# 最完整的Elasticsearch 基础教程

2014-10-21 15:32:01| 发布人：PHP_PentaKill| 浏览(12254)| 评论(1)

翻译：潘飞（tinylambda@gmail.com）

基础概念

Elasticsearch有几个核心概念。从一开始理解这些概念会对整个学习过程有莫大的帮助。

接近实时（NRT）

Elasticsearch是一个接近实时的搜索平台。这意味着，从索引一个文档直到这个文档能够被搜索到有一个轻微的延迟（通常是1秒）。

集群（cluster）

一个集群就是由一个或多个节点组织在一起，它们共同持有你整个的数据，并一起提供索引和搜索功能。一个集群由一个唯一的名字标识，这个名字默认就是“elasticsearch”。这个名字是重要的，因为一个节点只能通过指定某个集群的名字，来加入这个集群。在产品环境中显式地设定这个名字是一个好习惯，但是使用默认值来进行测试/开发也是不错的。

节点（node）

一个节点是你集群中的一个服务器，作为集群的一部分，它存储你的数据，参与集群的索引和搜索功能。和集群类似，一个节点也是由一个名字来标识的，默认情况下，这个名字是一个随机的漫威漫画角色的名字，这个名字会在启动的时候赋予节点。这个名字对于管理工作来说挺重要的，因为在这个管理过程中，你会去确定网络中的哪些服务器对应于Elasticsearch集群中的哪些节点。

一个节点可以通过配置集群名称的方式来加入一个指定的集群。默认情况下，每个节点都会被安排加入到一个叫做“elasticsearch”的集群中，这意味着，如果你在你的网络中启动了若干个节点，并假定它们能够相互发现彼此，它们将会自动地形成并加入到一个叫做“elasticsearch”的集群中。

在一个集群里，只要你想，可以拥有任意多个节点。而且，如果当前你的网络中没有运行任何Elasticsearch节点，这时启动一个节点，会默认创建并加入一个叫做“elasticsearch”的集群。

索引（index）

一个索引就是一个拥有几分相似特征的文档的集合。比如说，你可以有一个客户数据的索引，另一个产品目录的索引，还有一个订单数据的索引。一个索引由一个名字来标识（必须全部是小写字母的），并且当我们要对对应于这个索引中的文档进行索引、搜索、更新和删除的时候，都要使用到这个名字。

在一个集群中，如果你想，可以定义任意多的索引。

类型（type）

在一个索引中，你可以定义一种或多种类型。一个类型是你的索引的一个逻辑上的分类/分区，其语义完全由你来定。通常，会为具有一组共同字段的文档定义一个类型。比如说，我们假设你运营一个博客平台并且将你所有的数据存储到一个索引中。在这个索引中，你可以为用户数据定义一个类型，为博客数据定义另一个类型，当然，也可以为评论数据定义另一个类型。

文档（document）

一个文档是一个可被索引的基础信息单元。比如，你可以拥有某一个客户的文档，某一个产品的一个文档，当然，也可以拥有某个订单的一个文档。文档以JSON（Javascript Object Notation）格式来表示，而JSON是一个到处存在的互联网数据交互格式。

在一个index/type里面，只要你想，你可以存储任意多的文档。注意，尽管一个文档，物理上存在于一个索引之中，文档必须被索引/赋予一个索引的type。

分片和复制（shards & replicas）

一个索引可以存储超出单个结点硬件限制的大量数据。比如，一个具有10亿文档的索引占据1TB的磁盘空间，而任一节点都没有这样大的磁盘空间；或者单个节点处理搜索请求，响应太慢。

为了解决这个问题，Elasticsearch提供了将索引划分成多份的能力，这些份就叫做分片。当你创建一个索引的时候，你可以指定你想要的分片的数量。每个分片本身也是一个功能完善并且独立的“索引”，这个“索引”可以被放置到集群中的任何节点上。

分片之所以重要，主要有两方面的原因：

- 允许你水平分割/扩展你的内容容量
- 允许你在分片（潜在地，位于多个节点上）之上进行分布式的、并行的操作，进而提高性能/吞吐量

至于一个分片怎样分布，它的文档怎样聚合回搜索请求，是完全由Elasticsearch管理的，对于作为用户的你来说，这些都是透明的。

在一个网络/云的环境里，失败随时都可能发生，在某个分片/节点不知怎么的就处于离线状态，或者由于任何原因消失了，这种情况下，有一个故障转移机制是非 常有用并且是强烈推荐的。为此目的，Elasticsearch允许你创建分片的一份或多份拷贝，这些拷贝叫做复制分片，或者直接叫复制。

复制之所以重要，有两个主要原因：
- 在分片/节点失败的情况下，提供了高可用性。因为这个原因，注意到复制分片从不与原/主要（original/primary）分片置于同一节点上是非常重要的。
- 扩展你的搜索量/吞吐量，因为搜索可以在所有的复制上并行运行

总之，每个索引可以被分成多个分片。一个索引也可以被复制0次（意思是没有复制）或多次。一旦复制了，每个索引就有了主分片（作为复制源的原来的分片）和 复制分片（主分片的拷贝）之别。分片和复制的数量可以在索引创建的时候指定。在索引创建之后，你可以在任何时候动态地改变复制的数量，但是你事后不能改变 分片的数量。

默认情况下，Elasticsearch中的每个索引被分片5个主分片和1个复制，这意味着，如果你的集群中至少有两个节点，你的索引将会有5个主分片和另外5个复制分片（1个完全拷贝），这样的话每个索引总共就有10个分片。

这些问题搞清楚之后，我们就要进入好玩的部分了...

