如果没有配置节点名称，每次重启时会自动分配一个节点名

## 入门

### 文档

#### 面向文档

应用中的对象很少只是简单的键值列表，更多时候它拥有复杂的数据结构，比如包含日期、地理位置、另一个对象或者数组。

总有一天你会想到把这些对象存储到数据库中。将这些数据保存到由行和列组成的关系数据库中，就好像是把一个丰富，信息表现力强的对象拆散了放入一个非常大的表格中：你不得不拆散对象以适应表模式（通常一列表示一个字段），然后又不得不在查询的时候重建它们。

Elasticsearch是**面向文档(document oriented)**的，这意味着它可以存储整个对象或**文档(document)**。然而它不仅仅是存储，还会**索引(index)**每个文档的内容使之可以被搜索。在Elasticsearch中，你可以对文档（而非成行成列的数据）进行索引、搜索、排序、过滤。这种理解数据的方式与以往完全不同，这也是Elasticsearch能够执行复杂的全文搜索的原因之一。

#### JSON

ELasticsearch使用**Javascript对象符号(JavaScript Object Notation)**，也就是[**JSON**](http://en.wikipedia.org/wiki/Json)，作为文档序列化格式。JSON现在已经被大多语言所支持，而且已经成为NoSQL领域的标准格式。它简洁、简单且容易阅读。

以下使用JSON文档来表示一个用户对象：

{

"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"

}

尽管原始的user对象很复杂，但它的结构和对象的含义已经被完整的体现在JSON中了，在Elasticsearch中将对象转化为JSON并做索引要比在表结构中做相同的事情简单的多。

#### NOTE

尽管几乎所有的语言都有相应的模块用于将任意数据结构转换为JSON，但每种语言处理细节不同。具体请查看“serialization” or “marshalling”两个用于处理JSON的模块。Elasticsearch官方客户端会自动为你序列化和反序列化JSON。

### 索引

在elasticsearch中存储数据的行为叫做索引，不过在索引之前，我们需要明确数据应该存储在哪里。在elasticsearch中，文档归属于一种类型，而这些类型存在于索引中，可以类比传统关系型数据库来说明：

Relational DB -> Databases -> Tables -> Rows -> Columns

Elasticsearch -> Indices -> Types -> Documents -> Fields

Elasticsearch集群可以包含多个索引（indices）（数据库），每个索引可以包含多个类型（types）（表），每一个类型包含多个文档（行），每个文档包含多个字段（Field）（列）。

#### 索引含义的区分

索引（名词），一个索引（index）就像是传统关系数据库中的**数据库**，它是相关文档存储的地方，index的复数是**indices** 或**indexes**。

索引（动词） **「索引一个文档」**表示把一个文档存储到**索引（名词）**里，以便它可以被检索或者查询。这很像SQL中的INSERT关键字，差别是，如果文档已经存在，新的文档将覆盖旧的文档。

倒排索引 传统数据库为特定列增加一个索引，例如B-Tree索引来加速检索。Elasticsearch和Lucene使用一种叫做**倒排索引(inverted index)**的数据结构来达到相同目的。

默认情况下，文档中的所有字段都会被索引（拥有一个倒排索引），只有这样他们才是可被搜索的。

#### 创建

以创建员工目录为例，将进行如下操作：

* 为每个员工的**文档(document)**建立索引，每个文档包含了相应员工的所有信息。
* 每个文档的类型为employee。
* employee类型归属于索引megacorp。
* megacorp索引存储在Elasticsearch集群中。

可以通过一个命令完成以上操作：

PUT /megacorp/employee/1

{

"first\_name" : "John",

"last\_name" : "Smith",

"age" : 25,

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

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

}

Path：/megacorp/employee/1包含三部分信息：

| **名字** | **说明** |
| --- | --- |
| megacorp | 索引名 |
| employee | 类型名 |
| 1 | 这个员工的ID |

请求实体（JSON文档），包含了这个员工的所有信息。名字叫“John Smith”，25岁，喜欢攀岩。

它不需要我们做额外的管理操作，例如创建索引或者定义每个字段的数据类型，能够直接索引。

加入更多的员工信息：

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" ]

}

### 搜索

#### 检索文档

如何检索单个员工的信息，对于elasticsearch来说非常简单。我们只要执行HTTP GET请求并指出文档的“地址”——索引、类型和ID既可。根据这三部分信息，我们就可以返回原始JSON文档：

GET /megacorp/employee/1

响应的内容中包含一些文档的元信息，John Smith的原始JSON文档包含在\_source字段中：

{

"\_index" : "megacorp",

"\_type" : "employee",

"\_id" : "1",

"\_version" : 1,

"found" : true,

"\_source" : {

"first\_name" : "John",

"last\_name" : "Smith",

"age" : 25,

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

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

}

}

**我们通过HTTP方法GET来检索文档，同样的，我们可以使用DELETE方法删除文档，使用HEAD方法检查某文档是否存在。如果想更新已存在的文档，我们只需再PUT一次。**

#### 简单搜索

最简单的搜索全部员工的请求：

GET /megacorp/employee/\_search

可以看到我们依然使用megacorp索引和employee类型，但是我们在结尾使用关键字\_search来取代原来的文档ID。响应内容的hits数组中包含了我们所有的三个文档。默认情况下搜索会返回前10个结果。

{

"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：响应内容不仅会告诉我们哪些文档被匹配到，而且这些文档内容完整的被包含在其中。**

再搜索姓氏包含**“Smith”**的员工。要做到这一点，我们将在命令行中使用轻量级的搜索方法。这种方法常被称作**查询字符串(query string)**搜索，因为我们像传递URL参数一样去传递查询语句：

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

我们在请求中依旧使用\_search关键字，然后将查询语句传递给参数q=。这样就可以得到所有姓氏为Smith的结果：

{

...

"hits": {

"total": 2,

"max\_score": 0.30685282,

"hits": [

{

...

"\_source": {

"first\_name": "John",

"last\_name": "Smith",

"age": 25,

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

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

}

},

{

...

"\_source": {

"first\_name": "Jane",

"last\_name": "Smith",

"age": 32,

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

"interests": [ "music" ]

}

}

]

}

}

#### 使用DSL语句查询

查询字符串搜索便于通过命令行完成**特定(ad hoc)**的搜索，但是它也有局限性（参阅简单搜索章节）。Elasticsearch提供丰富且灵活的查询语言叫做**DSL查询(Query DSL)**,它允许你构建更加复杂、强大的查询。

**DSL(Domain Specific Language特定领域语言)**以JSON请求体的形式出现。我们可以这样表示之前关于“Smith”的查询:

POST /megacorp/employee/\_search

{

"query" : {

"match" : {

"last\_name" : "Smith"

}

}

}

这会返回与之前查询相同的结果。你可以看到有些东西改变了，我们不再使用**查询字符串(query string)**做为参数，而是使用请求体代替。这个请求体使用JSON表示，其中使用了match语句。

#### 更复杂的搜索

把搜索变的复杂一点，依旧搜索姓氏为“Smith”的员工，但是我们只想得到年龄大于30岁的员工。我们的语句将添加**过滤器(filter)**,它使得我们高效率的执行一个结构化搜索：

POST /megacorp/employee/\_search

{

"query" : {

"filtered" : {

"filter" : {

"range" : {

"age" : { "gt" : 30 } <1>

}

},

"query" : {

"match" : {

"last\_name" : "smith" <2>

}

}

}

}

}

 <1> 这部分查询属于**区间过滤器(range filter)**,它用于查找所有年龄大于30岁的数据——gt为"greater than"的缩写。

 <2> 这部分查询与之前的match**语句(query)**一致。

我们添加了一个**过滤器(filter)**用于执行区间搜索，然后重复利用了之前的match语句。现在我们的搜索结果只显示了一个32岁且名字是“Jane Smith”的员工：

{

...

"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

到目前为止搜索都很简单：搜索特定的名字，通过年龄筛选。让我们尝试一种更高级的搜索，全文搜索——一种传统数据库很难实现的功能。

我们将会搜索所有喜欢**“rock climbing”**的员工：

POST /megacorp/employee/\_search

{

"query" : {

"match" : {

"about" : "rock climbing"

}

}

}

可以看到我们使用了之前的match查询，从about字段中搜索**"rock climbing"**，我们得到了两个匹配文档：

{

...

"hits": {

"total": 2,

"max\_score": 0.16273327,

"hits": [

{

...

"\_score": 0.16273327, <1>

"\_source": {

"first\_name": "John",

"last\_name": "Smith",

"age": 25,

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

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

}

},

{

...

"\_score": 0.016878016, <2>

"\_source": {

"first\_name": "Jane",

"last\_name": "Smith",

"age": 32,

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

"interests": [ "music" ]

}

}

]

}

}

* <1><2> 结果相关性评分。

默认情况下，Elasticsearch根据结果相关性评分来对结果集进行排序，所谓的「结果相关性评分」就是文档与查询条件的匹配程度。很显然，排名第一的John Smith的about字段明确的写到**“rock climbing”**。

但是为什么Jane Smith也会出现在结果里呢？原因是**“rock”**在她的abuot字段中被提及了。因为只有**“rock”**被提及而**“climbing”**没有，所以她的\_score要低于John。

这个例子很好的解释了Elasticsearch如何在各种文本字段中进行全文搜索，并且返回相关性最大的结果集。**相关性(relevance)**的概念在Elasticsearch中非常重要，而这个概念在传统关系型数据库中是不可想象的，因为传统数据库对记录的查询只有匹配或者不匹配。

#### 短语搜索

目前我们可以在字段中搜索单独的一个词，这挺好的，但是有时候你想要确切的匹配若干个单词或者**短语(phrases)**。例如我们想要查询同时包含"rock"和"climbing"（并且是相邻的）的员工记录。

要做到这个，我们只要将match查询变更为match\_phrase查询即可:

POST /megacorp/employee/\_search

{

"query" : {

"match\_phrase" : {

"about" : "rock climbing"

}

}

}

该查询返回John Smith的文档：

{

...

"hits": {

"total": 1,

"max\_score": 0.23013961,

"hits": [

{

...

"\_score": 0.23013961,

"\_source": {

"first\_name": "John",

"last\_name": "Smith",

"age": 25,

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

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

}

}

]

}

}

#### 高亮

很多应用喜欢从每个搜索结果中**高亮(highlight)**匹配到的关键字，这样用户可以知道为什么这些文档和查询相匹配。在Elasticsearch中高亮片段是非常容易的。

让我们在之前的语句上增加highlight参数：

POST /megacorp/employee/\_search

{

"query" : {

"match\_phrase" : {

"about" : "rock climbing"

}

},

"highlight": {

"fields" : {

"about" : {}

}

}

}

当我们运行这个语句时，会命中与之前相同的结果，但是在返回结果中会有一个新的部分叫做highlight，这里包含了来自about字段中的文本，并且用<em></em>来标识匹配到的单词。

{

...

