# ELK

赵东阳

Elasticsearch

Logstash、kibana-日志分析

## 学习目标：

1. 能够理解ElasticSearch的作用以及实际生产环境下的应用场景

2. 完成ElasticSearch服务安装--6.8，7

3. 理解ElasticSearch的相关概念以及对数据的存储方式

4. 能够使用Postman、kibana等工具完成ElasticSearch初步数据操作

5. 理解ElasticSearch工作原理（索引、analysis、mapping等）

6. 能够使用ElasticSearch集成IK分词器

7. 完成对ElasticSearch的索引和文档的CRUD操作

8. Request Body Search和Search API(URI)学习

9. 了解ElasticSearch集群优势

10. 完成ElasticSearch集群搭建及集群监控

11. 理解ElasticSearch集群内部运行机制

12. 了解Logstash使用方法

13. 完成Logstash安装以及

## ElasticSearch简介

### 1.1、Elasticsearch

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

**特点：**

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

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

ES能做什么？

全文检索（全部字段）、模糊查询（搜索）、数据分析（提供分析语法，例如聚合）

### 1.2、Elasticsearch使用案例

（1）2013年初，GitHub抛弃了Solr，采取ElasticSearch 来做PB级的搜索。 “GitHub使用ElasticSearch搜索20TB的数据，包括13亿文件和1300亿行代码”

（2）维基百科：启动以elasticsearch为基础的核心搜索架构SoundCloud：“SoundCloud使用ElasticSearch为1.8亿用户提供即时而精准的音乐搜索服务”

（3）百度：百度目前广泛使用ElasticSearch作为文本数据分析，采集百度所有服务器上的各类指标数据及用户自定义数据，通过对各种数据进行多维分析展示，辅助定位分析实例异常或业务层面异常。目前覆盖百度内部20多个业务线（包括casio、云分析、网盟、预测、文库、直达号、钱包、风控等），单集群最大100台机器，200个ES节点，每天导入30TB+数据

（4）新浪使用ES 分析处理32亿条实时日志

（5）阿里使用ES 构建挖财自己的日志采集和分析体系

### 1.3、同类产品

Solr、ElasticSearch、Hermes（腾讯）（实时检索分析）

* Solr、ES

1. 源自搜索引擎，侧重搜索与全文检索。

2. 数据规模从几百万到千万不等，数据量过亿的集群特别少。

有可能存在个别系统数据量过亿，但这并不是普遍现象（就像Oracle的表里的数据规模有可能超过Hive里一样，但需要小型机）。

* Hermes

1. 一个基于大索引技术的海量数据实时检索分析平台。侧重数据分析。

2. 数据规模从几亿到万亿不等。最小的表也是千万级别。

在 腾讯17 台TS5机器，就可以处理每天450亿的数据(每条数据1kb左右)，数据可以保存一个月之久。

* Solr、ES区别

全文检索、搜索、分析。基于lucene

1. Solr 利用 Zookeeper 进行分布式管理，而 Elasticsearch 自身带有分布式协调管理功能;
2. Solr 支持更多格式的数据，而 Elasticsearch 仅支持json文件格式；
3. Solr 官方提供的功能更多，而 Elasticsearch 本身更注重于核心功能，高级功能多有第三方插件提供；
4. Solr 在传统的搜索应用中表现好于 Elasticsearch，但在处理实时搜索应用时效率明显低于 Elasticsearch-----附近的人

Lucene是一个[开放源代码](https://baike.baidu.com/item/%E5%BC%80%E6%94%BE%E6%BA%90%E4%BB%A3%E7%A0%81/114160)的全文检索引擎工具包，但它不是一个完整的全文检索引擎，而是一个全文检索引擎的架构，提供了完整的查询引擎和索引引擎，部分[文本分析](https://baike.baidu.com/item/%E6%96%87%E6%9C%AC%E5%88%86%E6%9E%90/11046544)引擎

搜索引擎产品简介

## ElasticSearch

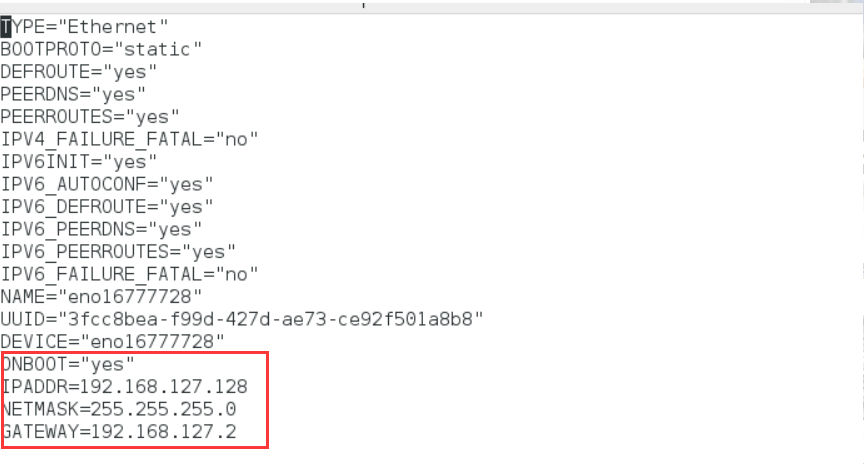
### 2.1、准备工作

安装Centos7、建议内存2G以上、安装java1.8环境

### 2.2、基本配置

* 设置IP地址

vi /etc/sysconfig/network-scripts/ifcfg-ens33



# 网络重置

service network restart

* 添加用户

[root@localhost ~]# adduser elk

[root@localhost ~]# passwd elk

以下授权步骤可省略

[root@localhost ~]# whereis sudoers

[root@localhost ~]# ls -l /etc/sudoers

[root@localhost ~]# vi /etc/sudoers

## Allow root to run any commands anywher

root ALL=(ALL) ALL

linuxidc ALL=(ALL) ALL #这个是新增的用户

[root@localhost ~]# chmod -v u-w /etc/sudoers

[root@localhost ~]# su elk

### 2.3、Java环境安装

* 解压安装包

[root@localhost jdk1.8]# tar -zxvf jdk-8u171-linux-x64.tar.gz

* 设置Java环境变量

[root@localhost jdk1.8.0\_171]# vi /etc/profile

在文件最后添加

export JAVA\_HOME=/home/elk1/jdk1.8/jdk1.8.0\_171

export JRE\_HOME=$JAVA\_HOME/jre

export CLASSPATH=.:$JAVA\_HOME/LIB:$JRE\_HOME/LIB:$CLASSPATH

export PATH=$JAVA\_HOME/bin:$JRE\_HOME/bin:$PATH

[root@localhost jdk1.8.0\_171]# source /etc/profile

[root@localhost jdk1.8.0\_171]# java -version

java version "1.8.0\_171"

Java(TM) SE Runtime Environment (build 1.8.0\_171-b11)

Java HotSpot(TM) 64-Bit Server VM (build 25.171-b11, mixed mode)

### 2.4、ElasticSerach单机安装

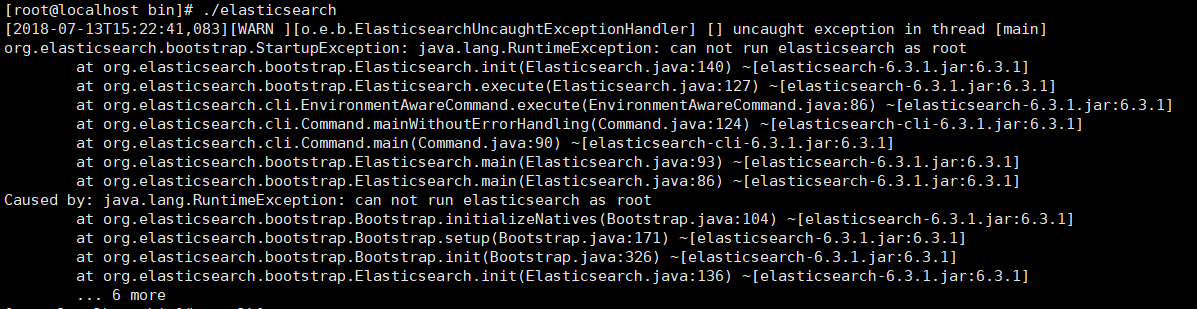
192.168.14.10 root elk

/home/elk/soft

[root@localhost elasticserach]# tar -zxvf elasticsearch-6.3.1.tar.gz

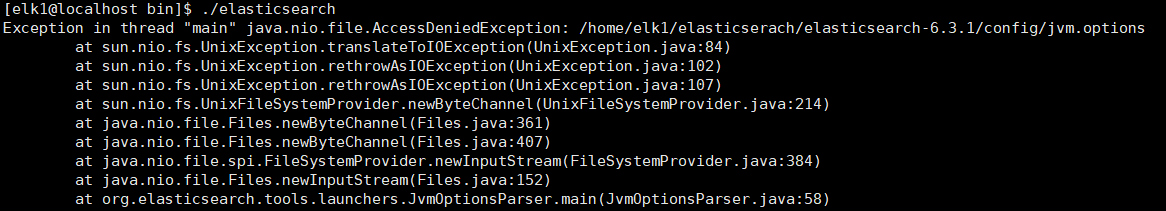
[root@localhost elasticserach]# cd elasticsearch-6.3.1/bin

[root@localhost bin]# ./elasticsearch



[root@localhost bin]# su elk1

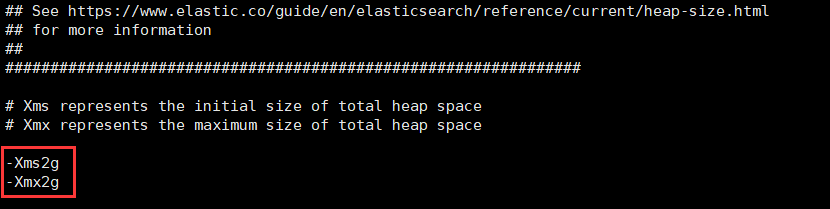
[elk1@localhost bin]$ ./elasticsearch



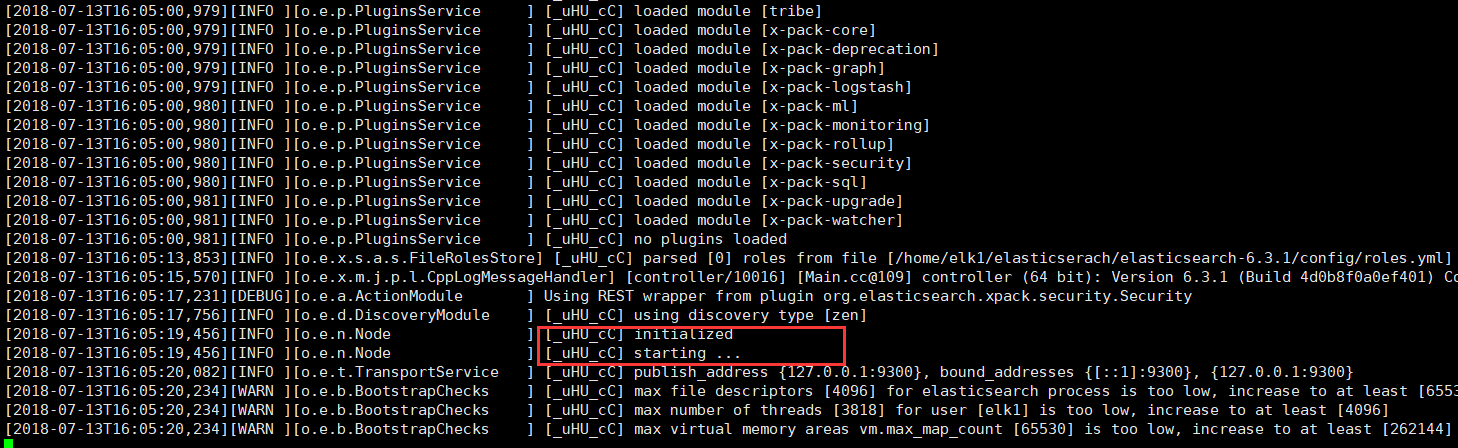
[root@localhost bin]# chown -R elk1:elk1 /home/elk1/elasticsearch