安装

Elasticsearch依赖Java 7。在本文写作的时候，推荐使用Oracle JDK 1.7.0_55版本。Java的安装，在各个平台上都有差异，所以我们不想在这里深入太多细节。我只想说，在你安装Elasticsearch之前，你 可以通过以下命令来检查你的Java版本（如果有需要，安装或者升级）：

```
java -version
echo $JAVA_HOME
```

一旦我们将Java安装完成，我们就可以下载并安装Elasticsearch了。其二进制文件可以从 www.elasticsearch.org/download这里下载，你也可以从这里下载以前发布的版本。对于每个版本，你可以在zip、tar、 DEB、RPM类型的包中选择下载。简单起见，我们使用tar包。

我们像下面一样下载Elasticsearch 1.1.1 tar包（Windows用户应该下载zip包）：

```
curl -L -O https://download.elasticsearch.org/elasticsearch/elasticsearch/elasticsearch-1.1.1.tar.gz
```

然后，如下将其解压（Windows下需要unzip响应的zip包）：

```
tar -xzvf elasticsearch-1.1.1.tar.gz
```

这将在你的当前目录下创建很多文件和目录。然后，我们进入到bin目录下：

```
cd elasticsearch-1.1.1/bin
```

至此，我们已经准备好开启我们的节点和单节点集群（Windows用户应该运行elasticsearch.bat文件）：

```
./elasticsearch
```

如果一切顺利，你将看到大量的如下信息：

./elasticsearch
[2014-03-13 13:42:17,218][INFO ][node ] [New Goblin] version[1.1.1], pid[2085], build[5c03844/2014-02-25T15:52:53Z]
[2014-03-13 13:42:17,219][INFO ][node ] [New Goblin] initializing ...
[2014-03-13 13:42:17,223][INFO ][plugins ] [New Goblin] loaded [], sites []
[2014-03-13 13:42:19,831][INFO ][node ] [New Goblin] initialized
[2014-03-13 13:42:19,832][INFO ][node ] [New Goblin] starting ...
[2014-03-13 13:42:19,958][INFO ][transport ] [New Goblin] bound_address {inet[/0:0:0:0:0:0:0:0:9300]}, publish_address {inet[/192.168.8.112:9300]}
[2014-03-13 13:42:23,030][INFO ][cluster.service] [New Goblin] new_master [New Goblin][rWMtGj3dQouz2r6ZFL9v4g][mwubuntu1][inet[/192.168.8.112:9300]], reason: zen-disco-join (elected_as_master)
[2014-03-13 13:42:23,100][INFO ][discovery ] [New Goblin] elasticsearch/rWMtGj3dQouz2r6ZFL9v4g
[2014-03-13 13:42:23,125][INFO ][http ] [New Goblin] bound_address {inet[/0:0:0:0:0:0:0:0:9200]}, publish_address {inet[/192.168.8.112:9200]}
[2014-03-13 13:42:23,629][INFO ][gateway ] [New Goblin] recovered [1] indices into cluster_state
[2014-03-13 13:42:23,630][INFO ][node ] [New Goblin] started

不去涉及太多细节，我们可以看到，一叫做“New Goblin”（你会见到一个不同的漫威漫画角色）的节点启动并且将自己选做单结点集群的master。现在不用关心master是什么东西。这里重要的就是，我们在一个集群中开启了一个节点。

正如先前提到的，我们可以覆盖集群或者节点的名字。我们可以在启动Elasticsearch的时候通过命令行来指定，如下：

./elasticsearch --cluster.name my_cluster_name --node.name my_node_name

也要注意一下有http标记的那一行，它提供了有关HTTP地址（192.168.8.112）和端口（9200）的信息，通过这个地址和端口我们就可以 访问我们的节点了。默认情况下，Elasticsearch使用9200来提供对其REST API的访问。如果有必要，这个端口是可以配置的。

## 探索你的集群

### rest接口

现在我们已经有一个正常运行的节点（和集群）了，下一步就是要去理解怎样与其通信了。幸运的是，Elasticsearch提供了非常全面和强大的 REST API，利用这个REST API你可以同你的集群交互。下面是利用这个API，可以做的几件事情：

- 检查你的集群、节点和索引的健康状态、和各种统计信息
- 管理你的集群、节点、索引数据和元数据
- 对你的索引进行CRUD（创建、读取、更新和删除）和搜索操作
- 执行高级的查询操作，像是分页、排序、过滤、脚本编写（scripting）、小平面刻画（faceting）、聚合（aggregations）和许多其它操作