"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> 原有文本中高亮的片段

### 聚合

#### 分析

最后还有一个需求要完成：允许管理者在职员目录中进行一些分析。Elasticsearch有一个功能叫做**聚合(aggregations)**，它允许你在数据上生成复杂的分析统计。它很像SQL中的GROUP BY但是功能更强大。

举个例子，让我们找到所有职员中最大的共同点（兴趣爱好）是什么：

POST /megacorp/employee/\_search

{

"aggs": {

"all\_interests": {

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

}

}

}

暂时忽略语法只看查询结果：

{

...

"hits": { ... },

"aggregations": {

"all\_interests": {

"buckets": [

{

"key": "music",

"doc\_count": 2

},

{

"key": "forestry",

"doc\_count": 1

},

{

"key": "sports",

"doc\_count": 1

}

]

}

}

}

我们可以看到两个职员对音乐有兴趣，一个喜欢林学，一个喜欢运动。这些数据并没有被预先计算好，它们是实时的从匹配查询语句的文档中动态计算生成的。如果我们想知道所有姓"Smith"的人最大的共同点（兴趣爱好），我们只需要增加合适的语句既可：

POST /megacorp/employee/\_search

{

"query": {

"match": {

"last\_name": "smith"

}

},

"aggs": {

"all\_interests": {

"terms": {

"field": "interests"

}

}

}

}

all\_interests聚合已经变成只包含和查询语句相匹配的文档了：

...

"all\_interests": {

"buckets": [

{

"key": "music",

"doc\_count": 2

},

{

"key": "sports",

"doc\_count": 1

}

]

}

聚合也允许分级汇总。例如，让我们统计每种兴趣下职员的平均年龄：

POST /megacorp/employee/\_search

{

"aggs" : {

"all\_interests" : {

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

"aggs" : {

"avg\_age" : {

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

}

}

}

}

}

返回的聚合结果有些复杂，但任然很容易理解：

...

"all\_interests": {

"buckets": [

{

"key": "music",

"doc\_count": 2,

"avg\_age": {

"value": 28.5

}

},

{

"key": "forestry",

"doc\_count": 1,

"avg\_age": {

"value": 35

}

},

{

"key": "sports",

"doc\_count": 1,

"avg\_age": {

"value": 25

}

}

]

}

该聚合结果比之前的聚合结果要更加丰富。我们依然得到了兴趣以及数量（指具有该兴趣的员工人数）的列表，但是现在每个兴趣额外拥有avg\_age字段来显示具有该兴趣员工的平均年龄。

即使还不理解语法，但也可以大概感觉到通过这个特性可以完成相当复杂的**聚合**工作，可以处理任何类型的数据。

### 分布式的特性

Elasticsearch可以扩展到上百（甚至上千）的服务器来处理PB级的数据，Elasticsearch为分布式而生，而且它的设计隐藏了分布式本身的复杂性。

Elasticsearch在分布式概念上做了很大程度上的透明化，在教程中你不需要知道任何关于分布式系统、分片、集群发现或者其他大量的分布式概念。所有的教程你既可以运行在你的笔记本上，也可以运行在拥有100个节点的集群上，其工作方式是一样的。

Elasticsearch致力于隐藏分布式系统的复杂性。以下这些操作都是在底层自动完成的：

* 将你的文档分区到不同的容器或者**分片(shards)**中，它们可以存在于一个或多个节点中。
* 将分片均匀的分配到各个节点，对索引和搜索做负载均衡。
* 冗余每一个分片，防止硬件故障造成的数据丢失。
* 将集群中任意一个节点上的请求路由到相应数据所在的节点。
* 无论是增加节点，还是移除节点，分片都可以做到无缝的扩展和迁移

## 分布式集群

Elasticsearch用于构建高可用和可扩展的系统。扩展的方式可以是购买更好的服务器(**纵向扩展(vertical scale or scaling up)**)或者购买更多的服务器（**横向扩展(horizontal scale or scaling out)**）。

Elasticsearch虽然能从更强大的硬件中获得更好的性能，但是纵向扩展有它的局限性。真正的扩展应该是横向的，它通过增加节点来均摊负载和增加可靠性。

对于大多数数据库而言，横向扩展意味着你的程序将做非常大的改动才能利用这些新添加的设备。对比来说，Elasticsearch天生就是分布式的：它知道如何管理节点来提供高扩展和高可用。这意味着你的程序不需要关心这些。

### 空集群

如果我们启动一个单独的节点，它还没有数据和索引，这个集群看起来就像图1。



图1：只有一个空节点的集群

一个**节点(node)**就是一个Elasticsearch实例，而一个**集群(cluster)**由一个或多个节点组成，它们具有相同的cluster.name，它们协同工作，分享数据和负载。当加入新的节点或者删除一个节点时，集群就会感知到并平衡数据。

集群中一个节点会被选举为**主节点(master)**,它将临时管理集群级别的一些变更，例如新建或删除索引、增加或移除节点等。主节点不参与文档级别的变更或搜索，这意味着在流量增长的时候，该主节点不会成为集群的瓶颈。任何节点都可以成为主节点。我们例子中的集群只有一个节点，所以它会充当主节点的角色。

做为用户，我们能够与集群中的任何节点通信，包括主节点。每一个节点都知道文档存在于哪个节点上，它们可以转发请求到相应的节点上。我们访问的节点负责收集各节点返回的数据，最后一起返回给客户端。这一切都由Elasticsearch处理。

### 集群健康

在Elasticsearch集群中可以监控统计很多信息，但是只有一个是最重要的：**集群健康(cluster health)**。集群健康有三种状态：green、yellow或red。

GET /\_cluster/health

在一个没有索引的空集群中运行如上查询，将返回这些信息：

{

"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 是我们最感兴趣的字段

status字段提供一个综合的指标来表示集群的的服务状况。三种颜色各自的含义：

| **颜色** | **意义** |
| --- | --- |
| green | 所有主要分片和复制分片都可用 |
| yellow | 所有主要分片可用，但不是所有复制分片都可用 |
| red | 不是所有的主要分片都可用 |

### 添加索引

为了将数据添加到Elasticsearch，我们需要**索引(index)**——一个存储关联数据的地方。实际上，索引只是一个用来指向一个或多个**分片(shards)**的**“逻辑命名空间(logical namespace)”**.

一个**分片(shard)**是一个最小级别**“工作单元(worker unit)”**,它只是保存了索引中所有数据的一部分。在接下来的《深入分片》一章，我们将详细说明分片的工作原理，但是现在我们只要知道分片就是一个Lucene实例，并且它本身就是一个完整的搜索引擎。我们的文档存储在分片中，并且在分片中被索引，但是我们的应用程序不会直接与它们通信，取而代之的是，直接与索引通信。

分片是Elasticsearch在集群中分发数据的关键。把分片想象成数据的容器。文档存储在分片中，然后分片分配到你集群中的节点上。当你的集群扩容或缩小，Elasticsearch将会自动在你的节点间迁移分片，以使集群保持平衡。

分片可以是**主分片(primary shard)**或者是**复制分片(replica shard)**。你索引中的每个文档属于一个单独的主分片，所以主分片的数量决定了索引最多能存储多少数据。

理论上主分片能存储的数据大小是没有限制的，限制取决于你实际的使用情况。分片的最大容量完全取决于你的使用状况：硬件存储的大小、文档的大小和复杂度、如何索引和查询你的文档，以及你期望的响应时间。

复制分片只是主分片的一个副本，它可以防止硬件故障导致的数据丢失，同时可以提供读请求，比如搜索或者从别的shard取回文档。

当索引创建完成的时候，主分片的数量就固定了，但是复制分片的数量可以随时调整。

让我们在集群中唯一一个空节点上创建一个叫做blogs的索引。默认情况下，一个索引被分配5个主分片，但是为了演示的目的，我们只分配3个主分片和一个复制分片（每个主分片都有一个复制分片）：

PUT /blogs

{

"settings" : {

"number\_of\_shards" : 3,

"number\_of\_replicas" : 1

}

}

附带索引的单一节点集群：



{

"cluster\_name": "elasticsearch",

"status": "yellow", <1>

"timed\_out": false,

"number\_of\_nodes": 1,

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

"active\_primary\_shards": 3,

"active\_shards": 3,

"relocating\_shards": 0,

"initializing\_shards": 0,

"unassigned\_shards": 3 <2>

}

 <1> 集群的状态现在是 yellow

 <2> 我们的三个复制分片还没有被分配到节点上

集群的健康状态yellow表示所有的**主分片(primary shards)**启动并且正常运行了——集群已经可以正常处理任何请求——但是**复制分片(replica shards)**还没有全部可用。事实上所有的三个复制分片现在都是unassigned状态——它们还未被分配给节点。在同一个节点上保存相同的数据副本是没有必要的，如果这个节点故障了，那所有的数据副本也会丢失。

现在我们的集群已经功能完备，但是依旧存在因硬件故障而导致数据丢失的风险

### 故障转移

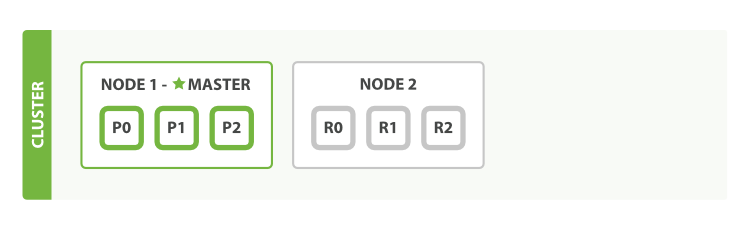
在单一节点上运行意味着有单点故障的风险——没有数据备份。幸运的是，要防止单点故障，我们唯一需要做的就是启动另一个节点。

为了测试在增加第二个节点后发生了什么，你可以使用与第一个节点相同的方式启动第二个节点（《运行Elasticsearch》一章），而且命令行在同一个目录——一个节点可以启动多个Elasticsearch实例。

只要第二个节点与第一个节点有相同的cluster.name（请看./config/elasticsearch.yml文件），它就能自动发现并加入第一个节点所在的集群。如果没有，检查日志找出哪里出了问题。这可能是网络广播被禁用，或者防火墙阻止了节点通信。

如果我们启动了第二个节点，这个集群看起来就像下图。（我没加成功）

双节点集群——所有的主分片和复制分片都已分配:



第二个节点已经加入集群，三个**复制分片(replica shards)**也已经被分配了——分别对应三个主分片，这意味着在丢失任意一个节点的情况下依旧可以保证数据的完整性。

