3.1 文档

**什么是文档？**

程序中大多的实体或对象能够被序列化为包含键值对的JSON对象，**键(key)**是**字段(field)**或**属性(property)**的名字，**值(value)**可以是字符串、数字、布尔类型、另一个对象、值数组或者其他特殊类型，比如表示日期的字符串或者表示地理位置的对象（me：值的合理类型就是json的数据类型）。

{

"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，我们可以认为**对象(object)**和**文档(document)**是等价相通的。不过，他们还是有所差别：对象(Object)是一个JSON结构体——类似于哈希、hashmap、字典或者关联数组；对象(Object)中还可能包含其他对象(Object)。 在Elasticsearch中，**文档(document)**这个术语有着特殊含义。它特指**最顶层**结构或者**根对象(root object)**序列化成的JSON数据（以唯一ID标识并存储于Elasticsearch中）。

**文档元数据**

一个文档不只有数据。它还包含了**元数据(metadata)**——**关于**文档的信息。三个必须的元数据节点是：

| **节点** | **说明** |
| --- | --- |
| \_index | 文档存储的地方 |
| \_type | 文档代表的对象的类 |
| \_id | 文档的唯一标识 |

**\_index**

**索引(index)**类似于关系型数据库里的“数据库”——它是我们存储和索引关联数据的地方。

提示：

事实上，我们的数据被存储和索引在**分片(shards)**中，索引只是一个把一个或多个分片分组在一起的**逻辑空间**。然而，这只是一些内部细节——我们的程序完全不用关心分片。对于我们的程序而言，文档存储在**索引(index)**中。剩下的细节由Elasticsearch关心既可。

我们将会在《索引管理》章节中探讨如何创建并管理索引，但现在，我们将让Elasticsearch为我们创建索引。我们唯一需要做的仅仅是选择一个索引名。这个名字必须是全部小写，不能以下划线开头，不能包含逗号。让我们使用website做为索引名。

**\_type**

在应用中，我们使用对象表示一些“事物”，例如一个用户、一篇博客、一个评论，或者一封邮件。每个对象都属于一个**类(class)**，这个类定义了属性或与对象关联的数据。user类的对象可能包含姓名、性别、年龄和Email地址。

在关系型数据库中，我们经常将相同类的对象存储在一个表里，因为它们有着相同的结构。同理，在Elasticsearch中，我们使用相同**类型(type)**的文档表示相同的“事物”，因为他们的数据结构也是相同的。

每个**类型(type)**都有自己的**映射(mapping)**或者结构定义，就像传统数据库表中的列一样。所有类型下的文档被存储在同一个索引下，但是类型的**映射(mapping)**会告诉Elasticsearch不同的文档如何被索引。 我们将会在《映射》章节探讨如何定义和管理映射，但是现在我们将依赖Elasticsearch去自动处理数据结构。

\_type的名字可以是大写或小写，不能包含下划线或逗号。我们将使用blog做为类型名。

**\_id**

**id**仅仅是一个字符串，它与\_index和\_type组合时，就可以在Elasticsearch中**唯一标识一个文档**。当创建一个文档，你可以自定义\_id，也可以让Elasticsearch帮你自动生成。

**其它元数据**

还有一些其它的元数据，我们将在《映射》章节探讨。使用上面提到的元素，我们已经可以在Elasticsearch中存储文档并通过ID检索——换言说，把Elasticsearch做为文档存储器使用了。

3.2 索引

## 索引一个文档

文档通过index API被索引——使数据可以被存储和搜索。但是首先我们需要决定文档所在。正如我们讨论的，文档通过其\_index、\_type、\_id唯一确定。们可以自己提供一个\_id，或者也使用index API 为我们生成一个。

### 使用自己的ID

如果你的文档有自然的标识符（例如user\_account字段或者其他值表示文档），你就可以提供自己的\_id，使用这种形式的index API：

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

{

"field": "value",

...

