**畅购电商系统开发**

**第5天**

传智播客.黑马程序员.深圳

# 学习目标

* Elasticsearch安装

docker安装Elasticsearch

系统参数问题

跨域操作

* IK分词器配置
* Kibana的使用

Kibana->DSL语句操作->Elasticsearch

* ES导入商品搜索数据

Sku数据导入到Elasticsearch

Map数据类型->Object

* 关键词搜索 能够实现搜索流程代码的编写
* 分类统计搜索

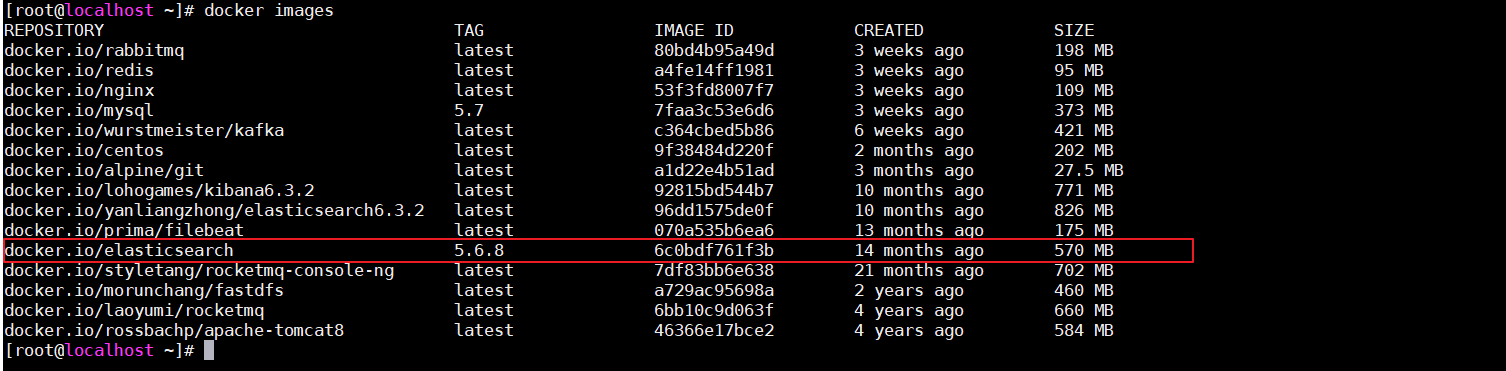
# Elasticsearch 安装

我们之前已经使用过elasticsearch了，这里不再对它进行介绍了，直接下载安装，本章节将采用Docker安装，不过在市面上还有很多采用linxu安装，关于linux安装，已经提供了安装手册，这里就不讲了。

## docker镜像下载

docker pull elasticsearch:5.6.8

注意：由于镜像有570MB，所以提供的虚拟机里已经下载好了该镜像，如下图：

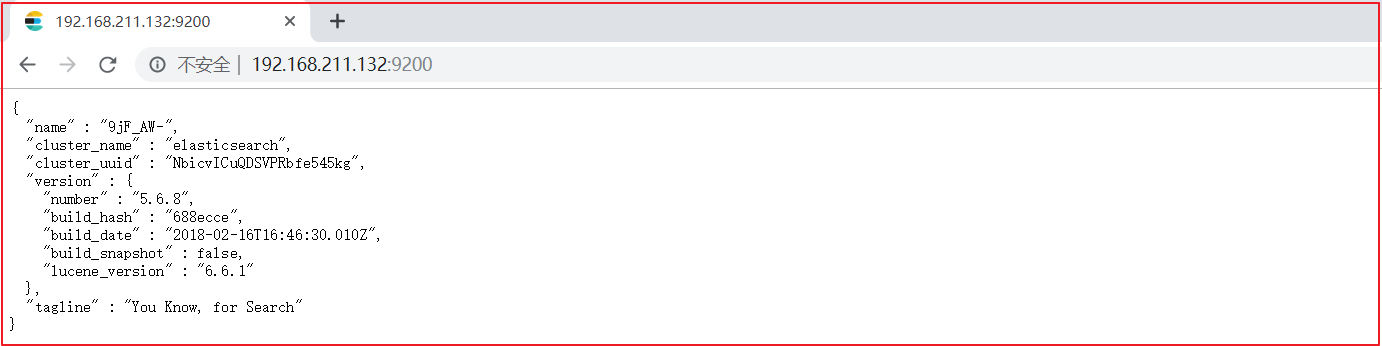


## 安装es容器

docker run -di --name=changgou\_elasticsearch -p 9200:9200 -p 9300:9300 elasticsearch:5.6.8

9200端口(Web管理平台端口) 9300(服务默认端口)

浏览器输入地址访问：<http://192.168.211.132:9200>



## 开启远程连接

上面完成安装后，es并不能正常使用，elasticsearch从5版本以后默认不开启远程连接，程序直接连接会报如下错误：

failed to load elasticsearch nodes : org.elasticsearch.client.transport.NoNodeAvailableException: None of the configured nodes are available: [{#transport#-1}{5ttLpMhkRjKLkvoY7ltUWg}{192.168.211.132}{192.168.211.132:9300}]

我们需要修改es配置开启远程连接，代码如下：

登录容器

docker **exec** -it changgou\_elasticsearch /bin/bash

查看目录结构 输入: dir

root@07f22eb41bb5:/usr/share/elasticsearch# dir

NOTICE.txt README.textile bin config data lib logs modules plugins

进入config目录

cd config

查看文件

root@07f22eb41bb5:/usr/share/elasticsearch/config# ls

elasticsearch.yml log4j2.properties scripts

修改elasticsearch.yml文件

root@07f22eb41bb5:/usr/share/elasticsearch/config# vi elasticsearch.yml

bash: vi: command **not** found

vi命令无法识别，因为docker容器里面没有该命令，我们可以安装该编辑器。

安装vim编辑器

apt-get **update**

apt-get install **vim**

安装好了后，修改elasticsearch.yml配置，如下图：

**vi** elasticsearch.yml

修改如下图：



同时添加下面一行代码：

cluster.name: **my**-application

重启docker

**docker** restart changgou\_elasticsearch

## 系统参数配置

重启后发现重启启动失败了，这时什么原因呢？这与我们刚才修改的配置有关，因为elasticsearch在启动的时候会进行一些检查，比如最多打开的文件的个数以及虚拟内存区域数量等等，如果你放开了此配置，意味着需要打开更多的文件以及虚拟内存，所以我们还需要系统调优

修改vi /etc/security/limits.conf ，追加内容 (nofile是单个进程允许打开的最大文件个数 soft nofile 是软限制 hard nofile是硬限制 )

\* soft nofile 65536

\* hard nofile 65536

修改vi /etc/sysctl.conf，追加内容 (限制一个进程可以拥有的VMA(虚拟内存区域)的数量 )

**vm.max\_map\_count**=655360

执行下面命令 修改内核参数马上生效

**sysctl** -p

重新启动虚拟机，再次启动容器，发现已经可以启动并远程访问

**reboot**

## 跨域配置

修改elasticsearch/config下的配置文件：elasticsearch.yml，增加以下三句命令，并重启:

**http.cors.enabled**: true

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

**network.host**: 192.168.211.132

其中：

http.cors.enabled: true：此步为允许elasticsearch跨域访问，默认是false。  
http.cors.allow-origin: "\*"：表示跨域访问允许的域名地址（\*表示任意）。

重启

**docker** restart changgou\_elasticsearch

小提示：如果想让容器开启重启，可以执行下面命令

docker **update** --restart=always 容器名称或者容器id

# IK分词器安装

## 安装ik分词器

IK分词器下载地址<https://github.com/medcl/elasticsearch-analysis-ik/releases>

将ik分词器上传到服务器上，然后解压，并改名字为ik

**unzip** elasticsearch-analysis-ik-5.6.8.zip

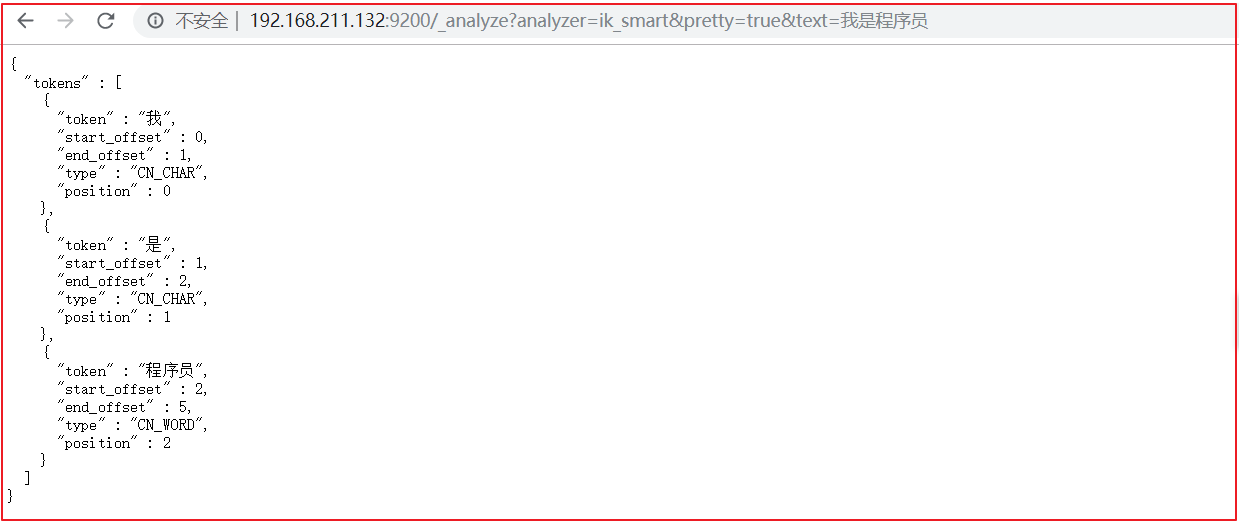
mv elasticsearch ik

将ik目录拷贝到docker容器的plugins目录下

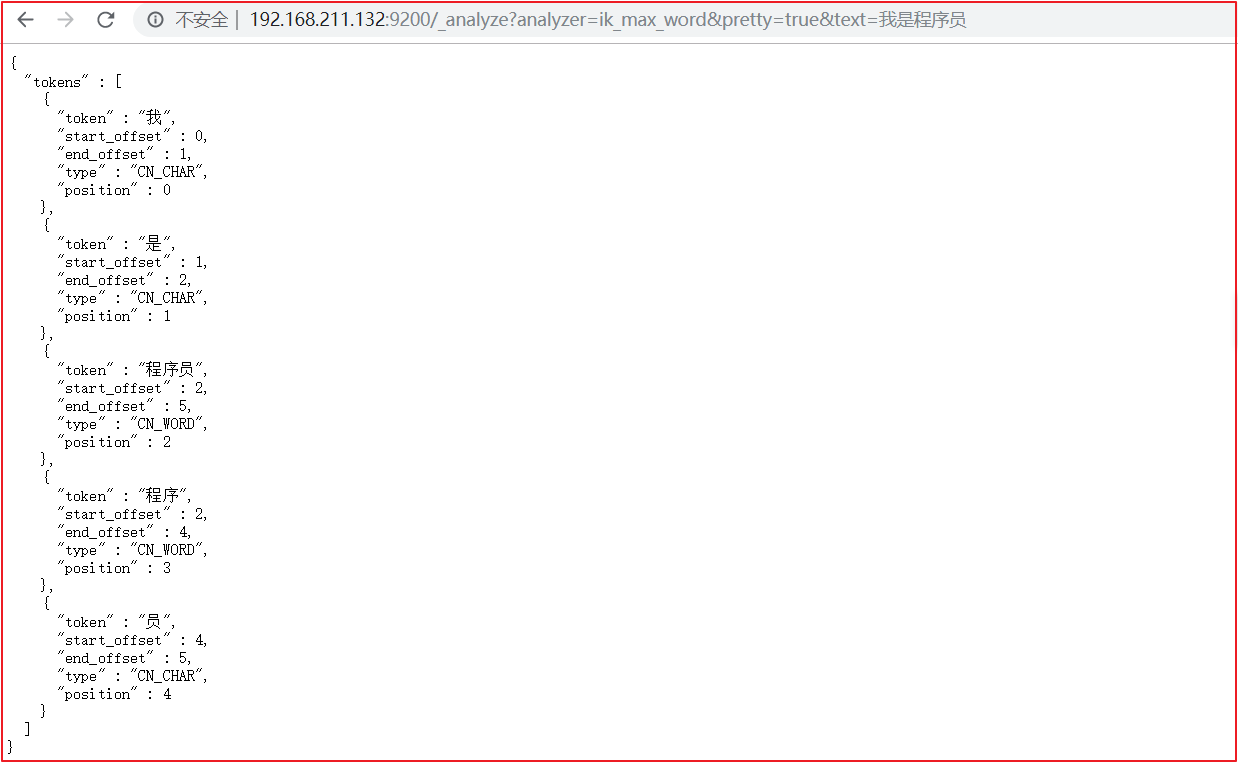
docker cp ./ik changgou\_elasticsearch:/usr/share/elasticsearch/plugins

## IK分词器测试

访问：<http://192.168.211.132:9200/_analyze?analyzer=ik_smart&pretty=true&text=我是程序员>



访问：<http://192.168.211.132:9200/_analyze?analyzer=ik_max_word&pretty=true&text=我是程序员>



# Kibana使用-掌握DSL语句

我们上面使用的是elasticsearch-head插件实现数据查找的，但是elasticsearch-head的功能比较单一，我们这里需要一个更专业的工具实现对日志的实时分析，也就是我们接下来要讲的kibana。

Kibana 是一款开源的数据分析和可视化平台，它是 Elastic Stack 成员之一，设计用于和 Elasticsearch 协作。您可以使用 Kibana 对 Elasticsearch 索引中的数据进行搜索、查看、交互操作。您可以很方便的利用图表、表格及地图对数据进行多元化的分析和呈现。

Kibana 可以使大数据通俗易懂。它很简单，基于浏览器的界面便于您快速创建和分享动态数据仪表板来追踪 Elasticsearch 的实时数据变化。

搭建 Kibana 非常简单。您可以分分钟完成 Kibana 的安装并开始探索 Elasticsearch 的索引数据 — 没有代码、不需要额外的基础设施。

## Kibana下载安装

我们项目中不再使用linux，直接使用Docker，所有这里就不演示在windows的下载安装了。

### 镜像下载

docker pull docker.io/kibana:5.6.8

为了节省时间，虚拟机中已经存在该版本的镜像了.

