**1.1为了搜索，你懂的**

Elasticsearch是一个基于[Apache Lucene(TM)](https://lucene.apache.org/core/" \t "_blank)的开源搜索引擎。无论在开源还是专有领域，Lucene可以被认为是迄今为止最先进、性能最好的、功能最全的搜索引擎库。

但是，Lucene只是一个库。想要使用它，你必须使用Java来作为开发语言并将其直接集成到你的应用中，更糟糕的是，Lucene非常复杂，你需要深入了解检索的相关知识来理解它是如何工作的。

Elasticsearch也使用Java开发并使用Lucene作为其核心来实现所有索引和搜索的功能，但是它的目的是通过简单的RESTful API来隐藏Lucene的复杂性，从而让全文搜索变得简单。

不过，Elasticsearch不仅仅是Lucene和全文搜索，我们还能这样去描述它：

* 分布式的实时文件存储，每个字段都被索引并可被搜索
* 分布式的实时分析搜索引擎
* 可以扩展到上百台服务器，处理PB级结构化或非结构化数据

而且，所有的这些功能被集成到一个服务里面，你的应用可以通过简单的RESTful API、各种语言的客户端甚至命令行与之交互。

上手Elasticsearch非常容易。它提供了许多合理的缺省值，并对初学者隐藏了复杂的搜索引擎理论。它开箱即用（安装即可使用），只需很少的学习既可在生产环境中使用。

Elasticsearch在[Apache 2 license](http://www.apache.org/licenses/LICENSE-2.0.html" \t "_blank)下许可使用，可以免费下载、使用和修改。

随着你对Elasticsearch的理解加深，你可以根据不同的问题领域定制Elasticsearch的高级特性，这一切都是可配置的，并且配置非常灵活。

**模糊的历史**

多年前，一个叫做Shay Banon的刚结婚不久的失业开发者，由于妻子要去伦敦学习厨师，他便跟着也去了。在他找工作的过程中，为了给妻子构建一个食谱的搜索引擎，他开始构建一个早期版本的Lucene。

直接基于Lucene工作会比较困难，所以Shay开始抽象Lucene代码以便Java程序员可以在应用中添加搜索功能。他发布了他的第一个开源项目，叫做“Compass”。

后来Shay找到一份工作，这份工作处在高性能和内存数据网格的分布式环境中，因此高性能的、实时的、分布式的搜索引擎也是理所当然需要的。然后他决定重写Compass库使其成为一个独立的服务叫做Elasticsearch。

第一个公开版本出现在2010年2月，在那之后Elasticsearch已经成为Github上最受欢迎的项目之一，代码贡献者超过300人。一家主营Elasticsearch的公司就此成立，他们一边提供商业支持一边开发新功能，不过Elasticsearch将永远开源且对所有人可用。

Shay的妻子依旧等待着她的食谱搜索……

## 2.安装Elasticsearch

理解Elasticsearch最好的方式是去运行它，让我们开始吧！

安装Elasticsearch唯一的要求是安装官方新版的Java，地址：[www.java.com](http://www.java.com/)

你可以从 [elasticsearch.org\/download](http://www.elasticsearch.org/download/) 下载最新版本的Elasticsearch。

curl -L -O http://download.elasticsearch.org/PATH/TO/VERSION.zip <1>

unzip elasticsearch-$VERSION.zip

cd elasticsearch-$VERSION

1. 从 [elasticsearch.org\/download](http://www.elasticsearch.org/download/) 获得最新可用的版本号并填入URL中

**提示：**

在生产环境安装时，除了以上方法，你还可以使用Debian或者RPM安装包，地址在这里：[downloads page](http://www.elasticsearch.org/downloads)，或者也可以使用官方提供的 [Puppet module](https://github.com/elasticsearch/puppet-elasticsearch) 或者 [Chef cookbook](https://github.com/elasticsearch/cookbook-elasticsearch)。

## 1.2.2安装Marvel

[Marvel](http://www.elasticsearch.com/marvel)是Elasticsearch的管理和监控工具，在开发环境下免费使用。它包含了一个叫做Sense的交互式控制台，使用户方便的通过浏览器直接与Elasticsearch进行交互。

Elasticsearch线上文档中的很多示例代码都附带一个View in Sense的链接。点击进去，就会在Sense控制台打开相应的实例。安装Marvel不是必须的，但是它可以通过在你本地Elasticsearch集群中运行示例代码而增加与此书的互动性。

