可以直接看到 个文 包括什 ，而不是从一列id 它 的内容。

然而，存 `_source` 字段的 要使用磁 空 。如果上面的原因 来 没有一个是重要的，可以用下面的映射禁用 `_source` 字段：

```
PUT /my_index
{
  "mappings": {
    "my_type": {
      "_source": {
        "enabled": false
      }
    }
  }
}
```

在一个搜索请求里，可以通过在请求体中指定 `_source` 参数，来达到只取特定的字段的效果：

```
GET /_search
{
  "query": { "match_all": {}},
  "_source": [ "title", "created" ]
}
```

这些字段的会从 `_source` 字段被提取和返回，而不是返回整个 `_source`。

Stored Fields 被存字段

之后的索，除了索引一个字段的，可以存原始字段。有Lucene使用背景的用被存字段来他想要在搜索结果里面返回的字段。事实上，`_source`字段就是一个被存的字段。

在Elasticsearch中，文的个字段置存的做法通常不是最好的。整个文已被存 `_source` 字段。使用 `_source` 参数提取需要的字段是更好的。

元数据: `_all` 字段

在量搜索中，我介了 `_all` 字段：一个把其它字段当作一个大字符串来索引的特殊字段。`query_string` 子句(搜索 `?q=john`)在没有指定字段使用 `_all` 字段。

`_all` 字段在新用的探索段，当不清楚文的最是比有用的。可以使用一个字段来做任何，并且有很大可能到需要的文：

```
GET /_search
{
  "match": {
    "_all": "john smith marketing"
  }
}
```

随着用的展，搜索需求得更加明，会自己越来越少使用 `_all` 字段。`_all` 字段是搜索的急之策。通过指定字段，的更加活、大，也可以相性最高的搜索果行更粒度的控制。

NOTE `relevance algorithm` 考的一个最重要的原是字段的度：字段越短越重要。在短的 `title` 字段中出的短可能比在的 `content` 字段中出的短更加重要。字段度的区在 `_all` 字段中不会出。

如果不再需要 `_all` 字段，可以通下面的映射来禁用：

```
PUT /my_index/_mapping/my_type
{
  "my_type": {
    "_all": { "enabled": false }
  }
}
```

通 `include_in_all` 置来逐个控制字段是否要包含在 `_all` 字段中，是 `true`。在一个象(或根象)上置 `include_in_all` 可以修改一个象中的所有字段的行。

可能想要保留 `_all` 字段作一个只包含某些特定字段的全文字段，例如只包含 `title`, `overview`, `summary` 和 `tags`。相较于完全禁用 `_all` 字段，可以所有字段禁用 `include_in_all`，在的字段上用：

```
PUT /my_index/my_type/_mapping
{
  "my_type": {
    "include_in_all": false,
    "properties": {
      "title": {
        "type": "string",
        "include_in_all": true
      },
      ...
    }
  }
}
```

住，`_all` 字段是一个分的 `string` 字段。它使用分器来分析它的，不管个原本所在字段指定的分器。就像所有 `string` 字段，可以配置 `_all` 字段使用的分器：

```
PUT /my_index/my_type/_mapping
{
  "my_type": {
    "_all": { "analyzer": "whitespace" }
  }
}
```

元数据：文

文 与四个元数据字段相：

`_id`

文 的 ID 字符串

`_type`

文 的 型名

`_index`

文 所在的索引

`_uid`

`_type` 和 `_id` 接在一起 造成 `type#id`

情况下，`_uid` 字段是被存（可取回）和索引（可搜索）的。`_type` 字段被索引但是没有存，`_id` 和 `_index` 字段既没有被索引也没有被存，意味着它并不是真存在的。

尽管如此，然可以像真字段一样使用 `_id` 字段。Elasticsearch 使用 `_uid` 字段来派生出 `_id`。然可以修改些字段的 `index` 和 `store` 置，但是基本上不需要做。

映射

当 Elasticsearch 遇到文 中以前未遇到的字段，它用 `dynamic mapping` 来定字段的数据型并自把新的字段添加到型映射。

有 是想要的行 有 又不希望 。通常没有人知道以后会有什 新字段加到文 ，但是又希望些字段被自的索引。也 只想忽略它 。如果Elasticsearch是作重要的数据存，可能就会期望遇到新字段就会出常，能及。

幸的是可以用 `dynamic` 配置来控制行，可接受的如下：

`true`

添加新的字段—省

`false`

忽略新的字段

`strict`

如果遇到新字段出常

配置参数 `dynamic` 可以用在根 `object` 或任何 `object` 型的字段上。可以将 `dynamic` 的 `strict`，而只在指定的内部象中 它，例如：

```

PUT /my_index
{
  "mappings": {
    "my_type": {
      "dynamic": "strict", ①
      "properties": {
        "title": { "type": "string"}, 
        "stash": {
          "type": "object",
          "dynamic": true ②
        }
      }
    }
  }
}

```

① 如果遇到新字段，象 `my_type` 就会出常。

② 而内部 象 `stash` 遇到新字段就会建新字段。

使用上述 映射，可以 `stash` 象添加新的可 索的字段：

```

PUT /my_index/my_type/1
{
  "title": "This doc adds a new field",
  "stash": { "new_field": "Success!" }
}

```

但是 根 点 象 `my_type` 行同 的操作会失：

```

PUT /my_index/my_type/1
{
  "title": "This throws a StrictDynamicMappingException",
  "new_field": "Fail!"
}

```

NOTE

把 `dynamic` 置 `false` 一点儿也不会改 `_source` 的字段内容。`_source` 然包含被索引的整个JSON文。只是新的字段不会被加到映射中也不可搜索。

自定 映射

如果 想在 行 加新的字段，可能会 用 映射。然而，有 候， 映射 可能不太智能。幸 的是，我 可以通 置去自定 些 ，以便更好的 用于 的数据。

日期

当 Elasticsearch 遇到一个新的字符串字段，它会 个字段是否包含一个可 的日期，比如 2014-

01-01。如果它像日期，一个字段就会被作 `date` 型添加。否，它会被作 `string` 型添加。

有些时候一个行可能致一些。想象下，有如下 的一个文：

```
{ "note": "2014-01-01" }
```

假是第一次 `note` 字段，它会被添加 `date` 字段。但是如果下一个文像：

```
{ "note": "Logged out" }
```

然不是一个日期，但已。一个字段已是一个日期型，一个 `不合法的日期` 将会造成一个常。

日期可以通过在根象上置 `date_detection` `false` 来：

```
PUT /my_index
{
  "mappings": {
    "my_type": {
      "date_detection": false
    }
  }
}
```

使用一个映射，字符串将始作 `string` 型。如果需要一个 `date` 字段，必手添加。

NOTE Elasticsearch 判断字符串日期的可以通过 [{ref}/dynamic-field-mapping.html#date-detection](#)[`dynamic_date_formats` setting] 来置。

模板

使用 `dynamic_templates`，可以完全控制新生成字段的映射。甚至可以通字段名称或数据型来用不同的映射。

个模板都有一个名称，可以用来描述个模板的用途，一个 `mapping` 来指定映射使用，以及至少一个参数(如 `match`)来定个模板用于个字段。

模板按照序来；第一个匹配的模板会被用。例如，我 `string` 型字段定个模板：

- `es`：以 `_es` 尾的字段名需要使用 `spanish` 分器。
- `en`：所有其他字段使用 `english` 分器。

我将 `es` 模板放在第一位，因它比匹配所有字符串字段的 `en` 模板更特殊：

```

PUT /my_index
{
  "mappings": {
    "my_type": {
      "dynamic_templates": [
        { "es": {
          "match": "*_es", ①
          "match_mapping_type": "string",
          "mapping": {
            "type": "string",
            "analyzer": "spanish"
          }
        }},
        { "en": {
          "match": "*", ②
          "match_mapping_type": "string",
          "mapping": {
            "type": "string",
            "analyzer": "english"
          }
        }}
      ]
    }
  }
}