### 安装kibana容器

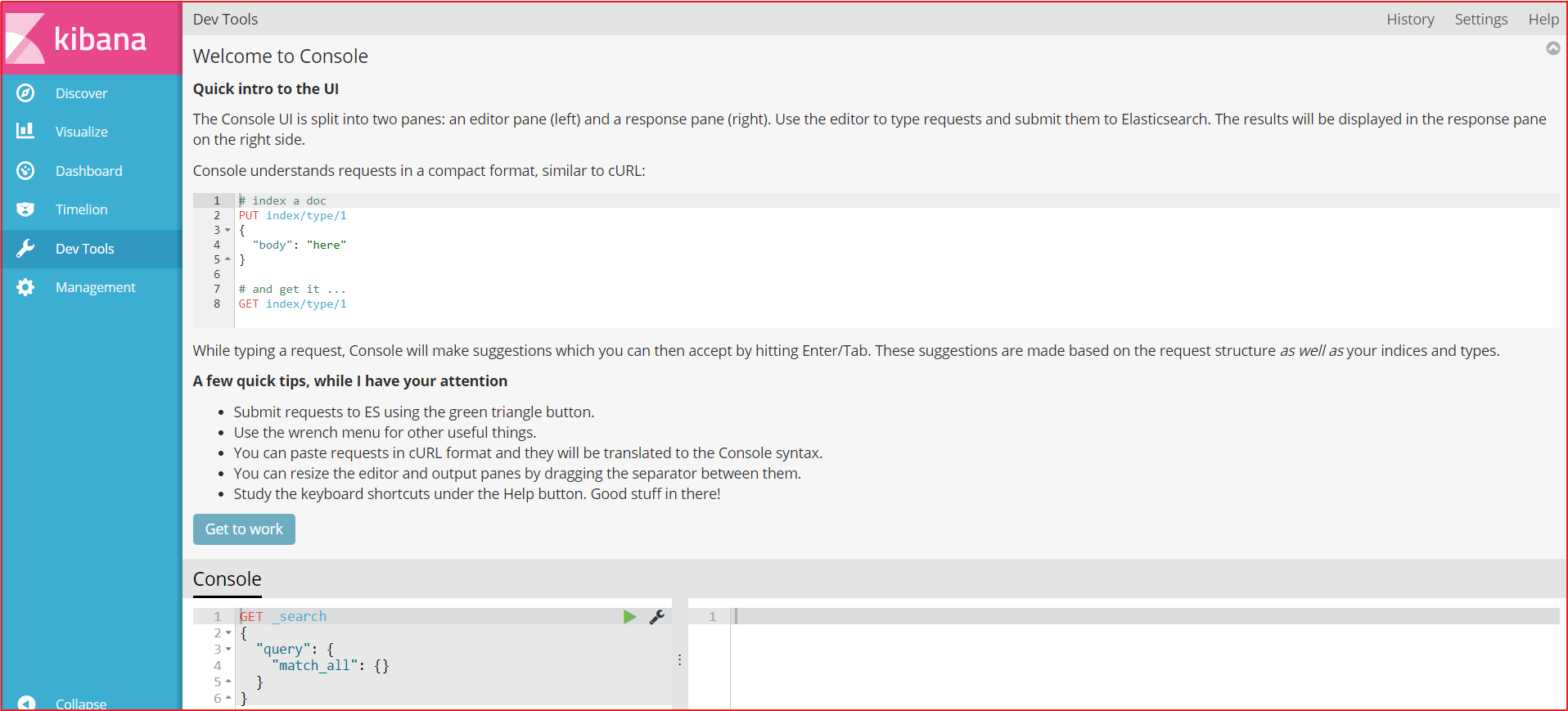
执行如下命令，开始安装kibana容器

docker run -**it** -d -e ELASTICSEARCH\_URL=**http**://192.168.211.132:9200 --name kibana --restart=always -p 5601:5601 kibana:5.6.8

* ELASTICSEARCH\_URL=http://192.168.211.132:9200：是指链接的ES地址
* restart=always:每次服务都会重启，也就是开启启动
* 5601:5601:端口号

### 访问测试

访问 <http://192.168.211.132:5601> 如下：

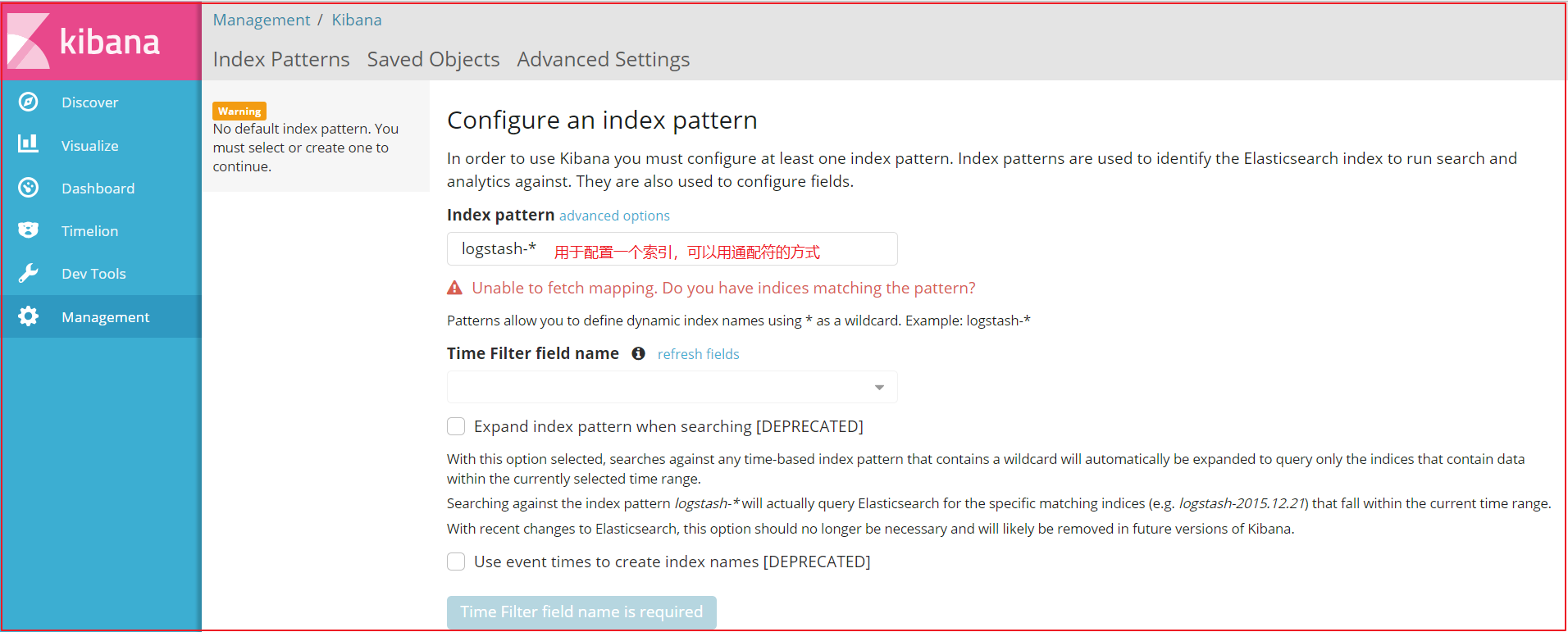


## Kibana使用

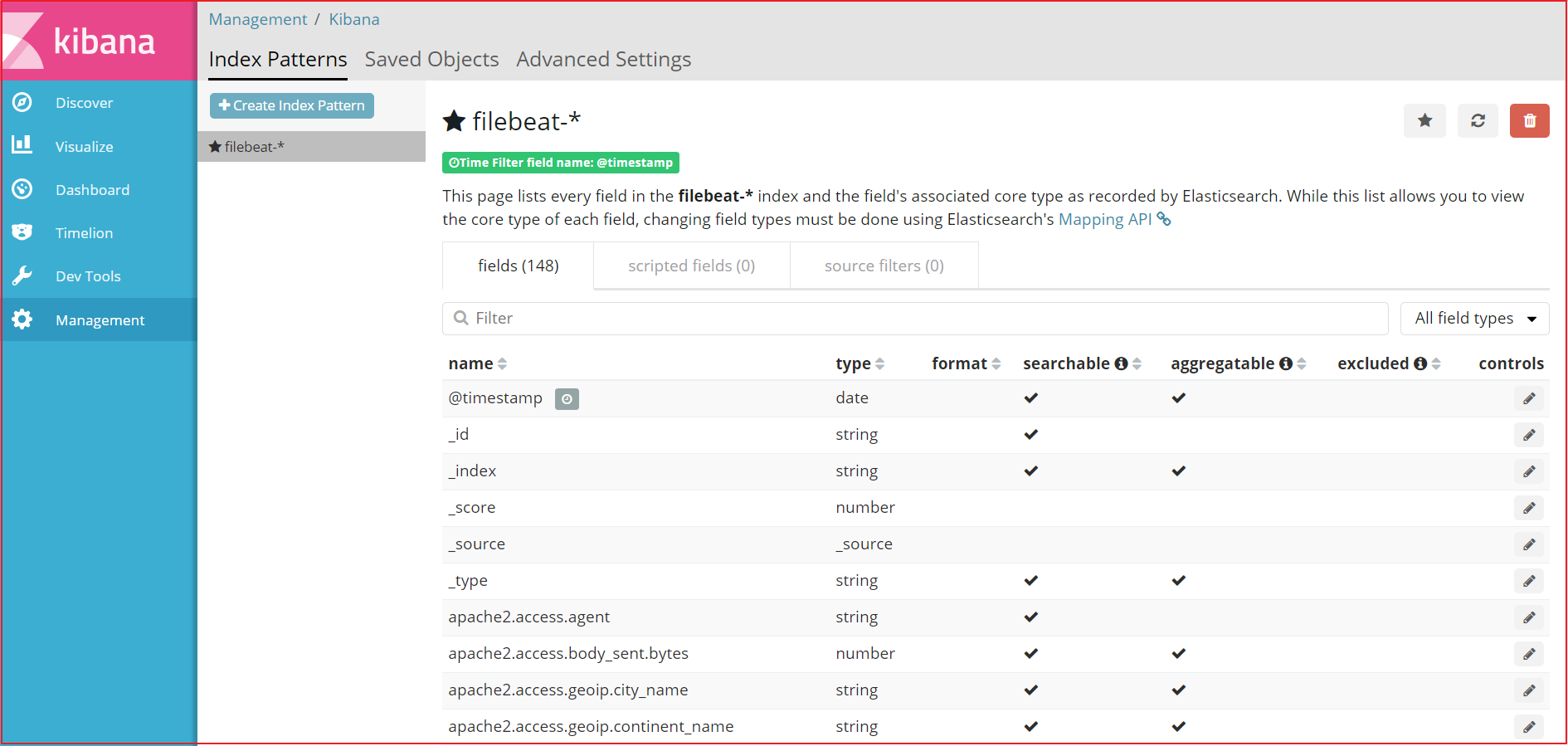
### 配置索引

要使用Kibana，您必须至少配置一个索引。索引用于标识Elasticsearch索引以运行搜索和分析。它们还用于配置字段。**(配置时如果要使用elasticSearch中的索引，就取相同的名字，否则不要取索引中已有的名字)**

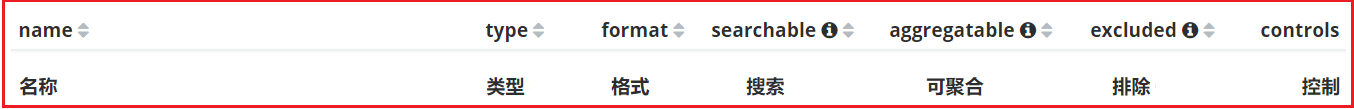
**因为最终和索引中名字相同，连上elasticsearch，还是操作的es中的库，影响数据**