}

例如我们的索引叫做“website”，类型叫做“blog”，我们选择的ID是“123”，那么这个索引请求就像这样：

PUT /website/blog/123

{

"title": "My first blog entry",

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

"date": "2014/01/01"

}

Elasticsearch的响应：

{

"\_index": "website",

"\_type": "blog",

"\_id": "123",

"\_version": 1,

"created": true

}

响应指出请求的索引已经被成功创建，这个索引中包含\_index、\_type和\_id元数据，以及一个新元素：\_version。

Elasticsearch中每个文档都有版本号，每当文档变化（包括删除）都会使\_version增加。在《版本控制》章节中我们将探讨如何使用\_version号确保你程序的一部分不会覆盖掉另一部分所做的更改。

### 自增ID

如果我们的数据没有自然ID，我们可以让Elasticsearch自动为我们生成。请求结构发生了变化：PUT方法——“在这个URL中存储文档”变成了POST方法——"在这个类型下存储文档"。（译者注：原来是把文档存储到某个ID对应的空间，现在是把这个文档添加到某个\_type下）。

URL现在只包含\_index和\_type两个字段：

POST /website/blog/

{

"title": "My second blog entry",

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

"date": "2014/01/01"

}

响应内容与刚才类似，只有\_id字段变成了自动生成的值：

{

"\_index": "website",

"\_type": "blog",

"\_id": "wM0OSFhDQXGZAWDf0-drSA",

"\_version": 1,

"created": true

}

自动生成的ID有22个字符长，URL-safe, Base64-encoded string universally unique identifiers, 或者叫 [UUIDs](http://en.wikipedia.org/wiki/Uuid)。

3.3获取

## 检索文档

想要从Elasticsearch中获取文档，我们使用同样的\_index、\_type、\_id，但是HTTP方法改为GET：

GET /website/blog/123?pretty

响应包含了现在熟悉的元数据节点，增加了\_source字段，它包含了在创建索引时我们发送给Elasticsearch的原始文档。

{

"\_index" : "website",

"\_type" : "blog",

"\_id" : "123",

"\_version" : 1,

"found" : true,

"\_source" : {

"title": "My first blog entry",

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

"date": "2014/01/01"

}

}

### pretty

在任意的查询字符串中增加pretty参数，类似于上面的例子。会让Elasticsearch**美化输出(pretty-print)**JSON响应以便更加容易阅读。\_source字段不会被美化，它的样子与我们输入的一致。

GET请求返回的响应内容包括{"found": true}。这意味着文档已经找到。如果我们请求一个不存在的文档，依旧会得到一个JSON，不过found值变成了false。

此外，HTTP响应状态码也会变成'404 Not Found'代替'200 OK'。我们可以在curl后加-i参数得到响应头：

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

现在响应类似于这样：

HTTP/1.1 404 Not Found

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

Content-Length: 83

{

"\_index" : "website",

"\_type" : "blog",

"\_id" : "124",

"found" : false

}

### 检索文档的一部分

通常，GET请求将返回文档的全部，存储在\_source参数中。但是可能你感兴趣的字段只是title。请求个别字段可以使用\_source参数。多个字段可以使用逗号分隔：

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

\_source字段现在只包含我们请求的字段，而且过滤了date字段：

{

"\_index" : "website",

"\_type" : "blog",

"\_id" : "123",

"\_version" : 1,

"exists" : true,

"\_source" : {

"title": "My first blog entry" ,

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

}

}

或者你只想得到\_source字段而不要其他的元数据，你可以这样请求：

GET /website/blog/123/\_source

它仅仅返回:

{

"title": "My first blog entry",

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

"date": "2014/01/01"

}

3.4存在

## 检查文档是否存在

如果你想做的只是检查文档是否存在——你对内容完全不感兴趣——使用HEAD方法来代替GET。HEAD请求不会返回响应体，只有HTTP头：

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

Elasticsearch将会返回200 OK状态如果你的文档存在：

HTTP/1.1 200 OK

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

Content-Length: 0

如果不存在返回404 Not Found：

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，这只表示你在查询的那一刻文档不存在，但并不表示几毫秒后依旧不存在。另一个进程在这期间可能创建新文档。

3.5 更新

**更新整个文档**

文档在Elasticsearch中是不可变的——我们不能修改他们。如果需要更新已存在的文档，我们可以使用《索引文档》章节提到的index API *重建索引(reindex)* 或者替换掉它。

PUT /website/blog/123

{

"title": "My first blog entry",

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

"date": "2014/01/02"