Marvel是一个插件，可在Elasticsearch目录中运行以下命令来下载和安装：

./bin/plugin -i elasticsearch/marvel/latest

你可能想要禁用监控，你可以通过以下命令关闭Marvel：

echo 'marvel.agent.enabled: false' >> ./config/elasticsearch.yml

## 1.2.3运行Elasticsearch

Elasticsearch已经准备就绪，执行以下命令可在前台启动：

./bin/elasticsearch

启动后，如果只有本地可以访问，尝试修改配置文件 elasticsearch.yml

中network.host(注意配置文件格式不是以#开头的要空一格， ：后要空一格) 为network.host: 0.0.0.0

如果想在后台以守护进程模式运行，添加-d参数。

打开另一个终端进行测试：

curl 'http://localhost:9200/?pretty'

你能看到以下返回信息：

{

"status": 200,

"name": "Shrunken Bones",

"version": {

"number": "1.4.0",

"lucene\_version": "4.10"

},

"tagline": "You Know, for Search"

}

这说明你的ELasticsearch集群已经启动并且正常运行，接下来我们可以开始各种实验了。

## 1.2.4集群和节点

**节点(node)**是一个运行着的Elasticsearch实例。**集群(cluster)**是一组具有相同cluster.name的节点集合，他们协同工作，共享数据并提供故障转移和扩展功能，当然一个节点也可以组成一个集群。

你最好找一个合适的名字来替代cluster.name的默认值，比如你自己的名字，这样可以防止一个新启动的节点加入到相同网络中的另一个同名的集群中。

你可以通过修改config/目录下的elasticsearch.yml文件，然后重启ELasticsearch来做到这一点。当Elasticsearch在前台运行，可以使用Ctrl-C快捷键终止，或者你可以调用shutdown API来关闭：

curl -XPOST 'http://localhost:9200/\_shutdown'

## 1.2.5查看Marvel和Sense

如果你安装了Marvel（作为管理和监控的工具），就可以在浏览器里通过以下地址访问它：

[http:\/\/localhost:9200\/\_plugin\/marvel\/](http://localhost:9200/_plugin/marvel/)

你可以在Marvel中通过点击dashboards，在下拉菜单中访问**Sense**开发者控制台，或者直接访问以下地址：

[http:\/\/localhost:9200\/\_plugin\/marvel\/sense\/](http://localhost:9200/_plugin/marvel/sense/)

## 3.与Elasticsearch交互

如何与Elasticsearch交互取决于你是否使用Java。

### Java API

Elasticsearch为Java用户提供了两种内置客户端：

#### 节点客户端(node client)：

节点客户端以无数据节点(none data node)身份加入集群，换言之，它自己不存储任何数据，但是它知道数据在集群中的具体位置，并且能够直接转发请求到对应的节点上。

#### 传输客户端(Transport client)：

这个更轻量的传输客户端能够发送请求到远程集群。它自己不加入集群，只是简单转发请求给集群中的节点。

两个Java客户端都通过9300端口与集群交互，使用Elasticsearch传输协议(Elasticsearch Transport Protocol)。集群中的节点之间也通过9300端口进行通信。如果此端口未开放，你的节点将不能组成集群。

**TIP**

Java客户端所在的Elasticsearch版本必须与集群中其他节点一致，否则，它们可能互相无法识别。

关于Java API的更多信息请查看相关章节：[Java API](http://www.elasticsearch.org/guide/)

### 基于HTTP协议，以JSON为数据交互格式的RESTful API

其他所有程序语言都可以使用RESTful API，通过9200端口的与Elasticsearch进行通信，你可以使用你喜欢的WEB客户端，事实上，如你所见，你甚至可以通过curl命令与Elasticsearch通信。

**NOTE**

Elasticsearch官方提供了多种程序语言的客户端——Groovy，Javascript， .NET，PHP，Perl，Python，以及 Ruby——还有很多由社区提供的客户端和插件，所有这些可以在[文档](http://www.elasticsearch.org/guide/)中找到。

向Elasticsearch发出的请求的组成部分与其它普通的HTTP请求是一样的：

curl -X<VERB> '<PROTOCOL>://<HOST>:<PORT>/<PATH>?<QUERY\_STRING>' -d '<BODY>'

* VERB HTTP方法：GET, POST, PUT, HEAD, DELETE
* PROTOCOL http或者https协议（只有在Elasticsearch前面有https代理的时候可用）
* HOST Elasticsearch集群中的任何一个节点的主机名，如果是在本地的节点，那么就叫localhost
* PORT Elasticsearch HTTP服务所在的端口，默认为9200
* PATH API路径（例如\_count将返回集群中文档的数量），PATH可以包含多个组件，例如\_cluster/stats或者\_nodes/stats/jvm
* QUERY\_STRING 一些可选的查询请求参数，例如?pretty参数将使请求返回更加美观易读的JSON数据
* BODY 一个JSON格式的请求主体（如果请求需要的话）