### 集群健康

让我们以基本的健康检查作为开始，我们可以利用它来查看我们集群的状态。此过程中，我们使用curl，当然，你也可以使用任何可以创建HTTP/REST 调用的工具。我们假设我们还在我们启动Elasticsearch的节点上并打开另外一个shell窗口。

要检查集群健康，我们将使用_cat API。需要事先记住的是，我们的节点HTTP的端口是9200：

```
curl 'localhost:9200/_cat/health?v'
```

相应的响应是：

```
epoch      timestamp cluster       status node.total node.data shards pri relo init unassign
1394735289 14:28:09  elasticsearch green           1         1      0   0    0    0        0
```

可以看到，我们集群的名字是“elasticsearch”，正常运行，并且状态是绿色。

当我们询问集群状态的时候，我们要么得到绿色、黄色或红色。绿色代表一切正常（集群功能齐全），黄色意味着所有的数据都是可用的，但是某些复制没有被分配（集群功能齐全），红色则代表因为某些原因，某些数据不可用。注意，即使是集群状态是红色的，集群仍然是部分可用的（它仍然会利用可用的分片来响应搜索请 求），但是可能你需要尽快修复它，因为你有丢失的数据。

也是从上面的响应中，我们可以看到，一共有一个节点，由于里面没有数据，我们有0个分片。注意，由于我们使用默认的集群名字

（elasticsearch），并且由于Elasticsearch默认使用网络多播（multicast）发现其它节点，如果你在你的网络中启动了多 个节点，你就已经把她们加入到一个集群中了。在这种情形下，你可能在上面的响应中看到多个节点。

我们也可以获得节集群中的节点列表：

```
curl 'localhost:9200/_cat/nodes?v'
```

对应的响应是:

```
curl 'localhost:9200/_cat/nodes?v'
host        ip        heap.percent ram.percent load node.role master name
mwubuntu1   127.0.1.1            8           4 0.00 d         *      New Goblin
```

这儿，我们可以看到我们叫做“New Goblin”的节点，这个节点是我们集群中的唯一节点。

列出所有的索引

让我们看一下我们的索引：

```
curl 'localhost:9200/_cat/indices?v'
```

响应是：

```
curl 'localhost:9200/_cat/indices?v'
health index pri rep docs.count docs.deleted store.size pri.store.size
```

这个结果意味着，在我们的集群中，我们没有任何索引。

创建一个索引

现在让我们创建一个叫做“customer”的索引，然后再列出所有的索引：

```
curl -XPUT 'localhost:9200/customer?pretty'
curl 'localhost:9200/_cat/indices?v'
```

第一个命令使用PUT创建了一个叫做“customer”的索引。我们简单地将pretty附加到调用的尾部，使其以美观的形式打印出JSON响应（如果有的话）。

响应如下：

```
curl -XPUT 'localhost:9200/customer?pretty'
{
  "acknowledged" : true
}


curl 'localhost:9200/_cat/indices?v'
health index    pri rep docs.count docs.deleted store.size pri.store.size
yellow customer   5   1          0            0      495b           495b
```

第二个命令的结果告知我们，我们现在有一个叫做customer的索引，并且它有5个主分片和1份复制（都是默认值），其中包含0个文档。

你可能也注意到了这个customer索引有一个黄色健康标签。回顾我们之前的讨论，黄色意味着某些复制没有（或者还未）被分配。这个索引之所以这样，是 因为Elasticsearch默认为这个索引创建一份复制。由于现在我们只有一个节点在运行，那一份复制就分配不了了（为了高可用），直到当另外一个节 点加入到这个集群后，才能分配。一旦那份复制在第二个节点上被复制，这个节点的健康状态就会变成绿色。

索引并查询一个文档

现在让我们放一些东西到customer索引中。首先要知道的是，为了索引一个文档，我们必须告诉Elasticsearch这个文档要到这个索引的哪个类型（type）下。

让我们将一个简单的客户文档索引到customer索引、“external”类型中，这个文档的ID是1，操作如下：

```
curl -XPUT 'localhost:9200/customer/external/1?pretty' -d '
{
```

```
  "name": "John Doe"
}'
```

响应如下：

```
curl -XPUT 'localhost:9200/customer/external/1?pretty' -d '
{
  "name": "John Doe"
}'
{
  "_index" : "customer",
  "_type" : "external",
  "_id" : "1",
  "_version" : 1,
  "created" : true
}
```

从上面的响应中，我们可以看到，一个新的客户文档在customer索引和external类型中被成功创建。文档也有一个内部id 1， 这个id是我们在索引的时候指定的。

有一个关键点需要注意，Elasticsearch在你想将文档索引到某个索引的时候，并不强制要求这个索引被显式地创建。在前面这个例子中，如果customer索引不存在，Elasticsearch将会自动地创建这个索引。

现在，让我们把刚刚索引的文档取出来：

```
curl -XGET 'localhost:9200/customer/external/1?pretty'
```

