## ElasticSearch

### ElasticSearch介绍

在微服务中，随着业务的发展，数据量越来越庞大，传统的MySQL数据库渐渐地难以满足复杂业务的需求，所以在微服务架构系统下，一般都会用到分布式搜索的技术，其中ElasticSearch是最流行的。

ElasticSearch：一款非常强大的开源的搜索引擎，可以帮助我们从海量数据中快速找到需要的内容（关键字搜索、高亮显示、模糊搜索、距离）。

### ELK介绍（elasticsearch、kibana、Logstash、Beats）

ElasticSearch结合kibana、Logstash、Beats，就是我们所说的elestic stack（ELK）。被广泛应用在微服务日志数据收集、分析、实时监控等领域。

#### 日志数据分析：

项目在运行过程中会产生海量的日志信息，这些日志信息可以方便我们去定位项目运行出现的问题，比如系统报错了方便找出错误，因为线上运行的应用不可能打断点，可以将日志数据可视化展示；

#### 实时监控：

项目的运行过程中项目的运行状态也是数据，每一刻都有状态；比如CPU情况、内存情况、访问频率，这些也会被管理以可视化展示出来，能非常清楚知道项目运行情况。

ELK技术栈中，核心是ElasticSearch，负责存储、计算、搜索、分析数据；Logstash、Beats负责数据抓取（当然，数据抓取不一定用Logstash、Beats）；Kibana负责数据可视化（当然，数据可视化不一定用Kibana）

### ElasticSearch底层实现：Lucence

ElasticSearch的底层实现用到了Lucene。Lucene是一个Java语言的搜索引擎类库（简单来说就是jar包），是Apache公司的顶级项目。

#### Lucene优势：

1：**易扩展**，只是实现了搜索引擎相关核心api，可以基于此做二次开放、定制，实现更高级的功能；

2：**高性能（基于倒排索引）**。

Lucene的缺点：

1：**只限于java语言开发**；

2：学习曲线陡峭；

3：不支持水平扩展。

#### ElasticSearch就是基于Lunece做的二次开发，优点是：

1：**天生就支持分布式，可水平扩展**（就是支持集群，所以支持海量数据和高并发场景）；

2：**提供Restful接口，跟语言无关**，可被任何语言调用，使用场景更加广泛。

总结：

Lunece是1999年，ElasticSearch是2010年问世，中间相隔十几年，所以，基于Lunece的搜索引擎并不只有ElasticSearch，其中还有比较出名的Solr，只不过ElasticSearch比较优秀，所以用的越来越多。

#### 小总结

什么是ElasticSearch?

一个开源的分布式搜索引擎，可以用来实现搜索、日志统计、分析、系统监控等功能

什么是elastic stack（ELGK）

是以ElasticSearch为核心的技术栈，包括beats、logstash、kibana、elasticsearch

什么是Lunece？

是Apache的开源的基于java的搜索引擎类库，提供了搜索引擎的核心api。

### 正向索引和倒排索引

ElasticSearch底层是基于Lunece来实现的，而**Lunece中的核心技术就是倒排索引**，倒排索引是与传统数据库，例如MySQL中正向索引去对比得出的名称，可见倒排索引一定和数据库的正向索引有比较大的区别。

#### 正向索引：

传统数据库（如mysql）采用的正向索引，例如常见的很多表中的id创建索引。形成一颗B+树，那么根据id检索的速度就会变得非常快。这种方式的索引就是正向索引，但是如果搜索的字段不是id，而是一个普通字段，因为普通字段一般不会加索引，即便加了索引，但是如果搜索的不是精确的字段值，搜索的是部分字段值，那么就得使用模糊匹配（like），而一但使用了模糊匹配，就算字段有索引也不生效，这种情况下（没有索引），就会采用逐条扫描的方式来判断表中每一行是否满足，不包含则丢弃，包含则把记录放入结果集。此时如果表的数据过多（如1000万条），那么就要扫描1000万次，全表扫描，性能可想而知地差。**正向索引做这种字段局部内容检索的时候效率比较差。**

#### 倒排索引：

文档document：每一条数据就是一个文档，参照数据库表中的一条记录；

词条term：文档按照语义分成的词语（重点是词语），如果是中文就按中文含义分词，如：华为小米充电器，可以分成（华为、小米、充电器、充电、电器），分出来得每个词语就是一个词条。

倒排索引创建的时候会形成一张新的表（倒排索引表），这张表里面有两个字段，一个字段是词条term，一个字段是文档id。也就是说倒排索引在存储是会先把文档中的内容分成词条去存，比如说一个文档中包含标题title（华为小米充电器），要对标题title创建倒排索引，那么就要把标题内容做分词，得到五个词语：华为、小米、充电器、充电、电器，这个时候就把分词得到的每个词条存到表中的term，然后记录对应的文档id。有多少个词语就存多个词条；如果新的文档分词后有想相同的词语，因为词条不能重复，那么就在对应词条的再加上文档的id，以此类推。所以不管是有四个文档，还是有四千万个文档，将来一定都可以分成一个一个的词条，并且这些词条肯定会有大量的重复，但是我们不用重复记录相同词条，只需要记录一个，同时记录重复文档id即可，这样可以确保倒排索引当中词条term字段是绝对不会出现重复的，因此，因为词条的唯一性，我们就可以给词条term创建索引了，将来根据词条查找文档的速度就相当快了。

比如我们搜索“华为手机”，第一步会把“华为手机”进行分词，得到“华为”、“手机”两个词条，然后去倒排索引表中进行查询，因为倒排索引表中根据词条建立了索引，所以根据“华为”、“手机”这两个词条去查询的速度是很快的，然后查找这两个词条对应的文档id，于是就会得到两组文档id，这个时候就会知道包含“华为”，“手机”词条的所有文档了，将来还可以根据文档id出现的频率进行排序，然后根据文档id就可以去正向索引表查询文档，因为正向索引表根据id建立了索引，那么就可以快速地定位到对应的文档。

综上，可以发现，倒排索引的过程其实是经过两次检索，第一次是根据用户输入内容的词条去词条列表中找到对应的文档id；第二次是根据文档id去找文档，虽然是两次，但每一次的检索都是通过索引去找的，所以搜索效率是很高的，是比全表扫描的效率要高得多得多的。同时也能看出来了，倒排索引之所以叫倒排索引，因为在正向索引中是逐行去查找，是先找到文档，看文档是否符合词条要求；而倒排索引是反过来的，是基于词条去创建索引，然后去关联到对应的文档，查找的时候是先去找到对应的词条，再根据词条找到对应的文档，即倒排索引是根据词找文档，而正向是是根据文档匹配词。倒排索引适合、擅长基于文档的部分内容进行搜索，即更复杂的搜索需求，比如浏览器关键字搜索，商城商品搜索，都是复杂的搜索场景。

#### 小总结：

什么是文档和词条：

每一条数据就是一个文档；

对文档中的内容分词，得到的词语就是词条

什么是正向索引：

基于文档id创建索引，查询词条时必须先找到文档，而后判断是否包含词条

什么是倒排索引：

对文档内容分词，对词条创建索引，并记录词条所在文档的信息，查询时先根据词条查询到文档id，而后获取到文档

### ElasticSearch概念：

#### 文档Document：

ElasticSearch是面向文档存储的，可以是数据库中的一条商品数据，一个订单信息。文档数据会被序列化为JSON格式后存储在elasticsearch中。类似数据库中表的行（Row），即一条记录。

#### 字段Field：

就是Json文档中的字段，类型数据库中表的列（Column）

#### 索引Index：

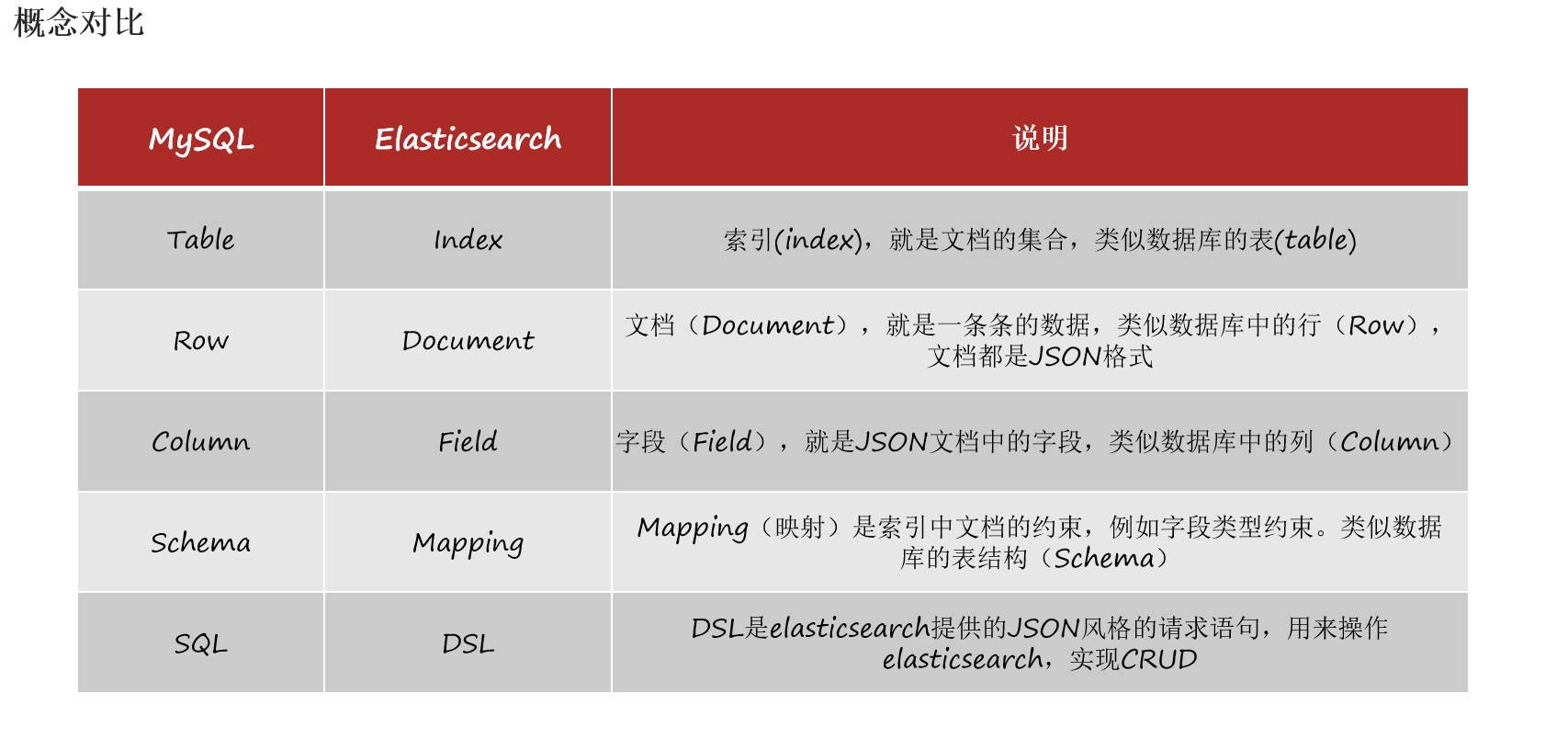
相同类型的文档的集合（可视为数据库中的一张表Table）。

#### 隐射Mapping：

索引中文档的字段约束信息（即字段名字叫什么、数据类型是什么。。），类似数据库中表的结构约束Schema。

#### DSL：

dsl是elasticsearch提供的**JSON风格的请求语句**，用来操作elasticsearch，实现CRUD，所以dsl和sql是一样的，就是一种语法，这种语法就是用来描述要干的事，在Mysql中，sql写好了通过Connection把语句发给MySQL，MySQL就能进行相应的处理，在ElasticSearch中，写好DSL后，要怎么发出去呢，是通过基于HTTP请求发出去的，因为ES对外暴露的Restful的接口。