举例说明，为了计算集群中的文档数量，我们可以这样做：

curl -XGET 'http://localhost:9200/\_count?pretty' -d '

{

"query": {

"match\_all": {}

}

}

'

Elasticsearch返回一个类似200 OK的HTTP状态码和JSON格式的响应主体（除了HEAD请求）。上面的请求会得到如下的JSON格式的响应主体：

{

"count" : 0,

"\_shards" : {

"total" : 5,

"successful" : 5,

"failed" : 0

}

}

我们看不到HTTP头是因为我们没有让curl显示它们，如果要显示，使用curl命令后跟-i参数:

curl -i -XGET 'localhost:9200/'

对于本书的其余部分，我们将简写curl请求中重复的部分，例如主机名和端口，还有curl命令本身。

一个完整的请求形如：

curl -XGET 'localhost:9200/\_count?pretty' -d '

{

"query": {

"match\_all": {}

}

}'

我们将简写成这样：

GET /\_count

{

"query": {

"match\_all": {}

}

}

事实上，在Sense控制台中也使用了与上面相同的格式。

## 4.面向文档

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

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

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

## 4.2JSON

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

以下使用JSON文档来表示一个用户对象：

{

"email": "john@smith.com",

"first\_name": "John",

"last\_name": "Smith",

"info": {

"bio": "Eco-warrior and defender of the weak",

"age": 25,

"interests": [ "dolphins", "whales" ]

},

"join\_date": "2014/05/01"

}

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

**NOTE**

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

## 5.1开始第一步

我们现在开始进行一个简单教程，它涵盖了一些基本的概念介绍，比如**索引(indexing)**、**搜索(search)**以及**聚合(aggregations)**。通过这个教程，我们可以让你对Elasticsearch能做的事以及其易用程度有一个大致的感觉。

我们接下来将陆续介绍一些术语和基本的概念，但就算你没有马上完全理解也没有关系。我们将在本书的各个章节中更加深入的探讨这些内容。

所以，坐下来，开始以旋风般的速度来感受Elasticsearch的能力吧！

## 让我们建立一个员工目录

假设我们刚好在**Megacorp**工作，这时人力资源部门出于某种目的需要让我们创建一个员工目录，这个目录用于促进人文关怀和用于实时协同工作，所以它有以下不同的需求：

* 数据能够包含多个值的标签、数字和纯文本。
* 检索任何员工的所有信息。
* 支持结构化搜索，例如查找30岁以上的员工。
* 支持简单的全文搜索和更复杂的**短语(phrase)**搜索
* 高亮搜索结果中的关键字
* 能够利用图表管理分析这些数据

## 索引员工文档

我们首先要做的是存储员工数据，每个文档代表一个员工。在Elasticsearch中存储数据的行为就叫做**索引(indexing)**，不过在索引之前，我们需要明确数据应该存储在哪里。

在Elasticsearch中，文档归属于一种**类型(type)**,而这些类型存在于**索引(index)**中，我们可以画一些简单的对比图来类比传统关系型数据库：

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

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

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

### 「索引」含义的区分

你可能已经注意到**索引(index)**这个词在Elasticsearch中有着不同的含义，所以有必要在此做一下区分:

* 索引（名词） 如上文所述，一个**索引(index)**就像是传统关系数据库中的**数据库**，它是相关文档存储的地方，index的复数是**indices**或**indexes**。
* 索引（动词） **「索引一个文档」**表示把一个文档存储到**索引（名词）**里，以便它可以被检索或者查询。这很像SQL中的INSERT关键字，差别是，如果文档已经存在，新的文档将覆盖旧的文档。
* 倒排索引 传统数据库为特定列增加一个索引，例如B-Tree索引来加速检索。Elasticsearch和Lucene使用一种叫做**倒排索引(inverted index)**的数据结构来达到相同目的。

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

我们将会在[倒排索引](https://es.xiaoleilu.com/052_Mapping_Analysis/35_Inverted_index.html)章节中更详细的讨论。

所以为了创建员工目录，我们将进行如下操作：

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

实际上这些都是很容易的（尽管看起来有许多步骤）。我们能通过一个命令执行完成的操作：

PUT /megacorp/employee/1

{

"first\_name" : "John",

"last\_name" : "Smith",

"age" : 25,

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

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

}

我们看到path:/megacorp/employee/1包含三部分信息：

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

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

很简单吧！它不需要你做额外的管理操作，比如创建索引或者定义每个字段的数据类型。我们能够直接索引文档，Elasticsearch已经内置所有的缺省设置，所有管理操作都是透明的。