响应如下：

```
curl -XGET 'localhost:9200/customer/external/1?pretty'
{
  "_index" : "customer",
  "_type" : "external",
  "_id" : "1",
  "_version" : 1,
  "found" : true, "_source" : { "name": "John Doe" }
}
```

除了一个叫做found的字段来指明我们找到了一个ID为1的文档，和另外一个字段——_source——返回我们前一步中索引的完整JSON文档之外，其它的都没有什么特别之处。

删除一个文档

现在让我们删除我们刚刚创建的索引，并再次列出所有的索引：

```
curl -XDELETE 'localhost:9200/customer?pretty'
curl 'localhost:9200/_cat/indices?v'
```

响应如下：

```
curl -XDELETE 'localhost:9200/customer?pretty'
{
  "acknowledged" : true
}
curl 'localhost:9200/_cat/indices?v'
health index pri rep docs.count docs.deleted store.size pri.store.size
```

这表明我们成功地删除了这个索引，现在我们回到了集群中空无所有的状态。

在更进一步之前，我们再细看一下一些我们学过的API命令：

```
curl -XPUT 'localhost:9200/customer'
curl -XPUT 'localhost:9200/customer/external/1' -d '
{
  "name": "John Doe"
}'
curl 'localhost:9200/customer/external/1'
```

curl -XDELETE 'localhost:9200/customer'

如果我们仔细研究以上的命令，我们可以发现访问Elasticsearch中数据的一个模式。这个模式可以被总结为：

curl - ://

这个REST访问模式普遍适用于所有的API命令，如果你能记住它，你就会为掌握Elasticsearch开一个好头。

修改你的数据

Elasticsearch提供了近乎实时的数据操作和搜索功能。默认情况下，从你索引/更新/删除你的数据动作开始到它出现在你的搜索结果中，大概会有1秒钟的延迟。这和其它类似SQL的平台不同，数据在一个事务完成之后就会立即可用。

索引/替换文档

我们先前看到，怎样索引一个文档。现在我们再次调用那个命令：

```
curl -XPUT 'localhost:9200/customer/external/1?pretty' -d '
{
  "name": "John Doe"
}'
```

再次，以上的命令将会把这个文档索引到customer索引、external类型中，其ID是1。如果我们对一个不同（或相同）的文档应用以上的命令，Elasticsearch将会用一个新的文档来替换（重新索引）当前ID为1的那个文档。

```
curl -XPUT 'localhost:9200/customer/external/1?pretty' -d '
{
  "name": "Jane Doe"
}'
```

以上的命令将ID为1的文档的name字段的值从“John Doe”改成了“Jane Doe”。如果我们使用一个不同的ID，一个新的文档将会被索引，当前已经在索引中的文档不会受到影响。

```
curl -XPUT 'localhost:9200/customer/external/2?pretty' -d '
{
  "name": "Jane Doe"
}'
```

以上的命令，将会索引一个ID为2的新文档。

在索引的时候，ID部分是可选的。如果不指定，Elasticsearch将产生一个随机的ID来索引这个文档。Elasticsearch生成的ID会作为索引API调用的一部分被返回。

以下的例子展示了怎样在没有指定ID的情况下来索引一个文档：

```
curl -XPOST 'localhost:9200/customer/external?pretty' -d '
{
  "name": "Jane Doe"
}'
```

注意，在上面的情形中，由于我们没有指定一个ID，我们使用的是POST而不是PUT。

更新文档

除了可以索引、替换文档之外，我们也可以更新一个文档。但要注意，Elasticsearch底层并不支持原地更新。在我们想要做一次更新的时候，Elasticsearch先删除旧文档，然后在索引一个更新过的新文档。

下面的例子展示了怎样将我们ID为1的文档的name字段改成“Jane Doe”：

```
curl -XPOST 'localhost:9200/customer/external/1/_update?pretty' -d '
{
  "doc": { "name": "Jane Doe" }
}'
```

下面的例子展示了怎样将我们ID为1的文档的name字段改成“Jane Doe”的同时，给它加上age字段：

```
curl -XPOST 'localhost:9200/customer/external/1/_update?pretty' -d '
{
  "doc": { "name": "Jane Doe", "age": 20 }
}'
```

更新也可以通过使用简单的脚本来进行。这个例子使用一个脚本将age加5：

```
curl -XPOST 'localhost:9200/customer/external/1/_update?pretty' -d '
{
  "script" : "ctx._source.age += 5"
```

```
}'
```

在上面的例子中，ctx._source指向当前要被更新的文档。

注意，在写作本文时，更新操作只能一次应用在一个文档上。将来，Elasticsearch将提供同时更新符合指定查询条件的多个文档的功能（类似于SQL的UPDATE-WHERE语句）。

## 删除文档

删除文档是相当直观的。以下的例子展示了我们怎样删除ID为2的文档：

```
curl -XDELETE 'localhost:9200/customer/external/2?pretty'
```