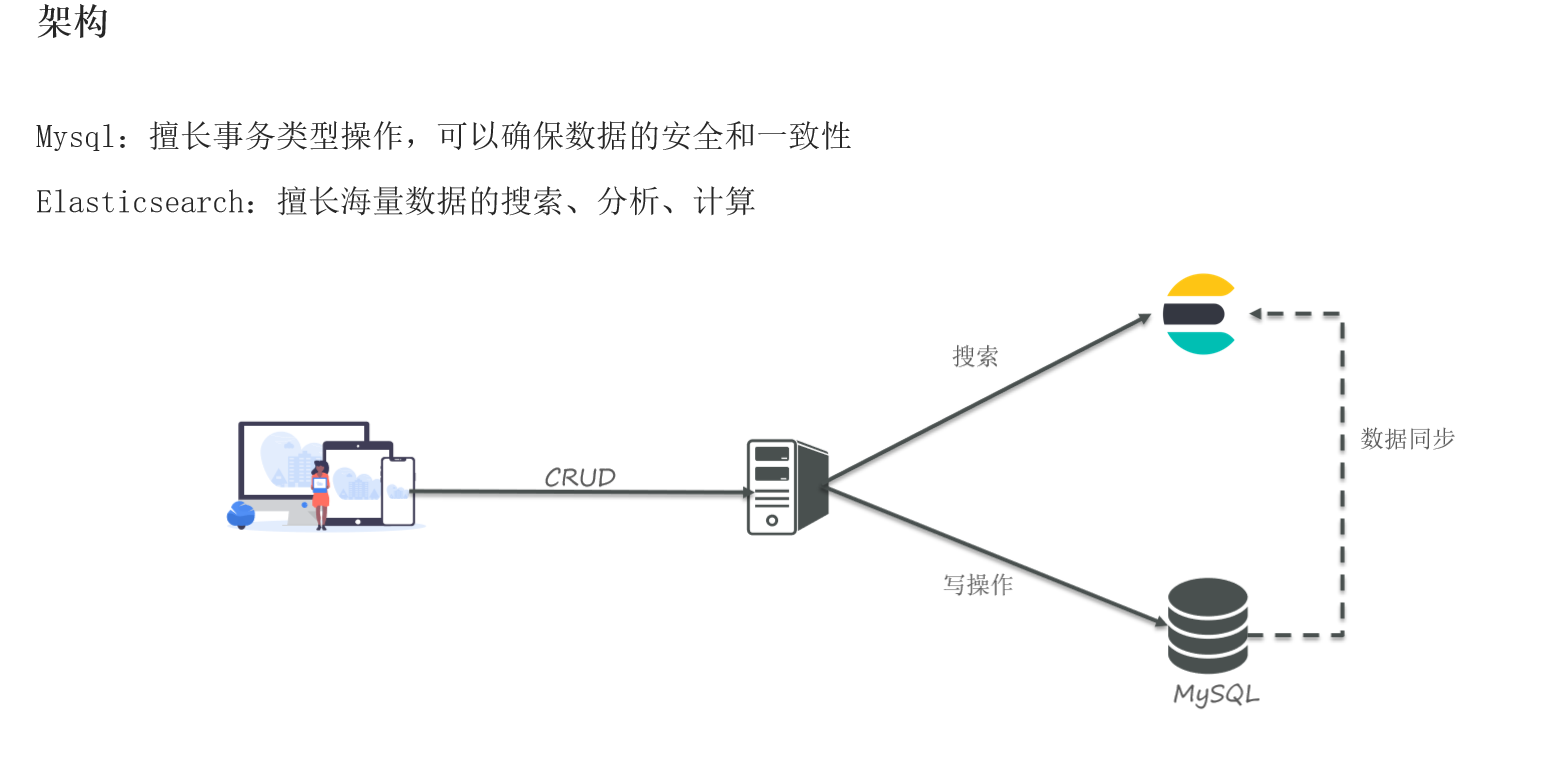
所以将来操作ES的时候，和操作数据库相同的道理，第一件事是先去创建索引库，即索引Index，因为创建索引，那么就要指定约束映射Mapping，用来约束索引中的字段“长什么样子”，然后添加文档Document。然后是文档的CRUD。在ES中，写好DSL通过HTTP的请求发送过去给ES。

Mysql和ES怎么选？

Mysql：擅长事务类型操作，因为事务有ACID原则，可以确保数据的安全和一致性；

ES：擅长海量数据的搜索、分析、计算。

两者是互补、不是替代关系！



#### 小总结：

文档：一条数据就是一个文档，es中是Json格式

字段：json文档中的字段

索引：同类文档的集合

银色：索引中文档的约束，比如字段名称、类型

### 安装ElasticSearch、kibana

为什么要安装kibana，是因为kibana中提供了dev tools工具，可以让我们非常方便去编写ElasticSearch中的DSL语句，从而去操作ElasticSearch，这就是我们要安装kibana的原因

#### 单点安装ElasticSearch

##### 1：因为我们还需要部署kibana容器，因此需要让es和kibana容器互联。这里先创建一个网络：

docker network create es-net

##### 2：加载镜像，上传elasticsearch镜像的tar包到虚拟机中，然后运行命令加载即可

docker load -i es.tar

##### 3：运行docker命令，部署单点es：

docker run -d \

--name es \

-e "ES\_JAVA\_OPTS=-Xms512m -Xmx512m" \

-e "discovery.type=single-node" \

-v es-data:/usr/share/elasticsearch/data \

-v es-plugins:/usr/share/elasticsearch/plugins \

--privileged \

--network es-net \

-p 9200:9200 \

-p 9300:9300 \

elasticsearch:7.12.1

命令解释：

-e "cluster.name=es-docker-cluster"：设置集群名称

-e "http.host=0.0.0.0"：监听的地址，可以外网访问

-e "ES\_JAVA\_OPTS=-Xms512m -Xmx512m"：内存大小，默认1G，不能低于512M

-e "discovery.type=single-node"：非集群模式，即es是单点模式运行

-v es-data:/usr/share/elasticsearch/data：挂载逻辑卷，绑定es的数据目录

-v es-logs:/usr/share/elasticsearch/logs：挂载逻辑卷，绑定es的日志目录

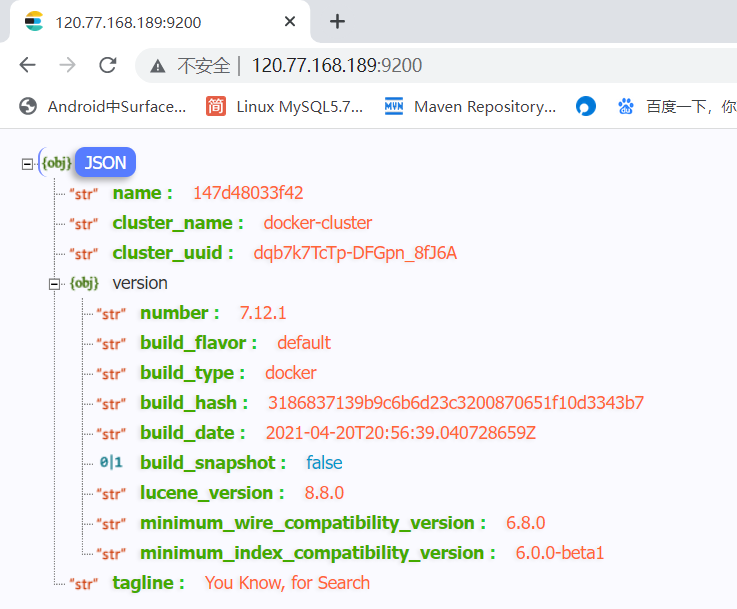
-v es-plugins:/usr/share/elasticsearch/plugins：挂载逻辑卷，绑定es的插件目录

--privileged：授予逻辑卷访问权

--network es-net ：加入一个名为es-net的网络中

-p 9200:9200：端口映射配置

##### 4：在浏览器中输入：http://120.77.168.189:9200 即可看到elasticsearch的响应结果：



#### 部署kibana

kibana的版本一定要与elasticSearch的版本保持一致！

##### 1：加载镜像，上传kibana镜像的tar包到虚拟机中，然后运行命令加载即可

Docker load -i kibana.tar

##### 2：运行docker命令，部署kibana

docker run -d \

--name kibana \

-e ELASTICSEARCH\_HOSTS=http://es:9200 \

--network=es-net \

-p 5601:5601 \

kibana:7.12.1

命令解释：

--network es-net ：

加入一个名为es-net的网络中，与elasticsearch在同一个网络中，所以kibana容器和elasticsearch可以用容器名互联，kibana可以帮助我们去操作es

-e ELASTICSEARCH\_HOSTS=http://es:9200"：

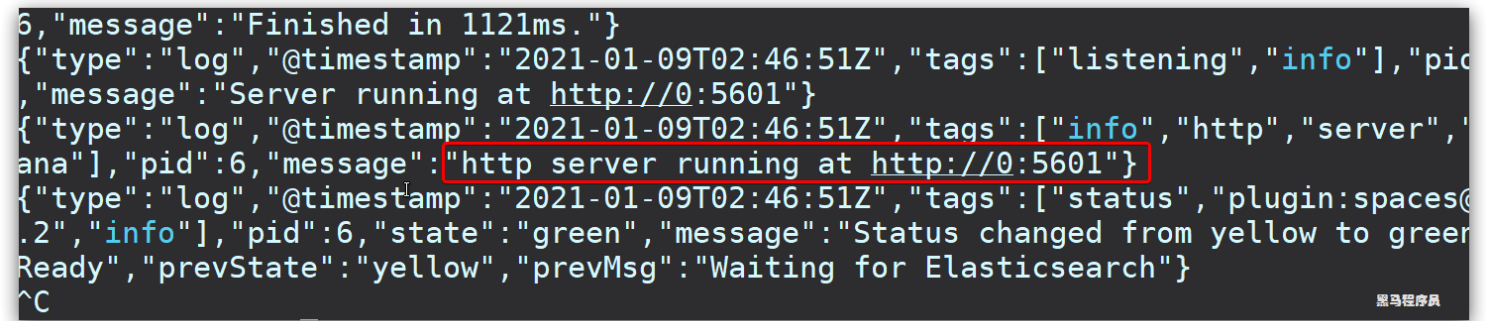
设置elasticsearch的地址，因为kibana已经与elasticsearch在一个网络，因此可以用容器名直接访问elasticsearch

