

## 课程介绍

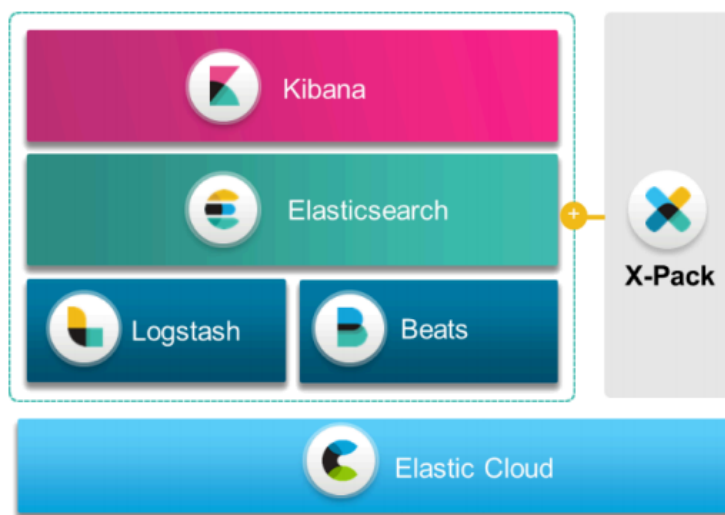
- Elastic Stack简介
- Elasticsearch的介绍与安装
- Elasticsearch的快速入门
- Elasticsearch的核心讲解
- 中文分词
- 全文搜索
- Elasticsearch集群
- Java客户端讲解

## 1、Elastic Stack简介

如果你没有听说过Elastic Stack，那你一定听说过ELK，实际上ELK是三款软件的简称，分别是Elasticsearch、Logstash、Kibana组成，在发展的过程中，又有新成员Beats的加入，所以就形成了Elastic Stack。所以说，ELK是旧的称呼，Elastic Stack是新的名字。

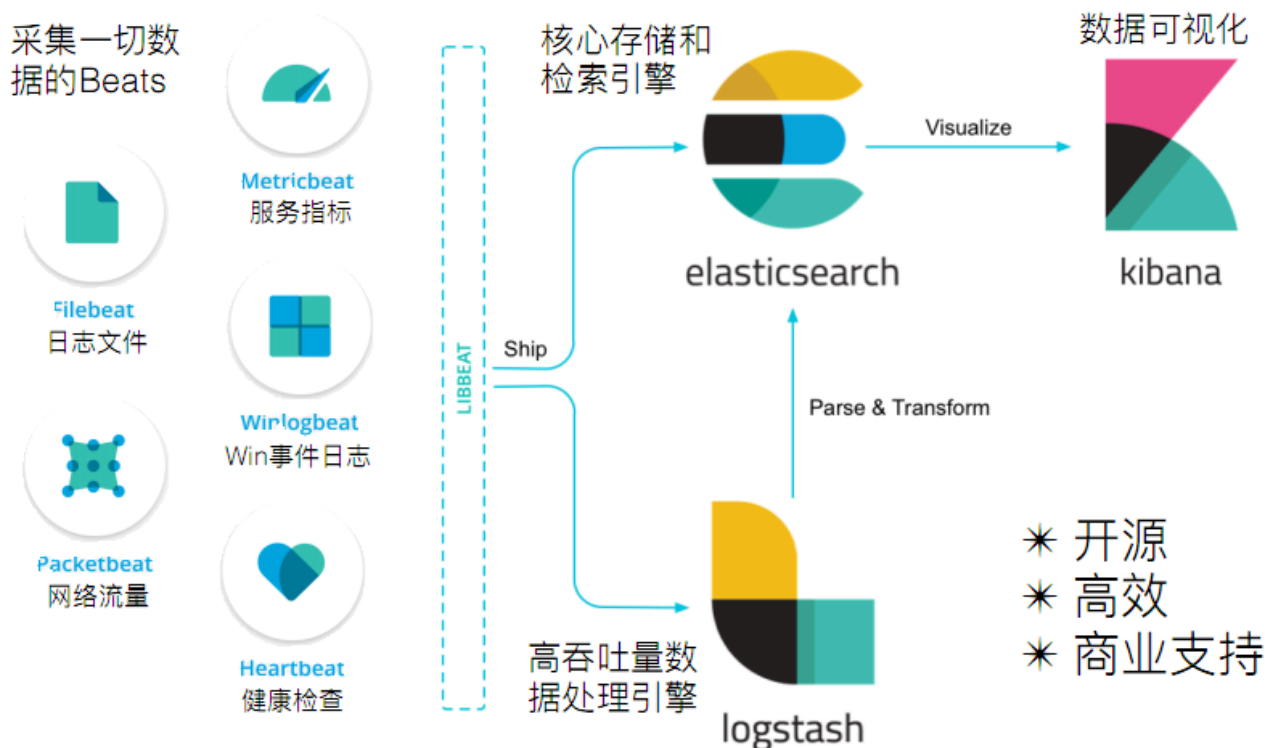


## The Brand New ElasticStack



全系的Elastic Stack技术栈包括：

# ElasticStack的组成



## Elasticsearch

Elasticsearch 基于java，是个开源分布式搜索引擎，它的特点有：分布式，零配置，自动发现，索引自动分片，索引副本机制，restful风格接口，多数据源，自动搜索负载等。

## Logstash

Logstash 基于java，是一个开源的用于收集、分析和存储日志的工具。

## Kibana

Kibana 基于nodejs，也是一个开源和免费的工具，Kibana可以为 Logstash 和 ElasticSearch 提供的日志分析友好的 Web 界面，可以汇总、分析和搜索重要数据日志。

## Beats

Beats是elastic公司开源的一款采集系统监控数据的代理agent，是在被监控服务器上以客户端形式运行的数据收集器的统称，可以直接把数据发送给Elasticsearch或者通过Logstash发送给Elasticsearch，然后进行后续的数据分析活动。

Beats由如下组成:

- Packetbeat：是一个网络数据包分析器，用于监控、收集网络流量信息，Packetbeat嗅探服务器之间的流量，解析应用层协议，并关联到消息的处理，其支持ICMP (v4 and v6)、DNS、HTTP、Mysql、PostgreSQL、Redis、MongoDB、Memcache等协议；
- Filebeat：用于监控、收集服务器日志文件，其已取代 logstash forwarder；
- Metricbeat：可定期获取外部系统的监控指标信息，其可以监控、收集 Apache、HAProxy、MongoDB MySQL、Nginx、PostgreSQL、Redis、System、Zookeeper等服务；

- Winlogbeat：用于监控、收集Windows系统的日志信息；

## 2、Elasticsearch

### 2.1、简介

ElasticSearch是一个基于Lucene的搜索服务器。它提供了一个分布式多用户能力的全文搜索引擎，基于RESTful web接口。Elasticsearch是用Java开发的，并作为Apache许可条款下的开放源码发布，是当前流行的企业级搜索引擎。设计用于云计算中，能够达到实时搜索，稳定，可靠，快速，安装使用方便。

我们建立一个网站或应用程序，并要添加搜索功能，但是想要完成搜索工作的创建是非常困难的。我们希望搜索解决方案要运行速度快，我们希望能有一个零配置和一个完全免费的搜索模式，我们希望能够简单地使用JSON通过HTTP来索引数据，我们希望我们的搜索服务器始终可用，我们希望能够从一台开始并扩展到数百台，我们要实时搜索，我们要简单的多租户，我们希望建立一个云的解决方案。因此我们利用Elasticsearch来解决所有这些问题及可能出现的更多其它问题。

官网：<https://www.elastic.co/cn/products/elasticsearch>

### Elastic Stack 的核心

Elasticsearch 是一个分布式、RESTful 风格的搜索和数据分析引擎，能够解决不断涌现出的各种用例。作为 Elastic Stack 的核心，它集中存储您的数据，帮助您发现意料之中以及意料之外的情况。



### 2.2、安装

#### 2.2.1、版本说明

Elasticsearch的发展是非常快速的，所以在ES5.0之前，ELK的各个版本都不统一，出现了版本号混乱的状态，所以从5.0开始，所有Elastic Stack中的项目全部统一版本号。目前最新版本是6.5.4，我们将基于这一版本进行学习。



### 2.2.2、下载

地址：<https://www.elastic.co/cn/downloads/elasticsearch>

GA RELEASE

PREVIEW RELEASE

Version:

6.5.4

Release date:

December 19, 2018

License:

Elastic License

Downloads:

⬇️ WINDOWS sha

⬇️ DEB sha

⬇️ MSI (BETA) sha

⬇️ MACOS/LINUX sha

⬇️ RPM sha

或者，使用资料中提供的已下载好的安装包。

### 2.2.3、单机版安装

```
1 #创建elsearch用户，Elasticsearch不支持root用户运行
2 useradd elsearch
3
4 #解压安装包
5 tar -xvf elasticsearch-6.5.4.tar.gz -C /itcast/es/
```



```
6
7 #修改配置文件
8 vim conf/elasticsearch.yml
9 network.host: 0.0.0.0 #设置ip地址，任意网络均可访问
10
11 #说明：在Elasticsearch中如果，network.host不是localhost或者127.0.0.1的话，就会认为是生产环境，
12 #会对环境的要求比较高，我们的测试环境不一定能够满足，一般情况下需要修改2处配置，如下：
13 #1：修改jvm启动参数
14 vim conf/jvm.options
15 -Xms128m #根据自己机器情况修改
16 -Xmx128m
17 #2：一个进程在VMAs(虚拟内存区域)创建内存映射最大数量
18 vim /etc/sysctl.conf
19 vm.max_map_count=655360
20
21 sysctl -p #配置生效
22
23 #启动ES服务
24 su - elsearch
25 cd bin
26 ./elasticsearch 或 ./elasticsearch -d #后台启动
27
28 #通过访问进行测试，看到如下信息，就说明ES启动成功了
29 {
30     "name": "dsQV6I8",
31     "cluster_name": "elasticsearch",
32     "cluster_uuid": "v5GPTWAtT5emxFdjigFg-w",
33     "version": {
34         "number": "6.5.4",
35         "build_flavor": "default",
36         "build_type": "tar",
37         "build_hash": "d2ef93d",
38         "build_date": "2018-12-17T21:17:40.758843z",
39         "build_snapshot": false,
40         "lucene_version": "7.5.0",
41         "minimum_wire_compatibility_version": "5.6.0",
42         "minimum_index_compatibility_version": "5.0.0"
43     },
44     "tagline": "You Know, for Search"
45 }
46
47 #停止服务
48 root@itcast:~# jps
49 68709 jps
50 68072 Elasticsearch
51
52 kill 68072 #通过kill结束进程
```

```
1 #启动出错，环境：Centos6
2 [1]: max file descriptors [4096] for elasticsearch process is too low, increase to at
3 least [65536]
4 #解决：切换到root用户，编辑limits.conf 添加类似如下内容
5 vi /etc/security/limits.conf
```



```
5 添加如下内容：
6 * soft nofile 65536
7 * hard nofile 131072
8 * soft nproc 2048
9 * hard nproc 4096
10
11
12 [2]: max number of threads [1024] for user [elasticsearch] is too low, increase to at least
    [4096]
13 #解决：切换到root用户，进入limits.d目录下修改配置文件。
14 vi /etc/security/limits.d/90-nproc.conf
15 #修改如下内容：
16 * soft nproc 1024
17 #修改为
18 * soft nproc 4096
19
20 [3]: system call filters failed to install; check the logs and fix your configuration
    or disable system call filters at your own risk
21
22 #解决：Centos6不支持SecComp，而ES5.2.0默认bootstrap.system_call_filter为true
23 vim config/elasticsearch.yml
24 添加：
25 bootstrap.system_call_filter: false
```

## 2.2.4、elasticsearch-head

由于ES官方并没有为ES提供界面管理工具，仅仅是提供了后台的服务。elasticsearch-head是一个为ES开发的一个页面客户端工具，其源码托管于GitHub，地址为：<https://github.com/mobz/elasticsearch-head>

head提供了4种安装方式：

- 源码安装，通过npm run start启动（不推荐）
- 通过docker安装（推荐）
- 通过chrome插件安装（推荐）
- 通过ES的plugin方式安装（不推荐）

通过docker安装

```
1 #拉取镜像
2 docker pull mobz/elasticsearch-head:5
3
4 #创建容器
5 docker create --name elasticsearch-head -p 9100:9100 mobz/elasticsearch-head:5
6
7 #启动容器
8 docker start elasticsearch-head
```

通过浏览器进行访问：



注意：

由于前后端分离开发，所以会存在跨域问题，需要在服务端做CORS的配置，如下：

```
vim elasticsearch.yml
```

```
http.cors.enabled: true http.cors.allow-origin: "*" 
```

通过chrome插件的方式安装不存在该问题。

chrome插件的方式安装

打开chrome的应用商店，即可安装<https://chrome.google.com/webstore/detail/elasticsearch-head/ffmkiejjmecolpfloofpjologoblkegm>



建议：推荐使用chrome插件的方式安装，如果网络环境不允许，就采用其它方式安装。

## 2.3、基本概念

### 索引

- 索引 ( index ) 是Elasticsearch对逻辑数据的逻辑存储，所以它可以分为更小的部分。
- 可以把索引看成关系型数据库的表，索引的结构是为快速有效的全文索引准备的，特别是它不存储原始值。
- Elasticsearch可以把索引存放在一台机器或者分散在多台服务器上，每个索引有一或多个分片 ( shard )，每个分片可以有多个副本 ( replica )。

### 文档

- 存储在Elasticsearch中的主要实体叫文档 ( document )。用关系型数据库来类比的话，一个文档相当于数据库表中的一行记录。

- Elasticsearch和MongoDB中的文档类似，都可以有不同的结构，但Elasticsearch的文档中，相同字段必须有相同类型。
- 文档由多个字段组成，每个字段可能多次出现在一个文档里，这样的字段叫多值字段（multivalued）。
- 每个字段的类型，可以是文本、数值、日期等。字段类型也可以是复杂类型，一个字段包含其他子文档或者数组。

#### 映射

- 所有文档写进索引之前都会先进行分析，如何将输入的文本分割为词条、哪些词条又会被过滤，这种行为叫做映射（mapping）。一般由用户自己定义规则。

#### 文档类型

- 在Elasticsearch中，一个索引对象可以存储很多不同用途的对象。例如，一个博客应用程序可以保存文章和评论。
- 每个文档可以有不同的结构。
- 不同的文档类型不能为相同的属性设置不同的类型。例如，在同一索引中的所有文档类型中，一个叫title的字段必须具有相同的类型。

## 2.4、RESTful API

在Elasticsearch中，提供了功能丰富的RESTful API的操作，包括基本的CRUD、创建索引、删除索引等操作。

### 2.4.1、创建非结构化索引

在Lucene中，创建索引是需要定义字段名称以及字段的类型的，在Elasticsearch中提供了非结构化的索引，就是不需要创建索引结构，即可写入数据到索引中，实际上在Elasticsearch底层会进行结构化操作，此操作对用户是透明的。

创建空索引：

```
1 PUT /haoke
2
3 {
4   "settings": {
5     "index": {
6       "number_of_shards": "2", #分片数
7       "number_of_replicas": "0" #副本数
8     }
9   }
10 }
11
12 #删除索引
13 DELETE /haoke
14 {
15   "acknowledged": true
16 }
```





## 2.4.2、插入数据

URL规则：

POST /{索引}/{类型}/{id}

```
1 POST /haoke/user/1001
2 #数据
3 {
4   "id":1001,
5   "name":"张三",
6   "age":20,
7   "sex":"男"
8 }
9
10 #响应
11 {
12   "_index": "haoke",
13   "_type": "user",
14   "_id": "1",
15   "_version": 1,
16   "result": "created",
17   "_shards": {
18     "total": 1,
19     "successful": 1,
20     "failed": 0
21   },
22   "_seq_no": 0,
23   "_primary_term": 1
24 }
```

The screenshot shows the Elasticsearch web interface. At the top, there's a header with the Elasticsearch logo, a URL bar showing 'http://172.16.55.185:9200/', and a '连接' (Connect) button. Below the header, there's a navigation bar with tabs: '概览' (Overview), '索引' (Index), '数据浏览' (Data Browser), '基本查询' (Basic Query), and '复合查询' (Composite Query). The '数据浏览' tab is selected. On the left, there's a sidebar with '所有索引' (All Indexes) and a list of indexes: 'haoke' and 'user'. The 'haoke' index is selected. Below the sidebar, there's a table showing search results. The table has columns: '\_index', '\_type', '\_id', '\_score', 'id', 'name', 'age', and 'sex'. The first row shows a result for index 'haoke', type 'user', id '1001', with a score of 1. The name is '张三', age is '20', and sex is '男'. The table is highlighted with a red border.

