

## 课程介绍

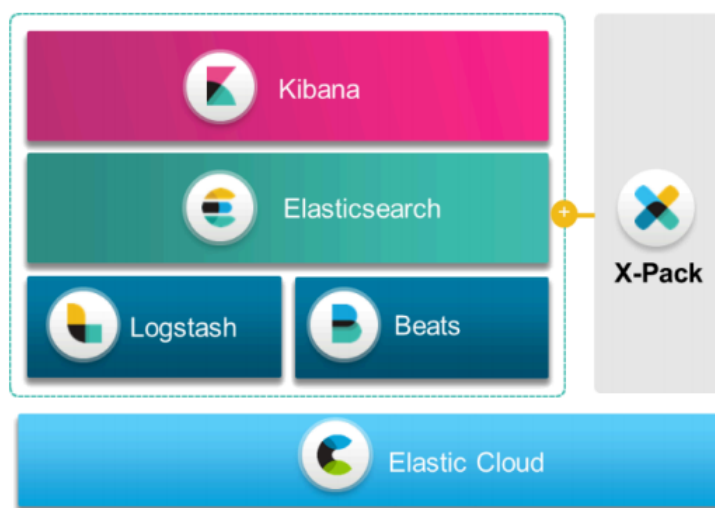
- Elastic Stack简介
- Elasticsearch的介绍与安装
- Elasticsearch的快速入门
- Elasticsearch的核心讲解
- 中文分词

