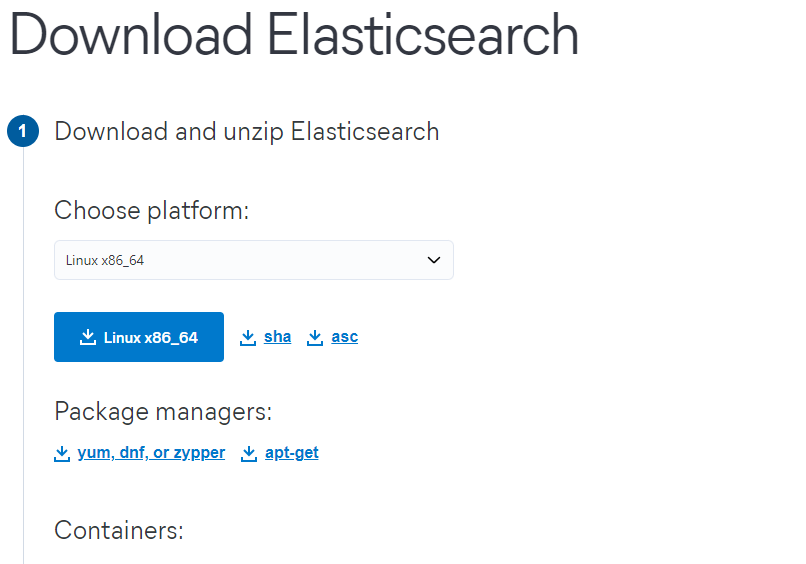
# Elasticsearch安装使用

## 1.1下载elasticsearch

下载[elasticsearch](https://so.csdn.net/so/search?q=elasticsearch&spm=1001.2101.3001.7020)，地址[下载中心 - Elastic 中文社区](https://elasticsearch.cn/download/)，[kibana](https://so.csdn.net/so/search?q=kibana&spm=1001.2101.3001.7020)也可以在这里下载

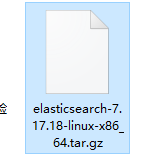
官网下载地址：<https://www.elastic.co/cn/downloads/elasticsearch>

参考https://blog.csdn.net/m0\_50287279/article/details/131819482





下载成功如下（我测试的版本是elasticsearch-7.17.18-linux-x86\_64.tar.gz）



## 1.2 Linux下安装

### 1.2.1 上传

创建一个文件夹，上传到服务器，/local

解压到当前目录 tar -xvf  elasticsearch-7.9.3-linux-x86\_64.tar.gz

### 1.2.1 创建用户

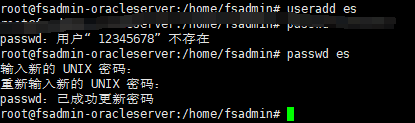
因为elasticsearch  不能使用root用户启动，需要用普通用户启动，如果本来就用普通用户登录，可不用更改

创建用户：

useradd es

设置密码：

passwd es



授权，为了方便安装，我们给整个/usr/local 授权，

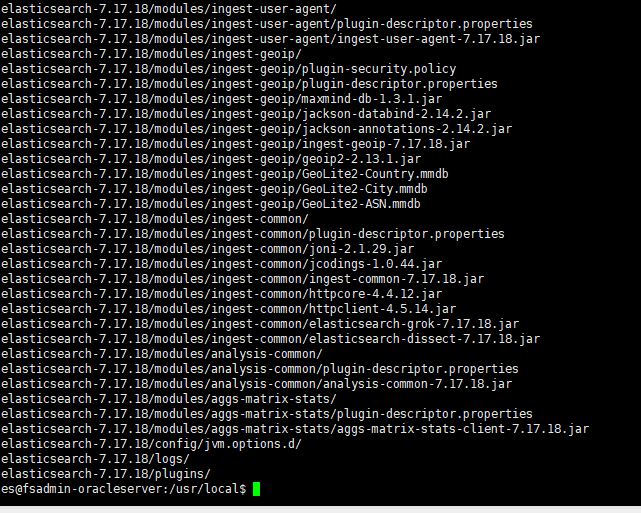


### 1.2.2解压

切换用户es，进入/usr/local目录下

tar –zxvf elasticsearch-7.9.3-linux-x86\_64.tar.gz

解压成功如下：



### 1.2.2修改虚拟内存

由于elasticsearch 默认分配 jvm空间大小为2g，如果服务器内存不大就会报错，所以我们需要修改 jvm空间，如果Linux服务器本来配置就很高，可以不用修改。

cd /usr/local/elasticsearch-7.17.18/config

vim jvm.options

增加如下配置：(根据实际情况修改大小，可以改为1g

Xms256m

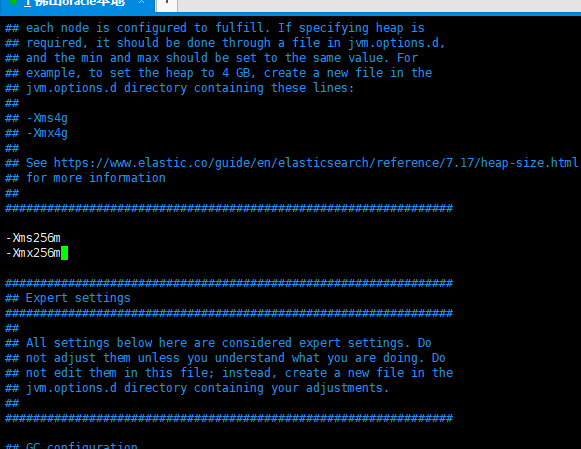
-Xmx256m

### 1.2.3创建数据文件

在/usr/local/elasticsearch-7.17.18下，创建一个文件夹data

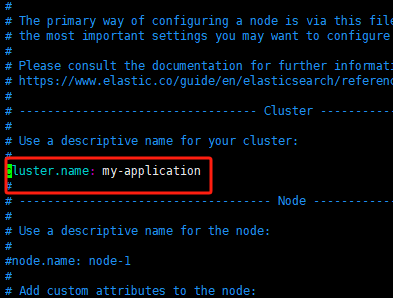
### 1.2.4修改配置文件elasticsearch.yml

在/usr/local/elasticsearch-7.17.18/config下，修改elasticsearch.yml，修改如下几点：



1.修改配置集群名字cluster.name：

cluster.name: my-application



2、配置节点

node.name: node-1 #节点名称，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

http.port: 9200

transport.tcp.port: 9300

#discovery.zen.ping.unicast.hosts: ["192.168.6.125","192.168.6.155"] # 集群个节点IP地址，也可以使用域名，需要各节点能够解析

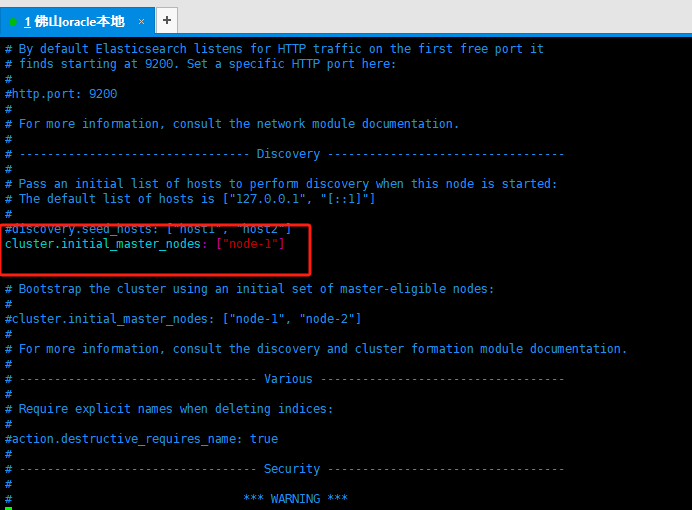


3、启动时加载集群节点

cluster.initial\_master\_nodes: ["node-1"]

注意：

**node-1必须是已经定义的节点**



4、修改数据文件和日志文件路径

path.data: /usr/local/elasticsearch-7.17.18/data

#

# Path to log files:

#

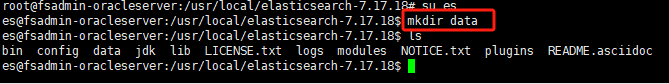
path.logs: /usr/local/elasticsearch-7.17.18/logs



5、设置允许远程连接

network.host: 0.0.0.0





### 1.2.5启动

进入安装目录的bin下，如/usr/local/elasticsearch-7.17.18/bin

执行如下命令，必须是非root用户

cd /usr/local/elasticsearch-7.17.18/bin

./elasticsearch

后台启动

./elasticsearch -d



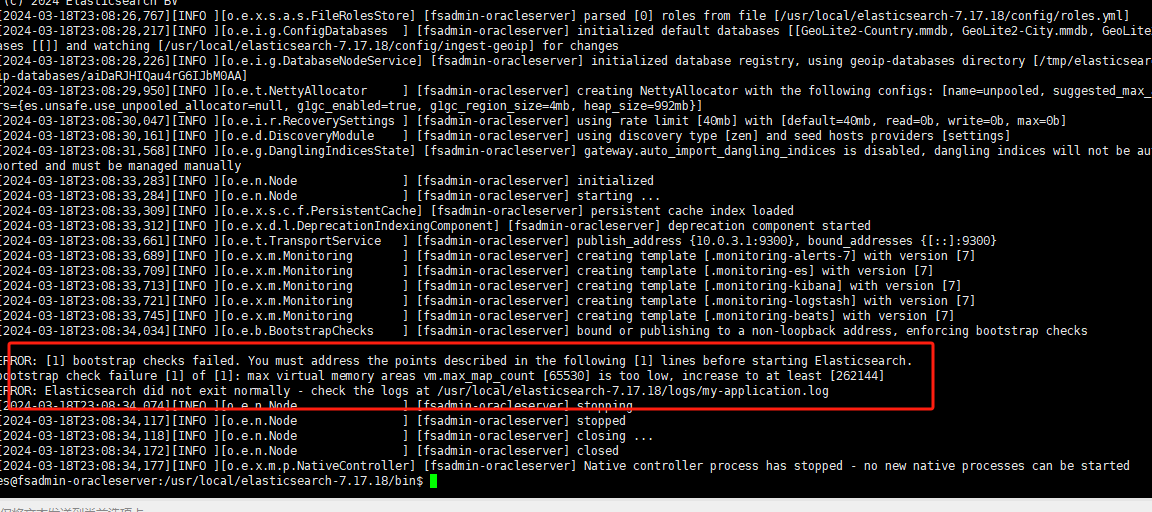
启动成功



遇到的错误

1. 进程数不够

bootstrap check failure [1] of [1]: max virtual memory areas vm.max\_map\_count [65530] is too low, increase to at least [262144]