-p 5601:5601：kibana端口映射配置

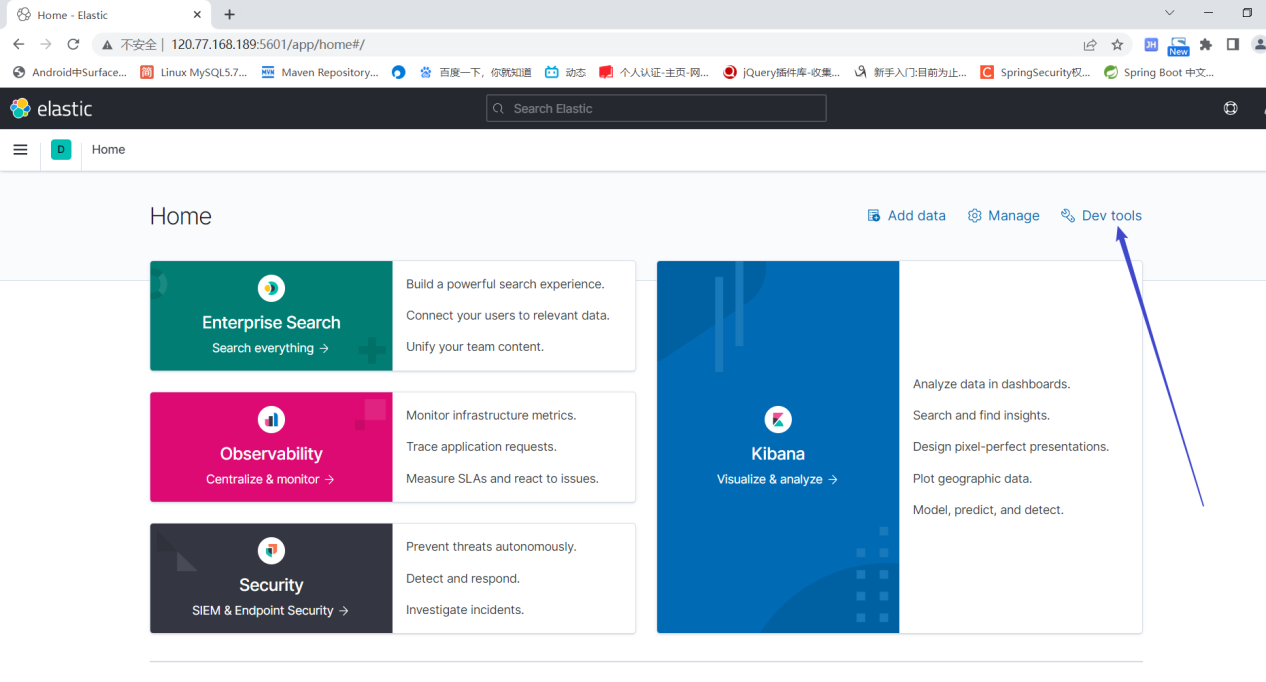
##### 3：kibana启动一般比较慢，需要多等待一会，可以通过命令：

docker logs -f kibana

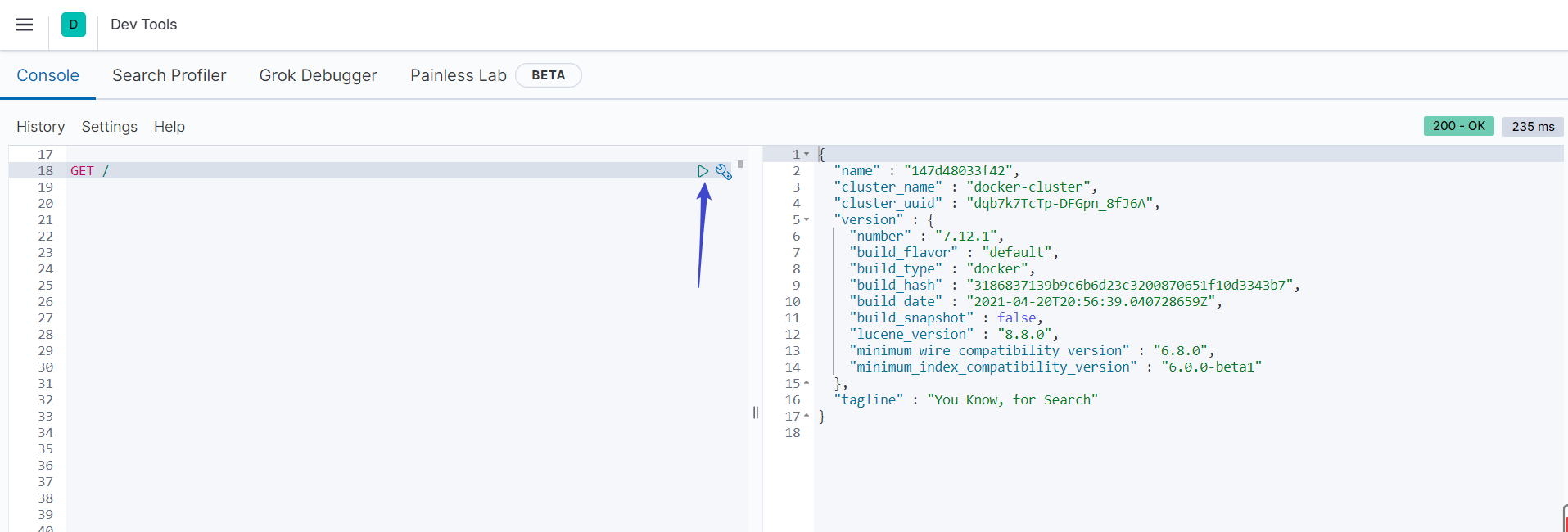
查看日志，看到有类似一下，说明运行kibana成功



##### 4：访问kibana，在浏览器输入地址访问：http://120.77.168.189:5601，即可看到结果



##### 在kibana模拟http://120.77.168.189:9200/：**GET /**



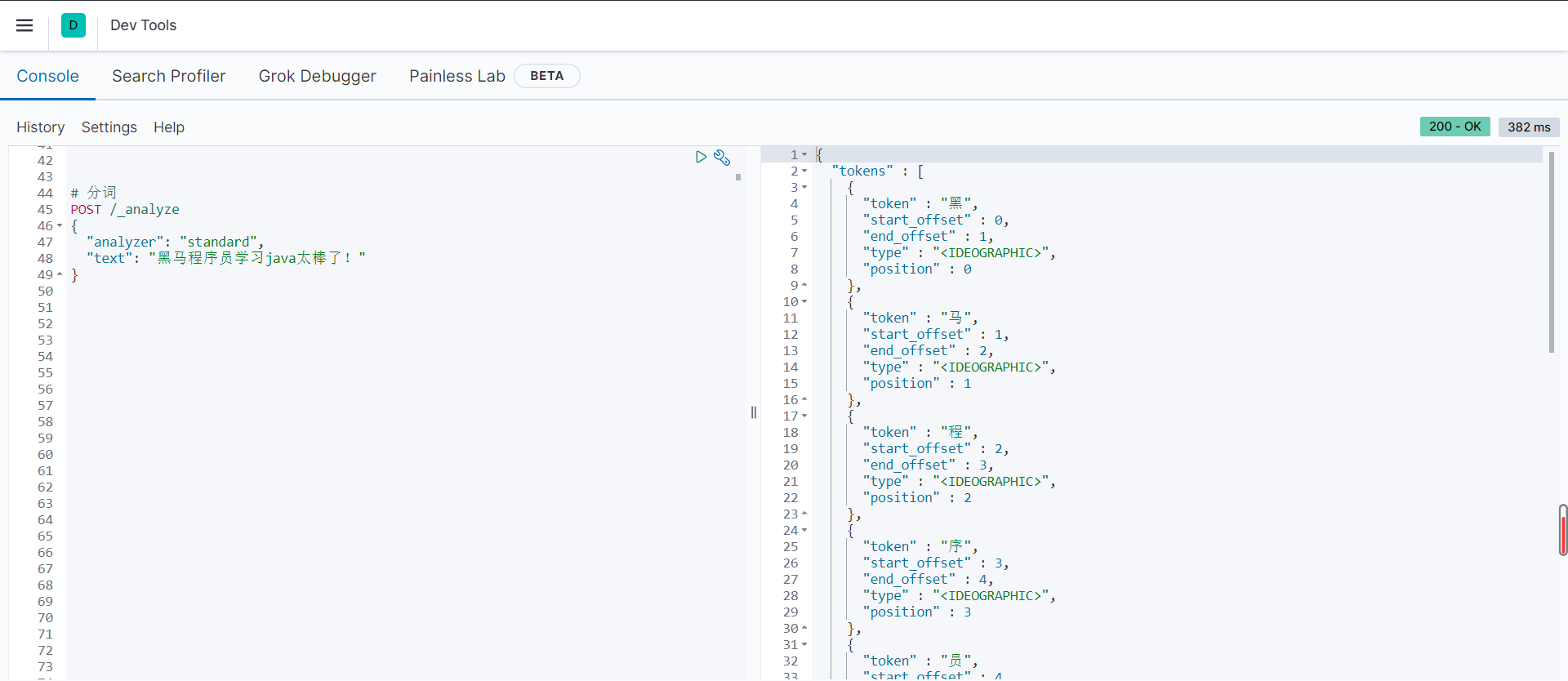
### 安装分词器

ElasticSearch在创建倒排索引时需要对文档内容进行分词，而且在搜索时，需要对用户输入内容分词。分词就需要对中文、英文的语义进行分析，按照词语含义进行拆分，这种拆分是比较复杂的，需要特殊的算法去支持，在ES中但默认的分词规则对中文处理并不友好。

我们可以在kibana的DevTools中测试：



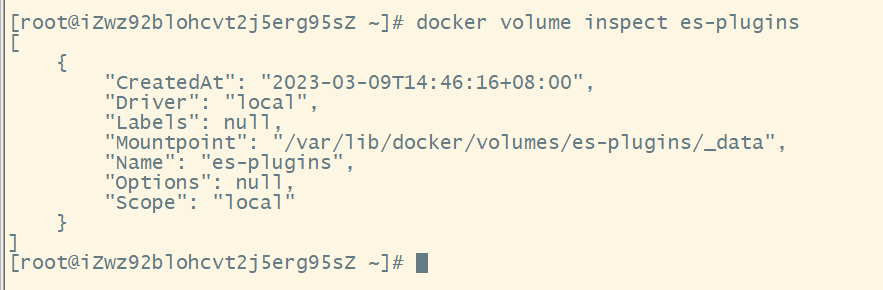
从下图结果可以看到，ES的默认分词器对中文会逐字分词！显然不符合，所以要对中文分词，必须替换默认的分词器：IK分词器。



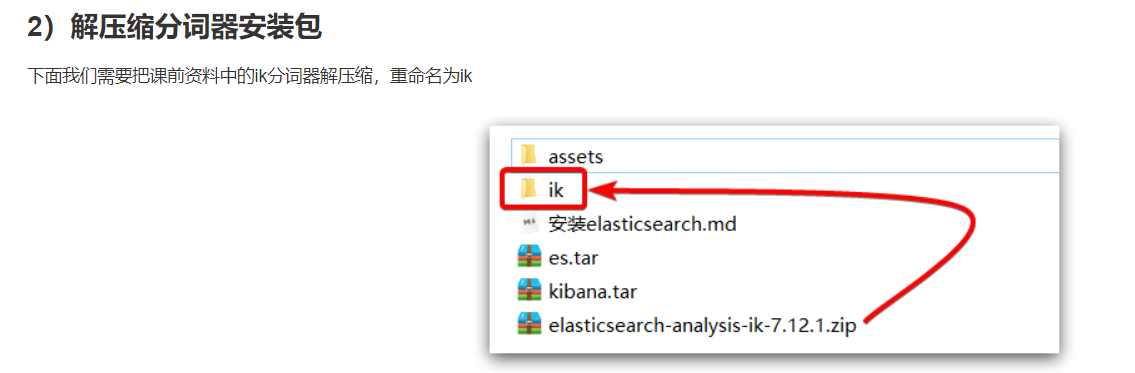
#### 离线安装IK分词器

##### 1：查看插件数据卷目录：docker volume inspect es-plugins

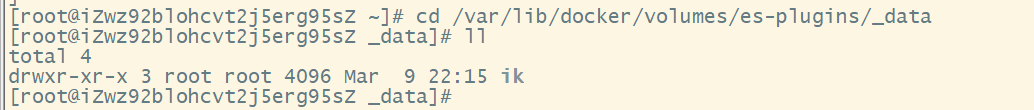
装插件需要知道elasticsearch的plugins目录位置，而我们用了数据卷挂载，因此需要查看elasticsearch的数据卷目录，通过下面命令查看:



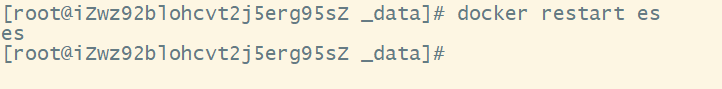
##### 2：解压IK分词器安装包



##### 3：用xftp将解压后的ik分词器文件夹上传到es-plugins插件数据卷中



##### 4：重启es容器：docker restart es



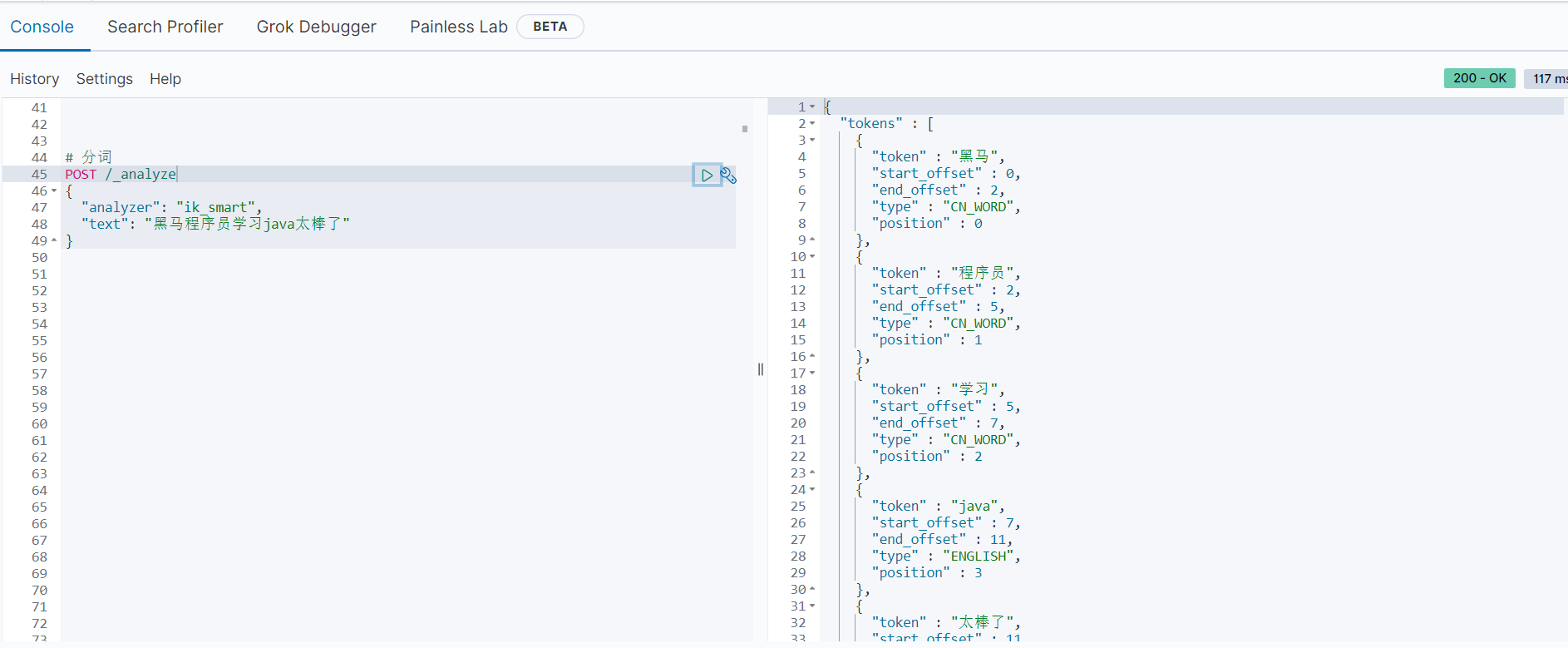
##### 5：查看是否安装IK分词器成功：查看es日志，docker logs -f es

##### 6：测试IK分词器

IK分词器包含两种模式：

ik\_smart：最少切分，粗粒度，占用内存少，搜索命中概率较低

ik\_max\_word：最细切分，细粒度，占用内存多，搜索命中概率高



#### IK分词器的扩展和停用字典

##### iK分词器原理

IK分词器分词的原理是什么：在底层一定有一个字典，在字典里一定会有各种各样的词语，提前罗列好了，当要分词的时候，一定会把要分词的内容拿去字典匹配，看字典里有没有，如果匹配到在词典里有，那么说明是一个词，那就分词出来，所以任何中文分词器，都会依赖于一个字典来去做分词。

字典是不会包含全世界所有的词，甚至是中文词也不可能全部包含，因为一是太多，二是词一直在发展变化，不停有很多新词出来，譬如网络用语“你XX”、“潘噶之交”、“白嫖”、“老六”、“鸡你太美”。

那么问题来了，我们要怎么去拓展分词器的字典，比如把“你XX”、“老六”、“鸡你太美”这些词加到字典中；另外对一些无意义的词可以不去分词，即停用，譬如：“的”、“呢”。。。。，因为分词了会占用内存；还有一些敏感词汇、禁忌词，这些也应该禁掉。