说明：非结构化的索引，不需要事先创建，直接插入数据默认创建索引。

不指定id插入数据：

```
1 POST /haoke/user/
2 {
3   "id":1002,
4   "name":"张三",
5   "age":20,
6   "sex":"男"
7 }
```

The screenshot shows the Elasticsearch web interface. At the top, there's a header with the Elasticsearch logo, a URL bar showing 'http://172.16.55.185:9200/', and a '连接' (Connect) button. Below the header, there's a navigation bar with tabs: '概览' (Overview), '索引' (Index), '数据浏览' (Data Browser), '基本查询' (Basic Query), and '复合查询' (Composite Query). The '数据浏览' tab is selected. On the left, there's a sidebar with '所有索引' (All Indexes) and a list of indexes: 'haoke' and 'user'. The 'haoke' index is selected. Below the sidebar, there's a table showing search results. The table has columns: '\_index', '\_type', '\_id', '\_score', 'id', 'name', 'age', and 'sex'. The first row shows a result for index 'haoke', type 'user', id '1001', with a score of 1. The name is '张三', age is '20', and sex is '男'. The second row shows a result for index 'haoke', type 'user', id 'BbPe\_WcB9cFOnF3uebvr', with a score of 1. The name is '张三', age is '20', and sex is '男'. The second row is highlighted with a red border and a red arrow pointing to the id field with the text '自动生成id' (Auto-generated id).

### 2.4.3、更新数据

在Elasticsearch中，文档数据是不为修改的，但是可以通过覆盖的方式进行更新。

```
1 PUT /haoke/user/1001
2 {
3   "id":1001,
4   "name":"张三",
5   "age":21,
6   "sex":"女"
7 }
8
9
```

更新结果如下：



查询 2 个分片中用的 2 个, 2 命中, 耗时 0.002 秒

_index	_type	_id	_score ▲	id	name	age	sex
haoke	user	BbPe_WcB9cFOnF3uebvr	1	1002	李四	40	男
haoke	user	1001	1	1001	张三	21	女

可以看到数据已经被覆盖了。

问题来了，可以局部更新吗？-- 可以的。

前面不是说，文档数据不能更新吗？其实是这样的：

在内部，依然会查询到这个文档数据，然后进行覆盖操作，步骤如下：

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

示例：

```
1 #注意：这里多了_update标识
2 POST /haoke/user/1001/_update
3
4 {
5   "doc":{
6     "age":23
7   }
8 }
9
```



查询 2 个分片中用的 2 个, 2 命中, 耗时 0.003 秒

_index	_type	_id	_score ▲	id	name	age	sex
haoke	user	BbPe_WcB9cFOnF3uebvr	1	1002	李四	40	男
haoke	user	1001	1	1001	张三	23	女

可以看到数据已经被局部更新了。

#### 2.4.4、删除数据

在Elasticsearch中，删除文档数据，只需要发起DELETE请求即可。

```
1 | DELETE /haoke/user/1001
```



```
Raw  JSON  Response
Copy to clipboard  Save as file
{
  _index: "haoke"
  _type: "user"
  _id: "1001"
  _version: 4
  result: "deleted"
  _shards: {
    total: 1
    successful: 1
    failed: 0
  }
  _seq_no: 15
  _primary_term: 1
}
```

需要注意的是，result表示已经删除，version也更加了。

如果删除一条不存在的数据，会响应404：



说明：

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

## 2.4.5、搜索数据

根据id搜索数据

```
1 GET /haoke/user/BbPe_wCB9cF0nF3uebvr
2
3 #返回的数据如下
4 {
5   "_index": "haoke",
6   "_type": "user",
7   "_id": "BbPe_wCB9cF0nF3uebvr",
8   "_version": 8,
9   "found": true,
10  "_source": { #原始数据在这里
11    "id": 1002,
```



```
12     "name": "李四",
13     "age": 40,
14     "sex": "男"
15 }
16 }
```

#### 搜索全部数据

```
1 GET /haoke/user/_search
```

响应：（默认返回10条数据）

```
1 {
2   "took": 26,
3   "timed_out": false,
4   "_shards": {
5     "total": 2,
6     "successful": 2,
7     "skipped": 0,
8     "failed": 0
9   },
10  "hits": {
11    "total": 4,
12    "max_score": 1,
13    "hits": [
14      {
15        "_index": "haoke",
16        "_type": "user",
17        "_id": "BbPe_wcB9cF0nF3uebvr",
18        "_score": 1,
19        "_source": {
20          "id": 1002,
21          "name": "李四",
22          "age": 40,
23          "sex": "男"
24        }
25      },
26      {
27        "_index": "haoke",
28        "_type": "user",
29        "_id": "1001",
30        "_score": 1,
31        "_source": {
32          "id": 1001,
33          "name": "张三",
34          "age": 20,
35          "sex": "男"
36        }
37      },
38      {
39        "_index": "haoke",
40        "_type": "user",
```



```
41         "_id": "1003",
42         "_score": 1,
43         "_source": {
44             "id": 1003,
45             "name": "王五",
46             "age": 30,
47             "sex": "男"
48         }
49     },
50     {
51         "_index": "haoke",
52         "_type": "user",
53         "_id": "1004",
54         "_score": 1,
55         "_source": {
56             "id": 1004,
57             "name": "赵六",
58             "age": 30,
59             "sex": "男"
60         }
61     }
62 ]
63 }
64 }
```

#### 关键字搜索数据

```
1 #查询年龄等于20的用户
2 GET /haoke/user/_search?q=age:20
3
```

结果：





```
Raw  JSON  Response
Copy to clipboard  Save as file
{
  took: 4
  timed_out: false
  _shards: {
    total: 2
    successful: 2
    skipped: 0
    failed: 0
  }
  -hits: {
    total: 1
    max_score: 1
    -hits: [1]
      -0: {
        _index: "haoke"
        _type: "user"
        _id: "1001"
        _score: 1
        -_source: {
          id: 1001
          name: "张三"
          age: 20
          sex: "男"
        }
      }
    ]
  }
}
```

## 2.4.6、DSL搜索

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。

```
1 POST /haoke/user/_search
2
3 #请求体
4 {
5   "query" : {
6     "match" : { #match只是查询的一种
7       "age" : 20
8     }
9   }
10 }
```

响应数据：



```
Raw  JSON  Response
Copy to clipboard  Save as file
{
  took: 5
  timed_out: false
  _shards: {
    total: 2
    successful: 2
    skipped: 0
    failed: 0
  }
  -hits: {
    total: 1
    max_score: 1
    -hits: [1]
      -0: {
        _index: "haoke"
        _type: "user"
        _id: "1001"
        _score: 1
        -_source: {
          id: 1001
          name: "张三"
          age: 20
          sex: "男"
        }
      }
    ]
  }
}
```

实现：查询年龄大于30岁的男性用户。

现有数据：

查询 2 个分片中用的 2 个, 4 命中, 耗时 0.004 秒

_index	_type	_id	_score ▲	id	name	age	sex
haoke	user	BbPe_WcB9cFOnF3uebvr	1	1002	李四	40	男
haoke	user	1001	1	1001	张三	20	男
haoke	user	1003	1	1003	王五	30	男
haoke	user	1004	1	1004	赵六	30	女

```
1 POST /haoke/user/_search
2 #请求数据
3 {
4   "query": {
5     "bool": {
6       "filter": {
```



```
7      "range": {  
8          "age": {  
9              "gt": 30  
10         }  
11     },  
12 },  
13 "must": {  
14     "match": {  
15         "sex": "男"  
16     }  
17 }  
18 }  
19 }  
20 }
```

查询结果：

Raw JSON Response

Copy to clipboard Save as file

```
{  
  took: 35  
  timed_out: false  
  _shards: {  
    total: 2  
    successful: 2  
    skipped: 0  
    failed: 0  
  }  
  -hits: {  
    total: 1  
    max_score: 0.35667494  
    -hits: [1]  
      -0: {  
        _index: "haoke"  
        _type: "user"  
        _id: "BbPe_WcB9cFOnF3uebvr"  
        _score: 0.35667494  
        -_source: {  
          id: 1002  
          name: "李四"  
          age: 40  
          sex: "男"  
        }  
      }  
    ]  
  }  
}
```

全文搜索



```
1 POST /haoke/user/_search
2 #请求数据
3 {
4     "query": {
5         "match": {
6             "name": "张三 李四"
7         }
8     }
9 }
```

```
-hits: {
  total: 2
  max_score: 2.4079456
  -hits: [2]
    -0: {
      _index: "haoke"
      _type: "user"
      _id: "BbPe_WcB9cFOnF3uebvr"
      _score: 2.4079456
      -_source: {
        id: 1002
        name: "李四"
        age: 40
        sex: "男"
      }
    }
    -1: {
      _index: "haoke"
      _type: "user"
      _id: "1001"
      _score: 2.4079456
      -_source: {
        id: 1001
        name: "张三"
        age: 20
        sex: "男"
      }
    }
  }
}
```

## 2.4.7、高亮显示

```
1 POST /haoke/user/_search
2
3 {
4     "query": {
5         "match": {
6             "name": "张三 李四"
```



```
7     }
8     },
9     "highlight": {
10         "fields": {
11             "name": {}
12         }
13     }
14 }
```

```
-hits: {
  total: 2
  max_score: 2.4079456
  -hits: [2]
    -0: {
      _index: "haoke"
      _type: "user"
      _id: "BbPe_WcB9cF0nF3uebvr"
      _score: 2.4079456
      -_source: {
        id: 1002
        name: "李四"
        age: 40
        sex: "男"
      }
      -highlight: {
        -name: [1]
          0: "<em>李</em><em>四</em>"
        }
      }
    -1: {
      _index: "haoke"
      _type: "user"
      _id: "1001"
      _score: 2.4079456
      -_source: {
        id: 1001
        name: "张三"
        age: 20
        sex: "男"
      }
      -highlight: {
        -name: [1]
          0: "<em>张</em><em>三</em>"
        }
      }
    }
  }
}
```

## 2.4.8、聚合

在Elasticsearch中，支持聚合操作，类似SQL中的group by操作。

```
1 POST /haoke/user/_search
2
3 {
4   "aggs": {
5     "all_interests": {
6       "terms": {
7         "field": "age"
8       }
9     }
10  }
11 }
```

结果：

```
-aggregations: {
  -all_interests: {
    doc_count_error_upper_bound: 0
    sum_other_doc_count: 0
    -buckets: [3]
      -0: {
        key: 30
        doc_count: 2
      }
      -1: {
        key: 20
        doc_count: 1
      }
      -2: {
        key: 40
        doc_count: 1
      }
    }
  }
}
```

从结果可以看出，年龄30的有2条数据，20的有一条，40的一条。

## 3、核心详解

### 3.1、文档

在Elasticsearch中，文档以JSON格式进行存储，可以是复杂结构，如：

```
1 {
2   "_index": "haoke",
3   "_type": "user",
4   "_id": "1005",
5   "_version": 1,
6   "_score": 1,
```

```
7      "_source": {  
8          "id": 1005,  
9          "name": "孙七",  
10         "age": 37,  
11         "sex": "女",  
12         "card": {  
13             "card_number": "123456789"  
14         }  
15     }  
16 }
```

其中，card是一个复杂对象，嵌套的Card对象。

#### 元数据 ( metadata )

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

节点	说明
<code>_index</code>	文档存储的地方
<code>_type</code>	文档代表的对象的类
<code>_id</code>	文档的唯一标识

#### `_index`

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

##### 提示：

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

#### `_type`

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

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

每个**类型(type)**都有自己的**映射(mapping)**或者结构定义，就像传统数据库表中的列一样。所有类型下的文档被存储在同一个索引下，但是类型的**映射(mapping)**会告诉Elasticsearch不同的文档如何被索引。

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

#### `_id`

**id**仅仅是一个字符串，它与 `_index` 和 `_type` 组合时，就可以在Elasticsearch中唯一标识一个文档。当创建一个文档，你可以自定义 `_id`，也可以让Elasticsearch帮你自动生成（32位长度）。

## 3.2、查询响应

### 3.2.1、pretty

可以在查询url后面添加pretty参数，使得返回的json更易查看。



### 3.2.2、指定响应字段

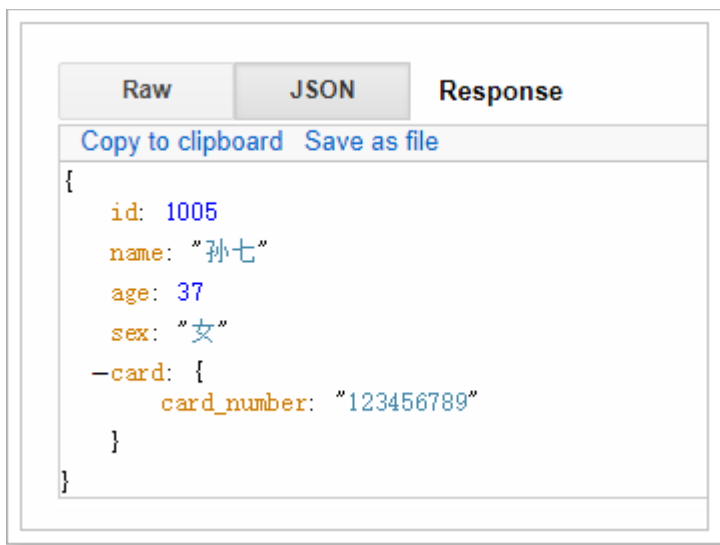
在响应的数据中，如果我们不需要全部的字段，可以指定某些需要的字段进行返回。

```
1 GET /haoke/user/1005?_source=id,name
2 #响应
3 {
4     "_index": "haoke",
5     "_type": "user",
6     "_id": "1005",
7     "_version": 1,
8     "found": true,
9     "_source": {
10         "name": "孙七",
11         "id": 1005
12     }
13 }
```

如不需要返回元数据，仅仅返回原始数据，可以这样：

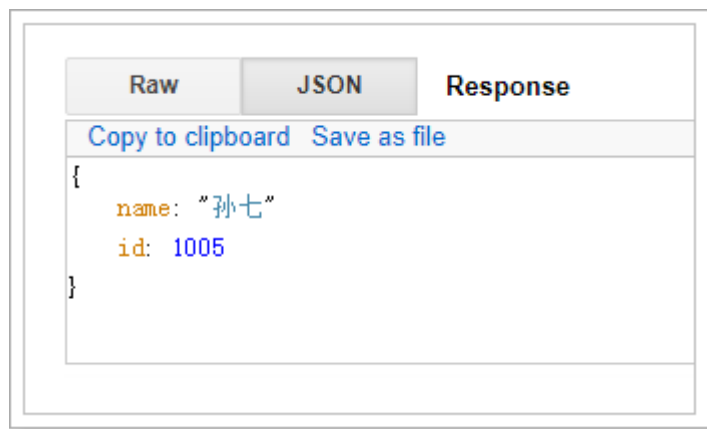
```
1 GET /haoke/user/1005/_source
```





还可以这样：

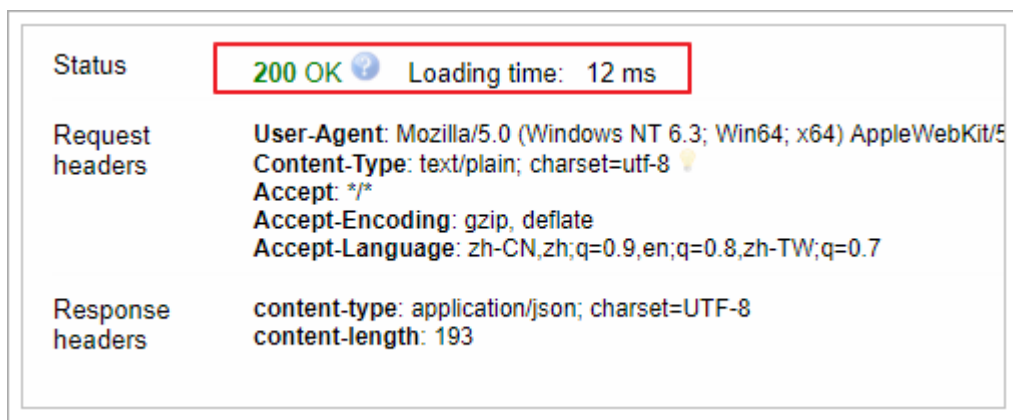
```
1 | GET /haoke/user/1005/_source?_source=id,name
```



### 3.3、判断文档是否存在

如果我们只需要判断文档是否存在，而不是查询文档内容，那么可以这样：

```
1 | HEAD /haoke/user/1005
```



1 | **HEAD** /haoke/user/1006

Status	404 Not Found ? Loading time: 8 ms
Request headers	User-Agent: Mozilla/5.0 (Windows NT 6.3; Win64; x64) AppleWebKit/537.36 (KHTML, Content-Type: text/plain; charset=utf-8 Accept: */* Accept-Encoding: gzip, deflate Accept-Language: zh-CN,zh;q=0.9,en;q=0.8,zh-TW;q=0.7
Response headers	content-type: application/json; charset=UTF-8 content-length: 60