## 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。

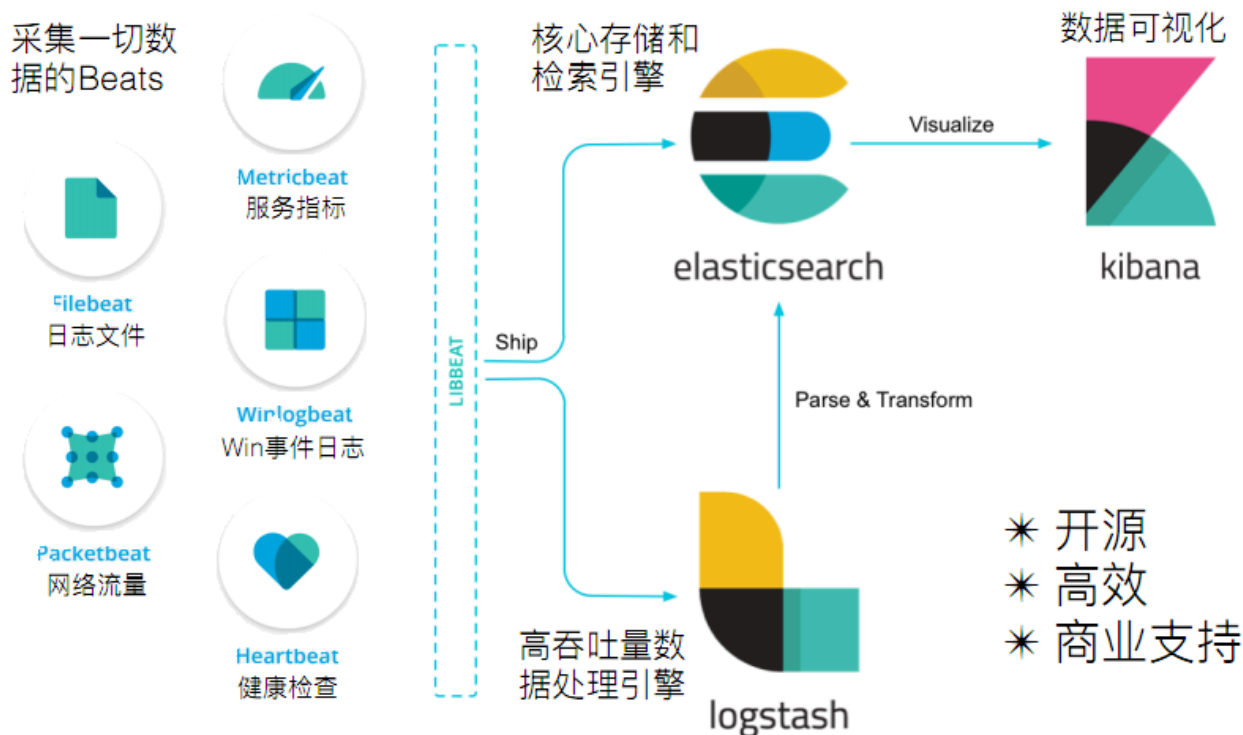


## The Brand New ElasticStack



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 /haoke/es/
```



```
6
7 #修改配置文件
8 vim conf/elasticsearch.yml
9 network.host: 172.16.55.185 #绑定的地址
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，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：
12 #1：修改jvm启动参数
13 vim conf/jvm.options
14 -Xms128m #根据自己机器情况修改
15 -Xmx128m
16 #2：单个进程中的最大线程数
17 vim /etc/sysctl.conf
18 vm.max_map_count=655360
19
20 #启动ES服务
21 su - elsearch
22 cd bin
23 ./elasticsearch 或 ./elasticsearch -d #后台系统
24
25 #通过访问http://172.16.55.185:9200进行测试，看到如下信息，就说明ES启动成功了
26 {
27     "name": "dsQV6I8",
28     "cluster_name": "elasticsearch",
29     "cluster_uuid": "v5GPTWAtT5emxFdjigFg-w",
30     "version": {
31         "number": "6.5.4",
32         "build_flavor": "default",
33         "build_type": "tar",
34         "build_hash": "d2ef93d",
35         "build_date": "2018-12-17T21:17:40.758843Z",
36         "build_snapshot": false,
37         "lucene_version": "7.5.0",
38         "minimum_wire_compatibility_version": "5.6.0",
39         "minimum_index_compatibility_version": "5.0.0"
40     },
41     "tagline": "You know, for Search"
42 }
43
44 #停止服务
45 root@itcast:~# jps
46 68709 Jps
47 68072 Elasticsearch
48
49 kill 68072 #通过kill结束进程
```

## 2.2.4、使用docker安装



```
1 #拉取镜像
2 docker pull elasticsearch:6.5.4
3
4 #创建容器
5 docker create --name elasticsearch --net host -e "discovery.type=single-node" -e
  "network.host=172.16.55.185" elasticsearch:6.5.4
6
7 #启动
8 docker start elasticsearch
9
10 #查看日志
11 docker logs elasticsearch
```



可以看到，效果一样。

需要说明的是：此docker安装是开发环境模式，并没有配置目录挂载等内容，生成集群环境的搭建后面讲解。

## 2.2.5、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>

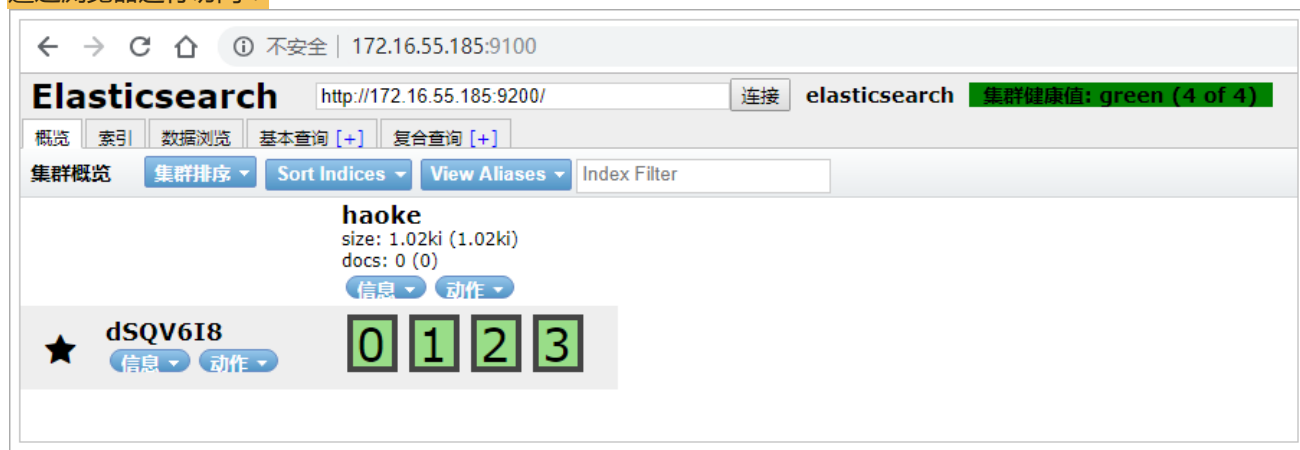
head提供了4种安装方式：

- 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/ffmkiejjme colpfoofpjologoblkegm>



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docker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。
- 可以把Elasticsearch的索引看成MongoDB里的一个集合。
- 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 http://172.16.55.185:9200/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 http://172.16.55.185:9200/haoke
14 {
15     "acknowledged": true
16 }
```