```

① 匹配字段名以 `_es` 尾的字段。

② 匹配其他所有字符串 型字段。

`match_mapping_type` 允用模板到特定 型的字段上，就像有 准 映射 的一， (例如 `string` 或 `long`)。

`match` 参数只匹配字段名称， `path_match` 参数匹配字段在 象上的完整路径，所以 `address.*.name` 将匹配 的字段：

```

{
  "address": {
    "city": {
      "name": "New York"
    }
  }
}

```

`unmatch` 和 `path_unmatch`将被用于未被匹配的字段。

更多的配置 {ref}/dynamic-mapping.html[映射文]。

省映射

通常，一个索引中的所有 型共享相同的字段和 置。 `default` 映射更加方便地指定通用 置，而不是

次 建新 型 都要重 置。 `default` 映射是新 型的模板。在 置 `default` 映射之后 建的所有 型都将 用 些 省的 置，除非 型在自己的映射中明 覆 些 置。

例如，我 可以使用 `default` 映射 所有的 型禁用 `_all` 字段，而只在 `blog` 型 用：

```
PUT /my_index
{
  "mappings": {
    "_default": {
      "_all": { "enabled": false }
    },
    "blog": {
      "_all": { "enabled": true }
    }
  }
}
```

`default` 映射也是一个指定索引 [dynamic templates](#) 的好方法。

重新索引 的数据

尽管可以 加新的 型到索引中，或者 加新的字段到 型中，但是不能添加新的分析器或者 有的字段 做改 。如果 那 做的 ， 果就是那些已 被索引的数据就不正 ， 搜索也不能正常工作。

有数据的 改 最 的 法就是重新索引：用新的 置 建新的索引并把文 从旧的索引 制到新的索引。

字段 `_source` 的一个 点是在Elasticsearch中已 有整个文 。 不必从源数据中重建索引，而且那 通常比 慢。

了有效的重新索引所有在旧的索引中的文 ，用 `scroll` 从旧的索引 索批量文 ， 然后用 `bulk API` 把文 推送到新的索引中。

从Elasticsearch v2.3.0 始， [{ref}/docs-reindex.html\[Reindex API\]](#) 被引入。它能 文 重建索引而不需要任何 件或外部工具。

批量重新索引

同 并行 行多个重建索引任 , 但是 然不希望 果有重 。正 的做法是按日期或者 的字段作 条件把大的重建索引分成小的任 :

```
GET /old_index/_search?scroll=1m
{
  "query": {
    "range": {
      "date": {
        "gte": "2014-01-01",
        "lt": "2014-02-01"
      }
    }
  },
  "sort": ["_doc"],
  "size": 1000
}
```

如果旧的索引持 会有 化, 希望新的索引中也包括那些新加的文 。那就可以 新加的文 做重 新索引, 但 是要用日期 字段 来匹配那些新加的文 。

索引 名和零停机

在前面提到的, 重建索引的 是必 更新 用中的索引名称。索引 名就是用来解决 个 的 !

索引 名 就像一个快捷方式或 接, 可以指向一个或多个索引, 也可以 任何一个需要索引名的 API来使用。 名 我 大的 活性, 允 我 做下面 些:

- 在 行的集群中可以无 的从一个索引切 到 一个索引
- 多个索引分 (例如, `last_three_months`)
- 索引的一个子集 建

在后面我 会 更多 于 名的使用。 在, 我 将解 使用 名在零停机下从旧索引切 到新索引 。

有 方式管理 名: `_alias` 用于 个操作, `_aliases` 用于 行多个原子 操作。

在本章中, 我 假 的 用有一个叫 `my_index` 的索引。事 上, `my_index` 是一个指向当前真 索引的 名。真 索引包含一个版本号: `my_index_v1`, `my_index_v2` 等等。

首先, 建索引 `my_index_v1`, 然后将 名 `my_index` 指向它 :

```
PUT /my_index_v1 ①
PUT /my_index_v1/_alias/my_index ②
```

① 建索引 `my_index_v1`。

② 置 名 `my_index` 指向 `my_index_v1`。

可以 一个 名指向 一个索引：

```
GET /*/_alias/my_index
```

或 些 名指向 个索引：

```
GET /my_index_v1/_alias/*
```

者都会返回下面的 果：

```
{
  "my_index_v1" : {
    "aliases" : {
      "my_index" : { }
    }
  }
}
```

然后，我 决定修改索引中一个字段的映射。当然，我 不能修改 存的映射，所以我 必 重新索引数据。首先，我 用新映射 建索引 `my_index_v2`：

```
PUT /my_index_v2
{
  "mappings": {
    "my_type": {
      "properties": {
        "tags": {
          "type": "string",
          "index": "not_analyzed"
        }
      }
    }
  }
}
```

然后我 将数据从 `my_index_v1` 索引到 `my_index_v2`，下面的 程在 [重新索引 的数据](#) 中已 描述。一旦我 定义 已 被正 地重索引了，我 就将 名指向新的索引。

一个 名可以指向多个索引，所以我 在添加 名到新索引的同 必 从旧的索引中 除它。 个操作需要原子化， 意味着我 需要使用 `_aliases` 操作：

```

POST /_aliases
{
  "actions": [
    { "remove": { "index": "my_index_v1", "alias": "my_index" }},
    { "add": { "index": "my_index_v2", "alias": "my_index" }}
  ]
}

```

的 用已 在零停机的情况下从旧索引 移到新索引了。

即使 在的索引 已 很完美了，在生 境中， 是有可能需要做一些修改的。

TIP 做好准：在 的 用中使用 名而不是索引名。然后 就可以在任何 候重建索引。 名的 很小， 广泛使用。

分片内部原理

在 集群内的原理， 我 介 了 分片， 并将它 描述成最小的 工作 元。但是究竟什 是一个分片，它是如何工作的？在 个章 ，我 回答以下 ：

- 什 搜索是近 的？
- 什 文 的 CRUD(建- 取-更新- 除)操作是 的？
- Elasticsearch 是 保 更新被持久化在断 也不 失数据？
- 什 除文 不会立刻 放空 ？
- `refresh`, `flush`, 和 `optimize` API 都做了什 ， 什 情况下 是用他 ？

最 的理解一个分片如何工作的方式是上一堂 史 。 我 将要 提供一个 近 搜索和分析的 分布式持久化数据存 需要解决的 。

内容警告

本章展示的 些信息 供 趣 。 了使用 Elasticsearch 并不需要理解和 所有的 。 个章 是 了了解工作机制，并且 了将来 需要 些信息 ，知道 些信息在 里。但是不要 被 些 所累。

使文本可被搜索

必 解决的第一个挑 是如何使文本可被搜索。 的数据 个字段存 个 ，但 全文 索并不 。文本字段中的 个 需要被搜索， 数据 意味着需要 个字段有索引多 (里指)的能力。

最好的支持 一个字段多个 需求的数据 是我 在 倒排索引 章 中介 的 倒排索引 。 倒排索引包含一个有序列表，列表包含所有文 出 的不重 个体，或称 ， 包含了它所有曾出 文 的列表。

Term	Doc 1	Doc 2	Doc 3	...
brown	X		X	...
fox	X	X	X	...
quick	X	X		...
the	X		X	...