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

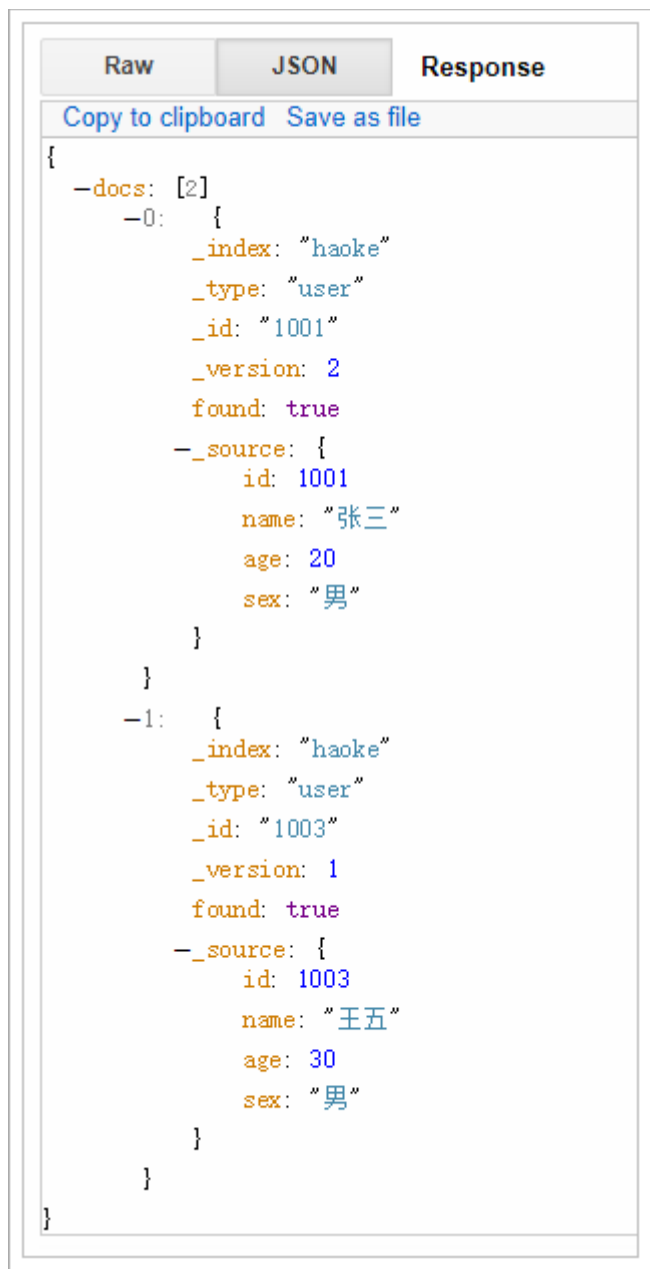
## 3.4、批量操作

有些情况下可以通过批量操作以减少网络请求。如：批量查询、批量插入数据。

### 3.4.1、批量查询

```
1 POST /haoke/user/_mget
2
3 {
4   "ids" : [ "1001", "1003" ]
5 }
```

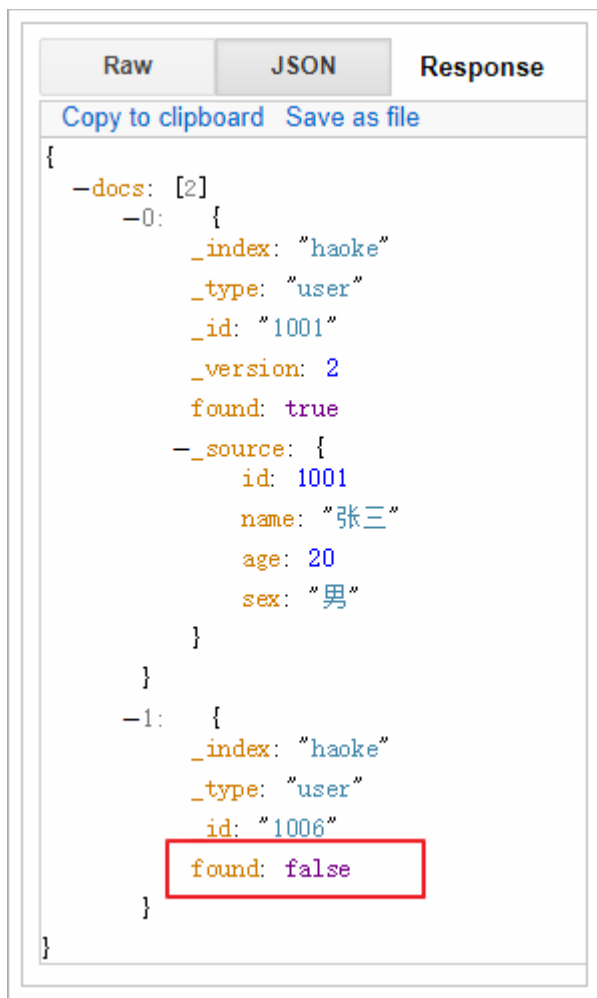
结果：



如果，某一条数据不存在，不影响整体响应，需要通过found的值进行判断是否查询到数据。

```
1 POST /haoke/user/_mget
2
3 {
4   "ids" : [ "1001", "1006" ]
5 }
```

结果：



### 3.4.2、\_bulk操作

在Elasticsearch中，支持批量的插入、修改、删除操作，都是通过\_bulk的api完成的。

请求格式如下：（请求格式不同寻常）

```
1 { action: { metadata }}\n
2 { request body      }\n
3 { action: { metadata }}\n
4 { request body      }\n
5 ...
```

批量插入数据：

```
1 {"create":{"_index":"haoke","_type":"user","_id":2001}}
2 {"id":2001,"name":"name1","age": 20,"sex": "男"}
3 {"create":{"_index":"haoke","_type":"user","_id":2002}}
4 {"id":2002,"name":"name2","age": 20,"sex": "男"}
5 {"create":{"_index":"haoke","_type":"user","_id":2003}}
6 {"id":2003,"name":"name3","age": 20,"sex": "男"}
7
```

注意最后一行的回车。



http://172.16.55.185:9200/haoke/user/\_bulk

☐ GET ☒ POST ☐ PUT ☐ PATCH ☐ DELETE ☐ HEAD ☐ OPTIONS ☐

Raw Form Headers

Raw Form Files (0) Payload

Encode payload Decode payload

```
{
  "create": { "_index": "haoke", "_type": "user", "_id": 2001 }
  { "id": 2001, "name": "name1", "age": 20, "sex": "男" }
  "create": { "_index": "haoke", "_type": "user", "_id": 2002 }
  { "id": 2002, "name": "name2", "age": 20, "sex": "男" }
  "create": { "_index": "haoke", "_type": "user", "_id": 2003 }
  { "id": 2003, "name": "name3", "age": 20, "sex": "男" }
}
```

响应结果：

```
1 {
2   "took": 17,
3   "errors": false,
4   "items": [
5     {
6       "create": {
7         "_index": "haoke",
8         "_type": "user",
9         "_id": "2001",
10        "_version": 1,
11        "result": "created",
12        "_shards": {
13          "total": 1,
14          "successful": 1,
15          "failed": 0
16        },
17        "_seq_no": 24,
18        "_primary_term": 1,
19        "status": 201
20      }
21    },
22    {
23      "create": {
24        "_index": "haoke",
25        "_type": "user",
26        "_id": "2002",
```



```
27         "_version": 1,
28         "result": "created",
29         "_shards": {
30             "total": 1,
31             "successful": 1,
32             "failed": 0
33         },
34         "_seq_no": 0,
35         "_primary_term": 1,
36         "status": 201
37     }
38 },
39 {
40     "create": {
41         "_index": "haoke",
42         "_type": "user",
43         "_id": "2003",
44         "_version": 1,
45         "result": "created",
46         "_shards": {
47             "total": 1,
48             "successful": 1,
49             "failed": 0
50         },
51         "_seq_no": 1,
52         "_primary_term": 1,
53         "status": 201
54     }
55 }
56 ]
57 }
```

批量删除：

```
1 {"delete":{"_index":"haoke","_type":"user","_id":2001}}
2 {"delete":{"_index":"haoke","_type":"user","_id":2002}}
3 {"delete":{"_index":"haoke","_type":"user","_id":2003}}
4
```

由于delete没有请求体，所以，action的下一行直接就是下一个action。



▶

☐ GET ☒ POST ☐ PUT ☐ PATCH ☐ DELETE ☐ HEAD ☐ OPTIONS

Raw Form Headers

Raw Form Files (0) Payload

Encode payload Decode payload

```
{
  "delete": {
    "_index": "haoke",
    "_type": "user",
    "_id": "2001"
  },
  "delete": {
    "_index": "haoke",
    "_type": "user",
    "_id": "2002"
  },
  "delete": {
    "_index": "haoke",
    "_type": "user",
    "_id": "2003"
  }
}
```

```
1 {
2   "took": 3,
3   "errors": false,
4   "items": [
5     {
6       "delete": {
7         "_index": "haoke",
8         "_type": "user",
9         "_id": "2001",
10        "_version": 2,
11        "result": "deleted",
12        "_shards": {
13          "total": 1,
14          "successful": 1,
15          "failed": 0
16        },
17        "_seq_no": 25,
18        "_primary_term": 1,
19        "status": 200
20      }
21    },
22    {
23      "delete": {
24        "_index": "haoke",
25        "_type": "user",
26        "_id": "2002",
27        "_version": 2,
28        "result": "deleted",
29        "_shards": {
30          "total": 1,
31          "successful": 1,
```



```
32         "failed": 0
33     },
34     "_seq_no": 2,
35     "_primary_term": 1,
36     "status": 200
37 }
38 },
39 {
40     "delete": {
41         "_index": "haoke",
42         "_type": "user",
43         "_id": "2003",
44         "_version": 2,
45         "result": "deleted",
46         "_shards": {
47             "total": 1,
48             "successful": 1,
49             "failed": 0
50         },
51         "_seq_no": 3,
52         "_primary_term": 1,
53         "status": 200
54     }
55 }
56 ]
57 }
```

其他操作就类似了。

一次请求多少性能最高？

- 整个批量请求需要被加载到接受我们请求节点的内存里，所以请求越大，给其它请求可用的内存就越小。有一个最佳的bulk请求大小。超过这个大小，性能不再提升而且可能降低。
- 最佳大小，当然并不是一个固定的数字。它完全取决于你的硬件、你文档的大小和复杂度以及索引和搜索的负载。
- 幸运的是，这个最佳点(sweetspot)还是容易找到的：试着批量索引标准的文档，随着大小的增长，当性能开始降低，说明你每个批次的大小太大了。开始的数量可以在1000~5000个文档之间，如果你的文档非常大，可以使用较小的批次。
- 通常着眼于你请求批次的物理大小是非常有用的。一千个1kB的文档和一千个1MB的文档大不相同。一个好的批次最好保持在5-15MB大小间。

## 3.5、分页

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

```
1 size: 结果数，默认10
2 from: 跳过开始的结果数，默认0
```

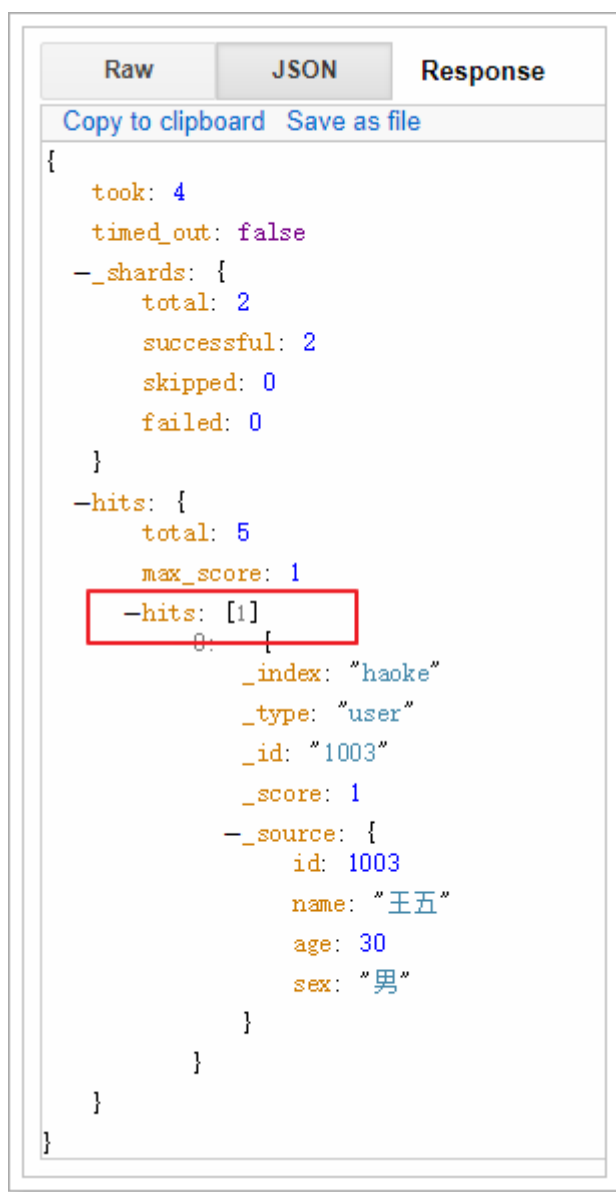
如果你想每页显示5个结果，页码从1到3，那请求如下：



```
1 GET /_search?size=5
2 GET /_search?size=5&from=5
3 GET /_search?size=5&from=10
```

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

```
1 GET /haoke/user/_search?size=1&from=2
```



## 在集群系统中深度分页

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

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

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

## 3.6、映射

前面我们创建的索引以及插入数据，都是由Elasticsearch进行自动判断类型，有些时候我们是需要进行明确字段类型的，否则，自动判断的类型和实际需求是不相符的。

自动判断的规则如下：

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

Elasticsearch中支持的类型如下：

类型	表示的数据类型
String	<code>string</code> , <code>text</code> , <code>keyword</code>
Whole number	<code>byte</code> , <code>short</code> , <code>integer</code> , <code>long</code>
Floating point	<code>float</code> , <code>double</code>
Boolean	<code>boolean</code>
Date	<code>date</code>

- `string`类型在ElasticSearch 旧版本中使用较多，从ElasticSearch 5.x开始不再支持`string`，由`text`和`keyword`类型替代。
- `text` 类型，当一个字段是要被全文搜索的，比如Email内容、产品描述，应该使用`text`类型。设置`text`类型以后，字段内容会被分析，在生成倒排索引以前，字符串会被分析器分成一个一个词项。`text`类型的字段不用于排序，很少用于聚合。
- `keyword`类型适用于索引结构化的字段，比如email地址、主机名、状态码和标签。如果字段需要进行过滤(比如查找已发布博客中`status`属性为`published`的文章)、排序、聚合。`keyword`类型的字段只能通过精确值搜索到。

创建明确类型的索引：

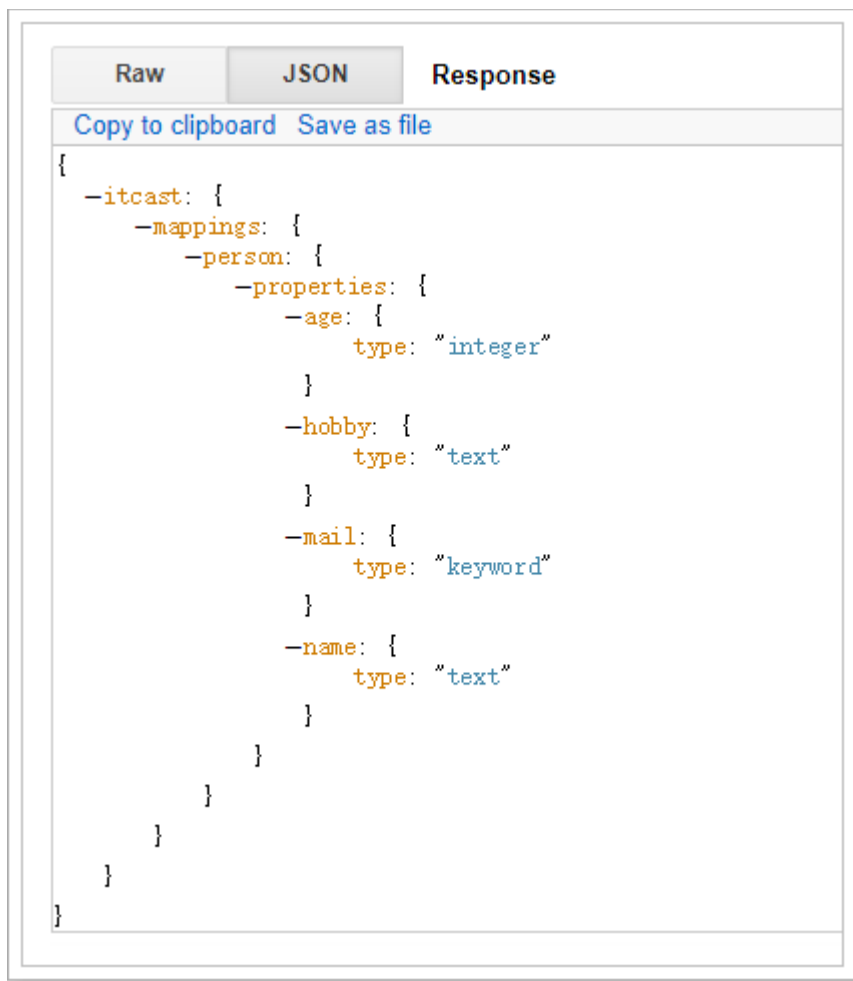
```
1 PUT /itcast
2 {
3   "settings": {
4     "index": {
5       "number_of_shards": "2",
```



```
6         "number_of_replicas": "0"
7     },
8 },
9 "mappings": {
10     "person": {
11         "properties": {
12             "name": {
13                 "type": "text"
14             },
15             "age": {
16                 "type": "integer"
17             },
18             "mail": {
19                 "type": "keyword"
20             },
21             "hobby": {
22                 "type": "text"
23             }
24         }
25     }
26 }
27 }
```