我们修改索引名称的匹配方式即可，下面2个选项不用勾选。点击create，会展示出当前配置的索引的域信息，如下图：

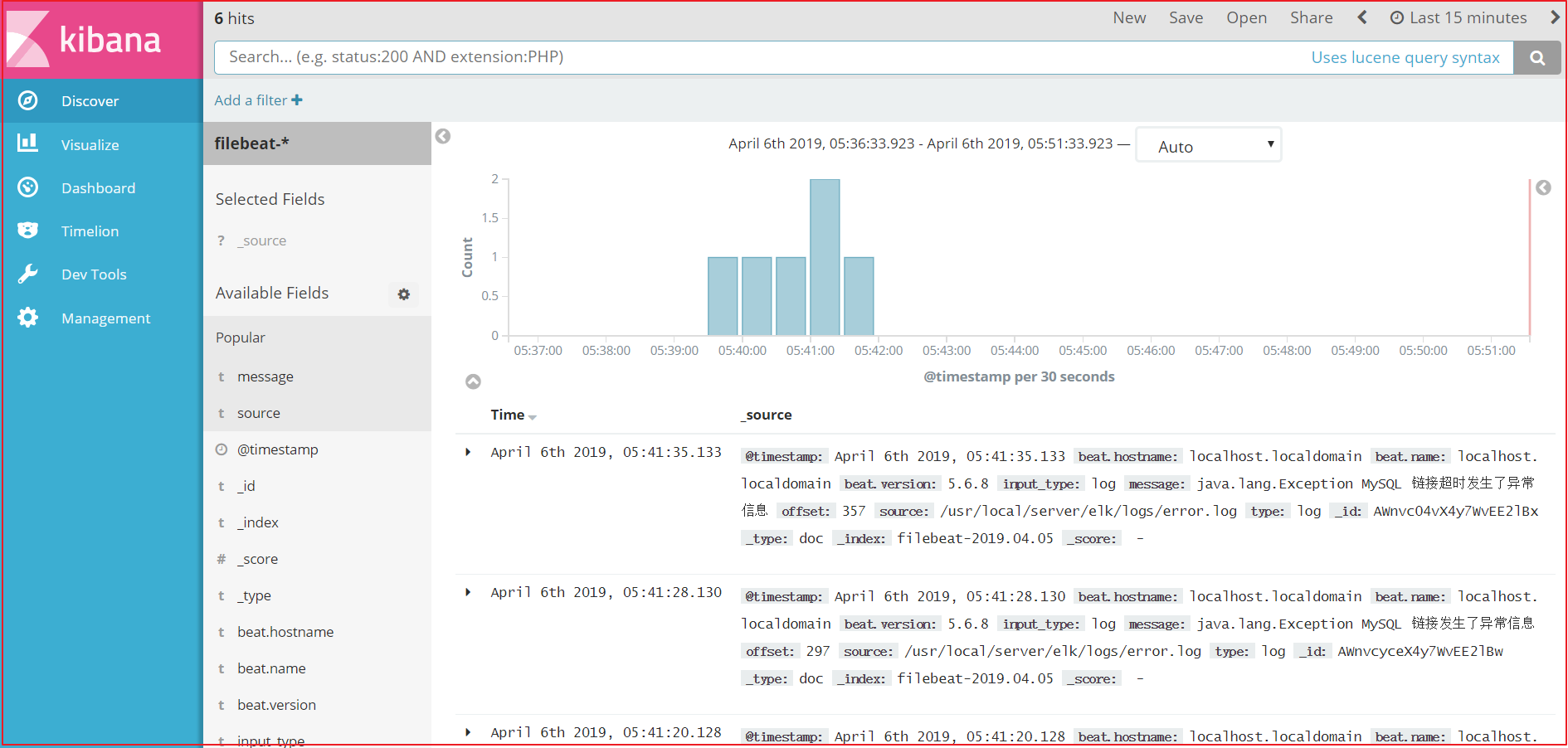


域的每个标题选项分别代表如下意思：

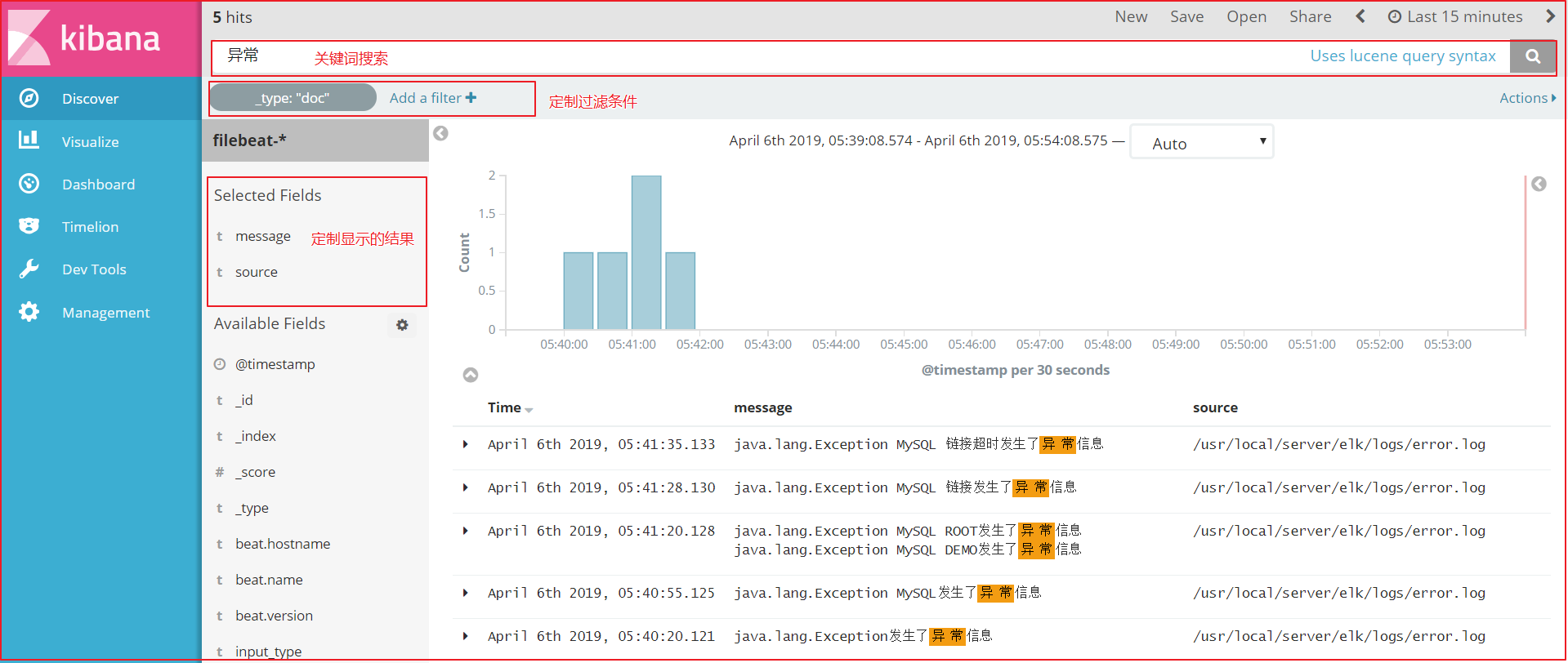


### 数据搜索

Discover为数据搜索部分，可以对日志信息进行搜索操作。



可以使用Discover实现数据搜索过滤和搜索条件显示以及关键词搜索，如下图：



## DSL语句使用

Domain Specific Language：领域特定语言，指的是针对特定应用领域而设计使用的计算机语言。

### Query DSL结构化查询介绍

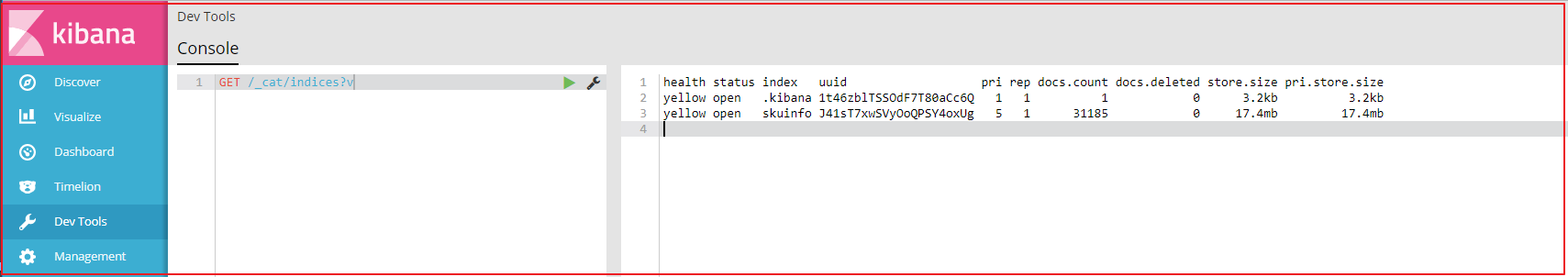
Query DSL是一个Java开源框架用于构建类型安全的SQL查询语句。采用API代替传统的拼接字符串来构造查询语句。目前Querydsl支持的平台包括JPA、JDO、SQL、Java Collections、RDF、Lucene、Hibernate Search。Elasticsearch提供了一整套基于JSON的查询DSL语言来定义查询。

Query DSL当作是一系列的抽象的查询表达式树(AST)特定查询能够包含其它的查询，(如 bool ), 有些查询能够包含过滤器(如 constant\_score), 还有的可以同时包含查询和过滤器 (如 filtered). 都能够从ES支持查询集合里面选择任意一个查询或者是从过滤器集合里面挑选出任意一个过滤器, 这样的话，我们就可以构造出任意复杂（maybe 非常有趣）的查询了。

### 索引操作

#### 查询所有索引

GET /\_cat/indices?v



#### 新增索引

PUT /user



#### 删除某个索引

**DELETE** /user



#### 创建映射

PUT /user/userinfo/\_mapping

{

"properties": {

"name":{

"type": "text",

"analyzer": "ik\_smart",

"search\_analyzer": "ik\_smart",

"store": false

},

"city":{

"type": "text",

"analyzer": "ik\_smart",

"search\_analyzer": "ik\_smart",

"store": false

},

"age":{

"type": "long",

"store": false

},

"description":{

"type": "text",

"analyzer": "ik\_smart",

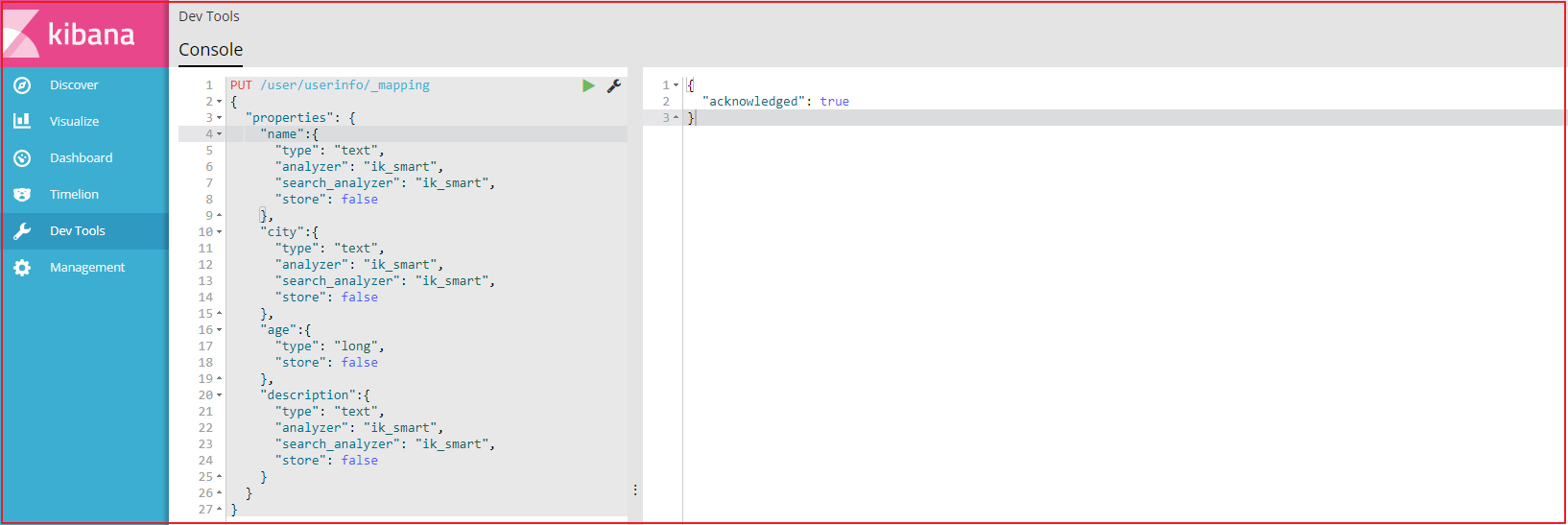
"search\_analyzer": "ik\_smart",

"store": false

}

}

}



#### 新增文档数据

PUT /user/userinfo/1 user:索引库名称 userinfo：库类型 1：id编号

{

"name":"李四",

"age":22,

"city":"深圳",

"description":"李四来自湖北武汉！"

}

效果如下：



我们再增加3条记录：

#新增文档数据 id=2

PUT /user/userinfo/2

{

"name":"王五",

"age":35,

"city":"深圳",

"description":"王五家住在深圳！"

}

#新增文档数据 id=3

PUT /user/userinfo/3

{

"name":"张三",

"age":19,

"city":"深圳",

"description":"在深圳打工，来自湖北武汉"

}

#新增文档数据 id=4

PUT /user/userinfo/4

{

"name":"张三丰",

"age":66,

"city":"武汉",

"description":"在武汉读书，家在武汉！"

}

#新增文档数据 id=5

PUT /user/userinfo/5

{

"name":"赵子龙",

"age":77,

"city":"广州",

"description":"赵子龙来自深圳宝安，但是在广州工作！",

"address":"广东省茂名市"

}

#新增文档数据 id=6

PUT /user/userinfo/6

{

"name":"赵毅",

"age":55,

"city":"广州",

"description":"赵毅来自广州白云区，从事电子商务8年！"

}

#新增文档数据 id=7

PUT /user/userinfo/7

{

"name":"赵哈哈",

"age":57,

"city":"武汉",

"description":"武汉赵哈哈，在深圳打工已有半年了，月薪7500！"

}

#### 修改数据

**a.替换操作**

更新数据可以使用之前的增加操作,这种操作会将整个数据替换掉，代码如下：

#更新数据,id=4

PUT /user/userinfo/4

{

"name":"张三丰",

"description":"在武汉读书，家在武汉！在深圳工作！"