NOTE 当倒排索引，我会到文引，因史原因，倒排索引被用来整个非化文本行引。Elasticsearch 中的文是有字段和的化 JSON 文。事上，在 JSON 文中，一个被索引的字段都有自己的倒排索引。

个倒排索引相比特定出的文列表，会包含更多其它信息。它会保存一个出的文数，在的文中的一个具体出的次数，在文中的序，个文的度，所有文的平均度，等等。些信息允 Elasticsearch 决定些比其它更重要，些文比其它文更重要，些内容在什 是相 性？中有描述。

了能预期功能，倒排索引需要知道集合中的所有文，是需要到的。

早期的全文索会整个文集合建立一个很大的倒排索引并将其写入到磁。一旦新的索引就，旧的就会被其替，最近的化便可以被索到。

不性

倒排索引被写入磁后是不可改的：它永不会修改。不性有重要的：

- 不需要。如果从来不更新索引，就不需要担心多程同修改数据的。
- 一旦索引被入内核的文件系存，便会留在里，由于其不性。只要文件系存中有足够的空，那大部分求会直接求内存，而不会命中磁。提供了很大的性能提升。
- 其它存(像filter存)，在索引的生命周期内始有效。它不需要在次数据改被重建，因数据不会化。
- 写入个大的倒排索引允数据被，少磁I/O和需要被存到内存的索引的使用量。

当然，一个不的索引也有不好的地方。主要事是它是不可的！不能修改它。如果需要一个新的文可被搜索，需要重建整个索引。要一个索引所能包含的数据量造成了很大的限制，要索引可被更新的率造成了很大的限制。

更新索引

下一个需要被解决的是在保留不性的前提下倒排索引的更新？答案是：用更多的索引。

通常加新的充索引来反映新近的修改，而不是直接重写整个倒排索引。一个倒排索引都会被流到—从最早的始—完后再果行合并。

Elasticsearch 基于 Lucene，个 java 引入了按段搜索的概念。一 *段*本身都是一个倒排索引，但 *索引* 在 Lucene 中除表示所有 *段* 的集合外，加了 *提交点* 的概念；一个列出了所有已知段的文件，就像在 [一个 Lucene 索引包含一个提交点和三个段](#) 中描的那。如 [一个 anchor="img-memory-segments"](#)

buffer">一个在内存 存中包含新文 的 Lucene 索引

所示，新的文 首先被添加到内存索引
存中，然后写入到一个基于磁 的段，如
commit">在一次提交后，一个新的段被添加到提交点而且 存被清空。

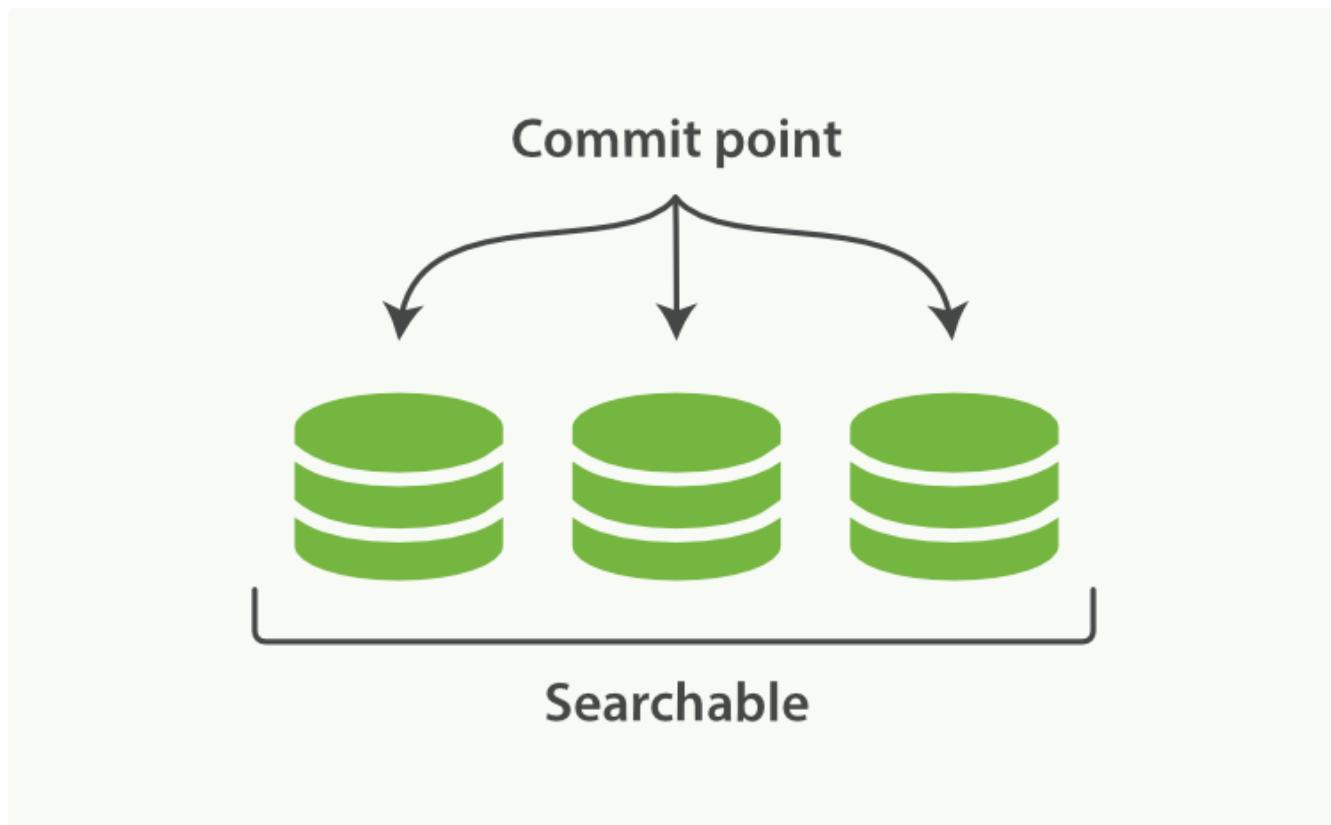


Figure 16. 一个 Lucene 索引包含一个提交点和三个段

索引与分片的比

被混 的概念是，一个 Lucene 索引 我 在 Elasticsearch 称作 分片 。一个 Elasticsearch 索引 是分片的集合。当 Elasticsearch 在索引中搜索的 候，他 送 到 一个属于索引的分片(Lucene 索引)，然后像 行分布式 索 提到的那 ，合并 个分片的 果到一个全局的 果集。

逐段搜索会以如下流程 行工作：

1. 新文 被收集到内存索引 存， 一个在内存 存中包含新文 的 Lucene 索引 。
2. 不 地， 存被 提交：
 - 一个新的段—一个追加的倒排索引—被写入磁 。
 - 一个新的包含新段名字的 提交点 被写入磁 。
 - 磁 行 同 — 所有在文件系 存中等待的写入都刷新到磁 ，以 保它 被写入物理文件。
3. 新的段被 ， 它包含的文 可 以被搜索。
4. 内存 存被清空， 等待接收新的文 。