## 2.4.2、插入数据

URL规则：

POST <http://172.16.55.185:9200/{索引}/{类型}/{id}>

```
1 POST http://172.16.55.185:9200/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, the URL is `http://172.16.55.185:9200/` and the cluster is `docker-cluster` with a health status of `green (2 of 2)`. The left sidebar shows the '数据浏览' (Data Browser) section with a tree view of the index `haoke` and type `user`. The main area displays a search result for the query `haoke` with the following table:

_index	_type	_id	_score	id	name	age	sex
haoke	user	1001	1	1001	张三	20	男

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 http://172.16.55.185:9200/haoke/user/
2 {
3   "id":1002,
4   "name":"张三",
5   "age":20,
6   "sex":"男"
7 }
```

The screenshot shows the Elasticsearch web interface with two search results for the query `haoke`. The table displays the following data:

_index	_type	_id	_score	id	name	age	sex
haoke	user	1001	1	1001	张三	20	男
haoke	user	BbPe_WcB9cFOnF3uebvr	1	1002	张三	20	男

A red box highlights the auto-generated `_id` `BbPe_WcB9cFOnF3uebvr` for the second document, with a red arrow pointing to it and the text **自动生成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 http://172.16.55.185:9200/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 http://172.16.55.185:9200/haoke/user/1001/_update
3
4 {
5   "doc": {
6     "age": 23
7   }
8 }
9
```



```
Raw  JSON  Response
Copy to clipboard  Save as file
{
  _index: "haoke"
  _type: "user"
  id: "1001"
  _version: 3
  result: "updated"
  _shards: {
    total: 1
    successful: 1
    failed: 0
  }
  _seq_no: 14
  _primary_term: 1
}
```

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_WcB9cFOhF3uebvr	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 http://172.16.55.185:9200/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 http://172.16.55.185:9200/haoke/user/BbPe_wCB9cFOnF3uebvr
2
3 #返回的数据如下
4 {
5   "_index": "haoke",
6   "_type": "user",
7   "_id": "BbPe_wCB9cFOnF3uebvr",
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 http://172.16.55.185:9200/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_wCB9cFOnF3uebvr",
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 http://172.16.55.185:9200/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 http://172.16.55.185:9200/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 http://172.16.55.185:9200/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     "must": {
13         "match": {
14             "sex": "男"
15         }
16     }
17 }
18 }
19 }
20 }
```

查询结果：

The screenshot shows a REST client interface with tabs for 'Raw', 'JSON', and 'Response'. The 'JSON' tab is selected, displaying the following response:

```
{
  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"
        }
      }
    ]
  }
}
```

A red box highlights the `_source` object in the hit, which contains the user's details: `id: 1002`, `name: "李四"`, `age: 40`, and `sex: "男"`.

全文搜索



```
1 POST http://172.16.55.185:9200/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 http://172.16.55.185:9200/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 http://172.16.55.185:9200/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

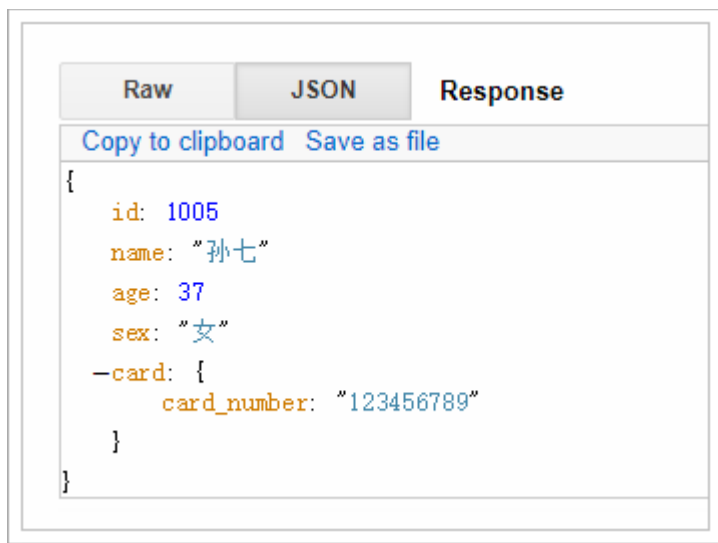
在响应的数据中，如果我们不需要全部的字段，可以指定某些需要的字段进行返回。