查看映射：

```
1 | GET /itcast/_mapping
```



插入数据：

```

1 POST /itcast/_bulk
2
3 {"index":{"_index":"itcast","_type":"person"}}
4 {"name":"张三","age": 20,"mail": "111@qq.com","hobby":"羽毛球、乒乓球、足球"}
5 {"index":{"_index":"itcast","_type":"person"}}
6 {"name":"李四","age": 21,"mail": "222@qq.com","hobby":"羽毛球、乒乓球、足球、篮球"}
7 {"index":{"_index":"itcast","_type":"person"}}
8 {"name":"王五","age": 22,"mail": "333@qq.com","hobby":"羽毛球、篮球、游泳、听音乐"}
9 {"index":{"_index":"itcast","_type":"person"}}
10 {"name":"赵六","age": 23,"mail": "444@qq.com","hobby":"跑步、游泳"}
11 {"index":{"_index":"itcast","_type":"person"}}
12 {"name":"孙七","age": 24,"mail": "555@qq.com","hobby":"听音乐、看电影"}
13
  
```

查询 2 个分片中用的 2 个, 5 命中, 耗时 0.002 秒

_index	_type	_id	_score ▼	name	age	mail	hobby
itcast	person	C7OM_2cB9cFOnF3ux7uD	1	孙七	24	555@qq.com	听音乐、看电影
itcast	person	B7OM_2cB9cFOnF3ux7uD	1	张三	20	111@qq.com	羽毛球、乒乓球、足球
itcast	person	CLOM_2cB9cFOnF3ux7uD	1	李四	21	222@qq.com	羽毛球、乒乓球、足球、篮球
itcast	person	CbOM_2cB9cFOnF3ux7uD	1	王五	22	333@qq.com	羽毛球、篮球、游泳、听音乐
itcast	person	CrOM_2cB9cFOnF3ux7uD	1	赵六	23	444@qq.com	跑步、游泳

测试搜索：

```
1 POST /itcast/person/_search
2 {
3     "query" : {
4         "match" : {
5             "hobby" : "音乐"
6         }
7     }
8 }
```



```
{
  took: 9
  timed_out: false
  _shards: {
    total: 2
    successful: 2
    skipped: 0
    failed: 0
  }
  -hits: {
    total: 2
    max_score: 2.1845279
    -hits: [2]
      -0: {
        _index: "itcast"
        _type: "person"
        _id: "Cb0M_2cB9cF0nF3ux7uD"
        _score: 2.1845279
        -_source: {
          name: "王五"
          age: 22
          mail: "333@qq.com"
          hobby: "羽毛球、篮球、游泳、听音乐"
        }
      }
      -1: {
        _index: "itcast"
        _type: "person"
        _id: "C70M_2cB9cF0nF3ux7uD"
        _score: 0.5753642
        -_source: {
          name: "孙七"
          age: 24
          mail: "555@qq.com"
          hobby: "听音乐、看电影"
        }
      }
    ]
  }
}
```

## 3.7、结构化查询

### 3.7.1、term查询

`term` 主要用于精确匹配哪些值，比如数字，日期，布尔值或 `not_analyzed` 的字符串(未经分析的文本数据类型)：



```
1 { "term": { "age": 26 } }
2 { "term": { "date": "2014-09-01" } }
3 { "term": { "public": true } }
4 { "term": { "tag": "full_text" } }
```

示例：

```
1 POST /itcast/person/_search
2 {
3   "query" : {
4     "term" : {
5       "age" : 20
6     }
7   }
8 }
```

```
-hits: {
  total: 1
  max_score: 1
  -hits: [1]
    -0: {
      _index: "itcast"
      _type: "person"
      _id: "B70M_2cB9cF0nF3ux7uD"
      _score: 1
      -_source: {
        name: "张三"
        age: 20
        mail: "111@qq.com"
        hobby: "羽毛球、乒乓球、足球"
      }
    }
  }
}
```

### 3.7.2、terms查询

`terms` 跟 `term` 有点类似，但 `terms` 允许指定多个匹配条件。如果某个字段指定了多个值，那么文档需要一起去匹配：

```
1 {
2   "terms": {
3     "tag": [ "search", "full_text", "nosql" ]
4   }
5 }
```

示例：



```
1 POST /itcast/person/_search
2 {
3     "query" : {
4         "terms" : {
5             "age" : [20,21]
6         }
7     }
8 }
```

```
-hits: {
  total: 2
  max_score: 1
  -hits: [2]
    -0: {
      _index: "itcast"
      _type: "person"
      _id: "B70M_2cB9cF0nF3ux7uD"
      _score: 1
      -_source: {
        name: "张三"
        age: 20
        mail: "111@qq.com"
        hobby: "羽毛球、乒乓球、足球"
      }
    }
    -1: {
      _index: "itcast"
      _type: "person"
      _id: "CLOM_2cB9cF0nF3ux7uD"
      _score: 1
      -_source: {
        name: "李四"
        age: 21
        mail: "222@qq.com"
        hobby: "羽毛球、乒乓球、足球、篮球"
      }
    }
  }
}
```

### 3.7.3、range查询

range 过滤允许我们按照指定范围查找一批数据：





```
1 {  
2   "range": {  
3     "age": {  
4       "gte": 20,  
5       "lte": 30  
6     }  
7   }  
8 }
```

范围操作符包含：

gt :: 大于

gte :: 大于等于

lt :: 小于

lte :: 小于等于

示例：

```
1 POST /itcast/person/_search  
2 {  
3   "query": {  
4     "range": {  
5       "age": {  
6         "gte": 20,  
7         "lte": 22  
8       }  
9     }  
10  }  
11 }
```



```
-hits: {
  total: 3
  max_score: 1
-hits: [3]
  -0: {
    _index: "itcast"
    _type: "person"
    _id: "B70M_2cB9cF0nF3ux7uD"
    _score: 1
    _source: {
      name: "张三"
      age: 20
      mail: "111@qq.com"
      hobby: "羽毛球、乒乓球、足球"
    }
  }
  -1: {
    _index: "itcast"
    _type: "person"
    _id: "CLOM_2cB9cF0nF3ux7uD"
    _score: 1
    _source: {
      name: "李四"
      age: 21
      mail: "222@qq.com"
      hobby: "羽毛球、乒乓球、足球、篮球"
    }
  }
  -2: {
    _index: "itcast"
    _type: "person"
    _id: "CbOM_2cB9cF0nF3ux7uD"
    _score: 1
    _source: {
      name: "王五"
      age: 22
      mail: "333@qq.com"
      hobby: "羽毛球、篮球、游泳、听音乐"
    }
  }
}
```

### 3.7.4、exists 查询

`exists` 查询可以用于查找文档中是否包含指定字段或没有某个字段，类似于SQL语句中的 `IS_NULL` 条件

```
1 {
2   "exists": {
3     "field": "title"
4   }
5 }
```

这两个查询只是针对已经查出一批数据来，但是想区分出某个字段是否存在的时候使用。

示例：

```
1 POST /haoke/user/_search
2 {
3   "query": {
4     "exists": { #必须包含
5       "field": "card"
6     }
7   }
8 }
9
```

```
-hits: {
  total: 1
  max_score: 1
  -hits: [1]
    -0: {
      _index: "haoke"
      _type: "user"
      _id: "1005"
      _score: 1
      -_source: {
        id: 1005
        name: "孙七"
        age: 37
        sex: "女"
        -card: {
          card_number: "123456789"
        }
      }
    }
  }
}
```

### 3.6.5、match查询

`match` 查询是一个标准查询，不管你需要全文本查询还是精确查询基本上都要用到它。

如果你使用 `match` 查询一个全文本字段，它会在真正查询之前用分析器先分析 `match` 一下查询字符：

```
1 {
2   "match": {
3     "tweet": "About Search"
4   }
5 }
```

如果用 `match` 下指定了一个确切值，在遇到数字，日期，布尔值或者 `not_analyzed` 的字符串时，它将为你搜索你给定的值：



```
1 { "match": { "age": 26 }}
2 { "match": { "date": "2014-09-01" }}
3 { "match": { "public": true }}
4 { "match": { "tag": "full_text" }}
```

### 3.7.6、bool查询

bool 查询可以用来合并多个条件查询结果的布尔逻辑，它包含一下操作符：

must :: 多个查询条件的完全匹配,相当于 and。

must\_not :: 多个查询条件的相反匹配，相当于 not。

should :: 至少有一个查询条件匹配, 相当于 or。

这些参数可以分别继承一个查询条件或者一个查询条件的数组：

```
1 {
2   "bool": {
3     "must": { "term": { "folder": "inbox" }},
4     "must_not": { "term": { "tag": "spam" }},
5     "should": [
6       { "term": { "starred": true }},
7       { "term": { "unread": true }}
8     ]
9   }
10 }
```

## 3.8、过滤查询

前面讲过结构化查询，Elasticsearch也支持过滤查询，如term、range、match等。

示例：查询年龄为20岁的用户。

```
1 POST /itcast/person/_search
2 {
3   "query": {
4     "bool": {
5       "filter": {
6         "term": {
7           "age": 20
8         }
9       }
10    }
11  }
12 }
```

结果：

```
Raw JSON Response
Copy to clipboard Save as file
{
  took: 10
  timed_out: false
  _shards: {
    total: 2
    successful: 2
    skipped: 0
    failed: 0
  }
  -hits: {
    total: 1
    max_score: 0
    -hits: [1]
      -0: {
        _index: "itcast"
        _type: "person"
        _id: "B70M_2cB9cF0nF3ux7uD"
        _score: 0
        -_source: {
          name: "张三"
          age: 20
          mail: "111@qq.com"
          hobby: "羽毛球、乒乓球、足球"
        }
      }
    ]
  }
}
```

#### 查询和过滤的对比

- 一条过滤语句会询问每个文档的字段值是否包含着特定值。
- 查询语句会询问每个文档的字段值与特定值的匹配程度如何。
  - 一条查询语句会计算每个文档与查询语句的相关性，会给出一个相关性评分 `_score`，并且按照相关性对匹配到的文档进行排序。这种评分方式非常适用于一个没有完全配置结果的全文本搜索。
- 一个简单的文档列表，快速匹配运算并存入内存是十分方便的，每个文档仅需要1个字节。这些缓存的过滤结果集与后续请求的结合使用是非常高效的。
- 查询语句不仅要查找相匹配的文档，还需要计算每个文档的相关性，所以一般来说查询语句要比过滤语句更耗时，并且查询结果也不可缓存。

建议：

做精确匹配搜索时，最好用过滤语句，因为过滤语句可以缓存数据。

## 4、中文分词

### 4.1、什么是分词

分词就是指将一个文本转化成一系列单词的过程，也叫文本分析，在Elasticsearch中称之为Analysis。

举例：我是中国人 --> 我/是/中国人

## 4.2、分词api

指定分词器进行分词

```
1 POST /_analyze
2 {
3     "analyzer":"standard",
4     "text":"hello world"
5 }
```

结果：



在结果中不仅可以看出分词的结果，还返回了该词在文本中的位置。

指定索引分词

```
1 POST /itcast/_analyze
2 {
3     "analyzer": "standard",
4     "field": "hobby",
5     "text": "听音乐"
6 }
```

```
Raw JSON Response
Copy to clipboard Save as file
{
  -tokens: [3]
    -0: {
      token: "听"
      start_offset: 0
      end_offset: 1
      type: "<IDEOGRAPHIC>"
      position: 0
    }
    -1: {
      token: "音"
      start_offset: 1
      end_offset: 2
      type: "<IDEOGRAPHIC>"
      position: 1
    }
    -2: {
      token: "乐"
      start_offset: 2
      end_offset: 3
      type: "<IDEOGRAPHIC>"
      position: 2
    }
  }
}
```

## 4.4、中文分词

中文分词的难点在于，在汉语中没有明显的词汇分界点，如在英语中，空格可以作为分隔符，如果分隔不正确就会造成歧义。

如：

我/爱/炒肉丝

我/爱/炒/肉丝

常用中文分词器，IK、jieba、THULAC等，推荐使用IK分词器。

IK Analyzer是一个开源的，基于java语言开发的轻量级的中文分词工具包。从2006年12月推出1.0版开始，IKAnalyzer已经推出了3个大版本。最初，它是以开源项目Lunatic为应用主体的，结合词典分词和文法分析算法的中文分词组件。新版本的IK Analyzer 3.0则发展为面向Java的公用分词组件，独立于Lucene项目，同时提供了对Lucene的默认优化实现。

采用了特有的“正向迭代最细粒度切分算法”，具有80万字/秒的高速处理能力 采用了多子处理器分析模式，支持：英文字母（IP地址、Email、URL）、数字（日期，常用中文数量词，罗马数字，科学计数法），中文词汇（姓名、地名处理）等分词处理。优化的词典存储，更小的内存占用。

IK分词器 Elasticsearch插件地址：<https://github.com/medcl/elasticsearch-analysis-ik>



```
1 #安装方法：将下载到的elasticsearch-analysis-ik-6.5.4.zip解压到/elasticsearch/plugins/ik目录下  
  即可。  
2 mkdir es/plugins/ik  
3 cp elasticsearch-analysis-ik-6.5.4.zip ./es/plugins/ik  
4  
5 #解压  
6 unzip elasticsearch-analysis-ik-6.5.4.zip  
7  
8 #重启  
9 ./bin/elasticsearch
```

测试：

```
1 POST /_analyze  
2 {  
3     "analyzer": "ik_max_word",  
4     "text": "我是中国人"  
5 }
```

结果：

```
1 {  
2     "tokens": [  
3         {  
4             "token": "我",  
5             "start_offset": 0,  
6             "end_offset": 1,  
7             "type": "CN_CHAR",  
8             "position": 0  
9         },  
10        {  
11            "token": "是",  
12            "start_offset": 1,  
13            "end_offset": 2,  
14            "type": "CN_CHAR",  
15            "position": 1  
16        },  
17        {  
18            "token": "中国人",  
19            "start_offset": 2,  
20            "end_offset": 5,  
21            "type": "CN_WORD",  
22            "position": 2  
23        },  
24        {  
25            "token": "中国",  
26            "start_offset": 2,  
27            "end_offset": 4,  
28            "type": "CN_WORD",  
29            "position": 3  
30        },  
31    ]  
}
```





```
32         "token": "国人",
33         "start_offset": 3,
34         "end_offset": 5,
35         "type": "CN_WORD",
36         "position": 4
37     }
38 ]
39 }
```

可以看到，已经对中文进行了分词。

## 5、全文搜索

全文搜索两个最重要的方面是：

- 相关性（Relevance）它是评价查询与其结果间的相关程度，并根据这种相关程度对结果排名的一种能力，这种计算方式可以是 TF/IDF 方法、地理位置邻近、模糊相似，或其他的某些算法。
- 分词（Analysis）它是将文本块转换为有区别的、规范化的 token 的一个过程，目的是为了创建倒排索引以及查询倒排索引。

### 5.1、构造数据

```
1  PUT /itcast
2  {
3      "settings": {
4          "index": {
5              "number_of_shards": "1",
6              "number_of_replicas": "0"
7          }
8      },
9      "mappings": {
10         "person": {
11             "properties": {
12                 "name": {
13                     "type": "text"
14                 },
15                 "age": {
16                     "type": "integer"
17                 },
18                 "mail": {
19                     "type": "keyword"
20                 },
21                 "hobby": {
22                     "type": "text",
23                     "analyzer": "ik_max_word"
24                 }
25             }
26         }
27     }
28 }
```

批量插入数据：



```
1 POST http://172.16.55.185:9200/itcast/_bulk
2
3 {"index":{"_index":"itcast","_type":"person"}}
4 {"name":"张三","age": 20,"mail": "111@qq.com","hobby":"羽毛球、乒乓球、足球"}
5 {"index":{"_index":"itcast","_type":"person"}}
6 {"name":"李四","age": 21,"mail": "222@qq.com","hobby":"羽毛球、乒乓球、足球、篮球"}
7 {"index":{"_index":"itcast","_type":"person"}}
8 {"name":"王五","age": 22,"mail": "333@qq.com","hobby":"羽毛球、篮球、游泳、听音乐"}
9 {"index":{"_index":"itcast","_type":"person"}}
10 {"name":"赵六","age": 23,"mail": "444@qq.com","hobby":"跑步、游泳、篮球"}
11 {"index":{"_index":"itcast","_type":"person"}}
12 {"name":"孙七","age": 24,"mail": "555@qq.com","hobby":"听音乐、看电影、羽毛球"}
13
```