}

效果如下：

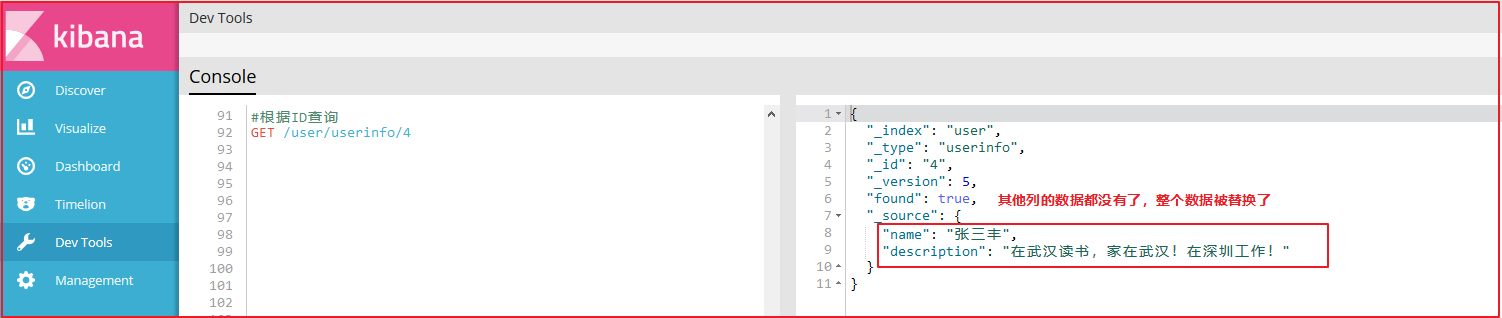


使用GET命令查看：

#根据ID查询

GET /user/userinfo/4

效果如下：



**b.更新操作**

我们先使用下面命令恢复数据：

#恢复文档数据 id=4

PUT /user/userinfo/4

{

"name":"张三丰",

"age":66,

"city":"武汉",

"description":"在武汉读书，家在武汉！"

}

使用POST更新某个列的数据

#使用POST更新某个域的数据

POST /user/userinfo/4/\_update

{

"doc":{

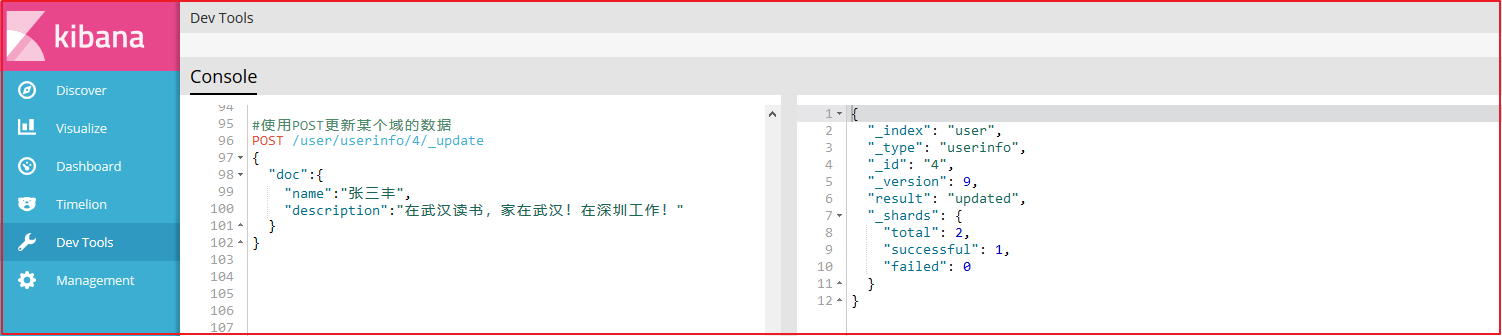
"name":"张三丰",

"description":"在武汉读书，家在武汉！在深圳工作！"

}

}

效果如下：



使用GET命令查看：

#根据ID查询

GET /user/userinfo/4

效果如下：



#### 删除Document

#删除数据

DELETE user/userinfo/7

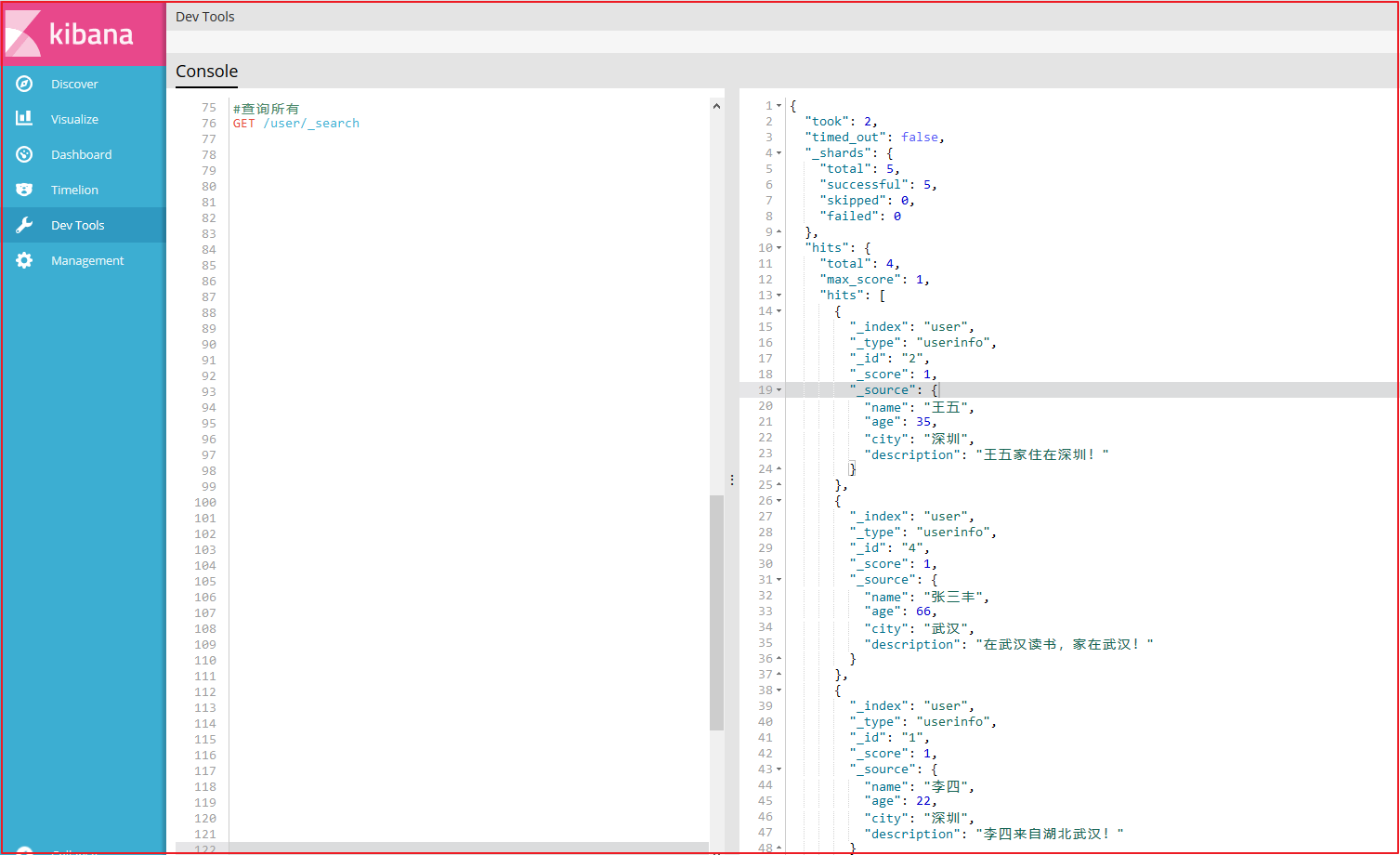
### 数据查询

#### 查询所有数据

#查询所有

GET /user/\_search

效果如下：



#### 根据ID查询

#根据ID查询

GET /user/userinfo/2

效果如下：



#### Sort排序

#搜索排序

GET /user/\_search

{

"query":{

"match\_all": {}

},

"sort":{

"age":{

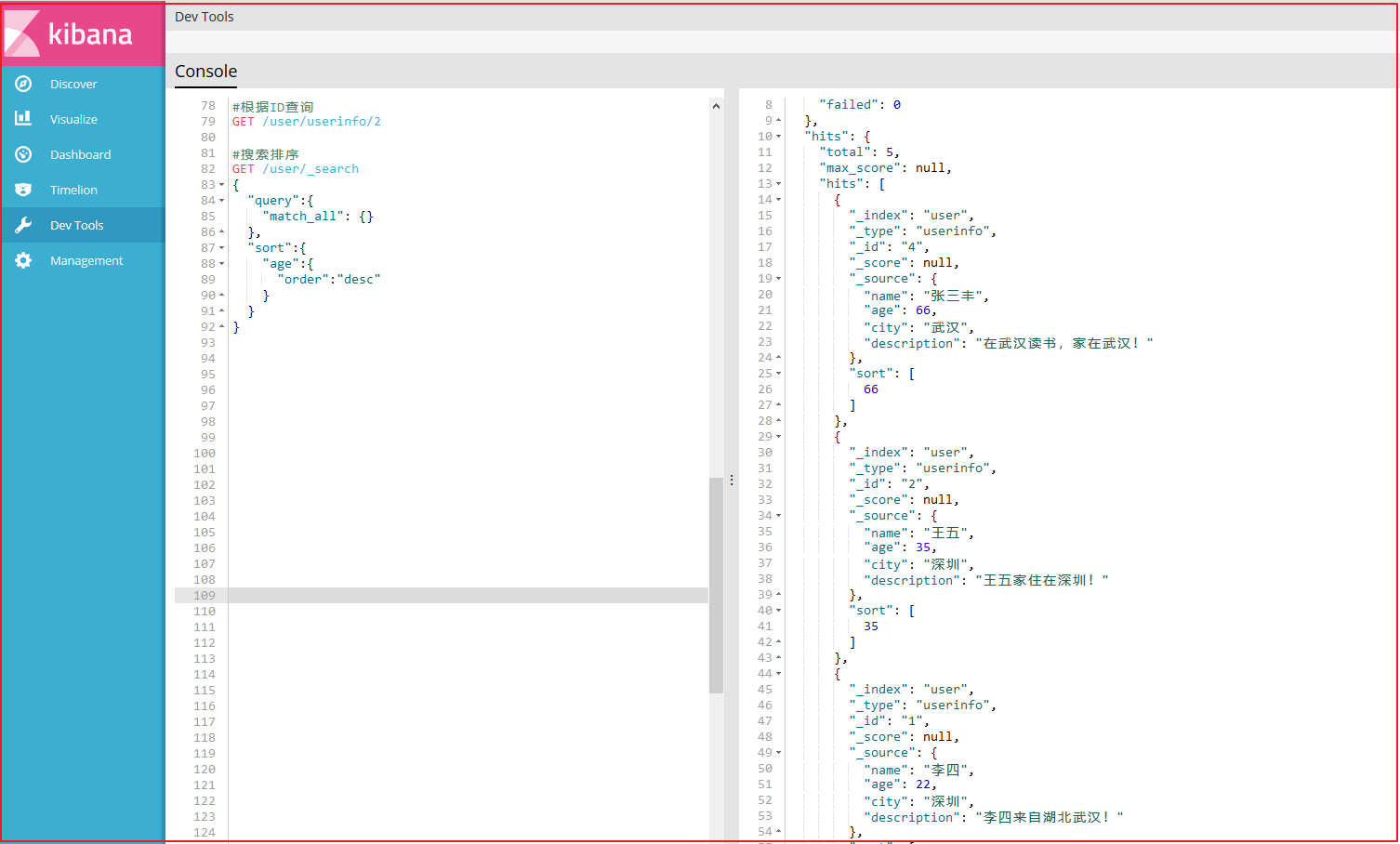
"order":"desc"

}

}

}

效果如下：



#### 分页

#分页实现

GET /user/\_search

{

"query":{

"match\_all": {}

},

"sort":{

"age":{

"order":"desc"

}

},

"from": 0,

"size": 2

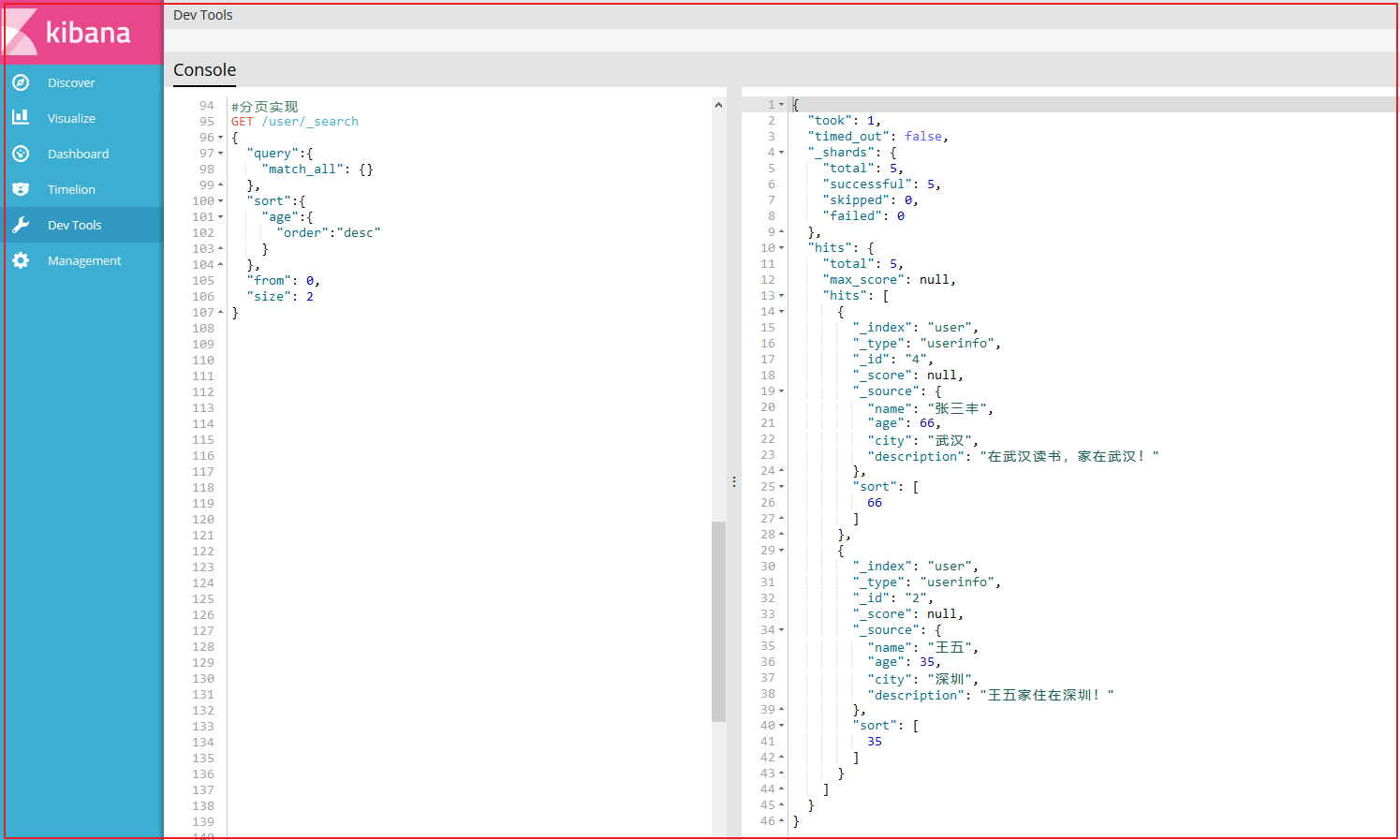
}

解释：

from:从下N的记录开始查询

size:每页显示条数

效果如下：



### 过滤查询

#### term过滤

term主要用于分词精确匹配，如字符串、数值、日期等（不适合情况：1.列中除英文字符外有其它值 2.字符串值中有冒号或中文 3.系统自带属性如\_version）