```
1 GET http://172.16.55.185:9200/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：

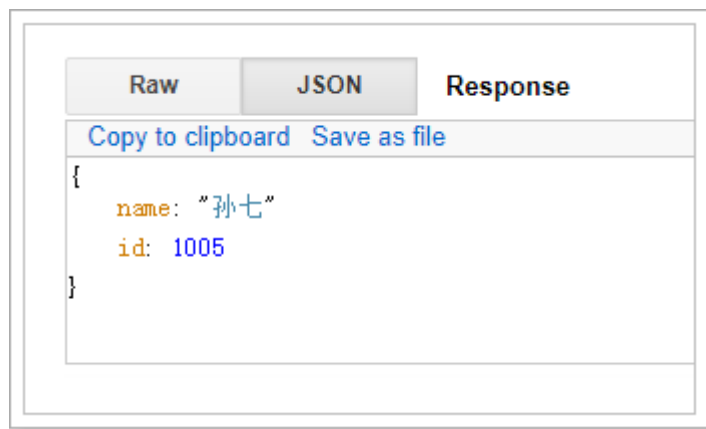
```
1 GET http://172.16.55.185:9200/haoke/user/1005/_source
```





还可以这样：

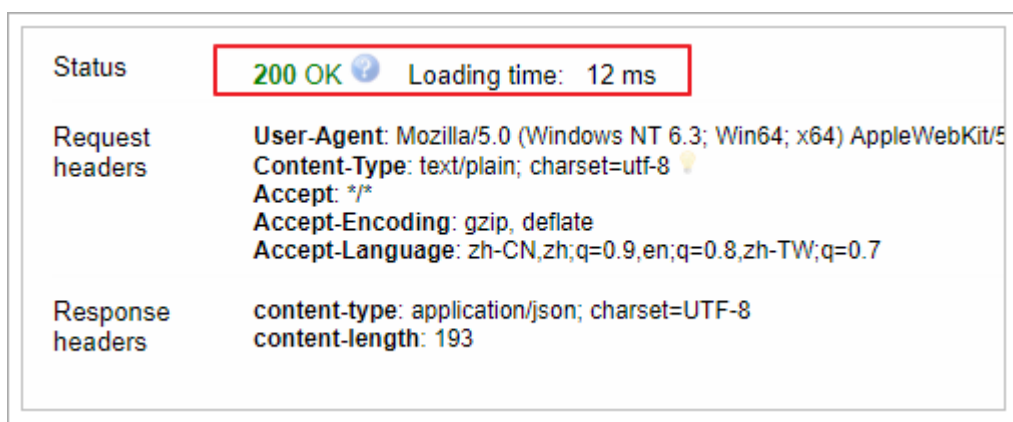
```
1 | GET http://172.16.55.185:9200/haoke/user/1005/_source?_source=id,name
```



### 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 http://172.16.55.185:9200/haoke/user/1005
```



```
1 | HEAD http://172.16.55.185:9200/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。

