# elasticsearch中文指南

endymecy

Published
with GitBook

# Table of Contents

# elasticsearch-guide

Elasticsearch是一个高可扩展的、开源的全文本搜索和分析工具。它允许你以近实时的方式快速存储、搜索、分析大容量的数据。

# 基本概念

Elasticsearch有几个核心概念。开始学习Elasticsearch之前理解这些概念会对整个学习过程有很大的帮助。

## 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很小的延迟（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， 它们共同持有你全部的数据， 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”。 这个名字很重要， 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。在生产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。

注意，一个集群中只包含一个节点是合法的。另外，你也可以拥有多个集群，集群以名字区分。

## 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 `Marvel` 角色的名字，这个名字会在节点启动时分配给它。这个名字对于管理工作来说很重要，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”的单节点集群。

## 索引（index）

一个索引就是一个拥有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，你能够创建任意多个索引。

## 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，会为具有一组相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、某个订单的一个文档。文档以JSON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，你可以存储任意多的文档。注意，一个文档物理上存在于一个索引之中，但文档必须被索引/赋予一个索引的type。

# 分片和复制（**shards and 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，而任一节点可能没有这样大的磁盘空间来存储或者单个节点处理搜索请求，响应会太慢。

为了解决这个问题，Elasticsearch提供了将索引划分成多片的能力，这些片叫做分片。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（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/节点因为某些原因处于离线状态或者消失的情况下，故障转移机制是非常有用且强烈推荐的。为此， 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，复制提供了高可用性。复制分片不与原/主要分片置于同一节点上是非常重要的。
- 因为搜索可以在所有的复制上并行运行，复制可以扩展你的搜索量/吞吐量

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（即没有复制） 或多次。一旦复制了，每个索引就有了主分片（作为复制源的分片）和复制分片（主分片的拷贝）。 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，但是你不能再改变分片的数量。

默认情况下，Elasticsearch中的每个索引分配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），这样每个索引总共就有10个分片。

# 安装

Elasticsearch需要Java 7。在本文写作的时候，推荐使用Oracle JDK 1.8.0_25 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涉及太多细节。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。

用下面的命令下载 Elasticsearch 1.4.2 tar包

```
curl -L -O https://download.elasticsearch.org/elasticsearch/elasticsearch/elasticsearch-1
```

将其解压并进入bin目录， 启动你的节点和单节点集群

```
tar -xvf elasticsearch-1.4.2.tar.gz
cd elasticsearch-1.4.2/bin
./elasticsearch
```

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
```

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。

# 操作集群

## rest 接口

现在我们已经有一个正常运行的节点（和集群），下一步就是要去理解怎样与其通信。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：

- 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(cluster health)

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。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，来使用该功能。 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 unass
1394735289 14:28:09  elasticsearch green           1         1      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。

当我们查看集群状态的时候，我们可能得到绿色、黄色或红色三种状态。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。

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， 你就已经把它们加入到集群中了。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， 我们可以看到叫“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'
```

对应的响应是:

```
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

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。

需要注意的是，当你想将文档索引到某个索引的时候，Elasticsearch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found字段-（指明我们找到了一个ID为1的文档）和_source字段（返回我们前一步中索引的完整JSON

文档）之外，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。

我们细看一下我们学过的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'
```

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 -X<REST Verb> <Node>:<Port>/<Index>/<Type>/<ID>
```

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。这和其它的SQL平台不同，它们的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"
}'
```

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则不会受到影响。

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

以上的命令，将会为一个ID为2的文档建立索引。

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。

下面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，然后再索引更新的新文档。

下面的例子展示了怎样将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`指向当前被更新的文档。

注意，目前的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

删除文档是非常直观的。以下的例子展示了怎样删除ID为2的文档：

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

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，删除满足请求体中的查询条件的索引。我们仍然使用DELETE动词。

## 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也提供了操作的批量处理功能，它通过使用_bulk API实现。这个功能之所以重要， 是因为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，它将会继续处理后面的动作。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。

# 操作数据

## 样本数据集

现在我们对于基本的东西已经有了一些认识，现在让我们尝试使用一些更加贴近现实的数据集。我准备了一些假想的客户银行账户信息的JSON文档样本。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"
}
```

可以通过www.json-generator.com/自动生成这些数据。

## 载入样本数据

你可以在这里下载样本数据集。将其解压到当前目录下并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（在account类型下）。

## 搜索API

现在，让我们以一些简单的搜索来开始学习。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终点(endpoint)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`终点），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"
    }, {
      "_index" : "bank",
      "_type" : "account",
      "_id" : "6",
      "_score" : 1.0, "_source" : {"account_number":6,"balance":5686,"firstname":"Hattie"
    }, {
      "_index" : "bank",
      "_type" : "account",
```

对于这个响应，我们可以看到如下的部分：

- `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`：匹配查询条件的文档的总数目
- `hits.hits`：真正的搜索结果数组（默认是前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": {} }
}'
```

与上面方法不同之处在于，并不是向URI中传递q=*，取而代之的是在_search API的请求体中POST了一个JSON格式的请求体。我们将在下一部分中讨论这个JSON查询。

有一点需要重点理解的是，一旦你取回了搜索结果，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子集， 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，这个数据集使用了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。这中语言称为DSL。这个查询语言非常全面，最好的学习方法就是以几个基础的例子来开始。

回到上一个例子，我们执行了这个查询：

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

分析以上的这个查询，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` 查询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` 查询并且返回第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 参数指明从哪个文档开始，size参数指明从from参数开始，要返回的文档数。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` 查询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，让我们进一步学习查询语言。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从搜索中只返回两个字段 `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` 两个字段。

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，我们学习了 `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词语(term)“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词语“mill” 或者“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、多层级的布尔逻辑。

下面这个例子返回40岁以上并且不生活在ID（aho）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

在前面的章节中，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（如 `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，其返回值是存款在20000到30000之间（闭区间)的所有账户。换句话说，我们想要找到存款大于等于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
          }
        }
      }
    }
```

```
  }
}'
```

分析上面的例子，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，一般情况下过滤器比查询用得更多。

除了 `match_all` , `match` , `bool` , `filtered` 和 `range` 查询，还有很多其它类型的查询/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一次返回，这避免了来回的网络通信，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，并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（aho）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存款（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。你可以在任意的聚合之中嵌套聚合，这样就可以从你的数据中抽取出想要的结果。

在前面的聚合的基础上，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，然后再用性别分组，最后为每一个年龄段的每组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"
              }
            }
          }
        }
      }
    }
  }
}'
```

# 文档APIs

- 索引API

# 搜索APIS

## 路由

当执行一个搜索，它将会广播到所有的索引/索引分片。被搜索的分片可以通过提供`routing`参数来控制。例如当索引推文时，用户名就可以作为路由的值。

```
$ curl -XPOST 'http://localhost:9200/twitter/tweet?routing=kimchy' -d '{
    "user" : "kimchy",
    "postDate" : "2009-11-15T14:12:12",
    "message" : "trying out Elasticsearch"
}
'
```

在这种情况下，如果你仅仅想查询特定用户的推文，我们可以将其指定为路由，从而在搜索时只命中相关的分片。

```
$ curl -XGET 'http://localhost:9200/twitter/tweet/_search?routing=kimchy' -d '{
    "query": {
        "filtered" : {
            "query" : {
                "query_string" : {
                    "query" : "some query string here"
                }
            },
            "filter" : {
                "term" : { "user" : "kimchy" }
            }
        }
    }
}
'
```

路由参数可以由逗号分隔的多个字符串表示，这将使查询命中与路由参数值相匹配的相关分片。

## 统计组

一个搜索可以与统计组相关联，每一个组都包含一个统计集合，它可以通过索引统计API在以后重新获取。例如，下面是一个与两个分组相关联的搜索的搜索体。

```
{
    "query" : {
        "match_all" : {}
    },
    "stats" : ["group1", "group2"]
}
```

## 目录

- 搜索
- URI搜索
- 请求体(request body)搜索
- 搜索模板
- 搜索分片API
- 聚合(aggregations)
- facets
- 启发者(suggesters)
- 多搜索API
- 计数API
- 搜索存在(search exist)API
- 验证API
- 解释API
- 过滤器(percolator)
- more like this api

# 搜索API

搜索API允许开发者执行搜索查询，返回匹配查询的搜索结果。这既可以通过查询字符串也可以通过查询体实现。

## 多索引多类型

所有的搜索API都可以跨多个类型使用，也可以通过多索引语法跨索引使用。例如，我们可以搜索twitter索引的跨类型的所有文档。

```
$ curl -XGET 'http://localhost:9200/twitter/_search?q=user:kimchy'
```

我们也可以带上特定的类型:

```
$ curl -XGET 'http://localhost:9200/twitter/tweet,user/_search?q=user:kimchy'
```

我们也可以搜索跨多个索引的所有文档

```
$ curl -XGET 'http://localhost:9200/kimchy,elasticsearch/tweet/_search?q=tag:wow'
```

或者我们也可以用`_all`占位符表示搜索所有可用的索引的所有推特。

```
$ curl -XGET 'http://localhost:9200/_all/tweet/_search?q=tag:wow'
```

或者搜索跨所有可用索引和所有可用类型的推特

```
$ curl -XGET 'http://localhost:9200/_search?q=tag:wow'
```

# uri搜索

一个搜索可以用纯粹的uri来执行查询。在这种模式下使用搜索，并不是所有的选项都是暴露的。它可以方便快速进行 `curl 测试` 。