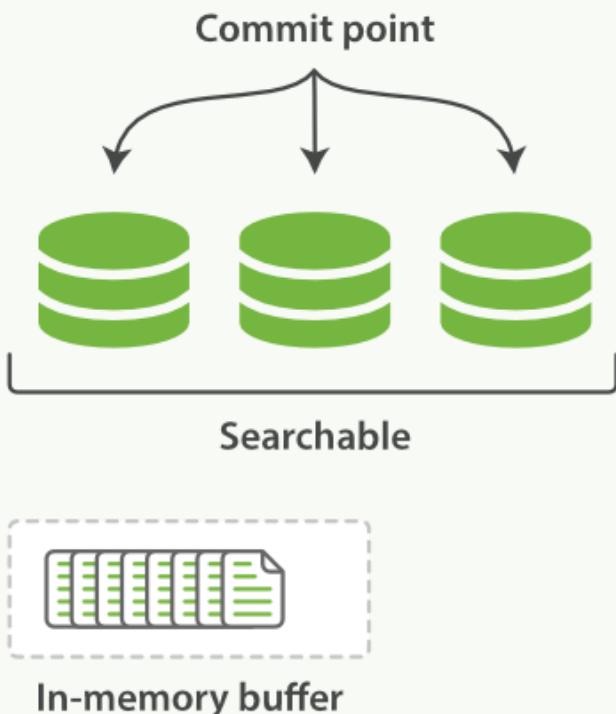


Figure 17. 一个在内存 存中包含新文 的 Lucene 索引

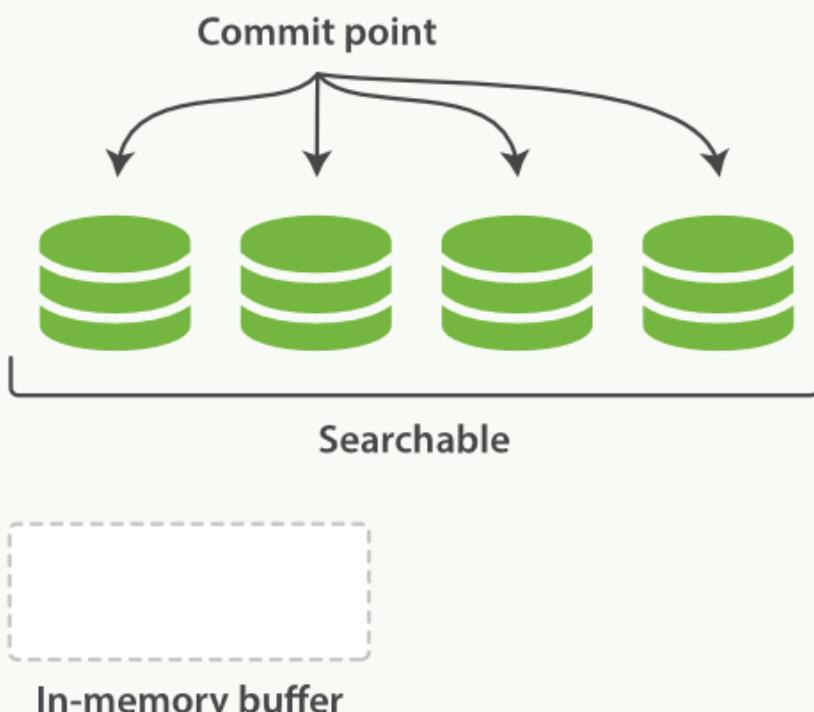


Figure 18. 在一次提交后，一个新的段被添加到提交点而且 存被清空。

当一个 被触 ，所有已知的段按 序被 。 会 所有段的 果 行聚合，以保 个 和 个文 的 都被准 算。 方式可以用相 低的成本将新文 添加到索引。

除和更新

段是不可改的，所以既不能从把文 从旧的段中移除，也不能修改旧的段来 行反映文 的更新。取而代之的是， 个提交点会包含一个 `.del` 文件，文件中会列出 些被 除文 的段信息。

当一个文 被“ 除” ，它 上只是在 `.del` 文件中被 除。一个被 除的文 然可以被匹配到，但它会在最 果被返回前从 果集中移除。

文 更新也是 似的操作方式：当一个文 被更新 ，旧版本文 被 除，文 的新版本被索引到一个新的段中。 可能 个版本的文 都会被一个 匹配到，但被 除的那个旧版本文 在 果集返回前就已被移除。

在 段合并，我 展示了一个被 除的文 是 被文件系 移除的。

近 搜索

随着按段 (per-segment) 搜索的 展，一个新的文 从索引到可被搜索的延 著降低了。新文 在几分 之内即可被 索，但 是不 快。

磁 在 里成 了瓶 。提交 (Committing) 一个新的段到磁 需要一个 `fsync` 来保段被物理性地写入磁 ， 在断 的 候就不会 失数据。 但是 `fsync` 操作代 很大； 如果次索引一个文 都去 行一次的 会造成很大的性能 。

我 需要的是一个更 量的方式来使一个文 可被搜索， 意味着 `fsync` 要从整个 程中被移除。

在Elasticsearch和磁 之 是文件系 存。 像之前描述的一 ， 在内存索引 冲区（ 在内存冲区中包含了新文 的 Lucene 索引 ）中的文 会被写入到一个新的段中（ 冲区的内容已被写入一个可被搜索的段中，但 没有 行提交 ）。 但是 里新段会被先写入到文件系 存— 一代 会比 低， 后再被刷新到磁 — 一代 比 高。不 只要文件已 在 存中， 就可以像其它文件一 被打 和 取了。

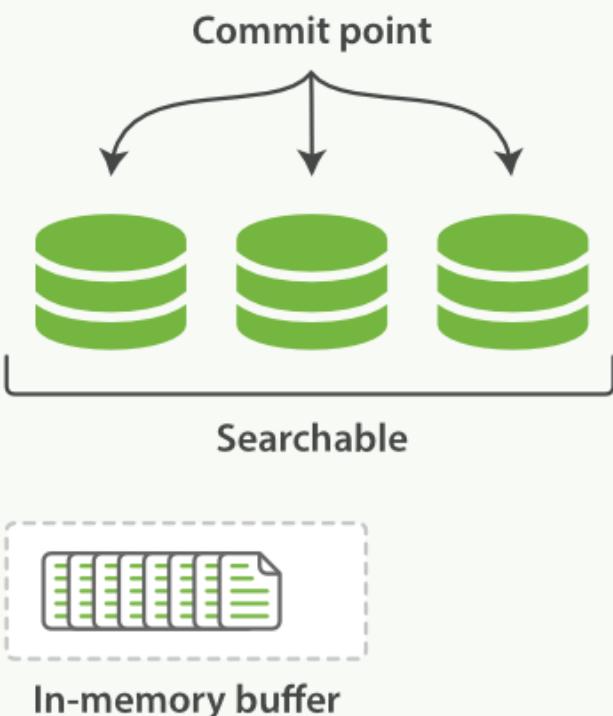


Figure 19. 在内存缓冲区中包含了新文档的Lucene索引

Lucene 允许新段被写入和打开 —使其包含的文档在未进行一次完整提交便能搜索到。这种方式比进行一次提交代价要小得多，并且在不影响性能的前提下可以被频繁地执行。

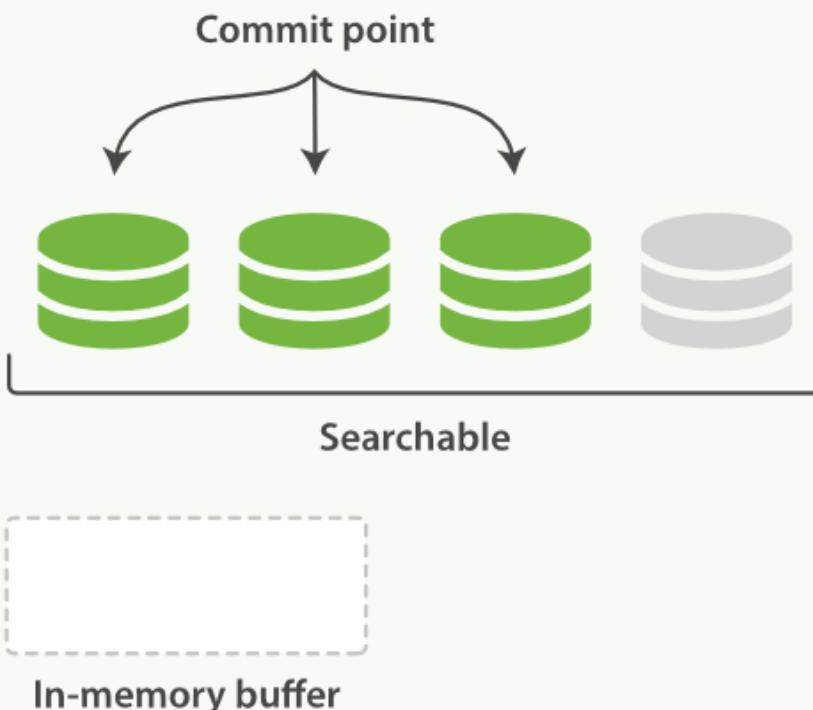


Figure 20. 缓冲区的内容已被写入一个可被搜索的段中，但没有进行提交

refresh API

在 Elasticsearch 中，写入和打一个新段的量的程叫做 `refresh`。情况下个分片会秒自刷新一次。就是什我 Elasticsearch 是近搜索：文的化并不是立即搜索可，但会在一秒之内可。