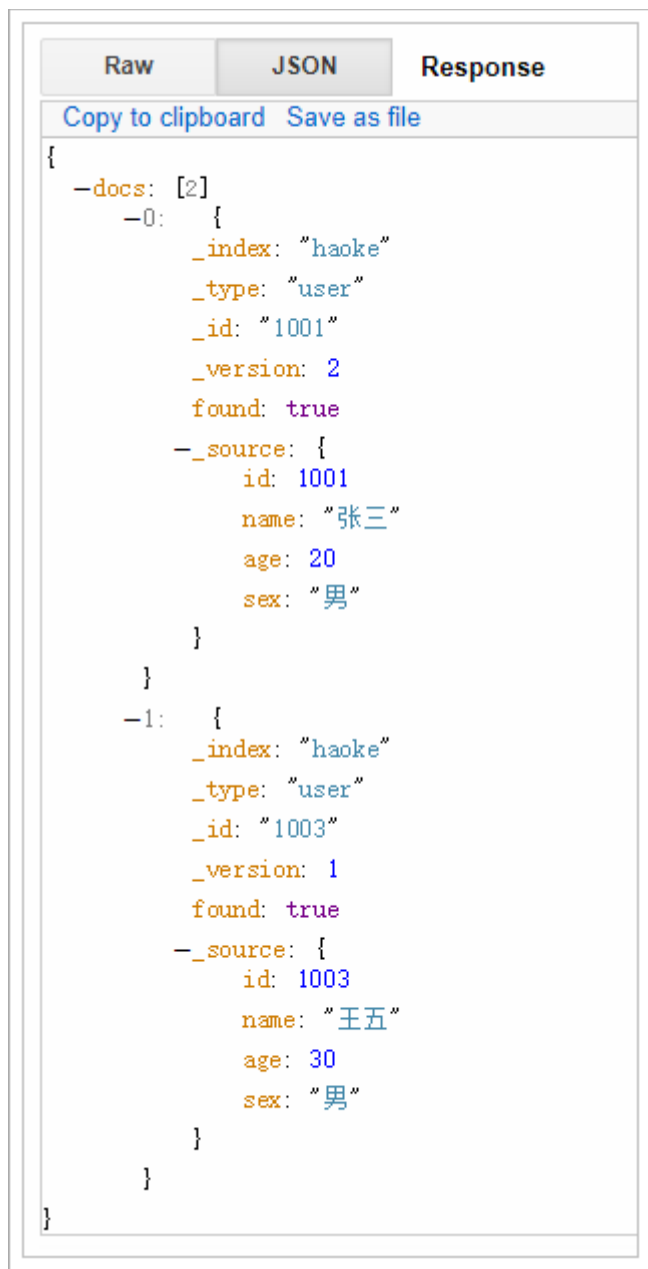
## 3.4、批量操作

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 http://172.16.55.185:9200/haoke/user/_mget
2 |
3 | {
4 |   "ids" : [ "1001", "1003" ]
5 | }
```

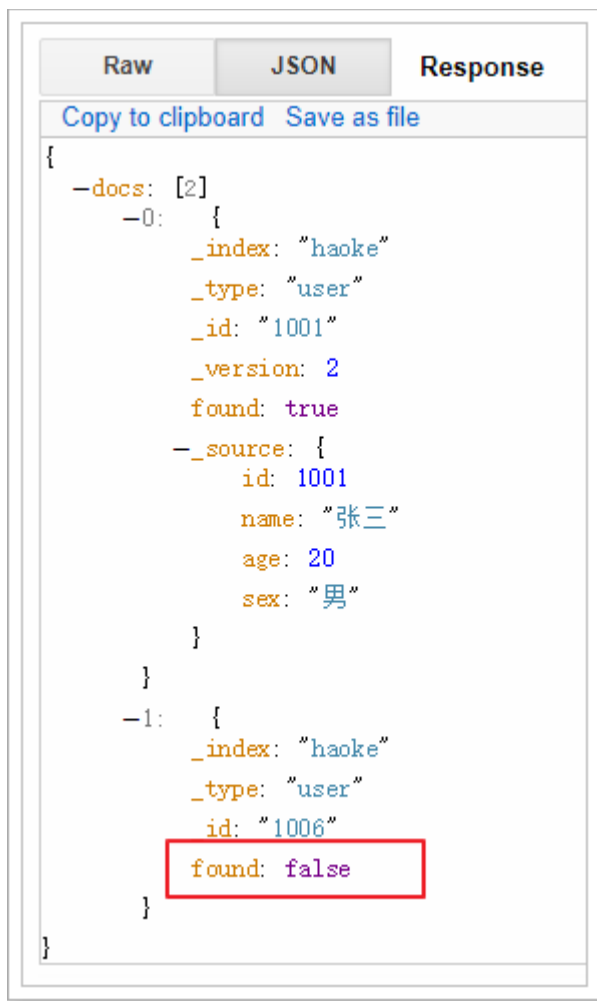
结果：



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 http://172.16.55.185:9200/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
```