[elk1@localhost bin]$ ./elasticsearch

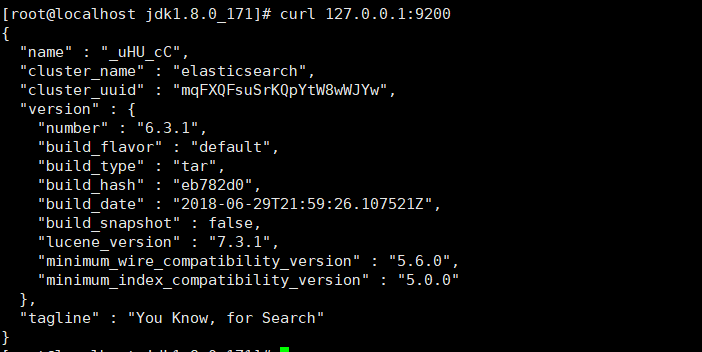
[elk1@localhost config]$ vi jvm.options



[elk1@localhost bin]$ ./elasticsearch



[root@localhost jdk1.8.0\_171]# curl 127.0.0.1:9200

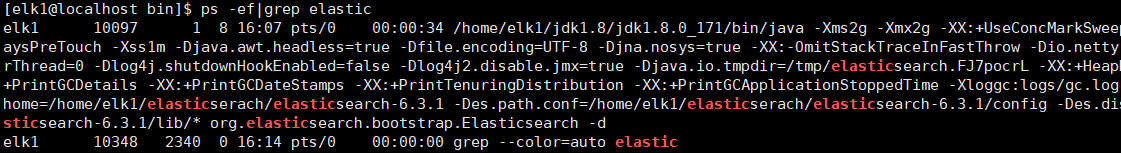


#后台启动

[elk1@localhost bin]$ ./elasticsearch -d

#关闭程序

[elk1@localhost bin]$ ps -ef|grep elastic

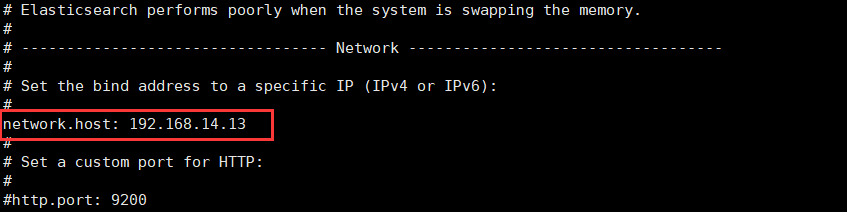


[elk1@localhost bin]$ kill 10097

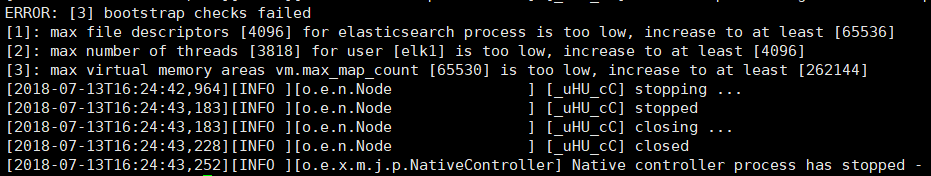
#设置浏览器访问

[root@localhost bin]systemctl stop firewalld

[root@localhost bin]vi config/elasticsearch.yml



安装问题：



1. [2]解决方案

[root@localhost bin]# vi /etc/security/limits.conf



nofile - 打开文件的最大数目

noproc - 进程的最大数目

soft 指的是当前系统生效的设置值

hard 表明系统中所能设定的最大值

\* hard nofile 65536

\* soft nofile 131072

\* hard nproc 4096

\* soft nproc 2048

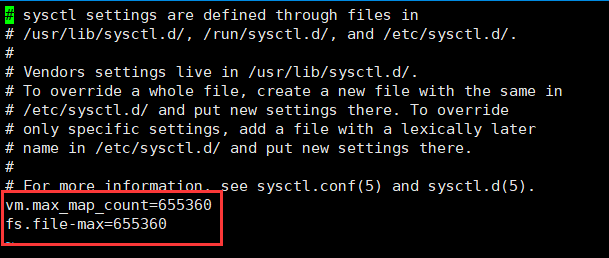
1. 解决方案

[root@localhost bin]# vi /etc/sysctl.conf

[root@localhost bin]# sysctl -p

vm.max\_map\_count=655360

fs.file-max=655360



vm.max\_map\_count=65530，因此缺省配置下，单个jvm能开启的最大线程数为其一半

file-max是设置 系统所有进程一共可以打开的文件数量

# 测试

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

浏览器访问：http://localhost:9200/?pretty

# 状态查看命令

语法：ip:post/\_cat/[args](?v|?format=json&pretty)

（?v表示显示字段说明,?format=json&pretty表示显示成json格式）

1. 查看所有索引

GET \_cat/indices?v

1. 查看es集群状态

GET \_cat/health?v

### 2.5、Elasticsearch的交互方式

1、基于HTTP协议，以JSON为数据交互格式的RESTful API

GET POST PUT DELETE HEAD

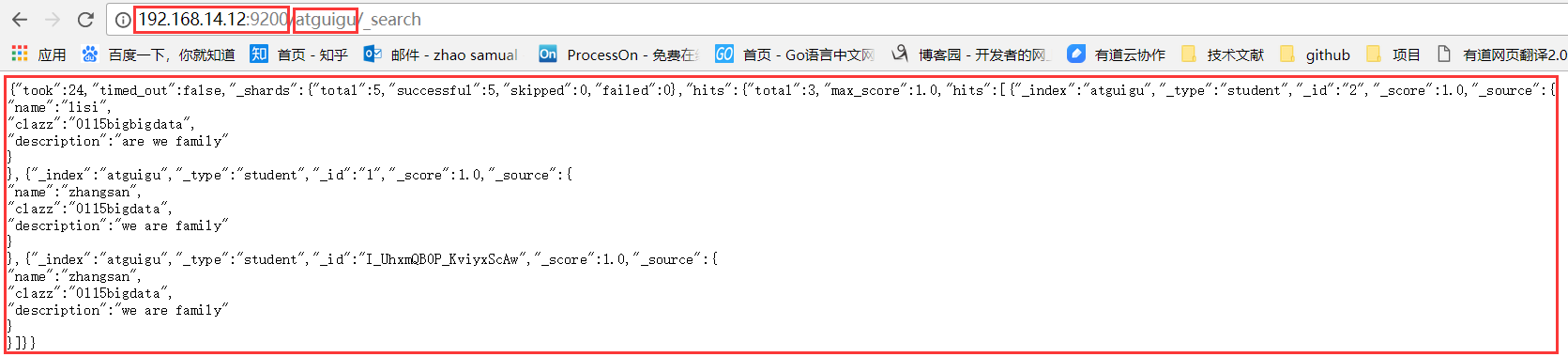
2、Elasticsearch官方提供了多种程序语言的客户端—java，Javascript，.NET，PHP，Perl，Python，以及 Ruby——还有很多由社区提供的客户端和插件

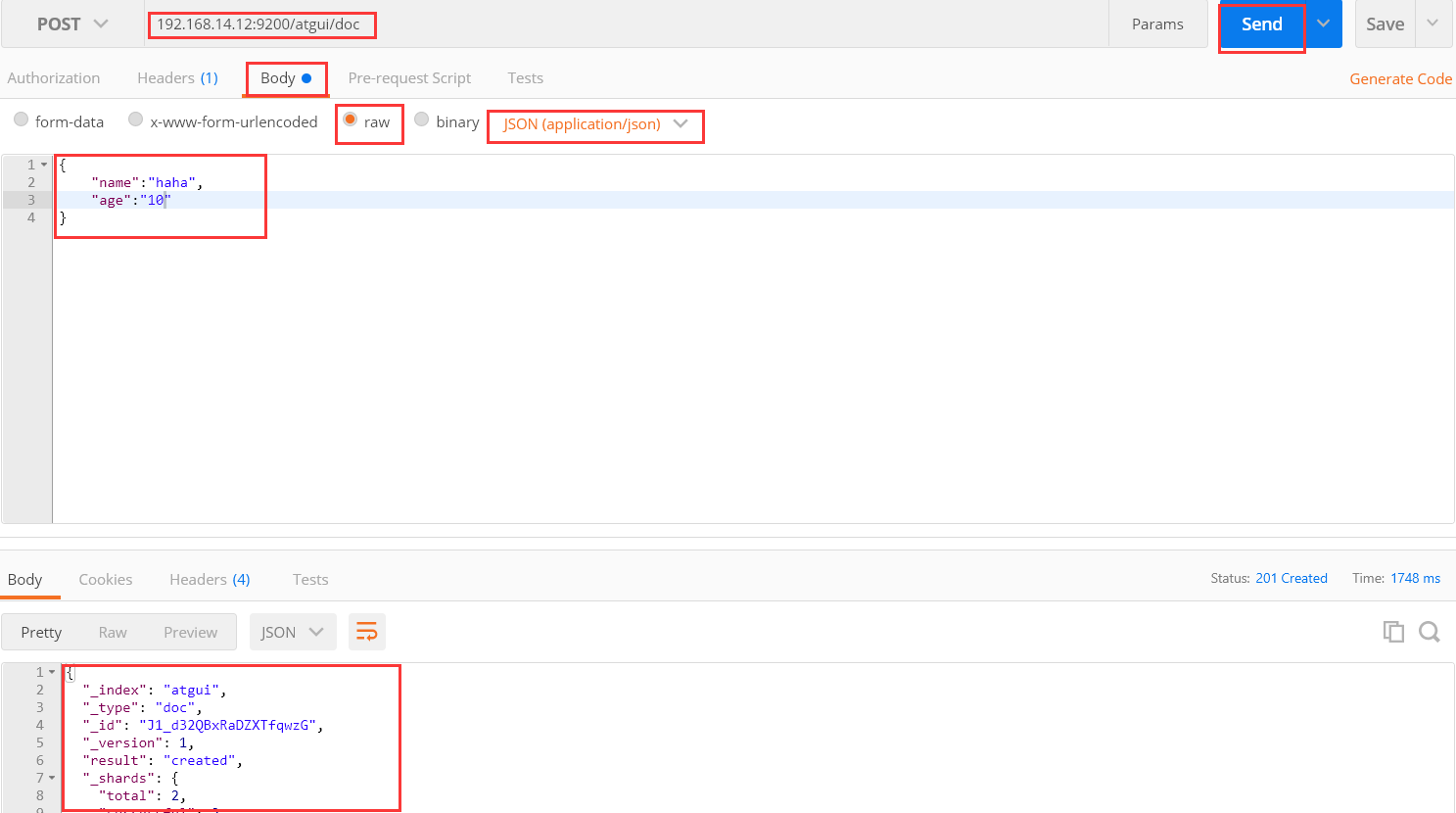
<https://www.elastic.co/guide/en/elasticsearch/client/java-rest/6.3/java-rest-high-getting-started-maven.html>