如下案例：

#过滤查询-term

GET \_search

{

"**query**":{

"**term**":{

"**city**":"武汉"

}

}

}

效果如下：



#### terms 过滤

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 。

案例如下：

#过滤查询-terms 允许多个Term

GET \_search

{

"**query**":{

"**terms**":{

"**city**":

[

"武汉",

"广州"

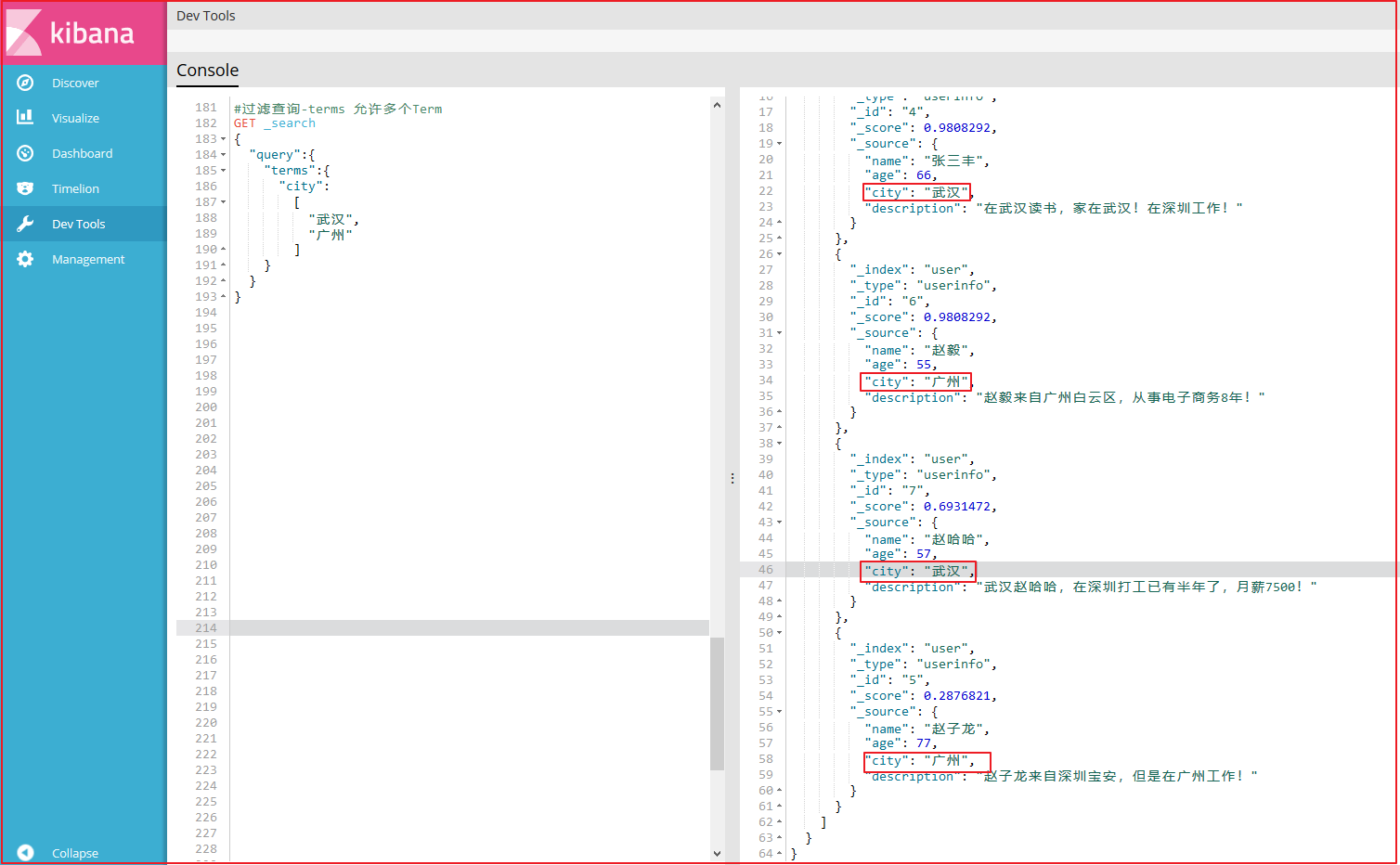
]

}

}

}

效果如下：



#### range 过滤

range过滤允许我们按照指定范围查找一批数据。例如我们查询年龄范围

案例如下：

#过滤-range 范围过滤

#gt表示> gte表示=>

#lt表示< lte表示<=

GET \_search

{

"**query**":{

"**range**": {

"**age**": {

"**gte**": 30,

"**lte**": 57

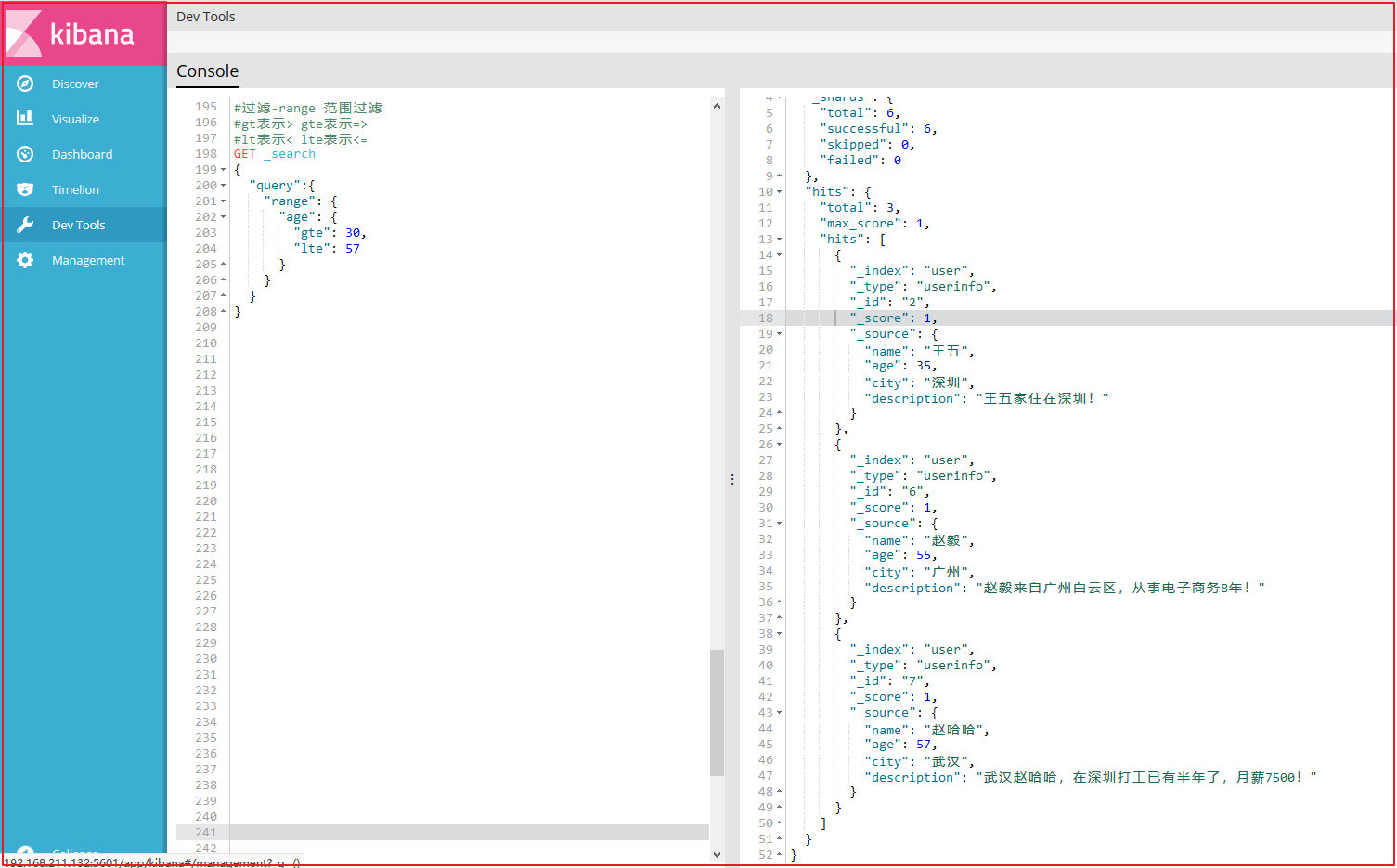
}

}

}

}

上图效果如下：



#### exists过滤

exists 过滤可以用于查找拥有某个域的数据

案例如下：

#过滤搜索 exists：是指包含某个域的数据检索

GET \_search

{

"**query**": {

"**exists**":{

"**field**":"address"

}

}

}

效果如下：



#### bool 过滤

bool 过滤可以用来合并多个过滤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过滤条件或者一个过滤条件的数组：

案例如下：

#过滤搜索 bool

#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。

GET \_search

{

"**query**": {

"**bool**": {

"**must**": [

{

"**term**": {

"**city**": {

"**value**": "深圳"

}

}

},

{

"**range**":{

"**age**":{

"**gte**":20,

"**lte**":99

}

}

}

]

}

}

}

效果如下：



#### match\_all 查询

可以查询到所有文档，是没有查询条件下的默认语句。

案例如下：

#查询所有 match\_all

GET \_search

{

"**query**": {

"**match\_all**": {}

}

}

#### 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：

案例如下：

#字符串匹配

GET \_search

{

"**query**": {

"**match**": {

"**description**": "武汉"

}

}

}

效果如下：



#### prefix 查询

以什么字符开头的，可以更简单地用 prefix ,例如查询所有以张开始的用户描述

案例如下：

#前缀匹配 prefix

GET \_search

{

"query": {

"prefix": {

"name": {

"value": "赵"

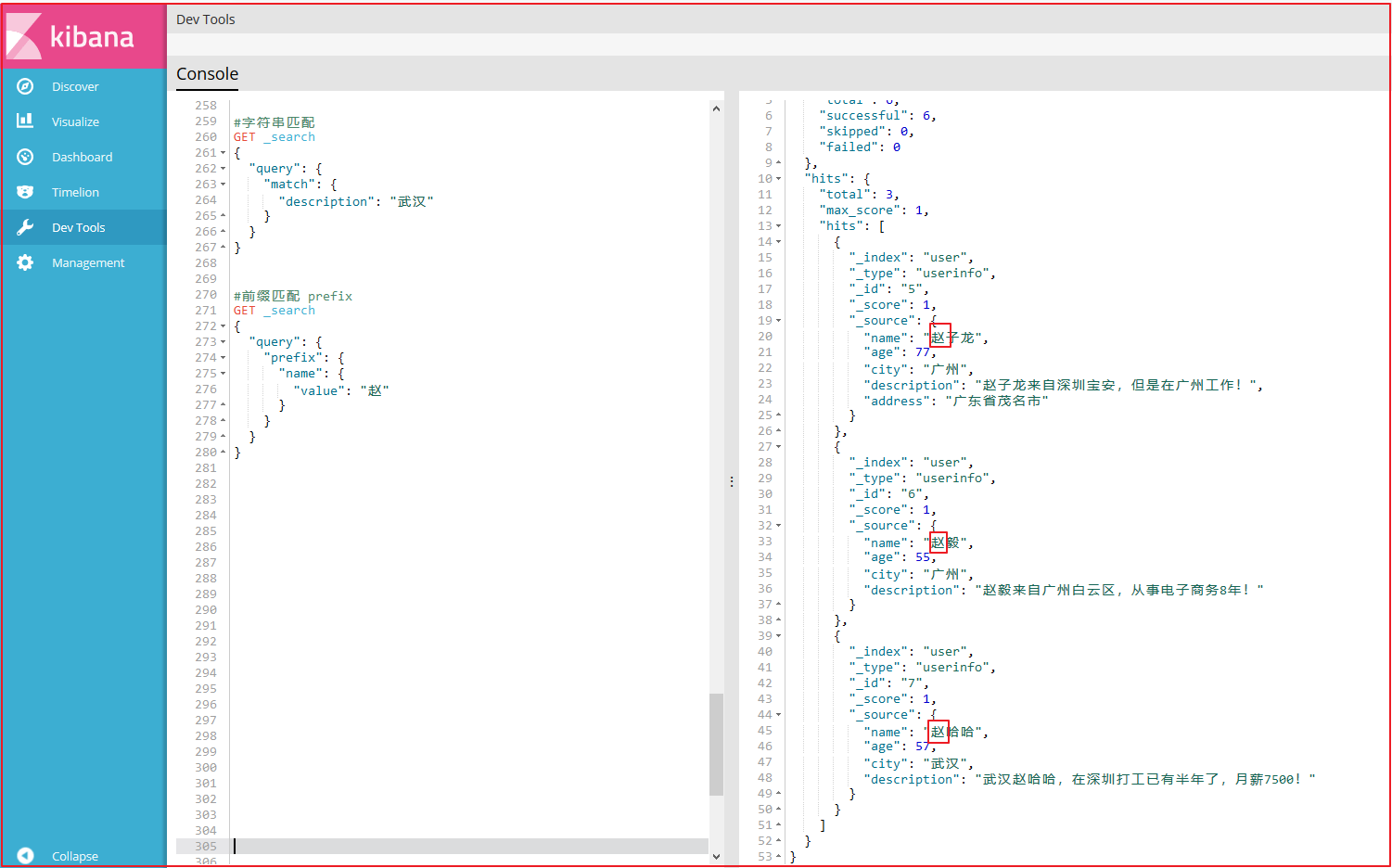
}

}

}

}

效果如下：



#### multi\_match 查询

multi\_match查询允许你做match查询的基础上同时搜索多个字段，在多个字段中同时查一个

案例如下：

#多个域匹配搜索

GET \_search

{

"**query**": {

"**multi\_match**": {

"**query**": "深圳",

"**fields**": [

"city",

"description"