以上即是对分词的字典做个性化的设置。

##### IK分词器扩展词库、停用词库：

要扩展IK分词器的词库，只需要**修改IK分词器插件目录中**的config目录中的IKAnalyzer.crg.xml文件，目录如下：

**/var/lib/docker/volumes/es-plugins/\_data/ik/config**

如下图：



###### 1：配置自己的拓展字典和拓展停止词字典：**ext.dic stopword.dic**

**<?xml version="1.0" encoding="UTF-8"?>**

**<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">**

**<properties>**

**<comment>IK Analyzer 扩展配置</comment>**

**<!--用户可以在这里配置自己的扩展字典 -->**

**<entry key="ext\_dict">ext.dic</entry>**

**<!--用户可以在这里配置自己的扩展停止词字典-->**

**<entry key="ext\_stopwords">stopword.dic</entry>**

**<!--用户可以在这里配置远程扩展字典 -->**

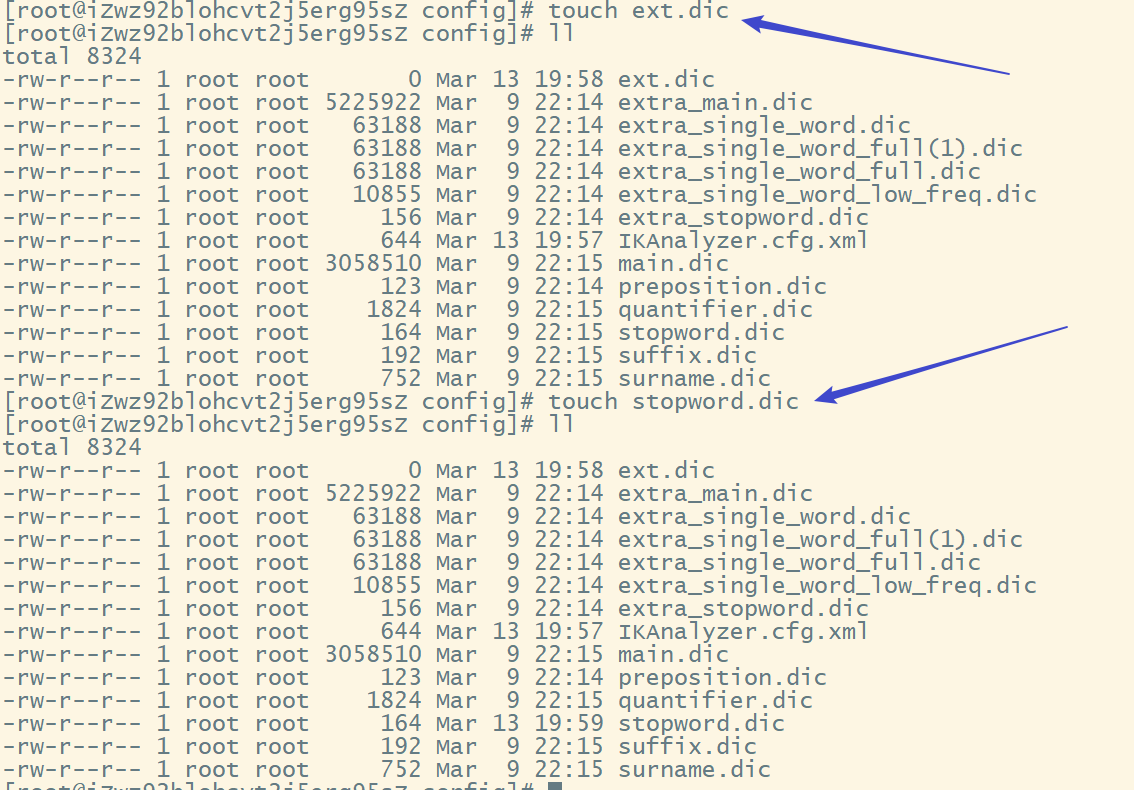
**<!-- <entry key="remote\_ext\_dict">words\_location</entry> -->**

**<!--用户可以在这里配置远程扩展停止词字典-->**

**<!-- <entry key="remote\_ext\_stopwords">words\_location</entry> -->**

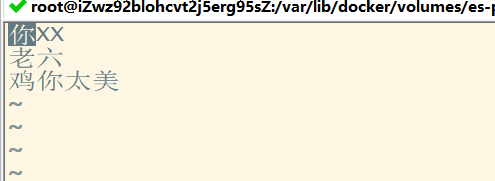
**</properties>**

###### 2：在config中创建ext.dic文件和stopword.dic文件（stopword.dic这个文件好像本来就有了）

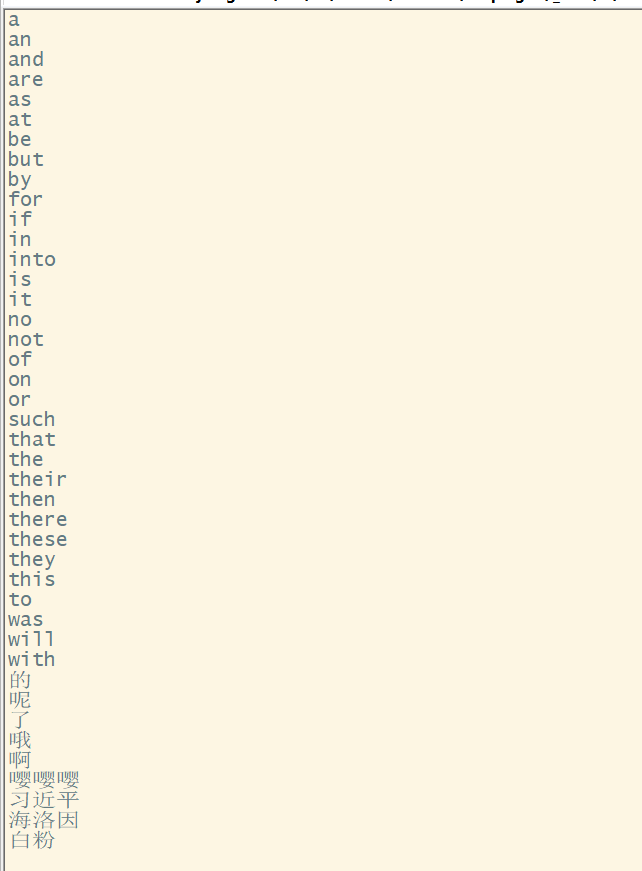


###### 3：在文件中添加想要拓展或停止的词语：vi ext.dic vi stopword.dic

ext.dic：



stopword.dic：



###### 4：重启elasticsearch：docker restart xx，重启需要过程，需要耐心等待一会

#### 小总结：

分词器的作用是什么？

两个场景需要用分词：1是将文档创建倒排索引的时候，要对文档的某些内容进行分词，2是当用户进行搜索的时候，要对输入的搜索内容进行分词，即：

1：创建倒排索引时对文档分词；

2：用户搜索时，对输入的内容分词。

IK分词器有几种模式？

1：Ik\_smart：智能切分，粗粒度

2：Ik\_max\_word：最细切分，细粒度

IK分词器如何拓展词条？如何停用词条？

1：利用config目录中的IKAnalyzer,cfg.xml文件添加拓展词典和停用词典

2：在词典中添加拓展词条或停用词条

### 索引库操作

我们先回顾一下，我们知道在es中有索引库的概念，索引库就相当于数据库中的表，而是索引库里有很多的文档，文档就相当于表中一行一行的记录，我们知道，数据库要先创建表，才能去添加记录，es也一样，先得有了索引库，才能往里面去添加文档，因此，我们要先来学习一下索引库的操作，再去学习文档的操作。

要想去创建一个索引库，就像去建表一样：在建表语句里要去指定Schema，即是对字段的约束。同样，在创建索引库时，需要去指定对应的mapping 映射：是对文档的约束。

#### Mapping映射：

Mapping映射是对索引库中文档的约束（比如字段什么类型啊？是否要索引啊？是否要分词啊？如果分词的话使用什么分词器啊），常见的mapping属性包括：

##### Type：字段的数据类型关键字，常见的简单类型有：

1：字符串：test（可分词的文本，例如：个人介绍、商品标题等）、keyword（精确值，例如：品牌、国家、IP地址、邮箱等）。所以，以后去设置字段映射时，一定要想清楚这个字段需不需要拆，不需要就是keyword，需要就是text；

2：数值类型：long、integer、short、byte、double、float

3：布尔：boolean

4：日期：date

5：对象：object

注意：es中是没有数组这中类型的，但是，允许你某个类型的字段有多个值，也就是说，只需要关注数组里元素的类型，如图中的数组的类型就是float或double

##### Index：是否创建索引，即是否要创建倒排索引，默认为true，则会为字段创建倒排索引，那么就可以参与搜索了，实际开发中，并不需要每个字段都需要去搜索，比如商品图片url地址、邮箱，没有搜索意义，那么因此，在创建字段映射时，一定要判断这个字段是否参与不参与搜索，不参与要设置成false。（注意：是否索引和是否分词没有必然的关系，但是要分词了的字段一般都是要做索引，但做索引的字段不一定分词）

##### Analyzer：使用哪种分词器，对内容做分词的，可想而知，这个属性用得比较少，因为在上面的数据类型里，只有一个数据类型需要分词：text，其他类型都无需分词，因此，这个Analyzer属性是结合text类型一起使用，值就是分词器的名称。

##### Properties：该字段的子字段，即object对象类型中子属性



#### 小总结：

##### Mapping的常见属性有哪些：

Type：数据类型；

Index：是否索引；

Analyzer：分词器

Properties：子字段

##### Type常见的字段有哪些：

字符串：text、keyword

数字：long、integer、short、byte、double、float

布尔：Boolean

日期：date

对象：object

#### 创建索引库

ES中通过restful请求操作索引库、文档。请求内容用DSL语句来表示、创建索引库和mapping的DSL语法如下：



总的来说就是：

**PUT /索引库名称**

**{**

**“mappings”: {**

**“properties”: {**

**“字段名”: {**

**“type”: “XXX”,**

**“analyzer”: “XXX”,**

**“index”: “ture or false”**

**}**

**}**

**}**

**}**

# GET表示请求方式；\_search表示请求资源 下面的json表示请求数据

GET /\_search

{

"query": {

"match\_all": {}

}

}

GET \_search

{

"query": {

"match\_all": {}

}

}

#模拟http://120.77.168.189:9200/

GET /

# 分词器

# 默认分词器，对中文支持很差，会逐字分词

POST /\_analyze

{

"analyzer": "standard",

"text": "黑马程序员学习java太棒了！"

}

# ik\_smert分词器 粗粒度

POST /\_analyze

{

"analyzer": "ik\_smart",

"text": "黑马程序员学习java太棒了"

}

POST /\_analyze

{

"analyzer": "ik\_smart",

"text": "哈登，我想对你说：你是我的神！"

}

# ik\_max\_word 细粒度

POST \_analyze

{

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

"text": "黑马程序员学习java太棒了"

}

POST /\_analyze

{

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

"text": "今天的天气好晴朗啊，处处good风光！广东粤安的员工黄远辉说的哦，好想白嫖啊"

}

# 创建索引库、映射mapping

##### # 创建heima索引库

PUT /heima

{

"mappings": {

"properties": {

"info": {

"type": "text",s

"analyzer": "ik\_smart"

},

"email": {

"type": "keyword",

"index": false

},

"name": {

"type": "object",

"properties": {

"firstName": {

"type": "keyword"

},

"lastName": {

"type": "keyword"

}

}

}

}

}

}

##### # 创建student索引库

PUT /student

{

"mappings": {

"properties": {

"info": {

"type": "text",

"analyzer": "ik\_smart"

},

"email": {

"type": "keyword",

"index": false

},

"grade": {

"type": "integer"

},

"age": {

"type": "integer"

},

"gender": {

"type": "keyword"

},

"address": {

"type": "text",

"analyzer": "ik\_smart"

},

"name": {

"type": "keyword"

}

}

}

}

#### 查看索引库：GET /索引库名

##### # 查询student索引库

GET /student

##### # 查询heima索引库

GET /heima

GET heima

#### 修改索引库（只支持增加属性）： PUT /索引库名/\_mapping

**PUT /索引库名/\_mapping**

**{**

**“properties”: {**

**“新字段名”: {**

**“type”: “XXX”,**

**。。。**

**}**

**}**

**}**

解释：\_mapping代表要修改的是索引库隐射

事实上，在es中，索引库是不允许修改的，因为索引库创建完了以后，她的数据结构，即索引库的mapping映射都已经定义好了，我们的ES会基于mapping映射去创建倒排索引，那么如果说，你要去修改索引库的一个字段的话，就会导致原有的整个倒排索引就会彻底失效，这样一来带来的影响是特别巨大的，于是在ES里面，是禁止修改索引库的，这点和数据库不一样，数据库中，一张表创建完了，是可以去修改表结构的（理论上可以，但是在生产环境下，数据库也是避免去修改的表结构的）。

但是，ES虽然禁止你直接去修改索引库原有的字段，但是允许你去增加新字段，这也是属于修改嘛

##### # 修改heima索引库

PUT /heima/\_mapping

{

"properties": {

"age": {

"type": "integer",

"index": false

}

}

}

#### 删除索引库：DELETE /索引库名

##### # 删除student索引库

DELETE /student

##### # 删除heima索引库

DELETE /heima

### 文档操作

上面已经搞定了索引库，库有了，那么就可以往库里面添加数据，并进行数据的操作

#### 添加文档

新增文档的DSL语法如下（如果不加文档id，es会认为你添加的文档没有id，就会给你随机生成一个id，所以一定别忘了加文档id）：

**POST /索引库名/\_doc/文档id**

**{**

**“字段1”: “值1”,**

**“字段2”: ”值2”,**

**“字段3”: {**

**“子属性1”: ”值3”,**

**“子属性2”: “值4”**

**},**

**// 。。。。。。**

**}**

# 新增文档 (注意加id，否则es会默认生成id)

POST /heima/\_doc/1

{

"info": "黑马程序员Java学员信息",

"email": "2278181701@qq.com",

"name": {

"firstName": "远辉",

"lastName": "黄"

},

"age": 28

}

#### 查询文档：GET /索引库名/\_doc/文档id

GET /heima/\_doc/1

#### 修改文档

##### # 方式一：全量修改，会删除旧文档，添加新文档

**PUT /索引库名/\_doc/文档id**

**{**

**“字段1”: “值1”,**

**“字段2”: “值2”**

**// 。。。。**

**}**

可以看到，和添加文档请求几乎一样的命令，唯一的区别就是，添加请求是POST，修改请求时PUT。为什么方式一是全量修改，因为会根据文档id去索引库找到对应的旧的文档，然后把就文档删掉，接着再把新文档添加进去，等于是旧的全盘否定，再加新的，所以叫做全量修改，那么如果传的文档id在索引库中本来就不存在，此时和新增文档。

可以认为这个方式一既可以做修改文档，又可以做新增文档：如果id在索引库中存在则修改，不存在则是新增

PUT /heima/\_doc/1

{

"info": "黑马程序员的Java学员信息",

"email": "2278181701@qq.com",

"name": {

"firstName": "远辉",

"lastName": "黄"