解决：

**临时方案：**

sysctl -w vm.max\_map\_count=262144

1）.切换到root用户执行命令：

sysctl -w vm.max\_map\_count=262144

2）.查看结果：

sysctl -a|grep vm.max\_map\_count

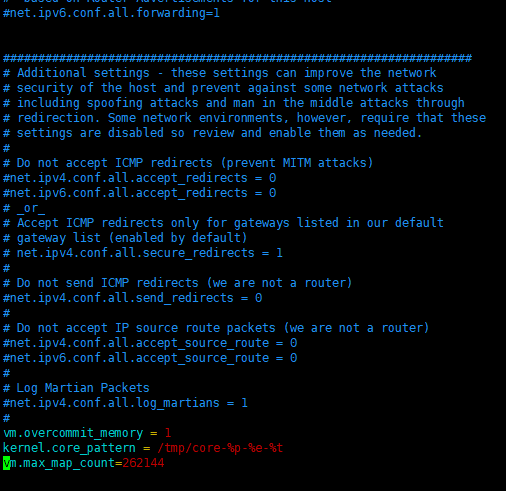
重启后会失效

**永久方案：**

修改/etc/sysctl.conf，在文件最后增加vm.max\_map\_count=262144配置

vim /etc/sysctl.conf

vm.max\_map\_count=262144



重启服务

**我还在网上找了一些**



、



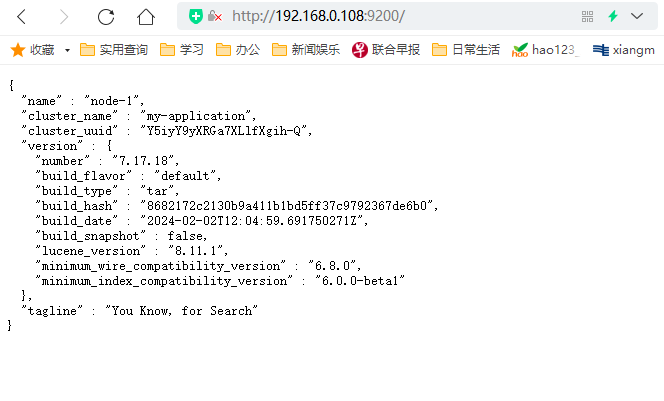


### 1.2.6测试是否启动

可以查看进程

ps -ef |grep  elasticsearch

在浏览器输入<http://192.168.0.108:9200/>



### 1.2.7停止服务

查看Elasticsearch进程  ps -ef |grep  elasticsearch  然后 kill -9  进程号

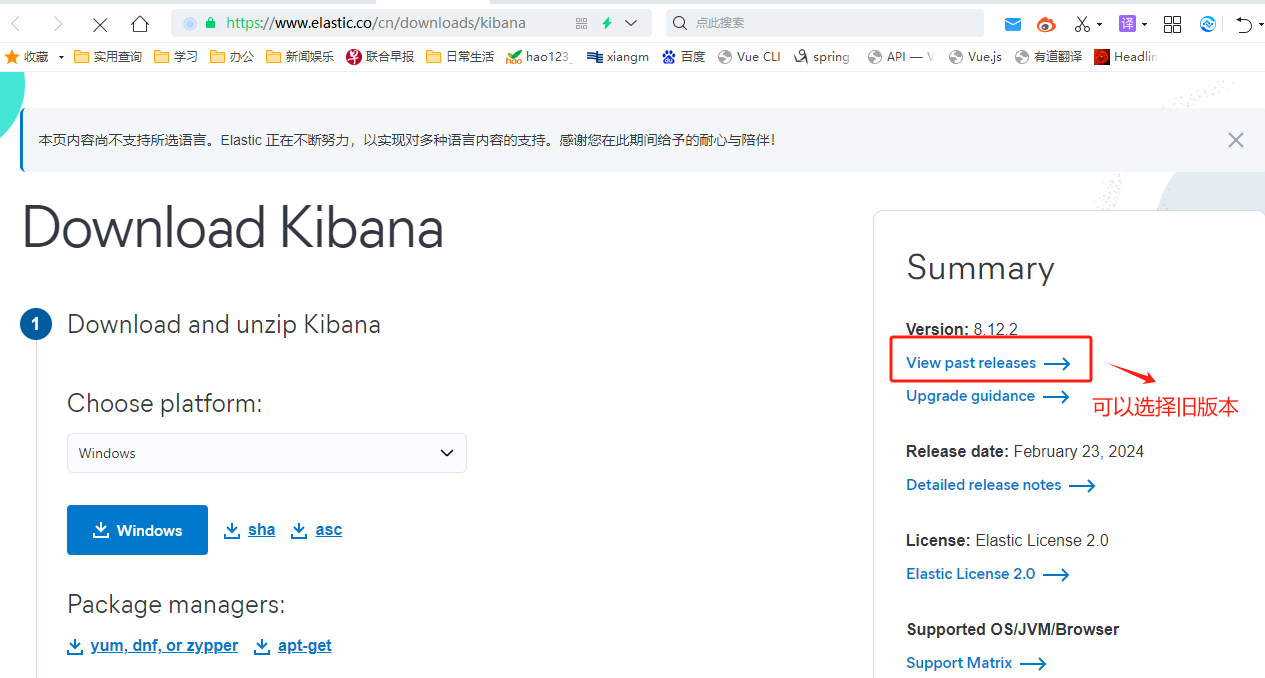
# kibana安装和使用

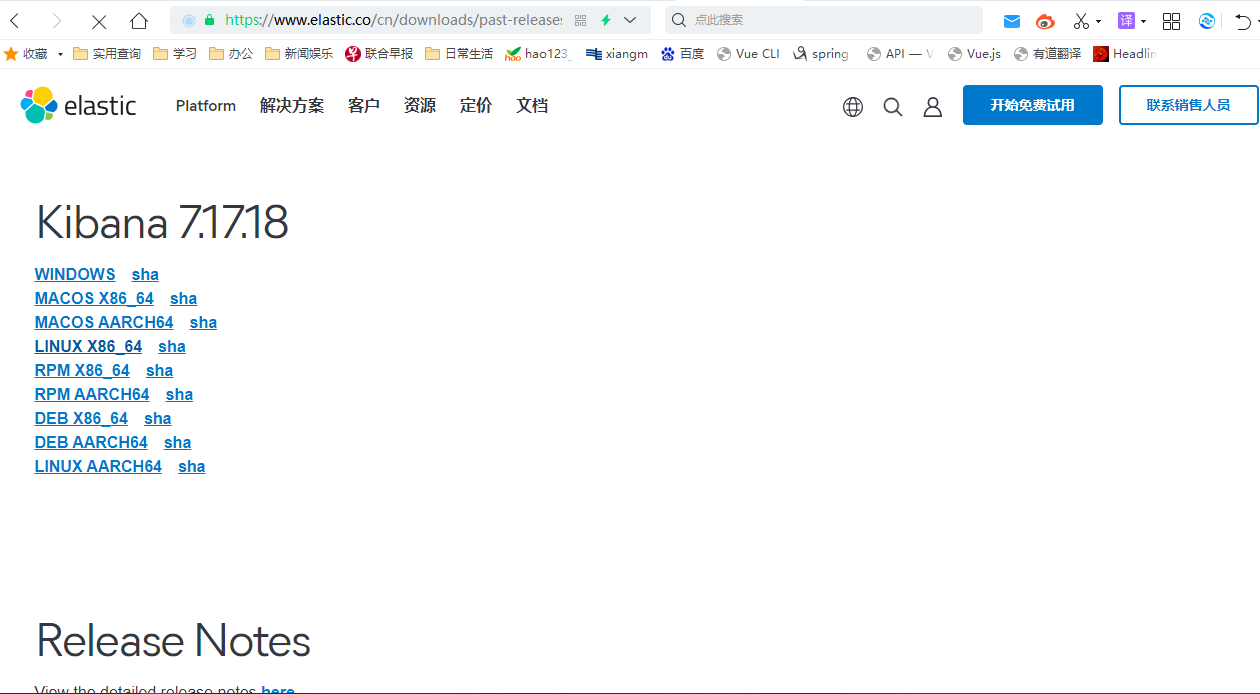
Kibana是一个基于Node.js的Elasticsearch索引库数据统计工具，专为[Elasticsearch](https://www.baidu.com/s?wd=Elasticsearch&rsv_idx=2&tn=62095104_15_oem_dg&usm=2&ie=utf-8&rsv_pq=dfd58bcb0010a400&oq=kibana%E8%AF%A6%E7%BB%86%E6%95%99%E7%A8%8B&rsv_t=09b64s9UefpWXa1MYtLFes3o2J89FT7c7kAyf6%2BmvGWsrffi3q2UeFU54oS4Z8KIBqH2YktaZa9l&sa=re_dqa_zy&icon=1)设计，是Elasticsearch的客户端工具，在Kibana可以对es数据操作。可以利用Elasticsearch的聚合功能，生成各种图表，如柱形图，线状图，饼图等。

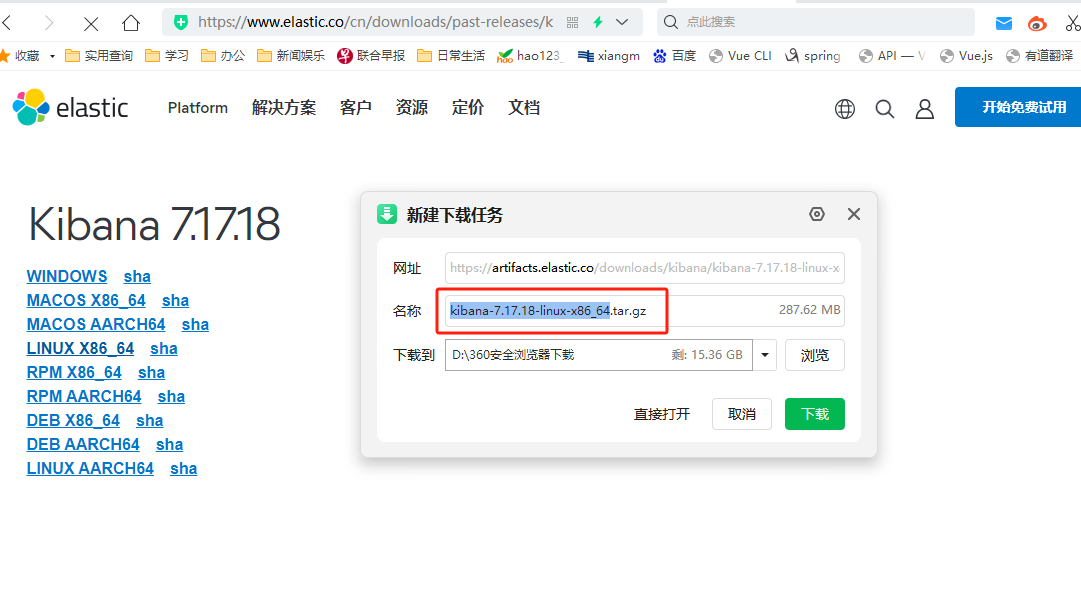
而且还提供了操作Elasticsearch索引数据的控制台，并且提供了一定的AP!提示，非常有利于我们学习Elasticsearch的语法。

## 2.1下载Kibana

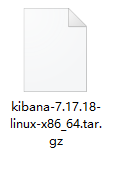
安装包下载地址：<https://www.elastic.co/cn/downloads/kibana>,打开官方，选择版本下载即可。也可以选择旧版本







下载成功如下：

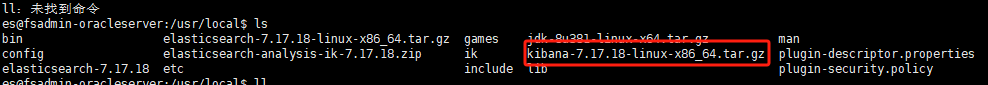


## 2.2 linux安装Kibana

参考https://blog.csdn.net/fengyuyeguirenenen/article/details/124036390

### 2.2.1 上传

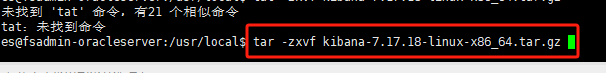
将kibana-7.17.18-linux-x86\_64.tar.gz上传到/usr/local/目录下



### 2.2.2 解压安装

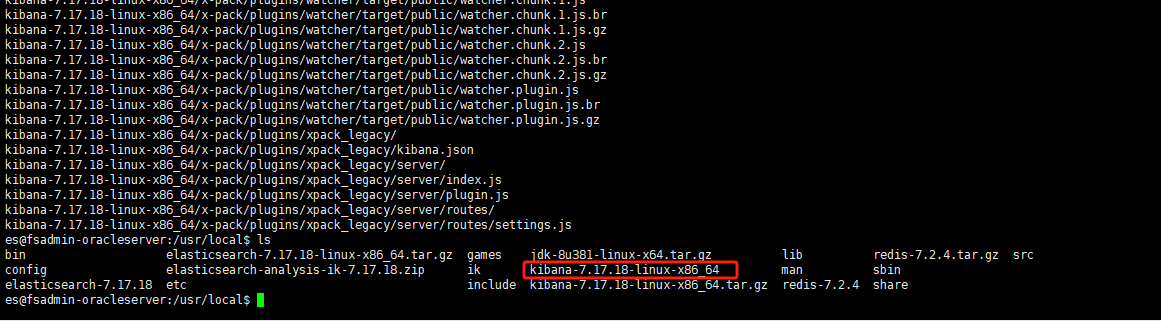
切换到es账号

tar –zxvf kibana-7.17.18-linux-x86\_64.tar.gz



解压过程比较慢，还有配置nodejs环境

解压完成后如下：



### 2.2.3修改配置文件

编辑kibana 安装目录下的config/kibana.yml

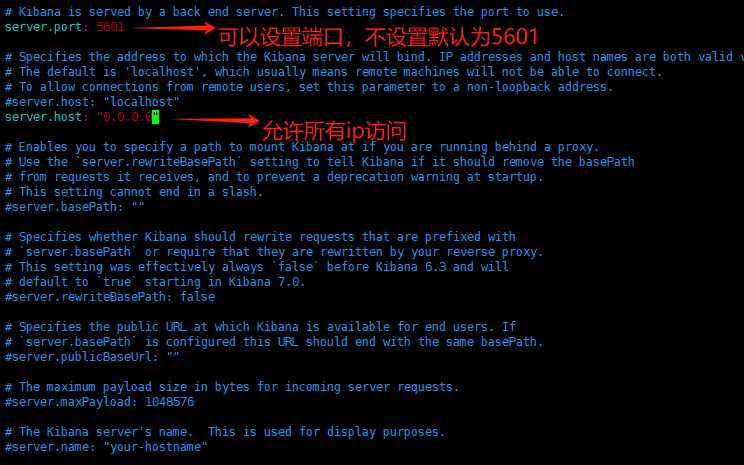
例如：/usr/local/kibana-7.17.18-linux-x86\_64/config/kibana.yml

设置监听端口号、es地址、索引名

主要设置如下：

server.port: 5601

server.host: "0.0.0.0"



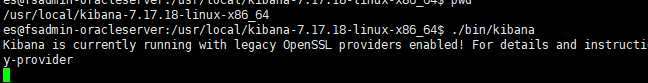




### 2.2.4启动

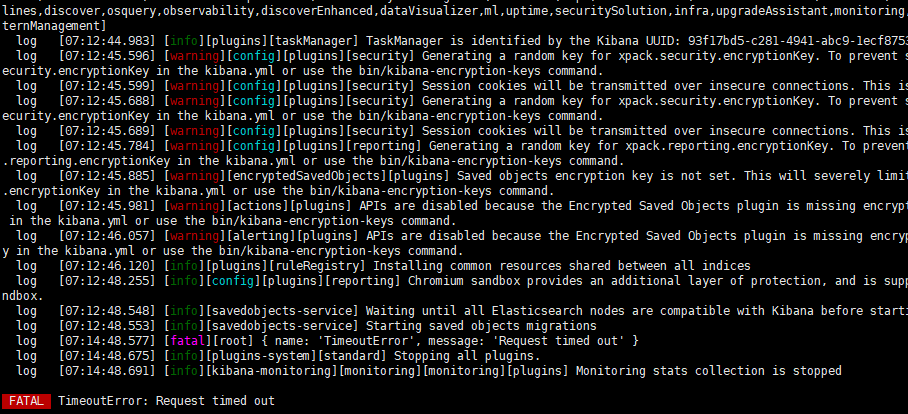
在安装目录下执行如下：

./bin/kibana



**遇到问题：**

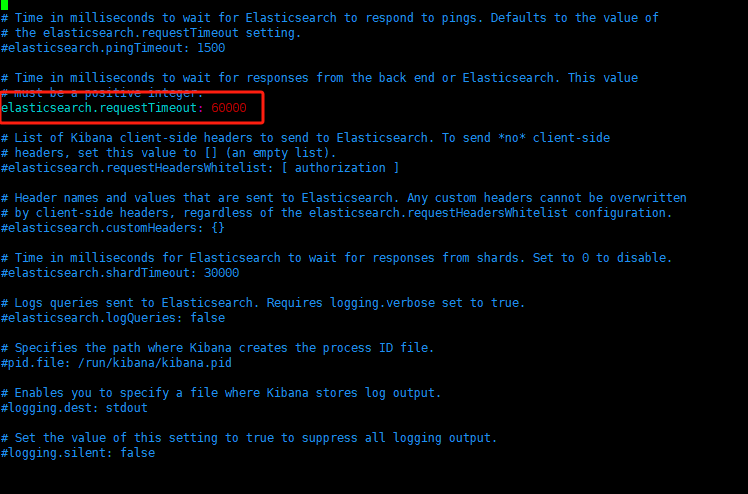
1. 超时



解决办法：

设置kibana 安装目录下的config/kibana.yml 中的属性elasticsearch.requestTimeout

elasticsearch.requestTimeout: 60000



1. FATAL {"error":{"root\_cause":[{"type":"master\_not\_discovered\_exception","reason":null}],"type":"master\_not\_discovered\_exception","reason":null},"status":503}



解决办法，在es的配置文件中设置集群节点

在/usr/local/elasticsearch-7.17.18/config下，修改elasticsearch.yml

node.name: node-1

node.master: true

node.data: true

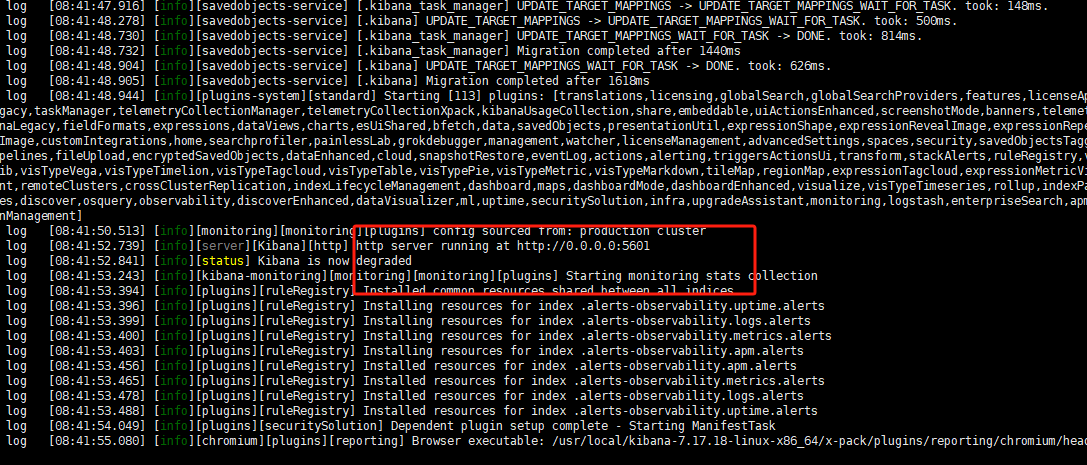
http.port: 9200

transport.tcp.port: 9300

#discovery.zen.ping.unicast.hosts: ["192.168.6.125","192.168.6.155"] # 集群个节点IP地址，也可以使用域名，需要各节点能够解析