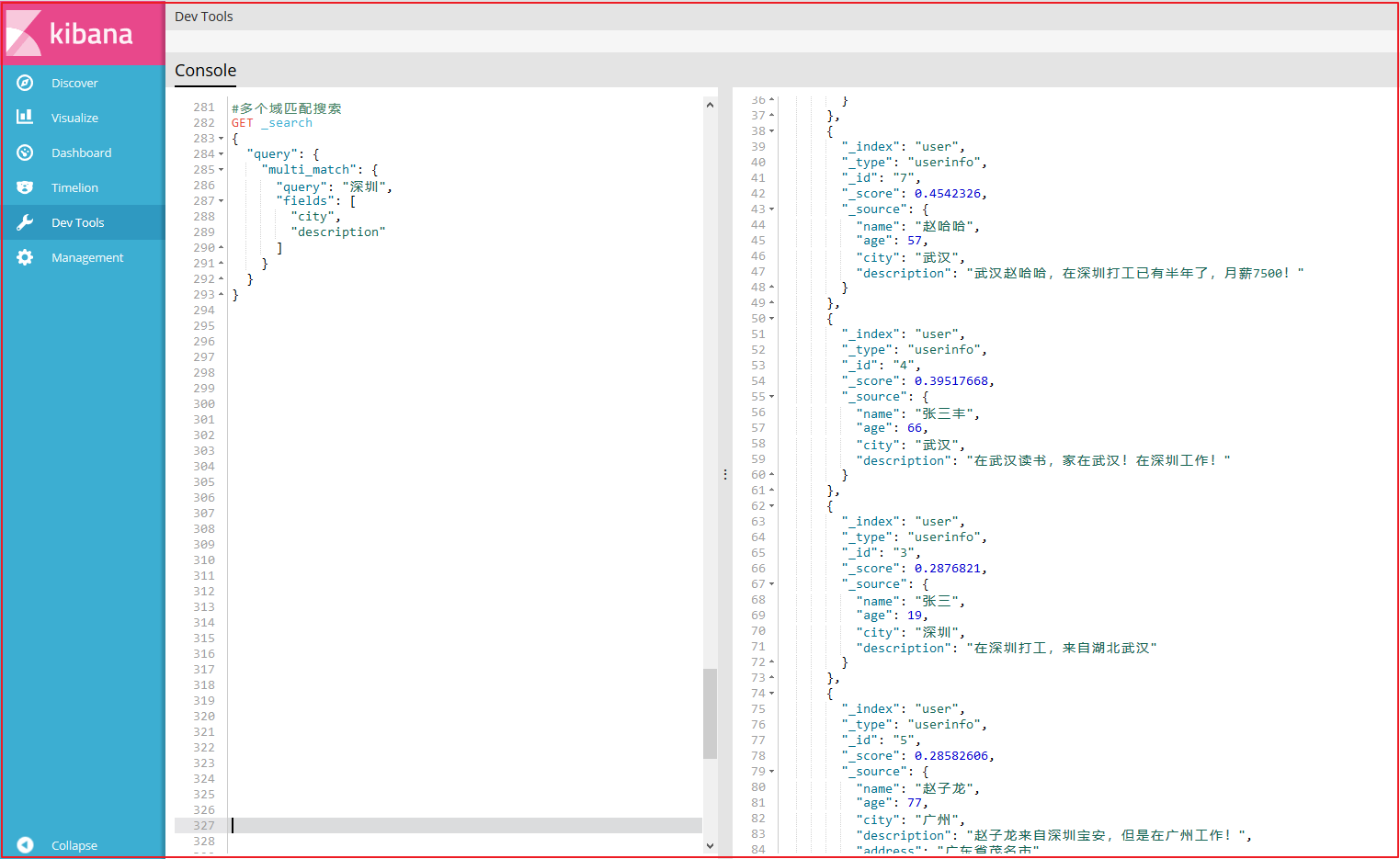
]

}

}

}

效果如下：



### 完整DSL语句代码

#查看所有索引

GET /\_cat/indices?v

#删除某个索引

DELETE /skuinfo

#新增索引

PUT /user

#创建映射

PUT /user/userinfo/\_mapping

{

"properties": {

"name":{

"type": "text",

"analyzer": "ik\_smart",

"search\_analyzer": "ik\_smart",

"store": false

},

"city":{

"type": "text",

"analyzer": "ik\_smart",

"search\_analyzer": "ik\_smart",

"store": false

},

"age":{

"type": "long",

"store": false

},

"description":{

"type": "text",

"analyzer": "ik\_smart",

"search\_analyzer": "ik\_smart",

"store": false

}

}

}

#新增文档数据 id=1

PUT /user/userinfo/1

{

"name":"李四",

"age":22,

"city":"深圳",

"description":"李四来自湖北武汉！"

}

#新增文档数据 id=2

PUT /user/userinfo/2

{

"name":"王五",

"age":35,

"city":"深圳",

"description":"王五家住在深圳！"

}

#新增文档数据 id=3

PUT /user/userinfo/3

{

"name":"张三",

"age":19,

"city":"深圳",

"description":"在深圳打工，来自湖北武汉"

}

#新增文档数据 id=4

PUT /user/userinfo/4

{

"name":"张三丰",

"age":66,

"city":"武汉",

"description":"在武汉读书，家在武汉！"

}

#新增文档数据 id=5

PUT /user/userinfo/5

{

"name":"赵子龙",

"age":77,

"city":"广州",

"description":"赵子龙来自深圳宝安，但是在广州工作！",

"address":"广东省茂名市"

}

#新增文档数据 id=6

PUT /user/userinfo/6

{

"name":"赵毅",

"age":55,

"city":"广州",

"description":"赵毅来自广州白云区，从事电子商务8年！"

}

#新增文档数据 id=7

PUT /user/userinfo/7

{

"name":"赵哈哈",

"age":57,

"city":"武汉",

"description":"武汉赵哈哈，在深圳打工已有半年了，月薪7500！"

}

#更新数据,id=4

PUT /user/userinfo/4

{

"name":"张三丰",

"description":"在武汉读书，家在武汉！在深圳工作！"

}

#根据ID查询

GET /user/userinfo/4

#恢复文档数据 id=4

PUT /user/userinfo/4

{

"name":"张三丰",

"age":66,

"city":"武汉",

"description":"在武汉读书，家在武汉！"

}

#使用POST更新某个域的数据

POST /user/userinfo/4/\_update

{

"doc":{

"name":"张三丰",

"description":"在武汉读书，家在武汉！在深圳工作！"

}

}

#根据ID查询

GET /user/userinfo/4

#删除数据

DELETE user/userinfo/4

#查询所有

GET /user/\_search

#根据ID查询

GET /user/userinfo/2

#搜索排序

GET /user/\_search

{

"query":{

"match\_all": {}

},

"sort":{

"age":{

"order":"desc"

}

}

}

#分页实现

GET /user/\_search

{

"query":{

"match\_all": {}

},

"sort":{

"age":{

"order":"desc"

}

},

"from": 0,

"size": 2

}

#过滤查询-term

GET \_search

{

"query":{

"term":{

"city":"武汉"

}

}

}

#过滤查询-terms 允许多个Term

GET \_search

{

"query":{

"terms":{

"city":

[

"武汉",

"广州"

]

}

}

}

#过滤-range 范围过滤

#gt表示> gte表示=>

#lt表示< lte表示<=

GET \_search

{

"query":{

"range": {

"age": {

"gte": 30,

"lte": 57

}

}

}

}

#过滤搜索 exists：是指包含某个域的数据检索

GET \_search

{

"query": {

"exists":{

"field":"address"

}

}

}

#过滤搜索 bool

#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。

GET \_search

{

"query": {

"bool": {

"must": [

{

"term": {

"city": {

"value": "深圳"

}

}

},

{

"range":{

"age":{

"gte":20,

"lte":99

}

}

}

]

}

}

}

#查询所有 match\_all

GET \_search

{

"query": {

"match\_all": {}

}

}

#字符串匹配

GET \_search

{

"query": {

"match": {

"description": "武汉"

}

}

}

#前缀匹配 prefix

GET \_search

{

"query": {

"prefix": {

"name": {

"value": "赵"

}

}

}

}

#多个域匹配搜索

GET \_search

{

"query": {

"multi\_match": {

"query": "深圳",

"fields": [

"city",

"description"

]

}

}

}

# 数据导入ES

## SpringData Elasticsearch介绍

### SpringData介绍

Spring Data是一个用于简化数据库访问，并支持云服务的开源框架。其主要目标是使得对数据的访问变得方便快捷，并支持map-reduce框架和云计算数据服务。 Spring Data可以极大的简化JPA的写法，可以在几乎不用写实现的情况下，实现对数据的访问和操作。除了CRUD外，还包括如分页、排序等一些常用的功能。

Spring Data的官网：<http://projects.spring.io/spring-data/>

### SpringData ES介绍

Spring Data ElasticSearch 基于 spring data API 简化 elasticSearch操作，将原始操作elasticSearch的客户端API 进行封装 。Spring Data为Elasticsearch项目提供集成搜索引擎。Spring Data Elasticsearch POJO的关键功能区域为中心的模型与Elastichsearch交互文档和轻松地编写一个存储库数据访问层。 官方网站：<http://projects.spring.io/spring-data-elasticsearch/>

## 搜索工程搭建

创建搜索微服务工程，changgou\_service\_search,该工程主要提供搜索服务以及索引数据的更新操作。

### API工程搭建

首先创建search的API工程,在changgou-service-api中创建changgou\_service\_search\_api，如下图：

*<?***xml version="1.0" encoding="UTF-8"***?>*<**project xmlns="http://maven.apache.org/POM/4.0.0"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"**>  
 <**parent**>  
 <**artifactId**>changgou\_service\_api</**artifactId**>  
 <**groupId**>com.changgou</**groupId**>  
 <**version**>1.0-SNAPSHOT</**version**>  
 </**parent**>  
 <**modelVersion**>4.0.0</**modelVersion**>  
 <**artifactId**>changgou\_service\_search\_api</**artifactId**>  
  
 <**dependencies**>  
 *<!--商品goods API-->* <**dependency**>  
 <**groupId**>com.changgou</**groupId**>  
 <**artifactId**>changgou\_service\_goods\_api</**artifactId**>  
 <**version**>1.0-SNAPSHOT</**version**>  
 </**dependency**>  
 *<!--SpringDataES依赖-->* <**dependency**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-data-elasticsearch</**artifactId**>  
 </**dependency**>  
 </**dependencies**>  
</**project**>

### 搜索微服务搭建

在changgou-service中搭建changgou\_service\_search微服务，并进行相关配置。

*<?***xml version="1.0" encoding="UTF-8"***?>*<**project xmlns="http://maven.apache.org/POM/4.0.0"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"**>  
 <**parent**>  
 <**artifactId**>changgou\_service</**artifactId**>  
 <**groupId**>com.changgou</**groupId**>  
 <**version**>1.0-SNAPSHOT</**version**>  
 </**parent**>  
 <**modelVersion**>4.0.0</**modelVersion**>  
 <**artifactId**>changgou\_service\_search</**artifactId**>  
  
 <**dependencies**>  
 <**dependency**>  
 <**groupId**>com.changgou</**groupId**>  
 <**artifactId**>changgou\_service\_search\_api</**artifactId**>  
 <**version**>1.0-SNAPSHOT</**version**>  
 </**dependency**>  
 </**dependencies**>  
</**project**>

#### application.yml配置

**server**:  
 **port**: 18085  
**spring**:  
 **application**:  
 **name**: search  
 **data**:  
 **elasticsearch**:  
 *# Elasticsearch的集群节点名称，这里需要和Elasticsearch集群节点名称保持一致* **cluster-name**: my-application  
 *# Elasticsearch节点通信地址* **cluster-nodes**: 192.168.211.132:9300  
**eureka**:  
 **client**:  
 **service-url**:  
 **defaultZone**: http://127.0.0.1:7001/eureka  
 **instance**:  
 **prefer-ip-address**: **true  
feign**:  
 **hystrix**:  
 **enabled**: **true***#超时配置***ribbon**:  
 *# Feign请求读取数据超时时间* **ReadTimeout**: 300000  
  
**hystrix**:  
 **command**:  
 **default**:  
 **execution**:  
 **isolation**:  
 **thread**:  
 *# 超时时间，默认1000ms* **timeoutInMilliseconds**: 10000