接下来，让我们在目录中加入更多员工信息：

PUT /megacorp/employee/2

{

"first\_name" : "Jane",

"last\_name" : "Smith",

"age" : 32,

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

"interests": [ "music" ]

}

PUT /megacorp/employee/3

{

"first\_name" : "Douglas",

"last\_name" : "Fir",

"age" : 35,

"about": "I like to build cabinets",

"interests": [ "forestry" ]

}

## 6.1检索文档

现在Elasticsearch中已经存储了一些数据，我们可以根据业务需求开始工作了。第一个需求是能够检索单个员工的信息。

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

GET /megacorp/employee/1

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

{

"\_index" : "megacorp",

"\_type" : "employee",

"\_id" : "1",

"\_version" : 1,

"found" : true,

"\_source" : {

"first\_name" : "John",

"last\_name" : "Smith",

"age" : 25,

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

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

}

}

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

## 简单搜索

GET请求非常简单——你能轻松获取你想要的文档。让我们来进一步尝试一些东西，比如简单的搜索！

我们尝试一个最简单的搜索全部员工的请求：

GET /megacorp/employee/\_search

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

{

"took": 6,

"timed\_out": false,

"\_shards": { ... },

"hits": {

"total": 3,

"max\_score": 1,

"hits": [

{

"\_index": "megacorp",

"\_type": "employee",

"\_id": "3",

"\_score": 1,

"\_source": {

"first\_name": "Douglas",

"last\_name": "Fir",

"age": 35,

"about": "I like to build cabinets",

"interests": [ "forestry" ]

}

},

{

"\_index": "megacorp",

"\_type": "employee",

"\_id": "1",

"\_score": 1,

"\_source": {

"first\_name": "John",

"last\_name": "Smith",

"age": 25,

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

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

}

},

{

"\_index": "megacorp",

"\_type": "employee",

"\_id": "2",

"\_score": 1,

"\_source": {

"first\_name": "Jane",

"last\_name": "Smith",

"age": 32,

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

"interests": [ "music" ]

}

}

]

}

}

**注意**：

响应内容不仅会告诉我们哪些文档被匹配到，而且这些文档内容完整的被包含在其中—我们在给用户展示搜索结果时需要用到的所有信息都有了。

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

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

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

{

...

"hits": {

"total": 2,

"max\_score": 0.30685282,

"hits": [

{

...

"\_source": {

"first\_name": "John",

"last\_name": "Smith",

"age": 25,

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

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

}

},

{

...

"\_source": {

"first\_name": "Jane",

"last\_name": "Smith",

"age": 32,

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

"interests": [ "music" ]

}

}

]

}

}

## 使用DSL语句查询

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

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

GET /megacorp/employee/\_search

{

"query" : {

"match" : {

"last\_name" : "Smith"

}

}

}

这会返回与之前查询相同的结果。你可以看到有些东西改变了，我们不再使用**查询字符串(query string)**做为参数，而是使用请求体代替。这个请求体使用JSON表示，其中使用了match语句（查询类型之一，具体我们以后会学到）。

## 更复杂的搜索

我们让搜索稍微再变的复杂一些。我们依旧想要找到姓氏为“Smith”的员工，但是我们只想得到年龄大于30岁的员工。我们的语句将添加**过滤器(filter)**,它使得我们高效率的执行一个结构化搜索：

GET /megacorp/employee/\_search

{

"query" : {

"filtered" : {

"filter" : {

"range" : {

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

}

},

"query" : {

"match" : {

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

}

}

}

}

}

* <1> 这部分查询属于**区间过滤器(range filter)**,它用于查找所有年龄大于30岁的数据——gt为"greater than"的缩写。
* <2> 这部分查询与之前的match**语句(query)**一致。

现在不要担心语法太多，我们将会在以后详细的讨论。你只要知道我们添加了一个**过滤器(filter)**用于执行区间搜索，然后重复利用了之前的match语句。现在我们的搜索结果只显示了一个32岁且名字是“Jane Smith”的员工：

{

...

"hits": {

"total": 1,

"max\_score": 0.30685282,

"hits": [

{

...

"\_source": {

"first\_name": "Jane",

"last\_name": "Smith",

"age": 32,

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

"interests": [ "music" ]

}

}

]

}

}

## 全文搜索

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

我们将会搜索所有喜欢**“rock climbing”**的员工：

GET /megacorp/employee/\_search

{

"query" : {

"match" : {

"about" : "rock climbing"

}

}

}

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

{

...

"hits": {

"total": 2,

"max\_score": 0.16273327,

"hits": [

{

...