结果：

查询 1 个分片中用的 1 个, 4 命中, 耗时 0.001 秒

_index	_type	_id	_score ▲	name	age	mail	hobby
itcast	person	UP0cDWgBR-bSw8-LpdkZ	1	张三	20	111@qq.com	羽毛球、乒乓球、足球
itcast	person	Uf0cDWgBR-bSw8-LpdkZ	1	李四	21	222@qq.com	羽毛球、乒乓球、足球、篮球
itcast	person	Uv0cDWgBR-bSw8-LpdkZ	1	王五	22	333@qq.com	羽毛球、篮球、游泳、听音乐
itcast	person	VP0cDWgBR-bSw8-LpdkZ	1	孙七	24	555@qq.com	听音乐、看电影、羽毛球

## 5.2、单词搜索

```
1 POST /itcast/person/_search
2
3 {
4   "query":{
5     "match":{
6       "hobby":"音乐"
7     }
8   },
9   "highlight": {
10    "fields": {
11      "hobby": {}
12    }
13  }
14 }
```

结果：

```
1 {
2   "took": 9,
3   "timed_out": false,
4   "_shards": {
5     "total": 1,
6     "successful": 1,
7     "skipped": 0,
8     "failed": 0
9   },
10  "hits": {
```



```
11     "total": 2,
12     "max_score": 0.6841192,
13     "hits": [
14         {
15             "_index": "itcast",
16             "_type": "person",
17             "_id": "Uv0cDWgBR-bsw8-Lpdkz",
18             "_score": 0.6841192,
19             "_source": {
20                 "name": "王五",
21                 "age": 22,
22                 "mail": "333@qq.com",
23                 "hobby": "羽毛球、篮球、游泳、听音乐"
24             },
25             "highlight": {
26                 "hobby": [
27                     "羽毛球、篮球、游泳、听<em>音乐</em>"
28                 ]
29             }
30         },
31         {
32             "_index": "itcast",
33             "_type": "person",
34             "_id": "VP0cDWgBR-bsw8-Lpdkz",
35             "_score": 0.6841192,
36             "_source": {
37                 "name": "孙七",
38                 "age": 24,
39                 "mail": "555@qq.com",
40                 "hobby": "听音乐、看电影、羽毛球"
41             },
42             "highlight": {
43                 "hobby": [
44                     "听<em>音乐</em>、看电影、羽毛球"
45                 ]
46             }
47         }
48     ]
49 }
50 }
```

#### 过程说明：

##### 1. 检查字段类型

爱好 hobby 字段是一个 text 类型（指定了IK分词器），这意味着查询字符串本身也应该被分词。

##### 2. 分析查询字符串。

将查询的字符串“音乐”传入IK分词器中，输出的结果是单个项 音乐。因为只有一个单词项，所以 match 查询执行的是单个底层 term 查询。

##### 3. 查找匹配文档。

用 term 查询在倒排索引中查找“音乐”然后获取一组包含该项的文档，本例的结果是文档：3、5。

#### 4. 为每个文档评分。

用 term 查询计算每个文档相关度评分 `_score`，这是种将 词频（term frequency，即词“音乐”在相关文档的 hobby 字段中出现的频率）和 反向文档频率（inverse document frequency，即词“音乐”在所有文档的 hobby 字段中出现的频率），以及字段的长度（即字段越短相关度越高）相结合的计算方式。

### 5.3、多词搜索

```
1 POST /itcast/person/_search
2 {
3     "query":{
4         "match":{
5             "hobby":"音乐 篮球"
6         }
7     },
8     "highlight": {
9         "fields": {
10             "hobby": {}
11         }
12     }
13 }
```

结果：

```
1 {
2     "took": 3,
3     "timed_out": false,
4     "_shards": {
5         "total": 1,
6         "successful": 1,
7         "skipped": 0,
8         "failed": 0
9     },
10    "hits": {
11        "total": 4,
12        "max_score": 1.3192271,
13        "hits": [
14            {
15                "_index": "itcast",
16                "_type": "person",
17                "_id": "Uv0cDWgBR-bSw8-LpdKz",
18                "_score": 1.3192271,
19                "_source": {
20                    "name": "王五",
21                    "age": 22,
22                    "mail": "333@qq.com",
23                    "hobby": "羽毛球、篮球、游泳、听音乐"
24                },
25                "highlight": {
26                    "hobby": [
27                        "羽毛球、<em>篮球</em>、游泳、听<em>音乐</em>"
28                    ]
29                }
30            }
31        ]
32    }
33 }
```

```
29     }
30     },
31     {
32         "_index": "itcast",
33         "_type": "person",
34         "_id": "VP0cDWgBR-bSw8-Lpdkz",
35         "_score": 0.81652206,
36         "_source": {
37             "name": "孙七",
38             "age": 24,
39             "mail": "555@qq.com",
40             "hobby": "听音乐、看电影、羽毛球"
41         },
42         "highlight": {
43             "hobby": [
44                 "听<em>音乐</em>、看电影、羽毛球"
45             ]
46         }
47     },
48     {
49         "_index": "itcast",
50         "_type": "person",
51         "_id": "Vf0gDWgBR-bSw8-LOdm_",
52         "_score": 0.6987338,
53         "_source": {
54             "name": "赵六",
55             "age": 23,
56             "mail": "444@qq.com",
57             "hobby": "跑步、游泳、篮球"
58         },
59         "highlight": {
60             "hobby": [
61                 "跑步、游泳、<em>篮球</em>"
62             ]
63         }
64     },
65     {
66         "_index": "itcast",
67         "_type": "person",
68         "_id": "Uf0cDWgBR-bSw8-Lpdkz",
69         "_score": 0.50270504,
70         "_source": {
71             "name": "李四",
72             "age": 21,
73             "mail": "222@qq.com",
74             "hobby": "羽毛球、乒乓球、足球、篮球"
75         },
76         "highlight": {
77             "hobby": [
78                 "羽毛球、乒乓球、足球、<em>篮球</em>"
79             ]
80         }
81     }
82 }
```

```
82     ]
83     }
84 }
```

可以看到，包含了“音乐”、“篮球”的数据都已经被搜索到了。

可是，搜索的结果并不符合我们的预期，因为我们想搜索的是既包含“音乐”又包含“篮球”的用户，显然结果返回的“或”的关系。

在Elasticsearch中，可以指定词之间的逻辑关系，如下：

```
1 POST /itcast/person/_search
2 {
3     "query":{
4         "match":{
5             "hobby":{
6                 "query":"音乐 篮球",
7                 "operator":"and"
8             }
9         }
10    },
11    "highlight": {
12        "fields": {
13            "hobby": {}
14        }
15    }
16 }
```

结果：

```
{
  "took": 3,
  "timed_out": false,
  "_shards": {
    "total": 1,
    "successful": 1,
    "skipped": 0,
    "failed": 0
  },
  "hits": {
    "total": 1,
    "max_score": 1.3192271,
    "hits": [
      {
        "_index": "itcast",
        "_type": "person",
        "_id": "Uv0cDWgBR-bSw8-LpdkZ",
        "_score": 1.3192271,
        "_source": {
          "name": "王五",
          "age": 22,
          "mail": "333@qq.com",
          "hobby": "羽毛球、篮球、游泳、听音乐"
        },
        "highlight": {
          "hobby": [
            "羽毛球、<em>篮球</em>、游泳、听<em>音乐</em>"
          ]
        }
      }
    ]
  }
}
```

可以看到结果符合预期。

前面我们测试了“OR”和“AND”搜索，这是两个极端，其实在实际场景中，并不会选取这2个极端，更有可能是选取这种，或者说，只需要符合一定的相似度就可以查询到数据，在Elasticsearch中也支持这样的查询，通过minimum\_should\_match来指定匹配度，如：70%；

示例：

```
1  {
2      "query": {
3          "match": {
4              "hobby": {
5                  "query": "游泳 羽毛球",
6                  "minimum_should_match": "80%"
7              }
8          }
9      },
10     "highlight": {
11         "fields": {
12             "hobby": {}
13         }
14     }
15 }
16
17 #结果：省略显示
18 "hits": {
19     "total": 4, #相似度为80%的情况下，查询到4条数据
20     "max_score": 1.621458,
21     "hits": [
22         .....
23     ]
24 }
25
26 #设置40%进行测试：
27 {
28     "query": {
29         "match": {
30             "hobby": {
31                 "query": "游泳 羽毛球",
32                 "minimum_should_match": "40%"
33             }
34         }
35     },
36     "highlight": {
37         "fields": {
38             "hobby": {}
39         }
40     }
41 }
42 #结果：
43 "hits": {
44     "total": 5, #相似度为40%的情况下，查询到5条数据
45     "max_score": 1.621458,
46     "hits": [
47         .....
48     ]
49 }
```



相似度应该多少合适，需要在实际的需求中进行反复测试，才可得到合理的值。

## 5.4、组合搜索

在搜索时，也可以使用过滤器中讲过的bool组合查询，示例：

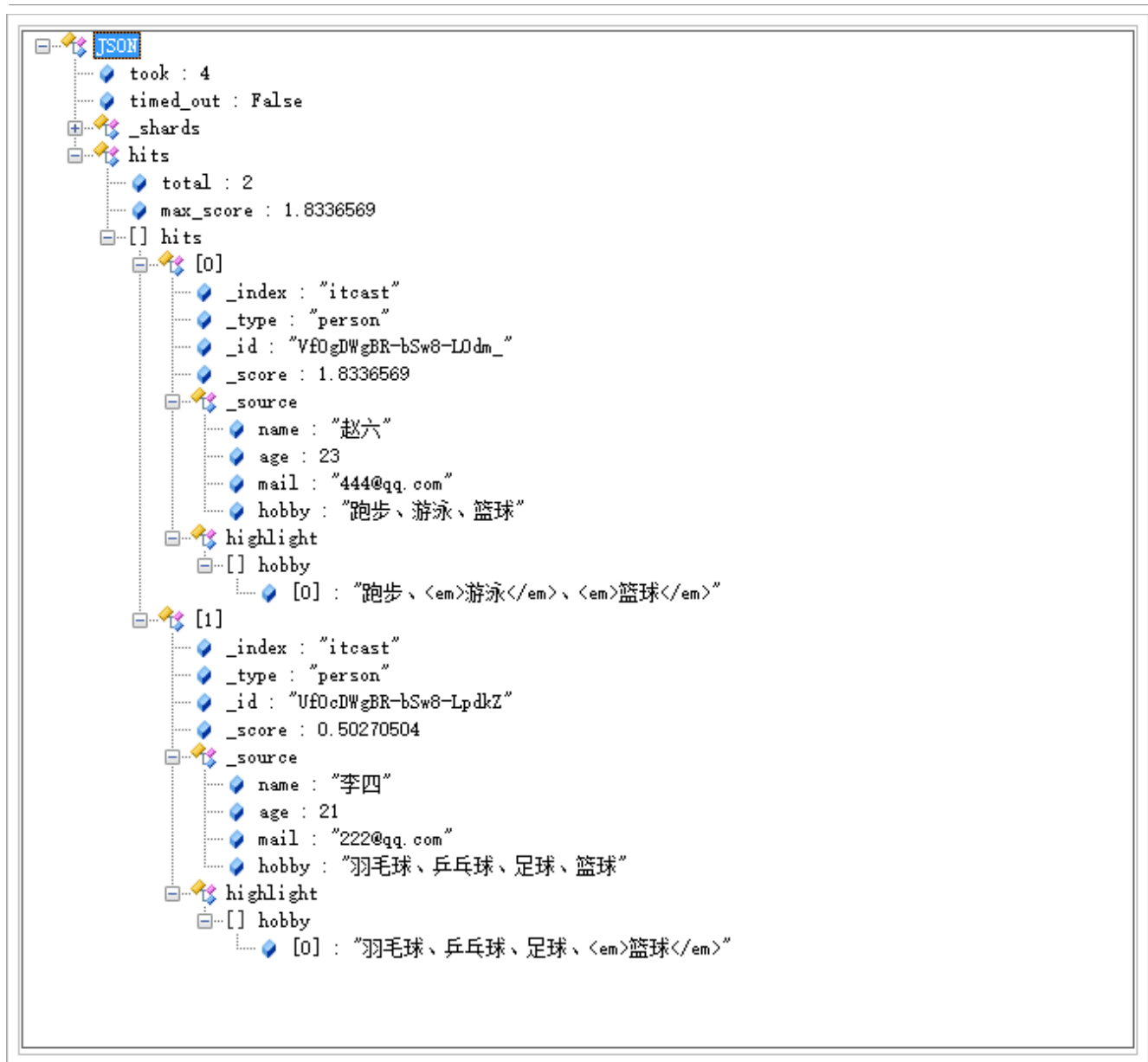
```
1 POST /itcast/person/_search
2
3 {
4   "query":{
5     "bool":{
6       "must":{
7         "match":{
8           "hobby":"篮球"
9         }
10      },
11      "must_not":{
12        "match":{
13          "hobby":"音乐"
14        }
15      },
16      "should":[
17        {
18          "match": {
19            "hobby":"游泳"
20          }
21        }
22      ]
23    }
24  },
25  "highlight": {
26    "fields": {
27      "hobby": {}
28    }
29  }
30 }
```

上面搜索的意思是：

搜索结果中必须包含篮球，不能包含音乐，如果包含了游泳，那么它的相似度更高。

结果：





### 评分的计算规则

bool 查询会为每个文档计算相关度评分 `_score`，再将所有匹配的 `must` 和 `should` 语句的分数 `_score` 求和，最后除以 `must` 和 `should` 语句的总数。

`must_not` 语句不会影响评分；它的作用只是将不相关的文档排除。

默认情况下，`should` 中的内容不是必须匹配的，如果查询语句中没有 `must`，那么就会至少匹配其中一个。当然了，也可以通过 `minimum_should_match` 参数进行控制，该值可以是数字也可以的百分比。

示例：

```
1 POST /itcast/person/_search
2
3 {
4   "query":{
5     "bool":{
6       "should":[
```



```
7      {
8          "match": {
9              "hobby": "游泳"
10         }
11     },
12     {
13         "match": {
14             "hobby": "篮球"
15         }
16     },
17     {
18         "match": {
19             "hobby": "音乐"
20         }
21     }
22 ],
23     "minimum_should_match": 2
24 }
25 },
26 "highlight": {
27     "fields": {
28         "hobby": {}
29     }
30 }
31 }
```

minimum\_should\_match为2，意思是should中的三个词，至少要满足2个。

结果：

```
JSON
{
  took : 5
  timed_out : False
  _shards : 
  hits :
    total : 2
    max_score : 2.135749
    hits :
      [0]
        _index : "itcast"
        _type : "person"
        _id : "Uv0cDWgBR-bSw8-LpdkZ"
        _score : 2.135749
        _source :
          name : "王五"
          age : 22
          mail : "333@qq.com"
          hobby : "羽毛球、篮球、游泳、听音乐"
        highlight :
          hobby :
            [0] : "羽毛球、<em>篮球</em>、<em>游泳</em>、听<em>音乐</em>"
      [1]
        _index : "itcast"
        _type : "person"
        _id : "Vf0gDWgBR-bSw8-L0dm_"
        _score : 1.8336569
        _source :
          name : "赵六"
          age : 23
          mail : "444@qq.com"
          hobby : "跑步、游泳、篮球"
        highlight :
          hobby :
            [0] : "跑步、<em>游泳</em>、<em>篮球</em>"