文档的索引将首先被存储在主分片中，然后并发复制到对应的复制节点上。这可以确保我们的数据在主节点和复制节点上都可以被检索。

cluster-health现在的状态是green，这意味着所有的6个分片（三个主分片和三个复制分片）都已可用：

{

"cluster\_name": "elasticsearch",

"status": "green", <1>

"timed\_out": false,

"number\_of\_nodes": 2,

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

"active\_primary\_shards": 3,

"active\_shards": 6,

"relocating\_shards": 0,

"initializing\_shards": 0,

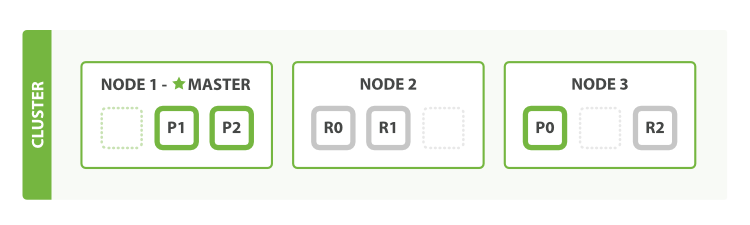
"unassigned\_shards": 0

}

<1> 集群的状态是green

### 横向扩展

随着应用需求的增长，当启动第三个节点时，集群会重新组织自己，如下图所示：

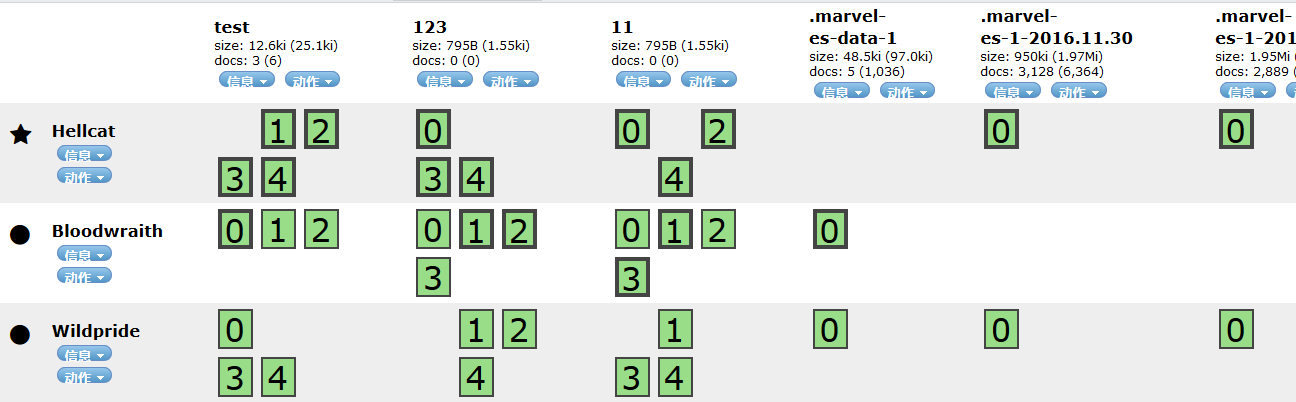


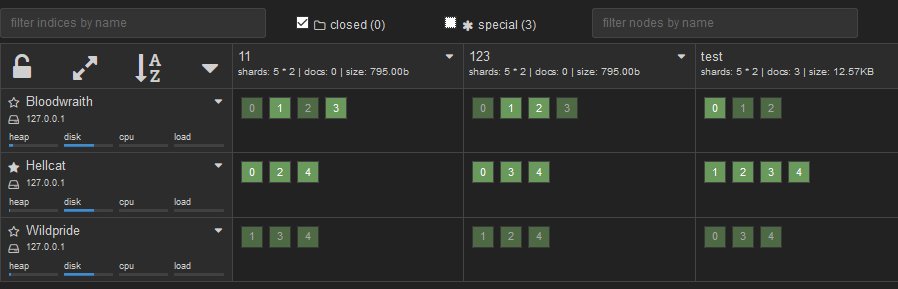
包含3个节点的集群——分片已经被重新分配以平衡负载

Node3包含了分别来自Node 1和Node 2的一个分片，这样每个节点就有两个分片，和之前相比少了一个，这意味着每个节点上的分片将获得更多的硬件资源（CPU、RAM、I/O）。

分片本身就是一个完整的搜索引擎，它可以使用单一节点的所有资源。我们拥有6个分片（3个主分片和三个复制分片），最多可以扩展到6个节点，每个节点上有一个分片，每个分片可以100%使用这个节点的资源。

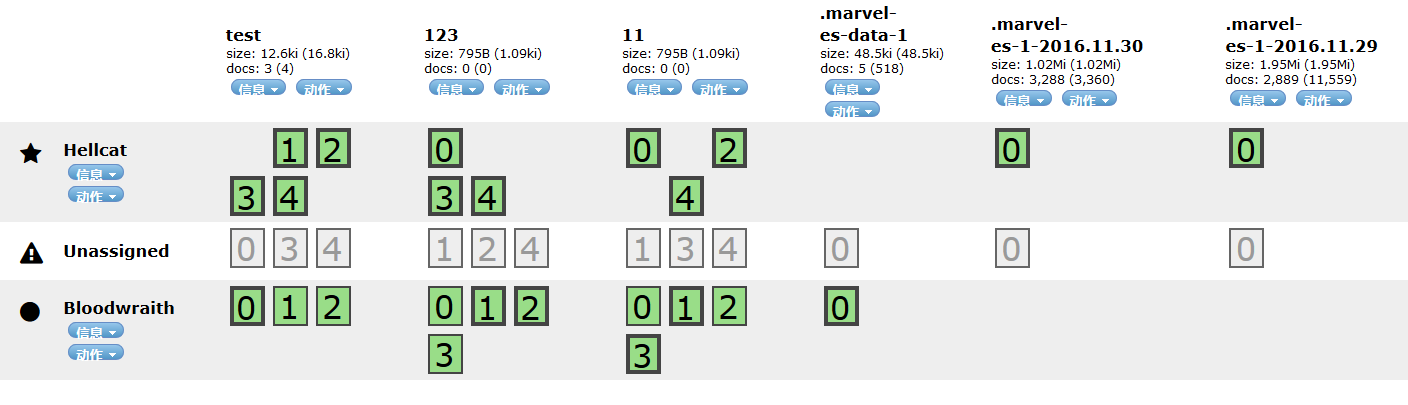
**具体测试的实例如下：**



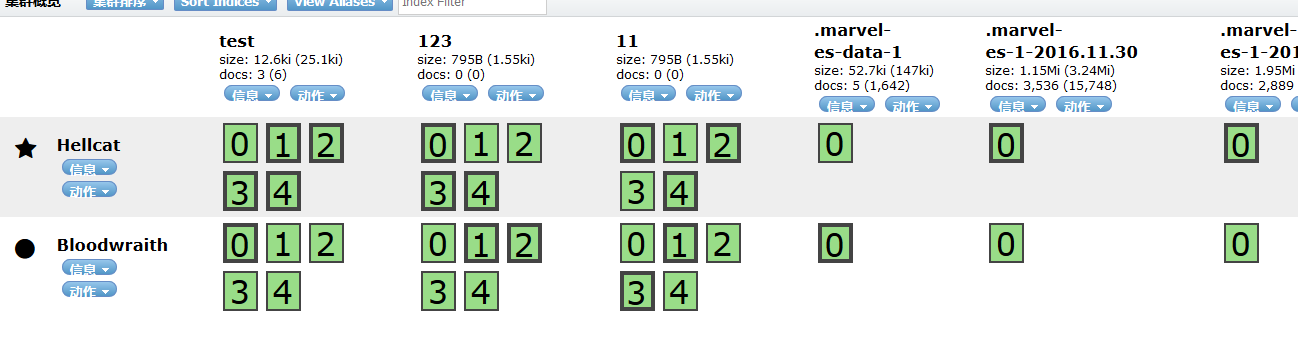


包含3个节点的内容，的确是被重新分配了内容

关闭掉一个节点初始为：



刷新后分片重新分配：



Kopf会自动刷新，head需要手动刷新才会呈现新的集群状态。

### 更多扩展

如果我们要扩展到6个以上的节点，要怎么做？

主分片的数量在创建索引时已经确定。实际上，这个数量定义了能存储到索引里数据的最大数量（实际的数量取决于你的数据、硬件和应用场景）。然而，主分片或者复制分片都可以处理读请求——搜索或文档检索，所以数据的冗余越多，我们能处理的搜索吞吐量就越大。

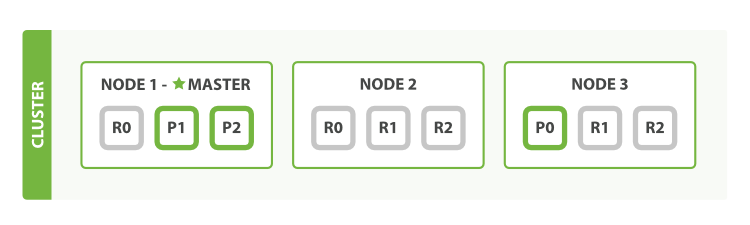
复制分片的数量可以在运行中的集群中动态地变更，这允许我们可以根据需求扩大或者缩小规模。让我们把复制分片的数量从原来的1增加到2：

PUT /blogs/\_settings 这里的blogs为索引名

{

"number\_of\_replicas" : 2

}

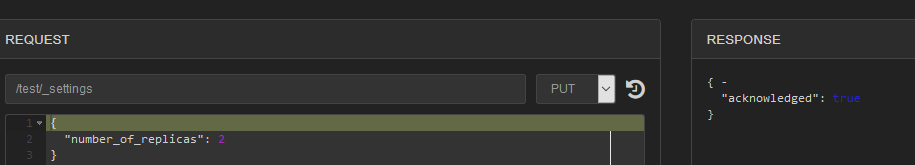


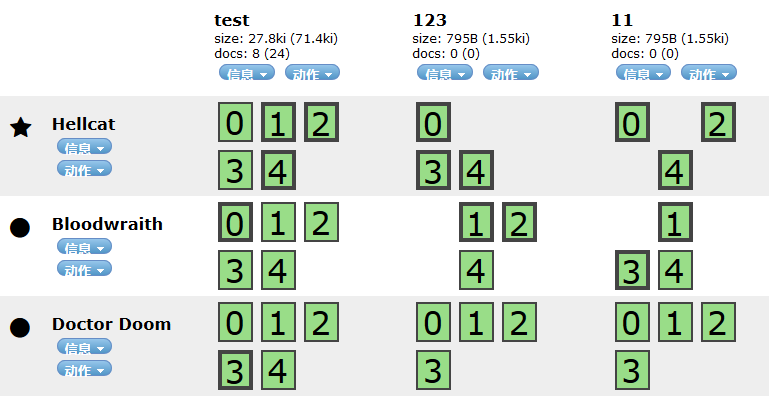
从图中可以看出，blogs索引现在有9个分片：3个主分片和6个复制分片。这意味着我们能够扩展到9个节点，再次变成每个节点一个分片。这样使我们的搜索性能相比原始的三节点集群增加**三倍**。

