Elasticsearch

## 集群原理

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

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

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

curl -X GET "localhost:9200/\_cluster/health" 获取集群健康信息

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。

索引 —— 保存相关数据的地方。 索引实际上是指向一个或者多个物理 分片 的 逻辑命名空间 。

一个 分片 是一个底层的 工作单元 ，它仅保存了 全部数据中的一部分。 在[分片内部机制](https://www.elastic.co/guide/cn/elasticsearch/guide/current/inside-a-shard.html" \o "分片内部原理)中，我们将详细介绍分片是如何工作的，而现在我们只需知道一个分片是一个 Lucene 的实例，以及它本身就是一个完整的搜索引擎。 我们的文档被存储和索引到分片内，但是应用程序是直接与索引而不是与分片进行交互。

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

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

在索引建立的时候就已经确定了主分片数，但是副本分片数可以随时修改。

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

一旦一个文档被存储在 Elasticsearch 中，它就是可以被集群中的任意节点检索到。

## **数据输入输出**

### 索引文档

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

{

"field": "value",

...

}

### **取回文档**

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

curl -X GET "localhost:9200/website/blog/123?pretty"

返回文档的一部分

curl -X GET "localhost:9200/website/blog/123?\_source=title,text"

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

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

### **更新文档**

（对已经存在的文档进行全部更新）

PUT /website/blog/123

{

"title": "My first blog entry",

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

"date": "2014/01/02"

}

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

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

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

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

### **创建新文档**

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

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

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

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

PUT /website/blog/123/\_create

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

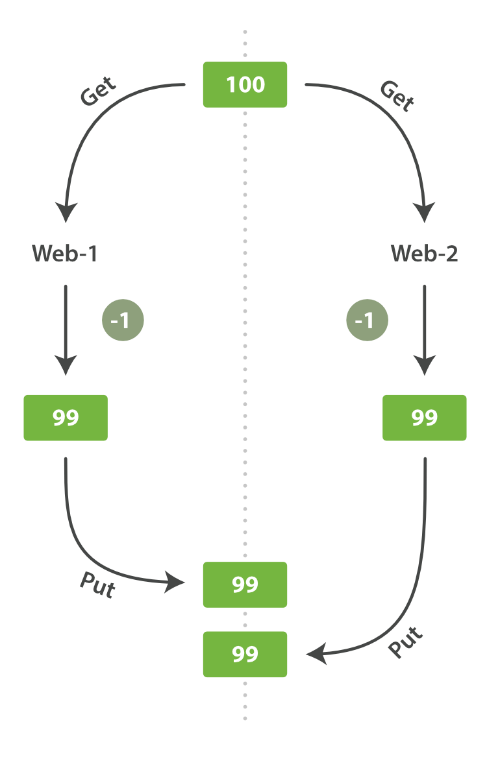
**处理冲突**

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

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

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

有一天，管理层决定做一次促销。突然地，我们一秒要卖好几个商品。 假设有两个 web 程序并行运行，每一个都同时处理所有商品的销售，如图所示。



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

在数据库领域中，有两种方法通常被用来确保并发更新时变更不会丢失：

**悲观并发控制**

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

**乐观并发控制**

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

**乐观并发控制**

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

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

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

然而，如果我们重新运行相同的索引请求，仍然指定 version=1 ， Elasticsearch 返回 409 Conflict

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

**通过外部系统使用版本控制**

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

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

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

### ****文档部分更新****

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

curl -X POST "localhost:9200/website/blog/1/\_update" -H 'Content-Type: application/json' -d'

{

"doc" : {

"tags" : [ "testing" ],

"views": 0

}

}

'

使用脚本更新部分文档

curl -X POST "localhost:9200/website/blog/1/\_update" -H 'Content-Type: application/json' -d'

{

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

}

'

retry\_on\_conflict=5 失败之前重试该更新5次。

POST /website/pageviews/1/\_update?retry\_on\_conflict=5

{

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

"upsert": {

"views": 0

}

}

### 取回多个文档

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

GET /\_mget

{

"docs" : [

{

"\_index" : "website",

"\_type" : "blog",

"\_id" : 2

},

{

"\_index" : "website",

"\_type" : "pageviews",

"\_id" : 1,

"\_source": "views"

}

]

}

NOTE:

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

### 代价较小的批量操作

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

bulk 与其他的请求体格式稍有不同，如下所示：

{ action: { metadata }}\n

{ request body }\n

{ action: { metadata }}\n

{ request body }\n

action/metadata 行指定 *哪一个文档* 做 *什么操作* 。

action 必须是以下选项之一:

create

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

index

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

update

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

delete

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

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

例如，一个 delete 请求看起来是这样的：

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

例

POST /\_bulk

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

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

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

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

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

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

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

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

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

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

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

## 分布式文档存储

文件是如何分布到集群的，又是如何从集群中获取的。 Elasticsearch 本意就是隐藏这些底层细节，让我们好专注在业务开发中，所以其实你不必了解这么深入也无妨。