}

在响应中，我们可以看到Elasticsearch把\_version增加了。

1

{

"\_index" : "website",

"\_type" : "blog",

"\_id" : "123",

"\_version" : 2,

"created": false <1>

}

* <1> created标识为false因为同索引、同类型下已经存在同ID的文档（me:这是一条update）。

在内部，Elasticsearch已经标记旧文档为删除并添加了一个完整的新文档。旧版本文档不会立即消失，但你也不能去访问它。Elasticsearch会在你继续索引更多数据时清理被删除的文档。

在本章的后面，我们将会在《局部更新》中探讨update API。这个API *似乎* 允许你修改文档的局部，但事实上Elasticsearch遵循与之前所说完全相同的过程，这个过程如下：

1. 从旧文档中检索JSON
2. 修改它
3. 删除旧文档
4. 索引新文档

（me:不是在以前的基础上改）

唯一的不同是update API完成这一过程只需要一个客户端请求既可，不再需要get和index请求了。

3.6 创建

## 创建一个新文档

当索引一个文档，我们如何确定是完全创建了一个新的还是覆盖了一个已经存在的呢？（me:看返回的create为true还是false，或者看result的值是create还是update）

请记住\_index、\_type、\_id三者唯一确定一个文档。所以要想保证文档是新加入的，最简单的方式是使用POST方法让Elasticsearch自动生成唯一\_id：

POST /website/blog/

{ ... }

(me：通过post方式然es自动生成id使得所有的文档都是新创建的)

然而，如果想使用自定义的\_id，我们必须告诉Elasticsearch应该在\_index、\_type、\_id三者都不同时才接受请求。为了做到这点有两种方法，它们其实做的是同一件事情。你可以选择适合自己的方式：

1

第一种方法使用op\_type查询参数：

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

{ ... }

或者第二种方法是在URL后加/\_create做为端点（me：与上面的方法等价，是一种简单写法）：

PUT /website/blog/123/\_create

{ ... }

如果请求成功的创建了一个新文档，Elasticsearch将返回正常的元数据且响应状态码是201 Created。

另一方面，如果包含相同的\_index、\_type和\_id的文档已经存在，Elasticsearch将返回409 Conflict响应状态码，错误信息类似如下：

{

"error" : "DocumentAlreadyExistsException[[website][4] [blog][123]:

document already exists]",

"status" : 409

}

Me:新版的响应如下：

{

"error": {

"root\_cause": [

{

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

"reason": "[blog][123]: version conflict, document already exists (current version [3])",

"index\_uuid": "rE0Zm\_PNReybaBoOI60HFg",

"shard": "0",

"index": "website"

}

],

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

"reason": "[blog][123]: version conflict, document already exists (current version [3])",

"index\_uuid": "rE0Zm\_PNReybaBoOI60HFg",

"shard": "0",

"index": "website"