```
$ curl -XGET 'http://localhost:9200/twitter/tweet/_search?q=user:kimchy'
```

# 参数

| Name | Description |
| --- | --- |
| `q` | 表示查询字符串 |
| `df` | 在查询中，当没有定义字段的前缀的情况下的默认字段前缀 |
| `analyzer` | 当分析查询字符串时，分析器的名字 |
| `default_operator` | 被用到的默认操作，有 `AND` 和 `OR` 两种，默认是 `OR` |
| `explain` | 对于每一个命中(hit)，对怎样得到命中得分的计算给出一个解释 |
| `_source` | 将其设置为false，查询就会放弃检索 `_source` 字段。你也可以通过设置 `_source_include` 和 `_source_exclude` 检索部分文档 |
| `fields` | 命中的文档返回的字段 |
| `sort` | 排序执行。可以以 `fieldName` 、 `fieldName:asc` 或者 `fieldName:desc` 的格式设置。 `fieldName` 既可以是存在的字段，也可以是 `_score` 字段。可以有多个sort参数 |
| `track_scores` | 当排序的时候，将其设置为true，可以返回相关度得分 |
| `timeout` | 默认没有timeout |
| `from` | 默认是0 |
| `size` | 默认是10 |
| `search_type` | 搜索操作执行的类型，有 `dfs_query_then_fetch` , `dfs_query_and_fetch` , `query_then_fetch` , `query_and_fetch` , `count` , `scan` 几种，默认是 `query_then_fetch` |
| `lowercase_expanded_terms` | terms是否自动小写，默认是true |
| `analyze_wildcard` | 是否分配通配符和前缀查询，默认是false |
| `terminate_after` | The maximum number of documents to collect for each shard, upon reaching which the query execution will terminate early. If set, the response will have a boolean field terminated_early to indicate whether the query execution has actually terminated_early. Defaults to no terminate_after. |

# 请求体搜索

有搜索DSL的搜索请求可以被执行。这些DSL包含在请求的请求体中。

```
$ curl -XGET 'http://localhost:9200/twitter/tweet/_search' -d '{
    "query" : {
        "term" : { "user" : "kimchy" }
    }
}
'
```

## 1 参数

| Name | Description |
| --- | --- |
| `timeout` | 默认没有timeout |
| `from` | 默认是0 |
| `size` | 默认是10 |
| `search_type` | 搜索操作执行的类型，有 `dfs_query_then_fetch`，`dfs_query_and_fetch`，`query_then_fetch`，`query_and_fetch`，`count`，`scan` 几种，默认是 `query_then_fetch` |
| `query_cache` | 当 `?search_type=count` 时，查询结果是否缓存 |
| `terminate_after` | The maximum number of documents to collect for each shard, upon reaching which the query execution will terminate early. If set, the response will have a boolean field terminated_early to indicate whether the query execution has actually terminated_early. Defaults to no terminate_after. |

`search_type` 和 `query_cache` 必须通过查询参数字符串传递。

`HTTP GET` 和 `HTTP POST` 都可以用来执行带有请求体的搜索。

## 2 查询

在搜索请求体中的查询元素允许用查询DSL定义一个查询

```
{
    "query" : {
        "term" : { "user" : "kimchy" }
    }
}
```

## 3 from / size

可以用from和size参数对结果进行分页。from表示你想获得的第一个结果的偏移量，size表示你想获得的结果的个数。from默认是0，size默认是10.

```
{
    "from" : 0, "size" : 10,
```

```
    "query" : {
        "term" : { "user" : "kimchy" }
    }
}
```

# 4 排序

一个特定的字段运行添加一个或者多个排序。排序定义在字段级别，特定的字段名`_score`是通过得分排序。

```
{
    "sort" : [
        { "post_date" : {"order" : "asc"}},
        "user",
        { "name" : "desc" },
        { "age" : "desc" },
        "_score"
    ],
    "query" : {
        "term" : { "user" : "kimchy" }
    }
}
```

## 4.1 排序值

### 4.1.1 排序选项

elasticsearch支持通过数组或者多值字段排序。`mode`选项控制获取排序文档的什么数组值来进行排序。`mode`选项有如下几种

- min：挑选最低的值
- max: 挑选最高的值
- sum：挑选所有值的和作为排序的值，仅用于数字
- avg：挑选所有值得平均作为排序的值，仅用于数字

### 4.1.2 例子

下面的例子按照文档的平均价格的升序进行排列

```
curl -XPOST 'localhost:9200/_search' -d '{
   "query" : {
    ...
   },
   "sort" : [
      {"price" : {"order" : "asc", "mode" : "avg"}}
   ]
}'
```

## 4.2 带有嵌套对象的排序

Elasticsearch支持字段中带有嵌套对象的排序，嵌套的字段排序在其它排序选项存在的基础上支持下面的

参数

- nested_path：Defines the on what nested object to sort. The actual sort field must be a direct field inside this nested object. The default is to use the most immediate inherited nested object from the sort field.
- nested_filter：A filter the inner objects inside the nested path should match with in order for its field values to be taken into account by sorting. Common case is to repeat the query / filter inside the nested filter or query. By default no nested_filter is active.

### 4.2.1 例子

在下面的例子中，`offer`是一个嵌套类型的字段。

```
curl -XPOST 'localhost:9200/_search' -d '{
   "query" : {
    ...
   },
   "sort" : [
       {
          "offer.price" : {
             "mode" :  "avg",
             "order" : "asc",
             "nested_filter" : {
                "term" : { "offer.color" : "blue" }
             }
          }
       }
    ]
}'
```

## 4.3 missing值

`missing`参数指定缺失字段的文档的处理方式：将`missing`值设置为`_last`，`_first`或者自定义值

```
{
    "sort" : [
        { "price" : {"missing" : "_last"} },
    ],
    "query" : {
        "term" : { "user" : "kimchy" }
    }
}
```

## 4.4 ignoring unmapped字段

默认情况下，如果没有与字段相关联的映射，搜索请求将会失败。`unmapped_type`选项允许忽略没有映射的字段，不用它们排序。这个参数的值指定哪些排序值可以忽略。

```
{
    "sort" : [
        { "price" : {"unmapped_type" : "long"} },
    ],
    "query" : {
```

```
        "term" : { "user" : "kimchy" }
    }
}
```

如果任何索引都没有price字段的映射，那么elasticsearch将会处理它，就好像有一个long类型的映射一样。

## 4.5 地理距离排序

通过 `_geo_distance` 排序。

```
{
    "sort" : [
        {
            "_geo_distance" : {
                "pin.location" : [-70, 40],
                "order" : "asc",
                "unit" : "km",
                "mode" : "min",
                "distance_type" : "sloppy_arc"
            }
        }
    ],
    "query" : {
        "term" : { "user" : "kimchy" }
    }
}
```

- distance_type：怎样计算距离可以有 `sloppy_arc` (默认), `arc` （更精确但是显著变慢）, `plane` (最快)

地理距离排序支持的排序 `mode` 有 `max`， `min` 和 `avg` 。

## 4.6 基于脚本的排序

```
{
    "query" : {
        ....
    },
    "sort" : {
        "_script" : {
            "script" : "doc['field_name'].value * factor",
            "type" : "number",
            "params" : {
                "factor" : 1.1
            },
            "order" : "asc"
        }
    }
}
```

## 4.7 追踪得分

当基于一个字段进行排序是，默认不计算score，通过设置track_scores为true，可以计算得分并且追踪。

```
{
    "track_scores": true,
    "sort" : [
        { "post_date" : {"reverse" : true} },
        { "name" : "desc" },
        { "age" : "desc" }
    ],
    "query" : {
        "term" : { "user" : "kimchy" }
    }
}
```

## 5 source过滤

用于控制 `_source` 字段的返回。默认情况下，操作返回 `_source` 字段的内容，除非你用到了 `fields` 参数，或者 `_source` 被禁用了。你能够通过 `_source` 参数关掉 `_source` 检索。

```
{
    "_source": false,
    "query" : {
        "term" : { "user" : "kimchy" }
    }
}
```

`_source` 也接受一个或者多个通配符模式控制返回值。

```
{
    "_source": "obj.*",
    "query" : {
        "term" : { "user" : "kimchy" }
    }
}
or
{
    "_source": [ "obj1.*", "obj2.*" ],
    "query" : {
        "term" : { "user" : "kimchy" }
    }
}
```

最后，对于完整的控制，我们可以包含include和exclude。

```
{
    "_source": {
        "include": [ "obj1.*", "obj2.*" ],
        "exclude": [ "*.description" ],
    }
    "query" : {
        "term" : { "user" : "kimchy" }
    }
}
```

# 6 字段

允许选择性地加载文档特定的存储字段。

```
{
    "fields" : ["user", "postDate"],
    "query" : {
        "term" : { "user" : "kimchy" }
    }
}
```

`*` 可以被用来加载文档中的所有字段。

如果 `fields` 数组为空，那么就只会返回 `_id` 和 `_type` 字段。

```
{
    "fields" : [],
    "query" : {
        "term" : { "user" : "kimchy" }
    }
}
```

为了向后兼容，如果 `fields` 参数指定的字段在文档中不存在，它将会加载 `_source`，并从中抽取这个字段。这个功能已经被source过滤器替代。

从文档中获取的字段值总以数组的方式返回。但是源数据字段如 `_routing` 和 `_parent` 却从不以数组的方式返回。

只有叶子字段可以通过 `field` 选项返回。所以对象字段不能被返回，这样的操作会报错。

## partial

从 `_source` 加载数据时，partial字段可以用来使用通配符来控制哪部分的 `_source` 将被加载。例如