当然，在同样数量的节点上增加更多的复制分片并不能提高性能，因为这样做的话平均每个分片的所占有的硬件资源就减少了（译者注：大部分请求都聚集到了分片少的节点，导致一个节点吞吐量太大，反而降低性能），你需要增加硬件来提高吞吐量。

不过这些额外的复制节点使我们有更多的冗余：通过以上对节点的设置，我们能够承受两个节点故障而不丢失数据。

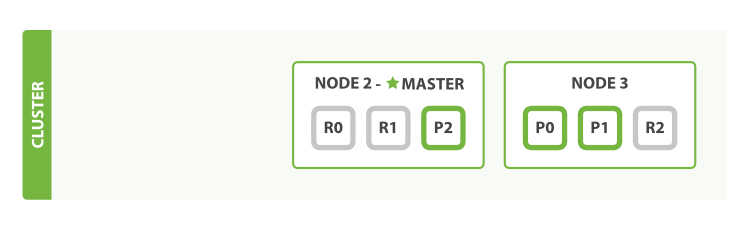
我们以test索引为例





### 应对故障

杀掉第一个节点的进程（以下简称杀掉节点），我们的集群看起来就像这样：



我们杀掉的节点是一个主节点。一个集群必须要有一个主节点才能使其功能正常，所以集群做的第一件事就是各节点选举了一个新的主节点：Node 2。

主分片1和2在我们杀掉Node 1时已经丢失，我们的索引在丢失主分片时不能正常工作。如果此时我们检查集群健康，我们将看到状态red：不是所有主分片都可用！

幸运的是丢失的两个主分片的完整拷贝存在于其他节点上，所以新主节点做的第一件事是把这些在Node 2和Node 3上的复制分片升级为主分片，这时集群健康回到yellow状态。这个提升是瞬间完成的，就好像按了一下开关。

为什么集群健康状态是yellow而不是green？我们有三个主分片，但是我们指定了每个主分片对应两个复制分片，当前却只有一个复制分片被分配，这就是集群状态无法达到green的原因，不过不用太担心这个：当我们杀掉Node 2，我们的程序依然可以在没有丢失数据的情况下继续运行，因为Node 3还有每个分片的拷贝。

如果我们重启Node 1，集群将能够重新分配丢失的复制分片，集群状况与上一节的 **图5：增加number\_of\_replicas到2** 类似。如果Node 1依旧有旧分片的拷贝，它将会尝试再利用它们，它只会从主分片上复制在故障期间有数据变更的那一部分。

**个人测试因插件问题未能看到主节点被杀之后的情景**

## 数据

但是如何能以对象的形式存储对象呢？相对于围绕表格去为我们的程序去建模，我们可以专注于**使用**数据，把对象本来的灵活性找回来。

**对象(object)**是一种语言相关，记录在内存中的的数据结构。为了在网络间发送，或者存储它，我们需要一些标准的格式来表示它。[JSON (JavaScript Object Notation)](http://en.wikipedia.org/wiki/Json)是一种可读的以文本来表示对象的方式。它已经成为NoSQL世界中数据交换的一种事实标准。当对象被序列化为JSON，它就成为**JSON文档(JSON document)**了。

Elasticsearch是一个分布式的**文档(document)**存储引擎。它可以实时存储并检索复杂数据结构——序列化的JSON文档。换言说，一旦文档被存储在Elasticsearch中，它就可以在集群的任一节点上被检索。

当然，我们不仅需要存储数据，还要快速的**批量**查询。虽然已经有很多NoSQL的解决方案允许我们以文档的形式存储对象，但它们依旧需要考虑如何查询这些数据，以及哪些字段需要被索引以便检索时更加快速。

在Elasticsearch中，**每一个字段的数据**都是**默认被索引**的。也就是说，每个字段专门有一个反向索引用于快速检索。而且，与其它数据库不同，它可以在**同一个查询中**利用所有的这些反向索引，以惊人的速度返回结果。

### 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。

{

"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，还包含了元数据——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。

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。

### 索引

文档通过index API被索引——使数据可以被存储和搜索。首先需要确定文档所在，文档通过其\_index、\_type、\_id唯一确定，可以自己提供一个\_id，或者也可使用index API为我们生成一个。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)。

### 获取

即检索

#### 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"

}

### 存在（功能待确认）

**检索文档是否存在：**

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。

### 更新

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。

{

"\_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。

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

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。

### 创建

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？

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/

{ ... }

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：

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：

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

}

### 删除

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。

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。

### 版本控制（？？？）

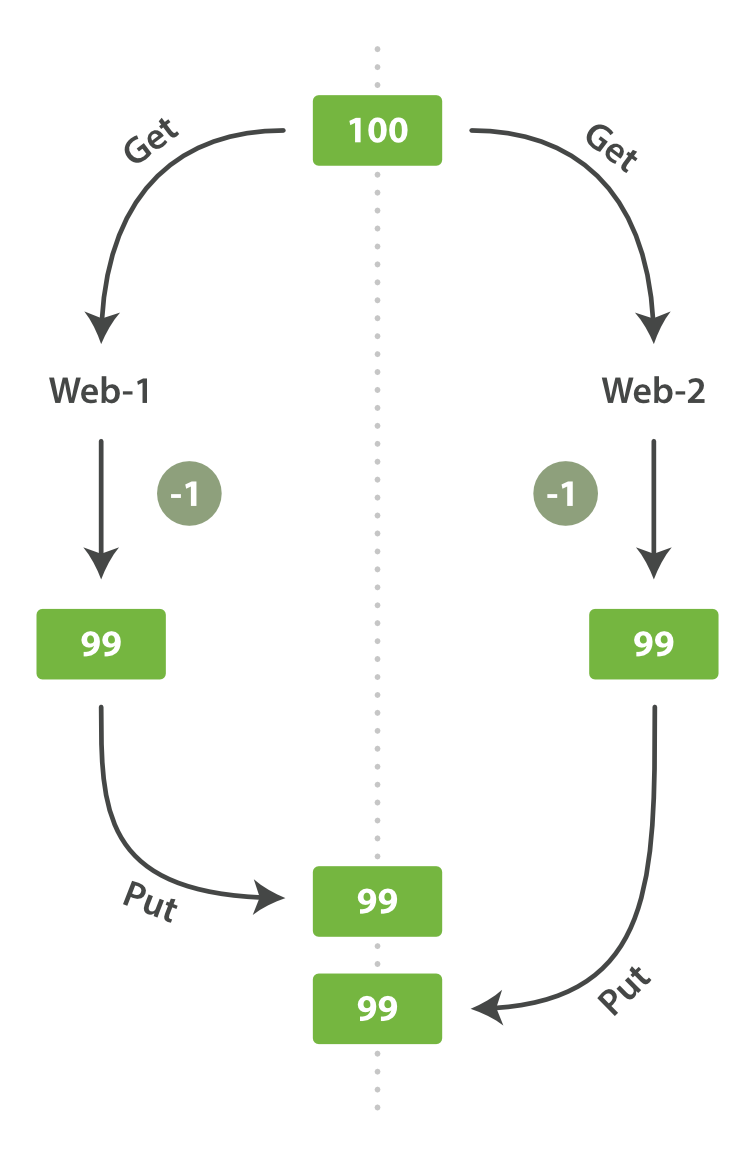
#### 处理冲突

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。

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。

##### 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。

#### 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。

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。

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：

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：

{

"\_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

}

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。

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：

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。

### 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。

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做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。

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。

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。

### 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：

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。

### 批量(适用于日志)

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。

#### Notice

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" }

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。

## 分布式增删改查

介绍数据是如何在分布式系统中存储的

### 路由

#### 路由文档到分片

当你索引一个文档，它被存储在单独一个主分片上。Elasticsearch是如何知道文档属于哪个分片的呢？当你创建一个新文档，它是如何知道是应该存储在分片1还是分片2上的呢？

进程不能是随机的，因为我们将来要检索文档。事实上，它根据一个简单的算法决定：

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

routing值是一个任意字符串，它默认是\_id但也可以自定义。这个routing字符串通过哈希函数生成一个数字，然后除以主切片的数量得到一个**余数(remainder)**，余数的范围永远是0到number\_of\_primary\_shards - 1，这个数字就是特定文档所在的分片。

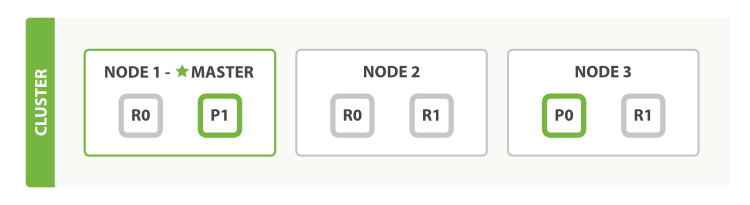
这也解释了为什么主分片的数量只能在创建索引时定义且不能修改：如果主分片的数量在未来改变了，所有先前的路由值就失效了，文档也就永远找不到了。

有时用户认为固定数量的主分片会让之后的扩展变得很困难。现实中，有些技术会在你需要的时候让扩展变得容易。我们将在《扩展》章节讨论。

所有的文档API（get、index、delete、bulk、update、mget）都接收一个routing参数，它用来自定义文档到分片的映射。自定义路由值可以确保所有相关文档——例如属于同一个人的文档——被保存在同一分片上。

### 分片交互

为了阐述意图，我们假设有三个节点的集群。它包含一个叫做bblogs的索引并拥有两个主分片。每个主分片有两个复制分片。相同的分片不会放在同一个节点上，所以我们的集群是这样的：



我们能够发送请求给集群中任意一个节点。每个节点都有能力处理任意请求。每个节点都知道任意文档所在的节点，所以也可以将请求转发到需要的节点。下面的例子中，我们将发送所有请求给Node 1，这个节点我们将会称之为**请求节点(requesting node)**

**提示：**

当我们发送请求，最好的做法是循环通过所有节点请求，这样可以平衡负载。

### 新建、索引和删除

新建、索引和删除请求都是**写(write)**操作，它们必须在主分片上成功完成才能复制到相关的复制分片上。



下面我们罗列在主分片和复制分片上成功新建、索引或删除一个文档必要的顺序步骤：

1. 客户端给Node 1发送新建、索引或删除请求。
2. 节点使用文档的\_id确定文档属于分片0。它转发请求到Node 3，分片0位于这个节点上。
3. Node 3在主分片上执行请求，如果成功，它转发请求到相应的位于Node 1和Node 2的复制节点上。当所有的复制节点报告成功，Node 3报告成功到请求的节点，请求的节点再报告给客户端。

客户端接收到成功响应的时候，文档的修改已经被应用于主分片和所有的复制分片。你的修改生效了。

有很多可选的请求参数允许你更改这一过程。你可能想牺牲一些安全来提高性能。这一选项很少使用因为Elasticsearch已经足够快，不过为了内容的完整我们将做一些阐述。