},

"status": 409

}

## 3.7 删除

## 删除文档

删除文档的语法模式与之前基本一致，只不过要使用DELETE方法：

DELETE /website/blog/123

如果文档被找到，Elasticsearch将返回200 OK状态码和以下响应体。注意\_version数字已经增加了。

{

"found" : true,

"\_index" : "website",

"\_type" : "blog",

"\_id" : "123",

"\_version" : 3

}

如果文档未找到，我们将得到一个404 Not Found状态码，响应体是这样的：

{

"found" : false,

"\_index" : "website",

"\_type" : "blog",

"\_id" : "123",

"\_version" : 4

}

尽管文档不存在——"found"的值是false——\_version依旧增加了。这是内部记录的一部分，它确保在多节点间不同操作可以有正确的顺序。

3

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

3.8 版本控制

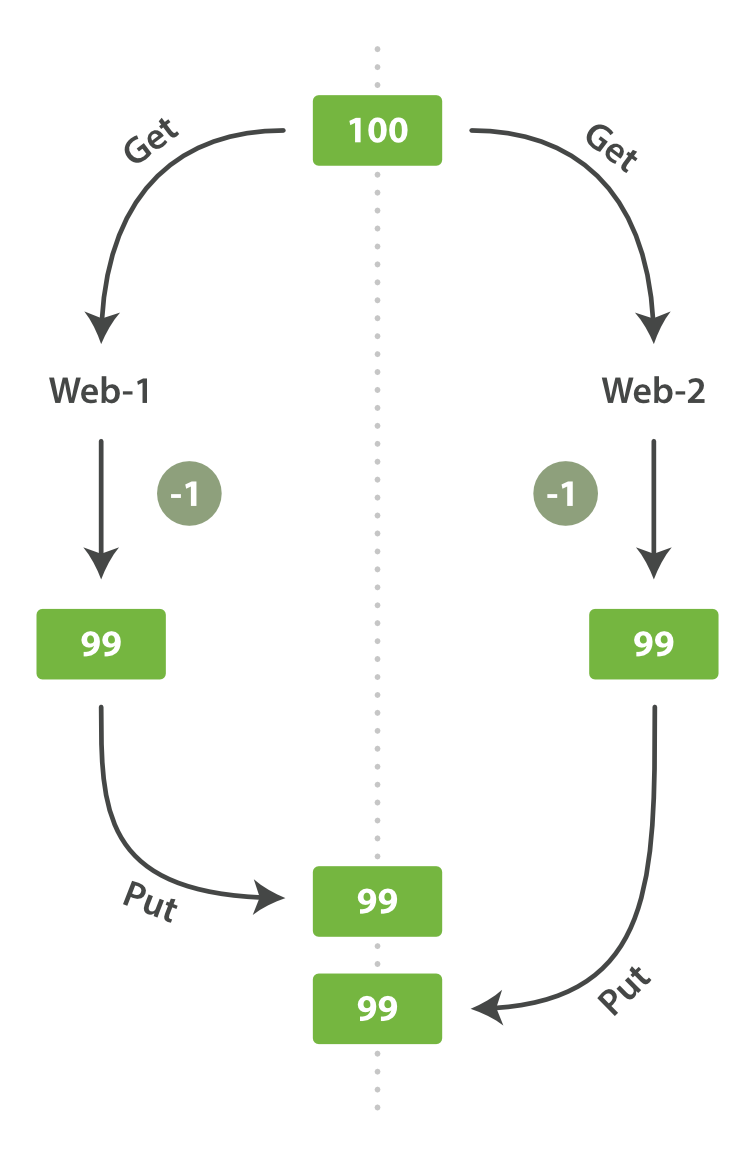
## 处理冲突

当使用index API更新文档的时候，我们读取原始文档，做修改，然后将**整个文档(whole document)**一次性重新索引。最近的索引请求会生效——Elasticsearch中只存储最后被索引的任何文档。如果其他人同时也修改了这个文档，他们的修改将会丢失。

很多时候，这并不是一个问题。或许我们主要的数据存储在关系型数据库中，然后拷贝数据到Elasticsearch中只是为了可以用于搜索。或许两个人同时修改文档的机会很少。亦或者偶尔的修改丢失对于我们的工作来说并无大碍。

但有时丢失修改是一个**很严重**的问题。想象一下我们使用Elasticsearch存储大量在线商店的库存信息。每当销售一个商品，Elasticsearch中的库存就要减一。

一天，老板决定做一个促销。瞬间，我们每秒就销售了几个商品。想象两个同时运行的web进程，两者同时处理一件商品的订单：



web\_1让stock\_count失效是因为web\_2没有察觉到stock\_count的拷贝已经过期（译者注：web\_1取数据，减一后更新了stock\_count。可惜在web\_1更新stock\_count前它就拿到了数据，这个数据已经是过期的了，当web\_2再回来更新stock\_count时这个数字就是错的。这样就会造成看似卖了一件东西，其实是卖了两件，这个应该属于幻读。）。结果是我们认为自己确实还有更多的商品，最终顾客会因为销售给他们没有的东西而失望。

2

变化越是频繁，或读取和更新间的时间越长，越容易丢失我们的更改。

在数据库中，有两种通用的方法确保在并发更新时修改不丢失：

### 悲观并发控制（Pessimistic concurrency control）

这在关系型数据库中被广泛的使用，假设冲突的更改经常发生，为了解决冲突我们把访问区块化。典型的例子是在读一行数据前锁定这行，然后确保只有加锁的那个线程可以修改这行数据。

2

### 乐观并发控制（Optimistic concurrency control）：

被Elasticsearch使用，假设冲突不经常发生，也不区块化访问，然而，如果在读写过程中数据发生了变化，更新操作将失败。这时候由程序决定在失败后如何解决冲突。实际情况中，可以重新尝试更新，刷新数据（重新读取）或者直接反馈给用户。