```
{
    "query" : {
        "match_all" : {}
    },
    "partial_fields" : {
        "partial1" : {
            "include" : "obj1.obj2.*",
        }
    }
}
```

或者

```
{
    "query" : {
        "match_all" : {}
    },
    "partial_fields" : {
```

```
        "partial1" : {
            "include" : "obj1.obj2.*",
            "exclude" : "obj1.obj3.*"
        }
    }
}
```

include和exclude都支持多模式

```
{
    "query" : {
        "match_all" : {}
    },
    "partial_fields" : {
        "partial1" : {
            "include" : ["obj1.obj2.*", "obj1.obj4.*"],
            "exclude" : "obj1.obj3.*"
        }
    }
}
```

## script fields

```
{
    "query" : {
        ...
    },
    "script_fields" : {
        "test1" : {
            "script" : "doc['my_field_name'].value * 2"
        },
        "test2" : {
            "script" : "doc['my_field_name'].value * factor",
            "params" : {
                "factor"  : 2.0
            }
        }
    }
}
```

script fields可以在没有保存的字段（如例子中的`my_field_name`）上工作，返回自定义的值（脚本算出的值）。

脚本也可以访问文档的`_source`字段，并抽取特定的元素（是一个对象类型）返回。

```
{
        "query" : {
            ...
        },
        "script_fields" : {
            "test1" : {
                "script" : "_source.obj1.obj2"
            }
        }
    }
```

了解 `doc['my_field'].value` 和 `_source.my_field` 之间的不同是很重要的。首先，使用doc关键字，会使相应的字段加载到内存，执行速度更快但是更耗费内存。第二， `doc[...]` 符号 仅允许简单的值字段，只在基于字段的非分析或者单个项上有意义。

另一方面， `_source` 加载、分析source，然后仅仅返回相关部分的json。

## field data fields

返回一个字段的字段数据表示，如下例

```
{
    "query" : {
        ...
    },
    "fielddata_fields" : ["test1", "test2"]
}
```

field data fields可以用于没有保存的字段。利用 `fielddata_fields` 参数会导致该字段的项加载到内存，增加内存的消耗。

## post filter

`post_filter` 在搜索查询的最后，在聚合操作已经被计算后，应用于搜索的 `hits` 。下面用一个例子来说明。

假设你正在卖衬衣，用户指定了两个过滤器： `color:red` 和 `brand:gucci` 。一般情况下，你可以用到 `filtered query`

```
curl -XGET localhost:9200/shirts/_search -d '
{
  "query": {
    "filtered": {
      "filter": {
        "bool": {
          "must": [
            { "term": { "color": "red"   }},
            { "term": { "brand": "gucci" }}
          ]
        }
      }
    }
  }
}
'
```

假设你有一个 `model` 字段允许用户限制它们的搜索结果为red Gucci `t-shirts` 或者 `dress-shirts` 。可以用 `terms aggregation` 。

```
curl -XGET localhost:9200/shirts/_search -d '
{
```

```
  "query": {
    "filtered": {
      "filter": {
        "bool": {
          "must": [
            { "term": { "color": "red"   }},
            { "term": { "brand": "gucci" }}
          ]
        }
      }
    }
  },
  "aggs": {
    "models": {
      "terms": { "field": "model" }
    }
  }
}
'
```

但是，也许你可能要告诉用户有多少其它颜色的Gucci shirts可以购买。如果你仅仅在 `color` 字段中加入 `terms` 聚合，那么你只会返回 `red` 的值，因为你的查询只返回红色的衬衣。

你想要在聚合中包含所有颜色的衬衣，然后只在搜索结果中应用 `colors` 过滤。可以用到 `post_filter`

```
curl -XGET localhost:9200/shirts/_search -d '
{
  "query": {
    "filtered": {
      "filter": {
1        { "term": { "brand": "gucci" }}
      }
    }
  },
  "aggs": {
    "colors": {
2      "terms": { "field": "color" },
    },
    "color_red": {
      "filter": {
3        "term": { "color": "red" }
      },
      "aggs": {
        "models": {
4          "terms": { "field": "model" }
        }
      }
    }
  },
5  "post_filter": {
    "term": { "color": "red" },
  }
}
'
```

第1点，查询所有的衬衣，不管它是什么颜色

第2点， `colors` 聚合返回流行颜色的衬衣

第3、4点， `color_red` 聚合利用子聚合 `models` 限制red Gucci shirt

第5点， `post_filter` 删除除了红色的其它颜色的结果

## search type

- query and fetch：参数是 `query_and_fetch` ,它在所有相关的分片上执行查询，返回结果。每个分片返回 `size` 个结果。因为每个分片返回 `size` 个结果，所以这个类型实际返回 `size` 乘以分片个数的结果。
- query then fetch：参数是 `query_then_fetch` ,查询也依赖于所有分片，但是只返回足够的信息(不是文档内容)。基于这个结果进行分类和排名，之后才访问相关分片的实际文档内容。 这个类型返回结果的实际个数是 `size` 。这是默认的类型，你不必指定一个特定的 `search_type` 。
- dfs query and fetch：参数是 `dfs_query_and_fetch` ,和 `query_and_fetch` 相似。除了初始scatter的阶段，这个阶段为了更精确的得分，计算分布式项频率。
- dfs_query_then_fetch：参数是 `dfs_query_then_fetch` ,和 `query_then_fetch` 相似。除了初始scatter的阶段，这个阶段为了更精确的得分，计算分布式项频率。
- count：参数是 `count` ,返回满足查询条件的 `hits` 的数量。
- scan：参数是 `scan` , `scan` 查询类型禁用排序，允许通过大型结果集非常有效的滚动（scrolling）。

## scroll

一个 `search` 查询返回一“页”的结果， `scroll` API可以用来检索大数量的结果(甚至所有结果)。它类似关系型数据库中的游标。

Scrolling并不用来作实时的用户查询，而是处理大数量的数据。

为了利用 `scrolling` ，初始的搜索请求应该在查询字符串中指定 `scroll` 参数，告诉Elasticsearch，需要保持 `搜索上下文` 存活多长时间。

```
curl -XGET 'localhost:9200/twitter/tweet/_search?scroll=1m' -d '
{
    "query": {
        "match" : {
            "title" : "elasticsearch"
        }
    }
}
'
```

上面的请求结果中包含一个 `scroll_id` ,它应该传递给 `scroll` API去检索下一批数据。

```
curl -XGET  'localhost:9200/_search/scroll?scroll=1m'   \
     -d       'c2Nhbjs2OzM0NDg1ODpzRlBLc0FXNlNyNm5JWUc1'
```

### 保持查询上下文存活

`scroll` 参数告诉Elasticsearch需要保持 `搜索上下文` 存活多长时间。它的值不需要存活足够长时间处理所有的数据，它只需要足够的时间处理前面的批结果数据。每一个 `scroll` 请求 设置了一个新的过期时间。

## clear scroll api

```
curl -XDELETE localhost:9200/_search/scroll \
     -d 'c2Nhbjs2OzM0NDg1ODpzRlBLc0FXNlNyNm5JWUc1'
```

```
curl -XDELETE localhost:9200/_search/scroll \
     -d 'c2Nhbjs2OzM0NDg1ODpzRlBLc0FXNlNyNm5JWUc1,aGVuRmV0Y2g7NTsxOnkxaDZ'
```

```
curl -XDELETE localhost:9200/_search/scroll/_all
```

# min_score

```
{
    "min_score": 0.5,
    "query" : {
        "term" : { "user" : "kimchy" }
    }
}
```

返回的文档的得分小于`min_score`。

# 搜索模板

# Java API

这节会介绍elasticsearch支持的Java API。所有的elasticsearch操作都使用Client对象执行。本质上，所有的操作都是并行执行的。

另外，Client中的操作有可能累积并通过Bulk执行。

## maven

Elasticsearch托管在Maven仓库中。例如，你可以在pom.xml中定义最新的版本。

```
<dependency>
    <groupId>org.elasticsearch</groupId>
    <artifactId>elasticsearch</artifactId>
    <version>${es.version}</version>
</dependency>
```

## 在jboss eap6模块中部署

Elasticsearch和Lucene类必须在相同的jboss模块中，你应该像如下方式定义`module.xml`。

```
<?xml version="1.0" encoding="UTF-8"?>
<module name="org.elasticsearch">
  <resources>
    <!-- Elasticsearch -->
    <resource-root path="elasticsearch-1.4.1.jar"/>
    <!-- Lucene -->
    <resource-root path="lucene-core-4.10.2.jar"/>
    <resource-root path="lucene-analyzers-common-4.10.2.jar"/>
    <resource-root path="lucene-queries-4.10.2.jar"/>
    <resource-root path="lucene-memory-4.10.2.jar"/>
    <resource-root path="lucene-highlighter-4.10.2.jar"/>
    <resource-root path="lucene-queryparser-4.10.2.jar"/>
    <resource-root path="lucene-sandbox-4.10.2.jar"/>
    <resource-root path="lucene-suggest-4.10.2.jar"/>
    <resource-root path="lucene-misc-4.10.2.jar"/>
    <resource-root path="lucene-join-4.10.2.jar"/>
    <resource-root path="lucene-grouping-4.10.2.jar"/>
    <resource-root path="lucene-spatial-4.10.2.jar"/>
    <resource-root path="lucene-expressions-4.10.2.jar"/>
    <!-- Insert other resources here -->
  </resources>
  <dependencies>
    <module name="sun.jdk" export="true" >
        <imports>
            <include path="sun/misc/Unsafe" />
        </imports>
    </module>
    <module name="org.apache.log4j"/>
    <module name="org.apache.commons.logging"/>
    <module name="javax.api"/>
  </dependencies>
</module>
```

# 客户端

有多个地方需要使用`Java client`：

- 在存在的集群中执行标准的index, get, delete和search
- 在集群中执行管理任务
- 当你要运行嵌套在你的应用程序中的Elasticsearch的时候或者当你要运行单元测试或者集合测试的时候，启动所有节点

获得一个Client是非常容易的，最通用的步骤如下所示：

- 创建一个嵌套的节点，充当集群的一个节点
- 从这个嵌套的节点请求一个Client

另外一种方式是创建一个`TransportClient`来连接集群。

重要提示： 客户端和集群端推荐使用相同的版本，如果版本不同，可能会出现一些不兼容的问题。

# 节点客户端

实例化一个节点的客户端是获得客户端的最简单的方式。这个Client可以执行elasticsearch相关的操作。