### 2.6、Elasticsearch操作工具

* REST访问ES方式（需要Http Method、URI）

1. 浏览器（postman）





1. Linux命令行

**请求：**

[root@localcurl -XPOST 'http://192.168.14.12:9200/atguig/doc' -i -H

"Content-Type:application/json" -d

'{"name":"haha","age":"10"}'

**响应：**

HTTP/1.1 201 Created

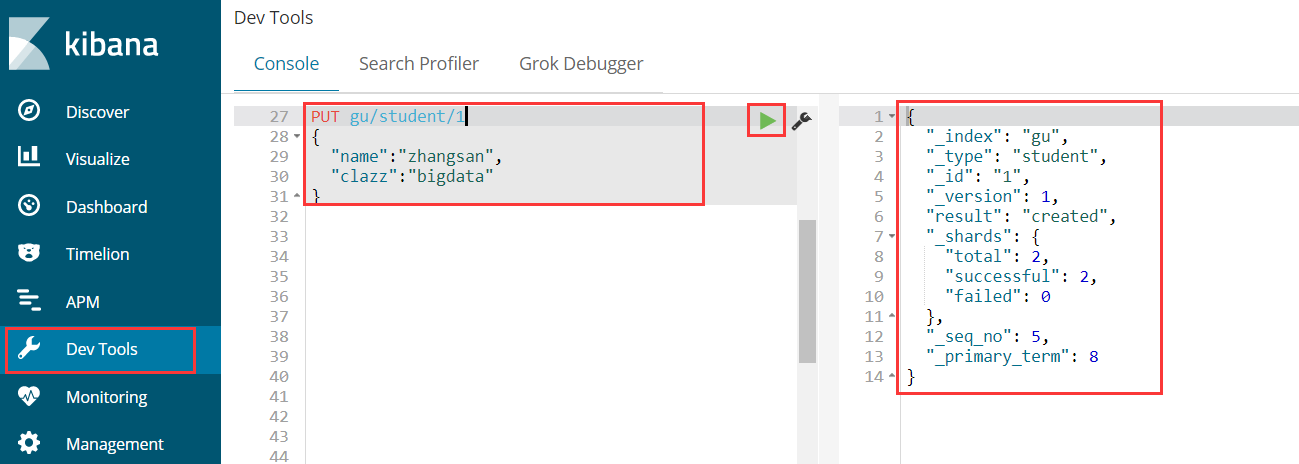
Location: /atguig/doc/KF\_t32QBxRaDZXTftAxg

content-type: application/json; charset=UTF-8

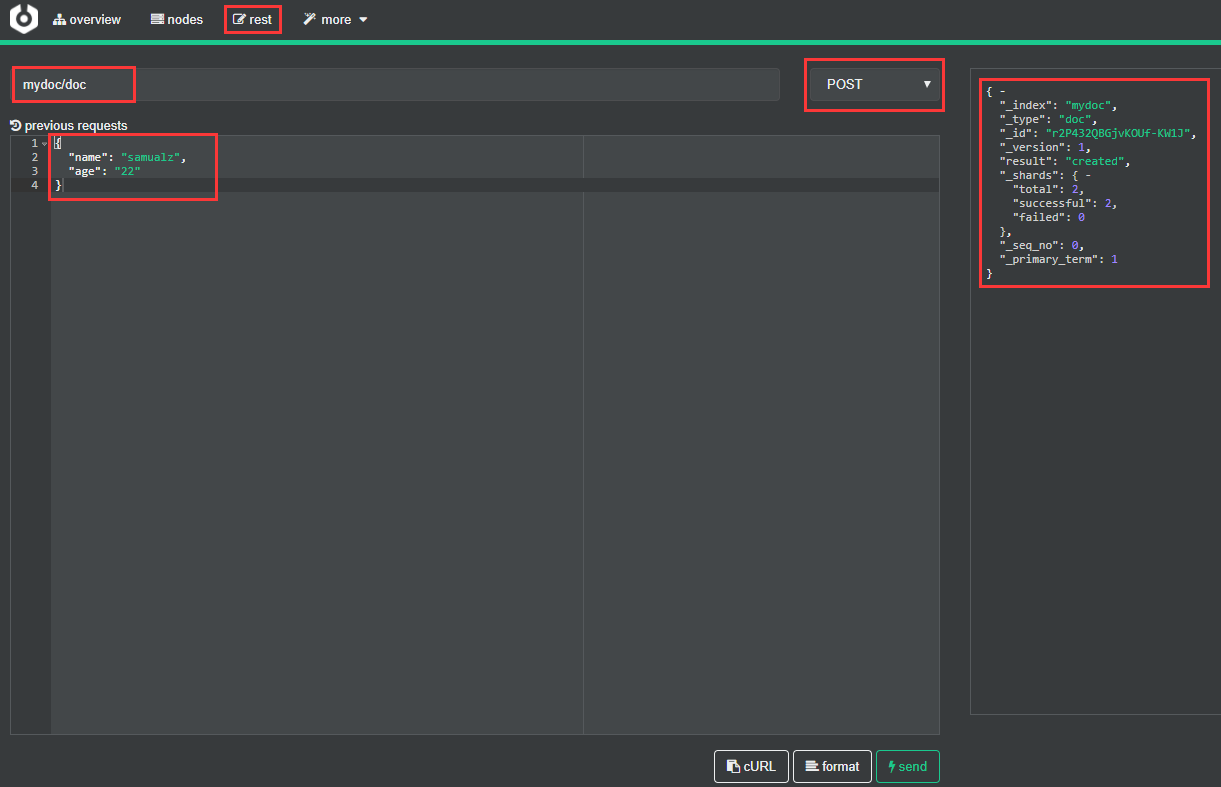
content-length: 172

{"\_index":"atguig","\_type":"doc","\_id":"KF\_t32QBxRaDZXTftAxg","\_version":1,"result": "created","\_shards":{"total":2,"successful":1,"failed":0},"\_seq\_no":0,"\_primary\_term": 1}

1. Kibana的Dev Tools



1. Cerebro插件



### 2.7、Elasticsearch数据存储方式

#### 2.7.1、Elasticsearch存储方式

（1）面向文档

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

（2）JSON

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

以下使用JSON文档来表示一个用户对象：

{

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

"first\_name": "John",

"last\_name": "Smith",

"info": {

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

"age": 25,

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

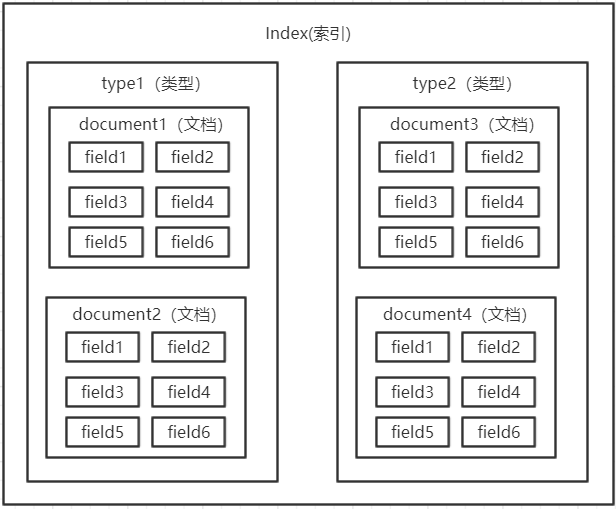
},

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

}

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

#### 2.7.2、Elasticsearch存储结构



Mysql数据与ES数据转化

1. 元数据

创建文档语句

PUT atguigu/doc

{

“name”:”zhangsan”,

“age”:10

}

\_index:文档所在索引名称

\_type:文档所在类型名称

\_id:文档唯一id

\_uid:组合id，由\_type和\_id组成（6.x后，\_type不再起作用，同\_id）

\_source:文档的原始Json数据，包括每个字段的内容

\_all:将所有字段内容整合起来，默认禁用（用于对所有字段内容检索）

（2）名词解释

* 索引 index

一个索引就是一个拥有几分相似特征的文档的集合。比如说，你可以有一个客户数据的索引，另一个产品目录的索引，还有一个订单数据的索引。一个索引由一个名字来标识（必须全部是小写字母的），并且当我们要对对应于这个索引中的文档进行索引、搜索、更新和删除的时候，都要使用到这个名字。在一个集群中，可以定义任意多的索引。

* 类型 type

Es6之后，一个index中只能有一个type

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

* 字段Field

相当于是数据表的字段，对文档数据根据不同属性进行的分类标识

* document

一个文档是一个可被索引的基础信息单元。比如，你可以拥有某一个客户的文档，某一个产品的一个文档，当然，也可以拥有某个订单的一个文档。文档以JSON（Javascript Object Notation）格式来表示，而JSON是一个到处存在的互联网数据交互格式。在一个index/type里面，你可以存储任意多的文档。注意，尽管一个文档，物理上存在于一个索引之中，文档必须被索引/赋予一个索引的type。

### 2.8、Elasticsearch检索

#### 2.8.1、检索文档

Mysql : select \* from user where id = 1

ES : GET /atguigu/doc/1

响应：

{

"\_index" : "megacorp",

"\_type" : "employee",

"\_id" : "1",

"\_version" : 1,

"found" : true,

"\_source" : {

"first\_name" : "John",

"last\_name" : "Smith",

"age" : 25,

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

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

}

}

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

#### 2.8.2、简单检索

Mysql : select \* from user

ES : GET /megacorp/employee/\_search

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

#### 2.8.3、全文检索

ES : GET /megacorp/employee/\_search?q=haha

查询出所有文档字段值为haha的文档

#### 2.8.4、搜索（模糊查询）

ES : GET /megacorp/employee/\_search?q=hello

查询出所有文档字段值分词后包含hello的文档

#### 2.8.5、聚合

Group by

Elasticsearch有一个功能叫做聚合(aggregations)，它允许你在数据上生成复杂的分析统计。它很像SQL中的GROUP BY但是功能更强大。

举个例子，让我们找到所有职员中最大的共同点（兴趣爱好）是什么：

GET /atguigu/doc/\_search

{

"aggs": {

"all\_interests": {

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

}

}

}

暂时先忽略语法只看查询结果：

{

...