1

## 乐观并发控制

Elasticsearch是分布式的。当文档被创建、更新或删除，文档的新版本会被复制到集群的其它节点。Elasticsearch即是同步的又是异步的，意思是这些复制请求都是平行发送的，并**无序(out of sequence)**的到达目的地。这就需要一种方法确保老版本的文档永远不会覆盖新的版本。

1

上文我们提到index、get、delete请求时，我们指出每个文档都有一个\_version号码，这个号码在文档被改变时加一。Elasticsearch使用这个\_version保证所有修改都被正确排序。当一个旧版本出现在新版本之后，它会被简单的忽略。

我们利用\_version的这一优点确保数据不会因为修改冲突而丢失。我们可以指定文档的version来做想要的更改。如果那个版本号不是现在的，我们的请求就失败了。（me:类似可以在数据库中增加一个字段控制并发，如果该字段的值表明我们不能更新则更新失败）

Let's create a new blog post: 让我们创建一个新的博文：

PUT /website/blog/1/\_create

{

"title": "My first blog entry",

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

}

响应体告诉我们这是一个新建的文档，它的\_version是1。现在假设我们要编辑这个文档：把数据加载到web表单中，修改，然后保存成新版本。

首先我们检索文档：

GET /website/blog/1

响应体包含相同的\_version是1

{

"\_index" : "website",

"\_type" : "blog",

"\_id" : "1",

"\_version" : 1,

"found" : true,

"\_source" : {

"title": "My first blog entry",

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

}

}

现在，当我们通过重新索引文档保存修改时，我们这样指定了version参数：

2

PUT /website/blog/1?version=1 <1>

{

"title": "My first blog entry",

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

}

* <1> 我们只希望文档的\_version是1时更新才生效。

This request succeeds, and the response body tells us that the \_version has been incremented to 2:

请求成功，响应体告诉我们\_version已经增加到2：

1

{

"\_index": "website",

"\_type": "blog",

"\_id": "1",

"\_version": 2

"created": false

}

然而，如果我们重新运行相同的索引请求，依旧指定version=1，Elasticsearch将返回409 Conflict状态的HTTP响应。响应体类似这样：

{

"error" : "VersionConflictEngineException[[website][2] [blog][1]:

version conflict, current [2], provided [1]]",

"status" : 409

}

me:现在的版本如下：

{

"error": {

"root\_cause": [

{

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

"reason": "[blog][1]: version conflict, current version [2] is different than the one provided [1]",

"index\_uuid": "rE0Zm\_PNReybaBoOI60HFg",

"shard": "3",

"index": "website"

}

],

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

"reason": "[blog][1]: version conflict, current version [2] is different than the one provided [1]",

"index\_uuid": "rE0Zm\_PNReybaBoOI60HFg",

"shard": "3",

"index": "website"

},

"status": 409

}

这告诉我们当前\_version是2，但是我们指定想要更新的版本是1。

我们需要做什么取决于程序的需求。我们可以告知用户其他人修改了文档，你应该在保存前再看一下。而对于上文提到的商品stock\_count，我们需要重新检索最新文档然后申请新的更改操作。

所有更新和删除文档的请求都接受version参数，它可以允许在你的代码中增加乐观锁控制。

## 使用外部版本控制系统

一种常见的结构是使用一些其他的数据库做为主数据库，然后使用Elasticsearch搜索数据，这意味着所有主数据库发生变化，就要将其拷贝到Elasticsearch中。如果有多个进程负责这些数据的同步，就会遇到上面提到的并发问题。

如果主数据库有版本字段——或一些类似于timestamp等可以用于版本控制的字段——是你就可以在Elasticsearch的查询字符串后面添加version\_type=external来使用这些版本号。版本号必须是整数，大于零小于9.2e+18——Java中的正的long。

1

外部版本号与之前说的内部版本号在处理的时候有些不同。它不再检查\_version是否与请求中指定的**一致**，而是检查是否**小于**指定的版本。如果请求成功，外部版本号就会被存储到\_version中。