```

## 5.5、权重

有些时候，我们可能需要对某些词增加权重来影响该条数据的得分。如下：

搜索关键字为“游泳篮球”，如果结果中包含了“音乐”权重为10，包含了“跑步”权重为2。

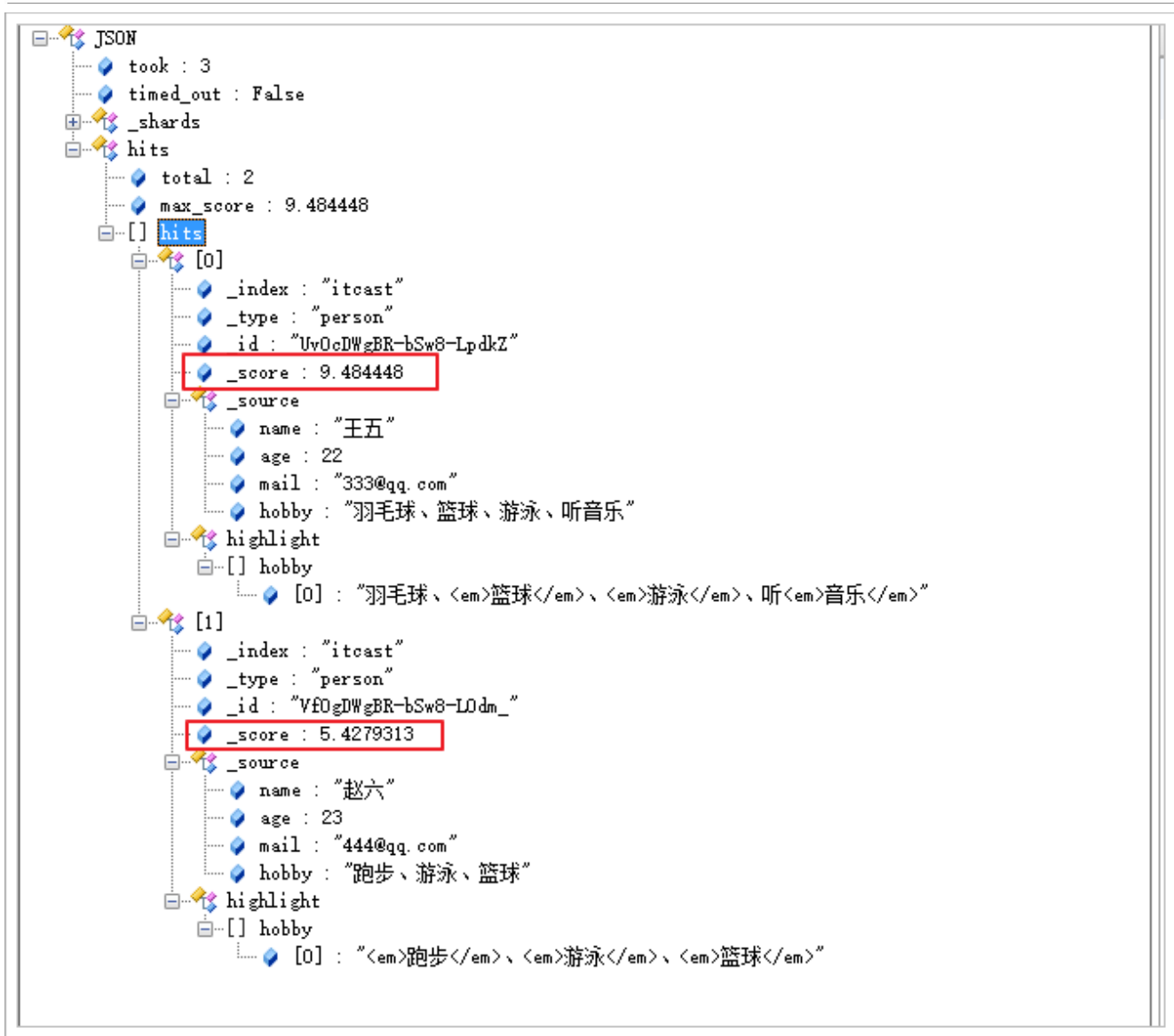
```
1 POST /itcast/person/_search
2 {
3   "query": {
4     "bool": {
5       "must": {
6         "match": {
7           "hobby": {
8             "query": "游泳篮球",
9             "operator": "and"
10          }
11        }
12      },

```



```
13     "should": [  
14         {  
15             "match": {  
16                 "hobby": {  
17                     "query": "音乐",  
18                     "boost": 10  
19                 }  
20             }  
21         },  
22         {  
23             "match": {  
24                 "hobby": {  
25                     "query": "跑步",  
26                     "boost": 2  
27                 }  
28             }  
29         }  
30     ]  
31 }  
32 },  
33 "highlight": {  
34     "fields": {  
35         "hobby": {}  
36     }  
37 }  
38 }
```

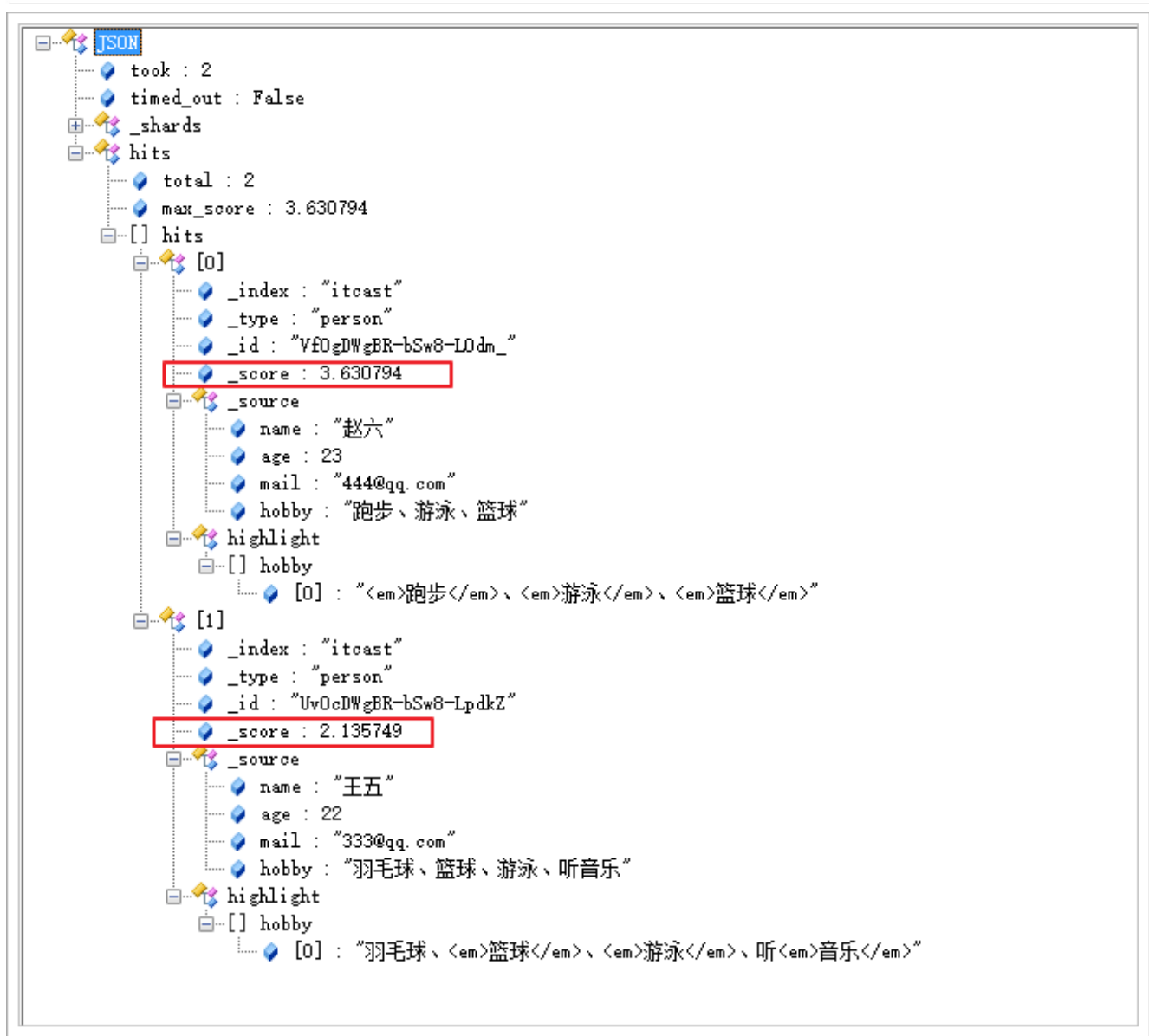
结果：



```
JSON
{
  took : 3
  timed_out : False
  _shards :
  hits :
    total : 2
    max_score : 9.484448
    hits :
      [0]
        _index : "itcast"
        _type : "person"
        id : "Uv0cDWgBR-bSw8-LpdkZ"
        _score : 9.484448
        _source :
          name : "王五"
          age : 22
          mail : "333@qq.com"
          hobby : "羽毛球、篮球、游泳、听音乐"
        highlight :
          hobby :
            [0] : "羽毛球、<em>篮球</em>、<em>游泳</em>、听<em>音乐</em>"
      [1]
        _index : "itcast"
        _type : "person"
        id : "Vf0gDWgBR-bSw8-L0dm_"
        _score : 5.4279313
        _source :
          name : "赵六"
          age : 23
          mail : "444@qq.com"
          hobby : "跑步、游泳、篮球"
        highlight :
          hobby :
            [0] : "<em>跑步</em>、<em>游泳</em>、<em>篮球</em>"

```

如果不设置权重的查询结果是这样：



## 6、Elasticsearch集群

### 6.1、集群节点

Elasticsearch的集群是由多个节点组成的，通过cluster.name设置集群名称，并且用于区分其它的集群，每个节点通过node.name指定节点的名称。

在Elasticsearch中，节点的类型主要有4种：

- master节点
  - 配置文件中node.master属性为true(默认为true)，就有资格被选为master节点。
  - master节点用于控制整个集群的操作。比如创建或删除索引，管理其它非master节点等。
- data节点
  - 配置文件中node.data属性为true(默认为true)，就有资格被设置成data节点。
  - data节点主要用于执行数据相关的操作。比如文档的CRUD。
- 客户端节点
  - 配置文件中node.master属性和node.data属性均为false。



- 该节点不能作为master节点，也不能作为data节点。
- 可以作为客户端节点，用于响应用户的请求，把请求转发到其他节点
- 部落节点
  - 当一个节点配置tribe.\*的时候，它是一个特殊的客户端，它可以连接多个集群，在所有连接的集群上执行搜索和其他操作。

## 6.2、搭建集群

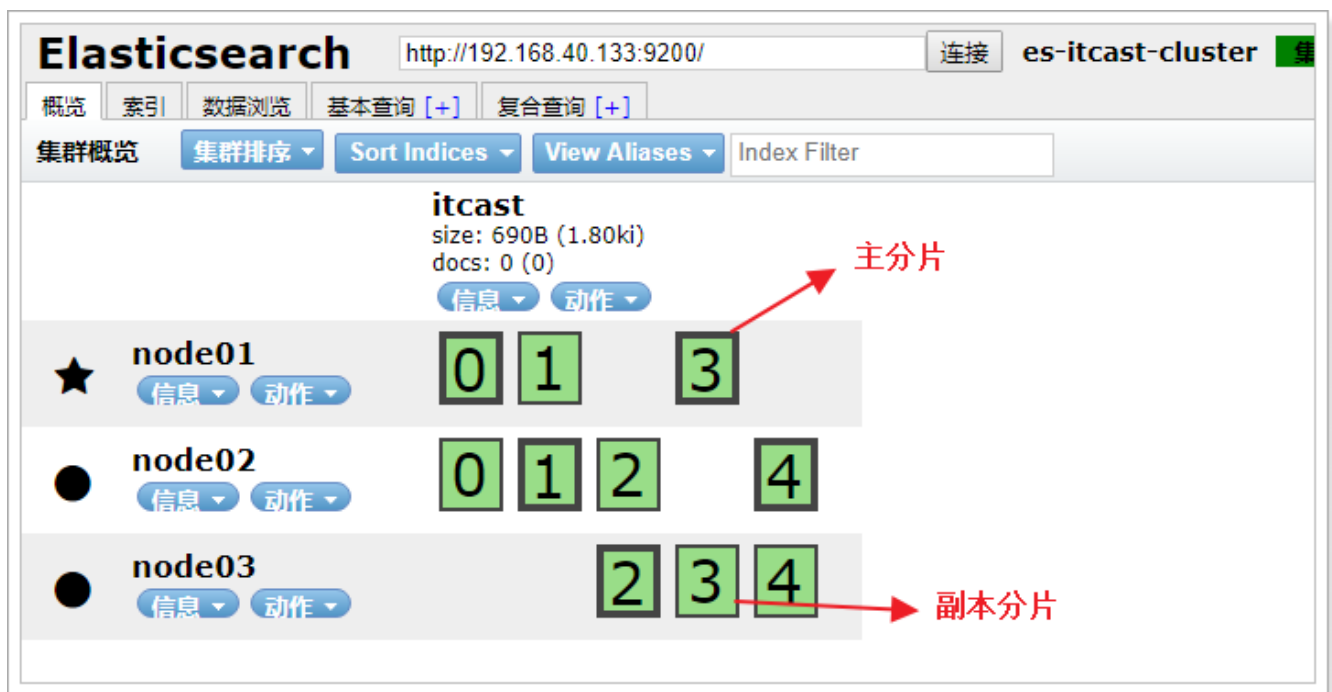
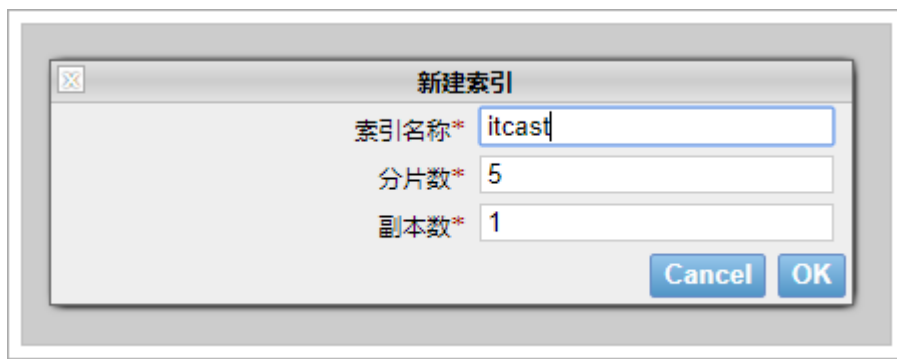
```
1  #启动3个虚拟机，分别在3台虚拟机上部署安装Elasticsearch
2  mkdir /itcast/es-cluster
3
4  #分发到其它机器
5  scp -r es-cluster elsearch@192.168.40.134:/itcast
6
7  #node01的配置：
8  cluster.name: es-itcast-cluster
9  node.name: node01
10 node.master: true
11 node.data: true
12 network.host: 0.0.0.0
13 http.port: 9200
14 discovery.zen.ping.unicast.hosts: ["192.168.40.133","192.168.40.134","192.168.40.135"]
15 discovery.zen.minimum_master_nodes: 2
16 http.cors.enabled: true
17 http.cors.allow-origin: "*"
18
19 #node02的配置：
20 cluster.name: es-itcast-cluster
21 node.name: node02
22 node.master: true
23 node.data: true
24 network.host: 0.0.0.0
25 http.port: 9200
26 discovery.zen.ping.unicast.hosts: ["192.168.40.133","192.168.40.134","192.168.40.135"]
27 discovery.zen.minimum_master_nodes: 2
28 http.cors.enabled: true
29 http.cors.allow-origin: "*"
30
31 #node03的配置：
32 cluster.name: es-itcast-cluster
33 node.name: node02
34 node.master: true
35 node.data: true
36 network.host: 0.0.0.0
37 http.port: 9200
38 discovery.zen.ping.unicast.hosts: ["192.168.40.133","192.168.40.134","192.168.40.135"]
39 discovery.zen.minimum_master_nodes: 2
40 http.cors.enabled: true
41 http.cors.allow-origin: "*"
42
43 #分别启动3个节点
44 ./elasticsearch
```



查看集群：



创建索引：



查询集群状态：`/_cluster/health`

响应：

```
1 {
```



```
2   cluster_name: "es-itcast-cluster"
3   status: "green"
4   timed_out: false
5   number_of_nodes: 3
6   number_of_data_nodes: 3
7   active_primary_shards: 5
8   active_shards: 10
9   relocating_shards: 0
10  initializing_shards: 0
11  unassigned_shards: 0
12  delayed_unassigned_shards: 0
13  number_of_pending_tasks: 0
14  number_of_in_flight_fetch: 0
15  task_max_waiting_in_queue_millis: 0
16  active_shards_percent_as_number: 100
17 }
```

集群状态的三种颜色：

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

## 6.3、分片和副本

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

- 一个分片(shard)是一个最小级别“工作单元(worker unit)”，它只是保存了索引中所有数据的一部分。
- 我们需要知道分片就是一个Lucene实例，并且它本身就是一个完整的搜索引擎。应用程序不会和它直接通信。
- 分片可以是主分片(primary shard)或者是复制分片(replica shard)。
- 索引中的每个文档属于一个单独的主分片，所以主分片的数量决定了索引最多能存储多少数据。
- 复制分片只是主分片的一个副本，它可以防止硬件故障导致的数据丢失，同时可以提供读请求，比如搜索或者从别的shard取回文档。
- 当索引创建完成的时候，主分片的数量就固定了，但是复制分片的数量可以随时调整。

## 6.4、故障转移

### 6.4.1、将data节点停止

这里选择将node02停止：



Elasticsearch cluster health status: yellow (6 of 10). The cluster is named 'es-itcast-cluster' and is connected to 'http://192.168.40.133:9200/'. The index 'itcast' has a size of 1.12ki (1.35ki) and 0 docs. The cluster is in a 'yellow' state, indicating that some shards are not available. The shards are distributed across three nodes: 'Unassigned' (0, 1, 2, 4), 'node01' (0, 1, 3), and 'node03' (2, 3, 4).

说明：

- 当前集群状态为黄色，表示主节点可用，副本节点不完全可用

过一段时间观察，发现节点列表中看不到node02，副本节点分配到了node01和node03，集群状态恢复到绿色。

Elasticsearch cluster health status: green (10 of 10). The cluster is named 'es-itcast-cluster' and is connected to 'http://192.168.40.133:9200/'. The index 'itcast' has a size of 1.12ki (2.25ki) and 0 docs. The cluster is in a 'green' state, indicating that all shards are available. The shards are distributed across two nodes: 'node01' (0, 1, 2, 3, 4) and 'node03' (0, 1, 2, 3, 4).

将node02恢复：

```
1 | ./node02/bin/elasticsearch
```

Elasticsearch cluster health status: green (10 of 10). The cluster is named 'es-itcast-cluster' and is connected to 'http://192.168.40.133:9200/'. The index 'itcast' has a size of 1.27ki (2.52ki) and 0 docs. The cluster is in a 'green' state, indicating that all shards are available. The shards are distributed across three nodes: 'node01' (0, 2, 3, 4), 'node02' (0, 1, 2), and 'node03' (1, 3, 4).