},

"age": 28

}

##### # 方式二：局部修改（增量修改，只修改指定字段值，别的不改）

**POST /索引库名/\_update/文档id**

**{**

**“doc”: {**

**“字段名”: “新的值”**

**}**

**}**

POST /heima/\_update/1

{

"doc": {

"age": 29,

"name": {

"firstName": "YuanHui",

"lastName": "Huang"

}

}

}

#### 删除文档： DELETE /索引库名/\_doc/文档id

DELETE /heima/\_doc/1

#### 小总结

文档操作有哪些？

1：创建文档： POST /索引库名/\_doc/文档id {json文档}

2：查询文档：GET /索引库名/\_doc/文档id

3：删除文档：DELETE/索引库名/\_doc/文档id

4：修改文档：

4.1：全量修改：PUT /索引库名/\_doc/文档id {json文档}

4.1：增量修改：POST /索引库名/\_update/文档id {“doc”: {字段}}

### RestClient操作索引库

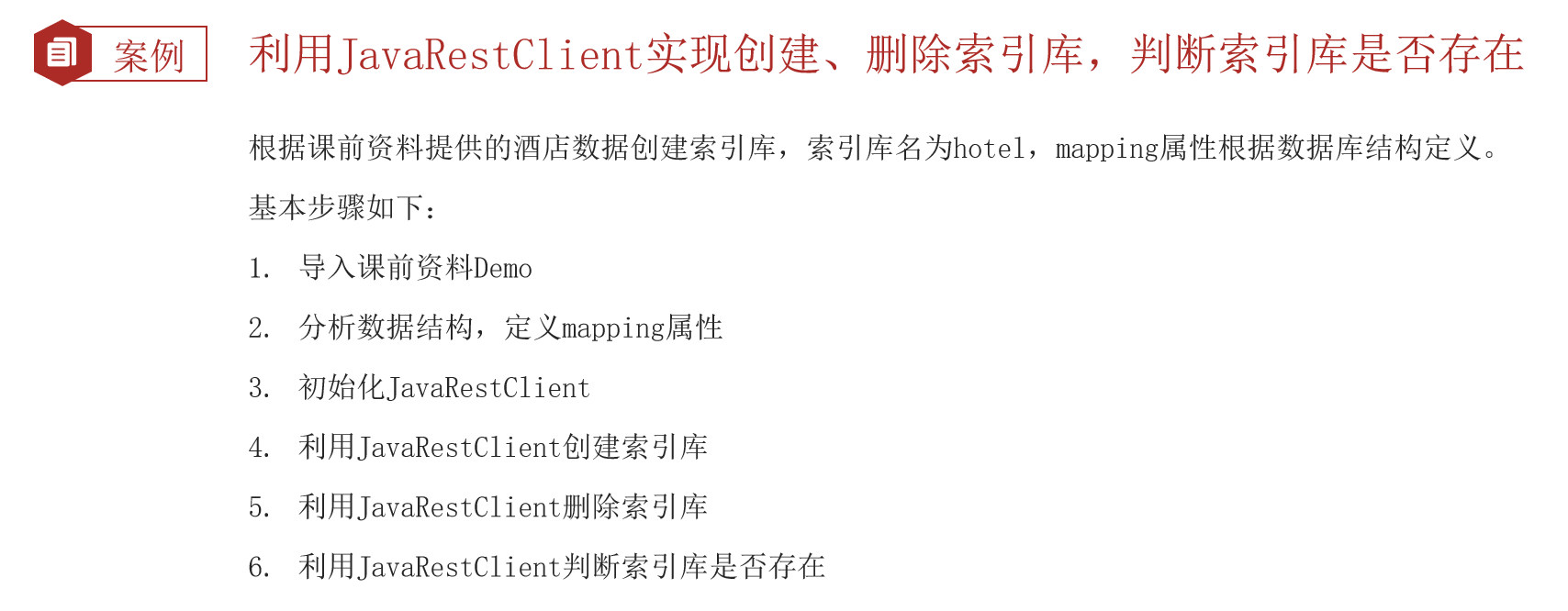
前面已经学习了如何使用DSL语句去操作ES，包括索引库操作、文档操作，不过，我们作为一名java程序员，肯定需要使用java代码去操作ES，要想实现java代码去操作ES，就必须使用ES官方提供的RestClient来实现。

#### 什么是RestClient

是ES官方提供了各种不同语言的客户端，用来操作ES。这些客户端的本质就是组装DSL语句，通过HTTP请求发送给ES，java的客户端如下图：



#### 使用restClient



#### 编写mapping映射：分析数据结构：mapping要考虑的问题：

字段名叫什么?

字段数据类型是什么?

字段是否参与搜索?

字段是否分词？

字段如果分词，分词器是什么？

其中字段名和数据类型我们可以基于数据库的表结构得出，字段名一目了然，数据类型则可以转为ES中对应的数据类型；至于是否搜索以及是否分词这两个就比较特殊了，则是与业务强相关的，比如当前是酒店业务，那么酒店名称一定是要参与搜索的字段，所以酒店名这个字段的index应该是true，即参与搜索，分词显然也是要分词的，分词器为了提高搜索概率，可以选用ik\_max\_word，综上，是与业务相关的，要根据业务场景综合考虑。

例如，下图中的tb\_hotel表，我们去分析下面的字段，去编写映射mapping。



映射分析过程如下：

首先id的数据类型不能是long，而应该是字符串，又因为id是一个整体，所以是keyword，因为索引库id这个字段比较特殊，是字符串类型；

name酒店名称显然一定是要参与搜索的字段，所以字段的index应该是true，即参与搜索，分词显然也是要分词的，分词器为了提高搜索概率，可以选用ik\_max\_word;

address这个字段，因为一般不会根据地址去搜酒店，所以没有分词的必要，那么type应该选择为keyword，又因为不搜索，那么index的值不能为默认true，应该为false；

price价格、score评分显然是要参与搜索的，数据类型显然是integer;

brand品牌、city城市、startName星级、business商圈显然是要参与搜索的，同时不需要分词，所以类型是keyword；

latitude和longtime经纬度地理坐标，既不是float也不是double，因为地理坐标在ES中比较特殊，有特殊的表现方式，有两种表现方式：geo\_point：由维度（latitude）和经度（longtime）确定地图上的一个点。例如”32.8752345, 120.2981576”；get\_shape：有多个geo\_point组成复杂几何图形，代表地图上的一个区域。例如一条直线：”LINESTRING (-77.03653 38.897676, -77.009051 38.889939)”。显然酒店应该是地图上的一个点，即应该是geo\_point就可以了，由上看出值是由经度和维度拼在一起的字符串，但是类型是叫geo\_point，所以，可以创建一个叫location，类型是geo\_point的字段。

pic是酒店图片，是一个url路径，显然不需要分词，即type为keyword，其次没有人会根据url去搜索，显然index为false，即既不分词也不搜索。

##### 字段拷贝

至此，我们就定义完酒店的所有字段了，但是，会发现有一个问题，即映射中的name酒店名、brand品牌、business商圈、city城市、startName星级这些字段都要参与搜索，也就意味着将来用户输入关键字去搜的时候，可能要根据多个字段去搜，也就是你的查询条件不是一个值，而是多个值，可以想一下，在用ES去做搜索的时候，是根据一个字段搜效率高，还是根据多个字段搜效率高，可以对比一下数据就知道了，显然是根据一个字段，但是现在的需求就是希望用户输入酒店名称能搜到、用户输入品牌能搜到、用户输入商圈也能搜到，即就是要根据多个字段，同时又要搜索性能好，那该怎么办？ES给我们提供了copy\_to属性来解决：字段拷贝可以使用copy\_to属性将当前字段拷贝到指定字段，比方说此处想根据name酒店名、brand品牌、business商圈、city城市、startName星级这些字段去搜，那么就可以把这些字段都拷贝到一个字段：all里面，all字段的类型显然是text，因为是输入的搜索内容是可以多个字段，分词器因为要提高搜索命中率，显然可以用ik\_max\_word，此时，意味着all字段就同时具备了name酒店名、brand品牌、business商圈、city城市、startName星级这些字段的值，也就是说实现了在一个字段里搜索到多个字段的内容，而且这种拷贝ES做了优化，并不是真的把文档拷贝进去了，而只是基于此做了倒排索引，所以在查的时候是看不到all这个字段的，好像不存在一样，但是搜却可以根据all字段去搜，这个all字段又叫复合字段。示例如下：

# 创建案例项目映射

PUT /hotel

{

"mappings": {

"properties": {

"id": {

"type": "keyword"

},

"name": {

"type": "text",

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

"copy\_to": "all"

},

"address": {

"type": "keyword",

"index": false

},

"price": {

"type": "integer"

},

"sorce": {

"type": "integer"

},

"brand": {

"type": "keyword",

"copy\_to": "all"

},

"city": {

"type": "keyword"

},

"starName": {

"type": "keyword"

},

"business": {

"type": "keyword",

"copy\_to": "all"

},

"location": {

"type": "geo\_point"

},

"pic": {

"type": "keyword",

"index": false

},

"all": {

"type": "text",

"analyzer": "ik\_max\_word"

}

}

}

}

#### 初始化JavaRestClient步骤

1：引入ES的RestHighLevelClient依赖；

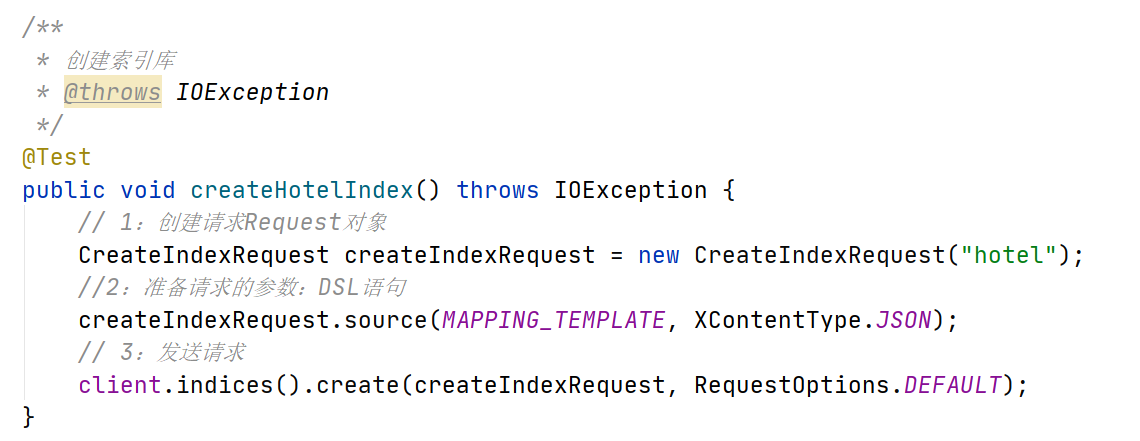
2：因为SpringBoot默认的ES版本是7.6.2，所以我们要覆盖默认的ES版本，必须和ES的版本保持一致；