外部版本号不仅在索引和删除请求中指定，也可以在**创建(create)**新文档中指定。

例如，创建一个包含外部版本号5的新博客，我们可以这样做：（me：version可以自定义）

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

{

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

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

}

在响应中，我们能看到当前的\_version号码是5：

{

"\_index": "website",

"\_type": "blog",

"\_id": "2",

"\_version": 5,

"created": true

}

现在我们更新这个文档，指定一个新version号码为10：

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

{

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

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

}

请求成功的设置了当前\_version为10：

{

"\_index": "website",

"\_type": "blog",

"\_id": "2",

"\_version": 10,

"created": false

}

如果你重新运行这个请求，就会返回一个像之前一样的冲突错误，因为指定的外部版本号不大于当前在Elasticsearch中的版本。

3.9 文档局部更新

## 文档局部更新

在《更新文档》一章，我们说了一种通过检索，修改，然后重建整文档的索引方法来更新文档。这是对的。然而，使用update API，我们可以使用一个请求来实现局部更新，例如增加数量的操作。

我们也说过文档是不可变的——它们不能被更改，只能被替换。update API**必须**遵循相同的规则。表面看来，我们似乎是局部更新了文档的位置，内部却是像我们之前说的一样简单的使用update API处理相同的检索-修改-重建索引流程，我们也减少了其他进程可能导致冲突的修改。

最简单的update请求表单接受一个局部文档参数doc，它会合并到现有文档中——对象合并在一起，存在的标量字段被覆盖，新字段被添加。举个例子，我们可以使用以下请求为博客添加一个tags字段和一个views字段：

POST /website/blog/1/\_update

{

"doc" : {

"tags" : [ "testing" ],

"views": 0

}

}

如果请求成功，我们将看到类似index请求的响应结果：

{

"\_index" : "website",

"\_id" : "1",

"\_type" : "blog",

"\_version" : 3

}

检索文档文档显示被更新的\_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> 我们新添加的字段已经被添加到\_source字段中。

### 使用脚本局部更新

### 使用Groovy脚本

这时候当API不能满足要求时，Elasticsearch允许你使用脚本实现自己的逻辑。脚本支持非常多的API，例如搜索、排序、聚合和文档更新。脚本可以通过请求的一部分、检索特殊的.scripts索引或者从磁盘加载方式执行。

默认的脚本语言是[Groovy](http://groovy.codehaus.org/)，一个快速且功能丰富的脚本语言，语法类似于Javascript。它在一个**沙盒(sandbox)**中运行，以防止恶意用户毁坏Elasticsearch或攻击服务器。

你可以在《脚本参考文档》中获得更多信息。

脚本能够使用update API改变\_source字段的内容，它在脚本内部以ctx.\_source表示。例如，我们可以使用脚本增加博客的views数量：

POST /website/blog/1/\_update

{

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

}

我们还可以使用脚本增加一个新标签到tags数组中。在这个例子中，我们定义了一个新标签作为参数而不是硬编码在脚本里。这允许Elasticsearch未来可以重复利用脚本，而不是在想要增加新标签时必须每次编译新脚本：

POST /website/blog/1/\_update

{

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

"params" : {

"new\_tag" : "search"

}

}

获取最后两个有效请求的文档：

{

"\_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标签已经被添加到tags数组。
* <2> views字段已经被增加。

通过设置ctx.op为delete我们可以根据内容删除文档：

POST /website/blog/1/\_update

{

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

"params" : {

"count": 1

}

}

### 更新可能不存在的文档

想象我们要在Elasticsearch中存储浏览量计数器。每当有用户访问页面，我们增加这个页面的浏览量。但如果这是个新页面，我们并不确定这个计数器存在与否。当我们试图更新一个不存在的文档，更新将失败。

在这种情况下，我们可以使用upsert参数定义文档来使其不存在时被创建。

POST /website/pageviews/1/\_update

{

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

"upsert": {

"views": 1

}

}

第一次执行这个请求，upsert值被索引为一个新文档，初始化views字段为1.接下来文档已经存在，所以script被更新代替，增加views数量。

### 更新和冲突

这一节的介绍中，我们介绍了如何在**检索(retrieve)**和**重建索引(reindex)**中保持更小的窗口，如何减少冲突性变更发生的概率，不过这些无法被完全避免，像一个其他进程在update进行重建索引时修改了文档这种情况依旧可能发生。