启动成功如下：



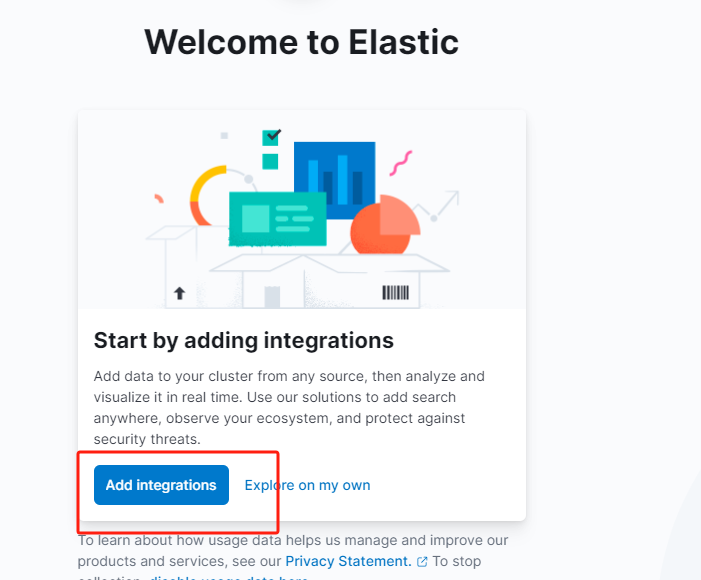
### 2.2.5进入管理界面测试

打开http://localhost:5601，画风如下

http://192.168.0.108:5601/

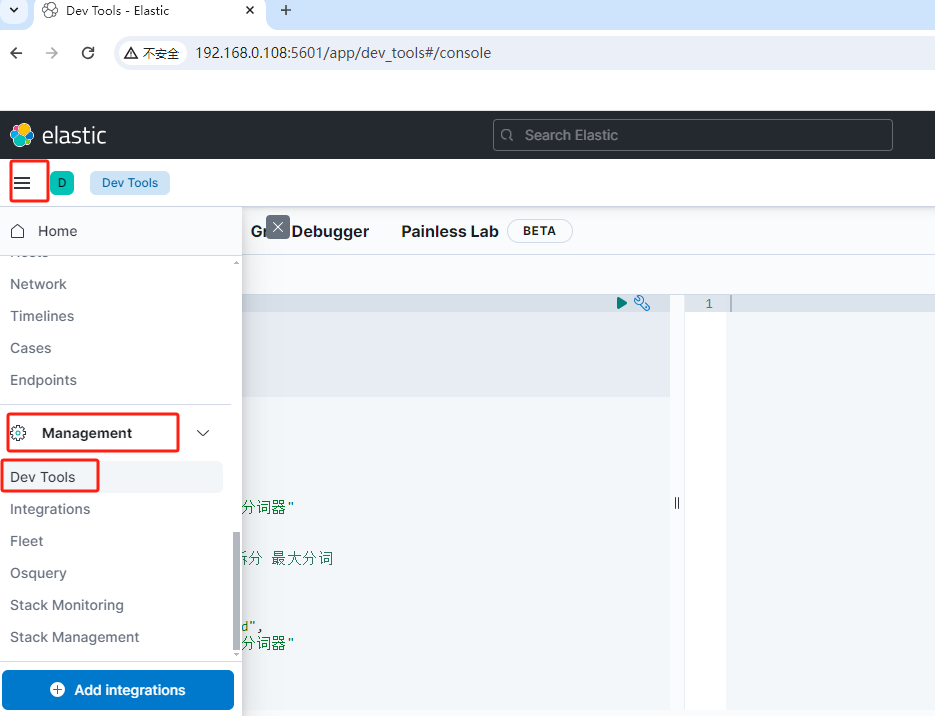


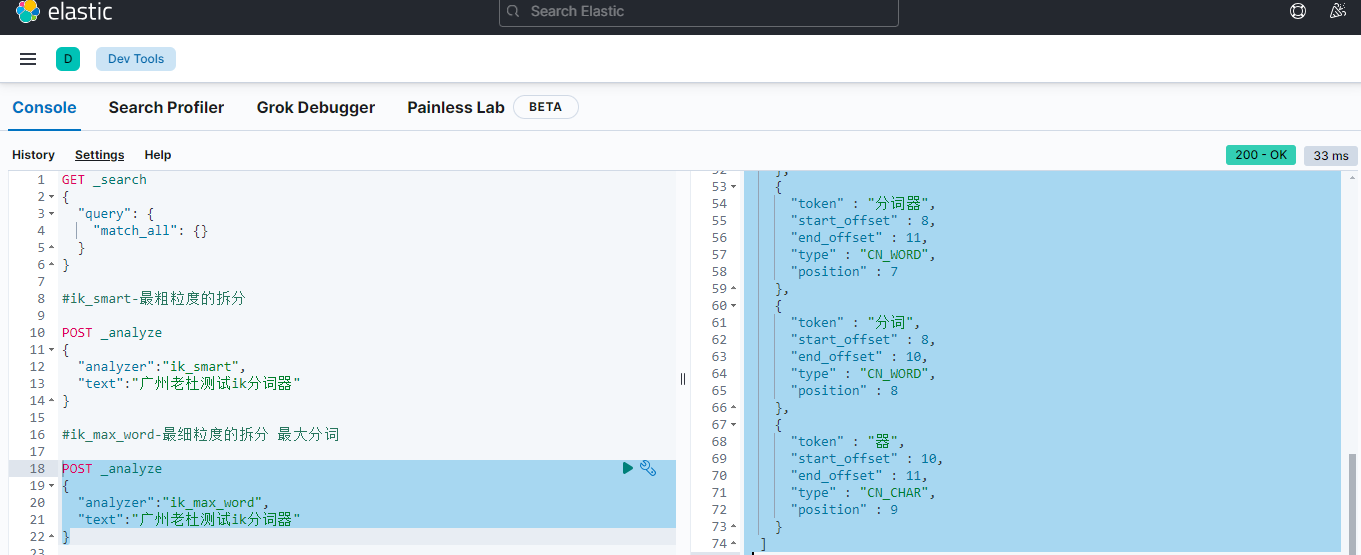
提示我们可以使用示例数据，也可以使用自己已有的数据，我把示例数据都下载了，单击侧面导航中的 Discover 进入 Kibana 的数据探索功能：



Kibana启动后，在浏览器输入<http://192.168.0.108:560/>，进入后台，点开左侧菜单，选择左侧的DevTools菜单，即可进入控制台页面，可以在这里测试分词器了：

或者直接输入<http://192.168.0.108:5601/app/dev_tools#/console>





详情见分词器使用

### 2.2.6切换为中文

在kibana安装目录的config/kibana.yml添加

i18n.locale: "zh-CN"