可以看到，node02恢复后，重新加入了集群，并且重新分配了节点信息。

## 6.4.2、将master节点停止

接下来，测试将node01停止，也就是将主节点停止。

Elasticsearch cluster health status: yellow (6 of 10). The cluster is named 'es-itcast-cluster'. The index 'itcast' has a size of 1.24ki (1.50ki) and 0 documents. The cluster is in a 'yellow' state. The nodes are: node02 (master, 0, 1, 2) and node03 (star, 1, 3, 4).

从结果中可以看出，集群对master进行了重新选举，选择node03为master。并且集群状态变成黄色。

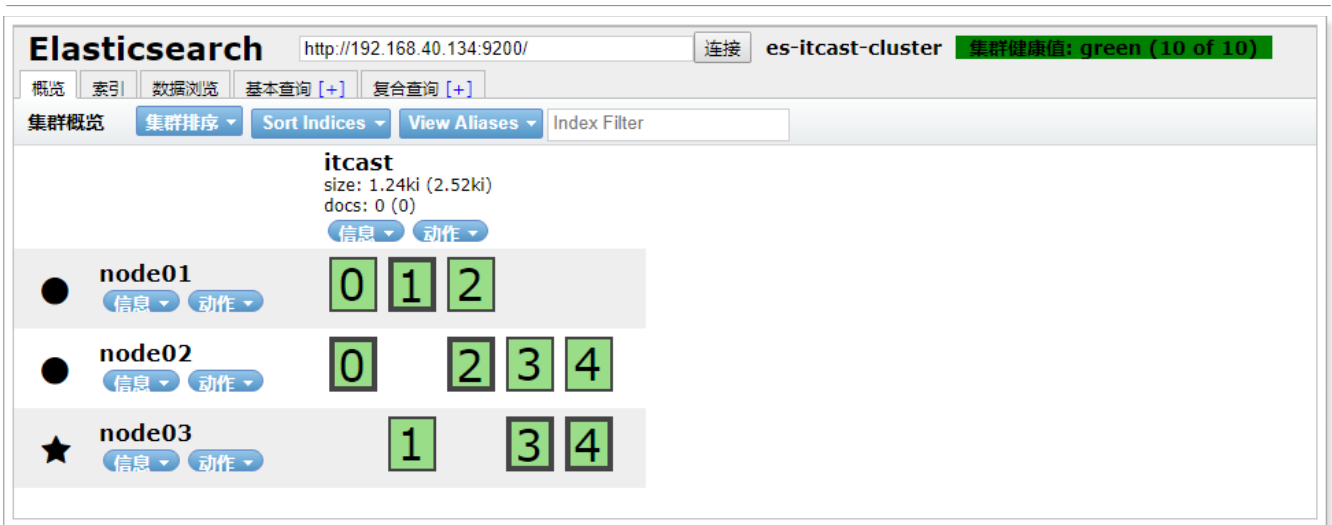
等待一段时间后，集群状态从黄色变为了绿色：

Elasticsearch cluster health status: green (10 of 10). The cluster is named 'es-itcast-cluster'. The index 'itcast' has a size of 1.24ki (2.49ki) and 0 documents. The cluster is in a 'green' state. The nodes are: node02 (master, 0, 1, 2, 3, 4) and node03 (star, 0, 1, 2, 3, 4).

恢复node01节点：

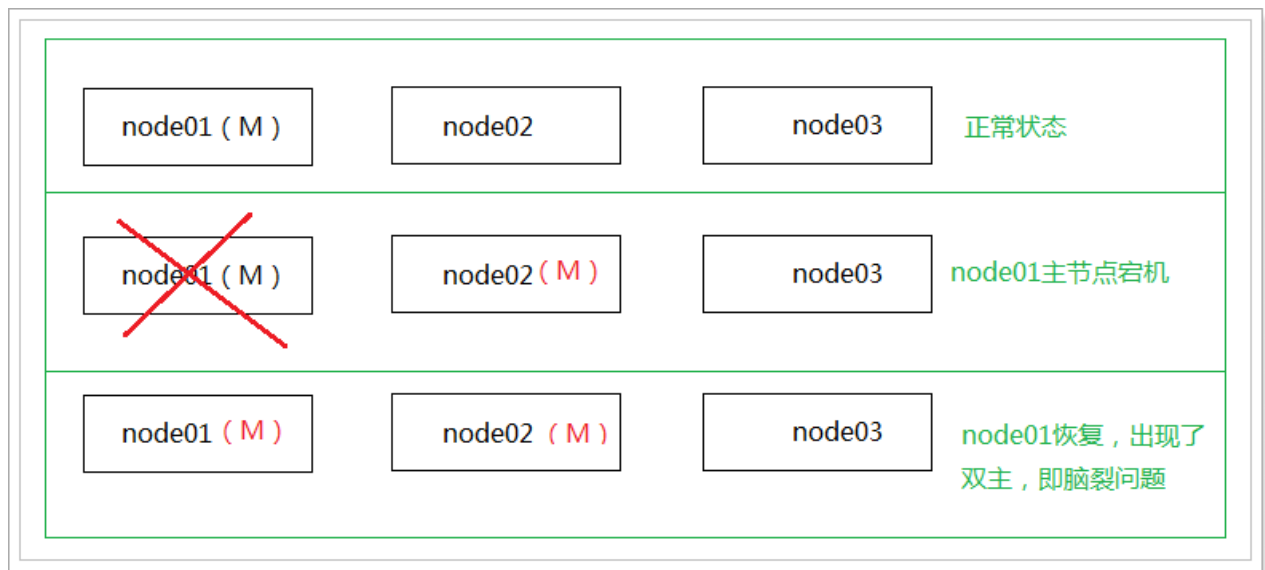
```
1 | ./node01/bin/elasticsearch
```

重启之后，发现node01可以正常加入到集群中，集群状态依然为绿色：



#### 特别说明：

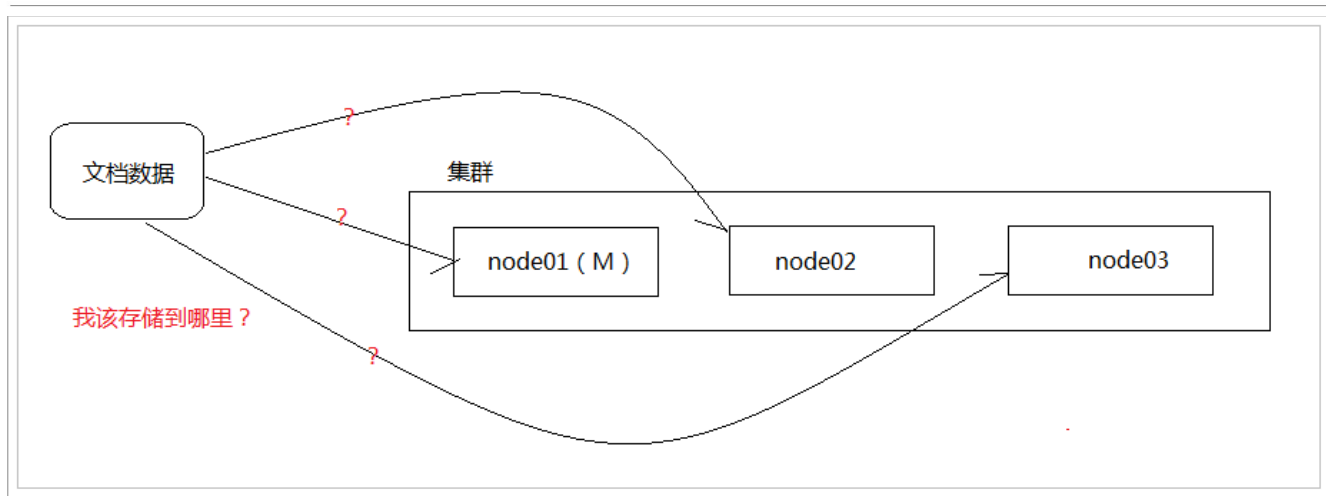
如果在配置文件中discovery.zen.minimum\_master\_nodes设置的不是N/2+1时，会出现脑裂问题，之前宕机的主节点恢复后不会加入到集群。



## 6.5、分布式文档

### 6.5.1、路由

首先，来看个问题：



如图所示：当我们想一个集群保存文档时，文档该存储到哪个节点呢？是随机吗？是轮询吗？

实际上，在Elasticsearch中，会采用计算的方式来确定存储到哪个节点，计算公式如下：

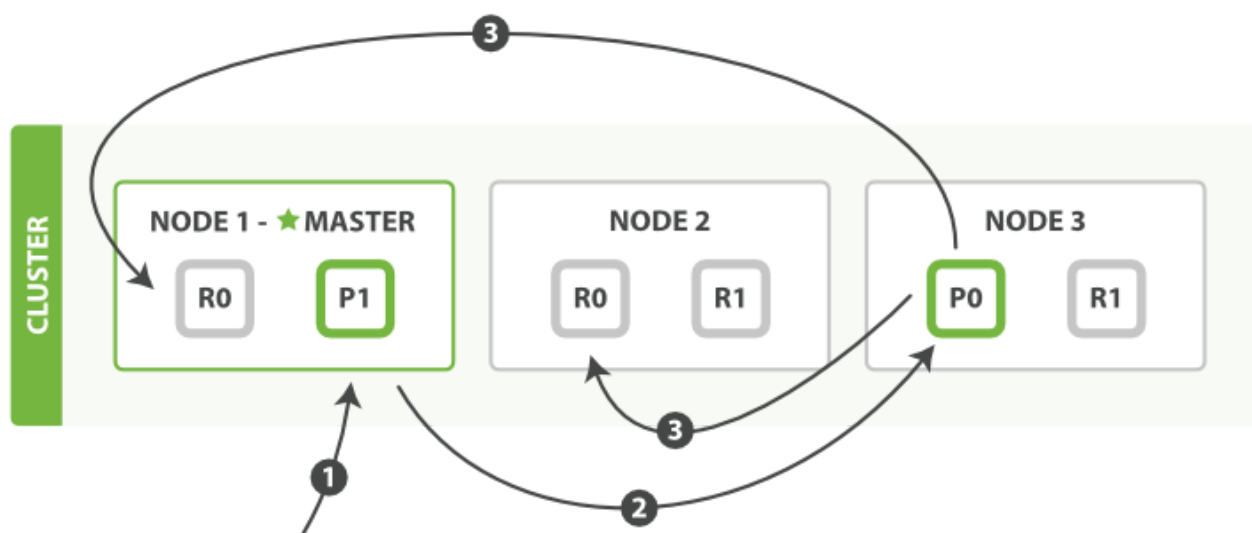
```
1 | shard = hash(routing) % number_of_primary_shards
```

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

这就是为什么创建了主分片后，不能修改的原因。

## 6.5.2、文档的写操作

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



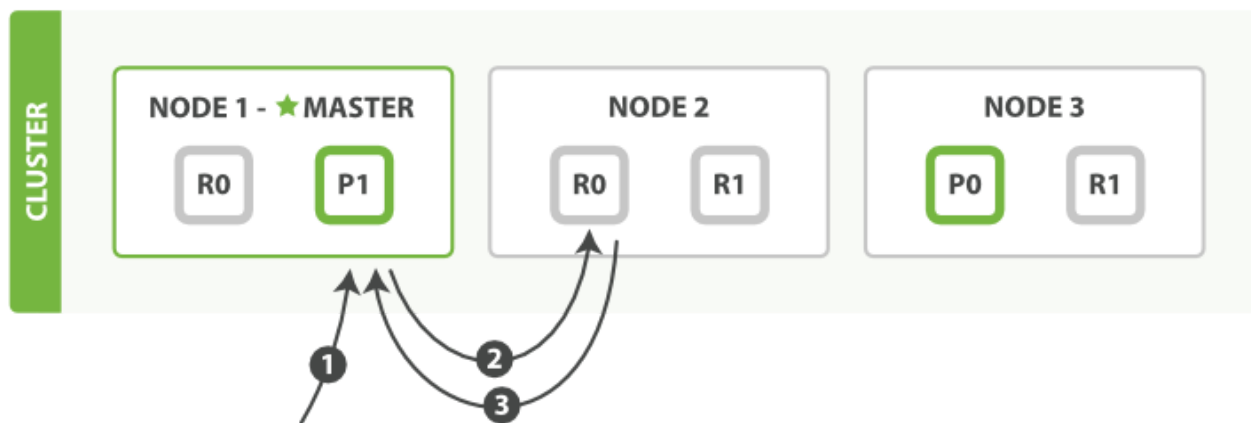
下面我们罗列在主分片和复制分片上成功新建、索引或删除一个文档必要的顺序步骤：

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

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

### 6.5.3、搜索文档（单个文档）

文档能够从主分片或任意一个复制分片被检索。



下面我们罗列在主分片或复制分片上检索一个文档必要的顺序步骤：

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

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

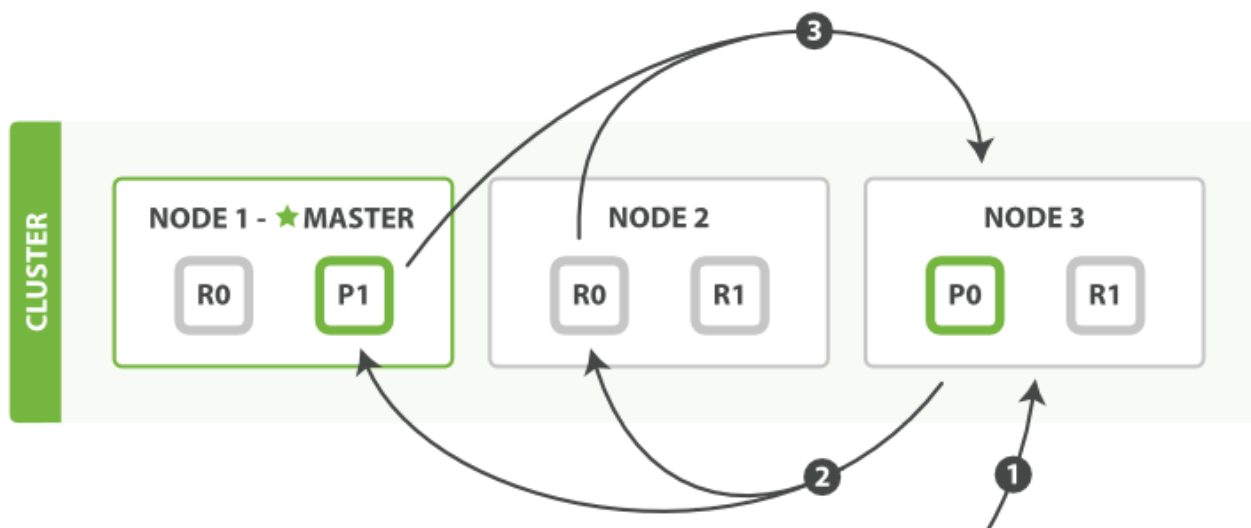
可能的情况是，一个被索引的文档已经存在于主分片上却还没来得及同步到复制分片上。这时复制分片会报告文档未找到，主分片会成功返回文档。一旦索引请求成功返回给用户，文档则在主分片和复制分片都是可用的。

### 6.5.4、全文搜索

对于全文搜索而言，文档可能分散在各个节点上，那么在分布式的情况下，如何搜索文档呢？

搜索，分为2个阶段，搜索（query）+取回（fetch）。

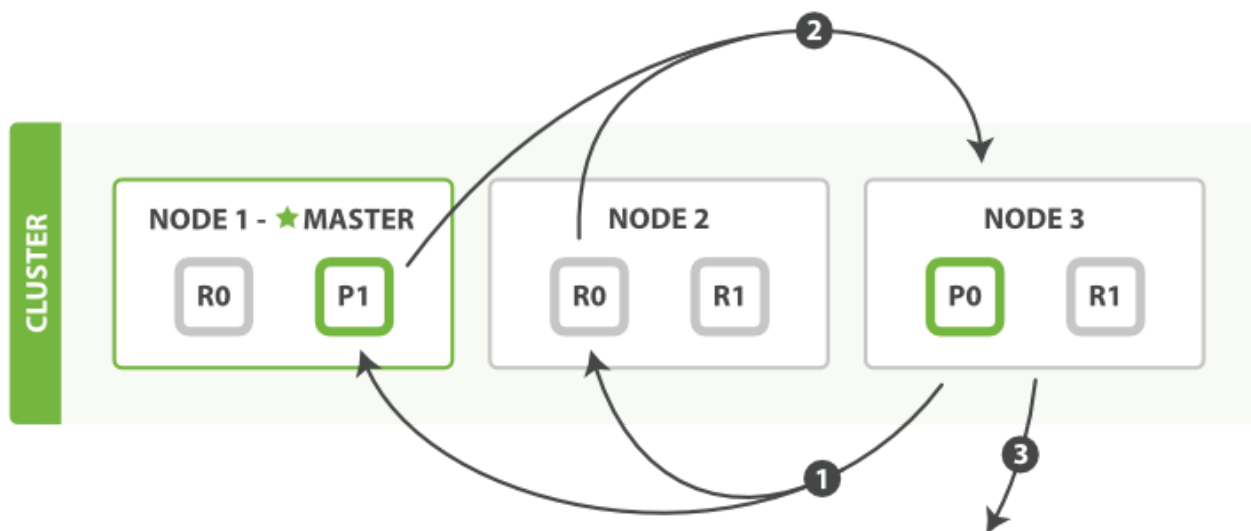
搜索（query）



查询阶段包含以下三步：

1. 客户端发送一个 `search` (搜索) 请求给 Node 3, Node 3 创建了一个长度为 `from+size` 的空优先级队
2. Node 3 转发这个搜索请求到索引中每个分片的原本或副本。每个分片在本地执行这个查询并且结果将结果到一个大小为 `from+size` 的有序本地优先队列里去。
3. 每个分片返回document的ID和它优先队列里的所有document的排序值给协调节点 Node 3。Node 3 把这些值合并到自己的优先队列里产生全局排序结果。