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

### 路由一个文档到一个分片当中

当索引一个文档的时候，文档会被存储到一个主分片中。 Elasticsearch 如何知道一个文档应该存放到哪个分片中呢？

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

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

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

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

我们可以发送请求到集群中的任一节点。 每个节点都有能力处理任意请求。 每个节点都知道集群中任一文档位置，所以可以直接将请求转发到需要的节点上

### 新建、索引、删除文档

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

在下面的例子中，将所有的请求发送到 Node 1 ，我们将其称为 *协调节点(coordinating node)* 。

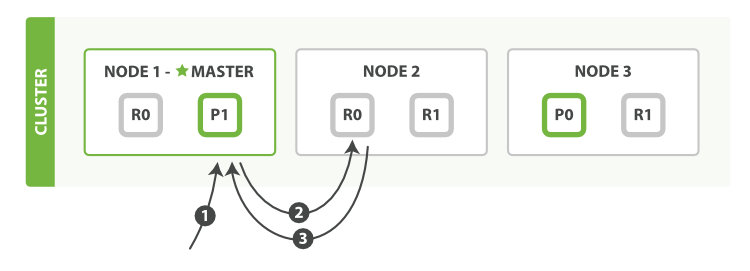


以下是在主副分片和任何副本分片上面 成功新建，索引和删除文档所需要的步骤顺序：

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

### 取回一个文档

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



以下是从主分片或者副本分片检索文档的步骤顺序：

1、客户端向 Node 1 发送获取请求。

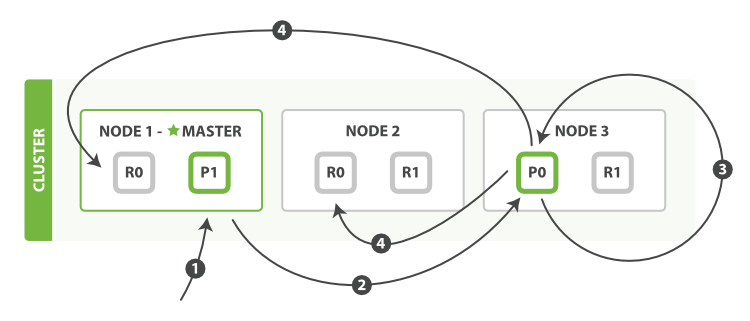
2、节点使用文档的 \_id 来确定文档属于分片 0 。分片 0 的副本分片存在于所有的三个节点上。 在这种情况下，它将请求转发到 Node 2 。

3、Node 2 将文档返回给 Node 1 ，然后将文档返回给客户端。

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

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

### 局部更新文档



以下是部分更新一个文档的步骤：

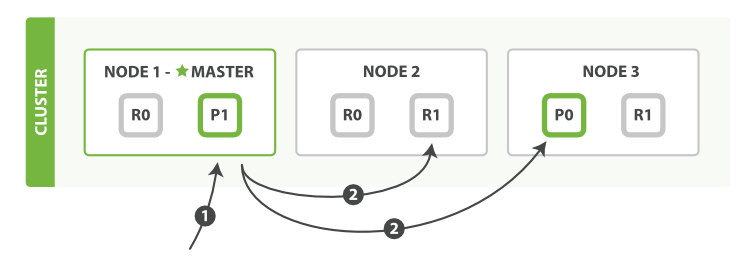
1. 客户端向 Node 1 发送更新请求。
2. 它将请求转发到主分片所在的 Node 3 。
3. Node 3 从主分片检索文档，修改 \_source 字段中的 JSON ，并且尝试重新索引主分片的文档。 如果文档已经被另一个进程修改，它会重试步骤 3 ，超过 retry\_on\_conflict 次后放弃。
4. 如果 Node 3 成功地更新文档，它将新版本的文档并行转发到 Node 1 和 Node 2 上的副本分片，重新建立索引。 一旦所有副本分片都返回成功， Node 3 向协调节点也返回成功，协调节点向客户端返回成功。

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

### 多文档模式

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

协调节点一旦收到来自每个节点的应答，就将每个节点的响应收集整理成单个响应，返回给客户端，如

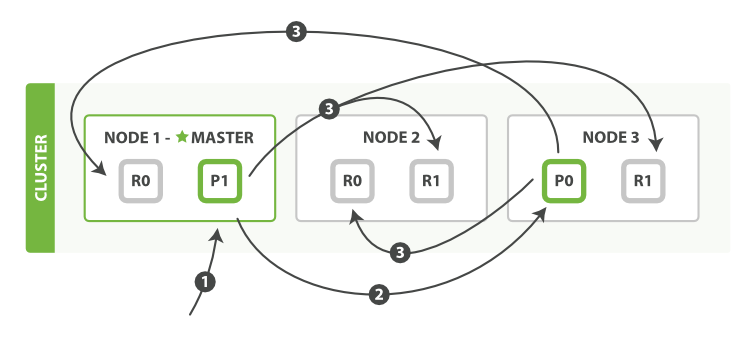


以下是使用单个 mget 请求取回多个文档所需的步骤顺序：

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

可以对 docs 数组中每个文档设置 routing 参数。

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



bulk API 按如下步骤顺序执行：

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

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

## 搜索—最基本的工具

### 多索引，多类型

GET /\_search

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

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

/\_search

在所有的索引中搜索所有的类型

/gb/\_search

在 gb 索引中搜索所有的类型

/gb,us/\_search

在 gb 和 us 索引中搜索所有的文档

/g\*,u\*/\_search

在任何以 g 或者 u 开头的索引中搜索所有的类型

/gb/user/\_search

在 gb 索引中搜索 user 类型

/gb,us/user,tweet/\_search

在 gb 和 us 索引中搜索 user 和 tweet 类型

/\_all/user,tweet/\_search

在所有的索引中搜索 user 和 tweet 类型

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

### 分页

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

Size：显示应该返回的结果数量，默认是 10

from：显示应该跳过的初始结果数量，默认是 0

GET /\_search?size=5

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

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

**在分布式系统中深度分页**

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

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

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