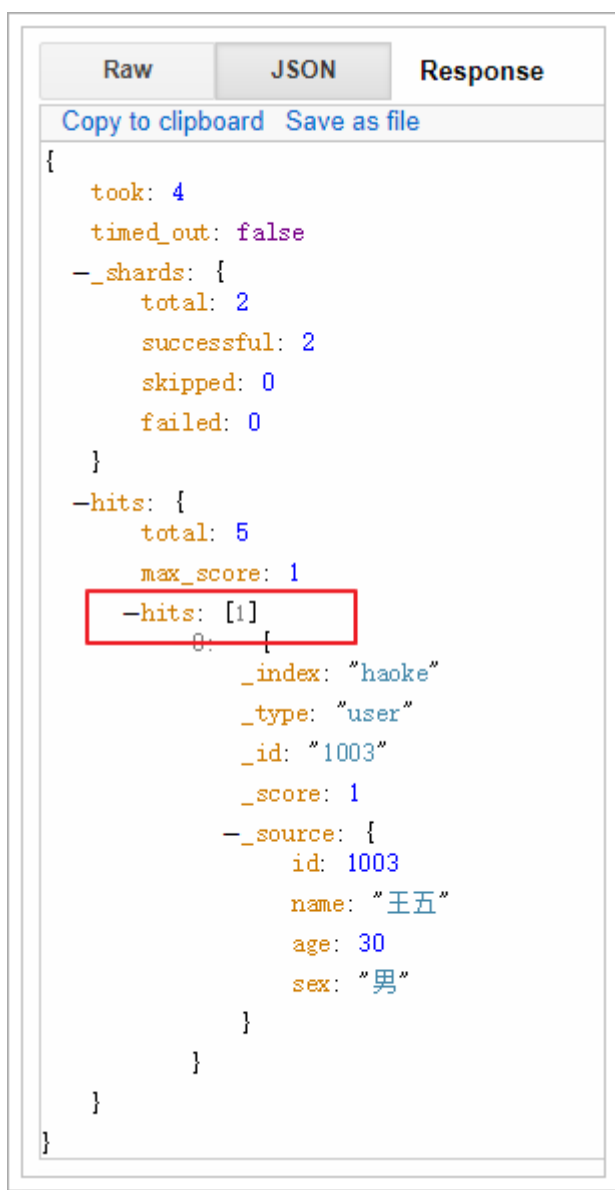
如果你想每页显示5个结果，页码从1到3，那请求如下：



```
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 http://172.16.55.185:9200/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：

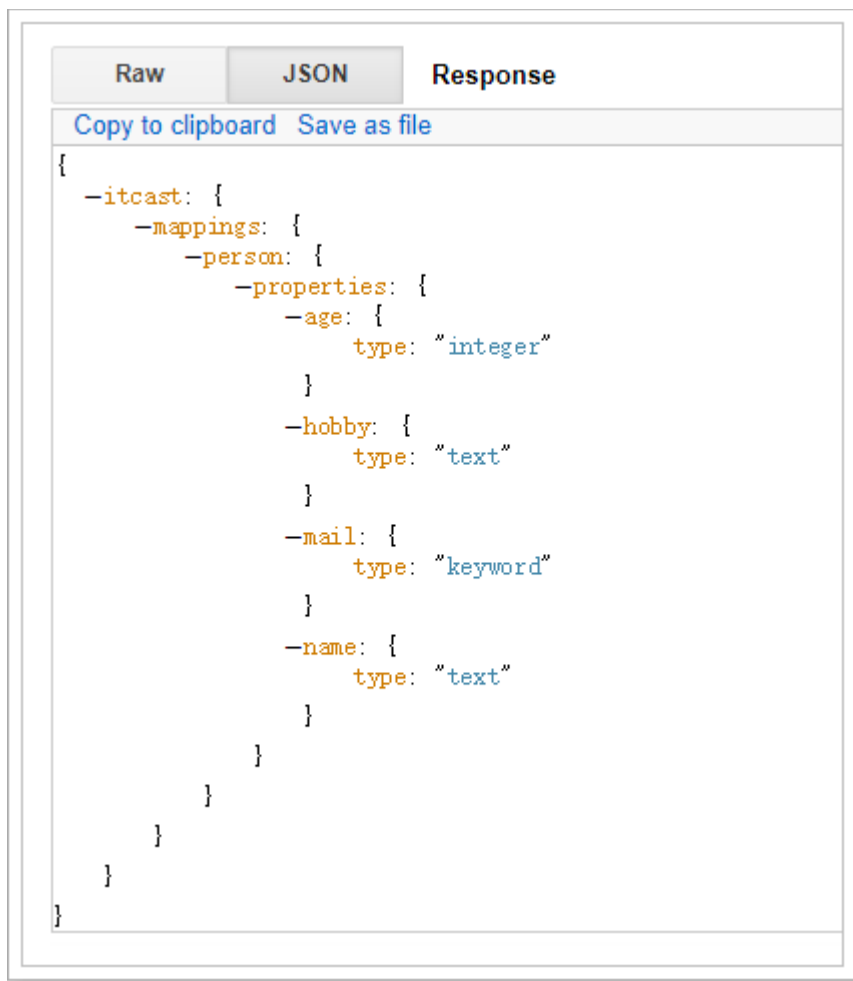
```
1 PUT http://172.16.55.185:9200/itcast
2 {
3   "settings": {
4     "index": {
5       "number_of_shards": "2",
```