重新启动，即可生效

# 分词器安装使用

ElasticSearch 内置了分词器，如标准分词器、简单分词器、空白词器等。但这些分词器对我们最常使用的中文并不友好，不能按我们的语言习惯进行分词。

ik分词器就是一个标准的中文分词器。它可以根据定义的字典对域进行分词，并且支持用户配置自己的字典，所以它除了可以按通用的习惯分词外，我们还可以定制化分词。

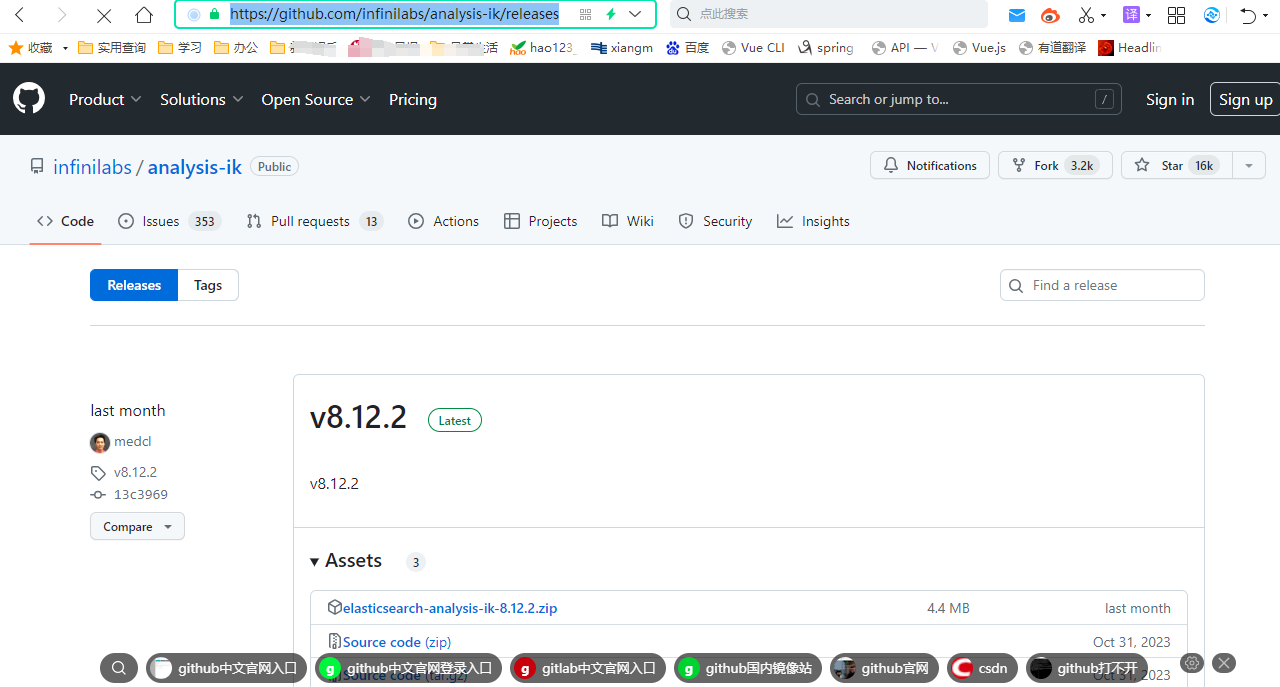
ik分词器是一个插件包，我们可以用插件的方式将它接入到ES。

**分词器使用之前，必须启动elasticsearch和kibana**

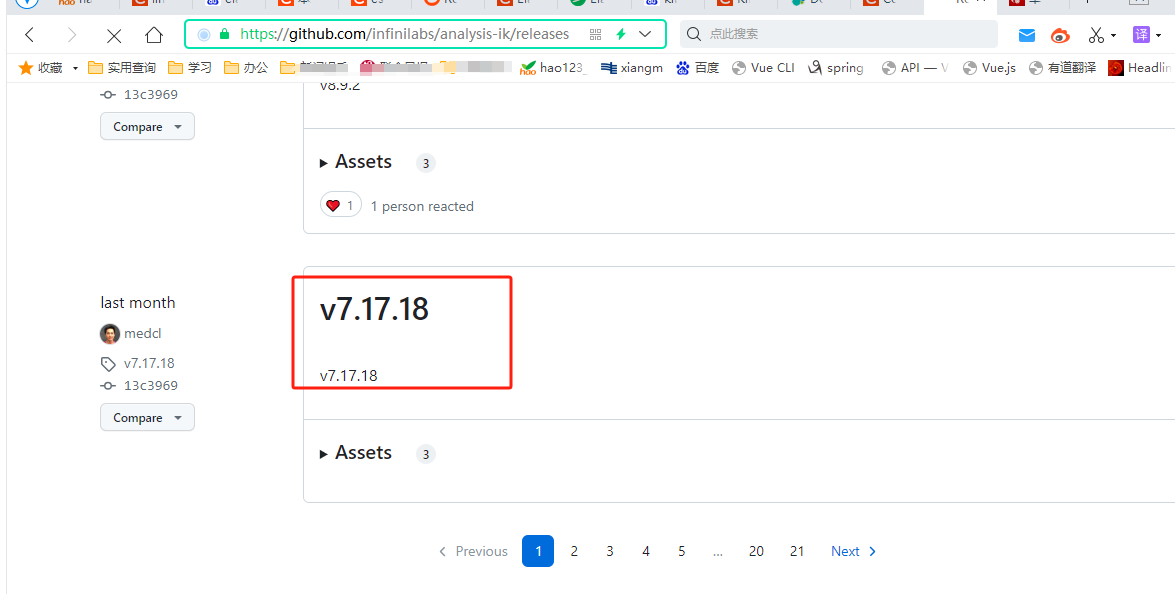
## 3.1分词器下载

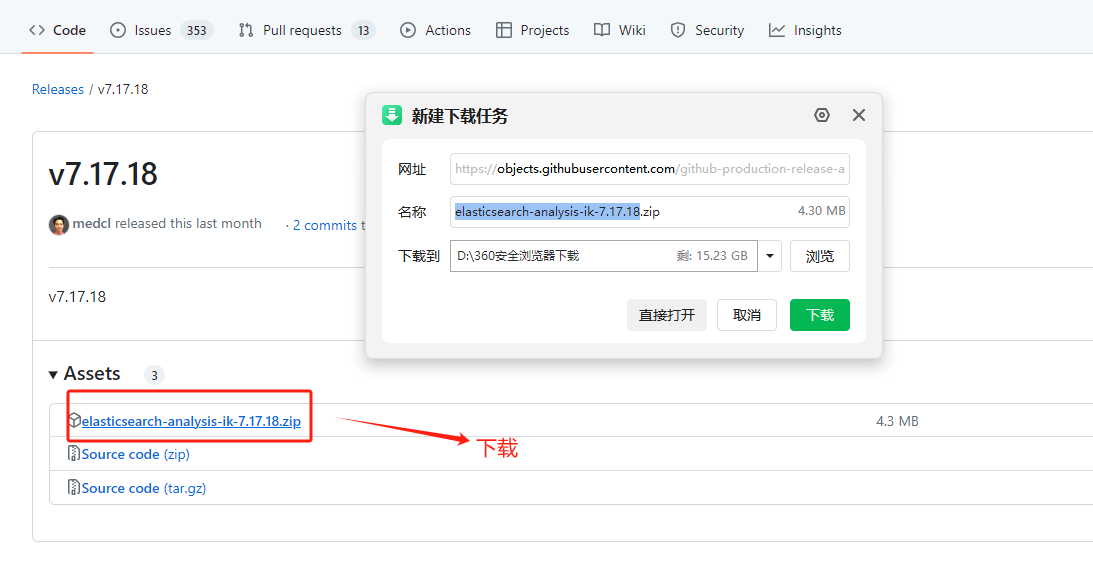
下载地址：

<https://github.com/infinilabs/analysis-ik/releases>，根据不同的版本下载分词器，我们

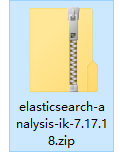


我们elasticsearch版本是7.17.18，所以分词器最好也下载7.17.18版本





下载成功如下：

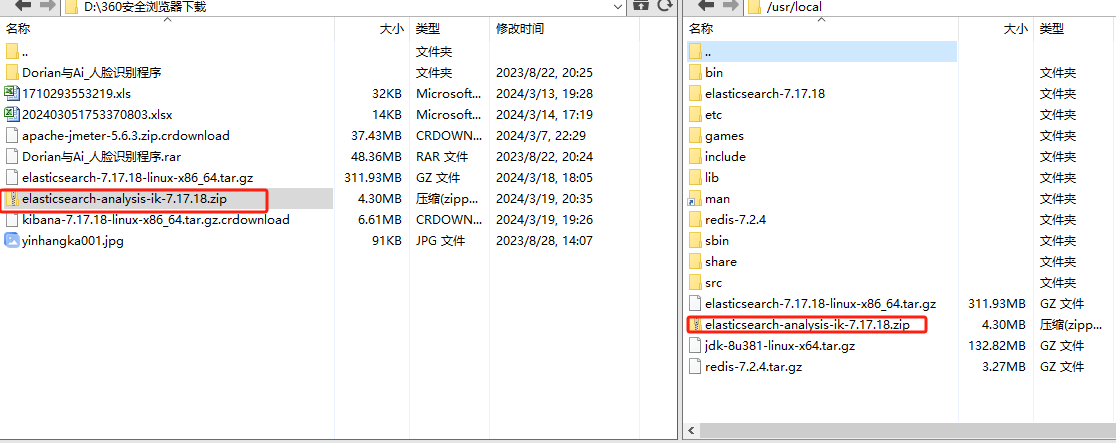


## 3.2 Linux安装分词器

参考https://blog.csdn.net/aq\_money/article/details/130440968

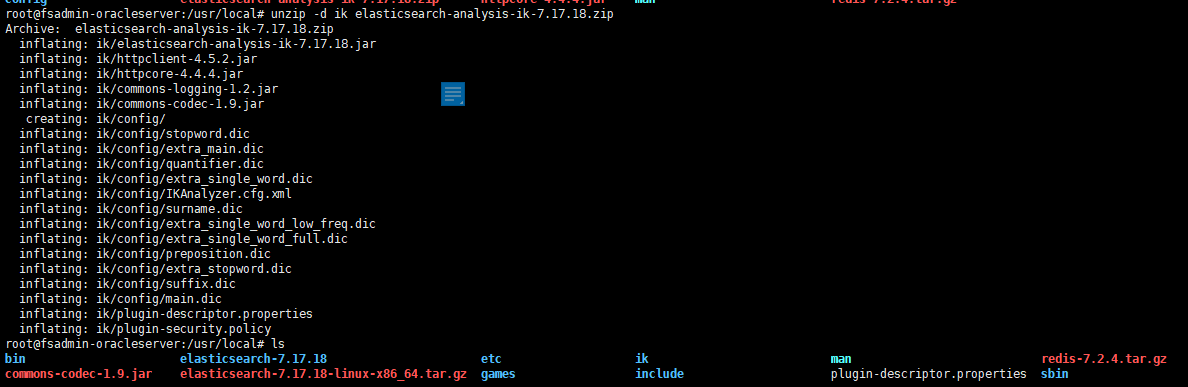
### 3.2.1 上传到Linux

把分词器上传到linux系统

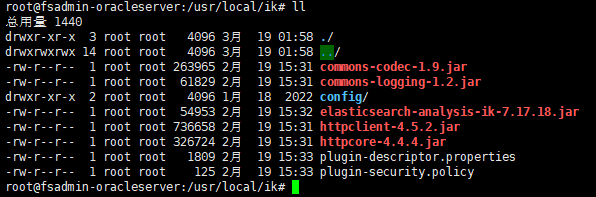


### 3.2.2 Linux解压

解压到ik目录下

unzip -d ik elasticsearch-analysis-ik-7.17.18.zip 

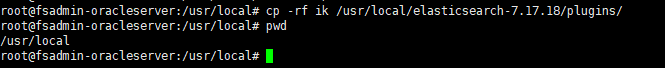
解压成功如下：



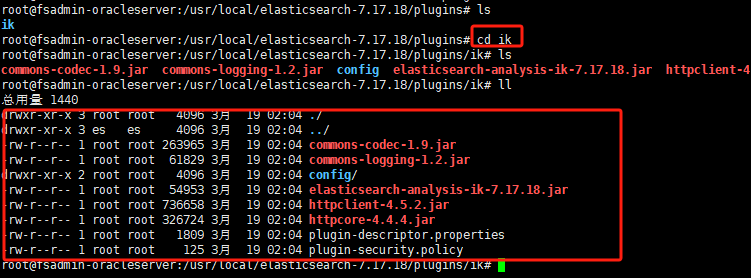
### 3.2.3 安装到elasticsearch

将解压的分词器ik整个文件夹拷贝到elasticsearch安装目录下的plugins下

cp -rf ik /usr/local/elasticsearch-7.17.18/plugins/



拷贝完成后如下图：

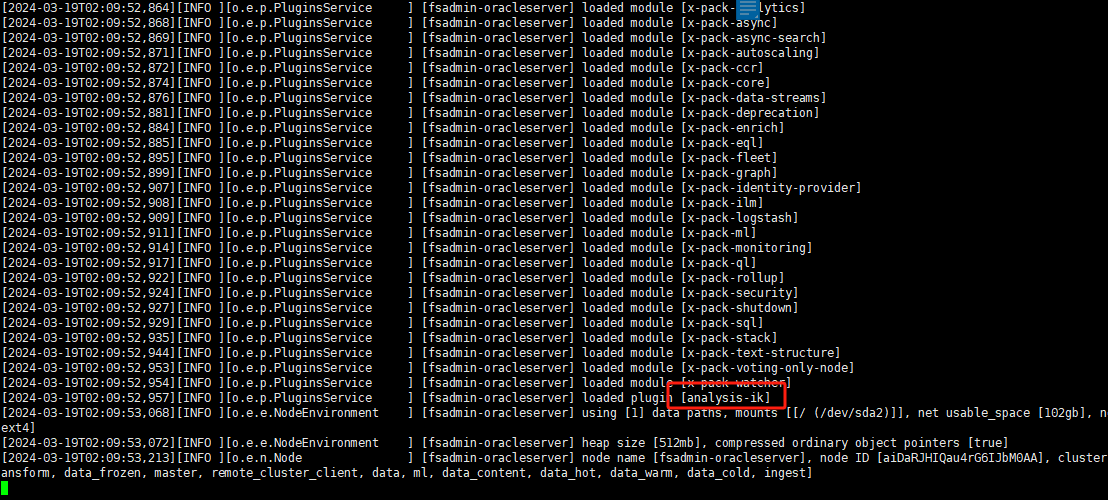


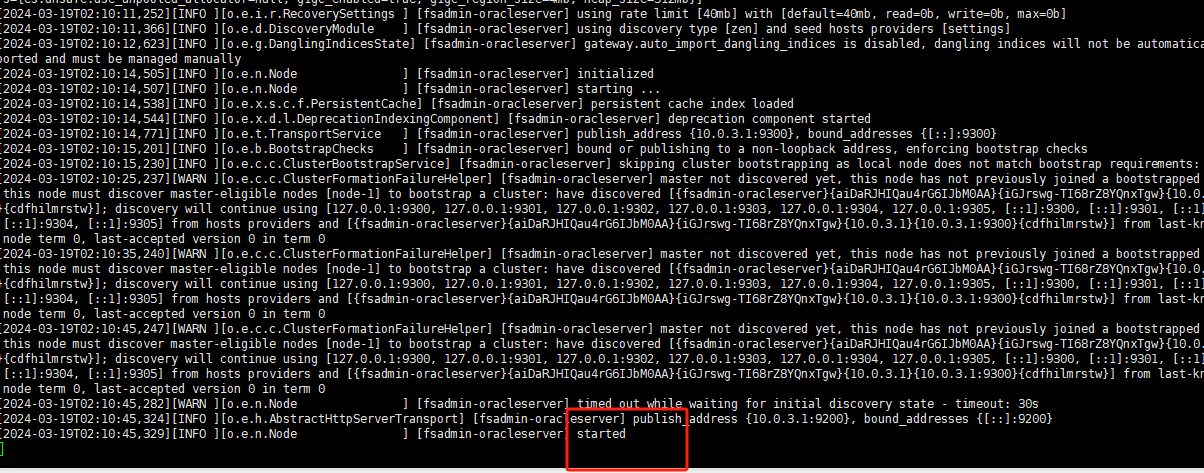
### 3.2.4 验证是否安装成功

重启elasticsearch

./elasticsearch

看到如下并成功启动说明安装成功





## 3.3分词器使用

IK分词器有两种分词模式：ik\_max\_word和ik\_smart模式。

**ik\_max\_word：**

会将文本做最细粒度的拆分，比如会将"曾舒琪董事长早上好"拆分为"曾、舒琪、董事长、董事、长、早上好、早上、上好"

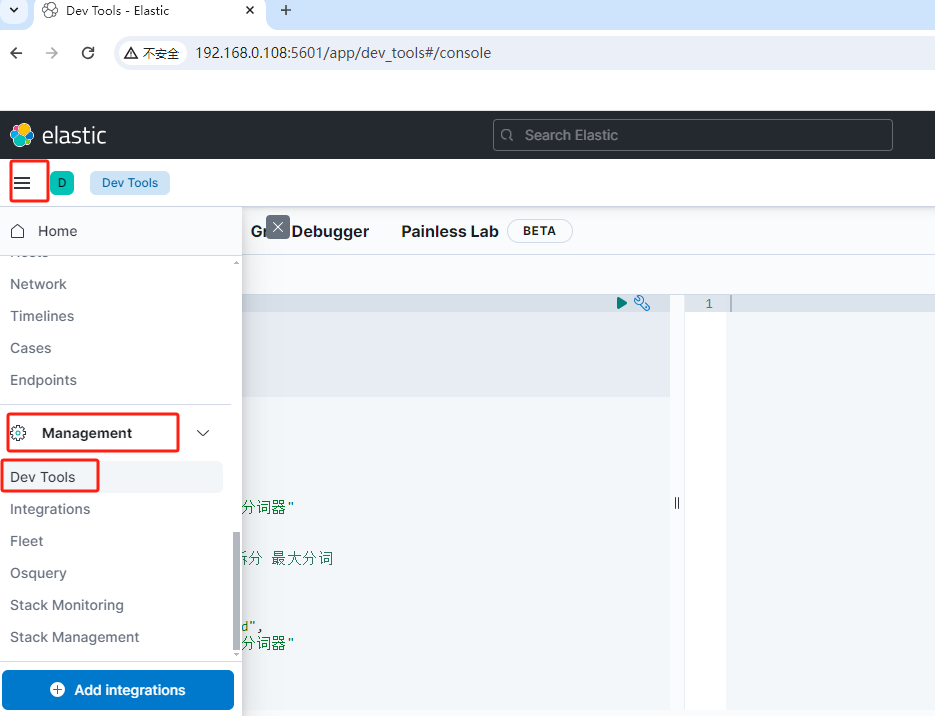
**ik\_smart：**

会做最粗粒度的拆分，比如会将"曾舒琪董事长早上好"拆分成"曾、舒琪、董事长、早上好"

### 3.3.1 进入连接es后台测试分词器

Kibana启动后，在浏览器输入<http://192.168.0.108:5601/>，进入后台，点开左侧菜单，选择左侧的DevTools菜单，即可进入控制台页面，可以在这里测试分词器了：

或者直接输入http://192.168.0.108:5601/app/dev\_tools#/console



### 3.3.2 分词器两张模式测试

**1、ik\_max\_word测试：**

在左侧输入

POST \_analyze

{

"analyzer":"ik\_max\_word",

"text":"广州老杜测试ik分词器"

}

拆分为：

{

"tokens" : [

{

"token" : "广州",

"start\_offset" : 0,

"end\_offset" : 2,

"type" : "CN\_WORD",

"position" : 0

},

{

"token" : "老",

"start\_offset" : 2,

"end\_offset" : 3,

"type" : "CN\_CHAR",

"position" : 1

},

{

"token" : "杜",

"start\_offset" : 3,

"end\_offset" : 4,

"type" : "CN\_CHAR",

"position" : 2

},

{

"token" : "测试",

"start\_offset" : 4,

"end\_offset" : 6,

"type" : "CN\_WORD",

"position" : 3

},

{

"token" : "ik",

"start\_offset" : 6,

"end\_offset" : 8,

"type" : "ENGLISH",

"position" : 4

},

{

"token" : "分词器",

"start\_offset" : 8,

"end\_offset" : 11,

"type" : "CN\_WORD",

"position" : 5

},

{

"token" : "分词",

"start\_offset" : 8,

"end\_offset" : 10,

"type" : "CN\_WORD",

"position" : 6

},

{

"token" : "器",

"start\_offset" : 10,

"end\_offset" : 11,

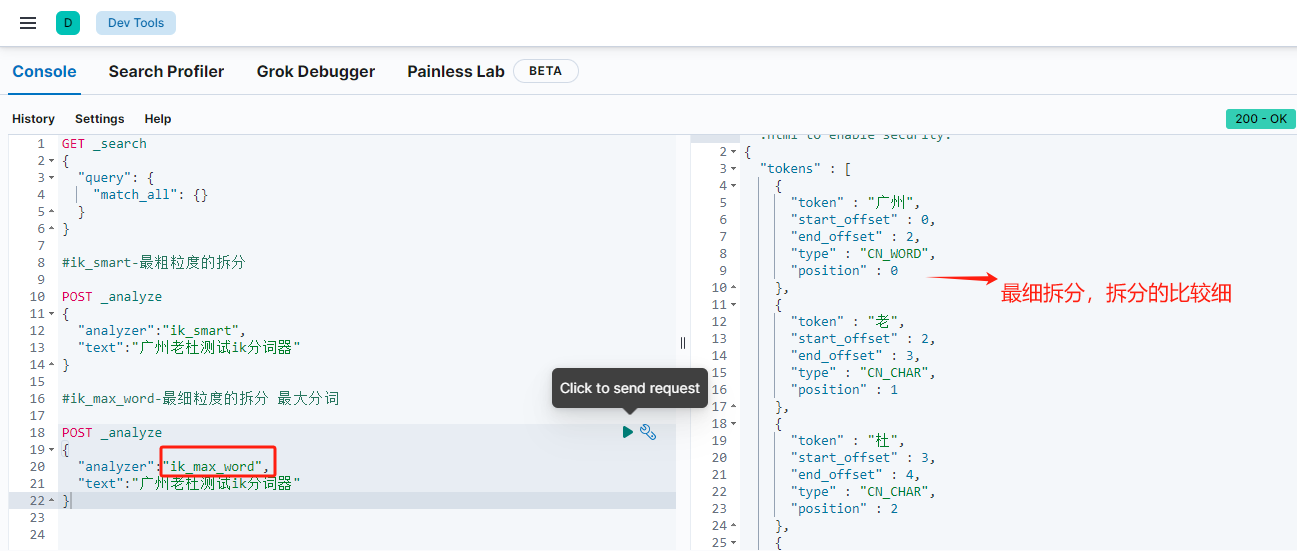
"type" : "CN\_CHAR",

"position" : 7

}

]

}



**2、ik\_smart：**

左侧输入

POST \_analyze

{

"analyzer":"ik\_smart",

"text":"广州老杜测试ik分词器"

}

拆分为：

{

"tokens" : [

{

"token" : "广州",

"start\_offset" : 0,

"end\_offset" : 2,

"type" : "CN\_WORD",

"position" : 0

},

{

"token" : "老",

"start\_offset" : 2,

"end\_offset" : 3,

"type" : "CN\_CHAR",

"position" : 1

},

{

"token" : "杜",

"start\_offset" : 3,

"end\_offset" : 4,

"type" : "CN\_CHAR",

"position" : 2

},

{

"token" : "测试",

"start\_offset" : 4,

"end\_offset" : 6,

"type" : "CN\_WORD",

"position" : 3

},

{

"token" : "ik",

"start\_offset" : 6,

"end\_offset" : 8,

"type" : "ENGLISH",

"position" : 4

},

{

"token" : "分词器",

"start\_offset" : 8,

"end\_offset" : 11,

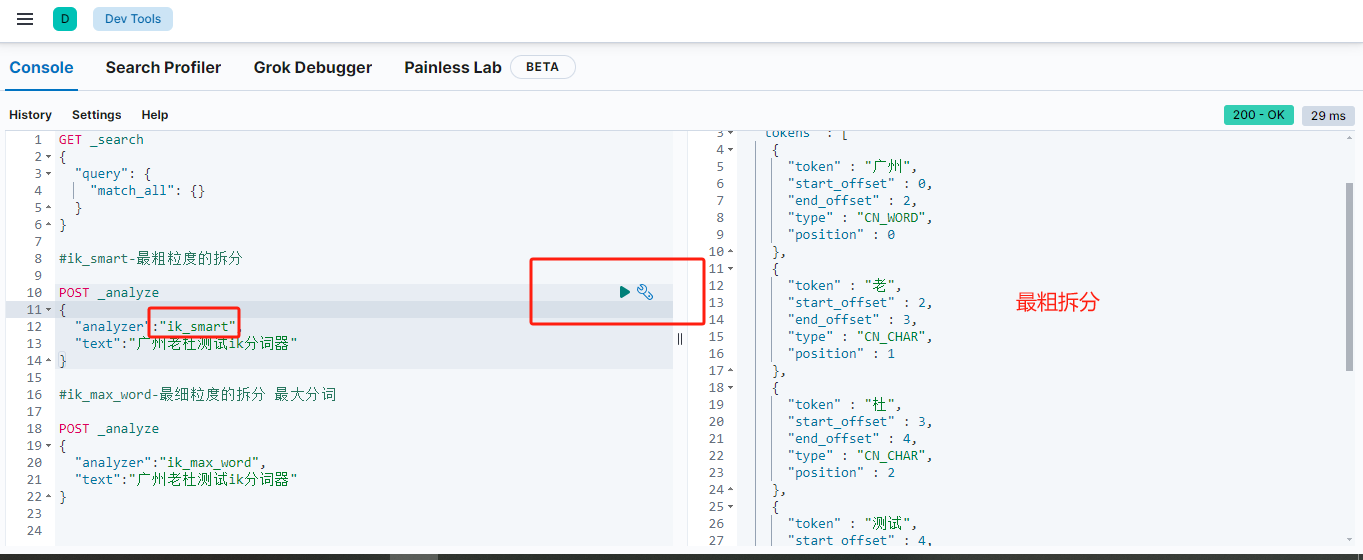
"type" : "CN\_WORD",

"position" : 5

}

]

}

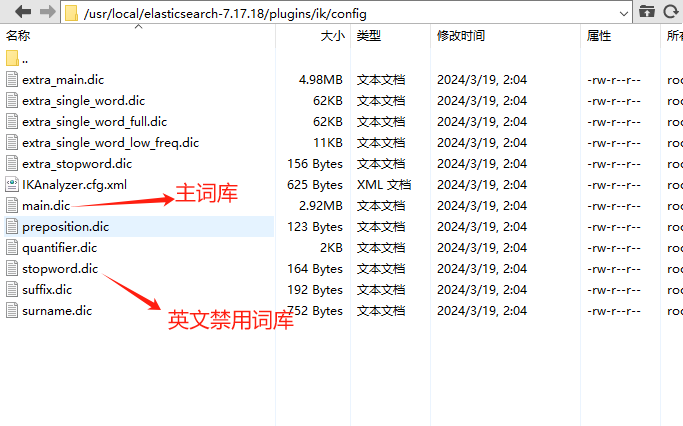


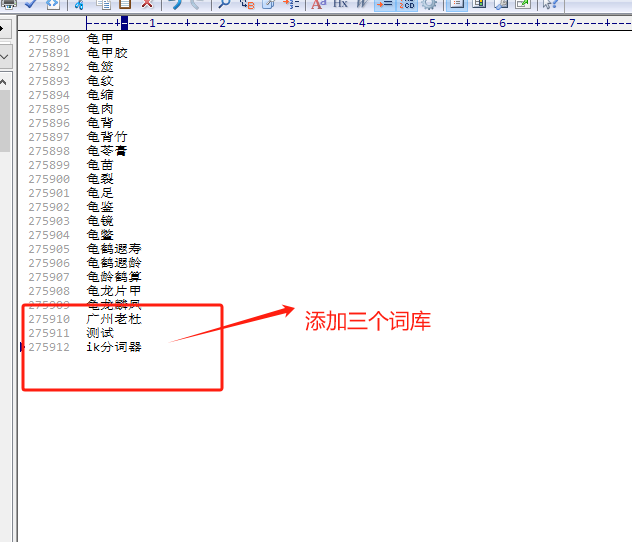
### 3.3.3 自定义词库设置和测试

因为分词器安装在es安装目录下的plugins中，自定义词库配置在分词器的安装目录下的config文件夹中

例如：

/usr/local/elasticsearch-7.17.18/plugins/ik/config

在主词库main.dic文件中添加我们定义的词库 广州老杜、测试、ik分词器



然后重启es和kibana服务

再测试分词器效果

1. **ik\_smart：**

输入：

POST \_analyze

{

"analyzer":"ik\_smart",

"text":"广州老杜测试ik分词器"