"hits": { ... },

"aggregations": {

"all\_interests": {

"buckets": [

{

"key": "music",

"doc\_count": 2

},

{

"key": "forestry",

"doc\_count": 1

},

{

"key": "sports",

"doc\_count": 1

}

]

}

}

}

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

GET /atguigu/doc/\_search

{

"query": {

"match": {

"last\_name": "smith"

}

},

"aggs": {

"all\_interests": {

"terms": {

"field": "interests"

}

}

}

}

all\_interests聚合已经变成只包含和查询语句相匹配的文档了：

...

"all\_interests": {

"buckets": [

{

"key": "music",

"doc\_count": 2

},

{

"key": "sports",

"doc\_count": 1

}

]

}

PUT atguigu/\_mapping/doc/

{

"properties": {

"interests": {

"type": "text",

"fielddata": true

}

}

}

### 2.9、Elasticsearch搜索原理

https://www.cs.usfca.edu/~galles/visualization/Algorithms.html

#### 2.9.1、正排索引和倒排索引

* 正排索引

记录文档Id到文档内容、单词的关联关系

尚硅谷：1，3

|  |  |
| --- | --- |
| docid | content |
| 1 | 尚硅谷，是，最好的，培训，机构 |
| 2 | php是世界上最好的语言 |
| 3 | 机构尚硅谷是如何诞生的 |

* 倒排索引

记录单词到文档id的关联关系，包含：

单词词典（Term DicTionary）：记录所有文档的单词，一般比较大

倒排索引（Posting List）：记录单词倒排列表的关联信息

例如：尚硅谷

1、Term Dictionary

尚硅谷

2、Posting List

|  |  |  |  |
| --- | --- | --- | --- |
| DocId | TF | Position | Offset |
| 1 | 1 | 0 | <0,2> |
| 3 | 1 | 0 | <0,2> |

DocId：文档id，文档的原始信息

TF：单词频率，记录该词再文档中出现的次数，用于后续相关性算分

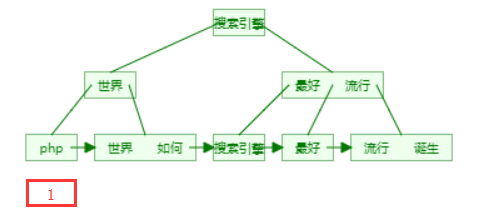
Position：位置，记录Field分词后，单词所在的位置，从0开始

Offset：偏移量，记录单词在文档中开始和结束位置，用于高亮显示等

1. 内存结构

<https://www.cs.usfca.edu/~galles/visualization/Algorithms.html>

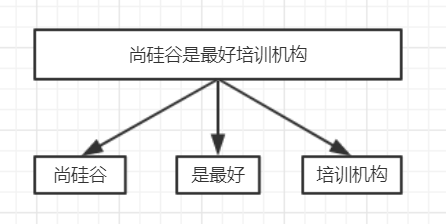
B+Tree



每个文档字段都有自己的倒排索引

#### 2.9.2、分词

分词是指将文本转换成一系列单词（term or token）的过程，也可以叫做文本分析，在es里面称为Analysis



* 分词机制

|  |  |  |
| --- | --- | --- |
| Character Filter | 对原始文本进行处理 | 例：去除html标签、特殊字符等 |
| Tokenizer | 将原始文本进行分词 | 例：培训机构-->培训，机构 |
| Token Filters | 分词后的关键字进行加工 | 例：转小写、删除语气词、近义词和同义词等 |

* 分词API

1. **直接指定测试（指定分词器）**

Request:

POST \_analyze

{

"analyzer": "standard",

"text":"hello 1111"

}

Response:

{

"tokens": [

{

"token": "hello", #分词

"start\_offset": 0, #开始偏移

"end\_offset": 5, #结束偏移

"type": "<ALPHANUM>", #单词类型

"position": 0 #位置

},

{

"token": "world",

"start\_offset": 6,

"end\_offset": 11,

"type": "<NUM>",

"position": 1

}

]

}

1. **针对索引的字段进行分词测试（利用该字段的分词器）**

Request：

POST atguigu/\_analyze

{

"field": "name",

"text":"hello world"

}

Response:

{

"tokens": [

{

"token": "hello",

"start\_offset": 0,

"end\_offset": 5,

"type": "<ALPHANUM>",

"position": 0

},

{

"token": "world",

"start\_offset": 6,

"end\_offset": 11,

"type": "<ALPHANUM>",

"position": 1

}

]

}

1. 自定义分词器

Request:

POST \_analyze

{

"tokenizer": "standard",

"filter": ["lowercase"],

"text":"Hello WORLD"

}

Response:

{

"tokens": [

{

"token": "hello",

"start\_offset": 0,

"end\_offset": 5,

"type": "<ALPHANUM>",

"position": 0

},

{

"token": "world",

"start\_offset": 6,

"end\_offset": 11,

"type": "<ALPHANUM>",

"position": 1

}

]

}

* Elasticsearch自带的分词器

|  |  |
| --- | --- |
| 分词器（Analyzer） | 特点 |
| Standard（es默认） | 支持多语言，按词切分并做小写处理 |
| Simple | 按照非字母切分，小写处理 |
| Whitespace | 按照空格来切分 |
| Stop | 去除语气助词，如the、an、的、这等 |
| Keyword | 不分词 |
| Pattern | 正则分词，默认\w+,即非字词符号做分割符 |
| Language | 常见语言的分词器（30+） |

* 中文分词

|  |  |  |  |
| --- | --- | --- | --- |
| 分词器名称 | 介绍 | 特点 | 地址 |
| IK | 实现中英文单词切分 | 自定义词库 | https://github.com/medcl/elasticsearch-analysis-ik |
| Jieba | python流行分词系统，支持分词和词性标注 | 支持繁体、自定义、并行分词 | http://github.com/sing1ee/elasticsearch-jieba-plugin |
| Hanlp | 由一系列模型于算法组成的java工具包 | 普及自然语言处理在生产环境中的应用 | https://github.com/hankcs/HanLP |
| THULAC | 清华大学中文词法分析工具包 | 具有中文分词和词性标注功能 | https://github.com/microbun/elasticsearch-thulac-plugin |

* Character Filters

在进行Tokenizer之前对原始文本进行处理，如增加、删除或替换字符等

|  |  |
| --- | --- |
| HTML Strip | 去除html标签和转换html实体 |
| Mapping | 字符串替换操作 |
| Pattern Replace | 正则匹配替换 |

注意：进行处理后，会影响后续tokenizer解析的position和offset

Request：

POST \_analyze

{

"tokenizer": "keyword",

"char\_filter": ["html\_strip"],

"text":"<div><h1>B<sup>+</sup>Trees</h1></div>"

}

Response:

{

"tokens": [

{

"token": """

B+Trees

""",

"start\_offset": 0,

"end\_offset": 38,

"type": "word",

"position": 0

}

]

}

* Token Filter

对输出的单词（term）进行增加、删除、修改等操作

|  |  |
| --- | --- |
| Lowercase | 将所有term转换为小写 |
| stop | 删除stop words |
| NGram | 和Edge NGram连词分割 |
| Synonym | 添加近义词的term |

Request:

POST \_analyze

{

"tokenizer": "standard",

"text":"a Hello World",

"filter": [

"stop",

"lowercase",

{

"type":"ngram",

"min\_gram":3,

"max\_gram":4

}

]

}

Response:

{

"tokens": [

{

"token": "hel",

"start\_offset": 2,

"end\_offset": 7,

"type": "<ALPHANUM>",

"position": 1

},

{

"token": "hell",

"start\_offset": 2,

"end\_offset": 7,

"type": "<ALPHANUM>",

"position": 1

},

{

"token": "ell",

"start\_offset": 2,

"end\_offset": 7,

"type": "<ALPHANUM>",

"position": 1

},

{

"token": "ello",

"start\_offset": 2,

"end\_offset": 7,

"type": "<ALPHANUM>",

"position": 1

},

{

"token": "llo",

"start\_offset": 2,

"end\_offset": 7,

"type": "<ALPHANUM>",

"position": 1

},

{

"token": "wor",

"start\_offset": 8,

"end\_offset": 13,

"type": "<ALPHANUM>",

"position": 2

},

{

"token": "worl",

"start\_offset": 8,

"end\_offset": 13,

"type": "<ALPHANUM>",

"position": 2

},

{

"token": "orl",

"start\_offset": 8,

"end\_offset": 13,

"type": "<ALPHANUM>",

"position": 2

},

{

"token": "orld",

"start\_offset": 8,

"end\_offset": 13,

"type": "<ALPHANUM>",

"position": 2

},

{

"token": "rld",

"start\_offset": 8,

"end\_offset": 13,

"type": "<ALPHANUM>",

"position": 2

}

]

}

* 自定义分词api

Request:

PUT my\_analyzer

{

"settings": {

"analysis": {

"analyzer": {

"my":{

"tokenizer":"punctuation",

"type":"custom",

"char\_filter":["emoticons"],

"filter":["lowercase","english\_stop"]

}

},

"tokenizer": {

"punctuation":{

"type":"pattern",

"pattern":"[.,!?]"

}

},

"char\_filter": {

"emoticons":{

"type":"mapping",

"mappings":[

":)=>\_happy\_",

":(=>\_sad\_"

]

}

},

"filter": {

"english\_stop":{

"type":"stop",

"stopwords":"\_english\_"

}

}

}

}

}

测试：

POST my\_analyzer/\_analyze

{

"analyzer": "my",

"text":"l'm a :) person,and you?"