```
import static org.elasticsearch.node.NodeBuilder.*;

// on startup
Node node = nodeBuilder().node();
Client client = node.client();

// on shutdown
node.close();
```

当你启动一个`node`,它就加入了elasticsearch集群。你可以通过简单的设置`cluster.name`或者明确地使用`clusterName`方法拥有不同的集群。

你能够在你项目的`/src/main/resources/elasticsearch.yml`文件中定义`cluster.name`。只要`elasticsearch.yml`在classpath目录下面，你就能够用到它来启动你的节点。

```
cluster.name: yourclustername
```

或者通过java：

```
Node node = nodeBuilder().clusterName("yourclustername").node();
Client client = node.client();
```

利用Client的好处是，操作可以自动地路由到这些操作被执行的节点，而不需要执行双跳(double hop)。例如，索引操作将会在该操作最终存在的分片上执行。

当你启动了一个节点，最重要的决定是是否它将保有数据。大多数情况下，我们仅仅需要用到clients，而

不需要分片分配给它们。这可以通过设置 `node.data` 为false或者设置 `node.client` 为true来简单实现。

```
import static org.elasticsearch.node.NodeBuilder.*;

// on startup
Node node = nodeBuilder().client(true).node();
Client client = node.client();

// on shutdown
node.close();
```

另外一个通用的用处就是启动node，然后利用Client进行单元/集成测试。在这种情况下，我们应该启动一个“本地”node。这只是启动node的一个简单的设置。注意，"本地"表示运行在本地 JVM上。运行在同一个JVM上的两个本地服务可以彼此发现并组成一个集群。

```
import static org.elasticsearch.node.NodeBuilder.*;

// on startup
Node node = nodeBuilder().local(true).node();
Client client = node.client();

// on shutdown
node.close();
```

## 传输（transport）客户端

`TransportClient` 利用transport模块远程连接一个elasticsearch集群。它并不加入到集群中，只是简单的获得一个或者多个初始化的transport地址，并以轮询的方式与这些地址进行通信。

```
// on startup
Client client = new TransportClient()
        .addTransportAddress(new InetSocketTransportAddress("host1", 9300))
        .addTransportAddress(new InetSocketTransportAddress("host2", 9300));

// on shutdown
client.close();
```

注意，如果你有一个与 `elasticsearch` 集群不同的集群，你可以设置机器的名字。

```
Settings settings = ImmutableSettings.settingsBuilder()
        .put("cluster.name", "myClusterName").build();
Client client =    new TransportClient(settings);
//Add transport addresses and do something with the client...
```

你也可以用 `elasticsearch.yml` 文件来设置。

这个客户端可以嗅到集群的其它部分，并将它们加入到机器列表。为了开启该功能，设置 `client.transport.sniff` 为true。

```
Settings settings = ImmutableSettings.settingsBuilder()
```

```
        .put("client.transport.sniff", true).build();
TransportClient client = new TransportClient(settings);
```

其它的transport客户端设置有如下几个：

| Parameter | Description |
| --- | --- |
| client.transport.ignore_cluster_name | true：忽略连接节点的集群名验证 |
| client.transport.ping_timeout | ping一个节点的响应时间，默认是5s |
| client.transport.nodes_sampler_interval | sample/ping 节点的时间间隔，默认是5s |

# 索引API

索引API允许开发者索引类型化的JSON文档到一个特定的索引，使其可以被搜索。

# 生成JSON文档

有几种不同的方式生成JSON文档

- 利用`byte[]`或者作为一个`String`手动生成
- 利用一个`Map`将其自动转换为相应的JSON
- 利用第三方库如Jackson去序列化你的bean
- 利用内置的帮助函数XContentFactory.jsonBuilder()

## 手动生成

需要注意的是，要通过Date Format编码日期。

```
String json = "{" +
        "\"user\":\"kimchy\"," +
        "\"postDate\":\"2013-01-30\"," +
        "\"message\":\"trying out Elasticsearch\"" +
    "}";
```

## 使用map

```
Map<String, Object> json = new HashMap<String, Object>();
json.put("user","kimchy");
json.put("postDate",new Date());
json.put("message","trying out Elasticsearch");
```

## 序列化bean

elasticsearch早就用到了Jackson，把它放在了`org.elasticsearch.common.jackson`下面。你可以在你的`pom.xml`文件里面添加你自己的Jackson版本。

```
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>2.1.3</version>
</dependency>
```

这样，你就可以序列化你的bean为JSON。

```
import com.fasterxml.jackson.databind.*;

// instance a json mapper
ObjectMapper mapper = new ObjectMapper(); // create once, reuse
```

```
// generate json
String json = mapper.writeValueAsString(yourbeaninstance);
```

## 利用elasticsearch帮助类

elasticsearch提供了内置的帮助类来将数据转换为JSON

```
import static org.elasticsearch.common.xcontent.XContentFactory.*;

XContentBuilder builder = jsonBuilder()
    .startObject()
        .field("user", "kimchy")
        .field("postDate", new Date())
        .field("message", "trying out Elasticsearch")
    .endObject()
```

注意，你也可以使用 `startArray(String)` 和 `endArray()` 方法添加数组。另外， `field` 可以接收任何类型的对象，你可以直接传递数字、时间甚至XContentBuilder对象。

可以用下面的方法查看json。

```
String json = builder.string();
```

# 索引文档

下面的例子将JSON文档索引为一个名字为“twitter”，类型为“tweet”，id值为1的索引。

```
import static org.elasticsearch.common.xcontent.XContentFactory.*;

IndexResponse response = client.prepareIndex("twitter", "tweet", "1")
        .setSource(jsonBuilder()
                    .startObject()
                        .field("user", "kimchy")
                        .field("postDate", new Date())
                        .field("message", "trying out Elasticsearch")
                    .endObject()
                  )
        .execute()
        .actionGet();
```

你也可以不提供id:

```
String json = "{" +
        "\"user\":\"kimchy\"," +
        "\"postDate\":\"2013-01-30\"," +
        "\"message\":\"trying out Elasticsearch\"" +
    "}";

IndexResponse response = client.prepareIndex("twitter", "tweet")
        .setSource(json)
        .execute()
```

```
        .actionGet();
```

`IndexResponse` 将会提供给你索引信息

```
// Index name
String _index = response.getIndex();
// Type name
String _type = response.getType();
// Document ID (generated or not)
String _id = response.getId();
// Version (if it's the first time you index this document, you will get: 1)
long _version = response.getVersion();
```

如果你在索引时提供了过滤，那么 `IndexResponse` 将会提供一个过滤器（percolator ）

```
IndexResponse response = client.prepareIndex("twitter", "tweet", "1")
        .setSource(json)
        .execute()
        .actionGet();

List<String> matches = response.matches();
```

# 获取API

获取API允许你通过id从索引中获取类型化的JSON文档，如下例：

```
GetResponse response = client.prepareGet("twitter", "tweet", "1")
        .execute()
        .actionGet();
```

# 操作线程

The get API allows to set the threading model the operation will be performed when the actual execution of the API is performed on the same node (the API is executed on a shard that is allocated on the same server).

默认情况下，`operationThreaded`设置为true表示操作执行在不同的线程上面。下面是一个设置为false的例子。

```
GetResponse response = client.prepareGet("twitter", "tweet", "1")
        .setOperationThreaded(false)
        .execute()
        .actionGet();
```

## 删除API

删除api允许你通过id，从特定的索引中删除类型化的JSON文档。如下例：

```
DeleteResponse response = client.prepareDelete("twitter", "tweet", "1")
        .execute()
        .actionGet();
```

## 操作线程

The get API allows to set the threading model the operation will be performed when the actual execution of the API is performed on the same node (the API is executed on a shard that is allocated on the same server).

默认情况下，`operationThreaded`设置为true表示操作执行在不同的线程上面。下面是一个设置为false的例子。

```
DeleteResponse response = client.prepareDelete("twitter", "tweet", "1")
        .setOperationThreaded(false)
        .execute()
        .actionGet();
```

# 更新API

你能够创建一个 `UpdateRequest` ,然后将其发送给client。

```
UpdateRequest updateRequest = new UpdateRequest();
updateRequest.index("index");
updateRequest.type("type");
updateRequest.id("1");
updateRequest.doc(jsonBuilder()
        .startObject()
            .field("gender", "male")
        .endObject());
client.update(updateRequest).get();
```

或者你也可以利用 `prepareUpdate` 方法

```
1 client.prepareUpdate("ttl", "doc", "1")
2         .setScript("ctx._source.gender = \"male\""  , ScriptService.ScriptType.INLINE)
3         .get();