}

拆分

{

"tokens" : [

{

"token" : "广州老杜",

"start\_offset" : 0,

"end\_offset" : 4,

"type" : "CN\_WORD",

"position" : 0

},

{

"token" : "测试",

"start\_offset" : 4,

"end\_offset" : 6,

"type" : "CN\_WORD",

"position" : 1

},

{

"token" : "ik分词器",

"start\_offset" : 6,

"end\_offset" : 11,

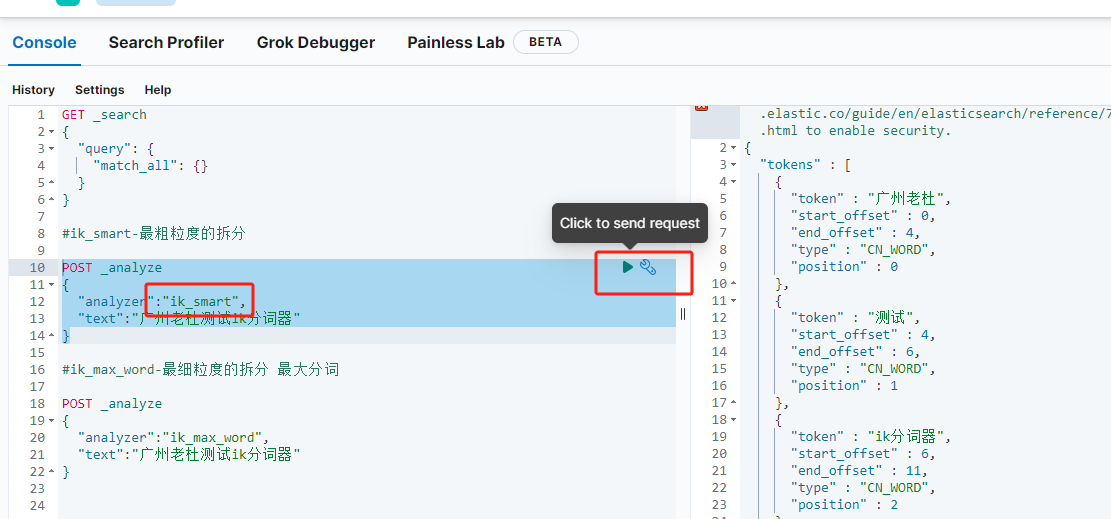
"type" : "CN\_WORD",

"position" : 2

}

]

}



1. **ik\_max\_word测试：**

输入：

POST \_analyze

{

"analyzer":"ik\_max\_word",

"text":"广州老杜测试ik分词器"

}

拆分为：

{

"tokens" : [

{

"token" : "广州老杜",

"start\_offset" : 0,

"end\_offset" : 4,

"type" : "CN\_WORD",

"position" : 0

},

{

"token" : "广州",

"start\_offset" : 0,

"end\_offset" : 2,

"type" : "CN\_WORD",

"position" : 1

},

{

"token" : "老",

"start\_offset" : 2,

"end\_offset" : 3,

"type" : "CN\_CHAR",

"position" : 2

},

{

"token" : "杜",

"start\_offset" : 3,

"end\_offset" : 4,

"type" : "CN\_CHAR",

"position" : 3

},

{

"token" : "测试",

"start\_offset" : 4,

"end\_offset" : 6,

"type" : "CN\_WORD",

"position" : 4

},

{

"token" : "ik分词器",

"start\_offset" : 6,

"end\_offset" : 11,

"type" : "CN\_WORD",

"position" : 5

},

{

"token" : "ik",

"start\_offset" : 6,

"end\_offset" : 8,

"type" : "ENGLISH",

"position" : 6

},

{

"token" : "分词器",

"start\_offset" : 8,

"end\_offset" : 11,

"type" : "CN\_WORD",

"position" : 7

},

{

"token" : "分词",

"start\_offset" : 8,

"end\_offset" : 10,

"type" : "CN\_WORD",

"position" : 8

},

{

"token" : "器",

"start\_offset" : 10,

"end\_offset" : 11,

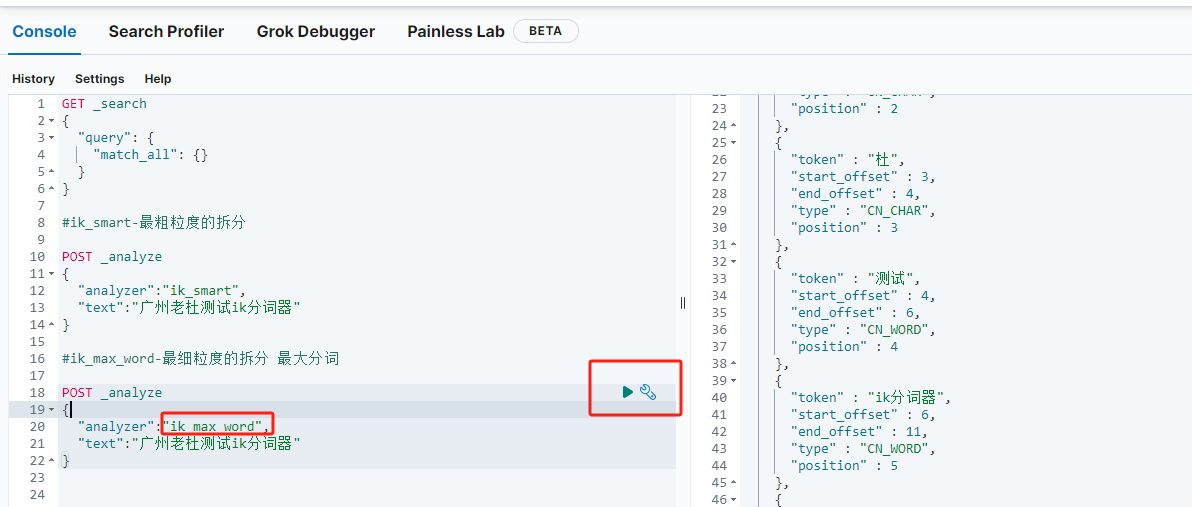
"type" : "CN\_CHAR",

"position" : 9

}

]

}



# Elasticsearch基本操作

ES是基于RESTfuI实现访问，基于REST的基本访问规范



Elasticsearch也是基于Lucene的全文检索库，本质也是存储数据，很多概念与MySQL类似的。

**对比关系：**

**索引（indices）----------------------Databases 数据库**

**类型（type）--------------------------Table 数据表 [7.x废弃]**

**文档（Document）----------------------Row 行，相当于数据表中的一条记录**

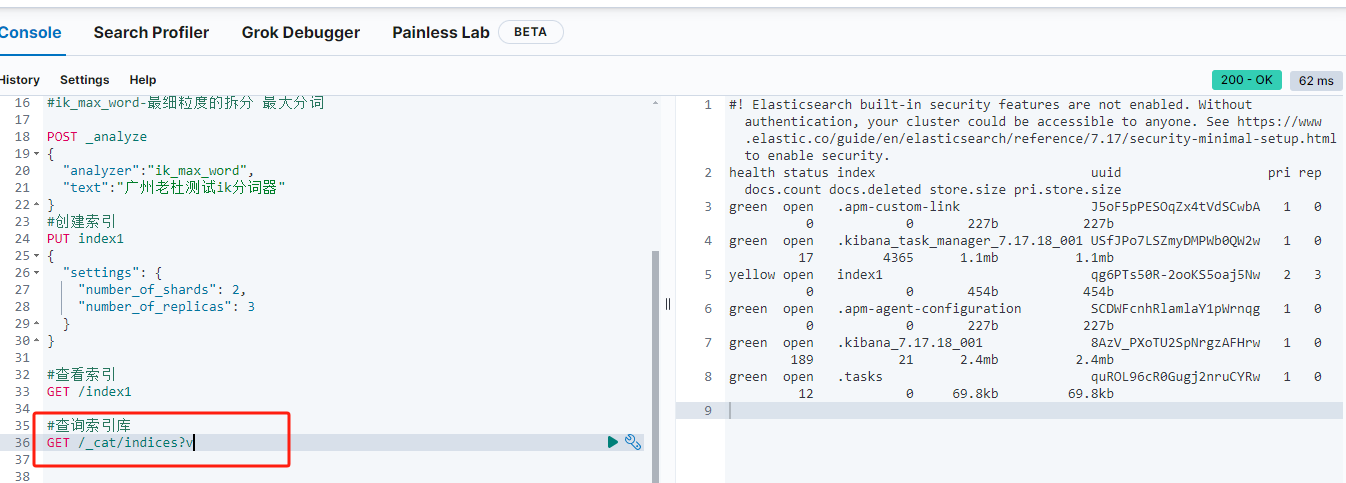
**字段（Field）-------------------------Columns 列**

## 4.1索引操作

### 4.1.1 查询索引库

**查看es中有哪些索引库（类似mysql数据库）：GET /\_cat/indices?v**

**GET /\_cat/indices?v**



### 4.1.2 创建索引

**PUT /索引名**

**参数可选：指定分片及副本，默认分片为3，副本为2。**

**例如**

PUT index1

或者

PUT /index1

例如：

{

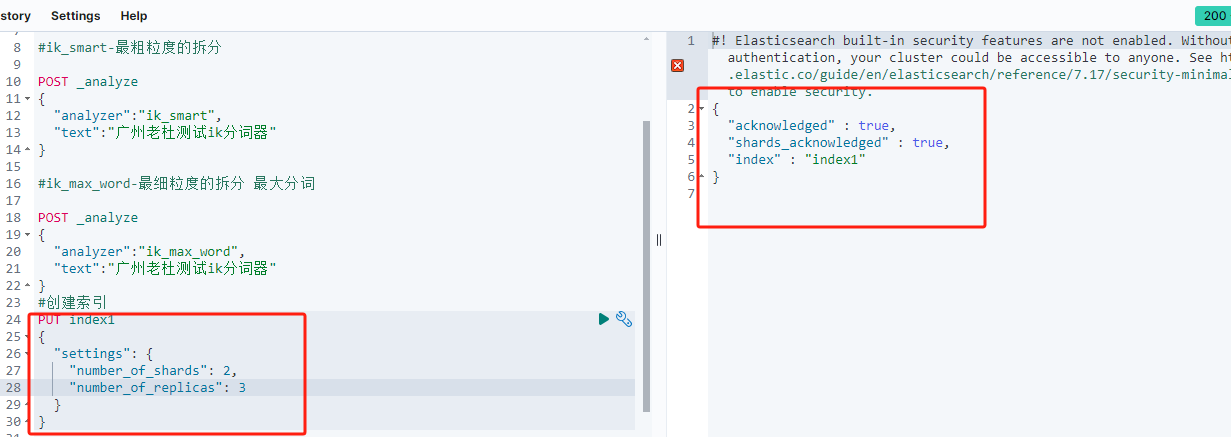
"settings": {

"number\_of\_shards": 2,

"number\_of\_replicas": 3

}

}



### 4.1.3 查看索引基本信息

**GET /索引名**

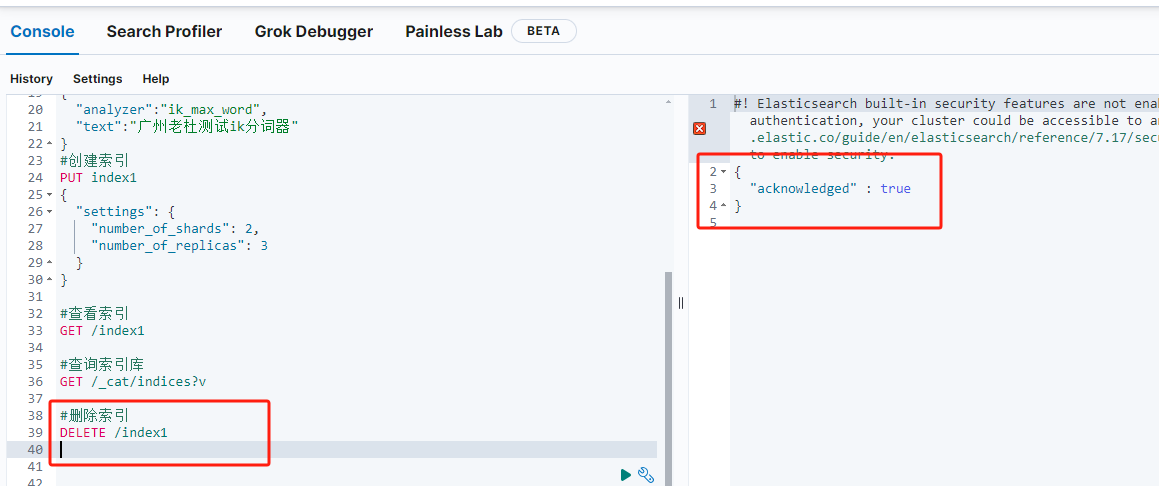
GET /index1



### 4.1.4 删除索引

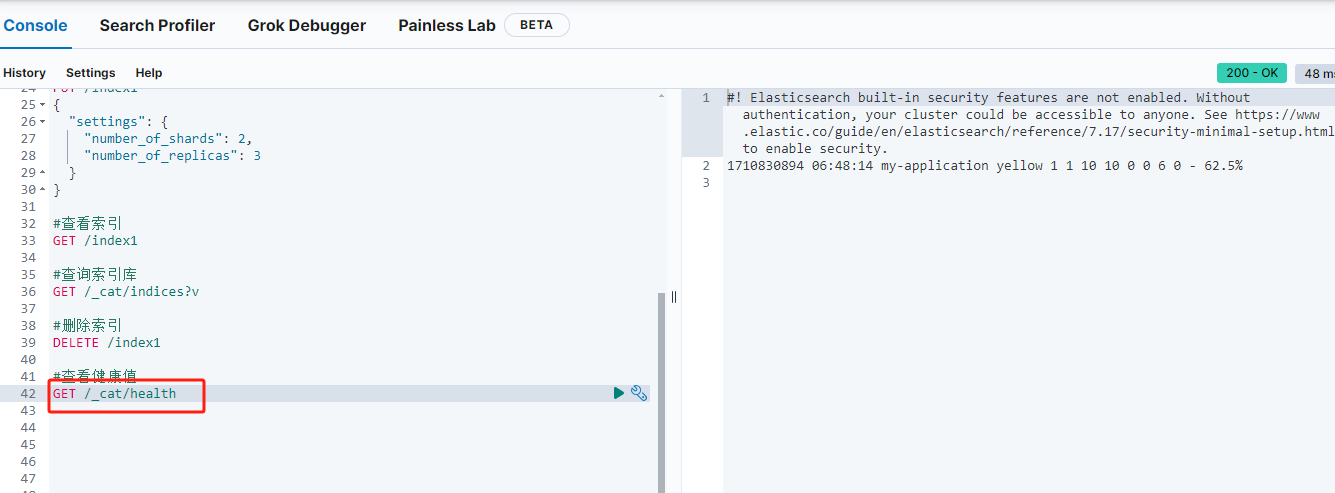
**DELETE /索引库名**

DELETE /index1



### 4.1.4 查看健康值

GET /\_cat/health



## 4.2映射配置（\_mapping）

**索引有了，接下来肯定是添加数据。但是，在添加数据之前必须定义映射。**

**什么是映射？**

**映射是定义文档的过程，文档包含哪些字段，这些字段是否保存，是否索引，是否分词等**

**只有配置清楚，Elasticsearch才会帮我们进行索引库的创建（不一定）**

### 4.2.1 创建映射字段

**语法：**

**PUT /索引库名/\_mapping/类型名称 或 索引库名/类型名称/\_mapping**

**{**

**"properties": {**

**"字段名": {**

**"type": "类型",**

**"index": true，**

**"store": true，**

**"analyzer": "分词器"**

**}**

**}**

**}**

**类型名称：就是前面将的type的概念，类似于数据库中的表名：任意填写，下面指定许多属性**

**Properties下**字段名：类似于列名，properties下可以指定许多字段。

每个字段可以有很多属性。例如：

type：类型，String（text keyword） Numeric（long integer float double） date boolean

index：是否索引，默认为true

store：是否存储，默认为false，即使为false也会存储到\_source中，如果为true则会额外存储一份

analyzer：分词器，这里使用ik分词器：ik\_max\_word或者ik\_smart

例如：

给index1这个索引库添加了一个名为goods的类型，并且在类型中设置了4个字段：  
title：商品标题  
subtitle: 商品子标题  
images：商品图片  
price：商品价格

PUT /index1/\_mapping/goods?include\_type\_name=true

{

"properties": {

"title": {

"type": "text",

"analyzer": "ik\_max\_word"

},

"subtitle": {

"type": "text",

"index": true,

"store": true,

"analyzer": "ik\_max\_word"

},

"images": {

"type": "keyword",

"index": "false"