}

{

"tokens": [

{

"token": "l'm a \_happy\_ person",

"start\_offset": 0,

"end\_offset": 15,

"type": "word",

"position": 0

},

{

"token": "and you",

"start\_offset": 16,

"end\_offset": 23,

"type": "word",

"position": 1

}

]

}

* 分词使用场景

1. 索引时分词：创建或更新文档时，会对相应得文档进行分词(指定字段分词)

PUT my\_test

{

“mappings”:{

“doc”:{

“properties”:{

“title”:{

“type”:”text”,

“analyzer”:”whitespace”

}

}

}

}

}

1. 查询时分词：查询时会对查询语句进行分词

POST my\_test/\_search

{

“query”:{

“match”:{

“message”:{

“query”:”hello”,

“analyzer”:”standard”

}

}

}

}

PUT my\_test

{

“mappings”:{

“doc”:{

“properties”:{

“title”:{

“type”:”text”,

“analyzer”:”whitespace”,

“search\_analyzer”:”standard” #查询指定分词器

}

}

}

}

}

一般不需要特别指定查询时分词器，直接使用索引时分词器即可，否则会出现无法匹配得情况，如果不需要分词将字段type设置成keyword，可以节省空间

#### 2.9.3、IK分词器

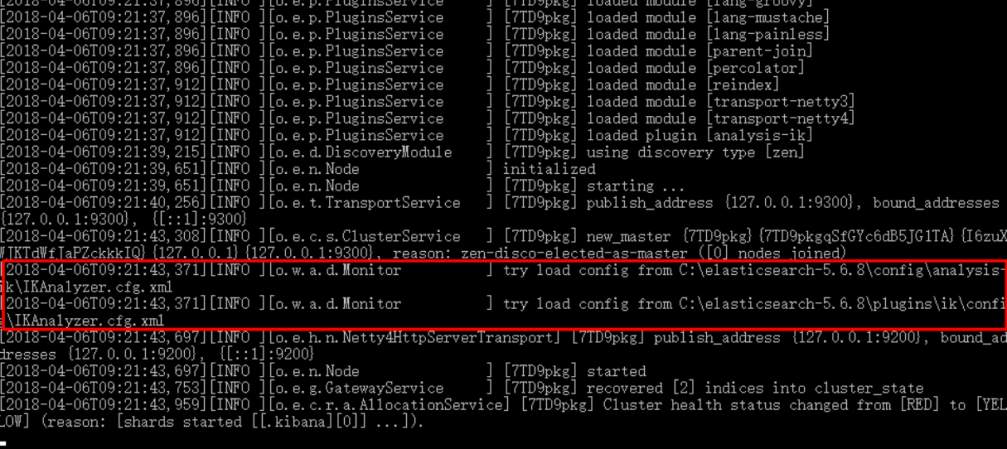
* IK分词器的安装

1. 下载地址：<https://github.com/medcl/elasticsearch-analysis-ik/releases>

下载与安装的ES相对应的版本

2）解压，将解压后的elasticsearch文件夹拷贝到elasticsearch-5.6.8\plugins下，并重命名文件夹为analysis-ik

3）重新启动ElasticSearch，即可加载IK分词器



* IK分词器测试

IK提供了两个分词算法ik\_smart 和 ik\_max\_word，其中 ik\_smart 为最少切分，ik\_max\_word为最细粒度划分

1. 最小切分：

在浏览器地址栏输入地址

<http://127.0.0.1:9200/_analyze?analyzer=ik_smart&pretty=true&text=我是程序员>

输出的结果为：

{

"tokens" : [

{

"token" : "我",

"start\_offset" : 0,

"end\_offset" : 1,

"type" : "CN\_CHAR",

"position" : 0

},

{

"token" : "是",

"start\_offset" : 1,

"end\_offset" : 2,

"type" : "CN\_CHAR",

"position" : 1

},

{

"token" : "程序员",

"start\_offset" : 2,

"end\_offset" : 5,

"type" : "CN\_WORD",

"position" : 2

}

]

}

2）最细切分：在浏览器地址栏输入地址

<http://127.0.0.1:9200/_analyze?analyzer=ik_max_word&pretty=true&text=我是程序员>

输出的结果为：

{

"tokens" : [

{

"token" : "我",

"start\_offset" : 0,

"end\_offset" : 1,

"type" : "CN\_CHAR",

"position" : 0

},

{

"token" : "是",

"start\_offset" : 1,

"end\_offset" : 2,

"type" : "CN\_CHAR",

"position" : 1

},

{

"token" : "程序员",

"start\_offset" : 2,

"end\_offset" : 5,

"type" : "CN\_WORD",

"position" : 2

},

{

"token" : "程序",

"start\_offset" : 2,

"end\_offset" : 4,

"type" : "CN\_WORD",

"position" : 3

},

{

"token" : "员",

"start\_offset" : 4,

"end\_offset" : 5,

"type" : "CN\_CHAR",

"position" : 4

}

]

}

### 2.10、Mapping

* 作用：

定义数据库中的表的结构的定义，通过mapping来控制索引存储数据的设置

1. 定义Index下的字段名（Field Name）
2. 定义字段的类型，比如数值型、字符串型、布尔型等
3. 定义倒排索引相关的配置，比如documentId、记录position、打分等

* 获取索引mapping

不进行配置时，自动创建的mapping

请求：

GET /atguigu/\_mapping

响应：

{

"atguigu": { #索引名称

"mappings": { #mapping设置

"student": { #type名称

"properties": { #字段属性

"clazz": {

"type": "text", #字段类型，字符串默认类型

"fields": { #子字段属性设置

"keyword": { #分词类型（不分词）

"type": "keyword",

"ignore\_above": 256

}

}

},

"description": {

"type": "text",

"fields": {

"keyword": {

"type": "keyword",

"ignore\_above": 256

}

}

},

"name": {

"type": "text",

"fields": {

"keyword": {

"type": "keyword",

"ignore\_above": 256

}

}

}

}

}

}

}

}

* 自定义mapping

请求：

PUT my\_index #索引名称

{

"mappings":{

"doc":{ #类型名称

"dynamic":false,

"properties":{

"title":{

"type":"text" #字段类型

},

"name":{

"type":"keyword"

},

"age":{

"type":"integer"

}

}

}

}

}

响应：

{

"acknowledged": true,

"shards\_acknowledged": true,

"index": "my\_index"

}

* Dynamic Mapping

es依靠json文档字段类型来实现自动识别字段类型，支持的类型

|  |  |
| --- | --- |
| JSON类型 | es类型 |
| null | 忽略 |
| boolean | boolean |
| 浮点类型 | float |
| 整数 | long |
| object | object |
| array | 由第一个非null值的类型决定 |
| string | 匹配为日期则设为data类型（默认开启）  匹配为数字的话设为float或long类型（默认关闭）  设为text类型，并附带keyword的子字段 |

* 注意：

mapping中的字段类型一旦设定后，禁止修改

原因：Lucene实现的倒排索引生成后不允许修改(提高效率)

如果要修改字段的类型，需要从新建立索引，然后做reindex操作

* dynamic设置

1. true：允许自动新增字段（默认的配置）
2. False：不允许自动新增字段，但是文档可以正常写入，无法对字段进行查询操作
3. strict：文档不能写入（如果写入会报错）

可以设置在type下，也可以设置在字段中（object类型的字段中）

例如：

put my\_index

{

“mappings”:{

“doc”:{

“dynamic”:false,

“properties”:{

“user”:{

“properties”:{

“name”:{

“type”:”text”

},”social\_networks”:{

“dynamic”:true,

“properties”:{}

}

}

}

}

}

}

}

* copy\_to

将该字段的值复制到目标字段，实现\_all的作用

不会出现在\_source中，只用来搜索

put my\_index

{

“mappings”:{

“doc”:{

“properties”:{

“frist\_name”:{

“type”:”text”,

“cope\_to”:”full\_name”

},”last\_name”:{

“type”:”text”,

“cope\_to”:“full\_name”

},”full\_name”:{

“type”:”text”

}

}

}

}

}

put my\_index/doc

{

“frist\_name”:”John”,

“last\_name”:”Smith”

}

GET my\_index/doc

{

“query”:{

“match”:{

“full\_name”:”John Smith”,

“operator”:”and”

}

}

}

* Index属性

Index属性，控制当前字段是否索引，默认为true，即记录索引，false不记录，即不可以搜索，比如：手机号、身份证号等敏感信息，不希望被检索

例如：

1、创建mapping

PUT my\_index

{

"mappings": {

"doc":{

"properties": {

"cookie":{

"type":"text",

"index": false

}

}

}

}

}

1. 创建文档

PUT my\_index/doc/1

{

"cookie":"123",

"name":"home"

}

1. 查询

GET my\_index/\_search

{

"query": {

"match": {

"cookie":"123"

}

}

}

#报错

GET my\_index/\_search

{

"query": {

"match": {

"name":"home"

}

}

}

#有结果

* Index\_options用于控制倒排索引记录的内容，有如下4中配置

docs：只记录docid

freqs：记录docid和term frequencies（词频）

position：记录docid、term frequencies、term position

Offsets：记录docid、term frequencies、term position、character offsets

text类型默认配置为position，其默认认为docs

记录的内容越多，占用的空间越大

### 2.11、数据类型

* 核心数据类型

字符串型：text、keyword

数值型：long、integer、short、byte、double、float、half\_float、scaled\_float

日期类型：date

布尔类型：boolean

二进制类型：binary

范围类型：integer\_range、float\_range、long\_range、double\_range、date\_range

* 复杂数据类型

数组类型：array

对象类型：object

嵌套类型：nested object

* 地理位置数据类型

geo\_point(点)、geo\_shape(形状)

* 专用类型

记录IP地址ip

实现自动补全completion

记录分词数：token\_count

记录字符串hash值母乳murmur3

* 多字段特性multi-fields

允许对同一个字段采用不同的配置，比如分词，例如对人名实现拼音搜索，只需要在人名中新增一个子字段为pinyin即可

1、创建mapping

PUT my\_index1

{

"mappings": {

"doc":{

"properties":{

"username":{

"type": "text",

"fields": {

"pinyin":{

"type": "text"

}

}

}

}

}

}

}

2、创建文档

PUT my\_index1/doc/1

{

"username":"haha heihei"

}

3、查询

GET my\_index1/\_search

{

"query": {

"match": {

"username.pinyin": "haha"

}

}

}

* Dynamic Mapping

es可以自动识别文档字段类型，从而降低用户使用成本

PUT /test\_index/doc/1

{

"username":"alfred",

"age":1

}

{

"test\_index": {

"mappings": {

"doc": {

"properties": {

"age": {

"type": "long"

},

"username": {

"type": "text",

"fields": {

"keyword": {

"type": "keyword",

"ignore\_above": 256

}

}

}

}

}

}

}

}

age自动识别为long类型，username识别为text类型

PUT test\_index/doc/1

{

"username":"samualz",

"age":14,

"birth":"1991-12-15",

"year":18,

"tags":["boy","fashion"],

"money":"100.1"

}

{

"test\_index": {

"mappings": {

"doc": {

"properties": {

"age": {

"type": "long"

},

"birth": {

"type": "date"