3：初始化RestHighLevelClient

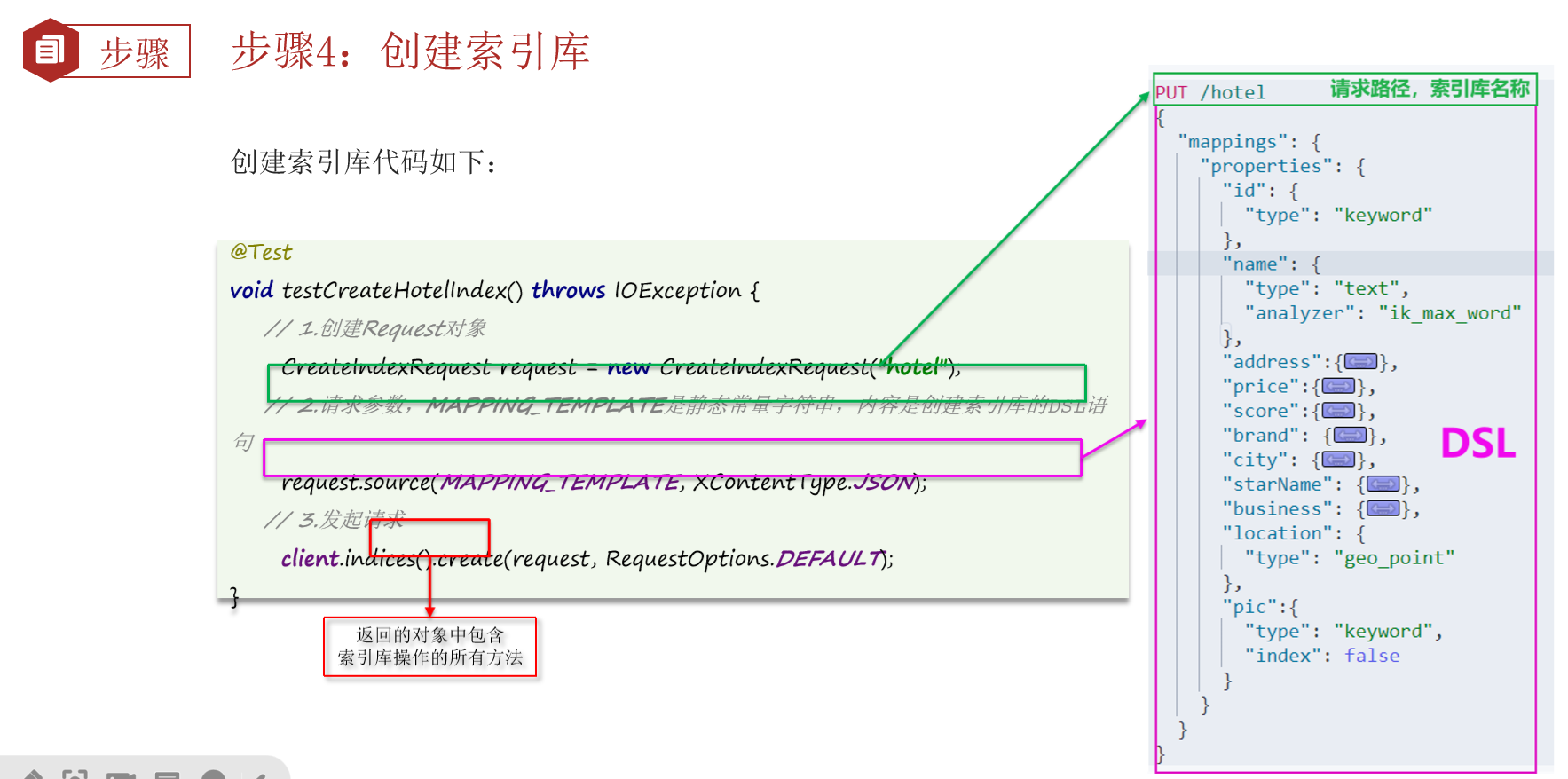


#### 创建索引库：

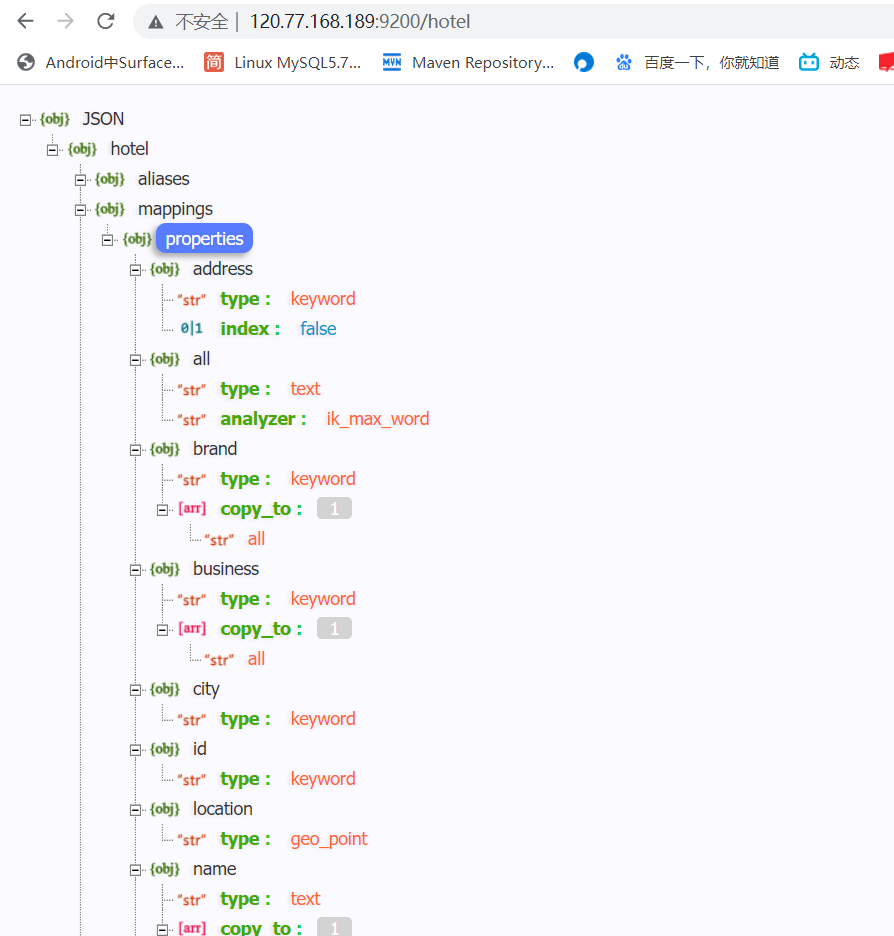
创建索引库的代码如下：



以上代码可以结合DSL语句，更便于理解，java代码做的就是用java代码来组织DSL，然后去发请求：

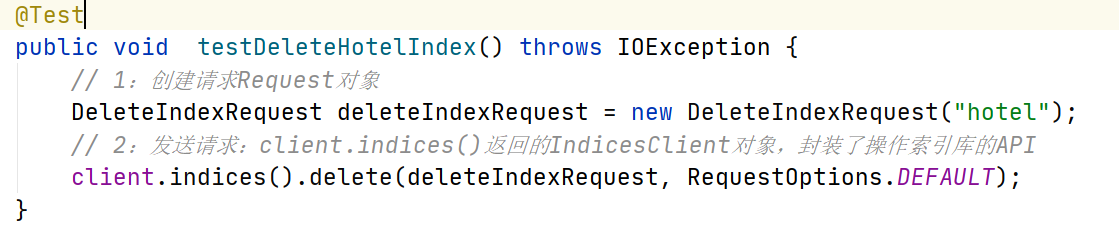


**在浏览器中查询是否成功：GET /hotel 或者 http://120.77.168.189:9200/hotel**



#### 删除索引库：

删除索引库代码如下：



#### 判断索引库是否存在：

判断索引库是否存在代码如下：



#### 小总结：

索引库操作的基本步骤：

1：初始化RestHighLevelClient

2：创建XxxIndexRequest对象，XXX分别是CREATE、Get、Delete

3：准备DSL（当然了，只有CREATE时需要）

4：发送请求。调用ResetHighLevelClient#indices().xxx()方法，xxx是create、exists、delete

### RestClient操作文档

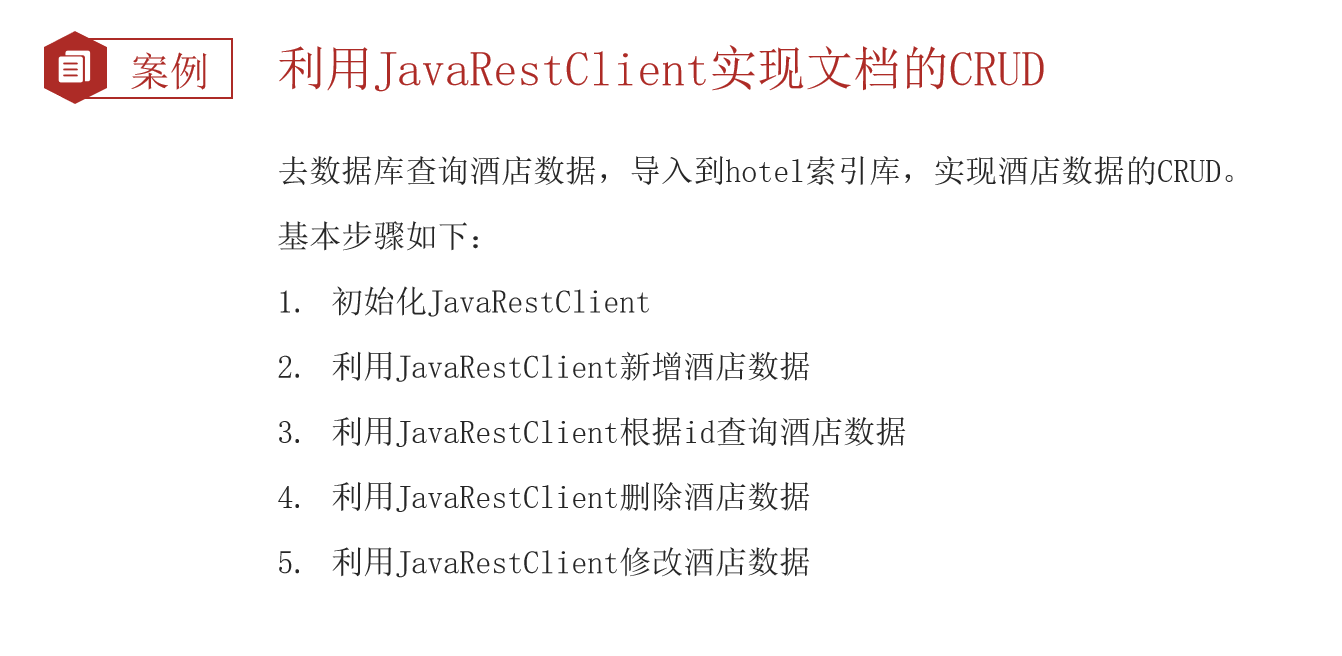
新增文档；

查询文档；

删除文档；

修改文档；

批量导入文档；



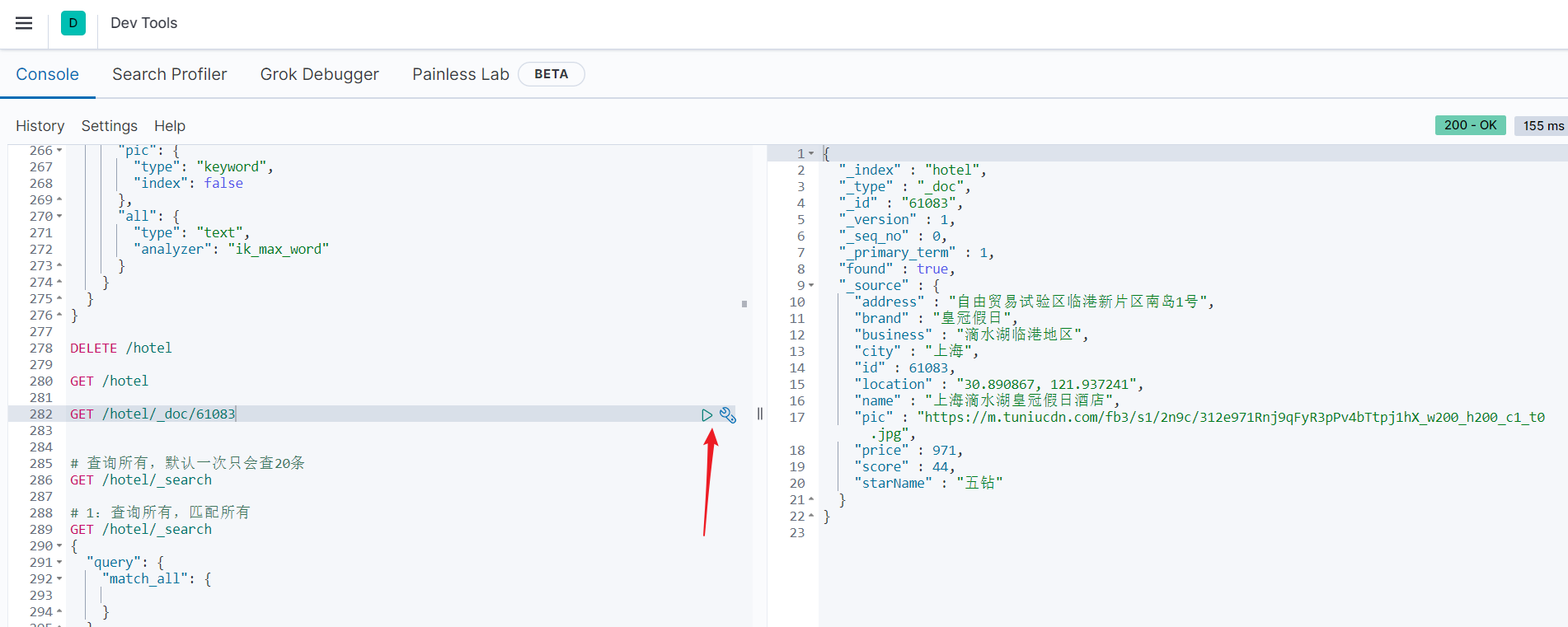
#### 新增文档

例子：添加酒店数据到索引库：先查询酒店数据，然后给这条数据创建倒排索引，即可完成添加

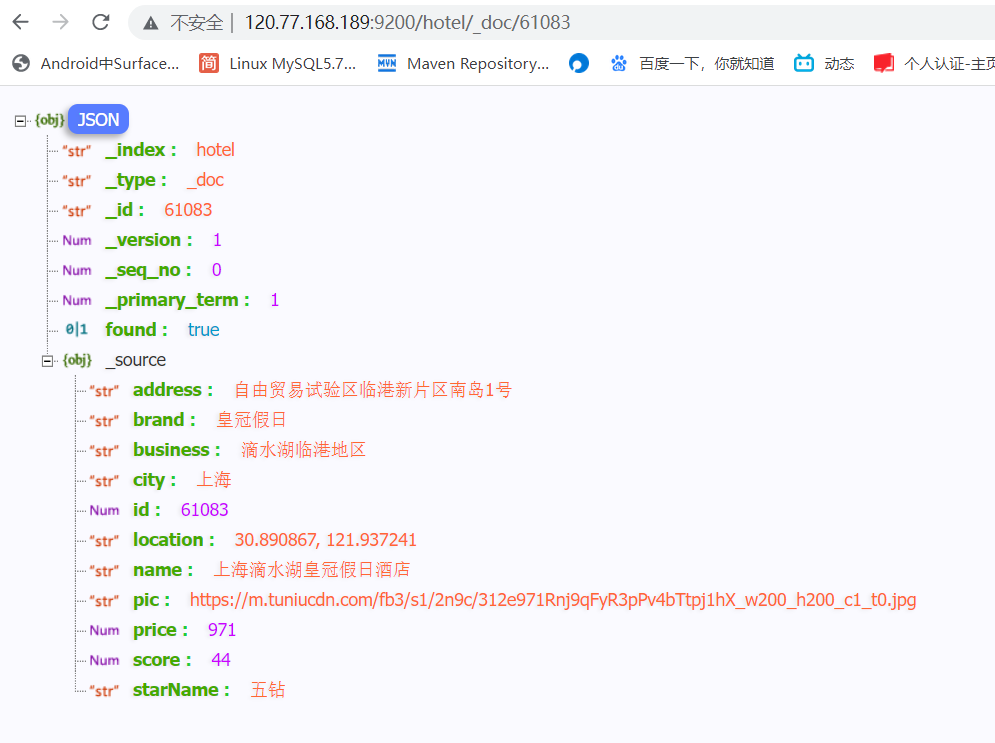


新增文档代码如下：

在浏览器中验证是否成功添加文档进索引库：GET /hotel/\_doc/61083