取回 ( fetch )



分发阶段由以下步骤构成：

1. 协调节点辨别出哪个document需要取回，并且向相关分片发出 `GET` 请求。
2. 每个分片加载document并且根据需要丰富 (enrich) 它们，然后再将document返回协调节点。
3. 一旦所有的document都被取回，协调节点会将结果返回给客户端。

## 7、Java客户端

在Elasticsearch中，为java提供了2种客户端，一种是REST风格的客户端，另一种是Java API的客户端。

<https://www.elastic.co/guide/en/elasticsearch/client/index.html>

## Elasticsearch Clients

- [Java REST Client \[6.5\] — other versions](#)
- [Java API \[6.5\] — other versions](#)
- [JavaScript API](#)
- [Groovy API \[2.4\] — other versions](#)
- [.NET API \[6.x\] — other versions](#)
- [PHP API \[6.0\] — other versions](#)
- [Perl API](#)
- [Python API](#)
- [Ruby API](#)
- [Community Contributed Clients](#)

### 7.1、REST客户端

Elasticsearch提供了2种REST客户端，一种是低级客户端，一种是高级客户端。

- Java Low Level REST Client：官方提供的低级客户端。该客户端通过http来连接Elasticsearch集群。用户在使用该客户端时需要将请求数据手动拼接成Elasticsearch所需JSON格式进行发送，收到响应时同样也需要将返回的JSON数据手动封装成对象。虽然麻烦，不过该客户端兼容所有的Elasticsearch版本。
- Java High Level REST Client：官方提供的高级客户端。该客户端基于低级客户端实现，它提供了很多便捷的API来解决低级客户端需要手动转换数据格式的问题。

### 7.2、构造数据

```
1 POST /haoke/house/_bulk
2
3 {"index":{"_index":"haoke","_type":"house"}}
4 {"id":"1001","title":"整租 · 南丹大楼 1居室 7500","price":"7500"}
5 {"index":{"_index":"haoke","_type":"house"}}
6 {"id":"1002","title":"陆家嘴板块，精装设计一室一厅，可拎包入住诚意租。","price":"8500"}
7 {"index":{"_index":"haoke","_type":"house"}}
8 {"id":"1003","title":"整租 · 健安坊 1居室 4050","price":"7500"}
9 {"index":{"_index":"haoke","_type":"house"}}
10 {"id":"1004","title":"整租 · 中凯城市之光+视野开阔+景色秀丽+拎包入住","price":"6500"}
11 {"index":{"_index":"haoke","_type":"house"}}
12 {"id":"1005","title":"整租 · 南京西路品质小区 21213三轨交汇 配套齐* 拎包入住","price":"6000"}
13 {"index":{"_index":"haoke","_type":"house"}}
14 {"id":"1006","title":"祥康里 简约风格 *南户型 拎包入住 看房随时","price":"7000"}
```



查询 6 个分片中用的 6 个, 6 命中, 耗时 0.012 秒

_index	_type	_id	_score ▲	id	title	price
haoke	house	F0pdE2gBCKv8opxuOj12	1	1003	整租·健安坊 1居室 4050	7500
haoke	house	FUpdE2gBCKv8opxuOj12	1	1001	整租·南丹大楼 1居室 7500	7500
haoke	house	GEpdE2gBCKv8opxuOj12	1	1004	整租·中凯城市之光+视野开阔+景色秀丽+拎包入住	6500
haoke	house	GkpdE2gBCKv8opxuOj12	1	1006	祥康里 简约风格 *南户型 拎包入住 看房随时	7000
haoke	house	GUpdE2gBCKv8opxuOj12	1	1005	整租·南京西路品质小区 21213三轨交汇 配套齐* 拎包入住	6000
haoke	house	FkpdE2gBCKv8opxuOj12	1	1002	陆家嘴板块, 精装设计一室一厅, 可拎包入住诚意租。	8500

## 7.3、REST低级客户端

### 7.3.1、创建工程

创建工程itcast-elasticsearch：

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <project xmlns="http://maven.apache.org/POM/4.0.0"
3      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4      xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
5      <modelVersion>4.0.0</modelVersion>
6
7      <groupId>cn.itcast.elasticsearch</groupId>
8      <artifactId>itcast-elasticsearch</artifactId>
9      <version>1.0-SNAPSHOT</version>
10
11     <dependencies>
12         <dependency>
13             <groupId>org.elasticsearch.client</groupId>
14             <artifactId>elasticsearch-rest-client</artifactId>
15             <version>6.5.4</version>
16         </dependency>
17         <dependency>
18             <groupId>junit</groupId>
19             <artifactId>junit</artifactId>
20             <version>4.12</version>
21         </dependency>
22
23         <dependency>
24             <groupId>com.fasterxml.jackson.core</groupId>
25             <artifactId>jackson-databind</artifactId>
26             <version>2.9.4</version>
27         </dependency>
28     </dependencies>
29
30     <build>
31         <plugins>
32             <!-- java编译插件 -->
33             <plugin>
34                 <groupId>org.apache.maven.plugins</groupId>

```



```
35         <artifactId>maven-compiler-plugin</artifactId>
36         <version>3.2</version>
37         <configuration>
38             <source>1.8</source>
39             <target>1.8</target>
40             <encoding>UTF-8</encoding>
41         </configuration>
42     </plugin>
43 </plugins>
44 </build>
45 </project>
```

### 7.3.2、编写测试用例

```
1 package cn.itcast.es.rest;
2
3 import com.fasterxml.jackson.databind.ObjectMapper;
4 import org.apache.http.HttpHost;
5 import org.apache.http.util.EntityUtils;
6 import org.elasticsearch.client.*;
7 import org.junit.After;
8 import org.junit.Before;
9 import org.junit.Test;
10
11 import java.io.IOException;
12 import java.util.HashMap;
13 import java.util.Map;
14
15 public class TestESREST {
16
17     private static final ObjectMapper MAPPER = new ObjectMapper();
18
19     private RestClient restClient;
20
21     @Before
22     public void init() {
23         RestClientBuilder restClientBuilder = RestClient.builder(
24             new HttpHost("172.16.55.185", 9200, "http"),
25             new HttpHost("172.16.55.185", 9201, "http"),
26             new HttpHost("172.16.55.185", 9202, "http"));
27
28         restClientBuilder.setFailureListener(new RestClient.FailureListener() {
29             @Override
30             public void onFailure(Node node) {
31                 System.out.println("出错了 -> " + node);
32             }
33         });
34
35         this.restClient = restClientBuilder.build();
36     }
37
38     @After
39     public void after() throws IOException {
```



```
40     restClient.close();
41 }
42
43 // 查询集群状态
44 @Test
45 public void testGetInfo() throws IOException {
46     Request request = new Request("GET", "/_cluster/state");
47     request.addParameter("pretty", "true");
48     Response response = this.restClient.performRequest(request);
49
50     System.out.println(response.getStatusLine());
51     System.out.println(EntityUtils.toString(response.getEntity()));
52 }
53
54
55 // 新增数据
56 @Test
57 public void testCreateData() throws IOException {
58     Request request = new Request("POST", "/haoke/house");
59
60     Map<String, Object> data = new HashMap<>();
61     data.put("id", "2001");
62     data.put("title", "张江高科");
63     data.put("price", "3500");
64
65     request.setJsonEntity(MAPPER.writeValueAsString(data));
66     Response response = this.restClient.performRequest(request);
67
68     System.out.println(response.getStatusLine());
69     System.out.println(EntityUtils.toString(response.getEntity()));
70 }
71
72 // 根据id查询数据
73 @Test
74 public void testQueryData() throws IOException {
75     Request request = new Request("GET", "/haoke/house/G0pFE2gBCKv8opxuRz1y");
76
77     Response response = this.restClient.performRequest(request);
78
79     System.out.println(response.getStatusLine());
80     System.out.println(EntityUtils.toString(response.getEntity()));
81 }
82
83 // 搜索数据
84 @Test
85 public void testSearchData() throws IOException {
86     Request request = new Request("POST", "/haoke/house/_search");
87     String searchJson = "{\"query\": {\"match\": {\"title\": \"拎包入住\"}}}";
88     request.setJsonEntity(searchJson);
89     request.addParameter("pretty", "true");
90
91     Response response = this.restClient.performRequest(request);
92 }
```

```
93     System.out.println(response.getStatusLine());
94     System.out.println(EntityUtils.toString(response.getEntity()));
95 }
96
97
98 }
99
```

从使用中，可以看出，基本和我们使用RESTful api使用几乎是一致的。

## 7.4、REST高级客户端

### 7.4.1、引入依赖

```
1 <dependency>
2   <groupId>org.elasticsearch.client</groupId>
3   <artifactId>elasticsearch-rest-high-level-client</artifactId>
4   <version>6.5.4</version>
5 </dependency>
```

### 7.4.2、编写测试用例

```
1 package cn.itcast.es.rest;
2
3 import org.apache.http.HttpHost;
4 import org.elasticsearch.action.ActionListener;
5 import org.elasticsearch.action.delete.DeleteRequest;
6 import org.elasticsearch.action.delete.DeleteResponse;
7 import org.elasticsearch.action.get.GetRequest;
8 import org.elasticsearch.action.get.GetResponse;
9 import org.elasticsearch.action.index.IndexRequest;
10 import org.elasticsearch.action.index.IndexResponse;
11 import org.elasticsearch.action.search.SearchRequest;
12 import org.elasticsearch.action.search.SearchResponse;
13 import org.elasticsearch.action.update.UpdateRequest;
14 import org.elasticsearch.action.update.UpdateResponse;
15 import org.elasticsearch.client.RequestOptions;
16 import org.elasticsearch.client.RestClient;
17 import org.elasticsearch.client.RestClientBuilder;
18 import org.elasticsearch.client.RestHighLevelClient;
19 import org.elasticsearch.common.Strings;
20 import org.elasticsearch.common.unit.TimeValue;
21 import org.elasticsearch.index.query.QueryBuilders;
22 import org.elasticsearch.search.SearchHit;
23 import org.elasticsearch.search.SearchHits;
24 import org.elasticsearch.search.builder.SearchSourceBuilder;
25 import org.elasticsearch.search.fetch.subphase.FetchSourceContext;
26 import org.junit.After;
27 import org.junit.Before;
28 import org.junit.Test;
29
30 import java.util.HashMap;
```



```
31 import java.util.Map;
32 import java.util.concurrent.TimeUnit;
33
34 public class TestRestHighLevel {
35
36     private RestHighLevelClient client;
37
38     @Before
39     public void init() {
40         RestClientBuilder restClientBuilder = RestClient.builder(
41             new HttpHost("172.16.55.185", 9200, "http"),
42             new HttpHost("172.16.55.185", 9201, "http"),
43             new HttpHost("172.16.55.185", 9202, "http"));
44
45         this.client = new RestHighLevelClient(restClientBuilder);
46     }
47
48     @After
49     public void after() throws Exception {
50         this.client.close();
51     }
52
53     /**
54      * 新增文档，同步操作
55      *
56      * @throws Exception
57      */
58     @Test
59     public void testCreate() throws Exception {
60
61         Map<String, Object> data = new HashMap<>();
62         data.put("id", "2002");
63         data.put("title", "南京西路 拎包入住 一室一厅");
64         data.put("price", "4500");
65
66         IndexRequest indexRequest = new IndexRequest("haoke", "house")
67             .source(data);
68
69         IndexResponse indexResponse = this.client.index(indexRequest,
70             RequestOptions.DEFAULT);
71         System.out.println("id->" + indexResponse.getId());
72         System.out.println("index->" + indexResponse.getIndex());
73         System.out.println("type->" + indexResponse.getType());
74         System.out.println("version->" + indexResponse.getVersion());
75         System.out.println("result->" + indexResponse.getResult());
76         System.out.println("shardInfo->" + indexResponse.getShardInfo());
77     }
78
79     /**
80      * 新增文档，异步操作
81      *
82      * @throws Exception
```



```
83      */
84      @Test
85      public void testCreateAsync() throws Exception {
86
87          Map<String, Object> data = new HashMap<>();
88          data.put("id", "2003");
89          data.put("title", "南京东路 最新房源 二室一厅");
90          data.put("price", "5500");
91
92          IndexRequest indexRequest = new IndexRequest("haoke", "house")
93              .source(data);
94
95          this.client.indexAsync(indexRequest, RequestOptions.DEFAULT, new
96          ActionListener<IndexResponse>() {
97
98              @Override
99              public void onResponse(IndexResponse indexResponse) {
100                  System.out.println("id->" + indexResponse.getId());
101                  System.out.println("index->" + indexResponse.getIndex());
102                  System.out.println("type->" + indexResponse.getType());
103                  System.out.println("version->" + indexResponse.getVersion());
104                  System.out.println("result->" + indexResponse.getResult());
105                  System.out.println("shardInfo->" + indexResponse.getShardInfo());
106              }
107
108              @Override
109              public void onFailure(Exception e) {
110                  System.out.println(e);
111              }
112          });
113
114          System.out.println("ok");
115
116          Thread.sleep(20000);
117      }
118
119      @Test
120      public void testQuery() throws Exception {
121          GetRequest getRequest = new GetRequest("haoke", "house",
122          "GkpdE2gBCKv8opxuOj12");
123
124          // 指定返回的字段
125          String[] includes = new String[]{"title", "id"};
126          String[] excludes = Strings.EMPTY_ARRAY;
127          FetchSourceContext fetchSourceContext =
128              new FetchSourceContext(true, includes, excludes);
129          getRequest.fetchSourceContext(fetchSourceContext);
130
131          GetResponse response = this.client.get(getRequest, RequestOptions.DEFAULT);
132
133          System.out.println("数据 -> " + response.getSource());
134      }
```



```
134
135     /**
136     * 判断是否存在
137     *
138     * @throws Exception
139     */
140     @Test
141     public void testExists() throws Exception {
142         GetRequest getRequest = new GetRequest("haoke", "house",
143         "Gkpde2gBCKv8opxu0j12");
144
145         // 不返回的字段
146         getRequest.fetchSourceContext(new FetchSourceContext(false));
147
148         boolean exists = this.client.exists(getRequest, RequestOptions.DEFAULT);
149
150         System.out.println("exists -> " + exists);
151     }
152
153     /**
154     * 删除数据
155     *
156     * @throws Exception
157     */
158     @Test
159     public void testDelete() throws Exception {
160         DeleteRequest deleteRequest = new DeleteRequest("haoke", "house",
161         "Gkpde2gBCKv8opxu0j12");
162         DeleteResponse response = this.client.delete(deleteRequest,
163         RequestOptions.DEFAULT);
164         System.out.println(response.status()); // OK or NOT_FOUND
165     }
166
167     /**
168     * 更新数据
169     *
170     * @throws Exception
171     */
172     @Test
173     public void testUpdate() throws Exception {
174         UpdateRequest updateRequest = new UpdateRequest("haoke", "house",
175         "G0pfe2gBCKv8opxuRz1y");
176
177         Map<String, Object> data = new HashMap<>();
178         data.put("title", "张江高科2");
179         data.put("price", "5000");
180
181         updateRequest.doc(data);
182
183         UpdateResponse response = this.client.update(updateRequest,
184         RequestOptions.DEFAULT);
185         System.out.println("version -> " + response.getVersion());
186     }
187 }
```



```
182
183     /**
184     * 测试搜索
185     *
186     * @throws Exception
187     */
188     @Test
189     public void testSearch() throws Exception {
190         SearchRequest searchRequest = new SearchRequest("haoke");
191         searchRequest.types("house");
192
193         SearchSourceBuilder sourceBuilder = new SearchSourceBuilder();
194         sourceBuilder.query(QueryBuilders.matchQuery("title", "拎包入住"));
195         sourceBuilder.from(0);
196         sourceBuilder.size(5);
197         sourceBuilder.timeout(new TimeValue(60, TimeUnit.SECONDS));
198
199         searchRequest.source(sourceBuilder);
200
201         SearchResponse search = this.client.search(searchRequest,
202             RequestOptions.DEFAULT);
203         System.out.println("搜索到 " + search.getHits().totalHits + " 条数据.");
204         SearchHits hits = search.getHits();
205         for (SearchHit hit : hits) {
206             System.out.println(hit.getSourceAsString());
207         }
208     }
209 }
210
```