},

"price": {

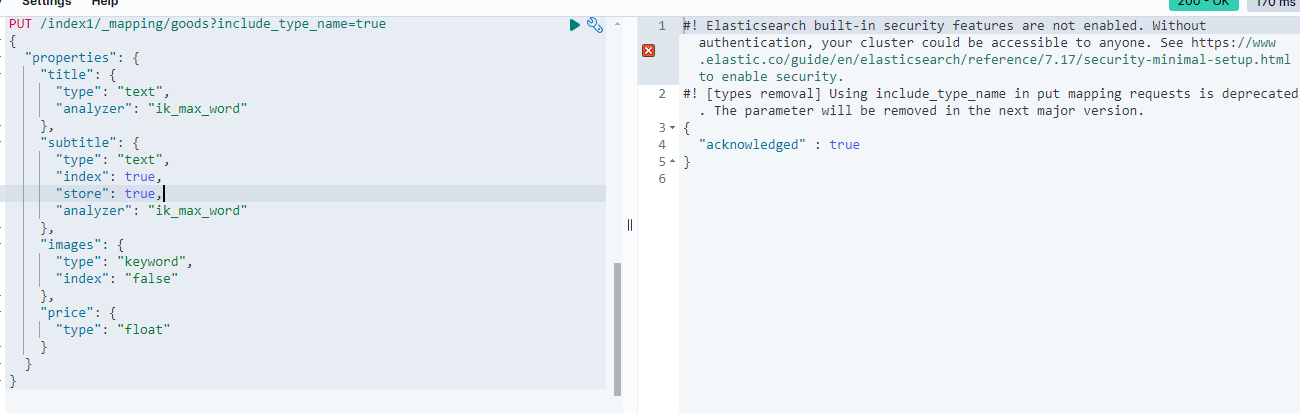
"type": "float"

}

}

}

如下图：



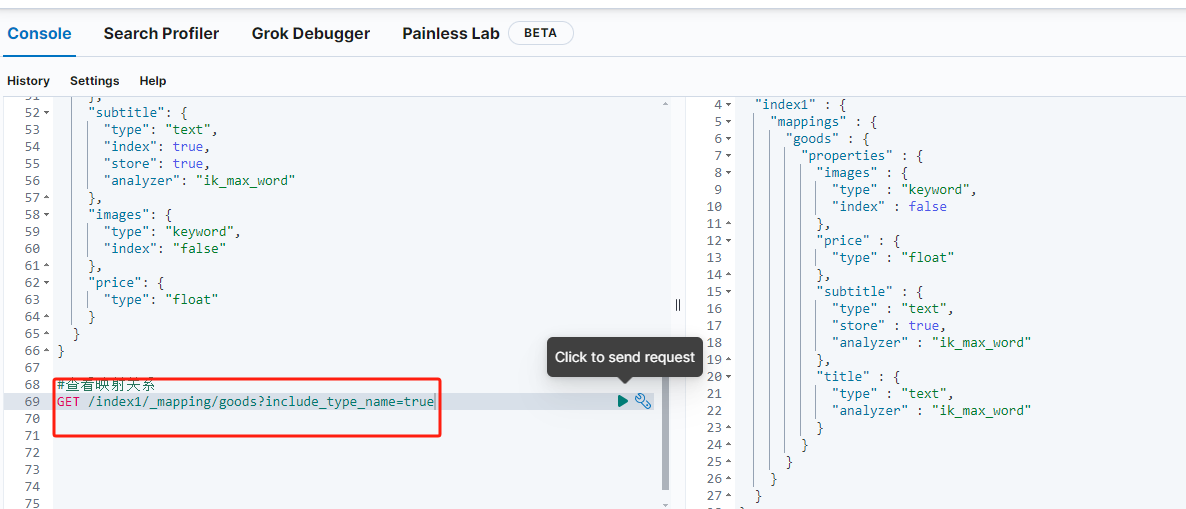
### 4.2.2 查看映射关系

**语法：**

**GET /索引库名/\_mapping/类型名**

例如：

GET /index1/\_mapping/goods?include\_type\_name=true



### 4.2.3 一次创建索引库和类型（常用）

刚才 的案例中我们是把创建索引库和类型分开来做，其实也可以在创建索引库的同时，直接制定索引库中的类型，基本语法：

put /索引库名

{

"settings":{

"索引库属性名":"索引库属性值"

},

"mappings":{

"类型名":{

"properties":{

"字段名":{

"映射属性名":"映射属性值"

}

}

}

}

}

例如：

创建索引为index2，类型为goods

PUT /index2?include\_type\_name=true

{

"settings": {},

"mappings": {

"goods": {

"properties": {

"title": {

"type": "text",

"analyzer": "ik\_max\_word"

}

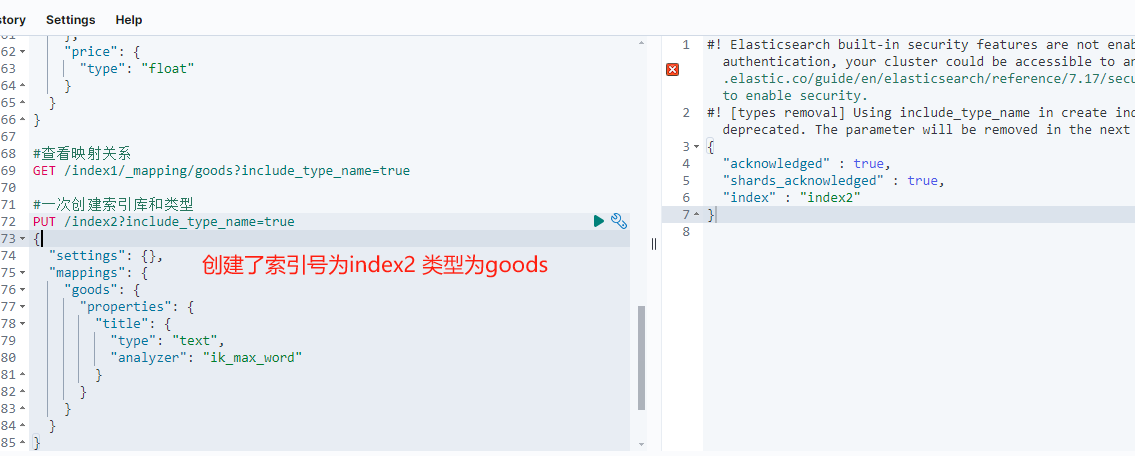
}

}

}

}

如下图：



## 4.3文档操作

文档，即索引库中某个类型下的数据，会根据规则创建索引，将来用来搜索。可以类比做数据库中的每一行数据。

**在ES7.x版本之后，一个index中只能创建一个type，默认已经创建，默认名称为\_doc**

### 4.3.1 创建文档（添加数据）

**创建文档就是给表中添加数据**

**如果我们想要自己新增的时候指定id，可以这么做：**

**POST /索引库名/\_doc/id值  
{  
    ...  
}**

创建文档，数据存储，将document对象添加到type中，也就是给（type）表中加数据

下列实例是给index1的默认type （\_doc）中加入了两条数据

POST /index1/\_doc/101

{

"title": "小米手机",

"images": "http://xiaomi.com/xiaomi.igp",

"price":2999

}

#增加第二条数据

POST /index1/\_doc/102

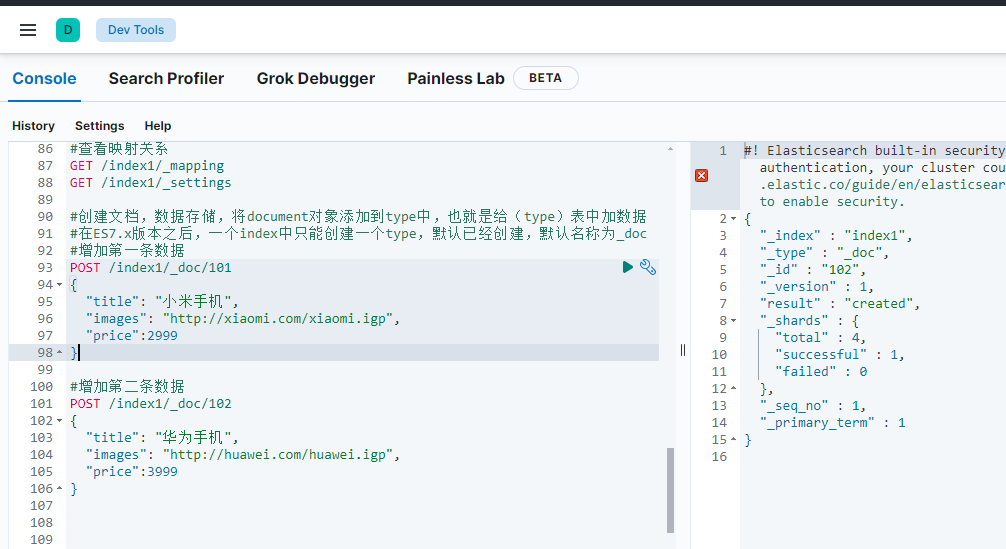
{

"title": "华为手机",

"images": "http://huawei.com/huawei.igp",

"price":3999

}



### 4.3.2 修改文档（修改整个数据）

**修改文档和添加文档语法和命令一致，如果ID不存在，就是创建，如果存在就是更新。在执行后的结果中如果是created就是创建，如果是updated就是更新。**

**语法：**

**POST /索引库名/\_doc/id值  
{  
    ...  
}**

**例如id是101的商品：**

POST /index1/\_doc/101

{

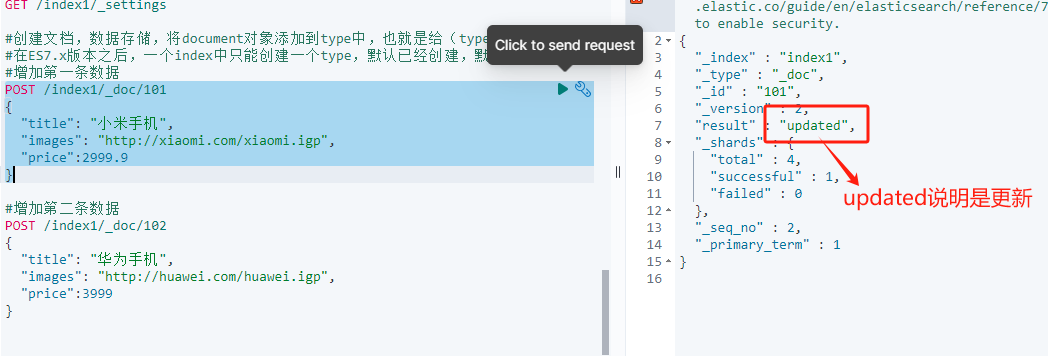
"title": "小米手机",

"images": "http://xiaomi.com/xiaomi.igp",

"price":2999.9

}

如下图



### 4.3.4 更新文档（查询某个字段）

**语法：**  
**POST /索引库名/\_update/id值  
{  
    doc:{  
        "属性":"值"  
    }  
}**

**例如：修改id为102的数据的price**

POST /index1/\_update/102

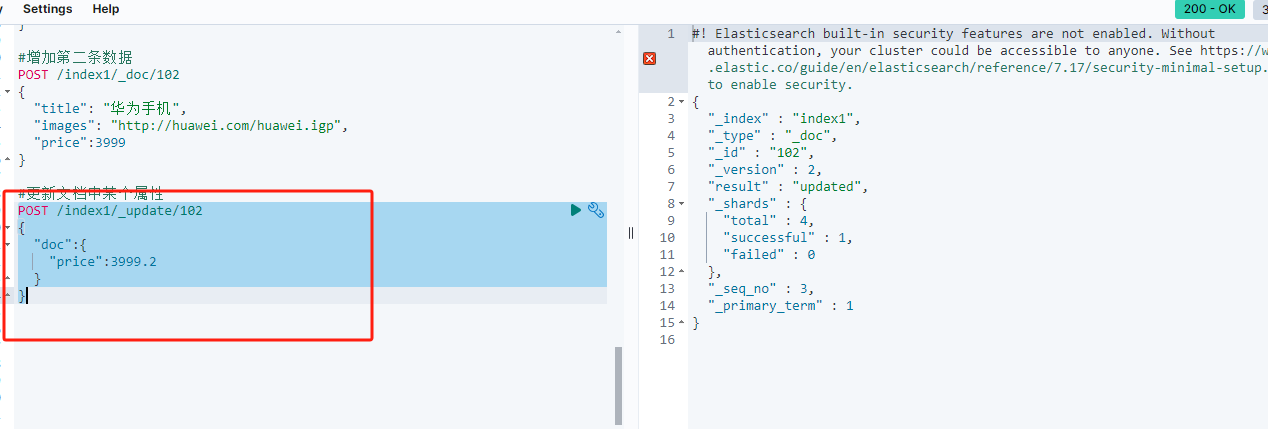
{

"doc":{

"price":3999.2

}

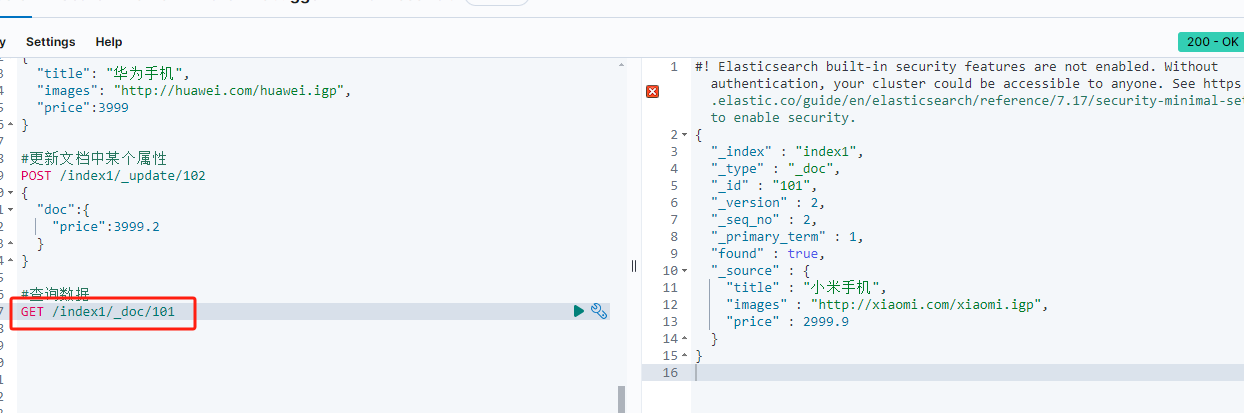
}



### 4.3.5 查看文档

**语法：**  
**GET /索引库名/id值**例如查询id为101的数据：

GET /index1/\_doc/101

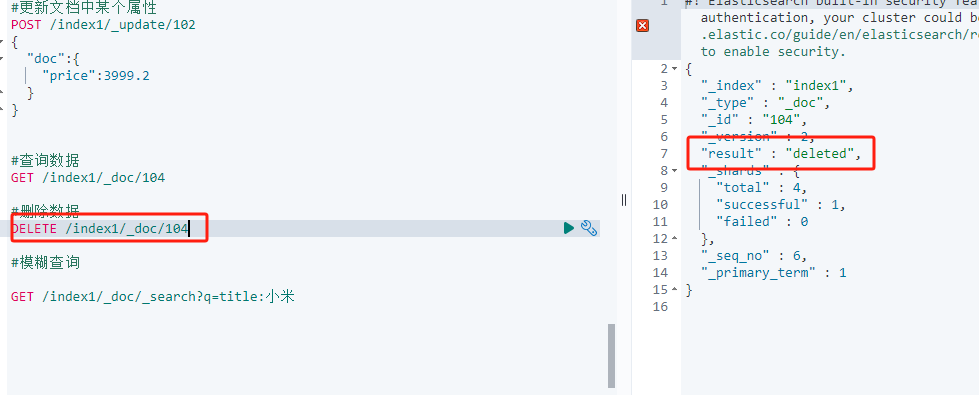


### 4.3.6 删除文档

**删除使用DELETE请求，同样，需要根据id进行删除：**

**语法：DELETE /索引库名/\_doc/id值**  
**示例：DELETE /index1/\_doc/103**

**DELETE /index1/\_doc/104**



## 4.4基本查询操作

基本语法

POST /索引库名/\_search

{

"query":{

"查询类型":{

"查询条件":"查询条件值"