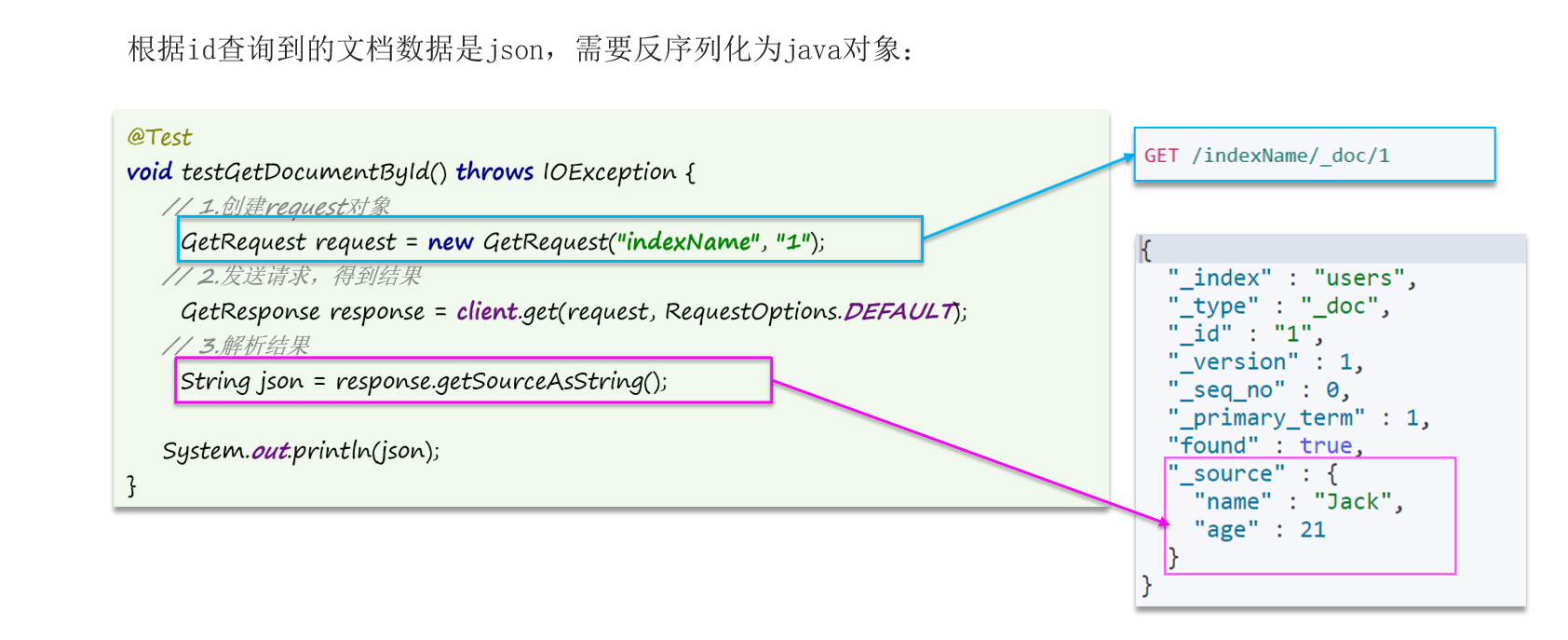


或者：http://120.77.168.189:9200/hotel/\_doc/61083

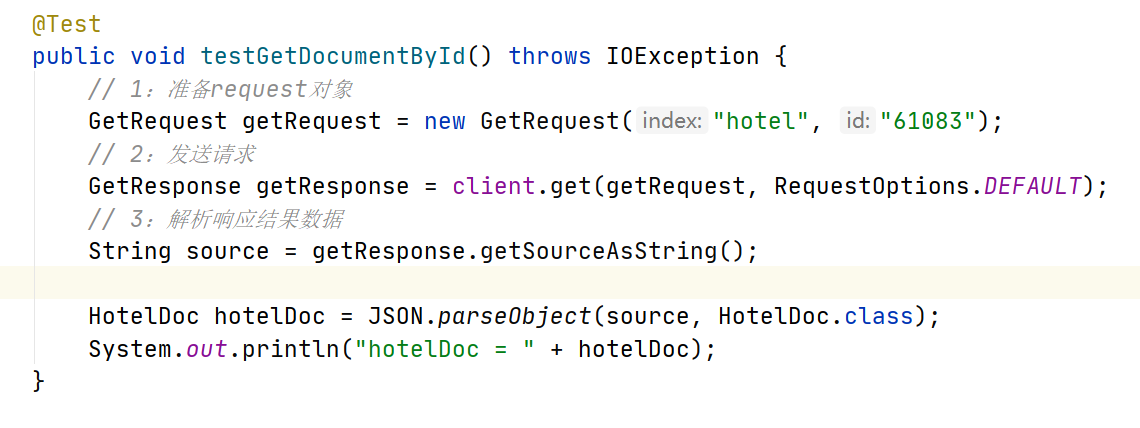


#### 查询文档

响应结果的解析，应该解析的是图中红框框住的信息：即原始的文档信息，返回的是JSON格式字符串。



查询文档代码如下：

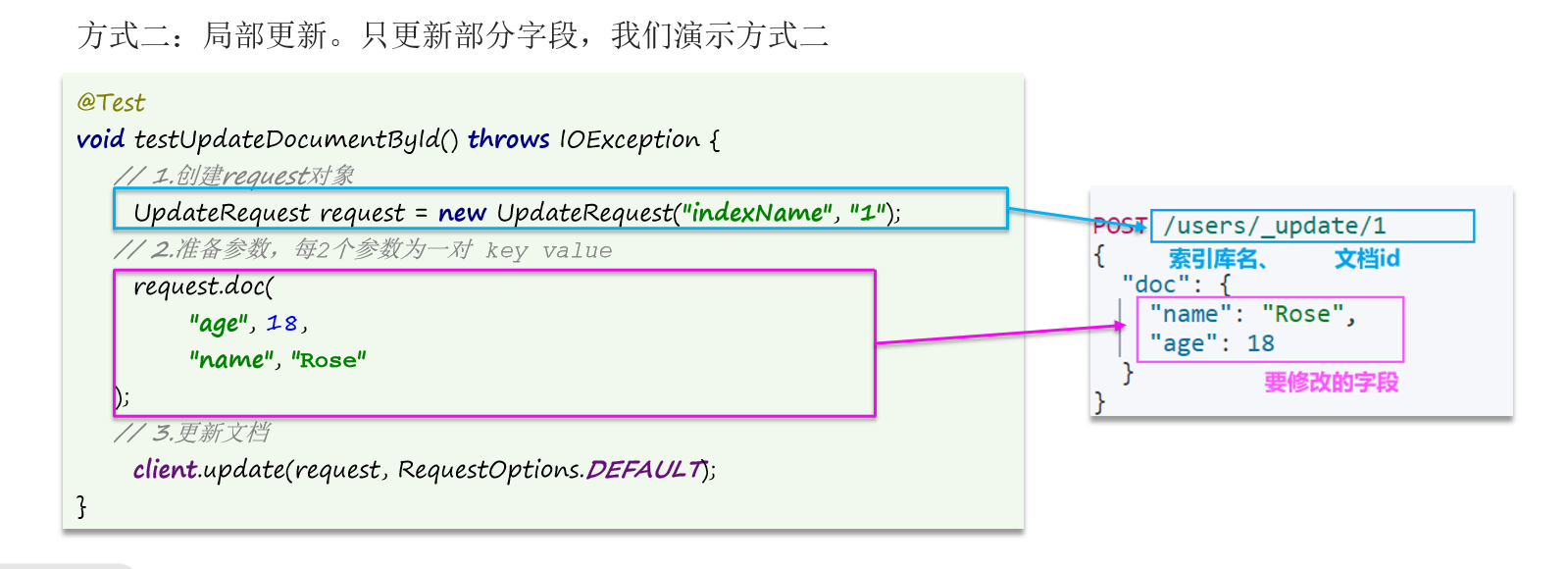


#### 修改文档

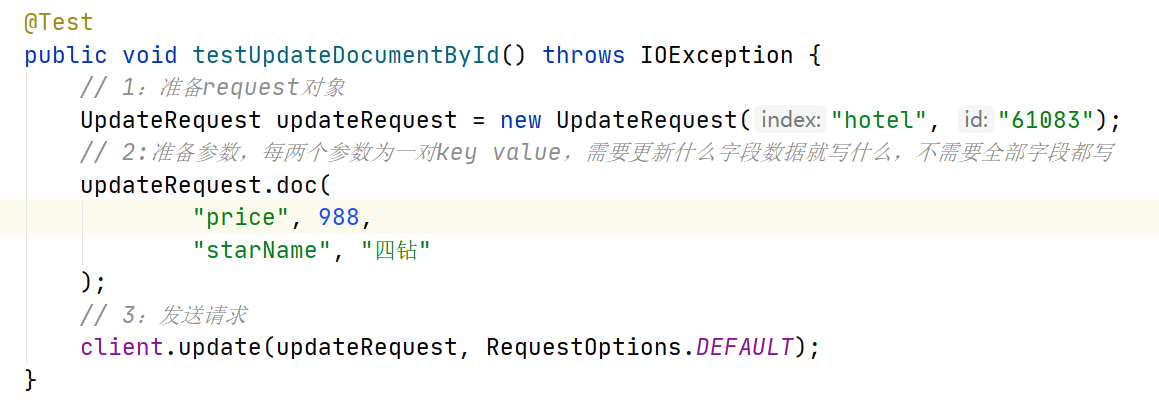
修改文档数据有两种方式：

方式一：全量更新。再次写入与id一样的文档，就会删除旧文档，添加新文档，如果没有，就直接插入新文档，java代码与之前的新增文档没有区别。此处不做演示。

方式二：局部更新。只更新部分字段，我们演示方式二：



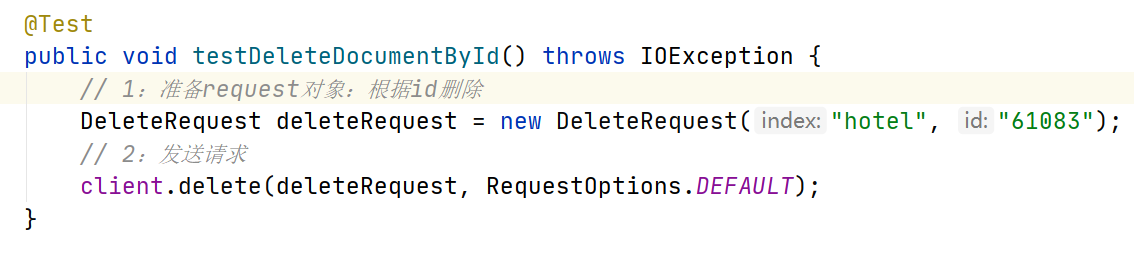
局部更新文档代码如下：



#### 删除文档



删除文档代码如下：



#### 小总结：

文档操作的基本步骤：

1：初始化RestHighLevelClient;

2：创建XxxRequest。XXX是Index、Get、Update、Delete；

3：准备参数（当然只有在新增Index和更新Update时需要）；

4：发送请求。调用的时RestHighLevelClient#xxx()方法，xxx是index、get、update、delete

5：解析结果（当然只有查询Get时需要）

#### 文档批量操作

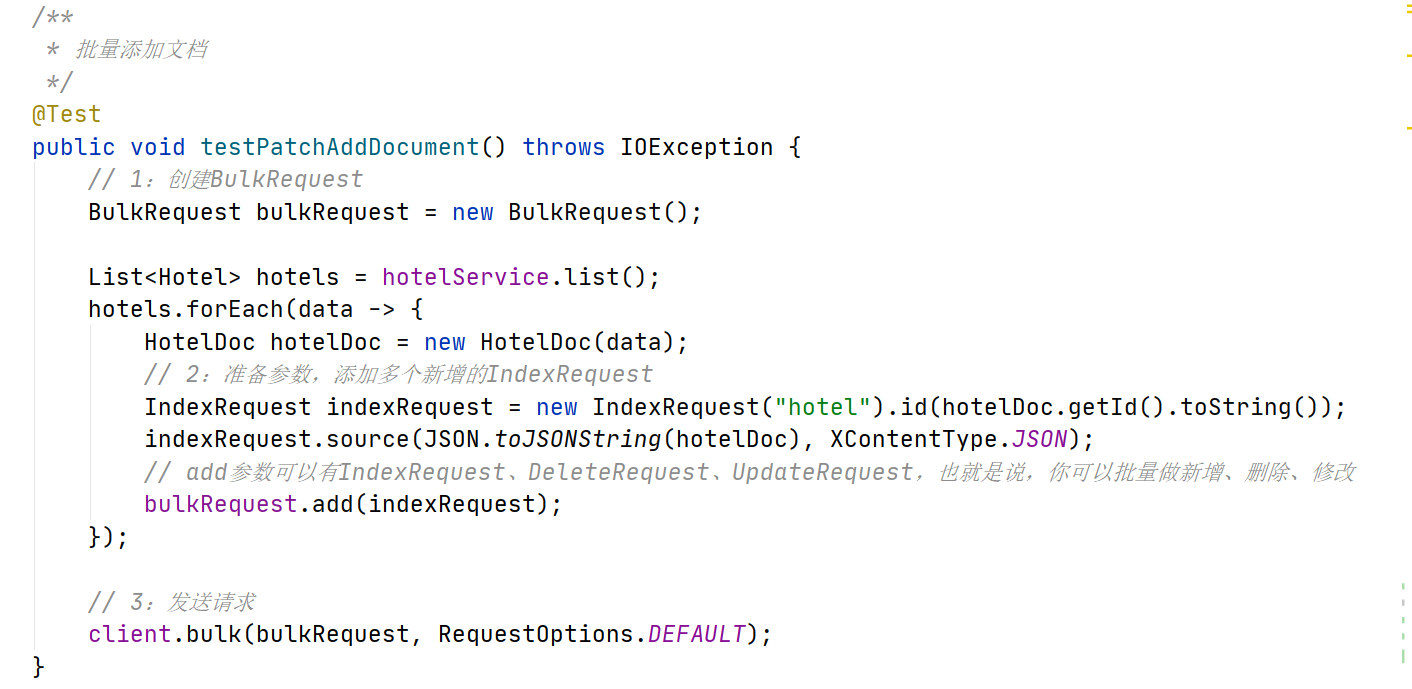
之前操作文档都是一次操作一个，显然在实际场景中不合理，往往需要批量操作：批量导入、批量删除等