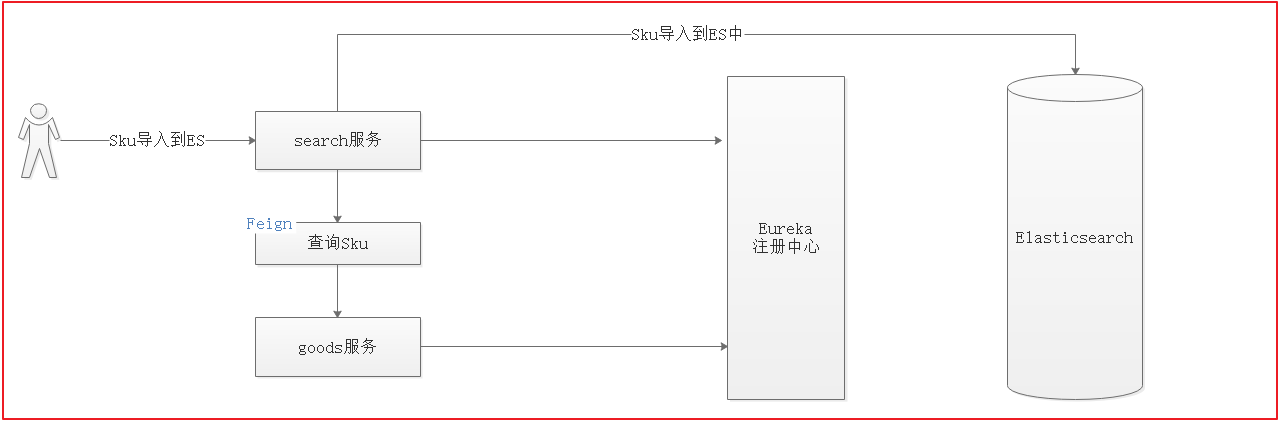
#### 启动引导类

创建SearchApplication作为搜索微服务工程的启动类，代码如下：

@SpringBootApplication(exclude={DataSourceAutoConfiguration.**class**})  
@EnableEurekaClient  
**public class** SearchApplication {  
  
 **public static void** main(String[] args) {  
 */\*\*  
 \* Springboot整合Elasticsearch 在项目启动前设置一下的属性，防止报错  
 \* 解决netty冲突后初始化client时还会抛出异常  
 \* availableProcessors is already set to [12], rejecting [12]  
 \*\*\*/* System.*setProperty*(**"es.set.netty.runtime.available.processors"**, **"false"**);  
 SpringApplication.*run*(SearchApplication.**class**,args);  
 }  
}

## 数据导入

现在需要将数据从数据库中查询出来，然后将数据导入到ES中。



数据导入流程如下：

1.请求search服务,调用数据导入地址

2.根据注册中心中的注册的goods服务的地址，使用Feign方式查询所有已经审核的Sku

3.使用SpringData Es将查询到的Sku集合导入到ES中

### 文档映射Bean创建（根据情况来分词和存入ES数据库中）

搜索商品的时候，会根据如下属性搜索数据,并且不是所有的属性都需要分词搜索，我们创建JavaBean，将JavaBean数据存入到ES中要以搜索条件和搜索展示结果为依据，部分关键搜索条件分析如下：

1.可能会根据商品名称搜素，而且可以搜索商品名称中的任意一个词语，所以需要分词

2.可能会根据商品分类搜索，商品分类不需要分词

3.可能会根据商品品牌搜索，商品品牌不需要分词

4.可能会根据商品商家搜索，商品商家不需要分词

5.可能根据规格进行搜索，规格时一个键值对结构，用Map

根据上面的分析，我们可以在changgou-service-search-api工程中创建com.changgou.search.pojo.SkuInfo

@Document(indexName = **"skuinfo"**,type = **"docs"**)  
**public class** SkuInfo **implements** Serializable {  
 *//商品id，同时也是商品编号* @Id  
 **private** Long **id**;  
  
 *//SKU名称* @Field(type = FieldType.***Text***, analyzer = **"ik\_smart"**)  
 **private** String **name**;  
  
 *//商品价格，单位为：元* @Field(type = FieldType.***Double***)  
 **private** Long **price**;  
  
 *//库存数量* **private** Integer **num**;  
  
 *//商品图片* **private** String **image**;  
  
 *//商品状态，1-正常，2-下架，3-删除* **private** String **status**;  
  
 *//创建时间* **private** Date **createTime**;  
  
 *//更新时间* **private** Date **updateTime**;  
  
 *//是否默认* **private** String **isDefault**;  
  
 *//SPUID* **private** Long **spuId**;  
  
 *//类目ID* **private** Long **categoryId**;  
  
 *//类目名称* @Field(type = FieldType.***Keyword***)  
 **private** String **categoryName**;  
  
 *//品牌名称* @Field(type = FieldType.***Keyword***)  
 **private** String **brandName**;  
  
 *//规格* **private** String **spec**;  
  
 *//规格参数* **private** Map<String,Object> **specMap**;  
  
 *//...略*}

### 搜索审核通过Sku

修改changgou-service-goods微服务，添加搜索审核通过的Sku，供search微服务调用。下面都是针对goods微服务的操作。

#### 业务层

不用添加代码，直接使用findList(Sku sku)方法

#### 控制层

修改com.changgou.goods.controller.SkuController，添加根据审核状态查询Sku方法，代码如下：

*/\*\*\*  
 \* 根据审核状态查询Sku  
 \** ***@param status*** *\** ***@return*** *\*/*@GetMapping(**"/status/{status}"**)  
**public** Result<List<Sku>> findByStatus(@PathVariable String status){  
 Sku where = **new** Sku();  
 where.setStatus(status);  
 List<Sku> list = **skuService**.findList(where);  
 **return new** Result<List<Sku>>(**true**,StatusCode.***OK***,**"查询成功"**,list);  
}

#### Feign配置

修改changgou-service-goods-api工程，在com.changgou.goods.feign.SkuFeign上添加findSkuList方法，代码如下：

@FeignClient(name = **"goods"**)  
@RequestMapping(**"sku"**)  
**public interface** SkuFeign {  
  
 */\*\*\*  
 \* 根据审核状态查询Sku  
 \** ***@param status*** *\** ***@return*** *\*/* @GetMapping(**"/status/{status}"**)  
 **public** Result<List<Sku>> findByStatus(@PathVariable String status);  
}

### Sku导入ES实现

#### Dao创建

修改changgou-service-search工程，创建com.changgou.search.dao.SkuEsMapper,该接口主要用于索引数据操作，主要使用它来实现将数据导入到ES索引库中，代码如下：

@Repository  
**public interface** SkuEsMapper **extends** ElasticsearchRepository<SkuInfo,Long> {  
}

#### 服务层创建

修改changgou-service-search工程，创建com.changgou.search.service.SkuService,代码如下：

**public interface** SkuService {  
  
 */\*\*\*  
 \* 导入SKU数据  
 \*/* **void** importSku();  
}

修改changgou-service-search工程，创建com.changgou.search.service.impl.SkuServiceImpl,实现Sku数据导入到ES中，代码如下：

@Service  
**public class** SkuServiceImpl **implements** SkuService {  
  
 @Autowired  
 **private** SkuFeign **skuFeign**;  
  
 @Autowired  
 **private** SkuEsMapper **skuEsMapper**;  
  
 */\*\*  
 \* 导入sku数据到es  
 \*/* @Override  
 **public void** importSku(){  
 *//调用changgou-service-goods微服务* Result<List<Sku>> skuListResult = **skuFeign**.findByStatus(**"1"**);  
 *//将数据转成search.Sku* List<SkuInfo> skuInfos= JSON.*parseArray*(JSON.*toJSONString*(skuListResult.getData()),SkuInfo.**class**);  
 **for**(SkuInfo skuInfo:skuInfos){  
 Map<String, Object> specMap= JSON.*parseObject*(skuInfo.getSpec()) ;  
 skuInfo.setSpecMap(specMap);  
 }  
 **skuEsMapper**.saveAll(skuInfos);  
 }  
}

#### 控制层配置

修改changgou-service-search工程，在com.changgou.search.controller.SkuController类中添加如下方法调用上述导入方法，代码如下：

@RestController  
@RequestMapping(value = **"search"**)  
@CrossOrigin  
**public class** SkuController {  
  
 @Autowired  
 **private** SkuService **skuService**;  
  
 */\*\*  
 \* 导入数据  
 \** ***@return*** *\*/* @GetMapping(**"import"**)  
 **public** Result search(){  
 **skuService**.importSku();  
 **return new** Result(**true**, StatusCode.***OK***,**"导入数据到索引库中成功！"**);  
 }  
}

#### 修改启动类

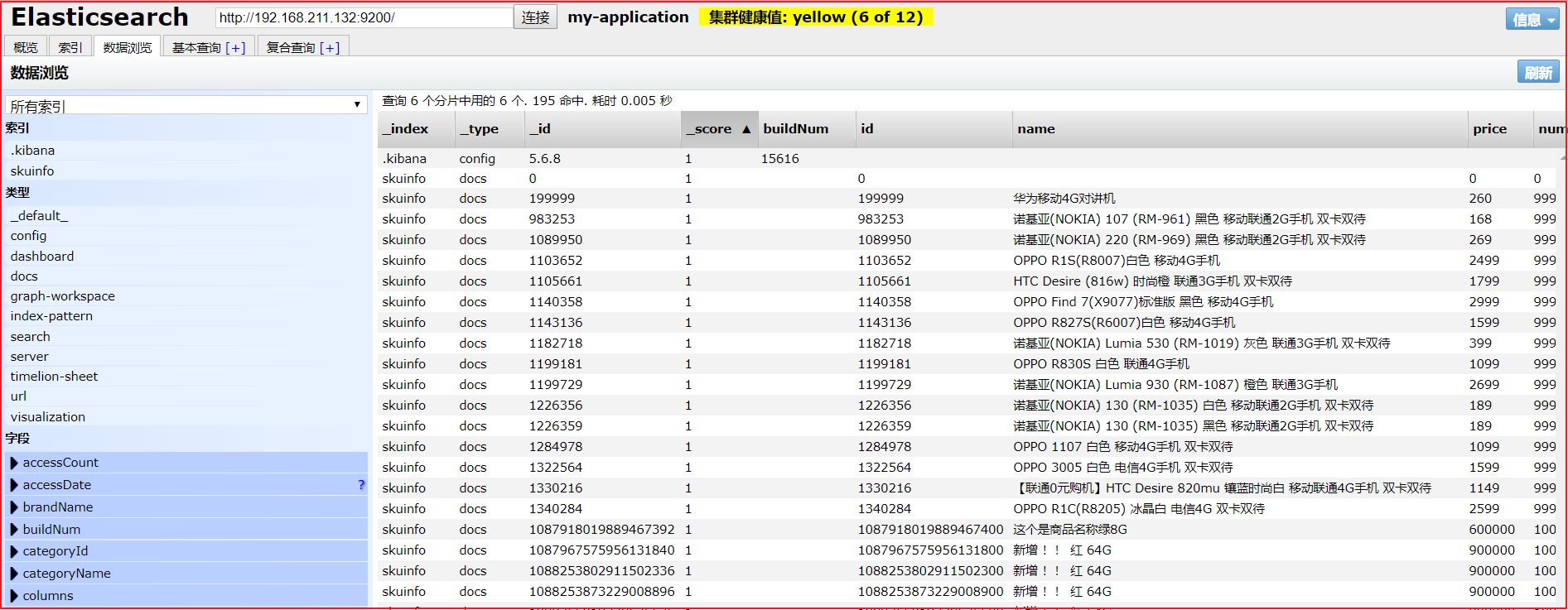
启动类中需要开启Feign客户端，并且需要添加ES包扫描，代码如下：

@SpringBootApplication(exclude={DataSourceAutoConfiguration.**class**})  
@EnableEurekaClient  
@EnableFeignClients(basePackages = **"com.changgou.goods.feign"**)  
@EnableElasticsearchRepositories(basePackages = **"com.changgou.search.dao"**)  
**public class** SearchApplication {  
  
 **public static void** main(String[] args) {  
 */\*\*  
 \* Springboot整合Elasticsearch 在项目启动前设置一下的属性，防止报错  
 \* 解决netty冲突后初始化client时还会抛出异常  
 \* availableProcessors is already set to [12], rejecting [12]  
 \*\*\*/* System.*setProperty*(**"es.set.netty.runtime.available.processors"**, **"false"**);  
 SpringApplication.*run*(SearchApplication.**class**,args);  
 }  
}

#### 测试

调用<http://localhost:18085/search/import> 进行测试

打开es-head可以看到如下数据：



# 关键字搜索



我们先使用SpringDataElasticsearch实现一个简单的搜索功能，先实现根据关键字搜索，从上面搜索图片可以看得到，每次搜索的时候，除了关键字外，还有可能有品牌、分类、规格等，后台接收搜索条件使用Map接收比较合适。

## 服务层实现

修改search服务的com.changgou.search.service.SkuService,添加搜索方法，代码如下：

*/\*\*\*  
 \* 搜索商品  
 \** ***@param searchMap*** *搜索条件  
 \** ***@return*** *结果集  
 \*/*Map search(Map<String, String> searchMap);