我们也能够一次删除符合某个查询条件的多个文档。以下的例子展示了如何删除名字中包含“John”的所有的客户：

```
curl -XDELETE 'localhost:9200/customer/external/_query?pretty' -d '
{
  "query": { "match": { "name": "John" } }
}'
```

注意，以上的URI变成了/_query，以此来表明这是一个“查询删除”API，其中删除查询标准放在请求体中，但是我们仍然使用DELETE。现在先不要担心查询语法，我们将会在本教程后面的部分中涉及。

## 批处理：

除了能够对单个的文档进行索引、更新和删除之外，Elasticsearch也提供了以上操作的批量处理功能，这是通过使用_bulk API实现的。这个功能之所以重要，在于它提供了非常高效的机制来尽可能快的完成多个操作，与此同时使用尽可能少的网络往返。

作为一个快速的例子，以下调用在一次bulk操作中索引了两个文档（ID 1 - John Doe and ID 2 - Jane Doe）：

```
curl -XPOST 'localhost:9200/customer/external/_bulk?pretty' -d '
{"index":{"_id":"1"}}
{"name": "John Doe" }
{"index":{"_id":"2"}}
{"name": "Jane Doe" }
'
```

以下例子在一个bulk操作中，首先更新第一个文档（ID为1），然后删除第二个文档（ID为2）：

```
curl -XPOST 'localhost:9200/customer/external/_bulk?pretty' -d '
{"update":{"_id":"1"}}
{"doc": { "name": "John Doe becomes Jane Doe" } }
{"delete":{"_id":"2"}}
'
```

注意上面的delete动作，由于删除动作只需要被删除文档的ID，所以并没有对应的源文档。

bulk API按顺序执行这些动作。如果其中一个动作因为某些原因失败了，将会继续处理它后面的动作。当bulk API返回时，它将提供每个动作的状态（按照同样的顺序），所以你能够看到某个动作成功与否。

## 探索你的数据

### 样本数据集

现在我们对于基本的东西已经有了一些感觉，现在让我们尝试使用一些更加贴近现实的数据集。我已经准备了一些假想的客户的银行账户信息的JSON文档的样本。文档具有以下的模式（schema）：

```
{
    "account_number": 0,
    "balance": 16623,
    "firstname": "Bradshaw",
    "lastname": "Mckenzie",
    "age": 29,
    "gender": "F",
    "address": "244 Columbus Place",
    "employer": "Euron",
    "email": "bradshawmckenzie@euron.com",
    "city": "Hobucken",
    "state": "CO"
}
```

我是在http://www.json-generator.com/上生成这些数据的。

### 载入样本数据

你可以从https://github.com/bly2k/files/blob/master/accounts.zip?raw=true下载这个样本数据集。将其解压到当前目录下，如下，将其加载到我们的集群里：

```
curl -XPOST 'localhost:9200/bank/account/_bulk?pretty' --data-binary @accounts.json
curl 'localhost:9200/_cat/indices?v'
```

响应是：

```
curl 'localhost:9200/_cat/indices?v'
health index pri rep docs.count docs.deleted store.size pri.store.size
yellow bank    5   1       1000            0    424.4kb        424.4kb
```

这意味着我们成功批量索引了1000个文档到银行索引中（account类型）。

搜索API

现在，让我们以一些简单的搜索来开始。有两种基本的方式来运行搜索：一种是在REST请求的URI中发送搜索参数，另一种是将搜索参数发送到REST请求 体中。请求体方法的表达能力更好，并且你可以使用更加可读的JSON格式来定义搜索。我们将尝试使用一次请求URI作为例子，但是教程的后面部分，我们将 仅仅使用请求体方法。

搜索的REST  API可以通过_search端点来访问。下面这个例子返回bank索引中的所有的文档：

```
curl 'localhost:9200/bank/_search?q=*&pretty'
```

我们仔细研究一下这个查询调用。我们在bank索引中搜索（_search端点），并且q=*参数指示Elasticsearch去匹配这个索引中所有的文档。pretty参数，和以前一样，仅仅是告诉Elasticsearch返回美观的JSON结果。

以下是响应（部分列出）：

```
curl 'localhost:9200/bank/_search?q=*&pretty'
{
  "took" : 63,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 1000,
    "max_score" : 1.0,
    "hits" : [ {
      "_index" : "bank",
      "_type" : "account",
      "_id" : "1",
      "_score" : 1.0, "_source" :
{"account_number":1,"balance":39225,"firstname":"Amber","lastname":"Duke","age":32,"gender":"M","address":"880 Holmes Lane","employer":"Pyrami","email":"amberduke@pyrami.com","city":"Brogan","state":"IL"}
    }, {
      "_index" : "bank",
      "_type" : "account",
      "_id" : "6",
      "_score" : 1.0, "_source" :
{"account_number":6,"balance":5686,"firstname":"Hattie","lastname":"Bond","age":36,"gender":"M","address":"671 Bristol Street","employer":"Netagy","email":"hattiebond@netagy.com","city":"Dante","state":"TN"}
    }, {
      "_index" : "bank",
      "_type" : "account",
```