些行可能会新用造成困惑：他索引了一个文然后搜索它，但却没有搜到。个的解决法是用 `refresh` API 行一次手刷新：

```
POST /_refresh ①  
POST /blogs/_refresh ②
```

① 刷新（Refresh）所有的索引。

② 只刷新（Refresh）`blogs` 索引。

TIP 尽管刷新是比提交量很多的操作，它是会有性能。当写的时候，手刷新很有用，但是不要在生境下次索引一个文都去手刷新。相反，的用需要意到 Elasticsearch 的近的性，并接受它的不足。

并不是所有的情况都需要秒刷新。可能正在使用 Elasticsearch 索引大量的日志文件，可能想化索引速度而不是近搜索，可以通置 `refresh_interval`，降低个索引的刷新率：

```
PUT /my_logs  
{  
  "settings": {  
    "refresh_interval": "30s" ①  
  }  
}
```

① 30秒刷新 `my_logs` 索引。

`refresh_interval`可以在既存索引上行更新。在生境中，当正在建立一个大的新索引，可以先自刷新，待始使用索引，再把它回来：

```
PUT /my_logs/_settings  
{ "refresh_interval": -1 } ①  
  
PUT /my_logs/_settings  
{ "refresh_interval": "1s" } ②
```

① 自刷新。

② 秒自刷新。

CAUTION `refresh_interval` 需要一个持，例如 `1s` (1秒) 或 `2m` (2分钟)。一个 `1` 表示的是 1 秒 -- 无疑会使的集群陷入。

持久化 更

如果没有用 `fsync` 把数据从文件系 存刷 (flush) 到硬 盘，我 不能保 数据在断 甚至是程序正常退出之后依然存在。为了保 Elasticsearch 的可 性，需要 保数据 化被持久化到磁 盘。

在 [更新索引](#)，我 一次完整的提交会将段刷到磁 盘，并写入一个包含所有段列表的提交点。Elasticsearch 在 或重新打 一个索引的 程中使用 个提交点来判断 些段隶属于当前分片。

即使通 秒刷新 (refresh) 了近 搜索，我 然需要 常 行完整提交来 保能从失 中恢 。但在 次提交之 生 化的文 ？我 也不希望 失掉 些数据。

Elasticsearch 加了一个 `translog`，或者叫事 日志，在 一次 Elasticsearch 行操作 均 行了日志 。通 `translog`，整个流程看起来是下面：