},

"money": {

"type": "text",

"fields": {

"keyword": {

"type": "keyword",

"ignore\_above": 256

}

}

},

"tags": {

"type": "text",

"fields": {

"keyword": {

"type": "keyword",

"ignore\_above": 256

}

}

},

"username": {

"type": "text",

"fields": {

"keyword": {

"type": "keyword",

"ignore\_above": 256

}

}

},

"year": {

"type": "long"

}

}

}

}

}

}

日期的自动识别可以自行配置日期格式，以满足各种需求

1. 自定义日期识别格式

PUT my\_index

{

"mappings":{

"doc":{

"dynamic\_date\_formats": ["yyyy-MM-dd","yyyy/MM/dd"]

}

}

}

2、关闭日期自动识别

PUT my\_index

{

"mappings": {

"doc": {

"date\_detection": false

}

}

}

字符串是数字时，默认不会自动识别为整形，因为字符串中出现数字时完全合理的

Numeric\_datection可以开启字符串中数字的自动识别

PUT my\_index

{

"mappings":{

"doc":{

"numeric\_datection": true

}

}

}

### 2.12、文档操作

CRUD

* 创建文档

1、索引一个文档

文档通过index API被索引——使数据可以被存储和搜索。但是首先我们需要决定文档所在。正如我们讨论的，文档通过其\_index、\_type、\_id唯一确定。们可以自己提供一个\_id，或者也使用index API 为我们生成一个。

PUT {index}/{type}/{id}

{

“”:””

}

2、使用自己的ID

如果你的文档有自然的标识符（例如user\_account字段或者其他值表示文档），你就可以提供自己的\_id，使用这种形式的index API：

PUT /{index}/{type}/{id}

{

"field": "value",

...

}

例如我们的索引叫做“website”，类型叫做“blog”，我们选择的ID是“123”，那么这个索引请求就像这样：

PUT /website/blog/123

{

"title": "My first blog entry",

"text": "Just trying this out...",

"date": "2014/01/01"

}

Elasticsearch的响应：

{

"\_index": "website",

"\_type": "blog",

"\_id": "123",

"\_version": 1,

"created": true

}

响应指出请求的索引已经被成功创建，这个索引中包含\_index、\_type和\_id元数据，以及一个新元素：\_version。

Elasticsearch中每个文档都有版本号，每当文档变化（包括删除）都会使\_version增加。\_version确保你程序的一部分不会覆盖掉另一部分所做的更改。

3、自增ID

如果我们的数据没有自然ID，我们可以让Elasticsearch自动为我们生成。请求结构发生了变化：PUT方法——“在这个URL中存储文档”变成了POST方法——"在这个类型下存储文档"。（译者注：原来是把文档存储到某个ID对应的空间，现在是把这个文档添加到某个\_type下）。

URL现在只包含\_index和\_type两个字段：

POST /website/blog/

{

"title": "My second blog entry",

"text": "Still trying this out...",

"date": "2014/01/01"

}

响应内容与刚才类似，只有\_id字段变成了自动生成的值：

{

"\_index": "website",

"\_type": "blog",

"\_id": "wM0OSFhDQXGZAWDf0-drSA",

"\_version": 1,

"created": true

}

自动生成的ID有22个字符长，URL-safe, Base64-encoded string universally unique identifiers, 或者叫 [UUIDs](http://en.wikipedia.org/wiki/Uuid)。

* 获取文档

1、检索文档

想要从Elasticsearch中获取文档，我们使用同样的\_index、\_type、\_id，但是HTTP方法改为GET：

GET /website/blog/123?pretty

响应包含了现在熟悉的元数据节点，增加了\_source字段，它包含了在创建索引时我们发送给Elasticsearch的原始文档。

{

"\_index" : "website",

"\_type" : "blog",

"\_id" : "123",

"\_version" : 1,

"found" : true,

"\_source" : {

"title": "My first blog entry",

"text": "Just trying this out...",

"date": "2014/01/01"

}

}

2、pretty

在任意的查询字符串中增加pretty参数，类似于上面的例子。会让Elasticsearch美化输出(pretty-print)JSON响应以便更加容易阅读。\_source字段不会被美化，它的样子与我们输入的一致。

GET请求返回的响应内容包括{"found": true}。这意味着文档已经找到。如果我们请求一个不存在的文档，依旧会得到一个JSON，不过found值变成了false。

此外，HTTP响应状态码也会变成'404 Not Found'代替'200 OK'。我们可以在curl后加-i参数得到响应头：

curl -i -XGET http://localhost:9200/website/blog/124?pretty

现在响应类似于这样：

HTTP/1.1 404 Not Found

Content-Type: application/json; charset=UTF-8

Content-Length: 83

{

"\_index" : "website",

"\_type" : "blog",

"\_id" : "124",

"found" : false

}

3、检索文档的一部分

通常，GET请求将返回文档的全部，存储在\_source参数中。但是可能你感兴趣的字段只是title。请求个别字段可以使用\_source参数。多个字段可以使用逗号分隔：

GET /website/blog/123?\_source=title,text

\_source字段现在只包含我们请求的字段，而且过滤了date字段：

{

"\_index" : "website",

"\_type" : "blog",

"\_id" : "123",

"\_version" : 1,

"exists" : true,

"\_source" : {

"title": "My first blog entry" ,

"text": "Just trying this out..."

}

}

或者你只想得到\_source字段而不要其他的元数据，你可以这样请求：

GET /website/blog/123/\_source

它仅仅返回:

{

"title": "My first blog entry",

"text": "Just trying this out...",

"date": "2014/01/01"

}

* 更新

POST /website/blog/123

{

"title": "My first blog entry",

"text": "I am starting to get the hang of this...",

"date": "2014/01/02"

}

在响应中，我们可以看到Elasticsearch把\_version增加了。

{

"\_index" : "website",

"\_type" : "blog",

"\_id" : "123",

"\_version" : 2,

"created": false <1>

}

* 删除文档

删除文档的语法模式与之前基本一致，只不过要使用DELETE方法：

DELETE /website/blog/123

* 局部更新

POST /website/blog/1/\_update

{

"doc" : {

"tags" : [ "testing" ],

"views": 0

}

}

如果请求成功，我们将看到类似index请求的响应结果：

{

"\_index" : "website",

"\_id" : "1",

"\_type" : "blog",

"\_version" : 3

}

检索文档文档显示被更新的\_source字段：

{

"\_index": "website",

"\_type": "blog",

"\_id": "1",

"\_version": 3,

"found": true,

"\_source": {

"title": "My first blog entry",

"text": "Starting to get the hang of this...",

"tags": [ "testing" ], <1>

"views": 0 <1>

}

}

* 批量插入

每个json之间不能有换行\n

POST test\_search\_index/doc/\_bulk

{

"index":{

"\_id":1

}

}

{

"username":"alfred way",

"job":"java engineer",

"age":18,

"birth":"1991-12-15",

"isMarried":false

}

{

"index":{

"\_id":2

}

}

{

"username":"alfred",

"job":"java senior engineer and java specialist",

"age":28,

"birth":"1980-05-07",

"isMarried":true

}

{

"index":{

"\_id":3

}

}

{

"username":"lee",

"job":"java and ruby engineer",

"age":22,

"birth":"1985-08-07",

"isMarried":false

}

{

"index":{

"\_id":4

}

}

{

"username":"lee junior way",

"job":"ruby engineer",

"age":23,

"birth":"1986-08-07",

"isMarried":false

}

* 检索多个文档

像Elasticsearch一样，检索多个文档依旧非常快。合并多个请求可以避免每个请求单独的网络开销。如果你需要从Elasticsearch中检索多个文档，相对于一个一个的检索，更快的方式是在一个请求中使用multi-get或者mget API。

mget API参数是一个docs数组，数组的每个节点定义一个文档的\_index、\_type、\_id元数据。如果你只想检索一个或几个确定的字段，也可以定义一个\_source参数：

POST /\_mget

{

"docs" : [

{

"\_index" : "website",

"\_type" : "blog",

"\_id" : 2

},

{

"\_index" : "website",

"\_type" : "pageviews",

"\_id" : 1,

"\_source": "views"

}

]

}

响应体也包含一个docs数组，每个文档还包含一个响应，它们按照请求定义的顺序排列。每个这样的响应与单独使用get request响应体相同：

{

"docs" : [

{

"\_index" : "website",

"\_id" : "2",

"\_type" : "blog",

"found" : true,

"\_source" : {

"text" : "This is a piece of cake...",

"title" : "My first external blog entry"

},

"\_version" : 10

},

{

"\_index" : "website",

"\_id" : "1",

"\_type" : "pageviews",

"found" : true,

"\_version" : 2,

"\_source" : {

"views" : 2

}

}

]

}

如果你想检索的文档在同一个\_index中（甚至在同一个\_type中），你就可以在URL中定义一个默认的/\_index或者/\_index/\_type。

你可以通过简单的ids数组来代替完整的docs数组：

POST /website/blog/\_mget

{

"ids" : [ "2", "1" ]

}

注意到我们请求的第二个文档并不存在。我们定义了类型为blog，但是ID为1的文档类型为pageviews。这个不存在的文档会在响应体中被告知。

{

"docs" : [

{

"\_index" : "website",

"\_type" : "blog",

"\_id" : "2",

"\_version" : 10,

"found" : true,

"\_source" : {

"title": "My first external blog entry",

"text": "This is a piece of cake..."

}

},

{

"\_index" : "website",

"\_type" : "blog",

"\_id" : "1",

"found" : false <1>

}

]

}

### 2.13、Search API(URI)

GET /\_search #查询所有索引文档

GET /my\_index/\_search #查询指定索引文档

GET /my\_index1,my\_index2/\_search #多索引查询

GET /my\_\*/\_search

2019-03-xxx

2019-04-vvv

* URI查询方式（查询有限制，很多配置不能实现）

GET /my\_index/\_search?q=user:alfred #指定字段查询

GET /my\_index/\_search?q=keyword&df=user&sort=age:asc&from=4&size=10&timeout=1s

q : 指定查询的语句，例如q=aa或q=user:aa

df:q中不指定字段默认查询的字段，如果不指定，es会查询所有字段

Sort：排序，asc升序，desc降序

timeout：指定超时时间，默认不超时

from，size：用于分页

* term与phrase

term相当于单词查询，phrase相当于词语查询

term：Alfred way等效于alfred or way

phrase：”Alfred way” 词语查询，要求先后顺序

* + 泛查询

Alfred等效于在所有字段去匹配该term(不指定字段查询)

* 指定字段

name:alfred

* + Group分组设定（），使用括号指定匹配的规则

（quick OR brown）AND fox：通过括号指定匹配的优先级

status:(active OR pending) title:(full text search)：把关键词当成一个整体

* 查询案例及详解

1. 批量创建文档

POST test\_search\_index/doc/\_bulk

{

"index":{

"\_id":1

}

}

{

"username":"alfred way",

"job":"java engineer",

"age":18,

"birth":"1991-12-15",

"isMarried":false

}

{

"index":{

"\_id":2

}

}

{

"username":"alfred",

"job":"java senior engineer and java specialist",

"age":28,

"birth":"1980-05-07",

"isMarried":true

}

{

"index":{

"\_id":3

}

}

{

"username":"lee",

"job":"java and ruby engineer",

"age":22,

"birth":"1985-08-07",

"isMarried":false

}

{

"index":{

"\_id":4

}

}

{

"username":"lee junior way",

"job":"ruby engineer",

"age":23,

"birth":"1986-08-07",

"isMarried":false

}

1. 泛查询

GET test\_search\_index/\_search?q=alfred

1. 查询语句执行计划查看

GET test\_search\_index/\_search?q=alfred

{

"profile":true

}

1. term查询

GET test\_search\_index/\_search?q=username:alfred way #alfred OR way

1. phrase查询

GET test\_search\_index/\_search?q=username:"alfred way"

6、group查询

GET test\_search\_index/\_search?q=username:(alfred way)

7、布尔操作符

（1）AND(&&),OR(||),NOT(!)