#### Replication

复制默认的值是sync。这将导致主分片得到复制分片的成功响应后才返回。

如果你设置replication为async，请求在主分片上被执行后就会返回给客户端。它依旧会转发请求给复制节点，但你将不知道复制节点成功与否。

上面的这个选项不建议使用。默认的sync复制允许Elasticsearch强制反馈传输。async复制可能会因为在不等待其它分片就绪的情况下发送过多的请求而使Elasticsearch过载。

#### Consistency

默认主分片在尝试写入时需要**规定数量(quorum)**或过半的分片（可以是主节点或复制节点）可用。这是防止数据被写入到错的网络分区。规定的数量计算公式如下：

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

consistency允许的值为one（只有一个主分片），all（所有主分片和复制分片）或者默认的quorum或过半分片。

注意number\_of\_replicas是在索引中的的设置，用来定义复制分片的数量，而不是现在活动的复制节点的数量。如果你定义了索引有3个复制节点，那规定数量是：

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

但如果你只有2个节点，那你的活动分片不够规定数量，也就不能索引或删除任何文档。

#### Timeout

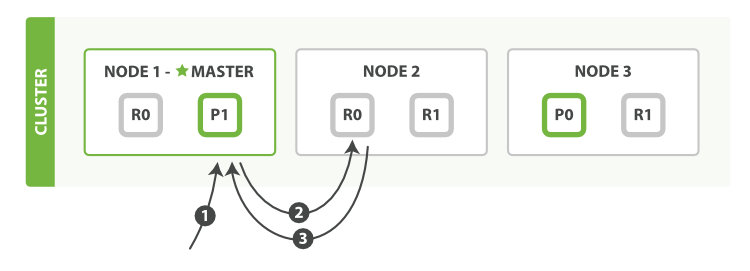
当分片副本不足时会怎样？Elasticsearch会等待更多的分片出现。默认等待一分钟。如果需要，你可以设置timeout参数让它终止的更早：100表示100毫秒，30s表示30秒。

注意：

新索引默认有1个复制分片，这意味着为了满足quorum的要求**需要**两个活动的分片。当然，这个默认设置将阻止我们在单一节点集群中进行操作。为了避开这个问题，规定数量只有在number\_of\_replicas大于一时才生效。

### 检索

文档能够从主分片或任意一个复制分片被检索。



下面我们罗列在主分片或复制分片上检索一个文档必要的顺序步骤：

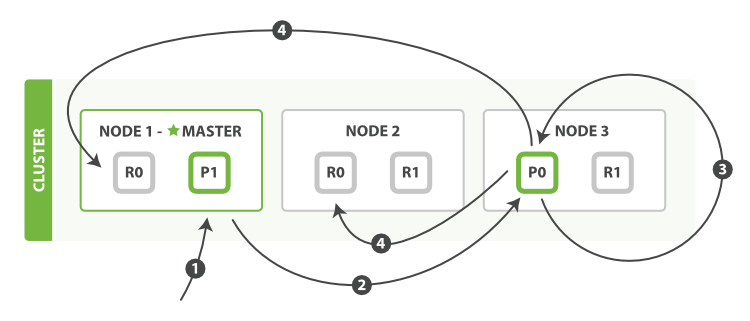
1. 客户端给Node 1发送get请求。
2. 节点使用文档的\_id确定文档属于分片0。分片0对应的复制分片在三个节点上都有。此时，它转发请求到Node 2。
3. Node 2返回文档(document)给Node 1然后返回给客户端。

对于读请求，为了平衡负载，请求节点会为每个请求选择不同的分片——它会循环所有分片副本。

**可能的情况是，一个被索引的文档已经存在于主分片上却还没来得及同步到复制分片上。这时复制分片会报告文档未找到，主分片会成功返回文档。一旦索引请求成功返回给用户，文档则在主分片和复制分片都是可用的**

### 局部更新

Update API结合了之前提到的读和写的模式。



下面我们罗列执行局部更新必要的顺序步骤：

1. 客户端给Node 1发送更新请求。
2. 它转发请求到主分片所在节点Node 3。
3. Node 3从主分片检索出文档，修改\_source字段的JSON，然后在主分片上重建索引。如果有其他进程修改了文档，它以retry\_on\_conflict设置的次数重复步骤3，都未成功则放弃。
4. 如果Node 3成功更新文档，它同时转发文档的新版本到Node 1和Node 2上的复制节点以重建索引。当所有复制节点报告成功，Node 3返回成功给请求节点，然后返回给客户端。

update API还接受《新建、索引和删除》章节提到的routing、replication、consistency和timout参数。

**基于文档的复制**

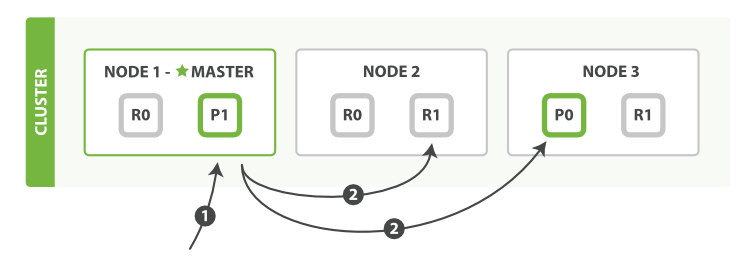
当主分片转发更改给复制分片时，并不是转发更新请求，而是转发整个文档的新版本。记住这些修改转发到复制节点是异步的，它们并不能保证到达的顺序与发送相同。如果Elasticsearch转发的仅仅是修改请求，修改的顺序可能是错误的，那得到的就是个损坏的文档。

### 批量请求（日志等）

#### 多文档模式

mget和bulk API与单独的文档类似。差别是请求节点知道每个文档所在的分片。它把多文档请求拆成**每个分片**的对文档请求，然后转发每个参与的节点。

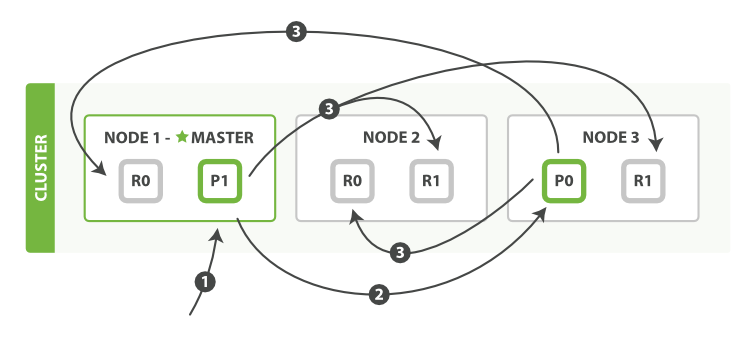
一旦接收到每个节点的应答，然后整理这些响应组合为一个单独的响应，最后返回给客户端。



下面我们将罗列通过一个mget请求检索多个文档的顺序步骤：

1. 客户端向Node 1发送mget请求。
2. Node 1为每个分片构建一个多条数据检索请求，然后转发到这些请求所需的主分片或复制分片上。当所有回复被接收，Node 1构建响应并返回给客户端。

routing 参数可以被docs中的每个文档设置。



下面我们将罗列使用一个bulk执行多个create、index、delete和update请求的顺序步骤：

1. 客户端向Node 1发送bulk请求。
2. Node 1为每个分片构建批量请求，然后转发到这些请求所需的主分片上。
3. 主分片一个接一个的按序执行操作。当一个操作执行完，主分片转发新文档（或者删除部分）给对应的复制节点，然后执行下一个操作。一旦所有复制节点报告所有操作已成功完成，节点就报告success给请求节点，后者(请求节点)整理响应并返回给客户端。

bulk API还可以在最上层使用replication和consistency参数，routing参数则在每个请求的元数据中使用。

### 批量格式（日志等）

当我们在《批量》一章中学习了批量请求后，你可能会问：“为什么bulk API需要带换行符的奇怪格式，而不是像mget API一样使用JSON数组？”

为了回答这个问题，我们需要简单的介绍一下背景：

批量中每个引用的文档属于不同的主分片，每个分片可能被分布于集群中的某个节点上。这意味着批量中的每个**操作(action)**需要被转发到对应的分片和节点上。

如果每个单独的请求被包装到JSON数组中，那意味着我们需要：

* 解析JSON为数组（包括文档数据，可能非常大）
* 检查每个请求决定应该到哪个分片上
* 为每个分片创建一个请求的数组
* 序列化这些数组为内部传输格式
* 发送请求到每个分片

这可行，但需要大量的RAM来承载本质上相同的数据，还要创建更多的数据结构使得JVM花更多的时间执行垃圾回收。

取而代之的，Elasticsearch则是从网络缓冲区中一行一行的直接读取数据。它使用换行符识别和解析**action/metadata**行，以决定哪些分片来处理这个请求。

这些行请求直接转发到对应的分片上。这些没有冗余复制，没有多余的数据结构。整个请求过程使用最小的内存在进行。

## 搜索

到目前为止，我们已经学会了如何使用elasticsearch作为一个简单的NoSQL风格的分布式文件存储器——我们可以将一个JSON文档扔给Elasticsearch，也可以根据ID检索它们。但Elasticsearch真正强大之处在于可以从混乱的数据中找出有意义的信息——从大数据到全面的信息。

这也是为什么我们使用结构化的JSON文档，而不是无结构的二进制数据。Elasticsearch不只会**存储(store)**文档，也会**索引(indexes)**文档内容来使之可以被搜索。

**每个文档里的字段都会被索引并被查询**。而且不仅如此。在简单查询时，Elasticsearch可以使用**所有**的索引，以非常快的速度返回结果。这让你永远不必考虑传统数据库的一些东西。

A *search* can be: **搜索(search)**可以：

* 在类似于gender或者age这样的字段上使用结构化查询，join\_date这样的字段上使用排序，就像SQL的结构化查询一样。
* 全文检索，可以使用所有字段来匹配关键字，然后按照**关联性(relevance)**排序返回结果。
* 或者结合以上两条。

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

| **概念** | **解释** |
| --- | --- |
| **映射(Mapping)** | 数据在每个字段中的解释说明 |
| **分析(Analysis)** | 全文是如何处理的可以被搜索的 |
| **领域特定语言查询(Query DSL)** | Elasticsearch使用的灵活的、强大的查询语言 |

以上提到的每个点都是一个巨大的话题，我们将在《深入搜索》一章阐述它们。本章节我们将介绍这三点的一些基本概念——仅仅帮助你大致了解搜索是如何工作的。

我们将使用最简单的形式开始介绍search API.