##### 批量导入

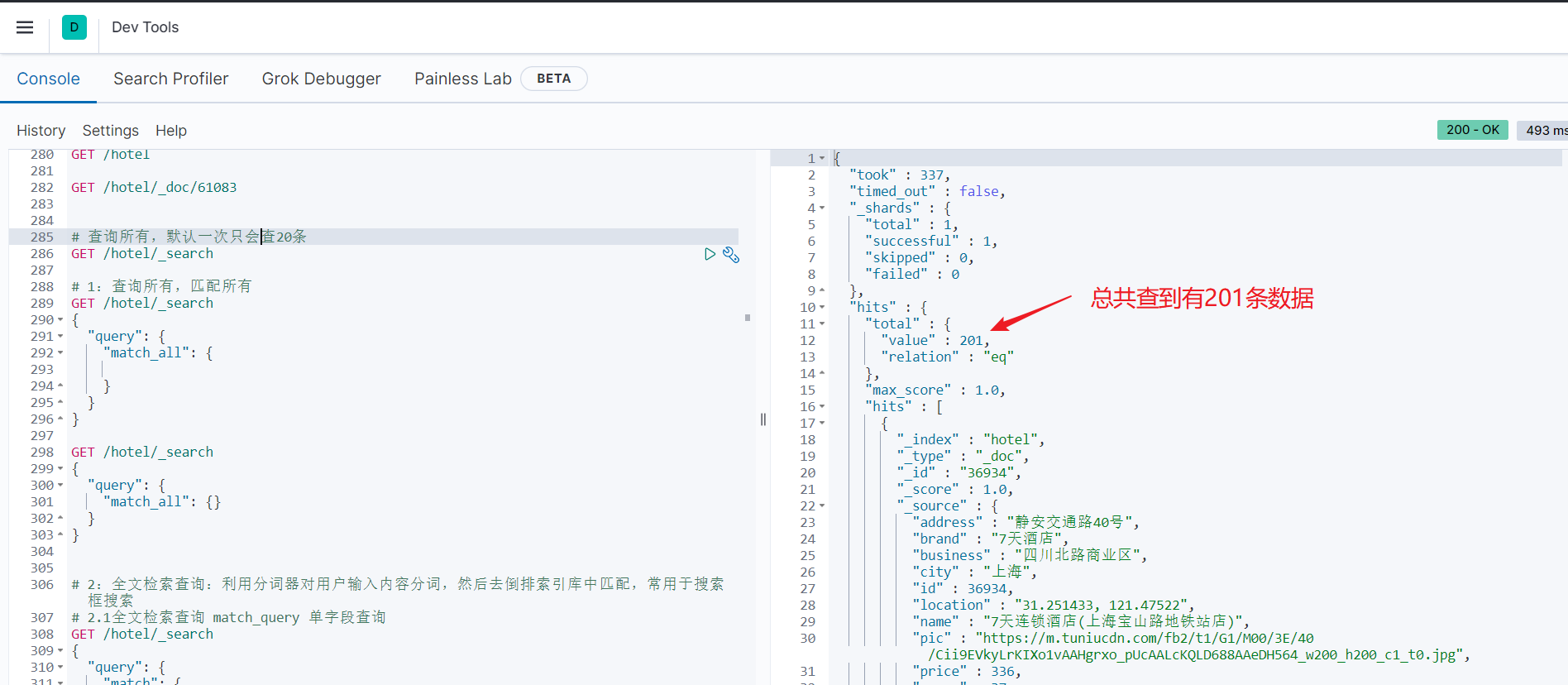
如下图可看到，批量导入文档是把新增请求IndexRequest创建出来，添加add到BulkRequest中，即把多个新增请求IndexRequest和合并到BulkRequest中，然后一次性完成提交。



批量新增文档代码如下：



在浏览器中批量查询，默认一次只会查20条：GET /hotel/\_search



或者：http://120.77.168.189:9200/hotel/\_search：



# 1：查询所有，匹配所有

GET /hotel/\_search

{

"query": {

"match\_all": {

}

}

}

GET /hotel/\_search

{

"query": {

"match\_all": {}

}

}

# 2：全文检索查询：利用分词器对用户输入内容分词，然后去倒排索引库中匹配，常用于搜索框搜索

# 2.1全文检索查询 match\_query 单字段查询

GET /hotel/\_search

{

"query": {

"match": {

"all": "如家外滩"

}

}

}

GET /hotel/\_search

{

"query": {

"match": {

"all": "如家外滩"

}

}

}

#2.2 multi\_match\_query 全文检索查询：多匹配查询 多字段查询 此时和上面结果一致，推荐用上面的，因为参与搜索的字段越多，效率越低

GET /hotel/\_search

{

"query": {

"multi\_match": {

"query": "如家外滩",

"fields": ["brand", "name", "business"]

}

}

}

GET /hotel/\_search

{

"query": {

"multi\_match": {

"query": "如家外滩",

"fields": ["brand", "name", "business"]

}

}

}

# 3：精确查询 查不分词的字段 如：keyword bool 数值 日期等

# 3.1：精确匹配 词条查询

GET /hotel/\_search

{

"query": {

"term": {

"city": {

"value": "上海"

}

}

}

}

GET /hotel/\_search

{

"query": {

"term": {

"brand": {

"value": "如家"

}

}

}

}

# 3.2：范围查询

GET /hotel/\_search

{

"query": {

"range": {

"price": {

"gte": 1000,

"lte": 3000

}

}

}

}

GET /hotel/\_search

{

"query": {

"range": {

"score": {

"gte": 40,

"lte": 45

}

}

}

}

# 4：地理查询

# 4.1: 中心点附近查询 distance查询

GET /hotel/\_search

{

"query": {

"geo\_distance": {

"distance": "10km",

"location": "31.21,121.5"

}

}

}

# 4.2：区域范围查询

GET /hotel/\_search

{

"query": {

"geo\_bounding\_box": {

"location": {

"top\_left": {

"lat": 31.1,

"lon": 121.5

},

"bottom\_right": "30.9,121.7"

}

}

}

}

# 5：复合查询：上面的都是简单查询，复合查询可以将其他简单查询组合起来，实现更复杂的搜索逻辑

# 5.1:function\_socre 算分排序 控制排序 百度竞价

GET /hotel/\_search

{

"query": {

"function\_score": {

"query": {

"match": {

"all": "外滩"

}

},

"functions": [

{

"filter": {

"term": {

"brand": "如家"

}

},

"weight": 2

}

],

"boost\_mode": "sum"

}

}

}

# 5.2：复合查询（布尔查询），一个或者多个子查询的组合 参与算分：must should 不参与算分：moust\_not filter。参与算分条件越多，性能越差

GET /hotel/\_search

{

"query": {

"bool": {

"must": [

{

"term": {

"city": {

"value": "上海"

}

}

}

],

"should": [

{"term": {

"brand": {

"value": "皇冠假日"

}

}},

{"term": {

"brand": {

"value": "华美达"

}

}}

],

"must\_not": [

{

"range": {

"price": {

"lte": 500

}

}

}

],

"filter": [

{

"range": {

"score": {

"gte": 45

}

}

}

]

}

}

}

GET /hotel/\_search

{

"query": {

"bool": {

"must": [

{

"match": {

"name": "如家"

}

}

],

"must\_not": [

{

"range": {

"price": {

"gt": 400

}

}

}

],

"filter": [

{

"geo\_distance": {

"distance": "10km",

"location": {

"lat": 31.21,

"lon": 121.5

}

}

}

]

}

}

}

# 6：搜索结果出了：排序 分页 高亮

# 6.1.1：排序 制定了排序，es会放弃默认算分

GET /hotel/\_search

{

"query": {

"match\_all": {}

},

"sort": [

{

"score": {

"order": "desc"

},

"price": "desc"

}

]

}

GET /hotel/\_search

{

"query": {

"match\_all": {}

},

"sort": [

{

"score": {

"order": "desc"

}

},

{

"price": {

"order": "desc"

}

}

]

}

# 6.1.2 地理排序

GET /hotel/\_search

{

"query": {

"match\_all": {}

},

"sort": [

{

"price": "asc"

},

{

"\_geo\_distance": {

"location": {

"lat": 31.034661,

"lon": 121.612282

},

"order": "asc",

"unit": "km"

}

}

]

}

GET /hotel/\_search

{

"query": {

"match\_all": {}

},

"sort": [

{

"\_geo\_distance": {

"location": "31.034661,121.612282",

"order": "asc",

"unit": "km"

}

}

]

}

# 6.3：分页查询

GET /hotel/\_search

{

"query": {"match\_all": {}},

"sort": [

{

"price": {

"order": "asc"

}

}

],

"from": 3,

"size": 3

}

GET /hotel/\_search

{

"query": {

"match": {

"brand": "如家"

}

},

"sort": [

{

"score": "desc"

}

],

"from": 0,

"size": 5

}

#3：高亮，默认情况下，es搜索字段必须和高亮字段一致，否则不会高亮

GET /hotel/\_search

{

"query": {

"match": {

"all": "如家"

}

},

"highlight": {

"fields": {

"name": {

"require\_field\_match": "false"

}

}

}

}

GET /hotel/\_search

{

"query": {

"match": {

"all": "外滩如家"

}

},

"highlight": {

"fields": {

"brand": {

"require\_field\_match": "false"

},

"name": {

"require\_field\_match": "false"

}

}

}

}

# 增加isAD字段 此方法会增加字段isAD

POST /hotel/\_update/2056126831

{

"doc": {

"isAD": true

}

}

POST /hotel/\_update/1989806195

{

"doc": {

"isAD": true

}

}

POST /hotel/\_update/2056105938

{

"doc": {

"isAD": true

}

}

# 桶聚合

GET /hotel/\_search

{

"size": 0,

"aggs": {

"brandAgg": {

"terms": {

"field": "brand",

"size": 20

}

},

"cityAgg": {

"terms": {

"field": "city",

"size": 20

}

},

"starAgg": {

"terms": {

"field": "starName",

"size": 20

}

}

}

}

# 桶聚合 自定义排序规则

GET /hotel/\_search

{

"size": 0,

"aggs": {

"brandAgg": {

"terms": {

"field": "brand",

"size": 20,

"order": {

"\_count": "asc"

}

}

}

}

}

# 桶聚合 限定聚合范围

GET /hotel/\_search

{

"query": {

"range": {

"price": {

"lte": 200

}

}

},

"size": 0,

"aggs": {

"brandAgg": {

"terms": {

"field": "brand",

"size": 20,

"order": {

"\_count": "asc"

}

}

}

}

}

# 度量聚合 (此处在brands聚合内部使用子聚合，就是分组后对每组分别计算)

GET /hotel/\_search

{

"size": 0,

"aggs": {

"brandAgg": {

"terms": {

"field": "brand",

"size": 20

},

"aggs": {

"score\_stats": {

"stats": {

"field": "score"

}

}

}

}

}

}

# 度量聚合 对度量聚合结果排序

GET /hotel/\_search

{

"size": 0,

"aggs": {

"brandAgg": {

"terms": {

"field": "brand",

"size": 20,

"order": {

"scoreAgg.avg": "desc"

}

},

"aggs": {

"scoreAgg": {

"stats": {

"field": "score"

}

}

}

}

}

}

# 测试py分词器

POST /test/\_analyze

{

"text": ["如家酒店还不错！"],

"analyzer": "my\_analyzer"

}

# 自定义拼音分词器

PUT /test

{

"settings": {

"analysis": {

"analyzer": {

"my\_analyzer": {

"tokenizer": "ik\_max\_word",

"filter": "py"

}

},

"filter": {

"py": {

"type": "pinyin",

"keep\_full\_pinyin": false,

"keep\_joined\_full\_pinyin": true,

"keep\_original": true,

"limit\_first\_letter\_length": 16,

"remove\_duplicated\_term": true,

"none\_chinese\_pinyin\_tokenize": false

}

}

}

},

"mappings": {

"properties": {

"name": {

"type": "text",

"analyzer": "my\_analyzer",

"search\_analyzer": "ik\_smart"

}

}

}

}

DELETE /test

POST /test/\_doc/1

{

"id": 1,

"name": "狮子"

}

POST /test/\_doc/2

{

"id": 2,

"name": "虱子"

}

GET /test/\_search

{

"query": {

"match": {

"name": "shizi"

}

}

}

# 自动补全的索引库

PUT /test2

{

"mappings": {

"properties": {

"title":{

"type": "completion"

}

}

}

}

GET /test2

# 示例数据

POST test2/\_doc

{

"title": ["Sony", "WH-1000XM3"]

}

POST test2/\_doc

{

"title": ["SK-II", "PITERA"]

}

POST test2/\_doc

{

"title": ["Nintendo", "switch"]

}

# 自动补全查询

GET /test2/\_search

{

"suggest": {

"titleSuggest": {

"text": "sw",

"completion": {

"field": "title",

"skip\_duplicates": true,

"size": 10

}

}

}

}