例如：name:(tom NOT lee)

#表示name字段中可以包含tom但一定不包含lee

1. +、-分别对应must和must\_not

例如：name:(tom +lee -alfred)

#表示name字段中，一定包含lee，一定不包含alfred，可以包含tom

注意：+在url中会被解析成空格，要使用encode后的结果才可以，为%2B

GET test\_search\_index/\_search?q=username:(alfred %2Bway)

* 范围查询，支持数值和日期

1. 区间：闭区间：[]，开区间:{}

age:[1 TO 10] #1<=age<=10

age:[1 TO 10} #1<=age<10

age:[1 TO ] #1<=age

age:[\* TO 10] #age<=10

1. 算术符号写法

age:>=1

age:(>=1&&<=10)或者age:(+>=1 +<=10)

* 通配符查询

?:1个字符

\*:0或多个字符

例如：name:t?m

name:tom\*

name:t\*m

注意：通配符匹配执行效率低，且占用较多内存，不建议使用，如无特殊要求，不要讲?/\*放在最前面

* 正则表达式

name:/[mb]oat/

* 模糊匹配fuzzy query

name:roam~1

匹配与roam差1个character的词，比如foam、roams等

* 近似度查询proximity search

“fox quick”~5

以term为单位进行差异比较，比如”quick fox” “quick brown fox”

### 2.14、Search API(Request Body Search)

* Match Query

对字段作全文检索，最基本和常用的查询类型

GET test\_search\_index/\_search

{

"profile":true, # 显示执行计划

"query":{

"match": {

"username": "alfred way"

}

}

}

通过operator参数可以控制单词间的匹配关系，可选项为or和and

GET test\_search\_index/\_search

{

"query":{

"match": {

"username": {

"query":"alfred way",

"operator":"and"

}

}

}

}

## Elasticsearch集群

### 3.1、ElasticSerach集群安装

* 修改配置文件elasticserach.yml

vim /elasticsearch.yml

cluster.name: aubin-cluster #必须相同

# 集群名称（不能重复）

node.name: els1（必须不同）

# 节点名称，仅仅是描述名称，用于在日志中区分（自定义）

#指定了该节点可能成为 master 节点，还可以是数据节点  
 node.master: true  
 node.data: true

path.data: /opt/data

# 数据的默认存放路径（自定义）

path.logs: /opt/logs

# 日志的默认存放路径

network.host: 192.168.0.1

# 当前节点的IP地址

http.port: 9200

# 对外提供服务的端口

transport.tcp.port: 9300

#9300为集群服务的端口

discovery.zen.ping.unicast.hosts: ["172.18.68.11", "172.18.68.12","172.18.68.13"]

# 集群个节点IP地址，也可以使用域名，需要各节点能够解析

discovery.zen.minimum\_master\_nodes: 2

# 为了避免脑裂，集群节点数最少为 半数+1

注意：清空data和logs数据

192.168.14.12:9200/\_cat/nodes?v

### 3.2、安装head插件

* 下载head插件

wget [https://github.com/mobz/elasticsearch-head/archive/elasticsearch-head-master.zip](https://github.com/mobz/elasticsearch-head/archive/master.zip)

也可以用git下载，前提yum install git

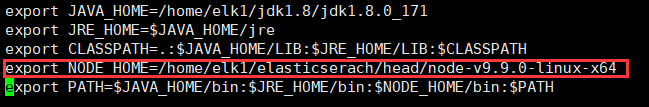
unzip elasticsearch-head-master.zip

* 安装node.js

wget <https://npm.taobao.org/mirrors/node/latest-v4.x/node-v4.4.7-linux-x64.tar.gz>

tar -zxvf node-v9.9.0-linux-x64.tar.gz

* 添加node.js到环境变量



source /etc/profile

* 测试

node -v

npm -v

* 安装grunt（grunt是一个很方便的构建工具，可以进行打包压缩、测试、执行等等的工作）

进入到elasticsearch-head-master

npm install -g grunt-cli

npm install

(npm install -g cnpm --registry=https://registry.npm.taobao.org)

* 修改Elasticsearch配置文件

编辑elasticsearch-6.3.1/config/elasticsearch.yml,加入以下内容：

http.cors.enabled: true

http.cors.allow-origin: "\*"

* 修改Gruntfile.js（注意’，’）

打开elasticsearch-head-master/Gruntfile.js，找到下面connect属性，新增hostname:’\*’:

connect: {

server: {

options: {

hostname: '\*',

port: 9100,

base: '.',

keepalive: true

}

}

}

* 启动elasticsearch-head

进入elasticsearch-head目录，执行命令：grunt server

* 后台启动elasticsearch-head

nohup grunt server &exit

* 关闭head插件

ps -aux|grep head

kill 进程号

### 3.3、集群简介

一个节点(node)就是一个Elasticsearch实例，而一个集群(cluster)由一个或多个节点组成，它们具有相同的cluster.name，它们协同工作，分享数据和负载。

当加入新的节点或者删除一个节点时，集群就会感知到并平衡数据。

#### 3.3.1、集群节点

1. 集群中一个节点会被选举为主节点(master)
2. 临时管理集群级别的一些变更，例如新建或删除索引、增加或移除节点等。
3. 主节点不参与文档级别的变更或搜索，这意味着在流量增长的时候，该主节点不会成为集群的瓶颈。
4. 任何节点都可以成为主节点。
5. 用户，我们能够与集群中的任何节点通信，包括主节点。
6. 每一个节点都知道文档存在于哪个节点上，它们可以转发请求到相应的节点上。
7. 我们访问的节点负责收集各节点返回的数据，最后一起返回给客户端。这一切都由Elasticsearch处理。

#### 3.3.2、集群健康

在Elasticsearch集群中可以监控统计很多信息，但是只有一个是最重要的：集群健康(cluster health)。集群健康有三种状态：green、yellow或red。

在一个没有索引的空集群中运行如上查询，将返回这些信息：

GET /\_cluster/health

{

"cluster\_name": "elasticsearch",

"status": "green",

"timed\_out": false,

"number\_of\_nodes": 1,

"number\_of\_data\_nodes": 1,

"active\_primary\_shards": 0,

"active\_shards": 0,

"relocating\_shards": 0,

"initializing\_shards": 0,

"unassigned\_shards": 0

}

status字段提供一个综合的指标来表示集群的的服务状况。三种颜色各自的含义：

| 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 |

#### 3.3.3、集群分片

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)”.

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，是一个Lucene实例，并且它本身就是一个完整的搜索引擎。我们的文档存储在分片中，并且在分片中被索引，但是我们的应用程序不会直接与它们通信，取而代之的是，直接与索引通信。

分片是Elasticsearch在集群中分发数据的关键。把分片想象成数据的容器。文档存储在分片中，然后分片分配到你集群中的节点上。当你的集群扩容或缩小，Elasticsearch将会自动在你的节点间迁移分片，以使集群保持平衡。

1. 主分片

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。

理论上主分片能存储的数据大小是没有限制的，限制取决于你实际的使用情况。分片的最大容量完全取决于你的使用状况：硬件存储的大小、文档的大小和复杂度、如何索引和查询你的文档，以及你期望的响应时间。

1. 副分片

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。

创建分片：

PUT /blogs

{

"settings" : {

"number\_of\_shards" : 3,

"number\_of\_replicas" : 1

}

}

增加副分片：

PUT /blogs/\_settings

{

"number\_of\_replicas" : 2

}

集群的健康状态yellow表示所有的主分片(primary shards)启动并且正常运行了——集群已经可以正常处理任何请求——但是复制分片(replica shards)还没有全部可用。事实上所有的三个复制分片现在都是unassigned状态——它们还未被分配给节点。在同一个节点上保存相同的数据副本是没有必要的，如果这个节点故障了，那所有的数据副本也会丢失。

#### 3.3.4、故障转移

在单一节点上运行意味着有单点故障的风险——没有数据备份。幸运的是，要防止单点故障，我们唯一需要做的就是启动另一个节点。

第二个节点已经加入集群，三个复制分片(replica shards)也已经被分配了——分别对应三个主分片，这意味着在丢失任意一个节点的情况下依旧可以保证数据的完整性。

文档的索引将首先被存储在主分片中，然后并发复制到对应的复制节点上。这可以确保我们的数据在主节点和复制节点上都可以被检索。

### 3.4、集群操作原理

#### 3.4.1、路由

当你索引一个文档，它被存储在单独一个主分片上。Elasticsearch是如何知道文档属于哪个分片的呢？当你创建一个新文档，它是如何知道是应该存储在分片1还是分片2上的呢？

进程不能是随机的，因为我们将来要检索文档。

算法决定：

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。

为什么主分片的数量只能在创建索引时定义且不能修改？

如果主分片的数量在未来改变了，所有先前的路由值就失效了，文档也就永远找不到了。

所有的文档API（get、index、delete、bulk、update、mget）都接收一个routing参数，它用来自定义文档到分片的映射。自定义路由值可以确保所有相关文档——例如属于同一个人的文档——被保存在同一分片上。