### 空搜索

最基本的搜索API表单是**空搜索(empty search)**，它没有指定任何的查询条件，只返回集群索引中的所有文档：

GET /\_search

{

"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

响应中最重要的部分是hits，它包含了total字段来表示匹配到的文档总数，hits数组还包含了匹配到的前10条数据。

hits数组中的每个结果都包含\_index、\_type和文档的\_id字段，被加入到\_source字段中这意味着在搜索结果中我们将可以直接使用全部文档。这不像其他搜索引擎只返回文档ID，需要你单独去获取文档。

每个节点都有一个\_score字段，这是**相关性得分(relevance score)**，它衡量了文档与查询的匹配程度。默认的，返回的结果中关联性最大的文档排在首位；这意味着，它是按照\_score降序排列的。这种情况下，我们没有指定任何查询，所以所有文档的相关性是一样的，因此所有结果的\_score都是取得一个中间值1

max\_score指的是所有文档匹配查询中\_score的最大值。

#### Took

搜索请求花费的毫秒数

#### Shards

\_shards节点告诉我们参与查询的分片数（total字段），有多少是成功的（successful字段），有多少的是失败的（failed字段）。通常我们不希望分片失败，不过这个有可能发生。如果我们遭受一些重大的故障导致主分片和复制分片都故障，那这个分片的数据将无法响应给搜索请求。这种情况下，Elasticsearch将报告分片failed，但仍将继续返回剩余分片上的结果

#### Timeout

time\_out值告诉我们查询超时与否。一般的，搜索请求不会超时。如果响应速度比完整的结果更重要，你可以定义timeout参数为10或者10ms（10毫秒），或者1s（1秒）

GET /\_search?timeout=10ms

Elasticsearch将返回在请求超时前收集到的结果。

超时不是一个断路器（circuit breaker）（译者注：关于断路器的理解请看警告）。

**警告**

需要注意的是timeout不会停止执行查询，它仅仅告诉你**目前**顺利返回结果的节点然后关闭连接。在后台，其他分片可能依旧执行查询，尽管结果已经被发送。

使用超时是因为对于你的业务需求（译者注：SLA，Service-Level Agreement服务等级协议，在此我翻译为业务需求）来说非常重要，而不是因为你想中断执行长时间运行的查询

### 多索引和多类型

你注意到空搜索的结果中不同类型的文档——user和tweet——来自于不同的索引——us和gb。

通过限制搜索的不同索引或类型，我们可以在集群中跨**所有**文档搜索。Elasticsearch转发搜索请求到集群中平行的主分片或每个分片的复制分片上，收集结果后选择顶部十个返回给我们。

通常，当然，你可能想搜索一个或几个自定的索引或类型，我们能通过定义URL中的索引或类型达到这个目的，像这样：

/\_search

在所有索引的所有类型中搜索

/gb/\_search

在索引gb的所有类型中搜索

/gb,us/\_search

在索引gb和us的所有类型中搜索

/g\*,u\*/\_search

在以g或u开头的索引的所有类型中搜索

/gb/user/\_search

在索引gb的类型user中搜索

/gb,us/user,tweet/\_search

在索引gb和us的类型为user和tweet中搜索

/\_all/user,tweet/\_search

在所有索引的user和tweet中搜索 search types user and tweet in all indices

当你搜索包含单一索引时，Elasticsearch转发搜索请求到这个索引的主分片或每个分片的复制分片上，然后聚集每个分片的结果。搜索包含多个索引也是同样的方式——只不过或有更多的分片被关联。

**搜索一个索引有5个主分片和5个索引各有一个分片事实上是一样的**。

### 分页

《空搜索》一节告诉我们在集群中有14个文档匹配我们的（空）搜索语句。但是只有10个文档在hits数组中。我们如何看到其他文档？

和SQL使用LIMIT关键字返回只有一页的结果一样，Elasticsearch接受from和size参数：

size: 结果数，默认10

from: 跳过开始的结果数，默认0

如果你想每页显示5个结果，页码从1到3，那请求如下：

GET /\_search?size=5

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

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

应该当心分页太深或者一次请求太多的结果。结果在返回前会被排序。但是记住一个搜索请求常常涉及多个分片。每个分片生成自己排好序的结果，它们接着需要集中起来排序以确保整体排序正确。

#### 在集群系统中深度分页

为了理解为什么深度分页是有问题的，让我们假设在一个有5个主分片的索引中搜索。当我们请求结果的第一页（结果1到10）时，每个分片产生自己最顶端10个结果然后返回它们给**请求节点(requesting node)**，它再排序这所有的50个结果以选出顶端的10个结果。

现在假设我们请求第1000页——结果10001到10010。工作方式都相同，不同的是每个分片都必须产生顶端的10010个结果。然后请求节点排序这50050个结果并丢弃50040个！

你可以看到在分布式系统中，排序结果的花费随着分页的深入而成倍增长。这也是为什么网络搜索引擎中任何语句不能返回多于1000个结果的原因。

### 查询字符串

#### 简易搜索

search API有两种表单：一种是“简易版”的**查询字符串(query string)**将所有参数通过查询字符串定义，另一种版本使用JSON完整的表示**请求体(request body)**，这种富搜索语言叫做结构化查询语句（DSL）

查询字符串搜索对于在命令行下运行**点对点(ad hoc)**查询特别有用。例如这个语句查询所有类型为tweet并在tweet字段中包含elasticsearch字符的文档：

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

下一个语句查找name字段中包含"john"和tweet字段包含"mary"的结果。实际的查询只需要：

+name:john +tweet:mary

但是**百分比编码(percent encoding)**（译者注：就是url编码）需要将查询字符串参数变得更加神秘：

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

"+"前缀表示语句匹配条件**必须**被满足。类似的"-"前缀表示条件**必须不**被满足。所有条件如果没有+或-表示是可选的——匹配越多，相关的文档就越多。

#### \_all字段

返回包含"mary"字符的所有文档的简单搜索：

GET /\_search?q=mary

在前一个例子中，我们搜索tweet或name字段中包含某个字符的结果。然而，这个语句返回的结果在三个不同的字段中包含"mary"：

* 用户的名字是“Mary”
* “Mary”发的六个推文
* 针对“@mary”的一个推文

Elasticsearch是如何设法找到三个不同字段的结果的？

当你索引一个文档，Elasticsearch把所有字符串字段值连接起来放在一个大字符串中，它被索引为一个特殊的字段\_all。例如，当索引这个文档：

{

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

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

"name": "Mary Jones",

"user\_id": 1

}

这好比我们增加了一个叫做\_all的额外字段值：

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

若没有指定字段，查询字符串搜索（即q=xxx）使用\_all字段搜索。

**TIP**

\_all字段对于开始一个新应用时是一个有用的特性。之后，如果你定义字段来代替\_all字段，你的搜索结果将更加可控。当\_all字段不再使用，你可以停用它

#### 更复杂的语句

下一个搜索推特的语句：

\_all field

* name字段包含"mary"或"john"
* date晚于2014-09-10
* \_all字段包含"aggregations"或"geo"

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

编码后的查询字符串变得不太容易阅读：

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

就像你上面看到的例子，**简单(lite)**查询字符串搜索惊人的强大。它的查询语法，会在《查询字符串语法》章节阐述。参考文档允许我们简洁明快的表示复杂的查询。这对于命令行下一次性查询或者开发模式下非常有用。

然而，你可以看到简洁带来了隐晦和调试困难。而且它很脆弱——查询字符串中一个细小的语法错误，像-、:、/或"错位就会导致返回错误而不是结果。

最后，查询字符串搜索允许任意用户在索引中任何一个字段上运行潜在的慢查询语句，可能暴露私有信息甚至使你的集群瘫痪。

**TIP**

因为这些原因，我们不建议直接暴露查询字符串搜索给用户，除非这些用户对于你的数据和集群可信。

取而代之的，生产环境我们一般依赖全功能的**请求体**搜索API，它能完成前面所有的事情，甚至更多。在了解它们之前，我们首先需要看看数据是如何在Elasticsearch中被索引的。

## 映射和分析

映射（mapping）机制用于进行字段类型确认，将每个字段匹配为一种确定的数据类型（string，number，booleans，date等）。

分析（analysis）机制用于进行全文文本（Full Text）的分词，以建立供搜索用的反向索引。

### 数据类型差异

#### 映射和分析

当在索引中处理数据时，我们注意到一些奇怪的事。有些东西似乎被破坏了：

在索引中有12个tweets，只有一个包含日期2014-09-15，但是我们看看下面查询中的total hits。

GET /\_search?q=2014 # 12 个结果

GET /\_search?q=2014-09-15 # 还是 12 个结果 !

GET /\_search?q=date:2014-09-15 # 1 一个结果

GET /\_search?q=date:2014 # 0 个结果 !

为什么全日期的查询返回所有的tweets，而针对date字段进行年度查询却什么都不返回？ 为什么我们的结果因查询\_all字段(译者注：默认所有字段中进行查询)或date字段而变得不同？

想必是因为我们的数据在\_all字段的索引方式和在date字段的索引方式不同而导致。

让我们看看Elasticsearch在对gb索引中的tweet类型进行mapping(也称之为模式定义[注：此词有待重新定义(schema definition)])后是如何解读我们的文档结构：

GET /gb/\_mapping/tweet

返回：

{

"gb": {

"mappings": {

"tweet": {

"properties": {

"date": {

"type": "date",

"format": "dateOptionalTime"

},

"name": {

"type": "string"

},

"tweet": {

"type": "string"

},

"user\_id": {

"type": "long"

}

}

}

}

}

}

Elasticsearch为对字段类型进行猜测，动态生成了字段和类型的映射关系。返回的信息显示了date字段被识别为date类型。\_all因为是默认字段所以没有在此显示，不过我们知道它是string类型。

date类型的字段和string类型的字段的索引方式是不同的，因此导致查询结果的不同，这并不会让我们觉得惊讶。

你会期望每一种核心数据类型(strings, numbers, booleans及dates)以不同的方式进行索引，而这点也是现实：在Elasticsearch中他们是被区别对待的。

但是更大的区别在于确切值(exact values)(比如string类型)及全文文本(full text)之间。

这两者的区别才真的很重要 - 这是区分搜索引擎和其他数据库的根本差异

### 确切值对决全文

#### 确切值(Exact values) vs. 全文文本(Full text)

Elasticsearch中的数据可以大致分为两种类型：

*确切值* 及 *全文文本*。

确切值是确定的，正如它的名字一样。比如一个date或用户ID，也可以包含更多的字符串比如username或email地址。

确切值"Foo"和"foo"就并不相同。确切值2014和2014-09-15也不相同。

全文文本，从另一个角度来说是文本化的数据(常常以人类的语言书写)，比如一篇推文(Twitter的文章)或邮件正文。

全文文本常常被称为非结构化数据，其实是一种用词不当的称谓，实际上自然语言是高度结构化的。

问题是自然语言的语法规则是如此的复杂，计算机难以正确解析。例如这个句子：

May is fun but June bores me.