5 client.prepareUpdate("ttl", "doc", "1")
6         .setDoc(jsonBuilder()
7             .startObject()
8                 .field("gender", "male")
9             .endObject())
10         .get();
```

1-3行用脚本来更新索引，5-10行用doc来更新索引。

当然，java API也支持使用 `upsert` 。如果文档还不存在，会根据 `upsert` 内容创建一个新的索引。

```
IndexRequest indexRequest = new IndexRequest("index", "type", "1")
        .source(jsonBuilder()
            .startObject()
                .field("name", "Joe Smith")
                .field("gender", "male")
            .endObject());
UpdateRequest updateRequest = new UpdateRequest("index", "type", "1")
        .doc(jsonBuilder()
            .startObject()
                .field("gender", "male")
            .endObject())
        .upsert(indexRequest);
client.update(updateRequest).get();
```

如果文档 `index/type/1` 已经存在，那么在更新操作完成之后，文档为：

```
{
    "name"  : "Joe Dalton",
    "gender": "male"
}
```

否则，文档为：

```
{
    "name" : "Joe Smith",
    "gender": "male"
}
```

# bulk API

bulk API允许开发者在一个请求中索引和删除多个文档。下面是使用实例。

```
import static org.elasticsearch.common.xcontent.XContentFactory.*;

BulkRequestBuilder bulkRequest = client.prepareBulk();

// either use client#prepare, or use Requests# to directly build index/delete requests
bulkRequest.add(client.prepareIndex("twitter", "tweet", "1")
        .setSource(jsonBuilder()
                    .startObject()
                        .field("user", "kimchy")
                        .field("postDate", new Date())
                        .field("message", "trying out Elasticsearch")
                    .endObject()
                  )
        );

bulkRequest.add(client.prepareIndex("twitter", "tweet", "2")
        .setSource(jsonBuilder()
                    .startObject()
                        .field("user", "kimchy")
                        .field("postDate", new Date())
                        .field("message", "another post")
                    .endObject()
                  )
        );

BulkResponse bulkResponse = bulkRequest.execute().actionGet();
if (bulkResponse.hasFailures()) {
    // process failures by iterating through each bulk response item
}
```

# 搜索API

搜索API允许开发者执行一个搜索查询，返回满足查询条件的搜索信息。它能够跨索引以及跨类型执行。查询既可以用Java查询API也可以用Java过滤API。 查询的请求体由`SearchSourceBuilder`构建。

```
import org.elasticsearch.action.search.SearchResponse;
import org.elasticsearch.action.search.SearchType;
import org.elasticsearch.index.query.FilterBuilders.*;
import org.elasticsearch.index.query.QueryBuilders.*;

SearchResponse response = client.prepareSearch("index1", "index2")
        .setTypes("type1", "type2")
        .setSearchType(SearchType.DFS_QUERY_THEN_FETCH)
        .setQuery(QueryBuilders.termQuery("multi", "test"))             // Query
        .setPostFilter(FilterBuilders.rangeFilter("age").from(12).to(18))   // Filter
        .setFrom(0).setSize(60).setExplain(true)
        .execute()
        .actionGet();
```

注意，所有的参数都是可选的。下面是最简洁的形式。

```
// MatchAll on the whole cluster with all default options
SearchResponse response = client.prepareSearch().execute().actionGet();
```

# 在Java中使用scrolls

```
import static org.elasticsearch.index.query.FilterBuilders.*;
import static org.elasticsearch.index.query.QueryBuilders.*;

QueryBuilder qb = termQuery("multi", "test");

SearchResponse scrollResp = client.prepareSearch(test)
        .setSearchType(SearchType.SCAN)
        .setScroll(new TimeValue(60000))
        .setQuery(qb)
        .setSize(100).execute().actionGet(); //100 hits per shard will be returned for ea
//Scroll until no hits are returned
while (true) {
    for (SearchHit hit : scrollResp.getHits()) {
        //Handle the hit...
    }
    scrollResp = client.prepareSearchScroll(scrollResp.getScrollId()).setScroll(new TimeV
    //Break condition: No hits are returned
    if (scrollResp.getHits().getHits().length == 0) {
        break;
    }
}
```

# 多搜索API

```
SearchRequestBuilder srb1 = node.client()
    .prepareSearch().setQuery(QueryBuilders.queryString("elasticsearch")).setSize(1);
SearchRequestBuilder srb2 = node.client()
    .prepareSearch().setQuery(QueryBuilders.matchQuery("name", "kimchy")).setSize(1);

MultiSearchResponse sr = node.client().prepareMultiSearch()
        .add(srb1)
        .add(srb2)
        .execute().actionGet();

// You will get all individual responses from MultiSearchResponse#getResponses()
long nbHits = 0;
for (MultiSearchResponse.Item item : sr.getResponses()) {
    SearchResponse response = item.getResponse();
    nbHits += response.getHits().getTotalHits();
}
```

## 使用聚合

下面的例子显示怎样添加两个聚合到你的搜索中。

```
SearchResponse sr = node.client().prepareSearch()
    .setQuery(QueryBuilders.matchAllQuery())
    .addAggregation(
            AggregationBuilders.terms("agg1").field("field")
    )
    .addAggregation(
            AggregationBuilders.dateHistogram("agg2")
                    .field("birth")
                    .interval(DateHistogram.Interval.YEAR)
    )
    .execute().actionGet();

// Get your facet results
Terms agg1 = sr.getAggregations().get("agg1");
DateHistogram agg2 = sr.getAggregations().get("agg2");
```

## 使用搜索模板

定义你的模板参数为 `Map<String,String>`

```
Map<String, String> template_params = new HashMap<>();
template_params.put("param_gender", "male");
```

你可以用你保存在 `config/scripts` 目录中的模板。例如，你拥有如下的文件 `config/scripts/template_gender.mustache`

```
{
    "template" : {
        "query" : {
            "match" : {
                "gender" : "{{param_gender}}"
            }
```

```
        }
    }
}
```

可以通过如下方式执行：

```
SearchResponse sr = client.prepareSearch()
        .setTemplateName("template_gender")
        .setTemplateType(ScriptService.ScriptType.FILE)
        .setTemplateParams(template_params)
        .get();
```

你也可以将模板存储在一个专门的索引中，这个索引名为 `.scripts`

```
client.preparePutIndexedScript("mustache", "template_gender",
        "{\n" +
        "    \"template\" : {\n" +
        "        \"query\" : {\n" +
        "            \"match\" : {\n" +
        "                \"gender\" : \"{{param_gender}}\"\n" +
        "            }\n" +
        "        }\n" +
        "    }\n" +
        "}").get();
```

为了用这个被索引的模板，需要用到 `ScriptService.ScriptType.INDEXED`：

```
SearchResponse sr = client.prepareSearch()
        .setTemplateName("template_gender")
        .setTemplateType(ScriptService.ScriptType.INDEXED)
        .setTemplateParams(template_params)
        .get();
```

# 计数API

计数API允许开发者简单的执行一个查询，返回和查询条件相匹配的文档的总数。它可以跨多个索引以及跨多个类型执行。

```
import static org.elasticsearch.index.query.xcontent.FilterBuilders.*;
import static org.elasticsearch.index.query.xcontent.QueryBuilders.*;

CountResponse response = client.prepareCount("test")
        .setQuery(termQuery("_type", "type1"))
        .execute()
        .actionGet();
```

# 操作线程

它有三种线程模式。`NO_THREADS` 模式表明操作在当前线程中执行；`SINGLE_THREAD` 模式表明操作在一个独立的不同线程中执行，所有的分片共用这个线程；`THREAD_PER_SHARD` 模式表明操作在一个独立的不同线程中执行，并且一个分片一个线程。默认的模式是 `SINGLE_THREAD`。

# 基于查询的删除API

基于查询的删除API允许开发者基于查询删除一个或者多个索引、一个或者多个类型。下面是一个例子。

```
import static org.elasticsearch.index.query.FilterBuilders.*;
import static org.elasticsearch.index.query.QueryBuilders.*;

DeleteByQueryResponse response = client.prepareDeleteByQuery("test")
        .setQuery(termQuery("_type", "type1"))
        .execute()
        .actionGet();
```

# facets

Elasticsearch提供完整的java API用来支持facets。在查询的过程中，将需要计数的facets添加到 `FacetBuilders` 中。然后将该 `FacetBuilders` 条件到查询请求中。

```
SearchResponse sr = node.client().prepareSearch()
        .setQuery( /* your query */ )
        .addFacet( /* add a facet */ )
        .execute().actionGet();
```

为了构建facets请求，需要用到 `FacetBuilders` 帮助类。你只需要在你的程序中导入它即可。

```
import org.elasticsearch.search.facet.FacetBuilders.*;
```

# terms facet

## 准备一个facet请求

下面的例子新建一个facet请求

```
FacetBuilders.termsFacet("f")
    .field("brand")
    .size(10);
```

## 利用facet响应

```
import org.elasticsearch.search.facet.terms.*;

// sr is here your SearchResponse object
TermsFacet f = (TermsFacet) sr.getFacets().facetsAsMap().get("f");

f.getTotalCount();      // Total terms doc count
f.getOtherCount();      // Not shown terms doc count
f.getMissingCount();    // Without term doc count

// For each entry
for (TermsFacet.Entry entry : f) {
    entry.getTerm();    // Term
    entry.getCount();   // Doc count
}
```

# 范围facet

## 准备一个facet请求

下面的例子新建一个facet请求

```
FacetBuilders.rangeFacet("f")
    .field("price")         // Field to compute on
    .addUnboundedFrom(3)    // from -infinity to 3 (excluded)
    .addRange(3, 6)         // from 3 to 6 (excluded)
    .addUnboundedTo(6);     // from 6 to +infinity
```

### 利用facet响应