```
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 http://172.16.55.185:9200/itcast/_mapping
```



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"}
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 http://172.16.55.185:9200/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 http://172.16.55.185:9200/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做匹配：

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

示例：



```
1 POST http://172.16.55.185:9200/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       "lt": 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 http://172.16.55.185:9200/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 http://172.16.55.185:9200/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 http://172.16.55.185:9200/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 http://172.16.55.185:9200/_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 http://172.16.55.185:9200/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
    }
  }
}
```

## 4.3、内置分词

### 4.3.1、Standard

Standard 标准分词，按单词切分，并且会转化成小写

```
1 POST http://172.16.55.185:9200/_analyze
2 {
3   "analyzer": "standard",
4   "text": "A man becomes learned by asking questions."
5 }
```

结果：

```
1 {
2   "tokens": [
3     {
4       "token": "a",
5       "start_offset": 0,
6       "end_offset": 1,
7       "type": "<ALPHANUM>",
8       "position": 0
```



```
9      },
10     {
11         "token": "man",
12         "start_offset": 2,
13         "end_offset": 5,
14         "type": "<ALPHANUM>",
15         "position": 1
16     },
17     {
18         "token": "becomes",
19         "start_offset": 6,
20         "end_offset": 13,
21         "type": "<ALPHANUM>",
22         "position": 2
23     },
24     {
25         "token": "learned",
26         "start_offset": 14,
27         "end_offset": 21,
28         "type": "<ALPHANUM>",
29         "position": 3
30     },
31     {
32         "token": "by",
33         "start_offset": 22,
34         "end_offset": 24,
35         "type": "<ALPHANUM>",
36         "position": 4
37     },
38     {
39         "token": "asking",
40         "start_offset": 25,
41         "end_offset": 31,
42         "type": "<ALPHANUM>",
43         "position": 5
44     },
45     {
46         "token": "questions",
47         "start_offset": 32,
48         "end_offset": 41,
49         "type": "<ALPHANUM>",
50         "position": 6
51     }
52 ]
53 }
```

### 4.3.2、Simple

Simple分词器，按照非单词切分，并且做小写处理





```
1 POST http://172.16.55.185:9200/_analyze
2 {
3     "analyzer": "simple",
4     "text": "If the document doesn't already exist"
5 }
```

结果：