}

}

}

这里的query代表一个查询对象，里面可以有不同的查询属性

查询类型：

例如：match\_all， match，term ， range 等等

查询条件：查询条件会根据类型的不同，写法也有差异，在后面

下边就以index1 为例查询创建文档的数据

### 4.4.1 查询所有（match\_all)

POST /index1/\_search

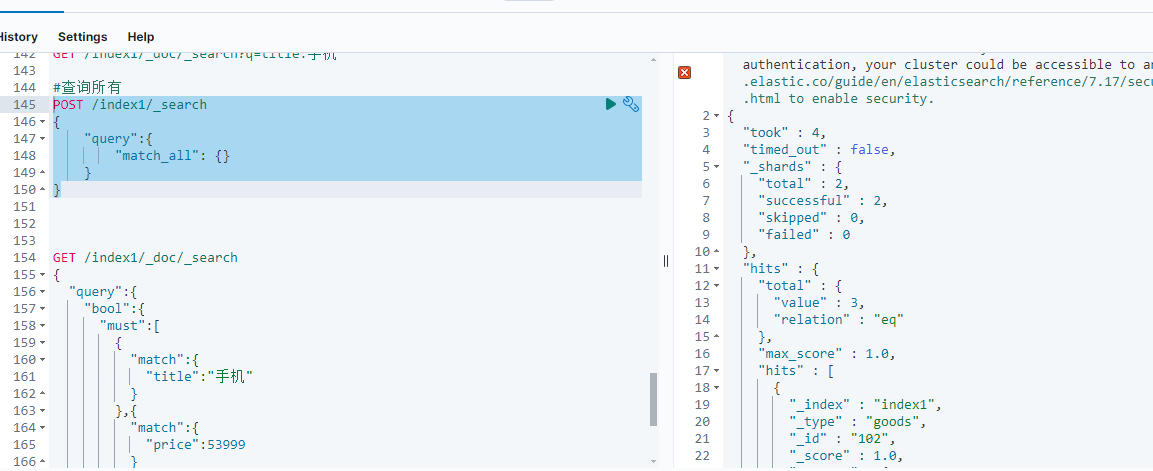
{

"query":{

"match\_all": {}

}

}



**结果中的数据解释**

took：查询花费时间，单位是毫秒

time\_out：是否超时

\_shards：分片信息

hits：搜索结果总览对象

total：搜索到的总条数

max\_score：所有结果中文档得分的最高分

hits：搜索结果的文档对象数组，每个元素是一条搜索到的文档信息

\_index：索引库

\_type：文档类型

\_id：文档id

\_score：文档得分

\_source：文档的源数据

### 4.4.2 匹配所有（match）

match类型查询，会把查询条件进行分词，然后进行查询,多个词条之间是or的关系，不仅会查询到电视，而且与小米相关的都会查询到，多个词之间是or的关系

POST /index1/\_search

{

"query": {

"match": {

"title": "小米电视"

}

}

}



### 4.4.3 多字段查询（multi\_match）

multi\_match与match类似，不同的是它可以在多个字段中查询

在title字段和subtitle字段中查询小米这个词

POST /index1/\_search

{

"query": {

"multi\_match": {

"query": "小米电视",

"fields": ["title","subtitle"]

}

}

}



### 4.4.4 词条匹配(term)

term 查询被用于精确值 匹配，这些精确值可能是数字、时间、布尔或者那些未分词的字符串

POST /index1/\_search

{

"query":{

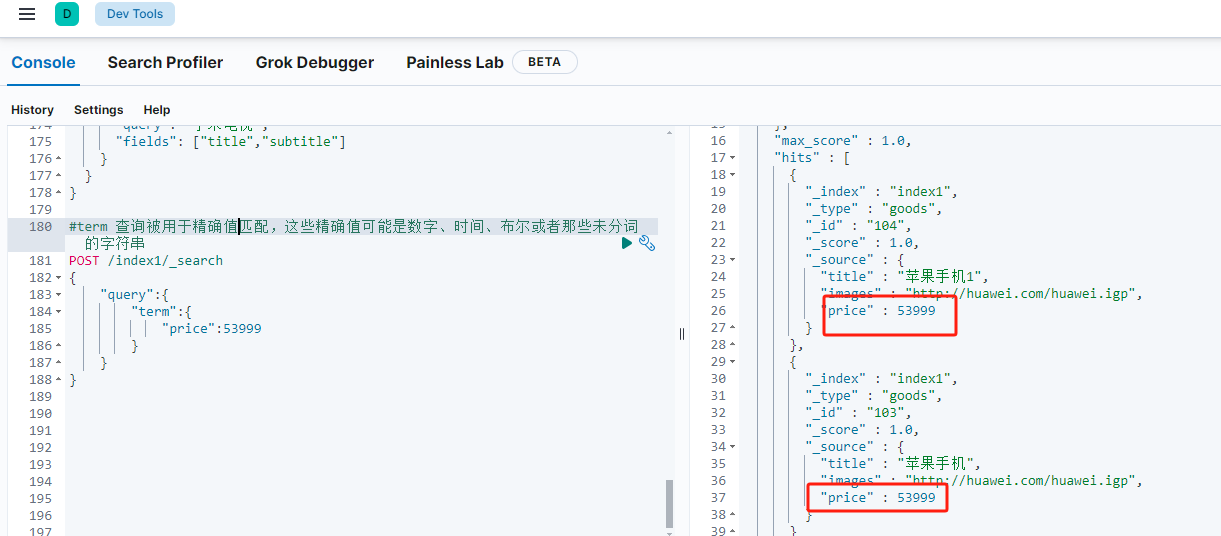
"term":{

"price":53999

}

}

}



### 4.4.5 多词条精确匹配(terms)

terms 查询和 term 查询一样，但它允许你指定多值进行匹配。如果这个字段包含了指定值中的任何一个值，那么这个文档满足条件，类似于mysql的in：

POST /index1/\_search

{

"query":{

"terms":{

"price":[3999,53999]

}

}

}



## 4.5逻辑查询操作

bool把各种其它查询通过must（与）、must\_not（非）、should（或）的方式进行组合

### 4.5.1 布尔组合（bool)

查找title中含有小米但不包含电视的数据

GET /index1/\_search

{

"query":{

"bool":{

"must": { "match": { "title": "小米" }},

"must\_not": { "match": { "title": "电视" }},

"should": { "match": { "title": "手机" }}

}

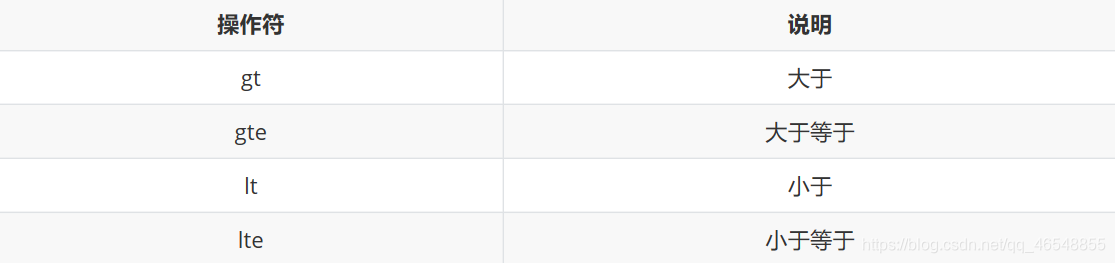
}

}



### 4.5.2 范围查询(range)

range 查询找出那些落在指定区间内的数字或者时间  
range查询允许以下字符：



POST /index1/\_search

{

"query":{

"range": {

"price": {

"gte": 3000,

"lt": 5000

}

}

}

}



### 4.5.3 模糊查询(fuzzy)

我们新增一个商品：

POST /index1/\_doc/108

{

"title":"apple手机",

"images":"http://image.leyou.com/12479122.jpg",

"price":5899.00

}

fuzzy自动将拼写错误的搜索文本，进行纠正，纠正以后去尝试匹配索引中的数据  
它允许用户搜索词条与实际词条出现偏差，但是偏差的编辑距离不得超过2：

POST /index1/\_search

{

"query": {

"fuzzy": {

"title": "appla"

}

}

}



## 4.6排序操作

### 4.6.1 单字节排序

sort 可以让我们按照不同的字段进行排序，并且通过order指定排序的方式

POST /index1/\_search

{

"query": {

"match\_all": {}

},

"sort": [

{"price": {"order": "desc"}}

]

}



https://blog.csdn.net/qq\_46548855/article/details/107386642

### 4.6.2 多字节排序

假定我们想要结合使用 price和 \_score（得分） 进行查询，并且匹配的结果首先按照价格排序，然后按照相关性得分排序：

#多字节排序，以price和\_score排序

POST /index1/\_search

{

"query":{

"match\_all":{}

},

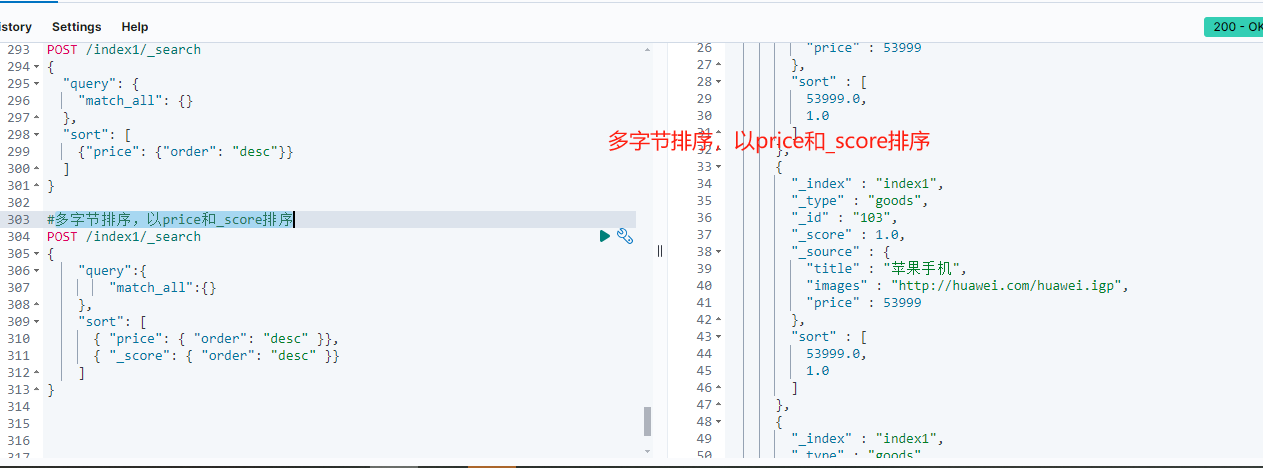
"sort": [

{ "price": { "order": "desc" }},

{ "\_score": { "order": "desc" }}

]

}



## 4.7高亮显示操作

elasticsearch中实现高亮的语法比较简单：

例如：

#高亮显示

POST /index1/\_search

{

"query": {

"match": {

"title": "手机"

}

},

"highlight": {

"pre\_tags": "<font color='pink'>",

"post\_tags": "</font>",

"fields": {

"title": {}

}

}

}

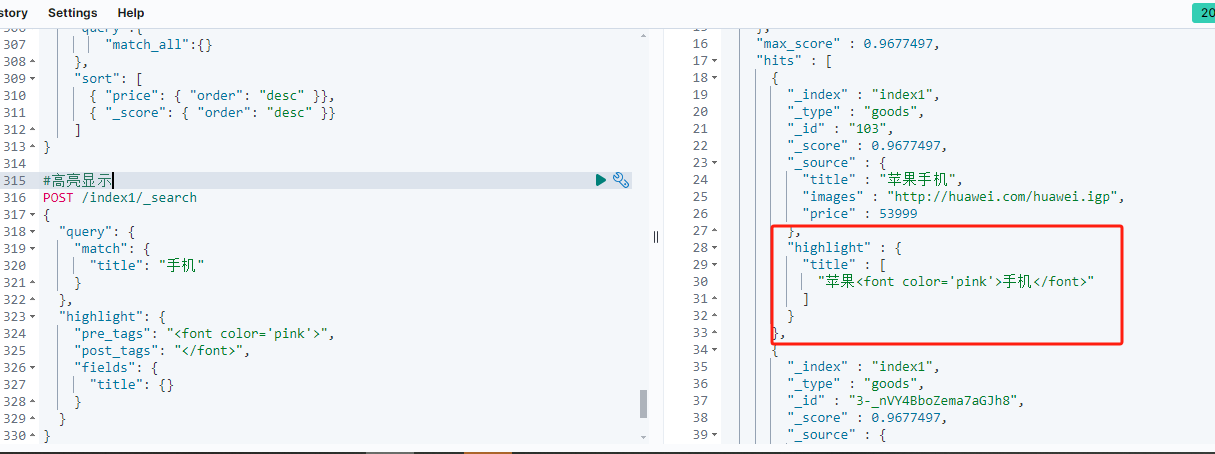
在使用match查询的同时，加上一个highlight属性：

pre\_tags：前置标签

post\_tags：后置标签

fields：需要高亮的字段

title：这里声明title字段需要高亮，后面可以为这个字段设置特有配置，也可以空



## 4.8分页操作

elasticsearch中实现分页的语法非常简单：

#分页操作

POST /index1/\_search

{

"query": {

"match\_all": {}

},

"size": 2,

"from": 0

}

**size:每页显示多少条**  
**from:当前页起始索引, int start = (pageNum - 1) \* size;**

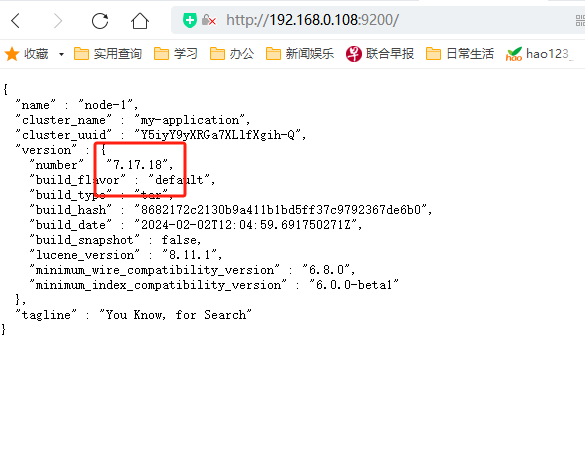


# SpringBoot整合Elasticsearch

## 5.1配置依赖

### 5.1.1 配置版本信息

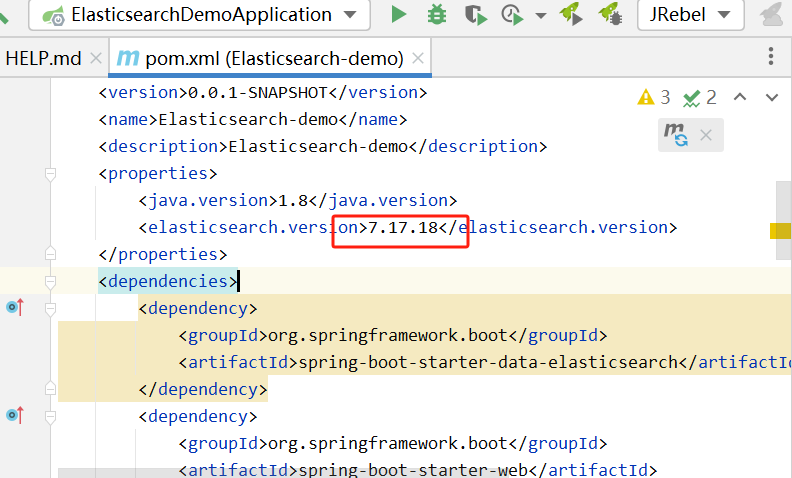
注意依赖版本和安装的版本一致，我安装的es版本是7.17.18，如下图查看



在pom.xml中，

<properties>  
 <java.version>1.8</java.version>  
 <elasticsearch.version>7.17.18</elasticsearch.version>  
</properties>

如下图：



### 5.1.2 导入es依赖

**1.High Level Java REST Client方式接入**

<dependency>  
 <groupId>org.elasticsearch</groupId>  
 <artifactId>elasticsearch</artifactId>  
 <version>${elasticsearch.version}</version>  
</dependency>  
*<!-- Java High Level REST Client -->*<dependency>  
 <groupId>org.elasticsearch.client</groupId> <artifactId>elasticsearch-rest-high-level-client</artifactId>  
 <version>${elasticsearch.version}</version>  
</dependency>

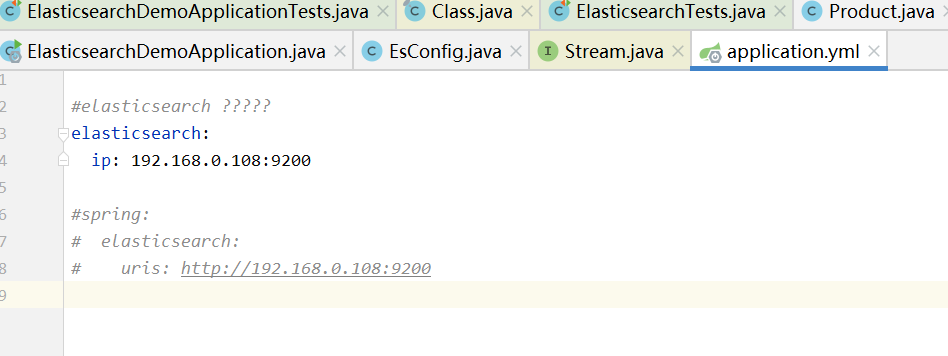
**2.或者**SpringBoot模版方式接入(不建议)

<dependency>  
 <groupId>org.springframework.boot</groupId>  
<artifactId>spring-boot-starter-data-elasticsearch</artifactId>  
</dependency>

直接模版方式的java api调用方式，后续官方会不支持了，不建议使用，要使用Java High Level REST Client来代替，Elasticsearch 8.0版本后直接移除，想想还是换人家建议的使用方式吧，免得以后更新换代还得做迁移