修改search服务的com.changgou.search.service.impl.SkuServiceImpl,添加搜索实现方法

**步骤**:

*//1、创建查询条件构建器-builder = new NativeSearchQueryBuilder()*

*//2、组装查询条件*

*//2.1关键字搜索-builder.withQuery(QueryBuilders.matchQuery(域名，内容))*

*//3、获取NativeSearchQuery搜索条件对象-builder.build()*

*//4.查询数据-esTemplate.queryForPage(条件对象,搜索结果对象)*

*//5、包装结果并返回*

代码：

@Autowired  
**private** ElasticsearchTemplate **esTemplate**;  
@Override  
**public** Map search(Map<String, String> searchMap) {  
 Map map = **new** HashMap();  
 *//1、创建查询条件构建器-builder = new NativeSearchQueryBuilder()* NativeSearchQueryBuilder builder = **new** NativeSearchQueryBuilder();  
 *//2、组装查询条件* **if**(searchMap != **null**){  
 *//2.1关键字搜索-builder.withQuery(QueryBuilders.matchQuery(域名，内容))* String keywords = searchMap.get(**"keywords"**) == **null** ? **""** : searchMap.get(**"keywords"**);  
 *//如果用户传入了关键字* **if**(StringUtils.*isNotEmpty*(keywords)){  
 *//查询name域* builder.withQuery(QueryBuilders.*matchQuery*(**"name"**, keywords));  
 }  
 }  
 *//3、获取NativeSearchQuery搜索条件对象-builder.build()* NativeSearchQuery query = builder.build();  
 *//4.查询数据-esTemplate.queryForPage(条件对象,搜索结果对象)* AggregatedPage<SkuInfo> page = **esTemplate**.queryForPage(query, SkuInfo.**class**);  
 *//5、包装结果并返回* map.put(**"rows"**, page.getContent());  
 map.put(**"total"**, page.getTotalElements());  
 map.put(**"totalPages"**, page.getTotalPages());  
 **return** map;  
}

## 控制层实现

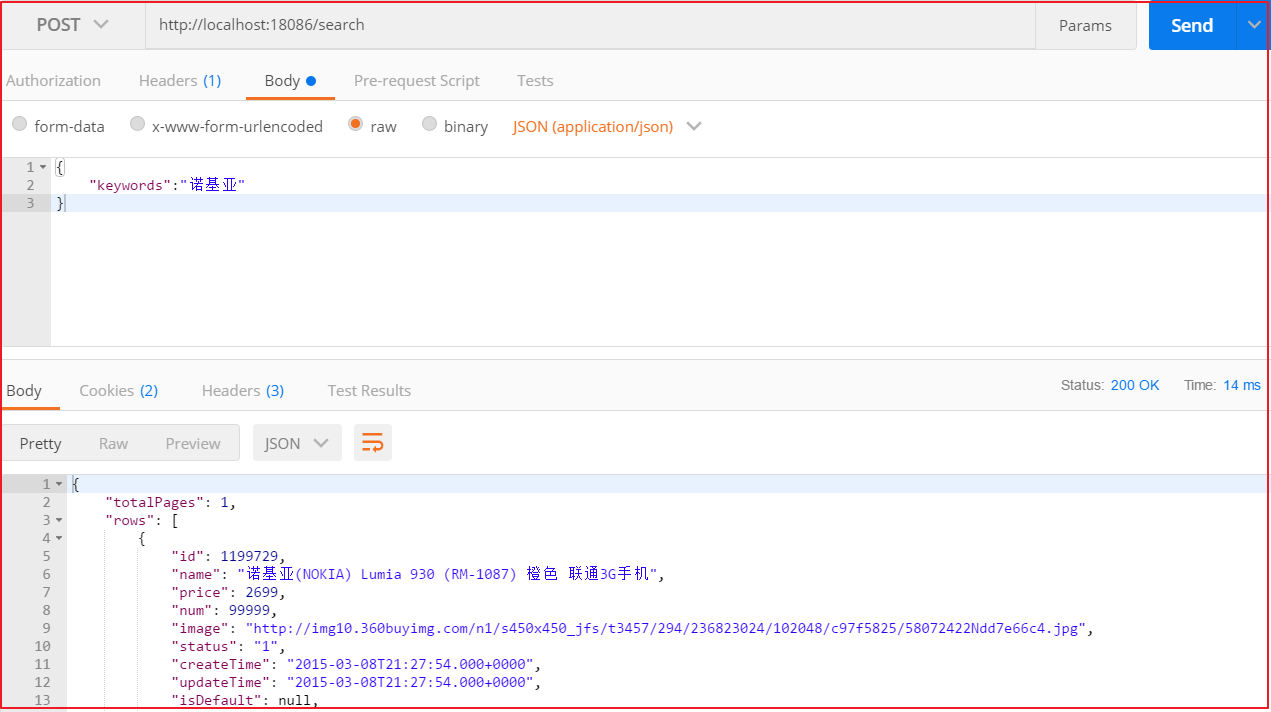
修改com.changgou.search.controller.SkuController，在控制层调用Service层即可，代码如下：

*/\*\*  
 \* 搜索商品  
 \** ***@return*** *\*/*@PostMapping  
**public** Map search(@RequestBody(required = **false**) Map searchMap){  
 **return skuService**.search(searchMap);  
}

## 测试

使用Postman工具，输入<http://localhost:18085/search>

选中POST提交



## 代码优化

由于后续我们商品查询业务非常复杂，为了提高代码可读性与重用代码，我们把刚才的代码提取出来

@Autowired  
**private** ElasticsearchTemplate **esTemplate**;  
@Override  
**public** Map search(Map<String, String> searchMap) {  
 Map map = **new** HashMap();  
 *//1、构建基本查询条件* NativeSearchQueryBuilder builder = builderBasicQuery(searchMap);  
 *//2、根据查询条件-搜索商品列表* searchList(map, builder);  
 **return** map;  
}  
  
*/\*\*  
 \* 根据查询条件-搜索商品列表  
 \** ***@param map*** *结果集  
 \** ***@param builder*** *查询条件构建器  
 \*/***private void** searchList(Map map, NativeSearchQueryBuilder builder) {  
 *//3、获取NativeSearchQuery搜索条件对象-builder.build()* NativeSearchQuery query = builder.build();  
 *//4.查询数据-esTemplate.queryForPage(条件对象,搜索结果对象)* AggregatedPage<SkuInfo> page = **esTemplate**.queryForPage(query, SkuInfo.**class**);  
 *//5、包装结果并返回* map.put(**"rows"**, page.getContent());  
 map.put(**"total"**, page.getTotalElements());  
 map.put(**"totalPages"**, page.getTotalPages());  
}  
  
*/\*\*  
 \* 构建基本查询条件  
 \** ***@param searchMap*** *用户传入的查询参数  
 \** ***@return*** *查询条件构建器  
 \*/***private** NativeSearchQueryBuilder builderBasicQuery(Map<String, String> searchMap) {  
 *//1、创建查询条件构建器-builder = new NativeSearchQueryBuilder()* NativeSearchQueryBuilder builder = **new** NativeSearchQueryBuilder();  
 *//2、组装查询条件* **if**(searchMap != **null**){  
 *//2.1关键字搜索-builder.withQuery(QueryBuilders.matchQuery(域名，内容))* String keywords = searchMap.get(**"keywords"**) == **null** ? **""** : searchMap.get(**"keywords"**);  
 *//如果用户传入了关键字* **if**(StringUtils.*isNotEmpty*(keywords)){  
 *//查询name域* builder.withQuery(QueryBuilders.*matchQuery*(**"name"**, keywords));  
 }  
 }  
 **return** builder;  
}

# 分类统计

## 分类统计分析

看下面的SQL语句，我们在执行搜索的时候，第1条SQL语句是执行搜，第2条语句是根据分类名字分组查看有多少分类，大概执行了2个步骤就可以获取数据结果以及分类统计，我们可以发现他们的搜索条件完全一样。

-- 查询所有

**SELECT** \* **FROM** tb\_sku **WHERE** name **LIKE** '%手机%';

-- 根据分类名字分组查询

**SELECT** category\_name **FROM** tb\_sku **WHERE** name **LIKE** '%手机%' **GROUP** **BY** category\_name;



我们每次执行搜索的时候，需要显示商品分类名称，这里要显示的分类名称其实就是符合搜素条件的所有商品的分类集合，我们可以按照上面的实现思路，使用ES根据分组名称做一次分组查询即可实现。

## 分类分组统计实现

修改search微服务的com.changgou.search.service.impl.SkuServiceImpl类，创建方法searchCategoryList(Map map,NativeSearchQueryBuilder builder)

**步骤：**

1.设置分组域名-termsAggregationBuilder = AggregationBuilders.terms(别名).field(域名);

2.添加分组查询参数-builder.addAggregation(termsAggregationBuilder)

3.执行搜索-esTemplate.queryForPage(builder.build(), SkuInfo.class)

4.获取所有分组查询结果集-page.getAggregations()

5.提取分组结果数据-stringTerms = aggregations.get(填入刚才查询时的别名)

6.定义分类名字列表-categoryList = new ArrayList<String>()

7.遍历读取分组查询结果-stringTerms.getBuckets().for

7.1获取分类名字，并将分类名字存入到集合中-bucket.getKeyAsString()

8.返回分类数据列表-map.put("categoryList", categoryList)

### 实现分组查询逻辑

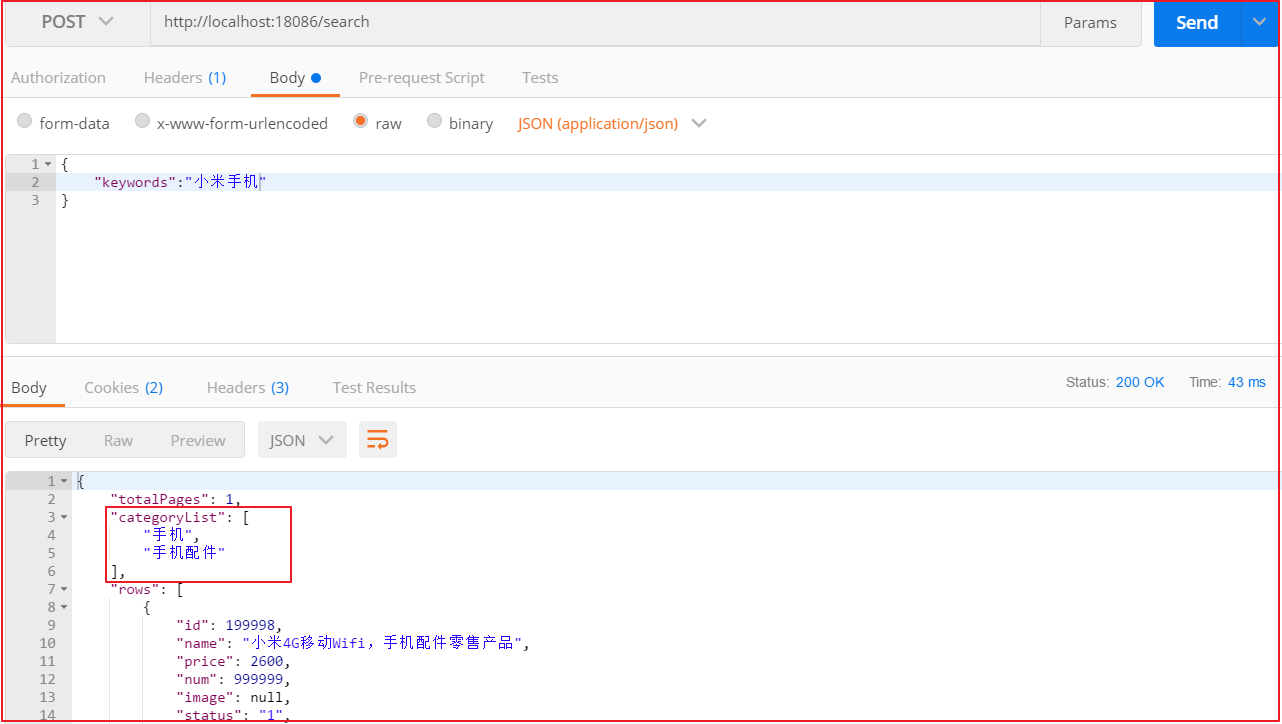
*/\*\*  
 \* 跟据查询条件-分组查询商品分类列表  
 \** ***@param map*** *结果集包装  
 \** ***@param builder*** *查询条件构建器  
 \*/***private void** searchCategoryList(Map map,NativeSearchQueryBuilder builder){  
 *//1.设置分组域名-termsAggregationBuilder = AggregationBuilders.terms(别名).field(域名);* TermsAggregationBuilder termsAggregationBuilder = AggregationBuilders.*terms*(**"group\_category"**).field(**"categoryName"**);  
 *//2.添加分组查询参数-builder.addAggregation(termsAggregationBuilder)* builder.addAggregation(termsAggregationBuilder);  
 *//3.执行搜索-esTemplate.queryForPage(builder.build(), SkuInfo.class)* AggregatedPage<SkuInfo> page = **esTemplate**.queryForPage(builder.build(), SkuInfo.**class**);  
 *//4.获取所有分组查询结果集-page.getAggregations()* Aggregations aggregations = page.getAggregations();  
 *//5.提取分组结果数据-stringTerms = aggregations.get(填入刚才查询时的别名)* StringTerms stringTerms = aggregations.get(**"group\_category"**);  
 *//6.定义分类名字列表-categoryList = new ArrayList<String>()* List<String> categoryList = **new** ArrayList<String>();  
 *//7.遍历读取分组查询结果-stringTerms.getBuckets().for* **for** (StringTerms.Bucket bucket : stringTerms.getBuckets()) {  
 *//7.1获取分类名字，并将分类名字存入到集合中-bucket.getKeyAsString()* categoryList.add(bucket.getKeyAsString());  
 }  
 *//8.返回分类数据列表-map.put("categoryList", categoryList)* map.put(**"categoryList"**, categoryList);  
}

### search方法调用查询逻辑

@Override  
**public** Map search(Map<String, String> searchMap) {  
 Map map = **new** HashMap();  
 *//1、构建基本查询条件* NativeSearchQueryBuilder builder = builderBasicQuery(searchMap);  
 *//2、根据查询条件-搜索商品列表* searchList(map, builder);  
 *//3、跟据查询条件-分组查询商品分类列表* searchCategoryList(map,builder);  
 **return** map;  
}

## 测试

请求<http://localhost:18086/search>



思考：

分类可以通过查询数据统计实现，那么规格和品牌呢？该如何实现呢？同学们思考一下，我们明天来实现。