```
1 {
2     "tokens": [
3         {
4             "token": "if",
5             "start_offset": 0,
6             "end_offset": 2,
7             "type": "word",
8             "position": 0
9         },
10        {
11            "token": "the",
12            "start_offset": 3,
13            "end_offset": 6,
14            "type": "word",
15            "position": 1
16        },
17        {
18            "token": "document",
19            "start_offset": 7,
20            "end_offset": 15,
21            "type": "word",
22            "position": 2
23        },
24        {
25            "token": "doesn",
26            "start_offset": 16,
27            "end_offset": 21,
28            "type": "word",
29            "position": 3
30        },
31        {
32            "token": "t",
33            "start_offset": 22,
34            "end_offset": 23,
35            "type": "word",
36            "position": 4
37        },
38        {
39            "token": "already",
40            "start_offset": 24,
41            "end_offset": 31,
42            "type": "word",
43            "position": 5
44        },
45    ]
46 }
```



```
45     {
46         "token": "exist",
47         "start_offset": 32,
48         "end_offset": 37,
49         "type": "word",
50         "position": 6
51     }
52 ]
53 }
```

### 4.3.3、Whitespace

Whitespace是按照空格切分。

示例：

```
1 POST http://172.16.55.185:9200/_analyze
2 {
3     "analyzer": "whitespace",
4     "text": "If the document doesn't already exist"
5 }
```

结果：

```
1 {
2     "tokens": [
3         {
4             "token": "If",
5             "start_offset": 0,
6             "end_offset": 2,
7             "type": "word",
8             "position": 0
9         },
10        {
11            "token": "the",
12            "start_offset": 3,
13            "end_offset": 6,
14            "type": "word",
15            "position": 1
16        },
17        {
18            "token": "document",
19            "start_offset": 7,
20            "end_offset": 15,
21            "type": "word",
22            "position": 2
23        },
24        {
25            "token": "doesn't",
26            "start_offset": 16,
27            "end_offset": 23,
28            "type": "word",
```



```
29         "position": 3
30     },
31     {
32         "token": "already",
33         "start_offset": 24,
34         "end_offset": 31,
35         "type": "word",
36         "position": 4
37     },
38     {
39         "token": "exist",
40         "start_offset": 32,
41         "end_offset": 37,
42         "type": "word",
43         "position": 5
44     }
45 ]
46 }
```

#### 4.3.4、Stop

Stop分词器，是去除Stop Word语气助词，如the、an等。

```
1 POST http://172.16.55.185:9200/_analyze
2 {
3     "analyzer": "stop",
4     "text": "If the document doesn't already exist"
5 }
```

结果：

```
1 {
2     "tokens": [
3         {
4             "token": "document",
5             "start_offset": 7,
6             "end_offset": 15,
7             "type": "word",
8             "position": 2
9         },
10        {
11            "token": "doesn",
12            "start_offset": 16,
13            "end_offset": 21,
14            "type": "word",
15            "position": 3
16        },
17        {
18            "token": "t",
19            "start_offset": 22,
20            "end_offset": 23,
21            "type": "word",
```



```
22         "position": 4
23     },
24     {
25         "token": "already",
26         "start_offset": 24,
27         "end_offset": 31,
28         "type": "word",
29         "position": 5
30     },
31     {
32         "token": "exist",
33         "start_offset": 32,
34         "end_offset": 37,
35         "type": "word",
36         "position": 6
37     }
38 ]
39 }
```

### 4.3.5、Keyword

Keyword分词器，意思是传入就是关键词，不做分词处理。

```
1 POST http://172.16.55.185:9200/_analyze
2 {
3     "analyzer": "keyword",
4     "text": "If the document doesn't already exist"
5 }
```

结果：

```
1 {
2     "tokens": [
3         {
4             "token": "If the document doesn't already exist",
5             "start_offset": 0,
6             "end_offset": 37,
7             "type": "word",
8             "position": 0
9         }
10    ]
11 }
```

## 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Luce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
3  #如果使用docker运行
4  docker cp /tmp/elasticsearch-analysis-ik-6.5.4.zip
   elasticsearch:/usr/share/elasticsearch/plugins/
5  #进入容器
6  docker exec -it elasticsearch /bin/bash
7  mkdir /usr/share/elasticsearch/plugins/ik
8  cd /usr/share/elasticsearch/plugins/ik
9  unzip elasticsearch-analysis-ik-6.5.4.zip
10
11 #重启容器即可
12 docker restart elasticsearch
```

测试：

```
1  POST http://172.16.55.185:9200/_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。