对于这个响应，我们看到了以下的部分：
- took —— Elasticsearch执行这个搜索的耗时，以毫秒为单位
- timed_out —— 指明这个搜索是否超时
- _shards —— 指出多少个分片被搜索了，同时也指出了成功/失败的被搜索的shards的数量
- hits —— 搜索结果
- hits.total —— 能够匹配我们查询标准的文档的总数目
- hits.hits —— 真正的搜索结果数据（默认只显示前10个文档）
- _score和max_score —— 现在先忽略这些字段

使用请求体方法的等价搜索是：

```
curl -XPOST 'localhost:9200/bank/_search?pretty' -d '
{
  "query": { "match_all": {} }
}'
```

这里的不同之处在于，并不是向URI中传递q=*，取而代之的是，我们在_search API的请求体中POST了一个JSON格式请求体。我们将在下一部分中讨论这个JSON查询。

响应是：

```
curl -XPOST 'localhost:9200/bank/_search?pretty' -d '
{
  "query": { "match_all": {} }
}'
{
```

```
  "took" : 26,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 1000,
    "max_score" : 1.0,
    "hits" : [ {
      "_index" : "bank",
      "_type" : "account",
      "_id" : "1",
      "_score" : 1.0, "_source" :
{"account_number":1,"balance":39225,"firstname":"Amber","lastname":"Duke","age":32,"gender":"M","address":"880 Holmes
Lane","employer":"Pyrami","email":"amberduke@pyrami.com","city":"Brogan","state":"IL"}
    }, {
      "_index" : "bank",
      "_type" : "account",
      "_id" : "6",
      "_score" : 1.0, "_source" :
{"account_number":6,"balance":5686,"firstname":"Hattie","lastname":"Bond","age":36,"gender":"M","address":"671 Bristol
Street","employer":"Netagy","email":"hattiebond@netagy.com","city":"Dante","state":"TN"}
    }, {
      "_index" : "bank",
      "_type" : "account",
      "_id" : "13",
```


有一点需要重点理解一下，一旦你取回了你的搜索结果，Elasticsearch就完成了使命，它不会维护任何服务器端的资源或者在你的结果中打开游标。 这是和其它类似SQL的平台的一个鲜明的对比， 在那些平台上，你可以在前面先获取你查询结果的一部分，然后如果你想获取结果的剩余部分，你必须继续返回服务端去取，这个过程使用一种有状态的服务器端游 标技术。

介绍查询语言

Elasticsearch提供一种JSON风格的特定领域语言，利用它你可以执行查询。这杯称为查询DSL。这个查询语言相当全面，第一眼看上去可能有些咄咄逼人，但是最好的学习方法就是以几个基础的例子来开始。

回到我们上一个例子，我们执行了这个查询：

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

分解以上的这个查询，其中的query部分告诉我查询的定义，match_all部分就是我们想要运行的查询的类型。match_all查询，就是简单地查询一个指定索引下的所有的文档。

除了这个query参数之外，我们也可以通过传递其它的参数来影响搜索结果。比如，下面做了一次match_all并只返回第一个文档：

```
curl -XPOST 'localhost:9200/bank/_search?pretty' -d '
{
  "query": { "match_all": {} },
  "size": 1
}'
```

注意，如果没有指定size的值，那么它默认就是10。

下面的例子，做了一次match_all并且返回第11到第20个文档：

```
curl -XPOST 'localhost:9200/bank/_search?pretty' -d '
{
  "query": { "match_all": {} },
  "from": 10,
  "size": 10
}'
```

其中的from参数（0-based）从哪个文档开始，size参数指明从from参数开始，要返回多少个文档。这个特性对于搜索结果分页来说非常有帮助。注意，如果不指定from的值，它默认就是0。

下面这个例子做了一次match_all并且以账户余额降序排序，最后返前十个文档：

```
curl -XPOST 'localhost:9200/bank/_search?pretty' -d '
{
  "query": { "match_all": {} },
  "sort": { "balance": { "order": "desc" } }
}'
```

执行搜索

现在我们已经知道了几个基本的参数，让我们进一步发掘查询语言吧。首先我们看一下返回文档的字段。默认情况下，是返回完整的JSON文档的。这可以通过 source来引用（搜索hits中的_sourcei字段）。如果我们不想返回完整的源文档，我们可以指定返回的几个字

段。

下面这个例子说明了怎样返回两个字段account_number和balance（当然，这两个字段都是指_source中的字段），以下是具体的搜索：

```
curl -XPOST 'localhost:9200/bank/_search?pretty' -d '
{
  "query": { "match_all": {} },
  "_source": ["account_number", "balance"]
}'
```

注意到上面的例子仅仅是简化了_source字段。它仍将会返回一个叫做_source的字段，但是仅仅包含account_number和balance来年改革字段。

如果你有SQL背景，上述查询在概念上有些像SQL的SELECT FROM。

现在让我们进入到查询部分。之前，我们看到了match_all查询是怎样匹配到所有的文档的。现在我们介绍一种新的查询，叫做match查询，这可以看成是一个简单的字段搜索查询（比如对应于某个或某些特定字段的搜索）。