1. 一个文 被索引之后，就会被添加到内存 冲区，并且 追加到了 `translog`，正如 新的文 被添加到内存 冲区并且被追加到了事 日志 描述的一 。

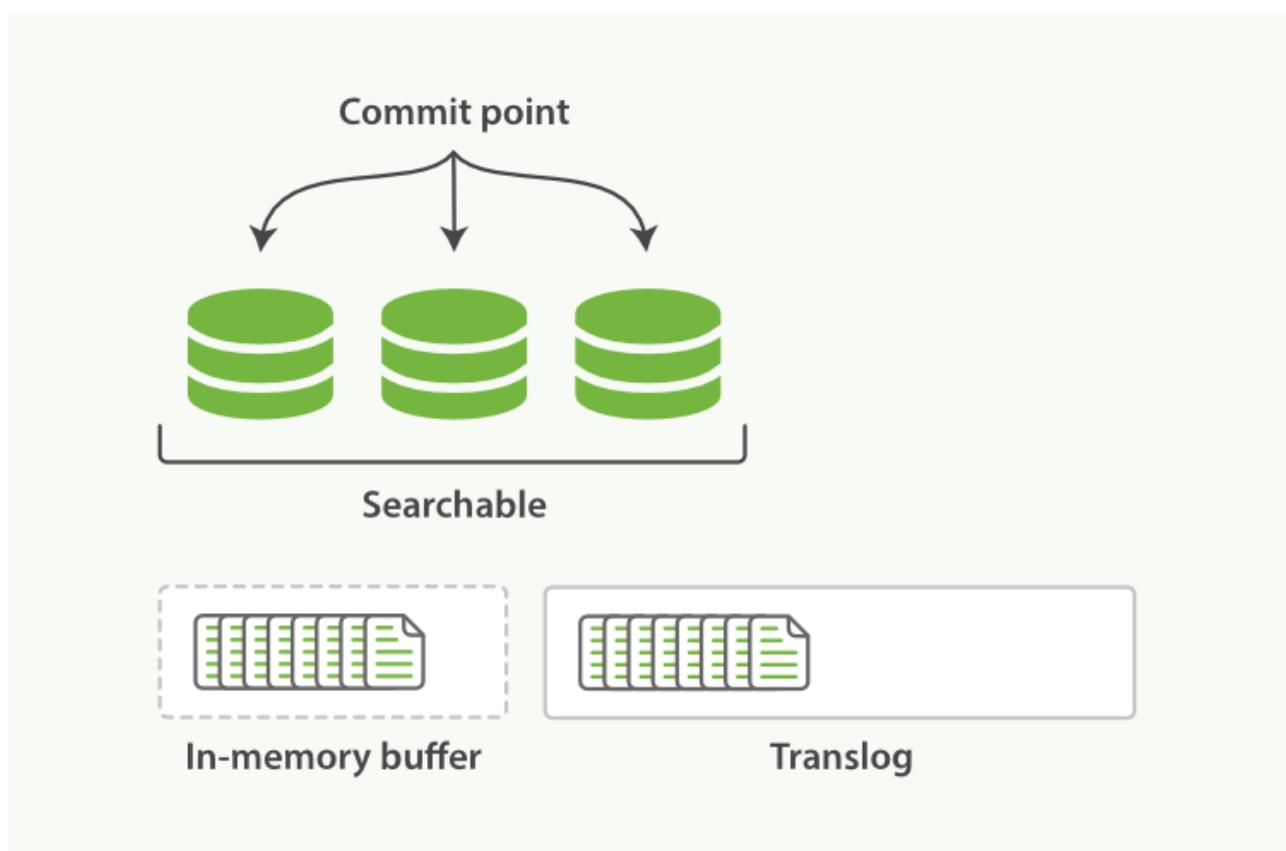


Figure 21. 新的文 被添加到内存 冲区并且被追加到了事 日志

2. 刷新 (refresh) 使分片 于 刷新 (refresh) 完成后，存被清空但是事 日志不会 描述的状 ，分片 秒被刷新 (refresh) 一次：

- 些在内存 冲区的文 被写入到一个新的段中，且没有 行 `fsync` 操作。
- 个段被打 ，使其可被搜索。
- 内存 冲区被清空。

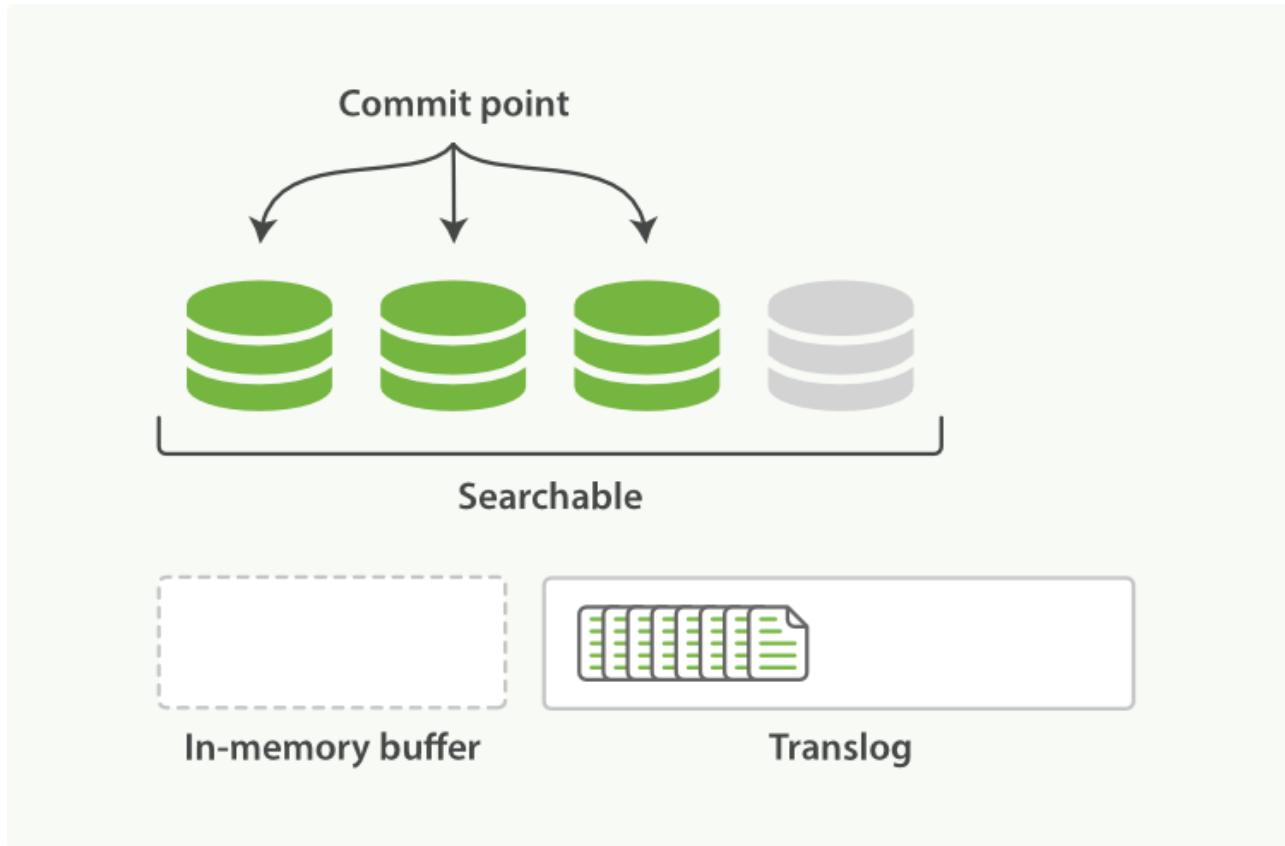


Figure 22. 刷新 (refresh) 完成后，存被清空但是事 日志不会

3. 一个 程 工作，更多的文 被添加到内存 冲区和追加到事 日志（ 事 日志不断 累文 ）。

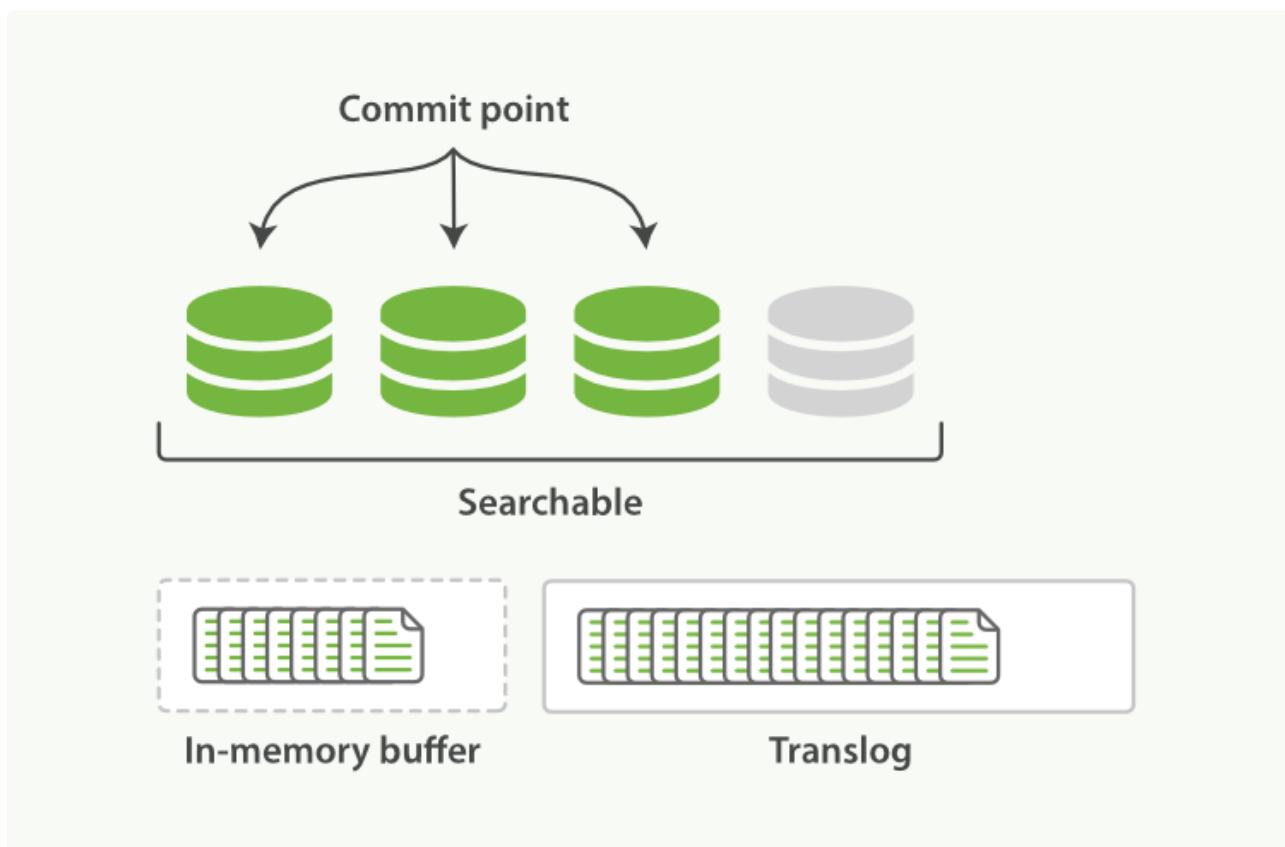


Figure 23. 事 日志不断 累文

4. 隔一段 —例如 translog 得越来越大—索引被刷新 (flush)；一个新的 translog 被建，并且一个全量提交被 行 (在刷新 (flush) 之后，段被全量提交， 并且事 日志被清空)：