"\_score": 0.16273327, <1>

"\_source": {

"first\_name": "John",

"last\_name": "Smith",

"age": 25,

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

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

}

},

{

...

"\_score": 0.016878016, <2>

"\_source": {

"first\_name": "Jane",

"last\_name": "Smith",

"age": 32,

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

"interests": [ "music" ]

}

}

]

}

}

* <1><2> 结果相关性评分。

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

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

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

## 短语搜索

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

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

GET /megacorp/employee/\_search

{

"query" : {

"match\_phrase" : {

"about" : "rock climbing"

}

}

}

毫无疑问，该查询返回John Smith的文档：

{

...

"hits": {

"total": 1,

"max\_score": 0.23013961,

"hits": [

{

...

"\_score": 0.23013961,

"\_source": {

"first\_name": "John",

"last\_name": "Smith",

"age": 25,

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

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

}

}

]

}

}

## 高亮我们的搜索

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

让我们在之前的语句上增加highlight参数：

GET /megacorp/employee/\_search

{

"query" : {

"match\_phrase" : {

"about" : "rock climbing"

}

},

"highlight": {

"fields" : {

"about" : {}

}

}

}

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

{

...

"hits": {

"total": 1,

"max\_score": 0.23013961,

"hits": [

{

...

"\_score": 0.23013961,

"\_source": {

"first\_name": "John",

"last\_name": "Smith",

"age": 25,

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

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

},

"highlight": {

"about": [

"I love to go <em>rock</em> <em>climbing</em>" <1>

]

}

}

]

}

}

* <1> 原有文本中高亮的片段

你可以在高亮章节阅读更多关于搜索高亮的部分。

## 7.1分析

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

举个例子，让我们找到所有职员中最大的共同点（兴趣爱好）是什么：

GET /megacorp/employee/\_search

{

"aggs": {

"all\_interests": {

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

}

}

}

暂时先忽略语法只看查询结果：

{

...

"hits": { ... },

"aggregations": {

"all\_interests": {

"buckets": [

{

"key": "music",

"doc\_count": 2

},

{

"key": "forestry",

"doc\_count": 1

},

{

"key": "sports",

"doc\_count": 1

}

]

}

}

}

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

GET /megacorp/employee/\_search

{

"query": {

"match": {

"last\_name": "smith"

}

},

"aggs": {

"all\_interests": {

"terms": {

"field": "interests"

}

}

}

}

all\_interests聚合已经变成只包含和查询语句相匹配的文档了：

...

"all\_interests": {

"buckets": [

{

"key": "music",

"doc\_count": 2

},

{

"key": "sports",

"doc\_count": 1

}

]

}

聚合也允许分级汇总。例如，让我们统计每种兴趣下职员的平均年龄：

GET /megacorp/employee/\_search

{

"aggs" : {

"all\_interests" : {

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

"aggs" : {

"avg\_age" : {

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

}

}

}

}

}

虽然这次返回的聚合结果有些复杂，但任然很容易理解：

...

"all\_interests": {

"buckets": [

{

"key": "music",

"doc\_count": 2,

"avg\_age": {

"value": 28.5

}

},

{

"key": "forestry",

"doc\_count": 1,

"avg\_age": {

"value": 35

}

},

{

"key": "sports",

"doc\_count": 1,

"avg\_age": {

"value": 25

}

}

]

}

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

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

## 8教程小结

希望这个简短的教程能够很好的描述Elasticsearch的功能。当然这只是一些皮毛，为了保持简短，还有很多的特性未提及——像推荐、定位、渗透、模糊以及部分匹配等。但这也突出了构建高级搜索功能是多么的容易。无需配置，只需要添加数据然后开始搜索！

可能有些语法让你觉得有些困惑，或者在微调方面有些疑问。那么，本书的其余部分将深入这些问题的细节，让你全面了解Elasticsearch的工作过程。

1.9分布式

## 9分布式的特性

在章节的开始我们提到Elasticsearch可以扩展到上百（甚至上千）的服务器来处理PB级的数据。然而我们的教程只是给出了一些使用Elasticsearch的例子，并未涉及相关机制。Elasticsearch为分布式而生，而且它的设计隐藏了分布式本身的复杂性。

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

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

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

当你阅读本书时，你可以遇到关于Elasticsearch分布式特性的补充章节。这些章节将教给你如何扩展集群和故障转移，如何处理文档存储，如何执行分布式搜索，分片是什么以及如何工作。

这些章节不是必读的——不懂这些内部机制也可以使用Elasticsearch的。但是这些能够帮助你更深入和完整的了解Elasticsearch。你可以略读它们，然后在你需要更深入的理解时再回头翻阅。