#### 3.4.2、操作数据节点工作流程

每个节点都有能力处理任意请求。每个节点都知道任意文档所在的节点，所以也可以将请求转发到需要的节点。

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。



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。

**replication**

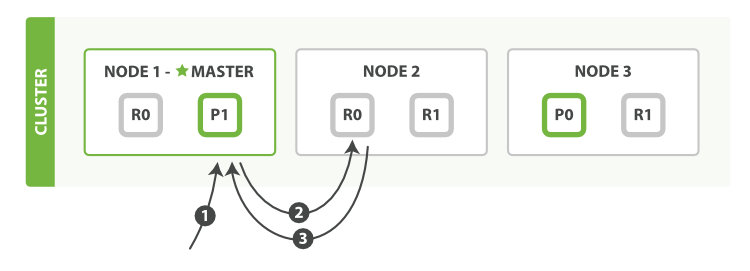
复制默认的值是sync。这将导致主分片得到复制分片的成功响应后才返回。

如果你设置replication为async，请求在主分片上被执行后就会返回给客户端。它依旧会转发请求给复制节点，但你将不知道复制节点成功与否。

上面的这个选项不建议使用。默认的sync复制允许Elasticsearch强制反馈传输。async复制可能会因为在不等待其它分片就绪的情况下发送过多的请求而使Elasticsearch过载。

#### 3.4.3、检索流程

文档能够从主分片或任意一个复制分片被检索。



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。

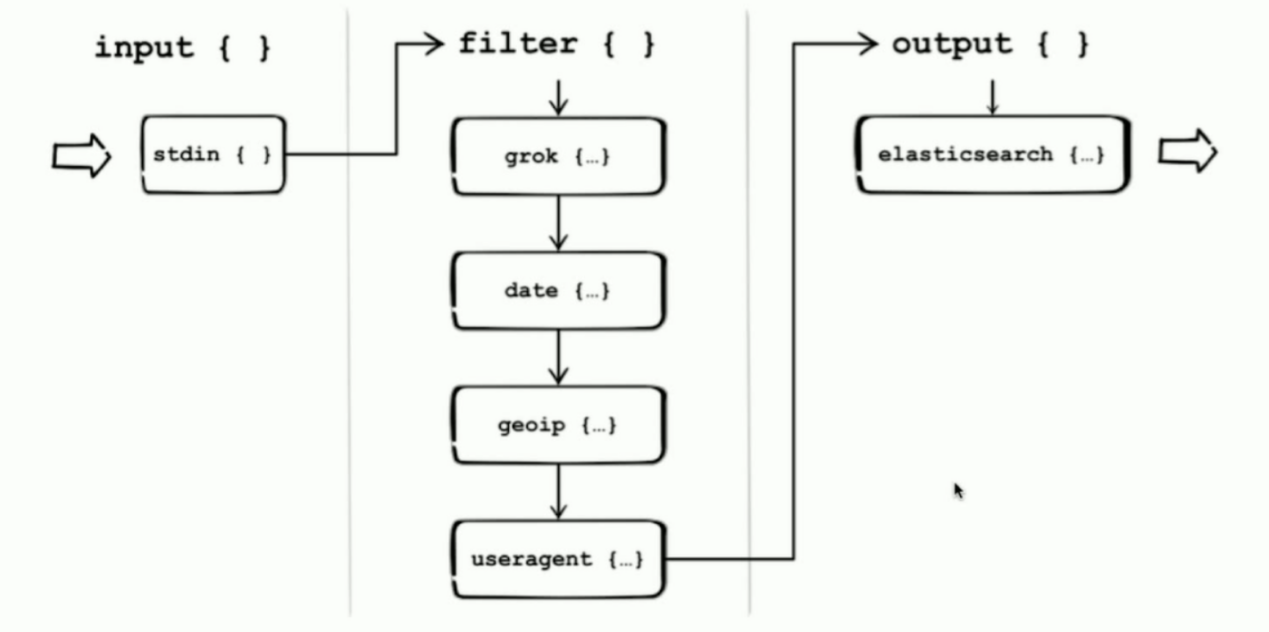
对于读请求，为了平衡负载，请求节点会为每个请求选择不同的分片——它会循环所有分片副本。

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。

## Logstash

### 4.1、logstsh架构

搜集---》过滤---》处理

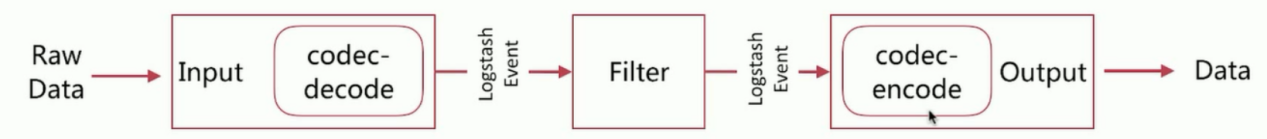


Grok:匹配需要收集的字段信息

Date：处理日期类型

Geoip:添加地理位置信息

Useragent:提取请求用户信息



### 4.2、logstash安装

[root@localhost logstash]# tar -zxvf logstash-6.3.1.tar.gz

mkdir config

[root@localhost logstash]# vi test.conf

input {

stdin { }

}

output {

stdout {codec=>”rubydebug”}

}

[root@localhost logstash-6.3.1]# ./bin/logstash -f config/test.conf

### 4.3、logstsh操作

[root@localhost logstash-6.3.1]# vi test.conf

input {

stdin {codec=>line}

}

output {

stdout {codec=>json}

}

[root@localhost logstash-6.3.1]# echo “foo

bar”./bin/logstash -f config/test.conf

### 4.4、logstsh input插件

* Stdin

输入插件：可以管道输入，也可以从终端交互输入

通用配置：

codec：类型为codec

type：类型为string自定义该事件类型，可用于后续判断

tags：类型为array，自定义事件的tag，可用于后续判断

add\_field：类型为hash，为该事件添加字段

input{

stdin{

codec => “plain”

tags => [“test”]

type => “std”

add\_field => {“key”:”value”}

}

}

output{

stdout{

codec => “rubydebug”

}

}

[root@localhost logstash-6.3.1]# echo “test”./bin/logstash -f config/test.conf

{

"@version" => "1",

"key" => "value",

"message" => "test",

"type" => "std",

"tags" => [

[0] "test"

],

"host" => "localhost",

"@timestamp" => 2019-03-24T12:20:16.334Z

}

* file

从文件读取数据，如常见的日志文件

配置：

path => [“/var/log/\*\*/\*.log”,”/var/log/message”] 文件位置

exclue => “\*.gz” 不读取哪些文件

sincedb\_path => “/var/log/message” 记录sincedb文件路径

start\_postion => “beginning” 或者”end” 是否从头读取文件

stat\_interval => 1000 单位秒，定时检查文件是否有更新，默认1S

input {

file {

path => ["/home/elk/logstsh/config/nginx\_logs"]

start\_position => "beginning"

type => "web"

}

}

output {

stdout {

codec => "rubydebug"

}

}

{

"path" => "/home/elk/logstsh/config/nginx\_logs",

"message" => "79.136.114.202 - - [04/Jun/2015:07:06:35 +0000] \"GET /downloads/product\_1 HTTP/1.1\" 404 334 \"-\" \"Debian APT-HTTP/1.3 (0.8.16~exp12ubuntu10.22)\"",

"@timestamp" => 2019-03-24T12:47:20.900Z,

"host" => "localhost",

"type" => "web",

"@version" => "1"

}

* Elasticsearch

input {

elasticsearch {

hosts => "192.168.14.10"

index => "atguigu"

query => '{ "query": { "match\_all": {} }}'

}

}

output {

stdout {

codec => "rubydebug"

}

}

### 4.5、logstsh filter

Filter是logstsh功能强大的原因，它可以对数据进行丰富的处理，比如解析数据、删除字段、类型转换等

date：日期解析

grok：正则匹配解析

dissect：分割符解析

mutate：对字段作处理，比如重命名、删除、替换等

json：按照json解析字段内容到指定字段中

geoip：增加地理位置数据

ruby：利用ruby代码来动态修改logstsh Event

input {

stdin {codec => “json”}

}

filter {

date {

match => ["logdate","MM dd yyyy HH:mm:ss"]

}

}

output {

stdout {

codec => "rubydebug"

}

}

{“logdate”:”Jan 01 2018 12:02:08”}

* Grok

正则匹配

%{IPORHOST:clientip} %{USER:ident} %{USER:auth} \[%{HTTPDATE:timestamp}\] “%{WORD:verb} %{DATA:request} HTTP/%{NUMBER:httpversion}” %{NUMBER:response:int} (?:-|%{NUMBER:bytes:int}) %{QS:referrer} %{QS:agent}

input {

http {port => 7474}

}

filter {

grok {

match => {

"message" => "%{IPORHOST:clientip} %{USER:ident} %{USER:auth} \[%{HTTPDATE:timestamp}\] “%{WORD:verb} %{DATA:request} HTTP/%{NUMBER:httpversion}” %{NUMBER:response:int} (?:-|%{NUMBER:bytes:int}) %{QS:referrer} %{QS:agent}"

}

}

}

output {

stdout {

codec => "rubydebug"

}

}

93.180.71.3 - - [17/May/2015:08:05:32 +0000] "GET /downloads/product\_1 HTTP/1.1" 304 0 "-" "Debian APT-HTTP/1.3 (0.8.16~exp12ubuntu10.21)"

93.180.71.3 - - [17/May/2015:08:05:23 +0000] "GET /downloads/product\_1 HTTP/1.1" 304 0 "-" "Debian APT-HTTP/1.3 (0.8.16~exp12ubuntu10.21)"

* Logstsh ouput

stdout

file :

file {

path => “/var/log/web.log”

codec => line {format => “%{message}”}

}

elasticsearch :

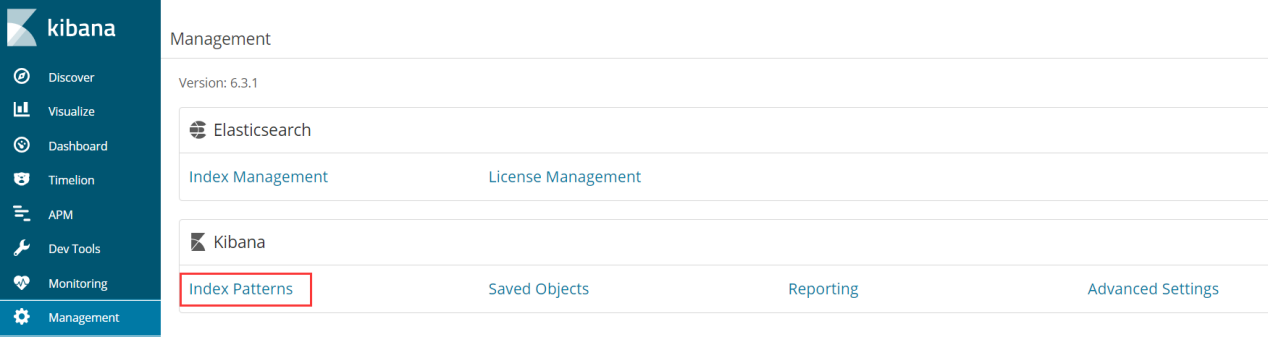
elasticsearch {

hosts => ["http://192.168.14.10:9200"]

index => "logstash-%{type}-%{+YYYY.MM.dd}"

}

## Kibana



5.1、visualize