- 所有在内存 冲区的文 都被写入一个新的段。
- 冲区被清空。
- 一个提交点被写入硬 。
- 文件系 存通 `fsync` 被刷新 (flush) 。
- 老的 translog 被 除。

translog 提供所有 没有被刷到磁 的操作的一个持久化 。当 Elasticsearch 的 候， 它会从磁 中使用最后一个提交点去恢 已知的段，并且会重放 translog 中所有在最后一次提交后 生的 更操作。

translog 也被用来提供 CRUD 。当 着通 ID 、更新、 除一个文 ， 它会在 从相 的段中 索之前， 首先 translog 任何最近的 更。 意味着它 是能 地 取到文 的最新版本。

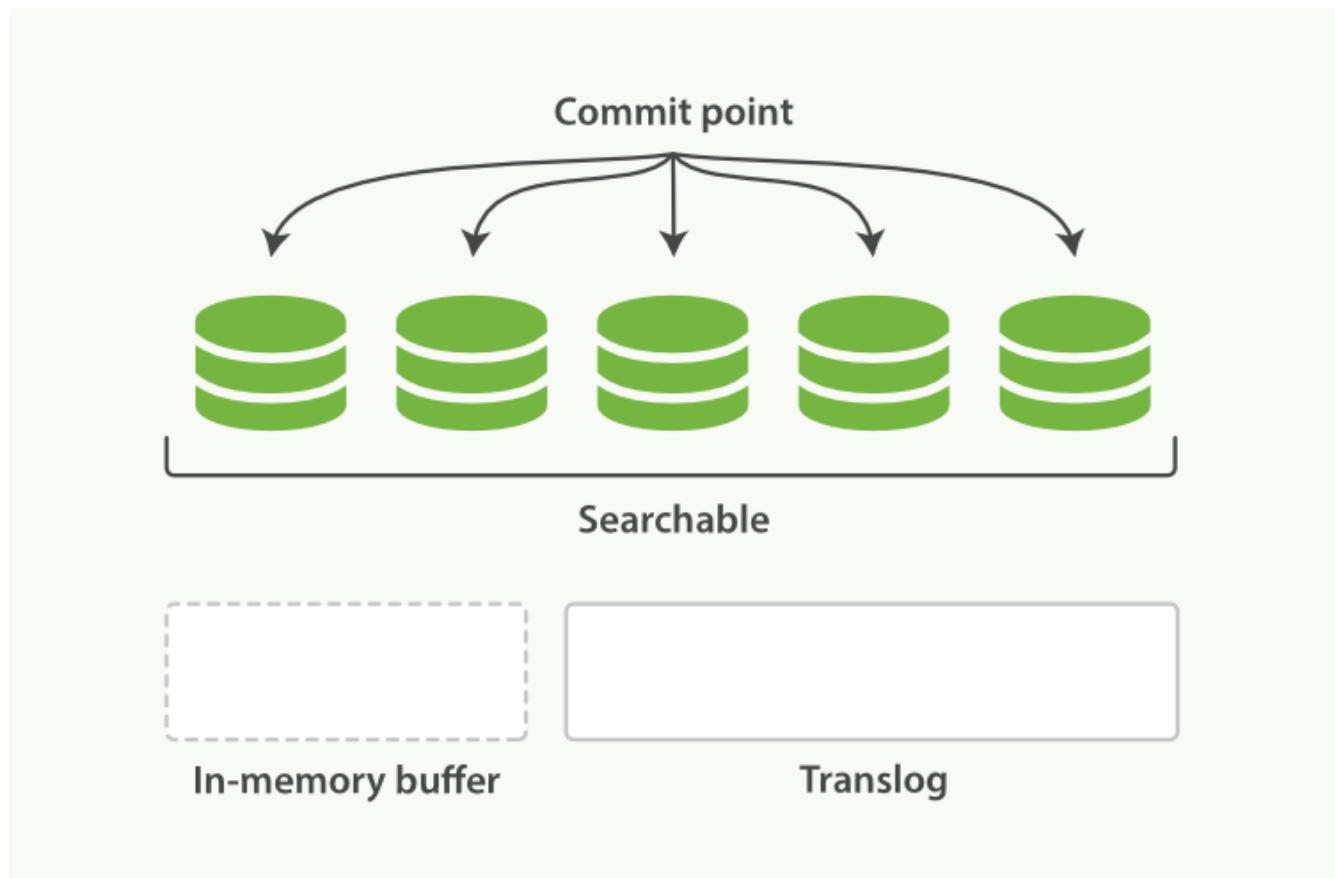


Figure 24. 在刷新 (flush) 之后，段被全量提交，并且事 日志被清空

flush API

个 行一个提交并且截断 translog 的行 在 Elasticsearch 被称作一次 `flush` 。 分片 30分 被自 刷新 (flush) , 或者在 translog 太大的 候也会刷新。 看 {ref}/index-modules-translog.html#_translog_settings[translog 文] 来 置， 它可以用来控制 些 :

{ref}/indices-flush.html[`flush` API] 可以被用来 行一个手工的刷新 (flush) :

```
POST /blogs/_flush ①
```

```
POST /_flush?wait_for_ongoing ②
```

① 刷新 (flush) `blogs` 索引。

② 刷新 (flush) 所有的索引并且等待所有刷新在返回前完成。

很少需要自己手动一个的 `flush` 操作；通常情况下，自动刷新就足够了。

就是，在重建点或索引之前执行 `flush` 有益于它的索引。当 Elasticsearch 做快照或重新打开一个索引，它需要重放 translog 中所有的操作，所以如果日志越短，恢复越快。

Translog 有多安全？

translog 的目的是保证操作不会丢失。引出了一个问题：Translog 有多安全？

在文件被 `fsync` 到磁盘前，被写入的文件在重写之后就会丢失。translog 是 5 秒被 `fsync` 刷新到硬盘，或者在一次写入完成之后（e.g. `index`, `delete`, `update`, `bulk`）。一个请求在主分片和副本分片都会产生。最坏情况下，意味着在整个请求被 `fsync` 到主分片和副本分片的 translog 之前，客户端不会得到一个 200 OK 响应。

在一次请求后都执行一个 `fsync` 会带来一些性能损失，尽管实践表明这种损失相对较小（特别是 bulk 入，它在一次请求中平均处理了大量文档）。

但是对于一些大容量的偶发性几秒数据丢失并不严重的集群，使用自动的 `fsync` 是比较有益的。比如，写入的数据被存到内存中，再每 5 秒执行一次 `fsync`。

一个请求可以通过设置 `durability` 参数为 `async` 来使用：

```
PUT /my_index/_settings
{
  "index.translog.durability": "async",
  "index.translog.sync_interval": "5s"
}
```

一个请求可以对索引独占，并且可以进行修改。如果决定使用自动的 translog，需要保证在发生 crash，丢失 `sync_interval` 范围内的数据也无所损失。在决定前知道这个特性。

如果不希望一个请求的后果，最好是使用 `request` 的参数（`"index.translog.durability": "request"`）来避免数据丢失。

段合并

由于自动生成流程每秒会建立一个新的段，会致短时间内段数量暴增。而段数目太多会带来很大的麻烦。一个段都会消耗文件句柄、内存和 CPU 行周期。更重要的是，一个搜索请求都需要流过多个段；所以段越多，搜索也就越慢。

Elasticsearch 通过后台进行段合并来解决这个问题。小的段被合并到大的段，然后一些大的段再被合并到更大的段。

段合并的时候会将那些旧的已删除文件从文件系统中清除。被删除的文件（或被更新文件的旧版本）不会被拷贝到新的大段中。

段合并不需要做任何事。索引和搜索会自行进行。这个流程像在一个提交了的段和一个未提交的段正在被合并到一个更大的段中提到的一样工作：

- 1、当索引的时候，刷新（refresh）操作会创建新的段并将段打以供搜索使用。
- 2、合并进程将一小部分大小相似的段，并且在后台将它们合并到更大的段中。并不会中断索引和搜索。