1

为了避免丢失数据，update API在**检索(retrieve)**阶段检索文档的当前\_version，然后在**重建索引(reindex)**阶段通过index请求提交。如果其他进程在**检索(retrieve)**和**重建索引(reindex)**阶段修改了文档，\_version将不能被匹配，然后更新失败。

对于多用户的局部更新，文档被修改了并不要紧。例如，两个进程都要增加页面浏览量，增加的顺序我们并不关心——如果冲突发生，我们唯一要做的仅仅是重新尝试更新既可。

这些可以通过retry\_on\_conflict参数设置重试次数来自动完成，这样update操作将会在发生错误前重试——这个值默认为0。

1

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

{

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

"upsert": {

"views": 0

}

}

* <1> 在错误发生前重试更新5次

这适用于像增加计数这种顺序无关的操作，但是还有一种顺序非常重要的情况。例如index API，使用**“保留最后更新(last-write-wins)”**的update API，但它依旧接受一个version参数以允许你使用**乐观并发控制(optimistic concurrency control)**来指定你要更细文档的版本。

3.10 Mget

## 检索多个文档

像Elasticsearch一样，检索多个文档依旧非常快。合并多个请求可以避免每个请求单独的网络开销。如果你需要从Elasticsearch中检索多个文档，相对于一个一个的检索，更快的方式是在一个请求中使用**multi-get**或者mget API。

mget API参数是一个docs数组，数组的每个节点定义一个文档的\_index、\_type、\_id元数据。如果你只想检索一个或几个确定的字段，也可以定义一个\_source参数：

2

POST /\_mget

{

"docs" : [

{

"\_index" : "website",

"\_type" : "blog",

"\_id" : 2

},

{

"\_index" : "website",

"\_type" : "pageviews",

"\_id" : 1,

"\_source": "views"

}

]

}

响应体也包含一个docs数组，每个文档还包含一个响应，它们按照请求定义的顺序排列。每个这样的响应与单独使用**get request**响应体相同：

{

"docs" : [

{

"\_index" : "website",

"\_id" : "2",

"\_type" : "blog",

"found" : true,

"\_source" : {

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

"title" : "My first external blog entry"

},

"\_version" : 10

},

{

"\_index" : "website",

"\_id" : "1",

"\_type" : "pageviews",

"found" : true,

"\_version" : 2,

"\_source" : {

"views" : 2

}

}

]

}

如果你想检索的文档在同一个\_index中（甚至在同一个\_type中），你就可以在URL中定义一个默认的/\_index或者/\_index/\_type。

你依旧可以在单独的请求中使用这些值：

POST /website/blog/\_mget

{

"docs" : [

{ "\_id" : 2 },

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

]

}

事实上，如果所有文档具有相同\_index和\_type，你可以通过简单的ids数组来代替完整的docs数组：

POST /website/blog/\_mget

{

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

}

注意到我们请求的第二个文档并不存在。我们定义了类型为blog，但是ID为1的文档类型为pageviews。这个不存在的文档会在响应体中被告知。

{

"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> 这个文档不存在

事实上第二个文档不存在并不影响第一个文档的检索。每个文档的检索和报告都是独立的。

注意：

尽管前面提到有一个文档没有被找到，但HTTP请求状态码还是200。事实上，就算所有文档都找不到，请求也还是返回200，原因是mget请求本身成功了。如果想知道每个文档是否都成功了，你需要检查found标志。

3.11 批量

## 更省时的批量操作

就像mget允许我们一次性检索多个文档一样，bulk API允许我们使用单一请求来实现多个文档的create、index、update或delete。这对索引类似于日志活动这样的数据流非常有用，它们可以以成百上千的数据为一个批次按序进行索引。

bulk请求体如下，它有一点不同寻常：

{ action: { metadata }}\n

{ request body }\n

{ action: { metadata }}\n

{ request body }\n

...