```
import org.elasticsearch.search.facet.range.*;

// sr is here your SearchResponse object
RangeFacet f = (RangeFacet) sr.getFacets().facetsAsMap().get("f");

// For each entry
for (RangeFacet.Entry entry : f) {
    entry.getFrom();    // Range from requested
    entry.getTo();      // Range to requested
    entry.getCount();   // Doc count
    entry.getMin();     // Min value
    entry.getMax();     // Max value
    entry.getMean();    // Mean
    entry.getTotal();   // Sum of values
}
```

## 直方图(Histogram) Facet

### 准备一个facet请求

下面的例子新建一个facet请求

```
HistogramFacetBuilder facet = FacetBuilders.histogramFacet("f")
    .field("price")
    .interval(1);
```

### 利用facet响应

```
import org.elasticsearch.search.facet.histogram.*;
// sr is here your SearchResponse object
HistogramFacet f = (HistogramFacet) sr.getFacets().facetsAsMap().get("f");

// For each entry
for (HistogramFacet.Entry entry : f) {
    entry.getKey();     // Key (X-Axis)
    entry.getCount();   // Doc count (Y-Axis)
}
```

## 日期直方图(Histogram) Facet

### 准备一个facet请求

下面的例子新建一个facet请求

```
FacetBuilders.dateHistogramFacet("f")
    .field("date")      // Your date field
    .interval("year");  // You can also use "quarter", "month", "week", "day",
                        // "hour" and "minute" or notation like "1.5h" or "2w"
```

### 利用facet响应

```
import org.elasticsearch.search.facet.datehistogram.*;
// sr is here your SearchResponse object
DateHistogramFacet f = (DateHistogramFacet) sr.getFacets().facetsAsMap().get("f");

// For each entry
for (DateHistogramFacet.Entry entry : f) {
    entry.getTime();    // Date in ms since epoch (X-Axis)
    entry.getCount();   // Doc count (Y-Axis)
}
```

## 过滤facet(不是facet过滤)

### 准备一个facet请求

下面的例子新建一个facet请求

```
FacetBuilders.filterFacet("f",
    FilterBuilders.termFilter("brand", "heineken"));    // Your Filter here
```

### 利用facet响应

```
import org.elasticsearch.search.facet.filter.*;

// sr is here your SearchResponse object
FilterFacet f = (FilterFacet) sr.getFacets().facetsAsMap().get("f");
f.getCount();   // Number of docs that matched
```

## 查询facet

### 准备一个facet请求

下面的例子新建一个facet请求

```
FacetBuilders.queryFacet("f",
    QueryBuilders.matchQuery("brand", "heineken"));
```

### 利用facet响应

```
import org.elasticsearch.search.facet.query.*;
// sr is here your SearchResponse object
QueryFacet f = (QueryFacet) sr.getFacets().facetsAsMap().get("f");

f.getCount();   // Number of docs that matched
```

## 统计

### 准备一个facet请求

下面的例子新建一个facet请求

```
FacetBuilders.statisticalFacet("f")
   .field("price");
```

### 利用facet响应

```
import org.elasticsearch.search.facet.statistical.*;
// sr is here your SearchResponse object
StatisticalFacet f = (StatisticalFacet) sr.getFacets().facetsAsMap().get("f");

f.getCount();           // Doc count
f.getMin();             // Min value
f.getMax();             // Max value
f.getMean();            // Mean
f.getTotal();           // Sum of values
f.getStdDeviation();    // Standard Deviation
f.getSumOfSquares();    // Sum of Squares
f.getVariance();        // Variance
```

## Terms Stats Facet

### 准备一个facet请求

下面的例子新建一个facet请求

```
FacetBuilders.termsStatsFacet("f")
    .keyField("brand")
    .valueField("price");
```

### 利用facet响应

```
// sr is here your SearchResponse object
TermsStatsFacet f = (TermsStatsFacet) sr.getFacets().facetsAsMap().get("f");
f.getTotalCount();      // Total terms doc count
f.getOtherCount();      // Not shown terms doc count
f.getMissingCount();    // Without term doc count
```

```
// For each entry
for (TermsStatsFacet.Entry entry : f) {
    entry.getTerm();           // Term
    entry.getCount();          // Doc count
    entry.getMin();            // Min value
    entry.getMax();            // Max value
    entry.getMean();           // Mean
    entry.getTotal();          // Sum of values
}
```

# 地理距离Facet

## 准备一个facet请求

下面的例子新建一个facet请求

```
FacetBuilders.geoDistanceFacet("f")
    .field("pin.location")             // Field containing coordinates we want to compar
    .point(40, -70)                    // Point from where we start (0)
    .addUnboundedFrom(10)              // 0 to 10 km (excluded)
    .addRange(10, 20)                  // 10 to 20 km (excluded)
    .addRange(20, 100)                 // 20 to 100 km (excluded)
    .addUnboundedTo(100)               // from 100 km to infinity (and beyond ;-) )
    .unit(DistanceUnit.KILOMETERS);    // All distances are in kilometers. Can be MILES
```

## 利用facet响应

```
// sr is here your SearchResponse object
GeoDistanceFacet f = (GeoDistanceFacet) sr.getFacets().facetsAsMap().get("f");

// For each entry
for (GeoDistanceFacet.Entry entry : f) {
    entry.getFrom();           // Distance from requested
    entry.getTo();             // Distance to requested
    entry.getCount();          // Doc count
    entry.getMin();            // Min value
    entry.getMax();            // Max value
    entry.getTotal();          // Sum of values
    entry.getMean();           // Mean
}
```

# facet过滤器（不是过滤facet）

默认情况下，不管过滤器存在与否，facet都是作用在查询的结果集上。如果你需要计数带有过滤器的facet，你能够通过 `AbstractFacetBuilder#facetFilter(FilterBuilder)` 添加 过滤器到任何facet上。

```
FacetBuilders
    .termsFacet("f").field("brand") // Your facet
    .facetFilter( // Your filter here
        FilterBuilders.termFilter("colour", "pale")
    );
```

例如，你可以在你的查询中重用创建的过滤器

```
// A common filter
FilterBuilder filter = FilterBuilders.termFilter("colour", "pale");

TermsFacetBuilder facet = FacetBuilders.termsFacet("f")
    .field("brand")
    .facetFilter(filter);                          // We apply it to the facet

SearchResponse sr = node.client().prepareSearch()
    .setQuery(QueryBuilders.matchAllQuery())
    .setFilter(filter)                             // We apply it to the query
    .addFacet(facet)
    .execute().actionGet();
```

## 作用域

默认情况下，facet作用在查询的结果集上。但是，不管是哪个查询，你可以用 `global` 参数去计算来自于索引中的所有文档的facet。

```
TermsFacetBuilder facet = FacetBuilders.termsFacet("f")
    .field("brand")
    .global(true);
```

# 实验实例

## 测试环境

两台4核8G内存的机器组成的集群。网络带宽为100m。JVM配置最大内存为2g。elasticsearch有10个分片，1个副本。通过导入mysql的数据来进行实验。

## 导入数据

需要下载相关插件。

将ODS的yourtable表中所有数据导入到Elasticsearch中，用来进行测试。

```
curl -XPUT 'localhost:9200/_river/my_jdbc_river/_meta' -d '{
"type" : "jdbc",
    "jdbc" : {
        "url" : "jdbc:mysql://yourhost:3306/yourdb",
        "user" : "*",
        "password" : "*",
        "index" : "top_user",
        "type" : " top_user",
        "sql" : "select * from yourtable"
}
}'
```

yourtable表中总共有18个字段，类型包括bigint，int，varchar，datetime，总共有15292729条数据，数据总量为4.2G。相同的数据，es单副本占用的空间比mysql占用的空间不会大多少，应该在10%以内。

# 基本操作

## 查看节点信息

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

## 查看索引信息

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

## 删除索引信息

```
curl -XDELETE 'localhost:9200/top_user?pretty'
curl -XDELETE 'localhost:9200/_river?pretty'
```

# 例子

这些例子统计了5次运行的时间。有些查询在mysql上查询非常缓慢，所以没有将其运行时间列出。

## 查看所有数据（默认返回10条）

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

| 运行次数 | 运行时间（ms） |
| --- | --- |
| 1 | 260 |
| 2 | 124 |
| 3 | 39 |
| 4 | 38 |
| 5 | 47 |

等价于的sql语句是：`select * from top_user limit 0,10;`

## 分页查询（20-40）

```
curl -XPOST 'localhost:9200/top_user/_search?pretty' -d '
{
"query": { "match_all": {} },
"from": 20,
"size": 40
}'
```

| 运行次数 | 运行时间（ms） |
| --- | --- |
| 1 | 54 |
| 2 | 59 |
| 3 | 45 |
| 4 | 49 |
| 5 | 76 |

等价于的sql语句是：`select * from top_user limit 20,40;`

## 按照字段排序（默认返回10条）

```
curl -XPOST 'localhost:9200/top_user/_search?pretty' -d '
{
"query": { "match_all": {} },
"sort": { "sex": { "order": "asc" } }
}'
```

| 运行次数 | 运行时间（ms） |
| --- | --- |
| 1 | 982 |

| | |
|---|---|
| 2 | 1087 |
| 3 | 208 |
| 4 | 147 |
| 5 | 114 |

等价于的sql语句是：`select * from top_user order by sex asc limit 0,10;`

## 指定_source字段