下面这个例子返回账户编号为20的文档：

```
curl -XPOST 'localhost:9200/bank/_search?pretty' -d '
{
  "query": { "match": { "account_number": 20 } }
}'
```

下面这个例子返回地址中包含“mill”的所有账户：

```
curl -XPOST 'localhost:9200/bank/_search?pretty' -d '
{
  "query": { "match": { "address": "mill" } }
}'
```

下面这个例子返回地址中包含“mill”或者包含“lane”的账户：

```
curl -XPOST 'localhost:9200/bank/_search?pretty' -d '
{
  "query": { "match": { "address": "mill lane" } }
}'
```

下面这个例子是match的变体（match_phrase），它会去匹配短语“mill lane”：

```
curl -XPOST 'localhost:9200/bank/_search?pretty' -d '
{
  "query": { "match_phrase": { "address": "mill lane" } }
}'
```

现在，让我们介绍一下布尔查询。布尔查询允许我们利用布尔逻辑将较小的查询组合成较大的查询。

现在这个例子组合了两个match查询，这个组合查询返回包含“mill”和“lane”的所有的账户：

```
curl -XPOST 'localhost:9200/bank/_search?pretty' -d '
{
  "query": {
    "bool": {
      "must": [
        { "match": { "address": "mill" } },
        { "match": { "address": "lane" } }
      ]
    }
  }
}'
```

在上面的例子中，bool must语句指明了，对于一个文档，所有的查询都必须为真，这个文档才能够匹配成功。

相反的，下面的例子组合了两个match查询，它返回的是地址中包含“mill”或者“lane”的所有的账户:

```
curl -XPOST 'localhost:9200/bank/_search?pretty' -d '
{
  "query": {
    "bool": {
      "should": [
        { "match": { "address": "mill" } },
        { "match": { "address": "lane" } }
      ]
    }
  }
}'
```

在上面的例子中，bool should语句指明，对于一个文档，查询列表中，只要有一个查询匹配，那么这个文档就被看成是匹配的。

现在这个例子组合了两个查询，它返回地址中既不包含“mill”，同时也不包含“lane”的所有的账户信息：

```
curl -XPOST 'localhost:9200/bank/_search?pretty' -d '
{
```

```
  "query": {
    "bool": {
      "must_not": [
        { "match": { "address": "mill" } },
        { "match": { "address": "lane" } }
      ]
    }
  }
}'
```

在上面的例子中， bool must_not语句指明，对于一个文档，查询列表中的的所有查询都必须都不为真，这个文档才被认为是匹配的。

我们可以在一个bool查询里一起使用must、should、must_not。此外，我们可以将bool查询放到这样的bool语句中来模拟复杂的、多等级的布尔逻辑。

下面这个例子返回40岁以上并且不生活在ID（daho）的人的账户：

```
curl -XPOST 'localhost:9200/bank/_search?pretty' -d '
{
  "query": {
    "bool": {
      "must": [
        { "match": { "age": "40" } }
      ],
      "must_not": [
        { "match": { "state": "ID" } }
      ]
    }
  }
}'
```

执行过滤器

在先前的章节中，我们跳过了文档得分的细节（搜索结果中的_score字段）。这个得分是与我们指定的搜索查询匹配程度的一个相对度量。得分越高，文档越相关，得分越低文档的相关度越低。

Elasticsearch中的所有的查询都会触发相关度得分的计算。对于那些我们不需要相关度得分的场景下，Elasticsearch以过滤器的形式 提供了另一种查询功能。过滤器在概念上类似于查询，但是它们有非常快的执行速度，这种快的执行速度主要有以下两个原因

- 过滤器不会计算相关度的得分，所以它们在计算上更快一些
- 过滤器可以被缓存到内存中，这使得在重复的搜索查询上，其要比相应的查询快出许多。

为了理解过滤器，我们先来介绍“被过滤”的查询，这使得你可以将一个查询（像是match_all，match，bool等）和一个过滤器结合起来。作为一个例子，我们介绍一下范围过滤器，它允许我们通过一个区间的值来过滤文档。这通常被用在数字和日期的过滤上。

这个例子使用一个被过滤的查询，其返回值是越在20000到30000之间（闭区间）的账户。换句话说，我们想要找到越大于等于20000并且小于等于30000的账户。

```
curl -XPOST 'localhost:9200/bank/_search?pretty' -d '
{
  "query": {
    "filtered": {
      "query": { "match_all": {} },
      "filter": {
        "range": {
          "balance": {
            "gte": 20000,
            "lte": 30000
          }
        }
      }
    }
  }
}'
```

分解上面的例子，被过滤的查询包含一个match_all查询（查询部分）和一个过滤器（filter部分）。我们可以在查询部分中放入其他查询，在 filter部分放入其它过滤器。在上面的应用场景中，由于所有的在这个范围之内的文档都是平等的（或者说相关度都是一样的），没有一个文档比另一个文档 更相关，所以这个时候使用范围过滤器就非常合适了。