到底是说的月份还是人呢？

确切值是很容易查询的，因为结果是二进制的 -- 要么匹配，要么不匹配。下面的查询很容易以SQL表达：

WHERE name = "John Smith"

AND user\_id = 2

AND date > "2014-09-15"

而对于全文数据的查询来说，却有些微妙。我们不会去询问这篇文档是否匹配查询要求？。 但是，我们会询问这篇文档和查询的匹配程度如何？。换句话说，对于查询条件，这篇文档的*相关性*有多高？

我们很少确切的匹配整个全文文本。我们想在全文中查询*包含*查询文本的部分。不仅如此，我们还期望搜索引擎能理解我们的*意图*：

* 一个针对"UK"的查询将返回涉及"United Kingdom"的文档
* 一个针对"jump"的查询同时能够匹配"jumped"， "jumps"， "jumping"甚至"leap"
* "johnny walker"也能匹配"Johnnie Walker"， "johnnie depp"及"Johnny Depp"
* "fox news hunting"能返回有关hunting on Fox News的故事，而"fox hunting news"也能返回关于fox hunting的新闻故事。

为了方便在全文文本字段中进行这些类型的查 询，Elasticsearch首先对文本**分析(analyzes)**，然后使用结果建立一个**倒排索引**。我们将在以下两个章节讨论倒排索引及分析过程。

### 倒排索引

Elasticsearch使用一种叫做**倒排索引(inverted index)**的结构来做快速的全文搜索。倒排索引由在文档中出现的唯一的单词列表，以及对于每个单词在文档中的位置组成。

例如，我们有两个文档，每个文档content字段包含：

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

为了创建倒排索引，我们首先切分每个文档的content字段为单独的单词（我们把它们叫做**词(terms)**或者**表征(tokens)**）（译者注：关于terms和tokens的翻译比较生硬，只需知道语句分词后的个体叫做这两个。），把所有的唯一词放入列表并排序，结果是这个样子的：

| **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 |  |

现在，如果我们想搜索"quick brown"，我们只需要找到每个词在哪个文档中出现即可：

| **Term** | **Doc\_1** | **Doc\_2** |
| --- | --- | --- |
| brown | X | X |
| quick | X |  |
| ----- | ------- | ----- |
| Total | 2 | 1 |

两个文档都匹配，但是第一个比第二个有更多的匹配项。 如果我们加入简单的**相似度算法(similarity algorithm)**，计算匹配单词的数目，这样我们就可以说第一个文档比第二个匹配度更高——对于我们的查询具有更多相关性。

但是在我们的倒排索引中还有些问题：

1. "Quick"和"quick"被认为是不同的单词，但是用户可能认为它们是相同的。
2. "fox"和"foxes"很相似，就像"dog"和"dogs"——它们都是同根词。
3. "jumped"和"leap"不是同根词，但意思相似——它们是同义词。

上面的索引中，搜索"+Quick +fox"不会匹配任何文档（记住，前缀+表示单词必须匹配到）。只有"Quick"和"fox"都在同一文档中才可以匹配查询，但是第一个文档包含"quick fox"且第二个文档包含"Quick foxes"。（译者注：这段真啰嗦，说白了就是单复数和同义词没法匹配）

用户可以合理地希望两个文档都能匹配查询，我们也可以做得更好。

如果我们将词为统一为标准格式，这样就可以找到不是确切匹配查询，但是足以相似从而可以关联的文档。例如：

1. "Quick"可以转为小写成为"quick"。
2. "foxes"可以被转为根形式"fox"。同理"dogs"可以被转为"dog"。
3. "jumped"和"leap"同义就可以只索引为单个词"jump"

现在的索引：

| **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 |

但我们还未成功。我们的搜索"+Quick +fox"*依旧*失败，因为"Quick"的确切值已经不在索引里，不过，如果我们使用相同的标准化规则处理查询字符串的content字段，查询将变成"+quick +fox"，这样就可以匹配到两个文档。

**注意：你只可以找到确实存在于索引中的词，所以索引文本和查询字符串都要标准化为相同的形式。这个标记化和标准化的过程叫做分词（analysis）。**

### 分析

#### 分析和分析器

**分析(analysis)**是这样一个过程：

* 首先，标记化一个文本块为适用于倒排索引单独的**词(term)**
* 然后标准化这些词为标准形式，提高它们的“可搜索性”或“查全率”

这个工作是**分析器(analyzer)**完成的。一个**分析器(analyzer)**只是一个包装用于将三个功能放到一个包里：

#### 字符过滤器

首先字符串经过**字符过滤器(character filter)**，它们的工作是在标记化前处理字符串。字符过滤器能够去除HTML标记，或者转换"&"为"and"。

#### 分词器

下一步，**分词器(tokenizer)**被标记化成独立的词。一个简单的**分词器(tokenizer)**可以根据空格或逗号将单词分开（译者注：这个在中文中不适用）。

#### 标记过滤

最后，每个词都通过所有**标记过滤(token filters)**，它可以修改词（例如将"Quick"转为小写），去掉词（例如停用词像"a"、"and"、"the"等等），或者增加词（例如同义词像"jump"和"leap"）

Elasticsearch提供很多开箱即用的字符过滤器，分词器和标记过滤器。这些可以组合来创建自定义的分析器以应对不同的需求。我们将在《自定义分析器》章节详细讨论。

#### 内建的分析器

不过，Elasticsearch还附带了一些预装的分析器，你可以直接使用它们。下面我们列出了最重要的几个分析器，来演示这个字符串分词后的表现差异：

"Set the shape to semi-transparent by calling set\_trans(5)"

#### 标准分析器

标准分析器是Elasticsearch默认使用的分析器。对于文本分析，它对于任何语言都是最佳选择（译者注：就是没啥特殊需求，对于任何一个国家的语言，这个分析器就够用了）。它根据[Unicode Consortium](http://www.unicode.org/reports/tr29/)的定义的**单词边界(word boundaries)**来切分文本，然后去掉大部分标点符号。最后，把所有词转为小写。产生的结果为：

set, the, shape, to, semi, transparent, by, calling, set\_trans, 5

#### 简单分析器

简单分析器将非单个字母的文本切分，然后把每个词转为小写。产生的结果为：

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

#### 空格分析器

空格分析器依据空格切分文本。它不转换小写。产生结果为：

Set, the, shape, to, semi-transparent, by, calling, set\_trans(5)

#### 语言分析器

特定语言分析器适用于很多语言。它们能够考虑到特定语言的特性。例如，english分析器自带一套英语停用词库——像and或the这些与语义无关的通用词。这些词被移除后，因为语法规则的存在，英语单词的主体含义依旧能被理解（译者注：stem English words这句不知道该如何翻译，查了字典，我理解的大概意思应该是将英语语句比作一株植物，去掉无用的枝叶，主干依旧存在，停用词好比枝叶，存在与否并不影响对这句话的理解。）。

english分析器将会产生以下结果：

set, shape, semi, transpar, call, set\_tran, 5

注意"transparent"、"calling"和"set\_trans"是如何转为词干的。

#### 当分析器被使用

当我们**索引(index)**一个文档，全文字段会被分析为单独的词来创建倒排索引。不过，当我们在全文字段**搜索(search)**时，我们要让查询字符串经过**同样的分析流程**处理，以确保这些词在索引中存在。

全文查询我们将在稍后讨论，理解每个字段是如何定义的，这样才可以让它们做正确的事：

* 当你查询**全文(full text)**字段，查询将使用相同的分析器来分析查询字符串，以产生正确的词列表。
* 当你查询一个**确切值(exact value)**字段，查询将不分析查询字符串，但是你可以自己指定。

现在你可以明白为什么《映射和分析》的开头会产生那种结果：

* date字段包含一个确切值：单独的一个词"2014-09-15"。
* \_all字段是一个全文字段，所以分析过程将日期转为三个词："2014"、"09"和"15"。

当我们在\_all字段查询2014，它一个匹配到12条推文，因为这些推文都包含词2014：

GET /\_search?q=2014 # 12 results

当我们在\_all字段中查询2014-09-15，首先分析查询字符串，产生匹配**任一**词2014、09或15的查询语句，它依旧匹配12个推文，因为它们都包含词2014。

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

当我们在date字段中查询2014-09-15，它查询一个**确切**的日期，然后只找到一条推文：

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

当我们在date字段中查询2014，没有找到文档，因为没有文档包含那个确切的日期：

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

#### 测试分析器

尤其当你是Elasticsearch新手时，对于如何分词以及存储到索引中理解起来比较困难。为了更好的理解如何进行，你可以使用analyze API来查看文本是如何被分析的。在查询字符串参数中指定要使用的分析器，被分析的文本做为请求体：

GET /\_analyze?analyzer=standard&text=Text to analyze

结果中每个节点在代表一个词：

{

"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是一个实际被存储在索引中的词。position指明词在原文本中是第几个出现的。start\_offset和end\_offset表示词在原文本中占据的位置。

analyze API 对于理解Elasticsearch索引的内在细节是个非常有用的工具，随着内容的推进，我们将继续讨论它。

#### 指定分析器

当Elasticsearch在你的文档中探测到一个新的字符串字段，它将自动设置它为全文string字段并用standard分析器分析。

你不可能总是想要这样做。也许你想使用一个更适合这个数据的语言分析器。或者，你只想把字符串字段当作一个普通的字段——不做任何分析，只存储确切值，就像字符串类型的用户ID或者内部状态字段或者标签。

为了达到这种效果，我们必须通过**映射(mapping)**人工设置这些字段。

### 映射

#### 映射

为了能够把日期字段处理成日期，把数字字段处理成数字，把字符串字段处理成全文本（Full-text）或精确的字符串值，Elasticsearch需要知道每个字段里面都包含了什么类型。这些类型和字段的信息存储（包含）在映射（mapping）中。

正如《数据吞吐》一节所说，索引中每个文档都有一个**类型(type)**。 每个类型拥有自己的**映射(mapping)**或者**模式定义(schema definition)**。一个映射定义了字段类型，每个字段的数据类型，以及字段被Elasticsearch处理的方式。映射还用于设置关联到类型上的元数据。

在《映射》章节我们将探讨映射的细节。这节我们只是带你入门。

#### 核心简单字段类型

Elasticsearch支持以下简单字段类型：

| **类型** | **表示的数据类型** |
| --- | --- |
| String | string |
| Whole number | byte, short, integer, long |
| Floating point | float, double |
| Boolean | boolean |
| Date | date |

当你索引一个包含新字段的文档——一个之前没有的字段——Elasticsearch将使用动态映射猜测字段类型，这类型来自于JSON的基本数据类型，使用以下规则：

| **JSON type** | **Field type** |
| --- | --- |
| Boolean: true or false | "boolean" |
| Whole number: 123 | "long" |
| Floating point: 123.45 | "double" |
| String, valid date: "2014-09-15" | "date" |
| String: "foo bar" | "string" |