```
curl -XPOST 'localhost:9200/top_user/_search?pretty' -d '
{
    "query": { "match_all": {} },
    "_source": ["buyer_nick", "sex"]
}'
```

| 运行次数 | 运行时间（**ms**） |
|---|---|
| 1 | 92 |
| 2 | 108 |
| 3 | 53 |
| 4 | 44 |
| 5 | 79 |

等价于的sql语句是：select buyer_nick from top_user;

## **match**查询

```
curl -XPOST 'localhost:9200/top_user/_search?pretty' -d '
{
"query": {
    "bool": {
        "should": [
            { "match": { "city": "上海" } },
            { "match": { "city": "北京" } }
        ]
     }
 }
}'
```

| 运行次数 | 运行时间（**ms**） |
|---|---|
| 1 | 1410 |
| 2 | 108 |
| 3 | 57 |
| 4 | 159 |
| 5 | 79 |

等价于的sql语句是：`select * from top_user where city="上海" or city="北京" limit 0,10;`

## 通过过滤器查询

```
curl -XPOST 'localhost:9200/top_user/_search?pretty' -d '
{
    "query": {
        "filtered": {
            "query": { "match_all": {} },
            "filter": {
                "range": {
                    "score": {
                        "gte": 200,
                        "lte": 2000
                    }
                }
            }
        }
    }
}'
```

| 运行次数 | 运行时间（**ms**） |
| --- | --- |
| 1 | 1644 |
| 2 | 1096 |
| 3 | 57 |
| 4 | 30 |
| 5 | 31 |

等价于的sql语句是：`select * from top_user where score>200 and score<2000 limit 0,10;`

## 聚合查询

```
curl -XPOST 'localhost:9200/top_user/_search?pretty' -d '
{
    "size": 0,
    "aggs": {
        "group_by_city": {
            "terms": {
                "field": "city"
            }
        }
    }
}'
```

| 运行次数 | 运行时间（**ms**） |
| --- | --- |
| 1 | 2941 |
| 2 | 1887 |
| 3 | 542 |
| 4 | 659 |
| 5 | 612 |

等价于的sql语句是: `select count(*) from top_user group by city order by count(*) desc limit`

`0,10;`

一下得到每个level的用户的平均得分，并按照得分排序

```
curl -XPOST 'localhost:9200/top_user/_search?pretty' -d '
{
    "size": 0,
    "aggs": {
        "group_by_level": {
            "terms": {
                "field": "level",
                "order": {
                    "average_score": "desc"
                }
            },
            "aggs": {
                "average_score": {
                    "avg": {
                        "field": "score"
                    }
                }
            }
        }
    }
}'
```

| 运行次数 | 运行时间（**ms**） |
| --- | --- |
| 1 | 3332 |
| 2 | 2706 |
| 3 | 599 |
| 4 | 633 |
| 5 | 542 |

等价于的sql语句是: `select max(score) as avg_score from top_user group by level order by avg_score desc limit 0,10;`

## 总结

与mysql数据库查询相对比，可以发现Elasticsearch在聚合查询和非主键排序上效果提升明显。如下面两种情况，mysql很慢，但是Elasticsearch较快。

- select * from top_user order by sex asc limit 20,40;
- select count*() from top_user group by city order by count*() desc limit 0,10;

# elasticsearch优点

elasticsearch相比于solr拥有一些重要特征：

- ElasticSearch是分布式的。不需要其他组件，分发是实时的，被叫做"Push replication"。
- ElasticSearch 完全支持Apache Lucene的接近实时的搜索。
- ElasticSearch 采用Gateway概念，使得全备份更简单。
- 支持更多的客户端库，如PHP, Ruby, Perl, Scala, Python, .NET, Javascript, Erlang, Clojure
- 自包含集群
- 自动化插件安装
- 集合脚本语言查询
- 导入性能更好，查询性能与solr持平

# elasticsearch vs solr

比较1

比较2

# elasticsearch社区版和商业版

社区版和商业版几乎没有区别，但是商用版提供了marvel工具，用于监控集群的状态。

# 下一步的工作

这里要特别提出权限验证的问题。对于商用产品权限验证是必须的。两者都不支持权限验证。需要用户自己增加这一块的功能。然而就目前来看，两者在增加该功能的难易程度来说，elasticsearch是相对容易很多。原因为elasticsearch内部节点之间使用自定义协议；而Solr4.x内部节点之间使用了和外部一样的http协议，并且节点间的通信关系混乱，所以对目前的Solr4.x增加权限验证以及ACL将很难寻找到一个优雅的解决方案，但并不是不可以。

现在，elasticsearch正在致力于开发shield项目，该项目为elasticsearch提供安全保证。它在不久的将来就会发布。但是我们并不清楚它是否会收费。

# 相关资料

elasticsearch权威指南

elasticsearch下载国内镜像

Kibana 中文指南

# river-jdbc

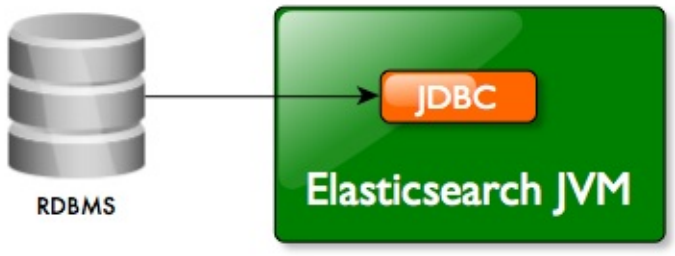
## 安装

```
./bin/plugin --install jdbc --url http://xbib.org/repository/org/xbib/elasticsearch/plugi
```

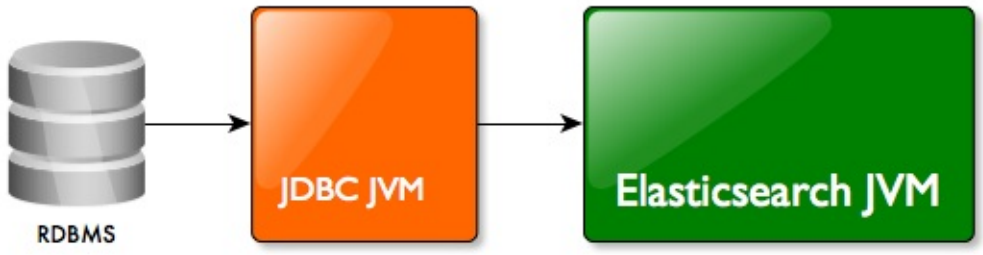
## 文档

### 两种方式：river或者feeder

该插件能够以“pull模式”执行river和以“push模式”执行feeder。在feeder模式下插件运行在不同的JVM中，可以连接到远程的Elasticsearch集群。

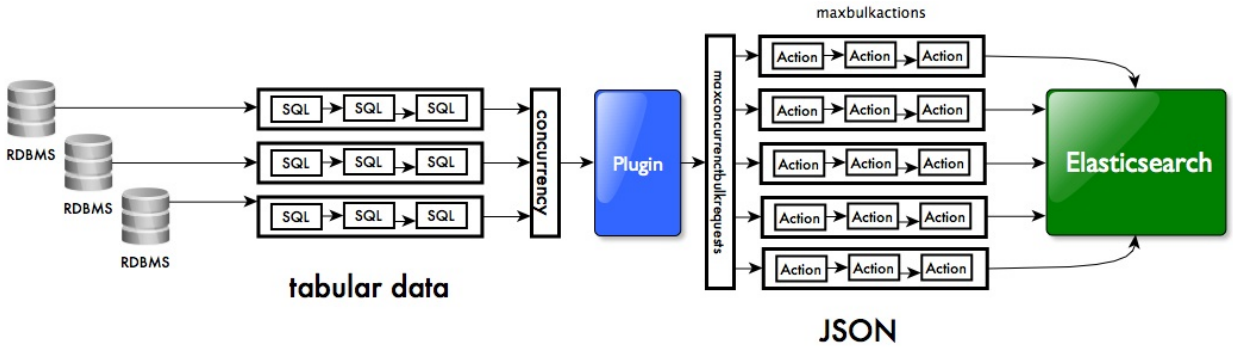



River Architecture




Feeder Architecture

该插件可以从不同的关系数据库源并行的获取数据。当索引到elasticsearch中时，多线程bulk模式确保了高吞吐。




## 安装运行river

```
#安装elasticsearch
curl -OL https://download.elasticsearch.org/elasticsearch/elasticsearch/elasticsearch-1.4

cd $ES_HOME
unzip path/to/elasticsearch-1.4.2.zip

#安装JDBC插件
./bin/plugin --install jdbc --url http://xbib.org/repository/org/xbib/elasticsearch/plugi

#下载mysql driver
curl -o mysql-connector-java-5.1.33.zip -L 'http://dev.mysql.com/get/Downloads/Connector-
cp mysql-connector-java-5.1.33-bin.jar $ES_HOME/plugins/jdbc/ chmod 644 $ES_HOME/plugins/

#启动elasticsearch
./bin/elasticsearch

#停止river
curl -XDELETE 'localhost:9200/_river/my_jdbc_river/'
```

## JDBC插件参数

JDBC插件一般的格式如下：

```
curl -XPUT 'localhost:9200/_river/<rivername>/_meta' -d '{
    <river parameters>
    "type" : "jdbc",
    "jdbc" : {
         <river definition>
    }
}'
```

例如

```
curl -XPUT 'localhost:9200/_river/my_jdbc_river/_meta' -d '{
    "type" : "jdbc",
    "jdbc" : {
        "url" : "jdbc:mysql://localhost:3306/test",
        "user" : "",
        "password" : "",
        "sql" : "select * from orders",
```