通常情况下，要决定是使用过滤器还是使用查询，你就需要问自己是否需要相关度得分。如果相关度是不重要的，使用过滤器，否则使用查询。如果你有SQL背 景，查询和过滤器在概念上类似于SELECT WHERE语句， although more so for filters than queries。

除了match_all, match, bool,filtered和range查询，还有很多其它类型的查uxn/过滤器，我们这里不会涉及。由于我们已经对它们的工作原理有了基本的理解，将其应用到其它类型的查询、过滤器上也不是件难事。

执行聚合

聚合提供了分组并统计数据的能力。理解聚合的最简单的方式是将其粗略地等同为SQL的GROUP BY和SQL聚合函数。在Elasticsearch中，你可以在一个响应中同时返回命中的数据和聚合结果。你可以使用简单的API同时运行查询和多个聚 合，并以一次返回，这避免了来回的网络通信，这是非常强大和高效的。

作为开始的一个例子，我们按照state分组，按照州名的计数倒序排序：

```
curl -XPOST 'localhost:9200/bank/_search?pretty' -d '
{
  "size": 0,
  "aggs": {
    "group_by_state": {
      "terms": {
        "field": "state"
      }
    }
  }
}'
```

在SQL中，上面的聚合在概念上类似于：
SELECT COUNT(*) from bank GROUP BY state ORDER BY COUNT(*) DESC

响应（其中一部分）是：

```
"hits" : {
    "total" : 1000,
    "max_score" : 0.0,
    "hits" : [ ]
  },
  "aggregations" : {
    "group_by_state" : {
      "buckets" : [ {
        "key" : "al",
        "doc_count" : 21
      }, {
        "key" : "tx",
        "doc_count" : 17
      }, {
        "key" : "id",
        "doc_count" : 15
      }, {
        "key" : "ma",
        "doc_count" : 15
      }, {
        "key" : "md",
        "doc_count" : 15
      }, {
        "key" : "pa",
        "doc_count" : 15
      }, {
        "key" : "dc",
        "doc_count" : 14
      }, {
        "key" : "me",
        "doc_count" : 14
      }, {
        "key" : "mo",
        "doc_count" : 14
      }, {
        "key" : "nd",
        "doc_count" : 14
      } ]
    }
  }
}
```

我们可以看到AL（abama）有21个账户，TX有17个账户，ID（daho）有15个账户，依此类推。

注意我们将size设置成0，这样我们就可以只看到聚合结果了，而不会显示命中的结果。

在先前聚合的基础上，现在这个例子计算了每个州的账户的平均余额（还是按照账户数量倒序排序的前10个州）：

```
curl -XPOST 'localhost:9200/bank/_search?pretty' -d '
{
  "size": 0,
  "aggs": {
    "group_by_state": {
      "terms": {
        "field": "state"
      },
      "aggs": {
        "average_balance": {
          "avg": {
            "field": "balance"
          }
        }
      }
    }
```

```
  }
}'
```

注意，我们把average_balance聚合嵌套在了group_by_state聚合之中。这是所有聚合的一个常用模式。你可以任意的聚合之中嵌套聚合，这样你就可以从你的数据中抽取出想要的概述。

基于前面的聚合，现在让我们按照平均余额进行排序：

```
curl -XPOST 'localhost:9200/bank/_search?pretty' -d '
{
  "size": 0,
  "aggs": {
    "group_by_state": {
      "terms": {
        "field": "state",
        "order": {
          "average_balance": "desc"
        }
      },
      "aggs": {
        "average_balance": {
          "avg": {
            "field": "balance"
          }
        }
      }
    }
  }
}'
```

下面的例子显示了如何使用年龄段（20-29，30-39，40-49）分组，然后在用性别分组，然后为每一个年龄段的每一个性别计算平均账户余额：

```
curl -XPOST 'localhost:9200/bank/_search?pretty' -d '
{
  "size": 0,
  "aggs": {
    "group_by_age": {
      "range": {
        "field": "age",
        "ranges": [
          {
            "from": 20,
            "to": 30
          },
          {
            "from": 30,
            "to": 40
          },
          {
            "from": 40,
            "to": 50
          }
        ]
      },
      "aggs": {
        "group_by_gender": {
          "terms": {
            "field": "gender"
          },
          "aggs": {
            "average_balance": {
              "avg": {
                "field": "balance"
              }
            }
          }
        }
      }
    }
  }
}'
```

有很多关于聚合的细节，我们没有涉及。如果你想做更进一步的实验，http://www.elasticsearch.org/guide/en/elasticsearch/reference/current/search-aggregations.html是一个非常好的起点。

## 总结

Elasticsearch既是一个简单的产品，也是一个复杂的产品。我们现在已经学习到了基础部分，它的一些原理，以及怎样用REST API来做一些工作。我希望这个教程已经使你对Elasticsearch是什么有了一个更好的理解，跟重要的是，能够激发你继续实验Elasticsearch的其它特性。