### 5.1.3 配置yml

elasticsearch:  
 ip: 192.168.0.108:9200



如果SpringBoot模版方式接入接入，建议配置

*spring:  
 elasticsearch:  
 uris: http://192.168.0.108:9200*

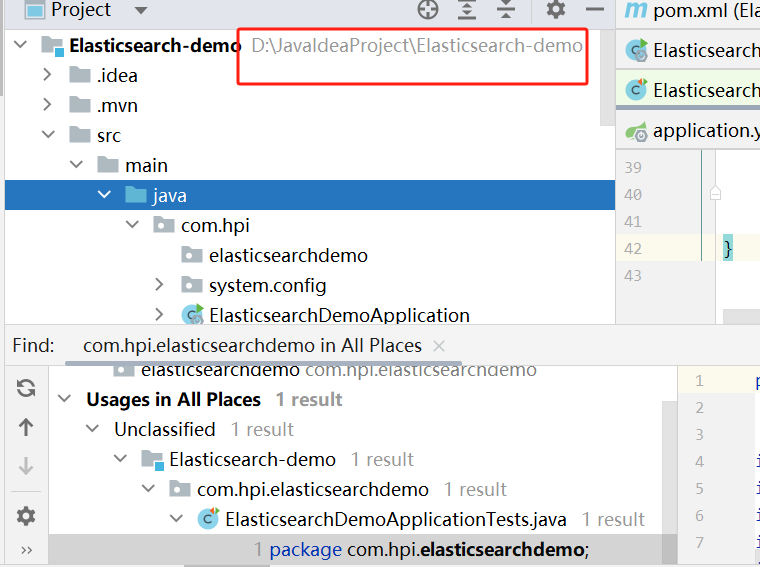
## 5.2创建并编写配置类

创建ElasticSearchConfig配置类

package com.hpi.system.config;  
import org.apache.http.HttpHost;  
import org.elasticsearch.client.RestClient;  
import org.elasticsearch.client.RestClientBuilder;  
import org.elasticsearch.client.RestHighLevelClient;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.beans.factory.annotation.Value;  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;  
  
*/\*\*  
 \* 类名称：ElasticSearchConfig  
 \* 描述: ElasticSearch配置类  
 \* 创建人：duhj  
 \* 创建时间：2024/3/21 10:15  
 \*/*@Configuration  
public class ElasticSearchConfig {  
  
 @Value("${elasticsearch.ip}")  
 String ipPort;  
 *// 注册 rest高级客户端* @Bean  
 public RestHighLevelClient restHighLevelClient(){  
 RestHighLevelClient client = new RestHighLevelClient(  
 */\*RestClient.builder(  
 new HttpHost("127.0.0.1",9200,"http")  
 )\*/* RestClient.*builder*(  
 makeHttpHost(ipPort)  
 )  
 *// RestClient.builder(makeHttpHost(ipPort));* );  
 return client;  
 }  
   
 private HttpHost makeHttpHost(String s) {  
 String[] address = s.split(":");  
 String ip = address[0];  
 int port = Integer.*parseInt*(address[1]);  
  
 return new HttpHost(ip, port, "http");  
 }  
  
}

## 5.3测试

我们的测试工程为：



### 5.3.1 创建索引

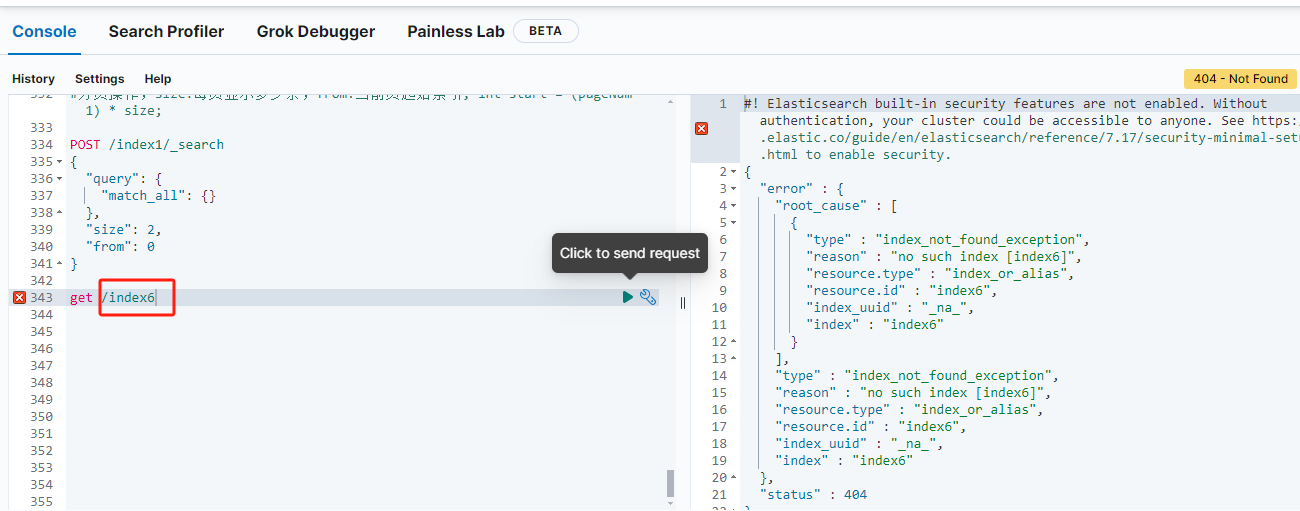
我们创建一个测试类测试，测试类代码如下：

package com.hpi.elasticsearchdemo;  
  
  
import com.hpi.ElasticsearchDemoApplication;  
import org.elasticsearch.client.RequestOptions;  
import org.elasticsearch.client.RestHighLevelClient;  
import org.elasticsearch.client.indices.CreateIndexRequest;  
import org.elasticsearch.client.indices.CreateIndexResponse;  
import org.junit.Test;  
import org.junit.runner.RunWith;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.boot.test.context.SpringBootTest;  
import org.springframework.test.context.junit4.SpringRunner;  
  
import java.io.IOException;  
  
@RunWith(SpringRunner.class)  
@SpringBootTest(classes = ElasticsearchDemoApplication.class )  
public class ElasticsearchDemoApplicationTests {  
  
 @Autowired  
 private RestHighLevelClient resthighLevelClient;  
 @Test  
 public void contextLoads() {  
 System.*out*.println(resthighLevelClient);  
 }  
  
 @Test  
 public void craateIndex() throws IOException {  
 CreateIndexRequest createIndexRequest =  
 new CreateIndexRequest("index6");  
 CreateIndexResponse createIndexResponse = resthighLevelClient.indices().create(createIndexRequest, RequestOptions.*DEFAULT*);  
 System.*out*.println(createIndexResponse);  
 }  
  
}

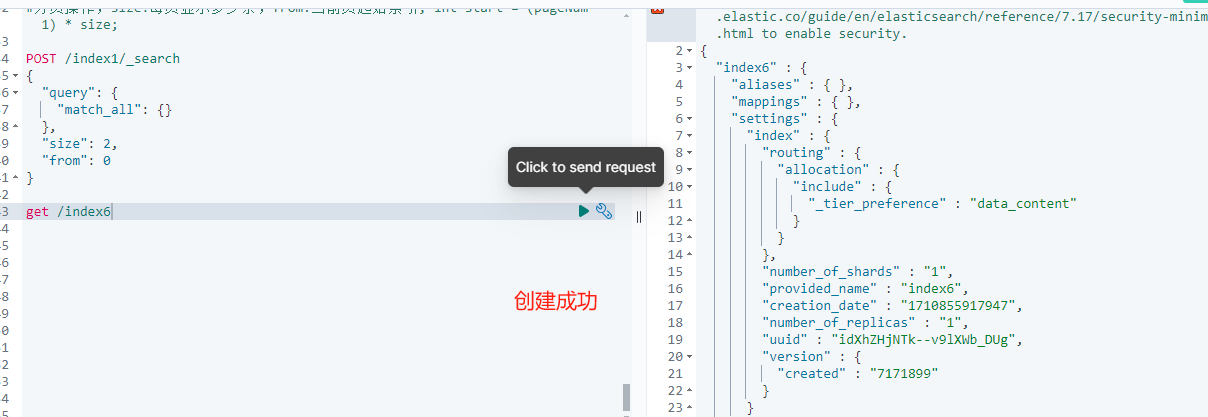
创建索引

public void craateIndex() throws IOException {  
 CreateIndexRequest createIndexRequest =  
 new CreateIndexRequest("index6");  
 CreateIndexResponse createIndexResponse = resthighLevelClient.indices().create(createIndexRequest, RequestOptions.*DEFAULT*);  
 System.*out*.println(createIndexResponse);  
}

创建之前：



创建之后：



### 5.3.2 删除索引

@Test  
public void deleteIndex() throws IOException {  
 Object result = "";  
 try {  
 *// 新建删除索引请求对象* DeleteIndexRequest request = new DeleteIndexRequest("index6");  
 *// 执行删除索引* AcknowledgedResponse acknowledgedResponse = resthighLevelClient.indices().delete(request, RequestOptions.*DEFAULT*);  
 *// 判断是否删除成功* boolean siDeleted = acknowledgedResponse.isAcknowledged();  
 System.*out*.println(siDeleted);  
 *// 根据具体业务逻辑返回不同结果，这里为了方便直接将结果返回* result = siDeleted;  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
}

### 5.3.3 获取索引

@Test  
public void getIndex() throws IOException {  
 try {  
 GetIndexRequest request = new GetIndexRequest("index6");  
 boolean exists = resthighLevelClient.indices().exists(request, RequestOptions.*DEFAULT*);  
 System.*out*.println(exists);*// 索引是否存在* resthighLevelClient.close();  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
}

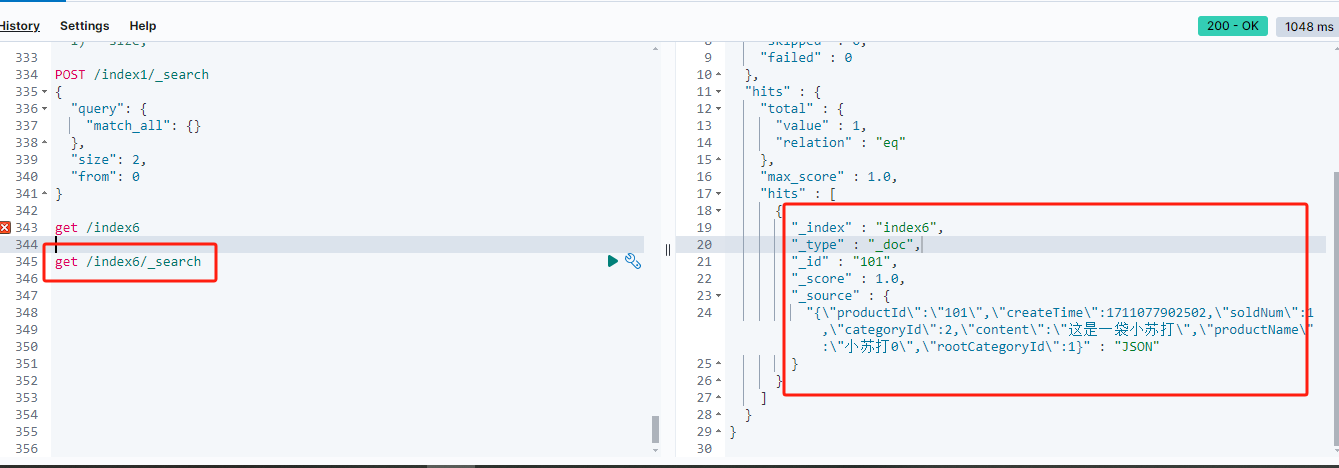
### 5.3.4 创建（修改）文档

给index6 中添加一个Product对象

@Test  
public void addDocumentToEs() throws IOException {  
 Product product = new Product();  
 product.setProductId("101");  
 product.setProductName("小苏打0");  
 product.setCreateTime(new Date());  
 product.setContent("这是一袋小苏打");  
 product.setSoldNum(1);  
 product.setRootCategoryId(1);  
 product.setCategoryId(2);  
 try {  
 IndexRequest indexRequest = new IndexRequest("index6");  
 indexRequest.id(product.getProductId()+"");  
 *//ObjectMapper objectMapper = new ObjectMapper();* indexRequest.source(JSON. *toJSONString*

(product), XContentType.*JSON*);  
 IndexResponse indexResponse = resthighLevelClient.index(indexRequest,RequestOptions.*DEFAULT*);  
 System.*out*.println(indexResponse);  
 resthighLevelClient.close();  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
}

添加完成后

对于已经存在的iD，如果执行上述方法，则修改文档

### 5.3.5 根据条件查询数据

public void searchData() throws IOException, NoSuchFieldException, IllegalAccessException, NoSuchMethodException, InvocationTargetException, IntrospectionException {  
 int pageSize=5;*//每页数据* int pageNum =1;*//页号  
 //用来封装检索条件* SearchRequest searchRequest = new SearchRequest("index6");  
  
 SearchSourceBuilder searchSourceBuilder = new SearchSourceBuilder();  
 *//查询条件  
 //searchSourceBuilder.query(QueryBuilders.matchAllQuery());//不带条件 //searchSourceBuilder.query(QueryBuilders.matchQuery("productName","电脑"));//一个字段查询  
 //searchSourceBuilder.query(QueryBuilders.matchQuery("电脑","productName"));//一个字段查询* searchSourceBuilder.query(QueryBuilders.*multiMatchQuery*("电脑","productName","content"));*//多个字段查询  
 //分页* searchSourceBuilder.from((pageNum -1)\*pageSize);*//从0页开始* searchSourceBuilder.size(pageSize);*//每页5行数据  
 //高亮显示：* HighlightBuilder highlightBuilder = new HighlightBuilder();  
 HighlightBuilder.Field field = new HighlightBuilder.Field("productName");*//高亮字段* highlightBuilder.field(field);  
 highlightBuilder.preTags("<font color='pink'>");*//标签前* highlightBuilder.postTags("</font>");*//标签后* searchSourceBuilder.highlighter(highlightBuilder);  
  
 searchRequest.source(searchSourceBuilder);  
 SearchResponse search = resthighLevelClient.search(searchRequest, RequestOptions.*DEFAULT*);  
 System.*out*.println("查询结果");  
 System.*out*.println(search.toString());  
 SearchHits hits = search.getHits();  
  
 hits.forEach(s ->{  
 System.*out*.println(s.getFields());  
 System.*out*.println(s.getId());  
 });  
 List<Product> products = new ArrayList<Product>();  
 Iterator<SearchHit> it =hits.iterator();  
 *//处理结果（获取的数据进行封装）* while (it.hasNext()){  
 SearchHit searchHit = it.next();  
 String sourceAsString = searchHit.getSourceAsString();*//json字符串* System.*out*.println(sourceAsString);  
  
 Product product=JSON.*parseObject*(sourceAsString,Product.class);  
 Map<String, HighlightField> highlightFields = searchHit.getHighlightFields();  
 if (highlightFields!=null){  
 for (Map.Entry< String,HighlightField> entry : highlightFields.entrySet()) {*//便利获取高亮字段* HighlightField highlightField=entry.getValue();  
 if (highlightField!=null){  
 Text[] fragments = highlightField.fragments();*//获取高亮显示的值对象* if (fragments!=null && fragments.length>0){  
 String fieldValue=Arrays.*toString*(fragments);*//转换为字符串* System.*out*.println("s=="+fieldValue);  
 Class<Product> cls = Product.class;  
 System.*out*.println(entry.getKey() );  
 String fieldName =entry.getKey();  
 *//获取属性,根据获取高亮字段对应设置商品的高亮字段值  
 //方式1：通过Field操作属性：* Field nameField = cls.getDeclaredField(fieldName);  
 nameField.setAccessible(true);  
 nameField.set(product, fieldValue);*//给productName字段重新设值（高亮的值）  
  
 //方式2：通过属性名动态给对象设值  
 /\*PropertyDescriptor namePd = new PropertyDescriptor(fieldName, cls);  
 Method writeMethod = namePd.getWriteMethod();  
 writeMethod.invoke(product, fieldValue);\*/  
  
 //方式三：通过set方法：  
 /\*fieldName=entry.getKey().substring(0,1).toUpperCase()+ entry.getKey().substring(1);  
 System.out.println("商品Name1111："+fieldName);  
 Method setName = cls.getMethod("set"+fieldName, String.class);  
 setName.invoke(product, fieldValue);//通过反射机制把获取的高亮值set到product中\*/* System.*out*.println("商品转换后高亮显示的值："+product.getProductName());  
 System.*out*.println("商品转换后高亮显示的值111："+product.getContent());  
 }  
  
 }  
  
 }  
 }  
  
 products.add(product);  
 }  
 }