这种格式类似于用"\n"符号连接起来的一行一行的JSON文档**流(stream)**。两个重要的点需要注意：

* 每行必须以"\n"符号结尾，**包括最后一行**。这些都是作为每行有效的分离而做的标记。
* 每一行的数据不能包含未被转义的换行符，它们会干扰分析——这意味着JSON不能被美化打印。

提示:

在《批量格式》一章我们介绍了为什么bulk API使用这种格式。

**action/metadata**这一行定义了**文档行为(what action)**发生在**哪个文档(which document)**之上。

**行为(action)**必须是以下几种：

| **行为** | **解释** |
| --- | --- |
| create | 当文档不存在时创建之。详见《创建文档》 |
| index | 创建新文档或替换已有文档。见《索引文档》和《更新文档》 |
| update | 局部更新文档。见《局部更新》 |
| delete | 删除一个文档。见《删除文档》 |

在索引、创建、更新或删除时必须指定文档的\_index、\_type、\_id这些**元数据(metadata)**。

例如删除请求看起来像这样：

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

**请求体(request body)**由文档的\_source组成——文档所包含的一些字段以及其值。它被index和create操作所必须，这是有道理的：你必须提供文档用来索引。

这些还被update操作所必需，而且请求体的组成应该与update API（doc, upsert, script等等）一致。删除操作不需要**请求体(request body)**。

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

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

如果定义\_id，ID将会被自动创建：

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

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

为了将这些放在一起，bulk请求表单是这样的：

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**行为(action)**没有请求体，它紧接着另一个**行为(action)**
* <2> 记得最后一个换行符

Elasticsearch响应包含一个items数组，它罗列了每一个请求的结果，结果的顺序与我们请求的顺序相同：

{

"took": 4,

"errors": false, <1>

"items": [

{ "delete": {

"\_index": "website",

"\_type": "blog",

"\_id": "123",

"\_version": 2,

"status": 200,

"found": true

}},

{ "create": {

"\_index": "website",

"\_type": "blog",

"\_id": "123",

"\_version": 3,

"status": 201

}},

{ "create": {

"\_index": "website",

"\_type": "blog",

"\_id": "EiwfApScQiiy7TIKFxRCTw",

"\_version": 1,

"status": 201

}},

{ "update": {

"\_index": "website",

"\_type": "blog",

"\_id": "123",

"\_version": 4,

"status": 200

}}

]

}}

* <1> 所有子请求都成功完成。

每个子请求都被独立的执行，所以一个子请求的错误并不影响其它请求。如果任何一个请求失败，顶层的error标记将被设置为true，然后错误的细节将在相应的请求中被报告：

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" }

响应中我们将看到create文档123失败了，因为文档已经存在，但是后来的在123上执行的index请求成功了：

{

"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> 一个或多个请求失败。
* <2> 这个请求的HTTP状态码被报告为409 CONFLICT。
* <3> 错误消息说明了什么请求错误。
* <4> 第二个请求成功了，状态码是200 OK。

这些说明bulk请求不是原子操作——它们不能实现事务。每个请求操作时分开的，所以每个请求的成功与否不干扰其它操作。

## 不要重复

你可能在同一个index下的同一个type里批量索引日志数据。为每个文档指定相同的元数据是多余的。就像mget API，bulk请求也可以在URL中使用/\_index或/\_index/\_type:

POST /website/\_bulk

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

{ "event": "User logged in" }

你依旧可以覆盖元数据行的\_index和\_type，在没有覆盖时它会使用URL中的值作为默认值：

POST /website/log/\_bulk

{ "index": {}}

{ "event": "User logged in" }

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

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

## 多大才算太大？

整个批量请求需要被加载到接受我们请求节点的内存里，所以请求越大，给其它请求可用的内存就越小。有一个最佳的bulk请求大小。超过这个大小，性能不再提升而且可能降低。

最佳大小，当然并不是一个固定的数字。它完全取决于你的硬件、你文档的大小和复杂度以及索引和搜索的负载。幸运的是，这个**最佳点(sweetspot)**还是容易找到的：

试着批量索引标准的文档，随着大小的增长，当性能开始降低，说明你每个批次的大小太大了。开始的数量可以在1000~5000个文档之间，如果你的文档非常大，可以使用较小的批次。

通常着眼于你请求批次的物理大小是非常有用的。一千个1kB的文档和一千个1MB的文档大不相同。一个好的批次最好保持在5-15MB大小间。

## 结语

现在你知道如何把Elasticsearch当作一个分布式的文件存储了。你可以存储、更新、检索和删除它们，而且你知道如何安全的进行这一切。这确实非常非常有用，尽管我们还没有看到更多令人激动的特性，例如如何在文档内搜索。但让我们首先讨论下如何在分布式环境中安全的管理你的文档相关的内部流程。