**注意**

这意味着，如果你索引一个带引号的数字——"123"，它将被映射为"string"类型，而不是"long"类型。然而，如果字段已经被映射为"long"类型，Elasticsearch将尝试转换字符串为long，并在转换失败时会抛出异常。

#### 查看映射

我们可以使用\_mapping后缀来查看Elasticsearch中的映射。在本章开始我们已经找到索引gb类型tweet中的映射：

GET /gb/\_mapping/tweet

这展示给了我们字段的映射（叫做**属性(properties)**），这些映射是Elasticsearch在创建索引时动态生成的：

{

"gb": {

"mappings": {

"tweet": {

"properties": {

"date": {

"type": "date",

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

},

"name": {

"type": "string"

},

"tweet": {

"type": "string"

},

"user\_id": {

"type": "long"

}

}

}

}

}

}

**小提示**

错误的映射，例如把age字段映射为string类型而不是integer类型，会造成查询结果混乱。

要检查映射类型，而不是假设它是正确的！

#### 自定义字段映射

虽然大多数情况下基本数据类型已经能够满足，但你也会经常需要自定义一些特殊类型（fields），特别是字符串字段类型。 自定义类型可以使你完成一下几点：

* 区分全文（full text）字符串字段和准确字符串字段（译者注：就是分词与不分词，全文的一般要分词，准确的就不需要分词，比如『中国』这个词。全文会分成『中』和『国』，但作为一个国家标识的时候我们是不需要分词的，所以它就应该是一个准确的字符串字段）。
* 使用特定语言的分析器（译者注：例如中文、英文、阿拉伯语，不同文字的断字、断词方式的差异）
* 优化部分匹配字段
* 指定自定义日期格式（译者注：这个比较好理解,例如英文的 Feb,12,2016 和 中文的 2016年2月12日）
* 以及更多

映射中最重要的字段参数是type。除了string类型的字段，你可能很少需要映射其他的type：

{

"number\_of\_clicks": {

"type": "integer"

}

}

string类型的字段，默认的，考虑到包含全文本，它们的值在索引前要经过分析器分析，并且在全文搜索此字段前要把查询语句做分析处理。

对于string字段，两个最重要的映射参数是index和analyer。

#### index

index参数控制字符串以何种方式被索引。它包含以下三个值当中的一个：

| **值** | **解释** |
| --- | --- |
| analyzed | 首先分析这个字符串，然后索引。换言之，以全文形式索引此字段。 |
| not\_analyzed | 索引这个字段，使之可以被搜索，但是索引内容和指定值一样。不分析此字段。 |
| no | 不索引这个字段。这个字段不能为搜索到。 |

string类型字段默认值是analyzed。如果我们想映射字段为确切值，我们需要设置它为not\_analyzed：

{

"tag": {

"type": "string",

"index": "not\_analyzed"

}

}

其他简单类型（long、double、date等等）也接受index参数，但相应的值只能是no和not\_analyzed，它们的值不能被分析。

#### 分析

对于analyzed类型的字符串字段，使用analyzer参数来指定哪一种分析器将在搜索和索引的时候使用。默认的，Elasticsearch使用standard分析器，但是你可以通过指定一个内建的分析器来更改它，例如whitespace、simple或english。

{

"tweet": {

"type": "string",

"analyzer": "english"

}

}

在《自定义分析器》章节我们将告诉你如何定义和使用自定义的分析器。

#### 更新映射

你可以在第一次创建索引的时候指定映射的类型。此外，你也可以晚些时候为新类型添加映射（或者为已有的类型更新映射）。

**重要**

你可以向已有映射中**增加**字段，但你不能**修改**它。如果一个字段在映射中已经存在，这可能意味着那个字段的数据已经被索引。如果你改变了字段映射，那已经被索引的数据将错误并且不能被正确的搜索到。

我们可以更新一个映射来增加一个新字段，但是不能把已有字段的类型那个从analyzed改到not\_analyzed。

为了演示两个指定的映射方法，让我们首先删除索引gb：

DELETE /gb

然后创建一个新索引，指定tweet字段的分析器为english：

PUT /gb <1>

{

"mappings": {

"tweet" : {

"properties" : {

"tweet" : {

"type" : "string",

"analyzer": "english"

},

"date" : {

"type" : "date"

},

"name" : {

"type" : "string"

},

"user\_id" : {

"type" : "long"

}

}

}

}

}

<1> 这将创建包含mappings的索引，映射在请求体中指定。

再后来，我们决定在tweet的映射中增加一个新的not\_analyzed类型的文本字段，叫做tag，使用\_mapping后缀:

PUT /gb/\_mapping/tweet

{

"properties" : {

"tag" : {

"type" : "string",

"index": "not\_analyzed"

}

}

}

注意到我们不再需要列出所有的已经存在的字段，因为我们没法修改他们。我们的新字段已经被合并至存在的那个映射中。

#### 测试映射

你可以通过名字使用analyze API测试字符串字段的映射。对比这两个请求的输出：

GET /gb/\_analyze?field=tweet&text=Black-cats <1>

GET /gb/\_analyze?field=tag&text=Black-cats <2>

<1> <2> 我们想要分析的文本被放在请求体中。

tweet字段产生两个词，"black"和"cat",tag字段产生单独的一个词"Black-cats"。换言之，我们的映射工作正常。

### 复合类型

#### 复合核心字段类型

除了之前提到的简单的标量类型，JSON还有null值，数组和对象，所有这些Elasticsearch都支持：

#### 多值字段

我们想让tag字段包含多个字段，这非常有可能发生。我们可以索引一个标签数组来代替单一字符串：

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

对于数组不需要特殊的映射。任何一个字段可以包含零个、一个或多个值，同样对于全文字段将被分析并产生多个词。

言外之意，这意味着**数组中所有值必须为同一类型**。你不能把日期和字符串混合。如果你创建一个新字段，这个字段索引了一个数组，Elasticsearch将使用第一个值的类型来确定这个新字段的类型。

当你从Elasticsearch中取回一个文档，任何一个数组的顺序和你索引它们的顺序一致。你取回的\_source字段的顺序同样与索引它们的顺序相同。

然而，数组是做为多值字段被**索引**的，它们没有顺序。在搜索阶段你不能指定“第一个值”或者“最后一个值”。倒不如把数组当作一个**值集合(bag of values)**

#### 空字段

当然数组可以是空的。这等价于有零个值。事实上，Lucene没法存放null值，所以一个null值的字段被认为是空字段。

这四个字段将被识别为空字段而不被索引：

"empty\_string": "",

"null\_value": null,

"empty\_array": [],

"array\_with\_null\_value": [ null ]

#### 多层对象

我们需要讨论的最后一个自然JSON数据类型是**对象(object)**——在其它语言中叫做hash、hashmap、dictionary 或者 associative array.

**内部对象(inner objects)**经常用于在另一个对象中嵌入一个实体或对象。例如，做为在tweet文档中user\_name和user\_id的替代，我们可以这样写：

{

"tweet": "Elasticsearch is very flexible",

"user": {

"id": "@johnsmith",

"gender": "male",

"age": 26,

"name": {

"full": "John Smith",

"first": "John",

"last": "Smith"

}

}

}

#### 内部对象的映射

Elasticsearch 会动态的检测新对象的字段，并且映射它们为 object 类型，将每个字段加到 properties 字段下

{

"gb": {

"tweet": { <1>

"properties": {

"tweet": { "type": "string" },

"user": { <2>

"type": "object",

"properties": {

"id": { "type": "string" },

"gender": { "type": "string" },

"age": { "type": "long" },

"name": { <3>

"type": "object",

"properties": {

"full": { "type": "string" },

"first": { "type": "string" },

"last": { "type": "string" }

}

}

}

}

}

}

}

}

<1> 根对象.

<2><3> 内部对象.

对user和name字段的映射与tweet类型自己很相似。事实上，type映射只是object映射的一种特殊类型，我们将 object 称为根对象。它与其他对象一模一样，除非它有一些特殊的顶层字段，比如 \_source, \_all 等等。

#### 内部对象是怎样被索引的

Lucene 并不了解内部对象。 一个 Lucene 文件包含一个键-值对应的扁平表单。 为了让 Elasticsearch 可以有效的索引内部对象，将文件转换为以下格式：

{

"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]

}

内部栏位可被归类至name，例如"first"。 为了区别两个拥有相同名字的栏位，我们可以使用完整路径，例如"user.name.first" 或甚至类型名称加上路径："tweet.user.name.first"。

注意： 在以上扁平化文件中，并没有栏位叫作user也没有栏位叫作user.name。 Lucene 只索引阶层或简单的值，而不会索引复杂的资料结构。

#### 对象-数组

#### 内部对象数组

最后，一个包含内部对象的数组如何索引。 我们有个数组如下所示：

{

"followers": [

{ "age": 35, "name": "Mary White"},

{ "age": 26, "name": "Alex Jones"},

{ "age": 19, "name": "Lisa Smith"}

]

}

此文件会如我们以上所说的被扁平化，但其结果会像如此：

{

"followers.age": [19, 26, 35],

"followers.name": [alex, jones, lisa, smith, mary, white]

}

{age: 35}与{name: Mary White}之间的关联会消失，因每个多值的栏位会变成一个值集合，而非有序的阵列。 这让我们可以知道：

* 是否有26岁的追随者？

但我们无法取得准确的资料如：

* 是否有26岁的追随者***且名字叫Alex Jones？***

关联内部对象可解决此类问题，我们称之为嵌套对象

## 自定义分析器

倒排索引存储了比包含了一个特定term的文档列表多地多的信息。它可能存储包含每个term的文档数量，一个term出现在指定文档中的频次，每个文档中term的顺序，每个文档的长度，所有文档的平均长度，等等。这些统计信息让Elasticsearch知道哪些term更重要，哪些文档更重要，也就是[相关性](http://es.xiaoleilu.com/075_Inside_a_shard/056_Sorting/90_What_is_relevance.md)。

写入磁盘的倒排索引是不可变的，它有如下好处：

* 不需要锁。如果从来不需要更新一个索引，就不必担心多个程序同时尝试修改。
* 一旦索引被读入文件系统的缓存(译者:在内存)，它就一直在那儿，因为不会改变。只要文件系统缓存有足够的空间，大部分的读会直接访问内存而不是磁盘。这有助于性能提升。
* 在索引的声明周期内，所有的其他缓存都可用。它们不需要在每次数据变化了都重建，因为数据不会变。
* 写入单个大的倒排索引，可以压缩数据，较少磁盘IO和需要缓存索引的内存大小。

当然，不可变的索引有它的缺点，首先是它不可变！你不能改变它。如果想要搜索一个新文档，必须重见整个索引。这不仅严重限制了一个索引所能装下的数据，还有一个索引可以被更新的频次。