```
        "index" : "myindex",
        "type" : "mytype",
        ...
    }
}'
```

如果一个数组传递给jdbc字段，多个river源也是可以的。

```
curl -XPUT 'localhost:9200/_river/my_jdbc_river/_meta' -d '{
     <river parameters>
    "type" : "jdbc",
    "jdbc" : [ {
         <river definition 1>
    }, {
         <river definition 2>
    } ]
}'
```

可以通过 `concurrency` 参数并行控制多个river源

```
curl -XPUT 'localhost:9200/_river/my_jdbc_river/_meta' -d '{
     <river parameters>
    "concurrency" : 2,
    "type" : "jdbc",
    "jdbc" : [ {
         <river definition 1>
    }, {
         <river definition 2>
    } ]
}'
```

### `jdbc` 块外部的参数

`strategy` - JDBC插件的策略。当前的实现有 `simple` 和 `column` 。

`schedule` - a single or a list of cron expressions for scheduled execution

`threadpoolsize` -scheduled executions的线程池大小

`interval` - 两个river启动的延迟时间

`max_bulk_actions` - 每个bulk索引请求提交的长度(默认是1000)

`max_concurrrent_bulk_requests` - bulk请求的并行数量（默认是2*cpu core）

`max_bulk_volume` - 一个bulk请求的最大容量(默认是10m)

`max_request_wait` - 一个bulk请求最大的等待时间（默认是60s）

`flush_interval` - flushing索引文档到bulk action的间隔时间

### `jdbc` 块内部的参数

`url` - the JDBC driver URL

`user` - the JDBC database user

`password` - the JDBC database password

`sql` - SQL语句。既可以是一个字符串也可以是一个列表。

```
"sql" : [
    {
        "statement" : "select ... from ... where a = ?, b = ?, c = ?",
        "parameter" : [ "value for a", "value for b", "value for c" ]
    },
    {
        "statement" : "insert into  ... where a = ?, b = ?, c = ?",
        "parameter" : [ "value for a", "value for b", "value for c" ],
        "write" : "true"
    },
    {
        "statement" : ...
    }
]
```

`sql.statement` - the SQL statement

`sql.write` - 如果为true，SQL语句解释为一个insert/update语句，这个语句写权限。默认为false

`sql.callable` - 如果为true，SQL语句解释为一个 `CallableStatement` 用于保存存储过程。默认为false

`sql.parameter` - 绑定参数到SQL语句。可以用到一些指定的值

- $now - the current timestamp
- $job - a job counter
- $count - last number of rows merged
- $river.name - the river name
- $last.sql.start - a timestamp value for the time when the last SQL statement started
- $last.sql.end - a timestamp value for the time when the last SQL statement ended
- $last.sql.sequence.start - a timestamp value for the time when the last SQL sequence started
- $last.sql.sequence.end - a timestamp value for the time when the last SQL sequence ended
- $river.state.started - the timestamp of river start (from river state)
- $river.state.timestamp - last timestamp of river activity (from river state)
- $river.state.counter - counter from river state, counts the numbers of runs

`locale` - the default locale (used for parsing numerical values, floating point character. Recommended values is "en_US")

`timezone` - the timezone for JDBC setTimestamp() calls when binding parameters with timestamp values

`rounding` - rounding mode for parsing numeric values. Possible values "ceiling", "down", "floor", "halfdown", "halfeven", "halfup", "unnecessary", "up"

`scale` - the precision of parsing numeric values

`autocommit` - true if each statement should be automatically executed. Default is false

`fetchsize` - the fetchsize for large result sets, most drivers use this to control the amount of rows in the buffer while iterating through the result set

`max_rows` - limit the number of rows fetches by a statement, the rest of the rows is ignored

`max_retries` - the number of retries to (re)connect to a database

`max_retries_wait` - a time value for the time that should be waited between retries. Default is "30s"

`resultset_type` - the JDBC result set type, can be TYPE_FORWARD_ONLY, TYPE_SCROLL_SENSITIVE, TYPE_SCROLL_INSENSITIVE. Default is TYPE_FORWARD_ONLY

`resultset_concurrency` - the JDBC result set concurrency, can be CONCUR_READ_ONLY, CONCUR_UPDATABLE. Default is CONCUR_UPDATABLE

`ignore_null_values` - if NULL values should be ignored when constructing JSON documents. Default is false

`prepare_database_metadata` - if the driver metadata should be prepared as parameters for acccess by the river. Default is false

`prepare_resultset_metadata` - if the result set metadata should be prepared as parameters for acccess by the river. Default is false

`column_name_map` - a map of aliases that should be used as a replacement for column names of the database. Useful for Oracle 30 char column name limit. Default is null

`query_timeout` - a second value for how long an SQL statement is allowed to be executed before it is considered as lost. Default is 1800

`connection_properties` - a map for the connection properties for driver connection creation. Default is null

`index` - the Elasticsearch index used for indexing

`type` - the Elasticsearch type of the index used for indexing

`index_settings` - optional settings for the Elasticsearch index

`type_mapping` - optional mapping for the Elasticsearch index type

### 默认的参数设置

```
{
    "strategy" : "simple",
    "schedule" : null,
    "interval" : 0L,
    "threadpoolsize" : 4,
    "max_bulk_actions" : 10000,
    "max_concurrent_bulk_requests" : 2 * available CPU cores,
    "max_bulk_volume" : "10m",
    "max_request_wait" : "60s",
    "flush_interval" : "5s",
    "jdbc" : {
```

```
        "url" : null,
        "user" : null,
        "password" : null,
        "sql" : null,
        "locale" : Locale.getDefault().toLanguageTag(),
        "timezone" : TimeZone.getDefault(),
        "rounding" : null,
        "scale" : 2,
        "autocommit" : false,
        "fetchsize" : 10, /* MySQL: Integer.MIN */
        "max_rows" : 0,
        "max_retries" : 3,
        "max_retries_wait" : "30s",
        "resultset_type" : "TYPE_FORWARD_ONLY",
        "resultset_concurreny" : "CONCUR_UPDATABLE",
        "ignore_null_values" : false,
        "prepare_database_metadata" : false,
        "prepare_resultset_metadata" : false,
        "column_name_map" : null,
        "query_timeout" : 1800,
        "connection_properties" : null,
        "index" : "jdbc",
        "type" : "jdbc",
        "index_settings" : null,
        "type_mapping" : null,
    }
}
```

## 结构化对象

SQL查询的一个优势是连接操作。从许多表获得数据形成新的元组。

```
curl -XPUT 'localhost:9200/_river/my_jdbc_river/_meta' -d '{
    "type" : "jdbc",
    "jdbc" : {
        "url" : "jdbc:mysql://localhost:3306/test",
        "user" : "",
        "password" : "",
        "sql" : "select \"relations\" as \"_index\", orders.customer as \"_id\", orders.c
    }
}'
```

sql结构是

```
mysql> select "relations" as "_index", orders.customer as "_id", orders.customer as "cont
+-----------+-------+------------------+------------------+
| _index    | _id   | contact.customer | contact.employee |
+-----------+-------+------------------+------------------+
| relations | Big   | Big              | Smith            |
| relations | Large | Large            | Müller           |
| relations | Large | Large            | Meier            |
| relations | Large | Large            | Schulze          |
| relations | Huge  | Huge             | Müller           |
| relations | Huge  | Huge             | Meier            |
| relations | Huge  | Huge             | Schulze          |
| relations | Good  | Good             | Müller           |
| relations | Good  | Good             | Meier            |
```

```
| relations | Good  | Good             | Schulze          |
| relations | Bad   | Bad              | Jones            |
+-----------+-------+------------------+------------------+
11 rows in set (0.00 sec)
```

得到的JSON对象为

```
index=relations id=Big {"contact":{"employee":"Smith","customer":"Big"}}
index=relations id=Large {"contact":{"employee":["Müller","Meier","Schulze"],"customer":"
index=relations id=Huge {"contact":{"employee":["Müller","Meier","Schulze"],"customer":"H
index=relations id=Good {"contact":{"employee":["Müller","Meier","Schulze"],"customer":"G
index=relations id=Bad {"contact":{"employee":"Jones","customer":"Bad"}}
```

## 怎样获取一个表

它dump一个表到Elasticsearch中。如果没有给定 `_id` 列，IDs将会自动生成。

```
curl -XPUT 'localhost:9200/_river/my_jdbc_river/_meta' -d '{
    "type" : "jdbc",
    "jdbc" : {
        "url" : "jdbc:mysql://localhost:3306/test",
        "user" : "",
        "password" : "",
        "sql" : "select * from orders"
    }
}'
```

结果是：

```
id=<random> {"product":"Apples","created":null,"department":"American Fruits","quantity":
id=<random> {"product":"Bananas","created":null,"department":"German Fruits","quantity":1
id=<random> {"product":"Oranges","created":null,"department":"German Fruits","quantity":2
id=<random> {"product":"Apples","created":1338501600000,"department":"German Fruits","qua
id=<random> {"product":"Oranges","created":1338501600000,"department":"English Fruits","q
```

## 怎样获得增量的数据

推荐使用时间戳来同步。下面的例子获取最后一次river运行之后添加的所有产品行。

```
{
    "type" : "jdbc",
    "jdbc" : {
        "url" : "jdbc:mysql://localhost:3306/test",
        "user" : "",
        "password" : "",
        "sql" : [
            {
                "statement" : "select * from \"products\" where \"mytimestamp\" > ?",
                "parameter" : [ "$river.state.last_active_begin" ]
            }
```

```
        ],
        "index" : "my_jdbc_river_index",
        "type" : "my_jdbc_river_type"
    }
}
```