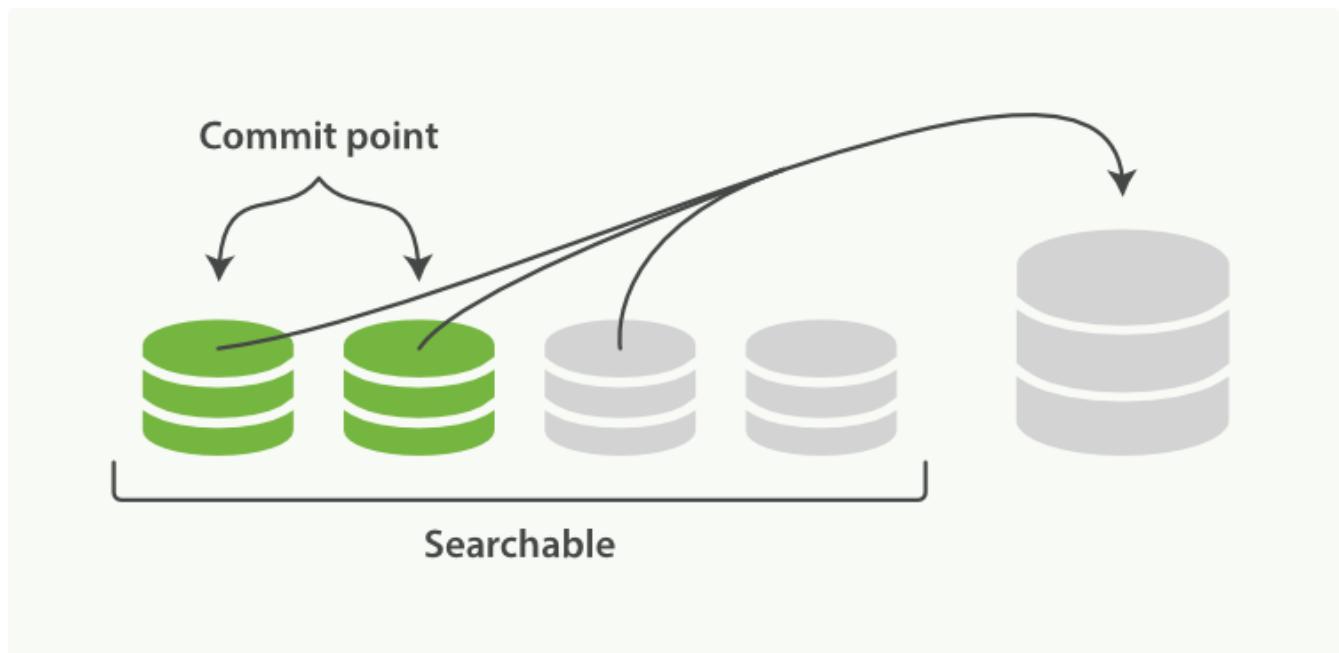


Figure 25. 一个提交了的段和一个未提交的段正在被合并到一个更大的段

- 3、一旦合并结束，老的段被删除。明合并完成的活：

- 新的段被刷新（flush）到了磁盘。
- 写入一个包含新段且排除旧的和小的段的新提交点。
- 新的段被打标记用来搜索。
- 老的段被删除。

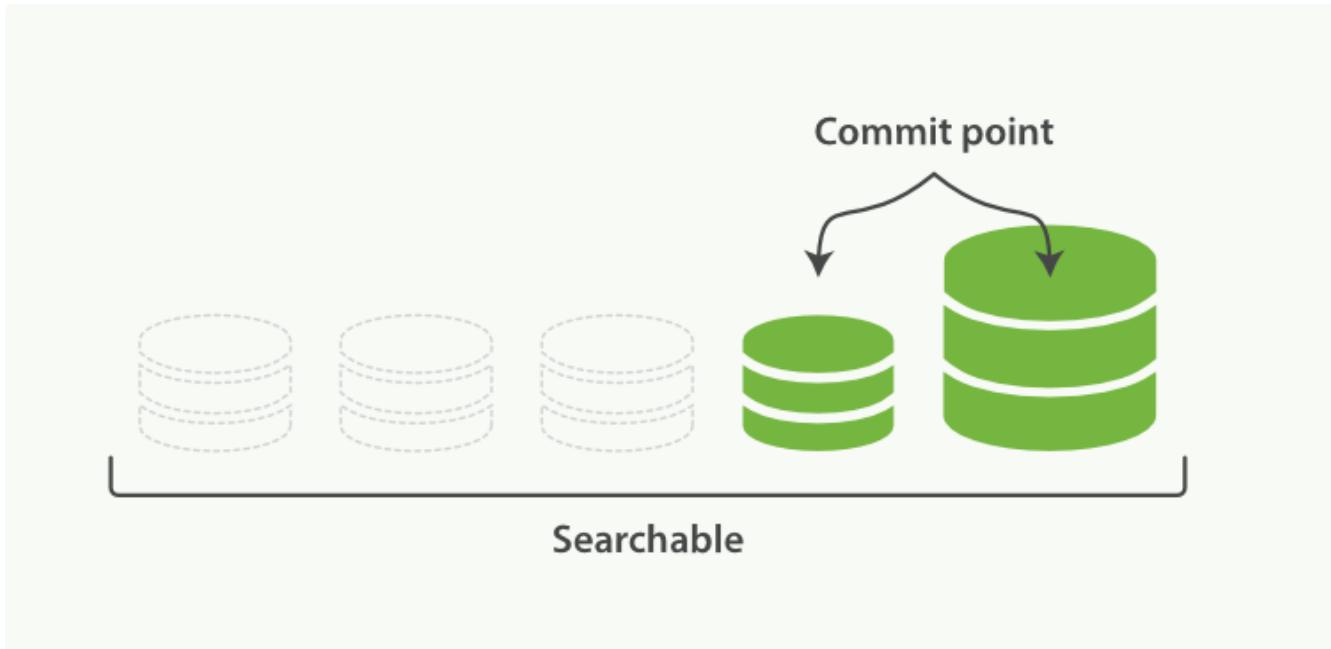


Figure 26. 一旦合并 束，老的段被 除

合并大的段需要消耗大量的I/O和CPU 源，如果任其 展会影 搜索性能。Elasticsearch在 情况下会 合并流程 行 源限制，所以搜索 然有足 的 源很好地 行。

TIP 看 [segments-and-merging] 来 的 例 取 于合并 整的建 。

optimize API

`optimize` API大可看做是 制合并 API。它会将一个分片 制合并到 `max_num_segments` 参数指定大小的段数目。 做的意 是 少段的数量（通常 少到一个），来提升搜索性能。

WARNING `optimize` API 不 被用在一个 索引——一个正在被活 更新的索引。后台合并流程已 可以很好地完成工作。 `optimizing` 会阻碍 个 程。不要干 它！

在特定情况下，使用 `optimize` API 有益 。例如在日志 用例下， 天、 周、 月的日志被存 在一个索引中。老的索引 上是只 的；它 也并不太可能会 生 化。

在 情况下，使用`optimize` 化老的索引，将 一个分片合并 一个 独的段就很有用了； 既可以 省 源，也可以使搜索更加快速：

```
POST /logstash-2014-10/_optimize?max_num_segments=1 ①
```

① 合并索引中的 个分片 一个 独的段

WARNING 注意，使用 `optimize` API 触 段合并的操作一点也不会受到任何 源上的限制。 可能会消耗掉 点上全部的I/O 源， 使其没有余裕来 理搜索 求，从而有可能使集群失去 。 如果 想要 索引 行 `optimize`， 需要先使用分片分配（ 看 [migrate-indices]）把索引移到一个安全的 点，再 行。